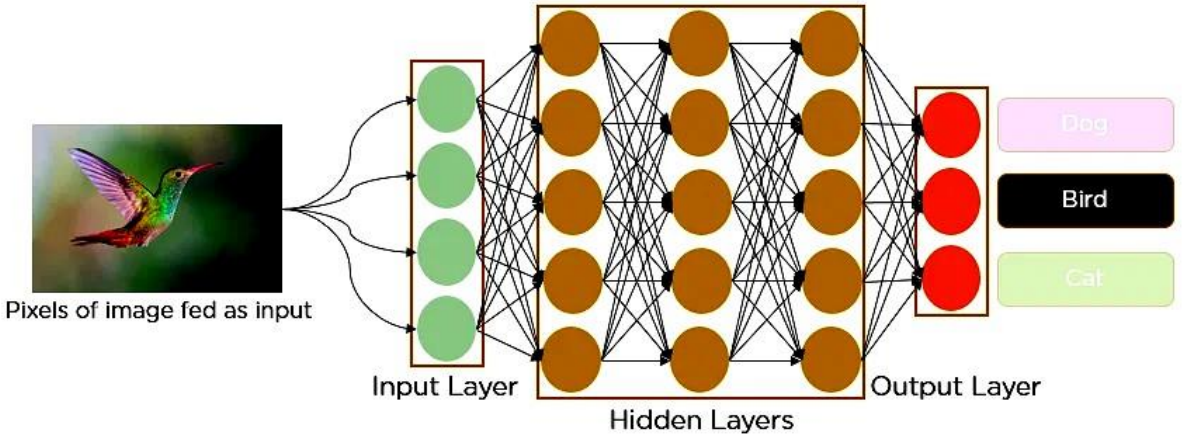
Week-1: Build a Convolution Neural Network for Image Recognition.

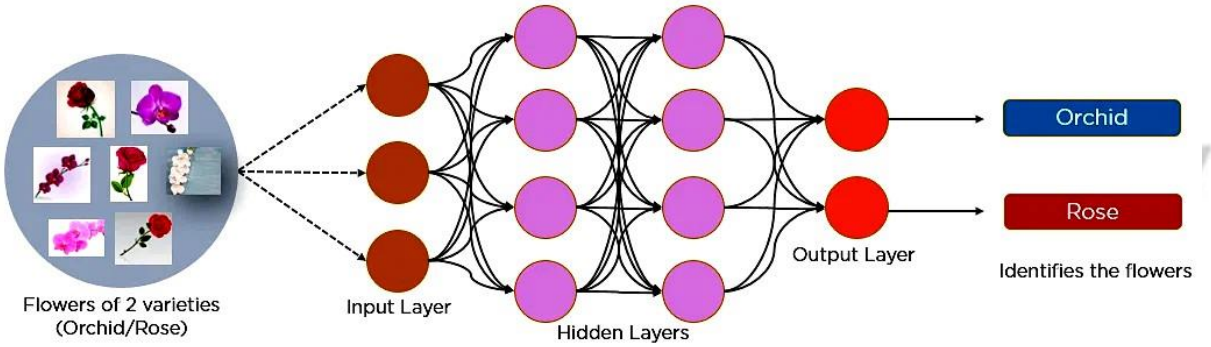
Aim: To build a Convolution Neural Network for Image Recognition.

Description: Convolutional Neural Network is one of the main categories to do image classification and image recognition in neural networks. Scene labeling, objects detections, and face recognition, etc., are some of the areas where convolutional neural networks are widely used.

Convolutional Neural Network to identify the image of a bird:



Neural Network that identifies two types of flowers: Orchid and Rose.



Stages involved in the implementation:

- Problem Identification
- Data Collection
- Data Preprocessing
- Select the Deep Learning Algorithm (CNN)
- Train and test the model
- Evaluate performance
- Development

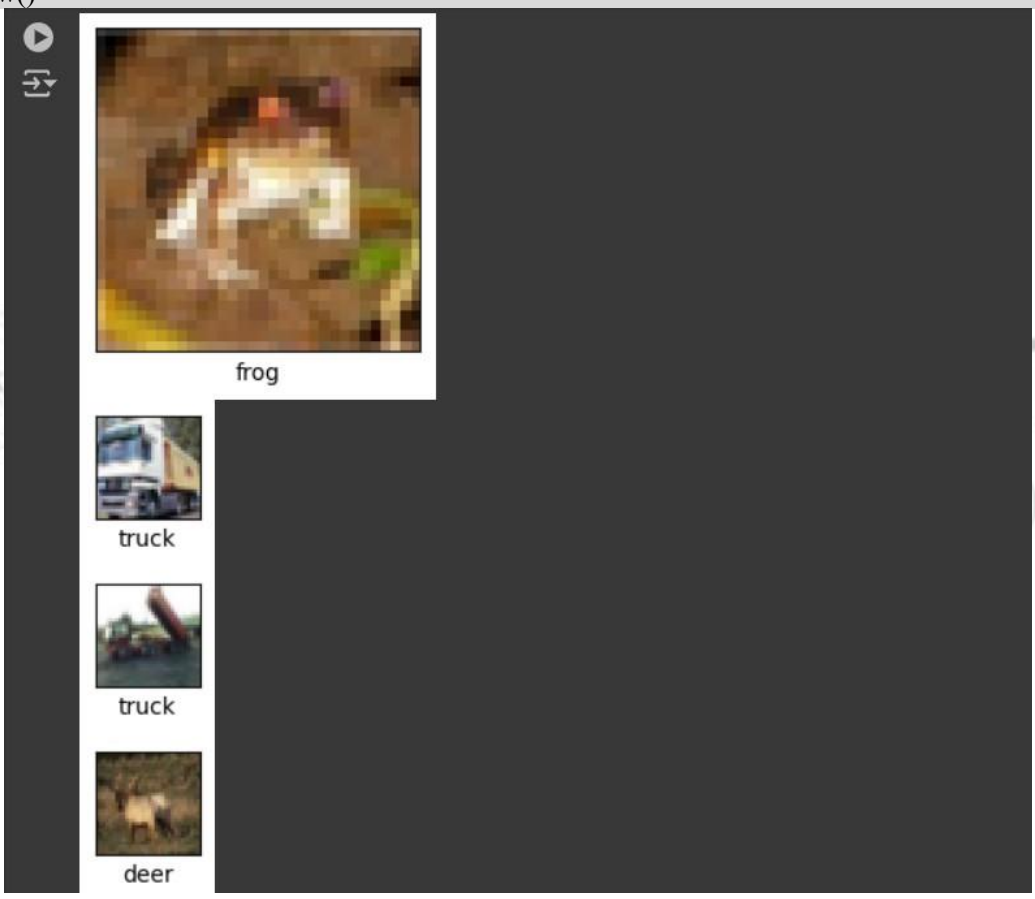
Steps:

- Import the package
- Download the dataset
- Normalization of Pixel Values
- Creating the CNN Model – Convolution – Pooling – Convolutional Layer and Pooling – Flattening – Full Connection
- Compile the Model

Program & Output:

```
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import datasets
(train_img, train_labels),(test_img, test_labels)=datasets.cifar10.load_data()
train_img, test_img=train_img/255.0,test_img/255.0
```

```
class_name=['airplane','automobile','bird','cat','deer','dog','frog','horse','ship','truck']
plt.figure(figsize=(15,15))
for i in range(10):
  plt.subplot(5,5,i+1)
  plt.xticks([])
  plt.yticks([])
  plt.imshow(train_img[i])
  plt.xlabel(class_name[train_labels[i][0]])
  plt.show()
```




```
#creating the CNN model
#initialization of model
from keras.models import Sequential
classifier=Sequential()
```

```
#step 1 - Convolution
from keras.layers import Conv2D
classifier.add(Conv2D(32,(3,3),input_shape=(32,32,3),activation='relu'))
#step 2 - pooling
from keras.layers import MaxPooling2D
classifier.add(MaxPooling2D(pool_size=(2,2)))
# the 32 feature maps from Conv2D output pass-through maxpooling of (2,2) size
```

