Experimental Performance Evaluation of Multi-Layer Perceptron Models on the MNIST Data Set

Ferdinand Lösch (17058320)
Module U08089: Machine Learning
DATE:25st November 2019

I. Introduction

A. What MNIST?

The MNIST data set of handwritten Digits consists of 60,000 training images and 10,000 testing images. With each image having a resolution of 28 x 28 pixels, which results in a total vector size of 784 per copy. Additionally, the images are in black-and-white so Grace Guild presented as a single-channel of 8 bits 0 to 255 additionally the values are normalised to $\frac{x}{255}$
. the dataset was put together by (Yann LeCun, Courant Institute, NYU), (Corinna Cortes, Google Labs, New York), (Christopher J.C. Burges, Microsoft Research, Redmond). currently, MNIST is commonly used as a benchmark to evaluate the performance of image classification algorithms. In terms of the expected performance of MLPs, the website lists that are two-layer MLP with 300 notes in the hidden layer should achieve anywhere from 4.7% to 0.7% Error for a highly optimised two two-layer MLP using cross-entropy as the loss function.

Fig. 1. Snapshot of MNIST dataset.

gr

B. MNIST vs Iris

MNIST is much more challenging in classification as it contains 784 dimensions compared to the relatively shallow 4 dimensions of the iris dataset. Additionally, MNIST contains ten classes, compared to the three classes in the Iris dataset[3]. This makes it much more difficult to visually separate the classes. For example, in figure [2], we can see MNIST visualised two three- dimensions using PAC(Principal Component Analysis)[4] dimensionality reduction. However, it is still challenging to see the separations of the classes you can make out a couple of somewhat clear distinctions, but in general, it is tough to spot those separations.

Fig. 2. MNIST dataset Visualised with PAC dimensionality reduction.

When using the same method of PAC dimensionality reduction on the iris dataset seen in figure [3], it becomes much more natural and clearer to see the distinct separations of the classes.

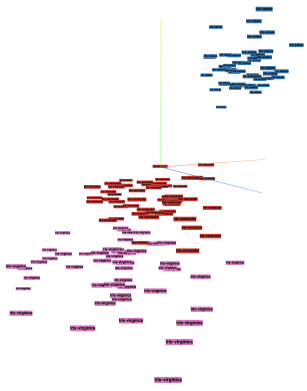


Fig. 3. Iris dataset Visualised with PAC dimensionality reduction.

Another critical factor which makes MNIST more challeng-

ing to classify then the Irish dataset is what commonly is referred to as the dimensionality curse[5], which refers to the more dimensions added in space, the smaller the chance that a feature point will be contained in that space.

II. Multi-layer perceptron and Backpropagation

A. Multi-layer perceptron

A multilayer perceptron (MLP) is a type of feedforward artificial neural network. MLPs typically consists of at least three layers of notes. An Input layer which includes the same number of nodes then vectors in the input data, so for the MNIST data set, it will be 784 input nodes. The hidden layers which can consist of any number of notes these nodes are connected to all nodes in the input layer/previous layer and subsequent layer; this is also sometimes referred to as the dense layers. Furthermore, an output layer which contains the same number of nodes then classes to be classified, Figure [4] shows this principle.

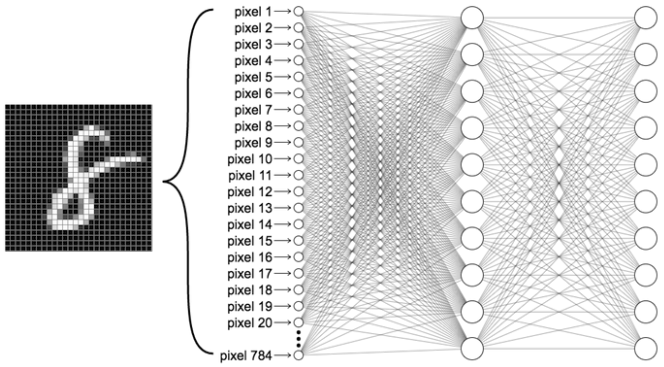


Fig. 4. MLP illustration

MLPs make use of an activation function in the perceptron. The activation function determines given a weight, input and bias if the perceptron should fire. The feedforward process can be described as follows.

