WEEK - 2

List of Exercises:

1. **Implement and demonstrate Multi-Layer Perceptron (MLP) model with back propagation to solve the XOR Boolean function.**
2. **Implementation of MLP for the XOR problem in Keras**
3. **MLP model using Keras with Iris dataset. Validating the model on the test data and then plotting the learning curve.**
4. **MLP model using Keras with Iris dataset. Validating the model on the test data and then plotting the accuracy and loss.**

****

EXERCISE - 1

XOR Boolean function

Python Program with Explanation:

1. Import numpy, array-processing package to work with the arrays.

```
import numpy as np
```

2. Define a function abs() that returns the absolute value of x.

```
def abs(x):
      return x if x>0 else -x
```

3. Define sigmoid(x) to implement a logistic sigmoid activation function.

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

4. Define sigmoid_derivative(x) used in computing error in back propagation phase.

```
def sigmoid_derivative(x):
    return x * (1 - x)
```

5. Define the error function/cost function which checks if there is error between the expected output and predicted output and returns a Boolean value.

```
def checkError(predicted_output):
      expected_output = [[0],[1],[1],[0]]
      for i,j in  zip(expected_output , predicted_output):
            if abs(i[0]-j[0]) > 0.001:
                  return True
      return False
```

6. Set input data and desired output of an XOR Boolean function.

```
#Input datasets
inputs = np.array([[0,0],[0,1],[1,0],[1,1]])
expected_output = np.array([[0],[1],[1],[0]])
```

7. Initialialize epoch to 0 and learning rate to 0.1.

```
epoch = 0
lr = 0.1
```

8. Define the MLP network which consists of 2 neurons in Input layer, 2 neurons in Hidden layer and 1 neuron in the Output layer. Accept the number of neurons in each layer.

```
# inputLayerNeurons = 2
# hiddenLayerNeurons = 2
# outputLayerNeurons = 1
inputLayerNeurons = int(input("enter no of inputLayer"))
hiddenLayerNeurons = int(input("enter no of hiddenlayer "))
outputLayerNeurons = int(input("enter no of outputlayer "))
```

9. Define a list called hidden_weights to store the weights of the hidden layer neurons in the
network.

```
hidden_weights = []
```

10. Receive the weights of the link from the input layer neurons to hidden layer neurons.

```
for i in range(1,inputLayerNeurons+1):
      hidden_weights_ind = []
      for j in
range(inputLayerNeurons+1,inputLayerNeurons+hiddenLayerNeurons+1):

hidden_weights_ind.append(float(input('w'+str(i)+str(j))))
      hidden_weights.append(hidden_weights_ind)
```

11. Define list called output_weights to store the weights of the output layer neurons in the network.

```
output_weights = []
```

12. Receive the weights of the link from the Hidden layer neurons to Output layer neurons.

```
for i in
range(inputLayerNeurons+1,inputLayerNeurons+hiddenLayerNeurons+1):
      output_weights_ind = []
      for j in
range(inputLayerNeurons+hiddenLayerNeurons+1,inputLayerNeurons+hiddenLayerN
eurons+outputLayerNeurons+1):

output_weights_ind.append(float(input('w'+str(i)+str(j))))
      output_weights.append(output_weights_ind)
```

13. Define lists called hidden_bias and output_bias to store the bias of the Hidden layer
neurons and Output layer neurons in the network.

```
hidden_bias = []
output_bias = []
```

14. Receive the Hidden layer biases and Output layer biases.

```
for i in
range(inputLayerNeurons+1,inputLayerNeurons+hiddenLayerNeurons+outputLayerN
eurons+1):
      if i > inputLayerNeurons+hiddenLayerNeurons:
            output_bias.append(float(input("o"+str(i))))
      else:
            hidden_bias.append(float(input("o"+str(i))))
```

15. Convert the weight lists and bias lists to array.

```
hidden_weights = np.asarray(hidden_weights)
hidden_bias = np.asarray([hidden_bias])
output_weights = np.asarray(output_weights)
output_bias = np.asarray([output_bias])
```

16. Print the initial hidden weights, hidden biases, output weights and output biases.

```
print("Initial hidden weights: ",end='')
print(*hidden_weights)
print("Initial hidden biases: ",end='')
print(*hidden_bias)
print("Initial output weights: ",end='')
print(*output_weights)
```

```
      print("Initial output biases: ",end='')
      print(*output_bias)
```

17. Initialize the predicted_output.

```
      predicted_output = [[0],[0],[0],[0]]
```

18. Train MLP until the predicted output converges to the desired output.

```
      while checkError(predicted_output):
            epoch += 1
```

Step 1: Forward Propagation.

