

LEARNING BETTER PHYSICS:
A MACHINE LEARNING APPROACH TO LATTICE GAUGE THEORY

by

Samuel Alfred Foreman

A thesis submitted in partial fulfillment
of the requirements for the
Doctor of Philosophy
degree in Physics
in the Graduate College of
The University of Iowa

August 2019

Thesis Supervisor: Professor Yannick Meurice

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

Samuel Alfred Foreman

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Physics at the August 2019 graduation.

Thesis Committee: _____
Yannick Meurice, Thesis Supervisor

James Osborn

Vincent Rodgers

Wayne Polyzou

Craig Kletzing

ACKNOWLEDGEMENTS

First and foremost I would like to thank my supervisor Yannick Meurice for his support and encouragement throughout my entire graduate research experience. I would also like to thank both James Osborn, and Xiao-Yong Jin for their incredible patience, guidance, and expertise throughout my time at Argonne National Laboratory.

I would like to thank the University of Iowa Graduate College for their financial support during the writing of this thesis, and all of the University of Iowa Department of Physics & Astronomy Faculty who served as mentors and helped me navigate the graduate school process.

Finally, I would like to thank Argonne National Laboratory and the US Department of Energy for their financial support and hospitality.

This research was supported in part by the Office of Science Graduate Student Research (SCGSR) program. The SCGSR program is administered by the Oak Ridge Institute for Science and Education (ORISE) for the DOE. ORISE is managed by ORAU under contract number DE-SC0014664.

ABSTRACT

In this work we explore how lattice gauge theory stands to benefit from new developments in machine learning, and look at two specific examples that illustrate this point. We begin with a brief overview of selected topics in machine learning for those who may be unfamiliar, and provide a simple example that helps to show how these ideas are carried out in practice.

After providing the relevant background information, we then introduce an example of renormalization group (RG) transformations, inspired by the tensor RG, that can be used for arbitrary image sets, and look at applying this idea to equilibrium configurations of the two-dimensional Ising model.

The second main idea presented in this thesis involves using machine learning to improve the efficiency of Markov Chain Monte Carlo (MCMC) methods. Explicitly, we describe a new technique for performing Hamiltonian Monte Carlo (HMC) simulations using an alternative leapfrog integrator that is parameterized by weights in a neural network. This work is based on the L2HMC ('Learning to Hamiltonian Monte Carlo') algorithm introduced in [1].

PUBLIC ABSTRACT

In recent years there has been a growing interest in expanding the scientific applications of machine learning. In this work we will explore two specific examples of how lattice gauge theory stands to benefit from these new developments. Lattice gauge theory is a sub-discipline of high energy physics that makes heavy use of computer simulations to test existing models and generate new results. In order to perform these simulations, spacetime is discretized as a lattice whose behavior can then be controlled using the laws of physics.

We begin by introducing a new technique that is capable of extracting information about a physical system by ‘looking’ at pictures of the system at different temperatures, and provide an analytical framework that explains how this is done behind the scenes. This result demonstrates that **our approach is capable of learning non-trivial information about the system without being told explicitly how to do so**, and without having to provide any information about the underlying physics.

Next, we look at how machine learning can be used to improve the efficiency of the *Hamiltonian Monte Carlo* (HMC) algorithm, a widely used technique in lattice gauge theory for generating *gauge configurations*. These gauge configurations are essentially ‘snapshots’ of the spacetime lattice, and are used to make predictions about quantum theory.

Currently, these configurations are generated via *Hamiltonian Monte Carlo simulations*, an algorithm that can be summarized as follows:

1. Start with a random initial configuration.
2. Propose a new configuration by (approximately) evolving the current state through time¹.
3. Check if this new configuration is better than the previous one. If so, we accept it, otherwise, we retain the current configuration.²

The difference between the current and proposed configurations can be controlled through the *step size*, a parameter that is (typically) fixed for the duration of the simulation. We can improve the

¹Using *Hamilton’s equations*, which describe how a system changes in time.

²Technically, configurations which are ‘worse’ will still be accepted occasionally, but this becomes increasingly unlikely as the drop in ‘quality’ increases.

likelihood of accepting a new configuration by using a smaller step size, but this leads to configurations that are highly correlated with each other, which is undesirable. Alternatively, we can take larger steps to try and reduce correlations, but this causes more of the proposed configurations to be rejected. Immediately we see that computational resources are being wasted each time we propose a new configuration that gets rejected, and is a major source of inefficiency in the algorithm.

The approach presented in this work attempts to combat this issue by using machine learning to reduce the number of wasted calculations. Explicitly, this is done by modifying the equations that govern how our system evolves in time, and then training the algorithm to identify those modifications that ultimately produce ‘better’ configurations. In doing so, we are able to reduce the number of unnecessary calculations (which don’t produce new configurations), thereby improving the efficiency of the algorithm as a whole.

CONTENTS

CONTENTS	vi
LIST OF FIGURES	x
1 INTRODUCTION	1
2 MACHINE LEARNING: AN OVERVIEW	3
2.1 Introduction	3
2.2 Supervised Learning and Gradient Descent	4
2.2.1 Example: Linear Regression	5
2.2.2 Probabilistic Interpretation	7
2.3 Feed-forward Neural Networks	8
2.3.1 Parameter optimization & Backpropagation	10
2.3.1.1 Derivatives of the Error Function	12
2.3.1.2 Output layer	13
2.3.1.3 Hidden layers	13
2.4 Convolutional Neural Networks	14
2.4.1 Local receptive fields	16
2.4.2 Shared weights	18
2.4.3 Pooling Layers	18
2.4.4 Hyperparameters	19
2.5 Conclusion	19

3 UNSUPERVISED LEARNING: ISING WORMS	21
3.1 Introduction	21
3.2 From loops to images	25
3.3 PCA and criticality	27
3.4 TRG coarse-graining	28
3.5 Image coarse-graining	32
3.6 Partial data collapse for blocked images	33
3.7 TRG calculation of $\langle N_b \rangle$	36
3.8 Technical results	39
3.8.1 Loop representation	39
3.8.2 Heat capacity	40
3.8.3 Monte Carlo implementation	41
3.8.4 Tests	41
3.8.5 Conjecture about λ_{\max}	43
3.8.6 Illustration of alternate blockings	47
3.9 Possible applications: From Images to Loops	49
3.10 Conclusions	50
4 MARKOV CHAIN MONTE CARLO (MCMC)	52
4.1 Markov Chains	52
4.1.1 Metropolis-Hastings algorithm	54
4.2 Aside: Bayesian Analysis	55
4.3 Hamiltonian Monte Carlo	56
4.3.1 Hamiltonian Dynamics	57
4.3.1.1 Properties of Hamiltonian Dynamics	58
5 SEMI-SUPERVISED LEARNING: L2HMC	60
5.1 Introduction	60
5.2 Generalizing the Leapfrog Integrator	61

5.2.1	Metropolis-Hastings Accept/Reject	62
5.3	Forward direction ($d = 1$):	64
5.4	Backward direction ($d = -1$):	64
5.5	Determinant of the Jacobian	65
5.6	Network Architecture	65
5.7	Training Procedure	66
5.8	Gaussian Mixture Model	69
5.8.1	Example	70
5.9	2D $U(1)$ Lattice Gauge Theory	70
5.9.1	Modified Network Architecture	74
5.9.2	Annealing Schedule	77
5.9.3	Modified loss function for $U(1)$ gauge model	77
5.9.4	Issues with the Average Plaquette	79
5.10	Conclusion	80
A	APPENDIX: L2HMC SOURCE CODE	82
A.1	README.md	82
A.1.1	l2hmc-qcd	82
A.1.1.1	Overview	82
A.1.1.2	Modified implementation for Lattice Gauge Theory / Lattice QCD models	82
A.1.1.3	Features	83
A.1.1.4	Organization	83
A.1.1.5	Contact	83
A.1.1.6	Citation	83
A.2	l2hmc-qcd/args/gauge_model_args.txt	84
A.3	l2hmc-qcd/gauge_model_main.py	85
A.4	l2hmc-qcd/models/gauge_model.py	93
A.5	l2hmc-qcd/dynamics/gauge_dynamics.py	104

A.6	l2hmc-qcd/lattice/lattice.py	116
A.7	l2hmc-qcd/network/conv_net3d.py	120
A.8	l2hmc-qcd/network/conv_net2d.py	129
A.9	l2hmc-qcd/network/generic_net.py	134
A.10	l2hmc-qcd/network/network_utils.py	137
A.11	l2hmc-qcd/trainers/gauge_model_trainer.py	140
A.12	l2hmc-qcd/runners/gauge_model_runner.py	144
A.13	l2hmc-qcd/loggers/run_logger.py	152
A.14	l2hmc-qcd/loggers/train_logger.py	160
A.15	l2hmc-qcd/plotters/gauge_model_plotter.py	164
A.16	l2hmc-qcd/plotters/leapfrog_plotters.py	170
A.17	l2hmc-qcd/plotters/plot_utils.py	176
A.18	l2hmc-qcd/utils/parse_args.py	179
A.19	l2hmc-qcd/utils/file_io.py	187
A.20	l2hmc-qcd/utils/tf_logging.py	191
A.21	l2hmc-qcd/utils/data_loader.py	194
B	APPENDIX: SYSTEMATIC DEBUGGING RESULTS OF THE AVERAGE PLAQUETTE	197
	REFERENCES	204

LIST OF FIGURES

Figure 2.1: Network diagram for a two-layer fully-connected neural network. The input, hidden, and output variables are shown as nodes while the weight parameters are the links between them.	10
Figure 2.2: Illustration of the local receptive fields in the input image and their corresponding weights in the first hidden layer. Additionally, we can see how these local receptive fields are slid across the input image to construct additional units in the hidden layer.	17
Figure 2.3: Illustration of a 2×2 max pooling layer.	19
Figure 3.1: (a) Picture of an eye with 4096 pixels; (b) black and white version with a graycut at 0.72; (c) boundaries of the black domains.	24
Figure 3.2: (a) Legal worm configuration on an $L \times L$ lattice with periodic boundary conditions and; (b) its equivalent representation as a $2L \times 2L$ black and white pixel image.	27
Figure 3.3: λ_{max} and $\frac{3}{2} \langle \Delta_{N_b}^2 \rangle / V$ (per unit volume) vs. T , illustrating the relation between the eigenvalue corresponding to the first principal component and the logarithmic divergence of the specific heat. The inset shows a qualitative agreement near the critical temperature.	29
Figure 3.4: Illustration of the tensor blocking discussed in the text. Each dot is a tensor at a lattice site with four lines coming out, each representing a tensor index. Lines connecting dots represent tensor contractions.	30
Figure 3.5: (a) T_{1100} vs. $T - T_c^{(2s)}$ for six successive iterations of the blocking transformation, beginning with an initial lattice $L = 64$; (b) T_{1100} vs. $(T - T_c^{(2s)})/L_{eff}$ illustrating the data collapse, where $T_c^{(2s)}$ is the critical temperature of the two state projection, beginning at iteration 0 on an $L = 64$ lattice.	31
Figure 3.6: Illustration of an elementary block in the image consisting of four sites, four internal bonds (red), eight external bonds (green), and four blocked external bonds (blue).	32

Figure 3.16: Example of the different coarse-graining (“blocking”) procedures applied to a sample worm configuration generated at $T = 2.0$. Note that in (3.16(a)) $m \in \{1, 2\}$, and double bonds are represented by blue lines. (3.16(b)), (3.16(c)), illustrate the results of applying different weights to the so-called “double bonds” in the images representing a blocked configuration. Note that in (3.16(b)) $1 + 1 \rightarrow 1$, double bonds are given the same weight as single bonds, and in (3.16(c)) $1 + 1 \rightarrow 2$, double bonds are given twice the weight as single bonds, appearing twice as dark.	48
Figure 3.17: Example of preprocessing steps for converting CIFAR-10 images to ‘worm-like’ images, illustrating the resulting image for different values of the grayscale cutoff. (a) Original image from CIFAR-10 dataset. (b) Image converted to grayscale. (c) Resulting image from cutoff values of 0.25, (d) 0.5, and (e) 0.75.	49
Figure 3.18: $\langle N_b \rangle$ and $\langle \Delta_{N_b}^2 \rangle$ vs. grayscale cutoff value for 500 randomly chosen images from the CIFAR-10 dataset.	50
Figure 4.1: Visualizing HMC for a 1D Gaussian (example from [42]). Each Hamiltonian Markov transition lifts the initial state onto a random level set of the Hamiltonian, $\mathcal{H}^{(-1)}(E)$, which can then be explored with a Hamiltonian trajectory before projecting back down to the target parameter space.	59
Figure 5.1: Example of how the determinant of the Jacobian can deform the energy landscape.	63
Figure 5.2: Illustration showing the generic (fully-connected) network architecture for training S_v , Q_v , and T_v	65
Figure 5.3: Flowchart illustrating the generic fully-connected network architecture including the intermediate variables computed at each hidden layer of the network.	67
Figure 5.4: Comparison of trajectories generated using L2HMC (top), and traditional HMC with $\varepsilon = 0.25$ (middle) and $\varepsilon = 0.5$ (bottom). Note that L2HMC is able to successfully mix between modes, whereas HMC is not.	71
Figure 5.5: Autocorrelation vs. gradient evaluations (i.e. MD steps). Note that L2HMC (blue) has a significantly reduced autocorrelation after the same number of gradient evaluations when compared to either of the two HMC trajectories .	72
Figure 5.6: Illustration of an elementary plaquette on the lattice.	74
Figure 5.7: (left) Example of topological freezing in the 2D $U(1)$ lattice gauge theory, generated from generic HMC sampling for a 16×16 lattice. Note that for the majority of the simulation $Q = -2$, making it virtually impossible to get a reasonable estimate of χ . (right) Topological charge vs. step generated using the trained L2HMC sampler.	75

Figure 5.8: Convolutional structure used for learning localized features of rectangular lattice.	75
Figure 5.9: Illustration taken from TensorBoard showing an overview of the network architecture for VNet. Note that the architecture is identical for XNet.	76
Figure 5.10: Detailed view of additional convolutional structure included to better account for rectangular geometry of lattice inputs.	76
Figure 5.11: (left): Average plaquette $\langle \phi_P \rangle$ vs. step for $L = 8$, $N_{LF} = 7$, and $\alpha_Q = 0.5$. Here the solid red line indicates the true value of the average plaquette (in the infinite volume limit, and is equal to 0.89338... (right): Difference between the observed and expected value of the average plaquette δ_{ϕ_P} vs. step. Note that $\delta_{\phi_P} \neq 0$.	79
Figure 5.12: Same quantities as in Fig 5.11, with $\alpha_Q = 0$. Note that the discrepancy δ_{ϕ_P} is no longer present.	79
Figure B.1: δ_{ϕ_P} vs MD step with $N_{LF} = 5$ for $\beta = 5.0$ (top row) and $\beta = 6.0$ (bottom row). The results from the trained L2HMC (generic HMC) sampler are shown in the left (right) column. As can be seen, the difference δ_{ϕ_P} remains roughly consistent for all values of α_Q .	197
Figure B.2: δ_{ϕ_P} vs MD step with $N_{LF} = 5$ for $\beta = 5.0$ (top row) and $\beta = 6.0$ (bottom row). The results from the trained L2HMC (generic HMC) sampler are shown in the left (right) column. As can be seen, the difference δ_{ϕ_P} remains roughly consistent for all values of α_Q .	198
Figure B.3: δ_{ϕ_P} vs MD step with $N_{LF} = 7$ As can be seen, the difference δ_{ϕ_P} is noticeably larger for $\alpha_Q = 0.75$, but remains roughly consistent for all other values of α_Q .	199
Figure B.4: δ_{ϕ_P} vs. MD step with $N_{LF} = 8$. As can be seen, the difference δ_{ϕ_P} remains roughly consistent for all values of α_Q .	200
Figure B.5: δ_{ϕ_P} vs MD step with $N_{LF} = 9$. As can be seen, the difference δ_{ϕ_P} is largest for $\alpha_Q = 0.0$ and $\alpha_Q = 0.5$.	201
Figure B.6: δ_{ϕ_P} vs MD step with $N_{LF} = 10$. As can be seen, the difference δ_{ϕ_P} is smallest for $\alpha_Q = 0.$, 0.75 and largest for $\alpha_Q = 0.25$, while $\alpha_Q = 0.5$, 2.0, takes on intermediate values.	202
Figure B.7: δ_{ϕ_P} vs MD step with $N_{LF} = 15$. As can be seen, the difference δ_{ϕ_P} varies for different values of α_Q .	203

1 | Introduction

In recent years there has been an explosive growth in the fields of machine learning (ML) and data science. This trend has had a significant impact on a wide variety of industries and has been used to produce incredible new technologies ranging from self-driving cars to personalized recommendation engines and facial recognition software, to name just a few. Just as importantly, (but not as glamorously) many scientific researchers have also begun looking for new ways to apply these ideas to their fields, producing new interdisciplinary efforts and mutually-beneficial collaborations.

One field in particular that stands to benefit from this new direction is high energy physics, which relies heavily on computational science through the use of data analysis and computer simulations. Historically, much of this work been done either through the use of ‘brute-force’ calculations, requiring tremendous computational resources, or ‘by-hand’ which tends to be pain-stakingly slow and is often subject to a whole variety of potential issues (e.g. incomplete information, poor statistics, incorrect models, unjustified approximations, human-error, etc.). Machine learning, on the other hand, aims to sidestep many of these problems entirely by automatically ‘learning’ information from data, and is capable of discovering meaningful patterns that are oftentimes imperceptible to humans. With projects such as the Large Hadron Collider at CERN producing roughly 15 petabytes of data per year, the need for new and better methods for dealing with this data continues to grow.

This thesis is primarily composed of the work completed during my graduate career and includes some relevant background information that may be useful for those looking to learn more information about how ideas from machine learning can be applied to problems in lattice gauge theory and lattice quantum chromodynamics (QCD). In particular, Chapter 2 provides a thorough

background on many of the machine learning tools used throughout the remainder of the thesis and provides concrete examples on their use. For example, topics such as supervised learning, gradient descent / backpropagation, feed-forward and convolutional neural networks are covered.

Chapter 3 covers an example of how unsupervised learning (specifically, principal component analysis) can be applied to extract information about the phase transition of the two-dimensional Ising model. By representing equilibrium configurations of the system as two-dimensional greyscale images, principal component analysis allows us to obtain a direct relationship between the specific heat capacity and the eigenvalue of the dominant principal component. In Sec. 3.4 and Sec. 3.5, a renormalization group transformation is proposed that can be applied to generic sets of images, which when applied to the images under consideration, leads to a finite-size scaling analysis of the critical point.

Chapter 5 describes a new technique for applying supervised learning to help improve the efficiency of Hamiltonian / Hybrid Monte Carlo (HMC) simulations. This new approach is called ‘Learning to Hamiltonian Monte Carlo’ (L2HMC), and is based off of the work described in [1]. In Chapter 4, a brief overview of Markov Chain Monte Carlo (MCMC) methods in general is discussed, and relevant notation introduced. In sections 4.3.1, 5.1, and 5.9, some of the current problems faced by HMC are discussed, particularly within the context of simulations in lattice gauge theory and lattice QCD. The details of the L2HMC algorithm are presented in Sec 5.2, and an example of this algorithm applied to a two-dimensional Gaussian Mixture Model is included in Sec. 5.8. Building on these results, we proceed to look at applying this approach to a two-dimensional $U(1)$ lattice gauge theory, the details of which are laid-out in Sec. 5.9. Finally, in Sec. 5.9.1- 5.9.3 we discuss some of the modifications that were introduced when applying this approach to the gauge model under consideration, and provide insight into why these modifications were both necessary and advantageous.

2 | Machine Learning: An Overview

2.1 Introduction

Broadly speaking, the subject of *machine learning* (ML), refers to the automated detection of meaningful patterns in data [2], and encompasses two major classes of problems: *estimation* and *prediction*. To better understand the difference between the two, we begin with an example. Suppose we observe some measurable quantity x (e.g. measurements of the acceleration of an object in a gravitational field) of the system under consideration that is related to some parameters w (e.g. the gravitational constant) of a model $p(x|w)$ that describes the probability of observing x given w . We can perform an experiment to obtain a dataset x , and use this data to “fit” our model. This procedure of fitting the model typically corresponds to finding the \hat{w} that maximizes the probability of observing x , i.e. $\hat{w} \equiv \text{argmax}_w\{p(x|w)\}$. In this context, *estimation* problems are those concerned with the accuracy of \hat{w} , whereas *prediction* problems are concerned with the ability of the model to predict new observations, i.e. the accuracy of $p(x|\hat{w})$.

For our purposes, we mainly restrict our attention to *prediction*-type problems, which includes the types of problems most frequently associated with machine learning (i.e. regression, classification, etc.). There are three generic approaches commonly used to tackle these types of problems, namely:

1. Supervised learning
2. Unsupervised learning
3. Reinforcement learning

While it is true that most machine learning problems fit nicely into one of these three categories, this is not always the case. Some of the most successful approaches to historically difficult problems have employed ideas from combinations of the three in new and unexpected ways (e.g. *variational auto-encoders* (VAEs) [3], and *generative adversarial networks* (GANs) [4]).

2.2 Supervised Learning and Gradient Descent

Supervised learning is the task of inferring a function from labeled training data. Our training data consists of (input, output) pairs: $\{(x^{(i)}, y^{(i)})\}, i = 1, \dots, n$, with $x^{(i)} \in \mathbb{R}^p$ being the inputs (or features) and $y^{(i)} \in \mathbb{R}^d$, being the outputs (or target) (often $d = 1$) that we wish to predict.¹ For concreteness, suppose we have a collection of n two-dimensional greyscale images, $x^{(1)}, x^{(2)}, \dots, x^{(n)}$, each of which contains p pixels flattened into a vector, and their associated labels $y^{(1)}, y^{(2)}, \dots, y^{(n)}$ matching the type of animal present in the image (e.g. dog, cat, frog, etc). Our goal then is to find a function f that is able to accurately approximate the output $y^{(i)}$ for a given input $x^{(i)}$. If our output is discrete, as in the present case, the problem is said to be a *classification* problem, whereas continuous outputs are referred to as *regression* problems. In order to measure how well our function performs, we often split the available data into two disjoint sets called the training set and test set respectively. The idea is to train our function (using a suitably chosen procedure) on the training data and then use the (previously unseen) data from the test set to evaluate the performance. The ability to accurately predict the desired output for a new, unused input is known as *generalizability* and is an important metric for measuring the quality of a given model.

In order to find this desired function f_w , we must introduce a way to measure the similarity between the expected ($y^{(i)}$) and predicted output $f_w(x^{(i)})$. One way to do this is to introduce a *loss* (or *cost*) function, $\mathcal{L} = \mathcal{L}[f_w(x^{(i)}), y^{(i)}] \equiv \mathcal{L}(w)$, with the idea being that the loss is small when $f_w(x^{(i)}) \simeq y^{(i)}$. In doing so, we are able to improve the quality of our desired function f_w by adjusting the values of w in such a way that the loss function is minimized. One of the most popular techniques for carrying out this optimization is an algorithm known as **gradient descent**. Explicitly, given some initial (often random) values of the parameters w , gradient descent repeatedly updates

¹Here superscript (i) indexes samples in the training set.

their values by “stepping” in the direction of steepest decrease of \mathcal{L} :

$$w_j := w_j - \alpha \frac{\partial}{\partial w_j} \mathcal{L}(w), \quad (2.1)$$

simultaneously for all values of $j = 0, \dots, n$.

Here, α is a *hyperparameter* called the **learning rate**, and it is responsible for determining the “step size” the algorithm takes with each update. It’s important to note that the value of the learning rate must be chosen appropriately since large steps can potentially cause stability issues (resulting from ‘over-shooting’ the minima), whereas taking steps which are too small can dramatically increase the amount of updates necessary to obtain the minimum.

2.2.1 Example: Linear Regression

To make all of these ideas explicit we consider the example of *linear regression*. We begin by assuming that, to good approximation, y can be expressed as a linear function of x :

$$f_w(x) = \sum_{i=0}^n w_i x_i = w^T x \quad (2.2)$$

where we’ve defined $x_0 \equiv 1$ as the intercept (or *bias*) term.

Next, we must choose a loss function \mathcal{L} . For this example, we can choose the least-squares cost function defined to be

$$\mathcal{L}(w) = \frac{1}{2} \sum_{i=1}^m (f_w(x_i) - y_i)^2. \quad (2.3)$$

In order to implement gradient descent using this cost function, we first calculate the partial derivatives for the case of a single training example (x, y) . This gives

$$\frac{\partial}{\partial w_j} \mathcal{L}(w) = \frac{\partial}{\partial w_j} \frac{1}{2} (f_w(x) - y)^2 \quad (2.4)$$

$$= (f_w(x) - y) \cdot \frac{\partial}{\partial w_j} \left(\sum_{i=0}^n w_i x_i - y \right) \quad (2.5)$$

$$= (f_w(x) - y) x_j, \quad (2.6)$$

and so

$$w_j := w_j + \alpha \left(y^{(i)} - f_w(x^{(i)}) \right) x_j^{(i)}. \quad (2.7)$$

To expand this result to the case of multiple training samples, we have two options:

1. Repeat the above update for every j until convergence, i.e. look at every example in the training set for every update step. This approach is known as **batch gradient descent**.
2. Repeatedly run through the training set, and for each individual training example, update the parameters according to the gradient of the error with respect to *that single training example only*. This approach is known as **stochastic gradient descent**.

Note that we can generalize this approach by constructing the *design matrix* X to be the $m \times n^2$ matrix $X_{ij} = x_j^{(i)}$, i.e. each row containing an individual training example. Similarly, we can write the target as an m -dimensional vector containing all the target values from the training set: $\mathbf{y} = [y^{(1)}, y^{(2)}, \dots, y^{(m)}]^T$. In terms of these then, our cost function becomes

$$\mathcal{L}(w) = \frac{1}{2}(Xw - \mathbf{y})^T(Xw - \mathbf{y}) \quad (2.8)$$

Again, computing the required gradient gives

$$\nabla_w \mathcal{L}(w) = \nabla_w \frac{1}{2}(Xw - \mathbf{y})^T(Xw - \mathbf{y}) \quad (2.9)$$

$$= \frac{1}{2} \nabla_w (w^T X^T X w - w^T X^T \mathbf{y} - \mathbf{y}^T X w + \mathbf{y}^T \mathbf{y}) \quad (2.10)$$

$$= \frac{1}{2} \nabla_w \text{tr} (w^T X^T X w - w^T X^T \mathbf{y} - \mathbf{y}^T X w + \mathbf{y}^T \mathbf{y}) \quad (2.11)$$

$$= \frac{1}{2} \nabla_w (\text{tr} w^T X^T X w - 2 \text{tr} \mathbf{y}^T X w) \quad (2.12)$$

$$= \frac{1}{2} (X^T X w + X^T X w - 2 X^T \mathbf{y}) \quad (2.13)$$

$$= X^T X w - X^T \mathbf{y} \quad (2.14)$$

Setting this equal to zero and solving gives the **normal equations**

$$X^T X w = X^T \mathbf{y}, \quad (2.15)$$

²technically $m \times n + 1$, accounting for the intercept term

from which we get

$$\hat{w} = (X^T X)^{-1} X^T \mathbf{y}. \quad (2.16)$$

2.2.2 Probabilistic Interpretation

To motivate the reasoning behind our choice of the cost function for linear regression, we adopt a probability approach. Assume

$$y^{(i)} = w^T x^{(i)} + \varepsilon^{(i)} \quad (2.17)$$

with $\varepsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$ is an error term that captures either unmodeled effects (e.g. caused by features that are not accounted for) or random noise. Assume $\varepsilon^{(i)}$ are i. i. d. The probability density is given by

$$p(\varepsilon^{(i)}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\varepsilon^{(i)})^2}{2\sigma^2}\right) \quad (2.18)$$

This implies that

$$p(y^{(i)}|x^{(i)}; w) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - w^T x^{(i)})^2}{2\sigma^2}\right) \quad (2.19)$$

and we say that $p(y^{(i)}|x^{(i)}; w)$ is the distribution of $y^{(i)}$ given $x^{(i)}$ and parameterized by w , or equivalently, $y^{(i)}|x^{(i)}; w \sim \mathcal{N}(w^T x^{(i)}; \sigma^2)$. The question we have now is: given the design matrix X and the weights w , what is the distribution of the $y^{(i)}$'s? The probability of the data is given by $p(\mathbf{y}|X; w)$, which is typically viewed as a function of \mathbf{y} (and perhaps X) for a fixed value of w . When we want to explicitly view this as a function of w , we will instead call it the **likelihood function**,

$$L(w) = L(w; X, \mathbf{y}) \equiv p(\mathbf{y}|X; w) \quad (2.20)$$

By the independence assumption on the $\varepsilon^{(i)}$'s (and consequently, also the $y^{(i)}$'s and the $x^{(i)}$'s) this can also be written as

$$L(w) = \prod_{i=1}^m p(y^{(i)}|x^{(i)}; w) \quad (2.21)$$

$$= \prod_{i=1}^m \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - w^T x^{(i)})^2}{2\sigma^2}\right) \quad (2.22)$$

By the principle of **maximum likelihood** [5], we should choose w so as to make the data as high probability as possible, i.e. we should choose w to maximize $L(w)$. Instead of maximizing $L(w)$ directly, we can also maximize any strictly increasing function of $L(w)$. In particular, we choose to maximize the **log likelihood** $\ell(w)$:

$$\ell(w) \equiv \log L(w) \quad (2.23)$$

$$= \log \prod_{i=1}^m \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - w^T x^{(i)})^2}{2\sigma^2}\right) \quad (2.24)$$

$$= \sum_{i=1}^m \log \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - w^T x^{(i)})^2}{2\sigma^2}\right) \quad (2.25)$$

$$= m \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{\sigma^2} \cdot \frac{1}{2} \sum_{i=1}^m (y^{(i)} - w^T x^{(i)})^2. \quad (2.26)$$

Hence, maximizing $\ell(w)$ gives the same answer as minimizing

$$\frac{1}{2} \sum_{i=1}^m (y^{(i)} - w^T x^{(i)})^2 \quad (2.27)$$

which is equivalent to $\mathcal{L}(w)$, the original loss function from our least squares example.

2.3 Feed-forward Neural Networks

We can construct a Neural Network model as a series of functional transformations. First, we construct M linear combinations of the input variables x_1, \dots, x_D in the form

$$a_j = \sum_{i=1}^D w_{ji}^{(1)} x_i + w_{j0}^{(1)} \quad (2.28)$$

where $j = 1, \dots, M$ and now here the superscript (1) indicates the ‘layer’ of the network. The quantities a_j are known as ‘activations’, each of which is transformed via a differentiable, nonlinear activation function $h(\cdot)$ to give

$$z_j = h(a_j) \quad (2.29)$$

which are often referred to as ‘hidden units’ or ‘nodes’ in the context of neural networks. These nonlinear functions are often chosen to be sigmoidal functions such as the tanh function or the rectified linear unit (ReLU), defined as

$$\text{ReLU}(x) \equiv \max(0, x). \quad (2.30)$$

Following Eq. 2.29, these values are again linearly combined to give *output unit activations*

$$a_k = \sum_{j=1}^M w_{kj}^{(2)} z_j + w_{k0}^{(2)} \quad (2.31)$$

where $k = 1, \dots, K$, and K is the total number of outputs. Depending on the context of the problem, the output unit activations may be transformed using an appropriate activation function $\sigma(\cdot)$ to give a set of network outputs $y_k = \sigma(a_k)$. For example, multi-class classification problems often use the standard (unit) softmax activation function $\sigma : \mathbb{R}^K \rightarrow \mathbb{R}^K$ given by

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \quad \text{for } i = 1, \dots, K \text{ and } \mathbf{z} = (z_1, \dots, z_K) \in \mathbb{R}^K \quad (2.32)$$

which takes as input a vector of K real numbers and normalizes it into a probability distribution consisting of K probabilities. A visualization of this network structure is shown in Fig 2.1. Often the bias parameters are absorbed into the set of weight parameters by defining an additional input variable x_0 whose value is fixed at $x_0 = 1$ so that Eq 2.28 becomes

$$a_j = \sum_{i=0}^D w_{ji}^{(1)} x_i, \quad (2.33)$$

and similarly for the second layer. Written like this then, we can write the overall network function as

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma \left(\sum_{j=0}^M w_{kj}^{(2)} h \left(\sum_{i=0}^D w_{ji}^{(1)} x_i \right) \right) \quad (2.34)$$

Note that this structure can be easily generalized an arbitrary depth by successively ‘stacking’ these hidden layers.

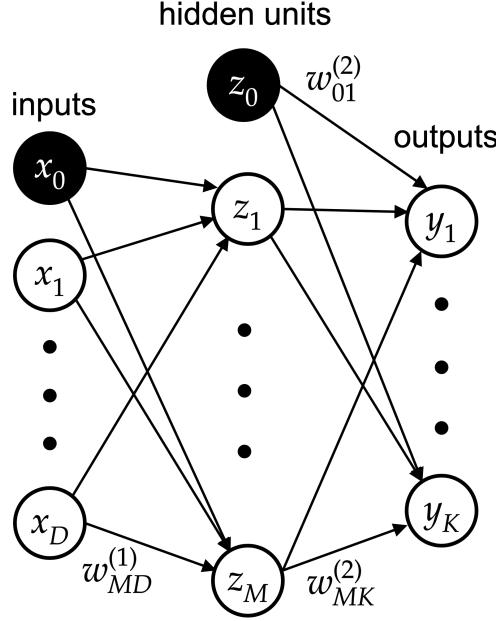


Figure 2.1: Network diagram for a two-layer fully-connected neural network. The input, hidden, and output variables are shown as nodes while the weight parameters are the links between them.

2.3.1 Parameter optimization & Backpropagation

Given a set of input vectors $\{\mathbf{x}_n\}$ for $n = 1, \dots, N$, together with a corresponding set of target vectors $\{\mathbf{t}_n\}$, our goal is to find the values \mathbf{w}^* that minimize some appropriately chosen loss function $\mathcal{L}(\mathbf{w})$ that measures the ‘closeness’ between the desired outputs \mathbf{t}_n and the calculated outputs produced by our network $\mathbf{y}_n = y(\mathbf{x}_n, \mathbf{w})$. Similar to the gradient descent algorithm discussed in Sec. 2.2, training a neural network with gradient descent requires the calculation of the gradient of the loss function \mathcal{L} with respect to the weights w_{ij}^k and biases, collectively denoted θ . Then, according to the learning rate α , each iteration of gradient descent updates the weights according to

$$\theta^{t+1} = \theta^t - \alpha \frac{\partial \mathcal{L}}{\partial \theta} \quad (2.35)$$

where θ^t denotes the parameters of the neural network at iteration t of gradient descent. Since the nodes in the hidden layers have no target output, we can’t simply define an error function that is specific to that node. Instead, any error function for that node will be dependent on the values of the parameters in the previous layers and following layers. This coupling of parameters between layers often leads to complicated expressions for the gradients and is often slow in practice. In

order to address these issues, an alternative approach is frequently used which relies upon the **backpropagation** algorithm, outlined below. For consistency, we adopt the following notations:

- $X = \{(\mathbf{x}_1, \mathbf{t}_1), \dots, (\mathbf{x}_N, \mathbf{t}_N)\}$: the dataset of N input-output pairs where \mathbf{x}_i is the input and \mathbf{t}_i is the desired output of the network on input \mathbf{x}_i
- w_{ij}^k : weight for node j in layer k for incoming node i
- $w_{0i}^k = b_i^k$: bias for node i in layer k with a fixed output of $z_0^{k-1} = 1$ for node 0 in layer $k-1$.
- a_i^k activation (i.e. the product sum plus bias) for node i in layer k
- z_i^k : output for node i in layer k
- r_k : the number of nodes in hidden layer k
- h : activation function for the hidden layer nodes
- h_z : activation function for the output layer nodes

For simplicity, we suppose that the our neural network produces a single, scalar valued output y for a given vector-valued input \mathbf{x} , and choose a loss function given by the mean-squared error, i.e.

$$\mathcal{L}(w) = \frac{1}{2N} \sum_{i=1}^N (y_i - t_i)^2 \quad (2.36)$$

Backpropagation then attempts to minimize this loss function with respect to the neural network's weights by calculating, for each weight w_{ij}^k , the value of $\frac{\partial \mathcal{L}}{\partial w_{ij}^k}$. Since the error function can be decomposed into a sum over individual loss terms for each individual input-output pair, the derivative can be calculated with respect to each input-output pair individually and then combined at the end (due to the fact that the derivative of a sum of functions is the sum of the derivatives of each function)

$$\frac{\partial \mathcal{L}(X, \theta)}{\partial w_{ij}^k} = \frac{1}{N} \sum_{d=1}^N \frac{\partial}{\partial w_{ij}^k} \left(\frac{1}{2} (y_d - t_d)^2 \right) \quad (2.37)$$

$$= \frac{1}{N} \sum_{d=1}^N \frac{\partial \mathcal{L}_d}{\partial w_{ij}^k}. \quad (2.38)$$

Because of this, we can restrict our attention to only one input-output pair and from this the general form for all input-output pairs can be generated by combining individual gradients. Our loss function then simplifies to

$$\mathcal{L} = \frac{1}{2}(y - t)^2 \quad (2.39)$$

2.3.1.1 Derivatives of the Error Function

We begin by applying the chain rule to the loss function partial derivative

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^k} = \frac{\partial \mathcal{L}}{\partial a_j^k} \frac{\partial a_j^k}{\partial w_{ij}^k} \quad (2.40)$$

The first term in the product is often referred to as the *error* and denoted by $\delta_j^k \equiv \partial \mathcal{L} / \partial a_j^k$. The second term can be calculated from the fact that

$$a_j^k = \sum_{j=0}^{r_{k-1}} w_{ij}^k z_j^{k-1} \quad (2.41)$$

to give

$$\frac{\partial a_j^k}{\partial w_{ij}^k} = \frac{\partial}{\partial w_{ij}^k} \left(\sum_{l=0}^{r_{k-1}} w_{ij}^k z_l^{k-1} \right) \quad (2.42)$$

$$= z_i^{k-1}. \quad (2.43)$$

Thus, the partial derivative of the loss function \mathcal{L} with respect to a weight w_{ij}^k is given by

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^k} = \delta_j^k z_i^{k-1}, \quad (2.44)$$

i.e. the partial derivative of a weight is a product of the error term δ_j^k at node j in layer k , and the output z_i^{k-1} of node i in layer $k - 1$. Since the error term δ_j^k depends on both the loss function, and as we will show, the values of the error terms in the next layer, the computation of these terms proceeds backwards from the output layer down to the input layer which is where the term *backpropagation* (of errors) gets its name.

2.3.1.2 Output layer

Starting from the final layer, backpropagation calculates the value³ δ_1^m . Expressing the loss function in terms of the value a_1^m (since δ_1^m is a partial derivative with respect to a_1^m) gives

$$\mathcal{L} = \frac{1}{2}(t - y)^2 = \frac{1}{2}(h_z(a_1^m) - t)^2 \quad (2.45)$$

where again $h_z(x)$ is the activation function for the output layer. Applying the partial derivative and using the chain rule gives

$$\delta_1^m = (h_z(a_1^m) - t)h'_z(a_1^m) = (y - t)h'_z(a_1^m) \quad (2.46)$$

and putting it all together, the partial derivative of the loss function \mathcal{L} with respect to a weight in the final layer w_{i1}^m is

$$\frac{\partial E}{\partial w_{i1}^m} = \delta_1^m z_i^{m-1} = (y - t)h'_z(a_1^m)z_i^{m-1}. \quad (2.47)$$

2.3.1.3 Hidden layers

Now, we calculate the partial derivatives for the hidden layers, again using the multivariate chain rule. Note that for $1 \leq k < m$, the error term δ_j^k is:

$$\delta_j^k = \frac{\partial \mathcal{L}}{\partial a_j^k} = \sum_{l=1}^{r_{k+1}} \frac{\partial \mathcal{L}}{\partial a_l^{k+1}} \frac{\partial a_l^{k+1}}{\partial a_j^k}, \quad (2.48)$$

where l ranges from 1 to r_{k+1} (the number of nodes in the next layer). Plugging in the error term δ_l^{k+1} gives

$$\delta_j^k = \sum_{l=1}^{r_{k+1}} \delta_l^{k+1} \frac{\partial a_l^{k+1}}{\partial a_j^k}. \quad (2.49)$$

Recalling that

$$a_l^{k+1} = \sum_{j=1}^{r_k} w_{jl}^{k+1} h(a_j^k) \quad (2.50)$$

³recall we are considering a one-output neural network, hence the subscript 1 and not j

where $h(x)$ is the activation function in the hidden layers, we are left with

$$\frac{\partial a_l^{k+1}}{\partial a_j^k} = w_{jl}^{k+1} h'(a_j^k). \quad (2.51)$$

Plugging this into the above equation gives a final equation for the error term δ_j^k in the hidden layers, a result known as the *backpropagation formula*:

$$\delta_j^k = \sum_{l=1}^{r_{k+1}} \delta_l^{k+1} w_{jl}^{k+1} h'(a_j^k) = h'(a_j^k) \sum_{l=1}^{r_{k+1}} w_{jl}^{k+1} \delta_l^{k+1} \quad (2.52)$$

Putting this all together we get the partial derivative of the loss function \mathcal{L} with respect to a weight in the hidden layers w_{ij}^k for $1 \leq k < m$:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^k} = \delta_j^k z_i^{k-1} = h'(a_j^k) z_i^{k-1} \sum_{l=1}^{r_{k+1}} w_{jl}^{k+1} \delta_l^{k+1}. \quad (2.53)$$

For completeness, we include the complete algorithm in Alg. 1.

2.4 Convolutional Neural Networks

Convolutional neural networks (CNNs or ConvNets) is a generic term used to describe any neural network whose architecture includes convolutional layers. Consequently, ConvNets are similar in many ways to the fully-connected feed-forward layers previously discussed, and are almost always used in conjunction with some combination of fully-connected layers. An example architecture can be found in Fig. 5.8. The difference now is that we expect our input data to have some rectangular structure (e.g. two-dimensional images), which are often represented as a three-dimensional volume of shape height \times width \times depth where height and width are, as we would expect, the height and width of the input image, and depth usually refers to the number of *color channels* (i.e. RGB) of the image. For example, if we are dealing with a typical two-dimensional RGB image, then depth = 3, which we could represent as a three-dimensional volume of height \times width \times [red, green, blue] pixels.

Additionally, in contrast to fully-connected networks, ConvNets combine three architectural

Algorithm 1: Backpropagation Algorithm

input : A data set $X = \{(\mathbf{x}_1, t_1), \dots, (\mathbf{x}_N, t_N)\}$ consisting of N input-output pairs, the learning rate α , the loss function \mathcal{L} , and the activation function h

1. **Forward pass:** For each input-output pair (\mathbf{x}_d, t_d) , store the results y_d , a_j^k , and z_j^k for each node j in layer k by proceeding from the input layer to the output layer m .
2. **Backward pass:** For each input-output pair (\mathbf{x}_d, t_d) , store the results $\partial \mathcal{L}_d / \partial w_{ij}^k$ for each weight w_{ij}^k connecting node i in layer $k - 1$ to node j in layer k by proceeding from layer m , the output layer, to the input layer.
 - Evaluate the error term for the final layer δ_1^m by using Eq. 2.47.
 - Backpropagate the error terms for the hidden layers δ_j^k , working backwards from the final hidden layer $k = m - 1$ by repeatedly using Eq. 2.52.
 - Evaluate the partial derivatives of the individual error \mathcal{L}_d with respect to w_{ij}^k by using Eq. 2.53.
3. **Combine individual gradients:** for each input-output pair $\frac{\partial \mathcal{L}_n}{\partial w_{ij}^k}$ to get the total gradient $\frac{\partial \mathcal{L}(X, \theta)}{\partial w_{ij}^k}$ for the entire set of input-output pairs using

$$\frac{\partial \mathcal{L}(X, \theta)}{\partial w_{ij}^k} = \frac{1}{N} \sum_{d=1}^N \frac{\partial}{\partial w_{ij}^k} \left(\frac{1}{2} (y_n - t_n)^2 \right) = \frac{1}{N} \sum_{d=1}^N \frac{\partial \mathcal{L}_d}{\partial w_{ij}^k}$$

ideas to ensure some degree of shift and distortion invariance [6]:

- *Local receptive fields*
- *Shared weights* (or weight replication)
- Spatial or temporal subsampling (or *pooling*)

2.4.1 Local receptive fields

When dealing with high-dimensional inputs such as images, it is impractical to fully-connect each neuron to all neurons in the previous volume because such a network architecture does not take into account the spatial structure of the data. Convolutional networks exploit spatially local correlation by enforcing a sparse local connectivity pattern between neurons of adjacent layers: each neuron is connected to only a small region of the input volume. The spatial extent of this connectivity is a hyperparameter, known as the *local receptive field*. The connections are local in space (along the height and width), but always extend along the entire depth of the input volume.

We begin with an example. Suppose we take as input a 16×16 square of pixels, corresponding to an image. Proceeding as before, we connect the input pixels to a layer of hidden units, with the exception that now we only make connections in small, localized regions of the input image. Explicitly, each neuron in the first hidden layer will be connected to a small region of the input units, say, for example, a 4×4 region, corresponding to 16 input pixels. The region in the input image is known as the *local receptive field* for the hidden unit. This local receptive field can be understood as a little window on the input pixels, with each connection learning a weight. During the forward pass through our network, each filter is convolved across the height and width of the input volume (as illustrated in Fig. 2.2) computing the dot product between the entries of the filter and the input, producing a two-dimensional activation map of that filter. As a result, the network learns filters that activate when it detects some specific type of feature at some spatial position in the input. Stacking these activation maps for all filters along the depth dimension forms the full output volume of the convolution layer. All together, there are three hyperparameters that control the size of the output volume of the convolutional layer: the *depth*, *stride length*, and *zero-padding*. The *depth* of the output volume controls the number of neurons in a layer that connect to the same

local receptive field of the input volume. These neurons learn to activate for different features in the input, e.g. various oriented edges or blobs of color. The *stride length*, S (chosen as $S = 1$ in Fig. 2.2), controls how depth columns around the spatial dimensions (height and width) are allocated. This can be understood as the distance by which the local receptive field is moved to construct a new hidden unit. Note that in our example we chose $S = 1$, but it is not uncommon to use different values (typically $S = 1, 2$), depending on the particulars of the problem at hand. Finally, it is sometimes convenient to pad the input with zeros on the border of the input volume. This hyperparameter is known as the *zero-padding*, and denoted by P . The spatial size of the output volume can be computed as a function of the input volume size W , the spatial size of the local receptive field (also known as the *kernel* or *filter size*), K , the stride length with which they are applied S , and the amount of zero-padding P used on the border. The equation for calculating how many neurons ‘fit’ inside a given volume is then given by

$$\frac{W - K + 2P}{S} + 1 \quad (2.54)$$

If this number is not an integer, then the strides are incorrect and the neurons cannot be tiled to fit across the input volume in a symmetric manner. In general, setting the zero padding to be $P = (K - 1)/2$ when the stride is $S = 1$ ensures that the input volume and output volume will have the same size spatially.

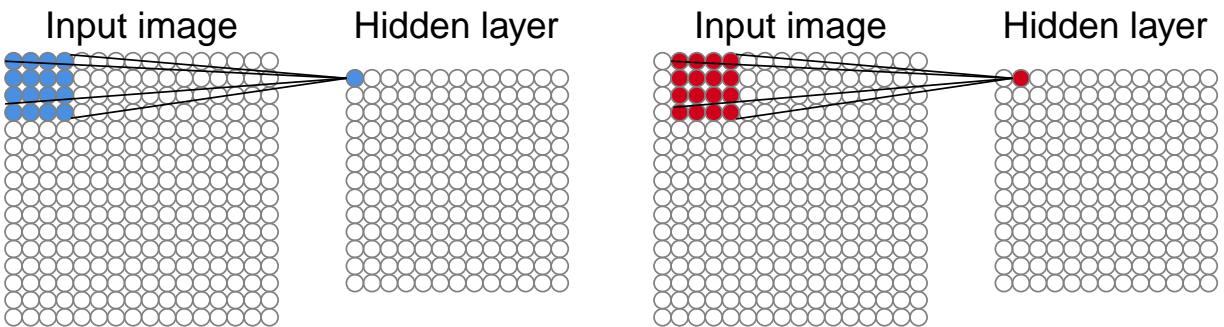


Figure 2.2: Illustration of the local receptive fields in the input image and their corresponding weights in the first hidden layer. Additionally, we can see how these local receptive fields are slid across the input image to construct additional units in the hidden layer.

2.4.2 Shared weights

A parameter sharing scheme is used in convolutional layers to control the number of free parameters. This scheme relies on the reasonable assumption that if a patch feature is useful to compute at some spatial position, then it should also be useful to compute at other positions. Equivalently, if we denote a single two-dimensional slice of depth as a *depth slice*, we constrain the neurons in each depth slice to use the same weights and biases. Since all neurons in a single depth slice share the same parameters, the forward pass in each depth slice of the convolutional layer can be computed as a convolution of the neuron's weights with the input volume. Because of this, the sets of weights which are convolved with the input are often referred to as a *filter* or *kernel*. The result of this convolution is then called the *activation map*, and the set of activation maps for each different filter are stacked together along the depth dimension to produce the output volume. This idea of parameter sharing helps contribute to the translation invariance of the CNN architecture.

2.4.3 Pooling Layers

Another important concept of CNNs is pooling, which is a form of non-linear down-sampling. This concept is implemented in what we call *pooling layers*, which are usually used immediately following convolutional layers. The pooling layer serves to progressively reduce the spatial size of the representation, to reduce the number of parameters, memory footprint and amount of computation in the network, and hence to also control overfitting [7, 8]. In doing so, the pooling operation provides another form of translation invariance. What these layers do is simplify the information in the output from the convolutional layer by summarizing some region of neurons in the previous layer. The most common form is a pooling layer with filters of size 2×2 applied with a stride of 2 downsamples at every depth slice in the input by 2 along both the height and width, discarding 75% of the activations. One common type of pooling layer is known as *max pooling*, in which a pooling unit simply outputs the maximum activation in the 2×2 input region. This behavior can be seen in Fig. 2.3.

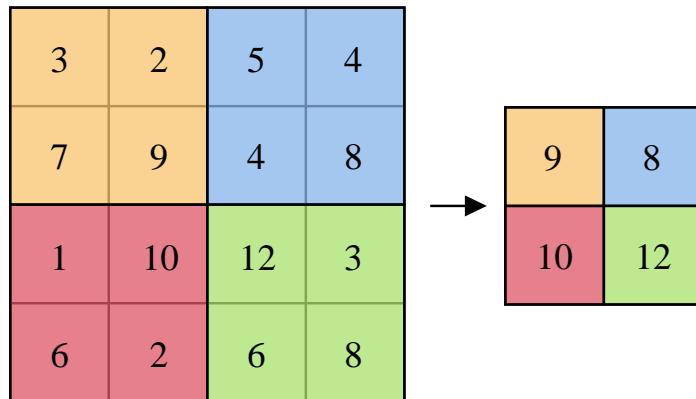


Figure 2.3: Illustration of a 2×2 max pooling layer.

2.4.4 Hyperparameters

Since CNNs use more hyperparameters than typical feed-forward fully-connected neural networks, it is often difficult to determine what values are appropriate for the problem at hand. One idea to keep in mind is that since the size of the feature maps decreases with increasing network depth, layers closer to the input layer will tend to have fewer filters while deeper layers will usually have more. The number of feature maps directly controls the layers' capacity and depends on the number of available examples and the complexity of the problem being studied. It is common to use small filters (e.g. 3×3 or at most 5×5) using a stride of $S = 1$, and padding the input volume with zeros in such a way that the convolutional layer does not alter the spatial dimensions of the input. Since the pooling layers are in charge of downsampling the spatial dimensions of the input, a common setting is to use max-pooling with 2×2 receptive fields (i.e. $K = 2$), and with a stride of $S = 2$. With this choice of hyperparameters, exactly 75% of the activations in an input volume are discarded (since we downsample by 2 in both height and width).

2.5 Conclusion

In this chapter we have introduced the concept of machine learning and provided a broad overview of some of the topics that will be used in the following chapters. The information provided here is by no means exhaustive, and for further reference I highly suggest the excellent book by Bishop [9].

Moreover, for those interested, there is no shortage of online materials available and a quick google search will provide an endless collection of useful resources. In many ways, we have just barely scratched the surface of machine learning and it would be nearly impossible to include everything in this work. The idea of applying ideas from machine learning to problems in physics has only just recently begun to be explored and it is quickly becoming a very active area of research.

3 | Unsupervised Learning: Ising Worms

3.1 Introduction

Machine learning (ML) is a general framework for recognizing patterns in data without detailed human elaboration of the rules for doing so. As an example, a very general function, with many parameters (for example, thousands or millions) can be optimized on a training set, where the desired output is known. The problem is typically nonconvex and plagued by over-fitting problems, and so advanced methods are necessary in order to get reliable answers. One tool that has been exploited is principal component analysis (PCA), which reduces the dimensionality of the data to the most important “directions.” Immediately the practitioner of renormalization group (RG) methods recognizes an analogy, since the RG techniques are also supposed to identify the most important directions in an enlarged space of Hamiltonians. One of the motivations of the present research is to make this analogy more concrete.

A number of papers [10–12] attempt to draw a connection between deep learning and the RG as it appears in physics. However, the analogies between RG flow and depth in a neural network would be strengthened if one could determine conditions under which fixed points can be identified. It would be helpful to show more explicitly how passing from one level to another in a neural network genuinely translates to a renormalization group transformation. There have been steps in the direction of making a full connection. For instance in [11], the principle of *causal influence* is emphasized. That is, when descending in depth, only neighboring nodes should influence the outcome of a lower level node. We have also implemented this in a simple training scheme in earlier work [13]. It can be called “cheap learning” because far fewer variational parameters are involved,

due to the constraints of locality. In [12] it is emphasized that deep neural networks outperform shallower networks for reasons which may ultimately be understood in terms of the power of the renormalization group. Other topics related to machine learning, such as principal component analysis [14] have been previously interpreted in terms of the renormalization group (in this case momentum shells). Machine learning has also been used to identify phase transitions in numerical simulations [15–18]. RG transformations are usually defined in a space of couplings/Hamiltonians, but typically, it is not possible to write down Hamiltonians directly associated with image sets. In this article we propose RG transformations that can be applied to a specific set of images but which could be generalized for other image sets, and can also be understood analytically without any graphical representation. We use the well-studied example of the two-dimensional Ising model on a square lattice. The spin configurations generated with importance sampling provide images with black and white pixels. They have features that can be used to attempt to recognize the temperature used to generate them. However, constructing blocked Hamiltonians in configuration space is a difficult task which involves approximations that are difficult to improve. In other words, it is very difficult to explicitly construct the exact RG transformation mapping the original couplings among the Ising spins into coarse grained ones.

A better control on the RG transformation can be achieved by using the tensor renormalization group (TRG) method [19–25]. The starting point for this reformulation is the character expansion of the Boltzmann weights which is also used in the duality transformation [26]. This leads to an exact expression of the partition function as a sum over closed paths which can be generated with importance sampling using the worm algorithm [27] and then pixelated. These samples will be our sets of images indexed by the temperature used to generate them. The procedure is reviewed in Sec. 3.2.

The goal of a RG analysis is to study systems with large correlation lengths in lattice spacing units and iteratively replace them by coarser ones with a larger effective lattice spacing. This process is useful if we can tune a parameter such as the temperature towards its critical value. Typical image sets such as the MNIST data can be thought as “far from criticality” and the use of RG methods for such a data set may be of limited interest [13]. Criticality may sometimes refer to the choice of parameters used in data analysis [28].

The PCA is a standard method to analyze sets of images. In configuration space, the PCA analysis is identical to the study of the spin-spin correlation matrix. In particular, the largest eigenvalue λ_{\max} is directly connected to the magnetic susceptibility which diverges at criticality [16]. In the loop representation (worms), we will show that λ_{\max} diverges logarithmically at criticality with a constant of proportionality which can be estimated quite precisely ($3/\pi$). This is explained in Sec. 3.3. More generally, it seems reasonable to identify the criticality with the divergence of λ_{\max} .

The advantage of rewriting the high-temperature expansion in terms of tensors is that it allows a very simple blocking (coarse-graining) procedure where a group of sites is replaced by a single site. In the TRG approach the blocking procedure is local. This leads to simple and exact coarse-graining formulas because we can separate the links into two categories: those links that are inside the blocks and integrated over, and those outside the blocks which are kept fixed and communicate between the blocks [22]. The main goal of this chapter is to relate blocking procedures that can be applied to sets of pixelated images, to approximate TRG transformations. A short summary of the TRG procedure is given in Sec. 3.4.

Having defined criticality, the next step is to define a RG transformation for sets of “legal” loop configurations, also called “worm configurations” later, sampled at various temperatures. In Sec. 3.5, we propose a family of transformations which replaces two parallel links in a block by a single link carrying a specific value x . We call this procedure $1 + 1 \rightarrow x$ hereafter. In the case $1 + 1 \rightarrow 0$, the blocked images follow the same rules (for legal configurations) as the original ones. There is a clear analogy with the 2-state approximation of the TRG method. In the 2-state TRG approximation, the average fraction of occupied links shows a characteristic crossing at a critical point and a collapse when the distance to the critical point is appropriately rescaled at each iteration. The average fraction of occupied links in the blocked worm configurations (with $1 + 1 \rightarrow 0$) shows a somewhat similar behavior in the low temperature phase. However, on the high-temperature side, we observe a “merging” rather than a crossing. In Sec. 3.6, we provide explanations for the similarities and differences between the two procedures.

