EPFL

Function approximation through Neural Networks

Raphael Attias

June 9, 2020

Abstract

With the continuous development of machine learning, feedforward neural networks are widely used to approximate continuous functions. In order to study its accuracy, we present multiple experiments on approximating arbitrary continuous function but also functions that are essentially neural networks. A particular emphasis is placed on the computational and experimental aspects of the problem, i.e. we discuss the possibility of realizing a feedforward neural network that achieves a prescribed degree of accuracy of the approximation, and the determination of the number of hidden layer neurons required to achieve this accuracy. We empirically study the circumstances in which a function can be approximated, and the probability to reach arbitrary precision when the function to approximate can be represented as a neural network.

Keywords: approximation, neural networks, ReLU activation, universal approximation

1 Introduction

Over the past twenty years, neural networks with rectified linear hidden units (ReLU) as activation functions have demonstrated impressive performance in many important applications, such as information system, image classification, text understanding, etc. These networks are usually trained with Stochastic Gradient Descent (SGD), where the gradient of loss function with respect to the weights can be efficiently computed via back propagation method

Many authors recently addressed the question of approximation by neural networks. They have been shown to be capable of approximating generic class of functions, including continuous and integrable ones. By increasing the number of neurons, we can obtain arbitrary precision in approximating continuous functions. This has been shown in 1989 by Cybenko [1] and this will be further discussed in section 3. Although networks with a single hidden layer are universal approximators, the width (i.e the number of neurons) of such networks has to be exponentially large. To counter this problem, from section 4 we will study the approximation of functions that are expressed as neural networks with a finite number of neurons. We will focus on the probability of reaching exactly the target function. One of the main conclusions of our experiments is that over-parameterization is both a necessary and sufficient condition to reach a sufficient amount of precision.

In this introduction, we will first describe the mathematical and algorithmic setup for this report, then we will present some recent theory on function approximation with neural networks.

1.1 Mathematical setup

In this report, we will be focusing on one dimensional functions, i.e from $\mathbb{R}$ to $\mathbb{R}$. Let $u(x)$ be our target function, and $u_\theta(x)$ a neural networks with parameters θ. We shall explain how neural networks are parametrized and the framework that we will use.

Neural networks are functions that can be essentialy described by the number of neurons in their layers and their associated parameters. In this study we focus on one dimensional neural networks, i.e $u_\theta : \mathbb{R} \to \mathbb{R}$. This translates to a network with one neuron in its input layer and one neuron in the output layer. We will

study neural networks with one layer of neurons between the input layer and hidden layer. This is called a *shallow network* and these networks are the subject of many theoretical results discussed later on. A neural network can be described as a graph, where the parameters resides in the edges and the operations are done in the vertices. This can be visualized on figure (1).

If the neural network has N neurons, we will use the parameters $\theta = (\boldsymbol{w}, \boldsymbol{b}, \boldsymbol{\alpha}) \in \mathbb{R}^N \times \mathbb{R}^N \times \mathbb{R}^N$, where $\boldsymbol{w}, \boldsymbol{b}, \boldsymbol{\alpha}$ are respectively called the *inner weights*, *inner bias* and *outer weights*. For simplicity, the parameters belonging to the same layer are on the same edges. Therefore the inner bias and inner weights are are on the first layer of edges, and the outer weights remain on the last layer of edges. This can be visualized on figure 2. The output of such one dimensional neural network is a linear combination of the first layer, passed through an activation function $\sigma(\cdot)$, and weighted by the outer weights:

$$u_\theta(x) = \sum_{j=1}^{N} \alpha_j \sigma(w_j x + b_j).$$

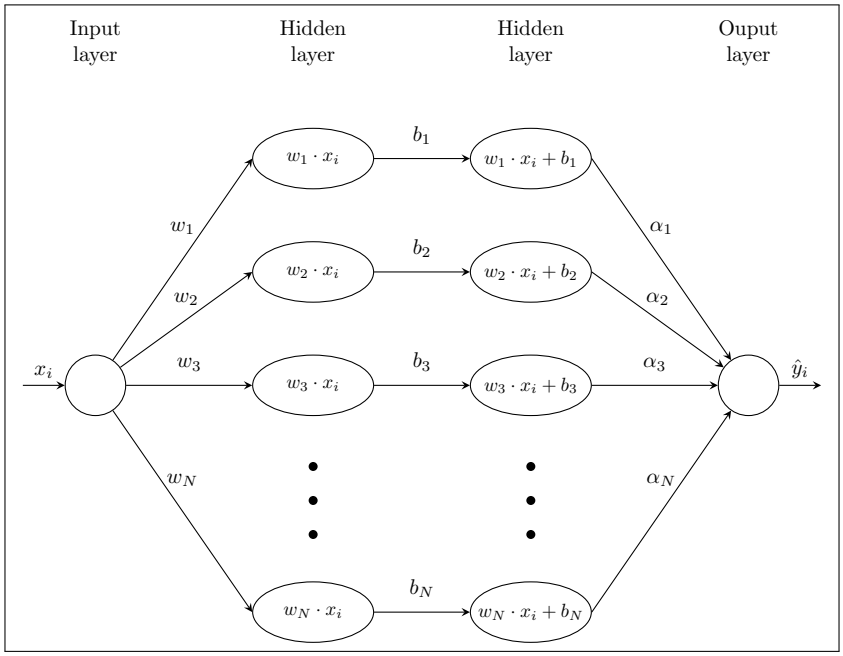

Figure 1: Feedforward one dimensional neural network with N neurons in the hidden layer, with no bias in the second layer.

Training a network consists in iteratively learning the best parameters θ for a given task. To train a network, we first need a *training set*. This is a set $\{(x_i, y_i)\}_{i=1}^n$ of inputs x_i and desired output y_i. In this report, we used inputs uniformly distributed in $[-1, 1]$, and the output a target function $y_i = u(x_i)$. When training a network, we hope that if the network performs well on the training set, it will be a good approximation of u for any input in $[-1, 1]$. At each iteration (or *epoch*), we evaluate the error made by our network in the prediction $\hat{y}_i = u_\theta(x_i)$ and y_i on the whole training set. We will use the sum of square error (SSE):

$$L(\theta) = \sum_{i=1}^{n} (u(x_i) - u_\theta(x_i))^2$$

Since the SSE is an indicator of the error made by the trained network, ultimately the goal is to minimize the SSE. A common algorithm to minimize this error is the Stochastic Gradient Descent algorithm (SGD). The parameters θ are updated with learning rate η_k according to:

$$\theta^{k+1} = \theta^k - \eta_k \left. \frac{\partial L(\theta)}{\partial \theta} \right|_{\theta=\theta^k}$$

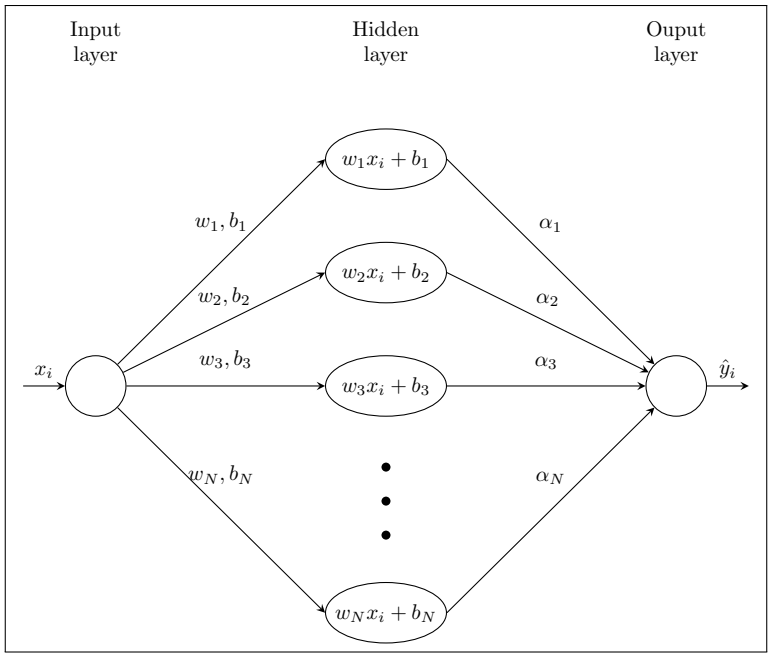

Figure 2: Feedforward one dimensional neural network with weights and bias visualized on the same edges

