Build the Image classification model by dividing the model into following 4 stages: a. Loading and preprocessing the image data b. Defining the model's architecture c. Training the model d. Estimating the model's performance

```
# Import necessary libraries

import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.preprocessing.image import ImageDataGenerator

import matplotlib.pyplot as plt


# a. Loading and preprocessing the image data


# Define your data directories for training and validation

train_dir = 'path/to/training_data'

val_dir = 'path/to/validation_data'


# Use ImageDataGenerator for data augmentation and normalization

train_datagen = ImageDataGenerator(

   rescale=1./255,

   shear_range=0.2,

   zoom_range=0.2,

   horizontal_flip=True

)


val_datagen = ImageDataGenerator(rescale=1./255)


# Set the image size and batch size

img_size = (150, 150)

batch_size = 32


# Create the data generators

train_generator = train_datagen.flow_from_directory(
```

```
    train_dir,
    target_size=img_size,
    batch_size=batch_size,
    class_mode='binary'  # Change to 'categorical' if more than two classes
)

val_generator = val_datagen.flow_from_directory(
    val_dir,
    target_size=img_size,
    batch_size=batch_size,
    class_mode='binary'
)

# b. Defining the model's architecture

# Create a simple convolutional neural network (CNN) model
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 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.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))  # Change to the number of classes for more than two classes

# Print the model summary
model.summary()
```

```
# c. Training the model

# Compile the model
model.compile(optimizer='adam',
              loss='binary_crossentropy',  # Change to 'categorical_crossentropy' if more than two classes
              metrics=['accuracy'])

# Train the model
epochs = 10  # You can increase this for better performance
history = model.fit(
    train_generator,
    steps_per_epoch=train_generator.samples // batch_size,
    epochs=epochs,
    validation_data=val_generator,
    validation_steps=val_generator.samples // batch_size
)

# d. Estimating the model’s performance

# Plot training history
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# Evaluate the model on the validation set
val_loss, val_acc = model.evaluate(val_generator)
print(f"Validation Accuracy: {val_acc*100:.2f}%")
print(f"Validation Loss: {val_loss:.4f}")# Import necessary libraries
```

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

# a. Loading and preprocessing the image data

# Define your data directories for training and validation
train_dir = 'path/to/training_data'
val_dir = 'path/to/validation_data'

# Use ImageDataGenerator for data augmentation and normalization
train_datagen = ImageDataGenerator(
    rescale=1./255,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
)

val_datagen = ImageDataGenerator(rescale=1./255)

# Set the image size and batch size
img_size = (150, 150)
batch_size = 32

# Create the data generators
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=img_size,
    batch_size=batch_size,
    class_mode='binary'  # Change to 'categorical' if more than two classes
```

```
)

val_generator = val_datagen.flow_from_directory(
    val_dir,
    target_size=img_size,
    batch_size=batch_size,
    class_mode='binary'
)

# b. Defining the model’s architecture

# Create a simple convolutional neural network (CNN) model
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 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.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))  # Change to the number of classes for more than two classes

# Print the model summary
model.summary()

# c. Training the model

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

```
        loss='binary_crossentropy',  # Change to 'categorical_crossentropy' if more than two classes
        metrics=['accuracy'])

# Train the model
epochs = 10  # You can increase this for better performance
history = model.fit(
    train_generator,
    steps_per_epoch=train_generator.samples // batch_size,
    epochs=epochs,
    validation_data=val_generator,
    validation_steps=val_generator.samples // batch_size
)

# d. Estimating the model's performance

# Plot training history
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# Evaluate the model on the validation set
val_loss, val_acc = model.evaluate(val_generator)
print(f"Validation Accuracy: {val_acc*100:.2f}%")
print(f"Validation Loss: {val_loss:.4f}")
```

```
# Import necessary libraries
import tensorflow as tf
from tensorflow.keras import layers, models
```

```
from tensorflow.keras.datasets import mnist

from tensorflow.keras.utils import to_categorical

import matplotlib.pyplot as plt


# a. Loading and preprocessing the image data


# Load and preprocess the MNIST dataset

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()


# Normalize pixel values to be between 0 and 1

train_images, test_images = train_images / 255.0, test_images / 255.0


# Add a channel dimension to the images (MNIST is grayscale, so channel=1)

train_images = train_images[..., tf.newaxis]

test_images = test_images[..., tf.newaxis]


# One-hot encode the labels

train_labels = to_categorical(train_labels)

test_labels = to_categorical(test_labels)


# b. Defining the model’s architecture


# Create a simple convolutional neural network (CNN) 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(128, (3, 3), activation='relu'))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Flatten())
```

```
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))  # 10 classes for digits 0-9

# Print the model summary
model.summary()

# c. Training the model

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

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

# d. Estimating the model's performance

# Plot training history
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

Evaluate the model on the test set

test_loss, test_acc = model.evaluate(test_images, test_labels)

print(f"Test Accuracy: {test_acc*100:.2f}%")

print(f"Test Loss: {test_loss:.4f}")

CBOW

Import necessary libraries

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Embedding, Dense, Lambda

import numpy as np

a. Data preparation

Sample paragraph for demonstration

paragraph = "Continuous Bag of Words (CBOW) is a neural network model for natural language processing."

Tokenize the paragraph into words

tokens = tf.keras.preprocessing.text.text_to_word_sequence(paragraph)

Create a word-to-index mapping

word_index = {word: idx for idx, word in enumerate(set(tokens))}

Convert words to indices

indexed_tokens = [word_index[word] for word in tokens]

b. Generate training data

Define the context window size

