Bayesian neural network estimation of next-to-leading-order cross sections

by

René Alexander Ask

THESIS

for the degree of

MASTER OF SCIENCE

Faculty of Mathematics and Natural Sciences
University of Oslo

Spring 2022

Bayesian neural network estimation of next-to-leading-order cross sections

René Alexander Ask

Abstract

This is my abstract.

Acknowledgments

Acknowledgments yo

Contents

Introduction

Motivation, context and problem.

Outline of the Thesis

Give outline of thesis

Chapter 1

The Physics Problem

Chapter 2

Machine Learning: Preliminaries

Machine learning is a field of study concerned with learning from known observations and prediction of unseen ones. In this thesis, we'll focus on *supervised* machine learning, which is a subfield of machine learning that fits models on data points x with definite targets y. We will confine ourselves even further and only study *regression* problems, which is a class of problems where the function we are trying to learn produces a continuous output, i.e a function $f : \mathbb{R}^p \to \mathbb{R}^d$.

2.1 Basic Concepts in Regression

The basic conceptual framework of a supervised machine learning problem is as follows. Assume a dataset D is a sequence of n datapoints $D = \{(x_i, y_i)\}_{i=1}^n$, where $x_i \in \mathbb{R}^p$ is the set of *features* and $y_i \in \mathbb{R}^d$ is the *target*. The next ingredient is to assume the targets are of the form

$$y_i = f(x_i) + \epsilon_i, \tag{2.1}$$

for some true function $f(x_i)$ (also known as the ground truth), where ϵ_i is introduced to account for random noise. To approximate the outputs y_i, the standard approach is to choose a model class $\hat{f}(x;\theta)$ combined with a procedure to choose parameters θ such that the model is as close to $f(x_i)$ as possible. This typically involves choosing a *metric* $\mathcal{L}$ to quantify the error, usually called a *loss* function (or a *cost* function, but we will adopt the former term in line with the terminology used in the TensorFlow framework), and minimize it with respect to the parameters of the model. The output of the model is usually denoted as

$$\hat{y}_i = \hat{f}(x_i;\theta), \tag{2.2}$$

for brevity.

2.1.1 Bias-Variance Trade-Off

From eq. (2.1), we can deduce a general feature of machine learning problems that proves challenging. We cannot directly probe the true function $f(x)$, because only $y = f(x) + \epsilon$ is observed. Because of this, choosing a model class is a delicate process in classical machine learning. If the model class is too simple (i.e few parameters θ), it is likely to capture very general features of the ground truth whilst more nuanced properties are missed entirely. Then we say that the model has a high bias and a low variance. Increasing the model complexity (i.e increasing number of parameters) allows the model to reproduce a growing number of nook-and-crannies of the data. A model that is too complex is said to have a low bias and a high variance. Finally, there is one last aspect that influences the choice of model class, and that is the size of the dataset. If it is small, a simpler model class is chosen because the data may not be particularly representative of the true underlying process. In a sense, there may occur fluctuations which would simply average out once more data is collected. Thus one opts for a simpler model class if the size of the dataset is small. From a Bayesian perspective, this is absurd, because we are implicitly assuming there is a true underlying process we want to learn. If the process is complex, then the model class should reflect this. Luckily, because Bayesian methods provides a natural way to assign uncertainty to a prediction, we can choose our model class according to how complex we think the process is, independently of how large the dataset is.

2.2 Loss Functions

For regression problems, two loss functions $\mathcal{L}$ are commonly chosen. The first is the *residual squared error* (RSS) given by

$$\text{RSS} = \sum_{i=1}^{n} \|\hat{y}_i - y_i\|_2^2, \tag{2.3}$$

where $\|\cdot\|_2$ denotes the L^2-norm. The second is the the *mean squared error* (MSE), given by

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n} \|\hat{y}_i - y_i\|_2^2. \tag{2.4}$$

For optimization purposes, they yield equivalent optimal parameters θ.

2.2.1 Regularization

With datasets of limited size, overfitting typically pose a problem yielding models that generalize poorly. One strategy to overcome this, is to tack on a regularization term to the loss-function. By *regularization*, we mean an additional term that limits the size of the allowed parameter space. The two most common ones are L^2-regularization, which adds a term to the loss function as

$$\mathcal{L} + \lambda\|\theta\|_2^2, \tag{2.5}$$

where λ is the so-called *regularization strength.* The second is L^1-regularization, which yields a loss

$$\mathcal{L} + \lambda\|\theta\|_1. \tag{2.6}$$

The terms *penalizes* large values of θ, effectively shrinking the allowed parameter space. The larger the value of the regularization strength λ, the smaller the allowed parameter space becomes.

2.3 Optimization

Once a model class and loss function is chosen, and an *optimizer* must be chosen. In this section, we will study several optimization schemes, with the ultimate goal of defining the state-of-the-art optimization in modern machine learning, namely ADAM.

2.3.1 Gradient Descent

Gradient descent is the most basic optimization scheme. The update rule for the parameters is given by

$$\theta_{t+1} = \theta_t - \eta_t \sum_{i=1}^{n} \nabla_\theta \mathcal{L}(\hat{f}(x_i;\theta_t), y_i), \tag{2.7}$$

where θ_t is the model parameters at iteration t and η_t is the *learning rate*, which in general is dependent on iteration t, hence the subscript.

2.3.2 Stochastic Gradient Descent

The standard gradient descent (SGD) algorithm has an inherent weakness in the sense that it computes the gradient using the whole dataset at each iteration. Stochastic gradient descent improves upon this algorithm by dividing the dataset into a set of *batches* B, each of which is a subset of the complete dataset. The parameter update is then performed using a randomly chosen batch $B_j \in B$ as follows:

$$\theta_{t+1} = \theta_t - \eta_t \sum_{(x_i, y_i) \in B_j} \nabla_\theta \mathcal{L}(\hat{f}(x_i; \theta_t), y_i). \tag{2.8}$$

An iteration over all batches $B_j \in B$ is called an *epoch*. To simplify notation somewhat, we introduce the notation

$$\nabla_\theta \mathcal{L}^B \equiv \sum_{(x_i, y_i) \in B_j} \nabla_\theta \mathcal{L}(\hat{f}(x_i; \theta_t), y_i). \tag{2.9}$$

Then the update rule for SGD can be recast as

$$\theta_{t+1} = \theta_t - \eta_t \nabla_\theta \mathcal{L}^B \tag{2.10}$$

2.3.3 Gradient Descent with Momentum

Stochastic gradient descent is usually accompanied by a so-called *momentum* term to compensate for random fluctuations that may occur when computing gradients on subsets of the full dataset. The momentum term stores a running average of previous gradients which yields a general direction in which the gradient points in parameter space. This helps the optimization process converge faster to a region of parameter space in which a minimum exist. Let v_t be defined by the recursive equation

$$v_t = \gamma v_{t-1} + \eta_t \nabla_\theta \mathcal{L}^B. \tag{2.11}$$

Then the update rule for the parameters is

$$\theta_{t+1} = \theta_t - v_t. \tag{2.12}$$

2.3.4 RMSprop

In RMSprop, we not only keep a running average of the first-order moment (the momentum), but we also store a running average of the second moment of the gradient. Let $s_t \equiv \langle g_t^2 \rangle$ be the running average of g_t, which is the gradient at iteration t. The update rule is then given by

$$\begin{aligned} g_t &= \nabla_\theta \mathcal{L}^B \\ s_t &= \beta s_{t-1} + (1-\beta) g_t^2 \\ \theta_{t+1} &= \theta_t - \eta_t \frac{g_t}{\sqrt{s_t + \epsilon}}, \end{aligned} \tag{2.13}$$

where β is a scalar that quantifies the averaging time of the second moment, roughly speaking, how far back in time it should track its value. Here ϵ is a scalar introduced to avoid division by zero. All other quantities are vectors. Division of these vectors is understood as element-wise.

