1. Introduction

Multilayer Perceptron (MLP) or multilayer Neural Network is a complex model where the aim is to "successively refine" and to compress the input information to match the desired output as each layer tends to eliminate the unnecessary features of the input information, and keeps and/or transforming the output information along the way through the network. The fully-connected neural network transforms the input information into a form in the final hidden layer, such that it is linearly separable by the output layer (Shams, 2017).

The following study involves a multi-layer neural network built over a ten-class classification problem applied on a dataset containing 60,000 rows of training data and 10,000 rows of testing data. In this study, we implement a neural network model with three hidden layers to train the model for a ten-class classification problem. The model is built without the use of external Deep Learning frameworks and auto-grad tools. Several experiments were conducted by turning on/off different modules or varying the hyper-parameters to study the effect on the performance of the classifier. Based on the results of the experiments, the model with dropout (Hinton, et. al., 2012) of 0.39, Beta1 value (in Adam optimizer) of 0.5, batch-size of 64 was found to be performing well. Weight decay (Krogh and Hertz, 1992) and batch normalization (Ioffe and Szegedy, 2015) were found to be having adverse results on the model's performance for the given dataset. We optimized our model with the use of gradient descent with momentum for updating the parameters and initializing the model's weights using Xavier initialization (Glorot and Bengio, 2010). To regularize the parameters involved, we used bias with the weights, weight decay, early stopping, dropout and batch normalization.

The following report discusses the aims and importance of the study in section 1, the details of data pre-processing and principles of the modules involved in section 2, the experiments and results in section 3, followed by discussion and conclusion in the further sections. The Appendix contains the steps to run the code and specifications of the hardware and software used for this study.

1.1. Aim of the study

Neural networks are the revolutionary algorithms that lay the foundation of deep learning techniques. They allow a machine to be able to learn the input data in a more abstract and composite way than the general machine learning algorithms. The neural networks often considered to be complex architectures are actually a simple build-up of several individual modules built on top of one another. The following study aims at using a given data with unknown semantic information and breaking the presumed complex structure of a neural network into its individual components and therefore presenting a deeper understanding of the functioning of each individual module that is utilized in our prediction (with a validation accuracy of 89.10%) of the given test dataset.

1.2. Importance of the study

An MLP has a large number of free parameters (the weights and biases between interconnected units) giving it the flexibility to fit highly complex data that other models are too simple to fit. Training a complex model like this is challenging, but this complexity ensures that the resultant model generalizes to the examples it is trained on large amounts of data like the one under consideration.

Researchers in the field discovered many empirical and heuristic techniques to improve the performance of neural networks in accuracy and convergence rate. The modules implemented in this study help us to understand the extent of effect of such techniques on the model performance by breaking each of them down to their base architectures. The study builds an MLP model from scratch to implement upon a classification problem. The model's architecture doesn't use any deep learning frameworks, or auto-grad tools and focuses on the overall development of the network's architecture and its regularization and optimization parameters using only the 'numpy' library.

2. Methods

The neural network uses multiple layers through which the data passes repeatedly in forward and backward passes enabling the model to learn its parameters. With each forward pass, the model compares the output generated from its parameters with the target outputs, and with each backward pass, the model regularizes and optimizes its parameters according to the learning rate and gradients. The methodology involved in the model along with its regularization and optimization processes is as below:

2.1. Data pre-processing

The dataset made use of the training data, its labels and test data made available in the form of 'training_data.h5py', 'label.h5py' and 'train_label.h5py' files respectively. The dataset from the files was loaded into a numpy array. The training data consists of 60,000 examples, divided equally among 10 classes. The data did not have any missing values. The following graph (Figure 1) shows the distribution of data for each class.

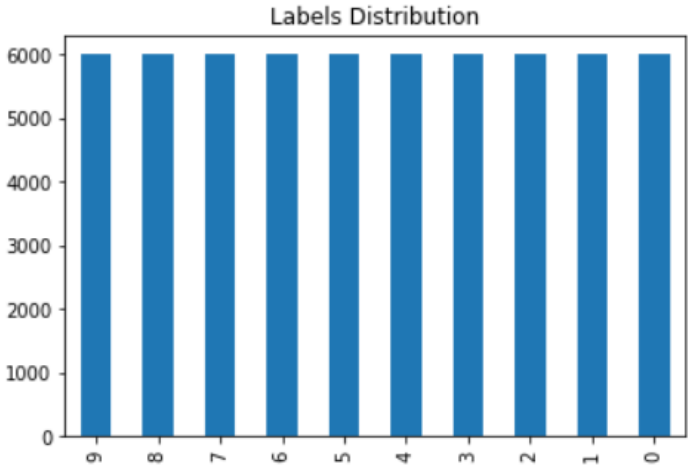


Figure 1: Distribution of data among different classes

2.1.1. Shuffling

Shuffling data serves the purpose of reducing variance and making sure that models remain general (not biased) and over-fit less, especially if the data is sorted by their class. We shuffle to ensure that the training/validation/test sets are representative of the overall distribution of the data. Shuffling reduces the risk of creation of batches not representing the overall dataset. It ensures that the data after each epoch will not be "stuck" with too many bad batches.

