

INDEX

Sr. No.	List of Practical	Date	Page No	Sign
1.	Implement the following		1	
A.	Performing matrix multiplication and finding eigenvectors and eigenvalues using TensorFlow	13/03/24	2	
2.	Implement the following		3	
A.	Solving XOR problem using deep feed forward network.	13/03/24	4	
3.	Implement the following		6	
A.	Implementing a deep neural network for performing binary classification tasks.	20/03/24	7	
4.	Implement the following		9	
A.	Using deep feed forward network with two hidden layers for performing classification and predicting the class	20/03/24	10	
B.	Using deep feed forward network with two hidden layers for performing classification and predicting the probability of class.	27/03/24	12	
C.	Using deep feed forward network with two hidden layers for performing linear regression and predicting values.	27/03/24	14	
5.	Implement the following		15	
A.	Evaluating feed forward deep network for regression using KFold cross validation.	03/04/24	16	
B.	Evaluating feed forward deep network for multiclass classification using KFold cross-validation.	03/04/24	17	
6.	Implement the following		20	
A.	Implementing regularization to avoid overfitting in binary classification.	10/04/24	21	
7.	Implement the following		26	
A.	Demonstrate recurrent neural network that learns to perform sequence analysis for stock price.	10/04/24	27	
8.	Implement the following		31	
A.	Performing encoding and decoding of images using deep autoencoders.	24/04/24	32	
9.	Implement the following		35	
A.	Implementation of convolutional neural network to predict numbers from number images.	24/04/24	36	
10.	Implement the following		38	
A.	Denoising of images using autoencoder.	01/05/24	39	

Practical 1

Theory:

Performing matrix multiplication, as well as finding eigenvectors and eigenvalues, are fundamental operations in linear algebra and play essential roles in various fields, including machine learning, computer graphics, and physics. TensorFlow, a popular open-source machine learning library developed by Google, provides efficient and scalable implementations of these operations, making it a powerful tool for numerical computation and deep learning.

Matrix multiplication involves multiplying two matrices to produce a third matrix, where each element in the resulting matrix is computed as the sum of the products of corresponding elements from the input matrices. TensorFlow provides the `tf.matmul()` function for matrix multiplication, which efficiently handles large matrices and can be used in a wide range of applications, such as linear regression, neural networks, and convolutional networks.

Eigenvalues and eigenvectors are key concepts in linear algebra that arise when studying the properties of square matrices. An eigenvector of a square matrix is a nonzero vector that, when multiplied by the matrix, results in a scaled version of itself. The scaling factor is called the eigenvalue corresponding to that eigenvector. TensorFlow offers the `tf.linalg.eig()` function to compute the eigenvalues and eigenvectors of a matrix efficiently. These eigenvalues and eigenvectors are widely used in various applications, including principal component analysis (PCA), dimensionality reduction, and solving differential equations.

In summary, TensorFlow provides powerful and efficient implementations of matrix multiplication, eigenvalue decomposition, and eigenvector computation, enabling researchers and practitioners to perform complex numerical computations and solve a wide range of problems in machine learning and scientific computing. These operations form the foundation of many advanced algorithms and techniques, making them essential components of TensorFlow's capabilities for building and training machine learning models.

A. Performing matrix multiplication and finding eigenvectors and eigenvalues using TensorFlow.

Code:

```
import tensorflow as tf
print("Matrix Multiplication")
x=tf.constant([1,2,3,4,5,6],shape=[2,3])
print(x)
y=tf.constant([7,8,9,10,11,12],shape=[3,2])
print(y)
z=tf.matmul(x,y)
print("Product:",z)
e_matrix_A=tf.random.uniform([2,2],minval=3,maxval=10,dtype=tf.float32,name="matrixA")
print("Matrix A:\n{}\n\n".format(e_matrix_A))
eigen_values_A,eigen_vectors_A=tf.linalg.eigh(e_matrix_A)
print("Eigen Vectors of Matrix A: \n{}\n\nEigen Values of Matrix A: \n{}\n\n".format(eigen_vectors_A,eigen_values_A))
```

Output:

```
Matrix Multiplication
tf.Tensor(
[[ 1  2  3]
 [ 4  5  6]], shape=(2, 3), dtype=int32)
tf.Tensor(
[[ 7  8]
 [ 9 10]
 [11 12]], shape=(3, 2), dtype=int32)
Product: tf.Tensor(
[[ 58  64]
 [139 154]], shape=(2, 2), dtype=int32)
Matrix A:
[[4.6240635 5.635919 ]
 [6.6197047 3.0183482]]

Eigen Vectors of Matrix A:
[[-0.6631739  0.74846536]
 [ 0.74846536  0.6631739 ]]

Eigen Values of Matrix A:
[-2.8470078 10.48942 ]
```

Practical 2

Theory:

The XOR problem is a classic problem in the field of artificial neural networks and serves as a benchmark for testing the capabilities of different network architectures. The XOR (exclusive OR) operation takes two binary inputs and returns 1 if the inputs are different and 0 if they are the same. While this operation is relatively simple to express with a truth table, it poses a challenge for traditional linear classifiers, such as logistic regression, due to its non-linear nature.

A deep feedforward network, also known as a multilayer perceptron (MLP), is a type of artificial neural network where neurons are arranged in layers, with connections only between adjacent layers. Each neuron in a layer receives inputs from the previous layer, applies an activation function to the weighted sum of its inputs, and passes the result to the next layer. In a deep feedforward network, there are typically one or more hidden layers between the input and output layers.

To solve the XOR problem using a deep feedforward network, we need to construct a network architecture that can capture the non-linear relationship between the input and output. One common approach is to use multiple layers with nonlinear activation functions, such as sigmoid, tanh, or ReLU (Rectified Linear Unit). These activation functions introduce non-linearity into the network, allowing it to approximate complex functions.

The network is trained using a supervised learning algorithm, such as gradient descent or its variants, to minimize the error between the predicted output and the true output. During training, the network adjusts its weights and biases iteratively based on the gradient of a loss function, which measures the difference between the predicted and true outputs. Backpropagation, an algorithm for efficiently computing gradients in deep networks, is typically used to update the parameters of the network.

By iteratively adjusting the parameters of the network using a training dataset that includes XOR examples, the network learns to approximate the XOR function. With a sufficiently large and well-constructed network architecture, along with appropriate training techniques, a deep feedforward network can successfully solve the XOR problem and learn to represent complex non-linear relationships between inputs and outputs.

A. Solving XOR problem using deep feed forward network.

Code:

```
import numpy as np
from keras.layers import Dense
from keras.models import Sequential
model=Sequential()
model.add(Dense(units=2,activation='relu',input_dim=2))
model.add(Dense(units=1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
print(model.summary())
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 2)	6
dense_1 (Dense)	(None, 1)	3

Total params: 9 (36.00 B)

Trainable params: 9 (36.00 B)

Non-trainable params: 0 (0.00 B)

None

```
print(model.get_weights())
X=np.array([[0.,0.],[0.,1.],[1.,0.],[1.,1.]])
Y=np.array([0.,1., 1.,0.])
model.fit(X,Y,epochs=1000,batch_size=4)
```

```
[array([[ 0.9962076 , -0.16683471],
        [-0.85846674,  1.0450855 ]], dtype=float32), array([0., 0.], dtype=float32), array([[ -0.7846273],
        [ 1.3534516]], dtype=float32), array([0.], dtype=float32)]
Epoch 1/1000
1/1 ————— 1s 636ms/step - accuracy: 0.5000 - loss: 0.8605
Epoch 2/1000
1/1 ————— 0s 37ms/step - accuracy: 0.5000 - loss: 0.8593
Epoch 3/1000
1/1 ————— 0s 38ms/step - accuracy: 0.5000 - loss: 0.8580
Epoch 4/1000
1/1 ————— 0s 37ms/step - accuracy: 0.5000 - loss: 0.8568
Epoch 5/1000
1/1 ————— 0s 44ms/step - accuracy: 0.5000 - loss: 0.8556
Epoch 6/1000
1/1 ————— 0s 36ms/step - accuracy: 0.5000 - loss: 0.8544
Epoch 7/1000
1/1 ————— 0s 36ms/step - accuracy: 0.5000 - loss: 0.8532
```

```

Epoch 993/1000
1/1 _____ 0s 34ms/step - accuracy: 0.7500 - loss: 0.5519
Epoch 994/1000
1/1 _____ 0s 32ms/step - accuracy: 0.7500 - loss: 0.5518
Epoch 995/1000
1/1 _____ 0s 35ms/step - accuracy: 0.7500 - loss: 0.5517
Epoch 996/1000
1/1 _____ 0s 39ms/step - accuracy: 0.7500 - loss: 0.5516
Epoch 997/1000
1/1 _____ 0s 20ms/step - accuracy: 0.7500 - loss: 0.5515
Epoch 998/1000
1/1 _____ 0s 27ms/step - accuracy: 0.7500 - loss: 0.5513
Epoch 999/1000
1/1 _____ 0s 27ms/step - accuracy: 0.7500 - loss: 0.5512
Epoch 1000/1000
1/1 _____ 0s 28ms/step - accuracy: 0.7500 - loss: 0.5511

```

```
[3]: <keras.src.callbacks.history.History at 0x19edf98bb10>
```

```

print("Model Weights: ",model.get_weights())
print("Model Prediction: ",model.predict(X,batch_size=4))

```

```

Model Weights: [array([[ 0.10622789, -0.84694576],
                        [-0.33309647,  0.94577515]], dtype=float32), array(
                        [ 1.9854522 ]], dtype=float32), array([-0.26225564]

```

```
1/1 _____ 0s 23ms/step
```

```

Model Prediction: [[0.41015866]
                   [0.80074966]
                   [0.4044863 ]
                   [0.42235163]]

```

Practical 3

Theory

A deep neural network (DNN) for binary classification tasks is a type of artificial neural network architecture designed to classify input data into one of two categories (binary classes). It consists of multiple layers of neurons, including an input layer, one or more hidden layers, and an output layer. Each neuron in the network applies a nonlinear activation function to the weighted sum of its inputs, allowing the network to capture complex relationships in the data.

Here's a step-by-step overview of implementing a DNN for binary classification tasks:

1. **Input Layer:** Neurons in the input layer represent input features, with the number of neurons equalling the dimensionality of the input data.
2. **Hidden Layers:** Hidden layers perform the bulk of computation, transforming input data into nonlinear representations. The number of hidden layers and neurons per layer are determined based on data complexity.
3. **Output Layer:** The output layer contains one neuron for binary classification, using a sigmoid activation function to output probabilities of class membership.
4. **Loss Function:** Binary cross-entropy loss measures the difference between predicted probabilities and true labels, penalizing misclassifications.
5. **Optimization:** Parameters (weights and biases) are optimized using optimization algorithms like stochastic gradient descent (SGD), updating them iteratively to minimize the loss function.
6. **Training:** The DNN is trained on labelled data, processing batches iteratively and updating parameters through backpropagation until convergence or a set number of epochs.
7. **Evaluation:** Model performance is evaluated on a separate dataset using metrics like accuracy, precision, recall, F1-score, and ROC-AUC.

By following these steps and appropriately configuring the network architecture, loss function, optimization algorithm, and evaluation metrics, you can implement a deep neural network for performing binary classification tasks effectively.

A. Implementing a deep neural network for performing binary classification tasks.

Code:

```
from numpy import loadtxt
from keras.models import Sequential
from keras.layers import Dense

dataset=pd.read_csv('/content/diabetes.csv',delimiter=',')
dataset
```



	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1
...
763	10	101	76	48	180	32.9	0.171	63	0
764	2	122	70	27	0	36.8	0.340	27	0
765	5	121	72	23	112	26.2	0.245	30	0
766	1	126	60	0	0	30.1	0.349	47	1
767	1	93	70	31	0	30.4	0.315	23	0

768 rows × 9 columns

```
X=dataset.iloc[:,0:8]
y=dataset.iloc[:,8]
X
y
```



	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
0	6	148	72	35	0	33.6	0.627	50
1	1	85	66	29	0	26.6	0.351	31
2	8	183	64	0	0	23.3	0.672	32
3	1	89	66	23	94	28.1	0.167	21
4	0	137	40	35	168	43.1	2.288	33
...
763	10	101	76	48	180	32.9	0.171	63
764	2	122	70	27	0	36.8	0.340	27
765	5	121	72	23	112	26.2	0.245	30
766	1	126	60	0	0	30.1	0.349	47
767	1	93	70	31	0	30.4	0.315	23

768 rows × 8 columns


```

0      1
1      0
2      1
3      0
4      1
..
763    0
764    0
765    0
766    1
767    0
Name: Outcome, Length: 768, dtype: int64

```

```

model=Sequential()
model.add(Dense(12,input_dim=8,activation='relu'))
model.add(Dense(8,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam',
,metrics=['accuracy'])
model.fit(X,y,epochs=150,batch_size=10)

```

```

Epoch 1/150
77/77 [=====] - 1s 2ms/step - loss: 4.3493 - accuracy: 0.5469
Epoch 2/150
77/77 [=====] - 0s 2ms/step - loss: 1.4138 - accuracy: 0.5872
Epoch 3/150
77/77 [=====] - 0s 2ms/step - loss: 1.2194 - accuracy: 0.6094
Epoch 4/150
77/77 [=====] - 0s 2ms/step - loss: 1.1071 - accuracy: 0.6016
Epoch 5/150
77/77 [=====] - 0s 2ms/step - loss: 1.0177 - accuracy: 0.6341
Epoch 6/150
77/77 [=====] - 0s 2ms/step - loss: 0.9700 - accuracy: 0.6341
Epoch 7/150

```

```

_,accuracy=model.evaluate(X,y)
print('Accuracy of model is',(accuracy*100))

```

```

24/24 [=====] - 0s 1ms/step - loss: 0.4501 - accuracy: 0.7943
Accuracy of model is 79.42708134651184

```

```

#prediction=model.predict_classes(X)
prediction=model.predict(X)

```

```

24/24 [=====] - 0s 1ms/step

```

Practical 4

Theory

A deep feedforward network with two hidden layers is a type of artificial neural network architecture designed to learn and represent complex relationships in data for various tasks such as classification, regression, and feature extraction. Let's break down its components and how it works:

1. **Input Layer:** The input layer consists of neurons representing the features of the input data. Each neuron corresponds to one feature, and the number of neurons in the input layer depends on the dimensionality of the input data.
2. **Hidden Layers:** The network contains two hidden layers between the input and output layers. These hidden layers enable the network to learn hierarchical representations of the input data. Each neuron in a hidden layer computes a weighted sum of its inputs, applies an activation function to the sum, and passes the result to the neurons in the next layer. The number of neurons in each hidden layer is a hyperparameter that can be adjusted based on the complexity of the data and the problem at hand.
3. **Activation Functions:** Nonlinear activation functions are applied to the output of each neuron in the hidden layers to introduce nonlinearity into the network and enable it to learn complex relationships in the data. Common activation functions include sigmoid, tanh, and Rectified Linear Unit (ReLU). ReLU is a popular choice due to its simplicity and efficiency in training deep neural networks.
4. **Output Layer:** The output layer produces the final predictions or representations of the input data. For classification tasks, the output layer typically consists of neurons corresponding to the number of classes in the problem. Each neuron's output represents the predicted probability of belonging to a particular class, and the softmax function is often applied to normalize these probabilities, ensuring they sum up to one.
5. **Training:** The network is trained using labelled data, where each input is associated with a corresponding target output. During training, the network's parameters (weights and biases) are adjusted iteratively using optimization algorithms such as stochastic gradient descent (SGD) or its variants. The objective is to minimize a loss function that measures the difference between the predicted outputs and the true targets.
6. **Backpropagation:** Backpropagation is the key algorithm used to compute the gradients of the loss function with respect to the network's parameters. These gradients are then used to update the parameters in the direction that minimizes the loss, effectively optimizing the network for the task at hand.
7. **Evaluation:** After training, the performance of the network is evaluated on a separate validation or test dataset to assess its generalization ability. Common evaluation metrics for classification tasks include accuracy, precision, recall, F1-score, and ROC-AUC.

Overall, a deep feedforward network with two hidden layers is a flexible and powerful architecture capable of learning and representing complex relationships in data, making it suitable for a wide range of machine learning tasks.

A. Using deep feed forward network with two hidden layers for performing classification and predicting the class.

Code:

```
from tensorflow import keras # Using tensorflow.keras instead
of keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.datasets import make_blobs
from sklearn.preprocessing import MinMaxScaler
import numpy as np
X, Y = make_blobs(n_samples=100, centers=2, n_features=2,
random_state=1)
scalar = MinMaxScaler()
scalar.fit(X)
X = scalar.transform(X)
model = Sequential()
model.add(Dense(4, input_dim=2, activation='relu'))
model.add(Dense(4, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam')
model.fit(X, Y, epochs=500)
Xnew, Yreal = make_blobs(n_samples=3, centers=2, n_features=2,
random_state=1)
Xnew = scalar.transform(Xnew)
# Use predict followed by argmax for class prediction
Ynew = np.argmax(model.predict(Xnew), axis=-1) # Import numpy
as np
for i in range(len(Xnew)):
    print("X=%s, Predicted=%s, Desired=%s" % (Xnew[i],
Ynew[i], Yreal[i]))
```

Output:

```
4/4 ————— 0s 2ms/step - loss: 0.6943
Epoch 3/500
4/4 ————— 0s 2ms/step - loss: 0.6945
Epoch 4/500
4/4 ————— 0s 2ms/step - loss: 0.6939
Epoch 5/500
4/4 ————— 0s 3ms/step - loss: 0.6936
Epoch 6/500
4/4 ————— 0s 2ms/step - loss: 0.6933
Epoch 7/500
4/4 ————— 0s 5ms/step - loss: 0.6922
Epoch 8/500
4/4 ————— 0s 2ms/step - loss: 0.6916
Epoch 9/500
4/4 ————— 0s 6ms/step - loss: 0.6905
Epoch 10/500
4/4 ————— 0s 2ms/step - loss: 0.6891
```

```

Epoch 482/500
4/4 ----- 0s 2ms/step - loss: 0.1094
Epoch 483/500
4/4 ----- 0s 2ms/step - loss: 0.1014
Epoch 484/500
4/4 ----- 0s 3ms/step - loss: 0.1120
Epoch 485/500
4/4 ----- 0s 2ms/step - loss: 0.0989
Epoch 486/500
4/4 ----- 0s 2ms/step - loss: 0.1081
Epoch 487/500
4/4 ----- 0s 2ms/step - loss: 0.1078
Epoch 488/500
4/4 ----- 0s 2ms/step - loss: 0.0940
Epoch 489/500
4/4 ----- 0s 2ms/step - loss: 0.1056
Epoch 490/500
4/4 ----- 0s 2ms/step - loss: 0.0977
Epoch 491/500
4/4 ----- 0s 2ms/step - loss: 0.0986
Epoch 492/500
4/4 ----- 0s 2ms/step - loss: 0.0964
Epoch 493/500
4/4 ----- 0s 3ms/step - loss: 0.1030
Epoch 494/500
4/4 ----- 0s 3ms/step - loss: 0.1015
Epoch 495/500
4/4 ----- 0s 3ms/step - loss: 0.0991
Epoch 496/500
4/4 ----- 0s 2ms/step - loss: 0.1011
Epoch 497/500
4/4 ----- 0s 3ms/step - loss: 0.1127
Epoch 498/500
4/4 ----- 0s 3ms/step - loss: 0.1093
Epoch 499/500
4/4 ----- 0s 2ms/step - loss: 0.1004
Epoch 500/500
4/4 ----- 0s 3ms/step - loss: 0.0947
1/1 ----- 0s 78ms/step
X=[0.89337759 0.65864154], Predicted=0, Desired=0
X=[0.29097707 0.12978982], Predicted=0, Desired=1
X=[0.78082614 0.75391697], Predicted=0, Desired=0

```

B. Using deep feed forward network with two hidden layers for performing classification and predicting the probability of class.

Code:

```
from keras.models import Sequential
from keras.layers import Dense
from sklearn.datasets import make_blobs
from sklearn.preprocessing import MinMaxScaler
import numpy as np
X, Y = make_blobs(n_samples=100, centers=2, n_features=2,
random_state=1)
scalar = MinMaxScaler()
scalar.fit(X)
X = scalar.transform(X)
model = Sequential()
model.add(Dense(4, input_dim=2, activation='relu'))
model.add(Dense(4, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam')
model.fit(X, Y, epochs=500)
Xnew, Yreal = make_blobs(n_samples=3, centers=2, n_features=2,
random_state=1)
Xnew = scalar.transform(Xnew) # Corrected variable assignment
here
Ynew_probabilities = model.predict(Xnew)
Ynew_classes = np.round(Ynew_probabilities).astype(int) #
Convert probabilities to classes
for i in range(len(Xnew)):
    print("X=%s, Predicted_probability=%s, Predicted_class=%s"
% (Xnew[i], Ynew_probabilities[i], Ynew_classes[i]))
```

Output:

```
Epoch 3/500
4/4 ————— 0s 0s/step - loss: 0.6515
Epoch 4/500
4/4 ————— 0s 2ms/step - loss: 0.6596
Epoch 5/500
4/4 ————— 0s 2ms/step - loss: 0.6504
Epoch 6/500
4/4 ————— 0s 2ms/step - loss: 0.6584
Epoch 7/500
4/4 ————— 0s 1ms/step - loss: 0.6516
Epoch 8/500
4/4 ————— 0s 0s/step - loss: 0.6492
Epoch 9/500
4/4 ————— 0s 664us/step - loss: 0.649:
```

```

4/4 ————— 0s 3ms/step - loss: 0.0071
Epoch 482/500
4/4 ————— 0s 1ms/step - loss: 0.0063
Epoch 483/500
4/4 ————— 0s 2ms/step - loss: 0.0064
Epoch 484/500
4/4 ————— 0s 2ms/step - loss: 0.0065
Epoch 485/500
4/4 ————— 0s 2ms/step - loss: 0.0065
Epoch 486/500
4/4 ————— 0s 2ms/step - loss: 0.0061
Epoch 487/500
4/4 ————— 0s 3ms/step - loss: 0.0059
Epoch 488/500
4/4 ————— 0s 2ms/step - loss: 0.0063
Epoch 489/500
4/4 ————— 0s 2ms/step - loss: 0.0059
Epoch 490/500
4/4 ————— 0s 2ms/step - loss: 0.0067
Epoch 491/500
4/4 ————— 0s 2ms/step - loss: 0.0064
Epoch 492/500
4/4 ————— 0s 3ms/step - loss: 0.0063
Epoch 493/500
4/4 ————— 0s 4ms/step - loss: 0.0061
Epoch 494/500
4/4 ————— 0s 2ms/step - loss: 0.0061
Epoch 495/500
4/4 ————— 0s 2ms/step - loss: 0.0060
Epoch 496/500
4/4 ————— 0s 2ms/step - loss: 0.0059
Epoch 497/500
4/4 ————— 0s 3ms/step - loss: 0.0058
Epoch 498/500
4/4 ————— 0s 2ms/step - loss: 0.0063
Epoch 499/500
4/4 ————— 0s 3ms/step - loss: 0.0063
Epoch 500/500
4/4 ————— 0s 1ms/step - loss: 0.0056
1/1 ————— 0s 73ms/step
X=[0.89337759 0.65864154], Predicted_probability=[0.01816906], Predicted_class=[0]
X=[0.29097707 0.12978982], Predicted_probability=[0.99586505], Predicted_class=[1]
X=[0.78082614 0.75391697], Predicted_probability=[0.00811532], Predicted_class=[0]

```

C. Using deep feed forward network with two hidden layers for performing linear regression and predicting values.

Code:

```
from keras.models import Sequential
from keras.layers import Dense
from sklearn.datasets import make_regression
from sklearn.preprocessing import MinMaxScaler
X,Y=make_regression(n_samples=100,n_features=2,noise=0.1,random_state=1)
scalarX, scalarY=MinMaxScaler(), MinMaxScaler()
scalarX.fit(X)
scalarY.fit(Y.reshape(100,1))
X=scalarX.transform(X)
Y=scalarY.transform(Y.reshape(100,1))
model=Sequential()
model.add(Dense(4,input_dim=2,activation='relu'))
model.add(Dense(4,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='mse', optimizer='adam')
model.fit(X, Y, epochs=1000, verbose=0)
Xnew,a=make_regression(n_samples=3,n_features = 2,noise=0.1,random_state=1)
Xnew=scalarX.transform(Xnew)
Ynew = model.predict(Xnew)
for i in range(len(Xnew)):
    print("X=%s, Predicted=%s"%(Xnew[i], Ynew[i]))
```

Output:

```
1/1 ————— 0s 40ms/step
X=[0.29466096 0.30317302], Predicted=[0.18638343]
X=[0.39445118 0.79390858], Predicted=[0.7539049]
X=[0.02884127 0.6208843 ], Predicted=[0.3971907]
```

Practical 5

Theory

Evaluating a feedforward deep network for regression using K-Fold cross-validation involves splitting the dataset into K subsets (folds), training the model on K-1 folds, and evaluating its performance on the remaining fold. This process is repeated K times, with each fold serving as the validation set once. Here's the theory behind it:

1. **Dataset Splitting:** The dataset is divided into K approximately equal-sized subsets (folds). Each fold contains a portion of the data that will be used for training and validation.
2. **Model Training and Validation:** For each iteration of K-Fold cross-validation, one fold is held out as the validation set, while the remaining K-1 folds are used for training the model. The model is trained on the training data and then evaluated on the validation data to assess its performance.
3. **Performance Metric:** For regression tasks, common performance metrics include Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and R-squared (R²) score. These metrics measure the discrepancy between the predicted and true target values.
4. **Cross-Validation Loop:** The K-Fold cross-validation process is repeated K times, with each fold serving as the validation set exactly once. This ensures that every data point is used for both training and validation, leading to a more robust estimate of the model's performance.
5. **Aggregate Performance:** After completing the K-Fold cross-validation process, the performance metrics obtained from each fold are averaged to obtain a single estimate of the model's performance. This aggregated performance metric provides a more reliable evaluation of the model's generalization ability.
6. **Model Selection:** K-Fold cross-validation can be used to compare different models or hyperparameter settings. By evaluating each model using the same cross-validation procedure, it is possible to identify the model that performs best on average across multiple folds.
7. **Bias-Variance Tradeoff:** K-Fold cross-validation helps assess the bias and variance of the model. A high average performance metric across folds indicates low bias and variance, while a large variability in performance metrics suggests high variance.

Overall, K-Fold cross-validation is a widely used technique for evaluating the performance of regression models, including feedforward deep networks. It provides a robust estimate of the model's generalization ability and helps identify potential issues such as overfitting or underfitting.

A. Evaluating feed forward deep network for regression using KFold cross validation.

Code:

```
!pip install scikeras
from pandas import read_csv
from keras.models import Sequential
from keras.layers import Dense
from scikeras.wrappers import KerasRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# load dataset
dataframe = read_csv("housing.csv", delim_whitespace=True,
header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:13]
Y = dataset[:,13]
# define wider model
def wider_model():
    # create model
    model = Sequential()
    model.add(Dense(20, input_shape=(13,),
kernel_initializer='normal', activation='relu'))
    model.add(Dense(1, kernel_initializer='normal'))
    # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model
# evaluate model with standardized dataset
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(model=wider_model,
epochs=100, batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = KFold(n_splits=10)
results = cross_val_score(pipeline, X, Y, cv=kfold,
scoring='neg_mean_squared_error')
print("Wider: %.2f (%.2f) MSE" % (results.mean(),
results.std()))
```

Output:

```
/usr/local/lib/python3.10/dist-packages/k
super().__init__(activity_regularizer=a
/usr/local/lib/python3.10/dist-packages/k
super().__init__(activity_regularizer=a
Wider: -22.93 (25.62) MSE
```

B. Evaluating feed forward deep network for multiclass classification using KFold cross-validation.**Code:**

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_iris
from sklearn.model_selection import KFold
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.metrics import accuracy_score, confusion_matrix
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Load Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

#One-hot encode target Labels
label_encoder=LabelEncoder()
integer_encoded = label_encoder.fit_transform(y)
onehot_encoder = OneHotEncoder(sparse=False)
integer_encoded =
integer_encoded.reshape(len(integer_encoded), 1)
y_encoded = onehot_encoder.fit_transform(integer_encoded)

#Define K-fold cross-validation
k_folds = 5
kf = KFold(n_splits=k_folds, shuffle=True, random_state=42)
# Define neural network architecture
def create_model():
    model = Sequential()
    model.add(Dense(10, input_dim=4, activation= 'relu'))
    model.add(Dense(8, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    model.compile(loss = 'categorical_crossentropy',
optimizer='adam', metrics=['accuracy'])
    return model

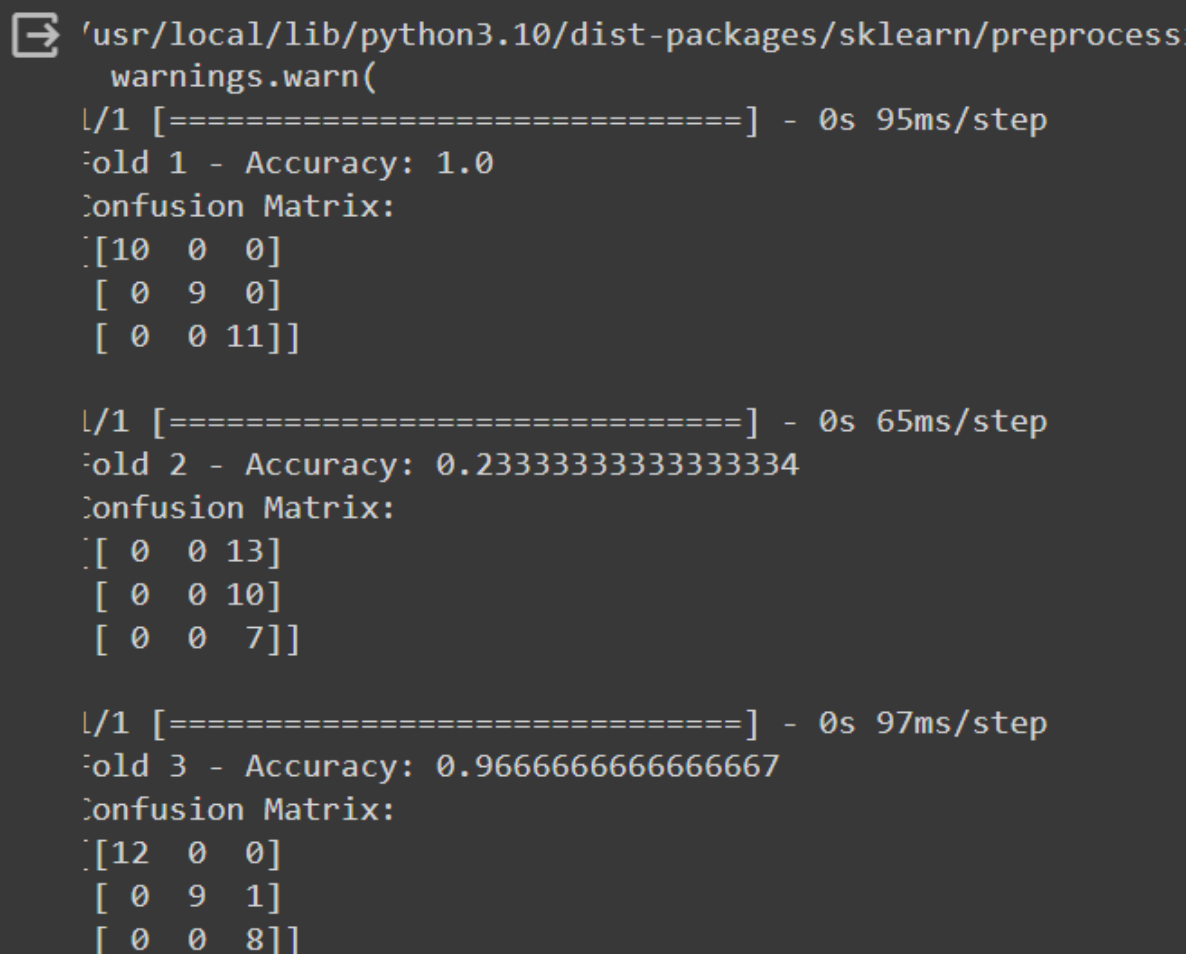
# Perform K-fold cross-validation
fold = 0
accuracies = []
conf_matrices = []
for train_index, test_index in kf.split(X):
    fold += 1
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y_encoded[train_index],
y_encoded[test_index]
    model = create_model()
    model.fit(X_train, y_train, epochs=100, batch_size=5,
verbose=0)
    y_pred = model.predict(X_test)
    y_pred_classes = np.argmax(y_pred, axis=1)
```

```

y_test_classes = np.argmax(y_test, axis=1)
accuracy = accuracy_score(y_test_classes, y_pred_classes)
accuracies.append(accuracy)
conf_matrix = confusion_matrix(y_test_classes,
y_pred_classes)
conf_matrices.append(conf_matrix)
print(f"Fold {fold} - Accuracy: {accuracy}")
print("Confusion Matrix: ")
print(conf_matrix)
print()
# Calculate and print average accuracy
avg_accuracy = np.mean(accuracies)
print(f'Average Accuracy: {avg_accuracy}')
#Visualize the distribution of accuracies across folds
plt.figure(figsize=(8, 6))
sns.boxplot(y=accuracies)
plt.title('Distribution of Accuracies Across Folds')
plt.xlabel('Accuracy')
plt.show()

```

Output:



```

/usr/local/lib/python3.10/dist-packages/sklearn/preprocess
warnings.warn(
l/1 [=====] - 0s 95ms/step
Fold 1 - Accuracy: 1.0
Confusion Matrix:
[[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]

l/1 [=====] - 0s 65ms/step
Fold 2 - Accuracy: 0.23333333333333334
Confusion Matrix:
[[ 0  0 13]
 [ 0  0 10]
 [ 0  0  7]]

l/1 [=====] - 0s 97ms/step
Fold 3 - Accuracy: 0.9666666666666667
Confusion Matrix:
[[12  0  0]
 [ 0  9  1]
 [ 0  0  8]]

