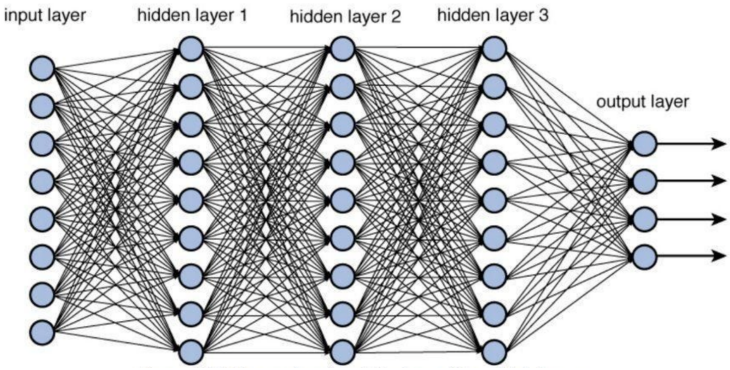
DEEP LEARNING : MULTI LAYER NEURAL NETWORK



HEMANT THAPA

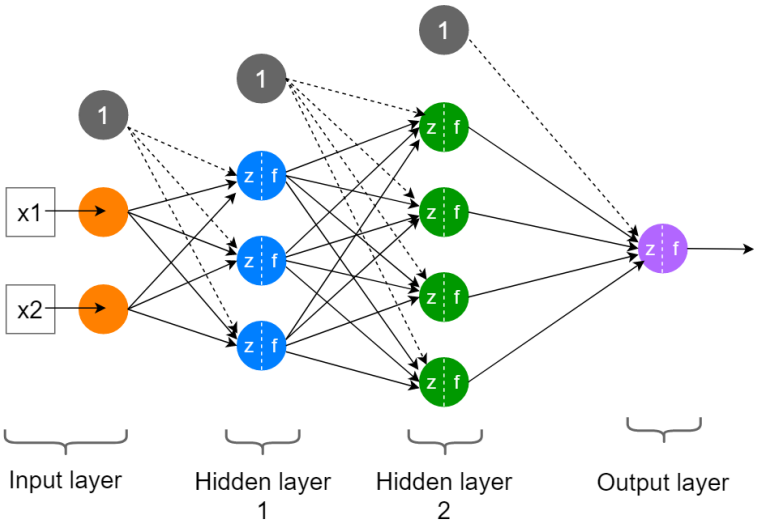
In [67]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import seaborn as sns
import requests
import tensorflow as tf
import yfinance as yf
```

In [2]:
```
from PIL import Image
from io import BytesIO
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from keras.layers import Activation
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

1. INPUT LAYER

The input layer is the initial layer of a neural network that takes in the raw input data or features of a problem. It serves as the entry point for the data into the neural network architecture. The main purpose of the input layer is to receive the input data and pass it on to the subsequent layers for processing.

The structure of the input layer depends on the nature of the data being processed. For example, if you're dealing with images, each neuron in the input layer might correspond to a pixel's intensity value. In natural language processing tasks, each neuron could represent a word or a token in a sentence. If the input data is numerical, there's no need for complex representations like pixels or tokens. Each neuron in the input layer directly represents a numerical value from your dataset.



NUMBER OF NEURONS

The number of neurons in the input layer is equal to the number of features or variables in your dataset. For example, if you're using historical stock data with attributes like opening price, closing price, volume, etc., you would have one neuron for each of these attributes in the input layer. If you're working with a 28x28 image, you would typically have 784 neurons in the input layer (assuming no flattening or resizing).

ACTIVATION FUNCTION

In most cases, the input layer doesn't use an activation function. The purpose of the input layer is to directly pass the input values to the next layer without altering them. The input values are usually normalized or standardized before being fed into the network.

Generally, for numerical data, the input layer doesn't have an activation function. The purpose of the input layer is to pass the numerical values to the subsequent layers without altering them.

CONNECTION TO HIDDEN LAYERS

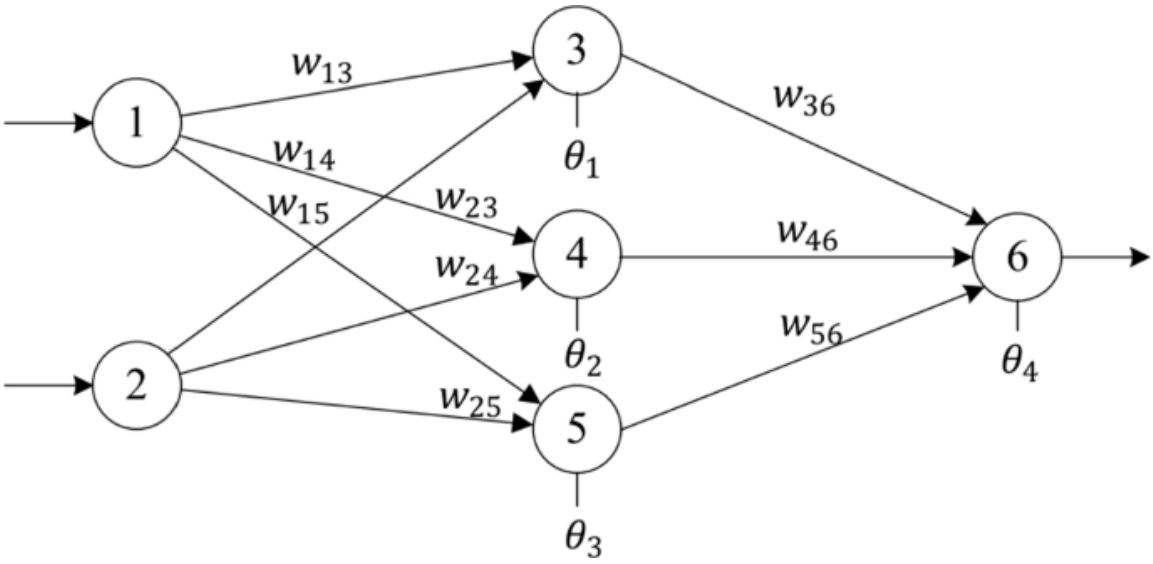
The neurons in the input layer are fully connected to the neurons in the subsequent hidden layers. Each connection between a neuron in the input layer and a neuron in the hidden layer has an associated weight that will be learned during the training process.

