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

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. 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 DESC0014664.

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 likelihood of

[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.

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 rep-

resenting 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, $\mathscr{L} = \mathscr{L}[f_w(x^{(i)}), y^{(i)}] \equiv \mathscr{L}(w)$, 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 their values by "stepping" in the direction of steepest decrease of $\mathscr{L}$:

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

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.

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

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 $\mathscr{L}$. For this example, we can choose the least-squares cost function defined to be

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

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

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

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

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

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

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

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

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

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

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

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 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

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

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

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** $\ell(w)$:

$$\ell(w) \equiv \log L(w) \tag{2.23}$$

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

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

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

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

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

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

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

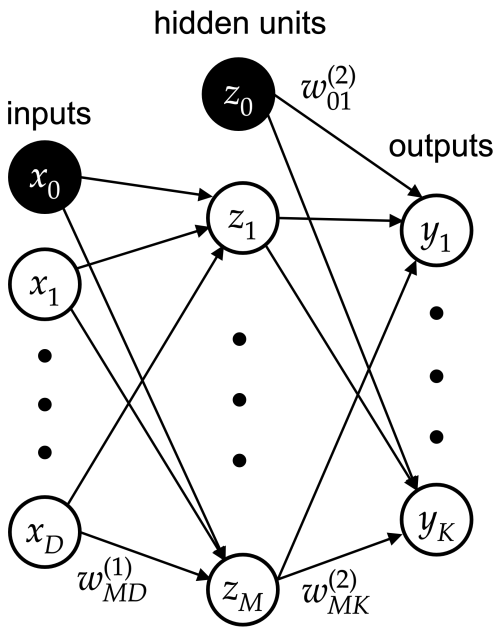


Figure 2.1: Network diagram for a two-layer fully-connected neural network. The input, hidden, and output variables are shown as nodes while the weight parameters are the links between them.

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.

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 $\mathscr{L}(\mathbf{w})$ 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 $\mathscr{L}$ 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

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

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

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

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

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

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:

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

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

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

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

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

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

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 volume of shape height $\times$ width $\times$ depth where height and width are, as we would expect, the height and width of the input image, and depth usually refers to the number of *color channels* (i.e. RGB) of the image. For example, if we are dealing with a typical two-dimensional RGB image, then depth = 3, which we could represent as a three-dimensional volume of height $\times$ width $\times$ [red, green, blue] pixels.

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

- *Local receptive fields*
- *Shared weights* (or weight replication)

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 $\mathscr{L}$, 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 $\partial \mathscr{L}_d \,/\, \partial w_{ij}^k$ 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 $\mathscr{L}_d$ with respect to w_{ij}^k by using Eq. 2.53.
3. **Combine individual gradients:** for each input-output pair $\frac{\partial \mathscr{L}_n}{\partial w_{ij}^k}$ to get the total gradient $\frac{\partial \mathscr{L}(X,\theta)}{\partial w_{ij}^k}$ for the entire set of input-output pairs using:

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

- 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.

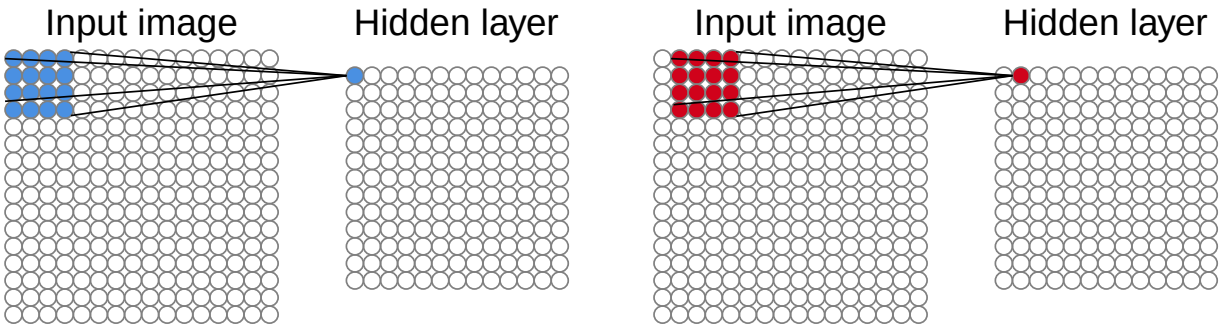


Figure 2.2: Illustration of the local receptive fields in the input image and their corresponding weights in the first hidden layer. Additionally, we can see how these local receptive fields are slid across the input image to construct additional units in the hidden layer.

2.4.2 Shared Weights

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

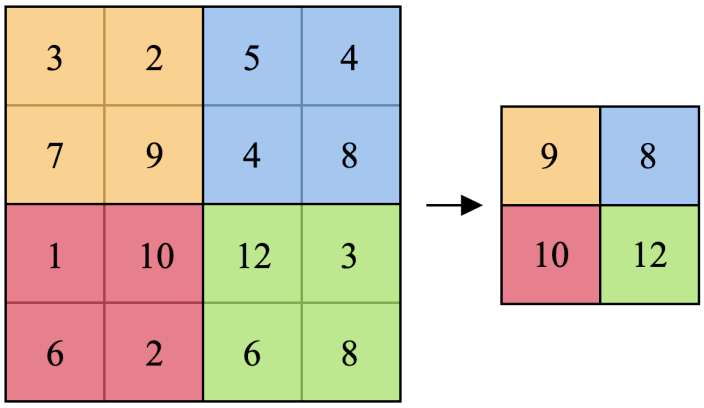


Figure 2.3: Illustration of a 2×2 max pooling layer.

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

Portions of this chapter have appeared in:

"Examples of renormalization group transformations for image sets", S. Foreman, J. Giedt, Y. Meurice, and J. Unmuth-Yockey, *Phys. Rev. E*, **98**, 052129 – Published 26 November 2018

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 $1+1 \longrightarrow x$ hereafter. In the case $1+1 \longrightarrow 0$, 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 $1 + 1 \longrightarrow 0$) 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 suggests 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

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

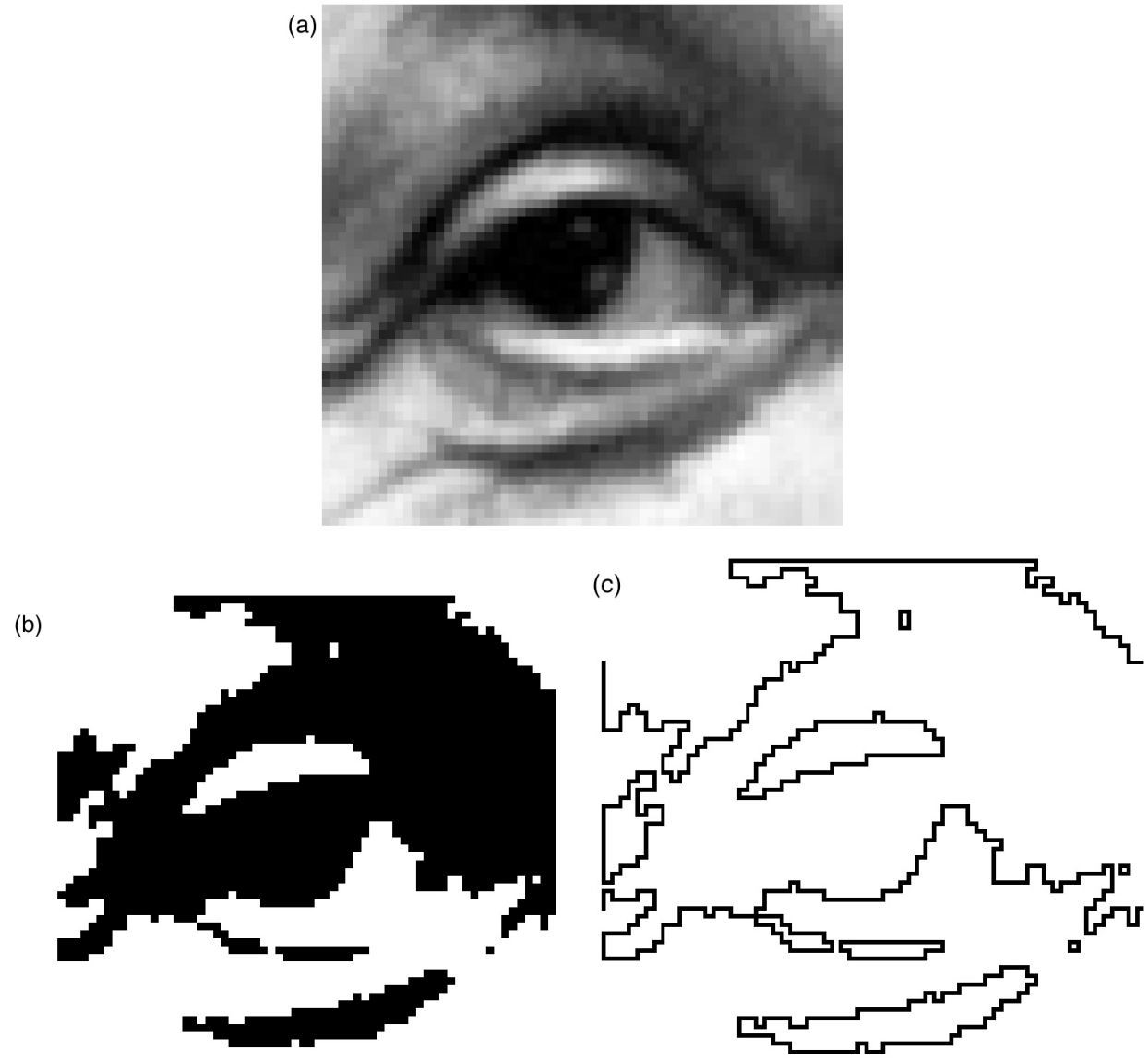


Figure 3.1: (a) Picture of an eye with 4096 pixels; (b) black and white version with a graycut at 0.72; (c) boundaries of the black domains.

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

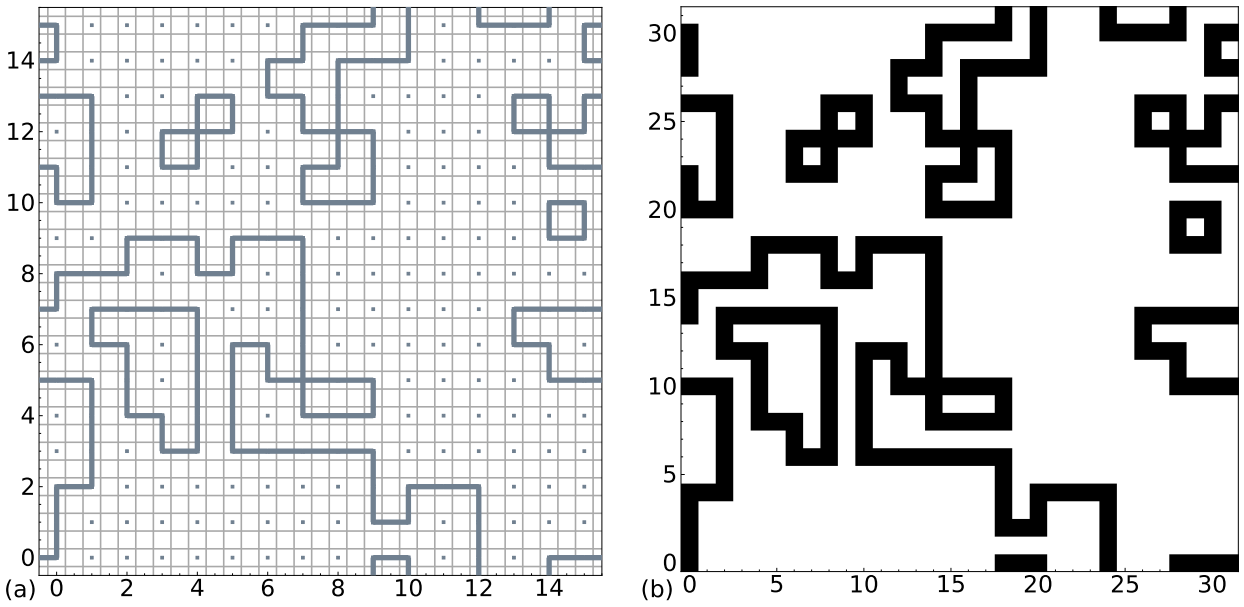


Figure 3.2: (a) Legal worm configuration on an $L \times L$ lattice with periodic boundary conditions and; (b) its equivalent representation as a $2L \times 2L$ black and white pixel image.

$v_i \in \{0, 1\}$. This allows us to write the 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.

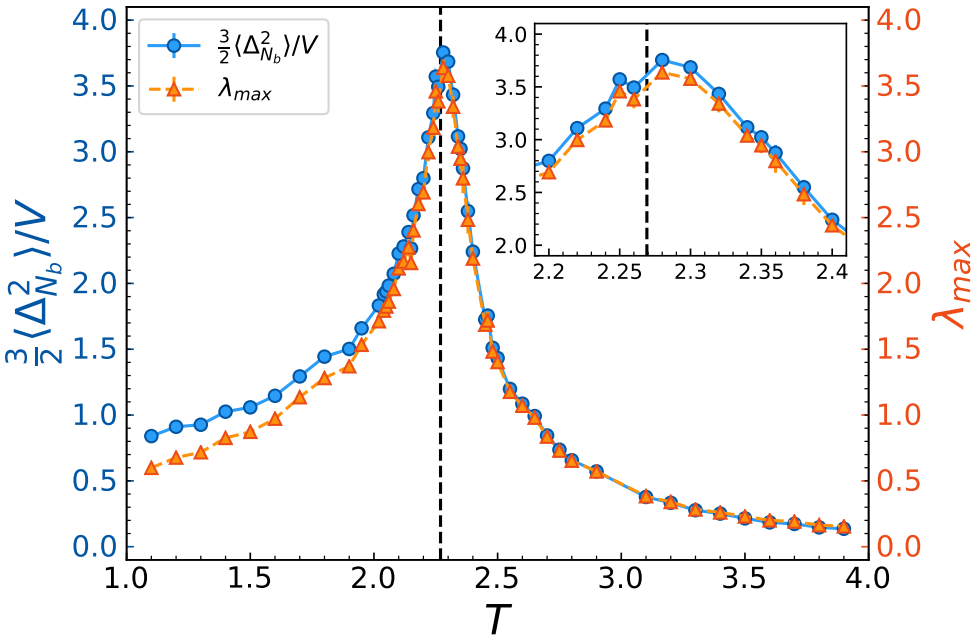


Figure 3.3: λ_{max} and $\frac{3}{2}\left\langle\Delta^2_{N_b}\right\rangle/V$ (per unit volume) vs. T, illustrating the relation between the eigenvalue corresponding to the first principal component and the logarithmic divergence of the specific heat. The inset shows a qualitative agreement near the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

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 variables 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

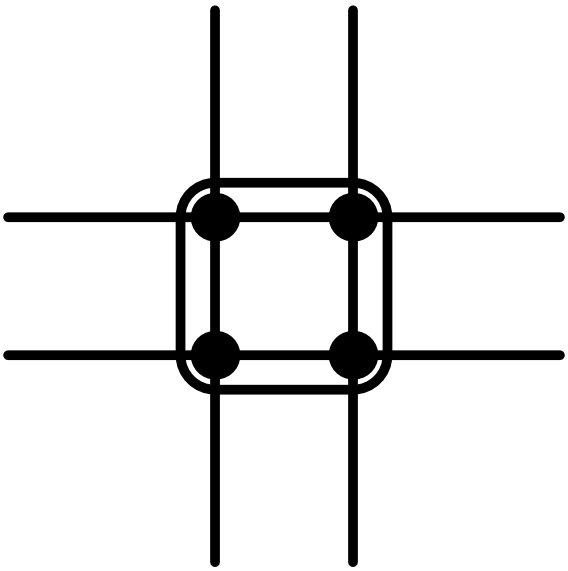


Figure 3.4: Illustration of the tensor blocking discussed in the text. Each dot is a tensor at a lattice site with four lines coming out, each representing a tensor index. Lines connecting dots represent tensor contractions.

practice, this exact procedure cannot be repeated indefinitely and truncations are necessary. This 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

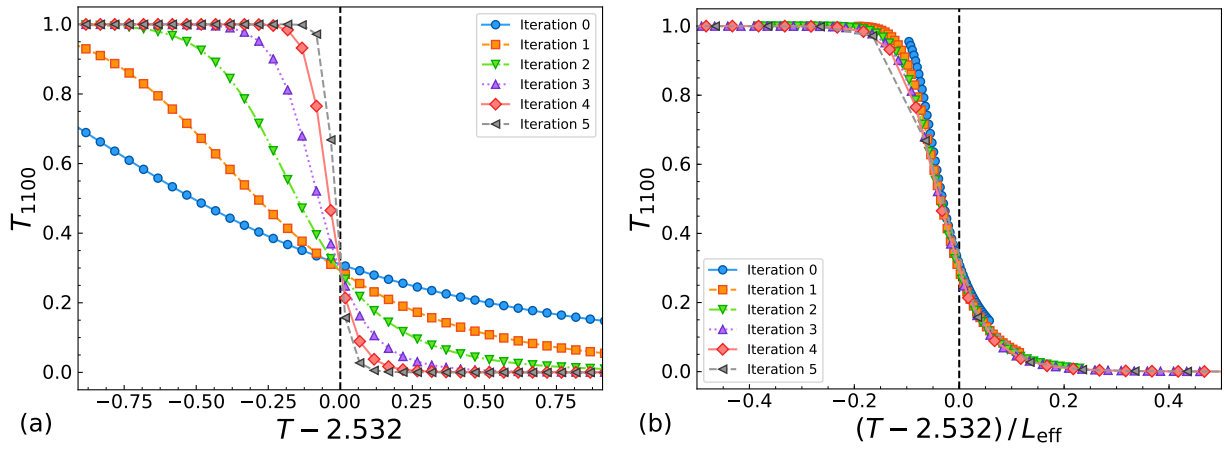


Figure 3.5: (a) T_{1100} vs. $T - T_c^{(2s)}$ for six successive iterations of the blocking transformation, beginning with an initial lattice $L = 64$; (b) T_{1100} vs. $(T - T_c^{(2s)})/L_{\text{eff}}$ illustrating the data collapse, where $T_c^{(2s)}$ is the critical temperature of the two state projection,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

relate the 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 single site with bonds

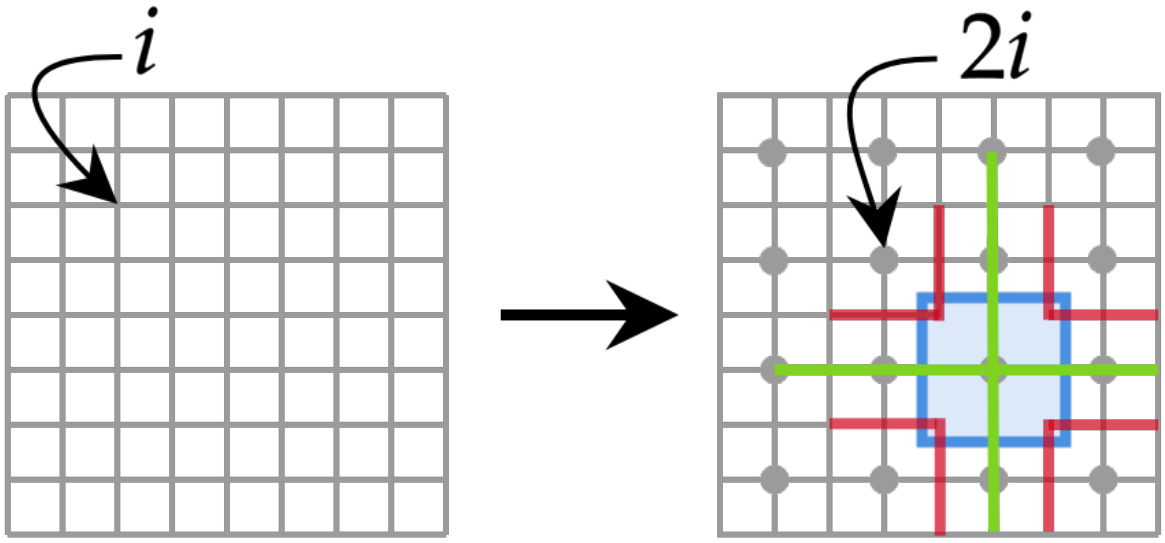


Figure 3.6: Illustration of the coarse-graining procedure in which the original lattice sites (i) are replaced by blocked sites ($2i$) (grey circles) with twice the original lattice spacing. In the coarse-grained lattice, an elementary block (blue) consists of four sites on the original lattice, eight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

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 "$1 + 1 \longrightarrow 0$". This approach respects the selection rule (conservation modulo 2) and has the advantage of maintaining the closed-path restriction. In other words with the $1 + 1 \longrightarrow 0$ 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 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 "$1 + 1 \longrightarrow 1$", and "$1 + 1 \longrightarrow 2$" 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 $1 + 1 \longrightarrow 0$ rule starting with configurations on a 64×64 lattice. A first observation is that the $1 + 1 \longrightarrow 0$ blocking preserves

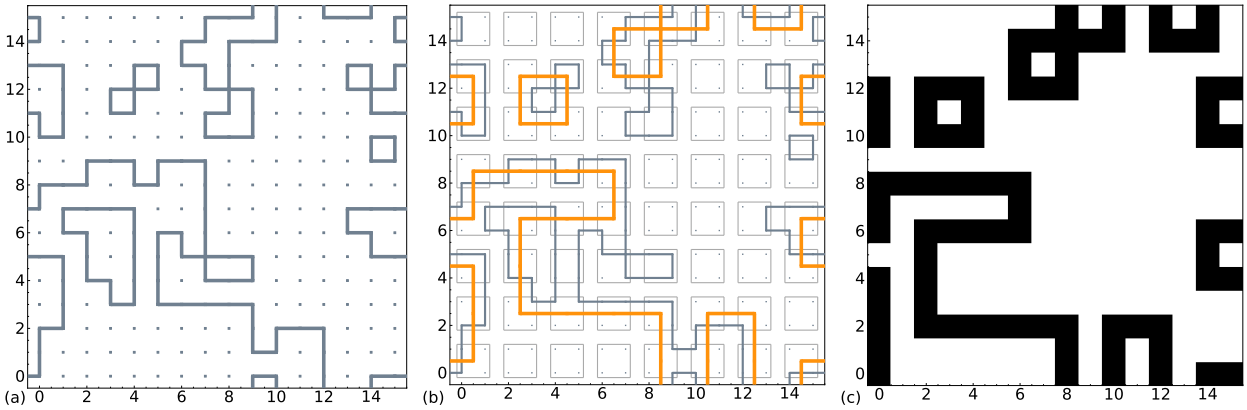


Figure 3.7: (a) Illustration of the $1 + 1 \longrightarrow 0$ blocking procedure discussed in the text: original configuration; (b) introduction of the blocks and replacement of single or double bounds according to the $1 + 1 \longrightarrow 0$ rule; (c) construction of the corresponding blocked image.

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 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 $1 + 1 \longrightarrow 0$ 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 $1 + 1 \longrightarrow 0$ 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 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.

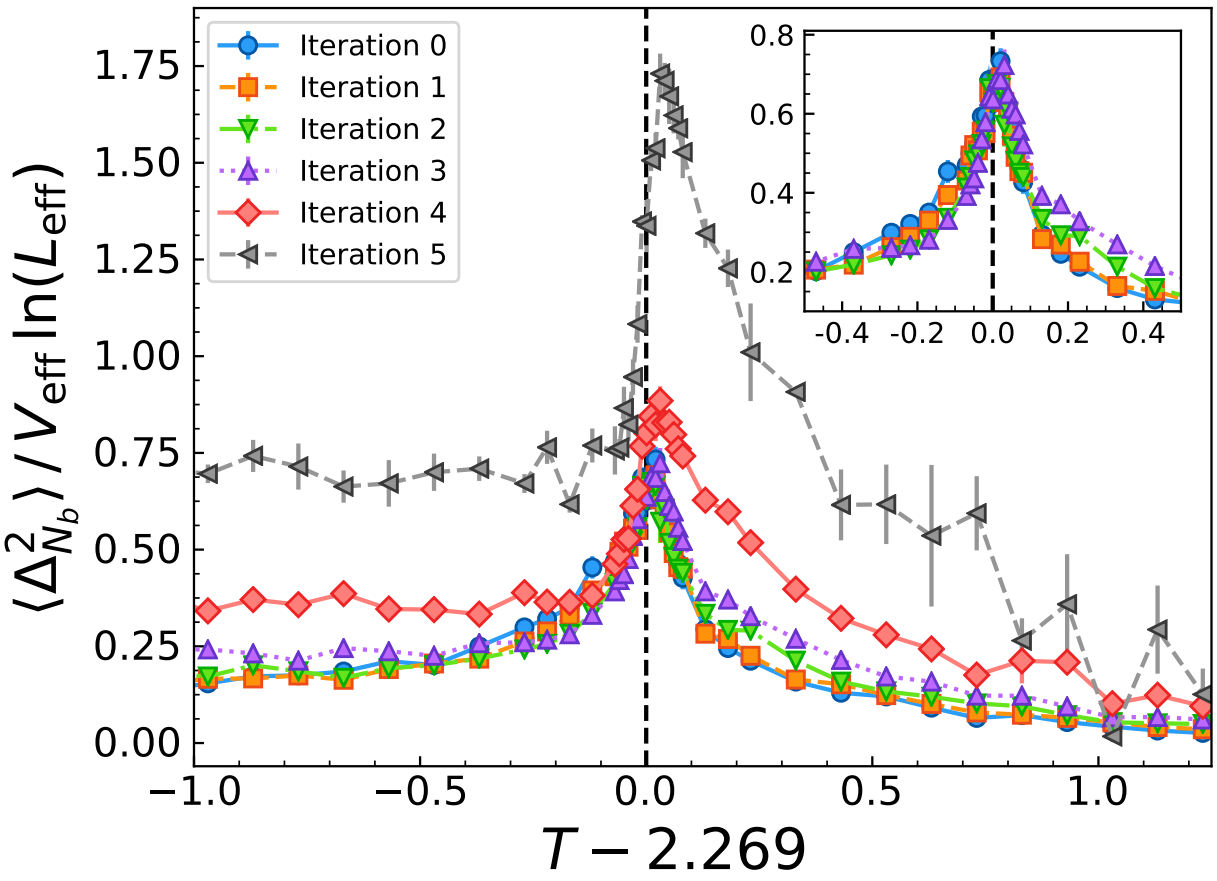


Figure 3.8: Fluctuations in the average number of bonds $\langle \Delta^2_{N_b} \rangle$ vs. temperature T under iterated blocking steps beginning with an initial lattice size of $L = 64$. The results are scaled by $1/V_{\text{eff}} \log(L_{\text{eff}})$ in order to demonstrate the data collapse near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

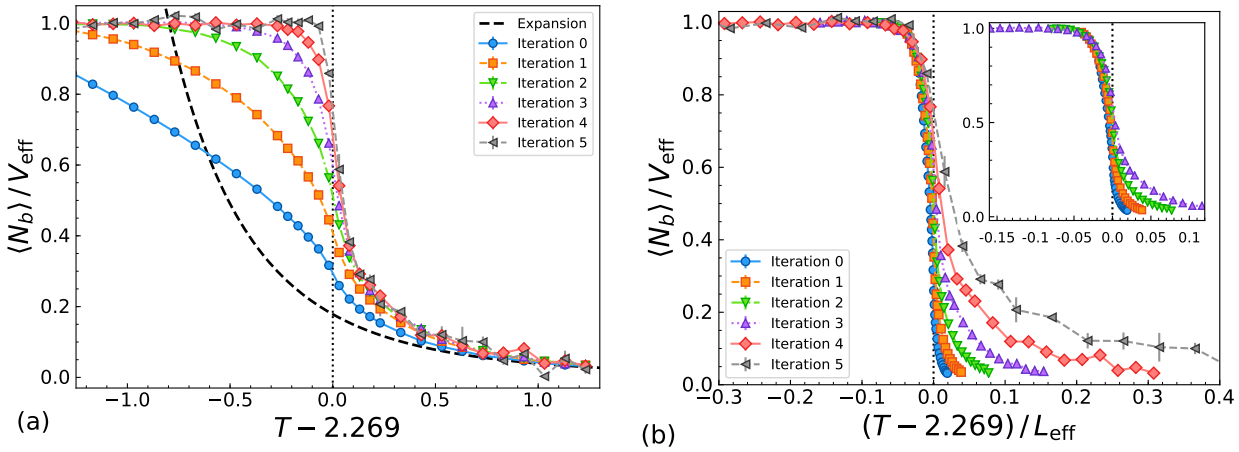


Figure 3.9: (a) Average number of bonds $\langle N_b \rangle$ vs. temperature T under iterated blocking steps beginning with an initial lattice size of $L = 64$. The dashed black line illustrates the high temperature expansion, showing that the dominant configurations are those consisting of small, isolated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

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

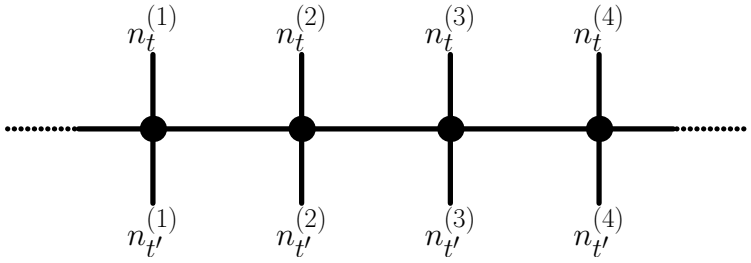


Figure 3.10: A pictorial representation of the transfer matrix made by contracting a fundamental tensor along a single time-slice.

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 a blocked tensor to the desired lattice size.

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

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

Once the original transfer matrix has been constructed, as well as the matrix with the insertion of n

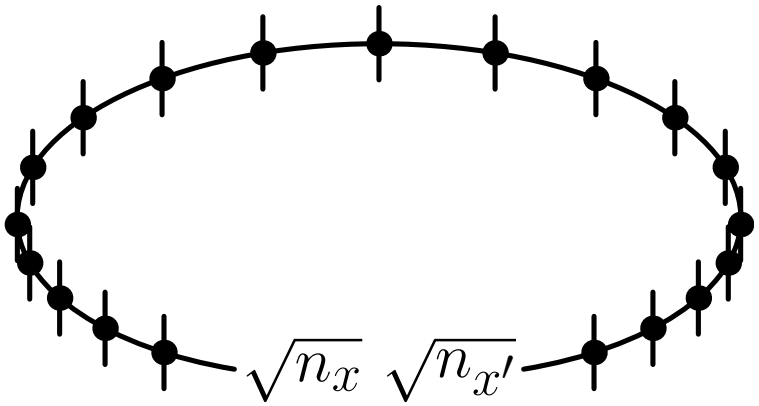


Figure 3.11: By multiplying the remaining free spatial indices by $\sqrt{n}$ and contracting for periodic boundary conditions in space we form an "impure" transfer matrix. Combining the resultant matrix with the original transfer matrix allows one to calculate $\langle n \rangle$.

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.

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

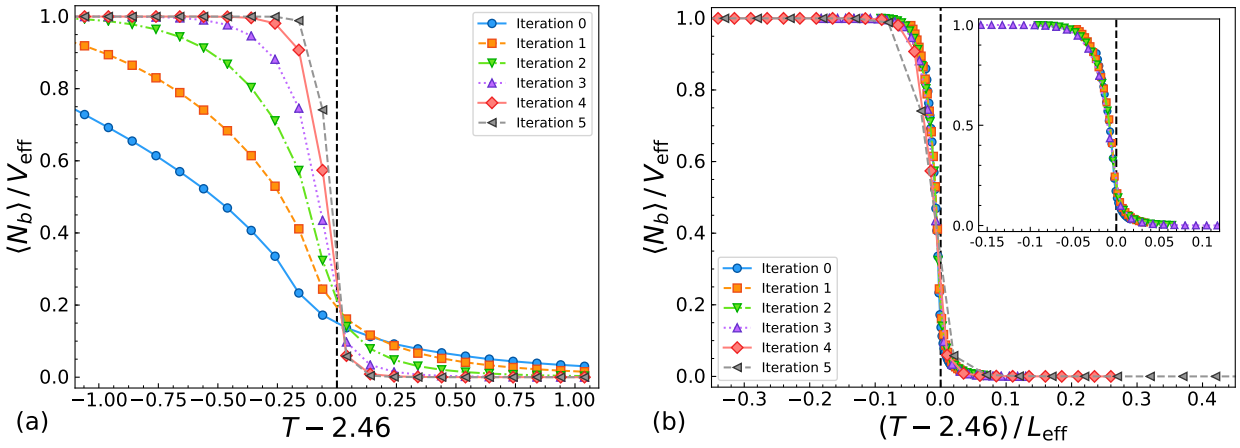


Figure 3.12: (a) $\langle N_b \rangle$ vs $(T - 2.46)$ under successive blocking steps calculated using 2-state HOTRG. (b) $\langle N_b \rangle$ vs $(T - 2.46)/L_{\text{eff}}$ under successive blocking steps calculated using 2-state HOTRG. Note that the value of 2.46 was determined qualitatively by choosing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

be written in the following high-temperature expansion:

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

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

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

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

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.

