

Practical 1

Matrix Multiplication, Eigen Vectors

Aim: Perform Matrix multiplication and finding eigen vectors and eigen values using tensorflow

Description:

Matrix:

- In mathematics, a matrix is a structured arrangement of numbers or symbols in rows and columns.
- It serves as a fundamental tool for organizing and manipulating data in various mathematical operations, such as addition, subtraction, multiplication, and inversion.
- Matrices find extensive applications in diverse fields ranging from computer graphics and quantum mechanics to economics and engineering.

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1q} \\ a_{21} & a_{22} & \dots & a_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ a_{p1} & a_{p2} & \dots & a_{pq} \end{bmatrix}$$

Vectors:

- Vectors are mathematical entities characterized by both magnitude and direction. They are represented geometrically as arrows in a multi-dimensional space, with their length signifying the magnitude and their orientation indicating the direction.
- Vectors play a crucial role in many areas of mathematics, physics, and engineering. They are used to represent forces, velocities, and displacements in physics; define points and directions in geometry; and model quantities such as preferences and probabilities in economics and statistics.

$$x = [x_1 \ x_2 \ x_3 \ \dots \ x_n] \quad \text{or} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

Eigen vectors:

- Eigen vectors are special vectors associated with linear transformations that retain their direction but may be scaled during the transformation process.
- They are characterized by the property that when a linear transformation is applied to them, they are only scaled by a scalar factor, without changing their direction.
- They provide valuable insights into the behavior of linear systems, help analyze stability and equilibrium points in dynamic systems

- $Av = \lambda v$

Eigen values:

- Eigen values are scalars associated with eigen vectors that represent the factor by which the corresponding eigen vectors are scaled during a linear transformation. They signify the amount of stretching or compression experienced by the eigen vectors when subjected to the transformation.
- Eigen values are crucial in understanding the behavior of linear systems, stability analysis, and solving differential equations.


Tensor:

- A tensor is a mathematical object that generalizes the concept of vectors and matrices to higher dimensions. It represents multidimensional arrays of data, characterized by multiple indices that specify components along different axes or directions.
- Tensors find widespread applications in physics, engineering, and computer science, where they are used to describe the properties of physical systems, model complex phenomena, and facilitate computations in various algorithms

Code

```
import tensorflow as tf
print("Matrix Multiplication Demo")
a=tf.constant([1,2,3,4,5,6],shape=[2,3])
print(a)
b=tf.constant([7,8,9,10,11,12],shape=[3,2])
print(b)
c=tf.matmul(a,b)
print("Product:",c)
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:\n{}\n\nEigen Values:\n{}\n".format(eigen_vectors_A,eigen_values_A))
```

 **Matrix Multiplication Demo**
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:
[[5.327501 4.2850094]
 [8.220428 3.3663926]]

Eigen Vectors:
[[-0.6639116 0.74781114]
 [0.74781114 0.6639116]]

Eigen Values:
[-3.9317572 12.625651]

Code and output:

```
import tensorflow as tf
print("Matrix Multiplication Demo")
a=tf.constant([8,3,7,9,1,4],shape=[2,3])
print(a)
b=tf.constant([4,6,3,7,5,1],shape=[3,2])
print(b)
```

```
c=tf.matmul(a,b)
print("Product:",c)
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:\n{}\n\nEigen Values:\n{}\n".format(eigen_vectors_A,eigen_values_A))
```

OUTPUT:

```
} Matrix Multiplication Demo
tf.Tensor(
[[8 3 7]
 [9 1 4]], shape=(2, 3), dtype=int32)
tf.Tensor(
[[4 6]
 [3 7]
 [5 1]], shape=(3, 2), dtype=int32)
Product: tf.Tensor(
[[76 76]
 [59 65]], shape=(2, 2), dtype=int32)
Matrix A:
[[7.464885 9.184841]
 [9.12247 9.998349]]

Eigen Vectors:
[[-0.75416803 -0.6566815 ]
 [ 0.6566815 -0.75416803]]

Eigen Values:
[-0.47838098 17.941614 ]
```

Practical 2

Solving XOR problem using deep feed forward network

Aim: Solving XOR problem using deep feed forward network

Description:

A deep feedforward network, also known as a feedforward neural network or multilayer perceptron (MLP), is a foundational architecture in deep learning. It consists of multiple layers of interconnected nodes, with each layer feeding its output forward as input to the next layer. These networks are characterized by their ability to learn complex representations of data through hierarchical feature extraction. They are widely used for supervised learning tasks such as classification and regression, where the input-output mapping is learned from labeled data. Training is typically done using techniques like backpropagation and stochastic gradient descent to minimize a specified loss function.

Sigmoid Function:

The sigmoid function is a popular activation function used in neural networks, especially in the output layer for binary classification tasks. It squashes the output of a neuron to a value between 0 and 1, which can be interpreted as a probability.

Mathematically, the sigmoid function is defined as:

$$f(x) = \frac{1}{1 + e^{-x}}$$

$$f(x) = \frac{1}{1 + e^{-x}}$$

Here's what the sigmoid function does:

- For large positive values of x , $f(x)$ approaches 1.
- For large negative values of x , $f(x)$ approaches 0.
- For $x = 0$, $f(x)$ is exactly 0.5.

The sigmoid function introduces non-linearity to the network, letting it learn from the error and adjust the weights during training. However, it's worth noting that sigmoid can suffer from the vanishing gradient problem, especially in deeper networks, which is why other activation functions like ReLU are often preferred for hidden layers.

Handwritten:

22306A1012

$$f(x; W, a, w, b) = w^T \max\{0, Wx + a\} + b$$

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$a = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$w = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$

Let X be design matrix containing all four points

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

multiply input matrix by first layer's weight matrix

$$XW = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$$

add bias a

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

Teacher's Signature:...

To finish computing value of h for each example, we apply rectified linear transformation.

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

we finish with multiplying by weight vector w

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Code:

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

X = np.array([[0.,0.],[0.,1.],[1.,0.],[1.,1.]])
print("Input data:")
print(X)

y = np.array([0.,1.,1.,0.])
print("\nTarget labels:")
print(y)

model.get_weights()
model.fit(X,y,epochs = 500)
predictions = model.predict(X)
print("\nPredictions after training:")
print(predictions)
```

Output:

```
Input data:
[[0. 0.]
 [0. 1.]
 [1. 0.]
 [1. 1.]]

Target labels:
[0. 1. 1. 0.]
*****

Epoch 1/500
1/1 [=====] - 1s 985ms/step -
Epoch 2/500

1/1 [=====] - 0s 11ms/step - loss: 0.5331 - accuracy: 0.7500
Epoch 499/500
1/1 [=====] - 0s 10ms/step - loss: 0.5329 - accuracy: 0.7500
Epoch 500/500
1/1 [=====] - 0s 13ms/step - loss: 0.5327 - accuracy: 0.7500
1/1 [=====] - 0s 50ms/step