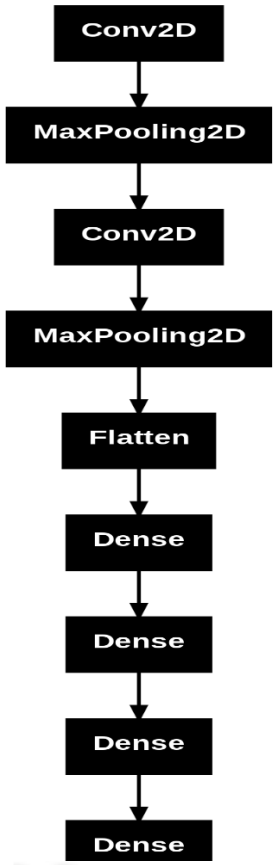
```
#Adding a second convolutional layer and Polling Layer
classifier.add(Conv2D (32,(3,3),activation='relu'))
classifier.add(MaxPooling2D(pool_size=(2,2)))

#step 3 - Flattening
from keras.layers import Flatten
classifier.add(Flatten())

#step 4 Full Connection
from keras.layers import Dense
classifier.add(Dense(units=64,activation='relu'))
#creates a fully connected neural network with 128 neurons
classifier.add(Dense(units=10,activation='softmax'))
#this fully connected layer should have number neurons as many as the class number
classifier.summary()
```

```
Model: "sequential_3"
```

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_3 (Conv2D)	(None, 13, 13, 32)	9,248
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 32)	0
flatten (Flatten)	(None, 1152)	0
dense (Dense)	(None, 64)	73,792
dense_1 (Dense)	(None, 10)	650
dense_2 (Dense)	(None, 64)	704
dense_3 (Dense)	(None, 10)	650

```
Total params: 85,940 (335.70 KB)
Trainable params: 85,940 (335.70 KB)
Non-trainable params: 0 (0.00 B)
```

```
# from tensorflow.keras.utils import plot_model
from keras.utils import plot_model
plot_model(classifier,to_file='cnn_mode.png')
```




```
#compile the model
#Compiling the CNN
classifier.compile(optimizer='adam'
                   ,loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
                   ,metrics=['accuracy'])
#optimizer: is an algorithm helps us to minimize (or maximize) an Objectivefunctionis.
history=classifier.fit(train_img,train_labels,epochs=10,validation_data=(test_img,test_labels))
```

```
Epoch 1/10
/usr/local/lib/python3.10/dist-packages/keras/src/backend/tensorflow/nn.py:609: UserWarning: "`sparse_categorical_crossentropy` received `from_logits=True`
  output, from_logits = _get_logits(
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 73s 44ms/step - accuracy: 0.2490 - loss: 1.9688 - val_accuracy: 0.4647 - val_loss: 1.4237
Epoch 2/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 81s 43ms/step - accuracy: 0.5000 - loss: 1.3633 - val_accuracy: 0.5622 - val_loss: 1.2039
Epoch 3/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 65s 42ms/step - accuracy: 0.5762 - loss: 1.1768 - val_accuracy: 0.5965 - val_loss: 1.1287
Epoch 4/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 66s 42ms/step - accuracy: 0.6164 - loss: 1.0731 - val_accuracy: 0.6071 - val_loss: 1.1042
Epoch 5/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 42ms/step - accuracy: 0.6445 - loss: 1.0020 - val_accuracy: 0.6293 - val_loss: 1.0608
Epoch 6/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 90s 47ms/step - accuracy: 0.6733 - loss: 0.9134 - val_accuracy: 0.6450 - val_loss: 1.0400
Epoch 7/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 74s 42ms/step - accuracy: 0.6921 - loss: 0.8671 - val_accuracy: 0.6560 - val_loss: 1.0062
Epoch 8/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 83s 43ms/step - accuracy: 0.7089 - loss: 0.8215 - val_accuracy: 0.6599 - val_loss: 1.0093
Epoch 9/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 42ms/step - accuracy: 0.7297 - loss: 0.7745 - val_accuracy: 0.6709 - val_loss: 0.9917
Epoch 10/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 43ms/step - accuracy: 0.7460 - loss: 0.7314 - val_accuracy: 0.6565 - val_loss: 1.0337
```

Result:

Thus, the Convolutional Neural Network for Image Recognition was executed successfully.

Week-2: Design Artificial Neural Networks for Identifying age group of an actor and classifying ar actor using Kaggle Dataset.

Aim: Design Artificial Neural Networks for Identifying age group of an actor and classifying an acto using Kaggle Dataset.

Description: Artificial Neural Networks contain artificial neurons which are called units . These units are arranged in a series of layers that together constitute the whole Artificial Neural Network in a system. A layer can have only a dozen units or millions of units as this depends on how the complex neural networks will be required to learn the hidden patterns in the dataset. Commonly, Artificial Neural Network has an input layer, an output layer as well as hidden layers. The input layer receives data from the outside world which the neural network needs to analyze or learn about. Then this data passes through one or multiple hidden layers that transform the input into data that is valuable for the output layer. Finally, the output layer provides an output in the form of a response of the Artificial Neural Networks.

Structure:

1. **Layers**: ANNs typically have an input layer, one or more hidden layers, and an output layer. Each layer consists of multiple neurons.
2. **Weights**: Connections between neurons have weights that adjust as learning proceeds, influencing the signal strength between neurons.
3. **Activation Function**: Each neuron applies an activation function to its input to determine its output. Common functions include sigmoid, ReLU, and tanh.

Learning Process: ANNs learn by adjusting weights through a process called backpropagation, which minimizes the difference between the predicted output and the actual output using a loss function.

Program:

```
import kagglehub
path = kagglehub.dataset_download("vishesh1412/celebrity-face-image-dataset")
print("Path to dataset files:", path)
```

```
Warning: Looks like you're using an outdated `kagglehub` version, please consider updating (latest version: 0.3.3)
Downloading from https://www.kaggle.com/api/v1/datasets/download/vishesh1412/celebrity-face-image-dataset?dataset_
100%|██████████| 52.9M/52.9M [00:00<00:00, 112MB/s] Extracting files...

Path to dataset files: /root/.cache/kagglehub/datasets/vishesh1412/celebrity-face-image-dataset/versions/1
```

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import layers, models
image_height = 128
image_width = 128
batch_size = 32
train_datagen = ImageDataGenerator(rescale=1./255, validation_split=0.2)
train_generator = train_datagen.flow_from_directory(
    '/root/.cache/kagglehub/datasets/vishesh1412/celebrity-face-image-dataset/versions/1',
    target_size=(image_height, image_width),
    batch_size=batch_size,
    class_mode='categorical',
    subset='training'
)
validation_generator = train_datagen.flow_from_directory(
    '/root/.cache/kagglehub/datasets/vishesh1412/celebrity-face-image-dataset/versions/1',
    target_size=(image_height, image_width),
    batch_size=batch_size,
    class_mode='categorical',
    subset='validation'
)
print("Classes found:", train_generator.class_indices)
num_classes = len(train_generator.class_indices)
model = models.Sequential([
```

```
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(image_height, image_width, 3)),
    layers.MaxPooling2D(pool_size=(2,  2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D(pool_size=(2,  2)),
    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(num_classes, activation='softmax' if num_classes > 1 else 'sigmoid')
])
loss_function = 'categorical_crossentropy' if num_classes > 1 else 'binary_crossentropy'
model.compile(optimizer='adam', loss=loss_function, metrics=['accuracy'])
model.fit(train_generator, validation_data=validation_generator, epochs=5)
model.save('trained_model_NEW_2_2_Dataset.h5')
```

