2.Implement Shallow Neural Network model:

- **Implement a binary classifi cation neural network with a single and multiple hidden layers.**
- **Implement a Multi-class classifi cation neural network with a single and multiple hidden layers.**
- **Vary the number of neurons at suitable layers**

```
import numpy as np

import matplotlib.pyplot as plt

import tensorflow as tf

from sklearn.model_selection import train_test_split

# Load MNIST dataset

(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()

# Normalize pixel values to between 0 and 1

X_train = X_train.astype('float32') / 255.0

X_test = X_test.astype('float32') / 255.0

# Flatten the images

X_train_flat = X_train.reshape((-1, 28 * 28))

X_test_flat = X_test.reshape((-1, 28 * 28))

# Define models

models = [

  # Binary Classification with Single Hidden Layer

  tf.keras.Sequential([

    tf.keras.layers.Flatten(input_shape=(28, 28)),

    tf.keras.layers.Dense(64, activation='relu'),

    tf.keras.layers.Dense(1, activation='sigmoid')

  ]),

  # Multi-class Classification with Single Hidden Layer

  tf.keras.Sequential([

    tf.keras.layers.Flatten(input_shape=(28, 28)),

    tf.keras.layers.Dense(64, activation='relu'),

    tf.keras.layers.Dense(10, activation='softmax')

  ]),

  # Binary Classification with Multiple Hidden Layers
```

```
    tf.keras.Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(1, activation='sigmoid')
    ]),
    # Multi-class Classification with Multiple Hidden Layers
    tf.keras.Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(10, activation='softmax')
    ])
]
# Compile and train models
for i, model in enumerate(models):
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy' if i % 2 != 0 else 'binary_crossentropy',
                  metrics=['accuracy'])
    history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2)
    loss, accuracy = model.evaluate(X_test, y_test)
    print("Test Accuracy for Model", i+1, ":", accuracy)
    plt.plot(history.history['accuracy'], label='Training Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title('Model ' + str(i+1) + ' Training Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()
```

3. Hyper parameter Tuning of a Neural Network model implemented for hand-written digit classifi cation:

● **Vary the type of activation functions.**

● **Choose suitable Loss functions.**

● **Vary the number of neurons at suitable layers.**

● **Vary Weight Initialization methods.**

● **Save the Best Model and load the saved model.**

```
import numpy as np

import matplotlib.pyplot as plt

import tensorflow as tf

from sklearn.model_selection import train_test_split

# Load MNIST dataset

(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()

# Normalize pixel values to between 0 and 1

X_train = X_train.astype('float32') / 255.0

X_test = X_test.astype('float32') / 255.0

# Flatten the images

X_train_flat = X_train.reshape((-1, 28 * 28))

X_test_flat = X_test.reshape((-1, 28 * 28))

# Split the data into training and validation sets

X_train, X_val, y_train, y_val = train_test_split(X_train_flat, y_train, test_size=0.2,
random_state=42)

# Define hyperparameters

activation_functions = ['relu', 'sigmoid', 'tanh']

loss_functions = ['sparse_categorical_crossentropy', 'binary_crossentropy']

neurons = [32, 64, 128]

weight_initializations = ['random_uniform', 'glorot_uniform']

best_val_accuracy = 0

best_model = None

# Perform hyperparameter tuning

for activation in activation_functions:

    for loss_fn in loss_functions:

        for neuron in neurons:

            for weight_init in weight_initializations:

                # Define the model
```

```
            model = tf.keras.Sequential([
                tf.keras.layers.Flatten(input_shape=(28, 28)),
                tf.keras.layers.Dense(neuron, activation=activation, kernel_initializer=weight_init),
                tf.keras.layers.Dense(10, activation='softmax')
            ])
            # Compile the model
            model.compile(optimizer='adam', loss=loss_fn, metrics=['accuracy'])

            # Train the model
            history = model.fit(X_train, y_train, epochs=5, validation_data=(X_val, y_val), verbose=0)
            # Evaluate the model on validation set
            val_loss, val_accuracy = model.evaluate(X_val, y_val, verbose=0)
            # Save the best model
            if val_accuracy > best_val_accuracy:
                best_val_accuracy = val_accuracy
                best_model = model
            print("Activation Function:", activation, "| Loss Function:", loss_fn, "| Neurons:", neuron, "|
Weight Initialization:", weight_init)
            print("Validation Accuracy:", val_accuracy)
# Save the best model
best_model.save("best_model.h5")
print("Best model saved!")
# Load the saved model
loaded_model = tf.keras.models.load_model("best_model.h5")
print("Best model loaded!")
```