```

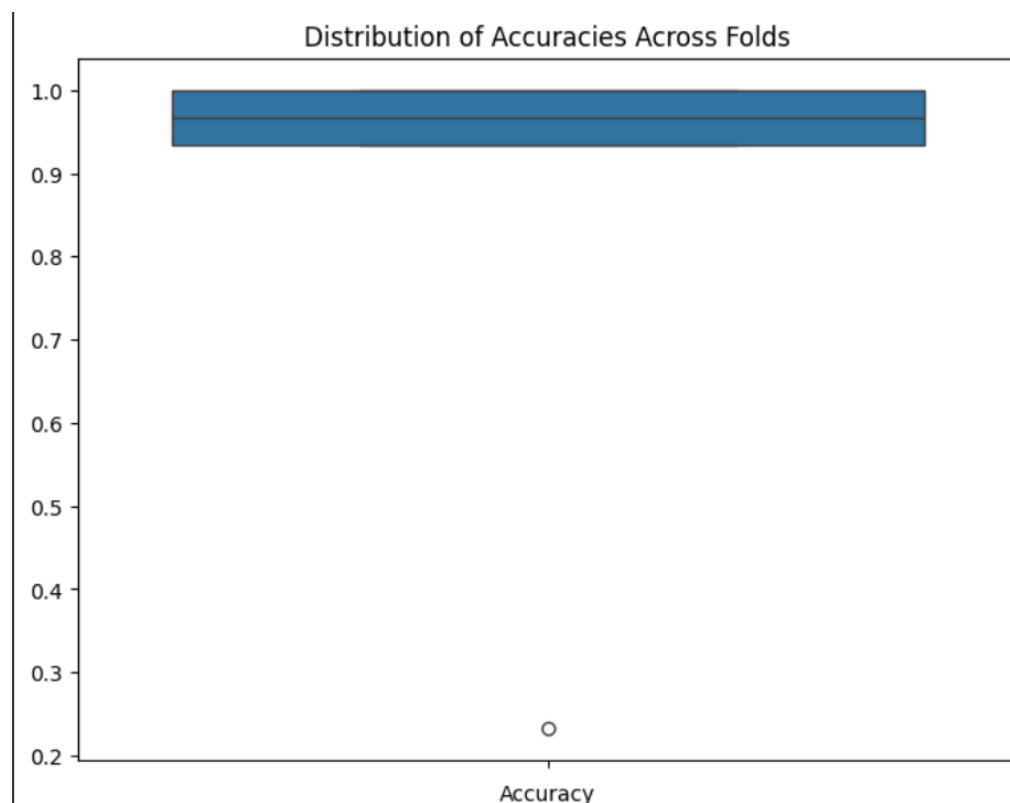
```
1/1 [=====] - 0s 65ms/step
Fold 2 - Accuracy: 0.23333333333333334
Confusion Matrix:
[[ 0  0 13]
 [ 0  0 10]
 [ 0  0  7]]

1/1 [=====] - 0s 97ms/step
Fold 3 - Accuracy: 0.9666666666666667
Confusion Matrix:
[[12  0  0]
 [ 0  9  1]
 [ 0  0  8]]

1/1 [=====] - 0s 62ms/step
Fold 4 - Accuracy: 0.9333333333333333
Confusion Matrix:
[[ 8  0  0]
 [ 0  8  2]
 [ 0  0 12]]

WARNING:tensorflow:5 out of the last 5 calls to <function Mode
1/1 [=====] - 0s 68ms/step
Fold 5 - Accuracy: 1.0
Confusion Matrix:
[[ 7  0  0]
 [ 0 11  0]
 [ 0  0 12]]

Average Accuracy: 0.8266666666666668
```



Practical 6

Theory

Regularization is a technique used to prevent overfitting in machine learning models, including those used for binary classification. Overfitting occurs when a model learns to fit the training data too closely, capturing noise and irrelevant patterns that do not generalize well to unseen data. Regularization introduces constraints on the model's parameters during training, discouraging overly complex models and promoting simpler, more generalizable solutions.

Here's the theory behind implementing regularization to avoid overfitting in binary classification:

1. Types of Regularization:

- L1 Regularization (Lasso): Adds a penalty term to the loss function proportional to the absolute value of the model's weights.
- L2 Regularization (Ridge): Adds a penalty term to the loss function proportional to the square of the model's weights.
- Elastic Net Regularization: Combines L1 and L2 regularization by adding both penalty terms to the loss function.

2. Regularization Parameter (λ): Regularization strength is controlled by a hyperparameter (λ) that determines the trade-off between fitting the training data and minimizing the regularization term.

3. Regularization in Binary Classification Models:

- Logistic Regression: In logistic regression, regularization is applied by adding the regularization term to the logistic loss function..
- Feedforward Neural Networks: Regularization techniques such as L1 or L2 regularization can be applied to the weights of the neural network's layers.

4. Training with Regularization:

- During model training, the regularization term is added to the loss function, and the model's parameters are updated using optimization algorithms like stochastic gradient descent (SGD) or its variants.

5. Evaluation: After training, the model's performance is evaluated on a separate validation or test dataset using appropriate evaluation metrics for binary classification tasks, such as accuracy, precision, recall, F1-score, and ROC-AUC.

By incorporating regularization techniques into binary classification models, practitioners can effectively prevent overfitting and build models that generalize well to unseen data, improving their robustness and reliability in real-world applications.

A. Implementing regularization to avoid overfitting in binary classification.**Code:**

```
from matplotlib import pyplot

from sklearn.datasets import make_moons

from keras.models import Sequential

from keras.layers import Dense

X, Y = make_moons(n_samples=100, noise=0.2, random_state=1)

n_train = 30

trainX, testX = X[:n_train, :], X[n_train:, :]

trainY, testY = Y[:n_train], Y[n_train:]

# print(trainX)

# print(trainY)

# print(testX)

# print(testY)

model = Sequential()

model.add(Dense(500, input_dim=2, activation='relu'))

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

model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])

history = model.fit(trainX, trainY, validation_data=(testX,
testY), epochs=4000)

pyplot.plot(history.history['accuracy'], label='train')

pyplot.plot(history.history['val_accuracy'], label='test')

pyplot.legend()

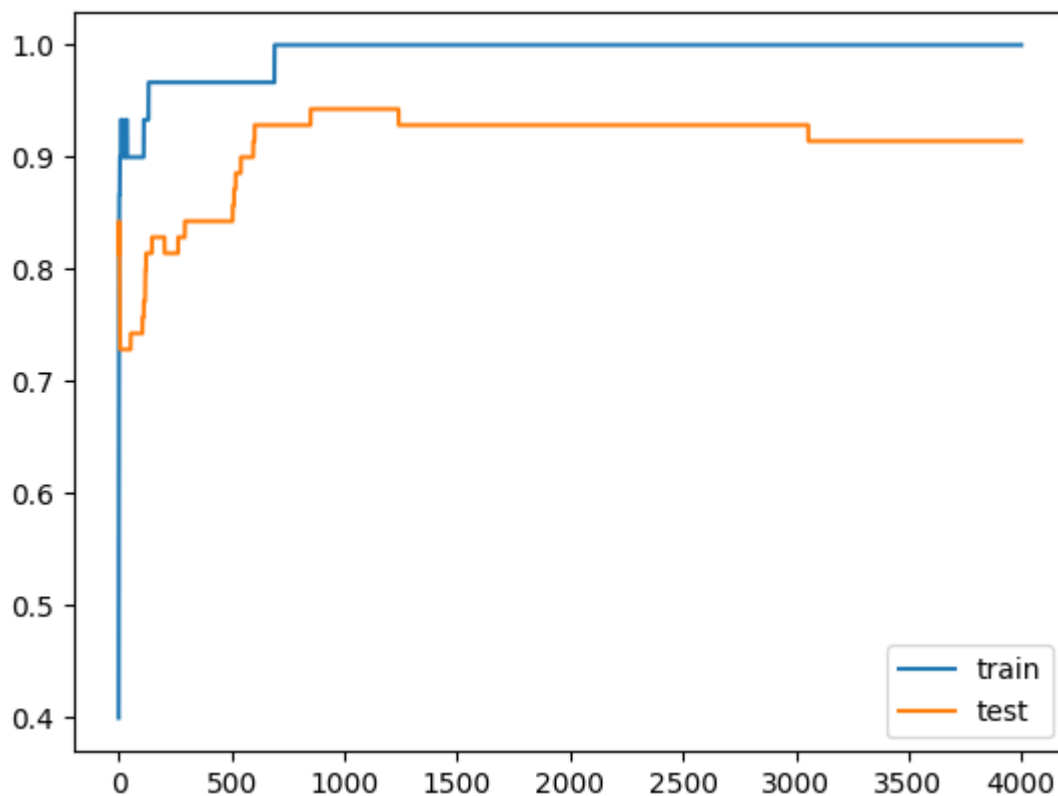
pyplot.show()
```

```

Epoch 1544/4000
1/1 [=====] - 0s 47ms/step - loss: 0.0038 - accuracy: 1.0000 - val_loss: 0.3195 - val_accuracy: 0.9286
Epoch 1545/4000
1/1 [=====] - 0s 70ms/step - loss: 0.0038 - accuracy: 1.0000 - val_loss: 0.3196 - val_accuracy: 0.9286
Epoch 1546/4000
1/1 [=====] - 0s 65ms/step - loss: 0.0038 - accuracy: 1.0000 - val_loss: 0.3197 - val_accuracy: 0.9286
Epoch 1547/4000
1/1 [=====] - 0s 66ms/step - loss: 0.0038 - accuracy: 1.0000 - val_loss: 0.3198 - val_accuracy: 0.9286
Epoch 1548/4000
1/1 [=====] - 0s 59ms/step - loss: 0.0037 - accuracy: 1.0000 - val_loss: 0.3199 - val_accuracy: 0.9286
Epoch 1549/4000
1/1 [=====] - 0s 60ms/step - loss: 0.0037 - accuracy: 1.0000 - val_loss: 0.3200 - val_accuracy: 0.9286
Epoch 1550/4000
1/1 [=====] - 0s 48ms/step - loss: 0.0037 - accuracy: 1.0000 - val_loss: 0.3201 - val_accuracy: 0.9286
Epoch 1551/4000
1/1 [=====] - 0s 49ms/step - loss: 0.0037 - accuracy: 1.0000 - val_loss: 0.3202 - val_accuracy: 0.9286
Epoch 1552/4000
1/1 [=====] - 0s 66ms/step - loss: 0.0037 - accuracy: 1.0000 - val_loss: 0.3203 - val_accuracy: 0.9286
Epoch 1553/4000
1/1 [=====] - 0s 102ms/step - loss: 0.0037 - accuracy: 1.0000 - val_loss: 0.3204 - val_accuracy: 0.9286
Epoch 1554/4000
1/1 [=====] - 0s 79ms/step - loss: 0.0037 - accuracy: 1.0000 - val_loss: 0.3205 - val_accuracy: 0.9286
Epoch 1555/4000
1/1 [=====] - 0s 60ms/step - loss: 0.0037 - accuracy: 1.0000 - val_loss: 0.3206 - val_accuracy: 0.9286

Epoch 3990/4000
1/1 [=====] - 0s 61ms/step - loss: 2.3177e-04 - accuracy: 1.0000 - val_loss: 0.4857 - val_accuracy: 0.9143
Epoch 3991/4000
1/1 [=====] - 0s 60ms/step - loss: 2.3158e-04 - accuracy: 1.0000 - val_loss: 0.4857 - val_accuracy: 0.9143
Epoch 3992/4000
1/1 [=====] - 0s 64ms/step - loss: 2.3140e-04 - accuracy: 1.0000 - val_loss: 0.4858 - val_accuracy: 0.9143
Epoch 3993/4000
1/1 [=====] - 0s 63ms/step - loss: 2.3123e-04 - accuracy: 1.0000 - val_loss: 0.4858 - val_accuracy: 0.9143
Epoch 3994/4000
1/1 [=====] - 0s 69ms/step - loss: 2.3105e-04 - accuracy: 1.0000 - val_loss: 0.4859 - val_accuracy: 0.9143
Epoch 3995/4000
1/1 [=====] - 0s 59ms/step - loss: 2.3086e-04 - accuracy: 1.0000 - val_loss: 0.4859 - val_accuracy: 0.9143
Epoch 3996/4000
1/1 [=====] - 0s 51ms/step - loss: 2.3066e-04 - accuracy: 1.0000 - val_loss: 0.4860 - val_accuracy: 0.9143
Epoch 3997/4000
1/1 [=====] - 0s 73ms/step - loss: 2.3049e-04 - accuracy: 1.0000 - val_loss: 0.4860 - val_accuracy: 0.9143
Epoch 3998/4000
1/1 [=====] - 0s 47ms/step - loss: 2.3031e-04 - accuracy: 1.0000 - val_loss: 0.4861 - val_accuracy: 0.9143
Epoch 3999/4000
1/1 [=====] - 0s 47ms/step - loss: 2.3013e-04 - accuracy: 1.0000 - val_loss: 0.4861 - val_accuracy: 0.9143
Epoch 4000/4000
1/1 [=====] - 0s 69ms/step - loss: 2.2995e-04 - accuracy: 1.0000 - val_loss: 0.4862 - val_accuracy: 0.9143

