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.

Additional support was provided by DOE grant DE-SC0010113.

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 time[1].
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

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

[2] Technically, 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

LIST OF FIGURES

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 imperceivable 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 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.

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.

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]. 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 [3]: *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) [4], and *generative adversarial networks* (GANs) [5]).

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: $\{(x^{(i)}, y^{(i)})\}, i = 1, \ldots, n$, 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, $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$, each of which contains p pixels flattened into a vector, and their associated labels $y^{(1)}, y^{(2)}, \ldots, y^{(n)}$ 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 to 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

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

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

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

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

Here, α is a *hyperparameter* called the **learning rate**, and it is responsible for determining the "step size" the algorithm takes with each update. Its 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:

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

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

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

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

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

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

$$= (f_w(x) - y) x_j, \tag{2.6}$$

and so

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

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: $\mathbf{y} = [y^{(1)}, y^{(2)}, \ldots, y^{(m)}]^T$. In terms of these then, our cost function becomes

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

Again, computing the required gradient gives

$$\begin{aligned}
\nabla_w \mathcal{L}(w) &= \nabla_w \frac{1}{2}(Xw - \mathbf{y})^T(Xw - \mathbf{y}) && (2.9)\\
&= \frac{1}{2}\nabla_w \left(w^T X^T X w - w^T X^T \mathbf{y} - \mathbf{y}^T X w + \mathbf{y}^T \mathbf{y} \right) && (2.10)\\
&= \frac{1}{2}\nabla_w \ \mathrm{tr} \left(w^T X^T X w - w^T X^T \mathbf{y} - \mathbf{y}^T X w + \mathbf{y}^T \mathbf{y} \right) && (2.11)\\
&= \frac{1}{2}\nabla_w \left(\mathrm{tr}\, w^T X^T X w - 2\mathrm{tr}\, \mathbf{y}^T X w \right) && (2.12)\\
&= \frac{1}{2}\left(X^T X w + X^T X w - 2X^T \mathbf{y} \right) && (2.13)\\
&= X^T X w - X^T \mathbf{y} && (2.14)
\end{aligned}$$

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

$$X^T X w = X^T \mathbf{y}, \tag{2.15}$$

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

from which we get

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

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

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

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 $\mathrm{i.i.d}$. The probability density is given by

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

This implies that

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

and we say that $p(y^{(i)}|x^{(i)};\, x)$ 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**,

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

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

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

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

By the principal of **maximum likelihood** [6], 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** $\ell(w)$:

$$\begin{aligned}
\ell(w) &\equiv \log L(w) && (2.23)\\
&= \log \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\left(y^{(i)} - w^T x^{(i)}\right)^2}{2\sigma^2}\right) && (2.24)\\
&= \sum_{i=1}^{m} \log \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\left(y^{(i)} - w^T x^{(i)}\right)^2}{2\sigma^2}\right) && (2.25)\\
&= m \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{\sigma^2} \cdot \frac{1}{2} \sum_{i=1}^{m} \left(y^{(i)} - w^T x^{(i)}\right)^2. && (2.26)
\end{aligned}$$

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

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

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 [7]. First, we construct M linear combinations of the input variables $x_1, \ldots, x_D$ in the form

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

where $j = 1, \ldots, M$ 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

$$z_j = h(a_j) \tag{2.29}$$

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

$$\mathrm{ReLU}(x) \equiv \max(0, x). \tag{2.30}$$

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

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

where $k = 1, \ldots, K$, 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 $\sigma : \mathbb{R}^K \to \mathbb{R}^K$ given by

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

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

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

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

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

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

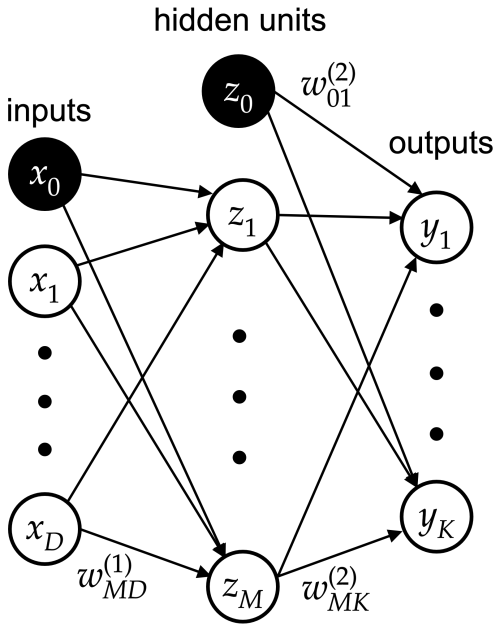


Figure 2.1: Network diagram for a two-layer fully-connected neural network. The input, hidden, and output variables are shown as nodes while the weight parameters are the links between them.

2.3.1 Parameter optimization & Backpropagation

Given a set of input vectors $\{\mathbf{x}_n\}$ for $n = 1, \ldots, N$, 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

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

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 [8–11], outlined below. For consistency, we adopt the following notations:

- $X = \{(\mathbf{x}_1, \mathbf{t}_1), \ldots, (\mathbf{x}_N, \mathbf{t}_N)\}$: 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.

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

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)

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

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

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

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

2.3.1.1 Derivatives of the Error Function

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

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^k} = \frac{\partial \mathcal{L}}{\partial a_j^k}\frac{\partial a_j^k}{\partial w_{ij}^k} \tag{2.40}$$

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

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

to give

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

$$= z_i^{k-1}. \tag{2.43}$$

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

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

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 value[3] 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

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

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

$$\delta_1^m = (h_z(a_1^m) - t)h_z'(a_1^m) = (y-t)h_z'(a_1^m) \tag{2.46}$$

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

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

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:

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

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

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

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

Recalling that

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

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

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

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*:

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

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 $1 \leq k < m$:

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

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 vol-

Algorithm 1: Backpropagation Algorithm

input :

- A data set $X = \{(\mathbf{x}_1, t_1), \ldots, (\mathbf{x}_N, t_N)\}$ 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:

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

ume of shape $\text{height} \times \text{width} \times \text{depth}$ 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 $\text{depth} = 3$, which we could represent as a three-dimensional volume of $\text{height} \times \text{width} \times [\text{red}, \text{green}, \text{blue}]$ pixels.

Additionally, in contrast to fully-connected networks, ConvNets combine three architectural ideas to ensure some degree of shift and distortion invariance [12]:

- *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 hyperparamter, 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 hyperparamter 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

$$\frac{W - K + 2P}{S} + 1 \tag{2.54}$$

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.

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

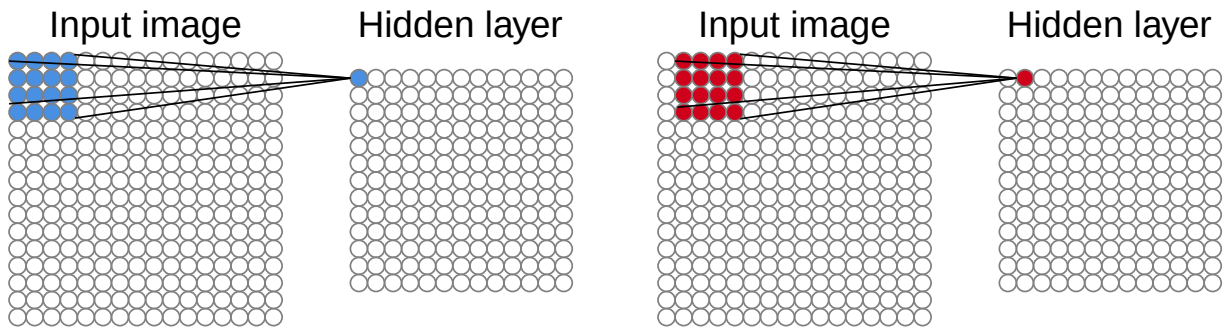


Figure 2.2: Illustration of the local receptive fields in the input image and their corresponding weights in the first hidden layer. Additionally, we can see how these local receptive fields are slid across the input image to construct additional units in the hidden layer.

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 [13].

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 [14]. 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 [15, 16]. 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.

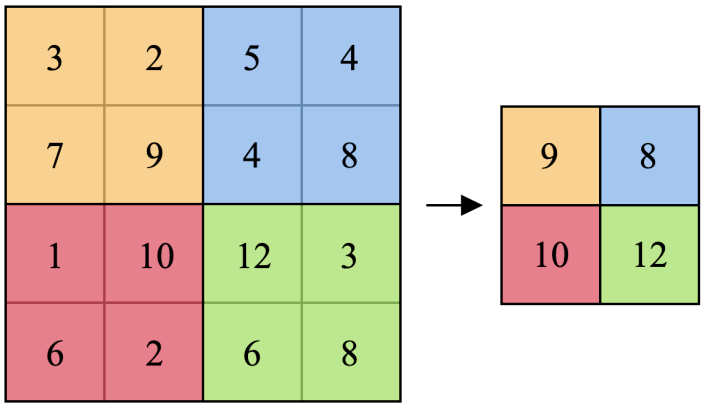


Figure 2.3: Illustration of a 2×2 max pooling layer.

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 [7]. 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 [17–19] 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 [18], 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 [20]. It can be called "cheap learning" because far fewer variational parameters are involved,

due to the constraints of locality. In [19] 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 [21] 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 [22–25]. 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 [26–32]. The starting point for this reformulation is the character expansion of the Boltzmann weights which is also used in the duality transformation [33]. 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 [34] 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 [20]. Criticality may sometimes refer to the choice of parameters used in data analysis [35].

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 $\lambda_{\max}$ is directly connected to the magnetic susceptibility which diverges at criticality [23]. In the loop representation (worms), we will show that $\lambda_{\max}$ 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 $\lambda_{\max}$.

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 [29]. 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 $1+1 \to x$ hereafter. In the case $1+1 \to 0$, 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 $1+1 \to 0$) 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 [33]: they are the boundary of the positive spin islands. This sug-

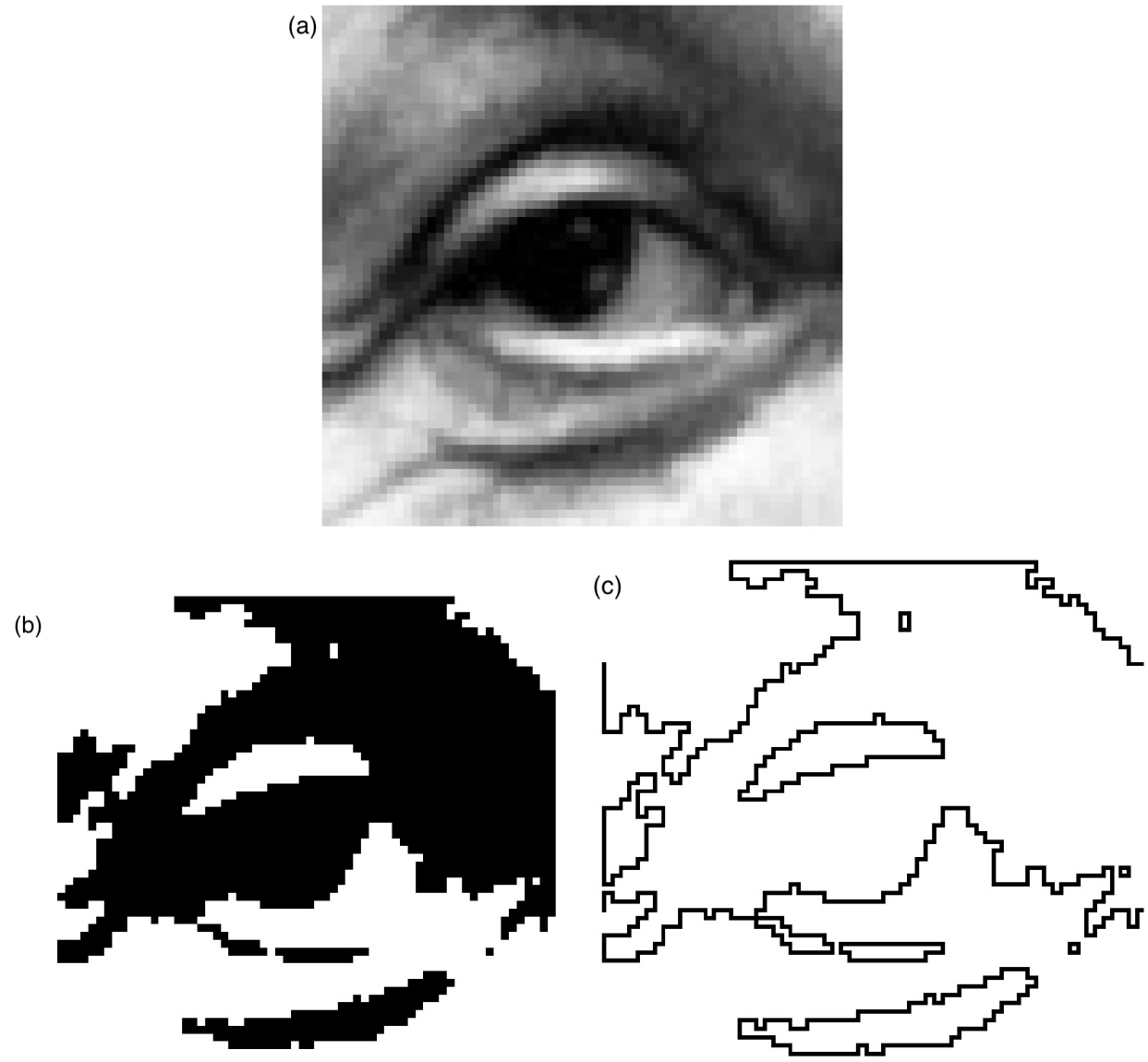


Figure 3.1: (a) Picture of an eye with 4096 pixels; (b) black and white version with a graycut at 0.72; (c) boundaries of the black domains.

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

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

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 [33]

$$\exp(\beta\sigma) = \cosh(\beta) + \sigma \sinh(\beta), \tag{3.2}$$

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 [33]. 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:

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

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,

$$\langle \Delta^2_{N_b} \rangle \equiv \langle (N_b - \langle N_b \rangle)^2 \rangle, \tag{3.4}$$

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

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

In the following we use interchangeably the "bond" terminology, for instance in N_b as in [34] 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 [34] 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

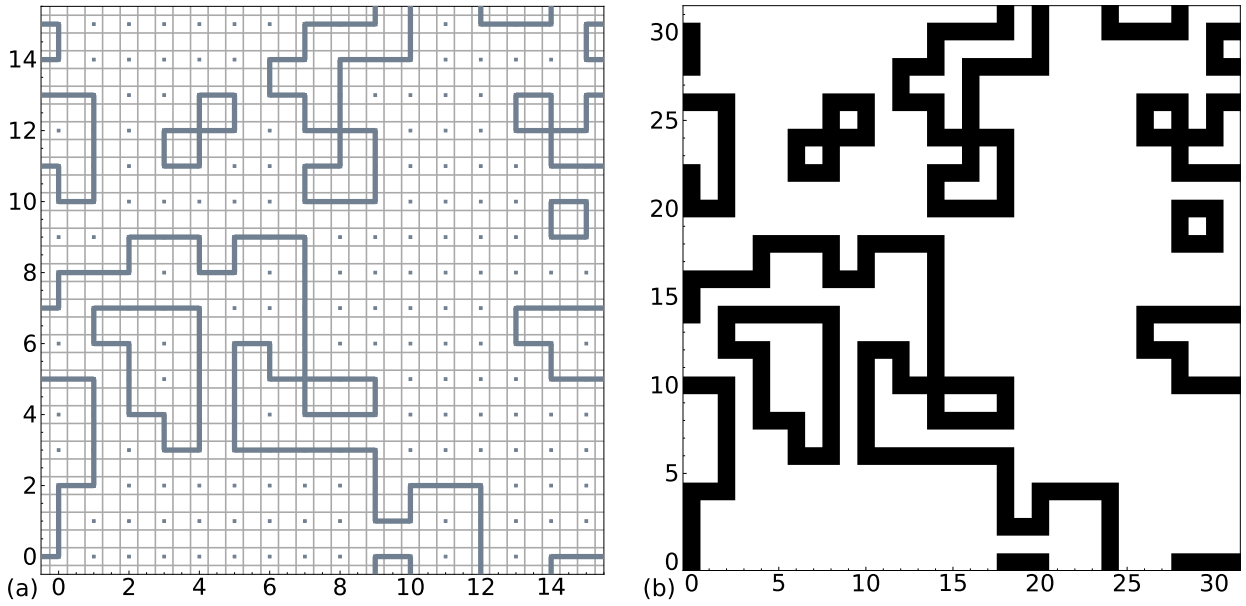


Figure 3.2: (a) Legal worm configuration on an $L \times L$ lattice with periodic boundary conditions and; (b) its equivalent representation as a $2L \times 2L$ black and white pixel image.

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

$$\sum_{j=bonds} v_j = N_b \tag{3.6}$$

3.3 PCA and criticality

Having now sets of images for a range of temperatures, we can apply PCA [7]. 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 $\{\mathbf{v}^n\}_{n=1}^N$:

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

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

$$Su_\alpha = \lambda_\alpha u_\alpha \tag{3.8}$$

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 [7]) by the first M principal components.

Illustrations of the PCA for the MNIST data can be found in Sec. 4 of Ref. [20], 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 $\lambda_{\max}$ and the logarithmic divergence of the specific heat, namely

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

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 [28–32].

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-

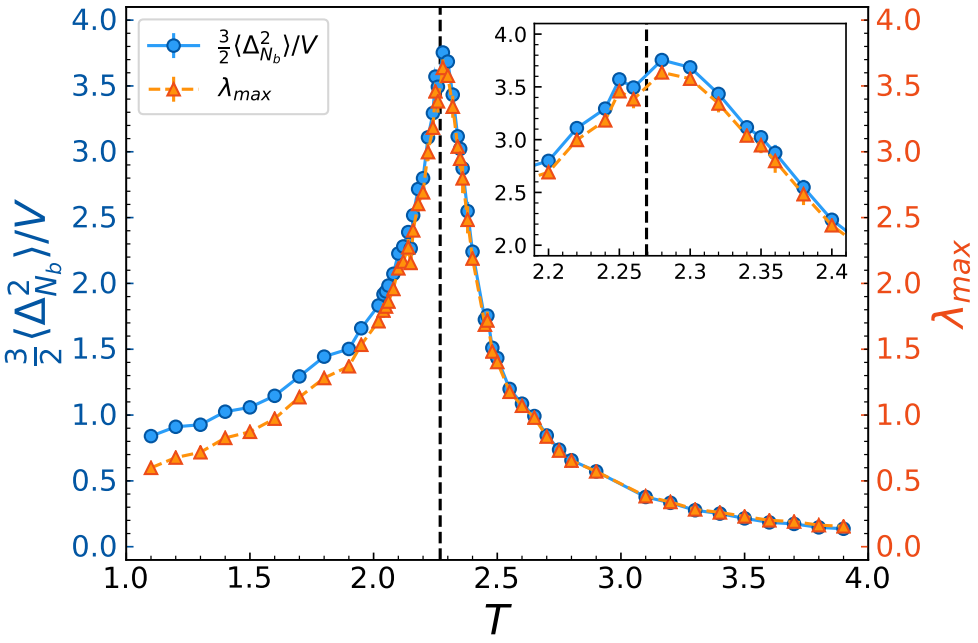


Figure 3.3: λ_{max} and $\frac{3}{2}\left\langle\Delta^2_{N_b}\right\rangle/V$ (per unit volume) vs. T, illustrating the relation between the eigenvalue corresponding to the first principal component and the logarithmic divergence of the specific heat. The inset shows a qualitative agreement near the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

$$T^{(i)}_{xx'yy'}(\beta) = [\tanh(\beta)]^{(n_x+n_{x'}+n_y+n_{y'})/2} \times \delta_{n_x+n_{x'}+n_y+n_{y'} \bmod 2,\, 0}. \tag{3.10}$$

Here the notation being used is that this tensor is located at the i^{th} site of the lattice, $n_{\hat{\mu}}$ 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,

$$Z = 2^V(\cosh(\beta))^{2V}\,\mathrm{Tr}\prod_i T^{(i)}_{xx'yy'} \tag{3.11}$$

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

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

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

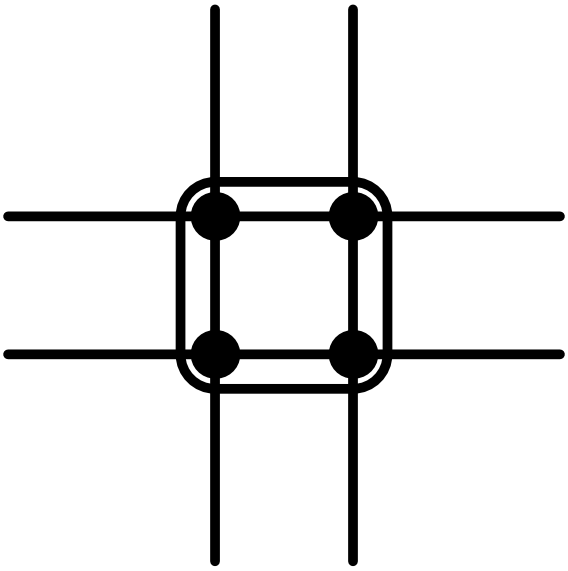


Figure 3.4: Illustration of the tensor blocking discussed in the text. Each dot is a tensor at a lattice site with four lines coming out, each representing a tensor index. Lines connecting dots represent tensor contractions.

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 [29] 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 [29], 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 [29]. To be completely specific, the exact T_c for the original model is $2/\ln(1+\sqrt{2}) = 2.269\ldots$ while for the two state projection with the second projection procedure of Ref. [29], it is $1/0.3948678 = 2.53249\ldots$ It is easy to relate the

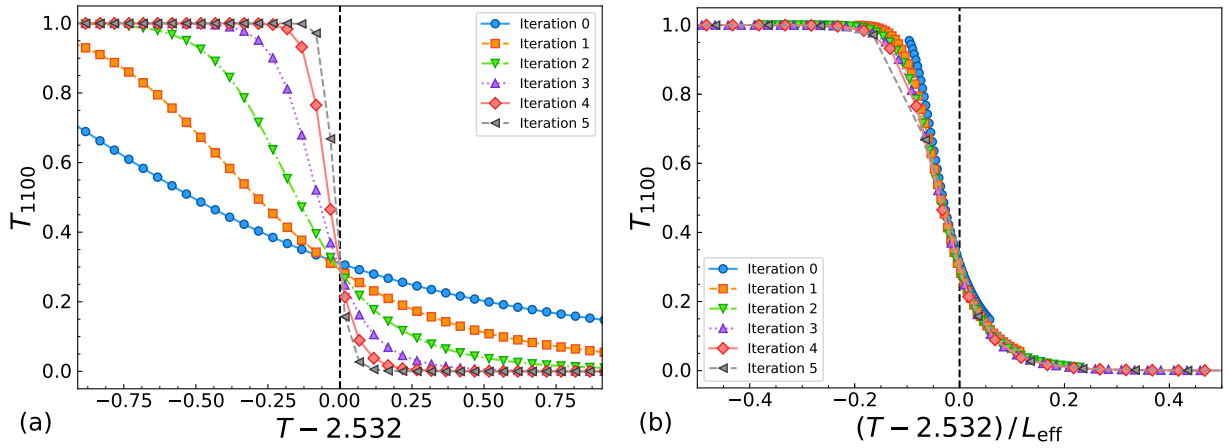


Figure 3.5: (a) T_{1100} vs. $T - T_c^{(2s)}$ for six successive iterations of the blocking transformation, beginning with an initial lattice $L = 64$; (b) T_{1100} vs. $(T - T_c^{(2s)})/L_{\text{eff}}$ illustrating the data collapse, where $T_c^{(2s)}$ is the critical temperature of the two state projection, 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 [29]. 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

$$\delta T_{1100} \equiv T_{1100} - T^*_{1100} \tag{3.12}$$

(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 λ=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 $L_{\text{eff}} = L/b^\ell$.

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

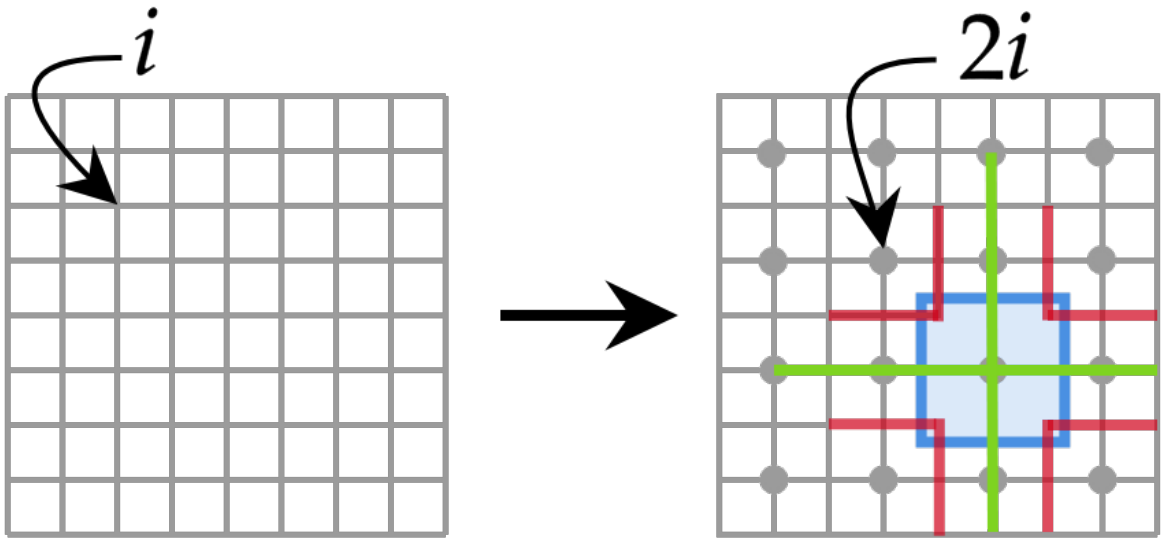


Figure 3.6: Illustration of the coarse-graining procedure in which the original lattice sites (i) are replaced by blocked sites ($2i$) (grey circles) with twice the original lattice spacing. In the coarse-grained lattice, an elementary block (blue) consists of four sites on the original lattice, eight external bonds (red), and four blocked external bonds (green).

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 "$1+1 \to 0$". This approach respects the selection rule (conservation modulo 2) and has the advantage of maintaining the closed-path restriction. In other words with the $1+1 \to 0$ 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

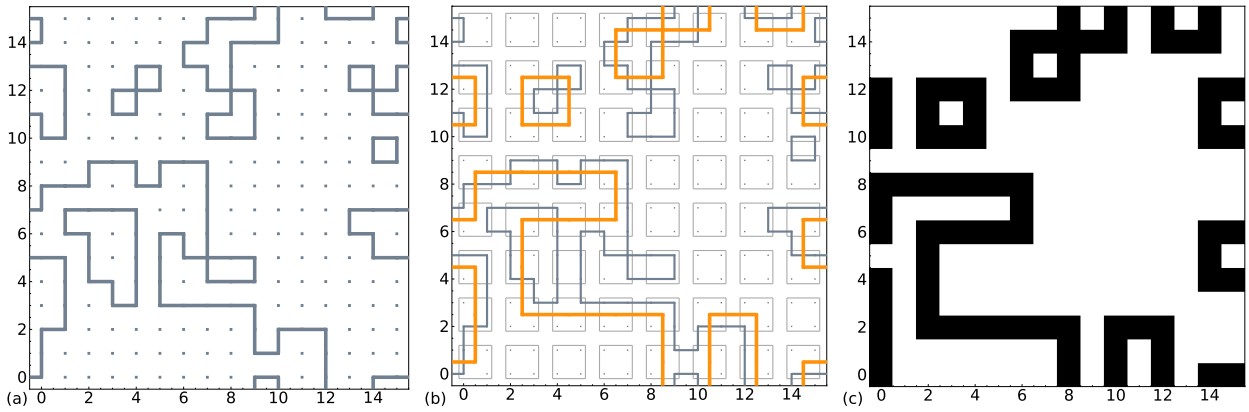


Figure 3.7: (a) Illustration of the $1+1 \to 0$ blocking procedure discussed in the text: original configuration; (b) introduction of the blocks and replacement of single or double bounds according to the $1+1 \to 0$ rule; (c) construction of the corresponding blocked image.

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 "$1+1 \to 1$", and "$1+1 \to 2$" 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 $1+1 \to 0$ rule starting with configurations on a 64×64 lattice. A first observation is that the $1+1 \to 0$ blocking preserves the location of the peak of the fluctuations $\langle \Delta^2_{N_b} \rangle$. 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

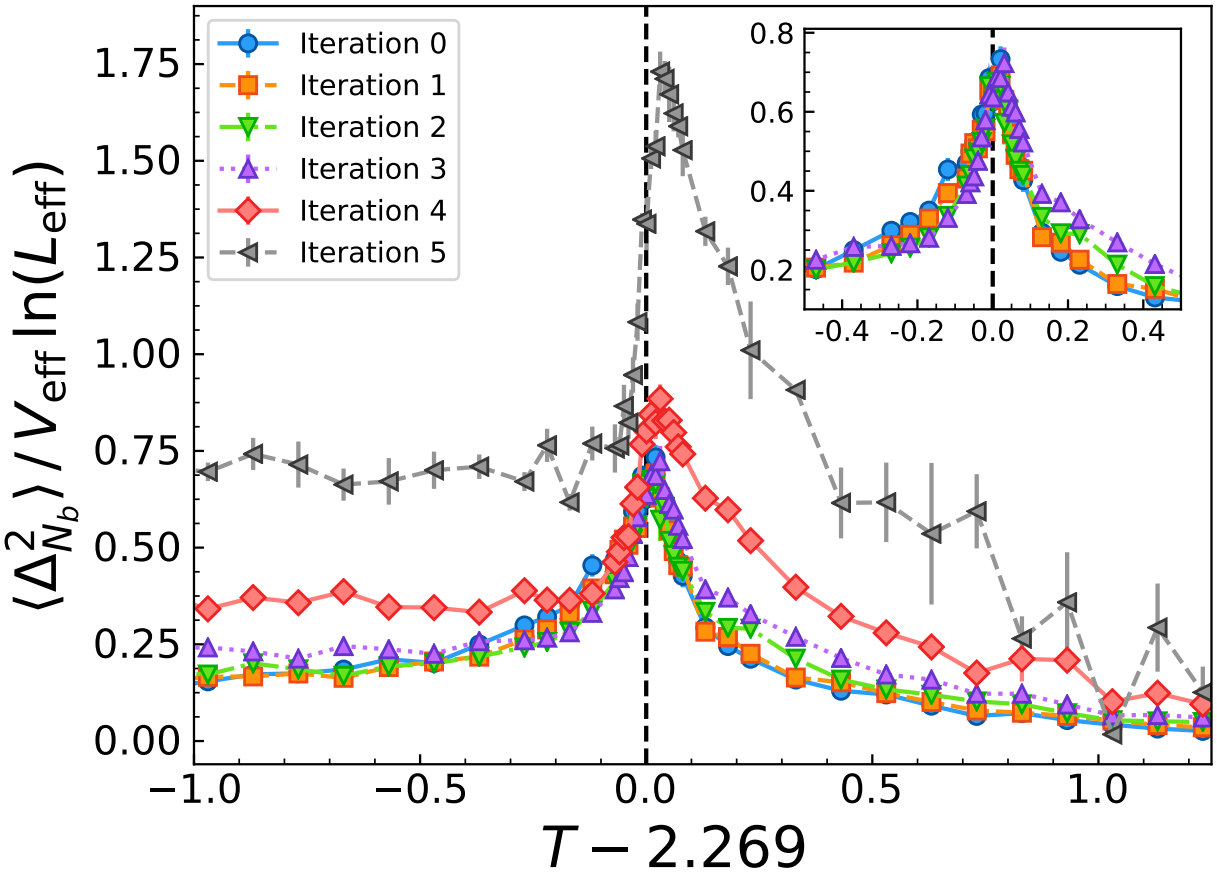


Figure 3.8: Fluctuations in the average number of bonds $\langle \Delta^2_{N_b} \rangle$ vs. temperature T under iterated blocking steps beginning with an initial lattice size of $L = 64$. The results are scaled by $1/V_{\text{eff}} \log(L_{\text{eff}})$ in order to demonstrate the data collapse near 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.

in the low temperature side, the curves sharpen in a way similar to T_{1100} in Fig. 3.5. However on 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 $1 + 1 \to 0$ 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 $1 + 1 \to 0$ 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

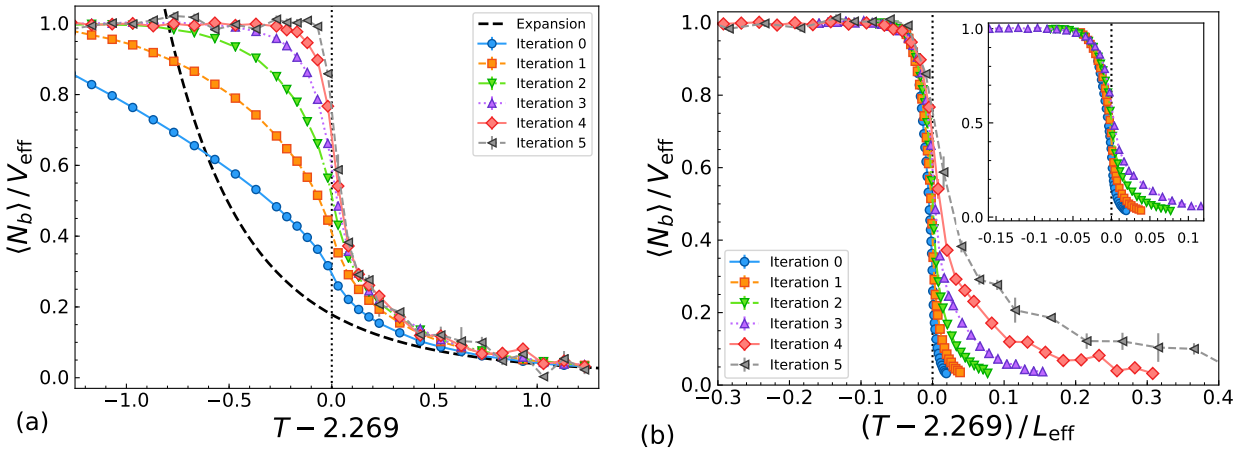


Figure 3.9: (a) Average number of bonds $\langle N_b \rangle$ vs. temperature T under iterated blocking steps beginning with an initial lattice size of $L = 64$. The dashed black line illustrates the high temperature expansion, showing that the dominant configurations are those consisting of small, isolated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

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:

$$\langle N_b \rangle = \frac{1}{Z} \sum_{\{n\}} \left(\sum_l n_l \right) \left(\prod_l \tanh^{n_l}(\beta) \right) \left(\prod_i \delta^{(i)}_{n_x + n_{x'} + n_y + n_{y'} \bmod 2,\, 0} \right). \tag{3.13}$$

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

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

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

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

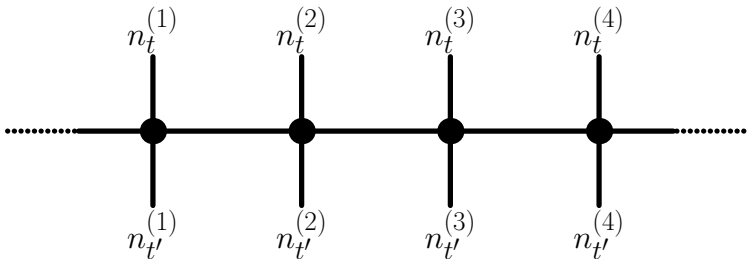


Figure 3.10: A pictorial representation of the transfer matrix made by contracting a fundamental tensor along a single time-slice.

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

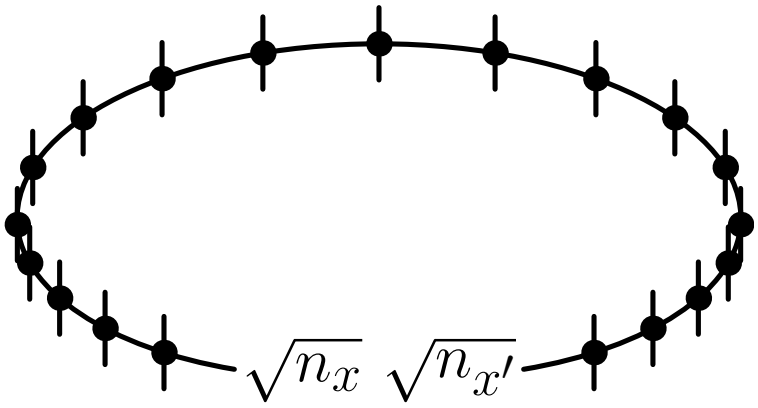


Figure 3.11: By multiplying the remaining free spatial indices by $\sqrt{n}$ and contracting for periodic boundary conditions in space we form an "impure" transfer matrix. Combining the resultant matrix with the original transfer matrix allows one to calculate $\langle n \rangle$.

level of approximation one will use in the blocking. For instance one could choose that the state $|1\ 1\rangle \to |0\rangle$ and assign $n = 0$ to that state. Alternatively one could preserve N_b and let $|1\ 1\rangle \to |2\rangle$ 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:

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

with

$$Z = \mathrm{Tr}[\mathbb{T}^{N_\tau}]. \tag{3.17}$$

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.

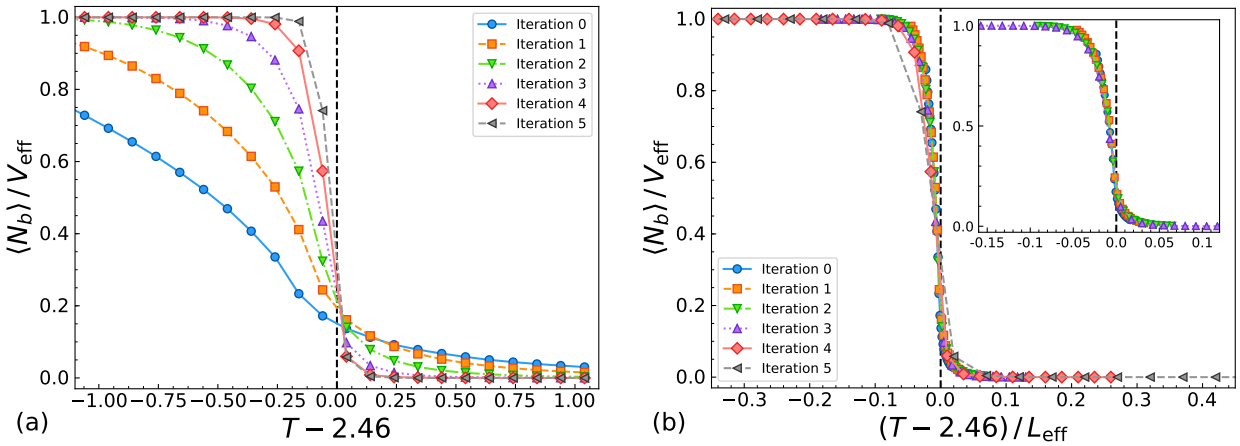


Figure 3.12: (a) $\langle N_b \rangle$ vs $(T - 2.46)$ under successive blocking steps calculated using 2-state HOTRG. (b) $\langle N_b \rangle$ vs $(T - 2.46)/L_{\text{eff}}$ under successive blocking steps calculated using 2-state HOTRG. Note that the value of 2.46 was determined qualitatively by choosing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

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

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

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 [36] and worm algorithms [34]. 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 [37].

