CS331 - Lab Assignment 2

Group ID = AyKaGaRo

Deep Learning

```
## Classes containing init, forward and backward functions.
```

```
## BASE FUNCTIONS

class _multiplication_layer:
    def __init__(self, X, W):
        self.X = X # X (numpy.ndarray): Input matrix.
        self.W = W # W (numpy.ndarray): Weight matrix.

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

    def backward(self):
        self.dZ_dW = (self.X).T
        self.dZ_daZ_prev = self.W

class _bias_addition_layer:
    def __init__(self, Z, b):
        self.B = b # b (numpy.ndarray): Bias matrix.
        self.Z = Z #  Z (numpy.ndarray): Input matrix.

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

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

```
: ## LOSSES

class _mean_squared_error_loss:
    def __init__(self, Y, Y_hat):
        self.Y = Y #  Y (numpy.ndarray): True labels.
        self.aZ = Y_hat # Y_hat (numpy.ndarray): Predicted labels.

    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


class _cross_entropy_loss:
    def __init__(self, Y, Y_pred): # Constructor for the cross-entropy
        self.Y = Y
        self.aZ = Y_pred
        self.epsilon = 1e-20

    def forward(self):
        """
        Forward pass of the cross-entropy loss.
        Computes the cross-entropy loss and stores the result in L.
        """
        self.L = - np.sum(self.Y * np.log(self.aZ+self.epsilon))

    def backward(self):
    # Computes the gradient with respect to the predicted labels (dL/da
        self.dL_daZ = -1*(self.Y/(self.aZ + self.epsilon)).T
```

:

```
## ACTIVATION FUNCTIONS

class _softmax:
    def __init__(self, Z):
        """
        Constructor for the softmax activation function.
        """
        self.Z = Z #  Z (numpy.ndarray): Input matrix.

    def forward(self):
        """
        Computes the softmax activation and stores the result in aZ.
        """
        max_Z = np.max( self.Z, axis=1 ,keepdims=True )
        self.aZ = (np.exp(self.Z - max_Z ))/np.sum( np.exp(self.Z - max

