Describtion for Function of Algorithm

Helper Function

Activation Function

[91]

```
def sigmoid(z):

    return 1/(1+np.exp(-z))
```

Acctivation Function

[93]

```
def plot_acc_msa_with_epochs(monitoring_df):
    fig,axes=plt.subplots(1,2,figsize=(15,2))
    monitoring_df.accuracy.plot(ax=axes[1],title="Accuracy")
    monitoring_df.mean_squared_error.plot(ax=axes[0],title="Mean Squared Error")
    fontdict = {'family': 'Arial',
        'color':  'darkred',
        'weight': 'heavy',
        'size': 15,}
    plt.text(3,0.96,'Learning rate=%s'%(0.06),fontdict=fontdict)
```

Function that Plot accuracy and MSE at each epoch . its argumend is data frame contain two column ,one for loss function through each epoch and another for accuracy through also each epoch

94]

```
def plot_TT_Curves(X,Y,model):
    X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.25,random_state=42)
    train_accuracy, test_accuracy = [], []

    for m in range(1, len(X_train)):

        monitoring=model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_test_predict =model.predict(X_test)
        train_accuracy.append(model.accuracy(y_train_predict,y_train[:m]))
        test_accuracy.append(model.accuracy(y_test_predict, y_test))
    plt.xlabel('Train Size')
    plt.ylabel('Accuracy')
    plt.plot(train_accuracy, "r-+", linewidth=2, label="train")
    plt.plot(test_accuracy, "b-", linewidth=3, label="test")
    plt.legend()
```

Function that plot accuracy of train and test through train set size ,so it shows difference in accuracy in train and test set

Class of NetWork

```
    def LayerWeight_initialization(self):
        # a small amount of randomization is necessary to
        # break symmetry; otherwise all hidden layers would
        # get updated in lockstep.
        if not self.n_hidden:
            layer_sizes = [ (self.n_output, self.n_input+1) ]
        else:
            layer_sizes = [ (self.n_hidden[0], self.n_input+1) ]
            previous_size = self.n_hidden[0]

            for size in self.n_hidden[1:]:
                layer_sizes.append( (size, previous_size+1) )
                previous_size = size

            layer_sizes.append( (self.n_output, previous_size+1) )

        self.layers = [[np.random.normal(0, 0.1, size=layer_size),np.zeros_like(np.random.normal(0, 0.1, size=layer_size)) ,sigmoid]for
layer_size in layer_sizes]
        self.v=0
```

- here initialize each layer as list contains in the first index the initialization of weights matrix for this layer .which number of nodes for previous layer is equal to number of rows for weight matrix and number of node in next layer equal to number of columns in weights matrix
- the secand index for list is velosity for momentum when i activate this term and i showe the equation that i used later in thie report . i use it in equation
- the second index is activation function to output of layer

```
    def fit(self,X,y,minibatch_size=10,momentum=False,momentum_Factor=0.9):
        self.n_input = X.shape[1]
        self.n_output = y.shape[1]
        self.LayerWeight_initialization()

        # fitting iterations
        for iteration in range(self.epochs):
            X,y=shuffle(X,y)

            self.forward_propagation(X)
            self.back_propagation(y,momentum,momentum_Factor)
        monitoring_df=pd.DataFrame(self.monitoring)

        return  monitoring_df
    def shuffle(self,X,y):
        data=np.concatenate([X, y],axis=1)
        col=data.shape[1]
        output_col=y.shape[1]
        X=data[:,:col-output_col]
        y=data[:,col-output_col:col]
        return X,y
```

- fit function use gradient descent in optimization and at each epoch shuffle data where implementation of shuffle function as shown in image
- after shuffling data at each epoch call forward propagation function to fit X and activation function at each layer and call backpropagation function to optimize weights on this output from activation function in forward function

- contain momentum argument that can i activate it by send to this function true ,and momentum factor

```
def forward_propagation(self, X):
    self._activations = []

    activation = X

    for W, v,activation_function in self.layers:
        bias = np.ones( (activation.shape[0], 1) )
        activation = np.hstack([bias, activation])
        self._activations.append(activation)
        activation = activation_function(activation @ W.T)

    self._activations.append(activation)
```

- here i calculate activation function for each layer as i mentioned above

```
def back_propagation(self, y,momentum,momentum_Factor):
    N = y.shape[0]
    y_hat = self._activations[-1]
    error = y_hat - y
    self.monitor(y_hat,y)
    for layer in range(self.n_layers-2, -1, -1):
        a = self._activations[layer]
        delta = (error.T @ a) / N
        if layer != self.n_layers-2:
            delta = delta[1:, :]
        W = self.layers[layer][0]
        v=self.layers[layer][1]
        if layer > 0:

            if layer != self.n_layers-2:
                error = error[:, 1:]

            error = (error @ W) * (a * (1-a))
        if momentum== True:
            # update weights
            v=momentum_Factor *v+self.learning_rate*delta
            W -=  v
        elif momentum ==False:
            W -= self.learning_rate * delta
```

- bachpropagation function depends on activation function that i calculated in forward function
- at the first , i calculated the error for the last layer that is stored in activatons list from forward propagation function
- propagating the error from each layer to the previous one.
- if momentum true so weights will update depends on equation below

$$V = \beta v + \alpha \bigtriangledown MSE$$

$$w = w - v$$

> beta is momentum vector

```
def monitor(self,y_hat,y):
    mse=self.mean_squared_error(y_hat,y)
    acc=self.accuracy(y_hat,y)
    self.monitoring["mean_squared_error"].append(mse)
    self.monitoring["accuracy"].append(acc)
```

- i call this function in back probagtion function that calculate MSE and accuracy at each epoch to monitor performace of algorithm and store these two values in data frame to use this frame in plotting cost function with iterations

- these function to measure accuracy and MSE(cost function)

```
        return x,y
    def predict(self, X):
        y_class_probabilities = self.predict_proba(X)

        return np.where(y_class_probabilities[:,:] < self.threshold, 0, 1)

    def predict_proba(self, X):
        self.forward_propagation(X)
        return self._activations[-1]
```

- predict_prob function that predict output of Neural network with floats numbers
- so in predict function just i round each value in predication to zero or one depend on threshold value

Equation I used in BackPropagation Algorithm

To predict output of activation function

$$a_0 = x \rightarrow inputLayer$$

$$Z_i = wa_{i-1}$$

> weight matrix contains bais vector

$$a_{i-1} = sigmoid(Z_i)$$

$$y = a_L$$

To Minimize Loss Function , differentiate it w.r.t weight of each layer

$$J = \frac{\sum_1^n (\hat{y} - y)^2}{2N}$$

$$\frac{\partial J}{\partial \Theta} = \frac{\partial J}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial \Theta}$$

$$\frac{\partial J}{\partial a} = \hat{y} - y$$

$$\frac{\partial a}{\partial z} = \frac{1}{1+e^{-z}}(1 - \frac{1}{1+e^{-z}})$$

$$\frac{\partial z}{\partial \Theta} = (X)or(a_{l-1}) \Rightarrow previous - layer$$

> **Thats what i tried to implement**

Prepare each data

Prepare Data

[23]

```
np.random.seed(0)
df = sns.load_dataset("iris")
df=shuffle(df)
y=pd.get_dummies(df.species).values
X=df.drop(["species"],axis=1).values
df.head(3)
```

	sepal_length	sepal_width	petal_length	petal_width	species
114	5.8	2.8	5.1	2.4	virginica
62	6.0	2.2	4.0	1.0	versicolor
33	5.5	4.2	1.4	0.2	setosa

Split Data with 70/30

[24]

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,random_state=42)
```

- here i split classes that is in target or class column to number of columns equal to number of classes and split data to 70 /30

