Universiteit Leiden
Opleiding Informatica

The effect of activation functions and network configurations
on the performance of handwritten character recognition

Name: Rafi Taleb

Date: 23/07/2018

1st supervisor: Dr. W.A. Kosters
2nd supervisor: Dr. J.M. de Graaf

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

The effect of activation functios and network configurations on the performance of handwritten character recognition

Rafi Taleb

Abstract

Neural networks are used for different learning tasks in the last years. We will discuss the performance of different activation functions and their limitations and also the suitable configuration of the networks for different learning problems.
In this thesis we compare the performance of different activation functions in learning problem, in particular for the recognition of handwritten letters, using neural networks. We will research the effect of increasing the size of the layers and the depth of the network on the learning accuracy and speed. We will compare the results in two aspects, the accuracy and the speed.

Contents

Abstract **i**

1 Introduction **1**

2 Definitions **2**

2.1 Neural network 2

2.2 Feed-forward, Error calculation, Back-propagation 6

3 Methodology **11**

3.1 Datasets 11

3.2 Implementation 13

3.3 Evaluation measures 13

4 Results **15**

4.1 Vibration level in the network 15

4.2 Producing an appropriate error signal 16

4.3 Inputs of the neurons 18

4.4 Preprocessing the inputs 21

4.5 Higher hidden layer 23

4.6 Weight initialization 24

4.7 Comparing the functions 25

4.8 Deeper networks 28

5 Conclusions **31**

5.1 Future work 32

Bibliography **33**

Chapter 1

Introduction

Over the last years researchers are improving and developing the machine learning techniques. As some popular example we can name *Deep Learning* [4] and *Neural Networks and Deep Learning* [9]. In this chapter we give an introduction to the problem addressed in this paper. There are different kinds of neural networks used in deep learning algorithms. In this research we are going to use a specific type of networks called back-propagation neural network (BPNN). We will use gradient descent to train this network.

A deep learning network is a big neural network consisting of many layers and trained with a huge number of instance data. With deep learning networks we try to mimic the structure and the learning procedure of the biological neural systems. With this system we try to teach the network complicated patterns that appear in a variety of forms and shapes. After a large number of epochs, the network will learn to recognize the patterns.

The structure of this thesis is as follows:

- In Chapter 2 we define some concepts of neural networks and training them used in this thesis we also define the error functions and some activation functions commonly use in the area of neural networks.
- In Chapter 3 we introduce the datasets used in the experiments, the evaluation measure used to compare the results and a short description of decisions we made in implementing the code used to run the experiments.
- In Chapter 4 we show the results of the experiments with different configurations on the datasets.
- In Chapter 5 we briefly explain the conclusions

This thesis is the result of a bachelor project studied at LIACS, and is supervised by Dr.W.A Kosters and Dr. J.M. de Graaf.

Chapter 2

Definitions

In this chapter we define the concepts of the neural networks and the activation functions used to construct the networks. We will explain how the different phases of training work.

2.1 Neural network

A conceptual model of a neural network is a network consisting of neurons represented by circles and arranged in layers. The neurons of each layer are connected to the neurons of the neighboring layers with arrows which contain a weight value.

Consider the network from Figure 2.1 which is a simple example.

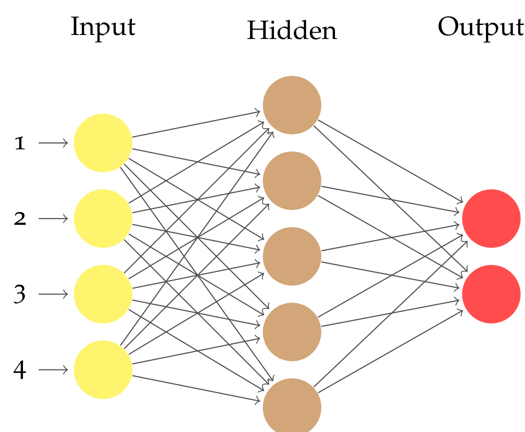


Figure 2.1: Neural Network

This network has 4 input neurons, 5 hidden neurons and 2 output neurons. We use the notation below in this thesis:

- $w_{i,j}^{\ell}$ refers to the weight of the arrow that connects the ith neuron in the ℓth layer to the jth neuron of the $(\ell+1)^{\text{th}}$ layer.

- x_i^{ℓ} refers to the input of the ith neuron in the ℓth layer.

- y_i^{ℓ} refers to the output of the ith neuron in the ℓth layer.

2.1.1 Targets

We define a target vector for each class which shows the desired form of the output when an instance of that class is detected by the network.
We define a target vector $T = (t_1, ..., t_N)$ with length N (N is the number of the classes) as the target. For the ith class the ith element t_i of this vector is 1 and all other elements are 0.
For example, if we have 10 classes, then the target vector for class 2 will be $T = (0, 1, 0, 0, 0, 0, 0, 0, 0, 0)$.

2.1.2 Activation functions:

Each neuron produces the output using an activation function. The activation function gets the input of the neuron to compute the output.
An activation function should have some properties to be suitable for using in neural networks. The activation function should preferably be **continuous differentiable**.
It should be differentiable because the network needs the derivative of the function to back-propagate the error. It should also be ***non-linear*** because we need a non-linear map from the input values to classify the instances. Because the derivative of a linear function is a constant value, this kind of function is not suitable for back-propagation networks. If we use a linear function as activation function in deep learning then it will act like a single layer linear system.

Symmetry of the activation function is a characteristic which has advantages in learning because a symmetric activation function covers the numbers on both sides of the origin.
We introduce here some common activation functions used in this research.

Softmax:

The softmax function is used in the output layer of the neural network in multi-class classification problems. Soft-max gets a vector of real numbers and converts it to a vector with the same size but the values are in the range $[0, 1]$ and the sum of all elements in the vector is equal to 1. The result is actually the categorical distribution of the classes in the K-dimensional output vector.

We define:

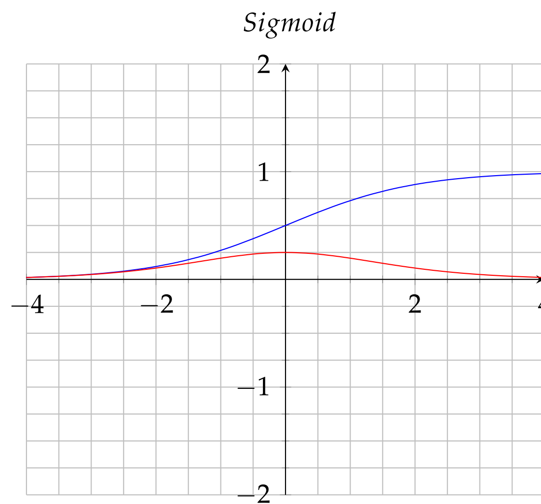
$$y_i = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}} \qquad (i = 1, \ldots, K) \tag{2.1}$$

$$\frac{\partial y_i}{\partial x_k} = y_i(\delta_{ik} - y_k) \tag{2.2}$$

$$\delta_{ik} = \begin{cases} 0 & \text{if } i \neq k \\ 1 & \text{if } i = k \end{cases} \tag{2.3}$$

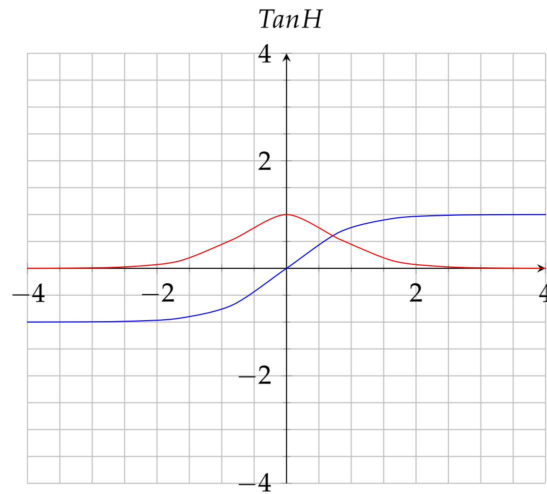
We define several activation functions:

1. Logistic (Sigmoid)



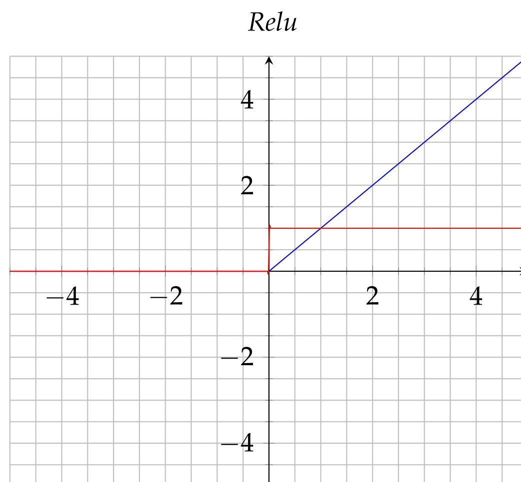
$$f(x) = \frac{1}{1 + e^{-x}}, \quad f'(x) = f(x)(1 - f(x)) \tag{2.4}$$

