

Deep Learning- CSE641

Assignment - 2 [Part 1]

Name: Akanksha Shrimal
Name: Vaibhav Goswami
Name: Shivam Sharma

Roll No: MT20055
Roll No: MT20018
Roll No: MT20121

FILES SUBMITTED : submitted .py file , .ipynb file and readme.pdf and output.pdf

ASSUMPTIONS :

- The fit function of the network class takes one hot encoded labels and preprocessed data.
- To display the Training and Testing accuracy during each epoch we have scored training dataset only on the first 1000 samples, to reduce the lag for each epoch.

UTILITY FUNCTIONS :

1. Save and Load models using Pickle

```
# Saving and Loading models using pickle
def save(filename, obj):
    with open(filename, 'wb') as handle:
        pickle.dump(obj, handle, protocol=pickle.HIGHEST_PROTOCOL)

def load(filename):
    with open(filename, 'rb') as handle:
        return pickle.load(handle)
```

2. Pre Processing data - Normalization and One Hot Encoding

```
# Utility function to normalize the data and one hot encode the labels
def pre_process_data(train_x, train_y, test_x, test_y):
    # Normalize
    train_x = train_x / 255.
    test_x = test_x / 255.
    enc = OneHotEncoder(sparse=False, categories='auto')
    train_y = enc.fit_transform(train_y.reshape(len(train_y), -1))
    test_y = enc.transform(test_y.reshape(len(test_y), -1))
    return train_x, train_y, test_x, test_y
```

3. Plotting functions

```
# function to plot double line graph
# Plot double line using X1 , Y1 and X2 , Y2
def plot_double_line_graph(X1,Y1,label1 ,X2 ,Y2,label2
,title,y_name):
    fig = plt.figure(figsize=(7,5))
    plt.subplot(111)
    plt.plot(X1,Y1 ,label=label1 ,marker = "x" , color="blue")
    plt.plot(X2, Y2 , label=label2 ,marker = "x" , color="red")
    plt.title(title)
```

```

plt.ylabel(y_name)
plt.xlabel('Epochs')
plt.legend( loc='upper left',prop={'size': 13})
plt.show()

# Plot single line using X1 , Y1
def plot_single_line_graph(X1,Y1,label1, title,name_y):
    fig = plt.figure(figsize=(7,5))
    plt.subplot(111)
    plt.plot(X1,Y1 ,label=label1 ,marker = "x" , color="blue")
    plt.title(title)

plt.ylabel(name_y)
plt.xlabel('Epochs')
plt.legend( loc='lower right',prop={'size': 13})
plt.show()

```

4. Plotting ROC Curve

```

# (7,7)
#https://www.dlology.com/blog/simple-guide-on-how-to-generate-roc-plot-for-keras-classifier/
def plot_roc(classes, y_test, y_score, figsize=(7,7)):
    n_classes = len(classes)
    # Plot linewidth.
    lw = 2

    # Compute ROC curve and ROC area for each class
    fpr = dict()
    tpr = dict()
    roc_auc = dict()
    for i in range(n_classes):
        fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_score[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])

    # Compute micro-average ROC curve and ROC area
    fpr["micro"], tpr["micro"], _ = roc_curve(y_test.ravel(),
y_score.ravel())

```

```

roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# Compute macro-average ROC curve and ROC area

# First aggregate all false positive rates
all_fpr = np.unique(np.concatenate([fpr[i] for i in
range(n_classes)]))

# Then interpolate all ROC curves at this points
mean_tpr = np.zeros_like(all_fpr)
for i in range(n_classes):
    mean_tpr += np.interp(all_fpr, fpr[i], tpr[i])

# Finally average it and compute AUC
mean_tpr /= n_classes

fpr["macro"] = all_fpr
tpr["macro"] = mean_tpr
roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])

# Plot all ROC curves
plt.figure(1)
plt.figure(figsize=figsize)
plt.plot(fpr["micro"], tpr["micro"],
         label='micro-average ROC curve (area = {0:0.2f})'
         ''.format(roc_auc["micro"]),
         color='deeppink', linestyle=':', linewidth=4)

plt.plot(fpr["macro"], tpr["macro"],
         label='macro-average ROC curve (area = {0:0.2f})'
         ''.format(roc_auc["macro"]),
         color='navy', linestyle=':', linewidth=4)

colors = cycle(['aqua', 'darkorange', 'cornflowerblue'])
for i, color in zip(range(n_classes), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=lw,
             label='ROC curve of class {0} (area = {1:0.2f})'
             ''.format(i, roc_auc[i]))

plt.plot([0, 1], [0, 1], 'k--', lw=lw)

```

```

plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Some extension of Receiver operating characteristic to
multi-class')
plt.legend(loc="lower right")
plt.show()

```

6. Plotting Confusion Matrix

```

def confusion_matrix_find(y, y_hat, nclasses):

    y = y.astype(np.int64)
    y_hat = y_hat.astype(np.int64)

    conf_mat = np.zeros((nclasses, nclasses))

    for i in range(y_hat.shape[0]):
        true, pred = y[i], y_hat[i]
        conf_mat[true, pred] += 1
    return conf_mat

# Plotting confusion matrix
def confusion_matrix_plot(cm, classes, title='Confusion matrix',
cmap=plt.cm.Blues, figsize=(7,7), path=None, filename=None):

    cm = cm.astype(np.int64)
    plt.figure(figsize=figsize)
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = 'd'

```

```
thresh = cm.max() / 2.  
for i, j in itertools.product(range(cm.shape[0]),  
range(cm.shape[1])):  
    plt.text(j, i, format(cm[i, j], fmt),  
             horizontalalignment="center",  
             color="white" if cm[i, j] > thresh else "black")  
plt.ylabel('True label')  
plt.xlabel('Predicted label')  
plt.tight_layout()
```

1. Neural Network:

- Input - 784, Output - 10
- Gradient Descent Optimizer
- Initialization - Random
- Activation:
 - Hidden - Sigmoid, ReLU, Tanh
 - Output - Softmax

FORWARD PROPAGATION:

1. Initialize the model at random.
2. The inputs are passed to the network layers.
3. A weighted sum of inputs is done which is an affine transformation of inputs.
$$z = w^T x + b$$
4. This weighted sum is passed through an activation function $g(z)$ like Sigmoid, ReLU, etc.
5. Loss is calculated on the final output at the output layer based on the ground truth and output of the network.

CROSS ENTROPY LOSS:

Loss is a measure of the performance of the model. Lower the loss, better the performance.

$$\text{Cross-entropy} = - \sum_x p(x) \cdot \log q(x)$$

where $p(X)$ is the probability of class X in target and $q(X)$ is the probability of X in prediction.

If the target is a one-hot vector then the loss is categorical cross-entropy loss.

$$\text{Categorical Cross-entropy} = -\log q(x)$$

x is the class —————
that is 1 in our TARGET

In binary classification, binary cross-entropy is used where the target is either 0 or 1.

$$\text{Binary Cross-entropy} = - \underbrace{(p(x) \cdot \log q(x))}_{\text{This cancels out if the target is 0}} + \underbrace{(1-p(x)) \cdot \log(1-q(x))}_{\text{This cancels out if the target is 1}}$$

In multi-label classification, the overall loss is calculated by summing up binary cross-entropy loss for each class calculated separately.

$$\text{Total Loss} = \sum_x \text{Binary Cross-entropy}_x$$

SOFTMAX ACTIVATION FUNCTION:

1. For multi-class classification, the softmax activation function is used at the output layer (generally).
2. This activation function outputs a multiclass categorical probability distribution, i.e. probability of each output label (total summing to 1).

$$y_c = \varsigma(\mathbf{z})_c = \frac{e^{z_c}}{\sum_{d=1}^C e^{z_d}} \quad \text{for } c = 1 \dots C$$

GRADIENT DESCENT:

1. It is an optimizing algorithm used to minimize a convex function (error or loss function generally in Machine Learning).
2. First, the values (weights) are randomly initialized.
3. The weights are updated as such:

$$weight^{(new)} = weight^{(old)} - constant \frac{\partial J(\Theta)}{\partial weight}$$

constant is the learning rate (alpha).

4. This is repeated until convergence, i.e. slope = 0.

VARIANTS OF GRADIENT DESCENT:

- Vanilla (Batch) Gradient Descent:
 1. It is a greedy approach where all the samples are taken into consideration together.
 2. The model weights are updated after all the samples have been evaluated (one epoch).
 3. This makes it less sensitive to noise but takes too much memory and is not efficient.

$$w = w - \alpha \nabla_w J(w)$$

- Stochastic Gradient Descent:
 1. In this method, the weights/parameters are updated after each training sample is evaluated.
 2. This makes SGD faster than Batch method but is prone to error if noise is in data.

$$w = w - \alpha \nabla_w J(x^i, y^i; w)$$

- Mini-Batch Gradient Descent:
 1. This method takes the advantages of above two methods and combines them to make an efficient algorithm.
 2. The updation of parameters is done after a few samples together (mini-batch) are evaluated.
 3. The weight updation is done using the average gradients across all the samples in a mini batch.

$$w = w - \alpha \nabla_w J(x^{\{i:i+b\}}, y^{\{i:i+b\}}; w)$$

BACKWARD PROPAGATION:

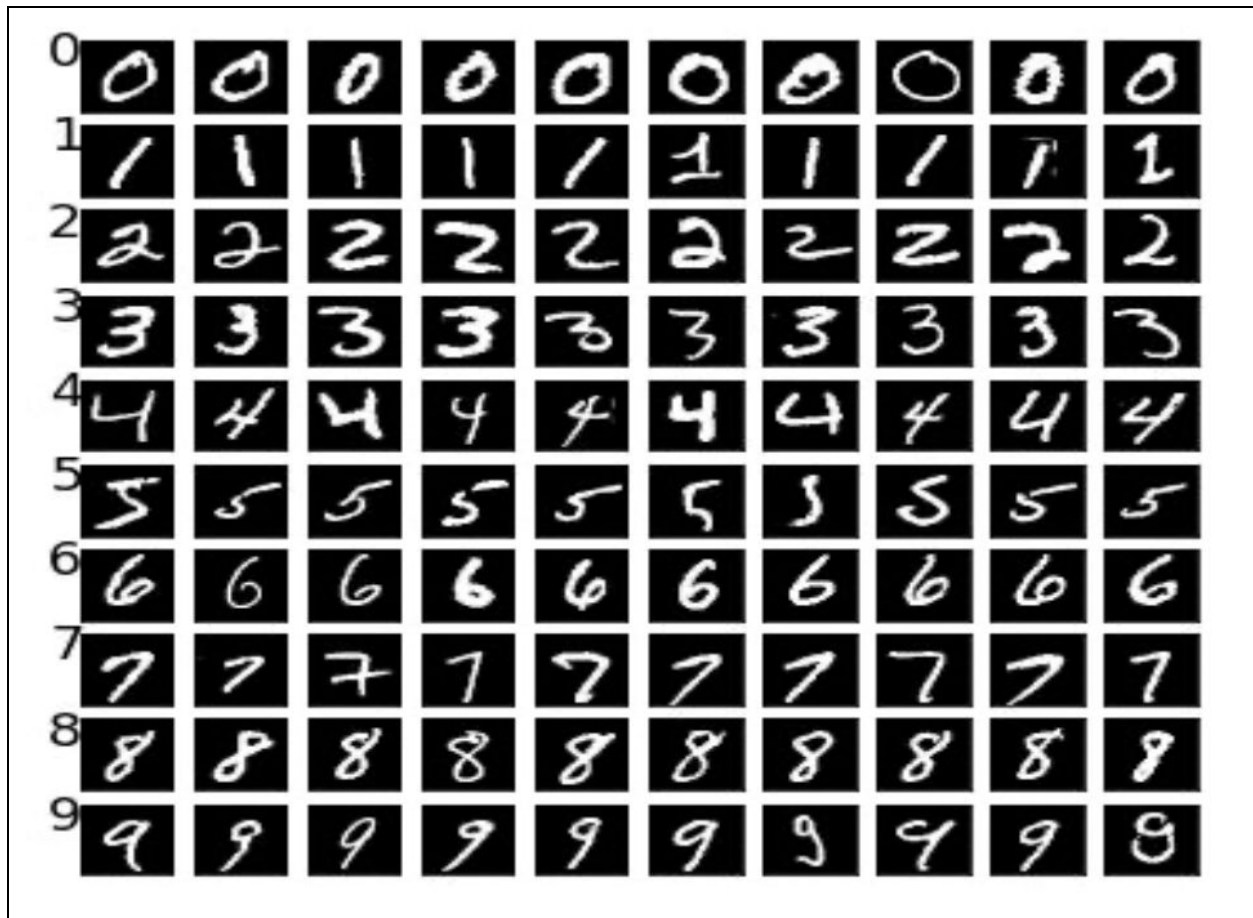
1. This allows information to go back from the output layer (cost) through the network computing the gradient at each step.
2. Gradient is final node output w.r.t. each node's weight.
3. In this propagation, weights are updated (fine-tuned) to minimize the loss.
4. Loss is calculated at every node to update the weight accordingly and finding which node is most responsible for the loss in every layer.
5. Finally, weights are updated for each layer, at each node.
6. The algorithm runs until convergence or until maximum iterations are reached.

DL Toolkit - Part-1

1.1 About dataset

A brief description of the dataset : MNIST

No of samples	10,000 (Training) , 2000 (Testing)
dimensions of each sample	(28,28)
Unique Labels	0,1,2,3,4,5,6,7,8,9



The following observations are made for the data :

- The data is categorical
- Each sample is a handwritten digit which are gray scale images with 28*28 pixels
- Every MNIST data point, every image can be thought as an array of numbers describing how dark these pixels is so each value in 28*28 represent intensity value between 0 and 1
- All the images are given labels among [0,1,2,3,4,5,6,7,8,9]

Training data is normalised using standard scaler from sklearn. And the same scaler is used to transform the Testing data.

One hot encoding is used for y-labels to do multi classification in MNIST.

1.2 Preprocessing dataset

Pre Processing Data

1. PIL Image is converted to np array
2. Image data is normalised using standard scalar from numpy.
3. Labels are one-hot encoded for multi-classification.

