Do install Follwoing libraries (if not already installed) for smooth working of code

In [42]:

```
# !pip install sklearn pandas numpy tqdm
```

Import required libraries

In [2]:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_boston , load_iris, load_digits
from sklearn.preprocessing import Normalizer, OneHotEncoder
from sklearn.model_selection import train_test_split
from tqdm import trange
```

Question 1

1. Matrix Multiplication Layer
2. Bias Addition Layer
3. Mean Squared loss layer
4. Softmax Activation
5. Sigmoid Activation
6. Cross Entropy Loss Layer
7. Linear Activation
8. tanh Activation
9. ReLU Activation

1. Matrix Multiplication Layer

In [3]:

```
class MultiplicationLayer :
    """
    Inputs : X in R^(1xd) , W in R^(dxK)
    This layer takes X & W as input and perform these 2 tasks:
    1. Forward Pass : Matrix multiplication,  Z = XW
    2. Backward Pass : dZ/dX , dZ/dW
    """
    def __init__(self, X, W) :
        self.X = X
        self.W = W

    def __str__(self,):
        return " An instance of Muliplication Layer."

    def forward(self):
        self.Z = np.dot(self.X, self.W)

    def backward(self):
        self.dZ_dW = (self.X).T  # dZ/dW
        self.dZ_daZ_prev = self.W  # dZ/dX
```

2 Bias Addition Layer

In [4]:

```
class BiasAdditionLayer :
    """
    Inputs : Z in R^(1xK), B in R^(1xK)
    This layer takes output Z of forward pass of Multiplication Layer as input and perform
    1. Forward Pass :  Z = Z + B
    2. Backward Pass : dZ/dB
    """
    def __init__(self, Z : np.ndarray , bias : np.ndarray ):
        self.B = bias
        self.Z = Z

    def __str__(self,):
        return "An instance of Bias Addition Layer."

    def forward(self,):
        self.Z = self.Z + self.B

    def backward(self,):
        self.dZ_dB = np.identity( self.B.shape[1] )
```

3. Mean Squared Loss Layer

In [5]:

```
class MeanSquaredLossLayer :
    """
    This layer implements Mean Square Loss Layer.
    Inputs : Y in R^(1xK) , Y_hat in R^(1xK)  where K --> dimesion of output layer
    This layer takes prediction Y_hat and true Y as input and perform these 2 opearations :
    1. Forward Pass : L = (1/n) * || Y_hat - Y||**2
    2. Backward Pass : dL/dY_hat = (2/n)*(Y_hat - Y).T   Note :Here instead of dL/dY_hat ,
                                                             derivative of loss w.r.t. outp
    """
    def __init__(self, Y : np.ndarray , Y_hat : np.ndarray):
        self.Y = Y
        self.aZ = Y_hat

    def __str__(self,):
        return "An instance of Mean Squared Loss Layer"

    def forward(self, ):
        self.L = np.mean( ( self.aZ - self.Y)**2 )

    def backward(self,):
        self.dL_daZ = (2/len(self.Y))*(self.aZ - self.Y).T
```

4. Soft Max Activation

In [6]:

```
class SoftMaxActivation :
    """
    This layer implements SoftMax Activation Function.
    Input : a numpy array Z in R^(1XK)
    1. Forward Pass : Apply Softmax Activation function, aZ = softmax(Z).T
    2. Backward Pass : daZ/dZ  = diag(aZ) - sZ*transpose(aZ)  --> here diag(aZ) is diagonal
                                                                   i-th diagnoal entry repl
    """
    def __init__(self, Z):
        self.Z = Z

    def __str__(self,):
        return "An instance of Softmax Activation Layer"

    def forward(self,):
        self.aZ = self.softmax(self.Z)

    def backward(self,):
        self.daZ_dZ = np.diag( self.aZ.reshape(-1) ) - (self.aZ.T)@( (self.aZ))  # Shape =

    @staticmethod
    def softmax(Z : np.ndarray):
        max_Z = np.max( Z, axis=1 ,keepdims=True )
        return (np.exp(Z - max_Z ))/np.sum( np.exp(Z - max_Z), axis=1 , keepdims=True)
```

5. Sigmoid Activation

In [7]:

```
class SigmoidActivation :
    """
    This layer implements Sigmoid Activation Function.
    Input : a numpy array Z of shape Kx1
    1. Forward Pass : aZ = sigmoid( Z )
    2. Backward Pass : daZ/dZ = diagonal matrix with entries aZ_i*(1-aZ_i) --> sigZ_i means
    """

    def __init__(self,Z ):
        self.Z = Z

    def __str__(self,):
        return "An instance of Sigmoid Activation Layer"

    def forward(self,):
        self.aZ = self.sigmoid( self.Z )  # sigmoid calculation

    def backward(self,):
        diag_entries = np.multiply(self.aZ, 1-self.aZ).reshape(-1)
        self.daZ_dZ = np.diag(diag_entries)

    @staticmethod
    def sigmoid( Z : np.ndarray ) :
        return  1./(1 + np.exp(-Z) )
```

---

6. Cross Entropy Loss Layer

In [8]:

```
class CrossEntropyLossLayer :
    """
    This layer implements Cross Entropy Loss Layer.
    Inputs : Y in R^(1xK) , Y_pred in R^(1xK)  where K --> dimesion of output layer
    This layer takes prediction Y_pred and true Y as input and perform these 2 opearations
    1. Forward Pass : L = -1 * dot product of Y & log(Y_pred)
    2. Backward Pass : dL/dY_pred in R^(Kx1)
    """
    def __init__(self, Y , Y_pred):
        self.Y = Y
        self.aZ = Y_pred
        self.epsilon = 1e-40


    def __str__(self, ):
        return "An instance of Cross Entropy Loss Layer"

    def forward(self, ):
        self.L = - np.sum( self.Y * np.log(self.aZ+self.epsilon) )

    def backward(self, ):
        self.dL_daZ = -1*(self.Y/(self.aZ + self.epsilon)).T # Element wise division
```

7. Linear Activation

In [9]:

```
class LinearActivation :
    """
    Implementation of linear activation function.
    Input : Z in R^(1xn)
    Ouput : linear(Z) = Z
    """
    def __init__(self, Z):
        self.Z = Z

    def __str__(self,):
        return "An instance of Linear Activation."

    def forward(self, ):
        self.aZ = self.Z

    def backward(self,):
        self.daZ_dZ = np.identity( self.Z.shape[1] )
```

8. tanh Activation

In [10]:

```
class tanhActivation :
    """
    Implementation of tanh activation function
    Input : a numpy array Z in R^(1xK)
    1. Forward Pass : aZ = tanh(Z)
    2. Backward Pass : daZ/dZ = np.diag(1 - aZ**2)   --> R^(KxK)
    """
    def __init__(self, Z):
        self.Z = Z

    def __str__(self,):
        return "An instance of tanhActivation class."

    def forward(self,):
        self.aZ = np.tanh(self.Z)

    def backward(self,):
        self.daZ_dZ = np.diag(1 - self.aZ.reshape(-1)**2)
```

9. ReLUActivation

In [40]:

```
class ReLUActivation :
    """
    Implementation of relu activatino function
    Input : a numpy array Z in R^(1xK)
    1. Forward Pass aZ = max(Z,0)
    2. Backward Pass : daZ_dZ = diag_matrix( 1 if aZ_i>0 else 0 )
    """
    def __init__(self, Z):
        self.Z = Z
        self.Leak = 0.01

    def __str__(self,):
        return "An instance of ReLU activation"

    def forward(self,):
        self.aZ = np.maximum(self.Z,0)

    def backward(self,):
        self.daZ_dZ = np.diag( [1. if x>=0 else self.Leak for x in self.aZ.reshape(-1)])
```

Question 2 & 3

- Question 2 : Boston House Price Prediction
- Question 3 -- MNIST Hand Written Digit Classification

Load Data and Train Test Split

In [12]:

```
def load_data(dataset_name='boston',
              normalize_X=False,
              normalize_y=False,
              one_hot_encode_y = False,
              test_size=0.2):
    if dataset_name == 'boston' :
        data = load_boston()
    elif dataset_name == 'iris' :
        data = load_iris()
    elif dataset_name == 'mnist':
        data = load_digits()
        data['data'] = 1*(data['data']>=8)

    X = data['data']
    y = data['target'].reshape(-1,1)

    if normalize_X == True :
        normalizer = Normalizer()
        X  = normalizer.fit_transform(X)

    if normalize_y == True :
        normalizer = Normalizer()
        y = normalizer.fit_transform(y)

    if one_hot_encode_y == True :
        encoder = OneHotEncoder()
        y = encoder.fit_transform(y).toarray()

    X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=test_size)
    return X_train, y_train, X_test, y_test
```

Stochastic Gradient Descent (SGD)

In [38]:

```
def forward_pass(X_sample, Y_sample, W, B, activation='linear', loss='mean_squared'):
    multiply_layer = MultiplicationLayer(X_sample, W)
    multiply_layer.forward()

    bias_add_layer = BiasAdditionLayer(multiply_layer.Z, B)
    bias_add_layer.forward()

    if activation == 'linear' :
        activation_layer = LinearActivation(bias_add_layer.Z)
    elif activation == 'softmax':
        activation_layer = SoftMaxActivation(bias_add_layer.Z)
    activation_layer.forward()

    if loss == 'mean_squared' :
        loss_layer = MeanSquaredLossLayer(Y_sample, activation_layer.aZ )
    elif loss=='cross_entropy' :
        loss_layer = CrossEntropyLossLayer(Y_sample, activation_layer.aZ )
    loss_layer.forward()


    return multiply_layer, bias_add_layer, activation_layer, loss_layer

def backward_pass(multiply_layer, bias_add_layer, activation_layer, loss_layer):

    loss_layer.backward()
    activation_layer.backward()
    bias_add_layer.backward()
    multiply_layer.backward()

    return loss_layer, activation_layer, bias_add_layer, multiply_layer


def StochasticGradientDescent( X_train,
                               y_train,
                               X_test,
                               y_test,
                               inp_shape = 1,    # dimension of input
                               out_shape = 1,    # dimension of output
                               n_iterations = 10000,
                               learning_rate = 0.01,
                               activation = 'linear',
                               loss = 'mean_squared',
                               seed = 42,
                               task='regression'   #  one of  [ 'regression', 'classificatio
                            ):

    np.random.seed(seed)