PREPROCESSING

It's a good practice to normalize or standardize your numerical data before feeding it into the neural network. This ensures that the data is on a similar scale, which can help with training stability and convergence.

Often, the input data is preprocessed before being fed into the neural network. This might involve tasks such as scaling the data to a specific range, normalizing the data, or encoding categorical variables into numerical representations.

2. HIDDEN LAYERS



A hidden layer in a neural network is a layer of neurons that sits between the input layer and the output layer. It's called "hidden" because its neurons are not directly connected to the input or output of the neural network. Hidden layers are where most of the computation and feature extraction occur in neural networks, allowing the network to learn complex patterns and relationships within the data.

FEATURE TRANSFORMATION

The neurons in hidden layers perform computations on the input data, transforming it into a more abstract and informative representation. Each neuron in a hidden layer receives inputs from all the neurons in the previous layer (whether that's the input layer or a previous hidden layer).

ACTIVATION FUNCTION

Each neuron in a hidden layer applies an activation function to the weighted sum of its inputs. This introduces non-linearity into the model, enabling the network to learn and represent complex relationships in the data.

DEPTH AND WIDTH

Neural networks can have multiple hidden layers, creating what's known as a "deep" neural network. The depth refers to the number of hidden layers, while the width refers to the number of neurons in each hidden layer. Deeper networks can capture more intricate features, but they also require more data and computational resources.

LEARNING REPRESENTATIONS

The purpose of hidden layers is to learn useful representations of the data that help in solving the specific task the network is designed for. In image recognition, for example, hidden layers might learn to recognize edges, shapes, and higher-level features.

WEIGHTS AND BIASES

The connections between neurons in different layers have associated weights and biases that are learned during the training process. These weights determine the influence of each input on the neuron's output.

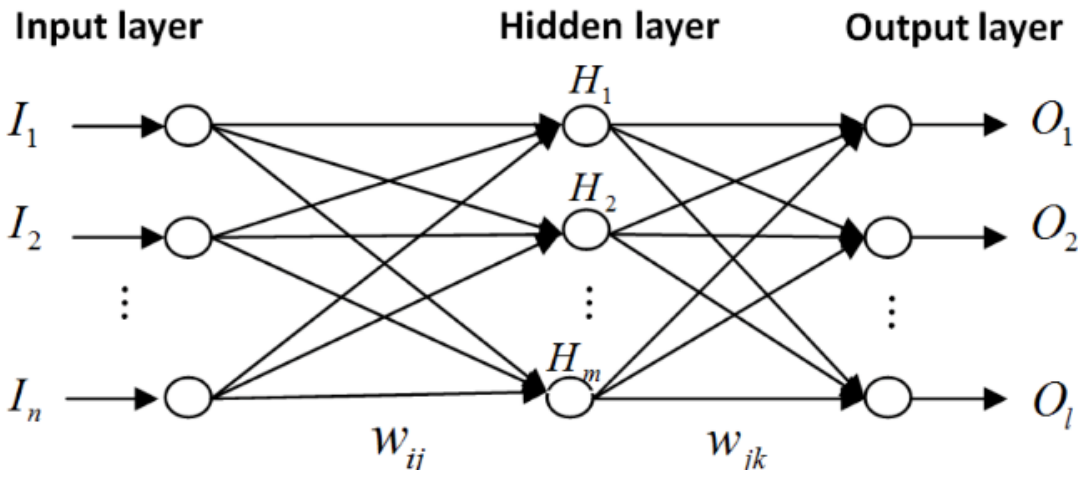
BACKPROPAGATION

The learning process involves both forward and backward passes through the network. During forward propagation, data is passed through the layers to make predictions. During backpropagation, the error between the predicted output and the actual target is used to adjust the weights and biases in order to improve the model's accuracy.

ACTIVATION FUNCTIONS

Activation functions like ReLU (Rectified Linear Unit), sigmoid, and tanh are commonly used in hidden layers to introduce non-linearities. These functions help the network approximate complex functions and relationships.

3. OUTPUT LAYER



The output layer in a neural network is the final layer that produces the network's predictions or outputs based on the computations and transformations performed in the preceding layers, including the hidden layers. The structure and configuration of the output layer depend on the specific task the neural network is designed to solve.

TASK-SPECIFIC STRUCTURE

The design of the output layer is tailored to the nature of the problem being solved. For example, in binary classification tasks, the output layer might consist of a single neuron with an activation function like sigmoid to produce a probability value. In multi-class classification, there might be multiple neurons corresponding to each class, often using an activation like softmax to produce class probabilities that sum to 1.

NUMBER OF NEURONS

The number of neurons in the output layer depends on the number of classes or dimensions of the output. For instance, if you're classifying images into 10 different categories, the output layer would typically have 10 neurons.

ACTIVATION FUNCTION

The activation function in the output layer is selected based on the problem. For regression tasks, the output neurons might use linear activation or a modified activation that maps to the desired output range. For classification tasks, softmax is commonly used for multi-class classification, while sigmoid is used for binary classification.

INTERPRETING OUTPUTS

The outputs of the neurons in the output layer can often be directly interpreted as probabilities, class predictions, or values relevant to the problem. For example, in a neural network for image classification, the neuron with the highest activation in the output layer corresponds to the predicted class.

LOSS FUNCTION

The choice of loss function depends on the task. For classification, cross-entropy loss is commonly used, while for regression, mean squared error or other appropriate loss functions might be used.