1.3 Training Network Methodology

- *Weight is Initialised randomly*

np.rand() is used to initialize the weights between 0 and 1, if np.randn() is used it may initialise weights with higher values i.e greater than 1 and may lead to problem of exploding gradients.

```
def initialize_network(self):
    if self._weight_init == 'random':
        np.random.seed(7)
        #np.seterr(over='raise')
        for i in range(len(self._layers)-1):
            self._weights[i] = np.random.rand(self._layers[i+1],
self._layers[i])*2-1
            self._bias[i] = np.random.rand(self._layers[i+1], 1)*2-1
            self._db[i] = np.zeros((self._layers[i+1], 1))
            self._dw[i] = np.zeros((self._layers[i+1],
self._layers[i]))
            self._optimizer_weight[i] = np.zeros((self._layers[i+1],
self._layers[i]))
            self._optimizer_bias[i] = np.zeros((self._layers[i+1], 1))

    if self._optimizer == 'adam':
        self._beta2 = 0.9
        self._optimizer2_weight = [None] * (len(self._layers)-1)
        self._optimizer2_bias = [None] * (len(self._layers)-1)
        for i in range(len(self._layers)-1):
            self._optimizer2_weight[i] = np.zeros((self._layers[i+1],
self._layers[i]))
            self._optimizer2_bias[i] = np.zeros((self._layers[i+1], 1))
```

- ***Data Loader Created***

Just like Pytorch a data loader is created to get data and labels into batches.

```
def loader(self, datas, labels, batch):
    for idx in range(0, datas.shape[0], batch):
        if idx == 0:
            yield datas[:batch, :], labels[:batch, :]
        else:
            yield datas[idx:idx+batch, :], labels[idx:idx+batch, :]
```

- ***Computing Loss***

Multi-class Cross Entropy loss is used to compute the Loss for each sample across the batch and then total loss sum is divided by the batch size to return average loss

```
# Compute the average loss across one batch passing the true
labels of batch
def get_loss_item(self, log_p, labels, batch_size):
    loss = -1*np.sum(np.multiply(labels
, np.log(log_p+self._eps)), axis=1)
    avg_loss = np.sum((loss)) * 1/self._batch_size
    return avg_loss
```

- *Functions for Activations*

Implemented activation and activation derivative functions for Relu, Tanh, Sigmoid.

```
def get_activation(self, name):
    if (name == 'sigmoid'):
        return self.sigmoid
    elif (name == 'relu'):
        return self.relu
    elif (name == 'tanh'):
        return self.tanh

def sigmoid(self, x):
    return 1/(1+np.exp(-x))

def relu(self, x):
    return max(0, x)

def tanh(self, x):
    a = np.exp(x)
    b = np.exp(-x)
    return (a - b)/(a + b)

def get_activation_derivative(self, name):
    if (name == 'sigmoid'):
        return self.der_sigmoid
    elif (name == 'relu'):
        return self.der_relu
    elif (name == 'tanh'):
        return self.der_tanh

def der_sigmoid(self, x):
    return x*(1-x)

def der_relu(self, x):
    return 1 if x>0 else 0

def der_tanh(self, x):
    return 1-(x**2)
```

- ***Zero grad function to make derivatives zero after each epoch***

```
def zero_grad(self):  
    for layer in self._dw:  
        layer.fill(0)  
    for layer in self._db:  
        layer.fill(0)
```

- ***Softmax Function***

Computes softmax for a numpy array , along a particular axis if mentioned.

```
def Mysoftmax(self,a, axis=None):  
    """  
    Computes exp(a)/sumexp(a); relies on scipy logsumexp  
    implementation.  
    :param a: ndarray/tensor  
    :param axis: axis to sum over; default (None) sums over  
    everything  
    """  
  
    lse = logsumexp(a, axis=axis) # this reduces along axis  
    if axis is not None:  
        lse = np.expand_dims(lse, axis) # restore that axis for  
    subtraction  
    return np.exp(a - lse)
```


- **Forward Propagation**

For Forward propagation weights and biases of each layer are stored in one array. A for loop is applied which takes input and performs $wx+b$ stores this value, calculates activated value, which is used as input for the next layer and repeats this process.

The activation for hidden layers is as specified and the activation for output layer is always softmax.

```
def forward_propagate(self):
    #print('forward_propagate')

    temp = self._y[0]
    for idx, (w_i, b_i) in
enumerate(zip(self._weights, self._bias)):
        z_i = np.dot(temp, w_i.T) + b_i.T
        self._z[idx] = z_i
        if (idx == len(self._weights)-1):
            y_i = self.Mysoftmax(z_i, axis=1)
        else:
            y_i = self._activation(z_i)
        self._y[idx+1] = y_i
    temp = y_i
```

- **Back Propagation**

A loop is applied in reverse fashion and loss from the last layer is propagated to the first layer. Also along this process the gradients for weights and biases are computed individually i.e dw and db. As this is a batch-gradient descent so we average out the gradients of all samples in a particular batch and that becomes our final d w and db.

The loss propagated to previous layers using softmax and cross entropy is $y_i - t_i$, where y_i is the predicted value of the output node and t_i is the actual label of the output node. (Reference :- <https://peterroelants.github.io/posts/cross-entropy-softmax/>)

```
def back_propagate(self, label):
    #print('back_propagate')
    for i in reversed(range(len(self._layers)-1)):
        if i == len(self._layers) - 2:
            self._delta[-1] = self._y[-1] - label
        else:
            if self._optimizer == 'nesterov':
                self._optimizer_weight[i+1] = self._beta *
self._optimizer_weight[i+1]
                self._optimizer_bias[i+1] = self._beta *
self._optimizer_bias[i+1]
                a1 = np.dot(self._delta[i+1],
self._weights[i+1]+self._optimizer_weight[i+1])
            else:
                a1 = np.dot(self._delta[i+1], self._weights[i+1])
                b1 = self._activation_derivative(self._y[i+1])
                self._delta[i] = np.multiply(a1,b1)
            cur_delta = self._delta[i]/self._batch_size
            self._db[i] = np.expand_dims(np.sum(cur_delta,axis=0),axis=1)
            for del_,inp in zip(cur_delta, self._y[i]):
                self._dw[i] += np.matmul(np.expand_dims(del_,axis=1),
np.expand_dims(inp,axis=0))
```

- *Update function for Gradient Descent*

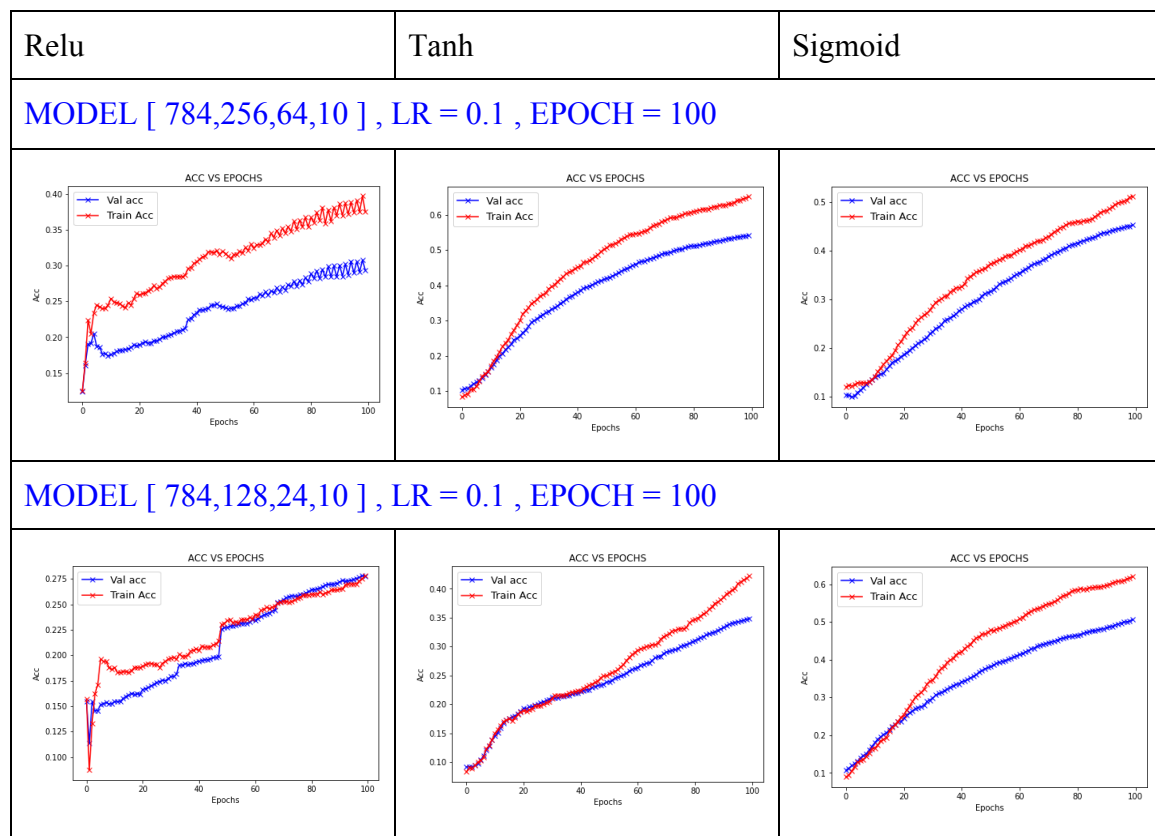
```
def gd(self):  
    # old = self._weights[0]  
    for i in range(len(self._weights)):  
        self._weights[i] = self._weights[i] - self._learning_rate*  
self._dw[i]  
        self._bias[i] = self._bias[i] - self._learning_rate*  
self._db[i]
```

1.3 Selection of Hyper Parameters

- **Selection of Network**

Two networks chosen :- [784,256,64,10] and [784,128,24,10]

Keeping all other hyper parameters same the model is analysed for both these networks over sigmoid, tanh and relu.



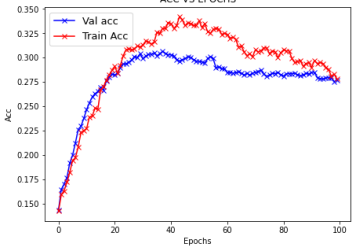
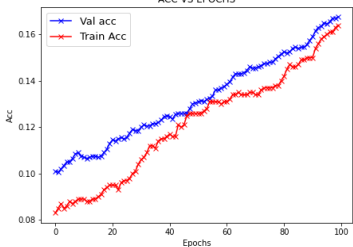
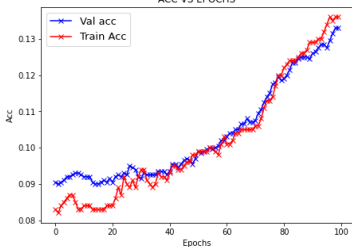
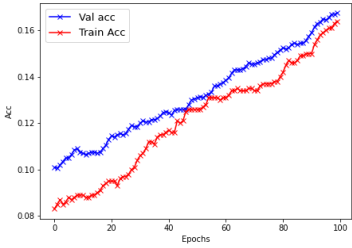
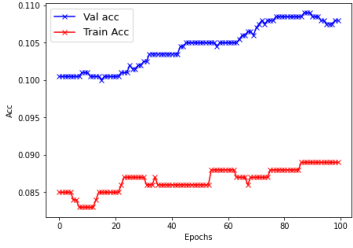
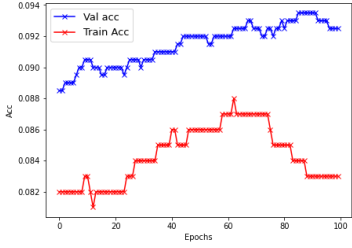
It can be observed that one is a complex model with more number of hidden nodes and one is a simpler model. Looking at the performance of both **(both perform equally similar across each activation)** and the resources available at hand we chose to go with a simpler model.

[784,128,24,10] chosen

- **Selection of Learning Rate**

Two Learning Rate chosen :- [0.01] and [0.001]

Keeping all other hyper parameters the same , the model is analysed for both these networks over sigmoid, tanh and relu.

Relu	Tanh	Sigmoid
MODEL [784,128,24,10] , LR = 0.01 , EPOCH = 100		
MODEL [784,128,24,10] , LR = 0.001 , EPOCH = 100		
		
		

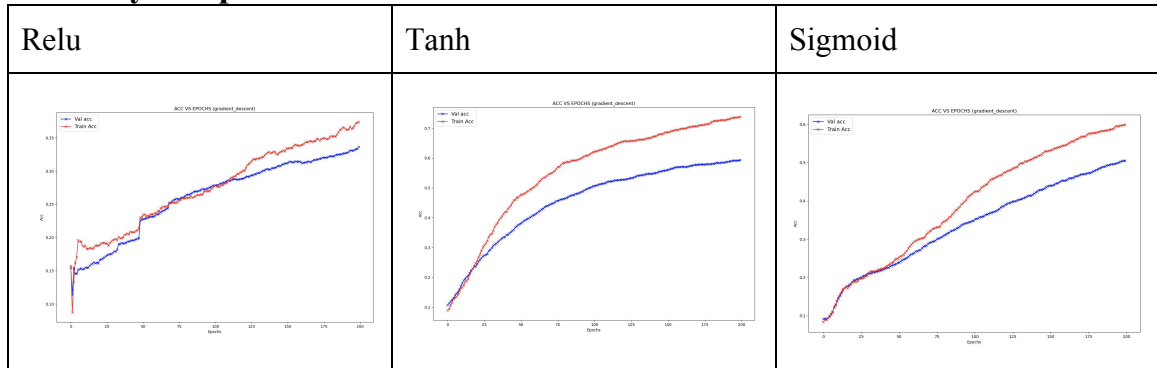
It is observed that Learning rate and number of epochs go hand in hand. If LR is chosen small then the model will take more number of epochs to converge and if taken small convergence can happen faster. Also if LR is kept too high then there might be too much deviation in losses across epochs, and if LR is kept too low then the model would not be able to converge.

0.01 chosen as LR

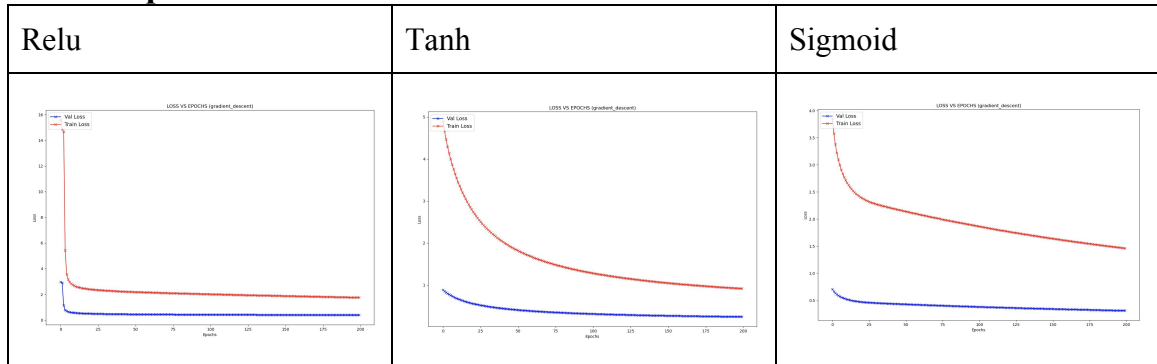
- **Comparison of Tanh , Relu , Sigmoid**

All the three activations are run for 100 epochs over the network [784,128,24,10] and LR=0.01. Following observations are made :-