```



```

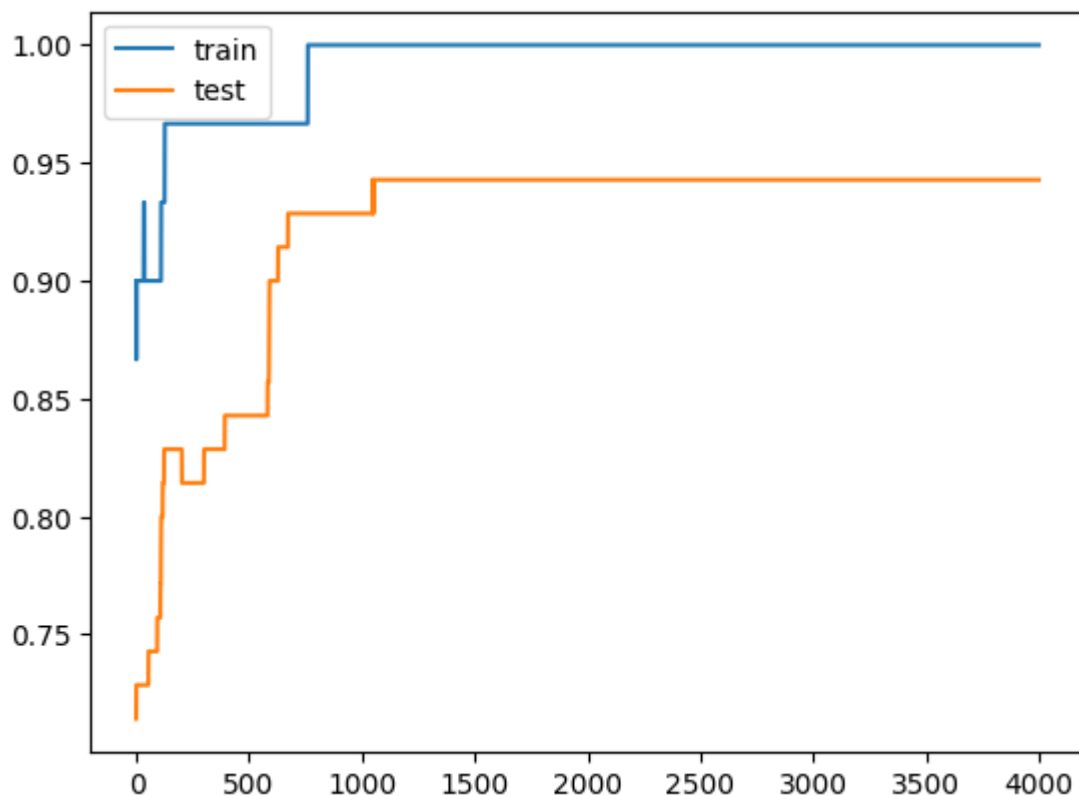
from sklearn.datasets import make_moons
from keras.models import Sequential
from keras.layers import Dense
from keras.regularizers import l2
X, Y = make_moons(n_samples=100, noise=0.2, random_state=1)
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainY, testY = Y[:n_train], Y[n_train:]
# print(trainX)
# print(trainY)
# print(testX)
# print(testY)
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu',
kernel_regularizer=l2(0.001)))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
history = model.fit(trainX, trainY, validation_data=(testX,
testY), epochs=4000)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

```

```

Epoch 3985/4000
1/1 [=====] - 0s 63ms/step - loss: 0.0150 - accuracy: 1.0000 - val_loss: 0.2714 - val_accuracy: 0.9429
Epoch 3986/4000
1/1 [=====] - 0s 65ms/step - loss: 0.0150 - accuracy: 1.0000 - val_loss: 0.2716 - val_accuracy: 0.9429
Epoch 3987/4000
1/1 [=====] - 0s 65ms/step - loss: 0.0150 - accuracy: 1.0000 - val_loss: 0.2718 - val_accuracy: 0.9429
Epoch 3988/4000
1/1 [=====] - 0s 66ms/step - loss: 0.0150 - accuracy: 1.0000 - val_loss: 0.2718 - val_accuracy: 0.9429
Epoch 3989/4000
1/1 [=====] - 0s 68ms/step - loss: 0.0150 - accuracy: 1.0000 - val_loss: 0.2718 - val_accuracy: 0.9429
Epoch 3990/4000
1/1 [=====] - 0s 61ms/step - loss: 0.0150 - accuracy: 1.0000 - val_loss: 0.2717 - val_accuracy: 0.9429
Epoch 3991/4000
1/1 [=====] - 0s 61ms/step - loss: 0.0150 - accuracy: 1.0000 - val_loss: 0.2716 - val_accuracy: 0.9429
Epoch 3992/4000
1/1 [=====] - 0s 69ms/step - loss: 0.0150 - accuracy: 1.0000 - val_loss: 0.2716 - val_accuracy: 0.9429
Epoch 3993/4000
1/1 [=====] - 0s 65ms/step - loss: 0.0150 - accuracy: 1.0000 - val_loss: 0.2716 - val_accuracy: 0.9429
Epoch 3994/4000
1/1 [=====] - 0s 61ms/step - loss: 0.0150 - accuracy: 1.0000 - val_loss: 0.2717 - val_accuracy: 0.9429
Epoch 3995/4000
1/1 [=====] - 0s 53ms/step - loss: 0.0150 - accuracy: 1.0000 - val_loss: 0.2718 - val_accuracy: 0.9429
Epoch 3996/4000
1/1 [=====] - 0s 63ms/step - loss: 0.0150 - accuracy: 1.0000 - val_loss: 0.2719 - val_accuracy: 0.9429
Epoch 3997/4000
1/1 [=====] - 0s 53ms/step - loss: 0.0150 - accuracy: 1.0000 - val_loss: 0.2719 - val_accuracy: 0.9429
Epoch 3998/4000
1/1 [=====] - 0s 59ms/step - loss: 0.0150 - accuracy: 1.0000 - val_loss: 0.2719 - val_accuracy: 0.9429
Epoch 3999/4000
1/1 [=====] - 0s 59ms/step - loss: 0.0150 - accuracy: 1.0000 - val_loss: 0.2718 - val_accuracy: 0.9429
Epoch 4000/4000
1/1 [=====] - 0s 72ms/step - loss: 0.0150 - accuracy: 1.0000 - val_loss: 0.2717 - val_accuracy: 0.9429

```

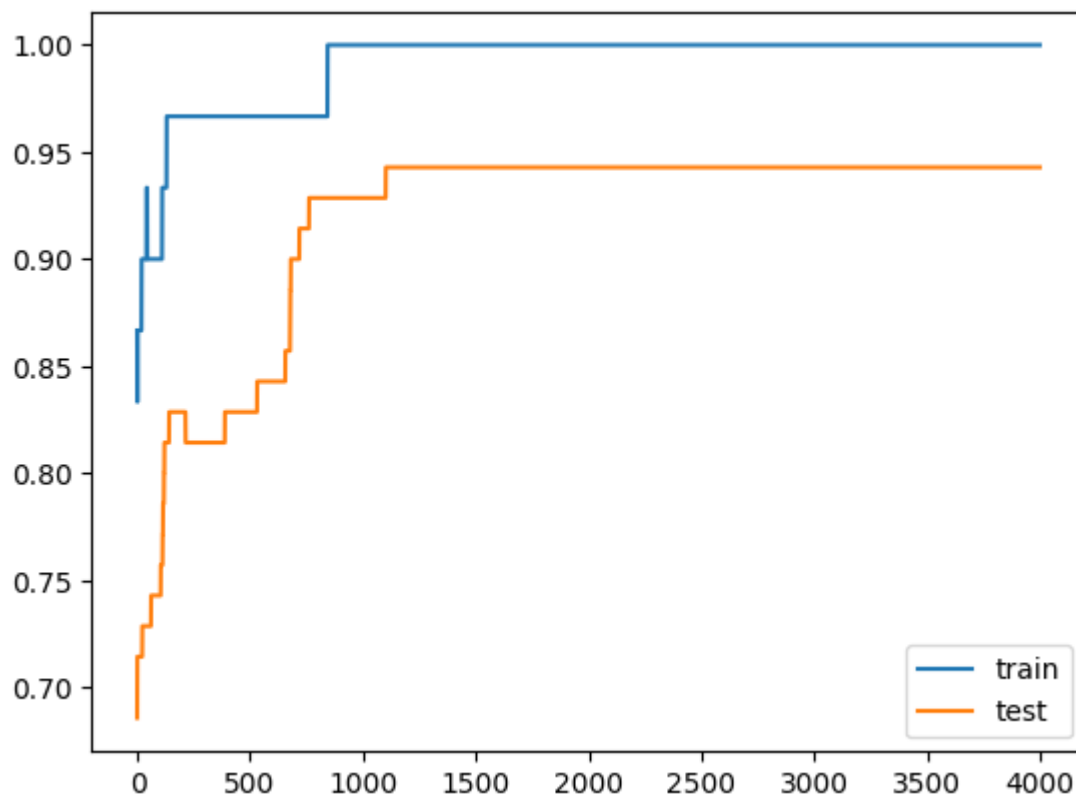



```

from matplotlib import pyplot
from sklearn.datasets import make_moons
from keras.models import Sequential
from keras.layers import Dense
from keras.regularizers import l1
X, Y = make_moons(n_samples=100, noise=0.2, random_state=1)
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainY, testY = Y[:n_train], Y[n_train:]
# print(trainX)
# print(trainY)
# print(testX)
# print(testY)
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu',
kernel_regularizer=l2(0.001)))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
history = model.fit(trainX, trainY, validation_data=(testX,
testY), epochs=4000)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