2. TanH



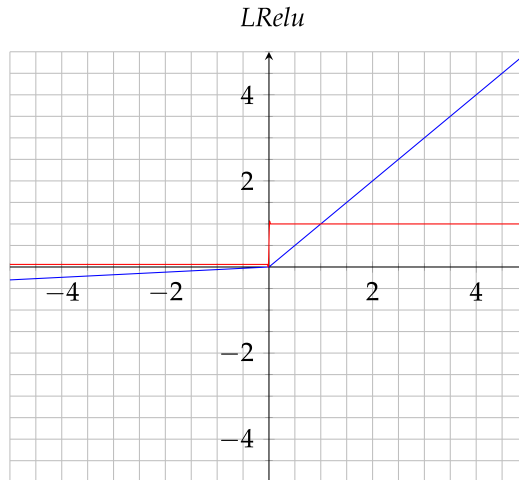
$$f(x) = \frac{2}{1 + e^{-2x}} - 1, \quad f'(x) = 1 - f(x)^2 \tag{2.5}$$

3. Relu



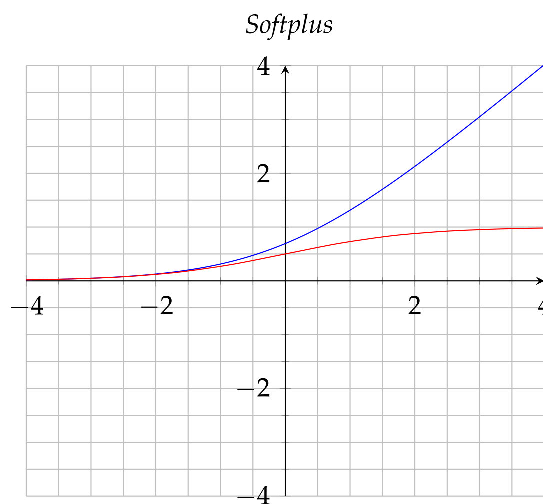
$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}, \quad f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases} \tag{2.6}$$

4. LRelu



$$f(x) = \begin{cases} 0.01 \cdot x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \quad , \quad f'(x) = \begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases} \tag{2.7}$$

5. Softplus



$$f(x) = \ln(1 + e^x) \quad , \quad f'(x) = \frac{1}{1 + e^{-x}} \tag{2.8}$$

2.2 Feed-forward, Error calculation, Back-propagation

The network is working like a function, it gets some input values and it calculate the output vector. We will explain later in details how it works.

Training a networks means feeding it with many instances and then computing the error; then we back-propagate the error in the network to change the values and parameters in the network in such a way that

it gets smaller error. We repeat this process until the network reaches a satisfactory result. The error is computed according to the distance between the actual network's output and the expected target vector.

2.2.1 Feed-forward

The input of the first layer is the input data, but the input of neurons of other layers are computed using the formula below:

$$x_j^\ell = \sum_{i=1}^{|layer_{\ell-1}|} y_i^{\ell-1} \times w_{i,j}^{\ell-1} \qquad (1 \leq j \leq |layer_\ell|) \tag{2.9}$$

where $|layer_\ell|$ is the number of nodes in the ℓth layer.

The process from the moment that the input data is fed to the network until the output of the last layer is computed is called the *forward-propagation*. Now we have a vector as output.

2.2.2 Error Function

In the last layer of the network we have to measure the distance between the actual output of the network and the target vector. The error estimates how wrong the network is in predicting the correct class. Error functions have two input vectors and they return the error value. We introduce here the cross-entropy that is a commonly used error-function. Another example of an error function is the squared error. There is a comparing research [3] on these two error functions and the cross-entropy shows better performance. In the equation of the error function below, t_i is the ith target and y_i is the ith output in the last layer.

cross-entropy:

$$E_{CE} = -\sum_{i=1}^{N} t_i \log y_i \tag{2.10}$$

2.2.3 Back-propagation

The purpose of the back-propagation phase is to back-propagate the error in the network to change the values in such a way that the error value decreases.We can do it using the **gradient descent** method.

Gradient descent algorithm: This is a method to minimize the value of a function. If we can calculate the slope of the function in point p then we can see in which direction the value of the function increases the most and we call it the positive gradient; in the opposite direction, thus the negative gradient, we have a decrease in the function value. Gradient descent works like a navigation system that guides us towards a minimum point of the error function.

To use the gradient descent algorithm in our neural network, first we need to calculate the derivative of our error function which tells us the slope of the error function in the direction which leads to smaller values of the function.

Because we want to back-propagate the error, first of all we will change the weights of arrows leading to the last layer of the network. We want to know how much change in this weight will cause appropriate change in E_{total}. We compute the gradient of the weights using the following equations.

We define:

$$\frac{\partial E_{total}}{\partial w_{i,j}^{\ell}} = \frac{\partial E_{total}}{\partial y_j^{\ell+1}} \cdot \frac{\partial y_j^{\ell+1}}{\partial x_j^{\ell+1}} \cdot \frac{\partial x_j^{\ell+1}}{\partial w_{i,j}^{\ell}} \tag{2.11}$$

$$\frac{\partial x_j^{\ell+1}}{\partial w_{i,j}^{\ell}} = y_i^{\ell} \tag{2.12}$$

$$\frac{\partial y_j^{\ell+1}}{\partial x_j^{\ell+1}} \tag{2.13}$$

equation number (2.13) is the slope of the activation function at point $x_j^{\ell+1}$.

Calculating $\partial E_{\text{total}}/\partial y_j^{\ell+1}$ depends on the error function and also the activation function of the last layer of the network wich are explained in Section 2.2.4.

2.2.4 Cross-entropy on softmax

When the last layer of the network is a softmax layer and we use the cross-entropy error function, then we calculate the error as below. Peter Sadowski explains in a note [11] an algebraic trick for cross-entropy calculations.

We define:

$$\frac{\partial E_{CE}}{\partial y_j^{\ell+1}} = -\frac{t_j}{y_j^{\ell+1}}$$

$$\frac{\partial y_k^{\ell+1}}{\partial x_j^{\ell+1}} = \begin{cases} y_k^{\ell+1} \cdot (1 - y_k^{\ell+1}) & (k = j) \\ -y_j^{\ell+1} \cdot y_k^{\ell+1} & (k \neq j) \end{cases}$$

$$\frac{\partial E_{CE}}{\partial x_j^{\ell+1}} = \sum_{k=1}^{N} \frac{\partial E_{CE}}{\partial y_k^{\ell+1}} \cdot \frac{\partial y_k^{\ell+1}}{\partial x_j^{\ell+1}} = y_j - t_j$$

2.2.5 vanishing gradient:

The vanishing gradient problem happens when we back propagate the error in the network. As the error propagates backwards in the network it is multiplied by values which are smaller than 1. This is the root of the problem, and causes a very small gradient value for the weights in the earlier layers. We call this the vanishing gradient problem. When the gradient vanishes the learning stops or it becomes very slow. Some activation functions exacerbate this problem. For example, the sigmoid function has a derivative which is never higher than $\frac{1}{4}$ which means the steepness of the function in the range $[-2, 2]$ is small, and this is the reason why using the sigmoid function makes the gradient vanish.

2.2.6 Updating the weights

During the back-propagation, we update the weights according to the calculated gradient using:

$$w_{i,j}^{\ell} \leftarrow w_{i,j}^{\ell} - \frac{\partial E_{\text{total}}}{\partial w_{i,j}^{\ell}} \tag{2.14}$$

Learning rate: minimizing a function means looking for a point of the function where the value of the function in that place is relatively small. We can search for that point with small steps sizes or with large step sizes. A small step size has the advantage of accuracy because we can be sure that that minimum point is not present in the previous calculated values, but searching with small step sizes cost much time. The other strategy is taking big steps. It is faster but always there is a risk that we jump over the minimum point when we are close to it. To mix all advantages of both strategies we mix both of them. We take big steps when we are far from the minimum point to increase the speed and we decrease the step size as we get closer to the minimum point to avoid jumping over it. In this way we take advantage of accuracy of short step method.

$$w_{i,j}^{\ell} \leftarrow w_{i,j}^{\ell} - \eta \cdot \frac{\partial E_{\text{total}}}{\partial w_{i,j}^{\ell}} \tag{2.15}$$

Here η is the learning rate.

Momentum: In the path towards the minimum point of the function there is alway the risk of being stuck in a local minimum.To solve this problem and accelerate the convergence, for each weight, we add a fraction of the last gradient and the new gradient together. The momentum will act like a gravity force to drag the function toward the global minimum. The momentum helps the system to converge faster and also it helps to overcome the local minima. Choosing a high momentum can cause the system to skip the optimum point. We have:

$$w_{i,j}^{\ell}(t) \leftarrow w_{i,j}^{\ell}(t) - \left(\mu \cdot w_{i,j}^{\ell}(t-1) + \eta \cdot \frac{\partial E_{\text{total}}}{\partial w_{i,j}^{\ell}(t)}\right) \tag{2.16}$$

where μ is the momentum and t is time.

Chapter 3

Methodology

In this section we will explain the methods we use to measure the performance of learning problems and also a short explanation of the implementation of the code and an introduction of the datasets we use in this thesis.

