Deep Learning Course

Homework 1

-

# MNIST Digits Classification with MLP

André Cavalheiro - 2019403065

## Contents

## Introduction

### Data

The provided data is a known dataset, MINIST, which contains 70,000 images (28x28 pixels) of handwritten digits (from 0 to 9). The data was split into training and validation using one seventh of the data for testing and the rest for training. The objective of this project is to train a neural network to predict which digit is displayed in the image with the highest accuracy possible (at least 98%).

During this assignment only fully connected neural networks were explored which means our input must be a single vector. Given the previously stated image size, the vector must have 784 entries, one for each pixel (28*28). By doing this we lose some information on spatial relationships, since we're not providing any information on which pixels are next to each other, but such themes are out of the range of this assignment.

The output of our neural net will be a 10-dimension vector with each entry representing the likelihood of that image representing the specified digit (from 0 to 9 respectively). We can then compare the output to a one-hot encoded vector representing the actual digit that was present in the image.

### Model

During this project we experimented with two different architectures of neural networks. One with a single hidden layer and another with two hidden layers. Models with such a small number of hidden layers can hardly belong to the subject of deep learning but are good for understanding the basic functionality of the algorithms.

Adding hidden layers to a neural network makes the model more flexible, which, for a naïve approach seems to be exactly what we want. If the point of a neural network is to learn a certain function, then if we make it more flexible won't it learn the function easier? Well, unfortunately it's not that simple. A machine learning algorithm learns from the training data, and if we give it to much freedom it will mold perfectly to this training data, learning the useful features and the noise altogether. This phenomenon is known as overfitting, we can verify if our model is overfitting by comparing evaluative measures from the training data and testing data, particularly precision.

A thing to consider whenever we talk about hidden layers is the width of the layer, or the number of hidden nodes in it. Since such a thing is a different thing to tune, and testing every single hypothesis is impossible, I decided to try values corresponding to powers of 2 ($2^n$).

## Hyperparameters

Later in this report we'll be comparing different activation functions, and different loss functions. In order to truly understand the impact of changing these we'll keep the rest of our hyperparameters fixed. For that we must choose some what decent values that won't cripple our future experiments. In this section we'll explore which values were chosen and why. In a perfect world we would use cross validation to find the optimal values for every single hyperparameter, but since each of them influences all the others and time is limited it was not possible to perform this for every single one of them. Instead I experimented with a few ranges of values which made sense for the current problem, made different combinations of them and empirically decided on which to use. They're probably not the optimal values, but since they resulted in pretty good results, I'd say they're good enough to take the required conclusions for this assignment.

It's also worth mentioning that the initial standard deviation for each layer was set at $\frac{1}{\sqrt{layer\ input\ size}}$.

## Learning rate

The learning rate is a parameter used to update the weights between neurons in the backpropagation process. It defines how large this update is and therefore tuning it can vastly change how long it takes for the algorithm to converge, or if it can even converge at all. Too big of a learning rate can force the algorithm to never converge, being stuck not being able to find the optimal way. Too small and it'll take a huge amount of time and computational power for it to converge, or even worse, you can get stuck in a local minimum that is still far away from the optimal solution. Basically, it determines how much an updating step influences the current value of the weights.

Normally it varies between 0.0001 to 1. For this project, I tried varying the order of magnitude between these values, finding very slight changes between some of them. I decided to fixate the value at 0.01 which is a safe bet and achieved pretty good results. This value was actually chosen taken into account the chosen value for the momentum hyperparameter which will be discussed in the next section.

Another method that I would have liked to explore is to have a dynamic learning rate, (also known as annealing learning rate) which would diminish as we approximate the convergence, using gradient error history to adapt this value. However, that falls under the optimization categories which I did not have time to explore in depth.

## Momentum

By using momentum, we're making use of previously calculated gradients to update our weights. We do this to try and avoid being stuck in local minimums, since using only the current gradient could be sometimes misleading. Basically, by using momentum we're smoothing the direction in which our optimization is heading.

However, just like with the learning rate, having to big of a momentum can mean we overshoot the optimal solution. Too low and is as if we're not using momentum at all. Whenever we're tuning in this parameter, we must also take into consideration the learning rate value, since both influence the magnitude of the updates. If both are kept at high values the chances of us overshooting and missing the optimal solution becomes higher.