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

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

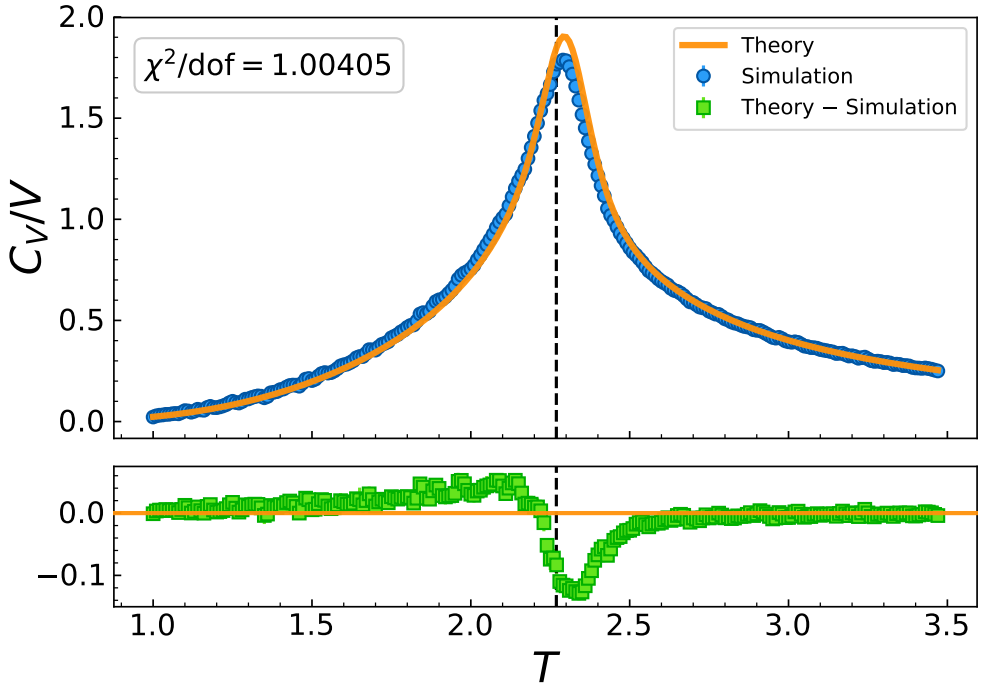


Figure 3.13: Comparison of the worm Monte Carlo computation of the specific heat C_v versus temperature and the exact results using the formula in [37], for an $L = 32$ lattice. Note that χ^2/dof represents the reduced chi-squared statistic. It can be seen that the agreement is excellent, except for 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.

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 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

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

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

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

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

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

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

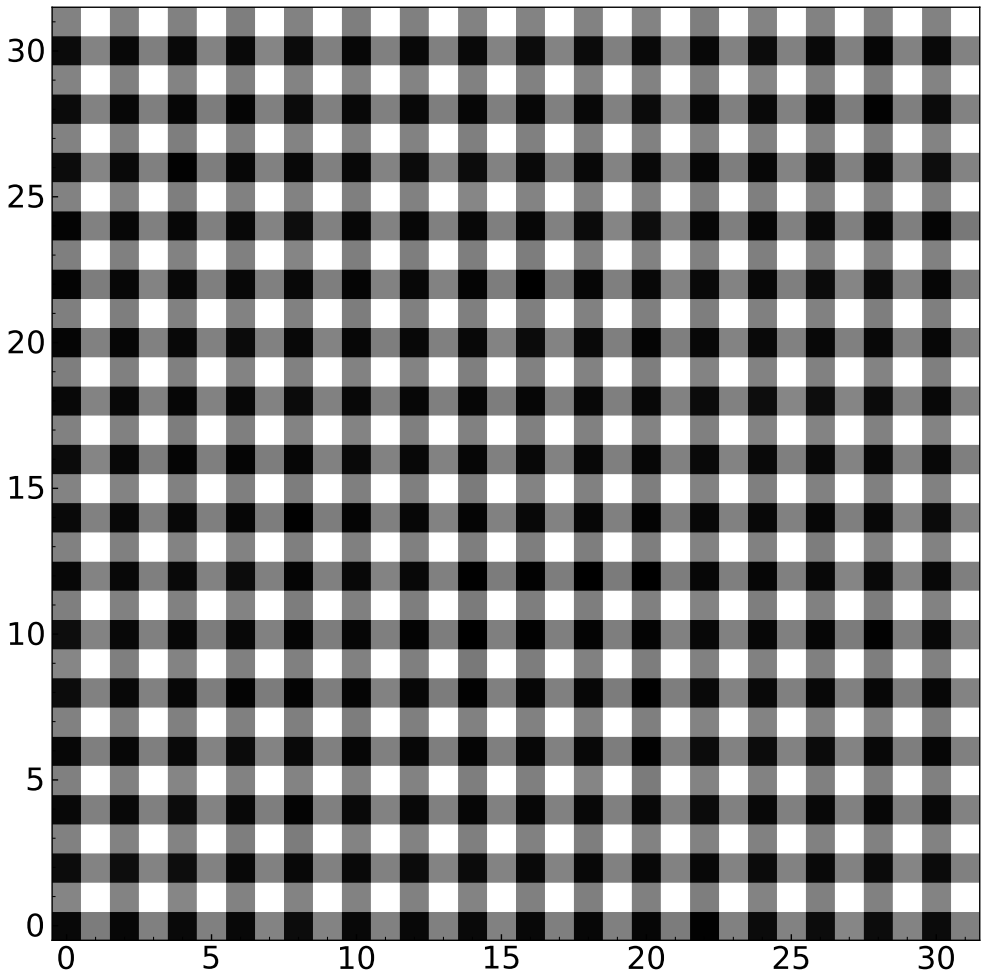


Figure 3.14: Average image $\langle \mathbf{v} \rangle$ calculated for the $L = 16$ lattice at $T = 2.0$, illustrating the tablecloth-like appearance.

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*,

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

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

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.

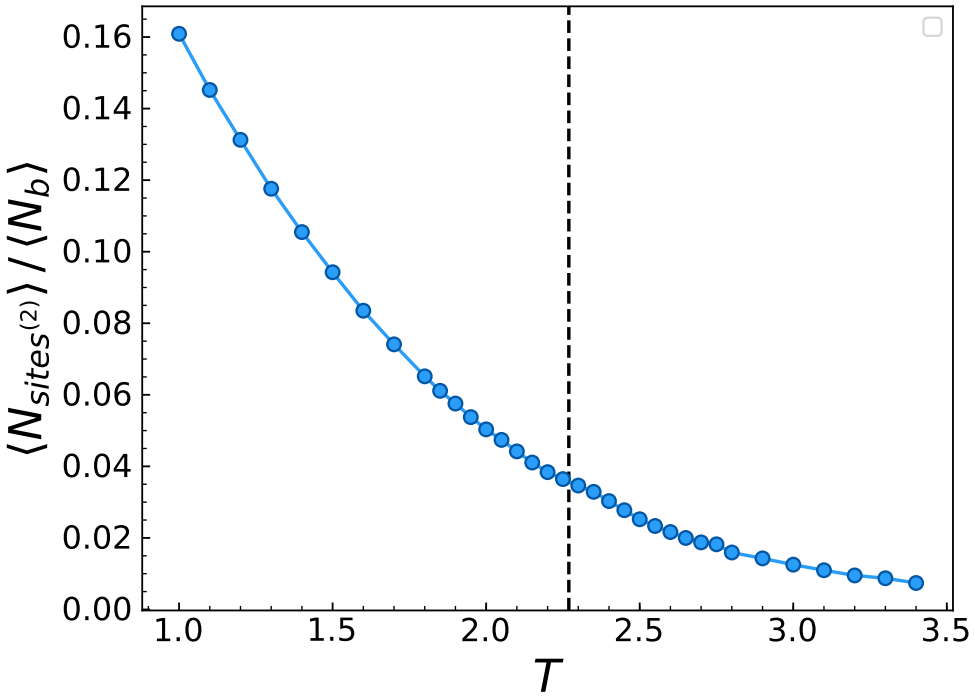


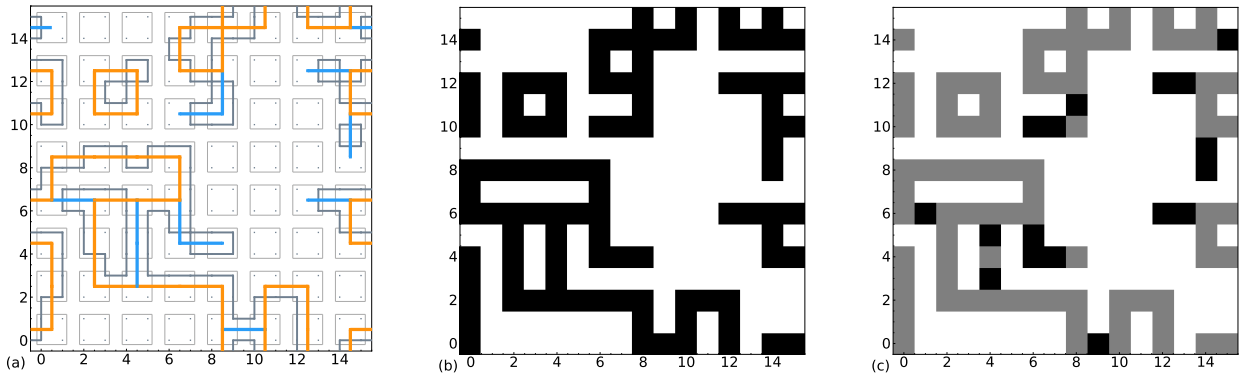
Figure 3.15: Ratio of the number of twice visited sites $\langle N_{sites^{(2)}} \rangle$ to the average number of bonds $\langle N_b \rangle$ versus temperature, for the $L = 32$ lattice. This clearly justifies our approximation $\langle \mathbf{v}_s \rangle \simeq 2\langle \mathbf{v}_b \rangle$, where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

3.8.6 Illustration of Alternate Blockings

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

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-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



(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.

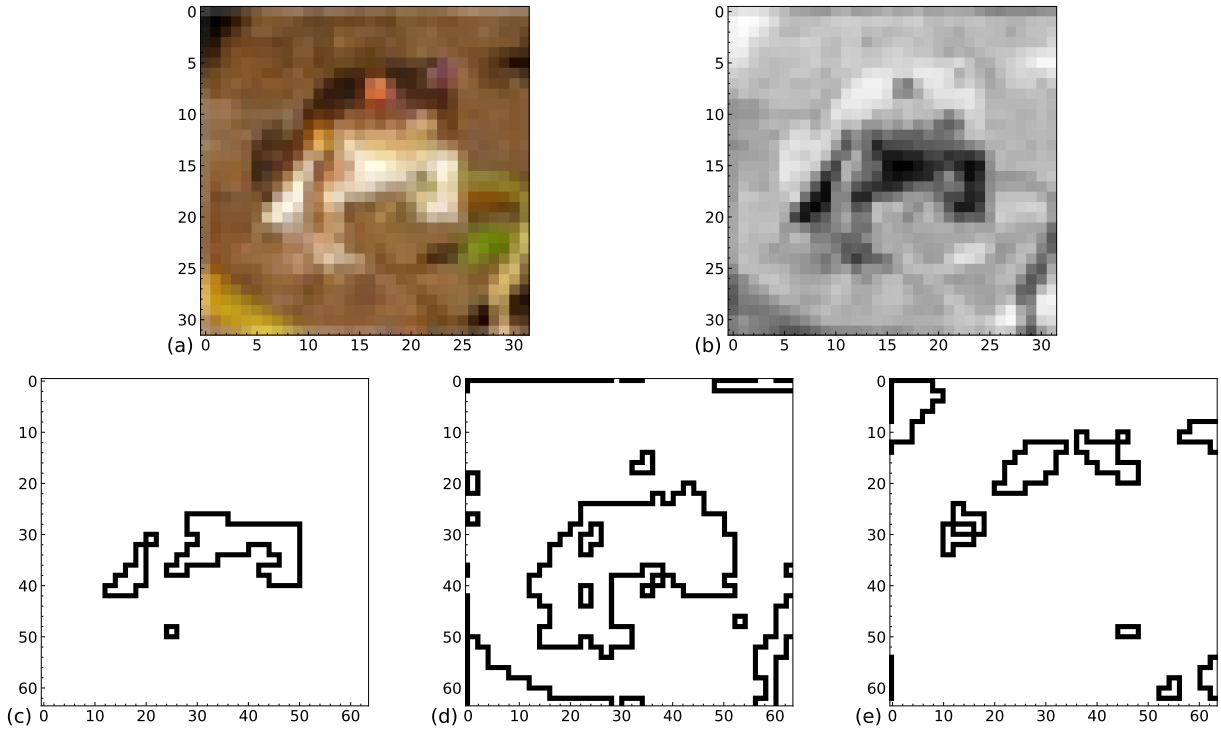


Figure 3.17: Example of preprocessing steps for converting CIFAR-10 images to 'worm-like' images, illustrating the resulting image for different values of the grayscale cuttoff. (a) Original image from CIFAR-10 dataset. (b) Image converted to grayscale. (c) Resulting image from cutoff values of 0.25,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

these images were then iteratively blocked using the $(1 + 1 \rightarrow 0)$ blocking procedure described in Sec. 3.4, calculating $\langle N_b \rangle$ and $\langle \Delta^2_{N_b} \rangle$ for each successive blocking step, as shown in Fig. 3.18. Immediately we see that

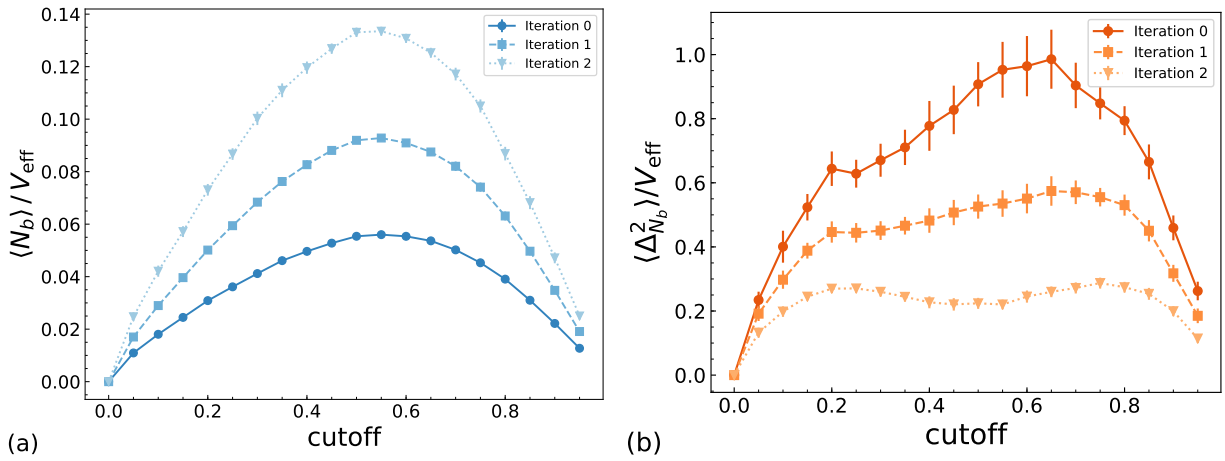


Figure 3.18: $\langle N_b \rangle$ and $\langle \Delta^2_{N_b} \rangle$ vs. grayscale cutoff value for 500 randomly chosen images from the CIFAR-10 dataset.

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

3.10 Conclusions

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

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

RG methods have been considered for assisting in image recognition [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.

Algorithm 2: Metropolis-Hastings Algorithm

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

Initialize $x_0 \sim \pi_0$.
for $t = 0$ **to** N **:**
  Sample $x' \sim q(\cdot|x_t)$
  Compute the acceptance probability: $A(x'|x_t) = \min\left(1, \frac{p(x')q(x_t|x')}{p(x_t)q(x'|x_t)}\right)$
  With probability A accept the proposed value and set $x_{t+1} = x'$. Otherwise set $x_{t+1} = x_t$.

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

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

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

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

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

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.

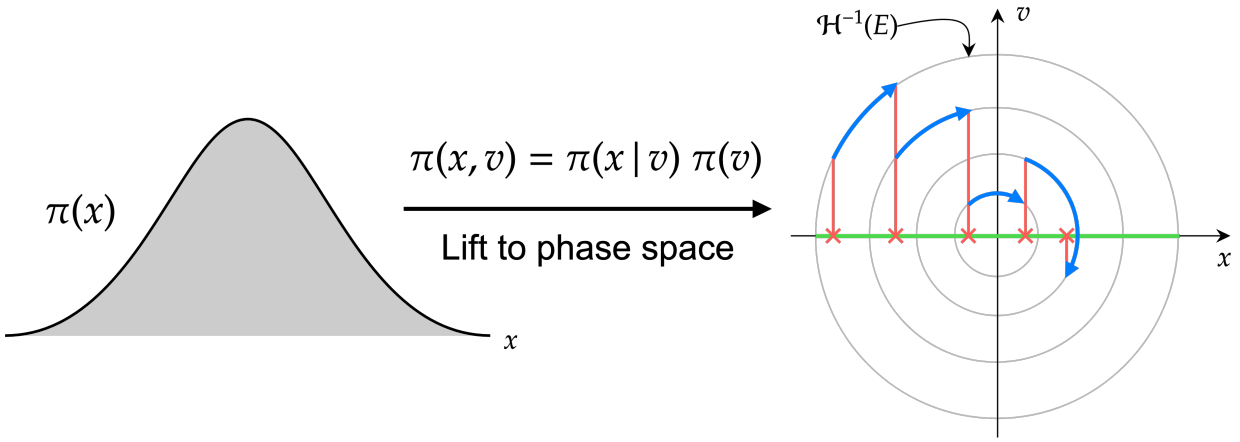


Figure 4.1: *Visualizing HMC for a 1D Gaussian* (example from [49], figure adapted with permission from [50]). Each Hamiltonian Markov transition lifts the initial state onto a random level set of the Hamiltonian, $\mathcal{H}^{(-1)}(E)$, which can then be explored with a Hamiltonian trajectory before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

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 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

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

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 update 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 $\mathbf{FL}_\theta\xi = \xi'$, 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:

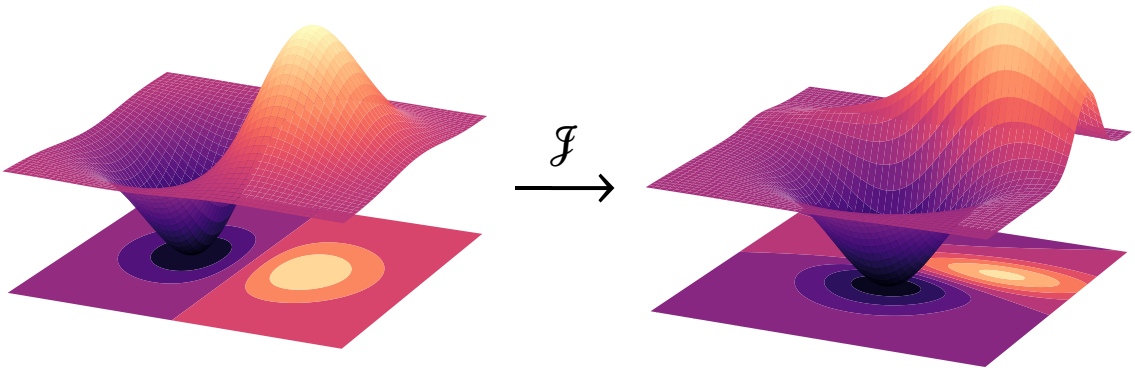


Figure 5.1: Example of how the determinant of the Jacobian can deform the energy landscape.

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 ($d = -1$):

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

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

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

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

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:

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

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

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.

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 $\tau(t) = \left(\cos\left(\frac{2\pi t}{N_{\text{LF}}}\right),\ \sin\left(\frac{2\pi t}{N_{\text{LF}}}\right)\right)$.

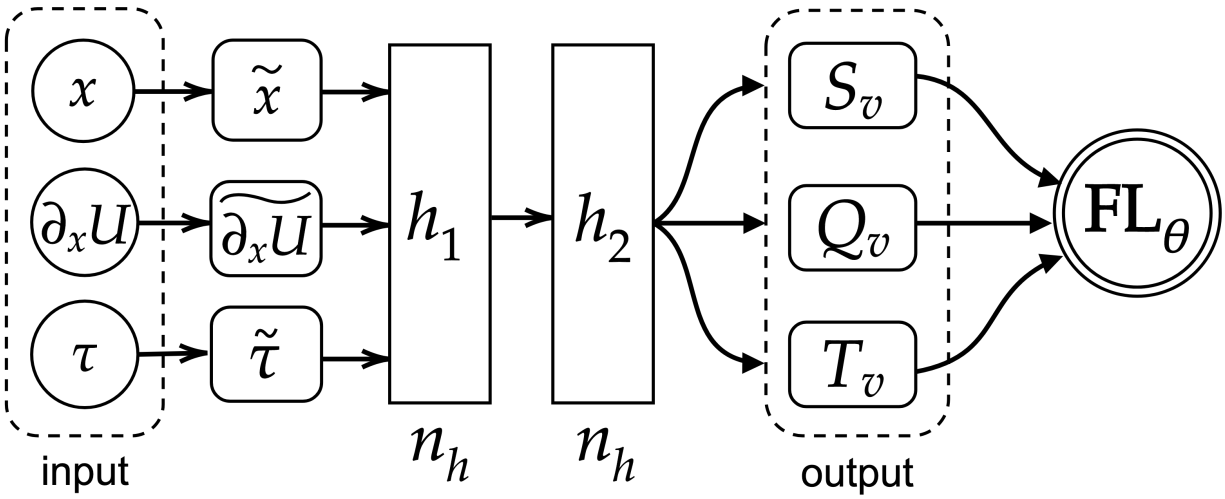


Figure 5.2: Illustration showing the generic (fully-connected) network architecture for training S_v, Q_v, and T_v. Figure adapted with permission from [50].

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.

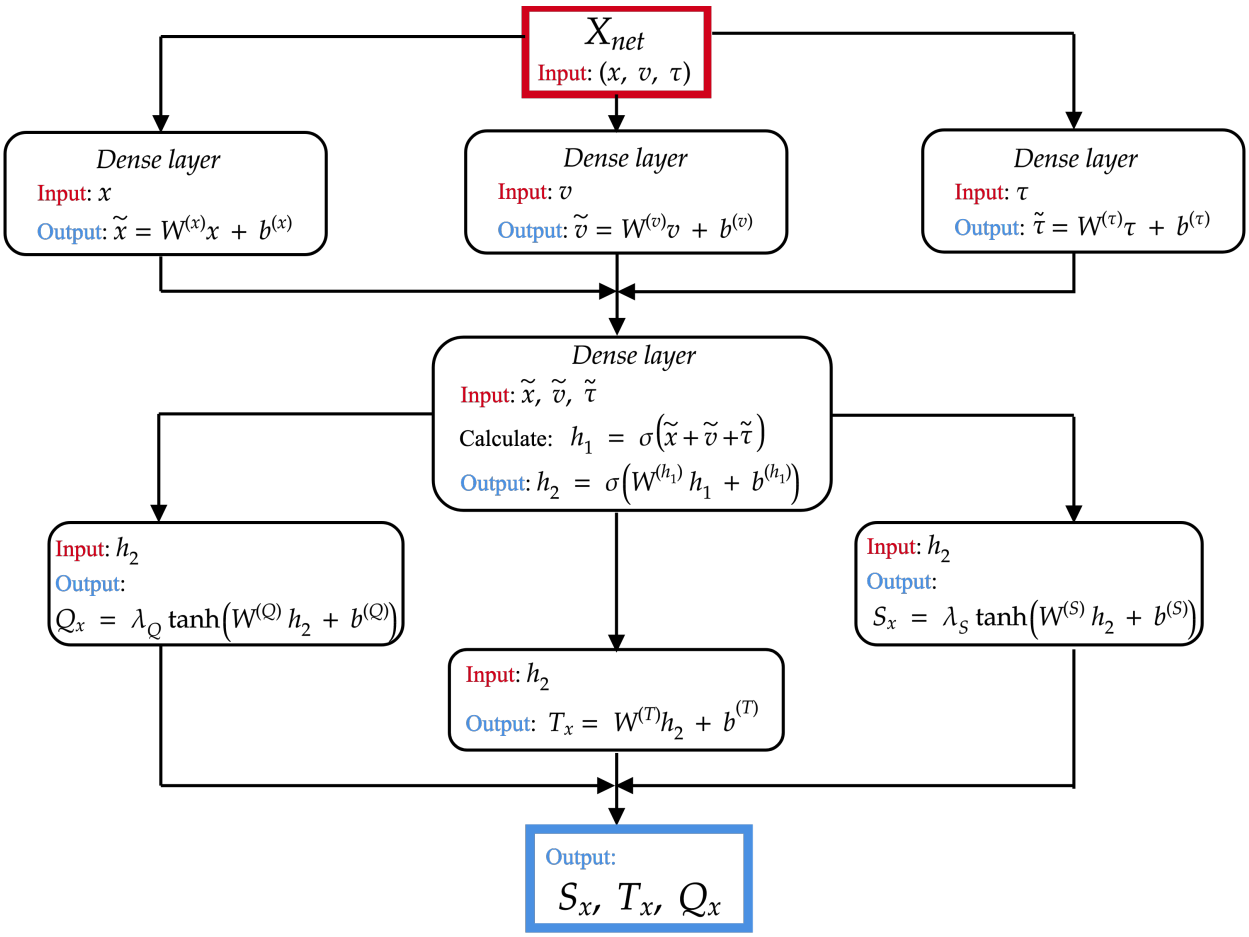


Figure 5.3: Flowchart illustrating the generic fully-connected network architecture including the intermediate variables computed at each hidden layer of the network.

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:

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.

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

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 $A(\mathbf{FL}_\theta \xi_p^{(i)} | \xi_p^{(i)})$
  $\theta \leftarrow \theta - \alpha_t \nabla_\theta \mathcal{L}$

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 component 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".

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

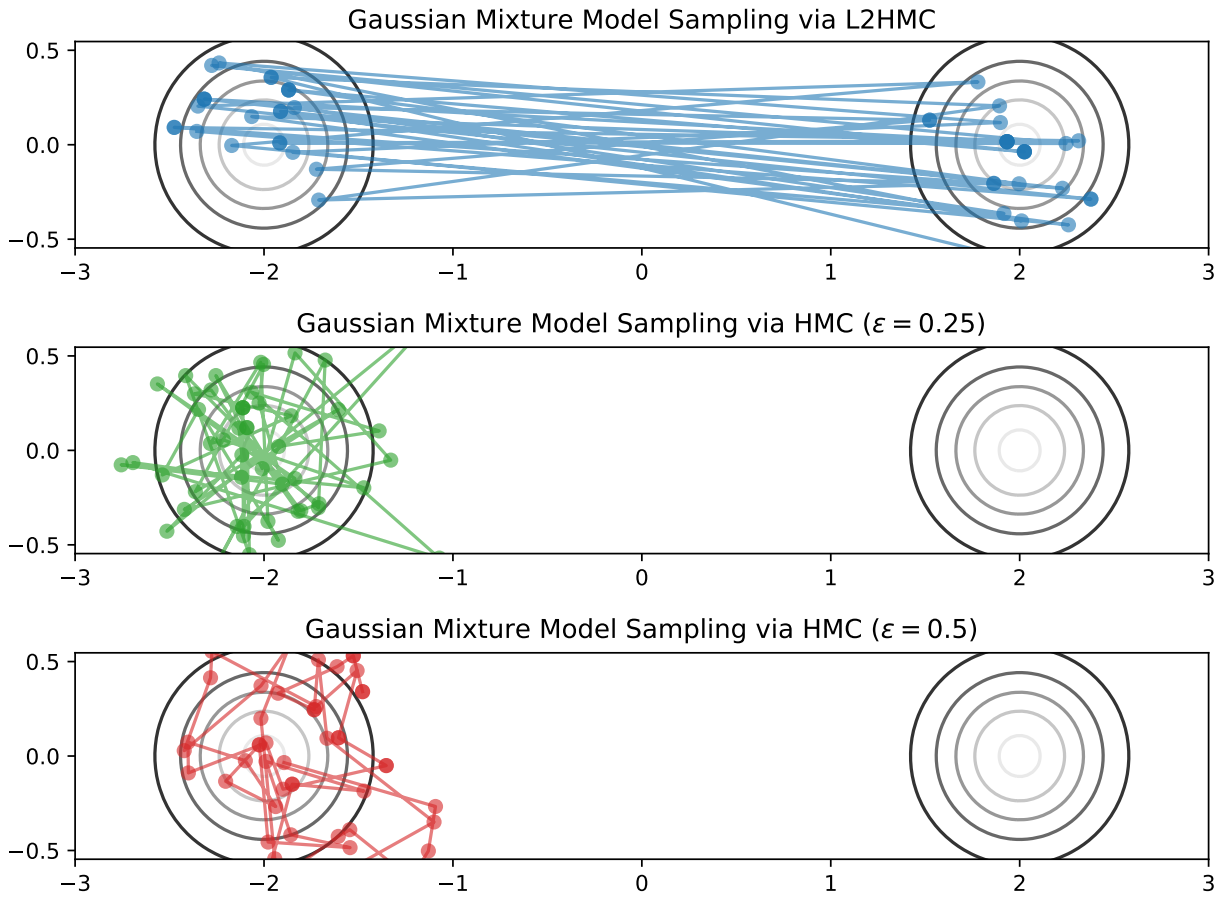


Figure 5.4: Comparison of trajectories generated using L2HMC (top), and traditional HMC with $\varepsilon = 0.25$ (middle) and $\varepsilon = 0.5$ (bottom). Note that L2HMC is able to successfully mix between modes, whereas HMC is not.

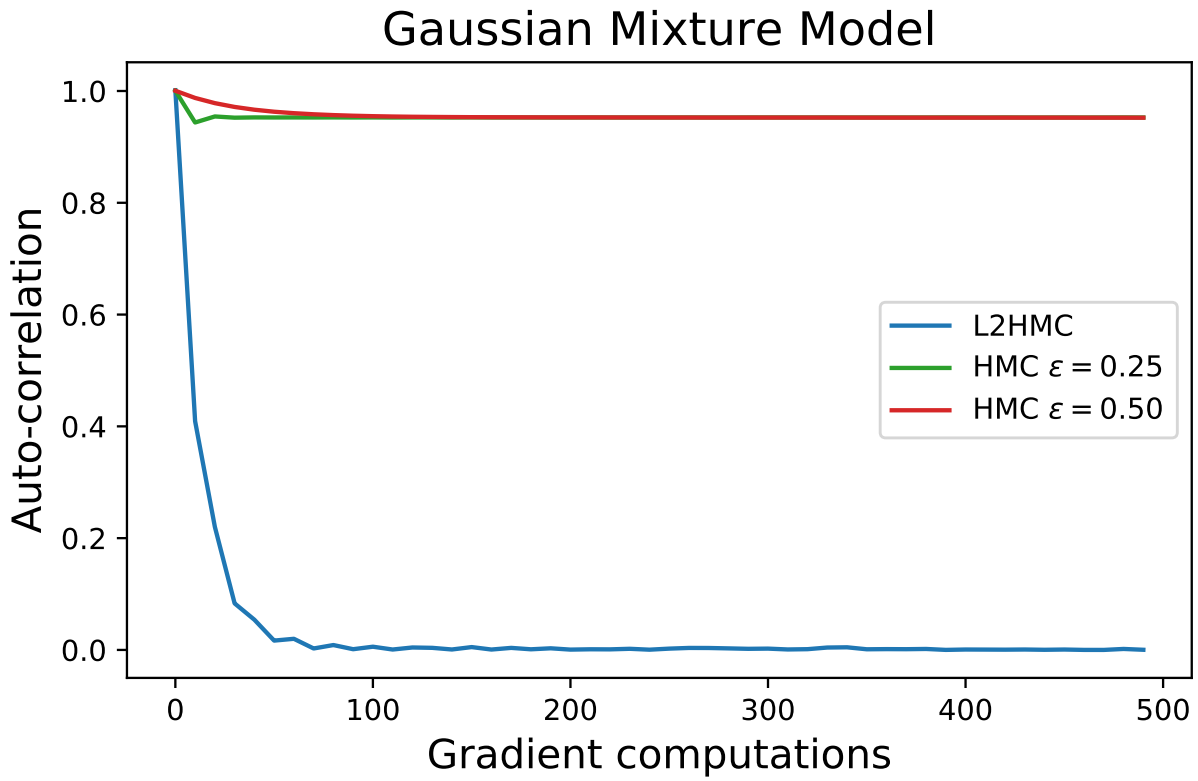


Figure 5.5: Autocorrelation vs. gradient evaluations (i.e. MD steps). Note that L2HMC (blue) has a significantly reduced autocorrelation after the same number of gradient evaluations when compared to either of the two HMC trajectories

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 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:

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

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

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

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

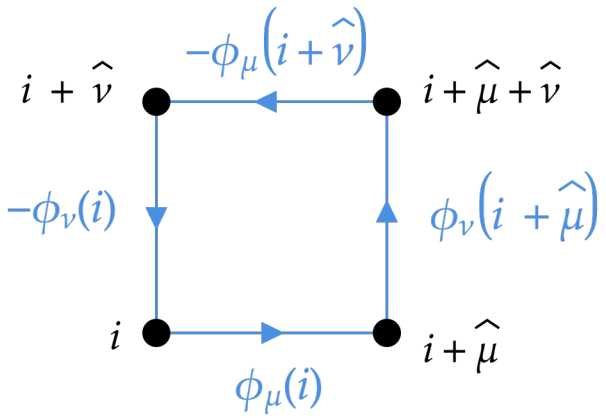


Figure 5.6: Illustration of an elementary plaquette on the lattice.