TRAINING AND BACKPROPAGATION

During training, the error between the predicted outputs and the actual target values is computed using the chosen loss function. This error is then used to adjust the weights and biases of the entire network through backpropagation, aiming to minimize the error over the training data.

FINAL DECISION OR PREDICTION

The outputs from the output layer represent the final decision or prediction made by the neural network. For example, in image recognition, the class label with the highest probability is often taken as the predicted class.

3. MNIST DATASET

In [3]:
```
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

In [4]:
```
print(x_train.shape)
print(y_train.shape)
```

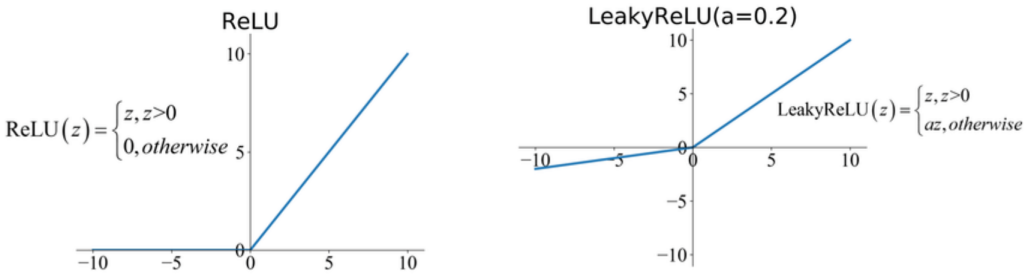
```
(60000, 28, 28)
(60000,)
```

In [5]:
```
print(x_test.shape)
print(y_test.shape)
```

```
(10000, 28, 28)
(10000,)
```

The input layer has 28x28 = 784 input units/neurons. This layer flattens the 2D image representation into a 1D array of length 784, which serves as the input for the subsequent layers.

This layer is the first hidden layer in the network, and it contains 128 neurons. The activation function used for this layer is the Rectified Linear Unit (ReLU), which is specified as 'relu'.



In [6]:
```
model = Sequential([
    Flatten(input_shape=(28, 28)),      # Flatten the 28x28 pixel images int
    Dense(128, activation='relu'),       # First hidden layer with 128 neuro
    Dense(64, activation='relu'),        # Second hidden layer with 64 neuro
    Dense(32, activation='relu'),        # Third hidden layer with 32 neuron
    Dense(10, activation='softmax')      # Output layer with 10 neurons (one
])
```

$$\mathrm{softmax}(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{ for j} = 1,\ldots,\text{K}$$

In [7]:
```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

In [8]:
```
model.fit(x_train, y_train, epochs=10)
```

```
Epoch 1/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.2536 - ac
curacy: 0.9241
Epoch 2/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.1067 - ac
curacy: 0.9678
Epoch 3/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0769 - ac
curacy: 0.9763
Epoch 4/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0597 - ac
curacy: 0.9813
Epoch 5/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0499 - ac
curacy: 0.9840
Epoch 6/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0391 - ac
curacy: 0.9878
Epoch 7/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0343 - ac
curacy: 0.9891
Epoch 8/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0282 - a
ccuracy: 0.9910
Epoch 9/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0272 - ac
curacy: 0.9912
Epoch 10/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0219 - ac
curacy: 0.9930
```

Out[8]:
```
<keras.callbacks.History at 0x7f9349263cd0>
```

In [9]:
```
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print("Test accuracy:", test_accuracy)
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.1005 - accu
racy: 0.9753
Test accuracy: 0.9753000140190125
```

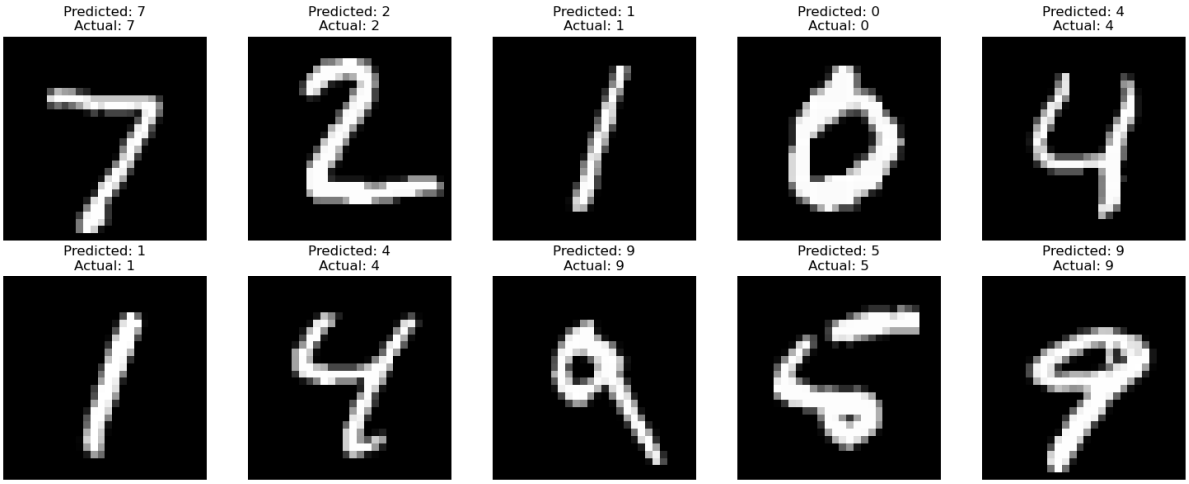
In [10]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 128)               100480

 dense_1 (Dense)             (None, 64)                8256

 dense_2 (Dense)             (None, 32)                2080

 dense_3 (Dense)             (None, 10)                330

=================================================================
Total params: 111,146
Trainable params: 111,146
Non-trainable params: 0
_________________________________________________________________
```

In [11]:
```
sample_images = x_test[:10]
predictions = model.predict(sample_images)
```