Predictions after training:
[[0.43769902]
 [0.7582889 ]
 [0.43766946]
 [0.36304653]]
```

Learning:

This code defines a neural network model using Keras to perform binary classification on a dataset with four samples, each containing two features. The model is trained for 500 epochs using binary crossentropy loss and the Adam optimizer. After training, it makes predictions on the input data, yielding the output probabilities of belonging to the positive class.

Practical 3

Binary Classification Task

Aim: Implementing deep neural network for performing binary classification task.

Description:

Binary Classification

- Goal: Classify data points into exactly two categories (classes).
- Applications: Spam filtering, sentiment analysis (positive/negative reviews), image recognition (cat/dog), fraud detection (fraudulent/legitimate transaction).
- Training Data: Labeled data examples with each point belonging to one of the two classes.
- Model Learning: A binary classification model learns to distinguish between the two classes based on the features (attributes) of the data points.
- Evaluation: Performance is often measured using metrics like accuracy (percentage of correctly classified examples), precision (proportion of true positives among predicted positives), recall (proportion of identified true positives), and F1-score (harmonic mean of precision and recall).

TensorFlow and Keras

TensorFlow and Keras are popular deep learning libraries frequently used in tandem for building and training neural networks. TensorFlow provides a robust framework for developing machine learning models, offering low-level control over model architecture and computation. Keras, on the other hand, offers a high-level API that simplifies the process of building neural networks, enabling rapid prototyping and experimentation. By integrating Keras within TensorFlow, users can leverage the simplicity and flexibility of Keras while benefiting from the scalability and performance optimizations of TensorFlow's backend. This combination empowers developers to efficiently create and train neural networks for a wide range of tasks, from image classification to natural language processing.

Code:

```
# pip install keras
from keras.models import Sequential
from keras.layers import Dense
import pandas as pd

names = [
    "No. of pregnancies",
    "Glucose level",
    "Blood Pressure",
    "skin thickness",
    "Insulin",
    "BMI",
    "Diabetes pedigree",
    "Age",
    "Class",
]

#csv file with no column names expected
df = pd.read_csv("/content/pima-indians-diabetes.data.csv", names=names)
df.head(3)
binaryc = Sequential()

from tensorflow.tools.docs.doc_controls import doc_in_current_and_subclasses

binaryc.add(Dense(units=10, activation="relu", input_dim=8))
binaryc.add(Dense(units=8, activation="relu"))
binaryc.add(Dense(units=1, activation="sigmoid"))
binaryc.compile(loss="binary_crossentropy", optimizer="adam", metrics="accuracy")
X = df.iloc[:, :-1]
y = df.iloc[:, -1]

from sklearn.model_selection import train_test_split

xtrain, xtest, ytrain, ytest = train_test_split(X, y, test_size=0.25, random_state=1)
xtrain.shape
ytrain.shape
binaryc.fit(xtrain, ytrain, epochs=200, batch_size=20)
```

```

predictions = binaryc.predict(xtest)
predictions.shape
class_labels = []
for i in predictions:
    if i > 0.5:
        class_labels.append(1)
    else:
        class_labels.append(0)
class_labels
from sklearn.metrics import accuracy_score

print("Accuracy Score", accuracy_score(ytest, class_labels))

```

Output:

2 df.head(3)
3

	No. of pregnancies	Glucose level	Blood Pressure	skin thickness	Insulin	BMI	Diabetes pedigree	Age	Class
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

Next steps: [Generate code with df](#) [View recommended plots](#)

```

6 binaryc.fit(xtrain, ytrain, epochs=200, batch_size=20)
7 predictions = binaryc.predict(xtest)
8 print(predictions.shape)

```

(576, 8)
(576,)
Epoch 1/200
29/29 [=====] - 2s 6ms/step - loss: 9.4945 - accuracy: 0.6545
Epoch 2/200
29/29 [=====] - 0s 5ms/step - loss: 4.0937 - accuracy: 0.6233
Epoch 3/200
29/29 [=====] - 0s 5ms/step - loss: 1.9831 - accuracy: 0.5868
Epoch 4/200
29/29 [=====] - 0s 7ms/step - loss: 1.3416 - accuracy: 0.5469
Epoch 5/200
29/29 [=====] - 0s 7ms/step - loss: 1.0911 - accuracy: 0.5955
Epoch 6/200

```
8 from sklearn.metrics import accuracy_score  
9  
10 print("Accuracy Score", accuracy_score(ytest, class_labels))
```

⇒ Accuracy Score 0.78125

Learning:

This code builds and trains a neural network using Keras to predict diabetes from given health data. It starts by importing necessary libraries and defining the dataset's features and labels. Then, a neural network model is created with three layers. The model is compiled with appropriate loss function and optimizer. Next, the data is split into training and testing sets. The model is trained on the training data, and predictions are made on the test data. Finally, the accuracy of the predictions is evaluated using the test labels, and the accuracy score is printed.

Practical 4

Multiclass Classification with feed forward network

Aim: Using feed Forward Network with multiple hidden layers for performing multiclass classification and predicting the class.

Description:

Describe multiclass classification in detail.

- Objective:
 - Multiclass classification aims to assign input data points to one of several predefined classes or categories.
- Dataset:
 - A dataset for multiclass classification consists of input features (independent variables) and corresponding class labels (dependent variables).
 - Each data point in the dataset is associated with a single class label from a set of multiple classes.
- Classes:
 - Classes represent the distinct categories into which data points can be classified.
 - For example, in a dataset of handwritten digit recognition, classes may represent digits from 0 to 9.
- Feature Representation:
 - Input features are represented as a feature vector for each data point.
 - These features could be numerical, categorical, or even text-based, depending on the nature of the problem.
- Model Training:
 - During training, the model learns patterns and relationships between input features and class labels from the labeled training data.

- Common machine learning algorithms used for multiclass classification include logistic regression, decision trees, random forests, support vector machines (SVM), and neural networks.
- Loss Function:
 - In multiclass classification, a suitable loss function is used to quantify the difference between predicted class probabilities and the true class labels.
 - Common loss functions for multiclass classification include cross-entropy loss, categorical cross-entropy loss, and softmax loss.
- Prediction:
 - After the model is trained, it can be used to predict the class labels for new, unseen data points.
 - The model computes the probability distribution over all classes for each input instance and assigns the class with the highest probability as the predicted class.
- Evaluation Metrics:
 - Evaluation metrics are used to assess the performance of the multiclass classification model.
 - Common evaluation metrics include accuracy, precision, recall, F1-score, confusion matrix, and ROC curves.
- Handling Class Imbalance:
 - In real-world scenarios, class imbalance may occur when some classes have significantly fewer instances than others.
 - Techniques such as oversampling, undersampling, and class-weighted loss functions can be employed to address class imbalance.
- Application:
 - Multiclass classification has various applications across different domains, including image recognition, natural language processing, sentiment analysis, medical diagnosis, and recommendation systems.

feedforward neural network

A feedforward neural network, often referred to as a multilayer perceptron (MLP), is a fundamental architecture in artificial neural networks. It consists of an input layer, one or more hidden layers, and an output layer. Information flows forward through the network, with each layer of neurons processing the input and passing its output to the next layer. Neurons within each layer compute a weighted sum of their inputs, followed by the application of an activation

function, which introduces nonlinearity to the network. The network's parameters, including weights and biases, are learned from labeled training data using techniques like gradient descent, enabling the network to map input data to output predictions.

Code:

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

df = pd.read_csv("/content/flower_1.csv")
df.head()

x=df.iloc[:, :-1].astype(float)
y=df.iloc[:, -1]

print(x.shape)
print(y.shape)

#labelencode y
from sklearn.preprocessing import LabelEncoder
lb=LabelEncoder()
y=lb.fit_transform(y)
y

import numpy as np
from tensorflow.keras.utils import to_categorical
#from keras.utils import np_utils
encoded_Y = to_categorical(y)
encoded_Y

#creating a model
model = Sequential()

model.add(Dense(units = 10, activation = 'relu', input_dim = 4))
model.add(Dense(units = 8, activation = 'relu'))
model.add(Dense(units = 3, activation = 'softmax'))

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

model.fit(x,encoded_Y,epochs = 400,batch_size = 10)
```

```
predict = model.predict(x)
print(predict)

for i in range(35,150,3):
    print(predict[i],encoded_Y[i])

actual = []

for i in range(0,150):
    actual.append(np.argmax(predict[i]))

print(actual)

newdf = pd.DataFrame(list(zip(actual,y)),columns = ['Actual','Predicted'])
newdf
```

Output:

```
[18] 1 df.head()
```

	sepal length	sepal width	petal length	petal width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

```
1 print(x.shape)
2 print(y.shape)
```

```
(150, 4)
(150,)
```


[illegible]

```
1 newdf = pd.DataFrame(newdf)
2 newdf
```

	Actual	Predicted
0	0	0
1	0	0
2	0	0
3	0	0
4	0	0
...
145	2	2
146	2	2
147	2	2
148	2	2
149	2	2

Learning:

This code snippet loads a dataset from a CSV file containing features and labels. It preprocesses the data by encoding the labels and one-hot encoding them. A feedforward neural network is then built using Keras, comprising input, hidden, and output layers. The model is trained using the compiled parameters and training data. After training, it predicts class probabilities for the input data and compares some predictions with actual labels. Finally, the predicted and actual labels are combined into a DataFrame for analysis and evaluation.