```

```
Epoch 3987/4000
1/1 [=====] - 0s 47ms/step - loss: 0.0153 - accuracy: 1.0000 - val_loss: 0.2746 - val_accuracy: 0.9429
Epoch 3988/4000
1/1 [=====] - 0s 56ms/step - loss: 0.0153 - accuracy: 1.0000 - val_loss: 0.2746 - val_accuracy: 0.9429
Epoch 3989/4000
1/1 [=====] - 0s 67ms/step - loss: 0.0153 - accuracy: 1.0000 - val_loss: 0.2746 - val_accuracy: 0.9429
Epoch 3990/4000
1/1 [=====] - 0s 67ms/step - loss: 0.0153 - accuracy: 1.0000 - val_loss: 0.2746 - val_accuracy: 0.9429
Epoch 3991/4000
1/1 [=====] - 0s 66ms/step - loss: 0.0153 - accuracy: 1.0000 - val_loss: 0.2746 - val_accuracy: 0.9429
Epoch 3992/4000
1/1 [=====] - 0s 46ms/step - loss: 0.0153 - accuracy: 1.0000 - val_loss: 0.2746 - val_accuracy: 0.9429
Epoch 3993/4000
1/1 [=====] - 0s 51ms/step - loss: 0.0153 - accuracy: 1.0000 - val_loss: 0.2746 - val_accuracy: 0.9429
Epoch 3994/4000
1/1 [=====] - 0s 66ms/step - loss: 0.0153 - accuracy: 1.0000 - val_loss: 0.2746 - val_accuracy: 0.9429
Epoch 3995/4000
1/1 [=====] - 0s 69ms/step - loss: 0.0153 - accuracy: 1.0000 - val_loss: 0.2746 - val_accuracy: 0.9429
Epoch 3996/4000
1/1 [=====] - 0s 65ms/step - loss: 0.0153 - accuracy: 1.0000 - val_loss: 0.2747 - val_accuracy: 0.9429
Epoch 3997/4000
1/1 [=====] - 0s 60ms/step - loss: 0.0153 - accuracy: 1.0000 - val_loss: 0.2748 - val_accuracy: 0.9429
Epoch 3998/4000
1/1 [=====] - 0s 79ms/step - loss: 0.0153 - accuracy: 1.0000 - val_loss: 0.2748 - val_accuracy: 0.9429
Epoch 3999/4000
1/1 [=====] - 0s 60ms/step - loss: 0.0153 - accuracy: 1.0000 - val_loss: 0.2747 - val_accuracy: 0.9429
Epoch 4000/4000
1/1 [=====] - 0s 72ms/step - loss: 0.0153 - accuracy: 1.0000 - val_loss: 0.2746 - val_accuracy: 0.9429
```



Practical 7

Theory

A Recurrent Neural Network (RNN) is a type of artificial neural network designed to process sequential data by maintaining an internal state, or memory, that captures information from previous time steps. Unlike feedforward neural networks, which process each input independently, RNNs can capture temporal dependencies and patterns in sequential data.

Here's how an RNN works:

Temporal Dynamics: RNNs are well-suited for sequential data such as time series, text, audio, and video, where the order of the data points matters. Each input data point in the sequence is associated with a specific time step, and the network processes the sequence one time step at a time.

Recurrent Connections: The defining feature of RNNs is the presence of recurrent connections, which allow information to persist across different time steps. At each time step, the network receives an input and combines it with the internal state (hidden state) from the previous time step. This combination of current input and previous state influences the output and the updated internal state for the current time step.

Hidden State Update: The internal state of an RNN, also known as the hidden state, evolves over time as the network processes the input sequence. The hidden state captures relevant information from previous time steps and serves as the memory of the network. The update rule for the hidden state is determined by the network's parameters (weights and biases) and activation functions.

Parameter Sharing: In RNNs, the same set of parameters (weights and biases) is used at each time step, enabling the network to learn to capture temporal dependencies and patterns in the data. This parameter sharing allows the network to generalize well to sequences of varying lengths and ensures that the memory of the network can adapt to different contexts.

Training: RNNs are trained using supervised learning, where the network learns to map input sequences to corresponding target sequences. During training, the network's parameters are updated using backpropagation through time (BPTT), an extension of the backpropagation algorithm that computes gradients over multiple time steps. This enables the network to learn to capture long-term dependencies in the data.

Applications: RNNs have applications in various domains, including natural language processing (e.g., language modeling, machine translation), time series analysis (e.g., stock price prediction, weather forecasting), speech recognition, and sequence generation (e.g., text generation, music composition).

Overall, RNNs are powerful tools for modeling sequential data and capturing complex temporal relationships. Their ability to maintain internal memory and process sequences of varying lengths makes them well-suited for a wide range of tasks involving sequential data analysis.

A Demonstrate recurrent neural network that learns to perform sequence analysis for stock price.**Code:**

```
#RNN on rain data
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
# Define sequence of 50 days of rain data
rain_data = np.array([2.3, 1.5, 3.1, 2.0, 2.5, 1.7, 2.9, 3.5,
3.0, 2.1,
                        2.5, 2.2, 2.8, 3.2, 1.8, 2.7, 1.9, 3.1,
3.3, 2.0,
                        2.5, 2.2, 2.4, 3.0, 2.1, 2.5, 3.2, 3.1,
1.9, 2.7,
                        2.2, 2.8, 3.1, 2.0, 2.5, 1.7, 2.9, 3.5,
3.0, 2.1,
                        2.5, 2.2, 2.8, 3.2, 1.8, 2.7, 1.9, 3.1,
3.3, 2.0])
#Create input and output sequences for training
def create_sequences(values, time_steps):
    x=[]
    y=[]
    for i in range(len(values)-time_steps):
        x.append(values[i:i+time_steps])
        y.append(values[i+time_steps])
    return np.array(x),np.array(y)

time_steps=4
x_train, y_train=create_sequences(rain_data, time_steps)

#Define RNN Model
model=tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(8, input_shape=(time_steps,1)),
    tf.keras.layers.Dense(1)
])
#Compile model
model.compile(optimizer="adam", loss="mse")
#Train model
history=model.fit(x_train.reshape(-1, time_steps, 1), y_train,
epochs=200)
```

```
Epoch 1/200
2/2 ██████████ 1s 14ms/step - loss: 9.5898
Epoch 2/200
2/2 ██████████ 0s 4ms/step - loss: 9.3640
Epoch 3/200
2/2 ██████████ 0s 4ms/step - loss: 9.1488
Epoch 4/200
2/2 ██████████ 0s 2ms/step - loss: 8.9149
Epoch 5/200
2/2 ██████████ 0s 3ms/step - loss: 8.7611
Epoch 6/200
2/2 ██████████ 0s 11ms/step - loss: 8.4666
Epoch 7/200
2/2 ██████████ 0s 3ms/step - loss: 8.2507
Epoch 8/200
2/2 ██████████ 0s 0s/step - loss: 8.0091
Epoch 9/200
2/2 ██████████ 0s 10ms/step - loss: 7.7380
Epoch 10/200
2/2 ██████████ 0s 7ms/step - loss: 7.5131
Epoch 11/200
2/2 ██████████ 0s 5ms/step - loss: 7.2551
Epoch 12/200
2/2 ██████████ 0s 3ms/step - loss: 7.0374
Epoch 13/200
2/2 ██████████ 0s 3ms/step - loss: 6.7705
Epoch 14/200
2/2 ██████████ 0s 6ms/step - loss: 6.4959
Epoch 15/200
2/2 ██████████ 0s 5ms/step - loss: 6.2855
Epoch 16/200
2/2 ██████████ 0s 3ms/step - loss: 5.9485
Epoch 17/200
2/2 ██████████ 0s 3ms/step - loss: 5.7857
Epoch 18/200
2/2 ██████████ 0s 9ms/step - loss: 5.4892
Epoch 19/200
2/2 ██████████ 0s 7ms/step - loss: 5.2607
Epoch 20/200
2/2 ██████████ 0s 4ms/step - loss: 5.0694
Epoch 21/200
2/2 ██████████ 0s 0s/step - loss: 4.9033
Epoch 22/200
2/2 ██████████ 0s 0s/step - loss: 4.6805
Epoch 23/200
2/2 ██████████ 0s 3ms/step - loss: 4.5167
```

```

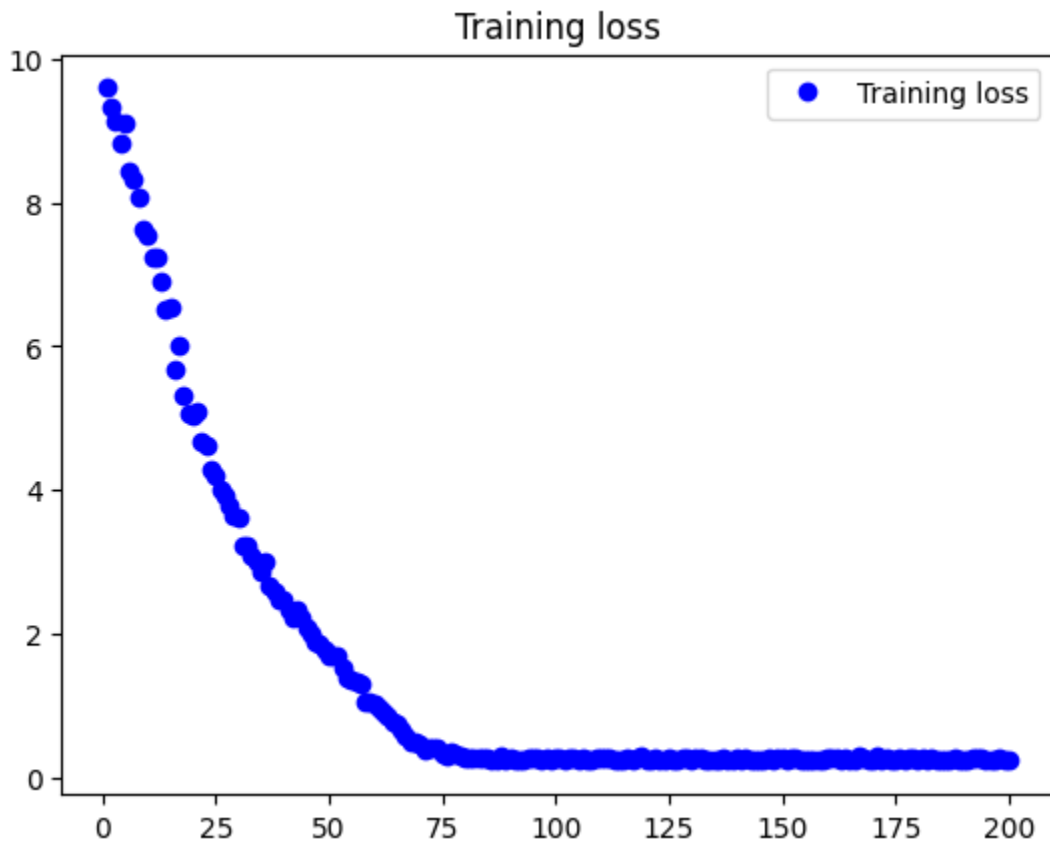
Epoch 179/200
2/2 ————— 0s 4ms/step - loss: 0.2544
Epoch 180/200
2/2 ————— 0s 3ms/step - loss: 0.2529
Epoch 181/200
2/2 ————— 0s 3ms/step - loss: 0.2548
Epoch 182/200
2/2 ————— 0s 0s/step - loss: 0.2536
Epoch 183/200
2/2 ————— 0s 4ms/step - loss: 0.2545
Epoch 184/200
2/2 ————— 0s 12ms/step - loss: 0.2537
Epoch 185/200
2/2 ————— 0s 4ms/step - loss: 0.2537
Epoch 186/200
2/2 ————— 0s 3ms/step - loss: 0.2529
Epoch 187/200
2/2 ————— 0s 3ms/step - loss: 0.2512
Epoch 188/200
2/2 ————— 0s 4ms/step - loss: 0.2537
Epoch 189/200
2/2 ————— 0s 6ms/step - loss: 0.2517
Epoch 190/200
2/2 ————— 0s 4ms/step - loss: 0.2532
Epoch 191/200
2/2 ————— 0s 4ms/step - loss: 0.2511
Epoch 192/200
2/2 ————— 0s 3ms/step - loss: 0.2551
Epoch 193/200
2/2 ————— 0s 3ms/step - loss: 0.2566
Epoch 194/200
2/2 ————— 0s 4ms/step - loss: 0.2537
Epoch 195/200
2/2 ————— 0s 3ms/step - loss: 0.2520
Epoch 196/200
2/2 ————— 0s 6ms/step - loss: 0.2515
Epoch 197/200
2/2 ————— 0s 2ms/step - loss: 0.2508
Epoch 198/200
2/2 ————— 0s 3ms/step - loss: 0.2536
Epoch 199/200
2/2 ————— 0s 4ms/step - loss: 0.2523
Epoch 200/200
2/2 ————— 0s 4ms/step - loss: 0.2498

