Eeshaan Ravi Tivari (2019465)
Manvi Goel (2019472)

OUTPUT
Assignment 1

CSE641 - Deep Learning

Question 1

Assumptions

1. Input is binary i.e. 0s and 1s
2. The maximum number of epochs can be changed.
3. If the weights are repeated inside the loop then the model is converged.

Working

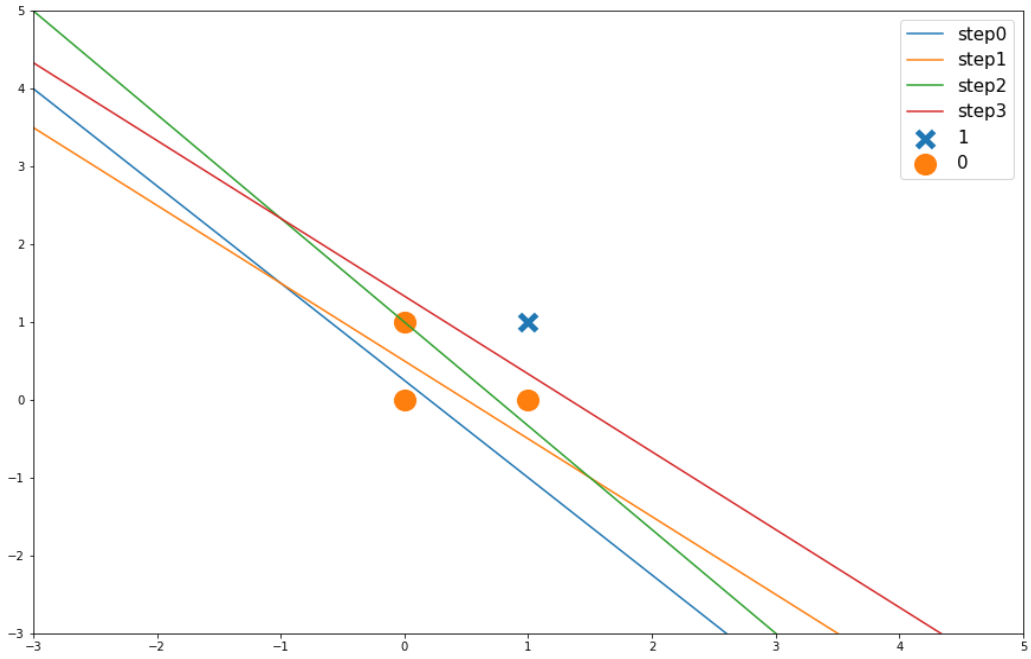
1. Calculate the predicted output using weights and apply the threshold function
2. For all samples that are misclassified update the weights.
3. Recalculate the outputs.
4. Repeat the process until the weights repeat or no training samples are misclassified.
5. If no training samples are misclassified the training algorithm terminates else if weights repeat, then there is no point to continue training as similar weights will come again and again indicating that the perceptron is unable to learn which is why it breaks out of this and indicate its inability to learn.

