SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Converting Neural Networks to Sampled Networks

Dhia Bouassida

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Converting Neural Networks to Sampled Networks

Umwandlung von neuronalen Netzwerken in abgetastete Netzwerke

Author:	Dhia Bouassida
Supervisor:	Dr. Felix Dietrich
Advisor:	Erik Lien Bolager
Submission Date:	16.10.2023

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 16.10.2023 Dhia Bouassida

Abstract

Neural networks have become dominant in machine learning due to their ability to automatically learn complex patterns. They are often trained using iterative gradient-based optimizers, with Adam being one of the most widely used methods. Sampled Networks, introduced by Bolager et al., offer an alternative method where network parameters are constructed directly by sampling data point pairs, eliminating the need for iterative optimization and enhancing efficiency and interpretability.

The primary contribution of this thesis is an algorithm to convert traditionally trained neural networks into equivalent sampled networks, in order to provide the interpretability and transparency that come with these networks. The emphasis is on converting a two-layer neural network employing the ReLU activation function into a sampled network. The main objective for the converted sampled network is to closely match the trained neural network in terms of its parameters, namely weights and biases, as well as its output. For the conversion algorithm, we introduce multiple approaches for converting both the hidden and the output layer parameters of the trained network. Numerical experiments provide a comparative analysis of each proposed approach, comparing the network parameters and outputs between the original trained network and the converted sampled network, as well as the runtime of the conversion algorithm.

Contents

1 Introduction

Machine Learning is a subfield of Artificial Intelligence. It enables computer systems to learn patterns and make decisions without setting up rules or being explicitly programmed [MM97]. A machine learning model uses statistical techniques to draw insights and make predictions or decisions based on given data [DHS01]. Among the various techniques in machine learning, neural networks have gained significant attention due to their versatility and robustness.

Neural networks, inspired by the structure and function of the biological neural circuits of the brain [Ako13], have become dominant in machine learning due to their ability to automatically learn complex patterns from data. They have proven very effective for tasks such as image recognition [Agg18], bioinformatics [LLB09], and medical diagnostics [Ama+13], to name a few examples.

Interconnected neurons are the core component of artificial neural networks. Each neuron receives input from the prior neurons, processes it, and produces an output. This output derives from a weighted sum of the inputs, adjusted by a certain bias, after which a nonlinear activation function is applied. Neurons are organized into sequential layers, with each layer's input being the output of the preceding layer.

The parameters of a neural network are essentially the weights and biases of its neurons. Training a neural network on a set of input and output data involves determining the optimal weights and biases, so that the discrepancy between the neural network's output and the given output is minimal.

Neural networks are often trained using iterative gradient-based optimization methods. These methods start with a random initialization of the network parameters. Iteratively, these parameters are updated to minimize a loss function, which quantifies the difference between the model's predictions using these parameters and the true outputs. One of the most commonly used optimizers is Adam [KB14].

However, this iterative training process can be computationally expensive for large neural networks. Furthermore, neural networks with many hidden layers, also referred to as deep learning models, are often called "black boxes" due to their inherent complexity and the difficulties in understanding their inner workings and mechanisms. This terminology stems from the fact that while we can input data and examine the output, the inner learned relationships and parameters, such as the weights, are not directly interpretable [Cas16].

Sampled networks, proposed by Bolager et al. [Bol+23], offer an alternative method to learning these parameters. Sampled networks construct the weights and biases of a neural network by directly sampling points from the input dataset, instead of relying on iterative optimization. This not only enhances efficiency compared to the traditional gradient-based optimizers but also increases interpretability, as it offers a clearer connection between the model's parameters and the dataset that it is trained on.

In this thesis, we aim to develop an algorithm to convert a traditionally trained neural network into a sampled network. The goal is for the converted sampled network to closely match the original trained network in terms of both its parameters and output. The main motivation is that the converted sampled networks can provide, by design, more insight into the models' parameters, making these models more interpretable and transparent.

Our implementation focuses on converting a traditionally trained neural network with one hidden layer that uses the ReLU activation function into a sampled network. For the conversion, we introduce different techniques to convert the network into a sampled network, by providing multiple approaches to sample data points for determining the sampled network's parameters.

In Chapter 2, we will provide background on neural networks, including their architecture, training methods, and relevant concepts. Chapter 3 will introduce sampled networks, how they are defined, and the algorithm to construct them. Our proposed conversion algorithm will be detailed in Chapter 4, where we present various techniques to sample data points and convert the hidden and output layer parameters. The results from numerical experiments comparing the different proposed approaches will be analyzed in Chapter 5.

2 Neural Networks

Artificial neural networks, also known as ANNs, have recently gained significant attention and popularity due to their learning capabilities and accurate prediction over diverse applications. They have the ability to learn directly from data without being explicitly programmed. This gives them great flexibility in solving problems that are difficult with algorithmic approaches. In this section, we give a brief introduction to their architecture and functioning, as well as some training algorithms.

2.1 Modelling a single neuron

Artificial neural networks are systems inspired by the human brain structure and function [Ako13]. The core components of neural networks are interconnected processing elements called nodes or neurons. Understanding the mechanism of biological neurons can help us comprehend the functioning of neural networks. Biological neurons receive input signals through branch-like structures called dendrites. The dendrites detect neurotransmitters released from other neurons. The neuron then internally processes the input signals using its biological mechanisms. If the input is sufficiently strong, the neuron will fire an output signal, neurotransmitters are then released. These neurotransmitters can then be picked up by dendrites on other neurons, propagating the signal through the network [Ako13]. Analogously, artificial neurons are the computational units of an artificial neural network that receive a set of inputs, process them, and output a signal to other neurons.

Each node in an artificial neural network receives its input from other nodes, and its computation is typically broken into a linear combination of inputs and a simple typically non-linear activation function. Formally, an artificial neuron $\mathcal{N}$ performs a mapping from an n-dimensional input feature vector to a real number, as $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}$. This process involves taking each feature in the input vector and multiplying it by a corresponding weight. The products are then summed up, a bias is subtracted, and the resulting sum is passed through an activation function ϕ to give the final output. The output is given by

$$\phi\left(\sum_{i=1}^{n} x_i w_i - b\right),$$

where n is the number of features in the input vector x, b is the bias, w_i's are the weights, and ϕ is the activation function. A model of a single neuron is presented in Figure 2.1. The activation function ϕ is a continuous, typically non-linear function $\phi : \mathbb{R} \to \mathbb{R}$. Common activation functions are the following:

- *Sigmoid*: The sigmoid activation function is a smooth function $\phi_{\text{Sigmoid}} : \mathbb{R} \to (0, 1)$. Its graph has an S-shape that is similar to the biological S-shaped growth curve [SSA20]. The mathematical expression is

$$\phi_{\text{Sigmoid}}(x) = \frac{1}{1 + e^{-x}}.$$

- *tanh*: The hyperbolic tangent function $\phi_{tanh} : \mathbb{R} \to (-1, 1)$ is similar to the sigmoid function but it is symmetric to the origin [SSA20]. It is defined as

$$\phi_{tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

- *ReLU*: ReLU stands for Rectified Linear Unit activation function. The ReLU function $\phi_{ReLU} : \mathbb{R} \to \mathbb{R}_{\geq 0}$ is recognized as one of the most commonly used activation functions in neural networks today [SSA20]. The function is defined by

$$\phi_{\text{ReLU}}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}.$$

The nonlinearity of these functions ensures that the system can learn complex patterns and relationships between the inputs and outputs. In each neuron of a neural network, a weighted sum of the inputs is applied. If, hypothetically only these linear operations were performed, the entire system would remain linear, since consecutive linear operations are performed by sequential neurons. Consequently, the system would not be able to represent more complex structures or learn intricate patterns in the data. This is where activation functions come into play.

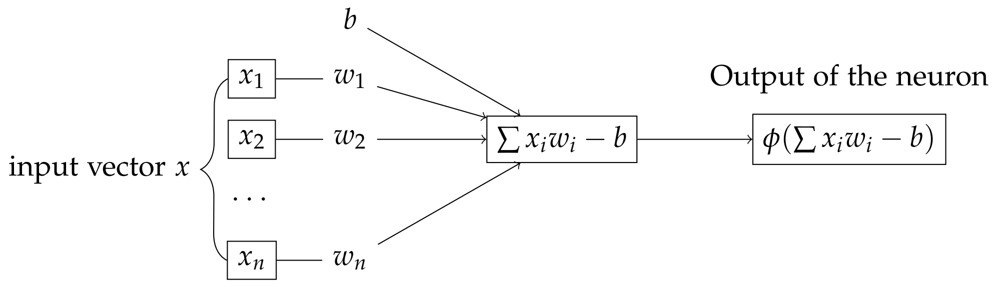

Figure 2.1: A simple diagram illustrating the structure of a single neuron

2.2 Multilayered Neural Networks

The nodes in a neural network are arranged in a sequential order called layers, where the output of a layer is the input of the next layer. A neural network typically consists of three main types of layers: input, hidden, and output layers.

The input layer receives the raw data from the input and propagates it further into the network. Each node or neuron in the input layer corresponds to a single feature of the data.

The hidden layers are intermediate layers between the input and the output. Depending on the problem's complexity and the network's design, both the number of hidden layers and the number of neurons in each layer can vary. Hidden layers convert the inputs into more abstract representations. Through a series of non-linear transformations made possible by the neurons' activation functions, they are able to capture and learn complex patterns and features from the input data. The output of each hidden layer serves as the next layer's input, enabling the network to create a hierarchical understanding of the data. This computation is later forwarded to the output layer, which produces the prediction/classification.

In the following, we establish certain notations and present a mathematical framework that will be utilized throughout this paper. We denote the neural network as Φ. We denote the number of its hidden layers as L, which means the total number of layers, including the input and output layers, equals $L + 2$. Let N_l be the number of neurons in each layer, where $l \in \{0, 1, 2, \cdots, L + 1\}$.

We use $\mathcal{X} \subseteq \mathbb{R}^D$ to denote the input space. Consequently, it follows that the number of neurons in the input layer, N_0, equals the dimension D of the input space.

The output of the lth layer of this neural network can be described as a mapping

$\Phi^{(l)} : \mathbb{R}^{N_0} \to \mathbb{R}^{N_l}$. We denote $\Phi^{(0)}(x) = x$. For the hidden layers, given an input $x \in \mathbb{R}^{N_0}$, the output of the lth layer $\Phi^{(l)}(x) \in \mathbb{R}^{N_l}$ is

$$\Phi^{(l)}(x) = \phi\left(\Phi^{(l-1)}(x)W_l - b_l\right),$$

where $W_l \in \mathbb{R}^{N_{l-1} \times N_l}$ is the weight matrix for the lth layer, $b_l \in \mathbb{R}^{N_l}$ is the bias vector for the lth layer, and ϕ is the activation function.

The process of obtaining the output of the neural network $\Phi^{(L+1)}$ from the input layer is referred to as a forward pass. During this pass, the network takes the input vector $x \in \mathbb{R}^{N_0}$ and transforms it iteratively through each layer to produce the final output $\Phi^{(L+1)}(x) \in \mathbb{R}^{N_{L+1}}$. More concretely, the output of the neural network is

$$\Phi^{(L+1)}(x) = \phi\left(\phi\left(\cdots \phi\left(xW_1 - b_1\right) W_2 - b_2\right) \cdots\right) W_{L+1} - b_{L+1}.$$

Note that in out model the activation function is not applied on the output layer. Figure 2.2 shows the hierarchy of an example of a multi-layered neural network.