$$a_i^{(k)} = \sum_{j=0}^{N^{(k-1)+1}} w_{ij}^{(k)}.v_j^{(k-1)}$$

B. Backpropagation

Backpropagation, short for "backward propagation of errors," is a commonly used algorithm for supervised learning for MLPs or other neural networks. The algorithm computes the gradient of the error function concerning the neural network's weights. Backprop was discovered in the1970s[6] as a general optimiser for automatic differentiation of complex nested functions, but it wasn't until 1986, with the publishing of an article by Rumelhart, Hinton, and Williams, titled "Learning Representations by Back-Propagating Errors,"[8] However, training the MLP with the gradient descent method requires a calculation of the error which we can do as follows.

$$E = \left\|\vec{y}.\vec{t}\right\|^2$$

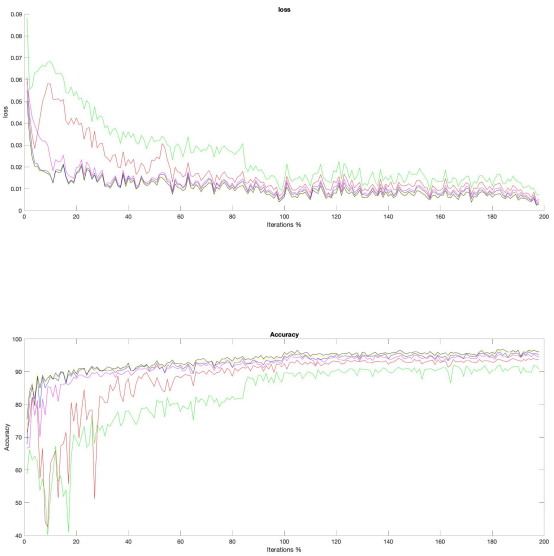
Now that the error is defined we need to compute the gradient and perform the backpropagation by updating the weights which we can do like this.

$$w_{ij}^{(k)} \leftarrow w_{ij}^{(k)} - \eta.\frac{\partial E}{\partial w_{ij}^{(k)}}$$

III. MLP Performance Evaluation

A. Finding optimal learning rate

Funding the optimal learning rate my experiment concludes of an MLP with 300 hidden neurons. I found this number optimal as it gives me approximately 4% Error, and it computes relatively fast on two Epochs as I will need to run this several times to find the minimal Loss in my gradient descent setting the right learning rate is essential as I do not want to overshoot in the gradient descent step.



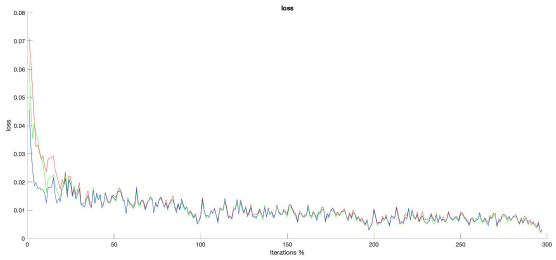
learning rate	colour	Error
0.02	green	9.0%
0.04	red	6.5%
0.06	magenta	5.2%
0.08	blue	4.6%
0.10	yellow	4.0%
0.12	black	4.1%

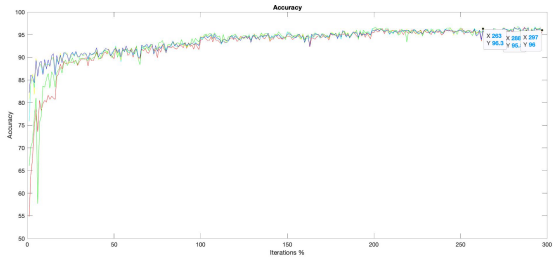
After analysing the data, it becomes clear that $\eta = 0.1$ is a reasonable learning rate as it provides the most exceptional in regards to accuracy and Loss. As well as staying stable without overshooting too much, however when trying larger values, accuracy and Loss are deteriorates as overshooting occurs in the gradient descent step.

B. Finding optimal Number of perceptron in hidden layer

When determining the best number of nodes, usually more is better for the hidden layer. However, eventually adding more neurons will not improve or sometimes cause a negative effect on accuracy. Additionally, when initialising the weights for the hidden layer, I found that when not choosing an appropriate range, the activation function will saturate and will return equal possibilities for all classes. so the variance for the layer is initialised as (variance / 1000) this makes sure that the saturation error does not occur.