```
1/1 [==============================] - 0s 163ms/step
```

In [12]:
```
plt.figure(figsize=(15, 6))
for i in range(10):
    plt.subplot(2, 5, i+1)
    plt.imshow(sample_images[i], cmap='gray')
    plt.title(f"Predicted: {predictions[i].argmax()}\nActual: {y_test[i]}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```



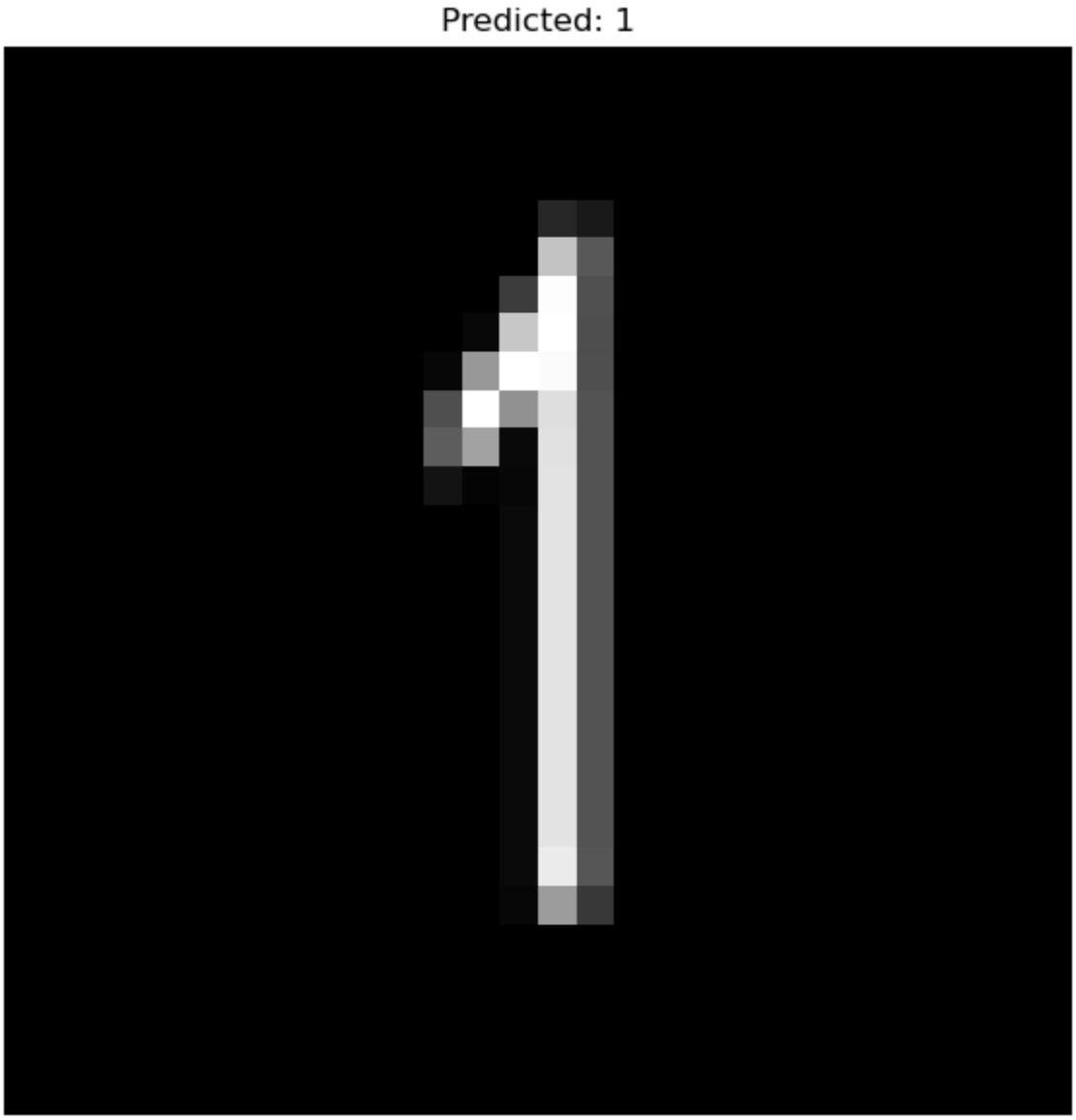
TEST 1

In [13]:
```
additional_image_urls = ['https://wallpaperset.com/w/full/c/7/f/402336.jpg']
additional_images = []

for url in additional_image_urls:
    response = requests.get(url)
    image = Image.open(BytesIO(response.content)).convert('L')  # Convert to
    image_resized = image.resize((28, 28))
    image_processed = np.array(image_resized) / 255.0
    additional_images.append(image_processed)
```

In [14]:
```
additional_images = np.array(additional_images)
additional_predictions = model.predict(additional_images)

# Plot the additional images along with their predicted labels
plt.figure(figsize=(6, 6))
```

```
for i in range(len(additional_images)):
    plt.subplot(1, len(additional_images), i + 1)
    plt.imshow(additional_images[i], cmap='gray')
    plt.title(f"Predicted: {additional_predictions[i].argmax()}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