Practical 5

K-Fold Cross Validation

5 a)

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

Description:

1. **Feedforward Deep Network:** A feedforward deep network, also known as a feedforward neural network, is a type of artificial neural network where connections between units do not form cycles. Information flows in one direction, from input to output layers, without any feedback loops. This architecture is commonly used for regression tasks, where the goal is to predict a continuous output value based on input features.
2. **Regression:** Regression is a type of supervised learning task where the goal is to predict a continuous output variable based on one or more input features. In the context of this code, the neural network model is trained to predict housing prices (a continuous variable) based on various features such as the number of rooms, crime rate, and accessibility to highways.
3. **KFold Cross-Validation:** KFold cross-validation is a technique used to assess the performance of a machine learning model. It involves splitting the dataset into K folds (or subsets) of approximately equal size. The model is trained K times, each time using K-1 folds for training and one fold for validation. This process allows for a more robust estimation of the model's performance compared to a single train-test split.
4. **Neural Network Architecture:** The neural network architecture used in this code consists of multiple layers of neurons (Dense layers) organized in a sequential manner. Each layer applies a transformation to the input data using activation functions like ReLU (Rectified Linear Unit). The final layer outputs a continuous value, making it suitable for regression tasks.
5. **Pipeline Construction:** A scikit-learn **Pipeline** is constructed to sequentially apply a list of transforms and a final estimator. In this case, the transforms include standardization (**StandardScaler**) to scale the input features and the final estimator is the Keras neural network model (**KerasRegressor**). This pipeline ensures consistent preprocessing of data during cross-validation and simplifies the evaluation process.

Code:

```
# !pip install keras (2.15.0)

# !pip install scikit_learn

# !pip install scikeras


import pandas as pd

from keras.models import Sequential

from keras.layers import Dense

# from keras.wrappers.scikit_learn import KerasRegressor

from scikeras.wrappers import KerasRegressor

from sklearn.model_selection import cross_val_score, KFold

from sklearn.preprocessing import StandardScaler

from sklearn.pipeline import Pipeline

from sklearn.neural_network import MLPRegressor


dataframe = pd.read_csv("MscIT\Semester 4\Deep_Learning\Practical05\housing.csv")

# dataframe = pd.read_csv("/content/housing.csv")

dataset = dataframe.values


# Print the shape of dataset to verify the number of features and samples

print("Shape of dataset:", dataset.shape)


# Ensure correct slicing for features and target variable

X = dataset[:, :-1] # Select all columns except the last one as features

Y = dataset[:, -1] # Select the last column as target variable
```

```
def wider_model():  
    model = Sequential()  
    model.add(Dense(15, input_dim=13, kernel_initializer='normal', activation='relu'))  
    # model.add(Dense(20, input_dim=13, kernel_initializer='normal', activation='relu'))  
    model.add(Dense(13, kernel_initializer='normal', activation='relu'))  
    model.add(Dense(1, kernel_initializer='normal'))  
    model.compile(loss='mean_squared_error', optimizer='adam')  
    return model  
  
estimators = []  
estimators.append(('standardize', StandardScaler()))  
estimators.append(('mlp', KerasRegressor(build_fn=wider_model, epochs=10, batch_size=5)))  
pipeline = Pipeline(estimators)  
kfold = KFold(n_splits=10)  
  
results = cross_val_score(pipeline, X, Y, cv=kfold)  
print("Wider: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

Output:

```
24/24          0s 2ms/step - loss: 17.0023  
Epoch 10/10  
92/92 ██████████ 0s 1ms/step - loss: 21.4643  
10/10 ██████████ 0s 2ms/step  
Wider: 0.07 (0.73) MSE
```



Learning:

1. **Importing and Installing Libraries:** The code starts by installing necessary libraries like Keras, scikit-learn, and scikeras using pip. These libraries are essential for building and evaluating the neural network model.
2. **Data Loading and Preprocessing:** The dataset is loaded using pandas from a CSV file named "housing.csv". The dataset is then split into input features (X) and target variable (Y). Standardization is performed on the input features using **StandardScaler** from scikit-learn to ensure all features have a mean of 0 and a standard deviation of 1.
3. **Neural Network Architecture:** A wider feedforward neural network model is defined using Keras. It consists of an input layer with 13 neurons (matching the number of features), a hidden layer with 15 neurons and ReLU activation function, another hidden layer with 13 neurons and ReLU activation function, and an output layer with 1 neuron for regression. The model is compiled with mean squared error loss function and Adam optimizer.
4. **Pipeline Construction:** A scikit-learn **Pipeline** is constructed to sequentially apply a list of transforms and a final estimator. In this case, the transforms include standardization (**StandardScaler**), followed by the Keras neural network model (**KerasRegressor**). This pipeline ensures consistent preprocessing of data during cross-validation.
5. **Cross-Validation and Model Evaluation:** Cross-validation is performed using scikit-learn's **cross_val_score** with 10 folds. During each fold, the pipeline is trained and evaluated on different subsets of the dataset. The mean squared error (MSE) is computed for each fold, and the average MSE across all folds is calculated and printed as the evaluation metric for the wider neural network model.
6. **Comparison with Other Models:** The code provides a benchmark for evaluating the performance of the wider neural network model. Other models or configurations could be tested similarly by defining different neural network architectures or adjusting hyperparameters, and their performance can be compared to determine the most effective model for the given regression task.

5 b)

Aim: Evaluating feed forward deep network for multiclass Classification using KFold crossvalidation.

Description:

1. **Feedforward Deep Network:** The term "feedforward deep network" refers to a type of artificial neural network where connections between units do not form cycles. In other words, information flows in one direction, from input to output layers. This architecture is commonly used for tasks such as classification, regression, and function approximation.
2. **Multiclass Classification:** Multiclass classification is a supervised learning task where the goal is to assign one of multiple class labels to each input sample. In this context, each sample can belong to only one class, and the goal is to predict the correct class label for new, unseen data points.
3. **KFold Cross-Validation:** KFold cross-validation is a technique used to assess the performance of a machine learning model. It involves splitting the dataset into K folds (or subsets) of approximately equal size. The model is trained K times, each time using K-1 folds for training and one fold for validation. This process allows for a more robust estimation of the model's performance compared to a single train-test split.
4. **Label Encoding:** Label encoding is a preprocessing step often used in machine learning for converting categorical labels into numeric format. In the context of multiclass classification, it involves mapping each class label to a unique integer identifier. This allows the model to understand and process the class labels during training.
5. **One-Hot Encoding:** One-hot encoding is another preprocessing step used in machine learning, particularly for multiclass classification tasks. It involves converting categorical variables into binary vectors, where each vector represents a unique class label. In this encoding scheme, a binary value of 1 is assigned to the position corresponding to the class label, while all other positions are set to 0. This representation is useful for training neural networks, as it ensures that class labels are represented in a meaningful and consistent way.
6. **Model Evaluation Metrics:** In the context of evaluating a deep neural network for multiclass classification, various evaluation metrics can be used to assess the model's performance. Common metrics include accuracy, precision, recall, F1 score, and

confusion matrix. These metrics provide insights into different aspects of the model's performance, such as its ability to correctly classify instances of each class and its overall predictive accuracy.

Code:

5B. Evaluating feed forward deep network for multiclass Classification using KFold cross-validation.

