DL lab programs – 6th sem CSE(AIML)

1. Not gate using Perceptron and without learning

2. AND/OR gate using Perceptron and without learning

3. XOR gate using multi layer perceptron and without learning

4. XOR gate using Keras

5. Full adder using Keras

6. Classification of MNIST data using dense layer and Keras

7. Implementation of autoencoder using Keras for MNIST dataset

8. Linear regression for Boston housing dataset without cross validation using Keras

9. Linear regression for Boston housing dataset with cross validation using Keras

10. Classification of MNIST data using CNN and Keras

**1. Not gate using Perceptron and without learning**

# importing Python library

import numpy as np

# define Unit Step Function

def unitStep(v):

    if v >= 0:

        return 1

    else:

        return 0

# design Perceptron Model

def perceptronModel(x, w, b):

    v = np.dot(w, x) + b

    y = unitStep(v)

    return y

# NOT Logic Function

# w = -1, b = 0.5

def NOT\_logicFunction(x):

    w = -1

    b = 0.5

    return perceptronModel(x, w, b)

# testing the Perceptron Model

test1 = np.array(1)

test2 = np.array(0)

print("NOT({}) = {}".format(1, NOT\_logicFunction(test1))) print("NOT({}) = {}".format(0, NOT\_logicFunction(test2)))

**2a. A simplest perceptron has a single-layer network whose weights and biases can be trained to produce a correct target vector when presented with the corresponding input vector. The training technique used is called the perceptron learning rule. Develop a python program to create a simple perceptron from scratch to simulate working of OR gate without the concept of learning.**

# importing Python library

import numpy as np

# define Unit Step Function

def unitStep(v):

    if v >= 0:

        return 1

    else:

        return 0

# design Perceptron Model

def perceptronModel(x, w, b):

    v = np.dot(w, x) + b

    y = unitStep(v)

    return y

# OR Logic Function

# w1 = 1, w2 = 1, b = -0.5

def OR\_logicFunction(x):

    w = np.array([1, 1])

    b = -0.5

    return perceptronModel(x, w, b)

# testing the Perceptron Model

test1 = np.array([0, 1])

test2 = np.array([1, 1])

test3 = np.array([0, 0])

test4 = np.array([1, 0])

print("OR({}, {}) = {}".format(0, 1, OR\_logicFunction(test1)))

print("OR({}, {}) = {}".format(1, 1, OR\_logicFunction(test2)))

print("OR({}, {}) = {}".format(0, 0, OR\_logicFunction(test3)))

print("OR({}, {}) = {}".format(1, 0, OR\_logicFunction(test4)))

**2b. A simplest perceptron has a single-layer network whose weights and biases can be trained to produce a correct target vector when presented with the corresponding input vector. The training technique used is called the perceptron learning rule. Develop a python program to create a simple perceptron from scratch to simulate working of AND gate without the concept of learning.**

# importing Python library

import numpy as np

# define Unit Step Function

def unitStep(v):

    if v >= 0:

        return 1

    else:

        return 0

# design Perceptron Model

def perceptronModel(x, w, b):

    v = np.dot(w, x) + b

    y = unitStep(v)

    return y

# AND Logic Function

# w1 = 1, w2 = 1, b = -1.5

def AND\_logicFunction(x):

    w = np.array([1, 1])

    b = -1.5

    return perceptronModel(x, w, b)

# testing the Perceptron Model

test1 = np.array([0, 1])

test2 = np.array([1, 1])

test3 = np.array([0, 0])

test4 = np.array([1, 0])

print("AND({}, {}) = {}".format(0, 1, AND\_logicFunction(test1)))

print("AND({}, {}) = {}".format(1, 1, AND\_logicFunction(test2)))

print("AND({}, {}) = {}".format(0, 0, AND\_logicFunction(test3)))

print("AND({}, {}) = {}".format(1, 0, AND\_logicFunction(test4

**3. A simplest perceptron has a single-layer network whose weights and biases can be trained to produce a correct target vector when presented with the corresponding input vector. The training technique used is called the perceptron learning rule. Develop a python program to create a simple perceptron from scratch to simulate working of XOR gate without the concept of learning.**

