COMP5329 - Assignment 1

ANDREA BOSIA: 520600843
FLORIAN GERON: 520354511
YASHAR TAVASOLI: 510570756

1 INTRODUCTION

In this assignment, we aim to construct a deep neural network in Python using only scientific computing libraries such as NumPy and SciPy. Fully developed packages for neural networks, such as TensorFlow and PyTorch, will not be used over the course of this assignment. This ensures that there is full control over all the elements of the neural network.

We will be working with a dataset given to us by the teaching team. This data set contains 50,000 training examples and 10,000 examples for testing. Each example has 128 features and 1 output label. There are 10 target categories. The exact meaning of this data has not been disclosed to us. Exploratory data analysis has been done to ascertain some basic information on the data set in section 2.

The aim of the study is to manually design a neural network that is able to classify the testing data in one of the 10 output categories. In order to achieve this goal, several different modules have been implemented to improve the accuracy of the neural network. Hyperparameter analyses and ablation studies have been performed to ascertain which modules improve the performance and what parameters should be used.

This study is important because it allows us to obtain a good understanding of (i) how neural networks and their modules are implemented and (ii) how the hyperparameters and modules of a neural network impact its performance. Indeed, when using libraries like TensorFlow, the underlying mathematical operations are hidden by the plug & play style of coding. Mastering these libraries is crucial to efficiently deploy complex neural networks however they make it hard to understand bit by bit the functioning of a neural network.

After this introduction, chapter 2 will deal with the aforementioned exploratory data analysis as well as the preprocessing steps that have been done on the data. In chapter 3, the modules that are implemented in the neural network are discussed theoretically. In chapter 4, the best performing model is presented, together with its performance metrics. In chapter 5, the hyperparameter analyses and ablation studies that led to the model of chapter 4 are presented. Chapter 6 contains a general reflection of this project and chapter 7 contains the final words. An appendix with a link to the code, as well as instructions for running that code, is provided in chapter 8.

2 EDA AND PREPROCESSING

2.1 Exploratory Data Analysis

The data that was provided consists of 50,000 training examples and 10,000 testing examples. Each example has 128 features and 1 output label. There are ten target labels. Figure 1 shows that the target labels are equally distributed over the 50,000 training examples, i.e. there are 5,000 examples for every category. This means that the training data is perfectly balanced.

Fig. 1. Distribution of target labels over training examples

Figure 2 contains a histogram showing the distribution of the values of the training data. Here, all 128 values of all the training examples have been taken together and visualised all at once. It is clear that the data is centred around 0 and that most of the values are rather small. However, there are outliers. The largest number in the data set is 25.58 and the smallest number is -23.42.

Fig. 2. Distribution of feature values in the training data

This begs the question; Are the high / low values clustered within one example? Perhaps they tend to get clustered in a specific dimension of each example? Or do they occur randomly?

We can gain additional insights by calculating the average value of each example, and plotting this information in another histogram. In figure 3a (left), the values are taken as they are given, but in figure 3b (right), the absolute value of each value is taken. We do this, because (i) we are interested in whether there are outliers, not in their sign and (ii) the positive and negative outliers might cancel each other out if they occur together in an example.

In figure 3a, it is clear that the average value of each example is between -0.5 and 0.5. In figure 3b, we see that the average absolute value mostly lies between 0 and 1, with very few exceptions. More importantly, there are no outliers. This implies that the large and small outlier values are not concentrated within specific examples; If this was the case, then the average absolute values of such examples would be very high. We can therefore conclude that the outlier values are randomly distributed between the examples.

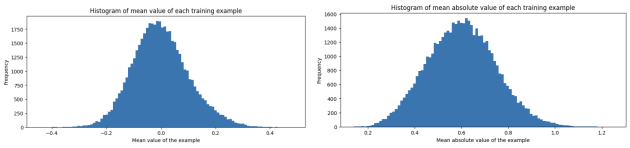

Fig. 3. Histogram of the average value (left) and average absolute value (right) of all of the 50,000 training examples

This leads us to the next question; Are the outliers concentrated in a specific dimension? Every example contains 128 feature dimensions. It is conceivable that one of these dimensions might have a higher average value than the other. A similar analysis was done for each of the feature dimensions. The results are shown in figure 4a and figure 4b.

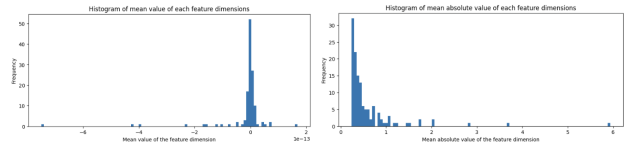

Fig. 4. Histogram of the average value (left) and average absolute value (right) of the 128 feature dimensions.

Figure 4a shows that the average value of each feature dimension is very small, namely in the order of magnitude of 1e-13. This is a consequence of the fact that (i) there are 50,000 examples for each feature dimension and (ii) they can take either positive or negative values. Due to the sheer number of examples, these values cancel each other out to result in a mean value that is very close to zero.

Figure 4b shows a more interesting situation; The average absolute value of the feature dimensions is not distributed normally. There is a pronounced outlier whose average absolute value is close to 6. Further analysis reveals that it is the feature dimension at index 0 that has this high average absolute value. Second place goes to the feature at index 1 and third place goes to the feature at index 2, and so on. This behaviour is visible in figure 5.