1.2 Algorithmic setup

Pytorch is an increasingly popular machine learning framework for python. This library is used by many researchers for its intuitiveness and the ease of implementation. Throughout this report we constantly used a learning rate of 0.08, with optimizer SGD and the criterion being the SSE. A model is first trained using a training set, then evaluated on a different set called the *test test* (or sometimes called in academia *validation set*). The purpose of the test set is to provide an unbiased evaluation of a final model fit on the training dataset. Since the model is trained on the training set, we expect a greater loss when we evaluate the model on the validation set. We will use a training set of size 50, and a validation/test set of size 150. The network described in figure 2 can be initialized in Pytorch with the following code:

```
# Model Initialization
 class network(torch.nn.Module):
    def __init__(self):
        super(network,self).__init__()
        self.fc1 = torch.nn.Linear(1,N,bias=True)
        self.fc2 = torch.nn.Linear(N,1,bias=False)
        self.relu = torch.nn.ReLU()

    def forward(self,x):
        x = self.relu(self.fc1(x))
        return self.fc2(x)
model = network()

# Model Training
criterion = torch.nn.MSELoss(reduction="sum")
optimizer = torch.optim.SGD(model.parameters(),lr=0.08)
for epoch in range(iter):
    for i, (inputs,labels) in enumerate(train_loader):
        y_pred = model(inputs)
        loss = criterion(y_pred,labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

2 Polynomial regression

As we are interested in approximating functions, polynomial regression is a natural candidate to consider. Let $u(x)$ be our target function and $u_\theta(x)$ the network with the coefficients θ obtained by training. In this experiment, $u_\theta(x)$ will be a polynomial of degree d from $\mathbb{R}$ to $\mathbb{R}$. During the training we aim to find the parameters $\theta \in \mathbb{R}^{d+1}$ that minimize the sum of squares error (SSE) on the training set $\{x_i\}_{i=1}^N$, $\{y_i\}_{i=1}^N$ where $y_i = u(x_i)$. This error is given by $L(\theta) = \sum_{i=1}^N (u(x_i) - u_\theta(x_i))^2$.

This is the standard Least Squares problem for polynomial regression. The coefficient β that minimise the SSE is well known and is given by:

$$\beta = (X^T X)^{-1} X^T \mathbf{y}, \quad \text{with } X = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^d \\ 1 & x_2 & x_2^2 & \dots & x_2^d \\ 1 & x_3 & x_3^2 & \dots & x_3^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & x_N^2 & \dots & x_N^d \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_N \end{bmatrix}$$

This problem can be solved analytically with the above equation but it is interesting to verify if it is possible to obtain the same solution by training a simple network. This network has no hidden layer and can be visualized in Figure 3. The algorithm aims to find the coefficients associated with each edge by minimizing iteratively the SSE. The estimate $\hat{y}_i$ for y_i computed by our trained network is given by: $\hat{y}_i = \sum_{k=0}^d \theta_k x_i^k$

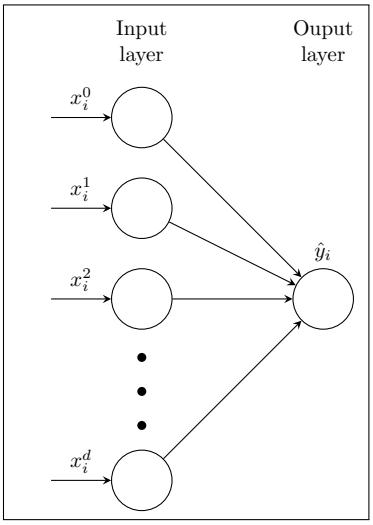

Figure 3: A polynomial function of degree d visualized as a neural network

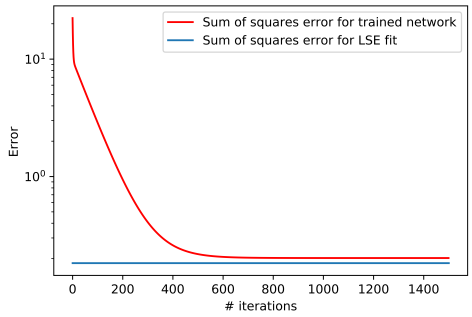

Figure 4: Loss of the trained model on figure 3 at each iteration of the training. The loss decreases by the SGD algorithm but doesn't reach the loss of the best coefficients β.

Experiment 2.1. In this experiment we chose $u(x) = cos(\frac{\pi}{2} + 5x)$, $d = 5$, $N = 30$. The initial parameters are randomly initialized with normal distribution. Both the training set and validation set follow the same uniform distribution and we observe as expected a lower SSE for the training set.

As we can see on figures 4 and 8, the algorithm reached a local minimum $\hat{\theta}$ in the space of the parameters, not the global minimum. The global minimum is unique and is known to be the parameter β. We know that a global minimum exists, since it is the exact representation given by the analytical solution β. This indicates that the algorithm can efficiently find a local minimum but may struggle to find the global minimum in the space of the parameters θ.

The next goal will be to study the behavior of a more complex network and understand how the global minimum can be reached.

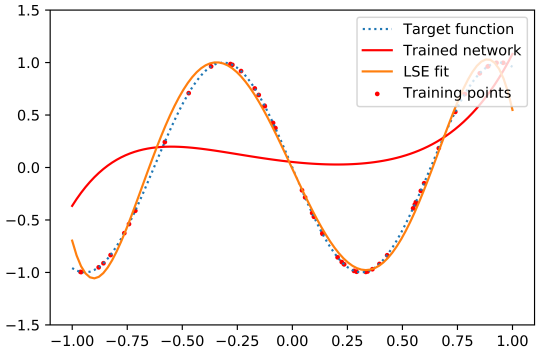

Figure 5: 1 iteration

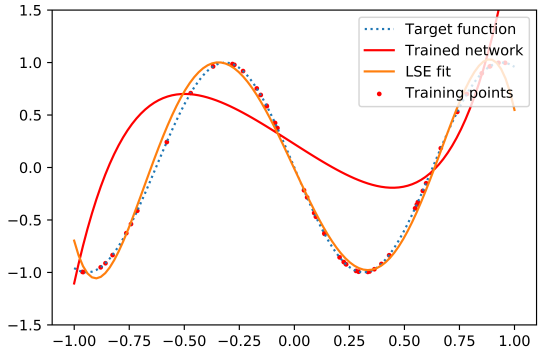

Figure 6: 5 iterations

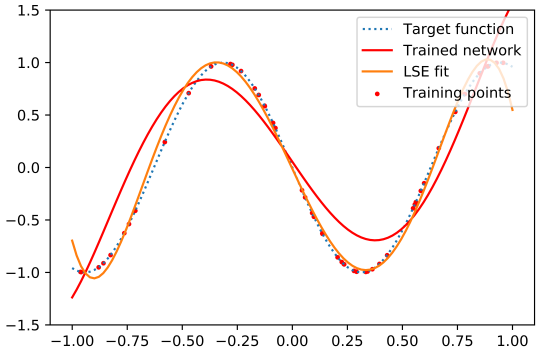

Figure 7: 100 iterations

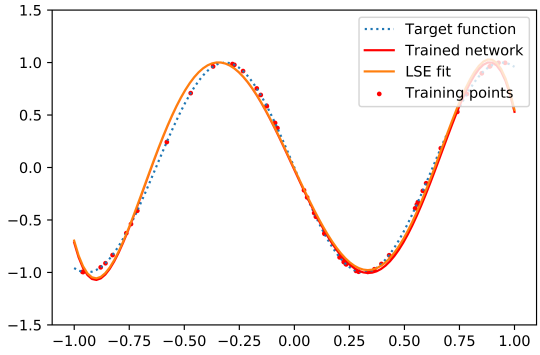

Figure 8: 1500 iterations

Figure 9: Visualization of the fitting of the polynomial of degree d trained as a network with d neurons in the first layer.

