COMP5329
DEEP LEARNING

Assignment 1

Multi-class classification using multilayer Neural Network

Xiaodong Zhao (490373431),
Syed Mehedi Hasan (480255897),
Mudassar Riaz (460238922)

TABLE OF CONTENTS

COMP5329 – DEEP LEARNING – ASSIGNMENT 1

Xiaodong Zhao (490373431), Syed Mehedi Hasan (480255897), Mudassar Riaz (460238922)

1 ABSTRACT

Deep learning is a field which solves complex data modelling problems by breaking them into simpler hierarchical problems. These problems are divided into simpler interconnected layers, each trying to solve the bigger problem by using outputs of previous layer. The most basic unit of a deep neural network is called a neuron, hence the name. These neurons are activated by simple functions, called activation functions. The whole concept lies on the understanding driven by Universal Approximation Theorem i.e., a single hidden layer neural is capable of approximating input to any continuous function, under imposed conditions.

The problem in hand is multi-class classification, to be modelled using a neural network. Though there are several state-of-the-art packages available which can be deployed, but in this project, all the modules of a neural network have been built using simple scientific libraries in Python like numpy. Several optimization methods are build and used to optimise the gradient descent method. The experiment has been run on several parameters and final settings have been explained in more detail where we are getting best accuracy. The basic idea of this experiment is to dig deep into the mathematics behind deep learning, its various functions, and their implementation in python. It is also important because, we can look at the non-linear activation functions and demonstrate that a simple non-linear function cascaded in layers, can take a shape of many complex functions. Moreover, various optimisation techniques have been studied & implemented.

2 INTRODUCTION

2.1 PROBLEM DEFINITION

The challenge in hand is a multi-class classification problem, which needs to be solved using a multilayer neural network, also called deep neural network. The data presented was divided into 2 sets, 60,000 training examples, while a separate set of 10,000 data points to be used as test set. Each data point had 128 features. As per standard practice in any data modelling problem, the labels for training dataset was provided. A quick glance at the labels indicates 10 labels, hence making it a multiclass classification problem.

2.2 MULTILAYER NEURAL NETWORKS

As indicated in section 2.1, we are going to use a Multilayer Neural Network to solve this classification problem.

First of all, since it is a multilayer network, as also mentioned in section 1, it has multiple layers of some basic unit (neurons) connected in a hierarchical structure. In our MLP, we are building the parameters of each layer based upon the output of previous layer. The first layer is input layer (x), while we refer output as last layer. Also, we are not taking any feedback hence we are building a feedforward network, as opposed to a recurrent neural network.

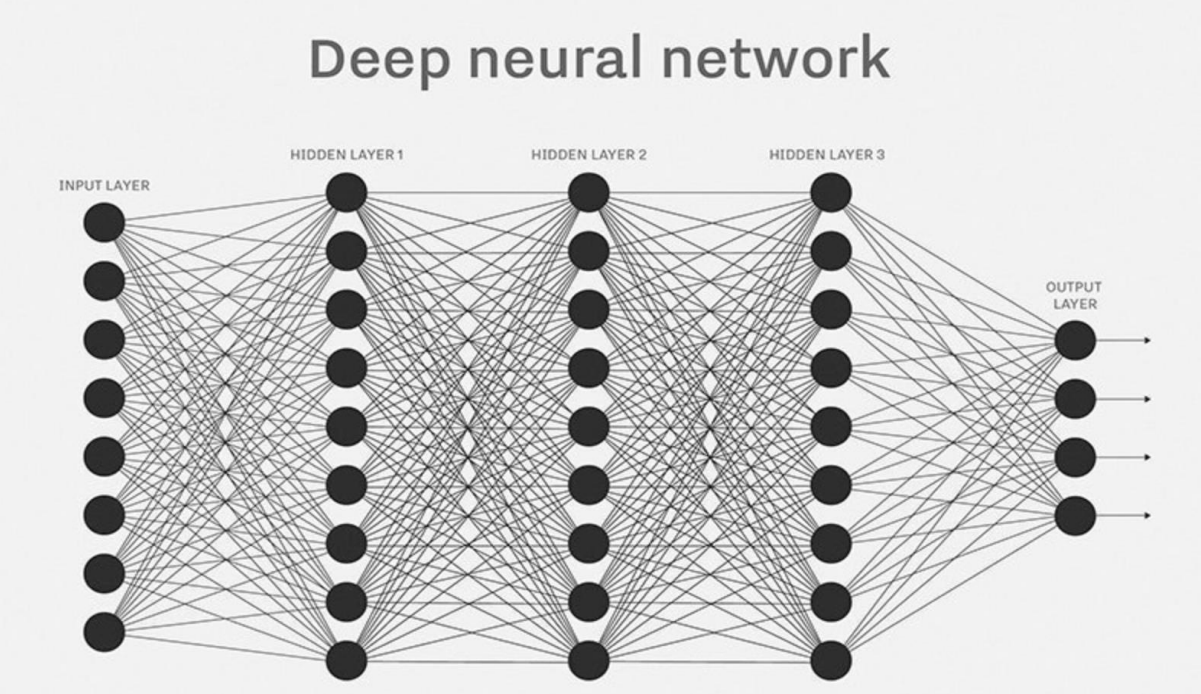


Figure 1 – a typical multilayer Neural Network (Rouse, 2018)

The most basic unit of this network (and each layer), as already mentioned above, is called a neuron. A neuron is a simple function which outputs the weighted sum of its inputs.

$$net = \sum_{i}^{d} w_i \cdot x_i$$

here x is the input to the neuron while w is the weight while x has d dimensions.