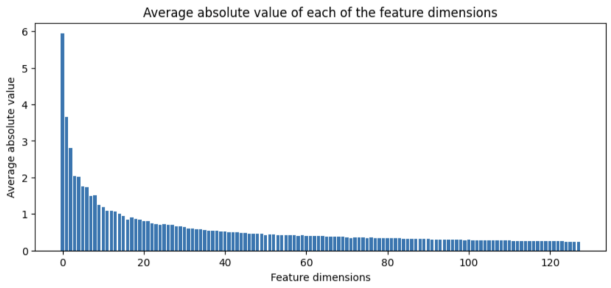

Fig. 5. Average absolute value of the 128 feature dimensions

It is also possible to plot the average value of the example, i.e. the average of the 128 features, in relation to the 10 target labels. This relationship is best visualised using a boxplot, as seen in figure 6. Figure 6a shows the average value of the examples and figure 6b shows the average absolute value.

These figures show that there is some variation between the average (absolute) value of the examples and the category they belong to. This is an indication that it will indeed be possible to predict the target category based on the input data. This is still a very low-level analysis though, as each example consists of 128 features, and only the average of these features is taken into account here.

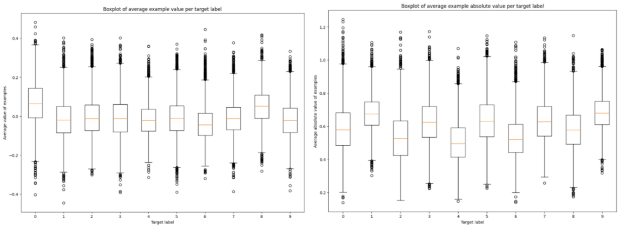

Fig. 6. Boxplot showing the distribution of average examples values (left) and average example absolute values (right) for each of the ten target categories

Lastly, analysis shows that there are no missing values in the training data.

2.2 Data preprocessing

2.2.1 *Standardisation and Normalisation.*

Standardisation means transforming the input data so that the mean of the data is 0 and that the standard deviation is equal to 1. Normalisation transforms the data to a range between 0 and 1. Using the preprocessing toolkit from scikit-learn, we can apply these transformations to our data. The result is visible in figure 7.

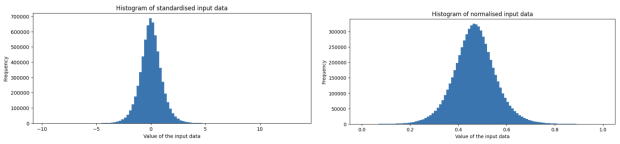

Fig. 7. Standardised (left) and normalised (right) version of the histogram plotting out the values of the training data

We see that the initial value distribution of figure 2 is somewhat similar to the standardised version in figure 7a. Standardisation as a method for preprocessing is particularly useful when features of the input data vary wildly in their size. This is most common in cases where the features are described in different units, such as cms and kms. This will cause the features with higher values to become more dominant in the neural network.

In our case, we have seen in figure 5 that there are indeed some differences in the average values of each of the 128 features. However, these are not differences at a level of orders of magnitude. Nevertheless, standardising the data might prove beneficial. Figure 5 has been recreated with the standardised data of figure 7a in figure

8a. It is clear that all features now have a similar average value. Normalisation achieves a similar result in terms of average value per feature in figure 8b.

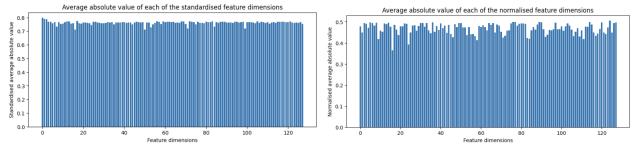

Fig. 8. Average absolute value of the 128 feature dimensions in case of standardisation (left) and normalisation (right)

After designing a basic neural network (chapter 3), we have done some experiments to ascertain which method of preprocessing yielded the best results. This discussion can be found in chapter 5.

2.2.2 *Principal component analysis.*

Principal component analysis (PCA) is a technique to transform data so that as much variance as possible is captured in each sequential feature dimension. In other words, the transformation captures as much variance as possible in the first dimension. The second dimension is perpendicular to the first and captures as much variance as possible as well. In principle, this process can be followed until we have reached the same number of feature dimensions we started with. These feature dimensions are now called 'principal components'.

In practice, PCA is often used as a dimensionality reduction technique. It is often possible to capture a large amount of the variance with a small number of principal components. By cutting down the number of feature dimensions, the computational effort can be heavily reduced.

Doing a PCA on the data set of this project yielded interesting results. Figure 9 shows that the captured variance increases linearly with each of the principal components. In other words, each of the original feature dimensions already carried more or less the same amount of variance. Therefore, it is not sensible to include PCA in this project, as the variance is already maximally distributed over the feature dimensions of the original data set.

2.2.3 *Label encoding (one-hot).*

The final piece of preprocessing pertains to the target labels. There are ten target categories, denoted by a value of 0 to 9. In order to be able to compare the predicted target category with the actual target category using cross-entropy loss, the target label needs to be a one-hot vector. For example, if a sample has as its target label the number 3, then this label will be transformed to [0, 0, 0, 1, 0, 0, 0, 0, 0, 0].