2.3.5 ADAM

The ADAM optimizer extends the former algorithm further by using the running average of the first moment $m_t = \langle g_t \rangle$ and the second moment s_t to adapt the learning rate for each direction in parameter space. The update rule is a follows.

$$\begin{aligned} g_t &= \nabla_\theta \mathcal{L}^B \\ m_t &= \beta_1 m_{t-1} + (1-\beta) g_t \\ s_t &= \beta_2 s_{t-1} + (1-\beta_2) g_t^2 \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{s}_t &= \frac{s_t}{1-\beta_2^t} \\ \theta_{t+1} &= \theta_t - \eta_t \frac{\hat{m}_t}{\sqrt{\hat{s}_t}+\epsilon}, \end{aligned} \tag{2.14}$$

where $\beta_1 = 0.9$ and $\beta_2 = 0.99$ are typically chosen. These scalars play the same role as in RMSprop, where the quantify roughly how far back in "time" to evaluate the running averages.

2.4 Bayesian Formulation

2.4.1 Bayes' Rule

2.4.2 Bayesian Viewpoint of Optimization

Succinctly, we can write the objective of optimization as finding the optimal parameter $\hat{\theta}$ as

$$\hat{\theta} = \arg\min_\theta \sum_i \mathcal{L}(\hat{f}(x_i;\theta), y_i). \tag{2.15}$$

Assuming a loss function is RSS with L^2-regularization, which is a common choice for regression tasks. Then the loss function for a dataset of n points has the form

$$\mathcal{L} = \frac{1}{2}\sum_i \left\| y^{(i)} - f(x^{(i)};\theta)\theta) \right\|_2^2 + \frac{\lambda}{2}\|\theta\|_2^2. \tag{2.16}$$

This is interpreted as the *negative log likelihood* of the posterior $p(\theta|D)$ such that

$$p(\theta|D) \propto \prod_i \exp\left(-\frac{1}{2}\left\| y^{(i)} - f(x^{(i)} \right\|_2^2\right) \exp\left(-\frac{\lambda}{2}\|\theta\|_2^2\right), \tag{2.17}$$

where the likelihood function is the first factor and the prior is the second factor. Minimizing the loss function is then equivalent to maximizing the posterior, known as the *maximum-a-posteriori* (MAP), written as

$$\hat{\theta} = \arg\max_\theta p(\theta|D). \tag{2.18}$$

2.4.3 Bias-Variance Trade-Off Vanishes

Chapter 3

Markov Chain Monte Carlo

In this chapter, we will discuss fundamental ideas pertaining to *Markov Chain Monte Carlo* (MCMC) methods. We shall confine the discussion to continuous sample spaces which is the category of sample space focused on in this thesis. We will commence with a discussion of expectation values and an important notion called the *typical set*. We will then define and discuss Markov chains and Markov transitions after which we shall discuss Metropolis-Hastings sampling and its limitations. We will adopt a geometric view to provide a natural transition to Hamiltonian Monte Carlo and the No-U-Turn sampler in the two following chapters. The treatment will closely follow [1]

3.1 Expectation Values and the Typical Set

Consider a probability density function $\pi(q)$ and a d-dimensional sample space Q where $q \in Q$. Consider $f(q)$ to be an arbitrary smooth function of q. The *expectation value* of a *target function* $f(q)$ with respect to the density $\pi(q)$ is then defined as

$$\mathbb{E}_\pi[f] = \int \mathrm{d}q\ \pi(q) f(q). \tag{3.1}$$

We shall interchangably refer to expectation values simply as *expectations*. For all but a few simple low-dimensional densities, computing eq. (3.1) is impossible to evaluate analytically. For high-dimensional spaces, evaluating the expectation over the entire sample space is computationally infeasible. Moreover, it is unlikely that the entire sample space contribute significantly to the expectation. Efficiently evaluating the expectations may thus only require evaluation of the integrand in specific regions of the sample space. For most purposes,the we are interested in the expectation of more than a single target function. For example, in Bayesian applications, we are often interested in both mean and variance of a quantity which introduces the need for several target functions. Thus the numerical method should not depend on the target function in question. Instead the focus is laid on the contribution from $\pi(q)\mathrm{d}q$ to the integrand and which region of sample space that makes this quantity non-neglible. This region of sample space is called the *typical set*. Numerical methods that efficiently sample points from the typical set are what we refer to as MCMC methods.

3.1.1 The Typical Set

MCMC methods are devised to efficiently sample points q of the typical set. For simplicity, we can divide sample space into three regions

1. High-probability density region. These are regions in the neighborhood of any mode of the target density.

2. The typical set. This refer to the regions in which $\pi(q)\mathrm{d}q$ provides a non-neglible contribution to any expectation. This may be thought of as the high-probability region of the sample space since $\pi(q)\mathrm{d}q$ is proportional to probability of a state q in a volume $\mathrm{d}q$.

3. Low-probability density regions. These are regions far away from any mode of the density.

The first and third region will yield neglible contributions to an expectation. The point to stress here is that the interesting part of sample space to explore is not the high-probability region, and is not the part MCMC methods sample from. Instead, they sample from the region in which $\pi(q)\mathrm{d}q$ is large. Although the notion of a typical set can be formalized precisely, we will intentionally operate with this somewhat imprecise definition. For our purposes, it suffices to use it merely as a conceptual notion to evaluate the quality of the samples stored from a MCMC chain.

3.2 Markov Chains and Markov Transitions

Since evaluation of eq. (3.1) in most interesting cases is intractible, we seek to evaluate them from *Markov chains.* A Markov chain is a sequence of points $q_1, ..., q_N$ generated sequentially using a random map called a *Markov transition.* A Markov transition is a conditional probability density $T(q'|q)$ that yields the probability of transition from a point q to q'. The Markov transition is often called the *transition kernel*, which is a term we will adopt.

An arbitrary transition kernel is not useful because the generated Markov chain is unlikely to have any relation to the target distribution of interest. To generate a useful Markov chain, we must use a transition kernel that preserves the target distribution. The condition imposed to ensure this is expressed as

$$\pi(q) = \int dq' \pi(q') T(q|q'). \tag{3.2}$$

The condition is formally called *detailed balance.* The interpretation of the condition is that the Markov chain is reversible. We can start from any q and use the transition kernel to produce a set of new states. The distribution generated by the Markov chain should be distributed according the target distribution regardless of which point we used to generate the chain from. A more important fact is that as long as this condition is satisfied, the Markov chain will converge to and stay within the typical set. The standard approach to approximate eq. (3.1) is with the MCMC *estimator*

$$\hat{f}_N = \frac{1}{N} \sum_{i=1}^{N} f(q_i). \tag{3.3}$$

For large enough N, the estimator will converge to the true expectation such that $\lim_{N\to\infty} \hat{f}_N = \mathbb{E}_\pi[f]$. Obviously, the knowledge that the estimator will asymptotically converge to the true expectations are of limited use when restricted to a practical computation in which only a finite chain can be generated. We are thus more interested in the properties of finite Markov chains.