Using the standard thermodynamical formula for the average energy

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

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

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

where we define

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

We can then use

$$C_V = \frac{\partial\langle E\rangle}{\partial T}, \tag{3.23}$$

to write

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

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 $\langle\Delta^2_{N_b}\rangle$ defined in Eq. (3.4). The singularity near T_c is known from Onsager's solution:

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

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 [34]. 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. [37]. The agreement can be seen in Fig. 3.13. Results for other lattice sizes that we have simulated are similar.

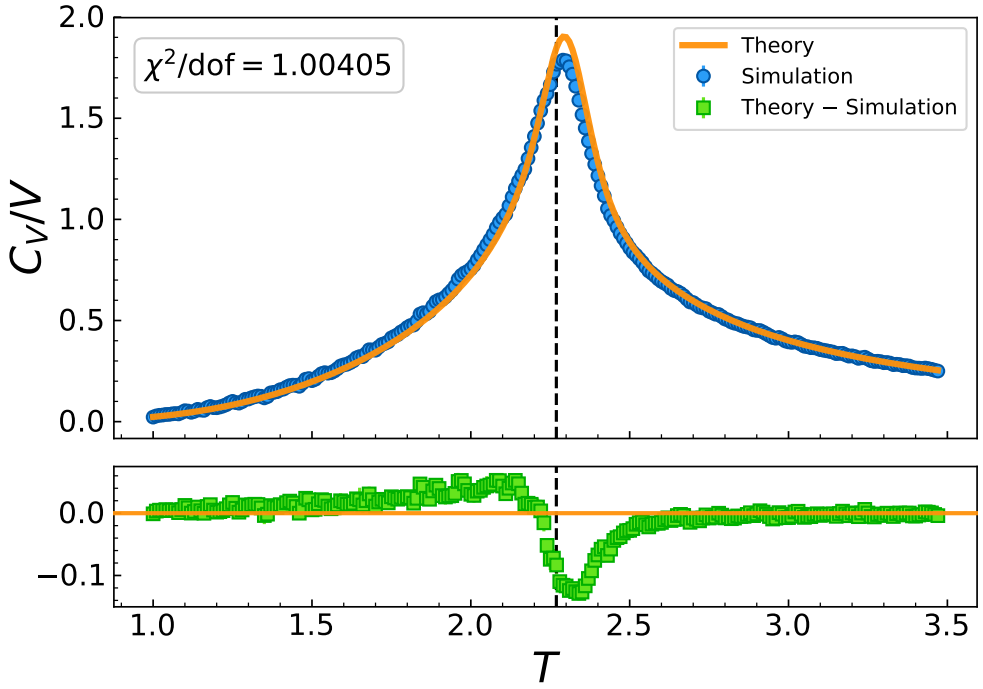


Figure 3.13: Comparison of the worm Monte Carlo computation of the specific heat C_v versus temperature and the exact results using the formula in [37], for an $L = 32$ lattice. Note that χ^2/dof represents the reduced chi-squared statistic. It can be seen that the agreement is excellent, except for a slight deviation at the critical temperature, where Monte Carlo algorithms tend to face difficulties with critical slowing down. This is mostly addressed with 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* $\lambda_{\max}$

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

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

$$= \left\langle \sum_{j=bonds} v_j \right\rangle \tag{3.27}$$

$$= 2V \langle \mathbf{v}_b \rangle. \tag{3.28}$$

From this, we have that

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

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,

$$\sum_{j=bonds} v_j = \sum_{j=sites} v_j. \tag{3.30}$$

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

$$\left\langle \sum_{j=bonds} v_j \right\rangle \simeq \left\langle \sum_{j=sites} v_j \right\rangle \Longrightarrow \tag{3.31}$$

$$\langle \mathbf{v}_b \rangle \simeq \frac{1}{2} \langle \mathbf{v}_s \rangle \tag{3.32}$$

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 site can be shared by either 0, 2, or 4 bonds. If the site is shared by two bonds, 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 us to break up the sum over bonds

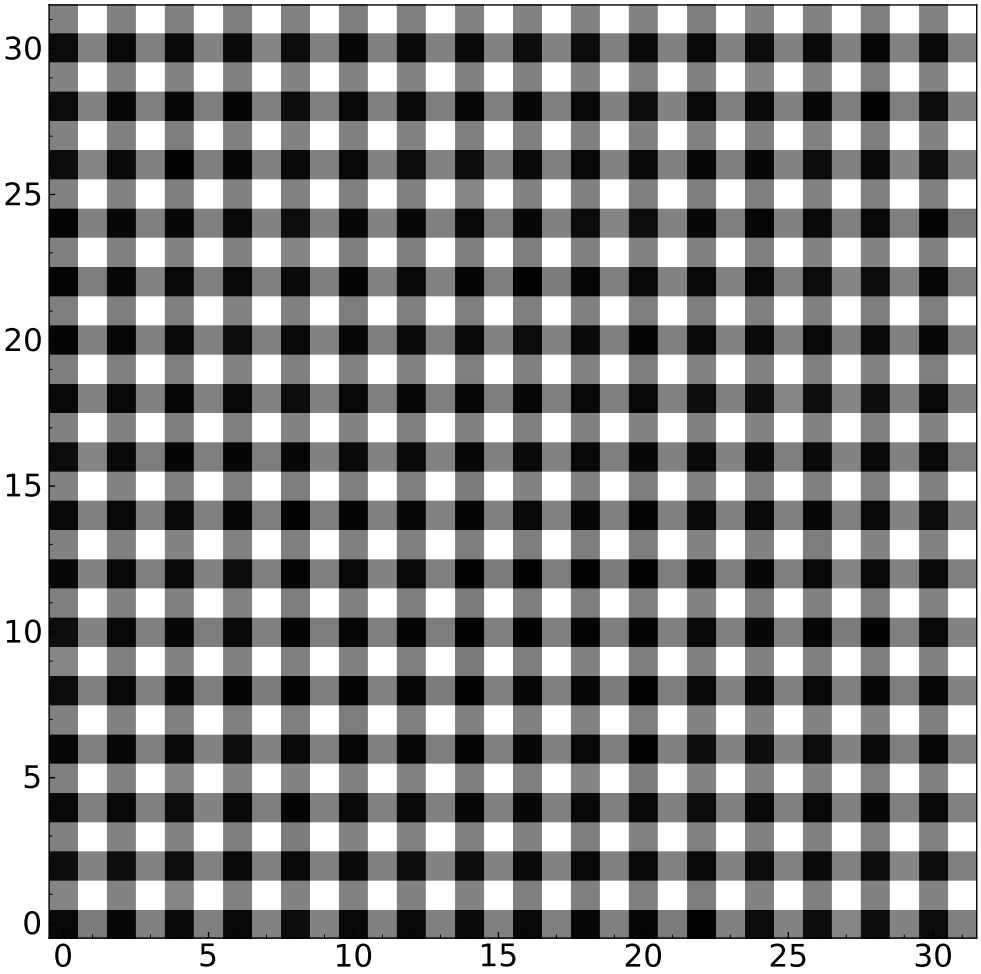


Figure 3.14: Average image $\langle \mathbf{v} \rangle$ calculated for the $L = 16$ lattice at $T = 2.0$, illustrating the tablecloth-like appearance.

into two terms

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

Rearranging and taking averages gives

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

$$= \left\langle \sum_{j=sites} v_j \right\rangle \tag{3.35}$$

$$= V \langle \mathbf{v}_s \rangle \tag{3.36}$$

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

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

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

We can rewrite the last equation using (3.28)

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

This suggests that a departure from a perfect tablecloth ($\langle \mathbf{v}_s \rangle = 2\langle \mathbf{v}_b \rangle$) contains information. Another useful construct is the *covariance matrix*,

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

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

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, $C \in \mathbb{R}^{N_{\text{configs}} \times 4L^2}$, 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,

$$C = W\Lambda W^T \tag{3.43}$$

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

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

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

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.

$$\mathbf{w}_1 \propto \langle\mathbf{v}\rangle. \tag{3.46}$$

From our results in 3.2, we can write

$$\langle\mathbf{v}\rangle^2 = \langle\mathbf{v}\rangle^T\langle\mathbf{v}\rangle \tag{3.47}$$

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

This suggests that

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

Moreover, we can write

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

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

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

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

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},

$$\mathbf{w}_1^T C \mathbf{w}_1 = \lambda^{(1)}$$

$$= \frac{1}{N_{\text{configs}}} \sum_{n=1}^{N_{\text{configs}}} \frac{\left[\langle \mathbf{v}_b \rangle \Delta_{N_b}^{(n)}\right]^2 + 2\langle \mathbf{v}_b \rangle \langle \mathbf{v}_s \rangle \Delta_{N_b}^{(n)} \Delta_{N_s}^{(n)} + \left[\langle \mathbf{v}_s \rangle \Delta_{N_s}^{(n)}\right]^2}{2\langle \mathbf{v}_b \rangle^2 + \langle \mathbf{v}_s \rangle^2}$$

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

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

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

$$= \frac{3}{2} \left\langle \Delta_{N_b}^2 \right\rangle. \tag{3.56}$$

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: $1 + 1 \to 1$ and $1 + 1 \to 2$.

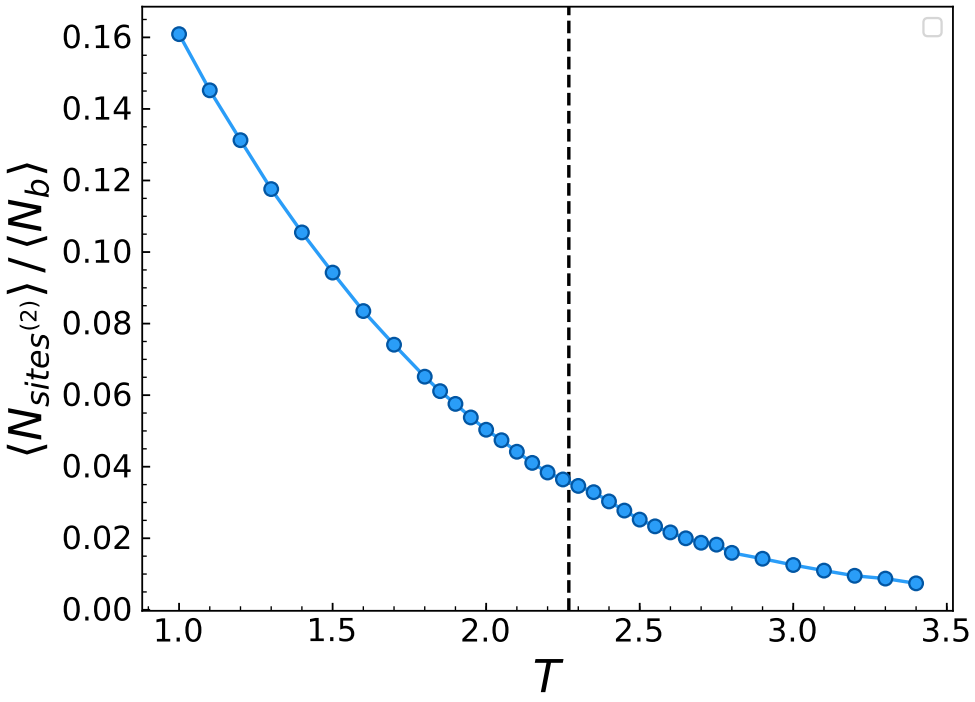
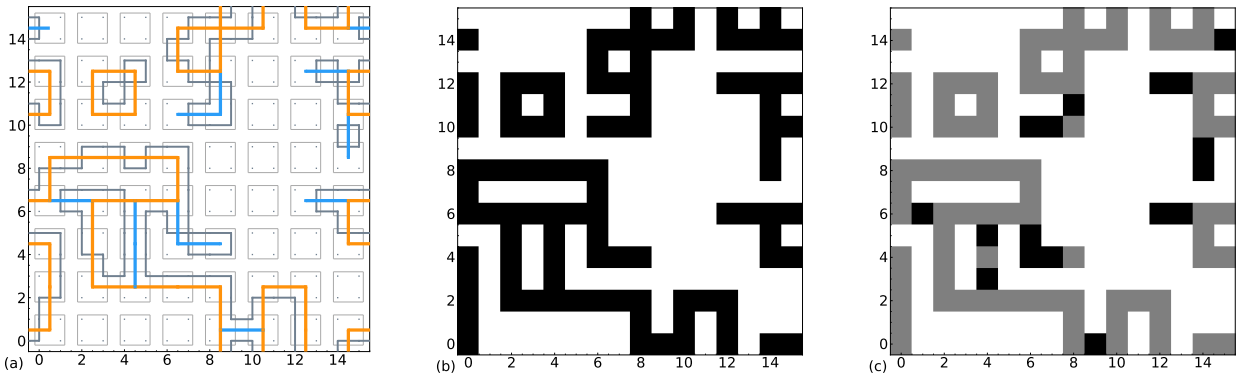


Figure 3.15: Ratio of the number of twice visited sites $\langle N_{sites^{(2)}} \rangle$ to the average number of bonds $\langle N_b \rangle$ versus temperature, for the $L = 32$ lattice. This clearly justifies our approximation $\langle \mathbf{v}_s \rangle \simeq 2\langle \mathbf{v}_b \rangle$, where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



(a) $1 + 1 = m$ (b) $1 + 1 \to 1$ (c) $1 + 1 \to 2$

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)) $1 + 1 \to 1$, double bonds are given the same weight as single bonds, and in (3.16(c)) $1 + 1 \to 2$, 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 [38] 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. This process was carried out on a mini-

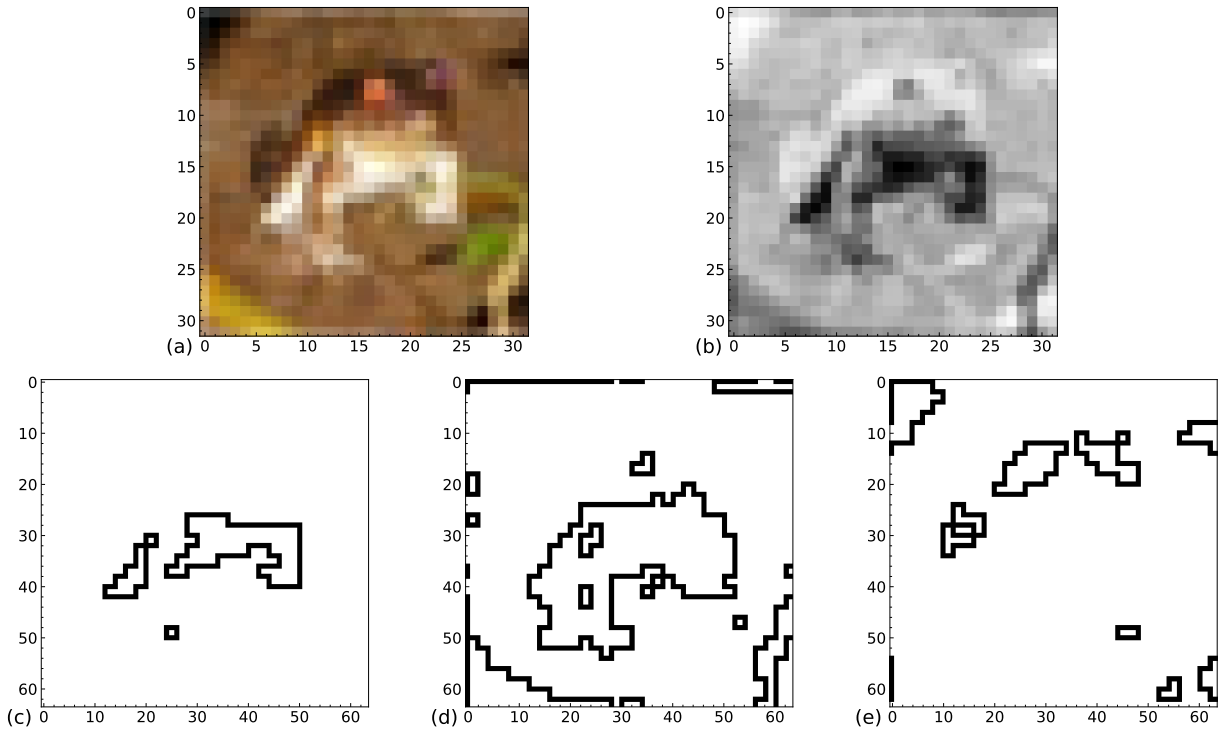


Figure 3.17: Example of preprocessing steps for converting CIFAR-10 images to 'worm-like' images, illustrating the resulting image for different values of the grayscale cuttoff. (a) Original image from CIFAR-10 dataset. (b) Image converted to grayscale. (c) Resulting image from cutoff values of 0.25,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

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 $\langle \Delta^2_{N_b} \rangle$ 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 $(1+1 \to 0)$ blocking procedure described in Sec. 3.4, calculating $\langle N_b \rangle$ and $\langle \Delta^2_{N_b} \rangle$ for each successive blocking step, as shown in Fig. 3.18. Immediately we see that there is no identifiable low temperature phase, and

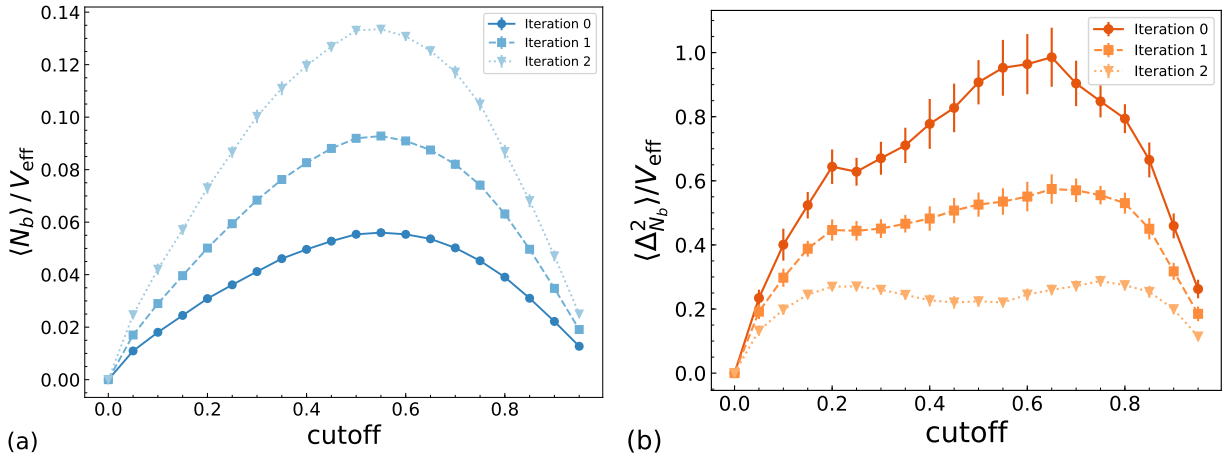


Figure 3.18: $\langle N_b \rangle$ and $\langle \Delta^2_{N_b} \rangle$ vs. grayscale cutoff value for 500 randomly chosen images from the CIFAR-10 dataset.

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 [39–41]. By mapping from fine to coarse in several ways, such as the $1+1 \to 0$ and $1+1 \to 1$ 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 [28, 42]. 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 repeeated and is very reminiscent of the procedure proposed by Hinton and Salakhutdinov [43] 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 [44]. 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 $\{x_1, x_2, \ldots, x_N\}$ 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 $\{x_1, x_2, \ldots, x_{t-1}\}$ [45]. 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 $\lambda = (\lambda_1, \ldots, \lambda_N)$ defined by

$$p(x_1 = x_i) = \lambda_i, \quad i = 1, \ldots, N \tag{4.1}$$

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

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

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 $i = 1, 2, \ldots, N$,

$$\sum_{j=1}^{N} p_{ij} = 1. \tag{4.3}$$

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

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

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

  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 $(x_{i+1}, \ldots, x_{i+k})$ and $(x_{i+k}, \ldots x_{i+1})$ 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 [46].

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

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

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

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

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

$$p(x') = \int K(x'|x)p(x)dx \tag{4.6}$$

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

$$p(x')K(x|x') = p(x)K(x'|x). \tag{4.7}$$

We can rewrite the detailed balance condition as

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

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.

In practice, a Gaussian distribution is often used as the proposal distribution, i.e. $q(x'|x) =$

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

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 $x_{t+1} = x_t$.

$\mathcal{N}(x'|x, \Sigma)$. In this case the acceptance probability reduces to

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

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 [47]. Recall Bayes' theorem:

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

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 as spam. The test for spam is that the message contains some flagged words B_i, i.e. $B_1 =$ 'winner', $B_2 =$ '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. $P(A|B_1) = 10^{-4}, P(A|B_2) = 0.05, \ldots$) 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 $p(v) \propto \exp\left(-\frac{1}{2}v^T v\right)$ 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*,

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

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.

$$\dot{x}_i = \frac{\partial \mathcal{H}}{\partial v_i} = v_i \tag{4.14}$$

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

It can be shown [48] 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:

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

$$x' = x + \varepsilon v^{\frac{1}{2}} \tag{4.17}$$

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

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:

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

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 [49]. To lift an initial point in parameter space into one on phase space, we simply sample from the conditional distribution over the momentum,

$$v \sim \pi(x|v). \tag{4.20}$$

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 $\xi \to \xi'$. We can then return to the target parameter space by simply projecting away the momentum,

$$(x, v) \to x \tag{4.21}$$

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 $\mathbf{L} : \xi(t) \to \xi' = \xi(t+s)$ is one-to-one, and consequently has an inverse $\mathbf{L}^{-1}$, obtained by negating the time

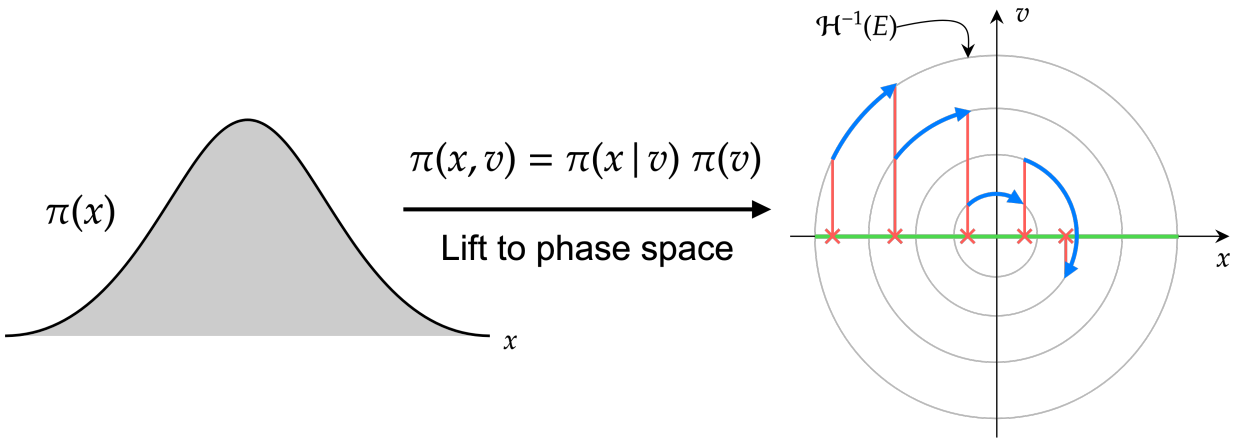


Figure 4.1: *Visualizing HMC for a* $1D$ *Gaussian* (example from [49], figure adapted with permission from [50]). Each Hamiltonian Markov transition lifts the initial state onto a random level set of the Hamiltonian, $\mathcal{H}^{(-1)}(E)$, which can then be explored with a Hamiltonian trajectory before projecting back down to the target parameter space.

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 $\mathbf{L}_\theta$, 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 $\mathbf{L}_\theta$.

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

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

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'':

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

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

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

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

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

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)* [51].

5.2.1 Metropolis-Hastings Accept/Reject

Written in terms of these transformations, the augmented leapfrog operator $\mathbf{L}_\theta$ 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

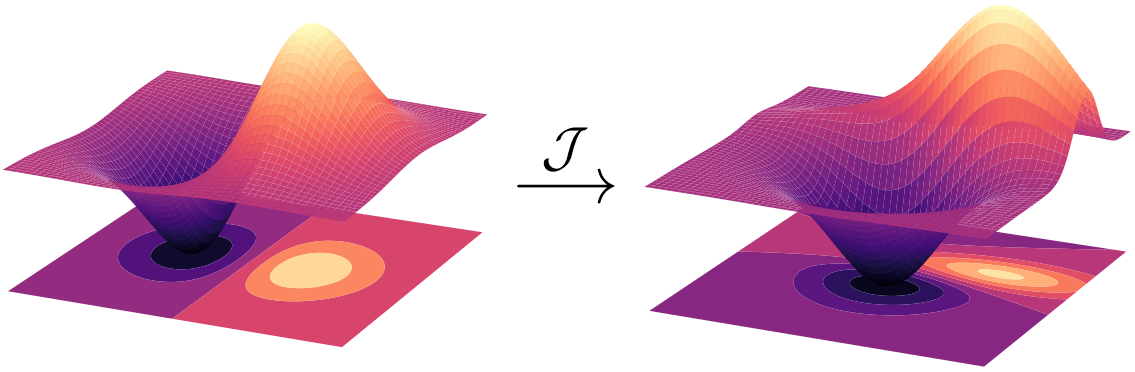


Figure 5.1: Example of how the determinant of the Jacobian can deform the energy landscape.

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

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

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. $v'' \to v'$, then $x'' \to x'$, followed by $x' \to x$ and finally $v' \to v$). 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 $(d = 1)$:

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

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

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

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

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 $(d = -1)$:

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

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

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

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

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:

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

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

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 paramterizing 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.

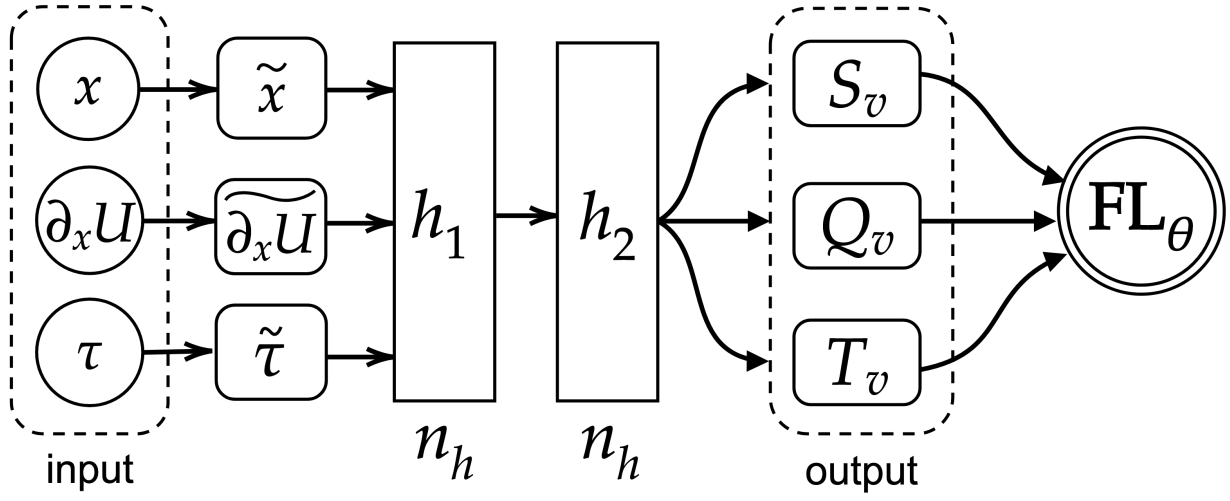


Figure 5.2: Illustration showing the generic (fully-connected) network architecture for training S_v, Q_v, and T_v. Figure adapted with permission from [50].

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 $\tau(t) = \left(\cos\left(\frac{2\pi t}{N_{\text{LF}}}\right),\ \sin\left(\frac{2\pi t}{N_{\text{LF}}}\right)\right)$. Each of the inputs is then passed through a fully-connected ('dense' layer), consisting of n_h hidden units

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

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

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

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

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

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

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

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

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

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

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

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:

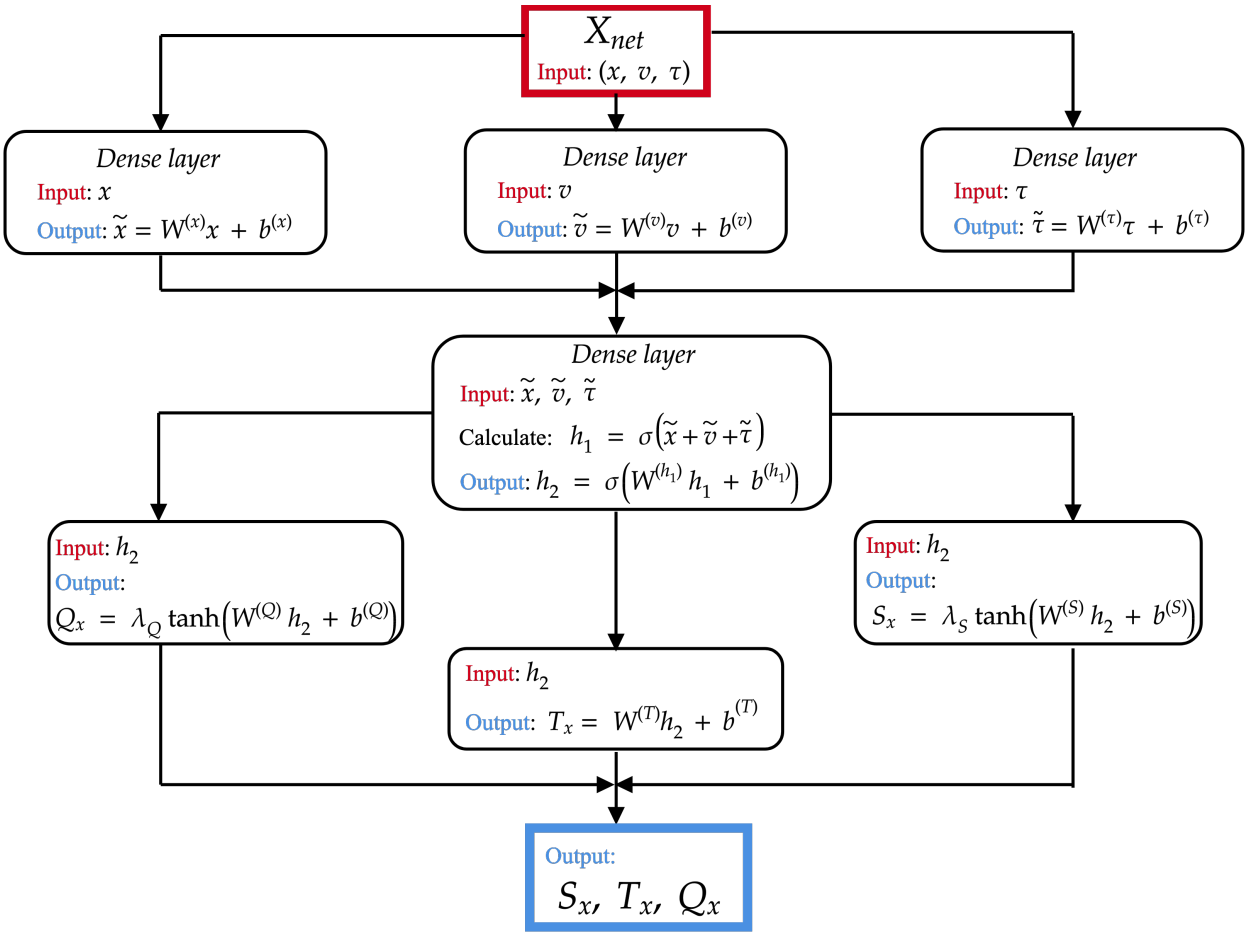


Figure 5.3: Flowchart illustrating the generic fully-connected network architecture including the intermediate variables computed at each hidden layer of the network.

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 [52], 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

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

Then, the expected squared jumped distance is given by $\mathbb{E}_{\xi\sim p(\xi)}\left[\delta(\mathbf{FL}_\theta\xi, \xi)A(\mathbf{FL}_\theta\xi|\xi)\right]$. 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

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

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

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

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, $U : \mathcal{X} \to \mathbb{R}$ and its gradient $\nabla_x U : \mathcal{X} \to \mathcal{X}$
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, $\left\{\xi_q^{(i)}\right\}_{i \leq N_{\text{samples}}}$ from $q(\xi)$.

  $\mathcal{L} \leftarrow 0$

  for $i = 1$ **to** N_{LF} **:**

    $\xi_p^{(i)} \leftarrow \mathbf{R}\, \xi_p^{(i)}$

    $\mathcal{L} \leftarrow \mathcal{L} + \ell_\lambda \left(\xi_p^{(i)}, \mathbf{FL}_\theta \xi_p^{(i)}, A(\mathbf{FL}_\theta \xi_p^{(i)} | \xi_p^{(i)})\right) + \lambda_b \ell_\lambda \left(\xi_q^{(i)}, \mathbf{FL}_\theta \xi_q^{(i)}, A(\mathbf{FL}_\theta \xi_q^{(i)} | \xi_q^{(i)})\right)$

    $\xi_p^{(i)} \leftarrow \mathbf{FL}_\theta \xi_p^{(i)}$ 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 $D \geq 1$:

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

where $\sum_{m=1}^{M} \pi_m = 1$, $\pi_M \in (0, 1)\ \forall m = 1, \ldots, M$ 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 $\boldsymbol{\Sigma}_m \equiv \mathbb{E}_{p(\mathbf{x}|m)}\left\{(\mathbf{x} - \boldsymbol{\mu}_m)(\mathbf{x} - \boldsymbol{\mu}_m)^T\right\} > 0$ 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

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

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

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

The results of trajectories generated using both traditional HMC and the L2HMC algorithm can be seen in Fig. 5.4. a 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".

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

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

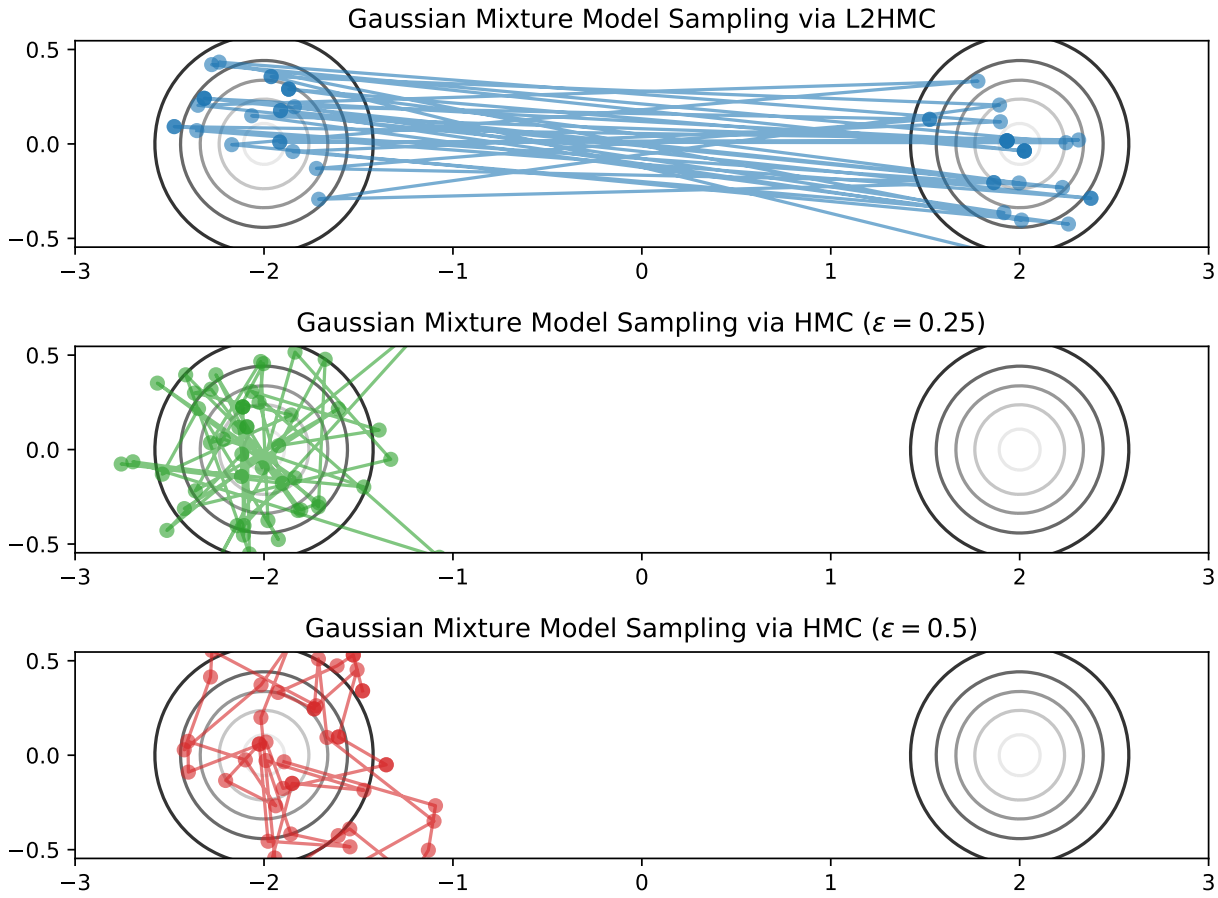


Figure 5.4: Comparison of trajectories generated using L2HMC (top), and traditional HMC with $\varepsilon = 0.25$ (middle) and $\varepsilon = 0.5$ (bottom). Note that L2HMC is able to successfully mix between modes, whereas HMC is not.