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

```python
class _sigmoid:
    def __init__(self, Z):
        """
        Constructor for the sigmoid activation function.
        """
        self.Z = Z

    def forward(self,):
        """
        Computes the sigmoid activation and stores the result in aZ.
        """
        self.aZ = 1./(1 + np.exp(-self.Z))

    def backward(self,):
        """
        Backward pass of the sigmoid activation function.
        Computes the Jacobian matrix of the sigmoid activation (daZ/dZ)
        """
        diag_entries = np.multiply(self.aZ, 1-self.aZ).reshape(-1)
        self.daZ_dZ = np.diag(diag_entries)
```

```python
class _linear:
    def __init__(self, Z):
        """
        Constructor for the linear activation function.
        """
        self.Z = Z # Z (numpy.ndarray): Input matrix.

    def forward(self, ):
        """
        Forward pass of the linear activation function.
        Directly sets aZ equal to Z.
        """
        self.aZ = self.Z

    def backward(self,):
        """
        Backward pass of the linear activation function.
        Computes the Jacobian matrix of the linear activation (daZ/dZ).
        """
        self.daZ_dZ = np.identity( self.Z.shape[1] )
```

```python
class _tanh:
    def __init__(self, Z):
        """
        Constructor for the hyperbolic tangent (tanh) activation functi
        """
        self.Z = Z # Z (numpy.ndarray): Input matrix.

    def forward(self,):
        """
        Forward pass of the tanh activation function.
        Computes the tanh activation and stores the result in aZ.
        """
        self.aZ = np.tanh(self.Z)

    def backward(self,):
        """
        Backward pass of the tanh activation function.
        Computes the Jacobian matrix of the tanh activation (daZ/dZ).
        """
        self.daZ_dZ = np.diag(1 - self.aZ.reshape(-1)**2)
```

```python
class _relu:
    def __init__(self, Z):
        """
        Constructor for the rectified linear unit (ReLU) activation fun

        Parameters:
        Z (numpy.ndarray): Input matrix.
        """
        self.Z = Z
        self.Leak = 0.01

    def forward(self,):
        """
        Forward pass of the ReLU activation function.
        Computes the ReLU activation and stores the result in aZ.
        """
        self.aZ = np.maximum(self.Z,0)

    def backward(self,):
        """
        Backward pass of the ReLU activation function.
        Computes the Jacobian matrix of the ReLU activation (daZ/dZ).
        """
        self.daZ_dZ = np.diag( [1. if x>=0 else self.Leak for x in self
```

```python
## For loading the dataset
```

```python
def load_data(dataset_name='boston', normalize_X=False, normalize_y=Fals
    """
    Load and preprocess different datasets based on the specified datase

    Parameters:
    - dataset_name (str): Name of the dataset to load ('boston', 'iris'
    - normalize_X (bool): Flag to indicate whether to normalize the fea
    - normalize_y (bool): Flag to indicate whether to normalize the tar
    - one_hot_encode_y (bool): Flag to indicate whether to perform one-h
    - test_size (float): Size of the test set when splitting the data.

    Returns:
    - X_train (numpy.ndarray): Training features.
    - y_train (numpy.ndarray): Training target variable.
    - X_test (numpy.ndarray): Test features.
    - y_test (numpy.ndarray): Test target variable.
    """
```

```
    if dataset_name == 'boston':
        # Load Boston dataset from URL
        data_url = "http://lib.stat.cmu.edu/datasets/boston"
        raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=N
        data_boston = np.hstack([raw_df.values[::2, :], raw_df.values[1
        result_boston = raw_df.values[1::2, 2]
        data = {'data': data_boston, 'target': result_boston}
    elif dataset_name == 'iris':
        # Load Iris dataset from sklearn
        data = load_iris()
    elif dataset_name == 'mnist':
        # Load MNIST-like dataset from sklearn and binarize it
        data = load_digits()
        data['data'] = 1*(data['data'] >= 8)
    # Extract features (X) and target variable (y)
    X = data['data']
    y = data['target'].reshape(-1, 1)

    # Normalize features if specified
    if normalize_X == True:
        normalizer = Normalizer()
        X = normalizer.fit_transform(X)

    # Normalize target variable if specified
    if normalize_y == True:
        normalizer = Normalizer()
        y = normalizer.fit_transform(y)
    # One-hot encode target variable if specified
    if one_hot_encode_y == True:
        encoder = OneHotEncoder()
        y = encoder.fit_transform(y).toarray()

    # Split the data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size

    return X_train, y_train, X_test, y_test
```

```
## Layer and Neural Network Class
```

```
class Layer:
    def __init__(self, inSize, outSize, activation="linear", seed=42):
        """
        Constructor for the neural network layer.

        - inSize (int): Number of input features.
        - outSize (int): Number of output features.
        - activation (str): Activation function to use ('linear', 'relu
        - seed (int): Seed for random number generation.
        """
        np.random.seed(seed)

        self.inSize = inSize
        self.outSize = outSize

        self.X = np.random.random((1, inSize))

        self.W = np.random.random((inSize, outSize))
        self.B = np.random.random((1, outSize))

        self.Z = np.random.random((1, outSize))

        # Initialize sub-layers
        self.mul_layer = _multiplication_layer(self.X, self.W)
        self.bias_layer = _bias_addition_layer(self.B, self.B)

        # Initialize activation layer based on specified activation func
        if activation == 'linear':
            self.activation_layer = _linear(self.Z)
        elif activation == 'relu':
            self.activation_layer = _relu(self.Z)
        elif activation == 'tanh':
            self.activation_layer = _tanh(self.Z)
        elif activation == 'sigmoid':
            self.activation_layer = _sigmoid(self.Z)
        elif activation == 'softmax':
            self.activation_layer = _softmax(self.Z)
        else:
            pass
```

