Deep Learning Report

- Multi-Class Classification Using a Multilayer Neural Network[†] -

Alkhorayef K.: 654133210
Gehring, S.: 500315150
Maslim F.: 490450547
May 01, 2020

THE UNIVERSITY OF SYDNEY

[†]Assignment 1, Semester 1 (S1C)

Contents

1 Introduction

1.1 Aim of the Study

This study aims to build a feedforward multi-layer perceptron (MLP) whilst implementing modern concepts within the field of deep learning. The process of building the model is done without the utilization of any deep learning library. Therefore, the study group has to dissect each element of the model. This improves the learning experience. Moreover, the implementation of concepts from scratch supported the understanding of the effects these methods have on the model. This understanding enabled better testing and tuning of parameters which result in a better preforming model.

1.2 Importance of the Study

In the field of deep learning, multi-layer perceptron (MLP) models that use advanced concepts and methods have seen breakthrough in commercial applications such as image recognition, natural language applications and self-driving vehicles. The study undergoes the challenge of building a sophisticated model by implementing many concepts and tuning different parameters. In addition, it allows the Group to become familiar with the latest trends and state of the art methods in research whilst designing a deep neural network model.

2 Methods

This section outlines different modules and functions implemented in the MLP model. The flowchart below, outlines the process of the model.

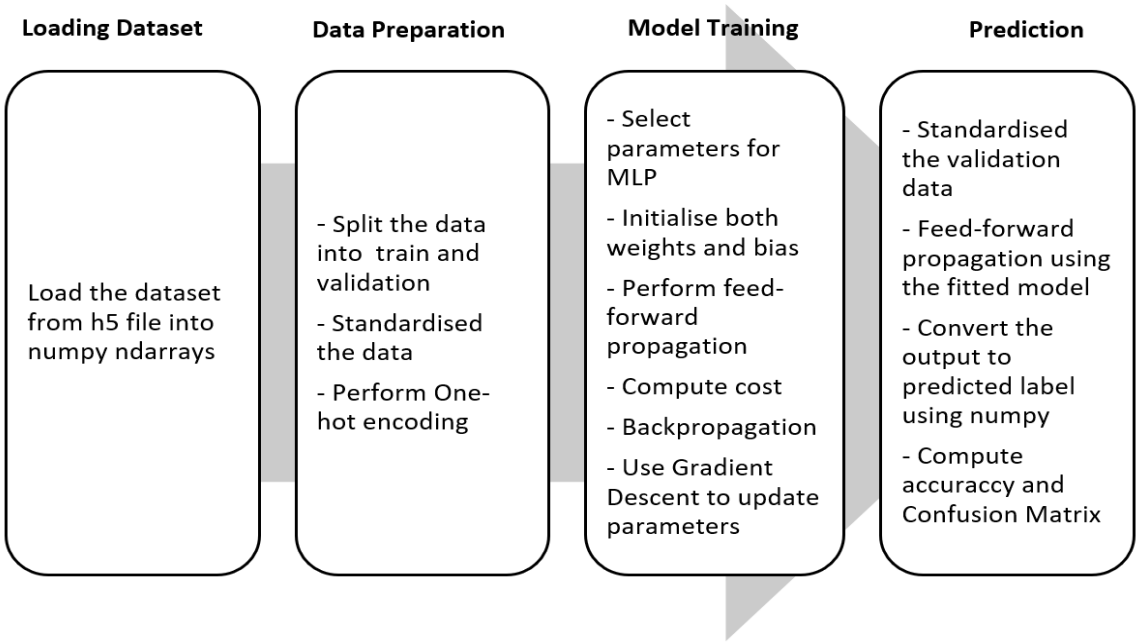


Figure 1: Flowchart of the Model

2.1 Basic MLP

The basic functions define the model and make a prediction possible. These functions are used in almost every MLP.

2.1.1 Forward Propagation

Forward propagation is one of the most important elements of our MLP model. During the forward propagation process, the input feature matrix is fed to the first hidden layer and forward propagated to the output layer.

The forward propagation implemented in the MLP model works by multiplying the input matrix with the weight and bias on each hidden layer to get the linear transformation of the input matrix called lin_output. During this process non-linearity was applied on lin_output using activation functions. The individual output matrix then act as the input for the next layer and undergoes transformations in a similar fashion. This process is repeated until it reaches the output layer.

2.1.2 Backward Propagation

The underlying MLP uses back propagation to calculate the gradients/derivatives of the loss function with respect to the weights and bias. Hereafter, the gradients are used iteratively in the gradient descent update steps, to update the weights and bias terms.

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{w_{ji}} \tag{1}$$

This minimizes the loss and improves the accuracy of the model until a minimum is found. To make the calculation more comprehensible, the sensitivity of the layers was used:

$$\delta_j = -\frac{\partial J}{\partial net_j} \tag{2}$$

2.1.3 Initialization of Weights and Bias

The weights in a MLP need to be initialised so that the activation result is neither too large nor too small. In general, bad weight initialisation could result in the exploding or vanishing gradient problem during backpropagation. In order to overcome this problem, the initial weights were set according to the Kaiming uniform distribution (Goel, S., 2019).

$$W \sim U\left[\pm\frac{\sqrt{6}}{\sqrt{n_{\text{out}}}}\right] \tag{3}$$

2.2 Activation Functions

Activation functions are used to introduce non-linearity (Hansen, 2019) (Hansen, C., 2020). The choice of the activation functions usually depends on the type of data and the problem that has to be solved. In our case, the ReLU activation (2.2.1) was chosen for the hidden layers and the Softmax activation (2.2.2) for the output layer.