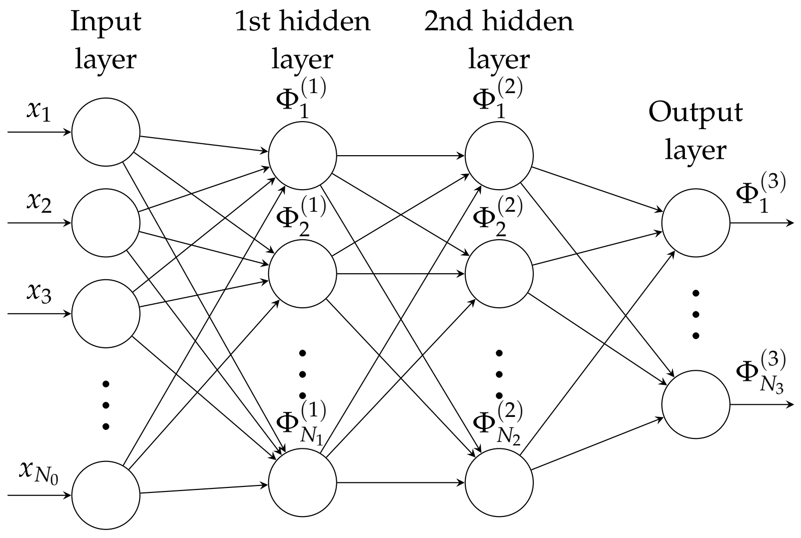

Figure 2.2: This diagram illustrates the hierarchy of layers in a neural network.

2.3 Training Neural Networks

Training a neural network on a dataset that consists of input-output pairs (x_i, y_i), where each $x_i \in X$ (a subset of the input space $\mathcal{X}$) is associated with an output $y_i \in Y$, involves learning the general patterns and relationships between inputs and their respective outputs. This enables the system to generate automated outputs for inputs not present in the training dataset. To achieve this, the neural network learns the optimal parameters W_l and b_l for each layer such that the output of the system closely matches the original target outputs. The discrepancy between the system's predictions and the target outputs is measured using a loss function. For regression tasks, for example, the mean squared error (MSE) is commonly used, while for classification problems, cross-entropy loss is used. In this section, we provide examples of algorithms for neural network training.

2.3.1 Classical Gradient Descent

Training a neural network entails finding the parameters W_l and b_l of each layer, such that the loss between the network's output and the actual data is minimized. An important algorithm for learning these weights and biases is gradient descent, which was first introduced by Rumelhart, D.R. et al. [RHW86]. Gradient descent works as follows: The gradient of the loss function with respect to the weights and biases of the network is calculated using the chain rule. The negative direction of this gradient indicates the direction of steepest descent. Taking a step in that direction goes into the direction of minimizing the loss of the function [RHW86]. The general formula for this algorithm is

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \alpha \nabla \mathcal{L}(\boldsymbol{w}^{(t)}), \quad t = 0, 1, 2, \ldots, \tag{2.1}$$

where $\boldsymbol{w}^{(t)}$ refers to the parameter vector $\boldsymbol{w}$ at iteration t, $\mathcal{L}$ refers to the loss function that we are trying to minimize and takes $\boldsymbol{w}$ as parameter, $\nabla \mathcal{L}(\boldsymbol{w}^{(t)})$ calculates the gradient of the function $\mathcal{L}$ with respect to $\boldsymbol{w}$ at the current parameters $\boldsymbol{w}^{(t)}$, and α is the learning rate. The learning rate α controls the length of the step taken in each iteration. In other words, we update the parameters, $\boldsymbol{w}$, of the network by taking a step in the negative gradient direction $\mathcal{L}(\boldsymbol{w}^{(t)})$, that is the direction of the steepest descent.

However, classical gradient descent has several limitations. First, for a function with multiple local minima, the gradient descent may get trapped in one local minimum and thus not reach the global minimum, since the iterations stop when the gradient's magnitude is zero or below a certain threshold. The same is true for saddle points, since the gradient at these points is zero and they do not represent minima. In larger dimensions, saddle points occur more frequently than local minima, and the gradient descent can be blocked in these cases. In addition, the gradient descent algorithm can suffer from slow convergence. The reason is that the learning rate α is constant during the iterations [GBC16].

2.3.2 Adam

Several optimizers, primarily based on the concept of gradient descent, have been developed to enhance performance and address issues associated with the classical gradient descent method used in training neural networks. A relevant algorithm in our paper is the Adam (short for Adaptive Moment Estimation) optimization algorithm [KB14]. Essentially, it combines the benefits of two previous extensions of gradient descent: Momentum [Pol64] and RMSProp [HSS12]. Adam addresses issues such as noise and sparsity in gradients, meaning that the gradients have a lot of unwanted fluctuations or that most of the elements of the gradient vector are zero, both of which are known to hinder the performance of the classical gradient descent method. Adam also utilizes an adaptive learning rate α during training iterations to make the search for the minimum more effective[KB14].

For this, Adam maintains two moving averages for each parameter to adaptively update the learning rates during training. The first one is for the gradients and the second one is for the square of gradients. Moreover, it has hyperparameters that control the exponential decay rates of these moving averages. The update rule of Adam follows

$$
\begin{aligned}
m^{(t)} &= \beta_1 m^{(t-1)} + (1-\beta_1)\nabla\mathcal{L}(\boldsymbol{w}^{(t)}),\\
v^{(t)} &= \beta_2 v^{(t-1)} + (1-\beta_2)(\nabla\mathcal{L}(\boldsymbol{w}^{(t)})^2),\\
\hat{m}^{(t)} &= \frac{m^{(t)}}{1-\beta_1^t},\\
\hat{v}^{(t)} &= \frac{v^{(t)}}{1-\beta_2^t},\\
\boldsymbol{w}^{(t+1)} &= \boldsymbol{w}^{(t)} - \alpha\frac{\hat{m}^{(t)}}{\sqrt{\hat{v}^{(t)}}+\epsilon}.
\end{aligned}
$$

Here, $\boldsymbol{w}^{(t)}$ represents the parameters of the model at the tth step and $\nabla\mathcal{L}(\boldsymbol{w}^{(t)})$ is the gradient of the loss function with respect to the parameters at the tth step. $m^{(t)}$ is the estimate of the first moment of the gradients, which is computed as a decaying average of past gradients. $v^{(t)}$ is the estimate of the second moment of the gradients, which is computed as a decaying average of the squared gradients. The decay rate in $m^{(t)}$ and $v^{(t)}$ are respectively β_1 and β_2. $\hat{m}^{(t)}$ and $\hat{v}^{(t)}$ are bias-corrected versions of the first and second moment estimates, and α is the learning rate. And finally ϵ is a small constant to prevent division by zero.

In this chapter, a general introduction to neural networks and optimizers was given. In the following chapter, we will discuss a novel method of finding the parameters of a neural network that is not based on gradient descent.

3 Sampled Networks

Artificial neural networks, as mentioned in the previous chapter, have mostly been trained using iterative gradient-based methods to find and optimize weights and biases. The sampled networks approach, introduced by Bolager E. et al. in the paper "Sampling weights of deep neural networks," represents a paradigm shift [Bol+23]. It allows for more efficient construction of neural networks without relying on iterative optimization of internal network parameters. The method uses both input and output training data points from the supervised learning problem to construct the weights and biases of the network.

3.1 Introduction to Sampled Networks

At its core, a sampled network links data points from the input to the weights and biases of hidden layers in the network. Instead of randomly initializing and iteratively optimizing the weights and biases, a sampled network uses pairs of data points from the input space to determine these parameters. This data-driven approach ensures the constructed network is inherently connected to the given dataset.

This method, which directly links data points to the network's parameters, offers several benefits [Bol+23]:
Efficiency and accuracy: Sampled networks avoid extensive iterative optimization, allowing for quicker construction of deep neural networks. And compared to data-agnostic sampling methods, this sampling schema gives a more accurate and width-efficient approximation.
Interpretability: With weights and biases derived directly from data points, sampled networks offer a clearer connection between both the model's decisions and functioning and the dataset it is trained on. This helps with understanding the model's workings and reasoning.

3.2 Constructing Sampled Networks

In a sampled network, the weight vector and bias of every neuron in each hidden layer are determined by two distinct points from the input space. The weight is derived from the difference between these two points divided by the square of their distance. The bias is computed as the inner product of the derived weight and one of the two points. This relationship between the weights, biases, and data points forms the foundation for sampled networks. After the weights and biases of all hidden layers are established, the only remaining task is to solve an optimization problem to find the parameters of the final linear layer. In the following, we cite the formal definition of a sampled network and present the proposed algorithm for creating one [Bol+23].

3.2.1 Definition of a Sampled Network

A neural network Φ with L hidden layers, having an input space $\mathcal{X} \subseteq \mathbb{R}^D$, is termed a sampled network if, for each layer l ranging from 1 to L and for every neuron i from 1 to N_l, the weights and biases are defined by pairs of data points, $(x_{0,i}^{(1)}, x_{0,i}^{(2)})$, sampled from $\mathcal{X} \times \mathcal{X}$, as

$$w_{l,i} = s_1 \frac{x_{l-1,i}^{(2)} - x_{l-1,i}^{(1)}}{\left\| x_{l-1,i}^{(2)} - x_{l-1,i}^{(1)} \right\|^2}, \qquad b_{l,i} = \left\langle w_{l,i}, x_{l-1,i}^{(1)} \right\rangle + s_2,$$

where s_1 and s_2 are scalar constants, $x_{l-1,i}^{(1)}$ and $x_{l-1,i}^{(2)}$ are $\Phi^{(l-1)}\left(x_{0,i}^{(1)}\right)$ and $\Phi^{(l-1)}\left(x_{0,i}^{(2)}\right)$, respectively, and $x_{l-1,i}^{(1)} \neq x_{l-1,i}^{(2)}$. The weights and biases of the output layer, W_{L+1} and b_{L+1}, are chosen to minimize a designated loss function $\mathcal{L}$.

The scalars s_1 and s_2 influence the mapping of the points after applying the activation function. For ReLU, s_1 and s_2 are set as $s_1 = 1$ and $s_2 = 0$, causing $x^{(1)}$ to map to zero and $x^{(2)}$ to one. With *tanh*, the values are $s_1 = 2s_2$ and $s_2 = \frac{\ln(3)}{2}$, mapping $x^{(1)}$ and $x^{(2)}$ to $-\frac{1}{2}$ and $\frac{1}{2}$ respectively, with their midpoint to zero.

3.2.2 SWIM Algorithm

The proposed method to create sampled networks is named "Sampling Where It Matters" or SWIM. The name suggests that the method focuses on effectively sampling the input space in regions where it is crucial for learning. For each hiddel layer l, a conditional probability distribution $P^{(l)}$ over pairs $(x^{(1)}, x^{(2)})$ from $\mathcal{X} \times \mathcal{X}$ is constructed.