As we keep increasing neurons in a layer, and then more and more layers in a network, our function become more and more powerful and hence capable of generalisation of more complex models with lesser losses.

In order to build a multilayer perceptron, we will be using feedforward neural network, where weights input to each neuron are calculated based upon the its inputs – which are coming from previous layer. (Gupta, 2017) (Karpathy, 2019)

2.2.1 LOSS FUNCTION

The ultimate goal of the model is to reduce the cost in terms of loss function. Few of the more known loss functions are Euclidean distance and cross entropy.

Euclidean distance – J (loss function) is given in terms of t and z,

$$J(t,z) = \frac{1}{2}\sum_{k=1}^{C}(t_k - z_k)^2 = \frac{1}{2}\|\boldsymbol{t} - \boldsymbol{z}\|^2$$

where t is true value, while z is the estimated value.

2.2.1.1 CROSS ENTROPY

If we want to get probability distributions, cross entropy loss function is the loss function of choice. It is defined as given by following equation.

$$J(t,z) = -\sum_{k=1}^{C} t_k \log z_k$$

where t is true value, while z is the estimated value.

Due to the nature of this multiclass classification problem, we have selected cross entropy as our loss function and it has been implemented as follow

```
#Calculate cross entropy loss from predicted and actual label
def cross_entropy_loss(self, y,y_hat):

    activation_deriv=Activation(self.activation[-1]).f_deriv
    delta=-(y-y_hat)*activation_deriv(y_hat)
    return np.sum(-y * np.log(1e-15 + y_hat)),delta

#calculate gradient of cross_entropy
def delta_cross_entropy(self,X,y):

   # X is the output from fully connected layer (num_examples x num_classes)
   # y is labels (num_examples x 1)
   # Note that y is not one-hot encoded vector.

    m = y.shape[0]
    grad = self.softmax(X)
    grad[range(m),y] -= 1
    grad = grad/m
    return grad
```

2.2.2 BACKPROPAGATION

The objective is always to minimise the value of loss function, and this reduction based over several iterations while getting feedback from results is what we can call optimisation. In our case, our objective is to find the optimal values of weights in order to minimise the loss. There are many techniques used to optimise a model. We will be using a technique called backpropagation to optimise the weights and hence our model in order to minimise the loss. (Karpathy, 2019)

We use deferential of loss function in order to optimise our weights. Basically, we take derivative of loss function. A derivative at any point gives us the speed with which a function is changing its value at a given point. The derivative is giving us the rate with which our error rate will change, if we change the weights by a certain value.

Taking differential of complex function is hard, so we break our function into smaller parts using chain rule and limit our derivatives local to each neuron, hence only calculating simple derivatives for each step and feeding them back to the input of the neuron to change the weights. We stack all these local derivatives and add them together to get the optimised values of weights. This is called backpropagation.

Practically we are calculating the function in forward direction called forward feed and calculating the derivatives of loss function in backward direction called backpropagation.

2.2.3 ACTIVATION FUNCTIONS

As stated in section 2.2 a neural network consists of several layers. All layers in between input layer, and output layer are hidden, and hence called hidden layers. These are layers of neurons, joined together as well as connected to input and output layers. The hidden layers are getting their input from previous layers, but the first layer needs to be fed by some values. This is called activation, and the function used to activate this layer is called activation function. Activation functions are extremely important and should be selected with great care. The most important property of an activation function is nonlinearity. They also need to be continuous, differentiable, and monotonicity. There are several commonly used activation functions, like sigmoid function, tanh, ReLU, leaky ReLU.

Each one of these functions have their merits and demerits.

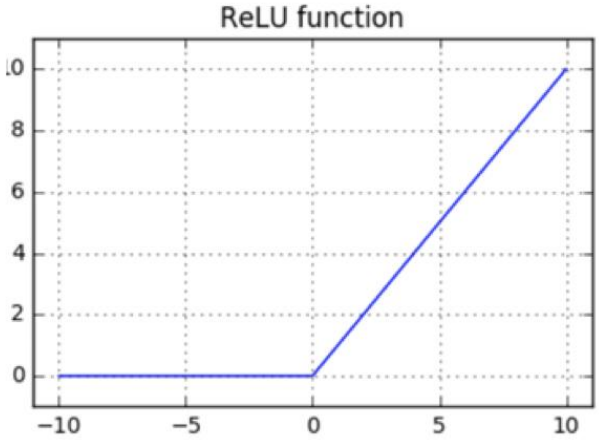
2.2.3.1 RELU FUNCTION

Among the above listed activation function, each one has its merits and demerits. Sigmoid and tanh functions for example can act linear on extreme values hence killing the purpose of a non-linear function. ReLU is a simple function which retains its input value for all positive values and gives zero value for negative inputs. We have used this function as out activation function for all layers except for the last one, right before the output layer.