```
np.random.seed(3)
model = NeuralNetwork(n_hidden=[4], epochs=2000,learning_rate=0.08)
monitor=model.fit(X_train,y_train,momentum=True,momentum_Factor=0.9)
y_hat = model.predict(X_test)
acc=model.accuracy(y_hat,y_test)
plot_acc_msa_with_epochs(monitor)
print(acc)
```

- Here when i call my algorthm and the first argument [n_hidden]is number of hidden layer and number of node for each layer

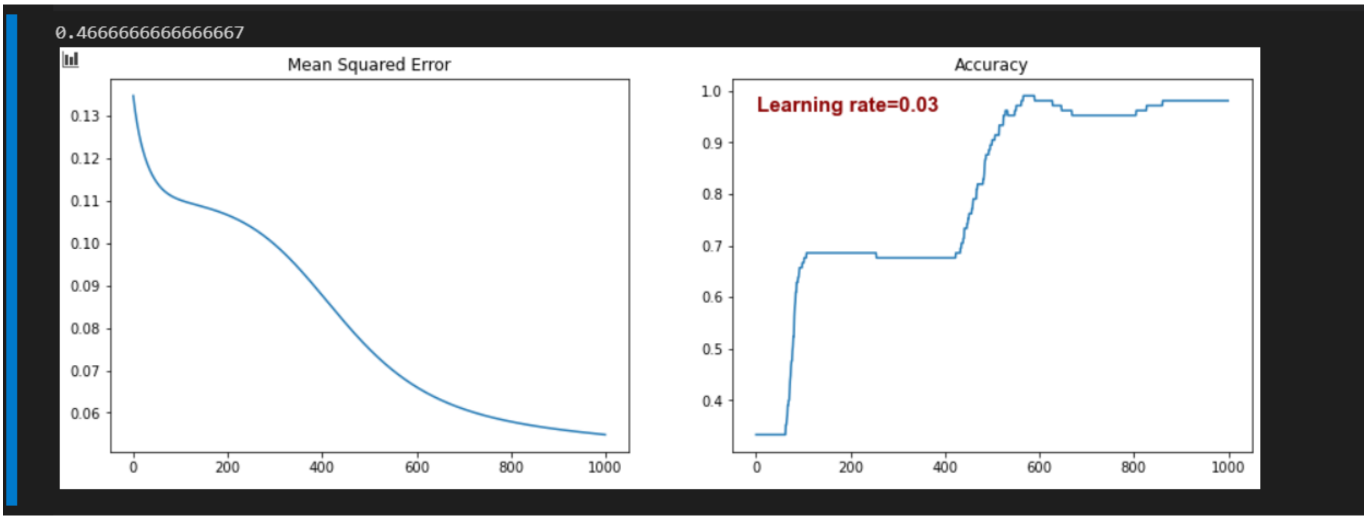
Iris Data

Graph for Performance of Algorithm at different Learning Rate

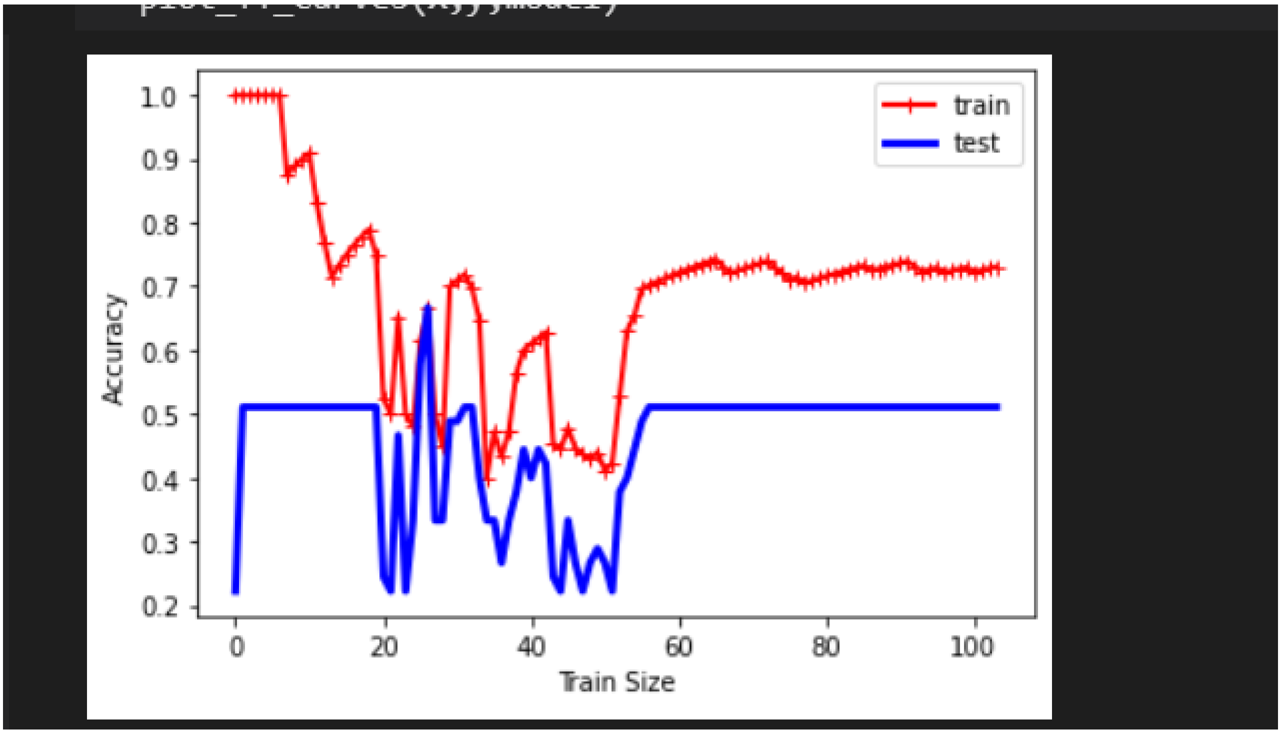
Using 7 node for hidden layer

at lr= 0.03

Graph for performance of traning set through epochs



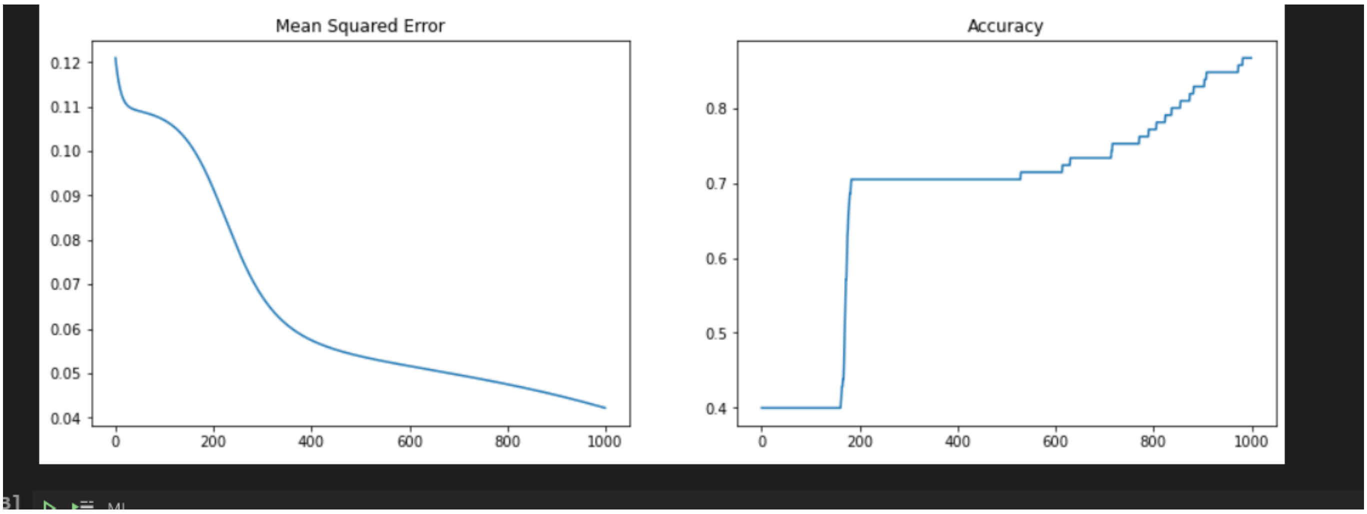