3 Universal Approximation Theorem

The Universal Approximation theorem states that a single hidden layer network with sufficient neurons can approximate any continuous function on compact subsets of $\mathbb{R}^n$. The first version of the theorem was proved by George Cybenko in 1989 for sigmoid activation functions. It was later proved for a larger class of activation functions [2].

Theorem 3.1 (Universal Approximation Theorem, Cybenko, 1989 [1])**.** Let σ be bounded measurable sigmoidal function. Then finite sums of the form

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma\left(w_j^T x + b_j\right)$$

are dense in $L^1(I_d)$. In other words, given any $f \in L^1(I_d)$ and $\varepsilon > 0$, there is a sum, $G(x)$, of the above form for which

$$\|G - f\|_{L^1} = \int_{I_d} |G(x) - f(x)| dx < \varepsilon$$

In this experiment will see that increasing the number N of neurons in the hidden layer allows us to arbitrary approximate any continuous function on a compact set. In this report, we will work only with a one-dimensional function. We will denote by $u^N(x)$ a trained network with N neurons in the hidden layer,

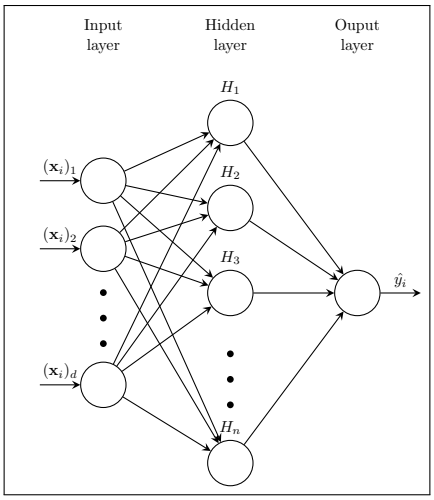

Figure 10: Network from $\mathbb{R}^d$ to $\mathbb{R}$

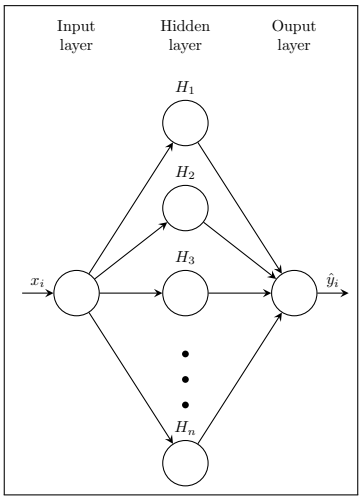

Figure 11: Network from $\mathbb{R}$ to $\mathbb{R}$

$u(x)$ the target function. Both the training and validation set are taken with an uniform distribution in $[-1, 1]$, and again the error is the sum of square error (SSE).

Experiment 3.2. Again, we can study the approximation of the function $u(x) = cos(\frac{\pi}{2} + 5x)$. We increase both the number of neurons in the hidden layer from 2 to 7, and the number of iterations. We can see from Figure 12, 13 and 14 that increasing the number of neurons seems to be a good strategy to increase the precision of our approximation, since the trained network fits our training points better.

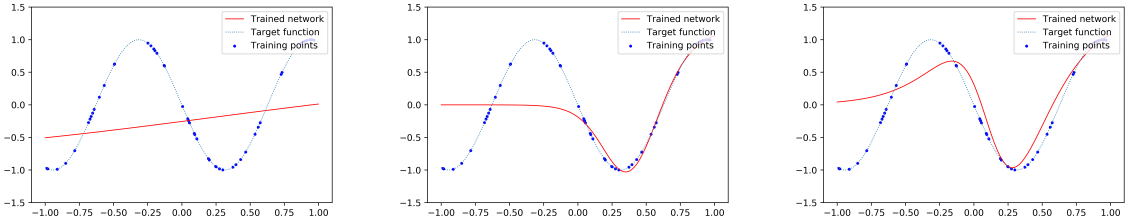

Figure 12: Fitting of $cos(\frac{\pi}{2} + 5x)$ with a 2 neurons trained network, using the sigmoid activation function, at 5, 200 and 1200 iterations.

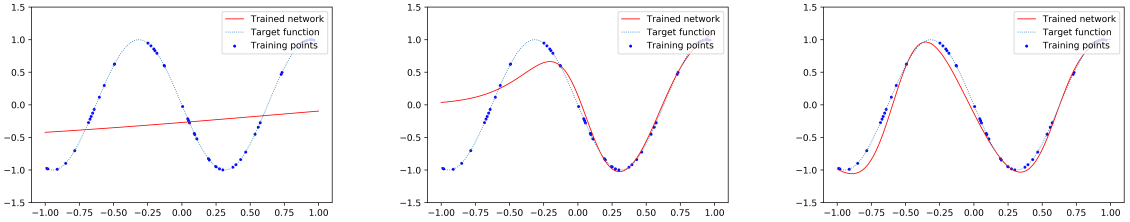

Figure 13: Fitting of $cos(\frac{\pi}{2} + 5x)$ with a 3 neurons trained network, using the sigmoid activation function, at 5, 200 and 1200 iterations.

We can confirm the expected behavior by theorem 3.1. By looking at figure 15, we have that the overall trend is that increasing the number of neurons leads to a lower SSE.

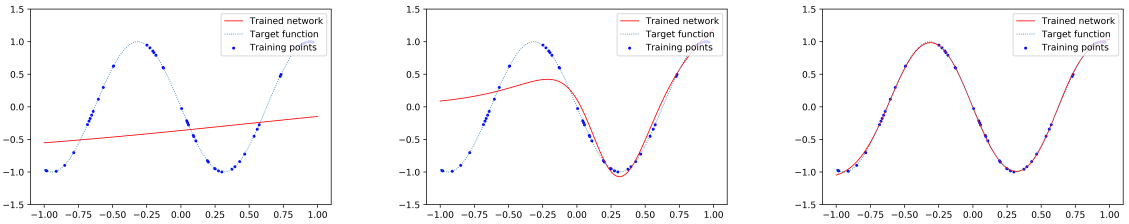

Figure 14: Fitting of $cos(\frac{\pi}{2} + 5x)$ with a 6 neurons trained network, using the sigmoid activation function, at 5, 200 and 1200 iterations. We can observe that we reach a satisfying fit of the function with only 6 neurons.

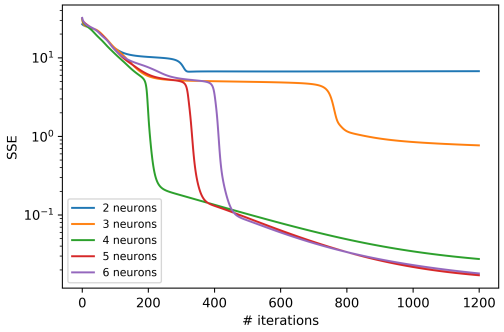

Figure 15: SSE of sinusoidal for 2 to 6 neurons

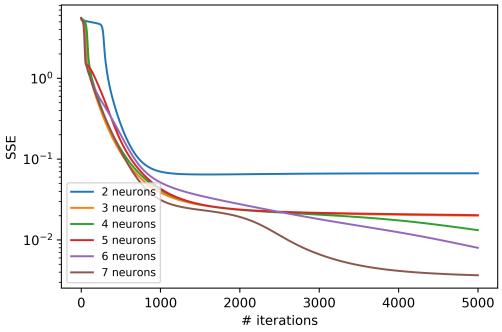

Figure 16: SSE of runge for 2 to 6 neurons

Experiment 3.3. Similarly we studied the runge function $u(x) = 1/(1 + 25x^2)$. As this function is known for its problem of oscillation at the edges of the interval when using polynomial interpolation, we checked if we could see any unusual behavior when approximating with our neural network. As expected, the SSE decreases as the number of neurons increases. From figure 16 we can see that once we reach a local minimum, increasing the number of iterations does not provide a better fit for our trained network. The conclusion seems to be that accordingly to theorem 3.1, we can reach arbitrary precision by increasing the number of neurons. However, we may not reach a local minima in the space of neural networks with n neurons. We will next see what the expected behavior when we know that a global minimum exists is and we will study the strategy to increase the probability to reach this minimum.

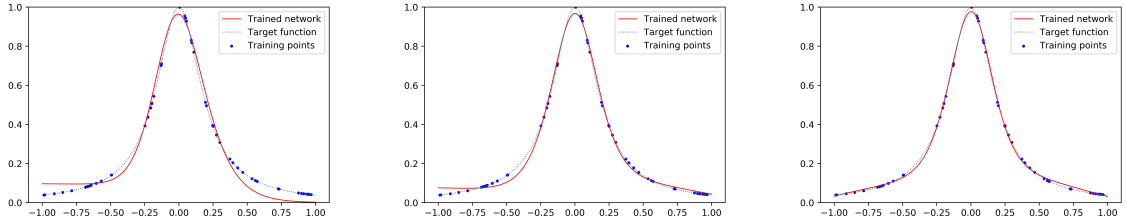

Figure 17: Fitting of the runge function $u(x) = 1/(1 + 25x^2)$ with trained network, using the sigmoid activation function, with 2, 4 and 7 neurons after 5000 iterations. We observe a similar conclusion than in the last experiment, increasing the number of neurons improves the fitting of the target function.

4 Converging to the exact representation

So far we have tried to approximate analytical target functions with trained network of the form:

$$u_\theta(x) = \sum_{j=1}^{N} \alpha_j \sigma(w_j x + b_j) \tag{4.1}$$

where α_j, w_j and $b_j \in \mathbb{R}$, $\sigma(.)$ is the activation function. In the previous experiment, the parameters were randomly initialized and trained by the stochastic gradient descent algorithm (SGD). The activation function was the sigmoid. We obtained satisfactory precision according to theorem 3.1.