A ReLU function can be given by following equation

$$f(s) = \max(0, s)$$

basically, it gives value of input for all positive values and 0 for negative values.



2.2.3.2 SOFTMAX FUNCTION

Another worth mentioning function is softmax function. It is the one used before the output layer for a multiclass classification, as it can assign conditional probabilities to each one of k classes.

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

where z is output, x is input

we have implemented this as follow

```
# Activation function to get probabilty output from output laer
def __softmax(self, x):
    """Compute softmax values for each sets of scores in x."""
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum(axis=0) # only difference
 # Derivative of softmax
def __softmax_deriv(self, a):
    """Compute softmax values for each sets of scores in x."""
    return self.__softmax(a)*(1-self.__softmax(a))
```

2.2.4 OPTIMIZATION

As explained in section 2.2.2, we use gradients and derivatives to optimize our weights in order to minimize the loss function. The most basic algorithm used is called Gradient descent. Where we optimize theta by subtracting the value of derivative of loss function for given x. this is given as

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta, \mathcal{X})$$

where greek letter (**η**) eta is learning rate, which decides the step of optimization, while J is denoting loss function, while theta is our parameters, including weights.

The problem with Gradient descent is that it is really hard to choose a good learning rate, where a large value can make it harder to converge while smaller value will make our model extremely slow and hence costly. Also, same learning rate is applied to all features.

There are several improved versions of gradient descent which can be used to speed up and improve the model convergence.

- **Momentum** – it moves faster where gradients are positive, i.e. model is converging – though learning rate is fixed for all features
- **Adagrad** – it adapts learning rate for various features
- **Adadelta** – It is an improvement from Adagrad, as it only considers learning rate over a said window
- **RMSprop** – it calculates velocity based upon previous value, which is exponentially decaying
- **Adam** – It combines both Adadelta and RMSprop

2.2.4.1 MINI-BATCH TRAINING

In SGD we pass a sample data as input, calculate output and pass that output to next layer as input until we get output from output layer. Once we reach final layer, we calculate error and pass derivative of error to immediate previous layer until we get to input layer. This is heavy on cost if data size is very large, and as we are updating weight only taking one input, the error and derivative fluctuate a lot. Also, if we take into account all points to calculate error it would be enormously slow process. In order to avoid both situations we use mini-batch training.

In mini-batch, we try to get the best of both worlds and also avoiding both extremes. We divide all examples into mini-batches from 8 – 200 and send randomly selected mini batch as our input. In this way, we are not running our algorithm for number of times equal to number of examples and also are avoiding sending all examples in our input.

```
#Calculate loss and gradient using mini-batch.
def mini_batch_SGD(self, X, y, n_epochs, learning_rate=0.1, batch_size=16, Drop_Out=0.25, Weight_Decay=0.003, Momentum=0.9):

    m=len(y)
    n_batches=int(m/batch_size)
    indices=np.random.permutation(m)
    X, y=X[indices], y[indices]
    loss=np.zeros(n_batches) #
    num_loss=0
    # Take a mini-bacth at a time
    for j in range(0,m,batch_size):
        X_j, y_j=X[j:j+batch_size], y[j:j+batch_size]
        i=np.random.randint(X_j.shape[0])
        m=0
        loss_m=np.zeros(batch_size)
        delta=np.zeros(10)
        #calculate delta and loss for mini-batch
        for _ in range(batch_size):
            y_hat = self.forward(X_j[i])
            loss_m[m], delta_m=self.cross_entropy_loss(y_j[i],y_hat)
            delta +=delta_m
            m +=1
        #calculate mean loss of a mini-batch
        loss[num_loss]= np.mean(loss_m)
        num_loss +=1
        #pass average delta of the mini-batch to the backward function
        self.backward(delta/batch_size)
        #Drop_Out=0.25, Weight_Decay=0.003
        #update weight and bias
        if self.Weight_Decay!=None:
          self.update_With_L2(learning_rate, n_epochs,lamda=Weight_Decay, gamma=Momentum) #by me for momentum
        else:
          self.update(learning_rate, n_epochs, gamma=Momentum)
```

2.2.4.2 MOMENTUM

When we train model with gradient descent, there is always chance of network falling into local minima, which not a desirable outcome. This happens because the error surfaces are not smoothly convex, so the surface may have many points that are local minima and are falsely considered as the global minimum. To, overcome this type of situation, there are several ways. Momentum is one of them widely used with SGD.

Momentum, simply adds a percentage of the previous weight update to the current weight update. This helps to prevent the model from getting stuck into local minima. Suppose we get to local minima, due to input from previous gradient into our calculation as given in below equation, we won't be stuck into a saddle point.

