Adaptive Intelligence : Assignment 1

Maxime Fontana
University of Sheffield
Sheffield, United Kingdon
mfontana1@sheffield.ac.uk

***Abstract*—This report provides an implementation of the Machine Learning algorithms described in the assignment as well as a discussion of the results. This document will base its study on the EMNIST dataset in order to provide supervised-learning based algorithms on handwritten letters.**

I. Introduction

When it comes to supervised learning, we aim at training a system based on the error with regard to already-provided solutions, or labels. In order to build a classifier, there are several mathematical models that can help us predict which label a particular element belongs to, here, we will focus on one of the most performant of all, the Neural Network.

Indeed, a Neural Network consists in successive layers made of a number of small mathematical units, or neurons. The process of training this model consists in two operations, feedforward and backpropagation, that are recursively going to be computed until an error, computed from an error function, is considered being sufficient.

Once the neural network's architecture has been defined and the initial weights initialised (see later), the training can start. We will therefore refer to the figure below in order to explain the process.

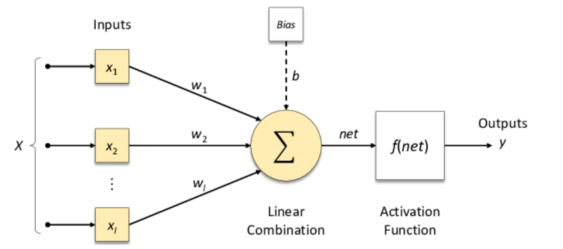


Fig. 1. Perceptron [1]

The initial input layer is going to represent the amount of features that we have to work with, it will also represent the weights. Therefore, the weights matrix can be modelled as :

$$W_{output,input}$$

Then, these input features are going to get multiplied together and be added a bias. This process is known as Neural Activation.

$$A_{output,input} = W_{output,input} \cdot \vec{x}_{input} + \vec{b}_{output}$$

Subsequently, this activated matrix is going to be fed into an activation function, this function serves the purpose of breaking linearity in the chaining process, this allows to solve more complex problems. The output matrix is then computed, which we will refer to as matrix $\boldsymbol{O}$. We now move on to the process of tweaking the neurons by making them learn, this process is known as Back-Propagation.

A. Back-Propagation

Once the computation process has arrived down to the output layer, the information is going to be backtracked. This process aims at adjusting the weights and biases to make the model fit to unknown data. Therefore, an error signal is gonna be computed which is simply the difference between the true label and the output of the model :

$$\vec{error}_{input} = \vec{t}_{input} - \vec{x}_{input}$$

Then, this error is going to fit into an update rule, for example, the delta rule which is going to update the weights accordingly using accumulated gradients across batches (how many times the back and forth process is going to be done before updating these gradients) :

$$W_{output,input} = \eta * \frac{\sum(\vec{error} * f'(\boldsymbol{O}) \bigotimes \vec{x}_{input})}{N}$$

In the equation above, $\boldsymbol{\eta}$ is the learning rate, $\boldsymbol{N}$ is the number of samples in the batch.

Thus, the weights will be sent back to the initial layer, and another cycle of training can be renewed.

In the examples above, a single layer perceptron has been taken as example, note that this training algorithm can be easily extended to deeper models as this whole principle relies on the chain rule.

B. Ending Criteria

Also, one important question would be to what extent this training could be done. It is important not to rely only on the error with regard to the training set, generally computed from the Mean Squared Error metric, but to consider how your model will behave once unknown data is being given as input. Not to underfit, which would be the result of poor training, and not to overfit which would be the result of too much training on the training set, and therefore poor generalisation to unknown data. This is why it is good practice to see how well the model is generalising on a validation set.

II. Methods

A. Algorithm Description

The algorithm I have implemented is similar to the perceptron described in the Introduction. Several differences however have been put in place in order to deal with the requirements. First of all, I presented a code similar to the one in the lab, thereby providing a flexible code that allows to add or remove multiple layers within the neural network. I will present here the perceptron algorithm. First of all, I used the Kaiming ('He') weights initialisation technique [7] which slightly differs from the Xavier initialisation technique [8] introduced in the labs. This strategy can be summarised by the following :

$$A_{output,input} = distribRand(W_{output,input})$$

$$W_{output,input} = A * \sqrt{2/neurons_{previousLayer(input)}}$$

As Jason Brownlee explains in his article [4], this initialisation technique seems to fit ReLU the best.
Also I normalised the entire dataset down to a range of 0 to 1, whereas the initial inputs ranges from 0 to 255 (intensities of pixels).