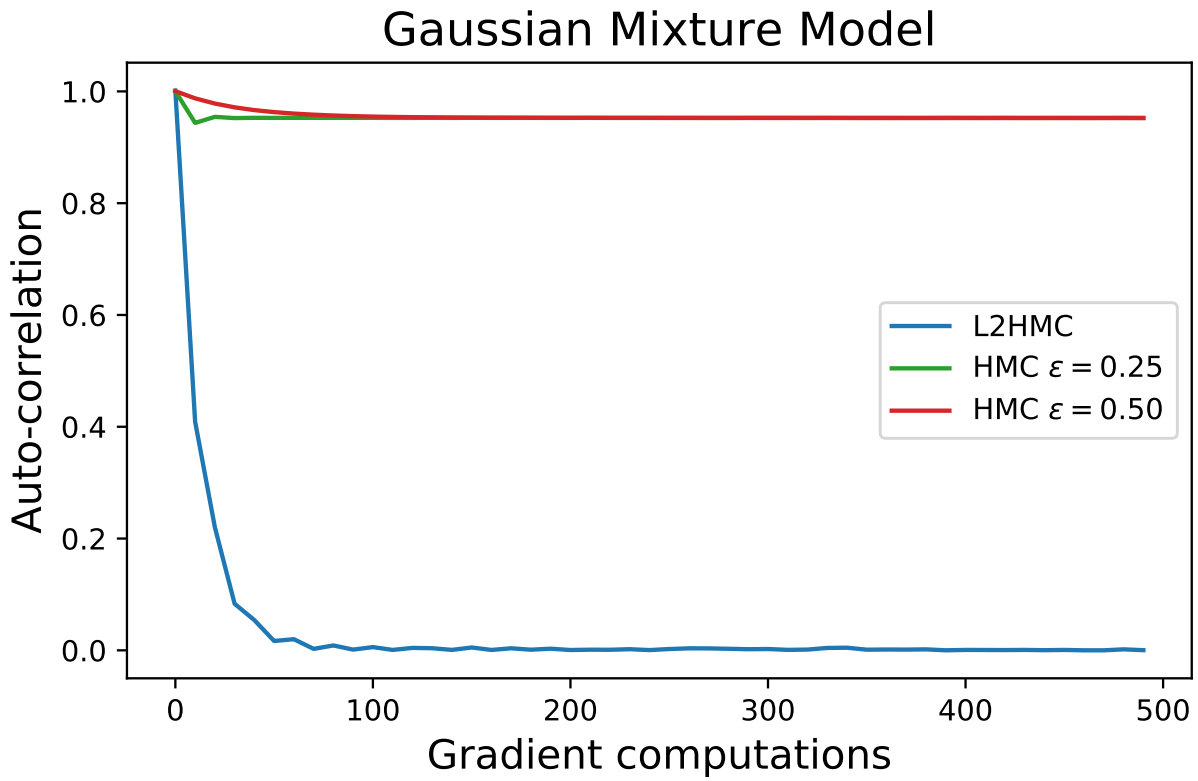


Figure 5.5: Autocorrelation vs. gradient evaluations (i.e. MD steps). Note that L2HMC (blue) has a significantly reduced autocorrelation after the same number of gradient evaluations when compared to either of the two HMC trajectories

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 $-\pi < \phi_\mu(i) \leq \pi$.

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

with the Wilson action defined as:

$$\beta S = \beta \sum_P \left(1 - \cos\left(\phi_P\right)\right) \tag{5.32}$$

where

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

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

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

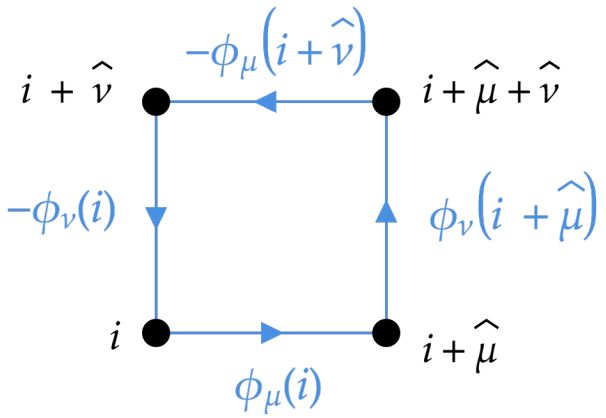


Figure 5.6: Illustration of an elementary plaquette on the lattice.

where

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

is the sum of the link variables around the elementary plaquette, projected onto the interval $[-\pi, \pi)$. From this, we can define topological susceptibility

$$\chi \equiv \frac{\langle Q^2 \rangle - \langle Q \rangle^2}{V} \tag{5.36}$$

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

$$\chi = \frac{\langle Q^2 \rangle}{V} \tag{5.37}$$

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

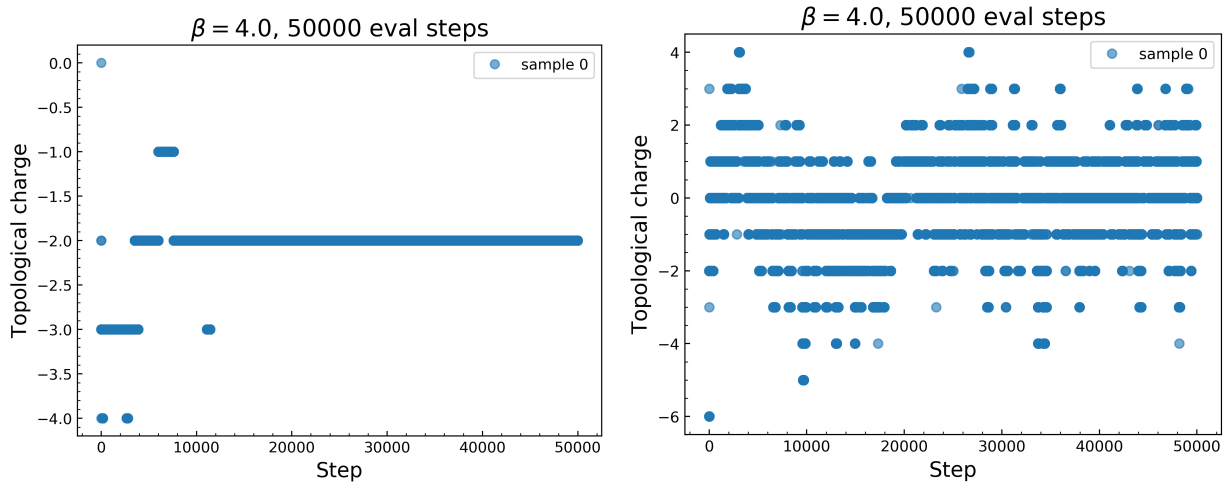


Figure 5.7: (left) Example of topological freezing in the $2D$ $U(1)$ lattice gauge theory, generated from generic HMC sampling for a 16×16 lattice. Note that for the majority of the simulation $Q = -2$, making it virtually impossible to get a reasonable estimate of χ. (right) Topological charge vs.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

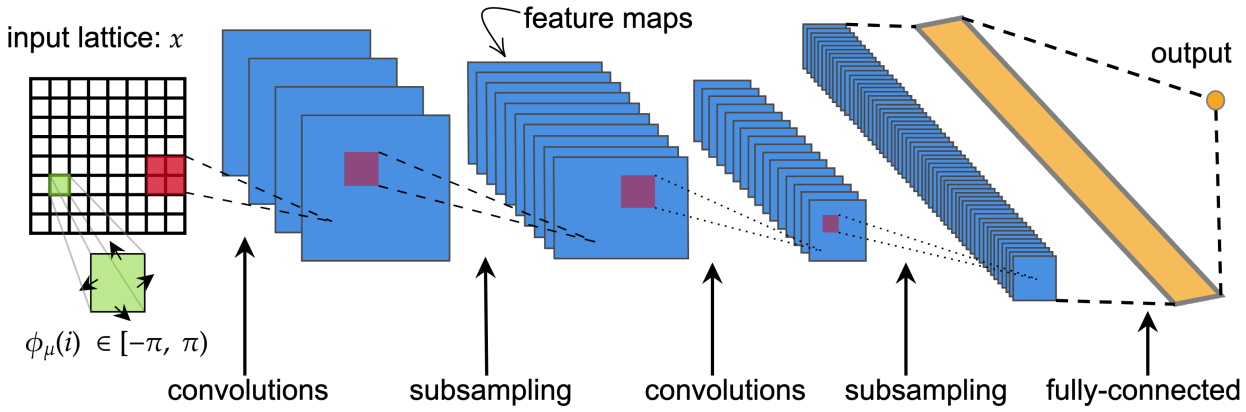


Figure 5.8: Convolutional structure used for learning localized features of rectangular lattice.

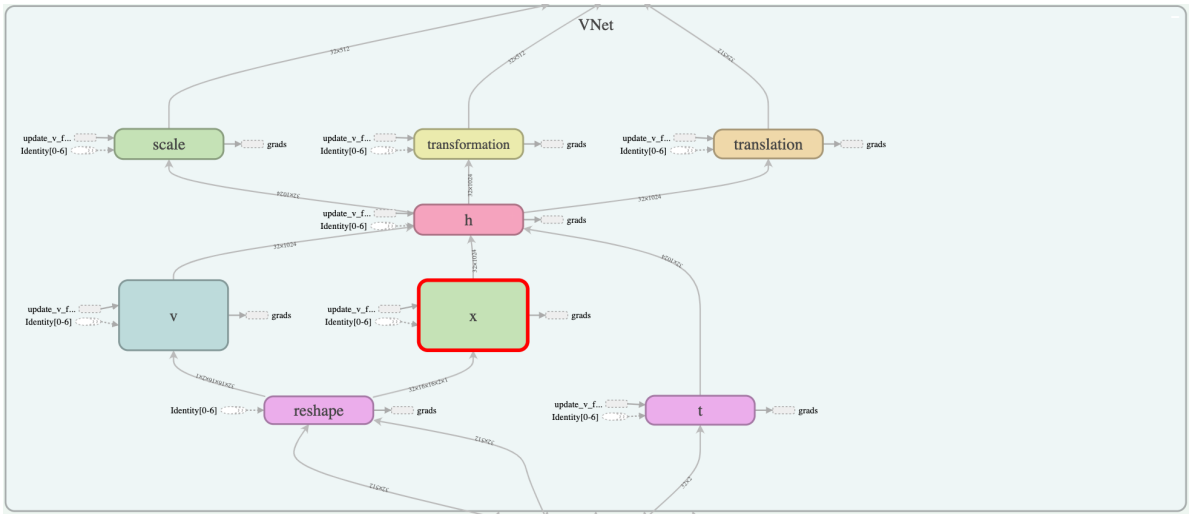


Figure 5.9: Illustration taken from TensorBoard showing an overview of the network architecture for VNet. Note that the architecture is identical for XNet.

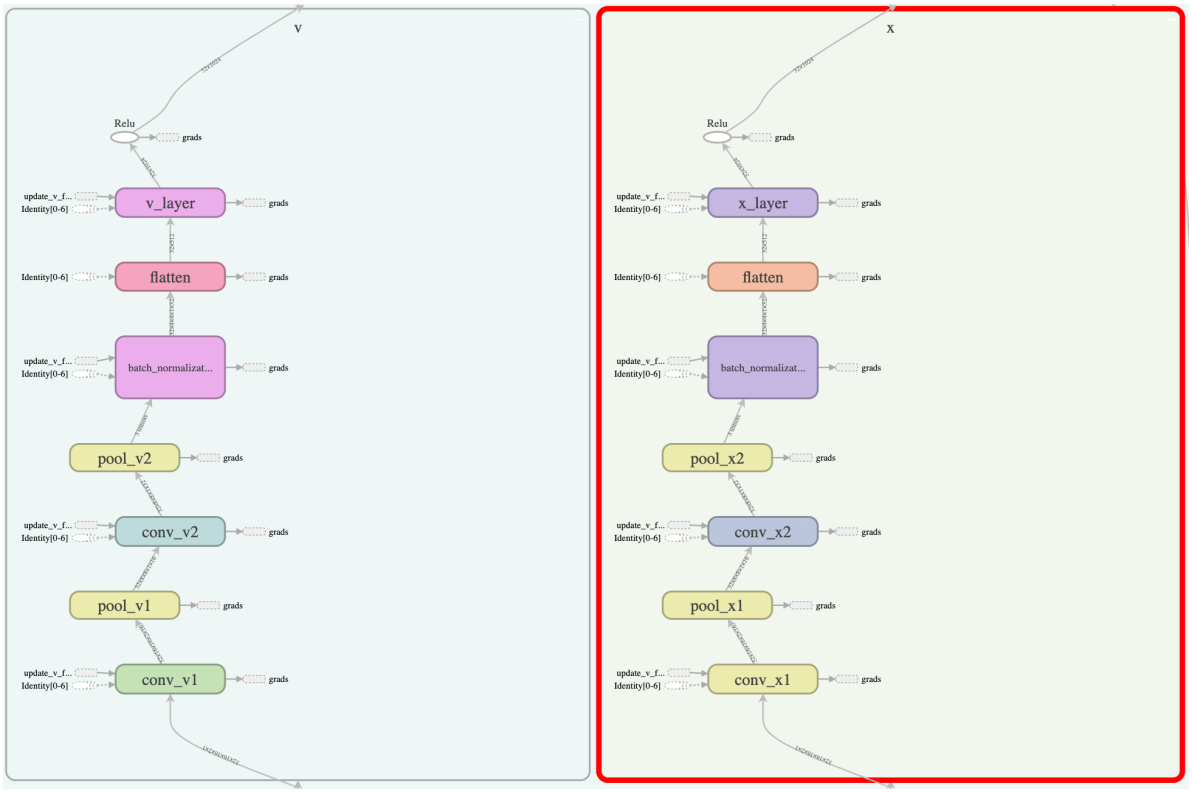


Figure 5.10: Detailed view of additional convolutional structure included to better account for rectangular geometry of lattice inputs.

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 gradients[1], (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 [53].

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.

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

Here $\beta(n)$ denotes the value of β to be used for the n^{th} training step ($n = 1, \ldots, N_{\text{train}}$), 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

$$\delta(\xi, \xi') \equiv 1 - \cos\left(\xi - \xi'\right) \tag{5.39}$$

where now $\xi \equiv \left(\phi^x_\mu(i),\, \phi^v_\mu(i),\, d\right)$, with ϕ^x_μ representing the lattice of ('position') gauge variables (what we called x previously), and ϕ^v_μ representing the lattice of ('momentum') gauge variables (what we called v previously). Note that i runs over all lattice sites[2] and $\mu = 0, 1$ for the two dimensional case. We see that this metric gives the expected behavior, since $\delta \to 0$ for $\xi \approx \xi'$.

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

[2] In 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

$$\delta_Q(\xi, \xi') = \big|Q(\xi) - Q(\xi')\big| \tag{5.40}$$

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

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

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

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,

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

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

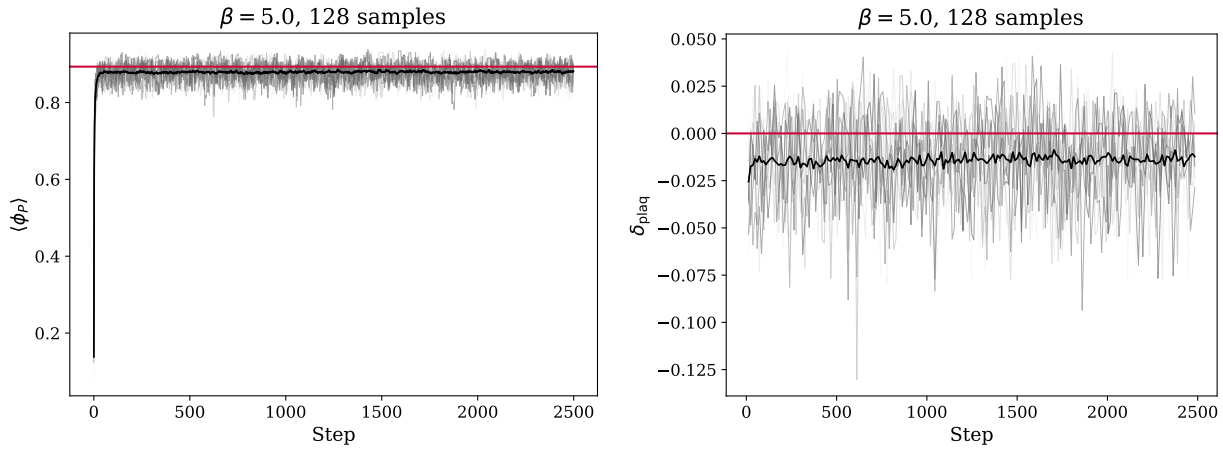


Figure 5.11: (left): Average plaquette $\langle \phi_P \rangle$ vs. step for $L = 8$, $N_{\text{LF}} = 7$, and $\alpha_Q = 0.5$. Here the solid red line indicates the true value of the average plaquette (in the infinite volume limit, and is equal to $0.89338\ldots$ **(right)**: Difference between the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

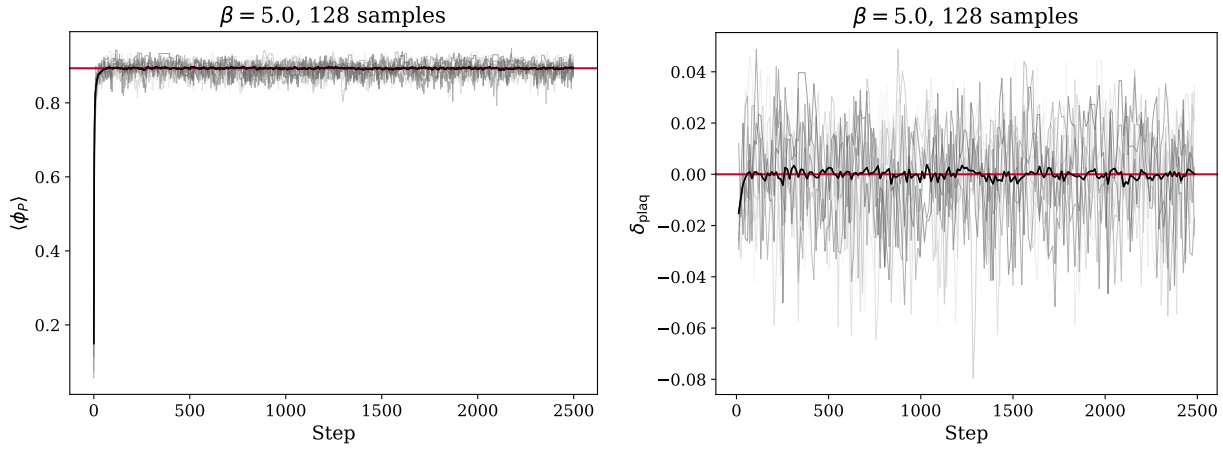


Figure 5.12: Same quantities as in Fig 5.11, with $\alpha_Q = 0$. Note that the discrepancy δ_{ϕ_P} is no longer present.

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

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

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 Appendix 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 super-computers. 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 distribution[1], 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 $2DU(1)$ 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`.

[1] simple 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 dividied 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}
}
```

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

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

```
# ================================================================#
#                                                                 #
#  + All available command line arguments can be found (along with  #
#    descriptions in the file:                                    #
#        `utils/parse_args.py`.                                   #
#                                                                 #
#  + To pass the arguments defined below to `gauge_model.py`:       #
#        `python3 gauge_model.py @gauge_model_args.txt`           #
#                                                                 #
# ================================================================#

--time_size 8
--space_size 8
--num_samples 128
#--rand
--num_steps 5
--eps 0.2
--annealing
--beta_init 2.
--beta_final 5.
--lr_init 0.001
--lr_decay_steps 1000
--lr_decay_rate 0.96
--train_steps 1000
--run_steps 2000
--save_steps 1000
--print_steps 1
--logging_steps 10
--network_arch 'conv2D'
--metric 'cos_diff'
--num_hidden 256
--aux_weight 1.
--std_weight 1.
--charge_weight 0.
--loss_scale 0.1
--summaries
--eps_trainable
--use_bn
--save_lf
--loop_net_weights
```

A.3 l2hmc-qcd/gauge_model_main.py

```
"""
gauge_model_main.py

Main method implementing the L2HMC algorithm for a 2D U(1) lattice gauge theory
with periodic boundary conditions.

Following an object oriented approach, there are separate classes responsible
for each major part of the algorithm:

    (1.) Creating the loss function to be minimized during training and
    building the corresponding TensorFlow graph.

        - This is done using the `GaugeModel` class, found in
        `models/gauge_model.py`.

        - The `GaugeModel` class depends on the `GaugeDynamics` class
        (found in `dynamics/gauge_dynamics.py`) that performs the augmented
        leapfrog steps outlined in the original paper.

    (2.) Training the model by minimizing the loss function over both the
    target and initialization distributions.
        - This is done using the `GaugeModelTrainer` class, found in
        `trainers/gauge_model_trainer.py`.

    (3.) Running the trained sampler to generate statistics for lattice
    observables.
        - This is done using the `GaugeModelRunner` class, found in
        `runners/gauge_model_runner.py`.

Author: Sam Foreman (github: @saforem2)
Date: 04/10/2019
"""
import os
import random
import time
import tensorflow as tf
import numpy as np

from tensorflow.python import debug as tf_debug
from tensorflow.python.client import timeline
from tensorflow.core.protobuf import rewriter_config_pb2

import utils.file_io as io

from globals import GLOBAL_SEED, NP_FLOAT
from utils.parse_args import parse_args
from utils.model_loader import load_model
from models.gauge_model import GaugeModel
from loggers.train_logger import TrainLogger
from loggers.run_logger import RunLogger
from trainers.gauge_model_trainer import GaugeModelTrainer
from plotters.gauge_model_plotter import GaugeModelPlotter
from plotters.leapfrog_plotters import LeapfrogPlotter
from runners.gauge_model_runner import GaugeModelRunner

try:
    import horovod.tensorflow as hvd
```

```
    HAS_HOROVOD = True
except ImportError:
    HAS_HOROVOD = False

try:
    import matplotlib.pyplot as plt
    HAS_MATPLOTLIB = True
except ImportError:
    HAS_MATPLOTLIB = False

if float(tf.__version__.split('.')[0]) <= 2:
    tf.logging.set_verbosity(tf.logging.INFO)

# ----------------------------------------------------------------------
# Set random seeds for tensorflow and numpy
# ----------------------------------------------------------------------
os.environ['PYTHONHASHSEED'] = str(GLOBAL_SEED)
random.seed(GLOBAL_SEED)        # `python` build-in pseudo-random generator
np.random.seed(GLOBAL_SEED)     # numpy pseudo-random generator
tf.set_random_seed(GLOBAL_SEED)


def create_config(FLAGS, params):
    """Create tensorflow config."""
    config = tf.ConfigProto()
    if FLAGS.time_size > 8:
        off = rewriter_config_pb2.RewriterConfig.OFF
        config_attrs = config.graph_options.rewrite_options
        config_attrs.arithmetic_optimization = off

    if FLAGS.gpu:
        # Horovod: pin GPU to be used to process local rank (one GPU per
        # process)
        config.gpu_options.allow_growth = True
        #  config.allow_soft_placement = True
        if HAS_HOROVOD and FLAGS.horovod:
            num_gpus = hvd.size()
            io.log(f"Number of GPUs: {num_gpus}")
            config.gpu_options.visible_device_list = str(hvd.local_rank())

    if HAS_MATPLOTLIB:
        params['_plot'] = True

    if FLAGS.theta:
        params['_plot'] = False
        io.log("Training on Theta @ ALCF...")
        params['data_format'] = 'channels_last'
        os.environ["KMP_BLOCKTIME"] = str(0)
        os.environ["KMP_AFFINITY"] = (
            "granularity=fine,verbose,compact,1,0"
        )
        # NOTE: KMP affinity taken care of by passing -cc depth to aprun call
        OMP_NUM_THREADS = 62
        config.allow_soft_placement = True
        config.intra_op_parallelism_threads = OMP_NUM_THREADS
        config.inter_op_parallelism_threads = 0

    return config, params


```

```
def count_trainable_params(out_file):
    t0 = time.time()
    io.log(f'Writing parameter counts to: {out_file}.')
    io.log_and_write(80 * '-', out_file)
    total_params = 0
    for var in tf.trainable_variables():
        # shape is an array of tf.Dimension
        shape = var.get_shape()
        io.log_and_write(f'var: {var}', out_file)
        #  var_shape_str = f'  var.shape: {shape}'
        io.log_and_write(f'  var.shape: {shape}', out_file)
        io.log_and_write(f'  len(var.shape): {len(shape)}', out_file)
        var_params = 1  # variable parameters
        for dim in shape:
            io.log_and_write(f'    dim: {dim}', out_file)
            #  dim_strs += f'    dim: {dim}\'
            var_params *= dim.value
        io.log_and_write(f'variable_parameters: {var_params}', out_file)
        io.log_and_write(80 * '-', out_file)
        total_params += var_params

    io.log_and_write(80 * '-', out_file)
    io.log_and_write(f'Total parameters: {total_params}', out_file)
    t1 = time.time() - t0
    io.log_and_write(f'Took: {t1} s to complete.', out_file)


def hmc(FLAGS, params=None, log_file=None):
    """Create and run generic HMC sampler using trained params from L2HMC."""
    condition1 = not FLAGS.horovod
    condition2 = FLAGS.horovod and hvd.rank() == 0
    is_chief = condition1 or condition2
    if not is_chief:
        return -1

    FLAGS.hmc = True

    FLAGS.log_dir = io.create_log_dir(FLAGS, log_file=log_file)

    if params is None:
        params = {}
        for key, val in FLAGS.__dict__.items():
            params[key] = val

    params['hmc'] = True
    params['use_bn'] = False
    params['log_dir'] = FLAGS.log_dir
    params['data_format'] = None
    params['eps_trainable'] = False

    figs_dir = os.path.join(params['log_dir'], 'figures')
    io.check_else_make_dir(figs_dir)

    io.log(80 * '-' + '\n')
    io.log('HMC PARAMETERS:')
    for key, val in params.items():
        io.log(f'  {key}: {val}')
    io.log(80 * '-' + '\n')

    # create tensorflow config (`config_proto`) to configure session
```

```
    config, params = create_config(FLAGS, params)
    tf.reset_default_graph()
    sess = tf.Session(config=config)

    model = GaugeModel(params=params)
    run_logger = RunLogger(model, params['log_dir'], save_lf_data=False)
    plotter = GaugeModelPlotter(model, run_logger.figs_dir)

    sess.run(tf.global_variables_initializer())

    runner = GaugeModelRunner(sess, model, run_logger)

    if FLAGS.hmc_beta is None:
        betas = [FLAGS.beta_final]
    else:
        betas = [float(FLAGS.hmc_beta)]

    for beta in betas:
        # to ensure hvd.rank() == 0
        if run_logger is not None:
            run_dir, run_str = run_logger.reset(model.run_steps, beta)

        t0 = time.time()

        runner.run(model.run_steps, beta)

        run_time = time.time() - t0
        io.log(f'Took: {run_time} s to complete run.')

        if plotter is not None and run_logger is not None:
            plotter.plot_observables(run_logger.run_data, beta, run_str)
            if FLAGS.save_lf:
                lf_plotter = LeapfrogPlotter(plotter.out_dir, run_logger)
                lf_plotter.make_plots(run_dir, num_samples=20)

    return sess, model, runner, run_logger


def train_l2hmc(FLAGS, log_file=None):
    """Train L2HMC using GaugeModelTrainer."""
    pass


def l2hmc(FLAGS, log_file=None):
    """Create, train, and run L2HMC sampler on 2D U(1) gauge model."""
    tf.keras.backend.clear_session()
    tf.reset_default_graph()

    FLAGS.log_dir = io.create_log_dir(FLAGS, log_file=log_file)

    params = {}
    for key, val in FLAGS.__dict__.items():
        params[key] = val

    if FLAGS.gpu:
        io.log("Using GPU for training.")
        params['data_format'] = 'channels_first'
    else:
        io.log("Using CPU for training.")
        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)
```

```
    tf.keras.backend.set_learning_phase(True)

    # set initial value of charge weight using value from FLAGS
    charge_weight_init = FLAGS.charge_weight
    net_weights_init = [1., 1., 1.]
    samples_init = np.reshape(np.array(model.lattice.samples, dtype=NP_FLOAT),
                              (model.num_samples, model.x_dim))
    beta_init = model.beta_init
    init_feed_dict = {
        model.x: samples_init,
        model.beta: beta_init,
        model.charge_weight: charge_weight_init,
        model.net_weights[0]: net_weights_init[0],  # scale_weight
        model.net_weights[1]: net_weights_init[1],  # transformation_weight
        model.net_weights[2]: net_weights_init[2],  # translation_weight
    }
    scaffold = tf.train.Scaffold(init_feed_dict=init_feed_dict)
    # The MonitoredTrainingSession takes care of session
    # initialization, restoring from a checkpoint, saving to a
    # checkpoint, and closing when done or an error occurs.
    sess = tf.train.MonitoredTrainingSession(
        checkpoint_dir=checkpoint_dir,
        scaffold=scaffold,
        hooks=hooks,
        config=config,
        save_summaries_secs=None,
        save_summaries_steps=None
    )
    trainer = GaugeModelTrainer(sess, model, train_logger)
    kwargs = {
        'samples_np': samples_init,
        'beta_np': beta_init,
        'charge_weight': charge_weight_init,
        'net_weights': net_weights_init
    }

    try:
        trainer.train(model.train_steps, **kwargs)
    except NameError:
        # i.e. Tensor had Inf / NaN values caused by high learning rate
        io.log('\n\n' + 80 * '-')
        io.log('Training crashed! Decreasing lr_init by 10% and retrying...')
        io.log(f'Previous lr_init: {FLAGS.lr_init}')
        FLAGS.lr_init
        io.log(f'New lr_init: {FLAGS.lr_init}')
        io.log('Restarting training...')
        io.log(80 * '-' + '\n\n')
        params['log_dir'] = FLAGS.log_dir = None
        sess.close()
        tf.keras.backend.clear_session()
        tf.reset_default_graph()
        l2hmc(FLAGS)

    trainable_params_file = os.path.join(FLAGS.log_dir, 'trainable_params.txt')
    count_trainable_params(trainable_params_file)

    tf.keras.backend.set_learning_phase(False)

    runner = GaugeModelRunner(sess, model, run_logger)
    betas = [model.beta_final]  # model.beta_final + 1]
```

```
    if FLAGS.loop_net_weights:
        net_weights_arr = np.array([[1, 1, 1],  # [Q, S, T]
                                    [0, 1, 1],
                                    [1, 0, 1],
                                    [1, 1, 0],
                                    [1, 0, 0],
                                    [0, 1, 0],
                                    [0, 0, 1],
                                    [0, 0, 0]], dtype=NP_FLOAT)
    else:
        net_weights_arr = np.array([[1, 1, 1]], dtype=NP_FLOAT)

    for net_weights in net_weights_arr:
        weights = {
            'charge_weight': charge_weight_init,
            'net_weights': net_weights
        }
        for beta in betas:
            if run_logger is not None:
                run_dir, run_str = run_logger.reset(model.run_steps, beta,
                                                    **weights)
            t0 = time.time()
            runner.run(model.run_steps, beta, **weights)
            run_time = time.time() - t0
            io.log(80 * '-')
            io.log(f'Took: {run_time} s to complete run.')
            io.log(80 * '-')

            if plotter is not None and run_logger is not None:
                plotter.plot_observables(run_logger.run_data,
                                         beta, run_str, **weights)
                lf_plotter = LeapfrogPlotter(plotter.out_dir, run_logger)
                lf_plotter.make_plots(run_dir, num_samples=20)

    return sess, model, train_logger


def run_hmc(FLAGS, params=None, log_file=None):
    """Run generic HMC."""
    condition1 = not FLAGS.horovod
    condition2 = FLAGS.horovod and hvd.rank() == 0
    is_chief = condition1 or condition2
    io.log('\n' + 80 * '-')
    io.log(("Running generic HMC algorithm "
            "with learned parameters from L2HMC..."))
    if is_chief:
        hmc_sess, _, _, _ = hmc(FLAGS, params, log_file)
        hmc_sess.close()
        tf.reset_default_graph()


def run_l2hmc(FLAGS, log_file=None):
    """Train and run L2HMC algorithm."""
    io.log('\n' + 80 * '-')
    io.log("Running L2HMC algorithm...")
    l2hmc_sess, l2hmc_model, l2hmc_train_logger = l2hmc(FLAGS, log_file)
    l2hmc_sess.close()
    tf.reset_default_graph()

    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     io.log(80 * '-')
472     io.log(f'Args received from `parse_args()`:')
473     for key, val in args.__dict__.items():
474         io.log(f'{key}: {val}')
475     io.log(80 * '-')
476     import pickle
477     args_file = 'args.pkl'
```

```
    io.log(f'writing args to: {args_file}.')
    with open('args.pkl', 'wb') as f:
        pickle.dump(args.__dict__, f)
    io.log('done.')
    main(args)
```

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

```python
"""
gauge_model.py

Implements GaugeModel class responsible for building computation graph used in
tensorflow.

Author: Sam Foreman (github: @saforem2)
Date: 04/12/2019
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os

import numpy as np
import tensorflow as tf

try:
    import horovod.tensorflow as hvd
    HAS_HOROVOD = True
except ImportError:
    HAS_HOROVOD = False

import utils.file_io as io

from globals import GLOBAL_SEED, TF_FLOAT
from lattice.lattice import GaugeLattice
from dynamics.gauge_dynamics import GaugeDynamics


def check_log_dir(log_dir):
    """Check log_dir for existing checkpoints."""
    assert os.path.isdir(log_dir)
    ckpt_dir = os.path.join(log_dir, 'checkpoints')
    if os.path.isdir(ckpt_dir):
        pass


class GaugeModel:
    def __init__(self, params=None):
        # ------------------------------------------------------
        # Create attributes from (key, val) pairs in params
        # ------------------------------------------------------
        self.loss_weights = {}
        self.params = params
        for key, val in self.params.items():
            if 'weight' in key and key != 'charge_weight':
                self.loss_weights[key] = val
            else:
                setattr(self, key, val)

        io.log(80 * '-')
        io.log(f'Args received by `GaugeModel`:')
        for key, val in params.items():
            io.log(f'{key}: {val}')
        io.log(80 * '-')
```

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

```
        with tf.name_scope('lattice'):
            lattice = GaugeLattice(time_size=self.time_size,
                                   space_size=self.space_size,
                                   dim=self.dim,
                                   link_type=self.link_type,
                                   num_samples=self.num_samples,
                                   rand=self.rand)

            assert lattice.samples.shape[0] == self.num_samples

            samples = tf.convert_to_tensor(lattice.samples, dtype=TF_FLOAT)

        return lattice, samples

    def _create_observables(self):
        """Create operations for calculating lattice observables."""
        obs_ops = {}
        with tf.name_scope('plaq_observables'):
            with tf.name_scope('plaq_sums'):
                obs_ops['plaq_sums'] = self.lattice.calc_plaq_sums(self.x)

            with tf.name_scope('actions'):
                obs_ops['actions'] = self.lattice.calc_actions(self.x)

            with tf.name_scope('avg_plaqs'):
                plaqs = self.lattice.calc_plaqs(self.x)
                avg_plaqs = tf.reduce_mean(plaqs, name='avg_plaqs')

                obs_ops['plaqs'] = plaqs
                obs_ops['avg_plaqs'] = avg_plaqs

            with tf.name_scope('top_charges'):
                obs_ops['charges'] = self.lattice.calc_top_charges(self.x,
                                                                   fft=False)

        #  return plaq_sums_op, actions_op, plaqs_op, charges_op
        return obs_ops

    def _create_inputs(self):
        """Create input paceholders (if not executing eagerly).