```
Found 1440 images belonging to 1 classes.
Found 360 images belonging to 1 classes.
Classes found: {'Celebrity Faces Dataset': 0}
Epoch 1/5
45/45 ━━━━━━━━━━━━━━━━━━━━ 57s 1s/step - accuracy: 0.9040 - loss: 0.0715 - val_accuracy: 1.0000 - val_loss: 0.0000e+00
Epoch 2/5
45/45 ━━━━━━━━━━━━━━━━━━━━ 82s 1s/step - accuracy: 1.0000 - loss: 2.0133e-38 - val_accuracy: 1.0000 - val_loss: 0.0000e+00
Epoch 3/5
45/45 ━━━━━━━━━━━━━━━━━━━━ 81s 1s/step - accuracy: 1.0000 - loss: 0.0000e+00 - val_accuracy: 1.0000 - val_loss: 0.0000e+00
Epoch 4/5
45/45 ━━━━━━━━━━━━━━━━━━━━ 83s 1s/step - accuracy: 1.0000 - loss: 4.3371e-39 - val_accuracy: 1.0000 - val_loss: 0.0000e+00
Epoch 5/5
45/45 ━━━━━━━━━━━━━━━━━━━━ 54s 1s/step - accuracy: 1.0000 - loss: 3.1564e-39 - val_accuracy: 1.0000 - val_loss: 0.0000e+00
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file for
```

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import load_model
from PIL import Image
import requests
from io import BytesIO
from IPython.display import display
model = load_model('trained_model_NEW_2_2_Dataset.h5')
# Function to load an image from a URL
def load_image_from_url(url):
    response = requests.get(url)
    img = Image.open(BytesIO(response.content))
    return img
image_url='https://static.independent.co.uk/s3fs-public/thumbnails/image/2013/09/12/17/potter.jpg'
new_face = load_image_from_url(image_url)
new_face = new_face.resize((320, 256))
display(new_face)
new_face = new_face.resize((image_width, image_height))
new_face = np.array(new_face) / 255.0  # Normalize the image
new_face = np.expand_dims(new_face, axis=0)
predictions = model.predict(new_face)
predicted_age_group = np.argmax(predictions)
print("Predictions are:", predictions)
print("Predicted Age Group Index:", predicted_age_group)
age_mapping = {0: 'YOUNG', 1: 'MIDDLE', 2: 'OLD'}
predicted_age_group_label = age_mapping[predicted_age_group]
print("Predicted Age Group:", predicted_age_group_label)
```

Result:

Thus, the Artificial Neural Networks for Identifying and classifying an actor using Kaggle Dataset was executed successfully.

Week-3: Design a CNN for Image Recognition which includes hyperparameter tuning.

Aim: To design a CNN for Image Recognition which includes hyperparameter tuning.

Description: Hyperparameter tuning is the process of selecting the optimal values for a machine learning model's hyperparameters. Hyperparameters are settings that control the learning process of the model such as the learning rate, the number of neurons in a neural network, or the kernel size in a support vecto machine. The goal of hyperparameter tuning is to find the values that lead to the best performance on a given task.

Hyperparameters in Neural Networks

Neural networks have several essential hyperparameters that need to be adjusted, including:

Learning rate: This hyperparameter controls the step size taken by the optimizer during each iteration o training. Too small a learning rate can result in slow convergence, while too large a learning rate can lea to instability and divergence.

Epochs: This hyperparameter represents the number of times the entire training dataset is passed through the model during training. Increasing the number of epochs can improve the model's performance but may lead to overfitting if not done carefully.

Number of layers: This hyperparameter determines the depth of the model, which can have a significan impact on its complexity and learning ability.

Number of nodes per layer: This hyperparameter determines the width of the model, influencing its capacity to represent complex relationships in the data.

Architecture: This hyperparameter determines the overall structure of the neural network, including the number of layers, the number of neurons per layer, etc.

Activation function: This hyperparameter introduces non-linearity into the model, allowing it to learn complex decision boundaries. Common activation functions include sigmoid, tanh, and Rectified Linear Unit (ReLU).

Program:

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import cifar10
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import LearningRateScheduler

# Load and preprocess the CIFAR-10 dataset
(train_images, train_labels), (test_images, test_labels) = cifar10.load_data()
train_images, test_images = train_images / 255.0, test_images / 255.0
# Split data into training and validation sets
train_images, val_images, train_labels, val_labels = train_test_split(train_images, train_labels, test_size=0.1, random_state=42)

# Define a CNN architecture
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
# Define a learning rate schedule
def lr_schedule(epoch):
  initial_lr = 0.001
  if epoch >= 40:
    return initial_lr * 0.1
  return initial_lr
```

```
# Compile the model
model.compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['accuracy'])
# Create data augmentation generator
datagen=ImageDataGenerator(rotation_range=15,width_shift_range=0.1,height_shift_range=0.1,horizo
ntal_flip=True,fill_mode='nearest')
# Train the model with data augmentation and learning rate schedule
history=model.fit(datagen.flow(train_images,train_labels,batch_size=64),epochs=2,steps_per_epoch=le
n(train_images)//64,validation_data=(val_images,val_labels),callbacks=[LearningRateScheduler(lr_sche
dule)])
# Evaluate the model on the test set
test_loss, test_acc = model.evaluate(test_images, test_labels)
print('Test accuracy:', test_acc)
```

```
Epoch 1/2
703/703 ━━━━━━━━━━━━━━━━━━━━ 93s 130ms/step - accuracy: 0.4891 - loss: 1.4146 - val_accuracy: 0.5490 - val_loss: 1.2480 - learning_rate: 0.0010
Epoch 2/2
  1/703 ━━━━━━━━━━━━━━━━━━━━ 52s 75ms/step - accuracy: 0.5469 - loss: 1.3576/usr/lib/python3.10/contextlib.py:153: UserWarning: Your input ran o
  self.gen.throw(typ, value, traceback)
703/703 ━━━━━━━━━━━━━━━━━━━━ 5s 7ms/step - accuracy: 0.5469 - loss: 1.3576 - val_accuracy: 0.5492 - val_loss: 1.2581 - learning_rate: 0.0010
313/313 ━━━━━━━━━━━━━━━━━━━━ 5s 15ms/step - accuracy: 0.5567 - loss: 1.2271
Test accuracy: 0.5534999966621399
```

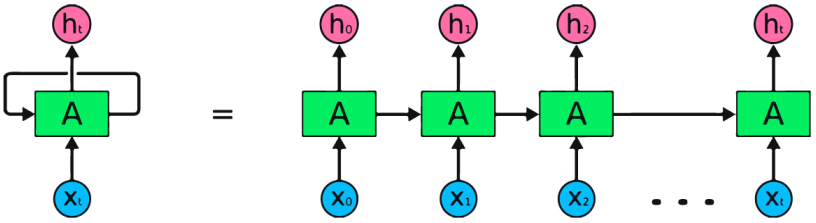
Result:

Thus, CNN for image recognition includes hyperparameter tunning was executed successfully.

Week-4: Implement a Recurrence Neural Network for Predicting Sequential Data.

Aim: To implement a Recurrence Neural Network for Predicting Sequential Data.

Description: A recurrent neural network (RNN) is the type of artificial neural network (ANN) that is used in Apple's Siri and Google's voice search. RNN remembers past inputs due to an internal memory which is useful for predicting stock prices, generating text, transcriptions, and machine translation. In the traditional neural network, the inputs and the outputs are independent of each other, whereas the output in RNN is dependent on prior elementals within the sequence. RNN shares the same weights within each layer of the network and during gradient descent, the weights and basis are adjusted individually to reduce the loss.



How Recurrent Neural Networks Work?

In RNN, the information cycles through the loop, so the output is determined by the current input and previously received inputs. The input layer X processes the initial input and passes it to the middle layer A. The middle layer consists of multiple hidden layers, each with its activation functions, weights, and biases. These parameters are standardized across the hidden layer so that instead of creating multiple hidden layers, it will create one and loop it over.

Instead of using traditional backpropagation, recurrent neural networks use backpropagation through time (BPTT) algorithms to determine the gradient. In backpropagation, the model adjusts the parameter by calculating errors from the output to the input layer. BPTT sums the error at each time step as RNN shares parameters across each layer.

RNN Advanced Architectures

The simple RNN repeating modules have a basic structure with a single tanh layer. RNN simple structure suffers from short memory, where it struggles to retain previous time step information in larger sequential data. These problems can easily be solved by long short term memory (LSTM) and gated recurrent unit (GRU), as they are capable of remembering long periods of information.

Long Short Term Memory (LSTM)

The Long Short Term Memory (LSTM) is the advanced type of RNN, which was designed to prevent both decaying and exploding gradient problems. Instead of having a single layer of tanh, LSTM has four interacting layers that communicate with each other. This four-layered structure helps LSTM retain long term memory and can be used in several sequential problems including machine translation, speech synthesis, speech and handwriting recognition.

Program & Output:

MasterCard Stock Price Prediction Using LSTM:

https://www.kaggle.com/datasets/kalilurrahman/mastercard-stock-data-latest-and-updated

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout, GRU, Bidirectional
from tensorflow.keras.optimizers import SGD
from tensorflow.random import set_seed
```

```
set_seed(455)
np.random.seed(455)
# Data Analysis
dataset=pd.read_csv("/content/Mastercard_stock_history.csv",index_col="Date",parse_dates=["Date"]
```