In Sec. 3.7 we discuss an approximate 2-state TRG method to calculate the average number of bonds through several iterations. The worm configurations can be directly connected to spin configurations using duality [26]: they are the boundary of the positive spin islands. This sug-

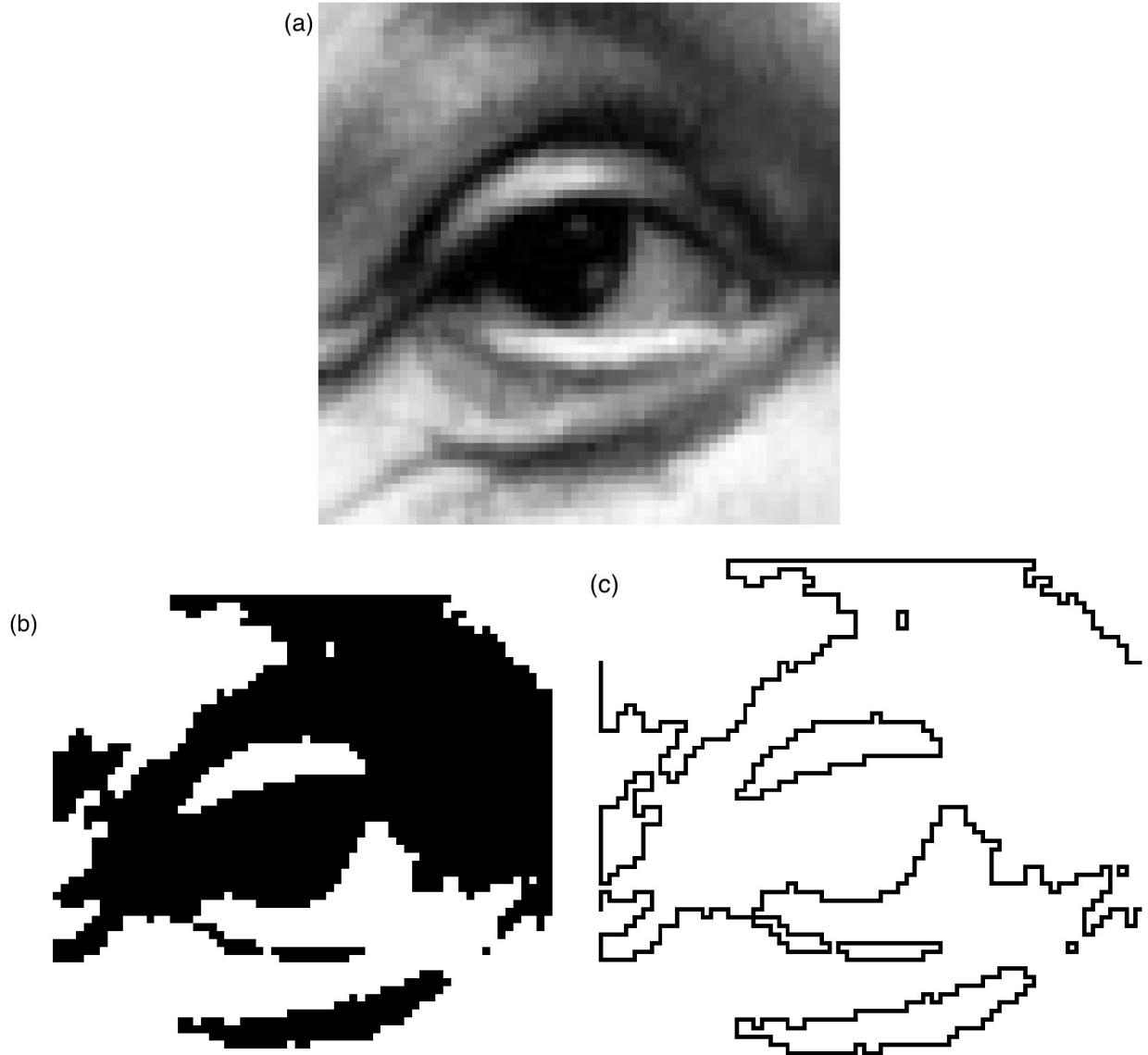


Figure 3.1: (a) Picture of an eye with 4096 pixels; (b) black and white version with a graycut at 0.72; (c) boundaries of the black domains.

gests that the methods discussed here could be applied to generic images. Boundaries of generic grayscale pictures can be defined by converting the picture to black and white pixels. A grayscale picture with gray values between 0 and 1 can be converted into an Ising spin configuration, by introducing a “graycut” below which the value is converted to 0 (spin down) and above which the value is converted to 1 (spin up). It is then possible to construct the boundaries of the spin up domains. This is illustrated in Fig. 3.1. Possible applications are briefly discussed in the Conclusions and illustrated with the CIFAR database in Section 3.9.

3.2 From loops to images

In the following we consider the two-dimensional Ising model with spins $\sigma_i = \pm 1$ on a square lattice. The partition function reads

$$Z = \sum_{\{\sigma_i\}} e^{\beta \sum_{\langle i,j \rangle} \sigma_i \sigma_j}, \quad (3.1)$$

where $\langle i, j \rangle$ denotes nearest neighbor sites on the square lattice. In some occasions we will use the notation $T = 1/\beta$ for the temperature. The partition function can be rewritten by using the character expansion [26]

$$\exp(\beta\sigma) = \cosh(\beta) + \sigma \sinh(\beta), \quad (3.2)$$

and integrating over the spins. Factoring out the $\cosh(\beta)$, each link can carry a weight 1 when unoccupied or $t \equiv \tanh(\beta)$ when occupied. The integration over the spins guarantees that an even number of occupied links is coming out of each site [26]. The set of occupied links then form a “legal graph” with N_b occupied links. The partition function can then be written as sum over such legal graphs. If $\mathcal{N}(N_b)$ denotes the number of legal graphs with N_b links we can write:

$$Z = 2^V (\cosh(\beta))^{2V} \sum_{N_b} t^{N_b} \mathcal{N}(N_b). \quad (3.3)$$

Using the fact that $\tanh(\beta) = \exp(-2\tilde{\beta})$, with $\tilde{\beta}$ the inverse dual temperature, Eq. (3.3) has the same form as a spectral decomposition using a density of states and a Boltzmann weight (with $2N_b$

playing the role of the energy). Details of this reformulation can be found in Section 3.8.1.

As shown in Section 3.8.2, we can use derivatives of the logarithm of the partition function to relate $\langle N_b \rangle$ to the average energy, and the bond number fluctuations,

$$\langle \Delta_{N_b}^2 \rangle \equiv \langle (N_b - \langle N_b \rangle)^2 \rangle, \quad (3.4)$$

to the specific heat per site. From the logarithmic singularity of the specific heat we find that

$$\langle \Delta_{N_b}^2 \rangle / V = -\frac{2}{\pi} \ln(|T - T_c|) + \text{regular}. \quad (3.5)$$

In the following we use interchangeably the “bond” terminology, for instance in N_b as in [27] and the link terminology more common in the lattice gauge theory context. In all our numerical simulations we use periodic boundary conditions which guarantees translation invariance.

We will show in Sec. 3.4 that the new form of the partition function in Eq. (3.3) can also be written in an equivalent way as a sum of products of tensors with four indices contracted along the links of the lattice.

The contributions to Eq. (3.3) can be sampled using a worm algorithm [27] outlined in Section 3.8.3. Using this algorithm, we generated multiple configurations at each temperature ($N_{\text{configs}} \approx 10,000$) which are then used for averaging. For example, we can calculate the average number of occupied bonds at a particular temperature by averaging over all configurations.

Using a legal graph (worm configuration), we can construct an image by introducing a lattice of $2L \times 2L$ pixels with a size of one half lattice spacing. One quarter of these pixels are attached to the sites, one quarter to the horizontal links and one quarter to the vertical links. The remaining quarter are in the middle of the plaquettes and always white. In this representation, each site, link, and plaquette are designated an individual pixel, where occupied links and their respective endpoints are colored black. An example of this representation is shown in Fig. 3.2. We can then flatten each of these images into a vector $\mathbf{v} \in \mathbb{R}^{4V}$, with $v_i \in \{0, 1\}$. This allows us to write the

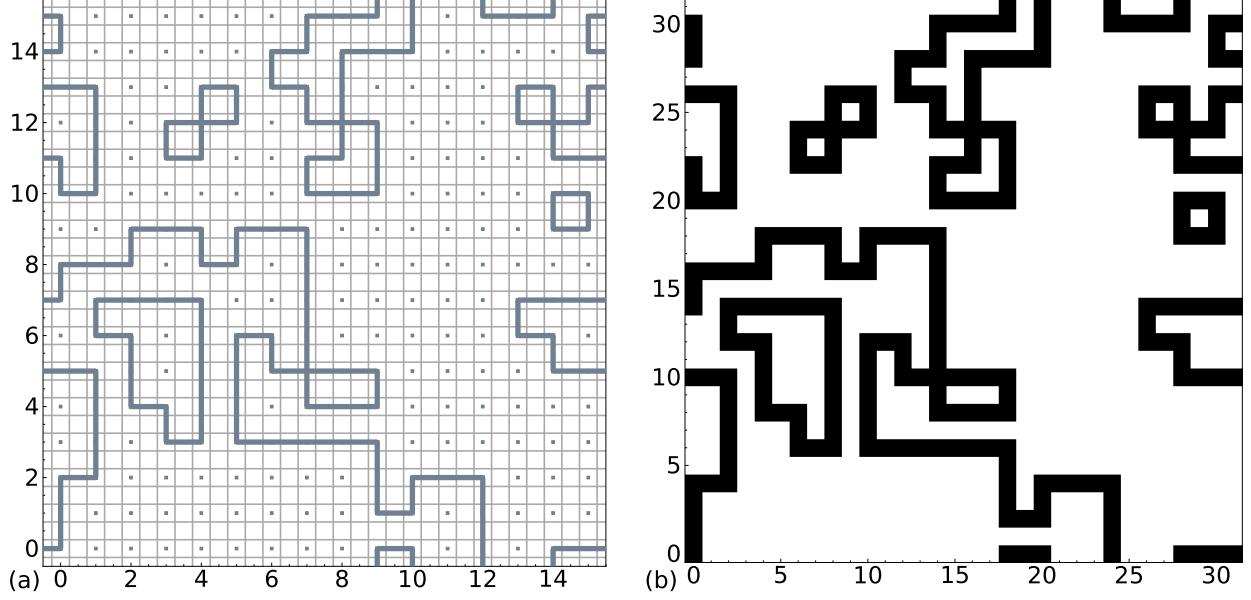


Figure 3.2: (a) Legal worm configuration on an $L \times L$ lattice with periodic boundary conditions and; (b) its equivalent representation as a $2L \times 2L$ black and white pixel image.

number of occupied bonds, in a single configuration, N_b as

$$\sum_{j=bonds} v_j = N_b \quad (3.6)$$

3.3 PCA and criticality

Having now sets of images for a range of temperatures, we can apply PCA [9]. PCA isolates the “most relevant” directions in the dataset. PCA is simply the computation of the eigenvalues λ_α and eigenvectors u_α of the covariance matrix for a dataset with N configurations corresponding to a given temperature $\{\mathbf{v}^n\}_{n=1}^N$:

$$S_{ij} = \frac{1}{N} \sum_{n=1}^N (v_i^n - \bar{v}_i)(v_j^n - \bar{v}_j). \quad (3.7)$$

In this equation, each sample \mathbf{v}_j is a vector in \mathbb{R}^{4V} , labeled by the indices $i, j = 1, \dots, 4V$. The PCA extracts solutions to

$$Su_\alpha = \lambda_\alpha u_\alpha \quad (3.8)$$

and orders them, in descending magnitude of λ_α , which are all non-negative. The usefulness of PCA is that one can approximate the data (see for instance the discussion in [9]) by the first M principal components.

Illustrations of the PCA for the MNIST data can be found in Sec. 4 of Ref. [13], where we show the eigenvectors corresponding to the largest eigenvalues and the approximation of the data by subspaces of the largest eigenvalues of dimensions 10, 20 etc.

It should be noted that the PCA is an analysis that can be performed for each temperature separately and not obviously connected to the closeness to criticality. However, we were able to find a relation between the largest PCA eigenvalue denoted λ_{\max} and the logarithmic divergence of the specific heat, namely

$$\lambda_{\max} \simeq \frac{3}{2} \langle \Delta_{N_b}^2 \rangle / V \simeq -\frac{3}{\pi} \ln(|T - T_c|). \quad (3.9)$$

This property was found by an approximate reasoning shown in Section 3.8.2 and relies on two assumptions. The first one is that the eigenvector associated with λ_{\max} is proportional to $\langle \mathbf{v} \rangle$ which is invariant under translation by two pixels in either direction. The second assumption is that in good approximation we can neglect the contributions from sites that are visited twice (four occupied links coming out of one site). Numerically, only 4% of sites are visited twice near the critical temperature which justifies the second assumption. Figure 3.3 provides an independent confirmation of the approximate validity of Eq. (3.9).

3.4 TRG coarse-graining

So far we have sampled the legal graphs of the high temperature expansion of the Ising model using the worm algorithm. An alternative approach is to use a tractable real-space renormalization group method known as the TRG [21–25].

In order to understand what we want to accomplish by blocking the loop configuration, it is useful to first understand the evolution of a tensor element using the TRG method.

The tensor formulation used here connects easily with the worm formulation used in this paper. After the character expansion has been carried out, one is left with new integer variables on the links of the lattice with constraints on the sites which guarantee the sum of the link vari-

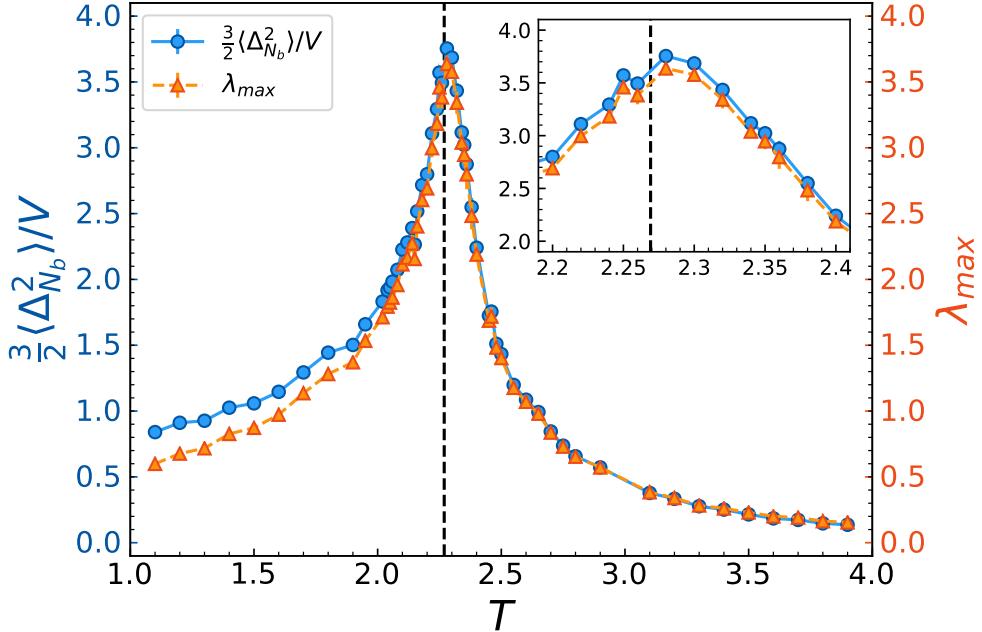


Figure 3.3: λ_{max} and $\frac{3}{2} \langle \Delta_{N_b}^2 \rangle / V$ (per unit volume) vs. T , illustrating the relation between the eigenvalue corresponding to the first principal component and the logarithmic divergence of the specific heat. The inset shows a qualitative agreement near the critical temperature.

ables associated with that site is even. Therefore we build a tensor using this constraint and the surrounding link weights. The tensor has the form

$$T_{xx'yy'}^{(i)}(\beta) = [\tanh(\beta)]^{(n_x + n_{x'} + n_y + n_{y'})/2} \times \delta_{n_x + n_{x'} + n_y + n_{y'}, 0 \bmod 2}. \quad (3.10)$$

Here the notation being used is that this tensor is located at the i^{th} site of the lattice, n_{μ} is the integer variable, taking value 0 or 1, on an adjacent link, and the Kronecker delta, $\delta_{i,j}$ is understood to be satisfied if the sum is even. By contracting these tensors together in the pattern of the lattice one recreates the closed-loop paths generated by the high-temperature expansion and exactly match those paths which are sampled by the worm algorithm.

Using these tensors one can write a partition function for the Ising model that is exactly equal to the original partition function,

$$Z = 2^V (\cosh(\beta))^{2V} \text{Tr} \prod_i T_{xx'yy'}^{(i)} \quad (3.11)$$

where Tr means contractions (sums over 0 and 1) over the links.

The most important aspect of this reformulation is that it can be coarse-grained efficiently. The process is illustrated in Fig. 3.4 where four fundamental tensors have been contracted to form a new “blocked” tensor. This new tensor has a squared number of degrees of freedom for each new effective index. The partition function can be written exactly as

$$Z = 2^V (\cosh(\beta))^{2V} \text{Tr} \prod_{2i} T'^{(2i)}_{XX'YY'} ,$$

where $2i$ denotes the sites of the coarser lattice with twice the lattice spacing of the original lattice. In practice, this exact procedure cannot be repeated indefinitely and truncations are necessary. This

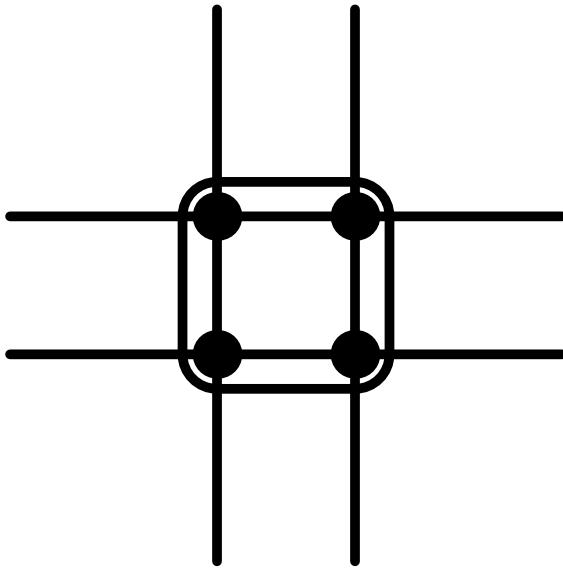


Figure 3.4: Illustration of the tensor blocking discussed in the text. Each dot is a tensor at a lattice site with four lines coming out, each representing a tensor index. Lines connecting dots represent tensor contractions.

can be accomplished by projecting the product states into a smaller number of states that optimizes the closeness to the exact answer. A two-state projection is discussed in [22] and will be followed hereafter. Note that in this procedure, T_{0000} is factored out and the final expression for the other blocked tensors are given in these units. For definiteness we consider T_{1100} which in the microscopic formulation is the weight associated with a horizontal line in a loop configuration. By looking at the fixed point equation [22], one can see that there is a high temperature fixed point where all the tensor elements except for T_{0000} are zero and a low temperature fixed point where all the tensor

elements are one. In between these two limits, there is a non-trivial fixed point illustrated by the crossing of iterated values of T_{1100} in Fig. 3.5. Note that because of the two-state approximation, the critical temperature T_c is slightly higher than the exact one [22]. To be completely specific, the exact T_c for the original model is $2/\ln(1 + \sqrt{2}) = 2.269\dots$ while for the two state projection with the second projection procedure of Ref. [22], it is $1/0.3948678 = 2.53249\dots$ It is easy to relate the

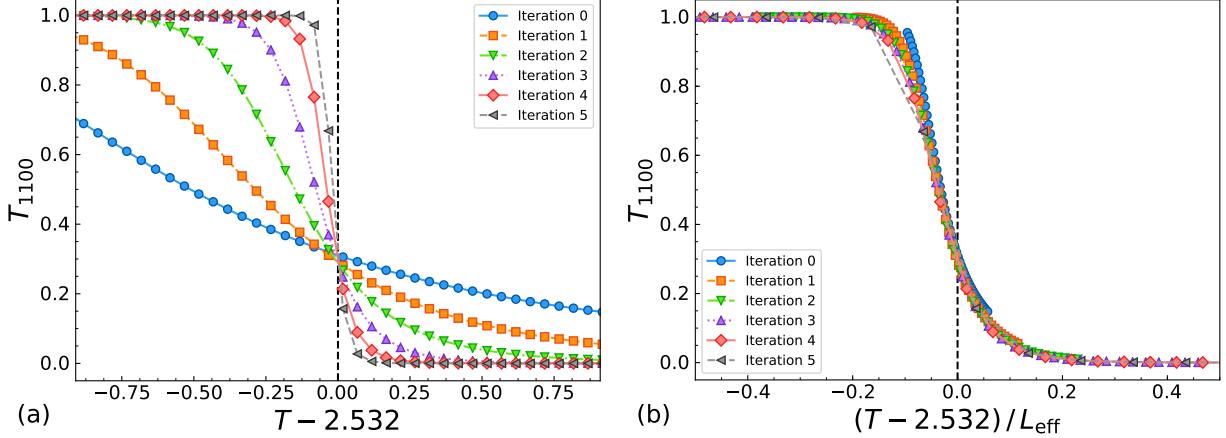


Figure 3.5: (a) T_{1100} vs. $T - T_c^{(2s)}$ for six successive iterations of the blocking transformation, beginning with an initial lattice $L = 64$; (b) T_{1100} vs. $(T - T_c^{(2s)})/L_{eff}$ illustrating the data collapse, where $T_c^{(2s)}$ is the critical temperature of the two state projection, beginning at iteration 0 on an $L = 64$ lattice.

properties of the iterated curves near the non-trivial fixed point using the linear RG approximation. Below we just state the results, for details and references see [22]. With the blocking procedure used, the scale factor is $b = 2$. The eigenvalue in the relevant direction is $\lambda = b^{1/\nu} = 2$ since $\nu = 1$. In Fig. 3.5 (a), one can see that the height

$$\delta T_{1100} \equiv T_{1100} - T_{1100}^* \quad (3.12)$$

(where T_{1100}^* is the value at the fixed point), nearly doubles each time the blocking procedure is performed, making the slope twice as large each time. A nice data collapse can be reached by offsetting this effect by rescaling the horizontal axis each iteration by $\lambda=2$ as shown in Fig. 3.5 (b). In numerical calculations, we start with a finite L (64 in Fig. 3.5) and then after ℓ iterations, we are left with an effective size $L_{eff} = L/b^\ell$.

The remainder of this chapter will be dedicated towards obtaining data collapse for $\langle N_b \rangle$

calculated with successive blockings.

3.5 Image coarse-graining

In an attempt to explicitly connect the ideas from RG theory to similar concepts in machine learning, we will implement a coarse-graining procedure directly on the images but in a way inspired by the TRG construction of Sec. 3.4. The construction relies on visual intuition and will be reanalyzed in the TRG context in Sec. 3.7.

As in the TRG coarse-graining procedure, the image is first divided up into blocks of 2×2 squares, as shown in Fig. 3.6. Each of these 2×2 squares are then replaced, or “blocked”, by a

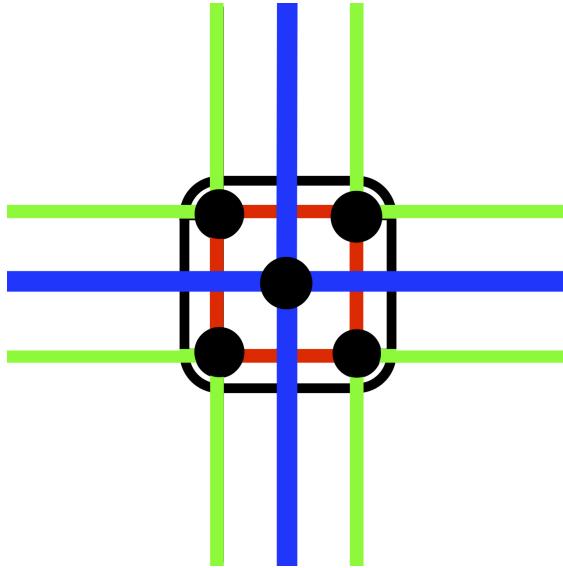


Figure 3.6: Illustration of an elementary block in the image consisting of four sites, four internal bonds (red), eight external bonds (green), and four blocked external bonds (blue).

single site with bonds determined by the number of occupied external bonds in the original square. In doing so, we reduce the size of each linear dimension by a factor of two, resulting in a new blocked configuration whose volume is one-quarter the original. In particular, if a given block has exactly one external bond in a given direction, the blocked site retains this bond in the blocked configuration. This seems to be a natural choice. However, if a given block has exactly two external bonds in a given direction, we can consider several options. The simplest option is to neglect the double bond entirely, and we denote this blocking scheme by “ $1 + 1 \rightarrow 0$ ”. This approach respects

the selection rule (conservation modulo 2) and has the advantage of maintaining the closed-path restriction. In other words with the $1 + 1 \rightarrow 0$ option, the blocked image corresponds to a legal graph and the procedure can be iterated without introducing new parameters. This procedure is illustrated for a specific configuration on a 16×16 lattice in Fig. 3.7. Alternatively, we can include

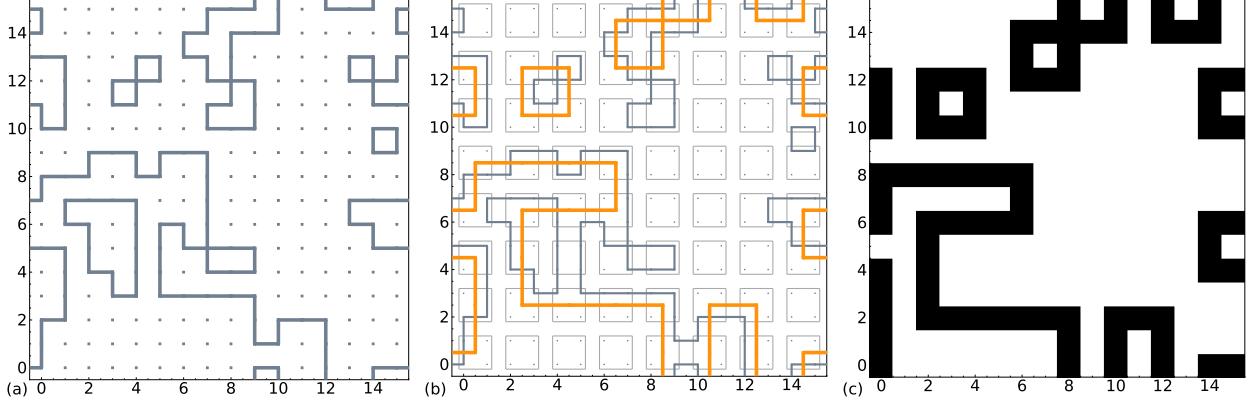


Figure 3.7: (a) Illustration of the $1 + 1 \rightarrow 0$ blocking procedure discussed in the text: original configuration; (b) introduction of the blocks and replacement of single or double bounds according to the $1 + 1 \rightarrow 0$ rule; (c) construction of the corresponding blocked image.

this double bond in the blocked configuration, and give it some weight m between 0 and 2. The examples of $m = 1$ and 2 are denoted “ $1 + 1 \rightarrow 1$ ”, and “ $1 + 1 \rightarrow 2$ ” respectively and are shown in Fig. 3.16. This blocking procedure introduces new elements and iterations require more involved procedures. This is not discussed hereafter.

3.6 Partial data collapse for blocked images

In this section, we study the properties of $\langle N_b \rangle$ obtained for successive blockings with the $1 + 1 \rightarrow 0$ rule starting with configurations on a 64×64 lattice. A first observation is that the $1+1 \rightarrow 0$ blocking preserves the location of the peak of the fluctuations $\langle \Delta_{N_b}^2 \rangle$. In addition it is possible to stabilize this quantity for a few iterations by dividing by $V_{\text{eff}} \ln(L_{\text{eff}})$. This is illustrated in Fig. 3.8. However, a very different scaling appears for the last two iterations which may be due to the very short effective sizes (4 and 2). This indicates the last two iterations are very different from the previous ones. We now consider $\langle N_b \rangle$ for successive iterations. The results are shown in Fig. 3.9. We see that in the low temperature side, the curves sharpen in a way similar to T_{1100} in Fig. 3.5. However on

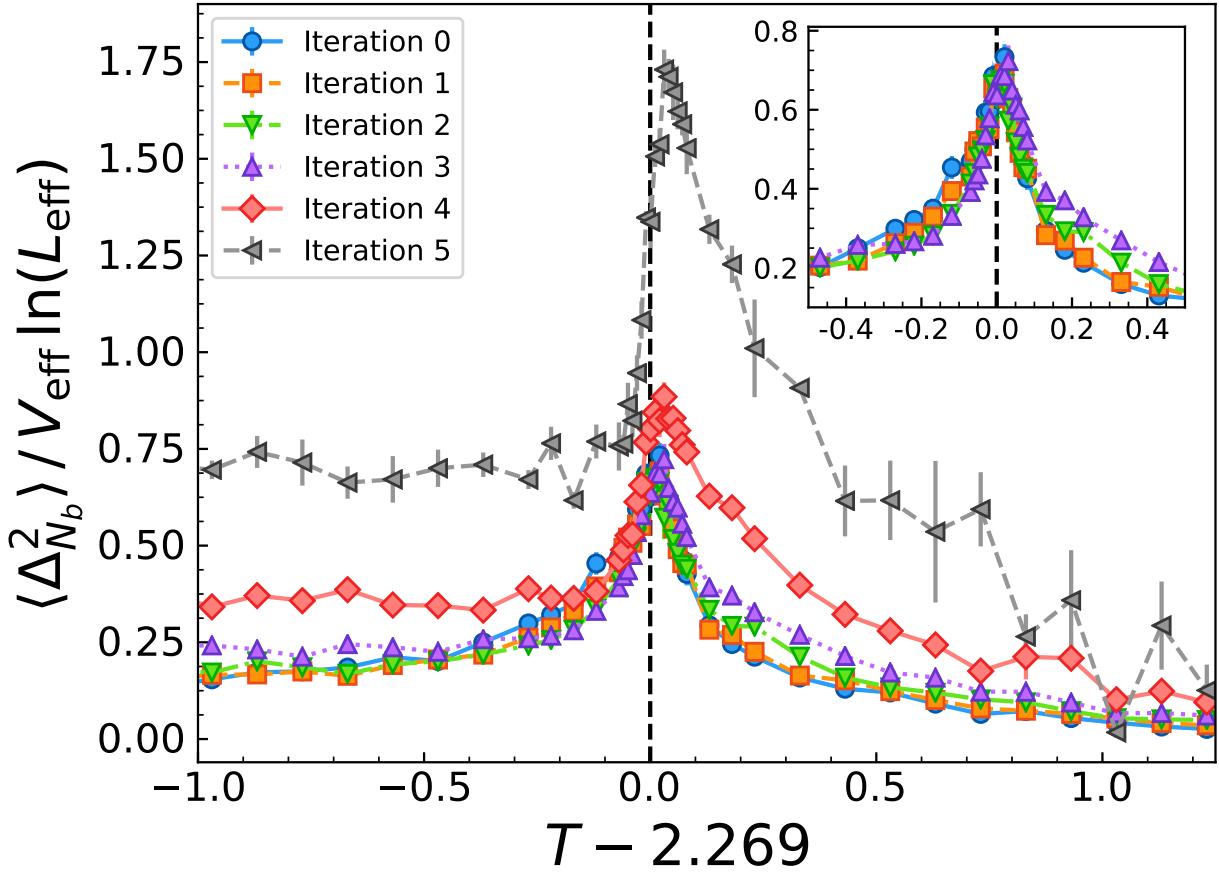


Figure 3.8: Fluctuations in the average number of bonds $\langle \Delta_{N_b}^2 \rangle$ vs. temperature T under iterated blocking steps beginning with an initial lattice size of $L = 64$. The results are scaled by $1/V_{\text{eff}} \log(L_{\text{eff}})$ in order to demonstrate the data collapse near the critical temperature. This collapse is especially apparent in the inset, which shows the results under the first three blocking steps, with $L_{\text{eff}} = 64, 32, 16$, and 8 .

the high temperature side, we observe a merging rather than a crossing. This can be explained as follows. In the high T regime occasionally a single loop, the size of a plaquette, forms. This is due to other configurations being highly suppressed. With the $1 + 1 \rightarrow 0$ rule, one out of four possible plaquettes becomes a larger plaquette which exactly compensates the change in V_{eff} which is also reduced by a factor of four. There are four kinds of plaquettes (see Fig. 3.7): those inside the blocks (they disappear after blocking), those between two neighboring blocks in the vertical or horizontal direction (these are double links between the blocks and so they disappear with the $1 + 1 \rightarrow 0$ rule), and finally those which share a corner with four blocks (they generate a larger plaquette). This last type can be seen at coordinate (4, 12) in Fig. 3.7. We now attempt to obtain data collapse

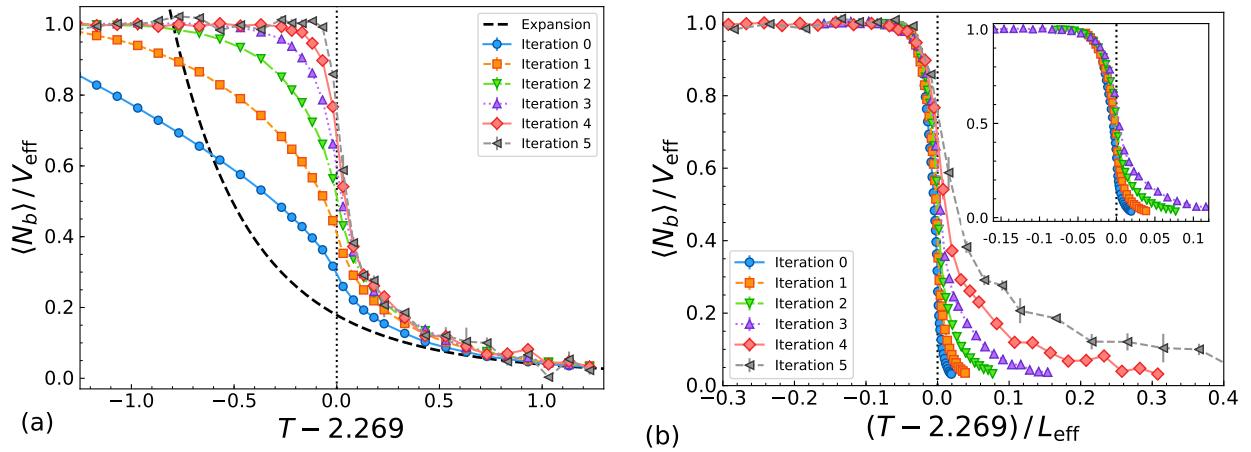


Figure 3.9: (a) Average number of bonds $\langle N_b \rangle$ vs. temperature T under iterated blocking steps beginning with an initial lattice size of $L = 64$. The dashed black line illustrates the high temperature expansion, showing that the dominant configurations are those consisting of small, isolated plaquettes. (b) Average number of bonds $\langle N_b \rangle$ vs. the rescaled temperature $(T - 2.269)/L_{eff}$ under successive blocking steps. Iteration 0 represents the original lattice before blocking, with $L_{eff} = 64$.

for $\langle N_b \rangle / V_{eff}$ by performing a rescaling of the temperature axis with respect to the critical value as in Fig. 3.5. After this rescaling by a factor 2 at each iteration, we observe a reasonable collapse on the low-temperature side. On the high temperature side, since the unrescaled curves merge, the rescaling splits them and there is no collapse on that side. This is illustrated in Fig. 3.9.

3.7 TRG calculation of $\langle N_b \rangle$

Using the tensor method we were able to calculate $\langle N_b \rangle$ to compare with the worm algorithm. Consider the equation for $\langle N_b \rangle$ with $N_b = \sum_l n_l$ the sum over bond numbers at every link:

$$\begin{aligned} \langle N_b \rangle &= \frac{1}{Z} \sum_{\{n\}} \left(\sum_l n_l \right) \left(\prod_l \tanh^{n_l}(\beta) \right) \\ &\quad \times \left(\prod_i \delta_{n_x + n_{x'} + n_y + n_{y'}, 0 \bmod 2}^{(i)} \right). \end{aligned} \quad (3.13)$$

This expression can be seen as $\langle N_b \rangle = \sum_l \langle n_l \rangle$, and because of translation and 90° rotational invariance, all $\langle n_l \rangle$ are equal. Thus, it is enough to calculate $\langle n_l \rangle$ for one particular link (just call it $\langle n \rangle$) and multiply it by $2V$: $\langle N_b \rangle = 2V\langle n \rangle$.

To calculate $\langle n \rangle$, it amounts to associating an n with one particular link on the lattice. This alters two tensors on the lattice such that the two tensors which contain that link as indices are now defined as

$$\tilde{T}_{n_x n_{x'} n_y n_{y'}}^{(1)} = \sqrt{n_x} T_{n_x n_{x'} n_y n_{y'}}(\beta), \quad (3.14)$$

$$\tilde{T}_{n_x n_{x'} n_y n_{y'}}^{(2)} = \sqrt{n_{x'}} T_{n_x n_{x'} n_y n_{y'}}(\beta), \quad (3.15)$$

where x and x' were chosen without loss of generality. It could just as well have been chosen as y and y' . One can see that when these two tensors are contracted along their shared link, the product picks up a factor of n for that link, which when combined with the other tensors in the lattice, and divided by Z , yields $\langle n \rangle$.

Knowing the above, one is free to block and construct the partition function, Z , and $\langle n \rangle$. This can be done by blocking symmetrically in both directions, or by constructing a transfer matrix by contracting only along a time-slice (i.e. a snapshot of the system at fixed t). This is shown in Fig. 3.10. In practice contracting to build a transfer matrix is optimum since one direction of the lattice is never renormalized and allows the easy calculation of $\langle n \rangle$. What was just described is a method to calculate $\langle n \rangle$ for the original, fine lattice. However, one can also calculate the same quantity for a coarse lattice. The actual blocking method is essentially identical, with a small

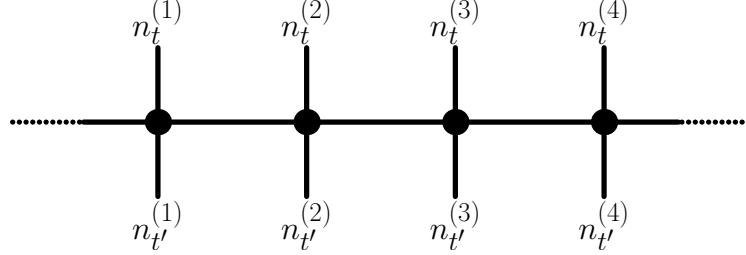


Figure 3.10: A pictorial representation of the transfer matrix made by contracting a fundamental tensor along a single time-slice.

difference. Instead of contracting the fundamental tensor to the desired lattice size, one contracts a blocked tensor to the desired lattice size.

For example, if one wanted to calculate $\langle N_b \rangle$ for a 32×32 lattice, one could contract the fundamental tensor along a time slice with itself five times. This would give a $2^{32} \times 2^{32}$ transfer matrix which could be used to build the whole partition function. Now, under a single coarse-graining step the 32×32 lattice becomes a 16×16 lattice of blocks. Therefore, to build this, one could contract four fundamental tensors in a block and consider this a new, effective fundamental tensor. This is shown in Fig. 3.4. Then one repeats the same steps to construct the transfer matrix, however only contracting four times with itself to create a matrix representing 16 lattice sites of the blocked tensor.

To actually calculate $\langle n \rangle$ by building the transfer matrix, one can take the final tensor, prior to contracting the dangling spatial indices, and multiply by \sqrt{n} against the indices n_x and $n_{x'}$. This is shown for the unblocked case in Fig. 3.11, however the procedure is identical for the blocked case once the blocked tensor has been constructed. This is also the point where one can choose the level of approximation one will use in the blocking. For instance one could choose that the state $|1 1\rangle \rightarrow |0\rangle$ and assign $n = 0$ to that state. Alternatively one could preserve N_b and let $|1 1\rangle \rightarrow |2\rangle$ and assign $n = 2$ to that state. This procedure was found to agree with the results obtained by changing the pixels of the worm configurations.

Once the original transfer matrix has been constructed, as well as the matrix with the insertion of n along a single link, one can combine these to find $\langle n \rangle$. This is done by simple matrix multiplication:

$$\langle n \rangle = \frac{1}{Z} \text{Tr}[\underbrace{\mathbb{T} \cdots \mathbb{T}'}_{N_\tau} \cdots \mathbb{T}], \quad (3.16)$$

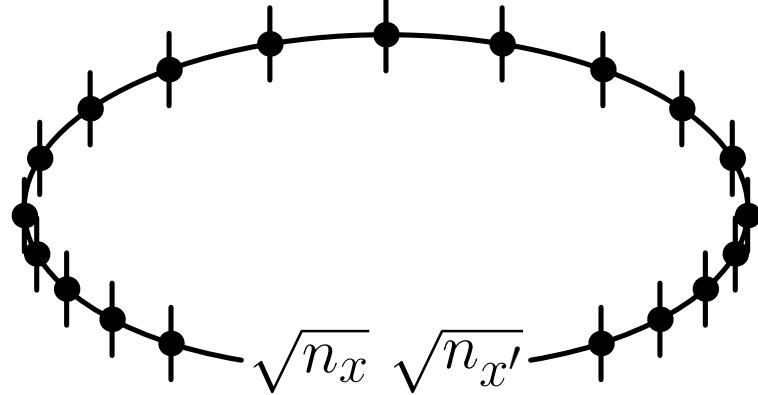


Figure 3.11: By multiplying the remaining free spatial indices by \sqrt{n} and contracting for periodic boundary conditions in space we form an “impure” transfer matrix. Combining the resultant matrix with the original transfer matrix allows one to calculate $\langle n \rangle$.

with

$$Z = \text{Tr}[\mathbb{T}^{N_\tau}]. \quad (3.17)$$

Here \mathbb{T} represents the transfer matrix built by contracting tensors along a time slice, and \mathbb{T}' represents the single (“impure”) transfer matrix at a time-slice with a single bond multiplied by n . Since the lattice has Euclidean temporal extent, $L = N_\tau$, there are that many matrices multiplied in each case. The values of $\langle N_b \rangle / V_{\text{eff}}$ obtained with this procedure are shown in Fig. 3.12. The rescaling by 2 at each iteration provides a good data collapse on both sides of the transition.

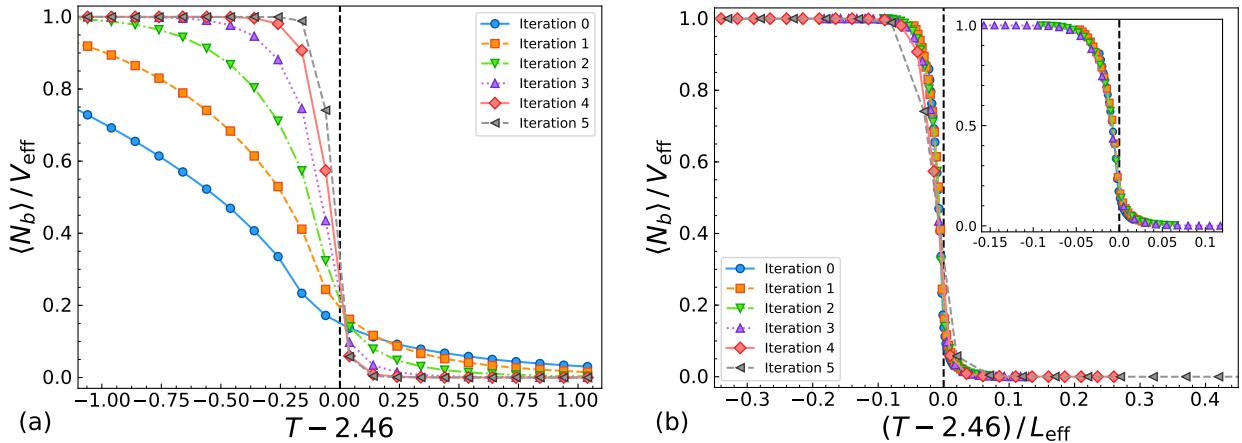


Figure 3.12: (a) $\langle N_b \rangle$ vs $(T - 2.46)$ under successive blocking steps calculated using 2-state HOTRG. (b) $\langle N_b \rangle$ vs $(T - 2.46)/L_{\text{eff}}$ under successive blocking steps calculated using 2-state HOTRG. Note that the value of 2.46 was determined qualitatively by choosing the value which gave the best resulting data collapse.

3.8 Technical results

3.8.1 Loop representation

We can rewrite the Ising partition function in terms of bonds between neighboring sites $\langle i, j \rangle$. The allowed bond configurations are concisely described by concepts in graph theory, because they form edges (bonds) between neighboring vertices (sites). Making use of well-known identities allows for the partition function to be written in the following high-temperature expansion:

$$Z = 2^{|V|} \cosh^{|E|} \beta \sum_{\Gamma \in \mathcal{C}(G)} \tanh^{|\Gamma|} \beta \quad (3.18)$$

$$= 2^{|V|} \cosh^{|E|} \beta \sum_{|\Gamma|} \tanh^{|\Gamma|} \beta n(|\Gamma|) \quad (3.19)$$

The notation is as follows. We have a graph $G = (V, E)$ that describes our lattice, where V are the vertices and E are the edges, which are the bonds between neighboring sites. If we restrict ourselves to subgraphs with only occupied bonds allowed by the Ising model, then the degree of each vertex is even. This is the number of bonds emanating from a particular vertex. The set of edges of such a subgraph is described as being “Eulerian.” The space of all such sets of edges is known as the cycle space $\mathcal{C}(G)$. The notation $|V|$, $|E|$, $|\Gamma|$ denotes the number of elements in each set (cardinality). In the second line, $n(|\Gamma|)$ counts the multiplicity of subgraphs of cardinality $|\Gamma|$, and is zero when $|\Gamma|$ does not correspond to a “legal” subgraph.

We now specialize the presentation to the case of the two-dimensional Ising model on a square lattice with periodic boundary conditions. In this case $|V|$ is $V = L^2$, the volume that we express in lattice units, and $|E| = 2V$. We introduce the notation $t \equiv \tanh(\beta)$ and we call N_b the number of bonds in a graph (values taken by $|\Gamma|$). With these notations we recover Eq. 3.3.

It is this bond formulation that is the basis of both random cluster algorithms [29] and worm algorithms [27]. In this paper we utilize the latter. Both of these classes of algorithms have the benefit of significantly avoiding critical slowing down. This is essential near the critical temperature T_c .

3.8.2 Heat capacity

One striking feature of the second order transition for the two-dimensional Ising model is the logarithmic divergence of the specific heat density at the critical temperature T_c . In this section, we review the way the specific heat can be calculated with the worm algorithm and we check our answer with the exact finite volume expressions [30].

Using the standard thermodynamical formula for the average energy

$$\langle E \rangle = -\frac{\partial}{\partial \beta} \ln Z, \quad (3.20)$$

with the expression Eq. (3.3) of Z , we get

$$\langle E \rangle = -\tanh(\beta) \left(2V + \frac{\langle N_b \rangle}{\sinh^2(\beta)} \right), \quad (3.21)$$

where we define

$$\langle f(N_b) \rangle \equiv \sum_{N_b} f(N_b) t^{N_b} n(N_b) / \sum_{N_b} t^{N_b} n(N_b). \quad (3.22)$$

We can then use

$$C_V = \frac{\partial \langle E \rangle}{\partial T}, \quad (3.23)$$

to write

$$\frac{C_V}{V} = \beta^2 \left[\frac{2}{\cosh^2(\beta)} - \frac{4 \cosh(2\beta)}{\sinh(2\beta)} \frac{\langle N_b \rangle}{V} + \left(\frac{2}{\sinh(2\beta)} \right)^2 \frac{\langle (N_b - \langle N_b \rangle)^2 \rangle}{V} \right]. \quad (3.24)$$

Since $\frac{\langle N_b \rangle}{V} \leq 2$ (in two dimensions), the only possibly divergent part is the variance of N_b per unit volume $\langle \Delta_{N_b}^2 \rangle$ defined in Eq. (3.4). The singularity near T_c is known from Onsager's solution:

$$\frac{C_V}{V} = -\frac{2}{\pi} \left(\ln(1 + \sqrt{2}) \right)^2 \ln(|T - T_c|) + \text{regular}. \quad (3.25)$$

This implies Eq. (3.5).

3.8.3 Monte Carlo implementation

We can proceed to sample the closed path configuration space using the worm algorithm [27]. A single Monte Carlo step is outlined below.

1. Randomly select a starting point on the lattice (i_0, j_0) .
2. Propose a move to a neighboring site (i', j') , selected at random.
3. If no link is present between these two sites, a bond is created with acceptance probability $P = \min\{1, \tanh \beta\}$. If the bond is accepted, we update the bond number for the present worm, $n_b = n_b + 1$.
4. If a link already exists between the two sites, it is removed with probability $P = 1$.
5. If $(i', j') = (i_0, j_0)$, i.e. we have a closed path, go to (1.). Otherwise, $(i', j') \neq (i_0, j_0)$, go to (2.)

The number of necessary Monte Carlo steps required to achieve sufficient statistics varies with the lattice size, thermalization time, and temperature. After each step, we calculate the energy in terms of the average number of active bonds N_b , and consider the system to be thermalized when fluctuations between subsequent values of the energy are less than 1×10^{-3} . We then save the resulting configuration, along with the final values for all physical quantities of interest. This process is then repeated many times over a range of different temperatures to generate sufficient statistics for physical observables. All errorbars are calculated using the block jackknife resampling technique.

3.8.4 Tests

The above formulas have been used to calculate C_V . Precise checks were performed by comparing with the exact results obtained from Ref. [30]. The agreement can be seen in Fig. 3.13. Results for other lattice sizes that we have simulated are similar.

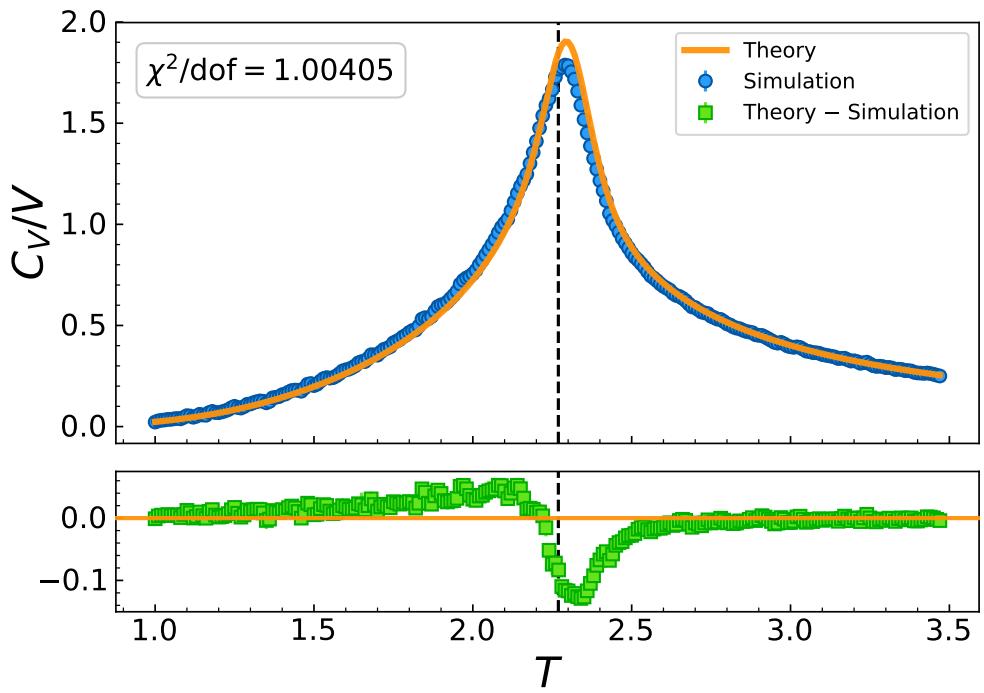


Figure 3.13: Comparison of the worm Monte Carlo computation of the specific heat C_v versus temperature and the exact results using the formula in [30], for an $L = 32$ lattice. Note that χ^2/dof represents the reduced chi-squared statistic. It can be seen that the agreement is excellent, except for a slight deviation at the critical temperature, where Monte Carlo algorithms tend to face the worm algorithm, in terms of having a dynamical scaling exponent that is zero rather than two, but there is (as can be seen), still a residual suppression of fluctuations in the immediate vicinity of T_c .

3.8.5 Conjecture about λ_{\max}

Using the Monte Carlo algorithm outlined above, we can calculate the average number of occupied bonds at a particular temperature by averaging over all configurations

$$\langle N_b \rangle \equiv \frac{1}{N_{\text{configs}}} \sum_{n=1}^{N_{\text{configs}}} N_b^{(n)} \quad (3.26)$$

$$= \left\langle \sum_{j=\text{bonds}} v_j \right\rangle \quad (3.27)$$

$$= 2V \langle \mathbf{v}_b \rangle. \quad (3.28)$$

From this, we have that

$$\langle \mathbf{v}_b \rangle = \frac{\langle N_b \rangle}{2V}, \quad (3.29)$$

where we have defined $\langle \mathbf{v}_b \rangle$ to be the average occupation of bonds, $N_b^{(n)}$ to be the number of occupied bonds in the n -th configuration, and we have used Eq. (3.6) in the second line. If we consider graphs with no self-intersections,

$$\sum_{j=\text{bonds}} v_j = \sum_{j=\text{sites}} v_j. \quad (3.30)$$

For small β (high T) this can be a good approximation,

$$\left\langle \sum_{j=\text{bonds}} v_j \right\rangle \simeq \left\langle \sum_{j=\text{sites}} v_j \right\rangle \Rightarrow \quad (3.31)$$

$$\langle \mathbf{v}_b \rangle \simeq \frac{1}{2} \langle \mathbf{v}_s \rangle \quad (3.32)$$

This agrees with our intuition, that the average image $\langle \mathbf{v} \rangle$ should resemble a “tablecloth”, where the site pixels are twice as dark as the link pixels. This can be seen clearly in Fig. 3.14. For a general graph, a link is shared by two sites (its endpoints), whereas a bond, it is only visited once, denoted $sites^{(1)}$, and if it is shared by four bonds, it is visited twice, denoted $sites^{(2)}$. This allows

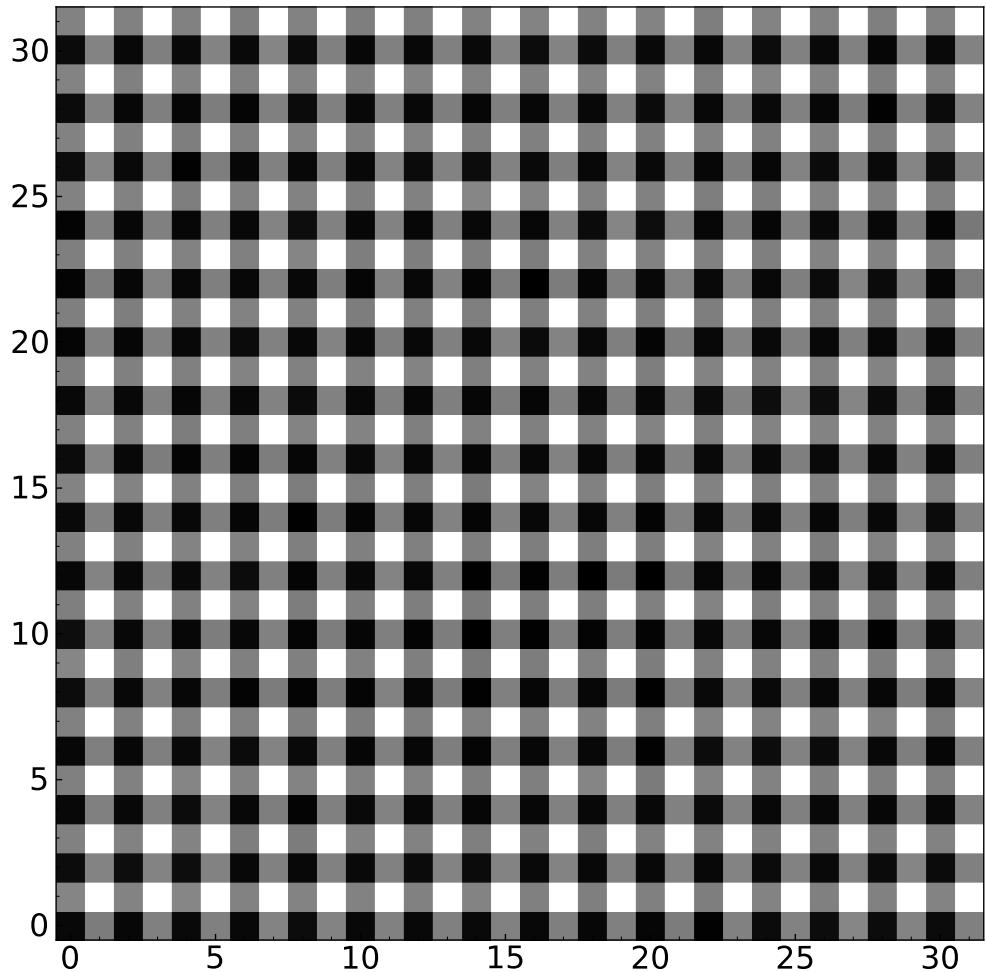


Figure 3.14: Average image $\langle v \rangle$ calculated for the $L = 16$ lattice at $T = 2.0$, illustrating the tablecloth-like appearance.

us to break up the sum over bonds into two terms

$$\sum_{j=bonds} v_j = \frac{2}{2} \sum_{j=sites^{(1)}} v_j + \frac{4}{2} \sum_{j=sites^{(2)}} v_j \quad (3.33)$$

Rearranging and taking averages gives

$$\left\langle \sum_{j=sites^{(1)}} v_j + \sum_{j=sites^{(2)}} v_j \right\rangle = \left\langle \sum_{j=bonds} v_j - \sum_{j=sites^{(2)}} v_j \right\rangle \quad (3.34)$$

$$= \left\langle \sum_{j=sites} v_j \right\rangle \quad (3.35)$$

$$= V \langle \mathbf{v}_s \rangle \quad (3.36)$$

$$= \left\langle \sum_{j=bonds} v_j \right\rangle - \left\langle \sum_{j=sites^{(2)}} v_j \right\rangle \quad (3.37)$$

$$= 2V \langle \mathbf{v}_b \rangle - \langle N_{sites^{(2)}} \rangle \implies \quad (3.38)$$

$$\frac{\langle N_{sites^{(2)}} \rangle}{V} = 2 \langle \mathbf{v}_b \rangle - \langle \mathbf{v}_s \rangle. \quad (3.39)$$

We can rewrite the last equation using (3.28)

$$\langle \mathbf{v}_s \rangle = \frac{\langle N_b \rangle}{V} - \frac{\langle N_{sites^{(2)}} \rangle}{V}. \quad (3.40)$$

This suggests that a departure from a perfect tablecloth ($\langle \mathbf{v}_s \rangle = 2 \langle \mathbf{v}_b \rangle$) contains information.

$$C_{ij} = \left\langle (v_i - \langle \mathbf{v} \rangle_i) (v_j - \langle \mathbf{v} \rangle_j)^T \right\rangle \quad (3.41)$$

$$= \frac{1}{N_{configs}} \sum_{n=1}^{N_{configs}} (v_i^{(n)} - \langle \mathbf{v} \rangle_i) (v_j^{(n)} - \langle \mathbf{v} \rangle_j)^T, \quad (3.42)$$

where we have defined $v_k^{(n)}$ to be the grayscale value of the k -th pixel in the n -th sample configuration, and $\langle \mathbf{v} \rangle_k$ is the average grayscale value of the k -th pixel over the set of configurations.

At some fixed temperature, the covariance matrix, $C \in \mathbb{R}^{N_{configs} \times 4L^2}$, where $N_{configs}$ is the number of sample configurations (images), with each configuration flattened into a vector of length

$4L^2$. We can then perform a singular value decomposition (SVD) on the covariance matrix,

$$C = W\Lambda W^T \quad (3.43)$$

where W is a $4L^2 \times 4L^2$ matrix whose columns (\mathbf{w}_k) are the eigenvectors of C , and Λ is the diagonal matrix of the absolute value of the eigenvalues $\lambda^{(k)}$ of C , arranged in decreasing order. Without loss of generality, we can assume that the eigenvectors \mathbf{w}_k are real and normalized such that $\mathbf{w}_k^T \mathbf{w}_k = 1$. Thus, we can write

$$C\mathbf{w}_k = \lambda^{(k)}\mathbf{w}_k \quad (3.44)$$

$$\mathbf{w}_k^T C \mathbf{w}_k = \lambda^{(k)} \quad (3.45)$$

For our purposes, we are interested in the first principal component, with eigenvalue $\lambda^{(1)} \equiv \lambda_1$ and corresponding eigenvector \mathbf{w}_1 .