3.2.1 Ideal Markov Chains

An ideal Markov chain can be divided into three phases.

1. A convergence phase. The Markov chain is initiated from some point q and the initially generated sequence lies in a region outside the typical set. Estimators evaluated using this part of the sequence are highly biased.

2. An exploration phase. The Markov chain has reached the typical set and begins its first traversal of it. In this phase, estimators will rapidly converge towards their true values.

3. A saturation phase. At this point, the Markov chain has explored most of the typical set and convergence of the estimators slow down significantly.

The ideal evaluation of estimators thus only use the parts of Markov chain generated in the second and third phase, discarding the the chain generated in the first phase. The notion of discarding the points from the first phase is called *burn-in.*

3.2.2 Pathologies

Unfortunately, many target distributions embody typical sets with pathological regions where any transition kernel that obeys eq. (3.2) is not sufficient to efficiently explore the typical set. Geometrically, this can be regions in which the target distribution rapidly changes in certain localized regions of the typical set. The pathological regions can be completely ignored by the chain for much of the exploration, leading to poor convergence and thus biased estimators. But as long as the transition kernel satisfies detailed balance, we know for a fact that the estimators *must* converge eventually. Consequentially, the Markov chain partially be stuck near pathological regions for long periods to compensate before it rapidly explores other parts of the typical set. This behaviour is repeated, which makes estimators oscillate. Regardless of when the MCMC chain is terminated, the estimator will likely be biased due to this oscillating behaviour.

3.2.3 Geometric Ergodicity

Generation of ideal Markov chains is a certainty if the transition kernel satisfies *geometric ergodicity* [2]. However, in most cases it is impossible to check that the condition is satisfied. Instead one uses a statistical quantitiy known as the *potential scale reduction factor* $\hat{R}$. The ideal value is $\hat{R} = 1$. For values far away from this target, it is unlikely that geometric ergodicity is satisfied.

3.3 Metropolis-Hastings

Construction of a transition kernel that ensures convergence to the typical set of the target distribution is a non-trivial problem in general. Fortunately, the Metropolis-Hastings algorithm provides a solution that lets us construct a transition kernel with this property [3, 4]. The algorithm consist of two components, a proposal of a new state and a correction step. Given a state q, we propose a new state q' by adding a random perturbation to the initial state. The correction step rejects an proposed state that ends moves away from the typical set of the target distribution. The proposed state is formally sampled from a proposal distribution $Q(q'|q)$. The so-called *acceptance probability* $a(q'|q)$, that is the probability of accepting a proposed state q' given an initial state q, is given by

$$a(q'|q) = \min\left(1, \frac{Q(q|q')\pi(q')}{Q(q'|q)\pi(q)}\right). \tag{3.4}$$

Algorithm 1 Metropolis-Hastings

```
procedure METROPOLIS-HASTINGS($q$)

    Sample $q' \sim Q(q'|q)$

    $a(q'|q) \leftarrow \min\left(1, \frac{Q(q|q')\pi(q')}{Q(q'|q)\pi(q)}\right)$
    Sample $u \sim \text{Uniform}(0,1)$.

    if $a(q'|q) \geq u$ then
        $\theta \leftarrow \theta'$                                ▷ Accept transition
    else
        $\theta \leftarrow \theta$                                 ▷ Reject transition
    end if

end procedure
```

There are many valid choices for proposal distribution. A common choice is a Gaussian distribution $Q(q'|q) = \mathcal{N}(q'|q, \Sigma)$ which we will refer to as *random walk Metropolis*. More precisely, this means that a proposed state is given by

$$q' = q + r \tag{3.5}$$

where $r \sim \mathcal{N}(0, I)$. This distribution is symmetric such that $Q(q'|q) = Q(q|q')$, implying that the acceptance probability reduces to

$$a(q'|q) = \min\left(1, \frac{\pi(q')}{\pi(q)}\right). \tag{3.6}$$

Hence, evaluation of the acceptance probability only requires evaluating the target distribution at the initial state and the proposed state.

3.3.1 Random Walk Metropolis

The random walk Metropolis algorithm does suffer from slow convergence to, and exploration of, the typical set in high-dimensional spaces. This can be understood because of the following: a random perturbation will likely cause the proposed state to lie outside the typical set, which leads to rejection of the proposed state. We can compensate for this flaw by reducing the standard deviation of each dimension, but this will naturally lead to slow movement through sample space. The slow exploration also leads to a Markov chain where consecutive embody a relatively large measure of correlation. The effect is that the effective sample size grows slowly and efficient evaluation of eq. (3.1) becomes difficult. Fortunately, there exists a solution; *gradient-informed* exploration of the sample space, manifested in the form of *Hamiltonian Monte Carlo*. This algorithm is a special case of a Metropolis-Hastings algorithm in which the proposal distribution $Q(q'|q)$ is a special one utilizing Hamiltonian dynamics and Gibbs sampling to produce a new proposal state q'. This is the topic of the next chapter.

3.4 Gibbs Sampling

The final standard MCMC algorithm we need is the *Gibbs* sampler. It plays a small part of the sampling in HMC and so we should therefore briefly discuss it. It is a MCMC sampling method used for multi-variate probability densities, and so is only meaningful to discuss for $d > 1$ dimensions. Suppose $\gamma^{(t)}$ represents the parameters at iteration t. The next sample $\gamma^{(t+1)}$ in the Markov chain is drawn according to a conditional distribution as follows.

$$\gamma_i^{(t+1)} \sim P(\gamma_i|\gamma_1^{(t+1)}, \dots \gamma_{i-1}^{(t+1)}, \gamma_{i+1}^{(t)}, \dots, \gamma_d^{(t)}). \tag{3.7}$$

We may summarize this as a function in algorithm 2 which given an initial state $\gamma^{(t)}$ returns a new state sampled according to eq. (3.7).

Algorithm 2 Gibbs sampling

```
function GIBBS($\gamma^{(t)}$)
    for $i = 1, \dots, d$ do
        Sample $\gamma_i^{(t+1)} \sim P(\gamma_i|\gamma_1^{(t+1)}, \dots \gamma_{i-1}^{(t+1)}, \gamma_{i+1}^{(t)}, \dots, \gamma_d^{(t)})$.
    end for
return $\gamma^{(t+1)} = \left(\gamma_1^{(t+1)}, \dots, \gamma_d^{(t+1)}\right)$.
end function
```

Chapter 4

Hamiltonian Monte Carlo

In this chapter, we will study the details of the Hamiltonian Monte Carlo (HMC) method. It is a Markov Chain Monte Carlo (MCMC) sampling technique that merges Gibbs sampling, Hamiltonian dynamics with a final Metropolis-Hastings update. It avoids random walk behaviour with a systematical exploration of the state space and generates successive samples with smaller correlation using gradient-informed exploration. We will begin with a survey of Lagrangian and Hamiltonian dynamics followed by a description of the *Leapfrog* integrator which is used to simulate the Hamiltonian systems. Once these are established, we will summarize the HMC method in a generic manner - applicable to any continuous distribution. Moreover, we will discuss important properties like conservation of the Hamiltonian and local phase space volume.

