Adaptive Intelligence Assignment 1: Supervised Learning of Handwritten letters

Luca Del Basso
dept. of Computer Science
The University of Sheffield
Sheffield, United Kingdom
ldelbasso1@sheffield.ac.uk

***Abstract*—This report focuses on the implementation of a multilayer perceptron under supervised learning conditions in order to classify handwritten letters from the EMNIST dataset [1].**

I. Introduction

In supervised learning a system is created, known as a model. The model is created through the training process. The goal of training is to develop an accurate model which can correctly answer classification questions related to its input most of the time. The process of training begins with a labelled data set of items you want to classify. Each item in the set, will contain information about factors that might be used to distinguish them from each other, known as features. Each item in the set is also labelled, meaning the correct class for each item in the set is known. The training data is used to incrementally improve the model's ability to predict the class of data that is not labelled. The model will contain parameters that are incrementally updated so that the total error evaluated using the training set is minimised. This evaluation is often done by splitting the training set into a smaller evaluation set which compares the model's output with the expected output. Training the model uses forward propagation and backwards propagation. By propagating forwards, the model applys the activation functions in addition to the weights and bias to produce an output, this output is used to calculate an error signal. Typically the difference between the actual output and the desired output. This signal is then used in the backpropagation in which the change in error with respect to the initial inputs is calculated and as a result the weights and bias are updated in such a way to minimise error on the next forward propagation.

II. Methods

A. Multilayer Perceptron

I implemented a multilayer perceptron (MLP) classifier that updates its parameters in mini-batches such that the gradient was updated multiple times per epoch, or training step. The MLP receives a flat array as an input, this represents the intensity of every pixel in a 28*28 image of a hand drawn letter, and it has been normalized such that each intensity value is between 0 and 1. The input is propagated forward through the network of perceptrons, where the output for each layer of nodes is the sum of the product of the weights and inputs in addition to some bias.

$$h_i^{(1)} = \sum_{i=1}^{784} w_{ij}^{(1)} x_j^{(0)} + b_i^{(1)}$$

The output is then passed through an activation function, in this case, ReLU.

$$f(h) = max(0.0, h)$$

If other layers exist, the process is repeated with the output of the previous layer passing through subsequent layers with the maths just described (but with different weights and bias values) until the output layer is reached and the output with the highest value is considered the class predicted by the model. after each mini-batch the weights and bias are updated through back propagation, where the change in weights is given by

$$\Delta w_{ij} = \eta(t_j - y_j) f'(h_j) x_i$$

where η is the non-zero learning rate, t_j is the target output, y_j is the actual output, h_j is the weighted sum of the neuron's input(as seen before), x_i is the i^{th} input and $f'(h)$ is the derivative of the ReLU activation function given below as:

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

Technically ReLU is not differentiable at $x = 0$ so common practice is to substitute it for a value such as a 0, 0.5 or in this case 1.

ReLU is a popular activation function to use in hidden layers as it is cheap to compute compared to others such as the sigmoid function. In practice ReLU also seems to show better convergance performance than sigmoid[4]. Although ReLU has some drawbacks, it can suffer from the "Dying ReLU problem" in which too many activations go below zero and will continuously output zero. Leaky-ReLU is a feasible solution to this where the max function is not comparing with zero but a small number.

It is important to normalize your input data, typically around a mean of zero and standard deviation of 1. This is because when using activation functions such as sigmoid, extremely high values will only increase or decrease by a small amount due to the gradient of this function. As the network propagates

forwards the variance of the outputs for nodes has the chance to grow significantly, due to the variance of the sum of random numbers is the variance of each of those numbers.

By shrinking the initial variance of the input data, by normalizing it and by initializing the weights to small uniformally distributed values, later variances have a better chance of not becoming too large and as a result slowing down convergance times.

This model uses mini-batch updates, meaning a random selection of the training data is used during an epoch to train the model. Full batch testing is when the entire training set is used every epoch to train the model. Finally online updating is updating the model after every individual example in the training set.