2.2.1 ReLU Activation

ReLU is a nonlinear function that returns the input itself or zero for negative input values. One main advantage of using ReLU is that it does not activate all neurons at the same time. This usually results in a faster training process. Moreover, ReLU has been proven to be superior compared to other simple activation functions such as Tanh and Sigmoid (Hansen, C., 2020).

$$\begin{aligned} f(x) &= MAX(0, x) \\ f'(x) &= \begin{cases} 0, & \text{if } x \leq 0 \\ 1, & \text{if } x > 0 \end{cases} \end{aligned} \tag{4}$$

2.2.2 Softmax

Softmax is an activation function that transforms the inputs into probabilities that sum to one (Oman, S., 2019). It is the core element of our MLP output layer, since it determines the predicted class label.

$$\begin{aligned} f(x) &= \frac{e^{x_i}}{\sum_j e^{x_j}} \\ f'(x) &= \sigma(j)\left(\delta_{ij} - \sigma(i)\right) \end{aligned} \tag{5}$$

2.3 Data Pre-processing

The study used a dataset that is similar to the MNIST dataset, which also uses 10 classes. The data provided is in H5 format and has a training data size of 60MB with 60000 rows and 128 features. The testing data is sized 10MB with 10000 rows and 128 features.

2.3.1 One-hot Encoding

One-hot Encoding is a pre-processing technique that helps to transform the dataset labels into a format that can be used to train our MLP model. Each category value is converted into a new column with binary value of either one or zero (Yadav, D., 2020). One-hot Encoding essentially enables the use of cross-entropy cost computation, which is required for our MLP model's backpropagation process.

2.3.2 Standardisation

Standardisation was implemented to convert the training data distribution to a standard normal distribution. During the standardisation process the mean of the dateset is subtracted from the data and the result is divided by its standard deviation (Lakhsmanan, S., 2019). The standardised dataset has mean zero and standard deviation of one. As a result, the range and variablity in the dataset is equalized. This helped to minimise biased results due to skewness in the data and ensured that large values are not more dominant than smaller values.

$$z = \frac{x - \mu}{\sigma} \tag{6}$$

2.4 Other Implemented Modules

To improve the basic functions of the model the following modules were implemented.

2.4.1 Mini-batch training

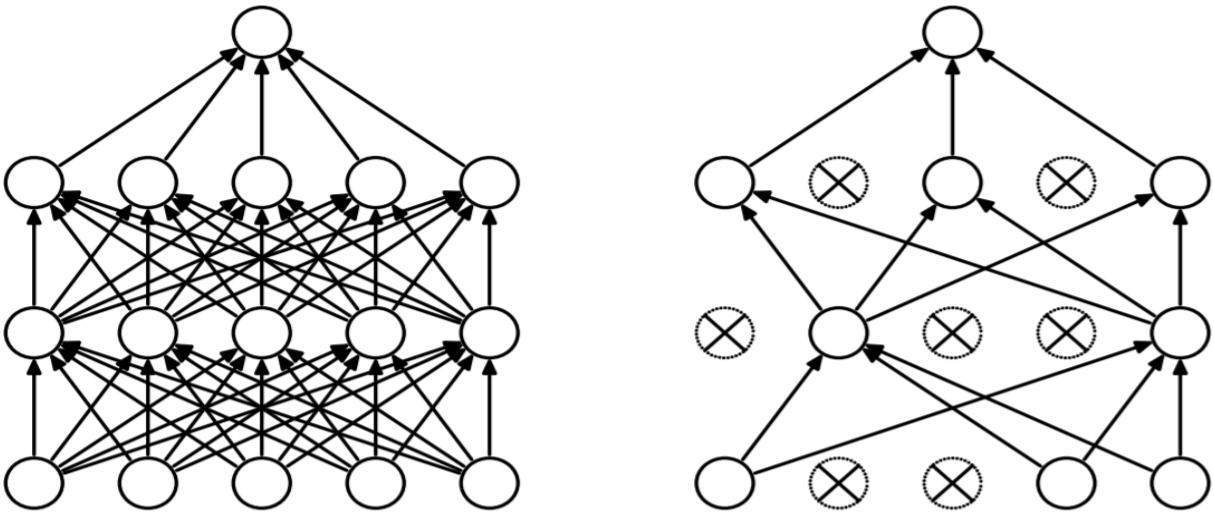
Mini-batch training reduces the time the gradient takes to converge. Therefore, the training data is divided in batches randomly.

$$\theta = \theta - \eta \cdot \nabla_\theta J\left(\theta, \mathcal{X}^{(i:i+n)}\right) \tag{7}$$

These batches are fed into the model successively. After the model has seen all data points within the mini-batch the loss function and the gradients are computed. The gradients are used to conduct the updates. This process is repeated until the preset number of iterations is reached. Since the size of the batch lies between the size of Gradient Descent and SGD, the characteristics also lie between the characteristics of the two methods.