4.1 Gradient-Informed Exploration

HMC uses gradient-informed exploration to efficiently traverse through the typical set. This is achieved by formation of a vector field which is tangent to the typical set at all points in the sample space. Recall that the typical does coincide with the modes of the target distribution $\pi(q)$. Therefore, the gradient of the target distribution itself is of limited value because it would move the Markov chain outside of the typical set. HMC overcomes this problem with an introduction of *auxilliary momentum* variables inspired by analytical mechanics.

Chapter 5

The No-U-Turn Sampler

HMC is considered a state-of-the-art sampling method, but suffers the need for manual tuning of the number of leapfrog steps L and the step size ϵ. In this chapter, we will study a sampling method called *The No-U-Turn sampler* (NUTS) [5] built upon HMC that dynamically adapts the integration length ϵL. Moreover, we will look at how we can adaptively set the step size to help NUTS reach a stopping criterion faster.

5.1 Preliminary definitions

The NUTS algorithm introduces a collection of new ideas that need proper development before we delve into the algorithm itself. Given generalized coordinates $q = (q_1, ..., q_d)$ and generalized momenta $p = (p_1, ..., p_d)$, we introduce a *slice* variable u with corresponding conditional distributions

$$p(u|q,p) = \text{Uniform}\left(u; [0, \exp\{-H(q,p)\}]\right), \tag{5.1}$$

and

$$p(q,p|u) = \text{Uniform}\left(q,p; \left\{q',p' \middle| \exp\{-H(q,p)\} \geq u\right\}\right). \tag{5.2}$$

This effectively allows for definition of the joint distribution

$$p(q,p,u) \propto \mathbb{I}\left[u \in [0, \exp\{-H(q,p)\}]\right], \tag{5.3}$$

where $\mathbb{I}[\cdot]$ evaluates to 1 if its argument is true and 0 otherwise. Integrating over u yields the marginal distribution over phase-space

$$p(q,p) \propto \int p(q,p,u)\mathrm{d}u = \int_0^{\exp\{-H(q,p)\}} \mathrm{d}u = \exp\{-H(q,p)\}. \tag{5.4}$$

which is the target distribution used in HMC. We further introduce the set $\mathcal{B}$ which consists of all states (q,p) traced out by the leapfrog integrator. We will get back to the generation of its elements shortly. We introduce another set $\mathcal{C}$ of candidate states which is a subset $\mathcal{C} \subseteq \mathcal{B}$. The set $\mathcal{C}$ will be chosen deterministically from $\mathcal{B}$ such that none of its elements violates detailed balance if used to produce a transition. Generation of the sets $\mathcal{B}$ and $\mathcal{C}$ given the parameters q, p, u and ϵ defines a conditional distribution $p(\mathcal{B},\mathcal{C}|q,p,u,\epsilon)$ on which the following properties are imposed:

1. All elements of $\mathcal{C}$ are volume perserving. This effectively translates to $p((q,p)|(q,p) \in \mathcal{C}) \propto p(q,p)$.
2. The current state must be included in $\mathcal{C}$, i.e $p\left((q,p) \in \mathcal{C}|q,p,u,\epsilon\right) = 1$.
3. Any state $(q',p') \in \mathcal{C}$, must be in the slice defined by u. Mathmetically, this is expressed as

$$p\left(u \leq \exp\{-H(q,p)\} \middle| (q',p') \in \mathcal{C}\right) = 1.$$

4. If $(q, p) \in \mathcal{C}$ and $(q', p') \in \mathcal{C}$, then for any $\mathcal{B}$ we impose $p(\mathcal{B}, \mathcal{C}|q, p, u, \epsilon) = p(\mathcal{B}, \mathcal{C}|q', p', u, \epsilon)$. Thus any point in $\mathcal{C}$ is equally likely. This can be encapsulated by introduction of the *transition kernel*

$$\frac{1}{|\mathcal{C}|} \sum_{(q,p)\in\mathcal{C}} T(q', p'|q, p, \mathcal{C}) = \frac{\mathbb{I}\left[(q', p') \in \mathcal{C}\right]}{|\mathcal{C}|}, \tag{5.5}$$

which expresses that a proposed point (q', p') is sampled uniformly from $\mathcal{C}$.

5.1.1 Generation of Candidate Points and Stopping Criterion

Conceptually, generation of $\mathcal{B}$ proceeds as follows. First a single Leapfrog step is integrated forwards or backwards in time, where the direction in time is chosen randomly. Then we repeat with two Leapfrog steps. And then we reiterate with four Leapfrog steps. And so on. The algorithm repeats this procedure until some hitherto undefined stopping criterion is reached. This effectively builds a balanced binary tree in which each node is an element in $\mathcal{B}$. The initial tree is a single node which naturally represents the initial state in phase space. Formally, we double the tree by choosing a random direction $v_j \in \text{Uniform}(\{-1, 1\})$ with $v_j = 1$ representing a trajectory forwards in time and $v_j = -1$ representing a trajectory backwards in time where j is the current depth of the tree. The initial tree node corresponds to a depth of $j = 0$. Consider a tree of depth j. NUTS will then consider the $2^j - 1$ balanced subtrees of the height j-tree with $j > 0$. Let q^- and p^- represent the leftmost leaves of one of its subtrees and q^+ and p^+ represent the rightmost leaves of the same subtree. If either

$$(q^+ - q^-) \cdot p^+ < 0 \quad \text{or} \quad (q^+ - q^-) \cdot p^- < 0, \tag{5.6}$$

the tree doubling is terminated. These conditions express that notion that the trajectory starts turning back toward regions that are already visited in phase space. A so-called "U-turn". Thus the algorithm requires $2^{j+1} - 2$ inner products for a tree of height j (two inner products per subtree it must consider). This is an added computational cost over standard HMC per leapfrog step. The added cost is, however, negligible for sufficiently large datasets and/or complex models because the computational cost will be dominated by the computation of gradients with respect to the model parameters.

The second stopping criterion considered by NUTS terminates the tree doubling if any of the nodes in the tree of height j yields an energy difference larger than for some maximum energy difference $E_{\max}$. More formally, the constraint follows roughly from the introduction of the slice variable u which required that

$$u \leq \exp\left\{-H(q, p)\right\}, \tag{5.7}$$

which results in

$$H(q, p) + \log u \leq 0. \tag{5.8}$$

NUTS loosens this constraint and simply requires that this sum is less than some max energy value. Thus the tree doubling is terminated if

$$H(q, p) + \log u > E_{\max}, \tag{5.9}$$

is satisfied.