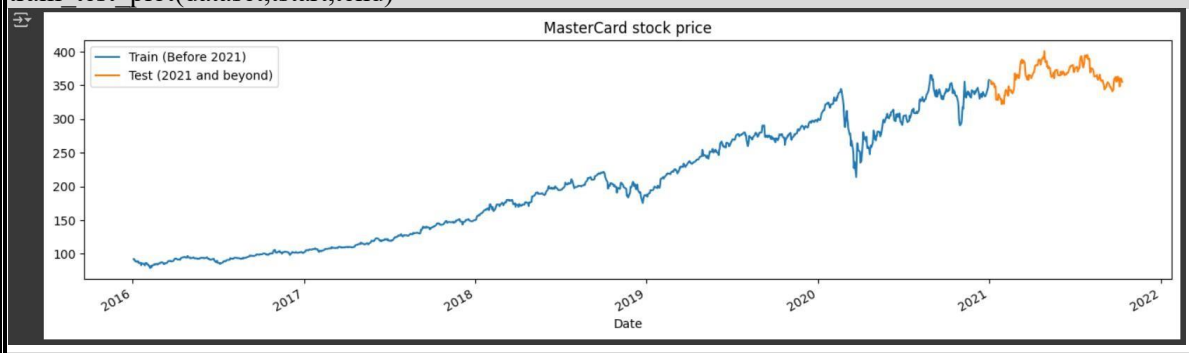
```
).drop(["Dividends", "Stock Splits"], axis=1)
print(dataset.head())
```

Date	Open	High	Low	Close	Volume
2006-05-25	3.748967	4.283869	3.739664	4.279217	395343000
2006-05-26	4.307126	4.348058	4.103398	4.179680	103044000
2006-05-30	4.183400	4.184330	3.986184	4.093164	49898000
2006-05-31	4.125723	4.219679	4.125723	4.180608	30002000
2006-06-01	4.179678	4.474572	4.176887	4.419686	62344000

```
print(dataset.describe())
```

	Open	High	Low	Close	Volume
count	3872.000000	3872.000000	3872.000000	3872.000000	3.872000e+03
mean	104.896814	105.956054	103.769349	104.882714	1.232250e+07
std	106.245511	107.303589	105.050064	106.168693	1.759665e+07
min	3.748967	4.102467	3.739664	4.083861	6.411000e+05
25%	22.347203	22.637997	22.034458	22.300391	3.529475e+06
50%	70.810079	71.375896	70.224002	70.856083	5.891750e+06
75%	147.688448	148.645373	146.822013	147.688438	1.319775e+07
max	392.653890	400.521479	389.747812	394.685730	3.953430e+08

```
tstart = 2016
tend = 2020
def train_test_plot(dataset, tstart, tend):
  dataset.loc[f"{tstart}":f"{tend}", "High"].plot(figsize=(16, 4), legend=True)
  dataset.loc[f"{tend+1}":, "High"].plot(figsize=(16, 4), legend=True)
  plt.legend([f"Train (Before {tend+1})", f"Test ({tend+1} and beyond)"])
  plt.title("MasterCard stock price")
  plt.show()
train_test_plot(dataset,tstart,tend)
```




```
# Data Preprocessing
def train_test_split(dataset, tstart, tend):
  train = dataset.loc[f"{tstart}":f"{tend}", "High"].values
  test = dataset.loc[f"{tend+1}":, "High"].values
  return train, test
training_set, test_set = train_test_split(dataset, tstart, tend)


sc = MinMaxScaler(feature_range=(0, 1))
training_set = training_set.reshape(-1, 1)
training_set_scaled = sc.fit_transform(training_set)
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    end_ix = i + n_steps
    if end_ix > len(sequence) - 1:
      break
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
```

```
    y.append(seq_y)
  return np.array(X), np.array(y)
n_steps = 60
features = 1
X_train, y_train = split_sequence(training_set_scaled, n_steps)
# Reshaping X_train for model
X_train = X_train.reshape(X_train.shape[0],X_train.shape[1],features)
```

```
# The LSTM architecture
model_lstm = Sequential()
model_lstm.add(LSTM(units=125, activation="tanh", input_shape=(n_steps, features)))
model_lstm.add(Dense(units=1))
model_lstm.compile(optimizer="RMSprop", loss="mse")
model_lstm.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 125)	63,500
dense (Dense)	(None, 1)	126

```
Total params: 63,626 (248.54 KB)
Trainable params: 63,626 (248.54 KB)
Non-trainable params: 0 (0.00 B)
```

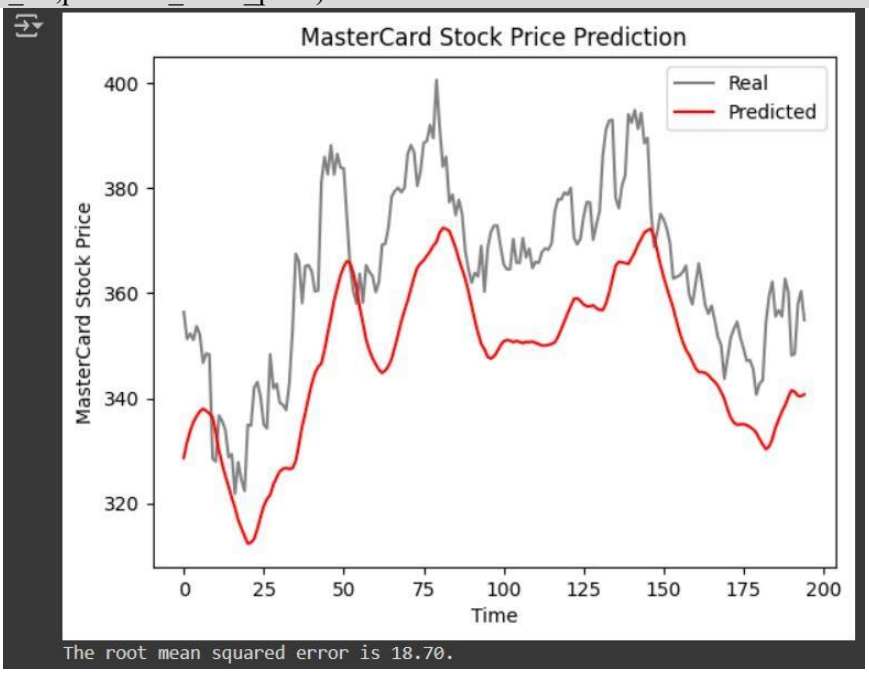
```
model_lstm.fit(X_train, y_train, epochs=5, batch_size=32)
```

```
Epoch 1/5
38/38 ━━━━━━━━━━━━━━━━━━━━ 2s 59ms/step - loss: 0.0022
Epoch 2/5
38/38 ━━━━━━━━━━━━━━━━━━━━ 3s 65ms/step - loss: 0.0015
Epoch 3/5
38/38 ━━━━━━━━━━━━━━━━━━━━ 2s 60ms/step - loss: 9.4287e-04
Epoch 4/5
38/38 ━━━━━━━━━━━━━━━━━━━━ 4s 93ms/step - loss: 8.3298e-04
Epoch 5/5
38/38 ━━━━━━━━━━━━━━━━━━━━ 4s 57ms/step - loss: 8.3966e-04
<keras.src.callbacks.history.History at 0x7c6808a24ac0>
```

```
dataset_total = dataset.loc[:,"High"]
inputs = dataset_total[len(dataset_total) - len(test_set) - n_steps :].values
inputs = inputs.reshape(-1, 1)
inputs = sc.transform(inputs)
X_test, y_test = split_sequence(inputs, n_steps)
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], features)
predicted_stock_price = model_lstm.predict(X_test)
predicted_stock_price = sc.inverse_transform(predicted_stock_price)
```