Notes in hidden layer	colour	error
100	red	10.2%
200	green	8.2%
500	cyan	5.8%
800	yellow	4.3%
1000	yellow	3.7%





It also helps to train the model for more Epoch in my testing; I found that when training is set to 17 Epoch at 2000 nodes in the hidden layer before the model starts to overfit with my best peak error at 2.32% on the testing set, which is on par with the accuracy described in the papers on the 3-layer NN Specified on the MNIST website, Nevertheless, this did take six hours to train. I achieved this by using a technique of saving only the model with the lowest loss this will ensure that I always see the model at peak performance and avoid the small dips in performance as depicted in the graphs.

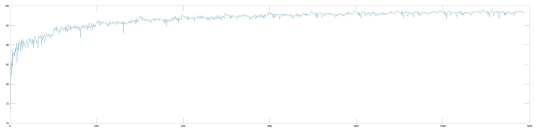


Fig. 5. MLP at 2000 notes in the hidden layer train to 2.32% on testing set.

C. Sigmoid or Softmax?

Sigmoid and SoftMax are activation functions typically used in neural networks, they transform an input and normalise it into probability distributions, which can be used to determine if a perceptron should fire, they can be defined as follows.

$$\sigma(a) = \frac{1}{1+e^{-a}}$$

$$P(C_i|x) = softmax(a)_i = \frac{exp(a_i)}{\sum_j exp(a_j))}$$

When choosing between Sigmoid and Softmax, it is essential to note that they are in fact equivalents in the meaning that one can be transformed into the other. So we can now converted it into a equivalent binary classifier that applies the sigmoid instead of the softmax activation function. First defined two classes $(C_0 or C_1)$. And pick a probability that we want the sigmoid to Output so lets go with C_1

$$a' = w'^t x + b$$

$$p(C_1|x) = \sigma(a') = \frac{1}{1+e^{-a'}}$$

$$P(0|\boldsymbol{x}) = 1 - \sigma(a')$$

So the classifiers are equivalent as long as the probabilities are the same.

$$\sigma(a') = softmax(a_0)$$

activation function	colour	error
Sigmoid	red	10.2%
Softmax	green	8.2%

However, this still does not answer the question on which activation function to use for the MNIST dataset. As the output will be mutually exclusive meaning that it can only be one of the ten digits and not multiple of them, we want to use the softmax activation function on the output layer. As this will enforce that the sum of the probabilities of the output classes is equal to one, so to increase the likelihood of a particular class, the model will correspondingly decrease the possibility of at least one of the other classes. This will lead to significant performance increases over Sigmoid in multiclass classification problems this behaviour can be seen clearly in figure 6 and 7

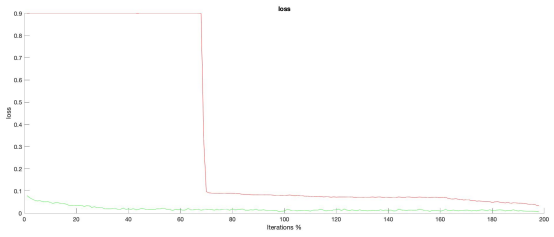


Fig. 6. Sigmoid and softmax loss graph

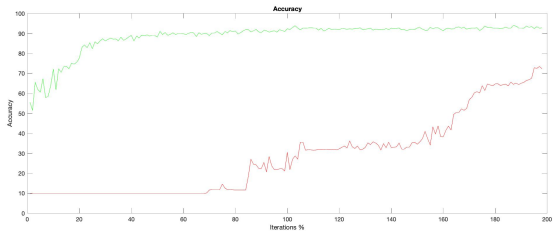


Fig. 7. Sigmoid and softmax accuracy graph

D. Online back prop VS normal mini Batch back prop

The ”normal” Batch backpropagation is the process of using the average gradient in a mini-batch for the backpropagation step. By applying the average gradients, we smoothen out the steps in the backpropagation resulting in a smoother, less shaky learning process as shown in the loss graph figure 8 with a much lower average loss compared to the online back propagation process. However, when considering accuracy, batch-back-Prop is considerably slower in learning than online as well as requiring significantly more samples in achieving the same Error. Nevertheless, it is also about three times as fast in my testing as it does not need to compute the full-back Prop process for every data point.