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 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].

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

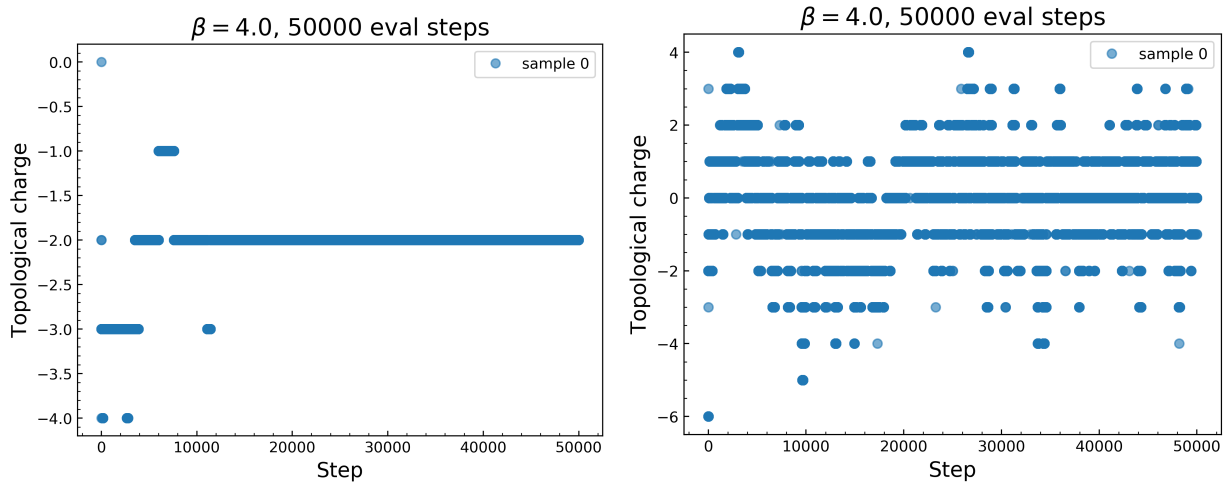


Figure 5.7: (left) Example of topological freezing in the $2D$ $U(1)$ lattice gauge theory, generated from generic HMC sampling for a 16×16 lattice. Note that for the majority of the simulation $Q = -2$, making it virtually impossible to get a reasonable estimate of χ. (right) Topological charge vs.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

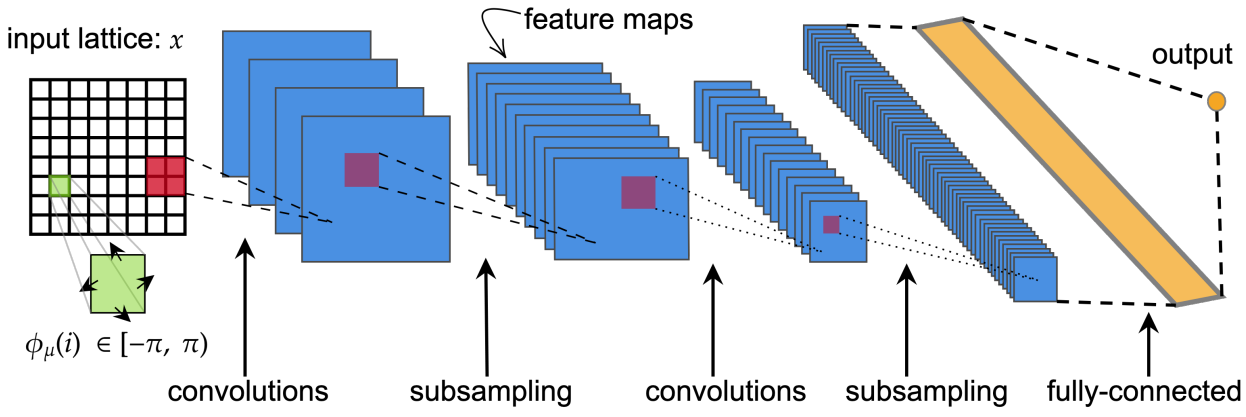


Figure 5.8: Convolutional structure used for learning localized features of rectangular lattice.

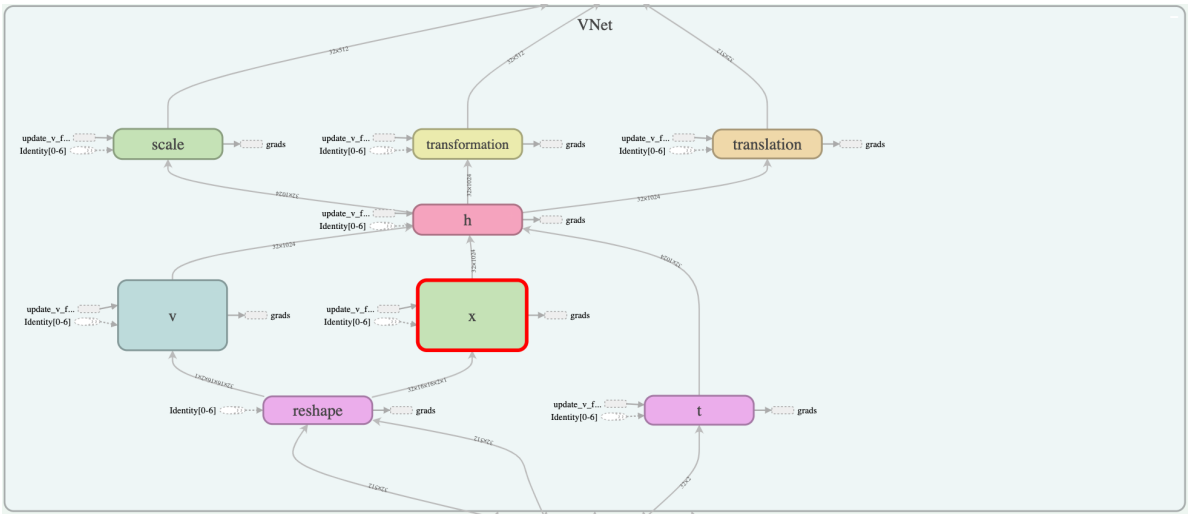


Figure 5.9: Illustration taken from TensorBoard showing an overview of the network architecture for VNet. Note that the architecture is identical for XNet.

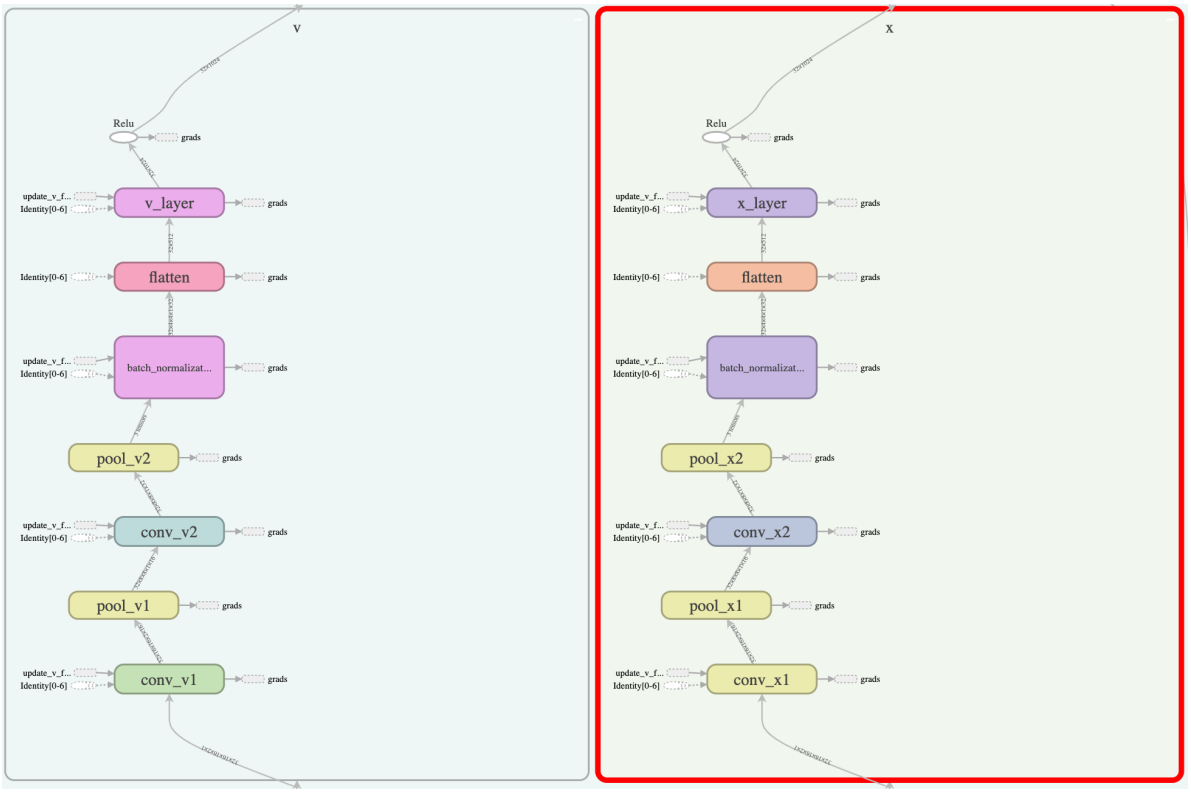


Figure 5.10: Detailed view of additional convolutional structure included to better account for rectangular geometry of lattice inputs.

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

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

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 $\xi \approx \xi'$.

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

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

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

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

[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.

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

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

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

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

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

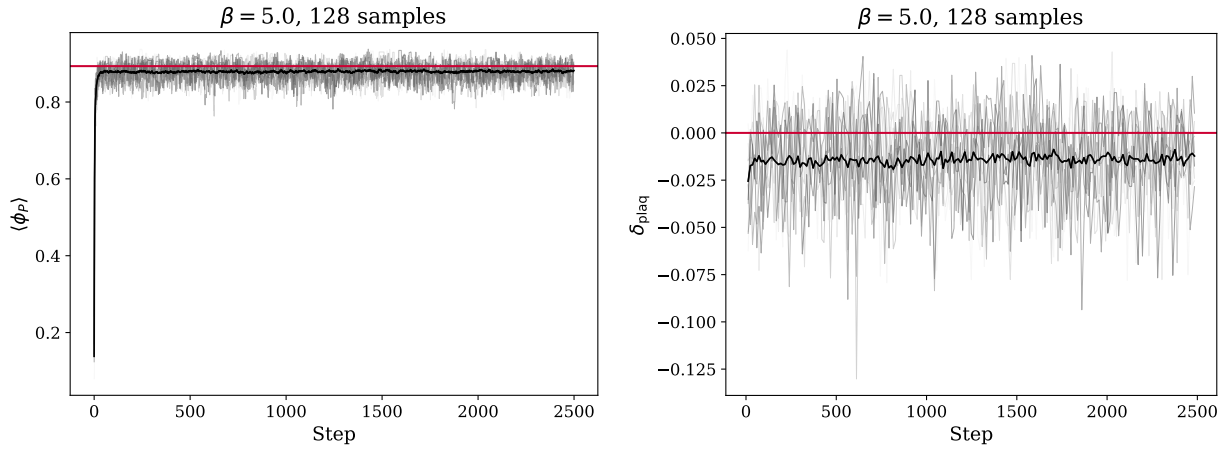


Figure 5.11: (left): Average plaquette $\langle \phi_P \rangle$ vs. step for $L = 8$, $N_{\text{LF}} = 7$, and $\alpha_Q = 0.5$. Here the solid red line indicates the true value of the average plaquette (in the infinite volume limit, and is equal to 0.89338...) **(right)**: Difference between the observed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

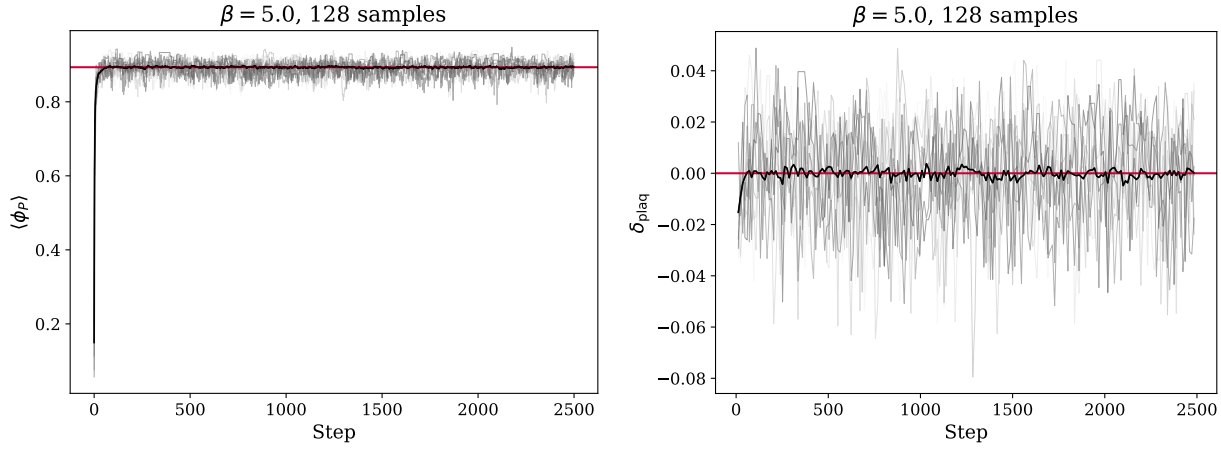


Figure 5.12: Same quantities as in Fig 5.11, with $\alpha_Q = 0$. Note that the discrepancy δ_{ϕ_P} is no longer present.

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 supercomputers. Because of this, even minor improvements in efficiency can dramatically reduce the amount of computational power required, allowing for increasingly complex models to be studied. The pursuit of better, more efficient algorithms is one of the major long-term goals of the lattice community, and is directly aligned with many of the goals outlined for high energy physics in the era of exascale computing. In particular, the results of these simulations are of central importance to the experiments being carried out at the Relativistic Heavy Ion Collider at Brookhaven National Laboratory (BNL), and the Large Hadron Collider (LHC) at CERN.

A | Appendix: L2HMC Source Code

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

A.1 README.md

A.1.1 l2hmc-qcd

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

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

A.1.1.1 Overview

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

Given an analytically described 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`.

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.

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

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_dyn` 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/args.txt

```
#================================================================#
#                                                                #
#    To pass the arguments defined below to `gauge_model.py`:    #
#        python3 gauge_model.py @gauge_model_args.txt            #
#                                                                #
#================================================================#
--time_size 8
--space_size 8
--rand
--num_samples 100
--num_steps 16
--eps 0.2
--beta_init 2.
--beta_final 5.
--lr_init 0.001
--lr_decay_steps 1000
--lr_decay_rate 0.96
--train_steps 1000
--logging_steps 10
--save_steps 1000
--print_steps 1
--network_arch 'generic'
--num_hidden1 50
--num_hidden2 50
--metric 'cos_diff'
--aux_weight 1.
--std_weight 1.
--charge_weight 0.
--loss_scale 0.1
--save_lf
######################
#--float64
#--zero_translation
#--warmup_lr
#--use_bn
#--dropout_prob 0.5
#--nnehmc_loss
#--inference
#--loop_net_weights


#=============================================================================================
#                    Description of command line arguments defined above                     #
#=============================================================================================
#   time_size:                      temporal extent of lattice                               #
#   space_size:                     spatial extent of lattice                                #
#   num_samples:                    number of samples per batch                              #
#   num_steps:                      number of leapfrog steps in MD update                    #
#   eps:                            step size in leapfrog integrator                         #
#   annealing:                      continuously anneal beta analogous to temperature)       #
#   beta_init:                      initial value of beta used in annealing schedule         #
#   beta_final:                     final value of beta in annealing schedule                #
#   lr_init:                        initial value of learning rate                           #
#   lr_decay_steps:                 steps after which lr is decayed by lr_decay_rate         #
#   lr_decay_rate:                  factor by which lr is decayed                            #
#   train_steps:                    number of training steps to perform                      #
#   train_steps:                    number of evaluation steps to run trained sampler for    #
#   save_steps:                     when to save model checkpoint                            #
#   print_steps:                    how often to print information during train ing          #
#   training_samples_steps:         when to run sampler during training                      #
#   training_samples_length:        how long sampler is ran for during traini ng             #
#   network_arch:                   network architecture ('conv3D', 'conv2D' , 'generic')    #
#   metric:                         must be one of 'l1', 'l2', 'cos2', 'cos _diff'           #
#   aux:                            include auxiliary variable in loss fun ction             #
#   aux_weight:                     factor multiplying aux. var contribution to loss fn      #
#   std_weight:                     factor multiplying std. var contribution to loss fn      #
#   charge_weight:                  factor multiplying charge. var contribution to loss fn   #
```

```
#   loss_scale:                    factor by which the loss is multiplie d                #
#   plaq_loss:                     include top. charge difference in th e loss fn         #
#   summaries:                     create summary objects for tensorbo ard                #
#   eps_trainable:                 allows the step size to be a train able paramet er     #
#============================================================================================
```

A.3 l2hmc-qcd/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
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os
import random
import time
import pickle
import inference
import tensorflow as tf
import numpy as np

from tensorflow.python import debug as tf_debug  # noqa: F401
from tensorflow.python.client import timeline    # noqa: F401
from tensorflow.core.protobuf import rewriter_config_pb2

import utils.file_io as io

from config import (
    GLOBAL_SEED, NP_FLOAT, HAS_HOROVOD, HAS_COMET, HAS_MATPLOTLIB
)
from update import set_precision
from utils.parse_args import parse_args
from models.model import GaugeModel
from loggers.train_logger import TrainLogger
from trainers.trainer import GaugeModelTrainer

if HAS_COMET:
    from comet_ml import Experiment

if HAS_HOROVOD:
    import horovod.tensorflow as hvd

#  if HAS_MATPLOTLIB:
#      import matplotlib.pyplot as plt
```

```
68
69  if float(tf.__version__.split('.')[0]) <= 2:
70      tf.logging.set_verbosity(tf.logging.INFO)
71
72  SEP_STR = 80 * '-'  # + '\n'
73
74  # ------------------------------------------
75  # Set random seeds for tensorflow and numpy
76  # ------------------------------------------
77  os.environ['PYTHONHASHSEED'] = str(GLOBAL_SEED)
78  random.seed(GLOBAL_SEED)        # `python` build-in pseudo-random generator
79  np.random.seed(GLOBAL_SEED)     # numpy pseudo-random generator
80  tf.set_random_seed(GLOBAL_SEED)
81
82
83  def create_config(params):
84      """Helper method for creating a tf.ConfigProto object."""
85      config = tf.ConfigProto(allow_soft_placement=True)
86      if params['time_size'] > 8:
87          off = rewriter_config_pb2.RewriterConfig.OFF
88          config_attrs = config.graph_options.rewrite_options
89          config_attrs.arithmetic_optimization = off
90
91      if params['gpu']:
92          # Horovod: pin GPU to be used to process local rank
93          # (one GPU per process)
94          config.gpu_options.allow_growth = True
95          #  config.allow_soft_placement = True
96          if HAS_HOROVOD and params['horovod']:
97              config.gpu_options.visible_device_list = str(hvd.local_rank())
98
99      if HAS_MATPLOTLIB:
100         params['_plot'] = True
101
102     if params['theta']:
103         params['_plot'] = False
104         io.log("Training on Theta @ ALCF...")
105         params['data_format'] = 'channels_last'
106         os.environ["KMP_BLOCKTIME"] = str(0)
107         os.environ["KMP_AFFINITY"] = (
108             "granularity=fine,verbose,compact,1,0"
109         )
110         # NOTE: KMP affinity taken care of by passing -cc depth to aprun call
111         OMP_NUM_THREADS = 62
112         config.allow_soft_placement = True
113         config.intra_op_parallelism_threads = OMP_NUM_THREADS
114         config.inter_op_parallelism_threads = 0
115
116     return config, params
117
118
119
120 def latest_meta_file(checkpoint_dir=None):
121     """Returns the most recent meta-graph (`.meta`) file in checkpoint_dir."""
122     if not os.path.isdir(checkpoint_dir) or checkpoint_dir is None:
123         return
124
125     meta_files = [i for i in os.listdir(checkpoint_dir) if i.endswith('.meta')]
126     step_nums = [int(i.split('-')[-1].rstrip('.meta')) for i in meta_files]
127     step_num = sorted(step_nums)[-1]
128     meta_file = os.path.join(checkpoint_dir, f'model.ckpt-{step_num}.meta')
129
130     return meta_file
131
132
133 def count_trainable_params(out_file, log=False):
134     """Count the total number of trainable parameters in a tf.Graph object.
135
136     Args:
137         out_file (str): Path to file where all trainable parameters will be
```

```
            written.
        log (bool): Whether or not to print trainable parameters to console
            (std-out).
    Returns:
        None
    """
    if log:
        writer = io.log_and_write
    else:
        writer = io.write

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

    writer(80 * '-', out_file)
    writer(f'Total parameters: {total_params}', out_file)


def train_setup(FLAGS, log_file=None):
    io.log(SEP_STR)
    io.log("Starting training using L2HMC algorithm...")
    tf.keras.backend.clear_session()
    tf.reset_default_graph()

    # ------------------------------------------------------------------------
    # Parse command line arguments; copy key, val pairs from FLAGS to params.
    # ------------------------------------------------------------------------
    try:
        kv_pairs = FLAGS.__dict__.items()
    except AttributeError:
        kv_pairs = FLAGS.items()

    params = {}
    for key, val in kv_pairs:
        params[key] = val
        #  if isinstance(val, bool) and not val:  # skip if option is `off`
        #      continue
        #  else:

    params['log_dir'] = io.create_log_dir(FLAGS, log_file=log_file)
    params['summaries'] = not FLAGS.no_summaries
    if 'no_summaries' in params:
        del params['no_summaries']

    if FLAGS.save_steps is None and FLAGS.train_steps is not None:
        params['save_steps'] = params['train_steps'] // 4

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

```

```
    if FLAGS.float64:
        io.log(f'INFO: Setting floating point precision to `float64`.')
        set_precision('float64')

    if FLAGS.horovod:
        params['using_hvd'] = True
        num_workers = hvd.size()
        params['num_workers'] = num_workers

        # ------------------------------------------------------
        # Horovod: Scale initial lr by of num GPUs.
        # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        # NOTE: Even with a linear `warmup` of the learning rate,
        #       the training remains unstable as evidenced by
        #       exploding gradients and NaN tensors.
        # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        #  params['lr_init'] *= num_workers
        # ------------------------------------------------------

        # Horovod: adjust number of training steps based on number of GPUs.
        #  params['train_steps'] //= num_workers

        # Horovod: adjust save_steps and lr_decay_steps accordingly.
        #  params['save_steps'] //= num_workers
        #  params['lr_decay_steps'] //= num_workers

        #  if params['summaries']:
        #      params['logging_steps'] //= num_workers

        # ------------------------------------------------------
        # Horovod: BroadcastGlobalVariablesHook broadcasts initial
        # variable states from rank 0 to all other processes. This
        # is necessary to ensure consistent initialization of all
        # workers when training is started with random weights or
        # restored from a checkpoint.
        # ------------------------------------------------------
        hooks = [hvd.BroadcastGlobalVariablesHook(0)]
    else:
        params['using_hvd'] = False
        hooks = []

    return params, hooks


def train_l2hmc(FLAGS, log_file=None, experiment=None):
    """Create, train, and run L2HMC sampler on 2D U(1) gauge model."""
    tf.keras.backend.set_learning_phase(True)
    params, hooks = train_setup(FLAGS, log_file)

    # ------------------------------------------------------------
    # NOTE: Conditionals required for file I/O if we're not using
    #       Horovod, `is_chief` should always be True otherwise,
    #       if using Horovod, we only want to perform file I/O
    #       on hvd.rank() == 0, so check that first.
    # ------------------------------------------------------------
    condition1 = not params['using_hvd']
    condition2 = params['using_hvd'] and hvd.rank() == 0
    is_chief = condition1 or condition2

    if is_chief:
        #  assert FLAGS.log_dir == params['log_dir']
        log_dir = params['log_dir']
        checkpoint_dir = os.path.join(log_dir, 'checkpoints/')
        io.check_else_make_dir(checkpoint_dir)

    else:
        log_dir = None
        checkpoint_dir = None

    io.log(SEP_STR)
```

```
    io.log('L2HMC PARAMETERS:')
    for key, val in params.items():
        io.log(f'  {key}: {val}')
    io.log(SEP_STR)

    # ------------------------------------------------------
    # Create model and train_logger
    # ------------------------------------------------------
    model = GaugeModel(params)

    if is_chief:
        train_logger = TrainLogger(model, log_dir, params['summaries'])
    else:
        train_logger = None

    # ------------------------------------------------------
    # Setup config and init_feed_dict for tf.train.Scaffold
    # ------------------------------------------------------
    config, params = create_config(params)

    # set initial value of charge weight using value from FLAGS
    charge_weight_init = params['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,    # NOTE
        model.net_weights[0]: net_weights_init[0],  # ~ scale (S fn)
        model.net_weights[1]: net_weights_init[1],  # ~ translation (T fn)
        model.net_weights[2]: net_weights_init[2],  # ~ transformation (Q fn)
        model.train_phase: True                     # ------------------------
    }

    # ensure all variables are initialized
    target_collection = []
    if is_chief:
        collection = tf.local_variables() + target_collection
    else:
        collection = tf.local_variables()

    local_init_op = tf.variables_initializer(collection)
    ready_for_local_init_op = tf.report_uninitialized_variables(collection)
    init_op = tf.global_variables_initializer()

    scaffold = tf.train.Scaffold(
        init_feed_dict=init_feed_dict,
        local_init_op=local_init_op,
        ready_for_local_init_op=ready_for_local_init_op
    )

    # ------------------------------------------------------------------
    #  Create MonitoredTrainingSession
    #
    #  NOTE: The MonitoredTrainingSession takes care of session
    #        initialization, restoring from a checkpoint, saving to a
    #        checkpoint, and closing when done or an error occurs.
    # ------------------------------------------------------------------
    sess_kwargs = {
        'checkpoint_dir': checkpoint_dir,
        'scaffold': scaffold,
        'hooks': hooks,
        'config': config,
        'save_summaries_secs': None,
        'save_summaries_steps': None
    }
```

```
348       sess = tf.train.MonitoredTrainingSession(**sess_kwargs)
349       tf.keras.backend.set_session(sess)
350       sess.run(init_op)
351
352       # ------------------------------------------------------------
353       #                       TRAINING
354       # ------------------------------------------------------------
355       trainer = GaugeModelTrainer(sess, model, train_logger)
356       train_kwargs = {
357           'samples_np': samples_init,
358           'beta_np': beta_init,
359           'net_weights': net_weights_init
360       }
361
362       trainer.train(model.train_steps, **train_kwargs)
363
364       if HAS_COMET and experiment is not None:
365           experiment.log_parameters(params)
366           g = sess.graph
367           experiment.set_model_graph(g)
368
369       params_file = os.path.join(os.getcwd(), 'params.pkl')
370       with open(params_file, 'wb') as f:
371           pickle.dump(model.params, f)
372
373       # Count all trainable paramters and write them out (w/ shapes) to txt file
374       count_trainable_params(os.path.join(params['log_dir'],
375                                           'trainable_params.txt'))
376
377       # close MonitoredTrainingSession and reset the default graph
378       sess.close()
379       tf.reset_default_graph()
380
381       return model, train_logger
382
383
384   def main(FLAGS):
385       """Main method for creating/training/running L2HMC for U(1) gauge model."""
386       log_file = 'output_dirs.txt'
387
388       if HAS_HOROVOD and FLAGS.horovod:
389           io.log("INFO: USING HOROVOD")
390           hvd.init()
391
392       condition1 = not FLAGS.horovod
393       condition2 = FLAGS.horovod and hvd.rank() == 0
394       is_chief = condition1 or condition2
395
396       if FLAGS.comet and is_chief:
397           experiment = Experiment(api_key="r7rKFO35BJuaY3KT1Tpj4adco",
398                                   project_name="l2hmc-qcd",
399                                   workspace="saforem2")
400           name = (f'{FLAGS.network_arch}_'
401                   f'lf{FLAGS.num_steps}_'
402                   f'batch{FLAGS.num_samples}_'
403                   f'qw{FLAGS.charge_weight}_'
404                   f'aux{FLAGS.aux_weight}')
405           experiment.set_name(name)
406
407       else:
408           experiment = None
409
410       if FLAGS.hmc:
411           # --------------------
412           #   run generic HMC
413           # --------------------
414           inference.run_hmc(FLAGS, log_file=log_file)
415       else:
416           # ------------------------
417           #   train l2hmc sampler
```

```
        # ------------------------
        model, train_logger = train_l2hmc(FLAGS, log_file, experiment)
        if experiment is not None:
            experiment.log_parameters(model.params)


if __name__ == '__main__':
    args = parse_args()
    t0 = time.time()

    main(args)

    io.log('\n\n' + SEP_STR)
    io.log(f'Time to complete: {time.time() - t0:.4g}')
```

A.4 l2hmc-qcd/inference.py

```
"""
gauge_model_inference.py

Runs inference using the trained L2HMC sampler contained in a saved model.

This is done by reading in the location of the saved model from a .txt file
containing the location of the checkpoint directory.

 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  NOTE:
 ------------------------------------------------------------------------
   If `--plot_lf` CLI argument passed, create the
   following plots:

     * The metric distance observed between individual
       leapfrog steps and complete molecular dynamics
       updates.

