Lab Assignment - 5

Mrinal Bhan DSAI - 211020428

Task - A

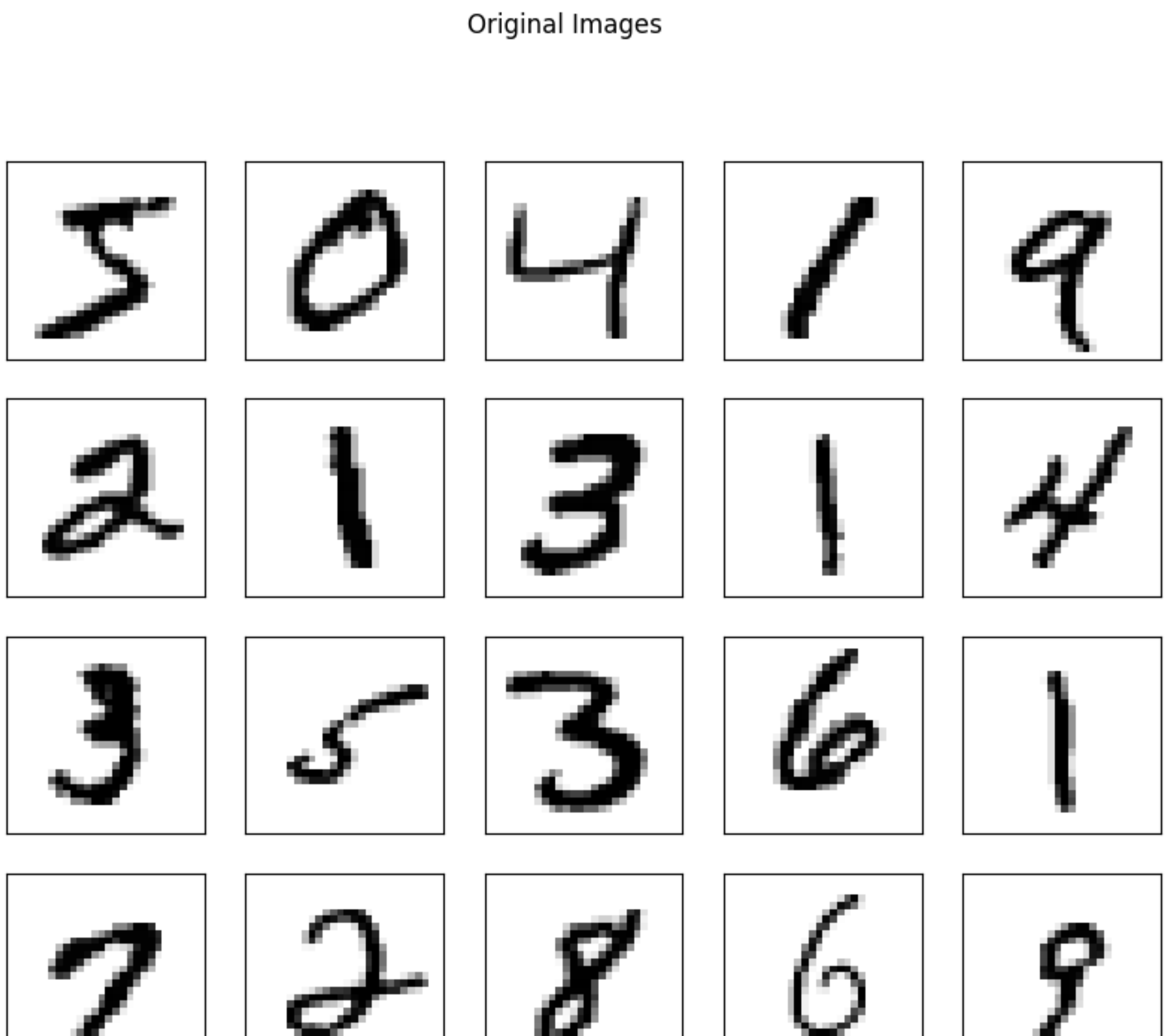
1. Normalize the dataset with at least two type of normalization techniques.

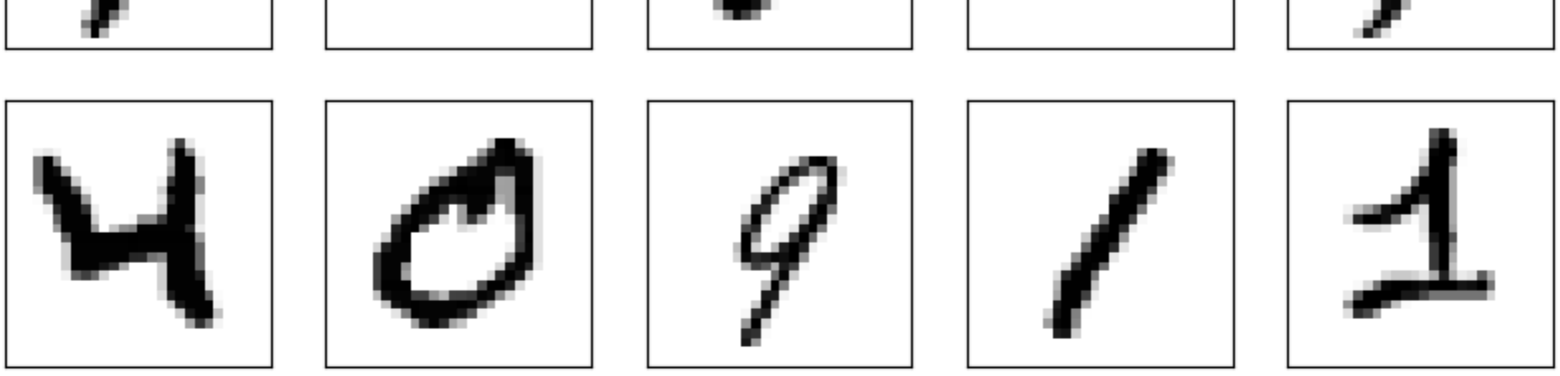
```
import numpy as np
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from tensorflow.keras.datasets import mnist

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

# Flatten the images for normalization
train_images = train_images.reshape((-1, 784))
test_images = test_images.reshape((-1, 784))


scaler_minmax = MinMaxScaler()
train_images_minmax = scaler_minmax.fit_transform(train_images)
test_images_minmax = scaler_minmax.transform(test_images)


scaler_zscore = StandardScaler()
train_images_zscore = scaler_zscore.fit_transform(train_images)
test_images_zscore = scaler_zscore.transform(test_images)