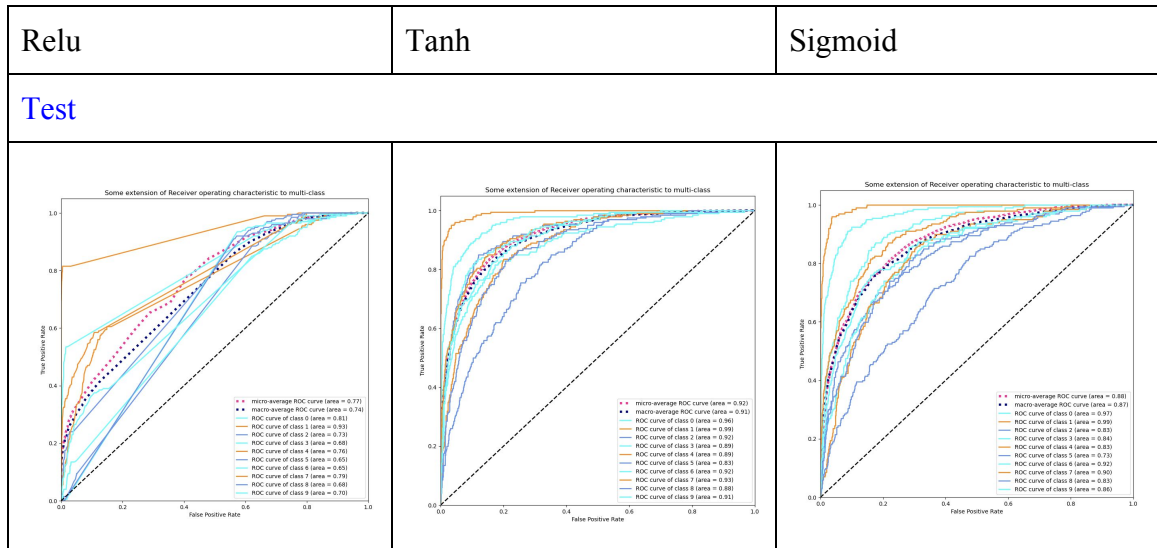
Accuracy vs Epoch



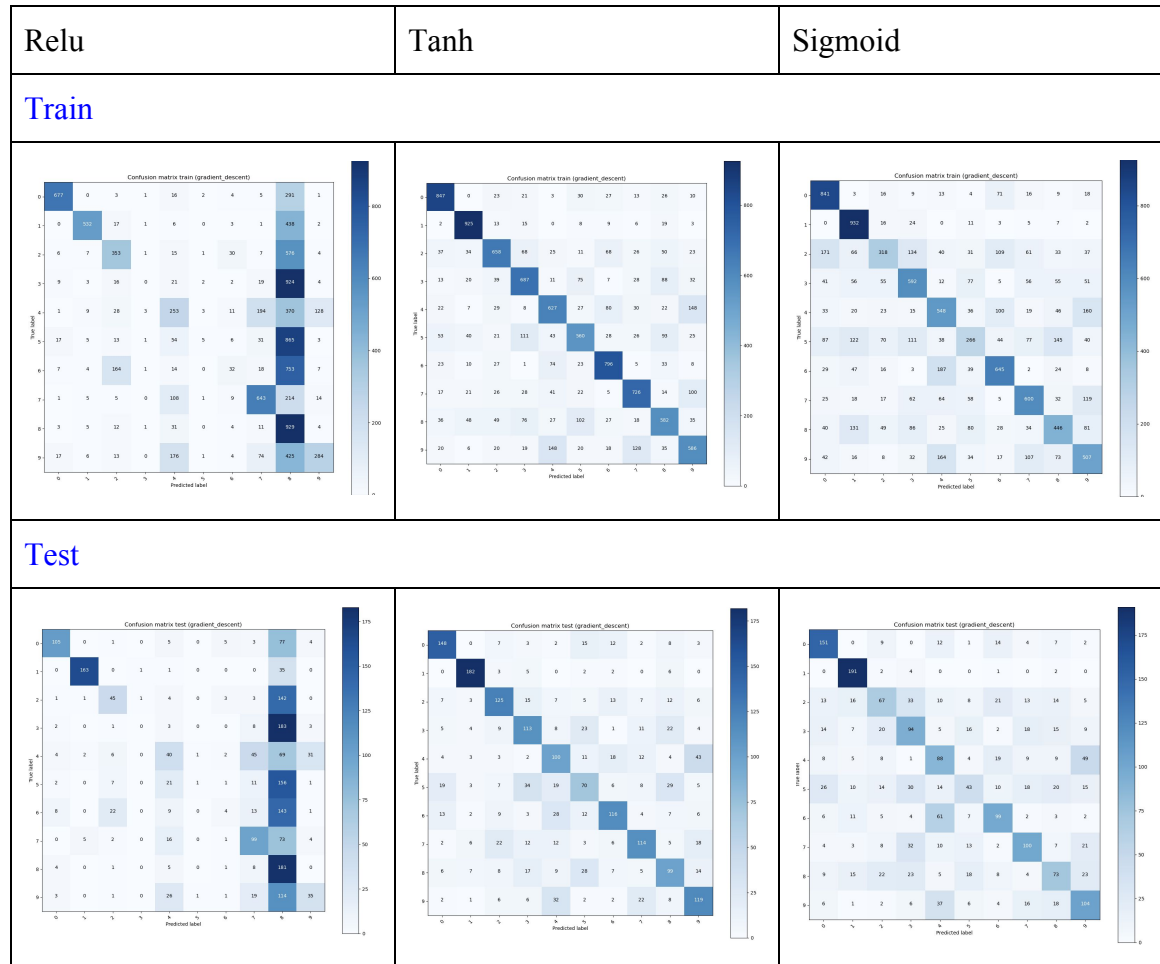
Loss vs Epoch



ROC curves



Confusion Matrix of Validation set



Observations :-

1. It can be observed that Relu is oscillating a lot at this learning rate and gets stuck on almost 35 percent accuracy only.
2. Sigmoid and Tanh comparatively improve consistently with each and every epoch , but in this case tanh performs better than sigmoid clearly. More over tanh is always considered as a better activation function compared to sigmoid.
3. So finally tanh is chosen as activation function.

Tanh chosen as the activation so final configuration becomes

Network = [784,128,24,10]

Learning Rate = 0.01

Epochs = Decided based on early stopping

Activation = Tanh

1.4 Finding Right number of epochs for Tanh activation

- **Tanh without early stopping**

Network = [784,128,24,10]

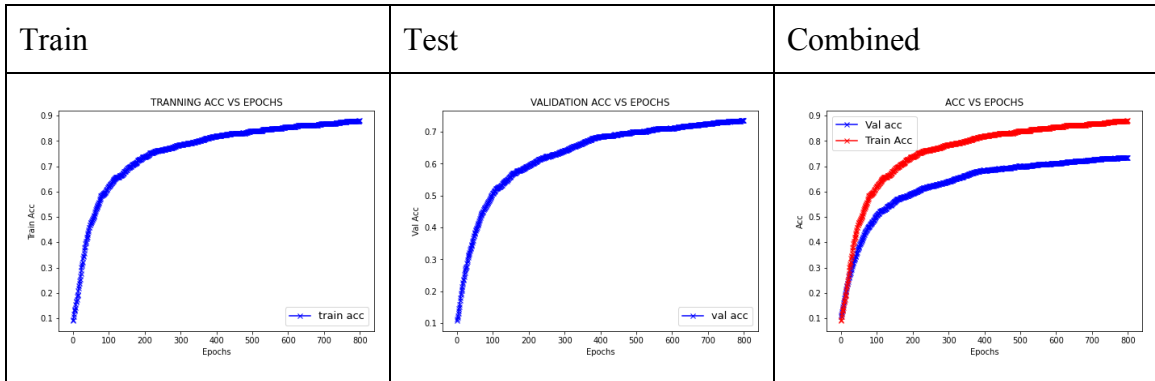
Learning Rate = 0.01

Epochs = 800

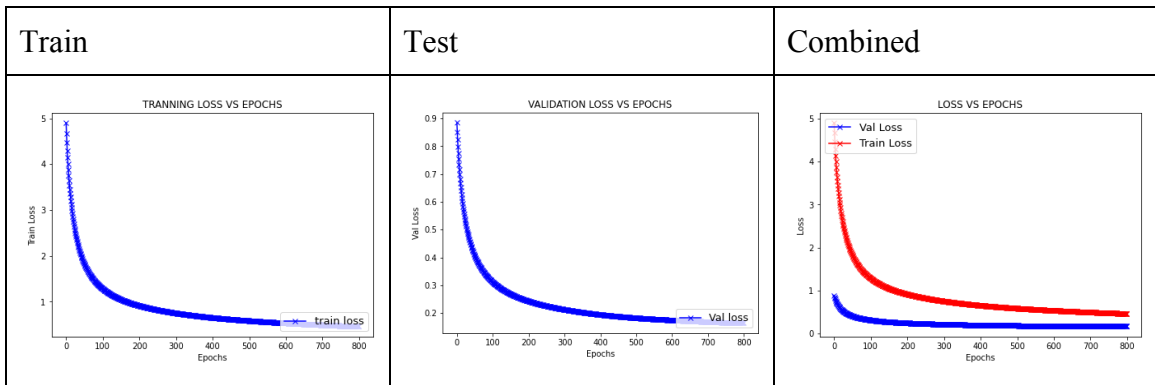
Activation = Tanh

It can be easily observed from the graphs that the model is overfitting after 450 epochs as loss has become constant and there is no significant change in accuracy.

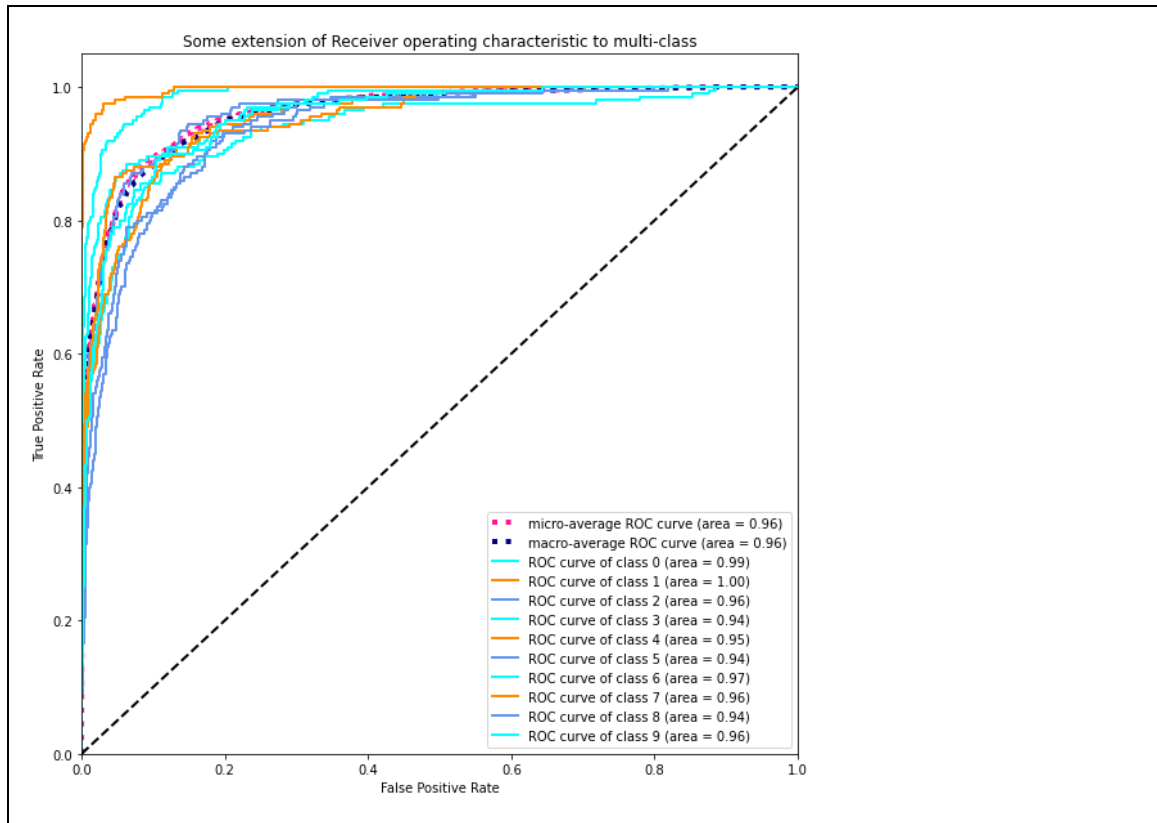
Accuracy vs Epoch



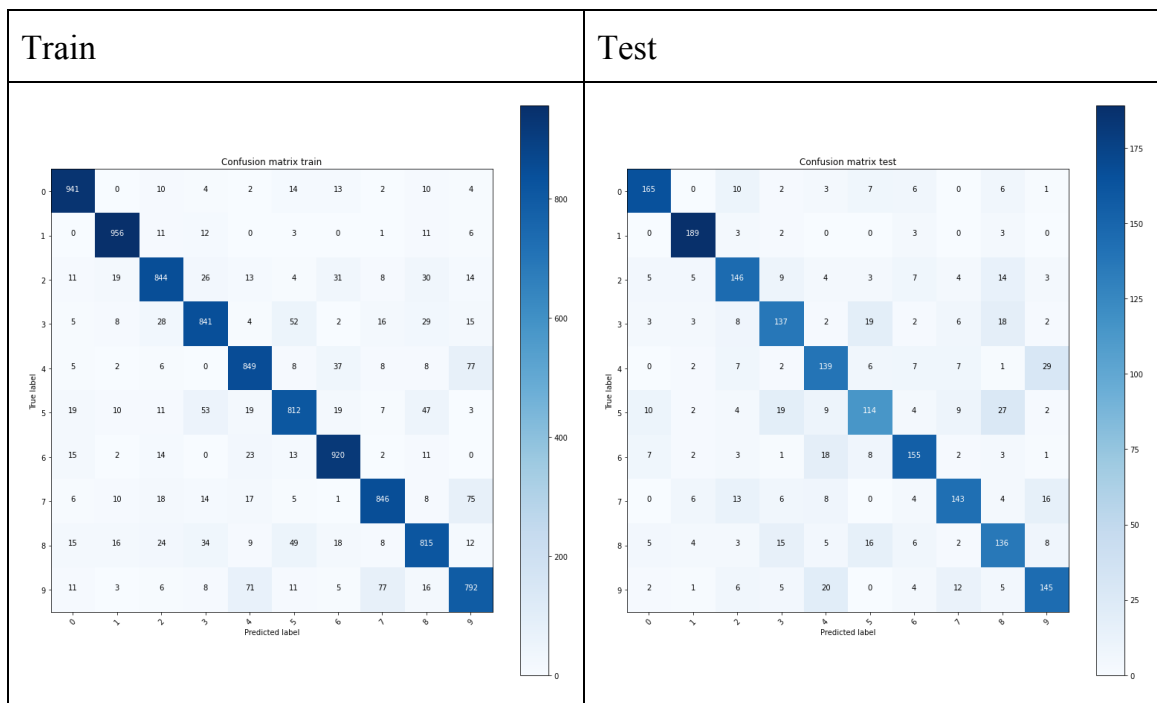
Loss vs Epoch



ROC curves



Confusion Matrix of Validation set

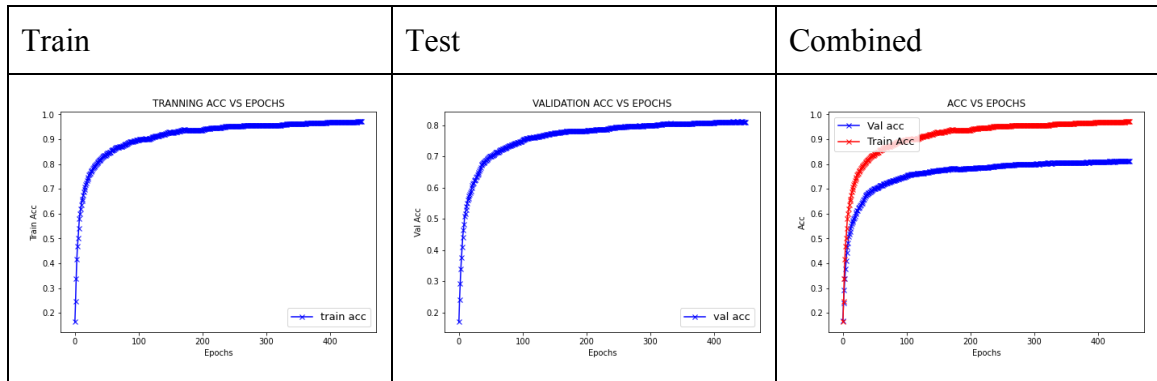


- **Tanh with early stopping**
Network = [784,128,24,10]
Learning Rate = 0.01
Epochs = 450
Activation = Tanh