2.4.2 Dropout

Dropout is a regularisation technique that helps preventing interdependent learning amongst neurons and reduces overfitting in neural networks (Budhiraja, A., 2016). It is both computationally cheap and easy to implement. The Dropout implemented in our MLP works by randomly dropping neurons in the hidden layers based on a binomial distribution. Figure 2 shows the used network before and after the implementation of the Dropout method (Srivastava, 2004).



(a) Standard Neural Net (b) After applying dropout.

Figure 2: Dropout Chart

2.4.3 Cross-entropy loss

Cross-entropy is commonly used to evaluate the performance of a classification model that has probability output between zero and one (Godoy, D., 2018). The Cross-entropy loss will be larger when the predicted probability diverges from the actual label. The derivation of the cross-entropy loss is used during backpropagation and gradient descent (2.1.2) to update the weights and improve the MLP model performance.

$$\begin{aligned} L &= -\frac{1}{N}\left(\sum_{i=1}^{N} y_i \cdot \log(\widehat{yi})\right) \\ \boldsymbol{L'} &= -\frac{\boldsymbol{y}_i}{\hat{\boldsymbol{y}}_i} \end{aligned} \tag{8}$$

2.4.4 Weight Decay

Many machine learning techniques require large weight vectors to correctly predict the training data points. Thus, incoherent or high dimensional problems in the training data can be explained and the training error converges to zero. Albeit a low training error can be a good sign, we don't want the model to overfit. Therefore, weight decay is used. The regularization method adds a penalty term for large weights to the loss function.

$$\min_{\theta} \hat{L}_R(\theta) = \hat{L}(\theta) + \frac{\lambda}{2}\|\theta\|_2^2 \tag{9}$$

The lambda term determines the magnitude of the negative effect of large weights and was set during the hyperparameter analysis process (3.2). The remaining term is the squared euclidean norm of the weight vector. The gradients of this loss function $\nabla \hat{L}_R(\theta)$ are used as update

gradients $\nabla_\theta J(\theta)$ in SGD with Momentum (2.4.5).

$$\nabla \hat{L}_R(\theta) = \nabla \hat{L}(\theta) + \lambda\theta \tag{10}$$

Figure 3 shows the implemented code:

```
#Weight Decay step of the individual layer
def weight_decay(self, lambd):
    #needed weight decay terms of each layer.
    weight_decay = np.square(self.W)*(lambd/2)
    #Storing the value for the weight decay gradient.
    self.grad_W_wd = self.grad_W + lambd * self.W
    #Returning the weight_decay term that is added to the Cross Entropy loss.
    return weight_decay
```

Figure 3: Weight Decay Implementation

Since, the algorithm minimizes the loss function the growth of the weights is limited.

2.4.5 Momentum in SGD

To reduce the oscillation around the minimum and accelerate the updates in the relevant direction, Momentum in SGD is used. Close to the local optima areas the surface curves are much more steeply in one direction than in another. This causes problems for the normal SGD. It jumps from one side to the other, slowly converging to the optimum. The momentum formula below uses the prior learning step to either enhance the magnitude of the update (current and prior gradient show in the same direction) or reduce the magnitude of the update (current and prior gradient show in different directions). Therefore, it calculates the gradient at the current location $\nabla_\theta J(\theta)$ and then takes an additional jump in the direction of the accumulated gradient v_{t-1}.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \tag{11}$$

```
#Momentum step of the individual layer
def momentum (self, gamma, lr):
    #Combining the weight decay gradient with the momentum term.
    self.vt = gamma * self.vt + lr * self.grad_W_wd
    #Returning the momentum update term for the weight updates.
    return self.vt
```

Figure 4: Momentum Implementation

The new implemented hyperparameter gamma determines the influence of the previous gradients and is set in subsection 3.2. The resulting algorithm converges faster and with reduced oscillation to the minimum (Figure 5).

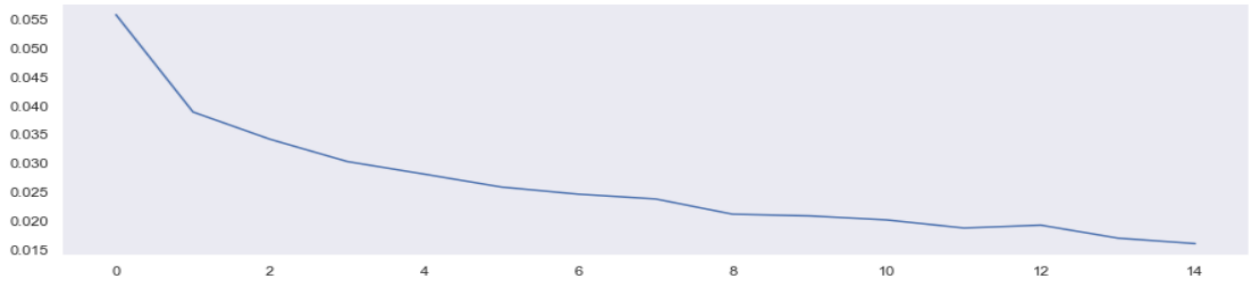


Figure 5: Loss Convergence

2.4.6 Batch Normalization

If one feature has a bigger magnitude than for the other datapoints, the differences in this feature can govern the overall distance between the datapoints. A possible result is called gradient explosion, which leads to no or slow convergence in the gradient descent steps. After normalization either all features use the same scale [0,1] or [-1,1]. The following formula shows the normalisation step:

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{12}$$

and the shift and scale mechanism:

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}\left(x_i\right) \tag{13}$$

For neural networks this is called Whitening. The normalized feature map is the same for scaled and non-scaled weights. Therefore, the gradients used in backpropagation are unaffected by the scale of the parameters before batch normalisation. This reduces training time due to less chance of exploding/vanishing gradients. Due to the lower risk for exploding/vanishing gradients, higher learning rates can be used. Moreover, it reduces the demand for regularisation since no memorisation of values and their correct answers is possible. However, too small batch sizes could lead to noisy estimations. The error increases rapidly when the batch size becomes too small, due to inaccurate batch statistics (mean, variance) estimation.