4.Building a Deep Neural Network:

- **Implement a multi-class classifi cation neural network with number of layers of your choice.**
- **Include Batch Normalization layers.**
- **Vary Optimization methods.**
- **Add drop out layers.**

```
import numpy as np
import matplotlib.pyplot as plt
```

```
import tensorflow as tf
from sklearn.model_selection import train_test_split
# Load MNIST dataset
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()
# Normalize pixel values to between 0 and 1
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0
# Flatten the images
X_train_flat = X_train.reshape((-1, 28 * 28))
X_test_flat = X_test.reshape((-1, 28 * 28))
# Convert labels to one-hot encoding
y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)
# Split the data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train_flat, y_train, test_size=0.2,
random_state=42)
# Define the model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(128, activation='relu', input_shape=(28*28,)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation='softmax')
])
# Compile the model with different optimization methods
optimizers = ['adam', 'rmsprop', 'sgd']
for optimizer in optimizers:
    print("\nOptimizer:", optimizer)
    model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
    # Train the model
```

```
    history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_val, y_val), verbose=1)
    # Evaluate the model on the test set
    loss, accuracy = model.evaluate(X_test_flat, y_test)
    print("Test Accuracy:", accuracy)
    # Plot the training history
    plt.plot(history.history['accuracy'], label='Training Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title('Model Training Accuracy with ' + optimizer.capitalize() + ' Optimizer')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()
```

5. Convolutional Neural Network Models.

- **Design a Convolutional neural network with the layers of your choice**
- **Compare the performance by changing the**
- **Kernel size**
- **Number of feature maps at each convolutional layer**
- **Stride.**
- **Padding.**
- **Number of fully connected layers.**

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn.model_selection import train_test_split
# Load MNIST dataset
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()
# Normalize pixel values to between 0 and 1
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0
# Reshape the images to add a channel dimension
X_train = np.expand_dims(X_train, axis=-1)
```

```
X_test = np.expand_dims(X_test, axis=-1)
# Convert labels to one-hot encoding
y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)
# Split the data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2, random_state=42)
# Define different CNN architectures

# CNN 1: Varying kernel size, feature maps, stride, padding, and fully connected layers
model1 = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
# CNN 2: Varying kernel size and feature maps
model2 = tf.keras.Sequential([
    tf.keras.layers.Conv2D(16, (5, 5), activation='relu', input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
# Compile and train the models
models = [(model1, "CNN 1"), (model2, "CNN 2")]
test_accuracies = []
for model, name in models:
```