3.1 Datasets

Here we describe some datasets that are used to run the experiments. There are 3 types of datasets with different levels of complexity and different number of classes. Wed use these variety to test the difficulty of recognizing the handwritten letters from different aspects.

3.1.1 MNIST

The MNIST dataset [7] is a dataset of handwritten decimal digits. It consists of two sets of black and white images: a training set and a test set. The training set contains 60000 images and the test set contains 10000 images. The size of the images is 28×28 pixels. Thus each image consists of 784 pixels.The pixels are represented as numbers between 0 and 255. The number determines the gray scale value for the corresponding pixel, 0 being white and 255 being black.

The MNIST dataset is made of handwritten samples written by many high school students and employees of the United States Census Bureau. The MNIST dataset is used to train and test in many researches for machine learning and image processing. In this research we use this dataset to train and test our network. Figure 3.1 contains some examples from MNIST.

Figure 3.1: MNIST digits

3.1.2 EMNIST

The characters in the EMNIST dataset have the same format and size as the MNIST dataset [2]. This dataset contains the digits and also the alphabet characters. This dataset has 6 different splits:

1. ByClass: 814, 255 instances in 62 unbalanced classes (26 uppercase, 26 lowercase, 10 digits).
2. ByMerge: 814, 255 instances in 47 unbalanced classes (37 merged letters, 10 digits).
3. Balanced: 131, 600 instances in 47 balanced classes (37 merged letters, 10 digits).
4. Letters: 145, 600 instances in 26 balanced classes.
5. Digits: 280, 000 instances in 10 balanced classes.
6. MNIST: 70, 000 instances in 10 balanced classes.

In a balanced split, the number of instances in the classes are equal. The merged datasets give the same label for the classes with similar uppercase and lowercase. For example, "p" and "P" have the same label. The characters and the digits in this dataset are rotated, see Figure 3.2
In this thesis we use the Balanced dataset

Figure 3.2: EMNIST characters

3.1.3 MNIST with background

The MNIST with background dataset is the same as the MNIST dataset except that there is background applied to this new dataset, see Figure 3.3. There are four splits of this dataset. The dataset contains 12000 train instances and 50000 test instances.

1. mnist-rot: The MNIST digits are rotated.
2. mnist-back-rand: A random background is applied on MNIST digits.
3. mnist-back-image: A black and white image is inserted to the background of MNIST.
4. mnist-rot-back-image: The MNIST-back-image is rotated.

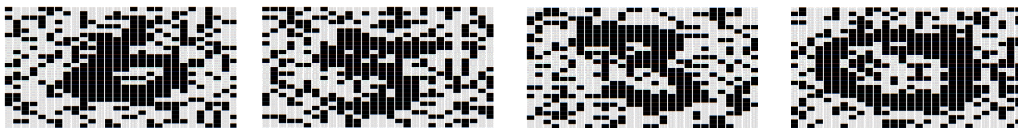


Figure 3.3: MNIST digits with noisy background

3.2 Implementation

The code used in this thesis to run the experiments is written in C++. We have built our own code in this thesis to have full control on the experiments.

The training and test data are stored in two-dimensional arrays of type "double" to benefit the contiguous locality of data in the memory and also make a balance between the space complexity and the precision of data type. Each row of the array corresponds to an instance and it keeps the pixels of that instance as double numbers.

The labels are stored in arrays of type "short integer" because all the labels are integer, and the short integer data type is suitable for them.

3.3 Evaluation measures

We need to measure the performance of different network configuration in two aspects, accuracy and speed. Note that there are many other measures, like precision and recall, but we will focus on accuracy.

3.3.1 Accuracy

To measure the accuracy of the network we will look at the percentage of the correctly classified instances in the test set.

3.3.2 Speed

To measure the speed of the networks we use two criteria; the number of the weights that the network has to update before it can achieve a certain accuracy and the complexity of the activation function used in the neurons. As the size of the network increases, the network has to update more weights in each pass. We compute the number of edges in the network and by multiplying it by the number of the instances fed to the network during the training, we can compute the number of weight updatings. We compare also the complexity of different activation functions used in this research.

Chapter 4

Results

In this chapter we will run experiments with a variety of configurations on the datasets to compare the performance of each one.

4.1 Vibration level in the network

Training a network means changing the weights in the network to achieve a combination of values of the weights that minimizes the error function for our learning problem. When we begin with the training, the network is in a state that has the most distance to that desired values, and after every training step the model hopefully changes its state to comes closer to that values. Because the network is fully connected, changing a weight value will affect many other neurons in the network; every time a weight changes in the network, we have a vibration that travels towards the output layer.

At the beginning of the training there are strong vibrations in the network and it is desired because the model is searching for correct values, but as the model learns more, the computed state becomes more valuable.

Learning every instance bring the model to a new state r, this means we will lose the previous state p and our model forgets what it has learned about the previous instance to adapt itself also to the new instance. We should try to bring the model to a state q which is between p and r to minimize this information loss. This is possible by making smaller vibrations and relatively local vibrations instead of global vibrations.

Decreasing the learning rate during the training is a way to make the vibrations smaller during the training. To make the vibrations local, we can force the network to update only some of the weights during the training. In other words, we should focus the strength of the error signal to a precise direction in the network. We call this a *spotlight method* .

4.2 Producing an appropriate error signal

In this section we are trying to find out what are the best settings for our network to achieve the highest possible accuracy. We will use a network with only 1 hidden layer with Relu functions and we use a softmax function in the output layer. After doing some experiments, we see that there is a dependence between the normalization of the input values of the softmax layer and the accuracy achieved by the network. The results of these experiments are shown in the graph below, see Figure 4.1.

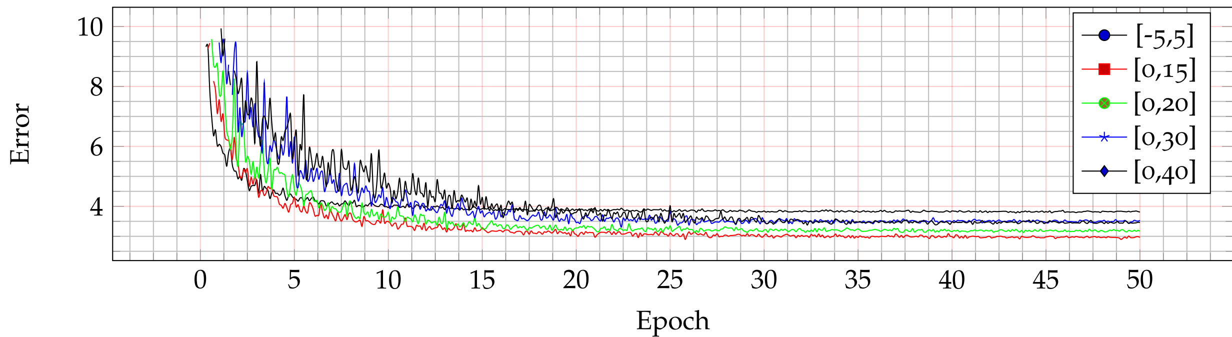


Figure 4.1: Softmax normalization on MNIST with 10 classes.

These experiments are done with a network with one hidden layer with size 30. All the setting are the same except the normalization interval for the inputs of the softmax layer (output layer). As it is shown by the results, the highest accuracy is achieved if the width of the interval is between 15 and 20. To know why this interval width is suitable for our case we look at the results shown in the tables below. In these experiments we simulate the last layer of a neural network. In this layer we use a softmax function with a cross entropy function. In the last row of the table the delta value calculated for each experiments. In the first experiment the normalization interval is $[0, 10]$, in the second experiment it is $[0, 20]$, and in the third it is $[0, 50]$.

Table 4.1: The distribution of error signal on Softmax with input interval $[0, 10]$.

	class 1	class 2	class 3	class 4	class 5	class 6	class 7	class 8	class 9	class 10
Input	2.7	2.7	2	1.8	1.5	1.3	1	0.8	0.7	0.3
Normalized	10	10	7.083	6.25	5	4.166	2.916	2.083	1.666	0
Softmax	0.478	0.478	0.025	0.011	0.003	0.001	0.0004	0.000 19	0.00011	2.1×10^{-5}
Target	1	0	0	0	0	0	0	0	0	0
Delta	-0.521	0.478	0.025	0.011	0.003	0.001	0.0004	0.00019	0.00011	2.1×10^{-5}

Table 4.2: The distribution of error signal on Softmax with input interval $[0, 20]$.

	class 1	class 2	class 3	class 4	class 5	class 6	class 7	class 8	class 9	class 10
Input	2.7	2.7	2	1.8	1.5	1.3	1	0.8	0.7	0.3
Normalized	20	20	14.166	12.5	10	8.333	5.833	4.166	3.333	0
Softmax	0.499	0.499	0.001	0.0002	2.2×10^{-5}	4.2×10^{-6}	3.5×10^{-7}	6.6×10^{-8}	2.8×10^{-8}	1.0×10^{-9}
Target	1	0	0	0	0	0	0	0	0	0
Delta	-0.500	0.499	0.001	0.0002	2.2×10^{-5}	4.2×10^{-6}	3.5×10^{-7}	6.6×10^{-8}	2.8×10^{-8}	1.0×10^{-9}

Table 4.3: The distribution of error signal on Softmax with input interval $[0, 50]$.

	class 1	class 2	class 3	class 4	class 5	class 6	class 7	class 8	class 9	class 10
Input	2.7	2.7	2	1.8	1.5	1.3	1	0.8	0.7	0.3
Normalized	50	50	35.416	31.25	25	20.833	14.583	10.416	8.333	0
Softmax	0.5	0.5	2.3×10^{-7}	3.5×10^{-9}	6.9×10^{-12}	1.0×10^{-13}	2.0×10^{-16}	3.2×10^{-18}	4.0×10^{-19}	9.6×10^{-23}
Target	1	0	0	0	0	0	0	0	0	0
Delta	-0.5	0.5	2.3×10^{-7}	3.5×10^{-9}	6.9×10^{-12}	1.0×10^{-13}	2.0×10^{-16}	3.2×10^{-18}	4.0×10^{-19}	9.6×10^{-23}

As it is shown in Table 4.3 the delta value is relatively high for the first two classes, the other classes have small delta values. This means the network will learn to change the values for two classes but it is not going to make change in the values for other classes. It also means the network will try to approach the target vector only in two dimensions and it is not learning in other dimensions. By choosing a very narrow interval as we see in Table 4.1, the delta value has a smoother distribution. In this case the network will learn in all dimensions, but because of small delta values in all dimensions it will learn very slowly.