After experimenting a bit, with both momentum and learning rate, the value was fixed at 0.5 which achieved the best results.

## Weight Decay

Weight decay is a known way of regularization in neural networks. Just like the parameters in the two previous sections, weight decay also influences our weight updates, this time by multiplying weights at the end of each batch for a defined constant. The principle behind regularization is simple, we penalize weights proportionally to their magnitude to prevent overfitting, and weights with too high of a value. The chosen value for this was 0.01

## Batch Size & Max epochs

Since we're using stochastic gradient descent, the batch size defines how many test samples are used between weight updates. The trick is to try and find an amount which converges to an optimal solution without making our program take an eternity to run. The maximum number of epochs obviously defines when our program stops, ideally it would always be when the accuracy increment after each batch is practically zero, which means our program as converged and further training will not be useful. Yet, once again this could mean that our program would take an eternity to end and therefore selecting a constant number of epochs is more practical for the current problem. Trying to find a balance between these two parameters the batch size was fixed at 100, while the maximum number of epochs was made dynamically, ranging from to 60 depending on the convergence or lack thereof for each case.

## Comparing Activation Functions

Activation functions are a very important part of every neural network since they are the ones that induce the non-linearity in the learned function. In this section we'll be comparing the performance of our models using two different activation functions, them being Sigmoid and RELU whose graphs appear in fig … .

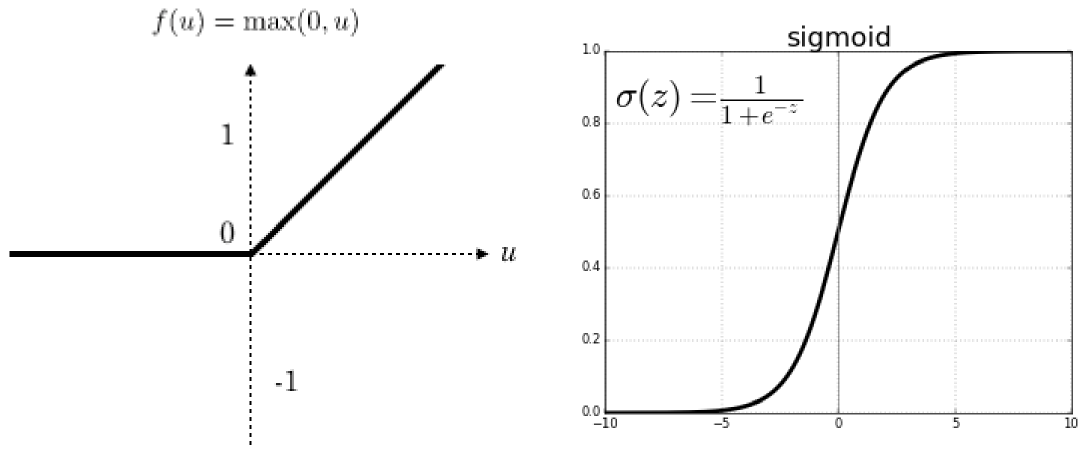

FIGURE 1 –ACTIVATION FUNCTION PLOTS

Both have their pros and cons. Sigmoid maps its input to a value between 0 and 1 which can be helpful when we want to see the output as a probability. It's also differentiable and monotonic although its derivative it's not which can lead to problems known as the exploding gradient and vanishing gradient (the exponential growth or shrinking of the gradient as it gets multiplied during backpropagation). The second be caused by the saturation of neurons (its output value being very close to zero or one), which causes the gradient to be nearly zero and learning extremely slow.

On the other hand, RELU is the most used activation function right now for its well-behaved derivative which avoids the previously mentioned problems. However, this function does have its fair share of problems. The fact that it's not differentiable at 0 which can be a problem to output continuous functions (which is not the case for the current problem) and can result in a dead neuron which always outputs the same value despite its input, this value being zero. Since the gradient at zero is also considered to be zero, the neuron will never recover from this state and will be utterly useless for the network.

It's worth mentioning that, in order to be able to test the functionality of the activation function alone, the loss function had to be fixed, and therefore the Euclidean loss function was chosen.