This probability distribution guides which pairs are more likely to be selected. $P^{(l)}$ favors pairs of points that are close in the representation space in the l^{th} layer but have a significant difference in the true output value. More precisely, for $l = 1, 2, \dots, L$ and $x_0^{(1)}, x_0^{(2)} \in \mathcal{X}$, the probability distribution of $P^{(l)}$ is defined by its conditional distribution with density $p^{(l)}$ as

$$p^{(l)}\left(x_0^{(1)}, x_0^{(2)} \mid \{W_j, b_j\}_{j=1}^{l-1}\right) \propto \begin{cases} \frac{\left\|y^{(2)}-y^{(1)}\right\|}{\left\|x_{l-1}^{(2)}-x_{l-1}^{(1)}\right\|} & x_{l-1}^{(1)} \neq x_{l-1}^{(2)} \\ 0, & \text{otherwise} \end{cases},$$

where $y^{(1)}$ and $y^{(2)}$ are the true outputs of $x_0^{(1)}$ and $x_0^{(2)}$ respectively.

In other words, the probability density $P^{(l)}$ for a pair $(x^{(1)}, x^{(2)})$ in a layer l is proportional to the difference in the output values between the two points, divided by the difference in their representations in the previous layer of the network. This results in giving more importance to points that are close but differ a lot with respect to their output.

A simplified algorithm for the SWIM method is given in Algorithm 1. The input of the algorithm is an input-output dataset (X, Y) with size M, where $X \subseteq \mathcal{X}$. For each hidden layer l, the function ComputeProbabilityDistribution calculates the probability distribution $P^{(l)}$ based on the input and output points and the representation of the input points in the previous layer. A data pair is then sampled using that probability distribution, and the weights and biases are computed based on that pair. Once all the hidden layers are processed, the output of the last hidden layer of the network, $\Phi^{(L)}$, is calculated. Then the parameters of the output layer are constructed to minimize the loss function $\mathcal{L}$, which is a measure of the discrepancy between the actual output values Y and those predicted by the network $\Phi^{(L+1)} = \Phi^{(L)}(X)W_{L+1} - b_{L+1}$. This involves determining the optimal weights W_{L+1} and biases b_{L+1} that minimize $\mathcal{L}$. A typical loss function is the Mean Squared Error (MSE).

Algorithm 1: The SWIM algorithm for an activation function ϕ and a loss function $\mathcal{L}$.

```
Data: $X = \{x_i : x_i \in \mathbb{R}^D, i = 1, 2, \dots, M\}$, $Y = \{y_i : y_i \in \mathbb{R}^{N_{L+1}}, i = 1, 2, \dots, M\}$
Result: $\{W_l, b_l\}_{l=1}^{L+1}$
Constant: $L \in \mathbb{N}_{>0}, \{N_l \in \mathbb{N}_{>0}\}_{l=1}^{L+1}$, and $s_1, s_2 \in \mathbb{R}$;
$\Phi^{(0)}(x) = x$;
for $l = 1, 2, \dots, L$ do
    $P^{(l)} = \textsc{ComputeProbabilityDistribution}(\Phi^{(l-1)}, X, Y)$;
    $W_l \in \mathbb{R}^{N_{l-1}, N_l}, b_l \in \mathbb{R}^{N_l}$;
    for $i = 1, 2, \dots, N_l$ do
        Sample $(x^{(1)}, x^{(2)})$ from $X \times X$, with probability proportional to $P^{(l)}$;
        $x^{(1)}_{l-1,i}, x^{(2)}_{l-1,i} \leftarrow \Phi^{(l-1)}(x^{(1)}), \Phi^{(l-1)}(x^{(2)})$;
        $W_l^{(i,:)} \leftarrow s_1 \frac{x^{(2)}_{l-1,i} - x^{(1)}_{l-1,i}}{\left\| x^{(2)}_{l-1,i} - x^{(1)}_{l-1,i} \right\|^2}$;
        $b_l^{(i)} \leftarrow \langle W_l^{(i,:)}, x^{(1)}_{l-1,i} \rangle + s_2$;
    end
    $\Phi^{(l)}(\cdot) \leftarrow \phi(\Phi^{(l-1)} W_l(\cdot) - b_l)$;
end
$W_{L+1}, b_{L+1} \leftarrow \arg\min \mathcal{L}(\Phi^{(L)}(X) W_{l+1} - b_{L+1}, Y)$;
```

4 Converting Neural Networks to Sampled Networks

In this Chapter, we first start by giving the motivation for converting neural networks into sampled networks. We then present an algorithm for the conversion process. Our implementation focuses on the case where the given network comprises only one hidden layer and one output layer, and uses the ReLU activation function in the hidden layer.

4.1 Motivation

Neural networks are often referred to as "black boxes" due to their complexity and the difficulties in understanding their inner workings and mechanisms. While these models have achieved an increasing performace, this accuracy has been achieved thanks to advanced optimizing algorithms. By converting these traditionally trained neural networks into sampled networks, we leverage the properties of the sampled networks to provide more transparency and interpretability to the trained networks. Specifically, sampled networks adopt a data-driven approach where the weights and bias for each neuron are constructed using a specific data pair from the input dataset. By definition, for a neuron, the weight is a constant times the difference between these two points divided by the squared norm of their difference. Meanwhile, the bias is the dot product between this sampled weight vector and one of the points of the pairs, added to a constant. This data-driven approach provides a more intuitive connection between the model's internal structures and the dataset it was trained on. In addition, with sampled networks we gain more insights into the inner workings of a neuron to receive its output. In fact, for a regression problem using the ReLU activation function, the sampled pair points $x^{(1)}$ and $x^{(2)}$ correspond to values of zero and one, respectively. This means that $x^{(1)}$ directly defines the activation boundary of the neuron, while other points are linearly interpolated between the two points. For a classification problem using *tanh*, the images of the points $x^{(1)}$ and $x^{(2)}$ after activation are $-1/2$ and $1/2$, while the midpoint between the two points is zero. This implies that a boundary is constructed if $x^{(1)}$ belongs to a different class than $x^{(2)}$ [Bol+23].

4.2 Implementation

The conversion process is detailed in Algorithm 2. The input is a traditionally trained neural network having two layers: a hidden layer and an output layer. These layers are defined by the weights and biases represented as $\{w_{l,i}, b_{l,i}\}_{l=1,i=1}^{2,N_l}$, where N_l indicates the number of neurons in the l^{th} layer. The sampling dataset is defined as $X = \{x_i : x_i \in \mathbb{R}^D, i = 1, 2, \dots, M\}$, where M is the number of samples, and D, equivalent to N_0, represents the dimension of the input space.

For each neuron i in the hidden layer, the algorithm calls the FindDataPair function to select a pair of data points from the input dataset. It then calculates new weights and biases, $\hat{w}_{1,i}$ and $\hat{b}_{1,i}$, using the formulas defined for sampled networks in Section 3.2.1. The objective of FindDataPair is to choose data points such that the resultant weights and biases of the sampled network align closely with the trained ones. For the output layer, the algorithm calls UpdateSecondLayer, to adjust the parameters based on the newly computed sampled weights and biases and the trained neural network parameters.

Algorithm 2: The algorithm for creating a Sampled Network from an traditionally trained network.

```
Data: $X = \{x_i : x_i \in \mathbb{R}^D, i = 1, 2, \dots, M\}, \{w_{l,i}, b_{l,i}\}_{l=1,i=1}^{2,N_l}$
for $i = 1, 2, \dots, N_1$ do
    $x^{(1)}, x^{(2)} \leftarrow$ FindDataPair$(X, w_{1,i}, b_{1,i})$;
    $\hat{w}_{1,i} \leftarrow \frac{x^{(2)} - x^{(1)}}{\left\| x^{(2)} - x^{(1)} \right\|^2}$;
    $\hat{b}_{1,i} \leftarrow \left\langle x^{(1)}, \hat{w}_{1,i} \right\rangle$;
end
$\{\hat{w}_{2,j}, \hat{b}_{2,j}\}_{j=1}^{N_2} \leftarrow$UpdateSecondLayer$(X, \{\hat{w}_{1,i}, \hat{b}_{1,i}\}_{i=1}^{N_1}, \{w_{l,i}, b_{l,i}\}_{l=1,i=1}^{2,N_l})$;
return $\left\{\hat{w}_{l,i}, \hat{b}_{l,i}\right\}_{l=1,i=1}^{2,N_l}$
```

4.2.1 Sampling for the Hidden Layer

The core of the conversion algorithm is to sample adequate points for each neuron, which are then used to determine the weights and biases of the sampled network. Our requirements are that the sampled weights and biases, $\hat{w}_{1,i}$ and $\hat{b}_{1,i}$, closely align with the trained ones. Concretely, for instance, we desire that the sampled weights align both in direction and in norm with the trained ones. Furthermore, the bias of each neuron, $\hat{b}_i$, should be close to the original bias b_i. The method employed to achieve this lies in the FINDDATAPAIR function. In this section, we will introduce various strategies to sample data points and construct sampled weights and biases by providing multiple implementations for the function FINDDATAPAIR. We will introduce different methods while focusing on both accuracy and performance.

4.2.1.1 Lowest Angle Approach

This strategy identifies a pair $(x^{(1)}, x^{(2)})$ based on the provided weights $w_{1,i}$ and biases $b_{1,i}$ of a specific node i, while giving constraints for both points. The general goal is that the resulting sampled weight $\hat{w}_{1,i}$ for each node closely aligns in direction with $w_{1,i}$, and that the bias $\hat{b}_{1,i}$ is close to $b_{1,i}$. The process is described in Algorithm 3.

Algorithm 3: The function FINDDATAPAIR samples a data pair for a neuron i in the first layer following the lowest angle approach.

```
Function FindDataPair($X, w_{1,i}, b_{1,i}$):
    Find $x^{(1)} \in X$ which minimizes $\{|\langle x, w_{1,i}\rangle - b_{1,i}| : x \in X\}$;
    Find $x^{(2)} \in X \setminus \left\{x^{(1)}\right\}$ which minimizes $\left\{\cos^{-1}\left(\frac{\langle x - x^{(1)}, w_{1,i}\rangle}{\|x - x^{(1)}\| \|w_{1,i}\|}\right) : x \in X\right\}$;
    return $x^{(1)}, x^{(2)}$;
```

Finding $x^{(1)}$: Algorithm 3 selects $x^{(1)}$ from the dataset X such that its dot product with the given weight minus the bias is minimized in absolute terms. This ensures that the output of the neuron for $x^{(1)}$, prior to applying the activation function, is as close to zero as possible. For the ReLU activation function, zero is the transition point from the activation or inactivation state of the neuron. That means that $x^{(1)}$ is the point nearest to this transition point.

Let us further visualize this from another perspective. For a certain node i in the first layer, let H be the hyperplane defined by

$$H := \{x \in \mathbb{R}^{N_0} : \langle x, w_{1,i}\rangle - b_{1,i} = 0\}.$$

As the neuron uses ReLU activation function, H represents the activation boundary of the neuron. This hyperplane partitions the input space into two regions, activating or not activating the neuron. The Euclidean distance from a point x to the hyperplane H

is given by the general formula

$$d(x,H) = \frac{|\langle x, w_{1,i}\rangle - b_{1,i}|}{\|w_{1,i}\|}.$$

Thus, finding $x^{(1)} \in X$ which minimizes $\{|\langle x, w_{1,i}\rangle - b_{1,i}| : x \in X\}$ is in essence finding $x^{(1)}$ from the input space that is the closest in Euclidean distance to the activation boundary H of the neuron, since $\|w_{1,i}\|$ is a positive constant.

In Figure 4.1, an explanatory example containing a sample dataset of four points and their projections to the activation boundary H is shown. In the upper part of Figure 4.2, we can see the images of the points before the activation, and thus visualize the relationship to the distance to H.

