# 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

# 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 $f : \mathbb{R}^p \to \mathbb{R}^d$.

## 1.1 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

$$y_i = f(x_i) + \epsilon_i, \tag{1.1}$$

for some true function $f(x_i)$ (also known as the ground truth), where $\epsilon_i$ 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 $\theta$ 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

$$\hat{y}_i = \hat{f}(x_i;\theta), \tag{1.2}$$

for brevity.

### 1.1.1 Bias-variance trade-off

From eq. (1.1), we can deduce a general feature of machine learning problems that proves challenging. We cannot directly probe the true function $f(\mathbf{x}_i)$, because only $y_i$ is observed. Because of this, choosing a model class is a delicate process. If the model class is too simple (i.e few parameters $\theta$), it is likely to capture very general features of the ground truth whilst more nuances 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.

Let's put these notions into mathematics.

## 1.2 Loss functions

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

$$\text{RSS} = \sum_{i=1}^{n} \|\hat{y}_i - y_i\|_2^2, \tag{1.3}$$

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

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n} \|\hat{y}_i - y_i\|_2^2. \tag{1.4}$$

For optimization purposes, they yield equivalent optimal parameters $\theta$.

### 1.2.1 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

$$\mathcal{L} + \lambda\|\theta\|_2^2, \tag{1.5}$$

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

$$\mathcal{L} + \lambda\|\theta\|_1. \tag{1.6}$$

The terms *penalizes* large values of $\theta$, effectively shrinking the allowed parameter space. The larger the value of the regularization strength $\lambda$, the smaller the allowed parameter space becomes.

## 1.3 Optimization

Once a model class and loss function is chosen, optimization of the model parameters commence. 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.

### 1.3.1 Gradient descent

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

$$\theta_{t+1} = \theta_t - \eta_t \sum_{i=1}^{n} \nabla_\theta \mathcal{L}(\hat{f}(x_i;\theta_t), y_i), \tag{1.7}$$

where $\theta_t$ is the model parameters at iteration $t$ and $\eta_t$ is the *learning rate*, which in general is dependent on iteration $t$, hence the subscript.

### 1.3.2 Stochastic gradient descent

The standard gradient descent 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:

$$\theta_{t+1} = \theta_t - \eta_t \sum_{(x_i,y_i)\in B_j} \nabla_\theta \mathcal{L}(\hat{f}(x_i;\theta_t), y_i). \tag{1.8}$$

An iteration over all batches $B_j \in B$ is called an *epoch*.

### 1.3.3 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. Let $v_t$ be defined by the recursive equation

$$v_t = \gamma v_{t-1} + \eta_t \sum_{(x_i, y_i) \in B_j} \nabla_\theta \mathcal{L}(\hat{f}(x_i; \theta_t), y_i). \tag{1.9}$$

Then the update rule for the parameters is

$$\theta_{t+1} = \theta_t - v_t. \tag{1.10}$$

### 1.3.4 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

## 1.4 Data preprocessing

### 1.4.1 Data splitting

### 1.4.2 Data scaling

## 1.5 Optimization formulated in terms of Bayesian statistics

# Chapter 2

# Markov Chain Monte Carlo

In this chapter, we will review preliminary, general theory behind Markov chain Monte Carlo (MCMC) methods. We will start with an abstract view before we delve into specific algorithms such as *Gibbs* sampling and *Metropolis-Hastings*. We will then discuss convergence diagnostics and metrics to assess the quality of the samples obtained by the MCMC chain. The field is vast, so we cannot cover every nook and cranny. We will instead focus on the parts that lay the foundation for the main algorithms used in this thesis, namely Hamiltonian Monte Carlo and the No-U-Turn sampler. Thus we will restrict our attention to samples from continuous distributions and ignore the theory for discrete spaces entirely. This choice is one born out of healthy pragmatism. We will not describe these algorithms in this chapter since the require some extra care and thus will have their own chapters devoted to them.