We conjectured that the first principal component, \mathbf{w}_1 of the covariance matrix C is directly proportional to the average worm configuration (image) $\langle \mathbf{v} \rangle$, i.e.

$$\mathbf{w}_1 \propto \langle \mathbf{v} \rangle. \quad (3.46)$$

From our results in 3.2, we can write

$$\langle \mathbf{v} \rangle^2 = \langle \mathbf{v} \rangle^T \langle \mathbf{v} \rangle \quad (3.47)$$

$$= 2V\langle \mathbf{v}_b \rangle^2 + V\langle \mathbf{v}_s \rangle^2. \quad (3.48)$$

This suggests that

$$\mathbf{w}_1 = \frac{\langle \mathbf{v} \rangle}{\sqrt{(2\langle \mathbf{v}_b \rangle^2 + \langle \mathbf{v}_s \rangle^2)V}}. \quad (3.49)$$

Moreover, we can write

$$\sum_i (v_i^{(n)} - \langle \mathbf{v} \rangle_i) \langle \mathbf{v} \rangle_i = \langle \mathbf{v}_b \rangle \sum_{j=bonds} v_j^{(n)} + \langle \mathbf{v}_s \rangle \sum_{j=sites} v_j^{(n)} - (2V\langle \mathbf{v}_b \rangle^2 + V\langle \mathbf{v}_s \rangle^2) \quad (3.50)$$

$$= \langle \mathbf{v}_b \rangle N_b^{(n)} + \langle \mathbf{v}_s \rangle N_s^{(n)} - V(2\langle \mathbf{v}_b \rangle^2 + \langle \mathbf{v}_s \rangle^2) \quad (3.51)$$

$$= \langle \mathbf{v}_b \rangle (N_b^{(n)} - \langle N_b \rangle) + \langle \mathbf{v}_s \rangle (N_s^{(n)} - \langle N_s \rangle) \quad (3.52)$$

$$\equiv \langle \mathbf{v}_b \rangle \Delta_{N_b}^{(n)} + \langle \mathbf{v}_s \rangle \Delta_{N_s}^{(n)}. \quad (3.53)$$

From this, we can extract a relationship between the eigenvalue corresponding to the first principal component, $\lambda^{(1)}$ and the fluctuations Δ_{N_b} and Δ_{N_s} ,

$$\begin{aligned} \mathbf{w}_1^T C \mathbf{w}_1 &= \lambda^{(1)} \\ &= \frac{1}{N_{\text{configs}}} \sum_{n=1}^{N_{\text{configs}}} \frac{\left[\langle \mathbf{v}_b \rangle \Delta_{N_b}^{(n)} \right]^2 + 2\langle \mathbf{v}_b \rangle \langle \mathbf{v}_s \rangle \Delta_{N_b}^{(n)} \Delta_{N_s}^{(n)} + \left[\langle \mathbf{v}_s \rangle \Delta_{N_s}^{(n)} \right]^2}{2\langle \mathbf{v}_b \rangle^2 + \langle \mathbf{v}_s \rangle^2} \end{aligned}$$

Now, if we consider the high temperature approximation where sites only have single visits (no self-intersections), $\langle \mathbf{v}_s \rangle \simeq 2\langle \mathbf{v}_b \rangle$, $\langle N_s \rangle \simeq \langle N_b \rangle$, and $\Delta_{N_b} \simeq \Delta_{N_s}$, we have that $2\langle \mathbf{v}_b \rangle^2 + \langle \mathbf{v}_s \rangle^2 \simeq 6\langle \mathbf{v}_b \rangle^2$ and

$$\lambda_1 \simeq \frac{\langle \mathbf{v}_b \rangle^2}{N_{\text{configs}}} \frac{9}{6\langle \mathbf{v}_b \rangle^2} \sum_{n=1}^{N_{\text{configs}}} (\Delta_{N_b}^{(n)})^2 \quad (3.54)$$

$$= \frac{3}{2} \frac{1}{N_{\text{configs}}} \sum_{n=1}^{N_{\text{configs}}} (\Delta_{N_b}^{(n)})^2 \quad (3.55)$$

$$= \frac{3}{2} \langle \Delta_{N_b}^2 \rangle. \quad (3.56)$$

A justification for making this approximation can be seen in Fig. 3.15.

3.8.6 Illustration of alternate blockings

In Fig 3.16 we consider the alternative blocking procedures: $1 + 1 \rightarrow 1$ and $1 + 1 \rightarrow 2$.

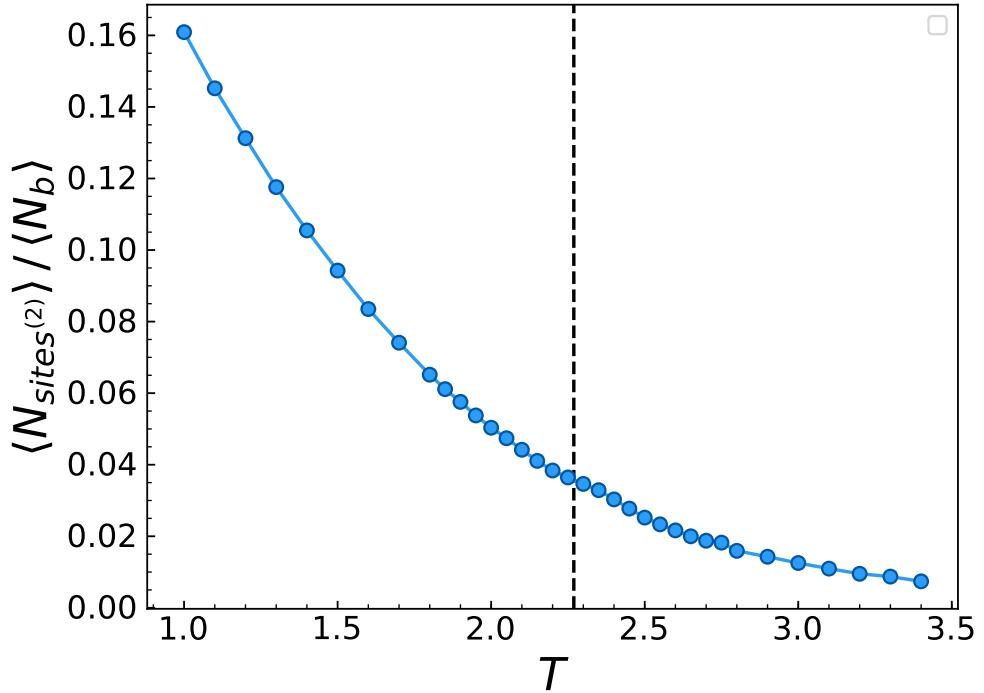


Figure 3.15: Ratio of the number of twice visited sites $\langle N_{sites^{(2)}} \rangle$ to the average number of bonds $\langle N_b \rangle$ versus temperature, for the $L = 32$ lattice. This clearly justifies our approximation $\langle v_s \rangle \simeq 2\langle v_b \rangle$, where we ignore the contribution from twice visited sites.

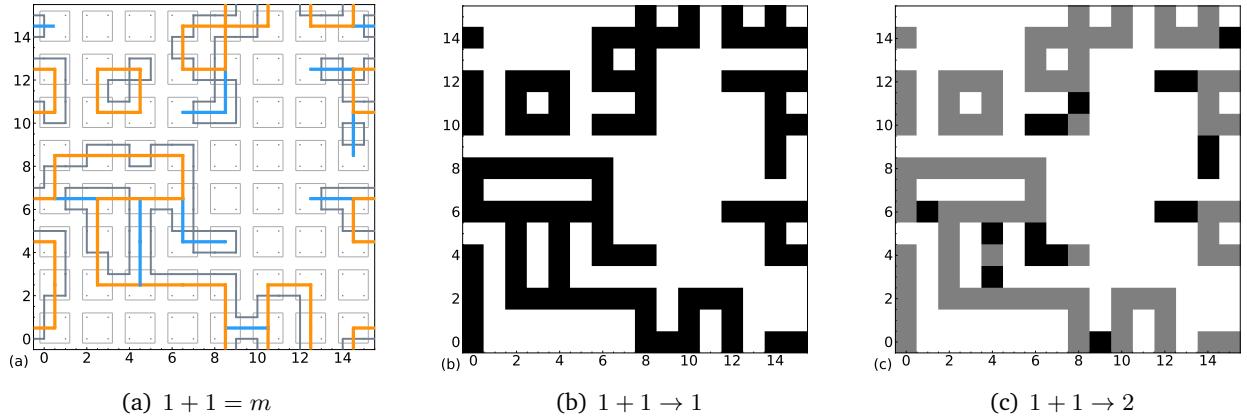


Figure 3.16: Example of the different coarse-graining (“blocking”) procedures applied to a sample worm configuration generated at $T = 2.0$. Note that in (3.16(a)) $m \in \{1, 2\}$, and double bonds are represented by blue lines. (3.16(b)), (3.16(c)), illustrate the results of applying different weights to the so-called “double bonds” in the images representing a blocked configuration. Note that in (3.16(b)) $1 + 1 \rightarrow 1$, double bonds are given the same weight as single bonds, and in (3.16(c)) $1 + 1 \rightarrow 2$, double bonds are given twice the weight as single bonds, appearing twice as dark.

3.9 Possible applications: From Images to Loops

Having better understood how these RG transformations can be used to describe the 2D Ising model near criticality, we began to look for possible applications to real-world datasets. For our analysis, we used the CIFAR-10 [31] image set consisting of 60,000 32×32 color images in 10 classes. First, each of the images were converted to a grayscale with pixel values in the range $[0, 1]$. Next, a grayscale cutoff value was chosen so that all pixels with values below the cutoff would become black, and pixels above the cutoff would become white, resulting in images consisting entirely of black and white pixels. Finally, each of these images were converted to ‘worm-like’ images by drawing the boundaries separating black and white collections of pixels. An example of these preprocessing steps are illustrated in Fig. 3.17.

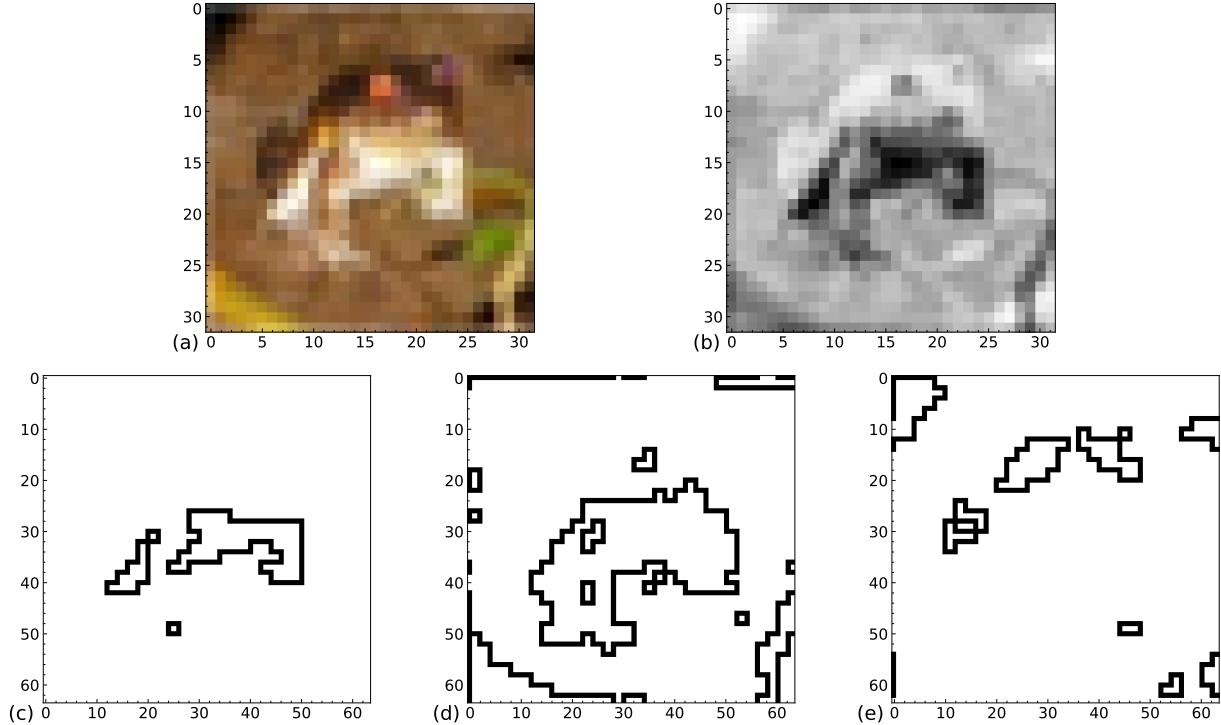


Figure 3.17: Example of preprocessing steps for converting CIFAR-10 images to ‘worm-like’ images, illustrating the resulting image for different values of the grayscale cutoff. (a) Original image from CIFAR-10 dataset. (b) Image converted to grayscale. (c) Resulting image from cutoff values of 0.25, (d) 0.5, and (e) 0.75.

This process was carried out on a mini-batch consisting of 500 randomly selected images from the CIFAR-10 image set. For each image in our mini-batch, we calculated $\langle N_b \rangle$ and $\langle \Delta_{N_b}^2 \rangle$

over a range of grayscale cutoff values in $[0, 1]$ in steps of 0.02. Each of these images were then iteratively blocked using the $(1 + 1 \rightarrow 0)$ blocking procedure described in Sec. 3.4, calculating $\langle N_b \rangle$ and $\langle \Delta_{N_b}^2 \rangle$ for each successive blocking step, as shown in Fig. 3.18. Immediately we see that there

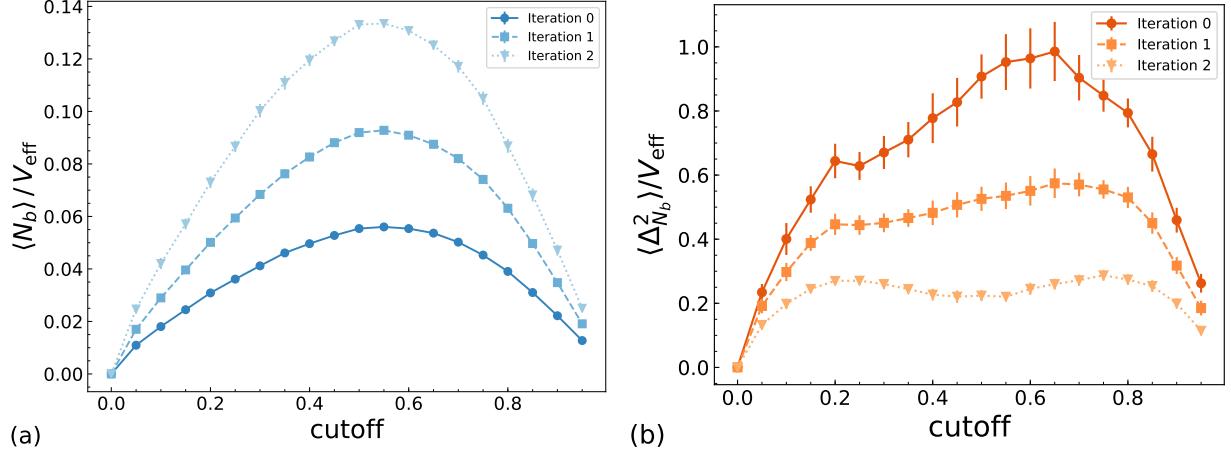


Figure 3.18: $\langle N_b \rangle$ and $\langle \Delta_{N_b}^2 \rangle$ vs. grayscale cutoff value for 500 randomly chosen images from the CIFAR-10 dataset.

is no identifiable low temperature phase, and that for cutoff values near both 0 and 1, we obtain images which are mostly empty, similar to the high temperature configurations obtained from the worm algorithm. This suggests that there is no such notion of criticality (as characterized by the abrupt transition from a low to high temperature phase) like we found for the two-dimensional Ising model.

3.10 Conclusions

In summary, we have motivated, constructed and applied a RG transformation to sets of worm configurations at various temperature. This transformation is approximate and the coarse-grained configurations are themselves worm configurations. This allows multiple iterations. We monitored the bond density at successive iterations and compared them with a two-state TRG approximation. We found clear similarities in the low-temperature side, where data collapse is observed for both procedures when the distance to the critical point is rescaled at each iteration. In the high-temperature phase, only the TRG approximation shows good data collapse.

Can the procedure developed here be applied to the boundary of arbitrary sets of images as

illustrated in the introduction? The gray cutoff could be used as a tunable parameter. However, in the limiting cases of a zero (one) gray cutoff we have uniform black (white) images which are both similar to the high-temperature phase, and we do not expect a phase transition. Applications to the CIFAR dataset are discussed in Section 3.9 and confirm this point of view.

RG methods have been considered for assisting in image recognition [32–34]. By mapping from fine to coarse in several ways, such as the $1+1 \rightarrow 0$ and $1+1 \rightarrow 1$ in our approach, one begins to see how the inverse process might go in replacing a degraded image with a higher resolution reconstruction. The physics of defining RG transformations and quantifying scheme dependence then guides such reconstructions using physical principles, which are expected to be embedded into real world image characteristics due to principles of universality.

It should be noted that the TRG procedure is often considered as a “local update” of the tensor. A more sophisticated approach consists of using the standard recursion to provide an environment for subsequent updates [21, 35]. An environment tensor E is propagated backward from the coarse to the fine scale. An improved version of the initial iteration can then be performed in an environment. This forward-backward procedure can be repeated and is very reminiscent of the procedure proposed by Hinton and Salakhutdinov [36] in the context of image recognition.

A better understanding of RG concepts in machine learning could enhance physics discovery, especially in the context of simulation and modeling of physical systems at a fundamental level [37]. The general idea is to render computational tools “smart,” i.e., that they would learn features and patterns without the intervention of a human “assistant,” and would, in the best possible scenario, guide the direction of further simulations. This could accelerate and deepen the process of understanding and characterizing the complex systems that are deemed important in pure and applied physics.

4 | Markov Chain Monte Carlo (MCMC)

For consistency with the original paper, we adopt much of the same notation. In order to better understand the theory behind the L2HMC algorithm, we begin with a general description of Markov Chain Monte Carlo (MCMC) methods.

4.1 Markov Chains

Generally speaking, MCMC methods are a class of algorithms that use Markov Chains to sample from a particular probability distribution that is often too complicated to sample from directly. These algorithms are of tremendous importance in a wide variety of fields and are of particular importance for simulations in lattice quantum chromodynamics (QCD) and lattice gauge theory. A Markov chain can be understood as a sequence of random variables $\{x_1, x_2, \dots, x_N\}$ sampled from a conditional probability distribution $p(x_{t+1}|x_t)$, with the condition that the next sample x_{t+1} depends only on the current state x_t and does not depend on the history of previous states $\{x_1, x_2, \dots, x_{t-1}\}$ [38]. The set in which the x_i take values is often referred to as the *state space* \mathcal{X} of the Markov chain. For finite state spaces, the initial distribution can be associated with a vector $\lambda = (\lambda_1, \dots, \lambda_N)$ defined by

$$p(x_1 = x_i) = \lambda_i, \quad i = 1, \dots, N \quad (4.1)$$

and the transition probabilities can be associated with a transition matrix (kernel) P with elements p_{ij} given by

$$p(x_{t+1} = x_j | x_t = x_i) = p_{ij}, \quad i = 1, \dots, N \quad j = 1, \dots, n \quad (4.2)$$

This says that the $(i,j)^{\text{th}}$ entry of the n^{th} power of P gives the probability of transitioning from state i to state j in n steps. For our purposes, we are usually interested in Markov chains that have stationary transition probabilities, i.e. the conditional distribution of x_{t+1} given x_t does not depend on t . Note that for $i = 1, 2, \dots, N$,

$$\sum_{j=1}^N p_{ij} = 1. \quad (4.3)$$

There are two main (defining) properties of Markov chains that are relevant for our discussion, namely *stationarity* and *reversibility*.

- **Stationarity:** A sequence $\{x_1, x_2, \dots, x_N\}$ of random elements is said to be *stationary* if for every positive integer k the distribution of the k -tuple

$$(x_{t+1}, \dots, x_{t+k})$$

does not depend on t . An initial distribution is said to be stationary if the Markov chain specified by the initial distribution and transition probability distribution is stationary.

- **Reversibility:** A Markov chain is said to be reversible if its transition probability is reversible with respect to its initial distribution. Note that this is a stronger condition than *stationarity* since if a Markov chain is reversible it is also stationary. This condition can be better understood by noticing that for any $i, k > 0 \in \mathbb{Z}$, the distributions of $(x_{i+1}, \dots, x_{i+k})$ and $(x_{i+k}, \dots, x_{i+1})$ are the same.

Additionally, we often desire that our Markov chain is

- **Irreducible:** For any state of the Markov chain, there is a positive probability of visiting all other states, i.e. any two configurations are connected by a finite number of steps..
- **Aperiodic:** The chain will not tend towards a periodic limit-cycle where we can only reach a certain subset of states at any step, i.e. we must be able to reach all states at any given time step [39].

More formally, if we define the *period* of a state $x \in \mathcal{X}$ to be:

$$d(x) = \gcd\{n \in \mathbb{N}^+ : P^n(x, x) > 0\}, \quad (4.4)$$

then, starting at x , the chain can return to x only at multiples of the period d , where d is the largest such integer. We say that x is *aperiodic* if $d(x) = 1$ and *periodic* if $d(x) > 1$.

If a (finite) chain satisfies both of these properties, it will always have a unique stationary distribution.

4.1.1 Metropolis-Hastings algorithm

The purpose of the Metropolis-Hastings algorithm is to generate a collection of states according to a desired distribution $p(x)$. More formally, let p be the target distribution defined up to a constant over a space \mathcal{X} . We wish to construct a Markov Chain with stationary distribution equal to the target distribution p . We can obtain samples by simulating a Markov process. Given an initial distribution π_0 and a transition matrix (kernel) K , we construct the following sequence of random variables

$$X_0 \sim \pi_0, \quad X_{t+1} \sim K(\cdot | X_t). \quad (4.5)$$

As discussed previously, for p to be the stationary distribution of the chain, K must be irreducible and aperiodic, and p has to be a *fixed point* of K . Note we can express the fixed point condition as

$$p(x') = \int K(x'|x)p(x)dx \quad (4.6)$$

We can ensure that stationarity is satisfied by requiring the (stronger) *detailed balance condition*¹

$$p(x')K(x|x') = p(x)K(x'|x). \quad (4.7)$$

We can rewrite the detailed balance condition as

$$\frac{K(x'|x)}{K(x|x')} = \frac{p(x')}{p(x)} \quad (4.8)$$

¹Note that this is equivalent to the reversibility condition defined above.

Given a proposal distribution $q(x'|x)$, we can construct a transition kernel satisfying detailed balance using Metropolis-Hastings accept/reject rules, as outlined in Alg. 2.

Algorithm 2: Metropolis-Hastings Algorithm

input : Transition kernel (proposal distribution), $q(x'|x)$

Initialize $x_0 \sim \pi_0$.

for $t = 0$ **to** N :

 Sample $x' \sim q(\cdot|x_t)$

 Compute the acceptance probability: $A(x'|x_t) = \min\left(1, \frac{p(x')q(x_t|x')}{p(x_t)q(x'|x_t)}\right)$

 With probability A accept the proposed value and set $x_{t+1} = x'$. Otherwise set

$x_{t+1} = x_t$.

In practice, a Gaussian distribution is often used as the proposal distribution, i.e. $q(x'|x) = \mathcal{N}(x'|x, \Sigma)$. In this case the acceptance probability reduces to

$$A(x'|x) = \min\left(1, \frac{p(x')}{p(x)}\right). \quad (4.9)$$

This ‘random walk’ approach is conceptually simple and can be easily implemented but performs poorly for the high-dimensional target distributions commonly encountered in lattice gauge theory and lattice QCD.

4.2 Aside: Bayesian Analysis

While not directly relevant to our discussion, it is interesting to note the connection between MCMC and Bayesian analysis, which has applications in a wide range of disciplines outside of physics [40]. Recall Bayes’ theorem:

$$P(B_i|A) = \frac{P(A|B_i)P(B_i)}{\sum_j P(A|B_j)P(B_j)} \quad (4.10)$$

Here, $p(A|B)$ is the conditional probability of A given B , i.e. the probability that A is true when the condition B is satisfied. For example, suppose we are interested in determining if an email is spam. The test for spam is that the message contains some flagged words B_i , i.e. $B_1 = \text{‘winner’}$, $B_2 = \text{‘Nigerian prince’}$, etc. In this case, $p(A|B_i)$ would then be the probability that the email is spam, given that it contains the word B_i . Suppose $p(A|B_i)$ and $p(B_i)$ are given (e.g.

$P(A|B_1) = 10^{-4}$, $P(A|B_2) = 0.05, \dots$) and we want to derive $p(B_i|A)$, the probability of an email containing the word B_i , given that it was classified as spam. To use an analogy from physics, we can identify B_i with some field ϕ , and $p(A|B_i)p(B_i)$ with the path integral weight $e^{-S[\phi]}[d\phi]$. The denominator $\sum_j p(A|B_i)p(B_i) = p(A)$ would then be regarded as the partition function Z . Using MCMC sampling, we can obtain $p(B_i|A) \sim \frac{e^{-S[\phi]}[d\phi]}{Z}$ via Bayes' theorem, i.e. we can collect many samples and see the resulting distribution.

4.3 Hamiltonian Monte Carlo

We can improve upon this random-walk guess and check strategy by “guiding” the simulation according to the systems natural dynamics using a method known as Hamiltonian (Hybrid) Monte Carlo (HMC). In HMC, model samples can be obtained by simulating a physical system governed by a Hamiltonian comprised of kinetic and potential energy functions that govern a particles dynamics. By transforming the density function to a potential energy function and introducing the auxiliary momentum variable v , HMC lifts the target distribution onto a joint probability distribution in phase space (x, v) , where x is the original variable of interest (e.g. position in Euclidean space). A new state is then obtained by solving the equations of motion for a fixed period of time using a volume-preserving integrator (most commonly the *leapfrog integrator*). The addition of random (typically normally distributed) momenta encourages long-distance jumps in state space with a single Metropolis-Hastings (MH) step. Let the ‘position’ of the physical state be denoted by a vector $x \in \mathbb{R}^n$ and the conjugate momenta of the physical state be denoted by a vector $v \in \mathbb{R}^n$. Then the Hamiltonian reads

$$\mathcal{H}(x, v) = U(x) + K(v) \tag{4.11}$$

$$= U(x) + \frac{1}{2}v^T v, \tag{4.12}$$

where $U(x)$ is the potential energy, and $K(v) = \frac{1}{2}v^T v$ the kinetic energy. We assume without loss of generality that the position and momentum variables are independently distributed. That is, we assume the target distribution of the system can be written as $p(x, v) = p(x)p(v)$. Further, instead of sampling $p(x)$ directly, HMC operates by sampling from the canonical distribution $p(x, v) =$

$\frac{1}{Z} \exp(-\mathcal{H}(x, v)) = p(x)p(v)$, for some partition function Z that provides a normalization factor. Additionally, we assume the momentum is distributed according to an identity-covariance Gaussian given by $p(v) \propto \exp(-\frac{1}{2}v^T v)$. For convenience, we will denote the combined state of the system by $\xi \equiv (x, v)$. From this augmented state ξ , HMC produces a proposed state $\xi' = (x', v')$ by approximately integrating Hamiltonian dynamics jointly on x and v . This integration is performed along approximate iso-probability contours of $p(x, v) = p(x)p(v)$ due to the Hamiltonians energy conservation.

4.3.1 Hamiltonian Dynamics

One of the characteristic properties of Hamilton's equations is that they conserve the value of the Hamiltonian. Because of this, every Hamiltonian trajectory is confined to an energy *level set*,

$$\mathcal{H}^{(-1)}(E) = \{x, v | \mathcal{H}(x, v) = E\}. \quad (4.13)$$

Our state $\xi = (x, v)$ then proceeds to explore this level set by integrating Hamilton's equations, which are shown as a system of differential equations in Eq. 4.15.

$$\dot{x}_i = \frac{\partial \mathcal{H}}{\partial v_i} = v_i \quad (4.14)$$

$$\dot{v}_i = -\frac{\partial \mathcal{H}}{\partial x_i} = -\frac{\partial U}{\partial x_i} \quad (4.15)$$

It can be shown [41] that the above transformation is volume-preserving and reversible, two necessary factors to guarantee asymptotic convergence of the simulation to the target distribution. The dynamics are simulated using the leapfrog integrator, which for a single time step consists of:

$$v^{\frac{1}{2}} = v - \frac{\varepsilon}{2} \partial_x U(x) \quad (4.16)$$

$$x' = x + \varepsilon v^{\frac{1}{2}} \quad (4.17)$$

$$v' = v - \frac{\varepsilon}{2} \partial_x U(x'). \quad (4.18)$$

We write the action of the leapfrog integrator in terms of an operator $\mathbf{L} : \mathbf{L}\xi \equiv \mathbf{L}(x, v) \equiv (x', v')$, and introduce a momentum flip operator $\mathbf{F} : \mathbf{F}(x, v) \equiv (x, -v)$. The Metropolis-Hastings acceptance probability for the HMC proposal is given by:

$$A(\mathbf{FL}\xi|\xi) = \min\left(1, \frac{p(\mathbf{FL}\xi)}{p(\xi)} \left| \frac{\partial[\mathbf{FL}\xi]}{\partial\xi^T} \right| \right), \quad (4.19)$$

Where $\left| \frac{\partial[\mathbf{FL}\xi]}{\partial\xi^T} \right|$ denotes the determinant of the Jacobian describing the transformation, and is equal to 1 for traditional HMC. In order to utilize these Hamiltonian trajectories to construct an efficient Markov transition, we need a mechanism for introducing momentum to a given point in the target parameter space. Fortunately, this can be done by exploiting the probabilistic structure of the system [42]. To lift an initial point in parameter space into one on phase space, we simply sample from the conditional distribution over the momentum,

$$v \sim \pi(x|v). \quad (4.20)$$

Sampling the momentum directly from the conditional distribution ensures that this lift will fall into the typical set in phase space. We can then proceed to explore the joint typical set by integrating Hamilton's equations as demonstrated above to obtain a new configuration $\xi \rightarrow \xi'$. We can then return to the target parameter space by simply projecting away the momentum,

$$(x, v) \rightarrow x \quad (4.21)$$

These three steps when performed in series gives a complete Hamiltonian Markov transition composed of random trajectories that rapidly explore the target distribution, as desired. An example of this process can be seen in Fig 4.1.

4.3.1.1 Properties of Hamiltonian Dynamics

There are three fundamental properties of Hamiltonian dynamics which are crucial to its use in constructing Markov Chain Monte Carlo updates.

1. **Reversibility:** Hamiltonian dynamics are *reversible* — the mapping from $\mathbf{L} : \xi(t) \rightarrow \xi' =$

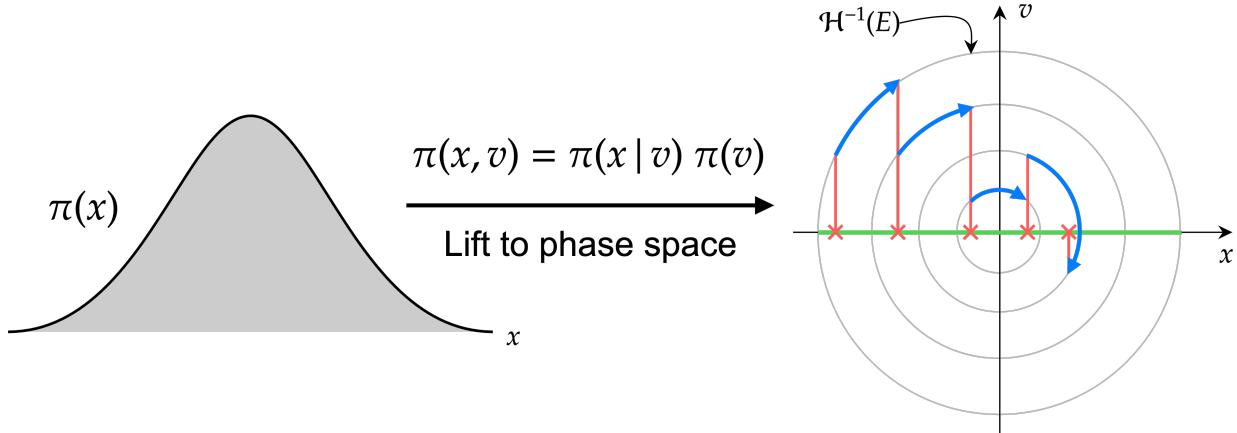


Figure 4.1: Visualizing HMC for a 1D Gaussian (example from [42]). Each Hamiltonian Markov transition lifts the initial state onto a random level set of the Hamiltonian, $\mathcal{H}^{(-1)}(E)$, which can then be explored with a [Hamiltonian trajectory](#) before [projecting back down](#) to the [target parameter space](#).

$\xi(t + s)$ is one-to-one, and consequently has an inverse \mathbf{L}^{-1} , obtained by negating the time derivatives in Eq. 4.15.

2. **Conservation of the Hamiltonian:** Moreover, the dynamics *keeps the Hamiltonian invariant*.
3. **Volume preservation:** The final property of Hamiltonian dynamics is that it *preserves volume* in (x, v) phase space (i.e. Liouville's Theorem).

All in all, HMC offers noticeable improvements compared to the ‘random-walk’ approach of generic MCMC, but tends to perform poorly on high-dimensional distributions. This becomes immediately apparent when it is used for simulations in lattice gauge theory and lattice QCD, where large autocorrelations and slow ‘burn-in’ can become prohibitively expensive.

5 | Semi-Supervised Learning: L2HMC

5.1 Introduction

We describe a new technique for performing Hamiltonian Monte-Carlo (HMC) simulations called: ‘Learning to Hamiltonian Monte Carlo’ (L2HMC) [1] which expands upon the traditional HMC by using a generalized version of the leapfrog integrator that is parameterized by weights in a neural network. In order to demonstrate the usefulness of this new approach, we use various metrics for measuring the performance of the trained (L2HMC) sampler vs. the generic HMC sampler. First, we will look at applying this algorithm to a two-dimensional Gaussian Mixture Model (GMM). The GMM is a notoriously difficult distribution for HMC due to the vanishingly small likelihood of the leapfrog integrator traversing the space between the two modes. Conversely, we see that through the use of a carefully chosen training procedure, the trained L2HMC sampler is able to successfully discover the existence of both modes, and mixes (‘tunnels’) between the two with ease. Additionally, we will observe that the trained L2HMC sampler mixes much faster than the generic HMC sampler, as evidenced through their respective autocorrelation spectra. This ability to reduce autocorrelations is an important metric for measuring the efficiency of a general MCMC algorithm, and is of great importance for simulations in lattice gauge theory and lattice QCD. Following this, we introduce the two-dimensional $U(1)$ lattice gauge theory and describe important modifications to the algorithm that are of particular relevance for lattice models. Ongoing issues and potential areas for improvement are also discussed, particularly within the context of high-performance computing and long-term goals of the lattice QCD community.

5.2 Generalizing the Leapfrog Integrator

As in the previously described HMC algorithm, we start by augmenting the current state $x \in \mathbb{R}^n$ with a continuous momentum variable $v \in \mathbb{R}^n$ drawn from a standard normal distribution. Additionally, we introduce a binary direction variable $d \in \{-1, 1\}$, drawn from a uniform distribution. The complete augmented state is then denoted by $\xi \equiv (x, v, d)$, with probability density $p(\xi) = p(x)p(v)p(d)$. To improve the overall performance of our model, for each step t of the leapfrog operator \mathbf{L}_θ , we assign a fixed random binary mask $m^t \in \{0, 1\}^n$ that will determine which variables are affected by each sub-update. The mask m^t is drawn uniformly from the set of binary vectors satisfying $\sum_{i=1}^n m_i^t = \lfloor \frac{n}{2} \rfloor$, i.e. half the entries of m^t are 0 and half are 1. Additionally, we write $\bar{m}^t = \mathbb{1} - m^t$ and $x_{m^t} = x \odot m^t$, where \odot denotes element-wise multiplication, and $\mathbb{1}$ the vector of 1's in each entry. We begin with a subset of the augmented space, $\zeta_1 \equiv (x, \partial_x U(x), t)$, independent of the momentum v . We introduce three new functions of ζ_1 : T_v , Q_v , and S_v . We can then perform a single time-step of our modified leapfrog integrator \mathbf{L}_θ .

First, we update the momentum v , which depends only on the subset ζ_1 . This update is written

$$v' = v \odot \underbrace{\exp\left(\frac{\varepsilon}{2} S_v(\zeta_1)\right)}_{\text{Momentum scaling}} - \frac{\varepsilon}{2} \left[\partial_x U(x) \odot \underbrace{\exp(\varepsilon Q_v(\zeta_1))}_{\text{Gradient scaling}} + \underbrace{T_v(\zeta_1)}_{\text{Translation}} \right]. \quad (5.1)$$

and the corresponding Jacobian is given by: $\exp\left(\frac{\varepsilon}{2} \mathbb{1} \cdot S_v(\zeta_1)\right)$. Next, we update x by first updating a subset of the coordinates of x (determined according to the mask m^t), followed by the complementary subset (determined from \bar{m}^t). The first update affects only x_{m^t} and produces x' . This update depends only on the subset $\zeta_2 \equiv (x_{\bar{m}^t}, v, t)$. Following this, we perform the second update which only affects $x'_{\bar{m}^t}$ and depends only on $\zeta_3 \equiv (x'_{m^t}, v, t)$, to produce x'' :

$$x' = x_{\bar{m}^t} + m^t \odot [x \odot \exp(\varepsilon S_x(\zeta_2)) + \varepsilon (v' \odot \exp(\varepsilon Q_x(\zeta_2)) + T_x(\zeta_2))] \quad (5.2)$$

$$x'' = x'_{m^t} + \bar{m}^t \odot [x' \odot \exp(\varepsilon S_x(\zeta_3)) + \varepsilon (v' \odot \exp(\varepsilon Q_x(\zeta_3)) + T_x(\zeta_3))]. \quad (5.3)$$

with Jacobians: $\exp(\varepsilon m^t \cdot S_x(\zeta_2))$, and $\exp(\varepsilon \bar{m}^t \cdot S_x(\zeta_3))$, respectively. Finally, we proceed to up-

date v again, using the subset $\zeta_4 \equiv (x'', \partial_x U'', t)$:

$$v'' = v' \odot \exp\left(\frac{\varepsilon}{2} S_v(\zeta_4)\right) - \frac{\varepsilon}{2} [\partial_x U \odot \exp(\varepsilon Q_v(\zeta_4)) + T_v(\zeta_4)]. \quad (5.4)$$

In order to build some intuition about each of these terms, we discuss below some of the subtleties contained in this approach and how they are (carefully) dealt with.

The first thing to notice about these equations is that if $S_i = Q_i = T_i = 0$ ($i = x, v$), we recover the previous equations for the generic leapfrog integrator (as we would expect since we are attempting to *generalize* HMC). We can also see a similarity between the equations for updating v and those for updating x : each update is generalized by *scaling* the previous value (v or x), and *scaling and translating* the updating value (either $\partial_x U(x)$ or x). It can be shown [1], that the scaling applied to the momentum in Eq 5.1 can enable, among other things, acceleration in low-density zones to facilitate mixing between modes, and that the scaling term applied to the gradient may allow better conditioning of the energy landscape (e.g., by learning a diagonal inertia tensor), or partial ignoring of the energy gradient for rapidly oscillating energies. Second, note that because the determinant of the Jacobian appears in the Metropolis-Hastings (MH) acceptance probability, we require the Jacobian of each update to be efficiently computable (i.e. independent of the variable actually being updated). For each of the momentum updates, the input is a subset $\zeta = (x, \partial_x U(x), t)$ of the augmented space and the associated Jacobian is $\exp\left(\frac{\varepsilon}{2} \mathbb{1} \cdot S_v(\zeta)\right)$ which is independent of v as desired. For the position updates however, things are complicated by the fact that the input ζ is x -dependent. In order to ensure that the Jacobian of the x update is efficiently computable, it is necessary to break the update into two parts following the approach outlined in *Real-valued Non-Volume Preserving transformations (RealNVP)* [43].

5.2.1 Metropolis-Hastings Accept/Reject

Written in terms of these transformations, the augmented leapfrog operator \mathbf{L}_θ consists of M sequential applications of the single-step leapfrog operator $\mathbf{L}_\theta \xi = \mathbf{L}_\theta(x, v, d) = (x''^{\times M}, v''^{\times M}, d)$, followed by the previously-defined momentum flip operator \mathbf{F} which flips the direction variable d , i.e. $\mathbf{F}\xi = (x, v, -d)$. Using these, we can express a complete molecular dynamics update step as

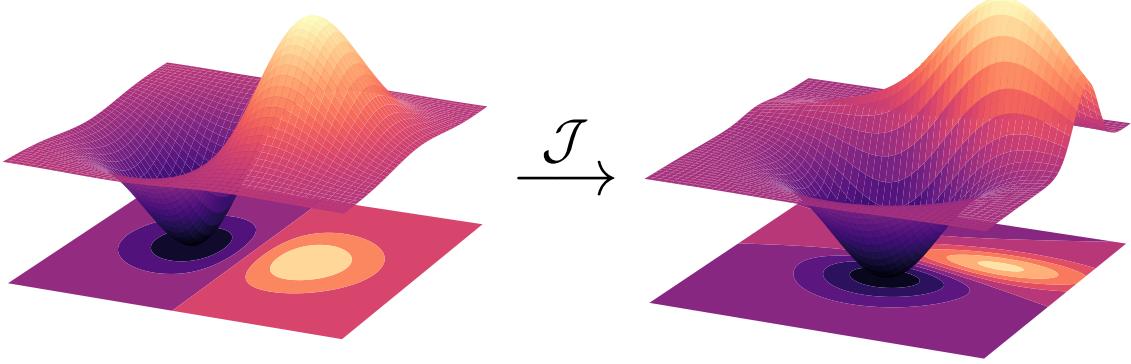


Figure 5.1: Example of how the determinant of the Jacobian can deform the energy landscape.

$\mathbf{FL}_\theta \xi = \xi'$, where now the Metropolis-Hastings acceptance probability for this proposal is given by

$$A(\mathbf{FL}\xi|\xi) = \min \left(1, \frac{p(\mathbf{FL}\xi)}{p(\xi)} \left| \frac{\partial[\mathbf{FL}\xi]}{\partial\xi^T} \right| \right), \quad (5.5)$$

Where $\left| \frac{\partial[\mathbf{FL}\xi]}{\partial\xi^T} \right|$ denotes the determinant of the Jacobian describing the transformation.

In contrast to generic HMC where $\left| \frac{\partial[\mathbf{FL}\xi]}{\partial\xi^T} \right| = 1$, we now have non-symplectic transformations (i.e. non-volume preserving) and so we must explicitly account for the determinant of the Jacobian. These non-volume preserving transformations have the effect of deforming the energy landscape, which, depending on the nature of the transformation, may allow for the exploration of regions of space which were previously inaccessible. To simplify our notation, introduce an additional operator \mathbf{R} that re-samples the momentum and direction, e.g. given $\xi = (x, v, d)$, $\mathbf{R}\xi = (x, v', d')$ where $v' \sim \mathcal{N}(0, I)$, $d' \sim \mathcal{U}(\{-1, 1\})$. A complete sampling step of our algorithm then consists of the following two steps:

1. $\xi' = \mathbf{FL}_\theta \xi$ with probability $A(\mathbf{FL}_\theta \xi|\xi)$ (Eq. 4.19), otherwise $\xi' = \xi$.
2. $\xi' = \mathbf{R}\xi$.

Note however, that for MH to be well-defined, this deterministic operator must be *invertible* and *have a tractable Jacobian* (i.e. we can compute its determinant). In order to make this operator invertible, we augment the state space (x, v) into (x, v, d) , where $d \in \{-1, 1\}$ is drawn with equal probability and represent the direction of the update. All of the previous expressions for the augmented leapfrog updates represent the forward ($d = 1$) direction. We can derive the expressions for

the backward direction ($d = -1$) by reversing the order of the updates (i.e. $v'' \rightarrow v'$, then $x'' \rightarrow x'$, followed by $x' \rightarrow x$ and finally $v' \rightarrow v$). For completeness, we include in Sec. 5.3 and Sec 5.4 all of the equations (both forward and backward directions) relevant for updating the variables of interest in our augmented leapfrog sampler.

5.3 Forward direction ($d = 1$):

$$v' = v \odot \exp\left(\frac{\varepsilon}{2} S_v(\zeta_1)\right) - \frac{\varepsilon}{2} [\partial_x U(x) \odot \exp(\varepsilon Q_v(\zeta_1)) + T_v(\zeta_1)] \quad (5.6)$$

$$x' = x_{\bar{m}^t} + m^t \odot [x \odot \exp(\varepsilon S_x(\zeta_2)) + \varepsilon (v' \odot \exp(\varepsilon Q_x(\zeta_2)) + T_x(\zeta_2))] \quad (5.7)$$

$$x'' = x'_{m^t} + \bar{m}^t \odot [x' \odot \exp(\varepsilon S_x(\zeta_3)) + \varepsilon (v' \odot \exp(\varepsilon Q_x(\zeta_3)) + T_x(\zeta_3))] \quad (5.8)$$

$$v'' = v' \odot \exp\left(\frac{\varepsilon}{2} S_v(\zeta_4)\right) - \frac{\varepsilon}{2} [\partial_x U(x'') \odot \exp(\varepsilon Q_v(\zeta_4)) + T_v(\zeta_4)] \quad (5.9)$$

With $\zeta_1 = (x, \partial_x U(x), t)$, $\zeta_2 = (x_{\bar{m}^t}, v, t)$, $\zeta_3 = (x'_{m^t}, v, t)$, $\zeta_4 = (x'', \partial_x U(x''), t)$.

5.4 Backward direction ($d = -1$):

$$v' = \left\{ v + \frac{\varepsilon}{2} [\partial_x U(x) \odot \exp(\varepsilon Q_v(\zeta_1)) + T_v(\zeta_1)] \right\} \odot \exp\left(-\frac{\varepsilon}{2} S_v(\zeta_1)\right) \quad (5.10)$$

$$x' = x_{m^t} + \bar{m}^t \odot [x - \varepsilon (\exp(\varepsilon Q_x(\zeta_2)) \odot v' + T_x(\zeta_2))] \odot \exp(-\varepsilon S_x(\zeta_2)) \quad (5.11)$$

$$x'' = x_{\bar{m}^t} + m^t \odot [x' - \varepsilon (\exp(\varepsilon Q_x(\zeta_3)) \odot v' + T_x(\zeta_3))] \odot \exp(-\varepsilon S_x(\zeta_3)) \quad (5.12)$$

$$v'' = \left\{ v' + \frac{\varepsilon}{2} [\partial_x U(x'') \odot \exp(\varepsilon Q_v(\zeta_1)) + T_v(\zeta_1)] \right\} \odot \exp\left(-\frac{\varepsilon}{2} S_v(\zeta_4)\right) \quad (5.13)$$

With $\zeta_1 = (x, \partial_x U(x), t)$, $\zeta_2 = (x_{m^t}, v, t)$, $\zeta_3 = (x'_{\bar{m}^t}, v, t)$, $\zeta_4 = (x'', \partial_x U(x''), t)$.

5.5 Determinant of the Jacobian

In terms of the auxiliary functions S_i, Q_i, T_i , we can compute the Jacobian:

$$\log |\mathcal{J}| = \log \left| \frac{\partial [\mathbf{FL}_\theta \xi]}{\partial \xi^T} \right| \quad (5.14)$$

$$= d \sum_{t \leq N_{LF}} \left[\frac{\varepsilon}{2} \mathbb{1} \cdot S_v(\zeta_1^t) + \varepsilon m^t \cdot S_x(\zeta_2^t) + \varepsilon \bar{m}^t \cdot S_x(\zeta_3^t) + \frac{\varepsilon}{2} \mathbb{1} \cdot S_v(\zeta_4^t) \right]. \quad (5.15)$$

where N_{LF} is the number of leapfrog steps, and ζ_i^t denotes the intermediary variable ζ_i^t at time step t and d is the direction of ξ , i.e. $d = 1$ (-1) for the forward (backward) update.

5.6 Network Architecture

As previously mentioned, each of the functions Q , S , and T , are implemented using multi-layer perceptrons with shared weights. It's important to note that we keep separate the network responsible for parameterizing the functions used in the position updates (' X_{net} ', i.e. Q_x, S_x , and T_x), and the network responsible for parameterizing the momentum updates (' V_{net} ', i.e. Q_v, S_v , and T_v). Since both networks are identical, we describe the architecture of V_{net} below, and include a flowchart for X_{net} illustrative purposes in Fig 5.3.

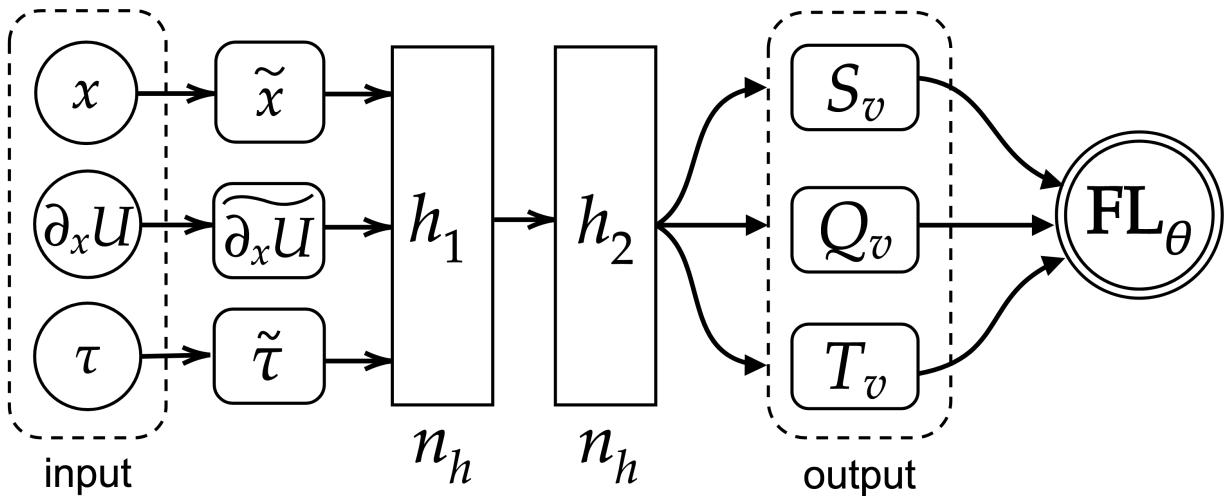


Figure 5.2: Illustration showing the generic (fully-connected) network architecture for training S_v , Q_v , and T_v .

The network takes as input $\zeta_1 = (x, \partial_x U(x), t)$, where $x, v \in \mathbb{R}^n$, and t is encoded as $\tau(t) = (\cos(\frac{2\pi t}{N_{\text{LF}}}), \sin(\frac{2\pi t}{N_{\text{LF}}}))$. Each of the inputs is then passed through a fully-connected ('dense' layer), consisting of n_h hidden units

$$\tilde{x} = W^{(x)}x + b^{(x)} \quad (\in \mathbb{R}^{n_h}) \quad (5.16)$$

$$\tilde{v} = W^{(v)}v + b^{(v)} \quad (\in \mathbb{R}^{n_h}) \quad (5.17)$$

$$\tilde{\tau} = W^{(\tau)}\tau + b^{(\tau)} \quad (\in \mathbb{R}^{n_h}). \quad (5.18)$$

Where $W^{(x)}, W^{(v)} \in \mathbb{R}^{n \times n_h}$, $W^{(t)} \in \mathbb{R}^{2 \times n_h}$, and $b^{(x)}, b^{(v)}, b^{(t)} \in \mathbb{R}^{n_h}$. From these, the network computes

$$h_1 = \sigma(\tilde{x} + \tilde{v} + \tilde{\tau}) \quad (\in \mathbb{R}^{n_h}). \quad (5.19)$$

Where $\sigma(x) = \max(0, x)$ denotes the rectified linear unit (ReLU) activation function. Next, the network computes

$$h_2 = \sigma(W^{(h_1)}h_1 + b^{(h_1)}) \quad (\in \mathbb{R}^{n_h}). \quad (5.20)$$

These weights (h_2) are then used to compute the network's output:

$$S_x = \lambda_s \tanh(W^{(s)}h_2 + b^{(s)}) \quad (\in \mathbb{R}^n) \quad (5.21)$$

$$Q_x = \lambda_q \tanh(W^{(q)}h_2 + b^{(q)}) \quad (\in \mathbb{R}^n) \quad (5.22)$$

$$T_x = W^{(T)}h_2 + b^{(T)} \quad (\in \mathbb{R}^n), \quad (5.23)$$

Where $W^{(s)}, W^{(q)}$, and $W^{(T)} \in \mathbb{R}^{n_h \times n}$ and $b^{(s)}, b^{(q)}$, and $b^{(T)} \in \mathbb{R}^n$. The parameters λ_s and λ_q are additional trainable variables initialized to zero. The network used for parameterizing the functions T_v , Q_v and S_v takes as input $(x, \partial_x U(x), t)$ where again t is encoded as above. The architecture of this network is the same, and produces outputs T_v , Q_v , and S_v .

5.7 Training Procedure

By augmenting traditional HMC methods with these trainable functions, we hope to obtain a sampler that has the following key properties:

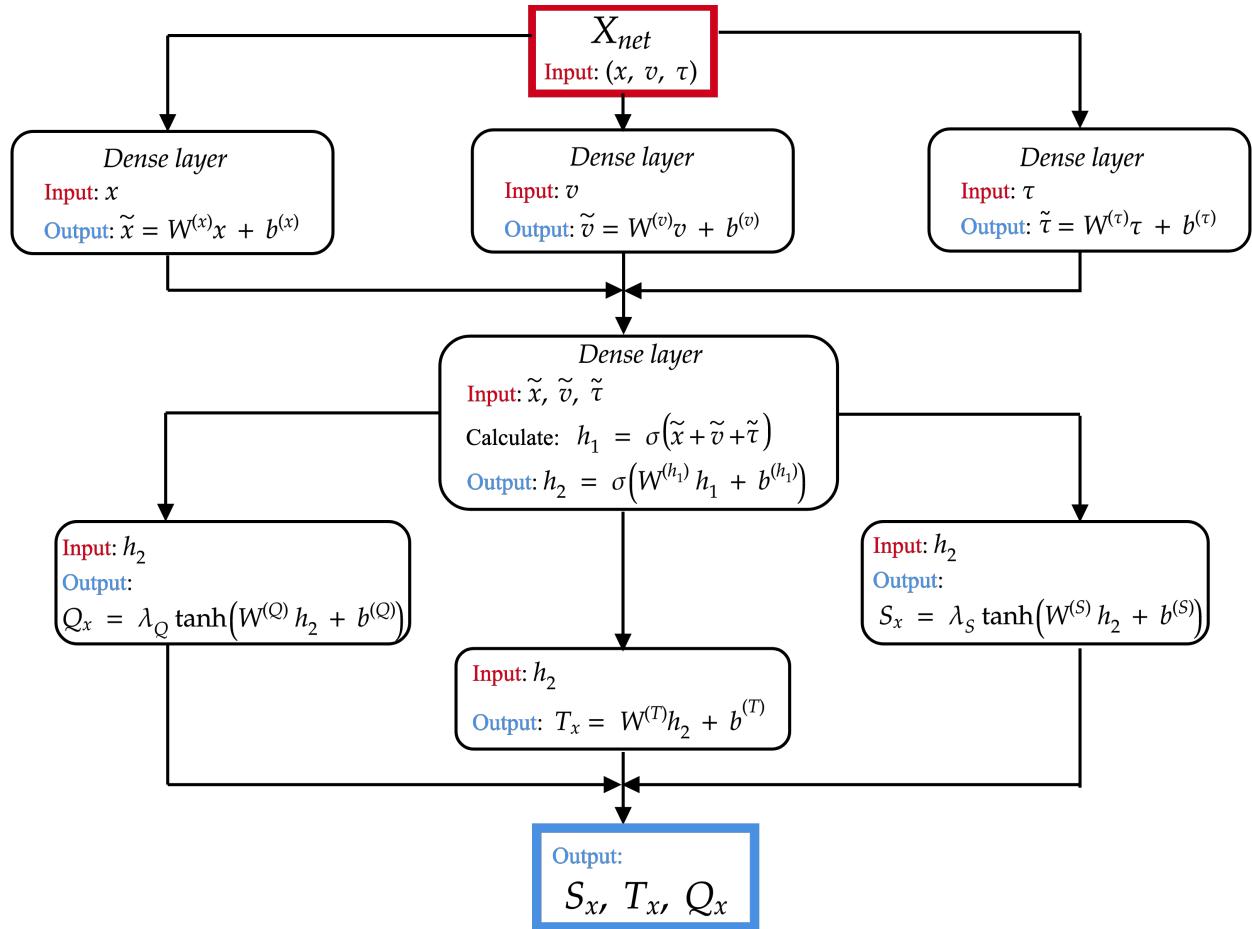


Figure 5.3: Flowchart illustrating the generic fully-connected network architecture including the intermediate variables computed at each hidden layer of the network.

1. Fast mixing (i.e. able to quickly produce uncorrelated samples).
2. Fast burn-in (i.e. rapid convergence to the target distribution).
3. Ability to mix across energy levels.
4. Ability to mix between modes.

Following the results in [44], we design a loss function with the goal of maximizing the expected squared jumped distance (or analogously, minimizing the lag-one autocorrelation). To do this, we first introduce

$$\delta(\xi, \xi') = \delta((x', v', d'), (x, v, d)) \equiv \|x - x'\|_2^2. \quad (5.24)$$

Then, the expected squared jumped distance is given by $\mathbb{E}_{\xi \sim p(\xi)} [\delta(\mathbf{FL}_\theta \xi, \xi) A(\mathbf{FL}_\theta \xi | \xi)]$. By maximizing this objective function, we are encouraging transitions that efficiently explore a local region of state-space, but may fail to explore regions where very little mixing occurs. To help combat this effect, we define a loss function

$$\ell_\lambda(\xi, \xi', A(\xi' | \xi)) = \frac{\lambda^2}{\delta(\xi, \xi') A(\xi' | \xi)} - \frac{\delta(\xi, \xi') A(\xi' | \xi)}{\lambda^2} \quad (5.25)$$

where λ is a scale parameter describing the characteristic length scale of the problem. Note that the first term helps to prevent the sampler from becoming stuck in a state where it cannot move effectively, and the second term helps to maximize the distance between subsequent moves in the Markov chain. The sampler is then trained by minimizing ℓ_λ over both the target and initialization distributions. Explicitly, for an initial distribution π_0 over \mathcal{X} , we define the initialization distribution as $q(\xi) = \pi_0(x) \mathcal{N}(v; 0, I)p(d)$, and minimize

$$\mathcal{L}(\theta) \equiv \mathbb{E}_{p(\xi)} [\ell_\lambda(\xi, \mathbf{FL}_\theta \xi, A(\mathbf{FL}_\theta \xi | \xi))] + \lambda_b \mathbb{E}_{q(\xi)} [\ell_\lambda(\xi, \mathbf{FL}_\theta \xi, A(\mathbf{FL}_\theta \xi | \xi))]. \quad (5.26)$$

For completeness, we include the full algorithm [1] used to train L2HMC in Alg. 3.