```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[37], line 14
      8 # # Flatten the images for normalization
      9 # train_images = train_images.reshape((-1, 784))
     10 # test_images = test_images.reshape((-1, 784))
     13 scaler_minmax = MinMaxScaler()
---> 14 train_images_minmax = scaler_minmax.fit_transform(train_images)
     15 test_images_minmax = scaler_minmax.transform(test_images)
     18 scaler_zscore = StandardScaler()

File ~\AppData\Roaming\Python\Python311\site-packages\sklearn\utils\_set_output.py:140, in _wrap_method_output.<locals>.wrapped(self, X, *args, **kwargs)
    138 @wraps(f)
    139 def wrapped(self, X, *args, **kwargs):
--> 140     data_to_wrap = f(self, X, *args, **kwargs)
    141     if isinstance(data_to_wrap, tuple):
    142         # only wrap the first output for cross decomposition
    143         return (
    144             _wrap_data_with_container(method, data_to_wrap[0], X, self),
    145             *data_to_wrap[1:],
    146         )

File ~\AppData\Roaming\Python\Python311\site-packages\sklearn\base.py:878, in TransformerMixin.fit_transform(self, X, y, **fit_params)
    874 # non-optimized default implementation; override when a better
    875 # method is possible for a given clustering algorithm
    876 if y is None:
    877     # fit method of arity 1 (unsupervised transformation)
--> 878     return self.fit(X, **fit_params).transform(X)
    879 else:
    880     # fit method of arity 2 (supervised transformation)
    881     return self.fit(X, y, **fit_params).transform(X)

File ~\AppData\Roaming\Python\Python311\site-packages\sklearn\preprocessing\_data.py:427, in MinMaxScaler.fit(self, X, y)
    425 # Reset internal state before fitting
    426 self._reset()
--> 427 return self.partial_fit(X, y)

File ~\AppData\Roaming\Python\Python311\site-packages\sklearn\preprocessing\_data.py:466, in MinMaxScaler.partial_fit(self, X, y)
    460     raise TypeError(
    461         "MinMaxScaler does not support sparse input. "
    462         "Consider using MaxAbsScaler instead."
    463     )
    465 first_pass = not hasattr(self, "n_samples_seen_")
--> 466 X = self._validate_data(
    467     X,
    468     reset=first_pass,
    469     dtype=FLOAT_DTYPES,
    470     force_all_finite="allow-nan",
    471 )
    473 data_min = np.nanmin(X, axis=0)
```

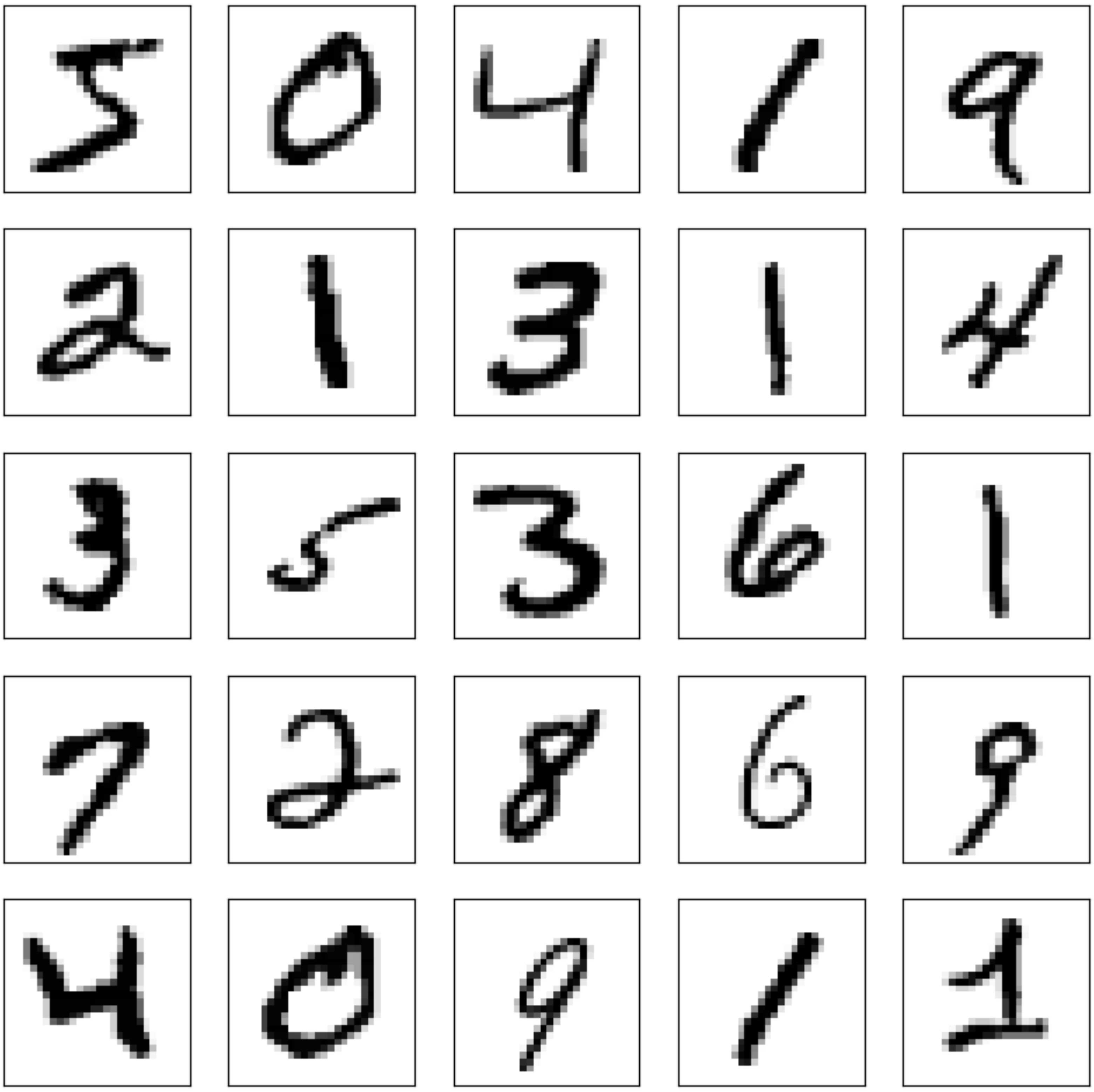
```
    474 data_max = np.nanmax(X, axis=0)