What precision can we expect if the target function is of the form (4.1)? In other words, can the trained network converge exactly to the target function when the latter can be represented as a network?

For the following experiment, we chose a setting that is simple enough to be studied. Take the piecewise linear interpolation of 6 points, $\{(x_i, y_i)\}_{i=1}^{6}$. Such a function can be written in terms of a polynomial basis of degree 1: $\{\phi_i\}_{i=1}^{4} \subset \mathbb{P}^1$ From [3] we know that there exists a shallow ReLU network with 5 neurons that interpolates these 6 points. We provide a formula to write a function written in a basis of $\mathbb{P}^1$ to a ReLU network.

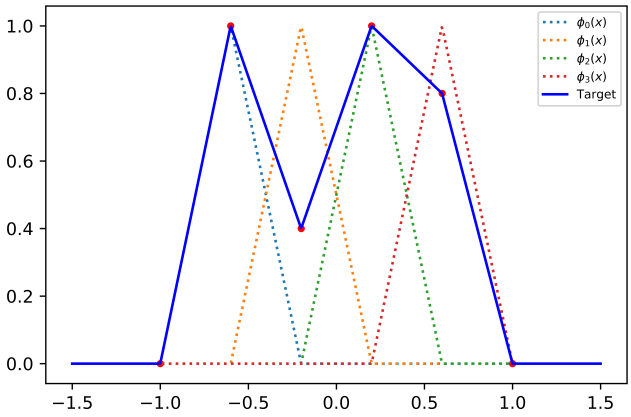

Figure 18: Target function with interpolated points and $\mathbb{P}^1$ basis

Experiment 4.1. We will consider in the following experiments trained networks of size $N = 6$ and with activation function ReLU. We can center all the 6 activation functions around each of the x_i by fixing $b_i = -x_i$, and we will fix $w_i = 1$ $\forall i$. Only the outer weights α_i are free and the problem becomes linear. Similarly to the representation 4.1, we obtain what we will call a *locked network*:

$$u_\alpha(x) = \sum_{j=1}^{N} \alpha_j \sigma(x - x_j) \tag{4.2}$$

As we can see from figure 20, we reach floating precision under Pytorch after approximately 1200 iterations. We may note that the validation error seems to stabilize above the floating precision plateau. This results from the fact that we are using more points for our validation than the training, and because we are using SSE and not mean square error (MSE) to prevent the presence of outliers. The next steps will be to free the inner bias, as the problem becomes nonlinear. Multiple questions will arise, such as the uniqueness of the exact representation or the probability to reach such parameters.

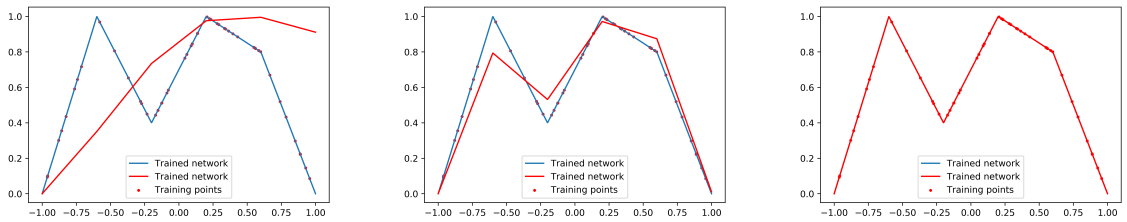

Figure 19: Fitting of the piecewise linear interpolation of 6 points, with a 6 neurons ReLU network, after 1, 100 and 1000 iterations. The target function can exactly be represented as a 6 neurons ReLU network.

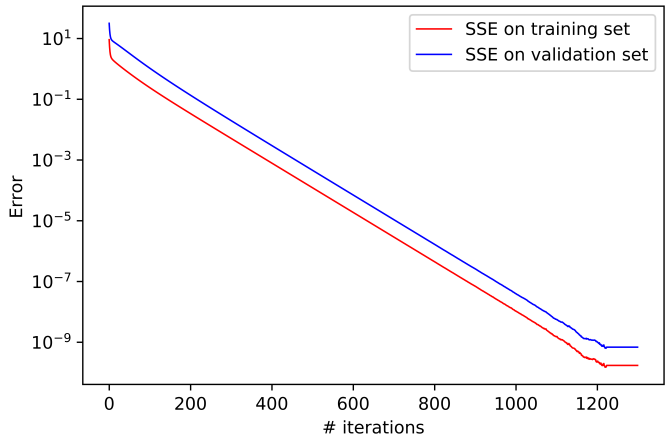

Figure 20: The trained 6 neurons ReLU network reached the exact representation of the piecewise linear interpolation of the 6 points. The SSE achieved machine precision.

Theorem 4.2. Consider the uniform grid of (a, b), with $a = x_0 < x_1 < ... < x_N = b$, with $x_{i+1} - x_i = h$, and the set $\{y_i\}_{i=0}^N$,with $y_0 = y_N = 0$. Let u be the piecewise linear interpolant of $\{(x_i, y_i)\}_{i=0}^N$:

$$u(x) = \sum_{i=0}^{N} y_i \phi_i(x) \tag{4.3}$$

with $\{\phi_i\}_{i=0}^N$ be the Lagrange polynomial basis of degree 1. Then there exist coefficients $\{\alpha_i\}_{i=1}^N$ such that u can be written as a locked network:

$$u(x) = \sum_{i=1}^{N} \alpha_i \sigma(x - x_{i-1}) \tag{4.4}$$

with

$$\alpha_1 = \frac{1}{h} y_1, \ \alpha_2 = \frac{1}{h}(-2y_1 + y_2),$$
$$\forall i \in \{3, .., N\}, \ \alpha_i = \frac{1}{h}(y_{i-2} - 2y_{i-1} + y_i)$$

Proof. After computing the coefficients α_1 and α_2, we will do the proof by induction. The proof is done by equating the equations (4.3) and (4.4). We recall the formula for the lagrange polynomial basis of degree 1:

$$\phi_i(x) = \begin{cases} \frac{x - x_{i-1}}{h} & \text{if} x \in [x_{i-1}, x_i] \\ \frac{x_{i+1} - x}{h} & \text{if } x \in [x_i, x_{i+1}] \\ 0 & \text{else} \end{cases}$$

Part 1. First we can see that since $y_0 = y_N = 0$, equation (4.3) can be written as $u(x) = \sum_{i=1}^{N-1} y_i \phi_i(x)$. We can equate the above equations:

$$\sum_{i=1}^{N-1} y_i \phi_i(x) = \sum_{i=1}^{N} \alpha_i \sigma(x - x_{i-1}) \tag{4.5}$$

Let $x \in [x_0, x_1]$, since $\phi_i(x)$ is on compact support $[x_{i-1}, x_{i+1}]$, $\forall i > 1 : \ \phi_i(x) = 0$. Since $\sigma(x) = max(0, x)$, $\forall i > 1 : \ \sigma(x - x_{i-1}) = 0$. We obtain:

$$\begin{aligned} y_1 \phi_1(x) &= \alpha_1 (x - x_0) \\ y_1 \frac{x - x_0}{h} &= \alpha_1 (x - x_0) \\ \alpha_1 &= \frac{y_1}{h} \end{aligned}$$

Part 2. Let $x \in [x_1, x_2]$, since $\phi_i(x)$ is on compact support $[x_{i-1}, x_{i+1}]$, $\forall i > 2 : \ \phi_i(x) = 0$. Similarly we obtain that $\forall i > 2 : \ \sigma(x - x_{i-1}) = 0$. Equation (4.5) becomes:

$$y_1 \phi_1(x) + y_2 \phi_2(x) = \alpha_1 (x - x_0) + \alpha_2 (x - x_1)$$

We apply the result from part 1 and we expand the basis functions:

$$\begin{aligned} \frac{y_1}{h}(x_2 - x) + \frac{y_2}{h}(x - x_1) &= \frac{y_1}{h}(x - x_0) + \alpha_2 (x - x_1) \\ \frac{y_1}{h}(x_2 - 2x + x_0) + \frac{y_2}{h}(x - x_1) &= \alpha_2 (x - x_1) \\ \frac{2y_1}{h}(x_1 - x) + \frac{y_2}{h}(x - x_1) &= \alpha_2 (x - x_1) \\ \alpha_2 &= \frac{1}{h}(-2y_1 + y_2) \end{aligned}$$