Algorithm 3: Training procedure for the L2HMC algorithm.

input :

1. A (potential) energy function, $U : \mathcal{X} \rightarrow \mathbb{R}$ and its gradient $\nabla_x U : \mathcal{X} \rightarrow \mathcal{X}$
2. Initial distribution over the augmented state space, q
3. Number of iterations, N_{train}
4. Number of leapfrog steps, N_{LF}
5. Learning rate schedule, $(\alpha_t)_{t \leq N_{\text{train}}}$
6. Batch size, N_{samples}
7. Scale parameter, λ
8. Regularization strength, λ_b

Initialize the parameters of the sampler, θ

Initialize $\{\xi_{p^{(i)}}\}_{i \leq N_{\text{samples}}}$ from $q(\xi)$

for $t = 0$ **to** N_{train} :

Sample a minibatch, $\{\xi_q^{(i)}\}_{i \leq N_{\text{samples}}}$ from $q(\xi)$.	$\mathcal{L} \leftarrow 0$ for $i = 1$ to N_{LF} : $\xi_p^{(i)} \leftarrow \mathbf{R} \xi_p^{(i)}$ $\mathcal{L} \leftarrow \mathcal{L} + \ell_\lambda(\xi_p^{(i)}, \mathbf{FL}_\theta \xi_p^{(i)}, A(\mathbf{FL}_\theta \xi_p^{(i)} \xi_p^{(i)})) + \lambda_b \ell_\lambda(\xi_q^{(i)}, \mathbf{FL}_\theta \xi_q^{(i)}, A(\mathbf{FL}_\theta \xi_q^{(i)} \xi_q^{(i)}))$ $\xi_p^{(i)} \leftarrow \mathbf{FL}_\theta \xi_p^{(i)}$ with probability $A(\mathbf{FL}_\theta \xi_p^{(i)} \xi_p^{(i)})$ $\theta \leftarrow \theta - \alpha_t \nabla_\theta \mathcal{L}$
---	--

5.8 Gaussian Mixture Model

The Gaussian Mixture Model (GMM) is a notoriously difficult example for traditional HMC to sample accurately due to the existence of multiple modes. In particular, HMC cannot mix between modes that are reasonably separated without recourse to additional tricks. This is due, in part, to the fact that HMC cannot easily traverse the low-density zones which exist between modes.

In the most general case, we consider a target distribution described by a mixture of $M > 1$ components in \mathbb{R}^D for $D \geq 1$:

$$p(\mathbf{x}) \equiv \sum_{m=1}^M p(m)p(\mathbf{x}|m) \equiv \sum_{m=1}^M \pi_m p(\mathbf{x}|m) \quad \forall \mathbf{x} \in \mathbb{R}^D \quad (5.27)$$

where $\sum_{m=1}^M \pi_m = 1$, $\pi_m \in (0, 1) \forall m = 1, \dots, M$ and each component distribution is a normal probability distribution in \mathbb{R}^D . So $\mathbf{x}|m \sim \mathcal{N}(\boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m)$, where $\boldsymbol{\mu}_m \equiv \mathbb{E}_{p(\mathbf{x}|m)} \{\mathbf{x}\}$ and $\boldsymbol{\Sigma}_m \equiv \mathbb{E}_{p(\mathbf{x}|m)} \{(\mathbf{x} - \boldsymbol{\mu}_m)(\mathbf{x} - \boldsymbol{\mu}_m)^T\} > 0$ are the mean vector and covariance matrix, respectively, of com-

ponent m .

5.8.1 Example

Consider a simple 2D case consisting of two Gaussians

$$\mathbf{x} \sim \pi_1 \mathcal{N}(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1) + \pi_2 \mathcal{N}(\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2) \quad (5.28)$$

with $\pi_1 = \pi_2 = 0.5$, $\boldsymbol{\mu}_1 = (-2, 0)$, $\boldsymbol{\mu}_2 = (2, 0)$ and

$$\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2 = \begin{bmatrix} 0.1 & 0 \\ 0 & 0.1 \end{bmatrix} \quad (5.29)$$

The results of trajectories generated using both traditional HMC and the L2HMC algorithm can be seen in Fig. 5.4. Note that traditional HMC performs poorly and is unable to mix between the two modes, whereas L2HMC is able to correctly sample from the target distribution without getting stuck in either of the individual modes.

The L2HMC sampler was trained using simulated annealing using the schedule shown in Eq 5.30 with a starting temperature of $T = 10$, for 5,000 training steps. By starting with a high temperature, the chain is able to move between both modes ('tunnel') successfully. Once it has learned this, we can lower the temperature back to $T = 1$ and recover the initial distribution while preserving information about tunneling in the networks "memory".

$$T(n) = (T_i - T_f) \cdot \left(1 - \frac{n}{N_{\text{train}}}\right) + T_f \quad (5.30)$$

5.9 2D $U(1)$ Lattice Gauge Theory

All lattice QCD simulations are performed at finite lattice spacing a and need an extrapolation to the continuum in order to be used for computing values of physical quantities. More reliable extrapolations can be done by simulating the theory at increasingly smaller lattice spacings. The

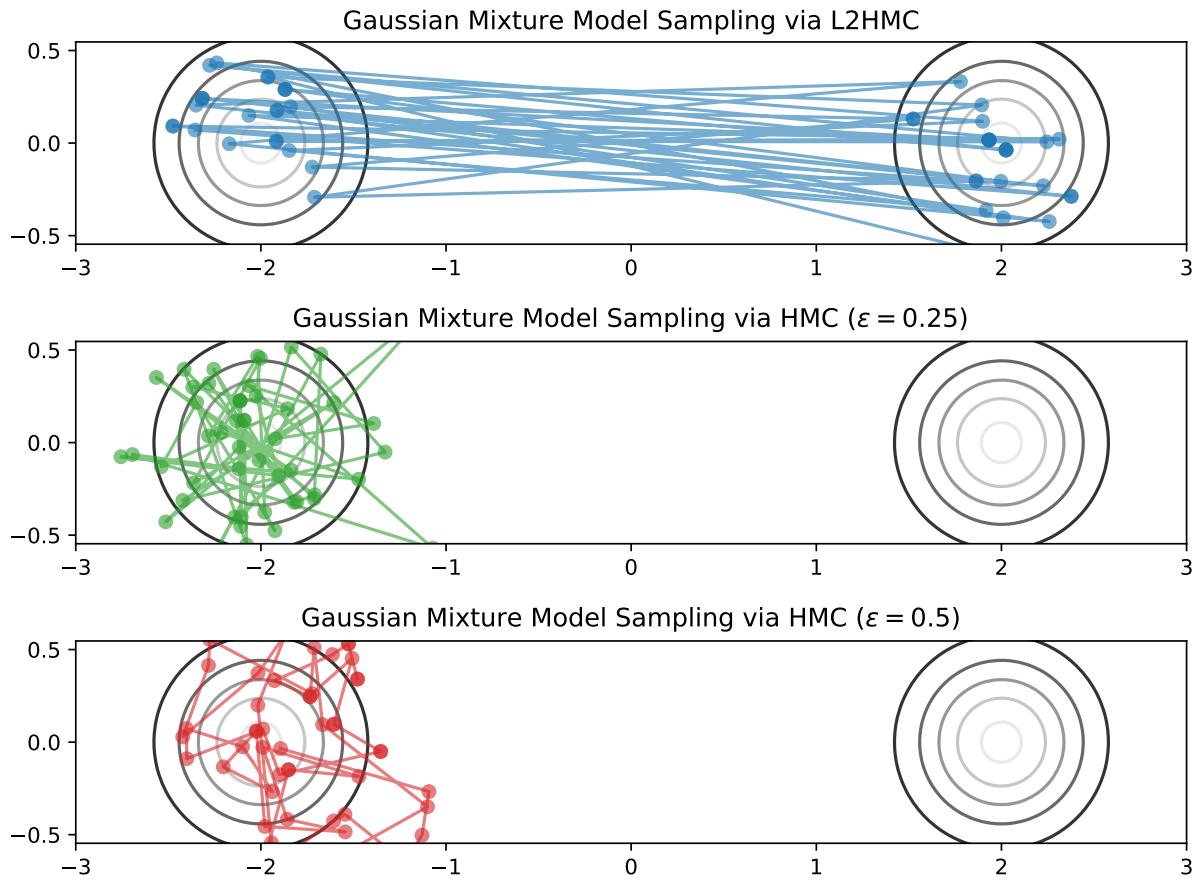


Figure 5.4: Comparison of trajectories generated using L2HMC (top), and traditional HMC with $\varepsilon = 0.25$ (middle) and $\varepsilon = 0.5$ (bottom). Note that L2HMC is able to successfully mix between modes, whereas HMC is not.

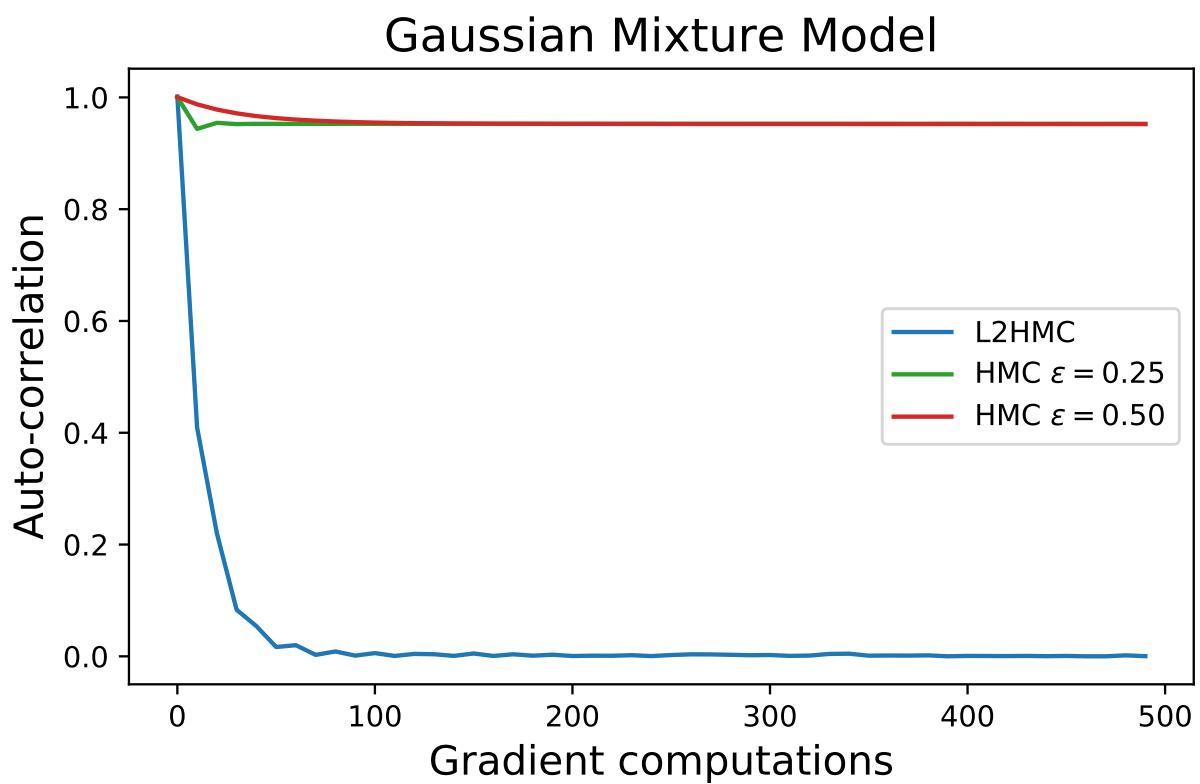


Figure 5.5: Autocorrelation vs. gradient evaluations (i.e. MD steps). Note that L2HMC (blue) has a significantly reduced autocorrelation after the same number of gradient evaluations when compared to either of the two HMC trajectories

picture that results when the lattice spacing is reduced and the physics kept constant is that all finite physical quantities of negative mass dimension diverge if measured in lattice units. In statistical mechanics language, this states that the continuum limit is a critical point of the theory since correlation lengths diverge. MCMC algorithms are known to encounter difficulties when used for simulating theories close to a critical point, an issue known as the *critical slowing down* of the algorithm. This effect is most prominent in the topological charge, whose auto-correlation time increases dramatically with finer lattice spacings. As a result, there is a growing interest in developing new sampling techniques for generating equilibrium configurations. In particular, algorithms that are able to offer improvements in efficiency through a reduction of statistical autocorrelations are highly desired. We begin with the two-dimensional $U(1)$ lattice gauge theory with dynamical variables $U_\mu(i)$ defined on the links of a lattice, where i labels a site and μ specifies the direction. Each link $U_\mu(i)$ can be expressed in terms of an angle $-\pi < \phi_\mu(i) \leq \pi$.

$$U_\mu(i) = e^{i\phi_\mu(i)} \quad (5.31)$$

with the Wilson action defined as:

$$\beta S = \beta \sum_P (1 - \cos(\phi_P)) \quad (5.32)$$

where

$$\phi_P \equiv \phi_{\mu\nu}(i) = \phi_\mu(i) + \phi_\nu(i + \hat{\mu}) - \phi_\mu(i + \hat{\nu}) - \phi_\nu(i) \quad (5.33)$$

and $\beta = 1/e^2$ is the gauge coupling, and the sum \sum_P runs over all plaquettes of the lattice. An illustration showing how these variables are defined for an elementary plaquette is shown in Fig. 5.6.

We can define the topological charge, $Q \in \mathbb{Z}$, as

$$Q \equiv \frac{1}{2\pi} \sum_P \tilde{\phi}_P = \frac{1}{2\pi} \sum_{\substack{i,\mu,\nu \\ \nu > \mu}} \tilde{\phi}_{\mu\nu}(i) \quad (5.34)$$

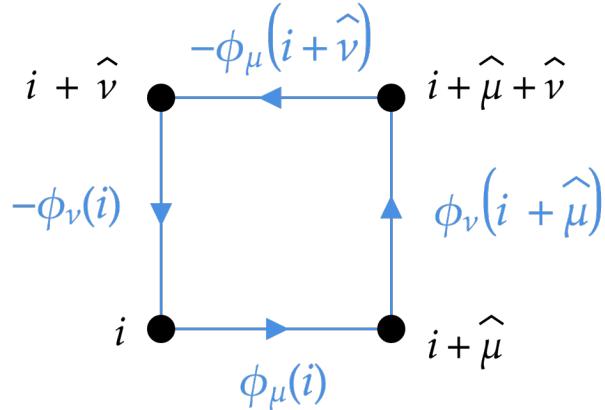


Figure 5.6: Illustration of an elementary plaquette on the lattice.

where

$$\tilde{\phi}_P \equiv \phi_P - 2\pi \left\lfloor \frac{\phi_P + \pi}{2\pi} \right\rfloor \quad (5.35)$$

is the sum of the link variables around the elementary plaquette, projected onto the interval $[-\pi, \pi]$.

From this, we can define topological susceptibility

$$\chi \equiv \frac{\langle Q^2 \rangle - \langle Q \rangle^2}{V} \quad (5.36)$$

By parity symmetry, $\langle Q \rangle = 0$, so we have that

$$\chi = \frac{\langle Q^2 \rangle}{V} \quad (5.37)$$

Unfortunately, the measurement of χ is often difficult due to the fact that the autocorrelation time with respect to Q tends to be extremely long. This is a consequence of the fact that the Markov chain tends to get stuck in a topological sector (characterized by $Q = const.$), a phenomenon known as *topological freezing*.

5.9.1 Modified Network Architecture

In order to better account for the rectangular geometry of the lattice, a stack of convolutional layers was prepended to the existing architecture, and can be seen in Fig. 5.8. The output from this convolutional structure is then fed to the generic network shown in Fig. 5.2. Additionally, the

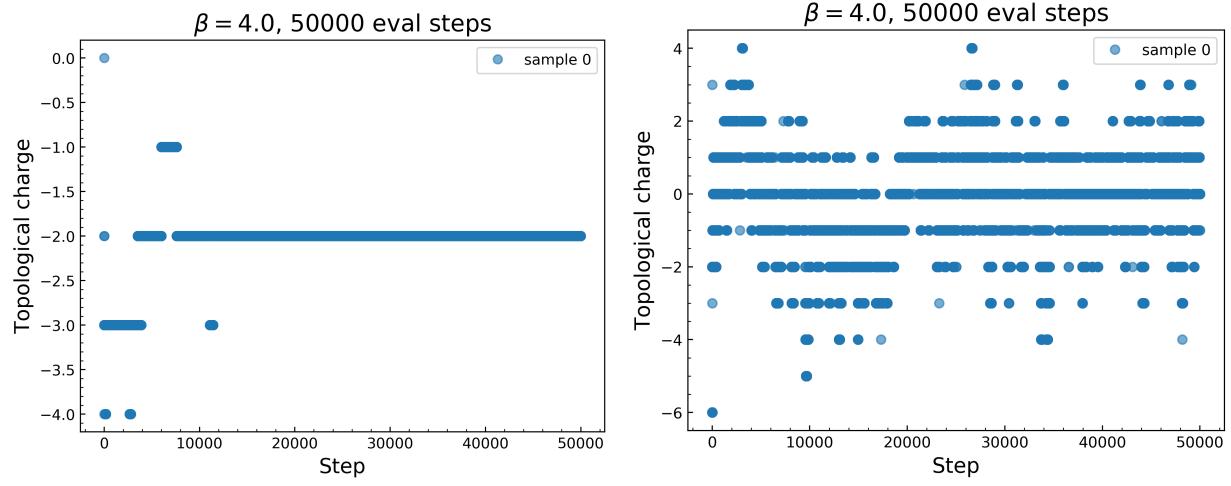


Figure 5.7: (left) Example of topological freezing in the 2D $U(1)$ lattice gauge theory, generated from generic HMC sampling for a 16×16 lattice. Note that for the majority of the simulation $Q = -2$, making it virtually impossible to get a reasonable estimate of χ . (right) Topological charge vs. step generated using the trained L2HMC sampler.

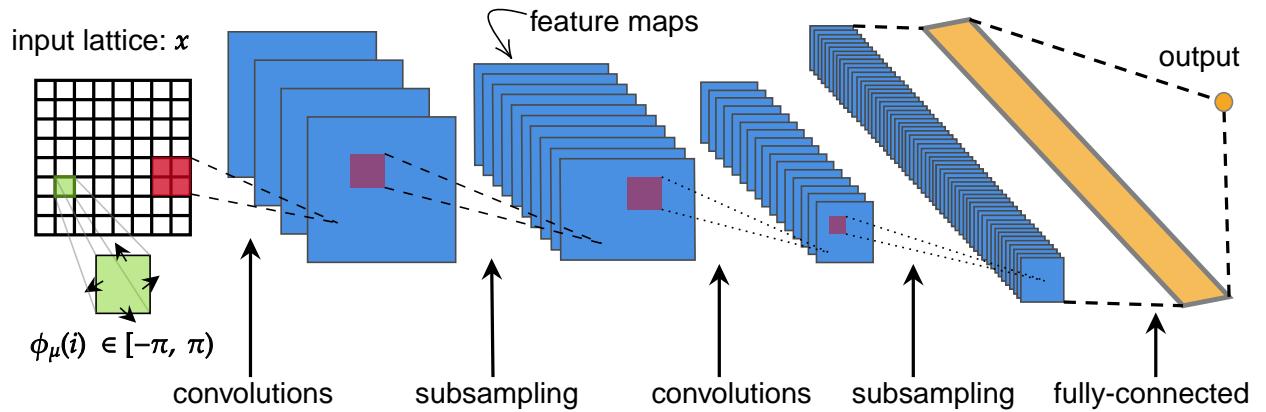


Figure 5.8: Convolutional structure used for learning localized features of rectangular lattice.

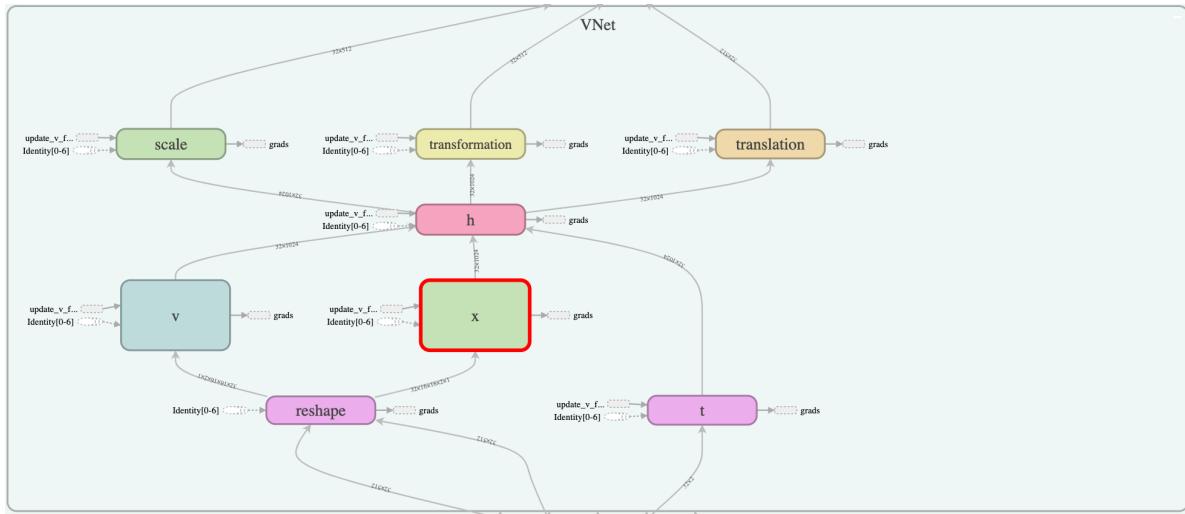


Figure 5.9: Illustration taken from TensorBoard showing an overview of the network architecture for VNet. Note that the architecture is identical for XNet.

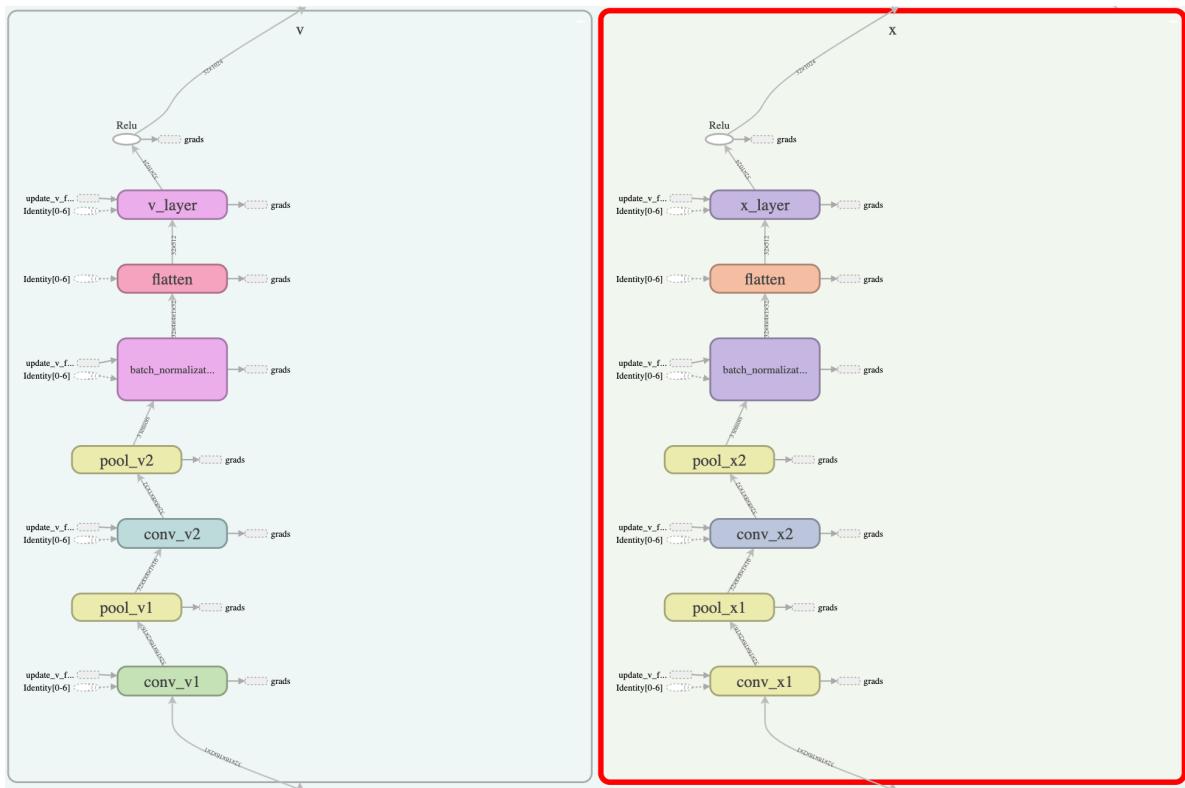


Figure 5.10: Detailed view of additional convolutional structure included to better account for rectangular geometry of lattice inputs.

network architecture was modified to include a batch normalization layer after the second MaxPool layer. Introducing batch normalization is a commonly used technique in practice, and is known to help prevent against diverging gradients¹, (an issue that was occasionally encountered during the training procedure). Additionally, it has been shown to improve model performance and generally requires fewer training steps to achieve similar performance as models trained without it [45].

5.9.2 Annealing Schedule

Proceeding as in the example of the Gaussian Mixture Model, we include a simulated annealing schedule in which the value of the gauge coupling β is continuously updated according to the annealing schedule shown in Eq. 5.38. This was done in order to encourage sampling from multiple different topological charge sectors, since our sampler is less ‘restricted’ at lower values of β .

$$\frac{1}{\beta(n)} = \left(\frac{1}{\beta_i} - \frac{1}{\beta_f} \right) \left(\frac{1-n}{N_{\text{train}}} \right) + \frac{1}{\beta_f} \quad (5.38)$$

Here $\beta(n)$ denotes the value of β to be used for the n^{th} training step ($n = 1, \dots, N_{\text{train}}$), β_i represents the initial value of β at the beginning of the training, and β_f represents the final value of β at the end of training. For a typical training session, $N_{\text{train}} = 25,000$, $\beta_i = 2$ and $\beta_f = 5$.

5.9.3 Modified loss function for $U(1)$ gauge model

In order to more accurately define the “distance” between two different lattice configurations, we redefine the metric in Eq. 5.24 to be

$$\delta(\xi, \xi') \equiv 1 - \cos(\xi - \xi') \quad (5.39)$$

where now $\xi = (\phi_\mu^x(i), \phi_\mu^v(i), d)$, with ϕ_μ^x representing the lattice of (‘position’) gauge variables (what we called x previously), and ϕ_μ^v representing the lattice of (‘momentum’) gauge variables (what we called v previously). Note that i runs over all lattice sites² and $\mu = 0, 1$ for the two dimensional case. We see that this metric gives the expected behavior, since $\delta \rightarrow 0$ for $\xi \approx \xi'$.

¹a numerical issue in which infinite values are generated when calculating the gradients in backpropagation

²In what follows, we will refrain from explicitly including the site index and make the assumption that it implicitly extends over all sites on the lattice.

While this new metric helps to better measure distances in this configuration space, it does nothing to encourage the exploration of different topological sectors since there may be configurations for which $\delta(\xi, \xi') \approx 1$ but $Q(\xi) = Q(\xi')$. In order to potentially address this issue, we modify the original loss function as follows.

First define $\xi' \equiv \mathbf{FL}_\theta \xi$ as the resultant configuration proposed by the augmented leapfrog integrator, and

$$\delta_Q(\xi, \xi') = |Q(\xi) - Q(\xi')| \quad (5.40)$$

$$\ell_Q(\xi, \xi', A(\mathbf{FL}_\theta \xi | \xi)) = \delta_Q(\xi, \xi') \times A(\xi' | \xi). \quad (5.41)$$

So we have that δ_Q measures the difference in topological charge between the initial and proposed configurations, and ℓ_Q gives the expected topological charge difference. Proceeding as before, we include an additional auxiliary term which is identical in structure to the one above, except the input is now a configuration of link variables ϕ_μ drawn from the initialization distribution q , which for our purposes was chosen to be the standard random normal distribution on $[0, 2\pi)$.

We can then write the topological loss term as

$$\mathcal{L}_Q(\theta) \equiv \mathbb{E}_{p(\xi)}[\ell_Q(\xi, \mathbf{FL}_\theta \xi, A(\mathbf{FL}_\theta \xi | \xi))] + \alpha_{\text{aux}} \mathbb{E}_{q(\xi)}[\ell_Q(\xi, \mathbf{FL}_\theta \xi, A(\mathbf{FL}_\theta \xi | \xi))] \quad (5.42)$$

If we denote the standard loss (with the modified metric function) defined in Eq. 5.26 as $\mathcal{L}_{\text{std}}(\theta)$, we can write the new total loss as a combination of these two terms,

$$\mathcal{L}(\theta) = \alpha_{\text{std}} \mathcal{L}_{\text{std}}(\theta) + \alpha_Q \mathcal{L}_Q(\theta) \quad (5.43)$$

where $\alpha_{\text{std}}, \alpha_Q$ are multiplicative factors that weigh the relative contributions to the total loss from the standard and topological loss terms respectively, and α_{aux} in Eq. 5.42 weighs the contribution of configurations drawn from the initialization distribution.

5.9.4 Issues with the Average Plaquette

Upon further testing, an issue was encountered in which the average plaquette $\langle \phi_P \rangle$ seems to converge to a value which is noticeably different from the expected value in the infinite volume limit. This behavior can be seen in Fig. 5.11, and seems to depend on both the number of augmented leapfrog steps used by our integrator, as well as the ‘strength’ of the topological loss term in Eq. 5.43. The parameters used in Fig 5.11 are as follows: $L = 8$, $N_{\text{LF}} = 7$, $\alpha_Q = 0.5$, and $N_{\text{samples}} = 128$, $N_{\text{train}} = 1 \times 10^4$. In Fig 5.12, the only change was the weight factor for the topological charge term in the loss function $\alpha_Q = 0$. In order to quantify this unexpected behavior, we can calculate the

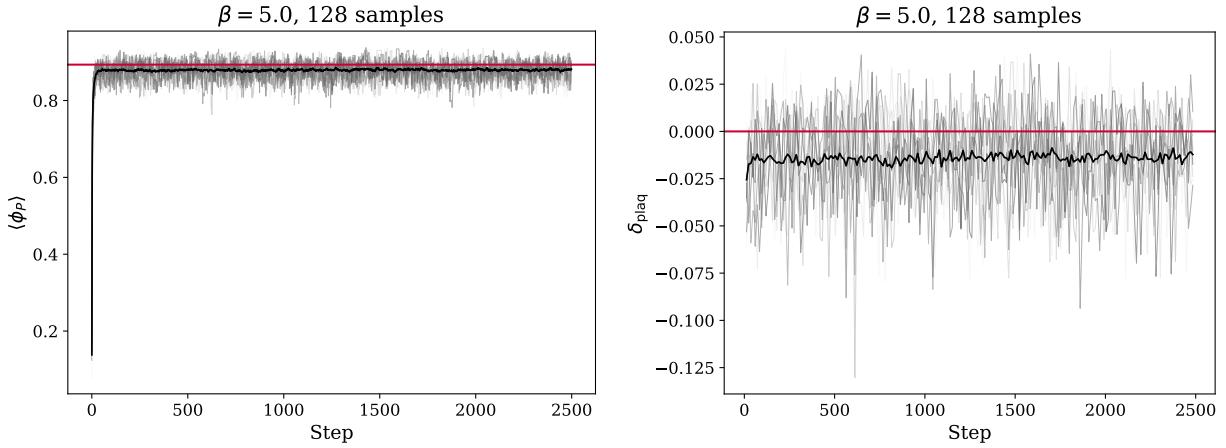


Figure 5.11: (left): Average plaquette $\langle \phi_P \rangle$ vs. step for $L = 8$, $N_{\text{LF}} = 7$, and $\alpha_Q = 0.5$. Here the solid red line indicates the true value of the average plaquette (in the infinite volume limit, and is equal to $0.89338\dots$) (right): Difference between the observed and expected value of the average plaquette δ_{ϕ_P} vs. step. Note that $\delta_{\phi_P} \neq 0$.

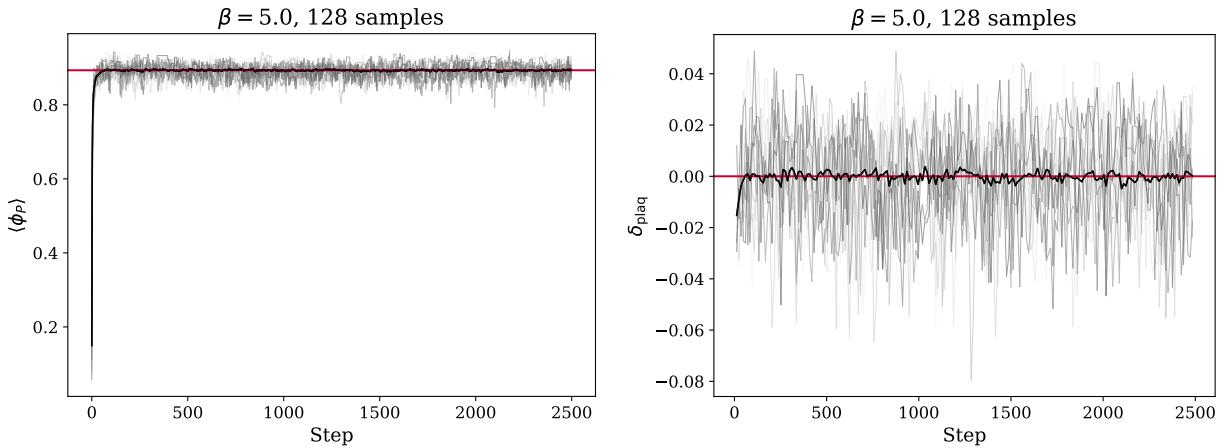


Figure 5.12: Same quantities as in Fig 5.11, with $\alpha_Q = 0$. Note that the discrepancy δ_{ϕ_P} is no longer present.

difference between the observed value of the average plaquette, $\langle \phi_P \rangle$ and the expected value $\phi_P^{(*)}$:

$$\delta_{\phi_P}(\alpha_Q, N_{\text{LF}}) \equiv \langle \phi_P \rangle - \phi_P^{(*)} \neq 0 \quad (5.44)$$

Which allows us to measure the severity of this discrepancy.

Initially it was believed that this behavior was due to the topological charge term in the loss function, however after testing using $\alpha_Q = 0$, this behavior was still present. Following this initial test, it was discovered that the discrepancy seemed to be more prevalent when using a larger number of leapfrog steps N_{LF} . In an additional systematic attempt to debug the problem, the sampler was trained and evaluated multiple times over a range of different values of both N_{LF} and α_Q . These results are included in Sec. B. Frustratingly, the issue seemed to be almost irreproducible. Running the training/evaluation processes multiple times using the exact same set of initial parameters on the exact same hardware it was observed that sometimes the discrepancy was present while other times it was not.

5.10 Conclusion

In conclusion, we have seen how the Hamiltonian Monte Carlo algorithm follows from the Metropolis-Hastings algorithm used in generic Markov Chain Monte Carlo methods, and why this approach is often preferred when attempting to sample from the high-dimensional distributions characteristic of lattice gauge theory models.

Following this, a detailed description of the Hamiltonian Monte Carlo algorithm was presented, as well as some of the common issues faced when it is applied to lattice quantum chromodynamics. In order to address some of these issues, we then proceeded to introduce a learned inference architecture that successfully generalized HMC by augmenting the traditional leapfrog integrator with a set of carefully-chosen functions which are parameterized by weights in a neural network. This system is then trained to learn a MCMC kernel that encourages fast mixing and convergence to the target distribution. While this transition kernel is no longer symplectic, we are able to retain the strong theoretical guarantee of HMC by enforcing a tractable MH accept/reject step, making L2HMC potentially capable of sampling from very challenging distributions. Having

introduced the necessary background, we then looked at applying this algorithm to the notoriously difficult two-dimensional Gaussian mixture model. In doing so, we found that the trained L2HMC sampler was capable of successfully mixing between modes in a way that traditional HMC was not. Additionally, we looked at the autocorrelation spectra of samples generated from the trained L2HMC sampler compared to those obtained from HMC. It was observed that the L2HMC sampler was able to produce samples which were noticeably less correlated in far fewer steps, indicating that the L2HMC algorithm significantly outperforms traditional HMC. In lattice QCD simulations, the ability of a sampler to quickly produce uncorrelated samples is one of the most important metrics for measuring its efficiency, and the approach outlined in this chapter shows promise in reducing the amount of computational resources required to generate new lattice configurations.

It remains to be seen how this algorithm performs when applied to more complicated models (e.g. models with fermions and non-Abelian gauge theories), a direction I plan to investigate more carefully in future research. The other, seemingly non-fundamental issue with this approach is the discrepancy between the observed and expected value of the average plaquette. As of now, I am inclined to believe that this is more of a ‘bug’ than an inherent problem with the algorithm itself.

MCMC methods have proven to be indispensable in exploring new physics beyond what can be achieved analytically, and has significantly advanced our understanding of quantum field theory and quantum chromodynamics. Unfortunately, in order to both perform and extract meaningful results from these simulations, tremendous amounts of computational resources are required, with cutting-edge simulations being carried out almost exclusively on some of the worlds largest supercomputers. Because of this, even minor improvements in efficiency can dramatically reduce the amount of computational power required, allowing for increasingly complex models to be studied. The pursuit of better, more efficient algorithms is one of the major long-term goals of the lattice community, and is directly aligned with many of the goals outlined for high energy physics in the era of exascale computing. In particular, the results of these simulations are of central importance to the experiments being carried out at the Relativistic Heavy Ion Collider at Brookhaven National Laboratory (BNL), and the Large Hadron Collider (LHC) at CERN.

A | Appendix: L2HMC Source Code

Included below is the source code used for building, training, running and analyzing the L2HMC algorithm on the $2D U(1)$ lattice gauge theory model. This code is also publicly hosted at <https://github.com/saforem2/l2hmc-qcd>.

A.1 README.md

A.1.1 l2hmc-qcd

Application of the L2HMC algorithm to simulations in lattice QCD. A description of the L2HMC algorithm can be found in the paper:

Generalizing Hamiltonian Monte Carlo with Neural Network
by Daniel Levy, Matt D. Hoffman and Jascha Sohl-Dickstein

A.1.1.1 Overview

NOTE: There are compatibility issues with `tensorflow.__version__ > 1.12` To be sure everything runs correctly, make sure `tensorflow==1.12.x` is installed.

Given an analytically described distribution¹, L2HMC enables training of fast-mixing samplers.

A.1.1.2 Modified implementation for Lattice Gauge Theory / Lattice QCD models.

This work is based on the original implementation which can be found at brain-research/l2hmc/.

My current focus is on applying this algorithm to simulations in lattice gauge theory and lattice QCD, in hopes of obtaining greater efficiency compared to generic HMC.

This new implementation includes the algorithm as applied to the $2D U(1)$ lattice gauge theory model (i.e. compact QED). Additionally, this implementation includes a convolutional neural network architecture that is prepended to the network described in the original paper. The purpose of this additional structure is to better incorporate information about the geometry of the lattice.

Lattice code can be found in `l2hmc-qcd/lattice/` and the particular code for the $2D U(1)$ lattice gauge model can be found in `l2hmc-qcd/lattice/lattice.py`.

¹simple examples can be found in `l2hmc-qcd/utils/distributions.py`

A.1.1.3 Features

This model can be trained using distributed training through `horovod`, by passing the `--horovod` flag as a command line argument.

A.1.1.4 Organization

To run `l2hmc-qcd/gauge_model_main.py` using one of the `.txt` files found in `l2hmc-qcd/args`, simply pass the `*.txt` file as the only command line argument prepended with `@`. For example, from within the `l2hmc-qcd/args` directory:

```
python3 ../gauge_model_main.py @gauge_model_args.txt
```

All of the relevant command line options are well documented and can be found in:²

```
l2hmc-qcd/utils/parse_args.py
```

Or by running `python3 gauge_model_main.py -help`.

Model information can be found in `l2hmc-qcd/models/gauge_model.py` which is responsible for building the graph and creating all the relevant tensorflow operations for training and running the L2HMC sampler.

The code responsible for actually implementing the L2HMC algorithm is divided up between `l2hmc-qcd/dynamics/gauge_dynamics.py` and `l2hmc-qcd/network/`.

The code responsible for performing the augmented leapfrog algorithm is implemented in the `GaugeDynamics` class defined in `l2hmc-qcd/dynamics/gauge_dynamics.py`.

There are multiple different neural network architectures defined in `l2hmc-qcd/network/` and different architectures can be specified as command line arguments defined in:

```
l2hmc-qcd/utils/parse_args.py
```

`l2hmc-qcd/notebooks/` contains a random collection of jupyter notebooks that each serve different purposes and should be somewhat self explanatory.

A.1.1.5 Contact

Code author: Sam Foreman

Pull requests and issues should be directed to: `saforem2`

A.1.1.6 Citation

If you use this code, please cite the original paper:

```
@article{levy2017generalizing,  
    title={Generalizing Hamiltonian Monte Carlo with Neural Networks},  
    author={Levy, Daniel and Hoffman, Matthew D. and Sohl-Dickstein, Jascha},  
    journal={arXiv preprint arXiv:1711.09268},  
    year={2017}  
}
```

²sample values for these arguments can be found in `l2hmc-qcd/args/gauge_model_args.txt`

A.2 l2hmc-qcd/args/gauge_model_args.txt

```
1 # =====#
2 #
3 # + All available command line arguments can be found (along with #
4 #   descriptions in the file: #
5 #     `utils/parse_args.py` . #
6 #
7 # + To pass the arguments defined below to `gauge_model.py` : #
8 #     `python3 gauge_model.py @gauge_model_args.txt` #
9 #
10# =====#
11
12--time_size 8
13--space_size 8
14--num_samples 128
15#--rand
16--num_steps 5
17--eps 0.2
18--annealing
19--beta_init 2.
20--beta_final 5.
21--lr_init 0.001
22--lr_decay_steps 1000
23--lr_decay_rate 0.96
24--train_steps 1000
25--run_steps 2000
26--save_steps 1000
27--print_steps 1
28--logging_steps 10
29--network_arch 'Conv2D'
30--metric 'cos_diff'
31--num_hidden 256
32--aux_weight 0.
33--std_weight 1.
34--charge_weight 0.
35--loss_scale 0.1
36--summaries
37--eps_trainable
38#--use_bn
39--save_lf
40--loop_net_weights
```

A.3 l2hmc-qcd/gauge_model_main.py

```
1 """
2 gauge_model_main.py
3
4 Main method implementing the L2HMC algorithm for a 2D U(1) lattice gauge theory
5 with periodic boundary conditions.
6
7 Following an object oriented approach, there are separate classes responsible
8 for each major part of the algorithm:
9
10    (1.) Creating the loss function to be minimized during training and
11        building the corresponding TensorFlow graph.
12
13        - This is done using the `GaugeModel` class, found in
14          `models/gauge_model.py`.
15
16        - The `GaugeModel` class depends on the `GaugeDynamics` class
17          (found in `dynamics/gauge_dynamics.py`) that performs the augmented
18          leapfrog steps outlined in the original paper.
19
20    (2.) Training the model by minimizing the loss function over both the
21        target and initialization distributions.
22        - This is done using the `GaugeModelTrainer` class, found in
23          `trainers/gauge_model_trainer.py` .
24
25    (3.) Running the trained sampler to generate statistics for lattice
26        observables.
27        - This is done using the `GaugeModelRunner` class, found in
28          `runners/gauge_model_runner.py` .
29
30 Author: Sam Foreman (github: @saforem2)
31 Date: 04/10/2019
32 """
33 import os
34 import random
35 import time
36 import tensorflow as tf
37 import numpy as np
38
39 from tensorflow.python import debug as tf_debug
40 from tensorflow.python.client import timeline
41 from tensorflow.core.protobuf import rewriter_config_pb2
42
43 import utils.file_io as io
44
45 from globals import GLOBAL_SEED, NP_FLOAT
46 from utils.parse_args import parse_args
47 from utils.model_loader import load_model
48 from models.gauge_model import GaugeModel
49 from loggers.train_logger import TrainLogger
50 from loggers.run_logger import RunLogger
51 from trainers.gauge_model_trainer import GaugeModelTrainer
52 from plotters.gauge_model_plotter import GaugeModelPlotter
53 from plotters.leapfrog_plotter import LeapfrogPlotter
54 from runners.gauge_model_runner import GaugeModelRunner
55
56 try:
57     import horovod.tensorflow as hvd
```

```

58     HAS_HOROVOD = True
59 except ImportError:
60     HAS_HOROVOD = False
61
62 try:
63     import matplotlib.pyplot as plt
64     HAS_MATPLOTLIB = True
65 except ImportError:
66     HAS_MATPLOTLIB = False
67
68 if float(tf.__version__.split('.')[0]) <= 2:
69     tf.logging.set_verbosity(tf.logging.INFO)
70
71 # -----
72 # Set random seeds for tensorflow and numpy
73 # -----
74 os.environ['PYTHONHASHSEED'] = str(GLOBAL_SEED)
75 random.seed(GLOBAL_SEED)          # `python` build-in pseudo-random generator
76 np.random.seed(GLOBAL_SEED)       # numpy pseudo-random generator
77 tf.set_random_seed(GLOBAL_SEED)
78
79
80 def create_config(FLAGS, params):
81     """Create tensorflow config."""
82     config = tf.ConfigProto()
83     if FLAGS.time_size > 8:
84         off = rewriter_config_pb2.RewriterConfig.OFF
85         config_attrs = config.graph_options.rewrite_options
86         config_attrs.arithmetic_optimization = off
87
88     if FLAGS.gpu:
89         # Horovod: pin GPU to be used to process local rank (one GPU per
90         # process)
91         config.gpu_options.allow_growth = True
92         # config.allow_soft_placement = True
93         if HAS_HOROVOD and FLAGS.horovod:
94             num_gpus = hvd.size()
95             io.log(f"Number of GPUs: {num_gpus}")
96             config.gpu_options.visible_device_list = str(hvd.local_rank())
97
98     if HAS_MATPLOTLIB:
99         params['_plot'] = True
100
101    if FLAGS.theta:
102        params['_plot'] = False
103        io.log("Training on Theta @ ALCF...")
104        params['data_format'] = 'channels_last'
105        os.environ["KMP_BLOCKTIME"] = str(0)
106        os.environ["KMP_AFFINITY"] = (
107            "granularity=fine,verbose,compact,1,0"
108        )
109        # NOTE: KMP affinity taken care of by passing -cc depth to aprun call
110        OMP_NUM_THREADS = 62
111        config.allow_soft_placement = True
112        config.intra_op_parallelism_threads = OMP_NUM_THREADS
113        config.inter_op_parallelism_threads = 0
114
115    return config, params
116
117

```

```

118 def count_trainable_params(out_file):
119     t0 = time.time()
120     io.log(f'Writing parameter counts to: {out_file}.')
121     io.log_and_write(80 * '-', out_file)
122     total_params = 0
123     for var in tf.trainable_variables():
124         # shape is an array of tf.Dimension
125         shape = var.get_shape()
126         io.log_and_write(f'var: {var}', out_file)
127         # var_shape_str = f' var.shape: {shape}'
128         io.log_and_write(f' var.shape: {shape}', out_file)
129         io.log_and_write(f' len(var.shape): {len(shape)}', out_file)
130         var_params = 1 # variable parameters
131         for dim in shape:
132             io.log_and_write(f'    dim: {dim}', out_file)
133             # dim_strs += f'    dim: {dim}\'
134             var_params *= dim.value
135             io.log_and_write(f'variable_parameters: {var_params}', out_file)
136             io.log_and_write(80 * '-', out_file)
137             total_params += var_params
138
139     io.log_and_write(80 * '-', out_file)
140     io.log_and_write(f'Total parameters: {total_params}', out_file)
141     t1 = time.time() - t0
142     io.log_and_write(f'Took: {t1} s to complete.', out_file)
143
144
145 def hmc(FLAGS, params=None, log_file=None):
146     """Create and run generic HMC sampler using trained params from L2HMC."""
147     condition1 = not FLAGS.horovod
148     condition2 = FLAGS.horovod and hvd.rank() == 0
149     is_chief = condition1 or condition2
150     if not is_chief:
151         return -1
152
153     FLAGS.hmc = True
154
155     FLAGS.log_dir = io.create_log_dir(FLAGS, log_file=log_file)
156
157     if params is None:
158         params = {}
159         for key, val in FLAGS.__dict__.items():
160             params[key] = val
161
162         params['hmc'] = True
163         params['use_bn'] = False
164         params['log_dir'] = FLAGS.log_dir
165         params['data_format'] = None
166         params['eps_trainable'] = False
167
168         figs_dir = os.path.join(params['log_dir'], 'figures')
169         io.check_else_make_dir(figs_dir)
170
171         io.log(80 * '-' + '\n')
172         io.log('HMC PARAMETERS:')
173         for key, val in params.items():
174             io.log(f' {key}: {val}')
175         io.log(80 * '-' + '\n')
176
177     # create tensorflow config (`config_proto`) to configure session

```

```

178     config, params = create_config(FLAGS, params)
179     tf.reset_default_graph()
180     sess = tf.Session(config=config)
181
182     model = GaugeModel(params=params)
183     run_logger = RunLogger(model, params['log_dir'], save_lf_data=False)
184     plotter = GaugeModelPlotter(model, run_logger.figs_dir)
185
186     sess.run(tf.global_variables_initializer())
187
188     runner = GaugeModelRunner(sess, model, run_logger)
189
190     if FLAGS.hmc_beta is None:
191         betas = [FLAGS.beta_final]
192     else:
193         betas = [float(FLAGS.hmc_beta)]
194
195     for beta in betas:
196         # to ensure hvd.rank() == 0
197         if run_logger is not None:
198             run_dir, run_str = run_logger.reset(model.run_steps, beta)
199
200         t0 = time.time()
201
202         runner.run(model.run_steps, beta)
203
204         run_time = time.time() - t0
205         io.log(f'Took: {run_time} s to complete run.')
206
207         if plotter is not None and run_logger is not None:
208             plotter.plot_observables(run_logger.run_data, beta, run_str)
209             if FLAGS.save_lf:
210                 lf_plotter = LeapfrogPlotter(plotter.out_dir, run_logger)
211                 lf_plotter.make_plots(run_dir, num_samples=20)
212
213     return sess, model, runner, run_logger
214
215
216 def train_l2hmc(FLAGS, log_file=None):
217     """Train L2HMC using GaugeModelTrainer."""
218     pass
219
220
221 def l2hmc(FLAGS, log_file=None):
222     """Create, train, and run L2HMC sampler on 2D U(1) gauge model."""
223     tf.keras.backend.clear_session()
224     tf.reset_default_graph()
225
226     FLAGS.log_dir = io.create_log_dir(FLAGS, log_file=log_file)
227
228     params = {}
229     for key, val in FLAGS.__dict__.items():
230         params[key] = val
231
232     if FLAGS.gpu:
233         io.log("Using GPU for training.")
234         params['data_format'] = 'channels_first'
235     else:
236         io.log("Using CPU for training.")
237         params['data_format'] = 'channels_last'

```

```

238
239     if FLAGS.horovod:
240         params['using_hvd'] = True
241         num_workers = hvd.size()
242         params['num_workers'] = num_workers
243         params['train_steps'] // num_workers
244         params['save_steps'] // num_workers
245         params['lr_decay_steps'] // num_workers
246         if params['summaries']:
247             params['logging_steps'] // num_workers
248         hooks = [
249             # Horovod: BroadcastGlobalVariablesHook broadcasts initial
250             # variable states from rank 0 to all other processes. This
251             # is necessary to ensure consistent initialization of all
252             # workers when training is started with random weights or
253             # restored from a checkpoint.
254             hvd.BroadcastGlobalVariablesHook(0),
255         ]
256         # params['run_steps'] // num_workers
257         # params['lr_init'] *= hvd.size()
258     else:
259         params['using_hvd'] = False
260         hooks = []
261
262     # conditionals required for file I/O
263     # if we're not using horovod, `is_chief` should always be True
264     # otherwise, if using Horovod, we only want to perform file I/O
265     # on hvd.rank() == 0, so check that first
266     condition1 = not FLAGS.horovod
267     condition2 = FLAGS.horovod and hvd.rank() == 0
268     is_chief = condition1 or condition2
269
270     if is_chief:
271         assert FLAGS.log_dir == params['log_dir']
272         log_dir = FLAGS.log_dir
273         checkpoint_dir = os.path.join(log_dir, 'checkpoints/')
274         io.check_else_make_dir(checkpoint_dir)
275
276     else:
277         log_dir = None
278         checkpoint_dir = None
279
280         io.log(80 * '-' + '\n')
281         io.log('L2HMC PARAMETERS:')
282         for key, val in params.items():
283             io.log(f' {key}: {val}')
284         io.log(80 * '-' + '\n')
285
286     model = GaugeModel(params=params)
287     if is_chief:
288         train_logger = TrainLogger(model, log_dir, FLAGS.summaries)
289         run_logger = RunLogger(model, train_logger.log_dir, save_lf_data=False)
290         plotter = GaugeModelPlotter(model, run_logger.figs_dir)
291     else:
292         train_logger = None
293         run_logger = None
294         plotter = None
295
296     config, params = create_config(FLAGS, params)
297     # sess = tf.Session(config=config)

```

```

298     tf.keras.backend.set_learning_phase(True)
299
300     # set initial value of charge weight using value from FLAGS
301     charge_weight_init = FLAGS.charge_weight
302     net_weights_init = [1., 1., 1.]
303     samples_init = np.reshape(np.array(model.lattice.samples, dtype=NP_FLOAT),
304                               (model.num_samples, model.x_dim))
305     beta_init = model.beta_init
306     init_feed_dict = {
307         model.x: samples_init,
308         model.beta: beta_init,
309         model.charge_weight: charge_weight_init,
310         model.net_weights[0]: net_weights_init[0], # scale_weight
311         model.net_weights[1]: net_weights_init[1], # transformation_weight
312         model.net_weights[2]: net_weights_init[2], # translation_weight
313     }
314     scaffold = tf.train.Scaffold(init_feed_dict=init_feed_dict)
315     # The MonitoredTrainingSession takes care of session
316     # initialization, restoring from a checkpoint, saving to a
317     # checkpoint, and closing when done or an error occurs.
318     sess = tf.train.MonitoredTrainingSession(
319         checkpoint_dir=checkpoint_dir,
320         scaffold=scaffold,
321         hooks=hooks,
322         config=config,
323         save_summaries_secs=None,
324         save_summaries_steps=None
325     )
326     trainer = GaugeModelTrainer(sess, model, train_logger)
327     kwargs = {
328         'samples_np': samples_init,
329         'beta_np': beta_init,
330         'charge_weight': charge_weight_init,
331         'net_weights': net_weights_init
332     }
333
334     try:
335         trainer.train(model.train_steps, **kwargs)
336     except NameError:
337         # i.e. Tensor had Inf / NaN values caused by high learning rate
338         io.log('\n\n' + 80 * '-')
339         io.log('Training crashed! Decreasing lr_init by 10% and retrying...')
340         io.log(f'Previous lr_init: {FLAGS.lr_init}')
341         FLAGS.lr_init
342         io.log(f'New lr_init: {FLAGS.lr_init}')
343         io.log('Restarting training...')
344         io.log(80 * '-' + '\n\n')
345         params['log_dir'] = FLAGS.log_dir = None
346         sess.close()
347         tf.keras.backend.clear_session()
348         tf.reset_default_graph()
349         l2hmc(FLAGS)
350
351     trainable_params_file = os.path.join(FLAGS.log_dir, 'trainable_params.txt')
352     count_trainable_params(trainable_params_file)
353
354     tf.keras.backend.set_learning_phase(False)
355
356     runner = GaugeModelRunner(sess, model, run_logger)
357     betas = [model.beta_final] # model.beta_final + 1]

```

```

358     if FLAGS.loop_net_weights:
359         net_weights_arr = np.array([[1, 1, 1], # [0, S, T]
360                                     [0, 1, 1],
361                                     [1, 0, 1],
362                                     [1, 1, 0],
363                                     [1, 0, 0],
364                                     [0, 1, 0],
365                                     [0, 0, 1],
366                                     [0, 0, 0]], dtype=NP_FLOAT)
367     else:
368         net_weights_arr = np.array([[1, 1, 1]], dtype=NP_FLOAT)
369
370     for net_weights in net_weights_arr:
371         weights = {
372             'charge_weight': charge_weight_init,
373             'net_weights': net_weights
374         }
375         for beta in betas:
376             if run_logger is not None:
377                 run_dir, run_str = run_logger.reset(model.run_steps, beta,
378                                                     **weights)
379             t0 = time.time()
380             runner.run(model.run_steps, beta, **weights)
381             run_time = time.time() - t0
382             io.log(80 * '-')
383             io.log(f'Took: {run_time} s to complete run.')
384             io.log(80 * '-')
385
386             if plotter is not None and run_logger is not None:
387                 plotter.plot_observables(run_logger.run_data,
388                                         beta, run_str, **weights)
389                 lf_plotter = LeapfrogPlotter(plotter.out_dir, run_logger)
390                 lf_plotter.make_plots(run_dir, num_samples=20)
391
392     return sess, model, train_logger
393
394
395 def run_hmc(FLAGS, params=None, log_file=None):
396     """Run generic HMC."""
397     condition1 = not FLAGS.horovod
398     condition2 = FLAGS.horovod and hvd.rank() == 0
399     is_chief = condition1 or condition2
400     io.log('\n' + 80 * '-')
401     io.log(("Running generic HMC algorithm "
402           "with learned parameters from L2HMC..."))
403     if is_chief:
404         hmc_sess, _, _, _ = hmc(FLAGS, params, log_file)
405         hmc_sess.close()
406         tf.reset_default_graph()
407
408
409 def run_l2hmc(FLAGS, log_file=None):
410     """Train and run L2HMC algorithm."""
411     io.log('\n' + 80 * '-')
412     io.log("Running L2HMC algorithm...")
413     l2hmc_sess, l2hmc_model, l2hmc_train_logger = l2hmc(FLAGS, log_file)
414     l2hmc_sess.close()
415     tf.reset_default_graph()
416
417     return l2hmc_model, l2hmc_train_logger

```

```

418
419
420 def main(FLAGS):
421     """Main method for creating/training/running L2HMC for U(1) gauge model."""
422     t0 = time.time()
423     if HAS_HOROVOD and FLAGS.horovod:
424         io.log("INFO: USING HOROVOD")
425         log_file = 'output_dirs.txt'
426         hvd.init()
427     else:
428         log_file = None
429
430     FLAGS.__dict__['while_loop'] = not FLAGS.for_loop
431
432     if FLAGS.hmc_eps is None:
433         eps_arr = [0.1, 0.15, 0.2, 0.25]
434     else:
435         eps_arr = [float(FLAGS.hmc_eps)]
436
437     if FLAGS.hmc:
438         run_hmc(FLAGS, log_file)
439         for eps in eps_arr:
440             FLAGS.eps = eps
441             run_hmc(FLAGS, log_file)
442
443     else:
444         model, logger = run_l2hmc(FLAGS, log_file)
445
446         if FLAGS.run_hmc:
447             # Run HMC with the trained step size from L2HMC (not ideal)
448             params = model.params
449             params['hmc'] = True
450             params['log_dir'] = FLAGS.log_dir = None
451             if logger is not None:
452                 params['eps'] = FLAGS.eps = logger._current_state['eps']
453             else:
454                 params['eps'] = FLAGS.eps
455
456             run_hmc(FLAGS, params, log_file)
457
458             for eps in eps_arr:
459                 params['log_dir'] = FLAGS.log_dir = None
460                 params['eps'] = FLAGS.eps = eps
461                 run_hmc(FLAGS, params, log_file)
462
463             io.log('\n\n')
464             io.log(80 * '-')
465             io.log(f'Time to complete: {time.time() - t0:.4g}')
466             io.log(80 * '-')
467
468
469     if __name__ == '__main__':
470         args = parse_args()
471         import pickle
472         args_file = 'args.pkl'
473         io.log(f'writing args to: {args_file}.')
474         with open('args.pkl', 'wb') as f:
475             pickle.dump(args.__dict__, f)
476         io.log('done.')
477         main(args)