## 2.1 Markov Chain Monte Carlo

A *Monte Carlo Markov chain* (MCMC) method is a scheme to sample points $\theta$ proportional to a distribution $\pi(\theta)$. It generates a new point $\theta_i$ given a point $\theta_{i-1}$. A *Markov chain* is a sequence of points $\theta_1, \theta_2, \ldots,$ that are possibly dependent, but occur in the sequence in proportion to $\pi(\theta)$. Note that $\pi(\theta)$ here is not an exact probability distribution because it need not be normalized to unity. However, suppose $P(\theta)$ is the underlying probability distribution, then $\pi(\theta) \propto P(\theta)$. Typically, in Bayesian applications, we have a prior $P(\theta)$ and a likelihood $P(D|\theta)$. In this case $\pi(\theta) = P(D|\theta)P(\theta)$ and $\pi(\theta) \propto P(\theta|D)$, that is, it's proportional to the posterior distribution.

A few important properties of the Markov chain, originally introduced by Metropolis et. al and built upon by Hastings [1] is worth mentioning:

1. **Ergodicity**. Each point $\theta_i$ is chosen from a distribution that only depends on the previous point in the sequence, $\theta_{i-1}$. For this, we introduce a transition probability that is $T(\theta_i|\theta_{i-1})$. This ensures that any point $\theta$ can eventually be reached given a long enough sequence of samples [2].
2. **Detailed balance**. The transition probability is chosen to obey
$$\pi(\theta)T(\theta'|\theta) = \pi(\theta')T(\theta|\theta'),$$
which ensures that the Markov chain is ergodic. Mathematically, we can express this condition as
$$\pi(\theta') = \int \pi(\theta)T(\theta'|\theta)\mathrm{d}\theta.$$
3. We allow the transition $\theta \to \theta$, hence the transition probability $T(\theta|\theta)$ may be non-zero.
4. The transition probablities integrate to unity, thus
$$\int T(\theta'|\theta)\mathrm{d}\theta = 1,$$
reflecting the notion that some transition is guaranteed to occur.

5. Finally, the transition probabilities are required to be time–independent.

## 2.2 Gibbs sampling

Gibbs sampling [3] is a sampling technique used to generate a Markov chain sequence from an underlying multivariate distribution $P(\gamma)$ for a multi-dimensional parameter $\gamma = (\gamma_1, \dots \gamma_d) \in \mathbb{R}^d$, for $d > 1$. Suppose $\gamma^{(t)}$ represents the parameters at iteration $t$. Then the parameters $\gamma^{(t+1)}$ at iteration $t+1$ are generated from $\gamma^{(t)}$ by the following procedure.

**Algorithm 1** Gibbs sampling

```
procedure GIBBS($\gamma^{(t)}$)
    Sample $\gamma_1^{(t+1)} \sim P(\gamma_1|\gamma_2^{(t)}, ..., \gamma_d^{(t)})$
    Sample $\gamma_2^{(t+1)} \sim P(\gamma_2|\gamma_1^{(t+1)}, ..., \gamma_d^{(t)})$
    $\vdots$    $\vdots$    $\vdots$    $\vdots$
    Sample $\gamma_d^{(t+1)} \sim P(\gamma_d|\gamma_1^{(t+1)}, ..., \gamma_{d-1}^{(t+1)})$
end procedure
```

Thus each new sample $\gamma_i^{(t+1)}$ is only dependent on the prior state of the other parameters through

$$\gamma_i^{(t+1)} \sim P(\gamma_i|\gamma_1^{(t+1)}, \dots \gamma_{i-1}^{(t+1)}, \gamma_{i+1}^{(t)}, \dots, \gamma_d^{(t)}), \tag{2.1}$$

which by definition makes it a Markov chain.