```
1/1 [==============================] - 0s 20ms/step
```

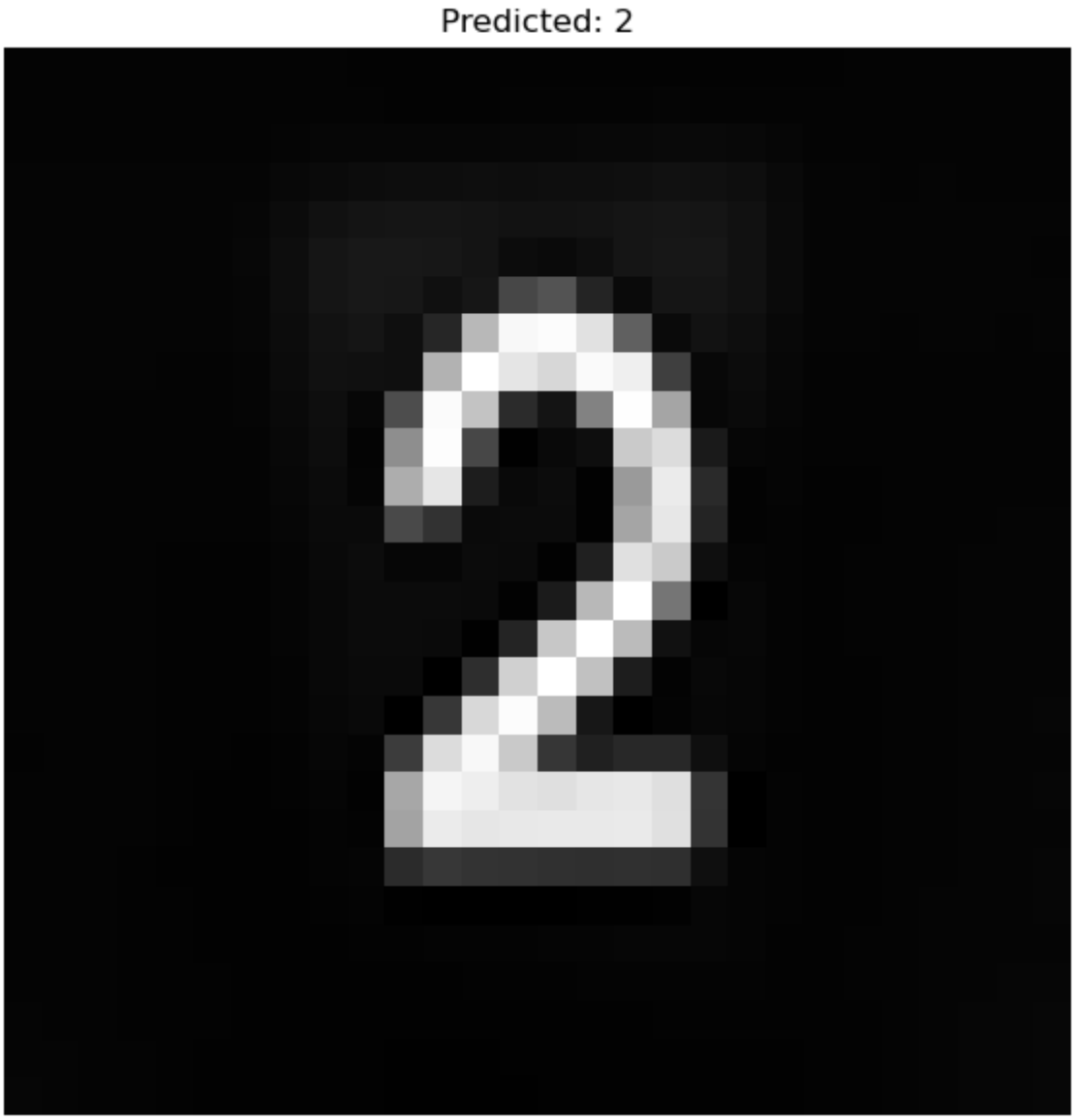


TEST 2

In [15]:
```
additional_image_urls = ['https://www.bighealey.co.uk/image/cache/data/C_Par
additional_images = []

for url in additional_image_urls:
    response = requests.get(url)
    image = Image.open(BytesIO(response.content)).convert('L')  # Convert to
    image_resized = image.resize((28, 28))
    image_processed = np.array(image_resized) / 255.0
    additional_images.append(image_processed)
```

In [16]:
```
additional_images = np.array(additional_images)
additional_predictions = model.predict(additional_images)

# Plot the additional images along with their predicted labels
plt.figure(figsize=(6, 6))
for i in range(len(additional_images)):
    plt.subplot(1, len(additional_images), i + 1)
    plt.imshow(additional_images[i], cmap='gray')
    plt.title(f"Predicted: {additional_predictions[i].argmax()}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

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



4. STOCK PREDICTION USING MULTI LAYER PART 1

Model Architecture:

Input Layer: A dense layer with ReLU activation, expecting input shape (num_samples, num_features) where num_features is determined by the number of polynomial features.

Hidden Layer 1: A dense layer with 64 units and ReLU activation.

Hidden Layer 2: A dense layer with 32 units and ReLU activation.

Output Layer: A dense layer with 1 unit and linear activation, predicting MSFT close prices.

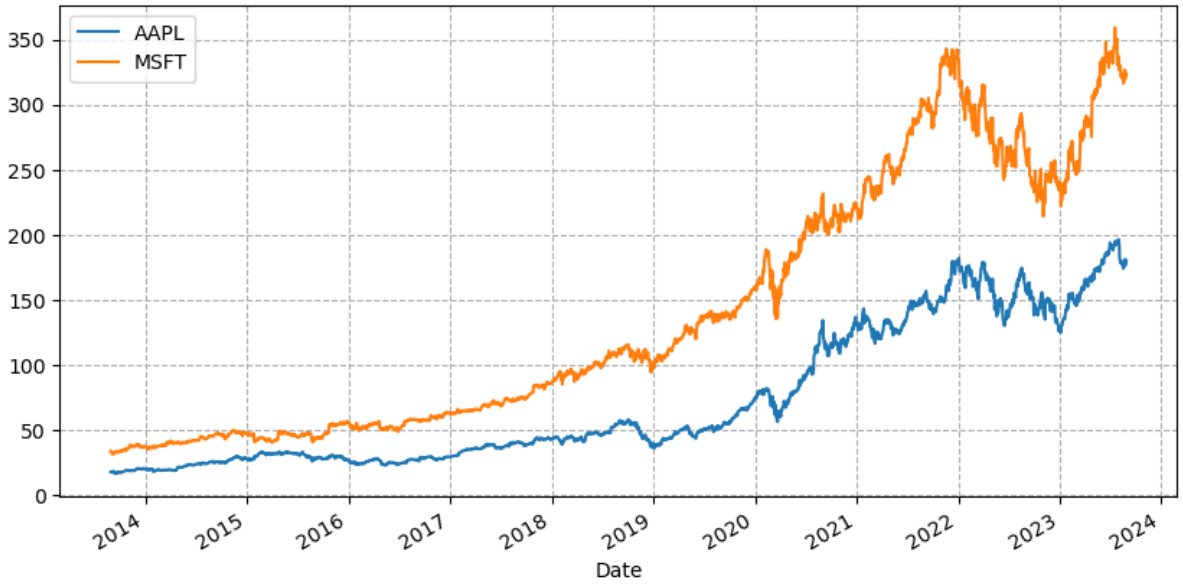
In [17]:
```
stock_data = yf.download(['AAPL', 'MSFT'], period="10Y")['Close']
```

```
[**********************100%**********************]  2 of 2 completed
```