3 MODULES

3.1 A multilayer perceptron (MLP) as a classifier

A multilayer perceptron is a neural network that consists of three types of layers: an input layer, hidden layers, and an output layer. There can be more than one hidden layer in an MLP. In this project, an MLP is developed that is trained on the 50,000 training examples and tested on the 10,000 testing examples. By adjusting its

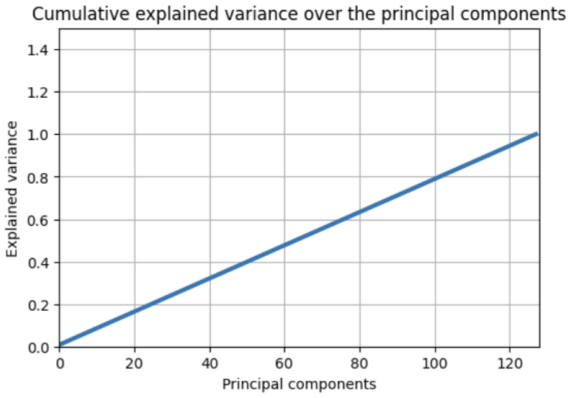

Fig. 9. Cumulative explained variance per principal component

parameters in the training stage, the neural network is able to learn how to distinguish data in one of the ten target categories based on the content of its 128 feature dimensions. This adjustment of parameters is realised in the backpropagation of the training stage.

3.2 Multiple hidden layers

A neural network consists of an input layer, a number of hidden layers, and finally an output layer. In principle, a neural network with one hidden layer can be used to approximate any kind of behaviour, as long as there are enough neurons in that layer. However, it might be more computationally effective to increase the number of layers instead of the number of neurons per layer. In chapter 5.3, we ran experiments with different numbers of hidden layers as well as various sizes for hidden layers to assess which is best for an increase in classification performance.

3.3 ReLU activation function

We added ReLU activation function to the list of our activation functions. The ReLU activation function has the advantage that it does not activate all the neurons at the same time. The neurons will be deactivated if the output of the linear transformation is less than 0. For the negative input values, the result is zero, that means the neuron does not get activated. (This is called "sparse activation".) Since only a certain number of neurons are activated, the ReLU function is computationally more efficient in comparison with the sigmoid and tanh function.

Another advantage of ReLU over the logistic and tanh functions, is that it suffers from fewer vanishing gradient problems. For very large and very small values, the gradient of the logistic and tanh functions are very small, meaning that backpropagation becomes very inefficient. ReLU does not suffer from this problem for positive values.

3.4 Weight decay

Weight decay or L2 Regularisation is a regularisation method. As such, it aims at reducing the risk of overfitting. This is a phenomenon where the neural network performs very well on the training data but performs poorly on the testing data. It has been proven

empirically that applying weight decay the chance of overfitting decreases.

Weight decay limits the growth of weights in the network. Indeed, large weights indicate a complex model which as such is more likely to have overfitted the training data. The value of weight decay is a hyperparameter of the model and as such has to be tuned; the greater the value chosen for weight decay the greater the penalisation of large weights.

3.5 Momentum in SGD

The reason for using momentum in SGD (Stochastic Gradient Descent) is to overcome some of the limitations of traditional optimization methods in SGD. One of the drawbacks of SGD is that it can get stuck in local minima, or slow down when encountering flat regions or saddle points in the loss function. However, momentum allows the optimization algorithm to continue moving even when the gradients are small, and it reduces oscillations, resulting in faster convergence to the optimal solution. Consequently, using momentum in SGD can enhance optimization performance and lead to faster convergence, especially when addressing complex and high-dimensional problems.

The main principle of momentum is that it remembers the general "direction" of the previous backpropagation step and incorporates it in the current step. Therefore, it will be less likely to suffer slow convergence or from deviation from its course due to local variations in the gradient.

3.6 Dropout

Dropout is another regularisation method. In dropout, during training, a certain proportion of perceptrons of each layer (except the output layer) are randomly "turned off", i.e. their output is set to zero. This forces the neural network not to become too dependent on any one neuron. Being a regularisation method, dropout aims at preventing any one weight becoming too large, as this is a sign of an overfitted neural network.

Dropout can be seen as a way of simulating multiple, smaller, NN architectures during training. Given this characteristic, a drawback of dropout is its impact on time required for training to converge, which depending on the case can be noticeably increased. Therefore, to achieve good performance, the model needs to be trained longer and it should be of a bigger size, meaning more layers and more neurons per layer.

3.7 Softmax and cross-entropy loss

Softmax is a function that yields a vector of probabilities. Indeed, given an input vector it returns the conditional probabilities for each output class; Hence its utility in classification tasks. In a neural network used as a classifier, the output layer is a vector with as many neurons as there are the classes; By applying Softmax we obtain the probability that a datapoint belongs to any of the possible classes, with all the given probabilities summing up to one. Softmax and cross-entropy loss often go hand in hand. Indeed, cross-entropy is the loss function used to measure the distance between the ground truth and the predicted probabilities. The loss function is used during training to adjust the learnable parameters of the network, the weights and biases. Here, the objective is to minimise the loss. Cross-entropy can be decomposed into two terms to better understand it: the Entropy of the ground truth and the KL divergence. The first term measures the degree of randomness of the output that we are trying to predict, the bigger it is the greater the penalty. The second one measures the information lost when trying to approximate the p.d.f. of the ground truth, the furthest away is our approximation the bigger the increment of the loss.