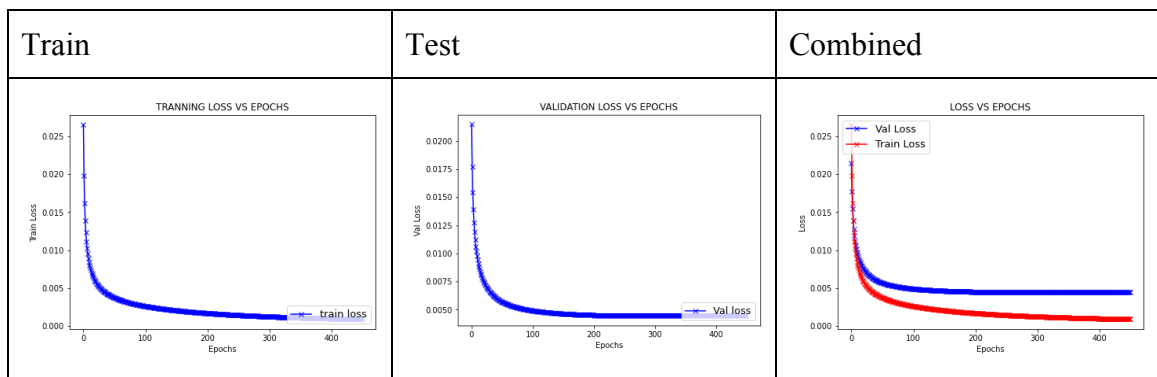
TESTING ACCURACY
81.10000000000001

TRAINING ACCURACY
96.61999999999999

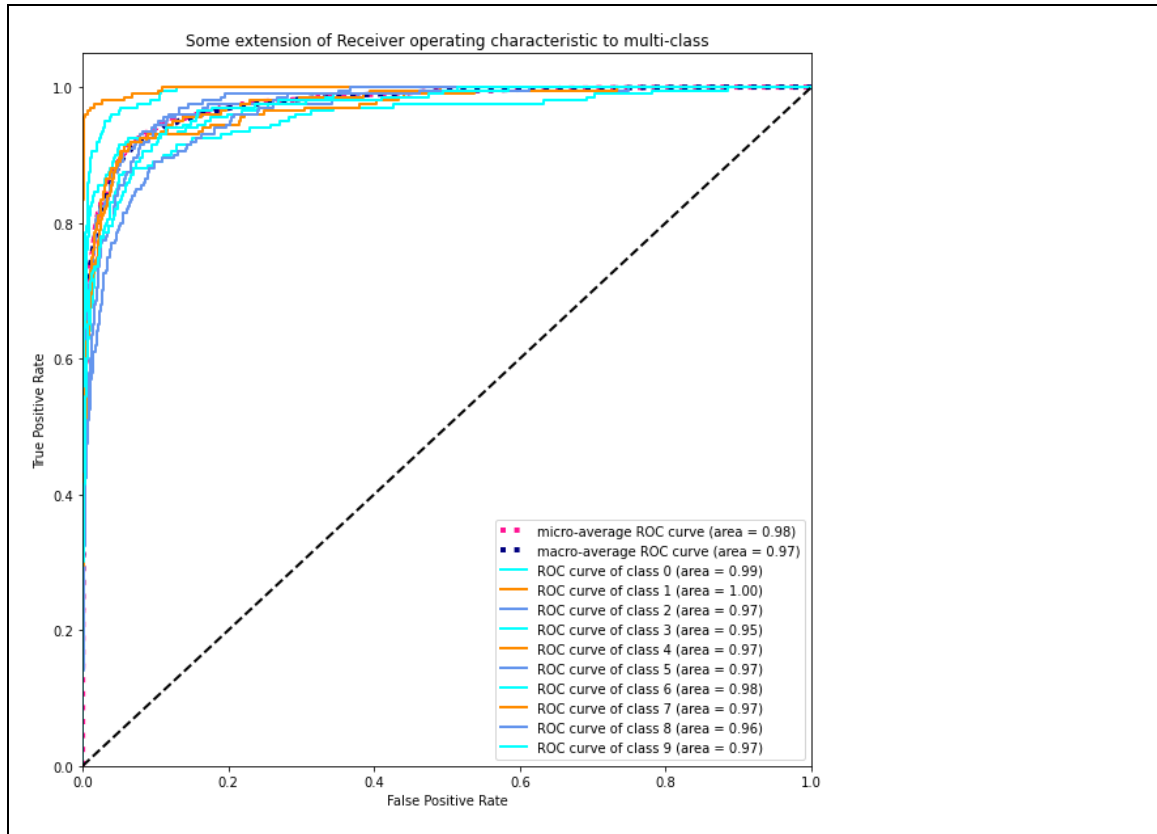
Accuracy vs Epoch



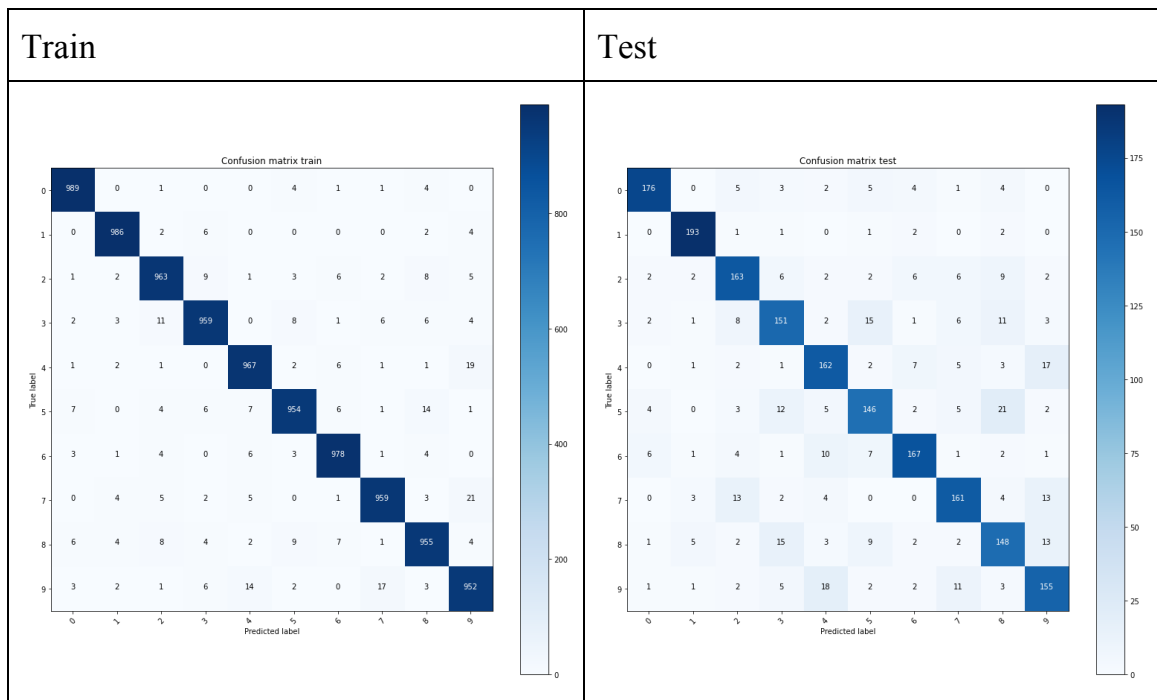
Loss vs Epoch



ROC curves (Val set)



Confusion Matrix of Validation set



1.5 Best Configuration form Part-1

To be used in later parts

Network = [784,128,24,10]

Learning Rate = 0.01

Epochs = 450

Activation = Tanh

Accuracy Stats for gradient_descent

Train Accuracy: 80.44

Train Accuracy: 69.1

=====

Optimizer: "gradient_descent"

Epochs: 450

Activation Fn(Hidden Layers): "tanh"

Activation Fn(Output Layer): "softmax"

Step size: 0.01

Weight initialization strategy: "random"

Regularization: "l2"

Dropout: 0.2

Batch size: 10000

Layer 1: (128, 784)

Layer 2: (24, 128)

Layer 3: (10, 24)

2. Optimizers:

Optimizers are algorithms or methods used to change the attributes of the neural network such as weights and learning rate to reduce the losses. Optimizers are used to solve optimization problems by minimizing the function.

1. Gradient Descent with Momentum:

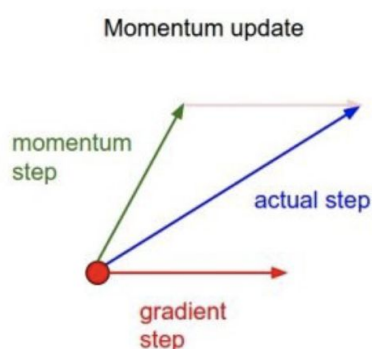
- In Stochastic Gradient Descent, the derivative over noisy data gave high variance. To overcome that problem, momentum is used.
- Momentum accelerates the steps (convergence) if in the right direction and decelerates it if in the wrong direction.
- It uses exponential weighting average which gives more weightage to recent updates and less weightage to previous ones.

$$V_t = \beta V_{t-1} + \alpha \nabla_w L(W, X, y)$$
$$W = W - V_t$$

2. Nesterov Accelerated Gradient:

- If the momentum is too large, the algorithm will miss the local minima and continue to rise up or oscillate.
- To resolve this issue, NAG is used which is a look ahead method. It looks one step ahead and then decides where to take the step resulting in faster convergence.

$$V_t = \beta V_{t-1} + \alpha \nabla_w L(W - \beta V_{t-1}, X, y)$$
$$W = W - V_t$$



3. Adaptive Gradient (AdaGrad):

- All the previous methods have a fixed learning rate for the parameters.
- The idea of this algorithm is to have an adaptive learning rate for each of the weights.
- The learning rate decreases with epochs.

$$W_t = W_{t-1} - (lr_t * (\nabla_{W_{t-1}} L))$$

$$g_t = \nabla_{W_{t-1}} L$$

$$W_t = W_{t-1} - (lr_t * g_t)$$

$$lr_t = \frac{lr}{\sqrt{\alpha_{t-1} + \varepsilon}}$$

$$\alpha_{t-1} = \sum_{i=1}^{t-1} g_i^2$$

4. RMSprop:

- It is an adaptive learning rate method, similar to AdaGrad.
- The algorithm adapts the diminishing learning rate method, i.e. the learning rate changes over time.
- RMSprop additionally divides the learning rate by an exponentially decaying average of squared gradients.

$$v_{dw} = \beta \cdot v_{dw} + (1 - \beta) \cdot dw^2$$

$$v_{db} = \beta \cdot v_{db} + (1 - \beta) \cdot db^2$$

$$W = W - \alpha \cdot \frac{dw}{\sqrt{v_{dw}} + \epsilon}$$

$$b = b - \alpha \cdot \frac{db}{\sqrt{v_{db}} + \epsilon}$$

5. Adam:

- It works with first order and second order moments of gradients.
- The first moment is the mean and the second moment is the uncentered variance.
- This algorithm combines the advantages of AdaGrad and Gradient Descent with momentum.
- It uses squared gradients to scale learning rate like RMSprop and uses the advantage of momentum by using moving average of gradient like SGD with momentum.
- To estimate the moments, Adam utilizes exponentially moving averages, computed on the gradient evaluated on a current mini-batch.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \left[\frac{\delta L}{\delta w_t} \right] \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2) \left[\frac{\delta L}{\delta w_t} \right]^2$$

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

$$w_{t+1} = w_t - \widehat{m}_t \left(\frac{\alpha}{\sqrt{\widehat{v}_t} + \epsilon} \right)$$

2.1 Methodology, Plots and Observations

1. Gradient Descent with Momentum Optimizer:-
2. Nesterov's Accelerated Gradient Optimizer:-
3. AdaGrad Optimizer:-
4. RMSprop Optimizer:-
5. Adam Optimizer:-

Gradient Descent with Momentum Optimizer:-

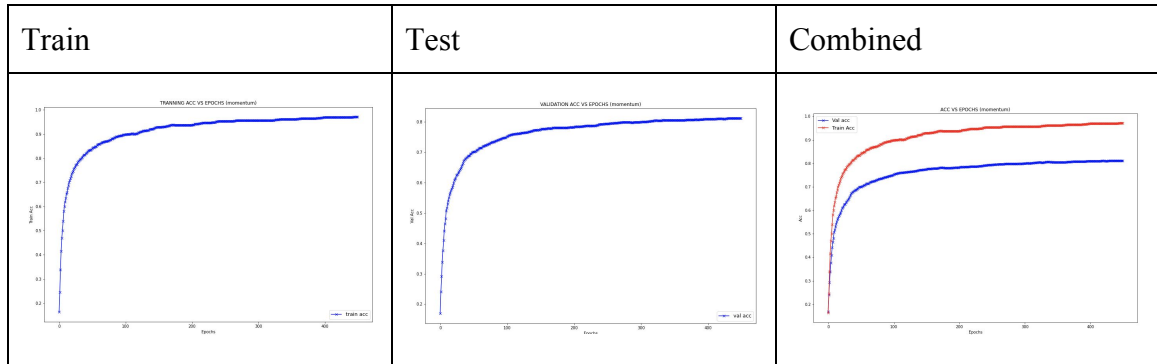
Methodology

```
def momentum_gd(self):  
    for ix in range(len(self._weights)):  
        self._optimizer_weight[ix] =  
            self._optimizer_weight[ix]*self._beta -  
            self._learning_rate*self._dw[ix]  
  
        self._optimizer_bias[ix] =  
            self._optimizer_bias[ix]*self._beta -  
            self._learning_rate*self._db[ix]  
  
        self._weights[ix] += self._optimizer_weight[ix]  
        self._bias[ix] += self._optimizer_bias[ix]
```

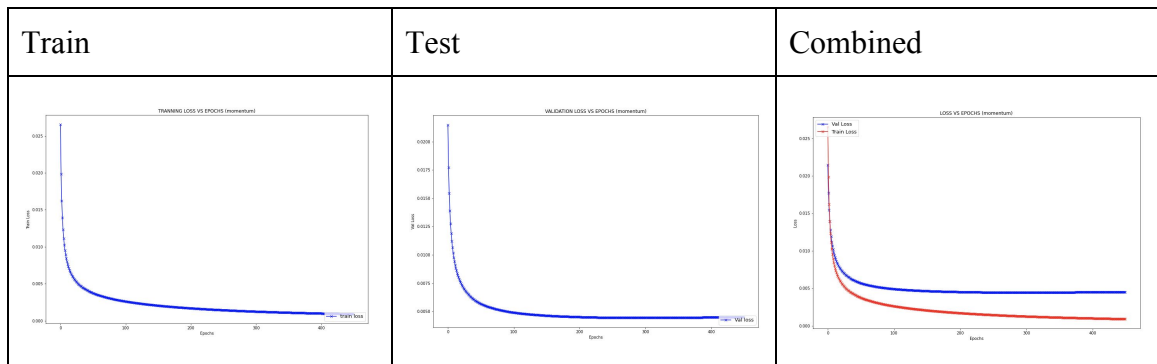
optimizer_weigh(\mathbf{m}_t) holds the parameters for momentum. We call momentum_gd to update the model weights after each batch which is 64 in our setup. momentum is first updated(\mathbf{m}_{t+1}) using the previous momentum(\mathbf{m}_t) * gamma along with batch gradient * step size.