In [18]:
```
aapl_close = stock_data['AAPL']
msft_close = stock_data['MSFT']
```

In [19]:
```
plt.figure(figsize=(10,5))
aapl_close.plot()
```

```
msft_close.plot()
plt.legend()
plt.grid(True, linestyle="--")
plt.show()
```



In [20]:
```
aapl_close = np.array(aapl_close)
msft_close = np.array(msft_close)
```

In [21]:
```
aapl_close = (aapl_close - aapl_close.mean()) / aapl_close.std()
msft_close = (msft_close - msft_close.mean()) / msft_close.std()
aapl_close = np.array(aapl_close).reshape(-1, 1)
msft_close = np.array(msft_close).reshape(-1, 1)
```

In [22]:
```
aapl_close[:5]
```

Out[22]:
```
array([[-1.02176061],
       [-1.02474723],
       [-1.0238406 ],
       [-1.01710071],
       [-1.01938067]])
```

In [23]:
```
msft_close[:5]
```

Out[23]:
```
array([[-1.09518049],
       [-1.09673583],
       [-1.1124969 ],
       [-1.11954787],
       [-1.11923681]])
```

In [24]:
```
aapl_close = aapl_close.reshape(-1, 1)
msft_close = msft_close.reshape(-1, 1)
```

In [25]:
```
from sklearn.preprocessing import PolynomialFeatures
```

In [26]:
```
poly = PolynomialFeatures(degree=1)
aapl_close_poly = poly.fit_transform(aapl_close)
```

In [27]:
```
aapl_train, aapl_test, msft_train, msft_test = train_test_split(aapl_close_p
                                                                msft_close,
```

RELU & LINEAR ACTIVATION FUNCTION

In [28]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_shape=(aapl_close_pol
    tf.keras.layers.Dense(32, activation='linear'),
    tf.keras.layers.Dense(1, activation='linear')
])
```

In [29]:
```
model.compile(optimizer='adam', loss='mean_squared_error')
history = model.fit(aapl_train, msft_train, epochs=50, batch_size=32, valida
```

1. Small Batch Size (e.g., 8, 16):

Faster convergence in terms of updates per epoch.

More noisy updates due to the smaller sample size.

More frequent weight updates, which can be both good and bad. It might help the model escape local minima, but it can also make the optimization process less stable.

2. Moderate Batch Size (e.g., 32, 64):

A balance between computation efficiency and stability.

Often a good choice for most applications and datasets.

Popular choice for a variety of neural network architectures.

3. Large Batch Size (e.g., 128, 256 or more):

Faster computation due to vectorized operations.

Can lead to smoother updates and more accurate gradient estimates.

Might require more memory and computational resources.

In [30]:
```
plt.figure(figsize=(10,5))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.grid(True, linestyle="--")
plt.show()
```

Over fitting model according to training & validation Loss

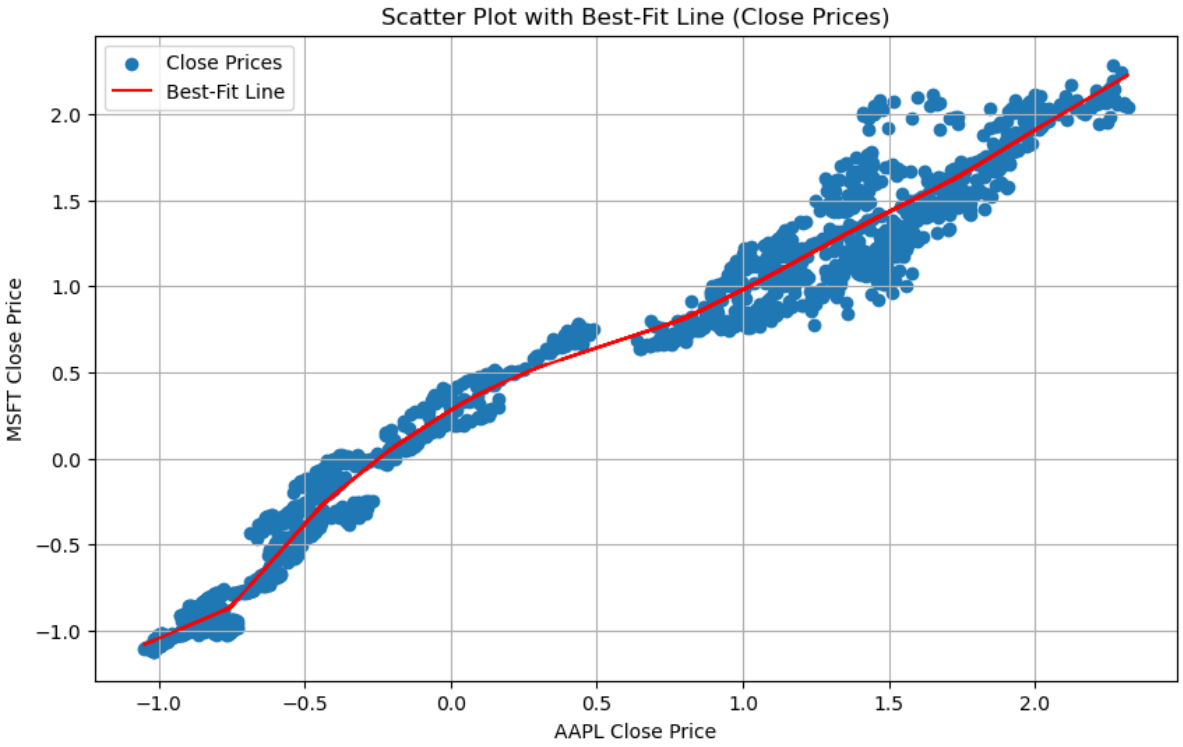
In [31]:
```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_4 (Dense)             (None, 64)                192