3.8 Mini-batch training

Mini-batch training can be seen as a crossover method in between batch gradient descent and stochastic gradient descent (SGD). Indeed mini-batch gradient descent yields a model with a less noisy, more reliable gradient compared to SGD. It also increases computational efficiency as well as reduces memory usage. This technique requires dividing the training set in batches of a fixed size, called mini-batches. The size of which (i.e. number of training examples contained), is a hyperparameter to be tuned. During training the update of the network's parameters happens once for every mini-batch, which makes the training more efficient; indeed, the greater the mini-batch size the greater the speed up in training time. Although a too large mini-batch size may also cause premature convergence of the model. The update exploits the average gradient (i.e. the gradients from every data point in the mini-batch is accumulated and the average is taken) which ultimately allows to reduce the variance observed in the gradient compared to SGD. Furthermore, dividing in mini-batches allows one to keep fewer training samples in memory simultaneously, compared to batch GD where the entire training set has to be stored in memory given that the update happens only once after the entire batch is processed. Perhaps the biggest advantage of mini-batch training resides in the vectorisation of the operations. In the vectorised version, rather than having a for loop that scans through all the elements of a mini-batch and evaluate the gradient, we feed the entire mini-batch to the network all at once and exploit matrix operations. This, in turn, enables parallelisation, hence allows to leverage GPU for training drastically decreasing computational time.

3.9 Batch normalisation

Batch Normalisation is a normalisation method that normalises and scales the inputs using the mean and variances of the all inputs of that layer. Batch normalisation changes the distribution of each neuron's activation to normal distribution (with the mean at zero and a standard deviation of one) so the next layers do not have to keep adapting to a change in distribution. If this change in distribution is systemic and accumulates over time, this is called a covariance shift. Batch normalisation is introduced exactly to reduce this covariance shift. By introducing batch normalisation, each neuron's activation becomes distributed in a Gaussian manner. In other words, it is usually not activated (as the mean is centred at zero), though sometimes it is a bit active, but it very rarely is very active. This behaviour reduces the chance of vanishing gradients for logistic and tanh activation functions. This has a myriad of benefits for training a neural network, such as faster convergence (reduced training times) and a reduced demand for regularisation. It also

allows for the use of saturating nonlinearities in deep neural networks, such as the logistic function or tanh, as the risk of getting stuck in saturating ranges is reduced by the batch normalisation.

3.10 Leaky ReLU and GELU activation functions

As previously mentioned, the ReLU (rectified linear unit) has some clear advantages over the logistic and tanh activation functions, as it has fewer vanishing gradient problems and has sparse activation. However, it does suffer from the "dying ReLU problem", where the gradient is not able to flow backward because the gradient is zero for negative values. This can cause neurons to become permanently inactive. To solve this problem the leaky ReLU was introduced. This generally comes at the cost of a reduced performance, which is confirmed in our experiments of section 5.2.

We have also implemented the Gaussian Error Linear Unit (GELU). It can be thought of as a smoother version of the ReLU function with a bump around zero, ensuring that the function remains differentiable at this point. GELU is generally considered more apt to learn complex patterns of data. GELU aims to combine two functionalities in one: the ReLU activation function, which deterministically multiplies inputs with either 0 or one, and dropout regularisation, which stochastically multiplies inputs with 0 or 1. The three functions, as well as their derivatives, are shown in figure 10.

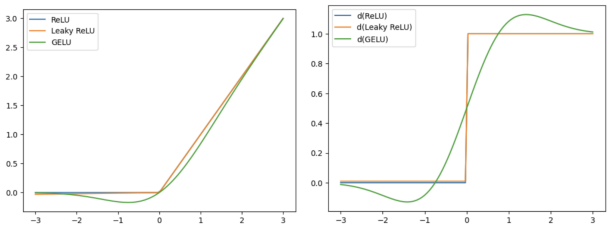

Fig. 10. Graph of three activation functions (left) and their derivatives (right)

4 DESIGN OF THE BEST PERFORMING MODEL

4.1 Evaluation metrics

Several evaluation metrics exist for gauging the performance of classification algorithms. We are dealing with a multiclass classification algorithm, with 10 target categories. The simplest evaluation metric is that of accuracy, i.e. the number of correct classifications divided by the total number of examples. Accuracy is not a great metric for unbalanced data sets, but since we are dealing with a balanced test set, this is not an issue here.

There are three more essential evaluation metrics: Precision, recall, and F1-score. These pertain to one specific target class. Precision indicates how likely it is for an example to be of the target class that it is predicted to. Recall indicates how likely an example will be classified in its correct target class. F1-score is the harmonic mean of these two metrics. Each target class has their own F1-score, but it is possible to average the ten F1-scores into a total F1-score for the model. We will use this model-wide F1-score as well as the accuracy score.

We will also measure and compare the time required to (i) train a neural network and to (ii) test a neural network.

4.2 Best performing model

Following from the experiments that are undertaken in chapter 5, the best performing model has the following properties as specified in table 1.

Table 1. Design of the best performing model

Module / Hyperparameter	Value
Preprocessing	None
Activation function	ReLU
Number of hidden layers	2
Neurons per layer	512, 256
Learning rate	0.001
Momentum	0.9
Weight decay	0
Dropout rate	0
Mini-batch training	No
Batch normalisation	No

The performance of the best performing model on test data after 25 epochs of training is given in table 2.

Table 2. Design of the best performing model

Performance metric	Value
Accuracy	54.42%
F1-score	54.17%
Training time	40 min 34 s
Testing time	1.66 s

5 EXPERIMENTS AND RESULTS

When designing a neural network, there are a vast number of design decisions that one has to make, including the tuning of hyperparameters and the decision whether or not to include certain modules and if so, to what extent. Here is a non-exhaustive list of such decisions:

- How many layers should the neural network have?
- How many neurons should each layer have?
- What activation function should each neuron have?
- At what value should we set the learning rate of the model? Should we include variable learning rate?
- How many epochs should the neural network be trained for?
- Should there be weight decay? If so, what value should we choose?
- Should there be momentum? If so, what value should the momentum term have?
- Should there be dropout? If so, what value should the dropout rate be?
- Should there be mini-batch training? If so, what should be the size of the batches?