5.1.1.1 Choosing candidate points

We now turn to the problem of choosing which points of $\mathcal{B}$ that should be part of $\mathcal{C}$. Recall the the first property we imposed on the distribution $p(\mathcal{B}, \mathcal{C}|q, p, u, \epsilon)$ was that any point in $\mathcal{C}$ must be volume preserving. Luckily, the Leapfrog integrator is volume preserving, so this condition is automatically satisfied. The second condition was simply that the current state must be included in $\mathcal{C}$, so we simply allow the possibly to transition back to the same state to satisfy this condition. The third condition stated that any point $(q, p) \in \mathcal{C}$ must be part of the slice defined by u. Thus, if we exclude any point (q, p) that does not satisfy $u \leq \exp\{-H(q, p)\}$, we fulfill the condition. The fourth condition was $p(\mathcal{B}, \mathcal{C}|q, p, u, \epsilon) = p(\mathcal{B}, \mathcal{C}|q', p', u, \epsilon)$ for any point $(q, p) \in \mathcal{C}$. For any $(q', p') \in \mathcal{B}$, there exists at most one unique sequence $v_0, ..., v_j$ that generates all other states in $\mathcal{B}$ through the doubling process described above. To satisfy the condition, we must exclude any point that from which it is impossible to generate $\mathcal{B}$. Such a point will either satisfy eq. (5.6) or eq. (5.9) before it manages to complete the tree structure, which halts the generation of the necessary points in $\mathcal{B}$. Assume that the doubling procedure stopping because either condition was satisfied during the last iteration. The final points added to $\mathcal{B}$ must be excluded from $\mathcal{C}$ because they will stop the doubling process and thus cannot produce $\mathcal{B}$ if used as a starting point.

5.2 The Naive NUTS Algorithm

The naive NUTS implementation uses recursion to implicitly store proposal points $\mathcal{C}$ whilst building the balanced binary tree. For convenience, the algorithm is split into two pieces; a function called "BUILDTREE" which can be found in algorithm 3, and a procedure called "NaiveNUTS" in algorithm 4 which performs one step of NUTS similar to the one-step procedure of HMC we discussed in algorithm **??**, producing a new point q^*. Let us discuss the computational cost of this algorithm. The algorithm demands $2^j - 1$ evaluations of $H(q,p)$ and its gradient. Moreover, an additional set of operations to determine if a stopping criterion is reached, which is of the order $\mathcal{O}(2^j)$. As argued earlier, though, the computational cost is comparable to standard HMC per leapfrog step when the model is sufficiently complex or the dataset large. However, in its current form it requires storage of 2^j positions and momenta, which for complex models or deep binary trees may results in an intractibly large storage requirement. In the next section we shall study a more efficient solution to reduce the memory footprint of the algorithm.

Algorithm 3 BuildTree function

```
function BUILDTREE($q, p, u, v, j, \epsilon$)
    if $j = 0$ then                    ▷ Initial state of the balanced binary tree. Base case.
        $(q', p') \leftarrow$ LEAPFROG$(q, p, v\epsilon)$.
        $\mathcal{C}' \leftarrow \{(q', p')\}$  if  $u \leq \exp\{-H(q', p')\}$  else  $\mathcal{C} \leftarrow \emptyset$.
        $s' \leftarrow \mathbb{I}[H(q,p) + \log u \leq E_{\max}]$                    ▷ Stopping criterion of eq.(5.9).
        return $q', p', q', p', \mathcal{C}', s'$.
    else                    ▷ Recursion case where $j > 0$. Builds left and right subtrees.
        $q^-, p^-, q^+, p^+, \mathcal{C}', s' \leftarrow$ BUILDTREE$(q, p, u, v, j-1, \epsilon)$
        if $v = 1$ then
            $q^-, p^-, -, -, \mathcal{C}'', s'' \leftarrow$ BUILDTREE$(q^-, p^-, u, v, j-1, \epsilon)$.
        else
            $-, -, q^+, p^+, \mathcal{C}'', s'' \leftarrow$ BUILDTREE$(q^+, p^+, u, v, j-1, \epsilon)$.
        end if
        $s' \leftarrow s's''\mathbb{I}[(q^+ - q^-) \cdot p^- \geq 0]\mathbb{I}[(q^+ - q^-) \cdot p^+ \geq 0]$.                    ▷ Stopping criterion from eq. (5.6)
        $\mathcal{C} \leftarrow \mathcal{C}' \cup \mathcal{C}''$                    ▷ Expand candidate sets
        return $q^-, p^-, q^+, p^+, \mathcal{C}', s'$.
    end if
end function
```

Algorithm 4 Naive NUTS sampler

```
procedure NaiveNUTS($q, \epsilon$)
    Sample $u \sim$ Uniform $([0, \exp\{-H(q,p)\}])$.    ▷ Slice variable.
    Initialize $s = 1$, $q^{\pm} = q$, $p^{\pm} = p$, $j = 0$, $\mathcal{C} = \{(q,p)\}$.
    Sample $p \sim \mathcal{N}(0, I)$.    ▷ Momenta
    while $s = 1$ do
        Sample $v_j \sim$ Uniform$(\{-1, 1\})$    ▷ Choose direction in phase space
        if $v_j = -1$ then
            $q^-, p^-, -, -, \mathcal{C}', s' \leftarrow$ BuildTree$(q^-, p^-, u, v_j, j, \epsilon)$.
        else
            $-, -, q^+, p^+, \mathcal{C}', s' \leftarrow$ BuildTree$(q^+, p^+, u, v_j, j, \epsilon)$.
        end if
        if $s' = 1$ then
            $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$    ▷ Expand set of candidate points if stopping criterion is not met.
        end if
        $s \leftarrow s' \mathbb{I}[(q^+ - q^-) \cdot p^- \geq 0] \mathbb{I}[(q^+ - q^-) \cdot p^+ \geq 0]$.    ▷ Stopping criterion of eq. (5.6)
        $j \leftarrow j + 1$    ▷ Increment tree depth.
    end while
    Sample $q^*$ uniformly from $\mathcal{C}$
    return $q^*$.
end procedure
```

5.3 Efficient NUTS

The implementation resulting from algorithm 3 and 4 yields approximately the same computational cost as standard HMC for complex models or large datasets. There are several weaknesses which can be improved upon:

1. The algorithm stores 2^j positions and momentum for a tree of depth j. For sufficiently complex models or deep enough tree depth, this may require a too large memory footprint.
2. The transition kernel used in algorithm 4 produces "short" transitions in parameter space. There exist alternative transition kernels which produces larger transitions in parameter space while obeying detailed balance with respect to a uniform distribution over $\mathcal{C}$.
3. If a stopping criterion is satisfied during the final doubling iteration, the proposed set $\mathcal{C}'$ is still completely built before termination. A more efficient solution is to terminate the creation of the final proposed set by simply terminating immediately when a stopping criterion is reached.