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

In the equation above, the greek letter γ is momentum, it's usual value is used 0.8 to 0.9, but as a hyperparameter we usually tune it in order to get validation accuracy. In our projects, we added momentum to update our weights and biases as below:

```
#Update weight and bias using Weight Decay(L2 regularisation) and Momentum (gamma)
def update_With_L2(self,lr, n_epochs, lamda=0.0003, gamma=0.65):

    for layer in self.layers:
        #Initialize gradient of weight and bias as 0 in first epoch
        if n_epochs == 0:
            layer.v_W=np.zeros_like(layer.grad_W)
            layer.v_b=np.zeros_like(layer.grad_b)
        #update gradient using Momentum(gamma) and learning rate
        layer.v_W = gamma * layer.v_W + lr * layer.grad_W
        layer.v_b = gamma * layer.v_b + lr * layer.grad_b

        layer.W=(1-lamda * lr) * layer.W -layer.v_W
        layer.b=(1-lamda * lr) * layer.b -layer.v_b
```

2.2.5 REGULARISATION

Deep neural networks contain several hidden neural layers which make them extremely powerful. This strength hence must be implemented with care, as especially in cases, where we have got limited training samples, the network tends to learn the function really well and hence causing overfitting. (Hinton & Srivastava, 2014). This is achieved by putting in place a phenomenon called regularization. There are several methods and algorithms designed to avoid this problem, which include following

- Adding noise – adding noise to inputs or weights
- Regularization – L1 & L2 – adding penalties to change to avoid big changes
- Data augmentation – adding more features – though the features must be chosen carefully
- Early stopping – stopping the model earlier than the best results
- Dropout – removing some of the neurons
- Batch normalization – though this is used to converge the model better but its side effect caused regularization

Methods we have used in this experiment as follow

2.2.5.1 WEIGHT- DECAY

To prevent overfitting of our model we need to use regularization. So, our model will work well on unknown test data. There are many regularization techniques we can use. Most popular and effective regularization is L2 regularization. It put a constraint that limit the growth of the weights in the model. As we can see on the equation below: a term Greek alpha is added to the loss function that penalises the large weights. As it penalises large weight it is named as weight decay.

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

2.2.5.2 EARLY STOPPING

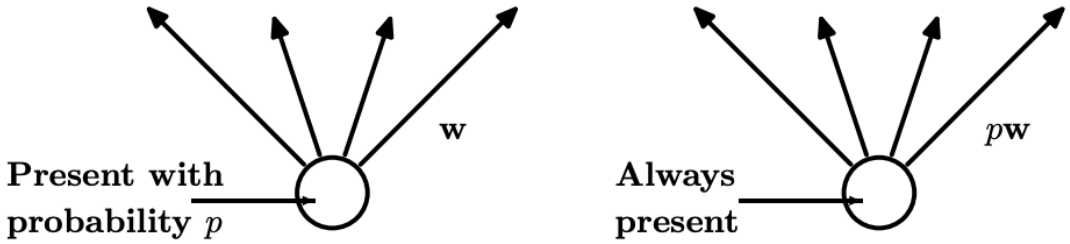
The idea behind early stopping is we keep train our model as long as validation accuracy increase. In our case we used mini-bacth, after passing a mini-batch to our model, we calculate training loss, derivative to update our weight and bias. After every small number of epochs, say 5 epochs, we calculate validation accuracy as well. As long as the accuracy keep on increasing we keep our model running for next 5 epochs, and keep updating parameters.

When validation accuracy stops increasing, or specifically in our algorithm, if it decreases in two consecutive checks, that's the point where we perform our early stopping. Even though the training loss might still be decreasing but that's the region of overfitting where we are increasing our test loss.

2.2.5.3 DROPOUT

As stated earlier, dropout can help reduce chances of overfitting. It is a simple but effective method, where your randomly drop some neurons and respective connections from a neural network during the training period.

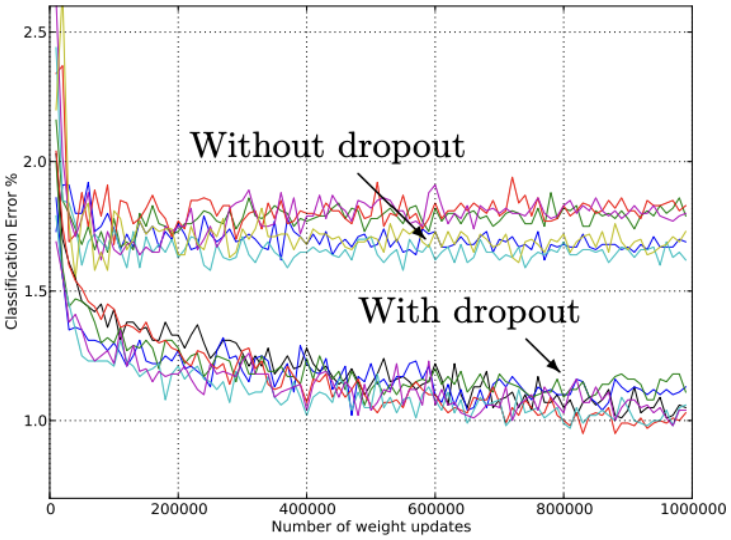
These dropouts are only done temporarily and later added back for other iterations as well as for testing. In the figure below, a neuron is shown during both training and test phases. It only appears during the training process with some given probability while it stays in the network at all times during the test period. The motivation for this method was taken from gender theory in evolution by Livnat in 2010 as stated by (Hinton & Srivastava, 2014).