 dense_5 (Dense)             (None, 32)                2080

 dense_6 (Dense)             (None, 1)                 33

=================================================================
Total params: 2,305
Trainable params: 2,305
Non-trainable params: 0
_________________________________________________________________
```

In [32]:
```
layer_index = 0
layer = model.layers[layer_index]
weights, biases = layer.get_weights()
```

In [33]:
```
plt.figure(figsize=(10, 6))
plt.scatter(aapl_close, msft_close, label='Close Prices')
plt.plot(aapl_close, model.predict(aapl_close_poly), color='red', label='Bes
plt.xlabel('AAPL Close Price')
plt.ylabel('MSFT Close Price')
plt.title('Scatter Plot with Best-Fit Line (Close Prices)')
plt.legend()
plt.grid()
plt.show()
```

```
79/79 [==============================] - 0s 1ms/step
```



In [34]:
```
from sklearn.metrics import r2_score
```

In [35]:
```
predictions = model.predict(aapl_test)
r2 = r2_score(msft_test, predictions)
print(f"R-squared Score on Test Set: {r2:.4f}")
```

```
16/16 [==============================] - 0s 2ms/step
R-squared Score on Test Set: 0.9826
```

In [36]:
```
predict_point = 200
predict_point = np.array([predict_point])
predict_point = predict_point.reshape(-1, 1)
predict_point_poly = poly.transform(predict_point)

# Make predictions
predictionOne = model.predict(predict_point_poly)

print(f"Predicted MSFT Close Price at AAPL Close Price {predict_point[0][0]}
```

```
1/1 [==============================] - 0s 23ms/step
Predicted MSFT Close Price at AAPL Close Price 200: 200.91
```

5. STOCK PREDICTION USING MULTI LAYER PART 2

Model Architecture:

Input Layer: A dense layer with linear activation, expecting input shape (num_samples, 1) where each sample is a single AAPL close price.

Dropout Layer 1: A dropout layer with a dropout rate of 0.2. Dropout helps prevent overfitting by randomly setting a fraction of input units to 0 during training.

Hidden Layer 1: A dense layer with linear activation and 64 units.

Dropout Layer 2: A dropout layer with a dropout rate of 0.2.

Hidden Layer 2: A dense layer with linear activation and 32 units.

Dropout Layer 3: A dropout layer with a dropout rate of 0.2.

Output Layer: A dense layer with a single unit for predicting MSFT close prices.

In [37]:
```
stock_data = yf.download(['AAPL', 'MSFT'], period="10Y")['Close']
aapl_close = stock_data['AAPL']
msft_close = stock_data['MSFT']
```

```
[**********100%**********]  2 of 2 completed
```

In [38]:
```
aapl_close = (aapl_close - aapl_close.mean()) / aapl_close.std()
msft_close = (msft_close - msft_close.mean()) / msft_close.std()
```

In [39]:
```
aapl_close[:5]
```

Out[39]:
```
Date
2013-08-29   -1.021558
2013-08-30   -1.024544
2013-09-03   -1.023637
2013-09-04   -1.016899
2013-09-05   -1.019178
Name: AAPL, dtype: float64
```

In [40]:
```
msft_close[:5]
```

Out[40]:
```
Date
2013-08-29   -1.094963
2013-08-30   -1.096518
2013-09-03   -1.112276
2013-09-04   -1.119325
2013-09-05   -1.119014
Name: MSFT, dtype: float64
```

In [41]:
```
aapl_close = np.array(aapl_close).reshape(-1, 1)
msft_close = np.array(msft_close).reshape(-1, 1)
```

In [42]:
```
aapl_train, aapl_test, msft_train, msft_test = train_test_split(aapl_close,
                                                                msft_close,
                                                                test_size=0.
```

In [43]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='linear', input_shape=(1,)),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(32, activation='linear'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(1)
])
```

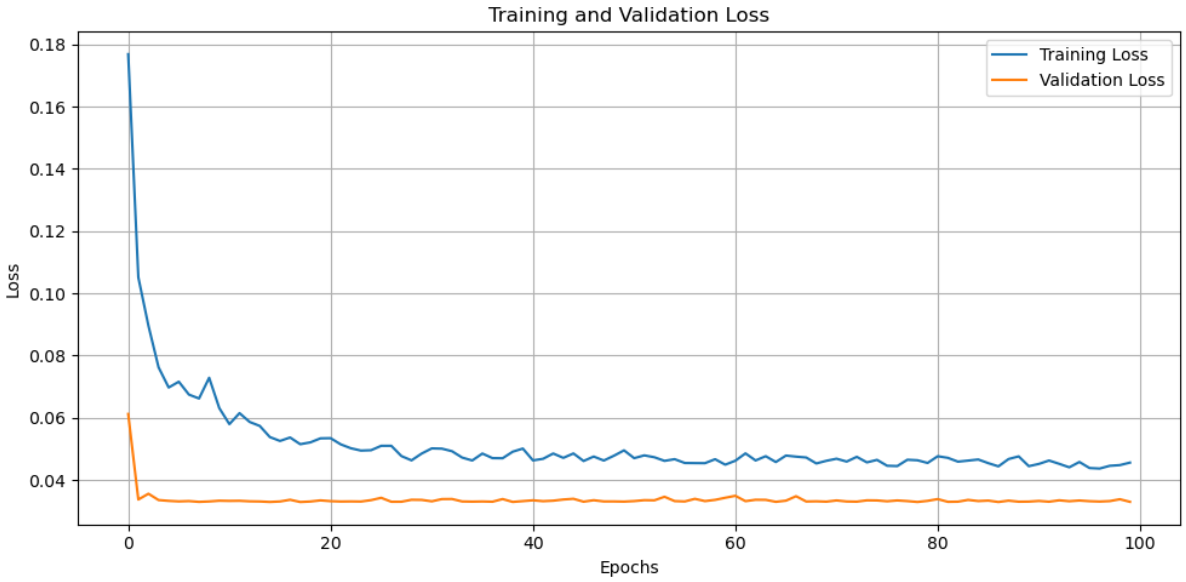
In [44]:
```
model.compile(optimizer='adam', loss='mean_squared_error')
```