**We can observe that, XOR(x1,x2)=AND(NOT(AND(x1,x2)),OR(x1,x2))**

# importing Python library

import numpy as np

# define Unit Step Function

def unitStep(v):

    if v >= 0:

        return 1

    else:

        return 0

# design Perceptron Model

def perceptronModel(x, w, b):

    v = np.dot(w, x) + b

    y = unitStep(v)

    return y

# NOT Logic Function

# wNOT = -1, bNOT = 0.5

def NOT\_logicFunction(x):

    wNOT = -1

    bNOT = 0.5

    return perceptronModel(x, wNOT, bNOT)

# AND Logic Function

# here w1 = wAND1 = 1,

# w2 = wAND2 = 1, bAND = -1.5

def AND\_logicFunction(x):

    w = np.array([1, 1])

    bAND = -1.5

    return perceptronModel(x, w, bAND)

# OR Logic Function

# w1 = 1, w2 = 1, bOR = -0.5

def OR\_logicFunction(x):

    w = np.array([1, 1])

    bOR = -0.5

    return perceptronModel(x, w, bOR)

# XOR Logic Function

# with AND, OR and NOT

# function calls in sequence

def XOR\_logicFunction(x):

    y1 = AND\_logicFunction(x)

    y2 = OR\_logicFunction(x)

    y3 = NOT\_logicFunction(y1)

    final\_x = np.array([y2, y3])

    finalOutput = AND\_logicFunction(final\_x)

    return finalOutput

# testing the Perceptron Model

test1 = np.array([0, 1])

test2 = np.array([1, 1])

test3 = np.array([0, 0])

test4 = np.array([1, 0])

print("XOR({}, {}) = {}".format(0, 1, XOR\_logicFunction(test1)))

print("XOR({}, {}) = {}".format(1, 1, XOR\_logicFunction(test2)))

print("XOR({}, {}) = {}".format(0, 0, XOR\_logicFunction(test3)))

print("XOR({}, {}) = {}".format(1, 0, XOR\_logicFunction(test4)))

**4) XOR using Keras**

import numpy as np

from keras.models import Sequential

from keras.layers.core import Dense

training\_data = np.array([[0,0],[0,1],[1,0],[1,1]], "float32")

target\_data = np.array([[0],[1],[1],[0]], "float32")

model = Sequential()

model.add(Dense(16, input\_dim=2, activation='relu'))

model.add(Dense(1, activation='sigmoid'))

model.summary()

model.compile(loss='mean\_squared\_error', optimizer='adam', metrics=['binary\_accuracy'])

model.fit(training\_data, target\_data, epochs=1000)

scores = model.evaluate(training\_data, target\_data)

print("\n%s: %.2f%%" % (model.metrics\_names[1], scores[1]\*100))

print (model.predict(training\_data).round())

**5) Full Adder using Keras**

import numpy as np

from keras.models import Sequential

from keras.layers.core import Dense

training\_data = np.array([[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]], "float32")

target\_data = np.array([[0,0],[0,1],[0,1],[1,0],[0,1],[1,0],[1,0],[1,1]], "float32")

model = Sequential()

model.add(Dense(16, input\_dim=2, activation='relu'))

model.add(Dense(1, activation='sigmoid'))

model.summary()

model.compile(loss='mean\_squared\_error',optimizer='adam',metrics=['binary\_accuracy'])

model.fit(training\_data, target\_data, epochs=1000)

scores = model.evaluate(training\_data, target\_data)

print("\n%s: %.2f%%" % (model.metrics\_names[1], scores[1]\*100))

print (model.predict(training\_data).round())

### 6. Neural network for classification of MNIST data using sequential dense model

Import tensorflow

From tensorflow import keras

from keras.datasets import mnist

(train\_images, train\_labels),(test\_images, test\_labels) = mnist.load\_data()

from keras import models

from keras import layers

network = models.Sequential()

network.add(layers.Dense(512, activation='relu', input\_shape=(28 \* 28,)))

network.add(layers.Dense(10, activation='softmax'))

network.compile(optimizer='rmsprop', loss='categorical\_crossentropy', metrics=['accuracy'])

