Ex.No: 1 Roll.No: 210701149

PYTHON PROGRAM TO BUILD A SIMPLE NEURAL NETWORK WITH KERAS

Aim:

To implement a simple neural network with keras using python language,

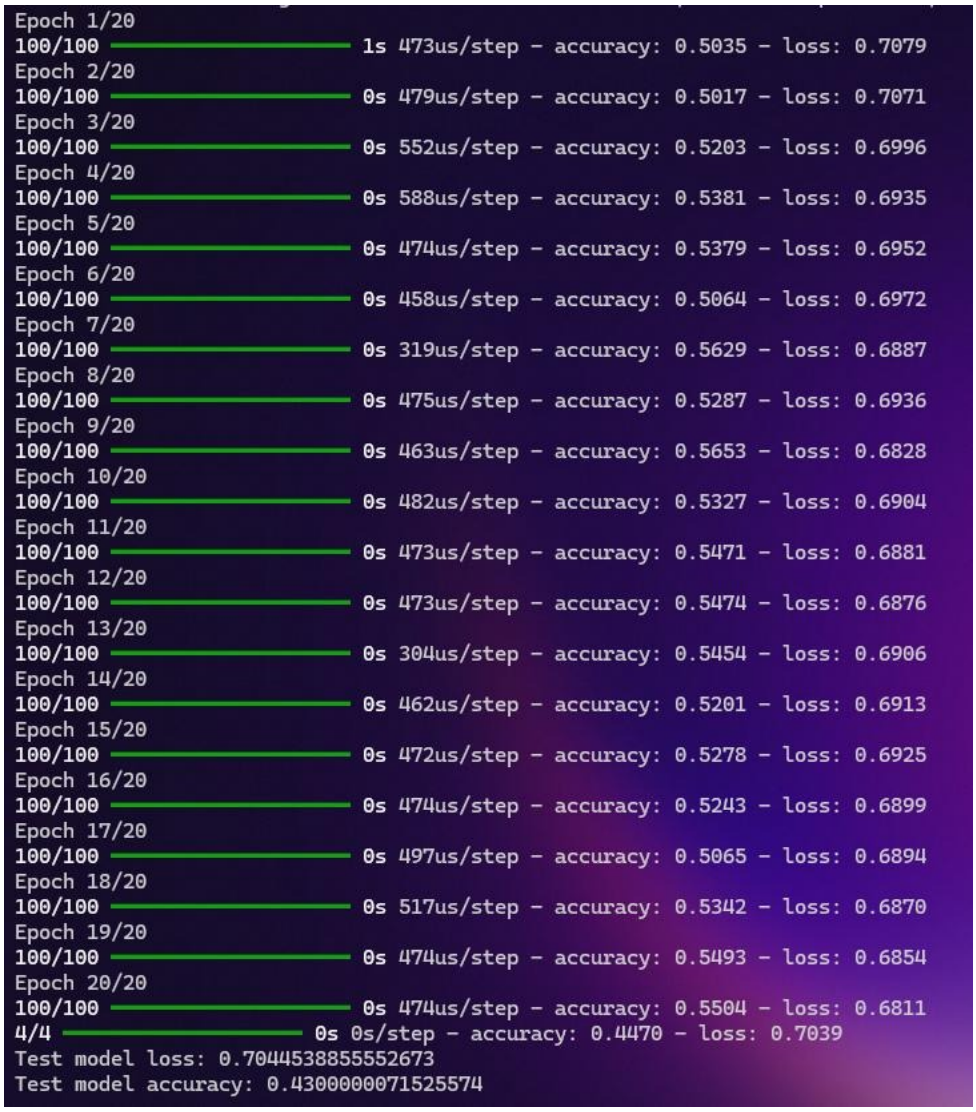
Procedure:

1. Import NumPy and necessary Keras modules for building the model.
2. Generate random dummy training data with 1000 samples and 10 features each.
3. Create random binary labels (0 or 1) for the training data.
4. Initialize a Sequential model for a simple feedforward neural network.
5. Add a Dense layer with 10 units and ReLU activation for the input.
6. Add another Dense layer with 1 unit and sigmoid activation for binary classification.
7. Compile the model using Adam optimizer and binary cross-entropy loss.
8. Train the model for 20 epochs with a batch size of 10 using the training data.
9. Generate random dummy test data with 100 samples and binary labels.
10. Evaluate the model on the test data and print the loss and accuracy values.

