Classification of Image Data Using a MultiLayer Perceptron

by Ahmad Hendie, Nadia Hawarri and Malena Mendilaharzu

ABSTRACT

Artificial neural networks (ANN) are computational techniques inspired by the architecture and dynamics of the brain. The fundamental principle behind artificial neural networks is that they learn, just as human and animal brains, by changing the connections between their neurons. The main purpose of this paper lies on studying how the multilayer perceptron (MLP) architecture proposed by Werbos in the late 70s, can be used for the classification of digits from images. More precisely, we studied whether changing both the number of hidden layers of MLPs and the number of processing elements in the hidden layers affected the accuracy of the model. We also investigated whether different activation functions yield different results and whether regularization and normalization techniques had an influence on the performance of the model. After constructing a MLP using stochastic gradient descent and performing numerous experiments, we found that with increasing number of layers and processing units, the model seemed to perform better. We also discovered that ReLU was the most efficient activation function. Finally, we found that the L2 regularization technique did not seem to provide much for this problem and that normalizing the images is a very important step when using gradient descent.

Introduction

Among numerous neural network architectures, a particular interesting one, with the name of perceptron was introduced by Rosenblatt in 1958. It consisted in connecting McCulloch-Pitts neurons into layered feed-forward networks to process information. The simplest version of perceptron allowed to graphically visualize and solve linear separable classification problems. Later, more complex perceptron models containing hidden layers were created to solve non-linearly separable problems: these were called the Multilayer Perceptron. They are formed from three different types of layers - the input layer, the hidden layer(s), and the output layer. The input layer receives the input signal to be processed. The hidden layers are placed between the input and output layer and are considered to be the true computational engine of the MLP. Finally, the output layer performs the task of prediction and classification. The neurons in the MLP are trained with a back propagation algorithm that allows to compute the gradient and the loss function with respect to the weights of the network.

Multilayer Perceptron is commonly used for pattern classification, recognition, prediction and approximation. They excel in machine translation and are in fact the computing engine behind Google Translate. In this paper, we studied how they can be used for the classification of handwritten digits obtained from the MNIST Dataset. We were particularly interested in testing which factors affected the accuracy of our model. Consistent with previous research, we found that "the number of hidden layers in a multilayer perceptron, and the number of nodes in each layer, can vary for a given problem. In general, more nodes offer greater sensitivity to the problem being solved, but also the risk of overfitting" **[1]**. Further, we found that different activation functions yield slightly different results: due to a number of factors we concluded that ReLU was the best choice for the activation function. Finally, we noticed that L2 regularization was not needed for this problem and that "input data normalization is crucial to obtain good results" **[2]**. In fact,

normalizing the images improved the ability of stochastic gradient descent to find local minima that were closer to the global minima.

The MNIST Dataset

The MNIST database is a large database of handwritten digits that is commonly used for training image processing systems. It contains small square 28x28 pixel gray scale images of digits between zero and nine. More precisely, it is divided between 60 000 images intended for training the model and 10 000 images intended to test it.

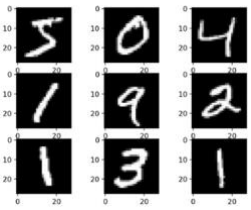

Figure 1.1: *Visualization of the MNIST Database*

In order to comprehend what the results of our experiments would be we decided to study the class distribution of each number to see if the database was properly balanced. Both the training and testing sets were well distributed with almost equal amounts of data for each class.

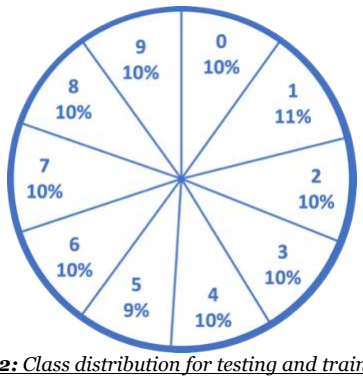

Figure 1.2: *Class distribution for testing and training sets of the MNIST Database*

Results

Architecture of the network vs Accuracy

We began by studying the effect of both the depth (number of hidden layers) and the width (number of units in each layer) on the accuracy of the model.