File ~\AppData\Roaming\Python\Python311\site-packages\sklearn\base.py:565, in BaseEstimator._validate_data(self, X, y, reset, validate_separately, **check_params)
    563     raise ValueError("Validation should be done on X, y or both.")
    564 elif not no_val_X and no_val_y:
--> 565     X = check_array(X, input_name="X", **check_params)
    566     out = X
    567 elif no_val_X and not no_val_y:

File ~\AppData\Roaming\Python\Python311\site-packages\sklearn\utils\validation.py:915, in check_array(array, accept_sparse, accept_large_sparse, dtype, order, copy, force_all_finite, ensure_2d, allow_nd, ensure_min_samples, ensure_min_features, estimator, input_name)
    910     raise ValueError(
    911         "dtype='numeric' is not compatible with arrays of bytes/strings."
    912         "Convert your data to numeric values explicitly instead."
    913     )
    914 if not allow_nd and array.ndim >= 3:
--> 915     raise ValueError(
    916         "Found array with dim %d. %s expected <= 2."
    917         % (array.ndim, estimator_name)
    918     )
    920 if force_all_finite:
    921     _assert_all_finite(
    922         array,
    923         input_name=input_name,
    924         estimator_name=estimator_name,
    925         allow_nan=force_all_finite == "allow-nan",
    926     )
```

```python
import matplotlib.pyplot as plt

def display_images(images, title):
    plt.figure(figsize=(10, 10))
    for i in range(25):
        plt.subplot(5, 5, i+1)
        plt.xticks([])
        plt.yticks([])
        plt.grid(False)
        plt.imshow(images[i].reshape(28, 28), cmap=plt.cm.binary)
    plt.suptitle(title)
    plt.show()

display_images(train_images[:25], title="Original Images")
display_images(train_images_minmax[:25], title="Min-Max Scaled Images")
display_images(train_images_zscore[:25], title="Z-score Standardized Images")

```

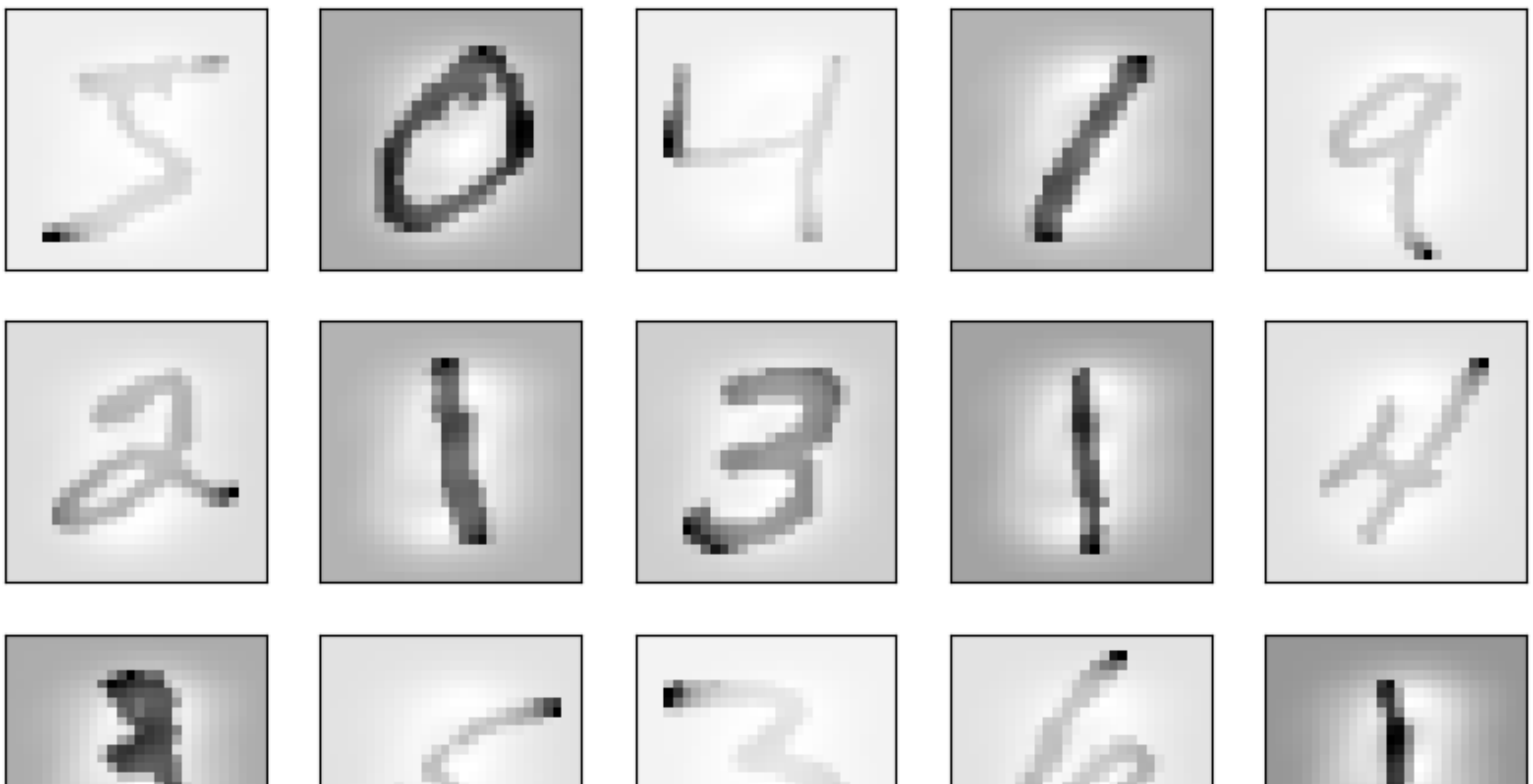




Min-Max Scaled Images



Z-score Standardized Images



2. Create a neural network architecture based auto encoder to classification of data. Typically,it consists of an encoder and a decoder.