        Returns:
            outputs: Dictionary with the following entries:
                x: Placeholder for input lattice configuration with
                    shape = (batch_size, x_dim) where x_dim is the number of
                    links on the lattice and is equal to lattice.time_size *
                    lattice.space_size * lattice.dim.
                beta: Placeholder for inverse coupling constant.
                charge_weight: Placeholder for the charge_weight (i.e. alpha_Q,
                    the multiplicative factor that scales the topological
                    charge term in the modified loss function) .
                net_weights: Array of placeholders, each of which is a
                    multiplicative constant used to scale the effects of the
                    various S, Q, and T functions from the original paper.
                    net_weights[0] = 'scale_weight', multiplies the S fn.
                    net_weights[1] = 'transformation_weight', multiplies the Q
                    fn.  net_weights[2] = 'translation_weight', multiplies the
                    T fn.
        """
        with tf.name_scope('inputs'):
```

```
178             if not tf.executing_eagerly():
179                 x = tf.placeholder(dtype=TF_FLOAT,
180                                    shape=(self.batch_size, self.x_dim),
181                                    name='x_placeholder')
182                 beta = tf.placeholder(dtype=TF_FLOAT,
183                                       shape=(),
184                                       name='beta')
185                 charge_weight = tf.placeholder(dtype=TF_FLOAT,
186                                                shape=(),
187                                                name='charge_weight')
188                 net_weights = [
189                     tf.placeholder(
190                         dtype=TF_FLOAT, shape=(), name='scale_weight'
191                     ),
192                     tf.placeholder(
193                         dtype=TF_FLOAT, shape=(), name='transformation_weight'
194                     ),
195                     tf.placeholder(
196                         dtype=TF_FLOAT, shape=(), name='translation_weight'
197                     )
198                 ]
199             else:
200                 x = tf.convert_to_tensor(
201                     self.lattice.samples.reshape((self.batch_size, self.x_dim))
202                 )
203                 beta = tf.convert_to_tensor(self.beta_init)
204                 charge_weight = tf.convert_to_tensor(0.)
205                 net_weights = tf.convert_to_tensor([1., 1., 1.])
206 
207         outputs = {
208             'x': x,
209             'beta': beta,
210             'charge_weight': charge_weight,
211             'net_weights': net_weights
212         }
213 
214         return outputs
215 
216     def _create_dynamics(self, lattice, samples, **kwargs):
217         """Initialize dynamics object."""
218         with tf.name_scope('dynamics'):
219             # default values of keyword arguments
220             dynamics_kwargs = {
221                 'eps': self.eps,
222                 'hmc': self.hmc,
223                 'network_arch': self.network_arch,
224                 'num_steps': self.num_steps,
225                 'eps_trainable': self.eps_trainable,
226                 'data_format': self.data_format,
227                 'use_bn': self.use_bn,
228                 'num_hidden': self.num_hidden,
229                 #  'scale_weight': self.scale_weight,
230                 #  'transformation_weight': self.transformation_weight,
231                 #  'translation_weight': self.translation_weight
232             }
233 
234             dynamics_kwargs.update(kwargs)
235             potential_fn = lattice.get_potential_fn(samples)
236             dynamics = GaugeDynamics(lattice=lattice,
237                                      potential_fn=potential_fn,
```

```
                                    **dynamics_kwargs)

        return dynamics, potential_fn

    @staticmethod
    def _create_metric_fn(metric):
        """Create metric fn for measuring the distance between two samples."""
        with tf.name_scope('metric_fn'):
            if metric == 'l1':
                def metric_fn(x1, x2):
                    return tf.abs(x1 - x2)

            elif metric == 'l2':
                def metric_fn(x1, x2):
                    return tf.square(x1 - x2)

            elif metric == 'cos':
                def metric_fn(x1, x2):
                    return tf.abs(tf.cos(x1) - tf.cos(x2))

            elif metric == 'cos2':
                def metric_fn(x1, x2):
                    return tf.abs(tf.cos(x1) - tf.cos(x2))

            elif metric == 'cos_diff':
                def metric_fn(x1, x2):
                    return 1. - tf.cos(x1 - x2)
            elif metric == 'tan_cos':
                def metric_fn(x1, x2):
                    cos_diff = 1. - tf.cos(x1 - x2)
                    return tf.tan(np.pi * cos_diff / 2)
            else:
                raise AttributeError(f"""metric={metric}. Expected one of:
                                     'l1', 'l2', 'cos', 'cos2', 'cos_diff', or
                                     'tan_cos'.""")

        return metric_fn

    def _create_optimizer(self, lr_init=None):
        """Create learning rate and optimizer."""
        if self.hmc:
            return

        if lr_init is None:
            lr_init = self.lr_init

        with tf.name_scope('global_step'):
            self.global_step = tf.train.get_or_create_global_step()
            #  self.global_step.assign(1)

        with tf.name_scope('learning_rate'):
            self.lr = tf.train.exponential_decay(lr_init,
                                                 self.global_step,
                                                 self.lr_decay_steps,
                                                 self.lr_decay_rate,
                                                 staircase=True,
                                                 name='learning_rate')
        with tf.name_scope('optimizer'):
            # Define update operations for batch normalization
            self.update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
```

```
            # Define optimizer
            with tf.control_dependencies(self.update_ops):
                self.optimizer = tf.train.AdamOptimizer(self.lr)
                if self.using_hvd:
                    self.optimizer = hvd.DistributedOptimizer(self.optimizer)

    def _calc_std_loss(self, config_init, config_proposed, accept_prob):
        """Calculate the (individual) standard contribution to the loss fn."""
        eps = 1e-4
        with tf.name_scope('_std_loss'):
            summed_diff = tf.reduce_sum(
                self.metric_fn(config_init, config_proposed), axis=1
            )

        return summed_diff * accept_prob + eps

    def calc_std_loss(self, inputs, **weights):
        """Calculate the standard contribution to the loss.

        Explicitly: This is calculated as the expectation value of the jump
        loss over both the target and initialization distributions.

        Args:
            x_tup: Tuple containing (x_init, x_proposed)
            z_tup: Tuple containing (z_init, z_proposed) (aux. variable)
            p_tup: Tuple containing (px, pz) acceptance probabilities.
            **weights: Dictionary of weights used as multiplicative scaling
                factors to control the contribution from individual terms to
                the total loss.
        Returns:
            std_loss: Tensorflow operation responsible for calculating the
                standard contribution to the loss function.
        """
        aux_weight = weights.get('aux_weight', 1.)
        std_weight = weights.get('std_weight', 1.)
        ls = self.loss_scale

        with tf.name_scope('x_std_loss'):
            x_std_loss = self._calc_std_loss(inputs['x_init'],
                                             inputs['x_proposed'],
                                             inputs['px'])
            x_loss = ls / x_std_loss - x_std_loss / ls
            tf.add_to_collection('losses', x_std_loss)
            tf.add_to_collection('losses', x_loss)

        with tf.name_scope('z_std_loss'):
            z_std_loss = self._calc_std_loss(inputs['z_init'],
                                             inputs['z_proposed'],
                                             inputs['pz'])
            z_loss = ls / z_std_loss - z_std_loss / ls
            tf.add_to_collection('losses', z_std_loss)
            tf.add_to_collection('losses', z_loss)

        with tf.name_scope('std_loss'):
            std_loss = tf.reduce_mean(
                std_weight * (x_loss + aux_weight * z_loss),
                axis=0, name='std_loss'
            )
            tf.add_to_collection('losses', std_loss)
```

```
        return std_loss

    def _calc_charge_loss(self, config_init, config_proposed, accept_prob):
        """Calculate the (individual) top. charge contribution to the loss fn

        Args:
            config_init: Initial configuration.
            config_proposed: Proposed configuration.
            accept_prob: Likelihood of accepting the proposed configuration,
                usd to calculate the expectation value of the topological
                charge difference.
        Returns:
            charge_loss: Individual contribution to the loss function from the
                topological charge difference between the initial and proposed
                configurations.
        """
        with tf.name_scope('charge_diff'):
            charge_diff = self.lattice.calc_top_charges_diff(config_init,
                                                             config_proposed,
                                                             fft=True)
        return accept_prob * charge_diff

    def calc_charge_loss(self, inputs, **weights):
        """Calculate the (individual) top. charge contribution to the loss fn

        Calculate the difference in topological charge between the initial
        and proposed configurations multiplied by the probability of
        acceptance to get the expected value of the difference in top. charge.

        Args:
            inputs: Dictionary containing initial and proposed configurations
                sampled from both the target ('x_init', 'x_proposed') and
                initialization ('z_init', 'z_proposed') distributions, as well
                as the calculated acceptance probabilities ('px', and 'pz').
            weights: Dictionary containing various multiplicative weights used
                to scale the contribution from individual terms to the total
                loss function
        Returns:
            charge_loss: Total contribution to the loss function from the
                topological charge difference between initial and proposed
                configuratons.
        """
        aux_weight = weights.get('aux_weight', 1.)
        with tf.name_scope('x_charge_loss'):
            x_charge_loss = self._calc_charge_loss(inputs['x_init'],
                                                   inputs['x_proposed'],
                                                   inputs['px'])
            tf.add_to_collection('losses', x_charge_loss)

        with tf.name_scope('z_charge_loss'):
            z_charge_loss = self._calc_charge_loss(inputs['z_init'],
                                                   inputs['z_proposed'],
                                                   inputs['pz'])
            tf.add_to_collection('losses', x_charge_loss)

        with tf.name_scope('total_charge_loss'):
            charge_loss = self.charge_weight * (x_charge_loss
                                                + aux_weight * z_charge_loss)
            charge_loss = tf.reduce_mean(charge_loss, axis=0,
                                         name='charge_loss')
```

```
418
419             tf.add_to_collection('losses', charge_loss)
420
421         return charge_loss
422
423     def calc_loss(self, x, beta, net_weights, **weights):
424         """Create operation for calculating the loss.
425
426         Args:
427             x: Input tensor of shape (self.num_samples,
428                 self.lattice.num_links) containing batch of GaugeLattice links
429                 variables.
430             beta (float): Inverse coupling strength.
431             net_weights: Tuple containing multiplicative weights that scale the
432                 contributions from the scale (S), transformation (Q) and
433                 translation (T) functions.
434             weights: Dictionary containing various multiplicative weights used
435                 to scale the contribution from individual terms to the total
436                 loss function
437
438         Returns:
439             loss (float): Operation responsible for calculating the total loss.
440             px (np.ndarray): Array of acceptance probabilities from
441                 Metropolis-Hastings accept/reject step. Has shape:
442                 (self.num_samples,)
443             x_out: Output samples obtained after Metropolis-Hastings
444                 accept/reject step.
445
446         NOTE: If proposed configuration is accepted following
447             Metropolis-Hastings accept/reject step, x_proposed and x_out are
448             equivalent.
449         """
450         with tf.name_scope('x_update'):
451             x_dynamics_output = self.dynamics(x, beta, net_weights,
452                                               while_loop=self.while_loop,
453                                               v_in=None,  # v_in
454                                               save_lf=self.save_lf)
455             x_proposed = x_dynamics_output['x_proposed']
456             px = x_dynamics_output['accept_prob']
457             x_out = x_dynamics_output['x_out']
458
459         # Auxiliary variable
460         with tf.name_scope('z_update'):
461             z = tf.random_normal(tf.shape(x), seed=GLOBAL_SEED, name='z')
462             z_dynamics_output = self.dynamics(z, beta, net_weights,
463                                               while_loop=self.while_loop,
464                                               v_in=None, save_lf=False)
465             z_proposed = z_dynamics_output['x_proposed']
466             pz = z_dynamics_output['accept_prob']
467
468         with tf.name_scope('top_charge_diff'):
469             x_dq = tf.cast(
470                 self.lattice.calc_top_charges_diff(x, x_out, fft=False),
471                 dtype=tf.int32
472             )
473
474         # NOTE:
475         #  std:_loss: 'standard' loss
476         #  charge_loss: Contribution from the difference in topological charge
477         #    betweween the initial and proposed configurations  to the total
```

```
        #     loss.
        inputs = {
            'x_init': x,
            'x_proposed': x_proposed,
            'px': px,
            'z_init': z,
            'z_proposed': z_proposed,
            'pz': pz
        }
        with tf.name_scope('calc_loss'):
            with tf.name_scope('std_loss'):
                std_loss = self.calc_std_loss(inputs, **weights)
            with tf.name_scope('charge_loss'):
                charge_loss = self.calc_charge_loss(inputs, **weights)

            total_loss = tf.add(std_loss, charge_loss, name='total_loss')
            tf.add_to_collection('losses', total_loss)

        return total_loss, x_dq, x_dynamics_output

    def calc_loss_and_grads(self, x, beta, net_weights, **weights):
        """Calculate loss its gradient with respect to all trainable variables.

        Args:
            x: Placeholder (tensor object)representing batch of GaugeLattice
                link variables.
            beta: Placeholder (tensor object) representing inverse coupling
            strength.

        Returns:
            loss: Operation for calculating the total loss.
            grads: Tensor containing the gradient of the loss function with
                respect to all trainable variables.
            x_out: Operation for obtaining new samples (i.e. output of
                augmented L2HMC algorithm.)
            accept_prob: Operation for calculating acceptance probabilities
                used in Metropolis-Hastings accept reject.
            x_dq: Operation for calculating the topological charge difference
                between the initial and proposed configurations.
        """
        # TODO: Fix eager execution logic to deal with self.lf_out
        with tf.name_scope('grads'):
            if tf.executing_eagerly():
                with tf.GradientTape() as tape:
                    loss, x_dq, dynamics_output = self.calc_loss(
                        x, beta, net_weights, **weights
                    )
                grads = tape.gradient(loss, self.dynamics.trainable_variables)
            else:
                loss, x_dq, dynamics_output = self.calc_loss(x, beta,
                                                             net_weights,
                                                             **weights)
                with tf.name_scope('grads'):
                    grads = tf.gradients(loss,
                                         self.dynamics.trainable_variables)
                    if self.clip_grads:
                        grads, _ = tf.clip_by_global_norm(grads,
                                                          self.clip_value)

        return loss, grads, x_dq, dynamics_output
```

```
538
539     def create_sampler(self):
540         """Create operation for generating new samples using dynamics engine.
541
542         NOTE: This method is to be used when running generic HMC to create
543         operations for generating new sampler.
544         """
545         with tf.name_scope('sampler'):
546             output = self.dynamics(self.x, self.beta,
547                                    save_lf=True, train=False)
548             self.x_out = output['x_out']
549             self.px = output['accept_prob']
550             if self.save_lf:
551                 self.lf_out_f = output['lf_out_f']
552                 self.pxs_out_f = output['accept_probs_f']
553                 self.lf_out_b = output['lf_out_b']
554                 self.pxs_out_b = output['accept_probs_b']
555                 self.masks_f = output['forward_mask']
556                 self.masks_b = output['backward_mask']
557                 self.logdets_f = output['logdets_f']
558                 self.logdets_b = output['logdets_b']
559                 self.sumlogdet_f = output['sumlogdet_f']
560                 self.sumlogdet_b = output['sumlogdet_b']
561
562     def build(self):
563         """Build Tensorflow graph."""
564         with tf.name_scope('output'):
565             #  self.loss_op, self.grads, self.x_out, self.px, x_dq = output
566             loss, grads, x_dq, dynamics_output = self.calc_loss_and_grads(
567                 x=self.x, beta=self.beta,
568                 net_weights=self.net_weights,
569                 **self.loss_weights
570             )
571             #  self.loss_op = outputs[0]
572             #  self.grads = outputs[1]
573             self.loss_op = loss
574             self.grads = grads
575             self.x_out = dynamics_output['x_out']
576             self.px = dynamics_output['accept_prob']
577             self.charge_diffs_op = tf.reduce_sum(x_dq) / self.num_samples
578             if self.save_lf:
579                 self.lf_out_f = dynamics_output['lf_out_f']
580                 self.lf_out_b = dynamics_output['lf_out_b']
581                 self.pxs_out_f = dynamics_output['accept_probs_f']
582                 self.pxs_out_b = dynamics_output['accept_probs_b']
583                 self.masks_f = dynamics_output['forward_mask']
584                 self.masks_b = dynamics_output['backward_mask']
585                 self.logdets_f = dynamics_output['logdets_f']
586                 self.logdets_b = dynamics_output['logdets_b']
587                 self.sumlogdet_f = dynamics_output['sumlogdet_f']
588                 self.sumlogdet_b = dynamics_output['sumlogdet_b']
589
590         with tf.name_scope('train'):
591             # ------------------------------------------------------
592             #  TODO:
593             #
594             #  update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
595             #  ! no update ops in the default graph
596             #  io.log("update_ops: ", update_ops)
597             #  Use the update ops of the model itself
```

```
598             #  io.log("model.updates: ", self.dynamics.updates)
599             # ------------------------------------------------------
600             grads_and_vars = zip(self.grads, self.dynamics.trainable_variables)
601             with tf.control_dependencies(self.dynamics.updates):
602                 self.train_op = self.optimizer.apply_gradients(
603                     grads_and_vars,
604                     global_step=self.global_step,
605                     name='train_op'
606                 )
```

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

```python
"""
Dynamics engine for L2HMC sampler on Lattice Gauge Models.

Reference [Generalizing Hamiltonian Monte Carlo with Neural
Networks](https://arxiv.org/pdf/1711.09268.pdf)

Code adapted from the released TensorFlow graph implementation by original
authors https://github.com/brain-research/l2hmc.

Author: Sam Foreman (github: @saforem2)
Date: 1/14/2019
"""
import numpy as np
import numpy.random as npr
import tensorflow as tf

import utils.file_io as io

from globals import GLOBAL_SEED, TF_FLOAT
from network.conv_net3d import ConvNet3D
from network.conv_net2d import ConvNet2D
from network.generic_net import GenericNet


def exp(x, name=None):
    """Safe exponential using tf.check_numerics."""
    return tf.check_numerics(tf.exp(x), f'{name} is NaN')


def flatten_tensor(tensor):
    """Flattens tensor along axes 1:, since axis=0 indexes sample in batch.

    Example: for a tensor of shape [b, x, y, t] -->
        returns a tensor of shape [b, x * y * t]
    """
    batch_size = tensor.shape[0]
    return tf.reshape(tensor, shape=(batch_size, -1))


class GaugeDynamics(tf.keras.Model):
    """Dynamics engine of naive L2HMC sampler."""

    def __init__(self, lattice, potential_fn, **kwargs):
        """Initialization.

        Args:
            lattice: Lattice object containing multiple sample lattices.
            potential_fn: Function specifying minus log-likelihood objective
            to minimize.

        NOTE: kwargs (expected)
            num_steps: Number of leapfrog steps to use in integrator.
            eps: Initial step size to use in leapfrog integrator.
            network_arch: String specifying network architecture to use.
                Must be one of `'conv2D', 'conv3D', 'generic'`. Networks
                are defined in `../network/`
            hmc: Flag indicating whether generic HMC (no augmented
```

```
            leapfrog) should be used instead of L2HMC. Defaults to
            False.
        eps_trainable: Flag indiciating whether the step size (eps)
            should be trainable. Defaults to True.
        np_seed: Seed to use for numpy.random.
    """
    super(GaugeDynamics, self).__init__(name='GaugeDynamics')
    npr.seed(GLOBAL_SEED)

    self.lattice = lattice
    self.potential = potential_fn
    self.batch_size = self.lattice.samples.shape[0]
    self.x_dim = self.lattice.num_links

    # create attributes from kwargs.items()
    for key, val in kwargs.items():
        if key != 'eps':  # want to use self.eps as tf.Variable
            setattr(self, key, val)

    io.log(80 * '-')
    io.log(f'Args received by `GaugeDynamics`:')
    for key, val in kwargs.items():
        io.log(f'{key}: {val}')
    io.log(80 * '-')
    io.log(f'network_arch: {self.network_arch}')

    if self.num_hidden is None:
        self.num_hidden = 2 * self.lattice.num_links

    with tf.name_scope('eps'):
        #  self.eps = exp(self.alpha, name='eps')
        self.eps = tf.Variable(
            initial_value=kwargs.get('eps', 0.4),
            name='eps',
            dtype=TF_FLOAT,
            trainable=self.eps_trainable
        )

    self._construct_masks()

    if self.hmc:
        self.x_fn = lambda inp: [
            tf.zeros_like(inp[0]) for t in range(3)
        ]
        self.v_fn = lambda inp: [
            tf.zeros_like(inp[0]) for t in range(3)
        ]

    else:
        if self.network_arch.upper() == 'CONV3D':  # should be 'conv3D'
            self._build_conv_nets_3D()
        if self.network_arch.upper() == 'CONV2D':  # should be 'conv2D'
            self._build_conv_nets_2D()
        else:
            self._build_generic_nets()

def _build_conv_nets_3D(self):
    """Build ConvNet3D architecture for x and v functions."""
    kwargs = {
        '_input_shape': (self.batch_size, *self.lattice.links.shape),
```

```
            'links_shape': self.lattice.links.shape,
            'x_dim': self.lattice.num_links,  # dimensionality of target space
            'factor': 2.,  # scale factor used in original paper
            'spatial_size': self.lattice.space_size,  # spatial size of lattice
            'num_hidden': self.num_hidden,  # num hidden nodes
            'num_filters': int(self.lattice.space_size),  # num conv. filters
            'filter_sizes': [(3, 3, 2), (2, 2, 2)],  # size of conv. filters
            'name_scope': 'position',  # namespace in which to create network
            'data_format': self.data_format,  # channels_first if using GPU
            'use_bn': self.use_bn,  # whether or not to use batch normalization
        }

        with tf.name_scope("DynamicsNetwork"):
            with tf.name_scope("XNet"):
                self.x_fn = ConvNet3D(model_name='XNet', **kwargs)

            kwargs['name_scope'] = 'momentum'  # update name scope
            kwargs['factor'] = 1.              # factor used in orig. paper
            with tf.name_scope("VNet"):
                self.v_fn = ConvNet3D(model_name='VNet', **kwargs)

    def _build_conv_nets_2D(self):
        """Build ConvNet architecture for x and v functions."""
        kwargs = {
            '_input_shape': (self.batch_size, *self.lattice.links.shape),
            'links_shape': self.lattice.links.shape,
            'x_dim': self.lattice.num_links,  # dimensionality of target space
            'factor': 2.,  # scale factor used in original paper
            'spatial_size': self.lattice.space_size,  # spatial size of lattice
            'num_hidden': self.num_hidden,  # num hidden nodes
            'num_filters': int(2 * self.lattice.space_size),  # num filters
            'filter_sizes': [(2, 2), (2, 2)],  # for 1st and 2nd conv. layer
            'name_scope': 'position',  # namespace in which to create network
            'data_format': self.data_format,  # channels_first if using GPU
            'use_bn': self.use_bn,  # whether or not to use batch normalization
        }

        with tf.name_scope("DynamicsNetwork"):
            with tf.name_scope("XNet"):
                self.x_fn = ConvNet2D(model_name='XNet', **kwargs)

            kwargs['name_scope'] = 'momentum'
            kwargs['factor'] = 1.
            with tf.name_scope("VNet"):
                self.v_fn = ConvNet2D(model_name='VNet', **kwargs)

    def _build_generic_nets(self):
        """Build GenericNet FC-architectures for x and v fns. """

        kwargs = {
            '_input_shape': (self.batch_size, *self.lattice.links.shape),
            'x_dim': self.lattice.num_links,  # dimensionality of target space
            'factor': 2.,  # scale factor used in original paper
            'num_hidden': self.num_hidden,  # num hidden nodes
            'name_scope': 'position',  # namespace in which to create network
            'use_bn': self.use_bn
        }

        with tf.name_scope("DynamicsNetwork"):
            with tf.name_scope("XNet"):
```

```
                self.x_fn = GenericNet(model_name='XNet', **kwargs)

            kwargs['factor'] = 1.
            kwargs['name_scope'] = 'momentum'
            with tf.name_scope("VNet"):
                self.v_fn = GenericNet(model_name='VNet', **kwargs)

    def call(self, x_in, beta, net_weights,
             while_loop=True, v_in=None, save_lf=False):
        """Call method."""
        return self.apply_transition(x_in, beta, net_weights,
                                     while_loop=while_loop,
                                     v_in=v_in, save_lf=save_lf)

    def apply_transition(self, x_in, beta, net_weights,
                         while_loop=True, v_in=None, save_lf=False):
        """Propose a new state and perform the accept/reject step.

        Args:
            x: Batch of (x) samples (batch of links).
            beta (float): Inverse coupling constant.

        Returns:
            x_proposed: Proposed x before accept/reject step.
            v_proposed: Proposed v before accept/reject step.
            accept_prob: Probability of accepting the proposed states.
            x_out: Samples after accept/reject step.
        """
        # Simulate dynamics both forward and backward
        # Use sampled masks to compute the actual solutions
        with tf.name_scope('apply_transition'):
            with tf.name_scope('transition_forward'):
                outputs_f = self.transition_kernel(x_in, beta,
                                                   net_weights,
                                                   while_loop=while_loop,
                                                   forward=True,
                                                   v_in=v_in,
                                                   save_lf=save_lf)
                xf = outputs_f['x_proposed']
                vf = outputs_f['v_proposed']
                accept_prob_f = outputs_f['accept_prob']

                if save_lf:
                    lf_out_f = outputs_f['lf_out']
                    logdets_f = outputs_f['logdets']
                    sumlogdet_f = outputs_f['sumlogdet']

            with tf.name_scope('transition_backward'):
                outputs_b = self.transition_kernel(x_in, beta,
                                                   net_weights,
                                                   while_loop=while_loop,
                                                   forward=False,
                                                   v_in=v_in,
                                                   save_lf=save_lf)
                xb = outputs_b['x_proposed']
                vb = outputs_b['v_proposed']
                accept_prob_b = outputs_b['accept_prob']

                if save_lf:
                    lf_out_b = outputs_b['lf_out']
```

```
                logdets_b = outputs_b['logdets']
                sumlogdet_b = outputs_b['sumlogdet']

            # Decide direction uniformly
            with tf.name_scope('transition_masks'):
                forward_mask = tf.cast(
                    tf.random_uniform((self.batch_size,),
                                      seed=GLOBAL_SEED) > 0.5,
                    TF_FLOAT,
                    name='forward_mask'
                )
                backward_mask = 1. - forward_mask

            # Obtain proposed states
            with tf.name_scope('x_proposed'):
                x_proposed = (forward_mask[:, None] * xf
                              + backward_mask[:, None] * xb)

            with tf.name_scope('v_proposed'):
                v_proposed = (forward_mask[:, None] * vf
                              + backward_mask[:, None] * vb)

            # Probability of accepting the proposed states
            with tf.name_scope('accept_prob'):
                accept_prob = (forward_mask * accept_prob_f
                               + backward_mask * accept_prob_b)

            # Accept or reject step
            with tf.name_scope('accept_mask'):
                accept_mask = tf.cast(
                    accept_prob > tf.random_uniform(tf.shape(accept_prob),
                                                    seed=GLOBAL_SEED),
                    TF_FLOAT,
                    name='acccept_mask'
                )
                reject_mask = 1. - accept_mask

            # Samples after accept / reject step
            with tf.name_scope('x_out'):
                x_out = (accept_mask[:, None] * x_proposed
                         + reject_mask[:, None] * x_in)

        outputs = {
            'x_proposed': x_proposed,
            'v_proposed': v_proposed,
            'accept_prob': accept_prob,
            'x_out': x_out
        }

        if save_lf:
            outputs['lf_out_f'] = lf_out_f
            outputs['accept_probs_f'] = accept_prob_f
            outputs['lf_out_b'] = lf_out_b
            outputs['accept_probs_b'] = accept_prob_b
            outputs['forward_mask'] = forward_mask
            outputs['backward_mask'] = backward_mask
            outputs['logdets_f'] = logdets_f
            outputs['logdets_b'] = logdets_b
            outputs['sumlogdet_f'] = sumlogdet_f
            outputs['sumlogdet_b'] = sumlogdet_b
```

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

```
            'v_proposed': v_proposed,
            'sumlogdet': sumlogdet
        }

        if save_lf:
            outputs['lf_out'] = lf_out
            outputs['logdets'] = logdets_out

        return outputs

    def _transition_while_loop(self, x_in, v_in, beta, net_weights,
                               forward=True, save_lf=False):
        """Implements the transition kernel using a `tf.while_loop`."""
        lf_fn = self._forward_lf if forward else self._backward_lf

        with tf.name_scope('while_loop_init'):
            x_proposed, v_proposed = x_in, v_in
            #  t = tf.constant(0., name='md_time', dtype=TF_FLOAT)
            step = tf.constant(0., name='md_step', dtype=TF_FLOAT)
            batch_size = tf.shape(x_in)[0]
            #  assert batch_size == self.batch_size
            logdet = tf.zeros((batch_size,))
            lf_out = tf.TensorArray(dtype=TF_FLOAT, size=self.num_steps + 1,
                                    dynamic_size=True, name='lf_out',
                                    clear_after_read=False)
            logdets_out = tf.TensorArray(dtype=TF_FLOAT, size=self.num_steps,
                                         dynamic_size=True, name='logdets_out',
                                         clear_after_read=False)
            lf_out.write(0, x_in)

        def body(lf_samples, v, beta, step, logdet, logdets):
            i = tf.cast(step, dtype=tf.int32, name='lf_step')
            x_in = lf_samples

        #  def body(x, v, beta, step, logdet, lf_samples, logdets):
        #      i = tf.cast(step, dtype=tf.int32, name='lf_step')  # cast as int
        #      with tf.name_scope('apply_lf'):
        #          new_x, new_v, j = lf_fn(x, v, beta, step, net_weights)
        #      with tf.name_scope('concat_lf_outputs'):
        #          lf_samples = (lf_samples.write(i, new_x)
        #                        if tf.greater(i, 0) else
        #                        lf_samples.write(i, x))
        #      with tf.name_scope('concat_logdets'):
        #          logdets = logdets.write(i, logdet+j)
        #      return (new_x, new_v, beta, step + 1,
        #              logdet + j, lf_samples, logdets)

        def cond(x, v, beta, step, logdet, lf_out, logdets):
            with tf.name_scope('check_lf_step'):
                return tf.less(step, self.num_steps)

        with tf.name_scope('while_loop'):
            outputs = tf.while_loop(
                cond=cond,
                body=body,
                loop_vars=[x_proposed, v_proposed,
                           beta, step, logdet,
                           lf_out, logdets_out]
            )
```

```
            x_proposed = outputs[0]
            v_proposed = outputs[1]
            beta = outputs[2]
            step = outputs[3]
            sumlogdet = outputs[4]
            lf_out = outputs[5].stack()
            logdets_out = outputs[6].stack()

        outputs = {
            'x_proposed': x_proposed,
            'v_proposed': v_proposed,
            'sumlogdet': sumlogdet
        }

        if save_lf:
            outputs['lf_out'] = lf_out
            outputs['logdets'] = logdets_out

        return outputs

    def _forward_lf(self, x, v, beta, step, net_weights):
        """One forward augmented leapfrog step."""
        with tf.name_scope('forward_lf'):
            with tf.name_scope('get_time'):
                t = self._get_time(step, tile=tf.shape(x)[0])

            if t.dtype != TF_FLOAT:
                t = tf.cast(t, dtype=TF_FLOAT)

            with tf.name_scope('get_mask'):
                mask, mask_inv = self._get_mask(step)

            with tf.name_scope('augmented_leapfrog'):
                sumlogdet = 0.

                v, logdet = self._update_v_forward(x, v, beta, t, net_weights)
                sumlogdet += logdet

                x, logdet = self._update_x_forward(x, v, t, net_weights,
                                                   mask, mask_inv)
                sumlogdet += logdet

                x, logdet = self._update_x_forward(x, v, t, net_weights,
                                                   mask_inv, mask)
                sumlogdet += logdet

                v, logdet = self._update_v_forward(x, v, beta, t, net_weights)
                sumlogdet += logdet

        return x, v, sumlogdet

    def _backward_lf(self, x, v, beta, step, net_weights):
        """One backward augmented leapfrog step."""
        # Reversed index/sinusoidal time
        with tf.name_scope('backward_lf'):
            with tf.name_scope('get_time'):
                t = self._get_time(self.num_steps - step - 1,
                                   tile=tf.shape(x)[0])

            if t.dtype != TF_FLOAT:
```

```
                t = tf.cast(t, dtype=TF_FLOAT)

            with tf.name_scope('get_mask'):
                mask, mask_inv = self._get_mask(self.num_steps - step - 1)

            with tf.name_scope('augmented_leapfrog'):
                sumlogdet = 0.

                v, logdet = self._update_v_backward(x, v, beta, t, net_weights)
                sumlogdet += logdet

                x, logdet = self._update_x_backward(x, v, t, net_weights,
                                                    mask_inv, mask)
                sumlogdet += logdet

                x, logdet = self._update_x_backward(x, v, t, net_weights,
                                                    mask, mask_inv)
                sumlogdet += logdet

                v, logdet = self._update_v_backward(x, v, beta, t, net_weights)
                sumlogdet += logdet

        return x, v, sumlogdet

    def _update_v_forward(self, x, v, beta, t, net_weights):
        """Update v in the forward leapfrog step.

        Args:
            x: input position tensor
            v: input momentum tensor
            beta: inverse coupling constant
            t: current leapfrog step
            net_weights: Placeholders for the multiplicative weights by which
                to multiply the S, Q, and T functions (scale, transformation,
                translation resp.)
        Returns:
            v: Updated (output) momentum
            logdet: Jacobian factor
        """
        with tf.name_scope('update_v_forward'):
            with tf.name_scope('grad_potential'):
                grad = self.grad_potential(x, beta)

            with tf.name_scope('v_fn'):
                # Sv: scale, Qv: transformation, Tv: translation
                scale, translation, transformation = self.v_fn(
                    [x, grad, t]
                )

            with tf.name_scope('scale'):
                scale *= 0.5 * self.eps * net_weights[0]
                scale_exp = exp(scale, 'vf_scale')

            with tf.name_scope('transformation'):
                transformation *= self.eps * net_weights[1]
                transformation_exp = exp(transformation, 'vf_transformation')

            with tf.name_scope('translation'):
                translation *= net_weights[2]
```

```
            with tf.name_scope('v_update'):
                v = (v * scale_exp - 0.5 * self.eps
                     * (grad * transformation_exp + translation))

        return v, tf.reduce_sum(scale, axis=1, name='vf_logdet')

    def _update_x_forward(self, x, v, t, net_weights, mask, mask_inv):
        """Update x in the forward leapfrog step."""
        with tf.name_scope('update_x_forward'):
            with tf.name_scope('x_fn'):
                scale, translation, transformation = self.x_fn(
                    [v, mask * x, t]
                )

            with tf.name_scope('scale'):
                scale *= self.eps * net_weights[0]
                scale_exp = exp(scale, 'xf_scale')

            with tf.name_scope('transformation'):
                transformation *= self.eps * net_weights[1]
                transformation_exp = exp(transformation, 'xf_transformation')

            with tf.name_scope('translation'):
                translation *= net_weights[2]

            with tf.name_scope('x_update'):
                x = (mask * x
                     + mask_inv * (x * scale_exp + self.eps
                                   * (v * transformation_exp + translation)))

        return x, tf.reduce_sum(mask_inv * scale, axis=1, name='xf_logdet')

    def _update_v_backward(self, x, v, beta, t, net_weights):
        """Update v in the backward leapfrog step. Invert the forward update"""
        with tf.name_scope('update_v_backward'):
            with tf.name_scope('grad_potential'):
                grad = self.grad_potential(x, beta)

            with tf.name_scope('v_fn'):
                scale, translation, transformation = self.v_fn(
                    [x, grad, t]
                )

            with tf.name_scope('scale'):
                scale *= -0.5 * self.eps * net_weights[0]
                scale_exp = exp(scale, 'vb_scale')

            with tf.name_scope('transformation'):
                transformation *= self.eps * net_weights[1]
                transformation_exp = exp(transformation, 'vb_transformation')

            with tf.name_scope('translation'):
                translation *= net_weights[2]

            with tf.name_scope('v_update'):
                v = (scale_exp * (v + 0.5 * self.eps
                                  * (grad * transformation_exp + translation)))

        return v, tf.reduce_sum(scale, axis=1, name='vb_logdet')
```

```
    def _update_x_backward(self, x, v, t, net_weights, mask, mask_inv):
        """Update x in the backward lf step. Inverting the forward update."""
        with tf.name_scope('update_x_backward'):
            with tf.name_scope('x_fn'):
                scale, translation, transformation = self.x_fn(
                    [v, mask * x, t]
                )

            with tf.name_scope('scale'):
                scale *= -self.eps * net_weights[0]
                scale_exp = exp(scale, 'xb_scale')

            with tf.name_scope('transformation'):
                transformation *= self.eps * net_weights[1]
                transformation_exp = exp(transformation, 'xb_transformation')

            with tf.name_scope('translation'):
                translation *= net_weights[2]

            with tf.name_scope('x_update'):
                x = (mask * x + mask_inv * scale_exp
                     * (x - self.eps * (v * transformation_exp + translation)))

        return x, tf.reduce_sum(mask_inv * scale, axis=1, name='xb_logdet')

    def _compute_accept_prob(self, xi, vi, xf, vf, sumlogdet, beta):
        """Compute the prob of accepting the proposed state given old state.
        Args:
            xi: Initial state.
            vi: Initial v.
            xf: Proposed state.
            vf: Proposed v.
            sumlogdet: Sum of the terms of the log of the determinant.
                (Eq. 14 of original paper).
            beta: Inverse coupling constant of gauge model.
        """
        with tf.name_scope('compute_accept_prob'):
            with tf.name_scope('old_hamiltonian'):
                old_hamil = self.hamiltonian(xi, vi, beta)
            with tf.name_scope('new_hamiltonian'):
                new_hamil = self.hamiltonian(xf, vf, beta)

            with tf.name_scope('prob'):
                prob = exp(tf.minimum(
                    (old_hamil - new_hamil + sumlogdet), 0.
                ), 'accept_prob')

        # Ensure numerical stability as well as correct gradients
        return tf.where(tf.is_finite(prob), prob, tf.zeros_like(prob))

    def _get_time(self, step, tile=1):
        """Format time as [cos(..), sin(...)]."""
        with tf.name_scope('get_time'):
            trig_t = tf.squeeze([
                tf.cos(2 * np.pi * step / self.num_steps),
                tf.sin(2 * np.pi * step / self.num_steps),
            ], name='md_time', )

        return tf.tile(tf.expand_dims(trig_t, 0), (tile, 1))
```

```
    def _construct_masks(self):
        """Construct different binary masks for different time steps."""
        self.masks = []
        for _ in range(self.num_steps):
            # Need to use npr here because tf would generate different random
            # values across different `sess.run`
            idx = npr.permutation(np.arange(self.x_dim))[:self.x_dim // 2]
            mask = np.zeros((self.x_dim,))
            mask[idx] = 1.
            mask = tf.constant(mask, dtype=TF_FLOAT)
            self.masks.append(mask[None, :])

    def _get_mask(self, step):
        """Get the mask, and its complement, 1. - mask at a given step."""
        if step.dtype != tf.int32:
            step = tf.cast(step, dtype=tf.int32)

        with tf.name_scope('get_mask'):
            if tf.executing_eagerly():
                m = self.masks[step]
            else:
                m = tf.gather(self.masks, step, name='gather_mask')
        return m, 1. - m

    def potential_energy(self, x, beta):
        """Compute potential energy using `self.potential` and beta."""
        with tf.name_scope('potential_energy'):
            potential_energy = tf.multiply(beta, self.potential(x),
                                           name='potential_energy')

        return potential_energy

    def kinetic_energy(self, v):
        """Compute the kinetic energy."""
        with tf.name_scope('kinetic_energy'):
            kinetic_energy = 0.5 * tf.reduce_sum(v**2, axis=1)

        return kinetic_energy

    def hamiltonian(self, x, v, beta):
        """Compute the overall Hamiltonian."""
        with tf.name_scope('hamiltonian'):
            with tf.name_scope('potential'):
                potential = self.potential_energy(x, beta)
            with tf.name_scope('kinetic'):
                kinetic = self.kinetic_energy(v)
            with tf.name_scope('hamiltonian'):
                hamiltonian = potential + kinetic

        return hamiltonian

    def grad_potential(self, x, beta):
        """Get gradient of potential function at current location."""
        with tf.name_scope('grad_potential'):
            if tf.executing_eagerly():
                x_tensor = tf.constant(x)
                beta_tensor = tf.constant(beta)
                with tf.GradientTape() as tape:
                    tape.watch(x_tensor)
                    potential_energy = self.potential_energy(x_tensor,
```

```
                                                        beta_tensor)
                    grad = tape.gradient(potential_energy, x_tensor)

            else:
                grad = tf.gradients(self.potential_energy(x, beta), x)[0]

        return grad
```

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

```python
"""
lattice.py

Contains implementation of GaugeLattice class.