```

```

#Plot loss over time
loss=history.history["loss"]
epochs=range(1, len(loss)+1)
plt.plot(epochs, loss, "bo", label="Training loss")
plt.title("Training loss")
plt.legend()
plt.show()

```



#Test model on new sequence

```
test_sequence=np.array([2.2,2.5,3.2,2.2])
x_test=np.array([test_sequence])
y_test=model.predict(x_test.reshape(-1, time_steps, 1))
#Print input, output and prediction
print("Previous days' rain data:", test_sequence)
print("Expected rain amount for next day:", y_test[0][0])
prediction=model.predict(np.array([test_sequence]).reshape(1,
time_steps, 1))
print("Prediction:", prediction[0][0])
```

```
1/1 ————— 0s 16ms/step
Previous days' rain data: [2.2 2.5 3.2 2.2]
Expected rain amount for next day: 2.6057234
1/1 ————— 0s 15ms/step
Prediction: 2.6057234
```

Practical 8

Theory

A Deep Autoencoder is a type of artificial neural network architecture used for unsupervised learning and dimensionality reduction. It is composed of an encoder and a decoder, both of which consist of multiple layers of neurons. The primary goal of a deep autoencoder is to learn a compressed representation, or encoding, of the input data in a lower-dimensional space and then reconstruct the original input data from this compressed representation.

Here's how a Deep Autoencoder works:

Encoder: The encoder takes the input data and maps it to a lower-dimensional representation. Each layer in the encoder applies a transformation to the input data, gradually reducing its dimensionality.

Decoder: The decoder takes the compressed representation from the encoder and maps it back to the original input space. Similar to the encoder, each layer in the decoder applies a transformation to the data, gradually increasing its dimensionality until it matches the dimensionality of the original input data.

Objective Function: The training objective of a deep autoencoder is to minimize the reconstruction error, or the difference between the original input data and the reconstructed output.

Training: Deep autoencoders are typically trained using unsupervised learning techniques such as backpropagation and gradient descent. During training, the network learns to optimize the parameters (weights and biases) of both the encoder and decoder simultaneously to minimize the reconstruction error.

Dimensionality Reduction: One of the main applications of deep autoencoders is dimensionality reduction. By learning a compressed representation of the input data in a lower-dimensional space, deep autoencoders can effectively capture the underlying structure and important features of the data while discarding noise and irrelevant information.

Variants: Variants of deep autoencoders include denoising autoencoders, sparse autoencoders, and variational autoencoders, each with different modifications to the architecture or training procedure to achieve specific objectives such as denoising, sparsity, or generative modelling.

Applications: Deep autoencoders have applications in various domains, including image denoising, feature learning, anomaly detection, and data compression.

Overall, deep autoencoders are powerful tools for learning compressed representations of high-dimensional data and extracting meaningful features, making them valuable in various machine learning and data analysis tasks.

A. Performing encoding and decoding of images using deep autoencoders.

Code:

```
import keras
from keras import layers
from keras.datasets import mnist
import numpy as np
encoding_dim=32
#this is our input image
input_img=keras.Input(shape=(784,))
#"encoded" is the encoded representation of the input
encoded=layers.Dense(encoding_dim,
activation='relu')(input_img)
#"decoded" is the lossy reconstruction of the input
decoded=layers.Dense(784, activation='sigmoid')(encoded)
#creating autoencoder model
autoencoder=keras.Model(input_img,decoded)
#create the encoder model
encoder=keras.Model(input_img,encoded)
encoded_input=keras.Input(shape=(encoding_dim,))
#Retrive the last layer of the autoencoder model
decoder_layer=autoencoder.layers[-1]
#create the decoder model
decoder=keras.Model(encoded_input,decoder_layer(encoded_input))
)
autoencoder.compile(optimizer='adam',loss='binary_crossentropy')
#scale and make train and test dataset
(X_train,_),(X_test,_)=mnist.load_data()
X_train=X_train.astype('float32')/255.
X_test=X_test.astype('float32')/255.
X_train=X_train.reshape((len(X_train),np.prod(X_train.shape[1:]))
)
X_test=X_test.reshape((len(X_test),np.prod(X_test.shape[1:]))
)
print(X_train.shape)
print(X_test.shape)
#train autoencoder with training dataset
autoencoder.fit(X_train,X_train,
                epochs=50,
                batch_size=256,
                shuffle=True,
                validation_data=(X_test,X_test))
encoded_imgs =encoder.predict(X_test)
decoded_imgs=decoder.predict(encoded_imgs)

import matplotlib.pyplot as plt
n = 10 # How many digits we will display
plt.figure(figsize=(40, 4))
for i in range(10):
    # display original
    ax=plt.subplot(3, 20, i+1)
    plt.imshow(X_test[i].reshape(28,28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
```

```

ax.get_yaxis().set_visible(False)
#display encoded image
ax = plt.subplot(3, 20, i+1 +20)
plt.imshow(encoded_imgs[i].reshape(8,4))
plt.gray()
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)
# display reconstruction
ax = plt.subplot(3, 20,2*20 +i+ 1)
plt.imshow(decoded_imgs[i].reshape(28,28))
plt.gray()
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)

plt.show()

```

Output:

```

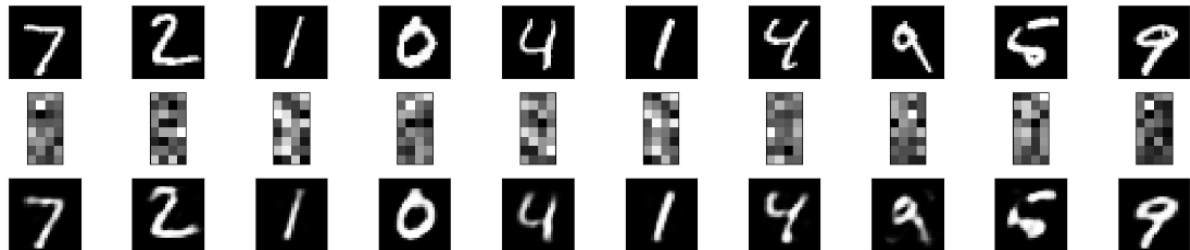
(60000, 784)
(10000, 784)
Epoch 1/50
235/235 ————— 1s 4ms/step - loss: 0.3825 - val_loss: 0.1936
Epoch 2/50
235/235 ————— 1s 4ms/step - loss: 0.1830 - val_loss: 0.1546
Epoch 3/50
235/235 ————— 1s 5ms/step - loss: 0.1501 - val_loss: 0.1335
Epoch 4/50
235/235 ————— 2s 7ms/step - loss: 0.1317 - val_loss: 0.1214
Epoch 5/50
235/235 ————— 2s 8ms/step - loss: 0.1206 - val_loss: 0.1130
Epoch 6/50
235/235 ————— 2s 6ms/step - loss: 0.1130 - val_loss: 0.1071
Epoch 7/50
235/235 ————— 1s 5ms/step - loss: 0.1075 - val_loss: 0.1029
Epoch 8/50
235/235 ————— 1s 4ms/step - loss: 0.1034 - val_loss: 0.0996
Epoch 9/50
235/235 ————— 1s 5ms/step - loss: 0.1006 - val_loss: 0.0974
Epoch 10/50
235/235 ————— 1s 5ms/step - loss: 0.0982 - val_loss: 0.0957
Epoch 11/50
235/235 ————— 1s 5ms/step - loss: 0.0967 - val_loss: 0.0946
Epoch 12/50
235/235 ————— 1s 5ms/step - loss: 0.0957 - val_loss: 0.0938
Epoch 13/50
235/235 ————— 1s 4ms/step - loss: 0.0952 - val_loss: 0.0933
Epoch 14/50
235/235 ————— 1s 4ms/step - loss: 0.0948 - val_loss: 0.0930
Epoch 15/50
235/235 ————— 2s 7ms/step - loss: 0.0945 - val_loss: 0.0927
Epoch 16/50
235/235 ————— 1s 5ms/step - loss: 0.0939 - val_loss: 0.0925
Epoch 17/50
235/235 ————— 1s 5ms/step - loss: 0.0940 - val_loss: 0.0924
Epoch 18/50
235/235 ————— 1s 5ms/step - loss: 0.0939 - val_loss: 0.0923
Epoch 19/50
235/235 ————— 1s 5ms/step - loss: 0.0938 - val loss: 0.0922

```

```

Epoch 37/50
235/235 — 1s 5ms/step - loss: 0.0928 - val_loss: 0.0915
Epoch 38/50
235/235 — 1s 5ms/step - loss: 0.0928 - val_loss: 0.0915
Epoch 39/50
235/235 — 1s 5ms/step - loss: 0.0926 - val_loss: 0.0914
Epoch 40/50
235/235 — 1s 5ms/step - loss: 0.0927 - val_loss: 0.0914
Epoch 41/50
235/235 — 1s 6ms/step - loss: 0.0926 - val_loss: 0.0914
Epoch 42/50
235/235 — 1s 6ms/step - loss: 0.0927 - val_loss: 0.0914
Epoch 43/50
235/235 — 1s 5ms/step - loss: 0.0925 - val_loss: 0.0914
Epoch 44/50
235/235 — 1s 5ms/step - loss: 0.0928 - val_loss: 0.0914
Epoch 45/50
235/235 — 2s 8ms/step - loss: 0.0925 - val_loss: 0.0913
Epoch 46/50
235/235 — 2s 6ms/step - loss: 0.0926 - val_loss: 0.0913
Epoch 47/50
235/235 — 1s 5ms/step - loss: 0.0925 - val_loss: 0.0913
Epoch 48/50
235/235 — 1s 5ms/step - loss: 0.0925 - val_loss: 0.0913
Epoch 49/50
235/235 — 1s 5ms/step - loss: 0.0927 - val_loss: 0.0913
Epoch 50/50
235/235 — 1s 6ms/step - loss: 0.0926 - val_loss: 0.0913
313/313 — 0s 1ms/step
313/313 — 0s 1ms/step

```



Practical 9

Theory

A Convolutional Neural Network (CNN) is a type of artificial neural network specially designed for processing structured grid-like data, such as images. CNNs have revolutionized the field of computer vision and are widely used for tasks such as image classification, object detection, and image segmentation. Here's how a CNN works:

Convolutional Layers: The core building blocks of a CNN are convolutional layers. Each convolutional layer consists of a set of learnable filters (also called kernels) that slide across the input image, computing the dot product between the filter and the input at every position.