For AND :-

Initial Weights : *{4, 5, -1}* Final Weights: *{3, 3, -4}*

No. of epochs taken: *3*

Decision Boundary for AND

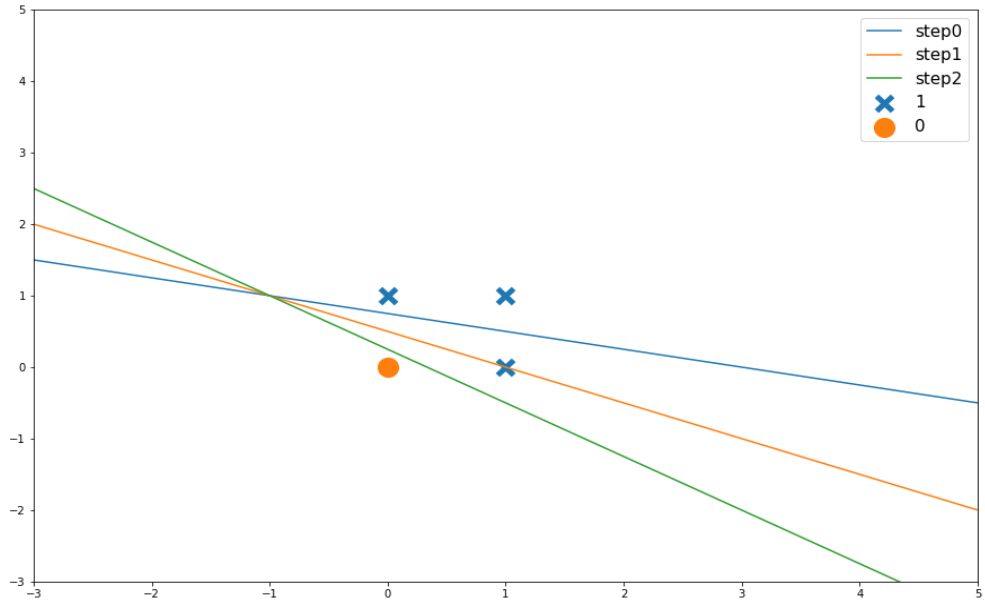


For OR: -

Initial Weights : *{4, 1, -3}* Final Weights: *{4, 3, -1}*

No. of epochs taken: *2*

Decision Boundary for OR

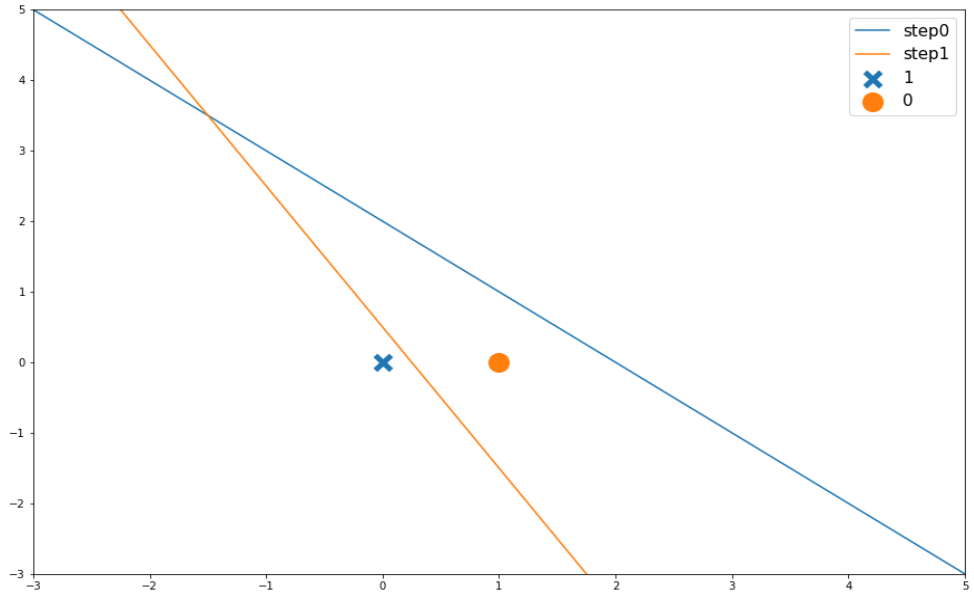


For NOT:

Initial weights: *{-1, 2}* Final Weights: *{-2, 1}*

No. of epochs taken: *1*

Decision Boundary for OR

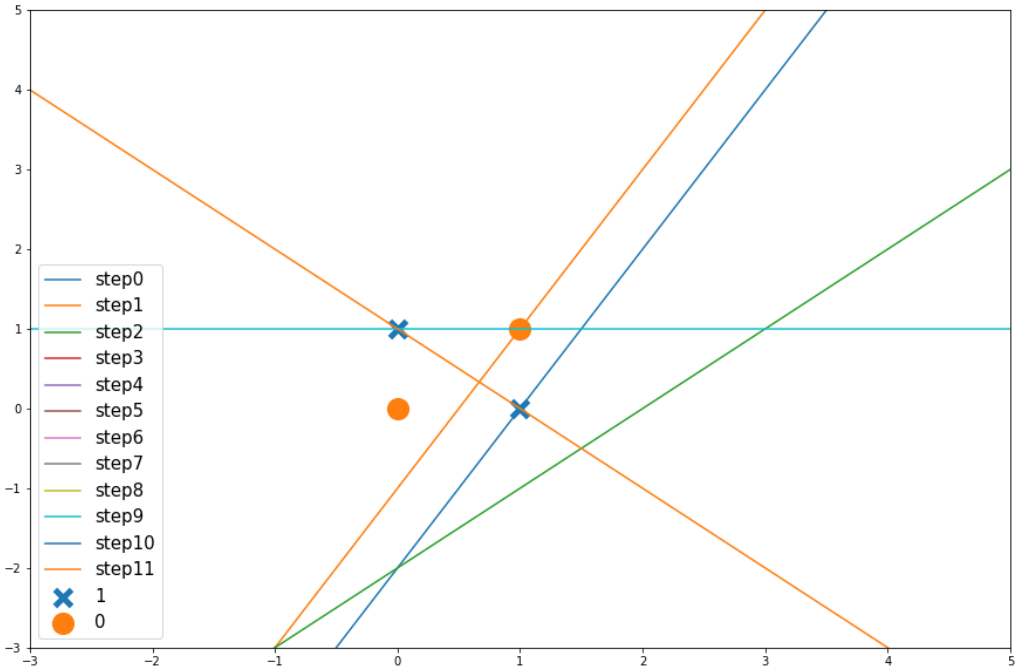


For XOR:-

Initial weights: *{1, -2, 2}* No of epochs taken to identify: *12*

Final Weights: *None* as the model doesn't converge

Decision Boundary for XOR



Explanation for XOR

We can see in the plot that the points are not linearly separable, hence a linear decision boundary can not separate them. Since PTA can only make linear boundaries, it is unable to separate the points and decision boundaries can not be set which is clearly indicative from the plot as out of the shown boundaries none is separating the data points.

In this example for PTA, we see that the weights were repeated after **12** steps, and since repeated weights imply that the steps would start repeating we can say that the model will not converge for XOR as perceptron will not be able to learn by repeating the same set of weights again and again.

Question 2

Assumptions

1. The model contains only one hidden layer, while the number of neurons in the hidden layer can be changed.
2. The model will run for a maximum of 200 steps.
3. The output of the training data is 0 and 1.

Working

1. Adaline.
   1.1 It uses an affine combination of inputs and weights and a threshold activation function in the forward pass. For each sample in the training data, if the output is incorrect, that is, the sample is misclassified then the weights are updated using learning rate and desired output.

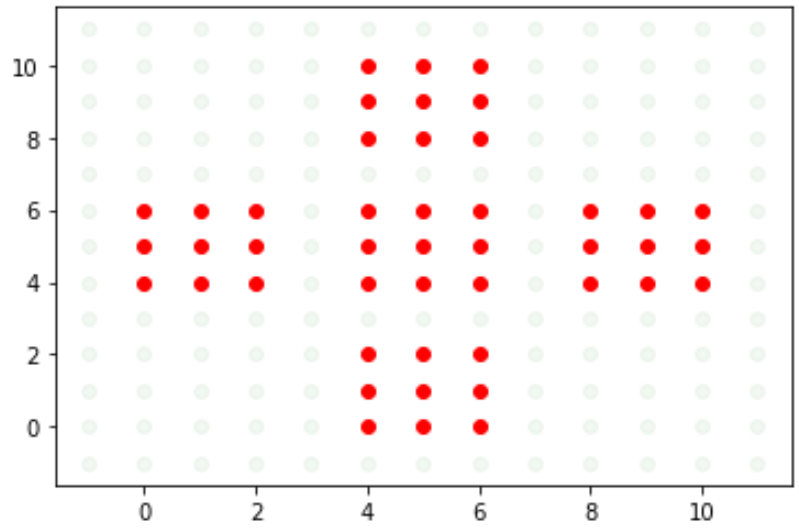
2. Madaline.
   2.1 It constructs a hidden layer with the number of adalines mentioned in the function. It uses the output of the hidden layer adalines to calculate the final output using an affine combination.

   2.2 If any sample is misclassified, then greedily select the combination of adalines with a minimum confidence level that improves the accuracy of the model and updates the weights. Two models, one using all combinations and another only using 1 neuron at a time are given for comparison.

   2.3 Run the model till all samples are correctly classified or the loop has run for 1e7 steps.

Outputs.

Training Data



The output for the madaline model.

The number of hidden adalines required is 21. This number gives the best accuracy.

The final accuracy of the model is 130 correctly classified out of 144 points or 90.2% accuracy.

Final Weights -

```
Training Complete
Model Converged
130
Final Weights
[ 0.35346359 -0.16746847  0.5589933 ]
[ 0.55887649  0.08289261 -0.06276473]
[0.37561918 0.27065409 0.400171  ]
[0.46984675 0.40976294 0.12804003]
[0.04898675 0.52174961 0.72504231]
[0.35393984 0.35483514 0.28002609]
[0.35700889 0.05513377 0.23309953]
[ 0.23703429 -0.00833473  0.45345687]
[0.18701586 0.15341378 0.47888035]
[0.27319916 0.27876397 0.36845797]
[ 0.28411773 -0.10521353  0.29346734]
[ 0.32424547 -0.09907642  0.24727731]
[0.36117453 0.36901519 0.26632185]
[0.11634801 0.41202563 0.60922041]
[0.19526746 0.60810423 0.54085569]
```

We can see from the updates weights, that given enough computation power the method can classify the points if enough neurons are given. But given the large number of adalines, the method is not computationally efficient.
This model is chosen after experiments for accuracy using different layer sizes and learning rates and thresholds.