Finding $x^{(2)}$: In general, the output of a node involves calculating the weighted sum of its inputs, which is the dot product between the input vector and the weight vector associated with that node. When the input vector is more aligned with the direction of the weight vector, it will produce a larger output. Thus, the direction of a weight vector plays an important role in calculating the output as it indicates which aspects of the input the node emphasizes or is sensitive to.

Given a specific vector $x^{(1)}$, we aim to find the second point $x^{(2)}$ such that the vector difference $x^{(2)} - x^{(1)}$ has the smallest angle with respect to the weight vector $w_{1,i}$. To achieve this, we find the point $x^{(2)} \in X \backslash \{x^{(1)}\}$ which minimizes $\{\cos^{-1}\left(\frac{\langle x - x^{(1)}, w_{1,i}\rangle}{\|x - x^{(1)}\|\|w_{1,i}\|}\right) : x \in X\}$. The term inside the cosine inverse, $\cos^{-1}$, represents the cosine similarity between the vector $x - x^{(1)}$ and $w_{1,i}$. By minimizing this expression, we ensure that the direction from $x^{(1)}$ to x aligns closely with the direction of the given weight $w_{1,i}$.

By finding an $x^{(2)}$ that satisfies the last condition and using the updated weight $\hat{w}_{1,i} = \frac{x^{(2)} - x^{(1)}}{\|x^{(2)} - x^{(1)}\|^2}$, we ensure that $\hat{w}_{1,i}$ remains closely aligned in direction with the original weight vector $w_{1,i}$. In Figure 4.2, after choosing the point D as $x^{(1)}$, we choose the point B as $x^{(2)}$, because the resulting sampled weight is the closest in angle to the original weight.

As for the bias, it is calculated according to the definition of a sampled network as $\hat{b}_{1,i} = \langle x^{(1)}, \hat{w}_{1,i}\rangle$. This results in the point $x^{(1)}$ now getting mapped to 0 before the activation, indicating a shift in the activation boundary in comparison with the trained network. To minimize that shift, we choose the point $x^{(1)}$ in Algorithm 3 as the closest point to the original activation boundary.

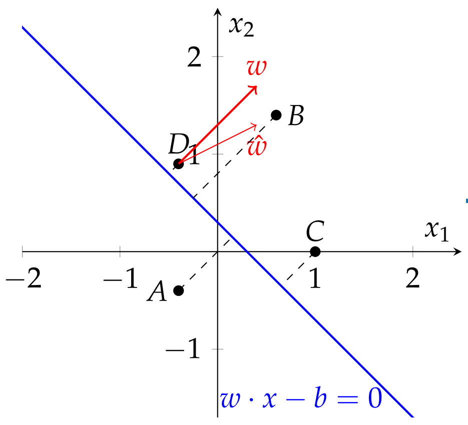

Figure 4.1: A two dimentional input dataset is plotted. The blue line is the activation boundary of the neuron. The distances to that activation boundary, as well the original and the sampled weight are shown.

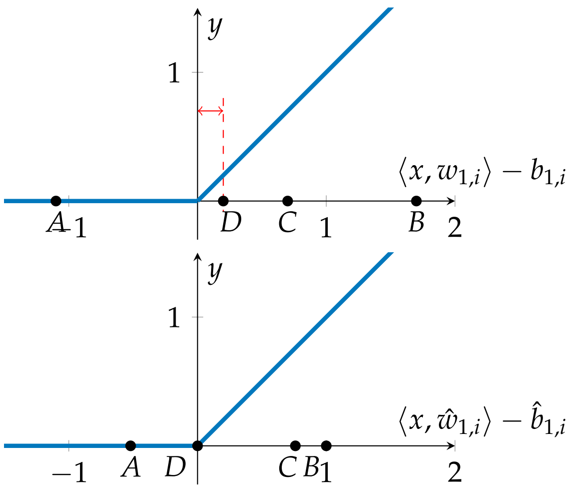

Figure 4.2: The upper plot shows the images of the input points in the original trained network. The lower plot shows the images in the sampled network. The red arrows in the upper plot shows the bias shift that happens in the conversion process.

4.2.1.2 Threshold Ratio and Weight Norm Preservation Approach

In this strategy, we explore an alternative approach to create sampled weights and biases that better align with the original parameters. At first, we extend the method of finding an adequate point $x^{(1)}$. We then propose a method to find the second point $x^{(2)}$, such that the resulting sampled weight does not only match the given weight in direction, but also in magnitude.

Threshold ratio r: Selecting a single point $x^{(1)}$, that is the closest point to the activation boundary, might not always yield an optimal solution. This selection could place $x^{(1)}$ in regions of the dataset with sparse data distributions along the direction of the weight vector $w_{1,i}$, potentially misrepresenting the overall geometry and directionality of the data relative to $w_{1,i}$.

To account for this, we introduce a candidate set X' that contains the points close to the activation boundary. To define this set, we use a threshold ratio $r \in [0, 1]$. X'

contains the points x from the dataset X that lie within a specified range given by $x \in X : |\langle x, w_{1,i} \rangle - b_{1,i}| < r\mathcal{M}$, where $\mathcal{M} = \max_{x \in X} |\langle x, w_{1,i} \rangle - b_{1,i}|$. Here, $\mathcal{M}$ serves as a normalizing factor, representing the maximum absolute value of the output of neuron i before activation across all points in X. In essence, the set X' contains the points that lie within a distance $r\mathcal{M}$ to the activation boundary. It provides a pool from which we can iteratively select the most suitable point $x^{(1)}$.

Introducing the threshold ratio r adds a layer of flexibility to choosing the point $x^{(1)}$. A smaller r might be overly restrictive, reverting to the initial problem. However, a larger r, which is closer to 1, might be too permissive, adding irrelevant data points and moving too far from the activation boundary, resulting in a sampled bias very different from the original one.

Normalization concerns: The sampled weight $\hat{w}_{1,i}$ of each neuron is computed as the difference between two samples $x^{(2)}$ and $x^{(1)}$ divided by the squared norm of their difference. Following the lowest angle strategy, this new weight vector is in a close direction to the original weight vector $w_{1,i}$. However, it does not guarantee the preservation of its magnitude. A significant difference in magnitude between the new and original weights can alter the network's dynamics.

While the direction of a weight vector indicates the direction in the input space it emphasizes, its magnitude or norm determines the strength of this emphasis. In this second strategy, we aim to select $x^{(2)}$ such that the resultant sampled weight vector closely aligns in both angle and magnitude with the trained one. This is achieved by minimizing the norm of their difference, $\|\hat{w}_{1,i} - w_{1,i}\|$. That means we want each element of the two vectors to be as close as possible. Another way to visualize this, is that the distance between the tips of the sampled and original trained weight vectors should be as low as possible.

Algorithm 4 describes this approach. First, we create a subset X' that contains the data points lying within the threshold ratio r. To account for potentially small values of r that might result in an empty subset X', we always include $x_{\min}$, the point closest to the activation boundary. The algorithm then iterates over all pairs $(x^{(1)}, x^{(2)})$ where $x^{(1)}$ is in X' and $x^{(2)}$ is in $X \setminus \{x^{(1)}\}$. Out of all these combinations, it identifies the weight vector that best approximates the trained $w_{1,i}$. Finally, the algorithm returns the pair of points that provide a sampled weight that closely aligns to the original weight.

In Figure 4.3, the difference between choosing $r = 0$, shown on the left plot, and

$r = 0.05$, shown on the right plot, is illustrated. On the right plot, for example, we see that, for the seventh and third vectors, the sampled points are different from the left plot. Specifically, for the seventh weight vector, the direction of $x^{(2)} - x^{(1)}$ on the right plot is more aligned with the given weight vectors compared to the left plot.

Algorithm 4: This function finds a data pair such that the resulting sampled weight is closest in distance to the given one.

```
Function FindDataPair($X, w_{1,i}, b_{1,i}, r$):
    $\mathcal{M} = \max_{x \in X} |\langle x, w_{1,i} \rangle - b_{1,i}|$;
    $x_{min} \leftarrow \arg\min_{x \in X} |\langle x, w_{1,i} \rangle - b_{1,i}|$;
    $X' \leftarrow \{x \in X : |x \cdot w_{1,i} - b_{1,i}| < r\mathcal{M}\} \cup \{x_{min}\}$;
    $min\_distance \leftarrow \infty$;
    for each $x^{(1)}$ in $X'$ do
        for each $x^{(2)}$ in $X \setminus \{x^{(1)}\}$ do
            $\hat{w}_{1,i} \leftarrow \frac{x^{(2)} - x^{(1)}}{\|x^{(2)} - x^{(1)}\|^2}$;
            $distance \leftarrow \|\hat{w}_{1,i} - w_{1,i}\|_2$;
            if $distance < min\_distance$ then
                $min\_distance \leftarrow distance$;
                $x^{(1)}_{\text{return}} \leftarrow x^{(1)}$;
                $x^{(2)}_{\text{return}} \leftarrow x^{(2)}$;
            end
        end
    end
    return $x^{(1)}_{return}, x^{(2)}_{return}$;
```

4.2.1.3 Pair Selection with Proximity Optimization

In Algorithm 4, identifying the point $x^{(2)}$ for a given $x^{(1)}$ involves iteratively scanning all data points in the inner loop. For each combination, it computes the resulting new weight vector $\hat{w}_{1,i}$ and updates the best pair based on the Euclidean distance between the new and reference weight vectors.

While this method is comprehensive, it is computationally demanding, particularly for large input datasets X and threshold values r that result in large set X'. The use of nested loops in Algorithm 4 leads to a worst case time complexity of $\mathcal{O}(|X|^2)$. Quadratic complexity with respect to the number of input samples is prohibitive in the context of

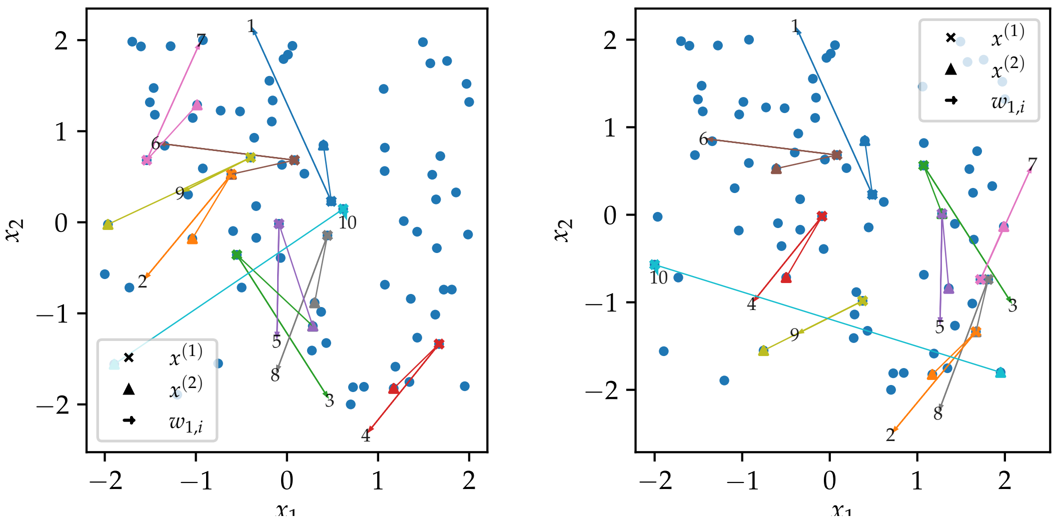

Figure 4.3: The dataset contains 80 points in the range $[-2, 2]^2$. The given network's hidden layer contain 10 nodes. Each plot shows the dataset points, the neural network's weight vectors, and the sampled points for each weight vector. The plot on the left uses the Sampled Weight Closest Distance approach with a radius of $r = 0$. The plot on the right uses a radius of $r = 0.05$.

neural networks. Consequently, we aim to further reduce the time complexity through reformulations of the problem.

