Trajectory Ensembles and Machine Learning:

From reinforcement learning for rare event sampling to training of neural network ensembles

by

Jamie F. Mair

A thesis submitted for fulfilment of the requirements for the degree of Doctor of Philosophy

in the

School of Physics & Astronomy
University of Nottingham

Supervisors:
Prof. Juan P. Garrahan
Dr. Adam Moss

June 2023

Abstract

This thesis builds on the existing body of work connecting the fields of statistical mechanics and machine learning. Many advances in machine learning have found their roots in statistical mechanics, e.g. simulated annealing for heuristic optimisation. Primarily, we aim to build bridges between trajectory ensemble techniques and current advances in machine learning, and in particular, deep learning. We explore these connections in two avenues: the first connects the study of rare events with that of reinforcement learning (RL); the second introduces a trajectory sampling algorithm for jointly training an ensemble of neural networks. We derive a framework for translating a rare event sampling problem into the language of RL, offering a way to leverage modern deep RL algorithms to obtain near-optimal sampling dynamics. Our work showcases a plethora of these RL algorithms and provides analysis on examples, including both finite and infinite time problems. Furthermore, we present a novel neural network ensemble training method, facilitated via Markov chain Monte Carlo algorithms, to produce coupled ensembles using gradient-free updates. We show that our coupled ensembles perform better, and are easier to train, than their uncoupled counterparts trained via neuroevolution, providing both analytic results on a linear problem and empirical evidence on the MNIST problem.

Contents

Acronyms **i**

Introduction **iii**

1 Introduction to Machine Learning **1**
1.1 Supervised Learning . . . 2
1.2 Linear Models . . . 3
1.3 Neural Networks . . . 4
1.3.1 Activation Functions . . . 5
1.3.2 Back-Propagation . . . 5
1.4 Training as an Optimisation Problem . . . 8
1.4.1 Gradient Descent . . . 8
1.4.2 Stochastic Gradient Descent . . . 9

2 Reinforcement Learning **11**
2.1 Mathematical Definition . . . 12
2.2 Grid World . . . 16
2.3 Dynamic Programming . . . 19
2.3.1 Bellman Equations and Optimality . . . 19
2.3.2 Policy Evaluation . . . 20
2.3.3 Policy Iteration . . . 22
2.4 Model-Free Methods . . . 24
2.4.1 Policy Evaluation through Interaction . . . 25
2.4.2 Monte Carlo Policy Iteration . . . 27
2.4.3 On/Off Policy Methods . . . 28
2.4.4 Bootstrapping: Updating Estimates with Estimates . . . 32
2.4.5 n-step Temporal Difference . . . 32
2.5 Approximate Solution Methods . . . 34
2.5.1 Large State Spaces and Generalisation . . . 35
2.5.2 The Deadly Triad . . . 35
2.6 Policy Gradient Methods . . . 36
2.6.1 Parameterising a Policy . . . 36
2.6.2 Policy Gradient: REINFORCE . . . 38
2.6.3 Variance of Vanilla Policy Gradient . . . 40
2.6.4 REINFORCE with Value Baseline . . . 45
2.6.5 Actor-Critic . . . 46

3 Large Deviation Theory **49**

3.1 Gärtner-Ellis Theorem . . . 50
3.2 Moments of the SCGF . . . 51
3.2.1 Example: Bernoulli Random Variables . . . 51
3.3 Dynamical Large Deviations . . . 53
3.3.1 Doob Dynamics . . . 56

4 Monte Carlo Sampling **57**
4.1 Monte Carlo Integration . . . 57
4.1.1 Error Estimates . . . 58
4.2 Importance Sampling . . . 59
4.3 Markov-Chain Monte Carlo Methods . . . 62
4.3.1 Metropolis-Hastings Algorithm . . . 63
4.3.2 Extensions to Trajectory Sampling . . . 64

5 Supporting Computational Infrastructure **67**
5.1 Course: *High Performance Computing in Julia* . . . 68
5.2 Package: `Experimenter.jl` . . . 69
5.2.1 Example Usage . . . 69
5.2.2 Limitations . . . 71
5.3 Package: `TransitionPathSampling.jl` . . . 72
5.3.1 Limitations . . . 72
5.3.2 Future Work . . . 73
5.4 Package: `SimpleNNs.jl` . . . 73
5.5 Computational Contributions to Other Published Works . . . 76

6 A Reinforcement Learning Approach to Rare Trajectory Sampling **77**

7 Training Neural Network Ensembles via Trajectory Sampling **117**

8 Minibatch Training of Neural Network Ensembles via Trajectory Sampling **131**

Conclusions & Outlook **143**

Bibliography **147**

Acronyms

AI artificial intelligence

API application programming interface

DL deep learning

DP dynamic programming

DRL deep reinforcement learning

GC garbage collector

GE Gärtner-Ellis

HMC Hamiltonian Monte Carlo

HPC high performance cluster

IID independent and identically distributed

LDP large deviation principle

LDT large deviation theory

MC Monte Carlo

MCMC Markov chain Monte Carlo

MDP Markov decision process

ME master equation

MGF moment generating function

ML machine learning

MSE mean-squared error

NN neural network

NNE neural network ensemble

ReLU rectified linear unit

RL reinforcement learning

SCGF scaled cumulant generating function

SGD stochastic gradient descent

SL supervised learning

TD temporal difference

TPS transition path sampling

USL unsupervised learning

Introduction

Machine learning is a growing field that is being applied to several domains, such as protein folding [1], natural language processing [2–6], healthcare [7–10], climate modelling [11], robotics [12–14], and even discovery of physics from data [15, 16]. The history of machine learning has deep-rooted connections with statistical mechanics [17–22], and the research continues (e.g. [23–25]), but there is much to explore. This thesis aims to add to the existing body of work, emphasising the connections between the two fields.

In particular, we first aim to connect the study of dynamical fluctuations with large deviation theory [26] to that of reinforcement learning (RL) [27], focusing on the study of rare events. Our research allows powerful RL algorithms to be used to train an optimal sampling dynamics. We do this by formulating a reward structure, whereby the optimal policy is equivalent to the optimal sampling dynamics, as measured by the Kullback-Leibler divergence. We present various standard techniques for solving RL problems, such as policy gradient methods and dynamic programming. Further, we demonstrate these techniques and their connection to rare event sampling on examples ranging from a discrete finite time excursion problem, to an infinite time horizon problem based on a particle hopping on a 1-dimensional ring. We cover the fundamental basis of this research in chapters 1, 2, and 3 and present the published research in Chapter 6.

Following from this, we aim to adapt methods commonly used for studying dynamical systems, specifically trajectory sampling techniques, into an ensemble training method, whereby a collection (or *ensemble*) of machine learning models are jointly optimised. We translate the problem of "training" into one of sampling a low loss trajectory, where the trajectory represents the ensemble and the loss acts as a surrogate for the optimisation target. In Chapter 7, we show that this technique is viable for training reasonably large neural network ensembles (NNEs), whose individual models are based on the convolutional neural network architecture, to obtain high accuracy on the MNIST problem [28]. In the same chapter, we additionally conduct a theoretical analysis of the technique, verified with results on a simple linear problem. Introductory material for the basis of this research is outlined in the chapters 1, 3, and 4.

We continue our investigation of our ensemble training technique in Chapter 8, where we present an algorithmic adjustment that vastly improves the computational efficiency of the method. In its original form, the training technique is computationally bound by the size of the dataset provided, now remedied by the introduction of a statistically compatible minibatch technique. Using the minibatch approach, we were able to decouple training time from the size of the dataset, vastly expanding the range of problems in which this technique is viable.

Much of the research presented in this thesis is underpinned by a substantial engineering

effort required to produce the results. The output of this effort is reviewed and summarised in Chapter 5, which primarily outlines several packages that can support future research. Second to this, the chapter also describes a large body of educational resources produced to aid future researchers in developing skills for writing high performance code, particularly that which can be reused and shared with others.

This thesis is organised in a *thesis by publication* style, with the first 4 chapters providing background information required for understanding and interpreting the research presented in chapters 6, 7 and 8.

Chapter 1

Introduction to Machine Learning

All the research projects undertaken and presented in this thesis are underpinned by the field of machine learning (ML). This field is the study of how to produce algorithms and techniques for producing programs which learn desired behaviour from data. Traditional algorithms, while enabling great progress in the field of computing, are often limited to well-defined tasks with well-defined inputs. Many of the large computing systems today have a vast amount of their infrastructure dedicated to ensuring these algorithms are fed with the right data which enables them to function. These algorithms are carefully hand-crafted by humans to produce reliable, predictable and verifiable results, while ML instead aims to produce models and methods which *learn* from existing data. This is of particular interest as it enables ML practitioners to solve, or attempt to solve, problems that currently have no known solution. For an example of where ML techniques have vast advantages over hand-crafted algorithms, one need look no further than *computer vision*. Early algorithms for recognising handwritten digits used hand-crafted features and simple techniques, which often had very poor performance. Adoption of ML techniques, in particular those of deep learning (DL) [29], now allow computer vision algorithms to compete with [30, 31] (or even outperform [32–35]) existing methods, or sometimes humans, on many vision-based tasks of classification and recognition.

At the time of writing, ML (and more generally its parent field of artificial intelligence (AI)) is making new headlines every day. These algorithms and methods are set to revolutionise modern society; understanding and building on the existing techniques is of vital importance.

We can break down ML into its three subfields:

- **Supervised learning (SL)** — Learning a mapping of features to labels, from a dataset consisting of features and labels (already supplied).
- **Unsupervised learning (USL)** — Learning underlying patterns in some unlabelled data, to find structure or meaning.
- **Reinforcement learning (RL)** — Learning to achieve a certain goal through actions, usually acting over time, defined in terms of maximising the cumulative scalar signal known as the reward.

In this thesis, the most pertinent topics are that of SL and RL. In this chapter, we will briefly cover some of the basics of SL and introduce neural networks (NNs), which are

at the heart of the research presented in chapters 6, 7 and 8. The thesis will not cover the details of USL — interested readers are directed towards [36]. We will dedicate the entirety of Chapter 2 to RL.

1.1 Supervised Learning

SL is one of the three subfields of ML, and one of the most extensively studied. This subfield focuses on learning (or "fitting") models to existing data for some process. In the context of supervised learning, we refer to a model as a (usually) parameterised function approximation. For example, if one has a set of **features**, commonly denoted as $\boldsymbol{X}$, with corresponding **labels**, or **targets**, $\boldsymbol{Y}$, one may want to find a model which best represents the mapping from features to labels. We refer to the combined features and labels as the **dataset**. We denote a model, parameterised by some variables θ, as f_θ, which maps the features to values we call **predictions**

$$\tilde{\boldsymbol{Y}} = f_\theta(\boldsymbol{X}). \tag{1.1}$$

In training, we aim to make these predictions similar to the corresponding labels.

When provided labels are continuous, one popular measure of fit is the mean-squared error (MSE), which defines a **loss function** that should be minimised. We write the loss as

$$L_{\text{MSE}}(\theta; \boldsymbol{X}, \boldsymbol{Y}) = \frac{1}{2} \sum_i^N [\boldsymbol{Y}_i - \tilde{\boldsymbol{Y}}_i]^2, \tag{1.2}$$

where N is the number of samples the dataset. On the right-hand side of the eq. (1.2), we note that $\tilde{\boldsymbol{Y}}$ implicitly relies on the model f and the parameters θ. For now, we assume that a prediction from a single sample is 1-dimensional, but any derivations can be extended to predict multidimensional outputs [36]. Additionally, one may also see the loss function written simply as $L_{\text{MSE}}(\theta)$, as this will implicitly rely on the dataset used.

Functionally, when this loss equals zero, the model and its parameters can precisely recreate the labels from the features in the dataset. Some models may not have the capacity to precisely model the supplied data, and instead aim for minimising the loss. Often, the goal of supervised learning is not just to create this encoding of features to labels exactly, but to be able to learn the underlying relationship so that one can obtain high quality predictions on *new* data that was not in the training dataset, a property known as **generalisation**. If the supplied training set contained the entire set of possible inputs to the model, then minimising the loss function (i.e. perfectly mapping the features to the labels) would also generalise, as the set of *unseen* data is empty. However, this is rarely achievable and if it is, one would almost certainly not require these techniques.

A concrete definition of generalisation can be given as [37]

> A model which performs at least as well on unseen examples from the possible set of inputs as it does on the supplied training set can be said to **generalise**. If performance on unseen data is much worse than on a training set, the model can be said to be **overfitting**.

While a sufficiently complex model can indeed fit any supplied data, it often behoves one to instead employ **Occam's razor** — "the simplest model that explains the data is

usually the best one". For this reason, we aim to find the least complex models which perform "*well*" on the training dataset, while still being able to generalise on unseen data.

The two most common categories of problems in SL are regression and classification, where the first deals with producing a numerical value for a problem, and the second deals with predicting the group that a sample belongs to. Both of these problems are discussed in the research presented in Chapter 7 and Chapter 8.

1.2 Linear Models

A common function approximation studied and employed for fitting data is the linear model. The linear model is well studied and understood, producing understandable and reliable output [36]. A linear model can be characterised by multiplying some input features by some parameters (the weights) and summing the result and adding a constant factor — the bias. If a single sample has a vector of N features, $x_i \in \boldsymbol{x}$, then it can be mapped to an output via

$$y = b + \sum_{i}^{N} w_i x_i = b + \boldsymbol{w} \cdot \boldsymbol{x}, \tag{1.3}$$

where b is a constant bias parameter and $\boldsymbol{w}$ is an N-dimensional vector of the weight parameters. For simplicity, we usually extend the input features with a constant such that $\tilde{\boldsymbol{x}} = [\boldsymbol{x}\ 1]$, allowing us to incorporate the bias b inside a modified weight vector $\tilde{\boldsymbol{w}} = [\boldsymbol{w}\ b]$ such that $y = \tilde{\boldsymbol{w}} \cdot \tilde{\boldsymbol{x}}$ for simplicity.

This type of model, while being very simple in nature, is suitable for some simple tasks, especially when augmented with feature selection or representation learning [38]. We can extend the model to make multiple predictions by transforming our $\tilde{\boldsymbol{x}} \in \mathbb{R}^{(d+1)\times 1}$ into a matrix $\tilde{\boldsymbol{X}} \in \mathbb{R}^{(d+1)\times(N)}$, where d is the dimensionality of the data and N is the number of samples in a dataset. These N feature vectors will have N corresponding output labels, now denoted as $\boldsymbol{Y} \in \mathbb{R}^{1\times N}$. Typically, if we want to train the model to predict the output, we use a MSE loss (see eq. (1.2)). This loss can be minimised using optimal weights $\tilde{\boldsymbol{w}}^*$ calculated via

$$\tilde{\boldsymbol{w}}^* = (\tilde{\boldsymbol{X}}\tilde{\boldsymbol{X}}^T)^{-1}\tilde{\boldsymbol{X}}\tilde{\boldsymbol{Y}}^T. \tag{1.4}$$

It is common to create one's own features for input into a linear model, such as squaring the inputs — this is the aforementioned feature selection. This process can be used to fit a non-linear model (such as a polynomial) using the same mechanisms as the linear model. Additionally, one can regularise the model by introducing a hyperparameter, usually denoted with λ, to penalise the weights for being too large, as this can cause overfitting. One can optionally introduce an additional $\frac{\lambda}{2}\tilde{\boldsymbol{w}}^T\tilde{\boldsymbol{w}}$ term to the loss given in eq. (1.2), which can then be exactly optimised via the optimal parameters

$$\tilde{\boldsymbol{w}}^* = (\lambda\boldsymbol{I} + \tilde{\boldsymbol{X}}\tilde{\boldsymbol{X}}^T)^{-1}\tilde{\boldsymbol{X}}\tilde{\boldsymbol{Y}}^T, \tag{1.5}$$

where $\boldsymbol{I}$ is the identity matrix.

An example of a learned linear model is shown in fig. 1.1.

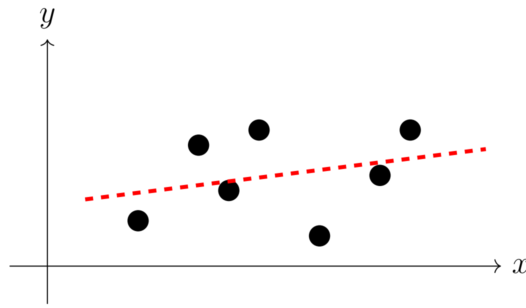

Figure 1.1: An example dataset of several (x, y) points, with a liner model, the red dashed line, fit to the data by minimising the MSE loss, with no regularisation. This method is also known as the *least-squares* method [39].

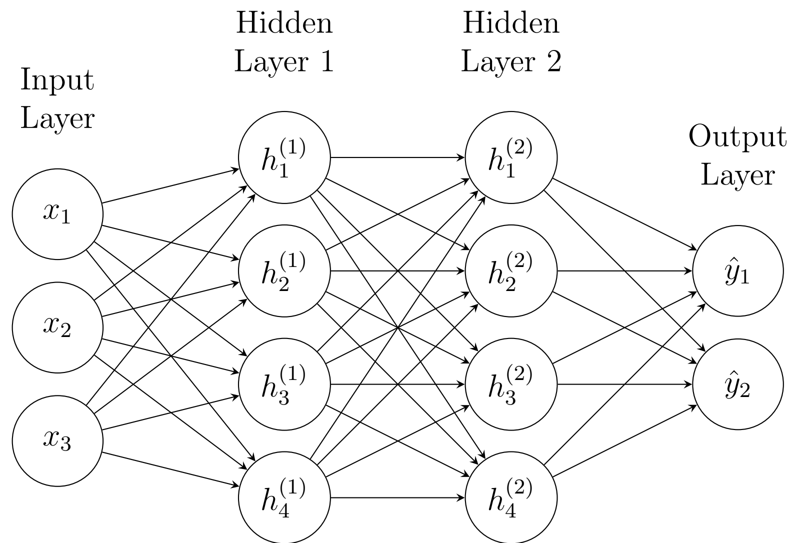

Figure 1.2: Traditional feed-forward neural network with 2 hidden layers.

1.3 Neural Networks

Neural networks (NNs) form the basis of the current field of DL. A NN processes inputs sequentially in "layers", reminiscent of linear models. Each of these layers has a specified number of outputs, known as neurons. In a *fully-connected* NN, each neuron is connected to every other neuron in the layers adjacent to its current layer, but not connected to the neurons in the same layer. Additionally, each neuron has an associated **activation function**, which is some non-linear function, chosen separately for each layer. Each neuron acts as its own linear model, taking as inputs all the neurons of the previous layer, and additionally passing its output through the non-linear activation function. This description is known as a *feed-forward neural network*, and a corresponding diagram is shown in fig. 1.2. This type of network will be the one used throughout this text, unless otherwise stated.

To simplify the equations, we will specify the output of a network in terms of matrices, and incorporate the bias into the weight matrix as in the previous section by using the $\tilde{\cdot}$ notation, such that

$$\tilde{\boldsymbol{A}} = \begin{bmatrix} \boldsymbol{A} \\ \boldsymbol{1} \end{bmatrix}, \quad \text{and} \quad \tilde{\boldsymbol{W}}_n = \begin{bmatrix} \boldsymbol{W} \\ \boldsymbol{b}_n. \end{bmatrix} \tag{1.6}$$

The nth hidden layer will have the output

$$\boldsymbol{H}^{(n)} = f_n(\tilde{\boldsymbol{W}}^{(n)}\tilde{\boldsymbol{H}}^{(n-1)}), \tag{1.7}$$

where $f_n(\cdot)$ represents the **activation function** of the nth layer, and we define $\boldsymbol{H}^{(0)} = \boldsymbol{X}$ and $\boldsymbol{H}^{(K+1)} = \hat{\boldsymbol{Y}}$ where K represents the number of hidden layers in the neural network.

1.3.1 Activation Functions

The activation function of each neural network layer must be *non-linear*, as any layer following a linear activation function will have its weight matrix combined with the current layer, resulting in redundancy of the parameters of the first layer. One of the most common activation functions is the rectified linear unit (ReLU), which can be expressed as

$$\text{ReLU}(x) = \max(0, x). \tag{1.8}$$

Another common activation function is the *logistic sigmoid* which can be written as

$$\sigma(x) = (1 + e^{-x})^{-1}, \tag{1.9}$$

which constrains the output as $0 < \sigma(x) < 1 \ \forall \ x \in \mathbb{R}$. This is useful is the output can be interpreted as a probability.

A final example is the *hyperbolic tangent*, which is expressed as

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}. \tag{1.10}$$

These are not all the possible activation functions, but are the most common. These three functions are graphed visually in fig. 1.3.

1.3.2 Back-Propagation

Most recent methods of training a neural network rely on gradient descent applied to the loss function in parameter space [36, 40, 41]. This means we need to take the gradient of the loss w.r.t the parameters of the model. There are numerous numerical techniques and algorithms for computing the gradient of an arbitrary function, however, the most efficient method is often **back-propagation**. As the loss is usually calculated using a forward pass (i.e. running the model forwards to obtain the predictions), we can cache the results of the outputs of the hidden layers and use them on a "backwards pass" through the model to calculate the gradients using the chain rule.

Back-propagation is an algorithm entirely derived by repeated application of the chain rule. We can calculate the derivative of the loss w.r.t the output neurons of a given layer, and then use this information to propagate these gradients to the next layer. As a simple example, let us briefly consider a simple loss function for a single data point with 1 output y.

$$L = \frac{1}{2}(y - \hat{y})^2. \tag{1.11}$$

The output of the network, $\hat{y}$, is a function that depends on the weights of the network. Our aim is to be able to calculate the derivatives of L w.r.t each of the weights of the

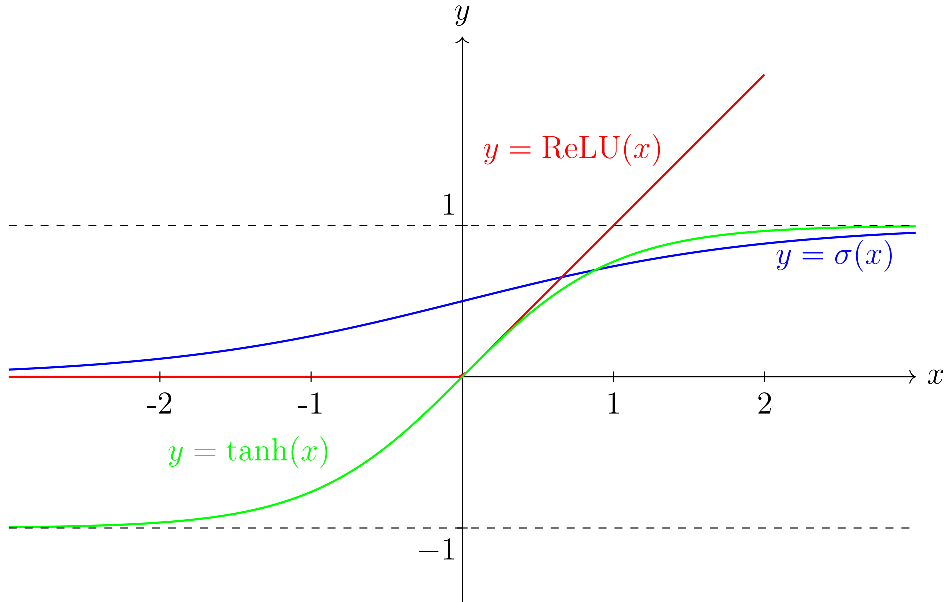

Figure 1.3: Shows three activation functions: *ReLU — Rectified Linear Unit* (in red), given by eq. (1.8); *logistic sigmoid* (in blue), given by eq. (1.9); and the *hyperbolic tangent* (in green), given by eq. (1.10).

network. Let us consider that the neural network has K layers, such that we express $\hat{y}$ as

$$\hat{y} = \sum_j W_{1,j}^{(K)} h_j^{(K-1)} + b_1^{(K)}, \tag{1.12}$$

where $\boldsymbol{W}^{(k)}$ and $\boldsymbol{b}^{(k)}$ represent the weight matrix and bias vector of the kth layer respectively. We know from calculus that the derivative of L with respect to the weights can be calculated via the chain rule

$$\frac{\partial L}{\partial W_{1,j}^{(K)}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W_{1,j}^{(K)}}. \tag{1.13}$$

Similarly, we know that

$$\frac{\partial L}{\partial b_1^{(K)}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b_1^{(K)}}. \tag{1.14}$$

Finally, we know that we can work out the partials $\frac{\partial L}{\partial h_j^{(K-1)}}$ again using $\frac{\partial L}{\partial \hat{y}}$. Using these chain rule equations, we can break up the calculation of the partial derivatives of the weights and biases to depend on the outputs of their layer. The partials of the outputs of the previous layer can also be calculated by application of the chain, allowing one to repeat the process back through the entire network.

In the remaining part of this section, we will comprehensively derive the equations for back-propagation for a simple neural network composed of fully connected layers. As back-propagation is usually defined by averaging gradients for a batch of N data points, it makes sense to perform this derivation using matrix calculus where the input features are defined as $\boldsymbol{X} \in \mathbb{R}^{d_{\text{in}} \times N}$ and target outputs $\boldsymbol{Y} \in \mathbb{R}^{d_{\text{out}} \times N}$. We allow for the outputs to have multiple dimensions per sample, which we evenly weight when contributing to the loss. Following the MSE loss given in eq. (1.2), we extend the definition to our matrix

notation as

$$L_{\text{MSE}}(\theta; \boldsymbol{X}, \boldsymbol{Y}) = \frac{1}{2N} \operatorname{tr}\left\{[\boldsymbol{Y} - f(\boldsymbol{X})][\boldsymbol{Y} - f(\boldsymbol{X})]^T\right\}$$
$$= \frac{1}{2N}\left[\operatorname{tr}\{\boldsymbol{Y}\boldsymbol{Y}^T\} + \operatorname{tr}\{f(\boldsymbol{X})f(\boldsymbol{X})^T\} - 2\operatorname{tr}\{f(\boldsymbol{X})\boldsymbol{Y}^T\}\right], \tag{1.15}$$

where $\operatorname{tr} A$ denotes the trace of the matrix A. We call the outputs of the network $\hat{\boldsymbol{Y}} = f(\boldsymbol{X})$.

We start by finding the derivative of the loss w.r.t to the neural network outputs to be given by

$$\nabla_{\hat{\boldsymbol{Y}}} L_{\text{MSE}} = \nabla_{\hat{\boldsymbol{Y}}} \frac{1}{2N}\left[\operatorname{tr}\{\boldsymbol{Y}\boldsymbol{Y}^T\} + \operatorname{tr}\{f(\boldsymbol{X})f(\boldsymbol{X})^T\} - 2\operatorname{tr}\{f(\boldsymbol{X})\boldsymbol{Y}^T\}\right]$$
$$= \frac{1}{2N}\left[\cancel{\nabla_{\hat{\boldsymbol{Y}}} \operatorname{tr}\{\boldsymbol{Y}\boldsymbol{Y}^T\}} + \nabla_{\hat{\boldsymbol{Y}}} \operatorname{tr}\{\hat{\boldsymbol{Y}}\hat{\boldsymbol{Y}}^T\} - 2\nabla_{\hat{\boldsymbol{Y}}} \operatorname{tr}\{\hat{\boldsymbol{Y}}\boldsymbol{Y}^T\}\right]$$
$$= \frac{1}{\cancel{2}N}\left[\cancel{2}\hat{\boldsymbol{Y}} - \cancel{2}\boldsymbol{Y}\right]$$
$$= \frac{1}{N}\left[\hat{\boldsymbol{Y}} - \boldsymbol{Y}\right]. \tag{1.16}$$

Using this derivative as the initial input, we aim to recursively apply the chain rule through each layer, caching our partial gradients along the way for use in the later equations. Recall that the NN is made up of layers defined by

$$\boldsymbol{H}^{(n)} = f_n(\tilde{\boldsymbol{W}}^{(n)}\tilde{\boldsymbol{H}}^{(n-1)})$$
$$= f_n(\boldsymbol{W}^{(n)}\boldsymbol{H}^{(n-1)} + \boldsymbol{b}^{(n)})$$
$$= f_n(\boldsymbol{A}^{(n)}), \tag{1.17}$$

where $\boldsymbol{A}^{(n)}$ is called the **activation** and $\boldsymbol{b}^{(n)}$ represents the bias vector of the nth layer. To start, we can write that the derivative of the loss w.r.t the activation is given by

$$\nabla_{\boldsymbol{A}^{(n)}} L_{\text{MSE}} = (\nabla_{\boldsymbol{H}^{(n)}} L_{\text{MSE}}) \odot {f_n}'(\boldsymbol{A}^{(n)}), \tag{1.18}$$

where $\odot$ represents the *Hadamard product* (or "*element-wise*" product) between two matrices. From here, we again apply the chain rule to calculate the gradients w.r.t the weights in the nth layer, such that

$$\nabla_{\boldsymbol{W}^{(n)}} L_{\text{MSE}} = \nabla_{\boldsymbol{A}^{(n)}} L_{\text{MSE}} \times {\boldsymbol{H}^{(n-1)}}^T. \tag{1.19}$$

Finally, we can calculate the gradients w.r.t the biases of the layer, given by

$$\nabla_{\boldsymbol{b}^{(n)}} L_{\text{MSE}} = \nabla_{\boldsymbol{A}^{(n)}} L_{\text{MSE}} \times \mathbf{1}, \tag{1.20}$$

where $\mathbf{1}$ represents a "ones" column vector of size $(N \times 1)$.

Altogether, we know that $\boldsymbol{H}^{K+1} = \hat{\boldsymbol{Y}}$, for which we have already calculated gradients. Using eq. (1.18), we are able to calculate the gradient term needed for both eq. (1.19) and eq. (1.20). We can calculate the gradients of the outputs of the next layer via

$$\nabla_{\boldsymbol{H}^{(n-1)}} L_{\text{MSE}} = \left[(\nabla_{\boldsymbol{A}^{(n)}} L_{\text{MSE}})\boldsymbol{W}^{(n)}\right]^T. \tag{1.21}$$

The process ends when the gradients of the first layer ($n = 1$) are calculated. One is usually not concerned with calculating the gradients w.r.t the input vector $\boldsymbol{X}$, but remember that $\boldsymbol{X} = \boldsymbol{H}^{(0)}$ and one can use eq. (1.21) to calculate these gradients if so desired.

To highlight the importance of the forward pass, notice that eq. (1.19) depends on the layer outputs of the previous layer and eq. (1.18) relies on the activation of that layer. An efficient implementation can cache these values during the forward pass so that they can be used in back-propagation, without needing to recalculate them (at the cost of higher memory usage). Additionally, one tends to use activation functions whose derivatives calculated w.r.t the inputs can be evaluated using the outputs. For example, consider the sigmoid activation

$$\sigma(x) = (1 + e^{-x})^{-1}, \tag{1.22}$$

whose derivative is

$$\sigma'(x) = \frac{e^{-x}}{(1 + e^{-x})^2}. \tag{1.23}$$

Notice that we can equivalently write $\sigma'(x)$ in terms of the output $\sigma(x)$,

$$\sigma'(x) = \sigma(x)\left[1 - \sigma(x)\right]. \tag{1.24}$$

This allows one to just store the output of the activation in memory on the forward pass, without needing to store the activation itself.

I have implemented this algorithm, applied to neural networks, available in the package `SimpleNNs.jl`, described in Section 5.4.

1.4 Training as an Optimisation Problem

Training a model (such as a NN) to perform well on a desired task can be difficult depending on the task itself. Machine learning provides a suite of common problem classes, such as regression or classification, which can act as a standard proxy for a large range of tasks and provide a common strategy for training models. Particularly in DL, one uses a loss function as a proxy for "*good*" performance on a task. One usually selects this proxy to be continuous, as to allow for gradient-based methods to optimise the loss via gradient descent. In this section, we will briefly cover the basics of this method. Combined with the back-propagation algorithm defined in the previous section, this provides the underpinnings of traditional training methods.

1.4.1 Gradient Descent

A vanilla version of gradient descent is comparable to the Newton-Raphson method [42] of root-finding, whereby one calculates the gradient of a given set of parameters on a loss function and proceeds to follow the negative of that gradient in fixed steps according to some step size α. This algorithm is an iterative method, with the update step given by

$$\theta_{n+1} = \theta_n - \alpha_n \nabla_{\theta_n} L(\theta_n), \tag{1.25}$$

where n is the current iteration index, θ are the parameters to optimise on the loss function L, and α_n (chosen such that $\alpha_n > 0$) is a *learning rate* which can depend on

the iteration number n. A similar algorithm is gradient *ascent*, where one substitutes $\alpha_n \to -\alpha_n$ into the above equation.

This algorithm has many issues in that it cannot effectively optimise a non-convex loss function as it may get trapped in *local minima* and not reach the *global minima* [36]. Additionally, each update of the parameter set is expensive as it requires the full evaluation and gradient calculation on all samples in the dataset. The vanilla version of gradient descent, as presented here, is rarely used in practice as a slight modification is favoured as the base of many gradient-based methods - stochastic gradient descent (SGD).

1.4.2 Stochastic Gradient Descent

SGD modifies the vanilla gradient descent algorithm by evaluating the loss function on a subset of the total available data which is chosen at random on each step. The choosing of data introduces stochasticity into the iterative optimisation process which allows for overcoming local minima [36]. One usually defines the loss on a subset of data $\mathcal{D}^*$ which is randomly chosen at each n such that the update now becomes

$$\theta_{n+1} = \theta_n - \alpha_n \nabla_{\theta_n} L^*(\theta_n), \tag{1.26}$$

where

$$L^*(\theta_n) = \frac{1}{|\mathcal{D}^*|} \sum_{X_i \in \mathcal{D}^*} L(\theta_n; X), \tag{1.27}$$

where $|\mathcal{D}^*|$ is the number of samples in the minibatch and the loss, $L(\theta_n; X_i)$, is evaluated on a single data point, X_i. For batch sizes of 1, this has similarities with *online learning* [43]. Typically, one uses larger batch sizes to parallelise the training process [44].

This technique is very powerful, and can reach global optimisation in the infinite time limit with probability 1, dependent on some constraints on the learning rate [45, 46], written as

$$\sum_{n=0}^{\infty} \alpha_n = \infty, \quad \sum_{n=0}^{\infty} |\alpha_n|^2 = c, \tag{1.28}$$

where c is just a finite constant. However, waiting for an infinite time is quite impractical, and instead, many researchers favour some momentum-based approaches that build on SGD — such as ADAM [41]. ADAM, derived from "*adaptive moment estimation*", is a momentum-based gradient optimisation method, and is arguably the most popular method for optimisation due to its empirically validated performance across a wide range of problems.

Chapter 2

Reinforcement Learning

RL is one of three core paradigms in ML. It aims to provide a framework on how to "teach" an agent (usually a ML model) to complete a specific goal in the most optimal way, even when we as teachers do not know the most optimal actions. This departs from the usual trend of SL, where one has access to a (usually static) dataset which is labelled, and only need "fit a curve" to the data. This form of learning often requires interaction with the environment to explore and test actions to assess their quality. For this reason, RL is often studied in the context of games like Chess and Go, where agents can be trained in a simulation, which eventually can outperform all human experts [47, 48]. As this technique has far ranging applicability, it is often studied in a range of fields, from psychology to operations research. This technique is incredibly powerful and general, having deep connections with fields such as control theory, operations research, and game theory. In the statistical mechanics literature, problems that study *optimal control* [49–53] can be linked to RL. In other fields, the term *approximate dynamic programming* [54, 55] usually refers to techniques that are also studied under RL.

During this chapter, we will take a deep dive into the field of reinforcement learning to provide context and background for the research presented in Chapter 6, where we explore the connections between RL and rare trajectory sampling. For readers wishing for a more complete review of the topic, see Sutton and Barto's excellent textbook [27].

At its core, SL can only hope to solve tasks that are already able to be solved by humans in some form, whether that be personally (e.g. image classification tasks [28, 56]), or via expensive simulations or algorithms [57–61]. SL relies on being able to give a learning agent information about correct or incorrect actions. One cannot use SL techniques to train a model to perform a task if no one knows how to label any data for that task. Similarly, USL can only uncover underlying trends in data. USL does not aim to have a model perform a certain behaviour, but simply, learn underlying patterns in supplied data. However, RL aims to provide a framework for learning optimal, complex, behaviour in scenarios in which we, as the teacher, are not able to provide optimal examples. RL is often deployed in scenarios where a model has to perform actions over multiple time steps, in order to achieve a certain goal. One can avoid giving examples of correct behaviour by instead constructing a scalar signal (known as the reward), which can inform the agent whether the actions taken were "*good*" or "*bad*". This process of translating a complex task into reward signals can often vastly simplify the task. For example, knowing what is a good move to make in Chess is extremely difficult, but it is obviously bad if one loses

and good if one wins. If one assigns a number to both winning and losing, say +1 and −1 respectively, then one can translate the task of "playing Chess well" into the problem of choosing an action in each situation that will maximise the chances of receiving a +1 reward.

The process of rewarding *good* behaviour and punishing *bad* behaviour explains the *Reinforcement* part of RL, having some origins in the field of Psychology [27, 62]. One tries to reinforce *good* actions and discourage *bad* actions. In RL, we attempt to perform the same procedure, under the **reward hypothesis** [27, 63, 64]:

> Any goal can be formalised as the outcome of maximising a cumulative, scalar, reward signal.

This form of learning is done through *interaction with our environment*. This is often a very active form of learning, as one can choose what information to gather through one's actions. Most importantly, using a reward signal for a goal definition can enable us to find optimal behaviour without any examples of optimal behaviour.

RL is a term given to three distinct concepts [27]:

1. A framework for mathematically defining a goal-oriented task as a RL problem.
2. The research field which studies the aforementioned problem.
3. The suite of algorithms that are developed to solve a RL problem.

Before we can dive into any more details, we must rigorously define what a RL problem consists of. This is the topic of the next section.

2.1 Mathematical Definition

We have already outlined some of the ways one can think of defining a RL problem. One usually refers to the diagram shown in fig. 2.1. A RL episode usually involves multiple time steps, hence the cycle in the diagram. We usually denote the current time step as t.

The process starts with the agent observing the state of the environment. This can be denoted as $O_t = O(S_t)$, where $O(s)$ denotes the observation function which maps the state of the environment to the observation that the agent sees — this is depicted as the *observer* in fig. 2.1. For simplicity, we often assume that $O_t = S_t$, which is a *fully-observable environment*. When the agent receives the first observation, an internal policy function maps this observation to an action. In a deterministic case, this is usually written as $A_t = \pi(S_t)$. However, sometimes the agent follows a stochastic policy, which defines a probability distribution over all actions possible in the state s, given by $\pi(a|s)$ where we normalise the distribution such that

$$\sum_{a \in \mathcal{A}(s)} \pi(a|s) = 1, \tag{2.1}$$

in which $\mathcal{A}(s)$ represents the set of actions that an agent can take in the state s.

Once the agent selects and carries out an action, it is applied to the environment. This, together with the previous state, is used to update the environment to the next time

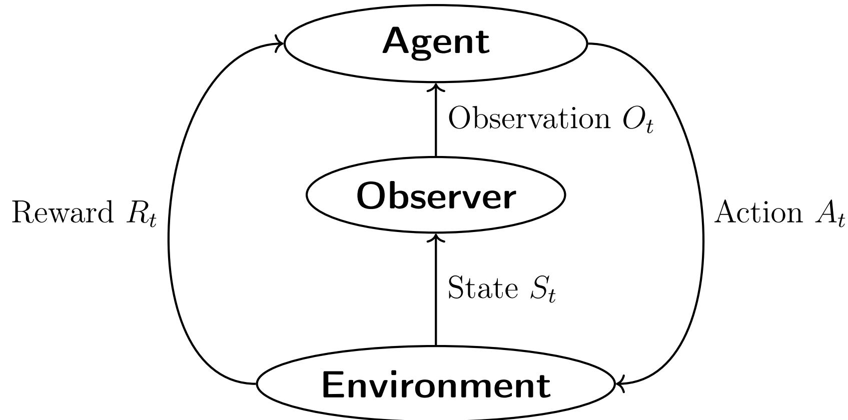

Figure 2.1: The typical setup of a RL problem. t represents the discrete time of the events. One reads the diagram clockwise, starting with the environment producing an initial state, S_0, which is converted to an observation and given to the agent. The agent decides which action to take, which is then sent to the environment. This alters the environment and produces a reward, which is sent to the agent, and a new state is observed and the cycle continues. Note that the agent and environment are depicted as separate for visual clarity, but usually the agent is contained within the environment itself.

step. In a deterministic setting, one usually writes this as $s' = f(s, a)$, where s' is the next state and $f(s, a)$ is the **environment function** which maps states to actions.

Again, we can extend the framework to have a stochastic environment, whose function instead defines a probability distribution over all possible next states and rewards $p(s', r|s, a)$, where

$$\sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s, a) = 1, \tag{2.2}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. We use s' to often denote a generic "next" state, s for the current generic state and r and a to be a reward and action respectively. We use the convention that $\mathcal{S}$, $\mathcal{R}$ and $\mathcal{A}(s)$ represent the state, reward, and action spaces respectively. Once a single transition has been made, a reward for the action taken can then be sampled using the reward function. In the deterministic reward setting, one can write this reward function as $r(s', s, a)$, read as the scalar reward for transitioning from state s to state s' using action a. In the stochastic setting, the probability distribution is usually combined with that of the next state, whose shorthand expression is given by

$$p(s', r|s, a) = f(s'|s, a)\phi(r|s', s, a), \tag{2.3}$$

where we have used $\phi(r|s', s, a)$ to define the probability distribution of receiving the reward r conditioned on transitioning from state s to s' using action a. This distribution is normalised, as given by

$$\int_{-\infty}^{+\infty} \phi(r|s', s, a)dr = 1. \tag{2.4}$$

This reward function defines the goal of the problem, and is usually constructed to influence the learning outcome (in a process known as *reward engineering* [65]). Alternatively, this reward function can also be learnt via *inverse reinforcement learning* [66] from observed optimal behaviour. Like with the other parts of the process, this reward can also

be stochastic in nature. The framework allows each individual process to have some stochasticity, however, pedagogically it is easiest to look at the deterministic case first.

We can now define a quantity, known as the **return**, which is the cumulative sum of rewards into the future, denoted as

$$G_t = R_t + R_{t+1} + R_{t+2} + \ldots + R_{T-1}, \tag{2.5}$$

where T indicates the time of reaching the **terminal** state. A terminal state defines an end to a trajectory, for which $R_{t'} = 0$ when $t' \geq T$. If a problem has a terminal state, we call it an **episodic problem**. One can alternatively specify non-episodic problems, which do not have any terminal state, but continue indefinitely. However, looking at eq. (2.5), one can see the return will blow up for very long or non-episodic ($T \to \infty$) problems. One way to adapt this return, is to introduce a parameter γ, which controls how much an agent should pay attention to short term gains compared to long term gains. We introduce this parameter to the return as

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots = \sum_{i=0}^{\infty} \gamma^i R_{t+i}, \tag{2.6}$$

which is called the **discounted return**, as the parameter γ discounts future rewards in favour of short term rewards. This parameter is subject to $\gamma \in [0, 1]$. One recovers the episodic case by setting $\gamma = 1$ and defining all rewards from (and including) time step T equal to 0. We will use the term *return* to, in general, refer to the discounted return. Sometimes γ is omitted in the case that $\gamma = 1$.

One must remember that eq. (2.5) and eq. (2.6) are defined in terms of the *sampled* returns. They are simply random variables, which are sampled from having an agent interact with the environment. We use the notation of a capital letter X to denote a random variable sample of the quantity usually denoted by the symbol x. For example, you will see R_t for the reward sampled at time t and S_t as state sampled at time t, whereas r and s represent a generic reward and state respectively.

As we regularly refer to a sequence of events in time, we use ω to denote a **trajectory**, which is made up of an array of state-action-reward tuples, written as

$$\omega = [S_0, A_0, R_0, S_1, A_1, R_1, \ldots, S_{T-1}, A_{T-1}, R_{T-1}, S_T], \tag{2.7}$$

here S_T is the terminal state. We use the notation $T(\omega)$ to indicate the terminal time step of the trajectory ω.

A useful construct is the **value function**, which calculates the expected return an agent will receive, given they are in the state s. This is usually written as

$$v_\pi(s) = \mathop{\mathbb{E}}_{\omega \sim \pi | S_0 = s} [G(\omega)], \tag{2.8}$$

where $\mathop{\mathbb{E}}_{\omega \sim \pi | S_0 = s} [A]$ denotes the average (or expected) value of the random variable A, which is sampled according to using the policy π, given that the initial state started at s. To summarise eq. (2.8), we average the total discounted return of all possible trajectories which start at state s, and follow the environment transition probabilities along with

the policy π. Another way of writing an expectation is as an average over all possible trajectories, weighted by the probability of a trajectory $p_\pi(\omega)$, given by

$$v_\pi(s) = \sum_\omega p_\pi(\omega)\delta_{S_0,s}G(\omega) = \sum_\omega p(\omega)\delta_{S_0,s} \sum_{t=0}^{T(\omega)-1} \gamma^t R_t, \tag{2.9}$$

where only trajectories where the initial state S_0 is equal to s contribute to the sum, given by the use of the *Kronecker delta function*[1] (also called an indicator function), $\delta_{a,b}$ [67].

It is crucial to remember that value functions can only be defined in terms of a policy. Additionally, one implicitly makes the value function depend on both the reward function and environment function. Given that $G_t = R_t + \gamma G_{t+1}$, we can write eq. (2.8) as a recursive equation, as shown in eq. (2.10).

$$\begin{aligned} v_\pi(s) &= \mathop{\mathbb{E}}_{\omega\sim\pi|S_0=s}[r_t + \gamma G_{t+1}] \\ &= \mathop{\mathbb{E}}_{\omega\sim\pi|S_0=s}[r_t] + \gamma \mathop{\mathbb{E}}_{\omega\sim\pi|S_0=s}[G_{t+1}] \\ &= \sum_{a\in\mathcal{A}(s)} \sum_{s',r} p(s',r|s,a)r(s',s,a) + \gamma \sum_{a\in\mathcal{A}(s)} \sum_{s',r} p(s',r|s,a)v(s') \\ &= \sum_{a\in\mathcal{A}(s)} \sum_{s',r} p(s',r|s,a)\left[r(s',s,a) + \gamma v(s')\right], \end{aligned} \tag{2.10}$$

which states that the value of a state s is defined only in terms of the values of the states that s can transition to, and the reward received for moving between them.

In the deterministic case, eq. (2.10) simplifies to

$$v_\pi(s) = r(f(s,\pi(s)), s, \pi(s)) + \gamma v_\pi(f(s,\pi(s))), \tag{2.11}$$

where the selected action in state s is given by $a = \pi(s)$ and the next state, $s' = f(s,\pi(s))$. The reward is also awarded deterministically depending on the current state, the action taken and the next state.

One can also extend this idea to choosing an arbitrary action, a, in the current state, but then following the same policy, π, for the rest of the trajectory. This allows us to compare immediate actions with one another. We can capture this idea in what is known as the **state-action-value function**, which is given the symbol $q_\pi(s,a)$ and is defined as

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)\left[r(s',s,a) + \gamma v(s')\right], \tag{2.12}$$

and in the deterministic case as

$$q_\pi(s,a) = r(f(s,a), s, a) + v_\pi(f(s,a)). \tag{2.13}$$

This allows us to make the assertion that if $q_\pi(s,a) \geq v_\pi(s)$ then we can improve the policy π by taking the action a in the state s instead of the action $\pi(s)$. One can

[1] $\delta_{a,b}$ equals 1 only when $a = b$ and 0 otherwise. This assumes that the states take discrete values. For the case of continuous variables, the corresponding indicator function is called a *Dirac delta function* and is written as $\delta(a-b)$ [67].

see that these definitions lend themselves to rigorously defining the quality of different policies, and reduce the problem to looking at single actions in single states, rather than considering entire trajectories.

The equations for $q_\pi(s,a)$ and $v_\pi(s)$ define the **Bellman equations** [27], which enable a rigorous, algorithmic optimisation strategy, known as *Dynamic Programming* (DP) [27, 68].

To end this section, we will define the goal of RL in these recently defined terms. We can state that the goal of RL is to find a policy π^*, which satisfies

$$\pi^* = \arg\max_\pi v_\pi(s) \ \forall \ s. \tag{2.14}$$

As a consequence of this definition, the optimal policy π^* is the policy, or set of policies which obey

$$v_{\pi^*}(s) \geq v_\pi(s) \ \forall \ s \in \mathcal{S} \text{ and } \pi. \tag{2.15}$$

In words, eq. (2.14) states that the goal of RL is to choose an optimal policy, which maximises the expected cumulative reward of an agent when interacting with an environment starting from any state.

One may also be interested in a special optimisation case, which focuses on maximising values under a set of initial states, as long as the probability distribution of being in that state is stationary and well-defined, yielding an optimal policy

$$\pi^* = \arg\max_\pi \left[\sum_{s \in \mathcal{S}} d(s) v_\pi(s) \right], \tag{2.16}$$

where $d(s)$ defines the probability distribution of state s being the initial state at $t = 0$. This definition reduces to eq. (2.14) in the case that $d(s)$ is uniform across all states. An optimal policy in eq. (2.14) is always an optimal policy under eq. (2.16). This is not true for the converse, as eq. (2.16) only cares about the policy in the states that are visited by the optimal policy from the initial state distribution; the policy in non-accessible states (from the initial starting states) could be random under eq. (2.16) but need to be optimal under eq. (2.14).

2.2 Grid World

As with any showcasing any complex theory, it is helpful to have a toy problem upon which one can apply the theory. A prototypical problem for RL is navigation through a 2-dimensional grid world, as shown in fig. 2.2, with the goal of reaching a particular tile on the map. We express the optimal behaviour as reaching the target tile in the shortest amount of time. Each different tile in the grid world takes a different amount of time to cross. We can think of each different tile as a different type of terrain, like on a map. Green represents grass, the easiest tile to cross, taking only 1 unit of time. Blue represents water, costing 3 units of time, while grey represents mountains, taking 5 units of time to cross. Finally, there is the gold tile, which represents the target destination in the grid world. In this simplified world, an agent can only move to adjacent tiles, and the "game" ends when the agent reaches the gold tile.

$(1,1)$	$(2,1)$	$(3,1)$	$(4,1)$	$(5,1)$	$(6,1)$	$(7,1)$
$(1,2)$	$(2,2)$	$(3,2)$	$(4,2)$	$(5,2)$	$(6,2)$	$(7,2)$
$(1,3)$	$(2,3)$	$(3,3)$	$(4,3)$	$(5,3)$	$(6,3)$	$(7,3)$
$(1,4)$	$(2,4)$	$(3,4)$	$(4,4)$	$(5,4)$	$(6,4)$	$(7,4)$
$(1,5)$	$(2,5)$	$(3,5)$	$(4,5)$	$(5,5)$	$(6,5)$	$(7,5)$
$(1,6)$	$(2,6)$	$(3,6)$	$(4,6)$	$(5,6)$	$(6,6)$	$(7,6)$
$(1,7)$	$(2,7)$	$(3,7)$	$(4,7)$	$(5,7)$	$(6,7)$	$(7,7)$

Figure 2.2: A representation of a grid world. Green represents grass, blue represents water, grey represents mountains and the yellow/gold colour represents the goal (or exit) tile. The labels of each tile shows the (x, y) position of the tile.

Using what we have learnt in the previous section, we can translate this problem into a reinforcement learning problem. We can start with the state of the problem. In this case, the state would be the position of the agent in the grid world. One can label the west to east (horizontal) and north to south (vertical) directions as x and y respectively. We will assume for now that the terrain map does not change, which means we do not have to include it in part of the state[2], as it becomes an implicit part of the problem. We can denote the state as $s = (x, y)$.

The set of actions of an agent can be denoted by $\mathcal{A} \in \{\uparrow, \downarrow, \rightarrow, \leftarrow\}$ for north, south, east and west respectively. The action which maps from state s to the next state s' is denoted as a. The environment function is deterministic and determines how s is mapped to s' using a via

$$s' = f(s, a) = \begin{cases} (x, y-1), & a \text{ is } \uparrow \\ (x, y+1), & a \text{ is } \downarrow \\ (x+1, y), & a \text{ is } \rightarrow \\ (x-1, y), & a \text{ is } \leftarrow \end{cases}. \tag{2.17}$$

Our final choice is the reward function. How are we to encode the goal of reaching the gold tile in the shortest possible time? The goal of a RL problem is to maximise a reward. As our grid world goal is to minimise the time, we can instead maximise the negative of the total time taken. We can encode the cost to travel on a given tile as

$$c(s) = \begin{cases} 0, & s \text{ is a goal (gold)} \\ 1, & s \text{ is grass (green)} \\ 3, & s \text{ is water (blue)} \\ 5, & s \text{ is mountains (grey)} \end{cases}, \tag{2.18}$$

[2] If we were training an agent to operate in any grid world configuration, we would have to include the details of the tile configuration in the state.

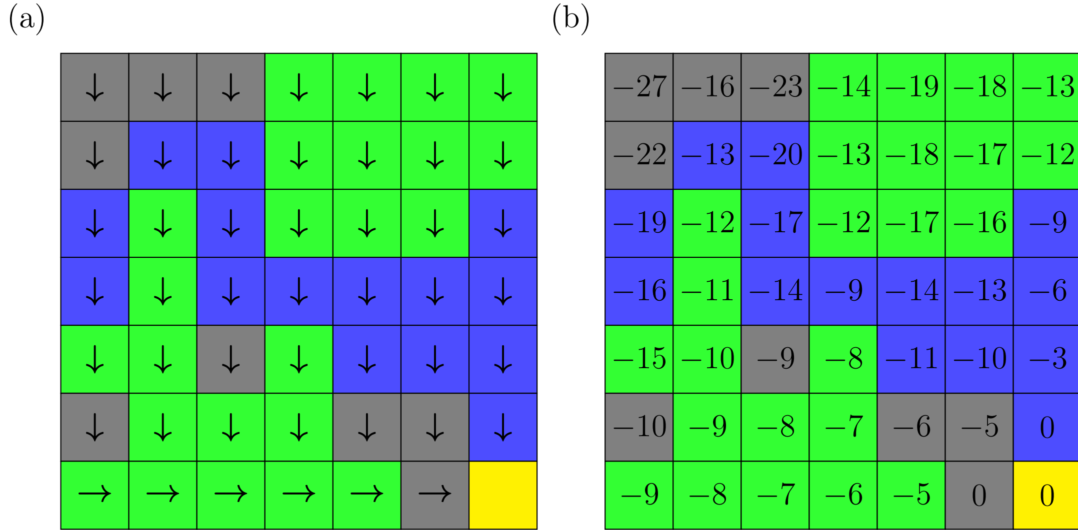

Figure 2.3: (a) shows the initial policy, π_0, in grid world. We do not allow actions that would cause the agent to exit the grid world shown. (b) shows the initial value function, v_{π_0}, in grid world. This value function is only defined in terms of the policy π_0, shown in (a).

where s represents the tile, indicated by an (x, y) position on the grid. The reward is given as

$$r(s', s, a) = -c(s'), \tag{2.19}$$

where the reward is deterministic and only depends on the tile that is travelled into. In general, the reward function can depend on any number of variables, but in this case, it only depends on the next tile entered. As an example, if a player moves from a grass tile (green) into a mountain tile (grey), they will receive a reward of -5. In fact, moving from any tile into a mountain tile will receive a reward of -5.

We can represent a deterministic policy with arrows in each tile, indicating the direction of travel from that tile, as shown in fig. 2.3(a). If we imagine starting at $(4, 6)$, we can see that the path followed is

$$\begin{aligned}
\omega_{\pi_0}(S_0 = (4,6)) = \{ & S_0 = (4,6), A_0 = \downarrow, R_0 = -1 \\
& S_1 = (4,7), A_1 = \rightarrow, R_1 = -1 \\
& S_2 = (5,7), A_2 = \rightarrow, R_2 = -5 \\
& S_3 = (6,7), A_3 = \rightarrow, R_3 = 0 \\
& S_4 = (7,7)\},
\end{aligned} \tag{2.20}$$

where we set the initial time to $t = 0$, the initial state at $S_0 = (4, 6)$ and follow the policy π_0 until the terminal goal state, giving us a trajectory $\omega_{\pi_0}(S_0 = (4, 6))$. We can calculate the return of the trajectory as $G_0 = R_0 + R_1 + R_2 + R_3 = -7$, using $\gamma = 1$. As we are in a deterministic setting, we know that $v_{\pi_0}(S_0 = (4, 6)) = G_0 = -7$ since there is only one possible trajectory starting at $(4, 6)$.

We can repeat this type of calculation, starting at each tile on the map, to calculate the value of each state under the initial policy. However, this method is very inefficient, as many trajectories contain sub-trajectories which may have already been evaluated.

Instead, we can use the recursive form of the value function to calculate each value by only visiting each state a single time.

We start by labelling the terminal state (gold) as having a value of 0, by definition, as the episode ends here and there are no more subsequent rewards. If you look carefully at the policy, one can see that there are no cycles and one always ends in the terminal state, regardless of the starting position. For this reason, we can use this initial value, along with eq. (2.11), to "*propagate*" the values backwards so that each state has a corresponding value (under the policy π_0). We then visit all the tiles which are adjacent to the labelled states, updating those that have actions that take an agent into one of our already labelled states, using eq. (2.11).

Under this new method, after giving the initial value to the terminal state, we visit states $(6, 7)$ and $(7, 6)$, which also have a value of 0, since you get a 0 reward for entering the terminal state and it has zero value in that state. Next, we visit $(5, 7)$, $(6, 6)$ and $(7, 5)$. They have actions taking an agent into a currently labelled state; the values of these next states are -5, -5 and -3 respectively, as the first two enter a mountain tile and the last enters a water tile. This process is iterated, until all tiles are labelled. The final result of this process is shown in fig. 2.3(b).

Notice that while both methods give the same answer, the second is much more efficient at calculating the answer. The efficiency comes from breaking the problem down into a single state transition, and choosing a sensible order in which to evaluate the states. The order in which we calculate the values is called a **sweep**.

2.3 Dynamic Programming

A traditional approach to solving RL problems is to use *dynamic programming*. This section will outline one approach to using this technique to solve the problems of interest *exactly*.

2.3.1 Bellman Equations and Optimality

In Section 2.1, we defined equations 2.11 and 2.13 for v_π and q_π respectively. We called these the **Bellman Equations**. These two functions are very important to RL as they provide a **partial ordering** of the quality of policies. A policy, π, is defined to be "better" than or equal to a policy, π', if the expected return is greater than or equal to the expected return of π' for all states [27]. This is codified by

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s) \; \forall \; s \in \mathcal{S}, \tag{2.21}$$

where $\mathcal{S}$ defines the state space of the problem, and $\iff$ is the symbol for *if and only if*. This is only a partial ordering, since it does not disambiguate two policies with the same values in each state, even if the policies are different. However, this condition is enough to ensure that we can define at least one optimal policy, denoted as π^*. Even though there may be multiple optimal policies, they all share the same state-value function, called the **optimal state-value function**, denoted as v^*, and defined as

$$v^*(s) \equiv \max_\pi v_\pi(s), \; \forall \; s \in \mathcal{S}. \tag{2.22}$$

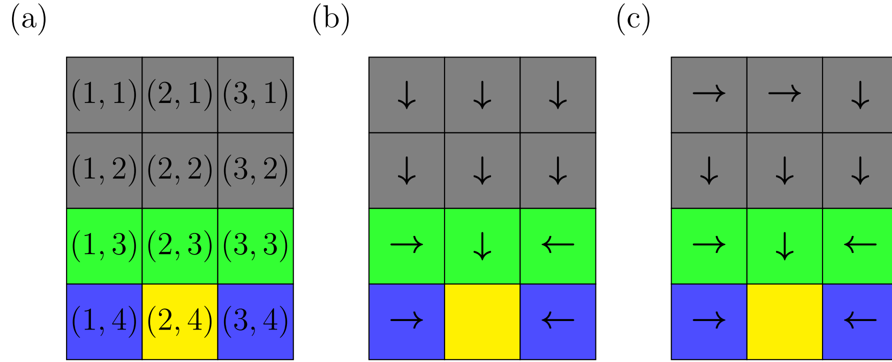

Figure 2.4: (a) The map, along with the coordinates in the form of (x, y). (b) The optimal policy w.r.t eq. (2.22). (c) A policy which is not optimal in terms of eq. (2.22) but is optimal in terms of eq. (2.16), in the case that $d(s) = 0$ for $y = 1$. This means that the probability of starting at the top of the map is zero, and hence, the optimal policy will never visit any of those states, so it is still an optimal policy when we take into account the initial states.

Optimal policies also share the same **optimal state-action-value function**, denoted as q^* and given by

$$q^*(s, a) \equiv \max_{\pi} q_\pi(s, a), \ \forall \ s \in \mathcal{S} \text{ and } a \in \mathcal{A}(s). \tag{2.23}$$

One can also write the above equation in terms of v^*, such that

$$q^*(s, a) \equiv \sum_{s', r} p(s', r|s, a) \left[r + \gamma v^*(s') \right]. \tag{2.24}$$

One can see that this is a stronger set of conditions for defining optimality than eq. (2.16), as it does not define an initial state (or distribution of initial states). If we restrict our search space of policies to one which does not explore part of the possible state space (as it is not optimal), then a policy which is not optimal in this area is still optimal in terms of the goal we care about. We can use the grid world problem to understand this.

Picture the grid world shown in fig. 2.4(a). If we were to follow the definition of optimality given by eq. (2.22), then we would arrive at an optimal policy given by fig. 2.4(b). However, if we know that we only start in tiles where $y > 1$ (i.e. excluding the top line of the map), then we know that it is optimal to avoid this line. As the agent never enters the state space in the top row, the policy here is irrelevant, as optimality can be defined in terms of the starting state, given by eq. (2.16). We can see that for all states actually visited by the policies, the values are identical. However, we do not always know which states will not be visited by an optimal policy starting at a distribution of initial states, and as such, we should prefer the more general definition of optimality. This ensures that we do not discard part of the state space which may be essential to finding the optimal policy.

2.3.2 Policy Evaluation

In our grid world example, we presented a few ways of calculating the values for a given state. These methods relied on the fact that we knew where the exit tile was, and knew

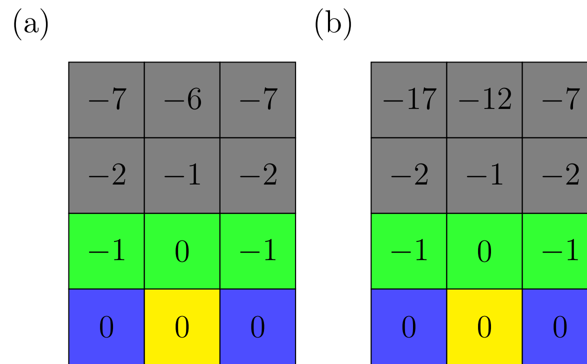

Figure 2.5: (a) and (b) show the value functions for the policy in fig. 2.4 (b) and (c) respectively. The values for the visited states under an initial starting state distribution of $y > 1$ are identical for both policies.

that it would be a terminal state. This sort of information is problem specific, and the dynamics of the environment are not always known, but need to be explored. It is therefore important that we discuss a more general way of constructing the values for a given policy.

Trajectory Evaluation

The first method that we should discuss is directly evaluating the returns of a trajectory from a state. By definition in eq. (2.8), we know that the value of a state is the expected discounted return of a trajectory starting at that state. If we wish to estimate this value, we can just run trajectories starting at the state s, and take an average. In the episodic and deterministic settings, such as grid world, we can just run a single trajectory starting at s until termination and calculate the return by adding up the discounted rewards. In a non-deterministic environment, or with a stochastic policy, one has to run many trajectories and take an average. This average can be estimated to any arbitrary precision at the cost of more trajectories. In the stochastic case, running trajectories to estimate quantities like returns is a form of **Monte Carlo**, covered in light detail in Section 4.3.

This approach is computationally expensive and does not benefit from the recursive nature of the Bellman equations and requires re-calculating values for a large state space. Additionally, in the non-episodic case, where $T \to \infty$, one usually has to settle for an estimate for the state, taking only $T_{\max}$ steps in the trajectory, where $T_{\max}$ is finite. Remember that $\gamma < 1$ and so the tth step in the trajectory has a prefactor of γ^{t-1}. If the rewards are static and finite, then the contributions from additional terms tend towards 0, making the approximation arbitrarily accurate.

Exact Value Calculation

If the dynamics of a problem are entirely known, then we have a system of equations, one for each state s, given by

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right]. \tag{2.25}$$

We have $|\mathcal{S}|$ unknowns (i.e. the number of states) and all equations are linear in the unknowns. This can be solved as a simultaneous equations problem. The dynamics is

entirely given by the probability distribution $p(s', r|s, a)$ and the policy π.

Value Iteration

Instead of performing many trajectories, one can instead use the Bellman equations to iterate the state values until they have converged. To start, we initialise a set of values for each state. This does not need to be a correct value, so a good choice is to usually set the values of each state to zero, as all terminal states will have the correct value, despite not knowing which states are terminal ahead of time. As this will be an iterative process, we will use the superscript (k) to specify the kth iteration of a quantity. Our initial conditions are

$$V_\pi^{(0)}(s) = 0 \ \forall \ s, \tag{2.26}$$

where we use the symbol $V_\pi^{(u)}$ to denote the approximate value function at iteration step u, whereas v_π is reserved for the true value function. Notation throughout this chapter will use a capital V to denote a current approximation of the value function and lowercase v to denote the true value function.

The only condition on our initialisation is to make sure that terminal states are set to zero. As we are initialising all states at zero, this ensures the terminal states are also initialised at zero. The way we update the value functions is by applying the Bellman equations, giving

$$V_\pi^{(k+1)}(s) = \sum_{a \in \mathcal{A}(s)} \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma V_\pi^{(k)}(s')\right] \ \forall \ s, \tag{2.27}$$

which, in the deterministic case, becomes

$$V_\pi^{(k+1)}(s) = r\left(f[s, \pi(s)], s, \pi(s)\right) + \gamma V_\pi^{(k)}\left(f[s, \pi(s)]\right) \ \forall \ s, \tag{2.28}$$

where $f[s, \pi(s)]$ is the state transitioned to, from s, using the action selected from the policy. One should pay attention to the fact that we update all states in one go, and only use values from the previous iteration to produce the next iteration.

This process will iterate towards the true value function. In the case of continuing problems, then one iterates until differences in subsequent iterations are within a chosen arbitrary threshold. We can apply this on the grid world problem in fig. 2.4(a), using the policy in fig. 2.4(b). Figure 2.6 shows the iterations of this happening until there are no more differences, starting from $k = 0$, until $k = 5$. Notice that $k = 4$ and $k = 5$ are identical, so the iteration process ends. The algorithm terminates when an iterations yields no changes. We can see that the final value function at $k = 5$ is the same as the one presented in fig. 2.5(a).

The process of value iteration is guaranteed to converge to the correct answer, provided that each state is visited infinitely often. This is because eq. (2.27) clearly has a fixed point when $V_\pi^{(k)} = v_\pi$.