```
# !pip install scikeras
# !pip install np_utils

# loading libraries
import pandas
from keras.models import Sequential
from keras.layers import Dense
from scikeras.wrappers import KerasClassifier
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import LabelEncoder

# loading dataset
df = pandas.read_csv('/content/flowers.csv', header=None) #remove , header=None if dataset
contains column name
print(df)

# splitting dataset into input and output variables
X = df.iloc[:, 0:4].astype(float)
y = df.iloc[:, 4]
# print(X)
# print(y)

# encoding string output into numeric output
encoder = LabelEncoder()
encoder.fit(y)
encoded_y = encoder.transform(y)
```



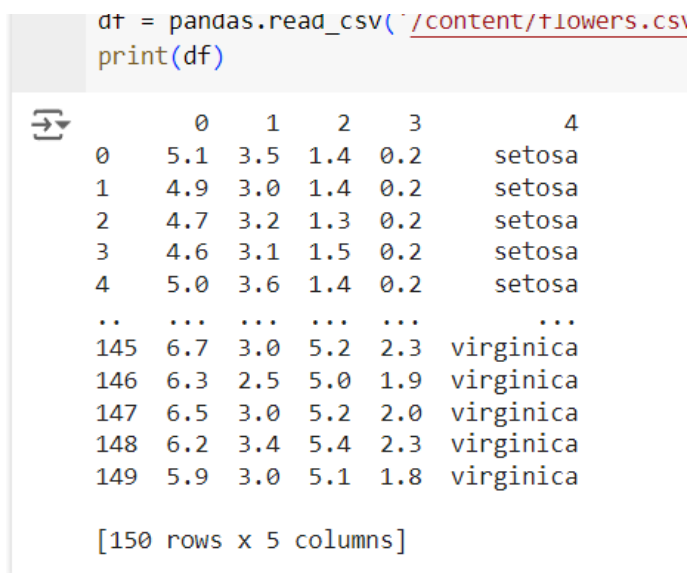
```
print(encoded_y)
dummy_Y = to_categorical(encoded_y)
print(dummy_Y)

def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(8, input_dim=4, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

estimator = baseline_model()
estimator.fit(X, dummy_Y, epochs=100, shuffle=True)
action = estimator.predict(X)
for i in range(25):
    print(dummy_Y[i])
    print('^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^')
for i in range(25):
    print(action[i])
```

Output:

```
df = pandas.read_csv('/content/flowers.csv')
print(df)
```



	0	1	2	3	4
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
...
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica

[150 rows x 5 columns]

```
dummy_Y = to_categorical(en
print(dummy_Y)
```



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

```
print(action[i])
```



```
Epoch 1/100
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning:
    super().__init__(activity_regularizer=activity_regularizer, **kwargs)
5/5 ————— 2s 5ms/step - accuracy: 0.3177 - loss: 3.0490
Epoch 2/100
5/5 ————— 0s 3ms/step - accuracy: 0.3268 - loss: 2.7846
Epoch 3/100
5/5 ————— 0s 3ms/step - accuracy: 0.3290 - loss: 2.6315
Epoch 4/100
5/5 ————— 0s 3ms/step - accuracy: 0.3242 - loss: 2.6313
Epoch 5/100
5/5 ————— 0s 3ms/step - accuracy: 0.3234 - loss: 2.5160
Epoch 6/100
5/5 ————— 0s 3ms/step - accuracy: 0.3051 - loss: 2.3749
Epoch 7/100
5/5 ————— 0s 3ms/step - accuracy: 0.3455 - loss: 2.0786
Epoch 8/100
5/5 ————— 0s 2ms/step - accuracy: 0.2908 - loss: 2.3395
Epoch 9/100
5/5 ————— 0s 3ms/step - accuracy: 0.3174 - loss: 2.1006
```

```
✓ [1. 0. 0.]  
s ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  
⇌ [0.37604138 0.3619987 0.26195988]  
[0.3978833 0.34762248 0.25449416]  
[0.38692015 0.3546491 0.25843075]  
[0.37794983 0.37557587 0.24647428]  
[0.36783838 0.3708288 0.26133272]  
[0.3411759 0.39997184 0.25885218]  
[0.36758184 0.38247693 0.24994132]  
[0.37296107 0.37235424 0.2546846 ]  
[0.3891353 0.36409342 0.2467713 ]  
[0.38633233 0.3591719 0.25449586]  
[0.36790323 0.36902478 0.26307192]  
[0.3614017 0.39187658 0.24672157]  
[0.39475307 0.3483042 0.25694266]  
[0.39766833 0.3379592 0.26437238]  
[0.3787997 0.3341694 0.28703088]  
[0.33936575 0.39567292 0.26496124]  
[0.3688776 0.36408493 0.26703748]  
[0.37630323 0.3656193 0.25807747]  
[0.35907134 0.38513625 0.25579235]  
[0.35434467 0.3903142 0.25534105]  
[0.3726926 0.37667748 0.2506298 ]  
[0.35796604 0.38656738 0.25546652]  
[0.3790055 0.3453873 0.27560708]  
[0.35096622 0.36856607 0.28046766]  
[0.33964843 0.41882122 0.24153031]
```

Learning:

1. **Importing and Installing Libraries:** The code begins with the installation of required libraries using pip, such as **scikeras** and **np_utils**, which are necessary for building and evaluating the neural network model.
2. **Data Loading and Preprocessing:** The dataset, presumably containing information about flowers, is loaded into a pandas DataFrame. The dataset is then split into input features (X) and target variable (y). Additionally, the target variable is encoded from string labels into numerical format using **LabelEncoder**, and one-hot encoding is applied using **to_categorical** to prepare it for multiclass classification.
3. **Neural Network Architecture:** A baseline feedforward neural network model is defined using Keras. It consists of an input layer, a hidden layer with 8 neurons and ReLU activation function, and an output layer with 3 neurons and softmax activation function for multiclass classification. The model is compiled with categorical cross-entropy loss function and Adam optimizer.
4. **Model Training:** The defined neural network model is trained on the input features (X) and the one-hot encoded target variable (dummy_Y) for 100 epochs. The **fit** function is used to train the model, with shuffling enabled to ensure randomness in the training data.
5. **Model Evaluation:** After training, the model's predictions are generated using the **predict** function on the input data (X). The predictions are then printed alongside the corresponding actual target values for comparison. This step allows for a qualitative assessment of the model's performance on the training data.
6. **Further Analysis:** The code also prints the actual target values and predicted values for the first 25 instances in the dataset, allowing for visual inspection of the model's behavior. This analysis can provide insights into the model's ability to correctly classify instances and identify any potential issues or patterns in its predictions.

Practical 6

6 a) Aim: Implementing regularization to avoid overfitting in binary classification.

Description:

Regularization is a technique used to prevent overfitting in binary classification models by adding a penalty to the loss function, discouraging the model from fitting the noise in the training data. Common regularization methods include L1 (Lasso) and L2 (Ridge) regularization, which add the absolute values or squared values of the weights to the loss function, respectively. Another effective method is dropout, where randomly selected neurons are ignored during training, preventing the model from becoming overly reliant on specific neurons. Additionally, early stopping can be used, which halts training once the model's performance on a validation set stops improving, preventing overfitting by limiting the number of training epochs. These techniques help improve the generalization capability of the model, ensuring better performance on unseen data.

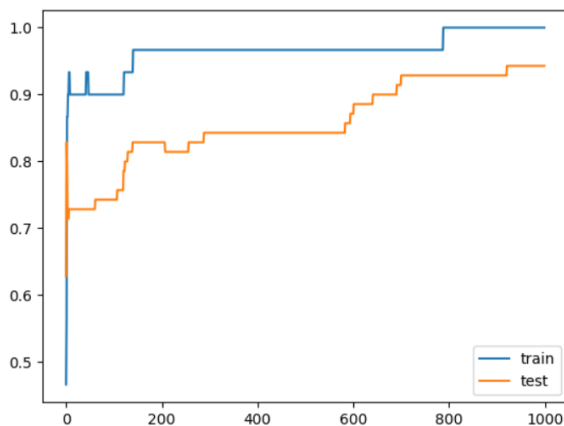
Code:

```
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=1000)
pyplot.plot(history.history['accuracy'],label='train')
pyplot.plot(history.history['val_accuracy'],label='test')
pyplot.legend()
pyplot.show()
pyplot.plot(history.history['accuracy'],label='train')
pyplot.plot(history.history['val_accuracy'],label='test')
pyplot.legend()
pyplot.show()
```

Output:

```
1/1 [=====] - 0s 68ms/step - loss: 0.0203 - accuracy: 1.0000 - val_loss: 0.2394 - val_accuracy: 0.9429
Epoch 999/1000
1/1 [=====] - 0s 61ms/step - loss: 0.0202 - accuracy: 1.0000 - val_loss: 0.2395 - val_accuracy: 0.9429
Epoch 1000/1000
1/1 [=====] - 0s 66ms/step - loss: 0.0201 - accuracy: 1.0000 - val_loss: 0.2397 - val_accuracy: 0.9429
```



6 b)

The above code and resultant graph demonstrate overfitting with accuracy of testing data less than accuracy of training data also the accuracy of testing data increases once and then start decreases gradually.

to solve this problem, we can use regularization Hence, we will add two lines in the above code as highlighted below to implement l2 regularization with $\alpha=0.001$

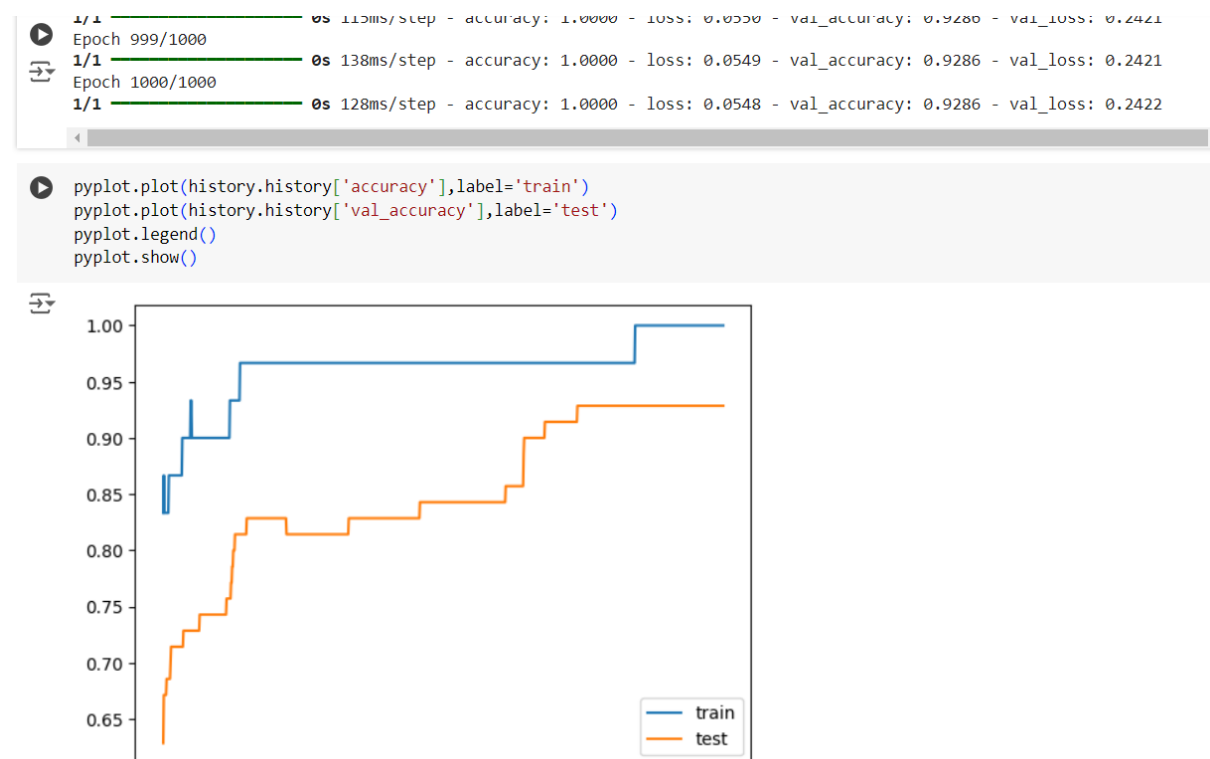
Code:

```
from matplotlib import pyplot
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=1000)
```

```
pyplot.plot(history.history['accuracy'],label='train')  
pyplot.plot(history.history['val_accuracy'],label='test')  
pyplot.legend()  
pyplot.show()
```

Output:



6 c)

By applying l1 and l2 regularizer we can observe the following changes in accuracy of both training and testing data. The changes in code are also highlighted.

Code:

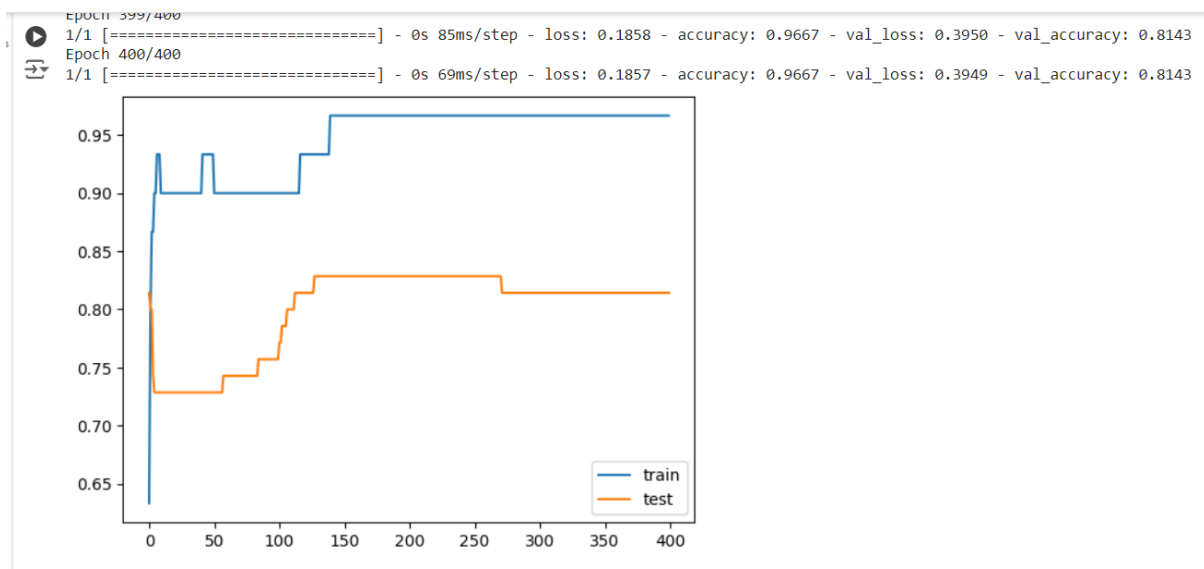
```
# !pip install pandas
# !pip install matplotlib
# !pip install keras
# !pip install tensorflow