```
import tensorflow as tf
import numpy as np
from tensorflow.keras import layers, models


# Define an autoencoder architecture
def create_autoencoder(input_shape, encoding_dim):
    # Encoder
    encoder_input = layers.Input(shape=input_shape)
    encoder = layers.Flatten()(encoder_input)
    encoder = layers.Dense(512, activation='relu')(encoder)
    encoder = layers.Dense(256, activation='relu')(encoder)
    encoder_output = layers.Dense(encoding_dim, activation='relu')(encoder)

    # Decoder
    decoder_input = layers.Input(shape=(encoding_dim,))
    decoder = layers.Dense(256, activation='relu')(decoder_input)
    decoder = layers.Dense(512, activation='relu')(decoder)
    decoder = layers.Dense(np.prod(input_shape), activation='sigmoid')(decoder)
    decoder_output = layers.Reshape(input_shape)(decoder)

    # Create the encoder and decoder models
    encoder_model = models.Model(encoder_input, encoder_output, name='encoder')
    decoder_model = models.Model(decoder_input, decoder_output, name='decoder')

    # Create the autoencoder model by chaining encoder and decoder
    autoencoder_input = layers.Input(shape=input_shape)
    autoencoder_output = decoder_model(encoder_model(autoencoder_input))
    autoencoder_model = models.Model(autoencoder_input, autoencoder_output, name='autoencoder')

    return encoder_model, autoencoder_model

# Define constants
input_shape = train_images_minmax[0].shape  # Shape of input data
encoding_dim = 128  # Dimension of the encoded representation

# Create the autoencoder and encoder models
encoder_model, autoencoder_model = create_autoencoder(input_shape, encoding_dim)
test_images_minmax

# Compile the autoencoder model
autoencoder_model.compile(optimizer='adam', loss='mean_squared_error')

# Train the autoencoder
autoencoder_model.fit(train_images_minmax, train_images_minmax, epochs=10, batch_size=128, shuffle=True, validation_data=(test_images_minmax

# Use encoder for classification
encoded_train_data = encoder_model.predict(train_images_minmax)
encoded_test_data = encoder_model.predict(test_images_minmax)

# Define a classification model (e.g., a simple feedforward network)
classifier = models.Sequential([
    layers.Input(shape=(encoding_dim,)),
    layers.Dense(128, activation='relu'),
    layers.Dense(100, activation='softmax')  # 100 classes for CIFAR-100
])
```

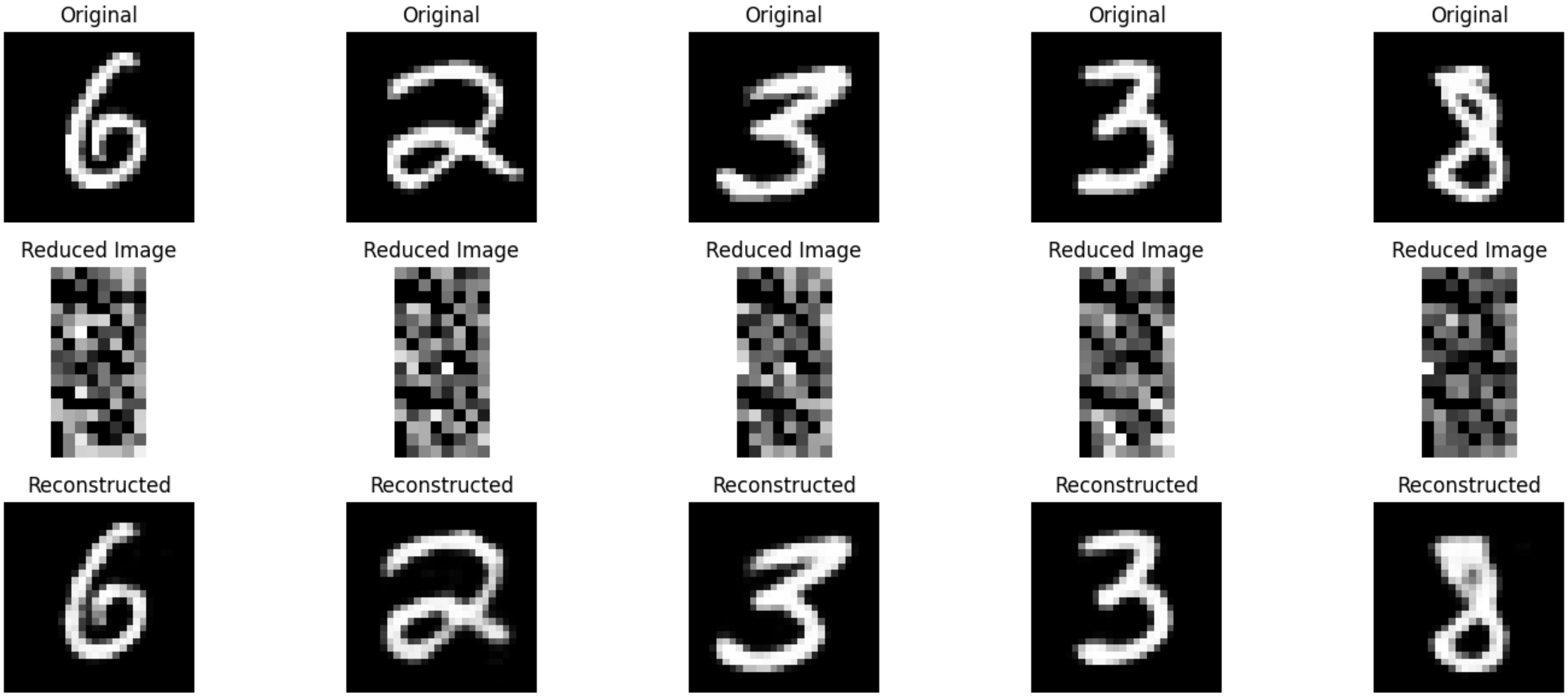
```
# Compile the classifier
classifier.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Train the classifier
classifier.fit(encoded_train_data, train_labels, epochs=10, batch_size=128, shuffle=True, validation_data=(encoded_test_data, test_labels))