2.1.2. Train and Validation Split

The model is split into training and validation sets by slicing the datasets after shuffling. Training set is utilized to train the model, whereas the validation is used to test its performance so as to perform the hyper-parameter tuning (Bronshtein, 2017). The model occasionally sees this data, but never does actually 'learn' from it. For our experiment, we have considered the validation dataset of 6000 examples and training dataset with 54,000 examples.

2.1.3. Feature Engineering

2.1.3.1. Standardization

The purpose of Standardization is often to attain Gaussian normal distribution for the given data. This avoids biased results due to any kind of skewness in the data and we can expect unbiased results. This means if inputs are not standardized (mean of 0 and standard deviation of 1), then the importance of each input could not be equally distributed, thereby making naturally large values to become dominant than smaller values during ANN training (Bhandari, 2020).

2.1.3.2. One-Hot Encoding

Each label was one hot encoded into a dimension 10 vector of 1s and 0s. This was used primarily to vectorise the loss and gradient calculations and speed up the computation process.

The resulting training and validation label set are matrices of shapes (54000, 10) and (6000, 10) respectively.

2.2 Modules involved and their principles

2.2.1. Activation Functions

An activation function is the function in a neural network neuron that delivers an output based on some functions applied over the input parameters. It must be non-linear, continuous and differentiable at almost all the points. In addition to this, it must possess the property of "monotonicity" in order to avoid additional local extrema in error surface. The choice of activation function is done based on the type of data and the problem being dealt with (Sharma, 2017). The most commonly used activation functions (sigmoid function, hyperbolic tangent (tanh) function, Rectified Linear Unit (ReLU) function, Softmax Function) are listed below:

- **Sigmoid/Logistic:** The function yields outputs within the range of (0,1). It is used for predicting probabilities. The computation of exponential function is expensive and there is a vanishing gradient function. The graph and the mathematical function is given below:

$$\text{Sigmoid Function } \sigma(x) = \frac{1}{1+e^{-x}}$$

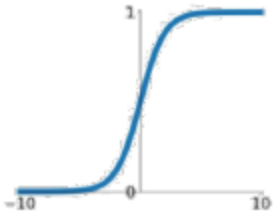


Figure 2: Sigmoid activation function (Xu, 2020)

- **Tanh:** The function yields the output values within the range of (-1,1). It is generally preferred over sigmoid, since it allows easier optimization due to steeper derivatives. The function has a vanishing gradient problem, so earlier layers of the network learn slowly due to small updates in the parameters. The mathematical representation and graph are:

$$\text{Tanh Function } tanh(x) = \frac{2}{1+e^{-2x}} - 1$$

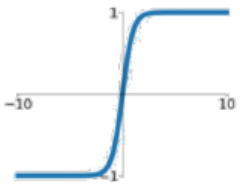
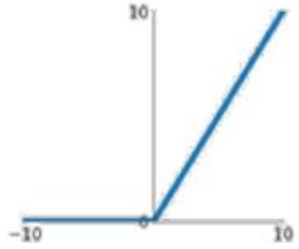


Figure 3: Tanh activation function (Xu, 2020)

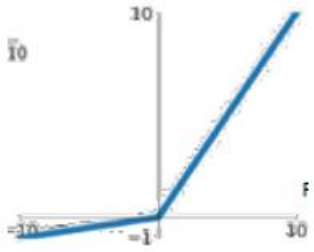
- ReLU: It forces the output of the layer to be sparse. Sparseness makes a neural network to be computationally efficient in processing the parameters. "Dead ReLU" issue may arise as some neurons will be dead. To resolve this issue, learning rate must not be very large

and weights must be properly initialized which is solved by the use of Leaky ReLU function. The mathematical representations and graphs are:



ReLU

$$Relu(x) = max(0, x)$$



Leaky ReLU

$$LeakyRelu(x) = max(ax, x), where\ a = constant$$

Figure 4: ReLU and Leaky ReLU activation functions (Xu, 2020)

- **Softmax:** It transforms the elements of vectors to be between 0 and 1, so it can be interpreted as a probability vector. This study uses Softmax function as the final activation to compute cross entropy. Mathematical representation is:

$$\widehat{\boldsymbol{P}}(\boldsymbol{class_k}|\mathrm{x}) = \boldsymbol{z_k} = \frac{\boldsymbol{e^{net_k}}}{\sum_{\boldsymbol{i=1}}^{\boldsymbol{K}} \boldsymbol{e^{net_i}}}$$

2.2.2. Optimization methods

The various techniques employed in the optimization of neural networks are as follows:

2.2.2.1. Batch gradient descent (BGD)

In this method, all the training data is taken into consideration with each time step. The average of the gradients of all the training examples is taken to iterate through backpropagation and parameter updation process with each epoch. Since the entire dataset is scanned with each epoch, this technique is more suitable for the small datasets.

2.2.2.2. Stochastic Gradient Descent (SGD)

In large datasets, to avoid the computational time limitation of the BGD method, the SGD considers just one example in each time step. Based on the gradient of each iteration (with one example), the model back-propagates and updates its parameters.