```
def plot_predictions(test, predicted):
  plt.plot(test, color="gray", label="Real")
  plt.plot(predicted, color="red", label="Predicted")
  plt.title("MasterCard Stock Price Prediction")
  plt.xlabel("Time")
  plt.ylabel("MasterCard Stock Price")
  plt.legend()
  plt.show()
def return_rmse(test, predicted):
  rmse = np.sqrt(mean_squared_error(test, predicted))
  print("The root mean squared error is {:.2f}.".format(rmse))
plot_predictions(test_set,predicted_stock_price)
```

```
return_rmse(test_set,predicted_stock_price)
```



Result:

Thus, the recurrence neural network for MasterCard Stock Price Prediction was executed successfully.

Week-5: Implement Multi-Layer Perceptron algorithm for Image denoising hyperparameter tuning.

Aim: To implement Multi-Layer Perceptron algorithm for Image denoising hyperparameter tuning.

Description: It is fully connected dense layers, which transform any input dimension to the desired dimension. A multi-layer perception is a neural network that has multiple layers. A multi-layer perceptron has one input layer and for each input, there is one neuron (or node), it has one output layer with a single node for each output and it can have any number of hidden layers and each hidden layer can have any number of nodes.

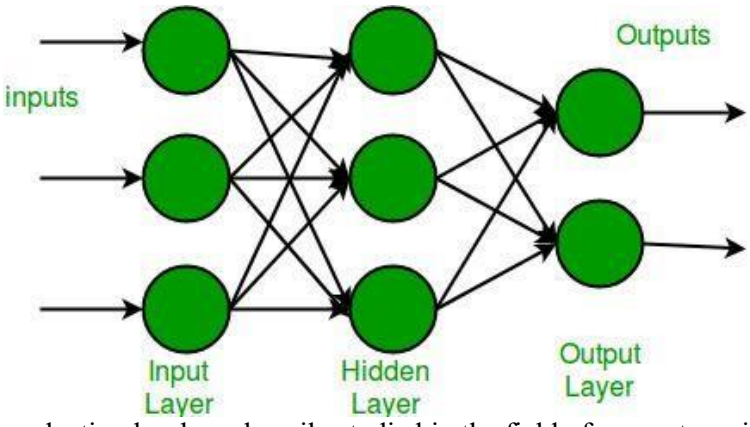


Image Noise: Noise reduction has been heavily studied in the field of computer science. Any sensor tha captures a signal is inherently prone to noise, where noise is an unwanted, unpredictable change in the signal. Image noise manifests itself as changes in pixel values. Noise reduction can be seen as a problem of mapping noisy data to a noise-free variant.

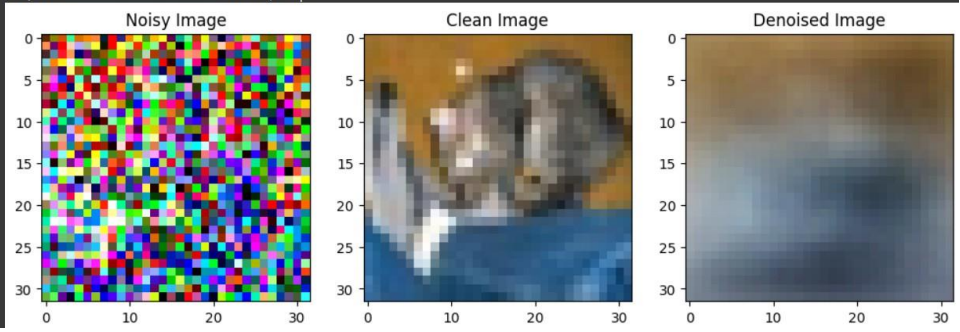
Program & Output:

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
def add_noise(images):
    noise_factor = 0.5
    noisy_images = images + noise_factor * np.random.randn(*images.shape)
    return np.clip(noisy_images, 0., 1.)
(x_train, _), (x_test, _) = keras.datasets.cifar10.load_data()
x_train, x_test = x_train.astype('float32') / 255.0, x_test.astype('float32') / 255.0
x_train_noisy = add_noise(x_train).reshape(-1, 32*32*3)
x_test_noisy = add_noise(x_test).reshape(-1, 32*32*3)
x_train = x_train.reshape(-1, 32*32*3)
x_test = x_test.reshape(-1, 32*32*3)
def create_model(hidden_units=128, activation='relu'):
    model = keras.Sequential([
        layers.Input(shape=(32*32*3,)),
        layers.Dense(hidden_units, activation=activation),
        layers.Dense(hidden_units, activation=activation),
        layers.Dense(32*32*3, activation='sigmoid')
    ])
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model
model = create_model()
model.fit(x_train_noisy, x_train, epochs=10, batch_size=256, validation_split=0.2)
test_loss = model.evaluate(x_test_noisy, x_test)
print("Test Loss:", test_loss)
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 2s 0us/step
Epoch 1/10
157/157 ━━━━━━━━━━━━━━━━━━━━ 8s 47ms/step - loss: 0.0530 - val_loss: 0.0363
Epoch 2/10
157/157 ━━━━━━━━━━━━━━━━━━━━ 6s 36ms/step - loss: 0.0346 - val_loss: 0.0312
Epoch 3/10
157/157 ━━━━━━━━━━━━━━━━━━━━ 10s 35ms/step - loss: 0.0292 - val_loss: 0.0267
Epoch 4/10
157/157 ━━━━━━━━━━━━━━━━━━━━ 7s 44ms/step - loss: 0.0250 - val_loss: 0.0228
Epoch 5/10
157/157 ━━━━━━━━━━━━━━━━━━━━ 6s 36ms/step - loss: 0.0221 - val_loss: 0.0211
Epoch 6/10
157/157 ━━━━━━━━━━━━━━━━━━━━ 11s 44ms/step - loss: 0.0207 - val_loss: 0.0199
Epoch 7/10
157/157 ━━━━━━━━━━━━━━━━━━━━ 10s 46ms/step - loss: 0.0196 - val_loss: 0.0192
Epoch 8/10
157/157 ━━━━━━━━━━━━━━━━━━━━ 6s 36ms/step - loss: 0.0190 - val_loss: 0.0189
Epoch 9/10
157/157 ━━━━━━━━━━━━━━━━━━━━ 6s 40ms/step - loss: 0.0184 - val_loss: 0.0184
Epoch 10/10
157/157 ━━━━━━━━━━━━━━━━━━━━ 9s 34ms/step - loss: 0.0182 - val_loss: 0.0182
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0184
Test Loss: 0.018378207460045815
```

```
import matplotlib.pyplot as plt
# Denoising and Visualization
def display_images(noisy, clean, denoised, index=0):
    plt.figure(figsize=(12, 4))
    plt.subplot(1, 3, 1)
    plt.title("Noisy Image")
    plt.imshow(noisy[index].reshape(32, 32, 3))
    plt.subplot(1, 3, 2)
    plt.title("Clean Image")
    plt.imshow(clean[index].reshape(32, 32, 3))
    plt.subplot(1, 3, 3)
    plt.title("Denoised Image")
    plt.imshow(denoised[index].reshape(32, 32, 3))
    plt.show()
denoised_images = model.predict(x_test_noisy)
display_images(x_test_noisy, x_test, denoised_images)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step
```



Result: Thus, the Multilayer perceptron algorithm using kaggle dataset.

Week-6: Implement Object Detection Using YOLO.

Aim: To implement Object Detection Using YOLO.

Description: Object Detection is a task of computer vision that helps to detect the objects in the image o video frame. It helps to recognize objects count the occurrences of them to keep records, etc. The objective of object detection is to identify and annotate each of the objects present in the media.

YOLO (You Only Look Once) is a state-of-the-art model to detect objects in an image or a video very precisely and accurately with very high accuracy. It deals with localizing a region of interest within an image and classifying this region like a typical image classifier. One image can include several regions o interest pointing to different objects. This makes object detection a more advanced problem of image classification.