First, consider the first and second weaknesses. We can introduce a kernel

$$T(q', p' | q, p, \mathcal{C}) = \begin{cases} \dfrac{\mathbb{I}[(q', p') \in \mathcal{C}_{\text{new}}]}{|C_{\text{new}}|} & \text{if} \quad |\mathcal{C}_{\text{new}}| > |\mathcal{C}_{\text{old}}|, \\ \\ \dfrac{|\mathcal{C}_{\text{new}}|}{|\mathcal{C}_{\text{old}}|} \dfrac{\mathbb{I}[(q', p') \in \mathcal{C}_{\text{new}}]}{|\mathcal{C}_{\text{new}}|} + \left(1 - \dfrac{|\mathcal{C}_{\text{new}}|}{|\mathcal{C}_{\text{old}}|}\right) \mathbb{I}[(q', p') = (q, p)] & \text{if} \quad |\mathcal{C}_{\text{new}}| \leq |\mathcal{C}_{\text{old}}|, \end{cases} \tag{5.10}$$

where $\mathcal{C}_{\text{new}}$ and $\mathcal{C}_{\text{old}}$ are disjoint subsets of $\mathcal{C}$ such that $\mathcal{C} = \mathcal{C}_{\text{old}} \cup \mathcal{C}_{\text{new}}$. Here $(q, p) \in \mathcal{C}_{\text{old}}$ represents elements already present in $\mathcal{C}$ before the final doubling iteration and $\mathcal{C}_{\text{new}}$ represents the set of elements added to $\mathcal{C}$ during the final doubling iteration. The transition kernel can be interpreted to describe a probability of a transition from a state in $\mathcal{C}_{\text{old}}$ to a randomly chosen state in $\mathcal{C}_{\text{new}}$. The move is accepted with probability $|\mathcal{C}_{\text{new}}|/|\mathcal{C}_{\text{old}}|$. The storage requirement can be reduced to the order $\mathcal{O}(j)$ by observing that

$$p(q, p | \mathcal{C}') = \frac{1}{|\mathcal{C}'|} = \frac{|\mathcal{C}_{\text{subtree}}|}{|\mathcal{C}'|} \frac{1}{|\mathcal{C}_{\text{subtree}}|} = p((q, p) \in \mathcal{C}_{\text{subtree}} | \mathcal{C}') P(q, p | (q, p) \in \mathcal{C}_{\text{subtree}}, \mathcal{C}'). \tag{5.11}$$

5.4 Dual-Averaging Step Size Adaptation

This section will introduce a step size adaptation scheme.

5.5 NUTS with Dual-Averaging Step Size Adaptation

This section will combine the two algorithms to the one used in most runs in this thesis.

Chapter 6

Bayesian Learning for Neural Networks

6.1 Neural Networks

In this chapter, we will introduce the mathematical formalism underpinning neural networks. For convenience, we will adopt the terminology used by Tensorflow[6] to help make the transition from the mathematics to their machine learning framework easier. We will stay general and assume a set of inputs $x \in \mathbb{R}^p$ and corresponding targets $y \in \mathbb{R}^d$. These serve as the training data on which the neural network is trained.

6.1.1 Basic Mathematical Structure

A neural network is most generally defined as a non-linear function $f : \mathbb{R}^p \to \mathbb{R}^d$. This non-linear function is built-up as follows:

- A set of L layers. Consider the ℓ'th layer. It consists of n_ℓ nodes all of which has a one-to-one correspondence to a real number. The conventional representation is through a real-valued vector $a^\ell \in \mathbb{R}^{n_\ell}$, where a^ℓ is colloquially called the *activation* of layer ℓ.
- For convenience, the layer with $\ell = 1$ is often called the *input layer* and the layer with $\ell = L$ is called the *output layer*, and the layers in between for $\ell = 2, ..., L-1$ are called the *hidden layers*. Although this distinction is merely conceptual and does not change the mathematics one bit, it provides useful categories for discussion later on.
- Each layer ℓ is supplied with a (possibly) non-linear function $\sigma_\ell : \mathbb{R}^{n_{\ell-1}} \to \mathbb{R}^{n_\ell}$. In other words, it defines a mapping $a^{\ell-1} \mapsto a^\ell$. The complete neural network function can thus be expressed as

$$f(x) = (\sigma_L \circ \sigma_{L-1} \circ \cdots \circ \sigma_\ell \circ \cdots \circ \sigma_2 \circ \sigma_1)\,(x). \tag{6.1}$$

- To each layer, we assign a *kernel* $W^\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$ and a *bias* $b^\ell \in \mathbb{R}^{n_\ell}$. Together, these parameters are called the *weights* of a layer.
- The complete set of neural network parameters $(W, b) = \{(W^\ell, b^\ell)\}_{\ell=1}^{L}$ are called the weights of the network. They serve as the *learnable* or *trainable* parameters of the model.
- Finally, we introduce the *logits* $z^\ell \in \mathbb{R}^{n_\ell}$ of layer ℓ.
- The permutation of chosen number of layers, number of nodes per layer and activation functions are collectively called the *architecture* of the neural network.

The activation in layer ℓ is computed through the recursive equation:

$$a_j^\ell = \sigma_\ell\left(\sum_k W_{jk}^\ell a_k^{\ell-1} + b_j^\ell\right) \equiv \sigma_\ell(z_j^\ell), \quad \text{for} \quad j = 1, 2, ..., n_\ell. \tag{6.2}$$

A special case of eq. (6.2) applies to $\ell = 1$ where $a^0 = x \in \mathbb{R}^p$ is assumed.

6.1.2 Backpropagation

The standard approach to train a neural network is by minimization of some loss function by employing the *backpropagation* algorithm[7]. The algorithm boils down to four equations defining a recursive algorithm that approximates the gradient with respect to the parameters of the model. Consider $\mathcal{L}$ as the loss function, quanitifying the error between the target and the model output. The first of the four equations quantifes the error in the output layer.

$$\Delta_j^L = \frac{\partial \mathcal{L}}{\partial z_j^L}. \tag{6.3}$$

The second equation allows us to compute the error at layer ℓ given we know the error at layer $\ell + 1$,

$$\Delta_j^\ell = \left(\sum_k \Delta_k^{\ell+1} W_{kj}^{\ell+1}\right) \sigma'_\ell(z_j^\ell). \tag{6.4}$$

The final two equations relate these errors to the gradient of the loss function with respect to the model parameters. For the kernels, we have

$$\frac{\partial \mathcal{L}}{\partial W_{jk}^\ell} = \frac{\partial \mathcal{L}}{\partial z_j^\ell}\frac{\partial z_j^\ell}{\partial W_{jk}^\ell} = \Delta_j^\ell a_k^{\ell-1}. \tag{6.5}$$

For the biases, the gradients are

$$\frac{\partial \mathcal{L}}{\partial b_j^\ell} = \frac{\partial \mathcal{L}}{\partial z_j^\ell}\frac{\partial z_j^\ell}{\partial b_j^\ell} = \Delta_j^\ell. \tag{6.6}$$

With these four equations, we can fit the neural network using minimization techniques such as stochastic gradient descent or more complex methods such as ADAM (pages 13-19 in [8]). Although not the focus of this thesis, we might use these methods in conjunction with HMC to speed up convergence to the stationary distribution. Furthermore, the computation of gradients in combination with HMC can be performed with the backpropagation algorithm.