Code:

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

# Generate some dummy data for training
x_train_data = np.random.random((1000, 10))
y_train_data = np.random.randint(2, size=(1000, 1))

# Building the model
model = Sequential()
model.add(Dense(10, activation='relu', input_dim=10))
model.add(Dense(1, activation='sigmoid'))

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

# Train the model
model.fit(x_train_data, y_train_data, epochs=20, batch_size=10)

# Generate some dummy test data
x_test_data = np.random.random((100, 10))
y_test_data = np.random.randint(2, size=(100, 1))

# Evaluating the model on the test data
loss, accuracy = model.evaluate(x_test_data, y_test_data)
print('Test model loss:', loss)
print('Test model accuracy:', accuracy)
```

Output:



Result:

Thus, to implement a simple neural networks using Keras in Python has been completed successfully.

Ex.No: 2 Roll.No: 210701157

PYTHON PROGRAM TO BUILD A CONVOLUTIONAL NEURAL NETWORK WITH KERAS

Aim:

To build a convolutional neural network with Keras in Python.

Procedure:

1. Import TensorFlow and Keras modules for building and training the model.
2. Load the CIFAR-10 dataset consisting of 60,000 32x32 color images across 10 classes.
3. Normalize the pixel values of the training and testing images to be between 0 and 1.
4. Print the shape of the training and test images and labels to verify the dataset dimensions.
5. Create a Sequential CNN model starting with a 32-filter Conv2D layer followed by MaxPooling.
6. Add two more Conv2D layers with 64 filters each, followed by MaxPooling and activation.
7. Flatten the feature map and add a Dense layer with 64 units and an output layer with 10 units.
8. Compile the model using Adam optimizer, Sparse Categorical Crossentropy loss, and accuracy metrics.
9. Train the model for 10 epochs with validation on test data and store the training history.
10. Evaluate the model's accuracy on the test set and visualize the accuracy and loss during training using plots.

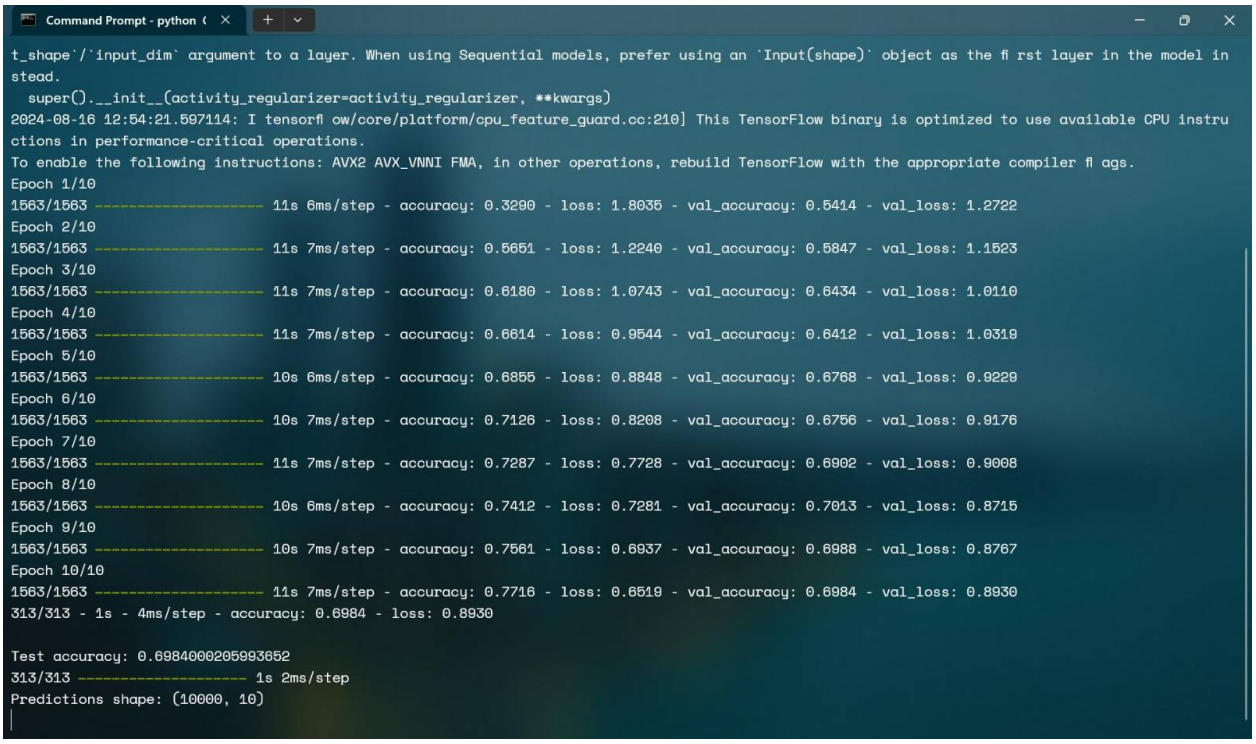
Code:

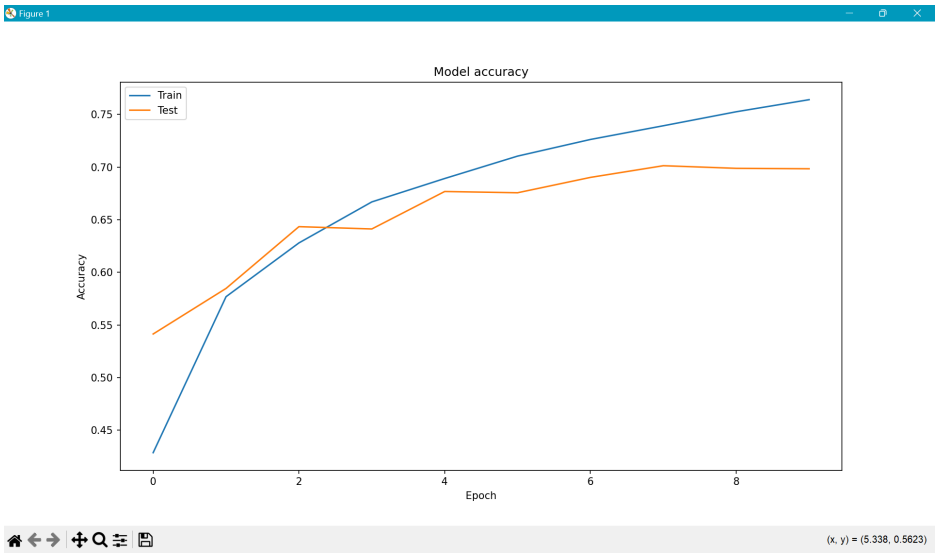
```
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt

# Load the CIFAR-10 dataset
(train_images, train_labels), (test_images, test_labels) =
tf.keras.datasets.cifar10.load_data()

# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0

# Print the shape of the dataset
print(f"Training images shape: {train_images.shape}")
print(f"Training labels shape: {train_labels.shape}")
print(f"Test images shape: {test_images.shape}")
print(f"Test labels shape: {test_labels.shape}")

model = models.Sequential([
    # Convolutional layer 1
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    layers.MaxPooling2D((2, 2)),

    # Convolutional layer 2
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    # Convolutional layer 3
    layers.Conv2D(64, (3, 3), activation='relu'),

    # Flatten and Fully Connected (Dense) Layers
```

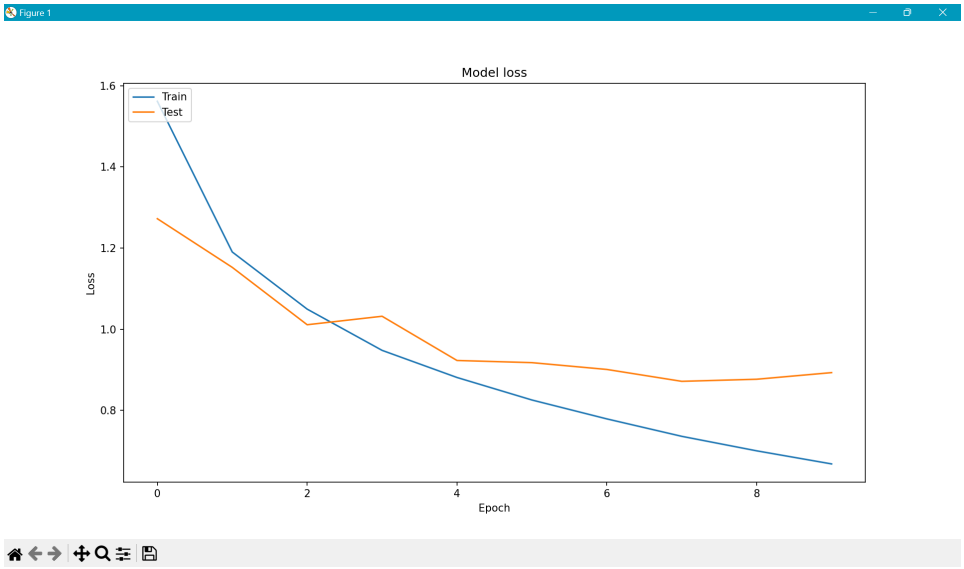
```
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10)  # 10 classes
])

model.compile(
    optimizer='adam',
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=['accuracy']
)

history = model.fit(
    train_images,
    train_labels,
    epochs=10,
    validation_data=(test_images, test_labels)
)

test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print(f'\nTest accuracy: {test_acc}')

predictions = model.predict(test_images)
print(f'Predictions shape: {predictions.shape}')

# Plot training & validation accuracy values
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['Train', 'Test'], loc='upper left')
```

```
plt.show()

# Plot training & validation loss values
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

Output:







Result:

Thus, to build a convolutional neural network using keras has been completed successfully.

Ex.No: 3 Roll.No: 210701157

PYTHON PROGRAM TO CREATE A NEURAL NETWORK TO RECOGNIZE HANDWRITTEN DIGITS USING MNIST DATASET

Aim:

To create a neural network to recognize handwritten digits using MNIST dataset in python.