#Before training, we will preprocess our data by reshaping it into the shape that the network expects, and scaling it so that all values are in the [0, 1] interval. Previously, our training images for instance were stored in an array of shape (60000, 28, 28) of type uint8 with values in the [0, 255] interval. We transform it into a float32 array of shape (60000, 28 \* 28) with values between 0 and 1.

train\_images = train\_images.reshape((60000, 28 \* 28))

train\_images = train\_images.astype('float32') / 255

test\_images = test\_images.reshape((10000, 28 \* 28))

test\_images = test\_images.astype('float32') / 255

#We also need to categorically encode the labels, a step which we explain in chapter 3:

from keras.utils import to\_categorical

train\_labels = to\_categorical(train\_labels)

test\_labels = to\_categorical(test\_labels)

#We are now ready to train our network, which in Keras is done via a call to the fit method of the network: we "fit" the model to its training data.

network.fit(train\_images, train\_labels, epochs=5, batch\_size=128)

#Two quantities are being displayed during training: the "loss" of the network over the training data, and the accuracy of the network over the training data.

#We quickly reach an accuracy of 0.989 (i.e. 98.9%) on the training data. Now let's check that our model performs well on the test set too:

test\_loss, test\_acc = network.evaluate(test\_images, test\_labels)

print('test\_acc:', test\_acc)

#test\_acc: 0.9777

#Our test set accuracy turns out to be 97.8%

**7) Autoencoders using keras for MNIST data**

import numpy as np

from tensorflow import keras

from keras.models import Sequential

from keras import datasets

from keras.layers.core import Dense

(x\_train, y\_train), (x\_test, y\_test) = keras.datasets.mnist.load\_data()

# Scale images to the [0, 1] range

x\_train = x\_train.astype("float32") / 255

x\_test = x\_test.astype("float32") / 255

print("x\_train shape:", x\_train.shape)

print(x\_train.shape[0], "train samples")

print(x\_test.shape[0], "test samples")

x\_train1=x\_train.reshape(60000, 784)

x\_test1=x\_test.reshape(10000,784)

model = Sequential()

model.add(Dense(392, input\_dim=784, activation='relu'))

model.add(Dense(784, activation='relu'))

model.compile(loss='mean\_squared\_error',

                       optimizer='adam',

                       metrics=['accuracy'])

model.fit(x\_train1,x\_train1, epochs=5)

scores = model.evaluate(x\_test1,x\_test1)

decoded\_imgs=model.predict(x\_test1)

decoded\_imgs1=decoded\_imgs.reshape(10000,28,28)

decoded\_imgs.shape[1]

import matplotlib.pyplot as plt

n = 10 # how many images we will display

plt.figure(figsize=(20, 4))

for i in range(n):

    # display original

    ax = plt.subplot(2, n, i + 1)

    plt.imshow(x\_test1[i].reshape(28, 28))

    plt.gray()

    ax.get\_xaxis().set\_visible(False)

    ax.get\_yaxis().set\_visible(False)

    # display reconstruction

    ax = plt.subplot(2, n, i + 1 + n)

    plt.imshow(decoded\_imgs1[i].reshape(28, 28))

    plt.gray()

    ax.get\_xaxis().set\_visible(False)

    ax.get\_yaxis().set\_visible(False)

plt.show()

### 8. Linear regression without cross validation for predicting house price (The Boston Housing Price dataset)

import keras

from keras.datasets import boston\_housing

(train\_data, train\_targets), (test\_data, test\_targets) =  boston\_housing.load\_data()

#Preprocessing of data

mean = train\_data.mean(axis=0)

train\_data -= mean

std = train\_data.std(axis=0)

train\_data /= std

test\_data -= mean

test\_data /= std

#Building our network

from keras import models

from keras import layers

def build\_model():

    # we use a function to construct it.

    model = models.Sequential()

    model.add(layers.Dense(64, activation='relu', input\_shape=(train\_data.shape[1],)))

    model.add(layers.Dense(64, activation='relu'))

    model.add(layers.Dense(1))

    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

    return model

'''Our network ends with a single unit, and no activation (i.e. it will be linear layer). This is a typical setup for scalar regression (i.e. regression where we are trying to predict a single continuous value).