     * The determinant of the Jacobian for each leapfrog
       step and the sum of the determinant of the Jacobian
       (sumlogdet) for each MD update.
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Author: Sam Foreman (github: @saforem2)
Date: 07/08/2019
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os
import time
import pickle
import tensorflow as tf
import numpy as np


import utils.file_io as io

from config import (PARAMS, NP_FLOAT, HAS_HOROVOD, HAS_MATPLOTLIB,
                    HAS_MEMORY_PROFILER)
from update import set_precision
#  from main import create_config
from tensorflow.core.protobuf import rewriter_config_pb2
#  from gauge_model_main import create_config

#  from utils.parse_args import parse_args
from utils.parse_inference_args import parse_args
from models.model import GaugeModel
from loggers.summary_utils import create_summaries
from loggers.run_logger import RunLogger
from plotters.gauge_model_plotter import GaugeModelPlotter
#  from plotters.plot_utils import plot_plaq_diffs_vs_net_weights
from plotters.leapfrog_plotters import LeapfrogPlotter
from runners.runner import GaugeModelRunner

if HAS_HOROVOD:
    import horovod.tensorflow as hvd

if HAS_MEMORY_PROFILER:
    import memory_profiler

#  if HAS_MATPLOTLIB:
#      import matplotlib.pyplot as plt

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

```
68
69
70  SEP_STR = 80 * '-'  # + '\n'
71
72
73  def create_config(params):
74      """Helper method for creating a tf.ConfigProto object."""
75      config = tf.ConfigProto(allow_soft_placement=True)
76      if params['time_size'] > 8:
77          off = rewriter_config_pb2.RewriterConfig.OFF
78          config_attrs = config.graph_options.rewrite_options
79          config_attrs.arithmetic_optimization = off
80
81      if params['gpu']:
82          # Horovod: pin GPU to be used to process local rank
83          # (one GPU per process)
84          config.gpu_options.allow_growth = True
85          #  config.allow_soft_placement = True
86          if HAS_HOROVOD and params['horovod']:
87              config.gpu_options.visible_device_list = str(hvd.local_rank())
88
89      if HAS_MATPLOTLIB:
90          params['_plot'] = True
91
92      if params['theta']:
93          params['_plot'] = False
94          io.log("Training on Theta @ ALCF...")
95          params['data_format'] = 'channels_last'
96          os.environ["KMP_BLOCKTIME"] = str(0)
97          os.environ["KMP_AFFINITY"] = (
98              "granularity=fine,verbose,compact,1,0"
99          )
100         # NOTE: KMP affinity taken care of by passing -cc depth to aprun call
101         OMP_NUM_THREADS = 62
102         config.allow_soft_placement = True
103         config.intra_op_parallelism_threads = OMP_NUM_THREADS
104         config.inter_op_parallelism_threads = 0
105
106     return config, params
107
108
109 def parse_flags(FLAGS, log_file=None):
110     """Parse command line FLAGS and create dictionary of parameters."""
111     try:
112         kv_pairs = FLAGS.__dict__.items()
113     except AttributeError:
114         kv_pairs = FLAGS.items()
115
116     params = {}
117     for key, val in kv_pairs:
118         params[key] = val
119
120     params['log_dir'] = io.create_log_dir(FLAGS, log_file=log_file)
121     params['summaries'] = not FLAGS.no_summaries
122     if 'no_summaries' in params:
123         del params['no_summaries']
124
125     if FLAGS.save_steps is None and FLAGS.train_steps is not None:
126         params['save_steps'] = params['train_steps'] // 4
127
128     if FLAGS.float64:
129         io.log(f'INFO: Setting floating point precision to `float64`.')
130         set_precision('float64')
131
132     return params
133
134
135 def load_params(params_pkl_file=None, log_file=None):
136     if params_pkl_file is None:
137         params_pkl_file = os.path.join(os.getcwd(), 'params.pkl')
```

```
138
139     if os.path.isfile(params_pkl_file):
140         with open(params_pkl_file, 'rb') as f:
141             params = pickle.load(f)
142     else:
143         io.log(f'INFO: Unable to locate: {params_pkl_file}.\n'
144                f'INFO: Using default parameters '
145                f'(PARAMS defined in `config.py`.')
146
147         params = PARAMS.copy()
148         params['log_dir'] = io.create_log_dir(params, log_file=log_file)
149
150     return params
151
152
153 def set_model_weights(model, dest='rand'):
154     """Randomize model weights."""
155     if dest == 'rand':
156         io.log('Randomizing model weights...')
157     elif 'zero' in dest:
158         io.log(f'Zeroing model weights...')
159
160     xnet = model.dynamics.x_fn
161     vnet = model.dynamics.v_fn
162
163     for xblock, vblock in zip(xnet.layers, vnet.layers):
164         for xlayer, vlayer in zip(xblock.layers, vblock.layers):
165             try:
166                 print(f'xlayer.name: {xlayer.name}')
167                 print(f'vlayer.name: {vlayer.name}')
168                 kx, bx = xlayer.get_weights()
169                 kv, bv = vlayer.get_weights()
170                 if dest == 'rand':
171                     kx_new = np.random.randn(*kx.shape)
172                     bx_new = np.random.randn(*bx.shape)
173                     kv_new = np.random.randn(*kv.shape)
174                     bv_new = np.random.randn(*bv.shape)
175                 elif 'zero' in dest:
176                     kx_new = np.zeros(kx.shape)
177                     bx_new = np.zeros(bx.shape)
178                     kv_new = np.zeros(kv.shape)
179                     bv_new = np.zeros(bv.shape)
180
181                 xlayer.set_weights([kx_new, bx_new])
182                 vlayer.set_weights([kv_new, bv_new])
183             except ValueError:
184                 print(f'Unable to set weights for: {xlayer.name}')
185                 print(f'Unable to set weights for: {vlayer.name}')
186
187     return model
188
189
190 def log_plaq_diffs(run_logger, net_weights_arr, avg_plaq_diff_arr):
191     """Log the average values of the plaquette differences.
192
193     NOTE: If inference was performed with either the `--loop_net_weights`
194           or `--loop_transl_weights` flags passed, we want to see how the
195           difference between the observed and expected value of the average
196           plaquette varies with different values of the net weights, so save
197           this data to `.pkl` file and  plot the results.
198     """
199     pd_tup = [
200         (nw, md) for nw, md in zip(net_weights_arr, avg_plaq_diff_arr)
201     ]
202     pd_out_file = os.path.join(run_logger.log_dir, 'plaq_diffs_data.pkl')
203     with open(pd_out_file, 'wb') as f:
204         pickle.dump(pd_tup, f)
205
206
207 def log_mem_usage(run_logger, m_arr):
```

```
    """Log memory usage."""
    m_pkl_file = os.path.join(run_logger.log_dir, 'memory_usage.pkl')
    m_txt_file = os.path.join(run_logger.log_dir, 'memory_usage.txt')
    with open(m_pkl_file, 'wb') as f:
        pickle.dump(m_arr, f)
    with open(m_txt_file, 'w') as f:
        for i in m_arr:
            _ = [f.write(f'{j}\n') for j in i]


def inference_setup(kwargs):
    """Set up relevant (initial) values to use when running inference."""
    if kwargs['loop_net_weights']:  # loop over different values of [S, T, Q]
        #  net_weights_arr = np.zeros((9, 3), dtype=NP_FLOAT)
        w = np.random.randn(3) + 1.
        net_weights_arr = np.array([[0.00, 0.00, 0.00],   # set weights to 0.
                                    # ----------------
                                    [0.00, 0.00, 0.25],   # loop over Q weights
                                    [0.00, 0.00, 0.50],
                                    [0.00, 0.00, 0.75],
                                    [0.00, 0.00, 1.00],
                                    [0.00, 0.00, 2.00],
                                    # ----------------
                                    [0.00, 0.25, 0.00],   # loop over T weights
                                    [0.00, 0.50, 0.00],
                                    [0.00, 0.75, 0.00],
                                    [0.00, 1.00, 0.00],
                                    [0.00, 2.00, 0.00],
                                    # ----------------
                                    [0.25, 0.00, 0.00],   # loop over S weights
                                    [0.50, 0.00, 0.00],
                                    [0.75, 0.00, 0.00],
                                    [1.00, 0.00, 0.00],
                                    [2.00, 0.00, 0.00],
                                    # ----------------
                                    [0.00, 1.00, 1.00],   # [ , T, Q]
                                    [1.00, 0.00, 1.00],   # [S,  , Q]
                                    [1.00, 1.00, 0.00],   # [S, T,  ]
                                    # ----------------
                                    [w[0], w[1], w[2]],   # randomize weights
                                    # ----------------
                                    [1.00, 1.00, 1.00]])  # set weights to 1.

    elif kwargs['loop_transl_weights']:
        net_weights_arr = np.array([[1.0, 0.00, 1.0],
                                    [1.0, 0.10, 1.0],
                                    [1.0, 0.25, 1.0],
                                    [1.0, 0.50, 1.0],
                                    [1.0, 0.75, 1.0],
                                    [1.0, 1.00, 1.0]], dtype=NP_FLOAT)

    else:  # set [S, T, Q] = [1, 1, 1]
        net_weights_arr = np.array([[1, 1, 1]], dtype=NP_FLOAT)

    # if a value has been passed in `kwargs['beta_inference']` use it
    # otherwise, use `model.beta_final`
    beta_final = kwargs.get('beta_final', None)
    beta_inference = kwargs.get('beta_inference', None)
    beta = beta_final if beta_inference is None else beta_inference
    #  betas = [beta_final if beta_inference is None else beta_inference]

    inference_dict = {
        'net_weights_arr': net_weights_arr,
        'beta': beta,
        #  'charge_weight': kwargs.get('charge_weight', 1.),
        'run_steps': kwargs.get('run_steps', 5000),
        'plot_lf': kwargs.get('plot_lf', False),
        'loop_net_weights': kwargs['loop_net_weights'],
        'loop_transl_weights': kwargs['loop_transl_weights']
    }
```

```
278
279         return inference_dict
280
281
282     def run_hmc(FLAGS, log_file=None):
283         """Run inference using generic HMC."""
284         condition1 = not FLAGS.horovod
285         condition2 = FLAGS.horovod and hvd.rank() == 0
286         is_chief = condition1 or condition2
287         if not is_chief:
288             return -1
289
290         FLAGS.hmc = True
291         FLAGS.log_dir = io.create_log_dir(FLAGS, log_file=log_file)
292
293         params = parse_flags(FLAGS)
294         params['hmc'] = True
295         params['use_bn'] = False
296         params['log_dir'] = FLAGS.log_dir
297         params['loop_net_weights'] = False
298         params['loop_transl_weights'] = False
299         params['plot_lf'] = False
300
301         figs_dir = os.path.join(params['log_dir'], 'figures')
302         io.check_else_make_dir(figs_dir)
303
304         io.log(SEP_STR)
305         io.log('HMC PARAMETERS:')
306         for key, val in params.items():
307             io.log(f'  {key}: {val}')
308         io.log(SEP_STR)
309
310         config, params = create_config(params)
311         tf.reset_default_graph()
312
313         model = GaugeModel(params=params)
314
315         sess = tf.Session(config=config)
316         sess.run(tf.global_variables_initializer())
317
318         run_ops = tf.get_collection('run_ops')
319         inputs = tf.get_collection('inputs')
320         run_summaries_dir = os.path.join(model.log_dir, 'summaries', 'run')
321         io.check_else_make_dir(run_summaries_dir)
322         # create summary objects without having to train model
323         _, _ = create_summaries(model, run_summaries_dir, training=False)
324         run_logger = RunLogger(params, inputs, run_ops, save_lf_data=False)
325         plotter = GaugeModelPlotter(params, run_logger.figs_dir)
326
327         inference_dict = inference_setup(params)
328
329         # ---------------------------------------
330         # Create GaugeModelRunner for inference
331         # ---------------------------------------
332         runner = GaugeModelRunner(sess, params, inputs, run_ops, run_logger)
333         run_inference(inference_dict, runner, run_logger, plotter)
334
335
336     def inference(runner, run_logger, plotter, **kwargs):
337         """Perform an inference run, if it hasn't been ran previously.
338
339         Args:
340             runner: GaugeModelRunner object, responsible for performing inference.
341             run_logger: RunLogger object, responsible for running `tf.summary`
342                 operations and accumulating/saving run statistics.
343             plotter: GaugeModelPlotter object responsible for plotting lattice
344                 observables from inference run.
345
346         NOTE: If inference hasn't been run previously with the param values passed,
347             return `avg_plaq_diff`, i.e. the average value of the difference
```

```
        between the expected and observed value of the average plaquette.

    Returns:
        avg_plaq_diff: If run hasn't been completed previously, else None
    """
    run_steps = kwargs.get('run_steps', 5000)
    beta = kwargs.get('beta', 5.)
    kwargs['eps'] = runner.eps

    run_str = run_logger._get_run_str(**kwargs)
    kwargs['run_str'] = run_str

    if not run_logger.existing_run(run_str):
        run_logger.reset(**kwargs)
        t0 = time.time()
        runner.run(**kwargs)
        io.log(SEP_STR)
        run_time = time.time() - t0
        io.log(SEP_STR + f'\nTook: {run_time}s to complete run.\n' + SEP_STR)
        avg_plaq_diff = plotter.plot_observables(run_logger.run_data, **kwargs)

        if kwargs.get('plot_lf', False):
            lf_plotter = LeapfrogPlotter(plotter.out_dir, run_logger)
            num_samples = runner.params.get('num_samples', 20)
            lf_plotter.make_plots(run_logger.run_dir,
                                  num_samples=num_samples)

        return avg_plaq_diff

    else:
        nw = kwargs.get('net_weights', [1., 1., 1.])
        io.log(SEP_STR)
        io.log('\n Inference has already been completed for:\n'
               f'\t net_weights: [{nw[0]}, {nw[1]}, {nw[2]}]\n'
               f'\t run_steps: {run_steps}\n'
               f'\t eps: {runner.eps}\n'
               f'\t beta: {beta}\n'
               f' Continuing...\n')
        io.log(SEP_STR)


def run_inference(runner, run_logger=None, plotter=None, **kwargs):
    """Run inference.

    Args:
        inference_dict: Dictionary containing parameters to use for inference.
        runner: GaugeModelRunner object that actually performs the inference.
        run_logger: RunLogger object that logs observables and other data
            generated during inference.
        plotter: GaugeModelPlotter object responsible for plotting observables
            generated during inference.
    """
    if plotter is None or run_logger is None:
        return

    dir_append = kwargs.get('dir_append', None)
    net_weights_arr = kwargs.get('net_weights_arr', [1., 1., 1.])
    avg_plaq_diff_arr = []
    m_arr = []
    for net_weights in net_weights_arr:
        if HAS_MEMORY_PROFILER:
            usage0 = memory_profiler.memory_usage()
            m_arr.append(usage0)
        run_logger.clear()
        if HAS_MEMORY_PROFILER:
            usage1 = memory_profiler.memory_usage()
            m_arr.append(usage1)
        kwargs.update({
            'net_weights': net_weights,
            'dir_append': dir_append
```

```
        })
        inference(runner, run_logger, plotter, **kwargs)

    log_mem_usage(run_logger, m_arr)
    if kwargs['loop_net_weights']:
        log_plaq_diffs(run_logger, net_weights_arr, avg_plaq_diff_arr)


def main_inference(inference_kwargs):
    """Perform inference using saved model."""
    params_file = inference_kwargs.get('params_file', None)
    params = load_params(params_file)  # load params used during training

    # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    # NOTE: We want to restrict all communication (file I/O) to only be
    # performed on rank 0 (i.e. `is_chief`) so there are two cases:
    #    1. We're using Horovod, so we have to check hvd.rank() explicitly.
    #    2. We're not using Horovod, in which case `is_chief` is always True.
    # ------------------------------------------------------------------------
    condition1 = not params['using_hvd']
    condition2 = params['using_hvd'] and hvd.rank() == 0
    is_chief = condition1 or condition2
    if not is_chief:
        return
    # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    checkpoint_dir = os.path.join(params['log_dir'], 'checkpoints/')
    assert os.path.isdir(checkpoint_dir)
    checkpoint_file = tf.train.latest_checkpoint(checkpoint_dir)

    config, params = create_config(params)
    sess = tf.Session(config=config)
    saver = tf.train.import_meta_graph(f'{checkpoint_file}.meta')
    saver.restore(sess, checkpoint_file)

    run_ops = tf.get_collection('run_ops')
    inputs = tf.get_collection('inputs')

    run_logger = RunLogger(params, inputs, run_ops, save_lf_data=False)
    plotter = GaugeModelPlotter(params, run_logger.figs_dir)

    # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    #  Set up relevant values to use for inference (parsed from kwargs)
    #
    #  NOTE: We are only interested in the command line arguments that
    #        were passed to `inference.py` (i.e. those contained in kwargs)
    inference_kwargs['loop_net_weights'] = True
    inference_dict = inference_setup(inference_kwargs)
    if inference_dict['beta'] is None:
        inference_dict['beta'] = params['beta_final']
    # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    net_weights_file = os.path.join(params['log_dir'], 'net_weights.txt')
    np.savetxt(net_weights_file, inference_dict['net_weights_arr'],
               delimiter=', ', newline='\n', fmt='%-.4g')

    # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    # Create GaugeModelRunner for inference
    # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    runner = GaugeModelRunner(sess, params, inputs, run_ops, run_logger)
    run_inference(runner, run_logger, plotter, **inference_dict)

    ###########################################################################
    #  NOTE: THE FOLLOWING WONT WORK WHEN RESTORING FROM CHECKPOINT (FOR
    #        INFERENCE) UNLESS `GaugeModel` IS ENTIRELY REBUILT:
    # -------------------------------------------------------------------------
    # set 'net_weights_arr' = [1., 1., 1.] so each Q, S, T contribute
    #  inference_dict['net_weights_arr'] = np.array([[1, 1, 1]],
    #                                               dtype=NP_FLOAT)
    # set 'betas' to be a single value
```

```
    #  inference_dict['betas'] = inference_dict['betas'][-1]
    #
    #
    #  # randomize the model weights and run inference using these weights
    #  runner.model = set_model_weights(runner.model, dest='rand')
    #  run_inference(inference_dict,
    #                runner, run_logger,
    #                plotter, dir_append='_rand')
    #
    #  #  zero the model weights and run inference using these weights
    #  runner.model = set_model_weights(runner.model, dest='zero')
    #  run_inference(inference_dict,
    #                runner, run_logger,
    #                plotter, dir_append='_zero')
    ##########################################################################


if __name__ == '__main__':
    FLAGS = parse_args()

    t0 = time.time()
    log_file = 'output_dirs.txt'
    kwargs = FLAGS.__dict__

    main_inference(kwargs)

    io.log('\n\n' + SEP_STR)
    io.log(f'Time to complete: {time.time() - t0:.4g}')
```

A.5 l2hmc-qcd/models/model.py

```
"""
gauge_model.py

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

------------------------------------------------------------
TODO (taken from [1.]):

update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
! no update ops in the default graph
io.log("update_ops: ", update_ops)
Use the update ops of the model itself
io.log("model.updates: ", self.dynamics.updates)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
PREVIOUS (WORKING but diverging grads as of 07/15/2019):
```
with tf.control_dependencies(self.dynamics.updates):
    self.train_op = self.optimizer.apply_gradients(
        grads_and_vars,
        global_step=self.global_step,
        name='train_op'
)
```
    self.train_op = tf.group(minimize_op,
                             self.dynamics.updates)
    try:
        self.train_op = self._append_update_ops(train_op)
    update_ops = [self.dynamics.updates,
                  tf.get_collection(tf.GraphKeys.UPDATE_OPS)]
    tf.add_to_collection(tf.GraphKeys.UPDATE_OPS,
                         *[i for i in self.dynamics.updates])
------------------------------------------------------------

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

import os
import functools

import numpy as np
import tensorflow as tf
import utils.file_io as io

from lattice.lattice import GaugeLattice
from utils.horovod_utils import warmup_lr
from dynamics.dynamics import GaugeDynamics
from config import GLOBAL_SEED, TF_FLOAT, TF_INT, PARAMS, HAS_HOROVOD
from tensorflow.python.ops import control_flow_ops as control_flow_ops

if HAS_HOROVOD:
    import horovod.tensorflow as hvd


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 = {}
        if params is None:
            params = PARAMS  # default parameters, defined in `variables.py`

        self.params = params
        for key, val in self.params.items():
            if 'weight' in key and key != 'charge_weight':
                self.loss_weights[key] = val
            elif key == 'charge_weight':
                pass
            else:
                setattr(self, key, val)

        self.eps_trainable = not self.eps_fixed
        self.charge_weight_np = params['charge_weight']

        self.lattice = self._create_lattice()
        self.batch_size = self.lattice.samples.shape[0]
        self.x_dim = self.lattice.num_links

        # build input placeholders for tensors
        inputs = self._create_inputs()
        _ = [setattr(self, key, val) for key, val in inputs.items()]

        self.dynamics = self._create_dynamics()

        # metric function used when calculating the loss
        self.metric_fn = self._create_metric_fn(self.metric)

        obs_ops = self._create_observables()
        _ = [setattr(self, key, val) for key, val in obs_ops.items()]

        self._build_sampler()
        run_ops = self._build_run_ops()

        if self.hmc:
            train_ops = {}
        else:
            self._create_lr()
            self._create_optimizer()
            self._apply_grads()
            train_ops = self._build_train_ops()

        self.ops_dict = {
            'run_ops': run_ops,
            'train_ops': train_ops
        }

        for key, val in self.ops_dict.items():
            _ = [tf.add_to_collection(key, op) for op in list(val.values())]

    def _create_lattice(self):
        """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)

        return lattice

    def _create_dynamics(self, **kwargs):
        """Initialize dynamics object."""
        with tf.name_scope('dynamics'):
```

```
            # default values of keyword arguments
            keys = ['eps', 'hmc', 'network_arch',
                    'num_steps', 'use_bn', 'dropout_prob',
                    'num_hidden1', 'num_hidden2', 'zero_translation']
            dynamics_kwargs = {k: getattr(self, k) for k in keys}
            dynamics_kwargs['eps_trainable'] = not self.eps_fixed
            #  dynamics_kwargs = {
            #      'eps': self.eps,
            #      'hmc': self.hmc,
            #      'network_arch': self.network_arch,
            #      'num_steps': self.num_steps,
            #      'eps_trainable': not self.eps_fixed,
            #     #  'data_format': self.data_format,
            #      'use_bn': self.use_bn,
            #      'dropout_prob': self.dropout_prob,
            #      'num_hidden1': self.num_hidden1,
            #      'num_hidden2': self.num_hidden2,
            #      'zero_translation': self.zero_translation
            #  }
            dynamics_kwargs.update(kwargs)
            samples = self.lattice.samples_tensor
            potential_fn = self.lattice.get_potential_fn(samples)
            dynamics = GaugeDynamics(lattice=self.lattice,
                                     potential_fn=potential_fn,
                                     **dynamics_kwargs)

        return dynamics

    def _create_observables(self):
        """Create operations for calculating lattice observables."""
        with tf.name_scope('observables'):
            plaq_sums = self.lattice.calc_plaq_sums(self.x)
            actions = self.lattice.calc_actions(plaq_sums=plaq_sums)
            plaqs = self.lattice.calc_plaqs(plaq_sums=plaq_sums)
            avg_plaqs = tf.reduce_mean(plaqs, name='avg_plaqs')
            charges = self.lattice.calc_top_charges(plaq_sums=plaq_sums)

        obs_ops = {
            'plaq_sums_op': plaq_sums,
            'actions_op': actions,
            'plaqs_op': plaqs,
            'avg_plaqs_op': avg_plaqs,
            'charges_op': charges,
        }

        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.
                train_phase: Boolean placeholder indicating if the model is
                    curerntly being trained.
        """
        with tf.name_scope('inputs'):
```

```
            if not tf.executing_eagerly():
                x = tf.placeholder(dtype=TF_FLOAT,
                                   shape=(self.batch_size, self.x_dim),
                                   name='x')

                beta = tf.placeholder(dtype=TF_FLOAT,
                                      shape=(),
                                      name='beta')

                charge_weight = tf.placeholder(dtype=TF_FLOAT,
                                               shape=(),
                                               name='charge_weight')

                scale_weight = tf.placeholder(dtype=TF_FLOAT,
                                              shape=(),
                                              name='scale_weight')
                transf_weight = tf.placeholder(dtype=TF_FLOAT,
                                               shape=(),
                                               name='transformation_weight')
                transl_weight = tf.placeholder(dtype=TF_FLOAT,
                                               shape=(),
                                               name='translation_weight')

                net_weights = [scale_weight, transf_weight, transl_weight]

                train_phase = tf.placeholder(tf.bool, name='is_training')

            else:
                x = tf.convert_to_tensor(
                    self.lattice.samples.reshape((self.batch_size,
                                                  self.x_dim))
                )
                beta = tf.convert_to_tensor(self.beta_init)
                charge_weight = tf.convert_to_tensor(0.)
                net_weights = tf.convert_to_tensor([1., 1., 1.])
                train_phase = True

        names = ['x', 'beta', 'charge_weight', 'train_phase', 'net_weights']
        tensors = [x, beta, charge_weight, train_phase, net_weights]
        outputs = dict(zip(names, tensors))

        for name, tensor in outputs.items():
            if name != 'net_weights':
                tf.add_to_collection('inputs', tensor)
            else:
                _ = [tf.add_to_collection('inputs', t) for t in tensor]

        return outputs

    @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 == 'sin_diff':
                def metric_fn(x1, x2):
                    return 1. - tf.sin(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_lr(self, lr_init=None):
        """Create learning rate."""
        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()

        with tf.name_scope('learning_rate'):
            # HOROVOD: When performing distributed training, it can be usedful
            # to "warmup" the learning rate gradually, done using the
            # `configure_learning_rate` method below..
            if self.warmup_lr:
                kwargs = {
                    'target_lr': lr_init,
                    'warmup_steps': int(0.1 * self.train_steps),
                    'global_step': self.global_step,
                    'decay_steps': self.lr_decay_steps,
                    'decay_rate': self.lr_decay_rate,
                }
                self.lr = warmup_lr(**kwargs)
            else:
                self.lr = tf.train.exponential_decay(lr_init,
                                                     self.global_step,
                                                     self.lr_decay_steps,
                                                     self.lr_decay_rate,
                                                     staircase=False,
                                                     name='learning_rate')

    def _create_optimizer(self):
        """Create learning rate and optimizer."""
        if not hasattr(self, 'lr'):
            self._create_lr()

        #  with tf.control_dependencies(update_ops):
        with tf.name_scope('optimizer'):
            #  update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
            self.optimizer = tf.train.AdamOptimizer(self.lr)
            if self.using_hvd:
                self.optimizer = hvd.DistributedOptimizer(self.optimizer)

    def _build_run_ops(self):
        """Build run_ops dict containing grouped operations for inference."""
        keys = ['x_out', 'px', 'actions_op',
                'plaqs_op', 'avg_plaqs_op',
                'charges_op', 'charge_diffs_op']
        run_ops = {k: getattr(self, k) for k in keys}

        if self.save_lf:
```

```
            keys = ['lf_out', 'pxs_out', 'masks',
                    'logdets', 'sumlogdet', 'fns_out']

            fkeys = [k + '_f' for k in keys]
            bkeys = [k + '_b' for k in keys]

            run_ops.update({k: getattr(self, k) for k in fkeys})
            run_ops.update({k: getattr(self, k) for k in bkeys})

        run_ops['dynamics_eps'] = self.dynamics.eps

        return run_ops

    def _build_train_ops(self):
        """Build train_ops dict containing grouped operations for training."""
        if self.hmc:
            train_ops = {}

        else:
            keys = ['train_op', 'loss_op', 'x_out',
                    'px', 'actions_op', 'plaqs_op',
                    'charges_op', 'charge_diffs_op', 'lr']
            train_ops = {k: getattr(self, k) for k in keys}

            train_ops['dynamics.eps'] = self.dynamics.eps

        return train_ops

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

        NOTE: In contrast to the original paper where the L2 difference was
        used, we are now using 1 - cos(x1 - x2).

        Args:
            x_tup: Tuple of (x, x_proposed) configurations.
            z_tup: Tuple of (z, z_proposed) configurations.
            p_tup: Tuple of (x, z) acceptance probabilities.
            weights: dictionary of weights giving relative weight of each term
                in the loss function.

        Returns:
            std_loss
        """
        eps = 1e-4
        aux_weight = weights.get('aux_weight', 1.)
        std_weight = weights.get('std_weight', 1.)

        if std_weight == 0.:  # don't bother calculating loss if weight is 0
            return 0.

        x_init = inputs['x_init']
        x_proposed = inputs['x_proposed']
        z_init = inputs['z_init']
        z_proposed = inputs['z_proposed']
        px = inputs['px']
        pz = inputs['pz']

        ls = self.loss_scale
        with tf.name_scope('std_loss'):
            with tf.name_scope('x_loss'):
                x_std_loss = tf.reduce_sum(
                    self.metric_fn(x_init, x_proposed), axis=1
                )
                x_std_loss *= px
                #  x_std_loss_avg = tf.reduce_sum(x_std_loss, axis=0,
                #                                 name='x_std_loss_avg')

                # Add eps for numerical stability
                x_std_loss = tf.add(x_std_loss, eps, name='x_std_loss')
```

```
                #  tf.add_to_collection('losses', x_std_loss)

            with tf.name_scope('z_loss'):
                z_std_loss = tf.reduce_sum(
                    self.metric_fn(z_init, z_proposed), axis=1
                )
                z_std_loss *= pz * aux_weight
                #  z_std_loss_avg = tf.reduce_mean(z_std_loss, axis=0,
                #                                  name='z_std_loss_avg')
                z_std_loss = tf.add(z_std_loss, eps, name='z_std_loss')
                #  tf.add_to_collection('losses', z_std_loss)

            with tf.name_scope('tot_loss'):
                loss = (ls * (1. / x_std_loss + 1. / z_std_loss)
                        - (x_std_loss + z_std_loss) / ls) * std_weight

                std_loss = tf.reduce_mean(loss, axis=0, name='std_loss')
                #  tf.add_to_collection('losses', std_loss)

        return std_loss  # , x_std_loss_avg, z_std_loss_avg

    def _calc_charge_loss(self, inputs, **weights):
        """Calculate contribution to total loss from charge difference.

        NOTE: This is an additional term introduced to the loss function that
        measures the difference in the topological charge between the initial
        configuration and the proposed configuration.

        Args:
            x_tup: Tuple of (x, x_proposed) configurations.
            z_tup: Tuple of (z, z_proposed) configurations.
            p_tup: Tuple of (x, z) acceptance probabilities.
            weights: dictionary of weights giving relative weight of each term
                in the loss function.

        Returns:
            std_loss
        """
        aw = weights.get('aux_weight', 1.)
        qw = weights.get('charge_weight', 1.)

        if qw == 0:
            return 0.

        x_init = inputs['x_init']
        x_proposed = inputs['x_proposed']
        z_init = inputs['z_init']
        z_proposed = inputs['z_proposed']
        px = inputs['px']
        pz = inputs['pz']

        # 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 topological
        # charge
        with tf.name_scope('charge_loss'):
            with tf.name_scope('x_loss'):
                x_dq = self.lattice.calc_top_charges_diff(x_init, x_proposed)
                xq_loss = px * x_dq
                #  xq_loss_avg = tf.reduce_mean(xq_loss, axis=0,
                #                               name='xq_loss_avg')

            with tf.name_scope('z_loss'):
                if aw > 0:
                    z_dq = self.lattice.calc_top_charges_diff(z_init,
                                                              z_proposed)
                else:
                    z_dq = tf.zeros_like(x_dq)

                zq_loss = aw * pz * z_dq
```

```
                #  zq_loss_avg = tf.reduce_mean(zq_loss, axis=0,
                #                               name='zq_loss_avg')


            with tf.name_scope('tot_loss'):
                # Each of the loss terms is scaled by the `loss_scale` which
                # introduces a universal multiplicative factor that scales the
                # value of the loss
                charge_loss = - qw * (xq_loss + zq_loss) / self.loss_scale
                charge_loss = tf.reduce_mean(charge_loss, axis=0,
                                             name='charge_loss')
                #  tf.add_to_collection('losses', charge_loss)

        return charge_loss  # , xq_loss_avg, zq_loss_avg

    def calc_loss(self, x, beta, net_weights, train_phase, **weights):
        """Create operation for calculating the loss.

        Args:
            x: Input tensor of shape (self.num_samples,
                self.lattice.num_links) containing batch of GaugeLattice links
                variables.
            beta (float): Inverse coupling strength.

        Returns:
            loss (float): Operation responsible for calculating the total loss.
            px (np.ndarray): Array of acceptance probabilities from
                Metropolis-Hastings accept/reject step. Has shape:
                (self.num_samples,)
            x_out: Output samples obtained after Metropolis-Hastings
                accept/reject step.

        NOTE: If proposed configuration is accepted following
            Metropolis-Hastings accept/reject step, x_proposed and x_out are
            equivalent.
        """
        with tf.name_scope('x_update'):
            x_dynamics_output = self.dynamics.apply_transition(x, beta,
                                                               net_weights,
                                                               train_phase,
                                                               self.save_lf)
            x_proposed = x_dynamics_output['x_proposed']
            px = x_dynamics_output['accept_prob']
            x_out = x_dynamics_output['x_out']

        # Auxiliary variable
        with tf.name_scope('z_update'):
            z = tf.random_normal(tf.shape(x),
                                 dtype=TF_FLOAT,
                                 seed=GLOBAL_SEED,
                                 name='z')
            z_dynamics_output = self.dynamics.apply_transition(z, beta,
                                                               net_weights,
                                                               train_phase,
                                                               save_lf=False)
            z_proposed = z_dynamics_output['x_proposed']
            pz = z_dynamics_output['accept_prob']

        with tf.name_scope('top_charge_diff'):
            x_dq = tf.cast(
                self.lattice.calc_top_charges_diff(x, x_out),
                dtype=TF_INT
            )

        # NOTE:
        #  std:_loss: 'standard' loss
        #  charge_loss: Contribution from the difference in topological charge
        #    betweeen the initial and proposed configurations  to the total
        #     loss.
        inputs = {
```

```
            'x_init': x,
            'x_proposed': x_proposed,
            'z_init': z,
            'z_proposed': z_proposed,
            'px': px,
            'pz': pz
        }

        with tf.name_scope('calc_loss'):
            std_loss = self._calc_std_loss(inputs, **weights)
            charge_loss = self._calc_charge_loss(inputs, **weights)
            total_loss = tf.add(std_loss, charge_loss, name='total_loss')

        #  losses = [*std_losses, *charge_losses]
        #  _ = [tf.add_to_collection('losses', i) for i in losses]

        return total_loss, x_dq, x_dynamics_output

    def calc_loss_and_grads(self,
                            x,
                            beta,
                            net_weights,
                            train_phase,
                            **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.
        """
        if tf.executing_eagerly():
            with tf.name_scope('grads'):
                with tf.GradientTape() as tape:
                    loss, x_dq, dynamics_output = self.calc_loss(x,
                                                                 beta,
                                                                 net_weights,
                                                                 train_phase,
                                                                 **weights)

                grads = tape.gradient(loss, self.dynamics.trainable_variables)
                if self.clip_value > 0.:
                    grads, _ = tf.clip_by_global_norm(grads, self.clip_value)
        else:
            loss, x_dq, dynamics_output = self.calc_loss(x, beta,
                                                         net_weights,
                                                         train_phase,
                                                         **weights)
            with tf.name_scope('grads'):
                grads = tf.gradients(loss, self.dynamics.trainable_variables)
                if self.clip_value > 0.:
                    grads, _ = tf.clip_by_global_norm(grads, self.clip_value)

        return loss, grads, x_dq, dynamics_output

    def _build_sampler(self):
        """Build TensorFlow graph."""
        with tf.name_scope('sampler'):
```

```
            #  self.loss_op, self.grads, self.x_out, self.px, x_dq = output
            loss, grads, x_dq, dynamics_output = self.calc_loss_and_grads(
                x=self.x,
                beta=self.beta,
                net_weights=self.net_weights,
                train_phase=self.train_phase,
                **self.loss_weights
            )
            self.loss_op = loss
            self.grads = grads
            self.x_out = dynamics_output['x_out']
            self.px = dynamics_output['accept_prob']
            self.charge_diffs_op = tf.reduce_sum(x_dq) / self.num_samples

            if self.save_lf:
                op_keys = ['masks_f', 'masks_b',
                           'lf_out_f', 'lf_out_b',
                           'pxs_out_f', 'pxs_out_b',
                           'logdets_f', 'logdets_b',
                           'fns_out_f', 'fns_out_b',
                           'sumlogdet_f', 'sumlogdet_b']
                for key in op_keys:
                    try:
                        op = dynamics_output[key]
                        setattr(self, key, op)
                    except KeyError:
                        continue

                with tf.name_scope('l2hmc_fns'):
                    self.l2hmc_fns = {
                        'out_fns_f': self.extract_l2hmc_fns(self.fns_out_f),
                        'out_fns_b': self.extract_l2hmc_fns(self.fns_out_b),
                    }

    def extract_l2hmc_fns(self, fns):
        """Method for extracting each of the Q, S, T functions as tensors."""
        if not self.save_lf:
            return

        # fns has shape: (num_steps, 4, 3, num_samples, lattice.num_links)
        fnsT = tf.transpose(fns, perm=[2, 1, 0, 3, 4], name='fns_transposed')

        out_fns = {}
        names = ['scale', 'translation', 'transformation']
        subnames = ['v1', 'x1', 'x2', 'v2']
        for idx, name in enumerate(names):
            out_fns[name] = {}
            for subidx, subname in enumerate(subnames):
                out_fns[name][subname] = fnsT[idx][subidx]

        return out_fns

    def _apply_grads(self):
        """Build Tensorflow graph."""
        # ------------------------------
        # Ref. [1.] in TODO (line 8)
        # ------------------------------
        with tf.name_scope('train'):
            grads_and_vars = zip(self.grads, self.dynamics.trainable_variables)
            with tf.control_dependencies(self.dynamics.updates):
                self.train_op = self.optimizer.apply_gradients(
                    grads_and_vars,
                    global_step=self.global_step,
                    name='train_op'
                )
```

A.6 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.