2.3.3 Policy Iteration

Now that we can calculate the value function, we can equally calculate the **state-action-value function**, via

$$q_\pi(s, a) = \sum_{s',r} p(s', r|s, a) \left[r(s', s, a) + \gamma v_\pi(s')\right], \tag{2.29}$$

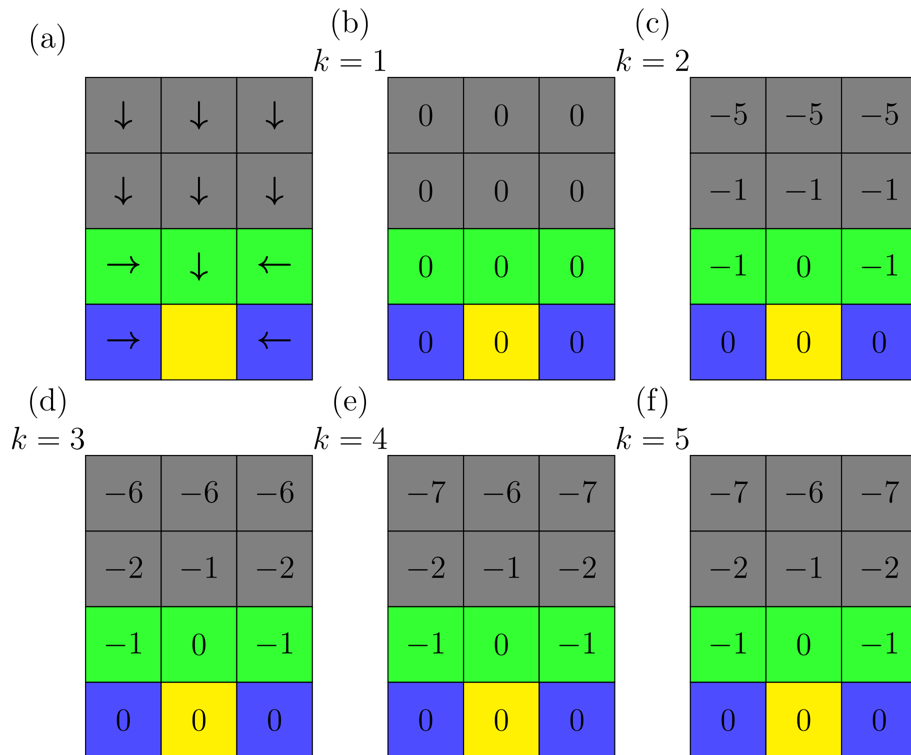

Figure 2.6: (a) A given policy π, same as fig. 2.4(a). (b)-(f) The value iteration sweeping over all states using eq. (2.27) at iteration k. A final round of value iteration is shown in (f) to confirm that this is a fixed point and is unchanged after update.

which in the deterministic regime becomes eq. (2.13).

Using $q_\pi(s, a)$ allows us to compare a single transition action a in the state s. In fact, if we have a policy π', we can prove that it is better than a current policy, π, under the condition that

$$v_{\pi'}(s) \geq v_\pi(s) \;\forall\; s. \tag{2.30}$$

One can see that if we create a new policy such that

$$\pi'(s) = \max_a q_\pi(s, a) \;\forall\; s, \tag{2.31}$$

then we are guaranteed to satisfy eq. (2.30).

We can use eq. (2.31) to generate an improved policy, π'. In turn, we can use any method to evaluate the new value function, $v_{\pi'}$, and then continue the process. We can see that the optimal policy, π^*, is a fixed point of this process. For the same reason that the value function iteration is guaranteed to converge, this process is also guaranteed to converge to the optimal policy. This allows us to solve most simple problems exactly.

The policy iteration algorithm to find an optimal policy is summarised below [27]:

1. Generate an initial policy π_0 and set $n = 0$, where π_n is the nth policy.
2. Initialise $V_{\pi_n}^{(k)}(s) = 0 \;\forall\; s$ and set $k = 0$.
3. Calculate $V_{\pi_n}^{(k+1)}$ using eq. (2.27), sweeping over all states.

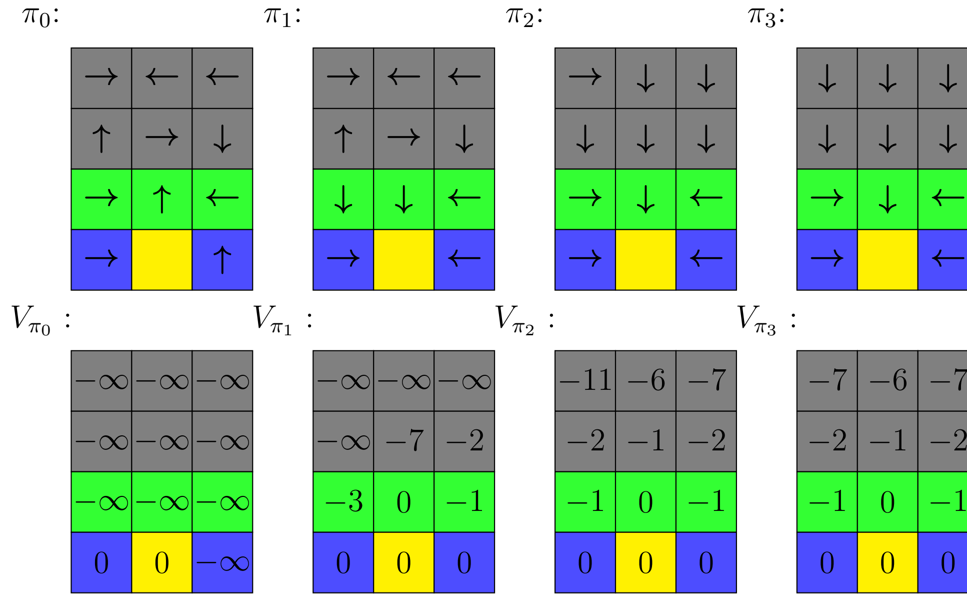

Figure 2.7: Process of generating an optimal policy from a randomly initialised policy via policy iteration. Values are calculated by running a trajectory, and detecting loops (to calculate $-\infty$ values), instead of the value iteration to avoid infinite loops. The final optimal policy generated is the same as fig. 2.4(b). The final policy iteration is not shown, as $\pi_4 = \pi_3$, showing that π_3 is the optimal policy .

4. Check for convergence between $V_{\pi_n}^{(k+1)}$ and $V_{\pi_n}^{(k)}$. If converged, set $V_{\pi_n} = V_{\pi_n}^{(k+1)}$ and continue, else go to step (3).
5. Create a new policy, π_{n+1} using eq. (2.31) based off π_n.
6. Stop if $\pi_{n+1} = \pi_n$ and output $\pi^* = \pi_n$ and $V^*(s) = V_{\pi_n}$.
7. Increment n, such that $n \leftarrow n + 1$ and go to step (2).

We can substitute any valid value function evaluation technique to calculate v_π into this algorithm.

Figure 2.6 shows this algorithm in practice, by applying it to a random policy. We only show the final value function for each policy, as the process is already detailed in fig. 2.6.

2.4 Model-Free Methods

In the last section, we covered the basic terminology of RL, and formulated a simple grid world game as a RL problem. We have explored exact methods of solving these problems using the Bellman equations and dynamic programming. We also introduced a basic, traditional, algorithm for finding the optimal policy of a generic RL problem.

However, the field of RL rarely focuses on the methods presented in this section, despite being the bedrock of the field. Many problems of interest suffer due to a lack of a model for the environment, rendering dynamic programming techniques unsuitable. Additionally, many problems that RL is applied to have an incredibly large state and/or action space, which make these methods computationally infeasible. In this section, we will discuss

ways of learning, without explicit access to a model, but rather from interactions with the environment using via *model-free methods*.

Previously, we assumed that one could solve problems by exhaustively searching the entire state space, and confidently knowing how the environment would respond to actions; we call this having a *model* of the environment. These assumptions are not always feasible. Imagine the scenario of training a self driving car. In this scenario, the behaviour of other objects in the environment, such as people, cannot be effectively contained in a model of the environment. There are some methods that try to learn a model of the environment to help with prediction, but it is common to have access to no model at all.

Additionally, most problems have a *combinatorial* state-space, e.g. backgammon has over 10^{20} states [27], rendering the exact methods infeasible. To add to the complexity, there are some problems with continuous action and state spaces which are not well handled by the exact methods; in practice, these are usually discretised or approximated, however, one can see how the state space could grow arbitrarily large.

It should be stated that the dynamic programming methods described previously, while impractical for large problems, are actually quite efficient. In the worst case, the time taken to reach an optimal policy is polynomial in the number of states and actions. If a problem has n states and k actions, then it is clear that one can choose k actions in each state n, and as such, the number of total policies is k^n. Despite the number of policies growing exponentially with the state size, dynamic programming can find the globally optimal policy in polynomial time. This is extremely efficient. Neththertheless, if n is too large to sweep over, then it is an impractical algorithm. Many problems in the real world fit into this category.

One way in which we can focus the search in the policy space is to learn through experience, focusing on the part of the state space that an agent is likely to visit, and therefore likely to be important in finding the optimal policy. These are collectively known as *Monte Carlo* methods and form the basis of most modern RL algorithms. Unlike the previous section, we do not assume any knowledge of the environment (i.e. the transition function/rates), making the techniques *model-free*. A typical Monte Carlo method only requires *experience* — a sample sequence of states, actions and rewards from actual or simulated interaction with an environment.

2.4.1 Policy Evaluation through Interaction

In Section 2.3.2, we discussed using trajectories to evaluate the value of a given state. While this is not the most efficient approach in some problems, such as grid world, Monte Carlo trajectories form a very important part of modern RL, as dynamic programming (DP) methods are often infeasible.

To remind ourselves, the definition of the value of a state is

$$v_\pi(s) = \mathop{\mathbb{E}}_{\omega \sim \pi | S_0 = s}[G(\omega)], \tag{2.32}$$

where $G(\omega)$ is the return (sum of discounted rewards) of a trajectory beginning at state s. One can imagine running a trajectory beginning at some initial state s drawn from a distribution of initial states with probability $d(s)$. The quality of a policy is given exactly by the expected discounted return of trajectories generated by the starting distribution

and the policy π. In terms of the values of the initial states, we write the expected return as

$$\langle G_\pi \rangle = \sum_{s \in \mathcal{S}} d(s) v_\pi(s). \tag{2.33}$$

As before, this provides a *partial ordering* over all policies, such that if $\langle G_{\pi'} \rangle > \langle G_\pi \rangle$, then $\pi' > \pi$. It is only a partial ordering, since if $\langle G_{\pi'} \rangle = \langle G_\pi \rangle$, then both policies are equal and cannot be distinguished using this metric. Notice that this is a slightly different metric to the Bellman policy ordering (given in eq. (2.21)), as it compares the weight of states, and does not require iterating over the entire state space to compare policies. We have relaxed the restriction that the policy must have the most optimal value function over *all* states, but instead focus on a weighted average of the values of the initial states. A true optimal policy (over all states) is also an optimal policy with respect to an initial distribution, but the converse cannot be guaranteed.

In the dynamic programming approach of policy evaluation, one would have to visit every single state of the problem iteratively. For many problems this can be prohibitively expensive as the state space of most problems is very large. Instead, one can consider which states are actually relevant to solving the problem, as some states may not even be reachable depending on the initialisation conditions. We touched on this earlier in this chapter in fig. 2.4 and fig. 2.5, where under the initialisation conditions of $y > 1$, these two policies have the same expected returns, as the trajectories from the initial states never enter the regions where the policy differs. Often, optimisation of dynamic programming algorithms comes in the form of biasing the sweep of states to those which are more likely to be relevant to solving the problem. Take the example of finding the shortest path between two points. The default algorithm for this is called "Dijkstra's algorithm" [69], which is often augmented for the A* algorithm [70], which prioritises searching in the direction of the target, as it deems these states *more relevant*.

Along with prioritising important states, interaction with the environment does not require the agent to have a model of the environment to learn. Dynamic programming explicitly requires a model of the environment to learn, which we called a *model-based* method. This usually also involves methods that have some aspect of *planning*, which allow an agent to plan out future actions, much like a Chess player visualising the board a few moves into the future[3].

In order to evaluate a given state, one can simply run a number of trajectories, $\omega_\pi^{(i)}$, using the current policy, π, starting at a state s to approximate the value of that state, such that

$$v_\pi(s) \approx \frac{1}{N} \sum_{i}^{N} G(\omega_\pi^{(i)}), \tag{2.34}$$

where $G(\omega)$ is the total return of the trajectory ω, and N is the total number of samples. Equation (2.34) becomes an equality in the limit of $N \to \infty$. Notice that one can sample ω without having to know the specific model of the environment. Instead, one need only be able to interact with the environment and does not need to know the explicit probabilities of certain transitions and rewards.

[3]Planning is not extensively covered here, but Chapter 8 of [27] provides a basic overview. Planning is also extensively used in *Monte Carlo Tree Search* (MCTS), which is one of the main algorithms behind AlphaGo [71], the first computer program to beat a world champion at the game of *Go*.

Data: Policy π
Result: Estimated state-value function $V_\pi(s) \approx v_\pi(s)$.
1 Initialise a state-value-function $V_\pi(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$;
2 Initialise an empty list, $Returns(s)$ of sampled returns for all $s \in \mathcal{S}$;
3 **for** $i \leftarrow 1$ **to** N_{max} **do**
4   Sample a trajectory, ω, using π: $\omega = (S_0, A_0, R_0 \ldots, S_T, A_{T-1}, R_{T-1})$;
5   $G \leftarrow 0$;
6   **for** t *in* $T-1, T-2, \ldots, 0$ **do**
7     $G \leftarrow \gamma G + R_t$;
8   Append G to $Returns(S_t)$;
9     $V_\pi(S_t) \leftarrow \text{average}(Returns(s))$;
10   **end**
11 **end**

Algorithm 1: Value-Function estimation using Monte Carlo policy evaluation.

The algorithm for estimating the value function $V_\pi(s) \approx v_\pi(s)$ using Monte Carlo is given in Algorithm 1. It should be noted that this is called the *every-visit* Monte Carlo method, which averages returns from every single visit to the state s. An alternative approach is to only alter the average to $V_\pi(S_t)$ if it is the first visit in the trajectory, known as the *first-visit Monte Carlo method*, discussed in more detail in Chapter 5 of [27].

2.4.2 Monte Carlo Policy Iteration

Estimating state-values tends to be less helpful when one does not have access to the model of the environment. Previously, we could construct a greedy policy with a value function $V_\pi(s)$ using

$$\pi^{(greedy)}(s) = \arg\max_a \left[\sum_{s' \in \mathcal{S}} p(s', r|s, a) \left(r + \gamma V_\pi(s') \right) \right], \tag{2.35}$$

where $p(s', r|s, a)$ is the model of the environment. However, without a model, we cannot compute this. Instead, we should attempt to estimate the state-action-value function, $q_\pi(s, a)$, for a given policy. One approach to estimating the state-action-value function is to randomly start in any state and select a random starting action. After this first transition, one follows the given policy π until the end of the trajectory. One can then calculate the return for the starting state and action, and use this as an estimate for the q_π value at the initial state, using the initial action. This is known as *exploring starts*. We do not consider this approach further, as it is prohibitively computationally expensive in most cases.

Instead, we consider making our policy stochastic, to ensure there is some exploration. One way to make a policy stochastic, is to make it *ϵ-greedy*, which means that with probability $(1-\epsilon)$, we pick the actions according to $a^*_\pi = \arg\max_{a'} q_\pi(s, a')$, and otherwise, choose a completely random action. The probability of this can be defined as

$$\pi(a|s) = \begin{cases} (1-\epsilon) + \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a = a^*_\pi \\ \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a \neq a^*_\pi \end{cases} \tag{2.36}$$

We choose ϵ to be between 0 and 1. Notice that we recover a greedy policy if we set ϵ to zero. Usually, one anneals ϵ towards zero to decrease exploration and maximise exploitation at the end of training.

```
   Data: Soft policy π
   Result: Estimated state-action-value function Q(s, a) ≈ q_π(s, a).
 1 Initialise a state-action-value-function Q(s, a) ∈ ℝ, arbitrarily, for all s ∈ S;
 2 Initialise an empty list, Returns(s, a) of sampled returns for all s ∈ S;
 3 for i ← 1 to N_max do
 4 |   Sample a trajectory, ω, using π: ω = (S_0, A_0, R_0 ..., S_T, A_{T-1}, R_{T-1});
 5 |   G ← 0;
 6 |   for t in T − 1, T − 2, ..., 0 do
 7 |   |   G ← γG + R_t;
 8 |   |   Append G to Returns(S_t, A_t);
 9 |   |   Q(S_t, A_t) ← average(Returns(S_t, A_t));
10 |   end
11 end
```

Algorithm 2: State-action-value function of the soft policy π, which ensures that all actions are taken with some finite, non-zero, probability.

We can now evaluate the state-action-value function of the exploring policy π, using Algorithm 2. Once the policy has been estimated, one can iterate the policy such that $\pi^{k+1}(a|s) = \arg\max_{a'} Q^{(k)}(s, a') \;\forall\; s \in \mathcal{S}$, similar to the policy iteration shown in Section 2.3.3. This iteration is guaranteed to converge to the optimal soft policy [27]. One can also imagine this soft policy as always choosing the greedy action, but the environment randomly (with probability ϵ) overrides the agent's action with a uniformly random one instead. In this case, the best that the agent can do is choose an action which maximises the state-action-value function, which takes into account this action randomisation.

2.4.3 On/Off Policy Methods

In the last section, we presented learning a *soft*-policy[4] π for optimisation, ensuring that samples have some exploration. However, this policy can be quite suboptimal. We will use an example from grid world to emphasise this.

Let us introduce a new type of tile to our map and call it "lava". This will symbolise a type of tile that is to be strongly avoided, and let us give it a cost of some high value, say 100, meaning that the agent will receive a -100 penalty for entering the tile, causing the episode to end. If we have an ϵ-*greedy* policy, then there is a chance the agent will randomly enter the tile if it is adjacent. For this reason, there is a strong influence to stay far away from any lava tiles, even if the optimal path to the exit goes near them.

Take the policies given in fig. 2.8, given a starting point of the south-west corner, we can clearly see that the optimal path is to avoid the lava, as shown in (a). However, if we were to instead calculate the optimal ϵ-*greedy* (soft)policy, we can see that this will diverge from the "true" optimal path, and we pay the price for baking in exploration to the existing policy.

[4] A soft policy is one which is stochastic, and takes every available action a with at least $\pi(a|s) > 0$.

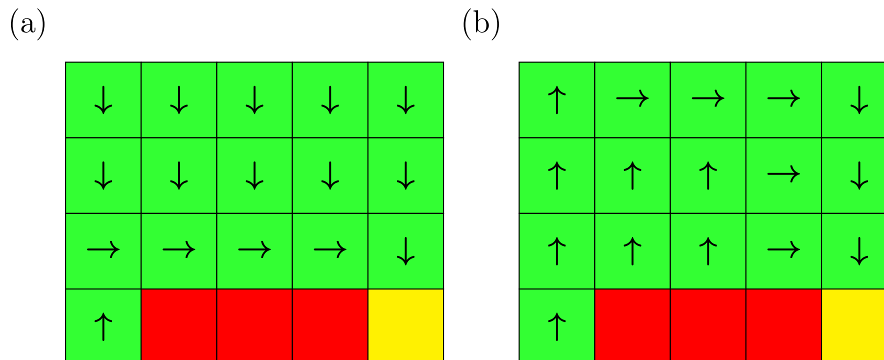

Figure 2.8: A grid world example with lava blocks at the bottom. A soft-policy has a small chance of entering the lava if the agent is in a tile near it. Imagine the environment has a strong wind that will move the agent at random into another tile with a small chance. In this situation, the only optimal policy is one which takes you far away from costly tiles, as these states will have a lower value as there is a contribution from the random chance that you are knocked into the adjacent costly tile, whatever your desired action. This example illustrates that the optimal hard policy, shown in (a), is not the optimal soft policy (b).

To keep exploration, but ensuring we learn the optimal policy, we can modify our algorithm to be **off-policy**. An off-policy algorithm uses a different policy to generate trajectories, from which we can learn the state-action-value function of our actual policy π, then used for policy iteration. In general, we can call this other policy, b, which is used to generate sample trajectories to learn from.

Imagine that we have a trajectory ω, generated using b, which has a probability $p_b(\omega)$. Under a different policy π, this probability would be $p_\pi(\omega)$. Remember that we can calculate the expected value of the return (under π) via

$$\langle G\rangle_\pi = \mathop{\mathbb{E}}_{\omega\sim\pi}[G(\omega)] = \sum_\omega p_\pi(\omega)G(\omega), \tag{2.37}$$

which assumes that ω starts in a state s with probability $d(s)$.

We can multiply each term inside the sum by $\frac{p_b(\omega)}{p_b(\omega)}$, as this is equal to 1, yielding

$$\begin{aligned}\langle G\rangle_\pi &= \sum_\omega p_\pi(\omega)G(\omega)\\ &= \sum_\omega \frac{p_b(\omega)}{p_b(\omega)}p_\pi(\omega)G(\omega)\\ &= \sum_\omega p_b(\omega)\left[\frac{p_\pi(\omega)}{p_b(\omega)}G(\omega)\right]\\ &= \mathop{\mathbb{E}}_{\omega\sim b}\left[\frac{p_\pi(\omega)}{p_b(\omega)}G(\omega)\right]\\ &= \mathop{\mathbb{E}}_{\omega\sim b}\left[\rho(\omega)G(\omega)\right],\end{aligned} \tag{2.38}$$

where $\rho(\omega) = \frac{p_\pi(\omega)}{p_b(\omega)}$. We can expand the probability of a particular trajectory under policy

μ into a product of probabilities, as this is a Markov decision process (MDP), yielding

$$p_\mu(\omega) = d(S_0)\ \mu(A_0|S_0)\ p(S_1|A_0, S_0)\ \mu(A_1|S_1) \ldots \mu(A_{T-1}, S_{T-1})\ p(S_T|A_{T-1}, S_{T-1}). \tag{2.39}$$

Notice that, for a given trajectory, the only terms unique to the policy are the ones given by $\mu(a_t|s_t)$. This means that if we find the ratio of $\frac{p_\pi(\omega)}{p_b(\omega)}$, all the terms except for the policy differences will cancel, meaning that the quantity ρ does not depend on the environment. We are left with

$$\begin{aligned}\frac{p_\pi(\omega)}{p_b(\omega)} &= \frac{\cancel{d(S_0)}\ \pi(A_0|S_0)\ \cancel{p(S_1|A_0, S_0)}\ \pi(A_1|S_1) \ldots \pi(A_{T-1}, S_{T-1})\ \cancel{p(S_T|A_{T-1}, S_{T-1})}}{\cancel{d(S_0)}\ b(A_0|S_0)\ \cancel{p(S_1|A_0, S_0)}\ b(A_1|S_1) \ldots b(A_{T-1}, S_{T-1})\ \cancel{p(S_T|A_{T-1}, S_{T-1})}} \\ &= \prod_{t=0}^{T-1} \frac{\pi(A_t|S_t)}{b(A_t|S_t)} = \rho(\omega). \end{aligned} \tag{2.40}$$

This technique of using different weights to estimate an expectation is called *importance sampling*. It has this name as it determines how to adjust the weighting of a given sample, depending on how representative it is under the other dynamics (other policy), or rather, how important it is to the expectation. We also discuss this technique in Chapter 4.

Let us take an example where $\rho(\omega) = 2$, such that ω is twice as likely to happen under π than b. In this case, it means that when we are sampling trajectories, ω will be half as likely to be sampled when we are using b, and therefore, we must double the contribution as a correction.

This technique of importance sampling is extremely powerful, as it generalises how to learn from experience, regardless of the policy under which the experience was generated. In the special case that $\pi = b$, then this returns to *on-policy* learning, and all the ρ factors will be equal to 1.

There are a few caveats we must address when sampling using b. If b does not sample all the possible experiences that π samples, then there are experiences for which $p_b(\omega) = 0$, and as such, the ratio in ρ becomes undefined. The only exception to this is when $p_\pi(\omega) = 0$, in which we set $\rho(\omega) = 0$. From this, we can learn that if, for an arbitrary state s, $\pi(a|s) > 0$ then we ensure that $b(a|s) > 0$. In general, we want b to be a stochastic policy in places where $\pi(a|s) > 0$.

So far, we have only come up with a way of estimating the average return of a trajectory. Let us adjust our notation so that we assess partial returns from a state s_t at time t. We will adjust ρ to be instead $\rho_{t:T}$, which is equal to

$$\rho_{t:T} = \prod_{t'=t}^{T-1} \frac{\pi(A_{t'}|S_{t'})}{b(A_{t'}|S_{t'})}. \tag{2.41}$$

We recover our original $\rho(\omega) = \rho_{0:T}$, for the entire trajectory. The partial return of the trajectory from t until the end is given as

$$G(\omega_{t:T}) = \sum_{t'=t}^{T-1} \gamma^{t'-t} R_{t'}. \tag{2.42}$$

With our notation adjusted, we can easily define the state-action-value function under π in terms of our behaviour policy b, written as

$$\begin{aligned} q_\pi(s,a) &= \mathop{\mathbb{E}}_{\omega\sim\pi}\left[G(\omega_{t:T})|S_t = s, A_t = a\right] \\ &= \mathop{\mathbb{E}}_{\omega\sim b}\left[\rho_{(t+1):T}G(\omega_{t:T})|S_t = s, A_t = a\right]. \end{aligned} \tag{2.43}$$

Under this notation, we take a trajectory ω which passes through s, taking action a at time t, and average the importance-sampled partial return from t. Notice how we are not including the term $\frac{\pi(A_t|S_t)}{b(A_t|S_t)}$, since by definition, we chose the action A_t, in state S_t at time t, as the first action in a q function is independent of a policy. Now that we have a formula, we can modify Algorithm 2 for estimating these q values, using an off-policy method, as shown in Algorithm 3.

Data: Policy π, and behaviour policy b
Result: Estimated state-action-value function $Q(s,a) \approx q_\pi(s,a)$.

```
1  Initialise a state-action-value-function Q(s,a) ∈ ℝ, arbitrarily, for all s ∈ S;
2  Initialise an empty list, Returns(s,a) of sampled returns for all s ∈ S;
3  for i ← 1 to N_max do
4      Sample a trajectory, ω, using b: ω = (S_0, A_0, R_0 ..., S_T, A_{T-1}, R_{T-1});
5      G ← 0;
6      ρ ← 1;
7      for t in T-1, T-2, ..., 0 do
8          G ← γG + R_t;
9          if t < T-1 then
10             ρ ← ρ × π(A_{t+1}|S_{t+1}) / b(A_{t+1}|S_{t+1});
11         end
12         Append (ρ × G) to Returns(S_t, A_t);
13         Q(S_t, A_t) ← average(Returns(S_t, A_t));
14     end
15 end
```

Algorithm 3: State-action-value function estimation of the policy π using the off-policy samples from b. This uses an every-visit Monte Carlo algorithm.

One should note that our estimates for $q_\pi(s,a)$ are biased using the every-visit Monte Carlo algorithm, but this bias shrinks asymptotically to zero [27] as $N_{\max} \to \infty$. One can remove this bias, using the first-visit variant, which modifies Algorithm 3 such that the importance-sampled partial return is only appended to the list of returns if the tuple (S_t, A_t) did not appear in the trajectory before t.

According to Sutton and Barto (Ref. [27]), this method of estimation can have extremely high variance, and as stated in the previous paragraph, some bias. This means that one needs to sample many trajectories to get a reliable estimate of the q function, meaning that off-policy learning can be extremely sample inefficient.

However, if one is able to get a good approximate q function, one can use policy iteration (as described in Section 2.3.3) to improve the policy π.

2.4.4 Bootstrapping: Updating Estimates with Estimates

Up until this point, we have estimated the value function using the Bellman equations or running trajectories and constructing an estimate via Monte Carlo (MC) methods. However, these methods are not mutually exclusive and can be synthesised into a combined method. Let us explore some of these methods in further detail.

Temporal Difference

We will first look at the simplest **temporal difference (TD)** method, known as TD(0) [27]. Remember that we defined the value of state to be

$$v_\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right], \tag{2.44}$$

which is just an expectation of the reward plus the discounted next state, i.e.

$$v_\pi(s) = \mathop{\mathbb{E}}_{\omega \sim \pi | S_t = s} \left[R_t + \gamma v_\pi(S_{t+1})\right]. \tag{2.45}$$

This equation tells us that we could use a single transition to estimate the value of a state s. Instead of having to generate an entire trajectory, we can just jump one step into the future, and gain an estimate of the current value. We can calculate an estimate for the "error" on a value estimate with the TD error

$$\delta_t = R_t + \gamma V_\pi(S_{t+1}) - V_\pi(S_t). \tag{2.46}$$

An algorithm to learn the value function over time with experience is given below in Algorithm 4. In essence, the aim is to minimise the expected temporal difference error, which is minimised by the actual value function v_π. Convergence is guaranteed under stochastic conditions, such as annealing α towards 0, but ensuring that

$$\lim_{T \to \infty} \left[\alpha_T\right] = 0 \tag{2.47}$$

$$\lim_{T \to \infty} \left[\sum_{t=0}^{T} \alpha_t\right] = \infty \tag{2.48}$$

$$\lim_{T \to \infty} \left[\sum_{t=0}^{T} \alpha_t^2\right] = \text{const}, \tag{2.49}$$

which are collectively known as the **Robbins-Monro conditions** [72], described briefly in Section 1.4.2.

Our algorithm is now able to update the estimate of the value function before actually finishing the trajectory, which is the real advantage of temporal difference methods as they can be used for online updates. The effectiveness of such updates depends on how accurate the estimate of the value function currently is. These methods are extremely helpful in non-episodic problems.

2.4.5 n-step Temporal Difference

Instead of including a single transition, we can include a finite number of transitions. The idea is simple, we can work with the definition of the value function and "unroll" some of

```
Data: Policy π, learning rate α ≪ 1 and discount γ.
Result: Estimated value function V_π(s) ≈ v_π(s).
1  Initialise a state-action-value-function V_π(s) ∈ ℝ, arbitrarily, for all s ∈ 𝒮,
     except for terminal states which are set to 0.;
2  for i ← 1 to N_max do
3  |  Sample an initial state s with probability s;
4  |  while s is not terminal do
5  |  |  Sample action a using π in state s;
6  |  |  Sample next state with s' ← f(s, a);
7  |  |  Sample the reward for the action r ← r(s', s, a);
8  |  |  Estimate TD error δ ← r + γV_π(s') − V_π(s);
9  |  |  V_π(s) ← V_π(s) + αδ;
10 |  |  Move to the next state with s ← s';
11 |  end
12 end
```

Algorithm 4: Value function estimation of the policy π by using 1 step temporal difference learning — TD(0).

the rewards up to a horizon, H, from the current state, such that

$$\begin{aligned} v_\pi(S_t) &= \mathop{\mathbb{E}}_{\omega\sim\pi}\left[R_t + \gamma R_{t+1} + \dots \gamma^H R_{t+H} + \gamma^{H+1} v_\pi(S_{t+H+1})\right] \\ &= \mathop{\mathbb{E}}_{\omega\sim\pi}\left[\gamma^{H+1} v_\pi(S_{t+H+1}) + \sum_{t'=t}^{t+H} \gamma^{t'-t} R_{t'}\right]. \end{aligned} \tag{2.50}$$

In the case of $H = 0$, we get back TD(0): $V_\pi(S_t) = \mathop{\mathbb{E}}_{\omega\sim\pi}\left[R_t + V_\pi(S_{t+1})\right]$. When $H \to \infty$ or stops when hitting S_T, the terminal state, we have an equation for estimating the value function via Monte Carlo (i.e. by running trajectories). This allows us to blend between temporal difference and Monte Carlo methods, in which a compromise can often lead to optimal learning.

Here, we can calculate our temporal difference error to be

$$\delta_t = \sum_{t'=t}^{t+H} \gamma^{t'-t} R_{t'} + \gamma^{H+1} V_\pi(S_{t+H+1}) - V_\pi(S_t). \tag{2.51}$$

Notice that we can also sum up the temporal difference errors for an entire trajectory, regardless of H, and provided that the value estimate is not updated during the sum, we return to the Monte Carlo "return-to-go" from the state S_t. We have therefore recovered a way of going from TD(0), all the way back to a Monte Carlo method, with a blended method for intermediate H.

Remember that the introduction of any amount of bootstrapping to replace a return will bias the temporal difference update, resulting in a "*semi-gradient*" method, as when we differentiate δ_t^2, we only take the derivate with respect to the current state, not the future state used to bootstrap. This bias is reduced asympotically to 0 when taking $H \to \infty$, but still exists. The bias is also removed once $V_\pi(s) \approx v_\pi(s)$ for all s.

Another method of mixing information from previous time steps is the TD(λ) algorithm, which is an *eligibility trace* method [27], which has good convergence properties [73,

74]. An advantage of these approaches is one can update the parameters in an online fashion, without requiring a full epsiode, or horizon to complete, making them suitable for non-episodic problems.

2.5 Approximate Solution Methods

Throughout the earlier RL sections, we have assumed that we can construct a value or state-action-value function for a problem by storing a number (or many numbers) for every single state of a problem. However, this assumes that we can enumerate the states of a problem. It may be the case that there are an infinite number of states, such as in continuous domains. In these situations, one can group states such that many original states are combined into a single representative "state". This process is sometimes called *binning* when discretising a set of values as you would do when computing a histogram. As an example, imagine a robot arm with some moving parts, whose limb orientation is measured by a set of continuous signals (outputting in degrees). To reduce the state space, we can instead round all the angles to the nearest degree, *discretising* the state-space to make it smaller and more manageable.

Additionally, in extremely large state spaces, it is possible that an agent will frequently encounter states that have not been visited before. By using a simple lookup table for the value function, the agent cannot *generalise* to unseen states and infer what actions are likely to be optimal behaviour. Training these lookup tables, therefore, has to ensure that enough of the state space is covered to be useful, making the training process computationally expensive.

Take an example of the Lunar Lander (found in OpenAI's gym [75], a common benchmark suite of environments for evaluating RL algorithms) game, whereby an agent controls various thrusters on a spacecraft, much like the one which landed on the Moon. The aim is to land the simulated craft safely on the surface. An episode of the game may randomise the surface of the Moon each time, along with the incoming initial speed of the lander, the fuel and maximum power of the thrusters. Even in this relatively simple game, one has many continuous inputs, and a huge variety of possible states and hence an incredibly large space of possibilities, infeasible to solve exactly via tabular methods[5]. An interesting extension of this game to the real world involves using RL algorithms for controlling actual spacecraft to land on the moon using image data [76]. Another example is training a self-driving car [77]. Even with extensive experience, in the real world, there will be scenarios that have not been seen before, and hence, an urgent need for the ability of the agent to generalise to new situations.

We can introduce *supervised learning* techniques to address issues with large (or continuous) state and action spaces, recognising that many states may have very similar values or the same policy, allowing a good function approximation to generalise to unseen states. SL studies various techniques to learn function approximations, based on data, that aim to describe the general function transformation from data to label, which can generalise on unseen data. RL often does not have access to labelled states, except when implementing *imitation learning* [78]. Even with this data, it is unlikely that it is exhaustive

[5]Where tabular methods refers to algorithms where the value and policy functions can be stored exhaustively (tabulated) for all states.

over every possible state, and some learning may be needed to learn a generalised policy. Despite the lack of labelled data, the techniques of function approximation are still incredibly useful in the RL domain, providing the foundations for the bulk of the most recent cutting edge advances in the field. In this section, we will cover the basics of applying function approximation to RL problems.

2.5.1 Large State Spaces and Generalisation

Any problems which have continuous state spaces, or even very large discrete ones, require too much memory to store a tabular value function, or a tabular policy. Not only is the memory requirement massive, but learning the optimal policy by searching through this state space can be infeasible due to computational constraints. We know that dynamic programming is incredibly efficient, requiring only polynomial time to reach a solution [27]; however, with large enough state spaces, this is still too large.

Eventually, one requires some sort of approximate function to step in for the value function (or commonly the state-action-value function), and sometimes the policy as well — this will be discussed in the last section of the chapter. Usually these approximations take the form of a deep neural network, yielding methods under the term deep reinforcement learning (DRL) (see reviews [79] and [80] for a more detailed overview).

2.5.2 The Deadly Triad

Sutton and Barto, Ref. [27], warn of the danger of instability and divergence that arises whenever one combines any of the three elements in RL:

1. **Function approximation**: A scalable and efficient way of generalising a function on a problem with a large state or action space.
2. **Bootstrapping**: An efficient way of updating target values using existing estimates (as in dynamic programming, n-step returns or temporal difference methods).
3. **Off-policy training**: Training on a distribution of transitions other than that produced by the target policy.

Each of these methods have their place, and can often be an essential part of successfully training a RL agent on difficult problems. However, they are a recipe for instability. Most of the cutting edge research into reinforcement learning focuses on heuristics and techniques to help reduce the instability caused by using or combining these methods.

Sutton and Barto in Ref. [27], refrain from commenting on specific solutions, as all are open research questions, and there are no clear strategies which can be used to improve training. However, there are a few techniques that are used commonly, such as replay buffers [81], target value functions (sometimes called *double q-learning* [82]), asynchronous updates [83], policy gradients [84, 85] and planning [71]. As with SL or USL, increasing the amount of data one has access to can also help mitigate some of these problems, which is achieved by efficiently simulating the reinforcement learning environment. Simulating your environments can help you scale up and parallelise the experience gathering process to get more accurate estimates of the quantities described in this thesis.

In the final section of this chapter, we will discuss in detail *policy gradient methods*, which assume some parameterisation (possibly a function approximation) approach to

the agent's policy function, which can be directly optimised via gradient ascent on the cumulative reward.

2.6 Policy Gradient Methods

Until now, we have discussed constructing a policy by calculating the associated value function, and then improving on that policy. This focused entirely on the value function and the estimates. However, some problems can have extremely complicated value functions, but relatively simple policies. If we only care about the value function in so far as it helps us find an optimal policy, it can be advantageous to directly optimise the policy.

Remember that we started with a value function as we could guarantee convergence to the optimal policy through policy evaluation and iteration. However, as we started to move to more complicated methods, introducing function approximation, off-policy learning and bootstrapping techniques, we relaxed any guarantees about optimality that we started with. For the most difficult problems, our aim is only to *attempt* to maximise the expected reward on a given problem, or get as close as possible, without guaranteeing the optimal behaviour.

Our relaxed constraint enables us to devise methods that do not involve calculating the value function, but instead, allow us to directly optimise a policy. From the title of this section "Policy *Gradient* Methods", we are implying that we will be discussing methods which have a differentiable, parameterised policy, such as a function approximation. These techniques are incredibly powerful and can be extended to problems in continuous state and action spaces. However, we will usually only be able to locally optimise the function (exploitation), and we often have to add in additional techniques to ensure exploration.

Policy gradient methods explored extensively in Chapter 6.

2.6.1 Parameterising a Policy

Choosing a policy until now has been fairly simple, relying on a value or state-action-value function to choose the actions. However, we can go directly from a state to a policy. The typical way of creating this mapping is to use a neural network as, in theory, it can represent any function due to its *universal-approximation* trait [86]. NNs[6] are also useful in that they are well studied in supervised learning, and there are many open source libraries which provide ways of constructing and optimising them.

Discrete Action Spaces

If we are trying to represent a policy for an environment which accepts only deterministic actions, then we should ensure the final layer of the neural network is able to output a probability distribution of selecting the available actions. We can have the neural network output a preference for each action, and the relative preferences decide the final probability distribution. In neural networks, we call these preferences *logits* [29]. If a neural network has D hidden layers (not including the input or final output layer), then the Dth layer should have a linear activation function to allow the logit to express any real

[6] See Section 1.3 for a brief overview, or Ref [29] for a more in-depth treatment.

valued preference, together with $(\max_s|\mathcal{A}(s)|)$ outputs (the maximum number of possible actions).

The final layer, $\hat{\boldsymbol{Y}}$, should have a *softmax* activation function. The softmax function can be written as

$$\hat{\boldsymbol{Y}}^{(j)} = \frac{\exp(H_D^{(j)})}{Z}, \tag{2.52}$$

where $x^{(j)}$ represents the jth component of the vector $\boldsymbol{x}$ and Z is the term that normalises the output distribution, allowing the outputs to be interpreted as a probability distribution. We see that the form of Z can be calculated via

$$\begin{aligned} 1 &= \sum_j \hat{\boldsymbol{Y}}^{(j)} \\ 1 &= \sum_j \frac{\exp(H_D^{(j)})}{Z} \\ Z &= \sum_j \exp(H_D^{(j)}). \end{aligned} \tag{2.53}$$

This can be greatly simplified in the binary case as we need only a single preference value to indicate the probability; we use the *logistic sigmoid* function, defined as

$$\sigma(x) = \frac{1}{1+e^{-x}}, \tag{2.54}$$

which has the properties that $\sigma(-x) = (1-\sigma(x))$. We interpret the input as the preference for the binary output being true. If $x = 0$ then there is an equal chance of choosing either action. Notice that when $x \to \infty$ then $\sigma(x) \to 1$ and when $x \to -\infty$ then $\sigma(x) \to 0$. The logistic sigmoid is therefore a valid probability distribution, bounding the probabilities between 0 and 1.

Forcing a softmax policy ensures that the agent will take all actions with some finite probability, ensuring exploration of the state space, while allowing for a smooth continuous gradient to be calculated. Eventually, as the agent trains, it can decrease the probability of taking known poor actions and focus on exploiting the most relevant area of the state space.

We can additionally use an off-policy method (such as ϵ-*greedy*) with the appropriate importance sampling on any expectation value over the trajectories. Since we have a soft policy in both cases, we will always get a well-defined importance sampling ratio, leading to better stability, while also providing exploration if needed during training.

Continuous Action Spaces

As we are directly parameterising the policy, we introduce the idea of continuous actions. These are common in practice, e.g. a robot trying to walk needs to tell each of the servo motors the angle that it should move towards. As before, one can discretise the space of continuous actions and use the method in the previous section, however, we are not limited by this method.

One method for directly choosing continuous actions is for the network to output the mean and variance of a normal distribution which is sampled to choose the actual continuous

action. The reason that we make the network output a probability distribution is that we can efficiently differentiate this probability distribution to update the weights of a network, along with introducing the exploration that comes hand-in-hand with a soft policy.

As a concrete example, we will use a linear network for a policy, which chooses the characteristic values for a normal distribution, which in turn samples the continuous actions for the policy. We can then see that the probability density of the policy, π, can be written as

$$\pi_\theta(a|s) = \frac{1}{\sigma_\theta(s)\sqrt{2\pi}} \exp\left(-\frac{1}{2}\frac{(a-\mu_\theta(s))^2}{\sigma_\theta(s)^2}\right), \tag{2.55}$$

where $\mu_\theta(s)$ and $\sigma_\theta(s)$ are the parameterised mean and standard deviation of the normal distribution, which describes the policy in the state s. As before, we represent the parameters of the model with θ.

In a continuous action space, it is known that the probability of selecting an exact action, a, is 0 for any action a. Instead, we can ask what is the probability that an action is chosen between two limits a_1 and a_2, such that $a_1 \leq a \leq a_2$, which is given by

$$\Pr\left[a_1 \leq a \leq a_2 | s\right] = \int_{a_1}^{a_2} \pi(a|s) da. \tag{2.56}$$

As a shorthand when working with continuous action spaces, we tend to use $\pi(a|s)$ to denote the probability density function of the policy.

Differentiating a Gaussian

We can differentiate the policy w.r.t the parameterised mean μ_θ, giving

$$\frac{\partial \pi_\theta(a|s)}{\partial \mu_\theta(s)} = \frac{a-\mu_\theta(s)}{\sigma_\theta(s)^2}\pi_\theta(a|s). \tag{2.57}$$

We can write a similar equation for the variance, which we calculate to be

$$\frac{\partial \pi_\theta(a|s)}{\partial \sigma_\theta(s)} = \frac{(a-\mu_\theta(s))^2 - \sigma_\theta(s)^2}{\sigma_\theta(s)^3}\pi_\theta(a|s). \tag{2.58}$$

Notice that we only calculated the partial derivatives. In general, our full derivative, w.r.t the parameters θ, is given by

$$\frac{d\pi_\theta(a|s)}{d\theta} = \frac{\partial \pi_\theta(a|s)}{\partial \mu_\theta(s)}\frac{d\mu_\theta(s)}{d\theta} + \frac{\partial \pi_\theta(a|s)}{\partial \sigma_\theta(s)}\frac{d\sigma_\theta(s)}{d\theta}. \tag{2.59}$$

The equations eq. (2.57) and eq. (2.58) are manually used to calculate the derivative, which can be used as part of the backpropagation routine provided by a chosen deep learning package like `PyTorch` [87] or `TensorFlow` [88].

2.6.2 Policy Gradient: REINFORCE

Now that we have discussed ways to parameterise a policy such that it is differentiable, we choose a loss function to optimise. The most sensible loss function to optimise is tied to the expected return. We follow the REINFORCE algorithm from Williams [84].

In the episodic case, we can define this as

$$\mathcal{L}(\theta) = \mathop{\mathbb{E}}_{\omega\sim\pi}\left[\sum_{t=0}^{T(\omega)} \gamma^t R_t\right], \tag{2.60}$$

where $T(\omega)$ is the length of the trajectory ω, such that $S_{T(\omega)}$ is the terminal state. As $R_{T(\omega)} = 0$ by definitition of the terminal state, we can ignore this time step in future summations. However, in the return form, it is unclear how the policy affects the loss. Alternatively, one can instead write

$$\mathcal{L}(\theta) = \sum_{\omega} p(\omega)\left[\sum_{t=0}^{T(\omega)-1} \gamma^t R_t\right], \tag{2.61}$$

wherein $p(\omega)$ can be expanded into

$$p(\omega) = d(S_0)\prod_{t=0}^{T(\omega)-1} \pi_\theta(A_t|S_t)p(S_{t+1}, R_t|A_t, S_t) \tag{2.62}$$

as usual. Notice here that $p(\omega)$ explicitly depends on θ, but only in the terms $\pi_\theta(A_t|S_t)$ and no other terms.

Further, we derive the derivative of our loss function w.r.t the parameters θ, resulting in

$$\begin{aligned}\nabla_\theta\mathcal{L}(\theta) &= \sum_{\omega}\nabla_\theta\left[p(\omega)\sum_{t=0}^{T(\omega)-1} \gamma^t R_t\right] \\ &= \sum_{\omega}\left[\sum_{t=0}^{T(\omega)-1} \gamma^t R_t\right]\nabla_\theta p(\omega). \end{aligned}\tag{2.63}$$

Here, we are using the notation

$$\nabla_\theta = [\frac{\partial}{\partial\theta_1}, \frac{\partial}{\partial\theta_2}, \dots, \frac{\partial}{\partial\theta_n}], \tag{2.64}$$

instead of $\frac{d}{d\theta}$, as θ is usually a vector of n parameters.

We know that the differential of the total discounted reward is not dependent on θ as these are just random variable values. The only dependence is on $p(\omega)$. We can explicitly work out the derivative of this probability,

$$\begin{aligned}\nabla_\theta p(\omega) &= \nabla_\theta d(S_0)\prod_{t=0}^{T(\omega)-1} \pi_\theta(A_t|S_t)p(S_{t+1}, R_t|A_t, S_t) \\ &= \left[d(S_0)\prod_{t'=0}^{T(\omega)-1} p(S_{t'+1}, R_{t'}|A_{t'}, S_{t'})\right]\nabla_\theta\prod_{t=0}^{T(\omega)-1}\pi_\theta(A_t|S_t), \end{aligned}\tag{2.65}$$

where we have moved all terms independent of θ to the left, so we can focus on the differential. Here, we use the product rule on the product of probabilities

$$\nabla_\theta\prod_{t=0}^{T(\omega)-1}\pi_\theta(A_t|S_t) = \left[\prod_{t''=0}^{T(\omega)-1}\pi_\theta(A_{t''}|S_{t''})\right]\sum_{t=0}^{T(\omega)-1}\frac{1}{\pi_\theta(A_t|S_t)}\nabla_\theta\pi_\theta(A_t|S_t). \tag{2.66}$$

We can use the fact that

$$\nabla_\theta \log \pi_\theta(a|s) = \frac{1}{\pi_\theta(a|s)} \nabla_\theta \pi_\theta(a|s) \tag{2.67}$$

to simplify eq. (2.66), such that

$$\nabla_\theta \prod_{t=0}^{T(\omega)-1} \pi_\theta(A_t|S_t) = \left[\prod_{t''=0}^{T(\omega)-1} \pi_\theta(A_{t''}|S_{t''})\right] \sum_{t=0}^{T(\omega)-1} \nabla_\theta \log \pi_\theta(a|s). \tag{2.68}$$

Fortunately, we can substitute this all back into eq. (2.65), combining all product terms to recover $p(\omega)$ at the front, leaving

$$\nabla_\theta p(\omega) = p(\omega) \sum_{t=0}^{T(\omega)-1} \nabla_\theta \log \pi_\theta(a_t|s_t). \tag{2.69}$$

Finally, this can be put back into eq. (2.63) to give

$$\begin{aligned}
\nabla_\theta \mathcal{L}(\theta) &= \sum_\omega p(\omega) \left[\sum_{t=0}^{T(\omega)-1} \gamma^t R_t\right] \times \left[\sum_{t'=0}^{T(\omega)-1} \nabla_\theta \log \pi_\theta(a_{t'}|s_{t'})\right] \\
&= \mathop{\mathbb{E}}_{\omega\sim\pi} \left[\left(\sum_{t=0}^{T(\omega)-1} \gamma^t R_t\right) \times \left(\sum_{t'=0}^{T(\omega)-1} \nabla_\theta \log \pi_\theta(a_{t'}|s_{t'})\right)\right] \\
&= \mathop{\mathbb{E}}_{\omega\sim\pi} \left[G_0(\omega) \times \left(\sum_{t=0}^{T(\omega)-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\right)\right],
\end{aligned} \tag{2.70}$$

where $G_0(\omega)$ represents the total discounted return from the initial state. Notice that the gradient is defined using expectation value over trajectories. This makes an estimate easy to construct by just sampling trajectories and averaging the gradient contributions from each.

In DL libraries, which provide automatic differentiation capabilities, one usually represents the loss function as

$$L(\theta) = \sum_{\omega\sim\pi} A(\omega) \sum_{S_t, A_t \in \omega} \log \pi_\theta(a_t|s_t), \tag{2.71}$$

where $A(\omega)$ is a weighting function that depends on the trajectory. In the case we have seen, we can use $G_0(\omega) = A(\omega)$ to perform vanilla REINFORCE. As we will learn in later sections, this number $A(\omega)$ can be constructed in many ways. This loss function is quantitatively different from the total expected reward; however, it has the same gradient which can be used to update the weights θ. One usually estimates the gradient using a *minibatch*, consisting of a small number of trajectories reminiscent of stochastic gradient descent.

2.6.3 Variance of Vanilla Policy Gradient

In policy gradient methods, we often talk about methods to "reduce the variance" of the gradient estimate, as a lower variance of estimates allows safe use of a high learning rate,

and hence accelerates the learning process. One estimate of the unbiased gradient can be done via MC, for example

$$\nabla_\theta \mathcal{L}(\theta) \approx \nabla_\theta L(\theta) = \frac{1}{N} \sum_i G(\omega^{(i)}) \left(\sum_{t=0}^{T(\omega_i)-1} \nabla_\theta \log \pi_\theta(A_t^{(i)} | S_t^{(i)}) \right), \tag{2.72}$$

where each trajectory, $\omega^{(i)}$, is sampled using π with N total samples. We can also use a single trajectory to try and estimate the gradient. The estimate of the gradient using one trajectory ω can be written as

$$\delta L = \sum_{t=0}^{T(\omega)-1} G(\omega) \nabla_\theta \log \pi_\theta(A_t | S_t). \tag{2.73}$$

As our estimate in eq. (2.72) is just a weighted mean over δL, we know that the variance should follow

$$\mathrm{Var}(\nabla_\theta L) = \frac{\mathrm{Var}(\delta L)}{N}, \tag{2.74}$$

where $\mathrm{Var}(X)$ denotes the variance of the random variable X. We can calculate the variance with the formula

$$\mathrm{Var}(\delta L) = \mathop{\mathbb{E}}_{\omega \sim \pi} \left[(\delta L)^2 \right] - \mathop{\mathbb{E}}_{\omega \sim \pi} \left[\delta L \right]^2. \tag{2.75}$$

However, we can just as easily sample a set of δL and plot a histogram to visually show the variance in the form of a wider distribution.

Linear Model

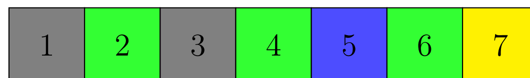

Figure 2.9: A simple 1-dimensional grid world environment in which the agent always starts in the left-most state — (1).

Let us look at a simple grid world environment with the map shown in fig. 2.9, which has the agent starting in the left-most state every time, and can only move east or west. Additionally, we only let the agent walk for a maximum of 20 steps and set $\gamma = 1$. Note that enforcing a maximum time step of 20 introduces time as a factor in the state; however, we will only take this into account when calculating the true value functions.

To analyse the various policy gradient methods later in this section, we will construct a policy approximation for demonstrations by using a linear function approximation. We can encode the input state as $\tilde{\boldsymbol{s}} = [x, 1]$, which is a column vector with size 2 by 1. Appending the constant 1 to the state allows us to encode a bias vector directly into the weight matrix. We encode the weight vector as $W_\theta = [\theta_1, \theta_2]$, such that

$$\pi_\theta(\rightarrow | s) = \sigma({W_\theta}^T \tilde{\boldsymbol{s}}) \tag{2.76}$$
$$\pi_\theta(\leftarrow | s) = \sigma(-{W_\theta}^T \tilde{\boldsymbol{s}}). \tag{2.77}$$

This ensures that the probabilities are normalised as $1 - \sigma(x) = \sigma(-x)$. We encode the actions of $(\leftarrow, \rightarrow)$ to be $(-1, +1)$ respectively, such that the environment update is simply

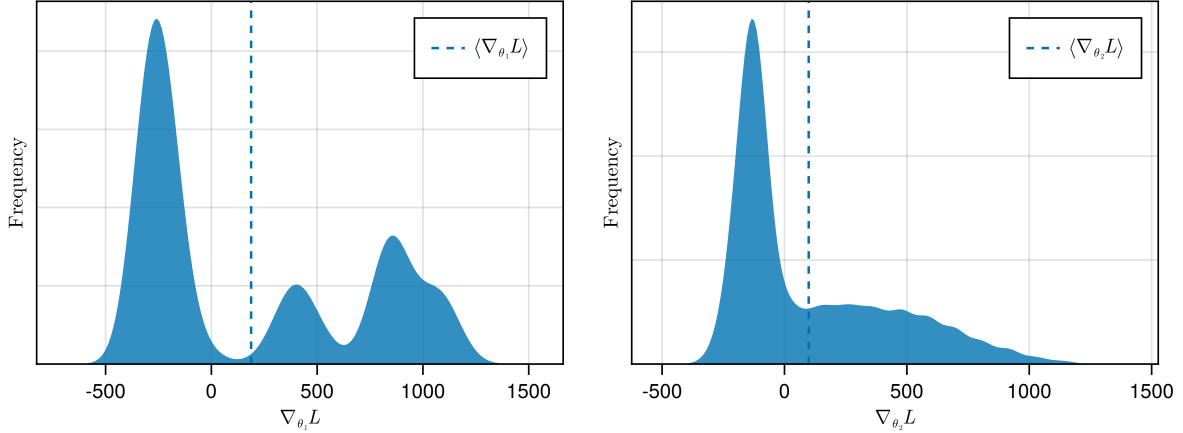

Figure 2.10: The probability density of the gradient estimates for θ_1 (left) and θ_2 (right). Both estimates will have a very high variance, as seen by the wide distributions; however, the means are accurate and unbiased. Averaging many estimates will lead to an accurate, unbiased, estimate of the true gradient. The PDFs (Probability Density Functions) are estimated by sampling 10^4 trajectories on the problem shown in fig. 2.9.

$f(x, a) = x + a$. Also, when we are at the left-most side, the probability is deterministic, always going to the right, which will have 0 gradient.

Remember that our gradient format requires evaluating $\nabla_\theta \log \pi_\theta(a|s)$, whose derivation is shown in eq. (2.78).

$$\begin{aligned}
\nabla_\theta \log \pi_\theta(a|s) &= \nabla_\theta \log \sigma(a{W_\theta}^T \tilde{\boldsymbol{s}}) \\
&= \frac{1}{\sigma(a{W_\theta}^T \tilde{\boldsymbol{s}})} \nabla_\theta \sigma(a{W_\theta}^T \tilde{\boldsymbol{s}}) \\
&= \frac{1}{\cancel{\sigma(a{W_\theta}^T \tilde{\boldsymbol{s}})}} (1 - \sigma(a{W_\theta}^T \tilde{\boldsymbol{s}}))\cancel{\sigma(a{W_\theta}^T \tilde{\boldsymbol{s}})} \nabla_\theta (a{W_\theta}^T \tilde{\boldsymbol{s}}) \\
&= (a\tilde{\boldsymbol{s}}) \times \sigma(-a{W_\theta}^T \tilde{\boldsymbol{s}}) \\
&= a\pi_\theta(-a|s)\tilde{\boldsymbol{s}}.
\end{aligned} \tag{2.78}$$

Variance of Linear Model

We can calculate an estimate for the REINFORCE gradient, substituting eq. (2.78) in for $\nabla_\theta \log \pi_\theta(a|s)$ to get an expression for the gradient of the loss, calculated as

$$\begin{aligned}
\nabla_\theta \mathcal{L}(\theta) &= \mathop{\mathbb{E}}_{\omega \sim \pi} [\nabla_\theta l(\omega)] \\
&= \mathop{\mathbb{E}}_{\omega \sim \pi} \left[G_0(\omega) \sum_{t=0}^{T(\omega)-1} A_t \pi_\theta(-A_t|S_t) \begin{bmatrix} X_t \\ 1 \end{bmatrix} \right].
\end{aligned} \tag{2.79}$$

Using eq. (2.79), we calculate the gradient contributions for a single trajectory. These gradient measurements are random variables. We observe their distribution by plotting the approximate probability density function, as shown in fig. 2.10.

We can see that if we take only a single example, we are very unlikely to be close to the actual mean. We can only obtain an estimate close to mean by averaging many independent samples. Having an accurate gradient allows us to take fewer steps to converge to a locally optimal policy as we can safely increase the step-size. Techniques that require few samples to determine the mean value are known as "low-variance". Low variance techniques are efficient as one can sample fewer trajectories at each update, or choose to make larger changes to the parameters, as the gradient estimate will be more accurate. In the next few examples, we will discuss ways of improving our estimate of the gradient by lowering the variance.

Removing the Past

In our basic estimate of the gradient in eq. (2.70), we notice that each term in the sum is multiplied by the same prefactor, $G_0(\omega)$. The other part of the term inside the sum over t is the $\nabla_\theta \log \pi_\theta(A_t|S_t)$ part, which can be interpreted as moving in the direction of increasing the likelihood of choosing the action A_t in the state S_t if multiplied by a positive number. However, we formulate all of our problems as MDPs, which means that the history of a trajectory is not relevant to the future, only the current state is. Since we are dealing with MDPs, it does not make sense that the prefactor for each of the terms should include information from the *past*, before state S_t was reached. Fortunately, our estimate of the gradient is invariant under a special transformation, allowing us to remove the past.

Take the expectation over $a \in \mathcal{A}(s)$ of the quantity $\beta(s)\nabla_\theta \log \pi_\theta(a|s)$, for any function β that only depends on the state s. We calculate this expectation as

$$\begin{aligned}
\sum_{a\in\mathcal{A}(s)} \pi_\theta(a|s)\beta(s)\nabla_\theta \log \pi_\theta(a|s) &= \beta(s) \sum_{a\in\mathcal{A}(s)} \pi_\theta(a|s)\nabla_\theta \log \pi_\theta(a|s) \\
&= \beta(s) \sum_{a\in\mathcal{A}(s)} \cancel{\pi_\theta(a|s)} \frac{1}{\cancel{\pi_\theta(a|s)}} \nabla_\theta \pi_\theta(a|s) \\
&= \beta(s) \sum_{a\in\mathcal{A}(s)} \nabla_\theta \pi_\theta(a|s) \\
&= \beta(s)\nabla_\theta \sum_{a\in\mathcal{A}(s)} \pi_\theta(a|s) \\
&= \beta(s)\nabla_\theta 1 \\
&= 0,
\end{aligned} \tag{2.80}$$

where we used the fact that $\pi_\theta(a|s)$ is a normalised probability distribution which sums to 1, a constant, whose gradient is 0. We can interpret this fact by saying that whenever we perform an expectation over the gradient of the log probabilities, multiplied by a function which does not depend on the action a, but only on the state s, the resulting expectation is zero. Said differently, our expectation value is *invariant* under additions of arbitrary functions $\beta(s)$ multiplying $\nabla_\theta \log \pi_\theta(a|s)$. We write the new gradient estimate in eq. (2.81).

$$\nabla_\theta \mathcal{L}(\theta) = \mathop{\mathbb{E}}_{\omega\sim\pi}\left[\sum_{t=0}^{T(\omega)-1} [G_0(\omega) - \beta(s)]\nabla_\theta \log \pi_\theta(a_t|s_t)\right]. \tag{2.81}$$

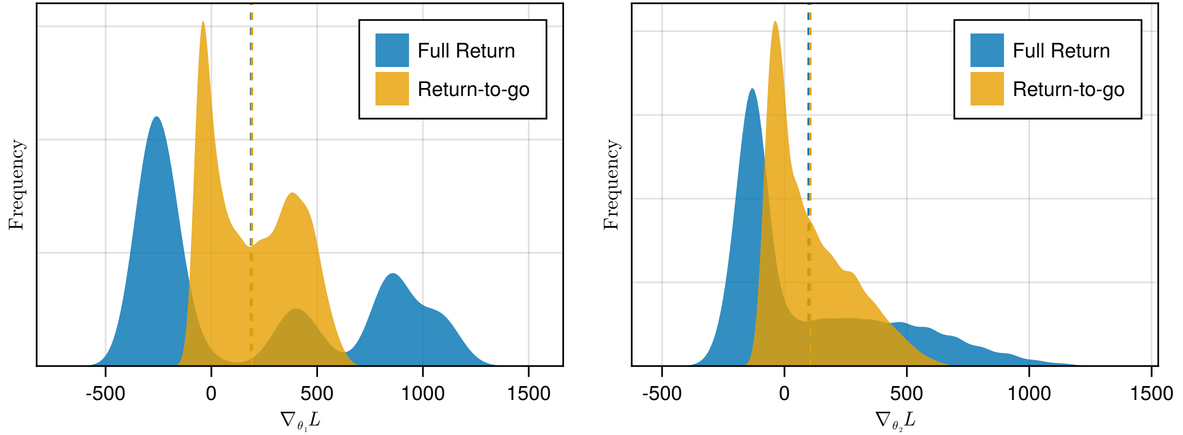

Figure 2.11: The probability density of the gradient estimates for θ_1 (left) and θ_2 (right) compared between using eq. (2.70), shown in blue, and eq. (2.83), shown in orange. Removing the past from the rewards changes the distribution of the gradients for each parameter, but does not change the mean of the distribution. Additionally, the variance is much reduced by removing the past, allowing one to obtain a more accurate gradient estimate with fewer trajectory samples.

This form of the estimate is commonly called the baseline, and there are many forms which the function β can take. $\beta(s)$ can even depend on the previous part of the trajectory, before t, without affecting the expected value. One popular choice for our baseline function is

$$\beta(s, t)_{\text{past}} = \sum_{t'=0}^{t-1} \gamma^{t'} R_{t'}, \tag{2.82}$$

which is simply just the first part of the discounted reward totalled up to just before time t. This turns our prefactor into simply $\sum_{t'=t}^{T(\omega)-1} \gamma^{t'} R_{t'}$, which is the *return-to-go* from state S_t, multiplied by a discount factor of γ^t.

Now, we have removed the past from our equation, getting an estimate using samples from

$$\nabla_\theta \mathcal{L}(\theta) = \mathop{\mathbb{E}}_{\omega \sim \pi} \left[\sum_{t=0}^{T(\omega)-1} \left(\sum_{t'=t}^{T(\omega)-1} \gamma^{t'} R_{t'} \right) \nabla_\theta \log \pi_\theta(a_t | s_t) \right]. \tag{2.83}$$

Removing the past leaves the gradient estimate unbiased. However, it makes a huge difference to the variance of the estimate. Looking at fig. 2.11, we can see that this alteration significantly reduces the variance of our estimate of the gradient for both parameters, while leaving the mean unchanged. Again, this means that we can construct an accurate (and unbiased) estimate for the gradient with very few trajectories.

We continue this section by returning to this idea of variance reduction, as it is the most important technique for increasing learning efficiency when using policy gradient methods.

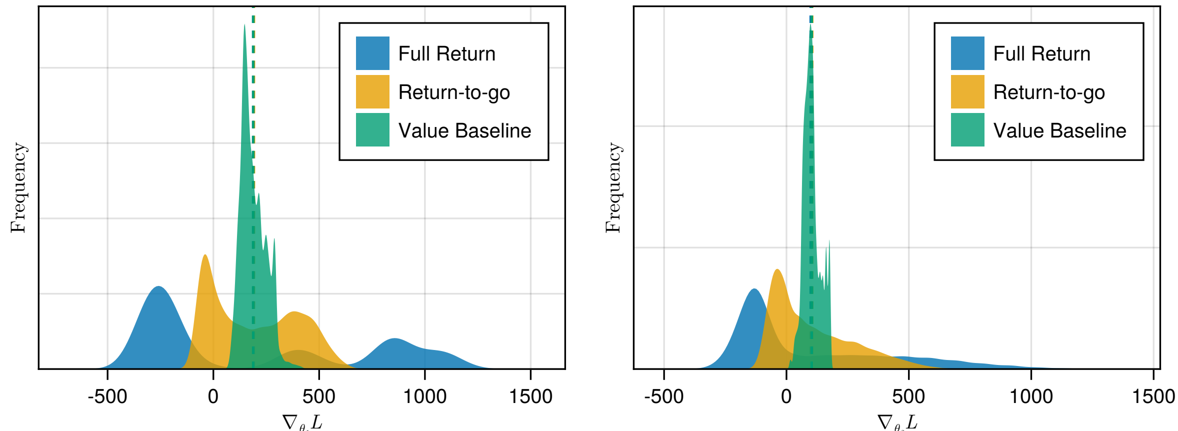

Figure 2.12: The probability density of the gradient estimates for θ_1 (left) and θ_2 (right) compared between using eq. (2.70), shown in blue, and eq. (2.83), shown in orange. In green, we see the effect of using a value baseline using eq. (2.84) with eq. (2.86). Adding a value baseline on top of removing the history significantly reduces the variance of the estimates, but the mean is largely unaffected. Value function was estimated using 10^4 trajectories for illustration purposes.