With a wide interval the error signal is strong and local and it is suitable to separate the classes when the distance between classes are small but with a narrow interval the the error signal is weak and global and is suitable when the different classe are not close to each other. The suitable size for the interval depends on the number of classes and the degree of similarity between different classes in the dataset.

We make some experiments on EMNIST-BALANCED data set, here we have 47 classes and there are more similarity between different classes than the MNIST, for example '2' and 'z' are similar and '9' and 'q' are similar. here our normalization interval should be $[0, 20]$ to have the highest accuracy.

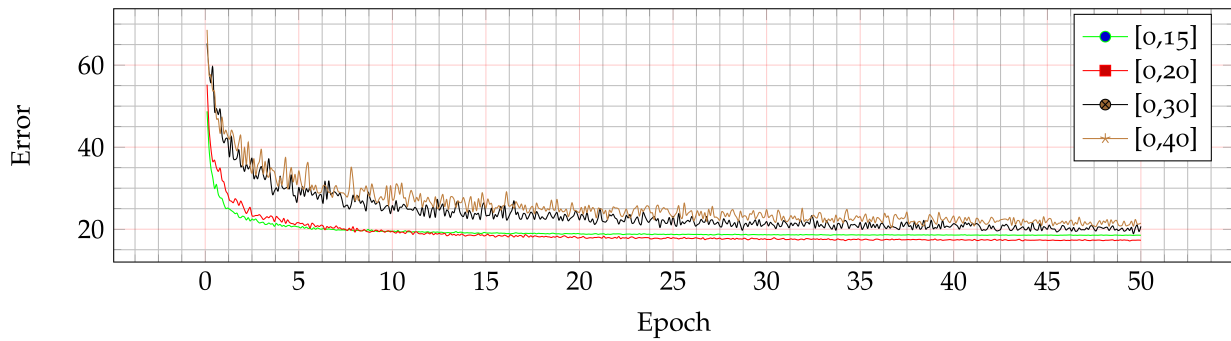


Figure 4.2: Softmax normalization on EMNIST-Balanced with 47 classes.

The strength of the error signal depends on the proportion of the value in different dimensions of the output vector in the softmax layer. It means the results for a vector with $I = (2.7, 2.7, 2, 1.8, 1.5, 1.3, 1, 0.8, 0.7, 0.3)$ is the same as the results for $J = (0.27, 0.27, 0.2, 0.18, 0.15, 0.13, 0.1, 0.08, 0.07, 0.03)$.

4.3 Inputs of the neurons

We know that the neuron works in two states, Learning state and non learning state. The neuron is in a learning state when its input value is in the learning interval of the activation function. The learning interval is an interval where the value of the derivative is larger than 0.001. In other words, when the slope of the function is larger than 0.001. Between 0.001 and zero is semi learning state. When the neuron is not in the learning state then we call it a dead neuron.

The input of the neurons is an accumulated value of the output of all connected neurons in the previous layer multiplied with the weight of the edge between them. Because it is an accumulation it can become a very large value depending on the weight of the edges and the size of the previous layer. Because we want to keep the neurons of the network in a learning state, we should take care that the input of most neurons in the network is in the learning interval.

To keep the input values in this range we apply a normalization function to the inputs of a layer before the activation function fires. The researches show that layer normalization and specially batch normalization has a significant impact on performance of the network [1,5,8]. If infinity is included in the range of the function, then normalization will help to prevent the neuron to produce large values on the output.

Setting the proper normalization range depends on the activation function used in that layer, see Table 4.4.

Table 4.4: Learning intervals of the activation functions.

	Range	Range of Derivative	Learning interval $f'(x) > 0.01$	Best results achieved
Relu	$[\,0\,,+\,\infty)$	$\{0,1\}$	$[0\,,+\,\infty)$	$[-1\,,1\,)$
LRelu	$\mathbb{R}$	$\{0.01,1\}$	$(-\,\infty,+\infty)$	$[-1\,,1\,)$
Sigmoid	$(\,0\,,1\,)$	$(\,0\,,0.25\,]$	$[\,-4.5\,,4.5\,]$	$[\,-7.0\,,7.0\,]$
TanH	$(\,-1\,,1\,)$	$(\,0\,,1\,]$	$(\,-3.0\,,+\,3.0\,)$	$(\,-7.0\,,+\,7.0\,)$
SoftPlus	$(\,0\,,+\,\infty)$	$(\,0\,,1\,]$	$[\,-4.6\,,+\,\infty)$	$[\,-6.0\,,6.0\,)$

After making some experiments on the MNIST dataset to find the appropriate normalization range for different activation functions, the results below are achieved, see Figures 4.3 to 4.7.

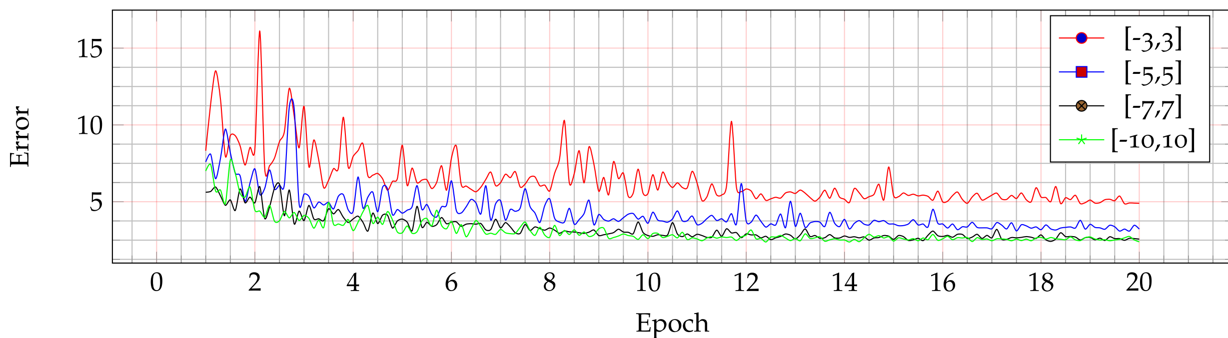


Figure 4.3: Sigmoid, comparing the normalization ranges.

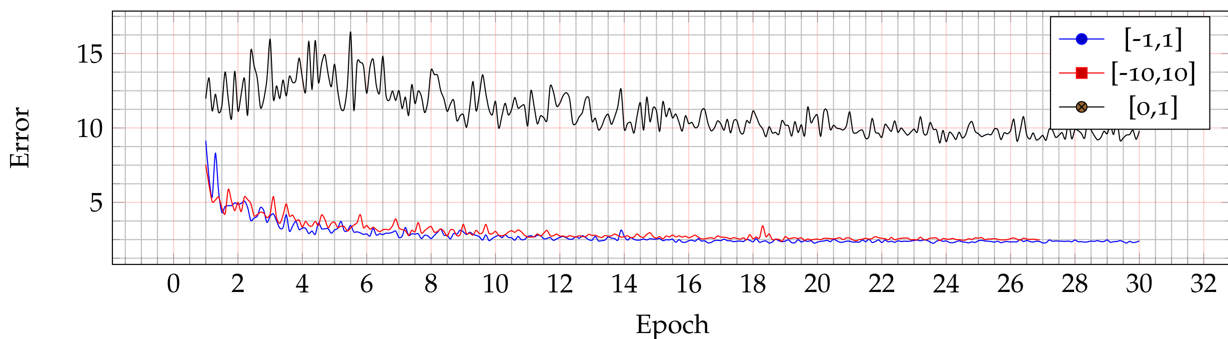


Figure 4.4: Relu, comparing the normalization ranges.