## One Hidden Layer

Below in Table 1 we present a few of the test cases, which seemed more relevant for the question at hand.

| *Layer Width* | **Sigmoid NN Accuracy (30 epochs)** | **RELU NN Accuracy (30 epochs)** | **Sigmoid NN Accuracy (60 epochs)** | **RELU NN Accuracy (60 epochs)** |
|---|---|---|---|---|
| *512* | .86 | .93 | - | - |
| *1024* | .86 | .97 | - | - |
| *2048* | .84 | .95 | .91 | .98 |

TABLE 1 - ACCURACY FOR A SINGLE HIDDEN LAYER NN WITH DIFFERENT ACTIVATION FUNCTIONS

One thing that can directly be pointed out is that the NN using the RELU activation function converged to its optimal solution a lot quicker than its competitor. The sigmoid activated NN displayed signs of very slow learning, taking several more epochs of training to achieve accuracy values close to what takes the RELU activated NN only a couple of epochs. This proved to be true in practically every single tested scenario.

From a certain point on, the accuracy improvement started to be painfully slow, increasing one a few decimals in the amount of accuracy displayed if so, which made it difficult to understand it the convergence was already achieved or not. To combat this problem, the tests under the best possible scenario for 30 epochs were replicated with 60 epochs, which displayed an increase in accuracy for both models. Probably the accuracy of both could yet be increased slightly but the time necessary to do so and rerun the experiments was not seemed as worth it.

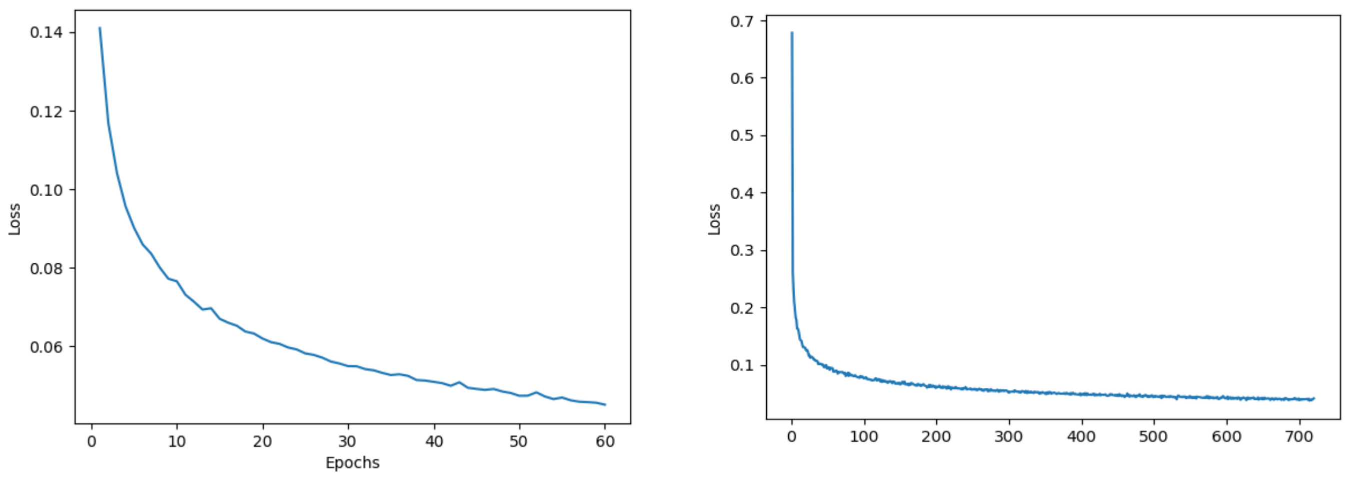

FIGURE 2 – LOSS VALUES DURING TESTING(A) AND TRAINING(B), FOR A RELU NN - 2048 NODES – 60 EPOCHS

## Two Hidden Layers

| Layer 1 Width | Layer 2 Width | Sigmoid NN Accuracy (30 epochs) | RELU NN Accuracy (30 epochs) | Sigmoid NN Accuracy (60 epochs) |
|---|---|---|---|---|
| 1024 | 256 | .86 | .97 | - |
| 1024 | 1024 | .87 | .98 | .88 |

TABLE 2 - ACCURACY FOR A 2 TWO LAYER NN WITH DIFFERENT ACTIVATION FUNCTIONS

Once again, the RELU proves to perform better than its competitor in all the tested cases. In table 2 we display some of the more meaningful results. Due to the number of parameters needed to tune in with multiple hidden layers, the creation of more complex networks would result in a ridiculous execution time. The obtained results are enough to conclude that in every circumstance tested, using RELU converges to an acceptable solution a lot faster than the when using sigmoid. To verify if the low accuracy was due to a small amount of training, the experimented was tested once again for the double number of epochs, yet since the accuracy remained almost the same, we can safely say that the sigmoid simply performs worst for the current problem.