    # initialize W & B
    W_shape = ( inp_shape,  out_shape )
    B_shape = ( 1, out_shape )

    W = np.random.random(W_shape)
    B  = np.random.random(B_shape)

    iterations = trange(n_iterations ,desc="Training...", ncols=100)

    for iteration, _ in enumerate(iterations) :
```

```
        randomIndx = np.random.randint( len(X_train) )
        X_sample = X_train[randomIndx, :].reshape(1, inp_shape)
        Y_sample = y_train[randomIndx, :].reshape(1, out_shape)

        # Forward Pass
        # 1) Z <-- XW
        # 2) Z <-- Z + Bias
        # 3) Z <-- activation( Z )
        # 4) find Loss L

        multiply_layer, bias_add_layer, activation_layer, loss_layer = forward_pass(X_sampl

        # Note : here whenever I write aZ it means it is output of some activation function

        # Backward Pass
        # 1) dL/daZ
        # 2) dL/dZ = dL/daZ* daZ/dZ
        # 3) dL/dW = dZ/dW * dL/dZ
        # 4) dL/dB = dZ/dB * dL/dB

        loss_layer, activation_layer, bias_add_layer, multiply_layer = backward_pass(multip

        dL_daZ = loss_layer.dL_daZ
        dL_dZ = np.dot( activation_layer.daZ_dZ, dL_daZ )
        dL_dW = np.dot( multiply_layer.dZ_dW , dL_dZ.T)
        dL_dB = np.dot( bias_add_layer.dZ_dB, dL_dZ).T

        # Update W & B
        W -=  learning_rate*dL_dW
        B -=  learning_rate*dL_dB

        if iteration%1000 == 0 :
            iterations.set_description( "Sample Error : %0.5f"%loss_layer.L, refresh=True )

    # Lets run forward pass for train and test data and check accuracy/error


    if task =='regression':
        if isinstance(loss_layer, MeanSquaredLossLayer) :
            _ , _, _,  loss_layer = forward_pass( X_train, y_train , W, B, activation, loss
            print("Mean Squared Loss Error (Train Data)  : %0.5f"% loss_layer.L)

            _ , _, _,  loss_layer = forward_pass( X_test, y_test , W, B, activation, loss)
            print("Mean Squared Loss error (Test Data) : %0.5f"%loss_layer.L)

    if task =='classification':
        if isinstance(loss_layer, CrossEntropyLossLayer):
            y_true = np.argmax(y_train, axis=1)
            _, _, _, loss_layer = forward_pass( X_train, y_train , W, B, activation, loss)
            y_pred = np.argmax( loss_layer.aZ, axis=1)

            acc = 1*(y_pred == y_true)
            print("Classification Accuracy (Training Data ): {0}/{1} = {2} %".format(sum(ac

            y_true = np.argmax(y_test,axis=1)
            _, _, _, loss_layer = forward_pass( X_test, y_test , W, B, activation, loss)
            y_pred = np.argmax( loss_layer.aZ, axis=1)

            acc = 1*(y_pred == y_true)
            print("Classification Accuracy (Testing Data ): {0}/{1} = {2} %".format(sum(acc
```

Question 2

In [14]:

```
X_train, y_train, X_test, y_test  = load_data('boston', normalize_X=True, test_size=0.2)
```

In [15]:

```
StochasticGradientDescent(X_train, y_train, X_test, y_test, inp_shape=X_train.shape[1], out
```

```
Sample Error : 3.51963: 100%|| 10000/10000 [0
0:00<00:00, 27320.97it/s]

Mean Squared Loss Error (Train Data)  : 61.26899
Mean Squared Loss error (Test Data) : 58.32278
```

Question 3

In [16]:

```
X_train, y_train, X_test, y_test = load_data('iris',normalize_X=True, one_hot_encode_y=True
```

In [17]:

```
StochasticGradientDescent(X_train,y_train,X_test,y_test, inp_shape=X_train.shape[1], \
                          out_shape=y_train.shape[1],
                          n_iterations=5000,
                          learning_rate=0.001,
                          activation='softmax',
                          task='classification',
                          loss='cross_entropy')
```

```
Sample Error : 0.89642: 100%|| 5000/5000 [0
0:00<00:00, 19016.56it/s]

Classification Accuracy (Training Data ): 79/120 = 65.83333333333333 %
Classification Accuracy (Testing Data ): 21/30 = 70.0 %
```

Question 4 & 5 & 6

1. Question 4 - Implement neural network
2. Question 5 - predicting boston house price using different neural netowork architectures
3. Question 6 - classifying digits in MNIST dataset using different neural network architectures

Question 4

In [18]:

```
class Layer :
    """
    Input - activation : Activation Layer Name ,n_inp : dimension of input ,  n_out :  Numb
    """
    def __init__(self, n_inp, n_out, activation_name = "linear" , seed=42 ):

        np.random.seed(seed) # for reproducability of code

        self.n_inp = n_inp
        self.n_out = n_out

        # random initialization of input X  and output Z
        self.X = np.random.random( (1, n_inp))   # assigned during SGD
        self.Z  = np.random.random( (1, n_out ))

        # Initialize W & B with some scaling to avoid over-flow
        self.W  =  np.random.random( (n_inp, n_out) )* np.sqrt( 2 / ( n_inp + n_out) )
        self.B  = np.random.random( (1,n_out) )*np.sqrt( 2 / (1 + n_out ) )

        # define multiplication layer, bias addition layer , and activation layer
        self.multiply_layer = MultiplicationLayer(self.X, self.W)
        self.bias_add_layer = BiasAdditionLayer( self.B, self.B )

        if activation_name == 'linear' :
            self.activation_layer = LinearActivation(self.Z)
        elif activation_name == 'sigmoid' :
            self.activation_layer = SigmoidActivation(self.Z)
        elif activation_name == 'softmax' :
            self.activation_layer = SoftMaxActivation(self.Z)
        elif activation_name == 'tanh' :
            self.activation_layer = tanhActivation(self.Z)
        elif activation_name == 'relu' :
            self.activation_layer = ReLUActivation(self.Z)

    def forward(self,):
        self.multiply_layer.X = self.X
        self.multiply_layer.forward()

        self.bias_add_layer.Z = self.multiply_layer.Z
        self.bias_add_layer.forward()

        self.activation_layer.Z = self.bias_add_layer.Z
        self.activation_layer.forward()

        self.Z = self.activation_layer.aZ  # output of given layer


    def backward(self,):
        self.activation_layer.backward()
        self.bias_add_layer.backward()
        self.multiply_layer.backward()


class NeuralNetwork(Layer) :
    """
    Input  - layers : list of layer objects , loss_name : Name of loss layer
    """
    def __init__(self, layers, loss_name = "mean_squared", learning_rate = 0.01, seed=42):
        np.random.seed(seed)
```

```
        self.layers = layers
        self.n_layers = len(layers)  # number of layers in neural network
        self.learning_rate = learning_rate

        self.inp_shape = self.layers[0].X.shape
        self.out_shape = self.layers[-1].Z.shape

        # random initialization of input X  and output Z
        self.X = np.random.random( self.inp_shape)   # assigned during SGD
        self.Y  = np.random.random( self.out_shape )  # output of neural network

        #define loss layer
        if loss_name == "mean_squared" :
            self.loss_layer = MeanSquaredLossLayer( self.Y , self.Y )
        if loss_name == "cross_entropy" :
            self.loss_layer = CrossEntropyLossLayer( self.Y, self.Y )


    def forward(self,):
        self.layers[0].X = self.X
        self.loss_layer.Y = self.Y

        self.layers[0].forward()
        for i in range(1, self.n_layers ):
            self.layers[i].X = self.layers[i-1].Z
            self.layers[i].forward()

        self.loss_layer.aZ = self.layers[-1].Z
        self.loss_layer.forward()

    def backward(self,):

        self.loss_layer.Z = self.Y
        self.loss_layer.backward()
        self.grad_nn = self.loss_layer.dL_daZ
        for i in range( self.n_layers-1, -1, -1 ):
            self.layers[i].backward()

            dL_dZ =  np.dot( self.layers[i].activation_layer.daZ_dZ, self.grad_nn )
            dL_dW =  np.dot( self.layers[i].multiply_layer.dZ_dW , dL_dZ.T)
            dL_dB =  np.dot( self.layers[i].bias_add_layer.dZ_dB, dL_dZ).T

            # Update W & B
            self.layers[i].W -=  self.learning_rate*dL_dW
            self.layers[i].B -=  self.learning_rate*dL_dB

            # Update outer_grad
            self.grad_nn  = np.dot( self.layers[i].multiply_layer.dZ_daZ_prev, dL_dZ )

            del dL_dZ, dL_dW, dL_dB
```

In [19]:

```
def createLayers(inp_shape, layers_sizes, layers_activations):
    layers = []
    n_layers = len(layers_sizes)
    layer_0 = Layer( inp_shape, layers_sizes[0], layers_activations[0] )
    layers.append(layer_0)
    inp_shape_next = layers_sizes[0]
    for  i in range(1,n_layers):
        layer_i = Layer( inp_shape_next, layers_sizes[i], layers_activations[i] )
        layers.append(layer_i)
        inp_shape_next = layers_sizes[i]

    out_shape = inp_shape_next
    return inp_shape, out_shape, layers
```

In [20]:

```
def SGD_NeuralNetwork( X_train,
                       y_train,
                       X_test,
                       y_test,
                       nn,
                       inp_shape = 1,    # dimension of input
                       out_shape = 1,    # dimension of output
                       n_iterations = 1000,
                       task = "regression"  # [ "regression", "classification"]
                       ):
    iterations = trange(n_iterations,desc="Training ...", ncols=100)

    for iteration, _ in enumerate(iterations):
        randomIndx = np.random.randint( len(X_train) )
        X_sample = X_train[randomIndx, :].reshape(1, inp_shape)
        Y_sample = y_train[randomIndx, :].reshape(1, out_shape)

        nn.X = X_sample
        nn.Y = Y_sample

        nn.forward()  # Forward Pass
        nn.backward() # Backward Pass

    # Lets run ONLY forward pass for train and test data and check accuracy/error

    if task == "regression":
        nn.X = X_train
        nn.Y = y_train
        nn.forward()
        train_error = nn.loss_layer.L
        nn.X = X_test
        nn.Y = y_test

        nn.forward()

        test_error = nn.loss_layer.L

        if isinstance(nn.loss_layer , MeanSquaredLossLayer):
            print("Mean Squared Loss Error (Train Data)  : %0.5f"% train_error)
            print("Mean Squared Loss Error (Test Data)  : %0.5f"% test_error)

    if task == "classification":
        nn.X = X_train
        nn.Y = y_train
        nn.forward()
        y_true = np.argmax( y_train, axis=1)
        y_pred = np.argmax( nn.loss_layer.aZ, axis=1)
        acc = 1*(y_true == y_pred)
        print("Classification Accuracy (Training Data ): {0}/{1} = {2} %".format(sum(acc),

        nn.X = X_test
        nn.Y = y_test
        nn.forward()
        y_true = np.argmax( y_test, axis=1)
        y_pred = np.argmax( nn.loss_layer.aZ, axis=1)
        acc = 1*(y_true == y_pred)
        print("Classification Accuracy (Testing Data ): {0}/{1} = {2} %".format(sum(acc), l
```

Question 5

In [21]:

```
X_train, y_train, X_test, y_test  = load_data('boston', normalize_X=True, normalize_y=False
```

Heads Up : If you think that mean square error are high just normalize_y = True in above line and see Magic !!

Netowrk Architecture 1

- 1 Layer
- Layer 1 with one output neuron and linear activation
- Mean squared loss

In [22]:

```
inp_shape = X_train.shape[1]
layers_sizes = [1]
layers_activations = ['linear']

inp_shape, out_shape, layers = createLayers(inp_shape, layers_sizes, layers_activations)
loss_nn = 'mean_squared'

nn = NeuralNetwork(layers, loss_nn, learning_rate=0.1)

SGD_NeuralNetwork(X_train,y_train,X_test,y_test,nn,inp_shape, out_shape,n_iterations=11111,
```

```
Training ...: 100%|██████████████████████████████████████████| 11111/11111 [0
0:00<00:00, 26723.83it/s]

Mean Squared Loss Error (Train Data)  : 56.95890
Mean Squared Loss Error (Test Data)  : 54.84047
```

Network Architecture 2

- 2 Layers
- Layer 1 with 13 output neurons and sigmoid activation
- Layer 2 with 1 output with linear activation
- Mean Squared loss

In [23]:

```
inp_shape = X_train.shape[1]
layers_sizes = [13,1]
layers_activations = ['sigmoid','linear']

inp_shape, out_shape, layers = createLayers(inp_shape, layers_sizes, layers_activations)
loss_nn = 'mean_squared'

nn = NeuralNetwork(layers, loss_nn, learning_rate=0.01)

SGD_NeuralNetwork(X_train,y_train,X_test,y_test,nn,inp_shape, out_shape,n_iterations=1000,t
```

```
Training ...: 100%|██████████| 1000/1000 [0
0:00<00:00, 14778.15it/s]

Mean Squared Loss Error (Train Data)  : 64.88076
Mean Squared Loss Error (Test Data)  : 63.65933
```

Network Architecture 3

- 3 Layers
- Layer 1 with 13 output neurons and sigmoid activation
- Layer 2 with 13 output neurons and sigmoid activation
- Layer 3 with 1 output neuron and linear activation
- Mean Squared loss

In [24]:

```
inp_shape = X_train.shape[1]
layers_sizes = [13,13,1]
layers_activations = ['sigmoid','sigmoid','linear']

inp_shape, out_shape, layers = createLayers(inp_shape, layers_sizes, layers_activations)
loss_nn = 'mean_squared'

nn = NeuralNetwork(layers, loss_nn, learning_rate=0.001)

SGD_NeuralNetwork(X_train,y_train,X_test,y_test,nn,inp_shape, out_shape,n_iterations=1000,t
```

```
Training ...: 100%|██████████| 1000/1000 [0
0:00<00:00, 10241.55it/s]

Mean Squared Loss Error (Train Data)  : 83.19427
Mean Squared Loss Error (Test Data)  : 90.22964
```

Question 6

In [25]:

```
X_train, y_train, X_test, y_test = load_data('mnist', one_hot_encode_y=True, test_size=0.3)
```

Netowrk Architecture 1

- 2 Layer
- Layer 1 with 89 output neurons and tanh activation
- Layer 2 with 10 output neurons and sigmoid activation
- Mean squared loss

In [26]:

```
inp_shape = X_train.shape[1]
layers_sizes = [89,10]
layers_activations = ['tanh','sigmoid']

inp_shape, out_shape, layers = createLayers(inp_shape, layers_sizes, layers_activations)
loss_nn = 'mean_squared'

nn = NeuralNetwork(layers, loss_nn, learning_rate=0.1)

SGD_NeuralNetwork(X_train,y_train,X_test,y_test,nn,inp_shape, out_shape,n_iterations=10000,
```

```
Training ...: 100%|██████████████████████████████| 10000/10000 [0
0:00<00:00, 11620.20it/s]

Classification Accuracy (Training Data ): 1194/1257 = 94.98806682577566 %
Classification Accuracy (Testing Data ): 490/540 = 90.74074074074075 %
```

Netowrk Architecture 2

- 2 Layer
- Layer 1 with 89 output neurons and tanh activation
- Layer 2 with 10 output neurons and (linear activation + softmax activation) = softmax activation
- Cross Entropy Loss

In [27]:

```
inp_shape = X_train.shape[1]
layers_sizes = [89,10]
layers_activations = ['tanh','softmax']

inp_shape, out_shape, layers = createLayers(inp_shape, layers_sizes, layers_activations)
loss_nn = 'cross_entropy'

nn = NeuralNetwork(layers, loss_nn, learning_rate=0.01)

SGD_NeuralNetwork(X_train,y_train,X_test,y_test,nn,inp_shape, out_shape,n_iterations=11111,
```

```
Training ...: 100%|██████████████████████████████| 11111/11111 [0
0:01<00:00, 10346.31it/s]

Classification Accuracy (Training Data ): 1201/1257 = 95.54494828957836 %
Classification Accuracy (Testing Data ): 494/540 = 91.48148148148148 %
```

Question 7

Convolutional Layer for 1-channel input and 1 channel output + flatten operation

All the assumptions are taken for 1 channel input and (n = 1) channel output

We are assuming that we have filter, input- inp define filter for convolutional layer, we are fixing the size to be 3x3 and stride to be 1. if it is not then we may face difficulty in finding proper zero-padding for the input

In next question we have implemented multi-channel input and multi-channel out with filter-size given by user. Stride in next question is also taken 1 as not only we will face finding proper zero-padding for input but also gradient calculation will become more tricky in backpropagation. For sake of simplicity we only considered stride=1 case

In [29]:

```
#Assuming we are given single channel input and initial filter to be a 3x3 matrix:

def convolutional_layer(zero_pad_input, l_filter):

    l = len(inp) #length of input matrix
    m = len(l_filter)  #length of filter
    c = len(zero_pad_input)  #size of zero-padded matrix
    s  = (c - m) + 1  #to be used for loop for filtering
    out = np.zeros((l, l))  #output after convolution

    #filtering-
    for i in range(s):
        for j in range(s):
            temp = np.zeros((m,m))
            row, col = np.indices((m,m))
            temp = np.multiply(zero_pad_input[row+i, col+j], l_filter)

            out[i][j] = np.sum(temp)

    return out

#----------------------------------------------------------------------------------



#Forward pass implementation-
def Forward_pass(inp, l_filter):
    l = len(inp)
    #Zero-padding of input layer-
    zero_pad_input = np.zeros((l+2, l+2))
    zero_pad_input[ 1:l+1, 1:l+1] = inp

    f_out = convolutional_layer(zero_pad_input, l_filter)
    return f_out


#----------------------------------------------------------------------------------

# Function to Rotate
# the matrix by 180 degree
def rotateMatrix(mat):
    N = len(mat)
    rot_mat = np.zeros((N,N))
    k = N - 1
    t1 = 0
    while(k >= 0 and t1 < 3):
        j = N - 1;
        t2 = 0
        while(j >= 0 and t2 < N):
            rot_mat[t1][t2] = mat[k][j]
            j = j - 1
            t2 = t2 + 1
        k = k - 1
        t1 = t1 + 1

    return rot_mat
#----------------------------------------------------------------------------------
```

```
#Backward pass implementation-

def Backward_pass(inp, output, l_filter):
    l = len(inp)

#--------------------------------Backward Pass--------------------------------------
    #Zero-padding of input layer-
    zero_pad_input = np.zeros((l+2, l+2))
    zero_pad_input[ 1:l+1, 1:l+1] = inp

    grad_filter = convolutional_layer(zero_pad_input, output)
    #we can use gradient of filter coefficient matrix to update the filter matrix:
    #-- l_filter - l_filter - alpha*grad_filter ,where alpha is learning rate

    #for gradient of loss w.r.t input, we need to rotate the filter by 180° and apply convo
    rotated_filter = rotateMatrix(l_filter)
    zero_pad_output = np.zeros((l+2, l+2))
    zero_pad_output[ 1:l+1, 1:l+1] = output
    grad_X = convolutional_layer(zero_pad_output, rotated_filter)

    return grad_filter, grad_X

#----------------------------------------------------------------------------------