Steps:

- Implement YOLO V8 on Google Colab
- Runtime -> GPU [Change Hardware Accelerator]
- Download Ultralytics
- Use github.com/ultralytics/ultralytics – to use a pretrained model yolov8m
- Upload cars.mp4
- Results has been saved to runs/detect/predict/cars.mp4

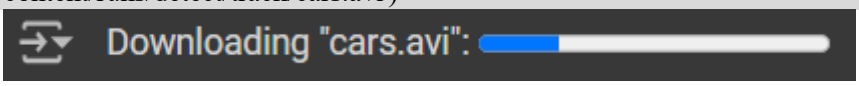
Program:

```
# Step 1: Install Ultralytics YOLO
!pip install ultralytics
# Step 2: Import Required Libraries
import cv2
from ultralytics import YOLO
# Step 3: Load the Pretrained YOLOv8 Model
model = YOLO('yolov8m.pt')
# Step 4: Upload Video
from google.colab import files
uploaded = files.upload()
# Assuming the uploaded video file is named 'cars.mp4'
video_path = 'cars.mp4'
# Step 5: Run YOLOv8 on the Uploaded Video
results = model.track(video_path, show=True, save=True)
# Results will be saved to runs/detect/predict/cars.mp4
```

```
Requirement already satisfied: ultralytics in /usr/local/lib/python3.10/dist-packages (8.3.18)
Requirement already satisfied: numpy>=1.23.0 in /usr/local/lib/python3.10/dist-packages (from ultralytics) (1.26.4)
Requirement already satisfied: matplotlib>=3.3.0 in /usr/local/lib/python3.10/dist-packages (from ultralytics) (3.7.1)
Requirement already satisfied: opencv-python>=4.6.0 in /usr/local/lib/python3.10/dist-packages (from ultralytics) (4.10.0.84)
Requirement already satisfied: pillow>=7.1.2 in /usr/local/lib/python3.10/dist-packages (from ultralytics) (10.4.0)
Requirement already satisfied: pyyaml>=5.3.1 in /usr/local/lib/python3.10/dist-packages (from ultralytics) (6.0.2)
Requirement already satisfied: requests>=2.23.0 in /usr/local/lib/python3.10/dist-packages (from ultralytics) (2.32.3)
Requirement already satisfied: scipy>=1.4.1 in /usr/local/lib/python3.10/dist-packages (from ultralytics) (1.13.1)
Requirement already satisfied: torch>=1.8.0 in /usr/local/lib/python3.10/dist-packages (from ultralytics) (2.4.1+cu121)
Requirement already satisfied: torchvision>=0.9.0 in /usr/local/lib/python3.10/dist-packages (from ultralytics) (0.19.1+cu121)
Requirement already satisfied: tqdm>=4.64.0 in /usr/local/lib/python3.10/dist-packages (from ultralytics) (4.66.5)
Requirement already satisfied: psutil in /usr/local/lib/python3.10/dist-packages (from ultralytics) (5.9.5)
Choose Files cars.mp4
• cars.mp4(video/mp4) - 6817246 bytes, last modified: 10/21/2024 - 100% done
Saving cars.mp4 to cars (1).mp4
```

```
video 1/1 (frame 1/315) /content/cars.mp4: 384x640 1 person, 909.3ms
video 1/1 (frame 2/315) /content/cars.mp4: 384x640 1 person, 946.0ms
video 1/1 (frame 3/315) /content/cars.mp4: 384x640 1 person, 1 cell phone, 899.8ms
video 1/1 (frame 4/315) /content/cars.mp4: 384x640 1 person, 1 cell phone, 897.9ms
video 1/1 (frame 5/315) /content/cars.mp4: 384x640 1 person, 1 cell phone, 963.2ms
video 1/1 (frame 6/315) /content/cars.mp4: 384x640 1 person, 1 cell phone, 1423.3ms
video 1/1 (frame 7/315) /content/cars.mp4: 384x640 1 person, 1 cell phone, 1493.4ms
video 1/1 (frame 8/315) /content/cars.mp4: 384x640 1 person, 1 cell phone, 899.3ms
video 1/1 (frame 9/315) /content/cars.mp4: 384x640 1 person, 1 cell phone, 904.9ms
video 1/1 (frame 10/315) /content/cars.mp4: 384x640 1 person, 1 cell phone, 896.4ms
Speed: 4.9ms preprocess, 1025.5ms inference, 1.3ms postprocess per image at shape (1, 3, 384, 640)
Results saved to runs/detect/track2
```

```
# After processing, you can download the resulting video by running:
from google.colab import files
files.download('/content/runs/detect/track/cars.avi')
```



Output:

Result:

Thus, the Object Detection using YOLO was executed successfully.

Week-7: Design a Deep learning Network for Robust Bi-Tempered Logistic Loss.

Aim: To design a Deep learning Network for Robust Bi-Tempered Logistic Loss.

Description: Robust Bi-Tempered Logistic Loss is a loss function designed to improve the performance of binary classification models, especially in the presence of outliers or imbalanced datasets. It modifies the traditional logistic loss to provide robustness against misclassified examples and outliers.

Key Concepts:

1. Binary Classification: The loss function is used in scenarios where the task is to classify data points into one of two categories (e.g., 0 or 1).
2. Bi-Tempered: The term "bi-tempered" refers to the dual scaling of the loss, which can be adjusted by two parameters (α and β):
    - α (Alpha): Controls the sensitivity of the loss to outliers. Higher values reduce the impact of misclassified points.
    - β (Beta): Adjusts the scaling of the predicted probabilities, affecting how the loss behaves at different confidence levels.
3. Logistic Loss: This is a standard loss function for binary classification, defined as:

$$\text{Loss} = -y \log(p) - (1-y) \log(1-p)$$

   where y is the true label and p is the predicted probability.
4. Robustness: The bi-tempered logistic loss modifies the traditional logistic loss to mitigate the influence of outliers. By adjusting the parameters, the loss can focus more on correctly classifying well-behaved data points and less on the noisy ones.

Applications:

- Imbalanced Datasets: It's particularly useful in cases where one class is underrepresented.
- Noisy Data: In datasets with significant noise or outliers, this loss helps to reduce their adverse effects on model training.

Program:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt
import kagglehub
import pandas as pd
path=kagglehub.dataset_download("naveengowda16/logistic-regression-heart-disease-prediction")
df=pd.read_csv(path+"/framingham_heart_disease.csv")

# Impute missing values using SimpleImputer for numeric values (using mean for numeric)
imputer = SimpleImputer(strategy='mean')
df[['education', 'cigsPerDay', 'totChol', 'BMI', 'heartRate', 'glucose']] = imputer.fit_transform(
    df[['education', 'cigsPerDay', 'totChol', 'BMI', 'heartRate', 'glucose']]
)
# For categorical column 'BPMeds', use the mode
bpm_imputer = SimpleImputer(strategy='most_frequent')
df['BPMeds'] = bpm_imputer.fit_transform(df[['BPMeds']])

# Prepare the features (X) and target (y)
X = df.drop(columns=['TenYearCHD']).values
y = df['TenYearCHD'].values

# Scale the features
scaler = StandardScaler()
```

```
X_scaled = scaler.fit_transform(X)
X_train, X_temp, y_train, y_temp = train_test_split(X_scaled, y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

# Define the Robust Bi-Tempered Logistic Loss Function
def bi_tempered_logistic_loss(y_true, y_pred, alpha=1.0, beta=1.0):
    y_pred = tf.clip_by_value(y_pred, tf.keras.backend.epsilon(), 1 - tf.keras.backend.epsilon())
    loss = tf.where(
        y_true == 1,
        -tf.math.log(y_pred) * (1 + beta * (1 - y_pred)) ** alpha,
        -tf.math.log(1 - y_pred) * (1 + beta * y_pred) ** alpha
    )
    return tf.reduce_mean(loss)
def create_model(input_shape):
    model = models.Sequential()
    model.add(layers.Input(shape=input_shape))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dropout(0.3))
    model.add(layers.Dense(32, activation='relu'))
    model.add(layers.Dropout(0.3))
    model.add(layers.Dense(1, activation='sigmoid'))
    return model
model = create_model(input_shape=(X_scaled.shape[1],))
model.compile(optimizer='adam', loss=bi_tempered_logistic_loss, metrics=['accuracy'])
history = model.fit(X_train, y_train,
                    epochs=30,
                    batch_size=32,
                    validation_data=(X_val, y_val))
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f'Test Loss: {test_loss}, Test Accuracy: {test_accuracy}')
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss')
plt.show()
```

Output:

```
Warning: Looks like you're using an outdated `kagglehub` version, please consider updating (latest version: 0.3.3)
Epoch 1/30
93/93 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.7423 - loss: 0.8592 - val_accuracy: 0.8585 - val_loss: 0.6111
Epoch 2/30
93/93 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8461 - loss: 0.6486 - val_accuracy: 0.8585 - val_loss: 0.5928
Epoch 3/30
93/93 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8433 - loss: 0.6424 - val_accuracy: 0.8585 - val_loss: 0.5883
Epoch 4/30
93/93 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8319 - loss: 0.6791 - val_accuracy: 0.8569 - val_loss: 0.5864
Epoch 5/30
93/93 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8437 - loss: 0.6416 - val_accuracy: 0.8553 - val_loss: 0.5840
Epoch 28/30
93/93 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8453 - loss: 0.6118 - val_accuracy: 0.8648 - val_loss: 0.5821
Epoch 29/30
93/93 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8602 - loss: 0.5717 - val_accuracy: 0.8648 - val_loss: 0.5856
Epoch 30/30
93/93 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8472 - loss: 0.5852 - val_accuracy: 0.8601 - val_loss: 0.5833
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8384 - loss: 0.6553
Test Loss: 0.6049070954322815, Test Accuracy: 0.8522012829780579
```

Result:

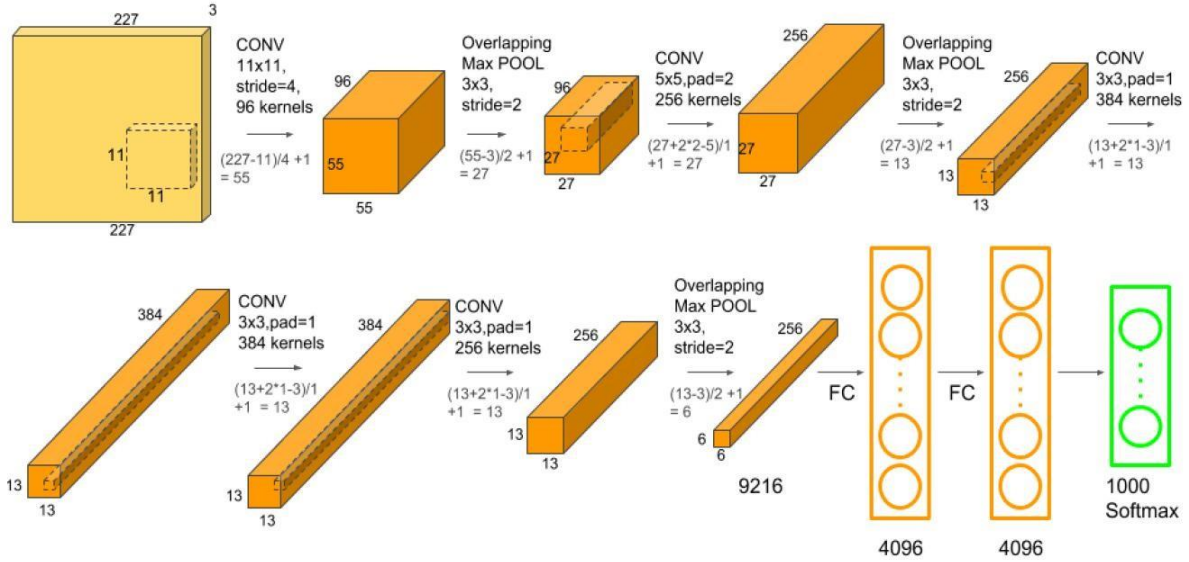
Thus, a Deep learning Network for Robust Bi-Tempered Logistic Loss was executed successfully.

Week-8: Build AlexNet using Advanced CNN.

Aim: To build AlexNet using Advanced CNN.

Description: AlexNet is the name given to a Convolutional Neural Network Architecture that won the LSVRC competition in 2012. LSVRC (Large Scale Visual Recognition Challenge) is a competition where research teams evaluate their algorithms on a huge dataset of labeled images (ImageNet) and compete to achieve higher accuracy on several visual recognition tasks. This made a huge impact on how teams approach the completion afterward.

AlexNet Architechture:



The AlexNet contains 8 layers with weights:

- 5 convolutional layers
- 3 fully connected layers.
- At the end of each layer, ReLu activation is performed except for the last one, which outputs with a softmax with a distribution over the 1000 class labels. Dropout is applied in the first two fully connected layers.
- As the figure above shows also applies Max-pooling after the first, second, and fifth convolutional layers. The kernels of the second, fourth, and fifth convolutional layers are connected only to those kernel maps in the previous layer, which reside on the same GPU.
- The neurons in the fully connected layers are connected to all neurons in the previous layer.

Program:

```
import tensorflow as tf
from tensorflow.keras import layers, models
def create_alexnet(input_shape=(224, 224, 3), num_classes=1000):
    model = models.Sequential()
    # Convolutional Layer 1
        model.add(layers.Conv2D(96, kernel_size=(11, 11), strides=(4, 4), activation='relu', input_shape=input_shape))
    model.add(layers.MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
    model.add(layers.BatchNormalization())
    # Convolutional Layer 2
    model.add(layers.Conv2D(256, kernel_size=(5, 5), padding='same', activation='relu'))
    model.add(layers.MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
    model.add(layers.BatchNormalization())
    # Convolutional Layer 3
    model.add(layers.Conv2D(384, kernel_size=(3, 3), padding='same', activation='relu'))
    model.add(layers.BatchNormalization())
    # Convolutional Layer 4
```

```
    model.add(layers.Conv2D(384, kernel_size=(3, 3), padding='same', activation='relu'))
    model.add(layers.BatchNormalization())
    # Convolutional Layer 5
    model.add(layers.Conv2D(256, kernel_size=(3, 3), padding='same', activation='relu'))
    model.add(layers.MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
    model.add(layers.BatchNormalization())
    # Flattening the layers
    model.add(layers.Flatten())
    # Fully Connected Layer 1
    model.add(layers.Dense(4096, activation='relu'))
    model.add(layers.Dropout(0.5))
    # Fully Connected Layer 2
    model.add(layers.Dense(4096, activation='relu'))
    model.add(layers.Dropout(0.5))
    # Fully Connected Layer 3
    model.add(layers.Dense(num_classes, activation='softmax'))
    return model
alexnet_model = create_alexnet()
alexnet_model.compile(optimizer='adam',loss='categorical  crossentropy',metrics=['accuracy'])
alexnet_model.summary()
```

Output:

Model: "sequential_8"

Layer (type)	Output Shape	Param #
conv2d_40 (Conv2D)	(None, 54, 54, 96)	34,944
max_pooling2d_24 (MaxPooling2D)	(None, 26, 26, 96)	0
batch_normalization_46 (BatchNormalization)	(None, 26, 26, 96)	384
flatten_7 (Flatten)	(None, 6400)	0
dense_21 (Dense)	(None, 4096)	26,218,496
dropout_14 (Dropout)	(None, 4096)	0
dense_22 (Dense)	(None, 4096)	16,781,312
dropout_15 (Dropout)	(None, 4096)	0
dense_23 (Dense)	(None, 1000)	4,097,000

Total params: 50,849,512 (193.98 MB)
Trainable params: 50,846,760 (193.96 MB)
Non-trainable params: 2,752 (10.75 KB)

Result:

Thus, the AlexNet using Advanced CNN was executed successfully.

Week-9: Demonstration of Application of Autoencoders.

Aim: To demonstrate the Application of Autoencoders.

Description: Autoencoders are a type of neural network used for unsupervised learning, primarily for tasks like dimensionality reduction, feature extraction, and denoising. They work by learning to encode input data into a compressed representation and then reconstructing the original data from that representation.

Architecture:

1. **Encoder**: This part of the network compresses the input data into a lower-dimensional representation. It typically consists of one or more layers of neurons that progressively reduce the size of the input.
2. **Bottleneck**: This is the central layer of the autoencoder, where the compressed representation resides. The size of this layer is crucial; it should be small enough to force the model to learn a compressed representation.
3. **Decoder**: This part reconstructs the input data from the compressed representation. It mirrors the encoder but in reverse, progressively increasing the size of the data until it matches the original input.

They can be applied in various domains for tasks such as data compression, denoising, feature learning, and anomaly detection.

Program & Output for Image Denoising:

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D

(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
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.)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

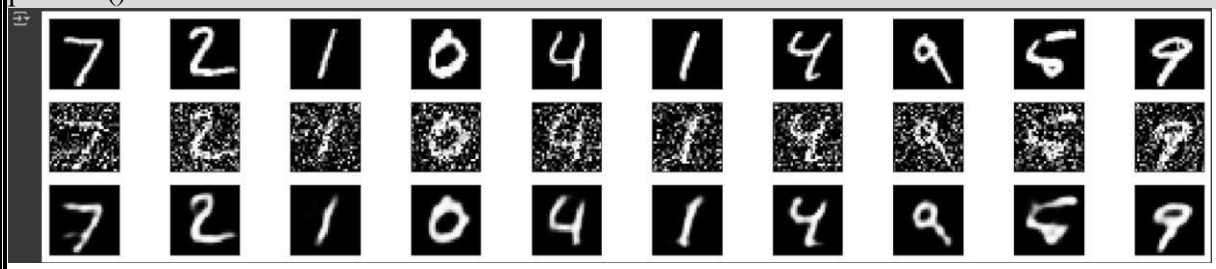
```
input_img = Input(shape=(28, 28, 1))
x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