Task : Create a Neural Network Class

A neural network is a computational model inspired by the human brain. It consists of layers of interconnected nodes, each node representing a neuron. The neural network class encapsulates the structure and functionality of these layers.

```
class NeuralNetwork(Layer):

    def __init__(self, layerList, loss="mean_squared", lr=0.01, seed=42
        """
        Constructor for the neural network.

        - layerList (list): List of Layer objects representing the netw
        - loss (str): Loss function to use ('mean_squared', 'cross_entr
        - lr (float): Learning rate for optimization.
        - seed (int): Seed for random number generation.
        """
        np.random.seed(seed)

        self.layerList = layerList
        self.num_layers = len(layerList)
        self.lr = lr

        self.inShape = self.layerList[0].X.shape
        self.outShape = self.layerList[-1].Z.shape

        self.X = None
        self.Y = None

        # Initialize the loss layer based on the specified loss functio
        if loss == "mean_squared":
            self.loss_layer = _mean_squared_error_loss(self.Y, self.Y)
        if loss == "cross_entropy":
            self.loss_layer = _cross_entropy_loss(self.Y, self.Y)
```

```
    def forward(self):
        self.layerList[0].X = self.X
        self.loss_layer.Y = self.Y

        self.layerList[0].forward()

        for i in range(1, self.num_layers):
            self.layerList[i].X = self.layerList[i-1].Z
            self.layerList[i].forward()

        self.loss_layer.aZ = self.layerList[-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.num_layers-1, -1, -1):
            self.layerList[i].backward()

            dL_dZ = np.dot(self.layerList[i].activation_layer.daZ_dZ, s
            dL_dW = np.dot(self.layerList[i].mul_layer.dZ_dW, dL_dZ.T)
            dL_dB = np.dot(self.layerList[i].bias_layer.dZ_dB, dL_dZ).T

            self.layerList[i].W -= self.lr*dL_dW
            self.layerList[i].B -= self.lr*dL_dB

            self.grad_nn = np.dot(self.layerList[i].mul_layer.dZ_daZ_pr

            del dL_dZ, dL_dW, dL_dB
```

TASK : Implement Specific Neural Networks

Linear Regression Model: Linear regression is a supervised learning algorithm for predicting a continuous outcome variable based on one or more predictor variables. It uses a linear relationship (Wx + b) between input features (x) and the output (prediction).

Two Layers with Sigmoid and Linear Activation: This model has two layers, the first with sigmoid activation for non-linearity, and the second with linear activation. It is suitable for non-linear mapping and regression tasks.

Three Layers with Sigmoid Activation: Similar to the second model, it adds an additional layer for increased model complexity and potential feature representation.

```
## SGD for Artificial Neural Network
```

```python
def makeLayers(inShape, layerSize, activation):
    layers = []
    n_layers = len(layerSize)

    for i in range(0, n_layers):
        if i==0:
            layer_i = Layer(inShape, layerSize[i], activation[i])
        else:
            layer_i = Layer(outShape, layerSize[i], activation[i])
        layers.append(layer_i)
        outShape = layerSize[i]

    return inShape, outShape, layers
```

```python
def StocasticGradiantDescentArtificialNeuralNetwork(X_train, y_train, X_

    for _ in range(epochs):
        randomIndx = np.random.randint(len(X_train))
        X_sample = X_train[randomIndx, :].reshape(1, inpShape)
        Y_sample = y_train[randomIndx, :].reshape(1, outShape)

        nn.X = X_sample
        nn.Y = Y_sample

        nn.forward()
        nn.backward()

    if problem == "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

        print("Train Data MSE : %0.5f" % train_error)
        print("Test Data MSE  : %0.5f" % test_error)

    if problem == "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("Training Data Accuracy : {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("Testing Data : {0}/{1} = {2} %".format(sum(acc), len(acc
```

TASK : Train the Models on Boston Dataset

The Boston dataset contains housing-related features, and the goal is to predict house prices. Training involves adjusting the model's parameters (weights and biases) using Stochastic Gradient Descent (SGD) to minimize the mean squared error loss.

```python
# Load and preprocess the Boston dataset
X_train, y_train, X_test, y_test = load_data('boston', normalize_X=True

# 'boston': Specify the dataset to load as the Boston Housing dataset.
# normalize_X=True: Normalize the features (X) using sklearn's Normaliz
# normalize_y=False: Do not normalize the target variable (y).
# test_size=0.2: Set the test set size to 20% of the total dataset.

# Resulting variables:
# X_train: Training features after normalization.
# y_train: Training target variable without normalization.
# X_test: Test features after normalization.
# y_test: Test target variable without normalization.
```

:

```python
def makeLayers(inShape, layerSize, activation):
    """
    Create a list of Layer objects to represent the layers of the neura
    - inShape (int): Number of input features.
    - layerSize (list): List of layer sizes for each layer in the netwo
    - activation (list): List of activation functions for each layer in
    - inShape (int): Number of input features.
    - outShape (int): Number of output features.
    - layers (list): List of Layer objects representing the layers of th
    """
    layers = []
    n_layers = len(layerSize)

    for i in range(0, n_layers):
         # Create a new layer based on the specified parameters
        if i==0:
            layer_i = Layer(inShape, layerSize[i], activation[i])
        else:
            layer_i = Layer(outShape, layerSize[i], activation[i])
        # Append the newly created layer to the list of layers
        layers.append(layer_i)

        # Update outShape for the next iteration
        outShape = layerSize[i]

    return inShape, outShape, layers
```

QUE 2.