B. The ReLU Activation Function

In the purpose of this work, I implemented the ReLU activation function. Therefore, I provide an implementation of both the ReLU function and it's derivative. Below you can find a plot showing how both of these behave.

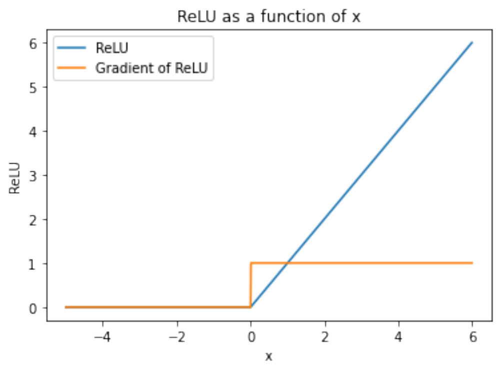


Fig. 2. ReLU and it's Gradient

This new activation function is going to slightly change the way we update our weights, indeed, the error signal is going to be multiplied by the mask provided by the gradient of ReLU introduced above.
Looking at the study on this function made by Jason Brownlee [4], we can clearly see the advantages of the ReLU function with respect to the Tanh and Sigmoid as ReLU is easy and fast to compute compared to the two previously mentioned as they require exponential calculations, this property clearly highlights the preference towards ReLU for deep networks. Also, it is known that the ReLU activation function does not saturate for positive values as it's limit tends towards $+\infty$, in other words, the model never converges to a good solution as the deepest layers remain unchanged as weights get too small once there, which strongly damages the learning capacity of the model.
However, it has been brought to fore that the ReLU has some drawbacks. For instance, in Aurélien Geron's book [3] it is stated that the ReLU activation function suffers from the 'dying ReLUs' problem which apparently occurs when the weights are tweaked in such a way that it always output 0 and drastically disables the learning. Please, refer to the appendix Figure 7 to see the code implementation of ReLU and it's gradient.

C. Different Strategies for Learning

There are multiple ways as to when to update the weights and then back-propagate them into the network. For instance, the Online Updating Rule updates weights after each sample, this is typically used when it nos computationally possible to iterate over the dataset, on the other hand, batch learning goes through the entire dataset before updating the weights and leads to a more accurate change in weights. However, it is possible to combine both strategies into mini-batch learning where the dataset is split into different subsets on which the training is going to be processed and after each the weights will be updated, this provides a reliable way to update weights as it provides accurate gradients.

D. Pre-processing Importance

In the paper from Glorot and Bengio [8], the two scientists introduce a way to deal with stable training and aim at mitigating the mentioned problems, all the initialisation strategies that serve this purpose use the variance, as Glorot and Bengio argue that the output of each layer must be equal to the variance of its inputs, Aurélien Geron comes up with a really good analogy involving a microphone and a chain of amplifiers stating that the voice must come out at the same amplitude as it came in throughout this chain. This is why using the proper initialisation strategy is top priority, it strongly mitigates the way gradients can drift away from a solution.
As for normalisation, various explanations exist, the most important of all is to scale your data into an interval that will be suitable for your activation function.

III. Results and Discussion

In this section, I will present multiple results from my models as well as several changes brought to it in order to assess the capacity of my models as well as improving their training.

A. The Average Weight Update Matrix

One way to see whether our model has converged is to check the changes in weights throughout training, per epoch. Therefore, we can monitor the results of the sum of the elements vs. epochs of an average weight update matrix. We want, by doing that, to see how the gradients behave as the training progresses, it should decrease. In this experiment, the model has been run 500 times (epochs) with a learning rate of $\boldsymbol{\eta}$ = 0.05, and a constant for the average weight update matrix update rule $\boldsymbol{\tau}$ = 0.01.

Figure 3 shows that my model is indeed converging, across multiple runs, we can observe a convergence similar to the one in the plot below.

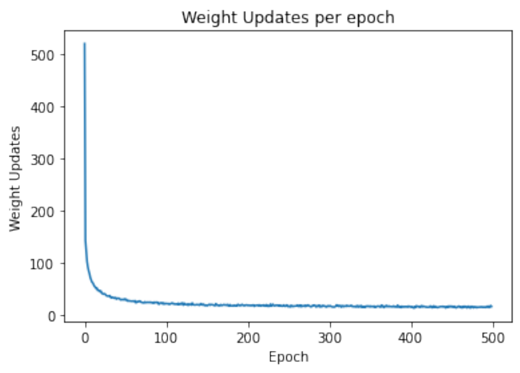


Fig. 3. Average Weight Update Matrix

Please refer to figure 8 in the appendix to see the code implementation for the weight update matrix.
Also, I noted that the learning rate has a huge impact on this curve, indeed, a too big learning rate would make the weights 'oscillate', make it look stochastic, towards the convergence. However, this would still lead to a reasonable accuracy and would require much less iterations. This experiment strongly highlights the trade-off at stake when it comes to training neural networks.