Procedure:

1. Import TensorFlow, Keras, and Matplotlib for building the model and plotting.
2. Load the MNIST dataset, consisting of handwritten digits.
3. Reshape and normalize the training and test images to have pixel values between 0 and 1.
4. Convert the training and test labels to one-hot encoded vectors.
5. Build a Sequential model and add a Conv2D layer with 32 filters and ReLU activation.
6. Add a MaxPooling layer, followed by another Conv2D layer with 64 filters and ReLU activation.
7. Add a third Conv2D layer, flatten the output, and add a Dense layer with 64 units and ReLU activation.
8. Add a final Dense layer with 10 units and softmax activation for classification.
9. Compile the model with Adam optimizer and categorical cross-entropy loss, and train it for 5 epochs with 20% validation split.
10. Evaluate the model on test data and plot the training and validation accuracy and loss over epochs.

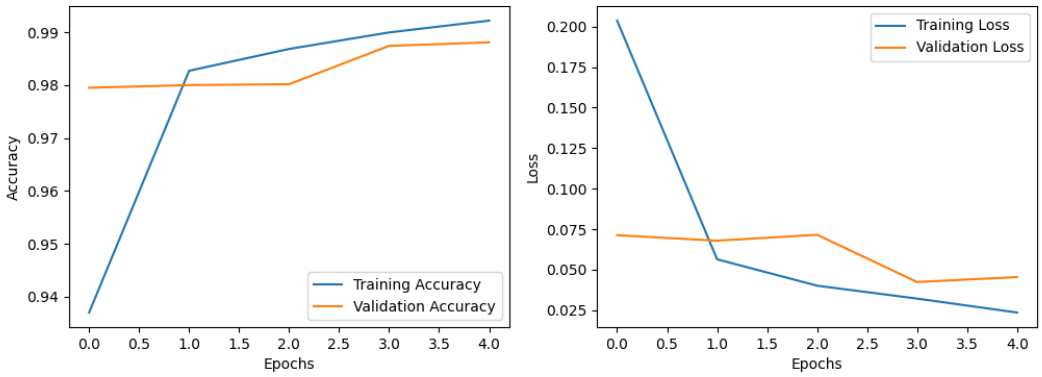
Code:

```
# Import necessary libraries
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt

# Load the MNIST dataset
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Preprocess the data
train_images = train_images.reshape((60000, 28, 28, 1)).astype('float32') / 255
test_images = test_images.reshape((10000, 28, 28, 1)).astype('float32') / 255

# Convert labels to one-hot encoding
train_labels = tf.keras.utils.to_categorical(train_labels)
test_labels = tf.keras.utils.to_categorical(test_labels)

# Build the neural network model
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
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(64, (3, 3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='adam',
```

```
        loss='categorical_crossentropy',
        metrics=['accuracy'])

# Train the model
history = model.fit(train_images, train_labels, epochs=5, batch_size=64,
validation_split=0.2)

# Evaluate the model on test data
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'Test accuracy: {test_acc}')

# Plot the accuracy and loss over epochs
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

Output:

```
Epoch 1/5
750/750 ━━━━━━━━━━━━━━━━━━━━ 9s 11ms/step - accuracy: 0.8431 - loss: 0.5073 - val_accuracy: 0.9792 - val_loss: 0.0715
Epoch 2/5
750/750 ━━━━━━━━━━━━━━━━━━━━ 8s 11ms/step - accuracy: 0.9789 - loss: 0.0655 - val_accuracy: 0.9812 - val_loss: 0.0605
Epoch 3/5
750/750 ━━━━━━━━━━━━━━━━━━━━ 9s 12ms/step - accuracy: 0.9865 - loss: 0.0421 - val_accuracy: 0.9861 - val_loss: 0.0473
Epoch 4/5
750/750 ━━━━━━━━━━━━━━━━━━━━ 9s 12ms/step - accuracy: 0.9889 - loss: 0.0347 - val_accuracy: 0.9886 - val_loss: 0.0408
Epoch 5/5
750/750 ━━━━━━━━━━━━━━━━━━━━ 9s 13ms/step - accuracy: 0.9913 - loss: 0.0287 - val_accuracy: 0.9888 - val_loss: 0.0410
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9866 - loss: 0.0403
Test accuracy: 0.9901000261306763
2024-09-16 12:59:58.143 Python[99718:1483561] +[IMKClient subclass]: chose IMKClient_Legacy
2024-09-16 12:59:58.143 Python[99718:1483561] +[IMKInputSession subclass]: chose IMKInputSession_Legacy
```



Result:

Thus, to implement neural network to recognize handwritten digits using MNIST dataset in python has been completed successfully.

Ex.No: 4 Roll.No: 210701149

PYTHON PROGRAM TO VISUALIZE AND DESIGN CNN WITH TRANSFER LEARNING

Aim:

To visualize and design CNN with transfer learning in python.

Procedure:

1. Import TensorFlow, CIFAR-10 dataset, VGG16 model, and necessary Keras layers for building the model.
2. Load the CIFAR-10 dataset, consisting of 60,000 images, and divide it into training and test sets.
3. Normalize the training and test images by scaling pixel values to the range [0, 1].
4. One-hot encode the training and test labels to convert them into categorical format.
5. Load the pre-trained VGG16 model with ImageNet weights, excluding the top fully connected layers.
6. Freeze the layers of the pre-trained VGG16 model to prevent them from being trained.
7. Initialize a Sequential model and add the pre-trained VGG16 as the base.
8. Add a Flatten layer followed by a Dense layer with 256 units and ReLU activation.
9. Add a final Dense layer with 10 units and softmax activation for CIFAR-10 classification.
10. Compile the model with Adam optimizer, train for 10 epochs, and evaluate it on test data to print the accuracy.

Code:

```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.applications import VGG16
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.utils import to_categorical

# Load CIFAR-10 dataset
(itrain, ltrain), (itest, ltest) = cifar10.load_data()

# Preprocess the data
itrain = itrain / 255.0
itest = itest / 255.0
ltrain = to_categorical(ltrain)
ltest = to_categorical(ltest)

# Load pre-trained VGG16 model (excluding the top fully-connected layers)
basem = VGG16(weights='imagenet', include_top=False, input_shape=(32, 32, 3))

# Freeze the pre-trained layers
for layer in basem.layers:
  layer.trainable = False

# Create a new model on top
semodel = Sequential()
semodel.add(basem)
semodel.add(Flatten())
semodel.add(Dense(256, activation='relu'))
semodel.add(Dense(10, activation='softmax'))  # CIFAR-10 has 10 classes
```

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

# Train the model
semodel.fit(itrain, ltrain, epochs=10, batch_size=32, validation_data=(itest, ltest))

# Evaluate the model on test data
ltest, atest = semodel.evaluate(itest, ltest)
print("Test accuracy:", atest)
```

Output:

```
Epoch 1/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 121s 77ms/step - accuracy: 0.4755 - loss: 1.4946 - val_accuracy: 0.5627 - val_loss: 1.2390
Epoch 2/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 127s 81ms/step - accuracy: 0.5919 - loss: 1.1764 - val_accuracy: 0.5817 - val_loss: 1.1992
Epoch 3/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 129s 82ms/step - accuracy: 0.6114 - loss: 1.1064 - val_accuracy: 0.5908 - val_loss: 1.1614
Epoch 4/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 125s 80ms/step - accuracy: 0.6290 - loss: 1.0572 - val_accuracy: 0.6099 - val_loss: 1.1226
Epoch 5/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 123s 79ms/step - accuracy: 0.6389 - loss: 1.0196 - val_accuracy: 0.6101 - val_loss: 1.1246
Epoch 6/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 122s 78ms/step - accuracy: 0.6570 - loss: 0.9742 - val_accuracy: 0.6099 - val_loss: 1.1118
Epoch 7/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 121s 77ms/step - accuracy: 0.6702 - loss: 0.9425 - val_accuracy: 0.6192 - val_loss: 1.1128
Epoch 8/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 115s 74ms/step - accuracy: 0.6810 - loss: 0.9023 - val_accuracy: 0.6108 - val_loss: 1.1179
Epoch 9/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 116s 74ms/step - accuracy: 0.6898 - loss: 0.8815 - val_accuracy: 0.6194 - val_loss: 1.1053
Epoch 10/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 115s 73ms/step - accuracy: 0.6964 - loss: 0.8575 - val_accuracy: 0.6141 - val_loss: 1.1101
313/313 ━━━━━━━━━━━━━━━━━━━━ 19s 61ms/step - accuracy: 0.6153 - loss: 1.1069
Test accuracy: 0.6140999794006348
```

Result:

Thus, to visualize and design CNN with transfer learning has been completed successfully.

Ex.No: 5 Roll.No: 210701157

PYTHON PROGRAM TO BUILD AN RNN WITH KERAS

Aim:

To build a RNN(Recurrent Neural Networks) with keras in python.

Procedure:

1. Import NumPy, TensorFlow, IMDB dataset utilities, RNN layers, and Matplotlib for plotting.
2. Set a random seed for reproducibility in NumPy and TensorFlow.
3. Load the IMDB dataset, restricting to the top 10,000 words and specifying a maximum sequence length of 200.
4. Pad sequences to ensure uniform input length across all samples.
5. Build a Sequential model with an Embedding layer, a SimpleRNN layer, and a Dense output layer for binary classification.
6. Compile the model using Adam optimizer and binary cross-entropy loss, with accuracy as the evaluation metric.
7. Print the model summary to check the architecture and layer details.
8. Train the model for 5 epochs with a batch size of 64, including validation data.
9. Evaluate the model on the test data and print the accuracy.
10. Plot training and validation accuracy and loss over epochs using Matplotlib.