(a) At training time (b) At test time

above figure taken from (Hinton & Srivastava, 2014). This explains a neuron which is only available with a given probability in case a, during the training time, while it is always present duirng the test time, though the weights have been corrected with same value p.

Results using drop out taken from (Hinton & Srivastava, 2014)



Here test error is shown comparing dropout and no-dropout for various networks, with hidden layers between 2 to 4.
The way we have implemented dropout, is during mini-batch, where both forward feed and backpropagation are run only on this reduced network where some neurons have been randomly removed or disconnected based upon the given probability.

```
#Drop percentage of nodes according to drop_out percentage provided
def drop_out(self, input_mat, drop_out=0.25):
    D =  np.random.rand(*input_mat.shape)
    D = D <   (1-drop_out )
    input_mat = input_mat * D
    input_mat = input_mat / (1-drop_out )

    return input_mat
```

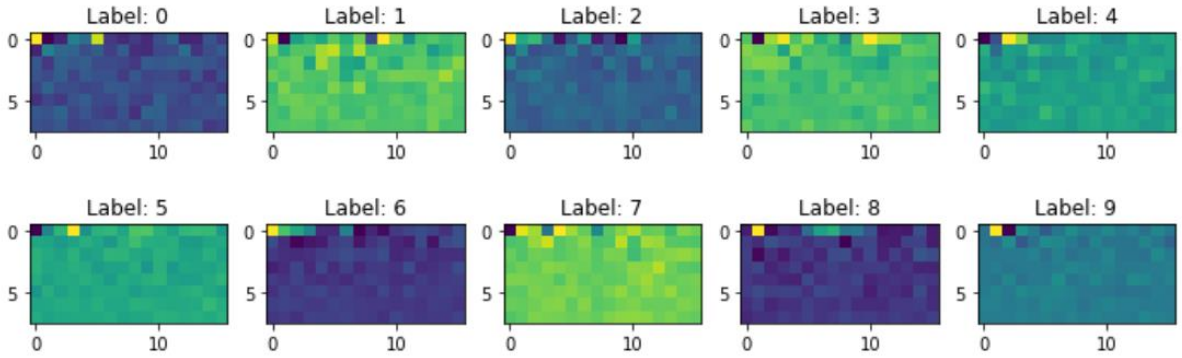
3 EXPERIMENT

3.1 DATA PREPARATION

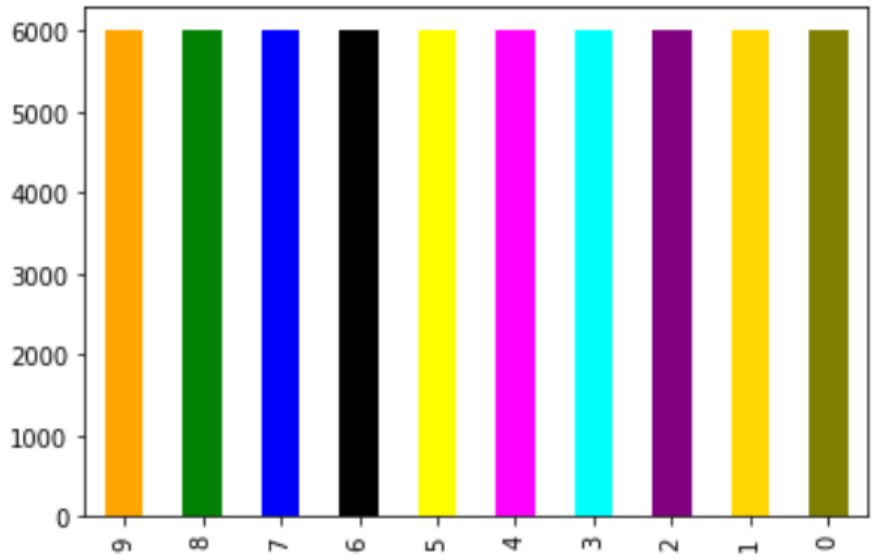
As indicated in section 2.1, we got training dataset with 128 dimensions, and 60,000 examples. We visualized this training set and realized this is image data. Each row corresponded to an image and 128 dimensions corresponded to pixel values for that image in each row. In order to start the data prep, the first step taken was to normalize all dimension hence dividing the values by 255, hence getting better classification by limiting our values between 0 to 1.