2.2.2.3. Mini-batch gradient descent

Although effective for large datasets, the SGD poses difficulties in vectorization, hence it may slow down the computation process. Hence, mini-batch gradient descent comes into picture which combines the advantages of both BGD and SGD. The method uses a batch of fixed training examples, thus saving up memory and being computationally efficient at the same time (Mainkar, 2018).

2.2.2.4. Momentum

Gradient Descent Techniques, especially SGD, has a difficulty in navigating through the “ravines”, the areas where the gradient is steeper than at other points. To overcome this issue, the concept of momentum was introduced. Momentum dampens the oscillations and accelerates the vectors of gradient descent in the relevant direction. The working principle underlying “Momentum” is to increase the dimensions whose gradients point in the same direction and reduce updates for the dimensions whose gradients change direction. It is a two-step process where the model first computes the gradient at the current location and thereafter jumps in the direction of the updated accumulated gradient.

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

$$\theta_t = \theta_{t-1} - v_t$$

where,

v_{t-1} = exponentially weighted average of past gradients

v_t = updated exponential weighted average at a time step

$\nabla_\theta J(\theta)$ = gradient of cost function with respect to weight vectors of current layer.

θ_{t-1} = weight vector in past iteration

θ_t = updated weight vector

η = Learning Rate

γ = momentum term, which is usually 0.9.

2.2.2.5. Adam

Adaptive learning rate methods evolved to overcome limitations of Gradient Descent and Momentum when it came to different levels of update for different features for sparse data with features having varying frequencies. “Adam” is a popular technique that adapts learning rate for different parameters. It uses squared gradients to scale the learning rate and takes the moving average of gradients with momentum. Since $v_t$and $m_t$vectors are zero initialized, they’re biased towards zero during initial steps or when the decay rate is small. Hence, to correct this Adam-bias, $\hat{m}_t$ and $\hat{v}_t$are computed and optimization is done using those vectors (Kathuria, 2018). The mathematical principle working behind it is shown below.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \ \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Where v_t = Exponentially decaying average of past square gradients.

m_t = Exponentially decaying average of past gradients

θ_t = Weight vector in t-th iteration.

θ_{t+1} = Updated weight vector.

ϵ = epsilon value (very small) to avoid zero in denominator

β_1^t, β_2^t = Hyperparameters to be trained.

η = Learning rate

2.2.3. Weight Initialization

Weight initialization is an important aspect of MLP. Weights may be randomly initialized to break the symmetry while updating parameters. However, the weights with respect to learning rate must neither be too small nor too large. Too small weights may lead to vanishing gradients whereas too large weights may cause the gradients to explode. Hence, while assigning weights it must be ensured that the mean of activations should be zero and the variance of all the activations should remain the same across every layer of MLP. For this purpose, Xavier Initialization is used to assign the weights from the Xavier uniform distribution. The range of the distribution depends on the choice of activation function. The following equations shows the weight initializations for two very popular methods, Xavier and He:

Table 1: Popular weight initialization methods

Weight initialization method	Normal Distribution	Uniform Distribution
Xavier (tanh activation assumed)	$N(0, \frac{2}{n^{[l-1]} + n^{[l]}})$	$U(-\frac{\sqrt{6}}{\sqrt{n^{[l-1]} + n^{[l]}}}, \frac{\sqrt{6}}{\sqrt{n^{[l-1]} + n^{[l]}}})$
He (ReLU activation assumed)	$N(0, \frac{2}{n})$	$U(-\sqrt{\frac{6}{n}}, \sqrt{\frac{6}{n}})$

(Xu, 2020)

2.2.4. Regularization techniques

Regularization techniques help overcome the overfitting issues. Some general ideas to prevent overfitting are training on a large dataset, eliminating useless features etc. Other techniques used are data augmentation, weight decay, early stopping, dropout. This study uses weight decay and dropout as regularization methods:

2.2.4.1. Weight decay

It tends to limit the growth of weights within a neural network by adding a term to the loss function so as to penalize large weights. There exist two viewpoints to get the value of weight decay. The researcher's viewpoint is more from a theoretical perspective whereas the engineer's viewpoint is based on experimental observations:

- Researcher's view: $\theta \leftarrow \theta - \eta \nabla \hat{L}_R(\theta)$
- Engineer's view: $\theta \leftarrow \theta * 0.98$

2.2.4.2. Dropout

The main idea behind Dropout is that each unit should work with a random sample of other units. While training each unit is retained with a probability 'p' (equal to value set for dropout). During testing, the network is used as a whole and weights are scaled-down by the factor 'p'. This method deals with overfitting by combining predictions of many large neural nets at test time. Dropout is scale-free as there is no penalization of large weights and is also invariant to parameter scaling making it robust.