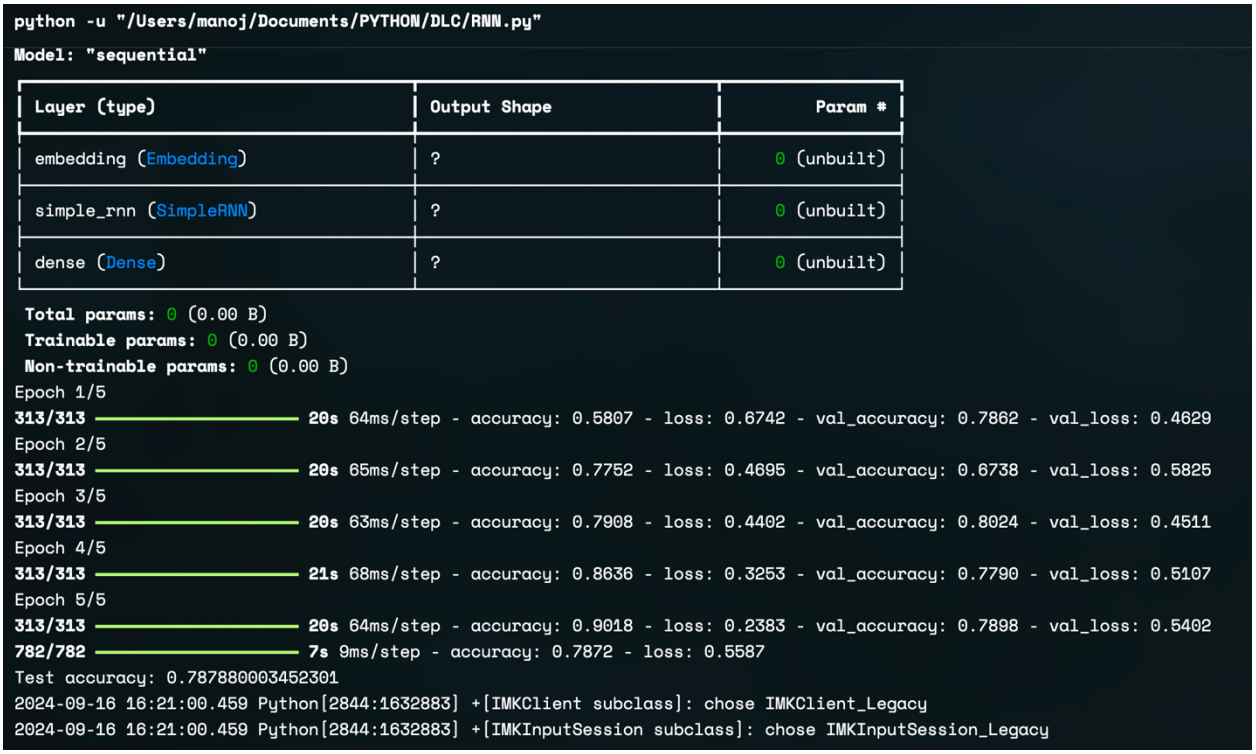
Code:

```
# Import necessary libraries
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing import sequence
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense
import matplotlib.pyplot as plt

# Set random seed for reproducibility
np.random.seed(42)
tf.random.set_seed(42)

# Load the IMDB dataset
max_features = 10000  # Number of words to consider as features
maxlen = 200  # Maximum length of input sequences

(train_data, train_labels), (test_data, test_labels) =
imdb.load_data(num_words=max_features)

# Pad sequences to ensure consistent input size
train_data = sequence.pad_sequences(train_data, maxlen=maxlen)
test_data = sequence.pad_sequences(test_data, maxlen=maxlen)

# Build the RNN model
model = Sequential()
model.add(Embedding(input_dim=max_features, output_dim=128,
input_length=maxlen))
model.add(SimpleRNN(128, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

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

# Print model summary
model.summary()

# Train the model
history = model.fit(train_data, train_labels,
                    epochs=5,
                    batch_size=64,
                    validation_split=0.2,
                    verbose=1)

# Evaluate the model
test_loss, test_acc = model.evaluate(test_data, test_labels)
print(f'Test accuracy: {test_acc}')

# Plot training and validation accuracy and loss
plt.figure(figsize=(12, 4))

# Accuracy plot
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
```

```
# Loss plot
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

Output:



Result:

Thus, to build a RNN using Keras in python has been completed successfully.

Ex.No: 6 Roll.No: 210701157

PYTHON PROGRAM TO BUILD AUTOENCODERS WITH KERAS

Aim:

To build autoencoders with keras in python.

Procedure:

1. Import NumPy and Keras modules for building and training the autoencoder.
2. Define the input dimension (e.g., 784 for flattened 28x28 MNIST images) and encoding dimension.
3. Create an input layer with the specified input dimension.
4. Build the encoder part of the autoencoder with three Dense layers, reducing dimensionality to the encoding dimension.
5. Construct the decoder part of the autoencoder with three Dense layers, reconstructing the input to the original dimension.
6. Define the autoencoder model with the input layer and the reconstructed output.
7. Create a separate encoder model to extract encoded features from the input.
8. Compile the autoencoder using Adam optimizer and binary cross-entropy loss.
9. Generate dummy training and test data and train the autoencoder for 50 epochs.
10. Use the encoder to obtain encoded representations and the autoencoder to reconstruct the input, printing the shapes of the results.

Code:

```
import numpy as np
from keras.layers import Input, Dense
from keras.models import Model

# Define input dimension
input_dim = 784  # For example, flattened 28x28 MNIST images

# Define encoding dimension
encoding_dim = 32

# Input layer
input_img = Input(shape=(input_dim,))

# Encoder layers
encoded = Dense(128, activation='relu')(input_img)
encoded = Dense(64, activation='relu')(encoded)
encoded = Dense(encoding_dim, activation='relu')(encoded)

# Decoder layers
decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(decoded)
decoded = Dense(input_dim, activation='sigmoid')(decoded)

# Autoencoder model
autoencoder = Model(input_img, decoded)

# Separate encoder model
encoder = Model(input_img, encoded)

# Compile the model
```

```
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

# Generate dummy data for demonstration
x_train = np.random.random((1000, input_dim))
x_test = np.random.random((200, input_dim))

# Train the autoencoder
autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))

# Use the encoder to encode some input
encoded_imgs = encoder.predict(x_test)

# Use the autoencoder to reconstruct some input
decoded_imgs = autoencoder.predict(x_test)

print("Shape of encoded images:", encoded_imgs.shape)
print("Shape of decoded images:", decoded_imgs.shape)
```

Output:

```
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.6878 - val_loss: 0.6923
Epoch 38/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.6878 - val_loss: 0.6922
Epoch 39/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.6876 - val_loss: 0.6922
Epoch 40/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.6875 - val_loss: 0.6922
Epoch 41/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.6874 - val_loss: 0.6922
Epoch 42/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.6872 - val_loss: 0.6922
Epoch 43/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.6870 - val_loss: 0.6921
Epoch 44/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.6869 - val_loss: 0.6921
Epoch 45/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.6868 - val_loss: 0.6921
Epoch 46/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.6867 - val_loss: 0.6921
Epoch 47/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.6866 - val_loss: 0.6921
Epoch 48/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.6864 - val_loss: 0.6921
Epoch 49/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.6862 - val_loss: 0.6921
Epoch 50/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.6860 - val_loss: 0.6922
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
Shape of encoded images: (200, 32)
Shape of decoded images: (200, 784)
```

Result:

Thus, to build autoencoders with Keras has been done successfully.

Ex.No: 7 Roll.No: 210701149

PYTHON PROGRAM TO BUILD GAN WITH KERAS

Aim:

To build GAN (Generative Adversarial Network) with keras in python.

Procedure:

1. Import necessary libraries including NumPy, Matplotlib, TensorFlow Keras components, and MNIST dataset.
2. Load and preprocess the MNIST dataset, normalizing images and reshaping them to (batch_size, 28, 28, 1).
3. Define image dimensions and random noise vector dimension for the GAN.
4. Build the Generator model with Dense layers, LeakyReLU activations, BatchNormalization, and a final output reshaped to the image dimensions.
5. Build the Discriminator model with Flatten, Dense layers, LeakyReLU activations, and a final sigmoid activation.
6. Build the GAN model by combining the Generator and Discriminator models, setting the Discriminator layers to non-trainable.
7. Compile the Discriminator and GAN models using Adam optimizer and binary cross-entropy loss.
8. Implement the training function to alternately train the Discriminator on real and fake images, then train the Generator.
9. Save generated images at specified intervals during training to visualize the Generator's progress.
10. Execute the training process for a defined number of epochs, saving generated images every save_interval epochs.

Code:

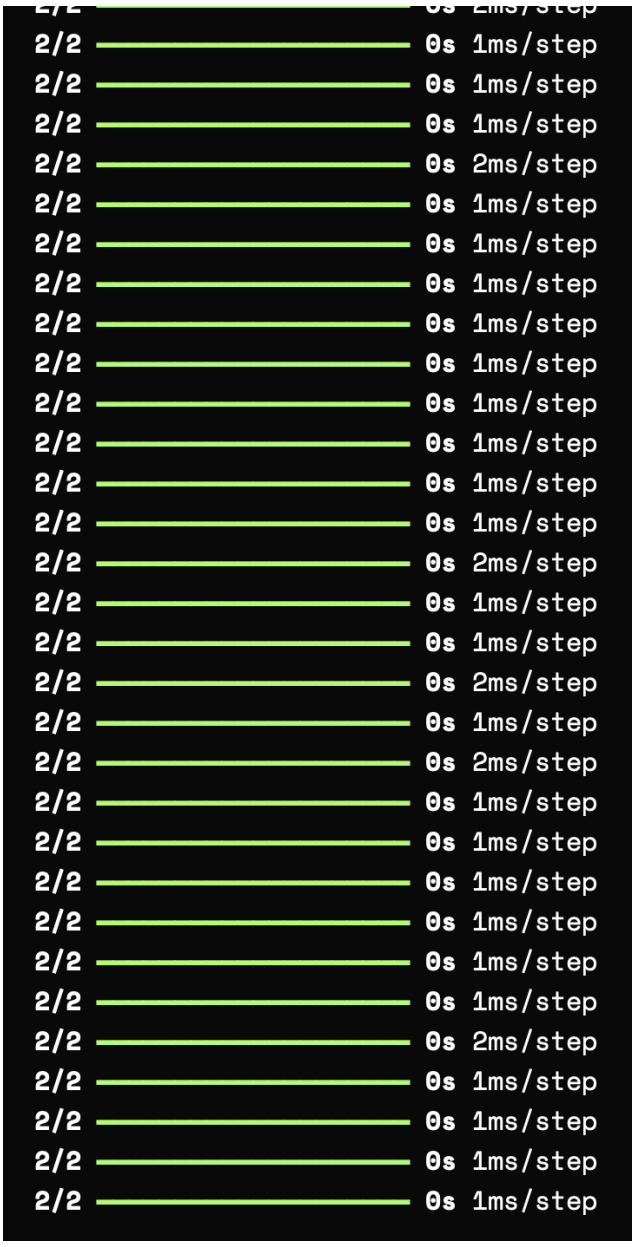
```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Dense, LeakyReLU, BatchNormalization,
Reshape, Flatten
from tensorflow.keras.optimizers import Adam