III. Results and Discussion

A. Linear model - baseline

Where

$$MSE = \frac{1}{2B}\sum_{\mu}^{B}\sum_{i}(t_{i\mu} - x_{i\mu}^{out})^2$$

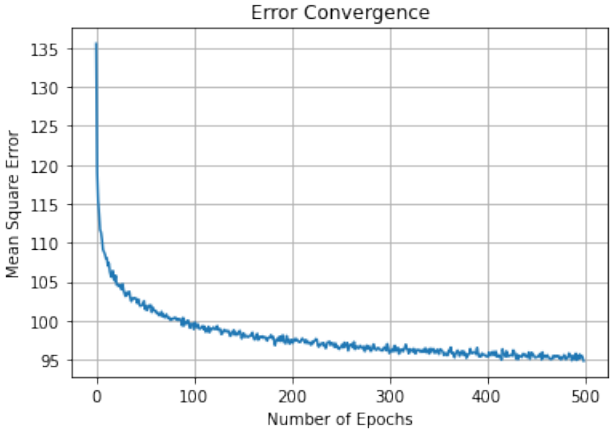

Fig. 1. Parameters: η : 0.1 , batch size: 50, number of epochs: 500

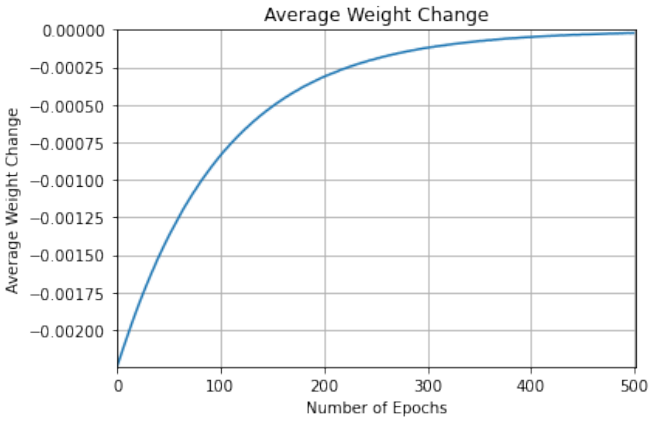

Fig. 2. Weight convergence, Xavier initialisation

IV. Classification Performance

With a single layer perceptron, averaged across five runs, I achieved a final classification accuracy of 54.6% on the test data. According to the EMNIST paper [1], the authors achieved an accuracy of 55.78%

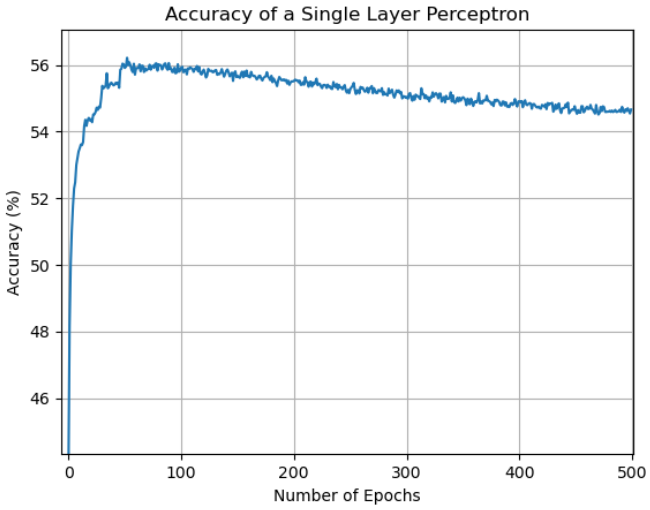

Fig. 3. SLP average accuracy

After approximately 50 epochs, the model starts to overfit the training data as shown by the gradual decline in accuracy with a peak of approximately 56%.

V. L_1 Regularisation Penalty

The regularisation penalty is given by

$$E_{L1} = \lambda_1 \sum_{k}^{N_{layers}} \sum_{ij} |w_{ij}^{(k)}|$$

Its derivative can be calculated by substituting $|w|$ for $\sqrt{w^2}$ and by using the chain rule

$$\frac{dy}{dw} = \frac{dy}{du} \times \frac{du}{dw}$$

$$u = w^2$$

$$y = |w| = \sqrt{w^2} = (w^2)^{\frac{1}{2}} = u^{\frac{1}{2}}$$

$$\frac{dy}{du} = \frac{1}{2} \times u^{-\frac{1}{2}} = \frac{1}{2} \times \frac{1}{\sqrt{u}} = \frac{1}{2\sqrt{u}}$$

$$\frac{du}{dw} = 2w$$

$$\frac{dy}{dw} = \frac{1}{2\sqrt{u}} \times 2w = \frac{2w}{2\sqrt{w^2}} = \frac{w}{|w|}$$

$$\therefore \frac{\delta E_{L1}}{\delta w_{ij}^{(k)}} = \lambda_1 \frac{w_{ij}^{(k)}}{|w_{ij}^{(k)}|}$$

A. Lambda Values

When $\lambda = 0$ the model becomes a standard MSE model. Each was trained as described in the spec. Below is the final validation error for each λ.