2.2.5. Cross-entropy loss

Loss functions (or cost function) are used to compare the goodness of a model's predictions over a given set of parameters. The gradient of the loss function curve is used for updating the parameters of a model (Machine Learning Glossary, 2017). Cross Entropy loss (log-likelihood method for error estimate) is a convex cost function, thereby making it easier to achieve the global minimum point (minimum loss), hence this has been opted as the cost function in this study. It is used to measure the performance of classifiers whose output is a probability value between 0 and 1. The loss increases as the predicted probability diverges from the actual label. Mathematically, it is represented as:

$$-\sum_{c=1}^{M} y_{o,c} log(p_{o,c})$$

where,

M = number of classes

log = the natural log

y = binary indicator (0 or 1) if class label

c = the actual classification for observation o

p = predicted probability observation for o

o = observation belonging to class c

2.2.6. Batch Normalization

Batch Normalization (BN) can be viewed as an additional layer for MLP that performs whitening on intermediate layers by reducing the internal covariate shift, that is, changing the distribution of the activation function of a component. Batch Normalization reduces the effects of exploding and vanishing gradients as it makes the parameters to be roughly normally distributed. A new layer is added so that the gradient is able to detect normalization and make adjustments as per the requirements. BN reduces the training time and demand for regularization. It also allows higher learning rates thereby accelerating the learning process. However, BN is also complicated,

computationally expensive and gives noisy estimation for smaller batch sizes. BN is generally inactive in a model's setting, being sometimes active a bit. The batch normalization in our model was slightly different from the usual implementation. The forward step details are as follows:

- Get pre-activations (i.e. the matrix dot product)
- Find mean, variance of pre-activations and standardize
- Feed these through to activation function

(Zakka, 2016)

2.3. Model Architecture

To implement the model's architecture, three classes were defined and were packaged with relevant functions forming the working foundation for each of them:

- Activation: This class contains the implementations of the functions for different types of activation functions (discussed in section 2.2.1.) and their respective derivatives required while forward and back propagation. The object when initialized takes the name of the activation function as an argument.
- Hidden Layer: This class initializes layers as per the given arguments and wraps within it the functions for batch normalization and dropout. When initializing the object of this class, we pass the number of neurons (n_in, n_out) in the previous and current layers respectively, activation function of the previous layer and the current layer and turn on/off the option for batch normalization and dropout. The created object has the weights, biases, velocity vectors initialized.
- MLP: This class represents the network as a whole encapsulated within it the functions for forward propagation, backpropagation, cost computation using cross entropy loss, updation of parameters, model training for given no. of epochs and prediction function. The predict function is a call to forward propagation function with training parameters set to 'False' as batch normalization and dropout needs to be turned off while testing/prediction. The object is initialized by passing a list layers with no. of neurons in each layer in the order from input to output layer, the list of activation functions for each hidden layer (input layer has no activation and output layer has Softmax activation function), and a list of boolean values to signify for which layer batch normalization option needs to be enabled.

2.3.1. Training the Model

The function fit() of the MLP class is used to train the MLP model. It takes as input data of the shape (no. of examples, the no. of features), train labels (no. of examples, no. of classes), the no. of epochs, batch size and optimization option. The training process is summarized in the following steps:

- Mini-Batching: For each epoch, the input data is sliced into batches loss and gradient update is done after each batch.

- Forward Propagation: For each hidden layer, pre-activations are computed and then fed through activation. Forward operations to compute outputs are performed by feeding the inputs through to layers sequentially. It is a composition of matrix multiplications and activation functions. Dropout and batch-normalization if enabled are also performed in this step.
- Loss Computation (Cross Entropy Loss): The predicted value after forward propagation (output of Softmax function) is used to compute the loss against the one-hot encoded matrix of actual target value for each batch. The loss is averaged over the batch size.
- Backpropagation: Calculate the losses with respect to parameters of each batch. Using this loss, the gradients of the loss with respect to parameters (weights and biases) are computed, one by one through the layers in reversed order. Updation of parameters happens throughout the layers.
- Updation of Parameters: This step involves the updation of the weights and bias vectors for each layer based on the choice of optimizer. If 'momentum' is chosen, then the velocity vectors are updated first followed by updation of weights as per the working principle of momentum. If 'adam' is chosen, then additionally defined beta1 and beta2 values are supplied to the adam_update() function to update the learning rate and parameters accordingly. If 'None' is chosen, then the default optimization SGD performs the updation of parameters.

2.4. Evaluation

2.4.1. Confusion Matrix

A confusion matrix is a summary of prediction results on a classification problem. The number of correct and incorrect predictions are summarized with count values and broken down by each class. This is the key to the confusion matrix. The confusion matrix shows the ways in which your classification model is confused when it makes predictions. It gives insights of not only into the errors being made by the classifier but more importantly the types of errors that are being made (Brownlee, 2016).

- To construct a confusion matrix, the following values were computed arranged as explained below:
- Calculate the number of correct predictions for each class.
- Calculate the number of incorrect predictions for each class, organized by the class that was predicted.
- Expected down the side: Each row of the matrix corresponds to a predicted class.
- Predicted across the top: Each column of the matrix corresponds to an actual class.
- The total number of correct predictions for a class go into the expected row for that class value and the predicted column for that class value.
- In the same way, the total number of incorrect predictions for a class go into the expected row for that class value and the predicted column for that class value.