Calculate Net Input and Output in the Hidden Layer and Output Layer.

```
            hidden_layer_activation = np.dot(inputs,hidden_weights)
            hidden_layer_activation += hidden_bias
            hidden_layer_output = sigmoid(hidden_layer_activation)
            output_layer_activation                                    =
np.dot(hidden_layer_output,output_weights)
            output_layer_activation += output_bias
            predicted_output = sigmoid(output_layer_activation)
```

Estimate error at the node in the Output Layer.

```
            error = expected_output - predicted_output
```

Step 2: Backward Propagation

Calculate Error at each node in the Output layer and Hidden layer.

```
            d_predicted_output = error *
sigmoid_derivative(predicted_output)
            error_hidden_layer = d_predicted_output.dot(output_weights.T)
            d_hidden_layer = error_hidden_layer *
      sigmoid_derivative(hidden_layer_output)
```

Update all Weights and Biases.

```
      output_weights += hidden_layer_output.T.dot(d_predicted_output) * lr
      output_bias += np.sum(d_predicted_output,axis=0,keepdims=True) * lr
            hidden_weights += inputs.T.dot(d_hidden_layer) * lr
            hidden_bias += np.sum(d_hidden_layer,axis=0,keepdims=True) * lr
```

19. Print the final learned weights and biases of the Hidden layer and Output layer.

```
      print("Final hidden weights: ",end='')
      print(*hidden_weights)
      print("Final hidden bias: ",end='')
      print(*hidden_bias)
      print("Final output weights: ",end='')
      print(*output_weights)
      print("Final output bias: ",end='')
      print(*output_bias)
```

20. Print the final output obtained for the input data set (i.e., for the XOR function)

```
      print("\nOutput from neural network: ",end='')
      print(*predicted_output)
```

21. Print the number of epochs executed to learn the weights and biases for the model to get the desired output.

```
      print("\nNo of epochs")
```

```
      print(epoch)
```