# Load and preprocess the MNIST dataset
(X_train, _), (_, _) = mnist.load_data()
X_train = X_train / 255.0  # Normalize images to [0, 1]
X_train = X_train.reshape(X_train.shape[0], 28, 28, 1)  # Reshape to (batch_size,
28, 28, 1)

# Define the dimensions
img_rows, img_cols, channels = 28, 28, 1
img_shape = (img_rows, img_cols, channels)
z_dim = 100  # Dimension of the random noise vector

# Build the Generator model
def build_generator():
    model = Sequential()
    model.add(Dense(256, input_dim=z_dim))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization())
    model.add(Dense(512))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization())
    model.add(Dense(1024))
```

```
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization())
    model.add(Dense(np.prod(img_shape), activation='tanh'))
    model.add(Reshape(img_shape))
    return model

# Build the Discriminator model
def build_discriminator():
    model = Sequential()
    model.add(Flatten(input_shape=img_shape))
    model.add(Dense(512))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dense(256))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dense(1, activation='sigmoid'))
    return model

# Build the GAN model
def build_gan(generator, discriminator):
    discriminator.trainable = False
    model = Sequential()
    model.add(generator)
    model.add(discriminator)
    return model

# Compile the models
def compile_models(generator, discriminator, gan):
    discriminator.compile(optimizer=Adam(), loss='binary_crossentropy',
metrics=['accuracy'])
    gan.compile(optimizer=Adam(), loss='binary_crossentropy')
```

```
# Training function
def train_gan(epochs, batch_size=128, save_interval=50):
    # Load the discriminator and generator models
    generator = build_generator()
    discriminator = build_discriminator()
    gan = build_gan(generator, discriminator)
    compile_models(generator, discriminator, gan)

    # Training loop
    for epoch in range(epochs):
        # Train the Discriminator
        idx = np.random.randint(0, X_train.shape[0], batch_size)
        real_imgs = X_train[idx]
        real_labels = np.ones((batch_size, 1))
        fake_imgs = generator.predict(np.random.randn(batch_size, z_dim))
        fake_labels = np.zeros((batch_size, 1))

        d_loss_real = discriminator.train_on_batch(real_imgs, real_labels)
        d_loss_fake = discriminator.train_on_batch(fake_imgs, fake_labels)
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

        # Train the Generator
        noise = np.random.randn(batch_size, z_dim)
        valid_labels = np.ones((batch_size, 1))
        g_loss = gan.train_on_batch(noise, valid_labels)

        # Print progress and save generated images
        if epoch % save_interval == 0:
            print(f"{epoch} [D loss: {d_loss[0]} | D accuracy: {100 * d_loss[1]]}%] [G
loss: {g_loss}]")
            save_generated_images(generator, epoch)
```

```
# Function to save generated images
def save_generated_images(generator, epoch, examples=16, dim=(4, 4),
figsize=(4, 4)):
    noise = np.random.randn(examples, z_dim)
    gen_imgs = generator.predict(noise)
    gen_imgs = 0.5 * gen_imgs + 0.5  # Rescale images to [0, 1]
    plt.figure(figsize=figsize)
    for i in range(examples):
        plt.subplot(dim[0], dim[1], i + 1)
        plt.imshow(gen_imgs[i, :, :, 0], cmap='gray')
        plt.axis('off')
    plt.tight_layout()
    plt.savefig(f"gan_generated_images_epoch_{epoch}.png")
    plt.close()

# Run the training
train_gan(epochs=10000, batch_size=64, save_interval=1000)
```

Output:



Result:

Thus, to build GAN with keras with python has been completed successfully.