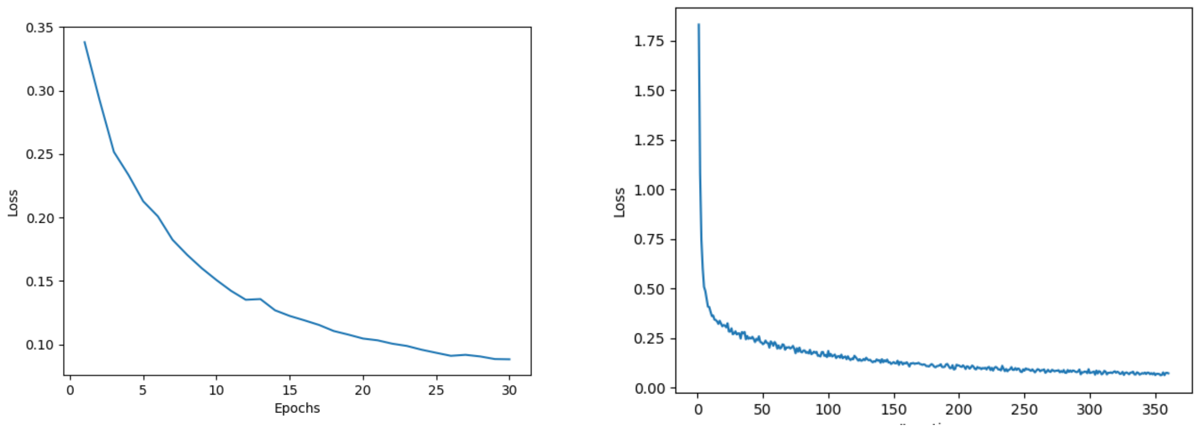

FIGURE 3 – LOSS VALUES DURING TESTING(A) AND TRAINING(B), FOR A RELU NN – 1024 - 1024 NODES

## Comparing Loss Functions

The final step of either one of our neural networks is always the loss layer whose job is to evaluate how far away our predicted values are from the actual results. The all point of our model is to minimize our loss function and therefore is of the most importance to choose the right one. We're now about to compare the Euclidean loss to the SoftMax cross entropy loss so first we should understand their strengths and weaknesses.

The Euclidean loss tries to minimize the squared distance between our predictions vector and the one-hot encoded vector representing the ground truth. This formula can be generated by performing maximum log-likelihood estimation over the conditional probability distribution of the output. However, for multi-class classification problems the cross-entropy loss function is more commonly used.

It's worth mentioning that, in order to be able to test the functionality of the loss function alone, the activation function had to be fixed, and therefore, taking into consideration the results of the last section, the RELU activation function was chosen.

## One Hidden Layer

Unlike the activation functions, the loss functions performed both well, achieving very close results. Despite his, the SoftMax cross-entropy seemed to slightly outperform the Euclidean loss for most of the situations. It's important to note that the maximum number of epochs was fixed at 30.

| ***Layer Width*** | **Euclidean NN Accuracy** | **SoftMax CE NN Accuracy** |
|---:|:---:|:---|
| *512* | .93 | .96 |
| *1024* | .97 | .96 |
| *2048* | .95 | .97 |

Table 3 - Accuracy for a single hidden layer NN with different activation functions