We call the batch normalization within the forward propagation:

```
#batch normalisation during forward propagation
def bn_forward(self, input, gamma, beta):
    mu = np.mean(input, axis=0)
    var = np.var(input, axis=0)
    input_hat = (input - mu) / np.sqrt(var + 1e-8)
    output = gamma * input_norm + beta
    cache = (input, input_hat, mu, var, gamma, beta)
    return output, cache, mu, var
```

Figure 6: Batch Normalisation Forward Call

and within the backward propagation:

```
#batch normalisation during backward propagation
def bn_backward(delta, cache):
    input, input_hat, mu, var, gamma, beta = self.bn_cache
    N,_ = input.shape
    input_mu = input - mu
    std_inv = 1. / np.sqrt(var + 1e-8)
    d_norm = delta * gamma
    dvar = np.sum(d_norm * input_mu, axis=0) * -0.5 * std_inv**3
    dmu = np.sum(d_norm * -std_inv, axis=0) + dvar * np.mean(-2*input_mu, axis=0)
    d = (d_norm * std_inv) + (dvar * 2 * input_mu / N) + (dmu / N)
    dgamma = np.sum(delta * input_hat, axis=0)
    dbeta = np.sum(delta, axis=0)
    return d, dgamma, dbeta
```

Figure 7: Batch Normalisation Backward Call

2.5 Evaluation Functions

The prediction accuracy is used as a base for the overall quality assessment of the algorithm. For a deeper understanding of the overall results and a starting point for the analysis in section 3 the confusion matrix is added.

2.5.1 Accuracy Evaluation

Accuracy is one of the most common measures used to evaluate classification models. It provides the percentage of correctly predicted data points. Accuracy is calculated by dividing the number of correctly classified points by the number of samples in the dataset.

$$accuracy = \frac{N_{\text{correctly classified labels}}}{N_{\text{samples}}} \times 100\% \tag{14}$$

2.5.2 Confusion Matrix

The Confusion matrix is a method to analyse multiclass classification models. It helps identifying the per class performance of the model. Additionally, the output can be used to measure different metrics such as Recall, F1-score, Precision, Specificity, Accuracy and most importantly AUC-ROC Curve (Narkhede, S., 2018). The Confusion matrix used in this project was implemented using Panda's Crosstab (Data to fish, 2019).

```
def confusion_matrix(label, predicted, normalised = True):
    confusion = pd.crosstab(label, predicted)
    if normalised == True:
        confusion_normalised = confusion / confusion.sum(axis = 1)
        return confusion_normalised.values
    return confusion.values
```

Figure 8: Confusion Matrix Implementation

3 Experiments and Results

3.1 Hardware Specifications

To assess the accuracy and the training time, information about the used hardware components is necessary. Figure 9 shows the specification table:

Software	**Version**
Python	3.7.3
Jupyter Notebook	4.4.0
OS	Microsoft Windows 10 Home V.10.0.18363 Build 18363
Pandas	1.0.3
Numpy	1.18.1
Hypy	2.10.0
Matplotlib	3.1.3
Scipy	1.4.1
Hardware	**Specifications**
CPU	Intel(R) Core (TM) i7-8750H CPU @ 2.20GHz
RAM	16 GB
GPU	NVIDIA GeForce RTX 2080 with Max-Q Design

Figure 9: Hardware Specification Table

3.2 Hyperparameter Analysis and Resulting Set-up

For an appropriate selection of parameters, we need to test possible value combinations in a step by step process. We can achieve this by setting the parameters to constants whilst testing and tuning one parameter at a time.

Firstly, we split the training data to obtain a validation set. The training data included 60,000 rows, with 50,000 rows for training (83.3%) and 10,000 rows as a validation set (16%). The number of neurons within the 5-layer neural network is set to 128, 300, 128, 300, 10 to each respective layer. This is done to minimize computing time whilst not sacrificing performance. The 128 neurons of the input layer are set regarding the number of features. The 10 neurons for the output layer correspond to the number of labels. We set a ReLU activation function (2.2.1) on each hidden layer and a SoftMax (2.2.2) function for the output layer.

3.2.1 Learning Rate and Gamma

We will start by tuning the learning rate and the momentum hyperparameter gamma. The other parameters are set to: Dropout = 10%, Learning Rate = 0.0005, Gamma = 0.6, lambda = 0.001, epoch = 5, batch size = 2048. The initial parameter values result from research and best guesses. Due to the nature of tuning and how momentum and learning rate interact with each other, we re-test the parameters with different values. For our first part we set the gamma to our original constant and test the accuracy for Lr = (0.0001, 0.0005, 0.001, 0.01). As we

test values in a range, we will develop an understanding of performance trends regarding the different values. Figure 10 shows the resulting accuracy values.