```
    model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
    history = model.fit(X_train, y_train, epochs=5, batch_size=32, validation_data=(X_val, y_val),
verbose=1)
    # Evaluate the model on the test set
    loss, accuracy = model.evaluate(X_test, y_test)
    test_accuracies.append(accuracy)
    print(name + " - Test Accuracy:", accuracy)
    # Plot the training history
    plt.plot(history.history['accuracy'], label='Training Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title(name + " - Model Training Accuracy")
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()
# Compare test accuracies
plt.bar(['CNN 1', 'CNN 2'], test_accuracies)
plt.title('Comparison of Test Accuracies')
plt.xlabel('CNN Architecture')
plt.ylabel('Test Accuracy')
plt.show()
```

6. Visualization of CNN Models.

- **Design a Convolutional Neural Network Model for image classifi cation.**
- **Plot Model Architecture.**
- **Visualize feature maps after training of CNN.**
- **Visualize class activation maps.**

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn.model_selection import train_test_split
from tensorflow.keras.applications.vgg16 import VGG16
```

```
from tensorflow.keras.models import Model

# Load MNIST dataset
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()
# Normalize pixel values to between 0 and 1
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0
# Reshape the images to add a channel dimension
X_train = np.expand_dims(X_train, axis=-1)
X_test = np.expand_dims(X_test, axis=-1)
# Convert labels to one-hot encoding
y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)
# Split the data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2, random_state=42)
# Define the CNN model
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
# Train the model
history = model.fit(X_train, y_train, epochs=5, batch_size=32, validation_data=(X_val, y_val),
verbose=1)
# Plot the model architecture
tf.keras.utils.plot_model(model, to_file='model_architecture.png', show_shapes=True,
show_layer_names=True)
```

```
# Visualize feature maps
sample_image = X_test[0]
sample_image = np.expand_dims(sample_image, axis=0)
layer_outputs = [layer.output for layer in model.layers]
activation_model = Model(inputs=model.input, outputs=layer_outputs)
activations = activation_model.predict(sample_image)
for i, activation in enumerate(activations):
    if len(activation.shape) == 4:
        plt.figure(figsize=(10, 10))
        for j in range(activation.shape[3]):
            plt.subplot(8, 8, j+1)
            plt.imshow(activation[0, :, :, j], cmap='viridis')
            plt.axis('off')
        plt.savefig('feature_maps_layer_{}.png'.format(i))
        plt.close()
# Visualize class activation maps
class_weights = model.layers[-1].get_weights()[0]
final_conv_layer = model.layers[-3]
heatmap_model = Model(inputs=model.input,
                      outputs=(final_conv_layer.output, model.output))
with tf.GradientTape() as tape:
    conv_outputs, predictions = heatmap_model(sample_image)
    loss = predictions[:, np.argmax(predictions[0])]
grads = tape.gradient(loss, conv_outputs)
pooled_grads = tf.reduce_mean(grads, axis=(0, 1, 2))
heatmap = tf.reduce_mean(tf.multiply(pooled_grads, conv_outputs), axis=-1)
heatmap = np.maximum(heatmap, 0)
heatmap /= np.max(heatmap)
heatmap = heatmap[0]
plt.matshow(heatmap)
plt.savefig('class_activation_map.png')
```

```
plt.close()
```

module 2

1.Using Deep pre-trained CNN model for feature extraction:

● Extract features from the FC1 of VGG network.

● Train any traditional ML model like SVM for classifi cation.

● Repeat the above by considering FC2 of VGG for feature extraction.

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.datasets import mnist
# Load MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# Normalize pixel values to between 0 and 1
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0
# Reshape the images to add a channel dimension
X_train = np.expand_dims(X_train, axis=-1)
X_test = np.expand_dims(X_test, axis=-1)
# Load pre-trained VGG16 model without top (fully connected) layers
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
# Add fully connected layers on top of VGG16 base model
x = Flatten()(base_model.output)
fc1 = Dense(4096, activation='relu', name='fc1')(x)
fc2 = Dense(4096, activation='relu', name='fc2')(fc1)
# Create models for extracting features from FC1 and FC2 layers
feature_extractor_fc1 = Model(inputs=base_model.input, outputs=fc1)
```