# Evaluate the classifier test_labels
test_loss, test_accuracy = classifier.evaluate(encoded_test_data, test_labels)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
```

```
Epoch 1/10
469/469 [==============================] - 10s 20ms/step - loss: 0.0311 - val_loss: 0.0143
Epoch 2/10
469/469 [==============================] - 9s 20ms/step - loss: 0.0110 - val_loss: 0.0091
Epoch 3/10
469/469 [==============================] - 10s 21ms/step - loss: 0.0083 - val_loss: 0.0075
Epoch 4/10
469/469 [==============================] - 10s 21ms/step - loss: 0.0070 - val_loss: 0.0065
Epoch 5/10
469/469 [==============================] - 10s 21ms/step - loss: 0.0061 - val_loss: 0.0059
Epoch 6/10
469/469 [==============================] - 10s 20ms/step - loss: 0.0055 - val_loss: 0.0054
Epoch 7/10
469/469 [==============================] - 9s 20ms/step - loss: 0.0051 - val_loss: 0.0050
Epoch 8/10
469/469 [==============================] - 12s 25ms/step - loss: 0.0047 - val_loss: 0.0047
Epoch 9/10
469/469 [==============================] - 9s 19ms/step - loss: 0.0044 - val_loss: 0.0045
Epoch 10/10
469/469 [==============================] - 9s 20ms/step - loss: 0.0042 - val_loss: 0.0044
1875/1875 [==============================] - 4s 2ms/step
313/313 [==============================] - 1s 2ms/step
Epoch 1/10
469/469 [==============================] - 2s 3ms/step - loss: 0.6939 - accuracy: 0.8047 - val_loss: 0.3473 - val_accuracy: 0.8985
Epoch 2/10
469/469 [==============================] - 1s 2ms/step - loss: 0.3288 - accuracy: 0.9025 - val_loss: 0.2792 - val_accuracy: 0.9182
Epoch 3/10
469/469 [==============================] - 1s 2ms/step - loss: 0.2763 - accuracy: 0.9188 - val_loss: 0.2496 - val_accuracy: 0.9264
Epoch 4/10
469/469 [==============================] - 1s 2ms/step - loss: 0.2367 - accuracy: 0.9306 - val_loss: 0.2184 - val_accuracy: 0.9342
Epoch 5/10
469/469 [==============================] - 1s 2ms/step - loss: 0.2072 - accuracy: 0.9381 - val_loss: 0.1832 - val_accuracy: 0.9465
Epoch 6/10
469/469 [==============================] - 1s 2ms/step - loss: 0.1805 - accuracy: 0.9468 - val_loss: 0.1648 - val_accuracy: 0.9515
Epoch 7/10
469/469 [==============================] - 1s 2ms/step - loss: 0.1639 - accuracy: 0.9515 - val_loss: 0.1618 - val_accuracy: 0.9525
Epoch 8/10
469/469 [==============================] - 1s 2ms/step - loss: 0.1474 - accuracy: 0.9561 - val_loss: 0.1677 - val_accuracy: 0.9483
Epoch 9/10
469/469 [==============================] - 1s 2ms/step - loss: 0.1359 - accuracy: 0.9598 - val_loss: 0.1374 - val_accuracy: 0.9597
Epoch 10/10
469/469 [==============================] - 1s 2ms/step - loss: 0.1244 - accuracy: 0.9630 - val_loss: 0.1233 - val_accuracy: 0.9632
313/313 [==============================] - 0s 1ms/step - loss: 0.1233 - accuracy: 0.9632
Test Accuracy: 96.32%
```

```
import matplotlib.pyplot as plt

# Display a few original, reduced shape, and reconstructed images
n = 5  # Number of images to display

# Select random images from the test set
random_indices = np.random.randint(0, len(test_images_minmax), n)
selected_original_images = test_images_minmax[random_indices]
selected_encoded_images = encoder_model.predict(selected_original_images)
selected_reconstructed_images = autoencoder_model.predict(selected_original_images)

# Create a figure to display the images
plt.figure(figsize=(15, 6))

for i in range(n):
    # Original Images
    ax = plt.subplot(3, n, i + 1)
    plt.imshow(selected_original_images[i].reshape(28, 28), cmap='gray')  # Reshape to (28, 28)
    plt.title("Original")
    plt.axis('off')

    # Reduced Images
    ax = plt.subplot(3, n, i + 1 + n)
    reduced_image = selected_encoded_images[i].reshape(16, 8)  # Assuming encoding_dim is 128
    plt.imshow(reduced_image, cmap='gray')
    plt.title("Reduced Image")
    plt.axis('off')

    # Reconstructed Images
    ax = plt.subplot(3, n, i + 1 + 2 * n)
    plt.imshow(selected_reconstructed_images[i].reshape(28, 28), cmap='gray')  # Reshape to (28, 28)
```