TODO:
    - Log separately the Q, S, T values for the 'x' and 'v' functions.
    - Look at raw phase values both before and after mod operation.
    - Try running generic net using 64-point precision.
    - See if 64-point precision issues with Conv3D are fixed in tf 1.13-1.14.
    - JLSE account.


Author: Sam Foreman (github: @saforem2)
Date: 1/14/2019
"""
from __future__ import absolute_import
from __future__ import print_function
from __future__ import division

import numpy as np
import numpy.random as npr
import tensorflow as tf

from config import GLOBAL_SEED, TF_FLOAT, TF_INT
from network.network import FullNet


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))


def hmc_network(inputs, train_phase):
    return [tf.zeros_like(inputs[0]) for _ in range(3)]


def _add_to_collection(collection, ops):
    if len(ops) > 1:
        _ = [tf.add_to_collection(collection, op) for op in ops]
    else:
        tf.add_to_collection(collection, ops)


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
    self.l2hmc_fns = {}

    # 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)

    _eps_np = kwargs.get('eps', 0.4)
    #  with tf.variable_scope(reus):
    #  self.eps = tf.get_variable('eps', dtype=TF_FLOAT,
    #                             trainable=self.eps_trainable,
    #                             initializer=tf.constant(_eps_np))

    self.log_eps = tf.Variable(
        initial_value=tf.log(tf.constant(_eps_np)),
        name='log_eps',
        dtype=TF_FLOAT,
        trainable=self.eps_trainable
    )

    self.eps = tf.exp(self.log_eps, 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, train_phase: [
            tf.zeros_like(inp[0]) for t in range(3)
        ]
        self.v_fn = lambda inp, train_phase: [
            tf.zeros_like(inp[0]) for t in range(3)
        ]

    else:
        num_filters = int(self.lattice.space_size)
        net_kwargs = {
            'network_arch': self.network_arch,
            'use_bn': self.use_bn,  # whether or not to use batch norm
            'dropout_prob': self.dropout_prob,
            'x_dim': self.lattice.num_links,  # dim of target space
            'links_shape': self.lattice.links.shape,
            'num_hidden1': self.num_hidden1,
```

```
                'num_hidden2': self.num_hidden2,
                'generic_activation': tf.nn.relu,
                'num_filters': [num_filters, int(2 * num_filters)],
                'name_scope': 'x',  # namespace in which to create network
                'factor': 2.,  # scale factor used in original paper
                '_input_shape': (self.batch_size, *self.lattice.links.shape),
                'zero_translation': self.zero_translation,
                #  'data_format': self.data_format,
            }

            if self.network_arch == 'conv3D':
                net_kwargs.update({
                    'filter_sizes': [(3, 3, 1), (2, 2, 1)],
                })
            elif self.network_arch == 'conv2D':
                net_kwargs.update({
                    'filter_sizes': [(3, 3), (2, 2)],
                })

            self.build_network(net_kwargs)

    def build_network(self, net_kwargs):
        """Build neural network used to train model."""
        with tf.name_scope("DynamicsNetwork"):
            self.x_fn = FullNet(model_name='XNet', **net_kwargs)

            net_kwargs['name_scope'] = 'v'  # update name scope
            net_kwargs['factor'] = 1.       # factor used in orig. paper
            self.v_fn = FullNet(model_name='VNet', **net_kwargs)

    def call(self, *args, **kwargs):
        """Call method."""
        return self.apply_transition(*args, **kwargs)

    def apply_transition(self,
                         x_in,
                         beta,
                         net_weights,
                         train_phase,
                         save_lf=False):
        """Propose a new state and perform the accept/reject step.

        We simulate the (molecular) dynamics update both forward and backward,
        and use sampled masks to compute the actual solutions.

        Args:
            x_in (placeholder): Batch of (x) samples (GaugeLattice.samples).
            beta (float): Inverse coupling constant.
            net_weights: Array of scaling weights to multiply each of the
                output functions (scale, translation, transformation).
            train_phase: Boolean tf.placeholder used to indicate if currently
                training model or running inference on trained model.
            save_lf: Flag specifying whether or not to save output leapfrog
                configs.

        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.
        """
        lf_dict = {}

        with tf.name_scope('transition_forward'):
            outputs_f = self.transition_kernel(x_in, beta,
                                               net_weights,
                                               train_phase,
                                               forward=True,
                                               save_lf=save_lf)
            xf = outputs_f['x_proposed']
```

```
            vf = outputs_f['v_proposed']
            lf_dict['pxs_out_f'] = outputs_f['accept_prob']

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

        def get_lf_keys(direction):
            base_keys = ['lf_out', 'logdets', 'sumlogdet', 'fns_out']
            new_keys = [k + f'_{direction}' for k in base_keys]
            return list(zip(new_keys, base_keys))

        keys_f = get_lf_keys('f')
        keys_b = get_lf_keys('b')

        if save_lf:
            lf_dict.update({k[0]: outputs_f[k[1]] for k in keys_f})
            lf_dict.update({k[0]: outputs_b[k[1]] for k in keys_b})

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

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

        with tf.name_scope('v_proposed'):
            v_proposed = (lf_dict['masks_f'][:, None] * vf
                          + lf_dict['masks_b'][:, None] * vb)

        # Probability of accepting the proposed states
        with tf.name_scope('accept_prob'):
            accept_prob = (lf_dict['masks_f'] * lf_dict['pxs_out_f']
                           + lf_dict['masks_b'] * lf_dict['pxs_out_b'])

        # Accept or reject step
        with tf.name_scope('accept_mask'):
            accept_mask = tf.cast(
                accept_prob > tf.random_uniform(tf.shape(accept_prob),
                                                dtype=TF_FLOAT,
                                                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.update(lf_dict)

        return outputs

    def transition_kernel(self,
                          x_in,
                          beta,
                          net_weights,
                          train_phase,
                          forward=True,
                          save_lf=False):
        """Transition kernel of augmented leapfrog integrator."""
        lf_fn = self._forward_lf if forward else self._backward_lf

        with tf.name_scope('refresh_momentum'):
            v_in = tf.random_normal(tf.shape(x_in),
                                    dtype=TF_FLOAT,
                                    seed=GLOBAL_SEED,
                                    name='refresh_momentum')

        with tf.name_scope('init'):
            x_proposed, v_proposed = x_in, v_in

            step = tf.constant(0., name='md_step', dtype=TF_FLOAT)
            batch_size = tf.shape(x_in)[0]
            logdet = tf.zeros((batch_size,), dtype=TF_FLOAT)
            #  fns0 = tf.zeros((4, 3, *x_in.shape),)
            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+1,
                                         dynamic_size=True,
                                         name='logdets_out',
                                         clear_after_read=False)
            fns_out = tf.TensorArray(dtype=TF_FLOAT,
                                     size=self.num_steps,
                                     dynamic_size=True,
                                     name='l2hmc_fns',
                                     clear_after_read=False)

            lf_out = lf_out.write(0, x_in)
            logdets_out = logdets_out.write(0, logdet)
            #  fns_out.write(0, fns0)

        def body(step, x, v, logdet, lf_samples, logdets, fns):
            # cast leapfrog step to integer
            i = tf.cast(step, dtype=tf.int32)
            new_x, new_v, j, _fns = lf_fn(x, v, beta, step,
                                          net_weights, train_phase)
            lf_samples = lf_samples.write(i+1, new_x)
            logdets = logdets.write(i+1, logdet+j)
            fns = fns.write(i, _fns)
            #  fns = fns.write(i, _fns)

            return (step+1, new_x, new_v, logdet+j, lf_samples, logdets, fns)

        def cond(step, *args):
            return tf.less(step, self.num_steps)

        outputs = tf.while_loop(
            cond=cond,
```

```
            body=body,
            loop_vars=[step, x_proposed, v_proposed,
                       logdet, lf_out, logdets_out, fns_out])

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

        accept_prob = self._compute_accept_prob(
            x_in,
            v_in,
            x_proposed,
            v_proposed,
            sumlogdet,
            beta
        )

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

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

        return outputs

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

            sumlogdet = 0.

            v, logdet, vf_fns = self._update_v_forward(x, v, beta, t,
                                                       net_weights,
                                                       train_phase)
            sumlogdet += logdet
            forward_fns.append(vf_fns)

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

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

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

```
            forward_fns.append(vf_fns)

        return x, v, sumlogdet, forward_fns

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

            sumlogdet = 0.

            v, logdet, vb_fns = self._update_v_backward(x, v, beta, t,
                                                        net_weights,
                                                        train_phase)
            sumlogdet += logdet
            backward_fns.append(vb_fns)

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

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

            v, logdet, vb_fns = self._update_v_backward(x, v, beta, t,
                                                        net_weights,
                                                        train_phase)
            sumlogdet += logdet
            backward_fns.append(vb_fns)

        return x, v, sumlogdet, backward_fns

    def _update_v_forward(self, x, v, beta, t, net_weights, train_phase):
        """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_vf'):
            grad = self.grad_potential(x, beta)

            scale, transl, transf = self.v_fn((x, grad, t), train_phase)

            with tf.name_scope('vf_mul'):
                scale *= 0.5 * self.eps * net_weights[0]
                transl *= net_weights[1]
```

```
                transf *= self.eps * net_weights[2]
                fns = [scale, transl, transf]

            with tf.name_scope('vf_exp'):
                scale_exp = tf.cast(exp(scale, 'scale_exp'), dtype=TF_FLOAT)
                transf_exp = tf.cast(exp(transf, 'transf_exp'),
                                     dtype=TF_FLOAT)

            with tf.name_scope('proposed'):
                v = (v * scale_exp
                     - 0.5 * self.eps * (grad * transf_exp - transl))

            logdet = tf.reduce_sum(scale, axis=1, name='logdet_vf')

        return v, logdet, fns

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

            with tf.name_scope('xf_mul'):
                scale *= self.eps * net_weights[0]
                transl *= net_weights[1]
                transf *= self.eps * net_weights[2]
                fns = [scale, transl, transf]

            with tf.name_scope('xf_exp'):
                scale_exp = exp(scale, 'scale_exp')
                transf_exp = exp(transf, 'transformation_exp')

            with tf.name_scope('proposed'):
                x = (mask * x + mask_inv
                     * (x * scale_exp + self.eps * (v * transf_exp + transl)))

            logdet = tf.reduce_sum(mask_inv * scale, axis=1, name='logdet_xf')

        return x, logdet, fns

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

            scale, transl, transf = self.v_fn([x, grad, t], train_phase)

            with tf.name_scope('vb_mul'):
                scale *= -0.5 * self.eps * net_weights[0]
                transl *= net_weights[1]
                transf *= self.eps * net_weights[2]
                fns = [scale, transl, transf]

            with tf.name_scope('vb_exp'):
                scale_exp = exp(scale, 'scale_exp')
                transf_exp = exp(transf, 'transformation_exp')

            with tf.name_scope('proposed'):
                v = scale_exp * (v + 0.5 * self.eps
                                 * (grad * transf_exp - transl))

            logdet = tf.reduce_sum(scale, axis=1, name='logdet_vb')

        return v, logdet, fns

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

```
            with tf.name_scope('xb_mul'):
                scale *= -self.eps * net_weights[0]
                transl *= net_weights[1]
                transf *= self.eps * net_weights[2]
                fns = [scale, transl, transf]

            with tf.name_scope('xb_exp'):
                scale_exp = exp(scale, 'xb_scale')
                transf_exp = exp(transf, 'xb_transformation')

            with tf.name_scope('proposed'):
                x = (mask * x + mask_inv * scale_exp
                     * (x - self.eps * (v * transf_exp + transl)))

            logdet = tf.reduce_sum(mask_inv * scale, axis=1,
                                   name='logdet_xb')

        return x, logdet, fns

    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
            accept_prob = tf.where(tf.is_finite(prob), prob,
                                   tf.zeros_like(prob))

        return accept_prob

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

            t = tf.tile(tf.expand_dims(trig_t, 0), (tile, 1))

        return t

    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):
        with tf.name_scope('get_mask'):
            m = tf.gather(self.masks, tf.cast(step, dtype=TF_INT))
            _m = 1. - m  # complementary mask
        return m, _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))

        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'):
            potential = self.potential_energy(x, beta)
            kinetic = self.kinetic_energy(v)
            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():
                tfe = tf.contrib.eager
                grad_fn = tfe.gradients_function(self.potential_energy,
                                                 params=["x"])
                grad = grad_fn(x, beta)[0]
            else:
                grad = tf.gradients(self.potential_energy(x, beta), x)[0]
        return grad
```

A.7 l2hmc-qcd/lattice/lattice.py

```python
"""
lattice.py

Contains implementation of GaugeLattice class.

Author: Sam Foreman (github: @saforem2)
Date: 01/15/2019
"""

import numpy as np
import tensorflow as tf
from scipy.special import i0, i1
from config 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 u1_plaq_exact_tf(beta):
    return tf.math.bessel_i1(beta) / tf.math.bessel_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=8,
                 space_size=8,
                 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)

        # create samples using @property method: `@samples.setter`
        self.samples = (num_samples, rand)

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

    @property
    def samples(self):
        return self._samples

    @samples.setter
    def samples(self, args):
        """Create samples."""
        num_samples, rand = args
        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)

        self._samples = 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.

        NOTE: self.samples.shape = (N, L, T, D), where:
            N = num_samples
            L = space_size
            T = time_size
            D = dimensionality
        """
        if samples is None:
            samples = self.samples

        with tf.name_scope('plaq_sums'):
            if samples.shape != self.samples.shape:
                samples = tf.reshape(samples, shape=(self.samples.shape))

            # assuming D = 2, plaq_sums will have shape: (N, L, T)
            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, plaq_sums=None):
        """Calculate the total action for each sample in samples."""
        if plaq_sums is None:
            if samples is None:
                samples = self.samples
            plaq_sums = self.calc_plaq_sums(samples)

        with tf.name_scope('actions'):
            total_actions = tf.reduce_sum(1. - tf.cos(plaq_sums),
                                          axis=(1, 2), name='actions')

        return total_actions

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

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

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

        with tf.name_scope('top_charges'):
            ps_proj = tf.sin(plaq_sums)
            #  if fft:
            #      ps_proj = tf.sin(self.calc_plaq_sums(samples))
            #  ps_proj = project_angle_fft(self.calc_plaq_sums(samples), N=1)
            #  else:
```

```
            #  ps_proj = tf.sin(self.calc_plaq_sums(samples))
            #  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

    def calc_top_charges_diff(self, x1, x2, plaq_sums1=None, plaq_sums2=None):
        """Calculate the difference in topological charge between x1 and x2."""
        with tf.name_scope('top_charges_diff'):
            charge_diff = tf.abs(self.calc_top_charges(x1, plaq_sums1)
                                 - self.calc_top_charges(x2, plaq_sums2))

        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.8 l2hmc-qcd/network/conv_net.py

```
"""
conv_net.py

Implements both 2D and 3D convolutional neural networks.


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

from config import GLOBAL_SEED, TF_FLOAT
from .network_utils import batch_norm

np.random.seed(GLOBAL_SEED)

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

######################################################################
# NOTE: Removed from `ConvNet2D.__init__` due to bugs when specifying
#       `data_format`
# --------------------------------------------------------------------
#  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'.")
######################################################################


class ConvNet2D(tf.keras.Model):
    """Convolutional block used in ConvNet3D."""
    def __init__(self, model_name, **kwargs):
        """
        Initialization method.

        Args:
            model_name: Name of the model.
            kwargs: Keyword arguments used to specify specifics of
                convolutional structure.
        """
        super(ConvNet2D, self).__init__(name=model_name)

        self.data_format = 'channels_last'

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

        self.activation = kwargs.get('conv_act', tf.nn.relu)
        activation2 = None if self.use_bn else self.activation

        if model_name == 'ConvNet2Dx':
            self.bn_name = 'batch_norm_x'
        elif model_name == 'ConvNet2Dv':
            self.bn_name = 'batch_norm_v'

        if self.name_scope == 'ConvNetX':
            conv1_name = 'conv_x1'
            pool1_name = 'pool_x1'
            conv2_name = 'conv_x2'
            pool2_name = 'pool_x2'
```

```
        elif self.name_scope == 'ConvNetV':
            conv1_name = 'conv_v1'
            pool1_name = 'pool_v1'
            conv2_name = 'conv_v2'
            pool2_name = 'pool_v2'

        with tf.name_scope(self.name_scope):

            self.conv1 = tf.keras.layers.Conv2D(
                filters=self.num_filters[0],
                kernel_size=self.filter_sizes[0],
                activation=self.activation,
                #  input_shape=self._input_shape[1:],
                name=conv1_name,
                padding='same',
                dtype=TF_FLOAT,
                data_format=self.data_format
            )

            #  with tf.name_scope('pool_x1'):
            self.max_pool1 = tf.keras.layers.MaxPooling2D(
                pool_size=(2, 2),
                strides=2,
                #  padding='same',
                name=pool1_name,
            )

            self.conv2 = tf.keras.layers.Conv2D(
                filters=self.num_filters[1],
                kernel_size=self.filter_sizes[1],
                activation=activation2,
                padding='same',
                name=conv2_name,
                dtype=TF_FLOAT,
                data_format=self.data_format
            )

            self.max_pool2 = tf.keras.layers.MaxPooling2D(
                pool_size=(2, 2),
                strides=2,
                #  padding='same',
                name=pool2_name,
            )

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

    def reshape_4D(self, tensor):
        """
        Reshape tensor to be compatible with tf.keras.layers.Conv3D.

        If self.data_format is 'channels_first', and input `tensor` has shape
        (N, 2, L, L), the output tensor has shape (N, 1, 2, L, L).

        If self.data_format is 'channels_last' and input `tensor` has shape
        (N, L, L, 2), the output tensor has shape (N, 2, L, L, 1).
        """
        N, H, W, C = self._input_shape
        if self.data_format == 'channels_first':
            #  N, C, H, W = self._input_shape
            #  N, D, H, W = tensor.shape
            if isinstance(tensor, np.ndarray):
                return np.reshape(tensor, (N, C, H, W))

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

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

```

```
            return tf.reshape(tensor, (N, H, W, C))

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

    def call(self, inputs, train_phase):
        """Forward pass through the network."""
        inputs = self.reshape_4D(inputs)

        inputs = self.max_pool1(self.conv1(inputs))
        inputs = self.conv2(inputs)
        if self.use_bn:
            inputs = self.activation(batch_norm(inputs, train_phase,
                                                axis=self.bn_axis,
                                                internal_update=True,
                                                scope=self.bn_name,
                                                reuse=tf.AUTO_REUSE))
        inputs = self.max_pool2(inputs)
        inputs = self.flatten(inputs)

        return inputs


class ConvNet3D(tf.keras.Model):
    """Convolutional block used in ConvNet3D."""
    def __init__(self, model_name, **kwargs):
        """
        Initialization method.

        Args:
            model_name: Name of the model.
            kwargs: Keyword arguments used to specify specifics of
                convolutional structure.
        """
        super(ConvNet3D, self).__init__(name=model_name)

        self.data_format = 'channels_last'

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

        if model_name == 'ConvNet3Dx':
            self.bn_name = 'batch_norm_x'
        elif model_name == 'ConvNet3Dv':
            self.bn_name = 'batch_norm_v'

        if self.name_scope == 'x_conv_block':
            conv1_name = 'conv_x1'
            pool1_name = 'pool_x1'
            conv2_name = 'conv_x2'
            pool2_name = 'pool_x2'
        elif self.name_scope == 'v_conv_block':
            conv1_name = 'conv_v1'
            pool1_name = 'pool_v1'
            conv2_name = 'conv_v2'
            pool2_name = 'pool_v2'

        with tf.name_scope(self.name_scope):
            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'.")

            self.activation = kwargs.get('conv_act', tf.nn.relu)
            activation2 = None if self.use_bn else self.activation
```

```
208
209             self.conv1 = tf.keras.layers.Conv3D(
210                 filters=self.num_filters[0],
211                 kernel_size=self.filter_sizes[0],
212                 activation=self.activation,
213                 #  input_shape=self._input_shape,
214                 padding='same',
215                 name=conv1_name,
216                 dtype=TF_FLOAT,
217                 data_format=self.data_format
218             )
219
220             self.max_pool1 = tf.keras.layers.MaxPooling3D(
221                 pool_size=(2, 2, 2),
222                 strides=2,
223                 padding='same',
224                 name=pool1_name,
225             )
226
227             self.conv2 = tf.keras.layers.Conv3D(
228                 filters=self.num_filters[1],
229                 kernel_size=self.filter_sizes[1],
230                 activation=activation2,
231                 #  activation=None if self.use_bn else tf.nn.relu,
232                 #  initializer=HE_INIT,
233                 padding='same',
234                 name=conv2_name,
235                 dtype=TF_FLOAT,
236                 data_format=self.data_format
237             )
238
239             self.max_pool2 = tf.keras.layers.MaxPooling3D(
240                 pool_size=(2, 2, 2),
241                 strides=2,
242                 padding='same',
243                 name=pool2_name,
244             )
245
246             self.flatten = tf.keras.layers.Flatten(name='flatten')
247
248         def reshape_5D(self, tensor):
249             """
250             Reshape tensor to be compatible with tf.keras.layers.Conv3D.
251
252             If self.data_format is 'channels_first', and input `tensor` has shape
253             (N, 2, L, L), the output tensor has shape (N, 1, 2, L, L).
254
255             If self.data_format is 'channels_last' and input `tensor` has shape
256             (N, L, L, 2), the output tensor has shape (N, 2, L, L, 1).
257             """
258             N, H, W, C = self._input_shape
259             if self.data_format == 'channels_first':
260                 #  N, C, H, W = self._input_shape
261                 #  N, D, H, W = tensor.shape
262                 if isinstance(tensor, np.ndarray):
263                     return np.reshape(tensor, (N, 1, C, H, W))
264
265                 return tf.reshape(tensor, (N, 1, C, H, W))
266
267             if self.data_format == 'channels_last':
268                 #  N, H, W, C = self._input_shape
269                 if isinstance(tensor, np.ndarray):
270                     return np.reshape(tensor, (N, H, W, C, 1))
271
272                 return tf.reshape(tensor, (N, H, W, C, 1))
273
274             raise AttributeError("`self.data_format` should be one of "
275                                  "'channels_first' or 'channels_last'")
276
277         def call(self, inputs, train_phase):
```

```
        """Forward pass through the network."""
        inputs = self.reshape_5D(inputs)

        inputs = self.max_pool1(self.conv1(inputs))
        inputs = self.conv2(inputs)
        if self.use_bn:
            inputs = self.activation(batch_norm(inputs, train_phase,
                                                axis=self.bn_axis,
                                                internal_update=True,
                                                scope=self.bn_name,
                                                reuse=tf.AUTO_REUSE))
        inputs = self.max_pool2(inputs)
        inputs = self.flatten(inputs)

        return inputs
```

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

```
"""
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

from config import TF_FLOAT, GLOBAL_SEED

from .network_utils import custom_dense


class GenericNet(tf.keras.Model):
    """Generic (fully-connected) network used in training L2HMC."""
    def __init__(self, model_name, **kwargs):
        """
        Initialization method.

        Args:
            model_name: Name of the model.
            kwargs: Keyword arguments used to specify specifics of
                convolutional structure.
        """
        super(GenericNet, self).__init__(name=model_name)

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

        self.activation = kwargs.get('generic_activation', tf.nn.relu)

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

            if self.dropout_prob > 0:
                self.dropout = tf.keras.layers.Dropout(self.dropout_prob,
                                                       seed=GLOBAL_SEED,)

            self.x_layer = custom_dense(self.num_hidden1,
                                        self.factor/3.,
                                        name='x_layer')
            self.v_layer = custom_dense(self.num_hidden1,
                                        1./3.,
                                        name='v_layer')
            self.t_layer = custom_dense(self.num_hidden1,
```

```
                                        1./3.,
                                        name='t_layer')

            self.h_layer = custom_dense(self.num_hidden2,
                                        name='hidden_layer')

            self.scale_layer = custom_dense(self.x_dim,
                                            0.001,
                                            name='scale_layer')
            if not self.zero_translation:
                self.translation_layer = custom_dense(self.x_dim,
                                                      0.001,
                                                      'translation_layer')

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

    def call(self, inputs, train_phase):
        v, x, t = inputs

        with tf.name_scope('v'):
            v = self.v_layer(v)
        with tf.name_scope('x'):
            x = self.x_layer(x)
        with tf.name_scope('t'):
            t = self.t_layer(t)

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

            # dropout gets applied to the output of the previous layer
            if self.dropout_prob > 0:
                h = self.dropout(h, training=train_phase)

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

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

        with tf.name_scope('translation'):
            if self.zero_translation:
                translation = tf.zeros_like(scale, name='translation')
            else:
                translation = self.translation_layer(h)

        return scale, translation, transformation
```

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

```python
"""
network.py

Implements network architecture used to train L2HMC algorithm on 2D U(1)
lattice gauge model.

Author: Sam Foreman (github: @saforem2)
Date: 07/22/2019
"""
import numpy as np
import tensorflow as tf
from .conv_net import ConvNet2D, ConvNet3D
from .generic_net import GenericNet
import utils.file_io as io

from config import GLOBAL_SEED


np.random.seed(GLOBAL_SEED)

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


class FullNet(tf.keras.Model):
    """Complete network used for training L2HMC model."""
    def __init__(self, model_name, **kwargs):
        """
        Initialization method.

        Args:
            model_name: Name of the model.
            kwargs: Keyword arguments used to specify specifics of
                convolutional structure.
        """
        super(FullNet, self).__init__(name=model_name)

        if model_name == 'XNet':
            generic_name_scope = 'GenericNetX'
        elif model_name == 'VNet':
            generic_name_scope = 'GenericNetV'

        with tf.name_scope(model_name):
            kwargs['name_scope'] = 'ConvNetX'
            network_arch = kwargs.get('network_arch', 'conv3D')

            if network_arch == 'conv2D':
                io.log('Using ConvNet2D architecture...')
                self.x_conv_net = ConvNet2D('ConvNet2Dx', **kwargs)

                kwargs['name_scope'] = 'ConvNetV'
                self.v_conv_net = ConvNet2D('ConvNet2Dv', **kwargs)

            elif network_arch == 'conv3D':
                io.log('Using ConvNet3D architecture...')
                self.x_conv_net = ConvNet3D('ConvNet3Dx', **kwargs)

                kwargs['name_scope'] = 'ConvNetV'
                self.v_conv_net = ConvNet3D('ConvNet3Dv', **kwargs)

            else:
                io.log('Using GenericNet architecture...')
                self.x_conv_net = self.v_conv_net = None

            kwargs['name_scope'] = generic_name_scope
            self.generic_net = GenericNet("GenericNet", **kwargs)

```

```
    def call(self, inputs, train_phase):
        v, x, t = inputs

        if self.x_conv_net is not None:
            v = self.v_conv_net(v, train_phase)
            x = self.x_conv_net(x, train_phase)

        scale, translation, transformation = self.generic_net([v, x, t],
                                                              train_phase)

        return scale, translation, transformation
```

A.11 l2hmc-qcd/network/network_utils.py

```python
import numpy as np
import tensorflow as tf
from tensorflow.contrib.framework import add_arg_scope, arg_scope

from config 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 activation_model(model):
    """Create Keras Model that outputs activations of all conv./pool layers.

    Args:
        model (tf.keraas.Model): Model for which we wish to visualize
            activations.
    Returns:
        activation_model (tf.keras.Model): Model that outputs the activations
            for each layer in `model.
    """
    layer_outputs = [layer.output for layer in model.layers]

    output_model = tf.keras.models.Model(inputs=model.input,
                                         output=layer_outputs)

    return output_model


def flatten(_list):
    return [item for sublist in _list for item in sublist]


def add_elements_to_collection(elements, collection_list):
    elements = flatten(elements)
    collection_list = flatten(collection_list)
    #  collection_list = tf.nest.flatten(collection_list)
    for name in collection_list:
        collection = tf.get_collection_ref(name)
        collection_set = set(collection)
        for element in elements:
            if element not in collection_set:
                collection.append(element)


def _assign_moving_average(orig_val, new_val, momentum, name):
    with tf.name_scope(name):
        scaled_diff = (1 - momentum) * (new_val - orig_val)
        return tf.assign_add(orig_val, scaled_diff)


@add_arg_scope
def batch_norm(x,
               phase,
               axis=-1,
               shift=True,
               scale=True,
               momentum=0.99,
               eps=1e-3,
               internal_update=False,
               scope=None,
               reuse=None):

    C = x._shape_as_list()[axis]
    ndim = len(x.shape)
```

```
    var_shape = [1] * (ndim - 1) + [C]

    with tf.variable_scope(scope, 'batch_norm', reuse=reuse):
        def training():
            m, v = tf.nn.moments(x, list(range(ndim - 1)), keep_dims=True)
            update_m = _assign_moving_average(moving_m,
                                              m, momentum,
                                              'update_mean')
            update_v = _assign_moving_average(moving_v,
                                              v, momentum,
                                              'update_var')
            #  tf.add_to_collection(tf.GraphKeys.UPDATE_OPS, update_m)
            #  tf.add_to_collection(tf.GraphKeys.UPDATE_OPS, update_v)
            tf.add_to_collection('update_ops', update_m)
            tf.add_to_collection('update_ops', update_v)

            if internal_update:
                with tf.control_dependencies([update_m, update_v]):
                    output = (x - m) * tf.rsqrt(v + eps)
            else:
                output = (x - m) * tf.rsqrt(v + eps)
            return output

        def testing():
            m, v = moving_m, moving_v
            output = (x - m) * tf.rsqrt(v + eps)
            return output

        # Get mean and variance, normalize input
        moving_m = tf.get_variable('mean', var_shape,
                                   initializer=tf.zeros_initializer,
                                   trainable=False)
        moving_v = tf.get_variable('var', var_shape,
                                   initializer=tf.ones_initializer,
                                   trainable=False)

        if isinstance(phase, bool):
            output = training() if phase else testing()
        else:
            output = tf.cond(phase, training, testing)

        if scale:
            output *= tf.get_variable('gamma', var_shape,
                                      initializer=tf.ones_initializer)

        if shift:
            output += tf.get_variable('beta', var_shape,
                                      initializer=tf.zeros_initializer)

    return output


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='truncated_normal',
            #  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,
            dtype=TF_FLOAT,
            uniform=False,
```

```
        )

    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)

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

A.12 l2hmc-qcd/trainers/trainer.py