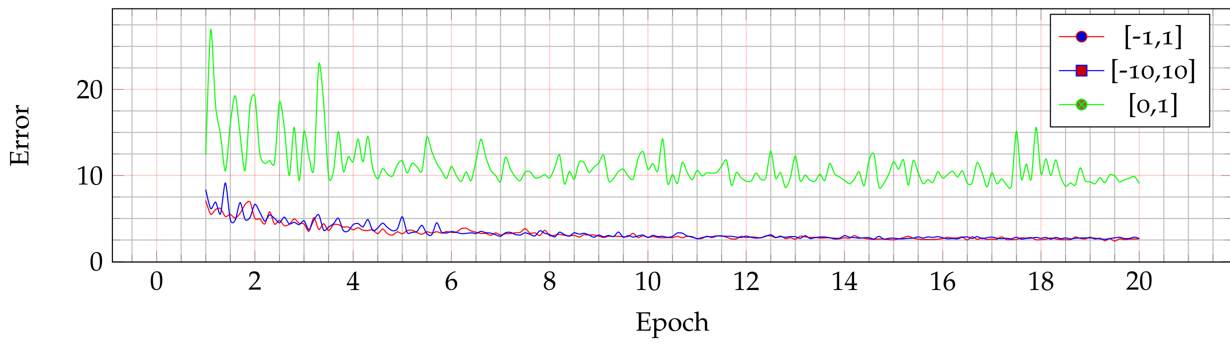


Figure 4.5: LRelu, comparing the normalization ranges.

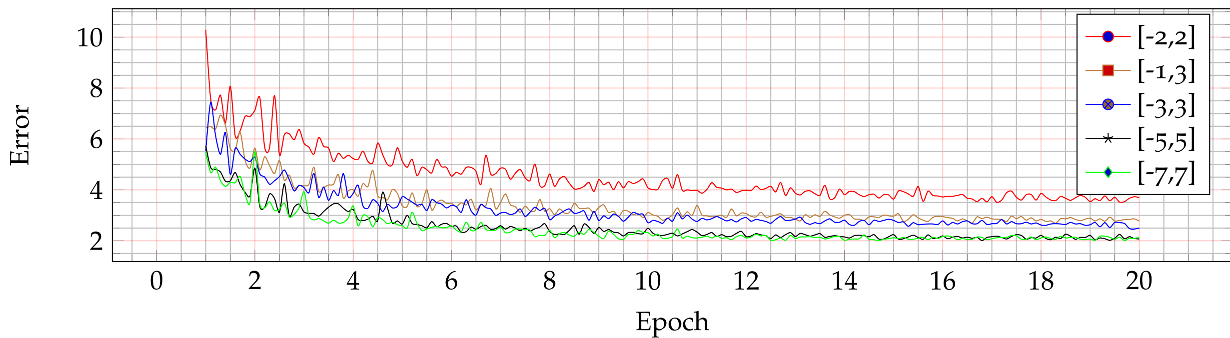


Figure 4.6: TanH, comparing the normalization ranges.

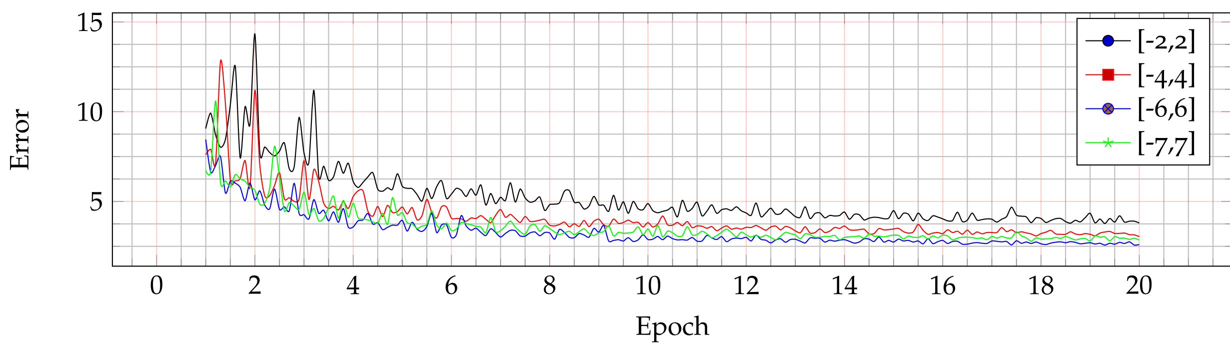


Figure 4.7: SoftPlus, comparing the normalization ranges.

As the results of the experiments show, the best interval for the normalization function is the interval in which some neurons are in high learning state. Keeping the inputs in an interval where some neurons are dead or are learning with very small values will help our model to make local vibrations rather than global vibrations and it gives the network more spotlight learning. That is the reason that in Figure 4.4 for the Relu function experiments we see that the interval $[0, 1]$ will reduce the accuracy of the model and by using the interval $[-1, 1]$ the accuracy increases.

4.4 Preprocessing the inputs

In this section we will apply some preprocessing on the input data. We will research the improvement in the accuracy and the speed of the network with different preprocessing.

4.4.1 Smaller input images

In this section we change the input images. We will make the images smaller to the $\frac{1}{2}$ size of the original images by removing the odd columns and rows in the images. the experiments shows us that using smaller instances leads to decreasing in the accuracy because we remove some input information. The time needed for training this smaller network with less edges in the first layer is much shorter.

4.4.2 Pooling

The other change to the image is making it bigger by adding extra information about the image to the end of the image. This extra information is made by calculating the average value of every 4 pixels with common border and adding this average value to the end of the image as an extra pixel. See Figure 4.8.

Figure 4.8: Example of pooling.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
3.5	4.5		

Pooling leads to decreasing in accuracy and it makes the training time much longer because the images are bigger. We think that adding this extra pooling pixels to the images make the distance between the instances of the same class larger.

4.4.3 Signature

The other preprocessing that we are going to test in this section is adding two extra rows in the image. We build a vector made of the average values of each row, we save this vector as the first row of the image and we store the average of each column at the bottom of that column. This can add extra information in the

input which gives the network a pattern that is a signature of that instance. This signature pattern will tell the network the elongation and density of the image. See Figure 4.9.

Figure 4.9: Example of Signature.

2.5	6.5		
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
7	8		

2.5	6.5	10.5	14.5
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
7	8	9	10

Adding the signature of the instance to the images did not leads to better accuracy.

4.4.4 Contrast

We change all pixels with a value smaller than 0.7 to 0, in this way we try to remove some gray halation of input information which may leads the network to confuse. According to our observation the accuracy drops after applying contrast function on the data. We think that these halation helps the network to learn more possible faces of that instance if the image was drawn some pixels to the right or left.

4.4.5 Cropping

By analysing the MNIST dataset we see that by removing 3 columns from the left side, two columns from the right side and 2 rows of the upper side there are only less than 10 percent of the images lose some pixels. we use this function to see there will be improvement in the accuracy of the network. Now we have 599 input neurons instead of 785, this means with a hidden layer with size 50 we will have 9350 less edges. As the results below shows The accuracy drops for about 0.03 percent but the training per epoch is 30 seconds shorter on our machine. We make another experiment with a cropped image but with a hidden layer with size 65 to have the same number of edges like an original size images in the first layer and we see again that the accuracy is not changed much. Thus it is always good idea to crop the image and make the input layer free of unnecessary whitespaces.

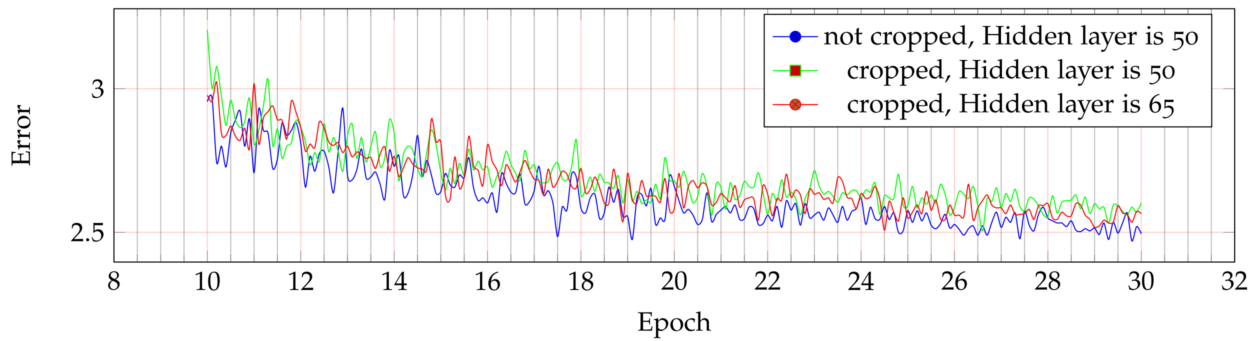


Figure 4.10: Accuracy experiments on cropped input images vs original images.

4.5 Higher hidden layer

In this section we still work with a three layer network, and we will research the impact of the size of the hidden layer on the accuracy achieved by the network. This question is also studied by other researchers [10] and our experiments approve their results.