We used the fact that the grid is uniform and $\frac{x_0 + x_2}{2} = x_1$

Part 3. Now we can do start the proof by induction for $i \geq 3$. In the following we will fix $y_i = 0 \ \forall i \leq 0$. Suppose $i = 3$, similarly to Part 2 we have that:

$$y_2 \phi_2(x) + y_3 \phi_3(x) = \alpha_1 (x - x_0) + \alpha_2 (x - x_1) + \alpha_3 (x - x_2)$$

We can inject the coefficients found in part 1 and 2 to obtain:

$$y_2 \phi_2(x) + y_3 \phi_3(x) = \frac{y_1}{h}(x - x_0) + \frac{1}{h}(-2y_1 + y_2)(x - x_1) + \alpha_3 (x - x_2)$$

Then we expand the basis functions:

$$\frac{y_2}{h}(x_3 - x) + \frac{y_3}{h}(x - x_2) = \frac{y_1}{h}(x - x_0) + \frac{1}{h}(-2y_1 + y_2)(x - x_1) + \alpha_3 (x - x_2)$$

We can factorize on the LHS and remark that $2x_1 - x_0 - x = x_2 - x$:

$$\frac{y_2}{h}(x_3 - 2x + x_1) - \frac{y_1}{h}(2x_1 - x - x_0) + \frac{y_3}{h}(x - x_2) = \alpha_3 (x - x_2)$$

$$\frac{2y_2}{h}(x_2 - x) + \frac{y_1}{h}(x - x_2) + \frac{y_3}{h}(x - x_2) = \alpha_3 (x - x_2)$$

$$\alpha_3 = \frac{1}{h}(y_1 - 2y_2 + y_3)$$

Suppose there exists $n \in \mathbb{N}$ such that we have the induction hypothesis:

$$\forall i \in \{3, .., n\},\ \alpha_i = \frac{1}{h}(y_{i-2} - 2y_{i-1} + y_i) \tag{4.6}$$

Let $x \in [x_n, x_{n+1}]$, equation (4.5) becomes:

$$\frac{y_n}{h}(x_{n+1} - x) + \frac{y_{n+1}}{h}(x - x_n) = \sum_{i=1}^{n+1} \alpha_i (x - x_{i-1})$$

$$\frac{y_n}{h}(x_{n+1} - x) + \frac{y_{n+1}}{h}(x - x_n) = \sum_{i=1}^{n} \alpha_i (x - x_{i-1}) + \alpha_{n+1}(x - x_n)$$

We apply the induction hypothesis (4.6) to the sum:

$$\frac{y_n}{h}(x_{n+1} - x) + \frac{y_{n+1}}{h}(x - x_n) = \frac{1}{h} \underbrace{\sum_{i=1}^{n} (y_{i-2} - 2y_{i-1} + y_i)(x - x_{i-1})}_{:=I} + \alpha_{n+1}(x - x_n)$$

It is possible to simplify I:

$$\begin{aligned}
I &= \sum_{i=1}^{n} (y_{i-2} - 2y_{i-1} + y_i)(x - x_{i-1}) \\
&= \sum_{i=1}^{n} y_{i-2}(x - x_{i-1}) - 2\sum_{i=1}^{n} y_{i-1}(x - x_{i-1}) + \sum_{i=1}^{n} y_i (x - x_{i-1}) \\
&= \sum_{i=-1}^{n-2} y_i (x - x_{i+1}) - 2\sum_{i=0}^{n-1} y_i (x - x_i) + \sum_{i=1}^{n} y_i (x - x_{i-1}) \\
&= \sum_{i=1}^{n-2} y_i (x - x_{i+1}) - 2\sum_{i=1}^{n-1} y_i (x - x_i) + \sum_{i=1}^{n} y_i (x - x_{i-1}) \\
&= \sum_{i=1}^{n-2} y_i (x - x_{i+1} - 2x + 2x_i + x - x_{i-1}) + y_n(x - x_{n-1})) + y_{n-1}(-x - x_{n-2} + 2x_{n-1}) \\
&= \sum_{i=1}^{n-2} y_i (2x_i - \underbrace{(x_{i+1} + x_{i-1})}_{=2x_i}) + y_n(x - x_{n-1}) + y_{n-1}(-x - x_{n-2} + 2x_{n-1}) \\
&= y_n(x - x_{n-1}) + y_{n-1}(-x - x_{n-2} + 2x_{n-1})
\end{aligned}$$

Going back to our original equation:

$$\frac{y_n}{h}(x_{n+1} - x) + \frac{y_{n+1}}{h}(x - x_n) = \frac{y_n}{h}(x - x_{n-1}) + \frac{y_{n-1}}{h}(-x - x_{n-2} + 2x_{n-1}) + \alpha_{n+1}(x - x_n)$$

$$\frac{y_n}{h}(x_{n+1} + 2x_{n-1} - x) + \frac{y_{n+1}}{h}(x - x_n) + \frac{y_{n-1}}{h}(x - 2x_{n-1} + x_{n-2}) = \alpha_{n+1}(x - x_n)$$

$$\frac{2y_n}{h}(x_n - x) + \frac{y_{n+1}}{h}(x - x_n) + \frac{y_{n-1}}{h}(x - x_n) = \alpha_{n+1}(x - x_n)$$

$$\alpha_{n+1} = \frac{1}{h}(y_{n+1} - 2y_n + y_{n-1})$$

□

5 Relaxing the bias

As we have seen in the previous section, by fixing the inner weights and inner biases the problem becomes linear and by theorem (4.2) we have an explicit solution for the exact representation of the function as a neural network. The algorithm converges to machine precision and we have a global optimum.

However what happens if we let the biases free? The problem of finding the coefficients of the neural network becomes nonlinear because of the activation function. We will study how the algorithm can still reach an optimum and the probability of finding a global optimum.

Experiment 5.1. Consider the same target function as experiment (4.1), which is the piecewise linear interpolation of 6 points. We now train network with adjustable outer weights and inner biases, but the inner weight remains fixed to 1. We obtain what we will call a *semi-locked network*:

$$u_\theta(x) = \sum_{j=1}^{N} \alpha_j \sigma(x + b_j) \tag{5.1}$$

where $\theta = (\boldsymbol{\alpha}, \boldsymbol{b}) \in \mathbb{R}^6 \times \mathbb{R}^6$. We initialize multiple networks with normal distribution for the inner biases and outer weights, and train them with the goal to approximate the target function to minimize the SSE. The trained networks have the same number of neurons as the target ($N = 6$). As we can see on figure

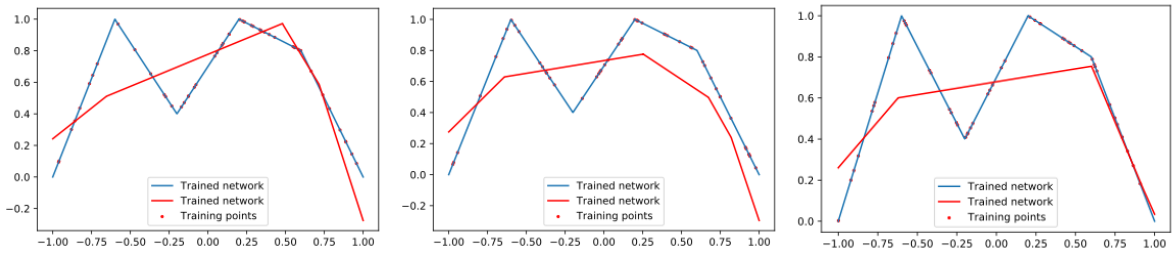

Figure 21: 3 examples of initialized 6 neurons ReLU networks, with this time the inner biases allowed to move freely, after 1500 iterations of training. These failed examples show the difficulty to reach the exact target on a semi-locked network.