## 2.3 Metropolis-Hastings

The Metropolis-Hastings algorithm [1] is a sampling algorithm based on random walks in parameter space used in MCMC chains to generate a new point $\theta'$ given a point $\theta$. Albeit efficient for some problems, it's not a suitable sampling technique in the context of neural networks. The parameter space of neural networks is high-dimensional. Random walk exploration of said space will yield highly correlated parameters per iteration. The random walk behaviour does not efficiently explore the parameter space. However, a rudimentary understanding of the algorithm will be useful before we embark upon the HMC sampling algorithm, because the final update of the algorithm is by application of the Metropolis-Hastings algorithm.

The transition probability in the Metropolis algorithm is chosen to be

$$T(\theta'|\theta) = q(\theta'|\theta)A(\theta, \theta'), \tag{2.2}$$

where $q(\theta'|\theta)$ is called the proposal distribution and $A(\theta, \theta')$ is the acceptance probability given by

$$A(\theta, \theta') = \min\left(1, \frac{\pi(\theta')q(\theta|\theta')}{\pi(\theta)q(\theta'|\theta)}\right). \tag{2.3}$$

In the Metropolis-Hastings algorithm, a symmetry constraint is imposed on the proposal distribution such that $q(\theta'|\theta) = q(\theta|\theta')$. Thus the acceptance probability reduces to

$$A(\theta, \theta') = \min\left(1, \frac{\pi(\theta')}{\pi(\theta)}\right). \tag{2.4}$$

The point $\theta'$ is accepted with probability $A(\theta, \theta')$.

**Algorithm 2** Metropolis-Hastings

```
procedure METROPOLIS-HASTINGS($\theta$)
    Sample $\theta' \sim q(\theta'|\theta)$
    $p \leftarrow \min\left(1, \frac{\pi(\theta')}{\pi(\theta)}\right)$
    Sample $u$ uniformly on $(0, 1)$.
    if $p \geq u$ then
        $\theta \leftarrow \theta'$                ▷ Accept transition
    else
        $\theta \leftarrow \theta$                 ▷ Reject transition
    end if
end procedure
```

## 2.4 Convergence diagnostics and strategies

### 2.4.1 Burn-in

### 2.4.2 Autocorrelation

### 2.4.3 Mixing

### 2.4.4 The Potential Scale Reduction Factor $\hat{R}$

# Chapter 3

# 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. 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. Moreoever, we will discuss important properties like conservation of the Hamiltonian and local phase space volume.

## 3.1 Hamiltonian dynamics

Hamiltonian dynamics [4] plays a central part in the HMC algorithm. For completeness, we will first survey Lagrangian mechanics from which we derive the Hamiltonian. The Hamiltonian then lays the foundation for Hamiltonian dynamics.

### 3.1.1 Lagrangian Mechanics

Assume a set of *generalized coordinates* $q = (q_1, ..., q_n)$. Generally, the Lagrangian can be written as

$$L(q, \dot{q}, t) = K(q, \dot{q}, t) - V(q, \dot{q}, t), \tag{3.1}$$

where $K$ is the kinetic energy and $V$ is the potential energy of the system. We shall restrict the treatment to the case where there is no explicit dependence on time $t$. The solutions $q(t)$ can be found by solving the *Euler–Lagrange* equations given by

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} = 0. \tag{3.2}$$

### 3.1.2 Hamiltonian Mechanics

The Hamiltonian is constructed by the Legendre transformation,

$$H(q, p, t) = \sum_i p_i \dot{q}_i(q, p) - L(q, \dot{q}(q, p), t), \tag{3.3}$$

where

$$p_i = \frac{\partial L}{\partial \dot{q}_i}. \tag{3.4}$$

The equations of motion, known as *Hamilton's* equations, are given by

$$\frac{\mathrm{d}q_i}{\mathrm{d}t} = \frac{\partial H}{\partial p_i}, \qquad \frac{\mathrm{d}p_i}{\mathrm{d}t} = -\frac{\partial H}{\partial q_i}. \tag{3.5}$$