The previous approach is effectively finding the point $x^{(2)} \in X \setminus \{x^{(1)}\}$, for a given point $x^{(1)}$ and weight vector $w_{1,i}$ that the norm of the difference between the sampled weight and the given weight,

$$\left\| \frac{x^{(2)} - x^{(1)}}{\|x^{(2)} - x^{(1)}\|^2} - w_{1,i} \right\|, \tag{4.1}$$

is minimal.

Let $\hat{x} \in \mathbb{R}^D$ be the point that would result in the norm of the difference equal to 0. This point $\hat{x}$ is not necessarily an element of the input dataset X. Our approach is to find the point $x^{(2)}$ that is closest in Euclidean distance to that point $\hat{x}$.

Plugging $\hat{x}$ into the equation 4.1 and setting to zero, we have

$$\frac{\hat{x} - x^{(1)}}{\|\hat{x} - x^{(1)}\|^2} = w_{1,i}. \tag{4.2}$$

Taking the norm on both sides as

$$\left\| \frac{\hat{x} - x^{(1)}}{\|\hat{x} - x^{(1)}\|^2} \right\| = \|w_{1,i}\|, \tag{4.3}$$

gives

$$\frac{1}{\|\hat{x} - x^{(1)}\|} = \|w_{1,i}\|. \tag{4.4}$$

Now substituting 4.4 into 4.2 gives

$$(\hat{x} - x^{(1)})\|w_{1,i}\|^2 = w_{1,i},$$

which means

$$\hat{x} = x^{(1)} + \frac{w_{1,i}}{\|w_{1,i}\|^2}.$$

Using this result, for any given point $x^{(1)}$ and weight vector $w_{1,i}$, we can compute the point $\hat{x} = x^{(1)} + \frac{w}{|w|^2}$ that results in perfect alignment. Subsequently, we identify the point $x^{(2)}$ from the input dataset that is nearest to this newly computed $\hat{x}$. This process can be optimized by utilizing specialized data structures such as KD-trees.

KD-trees, or k-dimensional trees, are used to organize points in a k-dimensional space. The tree is constructed by recursively choosing a dimension and dividing the data along the median of that dimension. The node having the median value defines a hyperrectangle. KD-trees provide logarithmic time complexity for the nearest neighbor search problem [Skr19].

In the Python implementation presented in Listing 1, we first construct the set X'. We then iterate over each element $x^{(1)}$ in that set. In each iteration, the point $\hat{x}$, which would result in an optimal alignment, is computed. We then query the closest point to it from our input dataset X to find $x^{(2)}$. If the closest point is equal to $x^{(1)}$, we query the next nearest point. At the end, the pair out of all possible combinations that has the best results is returned.

```python
import numpy as np
from scipy.spatial import KDTree

def find_data_pair(X, weight, bias, r=0):
    x_min = X[np.argmin(np.abs(np.dot(X, weight) - bias))]
    M = np.max(np.abs(np.dot(X, weight) - bias))
    X_prime = [x for x in X if abs(np.dot(x, weight) - bias) < r * M or x == x_min]

    min_distance = float('inf')
    x_1, x_2 = None, None
    X_tree = KDTree(X)

    for x_1_curr in X_prime:
        x_hat = x_1_curr + weight / np.linalg.norm(weight) ** 2
        dists, idxs = X_tree.query(x_hat, 2)
        chosen_idx = 0 if dists[0] != 0 else 1
        if dists[chosen_idx] < min_distance:
        min_distance = dists[chosen_idx]
        x_1, x_2 = x_1_curr, X_tree.data[idxs[chosen_idx]]
    return x_1, x_2
```

Listing 1: The implementation in Python for the pair selection with approximity optimization approach.

4.2.1.4 Minimizing the Bias Shift

For a neuron i, after selecting the data pair $(x^{(1)}, x^{(2)})$, the new bias is computed as $\hat{b}_{1,i} = \langle x^{(1)}, \hat{w}_{1,i} \rangle$. As discussed in Section 4.2.1.1, this computation leads to a shift in the neuron's activation boundary. This shift occurs because $x^{(1)}$ is now mapped to 0, and thus defining a new activation boundary. Consequently, the original neuron's activation boundary has been repositioned to cross through $x^{(1)}$, the point closest to it in the dataset.

To minimize the shift in the bias even further, we suggest projecting $x^{(1)}$ onto the hyperplane H to derive $x^{(1)}_{proj}$. We also shift $x^{(2)}$ with the same amount to receive $x^{(2)}_{proj}$, in a way that the relative position between $x^{(1)}_{proj}$ and $x^{(2)}_{proj}$ remains unchanged.

This modification does not alter $\hat{w}_{1,i}$ because the vector difference between $x^{(2)}$ and $x^{(1)}$ remains unchanged. However, it does affect $\hat{b}_{1,i}$, which now becomes

$$\hat{b}_{1,i} = \langle x^{(1)}_{proj}, \hat{w}_{1,i} \rangle.$$

This ensures that the mapped value of $x^{(1)}_{proj}$ remains zero, maintaining consistency with the activation transition.

This shift is executed after the data pair has been identified through the function FindDataPair, and is demonstrated in Algorithm 5. Initially, d, the signed distance from $x^{(1)}$ to the hyperplane H, is calculated. $\frac{w}{\|w\|}$ represents the unit vector in the direction of the normal to the hyperplane H. The adjustment vector δ is obtained by multiplying this unit vector with the distance d, and points from the point $x^{(1)}$ to its projection on the hyperplane H. The pair is then shifted accordingly to obtain $x^{(1)}_{proj}$ and $x^{(2)}_{proj}$.

Algorithm 5: Shift $x^{(1)}$ and $x^{(2)}$ such that $x^{(1)}_{\text{proj}}$ lies on the activation boundary.

```
Function ShiftToActivationBoundary($x^{(1)}, x^{(2)}, w, b$):
    $d \leftarrow \frac{\langle x^{(1)}, w \rangle - b}{\|w\|}$;
    $\delta \leftarrow d \cdot \frac{w}{\|w\|}$;
    $x^{(1)}_{proj} \leftarrow x^{(1)} - \delta$;
    $x^{(2)}_{proj} \leftarrow x^{(2)} - \delta$;
    return $x^{(1)}_{proj}, x^{(2)}_{proj}$;
```

4.2.1.5 Input Dataset Augmentation

One concern that might arise is when the input dataset is not large enough and has an insufficient number of points for the sampling algorithm. In this case, it would be challenging to find representative data pairs $(x^{(1)}, x^{(2)})$ that would allow the construction of a sampled network that aligns with the trained network.

While the input dataset is given and is a property of the problem, we can consider techniques for data augmentation to artificially expand the dataset and thereby improve the chances of finding representative data pairs. One method of data augmentation is to add Gaussian noise to the input variables [GBC16].

In our implementation, for each data point, we generate n new samples following a Gaussian distribution with a mean μ and a standard deviation σ. μ is set to 0 to avoid bias in the noise. The values of n and σ can be parameterized. n essentially determines how many times we multiply the number of samples of the input dataset. A large n could lead to better accuracy but a drop in performance. σ determines how spread out the new samples are. Low values of σ could lead to new points being close to the original ones, and thus not significantly increasing the accuracy. High values of σ could, however, lead to a loss of the underlying structure of the dataset.

4.2.2 Updating the Second Layer

Once the first layer of the network has been converted using the sampling approach detailed in previous sections, the next step is to adapt the second layer so that the overall function of the sampled network approximates that of the original network as closely as possible. In the following part, we present two strategies to achieve this: first, by retaining the weights of the second layer and adjusting only the biases; second, by updating both the weights and biases to minimize a loss function between the output of the sampled network and that of the original given network.

4.2.2.1 Only Bias Update of the Output Layer

Under the assumption that the weights and biases of the first layer in the sampled network closely align with the original ones, we retain the weights of the second layer as-is while only optimizing the biases.

We denote by $\hat{\Phi}^{(l)}(x)$ and $\Phi^{(l)}(x)$ the outputs of the sampled network and the given neural network at the point x on the l^{th} layer, respectively. Each of these outputs is a vector with a dimension of N_l. The goal is to calculate the bias $\hat{b}_{2,j}$ for each neuron j in the second layer in such a way that the mean squared error MSE between the output of the given and the sampled network for that neuron is minimized. To do this, for each neuron j in the second layer, we minimize the MSE function given by

$$\begin{aligned}\mathrm{MSE}(\hat{b}_{2,j}) &= \frac{1}{M}\sum_{i=1}^{M}\left(\hat{\Phi}_j^{(2)}(x_i) - \Phi_j^{(2)}(x_i)\right)^2 \\ &= \frac{1}{M}\sum_{i=1}^{M}\left(\langle\hat{\Phi}^{(1)}(x_i), w_{2,j}\rangle - \hat{b}_{2,j} - \Phi_j^{(2)}(x_i)\right)^2,\end{aligned}$$

where $\hat{\Phi}_j^{(2)}(x_i)$ and $\Phi_j^{(2)}(x_i)$ are the j^{th} elements of the output vectors $\hat{\Phi}^{(2)}(x_i)$ and $\Phi^{(2)}(x_i)$, respectively. To find the value of $\hat{b}_{2,j}$ that minimizes the MSE, we derive the equation with respect to $\hat{b}_{2,j}$ and set it to zero, which gives

$$\frac{1}{M}\sum_{i=1}^{M} -2\left(\langle\hat{\Phi}^{(1)}(x_i), w_{2,j}\rangle - \hat{b}_{2,j} - \Phi_j^{(2)}(x_i)\right) = 0.$$

Solving for $\hat{b}_{2,j}$ we find

$$\hat{b}_{2,j} = \frac{\sum_{i=1}^{M} \langle \hat{\Phi}^{(1)}(x_i), w_{2,j} \rangle - \sum_{i=1}^{M} \Phi_j^{(2)}(x_i)}{M}.$$

We repeat this process for each node in the second layer to construct the bias vector for the second layer, $\hat{b}_2$. As a result, the weights and biases of the second layer of the sampled network are given by

$$\hat{W}_2 = W_2, \quad \hat{b}_2 = \begin{bmatrix} \hat{b}_{2,1} \\ \hat{b}_{2,2} \\ \vdots \\ \hat{b}_{2,N_2} \end{bmatrix}.$$

4.2.2.2 Weight and Bias Update with Linear Regression

The second strategy is to adjust both the weights and biases of the second layer to minimize a loss between the output of the sampled network and the given network output. The goal is to find a weight matrix and a bias vector that satisfy

$$\begin{aligned} \hat{W}_2, \hat{b}_2 &= \arg\min \mathcal{L}(\hat{\Phi}^{(2)}(X), \Phi^{(2)}(X)) \\ &= \arg\min \mathcal{L}(\hat{\Phi}^{(1)}(X)\hat{W}_2 - \hat{b}_2, \Phi^{(2)}(X)). \end{aligned}$$