```
    plt.title("Reconstructed")
    plt.axis('off')

plt.tight_layout()
plt.show()

```

```
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 69ms/step
```



Task - B

1. Perform auto encoder where the encoder reduces the input data dimensions, and the decoder aims to reconstruct the original input.

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

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Normalize the data (scale it 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 = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

# Define the architecture of the autoencoder
input_layer = Input(shape=(784,))
encoded = Dense(128, activation='relu')(input_layer)
decoded = Dense(784, activation='sigmoid')(encoded)

autoencoder = Model(input_layer, decoded)

# Compile the model
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

# Train the autoencoder
autoencoder.fit(x_train, x_train, epochs=10, batch_size=256, shuffle=True, validation_data=(x_test, x_test))

# Extract the encoder part of the model
encoder = Model(input_layer, encoded)

# Define a classifier on top of the encoder
```

```
classifier_input = Input(shape=(128,))
classifier_output = Dense(10, activation='softmax')(classifier_input)

classifier = Model(classifier_input, classifier_output)

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

# Train the classifier on the encoded features
encoded_train = encoder.predict(x_train)
encoded_test = encoder.predict(x_test)

classifier.fit(encoded_train, y_train, epochs=10, batch_size=256, shuffle=True, validation_data=(encoded_test, y_test))
```

```
Epoch 1/10
235/235 [==============================] - 3s 10ms/step - loss: 0.2121 - val_loss: 0.1335
Epoch 2/10
235/235 [==============================] - 3s 11ms/step - loss: 0.1170 - val_loss: 0.1020
Epoch 3/10
235/235 [==============================] - 2s 11ms/step - loss: 0.0959 - val_loss: 0.0886
Epoch 4/10
235/235 [==============================] - 2s 10ms/step - loss: 0.0856 - val_loss: 0.0813
Epoch 5/10
235/235 [==============================] - 2s 9ms/step - loss: 0.0799 - val_loss: 0.0771
Epoch 6/10
235/235 [==============================] - 2s 9ms/step - loss: 0.0763 - val_loss: 0.0744
Epoch 7/10
235/235 [==============================] - 2s 9ms/step - loss: 0.0740 - val_loss: 0.0726
Epoch 8/10
235/235 [==============================] - 2s 9ms/step - loss: 0.0723 - val_loss: 0.0711
Epoch 9/10
235/235 [==============================] - 2s 8ms/step - loss: 0.0711 - val_loss: 0.0701
Epoch 10/10
235/235 [==============================] - 2s 8ms/step - loss: 0.0702 - val_loss: 0.0693
1875/1875 [==============================] - 2s 1ms/step
313/313 [==============================] - 0s 1ms/step
Epoch 1/10
235/235 [==============================] - 1s 2ms/step - loss: 2.8983 - accuracy: 0.3436 - val_loss: 1.1480 - val_accuracy: 0.6355
Epoch 2/10
235/235 [==============================] - 0s 2ms/step - loss: 0.8455 - accuracy: 0.7377 - val_loss: 0.5998 - val_accuracy: 0.8157
Epoch 3/10
235/235 [==============================] - 0s 2ms/step - loss: 0.5599 - accuracy: 0.8314 - val_loss: 0.4635 - val_accuracy: 0.8581
Epoch 4/10
235/235 [==============================] - 0s 2ms/step - loss: 0.4595 - accuracy: 0.8629 - val_loss: 0.3976 - val_accuracy: 0.8835
Epoch 5/10
235/235 [==============================] - 0s 2ms/step - loss: 0.4102 - accuracy: 0.8798 - val_loss: 0.3680 - val_accuracy: 0.8946
Epoch 6/10
235/235 [==============================] - 0s 2ms/step - loss: 0.3795 - accuracy: 0.8893 - val_loss: 0.3436 - val_accuracy: 0.9026
Epoch 7/10
235/235 [==============================] - 0s 2ms/step - loss: 0.3606 - accuracy: 0.8949 - val_loss: 0.3341 - val_accuracy: 0.9035
Epoch 8/10
235/235 [==============================] - 0s 2ms/step - loss: 0.3454 - accuracy: 0.9000 - val_loss: 0.3248 - val_accuracy: 0.9090
Epoch 9/10
235/235 [==============================] - 0s 2ms/step - loss: 0.3366 - accuracy: 0.9025 - val_loss: 0.3141 - val_accuracy: 0.9091
Epoch 10/10
235/235 [==============================] - 0s 2ms/step - loss: 0.3289 - accuracy: 0.9057 - val_loss: 0.3131 - val_accuracy: 0.9116
<keras.callbacks.History at 0x1a721f67ed0>
```

2. Apply Common activation functions include ReLU for hidden layers and sigmoid or softmax for the output layer of the decoder.

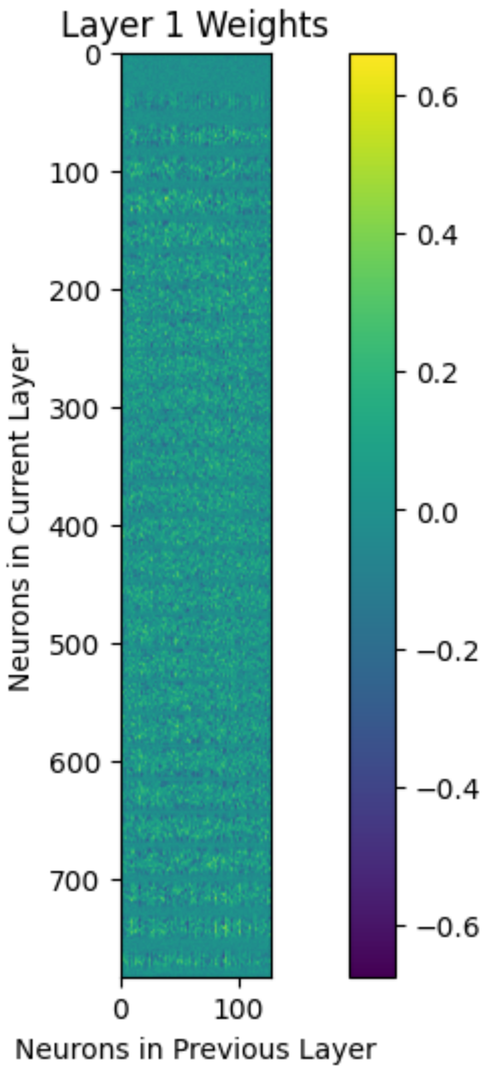
```
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model

# Load the MNIST dataset
(x_train, _), (x_test, _) = tf.keras.datasets.mnist.load_data()

# Normalize the data (scale it 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 = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

# Define the architecture of the autoencoder
input_layer = Input(shape=(784,))
encoded = Dense(128, activation='relu')(input_layer)
decoded = Dense(784, activation='sigmoid')(encoded)

autoencoder = Model(input_layer, decoded)

# Compile the model
```