6.1.3 Loss Function for Regression

In this thesis, we are concerned with regression tasks. The activation function of the final layer σ_L is then just the identity function. The typical loss function chosen to solve regression tasks is the L_2-norm, which for a single output can be written as

$$\mathcal{L}(y, \hat{y}) = \frac{1}{2}\|y - \hat{y}\|_2^2, \tag{6.7}$$

where $\hat{y}$ denotes the model output and y the ground-truth. Now, the model output in this case is $\hat{y}_j = a_j^L = z_j^L$. Therefore,

$$\Delta_j^L = \frac{\partial \mathcal{L}}{\partial z_j^L} = a_j^L - y_j. \tag{6.8}$$

We are now equipped to write down the backpropagation for a single datapoint. It's built up of a *forward pass* which takes an input x and applies the recursive eq. (6.2) which produces a model prediction $\hat{y} = a^L$. The second part of the algorithm is the *backward pass* which based on the prediction $\hat{y}$ and the target y, computes the gradients of the loss function $\mathcal{L}$ with respect to the model parameters. The forward pass of the neural network is summarized algorithm 5.

Algorithm 5 Backpropagation: Forward pass

```
procedure ForwardPass($x$)
    $a_j^0 = x_j$    for    $j = 1, \dots, p$                ▷ Initialize input
    for $\ell = 1, 2, .., L$ do
        for $j = 1, 2, .., n_\ell$ do
            $a_j^\ell \leftarrow \sigma_\ell \left( \sum_k W_{jk}^\ell a_k^{\ell-1} + b_j^\ell \right)$
        end for
    end for
end procedure
```

The backward pass of the algorithm is stated in algorithm 6.

Algorithm 6 Backpropagation: Backward pass

```
procedure BackwardPass($y$)
    for $j = 1, 2, \dots, n_L$ do
        $\Delta_j^L \leftarrow a_j^L - y_j$
        $\partial E / \partial b_j^L \leftarrow \Delta_j^L$
        $\partial E \Big/ \partial W_{jk}^L \leftarrow \Delta_j^L a_k^{L-1}$
    end for
    for $\ell = L - 1, \dots, 1$ do
        for $j = 1, \dots, n_\ell$ do
            $\Delta_j^\ell \leftarrow \left( \sum_k \Delta_k^{\ell+1} W_{kj}^{\ell+1} \right) \sigma'(z_j^\ell)$
            $\partial E / \partial b_j^\ell \leftarrow \Delta_j^\ell$
            $\partial E \Big/ \partial W_{jk}^\ell \leftarrow \Delta_j^\ell a_k^{\ell-1}$
            update $b_j^\ell$ and $W_{jk}^\ell$
        end for
    end for
end procedure
```

Note that for in all practical implementations in this thesis, we utilize *automatic differentiation* provided by TensorFlow to compute the gradients.

6.1.4 Regularization in Neural Networks

Neural networks often end up with a large number of parameters, which makes them prone to *overfit* training data. This means that the trained parameters of the model is adjusted to capture trends in the training data which may not represent the underlying process the model tries to learn. The consequence is that it *generalizes* poorly to new unseen data, i.e its predictions are poor. A typical strategy to avoid this problem, is to introduce some form of *regularization*. A common choice is L^2-*regularization*, which for a neural network tacks on two additional sums to the loss function as follows:

$$\mathcal{L} = \frac{1}{2} \sum_i \left\| \hat{y}^{(i)} - y^{(i)} \right\|_2^2 + \frac{\lambda_W}{2} \sum_\ell \left\| W^\ell \right\|_2^2 + \frac{\lambda_b}{2} \sum_\ell \left\| b^\ell \right\|_2^2, \tag{6.9}$$

where λ_W and λ_b are regularization strengths for the kernels and biases respectively. The L^2-norm $\|\cdot\|_2$ is the standard Euclidean norm in the case of a vector. For a matrix, we mean the following. Consider a matrix

$A \in \mathbb{R}^{m\times n}$. The matrix norm $\|\cdot\|_2$ is then given by *Fröbenius norm*

$$\|A\|_2 = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|A_{ij}|^2}. \tag{6.10}$$

L^2-regularization is sometimes called L^2-penalty because it penalizes assignment of large values to the model parameters. Its effect is thus shrinkage of the parameter space where accessible minima may reside.

6.2 Activation Functions

There are many common activation functions with various strengths used in modern neural networks. We will discuss a few common ones with an emphasis on the ones used in this thesis.

6.2.1 Sigmoid

The sigmoid activation function is given by

$$\sigma(x) = \frac{1}{1+\exp(-x)}. \tag{6.11}$$

It was a very common choice in neural networks early, likely due to its simple derivative. It has a significant drawback, however. Looking at eq. (6.11), we can easily deduce that $\sigma(\pm\infty) = 0$, and since its derivative is of the form $\sigma'(x) = \sigma(1-\sigma)$, the gradient computed during backpropagation vanishes if the input to the activation function as $|x| \to \infty$. This significantly hampers the progress during optimization. A popular alternative to the sigmoid function is $\tanh(x)$. This function too, however, suffers from the same vanishing gradient problem.

6.2.2 ReLU

To overcome the vanishing gradient problem, an activation function called the Rectifying Linear Unit (ReLU) became widely adopted, which is given by

$$\sigma(x) = x^+ = \max(0, x). \tag{6.12}$$

6.2.3 Swish

Recently, an activation function to replace ReLU was proposed in [9] known as *swish* or SiLU which was shown to outperform ReLU in deep neural networks on a number of challengig datasets. The activation function is given by

$$\sigma(x) = x \cdot \mathrm{sigmoid}(x). \tag{6.13}$$

6.3 Bayesian Inference

The foundation for Bayesian inference is Bayes' theorem, which can be formulated as

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}. \tag{6.14}$$

where D is observed data and θ are the model parameters. Some useful terminology is in order. $P(\theta)$ is called the *prior* distribution and embodies our prior knowledge of θ before any new observations are considered. $P(D|\theta)$ is called the *likelihood* function and provides information about θ learned from observing the data D. The *posterior* distribution $P(\theta|D)$ models our belief about θ after the data D is observed. More succinctly, we can write Bayes' theorem as

$$P(\theta|D) \propto P(\theta|D)P(\theta), \tag{6.15}$$

because its rarely of interest, or tractable, to compute $P(D)$, known as the *evidence*.

The objective of Bayesian inference is to compute the *predictive* distribution for an unseen datapoint y^*, which can be expressed as the integral over all parameters of the posterior distribution weighted by the likelihood. Given a dataset of observations $D = \{y^{(1)}, \ldots, y^{(n)}\}$, this implies

$$P(y^*|D) = \int P(y^*, \theta|D)\mathrm{d}\theta = \int P(y^*|\theta, D)P(\theta|D)\mathrm{d}\theta = \int P(y^*|\theta)P(\theta|D)\mathrm{d}\theta, \tag{6.16}$$

which loosely describes the probability of observing y^* given the observations in D.

6.4 Bayesian Framework for Neural Networks

We can specialize the equations used in Bayesian inference for neural networks in the context of regression. The predictive distribution $P(y|x, \theta)$ seeks to model a function $f : \mathbb{R}^p \to \mathbb{R}^d$ that for a given $x \in \mathbb{R}^p$ produces an output $y \in \mathbb{R}^d$. In the infinite data limit, the distribution should be a Dirac delta function. For finite datasets, however, we instead seek a distribution of outputs given the input features.