True Positives (TP): These are the correctly predicted positive values.

True Negatives (TN): These are the correctly predicted negative values.

False Positives (FP): When actual class is no and predicted class is yes.

False Negatives (FN): When actual class is yes but predicted class in no.

Accuracy: Accuracy is the most intuitive performance measure and it is simply a ratio of correctly predicted observation to the total observations. As the current dataset is a symmetric/balanced dataset, hence this measure helps to evaluate the performance of the classifier quite well.

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

Precision: Precision is the ratio of correctly predicted positive observations to the total predicted positive observations. High precision relates to the low false positive rate.

$$Precision = \frac{TP}{TP + FP}$$

Recall (Sensitivity): Recall is the ratio of correctly predicted positive observations to all observations in the actual class.

$$Recall = \frac{TP}{TP + FN}$$

F1 score: F1 Score is the weighted average of Precision and Recall. Therefore, this score takes both false positives and false negatives into account. Intuitively it is not as easy to understand as accuracy, but F1 is usually more useful than accuracy, especially if you have an uneven class distribution. Accuracy works best if false positives and false negatives have similar cost. If the cost of false positives and false negatives are very different, it's better to look at both Precision and Recall.

$$F1\ Score\ = \frac{2 * Recall\ *\ Precision}{Recall\ +\ Precision}$$

(Joshi, 2016)

2.4.2. Loss and Accuracy graphs

The graphs for training loss against the number of epochs helps to ensure that the training loss is decreasing over time and not increasing. The epochs versus accuracy graph in the following section shows the training and validation accuracy over the time (number of epochs).

3. Experiments and results

3.1. Baseline Model

The baseline model achieved a training loss of 0.3, training accuracy of 0.8938, validation accuracy of 0.8830 after 20 epochs. The following table (Table 2) shows the specifications of the baseline model.

Table 2: Specifications of the baseline model

Parameter	**Detail**
Number of layers	5 (3 hidden)
Number of neurons	[128,64,32,16,10]
Activations (in each layer)	[None,'tanh','tanh','relu','softmax']
Batch Norms	False
Dropouts	False
Optimizer	Adam
beta1	0.9
beta2	0.999
Learning rate	0.01
Weight decay	0
Batch size	128
Time taken	26.9 seconds

The below figure (Figure 5) shows the loss and accuracy graphs of the baseline model:

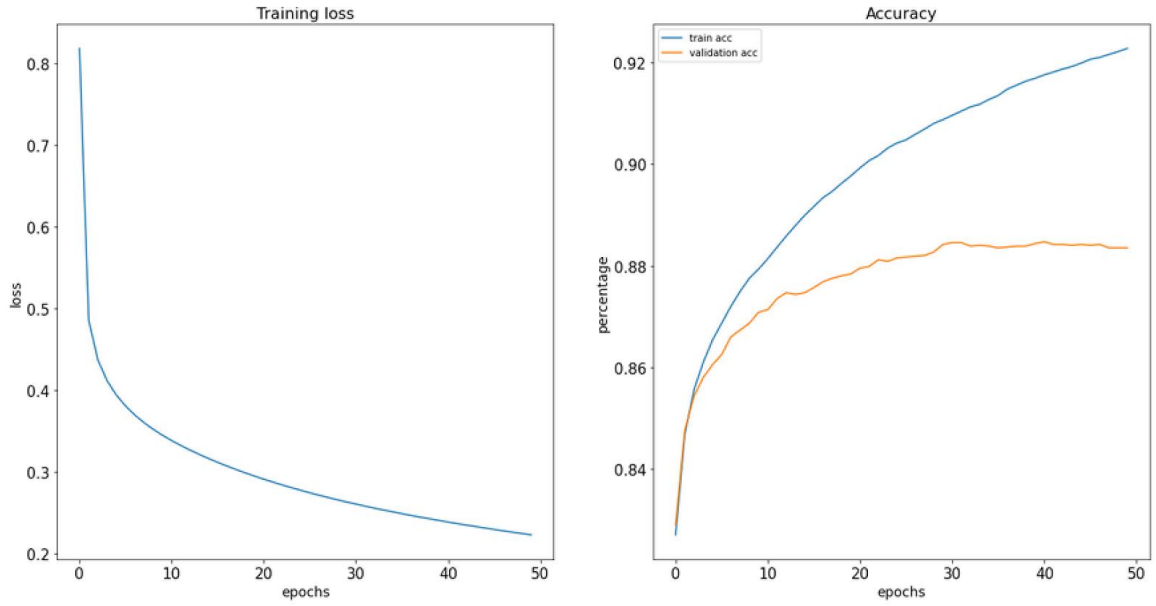


Figure 5: Loss and accuracy graphs of baseline model

3.2. Confusion matrix (Precision, Recall and F1 score)

The below table (Table 3) highlights the confusion matrix for the predictions The figure on the left displays the number of examples in each actual class predicted as different classes. The figure on the right displays the values of precision, recall, and f1_score against each class. Support is the number of examples contained by the class.