Plots

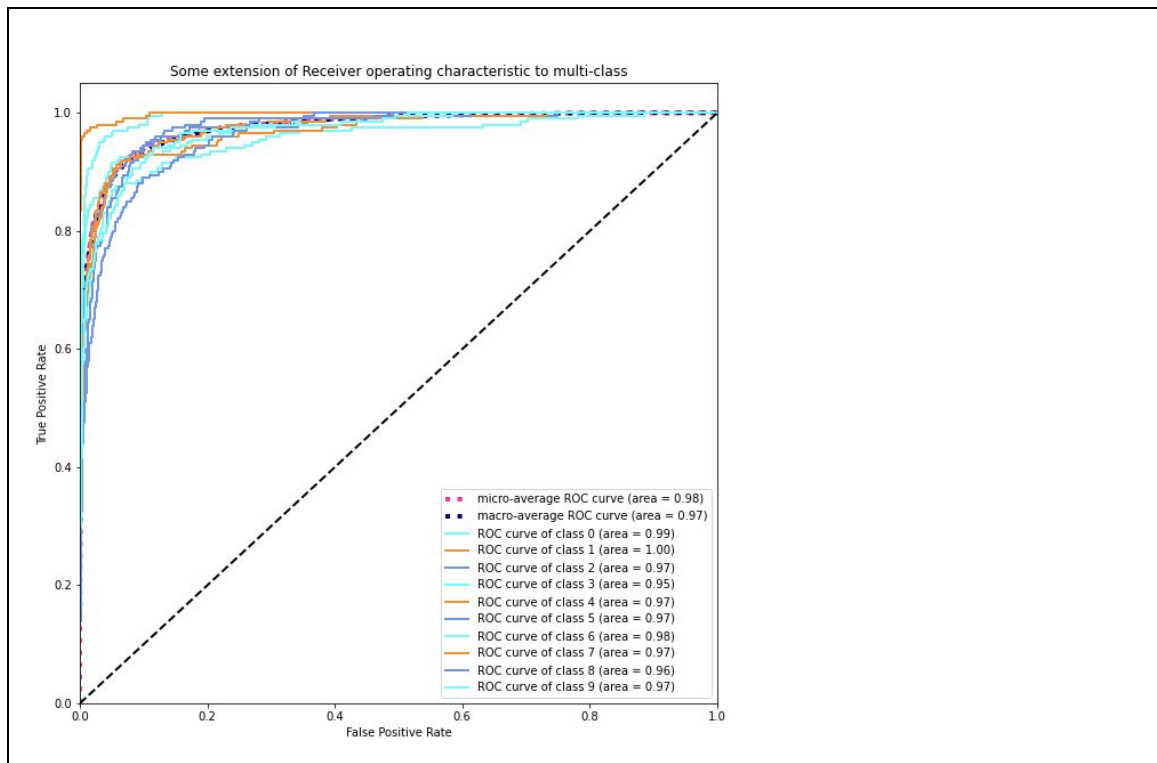
Accuracy vs Epoch



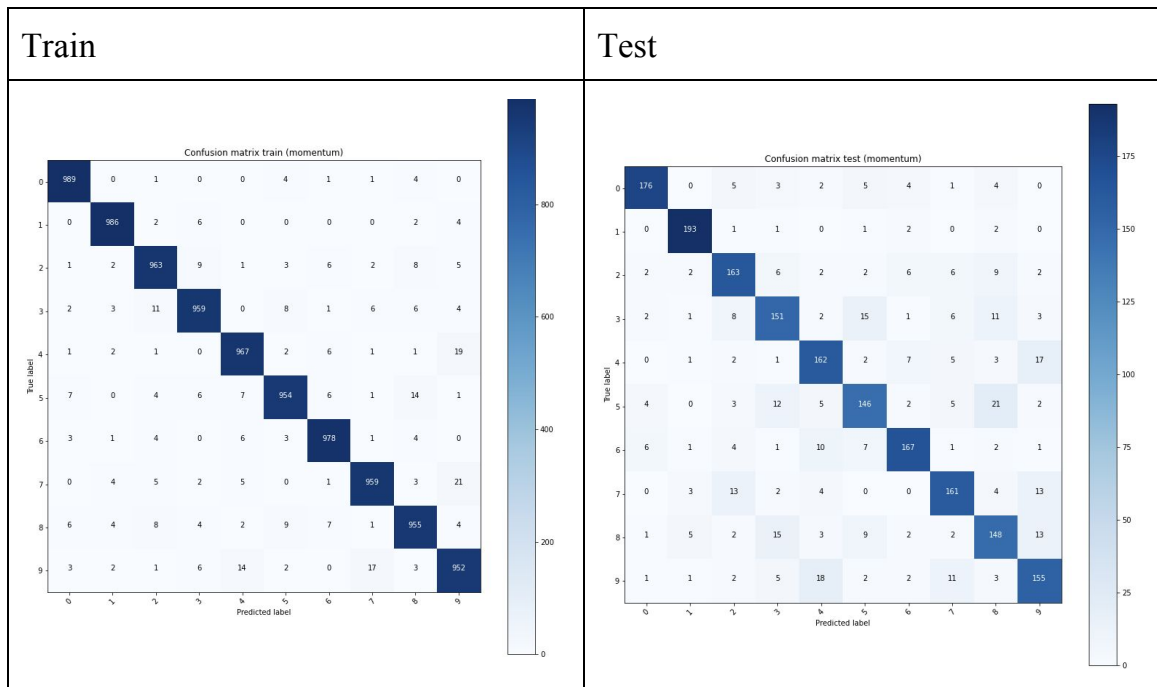
Loss vs Epoch



ROC curves (Val set)



Confusion Matrix of Validation set



Accuracy Stats for momentum
 Train Accuracy: 96.61999999999999
 Train Accuracy: 81.10000000000001

=====

Optimizer: "momentum"

Epochs: 450

Activation Fn(Hidden Layers): "tanh"

Activation Fn(Output Layer): "softmax"

Step size: 0.1

Weight initialization strategy: "random"

Regularization: "l2"

Dropout: 0.2

Batch size: 64

Layer 1: (128, 784)

Layer 2: (24, 128)

Layer 3: (10, 24)

Nesterov's Accelerated Gradient Optimizer:- Methodology

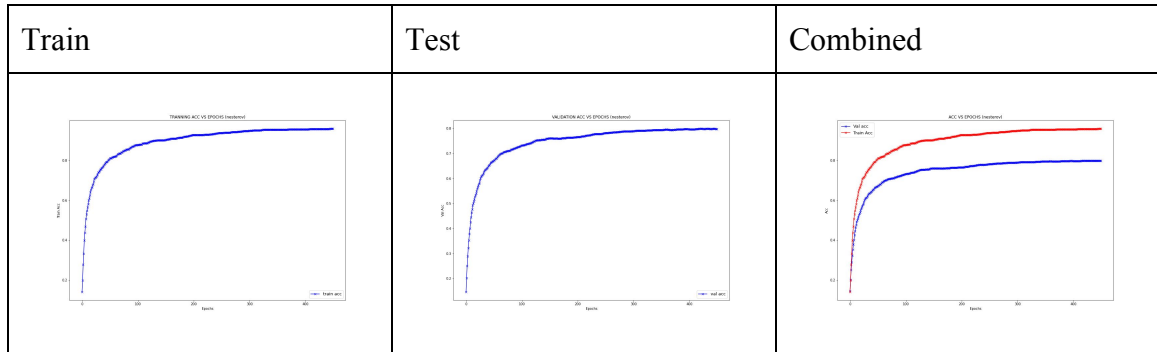
```
def nesterov_accelerated_gd(self):  
    for ix in range(len(self._weights)):  
        self._optimizer_weight[ix] =  
            self._optimizer_weight[ix]*self._beta -  
            self._learning_rate*self._dw[ix]  
  
        self._optimizer_bias[ix] =  
            self._optimizer_bias[ix]*self._beta -  
            self._learning_rate*self._db[ix]  
  
        self._weights[ix] += self._optimizer_weight[ix]  
        self._bias[ix] += self._optimizer_bias[ix]
```

```
if self._optimizer == 'nesterov':  
    self._optimizer_weight[i+1] = self._beta *  
self._optimizer_weight[i+1]  
    self._optimizer_bias[i+1] = self._beta * self._optimizer_bias[i+1]  
    a1 = np.dot(self._delta[i+1],  
self._weights[i+1]+self._optimizer_weight[i+1])  
    else:  
        a1 = np.dot(self._delta[i+1], self._weights[i+1])
```

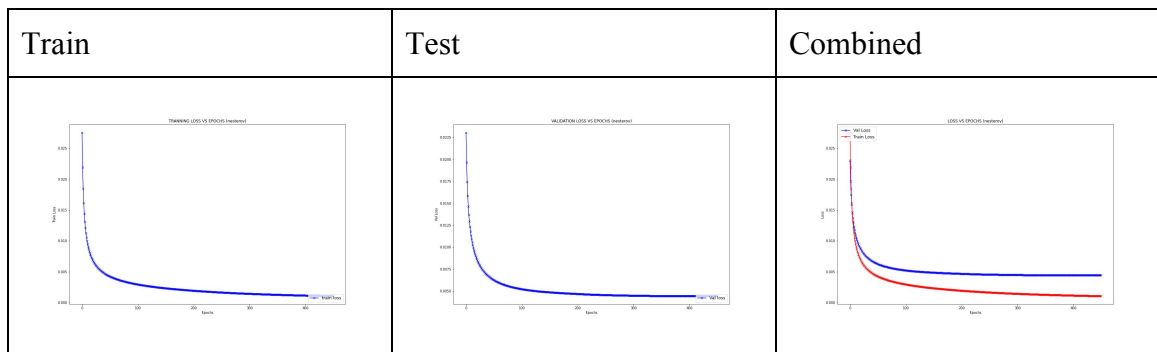
optimizer_weight(\mathbf{m}_t) holds the parameters for nesterov optimizer. We call nesterov_acceperated_gd to update the model weights after each batch which is 64 in our setup. momentum is first updated(\mathbf{m}_{t+1}) using the previous momentum(\mathbf{m}_t) * gamma along with batch gradient * step size. Nesterov is built over momentum, additionally it performs a look ahead for the expected W which helps to reduce oscillations around the local minimum.

Plots

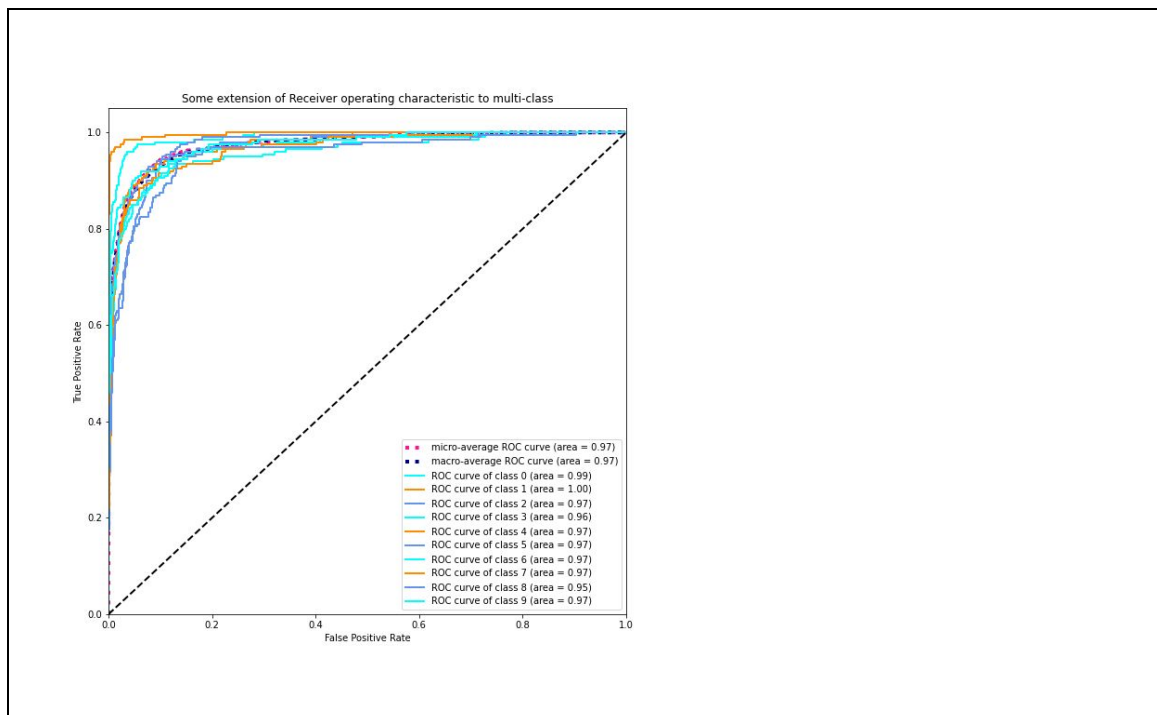
Accuracy vs Epoch



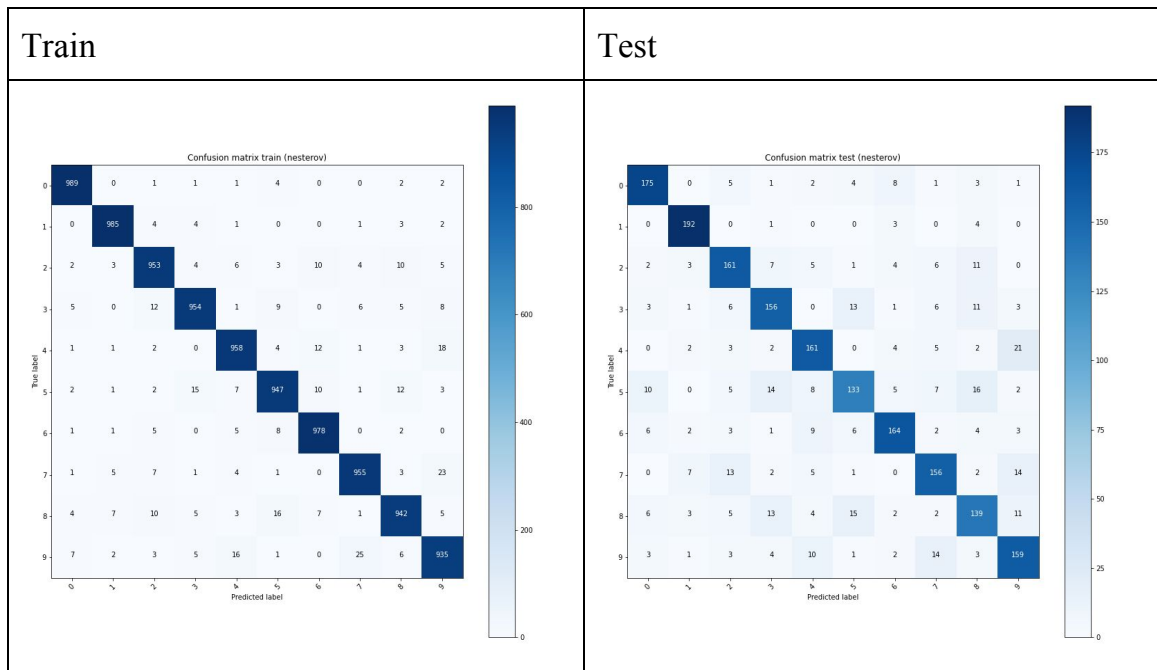
Loss vs Epoch



ROC curves (Val set)