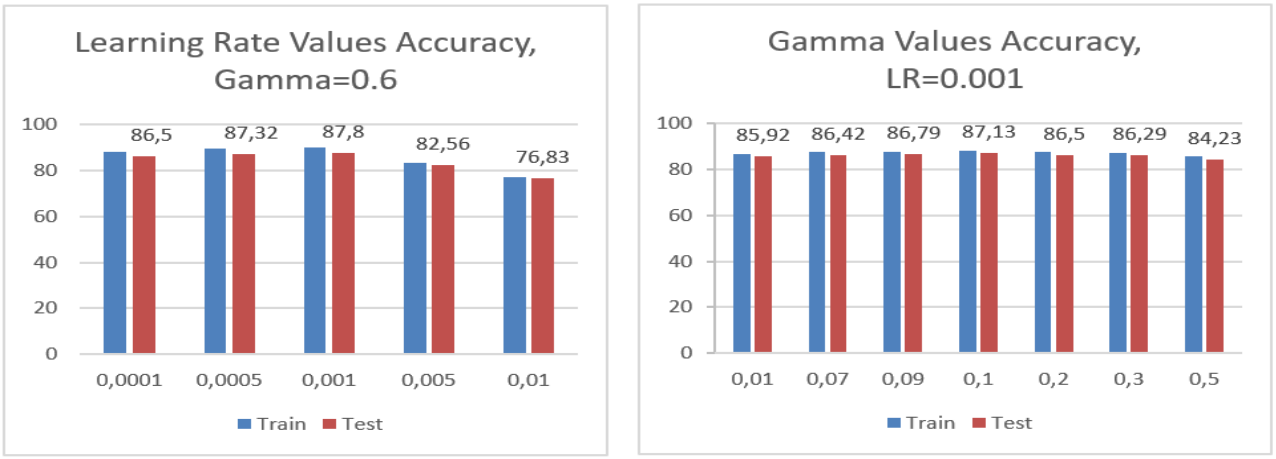


Figure 10: Learning Rate and Gamma

We can observe that the best test accuracy value (87.8%) is generated using a learning rate of 0.001. Afterwards we tested different gamma values. With the set learning rate, we can observe that our original 0.6 is the best gamma value. As visualised in the top right of Figure 10 other combinations show lower test accuracy values than 87.8%.

3.2.2 Lambda

Testing was done for different lambda candidates. The highest two figures were the initial test lambda = 0.001 and lambda = 0.005. Albeit the higher training accuracy while using lambda = 0.005, lambda = 0.001 results in a better test performance. After testing, we settled with lambda = 0.001 with a test accuracy of 87.86%.

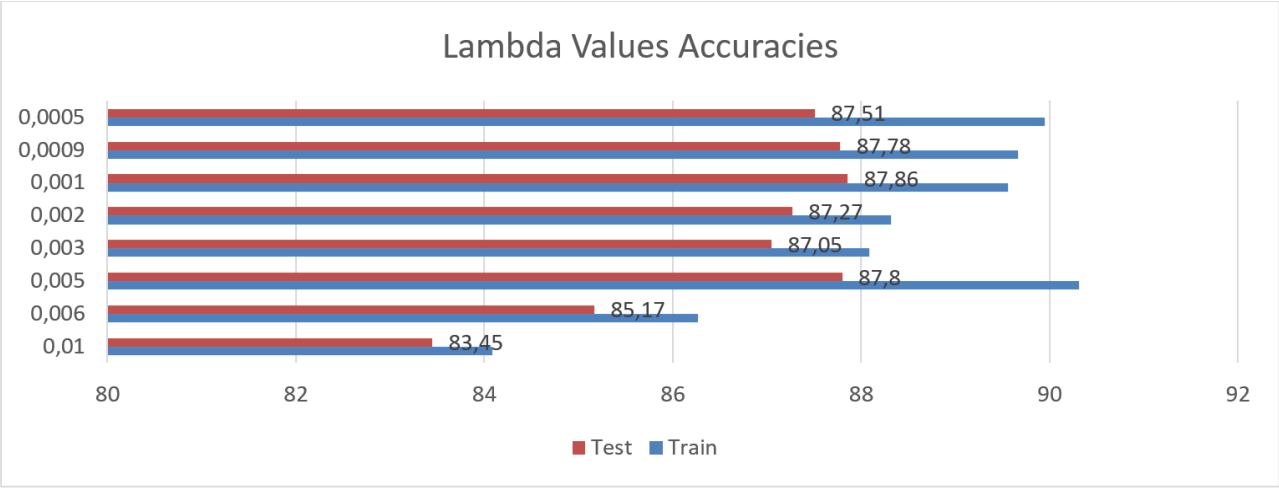


Figure 11: Lambda Accuracy Values

3.2.3 Dropout Parameter

Similar to the other parameters, we seek to find a dropout probability that results in a higher test accuracy. Our initial test values are (7.5%, 10%, 12%, 15%)