λ value	Error (%)
0	25.35
$1e^{-8}$	25.78
$3e^{-8}$	24.93
$1e^{-7}$	25.72
$3e^{-7}$	26.89
$1e^{-6}$	25.49
$3e^{-6}$	26.45

λ value	Error (%)
$1e^{-5}$	28.44
$3e^{-5}$	26.78
$1e^{-4}$	25.83
$3e^{-4}$	25.17
$1e^{-3}$	31.31
$3e^{-3}$	24.70

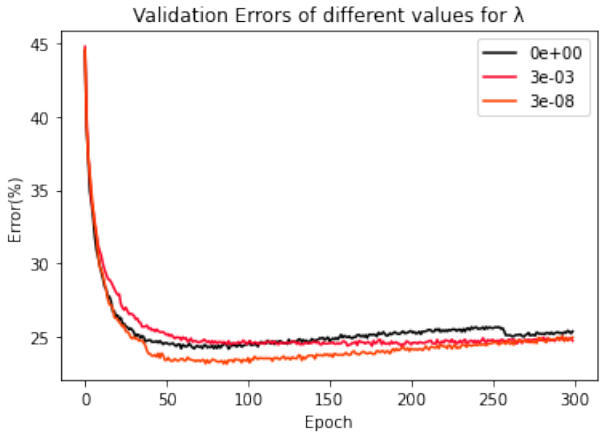

Fig. 4. Average Validation Error for $\lambda = 0, 3e^{-3}, 3e^{-8}$

Plotting the average validation error for the two lowest errors of λ and $\lambda = 0$ as a reference. $\lambda = 3e^{-8}$ consistently provided lower errors until the final few epochs. Whilst a low error is good, the graph shows that this value of λ is still over fitting the data shown by its steady increase in error. Whereas the error for $\lambda = 3e^{-3}$ stays consistently flat from around 75 on wards, showing that this value for lambda is preventing the model from over fitting. As a result, all further experiments will use this value of λ.

B. Other over fitting prevention methods

Reducing the number of dimensions of the parameter space or reducing the size of each dimension are the two main ways to reduce over fitting [2]. Pruning is a method in which nodes are removed from layers if they are deemed to not be contributing much to the performance of the model. A method for selecting nodes for removal is to remove nodes on either side of a node who's weight change is close to or zero. This can also help with "dead RelU", where nodes in a ReLU based model can be fixed in outputting zero. Other methods to prevent overfitting includes "dropout" layers, where nodes are activated and deactivated during training, this helps prevent the network from overfitting because the network does not over rely on paths through the network.

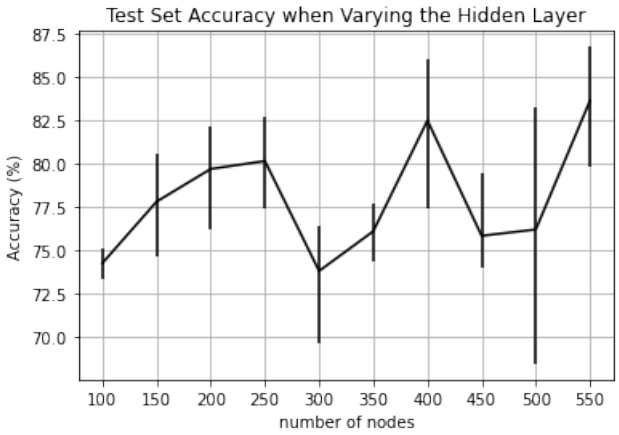

Fig. 5. incrementally increasing the nodes in the hidden layer from 100 to 550. Averaged over 3 runs for each.

VI. Varying the hidden layer

The data shows a general upwards trend in accuracy on the test set as the number of nodes in the hidden layer increases. However, in some cases the accuracy dips even though the number of nodes is increased.

The results found in the EMNIST paper also show an increase in accuracy when increasing the nodes in the hidden layer, although the increment of nodes between models is much greater than the increment in mine. All models performed better than the single layer perceptron. The reason for the improved performance over the linear perceptron is because this classification problem is not linearly separable, meaning the classes cannot be completley separated using straight line decision boundaries. The use of a hidden layer allows for the separation of data patterns with multiple lines and a third layer can create decision boundaries of any shape to separate the data.

My data, in conjunction with the EMNIST paper, suggests that increasing the number of nodes in a single hidden layer will improve the overall accuracy of the model,by increasing the number of nodes in the hidden layer you are increasing the capacity of the model to differentiate between patterns of the input to the output. Although, adding too many nodes can result in overfitting.