- Should there be batch normalisation between the layers?

These are all non-trivial decisions to make. In the design of neural networks, researchers often rely on grid search to do hyperparameter tuning. In grid search, the values of hyperparameters are changed one-by-one and the impact of each change is measured. Typically, a 10-fold cross-validation is used for each model. This means that each model is trained 10 times using 9/10 of the training data where the other 1/10 is used for testing. (This is usually referred to as validation data.) If you want to do a grid search for 10 hyperparameters, each having 10 possible values, this means you will have to train and test 10^{10} models 10 times. This requires a lot of computational effort, not to speak of the amount of time that is needed for this.

Because of the sheer number of decisions that we need to make in this project, we do not deem it realistic to have a full-fledged grid search included in this project, especially considering the fact that the code needs to be run in a reasonable amount of time. Therefore, we have decided to single out specific hyperparameters and perform experiments in isolation.

5.1 Method of preprocessing

The first experiment consisted a comparison of the performance of a basic neural network model where the input data was preprocessed in three different ways: No preprocessing, standardisation, and normalisation. In order to do this, we implemented a stratified 10 fold cross-validation on the training data to ensure there is no data leakage from the test set to the training set. We then took the average of four relevant evaluation metrics of the 10 separately trained neural networks for each of the three preprocessing methods. The results are visualised in figure 11.

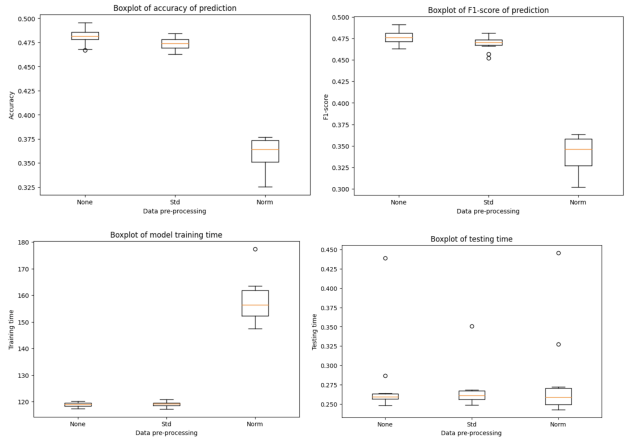

Fig. 11. Prediction accuracy, F1-score, training time, and testing time of neural networks for each of the three preprocessing methods

From this analysis, it is clear that normalisation works counterproductive to the accuracy, F1-score, and training time of the neural network. It also seems like standardisation has little impact, and even makes the prediction accuracy and F1-score slightly worse. This is likely a consequence of the fact that the original data was already nearly distributed in a standardised fashion. As a result of this experiment, we have decided not to preprocess the input data.

5.2 Impact of different activation functions

For the purposes of this report, we have compared the performance of the neural network for three nonlinear activation functions: ReLU, leaky ReLU, and GELU.

Figure 12 shows our findings for the effect of implementing these activation functions of the model's accuracy, F1-score, training time, and testing time. It is found that, for accuracy and F1-score, ReLU performs best, followed by leaky ReLU, followed by GELU. It seems that leaky ReLU has the shortest training time, but ReLU has the shortest testing time. Most striking is the marked increase of the training time for the neural network with GELU activation functions. This is a consequence of the fact that calculating the GELU, both its normal version and its derivative, requires a lot more computation than the simple derivative of both ReLU and leaky ReLU.

As we will see with weight decay and dropout, regularisation methods prove ineffective at increasing the performance of our model. This is a consequence of the fact that we have used a basic model, with only 2 hidden layers, to obtain most of the experimental results of chapter 5 This means that the model does not overfit on the data and regularisation methods do more harm than good. As GELU contains the functionality of dropout regularisation (see chapter 3.10), it results in a decrease in accuracy and F1-score for our network.

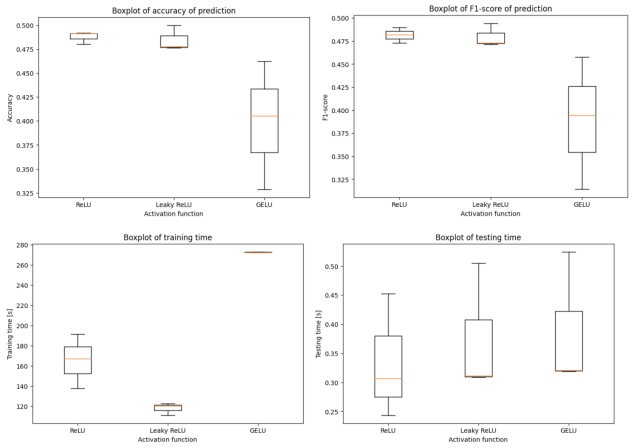

Fig. 12. Evaluation metrics of neural networks for each of the three nonlinear activation functions

5.3 Impact of number of hidden layers and their width

An important decision that needs to be made when designing a neural network, is the number of hidden layers and the number of neurons within each layer. In this subsection, we explore the performance of the model when varying these.