21, the algorithm fails to converge to a global optimum. The trained network finds a local optimum that minimizes locally the SSE but the obtained result is far from the target function.

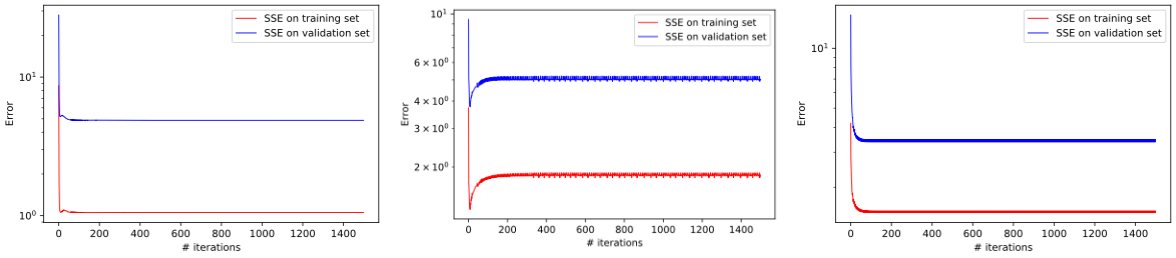

Figure 22: Error of the respective networks from the previous figure. The error is greater than 1 in all these examples and shows clearly a failure in the training.

On figure 22 the trained neural networks fails to leave the local minimum and the algorithm has stopped before 200 iterations. From this experiment we thought that the problem of finding a global optimum was highly correlated to finding appropriate initial parameters. This will be the subject of the next experiment.

Experiment 5.2. We know the exact representation of the piecewise linear function studied previously as a network. Using the theorem (4.2), the coefficients $\boldsymbol{\alpha}$ and $\boldsymbol{b}$ are given by:

$$\boldsymbol{\alpha} = (2.5, -4, 3, -2, -1.5, 0.245), \quad \boldsymbol{b} = (1, 0.6, 0.2, -0.6, -1)$$

For this experiment, we will choose the initial parameters $\boldsymbol{\alpha}$ and $\boldsymbol{b}$ instead of using random normal initialization (as in experiment 5.1). As we can see, $\alpha_i \in [-5, 5]$ and $b_i \in [-2, 2]$. So if we let $\alpha_i \in \{-5, 0, 5\}$ and $b_i \in \{-2, 2\}$ for $i \in \{1, 2, 3, 4, 5, 6\}$, we have in total $3^6 * 2^6 = 46656$ total possible initial parameters. After training with 200 iterations, with trained network the semi-locked model described and target function the piecewise linear function, we obtained 46656 final SSE. We can order all of the 727 weights and 64 bias and draw a heatmap of the SSEs. Note that the way we chose to draw the heatmap is actually a projection of the space of parameters in $\mathbb{R}^{12}$ into $\mathbb{R}^2$.

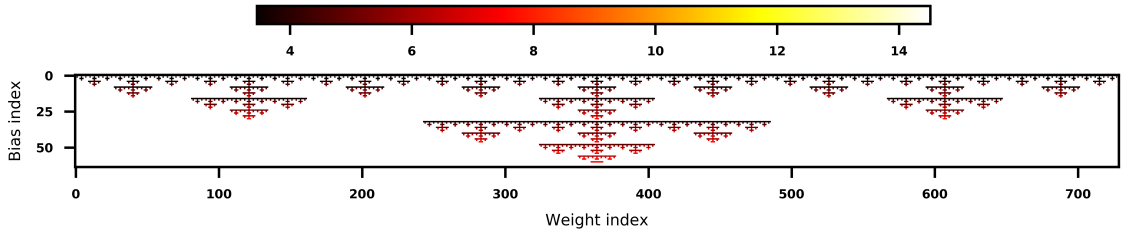

Figure 23: SSE on all the 46 656 initialized models, when the trained network and the target have the same number of neurons ($N = 6$). The inner bias and outer weights of the trained models are initialized by choosing one of the possible permutations of the parameters that we chose arbitrarily.

From figure 23 we can interpret multiple interesting behaviors. First we observe a lot of symmetries and repeating patterns. This comes eventually from the way we initialized our models, a lot of initial parameters are equal up to permutation of the coefficients in $\boldsymbol{\alpha}$ and $\boldsymbol{b}$.

We observe that none of the trials reached a global optimum. The number of local optimums reached in our experiment is low. We attained 5 different local optima with losses ranging from 3.45 to 14.51. Finally we can see a vertical line for the weight index 363. This initial weight corresponds to

$$\boldsymbol{\alpha} = (0, 0, 0, 0, 0, 0)$$

The next step will be to initialize the parameters in a greater range and unlocked the inner weights. By doing that, we hope to have a better understanding of the space of parameters and find a way to reach the global optimum.

6 Relaxing the inner weights

We will extend the discussion done in the experiment (5.2) by essentially modifying two conditions. First, we will take a smaller target and trained network by reducing the number of neurons from $N = 6$ to $N = 2$. This allows us to choose a greater range of possible parameters while avoiding the experiment to be too costly in operations. Second, we will train a network that have all the parameters free. We obtain the usual model that we will call an *unlocked network*:

$$u_\theta(x) = \sum_{j=1}^{N} \alpha_j \sigma(w_j x + b_j) \tag{6.1}$$

where $\theta = (\boldsymbol{\alpha}, \boldsymbol{w}, \boldsymbol{b}) \in \mathbb{R}^N \times \mathbb{R}^N \times \mathbb{R}^N$. Before running the trials, we may try to observe how well an unlocked network can approximate a fixed target network while having the same number of neurons in the hidden layer.

On figure 24 we can observe an example of a successful training that reached after 180 iterations the global optimum (i.e the exact representation) since the SSE is stable at machine precision. However this is not always the case, as many attempts may reach a non global optimum that can not be escaped in the space of parameters θ. On figure 25 we see a failed attempted to approximate our target function. Both networks were initialized with random normal parameters.

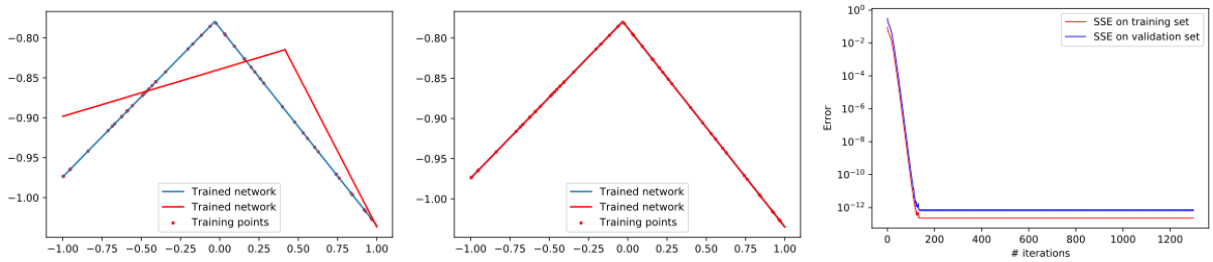

Figure 24: Successful 2 neurons ReLU network reached global optimum when targeting the same network model

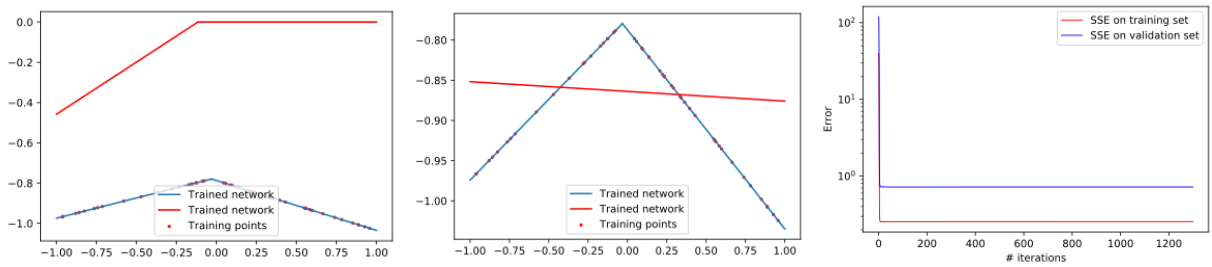

Figure 25: 2 neurons ReLU network reaching non global optimum when targeting a 2 neurons ReLU network. The non global optimum minimizes locally the loss but is far from being the exact target function.