A better estimate for the model can be obtained using XOR data which can be correctly classified using 2 hidden neurons completely.

```
Training Complete
Model Converged
4
Final Weights
[ 0.21658609 -0.19445202  0.47672022]
[ 0.20566966  0.29443695 -0.00040567]

0
1
1
0
```

(b) We can *not* classify all the sample points correctly in only two neurons, which can be seen from the model, which fails to classify the data even after exhausting all its possibilities. The number of neurons required is related to the decision boundaries, hence 2 neurons are not sufficient to represent the function that employs more than two decision boundaries. In the model, the 2 neurons would adapt to always output 0, as to maximize the accuracy but the function remains unlearned.

The 2 neurons can only divide the space into 3 regions, which is not sufficient for this function which has 5 bounded squares regions.

At least 12 neurons are required in the first hidden layer for the model to correctly classify the samples.

Question 3

Assumptions

1. The layers, sizes, optimization, activation functions can all be changed.
2. The input layer has a size equal to the input and the output layer is the number of classes in the training data.
3. $\varepsilon = 0.0000001$, $\gamma = 0.9$ and $\beta = 0.9$ are assumed for the below expression and their equivalent usage

   - RMSProp with momentum

$$m^{(t)} = \beta m^{(t-1)} - \eta \nabla J(w^{(t)})$$

$$v^{(t)} = \gamma v^{(t-1)} + (1-\gamma) \nabla J(w^{(t)})^2$$

Bias correction →

$$\hat{m}^{(t)} = \frac{m^{(t)}}{1-\beta^t} \qquad \hat{v}^{(t)} = \frac{v^{(t)}}{1-\gamma^t}$$

$$w^{(t+1)} = w^{(t)} - \frac{\eta}{\sqrt{v^{(t)} + \epsilon}} m^{(t)}$$

$$w^{(t+1)} = w^{(t)} - \frac{\eta}{\sqrt{\hat{v}^{(t)} + \epsilon}} \hat{m}^{(t)}$$

4. Default optimizer for the network will bt gradient descent optimizer
5. All momentums and learning rate scale factors in the case of adaptive learning rate optimizers are initialized with 0 value.
6. The loss function is cross-entropy loss function.
7. Training will terminate only after running for epoch no. of times not before that

Working

1. **Multi-layer perceptron**

   1. A class named MyNeuralNetwork() is defined in toolkit.py which defines our multi-layer perceptron. It initializes the weights, bias, and other hyperparameters of the network.
   2. To use an MLP classifier, a class object is made and that object is used for training the network.
   3. For training the network, training and validation data are passed as function parameters to the fit() function of class MyNeuralNetwork() which runs the training algorithm for epoch no. of times for every batch as per batch size.

4. For every batch first, forward propagation is done, then backward propagation is to calculate the gradients for network weights and bias
5. After that weights and bias are updated using update_network_parameters() function of class MyNeuralNetwork().
6. The validation and training data accuracy for every epoch are calculated and appended to their respective lists which are returned by the fit() after completing execution.
7. For predicting, test samples are passed as function parameters to predict() function of class MyNeuralNetwork() which returns the prediction for these samples.

2. Optimizers

1. The default Optimizer for class MyNeuralNetwork() is gradient descent, to use another optimizer, it has to be specified in hyperparameter 'optimizers' while making the object of MyNeuralNetwork().

GD	-	for Gradient descent optimizer
momentum	-	for momentum optimizer
NAG	-	for Nesterov's accelerated gradient optimizer
RMS	-	for RMSProp optimizer
AdaGrad	-	for AdaGrad optimizer
Adam	-	for Adam optimizer