Next step was to divide training data (60,000 * 128) into training and test sets. This was done by manually splitting the given data set with labels. First of all, the data was randomized and then divided into 50,000 examples as training set and left out 10,000 examples as validation data set. To manage index smoothly, we did join train data and train label in same array. For train-test split and shuffle, we keep both data and label together to avoid index mix-up.

At the part of data pre-processing, we visualise one example of each label as shown in original data. As the image size is 128, we make the image size as 8 x 16.



Then we visualise the frequency distribution of all labels using bar chart. As we can see all labels are equally distributed.



3.2 ACTIVATION CLASS

We used the MLP tutorial activation class and enhanced it by adding more function to it. The existing class had methods tanh & tanh_derivative and sigmoid & sigmoid_derivative. We added softmax function and a softmax_derivative and ReLU & ReLU_derivative functions, to be used for this classification problem. As we know from section 2.2.3, softmax function as input to output layer is highly useful for multiclass classification problems where we can get probability of outputs. The main reason for adding relu is the limitations that come along with sigmoid and tanh as both of them takes any function into linearity when the values are too big or too small.

```
# Activation function to get probabilty output from output laer
def __softmax(self, x):
    """Compute softmax values for each sets of scores in x."""
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum(axis=0) # only difference
 # Derivative of softmax
def __softmax_deriv(self, a):
    """Compute softmax values for each sets of scores in x."""
    return self.__softmax(a)*(1-self.__softmax(a))

#Activation function relu
def __relu(self, x):
    x=np.maximum(0,x)
    return x
# Derivative of relu
def __relu_deriv(self, x):
    x=np.maximum(0,1)
    return x
```

3.3 HIDDEN LAYER CLASS

Hidden layer class was taken from same tutorial and enhanced. The first enhancement was introduction of some input parameters, and changing last layer activation from tanh to newly created activation function, softmax. Along with these, new functions were created including min-batch, dropout, weight decay, and other optimisation methods.

3.4 MLP CLASS & PROCESS

Class MLP is the real engine behind the solution which is running the whole process of classification and estimation of function. The methods in this class calculate the optimized weights for minimum errors for selected loss function against given labels and then make predictions on 10,000 examples left out for testing as indicated in section 3.1.

We will here explain the process followed in out experiment and touch some of the important functions created and used in MLP class. An MLP object is created by providing number of layers, and names of activation functions to be used from Class Activation. As part of initialization process, empty layers and neurons are created.

Once an MLP object (nn) is created, it estimates a model for the provided training data set and training labels. This is done using ‘fit’ method. It takes training and test data and labels, along with learning rates and epochs to be run. Minibatch SGD algorithm is run for given epochs. Minibatch SGD starts by shuffling the training dataset and then running SGD only taking m examples at a time for number of examples/m times. For each minibatch defined, it calculates forward feed using locally defined forward method but ultimately accessing hidden layer forward method.

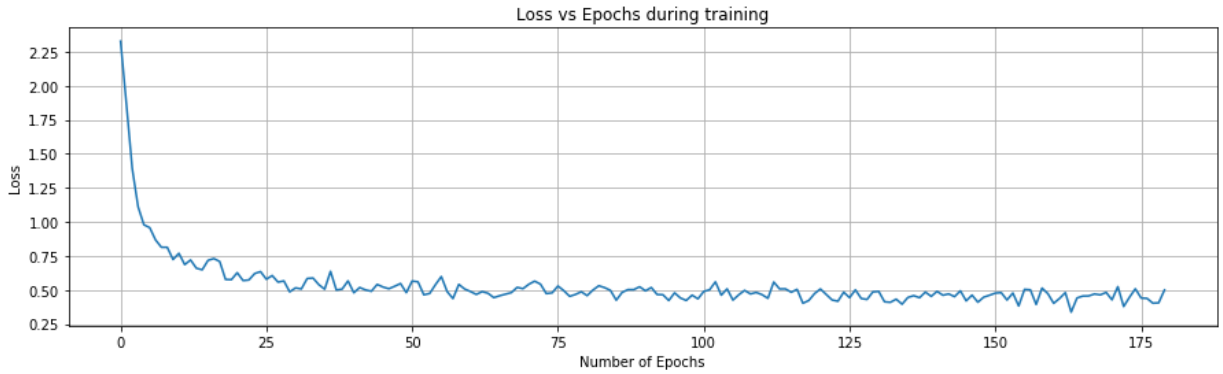
```
#forward with drop out rate provided
def forward_dropout(self, input,drop_out=0.25):
    '''
    :type input: numpy.array
    :param input: a symbolic tensor of shape (n_in,)
    '''
    lin_output = np.dot(input, self.W) + self.b
    #drop fraction of neurons provided by var drop_out
    lin_output=self.drop_out(lin_output, drop_out)
    self.output = (
        lin_output if self.activation is None
        else self.activation(lin_output)
    )
    self.input=input
    return self.output
```