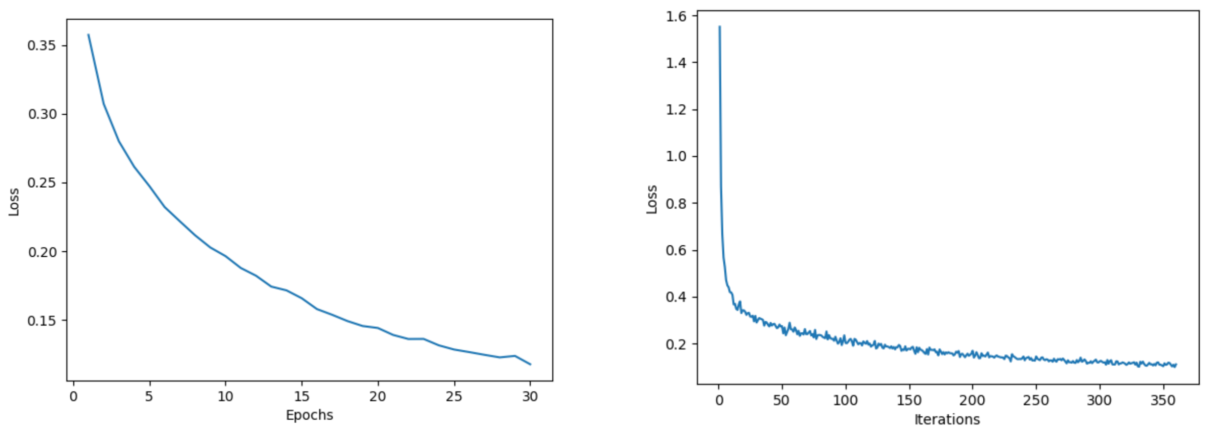

Figure 3 – loss values during testing(A) and training(B), for a SCE NN – 2048 nodes

## Two Hidden Layers

The results remain the same as for the previous case, proving both loss functions were suitable for the current problem. Although their functionality is different, they both serve the required purpose and can be used for this task. Although the results for 2 hidden layers does not exactly crush its predecessor, we can see that overall presents are slightly better, proving that the flexibility provided by an extra layer eases the learning task without overfitting the training data.

Taking into account both the results from table 3 and 4 we can say that it appears the softmax cross-entropy seems to be a slight better choice for the current problem, however given the fact that the results are so close to each other, we cannot be sure of this. The results would have to be repeated several times in order to ensure the truthfulness of this. Despite this fact we can safely say that both of the loss functions performed pretty well during this task.

| Layer 1 Width | Layer 2 Width | Euclidean NN Accuracy | SoftMax CE NN Accuracy |
|---|---|---|---|
| 1024 | 256 | .97 | .97 |
| 1024 | 1024 | .98 | 98 |

TABLE 4 - ACCURACY FOR TWO HIDDEN LAYER NN WITH DIFFERENT ACTIVATION FUNCTIONS

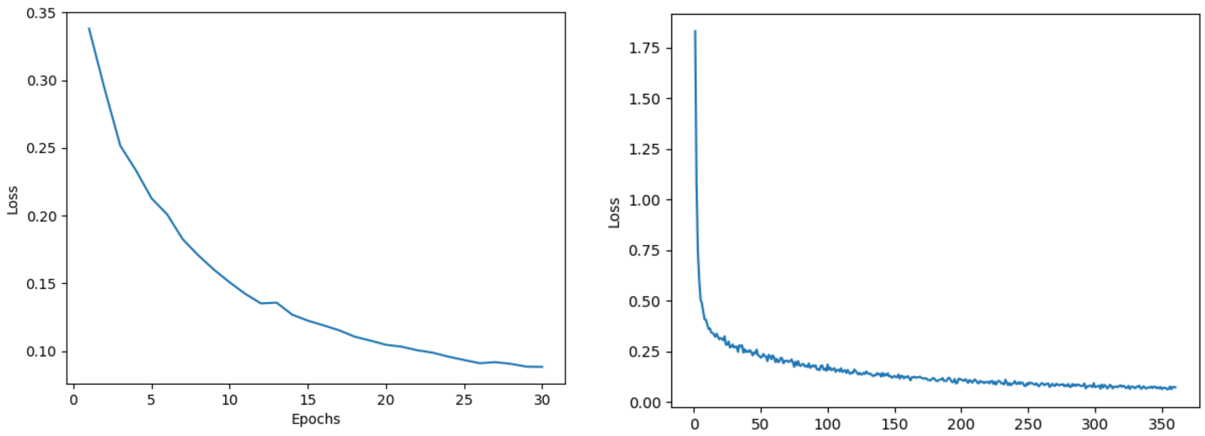

FIGURE 4 – LOSS VALUES DURING TESTING(A) AND TRAINING(B), FOR A SCE NN – 1024-1024 NODES