Given that $\hat{W}_2 \in \mathbb{R}^{N_{l-1} \times N_l}$ and $\hat{b}_2 \in \mathbb{R}^{N_l}$, we stack the vector $\hat{b}_2$ onto the first row of $\hat{W}_2$. We denote the resulting matrix by $\mathbf{W}$, given by

$$\mathbf{W} = \begin{bmatrix} \hat{b}_2 \\ \hat{W}_2 \end{bmatrix}.$$

Similarly, we construct a matrix $\mathbf{X}$ by adding a first column of values equal to -1 to the matrix $\hat{\Phi}^{(1)}(X)$ as

$$\mathbf{X} = \begin{bmatrix} -\mathbf{1} & \hat{\Phi}^{(1)}(X) \end{bmatrix}.$$

The output of the sampled network can now be expressed in terms of these matrices as

$$\hat{\Phi}^{(2)}(X) = \hat{\Phi}^{(1)}(X)\hat{W}_2 - \hat{b}_2 = \mathbf{XW}.$$

The problem now reduces to finding the matrix $\mathbf{W}$ that minimizes the loss function, by satisfying

$$\mathbf{W} = \arg\min \mathcal{L}(\mathbf{XW}, \Phi^{(2)}(X)).$$

One method for solving this linear optimization problem is the least squares method [Van05]. This method aims to minimize the squared discrepancies between the observed data and their expected values. In our case, we want to find the matrix **W** that minimizes

$$LSE(\mathbf{W}) = \sum_{i=1}^{M} \|\hat{\Phi}^{(2)}(x_i) - \Phi^{(2)}(x_i)\|^2 = \|\hat{\Phi}^{(2)}(X) - \Phi^{(2)}(X)\|^2 = \|\mathbf{X}\mathbf{W} - \Phi^{(2)}(X)\|^2.$$

Differentiating with respect to **W** yields

$$\frac{\partial LSE(\mathbf{W})}{\partial \mathbf{W}} = 2\mathbf{X}^T(\mathbf{X}\mathbf{W} - \Phi^{(2)}(X)).$$

Setting the derivative to zero and solving for **W** gives the result

$$\mathbf{W} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\Phi^{(2)}(X).$$

However, one notable limitaiton of LSE is when the input data samples are highly correlated. The method can even fail to find a unique solution when the matrix $\mathbf{X}^T\mathbf{X}$ is not invertible [HK70]. Arthur Hoerl and Robert Kennard proposed a method called ridge regression to account for this [HK70]. On top of minimizing he square differences between the observed data and their expected values, the ridge regression also minimizes the magnitudes of the parameter **W**. The loss function is denoted as

$$J(\mathbf{W}) = \|\mathbf{X}\mathbf{W} - \Phi^{(2)}(X)\|^2 + \lambda\|\mathbf{W}\|^2,$$

where λ is a regularization parameter that controls the amount of shrinkage applied to the parameters. To find the minimum of this loss function, we set its gradient to zero, giving

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = 2\mathbf{X}^T(\mathbf{X}\mathbf{W} - \Phi^{(2)}(X)) + 2\lambda\mathbf{W} = 0.$$

Solving for **W**, we find

$$\mathbf{W} = (\mathbf{X}^T\mathbf{X} + \lambda I)^{-1}\mathbf{X}^T\Phi^{(2)}(X).$$

Here, I is the identity matrix. This problem guarantees a unique solution for $\lambda > 0$, since the matrix $(\mathbf{X}^T\mathbf{X} + \lambda I)$ always possesses full rank.

Upon deriving the optimal **W**, we extract the second layer's weights and biases using the definition

$$\mathbf{W} = \begin{bmatrix} \hat{b}_2 \\ \hat{W}_2 \end{bmatrix}.$$

Discussion: While each of these two strategies presented to convert the second layer's parameters have their merits, there are some considerations that need to be addressed.

On the one hand, despite the simplicity of the bias-only update method, it may not always produce an alignment with the original network that is as good as updating both the weights and biases using regression. This especially happens when there are underlying discrepancies that occurred when converting the first layer that the second layer cannot correct with bias adjustments alone.

On the other hand, adjusting both weights and biases using linear regression, can potentially achieve a better alignment in the output with the trained network. However, this increased adaptability also introduces the risk of overfitting or shadowing underlying discrepancies from the first layer. The ability to correct outputs in the second layer using regression might mask some significant discrepancies occurred while converting the first layer. In other words, while the final output of the sampled network may appear accurate and matched that of the trained network, this does not necessarily guarantee that the internal representations, especially in the first layer, are true to the original network. This is especially concerning as regression from a large number of nodes in the hidden layer to the output layer can produce satisfactory outputs, but simultaneously mask potential high variances in the first layer's weights and biases of the sampled network.

5 Experiments and Results

In the previous chapter, we presented some approaches for converting a traditionally trained neural network containing one hidden layer into a sampled network. In the following, we will present and analyze the results obtained from each of these approaches. As a recapitulation, we will revisit the following approaches in order:

- **Converting the hidden layer**:
    1. Lowest angle approach
    2. Weight norm preservation approach with threshold ratio *r*
    3. Pair selection with proximity optimization
    4. Minimizing the bias shift
    5. Input dataset augmentation with Gaussian sampling
- **Converting the second layer**:
    1. Only updating the biases
    2. Updating the weights and biases with linear regression

5.1 Experimental Setup and Evaluation Metrics

First, we set up an input framework that will be used throughout the tests. Then, we present the metrics used to evaluate the different conversion algorithms.

5.1.1 Input Framework

The input space used throughout the experiments is $\mathcal{X} = [-2, 2]^2$. The choice of a 2-dimensional space allows for better visualization and interpretability of angles and norms. We opted for the interval $[-2, 2]$ to provide more variance in the data compared to the typical $[-1, 1]$, without introducing an overly large range.

The target function that the given neural network is trained on is the Laplacian of Gaussian (LoG) function, which is commonly used in the field of image processing

[MH80]. The function is defined as

$$LoG(x_1, x_2) = -\frac{1}{\pi\sigma^4}\left(1 - \frac{x_1^2 + x_2^2}{2\sigma^2}\right)e^{-\frac{x_1^2+x_2^2}{2\sigma^2}}. \tag{5.1}$$

The LoG function maps from $\mathcal{X}$ to $\mathbb{R}$. The value of σ is chosen to be 0.5. The plot of the function in the 3D space is shown in Figure 5.1.

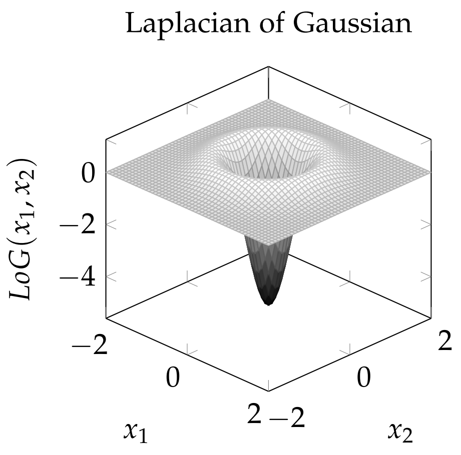

Figure 5.1: Plot of the Laplacian of Gaussian function with $\sigma = 0.5$ over the input space $[-2, 2]^2$

The neural network we aim to convert into a sampled network has input and output dimensions of $N_0 = 2$ and $N_2 = 1$, respectively. It contains a single hidden layer with $N_1 = 100$ nodes. This network is trained with the Adam optimizer on the target function using 5,000 data points. The training minimizes the mean squared error (MSE) loss function, spans 150 epochs, has a validation split of 0.2, and is initialized based on He et al. [He+15]. After training, the network achieves a loss of 0.0287 on the training set and 0.0308 on the validation set.

5.1.2 Evaluation Metrics

The metrics we use to assess the conversion measure how closely the converted sampled network matches the trained neural network. Essentially, they measure the similarity between the sampled weights and biases and the trained weights and biases of the initial neural network, as well as how well the output of the sampled network aligns with the trained network's output. More specifically, the metrics include:

Measures of similarity between the networks' parameters:

- Weight vectors angle mean (the lower the better): It is equal to the mean of the angle differences between each weight vector of the trained network's hidden layer $w_{1,i}$ and the corresponding weight vector of the sampled network $\hat{w}_{1,i}$.
- Weight norms Mean Absolute Error (MAE, the lower the better): This metric calculates the average of the absolute differences between the norms of the weights in the initial network's hidden layer $w_{1,i}$ and the norms of the corresponding weights in the sampled network $\hat{w}_{1,i}$.
- Weight norms correlation (the closer to 1 the better): This measures the correlation between the sampled network's weight and the trained network's weight norms.
- Biases Mean Absolute Error (MAE, the lower the better): Similarly, this metric computes the average of the absolute differences between the biases in the initial network's hidden layer $b_{1,i}$ and the biases of the sampled network $\hat{b}_{1,i}$.

Measures of similarity between the networks' outputs:

- Mean Squared Error of the output (MSE, the lower the better): This metric calculates the mean squared errors between the outputs of the sampled network and the trained network. To compute this, we use a test dataset X_{test} with $M_{test} = 1250$ points that are randomly and uniformly distributed over the input space $\mathcal{X}$. The formula is given by
$$\frac{1}{M_{test}} \sum_{i=1}^{M_{test}} \|\hat{\Phi}(x_i) - \Phi(x_i)\|^2.$$
- Coefficient of Determination (R^2 score, the closer to 1 the better): The R^2 score represents the proportion of the variance in the output of the converted sampled network that is predictable from the output of the trained network. It is defined as
$$R^2 = 1 - \frac{MSE}{Var(\Phi(X_{test}))}.$$

Taking into account the remarks raised in the discussion in Paragraph 4.2.2.2, to obtain a more representable and transparent outcome from the first layer's conversion methods, we will only update the bias of the second layer while retaining the weights from the initial trained network.

5.2 Results and Analysis

After establishing the test framework and evaluation metrics, we test and analyze each approach to the conversion algorithm. Each of the following subsections begins with a brief description of a conversion approach, followed by a discussion of its results.

5.2.1 Lowest Angle Approach

This approach aims to create sampled weights that align closely in direction with the weight vectors of the trained network. The trained neural network to be converted is the same as described in Section 5.1. The input dataset X used for the sampling approaches contains $M = 1000$ points, randomly and uniformly distributed over the input space $\mathcal{X}$.

The evaluation of the output is given by the results in Table 5.1. Analyzing the results, we can make several observations.

The mean of the angles between the sampled and the initial weight vectors is 0.83 degrees, suggesting that the directions of the vectors are generally well-aligned. However, there is a considerable difference in the weight norms, evidenced by a mean absolute error (MAE) between the norms of 1.019. This idea is further supported by the low correlation value of 0.1988, indicating a weak relationship between the norms of the trained weights and those of theconverted sampled weights.

This significant difference in norms has a noticeable impact on the output, leading to a high discrepancy between the outputs of the trained and the sampled networks. Indeed, the high MSE value of 7.53, with an extremely low R^2 score of -8.65, manifest the mismatch between the outputs of the two networks.

Table 5.1: The test metrics for the lowest angle approach show a considerably low angle mean between initial and sampled weights. The outputs, however, do not align well.

Metric	**Value**
Angle mean (degrees)	0.83
Weight norms MAE	1.019
Weight norms Correlation	0.1988
Bias difference MAE	0.37
MSE between the outputs	7.5308
R^2 score	-8.65

5.2.2 Weight Norm Preservation Approach with Threshold ratio r

The objective of this method is to address the issues that arise with the last approach by identifying sampled weights that match the weights of the initial network, both in angle and magnitude. Moreover, for a broader range of potential sampling pairs, a threshold ratio r is introduced, allowing more flexibility in selecting the first point $x^{(1)}$

of the sampling pair.