Author: Sam Foreman (github: @saforem2)
Date: 01/15/2019
"""
import os
import random
import pickle

import numpy as np
import tensorflow as tf
from functools import reduce
from scipy.linalg import expm
from scipy.special import i0, i1
from globals import TF_FLOAT, NP_FLOAT


def u1_plaq_exact(beta):
    """Computes the expected value of the `average` plaquette for U(1)."""
    return i1(beta) / i0(beta)


def pbc(tup, shape):
    """Returns tup % shape for implementing periodic boundary conditions."""
    return list(np.mod(tup, shape))


def pbc_tf(tup, shape):
    """Tensorflow implementation of `pbc` defined above."""
    return list(tf.mod(tup, shape))


def mat_adj(mat):
    """Returns the adjoint (i.e. conjugate transpose) of a matrix `mat`."""
    return tf.transpose(tf.conj(mat))  # conjugate transpose


def project_angle(x):
    """Returns the projection of an angle `x` from [-4pi, 4pi] to [-pi, pi]."""
    return x - 2 * np.pi * tf.math.floor((x + np.pi) / (2 * np.pi))


def project_angle_fft(x, N=10):
    """Use the fourier series representation `x` to approx `project_angle`.
    NOTE: Because `project_angle` suffers a discontinuity, we approximate `x`
    with its Fourier series representation in order to have a differentiable
    function when computing the loss.
    Args:
        x (array-like): Array to be projected.
        N (int): Number of terms to keep in Fourier series.
    """
    y = np.zeros(x.shape, dtype=NP_FLOAT)
    for n in range(1, N):
        y += (-2 / n) * ((-1) ** n) * tf.sin(n * x)
```

```
    return y


class GaugeLattice(object):
    """Lattice with Gauge field existing on links."""

    def __init__(self,
                 time_size,
                 space_size,
                 dim=2,
                 link_type='U1',
                 num_samples=None,
                 rand=False):
        """Initialization for GaugeLattice object.

        Args:
            time_size (int): Temporal extent of lattice.
            space_size (int): Spatial extent of lattice.
            dim (int): Dimensionality
            link_type (str):
                String representing the type of gauge group for the link
                variables. Must be either 'U1', 'SU2', or 'SU3'
        """
        assert link_type.upper() in ['U1', 'SU2', 'SU3'], (
            "Invalid link_type. Possible values: U1', 'SU2', 'SU3'"
        )

        self.time_size = time_size
        self.space_size = space_size
        self.dim = dim
        self.link_type = link_type
        self.link_shape = ()

        self.num_links = self.time_size * self.space_size * self.dim
        self.num_plaqs = self.time_size * self.space_size
        self.bases = np.eye(self.dim, dtype=np.int)

        self.samples = self._init_samples(num_samples, rand)
        self.samples_tensor = tf.convert_to_tensor(self.samples,
                                                   dtype=TF_FLOAT)
        self.links = self.samples[0]
        self.batch_size = self.samples.shape[0]
        self.links_shape = self.samples.shape[1:]

    def _init_samples(self, num_samples, rand):
        """Initialize samples."""
        links_shape = tuple(
            [self.time_size]
            + [self.space_size for _ in range(self.dim-1)]
            + [self.dim]
            + list(self.link_shape)
        )
        samples_shape = (num_samples, *links_shape)
        if rand:
            samples = np.array(
                np.random.uniform(0, 2*np.pi, samples_shape),
                dtype=NP_FLOAT
            )
        else:
            samples = np.zeros(samples_shape, dtype=NP_FLOAT)
```

```
        return samples

    def calc_plaq_sums(self, samples=None):
        """Calculate plaquette sums.

        Explicitly, calculate the sum of the link variables around each
        plaquette in the lattice for each sample in samples.

        Args:
            samples (tf tensor): Tensor of shape (N, D) where N is the batch
                size and D is the number of links on the lattice. If samples is
                None, self.samples will be used.

        Returns:
            plaq_sums (tf operation): Tensorflow operation capable of
                calculating the plaquette sums.
        """
        if samples is None:
            samples = self.samples

        if samples.shape != self.samples.shape:
            samples = tf.reshape(samples, shape=(self.samples.shape))

        with tf.name_scope('calc_plaq_sums'):
            plaq_sums = (samples[:, :, :, 0]
                         - samples[:, :, :, 1]
                         - tf.roll(samples[:, :, :, 0], shift=-1, axis=2)
                         + tf.roll(samples[:, :, :, 1], shift=-1, axis=1))

        return plaq_sums

    def calc_actions(self, samples=None):
        """Calculate the total action for each sample in samples."""
        if samples is None:
            samples = self.samples

        with tf.name_scope('calc_total_actions'):
            total_actions = tf.reduce_sum(
                1. - tf.cos(self.calc_plaq_sums(samples)), axis=(1, 2),
                name='total_actions'
            )

        return total_actions

    def calc_plaqs(self, samples=None):
        """Calculate the average plaq. values for each sample in samples."""
        if samples is None:
            samples = self.samples

        with tf.name_scope('calc_plaqs'):
            plaqs = tf.reduce_sum(tf.cos(self.calc_plaq_sums(samples)),
                                  axis=(1, 2), name='plaqs') / self.num_plaqs
        return plaqs

    def calc_top_charges(self, samples=None, fft=False):
        """Calculate topological charges for each sample in samples."""
        if samples is None:
            samples = self.samples
```

```
        with tf.name_scope('calc_top_charges'):
            if fft:
                ps_proj = project_angle_fft(self.calc_plaq_sums(samples), N=1)
            else:
                ps_proj = project_angle(self.calc_plaq_sums(samples))

            top_charges = (tf.reduce_sum(ps_proj, axis=(1, 2),
                                         name='top_charges')) / (2 * np.pi)
        return top_charges

    # pylint: disable=invalid-name
    def calc_top_charges_diff(self, x1, x2, fft=False):
        """Calculate the difference in topological charge between x1 and x2."""
        with tf.name_scope('calc_top_charges_diff'):
            charge_diff = tf.abs(self.calc_top_charges(x1, fft)
                                 - self.calc_top_charges(x2, fft))

        return charge_diff

    def get_potential_fn(self, samples):
        """Returns callable function used for calculating the energy."""
        def fn(samples):
            return self.calc_actions(samples)
        return fn
```

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

```
"""
conv_net3d.py

Convolutional neural network architecture for running L2HMC on a gauge lattice
configuration of links.

Reference [Generalizing Hamiltonian Monte Carlo with Neural
Networks](https://arxiv.org/pdf/1711.09268.pdf)

Code adapted from the released TensorFlow graph implementation by original
authors https://github.com/brain-research/l2hmc.

Author: Sam Foreman (github: @saforem2)
Date: 01/16/2019
"""
import numpy as np
import tensorflow as tf

from globals import GLOBAL_SEED, TF_FLOAT

from .network_utils import custom_dense


np.random.seed(GLOBAL_SEED)

if '2.' not in tf.__version__:
    tf.set_random_seed(GLOBAL_SEED)


class ConvNet3D(tf.keras.Model):
    """Conv. neural net with different initialization scale based on input."""

    def __init__(self, model_name, **kwargs):
        """Initialization method.

        Attributes:
            coeff_scale: Multiplicative factor (lambda_s in original paper p.
                13) multiplying tanh(W_s h_2 + b_s).
            coeff_transformation: Multiplicative factor (lambda_q in original
                paper p. 13) multiplying tanh(W_q h_2 + b_q).
            data_format: String (either 'channels_first' or 'channels_last').
                'channels_first' ('channels_last') is default for GPU (CPU).
                This value is automatically determined and set to the
                appropriate value.
        """
        super(ConvNet3D, self).__init__(name=model_name)

        for key, val in kwargs.items():
            setattr(self, key, val)

        if self.use_bn:
            if self.data_format == 'channels_first':
                self.bn_axis = 1
            elif self.data_format == 'channels_last':
                self.bn_axis = -1
            else:
                raise AttributeError("Expected 'data_format' to be "
```

```
                                  "'channels_first'  or 'channels_last'")

        with tf.name_scope(self.name_scope):
            with tf.name_scope('coeff_scale'):
                self.coeff_scale = tf.Variable(
                    initial_value=tf.zeros([1, self.x_dim]),
                    name='coeff_scale',
                    trainable=True,
                    dtype=TF_FLOAT
                )

            with tf.name_scope('coeff_transformation'):
                self.coeff_transformation = tf.Variable(
                    initial_value=tf.zeros([1, self.x_dim]),
                    name='coeff_transformation',
                    trainable=True,
                    dtype=TF_FLOAT
                )

            with tf.name_scope('conv_layers'):
                with tf.name_scope('conv_x1'):
                    self.conv_x1 = tf.keras.layers.Conv3D(
                        filters=self.num_filters,
                        kernel_size=self.filter_sizes[0],
                        activation=tf.nn.relu,
                        input_shape=self._input_shape,
                        padding='same',
                        name='conv_x1',
                        dtype=TF_FLOAT,
                        data_format=self.data_format
                    )

                with tf.name_scope('pool_x1'):
                    self.max_pool_x1 = tf.keras.layers.MaxPooling3D(
                        pool_size=(2, 2, 2),
                        strides=2,
                        padding='same',
                        name='pool_x1',
                    )

                with tf.name_scope('conv_v1'):
                    self.conv_v1 = tf.keras.layers.Conv3D(
                        filters=self.num_filters,
                        kernel_size=self.filter_sizes[0],
                        activation=tf.nn.relu,
                        input_shape=self._input_shape,
                        padding='same',
                        name='conv_v1',
                        dtype=TF_FLOAT,
                        data_format=self.data_format
                    )

                with tf.name_scope('pool_v1'):
                    self.max_pool_v1 = tf.keras.layers.MaxPooling3D(
                        pool_size=(2, 2, 2),
                        strides=2,
                        padding='same',
                        name='pool_v1'
                    )

```

```
                with tf.name_scope('conv_x2'):
                    self.conv_x2 = tf.keras.layers.Conv3D(
                        filters=2*self.num_filters,
                        kernel_size=self.filter_sizes[1],
                        activation=tf.nn.relu,
                        padding='same',
                        name='conv_x2',
                        dtype=TF_FLOAT,
                        data_format=self.data_format
                    )

                with tf.name_scope('pool_x2'):
                    self.max_pool_x2 = tf.keras.layers.MaxPooling3D(
                        pool_size=(2, 2, 2),
                        strides=2,
                        padding='same',
                        name='pool_x2'
                    )

                with tf.name_scope('conv_v2'):
                    self.conv_v2 = tf.keras.layers.Conv3D(
                        filters=2 * self.num_filters,
                        kernel_size=self.filter_sizes[1],
                        activation=tf.nn.relu,
                        padding='same',
                        name='conv_v2',
                        dtype=TF_FLOAT,
                        data_format=self.data_format
                    )

                with tf.name_scope('pool_v2'):
                    self.max_pool_v2 = tf.keras.layers.MaxPooling3D(
                        pool_size=(2, 2, 2),
                        strides=2,
                        padding='same',
                        name='pool_v2'
                    )

            with tf.name_scope('fc_layers'):
                with tf.name_scope('flatten'):
                    self.flatten = tf.keras.layers.Flatten(name='flatten')

                with tf.name_scope('x_layer'):
                    self.x_layer = custom_dense(self.num_hidden,
                                                self.factor/3.,
                                                name='x_layer')

                with tf.name_scope('v_layer'):
                    self.v_layer = custom_dense(self.num_hidden,
                                                1./3.,
                                                name='v_layer')

                with tf.name_scope('t_layer'):
                    self.t_layer = custom_dense(self.num_hidden,
                                                1./3.,
                                                name='t_layer')

                with tf.name_scope('h_layer'):
                    self.h_layer = custom_dense(self.num_hidden,
                                                name='h_layer')
```

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

```
            t --> t_layer --> t_out

            x_out + v_out + t_out --> h_layer --> h_out
        ============================================================
        * h_out is then fed to three separate layers:
        ------------------------------------------------------------
            (1.) h_out --> (scale_layer, tanh) * exp(coeff_scale)
                 output: scale (S function in orig. paper)

            (2.) h_out --> translation_layer --> translation_out
                 output: translation (T function in orig. paper)

            (3.) h_out -->
                    (transformation_layer, tanh) * exp(coeff_transformation)
                 output: transformation (Q function in orig. paper)
          ============================================================

        Returns:
            scale, translation, transformation (S, T, Q functions from paper)
        """
        v, x, t = inputs
        #  scale_weight = net_weights[0]
        #  transformation_weight = net_weights[1]
        #  translation_weight = net_weights[2]

        with tf.name_scope('reshape'):
            v = self.reshape_5D(v)
            x = self.reshape_5D(x)

        with tf.name_scope('x'):
            x = self.max_pool_x1(self.conv_x1(x))
            x = self.max_pool_x2(self.conv_x2(x))
            if self.use_bn:
                x = tf.keras.layers.BatchNormalization(axis=self.bn_axis)(x)
            x = self.flatten(x)
            x = tf.nn.relu(self.x_layer(x))

        with tf.name_scope('v'):
            v = self.max_pool_v1(self.conv_v1(v))
            v = self.max_pool_v2(self.conv_v2(v))
            if self.use_bn:
                v = tf.keras.layers.BatchNormalization(axis=self.bn_axis)(v)
            v = self.flatten(v)
            v = tf.nn.relu(self.v_layer(v))

        with tf.name_scope('t'):
            t = tf.nn.relu(self.t_layer(t))

        with tf.name_scope('h'):
            h = tf.nn.relu(v + x + t)
            h = tf.nn.relu(self.h_layer(h))

        def reshape(t, name):
            return tf.squeeze(
                tf.reshape(t, shape=self._input_shape, name=name)
            )

        with tf.name_scope('scale'):
            scale = (tf.exp(self.coeff_scale)
```

```
                    * tf.nn.tanh(self.scale_layer(h)))

        with tf.name_scope('transformation'):
            transformation = (tf.exp(self.coeff_transformation)
                              * tf.nn.tanh(self.transformation_layer(h)))

        with tf.name_scope('translation'):
            translation = self.translation_layer(h)

        return scale, translation, transformation


# pylint:disable=too-many-arguments, too-many-instance-attributes
class ConvNet2D(tf.keras.Model):
    """Conv. neural net with different initialization scale based on input."""

    def __init__(self, model_name, **kwargs):
        """Initialization method."""

        super(ConvNet2D, self).__init__(name=model_name)

        for key, val in kwargs.items():
            setattr(self, key, val)

        with tf.name_scope(self.name_scope):

            self.coeff_scale = tf.Variable(
                initial_value=tf.zeros([1, self.x_dim]),
                name='coeff_scale',
                trainable=True,
                dtype=TF_FLOAT
            )

            self.coeff_transformation = tf.Variable(
                initial_value=tf.zeros([1, self.x_dim]),
                name='coeff_transformation',
                trainable=True,
                dtype=TF_FLOAT
            )

            self.conv_x1 = tf.keras.layers.Conv2D(
                filters=self.num_filters,
                kernel_size=self.filter_sizes[0],
                activation=tf.nn.relu,
                input_shape=self._input_shape,
                name='conv_x1',
                dtype=TF_FLOAT,
                data_format=self.data_format

            )

            self.max_pool_x1 = tf.keras.layers.MaxPooling2D(
                pool_size=(2, 2),
                strides=2,
                name='max_pool_x1'
            )

            self.conv_v1 = tf.keras.layers.Conv2D(
                filters=self.num_filters,
                kernel_size=self.filter_sizes[0],
```

```
            activation=tf.nn.relu,
            input_shape=self._input_shape,
            name='conv_v1',
            dtype=TF_FLOAT,
            data_format=self.data_format
        )

        self.max_pool_v1 = tf.keras.layers.MaxPooling2D(
            pool_size=(2, 2),
            strides=2,
            name='max_pool_x1'
        )

        self.conv_x2 = tf.keras.layers.Conv2D(
            filters=2*self.num_filters,
            kernel_size=self.filter_sizes[1],
            activation=tf.nn.relu,
            name='conv_x2',
            dtype=TF_FLOAT,
            data_format=self.data_format
        )

        self.max_pool_x2 = tf.keras.layers.MaxPooling2D(
            pool_size=(2, 2),
            strides=2,
            name='max_pool_x1'
        )

        self.conv_v2 = tf.keras.layers.Conv2D(
            filters=2 * self.num_filters,
            kernel_size=self.filter_sizes[1],
            activation=tf.nn.relu,
            name='conv_v2',
            dtype=TF_FLOAT,
            data_format=self.data_format
        )

        self.max_pool_v2 = tf.keras.layers.MaxPooling2D(
            pool_size=(2, 2),
            strides=2,
            name='max_pool_x1'
        )

        self.flatten = tf.keras.layers.Flatten(name='flatten')

        self.x_layer = custom_dense(self.num_hidden, self.factor/3.,
                                    name='x_layer')

        self.v_layer = custom_dense(self.num_hidden, 1./3.,
                                    name='v_layer')

        self.t_layer = custom_dense(self.num_hidden, 1./3.,
                                    name='t_layer')

        self.h_layer = custom_dense(self.num_hidden, name='h_layer')

        self.scale_layer = custom_dense(self.x_dim, 0.001,
                                        name='scale_layer')

        self.translation_layer = custom_dense(self.x_dim, 0.001,
```

```
                                                name='translation_layer')

            self.transformation_layer = custom_dense(
                self.x_dim,
                0.001,
                name='transformation_layer'
            )

    # pylint: disable=invalid-name, arguments-differ
    def call(self, inputs):
        """call method.

        Args:
            inputs: Tuple consisting of (v, x, t) (momenta, x, time).

        Returns:
           scale, translation, transformation

        NOTE: Architecture looks like
            - inputs: x, v, t
                x -->
                    CONV_X1, MAX_POOL_X1, --> CONV_X1, MAX_POOL_X2 -->
                    FLATTEN_X --> X_LAYER --> X_OUT

                v -->
                    CONV_V1, MAX_POOL_V1, --> CONV_V1, MAX_POOL_V2 -->
                    FLATTEN_V --> V_LAYER --> V_OUT

                t --> T_LAYER --> T_OUT

                X_OUT + V_OUT + T_OUT --> H_LAYER --> H_OUT

            - H_OUT is then fed to three separate layers:
                (1.) H_OUT --> (SCALE_LAYER, TANH) * exp(COEFF_SCALE)
                     output: scale
                (2.) H_OUT --> TRANSLATION_LAYER --> TRANSLATION_OUT
                     output: translation
                (3.) H_OUT --> (TRANSFORMATION_LAYER, TANH)
                                * exp(COEFF_TRANSFORMATION)

                     output: transformation
        """
        v, x, t = inputs

        x = self.max_pool_x1(self.conv_x1(x))
        x = tf.nn.local_response_normalization(x)
        x = self.max_pool_x2(self.conv_x2(x))
        x = tf.nn.local_response_normalization(x)
        x = self.flatten(x)

        v = self.max_pool_v1(self.conv_v1(v))
        v = tf.nn.local_response_normalization(v)
        v = self.max_pool_v2(self.conv_v2(v))
        v = tf.nn.local_response_normalization(v)
        v = self.flatten(v)

        h = tf.nn.relu(self.v_layer(v) + self.x_layer(x) + self.t_layer(t))
        h = tf.nn.relu(self.h_layer(h))
        #  h = self.hidden_layer1(h)
```

```
        def reshape(t, name):
            return tf.reshape(t, shape=self._input_shape, name=name)

        translation = reshape(self.translation_layer(h), name='translation')

        scale = reshape(
            tf.nn.tanh(self.scale_layer(h)) * tf.exp(self.coeff_scale),
            name='scale'
        )

        transformation = reshape(
            self.transformation_layer(h) * tf.exp(self.coeff_transformation),
            name='transformation'
        )

        return scale, translation, transformation
```

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

```python
"""
conv_net2d.py

Implements a convolutional neural network using 2D convolutions.

Author: Sam Foreman (github: @saforem2)
Date: 06/14/2019
"""
import numpy as np
import tensorflow as tf

from globals import GLOBAL_SEED, TF_FLOAT
from .network_utils import custom_dense


np.random.seed(GLOBAL_SEED)

if '2.' not in tf.__version__:
    tf.set_random_seed(GLOBAL_SEED)


class ConvNet2D(tf.keras.Model):
    """Conv. neural net with different initialization scale based on input."""

    def __init__(self, model_name, **kwargs):
        super(ConvNet2D, self).__init__(name=model_name)

        for key, val in kwargs.items():
            setattr(self, key, val)

        if self.use_bn:
            if self.data_format == 'channels_first':
                self.bn_axis = 1
            elif self.data_format == 'channels_last':
                self.bn_axis = -1
            else:
                raise AttributeError("Expected 'data_format' to be "
                                     "'channels_first'  or 'channels_last'")

        with tf.name_scope(self.name_scope):
            with tf.name_scope('coeff_scale'):
                self.coeff_scale = tf.Variable(
                    initial_value=tf.zeros([1, self.x_dim]),
                    name='coeff_scale',
                    trainable=True,
                    dtype=TF_FLOAT
                )

            with tf.name_scope('coeff_transformation'):
                self.coeff_transformation = tf.Variable(
                    initial_value=tf.zeros([1, self.x_dim]),
                    name='coeff_transformation',
                    trainable=True,
                    dtype=TF_FLOAT
                )

            with tf.name_scope('conv_layers'):
```

```
                with tf.name_scope('conv_x1'):
                    self.conv_x1 = tf.keras.layers.Conv2D(
                        filters=self.num_filters,
                        kernel_size=self.filter_sizes[0],
                        activation=tf.nn.relu,
                        input_shape=self._input_shape[1:],
                        #  padding='same',
                        name='conv_x1',
                        dtype=TF_FLOAT,
                        data_format=self.data_format

                    )

                with tf.name_scope('pool_x1'):
                    self.max_pool_x1 = tf.keras.layers.MaxPooling2D(
                        pool_size=(2, 2),
                        strides=2,
                        #  padding='same',
                        name='pool_x1',
                    )

                with tf.name_scope('conv_v1'):
                    self.conv_v1 = tf.keras.layers.Conv2D(
                        filters=self.num_filters,
                        kernel_size=self.filter_sizes[0],
                        activation=tf.nn.relu,
                        input_shape=self._input_shape[1:],
                        #  padding='same',
                        name='conv_v1',
                        dtype=TF_FLOAT,
                        data_format=self.data_format
                    )

                with tf.name_scope('pool_v1'):
                    self.max_pool_v1 = tf.keras.layers.MaxPooling2D(
                        pool_size=(2, 2),
                        strides=2,
                        #  padding='same',
                        name='pool_v1'
                    )

                with tf.name_scope('conv_x2'):
                    self.conv_x2 = tf.keras.layers.Conv2D(
                        filters=2*self.num_filters,
                        kernel_size=self.filter_sizes[1],
                        activation=tf.nn.relu,
                        #  padding='same',
                        name='conv_x2',
                        dtype=TF_FLOAT,
                        data_format=self.data_format
                    )

                with tf.name_scope('pool_x2'):
                    self.max_pool_x2 = tf.keras.layers.MaxPooling2D(
                        pool_size=(2, 2),
                        strides=2,
                        #  padding='same',
                        name='pool_x2'
                    )

```

```
                with tf.name_scope('conv_v2'):
                    self.conv_v2 = tf.keras.layers.Conv2D(
                        filters=2*self.num_filters,
                        kernel_size=self.filter_sizes[1],
                        activation=tf.nn.relu,
                        #  padding='same',
                        name='conv_v2',
                        dtype=TF_FLOAT,
                        data_format=self.data_format
                    )

                with tf.name_scope('pool_v2'):
                    self.max_pool_v2 = tf.keras.layers.MaxPooling2D(
                        pool_size=(2, 2),
                        strides=2,
                        #  padding='same',
                        name='pool_v2'
                    )

            with tf.name_scope('fc_layers'):
                with tf.name_scope('flatten'):
                    self.flatten = tf.keras.layers.Flatten(name='flatten')

                with tf.name_scope('x_layer'):
                    self.x_layer = custom_dense(self.num_hidden,
                                                self.factor/3.,
                                                name='x_layer')

                with tf.name_scope('v_layer'):
                    self.v_layer = custom_dense(self.num_hidden,
                                                1./3.,
                                                name='v_layer')

                with tf.name_scope('t_layer'):
                    self.t_layer = custom_dense(self.num_hidden,
                                                1./3.,
                                                name='t_layer')

                with tf.name_scope('h_layer'):
                    self.h_layer = custom_dense(self.num_hidden,
                                                name='h_layer')

                with tf.name_scope('scale_layer'):
                    self.scale_layer = custom_dense(
                        self.x_dim, 0.001, name='scale_layer'
                    )

                with tf.name_scope('translation_layer'):
                    self.translation_layer = custom_dense(
                        self.x_dim, 0.001, 'translation_layer'
                    )

                with tf.name_scope('transformation_layer'):
                    self.transformation_layer = custom_dense(
                        self.x_dim, 0.001, 'transformation_layer'
                    )

    def _reshape(self, tensor):
        """Reshape tensor to be compatible with tf.keras.layers.Conv2D."""
        if self.data_format == 'channels_first':
```

```
            # batch_size, num_channels, height, width
            N, D, H, W = self._input_shape
            #  N, D, H, W = tensor.shape
            if isinstance(tensor, np.ndarray):
                return np.reshape(tensor, (N, D, H, W))

            return tf.reshape(tensor, (N, D, H, W))

        if self.data_format == 'channels_last':
            N, H, W, D = self._input_shape
            if isinstance(tensor, np.ndarray):
                return np.reshape(tensor, (N, H, W, D))

            return tf.reshape(tensor, (N, H, W, D))

        raise AttributeError("`self.data_format` should be one of "
                             "'channels_first' or 'channels_last'")

    def call(self, inputs):
        """Forward pass through the network."""
        v, x, t = inputs

        with tf.name_scope('reshape'):
            v = self._reshape(v)
            x = self._reshape(x)

        with tf.name_scope('x'):
            x = self.max_pool_x1(self.conv_x1(x))
            x = self.max_pool_x2(self.conv_x2(x))
            if self.use_bn:
                x = tf.keras.layers.BatchNormalization(axis=self.bn_axis)(x)
            x = tf.nn.relu(self.x_layer(self.flatten(x)))

        with tf.name_scope('v'):
            v = self.max_pool_v1(self.conv_v1(v))
            v = self.max_pool_v2(self.conv_v2(v))
            if self.use_bn:
                v = tf.keras.layers.BatchNormalization(axis=self.bn_axis)(v)
            v = tf.nn.relu(self.v_layer(self.flatten(v)))
            #  v = self.flatten(v)
            #  v = tf.nn.relu(self.v_layer(v))

        with tf.name_scope('t'):
            t = tf.nn.relu(self.t_layer(t))

        with tf.name_scope('h'):
            h = tf.nn.relu(v + x + t)
            h = tf.nn.relu(self.h_layer(h))

        with tf.name_scope('scale'):
            scale = (tf.exp(self.coeff_scale)
                     * tf.nn.tanh(self.scale_layer(h)))

        with tf.name_scope('transformation'):
            transformation = (tf.exp(self.coeff_transformation)
                              * tf.nn.tanh(self.transformation_layer(h)))

        with tf.name_scope('translation'):
            translation = self.translation_layer(h)

```

```
238         #  with tf.name_scope('scale'):
239         #      scale = (tf.nn.tanh(self.scale_layer(h))
240         #               * tf.exp(self.coeff_scale))
241         #
242         #  with tf.name_scope('transformation'):
243         #      transformation = (tf.nn.tanh(self.transformation_layer(h))
244         #                        * tf.exp(self.coeff_transformation))
245
246         return scale, translation, transformation
```

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

```python
"""
Generic, fully-connected neural network architecture for running L2HMC on a
gauge lattice configuration of links.

NOTE: Lattices are flattened before being passed as input to the network.

Reference [Generalizing Hamiltonian Monte Carlo with Neural
Networks](https://arxiv.org/pdf/1711.09268.pdf)

Code adapted from the released TensorFlow graph implementation by original
authors https://github.com/brain-research/l2hmc.

Author: Sam Foreman (github: @saforem2)
Date: 01/16/2019
"""
import tensorflow as tf
import numpy as np

from globals import TF_FLOAT

from .network_utils import custom_dense


class GenericNet(tf.keras.Model):
    """Conv. neural net with different initialization scale based on input."""
    def __init__(self, model_name='GenericNet', **kwargs):
        """Initialization method."""

        super(GenericNet, self).__init__(name=model_name)

        for key, val in kwargs.items():
            setattr(self, key, val)

        if self.name_scope is None:
            self.name_scope = model_name

        if self.use_bn:
            self.bn_axis = -1

        #  with tf.variable_scope(variable_scope):
        with tf.name_scope(self.name_scope):
            #  self.flatten = tf.keras.layers.Flatten(name='flatten')

            with tf.name_scope('x_layer'):
                self.x_layer = custom_dense(self.num_hidden,
                                            self.factor/3.,
                                            name='x_layer')

            with tf.name_scope('v_layer'):
                self.v_layer = custom_dense(self.num_hidden,
                                            1./3.,
                                            name='v_layer')

            with tf.name_scope('t_layer'):
                self.t_layer = custom_dense(self.num_hidden,
                                            1./3., name='t_layer')

```

```
            with tf.name_scope('h_layer'):
                self.h_layer = custom_dense(self.num_hidden,
                                            name='h_layer')

            with tf.name_scope('scale_layer'):
                self.scale_layer = custom_dense(self.x_dim, 0.001,
                                                name='scale_layer')

            with tf.name_scope('translation_layer'):
                self.translation_layer = custom_dense(
                    self.x_dim,
                    0.001,
                    name='translation_layer'
                )

            with tf.name_scope('transformation_layer'):
                self.transformation_layer = custom_dense(
                    self.x_dim,
                    0.001,
                    name='transformation_layer'
                )

            with tf.name_scope('coeff_scale'):
                self.coeff_scale = tf.Variable(
                    initial_value=tf.zeros([1, self.x_dim]),
                    name='coeff_scale',
                    trainable=True,
                    dtype=TF_FLOAT,
                )

            with tf.name_scope('coeff_transformation'):
                self.coeff_transformation = tf.Variable(
                    initial_value=tf.zeros([1, self.x_dim]),
                    name='coeff_transformation',
                    trainable=True,
                    dtype=TF_FLOAT
                )

        def _reshape(self, tensor):
            N, D, H, W = self._input_shape
            if isinstance(tensor, np.ndarray):
                return np.reshape(tensor, (N, D * H * W))

            return tf.reshape(tensor, (N, D * H * W))

        # pylint: disable=invalid-name, arguments-differ
        def call(self, inputs):
            """call method.

            NOTE Architecture looks like:

                * inputs: x, v, t
                    x --> FLATTEN_X --> X_LAYER --> X_OUT
                    v --> FLATTEN_V --> V_LAYER --> V_OUT
                    t --> T_LAYER --> T_OUT

                    X_OUT + V_OUT + T_OUT --> H_LAYER --> H_OUT

                * H_OUT is then fed to three separate layers:
                    (1.) H_OUT -->
```

```
                    TANH(SCALE_LAYER) * exp(COEFF_SCALE) --> SCALE_OUT

                input: H_OUT
                output: scale

            (2.) H_OUT --> TRANSLATION_LAYER --> TRANSLATION_OUT

                input: H_OUT
                output: translation

            (3.) H_OUT -->
                   TANH(SCALE_LAYER)*exp(COEFF_TRANSFORMATION) -->
                     TRANFORMATION_OUT

                input: H_OUT
                output: transformation

        Returns:
            scale, translation, transformation
        """
        v, x, t = inputs

        x = self._reshape(x)
        v = self._reshape(v)

        h = self.v_layer(v) + self.x_layer(x) + self.t_layer(t)
        if self.use_bn:
            h = tf.keras.layers.BatchNormalization(axis=self.bn_axis)(h)
        h = tf.nn.relu(h)
        h = self.h_layer(h)
        h = tf.nn.relu(h)

        with tf.name_scope('scale'):
            scale = (tf.exp(self.coeff_scale)
                     * tf.nn.tanh(self.scale_layer(h)))

        with tf.name_scope('transformation'):
            transformation = (tf.exp(self.coeff_transformation)
                              * tf.nn.tanh(self.transformation_layer(h)))

        with tf.name_scope('translation'):
            translation = self.translation_layer(h)

        #  scale = tf.nn.tanh(self.scale_layer(h)) * tf.exp(self.coeff_scale)

        #  transformation = (tf.nn.tanh(self.transformation_layer(h))
        #                    * tf.exp(self.coeff_transformation))

        return scale, translation, transformation
```

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

```python
import numpy as np
import tensorflow as tf

from globals import GLOBAL_SEED, TF_FLOAT, NP_FLOAT


np.random.seed(GLOBAL_SEED)

if '2.' not in tf.__version__:
    tf.set_random_seed(GLOBAL_SEED)


def custom_dense(units, factor=1., name=None):
    """Custom dense layer with specified weight intialization."""
    if '2.' not in tf.__version__:
        kernel_initializer = tf.keras.initializers.VarianceScaling(
            scale=factor,
            mode='fan_in',
            distribution='uniform',
            dtype=TF_FLOAT,
            seed=GLOBAL_SEED,
        )
    else:
        kernel_initializer = tf.contrib.layers.variance_scaling_initializer(
            factor=factor,
            mode='FAN_IN',
            seed=GLOBAL_SEED,
            uniform=True,
        )

    return tf.keras.layers.Dense(
        units=units,
        use_bias=True,
        kernel_initializer=kernel_initializer,
        bias_initializer=tf.constant_initializer(0., dtype=TF_FLOAT),
        name=name
    )


def variable_on_cpu(name, shape, initializer):
    """Helper to create a Variable stored on CPU memory.

    Args:
        name: name of the variable
        shape: list of ints
        initializer: initializer for Variable

    Returns:
        Variable Tensor
    """
    with tf.device('/cpu:0'):
        var = tf.get_variable(name, shape, initializer, TF_FLOAT)
    return var


def variable_with_weight_decay(name, shape, stddev, wd, cpu=True):
    """Helper to create an initialized Variable with weight decay.
```

```
    Note that the Variable is initialized with a truncated normal distribution.
    A weight decay is added only if one is specified.

    Args:
        name: Name of the variable
        shape: list of ints
        stddev: standard deviation of a truncated Gaussian
        wd: Add L2Loss weight decay multiplied by this float. If None, weight
            decay is not added for this variable.

    Returns:
        Variable Tensor
    """
    if cpu:
        var = variable_on_cpu(
            name, shape, tf.truncated_normal_initializer(stddev=stddev,
                                                         dtype=TF_FLOAT)
        )
    else:
        var = tf.get_variable(
            name, shape, tf.truncated_normal_initializer(stddev=stddev,
                                                         dtype=TF_FLOAT)
        )
    if wd is not None:
        weight_decay = tf.multiply(tf.nn.l2_loss(var), wd, name='weight_loss')
        tf.add_to_collection('losses', weight_decay)

    return var

def create_periodic_padding(samples, filter_size):
    """Create periodic padding for multiple samples, using filter_size."""
    original_size = np.shape(samples)
    N = original_size[1]  # number of links in lattice
    #  N = np.shape(samples)[1] # number of links in lattice
    padding = filter_size - 1

    samples = tf.reshape(samples, shape=(samples.shape[0], -1))

    x = []
    for sample in samples:
        padded = np.zeros((N + 2 * padding), N + 2 * padding, 2)
        # lower left corner
        padded[:padding, :padding, :] = sample[N-padding:, N-padding:, :]
        # lower middle
        padded[padding:N+padding, :padding, :] = sample[:, N-padding:, :]
        # loewr right corner
        padded[N+padding:, :padding, :] = sample[:padding, N-padding:, :]
        # left side
        padded[:padding, padding: N+padding, :] = sample[N-padding:, :, :]
        # center
        padded[:padding:N+padding, padding:N+padding, :] = sample[:, :, :]
        # right side
        padded[N+padding:, padding:N+padding:, :] = sample[:padding, :, :]
        # top middle
        padded[:padding:N+padding, N+padding:, :] = sample[:, :padding, :]
        # top right corner
        padded[N+padding:, N+padding:, :] = sample[:padding, :padding, :]

        x.append(padded)
```

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

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

```python
"""
gauge_model_trainer.py

Implements GaugeModelTrainer class responsible for training GaugeModel.

Author: Sam Foreman (github: @saforem2)
Date: 04/09/2019
"""
#  import os
import time
import numpy as np
#  import tensorflow as tf

try:
    import horovod.tensorflow as hvd
    HAS_HOROVOD = True
except ImportError:
    HAS_HOROVOD = False

import utils.file_io as io
from lattice.lattice import u1_plaq_exact
from globals import NP_FLOAT


h_str = ("{:^12s}{:^10s}{:^10s}{:^10s}{:^10s}"
         "{:^10s}{:^10s}{:^10s}{:^10s}{:^10s}")

h_strf = h_str.format("STEP", "LOSS", "t/STEP", "% ACC", "EPS",
                      "BETA", "ACTION", "PLAQ", "(EXACT)", "dQ", "LR")

dash0 = (len(h_strf) + 1) * '-'
dash1 = (len(h_strf) + 1) * '-'
TRAIN_HEADER = dash0 + '\n' + h_strf + '\n' + dash1


class GaugeModelTrainer:
    def __init__(self, sess, model, logger=None):
        """Initialization method.

        Args:
            sess: tf.Session object.
            model: GaugeModel object (defined in `models/gauge_model.py`)
            logger: TrainLogger object (defined in `loggers/train_logger.py`)
        """
        self.sess = sess
        self.model = model
        self.logger = logger

    def update_beta(self, step):
        """Returns new beta to follow annealing schedule."""
        temp = ((1. / self.model.beta_init - 1. / self.model.beta_final)
                * (1. - step / float(self.model.train_steps))
                + 1. / self.model.beta_final)
        new_beta = 1. / temp

        return new_beta

```

```python
    def train_step(self, step, samples_np, beta_np=None, **weights):
        """Perform a single training step.

        Args:
            step (int): Current training step.
            samples_np (np.ndarray): Array of input data.
            beta_np (float, optional): Input value for inverse coupling
                constant.