Confusion Matrix of Validation set



Observations

Accuracy Stats for nesterov
 Train Accuracy: 95.96000000000001
 Train Accuracy: 79.80000000000001

=====

Optimizer: "nesterov"

Epochs: 450

Activation Fn(Hidden Layers): "tanh"

Activation Fn(Output Layer): "softmax"

Step size: 0.01

Weight initialization strategy: "random"

Regularization: "l2"

Dropout: 0.2

Batch size: 64

Layer 1: (128, 784)

Layer 2: (24, 128)

Layer 3: (10, 24)

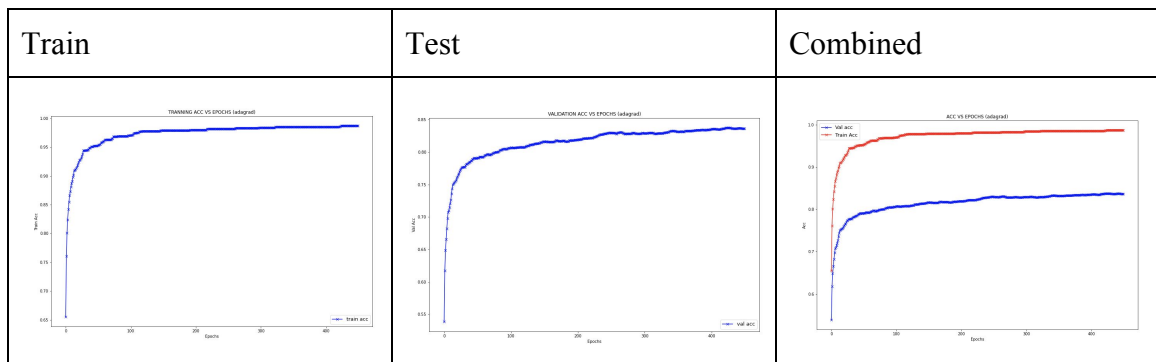
AdaGrad Optimizer:- Methodology

```
def adagrad(self):
    for ix in range(len(self._weights)):
        self._optimizer_weight[ix] += np.square(self._dw[ix])
        self._optimizer_bias[ix] += np.square(self._db[ix])
        self._weights[ix] -=
self._dw[ix]*self._learning_rate/np.sqrt(self._optimizer_weight[ix]+self
._eps)
        self._bias[ix] -=
self._db[ix]*self._learning_rate/np.sqrt(self._optimizer_bias[ix]+self._
eps)
```

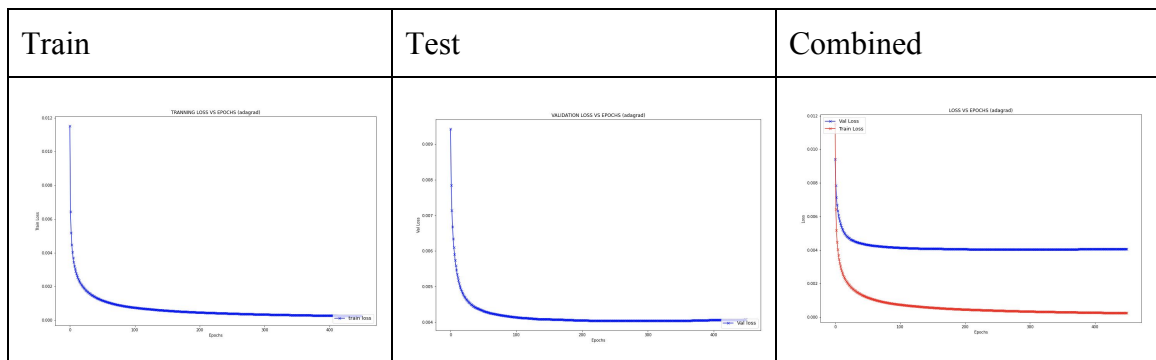
optimizer_weigh(G_t) holds the parameters for adagrad optimizer. Call to adagrad function updates the model weights after each batch which is 64 in our setup. adagrad parameters are first updated(G_{t+1}) using the square of the batch gradient. It helps adapt the learning parameters based on the occurrence of the features.

Plots

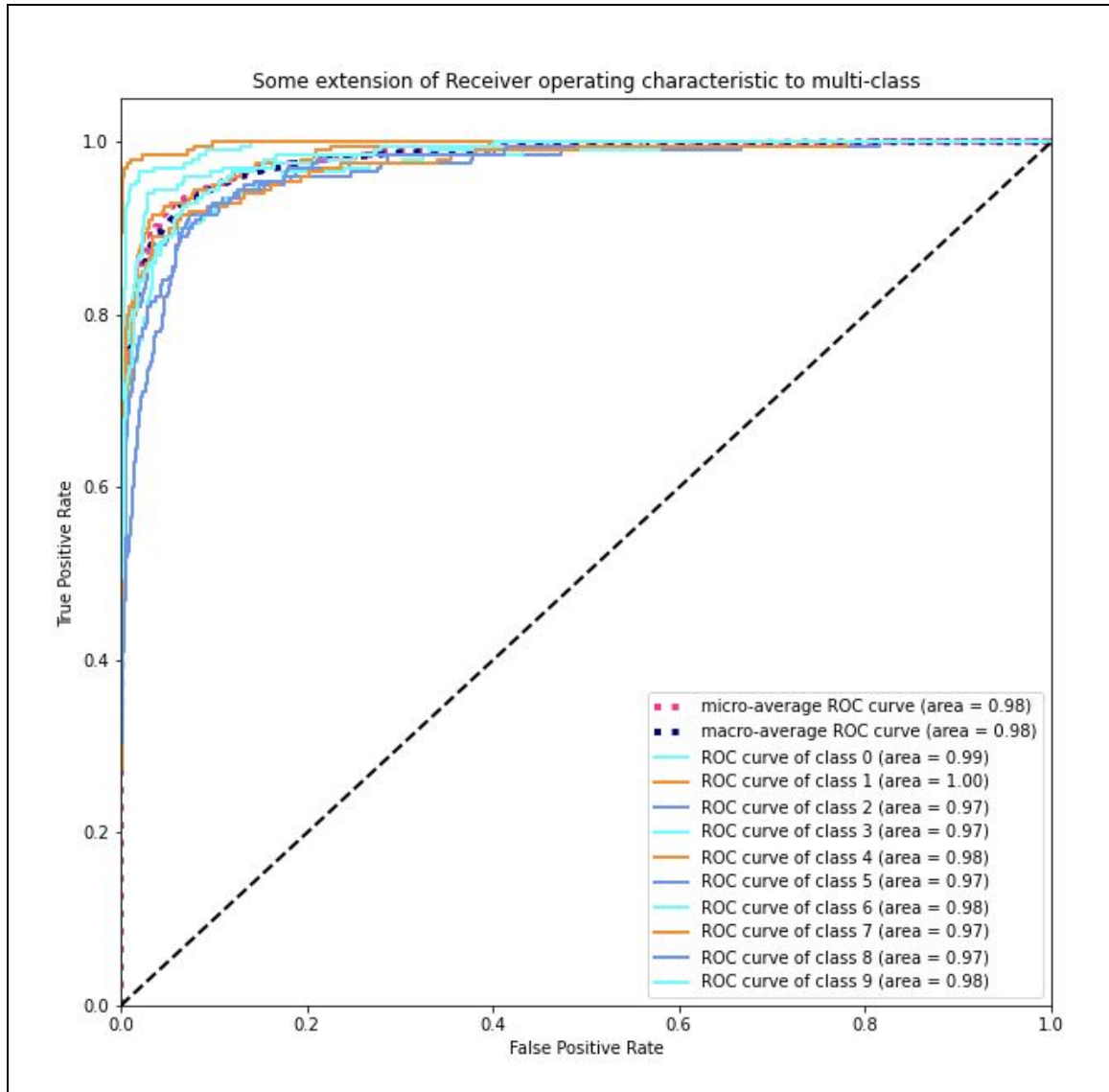
Accuracy vs Epoch



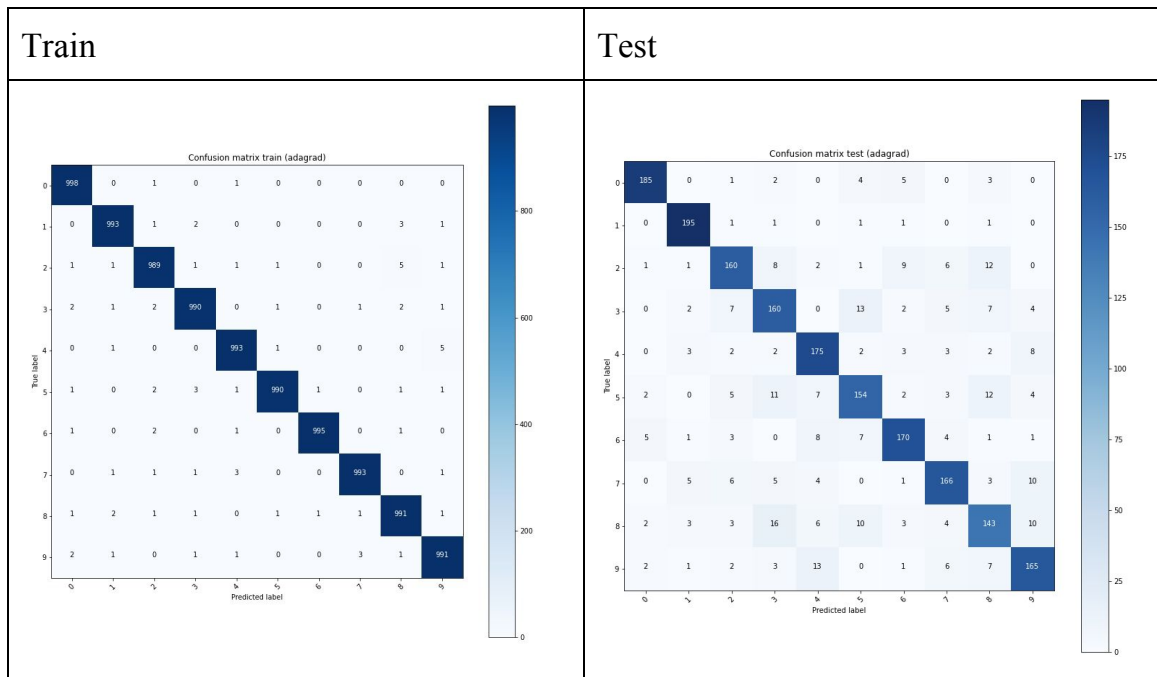
Loss vs Epoch



ROC curves (Val set)



Confusion Matrix of Validation set



Observations

Accuracy Stats for adagrad
 Train Accuracy: 99.22999999999999
 Train Accuracy: 83.65

=====

Optimizer: "adagrad"

Epochs: 450

Activation Fn(Hidden Layers): "tanh"

Activation Fn(Output Layer): "softmax"

Step size: 0.01

Weight initialization strategy: "random"

Regularization: "l2"

Dropout: 0.2

Batch size: 64

Layer 1: (128, 784)

Layer 2: (24, 128)

Layer 3: (10, 24)

RMSprop Optimizer:-

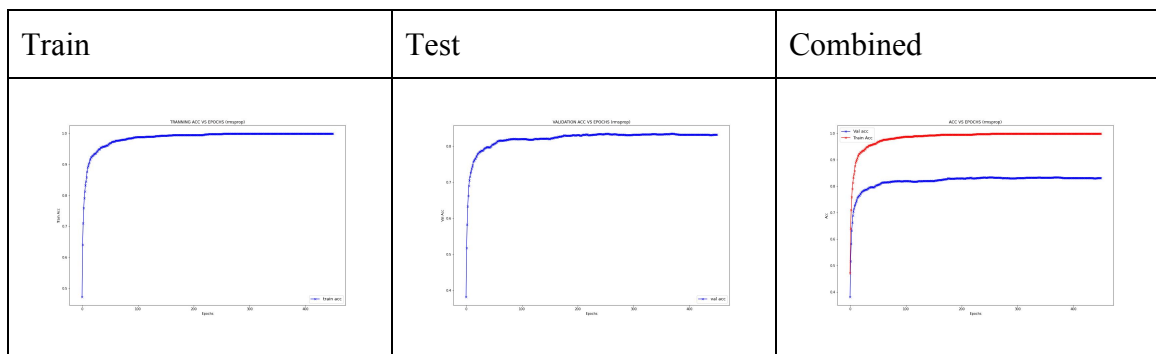
Methodology

```
def rmsprop(self):
    for ix in range(len(self._weights)):
        self._optimizer_weight[ix] = self._optimizer_weight[ix]*self._beta
        + (1-self._beta)*self._dw[ix]*self._dw[ix]
        self._optimizer_bias[ix] = self._optimizer_bias[ix]*self._beta +
        (1-self._beta)*self._db[ix]*self._db[ix]
        self._weights[ix] -=
        (self._dw[ix]*self._learning_rate)/np.sqrt(self._optimizer_weight[ix]+self._eps)
        self._bias[ix] -=
        (self._db[ix]*self._learning_rate)/np.sqrt(self._optimizer_bias[ix]+self._eps)
```

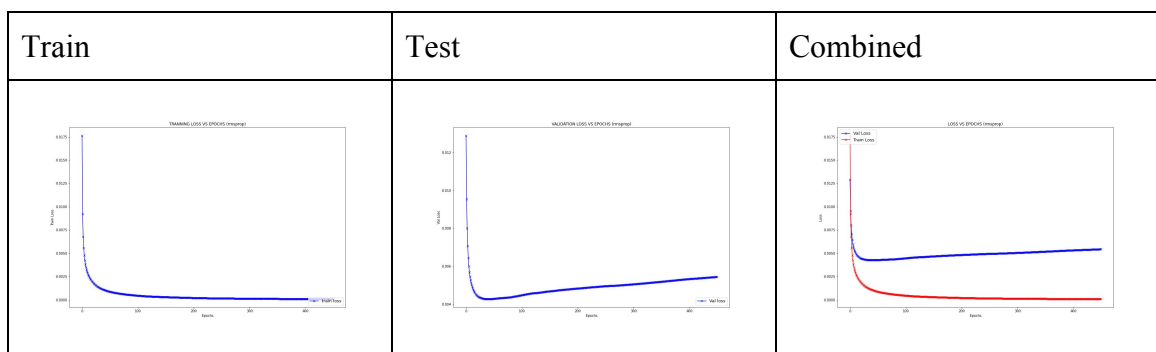
optimizer_weigh(E_t) holds the parameters for adagrad optimizer. Call to rmsprop function updates the model weights after each batch which is 64 in our setup. RMSProp similar to adagrad also takes the squared batch gradient but it avoids the accumulation problem using the weighted average. Model parameters are updated using the decaying rmsprop parameters. Similar to adagrad, it helps adapt the learning parameters based on the occurrence of the features.

