1. Develop a neural network model to classify images of handwritten digits (0-9) from the MNIST dataset into their corresponding digit categories with high accuracy.

AIM:

To develop a neural network model that can classify images of handwritten digits (0-9) from the MNIST dataset into their corresponding digit categories with high accuracy.

Step 1: Prepare the Data

The first step in developing a neural network model for handwritten digit classification is to prepare the data. This involves:

Collecting Data: Gathering a large dataset of images of handwritten digits (0-9). The MNIST dataset is a popular choice for this task.

Pre-processing Data: Normalizing the pixel values of the images to a common scale, typically between 0 and 1. This helps the neural network learn features that are not dependent on the scale of the input data.

Splitting Data: Dividing the dataset into training , validation , and testing sets. The training set is used to train the model, the validation set is used to evaluate the model during training, and the testing set is used to evaluate the final performance of the model.

Step 2: Design the Neural Network Architecture

The next step is to design the neural network architecture. For handwritten digit classification, a Convolutional Neural Network (CNN) is a suitable choice. A CNN typically consists of:

Convolutional Layers: These layers apply filters to the input data to extract features. Multiple convolutional layers can be stacked to extract features at different scales.

Pooling Layers: These layers down sample the feature maps to reduce the spatial dimensions and retain the most important information.

Dense Layers: These layers are used for classification. The output of the convolutional and pooling layers is flattened and fed into one or more dense layers to produce the final output.

Step 3: Compile the Model

Once the neural network architecture is defined, the next step is to compile the model. This involves:

Loss Function: Choosing a loss function that measures the difference between the model's predictions and the true labels. For classification problems, the categorical cross-entropy loss function is commonly used.

Optimizer: Choosing an optimization algorithm that updates the model's weights to minimize the loss function. The Adam optimizer is a popular choice for many deep learning tasks.

Step 4: Train the Model

The next step is to train the model using the training data. This involves:

Batching: Dividing the training data into batches to reduce memory usage and speed up training.

Epochs: Training the model for multiple epochs, where each epoch involves passing the entire training dataset through the model once.

Step 5: Evaluate the Model

After training the model, the next step is to evaluate its performance on the testing data. This involves:

Metrics: Choosing metrics to evaluate the model's performance, such as accuracy, precision, recall, and F1 score.

Evaluation: Passing the testing data through the model and calculating the chosen metrics.

CODE:

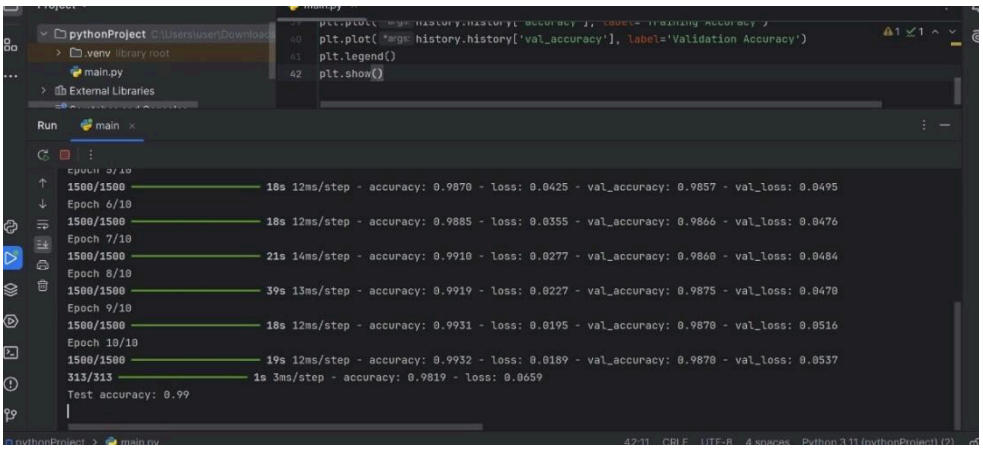
```
import numpy as np

import tensorflow as tf

from tensorflow import keras

from sklearn.model_selection import train_test_split

import matplotlib.pyplot as plt


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


X_train = X_train.reshape(-1, 28, 28, 1).astype('float32') / 255.0

X_test = X_test.reshape(-1, 28, 28, 1).astype('float32') / 255.0


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


model = keras.Sequential([

   keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
```