x = Conv2D(64, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(x_train_noisy.reshape(-1, 28, 28, 1),
                x_train.reshape(-1, 28, 28, 1),epochs=2,batch_size=128,shuffle=True,
                validation_data=(x_test_noisy.reshape(-1, 28, 28, 1), x_test.reshape(-1, 28, 28, 1)))
```

```
Epoch 1/2
469/469 ━━━━━━━━━━━━━━━━━━━━ 188s 401ms/step - loss: 0.1467 - val_loss: 0.1103
Epoch 2/2
469/469 ━━━━━━━━━━━━━━━━━━━━ 185s 395ms/step - loss: 0.1098 - val_loss: 0.1040
<keras.src.callbacks.history.History at 0x7cc880676bf0>
```

```
denoised_images = autoencoder.predict(x_test_noisy.reshape(-1, 28, 28, 1))
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 9s 28ms/step
```

```
n = 10  # Number of images to display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Original Images
    ax = plt.subplot(3, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    # Noisy Input Images
    ax = plt.subplot(3, n, i + 1 + n)
    plt.imshow(x_test_noisy[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    # Denoised Images
    ax = plt.subplot(3, n, i + 1 + 2 * n)
    plt.imshow(denoised_images[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



Result:

Thus, the demonstration of application of Autoencoders was executed successfully.

Week-10: Demonstration of GAN.

Aim: To demonstrate Generative Adversarial Network.

Description: A generative adversarial network (GAN) is a deep learning architecture. It trains two neural networks to compete against each other to generate more authentic new data from a given training dataset. A GAN is called adversarial because it trains two different networks and pits them against each other. One network generates new data by taking an input data sample and modifying it as much as possible. The other network tries to predict whether the generated data output belongs in the original dataset. In other words, the predicting network determines whether the generated data is fake or real. The system generates newer, improved versions of fake data values until the predicting network can no longer distinguish fake from original.

Program:

```
import torch
import torch.nn as nn

# Check if CUDA (GPU) is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Generate synthetic data
n = 1000
first_column = torch.rand(n, 1).to(device)
data = torch.cat([first_column, 2 * first_column, 4 * first_column], dim=1)

# Generator
class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(3, 50),
            nn.ReLU(),
            nn.Linear(50, 3)
        )
    def forward(self, x):
        return self.model(x)

# Discriminator
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(3, 50),
            nn.ReLU(),
            nn.Linear(50, 1),
            nn.Sigmoid()
        )
    def forward(self, x):
        return self.model(x)

# Initialize models and optimizers
generator = Generator().to(device)
discriminator = Discriminator().to(device)
criterion = nn.BCELoss()
optimizer_g = torch.optim.Adam(generator.parameters(), lr=0.001)
optimizer_d = torch.optim.Adam(discriminator.parameters(), lr=0.001)

# Training the GAN
```

```
for epoch in range(5000):
    # Train Discriminator
    optimizer_d.zero_grad()
    real_labels = torch.ones(n, 1).to(device)
    d_loss_real = criterion(discriminator(data), real_labels)

    noise = torch.randn(n, 3).to(device)
    fake_data = generator(noise)
    fake_labels = torch.zeros(n, 1).to(device)
    d_loss_fake = criterion(discriminator(fake_data.detach()), fake_labels)

    d_loss = d_loss_real + d_loss_fake
    d_loss.backward()
    optimizer_d.step()

    # Train Generator
    optimizer_g.zero_grad()
    g_loss = criterion(discriminator(fake_data), real_labels)
    g_loss.backward()
    optimizer_g.step()

    # Print losses every 1000 epochs
    if (epoch + 1) % 1000 == 0:
        print(f"Epoch [{epoch + 1}/5000], d_loss: {d_loss.item():.4f}, g_loss: {g_loss.item():.4f}")

# Generate and print synthetic data
with torch.no_grad():
    generated_data = generator(torch.randn(n, 3).to(device)).cpu().numpy()
print("Generated Data (First 10 rows):")
for i in range(10):
    print(generated_data[i])

# Validation
print("\nValidation (For the first 10 rows):")
for i in range(10):
    first = generated_data[i][0]
    second = generated_data[i][1]
    third = generated_data[i][2]
    print(f"First: {first:.4f}, Expected Second: {2 * first:.4f}, Actual Second: {second:.4f}")
    print(f"Second: {second:.4f}, Expected Third: {2 * second:.4f}, Actual Third: {third:.4f}\n")
```

Output:

```
Epoch [1000/5000], d_loss: 1.3807, g_loss: 0.7136
Epoch [2000/5000], d_loss: 1.3819, g_loss: 0.6897
Epoch [3000/5000], d_loss: 1.3865, g_loss: 0.7122
Epoch [4000/5000], d_loss: 1.3843, g_loss: 0.6772
Epoch [5000/5000], d_loss: 1.3809, g_loss: 0.7139
Generated Data (First 10 rows):
[0.52613646 1.1074238  2.2500124 ]
[0.7367899 1.5288975 3.1192412]
[0.64504904 1.3541396  2.7515311 ]
[0.43475056 0.91756713 1.8714025 ]
[0.2525273 0.5289705 1.0827212]
[0.5141439 1.0836086 2.2145708]
[0.3136536 0.6661093 1.3817189]
[0.84708637 1.7631962  3.6157594 ]
[0.4329918 0.9188332 1.8764479]
[0.7067039 1.4907482 3.0465863]
```

```
Validation (For the first 10 rows):
First: 0.5261, Expected Second: 1.0523, Actual Second: 1.1074
Second: 1.1074, Expected Third: 2.2148, Actual Third: 2.2500

First: 0.7368, Expected Second: 1.4736, Actual Second: 1.5289
Second: 1.5289, Expected Third: 3.0578, Actual Third: 3.1192

First: 0.6450, Expected Second: 1.2901, Actual Second: 1.3541
Second: 1.3541, Expected Third: 2.7083, Actual Third: 2.7515

First: 0.4348, Expected Second: 0.8695, Actual Second: 0.9176
Second: 0.9176, Expected Third: 1.8351, Actual Third: 1.8714

First: 0.2525, Expected Second: 0.5051, Actual Second: 0.5290
Second: 0.5290, Expected Third: 1.0579, Actual Third: 1.0827

First: 0.5141, Expected Second: 1.0283, Actual Second: 1.0836
Second: 1.0836, Expected Third: 2.1672, Actual Third: 2.2146

First: 0.3137, Expected Second: 0.6273, Actual Second: 0.6661
Second: 0.6661, Expected Third: 1.3322, Actual Third: 1.3817

First: 0.8471, Expected Second: 1.6942, Actual Second: 1.7632
Second: 1.7632, Expected Third: 3.5264, Actual Third: 3.6158

First: 0.4330, Expected Second: 0.8660, Actual Second: 0.9188
Second: 0.9188, Expected Third: 1.8377, Actual Third: 1.8764

First: 0.7067, Expected Second: 1.4134, Actual Second: 1.4907
Second: 1.4907, Expected Third: 2.9815, Actual Third: 3.0466
```

Result:

Thus, the GAN technology was executed successfully.