B. Accuracy of Perceptron

Firstly, on the EMNIST dataset [2], we can see that the authors produce an average accuracy of 55.78, this metric will serve as a baseline for our experiments. After several runs, for the single-layer network, or perceptron, the accuracy consistently goes up to 63 on the test set and 64 on the validation set, sometimes it goes up to 68/69 on both, quite unexpectedly.

C. L1 Regularisation

Firstly, we aim at implementing an error that depends on the magnitude of the weights in order to prevent our model from overfitting once training is done. This L1 regularisation penalty that we are going to add to our MSE is going to be suitable for our Multi-Layer Networks later on. Let's derive this term.
Initially, our penalty term is expressed as follows :

$$E_{L1} = \lambda_1 \sum_{k}^{N_{layers}} \sum_{ij} \mid w_{ij} \mid$$

Therefore, we can see above that in order to derive this term, we must derive the absolute function, based on the equality $\mid \boldsymbol{w} \mid = \sqrt{\boldsymbol{w^2}}$.

$$\mid x \mid = \sqrt{x^2}$$

$$\mid x \mid = \frac{2x}{2\sqrt{x^2}}$$

$$\mid x \mid = \frac{x}{\sqrt{x^2}}$$

This derivative allows us to point out 2 different cases. That is, if $\boldsymbol{w_{previous}} > \boldsymbol{0}$, then $\boldsymbol{w_{L_1}} = \boldsymbol{\lambda}$. And similarly, $\boldsymbol{w_{previous}} <= \boldsymbol{0}$, then $\boldsymbol{w_{L_1}} = -\boldsymbol{\lambda}$ and thus, for each single neuron of our layer $\boldsymbol{k}$. Afterwards, this penalty matrix will have to be subtracted to the updated weight matrix so the general weight update rule for this term is :

$$\delta w_{ij,L1}^{(k)} = \eta * (-\frac{\partial E_{L1}}{\partial w_{ij}^{(k)}})$$

I have obviously omitted every MSE-related calculations involved in the overall update process. Please, refer to the appendix to see the code implementation of that added term.

D. Different values of $\boldsymbol{\lambda}$ on Multi-Layer Perceptron

In order to fully assess what the l1 regularisation is doing, we present here results where we vary the strengths $\boldsymbol{\lambda}$, and measure the accuracy achieved on the validation set. For these experiments, the hyper-parameter $\boldsymbol{\eta}$ has been set to 0.05, I have allowed 250 epochs for the algorithm to converge. The results are summarised below.

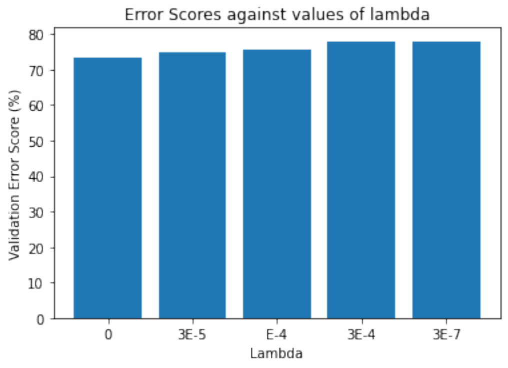


Fig. 4. Variation of the penalty strength

Thereby, we see that the penalty does contribute to the learning and allows a boost in the performance on the validation set, we can see the best value for $\boldsymbol{\lambda}$ is 7*10E-4, and is therefore, the value that is going to be used for the remaining of the experiments. However, the tests have shown that a too large value of $\boldsymbol{\lambda}$ (about the order of E-3), result in poor validation scores, in fact, the model struggles converging and may sometimes result in poor generalisation due to underfitting.
Moreover, there are other counter-measures to overfitting, for instance, Prechelt [5] points out Early Stopping which consists in examining the scores performed on the validation set along with the training to detect when the model is starting to overfit, Prechelt [5] argues that the stopping criteria must be slow at the expense of a much longer training time. Also, Dropout is a common method, introduced by Geoffrey Hinton [6] that

consists in randomly assigning units' weight to 0 for a certain period, this algorithm was proposed as an analogy to the human brain.