The value of r used in the tests is $r = 0.04$. The choice of this value is explained at the end of this section. The results presented in Table 5.2 allow for the extraction of several key observations.

Initially, we observe a significant reduction in the weight norms MAE, compared to the last approach, which now stands at a value of 0.008. Moreover, the correlation is closer to 1, recorded at a value of 0.9997. This indicates that the current approach performs better than the first one in creating sampled weights that better match the trained ones.

The mean angle between the sampled weight and trained weight vectors is reduced to 0.29 degrees. Although the previous method only focuses on minimizing the angle, the introduction of the ratio r helped in further reduction of the angle mean. This is because the threshold ratio r extends the range for selecting the first point $x^{(1)}$ of the sampling pair. This extended range allows a higher number of possible combinations, resulting in a lower angle mean.

Concerning the bias, one might think that introducing the threshold ratio r and allowing more points that are further away from the activation boundary to be a potential $x^{(1)}$ point might cause the sampled bias to decay. The definition of the sampled bias for a certain neuron i, $\hat{b}_{1,i} = \langle x^{(1)}, \hat{w}_{1,i} \rangle$, entails that for the same weight vector, the closer $x^{(1)}$ is to the activation boundary, the closer $\hat{b}_{1,i}$ is to $b_{1,i}$. However, in this approach, the closer alignment of the sampled weights $\hat{w}_{1,i}$ with $w_{1,i}$ results in decreased bias decay. The mean absolute error between the sampled biases and the initial biases is actually lower than the first approach, with a value of 0.108.

This better alignment between the parameters of the transformed sampled network and those of the initial network leads to improved accuracy in the output. This is reflected in a lower MSE between the outputs, noted at 0.0542, and an R^2 score that is closer to 1, with a value of 0.9.

Figures 5.2 and 5.3 visualize the differences in weights and biases of the sampled and trained networks' hidden layer using the first approach and this current approach respectively. In the upper plot of Figure 5.2, we observe that the weight norms are uncorrelated when applying the lowest angle approach. This issue is mitigated in the norm preservation approach, where a closer match with a lower MAE of the weight norms is clear. Furthermore, the sampled biases more accurately reflect the trained ones, as shown in the lower plots of both figures.

The choice of the value for r plays a crucial role in conversion process. As discussed in Chapter 4, a lower value of r might be too restrictive, reverting to the initial problem. In contrast, a larger value of r might introduce points farther from the activation

Table 5.2: The test metrics for the weight norm preservation approach with threshold ratio r show a better alignment in the model's parameters, with lower discrepancies in the output of the two networks.

Metric	Value
Angle mean	0.29
Weight norms MAE	0.008
Weight norms Correlation	0.9997
Bias difference MAE	0.108
MSE between the outputs	0.0542
R^2 score	0.9

boundary, leading to a larger bias shift. To find the best value for r, we conducted a hyperparameter search. Figure 5.4 shows that the minimum mean square error between the outputs of the initial network and the converted sampled network, as well as the minimum mean absolute error of the biases, occurs at a value above zero, specifically at $r = 0.04$. Larger values of r result in an increase in both the output MSE and the bias MAE.

We will use this sampling approach as a reference for comparison for the other methods that will come in the following. Moreover, we apply the bias shift minimization and the data augmentation techniques on this sampling approach.

5.2.3 Pair Selection with Proximity Optimization

This approach is very similar to the previous one, namely the weight norm preservation approach. For a neuron i, after selecting the first point $x^{(1)}$ of the pair, we first calculate the point $\hat{x}$ that would, in theory, result in a perfect alignment between the resulting sampled weight $\hat{w}_{1,i}$ and the initial weight $w_{1,i}$. This point is given by

$$\hat{x} = x^{(1)} + \frac{w_{1,i}}{\|w_{1,i}\|^2}.$$

Next, we find the point nearest to it from the input dataset using a special data structure called a KD-tree. The main goal here is to reduce the runtime complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log(n))$ by leveraging the properties of KD-trees.

To compare this conversion method with the previous one, we tested both methods on the same input set X and the neural network described in Section 5.1.

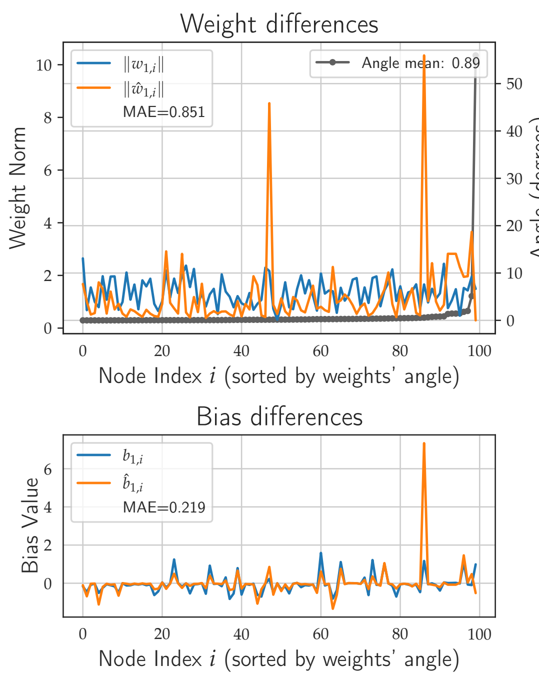

Figure 5.2: The two plots illustrate the differences in the hidden layer weights and biases between the sampled and initial network, as determined by the lowest angle approach.

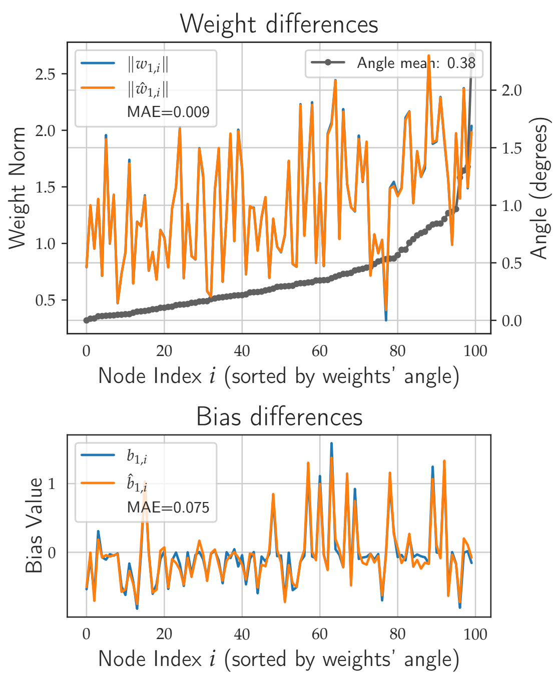

Figure 5.3: The two plots illustrate the differences in the hidden layer weights and biases between the sampled and initial network, as determined using the weight norm preservation approach with the threshold ratio.

The resulting sampled network of this method is very similar to the sampled network produced by the previous method, the weight norm preservation approach. In fact, in our tests, on average, 98 out of the 100 sampled pairs $(x^{(1)}, x^{(2)})$ were chosen identically in both methods, resulting in identical sampled weights and biases between the two converted sampled networks. For the different pairs, the differences between the resulting sampled weights and biases between the two algorithms were very minimal.

However, there is a significant difference in runtime. The runtime was measured 5 times on datasets of different sizes, and the average is shown in Table 5.3. We observe that there is a significant decrease in runtime, especially for large datasets.

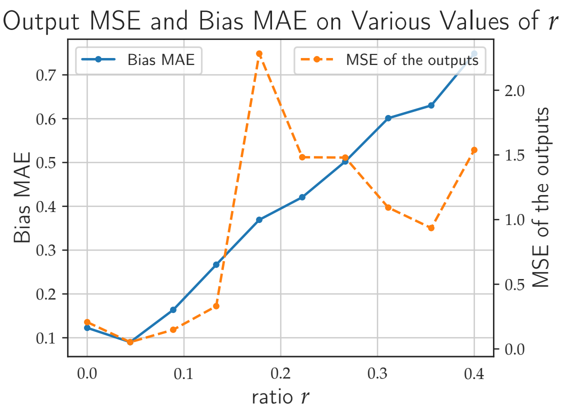

Figure 5.4: This plot illustrates the relationship between the value of r and both the MAE of the biases and the MSE of the outputs. $r = 0.04$ marks the minimum point on both plots, beyond which both plots show an increasing trend.

Table 5.3: The table compares runtime in seconds for both the Weight Norm Preservation (2nd method) and the Proximity Optimization (3rd method) approaches across different dataset sizes of X.

Method	$\lvert X \rvert = 1000$ runtime (s)	$\lvert X \rvert = 5000$ runtime (s)	$\lvert X \rvert = 10000$ runtime (s)
Weight Norm Preservation (2nd method)	0.572	8.612	32.047
Proximity Optimization (3rd method)	0.360	1.734	3.158

5.2.3.1 Minimizing the Bias Shift

This method aims to reduce the bias shift by adjusting the first point, $x^{(1)}$, to lie on the activation boundary hyperplane H. To achieve this, after choosing the pair $(x^{(1)}, x^{(2)})$, we project $x^{(1)}$ onto the hyperplane H, resulting in $x^{(1)}_{proj}$. Similarly, $x^{(2)}$ is shifted by an equivalent amount to ensure the relative position between $x^{(1)}$ and $x^{(2)}$ is maintained, and thus $x^{(2)}_{proj} - x^{(1)}_{proj} = x^{(2)} - x^{(1)}$. This ensures that the sampled vector $\hat{w}_{1,i}$ remains constant. Only the bias is modified, becoming $\hat{b}_{1,i} = \langle x^{(1)}_{proj}, \hat{w}_{1,i} \rangle$.

To test this method, we apply it after choosing the pair using the weight norm preservation approach (second approach). The left plot in Figure 5.5 displays the sampled biases compared to the initial biases for each neuron in the hidden layer, without using the bias shift minimization method. We note some discrepancies between the sampled and initial biases, with a mean absolute error (MAE) of 0.078. However, when using the bias shift minimization technique, there is a reduction in bias discrepancies, resulting in a decreased mean absolute error (MAE) of 0.009. As seen in the right plot of the Figure 5.5, the sampled bias $\hat{b}_{1,i}$ for each neuron i aligns almost perfectly with the initial bias $b_{1,i}$. We note that the sampled weights remain unchanged with or without the use of the bias shift minimization technique.

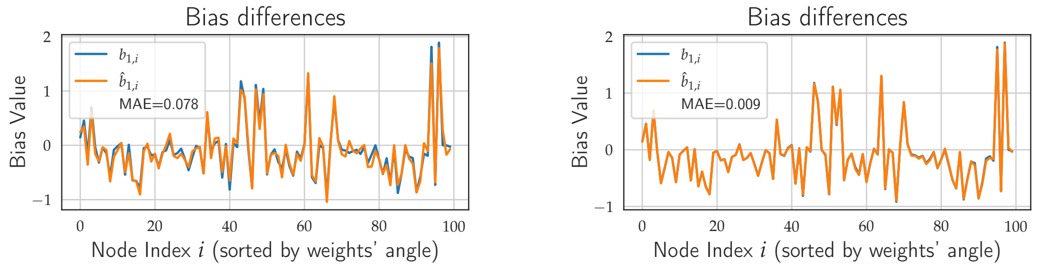

Figure 5.5: Comparison of biases in the initial and converted sampled networks. The left plot shows the biases without the bias shift minimization technique, while the right shows the biases when the bias shift minimization is applied.