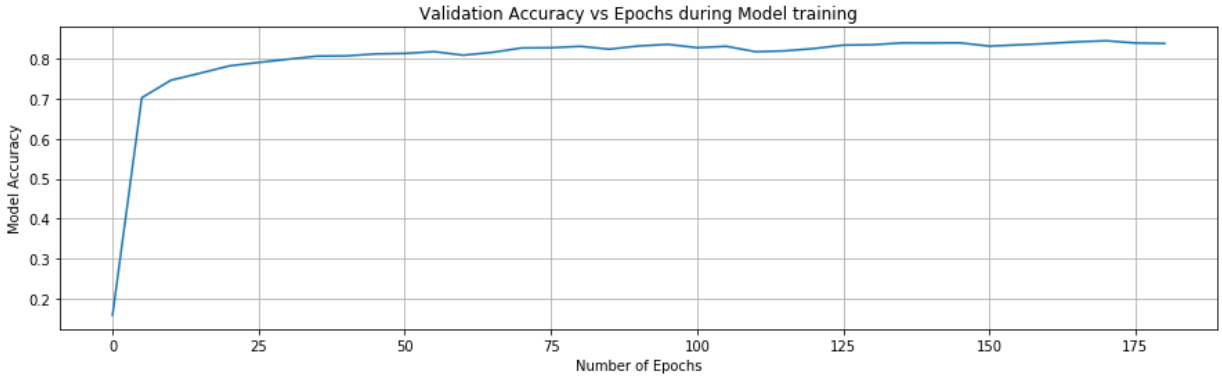
Next, it calculates loss and value of gradient using loss cross entropy loss function introduced in activation class. Once value of loss is obtained, back propagation is applied and values are updated using momentum along mini-batch.

```
def backward(self,delta):
    delta=self.layers[-1].backward(delta)
    for layer in reversed(self.layers[:-1]):
        delta=layer.backward(delta)
```

Once the model has been trained, the labels are predicted. This is done for both training dataset to estimate accuracy and then for the left-out test data to estimate accuracy on test data.

3.4.1 RESULTS

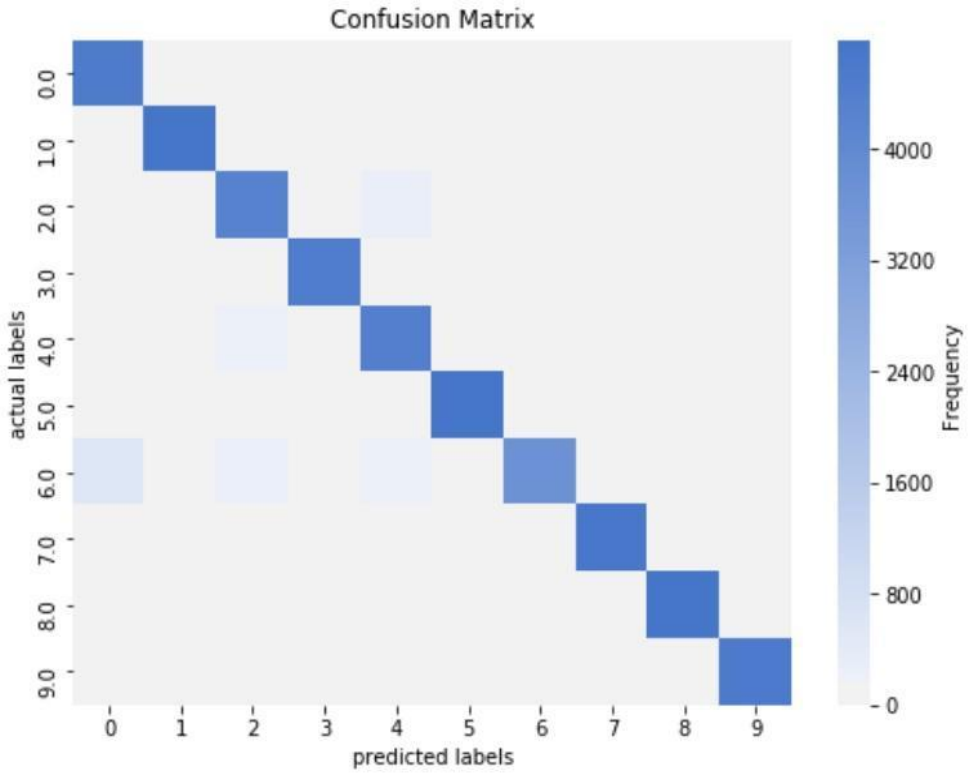




Our prediction as showing in confusion matrix as follows using our validation data set.

predicted labels	0	1	2	3	4	5	6	7	8	9
actual labels										
0.0	3947.0	20.0	52.0	411.0	19.0	19.0	482.0	3.0	66.0	2.0
1.0	3.0	4766.0	50.0	156.0	17.0	3.0	22.0	0.0	4.0	0.0
2.0	72.0	8.0	3147.0	57.0	872.0	10.0	783.0	1.0	51.0	2.0
3.0	151.0	61.0	41.0	4377.0	163.0	4.0	156.0	0.0	20.0	2.0
4.0	11.0	14.0	245.0	213.0	3866.0	2.0	651.0	0.0	27.0	0.0
5.0	2.0	1.0	0.0	4.0	0.0	4541.0	3.0	301.0	45.0	117.0
6.0	845.0	16.0	417.0	237.0	516.0	4.0	2884.0	3.0	80.0	4.0
7.0	0.0	0.0	0.0	0.0	0.0	269.0	0.0	4298.0	15.0	360.0
8.0	21.0	4.0	28.0	45.0	14.0	68.0	138.0	26.0	4629.0	15.0
9.0	0.0	1.0	0.0	0.0	0.0	111.0	0.0	185.0	5.0	4699.0