Activation Function: After the convolution operation, an activation is applied element-wise to introduce nonlinearity into the network.

Pooling Layers: Pooling layers are often inserted between successive convolutional layers to reduce the spatial dimensions of the feature maps while retaining important information.

Fully Connected Layers: After several convolutional and pooling layers, the feature maps are flattened into a one-dimensional vector and passed to one or more fully connected layers.

Output Layer: The final layer of the CNN is typically a fully connected layer with a softmax activation function for classification tasks or a linear activation function for regression tasks.

Training: CNNs are trained using backpropagation and stochastic gradient descent to minimize a loss function, such as categorical cross-entropy for classification tasks or mean squared error for regression tasks..

Preprocessing: Before feeding the input images into the CNN, preprocessing steps such as normalization, resizing, and data augmentation are often applied to improve the network's robustness and generalization ability.

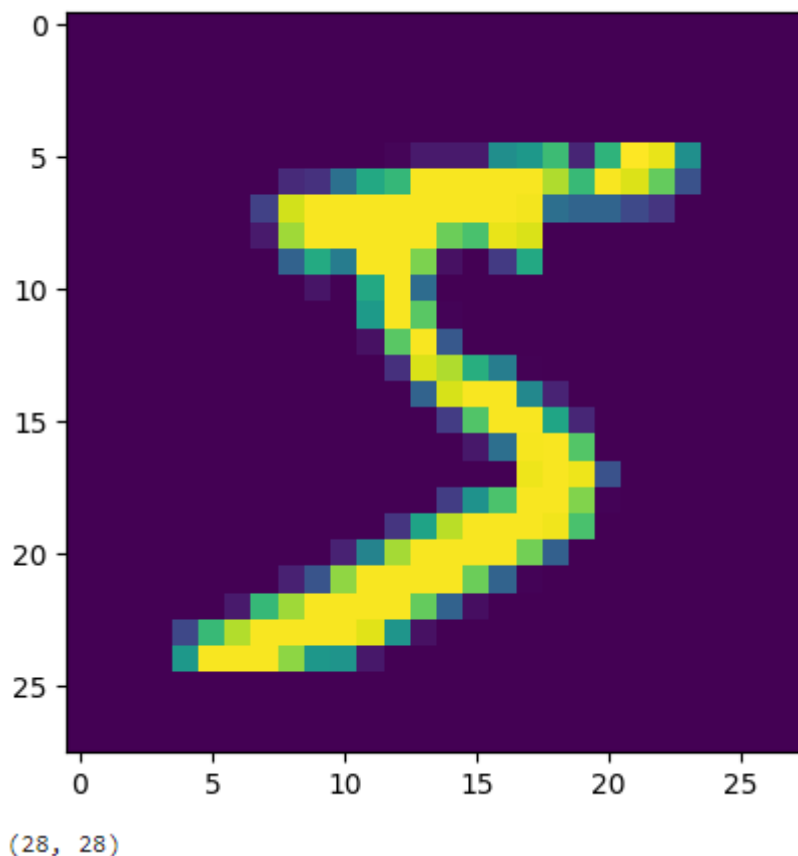
Transfer Learning: CNNs trained on large datasets (e.g., ImageNet) can be used as feature extractors for new tasks with limited data. This approach, known as transfer learning, involves fine-tuning the pre-trained CNN on the new dataset to adapt it to the specific task.

Overall, CNNs are powerful and versatile models for image processing tasks, capable of automatically learning hierarchical representations of visual data, from low-level features to high-level concepts. Their ability to capture spatial patterns and relationships makes them indispensable tools in computer vision applications.

A. Implementation of convolutional neural network to predict numbers from number images.

Code:

```
from keras.datasets import mnist
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten
import matplotlib.pyplot as plt
#download mnist data and split into train and test sets
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
#plot the first image in the dataset
plt.imshow(X_train[0])
plt.show()
print(X_train[0].shape)
```



```
X_train=X_train.reshape(60000,28,28,1)
X_test=X_test.reshape(10000,28,28,1)
Y_train=to_categorical(Y_train)
Y_test=to_categorical(Y_test)
Y_train[0]
print(Y_train[0])
model=Sequential()
#add model layers
#learn image features
model.add(Conv2D(64,kernel_size=3,activation='relu',input_shape=(28,28,1)))
model.add(Conv2D(32,kernel_size=3,activation='relu'))
model.add(Flatten())
```

```

model.add(Dense(10, activation='softmax'))
model.compile(optimizer='adam', loss='categorical_crossentropy',
              metrics=['accuracy'])
#train
model.fit(X_train, Y_train, validation_data=(X_test, Y_test), epochs=3)

```

```
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

```
Epoch 1/3
```

```
C:\Users\Admin\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:99: UserWarning: `input_shape` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first argument to the first layer.
  super().__init__(
```

```
1875/1875 ————— 22s 11ms/step - accuracy: 0.9035 - loss: 0.9964 - val_accuracy: 0.9691 - val_loss: 0.1092
```

```
Epoch 2/3
```

```
1875/1875 ————— 20s 11ms/step - accuracy: 0.9777 - loss: 0.0740 - val_accuracy: 0.9725 - val_loss: 0.0920
```

```
Epoch 3/3
```

```
1875/1875 ————— 20s 11ms/step - accuracy: 0.9863 - loss: 0.0437 - val_accuracy: 0.9778 - val_loss: 0.0863
```

```
<keras.src.callbacks.history.History at 0x1a01f5dacd0>
```

```

print(model.predict(X_test[:4]))
#actual results for 1st 4 images in the test set
print(Y_test[:4])

```

```
1/1 ————— 0s 94ms/step
```

```
[[8.04722546e-08 1.89558986e-16 1.22223832e-06 7.07511499e-05
 2.91497132e-13 4.65019516e-12 2.33310800e-18 9.99927878e-01
 1.23625625e-08 3.85950072e-09]
```

```
[1.02603713e-06 1.90077216e-08 9.99825299e-01 1.70770198e-09
 7.33869955e-14 2.89273618e-12 1.73665627e-04 3.18614411e-13
 7.03306524e-09 1.18175555e-14]
```

```
[1.33387402e-06 9.99977946e-01 3.33661751e-06 1.49757611e-07
 8.77386071e-07 8.43451673e-08 6.16722389e-08 5.01361728e-06
 1.11993759e-05 2.21395458e-10]
```

```
[1.00000000e+00 9.34987154e-16 1.69151165e-13 4.48174069e-15
 1.33610268e-15 1.94097031e-13 2.23832295e-08 2.11456248e-15
 1.52452310e-13 5.96708360e-12]]
```

```
[[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
```

```
[0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
```

```
[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
```

```
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

Practical 10

Theory

Denoising images using autoencoders is a technique that leverages the ability of autoencoders to learn meaningful representations of input data by training them to remove noise from corrupted images. Here's the theory behind this process:

Autoencoder Architecture: An autoencoder is a type of artificial neural network that consists of an encoder and a decoder. The encoder compresses the input data into a lower-dimensional representation, while the decoder reconstructs the original input data from this representation. The architecture aims to learn a compact and informative representation of the input data.

Noisy Input Data: To train an autoencoder for denoising, we need pairs of noisy images and their clean counterparts. The noisy images are created by adding random noise to the clean images, simulating real-world scenarios where images may be corrupted during acquisition or transmission.

Training Objective: The objective of training the denoising autoencoder is to minimize the reconstruction error between the denoised output and the clean input images. During training, the autoencoder learns to map the noisy input images to their corresponding clean versions by capturing the underlying structure and features of the data while filtering out the noise.

Loss Function: The loss function used to train the denoising autoencoder is typically a measure of dissimilarity between the denoised output and the clean input images. Common loss functions include Mean Squared Error (MSE) or Binary Cross-Entropy, depending on the pixel intensity range of the images.

Training Procedure: The denoising autoencoder is trained using backpropagation and gradient descent optimization techniques. The network's parameters (weights and biases) are adjusted iteratively to minimize the loss function, improving the quality of the reconstructed images.

Denoising Process: Once trained, the denoising autoencoder can be used to remove noise from new, unseen images by feeding them into the encoder, obtaining the compressed representation, and then passing it through the decoder to reconstruct the clean version of the image. The autoencoder effectively learns to filter out the noise while preserving the essential features of the input data.

Evaluation: The performance of the denoising autoencoder can be evaluated using various metrics such as Peak Signal-to-Noise Ratio (PSNR), Structural Similarity Index (SSIM), or visual inspection by human evaluators. These metrics assess how well the denoising autoencoder restores the clean images from the noisy input data.

Overall, denoising images using autoencoders is a valuable technique for improving the quality of images corrupted by noise, enabling better analysis and interpretation of visual data in various applications such as medical imaging, surveillance, and photography.

A. Denoising of images using autoencoder.

Code:

```
#Denoising of images using autoencoder
import keras
from keras.datasets import mnist
from keras import layers
import numpy as np
from keras.callbacks import TensorBoard
import matplotlib.pyplot as plt
(X_train, _), (X_test, _) = mnist.load_data()
X_train = X_train.astype('float32') / 255.
X_test = X_test.astype('float32') / 255.
X_train = np.reshape(X_train, (len(X_train), 28, 28, 1))
X_test = np.reshape(X_test, (len(X_test), 28, 28, 1))
noise_factor = 0.5
X_train_noisy = X_train + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=X_train.shape)
X_test_noisy = X_test + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=X_test.shape)
X_train_noisy = np.clip(X_train_noisy, 0., 1.)
X_test_noisy = np.clip(X_test_noisy, 0., 1.)
n = 10
plt.figure(figsize=(20, 2))
for i in range(1, n+1):
    ax = plt.subplot(1, n, i)
    plt.imshow(X_test_noisy[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

Downloading data from <https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz>
11490434/11490434 12s 1us/step



```
input_img = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = layers.MaxPooling2D((2, 2), padding='same')(x)
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(x)
encoded = layers.MaxPooling2D((2, 2), padding='same')(x)
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(encoded)
x = layers.UpSampling2D((2, 2))(x)
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = layers.UpSampling2D((2, 2))(x)
decoded = layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)
autoencoder = keras.Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(X_train_noisy, X_train,
```



```

epochs=3,
batch_size=128,
shuffle=True,
validation_data=(X_test_noisy,X_test),callbacks=[TensorBoard(log_dir='/tmo/tb',histogram_freq=0,write_graph=False)]
predictions=autoencoder.predict(X_test_noisy)
m=10
plt.figure(figsize=(20,2))
for i in range(1,m+1):
    ax=plt.subplot(1,m,i)
    plt.imshow(predictions[i].reshape(28,28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()

```

```

Epoch 1/3
469/469 — 20s 38ms/step - loss: 0.2436 - val_loss: 0.1183
Epoch 2/3
469/469 — 17s 37ms/step - loss: 0.1174 - val_loss: 0.1107
Epoch 3/3
469/469 — 17s 37ms/step - loss: 0.1104 - val_loss: 0.1061
313/313 — 2s 5ms/step

```