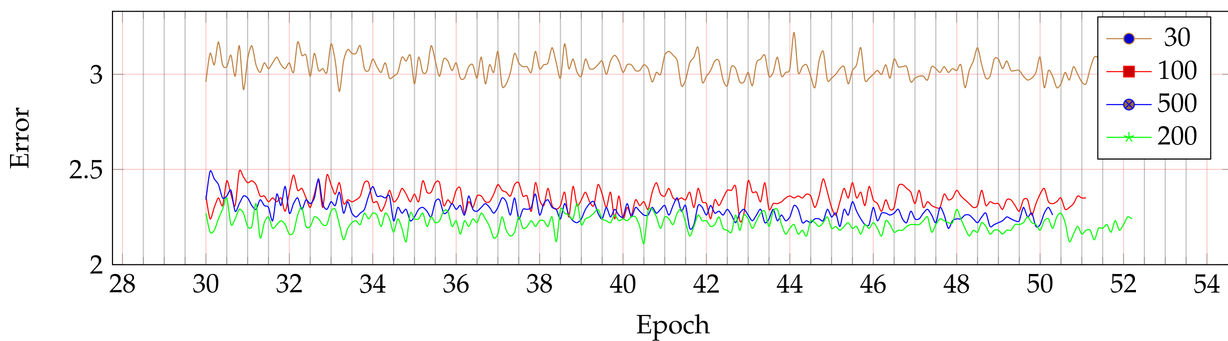


Figure 4.11: Accuracy, using different size of hidden layer.

As the results in Figure 4.11 show us, adding more hidden neurons will help the model to classify more instances correctly but there is limitation for this. Using a hidden layer with 200 neurons gives us the highest accuracy but when we increase the size to 500, we see a decrease in the accuracy and increasing in the training time. In this case when we use the MNIST dataset, adding 300 extra neurons means adding $(300 \times 784) + (300 \times 10) = 238,200$ edges and this increases the training time per epoch dramatically.

The edges are the memory of our network. Short hidden layer means small number of edges and it means the network does not have enough memory to learn the entire training dataset. With a too high hidden layer the network has enough memory to cover the entire training data set but we need more training to set all these values; it increases the training time and it can leads to overfitting. The suitable size for the hidden layer depends on the size of the training dataset.

4.6 Weight initialization

The proper weight initialization is an important topic in machine learning. Studies [6] on this topic show that choosing a suitable initialization method can improve the performance of the networks dramatically.
Consider a tennis player. Where is the best place for the player to stay when playing? If she positions her in the right corner of the court then he will lose the balls shot to the left corner. The best place for him is to stay in the middle of the court to have the smallest possible distance from both sides of the court, We have the same issue when initializing the weights in the neural networks. Because the network does not know what will be a suitable combination and the values of the weight after training, it is better for the network to initialize the weight with a normal distribution with mean zero. But what should be the variance of this normal distribution?

4.6.1 Variance

According to our experiments the variance should has a inverse correlation with the size of the network. A common formula to calculate the variance depending on the size of the layer is:

$$\frac{1}{\sqrt{h}} \tag{4.1}$$

Here h is the size of the layer.

By choosing a larger variance we can control the strength of the error signal in the last layer during the back-propagation phase. We should consider the type of the activation function when we initialize the weights, because some activation functions like the Sigmoid weaken the error signal.
In the experiment below we initialize the weights in the last layer with higher variances, by applying this the network achieved higher accuracy. See Figures 4.12 and Figure 4.13.

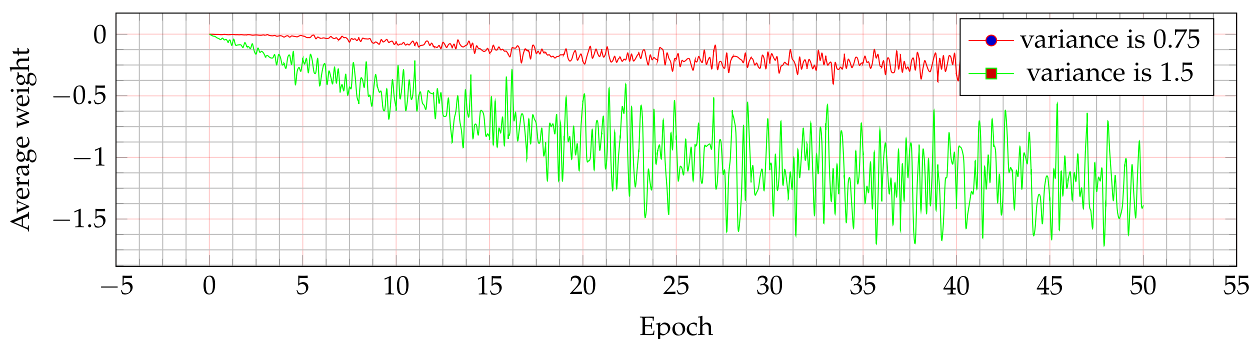


Figure 4.12: The average of the weight per different variances of the normal distribution in the last layer.

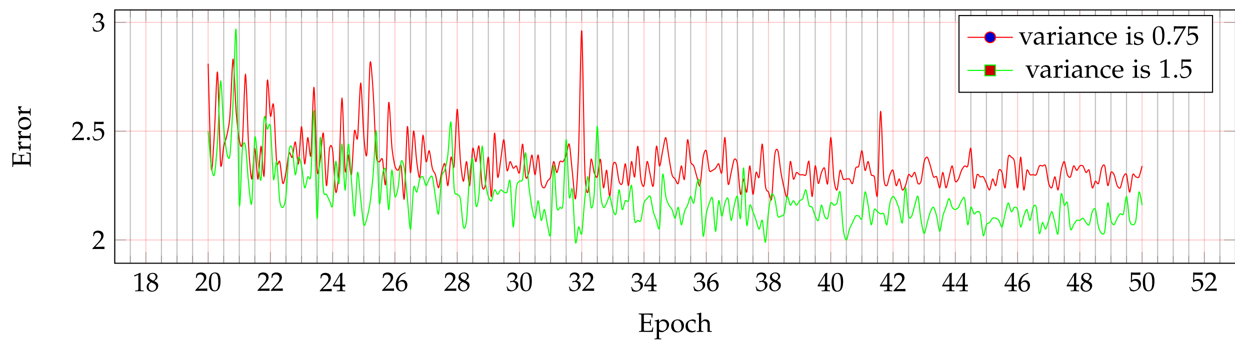


Figure 4.13: The accuracy achieved per different variances of the normal distribution in the last layer.

4.7 Comparing the functions

In this section we study the speed and accuracy of different activation functions.

4.7.1 Comparing calculating time

In this section we address the following problem: How many operations costs every edge in the network during the training?

In the forward propagation phase, the network has to do 1 multiplication and 1 sum per edge. During the back-propagation: 2 times multiplication, 1 addition and 1 subtraction.

Thus in total 3 times multiplication 2 times addition and 1 time subtraction are the all operations that an edges needs during the forward-propagation and back-propagation.

We have measured the time needed to calculate all these operations for 1,000,000 iteration to compare the time consuming by an edge and by activation function during the training. For these experiments we use a Intel® Core^{TM} $i5-3350P$ CPU @ 4.10 GHz and C++14. See Table 4.7.

Table 4.5: Activation functions time consumption comparison.

	Time in second
Relu	0.002
LRelu	0.002
TanH	0.03
Sigmoid	0.094
Edge	0.223
SoftPlus	0.27

The weight of edges in Figure 4.14 shows us how many times faster an activation function is than its adjacent.

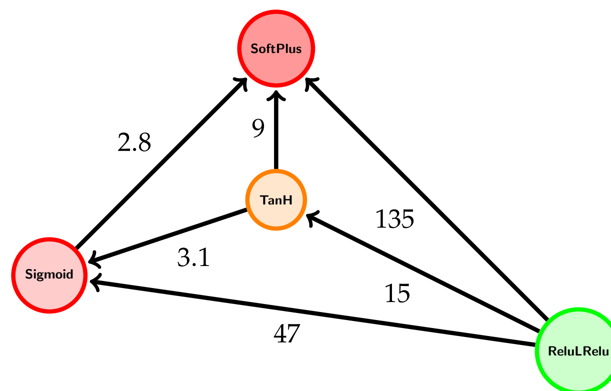


Figure 4.14: comparing the relative speed of activation functins.

Adding a single neuron to the hidden layer l will add $Size_{\ell-1} + Size_{\ell+1}$ extra edges to the network. Each single edge, needs 111 times more processing than a Relue neuron during the training. This is the reason why having high layers in neural networks is computationally expensive.

Although some activation functions are computationally more expensive than others but in the case of training neural networks for handwritten alphabet recognition it does not make much difference because the majority of computing work is spend on updating and calculating the weights of the edges, thus if we need a faster network it is better to use a compact configuration.
We build a 3 layer network, the size of the hidden layer is 200, we train the network for only one epoch on the MNIST and we compare the execution time of the training in case of using different activation function. The results shows no big difference between different functions, The Relu is 3 seconds faster than the Sigmoid and The TanH is 2 seconds faster than Sigmoid, The slowest was Softplus function which is 9 second slower than Relu per epoch.

4.7.2 Accuracy

The results shown in Figure 4.15 and Figure 4.16 show that the Relu function gives relatively better accuracy. These results are produced by one hidden layer with size 100.