To visually representing our prediction result we show our confusion matrix using heat map:



We also calculated precision, Recall & F1 for each label. Though also can be seen from above confusion matrix, but we can see label 2, 4 & 6 had worst classification performance in our model while it is extremely good at classifying labels 1, 5, 8 & 9.

	Precision	Recall	F1_Score
0	0.867498	0.939954	0.902274
1	0.995006	0.988834	0.991910
2	0.892420	0.899707	0.896048
3	0.944257	0.945255	0.944755
4	0.880909	0.914387	0.897336
5	0.972516	0.983731	0.978092
6	0.895332	0.771352	0.828731
7	0.954527	0.976225	0.965254
8	0.983058	0.988778	0.985910
9	0.987298	0.961829	0.974397

3.5 CONCLUSION

We performed several experiments by changing hyperparameters and by adding/removing different functionalities including number of hidden layers.

The maximum accuracy we got on validation data is 89.06% and on training data 93.70%. We got this accuracy using three hidden layers with number of neurons in each layer [128,128,64,64,32,10], activation function as [None, 'tanh', 'relu', 'tanh' ,'logistic', 'softmax'], Learning rate=0.0017, Momentum=0.90, weight decay=0.0003, number of epochs we did in this case was 200.We used here SGD mini-batch with batch-size=4.

When we implemented adding dropout=0.25 and mini-batch gradient descent with batch-size 100, using three hidden layers with number of neurons in each layer [128,128,64,10], activation function as [None, 'relu', 'tanh', 'softmax'], we got 84.85% accuracy on validation data and 84.81 on training data. Notable thing here is if we add dropout it is giving much better accuracy in validation data in compare to training data. That means dropout is a good solution of avoiding overfitting in our model.

3.5.1 RUN TIME:

Although the model can be run for large number epochs but since we have embedded early stopping in our system, it automatically stops the algorithm hence limiting to total number epochs and time taken to complete the job. Though we had set maximum epochs to higher values during various testing phases but the model trains well at 60-100 epochs and takes less than 6 to 10 mins to train. Every 10 epochs it takes around 1 minutes when run in anaconda prompt on my laptop with configurations as given in section 3.5.2.

3.5.2 HARDWARE AND SOFTWARE SPECIFICATIONS

Operating System: Windows 10

Processor: Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz, 2904 Mhz, 2 Core(s), 4 Logical Processor(s)

Memory: 16GB 2133 MHz LPDDR3

Graphics: Intel(R) HD Graphics 620

Graphics Memory: 1.0 GB

4 FUTURE WORK

In future, we plan to work on further improving the precision and accuracy of the model by applying some more concepts used in deep learning especially batch normalisation, which we got running but could not be incorporated in this experiment. Further experiments need to run by adding noise to inputs & weights to see the impact on the results. Data Augmentation has also not been used during this experiment, it would be really nice to try creating some new features and then try to run this experiment with various combinations. Convolutional Neural networks works extremely well for image data; hence we also plan to apply some methodologies using CNN to create convolutional layers with a combination of various filters and strides.

5 WORKS CITED

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning.* MIT Press.

Gupta, T. (2017, Jan). *Deep Learning : Feedforward Neural Nerwork*. Retrieved from towardsdatascience: https://towardsdatascience.com/deep-learning-feedforward-neural-network-26a6705dbdc7

Hinton, G., & Srivastava, N. (2014). Dropout: A simple way to Prevent Neural Netowrks from overfitting. *Journal of Machine Learning Research 15*.

Karpathy, A. (2019). *CS231na:Convolutional Neural Networks for Visual Recognition*. Retrieved from Standford University: http://cs231n.stanford.edu/

Rouse, M. (2018, July). *neural-network*. Retrieved from searchenterpriseai: https://searchenterpriseai.techtarget.com/definition/neural-network

Szegedy, & Ioffe. (2015). Batch Normalization : Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arcXiv*.

Xu, C. (n.d.). COMP5329 DeepLearning. *Lecture notes*.

6 HOW TO RUN THE CODE

METHOD

1. Set current working directory in your IDE (we have tested in spyder) to following path

   ./code/algorithm

2. Run the “Comp_5329_Assig1_Program.py” file from same folder
3. It will take data from input folder stored in code folder and it will write its Predicted Label as hdf5 file in output folder, stored inside code folder

ALTERNATE

This code can also work and is tested on windows command prompt as follow:

1. Browse to following folder
   ./code/algorithm
2. Run the file using command “python Comp_5329_Assig1_Program.py”