Table 3: Confusion matrix and precision, recall, f1-scores of each class

actual / predicted	0	1	2	3	4	5	6	7	8	9
0	4731	7	58	77	9	0	582	0	15	1
1	5	5370	1	30	4	1	8	0	1	0
2	87	4	4640	26	266	1	335	0	14	0
3	95	47	35	5043	112	1	81	0	14	0
4	10	4	377	147	4793	0	264	0	13	0
5	1	0	2	1	0	5312	0	21	2	16
6	455	7	305	62	206	2	4088	0	27	0
7	0	0	0	0	0	60	0	5273	5	123
8	28	2	16	10	8	3	37	4	5260	1
9	1	0	0	1	0	9	1	88	0	5254

	precision	recall	f1_score	support
0	0.8740	0.8633	0.8686	5413
1	0.9870	0.9908	0.9889	5441
2	0.8539	0.8636	0.8587	5434
3	0.9344	0.9291	0.9317	5397
4	0.8879	0.8547	0.8710	5398
5	0.9857	0.9920	0.9888	5389
6	0.7576	0.7935	0.7751	5396
7	0.9790	0.9656	0.9723	5386
8	0.9830	0.9797	0.9813	5351
9	0.9739	0.9813	0.9776	5395

The overall f1-score of the model is the average of the f1-scores of each individual class, which is 0.9214. The results show that the baseline model considered is pretty good at classification as it tries to predict each class. The model tends to predict classes 1,3,5,7,8,9 pretty well. However, it isn't able to distinguish much between the classes 0, 2, 6 and 4; the most apt reason being resemblance or similarity between features of these classes.

3.3. Hyper-parameter tuning

The baseline model was tuned across 5 hyper-parameters; dropout percentage for the first 2 layers, Beta1 (for Adam), weight decay percentage, batch size and presence of batch norm in the first layer.

When tuning each hyper-parameter, one hyperparameter was changed, other parameters of the baseline model were kept the same. The effectiveness of each particular hyper-parameter was assessed by its validation accuracy after a maximum of 20 epochs. Only a single training-validation split was used. While this considerably speeds up the process, it may cause distortions in accuracy due to the variance in data and weight/bias initializations.

Early stopping was implemented based on validation accuracy; if the validation accuracy of consecutive epochs decreases by 5% from the previous accuracy or the loss returns NaN, then the model will stop training. In the experiments below however, this early stopping criteria was never met.

3.3.1. Dropout

The table below (Table 4) shows the validation accuracy when tuning the dropout percentage of the first 2 hidden layers. The dropout range was 10 equally spaced points from 0 to 0.5, with the optimal dropout percentage being 0.39. From the validation accuracy column, dropout does not have much of an effect on accuracy.

Table 4: Hyperparameter tuning with dropout

	Dropout	Val accuracy	time (s)	optimal epochs
7	0.3911	0.8777	34.4246	20
3	0.1733	0.8768	33.3630	20
6	0.3367	0.8758	32.2731	20
0	0.0100	0.8732	28.4298	20
4	0.2278	0.8725	29.9957	20
5	0.2822	0.8722	29.1063	20
1	0.0644	0.8718	32.7733	20
9	0.5000	0.8718	32.1479	20
2	0.1189	0.8713	33.6318	20
8	0.4456	0.8700	34.4588	20

3.3.2. Beta1 (Adam optimizer)

The table below (Table 5) shows the validation accuracy when tuning beta1 of the Adam optimizer. The range was from 10 equally spaced points from 0.5 to 0.9, with the optimal value being 0.5.

Table 5: Hyperparameter tuning with Beta1 (Adam optimizer)

	betas	Val accuracy	time (s)	optimal epochs
0	0.5000	0.8910	36.4605	20
2	0.5889	0.8870	36.1799	20
4	0.6778	0.8867	32.3825	20
1	0.5444	0.8858	33.4271	20
3	0.6333	0.8858	34.4735	20
5	0.7222	0.8852	34.4209	20
7	0.8111	0.8840	32.3539	20
6	0.7667	0.8838	37.9875	20
8	0.8556	0.8778	33.8024	20
9	0.9000	0.8680	32.9444	20

3.3.3. Weight Decay

The table below (Table 6) shows the validation accuracy when tuning the weight decay percentage. The range was 10 equally spaced points from 0 to 0.5, with the optimal value being 0.0 (no weight decay). It appears that any form of weight decay produces bad results in terms of

validation accuracy. The reason of weight decay zero performing quite well may be due to the fact that it is used in combination with other optimization techniques and loses its effect. Generally, very small weight decays are preferred over large decays in order to fit the model.

Table 6: Hyperparameter tuning with weight decay

	decays	Val accuracy	time (s)	optimal epochs
0	0.0000	0.8743	30.5239	20
1	0.0556	0.1948	28.8135	20
2	0.1111	0.1007	44.4545	20
3	0.1667	0.1007	48.7856	20
4	0.2222	0.1007	43.1952	20
5	0.2778	0.1007	41.7992	20
6	0.3333	0.1007	45.3039	20
7	0.3889	0.1007	48.2214	20
8	0.4444	0.1007	49.5604	20
9	0.5000	0.1007	46.7139	20

3.3.4. Batch Size

The table below (Table 7) shows the validation accuracy when tuning batch size of the training data. The batch size list is [24, 64, 128, 256, 512, 1024], with the optimal value being 64. Very small batch size tends to increase the training time per epoch whereas very large batch-size consumes memory. However, a smaller batch size produces more variance when calculating gradients, potentially leading to a less smooth convergence.