E. Number of layers and neurons

One of the main hyper-parameters that we have in a model is the number of units and layers to use, we will see in this section if varying the number of neurons and layers in which they are contained affect the learning in any way.

1) Varying the number of neurons in first hidden layer: Varying the number of neurons on a single-hidden-layer network, we have performed experiments over a range varying from 100 to 550, incrementing the size by 50 at each iteration. At each iteration, we compute the accuracy performed on the test set. A number of 75 epochs has been chosen to be able to notice a trend in the convergence. Also, a learning rate η = 0.05 has been set to make sure the model does not diverge from potential good solutions. The results are depicted below.

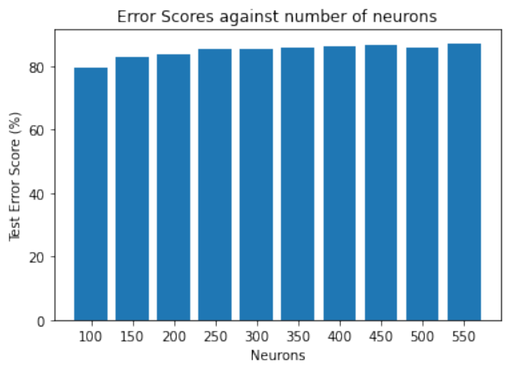


Fig. 5. Variation of the hidden layer size

In Fig. 5, we can clearly see that the accuracy score increases as the number of neurons grows for the range [100, 300] where a jump of 7 % can be observed between the 2 extremities. However, a stagnation of about $\simeq$ 2 % within the range [300, 550]. These results can be compared with the ones in the EMNIST paper [2], indeed, the authors present in their Fig 5 that over a larger range (from 1000 to 10000), a similar trend is measured. In conclusion, increasing the number of neurons in the hidden layer is beneficial to the learning, we have seen that by increasing the size by 500, a boost of up to almost 10 % has been achieved. This is however, at the expense of a longer running time.

2) Varying the number of neurons in second hidden layer: We have seen previously, that a single-hidden-layer perceptron could achieve an accuracy of nearly 85 % with a size of 500. Our goal in this section will be to vary the size of a second hidden layer, with a fixed size of the first one of 100 neurons, in order to derive general rules in relation with the performance and its comparison with a single-hidden-layer perceptron. Note here that the same hyper-parameters as the last section have been used for this experiment. We can observe the results in Figure 6.

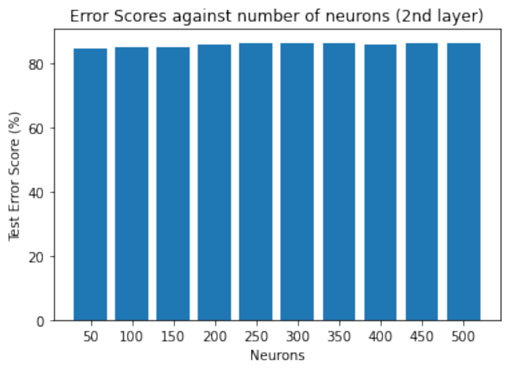


Fig. 6. Variation of the second hidden layer size

Here, we can see that the results for the tested range [50, 500] remain bounded between 85 $\pm 2\%$ which is the performance achieved for a single-hidden-layer perceptron for which the hidden layer has a size between 300 and 550. Therefore, we can say that in this particular case, the performance has not changed. We can draw the conclusion that a network with more smaller layers is not necessarily better than 1 larger layer, in fact, adding layers increase the computational complexity but could solve potentially harder problems, in this task, we can see that a single hidden layer is sufficient as the deeper neural network is not capable of fixing the incorrectly classified samples.

IV. Conclusion

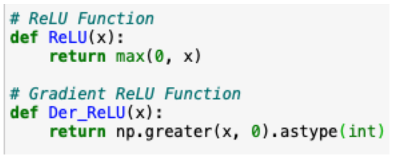
In this paper, we have seen what training a neural network consisted in, what was the mathematical operations behind it, introducing a new activation function at the heart of our network. Also, we have covered different techniques in order to optimise the learning process such as regularisation and overfitting-reduction strategies. Finally, we have presented results on different variations on the number of layers and their sizes on a dataset representing handwritten letters.