```
feature_extractor_fc2 = Model(inputs=base_model.input, outputs=fc2)
# Resize images to fit VGG16 input shape
X_train_resized = tf.image.resize(X_train, (224, 224)).numpy()
X_test_resized = tf.image.resize(X_test, (224, 224)).numpy()
# Extract features from FC1 and FC2 layers
X_train_features_fc1 = feature_extractor_fc1.predict(X_train_resized)
X_test_features_fc1 = feature_extractor_fc1.predict(X_test_resized)
X_train_features_fc2 = feature_extractor_fc2.predict(X_train_resized)
X_test_features_fc2 = feature_extractor_fc2.predict(X_test_resized)
# Flatten the features
X_train_features_fc1_flat = X_train_features_fc1.reshape((X_train_features_fc1.shape[0], -1))
X_test_features_fc1_flat = X_test_features_fc1.reshape((X_test_features_fc1.shape[0], -1))
X_train_features_fc2_flat = X_train_features_fc2.reshape((X_train_features_fc2.shape[0], -1))
X_test_features_fc2_flat = X_test_features_fc2.reshape((X_test_features_fc2.shape[0], -1))
# Train SVM model using features extracted from FC1
svm_fc1 = SVC()
svm_fc1.fit(X_train_features_fc1_flat, y_train)
y_pred_fc1 = svm_fc1.predict(X_test_features_fc1_flat)
accuracy_fc1 = accuracy_score(y_test, y_pred_fc1)
print("Accuracy using features from FC1:", accuracy_fc1)
# Train SVM model using features extracted from FC2
svm_fc2 = SVC()
svm_fc2.fit(X_train_features_fc2_flat, y_train)
y_pred_fc2 = svm_fc2.predict(X_test_features_fc2_flat)
accuracy_fc2 = accuracy_score(y_test, y_pred_fc2)
print("Accuracy using features from FC2:", accuracy_fc2)
```

2. Fine-tuning Deep pre-trained CNN for Classifi cation:

- **Fine-tune VGG network for the task under consideration.**
- **Check the performance by making.**
- **all the layers trainable.**
- **freezing the initial layers.**

● **freezing the entire network except the fi nal layer.**

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.applications import VGG16
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import SGD
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from tensorflow.keras.datasets import mnist
# Load MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# Normalize pixel values to between 0 and 1
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0
# Resize images to fit VGG16 input shape
X_train_resized = tf.image.resize(X_train[..., tf.newaxis], (224, 224))
X_test_resized = tf.image.resize(X_test[..., tf.newaxis], (224, 224))
# Convert labels to one-hot encoding
y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)
# Load pre-trained VGG16 model without top (fully connected) layers
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
# Fine-tuning with all layers trainable
model_all_trainable = Sequential([
    base_model,
    Flatten(),
    Dense(512, activation='relu'),
    Dense(10, activation='softmax')
])
```

```
# Compile the model

model_all_trainable.compile(optimizer=SGD(lr=0.001), loss='categorical_crossentropy',
metrics=['accuracy'])

# Train the model

history_all_trainable = model_all_trainable.fit(X_train_resized, y_train, epochs=5, batch_size=32,
validation_split=0.2)

# Evaluate the model

y_pred_all_trainable = model_all_trainable.predict(X_test_resized)

accuracy_all_trainable = accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_pred_all_trainable,
axis=1))

print("Accuracy with all layers trainable:", accuracy_all_trainable)

# Freezing the initial layers and fine-tuning only the later layers

for layer in base_model.layers[:15]:

    layer.trainable = False

model_fine_tune = Sequential([

    base_model,

    Flatten(),

    Dense(512, activation='relu'),

    Dense(10, activation='softmax')

])

# Compile the model

model_fine_tune.compile(optimizer=SGD(lr=0.001), loss='categorical_crossentropy',
metrics=['accuracy'])

# Train the model

history_fine_tune = model_fine_tune.fit(X_train_resized, y_train, epochs=5, batch_size=32,
validation_split=0.2)

# Evaluate the model

y_pred_fine_tune = model_fine_tune.predict(X_test_resized)

accuracy_fine_tune = accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_pred_fine_tune,
axis=1))

print("Accuracy with initial layers frozen:", accuracy_fine_tune)

# Freezing the entire network except the final layer

for layer in base_model.layers:

    layer.trainable = False
```

```
model_final_layer_trainable = Sequential([
    base_model,
    Flatten(),
    Dense(512, activation='relu'),
    Dense(10, activation='softmax')
])
# Compile the model
model_final_layer_trainable.compile(optimizer=SGD(lr=0.001), loss='categorical_crossentropy',
metrics=['accuracy'])
# Train the model
history_final_layer_trainable = model_final_layer_trainable.fit(X_train_resized, y_train, epochs=5,
batch_size=32, validation_split=0.2)
# Evaluate the model
y_pred_final_layer_trainable = model_final_layer_trainable.predict(X_test_resized)
accuracy_final_layer_trainable = accuracy_score(np.argmax(y_test, axis=1),
np.argmax(y_pred_final_layer_trainable, axis=1))
print("Accuracy with only final layer trainable:", accuracy_final_layer_trainable)
```

3. Design MLFFNN with 3-level stacked autoencoder based pre-training for Black and white image data, Display features extracted by diff erent levels of stacked autoencoder at the end of pre-training.