```

A.4 l2hmc-qcd/models/gauge_model.py

```
1 """
2 gauge_model.py
3
4 Implements GaugeModel class responsible for building computation graph used in
5 tensorflow.
6
7 Author: Sam Foreman (github: @saforem2)
8 Date: 04/12/2019
9 """
10 from __future__ import absolute_import
11 from __future__ import division
12 from __future__ import print_function
13
14 import os
15
16 import numpy as np
17 import tensorflow as tf
18
19 try:
20     import horovod.tensorflow as hvd
21     HAS_HOROVOD = True
22 except ImportError:
23     HAS_HOROVOD = False
24
25 import utils.file_io as io
26
27 from globals import GLOBAL_SEED, TF_FLOAT
28 from lattice.lattice import GaugeLattice
29 from dynamics.gauge_dynamics import GaugeDynamics
30
31
32 def check_log_dir(log_dir):
33     """Check log_dir for existing checkpoints."""
34     assert os.path.isdir(log_dir)
35     ckpt_dir = os.path.join(log_dir, 'checkpoints')
36     if os.path.isdir(ckpt_dir):
37         pass
38
39
40 class GaugeModel:
41     def __init__(self, params=None):
42         # -----
43         # Create attributes from (key, val) pairs in params
44         # -----
45         self.loss_weights = {}
46         self.params = params
47         for key, val in self.params.items():
48             if 'weight' in key and key != 'charge_weight':
49                 self.loss_weights[key] = val
50             else:
51                 setattr(self, key, val)
52
53         # -----
54         # Create lattice
55         # -----
56         self.lattice, self.samples = self._create_lattice()
57
```

```

58     self.batch_size = self.lattice.samples.shape[0]
59     self.x_dim = self.lattice.num_links
60
61     # -----
62     # Create input placeholders:
63     #   (x, beta, charge_weight, net_weights)
64     # -----
65     inputs = self._create_inputs()
66     self.x = inputs['x']
67     self.beta = inputs['beta']
68     self.charge_weight = inputs['charge_weight']
69     self.net_weights = inputs['net_weights']
70
71     # -----
72     # Create dynamics engine
73     # -----
74     self.dynamics, self.potential_fn = self._create_dynamics(
75         self.lattice,
76         self.samples
77     )
78
79     # -----
80     # Create metric function used in loss
81     # -----
82     self.metric_fn = self._create_metric_fn(self.metric)
83
84     # -----
85     # Create operations for calculating plaquette observables
86     # -----
87     obs_ops = self._create_observables()
88     self.plaq_sums_op = obs_ops['plaq_sums']
89     self.actions_op = obs_ops['actions']
90     self.plaqs_op = obs_ops['plaqs']
91     self.avg_plaqs_op = obs_ops['avg_plaqs']
92     self.charges_op = obs_ops['charges']
93
94     # -----
95     # Create optimizer, build graph, create / init. saver
96     # -----
97     if self.hmc:
98         self.create_sampler()
99     else:
100        self._create_optimizer()
101        self.build()
102
103    def load(self, sess, checkpoint_dir):
104        latest_ckpt = tf.train.latest_checkpoint(checkpoint_dir)
105        if latest_ckpt:
106            io.log(f"INFO: Loading model from {latest_ckpt}...\n")
107            self.saver.restore(sess, latest_ckpt)
108            io.log("Model loaded.")
109
110    def _create_lattice(self):
111        """Create GaugeLattice object."""
112        with tf.name_scope('lattice'):
113            lattice = GaugeLattice(time_size=self.time_size,
114                                   space_size=self.space_size,
115                                   dim=self.dim,
116                                   link_type=self.link_type,
117                                   num_samples=self.num_samples,

```

```

118                     rand=self.rand)
119
120     assert lattice.samples.shape[0] == self.num_samples
121
122     samples = tf.convert_to_tensor(lattice.samples, dtype=TF_FLOAT)
123
124     return lattice, samples
125
126 def _create_observables(self):
127     """Create operations for calculating lattice observables."""
128     obs_ops = {}
129     with tf.name_scope('plaq_observables'):
130         with tf.name_scope('plaq_sums'):
131             obs_ops['plaq_sums'] = self.lattice.calc_plaq_sums(self.x)
132
133         with tf.name_scope('actions'):
134             obs_ops['actions'] = self.lattice.calc_actions(self.x)
135
136         with tf.name_scope('avg_plaqs'):
137             plaqs = self.lattice.calc_plaqs(self.x)
138             avg_plaqs = tf.reduce_mean(plaqs, name='avg_plaqs')
139
140             obs_ops['plaqs'] = plaqs
141             obs_ops['avg_plaqs'] = avg_plaqs
142
143         with tf.name_scope('top_charges'):
144             obs_ops['charges'] = self.lattice.calc_top_charges(self.x,
145                                                               fft=False)
146
147 #    return plaq_sums_op, actions_op, plaqs_op, charges_op
148
149     return obs_ops
150
151 def _create_inputs(self):
152     """Create input placeholders (if not executing eagerly).
153
154     Returns:
155         outputs: Dictionary with the following entries:
156             x: Placeholder for input lattice configuration with
157                 shape = (batch_size, x_dim) where x_dim is the number of
158                 links on the lattice and is equal to lattice.time_size *
159                 lattice.space_size * lattice.dim.
160             beta: Placeholder for inverse coupling constant.
161             charge_weight: Placeholder for the charge_weight (i.e. alpha_Q,
162                             the multiplicative factor that scales the topological
163                             charge term in the modified loss function) .
164             net_weights: Array of placeholders, each of which is a
165                         multiplicative constant used to scale the effects of the
166                         various S, Q, and T functions from the original paper.
167                         net_weights[0] = 'scale_weight', multiplies the S fn.
168                         net_weights[1] = 'transformation_weight', multiplies the Q
169                         fn. net_weights[2] = 'translation_weight', multiplies the
170                         T fn.
171             """
172
173     with tf.name_scope('inputs'):
174         if not tf.executing_eagerly():
175             x = tf.placeholder(dtype=TF_FLOAT,
176                               shape=(self.batch_size, self.x_dim),
177                               name='x_placeholder')
178             beta = tf.placeholder(dtype=TF_FLOAT,
179                                  shape=(),

```

```

178                               name='beta')
179     charge_weight = tf.placeholder(dtype=TF_FLOAT,
180                                     shape=(),
181                                     name='charge_weight')
182     net_weights = [
183         tf.placeholder(
184             dtype=TF_FLOAT, shape=(), name='scale_weight'
185         ),
186         tf.placeholder(
187             dtype=TF_FLOAT, shape=(), name='transformation_weight'
188         ),
189         tf.placeholder(
190             dtype=TF_FLOAT, shape=(), name='translation_weight'
191         )
192     ]
193   else:
194     x = tf.convert_to_tensor(
195       self.lattice.samples.reshape((self.batch_size, self.x_dim))
196     )
197     beta = tf.convert_to_tensor(self.beta_init)
198     charge_weight = tf.convert_to_tensor(0.)
199     net_weights = tf.convert_to_tensor([1., 1., 1.])
200
201   outputs = {
202     'x': x,
203     'beta': beta,
204     'charge_weight': charge_weight,
205     'net_weights': net_weights
206   }
207
208   return outputs
209
210 def _create_dynamics(self, lattice, samples, **kwargs):
211   """Initialize dynamics object."""
212   with tf.name_scope('dynamics'):
213     # default values of keyword arguments
214     dynamics_kwargs = {
215       'eps': self.eps,
216       'hmc': self.hmc,
217       'network_arch': self.network_arch,
218       'num_steps': self.num_steps,
219       'eps_trainable': self.eps_trainable,
220       'data_format': self.data_format,
221       'use_bn': self.use_bn,
222       'num_hidden': self.num_hidden,
223       # 'scale_weight': self.scale_weight,
224       # 'transformation_weight': self.transformation_weight,
225       # 'translation_weight': self.translation_weight
226     }
227
228     dynamics_kwargs.update(kwargs)
229     potential_fn = lattice.get_potential_fn(samples)
230     dynamics = GaugeDynamics(lattice=lattice,
231                             potential_fn=potential_fn,
232                             **dynamics_kwargs)
233
234   return dynamics, potential_fn
235
236 @staticmethod
237 def _create_metric_fn(metric):

```

```

238     """Create metric fn for measuring the distance between two samples."""
239     with tf.name_scope('metric_fn'):
240         if metric == 'l1':
241             def metric_fn(x1, x2):
242                 return tf.abs(x1 - x2)
243
244         elif metric == 'l2':
245             def metric_fn(x1, x2):
246                 return tf.square(x1 - x2)
247
248         elif metric == 'cos':
249             def metric_fn(x1, x2):
250                 return tf.abs(tf.cos(x1) - tf.cos(x2))
251
252         elif metric == 'cos2':
253             def metric_fn(x1, x2):
254                 return tf.abs(tf.cos(x1) - tf.cos(x2))
255
256         elif metric == 'cos_diff':
257             def metric_fn(x1, x2):
258                 return 1. - tf.cos(x1 - x2)
259         elif metric == 'tan_cos':
260             def metric_fn(x1, x2):
261                 cos_diff = 1. - tf.cos(x1 - x2)
262                 return tf.tan(np.pi * cos_diff / 2)
263         else:
264             raise AttributeError(f"""metric={metric}. Expected one of:
265                             'l1', 'l2', 'cos', 'cos2', 'cos_diff', or
266                             'tan_cos'.""")
267
268     return metric_fn
269
270 def _create_optimizer(self, lr_init=None):
271     """Create learning rate and optimizer."""
272     if self.hmc:
273         return
274
275     if lr_init is None:
276         lr_init = self.lr_init
277
278     with tf.name_scope('global_step'):
279         self.global_step = tf.train.get_or_create_global_step()
280         # self.global_step.assign(1)
281
282     with tf.name_scope('learning_rate'):
283         self.lr = tf.train.exponential_decay(lr_init,
284                                              self.global_step,
285                                              self.lr_decay_steps,
286                                              self.lr_decay_rate,
287                                              staircase=True,
288                                              name='learning_rate')
289
290     with tf.name_scope('optimizer'):
291         # Define update operations for batch normalization
292         self.update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
293         # Define optimizer
294         with tf.control_dependencies(self.update_ops):
295             self.optimizer = tf.train.AdamOptimizer(self.lr)
296             if self.using_hvd:
297                 self.optimizer = hvd.DistributedOptimizer(self.optimizer)

```

```

298     def _calc_std_loss(self, config_init, config_proposed, accept_prob):
299         """Calculate the (individual) standard contribution to the loss fn."""
300         eps = 1e-4
301         with tf.name_scope('_std_loss'):
302             summed_diff = tf.reduce_sum(
303                 self.metric_fn(config_init, config_proposed), axis=1
304             )
305
306         return summed_diff * accept_prob + eps
307
308     def calc_std_loss(self, inputs, **weights):
309         """Calculate the standard contribution to the loss.
310
311         Explicitly: This is calculated as the expectation value of the jump
312         loss over both the target and initialization distributions.
313
314         Args:
315             x_tup: Tuple containing (x_init, x_proposed)
316             z_tup: Tuple containing (z_init, z_proposed) (aux. variable)
317             p_tup: Tuple containing (px, pz) acceptance probabilities.
318             **weights: Dictionary of weights used as multiplicative scaling
319                         factors to control the contribution from individual terms to
320                         the total loss.
321
322         Returns:
323             std_loss: Tensorflow operation responsible for calculating the
324                         standard contribution to the loss function.
325
326         """
327
328         aux_weight = weights.get('aux_weight', 1.)
329         std_weight = weights.get('std_weight', 1.)
330
331         x_init = inputs.get('x_init', None)
332         x_proposed = inputs.get('x_proposed', None)
333
334         z_init = inputs.get('z_init', None)
335         z_proposed = inputs.get('z_proposed', None)
336
337         px = inputs.get('px', None)
338         pz = inputs.get('pz', None)
339
340         with tf.name_scope('x_std_loss'):
341             x_loss = self._calc_std_loss(x_init, x_proposed, px)
342             x_loss_tot = tf.reduce_mean((self.loss_scale / x_loss
343                                         - x_loss / self.loss_scale),
344                                         axis=0, name='x_std_loss_tot')
345             tf.add_to_collection('losses', x_loss_tot)
346
347         if aux_weight > 0 and z_init is not None:
348             with tf.name_scope('z_std_loss'):
349                 z_loss = self._calc_std_loss(z_init, z_proposed, pz)
350                 z_loss_tot = tf.reduce_mean((self.loss_scale / z_loss
351                                             - z_loss / self.loss_scale),
352                                             axis=0, name='z_std_loss_tot')
353                 tf.add_to_collection('losses', z_loss_tot)
354
355         else:
356             z_loss_tot = 0.
357
358         with tf.name_scope('std_loss'):
359             loss_tot = tf.multiply(std_weight,
360                                   (x_loss_tot + aux_weight * z_loss_tot),
361                                   name='std_loss_tot')

```

```

358     tf.add_to_collection('losses', loss_tot)
359
360     return loss_tot
361
362 def _calc_charge_loss(self, config_init, config_proposed, accept_prob):
363     """Calculate the (individual) top. charge contribution to the loss fn
364
365     Args:
366         config_init: Initial configuration.
367         config_proposed: Proposed configuration.
368         accept_prob: Likelihood of accepting the proposed configuration,
369             used to calculate the expectation value of the topological
370             charge difference.
371     Returns:
372         charge_loss: Individual contribution to the loss function from the
373             topological charge difference between the initial and proposed
374             configurations.
375     """
376     with tf.name_scope('charge_diff'):
377         charge_diff = self.lattice.calc_top_charges_diff(config_init,
378                                         config_proposed,
379                                         fft=True)
380
381     return accept_prob * charge_diff
382
383 def calc_charge_loss(self, inputs, **weights):
384     """Calculate the (individual) top. charge contribution to the loss fn
385
386     Args:
387         inputs: Dictionary containing initial and proposed configurations
388             sampled from both the target ('x_init', 'x_proposed') and
389             initialization ('z_init', 'z_proposed') distributions, as well
390             as the calculated acceptance probabilities ('px', and 'pz').
391         weights: Dictionary containing various multiplicative weights used
392             to scale the contribution from individual terms to the total
393             loss function
394     Returns:
395         charge_loss: Total contribution to the loss function from the
396             topological charge difference between initial and proposed
397             configuratons.
398     """
399     x_init = inputs.get('x_init', None)
400     x_proposed = inputs.get('x_proposed', None)
401
402     z_init = inputs.get('z_init', None)
403     z_proposed = inputs.get('z_proposed', None)
404
405     px = inputs.get('px', None)
406     pz = inputs.get('pz', None)
407
408     aux_weight = weights.get('aux_weight', 1.)
409
410     # Calculate the difference in topological charge between the initial
411     # and proposed configurations multiplied by the probability of
412     # acceptance to get the expected value of the difference in top. charge
413     with tf.name_scope('x_charge_loss'):
414         x_charge_loss = self._calc_charge_loss(x_init, x_proposed, px)
415         tf.add_to_collection('losses', x_charge_loss)
416
417     with tf.name_scope('z_charge_loss'):
418         if aux_weight > 0 and z_init is not None:

```

```

418         z_charge_loss = self._calc_charge_loss(z_init, z_proposed, pz)
419     else:
420         z_charge_loss = 0.
421     tf.add_to_collection('losses', x_charge_loss)
422
423     with tf.name_scope('total_charge_loss'):
424         charge_loss = self.charge_weight * (x_charge_loss + z_charge_loss)
425         charge_loss = tf.reduce_mean(charge_loss, axis=0,
426                                     name='charge_loss')
427
428         tf.add_to_collection('losses', charge_loss)
429
430     return charge_loss
431
432 def calc_loss(self, x, beta, net_weights, **weights):
433     """Create operation for calculating the loss.
434
435     Args:
436         x: Input tensor of shape (self.num_samples,
437             self.lattice.num_links) containing batch of GaugeLattice links
438             variables.
439         beta (float): Inverse coupling strength.
440         net_weights: Tuple containing multiplicative weights that scale the
441             contributions from the scale (S), transformation (Q) and
442             translation (T) functions.
443         weights: Dictionary containing various multiplicative weights used
444             to scale the contribution from individual terms to the total
445             loss function
446
447     Returns:
448         loss (float): Operation responsible for calculating the total loss.
449         px (np.ndarray): Array of acceptance probabilities from
450             Metropolis-Hastings accept/reject step. Has shape:
451             (self.num_samples,)
452         x_out: Output samples obtained after Metropolis-Hastings
453             accept/reject step.
454
455     NOTE: If proposed configuration is accepted following
456         Metropolis-Hastings accept/reject step, x_proposed and x_out are
457         equivalent.
458     """
459     with tf.name_scope('x_update'):
460         x_dynamics_output = self.dynamics(x, beta, net_weights,
461                                             while_loop=self.while_loop,
462                                             v_in=None, # v_in
463                                             save_lf=self.save_lf)
464         # x_proposed = tf.mod(dynamics_output['x_proposed'], 2 * np.pi)
465         # x_out = tf.mod(x_dynamics_output['x_out'], 2 * np.pi)
466         x_proposed = x_dynamics_output['x_proposed']
467         px = x_dynamics_output['accept_prob']
468         x_out = x_dynamics_output['x_out']
469
470     # Auxiliary variable
471     if weights['aux_weight'] > 0:
472         with tf.name_scope('z_update'):
473             z = tf.random_normal(tf.shape(x), seed=GLOBAL_SEED, name='z')
474             z_dynamics_output = self.dynamics(
475                 z, beta, net_weights,
476                 while_loop=self.while_loop,
477                 v_in=None, save_lf=False

```

```

478         )
479         z_proposed = z_dynamics_output['x_proposed']
480         # z_proposed = tf.mod(z_dynamics_output['x_proposed'],
481         #                         2 * np.pi)
482         pz = z_dynamics_output['accept_prob']
483     else:
484         z = None
485         z_proposed = None
486         pz = None
487
488     with tf.name_scope('top_charge_diff'):
489         x_dq = tf.cast(
490             self.lattice.calc_top_charges_diff(x, x_out, fft=False),
491             dtype=tf.int32
492         )
493
494     # NOTE:
495     # std_loss: 'standard' loss
496     # charge_loss: Contribution from the difference in topological charge
497     # between the initial and proposed configurations to the total
498     # loss.
499     inputs = {
500         'x_init': x,
501         'x_proposed': x_proposed,
502         'px': px,
503         'z_init': z,
504         'z_proposed': z_proposed,
505         'pz': pz
506     }
507     with tf.name_scope('calc_loss'):
508         with tf.name_scope('std_loss'):
509             std_loss = self.calc_std_loss(inputs, **weights)
510         with tf.name_scope('charge_loss'):
511             charge_loss = self.calc_charge_loss(inputs, **weights)
512
513         total_loss = tf.add(std_loss, charge_loss, name='total_loss')
514         tf.add_to_collection('losses', total_loss)
515
516     return total_loss, x_dq, x_dynamics_output
517
518 def calc_loss_and_grads(self, x, beta, net_weights, **weights):
519     """Calculate loss its gradient with respect to all trainable variables.
520
521     Args:
522         x: Placeholder (tensor object) representing batch of GaugeLattice
523             link variables.
524         beta: Placeholder (tensor object) representing inverse coupling
525             strength.
526
527     Returns:
528         loss: Operation for calculating the total loss.
529         grads: Tensor containing the gradient of the loss function with
530             respect to all trainable variables.
531         x_out: Operation for obtaining new samples (i.e. output of
532             augmented L2HMC algorithm.)
533         accept_prob: Operation for calculating acceptance probabilities
534             used in Metropolis-Hastings accept reject.
535         x_dq: Operation for calculating the topological charge difference
536             between the initial and proposed configurations.
537     """

```

```

538     # TODO: Fix eager execution logic to deal with self.lf_out
539     with tf.name_scope('grads'):
540         if tf.executing_eagerly():
541             with tf.GradientTape() as tape:
542                 loss, x_dq, dynamics_output = self.calc_loss(
543                     x, beta, net_weights, **weights
544                 )
545                 grads = tape.gradient(loss, self.dynamics.trainable_variables)
546             else:
547                 loss, x_dq, dynamics_output = self.calc_loss(x, beta,
548                                                 net_weights,
549                                                 **weights)
550                 with tf.name_scope('grads'):
551                     grads = tf.gradients(loss,
552                                           self.dynamics.trainable_variables)
553                     if self.clip_grads:
554                         grads, _ = tf.clip_by_global_norm(grads,
555                                         self.clip_value)
556
557             return loss, grads, x_dq, dynamics_output
558
559     def create_sampler(self):
560         """Create operation for generating new samples using dynamics engine.
561
562         NOTE: This method is to be used when running generic HMC to create
563         operations for generating new sampler.
564         """
565         with tf.name_scope('sampler'):
566             output = self.dynamics(self.x, self.beta,
567                                     save_lf=True, train=False)
568             self.x_out = output['x_out']
569             self.px = output['accept_prob']
570             if self.save_lf:
571                 self.lf_out_f = output['lf_out_f']
572                 self.pxs_out_f = output['accept_probs_f']
573                 self.lf_out_b = output['lf_out_b']
574                 self.pxs_out_b = output['accept_probs_b']
575                 self.masks_f = output['forward_mask']
576                 self.masks_b = output['backward_mask']
577                 self.logdets_f = output['logdets_f']
578                 self.logdets_b = output['logdets_b']
579                 self.sumlogdet_f = output['sumlogdet_f']
580                 self.sumlogdet_b = output['sumlogdet_b']
581
582     def build(self):
583         """Build Tensorflow graph."""
584         with tf.name_scope('output'):
585             # self.loss_op, self.grads, self.x_out, self.px, x_dq = output
586             loss, grads, x_dq, dynamics_output = self.calc_loss_and_grads(
587                 x=self.x, beta=self.beta,
588                 net_weights=self.net_weights,
589                 **self.loss_weights
590             )
591             # self.loss_op = outputs[0]
592             # self.grads = outputs[1]
593             self.loss_op = loss
594             self.grads = grads
595             self.x_out = dynamics_output['x_out']
596             self.px = dynamics_output['accept_prob']
597             self.charge_diffs_op = tf.reduce_sum(x_dq) / self.num_samples

```

```

598     if self.save_lf:
599         self.lf_out_f = dynamics_output['lf_out_f']
600         self.lf_out_b = dynamics_output['lf_out_b']
601         self.pxs_out_f = dynamics_output['accept_probs_f']
602         self.pxs_out_b = dynamics_output['accept_probs_b']
603         self.masks_f = dynamics_output['forward_mask']
604         self.masks_b = dynamics_output['backward_mask']
605         self.logdets_f = dynamics_output['logdets_f']
606         self.logdets_b = dynamics_output['logdets_b']
607         self.sumlogdet_f = dynamics_output['sumlogdet_f']
608         self.sumlogdet_b = dynamics_output['sumlogdet_b']
609
610     with tf.name_scope('train'):
611         # -----
612         # TODO:
613         #
614         # update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
615         # ! no update ops in the default graph
616         # io.log("update_ops: ", update_ops)
617         # Use the update ops of the model itself
618         # io.log("model.updates: ", self.dynamics.updates)
619         # -----
620         grads_and_vars = zip(self.grads, self.dynamics.trainable_variables)
621         with tf.control_dependencies(self.dynamics.updates):
622             self.train_op = self.optimizer.apply_gradients(
623                 grads_and_vars,
624                 global_step=self.global_step,
625                 name='train_op'
626             )

```

A.5 l2hmc-qcd/dynamics/gauge_dynamics.py

```
1 """
2 Dynamics engine for L2HMC sampler on Lattice Gauge Models.
3
4 Reference [Generalizing Hamiltonian Monte Carlo with Neural
5 Networks](https://arxiv.org/pdf/1711.09268.pdf)
6
7 Code adapted from the released TensorFlow graph implementation by original
8 authors https://github.com/brain-research/l2hmc.
9
10 Author: Sam Foreman (github: @saforem2)
11 Date: 1/14/2019
12 """
13 import numpy as np
14 import numpy.random as npr
15 import tensorflow as tf
16
17 from globals import GLOBAL_SEED, TF_FLOAT
18 from network.conv_net3d import ConvNet3D
19 from network.conv_net2d import ConvNet2D
20 from network.generic_net import GenericNet
21
22
23 def exp(x, name=None):
24     """Safe exponential using tf.check_numerics."""
25     return tf.check_numerics(tf.exp(x), f'{name} is NaN')
26
27
28 def flatten_tensor(tensor):
29     """Flattens tensor along axes 1:, since axis=0 indexes sample in batch.
30
31     Example: for a tensor of shape [b, x, y, t] -->
32         returns a tensor of shape [b, x * y * t]
33     """
34     batch_size = tensor.shape[0]
35     return tf.reshape(tensor, shape=(batch_size, -1))
36
37
38 class GaugeDynamics(tf.keras.Model):
39     """Dynamics engine of naive L2HMC sampler."""
40
41     def __init__(self, lattice, potential_fn, **kwargs):
42         """Initialization.
43
44         Args:
45             lattice: Lattice object containing multiple sample lattices.
46             potential_fn: Function specifying minus log-likelihood objective
47             to minimize.
48
49         NOTE: kwargs (expected)
50             num_steps: Number of leapfrog steps to use in integrator.
51             eps: Initial step size to use in leapfrog integrator.
52             network_arch: String specifying network architecture to use.
53                 Must be one of 'conv2D', 'conv3D', 'generic'. Networks
54                 are defined in `../network/
55             hmc: Flag indicating whether generic HMC (no augmented
56                 leapfrog) should be used instead of L2HMC. Defaults to
57                 False.
```

```

58     eps_trainable: Flag indicating whether the step size (eps)
59         should be trainable. Defaults to True.
60     np_seed: Seed to use for numpy.random.
61 """
62 super(GaugeDynamics, self).__init__(name='GaugeDynamics')
63 npr.seed(GLOBAL_SEED)
64
65     self.lattice = lattice
66     self.potential = potential_fn
67     self.batch_size = self.lattice.samples.shape[0]
68     self.x_dim = self.lattice.num_links
69
70     # create attributes from kwargs.items()
71     for key, val in kwargs.items():
72         if key != 'eps': # want to use self.eps as tf.Variable
73             setattr(self, key, val)
74
75     if self.num_hidden is None:
76         self.num_hidden = 2 * self.lattice.num_links
77
78     with tf.name_scope('eps'):
79         # self.eps = exp(self.alpha, name='eps')
80         self.eps = tf.Variable(
81             initial_value=kwargs.get('eps', 0.4),
82             name='eps',
83             dtype=TF_FLOAT,
84             trainable=self.eps_trainable
85         )
86
87     self._construct_masks()
88
89     if self.hmc:
90         self.x_fn = lambda inp: [
91             tf.zeros_like(inp[0]) for t in range(3)
92         ]
93         self.v_fn = lambda inp: [
94             tf.zeros_like(inp[0]) for t in range(3)
95         ]
96
97     else:
98         if self.network_arch == 'conv3D': # should be 'conv3D'
99             self._build_conv_nets_3D()
100        if self.network_arch == 'conv2d': # should be 'conv2D'
101            self._build_conv_nets_2D()
102        elif self.network_arch == 'generic':
103            self._build_generic_nets()
104        else:
105            raise AttributeError(`self._network_arch` must be one of "
106                                `conv3D`, `conv2D`, `generic`.`")
107
108    def _build_conv_nets_3D(self):
109        """Build ConvNet3D architecture for x and v functions."""
110        kwargs = {
111            '_input_shape': (self.batch_size, *self.lattice.links.shape),
112            'links_shape': self.lattice.links.shape,
113            'x_dim': self.lattice.num_links, # dimensionality of target space
114            'factor': 2., # scale factor used in original paper
115            'spatial_size': self.lattice.space_size, # spatial size of lattice
116            'num_hidden': self.num_hidden, # num hidden nodes
117            'num_filters': int(self.lattice.space_size), # num conv. filters

```

```

118     'filter_sizes': [(3, 3, 2), (2, 2, 2)], # size of conv. filters
119     'name_scope': 'position', # namespace in which to create network
120     'data_format': self.data_format, # channels_first if using GPU
121     'use_bn': self.use_bn, # whether or not to use batch normalization
122 }
123
124 with tf.name_scope("DynamicsNetwork"):
125     with tf.name_scope("XNet"):
126         self.x_fn = ConvNet3D(model_name='XNet', **kwargs)
127
128     kwargs['name_scope'] = 'momentum' # update name scope
129     kwargs['factor'] = 1. # factor used in orig. paper
130     with tf.name_scope("VNet"):
131         self.v_fn = ConvNet3D(model_name='VNet', **kwargs)
132
133 def _build_conv_nets_2D(self):
134     """Build ConvNet architecture for x and v functions."""
135     kwargs = {
136         '_input_shape': (self.batch_size, *self.lattice.links.shape),
137         'links_shape': self.lattice.links.shape,
138         'x_dim': self.lattice.num_links, # dimensionality of target space
139         'factor': 2., # scale factor used in original paper
140         'spatial_size': self.lattice.space_size, # spatial size of lattice
141         'num_hidden': self.num_hidden, # num hidden nodes
142         'num_filters': int(2 * self.lattice.space_size), # num filters
143         'filter_sizes': [(2, 2), (2, 2)], # for 1st and 2nd conv. layer
144         'name_scope': 'position', # namespace in which to create network
145         'data_format': self.data_format, # channels_first if using GPU
146         'use_bn': self.use_bn, # whether or not to use batch normalization
147     }
148
149     with tf.name_scope("DynamicsNetwork"):
150         with tf.name_scope("XNet"):
151             self.x_fn = ConvNet2D(model_name='XNet', **kwargs)
152
153         kwargs['name_scope'] = 'momentum'
154         kwargs['factor'] = 1.
155         with tf.name_scope("VNet"):
156             self.v_fn = ConvNet2D(model_name='VNet', **kwargs)
157
158 def _build_generic_nets(self):
159     """Build GenericNet FC-architectures for x and v fns. """
160
161     kwargs = {
162         '_input_shape': (self.batch_size, *self.lattice.links.shape),
163         'x_dim': self.lattice.num_links, # dimensionality of target space
164         'factor': 2., # scale factor used in original paper
165         'num_hidden': self.num_hidden, # num hidden nodes
166         'name_scope': 'position', # namespace in which to create network
167         'use_bn': self.use_bn
168     }
169
170     with tf.name_scope("DynamicsNetwork"):
171         with tf.name_scope("XNet"):
172             self.x_fn = GenericNet(model_name='XNet', **kwargs)
173
174         kwargs['factor'] = 1.
175         kwargs['name_scope'] = 'momentum'
176         with tf.name_scope("VNet"):
177             self.v_fn = GenericNet(model_name='VNet', **kwargs)

```

```

178
179 def call(self, x_in, beta, net_weights,
180         while_loop=True, v_in=None, save_lf=False):
181     """Call method."""
182     return self.apply_transition(x_in, beta, net_weights,
183                                 while_loop=while_loop,
184                                 v_in=v_in, save_lf=save_lf)
185
186 def apply_transition(self, x_in, beta, net_weights,
187                      while_loop=True, v_in=None, save_lf=False):
188     """Propose a new state and perform the accept/reject step.
189
190     Args:
191         x: Batch of (x) samples (batch of links).
192         beta (float): Inverse coupling constant.
193
194     Returns:
195         x_proposed: Proposed x before accept/reject step.
196         v_proposed: Proposed v before accept/reject step.
197         accept_prob: Probability of accepting the proposed states.
198         x_out: Samples after accept/reject step.
199     """
200     # Simulate dynamics both forward and backward
201     # Use sampled masks to compute the actual solutions
202     with tf.name_scope('apply_transition'):
203         with tf.name_scope('transition_forward'):
204             outputs_f = self.transition_kernel(x_in, beta,
205                                              net_weights,
206                                              while_loop=while_loop,
207                                              forward=True,
208                                              v_in=v_in,
209                                              save_lf=save_lf)
210             xf = outputs_f['x_proposed']
211             vf = outputs_f['v_proposed']
212             accept_prob_f = outputs_f['accept_prob']
213
214             if save_lf:
215                 lf_out_f = outputs_f['lf_out']
216                 logdets_f = outputs_f['logdets']
217                 sumlogdet_f = outputs_f['sumlogdet']
218
219             with tf.name_scope('transition_backward'):
220                 outputs_b = self.transition_kernel(x_in, beta,
221                                                 net_weights,
222                                                 while_loop=while_loop,
223                                                 forward=False,
224                                                 v_in=v_in,
225                                                 save_lf=save_lf)
226                 xb = outputs_b['x_proposed']
227                 vb = outputs_b['v_proposed']
228                 accept_prob_b = outputs_b['accept_prob']
229
230                 if save_lf:
231                     lf_out_b = outputs_b['lf_out']
232                     logdets_b = outputs_b['logdets']
233                     sumlogdet_b = outputs_b['sumlogdet']
234
235             # Decide direction uniformly
236             with tf.name_scope('transition_masks'):
237                 forward_mask = tf.cast(

```

```

238         tf.random_uniform((self.batch_size,),  
239                             seed=GLOBAL_SEED) > 0.5,  
240             TF_FLOAT,  
241             name='forward_mask'  
242         )  
243     backward_mask = 1. - forward_mask  
244  
245     # Obtain proposed states  
246     with tf.name_scope('x_proposed'):  
247         x_proposed = (forward_mask[:, None] * xf  
248                         + backward_mask[:, None] * xb)  
249  
250     with tf.name_scope('v_proposed'):  
251         v_proposed = (forward_mask[:, None] * vf  
252                         + backward_mask[:, None] * vb)  
253  
254     # Probability of accepting the proposed states  
255     with tf.name_scope('accept_prob'):  
256         accept_prob = (forward_mask * accept_prob_f  
257                         + backward_mask * accept_prob_b)  
258  
259     # Accept or reject step  
260     with tf.name_scope('accept_mask'):  
261         accept_mask = tf.cast(  
262             accept_prob > tf.random_uniform(tf.shape(accept_prob),  
263                                         seed=GLOBAL_SEED),  
264             TF_FLOAT,  
265             name='accept_mask'  
266         )  
267         reject_mask = 1. - accept_mask  
268  
269     # Samples after accept / reject step  
270     with tf.name_scope('x_out'):  
271         x_out = (accept_mask[:, None] * x_proposed  
272                         + reject_mask[:, None] * x_in)  
273  
274     outputs = {  
275         'x_proposed': x_proposed,  
276         'v_proposed': v_proposed,  
277         'accept_prob': accept_prob,  
278         'x_out': x_out  
279     }  
280  
281     if save_lf:  
282         outputs['lf_out_f'] = lf_out_f  
283         outputs['accept_probs_f'] = accept_prob_f  
284         outputs['lf_out_b'] = lf_out_b  
285         outputs['accept_probs_b'] = accept_prob_b  
286         outputs['forward_mask'] = forward_mask  
287         outputs['backward_mask'] = backward_mask  
288         outputs['logdets_f'] = logdets_f  
289         outputs['logdets_b'] = logdets_b  
290         outputs['sumlogdet_f'] = sumlogdet_f  
291         outputs['sumlogdet_b'] = sumlogdet_b  
292  
293     return outputs  
294  
295 def transition_kernel(self, x_in, beta, net_weights, while_loop=True,  
296                      forward=True, v_in=None, save_lf=False):  
297     """Transition kernel of augmented leapfrog integrator."""

```

```

298
299     if v_in is None:
300         with tf.name_scope('refresh_momentum'):
301             v_in = tf.random_normal(tf.shape(x_in), seed=GLOBAL_SEED)
302
303     if while_loop:
304         outputs = self._transition_while_loop(x_in, v_in,
305                                              beta, net_weights,
306                                              forward, save_lf)
307     else:
308         outputs = self._transition_for_loop(x_in, v_in,
309                                              beta, net_weights,
310                                              forward, save_lf)
311
312     x_proposed = outputs['x_proposed']
313     v_proposed = outputs['v_proposed']
314     sumlogdet = outputs['sumlogdet']
315
316     with tf.name_scope('accept_prob'):
317         accept_prob = self._compute_accept_prob(x_in, v_in,
318                                               x_proposed,
319                                               v_proposed,
320                                               sumlogdet,
321                                               beta)
322
323     outputs['accept_prob'] = accept_prob
324
325     return outputs
326
327 def _transition_for_loop(self, x_in, v_in, beta, net_weights,
328                         forward=True, save_lf=False):
329     """Implements the transition kernel using a basic `for` loop."""
330     lf_fn = self._forward_lf if forward else self._backward_lf
331
332     with tf.name_scope('for_loop_init'):
333         x_proposed, v_proposed = x_in, v_in
334         sumlogdet = 0.
335         lf_out = [x_proposed]
336         batch_size = tf.shape(x_in)[0]
337         # assert batch_size == self.batch_size
338         logdets_out = [tf.zeros((self.batch_size,))]
339
340         for i in range(self.num_steps):
341             step = tf.convert_to_tensor(i, dtype=TF_FLOAT)
342             x_proposed, v_proposed, logdet = lf_fn(x_proposed,
343                                                     v_proposed,
344                                                     beta, step,
345                                                     net_weights)
346             sumlogdet += logdet
347             lf_out.append(x_proposed)
348             logdets_out.append(logdet)
349
350         outputs = {
351             'x_proposed': x_proposed,
352             'v_proposed': v_proposed,
353             'sumlogdet': sumlogdet
354         }
355
356         if save_lf:
357             outputs['lf_out'] = lf_out

```

```

358     outputs['logdets'] = logdets_out
359
360     return outputs
361
362 def _transition_while_loop(self, x_in, v_in, beta, net_weights,
363                             forward=True, save_lf=False):
364     """Implements the transition kernel using a `tf.while_loop`."""
365     lf_fn = self._forward_lf if forward else self._backward_lf
366
367     with tf.name_scope('while_loop_init'):
368         x_proposed, v_proposed = x_in, v_in
369         # t = tf.constant(0., name='md_time', dtype=TF_FLOAT)
370         step = tf.constant(0., name='md_step', dtype=TF_FLOAT)
371         batch_size = tf.shape(x_in)[0]
372         # assert batch_size == self.batch_size
373         logdet = tf.zeros((batch_size,))
374         lf_out = tf.TensorArray(dtype=TF_FLOAT, size=self.num_steps+1,
375                                dynamic_size=True, name='lf_out',
376                                clear_after_read=False)
377         logdets_out = tf.TensorArray(dtype=TF_FLOAT, size=self.num_steps+1,
378                                    dynamic_size=True, name='logdets_out',
379                                    clear_after_read=False)
380         lf_out = lf_out.write(0, x_in)
381         logdets_out.write(0, logdet)
382
383     def body(x, v, beta, step, logdet, lf_samples, logdets):
384         i = tf.cast(step, dtype=tf.int32, name='lf_step') # cast as int
385         with tf.name_scope('apply_lf'):
386             new_x, new_v, j = lf_fn(x, v, beta, step, net_weights)
387             with tf.name_scope('concat_lf_outputs'):
388                 lf_samples = lf_samples.write(i+1, new_x)
389             with tf.name_scope('concat_logdets'):
390                 logdets = logdets.write(i+1, logdet+j)
391         return (new_x, new_v, beta, step + 1,
392                 logdet + j, lf_samples, logdets)
393
394     def cond(x, v, beta, step, logdet, lf_out, logdets):
395         with tf.name_scope('check_lf_step'):
396             return tf.less(step, self.num_steps)
397
398     with tf.name_scope('while_loop'):
399         outputs = tf.while_loop(
400             cond=cond,
401             body=body,
402             loop_vars=[x_proposed, v_proposed,
403                       beta, step, logdet,
404                       lf_out, logdets_out]
405         )
406
407         x_proposed = outputs[0]
408         v_proposed = outputs[1]
409         beta = outputs[2]
410         step = outputs[3]
411         sumlogdet = outputs[4]
412         lf_out = outputs[5].stack()
413         logdets_out = outputs[6].stack()
414
415         outputs = {
416             'x_proposed': x_proposed,
417             'v_proposed': v_proposed,

```

```

418         'sumlogdet': sumlogdet
419     }
420
421     if save_lf:
422         outputs['lf_out'] = lf_out
423         outputs['logdets'] = logdets_out
424
425     return outputs
426
427 def _forward_lf(self, x, v, beta, step, net_weights):
428     """One forward augmented leapfrog step."""
429     with tf.name_scope('forward_lf'):
430         with tf.name_scope('get_time'):
431             t = self._get_time(step, tile=tf.shape(x)[0])
432
433         if t.dtype != TF_FLOAT:
434             t = tf.cast(t, dtype=TF_FLOAT)
435
436         with tf.name_scope('get_mask'):
437             mask, mask_inv = self._get_mask(step)
438
439         with tf.name_scope('augmented_leapfrog'):
440             sumlogdet = 0.
441
442             v, logdet = self._update_v_forward(x, v, beta, t, net_weights)
443             sumlogdet += logdet
444
445             x, logdet = self._update_x_forward(x, v, t, net_weights,
446                                              mask, mask_inv)
447             sumlogdet += logdet
448
449             x, logdet = self._update_x_forward(x, v, t, net_weights,
450                                              mask_inv, mask)
451             sumlogdet += logdet
452
453             v, logdet = self._update_v_forward(x, v, beta, t, net_weights)
454             sumlogdet += logdet
455
456     return x, v, sumlogdet
457
458 def _backward_lf(self, x, v, beta, step, net_weights):
459     """One backward augmented leapfrog step."""
460     # Reversed index/sinusoidal time
461     with tf.name_scope('backward_lf'):
462         with tf.name_scope('get_time'):
463             t = self._get_time(self.num_steps - step - 1,
464                               tile=tf.shape(x)[0])
465
466         if t.dtype != TF_FLOAT:
467             t = tf.cast(t, dtype=TF_FLOAT)
468
469         with tf.name_scope('get_mask'):
470             mask, mask_inv = self._get_mask(self.num_steps - step - 1)
471
472         with tf.name_scope('augmented_leapfrog'):
473             sumlogdet = 0.
474
475             v, logdet = self._update_v_backward(x, v, beta, t, net_weights)
476             sumlogdet += logdet
477

```

```

478     x, logdet = self._update_x_backward(x, v, t, net_weights,
479                                         mask_inv, mask)
480     sumlogdet += logdet
481
482     x, logdet = self._update_x_backward(x, v, t, net_weights,
483                                         mask, mask_inv)
484     sumlogdet += logdet
485
486     v, logdet = self._update_v_backward(x, v, beta, t, net_weights)
487     sumlogdet += logdet
488
489     return x, v, sumlogdet
490
491 def _update_v_forward(self, x, v, beta, t, net_weights):
492     """Update v in the forward leapfrog step.
493
494     Args:
495         x: input position tensor
496         v: input momentum tensor
497         beta: inverse coupling constant
498         t: current leapfrog step
499         net_weights: Placeholders for the multiplicative weights by which
500             to multiply the S, Q, and T functions (scale, transformation,
501             translation resp.)
502
503     Returns:
504         v: Updated (output) momentum
505         logdet: Jacobian factor
506     """
507
508     with tf.name_scope('update_v_forward'):
509         with tf.name_scope('grad_potential'):
510             grad = self.grad_potential(x, beta)
511
512         with tf.name_scope('v_fn'):
513             # Sv: scale, Qv: transformation, Tv: translation
514             scale, translation, transformation = self.v_fn(
515                 [x, grad, t]
516             )
517
518             with tf.name_scope('scale'):
519                 scale *= 0.5 * self.eps * net_weights[0]
520                 scale_exp = exp(scale, 'vf_scale')
521
522             with tf.name_scope('transformation'):
523                 transformation *= self.eps * net_weights[1]
524                 transformation_exp = exp(transformation, 'vf_transformation')
525
526             with tf.name_scope('translation'):
527                 translation *= net_weights[2]
528
529             with tf.name_scope('v_update'):
530                 v = (v * scale_exp - 0.5 * self.eps
531                      * (grad * transformation_exp + translation))
532
533     return v, tf.reduce_sum(scale, axis=1, name='vf_logdet')
534
535 def _update_x_forward(self, x, v, t, net_weights, mask, mask_inv):
536     """Update x in the forward leapfrog step."""
537     with tf.name_scope('update_x_forward'):
538         with tf.name_scope('x_fn'):
539             scale, translation, transformation = self.x_fn(

```

```

538             [v, mask * x, t]
539         )
540
541     with tf.name_scope('scale'):
542         scale *= self.eps * net_weights[0]
543         scale_exp = exp(scale, 'xf_scale')
544
545     with tf.name_scope('transformation'):
546         transformation *= self.eps * net_weights[1]
547         transformation_exp = exp(transformation, 'xf_transformation')
548
549     with tf.name_scope('translation'):
550         translation *= net_weights[2]
551
552     with tf.name_scope('x_update'):
553         x = (mask * x
554             + mask_inv * (x * scale_exp + self.eps
555                             * (v * transformation_exp + translation)))
556
557     return x, tf.reduce_sum(mask_inv * scale, axis=1, name='xf_logdet')
558
559 def _update_v_backward(self, x, v, beta, t, net_weights):
560     """Update v in the backward leapfrog step. Invert the forward update"""
561     with tf.name_scope('update_v_backward'):
562         with tf.name_scope('grad_potential'):
563             grad = self.grad_potential(x, beta)
564
565         with tf.name_scope('v_fn'):
566             scale, translation, transformation = self.v_fn(
567                 [x, grad, t]
568             )
569
570         with tf.name_scope('scale'):
571             scale *= -0.5 * self.eps * net_weights[0]
572             scale_exp = exp(scale, 'vb_scale')
573
574         with tf.name_scope('transformation'):
575             transformation *= self.eps * net_weights[1]
576             transformation_exp = exp(transformation, 'vb_transformation')
577
578         with tf.name_scope('translation'):
579             translation *= net_weights[2]
580
581         with tf.name_scope('v_update'):
582             v = (scale_exp * (v + 0.5 * self.eps
583                               * (grad * transformation_exp + translation)))
584
585     return v, tf.reduce_sum(scale, axis=1, name='vb_logdet')
586
587 def _update_x_backward(self, x, v, t, net_weights, mask, mask_inv):
588     """Update x in the backward lf step. Inverting the forward update."""
589     with tf.name_scope('update_x_backward'):
590         with tf.name_scope('x_fn'):
591             scale, translation, transformation = self.x_fn(
592                 [v, mask * x, t]
593             )
594
595         with tf.name_scope('scale'):
596             scale *= -self.eps * net_weights[0]
597             scale_exp = exp(scale, 'xb_scale')

```

```

598
599     with tf.name_scope('transformation'):
600         transformation *= self.eps * net_weights[1]
601         transformation_exp = exp(transformation, 'xb_transformation')
602
603     with tf.name_scope('translation'):
604         translation *= net_weights[2]
605
606     with tf.name_scope('x_update'):
607         x = (mask * x + mask_inv * scale_exp
608              * (x - self.eps * (v * transformation_exp + translation)))
609
610     return x, tf.reduce_sum(mask_inv * scale, axis=1, name='xb_logdet')
611
612 def _compute_accept_prob(self, xi, vi, xf, vf, sumlogdet, beta):
613     """Compute the prob of accepting the proposed state given old state.
614     Args:
615         xi: Initial state.
616         vi: Initial v.
617         xf: Proposed state.
618         vf: Proposed v.
619         sumlogdet: Sum of the terms of the log of the determinant.
620             (Eq. 14 of original paper).
621         beta: Inverse coupling constant of gauge model.
622     """
623     with tf.name_scope('compute_accept_prob'):
624         with tf.name_scope('old_hamiltonian'):
625             old_hamil = self.hamiltonian(xi, vi, beta)
626         with tf.name_scope('new_hamiltonian'):
627             new_hamil = self.hamiltonian(xf, vf, beta)
628
629         with tf.name_scope('prob'):
630             prob = exp(tf.minimum(
631                 (old_hamil - new_hamil + sumlogdet), 0.
632             ), 'accept_prob')
633
634     # Ensure numerical stability as well as correct gradients
635     return tf.where(tf.is_finite(prob), prob, tf.zeros_like(prob))
636
637 def _get_time(self, step, tile=1):
638     """Format time as [cos(..), sin(...)]."""
639     with tf.name_scope('get_time'):
640         trig_t = tf.squeeze([
641             tf.cos(2 * np.pi * step / self.num_steps),
642             tf.sin(2 * np.pi * step / self.num_steps),
643         ], name='md_time', )
644
645     return tf.tile(tf.expand_dims(trig_t, 0), (tile, 1))
646
647 def _construct_masks(self):
648     """Construct different binary masks for different time steps."""
649     self.masks = []
650     for _ in range(self.num_steps):
651         # Need to use np.random here because tf would generate different random
652         # values across different `sess.run`
653         idx = np.random.permutation(np.arange(self.x_dim))[:self.x_dim // 2]
654         mask = np.zeros((self.x_dim,))
655         mask[idx] = 1.
656         mask = tf.constant(mask, dtype=TF_FLOAT)
657         self.masks.append(mask[None, :])

```

```

658
659     def _get_mask(self, step):
660         if step.dtype != tf.int32:
661             step = tf.cast(step, dtype=tf.int32)
662
663         with tf.name_scope('get_mask'):
664             if tf.executing_eagerly():
665                 m = self.masks[step]
666             else:
667                 m = tf.gather(self.masks, step, name='gather_mask')
668         return m, 1. - m
669
670     def potential_energy(self, x, beta):
671         """Compute potential energy using `self.potential` and beta."""
672         with tf.name_scope('potential_energy'):
673             potential_energy = tf.multiply(beta, self.potential(x),
674                                             name='potential_energy')
675
676         return potential_energy
677
678     def kinetic_energy(self, v):
679         """Compute the kinetic energy."""
680         with tf.name_scope('kinetic_energy'):
681             kinetic_energy = 0.5 * tf.reduce_sum(v**2, axis=1)
682
683         return kinetic_energy
684
685     def hamiltonian(self, x, v, beta):
686         """Compute the overall Hamiltonian."""
687         with tf.name_scope('hamiltonian'):
688             with tf.name_scope('potential'):
689                 potential = self.potential_energy(x, beta)
690             with tf.name_scope('kinetic'):
691                 kinetic = self.kinetic_energy(v)
692             with tf.name_scope('hamiltonian'):
693                 hamiltonian = potential + kinetic
694
695         return hamiltonian
696
697     def grad_potential(self, x, beta):
698         """Get gradient of potential function at current location."""
699         with tf.name_scope('grad_potential'):
700             if tf.executing_eagerly():
701                 x_tensor = tf.constant(x)
702                 beta_tensor = tf.constant(beta)
703                 with tf.GradientTape() as tape:
704                     tape.watch(x_tensor)
705                     potential_energy = self.potential_energy(x_tensor,
706                                                               beta_tensor)
707                     grad = tape.gradient(potential_energy, x_tensor)
708
709             else:
710                 grad = tf.gradients(self.potential_energy(x, beta), x)[0]
711
712         return grad

```

A.6 l2hmc-qcd/lattice/lattice.py

```
1 """
2 lattice.py
3
4 Contains implementation of GaugeLattice class.
5
6 Author: Sam Foreman (github: @saforem2)
7 Date: 01/15/2019
8 """
9 import os
10 import random
11 import pickle
12
13 import numpy as np
14 import tensorflow as tf
15 from functools import reduce
16 from scipy.linalg import expm
17 from scipy.special import i0, i1
18 from globals import TF_FLOAT, NP_FLOAT
19
20 def u1_plaq_exact(beta):
21     """Computes the expected value of the `average` plaquette for U(1)."""
22     return i1(beta) / i0(beta)
23
24
25 def pbc(tup, shape):
26     """Returns tup % shape for implementing periodic boundary conditions."""
27     return list(np.mod(tup, shape))
28
29
30 def pbc_tf(tup, shape):
31     """Tensorflow implementation of `pbc` defined above."""
32     return list(tf.mod(tup, shape))
33
34
35 def mat_adj(mat):
36     """Returns the adjoint (i.e. conjugate transpose) of a matrix `mat`."""
37     return tf.transpose(tf.conj(mat)) # conjugate transpose
38
39
40 def project_angle(x):
41     """Returns the projection of an angle `x` from [-4pi, 4pi] to [-pi, pi]."""
42     return x - 2 * np.pi * tf.math.floor((x + np.pi) / (2 * np.pi))
43
44
45 def project_angle_fft(x, N=10):
46     """Use the fourier series representation `x` to approx `project_angle`.
47     NOTE: Because `project_angle` suffers a discontinuity, we approximate `x`
48     with its Fourier series representation in order to have a differentiable
49     function when computing the loss.
50     Args:
51         x (array-like): Array to be projected.
52         N (int): Number of terms to keep in Fourier series.
53     """
54
55     y = np.zeros(x.shape, dtype=NP_FLOAT)
56     for n in range(1, N):
57         y += (-2 / n) * ((-1) ** n) * tf.sin(n * x)
```

```

58     return y
59
60
61 class GaugeLattice(object):
62     """Lattice with Gauge field existing on links."""
63
64     def __init__(self,
65                  time_size,
66                  space_size,
67                  dim=2,
68                  link_type='U1',
69                  num_samples=None,
70                  rand=False):
71         """Initialization for GaugeLattice object.
72
73         Args:
74             time_size (int): Temporal extent of lattice.
75             space_size (int): Spatial extent of lattice.
76             dim (int): Dimensionality
77             link_type (str):
78                 String representing the type of gauge group for the link
79                 variables. Must be either 'U1', 'SU2', or 'SU3'
80
81             assert link_type.upper() in ['U1', 'SU2', 'SU3'], (
82                 "Invalid link_type. Possible values: U1', 'SU2', 'SU3'"
83             )
84
85         self.time_size = time_size
86         self.space_size = space_size
87         self.dim = dim
88         self.link_type = link_type
89         self.link_shape = ()
90
91         self.num_links = self.time_size * self.space_size * self.dim
92         self.num_plaqs = self.time_size * self.space_size
93         self.bases = np.eye(self.dim, dtype=np.int)
94
95         self.samples = self._init_samples(num_samples, rand)
96         self.samples_tensor = tf.convert_to_tensor(self.samples,
97                                                 dtype=TF_FLOAT)
98         self.links = self.samples[0]
99         self.batch_size = self.samples.shape[0]
100        self.links_shape = self.samples.shape[1:]
101
102    def _init_samples(self, num_samples, rand):
103        """Initialize samples."""
104        links_shape = tuple(
105            [self.time_size]
106            + [self.space_size for _ in range(self.dim-1)]
107            + [self.dim]
108            + list(self.link_shape))
109
110        samples_shape = (num_samples, *links_shape)
111        if rand:
112            samples = np.array(
113                np.random.uniform(0, 2*np.pi, samples_shape),
114                dtype=NP_FLOAT)
115        else:
116            samples = np.zeros(samples_shape, dtype=NP_FLOAT)

```

```

118
119     return samples
120
121 def calc_plaq_sums(self, samples=None):
122     """Calculate plaquette sums.
123
124     Explicitly, calculate the sum of the link variables around each
125     plaquette in the lattice for each sample in samples.
126
127     Args:
128         samples (tf tensor): Tensor of shape (N, D) where N is the batch
129             size and D is the number of links on the lattice. If samples is
130             None, self.samples will be used.
131
132     Returns:
133         plaq_sums (tf operation): Tensorflow operation capable of
134             calculating the plaquette sums.
135     """
136
137     if samples is None:
138         samples = self.samples
139
140     if samples.shape != self.samples.shape:
141         samples = tf.reshape(samples, shape=(self.samples.shape))
142
143     with tf.name_scope('calc_plaq_sums'):
144         plaq_sums = (samples[:, :, :, 0]
145                     - samples[:, :, :, 1]
146                     - tf.roll(samples[:, :, :, 0], shift=-1, axis=2)
147                     + tf.roll(samples[:, :, :, 1], shift=-1, axis=1))
148
149     return plaq_sums
150
151 def calc_actions(self, samples=None):
152     """Calculate the total action for each sample in samples."""
153
154     if samples is None:
155         samples = self.samples
156
157     with tf.name_scope('calc_total_actions'):
158         total_actions = tf.reduce_sum(
159             1. - tf.cos(self.calc_plaq_sums(samples)), axis=(1, 2),
160             name='total_actions')
161
162     return total_actions
163
164 def calc_plaqs(self, samples=None):
165     """Calculate the average plaq. values for each sample in samples."""
166
167     if samples is None:
168         samples = self.samples
169
170     with tf.name_scope('calc_plaqs'):
171         plaqs = tf.reduce_sum(tf.cos(self.calc_plaq_sums(samples)),
172                             axis=(1, 2), name='plaqs') / self.num_plaqs
173
174     return plaqs
175
176 def calc_top_charges(self, samples=None, fft=False):
177     """Calculate topological charges for each sample in samples."""
178
179     if samples is None:
180         samples = self.samples

```

```

178     with tf.name_scope('calc_top_charges'):
179         if fft:
180             ps_proj = project_angle_fft(self.calc_plaq_sums(samples), N=1)
181         else:
182             ps_proj = project_angle(self.calc_plaq_sums(samples))
183
184         top_charges = (tf.reduce_sum(ps_proj, axis=(1, 2),
185                                     name='top_charges')) / (2 * np.pi)
186     return top_charges
187
188 # pylint: disable=invalid-name
189 def calc_top_charges_diff(self, x1, x2, fft=False):
190     """Calculate the difference in topological charge between x1 and x2."""
191     with tf.name_scope('calc_top_charges_diff'):
192         charge_diff = tf.abs(self.calc_top_charges(x1, fft)
193                               - self.calc_top_charges(x2, fft))
194
195     return charge_diff
196
197 def get_potential_fn(self, samples):
198     """Returns callable function used for calculating the energy."""
199     def fn(samples):
200         return self.calc_actions(samples)
201     return fn