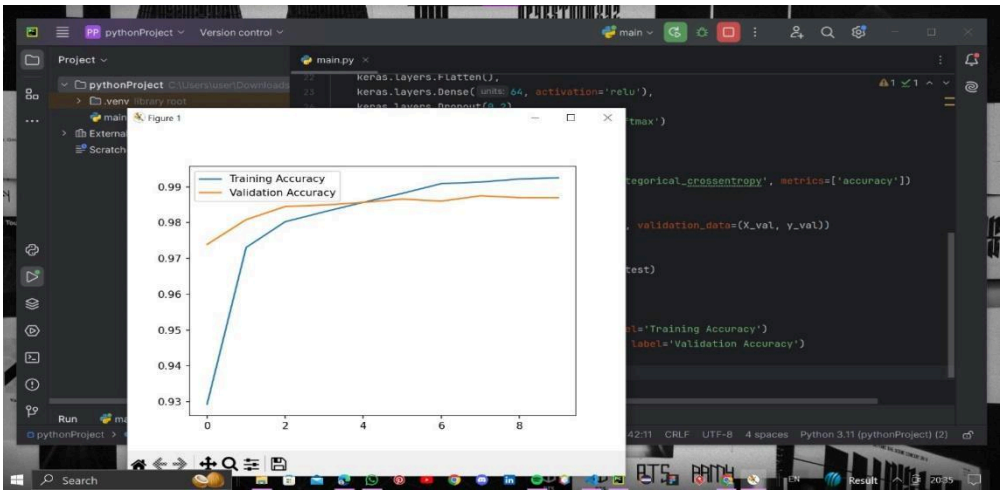
```
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Flatten(),
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
history = model.fit(X_train, y_train, epochs=10, validation_data=(X_val, y_val))
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f'Test accuracy: {test_acc:.2f}')
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.legend()
plt.show()
```

HAND WRITTEN INPUT:

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

OUTPUT:





RESULT:

Hence , the neural network model classifies handwritten digits (0-9) from the MNIST dataset with high accuracy by learning features from images and mapping them to correct digit labels.

2. IMPLEMENT AN AUTOENCODER TO REMOVE NOISE FROM GRAYSCALE IMAGES. THE GOAL IS TO TRAIN THE MODEL TO RECONSTRUCT CLEAN IMAGES FROM NOISY VERSIONS.

AIM:

To implement an autoencoder to remove noise from grayscale images by training the model to reconstruct clean images from noisy version.

STEP-BY-STEP ALGORITHM FOR DENOISING AUTOENCODER:

1. **Prepare the Data:**
    - **Collect Data:** Obtain a dataset of clean grayscale images. For example, you might use the MNIST dataset or any collection of grayscale images.
    - **Add Noise:** Create noisy versions of these images. You can do this by adding random noise (e.g., Gaussian noise) to each image. This noisy image will be the input to your autoencoder, while the clean image will be the target output.
    - **Split Data:** Divide your dataset into training, validation, and testing sets. For instance, you might use 80% for training, 10% for validation, and 10% for testing.
2. **Design the Autoencoder Architecture:**
    - **Encoder:**
        - **Input Layer:** Accepts the noisy grayscale image.
        - **Hidden Layers:** Use a few convolutional or dense layers to compress the image into a lower-dimensional representation (bottleneck). These layers will learn the most important features of the image.
    - **Bottleneck Layer:** The compressed representation of the image.
    - **Decoder:**
        - **Hidden Layers:** Use a few layers to reconstruct the image from the compressed representation. The goal here is to produce an image as close as possible to the original clean image.
        - **Output Layer:** Produces the final image reconstruction.