        Returns:
            out_data (dict)
        """
        start_time = time.time()

        if beta_np is None:
            beta_np = self.update_beta(step)

        charge_weight = weights.get('charge_weight', None)
        net_weights = weights.get('net_weights', None)

        if charge_weight is None:
            charge_weight = 0.

        if net_weights is None:
            # scale_weight, transformation_weight, translation_weight
            net_weights = [1., 1., 1.]

        fd = {
            self.model.x: samples_np,
            self.model.beta: beta_np,
            self.model.net_weights[0]: net_weights[0],
            self.model.net_weights[1]: net_weights[1],
            self.model.net_weights[2]: net_weights[2],
            self.model.charge_weight: charge_weight
        }

        global_step = self.sess.run(self.model.global_step)

        ops = [
            self.model.train_op,         # apply gradients
            self.model.loss_op,          # calculate loss
            self.model.x_out,            # get new samples
            self.model.px,               # calculate accept prob.
            self.model.dynamics.eps,     # calculate current step size
            self.model.actions_op,       # calculate avg. actions
            self.model.plaqs_op,         # calculate avg. plaqs
            self.model.charges_op,       # calculate top. charges
            self.model.charge_diffs_op,  # change in top. charge / num_samples
            self.model.lr,               # evaluate learning rate
        ]

        outputs = self.sess.run(ops, feed_dict=fd)

        dt = time.time() - start_time
        out_data = {
            'step': global_step,
            'loss': outputs[1],
            'samples': np.mod(outputs[2], 2 * np.pi),
            'px': outputs[3],
            'eps': outputs[4],
```

```
            'actions': outputs[5],
            'plaqs': outputs[6],
            'charges': outputs[7],
            'charge_diffs': outputs[8],
            'lr': outputs[9],
            'beta': beta_np
        }

        data_str = (
            f"{global_step:>5g}/{self.model.train_steps:<6g} "
            f"{outputs[1]:^9.4g} "              # loss value
            f"{dt:^9.4g} "                      # time / step
            f"{np.mean(outputs[3]):^9.4g}"      # accept prob
            f"{outputs[4]:^9.4g} "              # step size
            f"{beta_np:^9.4g} "                 # beta
            f"{np.mean(outputs[5]):^9.4g} "     # avg. actions
            f"{np.mean(outputs[6]):^9.4g} "     # avg. plaqs.
            f"{u1_plaq_exact(beta_np):^9.4g} "  # exact plaq.
            f"{outputs[8]:^9.4g} "              # charge diff
            f"{outputs[9]:^9.4g}"               # learning rate
        )

        return out_data, data_str

    def train(self, train_steps, **kwargs):
        """Train the L2HMC sampler for `train_steps`.

        Args:
            train_steps: Integer number of training steps to perform.
            **kwargs: Possible (key, value) pairs are
                'samples_np': Array of initial samples used to start
                    training.
                'beta_np': Initial value of beta used in annealing
                    schedule.
                'trace': Flag specifying that the training loop should be
                    ran through a profiler.
        """
        initial_step = kwargs.get('initial_step', 0)
        samples_np = kwargs.get('samples_np', None)
        beta_np = kwargs.get('beta_np', None)
        charge_weight = kwargs.get('charge_weight', None)
        net_weights = kwargs.get('net_weights', None)

        if beta_np is None:
            beta_np = self.model.beta_init

        if samples_np is None:
            samples_np = np.reshape(
                np.array(self.model.lattice.samples, dtype=NP_FLOAT),
                (self.model.num_samples, self.model.x_dim)
            )

        assert samples_np.shape == self.model.x.shape

        weights = {
            'charge_weight': charge_weight,
            'net_weights': net_weights
        }

        try:
```

```
            io.log(TRAIN_HEADER)
            for step in range(initial_step, train_steps):
                out_data, data_str = self.train_step(step, samples_np,
                                                     **weights)
                samples_np = out_data['samples']

                # NOTE: try/except faster than explicitly checking
                # if self.logger is not None
                # each training step
                try:
                    self.logger.update_training(self.sess, out_data,
                                                data_str, **weights)
                except AttributeError:
                    continue
            try:
                self.logger.write_train_strings()
            except AttributeError:
                pass

        except (KeyboardInterrupt, SystemExit):
            io.log("\nKeyboardInterrupt detected!")
            io.log("Saving current state and exiting.")
            if self.logger is not None:
                self.logger.update_training(self.sess, out_data,
                                            data_str, **weights)
```

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

```python
"""
runner.py

Implements GaugeModelRunner class responsible for running the L2HMC algorithm
on a U(1) gauge model.

Author: Sam Foreman (github: @saforem2)
Date: 04/09/2019
"""
import os
import time
import pickle
from scipy.stats import sem
from collections import Counter, OrderedDict
import numpy as np

import utils.file_io as io
from lattice.lattice import u1_plaq_exact
from globals import RUN_HEADER

#  ops = [                          # list of tensorflow operations to run
#      self.model.x_out,            # new samples (MD + MH accept/reject)
#      self.model.px,               # prob. of accepting proposed samples
#      self.model.actions_op,       # tot. action of each sample
#      self.model.plaqs_op,         # avg. plaquette of each sample
#      self.model.charges_op,       # topological charge Q, of each sample
#      self.model.charge_diffs_op   # Q(x_out) - Q(samples_in)
#  ]


class GaugeModelRunner:

    def __init__(self, sess, model, logger=None):
        """
        Args:
            sess: tf.Session() object.
            model: GaugeModel object (defined in `models/gauge_model.py`)
            logger: RunLogger object (defined in `loggers/run_logger.py`),
                defaults to None. This is to simplify communication when using
                Horovod since the RunLogger object exists only on
                hvd.rank() == 0, which is responsible for all file I/O.
        """
        self.sess = sess
        self.model = model
        self.logger = logger
        self.eps = self.sess.run(self.model.dynamics.eps)

    def calc_charge_autocorrelation(self, charges):
        autocorr = np.correlate(charges, charges, mode='full')
        return autocorr

    def save_run_data(self, run_data, run_strings, samples, **kwargs):
        """Save run information.

        Args:
            run_data: All run data generated from `run` method.
            run_strings: list of all strings generated from `run` method.
```

```
            samples: Optional collection of all samples generated from `run`
            method. Only relevant if model.sve_samples is True
        """
        run_dir = kwargs['run_dir']
        observables_dir = os.path.join(run_dir, 'observables')

        io.check_else_make_dir(run_dir)
        io.check_else_make_dir(observables_dir)

        if self.model.save_samples:
            samples_file = os.path.join(run_dir, 'run_samples.pkl')
            io.log(f"Saving samples to: {samples_file}.")
            with open(samples_file, 'wb') as f:
                pickle.dump(samples, f)

        run_stats = self.calc_observables_stats(run_data,
                                                kwargs['therm_frac'])

        data_file = os.path.join(run_dir, 'run_data.pkl')
        io.log(f"Saving run_data to: {data_file}.")
        with open(data_file, 'wb') as f:
            pickle.dump(run_data, f)

        stats_data_file = os.path.join(run_dir, 'run_stats.pkl')
        io.log(f"Saving run_stats to: {stats_data_file}.")
        with open(stats_data_file, 'wb') as f:
            pickle.dump(run_stats, f)

        for key, val in run_data.items():
            out_file = key + '.pkl'
            out_file = os.path.join(observables_dir, out_file)
            io.save_data(val, out_file, name=key)

        for key, val in run_stats.items():
            out_file = key + '_stats.pkl'
            out_file = os.path.join(observables_dir, out_file)
            io.save_data(val, out_file, name=key)

        history_file = os.path.join(run_dir, 'run_history.txt')
        _ = [io.write(s, history_file, 'a') for s in run_strings]

        self.write_run_stats(run_stats, **kwargs)

    def run_step(self, step, run_steps, inputs, **weights):
        """Perform a single run step.

        Args:
            step (int): Current step.
            run_steps (int): Total number of run_steps to perform.
            inputs (tuple): Tuple consisting of (samples_in, beta_np, eps,
                plaq_exact) where samples_in (np.ndarray) is the input batch of
                samples, beta (float)is the input value of beta, eps is the
                step size, and plaq_exact (float) is the expected avg. value of
                the plaquette at this value of beta.
        Returns:
            out_data: Dictionary containing the output of running all of the
            tensorflow operations in `ops` defined below.
        """
        samples_in, beta_np, eps, plaq_exact = inputs
```

```
        ops = [
            self.model.x_out,
            self.model.px,
            self.model.actions_op,
            self.model.plaqs_op,
            self.model.charges_op,
            self.model.charge_diffs_op,
        ]
        if self.model.save_lf:
            ops.extend([
                self.model.lf_out_f,
                self.model.pxs_out_f,
                self.model.lf_out_b,
                self.model.pxs_out_b,
                self.model.masks_f,
                self.model.masks_b,
                self.model.logdets_f,
                self.model.logdets_b,
                self.model.sumlogdet_f,
                self.model.sumlogdet_b
            ])

        charge_weight = weights['charge_weight']
        net_weights = weights['net_weights']

        feed_dict = {
            self.model.x: samples_in,
            self.model.beta: beta_np,
            self.model.charge_weight: charge_weight,
            self.model.net_weights[0]: net_weights[0],
            self.model.net_weights[1]: net_weights[1],
            self.model.net_weights[2]: net_weights[2],
        }

        start_time = time.time()
        outputs = self.sess.run(ops, feed_dict=feed_dict)
        dt = time.time() - start_time
        out_data = {
            'step': step,
            'beta': beta_np,
            'eps': self.eps,
            'samples': np.mod(outputs[0], 2 * np.pi),
            'px': outputs[1],
            'actions': outputs[2],
            'plaqs': outputs[3],
            'charges': outputs[4],
            'charge_diffs': outputs[5],
        }
        if self.model.save_lf:
            lf_outputs = {
                'lf_out_f': outputs[6],
                'pxs_out_f': outputs[7],
                'lf_out_b': outputs[8],
                'pxs_out_b': outputs[9],
                'masks_f': outputs[10],
                'masks_b': outputs[11],
                'logdets_f': outputs[12],
                'logdets_b': outputs[13],
                'sumlogdet_f': outputs[14],
                'sumlogdet_b': outputs[15],
```

```
            }
            out_data.update(lf_outputs)

        data_str = (f'{step:>5g}/{run_steps:<6g} '
                    f'{dt:^9.4g} '                      # time / step
                    f'{np.mean(outputs[1]):^9.4g} '     # accept. prob
                    f'{self.eps:^9.4g} '                # step size
                    f'{beta_np:^9.4g} '                 # beta val
                    f'{np.mean(outputs[2]):^9.4g} '     # avg. actions
                    f'{np.mean(outputs[3]):^9.4g} '     # avg. plaquettes
                    f'{plaq_exact:^9.4g} '              # exact plaquette val
                    f'{outputs[5]:^9.4g} ')             # top. charge diff

        return out_data, data_str

    def run(self, run_steps, beta=None, therm_frac=10, **weights):
        """Run the simulation to generate samples and calculate observables.

        Args:
            run_steps: Number of steps to run the sampler for.
            current_step: Integer passed when the sampler is ran intermittently
                during training, as a way to monitor the models performance
                during training. By passing the current training step as
                current_step, this data is saved to a unique directory labeled
                by the current_step.
            beta: Float value indicating the inverse coupling constant that the
                sampler is to be run at.

        Returns:
            observables: Tuple of observables dictionaries consisting of:
                (actions_dict, plaqs_dict, charges_dict, charge_diffs_dict).
        """
        run_steps = int(run_steps)

        if beta is None:
            beta = self.model.beta_final

        plaq_exact = u1_plaq_exact(beta)

        # start with randomly generated samples
        samples_np = np.random.randn(*(self.model.batch_size,
                                       self.model.x_dim))

        try:
            io.log(RUN_HEADER)
            for step in range(run_steps):
                inputs = (samples_np, beta, self.eps, plaq_exact)
                out_data, data_str = self.run_step(step,
                                                   run_steps,
                                                   inputs,
                                                   **weights)
                samples_np = out_data['samples']

                if self.logger is not None:
                    self.logger.update(out_data, data_str)

            if self.logger is not None:
                self.logger.save_run_data(therm_frac=therm_frac)

        except (KeyboardInterrupt, SystemExit):
```

```
            io.log("\nKeyboardInterrupt detected!")
            io.log("Saving current state and exiting.")
            if self.logger is not None:
                self.logger.save_run_data(therm_frac=therm_frac)

    def calc_observables_stats(self, run_data, therm_frac=10):
        """Calculate statistics for lattice observables.

        Args:
            run_data: Dictionary of observables data. Keys denote the
                observables name.
            therm_frac: Fraction of data to throw out for thermalization.

        Returns:
            stats: Dictionary containing statistics for each observable in
            run_data. Additionally, contains `charge_probs` which is a
            dictionary of the form {charge_val: charge_val_probability}.
        """
        def get_stats(data, t_frac=10):
            if isinstance(data, dict):
                arr = np.array(list(data.values()))
            elif isinstance(data, (list, np.ndarray)):
                arr = np.array(data)

            num_steps = arr.shape[0]
            therm_steps = num_steps // t_frac
            arr = arr[therm_steps:, :]
            avg = np.mean(arr, axis=0)
            err = sem(arr, axis=0)
            stats = np.array([avg, err]).T
            return stats

        actions_stats = get_stats(run_data['actions'], therm_frac)
        plaqs_stats = get_stats(run_data['plaqs'], therm_frac)

        charges_arr = np.array(list(run_data['charges'].values()), dtype=int)
        charges_stats = get_stats(charges_arr, therm_frac)

        suscept_arr = charges_arr ** 2
        suscept_stats = get_stats(suscept_arr)

        charge_probs = {}
        counts = Counter(list(charges_arr.flatten()))
        total_counts = np.sum(list(counts.values()))
        for key, val in counts.items():
            charge_probs[key] = val / total_counts

        charge_probs = OrderedDict(sorted(charge_probs.items(),
                                          key=lambda k: k[0]))

        stats = {
            'actions': actions_stats,
            'plaqs': plaqs_stats,
            'charges': charges_stats,
            'suscept': suscept_stats,
            'charge_probs': charge_probs
        }

        return stats

```

```
    def write_run_stats(self, stats, **kwargs):
        """Write statistics in human readable format to .txt file."""
        run_steps = kwargs['run_steps']
        beta = kwargs['beta']
        current_step = kwargs['current_step']
        therm_steps = kwargs['therm_steps']
        training = kwargs['training']
        run_dir = kwargs['run_dir']

        out_file = os.path.join(run_dir, 'run_stats.txt')

        actions_avg, actions_err = stats['actions'].mean(axis=0)
        plaqs_avg, plaqs_err = stats['plaqs'].mean(axis=0)
        charges_avg, charges_err = stats['charges'].mean(axis=0)
        suscept_avg, suscept_err = stats['suscept'].mean(axis=0)

        #  actions_arr = np.array(
        #      list(run_data['actions'].values())
        #  )[therm_steps:, :]
        #
        #  plaqs_arr = np.array(
        #      list(run_data['plaqs'].values())
        #  )[therm_steps:, :]
        #
        #  charges_arr = np.array(
        #      list(run_data['charges'].values()),
        #      dtype=np.int32
        #  )[therm_steps:, :]
        #
        #  charges_squared_arr = charges_arr ** 2
        #
        #  actions_err = sem(actions_arr, axis=None)
        #
        #  plaqs_avg = np.mean(plaqs_arr)
        #  plaqs_err = sem(plaqs_arr, axis=None)
        #
        #  q_avg = np.mean(charges_arr)
        #  q_err = sem(charges_arr, axis=None)
        #
        #  q2_avg = np.mean(charges_squared_arr)
        #  q2_err = sem(charges_squared_arr, axis=None)

        ns = self.model.num_samples
        suscept_k1 = f'  \navg. over all {ns} samples < Q >'
        suscept_k2 = f'  \navg. over all {ns} samples < Q^2 >'
        actions_k1 = f'  \navg. over all {ns} samples < action >'
        plaqs_k1 = f'  \n avg. over all {ns} samples < plaq >'

        _est_key = '  \nestimate +/- stderr'

        suscept_ss = {
            suscept_k1: f"{charges_avg:.4g} +/- {charges_err:.4g}",
            suscept_k2: f"{suscept_avg:.4g} +/- {suscept_err:.4g}",
            _est_key: {}
        }

        actions_ss = {
            actions_k1: f"{actions_avg:.4g} +/- {actions_err:.4g}\n",
            _est_key: {}
        }
```

```
358
359         plaqs_ss = {
360             'exact_plaq': f"{u1_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         suscept_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         suscept_ss = zip_keys_vals(suscept_ss, keys, suscept_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         suscept_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
```

```
            str2 = "Average plaquette" + run_str
            str3 = "Topological charge probabilities" + run_str[6:]
            therm_str = (
                f'Ignoring first {therm_steps} steps for thermalization.'
            )

        ss0 = (1 + max(len(str0), len(therm_str))) * '-'
        ss1 = (1 + max(len(str1), len(therm_str))) * '-'
        ss2 = (1 + max(len(str2), len(therm_str))) * '-'
        ss3 = (1 + max(len(str3), len(therm_str))) * '-'

        io.log(f"Writing statistics to: {out_file}")

        def log_and_write(sep_str, str0, therm_str, stats_strings, file):
            io.log(sep_str)
            io.log(str0)
            io.log(therm_str)
            io.log('')
            _ = [io.log(s) for s in stats_strings]
            io.log(sep_str)
            io.log('')

            io.write(sep_str, file, 'a')
            io.write(str0, file, 'a')
            io.write(therm_str, file, 'a')
            _ = [io.write(s, file, 'a') for s in stats_strings]
            io.write('\n', file, 'a')

        log_and_write(ss0, str0, therm_str, suscept_strings, out_file)
        log_and_write(ss1, str1, therm_str, actions_strings, out_file)
        log_and_write(ss2, str2, therm_str, plaqs_strings, out_file)
        log_and_write(ss3, str3, therm_str, charge_probs_strings, out_file)
```

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

```python
"""
run_logger.py

Implements RunLogger class responsible for saving/logging data
from `run` phase of GaugeModel.

Author: Sam Foreman (github: @saforem2)
Date: 04/24/2019
"""
import os
import pickle

import numpy as np

from collections import Counter, OrderedDict
from scipy.stats import sem
import utils.file_io as io

from globals import RUN_HEADER, NP_FLOAT

from lattice.lattice import u1_plaq_exact

from .train_logger import save_params


def arr_from_dict(d, key):
    return np.array(list(d[key]))


def autocorr(x):
    autocorr = np.correlate(x, x, mode='full')

    return autocorr[autocorr.size // 2:]


class RunLogger:
    def __init__(self, model, log_dir, save_lf_data=False):
        """
        Args:
            model: GaugeModel object.
            log_dir: Existing logdir from `TrainLogger`.
        """
        #  self.sess = sess
        self.model = model
        assert os.path.isdir(log_dir)
        self.log_dir = log_dir
        self.runs_dir = os.path.join(self.log_dir, 'runs')
        self.figs_dir = os.path.join(self.log_dir, 'figures')
        self.save_lf_data = save_lf_data
        io.check_else_make_dir(self.runs_dir)
        io.check_else_make_dir(self.figs_dir)

        self.run_steps = None
        self.beta = None
        self.run_data = {}
        self.run_stats = {}
        self.run_strings = [RUN_HEADER]
```

```
        if self.model.save_lf:
            self.samples_arr = []
            self.lf_out = {
                'forward': [],
                'backward': [],
            }
            self.pxs_out = {
                'forward': [],
                'backward': [],
            }
            self.masks = {
                'forward': [],
                'backward': [],
            }
            self.logdets = {
                'forward': [],
                'backward': [],
            }
            self.sumlogdet = {
                'forward': [],
                'backward': [],
            }

    def reset(self, run_steps, beta, **weights):
        """Reset run_data and run_strings to prep for new run."""
        self.run_steps = int(run_steps)
        self.beta = beta

        self.run_data = {
            'px': {},
            'actions': {},
            'plaqs': {},
            'charges': {},
            'charge_diffs': {},
        }
        self.run_stats = {}
        self.run_strings = []
        if self.model.save_lf:
            self.samples_arr = []
            self.lf_out = {
                'forward': [],
                'backward': [],
            }
            self.pxs_out = {
                'forward': [],
                'backward': [],
            }
            self.masks = {
                'forward': [],
                'backward': [],
            }
            self.logdets = {
                'forward': [],
                'backward': [],
            }
            self.sumlogdet = {
                'forward': [],
                'backward': [],
            }

```

```
        eps = self.model.eps
        charge_weight = weights['charge_weight']
        net_weights = weights['net_weights']

        if charge_weight is None:
            charge_weight = 0.
        if net_weights is None:
            net_weights = [1., 1., 1.]

        nw_str = [str(i).replace('.', '') for i in net_weights]
        w_str = nw_str[0] + nw_str[1] + nw_str[2]
        run_str = (f'steps_{run_steps}_beta_{beta}_'
                   f'eps_{eps:.3g}_weights_{w_str}')
        self.run_dir = os.path.join(self.runs_dir, run_str)
        io.check_else_make_dir(self.run_dir)
        save_params(self.model.params, self.run_dir)

        return self.run_dir, run_str

    def update(self, data, data_str):
        """Update run_data and append data_str to data_strings."""
        # projection of samples onto [0, 2pi) done in run_step above
        #  if self.model.save_samples:
        step = data['step']
        beta = data['beta']
        key = (step, beta)
        self.run_data['px'][key] = data['px']
        self.run_data['actions'][key] = data['actions']
        self.run_data['plaqs'][key] = data['plaqs']
        self.run_data['charges'][key] = data['charges']
        self.run_data['charge_diffs'][key] = data['charge_diffs']
        #  self.run_data['charge_weight'][key]

        if self.model.save_lf:
            samples_np = data['samples']
            self.samples_arr.append(samples_np)
            self.lf_out['forward'].extend(np.array(data['lf_out_f']))
            self.lf_out['backward'].extend(np.array(data['lf_out_b']))
            self.logdets['forward'].extend(np.array(data['logdets_f']))
            self.logdets['backward'].extend(np.array(data['logdets_b']))
            self.sumlogdet['forward'].append(np.array(data['sumlogdet_f']))
            self.sumlogdet['backward'].append(np.array(data['sumlogdet_b']))
            #  self.pxs_out['forward'].extend(np.array(data['pxs_out_f']))
            #  self.pxs_out['backward'].extend(np.array(data['pxs_out_b']))
            #  self.masks['forward'].extend(np.array(data['masks_f']))
            #  self.masks['backward'].extend(np.array(data['masks_b']))

        self.run_strings.append(data_str)

        if step % (10 * self.model.print_steps) == 0:
            io.log(data_str)

        if step % 100 == 0:
            io.log(RUN_HEADER)

    def calc_observables_stats(self, run_data, therm_frac=10):
        """Calculate statistics for lattice observables.

        Args:
            run_data: Dictionary of observables data. Keys denote the
```

```
            observables name.
        therm_frac: Fraction of data to throw out for thermalization.

    Returns:
        stats: Dictionary containing statistics for each observable in
        run_data. Additionally, contains `charge_probs` which is a
        dictionary of the form {charge_val: charge_val_probability}.
    """
    def get_stats(data, t_frac=10):
        if isinstance(data, dict):
            arr = np.array(list(data.values()))
        elif isinstance(data, (list, np.ndarray)):
            arr = np.array(data)

        num_steps = arr.shape[0]
        therm_steps = num_steps // t_frac
        arr = arr[therm_steps:, :]
        avg = np.mean(arr, axis=0)
        err = sem(arr, axis=0)
        stats = np.array([avg, err]).T
        return stats

    actions_stats = get_stats(run_data['actions'], therm_frac)
    plaqs_stats = get_stats(run_data['plaqs'], therm_frac)

    charges_arr = np.array(list(run_data['charges'].values()), dtype=int)
    charges_stats = get_stats(charges_arr, therm_frac)

    suscept_arr = charges_arr ** 2
    suscept_stats = get_stats(suscept_arr)

    charge_probs = {}
    counts = Counter(list(charges_arr.flatten()))
    total_counts = np.sum(list(counts.values()))
    for key, val in counts.items():
        charge_probs[key] = val / total_counts

    charge_probs = OrderedDict(sorted(charge_probs.items(),
                                      key=lambda k: k[0]))

    stats = {
        'actions': actions_stats,
        'plaqs': plaqs_stats,
        'charges': charges_stats,
        'suscept': suscept_stats,
        'charge_probs': charge_probs
    }

    return stats

def save_attr(self, name, attr, out_dir=None, dtype=NP_FLOAT):
    if out_dir is None:
        out_dir = self.run_dir

    assert os.path.isdir(out_dir)
    out_file = os.path.join(out_dir, name + '.npz')

    if not isinstance(attr, np.ndarray) or attr.dtype != dtype:
        try:
            attr = np.array(attr, dtype=dtype)
```

```
            except ValueError:
                import pdb
                pdb.set_trace()

        if os.path.isfile(out_file):
            io.log(f'File {out_file} already exists. Skipping.')
        else:
            io.log(f'Saving {name} to: {out_file}')
            np.savez_compressed(out_file, attr)

    def save_run_data(self, therm_frac=10):
        """Save run information."""
        observables_dir = os.path.join(self.run_dir, 'observables')

        io.check_else_make_dir(self.run_dir)
        io.check_else_make_dir(observables_dir)

        #  if self.model.save_lf:
        if self.save_lf_data:
            self.save_attr('samples_out', self.samples_arr)
            self.save_attr('lf_forward', self.lf_out['forward'])
            self.save_attr('lf_backward', self.lf_out['backward'])
            self.save_attr('masks_forward', self.masks['forward'])
            self.save_attr('masks_backward', self.masks['backward'])
            self.save_attr('logdets_forward', self.logdets['forward'])
            self.save_attr('logdets_backward', self.logdets['backward'])
            self.save_attr('sumlogdet_forward', self.sumlogdet['forward'])
            self.save_attr('sumlogdet_backward', self.sumlogdet['backward'])
            self.save_attr('accept_probs_forward', self.pxs_out['forward'])
            self.save_attr('accept_probs_backward', self.pxs_out['backward'])

        run_stats = self.calc_observables_stats(self.run_data, therm_frac)
        charges = self.run_data['charges']
        charges_arr = np.array(list(charges.values()))
        charges_autocorrs = [autocorr(x) for x in charges_arr.T]
        charges_autocorrs = [x / np.max(x) for x in charges_autocorrs]
        self.run_data['charges_autocorrs'] = charges_autocorrs

        data_file = os.path.join(self.run_dir, 'run_data.pkl')
        io.log(f"Saving run_data to: {data_file}.")
        with open(data_file, 'wb') as f:
            pickle.dump(self.run_data, f)

        stats_data_file = os.path.join(self.run_dir, 'run_stats.pkl')
        io.log(f"Saving run_stats to: {stats_data_file}.")
        with open(stats_data_file, 'wb') as f:
            pickle.dump(run_stats, f)

        for key, val in self.run_data.items():
            out_file = key + '.pkl'
            out_file = os.path.join(observables_dir, out_file)
            io.save_data(val, out_file, name=key)

        for key, val in run_stats.items():
            out_file = key + '_stats.pkl'
            out_file = os.path.join(observables_dir, out_file)
            io.save_data(val, out_file, name=key)

        history_file = os.path.join(self.run_dir, 'run_history.txt')
        io.write(RUN_HEADER, history_file, 'w')
```

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

        self.write_run_stats(run_stats, therm_frac)

    def write_run_stats(self, stats, therm_frac=10):
        """Write statistics in human readable format to .txt file."""
        #  run_steps = kwargs['run_steps']
        #  beta = kwargs['beta']
        #  current_step = kwargs['current_step']
        #  therm_steps = kwargs['therm_steps']
        #  training = kwargs['training']
        #  run_dir = kwargs['run_dir']
        therm_steps = self.run_steps // therm_frac

        out_file = os.path.join(self.run_dir, 'run_stats.txt')

        actions_avg, actions_err = stats['actions'].mean(axis=0)
        plaqs_avg, plaqs_err = stats['plaqs'].mean(axis=0)
        charges_avg, charges_err = stats['charges'].mean(axis=0)
        suscept_avg, suscept_err = stats['suscept'].mean(axis=0)

        ns = self.model.num_samples
        suscept_k1 = f'  \navg. over all {ns} samples < Q >'
        suscept_k2 = f'  \navg. over all {ns} samples < Q^2 >'
        actions_k1 = f'  \navg. over all {ns} samples < action >'
        plaqs_k1 = f'  \n avg. over all {ns} samples < plaq >'

        _est_key = '  \nestimate +/- stderr'

        suscept_ss = {
            suscept_k1: f"{charges_avg:.4g} +/- {charges_err:.4g}",
            suscept_k2: f"{suscept_avg:.4g} +/- {suscept_err:.4g}",
            _est_key: {}
        }

        actions_ss = {
            actions_k1: f"{actions_avg:.4g} +/- {actions_err:.4g}\n",
            _est_key: {}
        }

        plaqs_ss = {
            'exact_plaq': f"{u1_plaq_exact(self.beta):.4g}\n",
            plaqs_k1: f"{plaqs_avg:.4g} +/- {plaqs_err:.4g}\n",
            _est_key: {}
        }

        def format_stats(x, name=None):
            return [f'{name}: {i[0]:.4g} +/- {i[1]:.4g}' for i in x]

        def zip_keys_vals(stats_strings, keys, vals):
            for k, v in zip(keys, vals):
                stats_strings[_est_key][k] = v
            return stats_strings

        keys = [f"sample {idx}" for idx in range(ns)]

        suscept_vals = format_stats(stats['suscept'], '< Q^2 >')
        actions_vals = format_stats(stats['actions'], '< action >')
        plaqs_vals = format_stats(stats['plaqs'], '< plaq >')

```

```
        suscept_ss = zip_keys_vals(suscept_ss, keys, suscept_vals)
        actions_ss = zip_keys_vals(actions_ss, keys, actions_vals)
        plaqs_ss = zip_keys_vals(plaqs_ss, keys, plaqs_vals)

        def accumulate_strings(d):
            all_strings = []
            for k1, v1 in d.items():
                if isinstance(v1, dict):
                    for k2, v2 in v1.items():
                        all_strings.append(f'{k2} {v2}')
                else:
                    all_strings.append(f'{k1}: {v1}\n')

            return all_strings

        actions_strings = accumulate_strings(actions_ss)
        plaqs_strings = accumulate_strings(plaqs_ss)
        suscept_strings = accumulate_strings(suscept_ss)

        charge_probs_strings = []
        for k, v in stats['charge_probs'].items():
            charge_probs_strings.append(f'  probability[Q = {k}]: {v}\n')

        #  train_str = (f" stats after {current_step} training steps.\n"
        #               f"{ns} chains ran for {self.run_steps} steps at "
        #               f"beta = {self.beta}.")

        run_str = (f" stats for {ns} chains ran for {self.run_steps} steps "
                   f" at beta = {self.beta}.")

        #  if training:
        #      str0 = "Topological susceptibility" + train_str
        #      str1 = "Total actions" + train_str
        #      str2 = "Average plaquette" + train_str
        #      str3 = "Topological charge probabilities" + train_str[6:]
        #      therm_str = ''
        #  else:
        str0 = "Topological susceptibility" + run_str
        str1 = "Total actions" + run_str
        str2 = "Average plaquette" + run_str
        str3 = "Topological charge probabilities" + run_str[6:]
        therm_str = (
            f'Ignoring first {therm_steps} steps for thermalization.'
        )

        ss0 = (1 + max(len(str0), len(therm_str))) * '-'
        ss1 = (1 + max(len(str1), len(therm_str))) * '-'
        ss2 = (1 + max(len(str2), len(therm_str))) * '-'
        ss3 = (1 + max(len(str3), len(therm_str))) * '-'

        io.log(f"Writing statistics to: {out_file}")

        def log_and_write(sep_str, str0, therm_str, stats_strings, file):
            io.log(sep_str)
            io.log(str0)
            io.log(therm_str)
            io.log('')
            _ = [io.log(s) for s in stats_strings]
            io.log(sep_str)
            io.log('')
```

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

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

```python
"""
train_logger.py

Implements TrainLogger class responsible for saving/logging data from
GaugeModel.

Author: Sam Foreman (github: @saforem2)
Date: 04/09/2019
"""
import os
import pickle

import utils.file_io as io

from globals import TRAIN_HEADER
from utils.tf_logging import variable_summaries  # add_loss_summaries

import tensorflow as tf


def save_params(params, out_dir):
    io.check_else_make_dir(out_dir)
    params_txt_file = os.path.join(out_dir, 'parameters.txt')
    params_pkl_file = os.path.join(out_dir, 'parameters.pkl')
    with open(params_txt_file, 'w') as f:
        for key, val in params.items():
            f.write(f"{key}: {val}\n")
    with open(params_pkl_file, 'wb') as f:
        pickle.dump(params, f)


class TrainLogger:
    def __init__(self, model, log_dir, summaries=False):
        #  self.sess = sess
        self.model = model
        self.summaries = summaries

        self.charges_dict = {}
        self.charge_diffs_dict = {}

        self._current_state = {
            'step': 0,
            'beta': self.model.beta_init,
            'eps': self.model.eps,
            'lr': self.model.lr_init,
            'samples': self.model.samples,
        }

        self.train_data_strings = [TRAIN_HEADER]
        self.train_data = {
            'loss': {},
            'actions': {},
            'plaqs': {},
            'charges': {},
            'charge_diffs': {},
            'accept_probs': {}
        }
```

```
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
```

```
        with tf.name_scope('tunneling_events'):
            tf.summary.scalar('tunneling_events_per_sample',
                              self.model.charge_diffs_op)

        with tf.name_scope('avg_plaq'):
            tf.summary.scalar('avg_plaq', self.model.avg_plaqs_op)

        #  for var in tf.trainable_variables():
        #  for var in self.model.dynamics.trainable_variables():
        #      if 'batch_normalization' not in var.op.name:
        #          tf.summary.histogram(var.op.name, var)

        with tf.name_scope('summaries'):
            for grad, var in grads_and_vars:
                try:
                    layer, _type = var.name.split('/')[-2:]
                    name = layer + '_' + _type[:-2]
                except (AttributeError, IndexError):
                    name = var.name[:-2]

                if 'batch_norm' not in name:
                    variable_summaries(var, name)
                    variable_summaries(grad, name + '/gradients')
                    tf.summary.histogram(name + '/gradients', grad)

        self.summary_op = tf.summary.merge_all(name='summary_op')

    def save_current_state(self):
        """Save current state to pickle file.

        The current state contains the following, which makes life easier if
        we're trying to restore training from a saved checkpoint:
            * most recent samples
            * learning_rate
            * beta
            * dynamics.eps
            * training_step
        """
        with open(self.current_state_file, 'wb') as f:
            pickle.dump(self._current_state, f)

    def log_step(self, sess, step, samples_np, beta_np, **weights):
        """Update self.logger.summaries."""
        net_weights = weights['net_weights']
        charge_weight = weights['charge_weight']

        feed_dict = {
            self.model.x: samples_np,
            self.model.beta: beta_np,
            self.model.charge_weight: charge_weight,
            self.model.net_weights[0]: net_weights[0],
            self.model.net_weights[1]: net_weights[1],
            self.model.net_weights[2]: net_weights[2],
        }
        summary_str = sess.run(self.summary_op, feed_dict=feed_dict)

        self.writer.add_summary(summary_str, global_step=step)
        self.writer.flush()

    def update_training(self, sess, data, data_str, **weights):
```

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

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

```python
"""
plotters.py

Implements GaugeModelPlotter class, responsible for loading and plotting
gauge model observables.

Author: Sam Foreman (github: @saforem2)
Date: 04/10/2019
"""
import os
#  import matplotlib as mpl
import numpy as np
import utils.file_io as io

from collections import Counter, OrderedDict
from scipy.stats import sem

from lattice.lattice import u1_plaq_exact
#  from utils.data_loader import DataLoader
from globals import MARKERS, COLORS
from .plot_utils import plot_multiple_lines

try:
    import matplotlib.pyplot as plt
    HAS_MATPLOTLIB = True

except ImportError:
    HAS_MATPLOTLIB = False


def arr_from_dict(d, key):
    return np.array(list(d[key].values()))


def get_out_file(out_dir, out_str):
    out_file = os.path.join(out_dir, out_str + '.pdf')
    return out_file


class GaugeModelPlotter:

    def __init__(self, model, figs_dir=None):
        self.model = model
        self.figs_dir = figs_dir

    def calc_stats(self, data, therm_frac=10):
        """Calculate observables statistics.

        Args:
            data (dict): Run data.
            therm_frac (int): Percent of total steps to ignore to account for
            thermalization.

        Returns:
            stats: Dictionary containing statistics for actions, plaquettes,
            top. charges, and charge probabilities. For each of the
            observables (actions, plaquettes, charges), the dictionary values
```

```
            consist of a tuple of the form: (data, error), and
            charge_probabilities is a dictionary of the form:
                {charge_val: charge_val_probability}
        """
        actions = arr_from_dict(data, 'actions')
        plaqs = arr_from_dict(data, 'plaqs')
        charges = arr_from_dict(data, 'charges')

        charge_probs = {}
        counts = Counter(list(charges.flatten()))
        total_counts = np.sum(list(counts.values()))
        for key, val in counts.items():
            charge_probs[key] = val / total_counts

        charge_probs = OrderedDict(sorted(charge_probs.items(),
                                          key=lambda k: k[0]))

        def get_mean_err(x):
            num_steps = x.shape[0]
            therm_steps = num_steps // therm_frac
            x = x[therm_steps:, :]
            avg = np.mean(x, axis=0)
            err = sem(x)
            return avg, err

        stats = {
            'actions': get_mean_err(actions),
            'plaqs': get_mean_err(plaqs),
            'charges': get_mean_err(charges),
            'suscept': get_mean_err(charges ** 2),
            'charge_probs': charge_probs
        }

        return stats

    def plot_observables(self, data, beta, run_str, **weights):
        """Plot observables."""
        actions = arr_from_dict(data, 'actions')
        plaqs = arr_from_dict(data, 'plaqs')
        charges = np.array(arr_from_dict(data, 'charges'), dtype=int)
        charge_diffs = arr_from_dict(data, 'charge_diffs')
        charge_autocorrs = np.array(data['charges_autocorrs'])
        plaqs_diffs = plaqs - u1_plaq_exact(beta)

        num_steps, num_samples = actions.shape
        steps_arr = np.arange(num_steps)
        # skip 5% of total number of steps
        # between successive points when
        # plotting to help smooth out graph
        skip_steps = max((1, int(0.005 * num_steps)))
        # ignore first 10% of pts (warmup)
        warmup_steps = max((1, int(0.01 * num_steps)))

        #  _actions = actions[::skip_steps]
        #  _plaqs = plaqs[::skip_steps]

        #  _steps_arr = skip_steps * np.arange(_actions.shape[0])
        _charge_diffs = charge_diffs[warmup_steps:][::skip_steps]
        _plaqs_diffs = plaqs_diffs[warmup_steps:][::skip_steps]
        _steps_diffs = (
```

```
            skip_steps * np.arange(_plaqs_diffs.shape[0]) + skip_steps
        )
        #  steps_diffs = np.arange(plaqs_diffs.shape[0])

        #  out_dir = (f'{int(num_steps)}_steps_' f"beta_{beta}")
        #  self.out_dir = os.path.join(self.figs_dir, out_dir)
        #  self.out_dir = os.path.join
        self.out_dir = os.path.join(self.figs_dir, run_str)
        io.check_else_make_dir(self.out_dir)

        L = self.model.space_size
        lf_steps = self.model.num_steps
        bs = self.model.num_samples
        qw = weights['charge_weight']
        #  nw = weights['net_weight']

        title_str = (r"$L = $" + f"{L}, "
                     r"$N_{\mathrm{LF}} = $" + f"{lf_steps}, "
                     r"$\alpha_{Q} = $" + f"{qw}, "
                     r"$\beta = $" + f"{beta}, "
                     r"$N_{\mathrm{samples}} = $" + f"{bs}")

        kwargs = {
            'markers': False,
            'lines': True,
            'alpha': 0.6,
            'title': title_str,
            'legend': True,
            'ret': False,
            'out_file': [],
        }

        #  self._plot_actions((_steps_arr, _actions.T), **kwargs)
        #  self._plot_plaqs((_steps_arr, _plaqs.T), beta, **kwargs)
        self._plot_actions((steps_arr, actions.T), **kwargs)
        self._plot_plaqs((steps_arr, plaqs.T), beta, **kwargs)
        self._plot_charges((steps_arr, charges.T), **kwargs)
        self._plot_autocorrs((steps_arr, charge_autocorrs), **kwargs)
        self._plot_charge_probs(charges, **kwargs)
        self._plot_charge_diffs((_steps_diffs, _charge_diffs.T), **kwargs)
        self._plot_plaqs_diffs((_steps_diffs, _plaqs_diffs.T), **kwargs)
        #  self._plot_charge_diffs((steps_diffs, charge_diffs.T), **kwargs)
        #  self._plot_plaqs_diffs((steps_diffs, plaqs_diffs.T), **kwargs)

    def _plot_actions(self, xy_data, **kwargs):
        """Plot actions."""
        kwargs['out_file'] = get_out_file(self.out_dir, 'actions_vs_step')
        xy_labels = ('Step', 'Action')
        plot_multiple_lines(xy_data, xy_labels, **kwargs)

    def _plot_plaqs(self, xy_data, beta, **kwargs):
        """PLot average plaquette."""
        kwargs['out_file'] = None
        kwargs['ret'] = True
        xy_labels = ('Step', r"""$\langle \phi_{P} \rangle$""")

        _, ax = plot_multiple_lines(xy_data, xy_labels, **kwargs)
        _ = ax.axhline(y=u1_plaq_exact(beta),
                       color='#CC0033', ls='-', lw=1.5, label='exact')
        #  _ = ax.plot(xy_data[0], xy_data[1].mean(axis=0), lw=1.25,
```