The baseline model for this comparison consists of two hidden layers. The first hidden layer has 64 neurons and the second has 32. The table below shows the variations that were analysed. In terms

of varying the depth of the network, one network with four hidden layers and another with eight were included. In terms of varying the width, three networks were created: One with two times the number of neurons per layer, one with four times the neurons per layer, and one with eight times the number of neurons per layer. These different architectures are shown in table 3

Table 3. Different neural network architectures

Baseline	[128, 64, 32, 10]
Depth x2	[128, 64, 64, 32, 32, 10]
Depth x4	[128, 64, 64, 64, 64, 32, 32, 32, 32, 10]
Width x2	[128, 128, 64, 10]
Width x4	[128, 256, 128, 10]
Width x8	[128, 512, 256, 10]

Figure 13 shows how these networks performed. It is clear that increasing the depth of the network had a negative effect of accuracy and F1-score, while increasing the width had a positive effect. In terms of training and testing time, every increase in the number of neurons led to an increase in training and testing time. However, increasing the width had a larger impact than increasing the depth.

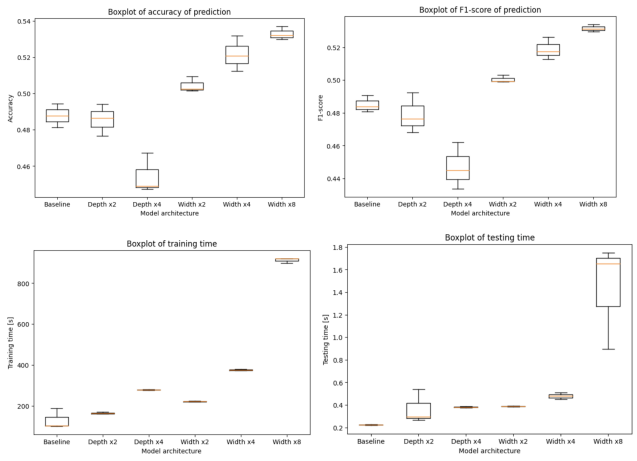

Fig. 13. Evaluation metrics of neural networks for each of the different architectures

5.4 Impact of learning rate

The learning rate of a neural network is an important hyperparameter to tune. It determines the size of the step we take after having determined the gradient of change in the backward pass. The trade-off is as follows: A small learning rate may result in a training process that is to small, i.e. the network does not converge within the allocated number of epochs, but a learning rate that is too large may result in an unstable learning process where the optimum is overshot or where the model converges to a suboptimal solution.

For the purposes of this project, we analysed the performance of the neural network for learning rate values of 0.0001, 0.001, 0.01, and 0.1. The results are visible in figure 14. We can see that the accuracy and F1-score of the predictions are highest for a learning rate of value 0.001. We can therefore conclude that a learning rate of 0.0001 is too small (i.e. no convergence within the number of epochs) and a learning rate of 0.01 is too large (i.e. unstable optimisation). In this experiment, only 10 epochs of training were included.

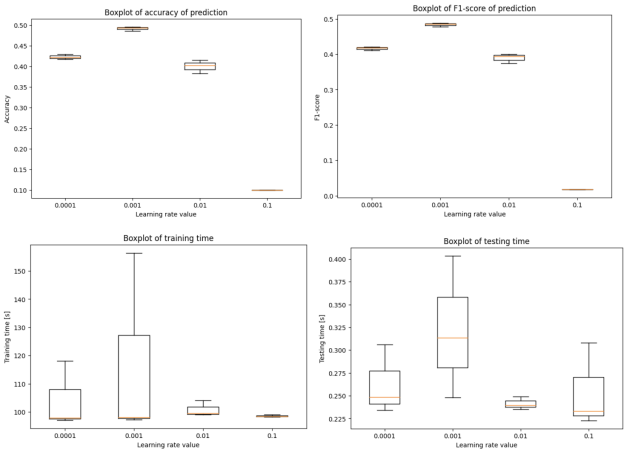

Fig. 14. Evaluation metrics of neural networks for different learning rate values

5.5 Impact of momentum

For momentum, we analysed the performance of the models for the values 0, 0.9, 0.95, and 0.99. The results are visible in figure 15. From the figure, it is clear that the training time decreases if the momentum value is not equal to 0. This is expected behaviour. Unfortunately, a momentum value that does not equal zero seems to result in lower accuracy and F1-scores as well. Nevertheless, we believe that including the momentum at a value of 0.9 is a wise decision, as it speeds up the training process significantly, it does not reduce the performance dramatically, and it produces more reliable results. This last reason refers to the fact that the whiskers of the box plot of figure 15 are much closer to each other for a momentum value of 0.9 than a momentum value of 0. This reliability in terms of results is an advantage in its own right.

5.6 Impact of weight decay

For weight decay, we analysed the performance of the model for the values 0, 0.01, and 0.02. From figure 16, it is clear that any weight decay value larger than 0 reduced the model's performance dramatically. Including weight decay reduced the accuracy of the model from 50% to 10%. Given it is a 10-class classification problem, this means that the model is assigning labels at random at this point.