```

A.7 l2hmc-qcd/network/conv_net3d.py

```
1 """
2 conv_net3d.py
3
4 Convolutional neural network architecture for running L2HMC on a gauge lattice
5 configuration of links.
6
7 Reference [Generalizing Hamiltonian Monte Carlo with Neural
8 Networks](https://arxiv.org/pdf/1711.09268.pdf)
9
10 Code adapted from the released TensorFlow graph implementation by original
11 authors https://github.com/brain-research/l2hmc.
12
13 Author: Sam Foreman (github: @saforem2)
14 Date: 01/16/2019
15 """
16 import numpy as np
17 import tensorflow as tf
18
19 from globals import GLOBAL_SEED, TF_FLOAT
20
21 from .network_utils import custom_dense
22
23
24 np.random.seed(GLOBAL_SEED)
25
26 if '2.' not in tf.__version__:
27     tf.set_random_seed(GLOBAL_SEED)
28
29
30 class ConvNet3D(tf.keras.Model):
31     """Conv. neural net with different initialization scale based on input."""
32
33     def __init__(self, model_name, **kwargs):
34         """Initialization method.
35
36             Attributes:
37                 coeff_scale: Multiplicative factor (lambda_s in original paper p.
38                             13) multiplying tanh(W_s h_2 + b_s).
39                 coeff_transformation: Multiplicative factor (lambda_q in original
40                             paper p. 13) multiplying tanh(W_q h_2 + b_q).
41                 data_format: String (either 'channels_first' or 'channels_last').
42                     'channels_first' ('channels_last') is default for GPU (CPU).
43                     This value is automatically determined and set to the
44                     appropriate value.
45
46             super(ConvNet3D, self).__init__(name=model_name)
47
48             for key, val in kwargs.items():
49                 setattr(self, key, val)
50
51             if self.use_bn:
52                 if self.data_format == 'channels_first':
53                     self.bn_axis = 1
54                 elif self.data_format == 'channels_last':
55                     self.bn_axis = -1
56                 else:
57                     raise AttributeError("Expected 'data_format' to be "
```

```

58                                     "'channels_first' or 'channels_last')")
59
60     with tf.name_scope(self.name_scope):
61         with tf.name_scope('coeff_scale'):
62             self.coeff_scale = tf.Variable(
63                 initial_value=tf.zeros([1, self.x_dim]),
64                 name='coeff_scale',
65                 trainable=True,
66                 dtype=TF_FLOAT
67             )
68
69         with tf.name_scope('coeff_transformation'):
70             self.coeff_transformation = tf.Variable(
71                 initial_value=tf.zeros([1, self.x_dim]),
72                 name='coeff_transformation',
73                 trainable=True,
74                 dtype=TF_FLOAT
75             )
76
77         with tf.name_scope('conv_layers'):
78             with tf.name_scope('conv_x1'):
79                 self.conv_x1 = tf.keras.layers.Conv3D(
80                     filters=self.num_filters,
81                     kernel_size=self.filter_sizes[0],
82                     activation=tf.nn.relu,
83                     input_shape=self._input_shape,
84                     padding='same',
85                     name='conv_x1',
86                     dtype=TF_FLOAT,
87                     data_format=self.data_format
88
89             )
90
91         with tf.name_scope('pool_x1'):
92             self.max_pool_x1 = tf.keras.layers.MaxPooling3D(
93                 pool_size=(2, 2, 2),
94                 strides=2,
95                 padding='same',
96                 name='pool_x1',
97             )
98
99         with tf.name_scope('conv_v1'):
100            self.conv_v1 = tf.keras.layers.Conv3D(
101                filters=self.num_filters,
102                kernel_size=self.filter_sizes[0],
103                activation=tf.nn.relu,
104                input_shape=self._input_shape,
105                padding='same',
106                name='conv_v1',
107                dtype=TF_FLOAT,
108                data_format=self.data_format
109
110
111         with tf.name_scope('pool_v1'):
112             self.max_pool_v1 = tf.keras.layers.MaxPooling3D(
113                 pool_size=(2, 2, 2),
114                 strides=2,
115                 padding='same',
116                 name='pool_v1'
117             )

```

```

118
119     with tf.name_scope('conv_x2'):
120         self.conv_x2 = tf.keras.layers.Conv3D(
121             filters=2 * self.num_filters,
122             kernel_size=self.filter_sizes[1],
123             activation=tf.nn.relu,
124             padding='same',
125             name='conv_x2',
126             dtype=TF_FLOAT,
127             data_format=self.data_format
128         )
129
130     with tf.name_scope('pool_x2'):
131         self.max_pool_x2 = tf.keras.layers.MaxPooling3D(
132             pool_size=(2, 2, 2),
133             strides=2,
134             padding='same',
135             name='pool_x2'
136         )
137
138     with tf.name_scope('conv_v2'):
139         self.conv_v2 = tf.keras.layers.Conv3D(
140             filters=2 * self.num_filters,
141             kernel_size=self.filter_sizes[1],
142             activation=tf.nn.relu,
143             padding='same',
144             name='conv_v2',
145             dtype=TF_FLOAT,
146             data_format=self.data_format
147         )
148
149     with tf.name_scope('pool_v2'):
150         self.max_pool_v2 = tf.keras.layers.MaxPooling3D(
151             pool_size=(2, 2, 2),
152             strides=2,
153             padding='same',
154             name='pool_v2'
155         )
156
157     with tf.name_scope('fc_layers'):
158         with tf.name_scope('flatten'):
159             self.flatten = tf.keras.layers.Flatten(name='flatten')
160
161         with tf.name_scope('x_layer'):
162             self.x_layer = custom_dense(self.num_hidden,
163                                         self.factor/3.,
164                                         name='x_layer')
165
166         with tf.name_scope('v_layer'):
167             self.v_layer = custom_dense(self.num_hidden,
168                                         1./3.,
169                                         name='v_layer')
170
171         with tf.name_scope('t_layer'):
172             self.t_layer = custom_dense(self.num_hidden,
173                                         1./3.,
174                                         name='t_layer')
175
176         with tf.name_scope('h_layer'):
177             self.h_layer = custom_dense(self.num_hidden,

```

```

178                                         name='h_layer')
179
180         with tf.name_scope('scale_layer'):
181             self.scale_layer = custom_dense(
182                 self.x_dim, 0.001, name='scale_layer'
183             )
184
185         with tf.name_scope('translation_layer'):
186             self.translation_layer = custom_dense(
187                 self.x_dim, 0.001, 'translation_layer'
188             )
189
190         with tf.name_scope('transformation_layer'):
191             self.transformation_layer = custom_dense(
192                 self.x_dim, 0.001, 'transformation_layer'
193             )
194
195     def reshape_5D(self, tensor):
196         """
197             Reshape tensor to be compatible with tf.keras.layers.Conv3D.
198
199             If self.data_format is 'channels_first', and input `tensor` has shape
200             (N, 2, L, L), the output tensor has shape (N, 1, 2, L, L).
201
202             If self.data_format is 'channels_last' and input `tensor` has shape
203             (N, L, L, 2), the output tensor has shape (N, 2, L, L, 1).
204         """
205
206         if self.data_format == 'channels_first':
207             N, D, H, W = self._input_shape
208             # N, D, H, W = tensor.shape
209             if isinstance(tensor, np.ndarray):
210                 return np.reshape(tensor, (N, 1, D, H, W))
211
212             return tf.reshape(tensor, (N, 1, D, H, W))
213
214         if self.data_format == 'channels_last':
215             # N, H, W, D = tensor.shape
216             N, H, W, D = self._input_shape
217             if isinstance(tensor, np.ndarray):
218                 return np.reshape(tensor, (N, H, W, D, 1))
219
220             return tf.reshape(tensor, (N, H, W, D, 1))
221
222         raise AttributeError(`self.data_format` should be one of "
223                               "'channels_first' or 'channels_last'`")
224
225     # pylint: disable=invalid-name, arguments-differ
226     def call(self, inputs):
227         """Forward pass through network.
228
229             NOTE: Data flow of forward pass is outlined below.
230             =====
231             * inputs: x, v, t
232             -----
233             x -->
234                 (conv_x1, max_pool_x1) --> (conv_x1, max_pool_x2) -->
235                 batch_norm --> flatten_x --> x_layer --> x_out
236
237             v -->
238                 (conv_v1, max_pool_v1), --> (conv_v1, max_pool_v2) -->

```

```

238         batch_norm --> flatten_v --> v_layer --> v_out
239
240         t --> t_layer --> t_out
241
242         x_out + v_out + t_out --> h_layer --> h_out
243 =====
244 * h_out is then fed to three separate layers:
245 -----
246     (1.) h_out --> (scale_layer, tanh) * exp(coeff_scale)
247         output: scale (S function in orig. paper)
248
249     (2.) h_out --> translation_layer --> translation_out
250         output: translation (T function in orig. paper)
251
252     (3.) h_out -->
253         (transformation_layer, tanh) * exp(coeff_transformation)
254         output: transformation (Q function in orig. paper)
255 =====
256
257 Returns:
258     scale, translation, transformation (S, T, Q functions from paper)
259 """
260
261 v, x, t = inputs
262 # scale_weight = net_weights[0]
263 # transformation_weight = net_weights[1]
264 # translation_weight = net_weights[2]
265
266 with tf.name_scope('reshape'):
267     v = self.reshape_5D(v)
268     x = self.reshape_5D(x)
269
270 with tf.name_scope('x'):
271     x = self.max_pool_x1(self.conv_x1(x))
272     x = self.max_pool_x2(self.conv_x2(x))
273     if self.use_bn:
274         x = tf.keras.layers.BatchNormalization(axis=self.bn_axis)(x)
275     x = self.flatten(x)
276     x = tf.nn.relu(self.x_layer(x))
277
278 with tf.name_scope('v'):
279     v = self.max_pool_v1(self.conv_v1(v))
280     v = self.max_pool_v2(self.conv_v2(v))
281     if self.use_bn:
282         v = tf.keras.layers.BatchNormalization(axis=self.bn_axis)(v)
283     v = self.flatten(v)
284     v = tf.nn.relu(self.v_layer(v))
285
286 with tf.name_scope('t'):
287     t = tf.nn.relu(self.t_layer(t))
288
289 with tf.name_scope('h'):
290     h = tf.nn.relu(v + x + t)
291     h = tf.nn.relu(self.h_layer(h))
292
293 def reshape(t, name):
294     return tf.squeeze(
295         tf.reshape(t, shape=self._input_shape, name=name)
296     )
297
298 with tf.name_scope('translation'):

```

```

298     translation = self.translation_layer(h)
299
300     with tf.name_scope('scale'):
301         scale = (tf.nn.tanh(self.scale_layer(h))
302                   * tf.exp(self.coeff_scale))
303
304     with tf.name_scope('transformation'):
305         transformation = (self.transformation_layer(h)
306                             * tf.exp(self.coeff_transformation))
307
308     return scale, translation, transformation
309
310
311 # pylint:disable=too-many-arguments, too-many-instance-attributes
312 class ConvNet2D(tf.keras.Model):
313     """Conv. neural net with different initialization scale based on input."""
314
315     def __init__(self, model_name, **kwargs):
316         """Initialization method."""
317
318         super(ConvNet2D, self).__init__(name=model_name)
319
320         for key, val in kwargs.items():
321             setattr(self, key, val)
322
323         with tf.name_scope(self.name_scope):
324
325             self.coeff_scale = tf.Variable(
326                 initial_value=tf.zeros([1, self.x_dim]),
327                 name='coeff_scale',
328                 trainable=True,
329                 dtype=TF_FLOAT
330             )
331
332             self.coeff_transformation = tf.Variable(
333                 initial_value=tf.zeros([1, self.x_dim]),
334                 name='coeff_transformation',
335                 trainable=True,
336                 dtype=TF_FLOAT
337             )
338
339             self.conv_x1 = tf.keras.layers.Conv2D(
340                 filters=self.num_filters,
341                 kernel_size=self.filter_sizes[0],
342                 activation=tf.nn.relu,
343                 input_shape=self._input_shape,
344                 name='conv_x1',
345                 dtype=TF_FLOAT,
346                 data_format=self.data_format
347             )
348
349
350             self.max_pool_x1 = tf.keras.layers.MaxPooling2D(
351                 pool_size=(2, 2),
352                 strides=2,
353                 name='max_pool_x1'
354             )
355
356             self.conv_v1 = tf.keras.layers.Conv2D(
357                 filters=self.num_filters,

```

```

358     kernel_size=self.filter_sizes[0],
359     activation=tf.nn.relu,
360     input_shape=self._input_shape,
361     name='conv_v1',
362     dtype=TF_FLOAT,
363     data_format=self.data_format
364 )
365
366 self.max_pool_v1 = tf.keras.layers.MaxPooling2D(
367     pool_size=(2, 2),
368     strides=2,
369     name='max_pool_x1'
370 )
371
372 self.conv_x2 = tf.keras.layers.Conv2D(
373     filters=2*self.num_filters,
374     kernel_size=self.filter_sizes[1],
375     activation=tf.nn.relu,
376     name='conv_x2',
377     dtype=TF_FLOAT,
378     data_format=self.data_format
379 )
380
381 self.max_pool_x2 = tf.keras.layers.MaxPooling2D(
382     pool_size=(2, 2),
383     strides=2,
384     name='max_pool_x1'
385 )
386
387 self.conv_v2 = tf.keras.layers.Conv2D(
388     filters=2 * self.num_filters,
389     kernel_size=self.filter_sizes[1],
390     activation=tf.nn.relu,
391     name='conv_v2',
392     dtype=TF_FLOAT,
393     data_format=self.data_format
394 )
395
396 self.max_pool_v2 = tf.keras.layers.MaxPooling2D(
397     pool_size=(2, 2),
398     strides=2,
399     name='max_pool_x1'
400 )
401
402 self.flatten = tf.keras.layers.Flatten(name='flatten')
403
404 self.x_layer = custom_dense(self.num_hidden, self.factor/3.,
405                             name='x_layer')
406
407 self.v_layer = custom_dense(self.num_hidden, 1./3.,
408                             name='v_layer')
409
410 self.t_layer = custom_dense(self.num_hidden, 1./3.,
411                             name='t_layer')
412
413 self.h_layer = custom_dense(self.num_hidden, name='h_layer')
414
415 self.scale_layer = custom_dense(self.x_dim, 0.001,
416                                 name='scale_layer')
417

```

```

418     self.translation_layer = custom_dense(self.x_dim, 0.001,
419                                             name='translation_layer')
420
421     self.transformation_layer = custom_dense(
422         self.x_dim,
423         0.001,
424         name='transformation_layer'
425     )
426
427 # pylint: disable=invalid-name, arguments-differ
428 def call(self, inputs):
429     """call method.
430
431     Args:
432         inputs: Tuple consisting of (v, x, t) (momenta, x, time).
433
434     Returns:
435         scale, translation, transformation
436
437     NOTE: Architecture looks like
438         - inputs: x, v, t
439             x -->
440                 CONV_X1, MAX_POOL_X1, --> CONV_X1, MAX_POOL_X2 -->
441                 FLATTEN_X --> X_LAYER --> X_OUT
442
443             v -->
444                 CONV_V1, MAX_POOL_V1, --> CONV_V1, MAX_POOL_V2 -->
445                 FLATTEN_V --> V_LAYER --> V_OUT
446
447             t --> T_LAYER --> T_OUT
448
449             X_OUT + V_OUT + T_OUT --> H_LAYER --> H_OUT
450
451         - H_OUT is then fed to three separate layers:
452             (1.) H_OUT --> (SCALE_LAYER, TANH) * exp(COEFF_SCALE)
453                 output: scale
454             (2.) H_OUT --> TRANSLATION_LAYER --> TRANSLATION_OUT
455                 output: translation
456             (3.) H_OUT --> (TRANSFORMATION_LAYER, TANH)
457                 * exp(COEFF_TRANSFORMATION)
458
459                 output: transformation
460
461     """
462
463     v, x, t = inputs
464
465     x = self.max_pool_x1(self.conv_x1(x))
466     x = tf.nn.local_response_normalization(x)
467     x = self.max_pool_x2(self.conv_x2(x))
468     x = tf.nn.local_response_normalization(x)
469     x = self.flatten(x)
470
471     v = self.max_pool_v1(self.conv_v1(v))
472     v = tf.nn.local_response_normalization(v)
473     v = self.max_pool_v2(self.conv_v2(v))
474     v = tf.nn.local_response_normalization(v)
475     v = self.flatten(v)
476
477     h = tf.nn.relu(self.v_layer(v) + self.x_layer(x) + self.t_layer(t))
478     h = tf.nn.relu(self.h_layer(h))
479     # h = self.hidden_layer1(h)

```

```
478
479     def reshape(t, name):
480         return tf.reshape(t, shape=self._input_shape, name=name)
481
482     translation = reshape(self.translation_layer(h), name='translation')
483
484     scale = reshape(
485         tf.nn.tanh(self.scale_layer(h)) * tf.exp(self.coeff_scale),
486         name='scale'
487     )
488
489     transformation = reshape(
490         self.transformation_layer(h) * tf.exp(self.coeff_transformation),
491         name='transformation'
492     )
493
494     return scale, translation, transformation
```

A.8 l2hmc-qcd/network/conv_net2d.py

```
1 """
2 conv_net2d.py
3
4 Implements a convolutional neural network using 2D convolutions.
5
6 Author: Sam Foreman (github: @saforem2)
7 Date: 06/14/2019
8 """
9 import numpy as np
10 import tensorflow as tf
11
12 from globals import GLOBAL_SEED, TF_FLOAT
13 from .network_utils import custom_dense
14
15
16 np.random.seed(GLOBAL_SEED)
17
18 if '2.' not in tf.__version__:
19     tf.set_random_seed(GLOBAL_SEED)
20
21
22 class ConvNet2D(tf.keras.Model):
23     """Conv. neural net with different initialization scale based on input."""
24
25     def __init__(self, model_name, **kwargs):
26         super(ConvNet2D, self).__init__(name=model_name)
27
28         for key, val in kwargs.items():
29             setattr(self, key, val)
30
31         if self.use_bn:
32             if self.data_format == 'channels_first':
33                 self.bn_axis = 1
34             elif self.data_format == 'channels_last':
35                 self.bn_axis = -1
36             else:
37                 raise AttributeError("Expected 'data_format' to be "
38                                     "'channels_first' or 'channels_last'")
39
40         with tf.name_scope(self.name_scope):
41             with tf.name_scope('coeff_scale'):
42                 self.coeff_scale = tf.Variable(
43                     initial_value=tf.zeros([1, self.x_dim]),
44                     name='coeff_scale',
45                     trainable=True,
46                     dtype=TF_FLOAT
47             )
48
49             with tf.name_scope('coeff_transformation'):
50                 self.coeff_transformation = tf.Variable(
51                     initial_value=tf.zeros([1, self.x_dim]),
52                     name='coeff_transformation',
53                     trainable=True,
54                     dtype=TF_FLOAT
55             )
56
57         with tf.name_scope('conv_layers'):
```

```

58     with tf.name_scope('conv_x1'):
59         self.conv_x1 = tf.keras.layers.Conv2D(
60             filters=self.num_filters,
61             kernel_size=self.filter_sizes[0],
62             activation=tf.nn.relu,
63             input_shape=self._input_shape[1:],
64             # padding='same',
65             name='conv_x1',
66             dtype=TF_FLOAT,
67             data_format=self.data_format
68
69     )
70
71     with tf.name_scope('pool_x1'):
72         self.max_pool_x1 = tf.keras.layers.MaxPooling2D(
73             pool_size=(2, 2),
74             strides=2,
75             # padding='same',
76             name='pool_x1',
77     )
78
79     with tf.name_scope('conv_v1'):
80         self.conv_v1 = tf.keras.layers.Conv2D(
81             filters=self.num_filters,
82             kernel_size=self.filter_sizes[0],
83             activation=tf.nn.relu,
84             input_shape=self._input_shape[1:],
85             # padding='same',
86             name='conv_v1',
87             dtype=TF_FLOAT,
88             data_format=self.data_format
89     )
90
91     with tf.name_scope('pool_v1'):
92         self.max_pool_v1 = tf.keras.layers.MaxPooling2D(
93             pool_size=(2, 2),
94             strides=2,
95             # padding='same',
96             name='pool_v1'
97     )
98
99     with tf.name_scope('conv_x2'):
100        self.conv_x2 = tf.keras.layers.Conv2D(
101            filters=2*self.num_filters,
102            kernel_size=self.filter_sizes[1],
103            activation=tf.nn.relu,
104            # padding='same',
105            name='conv_x2',
106            dtype=TF_FLOAT,
107            data_format=self.data_format
108    )
109
110    with tf.name_scope('pool_x2'):
111        self.max_pool_x2 = tf.keras.layers.MaxPooling2D(
112            pool_size=(2, 2),
113            strides=2,
114            # padding='same',
115            name='pool_x2'
116    )
117

```

```

118     with tf.name_scope('conv_v2'):
119         self.conv_v2 = tf.keras.layers.Conv2D(
120             filters=2*self.num_filters,
121             kernel_size=self.filter_sizes[1],
122             activation=tf.nn.relu,
123             # padding='same',
124             name='conv_v2',
125             dtype=TF_FLOAT,
126             data_format=self.data_format
127         )
128
129     with tf.name_scope('pool_v2'):
130         self.max_pool_v2 = tf.keras.layers.MaxPooling2D(
131             pool_size=(2, 2),
132             strides=2,
133             # padding='same',
134             name='pool_v2'
135         )
136
137     with tf.name_scope('fc_layers'):
138         with tf.name_scope('flatten'):
139             self.flatten = tf.keras.layers.Flatten(name='flatten')
140
141         with tf.name_scope('x_layer'):
142             self.x_layer = custom_dense(self.num_hidden,
143                                         self.factor/3.,
144                                         name='x_layer')
145
146         with tf.name_scope('v_layer'):
147             self.v_layer = custom_dense(self.num_hidden,
148                                         1./3.,
149                                         name='v_layer')
150
151         with tf.name_scope('t_layer'):
152             self.t_layer = custom_dense(self.num_hidden,
153                                         1./3.,
154                                         name='t_layer')
155
156         with tf.name_scope('h_layer'):
157             self.h_layer = custom_dense(self.num_hidden,
158                                         name='h_layer')
159
160         with tf.name_scope('scale_layer'):
161             self.scale_layer = custom_dense(
162                 self.x_dim, 0.001, name='scale_layer')
163
164         with tf.name_scope('translation_layer'):
165             self.translation_layer = custom_dense(
166                 self.x_dim, 0.001, 'translation_layer')
167
168         with tf.name_scope('transformation_layer'):
169             self.transformation_layer = custom_dense(
170                 self.x_dim, 0.001, 'transformation_layer')
171
172     def _reshape(self, tensor):
173         """Reshape tensor to be compatible with tf.keras.layers.Conv2D."""
174         if self.data_format == 'channels_first':

```

```

178     # batch_size, num_channels, height, width
179     N, D, H, W = self._input_shape
180     # N, D, H, W = tensor.shape
181     if isinstance(tensor, np.ndarray):
182         return np.reshape(tensor, (N, D, H, W))
183
184     return tf.reshape(tensor, (N, D, H, W))
185
186     if self.data_format == 'channels_last':
187         N, H, W, D = self._input_shape
188         if isinstance(tensor, np.ndarray):
189             return np.reshape(tensor, (N, H, W, D))
190
191         return tf.reshape(tensor, (N, H, W, D))
192
193     raise AttributeError(`self.data_format` should be one of "
194                           "'channels_first' or 'channels_last'")
195
196 def call(self, inputs):
197     """Forward pass through the network."""
198     v, x, t = inputs
199
200     with tf.name_scope('reshape'):
201         v = self._reshape(v)
202         x = self._reshape(x)
203
204     with tf.name_scope('x'):
205         x = self.max_pool_x1(self.conv_x1(x))
206         x = self.max_pool_x2(self.conv_x2(x))
207         if self.use_bn:
208             x = tf.keras.layers.BatchNormalization(axis=self.bn_axis)(x)
209         x = tf.nn.relu(self.x_layer(self.flatten(x)))
210
211     with tf.name_scope('v'):
212         v = self.max_pool_v1(self.conv_v1(v))
213         v = self.max_pool_v2(self.conv_v2(v))
214         if self.use_bn:
215             v = tf.keras.layers.BatchNormalization(axis=self.bn_axis)(v)
216         v = tf.nn.relu(self.v_layer(self.flatten(v)))
217         # v = self.flatten(v)
218         # v = tf.nn.relu(self.v_layer(v))
219
220     with tf.name_scope('t'):
221         t = tf.nn.relu(self.t_layer(t))
222
223     with tf.name_scope('h'):
224         h = tf.nn.relu(v + x + t)
225         h = tf.nn.relu(self.h_layer(h))
226
227     with tf.name_scope('translation'):
228         translation = self.translation_layer(h)
229
230     with tf.name_scope('scale'):
231         scale = (tf.nn.tanh(self.scale_layer(h))
232                  * tf.exp(self.coeff_scale))
233
234     with tf.name_scope('transformation'):
235         transformation = (tf.nn.tanh(self.transformation_layer(h))
236                            * tf.exp(self.coeff_transformation))
237

```


A.9 l2hmc-qcd/network/generic_net.py

```
1 """
2 Generic, fully-connected neural network architecture for running L2HMC on a
3 gauge lattice configuration of links.
4
5 NOTE: Lattices are flattened before being passed as input to the network.
6
7 Reference [Generalizing Hamiltonian Monte Carlo with Neural
8 Networks](https://arxiv.org/pdf/1711.09268.pdf)
9
10 Code adapted from the released TensorFlow graph implementation by original
11 authors https://github.com/brain-research/l2hmc.
12
13 Author: Sam Foreman (github: @saforem2)
14 Date: 01/16/2019
15 """
16 import tensorflow as tf
17 import numpy as np
18
19 from globals import TF_FLOAT
20
21 from .network_utils import custom_dense
22
23
24 class GenericNet(tf.keras.Model):
25     """Conv. neural net with different initialization scale based on input."""
26     def __init__(self, model_name='GenericNet', **kwargs):
27         """Initialization method."""
28
29         super(GenericNet, self).__init__(name=model_name)
30
31         for key, val in kwargs.items():
32             setattr(self, key, val)
33
34         if self.name_scope is None:
35             self.name_scope = model_name
36
37         if self.use_bn:
38             self.bn_axis = -1
39
40         # with tf.variable_scope(variable_scope):
41         with tf.name_scope(self.name_scope):
42             # self.flatten = tf.keras.layers.Flatten(name='flatten')
43
44             with tf.name_scope('x_layer'):
45                 self.x_layer = custom_dense(self.num_hidden,
46                                             self.factor/3.,
47                                             name='x_layer')
48
49             with tf.name_scope('v_layer'):
50                 self.v_layer = custom_dense(self.num_hidden,
51                                             1./3.,
52                                             name='v_layer')
53
54             with tf.name_scope('t_layer'):
55                 self.t_layer = custom_dense(self.num_hidden,
56                                             1./3., name='t_layer')
```

```

58     with tf.name_scope('h_layer'):
59         self.h_layer = custom_dense(self.num_hidden,
60                                     name='h_layer')
61
62     with tf.name_scope('scale_layer'):
63         self.scale_layer = custom_dense(self.x_dim, 0.001,
64                                         name='scale_layer')
65
66     with tf.name_scope('translation_layer'):
67         self.translation_layer = custom_dense(
68             self.x_dim,
69             0.001,
70             name='translation_layer'
71         )
72
73     with tf.name_scope('transformation_layer'):
74         self.transformation_layer = custom_dense(
75             self.x_dim,
76             0.001,
77             name='transformation_layer'
78         )
79
80     with tf.name_scope('coeff_scale'):
81         self.coeff_scale = tf.Variable(
82             initial_value=tf.zeros([1, self.x_dim]),
83             name='coeff_scale',
84             trainable=True,
85             dtype=TF_FLOAT,
86         )
87
88     with tf.name_scope('coeff_transformation'):
89         self.coeff_transformation = tf.Variable(
90             initial_value=tf.zeros([1, self.x_dim]),
91             name='coeff_transformation',
92             trainable=True,
93             dtype=TF_FLOAT
94         )
95
96     def _reshape(self, tensor):
97         N, D, H, W = self._input_shape
98         if isinstance(tensor, np.ndarray):
99             return np.reshape(tensor, (N, D * H * W))
100
101     return tf.reshape(tensor, (N, D * H * W))
102
103 # pylint: disable=invalid-name, arguments-differ
104 def call(self, inputs):
105     """call method.
106
107     NOTE Architecture looks like:
108
109     * inputs: x, v, t
110       x --> FLATTEN_X --> X_LAYER --> X_OUT
111       v --> FLATTEN_V --> V_LAYER --> V_OUT
112       t --> T_LAYER --> T_OUT
113
114       X_OUT + V_OUT + T_OUT --> H_LAYER --> H_OUT
115
116     * H_OUT is then fed to three separate layers:
117       (1.) H_OUT -->

```

```

118             TANH(SCALE_LAYER) * exp(COEFF_SCALE) --> SCALE_OUT
119
120             input: H_OUT
121             output: scale
122
123         (2.) H_OUT --> TRANSLATION_LAYER --> TRANSLATION_OUT
124
125             input: H_OUT
126             output: translation
127
128         (3.) H_OUT -->
129             TANH(SCALE_LAYER)*exp(COEFF_TRANSFORMATION) -->
130             TRANFORMATION_OUT
131
132             input: H_OUT
133             output: transformation
134
135     Returns:
136         scale, translation, transformation
137     """
138     v, x, t = inputs
139
140     x = self._reshape(x)
141     v = self._reshape(v)
142
143     h = self.v_layer(v) + self.x_layer(x) + self.t_layer(t)
144     if self.use_bn:
145         h = tf.keras.layers.BatchNormalization(axis=self.bn_axis)(h)
146     h = tf.nn.relu(h)
147     h = self.h_layer(h)
148     h = tf.nn.relu(h)
149
150     scale = tf.nn.tanh(self.scale_layer(h)) * tf.exp(self.coeff_scale)
151
152     translation = self.translation_layer(h)
153
154     transformation = (self.transformation_layer(h)
155                         * tf.exp(self.coeff_transformation))
156
156     return scale, translation, transformation

```

A.10 l2hmc-qcd/network/network_utils.py

```
1 import numpy as np
2 import tensorflow as tf
3
4 from globals import GLOBAL_SEED, TF_FLOAT, NP_FLOAT
5
6
7 np.random.seed(GLOBAL_SEED)
8
9 if '2.' not in tf.__version__:
10     tf.set_random_seed(GLOBAL_SEED)
11
12
13 def custom_dense(units, factor=1., name=None):
14     """Custom dense layer with specified weight initialization."""
15     if '2.' not in tf.__version__:
16         kernel_initializer = tf.keras.initializers.VarianceScaling(
17             scale=factor,
18             mode='fan_in',
19             distribution='uniform',
20             dtype=TF_FLOAT,
21             seed=GLOBAL_SEED,
22         )
23     else:
24         kernel_initializer = tf.contrib.layers.variance_scaling_initializer(
25             factor=factor,
26             mode='FAN_IN',
27             seed=GLOBAL_SEED,
28             uniform=True,
29         )
30
31     return tf.keras.layers.Dense(
32         units=units,
33         use_bias=True,
34         kernel_initializer=kernel_initializer,
35         bias_initializer=tf.constant_initializer(0., dtype=TF_FLOAT),
36         name=name
37     )
38
39
40 def variable_on_cpu(name, shape, initializer):
41     """Helper to create a Variable stored on CPU memory.
42
43     Args:
44         name: name of the variable
45         shape: list of ints
46         initializer: initializer for Variable
47
48     Returns:
49         Variable Tensor
50     """
51     with tf.device('/cpu:0'):
52         var = tf.get_variable(name, shape, initializer, TF_FLOAT)
53     return var
54
55
56 def variable_with_weight_decay(name, shape, stddev, wd, cpu=True):
57     """Helper to create an initialized Variable with weight decay.
```

```

58
59 Note that the Variable is initialized with a truncated normal distribution.
60 A weight decay is added only if one is specified.
61
62 Args:
63   name: Name of the variable
64   shape: list of ints
65   stddev: standard deviation of a truncated Gaussian
66   wd: Add L2Loss weight decay multiplied by this float. If None, weight
67     decay is not added for this variable.
68
69 Returns:
70   Variable Tensor
71 """
72
73 if cpu:
74     var = variable_on_cpu(
75         name, shape, tf.truncated_normal_initializer(stddev=stddev,
76                                         dtype=TF_FLOAT)
77     )
78 else:
79     var = tf.get_variable(
80         name, shape, tf.truncated_normal_initializer(stddev=stddev,
81                                         dtype=TF_FLOAT)
82     )
83 if wd is not None:
84     weight_decay = tf.multiply(tf.nn.l2_loss(var), wd, name='weight_loss')
85     tf.add_to_collection('losses', weight_decay)
86
87 return var
88
89 def create_periodic_padding(samples, filter_size):
90     """Create periodic padding for multiple samples, using filter_size."""
91     original_size = np.shape(samples)
92     N = original_size[1] # number of links in lattice
93     # N = np.shape(samples)[1] # number of links in lattice
94     padding = filter_size - 1
95
96     samples = tf.reshape(samples, shape=(samples.shape[0], -1))
97
98     x = []
99     for sample in samples:
100         padded = np.zeros((N + 2 * padding), N + 2 * padding, 2)
101         # lower left corner
102         padded[:padding, :padding, :] = sample[N-padding:, N-padding:, :]
103         # lower middle
104         padded[padding:N+padding, :padding, :] = sample[:, N-padding:, :]
105         # lower right corner
106         padded[N+padding:, :padding, :] = sample[:padding, N-padding:, :]
107         # left side
108         padded[:padding, padding: N+padding, :] = sample[N-padding:, :, :]
109         # center
110         padded[:padding:N+padding, padding:N+padding, :] = sample[:, :, :]
111         # right side
112         padded[N+padding:, padding:N+padding:, :] = sample[:padding, :, :]
113         # top middle
114         padded[:padding:N+padding, N+padding:, :] = sample[:, :padding, :]
115         # top right corner
116         padded[N+padding:, N+padding:, :] = sample[:padding, :padding, :]
117
118     x.append(padded)

```

```
118  
119     return np.array(x, dtype=NP_FLOAT).reshape(*original_size)
```

A.11 l2hmc-qcd/trainers/gauge_model_trainer.py

```
1 """
2 gauge_model_trainer.py
3
4 Implements GaugeModelTrainer class responsible for training GaugeModel.
5
6 Author: Sam Foreman (github: @saforem2)
7 Date: 04/09/2019
8 """
9 # import os
10 import time
11 import numpy as np
12 # import tensorflow as tf
13
14 try:
15     import horovod.tensorflow as hvd
16     HAS_HOROVOD = True
17 except ImportError:
18     HAS_HOROVOD = False
19
20 import utils.file_io as io
21 from lattice.lattice import ul_plaq_exact
22 from globals import NP_FLOAT
23
24 h_str = ("{:^12s}{:^10s}{:^10s}{:^10s}{:^10s}"
25          "{:^10s}{:^10s}{:^10s}{:^10s}{:^10s}")
26
27 h_strf = h_str.format("STEP", "LOSS", "t/STEP", "% ACC", "EPS",
28                      "BETA", "ACTION", "PLAQ", "(EXACT)", "dQ", "LR")
29
30 dash0 = (len(h_strf) + 1) * '-'
31 dash1 = (len(h_strf) + 1) * '-'
32 TRAIN_HEADER = dash0 + '\n' + h_strf + '\n' + dash1
33
34
35 class GaugeModelTrainer:
36     def __init__(self, sess, model, logger=None):
37         """Initialization method.
38
39         Args:
40             sess: tf.Session object.
41             model: GaugeModel object (defined in `models/gauge_model.py`)
42             logger: TrainLogger object (defined in `loggers/train_logger.py`)
43
44         """
45         self.sess = sess
46         self.model = model
47         self.logger = logger
48
49     def update_beta(self, step):
50         """Returns new beta to follow annealing schedule."""
51         temp = ((1. / self.model.beta_init - 1. / self.model.beta_final)
52                 * (1. - step / float(self.model.train_steps))
53                 + 1. / self.model.beta_final)
54         new_beta = 1. / temp
55
56         return new_beta
57
```

```

58     def train_step(self, step, samples_np, beta_np=None, **weights):
59         """Perform a single training step.
60
61         Args:
62             step (int): Current training step.
63             samples_np (np.ndarray): Array of input data.
64             beta_np (float, optional): Input value for inverse coupling
65                 constant.
66
67         Returns:
68             out_data (dict)
69         """
70         start_time = time.time()
71
72         if beta_np is None:
73             beta_np = self.update_beta(step)
74
75         charge_weight = weights.get('charge_weight', None)
76         net_weights = weights.get('net_weights', None)
77
78         if charge_weight is None:
79             charge_weight = 0.
80
81         if net_weights is None:
82             # scale_weight, transformation_weight, translation_weight
83             net_weights = [1., 1., 1.]
84
85         fd = {
86             self.model.x: samples_np,
87             self.model.beta: beta_np,
88             self.model.net_weights[0]: net_weights[0],
89             self.model.net_weights[1]: net_weights[1],
90             self.model.net_weights[2]: net_weights[2],
91             self.model.charge_weight: charge_weight
92         }
93
94         global_step = self.sess.run(self.model.global_step)
95
96         ops = [
97             self.model.train_op,          # apply gradients
98             self.model.loss_op,          # calculate loss
99             self.model.x_out,            # get new samples
100            self.model.px,              # calculate accept prob.
101            self.model.dynamics.eps,    # calculate current step size
102            self.model.actions_op,       # calculate avg. actions
103            self.model.plaqs_op,         # calculate avg. plaqs
104            self.model.charges_op,        # calculate top. charges
105            self.model.charge_diffs_op,   # change in top. charge / num_samples
106            self.model.lr,               # evaluate learning rate
107         ]
108
109         outputs = self.sess.run(ops, feed_dict=fd)
110
111         dt = time.time() - start_time
112         out_data = {
113             'step': global_step,
114             'loss': outputs[1],
115             'samples': np.mod(outputs[2], 2 * np.pi),
116             'px': outputs[3],
117             'eps': outputs[4],

```

```

118     'actions': outputs[5],
119     'plaqs': outputs[6],
120     'charges': outputs[7],
121     'charge_diffs': outputs[8],
122     'lr': outputs[9],
123     'beta': beta_np
124 }
125
126     data_str = (
127         f'{global_step:>5g}/{self.model.train_steps:<6g} '
128         f'{outputs[1]:^9.4g}' # loss value
129         f'{dt:^9.4g}' # time / step
130         f'{np.mean(outputs[3]):^9.4g}' # accept prob
131         f'{outputs[4]:^9.4g}' # step size
132         f'{beta_np:^9.4g}' # beta
133         f'{np.mean(outputs[5]):^9.4g}' # avg. actions
134         f'{np.mean(outputs[6]):^9.4g}' # avg. plaqs.
135         f'{u1_plaq_exact(beta_np):^9.4g}' # exact plaq.
136         f'{outputs[8]:^9.4g}' # charge diff
137         f'{outputs[9]:^9.4g}' # learning rate
138     )
139
140     return out_data, data_str
141
142 def train(self, train_steps, **kwargs):
143     """Train the L2HMC sampler for `train_steps`.  

144
145     Args:
146         train_steps: Integer number of training steps to perform.
147         **kwargs: Possible (key, value) pairs are
148             'samples_np': Array of initial samples used to start
149                 training.
150             'beta_np': Initial value of beta used in annealing
151                 schedule.
152             'trace': Flag specifying that the training loop should be
153                 ran through a profiler.
154
155     initial_step = kwargs.get('initial_step', 0)
156     samples_np = kwargs.get('samples_np', None)
157     beta_np = kwargs.get('beta_np', None)
158     charge_weight = kwargs.get('charge_weight', None)
159     net_weights = kwargs.get('net_weights', None)
160
161     if beta_np is None:
162         beta_np = self.model.beta_init
163
164     if samples_np is None:
165         samples_np = np.reshape(
166             np.array(self.model.lattice.samples, dtype=NP_FLOAT),
167             (self.model.num_samples, self.model.x_dim)
168         )
169
170     assert samples_np.shape == self.model.x.shape
171
172     weights = {
173         'charge_weight': charge_weight,
174         'net_weights': net_weights
175     }
176
177     try:

```

```
178     io.log(TRAIN_HEADER)
179     for step in range(initial_step, train_steps):
180         out_data, data_str = self.train_step(step, samples_np,
181                                              **weights)
182         samples_np = out_data['samples']
183
184         # NOTE: try/except faster than explicitly checking
185         # if self.logger is not None
186         # each training step
187         try:
188             self.logger.update_training(self.sess, out_data,
189                                         data_str, **weights)
190         except AttributeError:
191             continue
192         try:
193             self.logger.write_train_strings()
194         except AttributeError:
195             pass
196
197     except (KeyboardInterrupt, SystemExit):
198         io.log("\nKeyboardInterrupt detected!")
199         io.log("Saving current state and exiting.")
200         if self.logger is not None:
201             self.logger.update_training(self.sess, out_data,
202                                         data_str, **weights)
```

A.12 l2hmc-qcd/runners/gauge_model_runner.py

```
1 """
2 runner.py
3
4 Implements GaugeModelRunner class responsible for running the L2HMC algorithm
5 on a U(1) gauge model.
6
7 Author: Sam Foreman (github: @saforem2)
8 Date: 04/09/2019
9 """
10
11 import os
12 import time
13 import pickle
14 from scipy.stats import sem
15 from collections import Counter, OrderedDict
16 import numpy as np
17
18 import utils.file_io as io
19 from lattice.lattice import ul_plaq_exact
20 from globals import RUN_HEADER
21
22 # ops = [                      # list of tensorflow operations to run
23 #     self.model.x_out,          # new samples (MD + MH accept/reject)
24 #     self.model.px,             # prob. of accepting proposed samples
25 #     self.model.actions_op,     # tot. action of each sample
26 #     self.model.plaqs_op,       # avg. plaquette of each sample
27 #     self.model.charges_op,     # topological charge Q, of each sample
28 #     self.model.charge_diffs_op # Q(x_out) - Q(samples_in)
29 # ]
30
31
32 class GaugeModelRunner:
33
34     def __init__(self, sess, model, logger=None):
35         """
36             Args:
37                 sess: tf.Session() object.
38                 model: GaugeModel object (defined in `models/gauge_model.py`)
39                 logger: RunLogger object (defined in `loggers/run_logger.py`),
40                         defaults to None. This is to simplify communication when using
41                         Horovod since the RunLogger object exists only on
42                         hvd.rank() == 0, which is responsible for all file I/O.
43
44         self.sess = sess
45         self.model = model
46         self.logger = logger
47         self.eps = self.sess.run(self.model.dynamics.eps)
48
49     def calc_charge_autocorrelation(self, charges):
50         autocorr = np.correlate(charges, charges, mode='full')
51         return autocorr
52
53     def save_run_data(self, run_data, run_strings, samples, **kwargs):
54         """
55             Args:
56                 run_data: All run data generated from `run` method.
57                 run_strings: list of all strings generated from `run` method.
```

```

58     samples: Optional collection of all samples generated from `run`  

59     method. Only relevant if model.sve_samples is True  

60     """  

61     run_dir = kwargs['run_dir']  

62     observables_dir = os.path.join(run_dir, 'observables')  

63  

64     io.check_else_make_dir(run_dir)  

65     io.check_else_make_dir(observables_dir)  

66  

67     if self.model.save_samples:  

68         samples_file = os.path.join(run_dir, 'run_samples.pkl')  

69         io.log(f"Saving samples to: {samples_file}.")  

70         with open(samples_file, 'wb') as f:  

71             pickle.dump(samples, f)  

72  

73     run_stats = self.calc_observables_stats(run_data,  

74                                             kwargs['therm_frac'])  

75  

76     data_file = os.path.join(run_dir, 'run_data.pkl')  

77     io.log(f"Saving run_data to: {data_file}.")  

78     with open(data_file, 'wb') as f:  

79         pickle.dump(run_data, f)  

80  

81     stats_data_file = os.path.join(run_dir, 'run_stats.pkl')  

82     io.log(f"Saving run_stats to: {stats_data_file}.")  

83     with open(stats_data_file, 'wb') as f:  

84         pickle.dump(run_stats, f)  

85  

86     for key, val in run_data.items():  

87         out_file = key + '.pkl'  

88         out_file = os.path.join(observables_dir, out_file)  

89         io.save_data(val, out_file, name=key)  

90  

91     for key, val in run_stats.items():  

92         out_file = key + '_stats.pkl'  

93         out_file = os.path.join(observables_dir, out_file)  

94         io.save_data(val, out_file, name=key)  

95  

96     history_file = os.path.join(run_dir, 'run_history.txt')  

97     _ = [io.write(s, history_file, 'a') for s in run_strings]  

98  

99     self.write_run_stats(run_stats, **kwargs)  

100  

101 def run_step(self, step, run_steps, inputs, **weights):  

102     """Perform a single run step.  

103  

104     Args:  

105         step (int): Current step.  

106         run_steps (int): Total number of run_steps to perform.  

107         inputs (tuple): Tuple consisting of (samples_in, beta_np, eps,  

108                         plaq_exact) where samples_in (np.ndarray) is the input batch of  

109                         samples, beta (float) is the input value of beta, eps is the  

110                         step size, and plaq_exact (float) is the expected avg. value of  

111                         the plaquette at this value of beta.  

112     Returns:  

113         out_data: Dictionary containing the output of running all of the  

114             tensorflow operations in `ops` defined below.  

115     """  

116     samples_in, beta_np, eps, plaq_exact = inputs

```

```

118     ops = [
119         self.model.x_out,
120         self.model.px,
121         self.model.actions_op,
122         self.model.plaqs_op,
123         self.model.charges_op,
124         self.model.charge_diffs_op,
125     ]
126     if self.model.save_lf:
127         ops.extend([
128             self.model.lf_out_f,
129             self.model.pxs_out_f,
130             self.model.lf_out_b,
131             self.model.pxs_out_b,
132             self.model.masks_f,
133             self.model.masks_b,
134             self.model.logdets_f,
135             self.model.logdets_b,
136             self.model.sumlogdet_f,
137             self.model.sumlogdet_b
138         ])
139
140     charge_weight = weights['charge_weight']
141     net_weights = weights['net_weights']
142
143     feed_dict = {
144         self.model.x: samples_in,
145         self.model.beta: beta_np,
146         self.model.charge_weight: charge_weight,
147         self.model.net_weights[0]: net_weights[0],
148         self.model.net_weights[1]: net_weights[1],
149         self.model.net_weights[2]: net_weights[2],
150     }
151
152     start_time = time.time()
153     outputs = self.sess.run(ops, feed_dict=feed_dict)
154     dt = time.time() - start_time
155     out_data = {
156         'step': step,
157         'beta': beta_np,
158         'eps': self.eps,
159         'samples': np.mod(outputs[0], 2 * np.pi),
160         'px': outputs[1],
161         'actions': outputs[2],
162         'plaqs': outputs[3],
163         'charges': outputs[4],
164         'charge_diffs': outputs[5],
165     }
166     if self.model.save_lf:
167         lf_outputs = {
168             'lf_out_f': outputs[6],
169             'pxs_out_f': outputs[7],
170             'lf_out_b': outputs[8],
171             'pxs_out_b': outputs[9],
172             'masks_f': outputs[10],
173             'masks_b': outputs[11],
174             'logdets_f': outputs[12],
175             'logdets_b': outputs[13],
176             'sumlogdet_f': outputs[14],
177             'sumlogdet_b': outputs[15],

```

```

178     }
179     out_data.update(lf_outputs)
180
181     data_str = (f'{step:>5g}/{run_steps:<6g} '
182                 f'{dt:^9.4g} '                               # time / step
183                 f'{np.mean(outputs[1]):^9.4g} '            # accept. prob
184                 f'{self.eps:^9.4g} '                      # step size
185                 f'{beta_np:^9.4g} '                      # beta val
186                 f'{np.mean(outputs[2]):^9.4g} '            # avg. actions
187                 f'{np.mean(outputs[3]):^9.4g} '            # avg. plaquettes
188                 f'{plaq_exact:^9.4g} '                   # exact plaquette val
189                 f'{outputs[5]:^9.4g} '                    # top. charge diff
190
191     return out_data, data_str
192
193 def run(self, run_steps, beta=None, therm_frac=10, **weights):
194     """Run the simulation to generate samples and calculate observables.
195
196     Args:
197         run_steps: Number of steps to run the sampler for.
198         current_step: Integer passed when the sampler is ran intermittently
199             during training, as a way to monitor the models performance
200             during training. By passing the current training step as
201             current_step, this data is saved to a unique directory labeled
202             by the current_step.
203         beta: Float value indicating the inverse coupling constant that the
204             sampler is to be run at.
205
206     Returns:
207         observables: Tuple of observables dictionaries consisting of:
208             (actions_dict, plaqs_dict, charges_dict, charge_diffs_dict).
209     """
210     run_steps = int(run_steps)
211
212     if beta is None:
213         beta = self.model.beta_final
214
215     plaq_exact = ul_plaq_exact(beta)
216
217     # start with randomly generated samples
218     samples_np = np.random.randn(*self.model.batch_size,
219                                 self.model.x_dim))
220
221     try:
222         io.log(RUN_HEADER)
223         for step in range(run_steps):
224             inputs = (samples_np, beta, self.eps, plaq_exact)
225             out_data, data_str = self.run_step(step,
226                                               run_steps,
227                                               inputs,
228                                               **weights)
229             samples_np = out_data['samples']
230
231             if self.logger is not None:
232                 self.logger.update(out_data, data_str)
233
234             if self.logger is not None:
235                 self.logger.save_run_data(therm_frac=therm_frac)
236
237     except (KeyboardInterrupt, SystemExit):

```

```

238     io.log("\nKeyboardInterrupt detected!")
239     io.log("Saving current state and exiting.")
240     if self.logger is not None:
241         self.logger.save_run_data(therm_frac=therm_frac)
242
243 def calc_observables_stats(self, run_data, therm_frac=10):
244     """Calculate statistics for lattice observables.
245
246     Args:
247         run_data: Dictionary of observables data. Keys denote the
248             observables name.
249         therm_frac: Fraction of data to throw out for thermalization.
250
251     Returns:
252         stats: Dictionary containing statistics for each observable in
253             run_data. Additionally, contains `charge_probs` which is a
254             dictionary of the form {charge_val: charge_val_probability}.
255     """
256     def get_stats(data, t_frac=10):
257         if isinstance(data, dict):
258             arr = np.array(list(data.values()))
259         elif isinstance(data, (list, np.ndarray)):
260             arr = np.array(data)
261
262         num_steps = arr.shape[0]
263         therm_steps = num_steps // t_frac
264         arr = arr[therm_steps:, :]
265         avg = np.mean(arr, axis=0)
266         err = sem(arr, axis=0)
267         stats = np.array([avg, err]).T
268         return stats
269
270     actions_stats = get_stats(run_data['actions'], therm_frac)
271     plaqs_stats = get_stats(run_data['plaqs'], therm_frac)
272
273     charges_arr = np.array(list(run_data['charges'].values()), dtype=int)
274     charges_stats = get_stats(charges_arr, therm_frac)
275
276     suspect_arr = charges_arr ** 2
277     suspect_stats = get_stats(suspect_arr)
278
279     charge_probs = {}
280     counts = Counter(list(charges_arr.flatten()))
281     total_counts = np.sum(list(counts.values()))
282     for key, val in counts.items():
283         charge_probs[key] = val / total_counts
284
285     charge_probs = OrderedDict(sorted(charge_probs.items(),
286                                     key=lambda k: k[0]))
287
288     stats = {
289         'actions': actions_stats,
290         'plaqs': plaqs_stats,
291         'charges': charges_stats,
292         'suscept': suspect_stats,
293         'charge_probs': charge_probs
294     }
295
296     return stats
297

```

```

298     def write_run_stats(self, stats, **kwargs):
299         """Write statistics in human readable format to .txt file."""
300         run_steps = kwargs['run_steps']
301         beta = kwargs['beta']
302         current_step = kwargs['current_step']
303         therm_steps = kwargs['therm_steps']
304         training = kwargs['training']
305         run_dir = kwargs['run_dir']
306
307         out_file = os.path.join(run_dir, 'run_stats.txt')
308
309         actions_avg, actions_err = stats['actions'].mean(axis=0)
310         plaqs_avg, plaqs_err = stats['plaqs'].mean(axis=0)
311         charges_avg, charges_err = stats['charges'].mean(axis=0)
312         suspect_avg, suspect_err = stats['suscept'].mean(axis=0)
313
314         # actions_arr = np.array(
315         #     list(run_data['actions'].values()))
316         # )[therm_steps:, :]
317         #
318         # plaqs_arr = np.array(
319         #     list(run_data['plaqs'].values()))
320         # )[therm_steps:, :]
321         #
322         # charges_arr = np.array(
323         #     list(run_data['charges'].values()),
324         #     dtype=np.int32
325         # )[therm_steps:, :]
326         #
327         # charges_squared_arr = charges_arr ** 2
328         #
329         # actions_err = sem(actions_arr, axis=None)
330         #
331         # plaqs_avg = np.mean(plaqs_arr)
332         # plaqs_err = sem(plaqs_arr, axis=None)
333         #
334         # q_avg = np.mean(charges_arr)
335         # q_err = sem(charges_arr, axis=None)
336         #
337         # q2_avg = np.mean(charges_squared_arr)
338         # q2_err = sem(charges_squared_arr, axis=None)
339
340         ns = self.model.num_samples
341         suspect_k1 = f' \navg. over all {ns} samples < Q >'
342         suspect_k2 = f' \navg. over all {ns} samples < Q^2 >'
343         actions_k1 = f' \navg. over all {ns} samples < action >'
344         plaqs_k1 = f' \n avg. over all {ns} samples < plaq >'
345
346         _est_key = ' \nestimate +/- stderr'
347
348         suspect_ss = {
349             suspect_k1: f'{charges_avg:.4g} +/- {charges_err:.4g}',
350             suspect_k2: f'{suspect_avg:.4g} +/- {suspect_err:.4g}',
351             _est_key: {}
352         }
353
354         actions_ss = {
355             actions_k1: f'{actions_avg:.4g} +/- {actions_err:.4g}\n',
356             _est_key: {}
357         }

```

```

358
359     plaqs_ss = {
360         'exact_plaq': f'{ul_plaq_exact(beta):.4g}\n',
361         plaqs_k1: f'{plaqs_avg:.4g} +/- {plaqs_err:.4g}\n',
362         _est_key: {}
363     }
364
365     def format_stats(x, name=None):
366         return [f'{name}: {i[0]:.4g} +/- {i[1]:.4g}' for i in x]
367
368     def zip_keys_vals(stats_strings, keys, vals):
369         for k, v in zip(keys, vals):
370             stats_strings[_est_key][k] = v
371         return stats_strings
372
373     keys = [f"sample {idx}" for idx in range(ns)]
374
375     suspect_vals = format_stats(stats['suscept'], '< Q^2 >')
376     actions_vals = format_stats(stats['actions'], '< action >')
377     plaqs_vals = format_stats(stats['plaqs'], '< plaq >')
378
379     suspect_ss = zip_keys_vals(suspect_ss, keys, suspect_vals)
380     actions_ss = zip_keys_vals(actions_ss, keys, actions_vals)
381     plaqs_ss = zip_keys_vals(plaqs_ss, keys, plaqs_vals)
382
383     def accumulate_strings(d):
384         all_strings = []
385         for k1, v1 in d.items():
386             if isinstance(v1, dict):
387                 for k2, v2 in v1.items():
388                     all_strings.append(f'{k2} {v2}')
389             else:
390                 all_strings.append(f'{k1}: {v1}\n')
391
392         return all_strings
393
394     actions_strings = accumulate_strings(actions_ss)
395     plaqs_strings = accumulate_strings(plaqs_ss)
396     suspect_strings = accumulate_strings(suscept_ss)
397
398     charge_probs_strings = []
399     for k, v in stats['charge_probs'].items():
400         charge_probs_strings.append(f' probability[Q = {k}]: {v}\n')
401
402     train_str = (f" stats after {current_step} training steps.\n"
403                  f"{ns} chains ran for {run_steps} steps at "
404                  f"beta = {beta}.")
405
406     run_str = (f" stats for {ns} chains ran for {run_steps} steps "
407                  f" at beta = {beta}.")
408
409     if training:
410         str0 = "Topological susceptibility" + train_str
411         str1 = "Total actions" + train_str
412         str2 = "Average plaquette" + train_str
413         str3 = "Topological charge probabilities" + train_str[6:]
414         therm_str = ''
415     else:
416         str0 = "Topological susceptibility" + run_str
417         str1 = "Total actions" + run_str

```

```

418     str2 = "Average plaquette" + run_str
419     str3 = "Topological charge probabilities" + run_str[6:]
420     therm_str = (
421         f'Ignoring first {therm_steps} steps for thermalization.'
422     )
423
424     ss0 = (1 + max(len(str0), len(therm_str))) * '-'
425     ss1 = (1 + max(len(str1), len(therm_str))) * '-'
426     ss2 = (1 + max(len(str2), len(therm_str))) * '-'
427     ss3 = (1 + max(len(str3), len(therm_str))) * '-'
428
429     io.log(f"Writing statistics to: {out_file}")
430
431     def log_and_write(sep_str, str0, therm_str, stats_strings, file):
432         io.log(sep_str)
433         io.log(str0)
434         io.log(therm_str)
435         io.log('')
436         _ = [io.log(s) for s in stats_strings]
437         io.log(sep_str)
438         io.log('')
439
440         io.write(sep_str, file, 'a')
441         io.write(str0, file, 'a')
442         io.write(therm_str, file, 'a')
443         _ = [io.write(s, file, 'a') for s in stats_strings]
444         io.write('\n', file, 'a')
445
446     log_and_write(ss0, str0, therm_str, suspect_strings, out_file)
447     log_and_write(ss1, str1, therm_str, actions_strings, out_file)
448     log_and_write(ss2, str2, therm_str, plaqs_strings, out_file)
449     log_and_write(ss3, str3, therm_str, charge_probs_strings, out_file)

```

A.13 l2hmc-qcd/loggers/run_logger.py

```
1 """
2 run_logger.py
3
4 Implements RunLogger class responsible for saving/logging data
5 from `run` phase of GaugeModel.
6
7 Author: Sam Foreman (github: @saforem2)
8 Date: 04/24/2019
9 """
10 import os
11 import pickle
12
13 import numpy as np
14
15 from collections import Counter, OrderedDict
16 from scipy.stats import sem
17 import utils.file_io as io
18
19 from globals import RUN_HEADER, NP_FLOAT
20
21 from lattice.lattice import ul_plaq_exact
22
23 from .train_logger import save_params
24
25
26 def arr_from_dict(d, key):
27     return np.array(list(d[key]))
28
29
30 def autocorr(x):
31     autocorr = np.correlate(x, x, mode='full')
32
33     return autocorr[autocorr.size // 2:]
34
35
36 class RunLogger:
37     def __init__(self, model, log_dir, save_lf_data=False):
38         """
39             Args:
40                 model: GaugeModel object.
41                 log_dir: Existing logdir from `TrainLogger`.
42         """
43
44         # self.sess = sess
45         self.model = model
46         assert os.path.isdir(log_dir)
47         self.log_dir = log_dir
48         self.runs_dir = os.path.join(self.log_dir, 'runs')
49         self.figs_dir = os.path.join(self.log_dir, 'figures')
50         self.save_lf_data = save_lf_data
51         io.check_else_make_dir(self.runs_dir)
52         io.check_else_make_dir(self.figs_dir)
53
54         self.run_steps = None
55         self.beta = None
56         self.run_data = {}
57         self.run_stats = {}
58         self.run_strings = [RUN_HEADER]
```

```

58     if self.model.save_lf:
59         self.samples_arr = []
60         self.lf_out = {
61             'forward': [],
62             'backward': [],
63         }
64         self.pxs_out = {
65             'forward': [],
66             'backward': [],
67         }
68         self.masks = {
69             'forward': [],
70             'backward': [],
71         }
72         self.logdets = {
73             'forward': [],
74             'backward': [],
75         }
76         self.sumlogdet = {
77             'forward': [],
78             'backward': [],
79         }
80
81     def reset(self, run_steps, beta, **weights):
82         """Reset run_data and run_strings to prep for new run."""
83         self.run_steps = int(run_steps)
84         self.beta = beta
85
86         self.run_data = {
87             'px': {},
88             'actions': {},
89             'plaqs': {},
90             'charges': {},
91             'charge_diffs': {},
92         }
93         self.run_stats = {}
94         self.run_strings = []
95         if self.model.save_lf:
96             self.samples_arr = []
97             self.lf_out = {
98                 'forward': [],
99                 'backward': [],
100            }
101            self.pxs_out = {
102                'forward': [],
103                'backward': [],
104            }
105            self.masks = {
106                'forward': [],
107                'backward': [],
108            }
109            self.logdets = {
110                'forward': [],
111                'backward': [],
112            }
113            self.sumlogdet = {
114                'forward': [],
115                'backward': [],
116            }
117