2. Using this, different models for different optimizers are made, trained and saved.

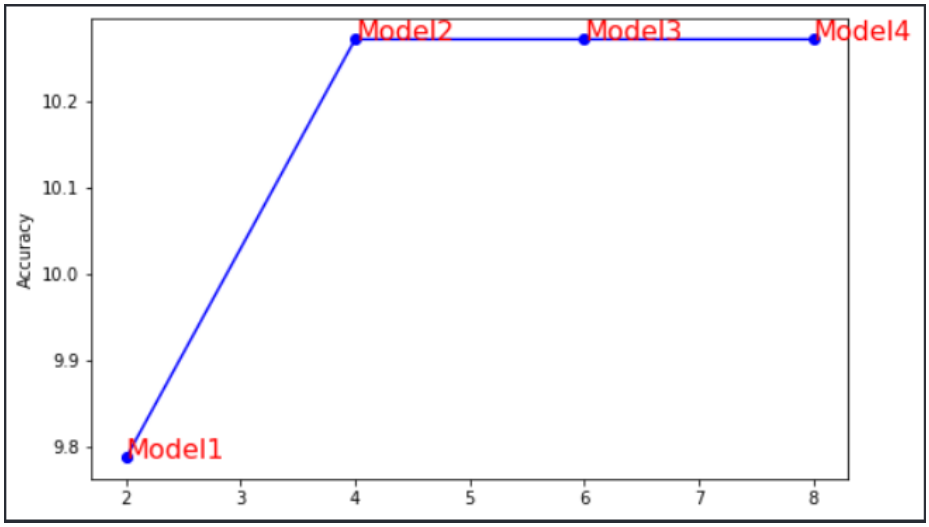
Experimentation

We have created various models for gradient descent optimizer and selected the best model configuration from them which gives the best test set accuracy, The experiments are as followed-

1. Different learning rates for *tanh* activation

```
Experimental_Nets = [
                MyNeuralNetwork(4, [784, 256, 64, 10], "tanh", 0.001, "random", X_train.shape[0], num_epochs =15,optimizers = "GD"),
                MyNeuralNetwork(4, [784, 256, 64, 10], "tanh", 0.01, "random", X_train.shape[0], num_epochs =15, optimizers = "GD"),
                MyNeuralNetwork(4, [784, 256, 64, 10], "tanh", 0.1, "random", X_train.shape[0], num_epochs=15,optimizers = "GD"),
                MyNeuralNetwork(4, [784, 256, 64, 10], "tanh", 1, "random", X_train.shape[0], num_epochs=15,optimizers = "GD")
]
```

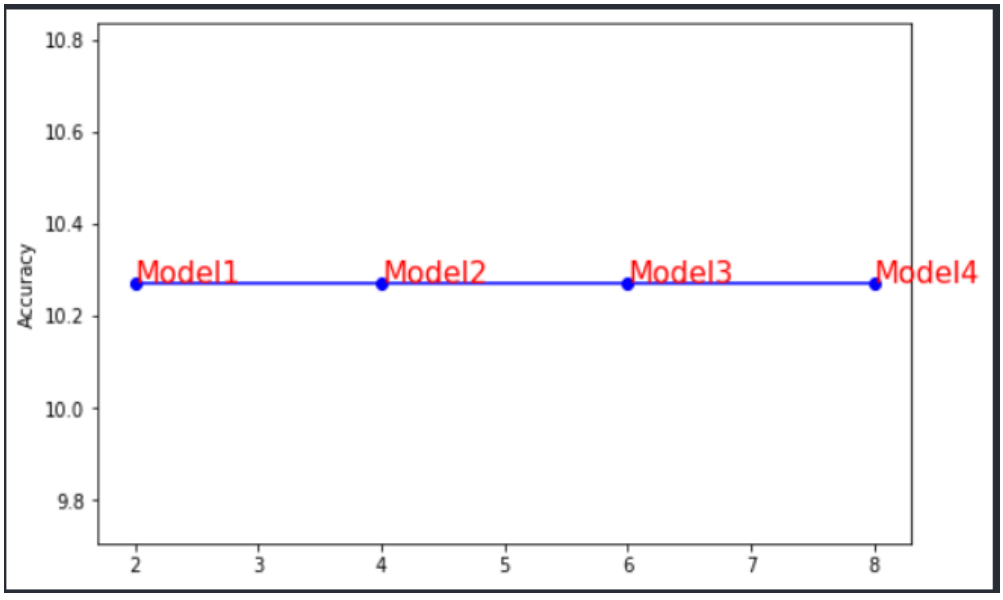
Plot for accuracies for above models



2. Different learning rates for *Relu* activation

```
Experimental_Nets = [
                    MyNeuralNetwork(4, [784, 256, 64, 10], "relu", 0.001, "random", X_train.shape[0], num_epochs =15,optimizers = "GD"),
                    MyNeuralNetwork(4, [784, 256, 64, 10], "relu", 0.01, "random", X_train.shape[0], num_epochs =15, optimizers = "GD"),
                    MyNeuralNetwork(4, [784, 256, 64, 10], "relu", 0.1, "random", X_train.shape[0], num_epochs=15,optimizers = "GD"),
                    MyNeuralNetwork(4, [784, 256, 64, 10], "relu", 1, "random", X_train.shape[0], num_epochs=15,optimizers = "GD")
]
```