#flatten operation:

def flatten(inp_mat):
    flatten_vector = []

    for i in range(len(inp_mat)):  #number of rows
        for j in range(len(inp_mat[0])):  #number of columns
            flatten_vector.append(inp_mat[i][j])

    flatten_vector = np.array(flatten_vector)
    return flatten_vector

#----------------------------------------------------------------------------------
```

Test Example

In [30]:

```
inp = np.array([[1,2,3,4],[2,3,4,5],[7,8,97,1],[1,2,3,4]])
l_filter = np.array([[1,0,0],[0,1,0],[0,0,1]])
forward_out = Forward_pass(inp,l_filter)
print('output for forward pass', forward_out)
```

```
output for forward pass [[  4.   6.   8.   4.]
 [ 10. 101.   7.   8.]
 [  9.  13. 104.   5.]
 [  1.   9.  11. 101.]]
```

In [31]:

```
dL_df, dL_dX= Backward_pass(inp,forward_out,l_filter)
t = np.zeros((3,3))
t = dL_df[:3, :3]  #assuming filter is of size 3x3
dL_df = t

print('Gradient of loss w.r.t filter from Backward pass:', '\n', dL_df)
print('Gradient of loss w.r.t input from Backward pass:', '\n',dL_dX)
```

```
Gradient of loss w.r.t filter from Backward pass:
 [[10445.  2010.  1842.]
 [ 2031. 11163.  2037.]
 [ 1827.  2010. 10433.]]
Gradient of loss w.r.t input from Backward pass:
 [[105.  13.  16.   4.]
 [ 23. 209.  18.  16.]
 [ 18.  34. 306.  12.]
 [  1.  18.  24. 205.]]
```

Question 8 & 9

- Question 8 : Convolutional Neural Network with multi-channel input and multi-channel output.

```
filter shape can be given by user (default = (1,1))
Stride is taken as 1 for reason explained in previous question
```

- Question 9 : MNIST hand written digits classification using CNN

Question 8 : Convolutional Neural Network

In [32]:

```
class ConvolutionalLayer :
    """
    Implementation of Convolutional Layer consist of Convolution  followed by flattening  a
    """
    def __init__(self,
            inp_shape ,                                         # inp_shape = (input_c
            activation ='tanh' ,
            filter_shape = (1,1),                               # filter_shape = (filt
            lr = 0.01,
            Co = 1 ,
            seed = 42):                                          # number of output c

        np.random.seed(seed)
        # Check if filter is valid or NOT
        assert ( inp_shape[1]>=filter_shape[0] and inp_shape[2]>= filter_shape[1]) , \
             "Error : Input {} incompatible with filter {}".format(inp.shape, filter_shape)

        self.inp = np.random.rand(*inp_shape)
        self.inp_shape = inp_shape
        self.Ci = self.inp.shape[0]                              # number of channels
        self.Co = Co                                             # number of output c
        self.filters_shape = ( self.Co , self.Ci,  *filter_shape )
        self.out_shape = (self.Co, self.inp.shape[1] - filter_shape[0] + 1, self.inp.shape[
        self.flatten_shape = self.out_shape[0]*self.out_shape[1]*self.out_shape[2]
        self.lr = lr

        # Randomly initialize filters, biases, output, flatten output
        self.filters = np.random.rand( *self.filters_shape )
        self.biases  = np.random.rand( *self.out_shape )
        self.out = np.random.rand( *self.out_shape )
        self.flatten_out = np.random.rand(1,self.flatten_shape)

        # Define activation function
        if activation == 'tanh':
            self.activation_layer = tanhActivation( self.out )


    def forward(self, ) :
        self.out = np.copy( self.biases )  # add bias to output
        for i in range( self.Co ) :
            for j in range( self.Ci ) :
                self.out[i] += self.convolve(self.inp[j], self.filters[i,j])

        self.flatten()
        self.activation_layer.Z = self.flatten_out
        self.activation_layer.forward()

    def backward(self, grad_nn ):

        self.activation_layer.backward()
        loss_gradient = np.dot( self.activation_layer.daZ_dZ, grad_nn )
        loss_gradient = np.reshape(loss_gradient, self.out_shape)  # reshape to (Co, H_out,

        self.filters_gradient = np.zeros( self.filters_shape )  # dL/dKij for each filter
        self.input_gradient = np.zeros( self.inp_shape )  #  dL/dXj
        self.biases_gradient = loss_gradient  # dL/dBi  = dL/dYi
        padded_loss_gradient = np.pad( loss_gradient, ((0,0), (self.filters_shape[2]-1,self

        for i in range(self.Co):
```

```
            for j in range( self.Ci ):
                self.filters_gradient[i,j] = self.convolve( self.inp[j], loss_gradient[i] )
                rot180_Kij = np.rot90( np.rot90( self.filters[i,j], axes=(0,1) ) , axes=(0,
                self.input_gradient[j] += self.convolve( padded_loss_gradient[i], rot180_Ki

        # update filters and biases
        self.filters -= self.lr*self.filters_gradient
        self.biases -= self.lr*self.biases_gradient


    # flattening output to 1 Dimension so it can be fed int neural network
    def flatten(self, ):
        self.flatten_out = self.out.reshape(1,-1)