Regarding the depth, we expected an increase in the accuracy of the model as we increased the number of hidden layers. We hypothesized that this would be due to the fact that as we increase the number of hidden layers, we are increasing the number of connections in the network or the parameters that need to be learned. The higher the number of connections the more complex the functions that can be learned by the network, and the ability to learn more complex functions means potentially better accuracy. In Figure 2.2 we can see how an MLP with 2 hidden layers performs better than one with one or zero hidden layers. However, after further research we concluded that although the accuracy of our model increases as we increase the number of hidden layers this is not always the case. In fact, as the Universal Function Approximation Theorem suggests, only one hidden layer with sufficient neurons is enough to model any problem. However, doing so may be inefficient due to the exponential increase in computations. Rather than checking how many hidden layers we have we must be checking if we have enough neurons to model the complexity of the problem. The same argument applies to the width: we observed that increasing the number of units in the hidden layers increased the accuracy of the model, however in the long run continued increase of the width will not result in an increase of accuracy.

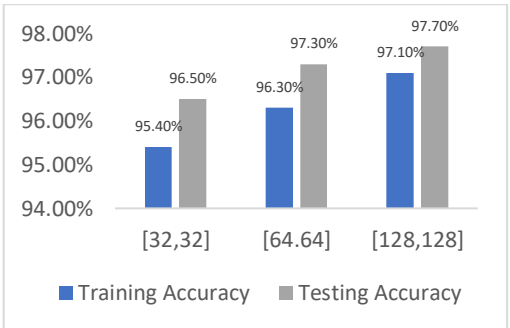

Figure 2.1: *Number of units of a 2 hidden layer MLP vs Accuracy*

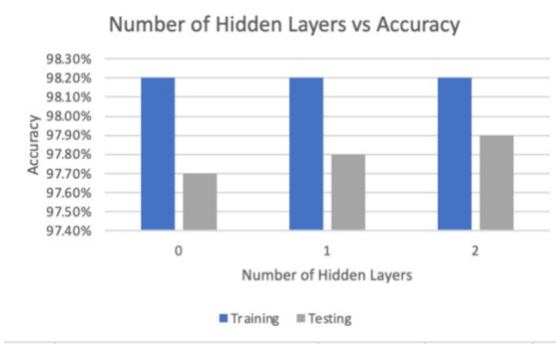

Figure 2.2: *Number of hidden layers vs accuracy*

Activation functions vs accuracy

Next, we studied how different activation functions affected the accuracy of our model. The activation functions under evaluation were ReLU (lr=0.01), tanh (lr=0.01), and sigmoid (lr=1). Close to optimal learning rates were chosen for each activation function so that we could compare them using their best performance. These tests were ran using a MLP with 2 hidden layers and 100 iterations. What we expected to find is that ReLU performs the best, and sigmoid performs the worst. The reason why we expected ReLU to perform the best is because it addresses the vanishing gradient issue which affects both tanh and sigmoid. This is because when we have multiple stacked sigmoid/tanh layers, by the backpropagation derivative rules we get multiple multiplications of their respective derivatives. And as we stack more and more layers, the maximum gradient decreases exponentially. ReLU on the other hand has a gradient of either 0 or 1, therefore eliminating the issue of the vanishing gradients. We also expected tanh to perform better than sigmoid, because the outputs of tanh center around 0 (whereas the ones for sigmoid around 0.5) which makes learning for the next layer a little bit easier. Our results were not consistent with our expectations. We found that sigmoid outperformed both tanh and ReLU. However, we still think the best activation function is ReLU as it requires simpler calculations (due to the fact that it doesn't need to compute complex derivatives at each iteration) and therefore achieves a good accuracy with a shorter training time. Considering that the difference in accuracy is only around one percent is it not worth it to do all the complex calculations required by sigmoid when a simpler approach leads to similar results.