```

```

118     eps = self.model.eps
119     charge_weight = weights['charge_weight']
120     net_weights = weights['net_weights']
121
122     if charge_weight is None:
123         charge_weight = 0.
124     if net_weights is None:
125         net_weights = [1., 1., 1.]
126
127     nw_str = [str(i).replace('.', '') for i in net_weights]
128     w_str = nw_str[0] + nw_str[1] + nw_str[2]
129     run_str = (f'steps_{run_steps}_beta_{beta}_'
130                 f'eps_{eps:.3g}_weights_{w_str}')
131     self.run_dir = os.path.join(self.runs_dir, run_str)
132     io.check_else_make_dir(self.run_dir)
133     save_params(self.model.params, self.run_dir)
134
135     return self.run_dir, run_str
136
137 def update(self, data, data_str):
138     """Update run_data and append data_str to data_strings."""
139     # projection of samples onto [0, 2pi) done in run_step above
140     # if self.model.save_samples:
141     step = data['step']
142     beta = data['beta']
143     key = (step, beta)
144     self.run_data['px'][key] = data['px']
145     self.run_data['actions'][key] = data['actions']
146     self.run_data['plaqs'][key] = data['plaqs']
147     self.run_data['charges'][key] = data['charges']
148     self.run_data['charge_diffs'][key] = data['charge_diffs']
149     # self.run_data['charge_weight'][key]
150
151     if self.model.save_lf:
152         samples_np = data['samples']
153         self.samples_arr.append(samples_np)
154         self.lf_out['forward'].extend(np.array(data['lf_out_f']))
155         self.lf_out['backward'].extend(np.array(data['lf_out_b']))
156         self.logdets['forward'].extend(np.array(data['logdets_f']))
157         self.logdets['backward'].extend(np.array(data['logdets_b']))
158         self.sumlogdet['forward'].append(np.array(data['sumlogdet_f']))
159         self.sumlogdet['backward'].append(np.array(data['sumlogdet_b']))
160         # self.pxs_out['forward'].extend(np.array(data['pxs_out_f']))
161         # self.pxs_out['backward'].extend(np.array(data['pxs_out_b']))
162         # self.masks['forward'].extend(np.array(data['masks_f']))
163         # self.masks['backward'].extend(np.array(data['masks_b']))
164
165         self.run_strings.append(data_str)
166
167     if step % (10 * self.model.print_steps) == 0:
168         io.log(data_str)
169
170     if step % 100 == 0:
171         io.log(RUN_HEADER)
172
173 def calc_observables_stats(self, run_data, therm_frac=10):
174     """Calculate statistics for lattice observables.
175
176     Args:
177         run_data: Dictionary of observables data. Keys denote the

```

```

178         observables name.
179         therm_frac: Fraction of data to throw out for thermalization.
180
181     Returns:
182     stats: Dictionary containing statistics for each observable in
183     run_data. Additionally, contains `charge_probs` which is a
184     dictionary of the form {charge_val: charge_val_probability}.
185     """
186     def get_stats(data, t_frac=10):
187         if isinstance(data, dict):
188             arr = np.array(list(data.values()))
189         elif isinstance(data, (list, np.ndarray)):
190             arr = np.array(data)
191
192         num_steps = arr.shape[0]
193         therm_steps = num_steps // t_frac
194         arr = arr[therm_steps: , :]
195         avg = np.mean(arr, axis=0)
196         err = sem(arr, axis=0)
197         stats = np.array([avg, err]).T
198         return stats
199
200     actions_stats = get_stats(run_data['actions'], therm_frac)
201     plaqs_stats = get_stats(run_data['plaqs'], therm_frac)
202
203     charges_arr = np.array(list(run_data['charges'].values()), dtype=int)
204     charges_stats = get_stats(charges_arr, therm_frac)
205
206     suspect_arr = charges_arr ** 2
207     suspect_stats = get_stats(suspect_arr)
208
209     charge_probs = {}
210     counts = Counter(list(charges_arr.flatten()))
211     total_counts = np.sum(list(counts.values()))
212     for key, val in counts.items():
213         charge_probs[key] = val / total_counts
214
215     charge_probs = OrderedDict(sorted(charge_probs.items(),
216                                         key=lambda k: k[0]]))
217
218     stats = {
219         'actions': actions_stats,
220         'plaqs': plaqs_stats,
221         'charges': charges_stats,
222         'suscept': suspect_stats,
223         'charge_probs': charge_probs
224     }
225
226     return stats
227
228     def save_attr(self, name, attr, out_dir=None, dtype=NP_FLOAT):
229         if out_dir is None:
230             out_dir = self.run_dir
231
232         assert os.path.isdir(out_dir)
233         out_file = os.path.join(out_dir, name + '.npz')
234
235         if not isinstance(attr, np.ndarray) or attr.dtype != dtype:
236             try:
237                 attr = np.array(attr, dtype=dtype)

```

```

238     except ValueError:
239         import pdb
240         pdb.set_trace()
241
242     if os.path.isfile(out_file):
243         io.log(f'File {out_file} already exists. Skipping.')
244     else:
245         io.log(f'Saving {name} to: {out_file}')
246         np.savez_compressed(out_file, attr)
247
248     def save_run_data(self, therm_frac=10):
249         """Save run information."""
250         observables_dir = os.path.join(self.run_dir, 'observables')
251
252         io.check_else_make_dir(self.run_dir)
253         io.check_else_make_dir(observables_dir)
254
255         # if self.model.save_lf:
256         if self.save_lf_data:
257             self.save_attr('samples_out', self.samples_arr)
258             self.save_attr('lf_forward', self.lf_out['forward'])
259             self.save_attr('lf_backward', self.lf_out['backward'])
260             self.save_attr('masks_forward', self.masks['forward'])
261             self.save_attr('masks_backward', self.masks['backward'])
262             self.save_attr('logdets_forward', self.logdets['forward'])
263             self.save_attr('logdets_backward', self.logdets['backward'])
264             self.save_attr('sumlogdet_forward', self.sumlogdet['forward'])
265             self.save_attr('sumlogdet_backward', self.sumlogdet['backward'])
266             self.save_attr('accept_probs_forward', self.pxs_out['forward'])
267             self.save_attr('accept_probs_backward', self.pxs_out['backward'])
268
269         run_stats = self.calc_observables_stats(self.run_data, therm_frac)
270         charges = self.run_data['charges']
271         charges_arr = np.array(list(charges.values()))
272         charges_autocorrs = [autocorr(x) for x in charges_arr.T]
273         charges_autocorrs = [x / np.max(x) for x in charges_autocorrs]
274         self.run_data['charges_autocorrs'] = charges_autocorrs
275
276         data_file = os.path.join(self.run_dir, 'run_data.pkl')
277         io.log(f"Saving run_data to: {data_file}.")
278         with open(data_file, 'wb') as f:
279             pickle.dump(self.run_data, f)
280
281         stats_data_file = os.path.join(self.run_dir, 'run_stats.pkl')
282         io.log(f"Saving run_stats to: {stats_data_file}.")
283         with open(stats_data_file, 'wb') as f:
284             pickle.dump(run_stats, f)
285
286         for key, val in self.run_data.items():
287             out_file = key + '.pkl'
288             out_file = os.path.join(observables_dir, out_file)
289             io.save_data(val, out_file, name=key)
290
291         for key, val in run_stats.items():
292             out_file = key + '_stats.pkl'
293             out_file = os.path.join(observables_dir, out_file)
294             io.save_data(val, out_file, name=key)
295
296         history_file = os.path.join(self.run_dir, 'run_history.txt')
297         io.write(RUN_HEADER, history_file, 'w')

```

```

298     _ = [io.write(s, history_file, 'a') for s in self.run_strings]
299
300     self.write_run_stats(run_stats, therm_frac)
301
302     def write_run_stats(self, stats, therm_frac=10):
303         """Write statistics in human readable format to .txt file."""
304         # run_steps = kwargs['run_steps']
305         # beta = kwargs['beta']
306         # current_step = kwargs['current_step']
307         # therm_steps = kwargs['therm_steps']
308         # training = kwargs['training']
309         # run_dir = kwargs['run_dir']
310         therm_steps = self.run_steps // therm_frac
311
312         out_file = os.path.join(self.run_dir, 'run_stats.txt')
313
314         actions_avg, actions_err = stats['actions'].mean(axis=0)
315         plaqs_avg, plaqs_err = stats['plaqs'].mean(axis=0)
316         charges_avg, charges_err = stats['charges'].mean(axis=0)
317         suspect_avg, suspect_err = stats['suscept'].mean(axis=0)
318
319         ns = self.model.num_samples
320         suspect_k1 = f' \navg. over all {ns} samples < Q >'
321         suspect_k2 = f' \navg. over all {ns} samples < Q^2 >'
322         actions_k1 = f' \navg. over all {ns} samples < action >'
323         plaqs_k1 = f' \n avg. over all {ns} samples < plaq >'
324
325         _est_key = ' \nestimate +/- stderr'
326
327         suspect_ss = {
328             suspect_k1: f'{suspect_avg:.4g} +/- {suspect_err:.4g}',
329             suspect_k2: f'{suspect_avg:.4g} +/- {suspect_err:.4g}',
330             _est_key: {}
331         }
332
333         actions_ss = {
334             actions_k1: f'{actions_avg:.4g} +/- {actions_err:.4g}\n',
335             _est_key: {}
336         }
337
338         plaqs_ss = {
339             'exact_plaq': f'{u1_plaq_exact(self.beta):.4g}\n',
340             plaqs_k1: f'{plaqs_avg:.4g} +/- {plaqs_err:.4g}\n',
341             _est_key: {}
342         }
343
344         def format_stats(x, name=None):
345             return f'{name}: {i[0]:.4g} +/- {i[1]:.4g}' for i in x
346
347         def zip_keys_vals(stats_strings, keys, vals):
348             for k, v in zip(keys, vals):
349                 stats_strings[_est_key][k] = v
350
351             return stats_strings
352
353             keys = [f"sample {idx}" for idx in range(ns)]
354
355             suspect_vals = format_stats(stats['suscept'], '< Q^2 >')
356             actions_vals = format_stats(stats['actions'], '< action >')
357             plaqs_vals = format_stats(stats['plaqs'], '< plaq >')

```

```

358     suspect_ss = zip_keys_vals(suscept_ss, keys, suspect_vals)
359     actions_ss = zip_keys_vals(actions_ss, keys, actions_vals)
360     plaqs_ss = zip_keys_vals(plaqs_ss, keys, plaqs_vals)
361
362     def accumulate_strings(d):
363         all_strings = []
364         for k1, v1 in d.items():
365             if isinstance(v1, dict):
366                 for k2, v2 in v1.items():
367                     all_strings.append(f'{k2} {v2}')
368             else:
369                 all_strings.append(f'{k1}: {v1}\n')
370
371     return all_strings
372
373     actions_strings = accumulate_strings(actions_ss)
374     plaqs_strings = accumulate_strings(plaqs_ss)
375     suspect_strings = accumulate_strings(suscept_ss)
376
377     charge_probs_strings = []
378     for k, v in stats['charge_probs'].items():
379         charge_probs_strings.append(f' probability[Q = {k}]: {v}\n')
380
381     # train_str = (f" stats after {current_step} training steps.\n"
382     #               f"{ns} chains ran for {self.run_steps} steps at "
383     #               f"beta = {self.beta}.")
384
385     run_str = (f" stats for {ns} chains ran for {self.run_steps} steps "
386                f" at beta = {self.beta}.")
387
388     # if training:
389     #     str0 = "Topological susceptibility" + train_str
390     #     str1 = "Total actions" + train_str
391     #     str2 = "Average plaquette" + train_str
392     #     str3 = "Topological charge probabilities" + train_str[6:]
393     #     therm_str = ''
394
395     # else:
396     str0 = "Topological susceptibility" + run_str
397     str1 = "Total actions" + run_str
398     str2 = "Average plaquette" + run_str
399     str3 = "Topological charge probabilities" + run_str[6:]
400     therm_str = (
401         f'Ignoring first {therm_steps} steps for thermalization.'
402     )
403
404     ss0 = (' + max(len(str0), len(therm_str))) * '-'
405     ss1 = (' + max(len(str1), len(therm_str))) * '-'
406     ss2 = (' + max(len(str2), len(therm_str))) * '-'
407     ss3 = (' + max(len(str3), len(therm_str))) * '-'
408
409     io.log(f"Writing statistics to: {out_file}")
410
411     def log_and_write(sep_str, str0, therm_str, stats_strings, file):
412         io.log(sep_str)
413         io.log(str0)
414         io.log(therm_str)
415         io.log('')
416         _ = [io.log(s) for s in stats_strings]
417         io.log(sep_str)
418         io.log('')

```

```
418     io.write(sep_str, file, 'a')
419     io.write(str0, file, 'a')
420     io.write(therm_str, file, 'a')
421     _ = [io.write(s, file, 'a') for s in stats_strings]
422     io.write('\n', file, 'a')
423
424     log_and_write(ss0, str0, therm_str, suspect_strings, out_file)
425     log_and_write(ss1, str1, therm_str, actions_strings, out_file)
426     log_and_write(ss2, str2, therm_str, plaqs_strings, out_file)
427     log_and_write(ss3, str3, therm_str, charge_probs_strings, out_file)
428     log_and_write(ss3, str3, therm_str, charge_probs_strings, out_file)
429     log_and_write(ss3, str3, therm_str, charge_probs_strings, out_file)
```

A.14 l2hmc-qcd/loggers/train_logger.py

```
1 """
2 train_logger.py
3
4 Implements TrainLogger class responsible for saving/logging data from
5 GaugeModel.
6
7 Author: Sam Foreman (github: @saforem2)
8 Date: 04/09/2019
9 """
10 import os
11 import pickle
12
13 import utils.file_io as io
14
15 from globals import TRAIN_HEADER
16 from utils.tf_logging import variable_summaries # add_loss_summaries
17
18 import tensorflow as tf
19
20
21 def save_params(params, out_dir):
22     io.check_else_make_dir(out_dir)
23     params_txt_file = os.path.join(out_dir, 'parameters.txt')
24     params_pkl_file = os.path.join(out_dir, 'parameters.pkl')
25     with open(params_txt_file, 'w') as f:
26         for key, val in params.items():
27             f.write(f"{key}: {val}\n")
28     with open(params_pkl_file, 'wb') as f:
29         pickle.dump(params, f)
30
31
32 class TrainLogger:
33     def __init__(self, model, log_dir, summaries=False):
34         # self.sess = sess
35         self.model = model
36         self.summaries = summaries
37
38         self.charges_dict = {}
39         self.charge_diffs_dict = {}
40
41         self._current_state = {
42             'step': 0,
43             'beta': self.model.beta_init,
44             'eps': self.model.eps,
45             'lr': self.model.lr_init,
46             'samples': self.model.samples,
47         }
48
49         self.train_data_strings = [TRAIN_HEADER]
50         self.train_data = {
51             'loss': {},
52             'actions': {},
53             'plaqs': {},
54             'charges': {},
55             'charge_diffs': {},
56             'accept_probs': {}
57         }
```

```

58
59     # log_dir will be None if using_hvd and hvd.rank() != 0
60     # this prevents workers on different ranks from corrupting checkpoints
61     # if log_dir is not None and self.is_chief:
62     self._create_dir_structure(log_dir)
63     save_params(self.model.params, self.log_dir)
64
65     if self.summaries:
66         # if tf.executing_eagerly():
67         #     self.writer = tf.contrib.summary.create_file_writer(
68         #         self.train_summary_dir, flush_millis=10000
69         #     )
70         if not tf.executing_eagerly():
71             self.writer = tf.summary.FileWriter(self.train_summary_dir,
72                                              tf.get_default_graph())
73         self.create_summaries()
74
75     def _create_dir_structure(self, log_dir):
76         """Create relevant directories for storing data.
77
78         Args:
79             log_dir: Root directory in which all other directories are created.
80
81         Returns:
82             None
83         """
84         io.check_else_make_dir(log_dir)
85         self.log_dir = log_dir
86         self.train_dir = os.path.join(self.log_dir, 'training')
87         self.checkpoint_dir = os.path.join(self.log_dir, 'checkpoints')
88         self.train_summary_dir = os.path.join(
89             self.log_dir, 'summaries', 'train'
90         )
91
92         self.train_log_file = os.path.join(self.train_dir, 'training_log.txt')
93         self.current_state_file = os.path.join(self.train_dir,
94                                              'current_state.pkl')
95
96         io.make_dirs([self.train_dir, self.train_summary_dir,
97                      self.checkpoint_dir])
98
99     def create_summaries(self):
100         """Create summary objects for logging in TensorBoard."""
101         ld = self.log_dir
102         self.summary_writer = tf.contrib.summary.create_file_writer(ld)
103
104         grads_and_vars = zip(self.model.grads,
105                               self.model.dynamics.trainable_variables)
106
107         with tf.name_scope('loss'):
108             tf.summary.scalar('loss', self.model.loss_op)
109
110         # self.loss_averages_op = self._add_loss_summaries(self.model.loss_op)
111
112         with tf.name_scope('learning_rate'):
113             tf.summary.scalar('learning_rate', self.model.lr)
114
115         with tf.name_scope('step_size'):
116             tf.summary.scalar('step_size', self.model.dynamics.eps)
117

```

```

118     with tf.name_scope('tunneling_events'):
119         tf.summary.scalar('tunneling_events_per_sample',
120                           self.model.charge_diffs_op)
121
122     with tf.name_scope('avg_plaq'):
123         tf.summary.scalar('avg_plaq', self.model.avg_plaqs_op)
124
125     for var in tf.trainable_variables():
126         if 'batch_normalization' not in var.op.name:
127             tf.summary.histogram(var.op.name, var)
128
129     with tf.name_scope('summaries'):
130         for grad, var in grads_and_vars:
131             try:
132                 layer, _type = var.name.split('/')[-2:]
133                 name = layer + '_' + _type[:-2]
134             except (AttributeError, IndexError):
135                 name = var.name[:-2]
136
137             if 'batch_norm' not in name:
138                 variable_summaries(var, name)
139                 variable_summaries(grad, name + '/gradients')
140                 tf.summary.histogram(name + '/gradients', grad)
141
142             self.summary_op = tf.summary.merge_all(name='summary_op')
143
144     def save_current_state(self):
145         """Save current state to pickle file.
146
147         The current state contains the following, which makes life easier if
148         we're trying to restore training from a saved checkpoint:
149             * most recent samples
150             * learning_rate
151             * beta
152             * dynamics.eps
153             * training_step
154
155         with open(self.current_state_file, 'wb') as f:
156             pickle.dump(self._current_state, f)
157
158     def log_step(self, sess, step, samples_np, beta_np, **weights):
159         """Update self.logger.summaries."""
160         net_weights = weights['net_weights']
161         charge_weight = weights['charge_weight']
162
163         feed_dict = {
164             self.model.x: samples_np,
165             self.model.beta: beta_np,
166             self.model.charge_weight: charge_weight,
167             self.model.net_weights[0]: net_weights[0],
168             self.model.net_weights[1]: net_weights[1],
169             self.model.net_weights[2]: net_weights[2],
170         }
171         summary_str = sess.run(self.summary_op, feed_dict=feed_dict)
172
173         self.writer.add_summary(summary_str, global_step=step)
174         self.writer.flush()
175
176     def update_training(self, sess, data, data_str, **weights):
177         """Update _current state and train_data."""

```

```

178     step = data['step']
179     beta = data['beta']
180     self._current_state['step'] = step
181     self._current_state['beta'] = beta
182     self._current_state['lr'] = data['lr']
183     self._current_state['eps'] = data['eps']
184     self._current_state['samples'] = data['samples']
185     self._current_state['net_weights'] = weights['net_weights']
186     self._current_state['charge_weight'] = weights['charge_weight']
187
188     key = (step, beta)
189
190     self.charges_dict[key] = data['charges']
191     self.charge_diffs_dict[key] = data['charge_diffs']
192
193     self.train_data['loss'][key] = data['loss']
194     self.train_data['actions'][key] = data['actions']
195     self.train_data['plaqs'][key] = data['plaqs']
196     self.train_data['charges'][key] = data['charges']
197     self.train_data['charge_diffs'][key] = data['charge_diffs']
198     self.train_data['accept_probs'][key] = data['px']
199
200     self.train_data_strings.append(data_str)
201     if step % self.model.print_steps == 0:
202         io.log(data_str)
203
204     if self.summaries and (step + 1) % self.model.logging_steps == 0:
205         self.log_step(sess, step, data['samples'], beta, **weights)
206
207     if (step + 1) % self.model.save_steps == 0:
208         # self.model.save(self.sess, self.checkpoint_dir)
209         self.save_current_state()
210
211     if step % 100 == 0:
212         io.log(TRAIN_HEADER)
213
214     def write_train_strings(self):
215         """Write training strings out to file."""
216         tlf = self.train_log_file
217         _ = [io.write(s, tlf, 'a') for s in self.train_data_strings]

```

A.15 l2hmc-qcd/plotters/gauge_model_plotter.py

```
1 """
2 plotters.py
3
4 Implements GaugeModelPlotter class, responsible for loading and plotting
5 gauge model observables.
6
7 Author: Sam Foreman (github: @saforem2)
8 Date: 04/10/2019
9 """
10 import os
11 # import matplotlib as mpl
12 import numpy as np
13 import utils.file_io as io
14
15 from collections import Counter, OrderedDict
16 from scipy.stats import sem
17
18 from lattice.lattice import ul_plaq_exact
19 # from utils.data_loader import DataLoader
20 from globals import MARKERS, COLORS
21 from .plot_utils import plot_multiple_lines
22
23 try:
24     import matplotlib.pyplot as plt
25     HAS_MATPLOTLIB = True
26
27 except ImportError:
28     HAS_MATPLOTLIB = False
29
30
31 def arr_from_dict(d, key):
32     return np.array(list(d[key].values()))
33
34
35 def get_out_file(out_dir, out_str):
36     out_file = os.path.join(out_dir, out_str + '.pdf')
37     return out_file
38
39
40 class GaugeModelPlotter:
41
42     def __init__(self, model, figs_dir=None):
43         self.model = model
44         self.figs_dir = figs_dir
45
46     def calc_stats(self, data, therm_frac=10):
47         """Calculate observables statistics.
48
49         Args:
50             data (dict): Run data.
51             therm_frac (int): Percent of total steps to ignore to account for
52             thermalization.
53
54         Returns:
55             stats: Dictionary containing statistics for actions, plaquettes,
56             top. charges, and charge probabilities. For each of the
57             observables (actions, plaquettes, charges), the dictionary values
```

```

58     consist of a tuple of the form: (data, error), and
59     charge_probabilities is a dictionary of the form:
60         {charge_val: charge_val_probability}
61     """
62     actions = arr_from_dict(data, 'actions')
63     plaqs = arr_from_dict(data, 'plaqs')
64     charges = arr_from_dict(data, 'charges')
65
66     charge_probs = {}
67     counts = Counter(list(charges.flatten()))
68     total_counts = np.sum(list(counts.values()))
69     for key, val in counts.items():
70         charge_probs[key] = val / total_counts
71
72     charge_probs = OrderedDict(sorted(charge_probs.items(),
73                                     key=lambda k: k[0]))
74
75     def get_mean_err(x):
76         num_steps = x.shape[0]
77         therm_steps = num_steps // therm_frac
78         x = x[therm_steps:, :]
79         avg = np.mean(x, axis=0)
80         err = sem(x)
81         return avg, err
82
83     stats = {
84         'actions': get_mean_err(actions),
85         'plaqs': get_mean_err(plaqs),
86         'charges': get_mean_err(charges),
87         'suscept': get_mean_err(charges ** 2),
88         'charge_probs': charge_probs
89     }
90
91     return stats
92
93     def plot_observables(self, data, beta, run_str, **weights):
94         """Plot observables."""
95         actions = arr_from_dict(data, 'actions')
96         plaqs = arr_from_dict(data, 'plaqs')
97         charges = np.array(arr_from_dict(data, 'charges'), dtype=int)
98         charge_diffs = arr_from_dict(data, 'charge_diffs')
99         charge_autocorrs = np.array(data['charges_autocorrs'])
100        plaqs_diffs = plaqs - ul_plaq_exact(beta)
101
102        num_steps, num_samples = actions.shape
103        steps_arr = np.arange(num_steps)
104        # skip 5% of total number of steps
105        # between successive points when
106        # plotting to help smooth out graph
107        skip_steps = max((1, int(0.005 * num_steps)))
108        # ignore first 10% of pts (warmup)
109        warmup_steps = max((1, int(0.01 * num_steps)))
110
111        # _actions = actions[::-skip_steps]
112        # _plaqs = plaqs[::-skip_steps]
113
114        # _steps_arr = skip_steps * np.arange(_actions.shape[0])
115        _charge_diffs = charge_diffs[warmup_steps:][::skip_steps]
116        _plaqs_diffs = plaqs_diffs[warmup_steps:][::skip_steps]
117        _steps_diffs = (

```

```

118     skip_steps * np.arange(_plaqs_diffs.shape[0])) + skip_steps
119 )
120 # steps_diffs = np.arange(plaqs_diffs.shape[0])
121
122 # out_dir = (f'{int(num_steps)}_steps_{f"beta_{beta}"})
123 # self.out_dir = os.path.join(self.figs_dir, out_dir)
124 # self.out_dir = os.path.join
125 self.out_dir = os.path.join(self.figs_dir, run_str)
126 io.check_else_make_dir(self.out_dir)
127
128 L = self.model.space_size
129 lf_steps = self.model.num_steps
130 bs = self.model.num_samples
131 qw = weights['charge_weight']
132 # nw = weights['net_weight']
133
134 title_str = (r"$L = $" + f"${L}, "
135             r"$N_{\mathrm{LF}} = $" + f"${lf_steps}, "
136             r"$\alpha_Q = $" + f"${qw}, "
137             r"$\beta = $" + f"${beta}, "
138             r"$N_{\mathrm{samples}} = $" + f"${bs}$")
139
140 kwargs = {
141     'markers': False,
142     'lines': True,
143     'alpha': 0.6,
144     'title': title_str,
145     'legend': False,
146     'ret': False,
147     'out_file': [],
148 }
149
150 # self._plot_actions(_steps_arr, actions.T), **kwargs)
151 # self._plot_plaqs(_steps_arr, plaqs.T), beta, **kwargs)
152 self._plot_actions((steps_arr, actions.T), **kwargs)
153 self._plot_plaqs((steps_arr, plaqs.T), beta, **kwargs)
154 self._plot_charges((steps_arr, charges.T), **kwargs)
155 self._plot_autocorrs((steps_arr, charge_autocorrs), **kwargs)
156 self._plot_charge_probs(charges, **kwargs)
157 self._plot_charge_diffs((steps_diffs, _charge_diffs.T), **kwargs)
158 self._plot_plaqs_diffs((steps_diffs, _plaqs_diffs.T), **kwargs)
159 # self._plot_charge_diffs((steps_diffs, charge_diffs.T), **kwargs)
160 # self._plot_plaqs_diffs((steps_diffs, plaqs_diffs.T), **kwargs)
161
162 def _plot_actions(self, xy_data, **kwargs):
163     """Plot actions."""
164     kwargs['out_file'] = get_out_file(self.out_dir, 'actions_vs_step')
165     xy_labels = ('Step', 'Action')
166     plot_multiple_lines(xy_data, xy_labels, **kwargs)
167
168 def _plot_plaqs(self, xy_data, beta, **kwargs):
169     """Plot average plaquette."""
170     kwargs['out_file'] = None
171     kwargs['ret'] = True
172     xy_labels = ('Step', r"""\$ \langle \phi_P \rangle """)
173
174     _, ax = plot_multiple_lines(xy_data, xy_labels, **kwargs)
175     _ = ax.axhline(y=u1_plaq_exact(beta),
176                     color='#CC0033', ls='--', lw=1.5, label='exact')
177     _ = ax.plot(xy_data[0], xy_data[1].mean(axis=0), lw=1.25,

```

```

178             color='k', label='average', alpha=0.75)
179
180     _ = plt.tight_layout()
181
182     out_file = get_out_file(self.out_dir, 'plaqs_vs_step')
183     io.log(f'Saving figure to: {out_file}')
184     plt.savefig(out_file, dpi=400, bbox_inches='tight')
185
186     def _plot_plaqs_diffs(self, xy_data, **kwargs):
187         kwargs['out_file'] = None
188         kwargs['ret'] = True
189         xy_labels = ('Step', r"$\delta_{\phi_i}$")
190         _, ax = plot_multiple_lines(xy_data, xy_labels, **kwargs)
191         _ = ax.axhline(y=0, color='#CC0033', ls='-', lw=1.5)
192         _ = ax.plot(xy_data[0], xy_data[1].mean(axis=0), lw=1.25,
193                     color='k', label='average', alpha=0.75)
194         _ = ax.legend(loc='best')
195
196     _ = plt.tight_layout()
197
198     out_file = get_out_file(self.out_dir, 'plaqs_diffs_vs_step')
199     io.log(f'Saving figure to: {out_file}')
200     plt.savefig(out_file, dpi=400, bbox_inches='tight')
201
202     def _plot_charges(self, xy_data, **kwargs):
203         """Plot topological charges."""
204         kwargs['out_file'] = get_out_file(self.out_dir, 'charges_vs_step')
205         kwargs['markers'] = True
206         kwargs['lines'] = False
207         kwargs['alpha'] = 1.
208         kwargs['ret'] = False
209         xy_labels = ('Step', r'$Q$')
210         plot_multiple_lines(xy_data, xy_labels, **kwargs)
211
212         charges = np.array(xy_data[1].T, dtype=int)
213         num_steps, num_samples = charges.shape
214
215         out_dir = os.path.join(self.out_dir, 'top_charge_plots')
216         io.check_else_make_dir(out_dir)
217         # if we have more than 10 chains in charges, only plot first 10
218         for idx in range(min(num_samples, 5)):
219             _, ax = plt.subplots()
220             _ = ax.plot(charges[:, idx],
221                         marker=MARKERS[idx],
222                         color=COLORS[idx],
223                         ls='',
224                         alpha=0.5,
225                         label=f'sample {idx}')
226             _ = ax.legend(loc='best')
227             _ = ax.set_xlabel(xy_labels[0], fontsize=14)
228             _ = ax.set_ylabel(xy_labels[1], fontsize=14)
229             _ = ax.set_title(kwargs['title'], fontsize=16)
230             _ = plt.tight_layout()
231             out_file = get_out_file(out_dir, f'top_charge_vs_step_{idx}')
232             io.check_else_make_dir(os.path.dirname(out_file))
233             io.log(f'Saving figure to: {out_file}')
234             plt.savefig(out_file, dpi=400, bbox_inches='tight')
235
236             xi = 10
237             xf = charges.shape[0] // 10

```

```

238     _ = ax.set_xlim((xi, xf))
239     out_file = out_file.rstrip('.pdf') + '_zoom.pdf'
240     io.log(f'Saving figure to: {out_file}')
241     plt.savefig(out_file, dpi=400, bbox_inches='tight')
242
243     plt.close('all')
244
245
246     def _plot_charge_diffs(self, xy_data, **kwargs):
247         """Plot tunneling events (change in top. charge)."""
248         out_file = get_out_file(self.out_dir, 'top_charge_diffs')
249         steps_arr, charge_diffs = xy_data
250
251         # ignore first two data points when plotting since the top. charge
252         # should change dramatically for the very first few steps when starting
253         # from a random configuration
254         _, ax = plt.subplots()
255         _ = ax.plot(xy_data[0][2:], xy_data[1][2:],
256                     marker='.', ls='', fillstyle='none', color='C0')
257         _ = ax.set_xlabel('Steps', fontsize=14)
258         _ = ax.set_ylabel(r'$\Delta_Q$', fontsize=14)
259         _ = ax.set_title(kwargs['title'], fontsize=16)
260         _ = plt.tight_layout()
261         io.log(f"Saving figure to: {out_file}")
262         plt.savefig(out_file, dpi=400, bbox_inches='tight')
263
264     def _plot_charge_probs(self, charges, **kwargs):
265         """Plot top. charge probabilities."""
266         num_steps, num_samples = charges.shape
267         charges = np.array(charges, dtype=int)
268         out_dir = os.path.join(self.out_dir, 'top_charge_probs')
269         io.check_else_make_dir(out_dir)
270         if 'title' in list(kwargs.keys()):
271             title = kwargs.pop('title')
272             # if we have more than 10 chains in charges, only plot first 10
273         for idx in range(min(num_samples, 5)):
274             counts = Counter(charges[:, idx])
275             total_counts = np.sum(list(counts.values()))
276             _, ax = plt.subplots()
277             ax.plot(list(counts.keys()),
278                     np.array(list(counts.values()) / total_counts),
279                     marker=MARKERS[idx],
280                     color=COLORS[idx],
281                     ls='',
282                     label=f'sample {idx}')
283             _ = ax.legend(loc='best')
284             _ = ax.set_xlabel(r"$Q$") # , fontsize=14
285             _ = ax.set_ylabel('Probability') # , fontsize=14
286             _ = ax.set_title(title) # , fontsize=16
287             _ = plt.tight_layout()
288             out_file = get_out_file(out_dir, f'top_charge_vs_step_{idx}')
289             io.check_else_make_dir(os.path.dirname(out_file))
290             io.log(f"Saving plot to: {out_file}.")
291             plt.savefig(out_file, dpi=400, bbox_inches='tight')
292             # for f in out_file:
293             #     io.check_else_make_dir(os.path.dirname(f))
294             #     io.log(f"Saving plot to: {f}.")
295         plt.close('all')
296
297         all_counts = Counter(list(charges.flatten()))

```

```

298     total_counts = np.sum(list(counts.values()))
299     _, ax = plt.subplots()
300     ax.plot(list(all_counts.keys()),
301             np.array(list(all_counts.values())) / (total_counts *
302                                              num_samples),
303             marker='o',
304             color='C0',
305             ls='',
306             alpha=0.6,
307             label=f'total across {num_samples} samples')
308     _ = ax.legend(loc='best')
309     _ = ax.set_xlabel(r"$Q$") # , fontsize=14)
310     _ = ax.set_ylabel('Probability') # , fontsize=14)
311     _ = ax.set_title(title) # , fontsize=16)
312     _ = plt.tight_layout()
313     out_file = get_out_file(self.out_dir, f'TOP_CHARGE_PROBS_ALL')
314     io.check_else_make_dir(os.path.dirname(out_file))
315     io.log(f"Saving plot to: {out_file}.")
316     plt.savefig(out_file, dpi=400, bbox_inches='tight')
317     # for f in out_file:
318     plt.close('all')
319
320 def _plot_autocorrs(self, xy_data, **kwargs):
321     """Plot topological charge autocorrelations."""
322     try:
323         kwargs['out_file'] = get_out_file(
324             self.out_dir, 'charge_autocorrs_vs_step'
325         )
326     except AttributeError:
327         kwargs['out_file'] = None
328     xy_labels = ('Step', 'Autocorrelation of ' + r'$Q$')
329     return plot_multiple_lines(xy_data, xy_labels, **kwargs)

```

A.16 l2hmc-qcd/plotters/leapfrog_plotters.py

```
1 import time
2
3 import os
4 try:
5     import psutil
6     HAS_PSUTIL = True
7 except ImportError:
8     HAS_PSUTIL = False
9
10 import pickle
11 import numpy as np
12 try:
13     import matplotlib as mpl
14     import matplotlib.pyplot as plt
15     HAS_MATPLOTLIB = True
16 except ImportError:
17     HAS_MATPLOTLIB = False
18
19 # from .formatters import latexify
20 import utils.file_io as io
21 from utils.data_loader import DataLoader
22
23
24 def load_and_sep(out_file, keys=('forward', 'backward')):
25     with open(out_file, 'rb') as f:
26         data = pickle.load(f)
27
28     return (np.array(data[k]) for k in keys)
29
30
31 params = {
32     # 'backend': 'ps',
33     # 'text.latex.preamble': [r'\usepackage{gensymb}'],
34     'axes.labelsize': 16,    # fontsize for x and y labels (was 10)
35     'axes.titlesize': 16,
36     'legend.fontsize': 10,   # was 10
37     'xtick.labelsize': 12,
38     'ytick.labelsize': 12,
39     # 'text.usetex': True,
40     # 'figure.figsize': [fig_width, fig_height],
41     'font.family': 'serif'
42 }
43
44 try:
45     mpl.rcParams.update(params)
46 except FileNotFoundError:
47     params['text.usetex'] = False
48     params['text.latex.preamble'] = None
49     try:
50         mpl.rcParams.update(params)
51     except FileNotFoundError:
52         pass
53
54
55 class LeapfrogPlotter:
56     def __init__(self, figs_dir, run_logger=None,
57                  run_dir=None, therm_perc=0.005, skip_perc=0.01):
```

```

58     self.figs_dir = figs_dir
59     self.pdfs_dir = os.path.join(self.figs_dir, 'pdfs_plots')
60     io.check_else_make_dir(self.pdfs_dir)
61
62     if run_logger is None:
63         if run_dir is None:
64             raise AttributeError(
65                 """Either a `run_logger` object containing data or a
66                 `run_dir` from which to load data must be specified.
67                 Exiting.
68                 """
69             )
70     else:
71         try:
72             data = self.load_data(run_dir)
73             self.samples = data[0]
74             self.lf_f, self.lf_b = data[1]
75             self.logdets_f, self.logdets_b = data[2]
76             self.sumlogdet_f, self.sumlogdet_b = data[3]
77         except FileNotFoundError:
78             io.log(f'''Unable to load leapfrog data from run_dir:
79                     {run_dir}. Exiting.''')
79             return
80
81     else:
82         self.samples = np.array(run_logger.samples_arr)
83         self.lf_f = np.array(run_logger.lf_out['forward'])
84         self.lf_b = np.array(run_logger.lf_out['backward'])
85         self.logdets_f = np.array(run_logger.logdets['forward'])
86         self.logdets_b = np.array(run_logger.logdets['backward'])
87         self.sumlogdet_f = np.array(run_logger.sumlogdet['forward'])
88         self.sumlogdet_b = np.array(run_logger.sumlogdet['backward'])
89
90         self.lf_f_diffs = 1. - np.cos(self.lf_f[1:] - self.lf_f[:-1])
91         self.lf_b_diffs = 1. - np.cos(self.lf_b[1:] - self.lf_b[:-1])
92         self.samples_diffs = 1. - np.cos(self.samples[1:] - self.samples[:-1])
93         self.tot_lf_steps = self.lf_f_diffs.shape[0]
94         self.tot_md_steps = self.samples_diffs.shape[0]
95         self.num_lf_steps = self.tot_lf_steps // self.tot_md_steps
96         self.indiv_kw_args = {
97             'ls': '-',
98             'alpha': 0.5,
99             'lw': 0.5,
100            'rasterized': True
101        }
102
103     self.avg_kw_args = {
104         'ls': '-',
105         'alpha': 0.6,
106         'lw': 0.75,
107         'rasterized': True
108     }
109     plot_lf_steps = self.tot_lf_steps // 4
110     plot_md_steps = self.tot_md_steps // 4
111     self.lf_f_diffs = self.lf_f_diffs[:plot_lf_steps]
112     self.lf_b_diffs = self.lf_b_diffs[:plot_lf_steps]
113     self.samples_diffs = self.samples_diffs[:plot_md_steps]
114     self.logdets_f = self.logdets_f[:plot_lf_steps]
115     self.logdets_b = self.logdets_b[:plot_lf_steps]
116     self.sumlogdet_f = self.sumlogdet_f[:plot_md_steps]

```

```

118     self.sumlogdet_b = self.sumlogdet_b[:plot_md_steps]
119
120     # self.therm_steps = int(therm_perc * self.tot_lf_steps)
121     # self.skip_steps = int(skip_perc * self.tot_lf_steps)
122     # self.step_multiplier = (
123     #     self.lf_f_diffs.shape[0] // self.samples_diffs.shape[0]
124     # )
125
126 def load_data(self, run_dir):
127     loader = DataLoader(run_dir)
128     io.log("Loading samples...")
129     samples = loader.load_samples(run_dir)
130     io.log('done.')
131     io.log("Loading leapfrogs...")
132     leapfrogs = loader.load_leapfrogs(run_dir)
133     io.log("Loading logdets...")
134     logdets = loader.load_logdets(run_dir)
135     io.log("Loading sumlogdets...")
136     sumlogdets = loader.load_sumlogdets(run_dir)
137     return (samples, leapfrogs, logdets, sumlogdets)
138
139 def print_memory(self):
140     if HAS_PSUTIL:
141         pid = os.getpid()
142         py = psutil.Process(pid)
143         memory_use = py.memory_info()[0] / 2. ** 30
144         io.log(80 * '-')
145         io.log(f'memory use: {memory_use}')
146         io.log(80 * '-')
147
148 def get_colors(self, num_samples=20):
149     reds_cmap = mpl.cm.get_cmap('Reds', num_samples + 1)
150     blues_cmap = mpl.cm.get_cmap('Blues', num_samples + 1)
151     idxs = np.linspace(0.2, 0.75, num_samples + 1)
152     reds = [reds_cmap(i) for i in idxs]
153     blues = [blues_cmap(i) for i in idxs]
154
155     return reds, blues
156
157 def update_figs_dir(self, figs_dir):
158     self.figs_dir = figs_dir
159     self.pdfs_dir = os.path.join(self.figs_dir, 'pdfs')
160
161     io.check_else_make_dir(figs_dir)
162     io.check_else_make_dir(self.pdfs_dir)
163
164 def make_plots(self, run_dir, num_samples=20, ret=False):
165     """Make plots of the leapfrog differences and logdets.
166
167     Immediately after creating and saving the plots, delete these
168     (no-longer needed) attributes to free up memory.
169
170     Args:
171         run_dir (str): Path to directory in which to save all of the
172             relevant instance attributes.
173         num_samples (int): Number of samples to include when creating
174             plots.
175         save (bool): Boolean indicating whether or not plotted data should
176             be saved.
177

```

```

178     NOTE:
179         `save` is very data intensive and will produce LARGE (compressed)
180         `.npz` files.
181     """
182     run_key = run_dir.split('/')[-1].split('_')
183     beta_idx = run_key.index('beta') + 1
184     beta = run_key[beta_idx]
185
186     self.print_memory()
187     fig_ax1 = self.plot_lf_diffs(beta, num_samples)
188
189     self.print_memory()
190     fig_ax2 = self.plot_logdets(beta, num_samples)
191
192     if ret:
193         return fig_ax1, fig_ax2
194
195     def plot_lf_diffs(self, beta, num_samples=20):
196         t0 = time.time()
197         reds, blues = self.get_colors(num_samples)
198         samples_y_avg = np.mean(self.samples_diffs, axis=(1, 2))
199         samples_x_avg = np.arange(len(samples_y_avg))
200
201         fig, (ax1, ax2) = plt.subplots(2, 1)
202         ax1.set_rasterization_zorder(1)
203         ax2.set_rasterization_zorder(1)
204
205         for idx in range(num_samples):
206             yf = np.mean(self.lf_f_diffs, axis=-1)
207             xf = np.arange(len(yf))
208             yb = np.mean(self.lf_b_diffs, axis=-1)
209             xb = np.arange(len(yb))
210
211             _ = ax1.plot(xf, yf[:, idx], color=reds[idx], **self.indiv_kw_args)
212             _ = ax1.plot(xb, yb[:, idx], color=blues[idx], **self.indiv_kw_args)
213
214         yf_avg = np.mean(self.lf_f_diffs, axis=(1, 2))
215         yb_avg = np.mean(self.lf_b_diffs, axis=(1, 2))
216         xf_avg = np.arange(len(yf))
217         xb_avg = np.arange(len(yb))
218
219         _ = ax1.plot(xf_avg, yf_avg, label='forward',
220                     color='r', **self.avg_kw_args)
221         _ = ax1.plot(xb_avg, yb_avg, label='backward',
222                     color='b', **self.avg_kw_args)
223
224         _ = ax2.plot(samples_x_avg, samples_y_avg, label='MD avg.',
225                     color='k', lw=1., rasterized=True, ls='--')
226
227         _ = ax1.set_xlabel('Leapfrog step', fontsize=16)
228         _ = ax2.set_xlabel('MD step', fontsize=16)
229
230         ylabel = r'$\langle \delta\phi_{\mu}(i) \rangle$'
231         _ = ax1.set_ylabel(ylabel, fontsize=16)
232         _ = ax2.set_ylabel(ylabel, fontsize=16)
233
234         _ = ax1.legend(loc='best', fontsize=10)
235         _ = ax2.legend(loc='best', fontsize=10)
236         fig.tight_layout()
237         # fig.subplots_adjust(hspace=0.5)

```

```

238
239     beta_str = str(beta).replace('.', '')
240     fn = f'leapfrog_diffs_beta{beta_str}'
241     out_file = os.path.join(self.pdfs_dir, fn + '.pdf')
242     out_file_zoom = os.path.join(self.pdfs_dir, fn + '_zoom.pdf')
243     out_file_zoom1 = os.path.join(self.pdfs_dir, fn + '_zoom1.pdf')
244     io.log(f'Saving figure to: {out_file}')
245     _ = plt.savefig(out_file, dpi=400, bbox_inches='tight')
246
247     lf_xlim = 100
248     md_xlim = lf_xlim // self.num_lf_steps
249
250     _ = ax1.set_xlim((0, lf_xlim))
251     _ = ax2.set_xlim((0, md_xlim))
252     _ = plt.savefig(out_file_zoom, dpi=400, bbox_inches='tight')
253
254     lf_xlim /= 4
255     md_xlim /= 4
256     _ = ax1.set_xlim((0, lf_xlim))
257     _ = ax1.set_ylim((-0.05, 0.25))
258     _ = ax2.set_xlim((0, md_xlim))
259     _ = plt.savefig(out_file_zoom1, dpi=400, bbox_inches='tight')
260
261     io.log(80 * '-')
262     io.log(f'Time spent plotting lf_diffs: {time.time() - t0}s')
263     io.log(80 * '-')
264
265     return fig, (ax1, ax2)
266
267 def plot_logdets(self, beta, num_samples=20):
268     t0 = time.time()
269     reds, blues = self.get_colors(num_samples)
270
271     sumlogdet_yf_avg = np.mean(self.sumlogdet_f, axis=-1)
272     sumlogdet_yb_avg = np.mean(self.sumlogdet_b, axis=-1)
273     sumlogdet_xf_avg = np.arange(len(sumlogdet_yf_avg))
274     sumlogdet_xb_avg = np.arange(len(sumlogdet_yb_avg))
275
276     fig, (ax1, ax2) = plt.subplots(2, 1)
277     for idx in range(num_samples):
278         yf = self.logdets_f[:, idx]
279         xf = np.arange(len(yf))
280         yb = self.logdets_b[:, idx]
281         xb = np.arange(len(yb))
282
283         _ = ax1.plot(xf, yf, color=reds[idx], **self.indiv_kwargs)
284         _ = ax1.plot(xb, yb, color=blues[idx], **self.indiv_kwargs)
285
286     yf_avg = np.mean(self.logdets_f, axis=-1)
287     yb_avg = np.mean(self.logdets_b, axis=-1)
288
289     xf_avg = np.arange(len(yf_avg))
290     xb_avg = np.arange(len(yb_avg))
291
292     _ = ax1.plot(xf_avg, yf_avg, label='forward',
293                  color='r', **self.avg_kwargs)
294     _ = ax1.plot(xb_avg, yb_avg, label='backward',
295                  color='b', **self.avg_kwargs)
296
297     _ = ax2.plot(sumlogdet_xf_avg, sumlogdet_yf_avg, label='forward',

```

```

298             color='r', lw=1.2, alpha=0.9, marker='.')
299
300     _ = ax2.plot(sumlogdet_xb_avg, sumlogdet_yb_avg, label='backward',
301                 color='b', lw=1.2, alpha=0.9, marker='.')
302
303     _ = ax1.set_xlabel('Leapfrog step', fontsize=16)
304     _ = ax1.set_ylabel(r'$\log|\mathcal{J}|^{(t)}|$', fontsize=16)
305     _ = ax2.set_xlabel('MD step', fontsize=16)
306     _ = ax2.set_ylabel(r'$\sum_t \log|\mathcal{J}|^{(t)}|$', fontsize=16)
307
308     _ = ax1.legend(loc='best', fontsize=10)
309     _ = ax2.legend(loc='best', fontsize=10)
310     _ = fig.tight_layout()
311     beta_str = str(beta).replace('.', '')
312     fn = f'avg_logdets_beta{beta_str}'
313     out_file = os.path.join(self.pdfs_dir, fn + '.pdf')
314     out_file_zoom = os.path.join(self.pdfs_dir, fn + '_zoom.pdf')
315     out_file_zoom1 = os.path.join(self.pdfs_dir, fn + '_zoom1.pdf')
316     io.log(f'Saving figure to: {out_file}')
317     _ = plt.savefig(out_file, dpi=400, bbox_inches='tight')
318
319     lf_xlim = 100
320     md_xlim = lf_xlim // self.num_lf_steps
321
322     _ = ax1.set_xlim((0, lf_xlim))
323     _ = ax2.set_xlim((0, md_xlim))
324     _ = plt.savefig(out_file_zoom, dpi=400, bbox_inches='tight')
325
326     lf_xlim //= 4
327     md_xlim //= 4
328     _ = ax1.set_xlim((0, lf_xlim))
329     _ = ax2.set_xlim((0, md_xlim))
330     _ = plt.savefig(out_file_zoom1, dpi=400, bbox_inches='tight')
331
332     io.log(80 * '-')
333     io.log(f'Time spent plotting log_dets: {time.time() - t0}s')
334     io.log(80 * '-')
335
336     return fig, (ax1, ax2)

```

A.17 l2hmc-qcd/plotters/plot_utils.py

```
1 """
2 plot_utils.py
3
4 Collection of helper methods used for creating plots in Matplotlib.
5
6 Author: Sam Foreman (github: @saforem2)
7 Date: 06/16/2019
8 """
9 import os
10
11 import numpy as np
12 import matplotlib as mpl
13 try:
14     import matplotlib.pyplot as plt
15     HAS_MATPLOTLIB = True
16 except ImportError:
17     HAS_MATPLOTLIB = False
18
19 from globals import MARKERS
20 import utils.file_io as io
21
22
23 params = {
24     # 'backend': 'ps',
25     # 'text.latex.preamble': [r'\usepackage{gensymb}'],
26     'axes.labelsize': 14,    # fontsize for x and y labels (was 10)
27     'axes.titlesize': 16,
28     'legend.fontsize': 10,   # was 10
29     'xtick.labelsize': 12,
30     'ytick.labelsize': 12,
31     # 'text.usetex': True,
32     # 'figure.figsize': [fig_width, fig_height],
33     # 'font.family': 'serif',
34 }
35
36 mpl.rcParams.update(params)
37
38
39 def get_colors(num_samples=10, cmaps=None):
40     if cmaps is None:
41         cmap0 = mpl.cm.get_cmap('Greys', num_samples + 1)
42         cmap1 = mpl.cm.get_cmap('Reds', num_samples + 1)
43         cmap2 = mpl.cm.get_cmap('Blues', num_samples + 1)
44         cmaps = (cmap0, cmap1, cmap2)
45
46     idxs = np.linspace(0.1, 0.75, num_samples + 1)
47     colors_arr = []
48     for cmap in cmaps:
49         colors_arr.append([cmap(i) for i in idxs])
50
51     # colors0 = [cmap0(i) for i in idxs]
52     # colors1 = [cmap1(i) for i in idxs]
53     # cmap0 = mpl.cm.get_cmap(cmaps[0], num_samples + 1)
54     # cmap1 = mpl.cm.get_cmap(cmaps[1], num_samples + 1)
55     # reds_cmap = mpl.cm.get_cmap('Reds', num_samples + 1)
56     # blues_cmap = mpl.cm.get_cmap('Blues', num_samples + 1)
```

```

58     # return colors0, colors1
59     return colors_arr
60
61
62 def plot_multiple_lines(data, xy_labels, **kwargs):
63     """Plot multiple lines along with their average."""
64     out_file = kwargs.get('out_file', None)
65     markers = kwargs.get('markers', False)
66     lines = kwargs.get('lines', True)
67     alpha = kwargs.get('alpha', 1.)
68     legend = kwargs.get('legend', False)
69     title = kwargs.get('title', None)
70     ret = kwargs.get('ret', False)
71     num_samples = kwargs.get('num_samples', 10)
72     greys, reds, blues = get_colors(num_samples)
73     if isinstance(data, list):
74         data = np.array(data)
75
76     try:
77         x_data, y_data = data
78     except (IndexError, ValueError):
79         x_data = np.arange(data.shape[0])
80         y_data = data
81
82     x_label, y_label = xy_labels
83
84     if y_data.shape[0] > num_samples:
85         y_sample = y_data[:num_samples, :]
86     else:
87         y_sample = y_data
88
89     fig, ax = plt.subplots()
90
91     marker = None
92     ls = '-'
93     fillstyle = 'full'
94     for idx, row in enumerate(y_sample):
95         if markers:
96             marker = MARKERS[idx]
97             fillstyle = 'none'
98             ls = '-'
99         if not lines:
100             ls = ''
101         _ = ax.plot(x_data, row, # label=f'sample {idx}',
102                     fillstyle=fillstyle, marker=marker,
103                     ls=ls, lw=0.5, alpha=alpha, color=greys[idx])
104
105     _ = ax.plot(
106         x_data, y_data.mean(axis=0), label='average',
107         alpha=1., lw=1.0, color='k'
108     )
109
110     ax.set_xlabel(x_label, fontsize=14)
111     ax.set_ylabel(y_label, fontsize=14)
112     plt.tight_layout()
113     if legend:
114         ax.legend(loc='best')
115     if title is not None:
116         ax.set_title(title)
117     if out_file is not None:

```

```
118     out_dir = os.path.dirname(out_file)
119     io.check_else_make_dir(out_dir)
120     io.log(f'Saving figure to {out_file}.')
121     fig.savefig(out_file, dpi=400, bbox_inches='tight')
122
123     if ret:
124         return fig, ax
125
126     return 1
```

A.18 l2hmc-qcd/utils/parse_args.py

```
1 """
2 parse_args.py
3
4 Implements method for parsing command line arguments for `gauge_model.py`
5
6 Author: Sam Foreman (github: @saforem2)
7 Date: 04/09/2019
8 """
9 import os
10 import sys
11 import argparse
12 import shlex
13
14 import utils.file_io as io
15
16 # from config import process_config
17 # from attr_dict import AttrDict
18
19 DESCRIPTION = (
20     'L2HMC model using U(1) lattice gauge theory for target distribution.'
21 )
22
23
24 def get_args():
25     argparser = argparse.ArgumentParser(description=DESCRIPTION)
26     argparser.add_argument(
27         '-c', '--config',
28         metavar='C',
29         default=None,
30         help='The configuration file'
31     )
32     args = argparser.parse_args()
33
34     return args
35
36
37 # =====
38 # * NOTE:
39 #     - if action == 'store_true':
40 #         The argument is FALSE by default. Passing this flag will cause the
41 #         argument to be ''stored true''.
42 #     - if action == 'store_false':
43 #         The argument is TRUE by default. Passing this flag will cause the
44 #         argument to be ''stored false''.
45 # =====
46 def parse_args():
47     """Parse command line arguments."""
48     parser = argparse.ArgumentParser(
49         description=DESCRIPTION,
50         fromfile_prefix_chars='@',
51     )
52
53     # -----
54     # Lattice parameters
55     # -----
56
57     parser.add_argument("--space_size",
```

```

58             dest="space_size",
59             type=int,
60             default=8,
61             required=False,
62             help="""Spatial extent of lattice.""")
63
64     parser.add_argument("--time_size",
65                         dest="time_size",
66                         type=int,
67                         default=8,
68                         required=False,
69                         help="""Temporal extent of lattice.""")
70
71     parser.add_argument("--link_type",
72                         dest="link_type",
73                         type=str,
74                         required=False,
75                         default='U1',
76                         help="""Link type for gauge model.""")
77
78     parser.add_argument("--dim",
79                         dest="dim",
80                         type=int,
81                         required=False,
82                         default=2,
83                         help="""Dimensionality of lattice.""")
84
85     parser.add_argument("--num_samples",
86                         dest="num_samples",
87                         type=int,
88                         default=10,
89                         required=False,
90                         help=("""Number of samples (batch size) to use for
91                               training.""))
92
93     parser.add_argument("--rand",
94                         dest="rand",
95                         action="store_true",
96                         required=False,
97                         help=("""Start lattice from randomized initial
98                               configuration.""))
99
100    # -----
101    # Leapfrog parameters
102    # -----
103
104    parser.add_argument("-n", "--num_steps",
105                         dest="num_steps",
106                         type=int,
107                         default=5,
108                         required=False,
109                         help=("""Number of leapfrog steps to use in (augmented)
110                               HMC sampler.""))
111
112    parser.add_argument("--for_loop",
113                         dest="for_loop",
114                         action="store_true",
115                         required=False,
116                         help=("""When passed, perform leapfrog updates using
117                               generic `for` loop. Otherwise, use (optimized)

```

```

118     `tf.while_loop`.""))
119
120     parser.add_argument("--eps",
121                         dest="eps",
122                         type=float,
123                         default=0.1,
124                         required=False,
125                         help="""Step size to use in leapfrog integrator.""")
126
127     parser.add_argument("--loss_scale",
128                         dest="loss_scale",
129                         type=float,
130                         default=1.,
131                         required=False,
132                         help=("""Scaling factor to be used in loss function.
133                               (lambda in Eq. 7 of paper)."""))
134
135 # -----
136 # Learning rate parameters
137 # -----
138
139     parser.add_argument("--lr_init",
140                         dest="lr_init",
141                         type=float,
142                         default=1e-3,
143                         required=False,
144                         help="""Initial value of learning rate.""")
145
146     parser.add_argument("--lr_decay_steps",
147                         dest="lr_decay_steps",
148                         type=int, default=500,
149                         required=False,
150                         help=("""Number of steps after which to decay learning
151                               rate.""""))
152
153     parser.add_argument("--lr_decay_rate",
154                         dest="lr_decay_rate",
155                         type=float, default=0.96,
156                         required=False,
157                         help=("""Learning rate decay rate to be used during
158                               training.""""))
159
160 # -----
161 # Annealing rate parameters
162 # -----
163
164     parser.add_argument("--annealing",
165                         dest="annealing",
166                         action="store_true",
167                         required=False,
168                         help=("""Flag that when passed will cause the model
169                               to perform simulated annealing during
170                               training.""""))
171
172     parser.add_argument("--hmc_beta",
173                         dest="hmc_beta",
174                         type=float,
175                         default=None,
176                         required=False,
177                         help=("""Flag specifying a singular value of beta at