For the purpose of utilizing this framework in the context of HMC, it's assumed that the form of the Lagrangian is

$$L(q, \dot{q}) = K(\dot{q}) - V(q), \tag{3.6}$$

where

$$K(\dot{q}) = \sum_i \frac{1}{2} m_i \dot{q}_i^2. \tag{3.7}$$

The generalized momentum of coordinate $q_i$ is

$$p_i = \frac{\partial K}{\partial \dot{q}_i} = m\dot{q}_i, \tag{3.8}$$

from which it follows that the Legendre transformed kinetic energy can be written as

$$K(p) = \sum_i \frac{p_i^2}{2m_i}. \tag{3.9}$$

Finally, we can write down the Hamiltonian as

$$H(q, p) = K(p) + V(q) = \sum_i \frac{p_i^2}{2m_i} + V(q). \tag{3.10}$$

From Hamilton's equations in eq. 3.5, we can easily show that the Hamiltonian is conserved in time $t$ by

$$\frac{\mathrm{d}H}{\mathrm{d}t} = \sum_i \left( \frac{\mathrm{d}q_i}{\mathrm{d}t} \frac{\partial H}{\partial q_i} + \frac{\mathrm{d}p_i}{\mathrm{d}t} \frac{\partial H}{\partial p_i} \right) = \sum_i \left( \frac{\partial H}{\partial p_i} \frac{\partial H}{\partial q_i} - \frac{\partial H}{\partial q_i} \frac{\partial H}{\partial p_i} \right) = 0. \tag{3.11}$$

This motives the need for a symplectic integrator, which we will discuss next.

### 3.1.3 Leapfrog integration

To run one step of HMC, we need to compute the time evolution of a Hamiltonian system of the form discussed in the former section, where the neural network parameters will play the role as the generalized coordinates $q$. The common choice of algorithm to integrate the equations of motion in eq. (3.5) is *Leapfrog* integration [5]. This integrator is *symplectic*, which means it conserves local volumes in phase space. This effectively translates to an approximately conserved value of $H(q, p)$ throughout a simulation, with slight oscillations about a mean value.

Assume we approximate the true coordinates and momenta by $(\hat{q}, \hat{p})$. A single Leapfrog integration step can then be written as in algorithm 3. Here $h$ represents the stepsize used in the algorithm.

**Algorithm 3** Leapfrog integration (single step)

```
procedure LEAPFROG($q, p, \lambda$)
    1. $\hat{p}_i(t + h/2) = \hat{p}_i(t) - \lambda \frac{h}{2}\, \partial V(\hat{q}(t))/\partial q_i$
    2. $\hat{q}_i(t + h) = \hat{q}_i(t) + \lambda \frac{h}{m_i} \hat{p}_i(t + h/2)$
    3. $\hat{p}_i(t + h) = \hat{p}_i(t + h/2) - \lambda \frac{h}{2}\, \partial V(\hat{q}(t + h))/\partial q_i$
end procedure
```

### 3.1.4 Conservation of local phase-space volume

## 3.2 Hamiltonian Monte Carlo

Hamiltonian Monte Carlo is largely developed and expanded upon by Radford Neal [6]. The probability distribution to sample from is expressed in terms the Canonical distribution

$$P(q) \propto \exp(-V(q)), \tag{3.12}$$

where $q$ represents the parameters of the model. $V(q)$ can then always be expressed as

$$V(q) = -\log Z - \log P(q) \tag{3.13}$$

for some normalization constant $Z$. This constant plays no actual role in the sampling procedure as we only need to compute the difference $V(q') - V(q)$ at two points $q'$ and $q$. We also need the gradient of $V$ with respect to $q$ where the constant term vanishes. To utilize the framework of Hamiltonian dynamics explained in section 3.1, we introduce momentum variables $p_i$ such that we can express the total Hamiltonian as

$$H(q,p) = K(p) + V(q), \tag{3.14}$$

with a corresponding canonical distribution over phase-space

$$P(q,p) \propto \exp\left(-H(q,p)\right). \tag{3.15}$$

The HMC scheme is summarized in algorithm 4.