3. **Compile the Model:**
    - **Loss Function:** Choose a loss function that measures how close the reconstructed image is to the clean image. Mean Squared Error (MSE) is commonly used for this purpose.
    - **Optimizer:** Select an optimization algorithm (like Adam or SGD) to update the model's weights during training.
4. **Train the Model:**
    - **Feed Noisy Images:** Pass the noisy images through the autoencoder.
    - **Calculate Loss:** Compute the loss between the output of the autoencoder (reconstructed image) and the clean image.
    - **Update Weights:** Use backpropagation and the optimizer to minimize the loss function by adjusting the model's weights.
    - **Validation:** Monitor the model's performance on the validation set to ensure it's not overfitting.
5. **Evaluate the Model:**
    - **Test the Model:** After training, test the autoencoder using the test set to see how well it removes noise from new images.
    - **Assess Performance:** Check metrics like MSE, PSNR (Peak Signal-to-Noise Ratio), or visually inspect the results to evaluate the quality of the denoised images.
6. **Fine-Tune and Optimize:**
    - **Adjust Hyperparameters:** If the performance is not satisfactory, you might need to tweak hyperparameters (e.g., learning rate, number of layers, etc.).
    - **Regularization:** Implement techniques like dropout or batch normalization if necessary to improve generalization and prevent overfitting.
7. **Deploy the Model:**
    - **Integration:** Once satisfied with the model's performance, integrate it into your application or system where you need to denoise grayscale images.
    - **Continuous Monitoring:** Keep an eye on how the model performs in real-world scenarios and retrain it periodically if needed.

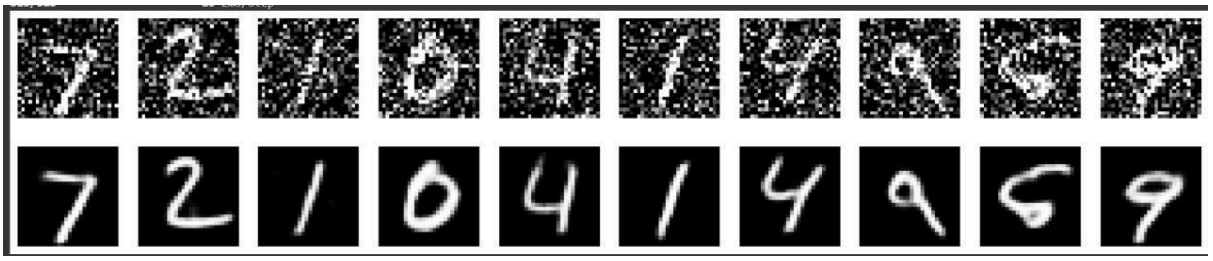
CODE:

```
import numpy as np
```

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

(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))
noise_factor = 0.5
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0,
size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0, scale=1.0,
size=x_test.shape)
x_train_noisy = np.clip(x_train_noisy, 0.0, 1.0)
x_test_noisy = np.clip(x_test_noisy, 0.0, 1.0)
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(32, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(32, (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, x_train, epochs=50, batch_size=128, shuffle=True,
validation_data=(x_test_noisy, x_test))

decoded_imgs = autoencoder.predict(x_test_noisy)

n = 10

plt.figure(figsize=(20, 4))

for i in range(n):

    ax = plt.subplot(2, n, i + 1)

    plt.imshow(x_test_noisy[i].reshape(28, 28), cmap='gray')

    plt.axis('off')

    ax = plt.subplot(2, n, i + 1 + n)

    plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='gray')

    plt.axis('off')


plt.show()
```

THE WHOLE PROCESS IN ONE IMAGE:



RESULT:

Hence , the autoencoder removes noise from grayscale images by training the model to reconstruct clean images from noisy versions.