Using the above create following networks with the following =

- **just one output neural with linear activation and least mean square loss.**
(This is linear regression).

- **two layers. Layer 1 with 13 output neurons with sigmoid activation. Layer 2 with one output neuron and linear activation. use mean squared loss**

- **three layers. Layer 1 with 13 output neurons with sigmoid activation.**

Layer 2 with 13 output neurons and sigmoid activation. Layer 3 with oneoutput neuron and linear activation. use mean squared loss Train this model on boston dataset using SGD.

2.1

```
# Extract the number of input features from the training features
inShape = X_train.shape[1]

# Specify the neural network architecture
size = [1]  # Number of neurons in each layer
layers_activations = ['linear']  # Activation function for each layer

# Create the neural network layers using the specified architecture
inShape, outShape, layers = makeLayers(inShape, size, layers_activation

# Initialize and train the neural network using Stochastic Gradient Desc
# - NeuralNetwork(layers, "mean_squared", lr=0.1): Create a neural netw
# - inShape, outShape: Number of input and output features.
# - epochs=10000: Number of training epochs.
# - problem="regression": Specify the type of problem as regression.
StocasticGradiantDescentArtificialNeuralNetwork(X_train, y_train, X_tes
```

```
Train Data MSE : 54.54307
Test Data MSE  : 37.15076
```

2.2

```
# Extract the number of input features from the training features
inp_shape = X_train.shape[1]

# Specify the neural network architecture
size = [13, 1]  # Number of neurons in each layer
layers_activations = ['sigmoid', 'linear']  # Activation function for e

# Create the neural network layers using the specified architecture

inp_shape, out_shape, layers = makeLayers(inp_shape, size, layers_activa
StocasticGradiantDescentArtificialNeuralNetwork(X_train, y_train, X_tes
```

```
Train Data MSE : 72.76918
Test Data MSE  : 61.25141
```

2.3

```
# Extract the number of input features from the training features
inp_shape = X_train.shape[1]

# Specify the neural network architecture
size = [13,13,1]  # Number of neurons in each layer
layers_activations = ['sigmoid', 'sigmoid', 'linear']  # Activation fun

# Create the neural network layers using the specified architecture
inp_shape, out_shape, layers = makeLayers(inp_shape, size, layers_activ

StocasticGradiantDescentArtificialNeuralNetwork(X_train, y_train, X_tes
```

```
Train Data MSE : 87.78990
Test Data MSE  : 72.83930
```

```
# Load and preprocess the MNIST-like dataset
X_train, y_train, X_test, y_test = load_data('mnist', one_hot_encode_y=
```

3. Using the above create following networks with the following
• two layers. Layer 1 with 89 output neurons with tanh activation. Layer 2 with ten output neuron and sigmoid activation. use mean squared loss
• two layers. Layer 1 with 89 output neurons with tanh activation. Layer 2 with ten output neuron and linear activation. use softmax with cross
entropy loss. Train this model on mnist (sklearn) dataset using SGD.

TASK : Train Models on MNIST Dataset

First Network (Two Layers): This network uses tanh activation for the first layer with 89 neurons and sigmoid activation for the second layer with 10 neurons. It aims to classify handwritten digits using mean squared loss.

Second Network (Two Layers): Similar to the first, but with linear activation for the second layer and softmax with cross-entropy loss. It is designed for multi-class classification tasks.