Consider a set of observations $D = \{(x^{(1)}, y^{(1)}), ..., (x^{(n)}, y^{(n)})\}$, where $x^{(i)} \in \mathbb{R}^p$ are the input features and $y^{(i)} \in \mathbb{R}^d$ are the targets. The equation for the predictive distribution of y^* given an input x^* changes to

$$P(y^*|x^*, D) = \int P(y^*|x^*, \theta)P(\theta|D)\mathrm{d}\theta. \tag{6.17}$$

Assuming that the the observations in D are drawn independently, the likelihood function can be expressed as

$$P(D|\theta) = \prod_{i=1}^{n} P(y^{(i)}|x^{(i)}, \theta). \tag{6.18}$$

In the context of regression, the likelihood for a given observation (x, y) is commonly chosen to be

$$P(y|x, \theta) \propto \exp\left(-\frac{\lambda}{2}\|y - f(x;\theta)\|_2^2\right). \tag{6.19}$$

where $f(x;\theta)$ is the output of the neural network and λ is a hyperparameter typically chosen to be identical for all inputs (x, y) during training. The likelihood found eq. (6.19) is equivalent to using L^2-norm as a loss function with regularization strength λ when framed as a minimization problem, which we will see shortly.

In practice, however, we instead sample a set of network parameters $\{\theta_1, ..., \theta_n\}$ from the posterior distribution

$$P(\theta|D) \propto P(D|\theta)P(\theta), \tag{6.20}$$

from which we can produce a set of predictions $\{\hat{y}_1, \ldots \hat{y}_n\}$ from the neural network model

$$\hat{y}_i = f(x, \theta_i). \tag{6.21}$$

Given this set of predictions, we can compute the sample mean

$$\hat{y}_{\mathrm{MLE}} = \frac{1}{n}\sum_i f(x, \theta_i), \tag{6.22}$$

which is an approximation to the *maximum likelihood estimate* (MLE). Furthermore, an estimate of the error is provided by the sample *covariance*

$$\mathrm{Cov}(\hat{y}) = \frac{1}{n-1}\sum_i (\hat{y}^{(i)} - \hat{y}_{\mathrm{MLE}})(\hat{y}^{(i)} - \hat{y}_{\mathrm{MLE}})^T. \tag{6.23}$$

The diagonal terms of eq. (6.23) yields the sample *variance* of the components of $\{\hat{y}^{(i)}\}_{i=1}^n$.

6.5 Bayesian learning using HMC

To learn from the data D using HMC, we need to define a potential energy function $V(q)$ and a kinetic energy function $K(p)$. In the case of a neural network, the potential energy can be specified in the generic form

$$V(W,b) = -\log Z - \sum_{\ell} \log P(W^{\ell}) - \sum_{\ell} \log P(b^{\ell}) - \sum_{i} \log P(y^{(i)}|x^{(i)}, W, b). \tag{6.24}$$

where W and b collectively denotes all the kernels and biases of the model, respectively. We thus need to specify a prior for kernels W^{ℓ} and the biases b^{ℓ}, as well as a likelihood given an input (x,y). The priors are typically chosen to be Gaussian,

$$P(W^{\ell}) \propto \exp\left(-\frac{\lambda_W}{2}\left\|W^{\ell}\right\|_2^2\right), \qquad P(b^{\ell}) \propto \exp\left(-\frac{\lambda_b}{2}\left\|b^{\ell}\right\|_2^2\right). \tag{6.25}$$

The likelihood for regressions tasks was defined in eq. (6.19). Combining this and eq. (6.25), we can write down an explicit expression for the potential energy as

$$V(W,b) = -\log Z + \frac{\lambda_W}{2}\sum_{\ell}\left\|W^{\ell}\right\|_2^2 + \frac{\lambda_b}{2}\sum_{\ell}\left\|b^{\ell}\right\|_2^2 + \frac{\lambda}{2}\sum_{i}\left\|y^{(i)} - f(x^{(i)}; W, b)\right\|_2^2, \tag{6.26}$$

given a set of datapoints $D = \{x^{(i)}, y^{(i)}\}_{i=1}^{n}$. Here Z denotes the normalization constant of the distribution $P \propto \exp(-V(W,b))$, but is of no importance for the sampling procedure because it either vanishes when the gradient of V is computed, or when we compute the energy difference between the initial state (W,b) and the proposed state (W^*, b^*). Finally, we need to specify a distribution for the kinetic energy. Since we interpret the Hamiltonian as the negative log likelihood of the target distribution, we have implicitly defined the distribution as Gaussian

$$P(p) \propto \prod_{\ell} \exp\left(-\frac{(p^{\ell})^T (M^{\ell})^{-1} p^{\ell}}{2}\right), \tag{6.27}$$

where M^{ℓ} is the diagonal mass matrix of layer ℓ with its mass elements corresponding to each individual momentum and p^{ℓ} denotes the generalized momenta of layers ℓ.

With the ingredients introduced hitherto, we can proceed to sample from a network with an arbitrary network architecture. The procedure is a follows.

1. Initialize the weights of the network sampled from the priors.
2. Train the network using the backpropagation algorithm to reach an initial network state in the high probability region of the posterior. This is done to reduce the number of burn-in steps needed for the the Markov chain to reach the stationary distribution.
3. Perform a small number of burn-in steps. This step may be redundant as the network is minimized using eq. (6.26), which will place us in a high probability region once the loss is minimized.
4. Sample a set of neural network parameters from the posterior distribution.

Chapter 7

Numerical Experiments

7.1 The Dataset

7.1.1 Data Generation

7.1.2 Data Scaling and Transformations

7.2 Performance Metrics

7.3 Results

7.3.1 Benchmarks of Hyperparameters

7.3.1.1 Baseline Model

7.3.1.2 Pretraining

7.3.1.3 Burn-in length

7.3.1.4 Number of model parameters

7.3.2 Neutralino-Neutralino Cross Sections

Chapter 8

Discussion

Conclusion

Conclusion here.

Appendices

Appendix A

A.1 Appendix 1 title

Some appendix stuff.

Bibliography

[1] M. Betancourt, *A conceptual introduction to hamiltonian monte carlo*, 2017. 10.48550/ARXIV.1701.02434.

[2] G. O. Roberts and J. S. Rosenthal, *General state space markov chains and MCMC algorithms*, *Probability Surveys* **1** (jan, 2004) .

[3] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller and E. Teller, *Equation of State Calculations by Fast Computing Machines*, *The Journal of Chemical Physics* **21** (1953) 1087–1092, [https://doi.org/10.1063/1.1699114].

[4] W. K. Hastings, *Monte Carlo sampling methods using Markov chains and their applications*, *Biometrika* **57** (04, 1970) 97–109, [https://academic.oup.com/biomet/article-pdf/57/1/97/23940249/57-1-97.pdf].

[5] M. D. Hoffman and A. Gelman, *The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo*, 2011.

[6] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro et al., *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015.

[7] D. E. Rumelhart, G. E. Hinton and R. J. Williams, *Learning representations by back-propagating errors*, *Nature* **323** (1986) 533–536.

[8] P. Mehta, M. Bukov, C.-H. Wang, A. G. Day, C. Richardson, C. K. Fisher et al., *A high-bias, low-variance introduction to machine learning for physicists*, *Physics Reports* **810** (May, 2019) .

[9] P. Ramachandran, B. Zoph and Q. V. Le, *Searching for activation functions*, *CoRR* **abs/1710.05941** (2017) , [1710.05941].