Graph of performance of train and test accuracy



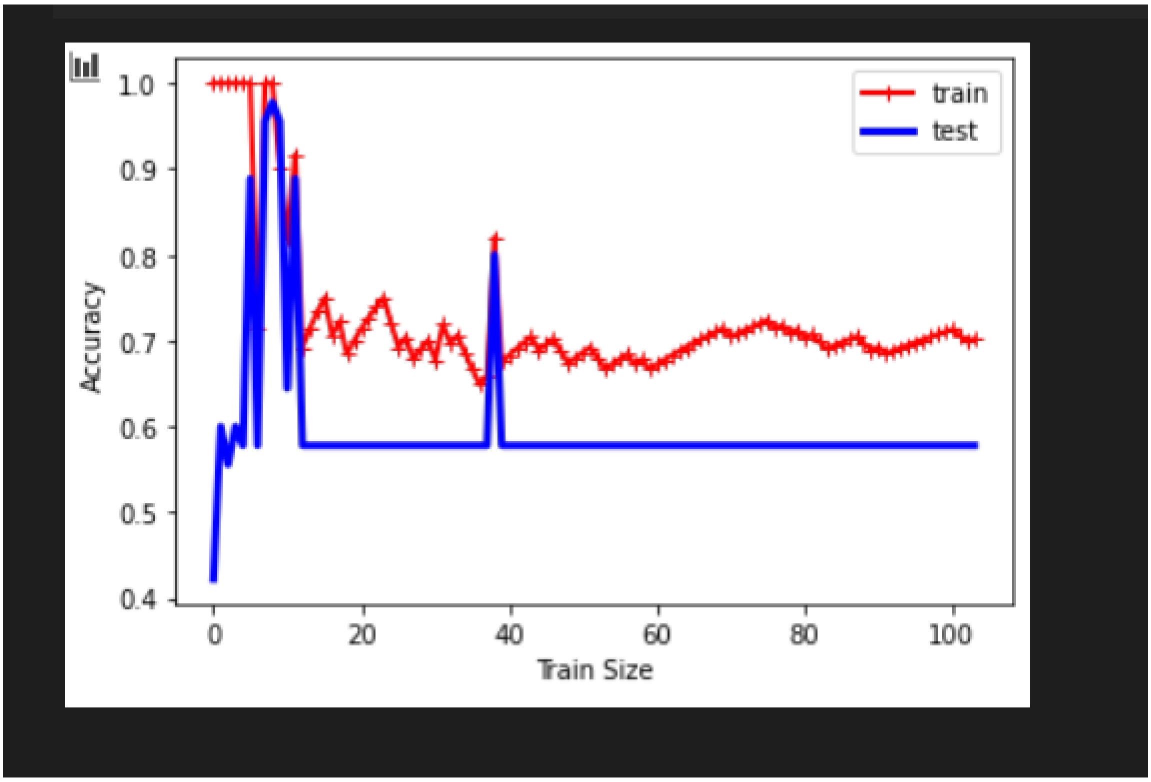
> There are a big variance between tran and test set in Accuracy

at lr= 0.06

Graph for performance of traning set through epochs



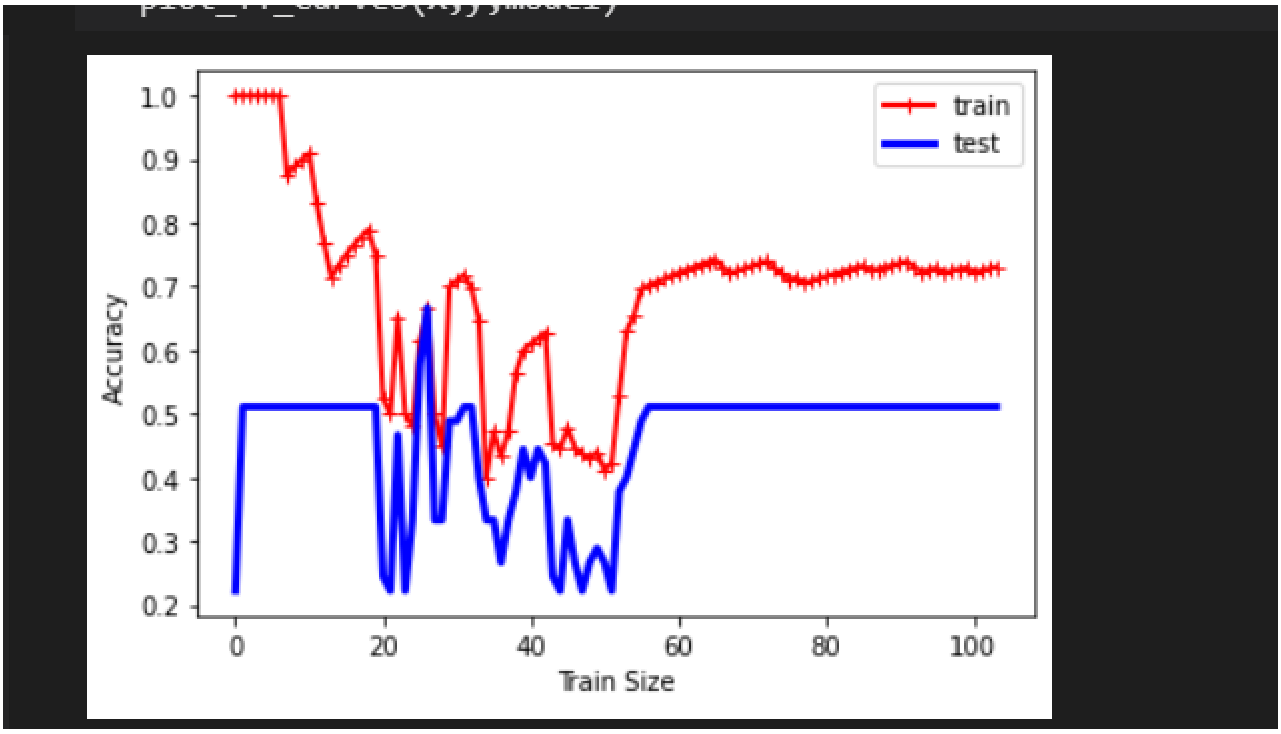
Graph of performance of train and test accuracy



> Here also is not the best senareo

at lr= 0.05

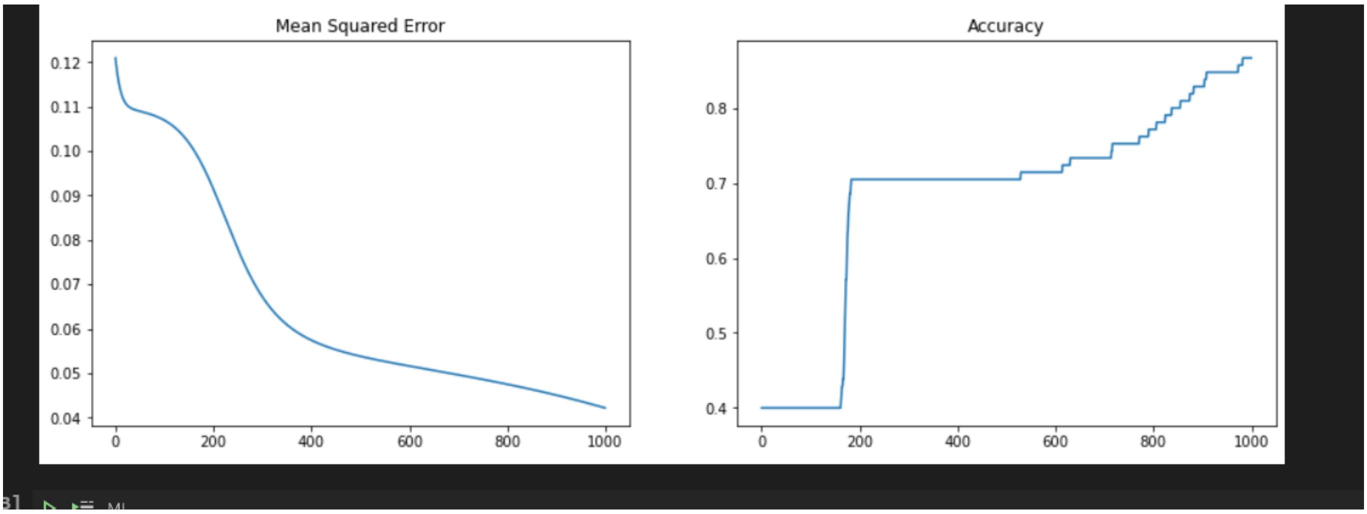
Graph of performance of train and test accuracy



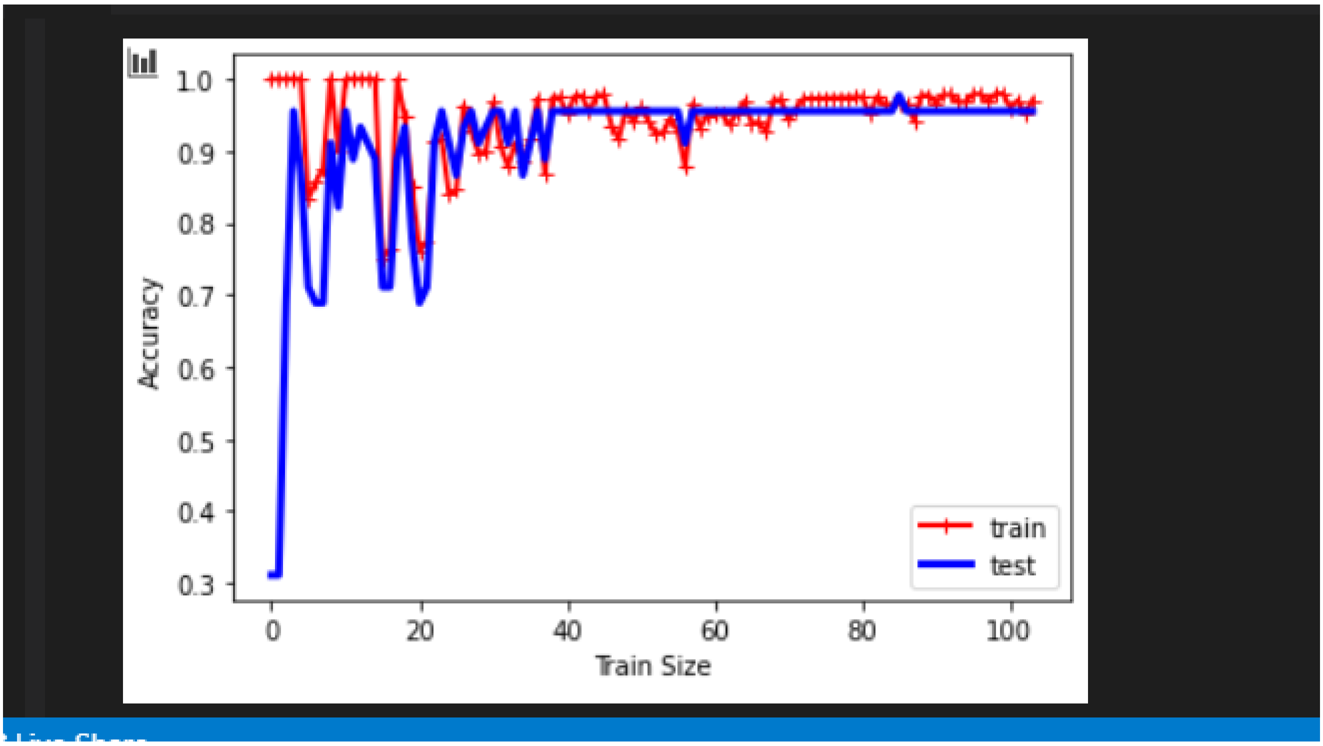
> There are a big variance between tran and test set in Accuracy

at lr= 0.08

Graph for performance of traning set through epochs



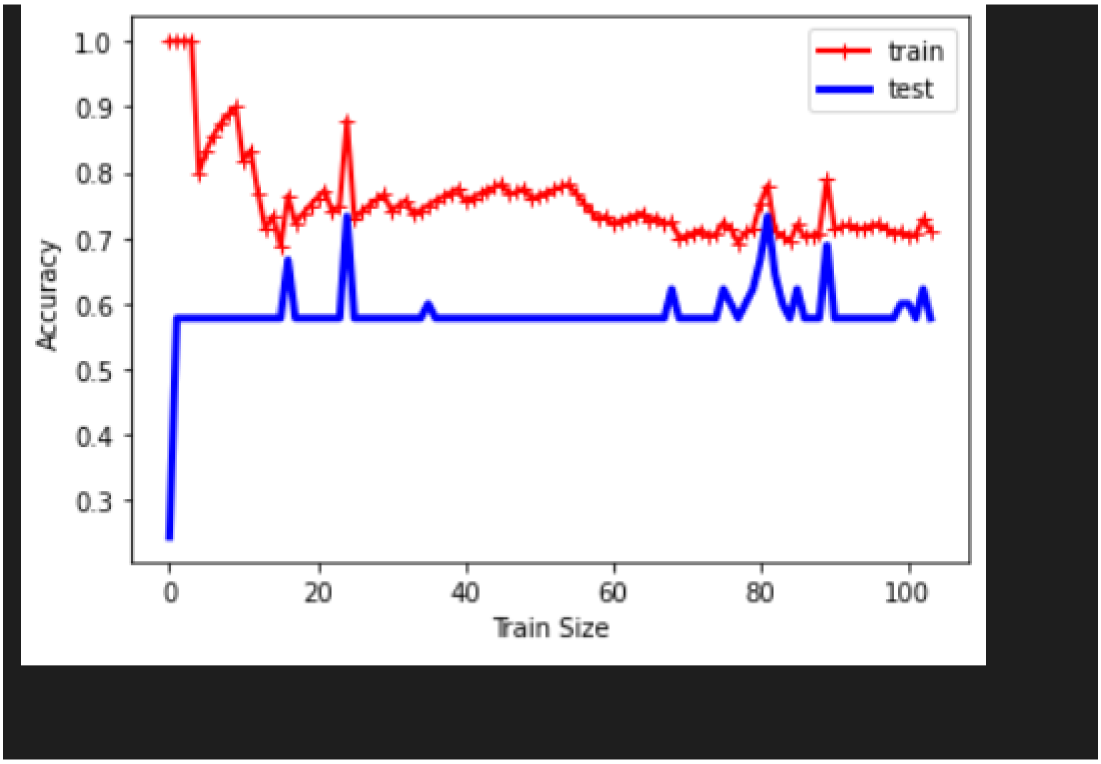