```
"""
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

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


h_str = ("{:^12s}{:^10s}{:^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, **kwargs):
        """Returns new beta to follow annealing schedule."""
        beta_init = kwargs.get('beta_init', self.model.beta_init)
        beta_final = kwargs.get('beta_final', self.model.beta_final)
        train_steps = kwargs.get('train_steps', self.model.train_steps)

        temp = ((1. / beta_init - 1. / beta_final)
                * (1. - step / float(train_steps))
                + 1. / beta_final)
        new_beta = 1. / temp

        return new_beta

    def run_l2hmc_fns(self, key, samples_np, **kwargs):
        """Get numerical values of the scale, transl, and transf fns."""
        l2hmc_fns = self.model.dynamics.l2hmc_fns[key]

        feed_dict = {
            self.model.x: samples_np,
            self.model.train_phase: True,
        }

        ops = [l2hmc_fns['scale'], l2hmc_fns['transl'], l2hmc_fns['transf']]

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

```
68
69          return outputs
70
71      def train_step(self, step, samples_np, **kwargs):
72          """Perform a single training step.
73
74          Args:
75              step (int): Current training step.
76              samples_np (np.ndarray): Array of input data.
77              beta_np (float, optional): Input value for inverse coupling
78                  constant.
79
80          Returns:
81              out_data (dict)
82          """
83          beta_np = kwargs.get('beta_np', None)
84
85          # net_weights: [scale_w, translation_w, transformation_w]
86          net_weights = kwargs.get('net_weights', [1., 1., 1.])
87          train_steps = kwargs.get('train_steps', self.model.train_steps)
88
89          if beta_np is None:
90              if self.model.fixed_beta:
91                  beta_np = self.model.beta_init
92              else:
93                  beta_np = self.update_beta(step, train_steps=train_steps)
94
95          feed_dict = {
96              self.model.x: samples_np,
97              self.model.beta: beta_np,
98              self.model.net_weights[0]: net_weights[0],
99              self.model.net_weights[1]: net_weights[1],
100             self.model.net_weights[2]: net_weights[2],
101             self.model.train_phase: True,
102         }
103
104         global_step = self.sess.run(self.model.global_step)
105
106         ops = [
107             self.model.train_op,         # apply gradients
108             self.model.loss_op,          # calculate loss
109             self.model.x_out,            # get new samples
110             self.model.px,               # calculate accept prob.
111             self.model.dynamics.eps,     # calculate current step size
112             self.model.actions_op,       # calculate avg. actions
113             self.model.plaqs_op,         # calculate avg. plaqs
114             self.model.charges_op,       # calculate top. charges
115             self.model.charge_diffs_op,  # change in top. charge/num_samples
116             self.model.lr,               # evaluate learning rate
117         ]
118
119         start_time = time.time()
120         outputs = self.sess.run(ops, feed_dict=feed_dict)
121         dt = time.time() - start_time
122
123         out_data = {
124             'step': global_step,
125             'loss': outputs[1],
126             'samples': np.mod(outputs[2], 2 * np.pi),
127             'samples_orig': outputs[2],
128             'px': outputs[3],
129             'eps': outputs[4],
130             'actions': outputs[5],
131             'plaqs': outputs[6],
132             'charges': outputs[7],
133             'charge_diffs': outputs[8],
134             'lr': outputs[9],
135             'beta': beta_np
136         }
137
```

```
        data_str = (
            f"{global_step:>5g}/{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)
        net_weights = kwargs.get('net_weights', [1., 1., 1.])

        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

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

                if self.logger is not None:
                    self.logger.update_training(self.sess,
                                                out_data,
                                                net_weights,
                                                data_str)

            if self.logger is not None:
                self.logger.write_train_strings()

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

A.13 l2hmc-qcd/runners/runner.py

```
"""
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 loggers.run_logger import RunLogger
from config import RUN_HEADER


class GaugeModelRunner:

    def __init__(self, sess, params, inputs, run_ops, 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.params = params
        self.logger = logger

        if logger is not None:
            self.inputs_dict = self.logger.inputs_dict
            self.run_ops_dict = self.logger.run_ops_dict
        else:
            self.inputs_dict = RunLogger.build_inputs_dict(inputs)
            self.run_ops_dict = RunLogger.build_run_ops_dict(params, run_ops)

        self.eps = self.sess.run(self.run_ops_dict['dynamics_eps'])

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

    def run_step(self, step, run_steps, inputs, net_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

        keys = ['x_out', 'px', 'actions_op',
                'plaqs_op', 'charges_op', 'charge_diffs_op']

        if self.params['save_lf']:
            keys.extend(['lf_out_f', 'pxs_out_f',
                         'lf_out_b', 'pxs_out_b',
                         'masks_f', 'masks_b',
                         'logdets_f', 'logdets_b',
                         'sumlogdet_f', 'sumlogdet_b'])

        ops = [self.run_ops_dict[k] for k in keys]

        feed_dict = {
            self.inputs_dict['x']: samples_in,
            self.inputs_dict['beta']: beta_np,
            self.inputs_dict['net_weights'][0]: net_weights[0],
            self.inputs_dict['net_weights'][1]: net_weights[1],
            self.inputs_dict['net_weights'][2]: net_weights[2],
            self.inputs_dict['train_phase']: False
        }

        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),
            'samples_orig': outputs[0],
            'px': outputs[1],
            'actions': outputs[2],
            'plaqs': outputs[3],
            'charges': outputs[4],
            'charge_diffs': outputs[5],
        }

        if self.params['save_lf']:
            out_data.update({k: v for k, v in zip(keys[6:], outputs[6:])})
            #  lf_outputs = {}
            #  for key, val in zip(keys[6:], outputs[6:]):
            #      lf_outputs[key] = val
            #  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, **kwargs):
        """Run the simulation to generate samples and calculate observables.

        Args:
            run_steps: Number of steps to run the sampler for.
            kwargs: Dictionary of keyword arguments to use for running
                inference.

        Returns:
            observables: Tuple of observables dictionaries consisting of:
                (actions_dict, plaqs_dict, charges_dict, charge_diffs_dict).
```

```
        """
        run_steps = int(kwargs.get('run_steps', 5000))

        beta = kwargs.get('beta_final', self.params.get('beta_final', 5))
        net_weights = kwargs.get('net_weights', [1., 1., 1.])
        therm_frac = kwargs.get('therm_frac', 10)

        if beta is None:
            beta = self.params['beta_final']

        if net_weights is None:
            # scale_weight, translation weight, transformation weight
            net_weights = [1., 1., 1.]

        plaq_exact = u1_plaq_exact(beta)

        x_dim = (self.params['space_size']
                 * self.params['time_size']
                 * self.params['dim'])

        # start with randomly generated samples
        samples_np = np.random.randn(*(self.params['num_samples'], 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, net_weights)
                samples_np = out_data['samples']

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

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

        except (KeyboardInterrupt, SystemExit):
            io.log(80 * '-' + '\n')
            io.log("\nWARNING: KeyboardInterrupt detected!")
            io.log("WARNING: Saving current state and exiting.")
            if self.logger is not None:
                self.logger.save_run_data(therm_frac=therm_frac)
```

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

```
"""
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 datetime
import shutil
import errno

import tensorflow as tf
import numpy as np

from collections import Counter, OrderedDict
from scipy.stats import sem
import utils.file_io as io

from config import RUN_HEADER, NP_FLOAT

from lattice.lattice import u1_plaq_exact
#  from utils.tf_logging import variable_summaries

from .train_logger import save_params


FB_DICT = {
    'forward': [],
    'backward': [],
}


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:]


def copy(src, dest):
    try:
        shutil.copytree(src, dest)
    except OSError as e:
        # If the error was caused because the source wasn't a directory
        if e.errno == errno.ENOTDIR:
            shutil.copy(src, dest)
        else:
            io.log(f'Directory not copied. Error: {e}')


class RunLogger:
    def __init__(self, params, inputs, run_ops, save_lf_data=False):
        """
        Args:
            model: GaugeModel object.
            log_dir: Existing logdir from `TrainLogger`.
        """
        self.params = params
        self.save_lf_data = save_lf_data
        self.summaries = params['summaries']
```

```
        assert os.path.isdir(params['log_dir'])
        self.log_dir = params['log_dir']

        runs_dir = os.path.join(self.log_dir, 'runs')
        figs_dir = os.path.join(self.log_dir, 'figures')
        run_summaries_dir = os.path.join(self.log_dir, 'summaries', 'run')
        if os.path.isdir(runs_dir) or os.path.isdir(figs_dir):
            now = datetime.datetime.now()
            time_str = now.strftime("%H%M")
            if os.path.isdir(runs_dir):
                renamed_runs_dir = runs_dir + f'_{time_str}'
                #  io.log(f'Renaming existing runs_dir to: {renamed_runs_dir}')
                io.log(f'Copying existing runs_dir to: {renamed_runs_dir}')
                #  io.check_else_make_dir(renamed_runs_dir)
                copy(runs_dir, renamed_runs_dir)
                #  os.rename(runs_dir, renamed_runs_dir)
            if os.path.isdir(figs_dir):
                renamed_figs_dir = figs_dir + f'_{time_str}'
                #  io.log(f'Renaming existing figs_dir to: {renamed_figs_dir}')
                io.log(f'Copying existing figs_dir to: {renamed_figs_dir}')
                #  io.check_else_make_dir(renamed_figs_dir)
                copy(figs_dir, renamed_figs_dir)
                #  os.rename(figs_dir, renamed_figs_dir)
            if os.path.isdir(run_summaries_dir):
                new_rsd = run_summaries_dir + f'_{time_str}'
                io.log(f'Copying existing run summaries dir to: {new_rsd}')
                #  io.check_else_make_dir(new_rsd)
                copy(run_summaries_dir, new_rsd)

        self.runs_dir = runs_dir
        io.check_else_make_dir(self.runs_dir)

        self.figs_dir = figs_dir
        io.check_else_make_dir(self.figs_dir)

        self.run_summaries_dir = run_summaries_dir
        io.check_else_make_dir(self.run_summaries_dir)

        #  self._reset_counter = 0
        self.run_steps = None
        self.beta = None
        self.run_data = {}
        self.run_stats = {}
        self.run_strings = [RUN_HEADER]

        if params['save_lf']:
            self.samples_arr = []
            self.lf_out = FB_DICT.copy()
            self.pxs_out = FB_DICT.copy()
            self.masks = FB_DICT.copy()
            self.logdets = FB_DICT.copy()
            self.sumlogdet = FB_DICT.copy()
            self.l2hmc_fns = FB_DICT.copy()

        self.run_ops_dict = self.build_run_ops_dict(params, run_ops)
        self.inputs_dict = self.build_inputs_dict(inputs)

        if self.summaries:
            #  self.run_summaries_dir = os.path.join(self.log_dir,
            #                                        'summaries', 'run')
            #  io.check_else_make_dir(self.run_summaries_dir)
            self.writer = tf.summary.FileWriter(self.run_summaries_dir,
                                                tf.get_default_graph())
            self.create_summaries()

    @staticmethod
    def build_run_ops_dict(params, run_ops):
        """Build dictionary of tensorflow operations used for inference."""
        def get_lf_keys(direction):
            base_keys = ['lf_out', 'pxs_out', 'masks',
```

```
                        'logdets', 'sumlogdet', 'fns_out']
            return [k + f'_{direction}' for k in base_keys]

        keys = ['x_out', 'px', 'actions_op',
                'plaqs_op', 'avg_plaqs_op',
                'charges_op', 'charge_diffs_op']

        run_ops_dict = {key: run_ops[idx] for idx, key in enumerate(keys)}

        if params['save_lf']:
            keys.extend(get_lf_keys('f'))
            keys.extend(get_lf_keys('b'))
            for key, val in zip(keys[7:], run_ops[7:]):
                run_ops_dict.update({key: val})

        run_ops_dict['dynamics_eps'] = run_ops[-1]

        return run_ops_dict

    @staticmethod
    def build_inputs_dict(inputs):
        """Build dictionary of tensorflow placeholders used as inputs."""
        keys = ['x', 'beta', 'charge_weight', 'train_phase']
        num_keys = len(keys)

        inputs_dict = dict(zip(keys, inputs[:num_keys]))
        inputs_dict.update({'net_weights': inputs[num_keys:]})

        return inputs_dict

    def create_summaries(self):
        """Create summary objects for logging in TensorBoard."""
        summary_list = tf.get_collection(tf.GraphKeys.SUMMARIES)
        ignore_strings = ['loss', 'learning_rate',
                          'step_size', 'train', 'eps']
        run_summaries = [
            i for i in summary_list if not any(
                s in i.name for s in ignore_strings
            )
        ]
        self.summary_op = tf.summary.merge(run_summaries)

    def log_step(self, sess, step, samples_np, beta_np, net_weights):
        """Update self.logger.summaries."""
        feed_dict = {
            self.inputs_dict['x']: samples_np,
            self.inputs_dict['beta']: beta_np,
            self.inputs_dict['net_weights'][0]: net_weights[0],
            self.inputs_dict['net_weights'][1]: net_weights[1],
            self.inputs_dict['net_weights'][2]: net_weights[2],
            self.inputs_dict['train_phase']: False
        }
        summary_str = sess.run(self.summary_op, feed_dict=feed_dict)

        self.writer.add_summary(summary_str, global_step=step)
        self.writer.flush()

    @classmethod
    def _clear(cls):
        cls.run_data = None
        cls.run_strings = None
        if cls.params['save_lf']:
            cls.samples_arr = None
            cls.lf_out = None
            cls.pxs_out = None
            cls.masks = None
            cls.logdets = None
            cls.sumlogdet = None
            cls.l2hmc_fns = None

```

```
208             cls.params['net_weights'] = None
209             cls.run_dir = None
210             if cls.summaries:
211                 cls.run_summary_dir = None
212                 cls.writer = None
213 
214         def clear(self):
215             self.run_data = None
216             self.run_strings = None
217             if self.params['save_lf']:
218                 self.samples_arr = None
219                 self.lf_out = None
220                 self.pxs_out = None
221                 self.masks = None
222                 self.logdets = None
223                 self.sumlogdet = None
224                 self.l2hmc_fns = None
225             self.params['net_weights'] = None
226             self.run_dir = None
227             if self.summaries:
228                 self.run_summary_dir = None
229                 self.writer = None
230 
231         def existing_run(self, run_str):
232             """Check if this run has been completed previously, if so skip it."""
233             run_dir = os.path.join(self.runs_dir, run_str)
234             run_summary_dir = os.path.join(self.run_summaries_dir, run_str)
235             if os.path.isdir(run_dir) and os.path.isdir(run_summary_dir):
236                 return True
237             return False
238 
239         def _get_run_str(self, **kwargs):
240             """Parse parameter values and create unique string to name the dir."""
241             run_steps = kwargs.get('run_steps', 5000)
242             beta = kwargs.get('beta', 5.)
243             net_weights = kwargs.get('net_weights', [1., 1., 1.])
244             eps_np = kwargs.get('eps', None)
245             dir_append = kwargs.get('dir_append', None)
246             #  eps = self.model.eps
247             eps_str = f'{eps_np:.3}'.replace('.', '')
248             beta_str = f'{beta:.3}'.replace('.', '')
249             #  qw_str = f'{charge_weight:.3}'.replace('.', '')
250             scale_wstr = f'{net_weights[0]:3.2f}'.replace('.', '')
251             transl_wstr = f'{net_weights[1]:3.2f}'.replace('.', '')
252             transf_wstr = f'{net_weights[2]:3.2f}'.replace('.', '')
253 
254             run_str = (f'steps{run_steps}'
255                        f'_beta{beta_str}'
256                        f'_eps{eps_str}'
257                        #  f'_qw_{qw_str:.2}'
258                        f'_S{scale_wstr}'
259                        f'_T{transl_wstr}'
260                        f'_Q{transf_wstr}')
261             #  f'_{self._reset_counter}')
262 
263             if dir_append is not None:
264                 run_str += dir_append
265 
266             return run_str
267 
268         def reset(self, **kwargs):
269             """Reset run_data and run_strings to prep for new run."""
270             run_steps = kwargs.get('run_steps', 5000)
271             beta = kwargs.get('beta', 5.)
272             net_weights = kwargs.get('net_weights', [1., 1., 1.])
273             #  eps_np = kwargs.get('eps', None)
274             #  dir_append = kwargs.get('dir_append', None)
275 
276             self.run_steps = int(run_steps)
277             self.beta = beta
```

```
278
279             self.run_data = {
280                 'px': {},
281                 'actions': {},
282                 'plaqs': {},
283                 'charges': {},
284                 'charge_diffs': {},
285             }
286             self.run_stats = {}
287             self.run_strings = []
288             #  if self.model.save_lf:
289             if self.params['save_lf']:
290                 self.samples_arr = []
291                 self.lf_out = FB_DICT.copy()
292                 self.pxs_out = FB_DICT.copy()
293                 self.masks = FB_DICT.copy()
294                 self.logdets = FB_DICT.copy()
295                 self.sumlogdet = FB_DICT.copy()
296                 self.l2hmc_fns = FB_DICT.copy()
297
298             #  params = self.model.params
299             self.params['net_weights'] = net_weights
300
301             run_str = self._get_run_str(**kwargs)
302             self._set_run_dirs(run_str)
303             self._save_net_weights(net_weights)
304
305             if self.summaries:
306                 self.writer = tf.summary.FileWriter(self.run_summary_dir,
307                                                     tf.get_default_graph())
308             save_params(self.params, self.run_dir)
309             #  self._reset_counter += 1
310
311             #  return self.run_dir, run_str
312
313         def _save_net_weights(self, net_weights):
314             """Write net weights to a `.txt` file; append if existing."""
315             #  def _round_float_as_str(f):
316             #      return f'{f:.3g}'
317             weights_txt_file = os.path.join(self.run_dir, 'net_weights.txt')
318             nw_str = [f'{w:.3g}' for w in net_weights]
319             #  nw_str = [_round_float_as_str(w) for w in net_weights]
320             w_str = '[' + nw_str[0] + nw_str[1] + nw_str[2] + ']'
321             if not os.path.isfile(weights_txt_file):
322                 with open(weights_txt_file, 'w') as f:
323                     f.write('[scale_weight,'
324                             ' translation_weight,'
325                             ' transformation_weight]')
326                     f.write(80 * '-' + '\n')
327             else:
328                 with open(weights_txt_file, 'a') as f:
329                     f.write(w_str)
330
331         def _set_run_dirs(self, run_str):
332             """Sets dirs containing data about inference run using run_str."""
333             self.run_dir = os.path.join(self.runs_dir, run_str)
334             io.check_else_make_dir(self.run_dir)
335             if self.summaries:
336                 self.run_summary_dir = os.path.join(self.run_summaries_dir,
337                                                     run_str)
338                 io.check_else_make_dir(self.run_summary_dir)
339
340         def update(self, sess, data, net_weights, data_str):
341             """Update run_data and append data_str to data_strings."""
342             # projection of samples onto [0, 2pi) done in run_step above
343             step = data['step']
344             beta = data['beta']
345             key = (step, beta)
346
347             obs_keys = ['px', 'actions', 'plaqs', 'charges', 'charge_diffs']
```

```
        for k in obs_keys:
            self.run_data[k][key] = data[k]

        if self.params['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 self.summaries and (step + 1) % self.params['logging_steps'] == 0:
            self.log_step(sess, step, data['samples'], beta, net_weights)

        if step % (10 * self.params['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:
            attr = np.array(attr, dtype=dtype)

        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.save_lf_data:
            keys = ['lf_out', 'masks', 'logdets', 'sumlogdet', 'pxs_out']
            for key in keys:
                f = 'forward'
                b = 'backward'
                self.save_attr(key + f'_{f}', getattr(self, key)[f])
                self.save_attr(key + f'_{b}', getattr(self, key)[b])

        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)
```

```
488
489         history_file = os.path.join(self.run_dir, 'run_history.txt')
490         io.write(RUN_HEADER, history_file, 'w')
491         _ = [io.write(s, history_file, 'a') for s in self.run_strings]
492
493         self.write_run_stats(run_stats, therm_frac)
494
495     def write_run_stats(self, stats, therm_frac=10):
496         """Write statistics in human readable format to .txt file."""
497         #  run_steps = kwargs['run_steps']
498         #  beta = kwargs['beta']
499         #  current_step = kwargs['current_step']
500         #  therm_steps = kwargs['therm_steps']
501         #  training = kwargs['training']
502         #  run_dir = kwargs['run_dir']
503         therm_steps = self.run_steps // therm_frac
504
505         out_file = os.path.join(self.run_dir, 'run_stats.txt')
506
507         actions_avg, actions_err = stats['actions'].mean(axis=0)
508         plaqs_avg, plaqs_err = stats['plaqs'].mean(axis=0)
509         charges_avg, charges_err = stats['charges'].mean(axis=0)
510         suscept_avg, suscept_err = stats['suscept'].mean(axis=0)
511
512         #  ns = self.model.num_samples
513         ns = self.params['num_samples']
514         suscept_k1 = f'  \navg. over all {ns} samples < Q >'
515         suscept_k2 = f'  \navg. over all {ns} samples < Q^2 >'
516         actions_k1 = f'  \navg. over all {ns} samples < action >'
517         plaqs_k1 = f'  \n avg. over all {ns} samples < plaq >'
518
519         _est_key = '  \nestimate +/- stderr'
520
521         suscept_ss = {
522             suscept_k1: f"{charges_avg:.4g} +/- {charges_err:.4g}",
523             suscept_k2: f"{suscept_avg:.4g} +/- {suscept_err:.4g}",
524             _est_key: {}
525         }
526
527         actions_ss = {
528             actions_k1: f"{actions_avg:.4g} +/- {actions_err:.4g}\n",
529             _est_key: {}
530         }
531
532         plaqs_ss = {
533             'exact_plaq': f"{u1_plaq_exact(self.beta):.4g}\n",
534             plaqs_k1: f"{plaqs_avg:.4g} +/- {plaqs_err:.4g}\n",
535             _est_key: {}
536         }
537
538         def format_stats(x, name=None):
539             return [f'{name}: {i[0]:.4g} +/- {i[1]:.4g}' for i in x]
540
541         def zip_keys_vals(stats_strings, keys, vals):
542             for k, v in zip(keys, vals):
543                 stats_strings[_est_key][k] = v
544             return stats_strings
545
546         keys = [f"sample {idx}" for idx in range(ns)]
547
548         suscept_vals = format_stats(stats['suscept'], '< Q^2 >')
549         actions_vals = format_stats(stats['actions'], '< action >')
550         plaqs_vals = format_stats(stats['plaqs'], '< plaq >')
551
552         suscept_ss = zip_keys_vals(suscept_ss, keys, suscept_vals)
553         actions_ss = zip_keys_vals(actions_ss, keys, actions_vals)
554         plaqs_ss = zip_keys_vals(plaqs_ss, keys, plaqs_vals)
555
556         def accumulate_strings(d):
557             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')

        run_str = (f" stats for {ns} chains ran for {self.run_steps} steps "
                   f" at beta = {self.beta}.")

        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('')

            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)
        log_and_write(ss3, str3, therm_str, charge_probs_strings, out_file)
        log_and_write(ss3, str3, therm_str, charge_probs_strings, out_file)
```

A.15 l2hmc-qcd/loggers/train_logger.py

```
"""
train_logger.py

Implements TrainLogger class responsible for saving/logging data from
GaugeModel.

Author: Sam Foreman (github: @saforem2)
Date: 04/09/2019
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os
import pickle

import tensorflow as tf

import utils.file_io as io

from config import TRAIN_HEADER
from .summary_utils import create_summaries
from utils.file_io import save_params


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.lattice.samples_tensor,
        }

        self.train_data_strings = [TRAIN_HEADER]
        self.train_data = {
            'loss': {},
            'actions': {},
            'plaqs': {},
            'charges': {},
            'charge_diffs': {},
            'px': {}
        }

        if self.model.save_lf:
            self.train_data['l2hmc_fns'] = {
                'forward': [],
                'backward': [],
            }

        # log_dir will be None if using_hvd and hvd.rank() != 0
        # this prevents workers on different ranks from corrupting checkpoints
        #  if log_dir is not None and self.is_chief:
        self._create_dir_structure(log_dir)
        save_params(self.model.params, self.log_dir)

        if self.summaries:
            #  with tf.variable_scope('train_summaries'):
            self.writer = tf.summary.FileWriter(self.train_summary_dir,
```

```
                                                tf.get_default_graph())
            self.summary_writer, self.summary_op = create_summaries(
                model, self.train_summary_dir, training=True
            )

    def _create_dir_structure(self, log_dir):
        """Create relevant directories for storing data.

        Args:
            log_dir: Root directory in which all other directories are created.

        Returns:
            None
        """
        dirs = {
            'log_dir': log_dir,
            'checkpoint_dir': os.path.join(log_dir, 'checkpoints'),
            'train_dir': os.path.join(log_dir, 'training'),
            'train_summary_dir': os.path.join(log_dir, 'summaries', 'train'),
        }
        files = {
            'train_log_file': os.path.join(dirs['train_dir'],
                                           'training_log.txt'),
            'current_state_file': os.path.join(dirs['train_dir'],
                                               'current_state.pkl')
        }

        for key, val in dirs.items():
            io.check_else_make_dir(val)
            setattr(self, key, val)

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

    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, net_weights):
        """Update self.logger.summaries."""
        feed_dict = {
            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.train_phase: True
        }
        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, net_weights, data_str):
        """Update _current state and train_data."""
        step = data['step']
        beta = data['beta']
        self._current_state['step'] = step
        self._current_state['beta'] = beta
        self._current_state['lr'] = data['lr']
```

```
        self._current_state['eps'] = data['eps']
        self._current_state['samples'] = data['samples']
        self._current_state['net_weights'] = net_weights

        key = (step, beta)

        self.charges_dict[key] = data['charges']
        self.charge_diffs_dict[key] = data['charge_diffs']

        obs_keys = ['loss', 'actions',
                    'plaqs', 'charges',
                    'charge_diffs', 'px']
        for obs_key in obs_keys:
            self.train_data[obs_key][key] = data[obs_key]

        self.train_data_strings.append(data_str)
        if step % self.model.print_steps == 0:
            io.log(data_str)

        if self.summaries and (step + 1) % self.model.logging_steps == 0:
            self.log_step(sess, step, data['samples'], beta, net_weights)

        if (step + 1) % self.model.save_steps == 0:
            self.save_current_state()

        if step % 100 == 0:
            io.log(TRAIN_HEADER)