2.6.4 REINFORCE with Value Baseline

From this point forward, we will use eq. (2.83) (removing the past) as our starting point. We now introduce the concept of the *advantage*, written as χ_t. The advantage enters the gradient equation as

$$\nabla_\theta \mathcal{L}(\theta) = \mathop{\mathbb{E}}_{\omega \sim \pi} \left[\sum_{t=0}^{T(\omega)-1} \gamma^t \chi_t \nabla_\theta \log \pi_\theta(a_t|s_t) \right]. \tag{2.84}$$

Our standard algorithm sets $\chi_t = G_t$, where G_t is the *return-to-go* for the trajectory ω, which is defined as

$$G_t = \sum_{t'=t}^{T(\omega)-1} \gamma^{t'-t} R_{t'}. \tag{2.85}$$

However, we know that we are free to offset χ_t by any quantity that only depends on the current state, or that occurs before t, without changing the estimate. Another popular choice for this quantity comes from subtracting the value of the state S_t under the current policy from G_t, such that

$$\chi_t = G_t - v_\pi(S_t), \tag{2.86}$$

which is where the name *advantage* comes from: since the value $v_\pi(S_t)$ is the average of G_t from that state, this quantity measures the advantage of taking the actions that led to G_t when compared with the average behaviour. The only issue here is that $v_\pi(S_t)$ is the value function of our policy, which is often unknown. However, using an estimate V_π instead will not bias your estimate, but may increase the variance if V_π is inaccurate. However, one can find that the more accurate V_π, the better it is at reducing the variance of your estimate [27]. As seen in fig. 2.12, using the true value function, the variance in our simple problem is greatly reduced.

2.6.5 Actor-Critic

In the previous methods, we still have to generate entire trajectories, which can take a very long time. Instead, it may be beneficial to be able to have a method that can learn from single transitions, so the methods can be effectively ported to non-episodic problems. Instead of generating an entire trajectory, we can instead sample single transitions and estimate the temporal difference in value between the two states. We can substitute our expression for G_t with a single step evaluation ($R_t + v_\pi(S_t)$) to get

$$\chi_t = R_t + \gamma v_\pi(S_t) - v_\pi(S_t), \tag{2.87}$$

which we call the *temporal-difference* error. When our estimate of v_π is not correct, this method introduces a bias into the gradient estimation. If our estimate of v_π becomes more and more accurate, then this bias will asymptotically approach zero. We can accept this bias, as this method can further reduce the variance from the baseline. The reason this method is called *Actor-Critic*, is because we have our policy that takes actions in the environment (the actor), and a separate function approximation for v_π, which we call the *critic*. We usually have a separate loss for the critic which aims to minimise loss given in eq. (2.88).

$$L(\phi) = \mathop{\mathbb{E}}_{\omega\sim\pi}\left[\frac{1}{2}\sum_{t=0} T(\omega) - 1(\gamma V_\phi(S_t) - v_\phi(S_t))^2\right] \tag{2.88}$$

Equation (2.88) approximates v_π by using a function approximation V_ϕ with parameters ϕ.

We can see the effect of using actor critic in fig. 2.13, where the method caused a higher variance in comparison to the standard value-baseline method. However, we should note that we are calculating the gradient for an entire trajectory. Instead, Actor-Critic can be used to calculate gradients for single transitions, making it much more effective when training on continuing problems, whereas standard value-baseline, which uses Monte Carlo returns cannot create an estimate from a single transition.

Actor-Critic is a type of bootstrapping technique, which can be used to increase the computational efficiency of the learning algorithms, but at the cost of introducing some bias. It is also important to realise that the value function needs to be quite accurate for the bias to be reduced, so if the value function is too complex, then actor critic methods may not converge depending on the function approximation used for the state. In contrast, value-baseline methods only affect the variance of the gradient estimate, not the mean, and therefore, even if the value function is too complex to approximate exactly, no bias will be introduced. The only reason we did not see any introduced bias when using Actor-Critic in fig. 2.13, is because we used an exact numerical evaluation of v_π as the critic.

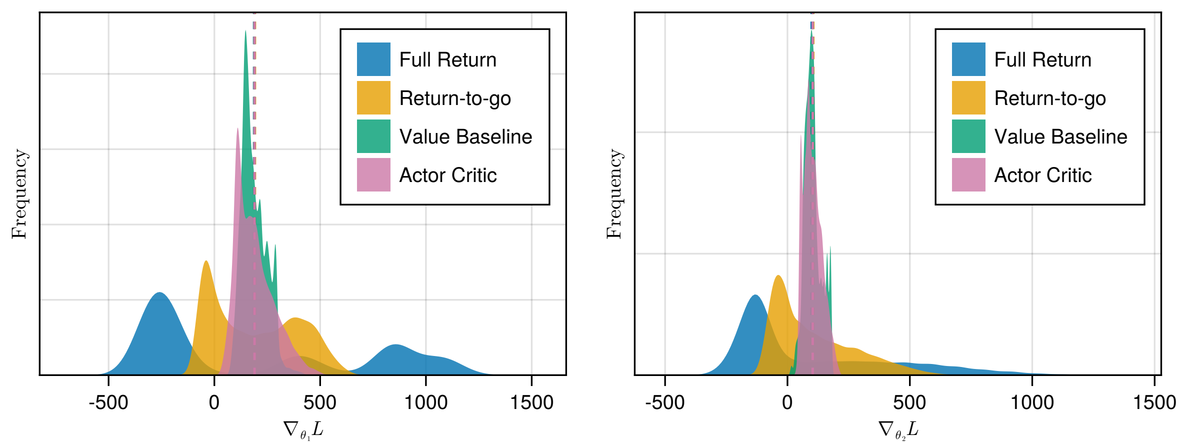

Figure 2.13: The probability density of the gradient estimates for θ_1 (left) and θ_2 (right), similar to fig. 2.12, with the addition of the Actor-Critic variance in pink. We can see that there is no bias when using an accurate value function. The variance for the trajectory is actually higher than just using a value baseline. However, this method allows one to sample single transitions from the trajectory, which can vastly reduce variance in online and continuing problems.

Chapter 3

Large Deviation Theory

In this chapter, we describe the physics background that binds all research presented in this thesis together: large deviation theory (LDT). While this chapter is not intended as a comprehensive overview of the subject, we will cover the essentials for understanding the background and motivations of Chapter 6, Chapter 7 and Chapter 8. For a more detailed reviews of LDT and its application to statistical physics, see Refs [89–92].

As a starting point, we shall consider a simple problem: quantifying the probability of outcomes of a series of coin flips. We can label a single coin flip as a random variable X which has a probability of c for heads and $1-c$ for tails. For now, we do not assume that the coin is fair and c can take any value between 0 and 1 inclusive. We can perform n coin flips and count the empirical frequency of heads with the random variable

$$F_n = \frac{1}{n}\sum_{i}^{n} X_i, \tag{3.1}$$

where X_i is a series of independent coin flips (all using the same coin). Using the binomial distribution we can quantify the probability that $F_n = f$, where f can take values of $0, \frac{1}{n}, \frac{2}{n}, \ldots, 1$, denoted as

$$P(F_n = f) = \frac{n!}{(fn)!(n-fn)!} c^{fn}(1-c)^{n-fn}. \tag{3.2}$$

We can ask what form this probability takes when n is large. For this, we can use Stirling's approximation for a factorial ($\ln n! \approx n\ln n - n$) to arrive at the result

$$P(F_n = f) \approx \exp\left(-n\left[f\ln\frac{f}{c} + (1-f)\ln\frac{1-f}{1-c}\right]\right). \tag{3.3}$$

The resulting probability can therefore be expressed as

$$P(F_n = f) \approx e^{-nI(f)}, \tag{3.4}$$

where

$$I(f) = f\ln\frac{f}{c} + (1-f)\ln\frac{1-f}{1-c}. \tag{3.5}$$

This means that the probability exponentially decays in n according to some rate given by $I(f)$, which is a constant with respect to n. Hence, we call $I(f)$ the **rate function**. We can see that if we set $f = c$, then the rate function vanishes, and hence, the

probability does not decay with increasing n, whereas all other values ($f \neq c$) decay exponentially. This recovers the strong law of large numbers, but we have also characterised the likelihood of measuring values of F_n away from c, for large n.

We say that the probability of F_n follows a large deviation principle (LDP) when it has the form given in eq. (3.4). While eq. (3.4) is only given as an approximate value, we can concretely define a probability to follow the LDP if the following is true [90]

$$\lim_{n\to\infty} -\frac{1}{n}\ln p(A_n = a) = I(a), \tag{3.6}$$

where $I(a)$ is a rate function that only contains terms that are sublinear in n. The notation that is commonly used to express the probability is

$$P(A_n = a) \asymp e^{-nI(a)}, \tag{3.7}$$

where $\asymp$ reflects the log asymptotically approaching in the limit $n \to \infty$. If $a_n \asymp b_n$ then

$$\lim_{n\to\infty} \frac{1}{n}\ln(a_n) = \lim_{n\to\infty} \frac{1}{n}\ln(b_n). \tag{3.8}$$

3.1 Gärtner-Ellis Theorem

Here, we shall present a useful result from LDT that helps us to identify random variables which follow a LDP. We can start with the definition of the scaled cumulant generating function (SCGF) of a random variable, A_n, parameterised by a non-negative integer, n, given by

$$\lambda(k) = \lim_{n\to\infty} \frac{1}{n}\ln\left\langle e^{nkA_n}\right\rangle, \tag{3.9}$$

where

$$\left\langle e^{nkA_n}\right\rangle = \int_{\mathbb{R}} da\; e^{nka} p(A_n \in [a, a+da]), \tag{3.10}$$

where the notation $p(A_n \in [a, a+da])$ is a PDF, indicating a probability of A_n falling between a and $a + da$.

If the SCGF exists and is differentiable for all $k \in \mathbb{R}$, then the Gärtner-Ellis (GE) theorem states that

1. A_n satisfies the large deviation principle $\therefore$ $P(A_n \in [a, a+da]) \asymp e^{-nI(a)}da$.
2. The rate function is given by the Legendre-Fenchel transform [93] of $\lambda(k)$, i.e.

$$I(a) = \sup_{k\in\mathbb{R}} \left\{ka - \lambda(k)\right\}. \tag{3.11}$$

Two notes should be made: if this method fails, it does not mean that A_n does not satisfy a LDP; second, that the transform is only valid when the rate function and SCGF are convex [90].

One can also obtain the SCGF using a rate function via another Legendre-Fenchel transform, as shown in eq. (3.12).

$$\lambda(k) = \sup_{a\in\mathbb{R}} \left\{ka - I(a)\right\}. \tag{3.12}$$

3.2 Moments of the SCGF

First, notice that when $k = 0$, the definition of $\lambda(0)$ is

$$\begin{aligned}\lambda(0) &= \lim_{n\to\infty} \frac{1}{n} \ln \left\langle e^0 \right\rangle \\ &= \lim_{n\to\infty} \frac{1}{n} \ln 1 \\ &= 0.\end{aligned} \tag{3.13}$$

To extend this, the expression for the derivative of λ w.r.t k is

$$\begin{aligned}\lambda'(0) &= \lim_{n\to\infty} \frac{1}{\cancel{n}} \frac{\left\langle \cancel{n} A_n e^{nkA_n} \right\rangle}{\left\langle e^{nkA_n} \right\rangle}\Bigg|_{k=0} \\ &= \lim_{n\to\infty} \frac{\langle A_n \rangle}{\langle 1 \rangle} \\ &= \langle A_n \rangle ,\end{aligned} \tag{3.14}$$

which means that $\lambda'(0)$ gives the mean of a random variable A_n, provided that $\lambda'(0)$ exists. Further, we can differentiate again

$$\begin{aligned}\lambda''(0) &= \lim_{n\to\infty} n \left(\frac{\left\langle A_n^2 e^{nkA_n} \right\rangle}{\left\langle e^{nkA_n} \right\rangle} - \frac{\left\langle A_n e^{nkA_n} \right\rangle^2}{\left\langle e^{nkA_n} \right\rangle^2} \right)\Bigg|_{k=0} \\ &= \lim_{n\to\infty} n \left(\left\langle A_n^2 \right\rangle - \langle A_n \rangle^2 \right) \\ &= \lim_{n\to\infty} n \mathrm{Var}(A_n),\end{aligned} \tag{3.15}$$

which means that if A_n is a sample mean of a variable X (*à la* eq. (3.1)), we can deduce that $\lambda''(0) = \mathrm{Var}(X)$. Hence, if we have an expression for $\lambda(k)$ in terms of k, one can use the above expressions to easily calculate the means and variances of a random variable.

3.2.1 Example: Bernoulli Random Variables

At the beginning of this chapter, we introduced the idea of measuring the frequency of heads or tails of a (possibly) weighted coin, which comes up heads with probability c. This type of random variable is a *Bernoulli* random variable [94], as it has a binary outcome with some fixed distribution. Here, we will derive $\lambda(k)$ and find its moments using eq. (3.14) and eq. (3.15).

To start with, let us define the discrete probability using a continuous function, such that

$$p(X)dX = (1-c)\delta(X)dX + c\delta(1-X)dX. \tag{3.16}$$

We use $\delta(x)$ to mean the delta function (see [67] for definition and properties). In order to work out $\lambda(k)$, we first need to calculate

$$\left\langle e^{nkF_n} \right\rangle = \int_{f\in\mathbb{R}} df p(F_n = f) e^{nkf}. \tag{3.17}$$

As $p(F_n = f) = \prod_{i=1}^n p(X_i = x_i)$, we can rewrite the integral as

$$\langle e^{nkF_n} \rangle = \int_{x_1} \int_{x_2} \dots \int_{x_n} \prod_{i=1}^{n} dx_i \; p(X_i = x_i) e^{kx_i}. \tag{3.18}$$

As each random variable X_i is independent of one another, we can move the product outside and calculate a single integral, yielding

$$\begin{aligned} \langle e^{nkF_n} \rangle &= \prod_{i=1}^{n} \int_{x_i \in \mathbb{R}} dx_i \; p(X_i = x_i) e^{kx_i} \\ &= \prod_{i=1}^{n} \int_{x_i \in \mathbb{R}} dx_i \; e^{kx_i} \left[(1-c)\delta(x_i) + c\delta(1-x_i) \right] \\ &= \prod_{i=1}^{n} \left[(1-c)e^{k\times 0} + ce^{k\times 1} \right] \\ &= \prod_{i=1}^{n} \left[(1-c) + ce^{k} \right] \\ &= \left[(1-c) + ce^{k} \right]^{n}. \end{aligned} \tag{3.19}$$

We plug this expression directly into eq. (3.9) to get

$$\lambda(k) = \ln \left[(1-c) + ce^k \right], \tag{3.20}$$

which expectedly satisfies $\lambda(0) = 0$. From here, we can derive $\lambda'(k)$ and $\lambda''(k)$ as

$$\lambda'(k) = \frac{ce^k}{(1-c) + ce^k}, \tag{3.21}$$

$$\lambda''(k) = \frac{ce^k}{(1-c) + ce^k} - \frac{c^2 e^{2k}}{\left((1-c) + ce^k\right)^2}, \tag{3.22}$$

$$\tag{3.23}$$

which, when evaluated at $k = 0$, give the expected results

$$\lambda'(0) = c, \tag{3.24}$$

$$\lambda''(0) = c(1-c). \tag{3.25}$$

While we carried out this derivation for a specific distribution of a sample mean, any sample mean of independent and identically distributed (IID) random variables can have the corresponding SCGF derived by calculating the *cumulant generating function* of a single of the IID variables.

$$\lambda(k) = \ln \langle e^{kX} \rangle. \tag{3.26}$$

Equation (3.26) is known as *Cramér's Theorem* [95, 96].

To end this example, let us re-derive

$$I(f) = \sup_{k \in \mathbb{R}} \{ kf - \lambda(k) \}, \tag{3.27}$$

using eq. (3.11). As $\lambda(k)$ is differentiable and convex [90], we can calculate

$$I(f) = k^*(f)f - \lambda(k^*(f)), \tag{3.28}$$

where $k^*(f)$ is the solution to $\lambda'(k^*) = f$, allowing us to continue the derivation, as seen in eq. (3.29).

$$\begin{aligned}
\frac{ce^{k^*}}{(1-c)+ce^{k^*}} &= f \\
ce^{k^*} &= f(1-c) + cfe^{k^*} \\
(1-f)ce^{k^*} &= f(1-c) \\
e^{k^*} &= \frac{f(1-c)}{c(1-f)} \\
k^* &= \ln\frac{f(1-c)}{c(1-f)}.
\end{aligned} \tag{3.29}$$

We substitute the above into eq. (3.28), yielding

$$I(f) = f\ln\frac{f}{c} + (1-f)\ln\frac{1-f}{1-c}, \tag{3.30}$$

the same result as previously derived in eq. (3.5).

3.3 Dynamical Large Deviations

LDT provides a mathematical framework for analysing and understanding the probability of events occurring in a system, such as the behaviour of extreme fluctuations of various quantities the so-called *rare events*. As we have seen in earlier sections, this can help us quantify large fluctuations of certain random variables. We can leverage the same machinery to examine fluctuations in a dynamical context — i.e. on trajectories of configurations. We can view a trajectory $\omega = [x_1, x_2, \ldots, x_\tau]$ generated by some dynamics p as a random variable. Further, we can begin to examine certain values of some observable $\mathcal{O}$ on the trajectory, and investigate its fluctuations.

As a starting point, let us define what we mean by a Markov Jump Process. Typically, we can express the dynamics via a master equation [97]

$$\frac{\partial}{\partial t}P(C,t) = \sum_{C'\neq C} W(C'\to C)P(C',t) - R(C)P(C,t), \tag{3.31}$$

where $P(C,t)$ represents the probability of being in configuration C at time t, $W(C'\to C)$ is the rate of transition from configuration C' to C and finally $R(C)$ is the **escape rate** from C to a *different* state C', given by

$$R(C) = \sum_{C,C'\neq C} W(C\to C'). \tag{3.32}$$

It is useful to write the master equation (ME) as an operator, such that

$$\frac{\partial}{\partial t}|P(t)\rangle = \mathcal{W}|P(t)\rangle, \tag{3.33}$$

where we choose an orthonormal configuration basis ($\langle C|C'\rangle = \delta_{C,C'}$) and set the probability vector $|P(t)\rangle$ as

$$|P(t)\rangle = \sum_C P(C,t)|C\rangle. \tag{3.34}$$

We are left with the *Master* operator, defined as

$$\mathcal{W} = \sum_{C,C'\neq C} W(C \to C')|C'\rangle\langle C| - \sum_C R(C)|C\rangle\langle C|. \tag{3.35}$$

We define two important states. The first is the "*flat*" state which is a sum of all configurations, denoted by

$$\langle -| = \sum_C \langle C|. \tag{3.36}$$

Second, we have the stationary state, denotes as $|P_{\text{ss}}\rangle$ which is an eigenvector of the stochastic operator $\mathcal{W}$. Both the *flat* state and the *stationary state* are eigenvectors of $\mathcal{W}$, corresponding to the left and right eigenvector with matching eigenvalue 0 respectively. The statement $\langle -|\mathcal{W} = 0$ implies probability conservation and $\mathcal{W}|P_{\text{ss}}\rangle = 0$ implies that the probability vector does not change over time [92].

We derive the probability of a given state by solving the ME, yielding

$$|P(t)\rangle = \mathcal{Z}^{-1} e^{t\mathcal{W}}|P(0)\rangle, \tag{3.37}$$

where $\mathcal{Z}$ is the normalisation factor satisfied by

$$\mathcal{Z} = \sum_C \langle C|e^{t\mathcal{W}}|P(0)\rangle. \tag{3.38}$$

Equation (3.38) is equivalent to

$$\mathcal{Z} = \langle -|e^{t\mathcal{W}}|P(0)\rangle. \tag{3.39}$$

Often, we are interested in measuring quantities on a given trajectory, which we refer to as *observables* — denoted by $\mathcal{O}(\omega_t)$. Usually, the quantities of interest can be expressed as a time integral over a trajectory, written in the most general form as

$$\mathcal{O}(\omega_t) = \int_0^t dt' \langle X_{t'+dt'}| \left(\sum_C \kappa(C,t')|C\rangle\langle C| + \sum_{C,C'\neq C} \nu(C \to C', t')|C'\rangle\langle C| \right) |X_{t'}\rangle, \tag{3.40}$$

where $\kappa(C,t)$ is a (generally) time-dependent counting field which weights contributions to $\mathcal{O}$ for the configuration C dependent also on how much time is spent in that state. $\nu(C \to C', t)$ as a weighting of transitioning from configuration C to C' at time t and weights the flux contribution to the observable. If we restrict κ and ν to be homogenous in time, one can simplify the observable calculation as

$$\mathcal{O}(\omega_t) = t \sum_C \kappa(C)\mu_C(\omega_t) + t \sum_{C,C'\neq C} \nu(C \to C') q_{C,C'}(\omega_t), \tag{3.41}$$

where

$$\mu_C(\omega_t) = \frac{1}{t} \int_0^t dt' \langle X_{t'+dt'}|C\rangle\langle C|X_{t'}\rangle, \tag{3.42}$$

which represents the average time spent in the state C, and

$$q_{C,C'}(\omega_t) = \frac{1}{t}\int_0^t dt' \langle X_{t'+dt'}|C'\rangle\langle C|X_{t'}\rangle, \tag{3.43}$$

which represents the time averaged flux of changing from C to C' across the trajectory. One refers to μ as the *empirical measure* as it acts as a distribution of likelihoods of being in a given state C [92]. Similarly, ν is sometimes called the *dynamical* or *empirical* flux [92].

The probability of observing a value O for a given observable $\mathcal{O}$ is given by

$$P_t(O) = \sum_{\omega_t} P(\omega_t)\delta_{\mathcal{O}(\omega_t),O}. \tag{3.44}$$

which, for an observable of the form of (3.41), usually satisfies a large deviation principle

$$P_t(O) \asymp e^{-tI(\frac{O}{t})}, \tag{3.45}$$

where $I(\frac{O}{t})$ is the rate function given by the *intensive* observable $\xi = \frac{O}{t}$, dependent on using sufficiently well-behaved underlying dynamics. In this context, it means that the dynamics converges to a unique stationary state in some finite time.

Similarly, one can derive a moment generating function given by

$$Z_t(s) = \sum_O P_t(O)e^{-sO}, \tag{3.46}$$

which has a large deviation form of

$$Z_t(s) \asymp e^{t\lambda(s)}. \tag{3.47}$$

$\lambda(s)$ is the SCGF, whose derivatives at $s = 0$ give the cumulants of O scaled by time. From earlier sections, we know that the rate function and the SCGF are related by a Legendre-Fenchel transform, which, for this instance, reads [26, 89, 90]

$$\lambda(s) = -\inf_{\xi\in\mathbb{R}}\left[\xi s + I(\xi)\right]. \tag{3.48}$$

One can write the moment generating function (MGF) as [26, 89, 90, 92]

$$Z_t(s) = \langle -|e^{t\mathcal{W}_s}|P(0)\rangle, \tag{3.49}$$

where we have defined a *tilted* Markov generator, given as

$$\mathcal{W}_s = \sum_C \sum_{C'\neq C} e^{-s\nu(C\to C')}W(C\to C')|C'\rangle\langle C| - \sum_C \left[R(C) + s\kappa(C)\right]|C\rangle\langle C|. \tag{3.50}$$

For long times, application of the operator $e^{t\mathcal{W}_s}$ has approximately the same effect as applying the exponentiated maximum eigenvalue of $\mathcal{W}_s$, scaled by t. This maximum eigenvalue is the SCGF, allowing us to recover eq. (3.47).

For examples of large deviation analysis and techniques on problems in statistical physics, see reviews [26, 91, 92] and a selection of examples [98–111].

3.3.1 Doob Dynamics

It often of interest to directly study the ensemble of trajectories biased towards some values of an observable. In particular, we aim to study the tilted dynamics given by

$$\tilde{P}_s(\omega_s) = \frac{e^{-s\mathcal{O}(\omega_t)}P(\omega_t)}{\mathcal{Z}_s}, \tag{3.51}$$

where s is some conjugate variable to the observable $\mathcal{O}$, $P(\omega_t)$ is some original, unbiased, dynamics and $\mathcal{Z}_s$ is some normalisation factor, dependent on s. This tilted dynamics allows one to study the tail ends of the distribution of $\mathcal{O}(\omega_t)$, along with the fluctuations around a particular value. For $s = 0$, we recover the original trajectory. In order to sample this distribution, one often needs to calculate $\mathcal{Z}_s$, which can be difficult or impossible depending on the problem; when analytical calculation is difficult, alternative methods, such as those described in Section 4.3, are used. If one can discover dynamics which equivalently produces an ensemble of trajectories with probabilities given by $\tilde{P}_s(\omega_s)$, we refer to this dynamics as the *Doob dynamics*. Methods of calculating, or even approximating, the Doob dynamics, are often an aim of current research (e.g. see [112–118]). Research presented in Chapter 6 presents a RL approach to learning the Doob dynamics.

Chapter 4

Monte Carlo Sampling

Many problems considered in this thesis require the precise sampling of probability distributions, which may not always be analytically tractable. It is useful for us to take some time to consider useful numerical techniques for accurate statistical sampling. The majority of numerical techniques used fall under the "Monte Carlo" umbrella, which is used to describe numerical methods which rely on stochastic processes to answer questions. We have already discussed a similar variety of applications and methods in Chapter 2, but we take a deeper dive into these techniques and expand further into sampling unknown distributions.

The first section in this chapter will discuss a simple Monte Carlo algorithm for calculating π to describe how this technique can be used to approximate integral values. We will expand upon the *importance-sampling* technique, first introduced in Chapter 2, and apply this to the π estimation problem with an example auxiliary sampling distribution.

Following this, we will discuss *Markov-chain Monte Carlo* methods to sample distributions which are not fully known. We cover the well-known *Metropolis-Hastings* algorithm and how this can be adapted for trajectory sampling, extensively studied in chapters 6, 7 and 8.

4.1 Monte Carlo Integration

The classic problem used when introducing Monte Carlo integration is that of calculating π. We set up the problem by making a dart board with a square box perfectly inscribed with a circle which just touches the edges of the square, as seen in fig. 4.1. We know that the area of the square is simply given by $4r^2$ and the area of the circle is given by πr^2. Therefore, the ratio ρ of the area of the circle to the area of the square is given by $\rho = \frac{\pi}{4}$. Now, we can approximate ρ using random numbers, i.e. analogously throw random darts at the board. If we select particularly unskilled players whose hits are uniformly likely to hit anywhere on the board, we can count the number of hits inside the circle and divide this by the total number of darts thrown, which will be an estimate for ρ, denoted as $\tilde{\rho}$. When the number of darts $N \to \infty$, this estimate becomes more accurate. Hence, we can approximate $\pi \approx 4\tilde{\rho}$.

As the title of this section suggests, we can relate this method to integration. Specifically, we are interested in the integral of an observable that measures whether a point is inside

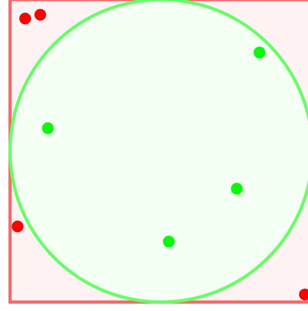

Figure 4.1: The π experiment setup, with a square inscribed with a circle. The green circles show a dart hit and the red circles show a miss outside the circle.

or outside the circle. This observable can be denoted as

$$\mathcal{O}(x, y) = \begin{cases} 1 & \text{if } x^2 + y^2 \leq 1 \\ 0 & \text{otherwise.} \end{cases} \tag{4.1}$$

The integral we wish to calculate is the expected value of this observable, using some probability distribution of $p(x, y)$, as given by

$$\langle \mathcal{O} \rangle = \int dxdy \; p(x, y)\mathcal{O}(x, y), \tag{4.2}$$

where integration is performed over all possible values of x and y. Under a uniform probability density in the region $-1 \leq x \leq 1$ and $-1 \leq y \leq 1$, we know that $\langle \mathcal{O} \rangle = \rho = \frac{\pi}{4}$. We can always approximate an expected value with an average over discrete values, such as

$$\langle \mathcal{O} \rangle \approx \frac{1}{N} \sum_i \mathcal{O}(x_i, y_i), \tag{4.3}$$

where $x_i \sim \mathcal{U}(-1, 1)$ and $y_i \sim \mathcal{U}(-1, 1)$. One can see this is equivalent to adding up all the darts that hit inside the circle. Varying N, we can calculate an approximation for π (using $\pi \approx 4\langle \mathcal{O} \rangle$), allow study of how accuracy changes with the number of samples. We show one instance of this experiment in fig. 4.2. One can see that this method is unbiased, but requires many random points to begin to converge (i.e. has a large variance).

4.1.1 Error Estimates

As our algorithm does not have access to the true value of π, how can we estimate the error of our approximation? As we are summing independent random variables, we can estimate the variance of the samples using

$$\mathrm{Var}(\langle \mathcal{O} \rangle) \approx \frac{1}{N} \left[\frac{1}{N} \sum_i \mathcal{O}(x, y)^2 - \left(\frac{1}{N} \sum_i \mathcal{O}(x, y) \right)^2 \right]. \tag{4.4}$$

We approximate the error as $\sigma = \sqrt{\mathrm{Var}(\langle \mathcal{O} \rangle)}$, which we call the *standard deviation*. In fig. 4.2, we plot the approximation with error bars calculated via the standard Gaussian error given by $\frac{\sigma}{\sqrt{N}}$. The random variable approximating $\langle \mathcal{O} \rangle$ in the limit of large N is Gaussian with a mean of $\frac{\pi}{4}$ and an error which is $\propto \frac{1}{\sqrt{N}}$. This means that increasing N by a factor of 100 only gives a factor of 10 reduction in error (i.e. a single decimal place).

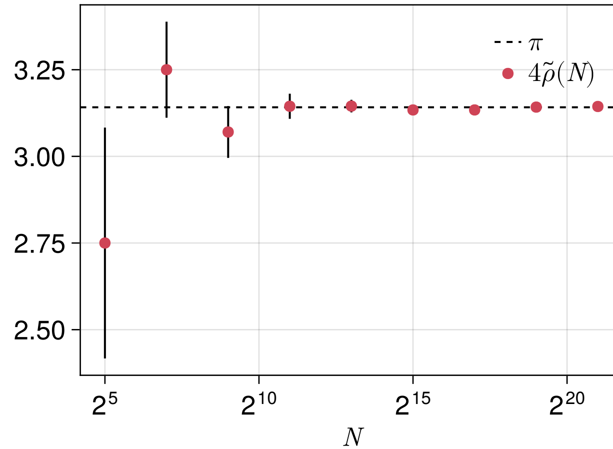

Figure 4.2: Convergence of the Monte Carlo integration method to π using darts. Error bars are calculated through the standard error.

Note that one only has an approximate error for the sample, which is a reliable indicator for large enough N. In some difficult situations, this error can vastly underestimate the true error [119]; e.g. if the samples generated are correlated (i.e. not independent).

4.2 Importance Sampling

In the previous example, we were able to successfully perform the integration as we were directly able to sample points from a known distribution. However, there are some situations where this direct sampling method can cause problems. Let us take the π calculation example and extend this into D dimensions (in the previous example we used $D = 2$). Instead of a square inscribed by a circle, we instead choose a hypercube with side length 2 and a hypersphere with radius 1, concentric with one another. For the sake of brevity, we will use the standard formula for the volume of a D-dimensional sphere with radius R [119], given by

$$V_D(R) = \frac{\pi^{\frac{D}{2}}}{\Gamma(\frac{D}{2}+1)} R^D, \tag{4.5}$$

where $\Gamma(x)$ is the generalisation of the factorial function extending to real numbers such that $\Gamma(x+1) = x\Gamma(x)$. In our case, we also need that $\Gamma(\frac{1}{2}) = \sqrt{\pi}$. In the limit of $D \to \infty$, we can use Stirling's approximation such that $\Gamma(\frac{D}{2}+1) \approx \frac{D}{2} \ln \frac{D}{2} - \frac{D}{2}$ which leaves

$$V_D(R) \underset{\sim}{\propto} e^{-\frac{D}{2} \ln \frac{D}{2}} R^D. \tag{4.6}$$

As we intend to approximate the ratio of the volume of the hypersphere to the hypercube (with volume $(2R)^D$), we can directly approximate these for both cases as

$$\rho = \frac{V_D(R)}{(2R)^D} \underset{\sim}{\propto} e^{-\frac{D}{2} \ln \frac{D}{2}}, \tag{4.7}$$

in the limit of $D \to \infty$. As the exponent is always negative, we can conclude that $\rho \to 0$ for large D, and hence larger dimensional geometries become super-exponentially harder

to sample with direct Monte Carlo. To see this, let us recreate the experiment for higher dimensions, defining our observable as

$$\mathcal{O}_D(\boldsymbol{x}) = \begin{cases} 1 & \text{if } ||\boldsymbol{x}||^2 \leq 1 \\ 0 & \text{otherwise,} \end{cases} \tag{4.8}$$

where $||\boldsymbol{x}||^2$ represents the square distance of the point from the origin, given by

$$||\boldsymbol{x}||^2 = \sum_j^D x_j^2. \tag{4.9}$$

Similar to before, we sample this vector with $x_j \sim \mathcal{U}(-1, 1)\ \forall\ j \in 1, \dots, D$ and label the sample $\boldsymbol{x}_i$. We can approximate the ratio of volumes of the hypersphere to the hypercube as

$$\langle \mathcal{O}_D \rangle \approx \frac{1}{N} \sum_i \mathcal{O}_D(\boldsymbol{x}_i). \tag{4.10}$$

The convergence to the mean depends on the variance of $\mathcal{O}_D$. As this is a Bernoulli random variable, the variance can be expressed as

$$\text{Var}(\mathcal{O}_D) = \langle \mathcal{O}_D \rangle (1 - \langle \mathcal{O}_D \rangle) \tag{4.11}$$

We can investigate the standard deviation as compared with the mean to get a relative error

$$\frac{\text{Var}(\mathcal{O}_D)}{\langle \mathcal{O}_D \rangle^2} = \frac{1}{\langle \mathcal{O}_D \rangle} - 1, \tag{4.12}$$

which for large D becomes approximately

$$\frac{\text{Var}(\mathcal{O}_D)}{\langle \mathcal{O}_D \rangle^2} \underset{\sim}{\propto} e^{\frac{D}{2} \ln \frac{D}{2}}, \tag{4.13}$$

which means that the relative error of the estimates will increase super-exponentially.

However, in these situations there are techniques which can be used to reduce this variance. One of these techniques is called *importance sampling* (briefly visited in Section 2.4.3 from the RL chapter), which allows us to choose a different distribution of our points $\boldsymbol{x}_i$ to estimate the integral with a (hopefully) lower variance.

Our original probability distribution for selecting points is uniform over space such that $p(\boldsymbol{x}) = c$ in the hypercube and 0 elsewhere. c satisfies the integral over the hypercube

$$\int dx_1 dx_2 \dots dx_D\ p(\boldsymbol{x}) = 1, \tag{4.14}$$

to ensure normalisation. For our sampling distribution, which we will denote as $\hat{p}$, we will assume that we want to construct the point using D independent samples from the same distribution such that $\hat{p}(\boldsymbol{x}) = \prod_i^D \hat{p}(x_i)$. As we want to bias the probability towards sampling $|x_i|$ close to zero, one valid choice is

$$\hat{p}(x) = \begin{cases} \alpha e^{-\beta x} & \text{for } 0 \leq x \leq 1, \\ 0 & \text{otherwise,} \end{cases} \tag{4.15}$$

with constants α and β. Note that we are only generating points in a unit hypercube instead of the width 2 hypercube, as the ratio of the hypersphere to the hypercube is the same in this region due to arguments of symmetry.

Given that we often can only generate uniform (or sometimes Gaussian) random numbers, we need a way of sampling from an arbitrary probability distribution. Hence, we will introduce an equation that is needed to transform the uniform random variable u (between 0 and 1 inclusive) into a random variable x distributed according to $\hat{p}(x)$. The correct relation ([119]) is

$$\hat{p}(x)dx = du, \tag{4.16}$$

which ensures probability conservation. We can integrate both sides to arrive at

$$-\frac{\alpha}{\beta}e^{-\beta x} + \gamma = u, \tag{4.17}$$

which can be rearranged to find x in terms of u, resulting in

$$x = -\beta \ln{(1 - \frac{u}{\gamma})}. \tag{4.18}$$

This means we can generate a sample $u \sim \mathcal{U}(0, 1)$, which can be transformed into x, which is distributed according to $\hat{p}(x)$, using eq. (4.18).

From here, we impose three conditions to get relations for our constants α, β and γ:

i. When $u = 0$, $x = 0$.

ii. When $u = 1$, $x = 1$.

iii. $\int_0^1 dx\ \hat{p}(x) = 1$.

The first of these conditions gives $\alpha = \gamma\beta$. The second gives $\gamma = (1 - e^{-\beta})^{-1}$. The third gives no new information, but is satisfied by the first two conditions. This means we have a choice of β which will determine γ and α. As an example, let us choose $\beta = \frac{D}{2}$, and use this to calculate α and γ,

$$\gamma =(1 - e^{-\frac{D}{2}})^{-1} \tag{4.19}$$

$$\alpha =\frac{D}{2(1 - e^{-\frac{D}{2}})}. \tag{4.20}$$

Our target is to calculate $\langle \mathcal{O}_D \rangle$ which is written as

$$\langle \mathcal{O}_D \rangle = \int dx_1 dx_2 \dots dx_D\ p(\boldsymbol{x})\mathcal{O}_D(\boldsymbol{x}). \tag{4.21}$$

The above equation can be rearranged by adding a factor of $\frac{\hat{p}(\boldsymbol{x})}{\hat{p}(\boldsymbol{x})}$, giving

$$\begin{aligned}\langle \mathcal{O}_D \rangle &= \int dx_1 dx_2 \dots dx_D\ p(\boldsymbol{x})\frac{\hat{p}(\boldsymbol{x})}{\hat{p}(\boldsymbol{x})}\mathcal{O}_D(\boldsymbol{x}) \\ &= \int dx_1 dx_2 \dots dx_D\ \hat{p}(\boldsymbol{x})\frac{p(\boldsymbol{x})}{\hat{p}(\boldsymbol{x})}\mathcal{O}_D(\boldsymbol{x}) \\ &= \int dx_1 dx_2 \dots dx_D\ \hat{p}(\boldsymbol{x})I(\boldsymbol{x})\mathcal{O}_D(\boldsymbol{x}),\end{aligned} \tag{4.22}$$

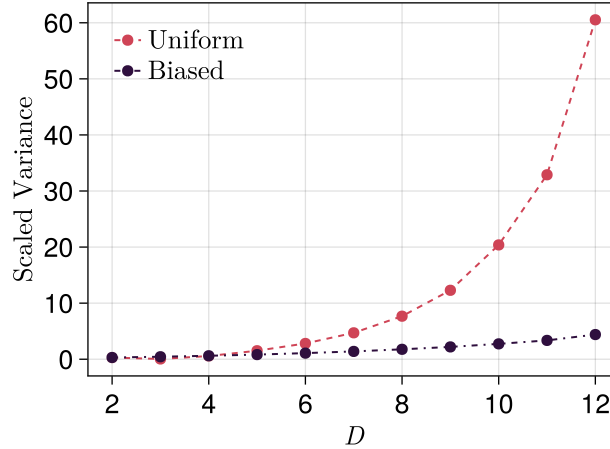

Figure 4.3: Comparison of the variance of Monte Carlo sampling methods on the hypercube-to-hypersphere ratio problem. Variance is scaled by the true ratio ρ, as calculated using eq. (4.5) and eq. (4.7). Each variance is calculated empirically using 10^6 samples. The biased curve uses the importance-sampling technique with points being sampled from eq. (4.15) using $\beta = \frac{D}{2}$.

where $I(\boldsymbol{x})$ represents the importance ratio of a sample at point $\boldsymbol{x}$, which scales the observable and is equal to

$$I(\boldsymbol{x}) = \frac{p(\boldsymbol{x})}{\hat{p}(\boldsymbol{x})}. \tag{4.23}$$

Importantly, we should note that we were able to do this as $\hat{p}(\boldsymbol{x}) \geq 0$ for any x when $p(\boldsymbol{x}) \geq 0$. If we translate this integral into a Monte Carlo approximation, we will get

$$\langle \mathcal{O}_D \rangle \approx \frac{1}{N} \sum_{i=1}^{N} I(\hat{\boldsymbol{x}}_i) \mathcal{O}_D(\hat{\boldsymbol{x}}_i), \tag{4.24}$$

where $\hat{\boldsymbol{x}}_i$ is sampled independently using $\hat{p}$. To illustrate the power of this importance sampling, let us perform an experiment where we measure the variance of the samples scaled by the square of the true mean for increasing dimensions. We can see the result of this in fig. 4.3, where it is clear that for higher D, direct sampling becomes increasingly inefficient. This means that one needs a much higher number of samples to get a good estimate of the integral, whereas the importance sampling method can converge on the true mean with fewer samples. For example, for $D = 12$ the ratio of the variances is around 13.8. This means that one would need approximately $13.8^2 \approx 190$ times as many samples to get to the same accuracy on the estimate.

4.3 Markov-Chain Monte Carlo Methods

In the previous section, we discussed a way of generating random numbers according to some non-standard distribution. This technique can be very useful if the probability distribution is known a-priori. However, this is not always the case, and we require a different technique to generate these samples — this will be the focus of this section.

To motivate this section, let us look at an example of a large deviation problem — sampling a biased probability distribution where we cannot directly calculate the re-

normalisation factor. Say that we have some original probability distribution of states X given by $p(X)$, and we want to examine a biased distribution weighted by the energy of the state, such that our biased (or tilted) distribution is given by

$$\tilde{p}(X; s) = \frac{e^{-sE(X)}p(X)}{\mathcal{Z}(s)}, \tag{4.25}$$

where $E(X)$ represents the energy of the state X, $\mathcal{Z}(s)$ represents the partition function which normalises the tilted ensemble and finally s is some control parameter which allows us to specify the strength of the biasing. We can calculate $\mathcal{Z}(s)$ by enforcing

$$\int dX \tilde{p}(X; s) = 1, \tag{4.26}$$

such that

$$\mathcal{Z}(s) = \int dX e^{-sE(X)}p(X). \tag{4.27}$$

This integral is over the entire space of possible configurations of X. If X is multidimensional and the expression for the energy is non-trivial, this calculation often cannot be done analytically. Instead, we want to come up with an algorithm that is able to numerically generate samples from $\tilde{p}$ without requiring this normalisation factor. We may be interested in generating samples so that we can evaluate an integral; e.g. the average value of an observable

$$\langle \mathcal{O} \rangle = \int dX \tilde{p}(X)\mathcal{O}(x). \tag{4.28}$$

As in the last section, we can approximate this integral using

$$\langle \mathcal{O} \rangle \approx \frac{1}{N} \sum_{i=1, X_i \sim \tilde{p}}^{N} \mathcal{O}(X_i), \tag{4.29}$$

where N is the number of samples used in the approximation. However, to perform this expectation, we need a way of drawing samples $X_i \sim \tilde{p}$. One method for creating such samples is the Metropolis-Hastings algorithm first introduced in the basic form in 1953 by Metropolis *et al* [120] and later generalised by Hastings [121]. For a review of the history of these methods see Ref [122].

4.3.1 Metropolis-Hastings Algorithm

We begin by giving a basic overview of this algorithm. First, let us consider starting at a configuration X_1 and drawing a new sample configuration, X_2, probabilistically, based only on the current configuration X_1. We continue this process to generate X_{n+1} with probability $p(X_{n+1}|X_n)$ to be later specified. This forms the basis of a Markov chain of correlated samples. The aim of the algorithm is to ensure that in the limit $n \to \infty$, distant samples become decorrelated and together follow a stationary distribution $\pi(X)$. The stationary distribution $\pi(X)$ exists if one can choose dynamics $p(X'|X)$ such that

$$\pi(X)p(X'|X) = \pi(X')p(X|X'). \tag{4.30}$$

However, we also must ensure that the state space is fully connected, such that starting from any configuration X, one can construct a Markov chain from X to X' with non-zero probability. We need to ensure that $p(X'|X)$ produces ergodic dynamics in the

configuration space, which is commonly achieved by using random walk dynamics, *à la* Ref. [120]. Metropolis-Hastings gives a general approach by allowing a split between a proposal generating dynamics $g(X'|X)$ and an acceptance regime $A(X', X)$ which is defined as

$$p(X'|X) = g(X'|X)A(X', X). \tag{4.31}$$

Therefore, we split our dynamics into two stages: (i) generate a sample X' using $g(X'|X)$ (ii) accept or reject this sample using the acceptance function $A(X', X)$. Usually, we are free to choose whatever ergodic dynamics we deem suitable for $g(X'|X)$, but we fix the choice of acceptance function such that the detailed balance condition (eq. (4.30)) is satisfied, and hence

$$\frac{A(X', X)}{A(X, X')} = \frac{\pi(X')g(X|X')}{\pi(X)g(X'|X)}. \tag{4.32}$$

There are a family of functions which satisfy the above condition, but the most common choice is the Metropolis criterion, defined as

$$A(X', X) = \min\left[1, \frac{\pi(X')g(X|X')}{\pi(X)g(X'|X)}\right]. \tag{4.33}$$

The ratio of $\frac{\pi(X')}{\pi(X)}$ is extremely important, especially when studying probability distributions with an unknown normalisation constant $\mathcal{Z}$ as this factor will cancel out when calculating the acceptance probability.

Input: Number of samples N, initial state X_1, proposal dynamics $g(X'|X)$ and target distribution ratio $\frac{\pi(X')}{\pi(X)}$
Output: Sequence of correlated samples $X_1 \to X_2 \to \dots X_N$.

```
1  for i ∈ 2, ..., N do
2      Sample new dynamics X' ~ g(X'|X_{i-1});
3      Calculate acceptance rate A(X', X_{i-1}) = min[1, π(X')g(X|X') / π(X)g(X'|X)];
4      Sample uniform random number V ~ U(0, 1);
5      if V < A(X', X_{i-1}) then
6          Accept proposal, X_i ← X';
7      else
8          Reject proposal and use old state X_i ← X_{i-1};
9      end
10 end
```

Algorithm 5: Metropolis-Hastings algorithm which generates a Markov Chain of correlated samples using proposal dynamics $g(X'|X)$ to sample $\pi(X)$.

The full algorithm is given in Algorithm 5.

4.3.2 Extensions to Trajectory Sampling

Markov chain Monte Carlo (MCMC) methods are usually very general and can be applied to a wide range of problems. In this section, we will cover a subgenre of sampling problems that deals with analysing a *trajectory ensemble* instead of an ensemble of states. We can think of the states (as described in the last section by X) now in terms of a trajectory ω which is a Markov Chain

$$\omega = X_1 \to X_2 \to \dots \to X_\tau, \tag{4.34}$$

where τ is the number of states in the trajectory and the probability of generating the next state is only dependent on the last. We will call the probability transition rates the *original dynamics*, denoted as $p(X_{i+1}|X_i)$. For now, we will consider a trajectory of fixed length at discrete unit time intervals[1]. Note that the time in the trajectory itself can be included in the state. We draw the starting state from some initial distribution $p(X_1)$.

We can note that the probability of the trajectory is given by

$$\pi(\omega) = p(X_1) \prod_{i=1}^{\tau-1} p(X_{i+1}|X_i). \tag{4.35}$$

In this thesis, we specifically consider trajectories that are exponentially weighted by some observable $\mathcal{O}$ with a tilted probability of

$$\tilde{\pi}(\omega) = \mathcal{Z}^{-1} e^{-s\mathcal{O}(\omega)} p(X_1) \prod_{i=1}^{\tau-1} p(X_{i+1}|X_i), \tag{4.36}$$

where, again, s is a *conjugate variable* to $\mathcal{O}$ and $\mathcal{Z}^{-1}$ normalises the distribution. Observables of interest are usually extensive in the length of the trajectory, such that they can be written as

$$\mathcal{O}(\omega) = \sum_{t=1}^{\tau} \mathcal{O}(X_t). \tag{4.37}$$

If one were to use the original dynamics $\pi(\omega)$ to sample the tilted dynamics, corrected by Metropolis-Hastings, it would be exponentially more difficult in τ. Problems with long τ would be inaccessible. However, we can solve this by using transition path sampling (TPS) methods instead. To motivate the methods, let us look at the tilted trajectory form of the acceptance ratio, given in eq. (4.38), in the Metropolis-Hastings algorithm (see Algorithm 5 for full algorithm).

$$A(\omega', \omega) = \min \left[1, \frac{\pi(\omega')g(\omega|\omega')}{\pi(\omega)g(\omega'|\omega)} \exp\left(-s \sum_{t=1}^{\tau} \mathcal{O}({X'}_t) - \mathcal{O}(X_t) \right) \right] \tag{4.38}$$

Notice that if we fix a portion of the trajectory, then those states' observables will cancel out. Additionally, terms in the probability ratio will cancel out and make this simpler to calculate. Furthermore, some observables have only local responses to chances in the state. If one fixes the majority of the configuration and performs a local update, one can locally calculate an update to the observable (such as in an Ising model [123]).

Following naturally from this arrangement, one can choose a proposal function that keeps a portion of the current trajectory fixed. Below, we outline a technique called *shooting* that does just this:

1. Randomly select a point in the trajectory t' uniformly from 1 to $\tau - 1$.
2. Create a new trajectory ω' where ${X'}_t = X_t \; \forall \; t \leq t'$.
3. Regenerate the end of the trajectory using unbiased dynamics $p({X'}_{t+1}|{X'}_t)$ for $t > t'$.

[1]One can extend these methods into the continuous time domain, but this is not considered in the scope of this thesis.

If the dynamics is time-reversible (e.g. a random walk), one can also fix the second half of the trajectory and rejuvenate the beginning half with the same benefits. We often choose the unbiased dynamics because it will obey detailed balance with itself such that the fraction

$$\frac{\pi(\omega')g(\omega|\omega')}{\pi(\omega)g(\omega'|\omega)}$$

will cancel out to 1.

Specific trajectory sampling techniques to increase the acceptance rate are presented in more detail in Chapter 7.

Chapter 5

Supporting Computational Infrastructure

A significant practical barrier to progress was the lack of computational frameworks that could simultaneously provide the high flexibility required for research and the high performance necessary to achieve the desired results. This challenge is common among researchers and is often referred to as the "*two-language problem*": programs are first written in a dynamic language like Python for initial development and testing and later translated into languages such as *C/C++* or *Fortran* when higher performance is needed. As this significantly impacts productivity, we decided to address this issue by utilising the *Julia* programming language [124]. *Julia* is a language that enables the creation of highly performant, yet still dynamic and generic programs.

To guide others on how to write high performance and highly parallel algorithms, I created a graduate-level course, taught as part of the Midlands Alliance Physics Graduate School (MPAGS) in 2023, accompanied by a book [125], specifically focused on writing high performance software in *Julia*. For the sake of brevity, the materials created for the course have been omitted. However, the first section of this chapter provides a concise overview of the covered topics.

Additionally, to support reproducibility and extensibility of my research, I have produced three generic software packages. These open-source packages are:

- `Experimenter.jl` [126] — A package to aid running experiments, distributed across multiple processes, i.e. on a HPC, and coordinating saving the final (and intermediate) results in a single database file.
- `TransitionPathSampling.jl` [127] — A package to aid efficient sampling of dynamics via TPS. This forms the base engine of the NNE algorithms researched and presented in Chapter 7 and Chapter 8.
- `SimpleNNs.jl` [128] — A package to provide a basic neural network library which has high performance on *small* neural networks, uses a *flat* parameter vector for the entire model, and also runs the neural networks on a GPU without excessive memory usage.

We explore the purpose and motivations behind these packages in more detail later in this chapter.

The final section briefly discusses contributions to other research articles, not presented in this thesis in the interest of succinctness.

5.1 Course: *High Performance Computing in Julia*

Modern research into ML, or scientific and numerical computing more broadly, requires use of highly efficient implementations of algorithms to take advantage of the vast computing resources of modern hardware. Much research relies on existing high performance libraries; in Python for example, one uses libraries such as `numpy` [129], `PyTorch` [87], `TensorFlow` [88] and many others as a base building block for numerical research. These libraries are incredibly successful at allowing users to take advantage of highly performant implementations using a Python API[1]. The aforementioned libraries have been very successful at providing high enough performance for the vast majority of use cases; the time a program spends facilitating the higher level instructions in Python is typically dominated by the time spent inside the library's high performance subroutines. Advantageously, these libraries are, to some extent, flexible, having constructs and patterns which can facilitate most research requirements. Unfortunately, these constructs cannot cover all use cases, as there are times when code cannot be easily contorted into the form demanded by these libraries. *Julia* offers an alternative approach which promises fast development speed, *à la* Python, while maintaining a high performance capability, rivalling programs written in languages like *C/C++* or *Fortran* — claiming to solve the "*two-language problem*" [124].

Admittedly, while it is certainly possible to write high performance code in Julia, it is by no means guaranteed that **any** code written will execute at speed. As with any language, if one cares about performance, one must *avoid* writing poorly performing code, instead of focusing on writing fast code. The course has the aim of teaching students how to write high performance on a range of systems, from a laptop to a supercomputing cluster.

For brevity, I have summarised the list of key topics included in the course below:

- Discussion of modern hardware, including CPU and GPU architectures.
- Discussion of software concepts, including: operating systems, compilers, multi-threading, hardware parallelism (e.g. SIMD[2]), memory management, data types and data structures.
- An overview of the *Julia* programming language.
- Measuring performance via benchmarking and profiling.
- Optimising serial code.
- Parallel algorithm design.
- Multithreaded programming on a single, multicore CPU.
- Multiprocessing across several machines, e.g. on a high performance cluster (HPC).
- Introduction to GPU programming using the `CUDA.jl` package.

[1] Application Programming Interface
[2] Single-Instruction Multiple Data

- Discussion of professional software engineering standards — in particular, how to produce reliable, reusable and reproducible programs.

All materials for the graduate course are available online through the website [125]. In particular, interested readers may gain insight from the accompanying book [130].

5.2 Package: `Experimenter.jl`

Whenever we conduct statistical studies, we usually have to run experiments over a range of parameters — and possibly take many repeats. Orchestrating this can involve writing a lot of boilerplate code for locating and naming data files. Additionally, many HPC system administrators advise against saving results in many smaller files and instead, encourage the use of a single (or a small number) of file(s) to reduce strain on the filesystem. As this is such a common task in research, I decided to develop a standard solution to this, making use of a centralised database to save results. The aims of this package are:

- To organise running and documentation of experiments, saving information about the input parameters of each experiment.
- To save experiment results into a single database file.
- To provide an optional way to distribute gathering of data across a cluster and aggregating results for saving in the database.
- To expose an optional interface (application programming interface (API)) to allow saving intermediate results, referred to as *snapshots*.
- To allow the continuation of stopped experiments, avoiding re-gathering existing results.

Most of this functionality is aimed at running on a HPC, where it is common to have computational job time limits, requiring intermediate results to be saved and continued in a second job. Additionally, many methods require human monitoring to see if the experiments need to be run for longer. Many of the techniques employed by researchers are *ad-hoc*, and this package aims to provide a standardised solution.

This package is open sourced for other users, hosted on GitHub [126]. Along with the source code, a documentation website is also available, guiding users on how to integrate the package with their existing software. The project contains a reasonably comprehensive test suite which makes use of unit testing to ensure the package works as intended. For the rest of this section we will quickly cover use of the package.

5.2.1 Example Usage

Firstly, we load the package and make available its functions with

```
using Experimenter
```

As we want to be able to save results, we open a database file locally, which uses SQLite [131] on the backend.

```
db = open_db("experiments.db")
```

This will create a new file in the present working directory called `"experiments.db"` and give a reference to this database stored in the variable `db`.

From here, we need an example experiment to run. For this experiment, we will just measure the final distance of a random walk with the following code (saved to a file called `"run.jl"`):

```
using random
function run_trial(config::Dict{Symbol,Any}, trial_id)
    results = Dict{Symbol, Any}()
    sigma = config[:sigma]
    N = config[:N]
    seed = config[:seed]
    rng = Random.Xoshiro(seed)
    results[:distance] = sum(randn(rng) * sigma for _ in 1:N)
    return results
end
```

Importantly, this function takes in exactly two arguments, the first of which is a dictionary[3] which maps parameter identifiers to its corresponding value, which can be of any type. Additionally, `Experimenter.jl` will also provide the function with a unique identifier for the specific trial being run, which enables saving snapshots (which is unused in our example). This function extracts the parameters contained in the configuration dictionary, performs some calculation and saves the results in a dictionary (with the same type as the input dictionary). These results are then communicated back (if conducted on another machine than the initial host) and saved to the database.

Once the wrapper code is written, we can define the experiment in a separate script:

```
config = Dict{Symbol,Any}(
    :N => IterableVariable([10, 20]),
    :seed => IterableVariable([1234, 4321]),
    :sigma => 1.0
)
experiment = Experiment(
    name="Test Experiment",
    include_file="run.jl",
    function_name="run_trial",
    configuration=deepcopy(config)
)
```

This simply creates the object with the necessary metadata for identifying the experiment, along with the experiment configuration. An experiment consists of an array of trials. With the custom type `IterableVariable`, one specifies that this parameter should form part of a grid search. Using the custom types "*tags*" the variable as something to be iterated over. These trials will be made up of the combinatorial product of these *tagged* variables. This file should also be checked into source control to keep a record of the experiment for later reproducibility.

[3] A data structure which maps from unique keys to values. Also known as a *hash map* in other languages.

The package provides the macro[4] `@experiment` to facilitate easy execution orchestration of the experiment. This macro allows for different modes to be specified, for example:

```
@execute experiment db SerialMode
```

The constant `SerialMode` can be replaced with either `MultithreadedMode` or `DistributedMode` to specify running in parallel in a single process (using multiple threads) or across multiple processes across one machine (or several). These parallel modes use the native threading and distributed libraries provided by Julia respectively. If running on a cluster, one should make use of the `ClusterManagers.jl` package to correctly launch multiple processes across the cluster and open the communication channels. As SLURM [132] is one of the most common schedulers on HPC systems, a documented example of how to launch the experiment runner is documented on the package documentation website.

As each process finishes running, the results are saved in the database. If the experiment process is cancelled before all trials are completed, one can simply rerun the script again, and only the uncompleted trials will be scheduled to run.

To extract the results for processing, simply open the database again, specifying the same file path as was used previously and use the `get_trials_by_name` function, with the name of the experiment chosen:

```
db = open_db("experiments.db");
trials = get_trials_by_name(db, "Test Experiment");
results = [t.results for t in trials];
```

This will give an array of dictionaries matching the results returned by the `run_trial` function. One can also access the configuration for each trial with:

```
configurations = [t.configuration for t in trials];
```

For additional features such as saving snapshots and re-running failed experiments, see the examples in the package documentation.

5.2.2 Limitations

The package is architected under the following assumptions:

- The overhead of sending, serialising and saving results to the database is negligible compared to the runtime of each trial.
- The size of the results being saved is relatively modest (e.g. $<$ 1TB for the whole experiment).
- Each individual trial is reasonably computationally expensive (e.g. longer than 10 minutes).

[4]Macros allow for meta-programming, i.e. code that writes code.

- There are a relatively small number (e.g. $< 10^4$) of trials per experiment.

I designed this package first and foremost for ease of use, not to be capable of handling extremely high throughput. Management of the database is strictly single threaded, which locks the database when saving results. Experiments which require high performance storage will require a more complex package instead. Possible future extensions may allow different back end storage solutions that allow for parallel reads and writes, such as HDF5 instead of SQLite, to increase the maximum throughput of the package.

5.3 Package: `TransitionPathSampling.jl`

While the other packages presented in this chapter have a wide range of possible use cases, this package aims to solve a targeted problem. This is for specifically performing the trajectory "*training*" dynamics presented in Chapter 7 and Chapter 8. In the package, I provide a set of utilities that allow for running discrete time random walk dynamics efficiently on an array of values. This includes the ability to sample trajectories according to a biased dynamics given by:

$$\tilde{p}(\Theta) = \frac{e^{-s\mathcal{O}(\Theta)}p(\Theta)}{\mathcal{Z}(s)}, \tag{5.1}$$

where $\mathcal{O}(\Theta)$ is some observable quantity of interest, s is a control parameter and $p(\Theta)$ represents a discrete time continuous space random walk dynamics on the trajectory. We use $\mathcal{Z}(s)$ to renormalise the distribution.

The dynamics is updated via TPS, using the shooting and bridging techniques described in Section 4.3.2 and Chapter 7. The implementation is written using abstractions, allowing execution to be device-agnostic — running on both the CPU and GPU. This enables the parameters to be updated on the GPU without needing to transfer back and forth between host and device memory when calculating the updates.

As will be presented in Chapter 8, an efficient algorithm for calculating parameter updates can be done via minibatching the calculation of the observable. This requires a tight coupling between the observable and the TPS minibatch algorithm, as this is an *adaptive* sampling technique. `TransitionPathSampling.jl` provides an API for writing the interface between one's custom observable calculation and the minibatch method.

As this package is designed for research purposes, `TransitionPathSampling.jl` also provides a mechanism for injecting callbacks to execute flexibly throughout the training loop, exposing all information used by the algorithm, allowing for detailed statistics to be collected. Along with this, there are some capabilities for custom iterators that stop only when convergence is detected — using a naïve heuristic approach.

5.3.1 Limitations

This package was designed with my own research goals in mind, and grew over time. As such, the scope of its usefulness is very limited. We can summarise the main limitations of the package as follows:

- The main algorithms in `TransitionPathSampling.jl` are centred around the specific dynamics of interest (discrete time continuous space independent random

walks).

- The algorithms assume that all sampling is of the biased form given in eq. (5.1), where the unbiased dynamics obey detailed balance with the perturbation dynamics.
- When minibatching, the algorithm assumes a finite set of samples and does not extend to problems with potentially infinite samples. This can be mitigated by using a fall back value for the maximum number of samples allowed per update.
- Also in minibatching, the error tolerance is currently calculated using a heuristic to avoid unnecessary quadratic scaling with the final minibatch size, and to avoid excessive memory allocation. Results in Chapter 8 explicitly remove this error calculation and therefore this heuristic leaves results unaffected.
- There is currently very little documentation on how to use the package, other than its usage in the unit test suite and in the two code repositories published with the research articles.

5.3.2 Future Work

This package, while providing the necessary backbone for my research, could be much more useful for a wider audience with further work. As the design of the package improved over time, it would be prudent to abstract and redesign some of the internals to remove the assumptions listed in the above section. For example, one can provide an interface for specifying the exact form of the acceptance criteria efficiently, while being flexible enough for varied user problems.

Some support for running multiple trajectory sampling techniques in parallel and aiming to converge towards a stationary state using automatic convergence detection would be a great improvement. This would save the user a great deal of time spent manually inspecting the results from several runs to find a suitable runtime for each parameter input.

Finally, additional utilities for specifically saving, resuming and safely cancelling training would vastly improve usability in research. Currently, there is a callback system which allows the user to specify in their own code implementations for these utilities. The current approach is not the most ergonomic and some standard callbacks would be preferable.

5.4 Package: `SimpleNNs.jl`

Aside from the RL research conducted in this thesis, much focus was on studying training collections of neural network models via trajectory sampling methods such as TPS. Importantly, the way these neural networks were trained is very different to standard gradient methods. We perform dynamics on the flat parameter vector of each model in the trajectory, rather than taking into account the architecture of the model — as in gradient descent. Most existing machine learning frameworks (e.g. `Flux.jl` [133, 134], `PyTorch` [87], `TensorFlow` [88]) are set up to modularise a neural network into separate layers, each with a local set of parameters. It is expected that if one wants to modify the parameters of the network, one would modify one layer at a time. This approach is very good for gradient descent optimisation, as the algorithms remain very simple and

yet flexible. However, performance is severely degraded when using a different memory access paradigm (such as modifying a random 25% of the parameters, spread across the model).

At first, `Flux.jl` (written in *Julia*), provided a way of deconstructing a model into a flat parameter vector, together with a function to reconstruct the model using the flat parameters. This allowed for dynamics to be natively handled in native *Julia* code; importantly, this allowed the mutated parameters to be easily passed back to the model framework for inferences needed to calculate the new loss. This technique was not meant to be used in a hot loop[5], and had poor performance in terms of memory usage. As *Julia* is a garbage collected language, this caused slowdowns due to a lot of garbage collector (GC) pressure. Additionally, as the GC pauses execution of concurrent threads, this severely impacts one's ability to take advantage of modern hardware to run multiple experiments in parallel. In fact, when using multiple threads running parallel experiments, the GC pressure on the GPU memory is so high that the GC cannot free the GPU memory fast enough, resulting in a program crash and loss of results. As gathering data for Chapter 7 took on the order of months, on the second I decided to implement my own NN library, which would be used for gathering results for Chapter 8, with four specific goals:

1. To run inference with a flat parameter vector of the model.
2. To run inference not only on the CPU, but also on the GPU.
3. To parallelise inference on a single GPU, without running out of memory.
4. To be competitive with existing popular ML frameworks for my particular use case.

I took inspiration from the `SimpleChains.jl` [135] package, which showed that *micro-optimisations* for small networks running on the CPU can be up to 5× faster than `PyTorch` implementations and around 2 to 22× faster than equivalent `JAX` [136] implementations. However, this library is heavily limited to CPU only execution (at the time of writing). My aim, was to produce a similar package which had simpler source code (to allow for extensibility), while also providing native GPU support (specifically for CUDA GPUs).

At the time of writing, my package — `SimpleNNs.jl` — has similar functionality to `SimpleChains.jl`, but can run on the GPU. The full list of features from the package are as follows:

- Uses a *flat* 1D parameter vector for representing the entire model, together with slices and transformations to extract the parameters for each layer.
- Allows for building simple neural networks with a choice of dense, convolutional and max pooling layers.
- Runs inference (i.e. a forward-pass) using preallocated buffers, removing **all** runtime memory allocations.
- Support for any user-defined activation functions, or standard functions (e.g. ReLU, logistic sigmoid or hyperbolic tangent) on forward passes.
- Support for automatic gradient calculation on basic loss functions like mean-squared error and cross-entropy loss using the back-propagation algorithm introduced in

[5] A loop in which the program spends most of its time.

Parameter	Value
Device	`RTX 3090 NVIDIA GPU`
Epochs	10^5
τ	32
s	50.0
σ	0.25
Minibatch chunk size	120
Model architecture	See Appendix B in Chapter 8
Problem	MNIST classification (all digits) [28]

Table 5.1: Training parameters for benchmarking the two different implementations using `SimpleNNs.jl` and `Flux.jl`. The performance benchmark runs the minibatch training algorithm described in chapter 8 for 10^5 iterations with the above parameters using the different neural network libraries to implement the model part of the algorithm.

Chapter 1. No automatic differentiation is implemented for user-defined functions, but can be added manually.