**Algorithm 4** Hamiltonian Monte Carlo

```
procedure HMC($L, q, p$)
    Sample $u \sim U(0,1)$.
    $\lambda = 1$  if  $u \geq 1/2$  else  $\lambda = -1$
    $(q^*, p^*) \leftarrow (q, p)$                                  ▷ Start from initial state.
    $p^* \leftarrow$ GIBBS($p^*$)
    for $l = 1, ..., L$ do                                           ▷ $L$ Leapfrog steps.
        $(q^*, p^*) \leftarrow$ LEAPFROG($q^*, p^*, \lambda$)
    end for
    $P = \min\left(1, \exp\left[-\left(H(q^*, p^*) - H(q,p)\right)\right]\right)$
    Sample $u \sim U(0,1)$                                      ▷ Uniform distribution on $(0,1)$.
    if $P \geq u$ then
        $(q, p) \leftarrow (q^*, p^*)$                              ▷ Accept proposed state.
    else
        $(q, p) \leftarrow (q, p)$                                  ▷ Reject proposed state.
    end if
end procedure
```

Chapter 4

# The No U-Turn Sampler

Hamiltonian Monte Carlo (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 $h$. In this chapter, we'll study a sampling method called *The No-U-Turn sampler* [] built upon HMC that dynamically adapts the number of leapfrog steps.

# Chapter 5

# Bayesian Learning for Neural Networks

## 5.1 Neural Networks

In this chapter, we will introduce the mathematical formalism underpinning neural networks. For convenience, we will adopt the terminology used by Tensorflow[7] 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.

### 5.1.1 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 $\ell$'th layer. It consists of $n_\ell$ 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 $a^\ell$ is colloquially called the *activation* of layer $\ell$.
- 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 $\ell$ 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

$$f(x) = (\sigma_L \circ \sigma_{L-1} \circ \cdots \circ \sigma_\ell \circ \cdots \circ \sigma_2 \circ \sigma_1)(x). \tag{5.1}$$

- 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 $\ell$.
- 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 $\ell$ is computed through the recursive equation:

$$a_j^\ell = \sigma_\ell \left( \sum_k W_{jk}^\ell a_k^{\ell-1} + b_j^\ell \right) \equiv \sigma_\ell(z_j^\ell), \quad \text{for} \quad j = 1, 2, ..., n_\ell. \tag{5.2}$$

A special case of eq. (5.2) applies to $\ell = 1$ where $a^0 = x \in \mathbb{R}^p$ is assumed.

### 5.1.2 Backpropagation

The standard approach to train a neural network is by minimization of some loss function by employing the *backpropagation* algorithm[8]. The algorithm boils down to four equations defining a recursive algorithm that approximates the gradient with respect to the parameters of the model. Consider $E$ 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.

$$\Delta_j^L = \frac{\partial E}{\partial z_j^L}. \tag{5.3}$$

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

$$\Delta_j^\ell = \left( \sum_k \Delta_k^{\ell+1} W_{kj}^{\ell+1} \right) \sigma_\ell'(z_j^\ell). \tag{5.4}$$

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

$$\frac{\partial E}{\partial W_{jk}^\ell} = \frac{\partial E}{\partial z_j^\ell} \frac{\partial z_j^\ell}{\partial W_{jk}^\ell} = \Delta_j^\ell a_k^{\ell-1}. \tag{5.5}$$

For the biases, the gradients are

$$\frac{\partial E}{\partial b_j^\ell} = \frac{\partial E}{\partial z_j^\ell} \frac{\partial z_j^\ell}{\partial b_j^\ell} = \Delta_j^\ell. \tag{5.6}$$

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 [9]). 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.