    def write_train_strings(self):
        """Write training strings out to file."""
        tlf = self.train_log_file
        _ = [io.write(s, tlf, 'a') for s in self.train_data_strings]
```

A.16 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 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 config import COLORS, MARKERS, HAS_MATPLOTLIB

from .plot_utils import MPL_PARAMS, plot_multiple_lines

if HAS_MATPLOTLIB:
    import matplotlib as mpl
    import matplotlib.pyplot as plt
    mpl.rcParams.update(MPL_PARAMS)


def arr_from_dict(d, key):
    return np.array(list(d[key].values()))


def get_out_file(out_dir, out_str):
    return os.path.join(out_dir, out_str + '.pdf')


class GaugeModelPlotter:
    def __init__(self, params, figs_dir=None, experiment=None):
        self.figs_dir = figs_dir
        self.params = params
        #  self.model = model
        if experiment is not None:
            self.experiment = experiment

    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 log_figure(self):
        try:
            self.experiment.log_figure()
        except AttributeError:
            pass

    def _parse_data(self, data, beta):
        """Helper method for extracting relevant data from `data`."""
        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)

        actions_avg = np.mean(actions, axis=1)
        actions_err = sem(actions, axis=1)

        plaqs_avg = np.mean(plaqs, axis=1)
        plaqs_err = sem(plaqs, axis=1)

        autocorrs_avg = np.mean(charge_autocorrs.T, axis=1)
        autocorrs_err = sem(charge_autocorrs.T, axis=1)

        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)))

        _charge_diffs = charge_diffs[warmup_steps:][::skip_steps]
        _plaq_diffs = plaqs_diffs[warmup_steps:][::skip_steps]
        _steps_diffs = (
            skip_steps * np.arange(_plaq_diffs.shape[0]) + skip_steps
        )
        _plaq_diffs_avg = np.mean(_plaq_diffs, axis=1)
        _plaq_diffs_err = sem(_plaq_diffs, axis=1)

        xy_data = {
            'actions': (steps_arr, actions_avg, actions_err),
            'plaqs': (steps_arr, plaqs_avg, plaqs_err),
            'charges': (steps_arr, charges.T),
            'charge_diffs': (_steps_diffs, _charge_diffs.T),
            'autocorrs': (steps_arr, autocorrs_avg, autocorrs_err),
```

```
            'plaqs_diffs': (_steps_diffs, _plaq_diffs_avg, _plaq_diffs_err)
        }

        return xy_data

    def _plot_setup(self, data, **kwargs):
        """Prepare for plotting observables."""
        beta = kwargs.get('beta', 5.)
        run_str = kwargs.get('run_str', None)
        net_weights = kwargs.get('net_weights', [1., 1., 1.])
        dir_append = kwargs.get('dir_append', None)

        if dir_append:
            run_str += dir_append

        self.out_dir = os.path.join(self.figs_dir, run_str)
        io.check_else_make_dir(self.out_dir)

        #  L = self.params['space_size']
        lf_steps = self.params['num_steps']
        bs = self.params['num_samples']  # batch size
        #  qw = weights['charge_weight']
        nw = net_weights
        sw, translw, transfw = nw
        title_str = (r"$N_{\mathrm{LF}} = $" + f"{lf_steps}, "
                     r"$N_{\mathrm{B}} = $" + f"{bs}, "
                     r"$\mathrm{nw} = $" + f"{nw[0], nw[1], nw[2]}")

        #  r"$L = $" + f"{L}, "
        #  r"$\beta = $ " + f"{beta}, "
        #  r"$\alpha_{Q} = $" + f"{qw}, "
        kwargs.update({
            'markers': False,
            'lines': True,
            'alpha': 0.6,
            'title': title_str,
            'legend': False,
            'ret': False,
            'out_file': [],
        })

        xy_data = self._parse_data(data, beta)

        return xy_data, kwargs

    def plot_observables(self, data, **kwargs):
        """Plot observables."""
        #  beta = kwargs.get('beta', 5.)
        #  run_str = kwargs.get('run_str', None)
        #  net_weights = kwargs.get('net_weights', [1., 1., 1.])
        #  dir_append = kwargs.get('dir_append', None)

        xy_data, kwargs = self._plot_setup(data, **kwargs)

        self._plot_actions(xy_data['actions'], **kwargs)
        self.log_figure()
        self._plot_plaqs(xy_data['plaqs'], **kwargs)
        self.log_figure()
        self._plot_charges(xy_data['charges'], **kwargs)
        self.log_figure()
        self._plot_charge_probs(xy_data['charges'][1], **kwargs)
        self.log_figure()
        self._plot_charge_diffs(xy_data['charge_diffs'], **kwargs)
        self.log_figure()
        self._plot_autocorrs(xy_data['autocorrs'], **kwargs)
        self.log_figure()
        mean_diff = self._plot_plaqs_diffs(xy_data['plaqs_diffs'], **kwargs)
        self.log_figure()

        return mean_diff
```

```
208
209         def _plot(self, xy_data, **kwargs):
210             """Basic plotting wrapper."""
211             x, y, yerr = xy_data
212
213             labels = kwargs.get('labels', None)
214             if labels is not None:
215                 xlabel = labels.get('x_label', '')
216                 ylabel = labels.get('y_label', '')
217             else:
218                 xlabel = ''
219                 ylabel = ''
220
221             _leg = kwargs.get('legend', False)
222
223             if kwargs.get('two_rows', False):
224                 fig, (ax0, ax1) = plt.subplots(
225                     nrows=2, ncols=1, gridspec_kw={'height_ratios': [2.5, 1],
226                                                    'hspace': 0.175}
227                 )
228                 n = len(x)
229                 mid = n // 2
230                 x0 = int(mid - 0.025 * n)
231                 x1 = int(mid + 0.025 * n)
232             else:
233                 fig, ax0 = plt.subplots()
234                 ax1 = None
235
236             plt_kwargs = {
237                 'color': 'k',
238                 #  'lw': 1.,
239                 #  'ls': '-',
240                 'alpha': 0.8,
241                 'marker': ',',
242             }
243             #  err_kwargs = plt_kwargs.update({'lw': 1.5, 'alpha': 0.7})
244
245             ax0.plot(x, y, **plt_kwargs)
246             ax0.errorbar(x, y, yerr=yerr,
247                          #  ls='-', lw=1.,
248                          alpha=0.7,
249                          color='k',
250                          ecolor='gray')
251
252             if ax1 is not None:
253                 ax1.plot(x[x0:x1:10], y[x0:x1:10], **plt_kwargs)
254                 ax1.errorbar(x[x0:x1:10], y[x0:x1:10], yerr=yerr[x0:x1:10],
255                              #  ls='-', lw=1.,
256                              alpha=0.7,
257                              color='k',
258                              ecolor='gray')
259
260             ax1.set_xlabel(xlabel, fontsize=14)
261             ax0.set_ylabel(ylabel, fontsize=14)
262             ax1.set_ylabel('', fontsize=14)
263             if _leg:
264                 ax0.legend(loc='best')
265
266             title = kwargs.get('title', None)
267             if title is not None:
268                 _ = ax0.set_title(title)
269
270             plt.tight_layout()
271             if kwargs.get('save', True):
272                 fname = kwargs.get('fname', f'plot_{np.random.randint(10)}')
273                 out_file = get_out_file(self.out_dir, fname)
274                 io.log(f'Saving figure to: {out_file}.')
275                 plt.savefig(out_file, dpi=400, bbox_inches='tight')
276
277             return fig, (ax0, ax1)
```

```
278
279         def _plot_actions(self, xy_data, **kwargs):
280             """Plot actions."""
281             #  kwargs['out_file'] = get_out_file(self.out_dir, 'actions_vs_step')
282             labels = {
283                 'x_label': 'Step',
284                 'y_label': 'Action',
285                 'plt_label': 'Action'
286             }
287
288             kwargs.update({
289                 'fname': 'actions_vs_step',
290                 'labels': labels,
291                 'two_rows': True,
292             })
293             self._plot(xy_data, **kwargs)
294             #  kwargs['bounds'] = [0.2, 0.6, 0.7, 0.3]
295
296             #  xy_labels = ('Step', 'Action')
297             #  plot_multiple_lines(xy_data, xy_labels, **kwargs)
298             #  plot_with_inset(xy_data, labels, **kwargs)
299
300         def _plot_plaqs(self, xy_data, beta, **kwargs):
301             """PLot average plaquette."""
302             labels = {
303                 'x_label': 'Step',
304                 'y_label': r"""$\langle \phi_{P} \rangle$""",
305                 'plt_label': r"""$\langle \phi_{P} \rangle$"""
306             }
307             kwargs.update({
308                 'labels': labels,
309                 'fname': 'plaqs_vs_step',
310                 'two_rows': True,
311                 'save': False,
312             })
313             fig, (ax0, ax1) = self._plot(xy_data, **kwargs)
314
315             ax0.axhline(y=u1_plaq_exact(beta),
316                         color='#CC0033', ls='-', lw=1.5, label='exact')
317             ax1.axhline(y=u1_plaq_exact(beta),
318                         color='#CC0033', ls='-', lw=1.5, label='exact')
319
320             plt.tight_layout()
321
322             out_file = get_out_file(self.out_dir, 'plaqs_vs_step')
323             io.log(f'Saving figure to: {out_file}')
324             plt.savefig(out_file, dpi=400, bbox_inches='tight')
325
326         def _plot_plaqs_diffs(self, xy_data, **kwargs):
327             kwargs['out_file'] = None
328             kwargs['ret'] = True
329             labels = {
330                 'x_label': 'Step',
331                 'y_label': r"$\delta_{\phi_{P}}$",
332                 'plt_label': r"$\delta_{\phi_{P}}$"
333             }
334             x, y, yerr = xy_data
335             y_mean = np.mean(y)
336             fig, ax = plt.subplots()
337             _ = ax.plot(x, y, label='', marker=',', color='k', alpha=0.8)
338             _ = ax.errorbar(x, y, yerr=yerr, label='', marker=None, ls='',
339                             alpha=0.7, color='gray', ecolor='gray')
340             _ = ax.axhline(y=0, color='#CC0033', ls='-', lw=2.)
341             _ = ax.axhline(y=y_mean, label=f'avg {y_mean:.5f}',
342                            color='C2', ls='-', lw=2.)
343
344             _ = ax.set_xlabel(labels['x_label'], fontsize=14)
345             _ = ax.set_ylabel(labels['y_label'], fontsize=14)
346             title = kwargs.get('title', None)
347             if title is not None:
```

```
            _ = ax.set_title(title)

        ax.legend(loc='best')

        _ = plt.tight_layout()
        out_file = get_out_file(self.out_dir, 'plaqs_diffs_vs_step')
        io.log(f'Saving figure to: {out_file}.')
        plt.savefig(out_file, dpi=400, bbox_inches='tight')

        return y_mean

    def _plot_charges(self, xy_data, **kwargs):
        """Plot topological charges."""
        kwargs['out_file'] = get_out_file(self.out_dir, 'charges_vs_step')
        kwargs['markers'] = True
        kwargs['lines'] = False
        kwargs['alpha'] = 1.
        kwargs['ret'] = False
        xy_labels = ('Step', r'$Q$')
        plot_multiple_lines(xy_data, xy_labels, **kwargs)

        charges = np.array(xy_data[1].T, dtype=int)
        num_steps, num_samples = charges.shape

        out_dir = os.path.join(self.out_dir, 'top_charge_plots')
        io.check_else_make_dir(out_dir)
        # if we have more than 10 chains in charges, only plot first 10
        for idx in range(min(num_samples, 5)):
            _, ax = plt.subplots()
            _ = ax.plot(charges[:, idx],
                        marker=MARKERS[idx],
                        color=COLORS[idx],
                        ls='',
                        alpha=0.5,
                        label=f'sample {idx}')
            _ = ax.legend(loc='best')
            _ = ax.set_xlabel(xy_labels[0], fontsize=14)
            _ = ax.set_ylabel(xy_labels[1], fontsize=14)
            _ = ax.set_title(kwargs['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 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()
        ax.plot(list(all_counts.keys()),
                np.array(list(all_counts.values()) / (total_counts *
                                                      num_samples)),
                marker='o',
                color='C0',
                ls='',
                alpha=0.6,
                label=f'total across {num_samples} samples')
        _ = 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(self.out_dir, f'TOP_CHARGE_PROBS_ALL')
        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:
        plt.close('all')

    def _plot_autocorrs(self, xy_data, **kwargs):
        """Plot topological charge autocorrelations."""
        #  xy_labels = ('Step', 'Autocorrelation of ' + r'$Q$')
        #  return plot_multiple_lines(xy_data, xy_labels, **kwargs)
        #  kwargs['out_file'] = get_out_file(self.out_dir,
        #                                    'charge_autocorrs_vs_step')
        #  kwargs['ret'] = True
        #  kwargs['bounds'] = [0.2, 0.6, 0.7, 0.3]
        #  try:
        #      kwargs['out_file'] = get_out_file(
        #          self.out_dir, 'charge_autocorrs_vs_step'
        #      )
        #  except AttributeError:
        #      kwargs['out_file'] = None
        labels = {
            'x_label': 'Step',
            'y_label': 'Autocorrelation of ' + r'$Q$',
            'plt_label': 'Autocorrelation of ' + r'$Q$',
        }
```

```
488         kwargs.update({
489             'labels': labels,
490             'fname': 'charge_autocorrs',
491             'two_rows': True,
492         })
493         self._plot(xy_data, **kwargs)
494         #  _, ax, axins = plot_with_inset(xy_data, labels, **kwargs)
495         #  xy_labels = ('Step', 'Autocorrelation of ' + r'$Q$')
496         #  return plot_multiple_lines(xy_data, xy_labels, **kwargs)
```

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

```python
import os
import pickle
import numpy as np
import utils.file_io as io
from utils.data_loader import DataLoader

from config import HAS_MATPLOTLIB, HAS_PSUTIL

if HAS_MATPLOTLIB:
    import matplotlib as mpl
    import matplotlib.pyplot as plt

if HAS_PSUTIL:
    import psutil


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 = {
    '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,
    '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


def print_memory():
    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 * '-')


class LeapfrogPlotter:
    def __init__(self, figs_dir, run_logger=None, run_dir=None):
        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.
                    """
                )
```

```
            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
        }

    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 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]

        #  print_memory()
        fig_ax1 = self.plot_lf_diffs(beta, num_samples)

        #  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):
        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)
        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)
```

```
208
209         beta_str = str(beta).replace('.', '')
210         fn = f'leapfrog_diffs_beta{beta_str}'
211         out_file = os.path.join(self.pdfs_dir, fn + '.pdf')
212         out_file_zoom = os.path.join(self.pdfs_dir, fn + '_zoom.pdf')
213         out_file_zoom1 = os.path.join(self.pdfs_dir, fn + '_zoom1.pdf')
214         io.log(f'Saving figure to: {out_file}')
215         _ = plt.savefig(out_file, dpi=400, bbox_inches='tight')
216
217         lf_xlim = 100
218         md_xlim = lf_xlim // self.num_lf_steps
219
220         _ = ax1.set_xlim((0, lf_xlim))
221         _ = ax2.set_xlim((0, md_xlim))
222         _ = plt.savefig(out_file_zoom, dpi=400, bbox_inches='tight')
223
224         lf_xlim //= 4
225         md_xlim //= 4
226         _ = ax1.set_xlim((0, lf_xlim))
227         _ = ax1.set_ylim((-0.05, 0.25))
228         _ = ax2.set_xlim((0, md_xlim))
229         _ = plt.savefig(out_file_zoom1, dpi=400, bbox_inches='tight')
230
231         return fig, (ax1, ax2)
232
233     def plot_logdets(self, beta, num_samples=20):
234         reds, blues = self.get_colors(num_samples)
235
236         sumlogdet_yf_avg = np.mean(self.sumlogdet_f, axis=-1)
237         sumlogdet_yb_avg = np.mean(self.sumlogdet_b, axis=-1)
238         sumlogdet_xf_avg = np.arange(len(sumlogdet_yf_avg))
239         sumlogdet_xb_avg = np.arange(len(sumlogdet_yb_avg))
240
241         fig, (ax1, ax2) = plt.subplots(2, 1)
242         for idx in range(num_samples):
243             yf = self.logdets_f[:, idx]
244             xf = np.arange(len(yf))
245             yb = self.logdets_b[:, idx]
246             xb = np.arange(len(yb))
247
248             _ = ax1.plot(xf, yf, color=reds[idx], **self.indiv_kwargs)
249             _ = ax1.plot(xb, yb, color=blues[idx], **self.indiv_kwargs)
250
251         yf_avg = np.mean(self.logdets_f, axis=-1)
252         yb_avg = np.mean(self.logdets_b, axis=-1)
253
254         xf_avg = np.arange(len(yf_avg))
255         xb_avg = np.arange(len(yb_avg))
256
257         _ = ax1.plot(xf_avg, yf_avg, label='forward',
258                      color='r', **self.avg_kwargs)
259         _ = ax1.plot(xb_avg, yb_avg, label='backward',
260                      color='b', **self.avg_kwargs)
261
262         _ = ax2.plot(sumlogdet_xf_avg, sumlogdet_yf_avg, label='forward',
263                      color='r', lw=1.2, alpha=0.9, marker='.')
264
265         _ = ax2.plot(sumlogdet_xb_avg, sumlogdet_yb_avg, label='backward',
266                      color='b', lw=1.2, alpha=0.9, marker='.')
267
268         _ = ax1.set_xlabel('Leapfrog step', fontsize=16)
269         _ = ax1.set_ylabel(r'$\log|\mathcal{J}^{(t)}|$', fontsize=16)
270         _ = ax2.set_xlabel('MD step', fontsize=16)
271         _ = ax2.set_ylabel(r'$\sum_{t}\log|\mathcal{J}^{(t)}|$',
272                            fontsize=16)
273         _ = ax1.legend(loc='best', fontsize=10)
274         _ = ax2.legend(loc='best', fontsize=10)
275         _ = fig.tight_layout()
276         beta_str = str(beta).replace('.', '')
277         fn = f'avg_logdets_beta{beta_str}'
```

```
278         out_file = os.path.join(self.pdfs_dir, fn + '.pdf')
279         out_file_zoom = os.path.join(self.pdfs_dir, fn + '_zoom.pdf')
280         out_file_zoom1 = os.path.join(self.pdfs_dir, fn + '_zoom1.pdf')
281         io.log(f'Saving figure to: {out_file}')
282         _ = plt.savefig(out_file, dpi=400, bbox_inches='tight')
283 
284         lf_xlim = 100
285         md_xlim = lf_xlim // self.num_lf_steps
286 
287         _ = ax1.set_xlim((0, lf_xlim))
288         _ = ax2.set_xlim((0, md_xlim))
289         _ = plt.savefig(out_file_zoom, dpi=400, bbox_inches='tight')
290 
291         lf_xlim //= 4
292         md_xlim //= 4
293         _ = ax1.set_xlim((0, lf_xlim))
294         _ = ax2.set_xlim((0, md_xlim))
295         _ = plt.savefig(out_file_zoom1, dpi=400, bbox_inches='tight')
296 
297         return fig, (ax1, ax2)
```

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.'
)


# ============================================================================
#  * 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.\n (Default: 8)""")

    parser.add_argument("--time_size",
                        dest="time_size",
                        type=int,
                        default=8,
                        required=False,
                        help="""Temporal extent of lattice.\n (Default: 8)""")

    parser.add_argument("--link_type",
                        dest="link_type",
                        type=str,
                        required=False,
                        default='U1',
                        help="""Link type for gauge model.\n
                        (Default: 'U1')""")

    parser.add_argument("--dim",
                        dest="dim",
                        type=int,
                        required=False,
```

```
                        default=2,
                        help="""Dimensionality of lattice.\n (Default: 2)""")

    parser.add_argument("--num_samples",
                        dest="num_samples",
                        type=int,
                        default=128,
                        required=False,
                        help=("""Number of samples (batch size) to use for
                              training.\n (Default: 20)"""))

    parser.add_argument("--rand",
                        dest="rand",
                        action="store_true",
                        required=False,
                        help=("""If passed, set `rand=True` and start lattice
                              from randomized initial configuration.\n
                              (Default: `rand=False`, i.e. NOT passed.)"""))

    ##########################################################################
    #                          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. (Default: 5)"""))

    parser.add_argument("--eps",
                        dest="eps",
                        type=float,
                        default=0.1,
                        required=False,
                        help=("""Step size to use in leapfrog integrator.
                              (Default: 0.1)"""))

    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). (Default: 1.)"""))

    ##########################################################################
    #                       Learning rate parameters                         #
    ##########################################################################

    parser.add_argument("--lr_init",
                        dest="lr_init",
                        type=float,
                        default=1e-3,
                        required=False,
                        help=("""Initial value of learning rate.
                              (Default: 1e-3"""))

    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. (Default: 500)"""))

    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. (Default: 0.96)"""))

    ##########################################################################
    #                        Annealing rate parameters                       #
    ##########################################################################

    parser.add_argument("--fixed_beta",
                        dest="fixed_beta",
                        action="store_true",
                        required=False,
                        help=("""Flag that when passed runs the training loop
                              at fixed beta (i.e. no annealing is done).
                              (Default: `fixed_beta=False`)"""))

    parser.add_argument("--beta_init",
                        dest="beta_init",
                        type=float,
                        default=2.,
                        required=False,
                        help=("""Initial value of beta (inverse coupling
                              constant) used in gauge model when
                              annealing. (Default: 2.)"""))

    parser.add_argument("--beta_final",
                        dest="beta_final",
                        type=float,
                        default=5.,
                        required=False,
                        help=("""Final value of beta (inverse coupling
                              constant) used in gauge model when
                              annealing. (Default: 5."""))

    parser.add_argument("--warmup_lr",
                        dest="warmup_lr",
                        action="store_true",
                        required=False,
                        help=("""FLag that when passed will 'warmup' the
                              learning rate (i.e. gradually scale it up to the
                              value passed to `--lr_init` (performs better when
                              using Horovod for distributed training)."""))

    ##########################################################################
    #                         Training parameters                            #
    ##########################################################################

    parser.add_argument("--train_steps",
                        dest="train_steps",
                        type=int,
                        default=5000,
                        required=False,
                        help=("""Number of training steps to perform.
                              (Default: 5000)"""))

    parser.add_argument("--trace",
                        dest="trace",
                        action="store_true",
                        required=False,
                        help=("""Flag that when passed will set `--trace=True`,
                              and create a trace during training loop.
                              (Default: `--trace=False`, i.e.  not passed)"""))

    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.
```

```
                              (Default: 50)"""))

    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. (Default: 1)"""))

    parser.add_argument("--logging_steps",
                        dest="logging_steps",
                        type=int,
                        default=10,
                        required=False,
                        help=("""Number of steps after which to write logs for
                              tensorboard. (default: 50)"""))

    # ------------------------------------------------------------------------
    # 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'`.
                              (Default: 'conv3D')"""))

    parser.add_argument('--num_hidden1',
                        dest='num_hidden1',
                        type=int,
                        default=100,
                        required=False,
                        help=("""Number of nodes to include in each of the
                              fully-connected hidden layers for x, v, and t.
                              (Default: 100)"""))

    parser.add_argument('--num_hidden2',
                        dest='num_hidden2',
                        type=int,
                        default=100,
                        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.
                              (Default: None)"""))

    parser.add_argument('--zero_translation',
                        dest='zero_translation',
                        action='store_true',
                        required=False,
                        help=("""Flag that when passed explicitly sets the
                              translation function (T in the original paper) to
                              be zero. (Default: False)"""))

    parser.add_argument('--no_summaries',
                        dest="no_summaries",
                        action="store_true",
                        required=False,
                        help=("""FLag that when passed will prevent tensorflow
                              from creating tensorboard summary objects."""))

    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_steps",
                        dest="run_steps",
                        type=int,
                        default=10000,
                        required=False,
                        help=("""Number of evaluation 'run' steps to perform
                              after training (i.e. length of desired chain
                              generate using trained L2HMC sample).
                              (Default: 10000)"""))

    parser.add_argument("--eps_fixed",
                        dest="eps_fixed",
                        action="store_true",
                        required=False,
                        help=("""Flag that when passed will cause the step size
                              `eps` to be a fixed (non-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("--nnehmc_loss",
                        dest="nnehmc_loss",
                        action="store_true",
                        required=False,
                        help=("""Flag that when passed will calculate the
                              'NNEHMC Loss' from
                              (https://infoscience.epfl.ch/record/264887/files/robust_parameter_estimation.pdf)
                              (Default: False)."""))

    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.
                              (Default: 1.)"""))

    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.
                              (Default: 1.)"""))

    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. (Default: 1.)"""))

    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("--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("--comet",
                        dest="comet",
                        action="store_true",
                        required=False,
                        help=("""Flag that when passed uses comet.ml for
                              parameter logging and additonal metric
                              tracking/displaying."""))

    parser.add_argument("--dropout_prob",
                        dest="dropout_prob",
                        type=float,
                        required=False,
                        default=0.,
                        help=("""Dropout probability in network. If > 0,
                              dropout will be used. (Default: 0.)"""))

    ########################
    #  (Mostly) Deprecated  #
    ########################

    parser.add_argument('--save_samples',
                        dest='save_samples',
                        action='store_true',
                        required=False,
                        help=("""Flag that when passed will set
                              `--save_samples=True`, and save the samples
                              generated during the `run` phase.
                              (Default: `--save_samples=False, i.e.
                              `--save_samples` is not passed).\n
                              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("--clip_value",
                        dest="clip_value",
                        type=float,
                        default=0.,
                        required=False,
                        help=("""Clip value, used for clipping value of
                              gradients by global norm. (Default: 0.) If a
```

```
                            value greater than 0. is passed, gradient
                            clipping will be performed."""))

    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("--theta",
                        dest="theta",
                        action="store_true",
                        required=False,
                        help=("""Flag that when passed indicates we're training
                              on theta @ ALCf."""))

    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("--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/parse_inference_args.py

```
"""
parse_inference_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 = 'Run inference on trained L2HMC model.'


# ============================================================================
#  * 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('--params_file',
                        dest='params_file',
                        required=False,
                        default=None,
                        help=("""Path to `params.pkl` or `parameters.pkl` file
                              containing the model parameters needed to run
                              inference."""))

    parser.add_argument("--run_steps",
                        dest="run_steps",
                        type=int,
                        default=5000,
                        required=False,
                        help=("""Number of evaluation 'run' steps to perform
                              after training (i.e. length of desired chain
                              generate using trained L2HMC sample).
                              (Default: 5000)"""))

    parser.add_argument("--beta_inference",
                        dest="beta_inference",
                        type=float,
                        default=None,
                        required=False,
                        help=("""Flag specifying a singular value of beta at
                              which to run inference using the trained
                              L2HMC sampler. (Default: None"""))

    parser.add_argument('--plot_lf',
```

```
                        dest='plot_lf',
                        action='store_true',
                        required=False,
                        help=("""Flag that when passed will set
                              `--plot_lf=True`, and will plot the 'metric'
                              distance between subsequent configurations, as
                              well as the determinant of the Jacobian of the
                              transformation for each individual leapfrog step,
                              as well as each molecular dynamics step (with
                              Metrpolis-Hastings accept/reject).\n
                              When plotting the determinant of the Jacobian
                              following the MD update, we actually calculate
                              the sum of the determinants from each individual
                              LF step since this is the quantity that actually
                              enters into the MH acceptance probability.
                              (Default: `--plot_lf=False`, i.e. `--plot_lf` is
                              not passed mostly just beause the plots are
                              extremely large (many LF steps during inference)
                              and take a while to actually generate.)"""))

    parser.add_argument('--loop_net_weights',
                        dest='loop_net_weights',
                        action='store_true',
                        required=False,
                        help=("""Flag that when passed sets
                              `--loop_net_weights=True`, and 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.
                              (Default: `--loop_net_weights=False, i.e.
                              `--loop_net_weights is not passed)"""))

    parser.add_argument('--loop_transl_weights',
                        dest='loop_transl_weights',
                        action='store_true',
                        required=False,
                        help=("""Flag that when passed will loop over different
                              values for the `translation_weight` in
                              `net_weights`, which is believed to be causing
                              the discrepancy between the observed and expected
                              value of the average plaquette when running
                              inference."""))

    parser.add_argument('--save_samples',
                        dest='save_samples',
                        action='store_true',
                        required=False,
                        help=("""Flag that when passed will set
                              `--save_samples=True`, and save the samples
                              generated during the `run` phase.
                              (Default: `--save_samples=False, i.e.
                              `--save_samples` is not passed).\n
                              WARNING!! This is very data intensive."""))

    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.20 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 config 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 check_else_make_dir(d):
    """If directory `d` doesn't exist, it is created.

    Args:
        d (str): Location where directory should be created if it doesn't
            already exist.
    """
    if not os.path.isdir(d):
        log(f"Creating directory: {d}")
        os.makedirs(d, exist_ok=True)

```

```python

def make_dirs(dirs):
    """Make directories if and only if hvd.rank == 0."""
    _ = [check_else_make_dir(d) for d in dirs]


def _parse_flags(FLAGS):
    """Helper method for parsing flags as both AttrDicts or generic dicts."""
    if isinstance(FLAGS, dict):
        flags_dict = FLAGS
    else:
        try:
            flags_dict = FLAGS.__dict__
        except (NameError, AttributeError):
            pass
    #  if isinstance(FLAGS, dict):
    try:
        LX = flags_dict['space_size']
        NS = flags_dict['num_samples']
        LF = flags_dict['num_steps']
        SS = flags_dict['eps']
        QW = flags_dict['charge_weight']
        NA = flags_dict['network_arch']
        BN = flags_dict['use_bn']
        DP = flags_dict['dropout_prob']
        AW = flags_dict['aux_weight']
        hmc = flags_dict['hmc']
        try:
            _log_dir = flags_dict['log_dir']
        except KeyError:
            _log_dir = ''

    except (NameError, AttributeError):
        LX = FLAGS.space_size
        NS = FLAGS.num_samples
        LF = FLAGS.num_steps
        SS = FLAGS.eps
        QW = FLAGS.charge_weight
        NA = FLAGS.network_arch
        BN = FLAGS.use_bn
        DP = FLAGS.dropout_prob
        AW = FLAGS.aux_weight
        hmc = FLAGS.hmc
        try:
            _log_dir = FLAGS.log_dir
        except AttributeError:
            _log_dir = ''

    out_dict = {
        'LX': LX,
        'NS': NS,
        'LF': LF,
        'SS': SS,
        'QW': QW,
        'NA': NA,
        'BN': BN,
        'DP': DP,
        'AW': AW,
        'hmc': hmc,
        '_log_dir': _log_dir
    }

    return out_dict


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):
```

```
138
139             `lattice{LX}_batch{NS}_lf{LF}_eps{SS}_qw{QW}`
140
141         Returns:
142             FLAGS, with FLAGS.log_dir being equal to the newly created log_dir.
143
144         NOTE: If log_dir does not already exist, it is created.
145         """
146         flags_dict = _parse_flags(FLAGS)
147         LX = flags_dict['LX']
148         NS = flags_dict['NS']
149         LF = flags_dict['LF']
150         SS = flags_dict['SS']
151         QW = flags_dict['QW']
152         NA = flags_dict['NA']
153         BN = flags_dict['BN']
154         DP = flags_dict['DP']
155         AW = flags_dict['AW']
156         _log_dir = flags_dict['_log_dir']
157
158         aw = str(AW).replace('.', '')
159         qw = str(QW).replace('.', '')
160         dp = str(DP).replace('.', '')
161
162         if flags_dict['hmc']:
163             run_str = f'HMC_lattice{LX}_batch{NS}_lf{LF}_eps{SS:.3g}'
164         else:
165             run_str = f'lattice{LX}_batch{NS}_lf{LF}_qw{qw}_aw{aw}_{NA}_dp{dp}'
166             if BN:
167                 run_str += '_bn'
168
169         now = datetime.datetime.now()
170         #  print(now.strftime("%b %d %Y %H:%M:%S"))
171         day_str = now.strftime('%Y_%m_%d')
172         time_str = now.strftime("%Y_%m_%d_%H%M")
173
174         project_dir = os.path.abspath(os.path.dirname(FILE_PATH))
175         #  if FLAGS.log_dir is None:
176         if _log_dir is None:
177             if root_dir is None:
178                 _dir = 'logs'
179             else:
180                 _dir = root_dir
181
182         else:
183             if root_dir is None:
184                 #  _dir = FLAGS.log_dir
185                 _dir = _log_dir
186             else:
187                 _dir = os.path.join(_log_dir, root_dir)
188         root_log_dir = os.path.join(project_dir, _dir, day_str, time_str, run_str)
189         check_else_make_dir(root_log_dir)
190         if any('run_' in i for i in os.listdir(root_log_dir)):
191             run_num = get_run_num(root_log_dir)
192             log_dir = os.path.abspath(os.path.join(root_log_dir,
193                                                    f'run_{run_num}'))
194         else:
195             log_dir = root_log_dir
196         if log_file is not None:
197             write(f'Output saved to: \n\t{log_dir}', log_file, 'a')
198             write(80*'-', log_file, 'a')
199
200         return log_dir
201
202
203     def _list_and_join(d):
204         """For each dir `dd` in `d`, return a list of paths ['d/dd1', ...]"""
205         contents = [os.path.join(d, i) for i in os.listdir(d)]
206         paths = [i for i in contents if os.path.isdir(i)]
207
```

```
    return paths


def list_and_join(d):
    """Deal with the case of `d` containing multiple directories."""
    if isinstance(d, (list, np.ndarray)):
        paths = []
        for dd in d:
            _path = _list_and_join(dd)[0]
            paths.append(_path)
    else:
        paths = _list_and_join(d)

    return paths


def save_data(data, out_file, name=None):
    """Save data to out_file using either pickle.dump or np.save."""
    if os.path.isfile(out_file):
        log(f"WARNING: File {out_file} already exists...")
        tmp = out_file.split('.')
        out_file = tmp[0] + '_1' + f'.{tmp[1]}'

    log(f"Saving {name} to {out_file}...")
    if out_file.endswith('.pkl'):
        with open(out_file, 'wb') as f:
            pickle.dump(data, f)

    elif out_file.endswith('.npy'):
        np.save(out_file, np.array(data))

    else:
        log("Extension not recognized! out_file must end in .pkl or .npy")


def save_params(params, out_dir):
    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)


def save_params_to_pkl_file(params, out_dir):
    """Save `params` dictionary to `parameters.pkl` in `out_dir.`"""
    check_else_make_dir(out_dir)
    params_file = os.path.join(out_dir, 'parameters.pkl')
    #  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

```

```
278     return run_num
279
280
281 def get_eps_from_run_history_txt_file(txt_file):
282     """Parse `run_history.txt` file and return `eps` (step size)."""
283     with open(txt_file, 'r') as f:
284         data_line = [f.readline() for _ in range(10)][-1]
285     eps = float([i for i in data_line.split(' ') if i != ''][3])
286
287     return eps
```

A.21 l2hmc-qcd/loggers/summary_utils.py

```python
"""
summary_utils.py

Collection of helper methods for creating various summary objects for logging
data in TensorBoard.

Author: Sam Foreman (github: @saforem2)
Date: 08/16/2019
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import tensorflow as tf
from lattice.lattice import u1_plaq_exact_tf


def grad_norm_summary(name_scope, grad):
    with tf.name_scope(name_scope + '_gradients'):
        grad_norm = tf.sqrt(tf.reduce_mean(grad ** 2))
        summary_name = name_scope + '_grad_norm'
        tf.summary.scalar(summary_name, grad_norm)


def check_var_and_op(name, var):
    return (name in var.name or name in var.op.name)


def variable_summaries(var, name=''):
    """Attach a lot of summaries to a Tensor (for TensorBoard visualization)"""
    mean_name = 'mean'
    stddev_name = 'stddev'
    max_name = 'max'
    min_name = 'min'
    hist_name = 'histogram'
    if name != '':
        mean_name = name + '/' + mean_name
        stddev_name = name + '/' + stddev_name
        max_name = name + '/' + max_name
        min_name = name + '/' + min_name
        hist_name = name + '/' + hist_name

    mean = tf.reduce_mean(var)
    tf.summary.scalar(mean_name, mean)
    with tf.name_scope('stddev'):
        stddev = tf.sqrt(tf.reduce_mean(tf.square(var - mean)))
    tf.summary.scalar(stddev_name, stddev)
    tf.summary.scalar(max_name, tf.reduce_max(var))
    tf.summary.scalar(min_name, tf.reduce_min(var))
    tf.summary.histogram(hist_name, var)
    # activation summaries
    #  tf.summary.histogram(tensor_name + '/activations', x)
    #  tf.summary.scalar(tensor_name + '/sparsity', tf.nn.zero_fraction(x))


def add_loss_summaries(total_loss):
    """Add summaries for losses in GaugeModel.

    Generates a moving average for all losses and associated summaries for
    visualizing the performance of the network.

    Args:
        total_loss: Total loss from model._calc_loss()

    Returns:
        loss_averages_op: Op for generating moving averages of losses.
    """
```

```
    # Compute the moving average of all individual losses and the total
    # 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, l)
        tf.summary.scalar(l.op.name + 'moving_avg', loss_averages.average(l))

    return loss_averages_op


def make_summaries_from_collection(collection, names):
    try:
        for op, name in zip(tf.get_collection(collection), names):
            variable_summaries(op, name)
    except AttributeError:
        pass


def _create_loss_summaries(total_loss):
    """Add summaries for losses in GaugeModel.

    Generates a moving average for all losses and associated summaries for
    visualizing the performance of the network.

    Args:
        total_loss: Total loss operation.
    Returns:
        loss_averages_op: Operation for generating moving averages of losses.
    """
    # Compute the moving avg. of all individ. losses and total loss
    loss_averages = tf.train.ExponentialMovingAverage(0.9, name='avg')
    losses = tf.get_collection('losses')
    loss_averages_op = loss_averages.apply(losses + [total_loss])

    std_names = ['std_loss', 'x_std_loss', 'z_std_loss']
    std_losses = losses[:3]
    for name, loss in zip(std_names, std_losses):
        with tf.name_scope(name):
            tf.summary.scalar(name + '/raw', loss)
            tf.summary.scalar(name + '/moving_avg',
                              loss_averages.average(loss))

    charge_names = ['charge_loss', 'xq_loss', 'zq_loss']
    charge_losses = losses[3:]
    for name, loss in zip(charge_names, charge_losses):
        with tf.name_scope(name):
            tf.summary.scalar(name + '/raw', loss)
            tf.summary.scalar(name + '/moving_avg',
                              loss_averages.average(loss))

    with tf.name_scope('total_loss'):
        tf.summary.scalar('total_loss/raw', total_loss)
        tf.summary.scalar('total_loss/moving_avg',
                          loss_averages.average(total_loss))

    return loss_averages_op


def _create_training_summaries(model):
    """Create summary objects for training operations in TensorBoard."""
    with tf.name_scope('loss'):
        tf.summary.scalar('loss', model.loss_op)