VII. Varying the Second hidden layer

The data shows a general upwards trend that slowly decreases again after around 350 nodes in the second hidden layer. Some configurations of this model performed better than the previous. For example, 300 nodes in the second hidden layer performed better than the previous configuration of a single hidden layer with the same amount. Whilst the first model records some accuracy scores exceeding 82.5%, the second model does not exceed this threshold. Whilst the second model exhibits no large dips in performance (that is of 5% or more such as in the first model) when increasing nodes, it does appear to have a larger variance. For example the error

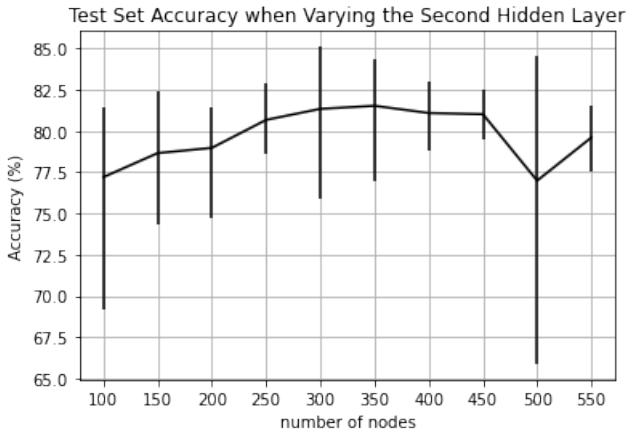

Fig. 6. incrementally increasing the nodes in the second hidden layer from 100 to 550. Averaged over 3 runs for each.

bars for 500 nodes in the second model appear to be quite extreme. This would be considered an anomaly and could be due to only running each experiment 3 times, a choice made due to time constraints in running each experiment.

VIII. Further Research

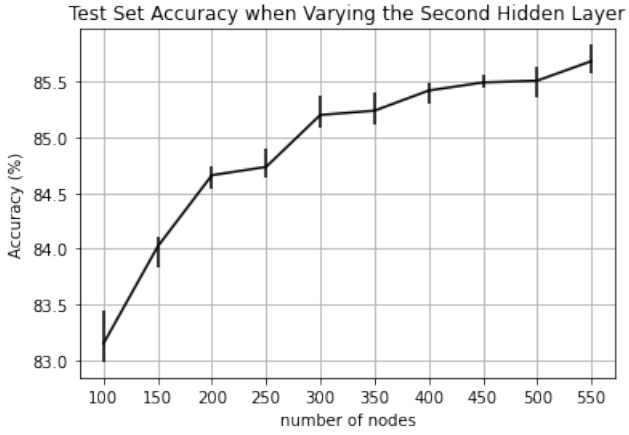

Fig. 7. incrementally increasing the nodes in the second hidden layer from 100 to 550. Averaged over 3 runs for each. Using SoftMax function on the output.

From my findings in the previous two experiments, I wanted to try and improve the accuracy and consistency of my model. The softmax function can be used on the output of the network, in order to turn the outputs into a probabilisitc distribution that sums to one[3]. The resulting selected class is the one with the highest probability. In this experiment I also reduced the number of epochs down from 250 to 125 as i saw previous models converge in 150 epochs and I was pushed for time.

the softmax function is given by:

$$s(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_i}}$$

The results of this test suggest that the dips in accuracy for some increased amount of neurons, as seen previously, was a result of using ReLU on the output of the last layer. Although more research would need to be done. Secondly, using the softmax function on the output layer seems to reduce the variance of the accuracy of the model, with the error bars being significantly reduced in this case. I hypothesise that the improved performace in this model came from the properties of softmax. Softmax normalizes your data, it is differentiable and it gives a minimal amount of probability to all elements in the output vector.

I chose to implement the softmax function, as oppose to the sigmoid function, on the output layer because the results of the model are mutally exclusive. That is to say, an output is either an ”a” or a ”b”, it cannot be both. The probabilistic propert of softmax means the most likely class's probability will increase and the others will decrease.

References

[1]]G.Cohen, S.Afshar, J.Tapson, and A.vanSchaik, “EMNIST: an extension of MNIST to hand written letters.”Retrieved from:http://arxiv.org/abs/1702.05373,(2017).

[2] [2]L.Prechelt , Early Stopping—But When?, pp.53–67. Berlin, Heidelberg: Springer Berlin Heidelberg ,2012

[3] Bendersky, E., 2016. The Softmax function and its derivative - Eli Bendersky's website. [online] Eli.thegreenplace.net. Available at: ¡https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/¿ [Accessed 15 March 2021].

[4] Krizhevsky, A., n.d. [online] Cs.toronto.edu. Available at: ¡http://www.cs.toronto.edu/ fritz/absps/imagenet.pdf¿ [Accessed 15 March 2021].