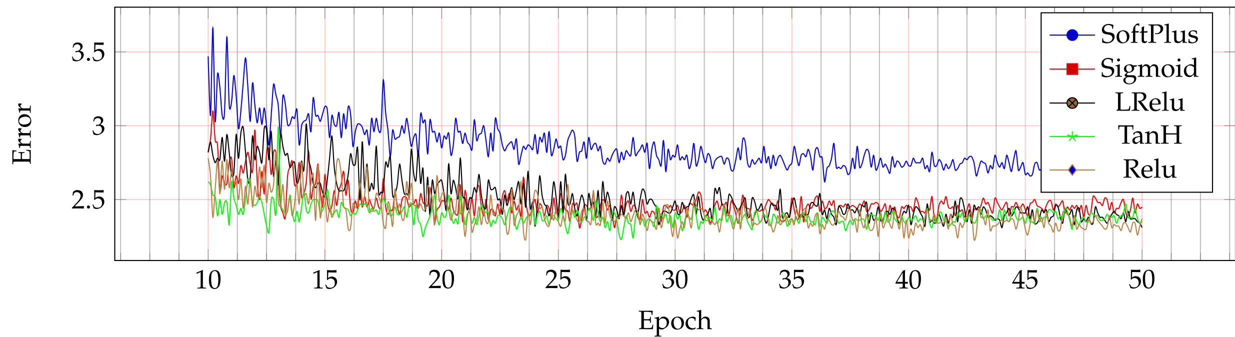


Figure 4.15: Accuracy of different activation functions on MNIST.

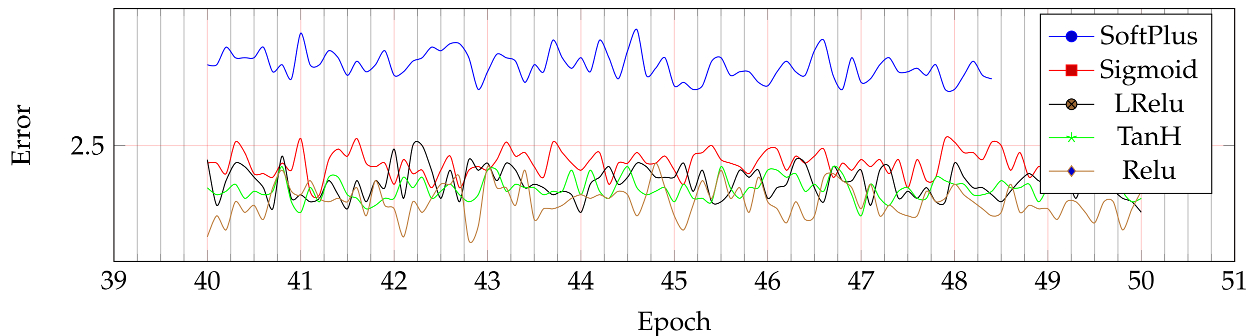


Figure 4.16: Zooming in on Figure 4.15.

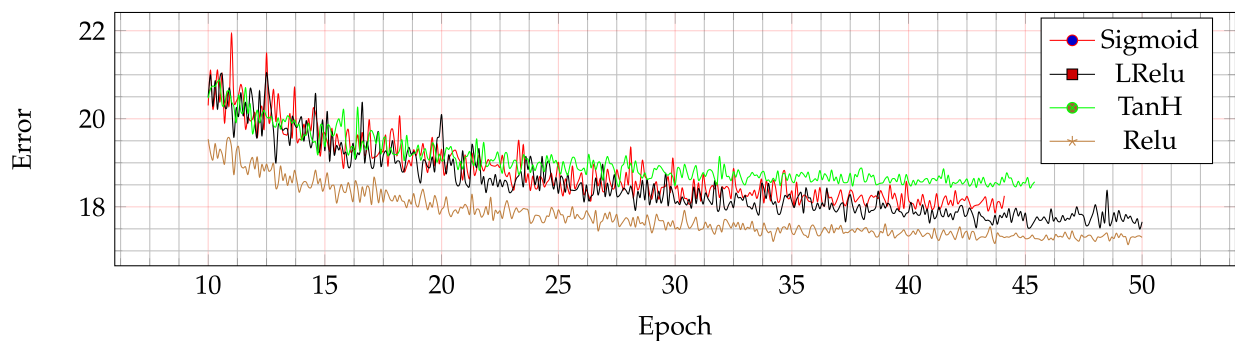


Figure 4.17: Accuracy of different activation functions on EMNIST-BALANCED.

As we can see in Figuur 4.17, the accuracy achieved on EMNIST-BALANCED dataset is about 83%, the reason is the similarity between some different classes which leads to confusion in classifying. For instance '9'and 'g' and 'q' are similar and 'o' and 'O'.

4.8 Deeper networks

Building deeper networks opens some challenges for our learning problems. The gradient may vanish or may explode. In this section we will answer two questions:

- How to avoid the problems in deep learning by using the suitable settings?
- Do we need deeper networks to classify the handwritten letters?

When we use a network with four layers the average weights of the layers grows dramatically and the network stops learning. In Figure 4.18 one can see how fast the average weight in the first layer of the network growing.

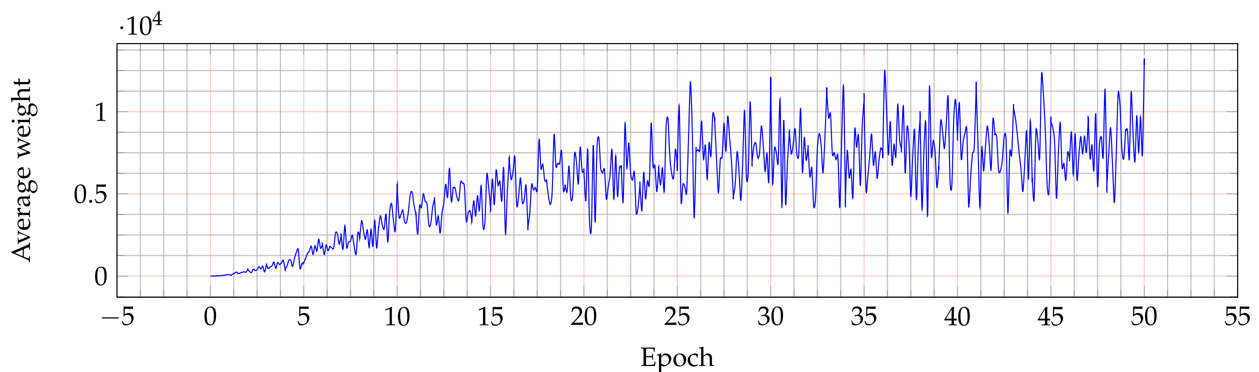


Figure 4.18: The average weight, 4 layers, Relu, learning rate is 0.5.

In this experiment we are trying to train a 4 layer network build of Relu neurons. The derivative of the Relu is 1 for the values larger than zero and it causes the gradient of the network to get big values during the back-propagation and it is the reason for heavy weight edges in the network. We use three techniques to reduce this effect in deeper networks:

1. Choosing a smaller learning rate.
2. Initializing the weights with smaller values.
3. Using a stabilizing layer.

In the experiment below one can see how the weights become smaller when we use a small learning rate like 0.009. See Figure 4.19.

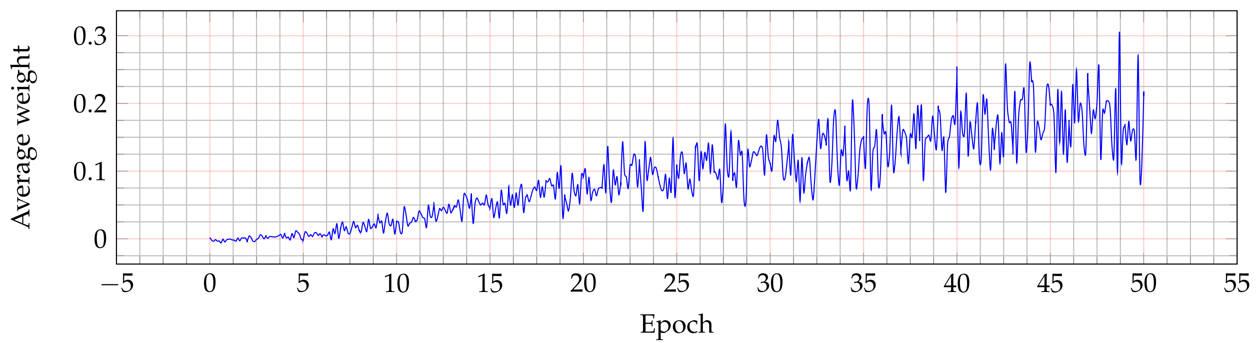


Figure 4.19: The average weight, 4 layers, Relu, learning rate is 0.009.

We do not have this exploding growth in the weights when using a sigmoid function (see Figure 4.20 and Figure 4.21), because the sigmoid function has a derivative in the range $[0, 0.25]$. We can use a sigmoid layer as an stabilize layer to avoid heavy weights. This will control the growing weights problem.