### 5.1.3 Loss function for regression

In this thesis, we are concerned with regression tasks. The activation function of the final layer $\sigma_L$ 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

$$E(y, \hat{y}) = \frac{1}{2} \|y - \hat{y}\|_2^2, \tag{5.7}$$

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,

$$\Delta_j^L = \frac{\partial E}{\partial z_j^L} = a_j^L - y_j. \tag{5.8}$$

We are now equipped to write down the backpropagation for a single datapoint. It's built up of a textitforward pass which takes an input $x$ and applies the recursive eq. (5.2) 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 with respect to the model parameters. The forward pass of the neural network is found in algorithm 5.

**Algorithm 5** Backpropagation: Forward pass

```
procedure ForwardPass($x$)
    $a_j^0 = x_j \quad$ for $\quad j = 1, \dots, p$                ▷ 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.

### 5.1.4 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:

$$E = \frac{1}{2} \sum_i \left\| \hat{y}^{(i)} - y^{(i)} \right\|_2^2 + \frac{\lambda_W}{2} \sum_\ell \left\| W^\ell \right\|_2^2 + \frac{\lambda_b}{2} \sum_\ell \left\| b^\ell \right\|_2^2, \tag{5.9}$$

where $\lambda_W$ and $\lambda_b$ 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*

$$\|A\|_2 = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|A_{ij}|^2}. \tag{5.10}$$

$L^2$-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.

### 5.1.5 Activation functions

## 5.2 Bayesian inference

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

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}. \tag{5.11}$$

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

$$P(\theta|D) \propto P(\theta|D)P(\theta), \tag{5.12}$$

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 $D = \{y^{(1)}, \dots y^{(n)}\}$, this implies

$$P(y^*|D) = \int P(y^*, \theta|D)\mathrm{d}\theta = \int P(y^*|\theta, D)P(\theta|D)\mathrm{d}\theta = \int P(y^*|\theta)P(\theta|D)\mathrm{d}\theta, \tag{5.13}$$

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

## 5.3 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

$$P(y^*|x^*, D) = \int P(y^*|x^*, \theta)P(\theta|D)\mathrm{d}\theta. \tag{5.14}$$

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

$$P(D|\theta) = \prod_{i=1}^{n} P(y^{(i)}|x^{(i)}, \theta). \tag{5.15}$$

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

$$P(y|x,\theta) \propto \exp\left(-\frac{\lambda}{2}\|y - f(x;\theta)\|_2^2\right). \tag{5.16}$$

where $f(x;\theta)$ is the output of the neural network and $\lambda$ is a hyperparameter typically chosen to be identical for all inputs $(x, y)$ during training. The likelihood found eq. (5.16) is equivalent to using $L^2$-norm as a loss function with regularization strength $\lambda$ 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

$$P(\theta|D) \propto P(D|\theta)P(\theta), \tag{5.17}$$

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

$$\hat{y}_i = f(x, \theta_i). \tag{5.18}$$

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

$$\hat{y}_{\text{MLE}} = \frac{1}{n}\sum_i f(x, \theta_i), \tag{5.19}$$

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

$$\text{Cov}(\hat{y}) = \frac{1}{n-1}\sum_i (\hat{y}^{(i)} - \hat{y}_{\text{MLE}})(\hat{y}^{(i)} - \hat{y}_{\text{MLE}})^T. \tag{5.20}$$

The diagonal terms of eq. (5.20) yields the sample *variance* of the components of $\{\hat{y}^{(i)}\}_{i=1}^n$.

## 5.4 Sources of uncertainty

The main motivation utilizing Bayesian inference to neural networks is to be able to quantify uncertainty. Uncertainty can be further divided into two categories:

- **Epistemic uncertainty**: the systematic uncertainty. It practice, this uncertainty can be quantified by the posterior of the model $P(\theta|D)$. Thus it quantifies the uncertainty in the model itself.