Plots

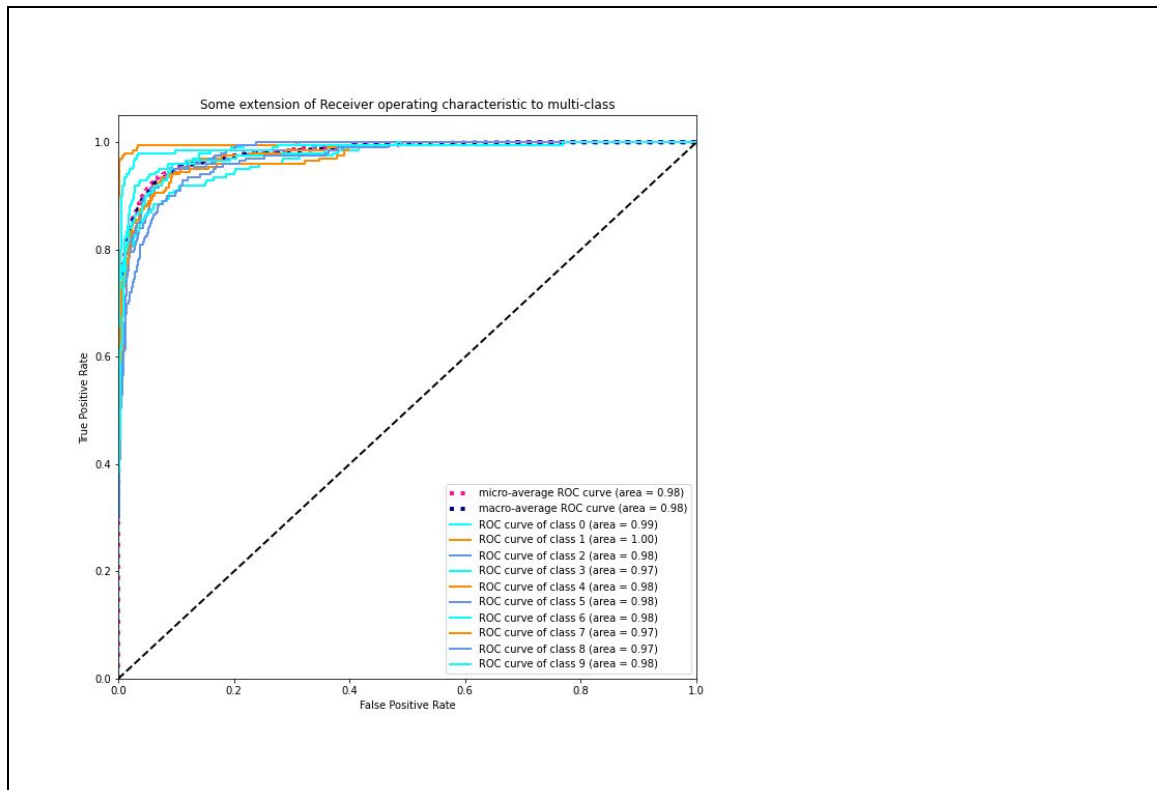
Accuracy vs Epoch



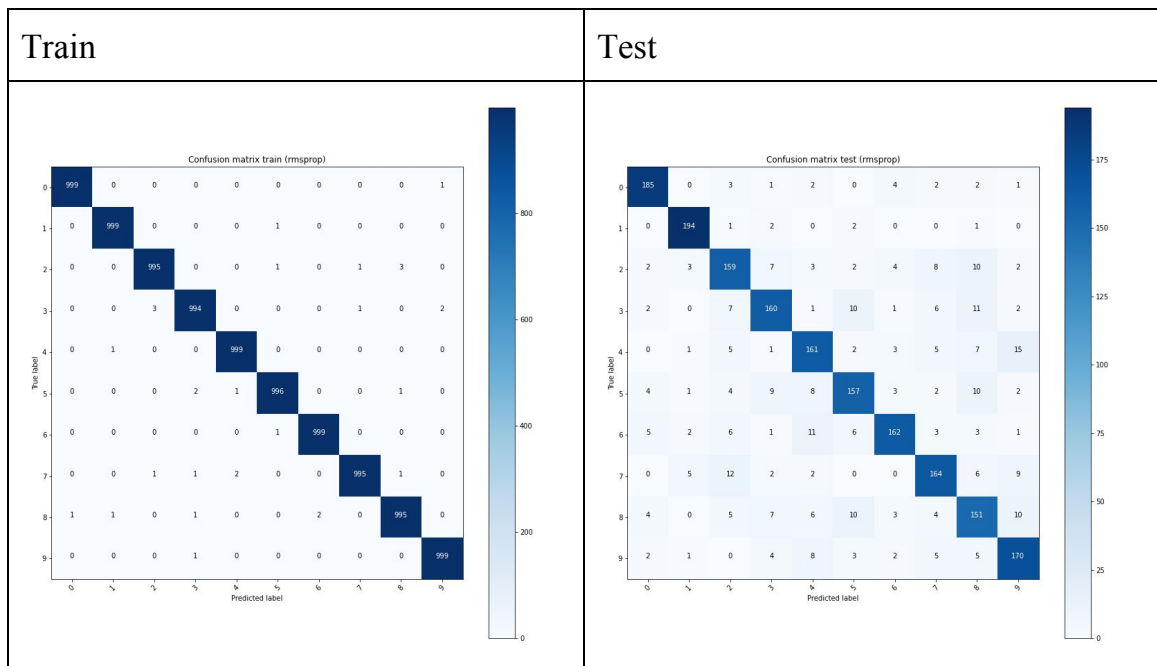
Loss vs Epoch



ROC curves (Val set)



Confusion Matrix of Validation set



Observations

Accuracy Stats for rmsprop
Train Accuracy: 99.7
Train Accuracy: 83.15

=====

Optimizer: "rmsprop"

Epochs: 450

Activation Fn(Hidden Layers): "tanh"

Activation Fn(Output Layer): "softmax"

Step size: 0.01

Weight initialization strategy: "random"

Regularization: "l2"

Dropout: 0.2

Batch size: 64

Layer 1: (128, 784)

Layer 2: (24, 128)

Layer 3: (10, 24)

Adam Optimizer:- Methodology

```
def adam(self, batch_no):
    for ix in range(len(self._weights)):
        n_beta1 = 1/(1-np.power(self._beta, batch_no+1))
        n_beta2 = 1/(1-np.power(self._beta2, batch_no+1))

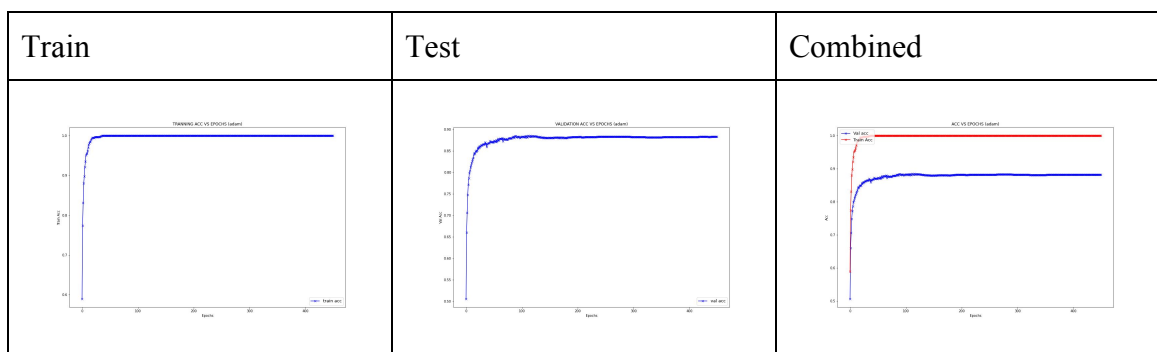
        self._optimizer_weight[ix] = self._optimizer_weight[ix]*self._beta
+ (1-self._beta)*self._dw[ix]
        self._optimizer_bias[ix] = self._optimizer_bias[ix]*self._beta +
(1-self._beta)*self._db[ix]

        self._optimizer2_weight[ix] =
self._optimizer2_weight[ix]*self._beta2 +
(1-self._beta2)*self._dw[ix]*self._dw[ix]
        self._optimizer2_bias[ix] = self._optimizer2_bias[ix]*self._beta2
+ (1-self._beta2)*self._db[ix]*self._db[ix]

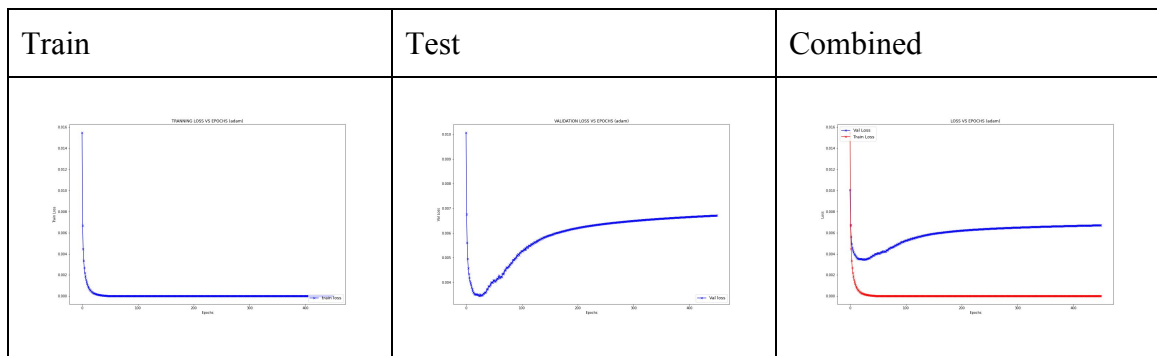
        self._weights[ix] -=
(self._optimizer_weight[ix]*self._learning_rate*n_beta1)/(np.sqrt(self._
optimizer2_weight[ix]*n_beta2)+self._eps)
        self._bias[ix] -=
(self._optimizer_bias[ix]*self._learning_rate*n_beta1)/(np.sqrt(self._op
timizer2_bias[ix]*n_beta2)+self._eps)
```

Plots

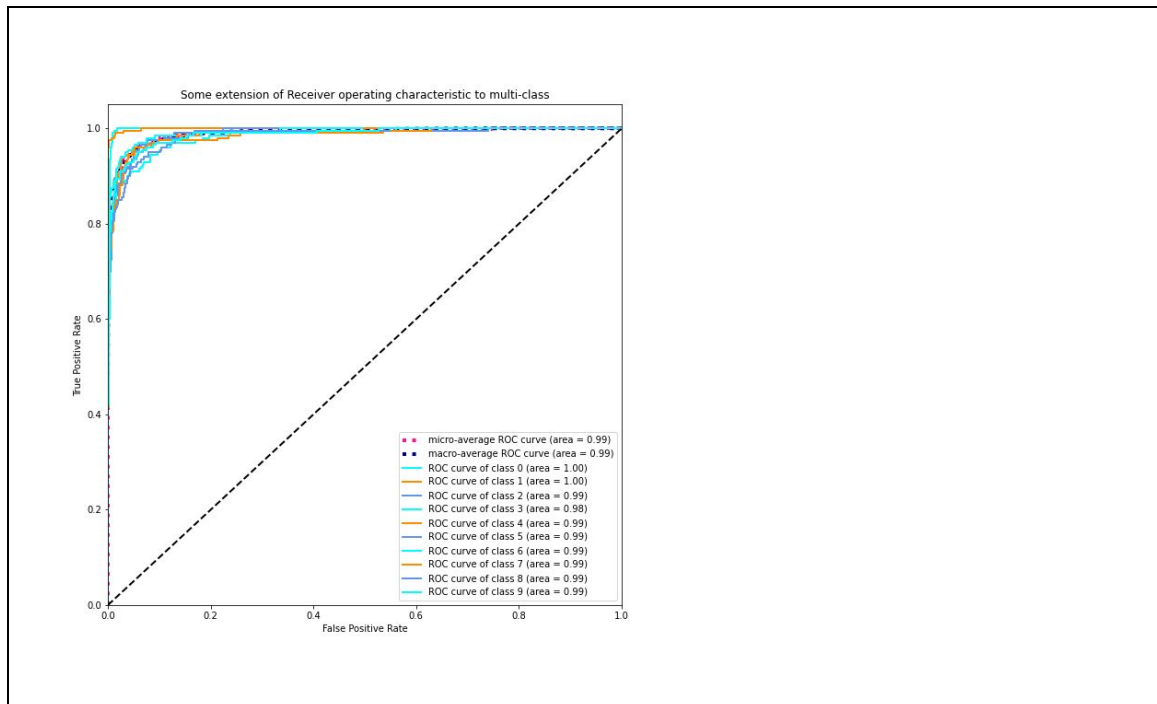
Accuracy vs Epoch



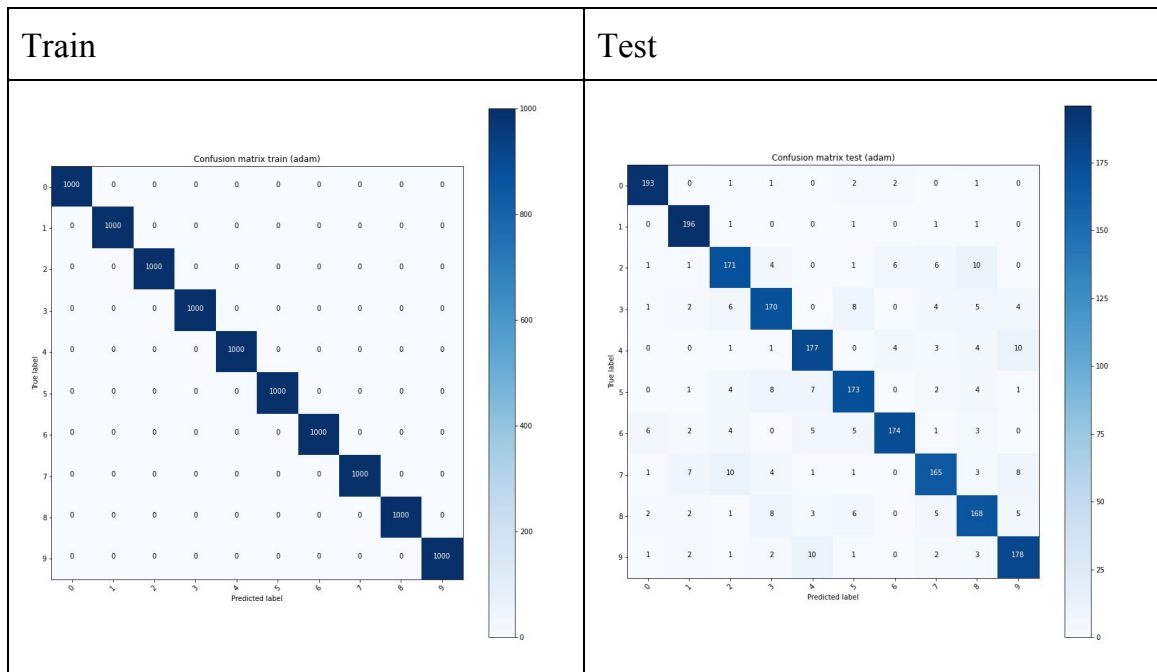
Loss vs Epoch



ROC curves (Val set)