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
# Generate dummy black and white image data
num_samples = 1000
image_size = 28 * 28
X_train = np.random.randint(0, 256, size=(num_samples, image_size))
# Normalize pixel values to between 0 and 1
X_train = X_train.astype('float32') / 255.0
```

```
# Define stacked autoencoder architecture
input_img = Input(shape=(image_size,))
encoded1 = Dense(256, activation='relu')(input_img)
encoded2 = Dense(128, activation='relu')(encoded1)
encoded3 = Dense(64, activation='relu')(encoded2)
decoded1 = Dense(128, activation='relu')(encoded3)
decoded2 = Dense(256, activation='relu')(decoded1)
decoded3 = Dense(image_size, activation='sigmoid')(decoded2)
# Create autoencoder models
autoencoder1 = Model(input_img, decoded1)
autoencoder2 = Model(input_img, decoded2)
autoencoder3 = Model(input_img, decoded3)
# Compile autoencoder models
autoencoder1.compile(optimizer='adam', loss='mean_squared_error')
autoencoder2.compile(optimizer='adam', loss='mean_squared_error')
autoencoder3.compile(optimizer='adam', loss='mean_squared_error')
# Train autoencoder models
history1 = autoencoder1.fit(X_train, X_train, epochs=10, batch_size=32, shuffle=True,
validation_split=0.2)
history2 = autoencoder2.fit(X_train, X_train, epochs=10, batch_size=32, shuffle=True,
validation_split=0.2)
history3 = autoencoder3.fit(X_train, X_train, epochs=10, batch_size=32, shuffle=True,
validation_split=0.2)
# Extract features from each level of stacked autoencoder
features1 = autoencoder1.predict(X_train)
features2 = autoencoder2.predict(X_train)
features3 = autoencoder3.predict(X_train)
# Define MLFFNN architecture using pre-trained weights from stacked autoencoder
input_features = Input(shape=(image_size,))
encoded_features1 = autoencoder1.layers[1](input_features)
encoded_features2 = autoencoder2.layers[1](encoded_features1)
encoded_features3 = autoencoder3.layers[1](encoded_features2)
output = Dense(10, activation='softmax')(encoded_features3)
```

```
# Create MLFFNN model
mlffnn = Model(input_features, output)
# Compile MLFFNN model
mlffnn.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
# Display features extracted by different levels of stacked autoencoder
plt.figure(figsize=(10, 10))
plt.subplot(1, 3, 1)
plt.title('Features Extracted by Level 1 Autoencoder')
plt.imshow(features1[0].reshape(16, 16), cmap='gray')
plt.subplot(1, 3, 2)
plt.title('Features Extracted by Level 2 Autoencoder')
plt.imshow(features2[0].reshape(8, 16), cmap='gray')
plt.subplot(1, 3, 3)
plt.title('Features Extracted by Level 3 Autoencoder')
plt.imshow(features3[0].reshape(8, 8), cmap='gray')
plt.show()
```

4. Sentiment Analysis

- **Pre-process the text.**
- **Convert the text into word embeddings.**
- **Implement the classifi cation network using LSTMs/ GRUs.**
- **Pre-process the text.**
- **Convert the text into word embeddings.**
- **Implement the classifi cation network using LSTMs/ GRUs.**

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense
from sklearn.model_selection import train_test_split
# Dummy data (replace with your dataset)
```

```
texts = ["I love this movie", "This movie is fantastic", "Great movie, highly recommended",
         "Awful movie, waste of time", "I hated this film", "The worst movie ever"]
labels = [1, 1, 1, 0, 0, 0]  # 1 for positive sentiment, 0 for negative sentiment
# Pre-process the text
max_words = 1000  # Maximum number of words to tokenize
max_len = 20  # Maximum length of each sequence
tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
X = pad_sequences(sequences, maxlen=max_len)
y = np.array(labels)
# Convert the text into word embeddings
embedding_dim = 50  # Dimension of word embeddings
vocab_size = len(tokenizer.word_index) + 1
# Define the LSTM model
model = Sequential()
model.add(Embedding(vocab_size, embedding_dim, input_length=max_len))
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))
# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test), verbose=1)
```