Note that we are compiling the network with the mse loss function -  Mean Squared Error, the square of the difference between the predictions and the targets, a widely used loss function for regression problems.

We are also monitoring a new metric during training: mae. This stands for Mean Absolute Error. It is simply the absolute value of the difference between the predictions and the targets.  '''

import numpy as np

    # Build the Keras model (already compiled)

model = build\_model()

    # Train the model (in silent mode, verbose=0)

model.fit(train\_data,  train\_targets, epochs=num\_epochs, batch\_size=1, verbose=0)

    # Evaluate the model on the validation data

val\_mse, val\_mae = model.evaluate(val\_data, val\_targets, verbose=0)

print(val\_mae)

**9. Linear Regression on Boston dataset with cross validation**

import keras

from keras.datasets import boston\_housing

(train\_data, train\_targets), (test\_data, test\_targets) =  boston\_housing.load\_data()

#Preprocessing of data

mean = train\_data.mean(axis=0)

train\_data -= mean

std = train\_data.std(axis=0)

train\_data /= std

test\_data -= mean

test\_data /= std

from keras import models

from keras import layers

def build\_model():

 # Because we will need to instantiate the same model multiple times, #we use a function to construct it.

    model = models.Sequential()

    model.add(layers.Dense(64, activation='relu', input\_shape=(train\_data.shape[1],)))

    model.add(layers.Dense(64, activation='relu'))

    model.add(layers.Dense(1))

    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

    return model

Validating our approach using K-fold validation

To evaluate our network while we keep adjusting its parameters (such as the number of epochs used for training), we could simply split the data into a training set and a validation set, as we were doing in our previous examples. However, because we have so few data points, the validation set would end up being very small (e.g. about 100 examples). A consequence is that our validation scores may change a lot depending on which data points we choose to use for validation and which we choose for training, i.e. the validation scores may have a high variance with regard to the validation split. This would prevent us from reliably evaluating our model.

The best practice in such situations is to use K-fold cross-validation. It consists of splitting the available data into K partitions (typically K=4 or 5), then instantiating K identical models, and training each one on K- partitions while evaluating on the remaining partition. The validation score for the model used would then be the average of the K validation scores obtained. In terms of code, this is straightforward:'''

import numpy as np

k = 4

num\_val\_samples = len(train\_data) // k

num\_epochs = 100

all\_scores = []

for i in range(k):

    print('processing fold #', i)

    # Prepare the validation data: data from partition # k

    val\_data = train\_data[i \* num\_val\_samples: (i + 1) \* num\_val\_samples]

    val\_targets = train\_targets[i \* num\_val\_samples: (i + 1) \* num\_val\_samples]

    # Prepare the training data: data from all other partitions

    partial\_train\_data = np.concatenate([train\_data[:i \* num\_val\_samples],  train\_data[(i + 1) \* num\_val\_samples:]],  axis=0)

    partial\_train\_targets = np.concatenate( [train\_targets[:i \* num\_val\_samples],train\_targets[(i + 1) \* num\_val\_samples:]],  axis=0)

    # Build the Keras model (already compiled)

    model = build\_model()

    # Train the model (in silent mode, verbose=0)

    model.fit(partial\_train\_data, partial\_train\_targets,epochs=num\_epochs, batch\_size=1, verbose=0)

    # Evaluate the model on the validation data

    val\_mse, val\_mae = model.evaluate(val\_data, val\_targets, verbose=0)

    all\_scores.append(val\_mae)

print(all\_scores)

np.mean(all\_scores)

#Let's try training the network for a bit longer: 500 epochs. To keep a record of how well the model did at each epoch, we will modify our training loop to save the per-epoch validation score log:

from keras import backend as K

# Some memory clean-up

K.clear\_session()

num\_epochs = 500

all\_mae\_histories = []

for i in range(k):

    print('processing fold #', i)

    # Prepare the validation data: data from partition # k

    val\_data = train\_data[i \* num\_val\_samples: (i + 1) \* num\_val\_samples]

    val\_targets = train\_targets[i \* num\_val\_samples: (i + 1) \* num\_val\_samples]