Graph of performance of train and test accuracy



> Here ,It is good performance

at lr= 0.09

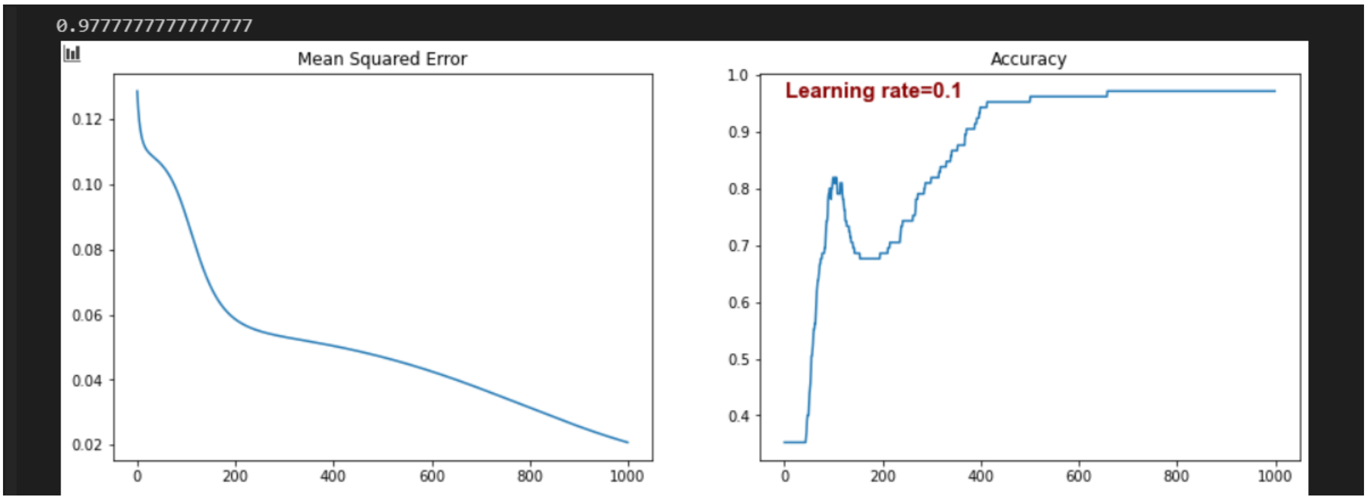
Graph of performance of train and test accuracy



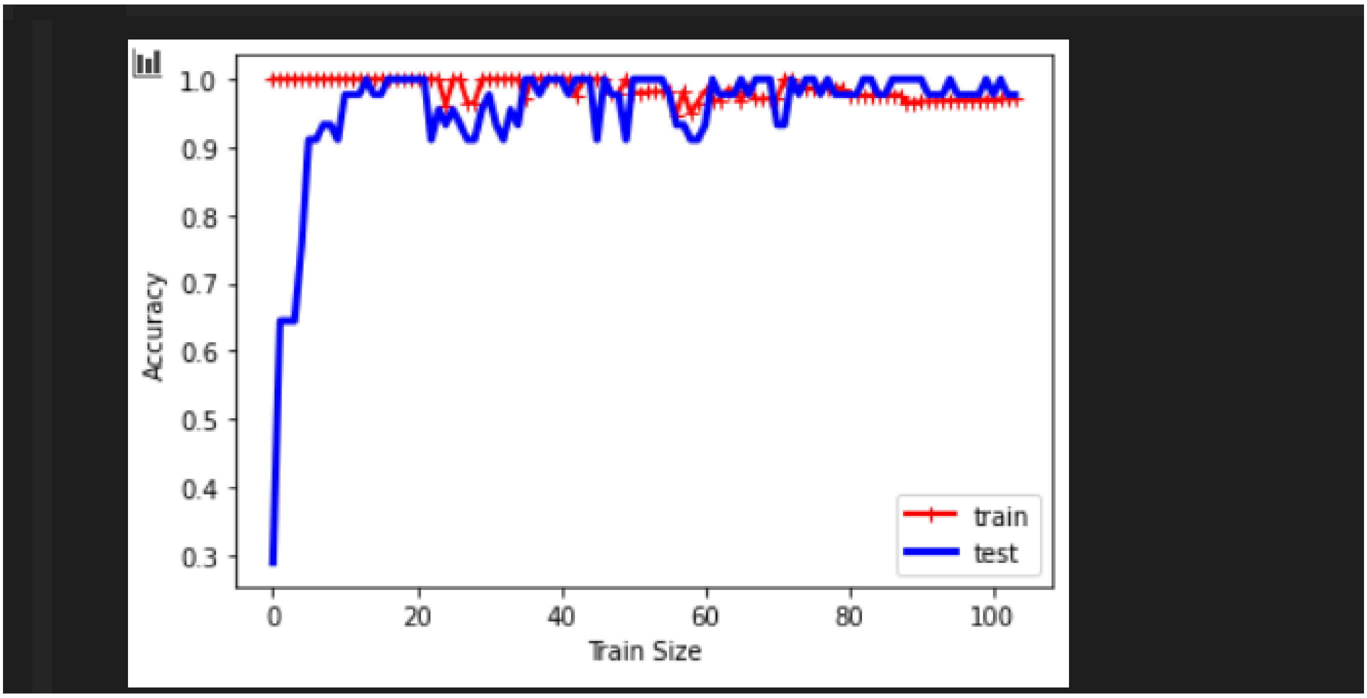
> Here IT is worse

at lr= 0.1

Graph for performance of traning set through epochs



Graph of performance of train and test accuracy



> **So, at lr=0.08 or .1 , are the best performance**

I Select lr=0.1

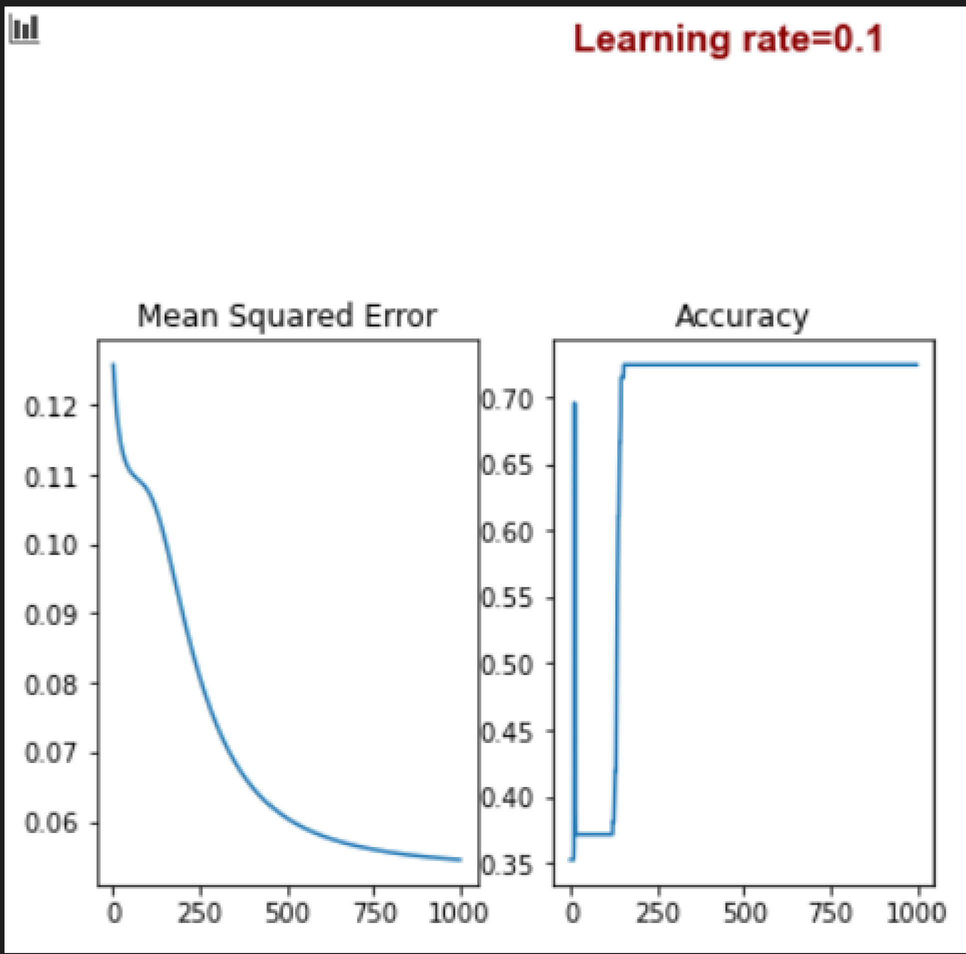
Graph for Performance of Algorithm at different numbers of nodes for hidden layer and lr=0.1

at n=1

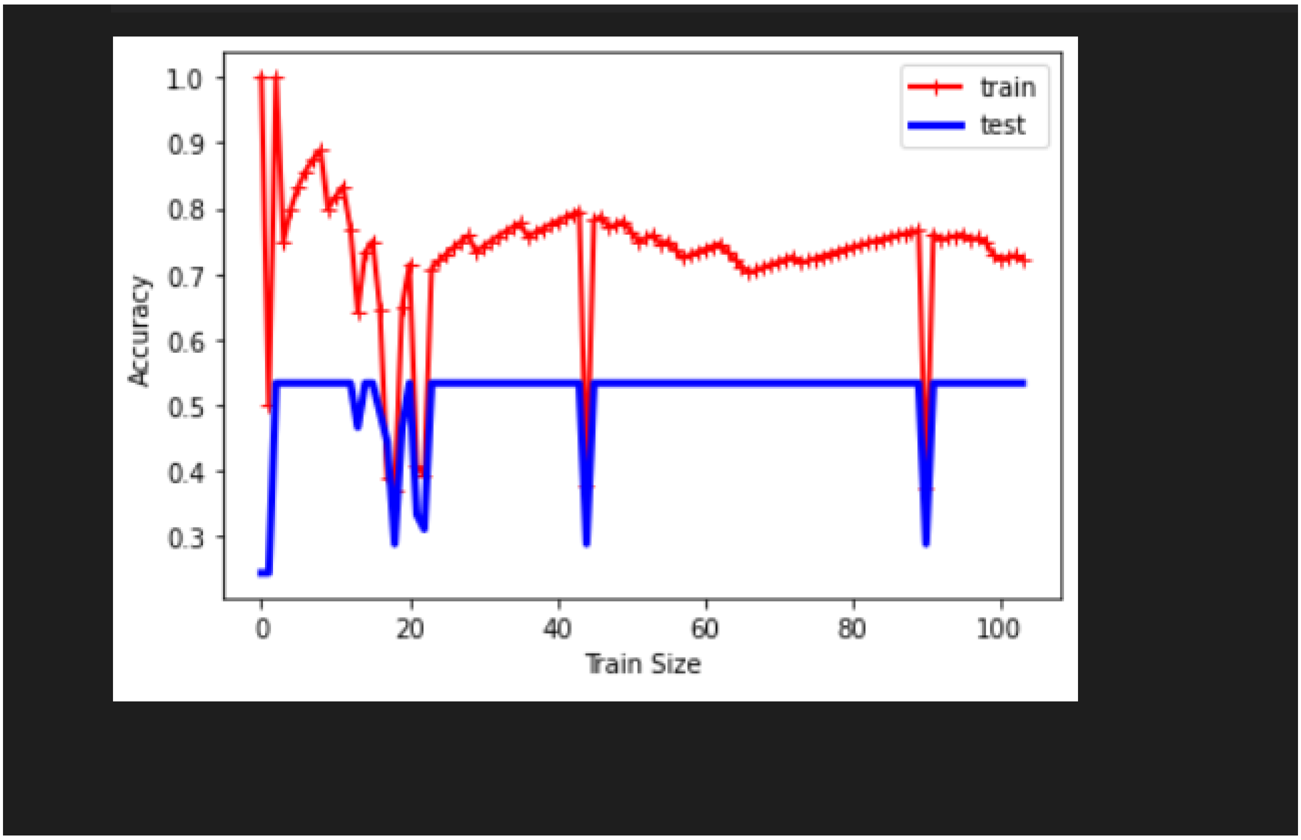
Graph for performance of traning set through epochs