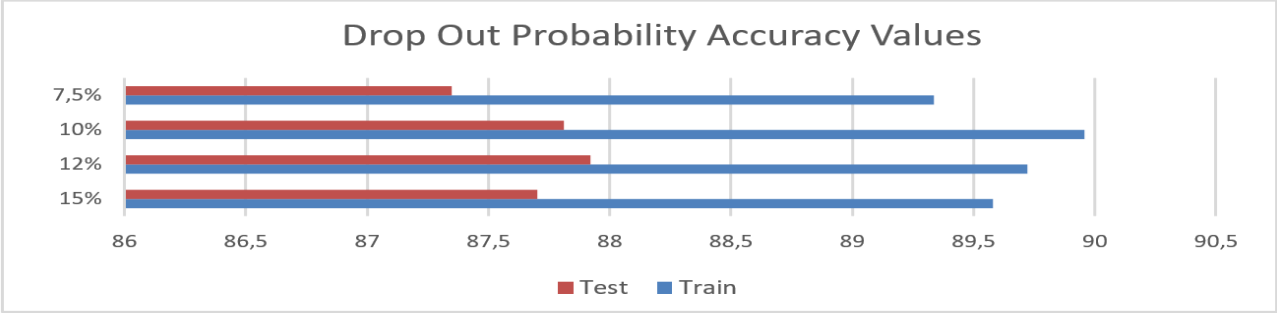


Figure 12: Dropout Probability Accuracy Values

The evaluation revealed that a dropout probability of 12% shows the best test accuracy. Therefore, we discarded our initial value of 10%. The used Dropout parameter is 12%.

3.2.4 Batch Size

Moreover, the accuracy using different batch sizes was evaluated. The test values were = (4096,2048,128,64,32).

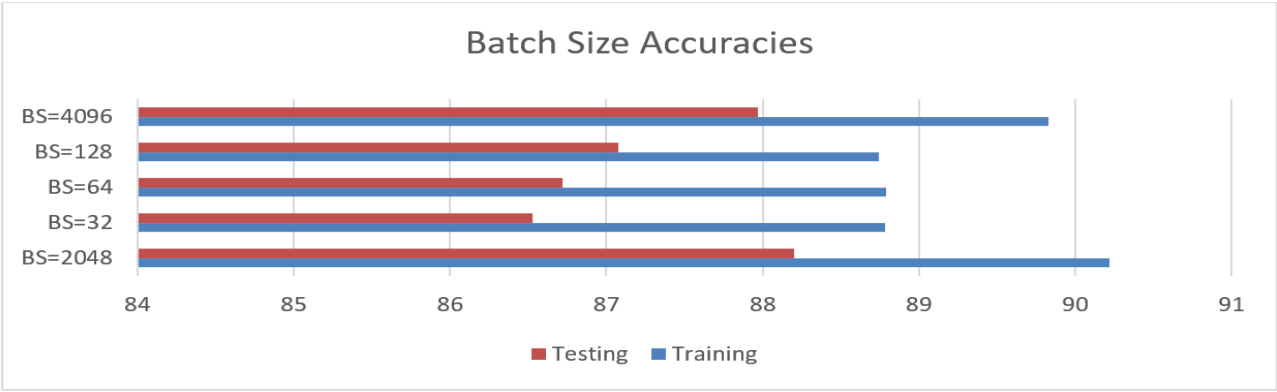


Figure 13: Batch Size Accuracies

In Figure 13 we can see that a batch size of 2048 gives the best training and testing accuracy. Larger batch sizes show similar results, but require more training time.

3.2.5 Epochs

After setting our parameters, we can evaluate different epochs to find an efficient ratio of accuracy and computing time.

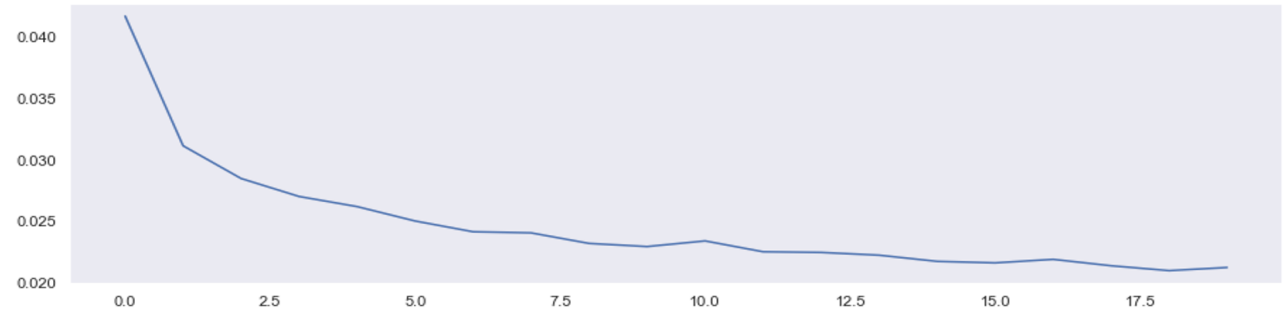


Figure 14: Epochs Accuray

Figure 14 shows the loss value development over 20 iterations. At the 2nd iteration the loss curve adjusts and starts to flatten. After 13 iterations the marginal loss decline per iteration becomes minimal. Therefore, we decided to cutoff the algorithm here, to not inefficiently increase the training time.