    # Prepare the training data: data from all other partitions

    partial\_train\_data = np.concatenate( [train\_data[:i \* num\_val\_samples],

         train\_data[(i + 1) \* num\_val\_samples:]], axis=0)

    partial\_train\_targets = np.concatenate( [train\_targets[:i \* num\_val\_samples],

         train\_targets[(i + 1) \* num\_val\_samples:]], axis=0)

    # Build the Keras model (already compiled)

    model = build\_model()

    # Train the model (in silent mode, verbose=0)

    history = model.fit(partial\_train\_data, partial\_train\_targets,validation\_data=(val\_data, val\_targets), epochs=num\_epochs, batch\_size=1, verbose=0)

    mae\_history = history.history['val\_mean\_absolute\_error']

    all\_mae\_histories.append(mae\_history)

#We can then compute the average of the per-epoch MAE scores for all folds:

average\_mae\_history = [ np.mean([x[i] for x in all\_mae\_histories]) for i in range(num\_epochs)]

#Let's plot this:

import matplotlib.pyplot as plt

plt.plot(range(1, len(average\_mae\_history) + 1), average\_mae\_history)

plt.xlabel('Epochs')

plt.ylabel('Validation MAE')

plt.show()

#It may be a bit hard to see the plot due to scaling issues and relatively high variance. Let's:

#Omit the first 10 data points, which are on a different scale from the rest of the curve.

#Replace each point with an exponential moving average of the previous points, to obtain a smooth curve.

def smooth\_curve(points, factor=0.9):

  smoothed\_points = []

  for point in points:

    if smoothed\_points:

      previous = smoothed\_points[-1]

      smoothed\_points.append(previous \* factor + point \* (1 - factor))

    else:

      smoothed\_points.append(point)

  return smoothed\_points

smooth\_mae\_history = smooth\_curve(average\_mae\_history[10:])

plt.plot(range(1, len(smooth\_mae\_history) + 1), smooth\_mae\_history)

plt.xlabel('Epochs')

plt.ylabel('Validation MAE')

plt.show()

### 10. Neural network for classification of MNIST data using sequential CNN model

import keras

from keras.datasets import mnist

(train\_images, train\_labels),(test\_images, test\_labels) = mnist.load\_data()

from keras import models

from keras import layers

network = models.Sequential()

keras.Input(shape=input\_shape),

layers.Conv2D(32, kernel\_size=(3, 3), activation="relu"),

layers.MaxPooling2D(pool\_size=(2, 2)),

layers.Conv2D(64, kernel\_size=(3, 3), activation="relu"),

layers.MaxPooling2D(pool\_size=(2, 2)),

layers.Flatten(),

layers.Dropout(0.5),

layers.Dense(num\_classes, activation="softmax"),

network.add(layers.Conv2D (32,kernel\_size=(3,3), activation='relu', input\_shape=(28 ,28,1)))

network.add(layers.MaxPooling2D(pool\_size=(2, 2)))

network.add(layers.Conv2D(64, kernel\_size=(3, 3), activation="relu"))

network.add(layers.MaxPooling2D(pool\_size=(2, 2)))

network.add(layers.Flatten())

network.add(layers.Dropout(0.5))

network.add(layers.Dense(10, activation="softmax"))

network.compile(optimizer='rmsprop', loss='categorical\_crossentropy', metrics=['accuracy'])

train\_images = train\_images.reshape((60000, 28 \* 28))

train\_images = train\_images.astype('float32') / 255

test\_images = test\_images.reshape((10000, 28 \* 28))

test\_images = test\_images.astype('float32') / 255

#We also need to categorically encode the labels, a step which we explain in chapter 3:

from keras.utils import to\_categorical

train\_labels = to\_categorical(train\_labels)

test\_labels = to\_categorical(test\_labels)

#We are now ready to train our network, which in Keras is done via a call to the fit method of the network: we "fit" the model to its training data.

network.fit(train\_images, train\_labels, epochs=5, batch\_size=128)

#Two quantities are being displayed during training: the "loss" of the network over the training data, and the accuracy of the network over the training data.

test\_loss, test\_acc = network.evaluate(test\_images, test\_labels)

#9536/10000 [===========================>..] - ETA: 0s

print('test\_acc:', test\_acc)