    # convolutional operation with stride=1
    def convolve(self, x, y ):
        x_conv_y  = np.zeros((x.shape[0] - y.shape[0] +1 , x.shape[1] - y.shape[1] + 1))
        for i in range(x.shape[0]-y.shape[0] + 1)   :
            for j in range( x.shape[1] - y.shape[1] + 1) :
                tmp = x[i:i+y.shape[0], j:j+y.shape[1] ]
                tmp = np.multiply(tmp, y)
                x_conv_y[i,j] = np.sum( tmp )
        return x_conv_y
```

In [33]:

```
class CNN :
    """
    Implementation of Convolutional Neural Network
    """
    def __init__(self,
                 convolutional_layer,                     # convolutional layer
                 nn,                                      # feed forward neural network
                 seed = 42):

        self.nn = nn
        self.convolutional_layer = convolutional_layer
        self.X = _ # assigned during SGD
        self.Y = _ # assigned during SGD

    def forward(self,):
        # forward pass of convolutional layer
        self.convolutional_layer.inp = self.X
        self.convolutional_layer.forward()

        # forward pass of neural network
        self.nn.X = self.convolutional_layer.activation_layer.aZ
        self.nn.Y = self.Y
        self.nn.forward()

    def backward(self,):
        # backward pass of neural network
        self.nn.backward()

        # backward pass of convolutional network
        self.convolutional_layer.backward( self.nn.grad_nn )
```

In [34]:

```
def SGD_CNN(X_train,
            y_train,
            X_test,
            y_test,
            cnn,
            inp_shape,
            out_shape,
            n_iterations=1000,
            task="classification"):

    iterations = trange(n_iterations,desc="Training ...", ncols=100)

    for iteration, _ in enumerate(iterations):
        randomIndx = np.random.randint( len(X_train) )
        X_sample = X_train[randomIndx, :].reshape(inp_shape)
        Y_sample = y_train[randomIndx, :].reshape(out_shape)

        cnn.X = X_sample
        cnn.Y = Y_sample

        cnn.forward()  # Forward Pass
        cnn.backward() # Backward Pass

    # Lets run ONLY forward pass for train and test data and check accuracy/error

    if task == "classification":
        X_train = X_train.reshape(-1,8,8)
        y_true = np.argmax( y_train, axis=1)
        acc = 0
        for i in range( len(X_train) ):
            cnn.X = X_train[i][np.newaxis, :, :]
            cnn.Y = y_train[i]
            cnn.forward()
            y_pred_i = np.argmax( cnn.nn.loss_layer.aZ, axis=1)
            if (y_pred_i == y_true[i]) : acc += 1
        print("Classification Accuracy (Training Data ): {0}/{1} = {2} %".format(acc, len(y

        X_test = X_test.reshape(-1,8,8)
        y_true = np.argmax( y_test, axis=1)
        acc = 0
        for i in range( len(X_test) ):
            cnn.X = X_test[i][np.newaxis, :, :]
            cnn.Y = y_test[i]
            cnn.forward()
            y_pred_i = np.argmax( cnn.nn.loss_layer.aZ, axis=1)
            if (y_pred_i == y_true[i]) : acc += 1
        print("Classification Accuracy (Testing Data ): {0}/{1} = {2} %".format(acc, len(y_
```

Question 9 : MNIST hand written digit classification using CNN

In [35]:

```
np.random.random((2,3,4)).size
```

Out[35]:

24

In [36]:

```
X_train, y_train, X_test, y_test = load_data('mnist', one_hot_encode_y=True)
```

Network Architecture

- 2 Layers
- Layer 1 with 16 output channels + flatten + tanh activation
- Layer 2 with 10 output neuron with (linear activation + softmax activation) = softmax activation
- Cross Entropy loss

In [37]:

```
conv_inp_shape = (1,8,8)   # sklearn digit dataset has images of shape 1 x 8 x 8
Co = 16  # 16 channel output
conv_filter_shape = (3,3)
conv_activation = 'tanh'
convolutional_layer = ConvolutionalLayer(conv_inp_shape,
                                        filter_shape = conv_filter_shape,
                                        Co = Co,
                                        activation = conv_activation,
                                        lr = 0.01)
nn_inp_shape = convolutional_layer.flatten_shape
layers_sizes = [10]
layers_activations = ['softmax']

nn_inp_shape, nn_out_shape, layers = createLayers(nn_inp_shape, layers_sizes, layers_activa
loss_nn = 'cross_entropy'

nn = NeuralNetwork(layers, loss_nn, learning_rate=0.01)

cnn = CNN( convolutional_layer, nn)
out_shape =  (1, layers_sizes[-1])  # one_hot encoded ouptut

SGD_CNN(X_train,y_train,X_test,y_test, cnn,conv_inp_shape, out_shape,n_iterations=5000)
```

```
Training ...: 100%|██████████████████████████████████████████| 5000/5000
[01:05<00:00, 76.47it/s]

Classification Accuracy (Training Data ): 1295/1437 = 90.11830201809325 %
Classification Accuracy (Testing Data ): 333/360 = 92.5 %
```

Type *Markdown* and LaTeX: α^2