3.3 Accuracy Results and Analysis

The following code shows the parameter implementation resulting from section 3:

```
nn = MLP([128,300,128,300,10], [None,'relu','relu','relu','softmax'], dropout = 3, drop_p = 0.12)
Loss = nn.fit(train, encoded_label[:50000], lr=0.001, lambd=0.001, gamma=0.6, epochs=13, minibatch_size =2048 )
```

Figure 15: Hyper Parameter Implementation

After training the model at different parameters it revealed that 13 iterations with a mini-batch size of 2048 was optimal in terms of the ratio between accuracy and training time. Moreover, dropout probability did not seem to make any significant changes at lower values but would hinder results at higher probabilities. Very effective was the implementation of the Momentum function (2.4.5) within the update process. This made the algorithm faster converging and smoothened the graph in Figure 5. To reduce the running time of the algorithm the mini-batch implementation (2.4.1) proofed itself to be an effective tool. Especially during the hyperparameter tuning and analysis phase it simplified the process. Lower batch-sizes were used to evaluate the effect of different parameter combinations faster.

The accuracy was measured on the training set and on the validation set according to the accuracy evaluation formula (2.5.1). Figure 16 shows the defined function used in the model.

```
1 # Define functions for model evaluation
2 def accuracy(label, predicted):
3     return np.sum(predicted == label)/ predicted.shape[0]
```

Figure 16: Accuracy Evaluation Function

The resulting accuracy of the model is **90.83%** on training data and **88.37%** on the validation set. The training time is **13 minutes and 25 seconds**.

Furthermore, we can observe the predicting performance on the test set, for each class in the confusion matrix (Figure 17).

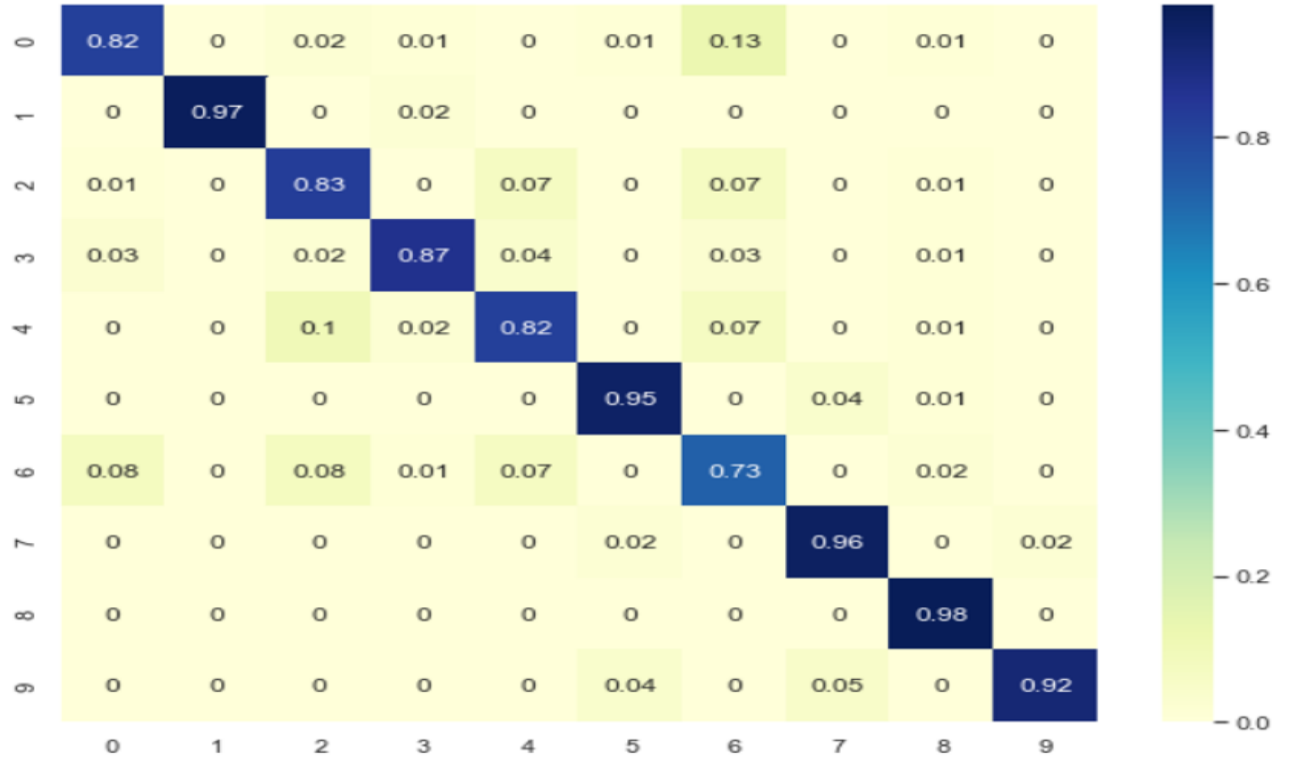


Figure 17: Confusion Matrix

We can observe the main issues whilst learning when predicting class 6. With 73% the accuracy is considerably low. Except for the labels 4 and 2, false predictions of the model can largely explained by label 6. E.g. the comparable bad accuracy of label 0 is almost entirely explained by the confusion with label 6. The labels 1, 5, 7, 8 and 9 show a high accuracy. Excluding class 6, values from 5 to 9 are characterized by a better accuracy than the lower classes.