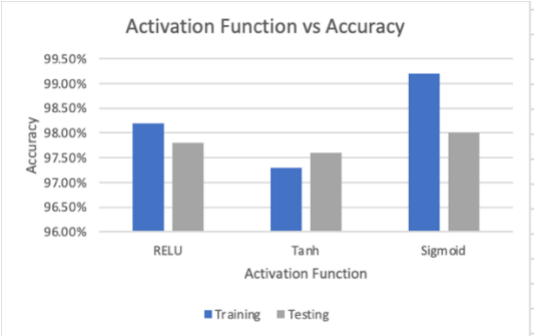

Figure 3: *Activation Functions vs Accuracy*

Effect of regularization on the accuracy

Regularization is a technique that helps reduce overfitting or reduce variance in our network by penalizing for complexity. Certain complexities in our model may make our model unlikely to generalize well even though it fits the training data. Therefore, when adding regularization, we are trading in some of the ability of our model to fit the training data well for the ability to have the model generalize better to unseen data. Therefore, we added a term to the loss function that penalizes large weights. L2 regularization consists of adding the sum of the squared norms of the weight matrices multiplied by a small constant. This constant is called the regularization parameter. The objective of SGD is to minimize the loss function. With a high regularization parameter, we are setting the weights close to zero, reducing the impact of some of our layers. This makes the model less complex which may in turn reduce the variance and overfitting. Therefore, a high regularization parameter will penalize larger weights and hence simplify the model. In our case, the model is not too complex and is not really overfitting, so we noticed that we don't need regularization at all. This is why we set the regulation parameter to 0.001 so that the model still achieves a performance as good as the one it was achieving before adding the L2

regularization. Anything higher than that caused the model to perform worse than before. Perhaps if both the depth and the width of the model were further increased, the numbers of parameters would become too high and stronger regularization would be needed. However, in our case, adjusting the learning rate was enough to produce a good classification of the digits. In the following figure we can find the performance of the model when setting L2 = 0.0001.

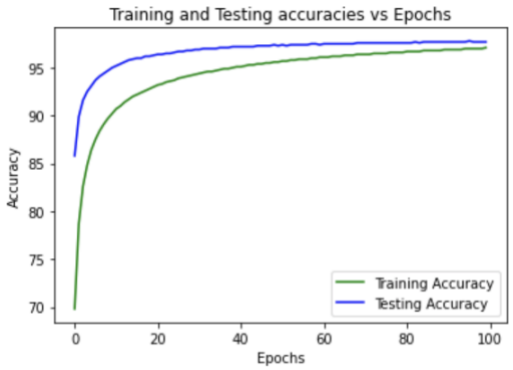

Figure 4: *Performance after adding L2 regularization*

By setting L2 to something higher we saw that both the training and testing error increased, which is a sign of over-regularization: the model is penalized so severely that it now underfits the data, meaning that it cannot explain much of the data variability via the predictors.

Effect of normalization on the accuracy [3]

The idea behind normalizing the data is that we want the input of our model to exist in a smaller region of space where certain input points are not dominated by others. The unnormalized input data might be spread and contain large values that are far apart from the rest of the points. By normalizing it, we shrink it down to a smaller space so that the gradient descent doesn't get stuck in local minima far away from the global minima. For instance, if we don't normalize the data, the sigmoid activation function might converge to the large values and hence lead to bad classification of the data.

As expected, and as shown in the figure below, experiments with normalized data performed better than the ones with unnormalized data.

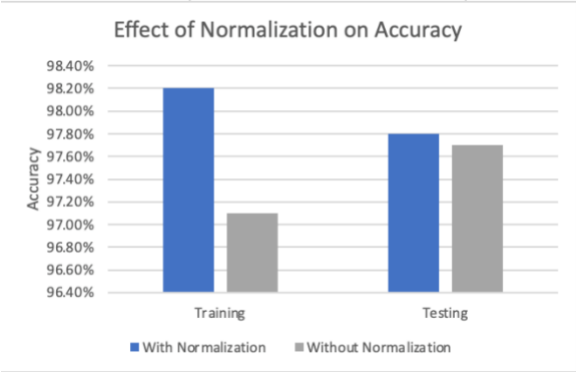

Figure 5: *Normalized, Unnormalized vs Accuracy*

Further improvements

- **Variable Learning Rate**

To see if our network could be further improved, we implemented Variable Learning Rate - halving the learning rate every N iterations.