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_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=l1_l2(l1=0.001,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=400)  
pyplot.plot(history.history['accuracy'],label='train')  
pyplot.plot(history.history['val_accuracy'],label='test')  
pyplot.legend()  
pyplot.show()
```

Output:



Learning:

This code demonstrates the creation and training of a neural network using the Keras library on a synthetic dataset generated by the **make_moons** function. The dataset is split into training and testing sets, with the neural network consisting of a single hidden layer with 500 neurons and a ReLU activation function, followed by an output layer with a sigmoid activation function for binary classification. The model is compiled using binary cross-entropy loss and the Adam optimizer. The training process is run for 1000 epochs, and the accuracy for both the training and validation sets is plotted to visualize the model's performance over time.

Practical 7

RNN for sequence analysis

7

Aim: Demonstrate recurrent neural network that learns to perform sequence analysis for stock price.

Description:

RNN

1. **Sequential Data Processing:** Recurrent Neural Networks (RNNs) are designed for sequential data processing, making them suitable for tasks like time series prediction, speech recognition, and natural language processing.
2. **Recurrent Connections:** RNNs utilize recurrent connections to preserve information about previous states, enabling them to capture temporal dependencies in the data.
3. **Variable-Length Inputs:** RNNs can handle inputs of variable lengths, making them versatile for tasks where input sequences may vary in length.
4. **Vanishing Gradient Problem:** RNNs may suffer from the vanishing gradient problem, where gradients diminish exponentially over time, affecting the learning of long-range dependencies.
5. **LSTM and GRU:** To address the vanishing gradient problem, variants like Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU) were developed, which incorporate gating mechanisms to regulate information flow.
6. **Bidirectional RNNs:** Bidirectional RNNs process sequences in both forward and backward directions, enhancing their ability to capture context from both past and future inputs.
7. **Applications:** RNNs find applications in various fields such as natural language processing for sentiment analysis, machine translation, and speech recognition, as well as in finance for stock price prediction and in biology for sequence analysis.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from keras.models import Sequential
from keras.layers import Dense, LSTM, Dropout
from sklearn.preprocessing import MinMaxScaler

# Read training dataset
dataset_train = pd.read_csv('Google_Stock_price_train.csv')
training_set = dataset_train.iloc[:, 1:2].values

# Scale the training set
sc = MinMaxScaler(feature_range=(0,1))
training_set_scaled = sc.fit_transform(training_set)

# Create X_train and Y_train
X_train = []
Y_train = []
for i in range(60, 1258):
    X_train.append(training_set_scaled[i-60:i, 0])
    Y_train.append(training_set_scaled[i, 0])
X_train, Y_train = np.array(X_train), np.array(Y_train)

# Reshape X_train for LSTM
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))

# Build the LSTM model
regressor = Sequential()
regressor.add(LSTM(units=50, return_sequences=True, input_shape=(X_train.shape[1], 1)))
regressor.add(Dropout(0.2))
regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.2))
regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.2))
regressor.add(LSTM(units=50))
regressor.add(Dropout(0.2))
regressor.add(Dense(units=1))
regressor.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
regressor.fit(X_train, Y_train, epochs=100, batch_size=32)

# Read test dataset
dataset_test = pd.read_csv('Google_Stock_price_Test.csv')
real_stock_price = dataset_test.iloc[:, 1:2].values
```

```

# Concatenate total dataset
dataset_total = pd.concat((dataset_train['Open'], dataset_test['Open']), axis=0)
inputs = dataset_total[len(dataset_total)-len(dataset_test)-60:].values
inputs = inputs.reshape(-1, 1)
inputs = sc.transform(inputs)

# Create X_test
X_test = []
for i in range(60, 80):
    X_test.append(inputs[i-60:i, 0])
X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))

# Predict stock prices
predicted_stock_price = regressor.predict(X_test)
predicted_stock_price = sc.inverse_transform(predicted_stock_price)

# Visualize results
plt.plot(real_stock_price, color='red', label='Real Google Stock Price')
plt.plot(predicted_stock_price, color='blue', label='Predicted Stock Price')
plt.xlabel('Time')
plt.ylabel('Google Stock Price')
plt.legend()
plt.show()

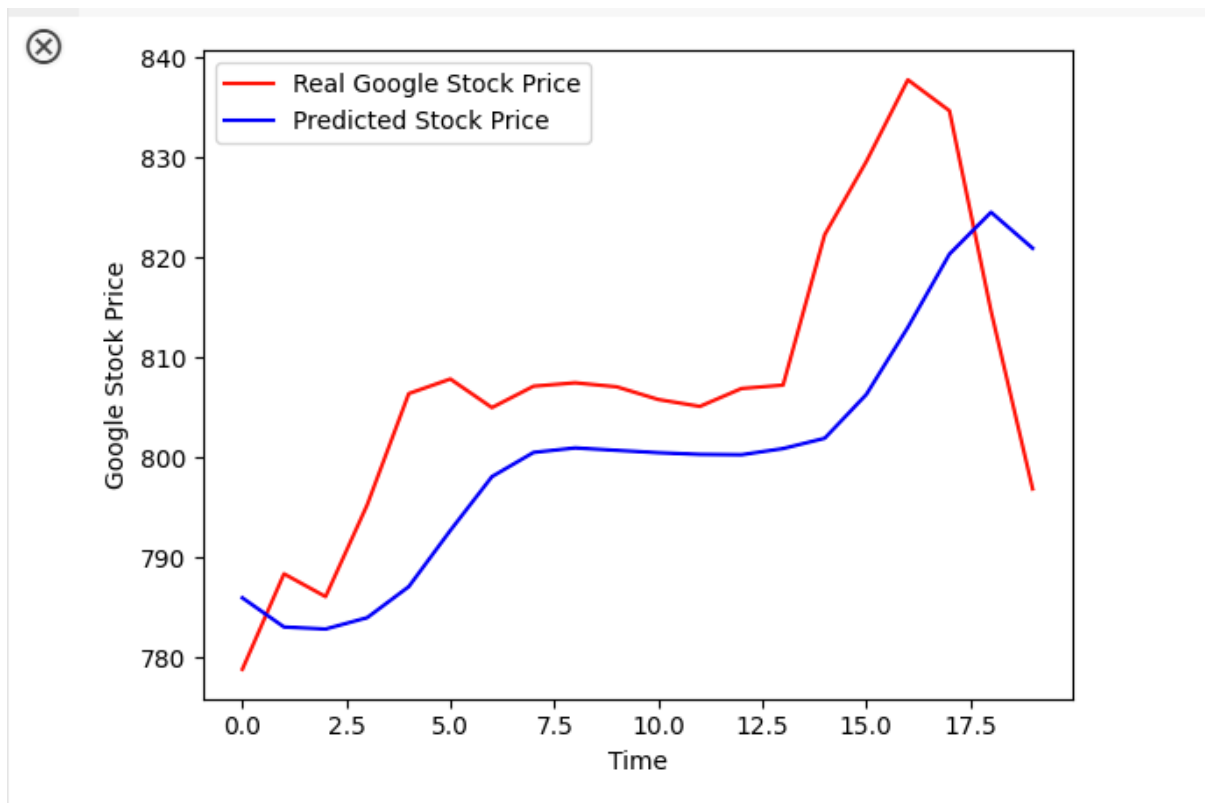
```

Output:

```

38/38 [-----] - 5s 120ms/step - loss: 0.0016
Epoch 95/100
38/38 [=====] - 5s 120ms/step - loss: 0.0016
Epoch 96/100
38/38 [=====] - 5s 136ms/step - loss: 0.0015
Epoch 97/100
38/38 [=====] - 5s 138ms/step - loss: 0.0015
Epoch 98/100
38/38 [=====] - 5s 120ms/step - loss: 0.0014
Epoch 99/100
38/38 [=====] - 6s 152ms/step - loss: 0.0013
Epoch 100/100
38/38 [=====] - 5s 122ms/step - loss: 0.0014

```



Learning:

Data preprocessing involves scaling the training dataset using MinMaxScaler.