- Allows extensibility to new types of layers and loss functions via *Julia*'s rich type system.
- Runs back-propagation (backward-pass) with minimal runtime memory allocations.

It should be noted that the scope of the package is limited to what is described above, and is not as flexible as a package like `Flux.jl`, which includes automatic differentiation capabilities for user-supplied code. Fortunately, these are all superfluous for the research conducted into NNEs.

To validate this package, I measured the relative speed-up of using `SimpleNNs.jl` in place of `Flux.jl` for calculating the minibatch losses required on each epoch of training. For this benchmark, I trained a NNE with parameters specified in Table 5.1. Refer to Chapter 8 for details on the experiment and the parameters.

I measured the total number of samples evaluated during training, divided by the time taken to conduct the training to get a speed. Timing did not include loading the dataset into memory or creating the model, as this is only done once at the beginning of training. When running on multiple concurrent threads, timing was measured to be when the last thread finished the 10^5 epochs — i.e. the total wall time. As `Flux.jl` did not run on multiple threads without the potential of crashing, this timing was only measured for running on a single thread. All speeds are presented relative to the speed of the single threaded `Flux.jl` implementation, measured to be approximately 140k images per second.

In fig. 5.1, we see that even on a single thread, `SimpleNNs.jl` outperforms `Flux.jl` with a speed increase of around 23%. We also see that increasing the number of threads (e.g. the number of experiments run in parallel using the same GPU) increases the total throughput of the GPU. Even using only 2 threads gives a 95% improvement over `Flux.jl`. We see that the performance increase saturates at 5 threads, where the scheduling of the GPU kernels starts to hinder performance, and adding additional threads only decreases overall throughput. At 5 threads we see approximately a 3.2 times throughput over `Flux.jl`.

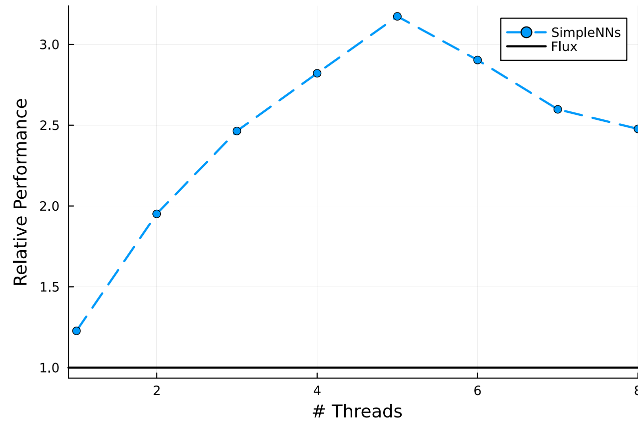

Figure 5.1: Speed of training of the `SimpleNNs.jl` implementation relative to that of `Flux.jl`. Data is based on real world experiments, and contains time evaluating the dynamics of the training process, reducing the relative performance of the two metrics.

As a reference calculation, a *single* figure for Chapter 8 used 168 individual experiments; each experiment takes, on average, around 20 hours to run on a single thread using `SimpleNNs.jl`. This implies that `Flux.jl` would take approximately 24.4 weeks of GPU compute to collect the same data. As running on 5 threads yields a significant speed-up, I was able to get the results with a comparative 7.6 weeks of compute time, saving around 16.8 weeks of compute overall.

5.5 Computational Contributions to Other Published Works

Using the techniques discussed in Section 5.1, I was able to contribute to other works that are not presented in full in this thesis.

The first of these contributions was to the work culminating in the article [137]

> "*Boundary conditions dependence of the phase transition in the quantum Newman-Moore model*" by K. Sfairopoulos, L. Causer, **J. F. Mair** and J. P. Garrahan *arXiv 2301.02826.*

For this work, I was able to significantly improve performance of the cellular automata methods employed during the research. This included algorithmic improvements for detecting cycles using Floyd's tortoise and hare algorithm [138] and implementation improvements by using fast bit shift operators.

The second contribution was to the computational work behind the article [139]

> "*Rejection-free quantum Monte Carlo in continuous time from transition path sampling*" by L. Causer, K. Sfairopoulos, **J. F. Mair** and J. P. Garrahan *arXiv 2305.08935.*

These contributions included discussing algorithmic implementation details and performance improvements via generic, *Julia*-specific programming strategies.

Chapter 6

A Reinforcement Learning Approach to Rare Trajectory Sampling

The following work is from the publication "*A reinforcement learning approach to rare trajectory sampling*" by D.C. Rose, **J. F. Mair** and J. P. Garrahan in the New Journal of Physics 23 013013 (2021) [140].

This work formalises the relationship between rare trajectory sampling, studied in large deviation theory (Chapter 3), and reinforcement learning (Chapter 2). In RL, optimal behaviour is defined by choosing actions which maximise the cumulative set of scalar signals received at each time step (called the rewards). We derive what the reward function must be in order to align the optimal behaviour with that of sampling rare events with a desired probability distribution. This is equivalent to finding the so-called *Doob* dynamics (see Section 3.3.1) which best sample rare events. Our approach is able to train approximate near-optimal auxiliary dynamics which can be corrected via trajectory sampling techniques like *transition path sampling* (see Section 4.3.2 and [141]), to produce samples of rare events.

Corrections:

- After Eq. (12), *"This cound ..."* should be *"This could ..."*

IOP Publishing New J. Phys. 23 (2021) 013013 https://doi.org/10.1088/1367-2630/abd7bd

New Journal of Physics
The open access journal at the forefront of physics

Deutsche Physikalische Gesellschaft DPG
IOP Institute of Physics
Published in partnership with: Deutsche Physikalische Gesellschaft and the Institute of Physics

OPEN ACCESS

RECEIVED
25 August 2020

REVISED
25 November 2020

ACCEPTED FOR PUBLICATION
31 December 2020

PUBLISHED
28 January 2021

Original content from this work may be used under the terms of the Creative Commons Attribution 4.0 licence.

Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.

PAPER

A reinforcement learning approach to rare trajectory sampling

Dominic C Rose* , Jamie F Mair and Juan P Garrahan

School of Physics and Astronomy and Centre for the Mathematics and Theoretical Physics of Quantum Non-Equilibrium Systems, University of Nottingham, Nottingham NG7 2RD, United Kingdom

* Author to whom any correspondence should be addressed.

E-mail: dominicdrose@gmail.com, dominic.rose1@nottingham.ac.uk, ppyjm13@nottingham.ac.uk and juan.garrahan@nottingham.ac.uk

Keywords: machine learning, reinforcement learning, large deviation theory, constrained dynamics, Monte-Carlo sampling

Abstract

Very often when studying non-equilibrium systems one is interested in analysing dynamical behaviour that occurs with very low probability, so called *rare events*. In practice, since rare events are by definition atypical, they are often difficult to access in a statistically significant way. What are required are strategies to 'make rare events typical' so that they can be generated on demand. Here we present such a general approach to adaptively construct a dynamics that efficiently samples atypical events. We do so by exploiting the methods of *reinforcement learning* (RL), which refers to the set of machine learning techniques aimed at finding the optimal behaviour to maximise a reward associated with the dynamics. We consider the general perspective of dynamical trajectory ensembles, whereby rare events are described in terms of ensemble reweighting. By minimising the distance between a reweighted ensemble and that of a suitably parametrised controlled dynamics we arrive at a set of methods similar to those of RL to numerically approximate the optimal dynamics that realises the rare behaviour of interest. As simple illustrations we consider in detail the problem of *excursions* of a random walker, for the case of rare events with a finite time horizon; and the problem of a studying current statistics of a particle hopping in a ring geometry, for the case of an infinite time horizon. We discuss natural extensions of the ideas presented here, including to continuous-time Markov systems, first passage time problems and non-Markovian dynamics.

1. Introduction

In physics, chemistry and many areas of science it is often the case that one wishes to study systems with dynamics which are highly variable and fluctuating, and where important information is contained in 'rare events', meaning particular instances of the dynamics which are very far from typical. Since analytical study of the statistics of trajectories is almost always intractable beyond the simplest model systems one must resort to sampling trajectories numerically. The main challenge is how to access in an efficient manner the atypical trajectories that give rise to the rare events of interest [1, 2].

A common problem is that of estimating the large deviation (LD) statistics [3] of time-extensive observables in systems with Markovian stochastic dynamics. This is difficult in general [4–21] as such observables are concentrated around their average values which makes accessing the tails of their distributions an exponentially in time hard numerical task. In the dynamical LD context, several approaches have been developed which attempt to ameliorate the exponential scarcity of rare trajectories within the original dynamics, often based either on population dynamics, such as cloning or splitting [4–6, 8, 22, 23], or on importance sampling in trajectory space, such as transition path sampling (TPS) [1, 24].

Since rare events by definition are hard to obtain with the original dynamics of the system, a key approach is to find an alternative sampling dynamics that gives access to rare trajectories in an optimal manner [25–37]. There is an intuitive similarity [38] in this search for an optimal sampling dynamics and the general problem of reinforcement learning (RL) [39]. Specifically, direct parametrisation of dynamics, such as the one done in the context above of trajectory sampling, is akin to policy gradient methods [40, 41]

© 2021 The Author(s). Published by IOP Publishing Ltd on behalf of the Institute of Physics and Deutsche Physikalische Gesellschaft

within RL. Exploring the connections between rare trajectory sampling and RL is the main aim of this paper.

The use of RL methods in physics is of course a rapidly growing area. Examples include applications in quantum state preparation and quantum control [42–47], quantum eigenstates [48, 49], policy guided Monte Carlo simulations [50], and evolutionary RL for LDs [51] and for thermodynamic control [52].

The key results and contributions of this paper are the following. (i) Using a generic formulation, which includes studying conditioned dynamics and cumulant generating functions as special cases, we demonstrate that the problem of optimizing a dynamics for sampling rare trajectories is identical to a form of regularized RL. This connection both allows the adaptation of RL techniques to be used in sampling rare trajectories, and provides a new range of problems on which RL techniques can be tested and compared. (ii) This form of regularized RL has not previously been considered using policy-gradient based techniques. We pedagogically present a range of such techniques for optimizing the sampling of rare trajectories. (iii) We review a small portion of the broad range of possible algorithms RL introduces through its connection with rare trajectory sampling. (iv) We specialize to the long-time limit, relevant to the LDs of Markov chains, finding that the regularized RL algorithms automatically estimate the scaled cumulant generating function (SCGF) in the process of optimizing the dynamics.

The approach we present here has connections—but also important differences—to recent works exploring related ideas [53], particularly in diffusive processes [54–56]. It is demonstrated throughout using problems based on random walkers.

The paper is organised as follows. In section 2 we review the trajectory ensemble method in systems with stochastic dynamics, discuss their reweighting, and how rare trajectories relate directly to such reweightings. In section 3 we pedagogically develop general methods for rare trajectory sampling based on RL, focussing on obtaining the optimal dynamics for finite problems. These methods are based on minimising expected likelihood, or a Kullback–Leibler (KL) divergence, and directly connect to maximum entropy RL and regularization [57–61]. We illustrate our approach with the simple (and solvable) example of random walk excursions [62]. We follow this in section 4 by reviewing a range of possible variations of these algorithms found in the RL literature, translated into our setting, which are made available by the connection between regularized RL and trajectory sampling. Section 5 extends the ideas of sections 2 and 3 to the case of long times, viewed as an infinite horizon problem, establishing the connection to LD theory. This connection implies these algorithms for optimizing the dynamics simultaneously provide an estimate for the SCGF, discussed in section 5.4. We conclude with section 6 outlining further extensions and possible adaptations of the methods presented here. This paper is intended to be the first in a series of works exploring connections between the physical and mathematical understanding of trajectory ensembles, and the computer science understanding of RL. Code produced to produce results for the examples shown in this paper is available on Github at [63].

2. Formulation and applications

We begin by introducing the formalism we use to describe trajectory ensembles, followed by a precise definition of the reweighted ensembles we consider. We then discuss how two cases in which rare trajectories have a significant impact, conditioned ensembles and cumulant generating functions, can be viewed as studies of reweighted trajectories ensemble. Finally, we discuss how our approach relates to—and crucially, differs from—the traditional formulation of RL.

2.1. Formalism and aim: trajectory ensembles and reweightings

We consider a system evolving over time t with state x_t. For simplicity we consider a discrete time dynamics given by Markovian transition probabilities $P(x_t|x_{t-1})$, with t taken to be a dimensionless integer denoting how many steps have occurred since the initial state. This can be simply extended to time-dependent transition probabilities $P(x_t|x_{t-1}, t)$. Further, in section 6 we discuss the extension to non-Markovian problems.

Trajectories consisting of sequences of states are labelled as

$$\omega_{t_0}^{T} = \{x_t\}_{t_0}^{T}, \tag{1}$$

where x_t is the state at time t, t_0 is the initial time and T is the final time. When ω appears multiple times in the same equation, we follow the convention that where their times overlap, they refer to the same states.

The probability of each trajectory is then given by

$$P\left(\omega_0^T\right) = \prod_{t=1}^{T} P(x_t|x_{t-1})P(x_0), \tag{2}$$

where $P(x_0)$ is the probability of a trajectory being initialized in the state x_0. These trajectory probabilities define a trajectory ensemble that we will frequently refer to as the **original dynamics** P. Throughout the paper we will make extensive use of expectation values over different trajectory ensembles, which we shall denote

$$\left\langle O\left(\omega_0^T\right)\right\rangle_P = \sum_{\omega_0^T} P\left(\omega_0^T\right) O\left(\omega_0^T\right), \tag{3}$$

where O is some function of the trajectory and the subscript denotes the trajectory ensemble over which the expectation is taken. We will also use conditional expectations over the future of a state

$$\left\langle O\left(\omega_t^T\right)\right\rangle_{P,X_t=x} = \frac{\sum_{\omega_0^T:x_t=x} P\left(\omega_0^T\right) O\left(\omega_t^T\right)}{\sum_{\omega_0^T:x_t=x} P\left(\omega_0^T\right)}, \tag{4}$$

where X_t denotes the random variable corresponding to the state at time t. Finally we will make use of the fact that the expectation of an expectation is simply the expected value: more specifically, we will use the identity

$$\left\langle f\left(\omega_t^T\right) g(x_t)\right\rangle_P = \left\langle \left\langle f\left(\omega_t^T\right)\right\rangle_{P,X_t=x_t} g(x_t)\right\rangle_P. \tag{5}$$

The problem we consider in this paper is finding a new dynamics which efficiently samples rare trajectories of some original Markovian dynamics P as defined above. In the next subsection we will provide examples showing many rare trajectory problems can be framed as the task of sampling a reweighting of the original trajectory ensemble. As such, we will now define what we generally mean by a reweighted trajectory ensemble. We will consider a weighting function which possesses a Markovian product structure: that is, the weight for each trajectory is given by

$$W\left(\omega_0^T\right) = \prod_{t=1}^{T} W(x_t, x_{t-1}, t), \tag{6}$$

where

$$W(x_t, x_{t-1}, t) \geqslant 0 \quad \forall\ (x_t, x_{t-1}, t). \tag{7}$$

This defines a reweighted trajectory ensemble as

$$P_W\left(\omega_0^T\right) = \frac{W\left(\omega_0^T\right) P\left(\omega_0^T\right)}{\left\langle W\left(\omega_0^T\right)\right\rangle_P}. \tag{8}$$

Our goal is then to find a new Markovian dynamics which generates a trajectory ensemble as close to this as possible, in a precise sense defined in terms of the KL divergence in section 3. While it is not immediately clear from equation (8), these trajectory probabilities can always be decomposed exactly into a set of time-dependent Markovian transition probabilities, as demonstrated in appendix A. Conditions for when this is the case in diffusive systems have previously been studied under the name penalizations in probability theory [64]. However, for complex problems it will be difficult to calculate this **exact dynamics**. It is for this reason that we present an approximate approach based on mapping the problem onto a regularized form of RL.

We note here that, similar to how this approach extends naturally to a non-Markovian original dynamics, more general trajectory reweightings can be considered than the Markovian product structure of equation (6). For more general reweightings the exact dynamics which reproduces the reweighted ensemble is naturally non-Markovian, even if the original dynamics is not. This is discussed further in section 6 and will be studied in future work.

2.2. Applications: rare trajectories as reweighted ensembles

We will now discuss how a variety of rare trajectory problems can be seen as a reweighting. In this case the reweighted ensemble is difficult to study using simulations based on the original dynamics, necessitating the use of alternative sampling schemes such as cloning and TPS, and/or the construction of an adapted sampling dynamics [1, 4–6, 8, 22–37]. Our work will supplement these by connecting the construction of an alternative sampling dynamics to RL.

To make our discussion of applications concrete, we will use a simple model as a recurring example: a random walker. That is, the original dynamics is that of a single particle hopping on a lattice, where the state x takes integer values, with Markovian transition probabilities $P(x \pm 1|x) = 1/2$. We will consider both infinite and periodic boundaries when we study rare events of this model in finite and long times, respectively. The probability of each trajectory takes a particularly simple form, being just $P(\omega_t^T) = 2^{-(T-t)}$. We will consider a variety of rare event problems based on this model, related either to its instantaneous position x or to an observable of the full trajectory, notably the area

$$A(\omega_t^T) = \sum_{t'=t}^{T} x_{t'}. \tag{9}$$

2.2.1. Conditioned dynamics

The first class of problems we consider are those in which the trajectory ensemble is conditioned on some observation of the trajectory. That is, given some statement about the trajectory that is either true or false, we wish to consider only the subset of trajectories for which the statement is true. Here the weight is simply a binary $W\left(\omega_0^T\right) = 0$ if the statement is false, and $W\left(\omega_0^T\right) = 1$ if the statement is true. The resulting ensemble then consists of rare trajectories of the original dynamics if the probability of the condition being true is small.

For example, we may condition the trajectory ensemble of the random walker on ending in the state $x_T = 0$, with an initial condition of $x_0 = 0$, often called a random walk bridge [62]. The weights for each transition would then be precisely defined as $W(x_T, x_{T-1}, T) = \delta(x_T)$ and $W(x_t, x_{t-1}, t) = 1$ for $0 < t < T$. Such a trajectory is relatively rare in the original dynamics. The probability of generating such a trajectory in the original dynamics is equal to the number of such trajectories, multiplied by their probability: the number of trajectories is simply the number of orderings of an equal number of up and down steps, resulting in $P(x_T = 0|x_0 = 0) \propto T^{-\frac{1}{2}}$.

A harder problem would be to retain the same constraint on the end, but additionally require $x_t \geqslant 0$ for all t, known as random walk excursions [62]. Using the step function $H(x_t)$, equal to zero for $x_t < 0$ and one otherwise, the weights may then be written $W(x_T, x_{T-1}, T) = \delta(x_T)$ and $W(x_t, x_{t-1}, t) = H(x_t)$ for $0 < t < T$. As can be seen in appendix A, in this case the number of trajectories relates to Catalan numbers, with $P(x_T = 0, x_t \geqslant 0\ \forall\, t|x_0 = 0) \propto T^{-\frac{3}{2}}$. Thus these excursions are substantially rarer than the bridges. Both excursions and bridges are have been studied extensively in a continuous time and space context of Brownian motion, see e.g. [62].

In our approach it will be necessary to have weights which are always non-zero. As such, to consider conditioned problems we will first need to **soften** the weights, setting the trajectory weight to 1 on correct trajectories and <1 on incorrect trajectories. In particular, we can consider the weightings to be given by some measure D which returns 0 when the condition is true and is positive when the condition is false

$$W\left(x_t, x_{t-1}, t\right) = \mathrm{e}^{-sD\left(x_t, x_{t-1}, t\right)}, \tag{10}$$

where s is a parameter determining how heavily suppressed incorrect trajectories will be: in the limit $s \to \infty$, only correct trajectories remain, recovering the ensemble of the hard constraint. For example, to recover a softened version of the random walk bridges or excursions, we may set

$$D\left(x_t, x_{t-1}, t\right) = x_t^2 \delta_{t,T} + b(1 - H(x_t)), \tag{11}$$

where b is a parameter, returning a softened bridge problem at $b = 0$ and a softened excursion problem at $b > 0$.

2.2.2. Tilted ensembles and cummulant generating functions

Suppose we wish to study the statistics of some time integrated observable

$$O\left(\omega_0^T\right) = \sum_{t=1}^{T} o(x_t, x_{t-1}, t). \tag{12}$$

This cound by done by considering conditioned ensembles for each of its possible values, however, this is often a difficult task even for a single value [34, 35]. While softened constraints are easier for individual values, annealing the constraint over a whole range of values could be computationally demanding. A common solution is to instead consider the observables **cumulant generating function**, given by

$$Z(s, T) = \left\langle \mathrm{e}^{-sO\left(\omega_0^T\right)} \right\rangle_P. \tag{13}$$

This tells us about the observables statistics by generating the observables cumulants through its derivatives at zero

$$\left.\frac{\partial^n Z}{\partial s^n}\right|_{s=0} = (-1)^n \left\langle O\left(\omega_0^T\right)^n \right\rangle_P. \tag{14}$$

For certain observables or values of s substantially different from 0, many trajectories may make negligible contribution to this expectation, i.e. it is dominated by rare events in the dynamics. To sample these rare events more efficiently, we may thus seek a dynamics corresponding to an ensemble reweighted according to the value of this observable, that is

$$P_W\left(\omega_0^T\right) = \frac{e^{-sO\left(\omega_0^T\right)} P\left(\omega_0^T\right)}{Z(s,T)}. \tag{15}$$

often referred to as the biased or titled ensemble of trajectories. For example, if we wanted to consider the statistics of the area, we would set $o(x_t, x_{t-1}, t) = x_t$ and have

$$W\left(x_t, x_{t-1}, t\right) = e^{-sx_t}, \tag{16}$$

and thus

$$W(\omega_0^T) = e^{-sA\left(\omega_0^T\right)} = \prod_{t=0}^{T} e^{-sx_t}. \tag{17}$$

A particular case of the above is the study of observables in the long time limit. For appropriate observables in many models, the probability of a particular value takes a LD form [3], finding

$$P(O|T) \propto e^{-T\phi\left(\frac{O}{T}\right)}, \tag{18}$$

where $\phi\left(\frac{O}{T}\right)$ is referred to as the rate function, describing the probability of the observable taking a particular value per unit time. In these cases the cumulant generating function additionally has a simplified form, in terms of the SCGF $\theta(s)$

$$Z(s,T) \propto e^{T\theta(s)}. \tag{19}$$

The SCGF $\theta(s)$ is thus often the aim of studies into the long-time statistics of time integrated observables, as it encodes the observables moments. As we will see in section 5, such problems can be considered using a continuing form of RL. In fact, a key result is that $\theta(s)$ ends up being directly related to the quantity we identify as our analogue of the return from RL, the precise quantity we will aim to maximize. Our algorithms thus provide a two-for-one: they both find a dynamics which approximately generates the tilted ensemble, while simultaneously finding a variational approximation to $\theta(s)$.

2.3. Relationship to standard reinforcement learning

Here we will briefly describe how our problem relates to the standard approach to RL. The aim of RL is to achieve some desired objective, by finding the best decisions or **actions** to make given some current information about the situation (the **state** of the environment) in which the objective must be achieved [39]. Actions are chosen within each state according to a **policy**, which influences the transition to the next state. The key ingredient of RL is inspired by behavioural psychology: the objective is encoded in a sequence of **rewards** received for each action in each state. Formally these rewards are assigned real numbers, with the magnitude and sign defining how good or bad a decision is. The resulting construction is referred to as a Markov decision process (MDP). The goal of RL is then simply to maximize the sum of rewards—the **return**—received, thus making the best decisions to achieve the objective: this is done by optimizing the policy according to which actions are taken.

Our problem can be seen as a simplified form of RL in which each 'action' precisely chooses the next state: we can therefore forgo the concept of actions and simply view the problem as choosing the best next state given the current state. The dynamics is thus completely defined by the policy of how the next state is chosen. To connect to RL, it thus remains to define the 'reward' in our problem. A natural suggestion for the return of each trajectory may be the log of the trajectories weight, which naturally produces a sum over terms associated to each transition

$$\ln\, W\left(\omega_0^T\right) = \sum_{t=1}^{T} \ln\, W(x_t, x_{t-1}, t). \tag{20}$$

Maximising the return would thus result in a dynamics which produces trajectories of maximal weight. However, while this is along the right lines, standard RL tends to produce a deterministic policy: in this

case, it would only produce trajectories with the maximum possible weight. Our goal is to approximately reproduce the reweighted trajectory ensemble, producing each trajectory proportional to its weight. This necessarily requires the transitions from each state to be probabilistic. While there are ad hoc approaches to making the learnt policy probabilistic, our key result is that there is in fact a natural way of framing our optimization problem as a **regularized** form of RL, based on the KL divergence. This regularized form is very similar to recent maximum-entropy RL techniques [59–61] and other suggested approaches to regularizing RL [57, 58], however, we are not aware of policy gradient techniques having been considered for the particular form of regularization our problem relates to. This relation to regularized forms of RL will be discussed further in section 3.5.

The most significant tool this connection allows us to take from RL is that of value functions, which naturally emerge in a slightly modified form in this regularized setting. These modified value functions satisfy a Bellman equation, as seen in section 3.3. Value focussed approaches to RL often use this as a starting point, as do some policy focussed approaches. Equally, there exist many techniques, such as pure Monte-Carlo sampling, which make no use of Bellman equations in formulation or algorithmic solution [39]. Further, they are not necessary in the initial introduction to policy-gradient techniques.

While important for our approach, we believe beginning our discussion by introducing both values and the Bellman equations they satisfy will serve to hide the simple connection between rare trajectory sampling and RL under further layers of abstraction. Further, it would result in the rapid introduction of a range of concepts which are not common knowledge within the physics community. As such, we choose to gradually introduce value functions and the Bellman equation as a natural tool in improving a gradient based approach, rather than a foundation, during the pedagogical development of the next section.

3. Gradient optimization of rare finite-time trajectory sampling

In our approach, we seek to search through a space of **parameterized dynamics** $P_\theta(x_t|x_{t-1}, t)$, conditional on the state and time, in order to make the trajectory ensemble it generates with probabilities given by

$$P_\theta(\omega_0^T) = \prod_{t=1}^{T} P_\theta(x_t|x_{t-1}, t)P(x_0), \tag{21}$$

as similar to the reweighted trajectory probabilities of equation (8) as possible. Similarity is defined by the KL divergence between the parameterized trajectory ensemble and the reweighted trajectory ensemble

$$D_{\mathrm{KL}}(P_\theta|P_W) = \sum_{\omega_0^T} P_\theta\left(\omega_0^T\right) \ln\left(\frac{P_\theta\left(\omega_0^T\right)}{P_W\left(\omega_0^T\right)}\right) = \left\langle \ln\left(\frac{P_\theta\left(\omega_0^T\right)}{P_W\left(\omega_0^T\right)}\right)\right\rangle_{P_\theta}, \tag{22}$$

taking value 0 only when the trajectories distributions P_θ and P_W are identical, a measure of similarity discussed in [30] in the context of continuous time. If these trajectory distributions agreed, we would refer to the parameterized dynamics $P_\theta(x_t|x_{t-1}, t)$ as the **optimal dynamics**. We take the expectation over the parameterized dynamics P_θ, since this is precisely what we have access to, and can thus run simulations to sample it. This differs from the approach recently considered for rare continuous-time diffusive trajectories in e.g. [55], where the KL divergence is treated with the distributions reversed: the expectation is taken with respect to the reweighted distribution P_W, with expectation then calculated through importance sampling. In principle, if P_W is contained within the set of parameterized dynamics P_θ, these KL divergences have the same minimum. However, when this is not the case the two perspectives will differ in their optimal dynamics.

We will conduct our search through the space of dynamics by performing gradient descent optimization on the KL divergence (22). We thus require that the parameterized dynamics $P_\theta(x_t|x_{t-1}, t)$ be differentiable with respect to the **weight** θ. We note that, to truly zero out the KL divergence, in general we would also have to parametrise and optimise the initial state distribution, as this will differ from its original form in the reweighted trajectory ensemble. For simplicity, we will forgo including this initial distribution parametrisation and the resulting modifications to the algorithms, but their inclusion is a simple extension to what we will develop.

In the following sections, we will pedagogically demonstrate how to minimize this function efficiently through a line-search gradient descent based approach, following estimates of the gradient of equation (22). Similar to the policy gradient algorithms of RL, and thus referred to as dynamical gradient algorithms in the physical context, the resulting methods are very similar in structure to those found in maximum-entropy RL [59–61], and closely related to current research in regularized MDPs [57, 58]. Following an analogous development to that of [39], we begin with a simple Monte Carlo sampling based algorithm closely related

to [56]. We then introduce an additional function approximation for the ‘value’ of each state, used to guide the dynamical gradient first as a comparative baseline, and then as a bootstrapping estimate, leading to a so-called ‘actor-critic (AC)’ algorithm. In particular, our use of a value function to guide the optimization of the dynamics is a first in approaches focussed on trajectory sampling: this provides a key example of the techniques that can be used due to our connection between trajectory sampling problems and RL. We will not provide proofs of convergence or quality of converged results of the proposed algorithms in this work, however, we will apply several algorithms to a toy model, and reference theoretical results for similar RL algorithms throughout the section.

3.1. Modifying transitions according to futures experienced: Monte Carlo returns

First, for clarity, we rewrite the normalization factor, or ‘partition function’, as

$$Z = \left\langle W\left(\omega_0^T\right)\right\rangle_P. \tag{23}$$

Substituting the definitions of the parameterized trajectory probability (21) and reweighted trajectory probability (8) into the KL divergence (22), we have

$$\begin{aligned} D_{\mathrm{KL}}(P_\theta|P_W) &= \left\langle \sum_{t=1}^{T} \ln\left(\frac{P_\theta(x_t|x_{t-1},t)}{P(x_t|x_{t-1})}\right) - \sum_{t=1}^{T} \ln\, W(x_t,x_{t-1},t) + \ln Z \right\rangle_{P_\theta} \\ &= -\left\langle R(\omega_0^T)\right\rangle_{P_\theta} + \ln Z, \end{aligned} \tag{24}$$

where we have defined the return R of a trajectory as

$$R(\omega_0^T) = \sum_{t=1}^{T} \ln\, W(x_t,x_{t-1},t) - \sum_{t=1}^{T} \ln\left(\frac{P_\theta(x_t|x_{t-1},t)}{P(x_t|x_{t-1})}\right), \tag{25}$$

encoding the contribution of each trajectory to the divergence, weighted by the probability. Clearly, minimization of the KL divergence is analogous to maximization of the expected value of this return, similar to the usual situation considered in RL. However, this differs from standard RL in the explicit dependence on the parameterized dynamics. As a result, in contrast to standard RL where the return associated to each trajectory constant, here the return for a given trajectory changes with the parameterized dynamics. This is the situation more commonly considered in maximum-entropy RL [59–61], where the attempt to maximize a return corresponding purely to the contribution of the weights is regularized by simultaneously trying to maximize the entropy of the trajectory ensemble. For us, maximizing the RL reward is replaced by maximising the log of the weighting, while maximising entropy is replaced by minimizing the KL divergence between the original (non-reweighted) trajectory ensemble and the ensemble of the parameterized dynamics, an objective closely connected to current research in regularized MDPs [57, 58].

For further clarity, we split the return into parts associated to each time step: specifically, we define an overall reward associated to each transition and time as

$$r(x_t,x_{t-1},t) = \ln\, W(x_t,x_{t-1},t) - \ln\left(\frac{P_\theta(x_t|x_{t-1},t)}{P(x_t|x_{t-1})}\right), \tag{26}$$

containing both the weighting and KL divergence contributions, such that the return on subsets of the trajectory is given by

$$R(\omega_{t-1}^{t'}) = \sum_{t''=t}^{t'} r(x_{t''},x_{t''-1},t''). \tag{27}$$

To minimize we will follow gradient descent on this objective, calculating its derivative with respect to the parameters θ: noting

$$\nabla_\theta P_\theta(\omega_0^T) = \nabla_\theta \prod_{t=1}^{T} P_\theta(x_t|x_{t-1})P(x_0) = P_\theta(\omega_0^T)\sum_{t=1}^{T} \nabla_\theta\, \ln\, P_\theta(x_t|x_{t-1},t), \tag{28}$$

$$\nabla_\theta R(\omega_0^T) = -\sum_{t=1}^{T} \nabla_\theta\, \ln\, P_\theta(x_t|x_{t-1},t), \tag{29}$$

we have

$$\nabla_\theta D_{\mathrm{KL}}(P_\theta|P_W) = -\left\langle \left[R(\omega_0^T) - 1\right] \sum_{t=1}^{T} \nabla_\theta \ln P_\theta(x_t|x_{t-1}, t) \right\rangle_{P_\theta}$$
$$= -\left\langle \sum_{t=1}^{T} R(\omega_{t-1}^T)\nabla_\theta \ln P_\theta(x_t|x_{t-1}, t) \right\rangle_{P_\theta}, \quad (30)$$

where in the second line, we have removed the factor of 1 and the return prior to the differentiated time step of each summand, since

$$\sum_{x_t} P_\theta(x_t|x_{t-1}, t)\nabla_\theta \ln P_\theta(x_t|x_{t-1}, t) = \nabla_\theta \sum_{x_t} P_\theta(x_t|x_{t-1}, t) = 0, \quad (31)$$

due to the normalization of $P_\theta(x_t|x_{t-1}, t)$. Written in terms of the return, this takes the exact same form as the negative of the usual policy gradient of RL [39], albeit with a regularized return.

As we will see below, equation (30) forms the basis of algorithms we will consider, as it can be manipulated into a wide variety of useful forms. However, as stated this already provides an immediate algorithmic approach.

The exact value of the gradient specified by the above equation will be impossible to calculate even for simple problems. Instead, since it takes the form of an expectation over trajectories, we can use Monte Carlo sampling of trajectories to construct an estimate, against which we will update the weights, before repeating the process. Suppose we sample a set of N trajectories $\{(\omega_i)_0^T\}_{i=1}^N$ using the current P_θ dynamics, each with partial returns after the state x_t^i of

$$R_{t-1}^i = R\left((\omega_i)_{t-1}^T\right). \quad (32)$$

We can construct an empirical estimate of the gradient as

$$\nabla_\theta D_{\mathrm{KL}}(P_\theta|P_W) \approx -\frac{1}{N}\sum_{i=1}^{N} \left[\sum_{t=1}^{T} R_{t-1}^i \nabla_\theta \ln P_\theta(x_t^i|x_{t-1}^i, t)\right]. \quad (33)$$

We then update the weights by moving a short distance against the gradient, in order to reduce the KL divergence according to this estimate, as

$$\theta_{n+1} = \theta_n + \alpha_n \frac{1}{N}\sum_{i=1}^{N} \left[\sum_{t=1}^{T} R_{t-1}^i \nabla_\theta \ln P_\theta(x_t^i|x_{t-1}^i, t)\right], \quad (34)$$

where α_n is the learning rate for step n. The estimate (33) can be calculated iteratively as each trajectory is created, updating the current average each new trajectory until a desired number has been run to reduce memory requirements. Alternatively, we may even choose to sample a single trajectory between each update

$$\theta_{n+1} = \theta_n + \alpha_n \sum_{t=1}^{T} R_{t-1}\nabla_\theta \ln P_\theta(x_t|x_{t-1}, t). \quad (35)$$

To gain an intuition for these updates, consider each term in the sum of equation (35) individually, along the sample excursion trajectory of four steps sketched out in figure 1(a). The state x_t at each time $t < T = 4$ has an associated return R_t, given by the future rewards, see figure 1(b). Each term in the update (35) then attempts to move the weights to increase or decrease the probability of the occurring transition, depending on the sign of the return: the size of the change is proportional to the magnitude of the resulting return. A more rewarding future leads to a larger increase in transition probability, and vice versa. As many of these updates are committed, competing transitions (those for the same origin state) are then repeatedly enhanced or suppressed according to the resulting returns, leading to an eventual equilibration to a particular balance between the probabilities, depending on the returns that follow them.

Approaching this balance requires consideration of the learning rate α_n: under ideal conditions on the function approximation and sampling, traditional RL convergence is expected provided the learning rate satisfies the requirements of the stochastic approximation

$$\sum_{n=0}^{\infty} \alpha_n = \infty, \qquad \sum_{n=0}^{\infty} |\alpha_n|^2 = c, \quad (36)$$

where c is any finite number [65, 66]. However, convergence is only expected in the limit of infinite updates, and decaying learning rates can often slow learning. In practice, learning rate which decrease (or even

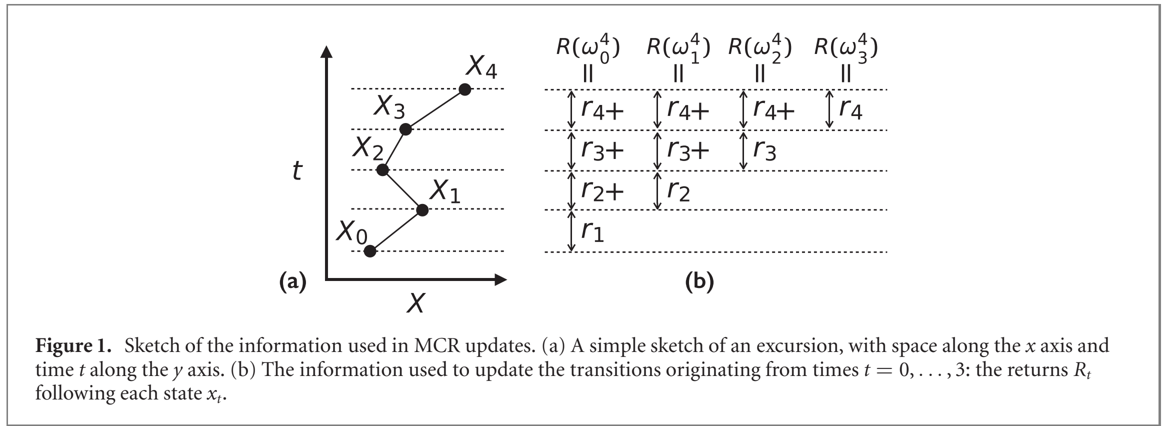

Figure 1. Sketch of the information used in MCR updates. (a) A simple sketch of an excursion, with space along the x axis and time t along the y axis. (b) The information used to update the transitions originating from times $t = 0, \dots, 3$: the returns R_t following each state x_t.

Algorithm 1. KL regularized MCR.

1: **Inputs** dynamical approximation $P_\theta(x_t|x_{t-1}, t)$
2: **Parameters** learning rate α_n; total updates N
3: **Initialize** choose initial weights θ, define iteration variables n and t, total error δ_P
4: $n \leftarrow 0$
5: **Repeat**
6: Generate a trajectory ω_0^T according to the dynamics given by $P_\theta(x_t|x_{t-1}, t)$, with returns R_t after each state x_t.
7: $t \leftarrow 0$
8: $\delta_P \leftarrow 0$
9: **Repeat**
10: $\delta_P \leftarrow \delta_P + R_{t-1}\nabla_\theta \ln P_\theta(x_t|x_{t-1}, t)$
11: $t \leftarrow t + 1$
12: **Until** $t = T + 1$
13: $\theta \leftarrow \theta + \alpha_n \delta_P$
14: $n \leftarrow n + 1$
15: **Until** $n = N$

increase) for a short period at the start of learning, before becoming constant, may be beneficial [39, 67]. For this algorithm, and standard RL algorithms without regularization, a constant learning rate will result in the weights fluctuating around a local minimum; for the KL divergence regularized setting we consider, it in fact turns out that the components used in the algorithms introduced in later sections cause a decay of the gradient to zero, even for individual samples, as optimality is approached [68, 69].

More generically, both update rules described above fall under the umbrella of stochastic gradient descent, where noisy estimates of the gradient are used to update the parameters stochastically [66]. The first of these updates is based on batches of trajectories, sometimes called mini-batches in the ML literatures, while the second is based on single samples.

The algorithm presented in this section is the simplest form of dynamical gradient algorithm, a regularized version of the classical REINFORCE algorithm [40, 41] based on return sampling, and as such we refer to this simply as KL regularized Monte Carlo returns (MCR). For clarity, this algorithm is outlined below in algorithm 1.

3.2. Comparing returns with past experiences: baselines and value functions

A downside of this simple approach is the large potential variance in the return following a transition in each trajectory, which may provide an extremely noisy gradient from which to learn, resulting in slow convergence. Fortunately, equation (30) possesses an invariance which can be used to tame this variability. Recalling how we used (31) to remove the factor of one and the history of the return from (30), we may use this property to instead introduce any desired function of the past trajectory. We introduce the **baseline** $b(x_t, t)$ as simply a function of the state and time, transforming (30) into

$$\nabla_\theta D_{\mathrm{KL}}(P_\theta|P_W) = -\left\langle \sum_{t=1}^{T} \left(R(\omega_t^T, x_{t-1}) - b(x_{t-1}, t-1)\right) \nabla_\theta \ \ln \ P_\theta(x_t|x_{t-1}, t) \right\rangle_{P_\theta}, \tag{37}$$

where the return following each transition is then contrasted with a baseline.

The choice of baseline can have a drastic impact on the variance of the gradient estimate, especially if we consider a small number of trajectories between updates. A reasonable choice of baseline to minimize variance would simply be the average value of the return following a given state at a given time, the

conditional expectation

$$V_{P_\theta}(x,t) = \left\langle R(\omega_t^T)\right\rangle_{P_\theta, X_t=x}, \tag{38}$$

as this would minimize the variance of the baseline error

$$\delta_b(\omega_{t-1}^T, t-1) = R(\omega_{t-1}^T) - b(x_{t-1}, t-1), \tag{39}$$

and therefore might be expected to minimize the variance of the overall gradient estimate. These **state values** encode the combined average weighting for the ensemble of sub-trajectories beginning from x at time t, and KL divergence to the original dynamics of this sub-trajectory ensemble: the higher this value, the higher the average weighting and/or lower the KL divergence of this ensemble relative to that of the original dynamics.

The resulting gradient is given by

$$\nabla_\theta D_{\mathrm{KL}}(P_\theta|P_W) = -\left\langle \sum_{t=1}^{T} \left(R(\omega_{t-1}^T) - V_{P_\theta}(x_{t-1}, t-1)\right) \nabla_\theta \ln P_\theta(x_t|x_{t-1}, t)\right\rangle_{P_\theta}. \tag{40}$$

Unfortunately, this is an ideal which cannot be achieved: calculating the value for each state visited exactly is impossible in most problems of interest. Instead, we introduce a second function approximation for the value function, $V_\psi(x_t, t)$, with weights $\psi \in \mathbb{R}^{d_V}$. The exact error in each of the values provided by this function approximation is then given by

$$L(\psi|x_t, t) = \frac{1}{2}\left(V_\psi(x_t, t) - V_{P_\theta}(x_t, t)\right)^2. \tag{41}$$

Even supposing we had an accurate result for the true value, we could not optimize these state-dependent loss functions one by one, as the resulting approximation would simply be overfitted on the last state optimized: instead, we must consider the states in unison. However, we need not consider them with uniform weighting, and indeed each state will not be equally relevant to a given sampling dynamics and the rare event problem it is being optimized for. The obvious choice for our aim is given by our current sampling dynamics: not only are we likely already using this to approximate the dynamical gradient, it will also prioritize the states which are most likely to occur in the current dynamics, and thus the most important to get accurate values for. We thus sample states according to this dynamics, defining the loss function averaged over trajectories as

$$L_V(\psi) = \left\langle \frac{1}{2}\sum_{t=0}^{T-1} \left(V_\psi(x_t, t) - V_{P_\theta}(x_t, t)\right)^2\right\rangle_{P_\theta}, \tag{42}$$

where the last time is neglected as the value is zero by definition.

Calculating the gradient of this loss, we have

$$\nabla_\psi L_V(\psi) = \left\langle \sum_{t=0}^{T-1} \left(V_\psi(x_t, t) - V_{P_\theta}(x_t, t)\right) \nabla_\psi V_\psi(x_t, t)\right\rangle_{P_\theta}, \tag{43}$$

giving a gradient in terms of the exact value similar to equation (40): to get a target that can be evaluated we simply substitute the definition of the value (38) and use (5) to find

$$\nabla_\psi L_V(\psi) = -\left\langle \sum_{t=0}^{T-1} \left(R(\omega_t^T) - V_\psi(x_t, t)\right) \nabla_\psi V_\psi(x_t, t)\right\rangle_{P_\theta}. \tag{44}$$

As with the dynamical gradient, to estimate the value loss functions gradient (44) we can simply sample one trajectory with states x_t followed by returns R_t leading to

$$\nabla_\psi L_V(\psi) \approx -\sum_{t=1}^{T} \left(R_{t-1} - V_\psi(x_{t-1}, t-1)\right) \nabla_\psi V_\psi(x_{t-1}, t-1). \tag{45}$$

Choosing a baseline $b(x_t, t) = V_\psi(x_t, t)$ then leads to an estimate for the policy gradient equation (40) for a given approximation of the value function, given by

$$\nabla_\theta D_{\mathrm{KL}}(P_\theta|P_W) \approx -\sum_{t=1}^{T} \left(R_{t-1} - V_\psi(x_{t-1}, t-1)\right) \nabla_\theta \ln P_\theta(x_t|x_{t-1}, t). \tag{46}$$

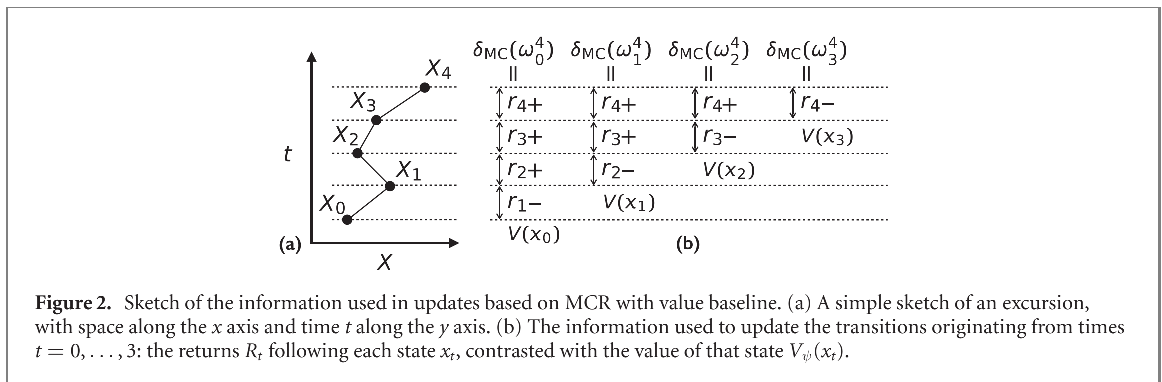

Figure 2. Sketch of the information used in updates based on MCR with value baseline. (a) A simple sketch of an excursion, with space along the x axis and time t along the y axis. (b) The information used to update the transitions originating from times $t = 0, \ldots, 3$: the returns R_t following each state x_t, contrasted with the value of that state $V_\psi(x_t)$.

As in the previous section, we can readily construct empirical averages over multiple trajectories instead of considering single trajectories.

We can get some intuition for how the two approximations affect each other by considering how they affect each others loss functions and updates. By construction, the dynamical gradient is on average independent of the baseline, and thus the optimal weights θ independent of the current values. However, the better the value approximates the true values for the current policy, the smaller the variance in the updates and the faster the dynamics will converge. We would thus desire the values to remain as accurate as possible to the current dynamics. In contrast, the value loss function depends strongly on the dynamics: through the probability of each future trajectory, the priority given to each state, and the reward function itself. The optimal value weights ψ will thus depend strongly on the dynamics, however, for small changes in the policy we would expect a small change in the optimal value weights. If the value function is reasonably accurate, a small change in the dynamics should thus only require a small number of updates to ψ for it to again become accurate.

Accounting for these observations, there is some choice in the usage of updates given by equations (45) and (46). We could simply alternate updating the value function and the dynamics, leaving one fixed while the other changes. This could range from letting the value function converge satisfactorily between updates to the dynamics, to simply alternating updates to the values and dynamics every trajectory. Alternatively, we could use the same trajectory samples to simultaneously update both the dynamics and the values. For a broader discussion of interleaving updates to the dynamics and values we refer to [39], where it is discussed in particular under the terms asynchronous and generalized policy iteration.

The chosen scheme for updating both the values and dynamics in this double-learning scenario can have a significant affect on aspects of algorithm performance such as data efficiency, stability, convergence speed and bias in the final result. For simplicity, we demonstrate using baselines with synchronous updates using a single trajectory for each update. We refer to this as KL regularized Monte Carlo reinforce with a value baseline, due to its similarity to the Monte Carlo REINFORCE algorithm with a value function of RL [39]. Intuitively, for each trajectory we contrast the value of each state with the return following it, cf figure 2(b), aiming to increase both the probability of a transition and the value of a state if the return following it is greater than the value, and decrease them if the return is less. We then conduct updates of the two weights θ and ψ after every trajectory with learning rates α_n^θ and α_n^ψ satisfying equation (36), in the directions suggested by the average of these return-value comparisons. In practice, the efficiency of this algorithm is enhanced by noting that the factor multiplying the gradients in both updates takes the same form

$$\delta_{\mathrm{MC}}(\omega_t^T, t) = R(\omega_t^T) - V_\psi(x_t, t), \tag{47}$$

which we refer to as the Monte Carlo value error. It is outlined below in algorithm 2.

Value baselines in the standard REINFORCE algorithm were considered in the original works on the algorithm [40, 41], but more recent work has proposed that alternative baselines may provide a lower variance in the Monte Carlo setting [70, 71], suggesting possible modifications to the above approach to further improve convergence rates. Despite this, for the algorithms we consider next, it appears that the value baseline may indeed be the best choice [72].

3.3. Replacing returns with past experiences: temporal differences and actor-critic methods

The Monte Carlo error (47), while better than the return alone, still possesses a relatively large variance if the remainder of the trajectory is long, the dynamics highly entropic and the weightings highly variable. Further reduction of this variance would require an alternative to the return for contrast with the states values. To this end, suppose we used many trajectory samples to construct an estimate of the gradient: transitions occurring multiple times will appear with their gradients multiplied by the average return

Algorithm 2. KL regularized Monte Carlo reinforce with value baseline.

1: **Inputs** dynamical approximation $P_\theta(x_t|x_{t-1},t)$, value approximation $V_\psi(x_t,t)$
2: **Parameters** learning rates α_n^θ, α_n^ψ; total updates N
3: **Initialize** choose initial weights θ and ψ, define iteration variables n and t, total errors δ_P, δ_V, individual error δ
4: $n \leftarrow 0$
5: **Repeat**
6: Generate a trajectory ω_0^T according to the dynamics given by $P_\theta(x_t|x_{t-1},t)$, with returns R_t after each state x_t.
7: $t \leftarrow 0$
8: $\delta_P \leftarrow 0$
9: $\delta_V \leftarrow 0$
10: **Repeat**
11: $\delta \leftarrow R_t - V_\psi(x_t,t)$
12: $\delta_P \leftarrow \delta_P + \delta\nabla_\theta \ln P_\theta(x_{t+1}|x_t,t+1)$
13: $\delta_V \leftarrow \delta_V + \delta\nabla_\psi V_\psi(x_t,t)$
14: $t \leftarrow t+1$
15: **Until** $t = T$
16: $\theta \leftarrow \theta + \alpha_n^\theta \delta_P$
17: $\psi \leftarrow \psi + \alpha_n^\psi \delta_V$
18: $n \leftarrow n+1$
19: **Until** $n = N$

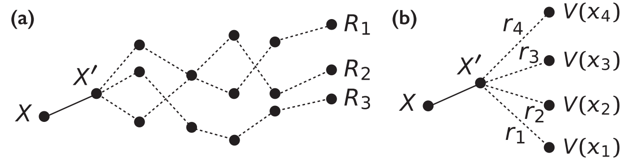

Figure 3. Comparison of updates used in MCR updates and AC updates. Whereas MCR (a) updates a transition $x \to x'$ according to the various possible returns following that transition, the one-step AC (b) update uses knowledge of only the reward during transition and estimates of the values of the states on either side of the transition.

following that transition, cf figure 3(a). Since the first reward is fixed by the transition, this average return would simply be the reward for that transition and the value of the state after transition. This suggests that rather than contrasting the value of the state prior to the transition with the return of a whole trajectory, we could simply contrast the prior state value with the reward associated to that transition, and the estimated value of the resulting state built from past sampled trajectories. If the estimated values are accurate, we would reasonably expect that on average this will result in the same gradients as using returns, cf figure 3(b).

Unsurprisingly, this emerges naturally from the construction considered. Beginning from equation (40) we immediately find

$$\nabla_\theta D_{\mathrm{KL}}(P_\theta|P_W) = -\left\langle \sum_{t=1}^{T} \left(R(\omega_t^T) + r(x_t,x_{t-1},t) - V_{P_\theta}(x_{t-1},t-1)\right) \nabla_\theta \ \ln\ P_\theta(x_t|x_{t-1},t) \right\rangle_{P_\theta}$$
$$= -\left\langle \sum_{t=1}^{T} \left(V_{P_\theta}(x_t,t) + r(x_t,x_{t-1},t) - V_{P_\theta}(x_{t-1},t-1)\right) \nabla_\theta \ \ln\ P_\theta(x_t|x_{t-1},t) \right\rangle_{P_\theta}, \quad (48)$$

where we have used (5) in the second line to replace the future return with the exact value. Since we do not have access to the exact values of each state, we must approximate this expression using a value approximation. Thus, defining a **temporal difference** (TD) error

$$\delta_{\mathrm{TD}}(x_t,x_{t-1},t) = V_\psi(x_t,t) + r(x_t,x_{t-1},t) - V_\psi(x_{t-1},t-1), \quad (49)$$

so-called since it provides the difference between the value of the current state and the reward plus the value of the state at the next time, we have simply

$$\nabla_\theta D_{\mathrm{KL}}(P_\theta|P_W) \approx -\left\langle \sum_{t=1}^{T} \delta_{\mathrm{TD}}(x_t,x_{t-1},t)\nabla_\theta \ \ln\ P_\theta(x_t|x_{t-1},t) \right\rangle_{P_\theta}, \quad (50)$$

which will be accurate whenever the value function is a good estimate for states which are commonly visited by the current dynamics P_θ. In RL, such an approach is referred to as AC, where the dynamics P_θ governing

transitions would be the actor, while the value function V_ψ judges the value of each state, playing the role of critic by informing the actor of whether a transition was good or bad.

For the critic, we could continue to use the Monte Carlo updates of the previous section, using the value function to construct approximate TD errors to update the dynamics. However, the TD errors can also be used to update the critic itself, a process of updating estimates using estimates referred to as **bootstrapping**. Beginning from equation (44), following similar manipulation as that used to reach equation (50), and substituting our approximation for the future value, we quickly arrive at

$$\nabla_\psi L_V(\psi) \approx -\left\langle \sum_{t=0}^{T-1} \delta_{\mathrm{TD}}(x_{t+1}, x_t, t) \nabla_\psi V_\psi(x_t, t) \right\rangle_{P_\theta}, \tag{51}$$

analogous to the basic one-step TD value updates of RL [73]. Clearly, for this to be an accurate approximation the value would already have to be accurate, thus suggesting this estimate would be poor when it matters: for weights ψ which produce inaccurate values. This brings into question how this gradient estimate could ever converge for an initially inaccurate set of weights. Despite this, it often produces very successful results when used for updating the value weights in RL problems.

To understand why, we need to adopt a different perspective. First we note that the exact value function satisfies a natural inductive definition

$$V_{P_\theta}(x_t, t) = \left\langle V_{P_\theta}(x_{t+1}, t+1) + r(x_{t+1}, x_t, t) \right\rangle_{P_\theta, X_t = x_t}, \tag{52}$$

commonly referred to as a Bellman equation, encoding the relationship between the value of state and other states visited in their immediate future. As an alternative to our original choice of loss function (42), using the returns along a trajectory, we could instead directly try to minimize the error in this equation for the approximation to the values. That is, we could minimize the mean-squared Bellman error along a trajectory

$$L_V^{\mathrm{BM}}(\psi) = \left\langle \frac{1}{2} \sum_{t=0}^{T-1} \left(\left\langle V_{P_\theta}(x_{t+1}, t+1) + r(x_{t+1}, x_t, t) \right\rangle_{P_\theta, X_t = x_t} - V_\psi(x_t, t) \right)^2 \right\rangle_{P_\theta}. \tag{53}$$

Taking the derivative of this as is—differentiating both the target expectation and the state sampled—results in a complex gradient to calculate in general: this approach is addressed by so-called gradient-TD algorithms in the RL literature [74–76], which have recently been extended to AC methods [77]. While the unknown stochastic environment presents an additional issue requiring a double sampling of the transitions in that context, in our case the resulting gradient could alternatively be calculated exactly for each state visited, albeit at a substantial computational cost.

To jump from this alternative loss to the gradient of equation (51) requires taking a slightly different view of the Bellman loss. Suppose we instead minimize the distance between the value of each state and a target value predicted by the expectation on the right of equation (52) for the current weights. That is, we keep the weights in the target expectation fixed and only differentiate the value of the state sampled from a trajectory. Differentiating equation (53) with this fixed target and manipulating the expectations then leads directly to equation (51), but with a different interpretation: rather than approximating the gradient of the return based loss function, we are directly targeting an alternative prediction of the value based on the current estimated value of other states. Such an approach is sometimes referred to as a 'semi-gradient' method in the RL literature [39], and has been seen to produce good results provided that the sampling of states is close to that of the dynamics the values are being estimated for, as discussed in more detail later.

To turn this discussion into an algorithm, as before we sample some number of trajectories and then construct estimates of equations (50) and (51): for a single trajectory ω_0^T with TDs $\delta_{\mathrm{TD}}(x_t, x_{t-1}, t)$ associated to transitions from x_{t-1} to x_t at time t, we have

$$\nabla_\psi L_V(\psi) \approx -\sum_{t=1}^{T} \delta_{\mathrm{TD}}(x_t, x_{t-1}, t) \nabla_\psi V_\psi(x_{t-1}, t-1), \tag{54}$$

and

$$\nabla_\theta D_{\mathrm{KL}}(P_\theta | P_W) \approx -\sum_{t=1}^{T} \delta_{\mathrm{TD}}(x_t, x_{t-1}, t) \nabla_\theta \ln P_\theta(x_t | x_{t-1}, t). \tag{55}$$

Intuitively, these updates follow exactly the discussion at the beginning of this section: along each trajectory, the value of each state is contrasted with the value of the state following it plus the reward received in between, cf figure 4(b). If the value of the resulting state combined with the reward is greater than the prior state, a contribution is added to the update which aims to increase the probability of this transition, along

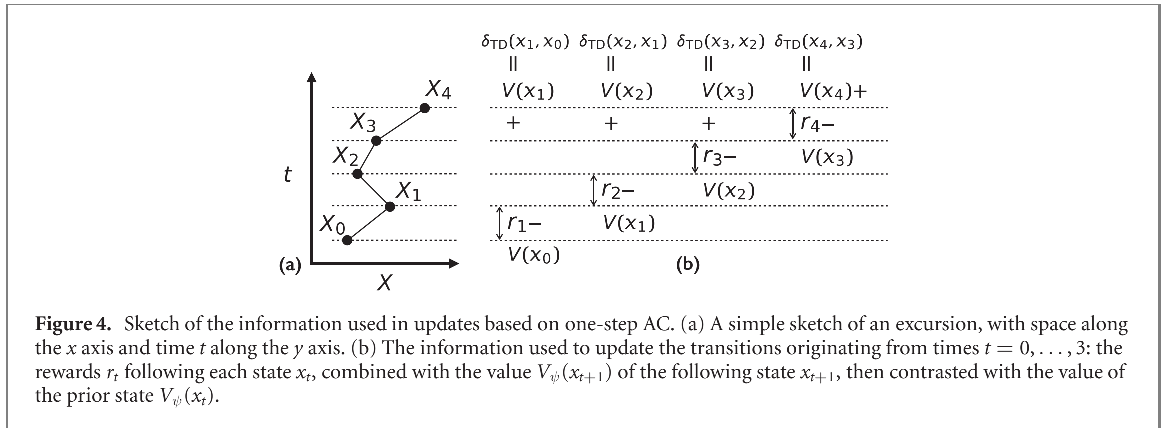

Figure 4. Sketch of the information used in updates based on one-step AC. (a) A simple sketch of an excursion, with space along the x axis and time t along the y axis. (b) The information used to update the transitions originating from times $t = 0, \ldots, 3$: the rewards r_t following each state x_t, combined with the value $V_\psi(x_{t+1})$ of the following state x_{t+1}, then contrasted with the value of the prior state $V_\psi(x_t)$.

Algorithm 3. KL regularized AC.

1: **Inputs** dynamical approximation $P_\theta(x_t, x_{t-1}, t)$, value approximation $V_\psi(x_t, t)$
2: **Parameters** learning rates α_n^θ, α_n^ψ; total updates N
3: **Initialize** choose initial weights θ and ψ, define iteration variables n and t, total errors δ_P, δ_V, individual error δ
4: $n \leftarrow 0$
5: **Repeat**
6: Generate a trajectory ω_0^T according to the dynamics given by $P_\theta(x_t, x_{t-1}, t)$, with rewards $r(x_t, x_{t-1}, t)$ after each state x_{t-1}.
7: $t \leftarrow 0$
8: $\delta_P \leftarrow 0$
9: $\delta_V \leftarrow 0$
10: **repeat**
11: $\delta \leftarrow V_\psi(x_{t+1}, t+1) + r(x_{t+1}, x_t, t+1) - V_\psi(x_t, t)$
12: $\delta_P \leftarrow \delta_P + \delta\nabla_\theta \ln P_\theta(x_{t+1}|x_t, t+1)$
13: $\delta_V \leftarrow \delta_V + \delta\nabla_\psi V_\psi(x_t, t)$
14: $t \leftarrow t+1$
15: **Until** $t = T$
16: $\theta \leftarrow \theta + \alpha_n^\theta \delta_P$
17: $\psi \leftarrow \psi + \alpha_n^\psi \delta_V$
18: $n \leftarrow n+1$
19: **Until**$n = N$

with the value of the prior state; the converse statements hold if the comparison is less. For each trajectory, these contributions are then averaged in an attempt to respect all the corresponding directions.

Actor critic algorithms were among some of the earliest considered for RL, recently returning to favour due to their ease of application to continuous state spaces, improved theoretical convergence properties over purely value focussed approaches, and speed compared with purely return based policy gradient methods. The algorithm 3 presented here is closely related to the recently proposed soft AC algorithm of RL [60], with the key difference being the use of an initial dynamics which is targeted, rather than simply maximising entropy.

In AC algorithms a poor value approximation will clearly lead to poor or even negative changes to the dynamics. One way to address this is by choosing learning rates in such algorithms tuned such that the value function learns faster than the dynamics, in the hope that it always provides a good approximation to the true value function for the current dynamics, and thus a good way of estimating the gradient. So that the value approximation is relatively accurate when updates to the dynamics begin, it may also be good to have a period where only the values are updated for a fixed initial dynamics, such as the original one. Even under these ideal conditions, AC algorithms do not converge to the weights corresponding to local minima of the original loss function (24), but have been shown to end up in a neighbourhood of such minima with high probability for linear function approximations [72].

This unavoidable inaccuracy is a result of the natural bias away from the true gradient introduced by using approximate TD errors. In many RL algorithms, this bias, causing eventual inaccuracy in the final result, is seen as the cost of the substantial reduction in the variance of gradient estimates they produce, allowing for significant improvements in convergence rates.

3.4. Finite horizon example: random walk excursions

We finish this section with a simple example of these techniques in practice, studying the excursion problem outlined in section 2.2.1. While the aim is to generate trajectories for the conditioned ensemble with weights $W(x_T, x_{T-1}, T) = \delta(x_T)$ and $W(x_t, x_{t-1}, t) = H(x_t)$ for $0 < t < T$, due to the zero weight given to

some trajectories, we must use a softened condition given by equations (10) and (11) as a target ensemble to optimize sampling for. This is an exactly solvable problem in the conditioned case, as outlined in appendix A, using a gauge transformation based approach which can in principle also be used calculate the exact optimal dynamics numerically for this simple softened problem. For evaluating how well we are targeting the softened ensemble, we use this same gauge transformation technique to numerically estimate the maximum return as outlined in appendix B. We test all three algorithms currently discussed: MCR shown in algorithm 1, Monte Carlo with a value baseline (MCVB) as in algorithm 2, and AC as outlined in algorithm 3.

For simplicity we start by testing them in a simple 'tabular' setting: that is, we associate a single weight $\theta(x,t)$ to each states transitions, and another single weight $\psi(x,t)$ to each states value for the algorithms which use them. The transition up is then given by this weight in terms of a sigmoid

$$P_\theta(x+1|x,t) = \sigma(\theta(x,t)) = \frac{\mathrm{e}^{\theta(x,t)}}{\mathrm{e}^{\theta(x,t)}+1}, \tag{56}$$

with the probability of transition down then fixed by normalization. The values are simply given by $V_\psi(x,t) = \psi(x,t)$. To perform gradient descent, we need the gradients of these with respect to the weights, simply given by

$$\frac{\partial \ln P_\theta(x\pm 1|x,t)}{\partial\theta(x',t')} = \pm\delta_{xx'}\delta_{tt'}P_\theta(x\mp 1|x,t), \tag{57}$$

and

$$\frac{\partial V_\psi(x\pm 1|x,t)}{\partial\psi(x',t')} = \delta_{xx'}\delta_{tt'}. \tag{58}$$

Note that since each state has an independent weight, as signified by the Kronecker deltas, we can simply update each of these weights independently rather than storing the whole vector of updates.

For evaluation of the dynamics during training, we calculate running averages of three quantities: the expected return, $\langle R\rangle_{P_\theta}$; the success rate, i.e. the probability of generating an excursion

$$\langle S\rangle = \left\langle \delta(x_T)\prod_{t=1}^{T-1}H(x_t)\right\rangle_{P_\theta}, \tag{59}$$

which is simply the expected weighting of the conditioned ensemble; and the entropy of the trajectory ensemble

$$\langle\mathcal{H}\rangle = -\left\langle \ln P_\theta(\omega_0^T)\right\rangle_{P_\theta}, \tag{60}$$

which in this case is a direct measure of the KL divergence between the optimized dynamics and the original dynamics, since $\langle\mathcal{H}\rangle = T\ln 2 - D_{\mathrm{KL}}(P_\theta|P)$. These running averages are calculated using a learning rate and the quantities sampled from each episode: i.e. given a sample O_i of one of the three observables from episode i, we update our average as $\langle O\rangle_i = \langle O\rangle_{i-1} + \alpha_O(O_i - \langle O\rangle_{i-1})$. Observable learning rates are chosen as $\alpha_R = 0.1$, $\alpha_S = 0.003$ and $\alpha_{\mathcal{H}} = 0.01$ for all three algorithms.