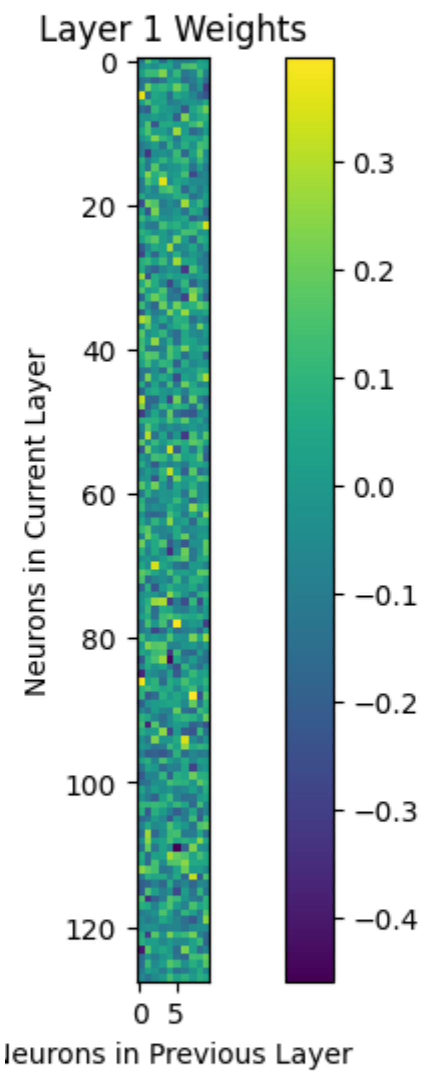
```
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

# Train the autoencoder
autoencoder.fit(x_train, x_train, epochs=10, batch_size=256, shuffle=True, validation_data=(x_test, x_test))

# Extract the encoder part of the model
encoder = Model(input_layer, encoded)

# Define a classifier on top of the encoder
classifier_input = Input(shape=(128,))
classifier_output = Dense(10, activation='softmax')(classifier_input)

classifier = Model(classifier_input, classifier_output)

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

# Train the classifier on the encoded features
encoded_train = encoder.predict(x_train)
encoded_test = encoder.predict(x_test)

classifier.fit(encoded_train, y_train, epochs=10, batch_size=256, shuffle=True, validation_data=(encoded_test, y_test))

```

```
Epoch 1/10
235/235 [==============================] - 3s 10ms/step - loss: 0.2129 - val_loss: 0.1350
Epoch 2/10
235/235 [==============================] - 2s 9ms/step - loss: 0.1178 - val_loss: 0.1028
Epoch 3/10
235/235 [==============================] - 2s 9ms/step - loss: 0.0964 - val_loss: 0.0888
Epoch 4/10
235/235 [==============================] - 2s 8ms/step - loss: 0.0857 - val_loss: 0.0812
Epoch 5/10
235/235 [==============================] - 2s 8ms/step - loss: 0.0796 - val_loss: 0.0768
Epoch 6/10
235/235 [==============================] - 2s 8ms/step - loss: 0.0760 - val_loss: 0.0741
Epoch 7/10
235/235 [==============================] - 2s 8ms/step - loss: 0.0736 - val_loss: 0.0722
Epoch 8/10
235/235 [==============================] - 2s 8ms/step - loss: 0.0721 - val_loss: 0.0709
Epoch 9/10
235/235 [==============================] - 2s 8ms/step - loss: 0.0709 - val_loss: 0.0699
Epoch 10/10
235/235 [==============================] - 2s 8ms/step - loss: 0.0700 - val_loss: 0.0692
1875/1875 [==============================] - 2s 1ms/step
313/313 [==============================] - 0s 1ms/step
Epoch 1/10
235/235 [==============================] - 1s 2ms/step - loss: 2.6237 - accuracy: 0.3915 - val_loss: 1.0302 - val_accuracy: 0.6720
Epoch 2/10
235/235 [==============================] - 0s 2ms/step - loss: 0.7804 - accuracy: 0.7543 - val_loss: 0.5655 - val_accuracy: 0.8279
Epoch 3/10
235/235 [==============================] - 0s 2ms/step - loss: 0.5325 - accuracy: 0.8369 - val_loss: 0.4402 - val_accuracy: 0.8681
Epoch 4/10
235/235 [==============================] - 0s 2ms/step - loss: 0.4449 - accuracy: 0.8670 - val_loss: 0.3899 - val_accuracy: 0.8842
Epoch 5/10
235/235 [==============================] - 0s 2ms/step - loss: 0.4007 - accuracy: 0.8823 - val_loss: 0.3549 - val_accuracy: 0.8978
Epoch 6/10
235/235 [==============================] - 0s 2ms/step - loss: 0.3748 - accuracy: 0.8901 - val_loss: 0.3391 - val_accuracy: 0.9015
Epoch 7/10
235/235 [==============================] - 0s 2ms/step - loss: 0.3561 - accuracy: 0.8968 - val_loss: 0.3251 - val_accuracy: 0.9066
Epoch 8/10
235/235 [==============================] - 0s 2ms/step - loss: 0.3427 - accuracy: 0.9004 - val_loss: 0.3159 - val_accuracy: 0.9097
Epoch 9/10
235/235 [==============================] - 0s 2ms/step - loss: 0.3335 - accuracy: 0.9034 - val_loss: 0.3168 - val_accuracy: 0.9070
Epoch 10/10
235/235 [==============================] - 0s 2ms/step - loss: 0.3274 - accuracy: 0.9062 - val_loss: 0.3069 - val_accuracy: 0.9111
<keras.callbacks.History at 0x1a72d1eca90>
```

```
import matplotlib.pyplot as plt

# Function to create and display a heat map
def display_layer_heatmap(model, layer_index):
    # Get the weights of the specified layer
    layer_weights = model.layers[layer_index].get_weights()[0]

    # Create a figure for the heat map
    plt.figure(figsize=(10, 6))

    # Plot the weights as a heat map
    plt.imshow(layer_weights, cmap='viridis')

    # Set labels and title
    plt.xlabel('Neurons in Previous Layer')
    plt.ylabel('Neurons in Current Layer')
    plt.title(f'Layer {layer_index} Weights')
    plt.colorbar()

```

```
    # Show the plot
    plt.show()

# Display the heat maps for the encoder layer and the classifier layer
display_layer_heatmap(autoencoder, 1)  # Display encoder layer (index 1)
display_layer_heatmap(classifier, 1)   # Display classifier layer (index 1)

```





3. Create a neural network architecture based auto encoder to classification of data. Typically, it consists of an encoder and a decoder. (it contain compilation, training, fitting, predicting) based on the above two points.