The LSTM model architecture comprises four layers with dropout regularization.

Model compilation uses the Adam optimizer and mean squared error loss.

Training the model on the training set for 100 epochs with a batch size of 32.

Finally, the model predicts stock prices on a test dataset, followed by inverse scaling and visualization of results using Matplotlib.

Practical 8

Encoding and decoding using deep autoencoder.

8

Aim: Performing encoding and decoding of images using deep autoencoder.

Description:

Image Compression and Reconstruction with Deep Autoencoders

Deep autoencoders offer a powerful technique for encoding and decoding images. Here's a breakdown of the process in 5 points:

1. **Network Architecture:** A deep autoencoder consists of two main parts: an encoder and a decoder. The encoder progressively reduces the image's dimensionality, capturing its essential features in a compressed "latent space."
2. **Learning Compressive Representation:** During training, the encoder learns to represent the image in this lower-dimensional space. This compressed representation discards redundant information while retaining key details.
3. **Decoding for Reconstruction:** The decoder then takes this compressed representation and attempts to reconstruct the original image.
4. **Training and Loss Function:** The autoencoder is trained to minimize the difference between the original image and the reconstructed image. This ensures the encoder captures the most important information for faithful reconstruction.
5. **Applications:** Deep autoencoders have various applications in image processing. They can be used for image compression, denoising (removing noise from images), and anomaly detection (identifying unusual patterns in images).

Code:

8. Performing encoding and decoding of images using deep autoencoder.

```
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,))

# Retrieve 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:

```

print(X_train.shape)
print(X_test.shape)

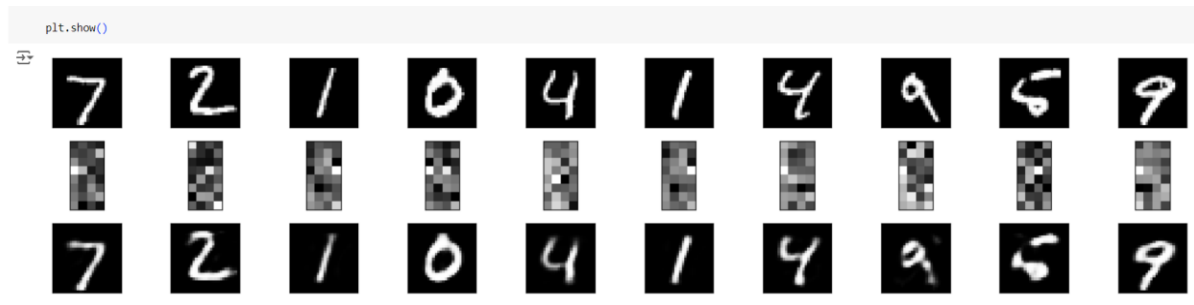
```

 Downloading data from <https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz>
 11490434/11490434 [=====] - 0s 0us/step
 (60000, 784)
 (10000, 784)

```

235/235 [=====] - 3s 14ms/step - loss: 0.0927 - val_loss: 0.0917
Epoch 48/50
235/235 [=====] - 4s 16ms/step - loss: 0.0926 - val_loss: 0.0917
Epoch 49/50
235/235 [=====] - 3s 12ms/step - loss: 0.0926 - val_loss: 0.0915
Epoch 50/50
235/235 [=====] - 2s 10ms/step - loss: 0.0926 - val_loss: 0.0915
313/313 [=====] - 1s 2ms/step
313/313 [=====] - 1s 2ms/step

```



Learning:

1. **Deep Autoencoder Architecture:** The code defines a deep autoencoder architecture using the Keras library, which consists of an encoder and a decoder. The encoder compresses the input images into a lower-dimensional latent space representation, while the decoder reconstructs the original images from this representation.
2. **Model Compilation and Training:** The autoencoder model is compiled with the Adam optimizer and binary cross-entropy loss function. It is then trained on the MNIST dataset, which consists of grayscale images of handwritten digits. Training is performed for 50 epochs with a batch size of 256, and the training and validation data are shuffled during training.
3. **Data Preprocessing:** The MNIST dataset is loaded and preprocessed. The pixel values of the images are scaled to the range [0, 1] by dividing by 255. Additionally, the shape of the input images is reshaped to a vector of length 784 before training.
4. **Encoder and Decoder Models:** Separate models for the encoder and decoder are created using the trained autoencoder model. These models allow for the encoding and decoding of images independently. The encoder model takes input images and outputs their encoded representations, while the decoder model takes encoded representations and reconstructs the original images.
5. **Visualization:** The code includes visualization of the original images, encoded representations, and reconstructed images for a sample of the test data. Matplotlib is used to display these images in a grid format, with each row showing the original image, its encoded representation, and the reconstructed image.

Practical 9

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

9

Aim: Implementation of convolutional neural network to predict numbers from number images

Description:

Learns Features: It uses convolutional layers to automatically learn features from images, like edges, shapes, and textures.

Stacked Layers: These convolutional layers are stacked together, allowing the network to learn increasingly complex features.

Classification or Detection: Finally, the network uses fully-connected layers to classify images (e.g., identifying digits) or detect objects within them.

Grid-like Approach: CNNs process images in a grid-like fashion using filters (kernels) that slide across the image. This helps capture spatial information and identify features regardless of their position.

Parameter Efficiency: Compared to fully-connected neural networks, CNNs have fewer parameters due to shared weights within filters. This reduces training complexity and helps prevent overfitting.

Wide Applications: Beyond image recognition, CNNs are used for tasks like video analysis, natural language processing (analyzing text structure), and even medical image analysis.

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)

# Reshape data for CNN (add channel dimension)
X_train = X_train.reshape(60000, 28, 28, 1)
X_test = X_test.reshape(10000, 28, 28, 1)

# One-hot encode labels
Y_train = to_categorical(Y_train)
Y_test = to_categorical(Y_test)

# Print an example of one-hot encoded label
print(Y_train[0])

# Define the model architecture
model = Sequential()
```

```
# Learn image features with convolutional layers
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())

# Add a dense layer with softmax activation for 10-class classification
model.add(Dense(10, activation='softmax'))

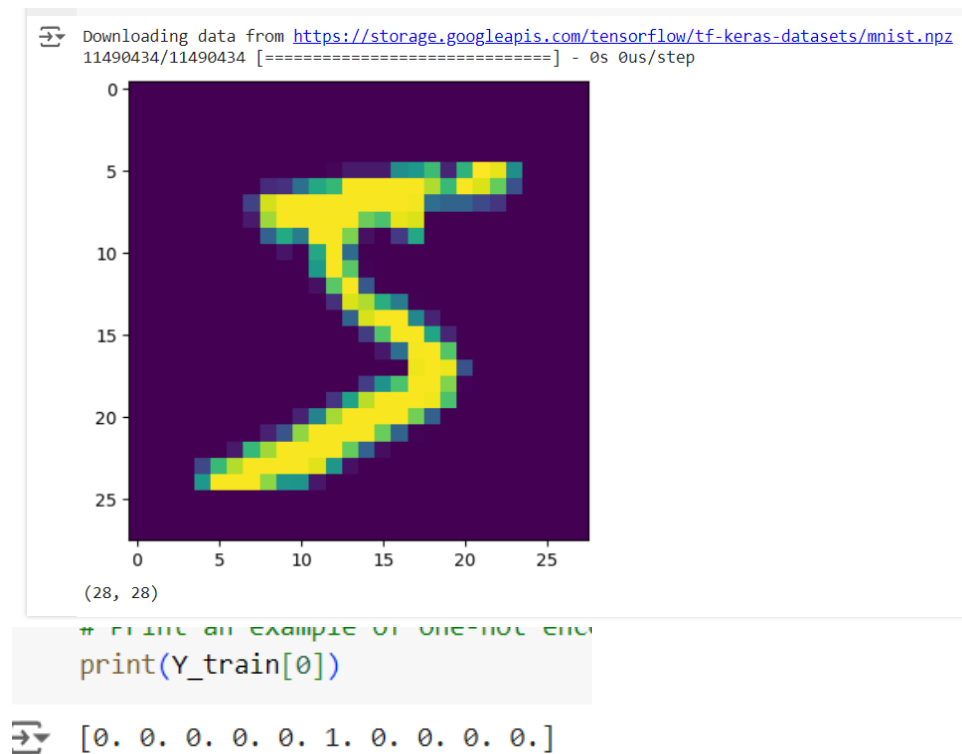
# Compile the model for training
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model with validation data
model.fit(X_train, Y_train, validation_data=(X_test, Y_test), epochs=3)

# Make predictions on the first 4 test images
predictions = model.predict(X_test[:4])
print(predictions) # Predicted probabilities for each class

# Print the actual labels for the first 4 test images
print(Y_test[:4]) # One-hot encoded labels
```