```
context_window = 2


# Generate training data in the CBOW format
def generate_cbow_data(indexed_tokens, context_window):
    training_data = []
    for i in range(context_window, len(indexed_tokens) - context_window):
        context = [indexed_tokens[i - j] for j in range(context_window)] + [indexed_tokens[i + j] for j in range(1, context_window + 1)]
        target = indexed_tokens[i]
        training_data.append((context, target))
    return training_data


cbow_training_data = generate_cbow_data(indexed_tokens, context_window)


# Convert training data to NumPy arrays
X_train = np.array([np.array(context) for context, _ in cbow_training_data])
y_train = np.array([target for _, target in cbow_training_data])


# c. Train model


# Set the embedding dimension
embedding_dim = 50


# Build the CBOW model
model = Sequential([
    Embedding(input_dim=len(word_index), output_dim=embedding_dim, input_length=context_window * 2),
    Lambda(lambda x: tf.reduce_mean(x, axis=1)),  # Average over the context window
    Dense(len(word_index), activation='softmax')  # Output layer for predicting the target word
])


# Compile the model
```

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

# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=32)

# d. Output

# Get the word embeddings from the trained model
word_embeddings = model.layers[0].get_weights()[0]

# Example: Get the embedding for a specific word
# Example: Get the embedding for a specific word
word_to_lookup = 'CBOW'

if word_to_lookup in word_index:
    word_index_lookup = word_index[word_to_lookup]
    embedding_lookup = word_embeddings[word_index_lookup]
    print(f"Embedding for '{word_to_lookup}': {embedding_lookup}")
else:
    print(f"Word '{word_to_lookup}' not found in the dataset.")
```

Autoencoder

```
# a. Import required libraries
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
import numpy as np

# b. Upload / access the dataset

# Load the MNIST dataset
```

```
(train_images, _), (test_images, _) = mnist.load_data()

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

# Add a channel dimension to the images (MNIST is grayscale, so channel=1)
train_images = train_images[..., tf.newaxis]
test_images = test_images[..., tf.newaxis]

# c. Encoder converts it into a latent representation

# Create an autoencoder model
encoder = models.Sequential()
encoder.add(layers.InputLayer(input_shape=(28, 28, 1)))
encoder.add(layers.Conv2D(16, (3, 3), activation='relu', padding='same'))
encoder.add(layers.MaxPooling2D((2, 2), padding='same'))
encoder.add(layers.Conv2D(8, (3, 3), activation='relu', padding='same'))
encoder.add(layers.MaxPooling2D((2, 2), padding='same'))
encoder.add(layers.Conv2D(4, (3, 3), activation='relu', padding='same'))
encoder.add(layers.MaxPooling2D((2, 2), padding='same'))

# Print the encoder summary
encoder.summary()

# d. Decoder networks convert it back to the original input

# Create a decoder model
decoder = models.Sequential()
decoder.add(layers.InputLayer(input_shape=(4, 4, 4)))  # Latent representation shape
decoder.add(layers.Conv2D(4, (3, 3), activation='relu', padding='same'))
decoder.add(layers.UpSampling2D((2, 2)))
```

```
decoder.add(layers.Conv2D(8, (3, 3), activation='relu', padding='same'))

decoder.add(layers.UpSampling2D((2, 2)))

decoder.add(layers.Conv2D(16, (3, 3), activation='relu'))

decoder.add(layers.UpSampling2D((2, 2)))

decoder.add(layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same'))


# Print the decoder summary

decoder.summary()


# e. Compile the models with Optimizer, Loss, and Evaluation Metrics


# Create the autoencoder model

autoencoder = models.Sequential([encoder, decoder])


# Compile the autoencoder

autoencoder.compile(optimizer='adam', loss='mean_squared_error')


# Train the autoencoder on normal data

autoencoder.fit(train_images, train_images, epochs=10, batch_size=128, shuffle=True,
validation_data=(test_images, test_images))


# Use the trained autoencoder to reconstruct test images

reconstructed_images = autoencoder.predict(test_images)


# Calculate the Mean Squared Error (MSE) between original and reconstructed images

mse = np.mean(np.square(test_images - reconstructed_images))

print(f'Mean Squared Error on Test Data: {mse:.4f}')
```

FNN

```
# a. Import the necessary packages

import tensorflow as tf
```

```
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt

# b. Load the training and testing data (MNIST)
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

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

# Flatten the images (convert 28x28 images to a 1D array of 784)
train_images = train_images.reshape((60000, 28 * 28))
test_images = test_images.reshape((10000, 28 * 28))

# One-hot encode the labels
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

# c. Define the network architecture using Keras
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(10, activation='softmax'))  # 10 classes for digits 0-9

# Print the model summary
model.summary()

# d. Train the model using SGD
```

```python
model.compile(optimizer='sgd',
            loss='categorical_crossentropy',
            metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=20, batch_size=128,
validation_data=(test_images, test_labels))

# e. Evaluate the network
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f"Test Accuracy: {test_acc*100:.2f}%")
print(f"Test Loss: {test_loss:.4f}")

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

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

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

```
plt.legend()

plt.tight_layout()
plt.show()
```