```
print(acc)
```

```
0.5333333333333333
```



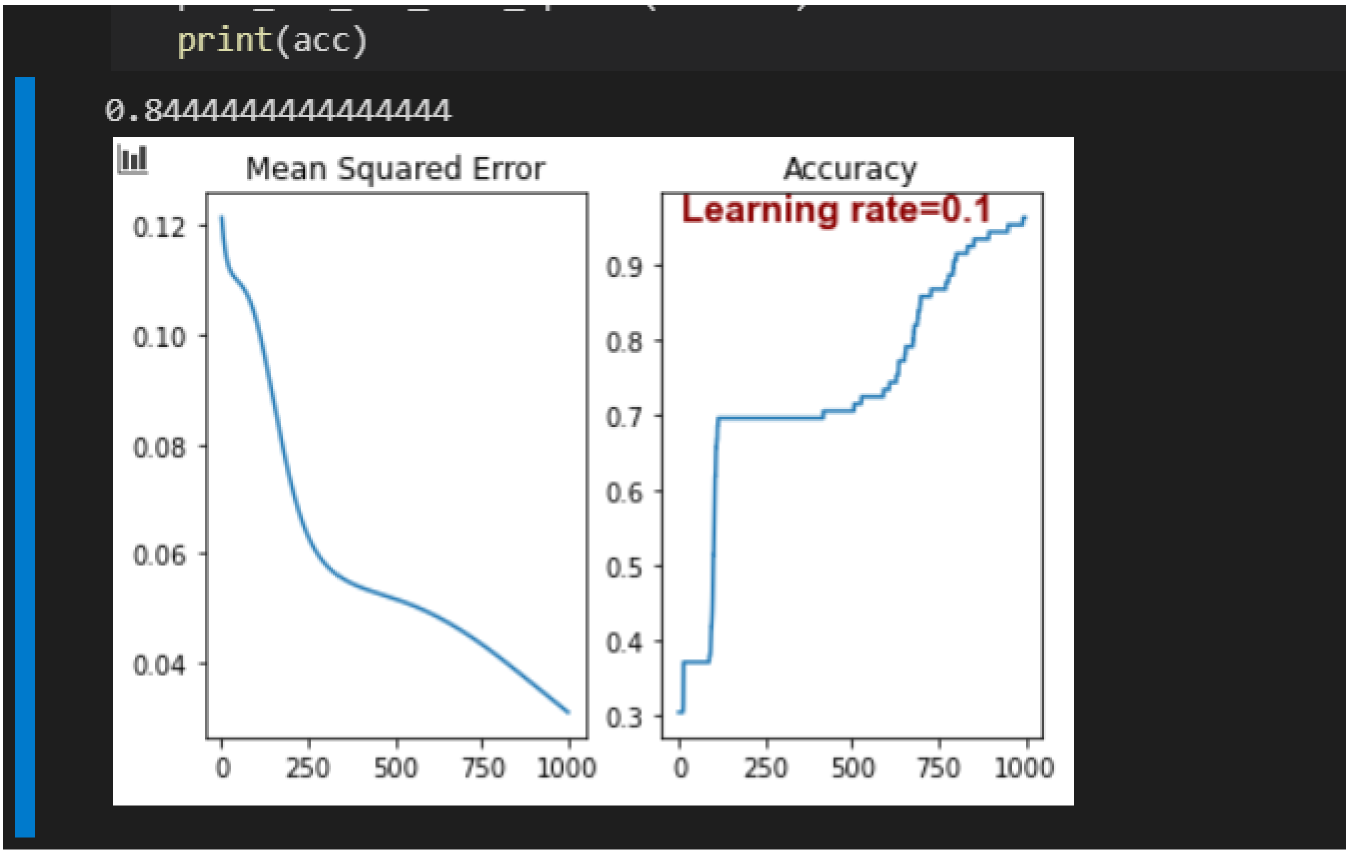
Graph of performance of train and test accuracy



> test set accuracy =0.533 and train acc=0.75, big variance

at n=2

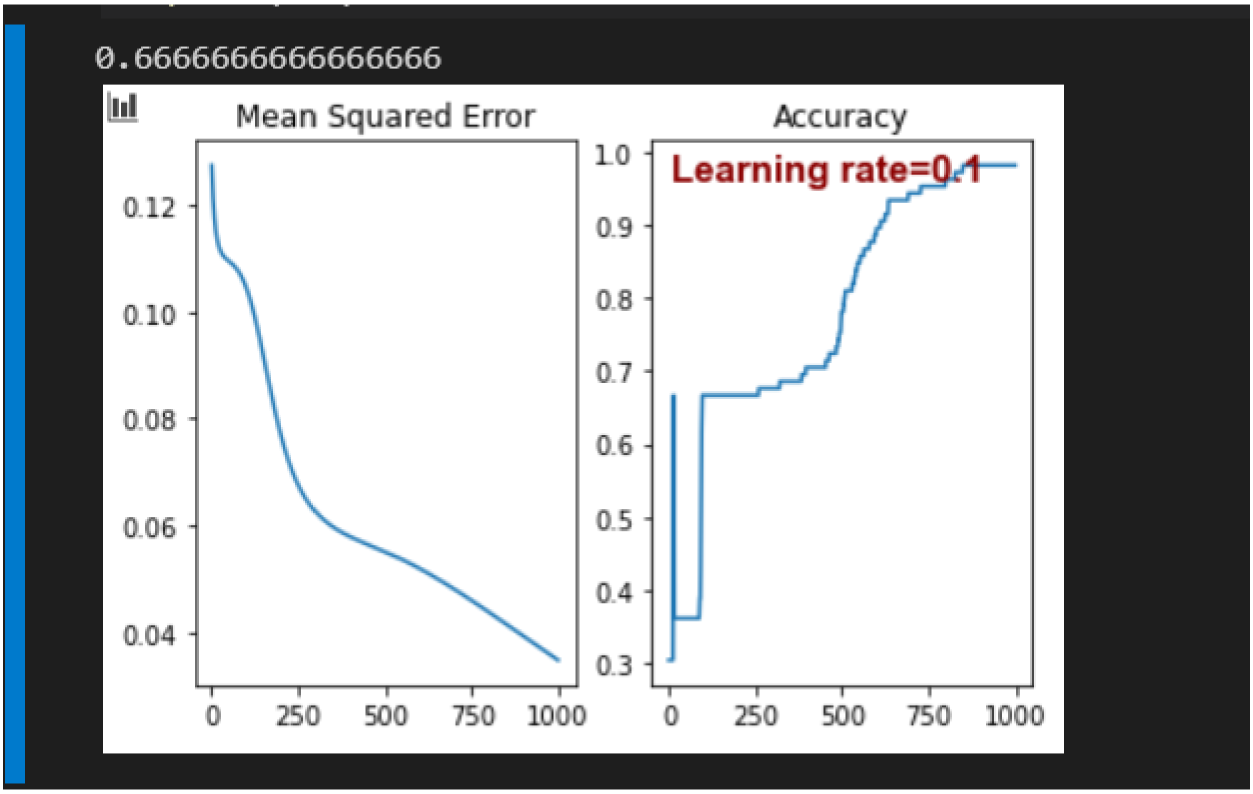
Graph for performance of traning set through epochs



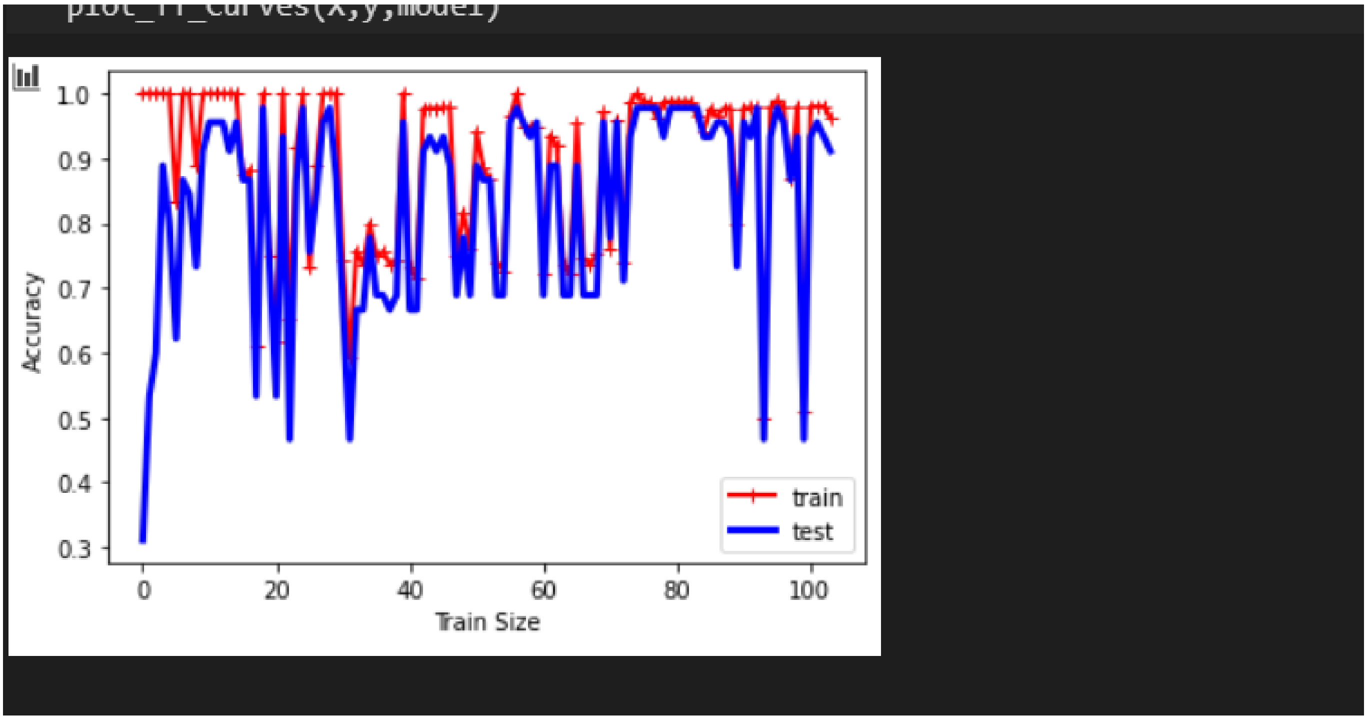
> test set accuracy =0.8 and train acc=0.95, big variance

at n=3

Graph for performance of traning set through epochs



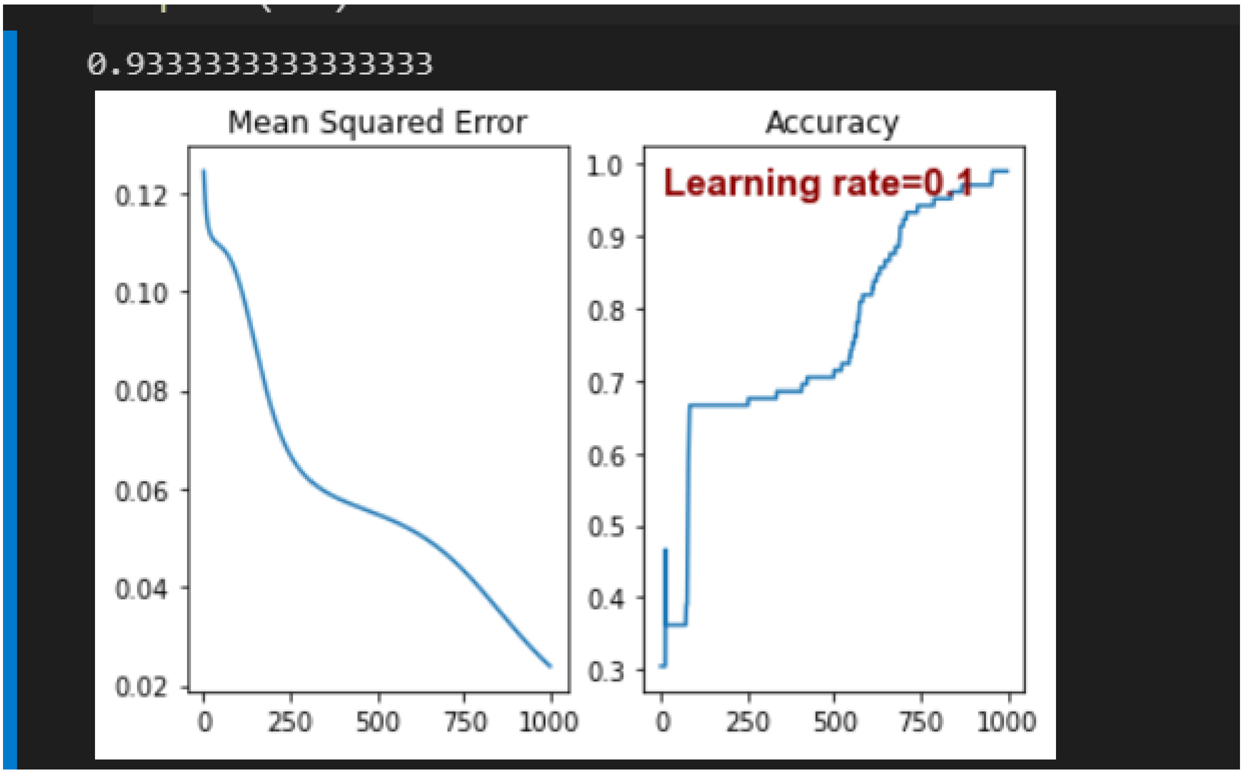
Graph of performance of train and test accuracy



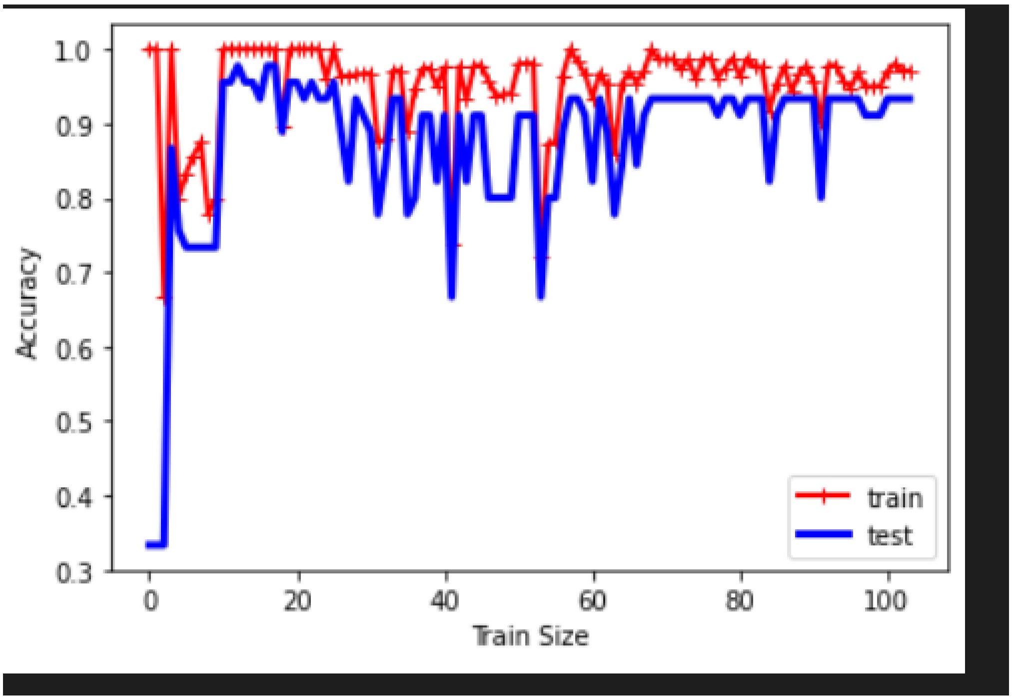
> test set accuracy =0.66 and train acc=0.99, big variance

at n=4

Graph for performance of traning set through epochs



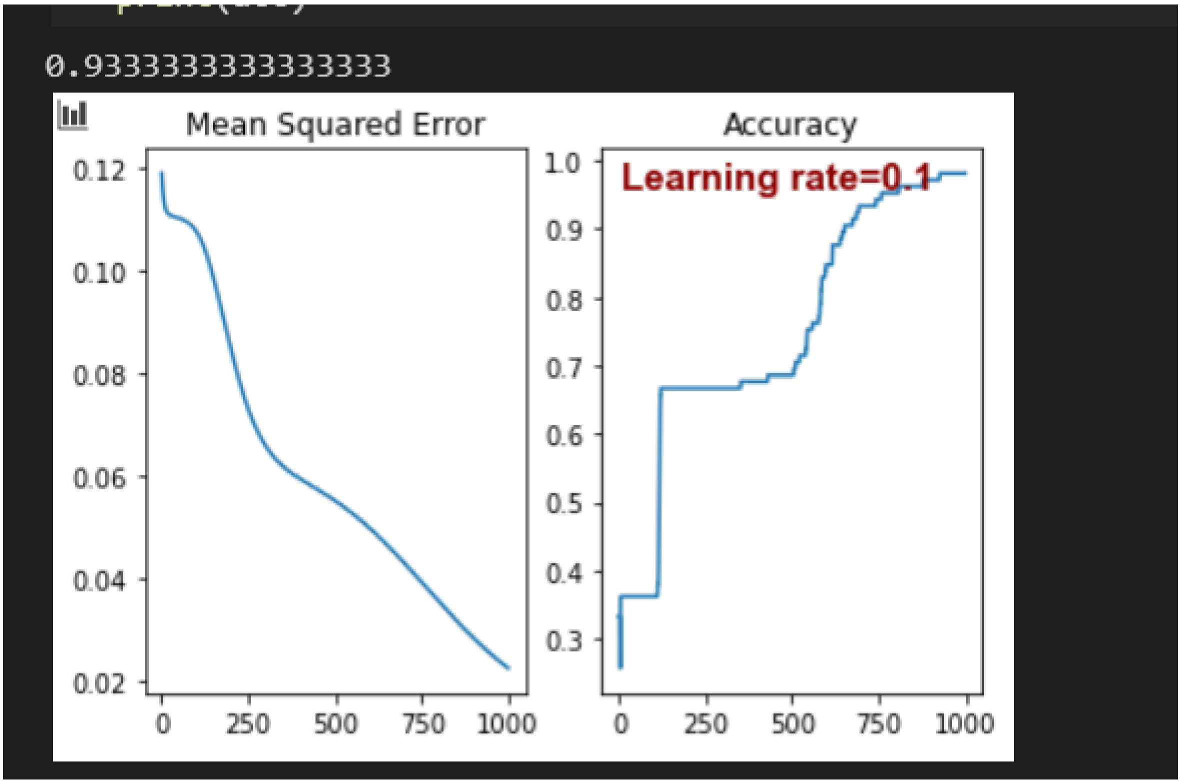
Graph of performance of train and test accuracy



> test set accuracy =0.93 and train acc=0.98, it is very good

at n=5

Graph for performance of traning set through epochs