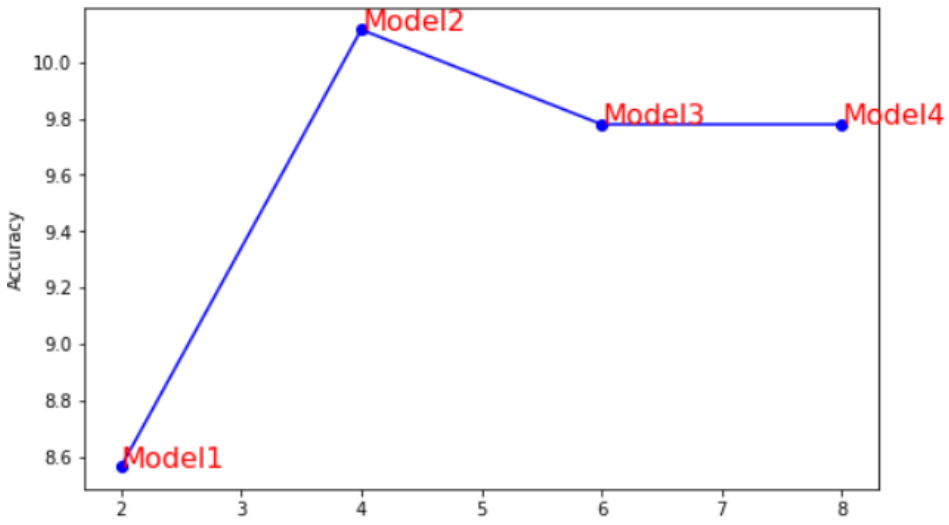
Plot for accuracies for above models



3. Different learning rates for *sigmoid* activation

```
Experimental_Nets = [
                    MyNeuralNetwork(4, [784, 256, 64, 10], "sigmoid", 0.001, "random", X_train.shape[0], num_epochs =15,optimizers = "GD
                    MyNeuralNetwork(4, [784, 256, 64, 10], "sigmoid", 0.01, "random", X_train.shape[0], num_epochs =15, optimizers = "GD
                    MyNeuralNetwork(4, [784, 256, 64, 10], "sigmoid", 0.1, "random", X_train.shape[0], num_epochs=15,optimizers = "GD"),
                    MyNeuralNetwork(4, [784, 256, 64, 10], "sigmoid", 1, "random", X_train.shape[0], num_epochs=15,optimizers = "GD")
]
```

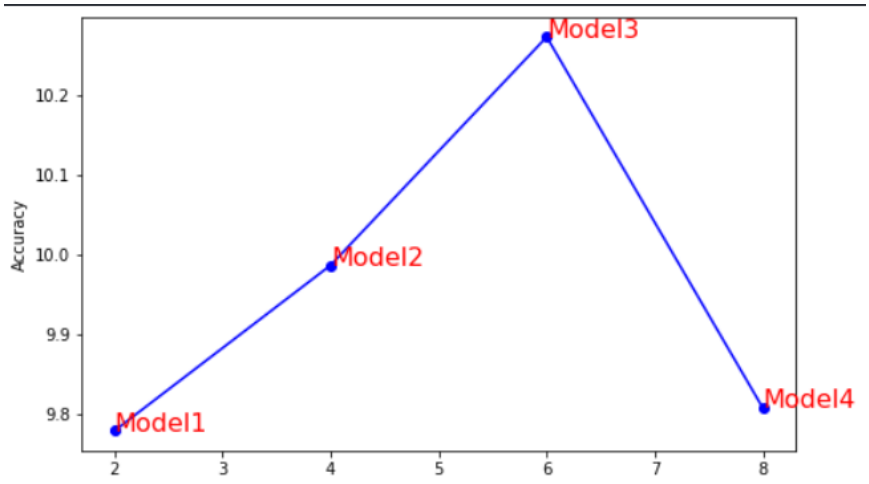
Plot for accuracies for above models



4. Different Hidden layer architecture (Shallow network)

```
Experimental_Nets = [
                    MyNeuralNetwork(3, [784, 256, 10], "tanh", 0.001, "random", X_train.shape[0], num_epochs =15, optimizers = "GD"),
                    MyNeuralNetwork(3, [784, 256, 10], "sigmoid", 0.001, "random", X_train.shape[0], num_epochs =15, optimizers = "GD"),
                    MyNeuralNetwork(3, [784, 256, 10], "tanh", 0.01, "random", X_train.shape[0], num_epochs =15, optimizers = "GD"),
                    MyNeuralNetwork(3, [784, 256, 10], "sigmoid", 0.01, "random", X_train.shape[0], num_epochs =15, optimizers = "GD"),
]
```

Plot for accuracies for above models



Best model configuration for Gradient Descent:

Activation function = sigmoid , learning rate = 0.001
Hidden Layer = [256 neurons] (single layer)

Since relu models are not learning anything from the start and got stuck at a performance, they are out. For tanh and sigmoid we found tanh gives same highest performance with shallow nets followed by shallow net of sigmoid function. So, we tried both model with different optimizer and found that tanh is not improving after achieving 10.37% which is why the final best model configuration is given by shallow network with sigmoid activation.