References

[1] Parmezan, Antonio Alves de Souza, Vinícius Batista, Gustavo. (2019). Evaluation of statistical and machine learning models for time series prediction: Identifying the state-of-the-art and the best conditions for the use of each model. Information Sciences. 10.1016/j.ins.2019.01.076.
[2] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik, "EMNIST: an extension of MNIST to handwritten letters." Retrieved from: http://arxiv.org/abs/1702. 05373, (2017).
[3] Aurélien Geron, Hands-on Machine Learning with Scikit-Learn, Keras and TensorFlow, 2019
[4] https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks, Jason Brownlee, 2019
[5] L. Prechelt, Early Stopping — But When?, pp. 53–67. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.
[6] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, Ruslan R. Salakhutdinov. Neural and Evolutionary Computing (cs.NE). arXiv:1207.0580.
[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, arXiv:1502.01852 , 2015.
[8] Understanding the difficulty of training deep feedforward neural networks, Xavier Glorot, Yoshua Bengio

Appendix

The code implementations makes use of numpy as all numpy functions work element-wise through the matrix they operate on which makes the use of numpy a strong asset to neural networks.

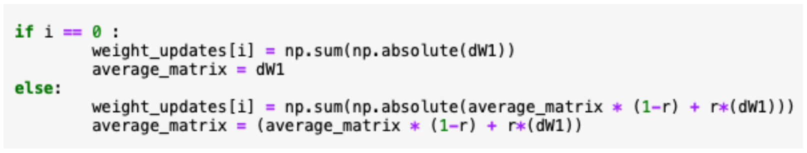


```
# ReLU Function
def ReLU(x):
    return max(0, x)

# Gradient ReLU Function
def Der_ReLU(x):
    return np.greater(x, 0).astype(int)
```

Fig. 7. Code for ReLU and it's derivative

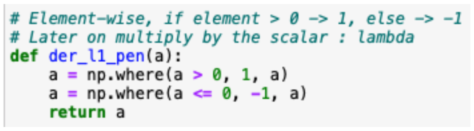
As for the implementation of the Weight Update Matrix, the recursive process was to store current results of the sum of the absolute values of the elements of the average weight update matrix, this is because we are interested in a measure of how much these weights change, as there are positive and negative elements, some elements could compensate for the others and would falsify the results. Thus, we perform these calculations element-wise. Only then, we modify the matrix so that we can make use the current Matrix at iteration $n+1$. Please note that here i refers to the epoch, and this snippet is scaled to a simple perceptron.



```
if i == 0 :
        weight_updates[i] = np.sum(np.absolute(dW1))
        average_matrix = dW1
else:
        weight_updates[i] = np.sum(np.absolute(average_matrix * (1-r) + r*(dW1)))
        average_matrix = (average_matrix * (1-r) + r*(dW1))
```

Fig. 8. Code for the Weight Update Matrix

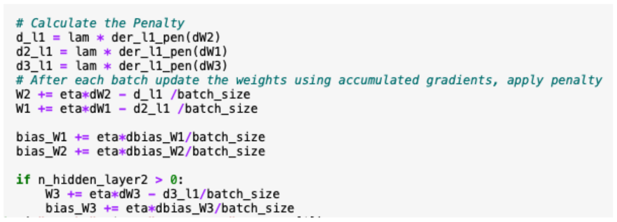
The implementation of the L1 Regularisation Penalty is depicted in the following figures.
First of all, the derivative that is going to be used for the backpropagation process.



```
# Element-wise, if element > 0 -> 1, else -> -1
# Later on multiply by the scalar : lambda
def der_l1_pen(a):
    a = np.where(a > 0, 1, a)
    a = np.where(a <= 0, -1, a)
    return a
```

Fig. 9. Derivative of L1 Regularisation Penalty

Then, once per batch, the penalty is computed and applied to the weight update process.



```
# Calculate the Penalty
d_l1 = lam * der_l1_pen(dW2)
d2_l1 = lam * der_l1_pen(dW1)
d3_l1 = lam * der_l1_pen(dW3)
# After each batch update the weights using accumulated gradients, apply penalty
W2 += eta*dW2 - d_l1 /batch_size
W1 += eta*dW1 - d2_l1 /batch_size

bias_W1 += eta*dbias_W1/batch_size
bias_W2 += eta*dbias_W2/batch_size

if n_hidden_layer2 > 0:
    W3 += eta*dW3 - d3_l1/batch_size
    bias_W3 += eta*dbias_W3/batch_size
```

Fig. 10. Update of L1 Regularisation Penalty