Graph of performance of train and test accuracy

> test set accuracy =0.93 and train acc=0.98, it is similar to n=4

> ## So I Select numbers of hidden nodes =4 and lr =0.1

`Vowel Data`

Graph for Performance of Algorithm at different Learning Rate

Using 7 node for hidden layer

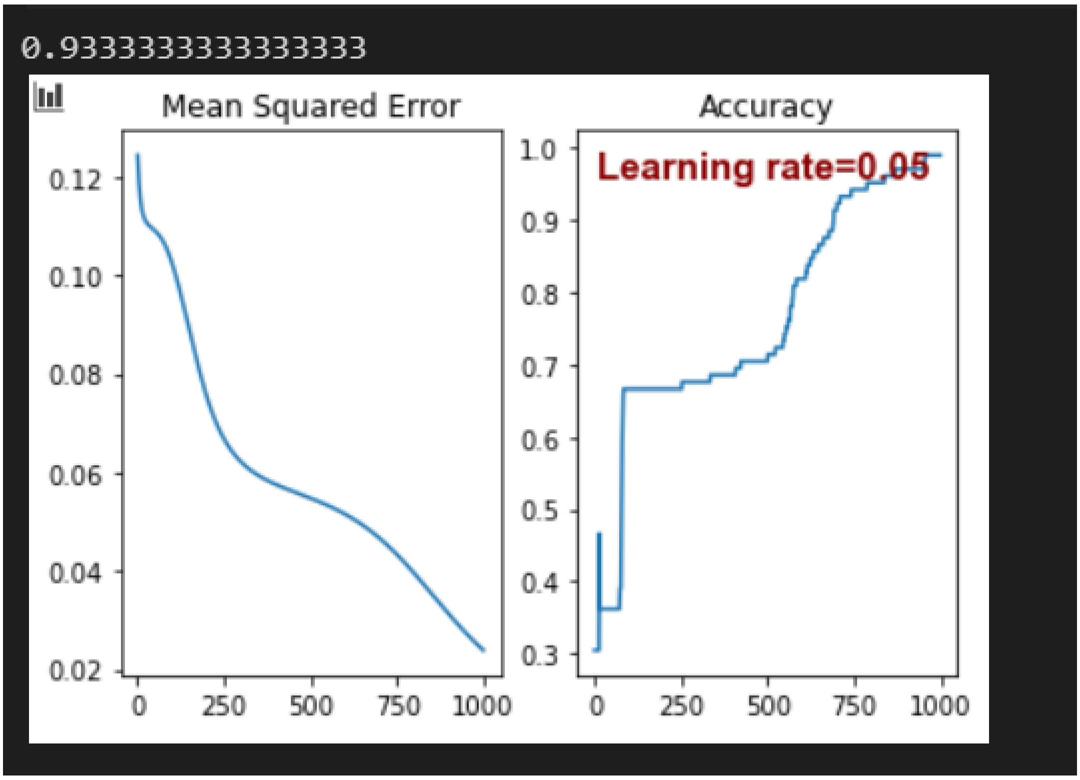
at lr= 0.03

Graph of performance of Loss function and accuracy

> here Train accuracy=0.98 and test accuracy= 0.6 , big valiance

at lr= 0.05

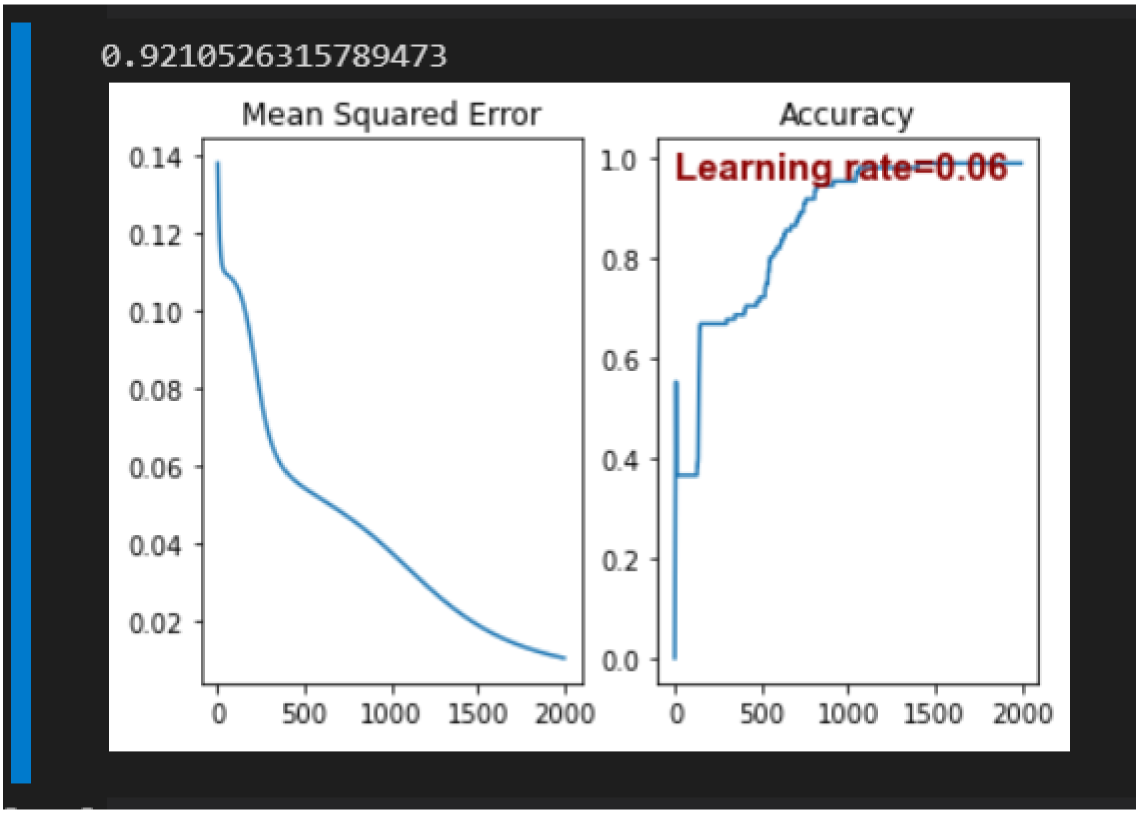
Graph of performance of Loss function and accuracy



> it is very good , train accuracy is =0.98 and test accuracy =0.93

at lr= 0.06

Graph of performance of Loss function and accuracy



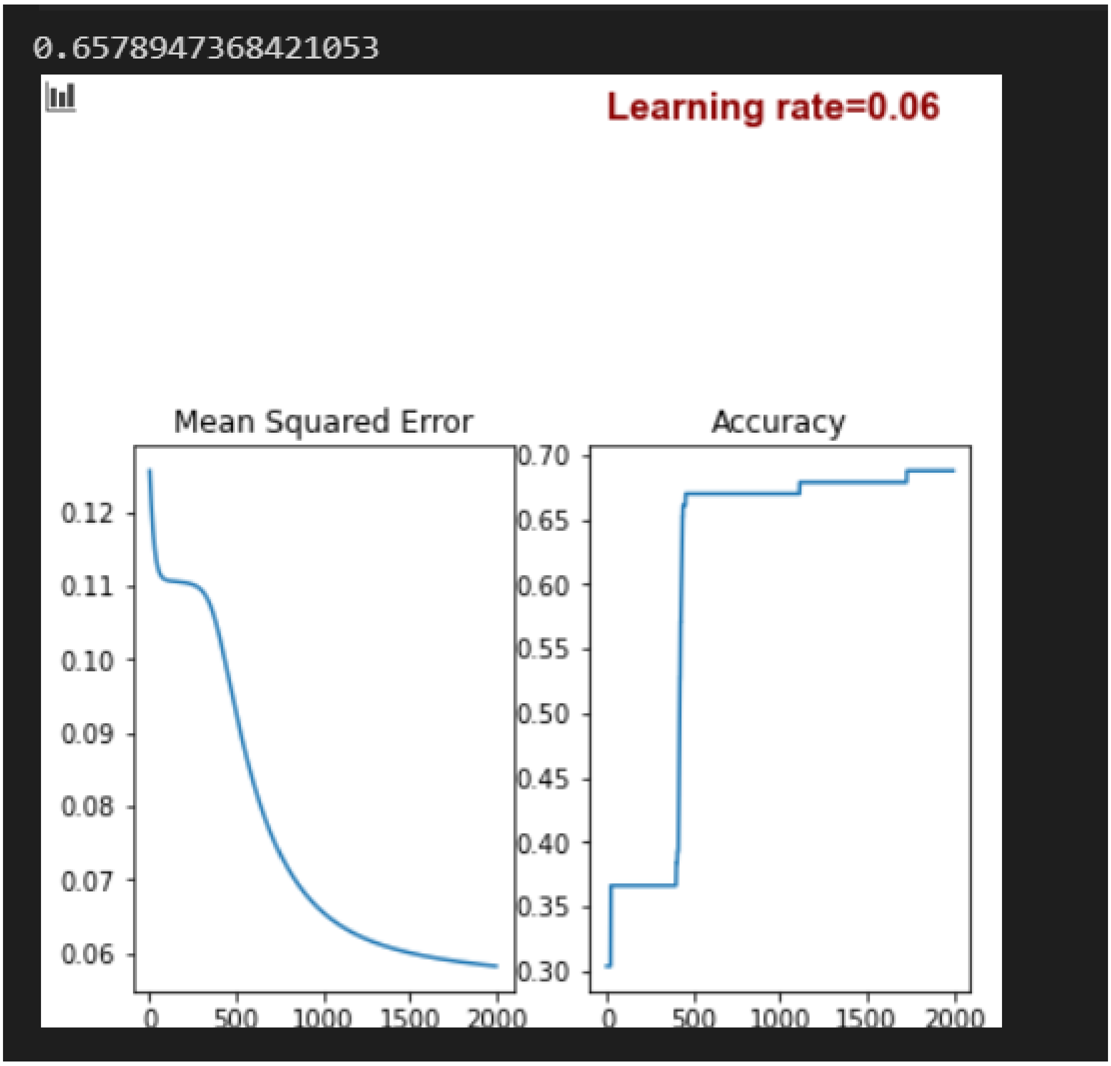
> it is similar to lr= 0.05

> **I select lr=0.06**

Graph for Performance of Algorithm at different numbers of nodes for hidden layer and lr=0.05

at n=1

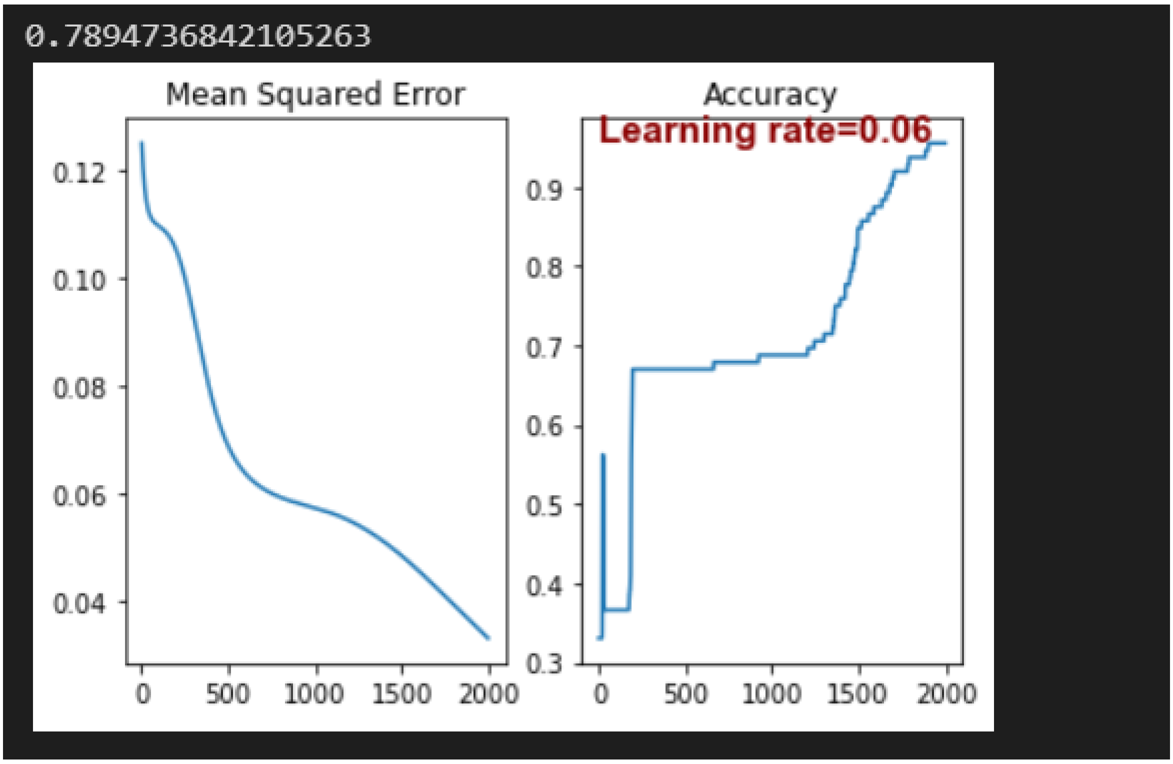