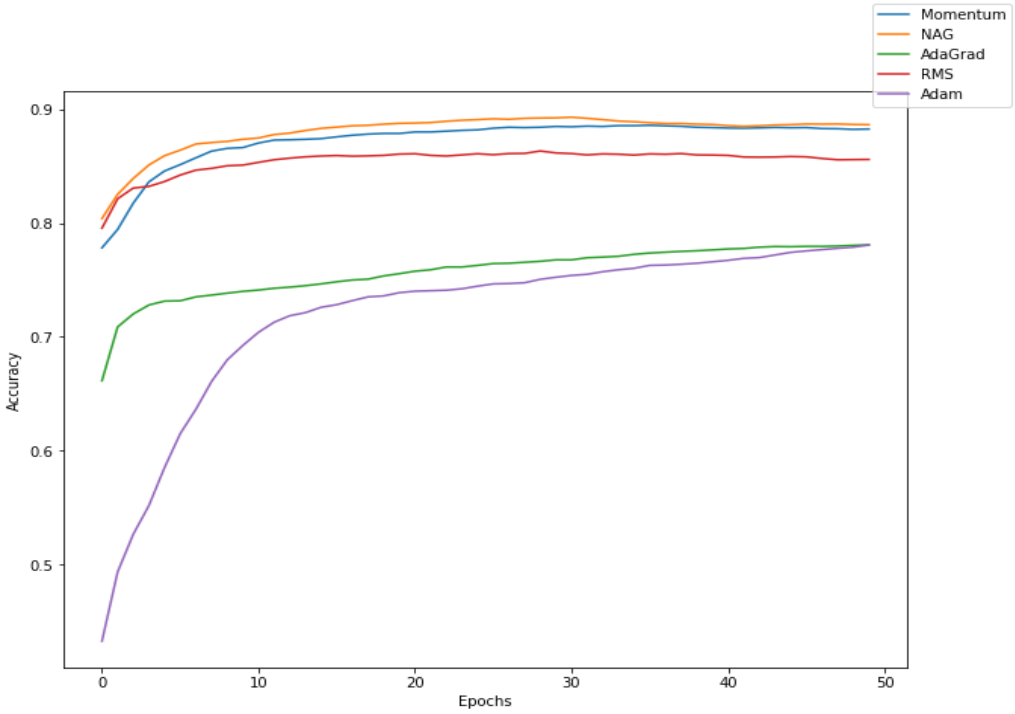
```
epochs = 50
# MyNeuralNetwork(N_layers, Layer_sizes, activation, learning_rate, weight_init, batch_size, num_epochs)
myNet = MyNeuralNetwork(3, [784, 256, 10], "tanh", 0.01, "random", 64, epochs,optimizers="momentum")
momentum_validationLoss, momentum_trainLoss = myNet.fit(X_train, y_train, X_validation, y_validation)
y_pred = myNet.predict(X_test)

filename = "/content/momentum.sav"
pickle.dump(myNet, open(filename, 'wb'))
```