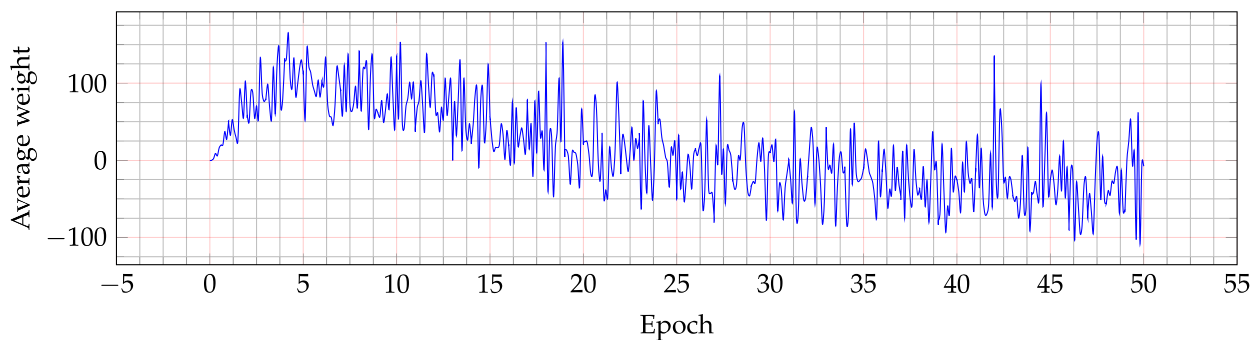


Figure 4.20: 4 layer, Sigmoid, learning rate is 0.9.

Our experiments show us that the TanH activation function is the easiest function to work with when training deep networks. By using the TanH function we can use larger learning rates without getting heavy networks. The range of this function is $(-1, 1)$. The range of Relu and Sigmoid consist of only positive values and we think this is the reason why working with TanH function is easier.

Do we really need deep networks for handwritten letters recognition?
According to our experiments on MNIST dataset, having only one hidden layer with a suitable size is enough for the task of classifying. Adding more hidden layers only leads to a longer training time and some difficulty to choose the appropriate setting for the networks. We think that the classification of MNIST is a simple task and that is the reason using a simple network is enough to learn this task.

We will train networks with some more complex datasets like MNIST with background. We hope that deeper networks will help to filter the background of the image and achieve better accuracy in comparison to the 3 layer networks.

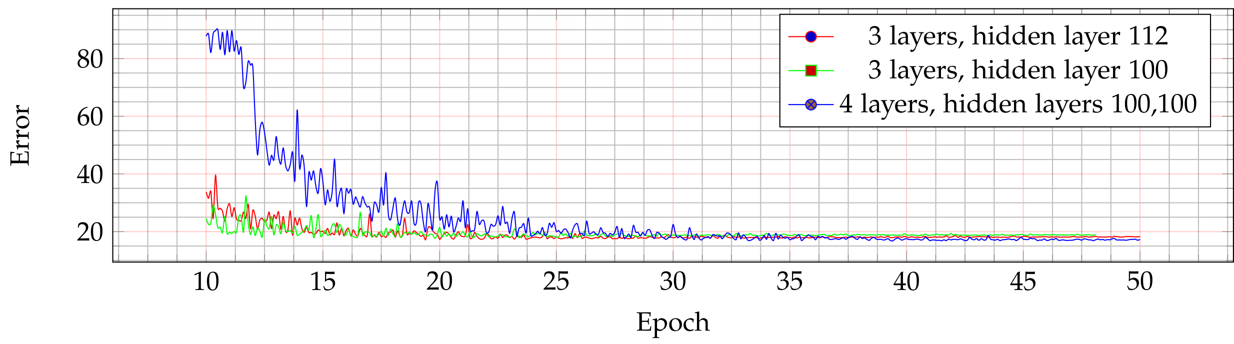


Figure 4.21: Accuracy of 1 hidden layer networks vs 2 hidden layers on MNIST˙WITHBACKGROUND.

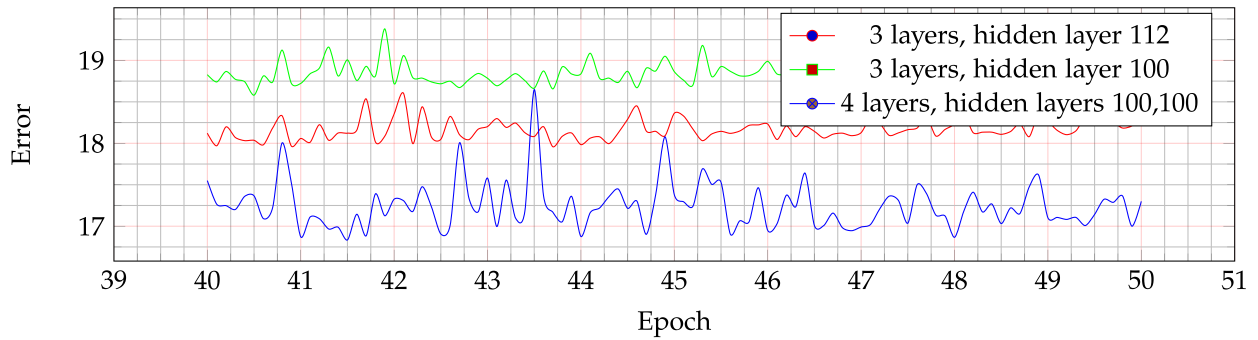


Figure 4.22: Zooming in on Figure 4.21.

The results in Figure 4.21 and Figure 4.22 show that adding an extra layer to the networks will improve the accuracy if we are working with some more complex data but the time needed to decrease the error is longer for a deeper network.

Chapter 5

Conclusions

In this thesis we have examined the influence of the size of the layers and the depth of the networks on the accuracy and the speed of learning. We have also studied the initialization methods and the normalization ranges for different activation functions and their performance.

In this research we conclude the following points from the experiments:

- If the input data has a simple representation like the MNIST dataset without several abstract dimensions, a network with only one hidden layer will learn faster and easier than a deeper network. Training a network with 2 hidden layers in case of complex input data with noisy background will give us better accuracy but more training epochs are needed for convergence.
- The size of the hidden layer depends on the size of the training set, bigger training sets need bigger hidden layers. For the MNIST dataset we get the best result with a hidden layer with size 200.
- Keep the vibration level low in order to stabilizing the network by:
  - Initializing the weights with appropriate values. The initial values of the weights depend on the size of the network, for bigger layers a smaller variance for the normal distribution should be used.
  - Normalizing the input values of the neurons such that only some of the neurons of the network work in learning state and others stay in semi learning state to maximize the generalization. This will give stability to the network.
  - Using appropriate batch size. The batch size depends on the training set size. We find that for the case of the MNIST dataset a batch size of 50 gives the best results.
  - Using the spotlight method by finding the correct normalization range for the Softmax layer to localize and concentrate the error signal to the faulty parts of the network. This normalization

range depends on the number of classes and the similarity between the different classes.

- The distance between the calculating time of different activation functions studied in this thesis is not so much that it can lead to a big difference in training time, but the accuracy and the speed of convergence achieved by the Relu function is the best in all our experiments.

- Using the TanH function which is a zero centered function, is the best choice if we want to work with a deeper network. TanH does not suffer from heavily growing weights in the deep networks and can work with bigger learning rates. The Sigmoid function is not suitable for deep networks because its derivative is not bigger that 0.25 and it leads to gradient vanishing.

5.1 Future work

We mention the following suggestions for future work:

- Using several connected cooperating networks. The network in the first layer classifies the similar classes, then the networks in the next layers recognize the exact subclass. In this way some networks will be specialized to recognize the details of the similar classes and will be able to classify them with more accuracy.

- Recognizing the letters individually and isolated from the words can lead to confusion in recognizing similar classes like '0' and 'O'. We think by recognizing the letters in the words we can classify the letters with more predictions. The future plan of the research can be the studying the combining the word recognition to help the recognition of the isolated letters.

Bibliography

[1] J. Lei Ba, J.R. Kiros, and G.E. Hinton. Layer normalization. 2016. ARXIV, eprint arXiv:1607.06450, `https://arxiv.org/pdf/1607.06450.pdf`.

[2] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik. EMNIST: An extension of MNIST to handwritten letters, 2017. `http://arxiv.org/abs/1702.05373`.

[3] P. Golik, P. Doetsch, and H. Ney. Cross-entropy vs. squared error training:a theoretical and experimental comparison. Interspeech, Lyon, France, Aug. 2013.

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

[5] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training byreducing internal covariate shift. 2015. ARXIV, eprint arXiv:1502.03167, `https://arxiv.org/abs/1502.03167`.

[6] S. Krishna Kumar. On weight initialization in deep neural networks. 2017. eprint arXiv:1704.08863 `https://arxiv.org/pdf/1704.08863.pdf`.

[7] Y. LeCun, C. Cortes, and C.J.C. Burges. MNIST: Database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`.

[8] Z. Liao and G. Carneiro. On the importance of normalisation layers in deep learning with piecewise linear activation units. 2015. ARXIV,eprint arXiv:1508.00330 `https://arxiv.org/abs/1508.00330`.

[9] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. `http://neuralnetworksanddeeplearning.com/`.

[10] F.S. Panchal and M. Panchal. Review on methods of selecting number of hidden nodes in artificial neural network. *International Journal of Computer Science and Mobile Computing*, 3:455–463, 2014.

[11] P. Sadowski. Notes on backpropagation. 2016. `https://www.ics.uci.edu/~pjsadows/notes.pdf`.pdf (online).