Confusion Matrix of Validation set



Observations

Accuracy Stats for adam

Train Accuracy: 100.0

Train Accuracy: 88.25

=====

Optimizer: "adam"

Epochs: 450

Activation Fn(Hidden Layers): "tanh"

Activation Fn(Output Layer): "softmax"

Step size: 0.01

Weight initialization strategy: "random"

Regularization: "l2"

Dropout: 0.2

Batch size: 64

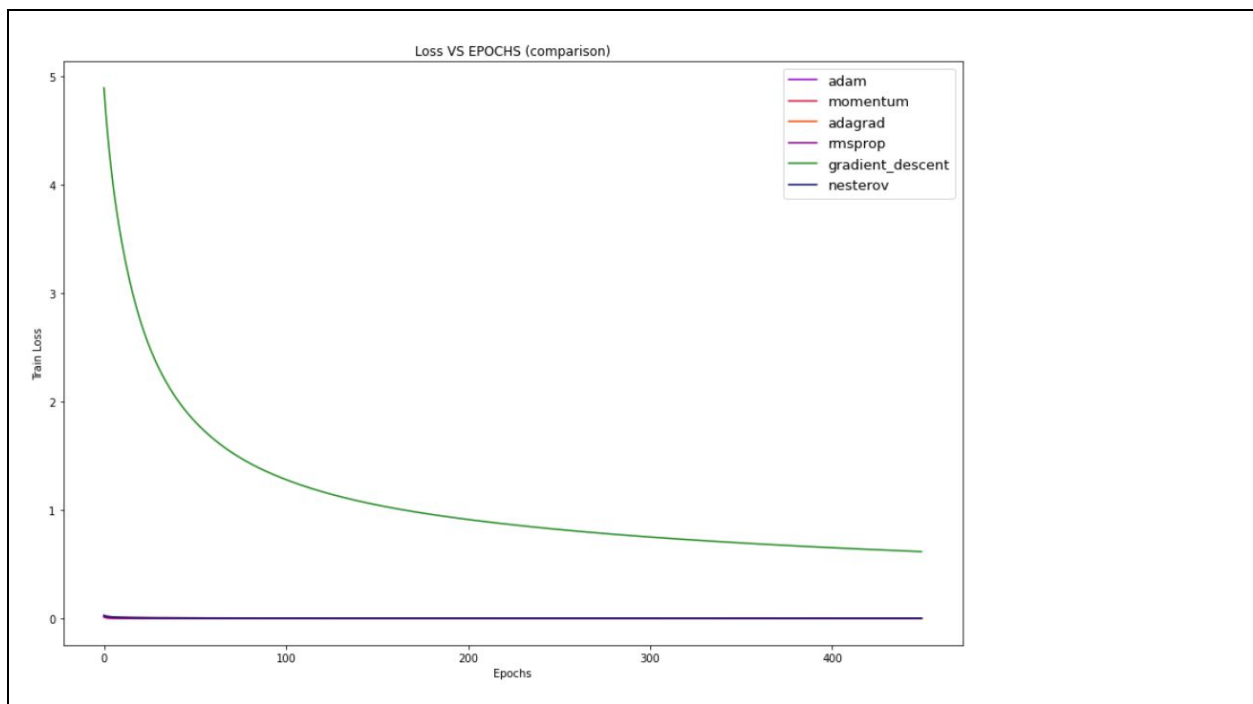
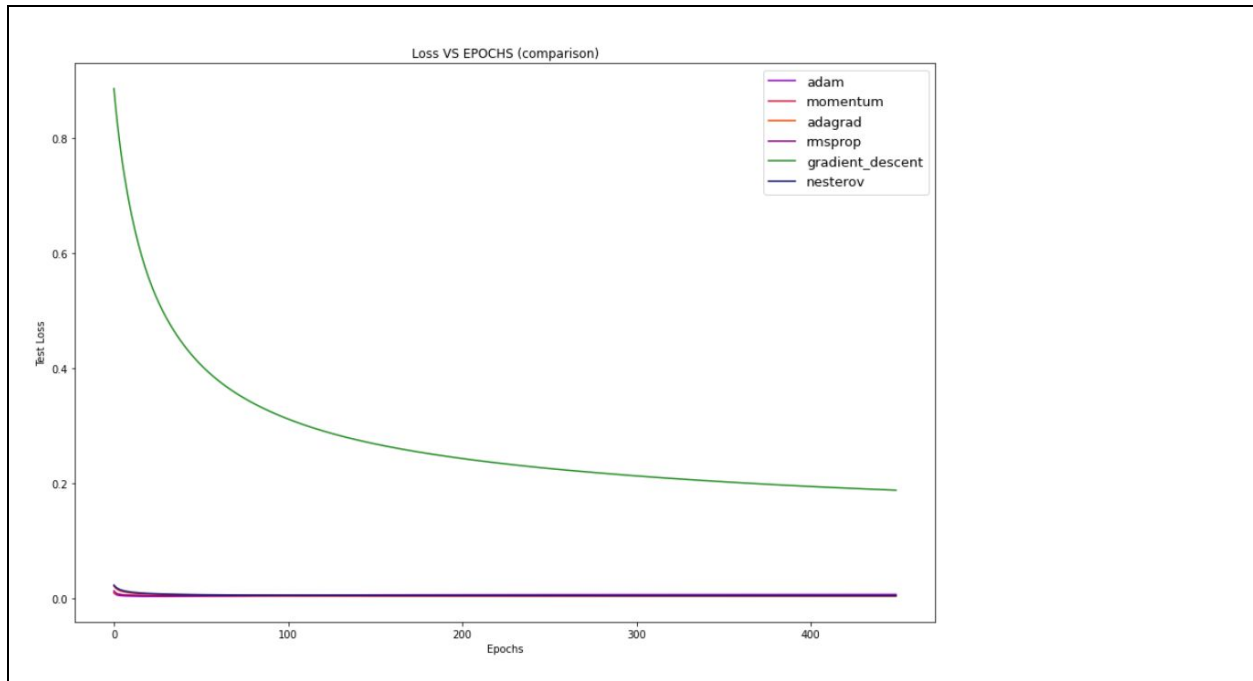
Layer 1: (128, 784)

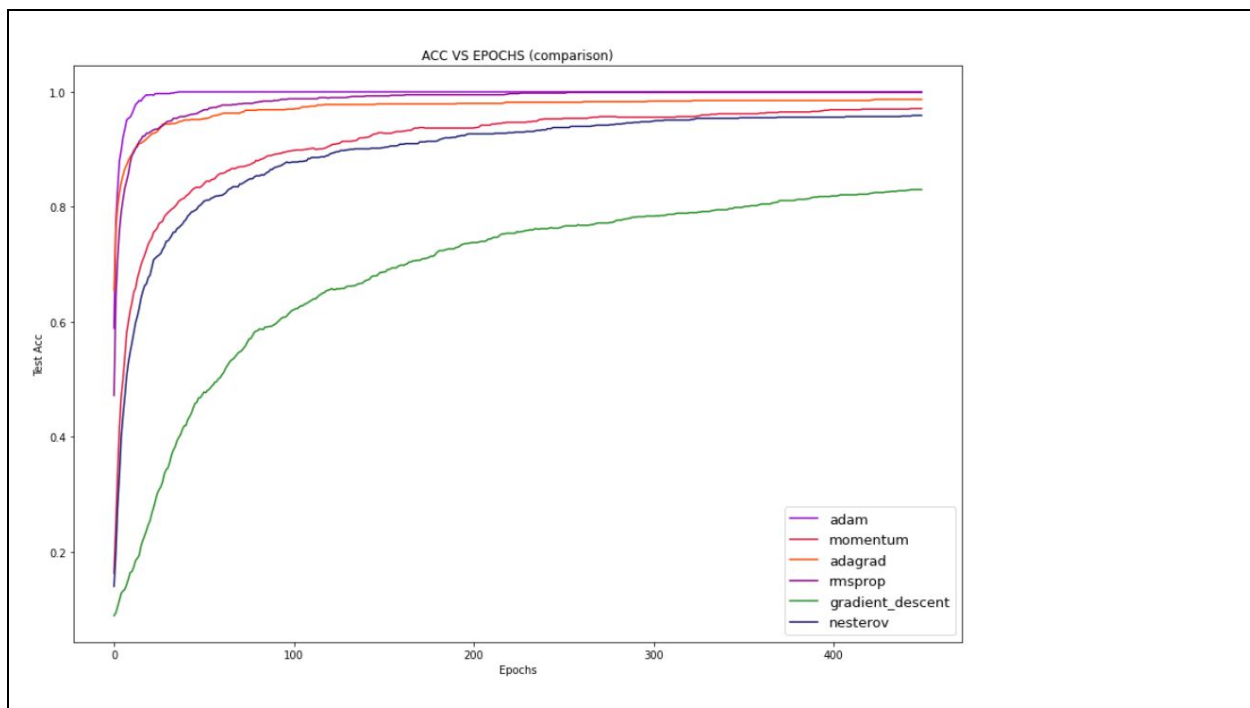
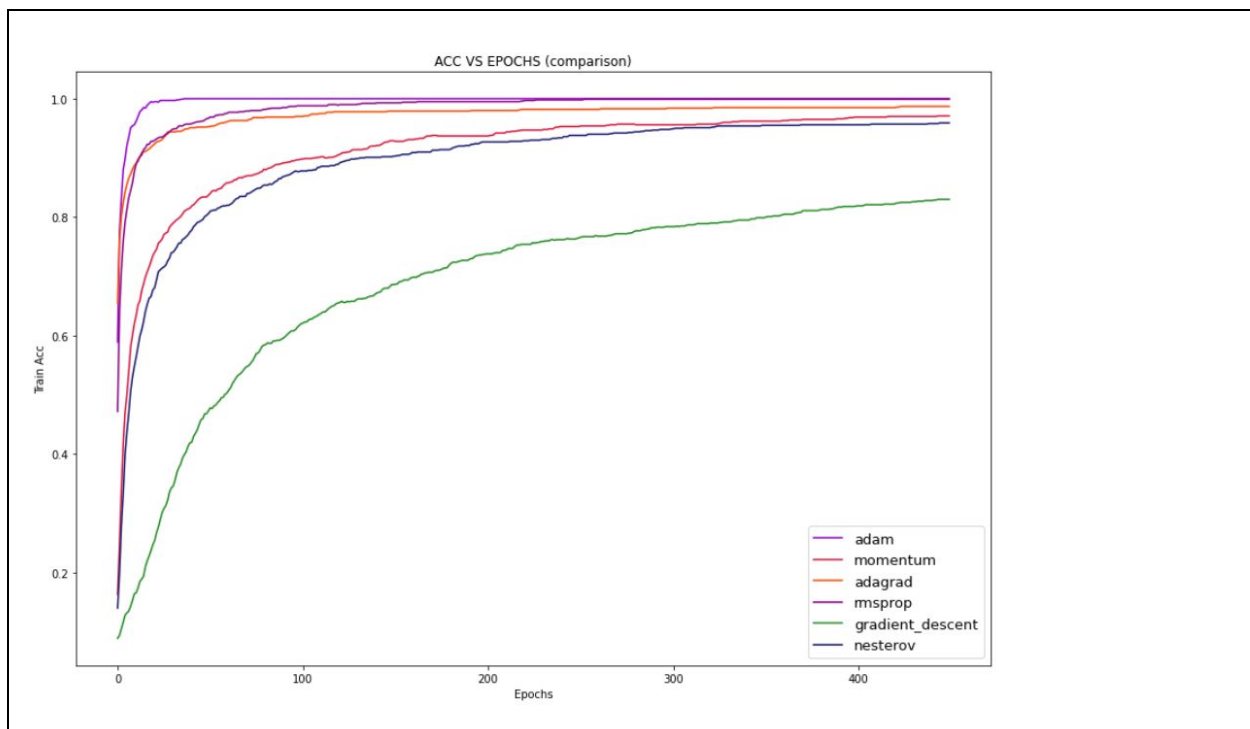
Layer 2: (24, 128)

Layer 3: (10, 24)

2.2 Best Optimizer and Model Comparisons

BEST OPTIMIZER AND MODEL :-





Our Best mode is Adam with the following architecture and accuracies :-

```
Accuracy Stats for adam
Train Accuracy: 100.0
Train Accuracy: 88.25
```

```
=====
Optimizer: "adam"
```

```
-----
Epochs: 450
```

```
-----
Activation Fn(Hidden Layers): "tanh"
```

```
-----
Activation Fn(Output Layer): "softmax"
```

```
-----
Step size: 0.01
```

```
-----
Weight initialization strategy: "random"
```

```
-----
Regularization: "l2"
```

```
-----
Dropout: 0.2
```

```
-----
Batch size: 64
```

```
-----
Layer 1: (128, 784)
```

```
-----
Layer 2: (24, 128)
```

```
-----
Layer 3: (10, 24)
```

Adam is basically a combination of RMS Prop and Momentum. Both Momentum and RMS Prop focus on different aspects to improve the convergence of models. Momentum accelerates the convergence towards the relevant direction and reduces the fluctuation to the irrelevant direction. And Rms prop helps in manually tuning the learning rate with the idea of slowing down as we reach close to the global minima.

Thus the combination of both Adam out performs by tuning learning rate and at the same time moving in the right direction.

OBSERVATIONS :-

1. Adam performs the best because it is a combination of Rms Prop and Momentum
2. Gradient Descent performs the worst among all the models because the learning rate is constant which is opposite to the fact that it is always good to keep a higher learning rate initially and then reduce it further as one reaches close to minimum. Also it may be the case that gradient descent gets stuck in local minima and is not able to get out of it and thus not able to achieve the highest accuracy.
3. In our case Momentum and Nesterov almost perform equally, though theoretically Nesterov is able to converge quickly because of the look ahead concept in it, But here in our cost function both performance is equally good.
Compared to gradient descent they both are better because they use weighted averaging and so are able to avoid the local minima and move in proper direction with less damping.
4. After momentum and Nesterov, AdaGrad gives better performance as it eliminates the need to manually tune the learning rate.
5. RmsProp is better compared AdaGrad as it overcomes one flaw in adagrad. The accumulated sum keeps growing during training in adagrad. This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge. Rms Prop solves this problem.
6. In the end Adam performs the best.
Adam optimizer is highly versatile, and works in a large number of scenarios. It takes the best of RMS-Prop and momentum, and combines them together.
As it can be anticipated, Adam accelerates and decelerates thanks to the component associated with momentum. At the same time, Adam keeps its learning rate adaptive which can be attributed to the component associated to RMS-Prop.

Contribution of Each Member

1. Shivam Sharma

Implemented Optimizers and Plots code and network class

2. Akanksha Shrimal

Implemented Gradient Descent with back propagation.

3. Vaibhav Goswami

Helped in report and Optimizers code