- **Aleatoric uncertainty**: the statistical uncertainty, related to the conditional distribution $P(y|x, \theta)$. This uncertainty quantifies the error due to random chance, and measures the random noise in the data.

In principle, in the infinite data limit, both errors goes to zero as there's only a single parametrization $\theta$ that explains all data $D$, and there's only a single output $y$ for a given $x$ and $\theta$. However, neural networks as so-called *over-parametrized* models in the sense that they have many equivalent parametrizations. These occur due to underlying symmetries of the model, two of which are *weight-space symmetry* and *scaling symmetry*. The former means that we can permute parameters in two adjacent and produce the same output as before the permutation. The latter case occur when the network uses non-linear activation that obey $\sigma(\alpha x) = \alpha\sigma(x)$. This relation implies that we can rescale the parameters of two adjacent layers by a factor of $\alpha$ and $1/\alpha$ respectively and produce the same output. Thus a parametrization that explains a set of observations does not exist for sufficiently complex models.

## 5.5 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 $K(p)$. To this end, we need to specify

1. A prior for the network parameters, i.e the kernels $W_{ij}^\ell$ and the biases $b_j^\ell$. A common choice is

$$P(W^\ell) \propto \exp\left(-\frac{\lambda_W}{2}\left\|W^\ell\right\|_2^2\right), \qquad P(b^\ell) \propto \exp\left(-\frac{\lambda_b}{2}\left\|b^\ell\right\|_2^2\right). \tag{5.21}$$

2. A distribution of the generalized momenta of the model. The kinetic energy in eq. (3.9) automatically imposes that
$$P(p) \propto \prod_i \exp\left(-\frac{p_i^2}{2m_i}\right), \tag{5.22}$$
since $K(p) = -\log P(p)$.

3. A likelihood function given an input $(x, y)$,

from which we can define the potential energy as follows:
$$V(W,b) = -\log Z - \sum_\ell \log P(W^\ell) - \sum_\ell \log P(b^\ell) - \sum_i \log P(y^{(i)}|x^{(i)}, W, b), \tag{5.23}$$
where $W$ denotes all the kernels and $b$ denotes all the biases of the model. Here $Z$ denotes the normalization constant of the distributino $P \propto \exp(-V(W,b))$, but is of no importance for the sampling procedure. Inserting the terms from eq. (5.21) and eq. (5.16), we get the following expression for the potential energy.
$$V(W,b) = -\log Z + \frac{\lambda_W}{2} \sum_\ell \left\|W^\ell\right\|_2^2 + \frac{\lambda_b}{2} \sum_\ell \left\|b^\ell\right\|_2^2 + \frac{\lambda}{2} \sum_i \left\|y^{(i)} - f(x^{(i)}; W, b)\right\|_2^2. \tag{5.24}$$
The terms in eq. (5.24) play the same role as in typical machine learning applications. Note that $Z$ here is some appropriate constant that doesn't really matter for the sampling process,

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

# Conclusion

Conclusion here.

# Appendices

# Appendix A

## A.1 Appendix 1 title

Some appendix stuff.

# Bibliography

[1] 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].

[2] W. H. P. et. al, *Numerical recipes in C : the art of scientific computing*, pp. 824–826. Second edition. Cambridge [Cambridgeshire] ; New York : Cambridge University Press, 1992., 1992.

[3] S. Geman and D. Geman, *Stochastic relaxation, Gibbs Distributions, and the Bayesian Restoration of Images*, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **PAMI-6** (1984) 721–741.

[4] J. S. Helbert Goldstein, Charles Poole, *Classical Mechanics, 3rd ed.*, ch. 2,8. Addison Wesley, 2000.

[5] C. M. Bishop, *Pattern Recognition and Machine Learning*, ch. 11, p. 551. Springer New York, 2006.

[6] R. Neal, *Handbook of Markov Chain Monte Carlo*, ch. 5. Chapman and Hall/CRC, May, 2011. http://dx.doi.org/10.1201/b10905. 10.1201/b10905.

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

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

[9] 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) .