```

```
    with tf.name_scope('learning_rate'):
        tf.summary.scalar('learning_rate', model.lr)

    with tf.name_scope('step_size'):
        tf.summary.scalar('step_size', model.dynamics.eps)


def _create_grad_norm_summaries(grad, var):
    """Create grad_norm summaries."""
    if 'XNet' in var.name:
        net_str = 'XNet/'
    elif 'VNet' in var.name:
        net_str = 'VNet/'
    else:
        net_str = ''
    with tf.name_scope(net_str):
        if 'scale' in var.name:
            grad_norm_summary(net_str + 'scale', grad)
            tf.summary.histogram(net_str + 'scale', grad)
        if 'transf' in var.name:
            grad_norm_summary(net_str + 'transformation', grad)
            tf.summary.histogram(net_str + 'transformation', grad)
        if 'transl' in var.name:
            grad_norm_summary(net_str + 'translation', grad)
            tf.summary.histogram(net_str + 'translation', grad)


def _create_pair_summaries(grad, var):
    """Create summary objects for a gradient, variable pair."""
    try:
        _name = var.name.split('/')[-2:]
        if len(_name) > 1:
            name = _name[0] + '/' + _name[1][:-2]
        else:
            name = var.name[:-2]
    except (AttributeError, IndexError):
        name = var.name[:-2]

    with tf.name_scope(name):
        var_name = var.name.replace(':', '')
        variable_summaries(var, name=var_name)

    grad_name = name + '/gradient'
    with tf.name_scope(grad_name):
        variable_summaries(grad, name=grad_name)


def _create_obs_summaries(model):
    """Create summary objects for physical observables."""
    with tf.name_scope('avg_charge_diffs'):
        tf.summary.scalar('avg_charge_diffs',
                          tf.reduce_mean(model.charge_diffs_op))

    with tf.name_scope('avg_plaq'):
        tf.summary.scalar('avg_plaq', model.avg_plaqs_op)

    with tf.name_scope('avg_plaq_diff'):
        tf.summary.scalar('avg_plaq_diff',
                          (u1_plaq_exact_tf(model.beta)
                           - model.avg_plaqs_op))


def _create_md_summaries(model):
    """Create summary objects for each of the MD functions and outputs."""
    for k1, v1 in model.l2hmc_fns['out_fns_f'].items():
        for k2, v2 in v1.items():
            with tf.name_scope(f'{k1}_fn_{k2}_f'):
                variable_summaries(v2)

    for k1, v1 in model.l2hmc_fns['out_fns_b'].items():
```

```
        for k2, v2 in v1.items():
            with tf.name_scope(f'{k1}_fn_{k2}_b'):
                variable_summaries(v2)

    with tf.name_scope('lf_out_f'):
        variable_summaries(model.lf_out_f)

    with tf.name_scope('lf_out_b'):
        variable_summaries(model.lf_out_b)


def create_summaries(model, summary_dir, training=True):
    """Create summary objects for logging in TensorBoard."""
    summary_writer = tf.contrib.summary.create_file_writer(summary_dir)

    grads_and_vars = zip(model.grads,
                         model.dynamics.trainable_variables)

    if training:
        _create_training_summaries(model)

    _create_obs_summaries(model)
    _create_md_summaries(model)
    #  _ = _create_loss_summaries(model.loss_op)

    for grad, var in grads_and_vars:
        _create_pair_summaries(grad, var)

        if 'kernel' in var.name:
            _create_grad_norm_summaries(grad, var)

    summary_op = tf.summary.merge_all(name='train_summary_op')

    return summary_writer, summary_op
```

A.22 l2hmc-qcd/utils/data_loader.py

```
"""
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)
```

```
138
139     def load_observables(self, run_dir=None):
140         if run_dir is None:
141             run_dir = self.run_dir
142
143         obs_dir = os.path.join(run_dir, 'observables')
144         files = os.listdir(obs_dir)
145
146         obs_names = [i.rstrip('.pkl') for i in files]
147         obs_files = [os.path.join(obs_dir, i) for i in files]
148
149         observables = {
150             n: self.load_pkl_file(f) for (n, f) in zip(obs_names, obs_files)
151         }
152
153         return observables
```

B | Appendix: Systematic Debugging Results of the Average Plaquette

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

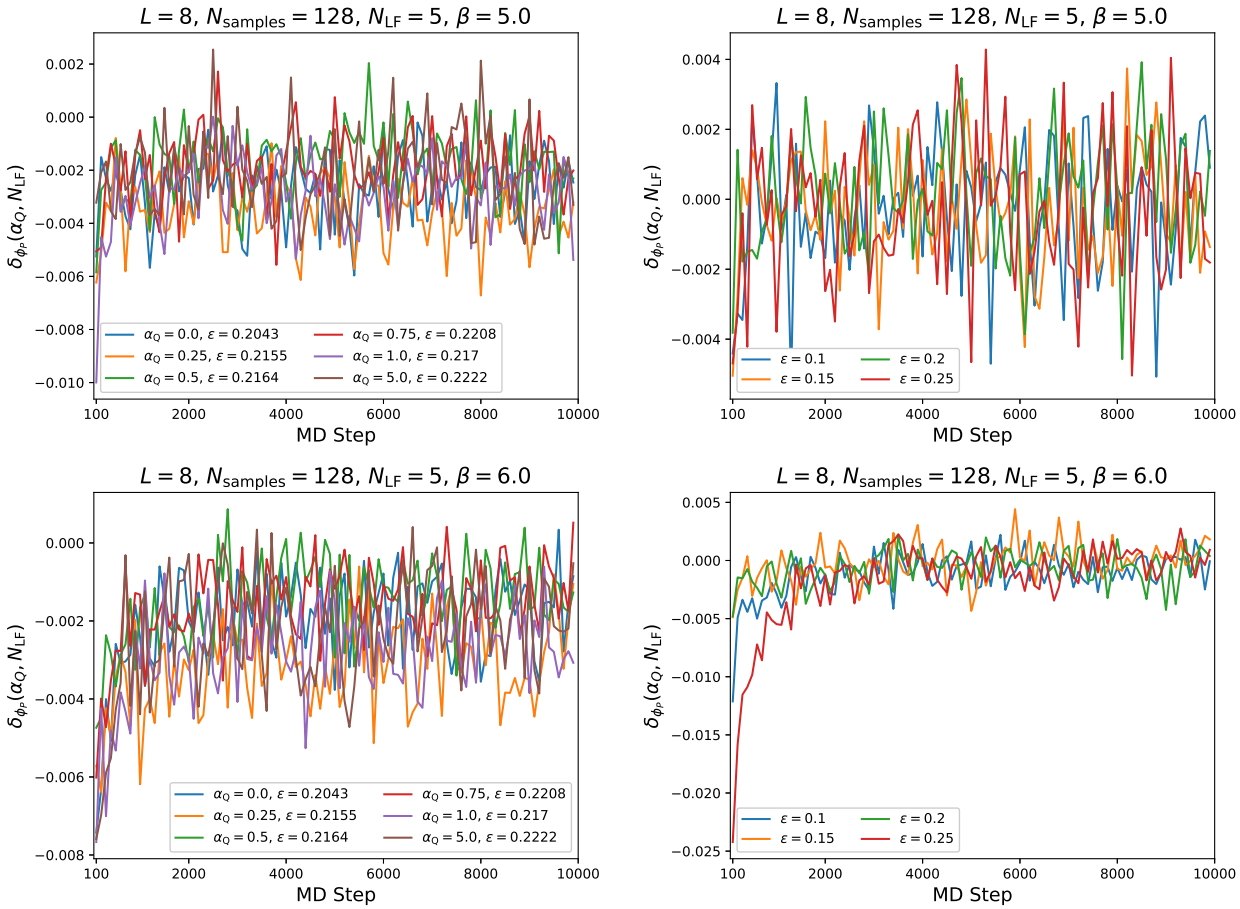


Figure B.1: δ_{ϕ_P} vs MD step with $N_{\text{LF}} = 5$ for $\beta = 5.0$ (top row) and $\beta = 6.0$ (bottom row). The results from the trained L2HMC (generic HMC) sampler are shown in the left (right) column. As can be seen, the difference δ_{ϕ_p} remains roughly consistent for all values of α_Q.

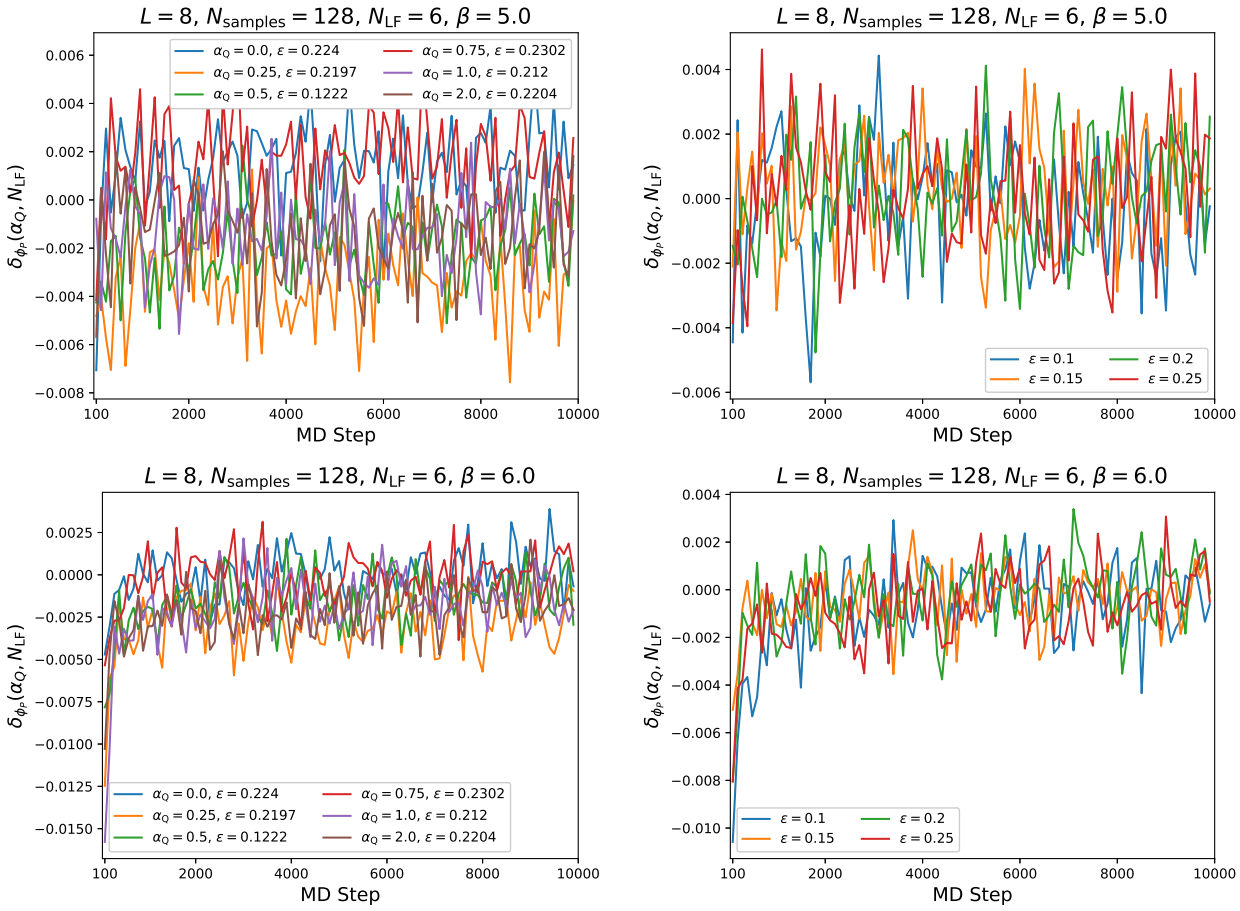


Figure B.2: δ_{ϕ_P} vs MD step with $N_{\text{LF}} = 5$ for $\beta = 5.0$ (top row) and $\beta = 6.0$ (bottom row). The results from the trained L2HMC (generic HMC) sampler are shown in the left (right) column. As can be seen, the difference δ_{ϕ_P} remains roughly consistent for all values of α_Q.

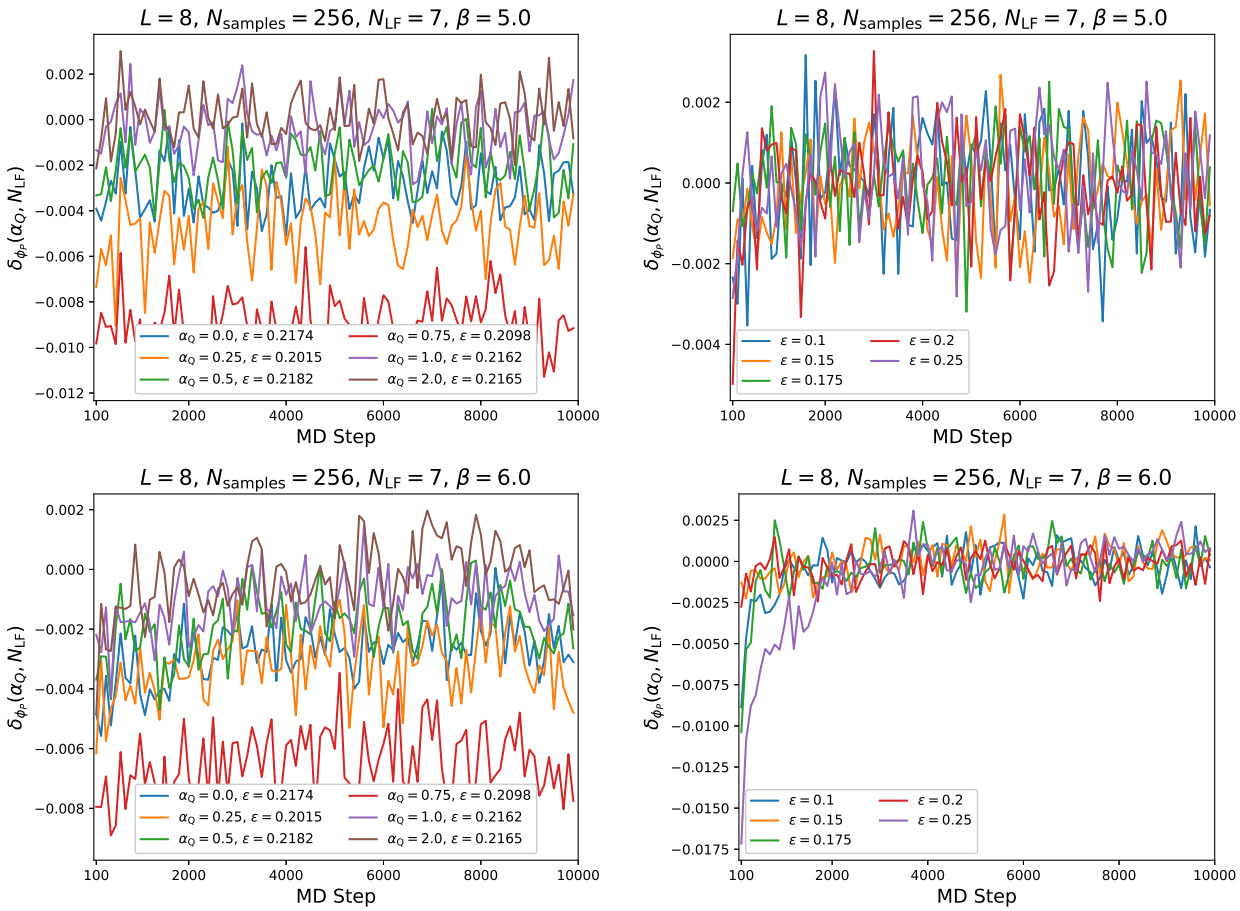


Figure B.3: δ_{ϕ_P} vs MD step with $N_{\text{LF}} = 7$ As can be seen, the difference δ_{ϕ_P} is noticeably larger for $\alpha_Q = 0.75$, but remains roughly consistent for all other values of α_Q.

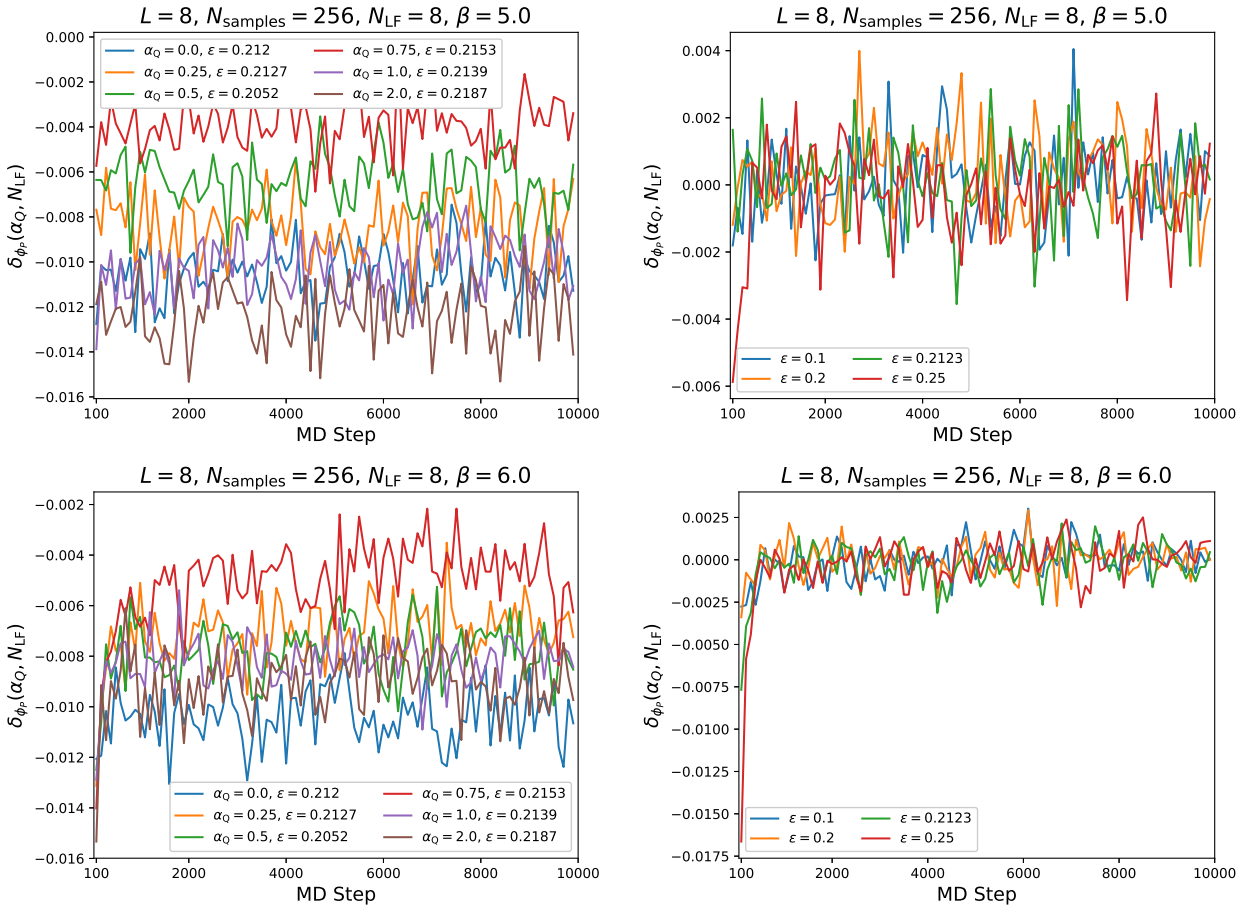


Figure B.4: δ_{ϕ_P} vs. MD step with $N_{\text{LF}} = 8$. As can be seen, the difference δ_{ϕ_P} remains roughly consistent for all values of α_Q.

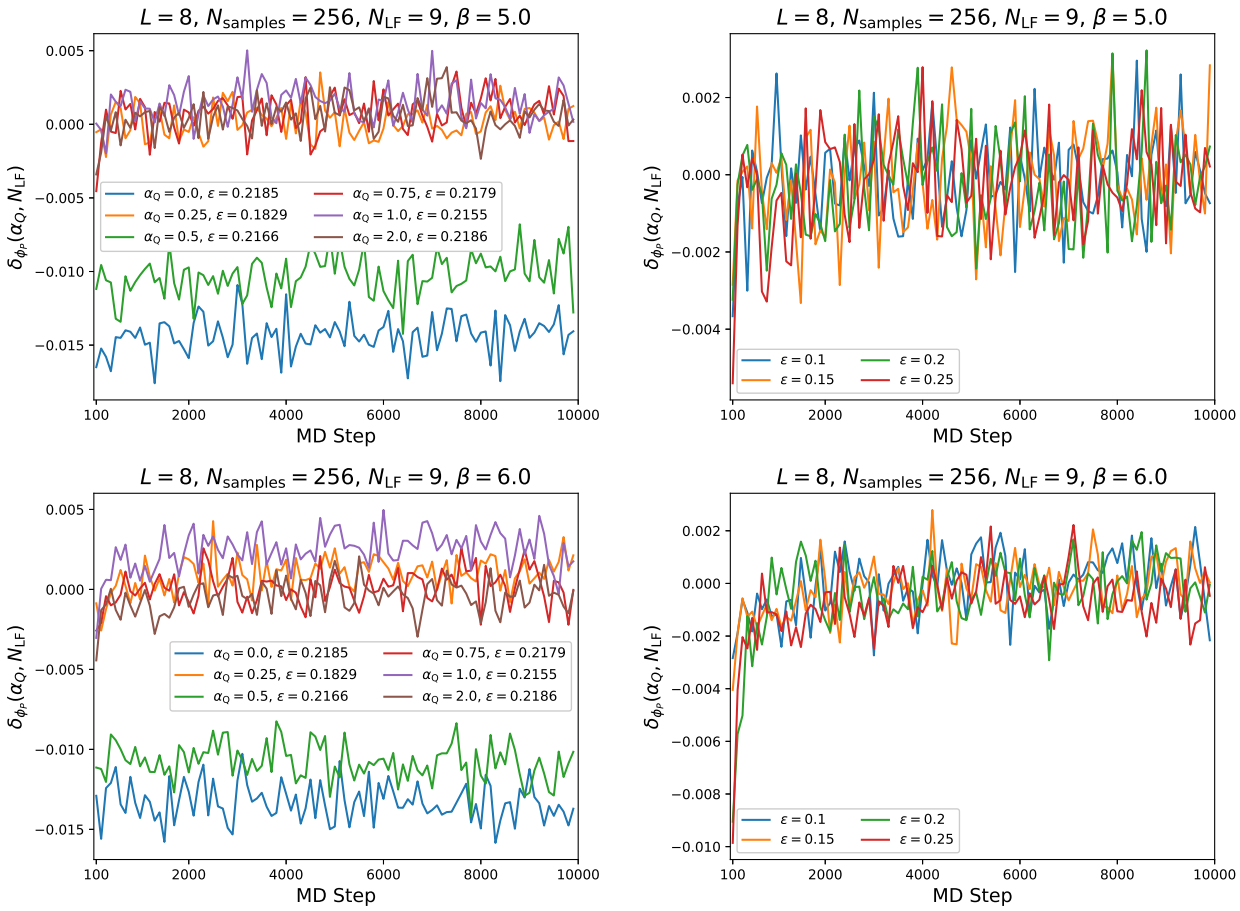


Figure B.5: δ_{ϕ_P} vs MD step with $N_{\text{LF}} = 9$. As can be seen, the difference δ_{ϕ_P} is largest for $\alpha_Q = 0.0$ and $\alpha_Q = 0.5$.

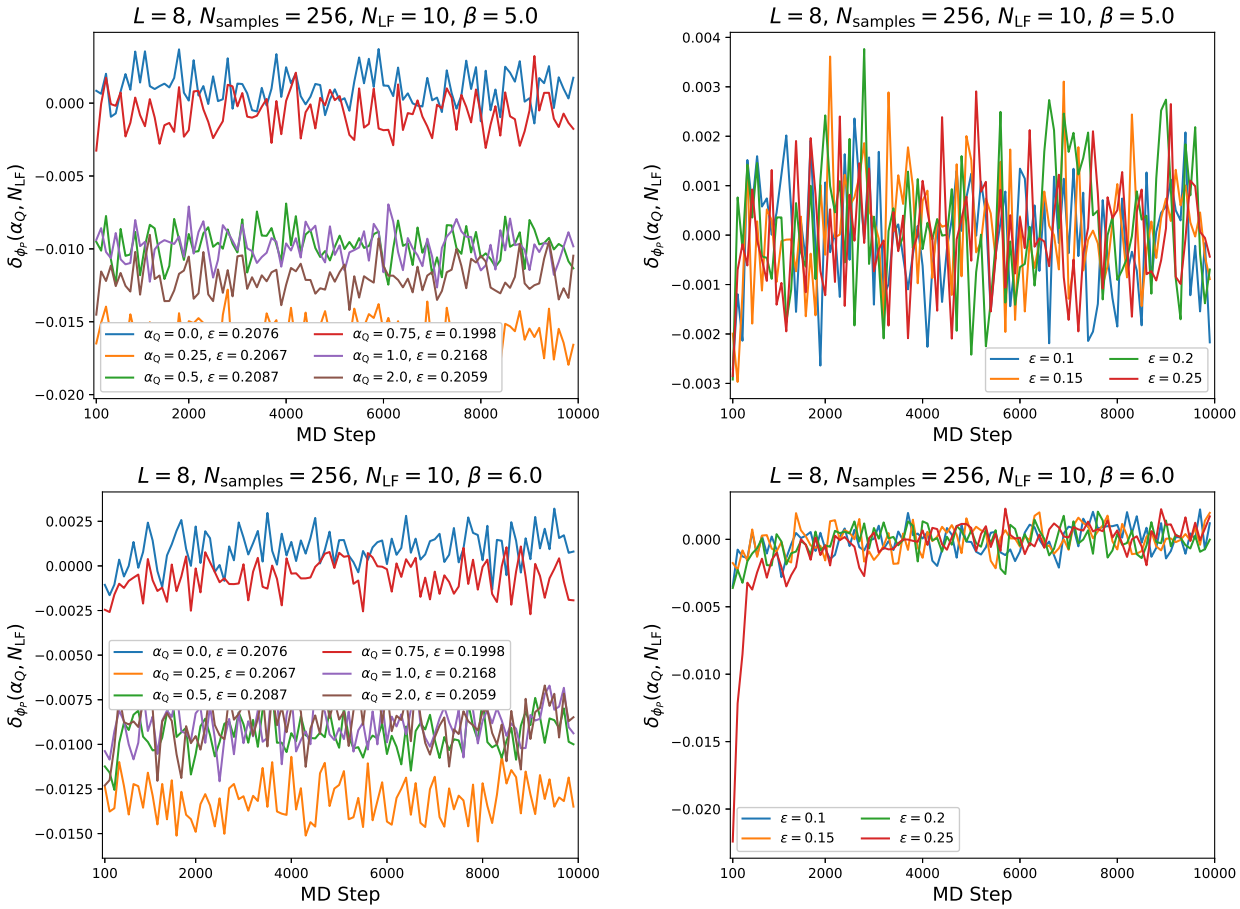


Figure B.6: δ_{ϕ_P} vs MD step with $N_{\text{LF}} = 10$. As can be seen, the difference δ_{ϕ_P} is smallest for $\alpha_Q = 0.,\ 0.75$ and largest for $\alpha_Q = 0.25$, while $\alpha_Q = 0.5,\ 2.0$, takes on intermediate values.

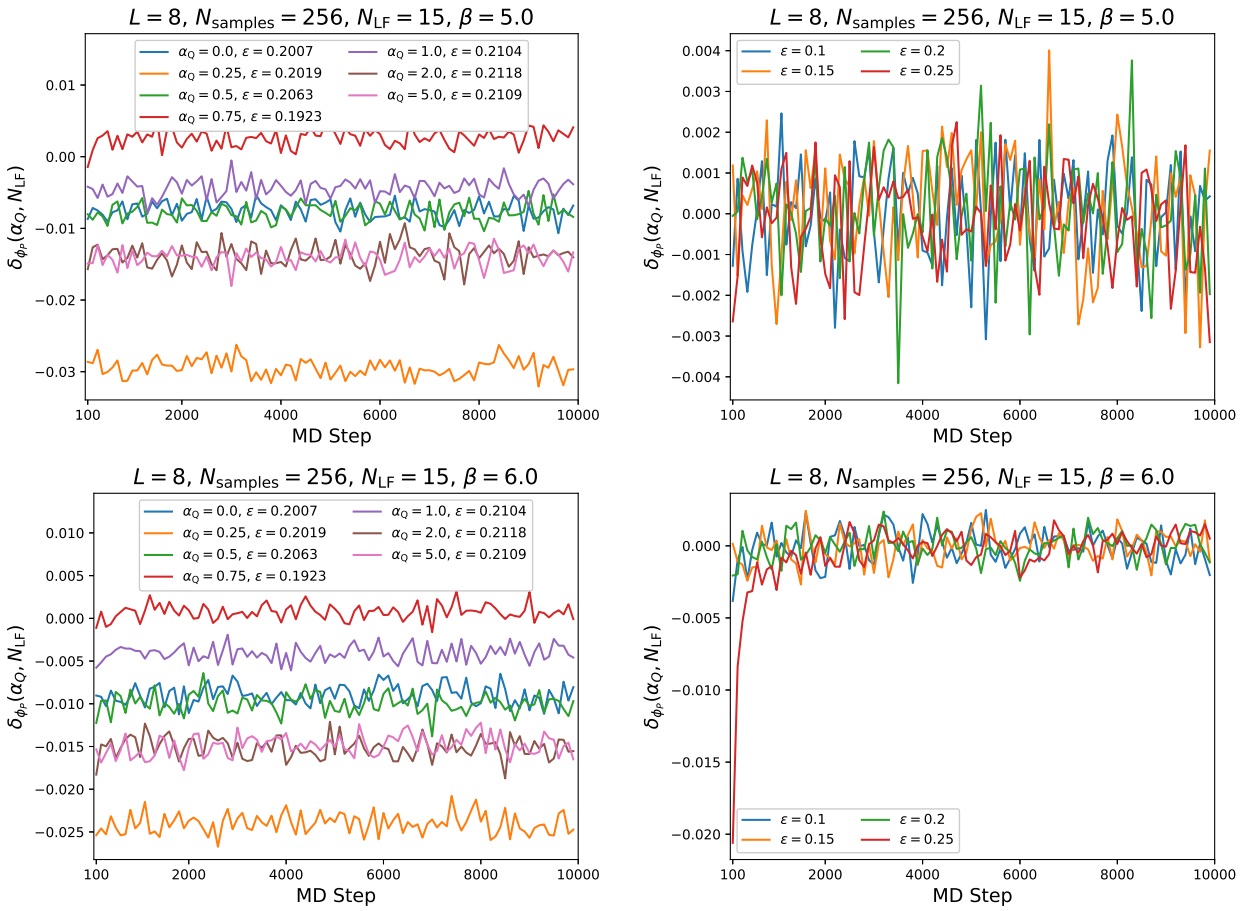


Figure B.7: δ_{ϕ_P} vs MD step with $N_{\text{LF}} = 15$. As can be seen, the difference δ_{ϕ_P} varies for different values of α_Q.

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).