Graph for performance of traning set through epochs



> accuracy of train is about 0.99 and test set is 0.6

at n=2

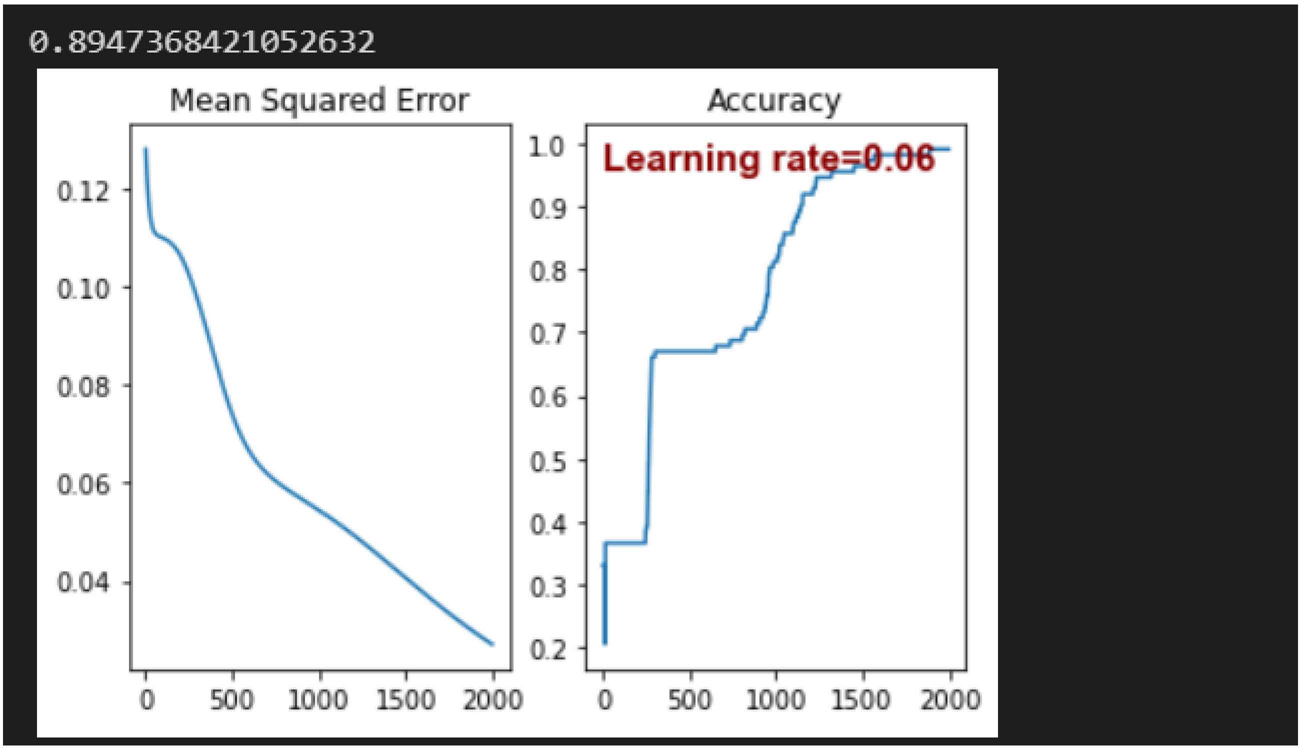
Graph for performance of traning set through epochs



> accuracy of train is about 0.95 and test set is 0.78

at n=3

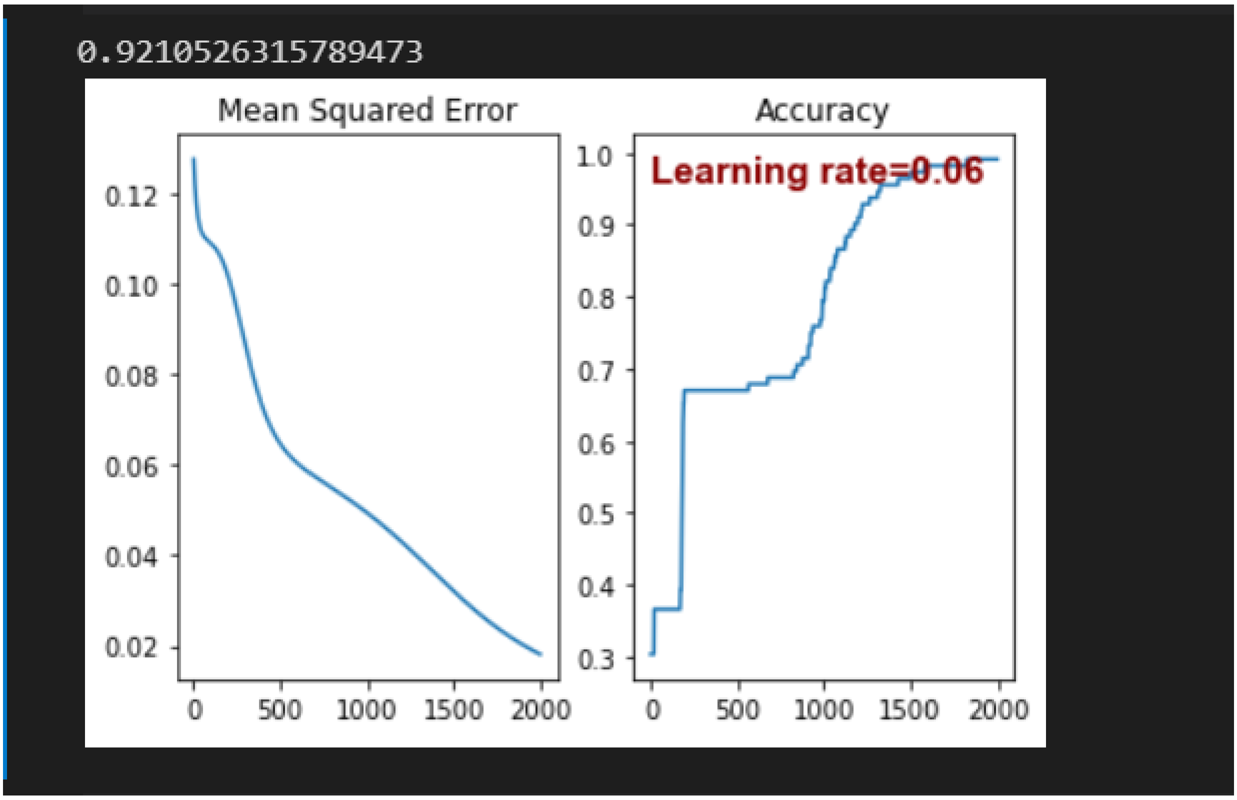
Graph for performance of traning set through epochs



> accuracy of train is about 0.99 and test set is 0.89

at n=4

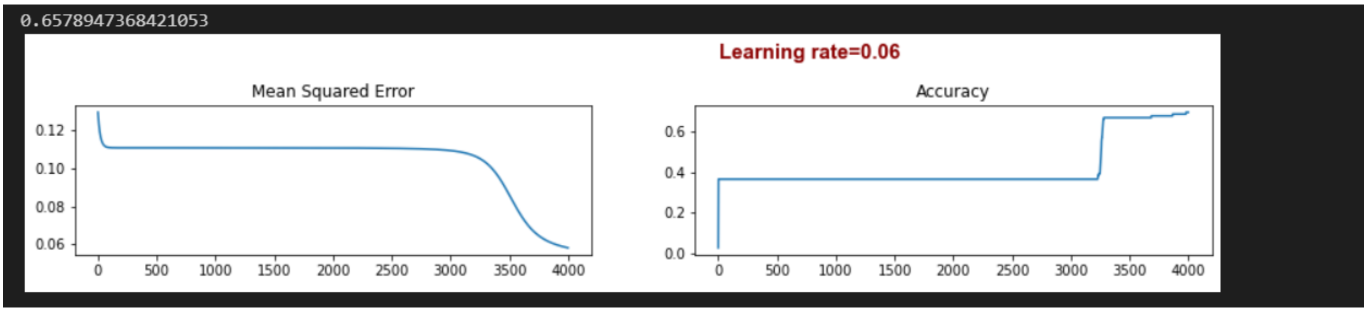
Graph for performance of traning set through epochs



> accuracy of train is about 0.98 and test set is 0.92, it is very good

> ## So I Select numbers of hidden nodes =4 and lr =0.05

2-hidden layer neural network



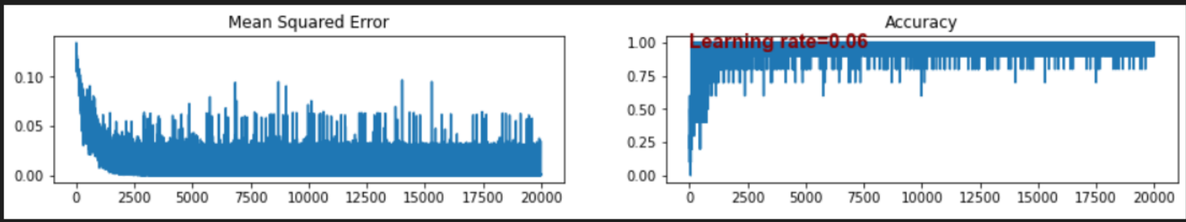
> At 4000 epochs ,accuracy of test set is =0.65 and train set is about 0.7

Using Momentum On `Iris Data`

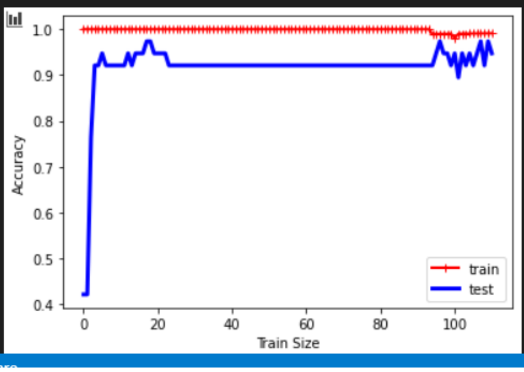
at number of hidden layer nodes =4 and learning rate =0.08