3.1

```
# Extract the number of input features from the training features
inp_shape = X_train.shape[1]

# Specify the neural network architecture
layers_sizes = [89, 10]  # Number of neurons in each layer
layers_activations = ['tanh', 'sigmoid']  # Activation function for eac

# Create the neural network layers using the specified architecture
inp_shape, out_shape, layers = makeLayers(inp_shape, layers_sizes, laye

# Specify the loss function for the neural network
loss_nn = 'mean_squared'

# Initialize the neural network with specified layers, loss function, a
nn = NeuralNetwork(layers, loss_nn, lr=0.1)

StocasticGradiantDescentArtificialNeuralNetwork(X_train,y_train,X_test,
```

```
Training Data Accuracy : 127/1257 = 10.103420843277645 %
Testing Data : 51/540 = 9.444444444444445 %
```

3.2

```
# Extract the number of input features from the training features
inp_shape = X_train.shape[1]

# Specify the neural network architecture
layers_sizes = [89, 10]  # Number of neurons in each layer
layers_activations = ['tanh', 'softmax']  # Activation function for eac

# Create the neural network layers using the specified architecture
inp_shape, out_shape, layers = makeLayers(inp_shape, layers_sizes, laye

# Specify the loss function for the neural network
loss_nn = 'cross_entropy'

# Initialize the neural network with specified layers, loss function, a

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

# Train the neural network using Stochastic Gradient Descent
StocasticGradiantDescentArtificialNeuralNetwork(X_train,y_train,X_test,
```

```
Training Data Accuracy : 127/1257 = 10.103420843277645 %
Testing Data : 51/540 = 9.444444444444445 %
```

4. Implement the convolution layer for 1 channel input and (n >= 1) channel output. Implement both forward and backward passes. Implement the flatten Operation.

5. (extra credit bonus:) generalize this for any number of input and any number of output channel. Implement both forward and backward passes

6. Train this CNN on mnist dataset. Layer 1: Convolution layer with 16 out-
put channels+flatten+tanh activation. Layer 2: 10 output neuron with linear activation. Softmax cross entropy loss