Results for these three quantities calculated during the learning process for excursions of length $T = 100$ are shown in figures 5(a)–(c), with AC performing best on all three metrics. In particular, we note that the AC is generally more stable, as it is less likely to get stuck in areas where the gradient of the dynamics is small, i.e. for large values of the potential $\theta(x,t)$. The MC methods are vulnerable to this since they use full returns: initially, these returns may be extremely negative, particularly for earlier states if a trajectory spends a significant amount of time below 0, causing a sudden jump to a very large value of the potential. This can cause the dynamics to become almost deterministic for a long time (cf the beginning of the samples in figure 5(d)); alternatively, the dynamics may get stuck taking incorrect actions such as going below zero for a long time, e.g. causing the initial low success rate for the MCVB training run in figure 5(b).

The slow propagation of information about the reward structure under AC training, one transition back at a time, suppresses these large negative returns early on, causing a greater emphasis on maintaining a high entropy (low KL divergence to the initial dynamics). On the other hand, in this case the MC methods can achieve a higher return earlier by emphasising successfully generating excursions, but struggle to later optimize the entropy, due to the high variance in futures after each transition.

Plots in figures 5(g)–(h) show the upward transition probability, state values and occupation probabilities resulting from the AC training run. The upward probabilities have the expected structure: going upwards from zero, they start at unity probability, reducing to 50–50 along the most commonly visited set of states, and further reducing to 0 as the edge of the backwards lightcone from $x = 0$, $t = 0$ is reached. After $t \sim 50$, transitions upwards are suppressed earlier than the edge of the backwards lightcone, due to the rapidly reducing trajectory entropy that would result from taking further steps upwards. The

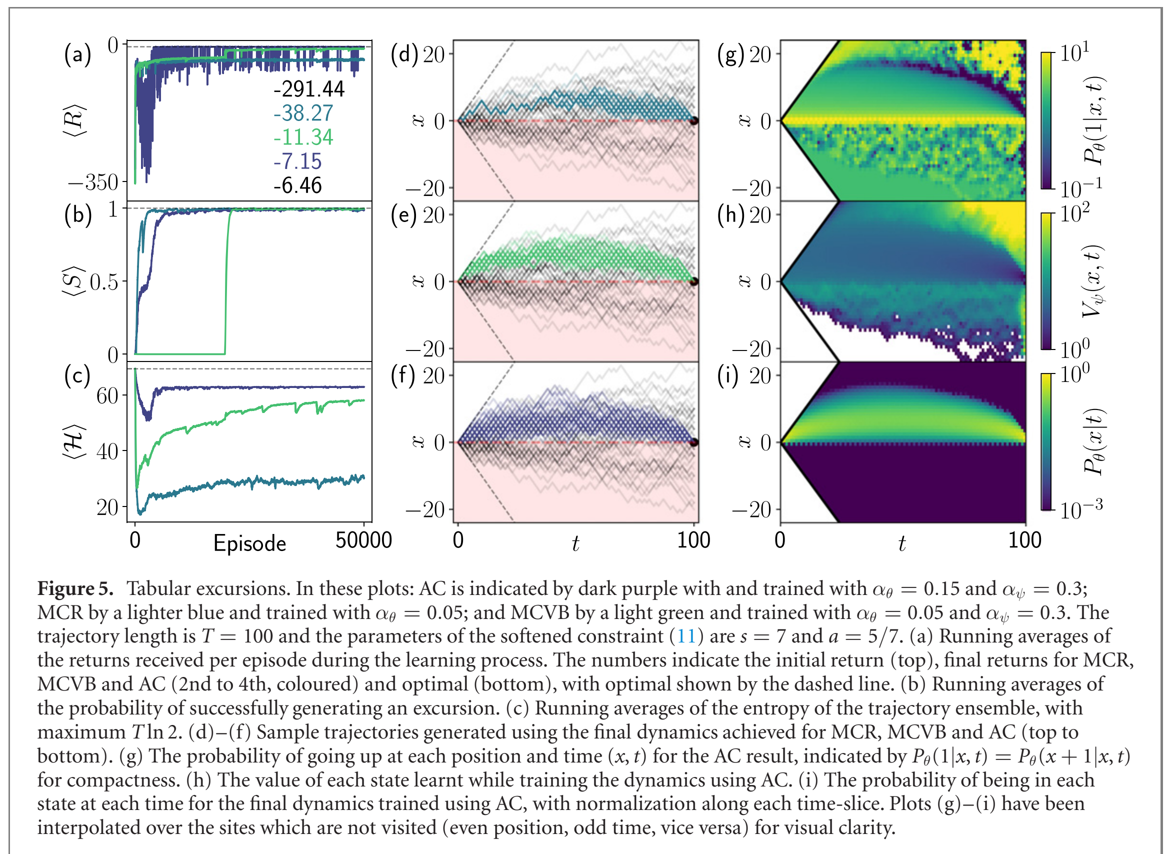

Figure 5. Tabular excursions. In these plots: AC is indicated by dark purple with and trained with $\alpha_\theta = 0.15$ and $\alpha_\psi = 0.3$; MCR by a lighter blue and trained with $\alpha_\theta = 0.05$; and MCVB by a light green and trained with $\alpha_\theta = 0.05$ and $\alpha_\psi = 0.3$. The trajectory length is $T = 100$ and the parameters of the softened constraint (11) are $s = 7$ and $a = 5/7$. (a) Running averages of the returns received per episode during the learning process. The numbers indicate the initial return (top), final returns for MCR, MCVB and AC (2nd to 4th, coloured) and optimal (bottom), with optimal shown by the dashed line. (b) Running averages of the probability of successfully generating an excursion. (c) Running averages of the entropy of the trajectory ensemble, with maximum $T \ln 2$. (d)–(f) Sample trajectories generated using the final dynamics achieved for MCR, MCVB and AC (top to bottom). (g) The probability of going up at each position and time (x, t) for the AC result, indicated by $P_\theta(1|x,t) = P_\theta(x+1|x,t)$ for compactness. (h) The value of each state learnt while training the dynamics using AC. (i) The probability of being in each state at each time for the final dynamics trained using AC, with normalization along each time-slice. Plots (g)–(i) have been interpolated over the sites which are not visited (even position, odd time, vice versa) for visual clarity.

occupation probability, normalized along each time-slice, rises away from these boundaries, peaking at around $x \sim \sqrt{100}$.

Overall, for this example we can see that the resulting increase in the speed of learning more than justifies the theoretical bias induced in the final results by the various steps involved in developing these algorithms, producing results of sufficient accuracy much more quickly.

3.5. Connection to regularized and maximum-entropy reinforcement learning

We now briefly discuss the relationship between the approach presented here and that of maximum-entropy RL [57–61]. In particular, first consider the 'deterministic' RL case, translating from our Markov chains to an MDP by associating each transition to an action, identifying the dynamics with the RL agents policy. Training with maximum-entropy RL is identical to training with our KL regularized algorithms, provided we choose the original dynamics to be that of the maximum-entropy trajectory ensemble, in which every trajectory has the same probability regardless of length, and the weighting is that given by biasing with respect to the reward function.

In the 'stochastic' case, the connection is less clear. Viewing our Markov chain as having a state space which consists of state-action pairs, and decomposing the dynamics into policy and environment components, it may be suspected that maximum-entropy RL can be recovered by choosing the original dynamics to be the one generated by a policy which produces the maximum-entropy trajectory ensemble, up to its ability to control the transitions around the environment. However, this turns out not to be the case: such a policy would necessarily take into account the entropy of the environment resulting from each action, something which standard maximum-entropy RL does not take into account, as this would require incorporating knowledge of the environment probabilities. Maximum-entropy RL in this case is recovered by choosing the original trajectory probabilities to consist of only the contributions of the environment, to each trajectory, normalized as required: it is not immediately clear that this ensemble itself decomposes into a Markovian structure. This distinction may suggest a novel model-based maximum-entropy RL algorithm, in which a known or learnt model is used to further try to maximize the entropy of the trajectory ensemble over considering the policy entropy alone.

4. A universe of algorithms: reviewing variations found in reinforcement learning

The optimization of the KL divergence can be further manipulated in a large number of ways, each corresponding to different algorithms for approximating the gradient. While we will not give an exhaustive

coverage of the possibilities presented in the RL literature, in this section we will review some key variations, translating them into the notation used in this paper. In particular, in section 4.3 we demonstrate how to adapt the algorithms to train neural networks, a powerful form of function approximation. It is hoped this will give the reader an idea of the range of techniques made available by connecting the problem of efficient trajectory sampling with RL. However, we have made later sections independent of this one: those interested in how the approach can be specialized to the long-time limit can skip this section on first reading and instead go to section 5.

4.1. Mixing estimates: expected errors, *n*-step temporal differences and weighted averages

Here we focus on two ways of modifying the AC approach, capable of reducing variance without introducing significant bias: making use of the dynamics to calculate exact expectations of TD errors and gradients associated to transitions for a particular state; and using the Bellman equation to look multiple steps ahead, producing a range of equally valid estimates which can then be averaged.

Firstly, rather than manipulating the value loss into the form shown in equation (51), we could instead use the current dynamics to calculate the expected target for each state visited along a trajectory, as suggested by equation (53), resulting in

$$\nabla_\psi L_V(\psi) \approx -\left\langle \sum_{t=0}^{T-1} \delta_{\mathbb{E}\mathrm{TD}}(x_t,t)\nabla_\psi V_\psi(x_t,t) \right\rangle_{P_\theta}, \tag{61}$$

written in terms of the expected value of the TD error

$$\delta_{\mathbb{E}\mathrm{TD}}(x_t,t) = \langle \delta_{\mathrm{TD}}(x_{t+1},x_t,t)\rangle_{P_\theta, X_t=x_t}, \tag{62}$$

producing updates similar to the expected SARSA algorithm [78].

Unfortunately this error cannot be used for the dynamical gradient, due to the dependence of the transition on the resulting state: however, we can manipulate equation (50) to arrive at

$$\nabla_\theta D_{\mathrm{KL}}(P_\theta|P_W) \approx -\left\langle \sum_{t=1}^{T} \langle \delta_{\mathrm{TD}}(x_t,x_{t-1},t)\nabla_\theta \ln P_\theta(x_t|x_{t-1},t)\rangle_{P_\theta, X_{t-1}=x_{t-1}} \right\rangle_{P_\theta}, \tag{63}$$

where for states sampled along each trajectory we calculate the expected product of the TD error and the gradient of the corresponding transition. This possibility has recently been studied indepth in the RL literature, named variously expected policy gradients and mean actor critic [79–80].

In contrast to updates based on equations (51) and (50), updates using (61) and/or (63) are reasonably expected to have much lower variance than their sampled-transition counterparts, thus resulting in improved convergence without the usual accompanying increase in bias of the final result. The pay-off is a much higher computational demand, in part due to the need to calculate the expectation and the gradients of each transition. Another technicality is the necessity of both updates using different quantities, whereas the updates in algorithm 3 are both built around the same TD errors. It is worth noting that recent work in RL has suggested the possibility of using a mixture of both updates, with the relative weighting varying over time [81]. This may be beneficial when the most likely transitions are to states for which the value is much more accurate, reducing the propagation of errors.

Secondly, we note that the inductive Bellman equation (52) for the exact value can be substituted into itself multiple times, arriving at an *n*-step equation

$$V_{P_\theta}(x_t,t) = \left\langle R\left(\omega_t^{t+n}\right) + V_{P_\theta}(x_{t+n},t+n)\right\rangle_{P_\theta, X_t=x_t}, \tag{64}$$

which inspires an approximate *n*-step TD error similar to the single step errors before

$$\delta_{\mathrm{TDn}}(\omega_t^{t+n},t) = V_\psi(x_{t+n},t+n) + R\left(\omega_t^{t+n}\right) - V_\psi(x_t,t). \tag{65}$$

Similar arguments and manipulation to that done for the one-step TD estimates of the gradients leads to the pair of approximations

$$\nabla_\theta D_{\mathrm{KL}}(P_\theta|P_W) \approx -\left\langle \sum_{t=1}^{T} \delta_{\mathrm{TDn}}(\omega_{t-1}^{t+n},t)\nabla_\theta \ln P_\theta(x_t|x_{t-1},t-1) \right\rangle_{P_\theta}, \tag{66}$$

and

$$\nabla_\psi L_V(\psi) \approx -\left\langle \sum_{t=}^{T-1} \delta_{\mathrm{TDn}}(\omega_t^{t+n},t)\nabla_\psi V_\psi(x_t,t) \right\rangle_{P_\theta}, \tag{67}$$

with values and rewards which would occur at or after the end of the trajectory in the above equation set to zero.

Empirical studies of algorithms based on these errors, simply replacing the TD error in 3 with (65), suggest that each problem has an optimal value of n: larger values result in higher variance errors, while allowing faster propagation of reward information. Values of n greater than the trajectory length recover the Monte Carlo techniques of the previous sections. Their benefit in gradient estimation on their own merits is limited, but as we will see next, they act as a building block in a more powerful estimation scheme.

While TD errors, particularly one-step errors, result in a particularly low variance for the gradient estimates, they can result in slow propagation of information about the reward structure. A large reward occurring on average n steps in the future of a particular transition, would require at least n trajectories for information about that reward to propagate back to that transition, likely many more. In contrast, were we using an n-step error, reward information would propagate more quickly, but result in increased variance of the errors.

A good compromise can be achieved by observing that rather than considering any single one of the possible n-step approximations to the gradient, we could just as justifiably consider a weighted average of them [82, 83]. That is, for some distribution $P(n)$ such that

$$\sum_{n=1}^{T} P(n) = 1, \tag{68}$$

we may consider for the dynamics

$$\nabla_\theta D_{\mathrm{KL}}(P_\theta|P_W) \approx -\left\langle \sum_{t=0}^{T} \delta_{\mathrm{TD}}^{P}(\omega_{t+1}^{T}, x_t, t)\nabla_\theta \ln P_\theta(x_t|x_{t-1}, t) \right\rangle_{P_\theta}, \tag{69}$$

with the weighted error

$$\delta_{\mathrm{TD}}^{P}(\omega_t^T, t) = \sum_{n=1}^{T-t} P(n)\delta_{\mathrm{TDn}}(\omega_t^{t+n}, t), \tag{70}$$

and a similar equation for the value loss gradient. Special cases of the distribution defining this error provide both the Monte Carlo and TD errors discussed previously, however, we can now perform updates according to an equal weighting of the Monte Carlo and one-step errors in each trajectory, or any other distribution we choose. Depending on this choice, we can achieve much faster propagation of information about the reward structure. Further, we can tune the distribution to minimize both the effect of the increased variance inherent in the considering more of the future of each sampled trajectory, and the effect of inaccurate value functions replacing the future.

A common distribution chosen in an attempt to achieve a balance between the variance of longer n-step errors and propagation of reward information is a normalized geometric series

$$P(n) = \frac{\lambda^{n-1}(1-\lambda)}{1-\lambda^T}, \tag{71}$$

which allows for efficient numerical implementation to be achieved by deriving inductive equations relating this return to its value at the next time step.

For completeness, we also note that the expected TD error can be extended in an n-step or λ-weighted form, related to the so-called tree-backup algorithm in RL [84]. Studies of n-step or λ-weighted adaptations of mean actor critic have yet to be conducted.

4.2. Online learning, importance sampling and eligibility traces

In this subsection we briefly discuss a trio of related RL techniques. First, many RL algorithms are designed to be implemented in an online manner, that is, updates may be applied after every transition, not after the end of each trajectory. This allows for experiences during the current trajectory to be used immediately, potentially leading to faster convergence, and as we will see in the next section is essential for infinite-horizon problems where trajectories do not end, rendering Monte Carlo methods impossible.

For a simple heuristic justification of this, note we may rewrite the gradients for the one-step TD approximations as

$$\nabla_\theta D_{\mathrm{KL}}(P_\theta|P_W) \approx -T\left\langle \delta_{\mathrm{TD}}(x_t, x_{t-1}, t)\nabla_\theta \ln P_\theta(x_t|x_{t-1}, t)\right\rangle_{P_\theta}, \tag{72}$$

$$\nabla_\psi L_V(\psi) \approx -T\left\langle \delta_{\mathrm{TD}}(x_t, x_{t-1}, t)\nabla_\psi V_\psi(x_{t-1}, t-1)\right\rangle_{P_\theta}, \tag{73}$$

where we are now viewing the expectation as sampling the triplet of a pair of consecutive states at a particular time, with time is sampled uniformly according to $1/T$. The pair of states are sampled at that time according to the state distribution and transition probabilities of the current dynamics. In reality, we produce correlated samples of this expectation by running trajectories, with the time of each sample being iterated along by one from the previous time. Ignoring technicalities caused by the correlations of the samples generated, from this perspective online algorithms simply apply stochastic gradient descent at the level of individual transitions, rather than individual trajectories.

We do, however, note a subtlety in this viewpoint: by using online updates during the sampling of trajectories, the transitions leading up to the current time are not sampled according to the current dynamical weights, but instead sampled according to the weights at the moment that transition was simulated. Thus, for the heuristic SGD perspective above to be completely valid, we would have to use an importance sampling factor to take into account the true probability of having arrived in the present state under the current dynamical weights. In practice, the small bias this induces is tolerated, as this importance sampling factor would be difficult to implement.

Importance sampling arises more commonly in RL through off-policy methods, in which data is collected using an alternative dynamics to the one being optimized. In this context we must take into account the alternative sampling probabilities twice: reweighting the past to account for the different likelihoods of arriving in a particular state at a particular time, and reweighting the errors themselves to account for the chance of the sampled transition occurring. The later is easy to compensate for, while the former is in principle a complex ratio of historical probabilities. For the values, ignoring the former is equivalent to choosing an alternative prioritization for which states to optimize with respect to. When using the semi-gradient methods described earlier, if this shifted priority differs too substantially from the current dynamics, this can result in a lack of convergence in learning algorithms; if close enough, the dynamics will converge, but be biased further away from the ideal weights [85, 86]. Since the effect in the prioritization of online learning will be minor, this later point is suggestive of the effect this will have on a learning algorithms results: while the weights would be expected to converge, perhaps faster than an offline approach, the end result may be less accurate than the best possible from offline learning.

While true stochastic gradient methods can address the lack of convergence in off-policy sampling [74–77], they do not address the incorrect priority of states. For the dynamics, ignoring the importance sampling ratio for the history is even more detrimental, implying we are not estimating a gradient of the loss function (22) which our main goal it is to minimize. We should therefore handle this lack of emphasis on the correct states in order to reach optimal weights. Off-policy policy gradient techniques are an open area of research in RL [87], however, progress has recently been made through techniques which estimate what the correct emphasis to give states [77, 88]. Despite the bias this emphasis induces in principle, removing it is difficult enough that many state-of-the-art algorithms forgo doing so, accepting any potential reduction in the quality of the final result.

Online learning may be used instantaneously with one-step errors or temporarily delayed for n-step errors. The weighted λ-errors can also be approximately implemented completely online through the use of so-called eligibility traces, closely related to Malliavin weights [89]. These approximate the true λ-error updates, due to the continual drift of the weights away from those associated to the particular transition the λ-error is being calculated for [39, 73, 82, 84, 87]. For linear function approximations this drift can be compensated efficiently, leading to very effective algorithms, however, for general non-linear functions the approximate nature of more general eligibility trace methods can in fact prevent convergence and lead to poor results [90]. It may thus be more desirable to implement λ-errors online by first truncating them to n-steps, then applying delayed updates calculated iteratively for equivalent computational complexity as eligibility trace approaches, at the expense of increased memory requirements [91–93]. However, as we discuss next, even taking this approach may result in instability for common non-linear function approximations.

4.3. Using neural networks: replay buffers and target networks

A powerful function approximation that has found substantial use across academia and industry in recent years is that of neural networks. Unfortunately, while powerful, training them in the straightforward manner described previously often proves to be extremely unstable. This is a consequence of the so-called ‘catastrophic interference’ that neural networks suffer from: their strong adaptability and broad representational power is accompanied by a tendency to forget all but the most recent experiences used in training them. In supervised and unsupervised problems this causes issues in sequentially learning one problem after another, transferring a learned network to a new problem, or when the data distribution is non-stationary in some real-world applications [94–97]. This can be traced back to correlations in the data samples used in training, resulting in non-IID sampling: in sequential or transfer learning, samples are

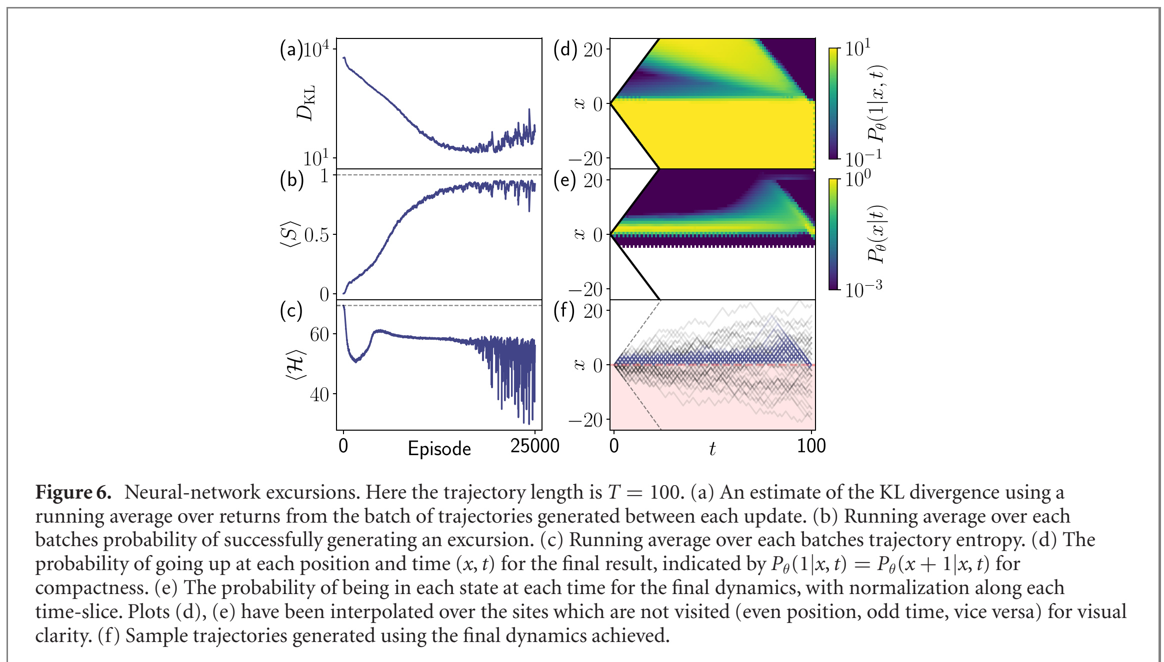

Figure 6. Neural-network excursions. Here the trajectory length is $T = 100$. (a) An estimate of the KL divergence using a running average over returns from the batch of trajectories generated between each update. (b) Running average over each batches probability of successfully generating an excursion. (c) Running average over each batches trajectory entropy. (d) The probability of going up at each position and time (x, t) for the final result, indicated by $P_\theta(1|x, t) = P_\theta(x + 1|x, t)$ for compactness. (e) The probability of being in each state at each time for the final dynamics, with normalization along each time-slice. Plots (d), (e) have been interpolated over the sites which are not visited (even position, odd time, vice versa) for visual clarity. (f) Sample trajectories generated using the final dynamics achieved.

correlated by the simple fact that they belong to one problem or another. While this issue also exists in transferring learned policies and value functions between control problems, in RL, catastrophic interference can in fact occur during training on individual problems, as data is naturally correlated when sampled from trajectories using a Markovian dynamics [98–100]. Often experienced most severely in online training, we even observed this phenomenon during offline training if the samples from a trajectory are strongly correlated, such as in the excursion problem of section 3.4. Further to this, RL is a highly non-stationary problem, with both the state distribution changing whenever the policy is updated, and the targets used in estimating the gradient changing whenever the value function is updated.

As a straightforward demonstration on the simple excursion problem discussed above, we chose to generate batches of 64 trajectories between each update, constructing estimates of both the policy and value gradients using the actor critic algorithm 3, averaging the TD errors for transitions present in the batch of trajectories. We used neural networks with input tuples of (x, t), processed through two 64 neuron hidden layers and one 32 neuron hidden layer for both the policy and value function, with the first two layers followed by a ReLu activation function: for the value function the final layer was linear, while for the policy this was followed by a sigmoid to return a probability between 0 and 1 for transitioning up. Learning rates for both networks were chosen to be a constant $\alpha^\theta = \alpha^\psi = 0.0004$. For the weighting, cf (10) and (11), we used $sb = -50$ reward for transitions to a negative position; for transitions to the final time state, the exponent is modified to a linear dependence on the final position with $s = 500$, $W(x_T, T) = \exp(-500|x_T|)$.

Results of this optimisation are shown in figure 6. Analogous to figures 5 and 6(a)–(c) show running averages of the KL divergence, success rate and trajectory ensemble entropy during the learning process. While the KL divergence remains much larger than the equivalent for the tabular approach, this is largely due to the magnitude of the weight exponents used: for example, note the initial KL divergence is on the order of 10^4, in comparison to order 10^2 for the tabular results, despite beginning at the same maximum entropy dynamics. Although a significant improvement over the original dynamics, the success rate and entropy do not quite achieve the levels seen in the tabular approach, owing to the difficulty in overcoming the instability mentioned above in order to optimize neural networks to a high degree of accuracy. The entropy in particular is lower than desired: the final up-transition probabilities in figure 6(d) show a significant region when the network has learnt to go up at higher values of the position, until the upper edge of the backwards lightcone from the target is reached. This is bordered below by a region where transitions down are almost certain, likely a result of the current dynamics closer to $x = 0$ being more entropic, and thus rewarding, compared to the higher position dynamics which simply goes up to the lightcone edge before going down. The resulting state distribution figure 6(e) (with sample trajectories demonstrated in figure 6(f)) is far more focussed around $x = 0$ than we would hope for, as seen in figure 5(i). These issues likely stem from the large exponents used for the weights, dominating the contribution of the entropy in the KL divergence. This makes it difficult for the learning algorithms to 'see' the entropy past the potentially large negative weight contributions, making optimization of the entropy a slow process which cannot be achieved before the training becomes unstable.

This instability in training neural networks with RL algorithms starts to become pronounced at longer training times, as seen by the noise present at the end of all three learning curves and the increasing value of the KL divergence, even while averaging over a large number of trajectories. More generally, in order to train a neural network, a variety of stabilizing techniques are often used, aimed at suppressing correlations between training samples [60, 101–104]. Typically, two main adaptations are used.

For the non-stationarity of the values used in bootstrapping estimates of the gradients, a third 'target' network is introduced: this is either periodically updated to the current weights of the value network, remaining fixed while the value network is updated in between [101–103], or slowly updated towards the current weights after each update of the value network using an exponential average [60]. However, the instability caused by these moving targets is largely a result of the semi-gradient approximation we made, and can alternatively be addressed instead by using the gradient TD methods [74–77] mentioned in section 3.3, which take into account the change in the target by considering its derivative.

Meanwhile, both the non-stationarity of the state-distribution and the correlation of trajectory-based sampling are partially addressed by the introduction of experience replay [60, 101, 102, 104, 105]: for example, a recent history of experienced transitions are stored in a replay buffer, from which we sample a random set of transitions for use in estimating the gradient. This sampling from the replay buffer reduces correlations between the samples used, as they are no longer sampled sequentially from a trajectory, and slows the change in the state distribution, at the expense of biasing the updates away from their true values for the current weights.

As an example, we now cover the use of experience replay in one-step AC algorithms in more detail. In this case, the basic information we store in the buffer $\mathcal{D}$ are individual transitions (x, t, x'). Rewards are then recalculated using the current dynamics whenever the transition is resampled from the buffer. The bias introduced by experience replay is a result of the differing probabilities of sampling state–state pairs corresponding to each transition, between the distribution of the current dynamics and the distribution of stored in the replay buffer. These probabilities can be decomposed into two parts: the probability of being in the state pre-transition, and the probability of that transition occurring. We can address the later of these easily. If we additionally store the probability μ of each transition at the time it was originally generated, we can multiply its contribution to the gradient when resampled by an importance sampling factor $P_\theta(x'|x, t)/\mu$, removing the resulting bias. The former of these is much more complicated to address, and as such the bias it causes is often accepted in pay off for the benefits of using a replay buffer. However, there exist various techniques which can be used to emphasise states more appropriately in the replay buffer [77, 88]. Given the correction for the transition bias, a gradient estimate is then constructed using a set of N samples $(x_i, t_i, x_i,\ \mu_i)$ randomly taken from the buffer, using

$$\nabla_\psi L_V(\psi) \approx -\sum_{i=1}^{N} \frac{P_\theta(x_i|x_i, t_i)}{\mu_i} \delta_{\mathrm{TD}}(x_i, x_i, t_i) \nabla_\psi V_\psi(x_i, t_i - 1), \tag{74}$$

and

$$\nabla_\theta D_{\mathrm{KL}}(P_\theta|P_W) \approx -\sum_{t=0}^{T} \frac{P_\theta(x_i|x_i, t_i)}{\mu_i} \delta_{\mathrm{TD}}(x_i, x_i, t_i) \nabla_\theta \ln P_\theta(x_i|x_i, t_i), \tag{75}$$

to update the weights.

Despite the limitations of our demonstration in comparison with our earlier tabular results, we believe these could be resolved by better tuning of algorithm parameters and use of the techniques mentioned above. Regardless, it is likely that to apply these techniques to more complex systems neural networks will be extremely useful if not essential. For simple or complex problems, even if the optimal dynamics cannot be reached, the resulting dynamics could be combined with techniques such at TPS to efficiently gather accurate statistics of the rare trajectories of interest.

Finally, we mention that while eligibility traces are powerful when used with tabular methods or linear approximations, the lack of ability to train neural networks using incremental data hinders their use. To this end, recent work has been done considering truncated λ returns [92, 93], and their reconciliation with experience replay [106].

4.4. Further variations

We briefly mention a variety of other possibilities from the RL literature to approach optimizing such problems:

- All algorithms described above are based on stochastic gradient descent, a commonly used line-search gradient method. Recently, RL algorithms have been developed based on natural gradients [72, 107–110], where the updates are modified to respect that changing the parametrization of the

dynamics, while leaving the manifold of possible dynamics invariant, should leave the gradient updates invariant. These are closely related to recent applications of trust-region based gradient methods to RL [69, 111–113], where the learning rates for updates are tamed in order to try and ensure updates do not overshoot and cause a negative change to the dynamics.

- As value functions are learnt from early experiences, transitions towards states that are currently estimated to be higher value will be increased, even if these states are in reality suboptimal, a problem referred to as maximization bias. A common solution to this is the use of double learning, where two value functions are learnt [60, 114, 115]. For each state visited, the value function which produces the lower estimate is then used in estimates of the dynamics gradients.
- When the action space is continuous, the MDP problem can be rephrased as learning a function approximation which generates an action, with inputs as the state and some random noise [60, 103, 116]. This leads to policy gradient estimate which takes into account how the target value changes when when the action parametrization changes, resulting in a lower variance estimate. This will be directly relevant to rare trajectory problems with continuous state spaces and an uncountable number of transitions, and is closely related to current optimal force learning approaches in diffusive problems [56].

An alternative but closely related adaptive approach is based on gauge transformations [32]. While there are simpler derivations, see appendix A, to see this connection note we may rewrite equation (24) as

$$D_{\mathrm{KL}}(P_\theta|P_W) = \sum_{t=0}^{T}\sum_{\omega_0^{t-1}} P_\theta(\omega_0^{t-1}) D_{\mathrm{KL}}\left(P_\theta(-|x_{t-1},t)\,\middle|\,\frac{W(-,x_{t-1},t)P(-|x_{t-1})g(-,t)}{g(x_{t-1},t-1)}\right), \tag{76}$$

where

$$g(x_t,t) = \mathbb{E}_{x_{t+1}\sim P}\left[W(x_{t+1},x_t,t+1)g(x_{t+1},t+1)\right], \tag{77}$$

with $g(x_T,T)=1$ is the inductive equation defining the gauge transformation g, with expectation taken over the original dynamics. Since minimizing each of these KL-divergences individually provides the exact solution, the optimal dynamics is given by the correct gauge transformation, and an alternative approach may be to approximate this gauge transformation directly. This approach has a long history in the mathematical literature [25, 26, 28, 117], and as exact solutions to some MDPs with deterministic environments [29]. Further, this has recently been adapted to diffusion processes [16]. It has also been discussed recently in the context of understanding RL from a statistical physics perspective [118]. From the RL perspective, these algorithms are all based on one-step TD methods, where equation (77) is viewed as a non-linear Bellman equation [119]. This approach could in future be developed into a broader set of RL algorithms which have more in common with the value-function based methods of RL, as opposed to the policy-gradient-like methods presented in this work.

5. Long time dynamics, large deviations and discounting

In many problems of relevance to physical sciences we are interested in the behaviour at long times, such that the system is in its stationary state, be it equilibrium (as in a system in contact with a thermal bath) or not (as in driven systems). Such situations where dynamics is time-homogeneous and the relevant times exceed those set by all relaxation rates, pertain to the regime of dynamical LDs [2, 3, 6, 33], an approach akin to equilibrium statistical mechanics for quantifying the statistical properties of long-time dynamics. For this kind of problem we can specialize our methods above to allow for solutions using genuine, infinitely long trajectories.

To consider these problems, for simplicity we restrict to cases where the original dynamics is time-independent, although the approach may be adapted to periodic dynamics. We can then consider the stationary state of some parameterized dynamics $P_\theta(x_t|x_{t-1})$, a probability distribution $P_\theta^{\mathrm{ss}}(x)$ such that

$$P_\theta^{\mathrm{ss}}(x) = \sum_{x'} P_\theta(x|x')P_\theta^{\mathrm{ss}}(x'). \tag{78}$$

For clarity, we will focus on systems with ergodic dynamics. Put simply, this means that for any pair of states, there exists a sequence of transitions which leads from either one to the other. For us, this means that there is a unique stationary state.

A common approach to studying such models is to consider long but finite trajectories, then use a method such as TPS to sample the reweighted ensemble. While we could take a similar approach using our adaptively learnt dynamics, either with or without TPS, the trajectory lengths may need to be extremely

long to achieve accurate results, and for a generic problem the length required is unknown. It may instead be desirable to directly study the infinite-horizon case, removing fears of incorrect results caused by finite-time effects. However, as it stands there are several problems with the algorithms presented ealier in section 3 for studying problems formulated with an infinite-horizon. In particular, the algorithms we detailed were 'offline', that is, they waited for trajectories to end before learning occurred: clearly in an infinite-horizon context where there is no end to a trajectory, we must necessarily use an online approach, as discussed in section 4.2.

There is a second, more substantial issue: as currently defined, the returns, and thus the resulting values, could diverge to infinity as the trajectory continues to run. Moreover, the value of each state would be almost identical even for sufficiently long but finite futures, as it would be dominated by the average return following states sampled from the stationary state distribution. The origin of these issues can be attributed to the fact that we provide equal emphasis to the value of a state for transitions which occur at any time in the future: for an ergodic system in which any correlation with the current state will eventually be lost, such a definition of value ignores the eventual independence of future states and transitions on the present state being valued.

In this section we will consider a pair of adaptations which remedy this failing of the finite-time value, so that online algorithms can be developed for the infinite-horizon case. First, we will discuss the differential returns and relative values arising from the average-return formulation of RL; second, we will introduce an approximate scheme based on discounting, which nonetheless can improve learning speed by reducing variance, at the expense of accuracy in the final result.

5.1. Comparing rewards with the average: differential returns and values

For RL problems involving an infinite-horizon, one choice of formulation, sometimes argued to be the correct formulation over the traditional one based on discounting [39, 120], is that of time-averaged returns [67, 121–123]. For us, this approach begins by reconsidering our loss function. In the continuing case, under the conditions of time-independence and ergodicity we mentioned in the previous section, there is no particular special time, such as when the trajectory is initialized. As such, the time averaged KL divergence is simply given by a steady state average of rewards on the next transition

$$\begin{aligned} d_{\mathrm{KL}}(P_\theta|P_W) &= \lim_{T\to\infty}\frac{1}{T}D_{\mathrm{KL}}(P_\theta|P_W) \\ &= -\lim_{T\to\infty}\frac{1}{T}\left[\sum_{\omega_0^T}P_\theta(\omega_0^T)R(\omega_0^T) - \ln\, Z\right] \\ &= -\sum_{x,x}P_\theta^{\mathrm{ss}}(x)P_\theta(x|x)r(x,x) + z, \end{aligned} \tag{79}$$

where we have simply defined

$$z = \lim_{T\to\infty}\frac{1}{T}\,\ln\, Z, \tag{80}$$

and $r(x,x)$ is the time-independent reward associated to this transition

$$r(x',x) = \ln\, W(x',x) - \ln\left(\frac{P_\theta(x'|x)}{P(x'|x)}\right). \tag{81}$$

For later clarity, we define the time-averaged return as

$$\bar{r}_\theta = \lim_{T\to\infty}\frac{1}{T}\sum_{\omega_0^T}P_\theta(\omega_0^T)R(\omega_0^T) = z - d_{\mathrm{KL}}(P_\theta|P_W). \tag{82}$$

As we will discuss further in section 5.4, z is related to the SCGF which is often of interest in LD studies. The connection between z and the average reward thus means our algorithms provide an estimate of the SCGF in the process of optimizing the dynamics.

While not immediately obvious from equation (79), the gradient of $d_{\mathrm{KL}}(P_\theta|P_W)$ can infact be written in terms of only the gradient of $P_\theta(x'|x)$, without reference to the gradient $P_\theta^{\mathrm{ss}}(x)$: that this is possible essentially follows from the fact that the steady state is defined by the dynamics. This is extremely useful numerically, as while the gradient of the stationary state may be extremely difficult to construct, the gradient of the transition probabilities is directly accessible using our approximation. However, to see this form of the gradient of equation (79) clearly, we must first define values in this continuing setting.

In order to construct useful values for states in the continuing case, we consider returns defined relative to the average of equation (82): that is, we define the differential return

$$\begin{aligned} R_{\mathrm{D}}(\omega_0^T) &= R(\omega_0^T) - T\bar{r}_\theta \\ &= \sum_{t=1}^{T} r(x_t, x_{t-1}) - \bar{r}_\theta. \end{aligned} \tag{83}$$

We can then consider the value of a state to be the difference between the average return following that state, and the average return following a state drawn from the stationary distribution, simply given by the average of differential returns following that state

$$V_{P_\theta}(x_0) = \lim_{T\to\infty} \langle R_{\mathrm{D}}(\omega_0^T) \rangle_{P_\theta, X_0 = x_0}, \tag{84}$$

where the limit is now convergent, as seen in the next section. In particular, we may relate these values iteratively in a Bellman equation as

$$V_{P_\theta}(x') = \sum_x P_\theta(x|x') \left[V_{P_\theta}(x) + r(x,x') - \bar{r}_\theta \right], \tag{85}$$

which can be simply rearranged to give an alternative equation for our time-averaged KL divergence

$$d_{\mathrm{KL}}(P_\theta|P_W) = z - \sum_x P_\theta(x|x') \left[V_{P_\theta}(x) + r(x,x') - V_{P_\theta}(x') \right], \tag{86}$$

which we note holds for all x'.

We can now write the gradient of our loss as

$$\begin{aligned} \nabla_\theta d_{\mathrm{KL}}(P_\theta|P_W) = &- \sum_x \nabla_\theta P_\theta(x|x') \left[V_{P_\theta}(x) + r(x,x') - V_{P_\theta}(x') \right] \\ &- \sum_x P_\theta(x|x') \left[\nabla_\theta V_{P_\theta}(x) - \nabla_\theta V_{P_\theta}(x') \right]. \end{aligned} \tag{87}$$

Since this equation holds for all x', we are free to average the right-hand side over the stationary state

$$\begin{aligned} \nabla_\theta d_{\mathrm{KL}}(P_\theta|P_W) = &-\sum_{x,x'} P_\theta^{\mathrm{ss}}(x') \nabla_\theta P_\theta(x|x') \left[V_{P_\theta}(x) + r(x,x') - V_{P_\theta}(x') \right] \\ &- \sum_{x,x'} P_\theta^{\mathrm{ss}}(x') P_\theta(x|x') \left[\nabla_\theta V_{P_\theta}(x) - \nabla_\theta V_{P_\theta}(x') \right] \\ = &-\sum_{x,x'} P_\theta^{\mathrm{ss}}(x') \nabla_\theta P_\theta(x|x') \left[V_{P_\theta}(x) + r(x,x') - V_{P_\theta}(x') \right] \\ &- \sum_x P_\theta^{\mathrm{ss}}(x) \nabla_\theta V_{P_\theta}(x) + \sum_{x'} P_\theta^{\mathrm{ss}}(x') \nabla_\theta V_{P_\theta}(x'), \end{aligned} \tag{88}$$

where by using the definition of the stationary state and the normalization of the transition probabilities, the last two terms are seen to be equal. Rewriting the gradient using $\nabla f = f\,\nabla \ln f$ we arrive at a quantity that can be sampled using transitions from trajectories

$$\nabla_\theta d_{\mathrm{KL}}(P_\theta|P_W) = -\sum_{x,x'} P_\theta(x|x') P_\theta^{\mathrm{ss}}(x') \left[V_{P_\theta}(x) + r(x,x') - V_{P_\theta}(x') \right] \nabla_\theta \ln P_\theta(x|x'), \tag{89}$$

which depends only on the gradient of the transitions.

This derivation has naturally left us with a baseline of the exact value function: the second value function term in this equation could be removed by conducting the sum over x. Indeed, if we introduce a baseline of $\bar{r}_\theta$ for all states, then the term in the bracket is the TD error resulting from rearranging equation (85). The gradient is then already in the form of those considered for the AC algorithms, with the critic in this case still providing the perfect values of each state.

To arrive at a functioning algorithm, we must again introduce a learnt critic. We do this as before: we target the true values V_{P_θ} with an approximation V_ψ, with a loss function given by the error in the Bellman equation (85) averaged over the stationary state

$$L_V(\psi') = \sum_{x'} P_\theta^{\mathrm{ss}}(x') \frac{1}{2} \left[\sum_x P_\theta(x|x') \left[V_\psi(x) + r(x,x') \right] - \bar{r}_\theta - V_{\psi'}(x') \right]^2, \tag{90}$$

Algorithm 4. KL regularized differential AC.

1: **Inputs** dynamical approximation $P_\theta(x, x')$, value approximation $V_\psi(x)$
2: **Parameters** learning rates α_n^θ, α_n^ψ, α_n^R; total updates N
3: **Initialize** choose initial weights θ and ψ, initial average $\bar{r}$, define iteration variable n, individual error δ
4: $n \leftarrow 0$
5: **Repeat**
6: Generate a transition from x' to $x = \{x, F(x, x')\}$ according to the dynamics given by $P_\theta(x, x')$.
7: $\delta \leftarrow V_\psi(x) + r(x, x') - \bar{r}_n - V_\psi(x')$
8: $\theta \leftarrow \theta + \alpha_n^\theta \delta \nabla_\theta \ln P_\theta(x|x')$
9: $\psi \leftarrow \psi + \alpha_n^\psi \delta \nabla_\psi V_\psi(x')$
10: $\bar{r} \leftarrow \bar{r} + \alpha_n^R \delta$
11: $n \leftarrow n + 1$
12: **Until** $n = N$

noting that the target from the right of the Bellman equation is fixed to the current weights ψ, taking a semi-gradient approach. The gradient evaluated at the current weights ψ is then

$$\nabla_\psi L_V(\psi) \approx -\sum_{x,x'} P_\theta(x|x') P_\theta^{\mathrm{ss}}(x') \left[V_\psi(x) + r(x, x') - \bar{r}_\theta - V_\psi(x')\right] \nabla_\psi V_\psi(x'), \tag{91}$$

the same as equation (51) up to the negation of the average off of the reward at each transition.

An added complexity comes from the presence of this average return, as both gradient estimates still assume we know the average exactly, which will almost certainly not be true. We must therefore also estimate this average return during our optimization. To do this, we could simply use the stochastic approximation with the rewards sampled over time. Were the dynamics fixed, this would eventually converge to the correct value; for dynamics that are optimized over time, this will continually chase the current value of the average, similar to how the weights of the value function chase the optimal weights for the current dynamics. However, we can speed up convergence, admittedly to a less accurate result, by using the TD error.

More precisely, we can rearrange the Bellman equation and average to get

$$\bar{r}_\theta = \sum_{x,x'} P_\theta^{\mathrm{ss}}(x') P_\theta(x|x') \left[V_{P_\theta}(x) + r(x, x') - V_{P_\theta}(x')\right], \tag{92}$$

which we can sample directly by running trajectories with the current dynamics. Replacing the exact values with our current estimates, we can then update our estimate of the average $\bar{r}_n$ every time a transition occurs, e.g. from x' to x, as

$$\bar{r}_{n+1} = \bar{r}_n + \alpha_n \left[V_\psi(x) + r(x, x') - \bar{r}_n - V_\psi(x')\right]. \tag{93}$$

To make a functioning algorithm, we then replace $\bar{r}_\theta$ in the above gradient estimates for the dynamical and value approximations with our current estimate $\bar{r}_n$.

With the equations (89), (91) and (93) in these forms, the updates for all three components—the dynamical weights θ, the value weights ψ, and the approximate $\bar{r}$—can be estimated using the same TD at each step, namely

$$\delta_{\mathrm{DTD}}(x', x) = V_\psi(x) + r(x, x') - \bar{r}_n - V_\psi(x'), \tag{94}$$

where the subscript DTD stands for 'differential temporal difference'. The online algorithm 4 based on this average construction, updating the two weights and the average at every transition, is stated below. Removing the components related to the average in this algorithm will provide an online algorithm which could easily be applied in the finite-horizon case.

As discussed in section 4.2, online algorithms introduce two issues. First, with the evolving weights, we almost certainly are not sampling the current stationary state of the dynamics: however, if the dynamics evolves slowly enough, the sampling is likely very similar, and certainly close enough to be confident of convergence. Second, the samples we get are not uncorrelated, like we would ideally have in constructing an empirical mean. For simple function approximations this is not an issue if correlations between samples decay quickly enough, however, as mentioned in section 4.3, for more powerful function approximations such as neural networks this can cause instability.

This algorithm, and the one discussed in the next section, can be extended in many of the ways previously discussed in section 4. Further, it can be manipulated to an approximate form which more closely matches the non AC algorithms of section 3. To see this, we consider modifying the algorithm to use an n-step update with extremely large n: in this case, the value function can be removed, as its contribution from the target n step state averages out over the stationary state to zero when n is sufficiently large. The resulting algorithm is equivalent to a continuing version of algorithm 2. Further, the current state value is

simply a baseline which can be removed, producing an algorithm equivalent to that used in [56], a continuing version of algorithm 1. This provides an approximate, value-free algorithm for the continuing case. Alternatively, the algorithm in [56] can be seen as making a finite time approximation to the problem itself, using algorithm 1 of the previous section with an additional average reward baseline.

5.2. An approximate approach: discounting

The more traditional approach in RL for continuing problems gets round the issue of divergent returns by discounting the contribution of rewards to the value of a state proportional to how long after the state the reward was given. That is, the value of a state is defined as

$$V^{\gamma}_{P_\theta}(x) = \lim_{T\to\infty}\left\langle \sum_{t'=t}^{T} \gamma^{t'-t} r(x_{t+1}, x_t) \right\rangle_{P_\theta, X_t = x}, \tag{95}$$

which is convergent for a discount rate γ less than 1.

For these values to be correct, the discounting must be introduced in the original definition of the problem: in this case, the interpretation of the discount is a probability of the system entering an absorbing state in which it receives no more reward [110]. Sampling states correctly then takes us back to a finite trajectory based approach, where we initialize according to some distribution, and end the trajectory at some variable time with probability $1-\gamma$ at each time step, causing infinite trajectories to be exponentially suppressed.

While this may be an interesting problem in its own right, this is not the problem we are aiming to solve. Instead, we introduce discounted values as an approximate approach to estimating the dynamical gradient for the average return problem outlined in the previous section. This allows us to cease tracking the average return, while often providing lower variance estimates for the gradient, at the expense of accuracy in the final result.

For this approximate approach to produce reasonable accuracy of the final result, theoretical work in the RL literature has suggested that the discount rate γ must be such that $1/(1-\gamma)$—the time-scale for the average time between transitions to the absorbing state—is larger than the mixing time of the current dynamics P_θ [110, 121, 124, 125].

To gain an intuition for why discounting works for large enough values, lets consider a slightly modified definition of the differential values. Truncating our earlier definition up to a finite time, we use the return up to that time averaged over time and an initial stationary distribution

$$\bar{r}^T_\theta = \frac{1}{T}\left\langle R(\omega_0^T)\right\rangle_{P_\theta} = \sum_{x_0} P^{\mathrm{ss}}_\theta(x_0)\left\langle R(\omega_0^T)\right\rangle_{P_\theta, X_0 = x_0}, \tag{96}$$

where $\lim_{T\to\infty}\bar{r}^T_\theta = \bar{r}_\theta$. We negate this average off the reward at each step to define our truncated differential values, finding

$$\begin{aligned} V^T_{P_\theta}(x_0) &= \left\langle R(\omega_0^T)\right\rangle_{P_\theta, X_0=x_0} - T\bar{r}^T_\theta \\ &= \sum_{t=1}^{T}\sum_{x_t, x_{t-1}} P_\theta(x_t, x_{t-1}) r(x_t, x_{t-1})\left[P_\theta(x_{t-1}|x_0) - P^{\mathrm{ss}}_\theta(x_{t-1})\right], \end{aligned} \tag{97}$$

where in the second line we have split the returns in to reach reward, summing over the possible paths up to each pair, with $P_\theta(x_{t-1}|x_0)$ used to represent the probability of reaching x_{t-1} under P_θ by any path initiated from x_0. Introducing an importance sampling factor, we may then rewrite the value function in terms of a return in which the rewards depend on the state being valued: given

$$R'(\omega_0^T) = \sum_{t=1}^{T} r(x_t, x_{t-1})\frac{P_\theta(x_{t-1}|x_0) - P^{\mathrm{ss}}_\theta(x_{t-1})}{P_\theta(x_{t-1}|x_0)}, \tag{98}$$

we have

$$V^T_{P_\theta}(x_0) = \left\langle R'(\omega_0^T)\right\rangle_{P_\theta, X_0=x_0}. \tag{99}$$

While this equation requires no knowledge of the average return, it does require extremely detailed knowledge of the probabilities of states conditioned on states multiple steps in the past, something not easily accessible. However, this form makes it transparent that by negating the average return, we are essentially decaying out the contribution of rewards received many steps in the future, in a fashion reminiscent of discounting: since we assume ergodicity, as the time after valuation extends into the future the conditional probability will converge to the stationary sate.

To see this decay we use a spectral decomposition of an operator which describes the evolution of probability distributions under the dynamics P_θ. Viewing $P_\theta(x|x')$ as the components of a transition matrix describing the evolution of a probability distribution

$$\mathcal{W}_\theta = \sum_{x,x'} P_\theta(x|x')|x\rangle\langle x'|. \tag{100}$$

This matrix can be diagonalized, resulting in left $\langle l_i|$ and right $|r_i\rangle$ eigenvectors

$$\langle l_i| = \sum_x l_i(x)\langle x|, \qquad |r_i\rangle = \sum_x r_i(x)|x\rangle, \tag{101}$$

which are orthogonal, $\langle l_i|r_j\rangle = \delta_{ij}$, with eigenvalues λ_i satisfying $\langle l_i|\mathcal{W}_\theta = \lambda_i\langle l_i|$ and $\mathcal{W}_\theta|r_i\rangle = \lambda_i|r_i\rangle$. The stationary state satisfies $\mathcal{W}_\theta|P^{\mathrm{ss}}_\theta\rangle = |P^{\mathrm{ss}}_\theta\rangle$, corresponding to an eigenvalue of 1, with associated left eigenvector the 'flat' state $\langle -|$ with value 1 for every component. It can further be shown that all eigenvalues will satisfy $|\lambda_i| < 1$, since we are assuming the model is ergodic and thus has a single stationary state.

Given this spectrum, we may expand the time evolution of a given initial probability distribution as

$$|P(t)\rangle = \mathcal{W}^t_\theta|P(0)\rangle = |P^{\mathrm{ss}}_\theta\rangle + \sum_{i=2}^{D} \lambda_i^t|r_i\rangle\langle l_i|P(0)\rangle, \tag{102}$$

where D is the dimension of the state space. This allows us to rewrite the probabilities $P_\theta(x_{t-1}|x_0)$ in a spectral expansion, by taking as our initial distribution $|P(0)\rangle = |x_0\rangle$ and projecting out the x_{t-1} component

$$P_\theta(x_{t-1}|x_0) = P^{\mathrm{ss}}_\theta(x_{t-1}) + \sum_{i=2}^{D} \lambda_i^{t-1} r_i(x_{t-1}) l_i(x_0). \tag{103}$$

Finally, substituting this into our alternative equation for the truncated values, we have

$$R'(\omega_1^T, x_0) = \sum_{t=1}^{T} r(x_t, x_{t-1}) \frac{\sum_{i=2}^{D} \lambda_i^{t-1} r_i(x_{t-1}) l_i(x_0)}{P_\theta(x_{t-1}|x_0)}. \tag{104}$$

Recalling $|\lambda_i| < 1$ for $i \neq 1$, all terms in this sum decay as time increases, and thus later rewards contribute less and less to the differential return. For later times this decaying contribution is dominated by the leading eigenvalue of the master operator, the inverse of the relaxation time of the Markov chain, with the denominator becoming the stationary distribution

$$R'(\omega_1^T, x_0) \approx \sum_{t=1}^{T} r(x_t, x_{t-1}) \frac{\lambda_2^{t-1} r_2(x_{t-1}) l_2(x_0)}{P^{\mathrm{ss}}_\theta(x_{t-1})}. \tag{105}$$

This is suggestive of the form of return used when discounting, with some similarity between λ_2 and the discount γ: indeed, the mixing time, which $1/(1-\gamma)$ must be less than for accuracy, is closely related to the relaxation time of the dynamics given by $1/(1-\lambda_2)$.

Replacing all of the above probabilities with a general discounting factor is clearly an approximation of the true differential values, and thus introduces a bias in the final results. However, it removes the need to track the average return in order to estimate the TDs, which can itself introduce errors and bias into the optimization. Discounting can also lower variance of the gradient estimate, as discounting reduces the impact of stochasticity by giving less weight to the further future. As such, we now detail how to use discounted values to guide the evolution of the dynamical weights.

To optimize an approximation for the discounted values, we note that the values of equation (95) satisfy a slightly modified Bellman equation

$$V_{P_\theta}(x') = \sum_x P_\theta(x|x')\left[\gamma V_{P_\theta}(x) + r(x,x')\right]. \tag{106}$$

We thus follow the same semi-gradient approach as previously, using the gradient estimate

$$\nabla_\psi L_V(\psi) \approx -\sum_{x,x'} P_\theta(x|x') P^{\mathrm{ss}}_\theta(x')\left[\gamma V_\psi(x) + r(x,x') - V_\psi(x')\right]\nabla_\psi V_\psi(x'), \tag{107}$$

given by the discounted TD error

$$\delta_{\gamma\mathrm{TD}}(x,x') = \gamma V_\psi(x) + r(x,x') - V_\psi(x'). \tag{108}$$

Algorithm 5. KL regularized discounted AC.

1: **Inputs** dynamical approximation $P_\theta(x, x')$, value approximation $V_\psi(x)$
2: **Parameters** learning rates $\alpha_n^\theta, \alpha_n^\psi$; total updates N, discount factor γ
3: **Initialize** choose initial weights θ and ψ, define iteration variable n, individual error δ
4: $n \leftarrow 0$
5: **Repeat**
6: Generate a transition from x' to $x = \{x, F(x, x')\}$ according to the dynamics given by $P_\theta(x, x')$.
7: $\delta \leftarrow \gamma V_\psi(x) + r(x, x') - V_\psi(x')$
8: $\theta \leftarrow \theta + \alpha_n^\theta \delta \nabla_\theta \ln P_\theta(x|x')$
9: $\psi \leftarrow \psi + \alpha_n^\psi \delta \nabla_\psi V_\psi(x')$
10: $n \leftarrow n + 1$
11: **Until** $n = N$

To approximate the dynamical gradient, we use this TD as an approximation to the one appearing in equation (89), arriving at

$$\nabla_\theta d_{\mathrm{KL}}(P_\theta|P_W) \approx -\sum_{x,x'} P_\theta(x|x')P_\theta^{\mathrm{ss}}(x') \left[\gamma V_\psi(x) + r(x, x') - V_\psi(x')\right] \nabla_\theta \ln P_\theta(x|x'). \tag{109}$$

The resulting online algorithm 5, almost identical to the one for differential returns, is given below.

5.3. Infinite horizon example: random walker on a ring

As a simple example to demonstrate both these algorithms, we return to our particle hopping on a chain example, making the chain periodic with length L, $x \in 0, \ldots, L-1$. The original dynamics we consider is inspired by a model in reference [16, 56]. We take a dynamics given by a periodic potential, specifically

$$P(x+1|x) = \sigma\left(u + v \sin\left(\frac{2\pi x}{L}\right)\right), \tag{110}$$

where $\sigma(y) = \mathrm{e}^y/(1+\mathrm{e}^y)$ is the sigmoid function, and u, v are parameters of the dynamics. Our goal is to study rare trajectories of the particles transition direction, with the sign of the bias s determining whether we focus on trajectories where the direction moved is largely positive or negative. To achieve this we introduce a soft condition by weighting transitions as

$$W(x, x') = \begin{cases} \mathrm{e}^{-s} & (x'-1) \mod L = x \\ \mathrm{e}^{s} & \text{otherwise} \end{cases}. \tag{111}$$

For function approximations, we could choose a tabular approach as we did for the excursions, which would work perfectly well in this simple scenario. To demonstrate a more sophisticated function approximation, making the algorithms learn faster while requiring less data, here we instead choose to use a linear expansion in set of Fourier modes. That is, we set the dynamics to $P_\theta(x+1|x) = \sigma\left(U(x)\right)$ with potential

$$U(x) = \sum_i \theta_i f_i(x), \tag{112}$$

where each f_i is chosen to be either a Fourier mode or the flat function $f_i(x) = 1$, and the values are set to

$$V_\psi(x) = \sum_i \psi_i f_i(x), \tag{113}$$

for the same set of functions f_i. The gradients of these approximations are closely related to the values of this 'feature vector' $\vec{f}$, with

$$\nabla_\psi V_\psi(x) = \vec{f}(x), \tag{114}$$

and

$$\nabla_\theta \ln P_\theta\left(x \pm 1|x\right) = \pm\vec{f}(x)P_\theta\left(x \mp 1|x\right). \tag{115}$$

We train these approximations using both the differential and discounted forms of AC, annealing the bias s across a range of values. By initiating the weights from those found training at nearby values of the bias, we can potentially reduce the number of updates required to achieve good results.

Results are shown in figure 7, with the first row showing: (a) the time-averaged reward $\bar{r}_\theta$; (b) an estimate of the entropy of the dynamics, defined by

$$h = -\sum_{x,x'} P_\theta(x'|x)P_\theta(x) \ln P_\theta(x'|x); \tag{116}$$

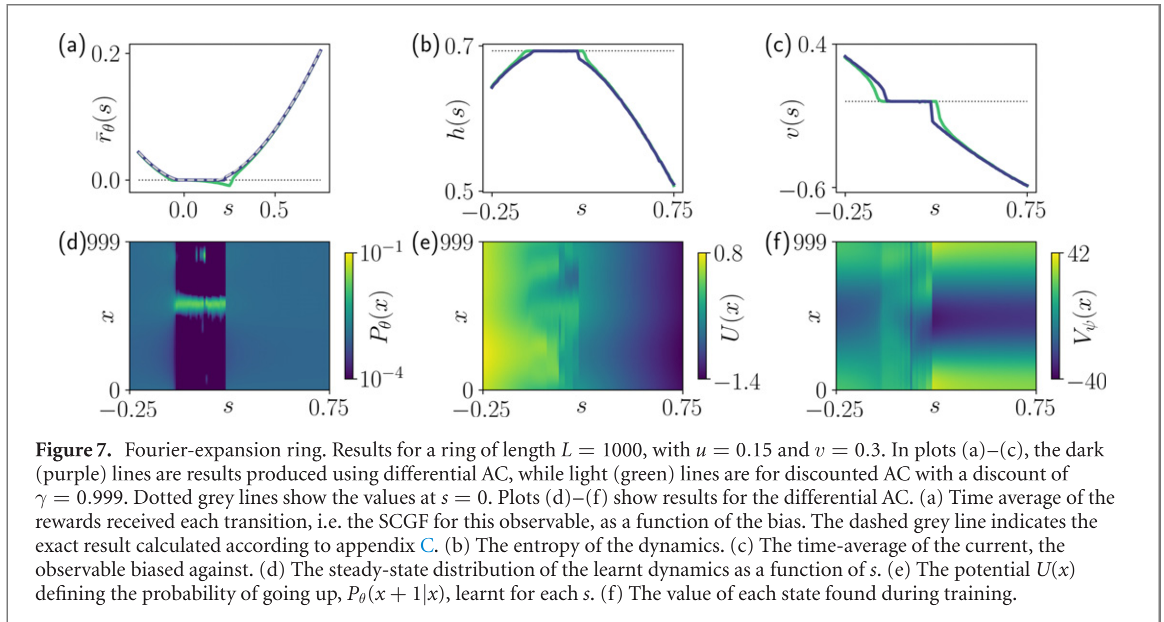

Figure 7. Fourier-expansion ring. Results for a ring of length $L = 1000$, with $u = 0.15$ and $v = 0.3$. In plots (a)–(c), the dark (purple) lines are results produced using differential AC, while light (green) lines are for discounted AC with a discount of $\gamma = 0.999$. Dotted grey lines show the values at $s = 0$. Plots (d)–(f) show results for the differential AC. (a) Time average of the rewards received each transition, i.e. the SCGF for this observable, as a function of the bias. The dashed grey line indicates the exact result calculated according to appendix C. (b) The entropy of the dynamics. (c) The time-average of the current, the observable biased against. (d) The steady-state distribution of the learnt dynamics as a function of s. (e) The potential $U(x)$ defining the probability of going up, $P_\theta(x+1|x)$, learnt for each s. (f) The value of each state found during training.

(c) an estimate of the time-averaged current

$$v = \frac{1}{s}\sum_{x,x'} P_\theta(x'|x)P_\theta(x)\ln\, W(x,x'), \tag{117}$$

with exact results for the time-averaged reward calculated for comparison as described in appendix C. As can be seen from plot figure 7(a), the differential AC provides results with a high degree of accuracy, while the discounting appears to be inaccurate near transitions in the trajectory statistics. This is as expected: near a transition, long-time correlations will become important to the statistics, and discounting puts a cap on how much of the future is taken into account. Figure 7(d) shows the steady state-distribution across the ring, with a region of localization occurring for values of positive bias which are not enough for the optimized dynamics to overcome the constant force of the model. Despite the low entropy of the steady state caused by this localization, this range of biases is in fact where the entropy of the dynamics is highest: here, transitions are likely to occur either up or down, causing the localization. Outside this range the majority of transitions are either up or down, depending on the sign of the bias. The potential defining the probability of going up, the term inside the sigmoid of equation (112), is shown in figure 7(e), with 0 causing equal probability of up or down. Outside the range of biases resulting in localization, we find a clear favour towards going in a direction prescribed by the bias, with the potential either taking significant positive or negative values. Inside the localized range, the potential has an oscillatory structure, which we note will only be accurate where the stationary state is non-negligible.

5.4. Connection to large deviation cumulant generating functions

The construction used in this section is closely related to the theory of LDs, as should be expected given recent connections between the LDs of trajectories and optimal control theory [30, 31]. The optimal dynamics for minimizing the time averaged KL divergence is in fact the dynamics resulting from the generalised Doob transformation [30, 126]. Additionally, the long-time average of the log of the partition function z of equation (80) is exactly the SCGF, the Legendre transform of which provides the probability distribution of the observable whose rare events we are studying. Rearranging equation (82), we have

$$z = \bar{r}_\theta + d_{\mathrm{KL}}(P_\theta|P_W), \tag{118}$$

which holds for any dynamics P_θ. While the KL divergence part of this equation is difficult to calculate, our algorithms are designed to minimize this term, approaching zero at optimality. While optimizing we can easily calculate $\bar{r}_\theta$: indeed, this is already a part of the differential AC algorithm. Thus, these algorithms provide direct access to the SCGF, and therefore the statistics of the rare events.

Minimizing the KL divergence is equivalent to maximizing the return, and since the KL divergence is non-negative we may rewrite

$$\begin{aligned} z &\geqslant \bar{r}_\theta \quad \forall\ \theta \\ &\geqslant \max_\theta \bar{r}_\theta \\ &\geqslant \max_\theta \sum_{x',x} P^{\mathrm{ss}}_\theta(x) P_\theta(x'|x) \left[\ln\ W(x',x) - \ln\left(\frac{P_\theta(x'|x)}{P(x'|x)}\right)\right], \end{aligned} \tag{119}$$

with the inequality saturable if the Doob dynamics is contained within the variational space spanned by θ for the chosen function approximation, that is

$$z = \max_{\tilde{P}} \sum_{x',x} \tilde{P}^{\mathrm{ss}}(x) \tilde{P}(x'|x) \left[\ln\ W(x',x) - \ln\left(\frac{\tilde{P}(x'|x)}{P(x'|x)}\right)\right], \tag{120}$$

as seen in the LD literature discussing connections to optimal control [30, 31]. The time-averaged reward estimated during training thus provides an efficient way of calculating at least a lower bound of the SCGF, with powerful function approximations and extensive training allowing access to an accurate value without needing to use any other form of statistical sampling. In cases where high degrees of accuracy are not possible, the learnt dynamics can be combined with sampling techniques such as TPS or cloning to calculate a better estimate.

6. Conclusions and outlook

In this work we have highlighted a general approach for developing numerical approaches to study questions about statistical ensembles of trajectories, with a particular focus on ensembles consisting of rare trajectories of some original dynamics. We have shown that gradient based optimization of a sampling dynamics for these trajectory ensembles naturally maps onto a regularized form of RL, closely related to maximum-entropy RL. We used this connection to pedagogically develop algorithms in a finite time setting, a key ingredient being the extensive use of value functions, a first in the rare trajectory sampling literature. Reviewing a range of modifications to learning algorithms and choices of function approximations found in the RL literature, we saw just how many possibilities this connection makes available for the study of rare trajectories. We then adapted the approach for time-homogeneous problems which have no unique time and can be viewed as single unending trajectories, for the study of statistics of time-averaged observables, and described how this connects to the theory of LDs for Markov processes and its relationship with optimal control theory. This development was supplemented by two examples: generating random walker excursions with the correct probabilities for the finite time case, and statistics of the time-averaged current for a particle on a ring in the infinite time case.