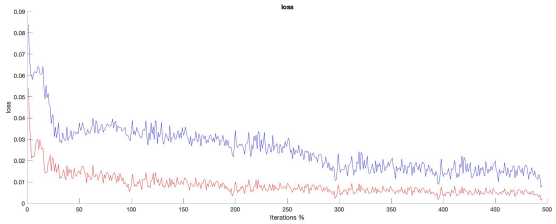


Fig. 8. red = online,Blue = batch learning

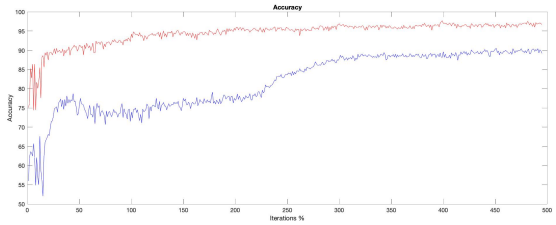


Fig. 9. red = online,Blue = batch learning

E. Going Deeper

Will adding more hidden layers help the accuracy of the MLP. The paper ”Deep Big Simple Neural Nets Excel on Hand- written Digit Recognitio”[1] claims that it is possible to achieve a 0.35% Error on MINST as this is the best accuracy recorded so far on an MLP. As Matlab is inherently slow for large matrix multiplication due to the lack of GPU acceleration for training. So I implemented the Deep multilayer MLP in Python using Keras and Google Colab to get access to cloud GPU training which accelerates the training to about two seconds per Epoch. With inspiration from the paper, I constructed an MLP with the following construction

layer type	number of nodes	activation function
input	784	sigmoid
hidden	1500	sigmoid
hidden	1000	sigmoid
hidden	500	sigmoid
Output	10	softmax

In the first training round I used a batch size of 50 and the Adadelta optimizer ”Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients. This way, Adadelta continues learning even when many updates have been done.”[9]

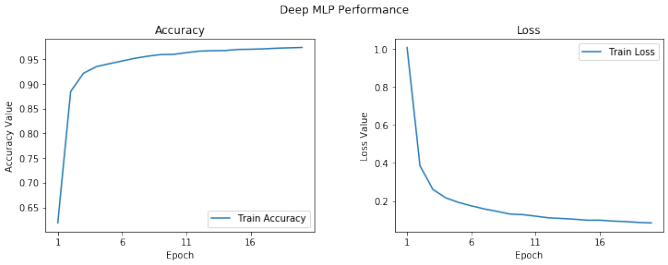


Fig. 10.

With the second round of training, I applied transfer learning on the previous weights, and I managed to achieve 0.49% Error on training data and 1.2% Error on the testing set.

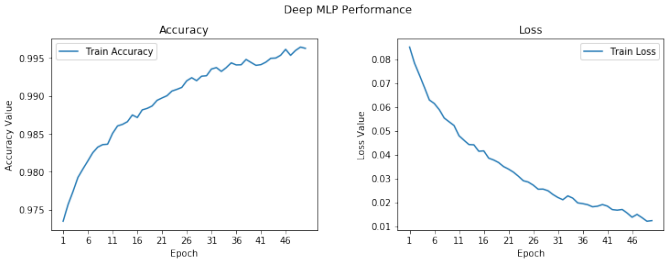


Fig. 11. Transfer learning

F. So can we do better with Convolutional Neural Networks (CNN)'s?

Convolutional Neural Networks are currently the state of the arc for image-based classification. It derives its name from the type of hidden layer used in the network typically CNN's are constructed out of convolution layers and Pooling layers for dimensionality reduction.

This CNN was built with three Convolutional layers sandwiched in between pooling layers for dimensionality reduction, followed by fully connected layers for the classification. In terms of tools used, I created it using Python, Keras and

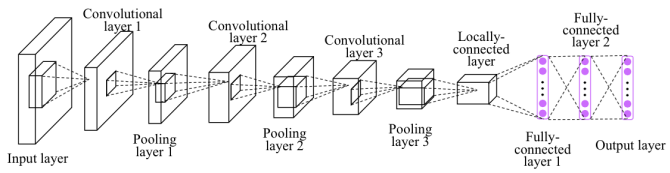


Fig. 12. Diagram of the CNNs construction

Google Colab for the GPU acceleration. The model construction is depicted in figure [12] — Relu was used for the activation function and softmax for the output layer. In terms of the optimiser, I again use Adadelta[9] and ReduceLROnPlateau[7] to optimise the model's performance.

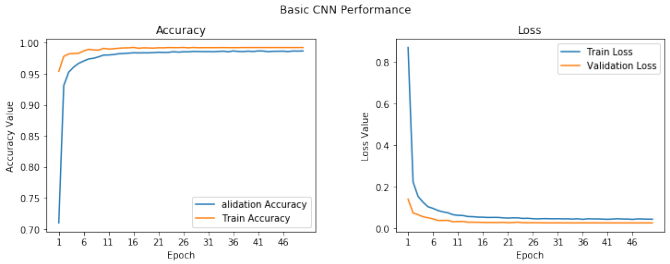


Fig. 13. CNN training graphs

After training on 50 Epoch and several transfer learning iterations, I achieved a 0.51% error on the testing set and a 0.41% error on the validation set, which is considered an excellent performance compared to the simple construction of this basic CNN. Nevertheless, the current record is 0.23% error on the MNIST data set which used a hierarchical system of convolutional neural networks[2].