Experiment 6.1. As we have done in experiment (5.2), take a fix target network with $N = 3$ neurons in the hidden layer and for training an unlocked network with the same number of neurons. Our target network has parameters ranging in $[-1, 1]$, so we will let the initial parameters

$$\forall i \in \{1, 2, 3\},\ \alpha_i, w_i, b_i, \in \{-1, -0.6, -0.2, 0.2, 0.6, 1\}$$

In this experiment we will have in total $6^2 * 6^2 * 6^2 = 46656$ total possible initial parameters. After training with 200 iterations and by indexing with the combination of outer weights, inner weights and inner biases, we obtain a 3D array containing all of the 46565 final SSE. Note this is essentially an arbitrary way to visualize the space of parameters in $\mathbb{R}^6$ into a projected cube in $\mathbb{R}^3$.

We can observe from figure 26 many of the same conclusions from the last experiment. We have repeating patterns and symmetry along the planes. This indicates that there may exist non-optimal minima in the space of the parameters that are hard for the algorithm to escape from. We observe that 56% of the initialized networks reached a SSE inferior to 5, and 1.2% reached a SEE inferior to 10^{-3}. It is hard to conclude from the way we initialized the parameters. This is why in the next experiments we will try to have a more precise idea of the probability of reaching a global optimum by taking this time a uniform range of parameters and using a Monte Carlo method.

7 Probability of reaching the global optimum

In the previous section, we have seen that we may encounter obstacles while trying to reach global optima. First, we know from the previous sections that the way the initial parameters are initialized is correlated to the probability of reaching a global optimum. Then we saw that freeing the parameters increased the probability to reach a global optimum. In this experiment, we will try to have a quantitative estimate of the probability to reach a global optimum as we increase the number of neurons. For this we will introduce a Monte Carlo experiment.

Monte Carlo methods designate a family of numerical algorithms with the intention of estimating a numerical value. These methods rely on repeated random sampling. Monte Carlo methods relies on the law

of large number, since the expected value of a random variable can be approximated by taking the empirical mean of independent sample of the variable.

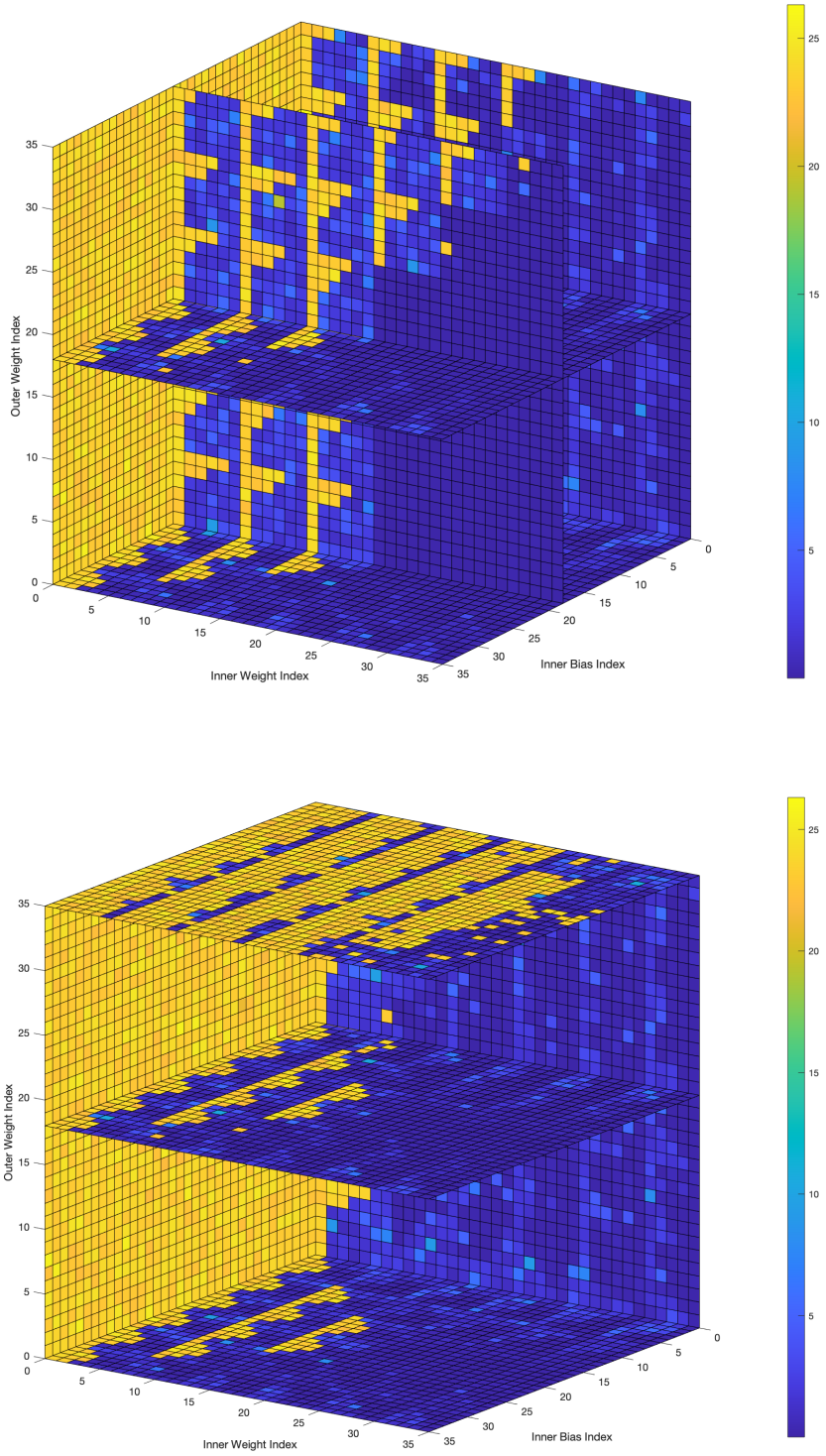

Figure 26: SSE on all the 46 656 initialized models, when the trained network and the target have the same number of neurons (N=6). The inner weights, inner bias and outer weights of the trained models are initialized by choosing one of the possible permutations of the parameters that we chose arbitrarily.

Experiment 7.1. In this experiment, we will use a Monte Carlo method to approximate the quantity :

$$q_N = \frac{global\ optima}{non\ global\ optima}$$

where N is the number of neurons of the trained networks. In the context of this experiment, q_N should be interpreted as an approximation of the quantity:

$$\frac{area\ of\ the\ region\ in\ parameter\ space\ leading\ to\ a\ global\ minimum}{area\ of\ the\ region\ in\ parameter\ space\ leading\ to\ a\ local\ minimum}$$

We use a fix 3 neurons ReLU network, it will be our target function for the entire experiment. At each of the 1500 trials, we initialize a new unlocked network with parameters ranging in $[-1, 1]$, following a uniform random distribution. Each network is then trained for 500 iterations and we observe the final SSE obtained. In our observations, we can confidently say that a network reached a global optimum if the SSE is inferior to 10^{-4} after the 500 iterations. If it is a global optimum, the quantity *global optima* increases of 1, else *non global optima* increases of 1.

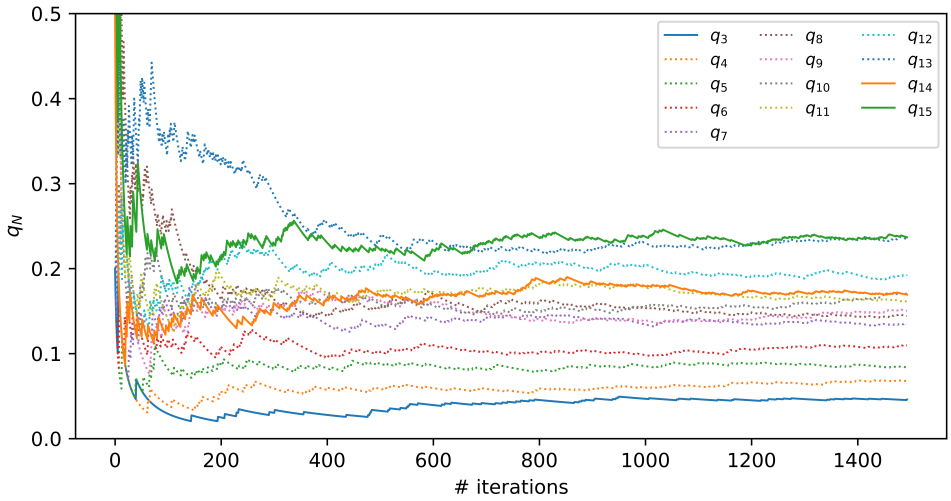

Figure 27: Ratio of global optima by non global optima, when a 3 neurons ReLU network is used as a target. As the number of neurons in the trained network increases, the ratio converges to greater value

As we can see from figure 27, we observe that increasing the number of neurons in the trained network increases the ratio q_N. This indicates that overfitting the number of neurons is a good strategy to reach global optimum.

In figure 28, we can observe the final ratio obtained after the 1500 trials of the Monte Carlo method. We have that $q_3 = 0.046$, this means that for every non global optima, the algorithm has found in average 0.046 global optima. This is a probability 0.044 of finding a global optimum. The ratio q_N increases with the number N of neurons used for training. Finally we observe that $q_{15} = 0.24$, which equates to an estimated probability of 0.19 of finding a global optimum.