There is a wide range of possible avenues for future research building on what we have presented here. An obvious one is using these algorithms to tackle more sophisticated problems than the simple models we used as illustration. For example, we may seek to apply the approach to study rare trajectories of many-body systems such as spin lattices or molecular dynamics, where the state space grows exponentially with the number of particles. In this situation, the algorithms are essentially unchanged: the difficulty comes in making an appropriate choice of function approximation, such that it can efficiently encode the dynamics. Analytical study of many problems can produce simple, physically inspired parameterizations of the dynamics in such many body systems, see e.g. [36, 51, 127]. Where these physically inspired approximations cease to be sufficient, or where it is difficult to gain such insight, we could instead resort to neural networks. These have proven to be an incredibly versatile function approximations, with extreme representative power. Their application to RL comes with a caveat, however: they are unstable with the simpler algorithms we have presented. As discussed in section 4.3, to overcome these issues, training of neural networks must therefore be conducted using modified algorithms. Further examples of the use of neural networks in LDs can be found in [37, 128].

Many-body problems will bring with them a separate issue to overcome: how to achieve sufficiently broad sampling of the state space, especially in models near phase transitions, where Markov chain sampling can become trapped in subsets of the state space. The trapping could lead to over fitting of the function approximation on the current area of the state space the Markov chain is sampling, forgetting the dynamics in previously visited regions. This is a problem which may be addressed by running multiple trajectories in parallel, or through the use of replay buffers to further sample previously visited regions of the state space.

Beyond applications, interesting generalizations and extensions include:

- **Limited control.** In certain situations it may be beneficial (or only possible) to make part of the dynamics adaptive. For example, in a many-body system where each particle has separate degrees of freedom such as a position and orientation, we may only control the orientational evolution while leaving the position unchanged from the original dynamics. In this setup, the evolution of the position takes on the role of an environment from the RL perspective, with the orientation under the control of the agent. While this may limit the effectiveness of the resulting dynamics for sampling, it could be much easier to optimize, requiring less parameters or having a more obvious choice of function approximation.
- **Non-Markovian original dynamics.** As discussed earlier, the approach developed in this work can be almost immediately extended to arbitrary non-Markovian original dynamics in the finite time case. For example, the MCR with a value baseline becomes based on the gradients

$$\nabla_\theta D_{\mathrm{KL}}(P_\theta|P_W) = -\left\langle \sum_{t=1}^{T} \left(R_W(\omega_0^T) - V_\psi(\omega_0^t)\right) \nabla_\theta \ln P_\theta(x_t|\omega_0^{t-1}) \right\rangle_{P_\theta}, \tag{121}$$

$$\nabla_\psi L_V(\psi) = -\left\langle \sum_{t=1}^{T} \left(R_W(\omega_0^T) - V_\psi(\omega_0^t)\right) \nabla_\psi V_\psi(\omega_0^t) \right\rangle_{P_\theta}, \tag{122}$$

  where we have simply replaced the state and time with the full history of the trajectory, sampling with a parameterized dynamics which is itself non-Markovian. While general, this is more likely to be applicable with approximation in studying the statistics of problems where the original dynamics has a limited amount of memory. An alternative use case is a side effect of using function approximations: since some useful information may be lost in processing the state, the dynamics is effectively non-Markovian. Making use of processed states, i.e. feature vectors, of a recent history of states may thus improve the accuracy of the dynamics further. A similar modification can be made for the infinite time case when the original dynamics has a limited range of non-Markovianity, or the weighting depends on a short part of the history of previous states. A particularly powerful function approximation to apply in such problems is that of recurrent neural networks.
- **Non-Markovian weights.** Rather than the original dynamics being non-Markovian, its possible that the weights may be non-Markovian. That is, rather than taking the transition-local product structure of (6), the weight of each trajectory may simply be some function $W\left(\omega_0^T\right)$. Generically this will result in a problem identical in structure to the one above in (121): even if the original dynamics remains Markovian, the non-Markovian nature of the weights will necessitate a non-Markovian parameterized dynamics to best sample the reweighted ensemble. However, many non-Markovian weights may only require a subset of the information contained in the trajectories history.

  For example, suppose we wish to consider the subset of random walks with a particular area A. To do this we would set the weights to

$$W\left(\omega_0^T\right) = \delta_{A,A\left(\omega_0^T\right)}. \tag{123}$$

  There is no obvious way to split this weight up, but we can observe that the only information about the history necessary to calculate the weight at the end is its area $A\left(\omega_0^T\right)$. As such, as a trajectory evolves the only information we need keep track of is the area up to each time A_t, updating it after each transition. It seems reasonable that the optimal dynamics to sample this ensemble may only be conditional on only the current state, time, and the area up to that point in the trajectory: that is, it should be sufficient to parameterize a conditional dynamics $P_\theta(x_{t+1}|x_t, t, A_t)$. This can in fact be proven, and presents a particular case of what we call a **generalized state**: the necessary information, in this case (x_t, t, A_t) from the trajectories history to be able to exactly reproduce the reweighted ensemble. In future work, we will further expand on the idea of generalized states, applying our approach to more complex conditional problems.
- **Fluctuating time ensembles.** Rather than ending trajectories at a fixed time, we could end trajectories according to some condition, for example, to study the statistics of rare first passages. Given that variable length trajectories are the natural setting of RL, these algorithms will have natural adaptations to sampling in these problems, with optimal sampling dynamics likely being time-independent.
- **Continuous time Markov processes.** Here for concreteness we presented our approach for discrete-time dynamics, but it can easily be generalised to both continuous-time jump processes, to diffusions, and to combinations of both. Indeed, there is already an extensive literature of work covering continuous time versions of RL [129–133]. In fact, the continuous time version of our

loss-functions have already been discussed in [30], where connections were made between LD theory and control theory. Further to this, there has already been some adaptive algorithms of a similar nature developed for sampling rare trajectories in the continuous time case. In particular, [56] uses an algorithm which is an approximation to an '∞-step' version of the differential AC algorithm described above. This allows the removal of the value function, since for the current state it is a baseline, and for the potential '∞-step' states the value averages to zero over the stationary state. Approximations result from truncating the partial return between these two times to a finite length. Additionally, in [55] the KL divergence is used with the parameterized and weighted distributions swapped around. Finally, a version of the LSTD algorithm [39, 134] applied to the non-linear Bellman equation (77) [119] has recently been developed for LDs of diffusive systems [16]. Despite the above developments, value functions and the many other techniques found in RL are not currently used for the sampling of rare, continuous time trajectories.

- **Use in TPS or cloning.** If the function approximation is incapable of achieving a sufficient accuracy to study the rare events (e.g. to directly estimate the SCGF using optimized trajectories) then TPS or cloning could be used to fix the statistics, with convergence sped up by the optimized dynamics [37, 56, 127].

Beyond these applications of RL-like techniques to statistical sampling, there is the obvious potential of taking this connection in the other direction, to gain further understanding of RL itself through the use of techniques and intuitions from the statistical physics perspective.

Acknowledgments

The authors thank A Lamacraft for bringing to our attention literature central to the development of this project. We also thank the referees for comments leading to a significant improvement in presentation. This research was funded in part EPSRC Grant No. EP/M014266/1, by University Nottingham Grant No. FiF1/3, and The Leverhulme Trust Grant No. RPG-2018-181. We are grateful for access to the University of Nottingham Augusta HPC service. We also acknowledge the use of Athena at HPC Midlands+, which was funded by the EPSRC on Grant EP/P020232/1 as part of the HPC Midlands+ consortium.

Appendix A. Exact optimal sampling and random walk excursions

In this appendix we demonstrate how the optimal dynamics can be calculated exactly, either analytically or numerically. This is done by propagating an iterative equation for a function of the state and time, which is used to rescale the original transition probabilities. While in principle this can solve any problem, it can be numerically unstable, and will not be applicable as presented to problems which are the target application of the current line of research: systems for which the state space is too large for a single value to be associated to every state. It is expected that these techniques can also be extended to generic function approximation (see reference [16] for linear approximations in diffusion processes), however, it is likely less stable than algorithms based on the KL divergence, due to multiplicative (rather than additive) nature of the objects involved frequently causing extremely large or small numerical values.

Beginning from

$$P_W\left(\omega_0^T\right) = \frac{\prod_{t=0}^T W(x_t, x_{t-1}, t) \prod_{t=1}^T P(x_t|x_{t-1}) P(x_0)}{\sum_{\omega_0^T} W\left(\omega_0^T\right) P\left(\omega_0^T\right)}, \tag{A.1}$$

we aim for a time dependent Markovian dynamics generating this ensemble. However, rather than assuming this is possible, we first calculate a decomposition into non-Markovian conditional probabilities, producing

$$P_W\left(\omega_0^T\right) = \prod_{t=0}^T P_W(x_t|\omega_0^{t-1}). \tag{A.2}$$

To do this, we use the definition of a conditional probability in terms of joint probability distributions: iterating backwards step by step we have

$$P_W\left(\omega_0^{t-1}\right) = \sum_{x_t} P_W\left(\omega_0^t\right), \tag{A.3}$$

and thus

$$P_W\left(x_t|\omega_0^{t-1}\right) = \frac{P_W\left(\omega_0^t\right)}{P_W\left(\omega_0^{t-1}\right)}. \tag{A.4}$$

Combining these definitions, for the final timestep we have

$$P_W\left(x_T|\omega_0^{T-1}\right) = \frac{\prod_{t=0}^{T} W(x_t, x_{t-1}, t)P\left(\omega_0^T\right)}{\sum_{x_T}\prod_{t=0}^{T} W(x_t, x_{t-1}, t)P\left(\omega_0^T\right)} = \frac{W(x_T, x_{T-1}, T)P\left(x_T|x_{T-1}\right)}{\sum_{x_T} W(x_T, x_{T-1}, T)P\left(x_T|x_{T-1}\right)} = \frac{W(x_T, x_{T-1}, T)P\left(x_T|x_{T-1}\right)}{\mathbb{E}_{x_T\sim P}\left[W(x_T, x_{T-1}, T)|x_{T-1}\right]}, \tag{A.5}$$

where we see that despite starting from joint probabilities over the whole history of the trajectory, the end result is invariant over all but the state prior to the transition, and thus we may write $P_W\left(x_T|\omega_0^{T-1}\right) = P_W\left(x_T|x_{T-1}, T\right)$ for all past trajectories up to the final transition. For earlier times we have

$$P_W\left(x_t|\omega_0^{t-1}\right) = \frac{\sum_{\omega_{t+1}^T}\prod_{t'=0}^{T} W(x_{t'}, x_{t'-1}, t')P\left(\omega_0^T\right)}{\sum_{\omega_t^T}\prod_{t'=0}^{T} W(x_{t'}, x_{t'-1}, t')P\left(\omega_0^T\right)} = \frac{\left[\sum_{\omega_{t+1}^T}\prod_{t'=t+1}^{T} W(x_{t'}, x_{t'-1}, t')P\left(\omega_{t+1}^T|x_t\right)\right] W(x_t, x_{t-1}, t)P\left(x_t|x_{t-1}\right)}{\sum_{\omega_t^T}\prod_{t'=t}^{T} W(x_{t'}, x_{t'-1}, t')P\left(\omega_t^T|x_{t-1}\right)} = \frac{\mathbb{E}_{\omega_{t+1}^T\sim P}\left[\prod_{t'=t+1}^{T} W(x_{t'}, x_{t'-1}, t')|x_t\right] W(x_t, x_{t-1}, t)P\left(x_t|x_{t-1}\right)}{\mathbb{E}_{\omega_t^T\sim P}\left[\prod_{t'=t}^{T} W(x_{t'}, x_{t'-1}, t')|x_{t-1}\right]}, \tag{A.6}$$

where similarly to the final transition, the dependence on the past prior to the state before the transitions at each time have cancelled out, allowing us to write $P_W\left(x_t|\omega_0^{t-1}\right) = P_W\left(x_t|x_{t-1}, t\right)$ for all times. Finally, the initial distribution is modified as

$$P_W(x_0) = \frac{\mathbb{E}_{\omega_1^T\sim P}\left[\prod_{t'=1}^{T} W(x_{t'}, x_{t'-1}, t')|x_t\right] W(x_0, 0)P\left(x_0\right)}{\mathbb{E}_{\omega_0^T\sim P}\left[\prod_{t'=0}^{T} W(x_{t'}, x_{t'-1}, t')\right]}. \tag{A.7}$$

These expectations represent the expected contribution to the weighting of the trajectories future given the current state and time. The individual contributions to the expectation play a similar role to the returns in our algorithms, however, now they have a product structure over the individual factors associated to each transition, rather than a sum structure. Labelling these expectations as

$$g(x_t, t) = \mathbb{E}_{\omega_{t+1}^T\sim P}\left[\prod_{t'=t+1}^{T} W(x_{t'}, x_{t'-1}, t')\middle| x_t\right], \tag{A.8}$$

with $g(x, T) = 1$ for all x, we have

$$P_W\left(x_t|x_{t-1}, t-1\right) = \frac{g(x_t, t)}{g(x_{t-1}, t-1)} W(x_t, x_{t-1}, t)P\left(x_t|x_{t-1}\right), \tag{A.9}$$

for all t. The function g, related to a gauge transformation of the trajectory probabilities, can then be efficiently calculated by iterating backwards, using

$$g(x_t, t) = \mathbb{E}_{x_{t+1}\sim P}\left[W(x_{t+1}, x_t, t+1)g(x_{t+1}, t+1)\right]. \tag{A.10}$$

Excursions. We now demonstrate the above approach by calculating the transformation for the conditioned random walk excursions case mentioned in section 2. This problem possesses a lightcone structure inherited from the original random walker dynamics: since each transition can only go up or down one, the position n steps in the future or past can only be n higher or lower than the present position. Since we are targeting a dynamics which will entirely end in a single state, this lightcone structure means the backwards iteration based on equation (A.10) will simplify significantly, allowing analytical solution.

With the weights defined by $W(x', x, T) = \delta(x')$ and $W(x', x, t) = H(x')$ we have

$$g(x, t) = \frac{1}{2}\left(H(x+1)g(x+1, t+1) + H(x-1)g(x-1, t+1)\right), \tag{A.11}$$

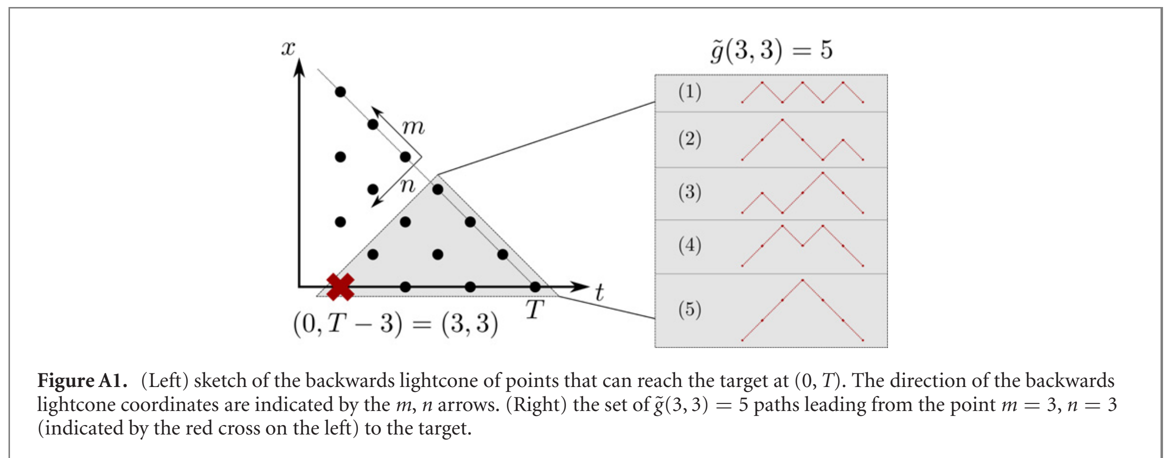

Figure A1. (Left) sketch of the backwards lightcone of points that can reach the target at $(0, T)$. The direction of the backwards lightcone coordinates are indicated by the m, n arrows. (Right) the set of $\tilde{g}(3, 3) = 5$ paths leading from the point $m = 3, n = 3$ (indicated by the red cross on the left) to the target.

for $t < T - 1$, with end condition $g(x, T) = 1$ for all x and

$$g(x, T-1) = \frac{1}{2}\left(\delta(x+1) + \delta(x-1)\right). \tag{A.12}$$

This immediately implies that $g(x, t) = 0$ if $x < -1$ from the heaviside step function, and $g(-1, t) = 0.5g(0, t)$ on the positive–negative boundary. The lightcone structure, imposed by the delta function at the final time, results in $g(x, t) = 0$ for $x > T - t$.

For the remaining components of the gauge transformation, those with $0 \leqslant x \leqslant T - t$ which correspond to the probability of the remainder of the trajectory being an excursion under the original dynamics, we apply two transformations. First, we set $g'(x, t) = 2^{-t}g(x, t)$, modifying the equations to

$$g'(x, t) = H(x+1)g'(x+1, t+1) + H(x-1)g'(x-1, t+1), \tag{A.13}$$

for $t < T - 1$, with end condition $g(x, T) = 1$ for all x and

$$g'(x, T-1) = \delta(x+1) + \delta(x-1). \tag{A.14}$$

Here g' is interpreted as measuring the number of paths leading from the current position to the target without going below zero. Next, we perform a coordinate transformation to backward-lightcone adapted coordinates (m, n), where m/n correspond to steps up/down going back in time (see figure A1), defined by $x = m - n$ and $t = T - m - n$. The gauge transformation in this coordinate system $\tilde{g}$ is then defined as $\tilde{g}(m, n) = g'(m - n, T - m - n)$: $\tilde{g}$ is intepreted as the number of ordered combinations of ups and downs going backwards in time for which, given any subsequence starting from the end, there are always less or equal downs than ups, i.e. $x \geqslant 0$. In these coordinates, the function $\tilde{g}$ satisfies the following set of equations

(a) $\tilde{g}(m, 0) = 1$ for $m \geqslant 0$,
(b) $\tilde{g}(m, 1) = n$ for $m \geqslant 1$,
(c) $\tilde{g}(m+1, n) = \tilde{g}(m+1, n-1) + \tilde{g}(m, n)$ for $1 < n < m + 1$,
(d) $\tilde{g}(m+1, m+1) = \tilde{g}(m+1, m)$ for $m \geqslant 1$,

which are precisely the equations defining Catalan's triangle, solved by

$$\tilde{g}(m, n) = \frac{(m+n)!(m-n+1)}{n!(m+1)!}, \tag{A.15}$$

as demonstrated in the right of figure A1. Reversing the transformations we find

$$g(x, t) = 2^t \tilde{g}\left(\frac{T+x-t}{2}, \frac{T-x-t}{2}\right), \tag{A.16}$$

and thus

$$g(x, t) = \frac{1}{2^t} \frac{(T-t)!\,(x+1)}{\left(\frac{T-x-t}{2}\right)!\left(\frac{T+x-t+2}{2}\right)!}. \tag{A.17}$$

Finally, given this transformation, we can now calculate the transition probabilities for the optimal sampling of random walk excursions, finding

$$P_W\left(x \pm 1 | x, t-1\right) = \frac{1}{2} 2^t \frac{(T-t)!\,(x \pm 1 + 1)}{\left(\frac{T-x\mp 1-t}{2}\right)!\left(\frac{T+x\pm 1-t+2}{2}\right)!} \frac{1}{2^{t-1}} \frac{\left(\frac{T-x-t+1}{2}\right)!\left(\frac{T+x-t+3}{2}\right)!}{(T-t+1)!\,(x+1)}$$

$$= \frac{1}{2}\left(1 \pm \frac{1}{x+1}\right)\left(1 \mp \frac{x+1 \mp 1}{T-t+1}\right). \tag{A.18}$$

Appendix B. Maximum return estimation

When training the dynamics for optimal rare trajectory sampling, the most efficient way to evaluate the current dynamics is by estimating the average return it produces. If this average increases over time, then the model is being successfully trained. To this end, in situations where it is available, it is useful to have an estimate for the maximum possible return over all possible transition matrices for precise evaluation of how good the model is.

This upper bound on the return can be estimates numerically by using the gauge transformations discussed in the appendix A. First, note that since the KL divergence must be greater than 0, equation (24) immediately implies an upper bound of

$$\sum_{\omega_0^T} P_\theta(\omega_0^T) R(\omega_0^T) \leqslant \ln Z, \tag{B.1}$$

which is saturated by setting $P_\theta(x'|x,t)$ to the gauge transformed dynamics in appendix A. We may then rewrite

$$Z = \sum_{\omega_0^T} W\left(\omega_0^T\right) P\left(\omega_0^T\right) = \sum_x g(x,0)p(x), \tag{B.2}$$

where $p(x)$ is the original initial state distribution. The upper bound may then be rewritten in terms of the gauge transformation

$$\sum_{\omega_0^T} P_\theta(\omega_0^T) R(\omega_0^T) \leqslant \ln\left(\sum_x g(x,0)p(x)\right). \tag{B.3}$$

For the excursion example, this takes a particularly simple form: since the initial state distribution is $p(x)\delta_{x0}$, only a single gauge component contributes

$$\sum_{\omega_0^T} P_\theta(\omega_0^T) R(\omega_0^T) \leqslant \ln g(0,0). \tag{B.4}$$

As such, for the upper bounds in section 3.4 we simply need to estimate this component of the gauge transformation by numerical back-propagation of the gauge.

Appendix C. Exact diagonalization for SCGF and optimal dynamics

In order to have an accurate result for evaluation of the infinite time algorithms, we use a common technique from LD theory, turning the issue of finding the SCGF and optimal (Doob) sampling dynamics into one of exact diagonalization. To this end, we first define the tilted master operator P_s with components

$$P_s(x'|x) = P(x'|x)W_s(x,x'), \tag{C.1}$$

with the weighting parametrized by the bias s. It follows simply from the definitions that the SCGF $\theta(s)$

$$\begin{aligned}\theta(s) &= \lim_{T\to\infty} \ln\left[\sum_{\omega_0^T} P(\omega_0^T) W_s(\omega_0^T)\right] \\ &= \lim_{T\to\infty} \ln\left[\langle -|P_s^T|P_{\mathrm{ss}}\rangle\right],\end{aligned} \tag{C.2}$$

where $|P_{\mathrm{ss}}\rangle$ is the steady state distribution, and thus in the infinite time limit the SCGF is simply the log of the leading eigenvalue of the matrix P_s.

Further to this, it is possible to calculate the optimal sampling dynamics by using this leading eigenvalue and its corresponding left eigenvector, which we label l_s with components $l_s(x)$. First, we scale the operator so that its eigenvalues are at or below zero, $P_s/\mathrm{e}^{\theta(s)}$. Next, we need the action of the flat state on the left of this matrix to result in zero for probability conservation: we therefore perform a basis transformation using a matrix with diagonal elements given by the components of l_s, finding the optimal dynamics

$$\tilde{P} = \frac{\mathrm{diag}(l_s)P_s\ \mathrm{diag}\,(l_s)^{-1}}{\mathrm{e}^{\theta(s)}}, \tag{C.3}$$

with the new stationary state given by component wise multiplication of the left and right eigenvectors

$$P_{ss}^{s}(x) = l_s(x) r_s(x). \tag{C.4}$$

That this is optimal can be derived more precisely from an infinite time version of the gauge-transformation related approach of appendices A and B.

ORCID iDs

Dominic C Rose https://orcid.org/0000-0001-5390-8635

References

[1] Bolhuis P G, Chandler D, Dellago C and Geissler P L 2002 TRANSITIONPATHSAMPLING: throwing ropes over rough mountain passes, in the dark *Annu. Rev. Phys. Chem.* **53** 291
[2] Garrahan J P 2018 Aspects of non-equilibrium in classical and quantum systems: slow relaxation and glasses, dynamical large deviations, quantum non-ergodicity, and open quantum dynamics *Physica* A **504** 130–54
[3] Touchette H 2009 The large deviation approach to statistical mechanics *Phys. Rep.* **478** 1–69
[4] Giardinà C, Kurchan J and Peliti L 2006 Direct evaluation of large-deviation functions *Phys. Rev. Lett.* **96** 120603
[5] Cérou F and Guyader A 2007 Adaptive multilevel splitting for rare event analysis *Stoch. Anal. Appl.* **25** 417–43
[6] Lecomte V and Tailleur J 2007 A numerical approach to large deviations in continuous time *J. Stat. Mech.* P03004
[7] Gorissen M, Hooyberghs J and Vanderzande C 2009 Density-matrix renormalization-group study of current and activity fluctuations near nonequilibrium phase transitions *Phys. Rev.* E **79** 020101
[8] Giardina C, Kurchan J, Lecomte V and Tailleur J 2011 Simulating rare events in dynamical processes *J. Stat. Phys.* **145** 787
[9] Nemoto T and Sasa S-i 2014 Computation of large deviation statistics via iterative measurement-and-feedback procedure *Phys. Rev. Lett.* **112** 090602
[10] Nemoto T, Bouchet F, Jack R L and Lecomte V 2016 Population-dynamics method with a multicanonical feedback control *Phys. Rev.* E **93** 062123
[11] Nemoto T, Jack R L and Lecomte V 2017 Finite-size scaling of a first-order dynamical phase transition: adaptive population dynamics and an effective model *Phys. Rev. Lett.* **118** 115702
[12] Nemoto T, Fodor É, Cates M E, Jack R L and Tailleur J 2018 Optimizing active work: dynamical phase transitions, collective motion, and jamming *Phys. Rev.* E **99** 022605
[13] Ray U, Chan G K-L and Limmer D T 2018 Exact fluctuations of nonequilibrium steady states from approximate auxiliary dynamics *Phys. Rev. Lett.* **120** 210602
[14] Ray U, Chan G K-L and Limmer D T 2018 Importance sampling large deviations in nonequilibrium steady states. I *J. Chem. Phys.* **148** 124120
[15] Klymko K, Geissler P L, Garrahan J P and Whitelam S 2018 Rare behavior of growth processes via umbrella sampling of trajectories *Phys. Rev.* E **97** 032123
[16] Ferré G and Touchette H 2018 Adaptive sampling of large deviations *J. Stat. Phys.* **172** 1525–44
[17] Bañuls M C and Garrahan J P 2019 Using matrix product states to study the dynamical large deviations of kinetically constrained models *Phys. Rev. Lett.* **123** 200601
[18] Helms P, Ray U and Chan G K-L 2019 Dynamical phase behavior of the single- and multi-lane asymmetric simple exclusion process via matrix product states *Phys. Rev.* E **100** 022101
[19] Jacobson D and Whitelam S 2019 Direct evaluation of dynamical large-deviation rate functions using a variational ansatz *Phys. Rev.* E **100** 052139
[20] Ray U and Chan G K-L 2020 Constructing auxiliary dynamics for nonequilibrium stationary states by variance minimization *J. Chem. Phys.* **152** 104107
[21] Helms P and Chan G K-L 2020 Dynamical phase transitions in a 2D classical nonequilibrium model via 2D tensor networks *Phys. Rev.* E **125** 140601
[22] Dean T and Dupuis P 2009 Splitting for rare event simulation: a large deviation approach to design and analysis *Stoch. Process. Appl.* **119** 562–87
[23] Carollo F and Pérez-Espigares C 2020 Entanglement statistics in Markovian open quantum systems: a matter of mutation and selection *Phys. Rev.* E **102** 030104(R)
[24] Hedges L O, Jack R L, Garrahan J P and Chandler D 2009 Dynamic order–disorder in atomistic models of structural glass formers *Science* **323** 1309
[25] Borkar V S 2002 Q-learning for risk-sensitive control *Math. Oper. Res.* **27** 294–311
[26] Borkar V S, Juneja S and Kherani A A 2003 Peformance analysis conditioned on rare events: an adaptive simulation scheme *Commun. Inf. Syst.* **3** 256–78
[27] Ahamed T P I, Borkar V S and Juneja S 2006 Adaptive importance sampling technique for Markov chains using stochastic approximation *Oper. Res.* **54** 489–504
[28] Basu A, Bhattacharyya T and Borkar V S 2008 A learning algorithm for risk-sensitive cost *Math. Oper. Res.* **33** 880–98
[29] Todorov E 2009 Efficient computation of optimal actions *Proc. Natl Acad. Sci.* **106** 11478–83
[30] Chetrite R and Touchette H 2015 Variational and optimal control representations of conditioned and driven processes *J. Stat. Mech.* P12001
[31] Jack R L and Sollich P 2015 Effective interactions and large deviations in stochastic processes *Eur. Phys. J. Spec. Top.* **224** 2351–67
[32] Garrahan J P 2016 Classical stochastic dynamics and continuous matrix product states: gauge transformations, conditioned and driven processes, and equivalence of trajectory ensembles *J. Stat. Mech.* 073208
[33] Jack R L 2020 Ergodicity and large deviations in physical systems with stochastic dynamics *Eur. Phys. J.* B **93** 74
[34] Derrida B and Sadhu T 2019 Large deviations conditioned on large deviations: I. Markov chain and Langevin equation *J. Stat. Phys.* **176** 773–805

[35] Derrida B and Sadhu T 2019 Large deviations conditioned on large deviations: II. Fluctuating hydrodynamics *J. Stat. Phys.* **177** 151–82

[36] Dolezal J and Jack R L 2019 Large deviations and optimal control forces for hard particles in one dimension *J. Stat. Mech.* 123208

[37] Oakes T H E, Moss A and Garrahan J P 2020 A deep learning functional estimator of optimal dynamics for sampling large deviations *Mach. Learn.: Sci. Technol.* **1** 035004

[38] Gillman E, Rose D C and Garrahan J P 2020 A tensor network approach to finite markov decision processes (arXiv:2002.05185)

[39] Sutton R S and Barto A G 2018 *Reinforcement Learning: An Introduction* 2nd edn) (Cambridge, MA: MIT Press)

[40] Williams R J 1987 Reinforcement-learning connectionist systems *Technical Report* Northeastern University

[41] Williams R J 1992 Simple statistical gradient-following algorithms for connectionist reinforcement learning *Mach. Learn.* **8** 229–56

[42] Bukov M, Day A G R, Sels D, Weinberg P, Polkovnikov A and Mehta P 2018 Reinforcement learning in different phases of quantum control *Phys. Rev.* X **8** 031086

[43] Bukov M 2018 Reinforcement learning for autonomous preparation of floquet-engineered states: inverting the quantum kapitza oscillator *Phys. Rev.* B **98** 224305

[44] Fösel T, Tighineanu P, Weiss T and Marquardt F 2018 Reinforcement learning with neural networks for quantum feedback *Phys. Rev.* X **8** 031084

[45] Chen F, Chen J-J, Wu L-N, Liu Y-C and You L 2019 Extreme spin squeezing from deep reinforcement learning *Phys. Rev.* A **100** 041801(R)

[46] Yao J, Bukov M and Lin L 2020 Policy gradient based quantum approximate optimization algorithm *Proc. 1st Mathematical and Scientific Machine Learning*

[47] Bolens A and Heyl M 2020 Reinforcement learning for digital quantum simulation (arXiv:2006.16269)

[48] Albarrán-Arriagada F, Retamal J C, Solano E and Lamata L 2020 Reinforcement learning for semi-autonomous approximate quantum eigensolver *Mach. Learn.: Sci. Technol.* **1** 015002

[49] Barr A, Gispen W and Lamacraft A 2020 Quantum ground states from reinforcement learning *Proc. 1st Mathematical and Scientific Machine Learning*

[50] Bojesen T A 2018 Policy-guided Monte Carlo: reinforcement-learning Markov chain dynamics *Phys. Rev.* E **98** 063303

[51] Whitelam S, Jacobson D and Tamblyn I 2020 Evolutionary reinforcement learning of dynamical large deviations *J. Chem. Phys.* **153** 044113

[52] Beeler C, Yahorau U, Coles R, Mills K, Whitelam S and Tamblyn I 2019 Optimizing thermodynamic trajectories using evolutionary reinforcement learning (arXiv:1903.08543)

[53] Todorov E 2007 Linearly-solvable Markov decision problems *Proc. 21st Int. Conf. on Neural Information Processing Systems* pp 1369–76

[54] Kappen H J, Gómez V and Opper M 2012 Optimal control as a graphical model inference problem *Mach. Learn.* **87** 159–82

[55] Kappen H J and Ruiz H C 2016 Adaptive importance sampling for control and inference *J. Stat. Phys.* **162** 1244–66

[56] Das A and Limmer D T 2019 Variational control forces for enhanced sampling of nonequilibrium molecular dynamics simulations *J. Chem. Phys.* **151** 244123

[57] Neu G, Jonsson A and Gómez V 2017 A unified view of entropy-regularized markov decision processes (arXiv:1705.07798)

[58] Geist M, Scherrer B and Pietquin O 2019 A theory of regularized Markov decision processes *Proc. 36th Int. Conf. on Machine Learning*

[59] Haarnoja T, Tang H, Abbeel P and Levine S 2017 Reinforcement learning with deep energy-based policies *Proc. 34th Int. Conf. on Machine Learning*

[60] Haarnoja T, Zhou A, Abbeel P and Levine S 2018 Soft actor-critic: off-policy maximum entropy deep reinforcement learning with a stochastic actor *Proc. 35th Int. Conf. on Machine Learning*

[61] Levine S 2018 Reinforcement learning and control as probabilistic inference: tutorial and review (arXiv:1805.00909)

[62] Majumdar S N and Orland H 2015 Effective Langevin equations for constrained stochastic processes *J. Stat. Mech.* P06039

[63] Mair J F and Rose D C 2020 Reinforcement learning for efficient discrete time trajectory sampling, Github https://github.com/JamieMair/rledts

[64] Roynette B and Yor M 2009 *Penalising Brownian Paths* (Berlin: Springer)

[65] Kushner H J and Yin G G 2003 *Stochastic Approximation and Recursive Algorithms and Applications* (Berlin: Springer)

[66] Borkar V S 2008 *Stochastic Approximation: A Dynamical Systems Viewpoint* (India: Hindustan Book Agency)

[67] Bertsekas D P and Tsitsiklis J N 1996 *Neuro-Dynamic Programming* (Berlin: Springer)

[68] Nachum O, Norouzi M, Xu K and Schuurmans D 2017 Bridging the gap between value and policy based reinforcement learning *Proc. 31st Int. Conf. on Neural Information Processing Systems, NIPS'17* (Red Hook, NY: Curran Associates Inc.) pp 2772–82

[69] Nachum O, Norouzi M, Xu K and Schuurmans D 2017 Trust-pcl: an off-policy trust region method for continuous control (arXiv:1707.01891)

[70] Greensmith E, Bartlett P L and Baxter J 2004 Variance reduction techniques for gradient estimates in reinforcement learning *J. Mach. Learn. Res.* **5** 1471–1530

[71] Dick T B 2015 Policy gradient reinforcement learning without regret *Master's Thesis* University of Alberta

[72] Bhatnagar S, Sutton R S, Ghavamzadeh M and Lee M 2009 Natural actor-critic algorithms *Automatica* **45** 2471–82

[73] Sutton R S 1988 Learning to predict by the methods of temporal differences *Mach. Learn.* **3** 9–44

[74] Sutton R S, Maei H R, Precup D, Bhatnagar S, Silver D, Szepesvári C and Wiewiora E 2009 Fast gradient-descent methods for temporal-difference learning with linear function approximation *Proc. 26th Int. Conf. on Machine Learning, ICML'09* (New York: ACM) pp 993–1000

[75] Maei H R, Szepesvári C, Bhatnagar S, Precup D, Silver D and Sutton R S 2009 Convergent temporal-difference learning with arbitrary smooth function approximation *Proc. 23rd Int. Conf. on Neural Information Processing Systems, NIPS'09* (Red Hook, NY: Curran Associates Inc.) pp 1204–12

[76] Maei H R 2011 Gradient temporal-difference learning algorithms *PhD Thesis* University of Alberta

[77] Maei H R 2018 Convergent actor-critic algorithms under off-policy training and function approximation (arXiv:1802.07842)

[78] van Seijen H, van Hasselt H, Whiteson S and Wiering M 2009 A theoretical and empirical analysis of expected sarsa *Proc. 2009 IEEE Symp. on Adaptive Dynamic Programming and Reinforcement Learning* pp 177–84

[79] Allen C, Asadi K, Roderick M, Mohamed A-r, Konidaris G and Littman M 2017 Mean actor critic (arXiv:1709.00503)

[80] Ciosek K and Whiteson S 2020 Expected policy gradients for reinforcement learning *J. Mach. Learn. Res* **21** 1–51

[81] Asis K D, Hernandez-Garcia J F, Holland G Z and Sutton R S 2017 Multi-step reinforcement learning: a unifying algorithm *Proc. 32nd AAAI Conf. on Artificial Intelligence*
[82] Watkins C J C H 1989 Learning from delayed rewards *PhD Thesis* Cambridge University
[83] Jaakkola T, Jordan M I and Singh S P 1994 On the convergence of stochastic iterative dynamic programming algorithms *Neural Comput.* **6** 1185
[84] Precup D, Sutton R S and Singh S P 2000 Eligibility traces for off-policy policy evaluation *Proc. 17th Int. Conf. on Machine Learning, ICML'00* (San Francisco, CA: Morgan Kaufmann Publishers Inc.) pp 759–66
[85] Sutton R S, Mcallester D, Singh S and Mansour Y 2000 Policy gradient methods for reinforcement learning with function approximation *Proc. 14th Int. Conf. on Neural Information Processing Systems*
[86] Phansalkar V V and Thathachar M A L 1995 Local and global optimization algorithms for generalized learning automata *Neural Comput.* **7** 950–73
[87] Degris T, White M and Sutton R S 2012 Off-policy actor-critic *Proc. 29th Int. Conf. on Machine Learning*
[88] Imani E, Graves E and White M 2018 An off-policy policy gradient theorem using emphatic weightings *Proc. 32nd Int. Conf. on Neural Information Processing Systems, NIPS'18* (Red Hook, NY: Curran Associates Inc.) pp 96–106
[89] Warren P and Allen R 2013 Malliavin weight sampling: a practical guide *Entropy* **16** 221–32
[90] van Seijen H, Mahmood A R, Pilarski P M, Machado M C and Sutton R S 2016 True online temporal-difference learning *J. Mach. Learn. Res.* **17** 1–40
[91] Cichosz P 1995 Truncating temporal differences: on the efficient implementation of td(lambda) for reinforcement learning *J. Artif. Intell. Res.* **2** 287–318
[92] van Seijen H 2016 Effective multi-step temporal-difference learning for non-linear function approximation (arXiv:1608.05151)
[93] Veeriah V, van Seijen H and Sutton R S 2017 Forward actor-critic for nonlinear function approximation in reinforcement learning *Proc. 16th Conf. on Autonomous Agents and MultiAgent Systems*
[94] McCloskey M and Cohen N J 1989 Catastrophic interference in connectionist networks: the sequential learning problem *Psychol. Learn. Motivation* **24** 109–65
[95] Ratcliff R 1990 Connectionist models of recognition memory: constraints imposed by learning and forgetting functions *Psychol. Rev.* **97** 285–308
[96] Kirkpatrick J *et al* 2017 Overcoming catastrophic forgetting in neural networks *Proc. Natl Acad. Sci. USA* **114** 3521–6
[97] Riemer M, Cases I, Ajemian R, Liu M, Tu Y and Tesauro G 2019 Learning to learn without forgetting by maximizing transfer and minimizing interference *Proc. 7th Int. Conf. on Learning Representations*
[98] Ghiassian S, Yu H, Rafiee B and Sutton R S 2018 Two geometric input transformation methods for fast online reinforcement learning with neural nets (arXiv:1805.07476)
[99] Nguyen C, Achille A, Lam M, Hassner T, Mahadevan V and Soatto S 2019 Toward understanding catastrophic forgetting in continual learning (arXiv:1908.01091)
[100] Lo Y L and Ghiassian S 2019 Overcoming catastrophic interference in online reinforcement learning with dynamic self-organizing maps (arXiv:1910.13213)
[101] Mnih V, Kavukcuoglu K, Silver D, Graves A, Antonoglou I, Wierstra D and Riedmiller M 2013 Playing atari with deep reinforcement learning (arXiv:1312.5602)
[102] Mnih V *et al* 2015 Human-level control through deep reinforcement learning *Nature* **518** 529–33
[103] Lillicrap T P, Hunt J J, Pritzel A, Heess N M O, Erez T, Tassa Y, Silver D and Wierstra D 2015 Continuous control with deep reinforcement learning (arXiv:1509.02971)
[104] Wang Z, Bapst V, Heess N, Mnih V, Munos R, Kavukcuoglu K and de Freitas N 2016 Sample efficient actor-critic with experience replay (arXiv:1611.01224)
[105] Lin L-J 1992 Self-improving reactive agents based on reinforcement learning, planning and teaching *Mach. Learn.* **8** 293–321
[106] Daley B and Amato C 2019 Reconciling lambda-returns with experience replay *Proc. 32nd Int. Conf. on Neural Information Processing Systems*
[107] Kakade S 2001 A natural policy gradient *Proc. 15th Int. Conf. on Neural Information Processing Systems, NIPS'01* (Cambridge, MA: MIT Press) pp 1531–8
[108] Peter J, Vijayakumar S and Schaal S 2003 Reinforcement learning for humanoid robotics *Proc. 3rd IEEE-RAS Int. Conf. on Humanoid Robots*
[109] Bagnell J A and Schneider J 2003 Covariant policy search *Proc. 18th Int. Joint Conf. on Artificial Intelligence, IJCAI'03* (San Francisco, CA: Morgan Kaufmann Publishers Inc.) pp 1019–24
[110] Thomas P 2014 Bias in natural actor-critic algorithms *Proc. 31th Int. Conf. on Machine Learning*
[111] Schulman J, Levine S, Abbeel M I, Jordan P and Moritz P 2015 Trust region policy optimization *Proc. 32nd Int. Conf. on Machine Learning*
[112] Schulman J, Wolski F, Dhariwal P, Radford O and Klimov A 2017 Proximal policy optimization algorithms (arXiv:1707.06347)
[113] Wu Y, Mansimov E, Liao S, Grosse R and Ba J 2017 Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation *Proc. 30th Int. Conf. on Neural Information Processing Systems*
[114] Hasselt H V 2010 Double q-learning *Proc. 24th Int. Conf. on Neural Information Processing Systems* ed J D Lafferty, C K I Williams, J Shawe-Taylor, R S Zemel and A Culotta (Red Hook, NY: Curran Associates, Inc.) pp 2613–21
[115] Fujimoto S, van Hoof H and Meger D 2018 Addressing function approximation error in actor-critic methods *Proc. 35th Int. Conf. on Machine Learning*
[116] Silver D, Lever G, Heess N M O, Degris T, Wierstra D and Riedmiller M A 2014 Deterministic policy gradient algorithms *Proc. 31st Int. Conf. on Machine Learning*
[117] Borkar V S 2010 Learning algorithms for risk-sensitive control *Proc. 19th Int. Symp. on Mathematical Theory of Networks and Systems*
[118] Rahme J and Adams R P 2019 A theoretical connection between statistical physics and reinforcement learning (arXiv:1906.10228)
[119] van Hasselt H, Quan J, Hessel M, Xu Z, Borsa D and Barreto A N 2019 General non-linear bellman equations (arXiv:1907.03687)
[120] Naik A, Shariff R, Yasui N, Yao H and Sutton R S 2019 Discounted reinforcement learning is not an optimization problem (arXiv:1910.02140)
[121] Marbach P and Tsitsiklis J N 2003 Approximate gradient methods in policy-space optimization of markov reward processes *Discrete Event Dyn. Syst.* **13** 111–48

[122] Schwartz A 1993 A reinforcement learning method for maximizing undiscounted rewards *Proc. 10th Int. Conf. on Machine Learning*

[123] Tsitsiklis J N and Van Roy B 1999 Average cost temporal-difference learning *Automatica* **35** 1799–808

[124] Kakade S 2001 Optimizing average reward using discounted rewards *Proc. 14th Int. Conf. on Computational Learning Theory* ed D Helmbold and B Williamson (Berlin: Springer) pp 605–15

[125] Bartlett P L and Baxter J 2002 Estimation and approximation bounds for gradient-based reinforcement learning *J. Comput. Syst. Sci.* **64** 133–50

[126] Jack R L and Sollich P 2010 Large deviations and ensembles of trajectories in stochastic models *Prog. Theor. Phys. Suppl.* **184** 304–17

[127] Oakes T, Powell S, Castelnovo C, Lamacraft A and Garrahan J P 2018 Phases of quantum dimers from ensembles of classical stochastic trajectories *Phys. Rev.* B **98** 064302

[128] Casert C, Vieijra T, Whitelam S and Tamblyn I 2020 Dynamical large deviations of two-dimensional kinetically constrained models using a neural-network state ansatz (arXiv:2011.08657)

[129] Bradtke S J and Duff M O 1994 Reinforcement learning methods for continuous-time markov decision problems *Proc. 8th Int. Conf. on Neural Information Processing Systems*

[130] Doya K 2000 Reinforcement learning in continuous time and space *Neural Comput.* **12** 219–45

[131] Munos R 2006 Policy gradient in continuous time *J. Mach. Learn. Res.* **7** 771–91

[132] Vamvoudakis K G and Lewis F L 2010 Online actor–critic algorithm to solve the continuous-time infinite horizon optimal control problem *Automatica* **46** 878

[133] Frémaux N, Sprekeler H and Gerstner W 2013 Reinforcement learning using a continuous time actor-critic framework with spiking neurons *PloS Comput. Biol.* **9** e1003024

[134] Bradtke S J and Barto A G 1996 *Linear Least-Squares Algorithms for Temporal Difference Learning Mach Learn* **22** 33–57

Chapter 7

Training Neural Network Ensembles via Trajectory Sampling

The following work is from the article "*Training neural network ensembles via trajectory sampling*" by **J. F. Mair**, D.C. Rose and J. P. Garrahan submitted for consideration for publication in Physical Review E [142].

In this work, we present a novel technique for jointly training an ensemble of neural networks via trajectory sampling methods, building on the techniques presented in chapters 1 and 4. The focus of our study is on ensembles of models with identical architectures, but distinct parameter sets. We build a trajectory of models to form the ensemble, where adjacent models in the trajectory are coupled by a parameter σ. We bias sampling of trajectories towards low total loss of models (which can be thought of as an energy), coupled with a parameter s which acts like an inverse temperature, and use a TPS-based approach to generate new samples. The ensemble is "trained" together by proposing new trajectories and accepting or rejecting using a metropolis condition, which will eventually converge to sampling the stationary state of the biased dynamics. If we set $\sigma \to \infty$, this is equivalent to independently training each model in the trajectory via neuroevolution, which has a correspondence with stochastic gradient descent (discussed in detail by Whitelam et al [143]). By coupling the models, we are able to efficiently reach lower average losses for the same fixed temperature, and hence, generate a higher performing ensemble of models.

We present an analytical derivation of this technique on an ensemble of linear models (using LDT, for which Chapter 3 is an introduction) on a simple regression task and produce empirical results using TPS, which reproduce these analytic results. Further, we go on to use a more complex Convolution Neural Network as the base architecture for our ensemble and train on the full MNIST [28] problem, showing the powerful improvement in ensemble performance over neuroevolution for the same temperature. This technique is powerful as it is not limited to training only neural network models, but can be considered a *black-box* technique, applicable to a wide range of models that can be represented by a vector of real-valued parameters. However, this work does not address the computational issue with requiring the entire dataset to train at each epoch, which can be prohibitively costly. This shortcoming is addressed in a later work, presented in Chapter 8, via the incorporation of a minibatch update.

arXiv:2209.11116v2 [cond-mat.stat-mech] 10 May 2023

Training neural network ensembles via trajectory sampling

Jamie F. Mair,[1,*] Dominic C. Rose,[2] and Juan P. Garrahan[1,3]

[1] *School of Physics and Astronomy, University of Nottingham, Nottingham, NG7 2RD, UK*
[2] *Department of Physics and Astronomy, University College London, Gower Street, London WC1E 6BT, UK*
[3] *Centre for the Mathematics and Theoretical Physics of Quantum Non-Equilibrium Systems, University of Nottingham, Nottingham, NG7 2RD, UK*

(Dated: May 11, 2023)

In machine learning, there is renewed interest in neural network ensembles (NNEs), whereby predictions are obtained as an aggregate from a diverse set of smaller models, rather than from a single larger model. Here, we show how to define and train a NNE using techniques from the study of rare trajectories in stochastic systems. We define an NNE in terms of the trajectory of the model parameters under a simple, and discrete in time, diffusive dynamics, and train the NNE by biasing these trajectories towards a small *time-integrated* loss, as controlled by appropriate counting fields which act as hyperparameters. We demonstrate the viability of this technique on a range of simple supervised learning tasks. We discuss potential advantages of our trajectory sampling approach compared with more conventional gradient based methods.

I. INTRODUCTION

The traditional approach in machine learning (ML), once the architecture of a model is defined (say a neural network, or NN, composed of layers of coupled neurons), is to learn one set of parameters (say the couplings and biases between the neurons) as optimally as possible from the data; for reviews see e.g. [1, 2]. This is done by minimising a (suitably regularised) objective or loss function of the parameters over a training data set [1, 2]. This optimisation amounts to gradient descent in the landscape defined by the loss function and the training data, often supplemented with tricks that help to speed up convergence, such as adding inertia or stochasticity to the optimisation dynamics [3, 4]. The properties of the training dynamics are controlled by so-called hyperparameters [1, 2]. In this approach, at the end of the learning process one gets a single trained model that is then is used to make inferences [1, 2].

However, there has been recent interest [5, 6] in an alternative approach, where rather than a single model, one trains a set of models. Such ensemble or committee of models [7–13] offers several advantages. Since training by gradient descent can converge to different solutions in a complex loss landscape, training an ensemble of models starting from different initial seeds gives a set of equivalent yet distinct models. Obtaining inferences as the mean or as a majority consensus of the ensemble members can therefore provide better estimates and attenuate uncertainty. Furthermore, an ensemble of smaller models may be more computationally efficient than a single large model, both to train and to run [6].

Here, we introduce a method to define and train neural network ensembles (NNEs) that is based on ideas from the sampling of rare stochastic trajectories. From a statistical mechanics perspective, training one ML model with stochastic gradient descent is equivalent to thermal annealing [14], where the dynamical variables are the model parameters, the training data plays the role of quenched disorder, the loss is the energy, and stochasticity comes from a thermal bath: at low temperature, an annealing dynamics of the ML parameters, such as Monte-Carlo, will converge to a state of low energy and therefore low loss. Analogously, we can think of a NNE in terms of a *trajectory of models*: given some dynamics of the parameters, the set of configurations visited over a period of time defines the ensemble. If we require the trajectory to have low *time-integrated loss* then we can obtain a well-trained NNE. This procedure can be implemented with modern trajectory sampling techniques [15–18], and as we show below, it is a viable way to define and train ML ensembles.

The paper is organised as follows. In Sec. II we review basic concepts about neural networks and NN ensembles. In Sec. III we introduce our method of defining NNEs in terms of trajectories of a stochastic dynamics. Section IV provides exact results for the simple case where the NN architecture is that of a linear perceptron. In Sec. V we show how to train NNEs by means of transition path sampling. In Sec. VI we illustrate our method with application to the linear perceptron, to a two-dimensional loss landscape, and to the textbook problem of classifying handwritten digits in the MNIST data set. Section VII gives our conclusions and outlook.

II. NEURAL NETWORKS AND NEURAL NETWORK ENSEMBLES

A. Neural Networks

Neural networks are computational models which are commonplace throughout ML [1, 2]. They are used as function approximations on problems where the exact structure of the mapping between input and output is unknown. A standard NN maps an input data vector $\boldsymbol{x}$

* jamie.mair@nottingham.ac.uk

to an output vector $\boldsymbol{y}$, via $\boldsymbol{y} = f(\boldsymbol{x})$. The structure of a standard feed-forward NN can be expressed as follows:

$$\boldsymbol{h}^{(i)} = \sigma^i(\boldsymbol{W}^{(i)}\boldsymbol{h}^{(i-1)} + \boldsymbol{b}^{(i)}), \quad (1)$$

where

$$\boldsymbol{h}^{(0)} = \boldsymbol{x} \quad (2)$$

$$\boldsymbol{h}^{(N)} = \boldsymbol{y}. \quad (3)$$

The model is parameterised by the weight matrices and the bias vectors, denoted by $\boldsymbol{W}^{(i)}$ and $\boldsymbol{b}^{(i)}$ respectively. The number of layers in the model is denoted by N, not counting the input layer. The activation function σ^i is any non-linear function, such as a hyperbolic tangent or a rectified linear unit (ReLU). The exception is that the output layer can be allowed to have a linear activation function, to not limit the range of outputs a model, depending on the type of function being approximated. We denote the parameters of the model $\boldsymbol{\theta}$ and the function mapping input to output in the model as $f_{\boldsymbol{\theta}}$. We treat $\boldsymbol{\theta}$ as a flat vector containing all the parameters of the weight matrices and bias vectors.

One can define an ML problem via the specification of a loss function, $L(\boldsymbol{\theta})$, which we view as a function of the model parameters. This translates the problem of finding the optimal parameters that define the model into one of minimizing the loss. Most modern ML problems use a NN as the function approximation, since the structure allows for efficient calculation of the gradient of the loss, with respect to the parameters of the model. These gradients can then be used to reduce the loss through a range of techniques for updating the parameters[4, 19–21]; all of which are variants of basic gradient descent, where parameters are updated via

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}), \quad (4)$$

where α is a learning rate which is used to tune the size of each update.

An alternative to gradient based optimization is gradient-free optimization, such as neuroevolution [14, 22, 23]. Such approaches are usually based on randomised Monte-Carlo changes to the model, which may modify both the parameters and the model structure. Each modification is evaluated using the loss function, and probabilistically accepted or rejected according to some chosen criteria. In particular, thermal annealing with Monte-Carlo can be shown [14] to be analogous to gradient descent.

B. Neural Network Ensembles

Instead of a single trained NN, one can use a collection of trained NNs to make an inference. The different models in such a NN ensemble [5–13] need not share a common structure, but it is important to define a way of combining the individual predictions from the constituent models to form an aggregate prediction. For example, in the case of classification problems, one can give each model in the ensemble a "vote" for each prediction, and the final prediction is the most voted for option. Alternatively, when the inference is a score, the aggregate score could be the mean over the ensemble.

We will focus on the more standard case of NNEs of models with identical structure but differing parameters. We denote the ensemble by the set of parameters in each of the models, $\boldsymbol{\Theta} = \{\boldsymbol{\theta}_t\}_{t=1}^{\tau}$, where τ is the number of models in the ensemble. A loss for the entire ensemble is defined as $\mathcal{L}(\boldsymbol{\Theta}) = \sum_t L(\boldsymbol{\theta}_t)$, where $L(\boldsymbol{\theta}_t)$ is the loss of a single model with parameters $\boldsymbol{\theta}_t$.

III. NEURAL NETWORK ENSEMBLES AS TRAJECTORIES

Our approach is to construct a probability distribution which puts a high weight on ensembles of models with low total loss, from which we then sample effective ensembles. This is achieved by taking inspiration from the trajectory ensemble methods for many-body stochastic processes, based on large deviations [17, 18, 24–27]. In what follows we show how to construct NNEs in terms of appropriately sampled stochastic trajectories of the parameters that define the individual NNs of the ensemble.

We can think of our trajectory NNE construction in two alternative but equivalent ways. The first one is as follows. We think of a NNE as a time ordered *trajectory* of models, $\boldsymbol{\Theta} = \theta_1 \to \theta_2 \cdots \to \theta_\tau$, where the sequence of models is generated by a stochastic dynamics of the model parameters. We start with an *unbiased* dynamics, which for simplicity we choose as a discrete in time random walk in the space of parameters, with symmetric Gaussian steps of variance σ^2. That is, under this unbiased dynamics, the trajectory $\boldsymbol{\Theta}$ has probability

$$P_\sigma(\boldsymbol{\Theta}) = \frac{1}{\mathcal{Z}_\tau(\sigma)} p(\theta_1) \prod_{t=1}^{\tau-1} \exp\left[-\frac{1}{2\sigma^2}(\boldsymbol{\theta}_t - \boldsymbol{\theta}_{t+1})^2\right], \quad (5)$$

Here $p(\theta_1)$ is the initial probability of the parameters of the first model, and $\mathcal{Z}_\tau(\sigma)$ a normalisation. A NNE generated in this way would have an arbitrary loss. In order to generate a useful NNE we wish to select trajectories with low *time-integrated loss*

$$\mathcal{L}(\boldsymbol{\Theta}) = \sum_{t=1}^{\tau} L(\boldsymbol{\theta}_t) \quad (6)$$

This is done by *tilting* Eq.(5) [17, 18, 24–27]

$$
\begin{aligned}
P_{\sigma,s}(\boldsymbol{\Theta}) =& \frac{1}{\mathcal{Z}_\tau(\sigma,s)} p(\theta_1) e^{-sL(\theta_1)} \\
& \times \prod_{t=1}^{\tau-1} \exp\left[-\frac{1}{2\sigma^2}(\boldsymbol{\theta}_t - \boldsymbol{\theta}_{t+1})^2\right] e^{-sL(\theta_t)} \qquad (7)\\
=& \frac{p(\theta_1)}{\mathcal{Z}_\tau(\sigma,s)} e^{-s\mathcal{L}(\boldsymbol{\Theta})} \prod_{t=1}^{\tau-1} \exp\left[-\frac{1}{2\sigma^2}(\boldsymbol{\theta}_t - \boldsymbol{\theta}_{t+1})^2\right],
\end{aligned}
$$

where the normalising factor is the dynamical partition sum, given by

$$
\begin{aligned}
\mathcal{Z}_\tau(\sigma,s) = & \int d\boldsymbol{\Theta}\; p(\theta_1) e^{-s\mathcal{L}(\boldsymbol{\Theta})} \\
& \times \prod_{t=1}^{\tau-1} \exp\left[-\frac{1}{2\sigma^2}(\boldsymbol{\theta}_t - \boldsymbol{\theta}_{t+1})^2\right]. \qquad (8)
\end{aligned}
$$

In $P_{\sigma,s}(\boldsymbol{\Theta})$ the unbiased probabilities of Eq.(5) are re-weighted by an exponential factor in the time-integrated loss.

The probability given by (7) for a trajectory, or NNE, is controlled by the "hyperparameters" σ and s. The first one determines how different subsequent models in the trajectory are, since larger σ corresponds to larger steps in the unbiased diffusive dynamics in parameter space (that is, σ is the conjugate to the "dynamical activity" of the trajectory [25, 28]). The second hyperparameter controls (i.e., is conjugate to) the time-integrated loss, since the larger s, the lower the total loss in the NNE.

Our aim is to sample NNEs from Eq.(7) at large enough s and, therefore, low enough overall ensemble loss. While generating trajectories with the unbiased probability Eq.(5) is done straightforwardly by simply running a diffusive dynamics on the parameters, obtaining trajectories compatible with (7) is more difficult. Difficulty arises as the tilted trajectories correspond to an atypical subset of trajectories of those generated by the diffusive dynamics, one which is exponentially suppressed in τ and in the number of parameters with respect to the typical trajectories. Nevertheless, as we show below, such subset can be efficiently accessed by means of rare event sampling techniques.

A. Connection to stochastic gradient descent

A second way to see the re-weighted, or biased, trajectories in Eq.(7) connects to more traditional approaches for NN training related to stochastic gradient descent. In the limit of $\tau = 1$ the NNE is simply a single model with probability, from Eq.(7),

$$
p_s(\boldsymbol{\theta}) = \frac{1}{\mathcal{Z}_s} e^{-sL(\boldsymbol{\theta})}, \qquad (9)
$$

where in the following we consider only $s > 0$, which allows us to normalise this probability distribution. Equation (9) is the equilibrium probability for a stochastic dynamics obeying detailed balance with respect to energy $L(\boldsymbol{\theta})$ at inverse temperature s, and where $\mathcal{Z}_s = \int d\boldsymbol{\theta} e^{-sL(\boldsymbol{\theta})}$. One such dynamics is so-called neuroevolution [14], in which the parameters of the model are updated by proposing random Gaussian increments, and accepting them with a Metropolis criterion $\min(1, e^{-s\Delta L})$, where ΔL is the change in loss. This process can be shown to be equivalent to stochastic gradient descent when averaged over many runs [14]. For the case of many models, $\tau > 1$, and $\sigma \to \infty$, Eq.(7) describes τ uncoupled models equilibrated under neuroevolution (or similar thermal annealing of the individual losses at inverse temperature s),

$$
P_{\infty,s}(\boldsymbol{\Theta}) = \frac{1}{\mathcal{Z}_\tau(\infty,s)} e^{-s\mathcal{L}(\boldsymbol{\Theta})} \qquad (10)
$$

with $\mathcal{Z}_\tau(\infty,s) = (\mathcal{Z}_s)^\tau$. While Eq.(10) does describe an NNE, all the models in the ensemble are distributed identically and independently, cf. Eq.(6), and the expected loss per model in the NNE is the same as the expected loss of an individual model under Eq.(9).

In order to reduce the NNE loss, one has to couple the different models in Eq.(10). This is precisely what Eq.(7) does when $\sigma < \infty$: in this case σ controls how much each successive model in the ensemble is allowed to differ from the previous one. That this will reduce the total loss of the ensemble can be seen from the fact that in the limit of vanishing σ, all the models have to be the same and Eq.(7) becomes

$$
P_{0,s}(\boldsymbol{\Theta}) \propto \exp[-s\tau L(\boldsymbol{\theta}_1)] \prod_{t=2}^{\tau} \delta(\boldsymbol{\theta}_t - \boldsymbol{\theta}_1) \qquad (11)
$$

so that the NNE is equivalent to a single model equilibrated as in Eq.(9) but at a lower temperature $(s\tau)^{-1}$, and thus a much lower average loss.

B. Training strategy

From studies of other problems with complex optimisation landscapes, such as glasses and spin glasses, it is well known that directly attempting to access low temperature states is riddled with slow convergence problems. This makes training an NNE by simply reducing the temperature, cf. Eq.(10), impractical. In contrast, the combination of the hyperparameters s, conjugate to the loss, and σ, conjugate to the dynamical activity can help overcome the convergence problem, as shown in large deviation studies of glassy systems [16].

Sampling trajectories distributed according to the tilted measure Eq.(7) is our method of training NNEs. The technical problem is that while trajectories are easy to generate via the diffusive dynamics that defines Eq.(5), they are notoriously difficult to generate for Eq.(7), as this represents a subset of rare diffusive trajectories with atypical time-integrated loss for $s > 0$. To do this, we

will employ transition path sampling (TPS) [15], supplemented by convergence-enhancing tricks for trajectory proposals, in order to efficiently sample trajectories from Eq.(7), as explained in detail in Sec. V.

IV. EXACT RESULTS FOR A LINEAR PERCEPTRON

As an elementary and analytically tractable example, we consider a regression problem using a linear perceptron with a mean-squared error (MSE) loss function. The linear perceptron is a model given by $\boldsymbol{y} = W\boldsymbol{x} + \boldsymbol{b}$, which we simplify by defining a modified input vector $\tilde{\boldsymbol{x}}^T = [\boldsymbol{x}^T\ 1]$, in turn defining an expanded weight matrix $\tilde{W} = [W\ \boldsymbol{b}]$. This simplifies the model to $\boldsymbol{y} = \tilde{W}\tilde{\boldsymbol{x}}$, then $\boldsymbol{y} \in \mathbb{R}^k$, $\tilde{\boldsymbol{x}} \in \mathbb{R}^{(d+1)}$ and $\tilde{W} \in \mathbb{R}^{k\times(d+1)}$, where k is the number of target output dimensions and d is the number of dimensions of the feature vector. Given N target labels, $\boldsymbol{y'}$, corresponding to some input features, $\boldsymbol{x}$, we define matrices $\tilde{\boldsymbol{X}} \in \mathbb{R}^{(d+1)\times N}$, $\boldsymbol{Y} = \tilde{W}\tilde{\boldsymbol{X}}$ and $\boldsymbol{Y'} \in \mathbb{R}^{k\times N}$. Using these we write the MSE loss as:

$$L(\boldsymbol{\theta}) = \frac{1}{2N}\mathrm{tr}\left([\boldsymbol{Y} - \boldsymbol{Y'}][\boldsymbol{Y} - \boldsymbol{Y'}]^T\right), \tag{12}$$

which can be further simplified to

$$L(\boldsymbol{\theta}) = \frac{1}{2}\left(\mathrm{tr}\left(\tilde{W}A\tilde{W}^T\right) - 2\mathrm{tr}\left(\tilde{W}B\right) + \mathrm{tr}\left(C\right)\right), \tag{13}$$

where

$$A = \frac{1}{N}\tilde{\boldsymbol{X}}\tilde{\boldsymbol{X}}^T, \tag{14}$$

$$B = \frac{1}{N}\tilde{\boldsymbol{X}}\boldsymbol{Y'}^T, \tag{15}$$

$$C = \frac{1}{N}\boldsymbol{Y'}\boldsymbol{Y'}^T. \tag{16}$$

If we express the trajectory parameters $\boldsymbol{\Theta}$ as a row vector of blocks $\tilde{W}_t$ for each time t, $\boldsymbol{\Theta} \in \mathbb{R}^{k\times\tau(d+1)}$, we can write the partition sum in Eq.(7) as a Gaussian integral

$$\mathcal{Z}_\tau(\sigma, s) = \int D\boldsymbol{\Theta} \exp[-\frac{1}{2}\mathrm{tr}(\boldsymbol{\Theta}\tilde{A}\boldsymbol{\Theta}^T) + \mathrm{tr}(\boldsymbol{\Theta}\tilde{B}) - \frac{s}{2}\tau\mathrm{tr}(C)], \tag{17}$$

where

$$\tilde{A} = \begin{bmatrix} \frac{\mathbb{1}}{\sigma^2} + sA & -\frac{\mathbb{1}}{\sigma^2} & & & 0 \\ -\frac{\mathbb{1}}{\sigma^2} & \frac{\mathbb{1}}{\sigma^2} + 2sA & -\frac{\mathbb{1}}{\sigma^2} & & \\ & -\frac{\mathbb{1}}{\sigma^2} & \ddots & & \\ & & & \frac{\mathbb{1}}{\sigma^2} + 2sA & -\frac{\mathbb{1}}{\sigma^2} \\ 0 & & & -\frac{\mathbb{1}}{\sigma^2} & \frac{\mathbb{1}}{\sigma^2} + sA \end{bmatrix}, \tag{18}$$

and

$$\tilde{B} = \begin{bmatrix} sB \\ \vdots \\ sB \end{bmatrix}. \tag{19}$$

We see that $\tilde{A} \in \mathbb{R}^{\tau(d+1)\times\tau(d+1)}$ and $\tilde{B} \in \mathbb{R}^{\tau(d+1)\times k}$. Note that when $\tau = 1$, $\tilde{A} = sA$, as there is no dependence on σ. It is easy to integrate over trajectories to obtain the partition sum for the linear perceptron

$$\mathcal{Z}_\tau(\sigma, s) = \exp\left\{-\frac{1}{2}\mathrm{tr}(\tilde{B}^T\tilde{A}^{-1}\tilde{B}) - \frac{1}{2}\log\det\tilde{A} + \frac{s}{2}\tau\mathrm{tr}(C)\right\}. \tag{20}$$

The dynamical partition sum Eq.(8) is the moment generating function for the time-integrated loss. From this, one can obtain the average time-integrated loss for arbitrary s [17, 18, 27],

$$\frac{1}{\tau}\mathbb{E}\left[\mathcal{L}(\tilde{\omega})\right] = -\frac{1}{\tau}\partial_s \log \mathcal{Z}_\tau(s). \tag{21}$$

For the linear perceptron problem, we can compute Eq.(21) directly from Eq.(20). The average loss as a function of s is shown in Fig. 1(a,b), for different values of τ and two values of σ.