Table 7: Hyperparameter tuning with batch sizes

	batch_sizes	Val accuracy	time (s)	optimal epochs
1	64	0.8757	43.2435	20
3	256	0.8753	29.3801	20
5	1024	0.8745	22.6388	20
4	512	0.8733	23.2913	20
2	128	0.8718	31.8110	20
0	24	0.8638	58.6757	20

3.3.5. Batch Norm

The table below (Table 8) shows validation accuracy when tuning whether there is a batch normalization operation in the first hidden layer. It appears that no batch norm performs better on the data. It appears that no batch normalization performs better on the data. The most probable reason is that because the input data is already standardized, further batch norm operations scale down important differences in data which are used discriminate classes.

Table 8: Hyperparameter tuning with Batch Normalization

	BN	Val accuracy	time (s)	optimal epochs
1	False	0.8712	27.2886	20
0	True	0.8647	29.7422	20

3.3.6. Summary of results

The graph below (Figure 12) compares the validation accuracy of the models with optimal hyper-parameters. The green line appears to predict the best; which is the baseline model with hyperparameter tuned for Beta1 = 0.5. The accuracy yielded by this setting is 89.10%.

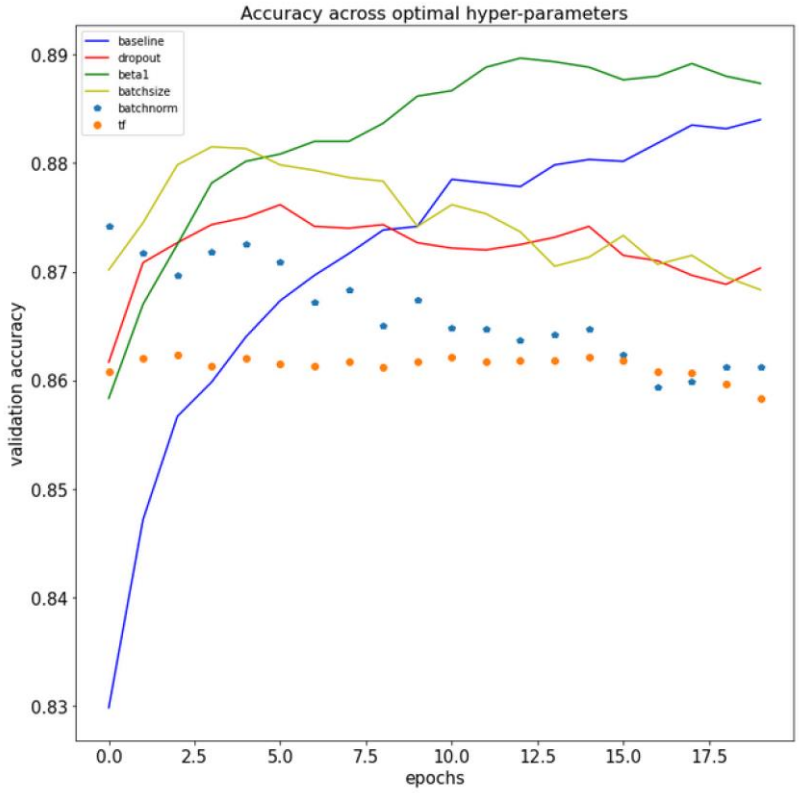


Figure 6: Effects of Hyperparameter tuning over validation accuracy

3.4. Tensorflow Hyperparameter Tuning

The neural network model involved in this study was also trained in Tensorflow using HParams dashboard, keeping the settings similar to the baseline model. It details the top 10 models by validation accuracy for various combinations of hyper-parameters:

Table 9: Specifications of hyperparameter tuning with Tensorflow

Parameter	**Range/Choices**
Num units (1st hidden layer)	[32, 64, 96]
Batch size	[24, 64, 128, 256, 512, 1024]
Dropout	Interval (0, 0.5)
Optimizer	[Adam, SGD]
Learning Rate	[0.001, 0.005, 0.1]
Beta1	(0.5, 0.9)
Batch Normalization (1st layer)	[Yes, No]

The table below (Table 10) is an excerpt from Tensorboard. The results were from 30 simulations of a random subset of the hyper-parameters. The best model giving an accuracy of 88.32% is as follows:

Number of units (1st hidden layer): 96, Batch size: 128, Dropout (1st hidden layer): 0.094, Optimizer: Adam, Learning Rate: 0.005, Beta1 (Adam): 0.71274, Batch Normalization in first layer: no.

The top 10 models produced in Tensorflow broadly agree with our own simulations; the optimal range of beta1 is around 0.5-0.7 and no batch normalization in the first layer produces better results. A lower dropout percentage and higher batch size however seems to produce a better result, which was not seen in our simulations.