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

```
            out_file = out_file.rstrip('.pdf') + '_zoom.pdf'
            io.log(f'Saving figure to: {out_file}')
            plt.savefig(out_file, dpi=400, bbox_inches='tight')

        plt.close('all')

    def _plot_charge_diffs(self, xy_data, **kwargs):
        """Plot tunneling events (change in top. charge)."""
        out_file = get_out_file(self.out_dir, 'top_charge_diffs')
        steps_arr, charge_diffs = xy_data

        # ignore first two data points when plotting since the top. charge
        # should change dramatically for the very first vew steps when starting
        # from a random configuration
        _, ax = plt.subplots()
        _ = ax.plot(xy_data[0][2:], xy_data[1][2:],
                    marker='.', ls='', fillstyle='none', color='C0')
        _ = ax.set_xlabel('Steps', fontsize=14)
        _ = ax.set_ylabel(r'$\delta_{Q}$', fontsize=14)
        _ = ax.set_title(kwargs['title'], fontsize=16)
        _ = plt.tight_layout()
        io.log(f"Saving figure to: {out_file}")
        plt.savefig(out_file, dpi=400, bbox_inches='tight')

    def _plot_charge_probs(self, charges, **kwargs):
        """PLot top. charge probabilities."""
        num_steps, num_samples = charges.shape
        charges = np.array(charges, dtype=int)
        out_dir = os.path.join(self.out_dir, 'top_charge_probs')
        io.check_else_make_dir(out_dir)
        if 'title' in list(kwargs.keys()):
            title = kwargs.pop('title')
        # if we have more than 10 chains in charges, only plot first 10
        for idx in range(min(num_samples, 5)):
            counts = Counter(charges[:, idx])
            total_counts = np.sum(list(counts.values()))
            _, ax = plt.subplots()
            ax.plot(list(counts.keys()),
                    np.array(list(counts.values()) / total_counts),
                    marker=MARKERS[idx],
                    color=COLORS[idx],
                    ls='',
                    label=f'sample {idx}')
            _ = ax.legend(loc='best')
            _ = ax.set_xlabel(r"$Q$")  # , fontsize=14)
            _ = ax.set_ylabel('Probability')  # , fontsize=14)
            _ = ax.set_title(title)  # , fontsize=16)
            _ = plt.tight_layout()
            out_file = get_out_file(out_dir, f'top_charge_vs_step_{idx}')
            io.check_else_make_dir(os.path.dirname(out_file))
            io.log(f"Saving plot to: {out_file}.")
            plt.savefig(out_file, dpi=400, bbox_inches='tight')
            #  for f in out_file:
            #      io.check_else_make_dir(os.path.dirname(f))
            #      io.log(f"Saving plot to: {f}.")
        plt.close('all')

        all_counts = Counter(list(charges.flatten()))
        total_counts = np.sum(list(counts.values()))
        _, ax = plt.subplots()
```

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

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

```python
import time

import os
try:
    import psutil
    HAS_PSUTIL = True
except ImportError:
    HAS_PSUTIL = False

import pickle
import numpy as np
try:
    import matplotlib as mpl
    import matplotlib.pyplot as plt
    HAS_MATPLOTLIB = True
except ImportError:
    HAS_MATPLOTLIB = False

#  from .formatters import latexify
import utils.file_io as io
from utils.data_loader import DataLoader


def load_and_sep(out_file, keys=('forward', 'backward')):
    with open(out_file, 'rb') as f:
        data = pickle.load(f)

    return (np.array(data[k]) for k in keys)


params = {
    #  'backend': 'ps',
    #  'text.latex.preamble': [r'\usepackage{gensymb}'],
    'axes.labelsize': 16,   # fontsize for x and y labels (was 10)
    'axes.titlesize': 16,
    'legend.fontsize': 10,  # was 10
    'xtick.labelsize': 12,
    'ytick.labelsize': 12,
    #  'text.usetex': True,
    #  'figure.figsize': [fig_width, fig_height],
    'font.family': 'serif'
}

try:
    mpl.rcParams.update(params)
except FileNotFoundError:
    params['text.usetex'] = False
    params['text.latex.preamble'] = None
    try:
        mpl.rcParams.update(params)
    except FileNotFoundError:
        pass


class LeapfrogPlotter:
    def __init__(self, figs_dir, run_logger=None,
                 run_dir=None, therm_perc=0.005, skip_perc=0.01):
```

```
        self.figs_dir = figs_dir
        self.pdfs_dir = os.path.join(self.figs_dir, 'pdfs_plots')
        io.check_else_make_dir(self.pdfs_dir)

        if run_logger is None:
            if run_dir is None:
                raise AttributeError(
                    """Either a `run_logger` object containing data or a
                    `run_dir` from which to load data must be specified.
                    Exiting.
                    """
                )
            else:
                try:
                    data = self.load_data(run_dir)
                    self.samples = data[0]
                    self.lf_f, self.lf_b = data[1]
                    self.logdets_f, self.logdets_b = data[2]
                    self.sumlogdet_f, self.sumlogdet_b = data[3]
                except FileNotFoundError:
                    io.log(f'''Unable to load leapfrog data from run_dir:
                           {run_dir}. Exiting.''')
                    return

        else:
            self.samples = np.array(run_logger.samples_arr)
            self.lf_f = np.array(run_logger.lf_out['forward'])
            self.lf_b = np.array(run_logger.lf_out['backward'])
            self.logdets_f = np.array(run_logger.logdets['forward'])
            self.logdets_b = np.array(run_logger.logdets['backward'])
            self.sumlogdet_f = np.array(run_logger.sumlogdet['forward'])
            self.sumlogdet_b = np.array(run_logger.sumlogdet['backward'])

        self.lf_f_diffs = 1. - np.cos(self.lf_f[1:] - self.lf_f[:-1])
        self.lf_b_diffs = 1. - np.cos(self.lf_b[1:] - self.lf_b[:-1])
        self.samples_diffs = 1. - np.cos(self.samples[1:] - self.samples[:-1])
        self.tot_lf_steps = self.lf_f_diffs.shape[0]
        self.tot_md_steps = self.samples_diffs.shape[0]
        self.num_lf_steps = self.tot_lf_steps // self.tot_md_steps
        self.indiv_kwargs = {
            'ls': '-',
            'alpha': 0.5,
            'lw': 0.5,
            'rasterized': True
        }

        self.avg_kwargs = {
            'ls': '-',
            'alpha': 0.6,
            'lw': 0.75,
            'rasterized': True
        }
        plot_lf_steps = self.tot_lf_steps // 4
        plot_md_steps = self.tot_md_steps // 4
        self.lf_f_diffs = self.lf_f_diffs[:plot_lf_steps]
        self.lf_b_diffs = self.lf_b_diffs[:plot_lf_steps]
        self.samples_diffs = self.samples_diffs[:plot_md_steps]
        self.logdets_f = self.logdets_f[:plot_lf_steps]
        self.logdets_b = self.logdets_b[:plot_lf_steps]
        self.sumlogdet_f = self.sumlogdet_f[:plot_md_steps]
```

```
        self.sumlogdet_b = self.sumlogdet_b[:plot_md_steps]

        #  self.therm_steps = int(therm_perc * self.tot_lf_steps)
        #  self.skip_steps = int(skip_perc * self.tot_lf_steps)
        #  self.step_multiplier = (
        #      self.lf_f_diffs.shape[0] // self.samples_diffs.shape[0]
        #  )

    def load_data(self, run_dir):
        loader = DataLoader(run_dir)
        io.log("Loading samples...")
        samples = loader.load_samples(run_dir)
        io.log('done.')
        io.log("Loading leapfrogs...")
        leapfrogs = loader.load_leapfrogs(run_dir)
        io.log("Loading logdets...")
        logdets = loader.load_logdets(run_dir)
        io.log("Loading sumlogdets...")
        sumlogdets = loader.load_sumlogdets(run_dir)
        return (samples, leapfrogs, logdets, sumlogdets)

    def print_memory(self):
        if HAS_PSUTIL:
            pid = os.getpid()
            py = psutil.Process(pid)
            memory_use = py.memory_info()[0] / 2. ** 30
            io.log(80 * '-')
            io.log(f'memory use: {memory_use}')
            io.log(80 * '-')

    def get_colors(self, num_samples=20):
        reds_cmap = mpl.cm.get_cmap('Reds', num_samples + 1)
        blues_cmap = mpl.cm.get_cmap('Blues', num_samples + 1)
        idxs = np.linspace(0.2, 0.75, num_samples + 1)
        reds = [reds_cmap(i) for i in idxs]
        blues = [blues_cmap(i) for i in idxs]

        return reds, blues

    def update_figs_dir(self, figs_dir):
        self.figs_dir = figs_dir
        self.pdfs_dir = os.path.join(self.figs_dir, 'pdfs')

        io.check_else_make_dir(figs_dir)
        io.check_else_make_dir(self.pdfs_dir)

    def make_plots(self, run_dir, num_samples=20, ret=False):
        """Make plots of the leapfrog differences and logdets.

        Immediately after creating and saving the plots, delete these
        (no-longer needed) attributes to free up memory.

        Args:
            run_dir (str): Path to directory in which to save all of the
                relevant instance attributes.
            num_samples (int): Number of samples to include when creating
                plots.
            save (bool): Boolean indicating whether or not plotted data should
                be saved.

```

```
        NOTE:
            `save` is very data intensive and will produce LARGE (compressed)
            `.npz` files.
        """
        run_key = run_dir.split('/')[-1].split('_')
        beta_idx = run_key.index('beta') + 1
        beta = run_key[beta_idx]

        self.print_memory()
        fig_ax1 = self.plot_lf_diffs(beta, num_samples)

        self.print_memory()
        fig_ax2 = self.plot_logdets(beta, num_samples)

        if ret:
            return fig_ax1, fig_ax2

    def plot_lf_diffs(self, beta, num_samples=20):
        t0 = time.time()
        reds, blues = self.get_colors(num_samples)
        samples_y_avg = np.mean(self.samples_diffs, axis=(1, 2))
        samples_x_avg = np.arange(len(samples_y_avg))

        fig, (ax1, ax2) = plt.subplots(2, 1)
        ax1.set_rasterization_zorder(1)
        ax2.set_rasterization_zorder(1)

        for idx in range(num_samples):
            yf = np.mean(self.lf_f_diffs, axis=-1)
            xf = np.arange(len(yf))
            yb = np.mean(self.lf_b_diffs, axis=-1)
            xb = np.arange(len(yb))

            _ = ax1.plot(xf, yf[:, idx], color=reds[idx], **self.indiv_kwargs)
            _ = ax1.plot(xb, yb[:, idx], color=blues[idx], **self.indiv_kwargs)

        yf_avg = np.mean(self.lf_f_diffs, axis=(1, 2))
        yb_avg = np.mean(self.lf_b_diffs, axis=(1, 2))
        xf_avg = np.arange(len(yf))
        xb_avg = np.arange(len(yb))

        _ = ax1.plot(xf_avg, yf_avg, label='forward',
                     color='r', **self.avg_kwargs)
        _ = ax1.plot(xb_avg, yb_avg, label='backward',
                     color='b', **self.avg_kwargs)

        _ = ax2.plot(samples_x_avg, samples_y_avg, label='MD avg.',
                     color='k', lw=1., rasterized=True, ls='-')

        _ = ax1.set_xlabel('Leapfrog step', fontsize=16)
        _ = ax2.set_xlabel('MD step', fontsize=16)

        ylabel = r'$\langle \delta\phi_{\mu}(i)\rangle$'
        _ = ax1.set_ylabel(ylabel, fontsize=16)
        _ = ax2.set_ylabel(ylabel, fontsize=16)

        _ = ax1.legend(loc='best', fontsize=10)
        _ = ax2.legend(loc='best', fontsize=10)
        fig.tight_layout()
        #  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',
```

```
                    color='r', lw=1.2, alpha=0.9, marker='.')

        _ = ax2.plot(sumlogdet_xb_avg, sumlogdet_yb_avg, label='backward',
                    color='b', lw=1.2, alpha=0.9, marker='.')

        _ = ax1.set_xlabel('Leapfrog step', fontsize=16)
        _ = ax1.set_ylabel(r'$\log|\mathcal{J}^{(t)}|$', fontsize=16)
        _ = ax2.set_xlabel('MD step', fontsize=16)
        _ = ax2.set_ylabel(r'$\sum_{t}\log|\mathcal{J}^{(t)}|$',
                           fontsize=16)
        _ = ax1.legend(loc='best', fontsize=10)
        _ = ax2.legend(loc='best', fontsize=10)
        _ = fig.tight_layout()
        beta_str = str(beta).replace('.', '')
        fn = f'avg_logdets_beta{beta_str}'
        out_file = os.path.join(self.pdfs_dir, fn + '.pdf')
        out_file_zoom = os.path.join(self.pdfs_dir, fn + '_zoom.pdf')
        out_file_zoom1 = os.path.join(self.pdfs_dir, fn + '_zoom1.pdf')
        io.log(f'Saving figure to: {out_file}')
        _ = plt.savefig(out_file, dpi=400, bbox_inches='tight')

        lf_xlim = 100
        md_xlim = lf_xlim // self.num_lf_steps

        _ = ax1.set_xlim((0, lf_xlim))
        _ = ax2.set_xlim((0, md_xlim))
        _ = plt.savefig(out_file_zoom, dpi=400, bbox_inches='tight')

        lf_xlim //= 4
        md_xlim //= 4
        _ = ax1.set_xlim((0, lf_xlim))
        _ = ax2.set_xlim((0, md_xlim))
        _ = plt.savefig(out_file_zoom1, dpi=400, bbox_inches='tight')

        io.log(80 * '-')
        io.log(f'Time spent plotting log_dets: {time.time() - t0}s')
        io.log(80 * '-')

        return fig, (ax1, ax2)
```

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

```python
"""
plot_utils.py

Collection of helper methods used for creating plots in Matplotlib.

Author: Sam Foreman (github: @saforem2)
Date: 06/16/2019
"""
import os

import numpy as np
import matplotlib as mpl
try:
    import matplotlib.pyplot as plt
    HAS_MATPLOTLIB = True
except ImportError:
    HAS_MATPLOTLIB = False

from globals import MARKERS
import utils.file_io as io


params = {
    #  'backend': 'ps',
    #  'text.latex.preamble': [r'\usepackage{gensymb}'],
    'axes.labelsize': 14,   # fontsize for x and y labels (was 10)
    'axes.titlesize': 16,
    'legend.fontsize': 10,  # was 10
    'xtick.labelsize': 12,
    'ytick.labelsize': 12,
    #  'text.usetex': True,
    #  'figure.figsize': [fig_width, fig_height],
    #  'font.family': 'serif',
}

mpl.rcParams.update(params)


def get_colors(num_samples=10, cmaps=None):
    if cmaps is None:
        cmap0 = mpl.cm.get_cmap('Greys', num_samples + 1)
        cmap1 = mpl.cm.get_cmap('Reds', num_samples + 1)
        cmap2 = mpl.cm.get_cmap('Blues', num_samples + 1)
        cmaps = (cmap0, cmap1, cmap2)

    idxs = np.linspace(0.1, 0.75, num_samples + 1)
    colors_arr = []
    for cmap in cmaps:
        colors_arr.append([cmap(i) for i in idxs])

        #  colors0 = [cmap0(i) for i in idxs]
        #  colors1 = [cmap1(i) for i in idxs]
        #  cmap0 = mpl.cm.get_cmap(cmaps[0], num_samples + 1)
        #  cmap1 = mpl.cm.get_cmap(cmaps[1], num_samples + 1)
    #  reds_cmap = mpl.cm.get_cmap('Reds', num_samples + 1)
    #  blues_cmap = mpl.cm.get_cmap('Blues', num_samples + 1)

```

```
    #  return colors0, colors1
    return colors_arr


def plot_multiple_lines(data, xy_labels, **kwargs):
    """Plot multiple lines along with their average."""
    out_file = kwargs.get('out_file', None)
    markers = kwargs.get('markers', False)
    lines = kwargs.get('lines', True)
    alpha = kwargs.get('alpha', 1.)
    legend = kwargs.get('legend', False)
    title = kwargs.get('title', None)
    ret = kwargs.get('ret', False)
    num_samples = kwargs.get('num_samples', 10)
    greys, reds, blues = get_colors(num_samples)
    if isinstance(data, list):
        data = np.array(data)

    try:
        x_data, y_data = data
    except (IndexError, ValueError):
        x_data = np.arange(data.shape[0])
        y_data = data

    x_label, y_label = xy_labels

    if y_data.shape[0] > num_samples:
        y_sample = y_data[:num_samples, :]
    else:
        y_sample = y_data

    fig, ax = plt.subplots()

    marker = None
    ls = '-'
    fillstyle = 'full'
    for idx, row in enumerate(y_sample):
        if markers:
            marker = MARKERS[idx]
            fillstyle = 'none'
            ls = '-'
        if not lines:
            ls = ''
        _ = ax.plot(x_data, row,  # label=f'sample {idx}',
                    fillstyle=fillstyle, marker=marker,
                    ls=ls, lw=0.5, alpha=alpha, color=greys[idx])

    _ = ax.plot(
        x_data, y_data.mean(axis=0), label='average',
        alpha=1., lw=1.0, color='k'
    )

    ax.set_xlabel(x_label, fontsize=14)
    ax.set_ylabel(y_label, fontsize=14)
    plt.tight_layout()
    if legend:
        ax.legend(loc='best')
    if title is not None:
        ax.set_title(title)
    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

```
"""
parse_args.py

Implements method for parsing command line arguments for `gauge_model.py`

Author: Sam Foreman (github: @saforem2)
Date: 04/09/2019
"""
import os
import sys
import argparse
import shlex

import utils.file_io as io

#  from config import process_config
#  from attr_dict import AttrDict

DESCRIPTION = (
    'L2HMC model using U(1) lattice gauge theory for target distribution.'
)


def get_args():
    argparser = argparse.ArgumentParser(description=DESCRIPTION)
    argparser.add_argument(
        '-c', '--config',
        metavar='C',
        default=None,
        help='The configuration file'
    )
    args = argparser.parse_args()

    return args


# ============================================================================
#  * NOTE:
#      - if action == 'store_true':
#          The argument is FALSE by default. Passing this flag will cause the
#          argument to be ''stored true''.
#      - if action == 'store_false':
#          The argument is TRUE by default. Passing this flag will cause the
#          argument to be ''stored false''.
# ============================================================================
def parse_args():
    """Parse command line arguments."""
    parser = argparse.ArgumentParser(
        description=DESCRIPTION,
        fromfile_prefix_chars='@',
    )

    # ------------------------------------------------------------------------
    # Lattice parameters
    # ------------------------------------------------------------------------

    parser.add_argument("--space_size",
```

```
                        dest="space_size",
                        type=int,
                        default=8,
                        required=False,
                        help="""Spatial extent of lattice.""")

    parser.add_argument("--time_size",
                        dest="time_size",
                        type=int,
                        default=8,
                        required=False,
                        help="""Temporal extent of lattice.""")

    parser.add_argument("--link_type",
                        dest="link_type",
                        type=str,
                        required=False,
                        default='U1',
                        help="""Link type for gauge model.""")

    parser.add_argument("--dim",
                        dest="dim",
                        type=int,
                        required=False,
                        default=2,
                        help="""Dimensionality of lattice.""")

    parser.add_argument("--num_samples",
                        dest="num_samples",
                        type=int,
                        default=10,
                        required=False,
                        help=("""Number of samples (batch size) to use for
                              training.""""))

    parser.add_argument("--rand",
                        dest="rand",
                        action="store_true",
                        required=False,
                        help=("""Start lattice from randomized initial
                              configuration.""""))

    # ---------------------------------------------------------------------
    # Leapfrog parameters
    # ---------------------------------------------------------------------

    parser.add_argument("-n", "--num_steps",
                        dest="num_steps",
                        type=int,
                        default=5,
                        required=False,
                        help=("""Number of leapfrog steps to use in (augmented)
                              HMC sampler.""""))

    parser.add_argument("--for_loop",
                        dest="for_loop",
                        action="store_true",
                        required=False,
                        help=("""When passed, perform leapfrog updates using
                              generic `for` loop. Otherwise, use (optimized)
```

```
                            `tf.while_loop`."""))

    parser.add_argument("--eps",
                        dest="eps",
                        type=float,
                        default=0.1,
                        required=False,
                        help="""Step size to use in leapfrog integrator.""")

    parser.add_argument("--loss_scale",
                        dest="loss_scale",
                        type=float,
                        default=1.,
                        required=False,
                        help=("""Scaling factor to be used in loss function.
                              (lambda in Eq. 7 of paper)""".""))

    # ------------------------------------------------------------------------
    # Learning rate parameters
    # ------------------------------------------------------------------------

    parser.add_argument("--lr_init",
                        dest="lr_init",
                        type=float,
                        default=1e-3,
                        required=False,
                        help="""Initial value of learning rate.""")

    parser.add_argument("--lr_decay_steps",
                        dest="lr_decay_steps",
                        type=int, default=500,
                        required=False,
                        help=("""Number of steps after which to decay learning
                              rate.""".""))

    parser.add_argument("--lr_decay_rate",
                        dest="lr_decay_rate",
                        type=float, default=0.96,
                        required=False,
                        help=("""Learning rate decay rate to be used during
                              training.""".""))

    # ------------------------------------------------------------------------
    # Annealing rate parameters
    # ------------------------------------------------------------------------

    parser.add_argument("--annealing",
                        dest="annealing",
                        action="store_true",
                        required=False,
                        help=("""Flag that when passed will cause the model
                              to perform simulated annealing during
                              training.""".""))

    parser.add_argument("--hmc_beta",
                        dest="hmc_beta",
                        type=float,
                        default=None,
                        required=False,
                        help=("""Flag specifying a singular value of beta at
```

```
                            which to run the generic HMC sampler."""))

    parser.add_argument("--hmc_eps",
                        dest="hmc_eps",
                        type=float,
                        default=None,
                        required=False,
                        help=("""Flag specifying a singular step size value
                              (`eps`) to use when running the generic HMC
                              sampler.""""))

    parser.add_argument("--beta_init",
                        dest="beta_init",
                        type=float,
                        default=1.,
                        required=False,
                        help=("""Initial value of beta (inverse coupling
                              constant) used in gauge model when
                              annealing.""""))

    parser.add_argument("--beta_final",
                        dest="beta_final",
                        type=float,
                        default=8.,
                        required=False,
                        help=("""Final value of beta (inverse coupling
                              constant) used in gauge model when
                              annealing.""""))

    # ------------------------------------------------------------------------
    # Training parameters
    # ------------------------------------------------------------------------

    parser.add_argument("--train_steps",
                        dest="train_steps",
                        type=int,
                        default=5000,
                        required=False,
                        help="""Number of training steps to perform.""")

    parser.add_argument("--run_steps",
                        dest="run_steps",
                        type=int,
                        default=50000,
                        required=False,
                        help=("""Number of evaluation 'run' steps to perform
                              after training (i.e. length of desired chain
                              generate using trained L2HMC sampler.)""""))

    parser.add_argument("--trace",
                        dest="trace",
                        action="store_true",
                        required=False,
                        help=("""Flag that when passed will create trace during
                              training loop.""""))

    parser.add_argument("--save_steps",
                        dest="save_steps",
                        type=int,
                        default=50,
```

```
                        required=False,
                        help=("""Number of steps after which to save the model
                              and current values of all parameters."""))

    parser.add_argument("--print_steps",
                        dest="print_steps",
                        type=int,
                        default=1,
                        required=False,
                        help=("""Number of steps after which to display
                              information about the loss and various
                              other quantities."""))

    parser.add_argument("--logging_steps",
                        dest="logging_steps",
                        type=int,
                        default=50,
                        required=False,
                        help=("""Number of steps after which to write logs for
                              tensorboard."""))

    # ------------------------------------------------------------------------
    # Model parameters
    # ------------------------------------------------------------------------

    parser.add_argument('--network_arch',
                        dest='network_arch',
                        type=str,
                        default='conv3D',
                        required=False,
                        help=("""String specifying the architecture to use for
                              the neural network. Must be one of:
                              `'conv3D', 'conv2D', 'generic'`."""))

    parser.add_argument('--num_hidden',
                        dest='num_hidden',
                        type=int,
                        default=None,
                        required=False,
                        help=("""Number of nodes to include in fully-connected
                              hidden layer `h`. If not explicitly passed, will
                              default to 2 * lattice.num_links."""))

    parser.add_argument('--summaries',
                        dest="summaries",
                        action="store_true",
                        required=False,
                        help=("""Flag that when passed creates
                              summaries of gradients and vaiables for
                              monitoring in tensorboard."""))

    parser.add_argument('--save_samples',
                        dest='save_samples',
                        action='store_true',
                        required=False,
                        help=("""Flag that when passed will save the samples
                              generated during the `run` phase.
                              WARNING: This is very data intensive."""))

    parser.add_argument('--save_lf',
```

```
                        dest='save_lf',
                        action='store_true',
                        required=False,
                        help=("""Flag that when passed will save the
                              output from each leapfrog step"""))

    parser.add_argument('--loop_net_weights',
                        dest='loop_net_weights',
                        action='store_true',
                        help=("""Flag that when passed will iterate over
                              multiple values of `net_weights`, which are
                              multiplicative scaling factors applied to each of
                              the Q, S, T functions when running the trained
                              sampler."""))

    parser.add_argument('--long_run',
                        dest='long_run',
                        action='store_true',
                        required=False,
                        help=("""Flag that when passed runs the trained sampler
                              at model.beta_final and model.beta_final +
                              1."""))

    parser.add_argument("--hmc",
                        dest="hmc",
                        action="store_true",
                        required=False,
                        help=("""Use generic HMC (without augmented leapfrog
                              integrator described in paper). Used for
                              comparing against L2HMC algorithm."""))

    parser.add_argument("--run_hmc",
                        dest="run_hmc",
                        action="store_true",
                        required=False,
                        help=("""Flag that when passed causes generic HMC
                              to be ran after running the trained L2HMC
                              sampler. (Default: False)"""))

    parser.add_argument("--eps_trainable",
                        dest="eps_trainable",
                        action="store_true",
                        required=False,
                        help=("""Flag that when passed will allow the step size
                              `eps` to be a trainable parameter."""))

    parser.add_argument("--metric",
                        dest="metric",
                        type=str,
                        default="cos_diff",
                        required=False,
                        help=("""Metric to use in loss function. Must be one
                              of: `l1`, `l2`, `cos`, `cos2`, `cos_diff`."""))

    parser.add_argument("--std_weight",
                        dest="std_weight",
                        type=float,
                        default=1.,
                        required=False,
                        help=("""Multiplicative factor used to weigh relative
```

```
                            strength of stdiliary term in loss function."""))

    parser.add_argument("--aux_weight",
                        dest="aux_weight",
                        type=float,
                        default=1.,
                        required=False,
                        help=("""Multiplicative factor used to weigh relative
                              strength of auxiliary term in loss function.""""))

    parser.add_argument("--charge_weight",
                        dest="charge_weight",
                        type=float,
                        default=1.,
                        required=False,
                        help=("""Multiplicative factor used to weigh relative
                              strength of top. charge term in loss
                              function"""))

    parser.add_argument("--clip_grads",
                        dest="clip_grads",
                        action="store_true",
                        required=False,
                        help=("""Flag that when passed will clip gradients by
                              global norm using `--clip_value` command line
                              argument. If `--clip_value` is not passed, "
                              it defaults to 100."""))

    parser.add_argument("--clip_value",
                        dest="clip_value",
                        type=float,
                        default=1.,
                        required=False,
                        help=("""Clip value, used for clipping value of
                              gradients by global norm."""))

    parser.add_argument("--log_dir",
                        dest="log_dir",
                        type=str,
                        default=None,
                        required=False,
                        help=("""Log directory to use from previous run.
                              If this argument is not passed, a new
                              directory will be created."""))

    parser.add_argument("--restore",
                        dest="restore",
                        action="store_true",
                        required=False,
                        help=("""Restore model from previous run.  If this
                              argument is passed, a `log_dir` must be specified
                              and passed to `--log_dir argument."""))

    parser.add_argument("--profiler",
                        dest='profiler',
                        action="store_true",
                        required=False,
                        help=("""Flag that when passed will profile the graph
                              execution using `TFProf`."""))

```

```
    parser.add_argument("--gpu",
                        dest="gpu",
                        action="store_true",
                        required=False,
                        help=("""Flag that when passed indicates we're training
                              using an NVIDIA GPU."""))

    parser.add_argument("--theta",
                        dest="theta",
                        action="store_true",
                        required=False,
                        help=("""Flag that when passed indicates we're training
                              on theta @ ALCf."""))

    parser.add_argument("--use_bn",
                        dest='use_bn',
                        action="store_true",
                        required=False,
                        help=("""Flag that when passed causes batch
                              normalization layer to be used in ConvNet."""))

    parser.add_argument("--horovod",
                        dest="horovod",
                        action="store_true",
                        required=False,
                        help=("""Flag that when passed uses Horovod for
                              distributed training on multiple nodes."""))

    parser.add_argument("--num_intra_threads",
                        dest="num_intra_threads",
                        type=int,
                        default=0,
                        required=False,
                        help=("""Number of intra op threads to use for
                              tf.ConfigProto.intra_op_parallelism_threads"""))

    parser.add_argument("--num_inter_threads",
                        dest="num_intra_threads",
                        type=int,
                        default=0,
                        required=False,
                        help=("""Number of intra op threads to use for
                              tf.ConfigProto.intra_op_parallelism_threads"""))

    parser.add_argument("--float64",
                        dest="float64",
                        action="store_true",
                        required=False,
                        help=("""When passed, using 64 point floating precision
                              by settings globals.TF_FLOAT = tf.float64. False
                              by default (use tf.float32)."""))

    if sys.argv[1].startswith('@'):
        args = parser.parse_args(shlex.split(open(sys.argv[1][1:]).read(),
                                             comments=True))
    else:
        args = parser.parse_args()

    return args
```

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

```python
"""
Helper methods for performing file IO.

Author: Sam Foreman (github: @saforem2)
Created: 2/27/2019
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
import os
import datetime
import pickle
import numpy as np

# pylint:disable=invalid-name

try:
    import horovod.tensorflow as hvd

    HAS_HOROVOD = True
    hvd.init()

except ImportError:
    HAS_HOROVOD = False

from globals import FILE_PATH


def log(s, nl=True):
    """Print string `s` to stdout if and only if hvd.rank() == 0."""
    try:
        if HAS_HOROVOD and hvd.rank() != 0:
            return
        print(s, end='\n' if nl else '')
    except NameError:
        print(s, end='\n' if nl else '')


def write(s, f, mode='a', nl=True):
    """Write string `s` to file `f` if and only if hvd.rank() == 0."""
    try:
        if HAS_HOROVOD and hvd.rank() != 0:
            return
        with open(f, mode) as ff:
            ff.write(s + '\n' if nl else '')
    except NameError:
        with open(f, mode) as ff:
            ff.write(s + '\n' if nl else '')


def log_and_write(s, f):
    """Print string `s` to std out and also write to file `f`."""
    log(s)
    write(s, f)


def create_log_dir(FLAGS, root_dir=None, log_file=None):
```

```
    """Automatically create and name `log_dir` to save model data to.

    The created directory will be located in `logs/YYYY_M_D/`, and will have
    the format (without `_qw{QW}` if running generic HMC):

        `lattice{LX}_batch{NS}_lf{LF}_eps{SS}_qw{QW}`

    Returns:
        FLAGS, with FLAGS.log_dir being equal to the newly created log_dir.

    NOTE: If log_dir does not already exist, it is created.
    """
    LX = FLAGS.space_size
    NS = FLAGS.num_samples
    LF = FLAGS.num_steps
    #  SS = str(FLAGS.eps).lstrip('0.')
    SS = FLAGS.eps
    QW = FLAGS.charge_weight
    if FLAGS.hmc:
        run_str = f'HMC_lattice{LX}_batch{NS}_lf{LF}_eps{SS:.3g}'
    else:
        run_str = f'lattice{LX}_batch{NS}_lf{LF}_eps{SS:.3g}_qw{QW}'

    now = datetime.datetime.now()
    #  print(now.strftime("%b %d %Y %H:%M:%S"))
    day_str = now.strftime('%Y_%m_%d')
    time_str = now.strftime("%Y_%m_%d_%H%M")

    #  day_str = f'{now.year}_{now.month}_{now.day}'
    #  time_str = day_str + f'_{now.hour}{now.minute}'
    project_dir = os.path.abspath(os.path.dirname(FILE_PATH))
    if FLAGS.log_dir is None:
        if root_dir is None:
            _dir = 'logs'
        else:
            _dir = root_dir

    else:
        if root_dir is None:
            _dir = FLAGS.log_dir
        else:
            _dir = os.path.join(FLAGS.log_dir, root_dir)
    root_log_dir = os.path.join(project_dir, _dir, day_str, time_str, run_str)
    check_else_make_dir(root_log_dir)
    run_num = get_run_num(root_log_dir)
    log_dir = os.path.abspath(os.path.join(root_log_dir,
                                           f'run_{run_num}'))
    if log_file is not None:
        write(f'Output saved to: \n\t{log_dir}', log_file, 'a')
        write(80*'-', log_file, 'a')

    return log_dir


def _list_and_join(d):
    """For each dir `dd` in `d`, return a list of paths ['d/dd1', ...]"""
    contents = [os.path.join(d, i) for i in os.listdir(d)]
    paths = [i for i in contents if os.path.isdir(i)]

    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}.")
```

```
    log(f"Saving params to: {params_file}.")
    with open(params_file, 'wb') as f:
        pickle.dump(params, f)


def get_run_num(log_dir):
    """Get integer value for next run directory."""
    check_else_make_dir(log_dir)
    contents = os.listdir(log_dir)
    if contents in ([], ['.DS_Store']):
        return 1
    try:
        run_dirs = [i for i in os.listdir(log_dir) if 'run' in i]
        run_nums = [int(i.split('_')[-1]) for i in run_dirs]
        run_num = sorted(run_nums)[-1] + 1
    except (ValueError, IndexError):
        log(f"No previous runs found in {log_dir}, setting run_num=1.")
        run_num = 1

    return run_num


def _get_run_num(log_dir):
    check_else_make_dir(log_dir)

    contents = os.listdir(log_dir)
    if contents in ([], ['.DS_Store']):
        return 1

    run_nums = []
    for item in contents:
        try:
            run_nums.append(int(item.split('_')[-1]))
        except ValueError:
            continue
    if run_nums == []:
        return 1

    return sorted(run_nums)[-1] + 1
```

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

```python
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os
from globals import ROOT_DIR
import utils.file_io as io

import tensorflow as tf

TOWER_NAME = 'tower'


def get_run_num(log_dir):
    if not os.path.isdir(log_dir):
        os.makedirs(log_dir)
    contents = os.listdir(log_dir)
    if contents == []:
        return 1
    else:
        run_nums = []
        for item in contents:
            try:
                run_nums.append(int(item.split('_')[-1]))
                #  run_nums.append(int(''.join(x for x in item if
                #  x.isdigit())))
            except ValueError:
                continue
        return sorted(run_nums)[-1] + 1
    #  if contents == ['.DS_Store']:
    #      return 1
    #  else:
    #      for item in contents:
    #          if os.path.isdir(log_dir + item):
    #              run_dirs.append(item)
    #      run_nums = [int(str(i)[3:]) for i in run_dirs]
    #      prev_run_num = max(run_nums)
    #      return prev_run_num + 1


def make_run_dir(log_dir):
    if log_dir.endswith('/'):
        _dir = log_dir
    else:
        _dir = log_dir + '/'
    run_num = get_run_num(_dir)
    run_dir = _dir + f'run_{run_num}/'
    if os.path.isdir(run_dir):
        raise f'Directory: {run_dir} already exists, exiting!'
    else:
        print(f'Creating directory for new run: {run_dir}')
        os.makedirs(run_dir)
    return run_dir


def check_log_dir(log_dir):
    if not os.path.isdir(log_dir):
```

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

```
    loss_averages = tf.train.ExponentialMovingAverage(0.9, name='avg')
    losses = tf.get_collection('losses')
    loss_averages_op = loss_averages.apply(losses + [total_loss])

    # Attach a scalar summary to all individual losses and the total loss;
    # do the same for the averaged version of the losses.
    for l in losses + [total_loss]:
        # Name each loss as '(raw)' and name the moving average version of
        # the loss as the original loss name.
        tf.summary.scalar(l.op.name + ' (raw)', l)
        tf.summary.scalar(l.op.name, loss_averages.average(l))

    return loss_averages_op


def activation_summary(x):
    """Helper to create summaries for activations.A

    Creates a summary that provides a histogram of activations.
    Creates a summary that measures the sparisty of activations.