4 Discussion

The project was challenging in terms of coding and teamwork. Albeit allocating the implementation of the required methods among the three team members resulted in a fast development of a base-model, improving the accuracy afterwards proved to be difficult. It was necessary to make adjustments to the individual functions, so that they better work in the context of the model. Each member had to evaluate and understand the code parts implemented by the others to fully reproduce how the model works in detail. This was time consuming but a great learning experience. The group internalized the basic methods and functions such as forward and backward propagation and Xavier initialization (2.1) of the MLP model. To improve the accuracy, different activation functions (2.2), data processing (2.3) and advanced modules were implemented (2.4). Due to the prohibiton of using deep learning libraries the implementation of the dissected parts required a precise understanding of the theory and the effect on the MLP predictions.

The dissection of the model would be no necessity and to time-consuming in a practical envi-

ronment. However, it enforces a look behind the scenes and is thus appropriate in a university project. The gained knowledge could already be used during the experimental phase in section 3.2, and will be an asset for future work in the field of deep learning and its commercial applications.

5 Conclusions

The results have shown that the implemented MLP is able to predict most data points correctly (3.3). The parameter set-up (3.2) and interaction between the numerous implemented modules (2.4), pre-processing methods (2.3), activation functions (2.2) and basic MLP methods (2.1) revealed as key factors to improve the accuracy. Regarding the performance for the individual labels presented in the confusion matrix (17) the prediction accuracy for the individual classes ranges from 0.73% to 0.98%. To further improve the model accuracy the labels 0, 2, 4 and 6 have to be investigated and additional methods should be considered.

List of Figures

References

Budhiraja, A. Learning less to learn better—dropout in (deep) machine learning., 2016. Online available under https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-7433 accessed on April 28, 2020.

Data to fish. Understanding confusion matrix, 2019.

Godoy, D. Understanding binary cross-entropy / log loss: a visual explanation., 2018. Online available under https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a; accessed on April 28, 2020.

Goel, S. Kaiming he initialization, 2019. Online available under https://medium.com/@shoray.goel/kaiming-he-initialization-a8d9ed0b5899; accessed on April 29, 2020.

Hansen, C. Activation functions explained - gelu, selu, elu, relu and more., 2020. Online available under https://mlfromscratch.com/activation-functions-explained/#/; accessed on April 29, 2020.

Lakhsmanan, S. How, when and why should you normalize/standardize/rescale your data?, 2019. Online available under https://medium.com/@swethalakshmanan14/how-when-and-why-should-you-normalize-standardize-rescale-your-data-3f083def38ff; accessed on April 28, 2020.

Narkhede, S. Understanding confusion matrix, 2018. Online available under https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62; accessed on April 28, 2020.

Oman, S. The softmax function derivative, 2019. Online available under https://aimatters.wordpress.com/2019/06/17/the-softmax-function-derivative/; accessed on April 29, 2020.

G.E.; Krizhevsky A.; Sutskever I.; Salakhutdinov R. Srivastava, N.; Hinton. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning*, 15: 1929–1958, 2004. Online available under http://jmlr.org/papers/volume15/srivastava14a.old/srivastava14a.pdf; accessed on April 21, 2020.

Yadav, D. Categorical encoding using label-encoding and one-hot-encoder, 2020. Online available under https://towardsdatascience.com/categorical-encoding-using-label-encoding-and-one-hot-encoder-911ef77fb5bd; accessed on April 29, 2020.

A Appendix: Instructions to run the Code

Steps to run the MLP classifier (Test Accuracy - 88,37% and Time: appr. 13 minutes)

1) Run the first cell to load all the necessary libraries and set Numpy random seed.

2) Run second cell to define functions for both accuracy and confusion matrix.

3) Run third cell to load the input training, test data, the paths of the train_128.h5, train_label.h5 and test_128.h5.

4) From the label data we obtained the number of levels in the label data. In our case, we have 10 levels/classes.

5) Run cell nine to perform One-hot encoding on the label data. This turns the labels to a format that can be used to train the MLP model.

6) Running cell ten, splits the data into training and validation sets. 50,000 will be used for training while the remaining 10,000 will be used for validation.

7) Cell ten also standardises the training, validation and test data by subtracting the mean from the data and dividing the result by its standard deviation.

8) Run cells 11, 12 and 13 to define the class and functions required to build the MLP model.

9) Run cell 14 to build the model. Five different parameters are required: the number of layers, the neurons and activation functions for each layer as well as the dropout layer and rate.

The following parameter will give the best validation accuracy:

- Dropout rate (dropout): 0.12
- Dropout layer (drop_p): 3
- Input layer: 128; activation function: None
- Hidden layer one: 300; activation function: ReLU
- Hidden layer two: 128; activation function: ReLU
- Hidden layer three: 300; activation function: ReLU
- Output layer: 10; activation function: Softmax

10) Running cell 15 trains the MLP model. For that purpose, the following parameters are required: epochs, learning rate, lambda, gamma and mini-batch size. The model is then trained by passing these parameters to the fit function.

The best parameters are:

- Epochs: 13

- Learning rate: 0.001
- Lambda: 0.001
- Gamma: 0.6
- Mini-batch: 2,048

11) Afterwards, the model predicts the validation and training set to get both training and validation accuracy.

12) The accuracy and confusion matrix of the model is then computed using the comparison of the actual and predicted labels.

13) Run cell 23 to pass the standardized test data to our trained MLP model. This will return the predicted label vector for our test dataset.

14) Run cell 24 to write to the predicted_label.h5 file in the '../Output' folder.