```

```

178             which to run the generic HMC sampler.""))
179
180     parser.add_argument("--hmc_eps",
181                         dest="hmc_eps",
182                         type=float,
183                         default=None,
184                         required=False,
185                         help="""Flag specifying a singular step size value
186                               (`eps`) to use when running the generic HMC
187                               sampler."""))
188
189     parser.add_argument("--beta_init",
190                         dest="beta_init",
191                         type=float,
192                         default=1.,
193                         required=False,
194                         help="""Initial value of beta (inverse coupling
195                               constant) used in gauge model when
196                               annealing."""))
197
198     parser.add_argument("--beta_final",
199                         dest="beta_final",
200                         type=float,
201                         default=8.,
202                         required=False,
203                         help="""Final value of beta (inverse coupling
204                               constant) used in gauge model when
205                               annealing."""))
206
207 # -----
208 # Training parameters
209 # -----
210
211     parser.add_argument("--train_steps",
212                         dest="train_steps",
213                         type=int,
214                         default=5000,
215                         required=False,
216                         help="""Number of training steps to perform.""")
217
218     parser.add_argument("--run_steps",
219                         dest="run_steps",
220                         type=int,
221                         default=50000,
222                         required=False,
223                         help="""Number of evaluation 'run' steps to perform
224                               after training (i.e. length of desired chain
225                               generate using trained L2HMC sampler.)"""))
226
227     parser.add_argument("--trace",
228                         dest="trace",
229                         action="store_true",
230                         required=False,
231                         help="""Flag that when passed will create trace during
232                               training loop."""))
233
234     parser.add_argument("--save_steps",
235                         dest="save_steps",
236                         type=int,
237                         default=50,

```

```

238     required=False,
239     help="""Number of steps after which to save the model
240         and current values of all parameters."""))
241
242 parser.add_argument("--print_steps",
243     dest="print_steps",
244     type=int,
245     default=1,
246     required=False,
247     help="""Number of steps after which to display
248         information about the loss and various
249         other quantities.""")
250
251 parser.add_argument("--logging_steps",
252     dest="logging_steps",
253     type=int,
254     default=50,
255     required=False,
256     help="""Number of steps after which to write logs for
257         tensorboard.""")
258
259 # -----
260 # Model parameters
261 # -----
262
263 parser.add_argument('--network_arch',
264     dest='network_arch',
265     type=str,
266     default='conv3D',
267     required=False,
268     help="""String specifying the architecture to use for
269         the neural network. Must be one of:
270         `conv3D`, `conv2D`, `generic`""")
271
272 parser.add_argument('--num_hidden',
273     dest='num_hidden',
274     type=int,
275     default=None,
276     required=False,
277     help="""Number of nodes to include in fully-connected
278         hidden layer `h`. If not explicitly passed, will
279         default to 2 * lattice.num_links.""")
280
281 parser.add_argument('--summaries',
282     dest="summaries",
283     action="store_true",
284     required=False,
285     help="""Flag that when passed creates
286         summaries of gradients and variables for
287         monitoring in tensorboard.""")
288
289 parser.add_argument('--save_samples',
290     dest='save_samples',
291     action='store_true',
292     required=False,
293     help="""Flag that when passed will save the samples
294         generated during the `run` phase.
295         WARNING: This is very data intensive.""")
296
297 parser.add_argument('--save_lf',

```

```

298     dest='save_lf',
299     action='store_true',
300     required=False,
301     help="""Flag that when passed will save the
302         output from each leapfrog step""")
303
304     parser.add_argument('--loop_net_weights',
305                         dest='loop_net_weights',
306                         action='store_true',
307                         help="""Flag that when passed will iterate over
308                 multiple values of `net_weights`, which are
309                 multiplicative scaling factors applied to each of
310                 the Q, S, T functions when running the trained
311                 sampler.""""))
312
313     parser.add_argument('--long_run',
314                         dest='long_run',
315                         action='store_true',
316                         required=False,
317                         help="""Flag that when passed runs the trained sampler
318                 at model.beta_final and model.beta_final +
319                 1."""))
320
321     parser.add_argument('--hmc',
322                         dest="hmc",
323                         action="store_true",
324                         required=False,
325                         help="""Use generic HMC (without augmented leapfrog
326                 integrator described in paper). Used for
327                 comparing against L2HMC algorithm."""))
328
329     parser.add_argument('--run_hmc',
330                         dest="run_hmc",
331                         action="store_true",
332                         required=False,
333                         help="""Flag that when passed causes generic HMC
334                 to be ran after running the trained L2HMC
335                 sampler. (Default: False)"""))
336
337     parser.add_argument('--eps_trainable',
338                         dest="eps_trainable",
339                         action="store_true",
340                         required=False,
341                         help="""Flag that when passed will allow the step size
342                 `eps` to be a trainable parameter.""""))
343
344     parser.add_argument('--metric',
345                         dest="metric",
346                         type=str,
347                         default="cos_diff",
348                         required=False,
349                         help="""Metric to use in loss function. Must be one
350                 of: `l1`, `l2`, `cos`, `cos2`, `cos_diff`."""))
351
352     parser.add_argument('--std_weight',
353                         dest="std_weight",
354                         type=float,
355                         default=1.,
356                         required=False,
357                         help="""Multiplicative factor used to weigh relative

```

```

358                                         strength of stdiliary term in loss function.""))
359
360     parser.add_argument("--aux_weight",
361                         dest="aux_weight",
362                         type=float,
363                         default=1.,
364                         required=False,
365                         help=( """Multiplicative factor used to weigh relative
366                               strength of auxiliary term in loss function.""" ))
367
368     parser.add_argument("--charge_weight",
369                         dest="charge_weight",
370                         type=float,
371                         default=1.,
372                         required=False,
373                         help=( """Multiplicative factor used to weigh relative
374                               strength of top. charge term in loss
375                               function"""))
376
377     parser.add_argument("--clip_grads",
378                         dest="clip_grads",
379                         action="store_true",
380                         required=False,
381                         help=( """Flag that when passed will clip gradients by
382                               global norm using `--clip_value` command line
383                               argument. If `--clip_value` is not passed,
384                               it defaults to 100.""" ))
385
386     parser.add_argument("--clip_value",
387                         dest="clip_value",
388                         type=float,
389                         default=1.,
390                         required=False,
391                         help=( """Clip value, used for clipping value of
392                               gradients by global norm.""" ))
393
394     parser.add_argument("--log_dir",
395                         dest="log_dir",
396                         type=str,
397                         default=None,
398                         required=False,
399                         help=( """Log directory to use from previous run.
400                               If this argument is not passed, a new
401                               directory will be created.""" ))
402
403     parser.add_argument("--restore",
404                         dest="restore",
405                         action="store_true",
406                         required=False,
407                         help=( """Restore model from previous run. If this
408                               argument is passed, a `log_dir` must be specified
409                               and passed to `--log_dir` argument.""" ))
410
411     parser.add_argument("--profiler",
412                         dest='profiler',
413                         action="store_true",
414                         required=False,
415                         help=( """Flag that when passed will profile the graph
416                               execution using `TFProf`.""" ))
417

```

```

418     parser.add_argument("--gpu",
419                         dest="gpu",
420                         action="store_true",
421                         required=False,
422                         help="""Flag that when passed indicates we're training
423                               using an NVIDIA GPU."""))
424
425     parser.add_argument("--theta",
426                         dest="theta",
427                         action="store_true",
428                         required=False,
429                         help="""Flag that when passed indicates we're training
430                               on theta @ ALCf."""))
431
432     parser.add_argument("--use_bn",
433                         dest='use_bn',
434                         action="store_true",
435                         required=False,
436                         help="""Flag that when passed causes batch
437                               normalization layer to be used in ConvNet."""))
438
439     parser.add_argument("--horovod",
440                         dest="horovod",
441                         action="store_true",
442                         required=False,
443                         help="""Flag that when passed uses Horovod for
444                               distributed training on multiple nodes."""))
445
446     parser.add_argument("--num_intra_threads",
447                         dest="num_intra_threads",
448                         type=int,
449                         default=0,
450                         required=False,
451                         help="""Number of intra op threads to use for
452                               tf.ConfigProto.intra_op_parallelism_THREADS"""))
453
454     parser.add_argument("--num_inter_threads",
455                         dest="num_intra_threads",
456                         type=int,
457                         default=0,
458                         required=False,
459                         help="""Number of intra op threads to use for
460                               tf.ConfigProto.intra_op_parallelism_THREADS"""))
461
462     parser.add_argument("--float64",
463                         dest="float64",
464                         action="store_true",
465                         required=False,
466                         help="""When passed, using 64 point floating precision
467                               by settings globals.TF_FLOAT = tf.float64. False
468                               by default (use tf.float32.)""")
469
470     if sys.argv[1].startswith('@'):
471         args = parser.parse_args(shlex.split(open(sys.argv[1][1:]).read(),
472                                           comments=True))
473     else:
474         args = parser.parse_args()
475
476     return args

```

A.19 l2hmc-qcd/utils/file_io.py

```
1 """
2 Helper methods for performing file IO.
3
4 Author: Sam Foreman (github: @saforem2)
5 Created: 2/27/2019
6 """
7 from __future__ import absolute_import
8 from __future__ import division
9 from __future__ import print_function
10 import os
11 import datetime
12 import pickle
13 import numpy as np
14
15 # pylint:disable=invalid-name
16
17 try:
18     import horovod.tensorflow as hvd
19
20     HAS_HOROVOD = True
21     hvd.init()
22
23 except ImportError:
24     HAS_HOROVOD = False
25
26 from globals import FILE_PATH
27
28
29 def log(s, nl=True):
30     """Print string `s` to stdout if and only if hvd.rank() == 0."""
31     try:
32         if HAS_HOROVOD and hvd.rank() != 0:
33             return
34         print(s, end='\n' if nl else '')
35     except NameError:
36         print(s, end='\n' if nl else '')
37
38
39 def write(s, f, mode='a', nl=True):
40     """Write string `s` to file `f` if and only if hvd.rank() == 0."""
41     try:
42         if HAS_HOROVOD and hvd.rank() != 0:
43             return
44         with open(f, mode) as ff:
45             ff.write(s + '\n' if nl else '')
46     except NameError:
47         with open(f, mode) as ff:
48             ff.write(s + '\n' if nl else '')
49
50
51 def log_and_write(s, f):
52     """Print string `s` to std out and also write to file `f`."""
53     log(s)
54     write(s, f)
55
56
57 def create_log_dir(FLAGS, root_dir=None, log_file=None):
```

```

58     """Automatically create and name `log_dir` to save model data to.
59
60     The created directory will be located in `logs/YYYY_M_D/`, and will have
61     the format (without `_{QW}` if running generic HMC):
62
63     `lattice{LX}_batch{NS}_lf{LF}_eps{SS}_{QW}{QW}`
64
65     Returns:
66         FLAGS, with FLAGS.log_dir being equal to the newly created log_dir.
67
68     NOTE: If log_dir does not already exist, it is created.
69     """
70
71     LX = FLAGS.space_size
72     NS = FLAGS.num_samples
73     LF = FLAGS.num_steps
74     # SS = str(FLAGS.eps).lstrip('0.')
75     SS = FLAGS.eps
76     QW = FLAGS.charge_weight
77     if FLAGS.hmc:
78         run_str = f'HMC_lattice{LX}_batch{NS}_lf{LF}_eps{SS:.3g}'
79     else:
80         run_str = f'lattice{LX}_batch{NS}_lf{LF}_eps{SS:.3g}_{QW}{QW}'
81
82     now = datetime.datetime.now()
83     # print(now.strftime("%b %d %Y %H:%M:%S"))
84     day_str = now.strftime('%Y_%m_%d')
85     time_str = now.strftime("%Y_%m_%d_%H%M")
86
87     # day_str = f'{now.year}-{now.month}-{now.day}'
88     # time_str = day_str + f'_{now.hour}{now.minute}'
89     project_dir = os.path.abspath(os.path.dirname(FILE_PATH))
90     if FLAGS.log_dir is None:
91         if root_dir is None:
92             _dir = 'logs'
93         else:
94             _dir = root_dir
95     else:
96         if root_dir is None:
97             _dir = FLAGS.log_dir
98         else:
99             _dir = os.path.join(FLAGS.log_dir, root_dir)
100    root_log_dir = os.path.join(project_dir, _dir, day_str, time_str, run_str)
101    check_else_make_dir(root_log_dir)
102    run_num = get_run_num(root_log_dir)
103    log_dir = os.path.abspath(os.path.join(root_log_dir,
104                                         f'run_{run_num}'))
105    if log_file is not None:
106        write(f'Output saved to: \n\t{log_dir}', log_file, 'a')
107        write(80*'-', log_file, 'a')
108
109    return log_dir
110
111
112 def _list_and_join(d):
113     """For each dir `dd` in `d`, return a list of paths ['d/dd1', ...]"""
114     contents = [os.path.join(d, i) for i in os.listdir(d)]
115     paths = [i for i in contents if os.path.isdir(i)]
116
117     return paths

```

```

118
119
120 def list_and_join(d):
121     """Deal with the case of `d` containing multiple directories."""
122     if isinstance(d, (list, np.ndarray)):
123         paths = []
124         for dd in d:
125             _path = _list_and_join(dd)[0]
126             paths.append(_path)
127     else:
128         paths = _list_and_join(d)
129
130     return paths
131
132
133 def get_eps_from_run_history_txt_file(txt_file):
134     """Parse `run_history.txt` file and return `eps` (step size)."""
135     with open(txt_file, 'r') as f:
136         data_line = [f.readline() for _ in range(10)][-1]
137         eps = float([i for i in data_line.split(' ') if i != ''][3])
138
139     return eps
140
141
142 def check_else_make_dir(d):
143     """If directory `d` doesn't exist, it is created."""
144     if not os.path.isdir(d):
145         log(f"Creating directory: {d}")
146         os.makedirs(d, exist_ok=True)
147
148
149 def make_dirs(dirs):
150     """Make directories if and only if hvd.rank == 0."""
151     _ = [check_else_make_dir(d) for d in dirs]
152
153
154 def save_data(data, out_file, name=None):
155     """Save data to out_file using either pickle.dump or np.save."""
156     if os.path.isfile(out_file):
157         log(f"WARNING: File {out_file} already exists...")
158         tmp = out_file.split('.')
159         out_file = tmp[0] + '_1' + f'.{tmp[1]}'
160
161     log(f"Saving {name} to {out_file}...")
162     if out_file.endswith('.pkl'):
163         with open(out_file, 'wb') as f:
164             pickle.dump(data, f)
165
166     elif out_file.endswith('.npy'):
167         np.save(out_file, np.array(data))
168
169     else:
170         log("Extension not recognized! out_file must end in .pkl or .npy")
171
172
173 def save_params_to_pkl_file(params, out_dir):
174     """Save `params` dictionary to `parameters.pkl` in `out_dir`."""
175     check_else_make_dir(out_dir)
176     params_file = os.path.join(out_dir, 'parameters.pkl')
177     # print(f"Saving params to: {params_file}.")

```

```

178     log(f"Saving params to: {params_file}.")
179     with open(params_file, 'wb') as f:
180         pickle.dump(params, f)
181
182
183     def get_run_num(log_dir):
184         """Get integer value for next run directory."""
185         check_else_make_dir(log_dir)
186         contents = os.listdir(log_dir)
187         if contents in ([], ['.DS_Store']):
188             return 1
189         try:
190             run_dirs = [i for i in os.listdir(log_dir) if 'run' in i]
191             run_nums = [int(i.split('_')[-1]) for i in run_dirs]
192             run_num = sorted(run_nums)[-1] + 1
193         except (ValueError, IndexError):
194             log(f"No previous runs found in {log_dir}, setting run_num=1.")
195             run_num = 1
196
197         return run_num
198
199
200     def _get_run_num(log_dir):
201         check_else_make_dir(log_dir)
202
203         contents = os.listdir(log_dir)
204         if contents in ([], ['.DS_Store']):
205             return 1
206
207         run_nums = []
208         for item in contents:
209             try:
210                 run_nums.append(int(item.split('_')[-1]))
211             except ValueError:
212                 continue
213         if run_nums == []:
214             return 1
215
216     return sorted(run_nums)[-1] + 1

```

A.20 l2hmc-qcd/utils/tf_logging.py

```
1  from __future__ import absolute_import
2  from __future__ import division
3  from __future__ import print_function
4
5  import os
6  import re
7
8  import tensorflow as tf
9
10 TOWER_NAME = 'tower'
11
12 def get_run_num(log_dir):
13     if not os.path.isdir(log_dir):
14         os.makedirs(log_dir)
15     contents = os.listdir(log_dir)
16     if contents == []:
17         return 1
18     else:
19         run_nums = []
20         for item in contents:
21             try:
22                 run_nums.append(int(item.split('_')[-1]))
23                 # run_nums.append(int(''.join(x for x in item if x.isdigit())))
24             except ValueError:
25                 continue
26         return sorted(run_nums)[-1] + 1
27     # if contents == ['.DS_Store']:
28     #     return 1
29     # else:
30     #     for item in contents:
31     #         if os.path.isdir(log_dir + item):
32     #             run_dirs.append(item)
33     #     run_nums = [int(str(i)[3:]) for i in run_dirs]
34     #     prev_run_num = max(run_nums)
35     #     return prev_run_num + 1
36
37 def make_run_dir(log_dir):
38     if log_dir.endswith('/'):
39         _dir = log_dir
40     else:
41         _dir = log_dir + '/'
42     run_num = get_run_num(_dir)
43     run_dir = _dir + f'run_{run_num}/'
44     if os.path.isdir(run_dir):
45         raise f'Directory: {run_dir} already exists, exiting!'
46     else:
47         print(f'Creating directory for new run: {run_dir}')
48         os.makedirs(run_dir)
49     return run_dir
50
51
52 def check_log_dir(log_dir):
53     if not os.path.isdir(log_dir):
54         raise ValueError(f'Unable to locate {log_dir}, exiting.')
55     else:
56         if not log_dir.endswith('/'):
57             log_dir += '/'
```

```

58     info_dir = log_dir + 'run_info/'
59     figs_dir = log_dir + 'figures/'
60     if not os.path.isdir(info_dir):
61         os.makedirs(info_dir)
62     if not os.path.isdir(figs_dir):
63         os.makedirs(figs_dir)
64     return log_dir, info_dir, figs_dir
65
66
67 def create_log_dir():
68     """Create directory for storing information about experiment."""
69     # root_log_dir = '../../../../../log_mog_tf/'
70     # root_log_dir = os.path.join(ROOT_DIR, log_mog_tf)
71     root_log_dir = os.path.join(os.path.split(ROOT_DIR)[0], 'log_mog_tf')
72     log_dir = make_run_dir(root_log_dir)
73     info_dir = log_dir + 'run_info/'
74     figs_dir = log_dir + 'figures/'
75     if not os.path.isdir(info_dir):
76         os.makedirs(info_dir)
77     if not os.path.isdir(figs_dir):
78         os.makedirs(figs_dir)
79     return log_dir, info_dir, figs_dir
80
81
82 def variable_summaries(var, name):
83     """Attach a lot of summaries to a Tensor (for TensorBoard visualization)"""
84     # with tf.name_scope('summaries'):
85     with tf.name_scope(name):
86         mean = tf.reduce_mean(var)
87         tf.summary.scalar('mean', mean)
88         with tf.name_scope('stddev'):
89             stddev = tf.sqrt(tf.reduce_mean(tf.square(var - mean)))
90             tf.summary.scalar('stddev', stddev)
91             tf.summary.scalar('max', tf.reduce_max(var))
92             tf.summary.scalar('min', tf.reduce_min(var))
93             tf.summary.histogram('histogram', var)
94
95
96 def add_loss_summaries(total_loss):
97     """Add summaries for losses in GaugeModel.
98
99     Generates a moving average for all losses and associated summaries for
100    visualizing the performance of the network.
101
102    Args:
103        total_loss: Total loss from model._calc_loss()
104
105    Returns:
106        loss_averages_op: Op for generating moving averages of losses.
107    """
108
109    # Compute the moving average of all individual losses and the total
110    # loss.
111    loss_averages = tf.train.ExponentialMovingAverage(0.9, name='avg')
112    losses = tf.get_collection('losses')
113    loss_averages_op = loss_averages.apply(losses + [total_loss])
114
115    # Attach a scalar summary to all individual losses and the total loss;
116    # do the same for the averaged version of the losses.
117    for l in losses + [total_loss]:
118        # Name each loss as '(raw)' and name the moving average version of

```

```

118     # the loss as the original loss name.
119     tf.summary.scalar(l.op.name + ' (raw)', l)
120     tf.summary.scalar(l.op.name, loss_averages.average(l))
121
122     return loss_averages_op
123
124 def activation_summary(x):
125     """Helper to create summaries for activations.
126
127     Creates a summary that provides a histogram of activations.
128     Creates a summary that measures the sparsity of activations.
129
130     Args:
131         x: Tensor
132     Returns:
133         None
134     """
135
136     # Remove 'tower_[0-9]/' from the name in case this is a multi-GPU training
137     # session. This helps the clarity of presentation in tensorboard.
138     # tensor_name = re.sub('%s_[0-9]*/' % TOWER_NAME, '', x.op.name)
139     try:
140         tensor_name = x.op.name
141     except AttributeError:
142         tensor_name = x.name
143     tf.summary.histogram(tensor_name + '/activations', x)
144     tf.summary.scalar(tensor_name + '/sparsity', tf.nn.zero_fraction(x))

```

A.21 l2hmc-qcd/utils/data_loader.py

```
1 """
2 data_loader.py
3
4 Implements DataLoader class responsible for loading run_data.
5
6 Author: Sam Foreman (github: @saforem2)
7 Date: 05/03/2019
8 """
9 import os
10 import pickle
11
12 import numpy as np
13
14
15 def load_params_from_dir(d):
16     params_file = os.path.join(d, 'params.pkl')
17     with open(params_file, 'rb') as f:
18         params = pickle.load(f)
19
20     return params
21
22
23 def get_run_dirs(root_dir):
24     run_dirs = []
25     root_dir = os.path.abspath(root_dir)
26     for dirpath, dirnames, filenames in os.walk(root_dir):
27         for dirname in dirnames:
28             keys = ['steps_', '_beta_', '_eps_']
29             conditions = [key in dirname for key in keys]
30             if np.alltrue(conditions):
31                 run_dirs.append(os.path.join(dirpath, dirname))
32
33     return run_dirs
34
35 # def make_fig_dirs(run_dirs):
36 #     for dirpath, dirnames, filenames in os.walk(root_dir):
37 #         for dirname in dirnames:
38 #             keys = ['steps_', '_beta_', '_eps_']
39 #             conditions = [key in dirname for key in keys]
40 #             if np.alltrue(conditions):
41 #                 run_dirs.append(os.path.join(dirpath, dirname))
42 #     return run_dirs
43
44
45 class DataLoader:
46     def __init__(self, run_dir=None):
47         self.run_dir = None
48
49     def load_pkl_file(self, pkl_file):
50         with open(pkl_file, 'rb') as f:
51             contents = pickle.load(f)
52
53     return contents
54
55     def load_npz_file(self, npz_file):
56         arr = np.load(npz_file)
57
```

```

58     return arr.f.arr_0
59
60 def _load_observable(self, observable_str, run_dir=None):
61     if run_dir is None:
62         run_dir = self.run_dir
63
64     if not observable_str.endswith('.pkl'):
65         observable_str += '.pkl'
66
67     obs_dir = os.path.join(run_dir, 'observables')
68     obs_file = os.path.join(obs_dir, observable_str)
69
70     return self.load_pkl_file(obs_file)
71
72 def load_samples(self, run_dir):
73     samples_file = os.path.join(run_dir, 'samples_out.npz')
74     samples = self.load_npz_file(samples_file)
75
76     return samples
77
78 def load_leapfrogs(self, run_dir):
79     lf_f_file = os.path.join(run_dir, 'lf_forward.npz')
80     lf_b_file = os.path.join(run_dir, 'lf_backward.npz')
81     lf_f = self.load_npz_file(lf_f_file)
82     lf_b = self.load_npz_file(lf_b_file)
83
84     return (lf_f, lf_b)
85
86 def load_logdets(self, run_dir):
87     logdets_f_file = os.path.join(run_dir, 'logdets_forward.npz')
88     logdets_b_file = os.path.join(run_dir, 'logdets_backward.npz')
89
90     logdets_f = self.load_npz_file(logdets_f_file)
91     logdets_b = self.load_npz_file(logdets_b_file)
92
93     return (logdets_f, logdets_b)
94
95 def load_sumlogdets(self, run_dir):
96     sumlogdet_f_file = os.path.join(run_dir, 'sumlogdet_forward.npz')
97     sumlogdet_b_file = os.path.join(run_dir, 'sumlogdet_backward.npz')
98
99     sumlogdet_f = self.load_npz_file(sumlogdet_f_file)
100    sumlogdet_b = self.load_npz_file(sumlogdet_b_file)
101
102    return (sumlogdet_f, sumlogdet_b)
103
104 def load_plaqs(self, run_dir):
105     obs_dir = os.path.join(run_dir, 'observables')
106     plaqs_file = os.path.join(obs_dir, 'plaqs.pkl')
107
108     return self.load_pkl_file(plaqs_file)
109
110 def load_autocorrs(self, run_dir):
111     obs_dir = os.path.join(run_dir, 'observables')
112     autocorrs_file = os.path.join(obs_dir, 'charges_autocorrs.pkl')
113
114     return self.load_pkl_file(autocorrs_file)
115
116 def load_params(self, run_dir=None):
117     if run_dir is None:

```

```

118         run_dir = self.run_dir
119         params_file = os.path.join(run_dir, 'parameters.pkl')
120
121     return self.load_pkl_file(params_file)
122
123 def load_run_data(self, run_dir=None):
124     if run_dir is None:
125         run_dir = self.run_dir
126
127     run_data_file = os.path.join(run_dir, 'run_data.pkl')
128
129     return self.load_pkl_file(run_data_file)
130
131 def load_run_stats(self, run_dir=None):
132     if run_dir is None:
133         run_dir = self.run_dir
134
135     run_stats_file = os.path.join(run_dir, 'run_stats.pkl')
136
137     return self.load_pkl_file(run_stats_file)
138
139 def load_observables(self, run_dir=None):
140     if run_dir is None:
141         run_dir = self.run_dir
142
143     obs_dir = os.path.join(run_dir, 'observables')
144     files = os.listdir(obs_dir)
145
146     obs_names = [i.rstrip('.pkl') for i in files]
147     obs_files = [os.path.join(obs_dir, i) for i in files]
148
149     observables = {
150         n: self.load_pkl_file(f) for (n, f) in zip(obs_names, obs_files)
151     }
152
153     return observables

```

B | Appendix: Systematic Debugging Results of the Average Plaquette

In each of the plots below, the L2HMC sampler was trained for 25,000 steps with a simulated annealing schedule beginning at $\beta = 2.0$ and ending at $\beta = 5.0$. Additionally, the training was distributed across 16 nodes on COOLEY using horovod. The results are averaged over all N_{samples} samples in the mini-batch.

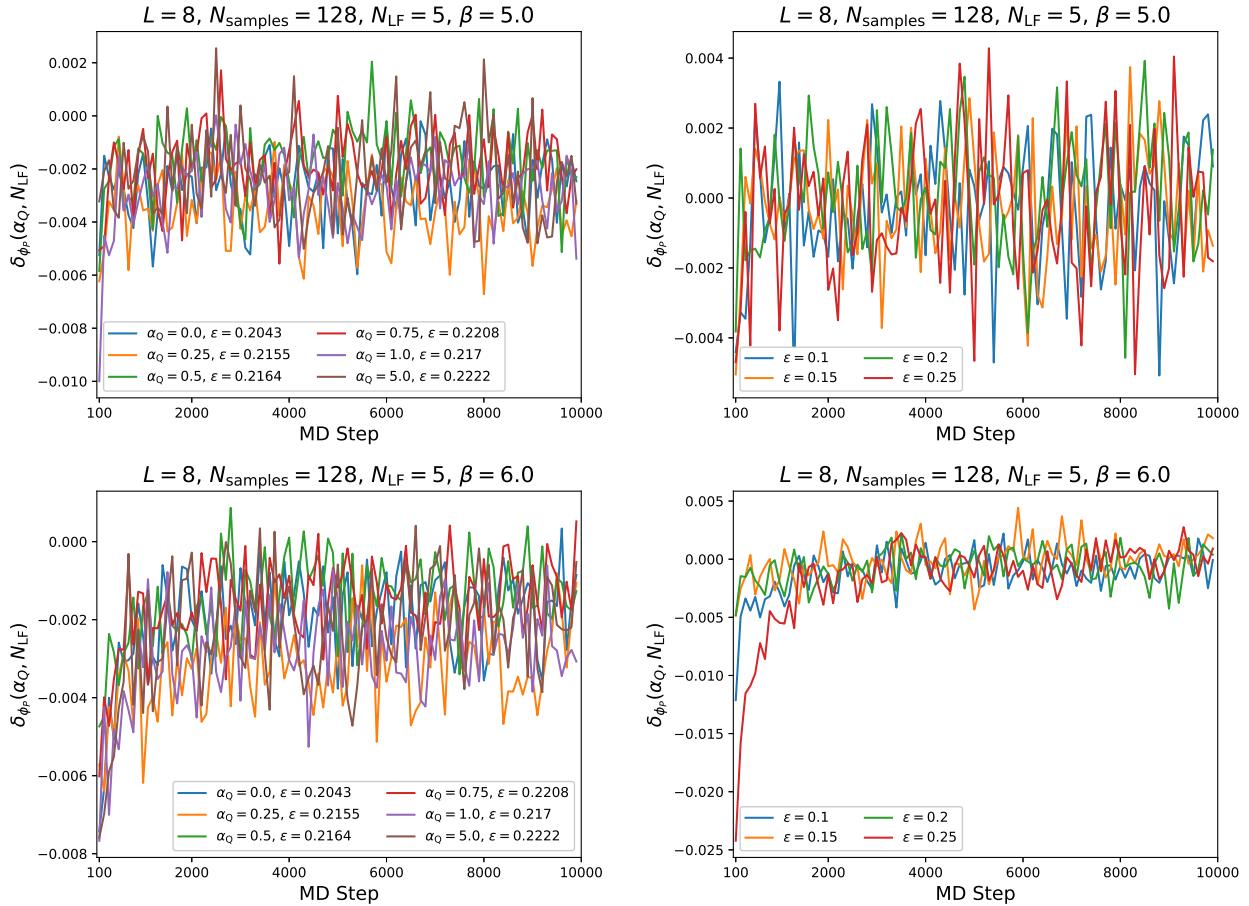


Figure B.1: δ_{ϕ_P} vs MD step with $N_{\text{LF}} = 5$ for $\beta = 5.0$ (top row) and $\beta = 6.0$ (bottom row). The results from the trained L2HMC (generic HMC) sampler are shown in the left (right) column. As can be seen, the difference δ_{ϕ_P} remains roughly consistent for all values of α_Q .

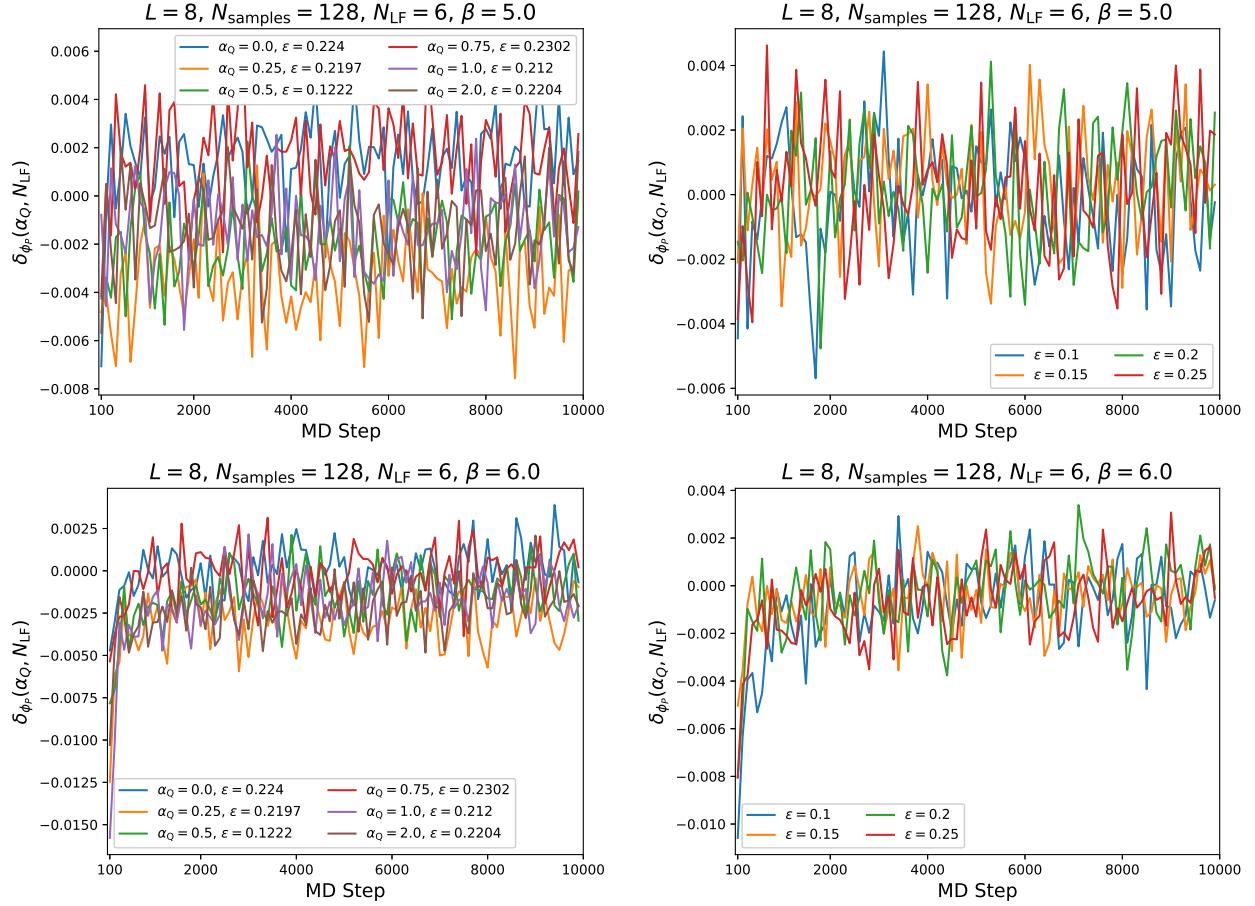


Figure B.2: δ_{ϕ_P} vs MD step with $N_{LF} = 5$ for $\beta = 5.0$ (top row) and $\beta = 6.0$ (bottom row). The results from the trained L2HMC (generic HMC) sampler are shown in the left (right) column. As can be seen, the difference δ_{ϕ_P} remains roughly consistent for all values of α_Q .

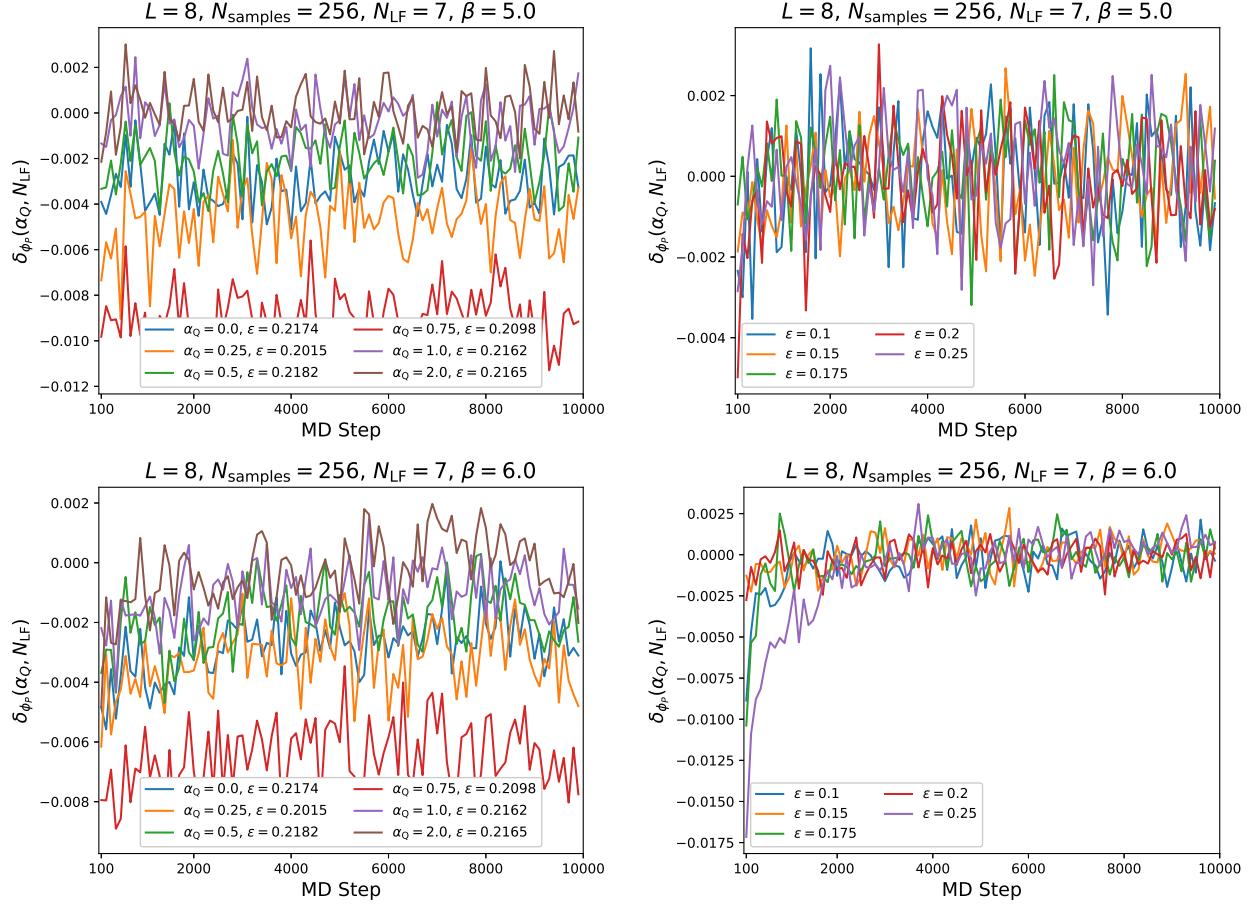


Figure B.3: δ_{ϕ_P} vs MD step with $N_{LF} = 7$. As can be seen, the difference δ_{ϕ_P} is noticeably larger for $\alpha_Q = 0.75$, but remains roughly consistent for all other values of α_Q .

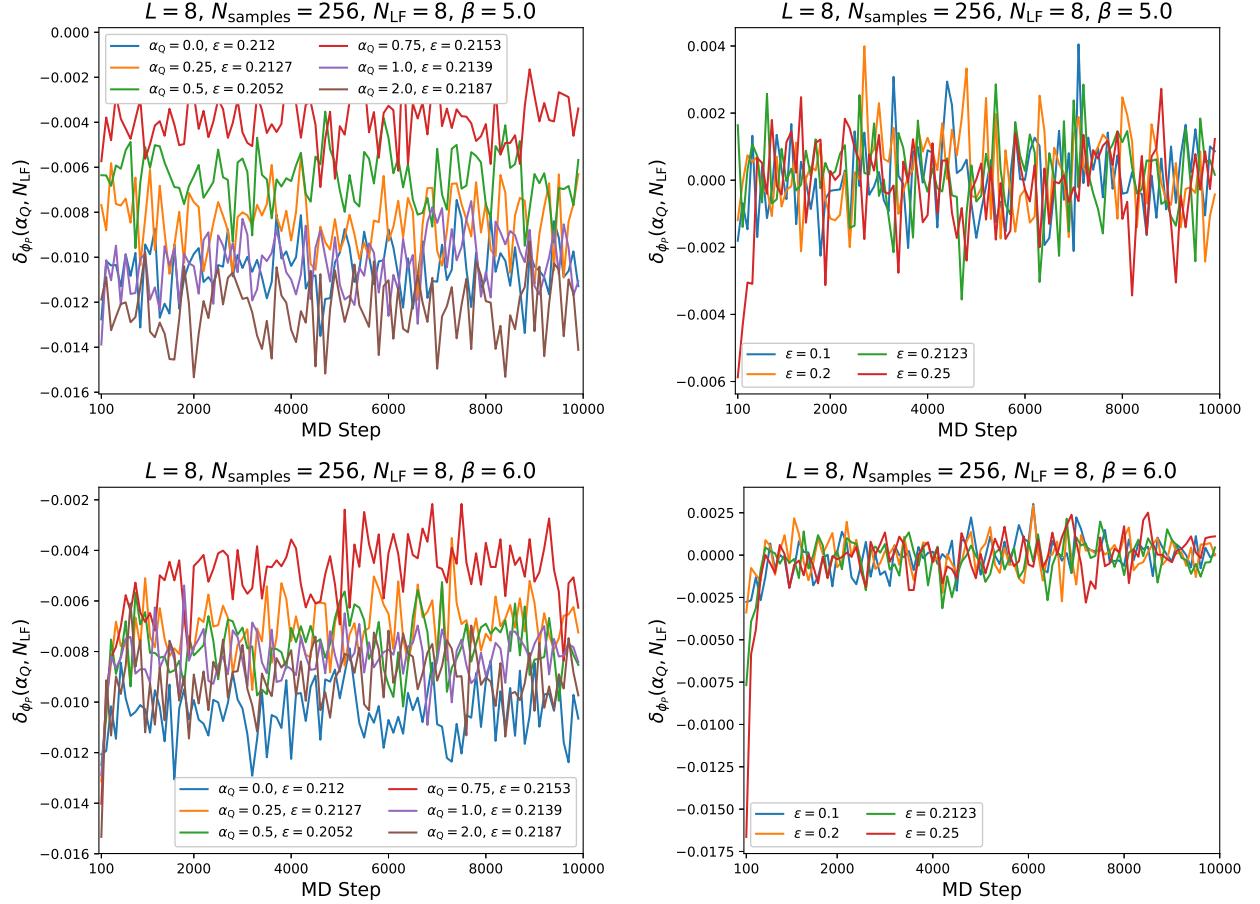


Figure B.4: δ_{ϕ_P} vs. MD step with $N_{\text{LF}} = 8$. As can be seen, the difference δ_{ϕ_P} remains roughly consistent for all values of α_Q .

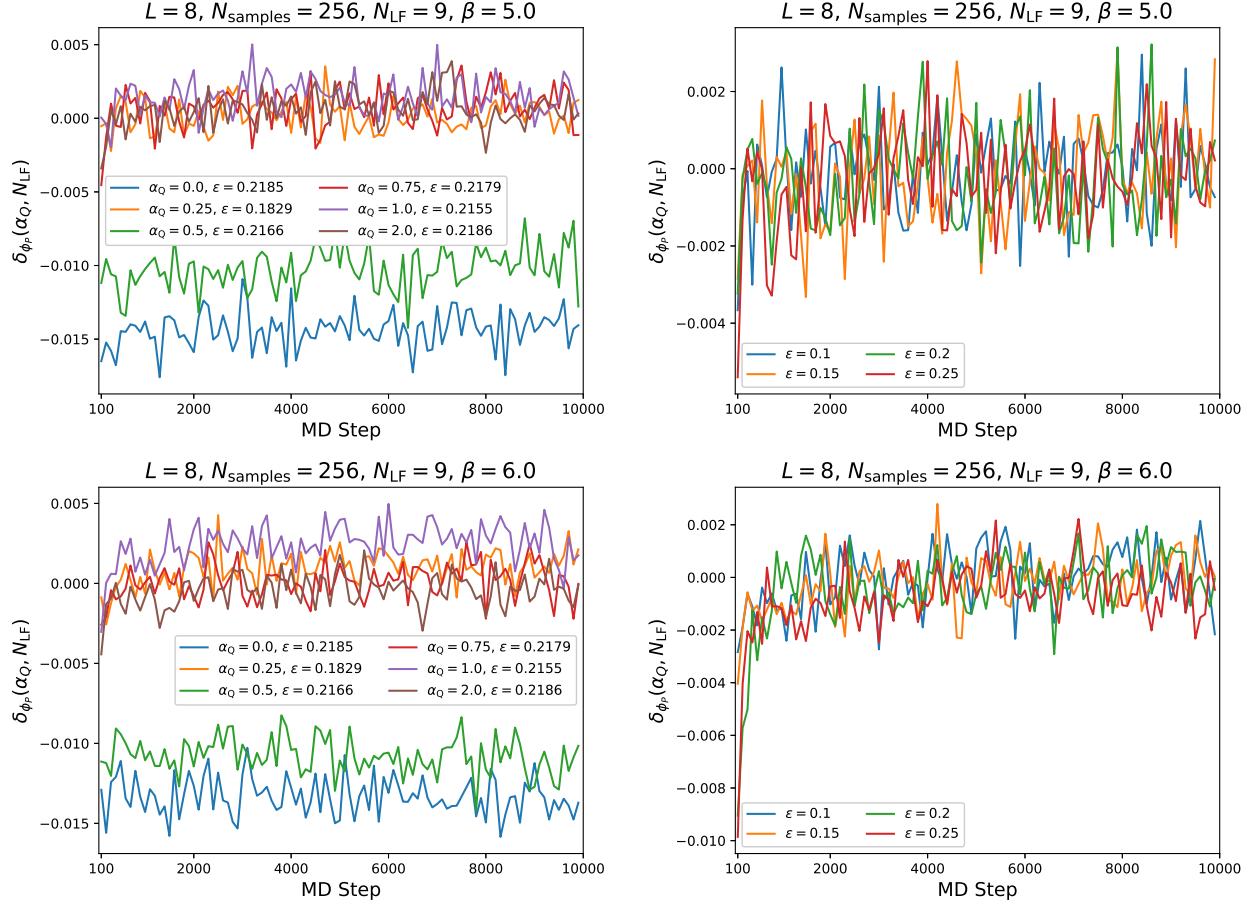


Figure B.5: δ_{ϕ_P} vs MD step with $N_{\text{LF}} = 9$. As can be seen, the difference δ_{ϕ_P} is largest for $\alpha_Q = 0.0$ and $\alpha_Q = 0.5$.

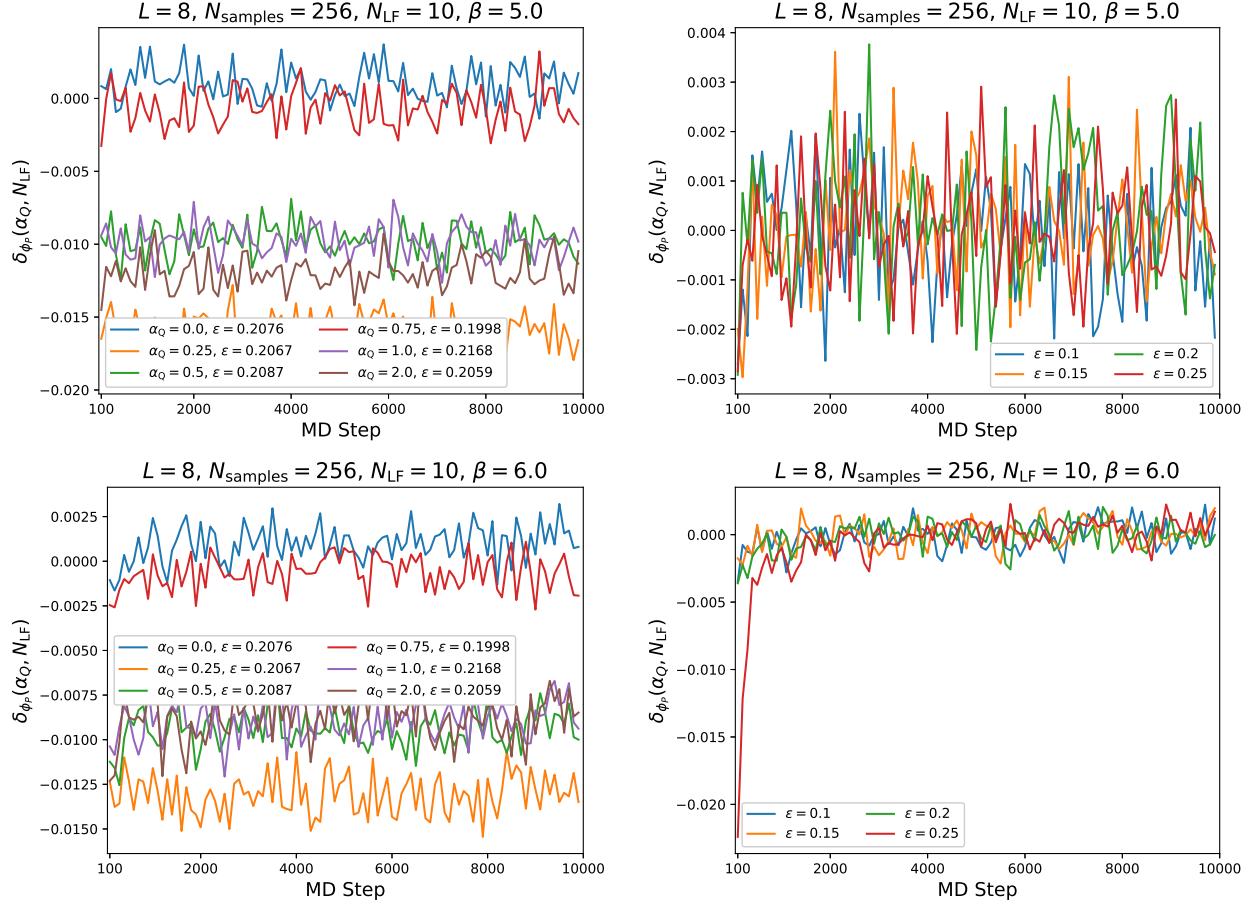


Figure B.6: δ_{ϕ_P} vs MD step with $N_{LF} = 10$. As can be seen, the difference δ_{ϕ_P} is smallest for $\alpha_Q = 0., 0.75$ and largest for $\alpha_Q = 0.25$, while $\alpha_Q = 0.5, 2.0$, takes on intermediate values.

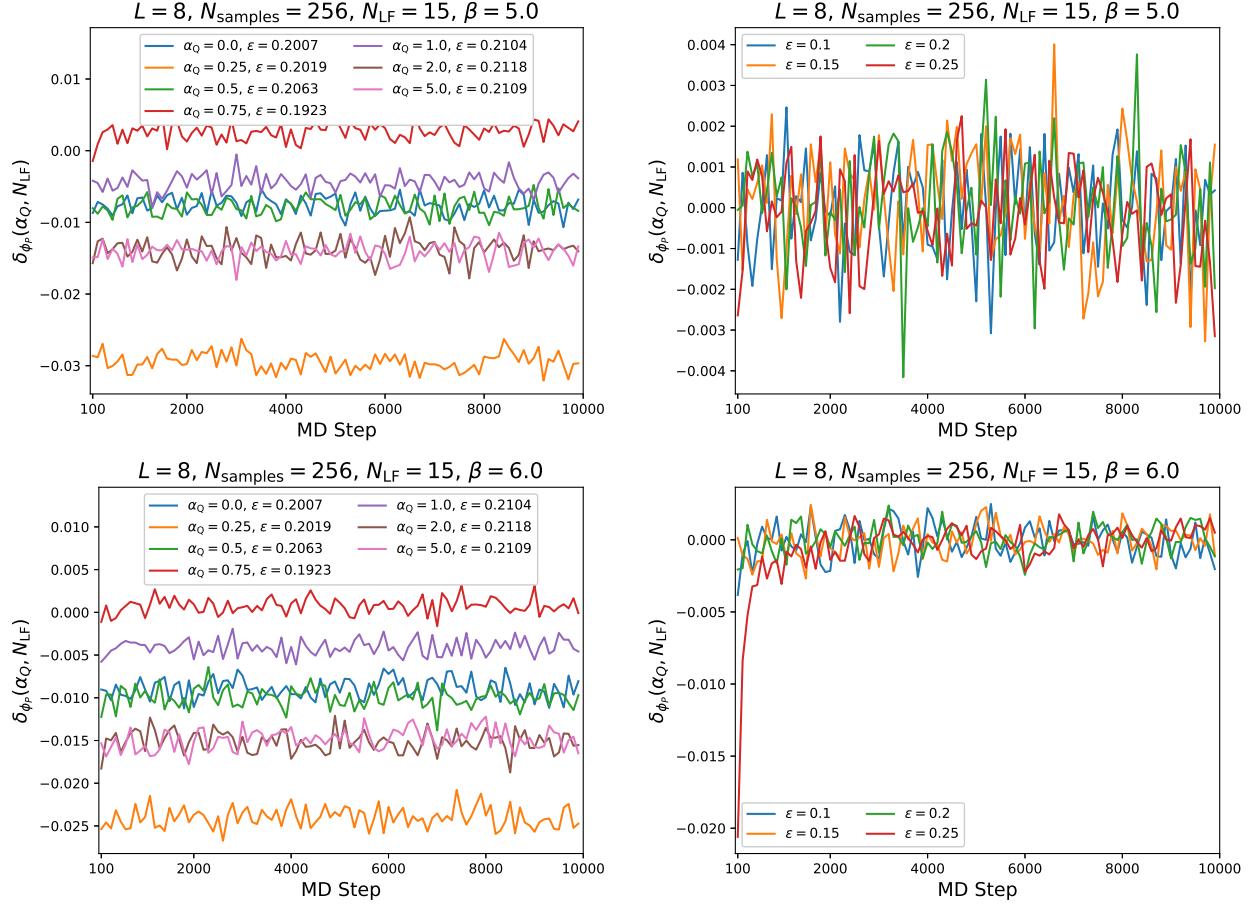


Figure B.7: δ_{ϕ_P} vs MD step with $N_{LF} = 15$. As can be seen, the difference δ_{ϕ_P} varies for different values of α_Q .

References

- [1] D. Levy, M. D. Hoffman, and J. Sohl-Dickstein, [arXiv:1711.09268 \[cs, stat\]](#), [arXiv: 1711.09268 \(2017\)](#).
- [2] S. Shalev-Shwartz and S. Ben-David, *Understanding machine learning: from theory to algorithms* (Cambridge university press, 2014).
- [3] D. P. Kingma and M. Welling, [arXiv:1312.6114 \[cs, stat\]](#), [arXiv: 1312.6114 \(2013\)](#).
- [4] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, [arXiv:1406.2661 \[cs, stat\]](#), [arXiv: 1406.2661 \(2014\)](#).
- [5] A. Ng, “Cs229 lecture notes - supervised learning”, 2012.
- [6] Y. LeCun, Y. Bengio, et al., The handbook of brain theory and neural networks **3361**, 1995 (1995).
- [7] D. Scherer, A. Müller, and S. Behnke, in International conference on artificial neural networks (Springer, 2010), pp. 92–101.
- [8] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, [arXiv:1412.6806 \[cs\]](#), [arXiv: 1412.6806 \(2014\)](#).
- [9] C. M. Bishop, *Pattern recognition and machine learning*, Information science and statistics (Springer, 2006).
- [10] S.-H. Li and L. Wang, *Physical Review Letters* **121**, 260601, 260601 (2018).
- [11] C. Bény, [arXiv:1301.3124 \[quant-ph\]](#), [arXiv: 1301.3124 \(2013\)](#).
- [12] P. Mehta and D. J. Schwab, [arXiv:1410.3831 \[cond-mat, stat\]](#), [arXiv: 1410.3831 \(2014\)](#).
- [13] S. Foreman, J. Giedt, Y. Meurice, and J. Unmuth-Yockey, *EPJ Web of Conferences* **175**, edited by M. Della Morte, P. Fritzsch, E. Gámiz Sánchez, and C. Pena Ruano, 11025 (2018).
- [14] S. Bradde and W. Bialek, *Journal of Statistical Physics* **167**, 462 (2017).
- [15] J. Carrasquilla and R. G. Melko, *Nature Physics* **13**, 431 (2017).
- [16] L. Wang, *Phys. Rev. B* **94**, 195105 (2016).
- [17] W. Hu, R. R. P. Singh, and R. T. Scalettar, *Phys. Rev. E* **95**, 062122, 062122 (2017).
- [18] S. J. Wetzel, *Phys. Rev. E* **96**, 022140, 022140 (2017).
- [19] M. Levin and C. P. Nave, *Phys. Rev. Lett.* **99**, 120601 (2007).
- [20] Z.-C. Gu, M. Levin, B. Swingle, and X.-G. Wen, *Phys. Rev. B* **79**, 085118 (2009).
- [21] Z. Y. Xie, J. Chen, M. P. Qin, J. W. Zhu, L. P. Yang, and T. Xiang, *Phys. Rev. B* **86**, 045139 (2012).

- [22] Y. Meurice, *Phys. Rev.* **B87**, 064422 (2013).
- [23] Y. Liu, Y. Meurice, M. P. Qin, J. Unmuth-Yockey, T. Xiang, Z. Y. Xie, J. F. Yu, and H. Zou, *Phys. Rev.* **D88**, 056005 (2013).
- [24] A. Denbleyker, Y. Liu, Y. Meurice, M. P. Qin, T. Xiang, Z. Y. Xie, J. F. Yu, and H. Zou, *Phys. Rev.* **D89**, 016008 (2014).
- [25] J. F. Yu, Z. Y. Xie, Y. Meurice, Y. Liu, A. Denbleyker, H. Zou, M. P. Qin, and J. Chen, *Phys. Rev.* **E89**, 013308 (2014).
- [26] R. Savit, *Rev. Mod. Phys.* **52**, 453 (1980).
- [27] N. Prokof'ev and B. Svistunov, *Phys. Rev. Lett.* **87**, 160601 (2001).
- [28] T. A. Enßlin and M. Frommert, *Phys. Rev. D* **83**, 105014 (2011).
- [29] R. H. Swendsen and J.-S. Wang, *Phys. Rev. Lett.* **58**, 86 (1987).
- [30] B. Kaufman, *Phys. Rev.* **76**, 1232 (1949).
- [31] A. Krizhevsky and G. Hinton, Master's thesis, Department of Computer Science, University of Toronto (2009).
- [32] B. Gidas, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **11**, 164 (1989).
- [33] K. Tanaka, *Journal of Physics A: Mathematical and General* **35**, R81 (2002).
- [34] T. Horiguchi, Y. Honda, and M. Miya, *Physics Letters A* **227**, 319 (1997).
- [35] Z. Y. Xie, H. C. Jiang, Q. N. Chen, Z. Y. Weng, and T. Xiang, *Phys. Rev. Lett.* **103**, 160601 (2009).
- [36] G. E. Hinton and R. R. Salakhutdinov, *Science* **313**, 504 (2006).
- [37] P. E. Shanahan, D. Trewartha, and W. Detmold, *Phys. Rev.* **D97**, 094506 (2018).
- [38] S. Brooks, A. Gelman, G. Jones, and X.-L. Meng, *Handbook of markov chain monte carlo* (CRC press, 2011).
- [39] *Scribe_note_lecture17.pdf*, https://www.cs.cmu.edu/~epxing/Class/10708-14/scribe_notes/scribe_note_lecture17.pdf, (Accessed on 06/18/2019).
- [40] M. Hanada, arXiv:1808.08490 [cond-mat, physics:hep-lat, physics:hep-th], arXiv: 1808.08490 (2018).
- [41] R. M. Neal, arXiv:1206.1901 [physics, stat], arXiv: 1206.1901 (2012).
- [42] M. Betancourt, arXiv:1701.02434 [stat], arXiv: 1701.02434 (2017).
- [43] L. Dinh, J. Sohl-Dickstein, and S. Bengio, arXiv:1605.08803 [cs, stat], arXiv: 1605.08803 (2016).
- [44] C. Pasarica and A. Gelman, *Statistica Sinica* **20**, 343 (2010).
- [45] S. Ioffe and C. Szegedy, arXiv:1502.03167 [cs], arXiv: 1502.03167 (2015).