The improved bias alignment results in a more accurate alignment between the output of the sampled network and the output of the initial network. Table 5.4 displays the test metrics for the weight norm preservation approach using the bias shift minimization technique. Since the sampled weights remain unchanged with or without using the bias shift minimization technique, the test metrics for the weights also remain the same. Specifically, the angle mean, the weight norms mean absolute error, and the weight norms correlation are identical to those in Table 5.2. As mentioned in the previous paragraph, the biases mean absolute error has decreased to 0.009. The MSE between the output of the sampled network and the initial network has been reduced to 0.002, and the R^2 score is nearer to 1 with a value of 0.9918.

5.2.3.2 Input Dataset Augmentation

This method addresses the issue posed by a low number of input samples by generating new data points through the addition of Gaussian Noise. In the experiments, we employ the weight norm preservation approach (2nd method) with $r = 0.04$. Initially, this conversion approach is applied using a dataset comprising $M = 100$ samples. Subsequently, Gaussian noise with parameters $n = 10$ and $\sigma = 0.5$ is added, leading to an augmented dataset of $M = 1000$ samples. We proceed to convert the neural network

Table 5.4: The table shows the test metrics evaluating the effectiveness of the bias shift minimization technique when applied after sampling the pair using the weight norm preservation approach.

Metric	**Value**
Angle mean	0.29
Weight norms MAE	0.008
Weight norms Correlation	0.9997
Bias difference MAE	0.009
MSE between the outputs	0.002
R^2 score	0.9918

using both the initial and the augmented datasets.

Figure 5.6 presents the conversion results for the neural network when sampling from the original dataset, and Figure 5.7 shows the results when using the augmented dataset. Notably, when using the dataset augmentation, a better alignment in biases and weight vectors is observed. Furthermore, the MSE of the output decreases, dropping from 2.1831 with the original dataset to 0.0626 when using the augmented dataset.

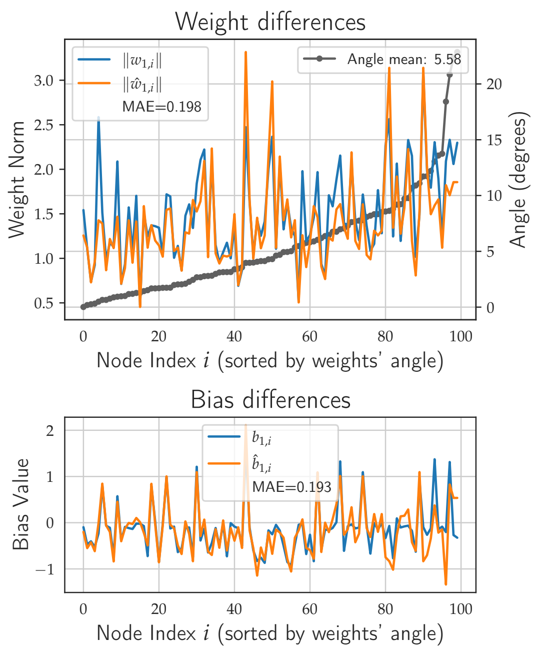

Figure 5.6: The two plots show the differences in the weights and biases of the initial and converted network when the dataset with 100 points is used for the sampling.

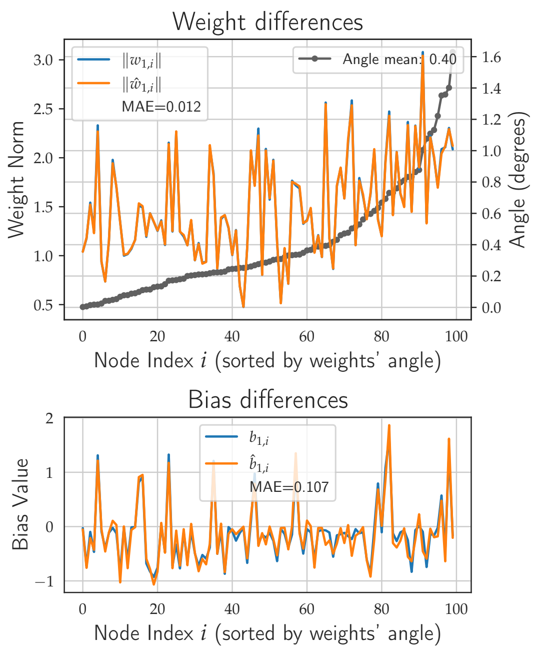

Figure 5.7: The two plots show the differences in the weights and biases of the initial and converted network when the dataset is augment using the Gaussian Noise with parameters $n = 10$ and $\sigma = 0.5$.

6 Conclusion and Future Work

6.1 Conclusion

In this thesis, we proposed and analyzed techniques to convert traditionally trained neural networks into sampled networks. The key motivation was to leverage the inherent interpretability of sampled networks to provide more transparency into already trained neural networks. Specifically, sampled networks directly link the input dataset to the network's parameters by constructing the weights and biases using pairs of data points. The weight is derived from the difference between these two points divided by the squared norm of their distance, while the bias is computed as the inner product of the derived weight and one of the two points.

Our implementation focused on converting neural networks containing one single hidden layer and an output layer and employing the ReLU activation function. The main requirement of the conversion process is that the resultant sampled network maintains the geometrical properties of the weights and biases of the trained networks in order to have similar outputs.

For the hidden layer, in order to convert the weights, we proposed three methods for selecting data pairs. The first one, the lowest angle approach, creates sampled weights that closely match the direction of the trained weights by minimizing the angle between the trained and sampled weight vectors. While the direction of the weight vector is important, as it tells which direction of the input space it emphasizes, the magnitude of the weight vector is also crucial. It indicates how strong this emphasis is. For that, we introduced the weight norm preservation approach, which, as the name suggests, samples a data pair that minimizes the norm of the difference between the trained and sampled weight vector. This method, while it may provide the most accuracy, has a worst case runtime complexity in terms of the number of input data of $\mathcal{O}(n^2)$. For that, we introduced the pair selection with proximity optimization approach that makes use of a special data structure to optimize the search. With this, the time complexity is lowered to $\mathcal{O}(n \log(n))$.

As for the bias, all the previous methods result in a minor bias shift from the trained one. For the ReLU activation function, this also shift the activation boundary of the neuron, thus altering the network's dynamics. To address that, we added the possibility

of projecting the first of the sampling pair onto the activation boundary of the neuron.
Another issue arises when the sampling dataset is not large enough and not representative of the overall directionality and geometry of the weights and biases of the trained network. For that, we added the possibility of generating new data points using Gaussian sampling.

For the output layer, we can either retain the original weights and only update the biases by minimizing the mean error between the output of the sampled and trained network, or we can update both the weights and biases in the sampled network by using the Ridge regression. While the second method might yield more accurate results, retaining the weights and only updating the bias provides more transparency with the trained network.

Overall, this thesis provides an interpretability means for traditionally trained neural networks by making use of the properties of sampled networks.

6.2 Future Work

While this thesis focuses on converting a simple neural network containing one hidden layer, the conversion techniques could be extended to larger and deeper neural architectures. The pair selection strategies may need to be adapted for networks with more complex representations.

Another potential area of future work is developing theoretical guarantees on the equivalence between a trained network and its sampled converted version.

The conversion process could also be extended to support different network architectures, activation functions, and training algorithms.

Finally, it would be interesting to apply the conversion on neural networks trained on real-world datasets. The tests and experiments in this thesis were run on synthetic and scaled datasets. This helped with better interpreting the angles and norms in data pairs. That might not be optimal with higher-dimensional raw data.

List of Figures

List of Tables

Bibliography

[Agg18] C. C. Aggarwal. *Neural Networks and Deep Learning. A Textbook*. Cham: Springer, 2018, p. 497. ISBN: 978-3-319-94462-3. DOI: 10.1007/978-3-319-94463-0.

[Ako13] D. Akomolafe. "Scholars Research Library Comparative study of biological and artificial neural networks." In: *European Journal of Applied Engineering and Scientific Research* 2 (Jan. 2013), pp. 36–46.

[Ama+13] F. Amato, A. López-Rodríguez, E. Peña-Méndez, P. Vaňhara, A. Hampl, and J. Havel. "Artificial neural networks in medical diagnosis." In: *J Appl Biomed* 11 (Dec. 2013), pp. 47–58. DOI: 10.2478/v10136-012-0031-x.

[Bol+23] E. L. Bolager, I. Burak, C. Datar, Q. Sun, and F. Dietrich. *Sampling weights of deep neural networks*. 2023. arXiv: 2306.16830 [cs.LG].

[Cas16] D. Castelvecchi. "Can we open the black box of AI?" In: *Nature News* 538.7623 (2016), pp. 20–23.

[DHS01] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern classification*. 2nd. John Wiley & Sons, 2001.

[GBC16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[He+15] K. He, X. Zhang, S. Ren, and J. Sun. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. 2015. arXiv: 1502.01852 [cs.CV].

[HK70] A. E. Hoerl and R. W. Kennard. "Ridge Regression: Biased Estimation for Nonorthogonal Problems." In: *Technometrics* 12.1 (1970).

[HSS12] G. Hinton, N. Srivastava, and K. Swersky. *Neural networks for machine learning lecture 6a overview of mini-batch gradient descent*. Lecture notes. 2012.

[KB14] D. P. Kingma and J. Ba. "Adam: A method for stochastic optimization." In: *arXiv preprint arXiv:1412.6980* (2014).

[LLB09] L. J. Lancashire, C. Lemetre, and G. R. Ball. "An introduction to artificial neural networks in bioinformatics–application to complex microarray and mass spectrometry datasets in cancer studies." In: *Briefings in Bioinformatics* 10.3 (May 2009). Epub 2009 Mar 23, pp. 315–329. ISSN: 19307287. DOI: `10.1093/bib/bbp012`.

[MH80] D. Marr and E. Hildreth. "Theory of Edge Detection." In: *Proceedings of the Royal Society of London. Series B, Biological Sciences* 207.1167 (1980), pp. 187–217.

[MM97] T. M. Mitchell and T. M. Mitchell. *Machine learning*. Vol. 1. 9. McGraw-hill New York, 1997.

[Pol64] B. T. Polyak. "Some methods of speeding up the convergence of iteration methods." In: *USSR Computational Mathematics and Mathematical Physics* 4.5 (1964), pp. 1–17.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning representations by back-propagating errors." In: *Nature* 323.6088 (1986), pp. 533–536.

[Skr19] M. Skrodzki. *The k-d tree data structure and a proof for neighborhood computation in expected logarithmic time*. 2019. arXiv: `1903.04936 [cs.DS]`.

[SSA20] S. Sharma, S. Sharma, and A. Athaiya. "ACTIVATION FUNCTIONS IN NEURAL NETWORKS." In: *International Journal of Engineering Applied Sciences and Technology* 04 (May 2020), pp. 310–316. DOI: `10.33564/IJEAST.2020.v04i12.054`.

[Van05] S. A. Van de Geer. "Least Squares Estimation." In: *Encyclopedia of Statistics in Behavioral Science*. Ed. by B. S. Everitt and D. C. Howell. Vol. 2. Chichester: John Wiley & Sons, Ltd, 2005, pp. 1041–1045. ISBN: 978-0-470-86080-9.