As explained in chapter 3.4, weight decay is a regularisation method. It aims at reducing the size of the weights of the model in a bid to reduce the chance that the model is overfitting. Since we are dealing with a rather small model here (only two hidden layers of size 64 and 32 and trained for only 10 epochs), overfitting is not a problem to start with. Therefore, by reducing the weights of the

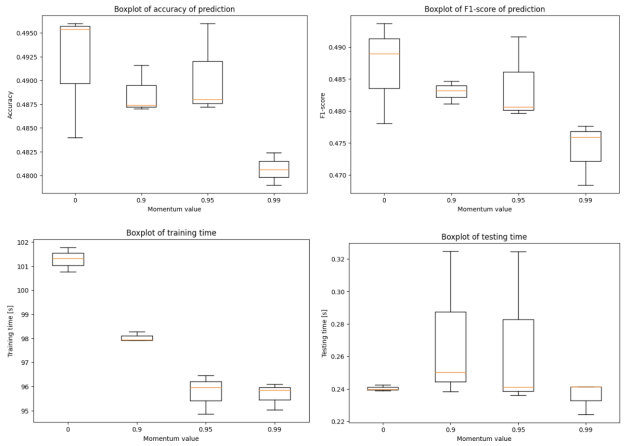

Fig. 15. Evaluation metrics of neural networks for different momentum values

model, we are simply actively preventing the model from learning the characteristics of the training data.

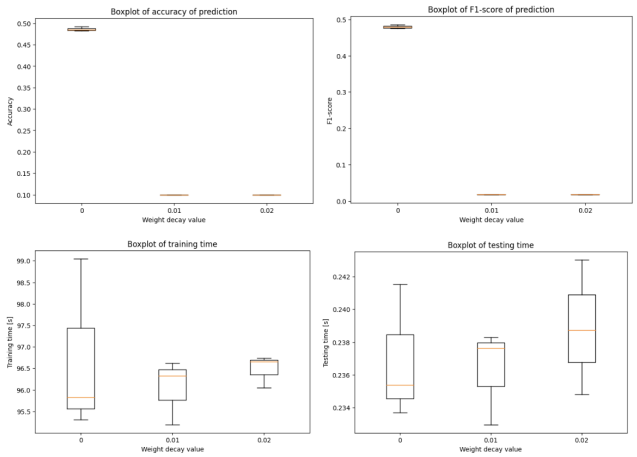

Fig. 16. Evaluation metrics of neural networks for different weight decay values

5.7 Impact of dropout

We have analysed the impact that including dropout with a dropout rate of 0.5 has on each of the four evaluation metrics. This is done three times; One time for a simple model, one time for a model that is trained 5 times as long, and one time for a model that is four times as wide. We have included an analysis of these three models, because we believe that the more complex models have a higher chance of overfitting the data. Thus, dropout should play a more important role in these networks. Here is an overview of the three networks:

- Baseline model: Two hidden layers (64 and 32 neurons), trained for 10 epochs.
- Second model: Two hidden layers (64 and 32 neurons), trained for 50 epochs.
- Third model: Two hidden layers (256 and 128 neurons), trained for 10 epochs.

The results are visible in figure 17. We can clearly see that our implementation of dropout, as a separate layer, has little impact on the training and test time. It is very clear that the training time basically increases by a factor of five when increasing the epochs from 10 to 50. This is expected behaviour.

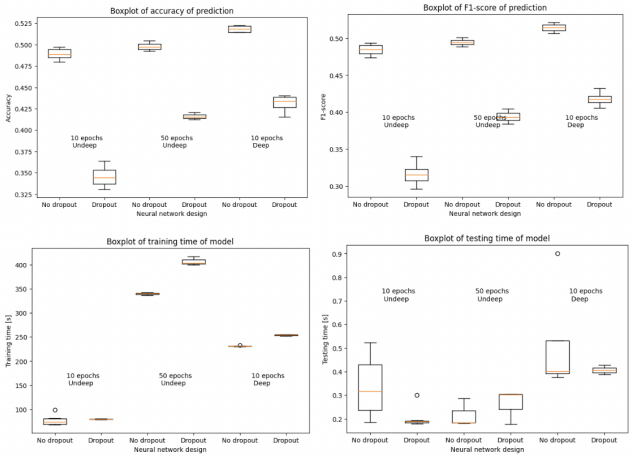

Fig. 17. Evaluation metrics of neural networks for different dropout rates, different numbers of training epochs, and different architectures

In terms of accuracy and F1-score, it is remarkable that the network with dropout performs seriously worse. Dropout is a regularisation method that prevents overfitting, which might occur if a network has a lot of parameters (layers and neurons) or if a network is trained for many epochs. We can therefore conclude that our network is not complex enough and not trained for enough epochs for dropout to have a beneficial effect. However, we can already see that the performance of the network with dropout increases relatively to the same model without dropout if the model is trained for 50 epochs instead of 10 or when it consists of more neurons per layer. This holds with the aforementioned logic.

5.8 Impact of mini-batch training

To finetune the size used in the mini-batches we implemented a grid search testing different combinations of mini-batches sizes and epochs. The results were compared against each other as well as against a model with no mini-batch training. All the different sizes tested are powers of two since CPU as well as GPU come with a storage capacity with power of two. Our implemented version of mini-batch training is not vectorised, indeed it executes a "for-loop" as many times as are the samples in a mini-batch, hence our implementation can not make use of the GPU parallelising matrix operations. However, since the update of the network parameters happens only once for each mini-batch, we still expect computational time to decrease as the batch size increases. Given the speed up in training time associated with larger mini-batch

sizes, models trained with greater mini-batch sizes were also run for more epochs.Hence the following hyperparameter couples (mini batch size, epochs) were tested: (32,10),(64,15),(256,25), (1024,50), (no MB,10).

From the results obtained in figure 18 it is clear how in this scenario as the mini-batch size increases the accuracy as well as the F1-score obtained by the model drops. Hence, in the final architecture, SGD was preferred over mini-batch training.