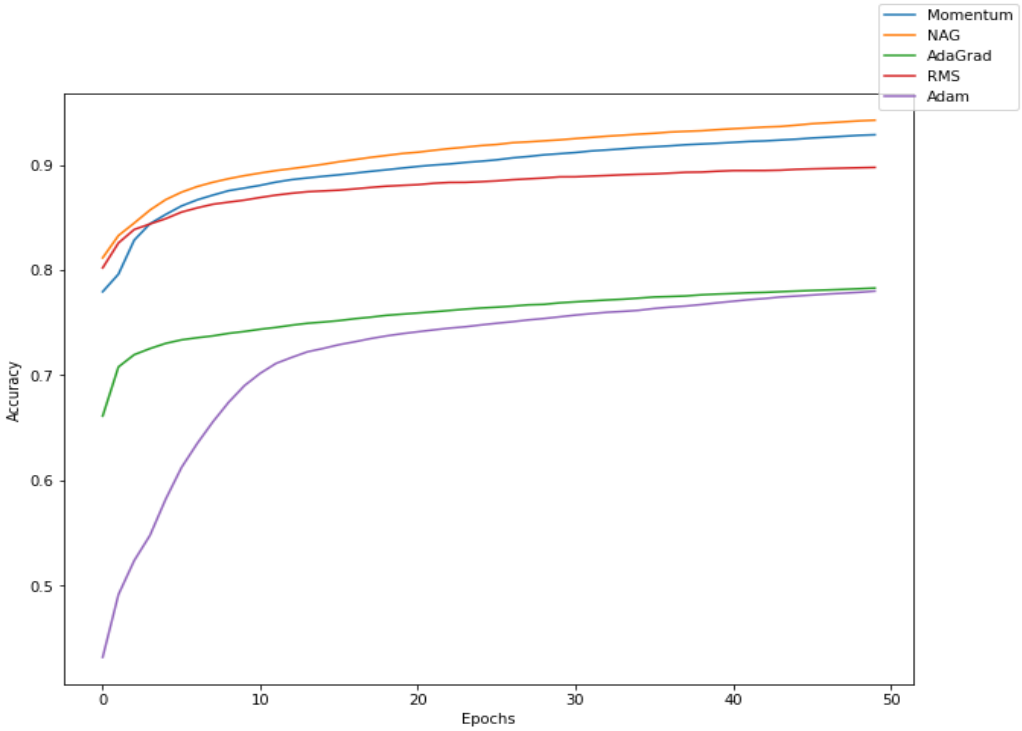
```
Epoch: 1, Time Spent: 4.84s, Validation Accuracy: 10.04%
Epoch: 2, Time Spent: 10.71s, Validation Accuracy: 10.04%
Epoch: 3, Time Spent: 16.62s, Validation Accuracy: 10.04%
Epoch: 4, Time Spent: 22.57s, Validation Accuracy: 10.04%
Epoch: 5, Time Spent: 28.49s, Validation Accuracy: 10.04%
Epoch: 6, Time Spent: 34.49s, Validation Accuracy: 10.04%
Epoch: 7, Time Spent: 40.38s, Validation Accuracy: 10.04%
Epoch: 8, Time Spent: 46.29s, Validation Accuracy: 10.04%
Epoch: 9, Time Spent: 52.17s, Validation Accuracy: 10.04%
Epoch: 10, Time Spent: 58.03s, Validation Accuracy: 10.04%
Epoch: 11, Time Spent: 63.92s, Validation Accuracy: 10.04%
Epoch: 12, Time Spent: 69.80s, Validation Accuracy: 10.04%
```

Accuracy results for other optimizers

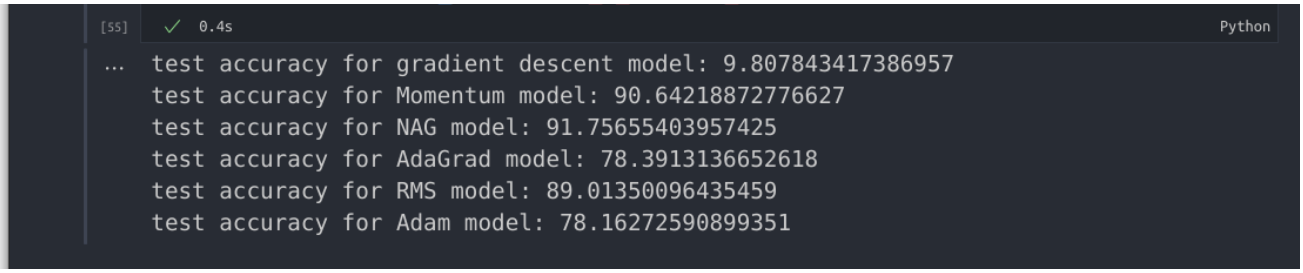
1. **Epoch vs Validation accuracy score**



2. **Epoch vs Training accuracy score**



3. **Test Accuracy for optimizers -**



Results and inferences

1. Nesterov's Accelerated Gradient works best giving both the highest training and validation accuracy.
2. NAG also gives the highest test set accuracy of 91.75%
3. The order of performance for optimizers is NAG > Momentum > RMS > AdaGrad >Adam > gradient descent.
4. We can also see that even though Adam combines the best of RMSProp and momentum optimizer, still its combined results are the worst of all other special optimizers than gradient descent.
5. Since NAG works better than the momentum model, we can also infer that using momentum at the current step before finding a gradient turns out to be slightly better than doing the reverse steps in our case.
6. The rate of convergence is the same as the rate of performance based on the test set accuracy after the final training i.e NAG > Momentum > RMS > AdaGrad >Adam > gradient descent.
7. However, we can see that till epoch=3, the RMSProp optimizer was converging faster than the momentum model but from epoch=4, momentum overtakes RMSProp and maintained this position.
8. Overall, using fixed learning rate is better than adaptive learning rates for our model configuration on fashion MNIST data.