Table 10: Hyperparameter tuning with TensorFlow

Trial ID	Show Metrics	num_units	batch_size	dropout	optimizer	lr	beta1	bn	Accuracy
b	[	96.000	128.00	0.094429	adam	0.0050000	0.71274	No	0.88317
3	[	96.000	256.00	0.13312	adam	0.0050000	0.83383	Yes	0.88300
4	[	32.000	512.00	0.15988	adam	0.0050000	0.54839	No	0.87850
0	[	32.000	1024.0	0.044092	adam	0.0050000	0.70880	No	0.87667
e	[	64.000	1024.0	0.015815	adam	0.010000	0.63763	No	0.87417
d	[	32.000	1024.0	0.031642	adam	0.0050000	0.75944	No	0.87367
d	[	32.000	1024.0	0.16802	adam	0.010000	0.74185	Yes	0.87350
c	[	64.000	512.00	0.048421	adam	0.0010000	0.90252	No	0.87333
8	[	64.000	512.00	0.12603	adam	0.0010000	0.50794	Yes	0.87233
d	[	96.000	1024.0	0.052011	adam	0.0010000	0.83257	No	0.87233

3.5 The Optimal Model

The optimal model was the baseline model with beta1 set to 0.5. This was trained for 20 epochs on the full train dataset (60000 examples) and used to generate the test set predictions.

4. Discussion

Writing a neural network from scratch is a great way to understand how all the individual modules and functions of deep learning work together. As the given data didn't contain any semantic information of the features involved, the focus of the assignment was on the familiarization of different neural network modules.

There were many problems experienced during training of the neural network, the most significant being exploding/vanishing gradients. This occurs when the gradients of the parameters become too large or small. In our case, using the Relu activation in the first 2 hidden layers led to exploding gradients within the first 10 epochs. This was mostly caused by the sparseness of activations; as training progressed, zero-valued elements increased while non-zero valued elements became larger and eventually created overflows. Exploding gradients occurred faster using SGD with high learning rates.

Using the Adam optimizer as opposed to the vanilla SGD decreased the likelihood of exploding gradients, as Adam adjusts the learning rate depending on the size of gradients. Weight decay and dropout also reduced the chance of exploding gradients. Future models can be improved by techniques such as gradient clipping or different initialisation methods, which are designed to reduce the risk of gradient explosion.

Another potential problem with our model was with regards to hyper-parameter tuning to find the optimal model. As our evaluation was done only with one training-validation split, the validation accuracy of the hyper-parameters was dependent upon one specific subset of the data. Given the sensitivity of neural networks to perturbations in parameter initialisations and input data, this

means that minor differences in validation accuracy for different hyper-parameters cannot be considered as significant. Large differences however are still noteworthy, such as the drops in accuracy when increasing weight decay percentage. In the future, this could be improved by cross validation and multiple splits of training data for training/validation.

5. Conclusion

Our baseline model produces good prediction accuracies across all classes. This is evident from the confusion matrix, which shows good f1 scores for all classes, with the exception of class 6. The loss diagram of loss against epochs also shows that the model is learning the patterns of the data well throughout training. One of the main reasons for the good results is no class imbalance; the training data is evenly distributed across all 10 classes, meaning that the model theoretically can learn the patterns of all classes well.

Our hyper-parameter tuning also provided some good insights into the effect of various modules on model accuracy. Tuning done on our own model demonstrated the effect of individual modules on accuracy whilst keeping other parameters constant. Whilst hyper-parameters may have a confounding effect (e.g. higher dropout may have better effect when learning rate is high) which cannot be examined in isolation, this was examined by using Tensorflow by tuning all hyper-parameters at once. It is quite clear from our experiments that batch norm and negative effect and the optimal value for beta1 of the ADAM optimizer is from 0.5-0.7.

The optimal model used for test predictions was then selected from the results of hyper-parameter tuning.

6. References

Bhandari, A 2020, 'Feature Scaling for Machine Learning: Understanding the Difference Between Normalization vs. Standardization', *Analytics Vidhya*, web blog post, 3 April, viewed 30 April 2020, https://www.analyticsvidhya.com/blog/2020/04/feature-scaling-machine-learning-normalization-standardization/

Bronshtein, A 2017, 'Train/Test split and Cross Validation in Python', *Towards Data Science*, web blog post, 18 May, viewed 20 April 2020, https://towardsdatascience.com/train-test-split-and-cross-validation-in-python-80b61beca4b6

Brownlee, J 2016, 'What is a Confusion Matrix in Machine Learning', *Machine Learning Mastery*, web blog post, 18 November, viewed 30 April 2020, https://machinelearningmastery.com/confusion-matrix-machine-learning/

Glorot, X, & Bengio, Y 2010, 'Understanding the difficulty of training deep feedforward neural networks', In *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249-256).

Hinton, GE, Srivastava, N, Krizhevsky, A, Sutskever, I, & Salakhutdinov, RR 2012, 'Improving neural networks by preventing co-adaptation of feature detectors', *arXiv preprint arXiv:1207.0580.*