Experiment 7.2. The previous experiment was done with a simple target function of 3 neurons. We see that the probability of reaching a global optimum increases quickly, but can we expect the same probability if the target network has more neurons? We try the same experiment but with different target functions. First we 4 neurons in the target network, we observe

In figure 29 and 30 we observe a similar behavior. When we increase the number of neurons the probability of finding a global optimum increases. For example, when the target neural network has 5 neurons, we have $q_5 = 0.0027$ hence the probability of finding the global optimum is extremely low. However we observe that by increasing the number of neurons, $q_15 = 0.1038$. The probability becomes then 0.094. We conclude from

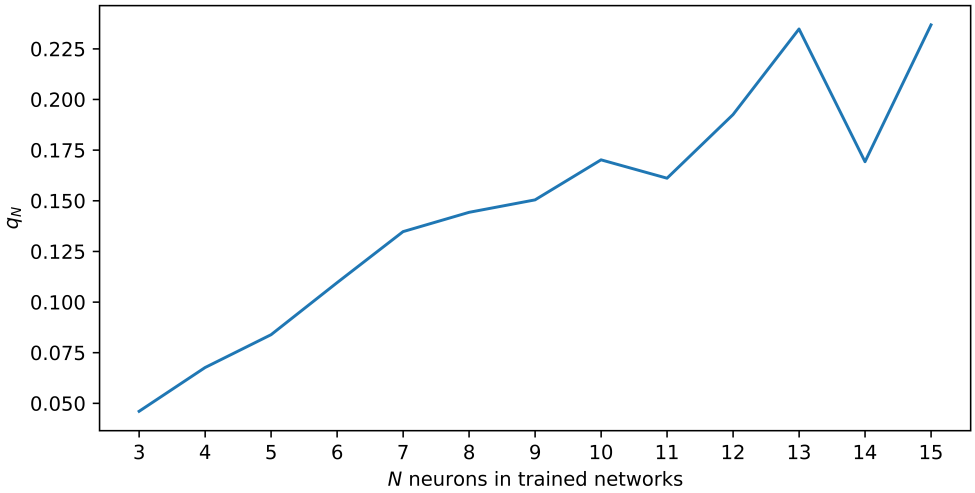

Figure 28: Final ratio of global optima by non global optima, when a 3 neurons ReLU network is used as a target. This ratio steadily increases with the number of neurons used for training

.

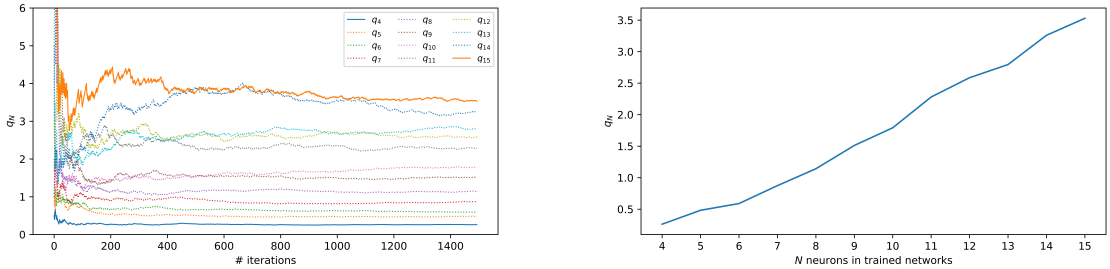

Figure 29: Ratio of global optima by non global optima when the target is a 4 neurons network.

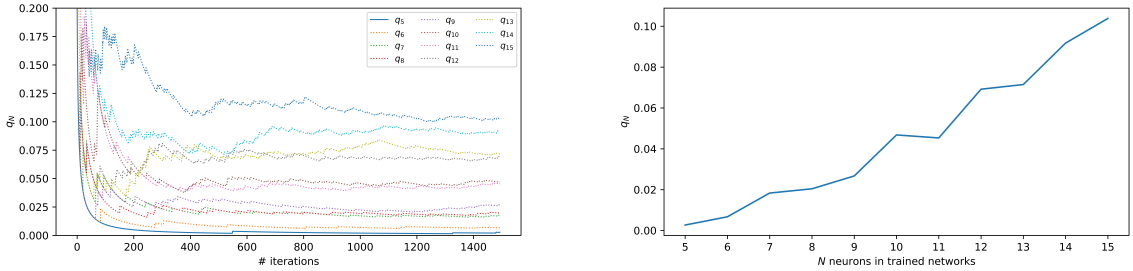

Figure 30: Ratio of global optima by non global optima when the target is a neurons network.

this experiment that overfitting the number of neurons in the trained networks is a good strategy if one wants to increase the probability of finding a network that eventually reaches machine precision.

8 Conclusion and prospects

We have discussed the approximation of neural networks from the mathematical and experimental points of view in this paper. First, we presented and constructed the neural network model studied in this report. Second, we investigated through multiple experiments how shallow neural networks can approximate continuous functions. Third, we moved our target from continuous functions to neural networks to better understand the likelihood of reaching a desired network. We provided a formula for the weights of a network interpolating a training set. In other words, we found that we can obtain the weights of an exact neural approximation without train. We observed that for a ReLU network, the problem of converging to a given network was correlated to the initialization of the parameters. Using a Monte Carlo method, we deduced that increasing the number of neurons of the trained networks, when targeting a network, increased the probability of reaching a global optimum. We wrap up this paper with the following prospects:

1. As discussed in some recent articles [3], an upper bound of the probability of reaching the exact neural network interpolation exists. Since the problem of reaching a neural network is correlated to the weight initialization method, it would be interesting to see how we could initialize the parameters to have a better estimate on this probability.
2. This report was for the most part done on ReLU network, but sigmoid or tangent activation functions are worth looking more into since they allow the SGD algorithm to escape from a local minimum. Hence, we could expect the probability of reaching the target network when using such activation functions to increase.
3. A difficulty encountered was to visualize and to have an intuition of the space of parameters for a neural network. A partial solution we found is to index the possible combinations of parameters. It may be interesting to further study this space and the information it could provide on the convergence of the associated networks.
4. Finally it is interesting and significant to extend the main experiments in this paper to multivariate functions.

Acknowledgements

I would like to thank Doctoral Assistant Boris Bonev for his counsels and precious advice during this semester. His insights in the work of a mathematician were incredibly helpful and the scientific method taught will surely shape my future projects. I would like to express my special thanks of gratitude to my project advisor Prof. Jan S Hesthaven for allowing me to study a subject that fascinates me in the MCSS laboratory. His guidance and experience in this field taught me further and beyond what I could expect from a bachelor project.

References

[1] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, December 1989.

[2] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251 – 257, 1991.

[3] Yeonjong Shin and George E. Karniadakis. Trainability and data-dependent initialization of over-parameterized relu neural networks. *CoRR*, abs/1907.09696, 2019.