We observe the following features: (i) for a given τ, the loss per unit time decreases with increasing s; (ii) for fixed s, the loss decreases with τ; and (iii) the loss curves are systematically lower in values the smaller σ. These can be explained from the exact form of the average time-integrated loss of the linear perceptron trajectories. In that expression we find two regimes:

$$\frac{1}{\tau}\mathbb{E}\left[\mathcal{L}(\tilde{\omega})\right] \approx \frac{1}{2}\mathrm{tr}(B^T B) - \frac{1}{2}\mathrm{tr}(C) - \begin{cases} \frac{1}{2s\tau} & s\sigma^2 \ll 1 \\ \frac{1}{2s} & s\sigma^2 \gg 1 \end{cases} \tag{22}$$

The first two terms of are constants, given by samples in the dataset. This specifies the minimum loss achievable, given the data. When $s\sigma^2 \ll 1$, a dependence on τ^{-1} emerges, which is responsible for the banding in Fig.1 for small values of s: a higher value of τ will lead to a lower loss. When $s\sigma^2 \gg 1$, the banding effect becomes negligible, and all values of τ converge to the same loss per unit time for a given s.

The parameter σ controls where the banding occurs, see Fig.1: comparing panels (a) and (b) we see that banding persists in (a) much longer than in (b); the s values at which the curves converge, i.e. the loss of advantage of longer trajectories, differs by a factor of σ^2. [For $\tau = 1$, the conditions in Eq.(22) become the same, and σ plays no role; this is the limit of a single NN trained via gradient descent or neuroevolution, as discussed in Sec. IIIA.]

The interplay between the hyperparameters s, τ and σ provides in our trajectory method a mechanism for balancing between exploitation versus exploration in the training of the NNE. As we will show below, all these features that we can compute exactly for an ensemble of linear perceptrons generalise qualitatively to more complex architectures: the loss of a NNE obtained from our trajectory approach is reduced by increasing s, trajectory length, and decreasing σ.

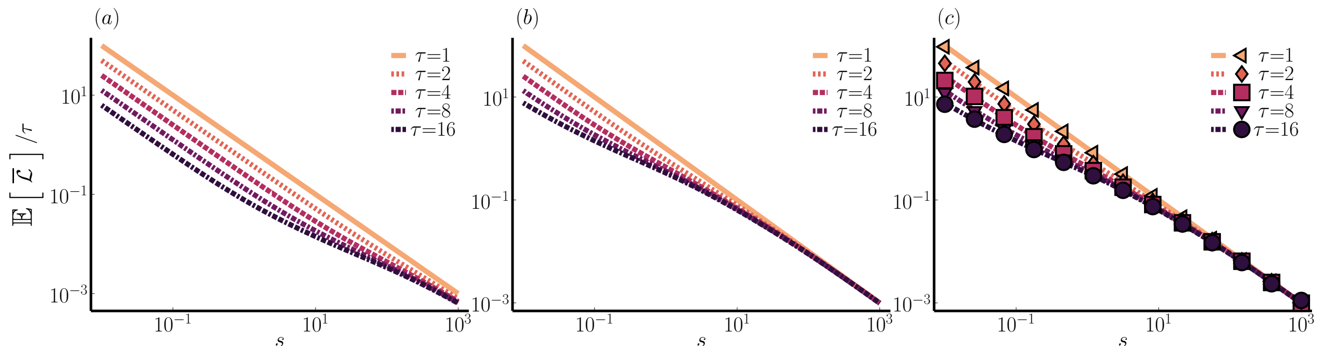

FIG. 1. **NNEs as trajectories in the linear perceptron.** (a) Exact time-averaged loss, Eq.(21), as a function of s for various trajectory lengths τ and $\sigma = 0.1$. (b) Same for $\sigma = 1$. (c) Comparison between exact results (lines) and TPS sampling (symbols). Empirical samples were generated using methods described in Sec. V, using an adaptive annealing technique in s to decrease convergence time, similar to the process shown in Fig.3. Full source code to reproduce this data is supplied in [29, 30].

V. TRAINING NNES WITH TRANSITION PATH SAMPLING

In order to sample good NN ensembles from the biased distribution Eq.(7) we use Transition Path Sampling (TPS) [15], a form of Monte-Carlo in trajectory space that can be used to converge to sample arbitrary rare trajectories from some unbiased dynamics. It is particularly well suited for sampling dynamics where the biased set is due to tilting by a time-integrated function of the trajectories, see e.g. [16].

TPS proceeds as in standard Monte-Carlo by generalising from configurations to trajectories. In our case we wish to sample from $P_{\sigma,s}(\mathbf{\Theta})$, Eq.(7), so the starting point of TPS is to construct transition probabilities $P(\mathbf{\Theta}'|\mathbf{\Theta})$ which satisfies *detailed balance* with respect to that distribution: $P(\mathbf{\Theta}'|\mathbf{\Theta})P_{\sigma,s}(\mathbf{\Theta}) = P(\mathbf{\Theta}|\mathbf{\Theta}')P_{\sigma,s}(\mathbf{\Theta}')$. This guarantees that the stationary distribution of trajectories generated sequentially using $P(\mathbf{\Theta}'|\mathbf{\Theta})$ will converge to those sampled from $P_{\sigma,s}(\mathbf{\Theta})$. The transition probabilities are decomposed into two sub-steps known as the *proposal* and *acceptance-rejection* steps

$$P(\mathbf{\Theta}'|\mathbf{\Theta}) = g(\mathbf{\Theta}'|\mathbf{\Theta})A(\mathbf{\Theta}',\mathbf{\Theta}), \tag{23}$$

where $g(\mathbf{\Theta}'|\mathbf{\Theta})$ is the conditional probability of proposing a trajectory $\mathbf{\Theta}'$ given a current trajectory of $\mathbf{\Theta}$. The acceptance probability $A(\mathbf{\Theta}',\mathbf{\Theta})$ then specifies how likely it is to accept the proposed trajectory, given the current trajectory. Inserting this decomposition into the detailed balance leads to a relation which the acceptance must satisfy for the combined dynamics to possess the desired detailed balance:

$$\frac{A(\mathbf{\Theta}',\mathbf{\Theta})}{A(\mathbf{\Theta},\mathbf{\Theta}')} = \frac{P_{\sigma,s}(\mathbf{\Theta}')}{P_{\sigma,s}(\mathbf{\Theta})}\frac{g(\mathbf{\Theta}|\mathbf{\Theta}')}{g(\mathbf{\Theta}'|\mathbf{\Theta})}. \tag{24}$$

A very common choice for the acceptance ratio which fulfils the above expression is the Metropolis one

$$A(\mathbf{\Theta}',\mathbf{\Theta}) = \min\left(1, \frac{P_{\sigma,s}(\mathbf{\Theta}')}{P_{\sigma,s}(\mathbf{\Theta})}\frac{g(\mathbf{\Theta}|\mathbf{\Theta}')}{g(\mathbf{\Theta}'|\mathbf{\Theta})}\right). \tag{25}$$

Given that only ratios of the target distribution appear in the acceptance probability, we can simplify the above expression by choosing proposal moves that satisfy detailed balance with respect to the unbiased probability (5),

$$P_\sigma(\mathbf{\Theta}')g(\mathbf{\Theta}|\mathbf{\Theta}') = P_\sigma(\mathbf{\Theta})g(\mathbf{\Theta}'|\mathbf{\Theta}). \tag{26}$$

Inserting into (25) we get

$$A(\mathbf{\Theta}',\mathbf{\Theta}) = \min\left(1, e^{-s[\mathcal{L}(\mathbf{\Theta}')-\mathcal{L}(\mathbf{\Theta})]}\right). \tag{27}$$

The above means that trajectories are generated with the unbiased dynamics, Eq.(26), and accepted or rejected according to the change in time-integrated loss, Eq.(27).

While generating trajectories with the unbiased dynamics is a big simplification, for two arbitrary trajectories the difference in time-integrated loss is extensive in time and (at least) also extensive in number of parameters, making acceptance (27), in general, exponentially small. For that matter, most of the art in TPS is to design moves that both satisfy detailed balance in trajectory space and make acceptance efficient.

A. Generating dynamics: shooting plus Brownian bridges

A very common choice for generating trajectory moves in TPS is *shooting* [15], which involves choosing a fixed time in the trajectory and a direction, forwards or backwards, and evolving with the original unbiased dynamics until reaching the end of the trajectory (with detailed balance, a backward shooting move can be generated forwards and time-reversed). Shooting with the original dynamics Eq.(5) is outlined in Algorithm Alg.1. Under shooting, g obeys Eq.(26) and therefore Eq.(27) is the corresponding acceptance probability. As shooting leaves a portion of the trajectory unchanged, one need only calculate the change in loss of the modified portions

of the trajectory. The shooting method is sketched in Fig.2(a,b).

Algorithm 1 Shooting TPS

1: **input** Current trajectory $\boldsymbol{\Theta}$
2: **parameters** Variance of the Gaussian noise σ^2
3: Choose t uniformly from $[1, \tau]$, where t is an integer
4: Choose a direction randomly $\kappa \in \{-1, 1\}$
5: Initialise $\boldsymbol{\Theta}' \leftarrow \boldsymbol{\Theta}$
6: **while** $t + \kappa \geq 1$ **and** $t + \kappa \leq \tau$ **do**
7: Sample all elements of $\Delta\boldsymbol{\theta}'$ i.i.d. from $\mathcal{N}(0, \sigma^2)$
8: $\boldsymbol{\theta}'_{t+\kappa} \leftarrow \boldsymbol{\theta}'_t + \Delta\boldsymbol{\theta}'$
9: $t \leftarrow t + \kappa$
10: **end while**
11: **output** $\boldsymbol{\Theta}'$

A common issue with shooting is that it is difficult to generate successful updates towards the centre of a trajectory. The reason is that when shooting from the bulk of the trajectory, the difference in time-integrated loss between the proposed and current trajectories scales with the trajectory length, making acceptance exponentially suppressed with time. To address this issue, we exploit the fact that the unbiased dynamics we use to generate moves is Brownian: we supplement shooting with moves generated by *Brownian bridges*, see e.g. [31, 32], that is, a proposed move consists of replacing a portion of the trajectory by a bridge between the same initial and final points of the replaced segment, see sketch in Fig.2(c). By controlling the time extent of the bridge, we can attenuate the loss difference in Eq.(27) thus enhancing acceptance. Details on the Brownian bridges is given in the Appendix. The key formulae are as follows: if the portion of the trajectory to replace is between times t_1 and t_2, keeping x_{t_1} and x_{t_2} fixed, by generating a bridge with the dynamics

$$P_{\mathrm{B}}\left(x_t | x_{t-1}, t\right) = \frac{e^{-\frac{[x_t - \mu(x_{t-1}, t)]^2}{2v(t)}}}{\sqrt{2\pi v(t)}}, \tag{28}$$

with *time-dependent* mean and variance

$$\mu(x, t) = \frac{x_{t_2} + (t_2 - t)x}{t_2 - t + 1}, \tag{29}$$

$$v(t) = \sigma^2 \frac{t_2 - t}{t_2 - t + 1}. \tag{30}$$

We demonstrate how a bridge is generated in Alg.2. One has the choice of how to generate t_1 and t_2. In our simulations, we favour choosing t_1 uniformly from $[1, \tau - 2]$ and then setting $t_2 = t_1 + 2$, to give a bridge of a single time step, so there is only one updated state.

Algorithm 2 Brownian bridges for TPS

1: **input** Current trajectory $\boldsymbol{\Theta}$
2: **parameters** Variance of the Gaussian noise σ^2
3: Choose t_1 and t_2 uniformly from $[1, \tau]$, where $t_1, t_2 \in \mathcal{Z}$ and $t_2 > t_1$.
4: Initialise $\boldsymbol{\Theta}' \leftarrow \boldsymbol{\Theta}$
5: Initialise $t \leftarrow t_1 + 1$
6: **while** $t < t_2$ **do**
7: Calculate mean $\boldsymbol{\mu}_t$ for $(\boldsymbol{\theta}'_{t-1}, t)$ from Eq.(29)
8: Calculate variance $v(t)$ from Eq.(30)
9: Sample θ'_a from $\mathcal{N}[(\boldsymbol{\mu}_t)_a, v_t]$ for all components a
10: $\boldsymbol{\theta}'_t \leftarrow \boldsymbol{\theta}'$
11: $t \leftarrow t + 1$
12: **end while**
13: **output** $\boldsymbol{\Theta}'$

One cannot guarantee ergodicity in trajectory space using only bridges, as it requires two ends to be fixed, meaning the end of the trajectories will not be modified. Instead, we use a combined approach consisting of choosing the shooting algorithm (Alg.1) with probability p_{shoot} or the bridge algorithm (Alg.2) with $1 - p_{\mathrm{shoot}}$. For shorter trajectories $\tau \leq 4$, we choose $p_{\mathrm{shoot}} = 1$, as the trajectories are not long enough for centre trajectory updates to become inefficient. When $\tau > 4$, we set $p_{\mathrm{shoot}} = \frac{2}{\tau}$ and modify the shooting algorithm to only shoot forwards from $t = \tau - 1$ or backwards from $t = 2$ with equal probability, and choose bridges that only alter a single time in the trajectory as described earlier. This choice was to improve acceptance, and ensure that each model in the trajectory had an equal probability of being mutated.

VI. NUMERICAL RESULTS

A. Linear perceptron

As an elementary demonstration of our TPS scheme, we applied it to the linear perceptron model of Sec. IV. Figure 1(c) shows that TPS reproduces the exact results for the average time-integrated loss as a function of s.

The quantity of interest is the expected time-averaged loss of the NNE. We can estimate this quantity as a running average of the TPS iterations, or "epochs" of the training,

$$\mathbb{E}\left[\mathcal{L}(\boldsymbol{\Theta})\right] \approx \frac{1}{M} \sum_{m=M_{\mathrm{rel}}}^{M_{\mathrm{rel}}+M} \mathcal{L}(\boldsymbol{\Theta}^{(m)}), \tag{31}$$

where $\boldsymbol{\Theta}^{(m)}$ is the parameter trajectory at TPS epoch m. In the above, the time-integrated loss is an empirical average over M epochs, calculated after allowing TPS to relax for M_{rel} epochs, large enough so that TPS converges to stationarity. In the limit of $M \to \infty$ Eq.(31) becomes an equality. As is standard practice, we check for TPS relaxation empirically. A common technique for speeding up convergence is to anneal the s parameter,

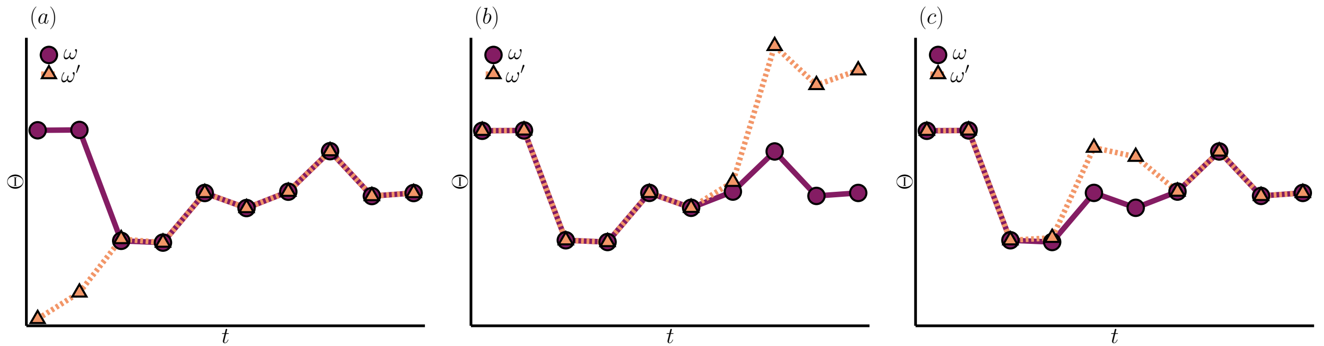

FIG. 2. **TPS scheme.** The original trajectory is ω and the proposed new trajectory is ω'. (a) Backwards shooting, fixing $t \geq 4$. (b) Forwards shooting, fixing $t \leq 6$. (c) Brownian bridge, fixing $t \leq 2$ and $t \geq 7$.

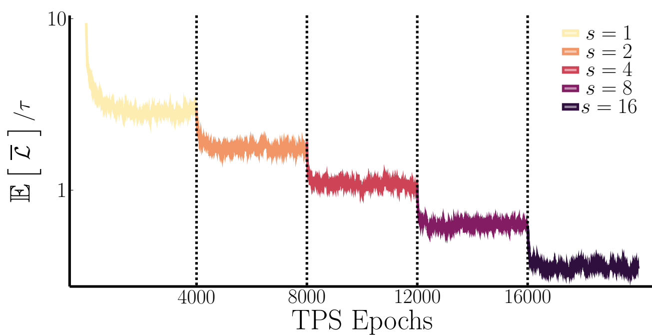

FIG. 3. **TPS annealing.** Training of the NNE for the linear perceptron through stepwise decrease of s, for $\tau = 8$ and $\sigma = 1.0$.

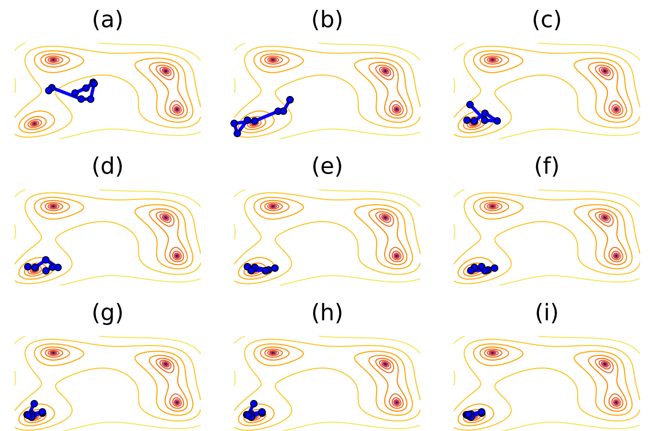

FIG. 4. **Trajectory evolution under TPS.** Progression of the NNE training for the 2D loss landscape. Snapshots of the trajectory every 50 TPS epochs.

starting at a low s (i.e. close to the unbiased dynamics), and progressively increasing s towards the desired value; see Fig.3.

B. Two-dimensional loss landscape

To illustrate how TPS works on a trajectory representing a model, and in particular how biasing the diffusive dynamics allows for sampling low loss trajectories, we consider a simple, yet non-trivial, toy problem where the model has only two parameters.

We consider the Hummelblau's function:

$$h(x, y) = (x^2 + y - 11)^2 + (x + y^2)^2, \qquad (32)$$

and choose x and y to be within $[-5, 5]$. The aim is to train an ensemble of models, where each model has parameters $\theta = (x, y)$ and $h(x, y)$ is the loss. We can then construct a trajectory, $\mathbf{\Theta}$, which is initialised under Gaussian dynamics with $\sigma = 1.0$, with $\theta_0 = (x_0, y_0)$ randomly chosen from $[-1, 1] \times [-1, 1]$. As before, we choose the observable of a trajectory of length τ to be the time-integrated loss, $\mathcal{L}(\mathbf{\Theta}) = \sum_{t=1}^{\tau} h(x_t, y_t)$.

Figure 4 shows the evolution of the ensemble trajectory under TPS. The first snapshot in panel (a) is the randomly initialised trajectory, and progress is shown every 50 TPS iterations. We see that as TPS progresses, the trajectory evolves towards being localised in the minimum of the loss landscape. This represents the training of the ensemble by means of trajectory sampling.

In this simple example, it is easy to visualize two features of the training. First, loss landscapes can have a natural length scale for each dimension, which justifies a choice of σ to enable a high enough acceptance rate. Second, both s and σ dictate the evolution of a trajectory under TPS, and therefore the training of the ensemble. A lower value of s and a higher value of σ means that the trajectory is free to spread out across the loss landscape, fairly unimpeded. A higher value of s and a lower value of σ restricts a trajectory to cover a small range of the loss landscape, making the trajectory much more likely to be localised in a local minimum.

C. MNIST classification

As a final example, we consider a standard ML classification task. In this case the relevant loss function is the mean cross entropy loss over the data. For a single NN this reads

$$L(\boldsymbol{\theta}) = -\frac{1}{N_{\rm sp}} \sum_{i=1}^{N_{\rm sp}} \sum_{j=1}^{N_{\rm cl}} \delta_{z_i,j} \ln y_j(x_i;\boldsymbol{\theta}), \tag{33}$$

where x_i and z_i represent the features and class label index of the $i^{\rm th}$ data point (in a dataset of size $N_{\rm sp}$), and $y_j(x_i;\boldsymbol{\theta})$ is the probability of selecting class j (out of $N_{\rm cl}$ classes) using the NN parameterised by $\boldsymbol{\theta}$. This probability is usually calculated using a softmax layer on the end of the NN. If a model predicts that all classes have an equal probability, then the mean cross entropy loss is $\ln(N_{\rm cl})$.

We now consider the classification of handwritten digits from the MNIST [33] dataset, a standard benchmark for many ML algorithms. Examples of each of the 10 digits in MNIST are shown in Fig.5(a). MNIST classification is a typical example of a problem where one would apply the cross entropy loss function (33) to train a NN. Being high dimensional, this is also a good problem to test our method of producing trained NNEs via trajectory sampling.

As we are interested in studying the training process under our ensemble method, we have simplified the problem by restricting the dataset to 2048 samples, down from the usual 60,000. The distribution of each digit is uniform in the sampled training set. The basic architecture of each NN in the ensemble uses a mix of convolution and fully connected layers, based on LeNet[34]. The aim is to train the NNE to output a probability of selecting a digit, based on the softmax distribution of the final output layer of the NN. The cross-entropy loss function, Eq.(33), is calculated using the logits (the input to the softmax activation) for numerical stability.

In a classification problem, one uses the loss function as a proxy for accuracy. These two measures are highly correlated as a low loss likely correlates with high accuracy. Effectively, to achieve a high accuracy on the training dataset, there is a low loss region which should be reached. For training the NNE via trajectory sampling, we tuned the range of s such that the models converged to a low enough loss region, which corresponded to the full range of accuracies. For the MNIST problem, this range was between $s = 5$ and $s = 50$. To provide banding in the chosen s domain, we used $\sigma = 0.05$ to effectively shift the banding region into these higher values of s. Reducing the value of σ provides the additional benefit of making smaller updates in TPS, leading to a higher acceptance rate and speeding TPS convergence.

We ran a combination of shooting and bridging TPS, as detailed previously, for a number of independent trajectories until the time-averaged loss function appeared numerically converged, similar to what can be seen in Fig.3. Figure 5(b) shows the time-integrated loss per unit time as a function of s of the trained NNEs. We can see an analogous banding to that of the simple linear perceptron: larger values of s have lower ensemble loss, and this is significantly enhanced by increasing the trajectory length.

In Fig.5(b) we also see differences with the perceptron arising from the non-polynomial form of the loss, Eq.(33). First is the effect of the minimum possible value of the loss that the chosen NN architecture can reach. This minimum should be approached by curves as $s \to \infty$. In Fig.5(c) we show that this tends towards 100% accuracy of the NNE, already achieved for $s = 50$ and $\tau = 32$. The second difference is in the small s regime, where the loss function exerts less influence over the sampling. Since under the unbiased dynamics each output logit is randomly distributed in the long time limit, for vanishing s the mean cross-entropy loss per unit time will converge to $\ln(N_{\rm cl})$, provided that the classes are balanced. This produces an upper loss plateau in the dynamics. Increasing the loss beyond this point requires a model to be "intelligently wrong", by lowering the probability of choosing the correct answer beyond uncorrelated random chance. In contrast, for a polynomial loss function as in the linear perceptron case, the loss diverges as $s \to 0$, something that does not occur in a classification problem like this MNIST.

The highest value of s we used for the MNIST problem is large enough to train the longer trajectories to very low losses, which correspond to very high accuracies, see Fig.5(c). While we could in principle reduce the losses to arbitrarily small values, this often causes overfitting, reducing the ability of the models to generalise to unseen examples, see e.g. [1, 2]. Additionally, there is significant cost in using higher values of s, as they reduce the acceptance rate, which can significantly increase training time and cost. We demonstrated that our method is capable of training a standard convolutional neural network to high train accuracy on a subset of the MNIST problem, as seen in Fig.5(c).

VII. CONCLUSIONS

Here, we have presented a method to train neural network ensembles using trajectory sampling techniques more often applied in the statistical mechanics of non-equilibrium systems. In our approach the set of neural networks that form the NNE corresponds to the sequence of configurations of the NN parameters that are visited in a stochastic trajectory. By biasing trajectories to have low time-integrated loss we showed we could train NNEs to perform well in standard machine learning tasks such as MNIST classification. For concreteness, we focused on trajectories from dynamics which is discrete in time, continuous in space, and where the changes at each time step are synchronous, in the sense that all parameters can be updated simultaneously. None of these is a requirement:

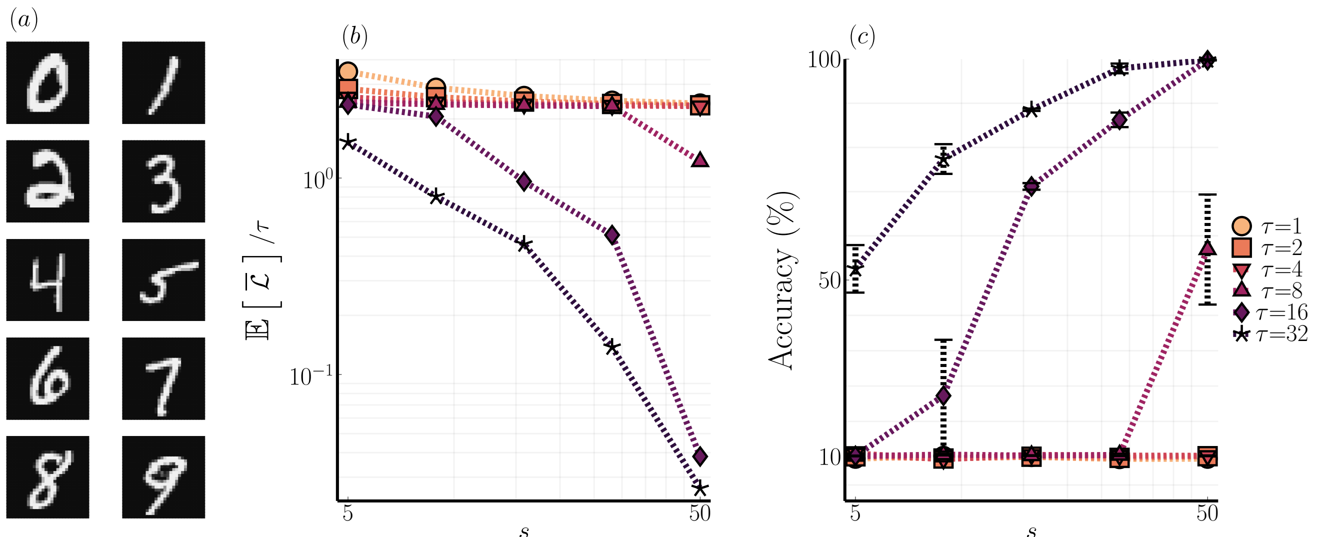

FIG. 5. **NN ensembles training for MNIST.** (a) Representative images for the $N_{\rm cl} = 10$ classes of the MNIST dataset. Each digit is a 28×28 greyscale image, with the brightness of each pixel encoded between 0.0 (black) and 1.0 (white). (b) Time-averaged cross entropy loss for different values of s and τ, for $\sigma = 0.05$. (c) Time-averaged mean accuracy from the NNE at the final TPS epoch of the training, for the same hyperparameters of (b). The accuracy expected for a random NN is around 10%, as there are 10 classes in the MNIST problem. This corresponds to the loss plateau at low s for small τ.

the underlying dynamics can equally be taken to be continuous in time, as in a continuous-time Markov chain or in diffusions, and parameter updates do not need to be Gaussian; in such a case, the rest of the approach for training and sampling would be essentially the same as the one above.

Our trajectory NNE method has to be compared to those based on gradient descent, which focus on the shortest route to a set of parameters for a NN which locally minimise the loss. In contrast, our method is in the spirit of thermal sampling, where low loss configurations are searched by controlling a parameter (such as temperature in thermal annealing, or s in our trajectory method) which being coupled to the quantity of interest (energy or time-integrated loss) pushes towards low values, balanced with exploring state or trajectory space. An obvious drawback of gradient descent is its inability to escape local traps, and this is the reason that modern ML supplements it with noise and inertia to make it efficient. Gradient-free methods like ours are less sensitive to local trapping, which is especially prominent in smaller and non-over-parameterised NNs, with their physical interplay between minimising the observable and maximising the entropy playing an analogous role as the exploit/explore trade-off of ML learning techniques.

The above, together with the ability of NNEs to reduce overfitting, suggest to us that the approach proposed here will be most useful when constructing ensembles of smaller models as compared to a single, larger, NN. While in this paper we focused only on introducing the trajectory NNE method and showing its viability, we hope to report in future work on systematic comparison on performance and training cost between our NNEs and a single NN.

Our approach relies on converging to the stationary state in trajectory space determined by the hyperparameters s, σ and τ. Their meaning is clear: s controls the level of the ensemble loss, with larger s leading to lower overall loss; σ controls exploration, with larger σ allowing for larger fluctuations in the trajectory of models; and τ determines the size of the ensemble. While increasing s, decreasing σ, and increasing τ all reduce the NNE loss, the ability to control the three hyperparameters separately provides much flexibility for the training. The same applies to numerical "convergence": while ideally one would like to sample trajectories from the stationary state of Eq.(7), in practice all that is required is that the TPS iterations reach trajectories of low enough loss for the problem at hand.

The effectiveness of TPS relies on a reasonable acceptance for proposed trajectory updates. In general, acceptance is exponentially suppressed in trajectory length and size of the system. We resolved the exponential in time problem by proposing bridge moves which are localised in time. For a NN system where the loss is a fully connected function of the weights, the exponential in size cost problem is more difficult to solve. For training NNEs with larger NN constituents this might become a limiting factor. A related issue is that of batching the data when calculating the loss: in the ML/statistical mechanics analogy, under learning dynamics the parameters of the NN are the fluctuating variables, while the training dataset is like quenched disorder in the interactions defining the loss. Using data batches to calculate the loss (a

standard trick in ML that gives rise to stochastic gradients), is equivalent to having (slow) fluctuating disorder, something which has not been studied in as much detail in the context of trajectory sampling. Further integration of ML and non-equilibrium ideas will help improve the trajectory NN ensemble even further. We hope to report on such developments in the future.

ACKNOWLEDGMENTS

We acknowledge financial support from the Leverhulme Trust Grant RPG-2018-181 and University of Nottingham grant no. FiF1/3. DCR was supported by funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 853368). Simulations were performed using the University of Nottingham Augusta HPC cluster, and the Sulis Tier 2 HPC platform hosted by the Scientific Computing Research Technology Platform at the University of Warwick. Sulis is funded by EPSRC Grant EP/T022108/1 and the HPC Midlands+ consortium. We thank the creators and community of the Julia programming language [35], and acknowledge use of the packages `CUDA.jl` [36, 37], `Flux.jl` [38, 39], `Plots.jl` [40] and `ForwardDiff.jl` [41]. Our TPS implementation package is available through GitHub, `TransitionPathSampling.jl` [29], together with the source code to generate the figures and results in the paper [30].

APPENDIX: EXACT BRIDGE DYNAMICS FOR A DISCRETE TIME GAUSSIAN PROCESS

Consider an original dynamics of a position $x \in \mathbb{R}$ given by gaussian movements with variance v at each time step

$$P(x'|x) = \int \frac{dw}{\sqrt{2v\pi}} e^{-\frac{w^2}{2v}} \delta(x' - x - w), \tag{34}$$

$$= \frac{e^{-\frac{(x'-x)^2}{2v}}}{\sqrt{2v\pi}}, \tag{35}$$

and initial probability distribution $P(x_0) = \delta(x_0 - x_i)$. The probability of an individual trajectory $\omega_0^T = \{x_t\}_{t=0}^T$ of length T is given by

$$P\left(\omega_0^T\right) = \prod_{t=1}^{T} P(x_t|x_{t-1}) P(x_0). \tag{36}$$

We seek a dynamics which produced, with the correct relative probabilities, the subset of trajectories given by this dynamics such that they all end at $x_T = x_f$, so-called bridge trajectories. That is, we seek a Markovian dynamics which generates trajectories with probability

$$P_B\left(\omega_0^T\right) = \frac{\delta(x_T - x_f) P\left(\omega_0^T\right)}{\sum_{\omega_0^T} \delta(x_T - x_f) P\left(\omega_0^T\right)}. \tag{37}$$

We can expand this trajectory probability using the probabilistic chain rule as

$$P_B\left(\omega_0^T\right) = \prod_{t=1}^{T} P_B(x_t|\omega_0^{t-1}) P_B(x_0), \tag{38}$$

solving for these probabilities, which will turn out to be Markovian, iteratively. For the last time step we find

$$P_B\left(x_T|\omega_0^{T-1}\right) = \frac{P_B\left(\omega_0^T\right)}{P_B\left(\omega_0^{T-1}\right)} \tag{39}$$

$$= \frac{\delta(x_T - x_f) P\left(\omega_0^T\right)}{\int dx_T \delta(x_T - x_f) P\left(\omega_0^T\right)} \tag{40}$$

$$= \frac{\delta(x_T - x_f) P\left(x_T|x_{T-1}\right)}{\int dx_T \delta(x_T - x_f) P\left(x_T|x_{T-1}\right)} \tag{41}$$

$$= \delta(x_T - x_f) := P_B\left(x_T|x_{T-1}, T\right), \tag{42}$$

while for the rest we find

$$P_B\left(x_t|\omega_0^{t-1}\right) = \frac{P_B\left(\omega_0^t\right)}{P_B\left(\omega_0^{t-1}\right)} \tag{43}$$

$$= \frac{\sum_{\omega_{t+1}^T} \delta(x_T - x_f) P\left(\omega_0^T\right)}{\sum_{\omega_t^T} \delta(x_T - x_f) P\left(\omega_0^T\right)} \tag{44}$$

$$= \frac{g(x_t, t) P(x_t|x_{t-1})}{g(x_{t-1}, t-1)} \tag{45}$$

$$:= P_B\left(x_t|x_{t-1}, t\right) \tag{46}$$

where we have defined

$$g(x_t, t) = \sum_{\omega_{t+1}^T} \delta(x_T - x_f) P\left(\omega_{t+1}^T|x_t\right). \tag{47}$$

Finding these scaling factors thus returns the desired Markovian dynamics. First, for $g(x, T-1)$ we find

$$g(x, T-1) = \int dx' \delta(x_T - x_f) P(x'|x) = \frac{e^{\frac{(x_f - x)^2}{2v}}}{\sqrt{2v\pi}}. \tag{48}$$

To find the rest, we note that these scaling factors satisfy an inductive equation, a non-linear Bellman equation, due to the normalization of the dynamics

$$g(x, t-1) = \sum_{x'} P(x'|x) g(x', t), \tag{49}$$

which we can solve inductively. We consider the ansatz

$$g(x, T-i) = \frac{e^{\frac{1}{v}\left(a_i x_f^2 + b_i x^2 + c_i x_f x\right)}}{n_i \sqrt{2v\pi}}, \tag{50}$$

for $i \geq 1$, with $n_1 = 1$, $a_1 = -\frac{1}{2}$, $b_1 = -\frac{1}{2}$, $c_1 = 1$. Using the Bellman equation (49) we thus find

$$n_{i+1} = n_i \sqrt{1 - 2b_i}, \tag{51}$$

$$a_{i+1} = a_i - \frac{c_i^2}{4b_i - 2}, \tag{52}$$

$$b_{i+1} = -\frac{1}{2} - \frac{1}{4b_i - 2}, \tag{53}$$

$$c_{i+1} = -\frac{2c_i}{4b_i - 2}, \tag{54}$$

which are solved by

$$n_i = \sqrt{i}, \tag{55}$$

$$a_i = -\frac{1}{2i}, \tag{56}$$

$$b_i = -\frac{1}{2i}, \tag{57}$$

$$c_i = \frac{1}{i}. \tag{58}$$

Substituting into (45) and rearranging we find a Gaussian with time and position dependent mean and time dependent variance

$$P_B\left(x_t | x_{t-1}, t\right) = \frac{e^{-\frac{\left[x_t - \mu(x_{t-1}, t)\right]^2}{2v(t)}}}{\sqrt{2\pi v(t)}}, \tag{59}$$

where

$$\mu(x, t) = \frac{x_f + (T - t)x}{T - t + 1}, \tag{60}$$

$$v(t) = v\frac{T - t}{T - t + 1} \tag{61}$$

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT Press, 2016).

[2] P. Mehta, M. Bukov, C.-H. Wang, A. G. Day, C. Richardson, C. K. Fisher, and D. J. Schwab, A high-bias, low-variance introduction to machine learning for physicists, Phys. Rep. **810**, 1 (2019).

[3] L. Bottou, Stochastic learning, in *Summer School on Machine Learning* (Springer, 2003) pp. 146–168.

[4] D. P. Kingma and J. Ba, Adam: A method for stochastic optimization, arXiv:1412.6980 (2014).

[5] Y. Wen, D. Tran, and J. Ba, Batchensemble: an alternative approach to efficient ensemble and lifelong learning, arXiv:2002.06715 (2020).

[6] X. Wang, D. Kondratyuk, E. Christiansen, K. M. Kitani, Y. Alon, and E. Eban, Wisdom of committees: An overlooked approach to faster and more accurate models, arXiv:2012.01988 (2020).

[7] L. Hansen and P. Salamon, Neural network ensembles, IEEE Trans. Pattern Anal. Mach. Intell. **12**, 993 (1990).

[8] M. P. Perrone and L. N. Cooper, *When networks disagree: Ensemble methods for hybrid neural networks*, Tech. Rep. (Brown Univ Providence Ri Inst for Brain and Neural Systems, 1992).

[9] A. Krogh and J. Vedelsby, Neural network ensembles, cross validation, and active learning, Adv Neural Inf Process Syst. **7** (1994).

[10] D. Opitz and R. Maclin, Popular ensemble methods: An empirical study, J. Artif. Intell. **11**, 169 (1999).

[11] B. Lakshminarayanan, A. Pritzel, and C. Blundell, Simple and scalable predictive uncertainty estimation using deep ensembles, Adv Neural Inf Process Syst. **30** (2017).

[12] G. Huang, Y. Li, G. Pleiss, Z. Liu, J. E. Hopcroft, and K. Q. Weinberger, Snapshot ensembles: Train 1, get m for free, arXiv:1704.00109 (2017).

[13] T. Kurutach, I. Clavera, Y. Duan, A. Tamar, and P. Abbeel, Model-ensemble trust-region policy optimization, arXiv:1802.10592 (2018).

[14] S. Whitelam, V. Selin, S.-W. Park, and I. Tamblyn, Correspondence between neuroevolution and gradient descent, arXiv:2008.06643 (2021).

[15] P. G. Bolhuis, D. Chandler, C. Dellago, and P. L. Geissler, Transition path sampling: throwing ropes over rough mountain passes, in the dark, Annu. Rev. Phys. Chem. **53**, 291 (2002).

[16] L. O. Hedges, R. L. Jack, J. P. Garrahan, and D. Chandler, Dynamic order-disorder in atomistic models of structural glass formers, Science **323**, 1309 (2009).

[17] J. P. Garrahan, Aspects of non-equilibrium in classical and quantum systems: Slow relaxation and glasses, dynamical large deviations, quantum non-ergodicity, and open quantum dynamics, Physica A **504**, 130 (2018).

[18] R. L. Jack, Ergodicity and large deviations in physical systems with stochastic dynamics, arXiv:1910.09883 (2019).

[19] M. Riedmiller and H. Braun, A direct adaptive method for faster backpropagation learning: The rprop algorithm, in *IEEE INTERNATIONAL CONFERENCE ON NEURAL NETWORKS* (1993) pp. 586–591.

[20] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, On the importance of initialization and momentum in deep learning, in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13 (JMLR.org, 2013) pp. III–1139–III–1147.

[21] A. Graves, Generating sequences with recurrent neural networks, arXiv:1308.0850 (2013).

[22] G. Morse, Training neural networks through the integration of evolution and gradient descent (2019).

[23] N. M. Rodrigues, S. Silva, and L. Vanneschi, A study of generalization and fitness landscapes for neuroevolution, IEEE Access **8**, 108216 (2020).

[24] V. Lecomte, C. Appert-Rolland, and F. van Wijland, Thermodynamic formalism for systems with markov dynamics, J. Stat. Phys. **127**, 51 (2007).

[25] J. P. Garrahan, R. L. Jack, V. Lecomte, E. Pitard, K. van Duijvendijk, and F. van Wijland, Dynamical first-order phase transition in kinetically constrained models of glasses, Phys. Rev. Lett. **98**, 195702 (2007).

[26] J. P. Garrahan, R. L. Jack, V. Lecomte, E. Pitard, K. van Duijvendijk, and F. van Wijland, First-order dynamical phase transition in models of glasses: an approach based on ensembles of histories, J. Phys. A **42**, 075007 (2009).

[27] H. Touchette, The large deviation approach to statistical mechanics, Phys. Rep. **478**, 1 (2009).

[28] C. Maes, Frenesy: Time-symmetric dynamical activity in nonequilibria, Phys. Rep. **850**, 1 (2020).

[29] jl package (2022), available at `https://github.com/JamieMair/TransitionPathSampling.jl`.

[30] Neural Network Ensembles (NNEs), as realised through trajectory sampling (2022), available at `https://github.com/JamieMair/nne-trajectory-sampling-code`.

[31] J. Grela, S. N. Majumdar, and G. Schehr, Non-intersecting brownian bridges in the flat-to-flat geometry, J. Stat. Phys. **183**, 49 (2021).

[32] B. De Bruyne, S. N. Majumdar, and G. Schehr, Generating discrete-time constrained random walks and lévy flights, Phys. Rev. E **104**, 024117 (2021).

[33] L. Deng, The mnist database of handwritten digit images for machine learning research, IEEE Signal Process. Mag. **29**, 141 (2012).

[34] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, Gradient-based learning applied to document recognition, Proc. IEEE **86**, 2278 (1998).

[35] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, Julia: A fresh approach to numerical computing, SIAM Rev Soc Ind Appl Math **59**, 65 (2017).

[36] T. Besard, C. Foket, and B. De Sutter, Effective extensible programming: Unleashing Julia on GPUs, IEEE Trans. Parallel. Distrib. Syst. **30**, 827 (2018).

[37] T. Besard, V. Churavy, A. Edelman, and B. De Sutter, Rapid software prototyping for heterogeneous and distributed platforms, Adv. Eng. Softw. **132**, 29 (2019).

[38] M. Innes, E. Saba, K. Fischer, D. Gandhi, M. C. Rudilosso, N. M. Joy, T. Karmali, A. Pal, and V. Shah, Fashionable modelling with flux, arXiv:1811.01457 (2018).

[39] M. Innes, Flux: Elegant machine learning with julia, J. Open Source Softw. **3**, 602 (2018).

[40] T. Breloff, Plots.jl, Zenodo 10.5281/zenodo.7074366 (2022).

[41] J. Revels, M. Lubin, and T. Papamarkou, Forward-mode automatic differentiation in Julia, arXiv:1607.0789 (2016).

Chapter 8

Minibatch Training of Neural Network Ensembles via Trajectory Sampling

The following work is from the *arXiv* pre-print [144] "*Minibatch training of neural network ensembles via trajectory sampling*" by **J. F. Mair**, L. Causer and J. P. Garrahan.

We build on the work presented in Chapter 7 by modifying the algorithm to allow for minibatch updating, while still respecting the overall properties of the ensemble training method. This brings the technique into line with current state-of-the-art methods which use minibatching for parameter updates with an aim to decouple training time from the size of the dataset. We illustrate that this approach is viable on the trajectory sampling training method previously explored, and validate our results on the MNIST classification problem [28].

arXiv:submit/4976721 [cond-mat.stat-mech] 27 Jun 2023

Minibatch training of neural network ensembles via trajectory sampling

Jamie F. Mair,[1, *] Luke Causer,[1, 2] and Juan P. Garrahan[1, 2]

[1] *School of Physics and Astronomy, University of Nottingham, Nottingham, NG7 2RD, UK*
[2] *Centre for the Mathematics and Theoretical Physics of Quantum Non-Equilibrium Systems, University of Nottingham, Nottingham, NG7 2RD, UK*

(Dated: June 27, 2023)

Most iterative neural network training methods use estimates of the loss function over small random subsets (or *minibatches*) of the data to update the parameters, which aid in decoupling the training time from the (often very large) size of the training datasets. Here, we show that a minibatch approach can also be used to train neural network ensembles (NNEs) via trajectory methods in a highly efficient manner. We illustrate this approach by training NNEs to classify images in the MNIST datasets. This method gives an improvement to the training times, allowing it to scale as the ratio of the size of the dataset to that of the average minibatch size which, in the case of MNIST, gives a computational improvement typically of two orders of magnitude. We highlight the advantage of using longer trajectories to represent NNEs, both for improved accuracy in inference and reduced update cost in terms of the samples needed in minibatch updates.

I. INTRODUCTION

Traditional machine learning (ML) applications aim to train a single model, usually by adjusting the parameters that define a complex function approximator like a neural network (NN), to perform well on some desired outcome as measured by a proxy loss function. A high performance on an appropriate loss function will entail a high performance on the metrics one cares about, for example accuracy in a classification problem [1]. There is strong empirical evidence from numerical experiments that increasing the scale of single models improves performance, as for example in the timely class of large language models (LLMs) [2].

However, to counteract the seemingly ever-increasing size of LLMs, there has also been significant work towards devising smaller models with similar capabilities in order to reduce the computational cost of training and of inference. A notable recent example is Stanford's Alpaca [3], based on LLaMA [4, 5], which can match GPT3.5 [6] despite being over an order of magnitude smaller. Another possibility is to replace one large model by an *ensemble* of smaller models which can provide similar or better inferences while also being less costly to train and evaluate [7, 8]. This is the class of problems we focus on here.

Recently, we introduced an approach to train collectively an ensemble of models [9], in particular neural network ensembles (NNEs) where predictions at inference time are aggregated in a committee-like fashion (for classification, the ensemble prediction is the most voted for option, while for scoring, the ensemble prediction is the mean ensemble score). In Ref. [9] we defined an NNE in terms of the trajectory of the model parameters under a simple (discrete in time, diffusive in parameter space) dynamics, and trained it by biasing the trajectory that defines the NNE towards a small time-integrated loss. That is, once training is converged, the NNE corresponds to a discrete trajectory of the model parameters sampled from a distribution of trajectories exponentially "tilted" to have low time-integrated loss. This approach is borrowed from the study of glassy systems [10], where biasing dynamics according to time-integrated observables (e.g. the dynamical activity [11, 12]) is known to access low energy states for the configurations in the trajectory. Such low-loss trajectories can be accessed via importance sampling in trajectory space, such as transition path sampling (TPS) [13] as adapted to stationary dynamics and large deviation problems [14]. The ensuing trained NNE is a collection of NN models correlated by the underlying dynamics of the parameters and with a low value of the total loss due to the tilting.

While Ref. [9] provides a proof of principle of the trajectory sampling approach, it suffers from a significant computational bottleneck: importance sampling is a Monte Carlo scheme on trajectories, where updates are determined according to changes in the (time-integrated) loss evaluated over the *whole training set*, so that each Monte Carlo iteration scales with the size of the training data. For example, when training for the textbook MNIST digit classification problem in Ref. [9], we used only a small amount of the entire training dataset (2048 samples from the available 60000) to make the problem tractable for a comprehensive study. On the contrary, it is well known that ML models generalise poorly with small datasets [1]. This computational limitation makes the method of Ref. [9] impractical for more complex tasks. This has to be contrasted with gradient descent [1], where there is no need to sample faithfully from a distribution, so that the gradient of the loss can be estimated efficiently only on very small subsets of training data, known as *minibatches*, giving rise to stochastic gradient descent (where the noise from the difference between the minibatch estimate and the full loss actually helps convergence to a good local minimum [1]).

In this paper, we resolve the problem above by implementing a minibatch method in the trajectory sampling

* Jamie.Mair@nottingham.ac.uk

used in the training of the NNEs. We build on the approach of Ref. [15] for doing Monte Carlo sampling with small data batches. We show that our new method reduces the training cost by a factor given by the ratio of the average minibatch size (which we determine in an adaptive manner) to the size of the dataset. We illustrate this more efficient method on MNIST classification (using the whole MNIST dataset), showing in this case a computational gain of about two orders of magnitude. Our minibatch approach also allows us to highlight the key features of the trajectory NNE method, showing the advantage of using longer trajectories to represent NNEs both in terms of accuracy and data requirement for training.

The rest of the paper is organised as follows. In Sec. II we describe the theory, reviewing the idea of NNEs as trajectories of a stochastic dynamics, training as tilted trajectory sampling, and the central approach to perform mini-batch trajectory Monte Carlo. In Sec. III we present the adaptive minibatch trajectory sampling method for training NNEs. We illustrate the method with two examples in Sec. IV, an exactly solvable linear perceptron, and the full MNIST digit classification problem. In Sec. V we give our conclusions, and further technical details are provided in the Appendices.

II. THEORETICAL BACKGROUND

A. Neural network ensemble as a trajectory of neural networks

In Ref. [9], we proposed that a NNE could be obtained by evolving the parameters of a NN model under a suitable stochastic dynamics, where the NNE is composed of the sequence of NNs in time. If, at time step t, the NN is defined by $\boldsymbol{\theta}_t$, this dynamics would give rise to a trajectory $\boldsymbol{\theta}_1 \to \boldsymbol{\theta}_2 \to \ldots \to \boldsymbol{\theta}_\tau$, with the NNE as the set of visited models under the dynamics, $\boldsymbol{\Theta} = [\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots, \boldsymbol{\theta}_\tau]$. As the aim is to minimise the loss over the ensemble

$$\mathcal{L}(\boldsymbol{\Theta}) = \sum_{t=1}^{\tau} L(\boldsymbol{\theta}_t) \tag{1}$$

where $L(\boldsymbol{\theta}_t)$ is the standard loss for the t-th model (see below for a specific form of the loss), training is equivalent to finding a suitable dynamics whose typical trajectories are those with low time-aggregated loss, $\mathcal{L}(\boldsymbol{\Theta})$.

Once trained, this dynamics is defined in terms of (in general time-dependent) stochastic dynamics $\mathcal{M}(\tau, \sigma, s) \equiv \{M_{t;\sigma,s}\}_{t=1}^{\tau-1}$, where $M_{t;\sigma,s}(\boldsymbol{\theta}'|\boldsymbol{\theta})$ are the transition probabilities at each time step, such that the NNE corresponds to a trajectory generated using dynamics $\mathcal{M}(\tau, \sigma, s)$. This approach is illustrated in Fig. 1(a): the NNE is a discrete-time trajectory, where each state along the trajectory corresponds to one of the NNs that form the ensemble. Starting from the first model, $\boldsymbol{\theta}_1$, each subsequent model is sampled according

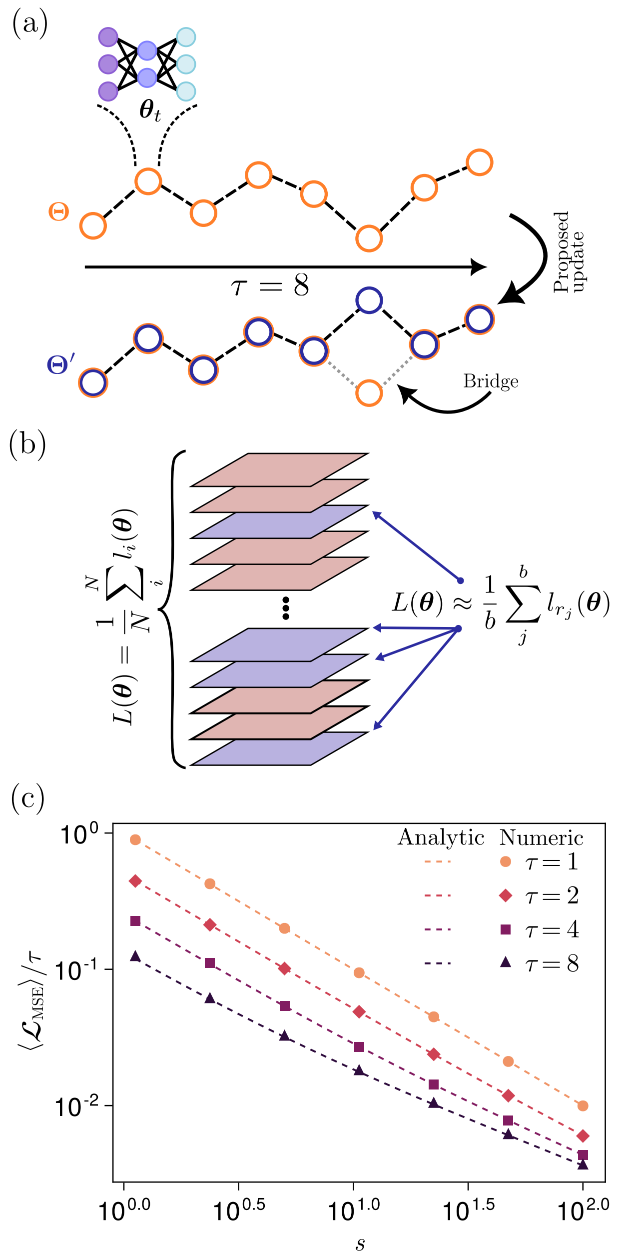

FIG. 1. (a) An NNE as a stochastic trajectory, and a sketch of trajectory sampling. Each state in the trajectory corresponds to one NN model in the NNE. The proposed path sampling update from trajectory $\boldsymbol{\Theta}$ to trajectory $\boldsymbol{\Theta}'$ is via a stochastic bridge in which only one model is modified. (b) The loss function of a model with parameters $\boldsymbol{\theta}$. Each layer represents a data element. The loss, $L(\boldsymbol{\theta})$, is given by the average of the individual loss for each of the elements of the dataset, $\{l_i(\theta)\}_{1:N}$. The minibatch estimate of the loss is instead the average over a random selection $\{r_j\}_{1:b}$ of b data points in the dataset. (c) Exactly solvable linear perceptron: mean NNE loss $\langle \mathcal{L} \rangle$ per NNE size τ, as a function of s for various τ. Lines are analytical results. Symbols are the numerical results using the minibatch TPS algorithm for training (with 20×10^6 TPS epochs as a "burn-in" followed by 20×10^6 epochs for convergence of training).

to $\mathcal{M}(\tau, \sigma, s)$. Three hyperparameters determine the dynamics that produces the NNE: the final time τ sets the number of models in the ensemble; σ sets the "stiffness" of the chain (see below for details), that is, how correlated subsequent models are to each other, with small σ corresponding to large stiffness; and s controls the level of the overall loss, with larger s corresponding to lower loss. The central idea is that a correlated chain of models generated as a trajectory from dynamics $\mathcal{M}(\tau, \sigma, s)$ at large s provides a well trained NNE [9].

B. Learning as a trajectory sampling problem

Obtaining a suitable dynamics $\mathcal{M}(\tau, \sigma, s)$ that produces well-trained NNEs as its typical trajectories is a difficult task. We can however resolve this problem by means of trajectory sampling techniques [9]. Consider as a starting point an untrained dynamics $\mathcal{M}(\tau, \sigma, 0)$ with the same transition probabilities at every time step t [9]

$$M_\sigma(\boldsymbol{\theta}_{t+1}|\boldsymbol{\theta}_t) \propto \exp\left[-\frac{1}{2\sigma^2}(\boldsymbol{\theta}_t - \boldsymbol{\theta}_{t+1})^2\right], \qquad (2)$$

with $\int_{\boldsymbol{\theta}'} M_\sigma(\boldsymbol{\theta}'|\boldsymbol{\theta}) = 1$. This dynamics corresponds to a discrete-time Gaussian diffusion process that knows nothing about the loss (1). As such, a typical trajectory drawn from it will correspond to a random (and therefore untrained) NNE. As indicated above, the parameter σ sets the variance of the diffusive steps, so that for smaller σ subsequent models are more correlated, while for $\sigma \to \infty$ all the models of the chain are uncoupled.

The dynamics (2) produces an *ensemble of trajectories* (and therefore an ensemble of NNEs) with each trajectory having probability

$$P(\boldsymbol{\Theta}; \sigma) = \frac{1}{\mathcal{Z}_\tau(\sigma)} p(\theta_1) \prod_{t=1}^{\tau-1} \exp\left[-\frac{1}{2\sigma^2}(\boldsymbol{\theta}_t - \boldsymbol{\theta}_{t+1})^2\right], \qquad (3)$$

given by the product of the M_σ at each step. Here $p(\theta_1)$ is the probability used to draw the first model, and $\mathcal{Z}_\tau(\sigma)$ a normalisation constant (the "partition sum" of the trajectory ensemble). In order to obtain trajectories with low overall loss, what we aim is to define a new trajectory ensemble that is exponentially "tilted" with respect to (3), as is standard in large deviation studies of dynamics (e.g., Ref. [16]), that is [9]

$$P(\boldsymbol{\Theta}; \sigma, s) = \frac{1}{\mathcal{Z}_\tau(\sigma, s)} e^{-s\mathcal{L}(\boldsymbol{\Theta})} P(\boldsymbol{\Theta}; \sigma). \qquad (4)$$

For large s, a typical trajectory from this ensemble will correspond to a NNE with low overall loss. The learned dynamics $\mathcal{M}(\tau, \sigma, s)$ of the previous subsection would be the dynamics that produces trajectories distributed according to the tilted distribution (4).

One way to avoid having to determine the $\mathcal{M}(\tau, \sigma, s)$ dynamics explicitly is to directly sample trajectories from the tilted distribution (4). In this way, convergence of the training, that is, finding $\mathcal{M}(\tau, \sigma, s)$, coincides with convergence of the trajectory sampling of (4), as we do here by means of an importance sampling method in trajectory space based on transition path sampling (TPS) [13].

C. Monte Carlo in trajectory space

Consider a Monte Carlo scheme for sampling trajectories, specifically a Metropolis-Hastings approach [13]: given a current trajectory $\boldsymbol{\Theta}$, the probability to change to a new trajectory $\boldsymbol{\Theta}'$ is given by

$$p(\boldsymbol{\Theta}'|\boldsymbol{\Theta}) = g(\boldsymbol{\Theta}'|\boldsymbol{\Theta}) A(\boldsymbol{\Theta}', \boldsymbol{\Theta}), \qquad (5)$$

where the factor $g(\boldsymbol{\Theta}'|\boldsymbol{\Theta})$ is the probability to propose the move, and $A(\boldsymbol{\Theta}', \boldsymbol{\Theta})$ is that to accept it. For the above to converge to (4) we need to impose that it obeys detailed balance with respect to (4), which implies

$$\frac{A(\boldsymbol{\Theta}', \boldsymbol{\Theta})}{A(\boldsymbol{\Theta}, \boldsymbol{\Theta}')} = \frac{P(\boldsymbol{\Theta}'; \sigma) g(\boldsymbol{\Theta}|\boldsymbol{\Theta}')}{P(\boldsymbol{\Theta}; \sigma) g(\boldsymbol{\Theta}'|\boldsymbol{\Theta})} \qquad (6)$$

If the proposed moves obey detailed balance with respect to the original untilted dynamics (3),

$$\frac{g(\boldsymbol{\Theta}'|\boldsymbol{\Theta})}{g(\boldsymbol{\Theta}|\boldsymbol{\Theta}')} = \frac{P(\boldsymbol{\Theta}'; \sigma)}{P(\boldsymbol{\Theta}; \sigma)}, \qquad (7)$$

then the acceptance ratio reduces to

$$\frac{A(\boldsymbol{\Theta}', \boldsymbol{\Theta})}{A(\boldsymbol{\Theta}, \boldsymbol{\Theta}')} = e^{-s\left[\mathcal{L}(\boldsymbol{\Theta}') - \mathcal{L}(\boldsymbol{\Theta})\right]} \qquad (8)$$

In standard TPS, (7) is realised by proposing trajectories by simply running the original dynamics (3) (via "shooting" or "shifting" moves, see Ref. [13]). This approach, however, carries an exponential cost in the time extent of trajectories, since the loss difference in the exponent of (8) scales linearly with time. This can be mitigated [9] by proposing small changes in a trajectory, see Fig. 1(a): the proposed trajectory is one where only the state at one time is modified; as this has to obey (7), it has to be done as a *Brownian bridge* [17, 18]. That is, as conditioned dynamics starting in the previous state and retuning to the state after the one changed (see Ref. [9] for details).

III. MINIBATCH PATH SAMPLING

While the Brownian bridge version of TPS ameliorates the exponential-in-time cost in the trajectory sampling, there is another source of computational slowness coming from the evaluation of the trajectory loss, cf. (8). With the Brownian bridge TPS, Fig. 1(a), only a single model changes between the current and proposed trajectory, and the change in trajectory loss in (8) is therefore

given by that the change of that model's loss. If this change is at time t, this requires the evaluation of $L(\boldsymbol{\theta}'_t)$, which at training is the average of the loss under that model for each of the N training data points,

$$L(\boldsymbol{\theta}'_t) = \frac{1}{N}\sum_{i=1}^{N} l_i(\boldsymbol{\theta}'_t) \tag{9}$$

where $l_i(\boldsymbol{\theta}_t)$ is the loss for the inference for data point i. This means that in each Monte Carlo iteration, computing the change in loss inevitably scales with the training set size N (together with a cost that depends on the size and architecture of the NN being considered). This evaluation can therefore become computationally infeasible for larger datasets. For example, in Ref. [9], we had to reduce the training dataset by almost an order of magnitude to show a proof-of-principle of the method with for the MNIST classification problem.

In contrast to gradient descent, one cannot simply replace the loss over the whole training set for an estimate based on a small subset, or minibatch. For gradient descent, the error that this introduces becomes a source of noise, converting it into stochastic gradient descent (and its adaptive variants [1, 19, 20]). This in turn gives rise to the usual advantages that an exploit/explore strategy brings, in this case to minimise the loss locally descending the gradient versus exploration of the loss landscape. Since Monte Carlo aims to sample from a distribution, Eq. (4) in our case, a straightforward replacement of the loss by a minibatch approximation would lead to failure of the necessary detailed balance condition.

This problem has been considered before in the context of Bayesian inference, where so-called "tall datasets" make Monte Carlo inefficient, see e.g., Refs. [21–23]. In what follows we build on the approach put forward in Ref. [15] to develop an adaptive minibatch trajectory sampling method.

A. Monte Carlo with minibatches

We first describe the scheme of Ref. [15] in the context of the Monte Carlo annealing of a system with degrees of freedom $\boldsymbol{\Theta}$ (a NNE in our case) and target distribution (4), and in the next subsection we extend the approach to integrate it with TPS in an adaptive manner.

Let us define the quantity $\Delta(\boldsymbol{\theta}', \boldsymbol{\theta})$ as the logarithm of the change in weight under a proposed move,

$$\Delta(\boldsymbol{\Theta}', \boldsymbol{\Theta}) = -s\left[\mathcal{L}(\boldsymbol{\Theta}') - \mathcal{L}(\boldsymbol{\Theta})\right], \tag{10}$$

and choose our acceptance function as

$$A(\boldsymbol{\Theta}', \boldsymbol{\Theta}) = (1 + e^{\Delta(\boldsymbol{\Theta}', \boldsymbol{\Theta})})^{-1}, \tag{11}$$

which satisfies the detailed balance condition (8). Monte Carlo works by generating a proposed move from $g(\boldsymbol{\Theta}'|\boldsymbol{\Theta})$, and then accepting the move if

$$A(\boldsymbol{\Theta}', \boldsymbol{\Theta}) > V, \tag{12}$$

where V is a uniformly distributed random number, $V \sim \mathcal{U}(0, 1)$. As (11) is a logistic (or sigmoid) function whose inverse is also its derivative, we can equivalently write the acceptance test as $\Delta(\boldsymbol{\Theta}', \boldsymbol{\Theta}) > X_{\mathrm{log}}$, where X_{log} is a logistically sampled random variable. As this distribution is symmetric around zero, we can equally write the test (12) as

$$\Delta(\boldsymbol{\Theta}', \boldsymbol{\Theta}) + X_{\mathrm{log}} > 0. \tag{13}$$

The loss $\mathcal{L}(\boldsymbol{\Theta})$ that enters in (10) is the average of the loss over the entire training data set

$$\mathcal{L}(\boldsymbol{\Theta}) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}_i(\boldsymbol{\Theta}) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{\tau} l_i(\boldsymbol{\theta}_t), \tag{14}$$

where $\mathcal{L}_i(\boldsymbol{\Theta})$ is the (trajectory) loss for the inference on data point i. Consider now an approximation of the loss difference in terms of a random minibatch of size b

$$\mathcal{L}(\boldsymbol{\Theta}) \approx \frac{1}{b}\sum_{j=1}^{b}\mathcal{L}_{r(j)}(\boldsymbol{\Theta}), \tag{15}$$

where $r(j)$ specifies a random permutation of the indices of the elements in the training dataset, cf. Fig. 1(b). In terms of the above we can define an approximation to (10) [15],

$$\Delta^*(\boldsymbol{\Theta}, \boldsymbol{\Theta}') = -\frac{s}{b}\sum_{j=1}^{b}\left[\mathcal{L}_{r(j)}(\boldsymbol{\Theta}') - \mathcal{L}_{r(j)}(\boldsymbol{\Theta})\right]. \tag{16}$$

If one could replace $\Delta(\boldsymbol{\Theta}, \boldsymbol{\Theta}')$ by $\Delta^*(\boldsymbol{\Theta}, \boldsymbol{\Theta}')$, then there would be a computational gain that would scale as the ratio of the size N of the training dataset to that of the minibatch b (as it is necessary to compute $\Delta(\boldsymbol{\Theta}, \boldsymbol{\Theta}')$ for each Monte Carlo iteration). The way to do so is as follows [15].

Since the elements of the minibatch are chosen in an identical and independent manner, from the central limit theorem and for large enough b, we expect Δ^* to be normally distributed around Δ with some variance $\rho^2(\Delta^*)$. That is, $\Delta^* = \Delta + X_{\mathrm{norm}}$, where X_{norm} is an approximately normal zero-mean random correction of variance $\rho^2(\Delta^*)$. As a logistically distributed random variable, X_{log}, is almost normally distributed, we can write X_{log} as $X_{\mathrm{log}} = X_{\mathrm{norm}} + X_{\mathrm{corr}}$, where X_{corr} is the (hopefully small) correction to normality. Inserting this decomposition of X_{log} into the acceptance test (13), we can replace Δ with the minibatch estimate:

$$\Delta^*(\boldsymbol{\Theta}', \boldsymbol{\Theta}) + X_{\mathrm{corr}} > 0. \tag{17}$$

This new acceptance only depends on the minibatch estimates of the loss and — if accurate — will be efficient for $b \ll N$. The test (17) will asymptotically give the correct acceptance distribution provided that (i) that the fluctuations X_{norm} of Δ^* around Δ are normally distributed (which can be checked to adjust the size of b), and (ii)

that the distribution $C_{\text{corr}}(X;\rho)$ for the random correction X_{corr} can be numerically computed with low error (see [15] for analysis of the errors).