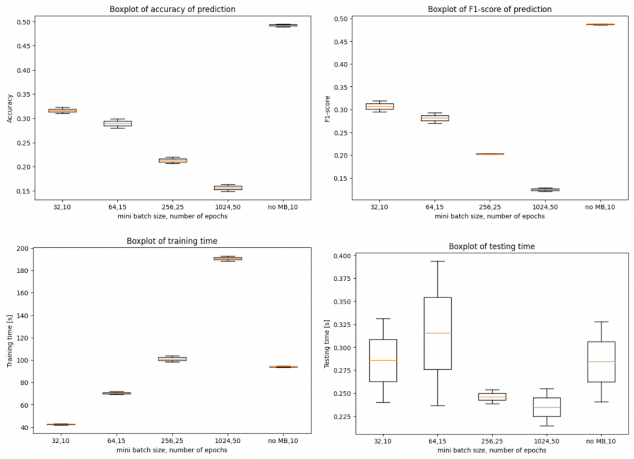

Fig. 18. Evaluation metrics of neural networks with and without mini-batch training

5.9 Impact of batch normalisation

In a simple ablation experiment, we assessed whether including batch normalisation has a positive or a negative effect on our model's performance. From figure 19, it is clear that including this module has a negative effect: It reduces the accuracy and F1-score significantly, while doubling the training time.

The disappointing results of including the batch normalisation can be explained by two factors: For one, the network that was used to test the performance was not very deep, only consisting of two hidden layers. Hence, covariance shift was most probably not a problem that needed to be dealt with. Furthermore, ReLU was used as the activation function of the network. Batch normalisation reduces the vanishing gradient problems, but this problem mainly occurs for the logistic and tanh functions in the first place. So including the batch normalisation in this model was not necessary, as the two problems that it aims to solve were not present in the model.

6 DISCUSSION/REFLECTION

At the start of this project, we have designed a neural network to classify data into one of ten target categories. This necessitated the implementation of a cross-entropy loss function that is preceded by a softmax operation. After this initial stage, we implemented several modules and tweaked the parameters of these modules to see what effect they had on the model's performance.

In doing these experiments, we held one consideration in the back of our minds; The running time of the resulting Google Colab sheet

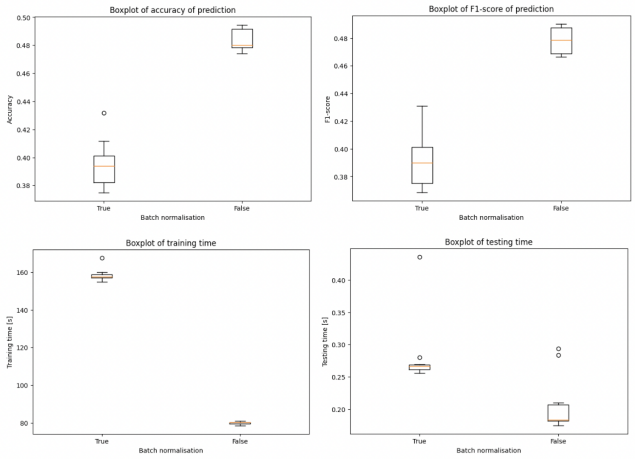

Fig. 19. Evaluation metrics of neural networks with and without batch normalisation

should run in "feasible time", which was taken to mean a couple of hours. Because of the large number of tests that we needed to carry out (this refers to the hyperparameter analyses and ablation studies), we therefore needed to keep the complexity of each individual neural network rather limited in order to keep the runtime of the entire Google Colab sheet within reasonable bounds. Concretely, this resulted in neural networks of only a few layers deep (usually only two hidden layers) and maximally 512 neurons wide (but usually only 128 neurons wide).

Because of these constraints, the methods aimed at regularisation, i.e. countering overfitting, did not yield any positive results. This applies to weight decay, dropout, and also the use of GELU as an activation function for example. Furthermore, batch normalisation also did not yield any improvement, as it reduces covariance shift, but this only occurs in deep neural networks in the first place.

Nevertheless, doing these hyperparameter analyses and ablation studies were useful, as they allowed us to gain insight into the effects that including each module has on the performance of the neural network. Furthermore, the analyses of the activation functions and the optimal value of the learning rate and momentum helped us a great deal in obtaining a high-performing final model.

7 CONCLUSION

In this project, we designed a neural network to be trained as a classification model. It classifies a data point with 128 feature dimensions in one of ten target categories. The basic neural network is expanded with several modules: Multiple hidden layers, advanced activation functions such as ReLU and GELU, weight decay, momentum, dropout, mini-batch training, and batch normalisation. The impact that each of these modules has on the performance has been analysed and documented in this report. Hyperparameter analyses for the learning rate and parameters such as the momentum value and batch size for mini-batch training have been performed as well. We evaluated the performance based on four metrics: Accuracy, F1-score, training time, and testing time. This has resulted in the

design that was presented as the best performing model in chapter 44 of this report.

8 APPENDIX

8.1 Link to code

8.2 Instruction for running the code

The code is provided as a Google Colab file. To run the code, please run each individual cell sequentially. To do this, you can manually run each cell, or you can choose to run all cells sequentially automatically by pressing CMD+F9 (MAC) or CTRL+F9 (PC).

The second cell in the sheet downloads the relevant data from Google Drive. When this cell is run, you will need to give Google Colab permission to access Google Drive. After this, all cells will run without any further input needed from your side.

8.3 Specifications

Hardware: MacBook Pro 2021, Apple M1 Pro chip, 32GB RAM
Software: macOS 12.2.1, Google Colab running Python 3.9