    Args:
        x: Tensor
    Returns:
        None
    """
    # Remove 'tower_[0-9]/' from the name in case this is a multi-GPU training
    # session. This helps the clarity of presentation in tensorboard.
    #  tensor_name = re.sub('%s_[0-9]*/' % TOWER_NAME, '', x.op.name)
    try:
        tensor_name = x.op.name
    except AttributeError:
        tensor_name = x.name
    tf.summary.histogram(tensor_name + '/activations', x)
    tf.summary.scalar(tensor_name + '/sparsity', tf.nn.zero_fraction(x))
```

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

```python
"""
data_loader.py

Implements DataLoader class responsible for loading run_data.

Author: Sam Foreman (github: @saforem2)
Date: 05/03/2019
"""
import os
import pickle

import numpy as np


def load_params_from_dir(d):
    params_file = os.path.join(d, 'params.pkl')
    with open(params_file, 'rb') as f:
        params = pickle.load(f)

    return params


def get_run_dirs(root_dir):
    run_dirs = []
    root_dir = os.path.abspath(root_dir)
    for dirpath, dirnames, filenames in os.walk(root_dir):
        for dirname in dirnames:
            keys = ['steps_', '_beta_', '_eps_']
            conditions = [key in dirname for key in keys]
            if np.alltrue(conditions):
                run_dirs.append(os.path.join(dirpath, dirname))

    return run_dirs

#  def make_fig_dirs(run_dirs):
#      for dirpath, dirnames, filenames in os.walk(root_dir):
#          for dirname in dirnames:
#              keys = ['steps_', '_beta_', '_eps_']
#              conditions = [key in dirname for key in keys]
#              if np.alltrue(conditions):
#                  run_dirs.append(os.path.join(dirpath, dirname))
#      return run_dirs


class DataLoader:
    def __init__(self, run_dir=None):
        self.run_dir = None

    def load_pkl_file(self, pkl_file):
        with open(pkl_file, 'rb') as f:
            contents = pickle.load(f)

        return contents

    def load_npz_file(self, npz_file):
        arr = np.load(npz_file)

```

```
        return arr.f.arr_0

    def _load_observable(self, observable_str, run_dir=None):
        if run_dir is None:
            run_dir = self.run_dir

        if not observable_str.endswith('.pkl'):
            observable_str += '.pkl'

        obs_dir = os.path.join(run_dir, 'observables')
        obs_file = os.path.join(obs_dir, observable_str)

        return self.load_pkl_file(obs_file)

    def load_samples(self, run_dir):
        samples_file = os.path.join(run_dir, 'samples_out.npz')
        samples = self.load_npz_file(samples_file)

        return samples

    def load_leapfrogs(self, run_dir):
        lf_f_file = os.path.join(run_dir, 'lf_forward.npz')
        lf_b_file = os.path.join(run_dir, 'lf_backward.npz')
        lf_f = self.load_npz_file(lf_f_file)
        lf_b = self.load_npz_file(lf_b_file)

        return (lf_f, lf_b)

    def load_logdets(self, run_dir):
        logdets_f_file = os.path.join(run_dir, 'logdets_forward.npz')
        logdets_b_file = os.path.join(run_dir, 'logdets_backward.npz')

        logdets_f = self.load_npz_file(logdets_f_file)
        logdets_b = self.load_npz_file(logdets_b_file)

        return (logdets_f, logdets_b)

    def load_sumlogdets(self, run_dir):
        sumlogdet_f_file = os.path.join(run_dir, 'sumlogdet_forward.npz')
        sumlogdet_b_file = os.path.join(run_dir, 'sumlogdet_backward.npz')

        sumlogdet_f = self.load_npz_file(sumlogdet_f_file)
        sumlogdet_b = self.load_npz_file(sumlogdet_b_file)

        return (sumlogdet_f, sumlogdet_b)

    def load_plaqs(self, run_dir):
        obs_dir = os.path.join(run_dir, 'observables')
        plaqs_file = os.path.join(obs_dir, 'plaqs.pkl')

        return self.load_pkl_file(plaqs_file)

    def load_autocorrs(self, run_dir):
        obs_dir = os.path.join(run_dir, 'observables')
        autocorrs_file = os.path.join(obs_dir, 'charges_autocorrs.pkl')

        return self.load_pkl_file(autocorrs_file)

    def load_params(self, run_dir=None):
        if run_dir is None:
```

```
            run_dir = self.run_dir
        params_file = os.path.join(run_dir, 'parameters.pkl')

        return self.load_pkl_file(params_file)

    def load_run_data(self, run_dir=None):
        if run_dir is None:
            run_dir = self.run_dir

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

        return self.load_pkl_file(run_data_file)

    def load_run_stats(self, run_dir=None):
        if run_dir is None:
            run_dir = self.run_dir

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

        return self.load_pkl_file(run_stats_file)

    def load_observables(self, run_dir=None):
        if run_dir is None:
            run_dir = self.run_dir

        obs_dir = os.path.join(run_dir, 'observables')
        files = os.listdir(obs_dir)

        obs_names = [i.rstrip('.pkl') for i in files]
        obs_files = [os.path.join(obs_dir, i) for i in files]

        observables = {
            n: self.load_pkl_file(f) for (n, f) in zip(obs_names, obs_files)
        }

        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.

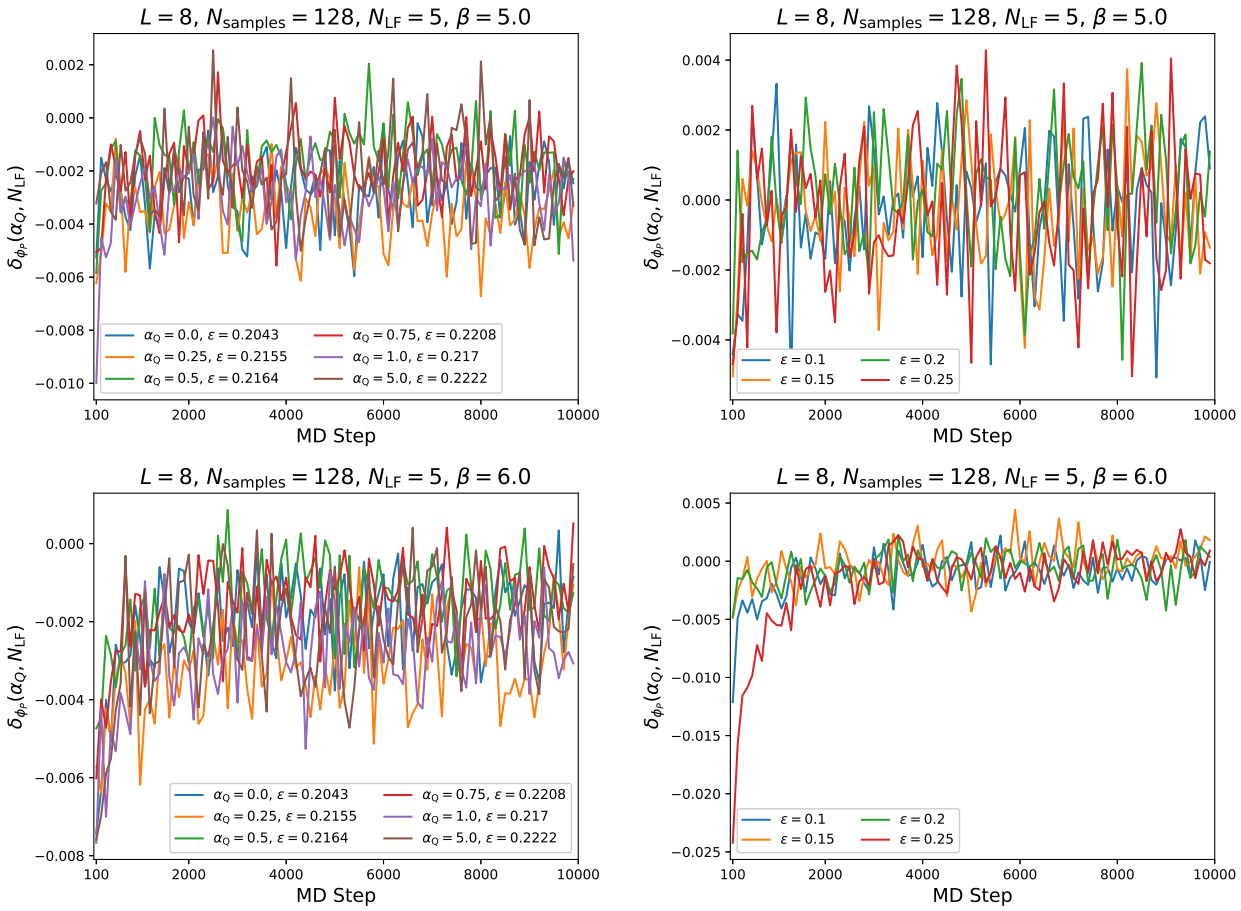


Figure B.1: δ_{ϕ_P} vs MD step with $N_{\text{LF}} = 5$ for $\beta = 5.0$ (top row) and $\beta = 6.0$ (bottom row). The results from the trained L2HMC (generic HMC) sampler are shown in the left (right) column. As can be seen, the difference δ_{ϕ_P} remains roughly consistent for all values of α_Q.

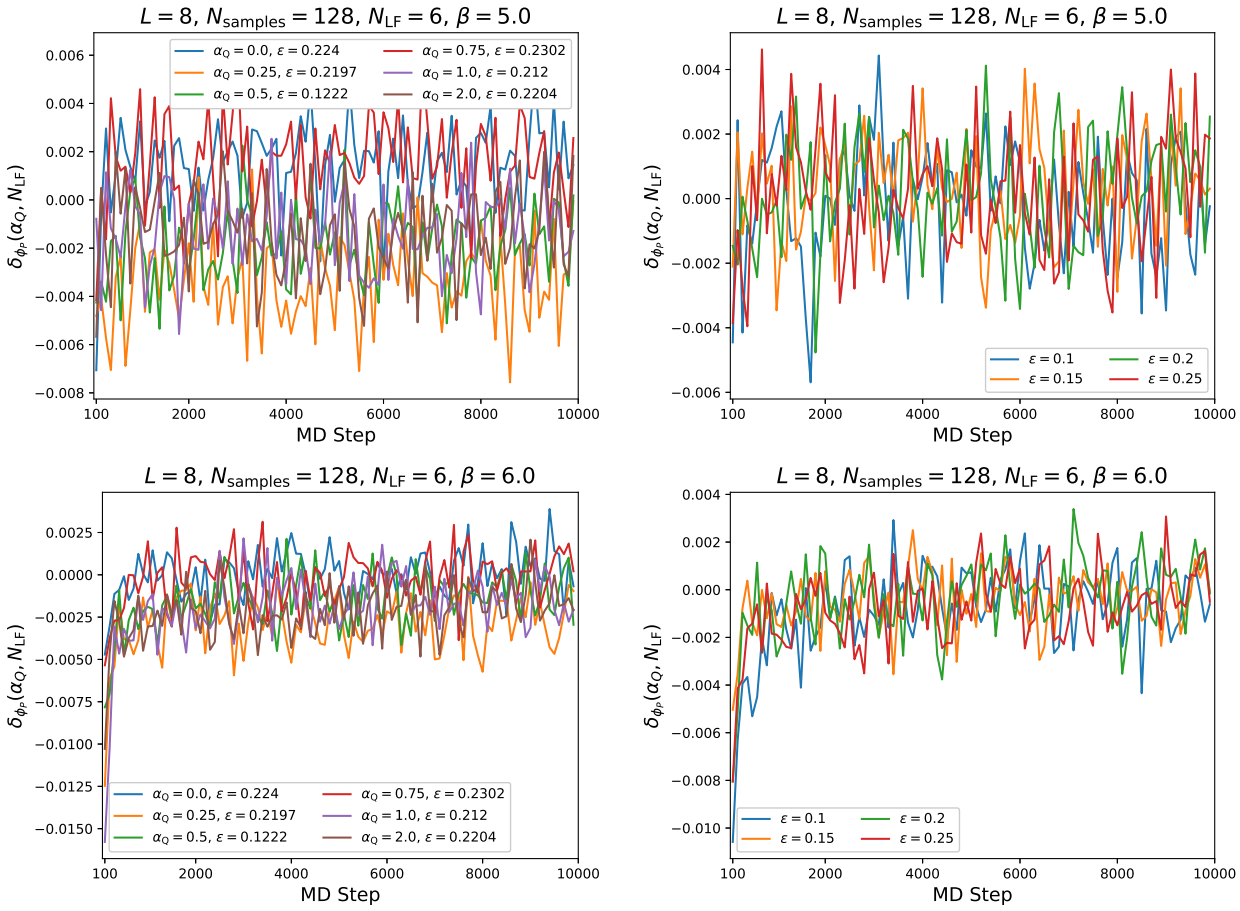


Figure B.2: δ_{ϕ_P} vs MD step with $N_{\text{LF}} = 5$ for $\beta = 5.0$ (top row) and $\beta = 6.0$ (bottom row). The results from the trained L2HMC (generic HMC) sampler are shown in the left (right) column. As can be seen, the difference δ_{ϕ_P} remains roughly consistent for all values of α_Q.

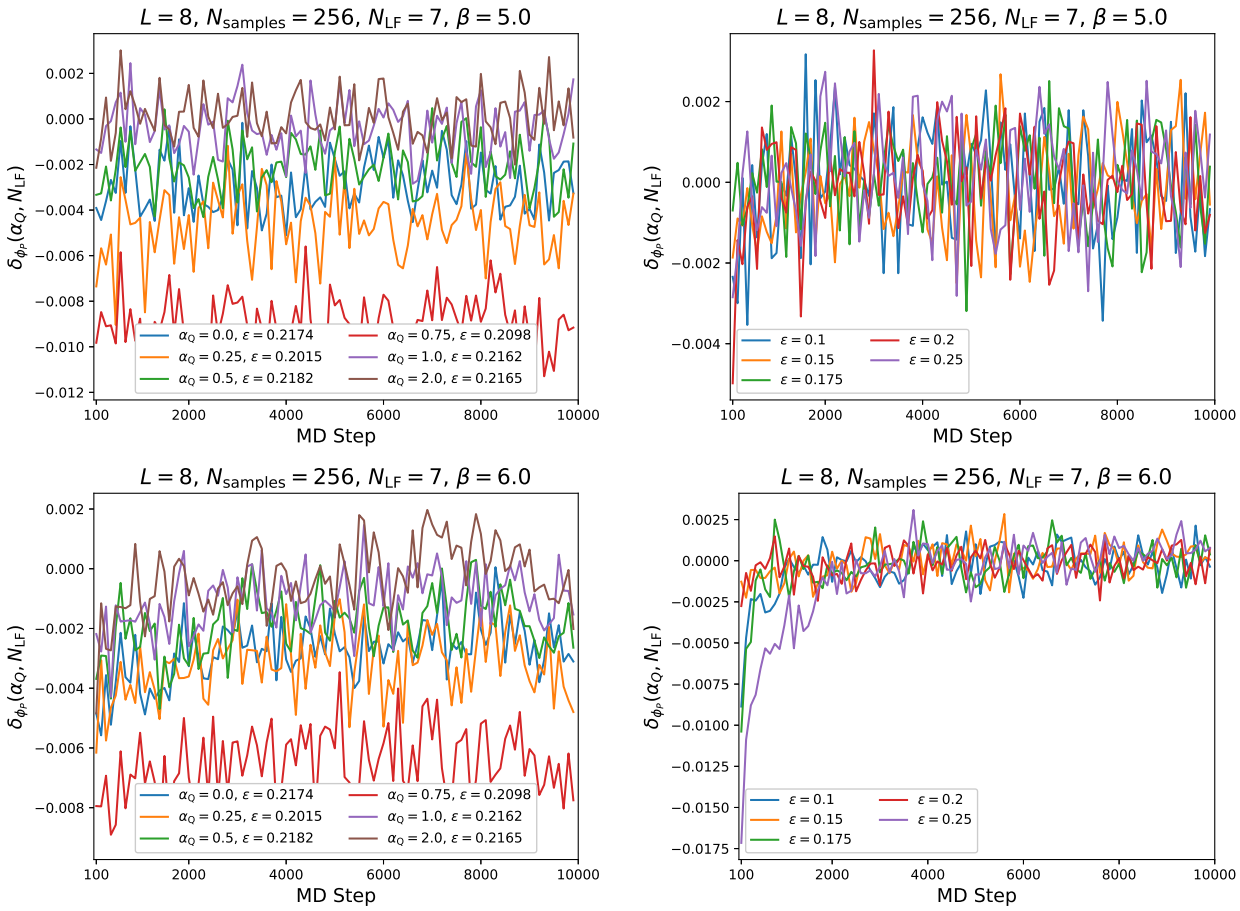


Figure B.3: δ_{ϕ_P} vs MD step with $N_{\text{LF}} = 7$ As can be seen, the difference δ_{ϕ_P} is noticeably larger for $\alpha_Q = 0.75$, but remains roughly consistent for all other values of α_Q.

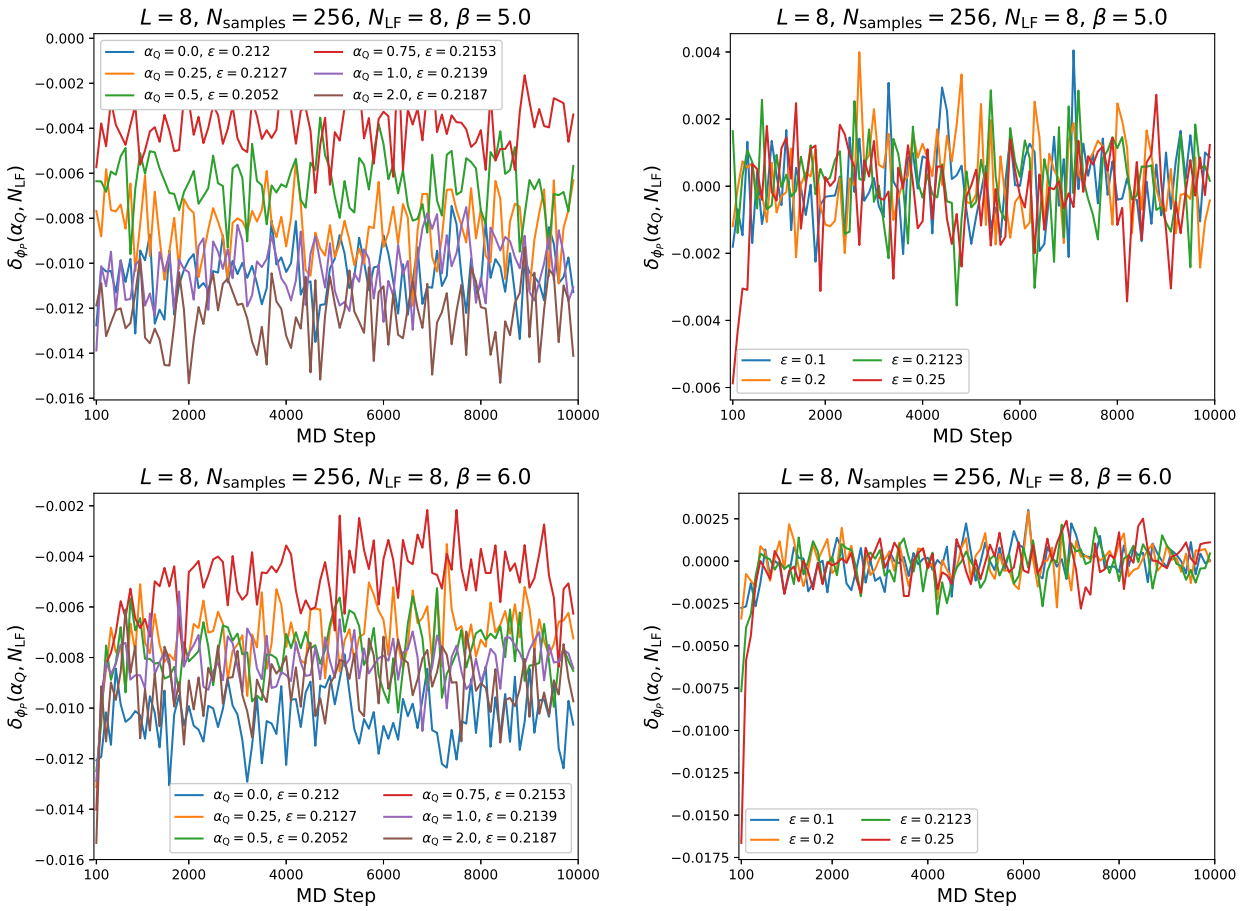


Figure B.4: δ_{ϕ_P} vs. MD step with $N_{\text{LF}} = 8$. As can be seen, the difference δ_{ϕ_P} remains roughly consistent for all values of α_Q.

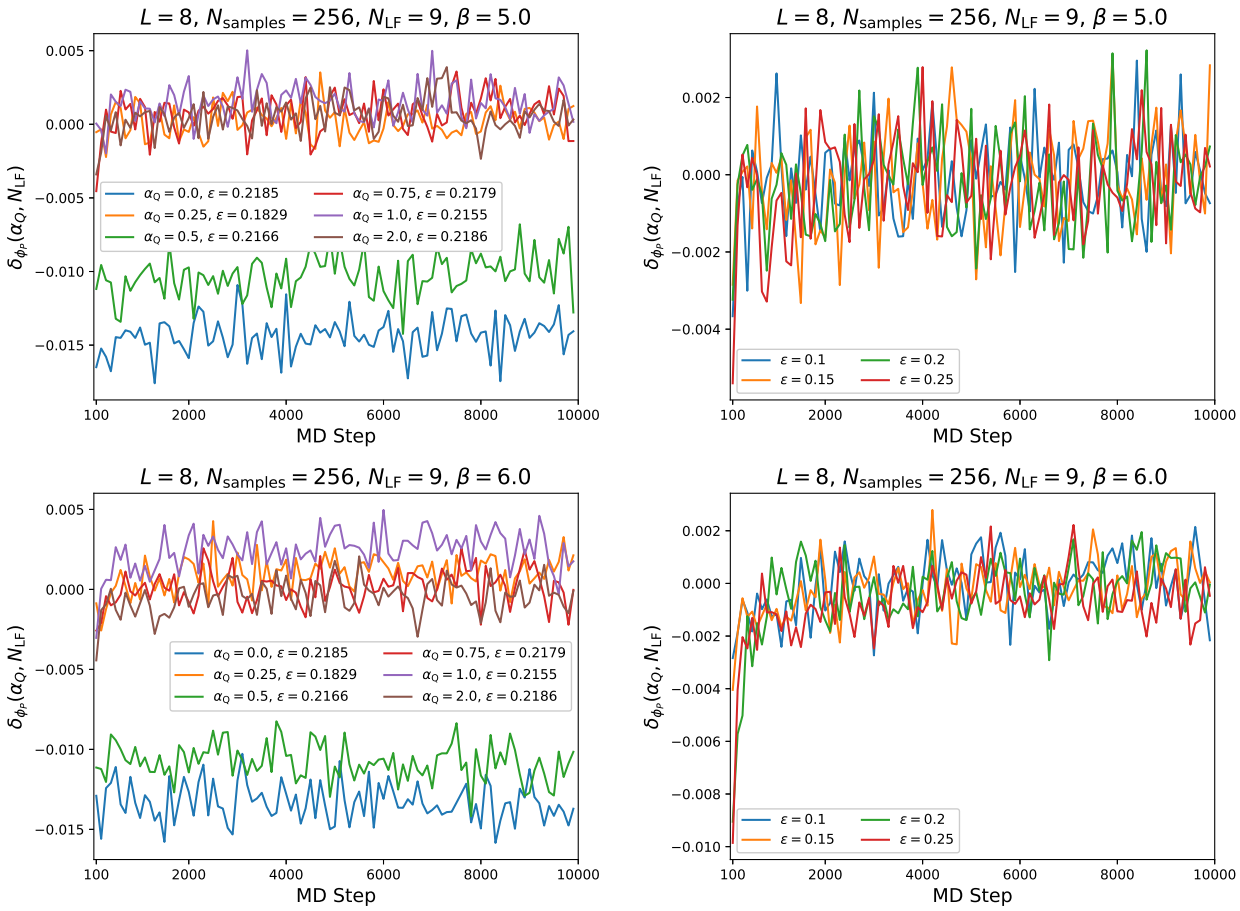


Figure B.5: δ_{ϕ_P} vs MD step with $N_{\text{LF}} = 9$. As can be seen, the difference δ_{ϕ_P} is largest for $\alpha_Q = 0.0$ and $\alpha_Q = 0.5$.

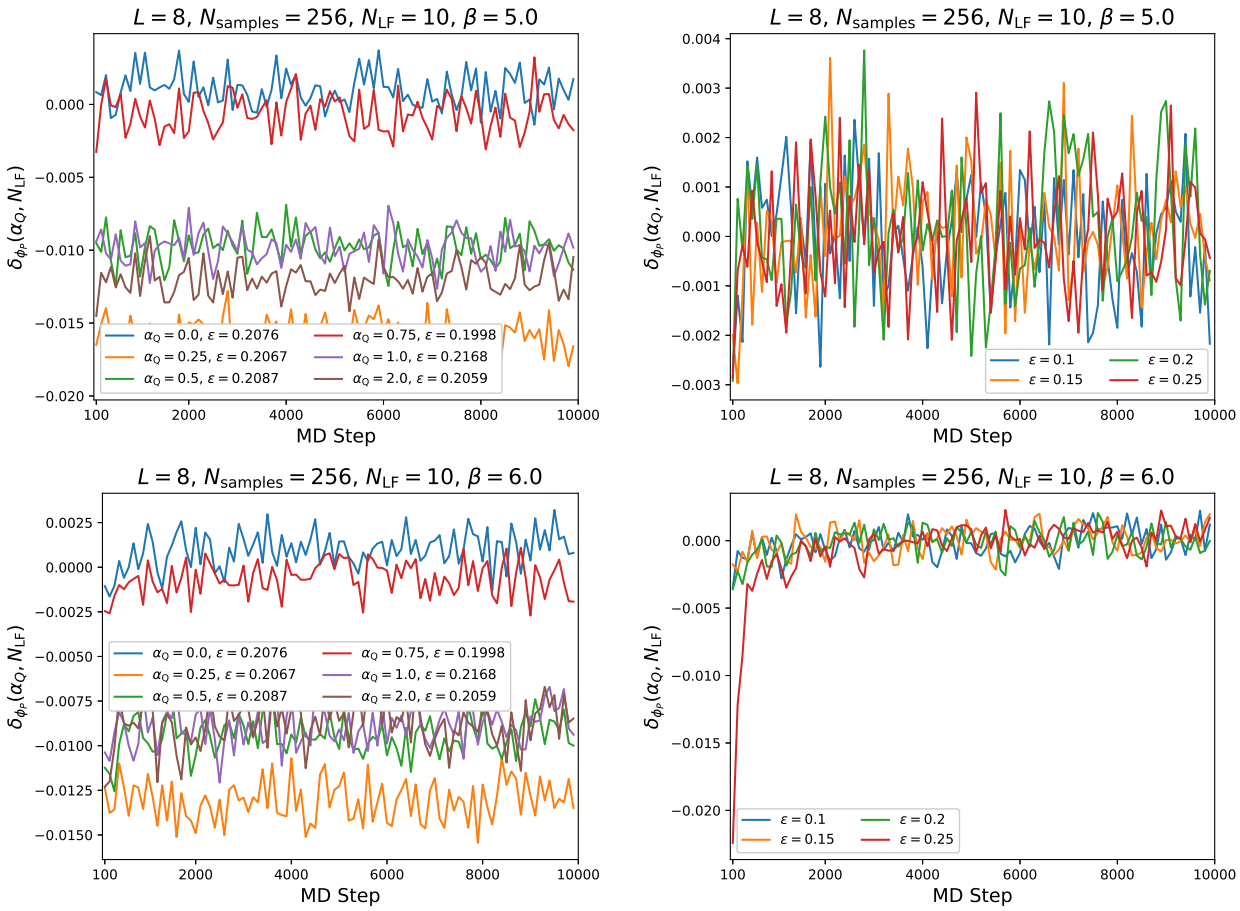


Figure B.6: δ_{ϕ_P} vs MD step with $N_{\text{LF}} = 10$. As can be seen, the difference δ_{ϕ_P} is smallest for $\alpha_Q = 0.,\ 0.75$ and largest for $\alpha_Q = 0.25$, while $\alpha_Q = 0.5,\ 2.0$, takes on intermediate values.

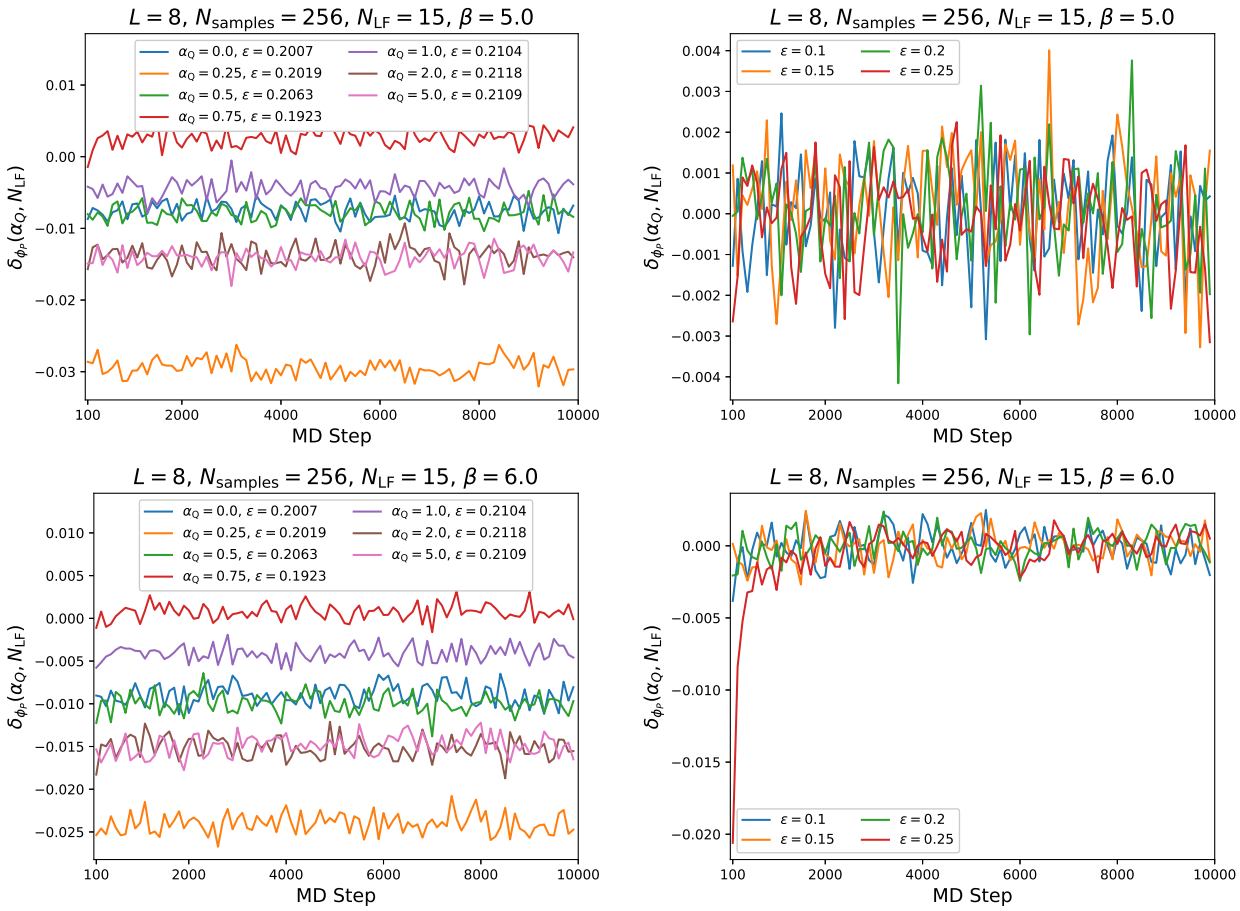


Figure B.7: δ_{ϕ_P} vs MD step with $N_{\text{LF}} = 15$. As can be seen, the difference δ_{ϕ_P} varies for different values of α_Q.

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] P. Mehta, M. Bukov, C.-H. Wang, A. G. R. Day, C. Richardson, C. K. Fisher, and D. J. Schwab, Physics Reports **810**, arXiv: 1803.08823, 1 (2019).

[4] D. P. Kingma and M. Welling, arXiv:1312.6114 [cs, stat], arXiv: 1312.6114 (2013).

[5] 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).

[6] A. Ng, "Cs229 lecture notes - supervised learning", 2012.

[7] C. M. Bishop, *Pattern recognition and machine learning*, Information science and statistics (Springer, 2006).

[8] S.-i. Horikawa, T. Furuhashi, and Y. Uchikawa, IEEE Transactions on Neural Networks **3**, 801 (1992).

[9] J. Schmidhuber, Neural Networks **61**, arXiv: 1404.7828, 85 (2015).

[10] M. Nielsen, 224 (2015).

[11] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Nature **323**, 533 (1986).

[12] Y. LeCun, Y. Bengio, et al., The handbook of brain theory and neural networks **3361**, 1995 (1995).

[13] W. Zhang, K. Itoh, J. Tanida, and Y. Ichioka, Applied Optics **29**, 4790 (1990).

[14] G. E. Hinton, S. Osindero, and Y.-W. Teh, Neural Computation **18**, PMID: 16764513, 1527 (2006).

[15] D. Scherer, A. Müller, and S. Behnke, in International conference on artificial neural networks (Springer, 2010), pp. 92–101.

[16] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, arXiv:1412.6806 [cs], arXiv: 1412.6806 (2014).

[17] S.-H. Li and L. Wang, Physical Review Letters **121**, 260601, 260601 (2018).

[18] C. Bény, arXiv:1301.3124 [quant-ph], arXiv: 1301.3124 (2013).

[19] P. Mehta and D. J. Schwab, arXiv:1410.3831 [cond-mat, stat], arXiv: 1410.3831 (2014).

[20] 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).

[21] S. Bradde and W. Bialek, Journal of Statistical Physics **167**, 462 (2017).

[22] J. Carrasquilla and R. G. Melko, Nature Physics **13**, 431 (2017).

[23] L. Wang, Phys. Rev. B **94**, 195105 (2016).

[24] W. Hu, R. R. P. Singh, and R. T. Scalettar, Phys. Rev. E **95**, 062122, 062122 (2017).

[25] S. J. Wetzel, Phys. Rev. E **96**, 022140, 022140 (2017).

[26] M. Levin and C. P. Nave, Phys. Rev. Lett. **99**, 120601 (2007).

[27] Z.-C. Gu, M. Levin, B. Swingle, and X.-G. Wen, Phys. Rev. B **79**, 085118 (2009).

[28] Z. Y. Xie, J. Chen, M. P. Qin, J. W. Zhu, L. P. Yang, and T. Xiang, Phys. Rev. B **86**, 045139 (2012).

[29] Y. Meurice, Phys. Rev. **B87**, 064422 (2013).

[30] 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).

[31] 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).

[32] 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).

[33] R. Savit, Rev. Mod. Phys. **52**, 453 (1980).

[34] N. Prokof'ev and B. Svistunov, Phys. Rev. Lett. **87**, 160601 (2001).

[35] T. A. Enßlin and M. Frommert, Phys. Rev. D **83**, 105014 (2011).

[36] R. H. Swendsen and J.-S. Wang, Phys. Rev. Lett. **58**, 86 (1987).

[37] B. Kaufman, Phys. Rev. **76**, 1232 (1949).

[38] A. Krizhevsky and G. Hinton, Master's thesis, Department of Computer Science, University of Toronto (2009).

[39] B. Gidas, IEEE Transactions on Pattern Analysis and Machine Intelligence **11**, 164 (1989).

[40] K. Tanaka, Journal of Physics A: Mathematical and General **35**, R81 (2002).

[41] T. Horiguchi, Y. Honda, and M. Miya, Physics Letters A **227**, 319 (1997).

[42] Z. Y. Xie, H. C. Jiang, Q. N. Chen, Z. Y. Weng, and T. Xiang, Phys. Rev. Lett. **103**, 160601 (2009).

[43] G. E. Hinton and R. R. Salakhutdinov, Science **313**, 504 (2006).

[44] P. E. Shanahan, D. Trewartha, and W. Detmold, Phys. Rev. **D97**, 094506 (2018).

[45] S. Brooks, A. Gelman, G. Jones, and X.-L. Meng, *Handbook of markov chain monte carlo* (CRC press, 2011).

[46] *Scribe_note_lecture17.pdf*, `https://www.cs.cmu.edu/~epxing/Class/10708-14/scribe_notes/scribe_note_lecture17.pdf`, (Accessed on 06/18/2019).

[47] M. Hanada, arXiv:1808.08490 [cond-mat, physics:hep-lat, physics:hep-th], arXiv: 1808.08490 (2018).

[48] R. M. Neal, arXiv:1206.1901 [physics, stat], arXiv: 1206.1901 (2012).

[49] M. Betancourt, arXiv:1701.02434 [stat], arXiv: 1701.02434 (2017).

[50] *30 joeylitalien/l2hmc: iclr 2018 reproducibility challenge: generalizing hamiltonian monte carlo with neural networks*, https://joeylitalien.github.io/assets/reports/l2hmc.pdf, (Accessed on 06/21/2019).

[51] L. Dinh, J. Sohl-Dickstein, and S. Bengio, arXiv:1605.08803 [cs, stat], arXiv: 1605.08803 (2016).

[52] C. Pasarica and A. Gelman, Statistica Sinica **20**, 343 (2010).

[53] S. Ioffe and C. Szegedy, arXiv:1502.03167 [cs], arXiv: 1502.03167 (2015).