Complete Program:

```
import numpy as np

def abs(x):
      return x if x>0 else -x

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

def sigmoid_derivative(x):
    return x * (1 - x)

def checkError(predicted_output):
      expected_output = [[0],[1],[1],[0]]
      for i,j in  zip(expected_output , predicted_output):
            if abs(i[0]-j[0]) > 0.001:
                  return True
      return False

#Input datasets
inputs = np.array([[0,0],[0,1],[1,0],[1,1]])
expected_output = np.array([[0],[1],[1],[0]])

epoch = 0
lr = 0.1

# inputLayerNeurons = 2
# hiddenLayerNeurons = 2
# outputLayerNeurons = 1
inputLayerNeurons = int(input("enter no of inputLayer"))
hiddenLayerNeurons = int(input("enter no of hiddenlayer "))
outputLayerNeurons = int(input("enter no of outputlayer "))

hidden_weights = []
for i in range(1,inputLayerNeurons+1):
      hidden_weights_ind = []
      for                                  j                                  in
range(inputLayerNeurons+1,inputLayerNeurons+hiddenLayerNeurons+1):
            hidden_weights_ind.append(float(input('w'+str(i)+str(j))))
      hidden_weights.append(hidden_weights_ind)

output_weights = []
for i in range(inputLayerNeurons+1,inputLayerNeurons+hiddenLayerNeurons+1):
      output_weights_ind = []
      for j in
range(inputLayerNeurons+hiddenLayerNeurons+1,inputLayerNeurons+hiddenLayerN
eurons+outputLayerNeurons+1):
            output_weights_ind.append(float(input('w'+str(i)+str(j))))
      output_weights.append(output_weights_ind)

hidden_bias = []
output_bias = []
for i in
range(inputLayerNeurons+1,inputLayerNeurons+hiddenLayerNeurons+outputLayerN
eurons+1):
      if i > inputLayerNeurons+hiddenLayerNeurons:
            output_bias.append(float(input("o"+str(i))))
```

```
      else:
            hidden_bias.append(float(input("o"+str(i))))

hidden_weights = np.asarray(hidden_weights)
hidden_bias = np.asarray([hidden_bias])
output_weights = np.asarray(output_weights)
output_bias = np.asarray([output_bias])

print("Initial hidden weights: ",end='')
print(*hidden_weights)
print("Initial hidden biases: ",end='')
print(*hidden_bias)
print("Initial output weights: ",end='')
print(*output_weights)
print("Initial output biases: ",end='')
print(*output_bias)

predicted_output = [[0],[0],[0],[0]]

#Training algorithm

while checkError(predicted_output):
      epoch += 1
      #Forward Propagation
      hidden_layer_activation = np.dot(inputs,hidden_weights)
      hidden_layer_activation += hidden_bias
      hidden_layer_output = sigmoid(hidden_layer_activation)

      output_layer_activation = np.dot(hidden_layer_output,output_weights)
      output_layer_activation += output_bias
      predicted_output = sigmoid(output_layer_activation)

      #Backpropagation
      error = expected_output - predicted_output
      d_predicted_output = error * sigmoid_derivative(predicted_output)

      error_hidden_layer = d_predicted_output.dot(output_weights.T)
      d_hidden_layer           =           error_hidden_layer           *
sigmoid_derivative(hidden_layer_output)

      #Updating Weights and Biases
      output_weights += hidden_layer_output.T.dot(d_predicted_output) * lr
      output_bias += np.sum(d_predicted_output,axis=0,keepdims=True) * lr
      hidden_weights += inputs.T.dot(d_hidden_layer) * lr
      hidden_bias += np.sum(d_hidden_layer,axis=0,keepdims=True) * lr

print("Final hidden weights: ",end='')
print(*hidden_weights)
print("Final hidden bias: ",end='')
print(*hidden_bias)
print("Final output weights: ",end='')
print(*output_weights)
print("Final output bias: ",end='')
print(*output_bias)

print("\nOutput from neural network: ",end='')
print(*predicted_output)
print("\nNo of epochs")
print(epoch)
```

Output:

```
= RESTART: multilayer_perceptron_xor.py
enter no of inputLayer2
enter no of hiddenlayer 2
enter no of outputlayer 1
w133
w146
w234
w245
w352
w454
o31
o4-6
o5-3.93
Initial hidden weights: [3. 6.] [4. 5.]
Initial hidden biases: [ 1. -6.]
Initial output weights: [2.] [4.]
Initial output biases: [-3.93]
Final hidden weights: [ 6.12370882 10.03281141] [ 6.12342151 10.0272853 ]
Final hidden bias: [-9.34571401 -4.50662615]
Final output weights: [-15.62277004] [14.81103743]
Final output bias: [-7.06705554]

Output from neural network: [0.001] [0.99916398] [0.99916411] [0.00085368]

No of epochs
14905082
>>>
```

Screenshot of the Output:

```
while checkError(predicted_output):

        epoch += 1

        #Forward Propagation
        hidden_layer_activation = np.dot(inputs,hidden_weights)
        hidden_layer_activation += hidden_bias
        hidden_layer_output = sigmoid(hidden_layer_activation)

        output_layer_activation = np.dot(hidden_layer_output,output_weights)
        output_layer_activation += output_bias
        predicted_output = sigmoid(output_layer_activation)

        #Backpropagation
        error = expected_output - predicted_output
        d_predicted_output = error * sigmoid_derivative(predicted_output)

        error_hidden_layer = d_predicted_output.dot(output_weights.T)
        d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_ou

        #Updating Weights and Biases
        output_weights += hidden_layer_output.T.dot(d_predicted_output) * lr
        output_bias += np.sum(d_predicted_output,axis=0,keepdims=True) * lr
        hidden_weights += inputs.T.dot(d_hidden_layer) * lr
        hidden_bias += np.sum(d_hidden_layer,axis=0,keepdims=True) * lr

print("Final hidden weights: ",end='')
print(*hidden_weights)
print("Final hidden bias: ",end='')
print(*hidden_bias)
print("Final output weights: ",end='')
print(*output_weights)
print("Final output bias: ",end='')
print(*output_bias)

print("\nOutput from neural network: ",end='')
print(*predicted_output)
print("\nNo of epochs")
print(epoch)
```

```
enter no of inputLayer2
enter no of hiddenlayer 2
enter no of outputlayer 1
w133
w146
w234
w245
w352
w454
o31
o4-6
o5-3.93
Initial hidden weights: [3. 6.] [4. 5.]
Initial hidden biases: [ 1. -6.]
Initial output weights: [2.] [4.]
Initial output biases: [-3.93]
Final hidden weights: [ 6.12370882 10.03281141] [ 6.12342151 10.0272853 ]
Final hidden bias: [-9.34571401 -4.50662615]
Final output weights: [-15.62277004] [14.81103743]
Final output bias: [-7.06705554]

Output from neural network: [0.001] [0.99916398] [0.99916411] [0.00085368]

No of epochs
14905082
>>>
```