Output:



Learning:

- Imports: Grabs libraries for data, model building, and visualization.
- Data Prep: Loads MNIST dataset, reshapes for CNNs, and one-hot encodes labels.
- CNN Model: Builds a sequential model with convolutional layers for feature extraction and a dense layer for classification.
- Compilation: Sets up the model for training with optimizer, loss function, and accuracy metric.
- Training: Trains the model on data with validation for performance monitoring.

Practical 10

Denoising of images using autoencoder.

10

Aim: Denoising images using autoencoder.

Description:

autoencoder

1. **Unsupervised Learning:** Autoencoders are a type of neural network used for unsupervised learning. They don't require labeled data for training. Instead, they learn by trying to reconstruct the input data itself.
2. **Encoder-Decoder Structure:** An autoencoder has two main parts: an encoder and a decoder. The encoder compresses the input data into a lower-dimensional representation, often called the latent space. This captures the essential features of the data.
3. **Latent Space:** The latent space is the compressed version of the input data created by the encoder. It's like a bottleneck that forces the network to learn efficient representations.
4. **Reconstruction:** The decoder then takes this latent space representation and tries to rebuild the original input data as accurately as possible.
5. **Dimensionality Reduction:** By forcing the data through the bottleneck of the latent space, autoencoders can learn to represent the data in a more compact way. This can be useful for tasks like data compression or anomaly detection.
6. **Feature Extraction:** The latent space representation can also be used as a new set of features for the data. These features can be helpful for other machine learning tasks like classification or clustering.
7. **Variational Autoencoders (VAE):** A variation of the autoencoder is the Variational Autoencoder (VAE). VAEs introduce randomness into the latent space, allowing them to learn more complex representations of the data.

Code:

10. 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()

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')
```

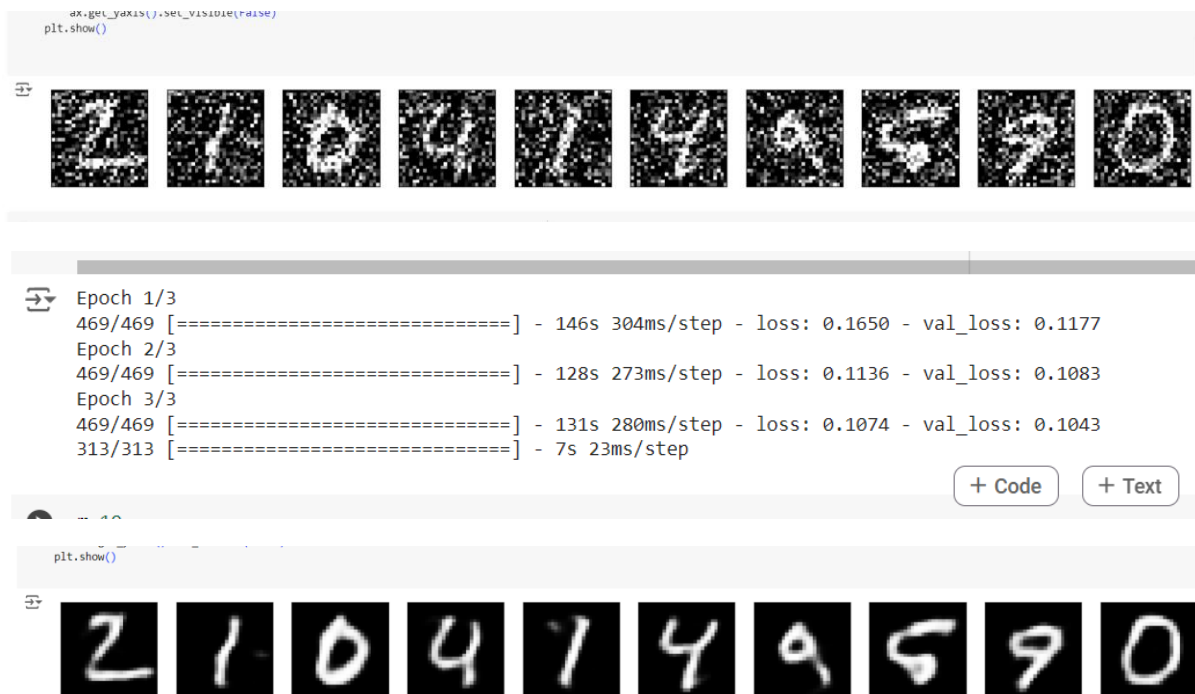


```
autoencoder.fit(X_train_noisy,X_train, epochs=3, batch_size=128, shuffle=True,
validation_data=(X_test_noisy,X_test),
callbacks=[TensorBoard(log_dir='/tmo/tb',histogram_freq=0,write_graph=False)])
```

```
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()
```

Output:



Learning:

1. **Denoising with Autoencoders:** This code demonstrates using an autoencoder for image denoising. It adds artificial noise to the MNIST dataset and trains the autoencoder to remove the noise while reconstructing the original images.
2. **Data Preprocessing:** The code preprocesses the MNIST data by converting it to float32 format, normalizing the pixel values between 0 and 1, and reshaping it to include the channel dimension (as MNIST images are grayscale).
3. **Convolutional Autoencoder Architecture:** The autoencoder uses a convolutional neural network (CNN) architecture with alternating convolutional and pooling layers in the encoder and upsampling layers in the decoder. This allows the network to learn spatial features of the images.
4. **Training and Validation:** The code trains the autoencoder on the noisy training data with the Adam optimizer and binary crossentropy loss function. It also uses a validation set to monitor performance on unseen noisy data.
5. **Visualization:** The code visualizes both the noisy test images and the denoised predictions from the autoencoder. This allows qualitative assessment of the model's ability to remove noise.