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

# Define the architecture of the autoencoder
input_layer = Input(shape=(784,))
encoded = Dense(128, activation='relu')(input_layer)
decoded = Dense(784, activation='sigmoid')(encoded)
```

```python
autoencoder = Model(input_layer, decoded)


autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(x_train, x_train, epochs=10, batch_size=256, shuffle=True, validation_data=(x_test, x_test))
encoder = Model(input_layer, encoded)

# Define a classifier on top of the encoder
classifier_input = Input(shape=(128,))
classifier_output = Dense(10, activation='softmax')(classifier_input)

classifier = Model(classifier_input, classifier_output)
classifier.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

encoded_train = encoder.predict(x_train)
encoded_test = encoder.predict(x_test)
classifier.fit(encoded_train, y_train, epochs=10, batch_size=256, shuffle=True, validation_data=(encoded_test, y_test))

test_loss, test_accuracy = classifier.evaluate(encoded_test, y_test)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")

predictions = classifier.predict(encoded_test)

```

```
Epoch 1/10
235/235 [==============================] - 2s 8ms/step - loss: 0.2118 - val_loss: 0.1351
Epoch 2/10
235/235 [==============================] - 2s 8ms/step - loss: 0.1185 - val_loss: 0.1035
Epoch 3/10
235/235 [==============================] - 2s 8ms/step - loss: 0.0970 - val_loss: 0.0897
Epoch 4/10
235/235 [==============================] - 2s 9ms/step - loss: 0.0863 - val_loss: 0.0818
Epoch 5/10
235/235 [==============================] - 2s 9ms/step - loss: 0.0801 - val_loss: 0.0772
Epoch 6/10
235/235 [==============================] - 2s 10ms/step - loss: 0.0764 - val_loss: 0.0744
Epoch 7/10
235/235 [==============================] - 2s 10ms/step - loss: 0.0740 - val_loss: 0.0724
Epoch 8/10
235/235 [==============================] - 3s 12ms/step - loss: 0.0723 - val_loss: 0.0710
Epoch 9/10
235/235 [==============================] - 3s 11ms/step - loss: 0.0710 - val_loss: 0.0700
Epoch 10/10
235/235 [==============================] - 3s 11ms/step - loss: 0.0701 - val_loss: 0.0693
1875/1875 [==============================] - 3s 1ms/step
313/313 [==============================] - 0s 1ms/step
Epoch 1/10
235/235 [==============================] - 1s 2ms/step - loss: 2.4455 - accuracy: 0.4030 - val_loss: 0.9809 - val_accuracy: 0.6835
Epoch 2/10
235/235 [==============================] - 0s 2ms/step - loss: 0.7511 - accuracy: 0.7656 - val_loss: 0.5589 - val_accuracy: 0.8329
Epoch 3/10
235/235 [==============================] - 0s 2ms/step - loss: 0.5218 - accuracy: 0.8456 - val_loss: 0.4469 - val_accuracy: 0.8702
Epoch 4/10
235/235 [==============================] - 0s 2ms/step - loss: 0.4382 - accuracy: 0.8729 - val_loss: 0.3879 - val_accuracy: 0.8877
Epoch 5/10
235/235 [==============================] - 0s 2ms/step - loss: 0.3939 - accuracy: 0.8877 - val_loss: 0.3640 - val_accuracy: 0.8949
Epoch 6/10
235/235 [==============================] - 0s 2ms/step - loss: 0.3693 - accuracy: 0.8937 - val_loss: 0.3393 - val_accuracy: 0.9039
Epoch 7/10
235/235 [==============================] - 0s 2ms/step - loss: 0.3509 - accuracy: 0.8998 - val_loss: 0.3259 - val_accuracy: 0.9086
Epoch 8/10
235/235 [==============================] - 0s 2ms/step - loss: 0.3381 - accuracy: 0.9028 - val_loss: 0.3199 - val_accuracy: 0.9106
Epoch 9/10
235/235 [==============================] - 0s 2ms/step - loss: 0.3301 - accuracy: 0.9056 - val_loss: 0.3128 - val_accuracy: 0.9091
Epoch 10/10
235/235 [==============================] - 0s 2ms/step - loss: 0.3244 - accuracy: 0.9071 - val_loss: 0.3042 - val_accuracy: 0.9148
313/313 [==============================] - 0s 1ms/step - loss: 0.3042 - accuracy: 0.9148
Test Accuracy: 91.48%
313/313 [==============================] - 0s 997us/step
```

```python
def display_predictions(images, true_labels, predicted_labels):
    plt.figure(figsize=(10, 6))
    for i in range(len(images)):
        plt.subplot(2, len(images)//2, i+1)
        plt.imshow(images[i].reshape(28, 28), cmap='gray')
        plt.title(f'True: {true_labels[i]}\nPredicted: {predicted_labels[i]}')
        plt.axis('off')
    plt.tight_layout()
    plt.show()

random_indices = np.random.randint(0, len(x_test), 6)
sample_images = x_test[random_indices]
true_labels = y_test[random_indices]

# Predict labels using the classifier
predicted_labels = np.argmax(classifier.predict(encoder.predict(sample_images)), axis=1)

```

```
display_predictions(sample_images, true_labels, predicted_labels)

```

```
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
```