TASK : Implement Convolution Layer

Convolutional layers are fundamental in image processing for feature extraction. They involve sliding a filter (kernel) over the input to detect patterns. The flatten operation reshapes the output into a vector for further processing.

```
## Convolutional Layer and Convolutional Neural Network
```

```
class ConvolutionalLayer:
    def __init__(self, inp_shape, activation='tanh', filter_shape=(1, 1
        """
        Initialize a Convolutional Layer.
        - inp_shape (tuple): Shape of the input data (channels, height,
        - activation (str): Activation function for the layer.
        - filter_shape (tuple): Shape of the filters (channels, height,
        - lr (float): Learning rate for gradient descent.
        - Co (int): Number of output channels (number of filters).
        - seed (int): Random seed for reproducibility.
        """

        np.random.seed(seed)

        self.inp = np.random.rand(*inp_shape)
        self.inp_shape = inp_shape

        self.Ci = self.inp.shape[0]
        self.Co = Co
        self.filters_shape = (self.Co, self.Ci, *filter_shape)
        self.out_shape = ( self.Co, self.inp.shape[1] - filter_shape[0]
        self.flatten_shape = self.out_shape[0] * self.out_shape[1]*self
        self.lr = lr

        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.flat = np.random.rand(1, self.flatten_shape)

        if activation == 'tanh':
            self.activation_layer = _tanh(self.out)
```

```python
    def flatten(self):
        """
        Flatten the output for further processing.
        """
        self.flat = self.out.reshape(1, -1)

    def convolve(self, x, y):
        """
        Perform convolution operation between two matrices x and y.
        - x (numpy.ndarray): Input matrix.
        - y (numpy.ndarray): Filter matrix.
        """
        m = x.shape[0] - y.shape[0] + 1
        n = x.shape[1] - y.shape[1] + 1
        x_conv_y = np.zeros((m, n))
        for i in range(m):
            for j in range(n):
                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

    def forward(self):
        """
        Perform forward pass through the convolutional layer.
        """
        self.out = np.copy(self.biases)
        for i in range(self.Co):
            for j in range(self.Ci):
                self.out[i] += self.convolve(self.inp[j], self.filters[

        self.flatten()
        self.activation_layer.Z = self.flat
        self.activation_layer.forward()
```

```python
    def backward(self, grad_nn):
        """
        Perform backward pass through the convolutional layer.
        - grad_nn (numpy.ndarray): Gradient from the neural network.

        Updates:
        - self.filters: Update filter weights.
        - self.biases: Update biases.
        """

        self.activation_layer.backward()
        loss_gradient = np.dot(self.activation_layer.daZ_dZ, grad_nn)
        loss_gradient = np.reshape(loss_gradient, self.out_shape)

        self.filters_gradient = np.zeros(self.filters_shape)
        self.input_gradient = np.zeros(self.inp_shape)
        self.biases_gradient = loss_gradient
        padded_loss_gradient = np.pad(loss_gradient, ((
            0, 0), (self.filters_shape[2]-1, self.filters_shape[2]-1),

        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_Kij)

        self.filters -= self.lr*self.filters_gradient
        self.biases -= self.lr*self.biases_gradient
```

```python
class ConvolutionalNeuralNetwork :
    def __init__(self, convolutional_layer, nn, seed = 42):
        """
        Initialize a Convolutional Neural Network.
        - convolutional_layer (ConvolutionalLayer): Convolutional layer
        - nn (NeuralNetwork): Neural network instance.
        - seed (int): Random seed for reproducibility.
        """
        self.nn = nn
        self.convolutional_layer = convolutional_layer
        self.X = None
        self.Y = None

    def forward(self,):
        """
        Perform forward pass through the Convolutional Neural Network.
        """
        self.convolutional_layer.inp = self.X
        self.convolutional_layer.forward()

        self.nn.X = self.convolutional_layer.activation_layer.aZ
        self.nn.Y = self.Y

        self.nn.forward()

    def backward(self,):
        self.nn.backward()

        self.convolutional_layer.backward( self.nn.grad_nn )
```

```
def StocasticGradiantDescentCNN(X_train, y_train, X_test, y_test, cnn, i
    """
    Perform Stochastic Gradient Descent for training a Convolutional Neu
    - X_train (numpy.ndarray): Training features.
    - y_train (numpy.ndarray): Training target variable.
    - X_test (numpy.ndarray): Test features.
    - y_test (numpy.ndarray): Test target variable.
    - cnn (ConvolutionalNeuralNetwork): Convolutional Neural Network ins
    - inShape (tuple): Shape of the input data.
    - outShape (tuple): Shape of the output data.
    - epochs (int): Number of training epochs.
    - problem (str): Type of problem, 'classification' or 'regression'.
    """

    for _ in range(epochs):
        randomIndx = np.random.randint(len(X_train))

        # Extract a random sample from the training data
        X_sample = X_train[randomIndx, :].reshape(inShape)
        Y_sample = y_train[randomIndx, :].reshape(outShape)

        # Set the input and target variables for the CNN
        cnn.X = X_sample
        cnn.Y = Y_sample

        # Perform forward and backward passes through the CNN
        cnn.forward()
        cnn.backward()

    # Evaluate training accuracy
    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("Training Data Accuracy :" + str(acc) + "/" + str(len(y_true)
```

```python
# Evaluate testing accuracy
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("Testing Data Accuracy :" + str(acc) + "/" + str(len(y_true))
```

TASK : Train CNN on MNIST Dataset

A Convolutional Neural Network (CNN) is employed for image classification. It comprises a convolutional layer with 16 output channels and tanh activation, followed by a fully connected layer with linear activation. Softmax cross-entropy loss is used for multi-class classification.

```python
# Load the MNIST dataset with one-hot encoding for target variable
X_train, y_train, X_test, y_test = load_data('mnist', one_hot_encode_y=True)

# Initialize the Convolutional Layer
convolutional_layer = ConvolutionalLayer((1,8,8), filter_shape=(3,3), Co=16, activation='tanh', lr=0.01)

# Extract the output shape of the convolutional layer for the Neural Network
nn_inp_shape = convolutional_layer.flatten_shape

# Specify the architecture of the Neural Network layers after the convolutional layer
nn_inp_shape, nn_out_shape, layers = makeLayers(nn_inp_shape, [10], ['softmax'])

# Create the Convolutional Neural Network with the initialized convolutional layer and neural network layers
cnn = ConvolutionalNeuralNetwork(convolutional_layer, NeuralNetwork(layers, 'cross_entropy', lr=0.01))

# Specify the output shape for the CNN
out_shape = (1, layers_sizes[-1])

# Perform Stochastic Gradient Descent training for the Convolutional Neural Network
StocasticGradiantDescentCNN(X_train,y_train,X_test,y_test, cnn,(1,8,8), out_shape,epochs=5000)
```

```
Training Data Accuracy :1322/1437 = 91.9972164231037 %
Testing Data Accuracy :331/360 = 91.94444444444444 %
```

<u>THANK YOU</u>