The hyperparameters "variable_duration" and "min_lr" were added with default values 10 and 0.001 respectively. This meant that every 10 iterations, the learning rate would get halved until reaching the minimum learning rate of 0.001.

This implementation was proved too simplistic and unhelpful as no significant improvement was observed. We suspect this was because the learning rate adjustments are not responsive: by the time the optimal value for the learning rate becomes too high, the learning rate has already been lowered to a very low value. Therefore, it seems that initially tuning the learning rate to its correct value is more important than adjusting the learning rate during training. This is consistent with the current consensus that "the learning rate is perhaps the most important hyperparameter. If you have time to tune only one hyperparameter, tune the learning rate" **[4]**

In the future, we suspect that a momentum based variable learning rate might yield accuracy gains.

- **Xavier initialization**

We then decided to test whether a different technique for weight initialization could help solve the vanishing gradient problem. Random initialization can cause issues and instabilities in the training process. Therefore, we decided to implement Xavier's initialization to see if better results could be achieved. Following Glorot and Bengio's research, "we initialized the biases to be 0 and the weights Wij at each layer with the following commonly used heuristic:

$$Wij \sim U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$$

where U[-a, a] is the uniform distribution in the interval (-a, a) and n is the size of the previous layer" **[5]**.

The idea behind this initialization is to try to make the variance of the outputs of a layer equal to the variance of its inputs. By doing so, it reduces the vanishing and exploding gradient problem affecting both sigmoid and tanh.

Discussions & Conclusion

In conclusion, we found that Multilayer Perceptron is an effective technique to classify data from images. Although the architecture of the model is important for its accuracy, many other factors need to be taken into account when designing it. The depth and width of the model need to be adjusted accordingly to the complexity of the problem. Further, depending on what our goal is, many activation functions need to be considered and their respective learning rates must be tuned. We found that, due to its simplicity and efficiency, ReLU seems to be the most prominent activation function. Furthermore, L2 regularization needs to be taken into account when dealing with complex architecture and for preventing overfitting. In such cases, it can help improve the accuracy of our model by simplifying it. However, it might not be needed for models that are already performing well. We also discovered the importance of normalization for the ability of the gradient descent to find local minima that are close to the global minima. Finally, we showed that different weight initialization techniques can help reduce the vanishing gradient problem.

For future work, as L2 regularization did not seem to provide much for this problem, we could explore whether other regularization techniques could improve our accuracy. We could also consider testing different gradient descent techniques to see if a shorter training time can be achieved.

Contributions

Ahmad worked on the code while Malena and Nadia worked on the collection of data from experiments, the analysis and the report.

Sources:

[1]: Fionn Murtagh, Multilayer perceptron for classification and regression, Neurocomputing, Volume 2, Issues 5–6, 1991, Pages 183-197, ISSN 0925-2312, https://doi.org/10.1016/09252312(91)90023-5.

[2]: J. Sola and J. Sevilla, "Importance of input data normalization for the application of neural networks to complex industrial problems," in *IEEE Transactions on Nuclear Science*, vol. 44, no. 3, pp. 1464-1468, June 1997, doi: 10.1109/23.589532.

[3]: **https://stats.stackexchange.com/questions/185853/why-do-we-need-to-normalize-the-images-before-we-put-them-into-cnn**

[4]: Ian Goodfellow, Yoshua Bengio and Aaron Courville, "Deep Learning", (2016), page 429.

[5] : Xavier Glorot and Yoshua Bengio, Understanding the difficulty of training deep feedforward neural networks, DIRO, Universite de Montreal, Montreal, Quebec, Canada.