```
np.random.seed(3)
model = NeuralNetwork(n_hidden=[4], epochs=2000,learning_rate=0.08)
monitor=model.fit(X_train,y_train,momentum=True,momentum_Factor=0.9)
y_hat = model.predict(X_test)
acc=model.accuracy(y_hat,y_test)
plot_acc_msa_with_epochs(monitor)
print(acc)
```

0.9777777777777777




```
plot_TT_Curves(X,y,model)
```



> I change here function of optimization from gradiant descent to mini batch stochastic gradient descent , and mini batch size is =10

- not big difference in result , just i did it to expriment
- here impelementation for this part

```
def _get_batch(self, X, y, batch_size=10):
    indexes = np.random.randint(len(X), size=batch_size)
    return X[indexes,:], y[indexes,:]

def fit(self,X,y,minibatch_size=10,momentum=False,momentum_Factor=0.9):
    self.n_input = X.shape[1]
    self.n_output = y.shape[1]
    # y = np.atleast_2d(y)
    self.LayerWeight_initialization()

    # fitting iterations
    for iteration in range(self.epochs):
        X,y=shuffle(X,y)

        # Randomize data point
        for i in range(10):
            X_batch, y_batch = self._get_batch(X, y)
            self.forward_propagation(X_batch)
            self.back_propagation(y_batch,momentum,momentum_Factor)
    monitoring_df=pd.DataFrame(self.monitoring)
```

> **Accuracy with momentm is very good , 0.97 in test set and in train set about 0.99**

Using Momentum On `Vowel Data`

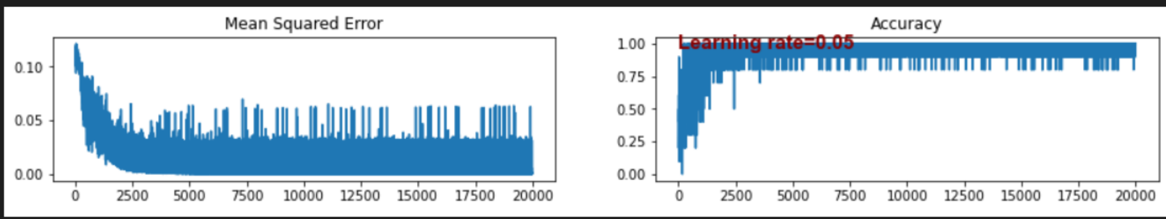
at number of hidden layer nodes =4 and learning rate =0.05

Use single hidden layer For vowel Data

[32]

```
model_vowel = NeuralNetwork(n_hidden=[4], epochs=2000,learning_rate=0.05)
monitor_v=model_vowel.fit(X_train_v,y_train_v,momentum=True,momentum_Factor=0.9)
y_hat_v = model_vowel.predict(X_test_v)
acc_v=model.accuracy(y_hat_v,y_test_v)
plot_acc_msa_with_epochs(monitor_v)
print(acc_v)
```

```
0.9736842105263158
```



> **Accuracy also in this data with momentm is very good , 0.97 in test set and in train set about 0.99**

Part five i discussed every Point with graphs