Thanks to the CLT, condition (i) is relatively easy to satisfy for a large enough minibatch sample size. If the batch size grows beyond the size of the dataset, we do not need this approximation and can use (12). Condition (ii) holds when the sample error on Δ^* is sufficiently small [15]. In practice one can only compute the distribution of $C_{\text{corr}}(X;\rho)$ accurately enough only for standard deviations of X_{norm} such that $\rho \lesssim 1.1$ (see below for our implementation).

As computing $C_{\text{corr}}(X;\rho)$ is numerically expensive (see Appendix A for details) for each empirical $\rho \approx 1$, we can instead compute a $\rho^2 = 1$ correction and then add a further "normal correction" with small variance $1-\rho^2$ to make a total normal random variable with fixed variance 1. Putting all of these together, we get the minibatch acceptance test that we use

$$\Delta^*(\boldsymbol{\Theta}, \boldsymbol{\Theta}') + X_{\text{nc}} + X_{\text{corr}} > 0, \tag{18}$$

where X_{nc} stands for the normal correction random variable, of zero mean and variance $1-\rho^2$.

Generalisation to enable training with TPS

The TPS scheme that we use relies on proposing trajectory updates, cf. Fig. 1(a), consisting of *bridging* moves (for changes in the middle of the trajectory) and *shooting* moves to get changes in the endpoints, see Ref. [9] for details. In either case, only a single model (i.e. a single time step) is altered, thus reducing the size of the update and improving the acceptance rate.

Once a candidate trajectory is proposed, the minibatch acceptance criterion of the previous subsection is applied. The specific steps are as follows, defining an adaptive minibatch scheme:

- We draw m random samples from the training set, which are used to calculate an estimate of $\Delta^*(\boldsymbol{\Theta}, \boldsymbol{\Theta}')$ and $\rho^2(\Delta^*)$, cf. (16). If $\rho^2 > 1$, m more samples are drawn without replacement, updating Δ^* and ρ accordingly (and terminating if all samples are used). In this way, we form a minibatch of overall size b such that the sample variance of Δ^* is strictly less than or equal to 1.

- We draw the random correction X_{nc} from a normal distribution and X_{corr} from $C_{\text{corr}}(X;\rho)$. With these we use (18) to accept or reject the proposed change to the trajectory. [If the total minibatch size b equals N, then we use the original test (12) as Δ^* coincides with Δ in that case.] Unlike Ref. [15], exact sampling of (4) is not required to effectively train our NNE, and we do not test the normality assumption of $\Delta^*(\boldsymbol{\Theta}, \boldsymbol{\Theta}')$. This is equivalent to setting their threshold $\delta \to \infty$ in Ref. [15]. Justification for this simplifying choice is given in Appendix C.

Algorithm 1 Minibatch TPS Training

1: **input** Initial trajectory $\boldsymbol{\Theta}_1$, dataset $\{x_1, \ldots, x_N\}$, trajectory length τ, trajectory coupling σ, minibatch chunk size m, pre-computed correction $C_{\rho=1}(X)$ distribution, cut-off hyperparameters c_0 and c_1 and training epochs E.
2: **output** Sequence of trajectories $\{\boldsymbol{\Theta}_1, \boldsymbol{\Theta}_1, \ldots, \boldsymbol{\Theta}_{E+1}\}$
3: **for** $k \in [1, 2, \ldots, E]$ **do**
4:     Sample proposal $\boldsymbol{\Theta}'$ using $g(\boldsymbol{\Theta}'|\boldsymbol{\Theta}_k, \tau, \sigma)$ (i.e. shooting or bridging)
5:     Sample $\Delta^*(\boldsymbol{\Theta}', \boldsymbol{\Theta}_k)$ and ρ^2 using m randomly selected samples, without replacement
6:     $b \leftarrow N$
7:     **while** $\rho^2 > 1$ **and** $b < N$, **and** $\max(\frac{|\Delta^*(\boldsymbol{\Theta}, \boldsymbol{\Theta}')|}{\rho} - c_1, 0) <= c_0$ **do**
8:         Select m more randomly selected samples, without replacement and update estimates for $\Delta^*(\boldsymbol{\Theta}', \boldsymbol{\Theta}_k)$ and ρ^2
9:         $b \leftarrow b + m$
10:     **end while**
11:     **if** $b = N$ **then**
12:         Sample random number $V \sim \mathcal{U}(0,1)$
13:         **if** $V < g(\Delta^*(\boldsymbol{\Theta}', \boldsymbol{\Theta}_k))$ **then**
14:             Accept with $\boldsymbol{\Theta}_{k+1} \leftarrow \boldsymbol{\Theta}'$
15:         **else**
16:             Reject with $\boldsymbol{\Theta}_{k+1} \leftarrow \boldsymbol{\Theta}_k$
17:         **end if**
18:         `continue`
19:     **end if**
20:     Sample $X_{\text{nc}} \sim \mathcal{N}(0, 1-\rho^2)$ and $X_{\text{corr}} \sim C_{\rho=1}(X)$
21:     **if** $\Delta^*(\boldsymbol{\Theta}', \boldsymbol{\Theta}_k) + X_{\text{nc}} + X_{\text{corr}} > 0$ **then**
22:         Accept with $\boldsymbol{\Theta}_{k+1} \leftarrow \boldsymbol{\Theta}'$
23:     **else**
24:         Reject with $\boldsymbol{\Theta}_{k+1} \leftarrow \boldsymbol{\Theta}_k$
25:     **end if**
26: **end for**

One issue with this simple algorithm is that the minibatch size grows if the sample variance is much larger than 1. The biasing parameter, s, scales the sample variance with s^2, while taking b samples only reduces this variance by a factor of b. For fixed $\boldsymbol{\Theta}$ and $\boldsymbol{\Theta}'$, we would expect the minibatch size to change with s to compensate for the increased sample variance. For $s \to \infty$, this would revert the minibatch method to the original full-dataset acceptance test. To avoid this, we introduce a *cut-off test* which halts increasing the minibatch size when Δ^* is sufficiently far away from the origin and performs an alternative acceptance test. The alternative acceptance test is broken into two stages. Firstly, we approximate the acceptance function to be equal to 0 when $x < -c_0$ and 1 when $x > c_0$ for some positive constant c_0. We choose c_0 to be sufficiently high that the approximate acceptance function is unchanged for $-c_0 \leq x \leq c_0$ without having to renormalise. Secondly, we choose another threshold for which the true mean Δ is approximately guaranteed to be within $\Delta^* \pm c_1\rho$. We use $\max(\rho^{-1}|\Delta^*(\boldsymbol{\Theta}, \boldsymbol{\Theta}')| - c_1, 0) > c_0$ as our cut-off acceptance test. In our experiments, using

$c_1 = 10$ and $c_0 = 5$ gave good results. The final combined algorithm is given in Algorithm 1. Setting c_0 or c_1 to be high will make this cut-off less likely to be triggered, increasing minibatch size and therefore computational cost, however, setting them too close to 0 will result in an inaccurate acceptance test which does not obey detailed balance.

IV. EXAMPLES OF TRAINING NN ENSEMBLES VIA MINIBATCH TRAJECTORY SAMPLING

We now apply the method of Sec. III for the training of NNEs in two illustrative problems. The first is that of a linear perceptron, which is simple enough to be solved exactly, allowing us to directly compare our method with the expected results. The second is the more complex, but now standard, problem of MNIST digit classification [24].

A. NNE of linear perceptrons

We first test the method with a linear classification problem, also considered in Ref. [9]. This problem can be defined as follows: we generate a set of independent random D dimensional points, $\{\boldsymbol{x}_i\}_{i=1}^N$ (setting $x_N = 1$), together with a D-dimensional random weight vector $\boldsymbol{w}$ that we use to assign labels $y_i = \boldsymbol{w} \cdot \boldsymbol{x}_i$ to each of the points $\boldsymbol{x}_i$. The aim is to train the parameters $\boldsymbol{\Theta}$ of an ensemble of τ linear perceptrons, where the prediction of the t-th perceptron for the i-th data point is $\boldsymbol{\theta}_t \cdot \boldsymbol{x}_i$. For training, we consider the mean-squared sample loss for each model in the NNE, which for data point i reads

$$l_i^{(\mathrm{MSE})}(\boldsymbol{\theta}_t) = \frac{1}{2}(y_i - \boldsymbol{\theta}_t \cdot \boldsymbol{x}_i)^2. \tag{19}$$

The NNE loss over the training dataset, cf. (14), in turn reads

$$\mathcal{L}(\boldsymbol{\Theta}) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{\tau}\frac{1}{2}(y_i - \boldsymbol{\theta}_t \cdot \boldsymbol{x}_i)^2 \tag{20}$$

We now implement the adaptive minibatch estimation of the trajectory loss described in Alg. 1 to train this NNE for various τ and s. Figure 1(c) demonstrates that the numerics obtained in this way coincide with the analytic results from the exact trajectory distribution (4) [9]. This is an elementary proof-of-principle of the method.

Exact distributions for $\boldsymbol{w}$ and $\boldsymbol{x}$, along with experimental hyperparameters are provided in Appendix B 1.

B. NNE for MNIST digit classification

The second problem we consider is that of an ensemble of models for classification of digits using the standard set of handwritten MNIST images [24], see Fig. 2(a). In this case, each NN in the NNE is a small convolutional neural network (CNN) whose architecture is described in Appendix B 2. For a given image X, one of these CNNs with parameters $\boldsymbol{\theta}$ provides the probability $y(k|X;\boldsymbol{\theta})$ that the image corresponds to digit k, for $k = 0, \ldots, 9$. The appropriate loss function is the mean cross entropy, which for the data point i and the t-th model reads

$$l_i^{(\mathrm{MNIST})}(\boldsymbol{\theta}_t) = -\sum_{k=0}^{9}\delta_{z_i,k}\log y(k|X_i;\boldsymbol{\theta}_t), \tag{21}$$

where z_i is the true classification of X_i. The training loss for the NNE then reads

$$\mathcal{L}(\boldsymbol{\Theta}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{\tau}\sum_{k=0}^{9}\delta_{z_i,k}\log y(k|X_i;\boldsymbol{\theta}_t), \tag{22}$$

We train trajectories for a fixed number of epochs E (i.e., TPS iterations), which we choose to be large enough for the trajectory loss to appear to converge, which we track in terms of the minibatch loss estimate to avoid further computational cost. One can observe convergence for a small number of sample runs in Figs. 3(a, b). Specifically, we allow for a "burn-in" of the initial 20×10^6 epochs, and subsequently observe the average trajectory loss for the next 20×10^6 epochs. For each set of hyperparameters (τ, s) we perform six independent trainings starting each training run from a random initial seed trajectory. The time-averaged loss thus obtained is shown in Fig. 2(b) as a function of s for various NNE sizes τ. We note the following: (i) for every τ the loss per model in the trained NNE decreases with s, as should be the case when converging to (4); (ii) the larger τ the lower the loss, indicating that the longer trajectories give rise to more accurate NNEs; (iii) there appears to be a transition from high to low loss with s which could indicate (dynamical) phase coexistence, as seen in many other trajectory ensemble problems [25].

A similar trend to that of the loss is observed in the accuracy on the generalisation test set. In Fig. 2(c) we show the accuracy of the final ensembles obtained after all training epochs. We use the NNE to collectively make a prediction on each sample by letting each model "vote" for their predicted class, with the class with the most votes getting selected (in the event of a tie, the smaller digit is selected). We plot this ensemble accuracy, averaged over six independent runs, for the same hyperparameters of panel (b).

As a proxy for the computational cost of training, in Fig. 2(d) we show the average batch size per epoch $\langle b \rangle$ necessary for training to a certain value of the loss per model. Since the size of the training set is $N = 6 \times 10^4$, the ratio $N/\langle b \rangle$ gives the computational gain of using the minibatch method. From Figs. 2(b, c), we know that longer trajectories can yield a lower loss at smaller values of s. From Fig. 2(d), we see that longer trajectories (larger NNEs) are also computationally more efficient to

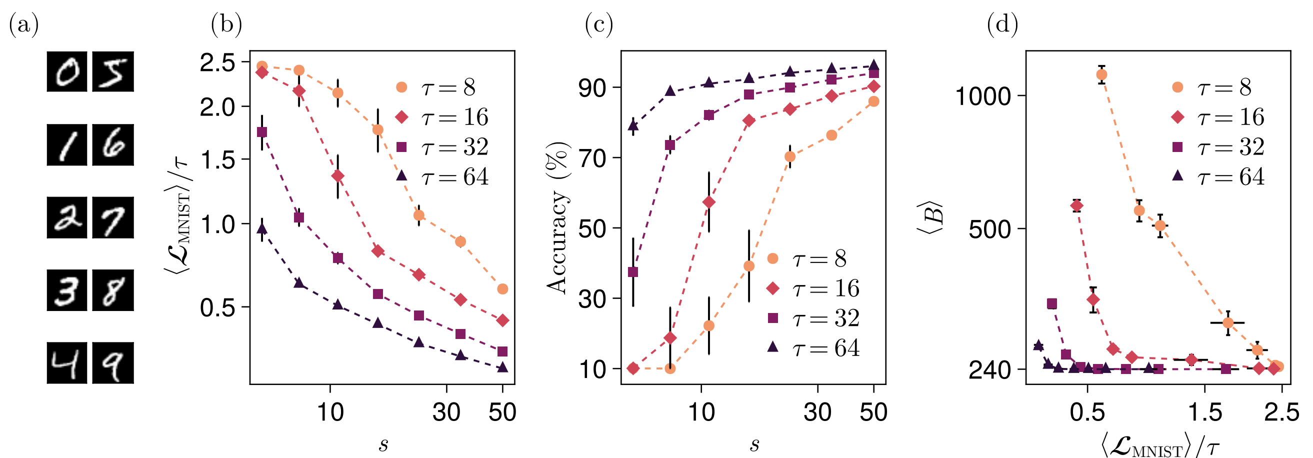

FIG. 2. (a) Representative examples of the MNIST digit images used for training. (b) Mean value of the time-averaged loss vs s. Each data point is calculated after 2×10^7 TPS epochs over the following 2×10^7 TPS epochs. (c) Final accuracy on the standard 10,000 test images of a single trained NNE (via majority vote), taken at the end of the 4×10^7 epochs. (d) Average batch size per epoch, $\langle b \rangle$ as a function of the converged mean NNE loss after 2×10^7 TPS epochs. The mean batch size per epoch is obtained from the 4×10^7 TPS training epochs.
All results and error bars are from averaging over 6 independent runs with different initial trajectories.

train, requiring a smaller mean minibatch size, $\langle b \rangle$, than smaller NNEs for the same level of overall loss, showing a computational gain in excess of two orders of magnitude of the minibatch method to the original full loss method of Ref. [9].

V. CONCLUSIONS

In this paper we have presented a variant of the minibatch Monte Carlo method of Ref. [15] adapted to the sampling of trajectories that correspond to neural network ensembles [9]. We have shown that this technique can be used to train NNEs via trajectory sampling to give an improvement in computational efficiency up to two orders of magnitude. While we have focused, for concreteness, on supervised learning applications, we note that an adaptive trajectory sampling technique like the one presented here should be also very useful in Monte Carlo based *reinforcement learning* (RL), where datasets do not have a fixed size. We expect that this method will provide a stable training technique on these RL problems, which have exhibited brittle behaviour when continuously trained on changing objectives [26–29].

Our results here add to the growing number of recent works studying the training dynamics of NNs from the statistical mechanics point of view, see e.g., Refs. [30–34]. Most of these consider the training of a single NN in terms of a stochastic dynamics akin to thermal annealing, cf. Ref. [32]. In contrast, our approach based on sampling trajectories of NNs shares more similarities to training by quantum annealing, see for example Refs. [35, 36]. Note that this similarity is not referring to actual unitary dynamics, but to the fact the computation of a trajectory ensemble in (4) is similar to that of a quantum partition sum (in terms of imaginary-time trajectories). Furthermore, the improved computational efficiency provided by the minibatch method we introduced here allowed us to highlight the benefit of larger NNEs (i.e., longer trajectories) capable of accessing lower loss regions of state space using far less data than single NNs or small ensembles.

CODE AVAILABILITY

Our TPS implementation package is available through GitHub, `TransitionPathSampling.jl` [37], together with the source code to generate the figures and results in the paper [38].

ACKNOWLEDGMENTS

We acknowledge support from EPSRC Grant no. EP/V031201/1 and University of Nottingham grant no. FiF1/3. LC was supported by an EPSRC Doctoral prize from the University of Nottingham. Simulations were performed using the University of Nottingham Augusta HPC cluster, and the Sulis Tier 2 HPC platform hosted by the Scientific Computing Research Technology Platform at the University of Warwick. Sulis is funded by EPSRC Grant EP/T022108/1 and the HPC Midlands+ consortium. We thank the creators and community of the Julia programming language [39], and acknowledge use of the packages `CUDA.jl` [40, 41], `Makie.jl` [42] and `ForwardDiff.jl` [43].

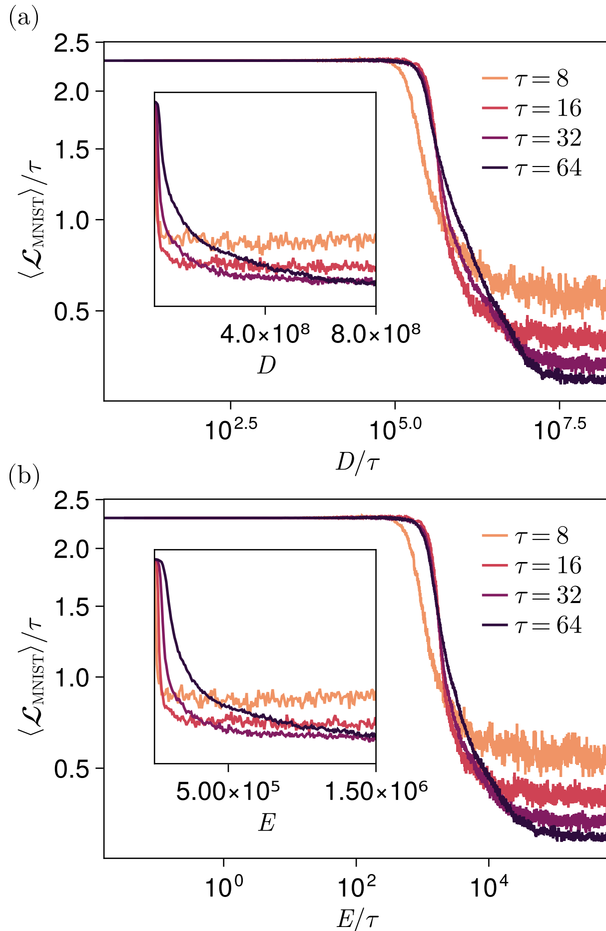

FIG. 3. Training curves for MNIST classification NNE. (a) Average ensemble loss per model in the NNE as a function of cumulative data usage D, for $s = 50$ and various NNE sizes τ. Note that the abscissa is scaled by τ, so curves are in terms of "per-model" epochs. In terms of D/τ, for equivalent training time lower losses are reached for larger NNEs. (Loss curves has been down sampled for clarity.) Inset: same in linear D scale. (b) Same training curves but now plotted in terms of epochs E.

Appendix A: Correction Distribution

Seita et al. [15] show that we can calculate the X_{corr} distribution numerically. We introduce a parameter V to specify the range of values to sample the distribution over. We construct two discrete vectors X and Y with the elements of X going linearly from $-2V$ to $+2V$ and Y going from $-V$ to $+V$. The vector X has $4N + 1$ elements and the vector Y has $2N + 1$ elements.

From here, we define a matrix M with elements

$$M_{ij} = \Phi_\sigma(X_i - Y_j), \tag{A1}$$

where Φ_σ is the cumulative distribution function (CDF) of a normal distribution with variance σ^2. Additionally, we construct a new vector v such that

$$v_i = S(X_i), \tag{A2}$$

where S is the logistic sigmoid function, i.e. the CDF of a logistically distributed random variable. Finally, we define the vector u to be $u_j = C_\sigma(Y_j)$ which is our target to calculate. This can be calculated using the formula

$$u = (M^T M + \lambda I)^{-1} M^T v, \tag{A3}$$

where λ is regularisation parameter. We followed recommendations from Seita et al [15] and used $V = 10$, $N = 4000$ and $\lambda = 10$ to construct our numerical approximation of C_σ. We set any negative elements equal to zero and re-normalise the CDF to ensure the area under the curve will equal 1.

Fortunately, the sampling algorithm allows us to calculate the distribution for a single value of σ to save on computation and memory. This distribution can be calculated once and cached for future use. We can alter the acceptance condition to be

$$\Delta^*(\mathbf{\Theta}, \mathbf{\Theta}') + X_{\text{nc}} + X_{\text{corr}} > 0, \tag{A4}$$

where X_{corr} is sampled when $\sigma = 1$ and $X_{\text{nc}} \sim \mathcal{N}(0, 1 - \text{Var}[\Delta^*])$, requiring that $\text{Var}[\Delta^*] < 1$.

Sampling the correction distribution

The distribution can be efficiently sampled using the CDF: the cumulative sum of the probability distribution function (PDF). The CDF is a monotonically increasing set of y values from 0 to 1. These values have corresponding X values in the domain $-2V$ to $2V$. In order to sample this distribution we draw a random number, u, uniformly between 0 and 1. We find the X which correponds to the intersection of $u = C_\sigma(X)$ by bisection on the discretised points and then linear interpolation between discretised values.

Appendix B: Experiment Configurations

Here, we provide the exact parameters used to generate data provided in the results.

1. Linear Perceptron

The linear perceptron model was trained on a simple 1D problem, which was generated via $\boldsymbol{y} = m\boldsymbol{x} + c$, where $x_i \sim \mathcal{U}(0, 1)$ and $m \sim \mathcal{U}(-1, 1)$ and $c \sim \mathcal{U}(-2, 0)$. We randomly sampled 256 points to use in the distribution. The minibatch method used batch sizes of 32 and set $\sigma = 0.1$ for the coupling between models in the trajectories. Experiments were run for 4×10^7 epochs. Averages of observables were taken by discarding the first half of the data, to allow for a *burn-in* time, and using minibatch estimates on the data.

2. MNIST

The convolutional neural network (CNN) model architecture used for our MNIST experiments was as follows:

1. Input 28×28 single channel image.
2. Convolution layer with a 5×5 kernel and 16 ouput channels.
3. 2×2 max pooling layer.
4. Convolution layer with a 3×3 kernel and 8 ouput channels.
5. 4×4 max pooling layer.
6. Fully connected dense layer with 10 outputs.
7. Softmax layer to normalise probabilities.

This model would output a normalised probability vector for each input image, specifying the "liklihood" of the image being a certain digit. The model contained 1906, 32-bit, floating point parameters.

For our experiments, we did not anneal the s parameter, but instead, chose a fixed duration of 2×10^7 epochs to allow for some "burn-in" time. The models were then run for another 2×10^7 epochs, to allow for measuring the loss as an observable. Accuracies were only measured at the end of the 4×10^7 epochs, due to the high computational demand.

We ran all of our experiments using $\sigma = 0.05$ and only changed a random 25% of the parameters of each model on each perturbation. Each perturbation changed only a single model in the trajectory, uniformly randomly.

We ran 6 independent experiments for each set of presented parameters and calculated averages to present the results in Figure 2, along with calculating the error bars using the averages' sample variance.

Appendix C: Normality Investigation

To justify setting the error threshold $\delta \to \infty$, we ran a training experiment for two different τ at $s = 50$ using a base batch size of 240. These experiments were run for $20,000$ epochs and samples at intervals of $5,000$ epochs. Histograms of the individual Δ samples across the entire batch are plotted, along with a fitted curve showing the expected normal distribution given the empirical mean and variance of the batch data. This is presented in Figure 4.

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT Press, 2016).

[2] J. W. Rae, S. Borgeaud, T. Cai, K. Millican, J. Hoffmann, F. Song, J. Aslanides, S. Henderson, R. Ring, S. Young, E. Rutherford, T. Hennigan, J. Menick, A. Cassirer, R. Powell, G. van den Driessche, L. A. Hendricks, M. Rauh, P.-S. Huang, A. Glaese, J. Welbl, S. Dathathri, S. Huang, J. Uesato, J. Mellor, I. Higgins, A. Creswell, N. McAleese, A. Wu, E. Elsen, S. Jayakumar, E. Buchatskaya, D. Budden, E. Sutherland, K. Simonyan, M. Paganini, L. Sifre, L. Martens, X. L. Li, A. Kuncoro, A. Nematzadeh, E. Gribovskaya, D. Donato, A. Lazaridou, A. Mensch, J.-B. Lespiau, M. Tsimpoukelli, N. Grigorev, D. Fritz, T. Sottiaux, M. Pajarskas, T. Pohlen, Z. Gong, D. Toyama, C. de Masson d'Autume, Y. Li, T. Terzi, V. Mikulik, I. Babuschkin, A. Clark, D. de Las Casas, A. Guy, C. Jones, J. Bradbury, M. Johnson, B. Hechtman, L. Weidinger, I. Gabriel, W. Isaac, E. Lockhart, S. Osindero, L. Rimell, C. Dyer, O. Vinyals, K. Ayoub, J. Stanway, L. Bennett, D. Hassabis, K. Kavukcuoglu, and G. Irving, Scaling language models: Methods, analysis & insights from training gopher, arXiv:2112.11446 (2022).

[3] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto, Stanford alpaca: An instruction-following llama model, GitHub repository (2023).

[4] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, Llama: Open and efficient foundation language models, arXiv:2302.13971 (2023).

[5] Y. Wang, Y. Kordi, S. Mishra, A. Liu, N. A. Smith, D. Khashabi, and H. Hajishirzi, Self-instruct: Aligning language model with self generated instructions, arXiv:2212.10560 (2022).

[6] Y. Liu, T. Han, S. Ma, J. Zhang, Y. Yang, J. Tian, H. He, A. Li, M. He, Z. Liu, Z. Wu, D. Zhu, X. Li, N. Qiang, D. Shen, T. Liu, and B. Ge, Summary of chatgpt/gpt-4 research and perspective towards the future of large language models, arXiv:2304.01852 (2023).

[7] Y. Wen, D. Tran, and J. Ba, Batchensemble: An alternative approach to efficient ensemble and lifelong learning, arXiv:2002.06715 (2020).

[8] X. Wang, D. Kondratyuk, E. Christiansen, K. M. Kitani, Y. Alon, and E. Eban, Wisdom of committees: An overlooked approach to faster and more accurate models, arXiv:2012.01988 (2022).

[9] J. F. Mair, D. C. Rose, and J. P. Garrahan, Training neural network ensembles via trajectory sampling, arXiv:2209.11116 (2022).

[10] D. Chandler and J. P. Garrahan, Dynamics on the Way to Forming Glass: Bubbles in Space-Time, Annu. Rev. Phys. Chem. **61**, 191 (2010).

[11] J. P. Garrahan, R. L. Jack, V. Lecomte, E. Pitard, K. van Duijvendijk, and F. van Wijland, Dynamical first-order phase transition in kinetically constrained models of glasses, Phys. Rev. Lett. **98**, 195702 (2007).

[12] R. L. Jack, L. O. Hedges, J. P. Garrahan, and D. Chandler, Preparation and relaxation of very stable glassy states of a simulated liquid, Phys. Rev. Lett. **107**, 275702 (2011).

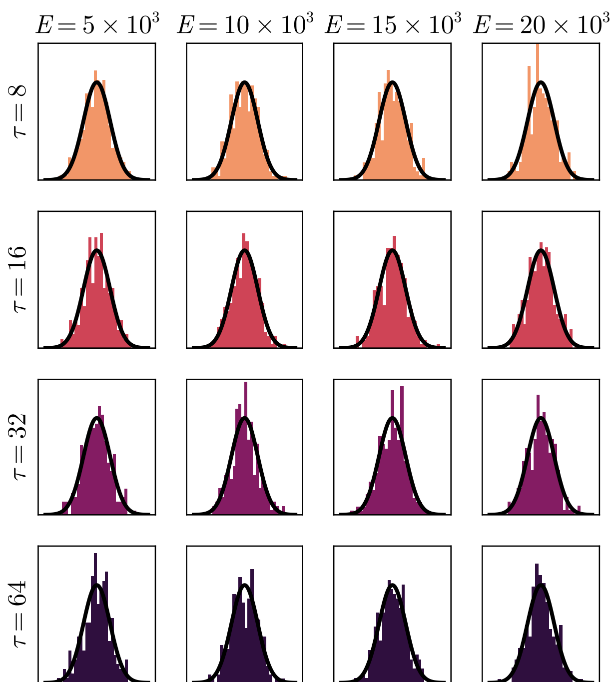

FIG. 4. A plot visually inspecting the normality of the Δ^* variable various values of τ at different specific epochs in the training process. This is done by sampling the delta loss for a single proposed change on the entire dataset of samples. These samples are split into minibatches of size 240 and averages are taken, this gives a sample of Δ^*. The distribution for the minibatches is plotted in each subfigure, along with a normal PDF (plotted as a solid black line) with a mean matching the true mean Δ and variance matching the empirical variance of the samples. Each row investigates a different τ and each column represents a different epoch in the training process, denoted by E.

[13] P. G. Bolhuis, D. Chandler, C. Dellago, and P. L. Geissler, Transition path sampling: throwing ropes over rough mountain passes, in the dark, Annu. Rev. Phys. Chem. **53**, 291 (2002).

[14] L. O. Hedges, R. L. Jack, J. P. Garrahan, and D. Chandler, Dynamic order-disorder in atomistic models of structural glass formers, Science **323**, 1309 (2009).

[15] D. Seita, X. Pan, H. Chen, and J. Canny, An efficient minibatch acceptance test for metropolis-hastings, in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI'18 (AAAI Press, 2018) pp. 5359–5363.

[16] H. Touchette, The large deviation approach to statistical mechanics, Phys. Rep. **478**, 1 (2009).

[17] J. Grela, S. N. Majumdar, and G. Schehr, Non-intersecting brownian bridges in the flat-to-flat geometry, J. Stat. Phys. **183**, 49 (2021).

[18] B. De Bruyne, S. N. Majumdar, and G. Schehr, Generating discrete-time constrained random walks and lévy flights, Phys. Rev. E **104**, 024117 (2021).

[19] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, Gradient-based learning applied to document recognition, Proc. IEEE **86**, 2278 (1998).

[20] L. Bottou, Large-scale machine learning with stochastic gradient descent, in *Proceedings of COMPSTAT'2010*, edited by Y. Lechevallier and G. Saporta (Physica-Verlag HD, Heidelberg, 2010) pp. 177–186.

[21] A. Korattikara, Y. Chen, and M. Welling, Austerity in mcmc land: Cutting the metropolis-hastings budget, arXiv:1304.5299 (2014).

[22] R. Bardenet, A. Doucet, and C. Holmes, Towards scaling up markov chain monte carlo: an adaptive subsampling approach, in *Proceedings of the 31st International Conference on Machine Learning*, Proceedings of Machine Learning Research, Vol. 32, edited by E. P. Xing and T. Jebara (PMLR, Bejing, China, 2014) pp. 405–413.

[23] R. Bardenet, A. Doucet, and C. Holmes, On markov chain monte carlo methods for tall data, J. Mach. Learn. Res. **18** (2017).

[24] L. Deng, The mnist database of handwritten digit images for machine learning research, IEEE Signal Processing Magazine **29**, 141 (2012).

[25] G. Folena, A. Manacorda, and F. Zamponi, Introduction to the dynamics of disordered systems: Equilibrium and gradient descent, Physica A: , 128152 (2022).

[26] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction* (A Bradford Book, Cambridge, MA, USA, 2018).

[27] E. Bengio, J. Pineau, and D. Precup, Interference and generalization in temporal difference learning, arXiv:2003.06350 (2020).

[28] M. McCloskey and N. J. Cohen, Catastrophic interference in connectionist networks: The sequential learning problem (Academic Press, 1989) pp. 109–165.

[29] K. Khetarpal, M. Riemer, I. Rish, and D. Precup, Towards continual reinforcement learning: A review and perspectives, J. Artif. Intell. Res. **75**, 1401 (2022).

[30] S. Mei, A. Montanari, and P.-M. Nguyen, A mean field view of the landscape of two-layer neural networks, Proc. Natl. Acad. Sci. USA **115**, E7665 (2018).

[31] G. Rotskoff and E. Vanden-Eijnden, Trainability and accuracy of artificial neural networks: An interacting particle system approach, Commun. Pure. Appl. Math. **75**, 1889 (2022).

[32] S. Whitelam, V. Selin, S.-W. Park, and I. Tamblyn, Correspondence between neuroevolution and gradient descent, arXiv:2008.06643 (2021).

[33] R. Veiga, L. Stephan, B. Loureiro, F. Krzakala, and L. Zdeborová, Phase diagram of stochastic gradient descent in high-dimensional two-layer neural networks, arXiv:2202.00293 (2022).

[34] S. Adhikari, A. Kabakçıoğlu, A. Strang, D. Yuret, and M. Hinczewski, Machine learning in and out of equilibrium, arXiv:2306.03521 (2023).

[35] C. Baldassi and R. Zecchina, Efficiency of quantum vs. classical annealing in nonconvex learning problems, Proc. Natl. Acad. Sci. USA **115**, 1457 (2018).

[36] G. Lami, P. Torta, G. E. Santoro, and M. Collura, Quantum annealing for neural network optimization problems: A new approach via tensor network simulations, SciPost Phys. **14**, 117 (2023).

[37] J. Mair, TransitionPathSampling.jl, Available at https://github.com/JamieMair/TransitionPathSampling.jl ().

[38] J. Mair, MinibatchTPS.jl, Available at https://github.com/JamieMair/MinibatchTPS.jl ().

[39] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, Julia: A fresh approach to numerical computing, SIAM Rev. Soc. Ind. Appl. Math. **59**, 65 (2017).

[40] T. Besard, C. Foket, and B. De Sutter, Effective extensible programming: Unleashing Julia on GPUs, IEEE Trans. Parallel. Distrib. Syst. **30**, 827 (2018).

[41] T. Besard, V. Churavy, A. Edelman, and B. De Sutter, Rapid software prototyping for heterogeneous and distributed platforms, Adv. Eng. Softw. **132**, 29 (2019).

[42] S. Danisch and J. Krumbiegel, Makie.jl: Flexible high-performance data visualization for Julia, J. Open Source Softw. **6**, 3349 (2021).

[43] J. Revels, M. Lubin, and T. Papamarkou, Forward-mode automatic differentiation in Julia, arXiv:1607.0789 (2016).

Conclusions & Outlook

This body of work adds to the growing list of resources that make a connection between statistical mechanics and machine learning. While the connection between these topics has a very long history (e.g. see reviews [17–19, 23]), recent advances in machine learning can further help to bridge the gap between the topics.

The first contribution from this thesis is the discovery of the connection between rare trajectory sampling and reinforcement learning. In Chapter 6, we presented a theoretical framework where one can formulate generating optimal dynamics as a reinforcement learning problem. One constructs a reward function which is optimised by a policy which generates an ensemble of trajectories consistent with a biased (or tilted) dynamics. Existing bodies of work into reinforcement learning are usually only concerned with, in general, a deterministic optimal policy [27]. The comparative lack of approaches for generating specific stochastic policies made training an optimal ensemble generator practically impossible prior to our work, as traditional methods failed to accurately converge on a desired policy. Our work remedies this issue, and allows the application of reinforcement learning to learn optimal sampling dynamics, even in high dimensional scenarios, where existing methods require domain knowledge or heuristics to obtain useful auxiliary dynamics.

Our second contribution, presented in Chapter 7, is a novel training method based on MCMC statistical sampling methods. While MCMC sampling methods are commonly used in Bayesian machine learning [145], they are most often focused on inference rather than training. Our approach extends the idea of simulated annealing [20] (sometimes called neuroevolution [143] when applied to neural networks), by instead training an ensemble of models, upon which we impose some dynamics connecting the parameters — forming a "trajectory". This dynamics encourages adjacent models in the trajectory to have similar parameters (tuned with a hyperparameter σ), and be further biased towards having a low total loss. Coupling the model in this way produces an ensemble that far improves on the existing neuroevolution technique, resulting in access to lower loss models for the same value of the hyperparameter of temperature. In comparison with traditional gradient-based methods, our approach also makes use of the temperature hyperparameter, which can be tuned to ensure that the resulting ensemble of models does not overfit on a small dataset, making it viable at small scales where data is very limited and generalisation is difficult.

The third contribution further develops this line of research, presented in Chapter 8, where we co-opt the traditional approach of minibatching to improve the computational efficiency of the ensemble training method. This allows the computational complexity of updates to the parameters to be decoupled from the size of the dataset, and even allows

for studying infinite dataset problems (such as stochastic RL problems).

This research has many avenues for future development and work. Firstly, there has already been some effort into extending the reinforcement learning applications to sampling rare molecular dynamics trajectories [146]. Work is currently underway to build a suite of benchmarks, much like OpenAI's gym [75], specifically tailored towards statistical mechanics problems, to better develop RL algorithms for use in statistical mechanics.

An interesting extension to our NNE research is to assess whether the ensemble training approach would be applicable to RL to help avoid catastrophic forgetting [147] and enable continual learning [148]. We hypothesise that our NNE training approach will prove very stable on RL problems, similar to those studied in Chapter 6.

While we have demonstrated the ensemble training approach on a reasonably complex neural architecture, it would be interesting to examine the limits of this approach in training large neural networks with potentially millions of parameters. We hypothesise that this will be feasible, but improvements to the training time of these NNEs are of more immediate interest, as these improvements will inevitably aid scaling up to larger networks.

There are many avenues to explore with the goal of improving the training time of these NNEs. A simple extension would be to introduce an adaptive step size, reminiscent of how gradient descent takes steps proportional to the size of the gradient. To facilitate this, the coupling between adjacent models in the trajectory could be dynamically adjusted based on factors such as the acceptance rate. While this would adjust the target of a stationary distribution, it could allow automatic adjustment throughout training and lead to a faster trained (and potentially lower loss) NNE. If one wishes to keep a fixed target distribution, one could instead adjust the perturbation dynamics to alter a higher proportion of parameters, or number of models, based on the acceptance rate. This could vastly improve training time by serving as an "automatic" hyperparameter optimisation heuristic which runs online during training.

Other future training optimisations could focus on practical implementation improvements, such as parallelising the training of the NNE. Knowing that the trajectory sampling methods only make updates, local to only parts of the trajectories, we can segment the trajectories across multiple machines. Only models on the edge of the segment boundaries would need to communicate their parameters with the other machines, avoiding the introduction of a large communication bottleneck. In effect, one could train very large NNEs (i.e. long trajectory) in parallel, with each part of the trajectory evolving almost independently of one another, with some long distance interactions facilitated by edges of the segments.

Additionally, there is a large body of research dedicated to improving the efficiency of MCMC methods, as they are known to be computationally expensive. One such method is Hamiltonian Monte Carlo (HMC) [149], which extends the original Metropolis-Hastings algorithm used throughout this body of work. HMC aims to introduce momentum into the proposal dynamics, incorporating the gradient of the energy function into the updates, allowing an effective large step size to be used while maintaining a high acceptance rate. While we do not propose directly applying HMC to our NNE training technique, integrating gradient-based techniques *à la* HMC with our stochastic process may have positive effects on the training time and quality of the NNE produced.

For our final suggestion, we start by noting that the NNE training technique currently is completely architecture agnostic. While enforce a homogenous architecture on all models in the trajectory, the actual structure of the model does not inform the training process beyond its effect on the loss. Currently, there is only one coupling parameter which is not only homogenous in time (i.e. along the trajectory), but homogenous in the space of parameters. While this choice makes the implementation and analysis simpler, we hypothesise that it is neither necessary nor optimal for effective training of a NNE. Rather, one can imagine using a heterogeneous coupling vector for the parameters, adjusted based on the architecture of the neural network itself. This may be particularly advantageous in networks where the number of parameters in the layers varies across several orders of magnitude.

Bibliography

[1] J. Jumper, R. Evans, A. Pritzel, *et al.* "Highly accurate protein structure prediction with AlphaFold". In: *Nature* 596.7873 (2021), pp. 583–589.

[2] OpenAI. "GPT-4 Technical Report". 2023. arXiv: 2303.08774.

[3] A. Vaswani, N. Shazeer, N. Parmar, *et al.* "Attention is all you need". In: *Advances in Neural Information Processing Systems* 30 (2017).

[4] H. Touvron, T. Lavril, G. Izacard, *et al.* "LLaMA: Open and Efficient Foundation Language Models". 2023. arXiv: 2302.13971.

[5] A. Köpf, Y. Kilcher, D. von Rütte, *et al.* "OpenAssistant Conversations–Democratizing Large Language Model Alignment". 2023. arXiv: 2304.07327.

[6] A. Chowdhery, S. Narang, J. Devlin, *et al.* "PaLM: Scaling Language Modeling with Pathways". 2022. arXiv: 2204.02311.

[7] J. Scharcanski and M. E. Celebi. *Computer vision techniques for the diagnosis of skin cancer*. Springer, 2013.

[8] W. F. Cueva, F Muñoz, G Vásquez, *et al.* "Detection of skin cancer" Melanoma" through computer vision". In: IEEE. 2017, pp. 1–4.

[9] K Shailaja, B. Seetharamulu, and M. Jabbar. "Machine learning in healthcare: A review". In: IEEE. 2018, pp. 910–914.

[10] R. Bhardwaj, A. R. Nambiar, and D. Dutta. "A study of machine learning in healthcare". In: vol. 2. IEEE. 2017, pp. 236–241.

[11] K. Kashinath, M Mustafa, A. Albert, *et al.* "Physics-informed machine learning: case studies for weather and climate modelling". In: *Philosophical Transactions of the Royal Society A* 379.2194 (2021), p. 20200093.

[12] I. Akkaya, M. Andrychowicz, M. Chociej, *et al.* "Solving rubik's cube with a robot hand". 2019. arXiv: 1910.07113.

[13] S. Gu, E. Holly, T. Lillicrap, *et al.* "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates". In: 2017, pp. 3389–3396. DOI: 10.1109/ICRA.2017.7989385.

[14] S. Levine, P. Pastor, A. Krizhevsky, *et al.* "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection". In: *The International Journal of Robotics Research* 37.4-5 (2018), pp. 421–436.

[15] S. L. Brunton, J. L. Proctor, and J. N. Kutz. "Discovering governing equations from data by sparse identification of nonlinear dynamical systems". In: *Proceedings of the National Academy of Sciences* 113.15 (2016), pp. 3932–3937. DOI: 10.1073/pnas.1517384113.

[16] B. M. de Silva, D. M. Higdon, S. L. Brunton, *et al.* "Discovery of Physics From Data: Universal Laws and Discrepancies". In: *Frontiers in Artificial Intelligence* 3 (2020). DOI: 10.3389/frai.2020.00025.

[17] L. Zdeborová and F. Krzakala. "Statistical physics of inference: thresholds and algorithms". In: *Advances in Physics* 65.5 (2016), pp. 453–552. DOI: 10.1080/00018732.2016.1211393.

[18] J. A. Hertz. *Introduction to the Theory of Neural Computation*. 1st. CRC Press, 1991. DOI: 10.1201/9780429499661.

[19] A. Coolen, R. Kühn, and P. Sollich. *Theory of Neural Information Processing Systems*. OUP Oxford, 2005.

[20] P. J. Van Laarhoven, E. H. Aarts, P. J. van Laarhoven, *et al. Simulated annealing*. Springer, 1987.

[21] T. L. H. Watkin, A. Rau, and M. Biehl. "The statistical mechanics of learning a rule". In: *Reviews of Modern Physics* 65 (2 1993), pp. 499–556. DOI: 10.1103/RevModPhys.65.499.

[22] H. S. Seung, H. Sompolinsky, and N. Tishby. "Statistical mechanics of learning from examples". In: *Physical review A* 45.8 (1992), p. 6056.

[23] Y. Bahri, J. Kadmon, J. Pennington, *et al.* "Statistical mechanics of deep learning". In: *Annual Review of Condensed Matter Physics* 11 (2020), pp. 501–528.

[24] D. Wu, L. Wang, and P. Zhang. "Solving Statistical Mechanics Using Variational Autoregressive Networks". In: *Physical Review Letters* 122 (8 2019), p. 080602. DOI: 10.1103/PhysRevLett.122.080602.

[25] J. Carrasquilla. "Machine learning for quantum matter". In: *Advances in Physics: X* 5.1 (2020), p. 1797528. DOI: 10.1080/23746149.2020.1797528.

[26] H. Touchette. "Introduction to dynamical large deviations of Markov processes". In: *Physica A: Statistical Mechanics and its Applications* 504 (2018), pp. 5–19. DOI: 10.1016/j.physa.2017.10.046.

[27] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. 2nd ed. MIT Press, 2018.

[28] L. Deng. "The mnist database of handwritten digit images for machine learning research". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142.

[29] Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444.

[30] P. Wang, E. Fan, and P. Wang. "Comparative analysis of image classification algorithms based on traditional machine learning and deep learning". In: *Pattern Recognition Letters* 141 (2021), pp. 61–67. DOI: https://doi.org/10.1016/j.patrec.2020.07.042.

[31] N. Sharma, V. Jain, and A. Mishra. "An Analysis Of Convolutional Neural Networks For Image Classification". In: *Procedia Computer Science* 132 (2018). International Conference on Computational Intelligence and Data Science, pp. 377–384. DOI: https://doi.org/10.1016/j.procs.2018.05.198.

[32] K. Lee, J. Zung, P. Li, *et al.* "Superhuman Accuracy on the SNEMI3D Connectomics Challenge". 2017. arXiv: 1706.00120.

[33] P. H. Yi, T. K. Kim, J. Wei, *et al.* "Automated semantic labeling of pediatric musculoskeletal radiographs using deep learning". In: *Pediatric Radiology* 49.8 (2019), pp. 1066–1070. DOI: 10.1007/s00247-019-04408-2.

[34] S. Haggenmüller, R. C. Maron, A. Hekler, *et al.* "Skin cancer classification via convolutional neural networks: systematic review of studies involving human experts". In: *European Journal of Cancer* 156 (2021), pp. 202–216. DOI: https://doi.org/10.1016/j.ejca.2021.06.049.

[35] F. M. Calisto, C. Santiago, N. Nunes, *et al.* "Introduction of human-centric AI assistant to aid radiologists for multimodal breast image classification". In: *International Journal of Human-Computer Studies* 150 (2021), p. 102607. DOI: https://doi.org/10.1016/j.ijhcs.2021.102607.

[36] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[37] C. M. Bishop and N. M. Nasrabadi. *Pattern recognition and machine learning*. Vol. 4. 4. Springer, 2006.

[38] J. Li, K. Cheng, S. Wang, *et al.* "Feature selection: A data perspective". In: *ACM Computing Surveys (CSUR)* 50.6 (2017), pp. 1–45.

[39] Å. Björck. "Least squares methods". In: *Handbook of Numerical Analysis* 1 (1990), pp. 465–652.

[40] A. Plaat. *Deep Reinforcement Learning*. Springer, 2022.

[41] D. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization". 2015. arXiv: 1412.6980.

[42] T. J. Ypma. "Historical development of the Newton–Raphson method". In: *SIAM review* 37.4 (1995), pp. 531–551.

[43] M Biehl and H Schwarze. "Learning by on-line gradient descent". In: *Journal of Physics A: Mathematical and General* 28.3 (1995), p. 643. DOI: 10.1088/0305-4470/28/3/018.

[44] M. Li, T. Zhang, Y. Chen, *et al.* "Efficient mini-batch training for stochastic optimization". In: 2014, pp. 661–670.

[45] V. S. Borkar. *Stochastic approximation: a dynamical systems viewpoint*. Vol. 48. Springer, 2009.

[46] J Harold, G Kushner, and G. Yin. "Stochastic approximation and recursive algorithm and applications". In: *Application of Mathematics* 35 (1997).

[47] D. Silver, T. Hubert, J. Schrittwieser, *et al.* "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018), pp. 1140–1144.

[48] D. Silver, J. Schrittwieser, K. Simonyan, *et al.* "Mastering the game of go without human knowledge". In: *Nature* 550.7676 (2017), pp. 354–359.

[49] G. Thömmes, R. Pinnau, M. Seaid, *et al.* "Numerical methods and optimal control for glass cooling processes". In: *Transport Theory and Statistical Physics* 31.4-6 (2002), pp. 513–529.

[50] M. Frank, A. Klar, and R. Pinnau. "Optimal control of glass cooling using simplified PN theory". In: *Transport theory and statistical physics* 39.2-4 (2010), pp. 282–311.

[51] H. J. Kappen. "Path integrals and symmetry breaking for optimal control theory". In: *Journal of Statistical Mechanics: Theory and Experiment* 2005.11 (2005), P11011.

[52] V. Y. Chernyak, M. Chertkov, J. Bierkens, *et al.* "Stochastic optimal control as non-equilibrium statistical mechanics: Calculus of variations over density and current". In: *Journal of Physics A: Mathematical and Theoretical* 47.2 (2013), p. 022001.

[53] S. A. Vaghefi, M. A. Jafari, J. Zhu, *et al.* "A hybrid physics-based and data driven approach to optimal control of building cooling/heating systems". In: *IEEE Transactions on Automation Science and Engineering* 13.2 (2014), pp. 600–610.

[54] W. B. Powell. *Approximate Dynamic Programming: Solving the curses of dimensionality*. Vol. 703. John Wiley & Sons, 2007.

[55] J. Si, A. G. Barto, W. B. Powell, *et al. Handbook of learning and approximate dynamic programming*. Vol. 2. John Wiley & Sons, 2004.

[56] A. Krizhevsky, G. Hinton, *et al.* "Learning multiple layers of features from tiny images". In: (2009).

[57] P. Kumar, K. Sinha, N. K. Nere, *et al.* "A machine learning framework for computationally expensive transient models". In: *Scientific reports* 10.1 (2020), pp. 1–11.

[58] F. Noé, A. Tkatchenko, K.-R. Müller, *et al.* "Machine learning for molecular simulation". In: *Annual review of physical chemistry* 71 (2020), pp. 361–390.

[59] D. Kochkov, J. A. Smith, A. Alieva, *et al.* "Machine learning–accelerated computational fluid dynamics". In: *Proceedings of the National Academy of Sciences* 118.21 (2021), e2101784118.

[60] J. Wang, S. Olsson, C. Wehmeyer, *et al.* "Machine learning of coarse-grained molecular dynamics force fields". In: *ACS central science* 5.5 (2019), pp. 755–767.

[61] J. Westermayr, M. Gastegger, M. F. Menger, *et al.* "Machine learning enables long time scale molecular photodynamics simulations". In: *Chemical Science* 10.35 (2019), pp. 8100–8107.

[62] J. E. Staddon and D. T. Cerutti. "Operant conditioning". In: *Annual Review of Psychology* 54.1 (2003), pp. 115–144.

[63] H. van Hasselt, D. Borsa, and M. Hessel. "Reinforcement Learning Lecture Series". 2021.

[64] D. Silver, S. Singh, D. Precup, *et al.* "Reward is enough". In: *Artificial Intelligence* 299 (2021), p. 103535. DOI: https://doi.org/10.1016/j.artint.2021.103535.

[65] D. Dewey. "Reinforcement learning and the reward engineering principle". In: 2014.

[66] A. Ng and S. Russell. "Algorithms for inverse reinforcement learning". In: vol. 1. 2000, p. 2.

[67] M. Stone. *Mathematics for physics : a guided tour for graduate students / Michael Stone, Paul Goldbart*. Cambridge University Press, 2009.

[68] R. Bellman. "Dynamic programming". In: *Science* 153.3731 (1966), pp. 34–37.

[69] E. W. Dijkstra. "A note on two problems in connexion with graphs". In: 2022, pp. 287–290.

[70] P. E. Hart, N. J. Nilsson, and B. Raphael. "A formal basis for the heuristic determination of minimum cost paths". In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[71] D. Silver, A. Huang, C. Maddison, *et al.* "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (2016), pp. 484–489. DOI: 10.1038/nature16961.

[72] K. P. Murphy. *Probabilistic machine learning: an introduction*. MIT press, 2022.

[73] P. Dayan. "The convergence of TD (λ) for general λ". In: *Machine Learning* 8 (1992), pp. 341–362.

[74] P. Dayan and T. J. Sejnowski. "TD (λ) converges with probability 1". In: *Machine Learning* 14 (1994), pp. 295–301.

[75] G. Brockman, V. Cheung, L. Pettersson, *et al.* "OpenAI Gym". 2016. arXiv: 1606.01540.

[76] A. Scorsoglio, R. Furfaro, R. Linares, *et al.* "Image-based deep reinforcement learning for autonomous lunar landing". In: 2020, p. 1910.

[77] C. Badue, R. Guidolini, R. V. Carneiro, *et al.* "Self-driving cars: A survey". In: *Expert Systems with Applications* 165 (2021), p. 113816.

[78] A. Hussein, M. M. Gaber, E. Elyan, *et al.* "Imitation learning: A survey of learning methods". In: *ACM Computing Surveys (CSUR)* 50.2 (2017), pp. 1–35.

[79] K. Arulkumaran, M. P. Deisenroth, M. Brundage, *et al.* "Deep Reinforcement Learning: A Brief Survey". In: *IEEE Signal Processing Magazine* 34.6 (2017), pp. 26–38. DOI: 10.1109/MSP.2017.2743240.

[80] V. François-Lavet, P. Henderson, R. Islam, *et al.* "An introduction to deep reinforcement learning". In: *Foundations and Trends® in Machine Learning* 11.3-4 (2018), pp. 219–354.

[81] S. Zhang and R. S. Sutton. "A deeper look at experience replay". 2017. arXiv: 1712.01275.

[82] H. Van Hasselt, A. Guez, and D. Silver. "Deep reinforcement learning with double q-learning". In: vol. 30. 1. 2016.

[83] V. Mnih, A. P. Badia, M. Mirza, *et al.* "Asynchronous methods for deep reinforcement learning". In: PMLR. 2016, pp. 1928–1937.

[84] R. J. Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Reinforcement Learning* (1992), pp. 5–32.

[85] R. S. Sutton, D. McAllester, S. Singh, *et al.* "Policy gradient methods for reinforcement learning with function approximation". In: *Advances in Neural Information Processing Systems* 12 (1999).

[86] K. Hornik, M. Stinchcombe, and H. White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8.

[87] A. Paszke, S. Gross, F. Massa, *et al.* "PyTorch: An Imperative Style, High-Performance Deep Learning Library". 2019. arXiv: 1912.01703.

[88] M. Abadi, A. Agarwal, P. Barham, *et al.* "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems". Software available from tensorflow.org. 2015.

[89] H. Touchette. "A basic introduction to large deviations: Theory, applications, simulations". 2012. arXiv: 1106.4146.

[90] H. Touchette. "The large deviation approach to statistical mechanics". In: *Physics Reports* 478.1-3 (2009), pp. 1–69. DOI: 10.1016/j.physrep.2009.05.002.

[91] R. S. Ellis. "An overview of the theory of large deviations and applications to statistical mechanics". In: *Scandinavian Actuarial Journal* 1995.1 (1995), pp. 97–142.

[92] J. P. Garrahan. "Aspects of non-equilibrium in classical and quantum systems: Slow relaxation and glasses, dynamical large deviations, quantum non-ergodicity, and open quantum dynamics". In: *Physica A: Statistical Mechanics and its Applications* 504 (2018), pp. 130–154. DOI: 10.1016/j.physa.2017.12.149.

[93] R. T. Rockafellar. *Convex analysis*. Vol. 11. Princeton university press, 1997.

[94] J. V. Uspensky *et al.* "Introduction to mathematical probability". In: (1937).

[95] H. Cramér. "Sur un nouveau théoreme-limite de la théorie des probabilités". In: *Actualites Scientifiques et Industrielles* 736 (1938), pp. 5–23.

[96] H. Cramér and H. Touchette. "On a new limit theorem in probability theory (Translation of 'Sur un nouveau théorème-limite de la théorie des probabilités')". 2022. arXiv: 1802.05988.

[97] C. W. Gardiner *et al.* *Handbook of stochastic methods*. Vol. 3. springer Berlin, 1985.

[98] J. P. Garrahan and F. Pollmann. "Topological phases in the dynamics of the simple exclusion process". 2022. arXiv: 2203.08200.

[99] J. W. P. Wilkinson, T. Prosen, and J. P. Garrahan. "Exact solution of the "Rule 150" reversible cellular automaton". In: *Physical Review E* 105.3 (2022). DOI: 10.1103/physreve.105.034124.

[100] L. Causer, M. C. Bañuls, and J. P. Garrahan. "Finite Time Large Deviations via Matrix Product States". In: *Physical Review Letters* 128.9 (2022). DOI: 10.1103/physrevlett.128.090605.

[101] L. Causer, M. C. Bañuls, and J. P. Garrahan. "Optimal sampling of dynamical large deviations via matrix product states". In: *Physical Review E* 103.6 (2021). DOI: 10.1103/physreve.103.062144.

[102] F. Carollo, J. P. Garrahan, and R. L. Jack. "Large Deviations at Level 2.5 for Markovian Open Quantum Systems: Quantum Jumps and Quantum State Diffusion". In: *Journal of Statistical Physics* 184.1 (2021). DOI: 10.1007/s10955-021-02799-x.

[103] L. Causer, I. Lesanovsky, M. C. Bañuls, *et al.* "Dynamics and large deviation transitions of the XOR-Fredrickson-Andersen kinetically constrained model". In: *Physical Review E* 102.5 (2020). DOI: 10.1103/physreve.102.052132.

[104] L. M. Vasiloiu, T. H. E. Oakes, F. Carollo, *et al.* “Trajectory phase transitions in noninteracting spin systems”. In: *Physical Review E* 101.4 (2020). DOI: 10.1103/physreve.101.042115.

[105] M. C. Bañuls and J. P. Garrahan. “Using Matrix Product States to Study the Dynamical Large Deviations of Kinetically Constrained Models”. In: *Physical Review Letters* 123.20 (2019). DOI: 10.1103/physrevlett.123.200601.

[106] B. Buč a, J. P. Garrahan, T. Prosen, *et al.* “Exact large deviation statistics and trajectory phase transition of a deterministic boundary driven cellular automaton”. In: *Physical Review E* 100.2 (2019). DOI: 10.1103/physreve.100.020103.

[107] F. Carollo, R. L. Jack, and J. P. Garrahan. “Unraveling the Large Deviation Statistics of Markovian Open Quantum Systems”. In: *Physical Review Letters* 122.13 (2019). DOI: 10.1103/physrevlett.122.130605.

[108] C. Pé rez-Espigares, I. Lesanovsky, J. P. Garrahan, *et al.* “Glassy dynamics due to a trajectory phase transition in dissipative Rydberg gases”. In: *Physical Review A* 98.2 (2018). DOI: 10.1103/physreva.98.021804.

[109] T. Oakes, S. Powell, C. Castelnovo, *et al.* “Phases of quantum dimers from ensembles of classical stochastic trajectories”. In: *Physical Review B* 98.6 (2018). DOI: 10.1103/physrevb.98.064302.

[110] J. P. Garrahan, R. L. Jack, V. Lecomte, *et al.* “First-order dynamical phase transition in models of glasses: an approach based on ensembles of histories”. In: *Journal of Physics A: Mathematical and Theoretical* 42.7 (2009), p. 075007. DOI: 10.1088/1751-8113/42/7/075007.

[111] R. L. Jack and J. P. Garrahan. “Metastable states and space-time phase transitions in a spin-glass model”. In: *Physical Review E* 81.1 (2010). DOI: 10.1103/physreve.81.011111.

[112] L. Causer, M. C. Bañuls, and J. P. Garrahan. “Optimal sampling of dynamical large deviations via matrix product states”. In: *Physical Review E* 103 (6 2021), p. 062144. DOI: 10.1103/PhysRevE.103.062144.

[113] D. Simon. “Construction of a coordinate Bethe ansatz for the asymmetric simple exclusion process with open boundaries”. In: *Journal of Statistical Mechanics: Theory and Experiment* 2009.07 (2009), P07017. DOI: 10.1088/1742-5468/2009/07/P07017.

[114] V. Popkov, G. M. Schütz, and D. Simon. “ASEP on a ring conditioned on enhanced flux”. In: *Journal of Statistical Mechanics: Theory and Experiment* 2010.10 (2010), P10007. DOI: 10.1088/1742-5468/2010/10/P10007.

[115] R. L. Jack and P. Sollich. “Large Deviations and Ensembles of Trajectories in Stochastic Models”. In: *Progress of Theoretical Physics Supplement* 184 (Mar. 2010), pp. 304–317. DOI: 10.1143/PTPS.184.304.

[116] R. Chetrite and H. Touchette. “Nonequilibrium Markov Processes Conditioned on Large Deviations”. In: *Annales Henri Poincaré* 16.9 (2015), pp. 2005–2057. DOI: 10.1007/s00023-014-0375-8.

[117] T. Oakes, S. Powell, C. Castelnovo, *et al.* “Phases of quantum dimers from ensembles of classical stochastic trajectories”. In: *Physical Review B* 98 (6 2018), p. 064302. DOI: 10.1103/PhysRevB.98.064302.

[118] F. Carollo, J. P. Garrahan, I. Lesanovsky, *et al.* "Making rare events typical in Markovian open quantum systems". In: *Physical Review A* 98 (1 2018), p. 010103. DOI: 10.1103/PhysRevA.98.010103.

[119] W. Krauth. "Statistical Mechanics: Algorithms and Computations". In: 2006.

[120] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, *et al.* "Equation of state calculations by fast computing machines". In: *The Journal of Chemical Physics* 21.6 (1953), pp. 1087–1092.

[121] W. K. Hastings. "Monte Carlo sampling methods using Markov chains and their applications". In: (1970).

[122] C. Robert and G. Casella. "A Short History of Markov Chain Monte Carlo: Subjective Recollections from Incomplete Data". In: *Statistical Science* 26.1 (2011), pp. 102 –115. DOI: 10.1214/10-STS351.

[123] B. A. Cipra. "An Introduction to the Ising Model". In: *The American Mathematical Monthly* 94.10 (1987), pp. 937–959. DOI: 10.1080/00029890.1987.12000742.

[124] J. Bezanson, A. Edelman, S. Karpinski, *et al.* "Julia: A fresh approach to numerical computing". In: *SIAM Review* 59.1 (2017), pp. 65–98.

[125] J. Mair. "High Performance Computing in Julia, Graduate Course Website". https://jamiemair.github.io/mpags-high-performance-computing. 2023.

[126] J. Mair. "*Experimenter.jl*". Version 0.1.1. Mar. 2022. URL: https://github.com/JamieMair/Experimenter.jl.

[127] J. Mair. "*TransitionPathSampling.jl*". Version 0.3.2. June 2022. URL: https://github.com/JamieMair/TransitionPathSampling.jl.

[128] J. Mair. "*SimpleNNs.jl*". Version 0.1.0. June 2022. URL: https://github.com/JamieMair/SimpleNNs.jl.

[129] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.* "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[130] J. Mair. *High Performance Computing in Julia: from the ground up*. 2023. URL: \url{https://jamiemair.github.io/mpags-high-performance-computing/assets/book_v1.2.pdf}.

[131] R. D. Hipp. "SQLite". Version 3.31.1. 2020.

[132] A. B. Yoo, M. A. Jette, and M. Grondona. "SLURM: Simple Linux Utility for Resource Management". In: Springer Berlin Heidelberg, 2003, pp. 44–60.

[133] M. Innes, E. Saba, K. Fischer, *et al.* "Fashionable Modelling with Flux". 2018. arXiv: 1811.01457.

[134] M. Innes. "Flux: Elegant Machine Learning with Julia". In: *Journal of Open Source Software* (2018). DOI: 10.21105/joss.00602.

[135] C. Elrod, N. Korsbo, and C. Rackauckas. "Doing small network scientific machine learning in Julia 5x faster than PyTorch". 2022.

[136] J. Bradbury, R. Frostig, P. Hawkins, *et al.* "*JAX: composable transformations of Python+NumPy programs*". Version 0.3.13. 2018. URL: http://github.com/google/jax.

[137] K. Sfairopoulos, L. Causer, J. F. Mair, *et al.* "Boundary conditions dependence of the phase transition in the quantum Newman-Moore model". 2023. arXiv: 2301.02826.

[138] R. W. Floyd. "Nondeterministic algorithms". In: *Journal of the ACM (JACM)* 14.4 (1967), pp. 636–644.

[139] L. Causer, K. Sfairopoulos, J. F. Mair, *et al.* "Rejection-free quantum Monte Carlo in continuous time from transition path sampling". 2023. arXiv: 2305.08935.

[140] D. C. Rose, J. F. Mair, and J. P. Garrahan. "A reinforcement learning approach to rare trajectory sampling". In: *New Journal of Physics* 23.1 (2021), p. 013013. DOI: 10.1088/1367-2630/abd7bd.

[141] P. G. Bolhuis, D. Chandler, C. Dellago, *et al.* "Transition path sampling: Throwing ropes over rough mountain passes, in the dark". In: *Annual Review of Physical Chemistry* 53.1 (2002), pp. 291–318.

[142] J. F. Mair, D. C. Rose, and J. P. Garrahan. "Training neural network ensembles via trajectory sampling". 2023. arXiv: 2209.11116.

[143] S. Whitelam, V. Selin, S.-W. Park, *et al.* "Correspondence between neuroevolution and gradient descent". In: *Nature Communications* 12.1 (Nov. 2021), p. 6317.

[144] J. F. Mair, L. Causer, and J. P. Garrahan. "Minibatch training of neural network ensembles via trajectory sampling". 2023. arXiv: 2306.13442.

[145] C. Andrieu, N. De Freitas, A. Doucet, *et al.* "An introduction to MCMC for machine learning". In: *Machine Learning* 50 (2003), pp. 5–43.

[146] A. Das, D. C. Rose, J. P. Garrahan, *et al.* "Reinforcement learning of rare diffusive dynamics". In: *The Journal of Chemical Physics* 155.13 (2021), p. 134105. DOI: 10.1063/5.0057323.

[147] M. McCloskey and N. J. Cohen. "Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem". In: vol. 24. Academic Press, 1989, pp. 109–165. DOI: https://doi.org/10.1016/S0079-7421(08)60536-8.

[148] L. Wang, X. Zhang, H. Su, *et al.* "A Comprehensive Survey of Continual Learning: Theory, Method and Application". 2023. arXiv: 2302.00487.

[149] M. Betancourt. "A Conceptual Introduction to Hamiltonian Monte Carlo". 2018. arXiv: 1701.02434.