****

EXERCISE - 2

MLP for the XOR problem in Keras

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# 1. Define the XOR inputs and expected outputs
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], "float32") # Inputs
```

```
y = np.array([[0], [1], [1], [0]], "float32") # Expected outputs

# 2. Design the neural network model
model = Sequential()
# Add a hidden layer with 4 neurons and 'tanh' activation function (or
'relu')
model.add(Dense(4, input_dim=2, activation='tanh'))
# Add the output layer with 1 neuron and 'sigmoid' activation for binary
classification
model.add(Dense(1, activation='sigmoid'))

# 3. Compile the model
# Use binary cross-entropy loss for binary classification and the 'adam'
optimizer
model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])

# 4. Train the model
# Train for a sufficient number of epochs (e.g., 1000 or more)
model.fit(X, y, epochs=1000, verbose=0) # verbose=0 suppresses training
output

# 5. Evaluate the model
_, accuracy = model.evaluate(X, y, verbose=0)
print(f'Accuracy: {accuracy * 100:.2f}%\n')

# 6. Make predictions
predictions = model.predict(X)
print("Predictions:")
for i in range(len(X)):
    # Round predictions to 0 or 1 for clarity
    print(f"Input: {X[i]} | Predicted Output: {predictions[i][0]:.4f} |
Rounded: {round(predictions[i][0])} | Expected: {y[i][0]}")
```

****

EXERCISE - 3

MLP model using Keras with Iris dataset. Validating the model on the test data and then plotting the learning curve.

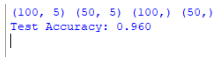
Program Code:

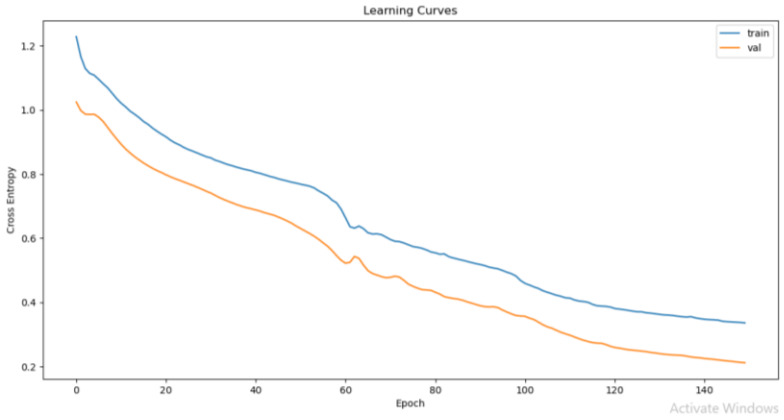
```
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD
from matplotlib import pyplot

#  Read the dataset 'Iris.csv'
df = read_csv('Iris.csv')

#  Split the Iris features into input and output columns
X = df.values[:, :-1]
y = df.values[:, -1]

# Check all data are floating point values
X = X.astype('float32')

# Encode the strings of labels to integer values
```

```
y = LabelEncoder().fit_transform(y)

# Split the data matrix into train and test dataset and Print the shape of
train dataset and test dataset.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
shuffle = True, random_state = 123)
print("X_train shape: {}".format(X_train.shape))
print("X_test shape: {}".format(X_test.shape))
print("y_train shape: {}".format(y_train.shape))
print("y_test shape: {}".format(y_test.shape))
# split randomly the data matrix into training dataset and validation
dataset
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
test_size=0.2, random_state=1)