IV. Conclusion:

In conclusion, the experiment showed that MLPs are exceptionally capable in handling classification of the MNIST dataset. With the peak performance of my MLP reaching 2.32% error after several prior optimisation experiments. The learning rate of $0.1\eta = 0.1$ proved to be ideal even with more neurons in later experiments. This shows that some- hypa parameters can be considered stable as long as the model architecture does not change too drastically. Additionally, the number of neurons in the hidden layer can make a massive difference in terms of performance of the network. In my testing, it shows that accuracy increases with the number of neurons with my best results at 1000 in the hidden layer; however, the computing cost increases quadratically to the linear increase of the size of the matrix when adding more neurons, this can be contributed to the matrix multiplication which has a quadratic cost. Furthermore, I achieved a significant performance boost when utilising softMax rather than sigmoid for the output layer due to its inherent nature of distributing the probabilities between one and zero giving the most likely class a boost in probability.

When comparing my two-layer Matlab MLP to other algorithms, including my deep MLP and my CNN in terms of performance and processing speed, when taking the peak error at 2.32% it does not fall that far behind when compared to an MLP with four hidden layers with the difference only being about 1,12%, in addition, it should be noted that the deep MLP also used a much more advanced backpropagation function Adadelta [9] compared to standard stochastic gradient descent. Furthermore, its performance is comparable to the networks listed on the MNIST website. It is even outperforming most of the networks listed in its class which I consider a success. However, when comparing the accuracy to my CNN, my MLP is hopelessly inferior with the CNN topping out at just zero point 0.51% error which represents a 4.5 X improvement over the MLP. additionally, my CNN outperforms all CNN listed on the website except Convolutional net, cross-entropy [elastic distortions] which takes the lead by about 0.02%.

In terms of improvements, some of the significant elements to improve on my MLP. could be the gradient descent optimiser Adadelta Or similar optimiser could further yield a 2 X improvement in performance. As my MLP currently is limited by stochastic gradient descent which is prone to be stuck in local minimums.

Additionally, there is more room for improvement in the batch propagation function to yield further speedups in training. As in my experiment, training time was a major limiting factor hindering me of trying more experiments with even more hidden neurons or epoch.

Furthermore, hidden layers could be added to the MLP. This would additionally open up the possibility of experiments of different layer sizes.

References

[1] Dan Claudiu Ciresan et al. "Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition". In: *CoRR* abs/1003.0358 (2010). arXiv: 1003.0358. URL: http://arxiv.org/abs/1003.0358.

[2] Dan Ciresan, Ueli Meier, and Jürgen Schmidhuber. "Multi-column deep neural networks for image classification". In: *IN PROCEEDINGS OF THE 25TH IEEE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION (CVPR 2012*. 2012, pp. 3642–3649.

[3] R. A. Fisher. *The use of multiple measurements in taxonomic problems*. 1936.

[4] J.E Jackson. *Principal component analysis*. URL: https://en.wikipedia.org/wiki/Principal_component_analysis.

[5] Eamonn Keogh and Abdullah Mueen. "Curse of Dimensionality". In: *Encyclopedia of Machine Learning and Data Mining*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2017, pp. 314–315. ISBN: 978-1-4899-7687-1. DOI: 10.1007/978-1-4899-7687-1_192. URL: https://doi.org/10.1007/978-1-4899-7687-1_192.

[6] Seppo Linnainmaa. *reverse mode of automatic differentiation*. 1970.

[7] *ReduceLROnPlateau*. URL: https://keras.io/callbacks/#reducelronplateau.

[8] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning Representations by Back-propagating Errors". In: *Nature* 323.6088 (1986), pp. 533–536. DOI: 10.1038/323533a0. URL: http://www.nature.com/articles/323533a0.

[9] Matthew D. Zeiler. "ADADELTA: An Adaptive Learning Rate Method". In: *CoRR* abs/1212.5701 (2012). arXiv: 1212.5701. URL: http://arxiv.org/abs/1212.5701.

CONTENTS