# Determine the number of input features
n_features = X_train.shape[1]

# Define the model
model = Sequential()
model.add(Dense(10, activation='relu', kernel_initializer='he_normal',
input_shape=(n_features,)))
model.add(Dense(8, activation='relu', kernel_initializer='he_normal'))
model.add(Dense(3, activation='softmax'))

# Compile the model
sgd = SGD(learning_rate=0.001, momentum=0.8)
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

# Fit the model
history = model.fit(X_train, y_train, epochs=150, batch_size=32, verbose=0,
validation_split=0.3)

# Evaluate the model and print the accuracy
loss, acc = model.evaluate(X_test, y_test, verbose=0)
print('Test Accuracy: %.3f' % acc)

# Visualize by plotting the learning curves
pyplot.title('Learning Curves')
pyplot.xlabel('Epoch')
pyplot.ylabel('Cross Entropy')
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='val')
pyplot.legend()
pyplot.show()
```

Output:

```
(100, 5) (50, 5) (100,) (50,)
Test Accuracy: 0.960
>>>
```





****

EXERCISE - 4

MLP model using Keras with Iris dataset. Validating the model on the test data and then plotting the accuracy and loss.

Program Code:

```
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from matplotlib import pyplot as plt
#  Read the dataset 'Iris.csv'
df = read_csv('Iris.csv')
#  Split the Iris features into input and output columns
X = df.values[:, :-1]
y = df.values[:, -1]

# Check all data are floating point values
X = X.astype('float32')
# Encode the strings of labels to integer values
y = LabelEncoder().fit_transform(y)
# Split the data matrix into train and test dataset and Print the shape of
train dataset and test dataset.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
shuffle = True, random_state = 123)
print("X_train shape: {}".format(X_train.shape))
print("X_test shape: {}".format(X_test.shape))
print("y_train shape: {}".format(y_train.shape))
print("y_test shape: {}".format(y_test.shape))
# split randomly the data matrix into training dataset and validation
dataset
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
test_size=0.2, random_state=1)

# Determine the number of input features
n_features = X_train.shape[1]

# Define model
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(n_features,)),
    keras.layers.Dense(4, activation=tf.nn.relu),
    keras.layers.Dense(4, activation=tf.nn.relu),
```

```
    keras.layers.Dense(1, activation=tf.nn.sigmoid),
])
# Compile the model
model.compile(optimizer='adam', loss='mse', metrics=['accuracy'])
# Fit the model
history = model.fit(X_train, y_train, epochs=34, batch_size=32, verbose=0,
validation_data=(X_val, y_val))

# Evaluate the model and print the accuracy
loss, acc = model.evaluate(X_test, y_test, verbose=0)
print('Test Accuracy: %.3f' % acc)

# Visualize 'Training vs. Validation loss'
loss_train = history.history['loss']
loss_val = history.history['val_loss']
epochs = range(1,35)
plt.plot(epochs, loss_train, 'g', label='Training loss')
plt.plot(epochs, loss_val, 'b', label='validation loss')
plt.title('Training vs. Validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Visualize 'Training  Accuracy vs. Validation accuracy'
loss_train = history.history['accuracy']
loss_val = history.history['val_accuracy']
epochs = range(1,35)
plt.plot(epochs, loss_train, 'g', label='Training accuracy')
plt.plot(epochs, loss_val, 'b', label='validation accuracy')
plt.title('Training vs. Validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

Output:

```
X_train shape: (120, 5)
X_test shape: (30, 5)
y_train shape: (120,)
y_test shape: (30,)
Test Accuracy: 0.200
```

****

Programming Assignment:

Let us now look at an example of learning in a Multi –Layer Perceptron. The given MLP consists of an Input layer, one Hidden layer and an Output layer. Input layer has 3neurons, Hidden layer has 2 neurons and a single neuron in the Output layer.

X_1	X_2	X_3	$O_{Desired}$
1	1	0	1

Learning rate: =0.6

The weights and biases are tabulated in Table 17.1.

Table 17.1: Weights and Biases

X_1	X_2	X_3	W_{14}	W_{15}	W_{24}	W_{25}	W_{34}	W_{35}	W_{46}	W_{56}	$\theta 4$	$\theta 5$	$\theta 6$
1	1	0	0.2	0.3	-0.1	0.2	0.3	-0.4	-0.3	0.1	0.1	0.3	-0.3