In [45]:
```
history = model.fit(aapl_train, msft_train, epochs=100, batch_size=128, vali
```

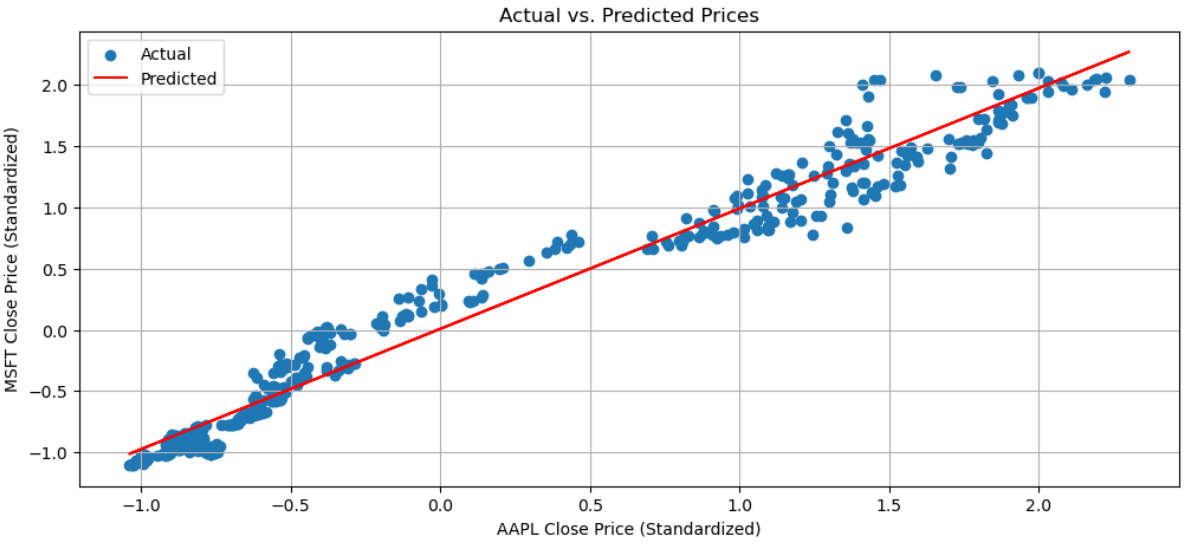
In [46]:
```
predictions = model.predict(aapl_test)
r2 = r2_score(msft_test, predictions)
print(f"R-squared Score on Test Set: {r2:.4f}")
```

```
16/16 [==============================] - 0s 1ms/step
R-squared Score on Test Set: 0.9676
```

In [47]:
```
plt.figure(figsize=(10, 5))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.grid()
plt.tight_layout()
plt.show()
```



In [48]:
```
plt.figure(figsize=(12, 5))
plt.scatter(aapl_test, msft_test, label='Actual')
plt.plot(aapl_test, predictions, color='red', label='Predicted')
plt.xlabel('AAPL Close Price (Standardized)')
plt.ylabel('MSFT Close Price (Standardized)')
plt.title('Actual vs. Predicted Prices')
plt.legend()
plt.grid()
```



In [49]:
```
predict_point = 200
predict_point = np.array([predict_point])
predictionTwo = model.predict(predict_point)
```

```
1/1 [==============================] - 0s 58ms/step
```

In [50]:
```
predictionTwo
```

Out[50]:
```
array([[196.58853]], dtype=float32)
```

6. STOCK PREDICTION USING MULTI LAYER PART 2

Model Architecture:

Input Layer: A dense layer with linear activation, expecting input shape (num_samples, 1) where each sample is a single AAPL close price.

Dropout Layer 1: A dropout layer with a dropout rate of 0.2. Dropout helps prevent overfitting by randomly setting a fraction of input units to 0 during training.

Hidden Layer 1: A dense layer with linear activation and 64 units.

Dropout Layer 2: A dropout layer with a dropout rate of 0.2.

Hidden Layer 2: A dense layer with linear activation and 32 units.

Dropout Layer 3: A dropout layer with a dropout rate of 0.2.

Output Layer: A dense layer with a single unit for predicting MSFT close prices.

In [51]:
```
stock_data = yf.download(['AAPL', 'MSFT'], period="10Y")['Close']
aapl_close = stock_data['AAPL']
msft_close = stock_data['MSFT']
```

```
[**********************100%**********************]  2 of 2 completed
```

In [52]:
```
aapl_close = (aapl_close - aapl_close.mean()) / aapl_close.std()
msft_close = (msft_close - msft_close.mean()) / msft_close.std()
aapl_close = np.array(aapl_close).reshape(-1, 1)
msft_close = np.array(msft_close).reshape(-1, 1)
```

In [53]:
```
aapl_train, aapl_test, msft_train, msft_test = train_test_split(aapl_close,
                                                                test_size=0.
```

In [54]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='linear', input_shape=(1,)),
    tf.keras.layers.Dense(32, activation='linear'),
    tf.keras.layers.Dense(1, activation='linear')
])
```

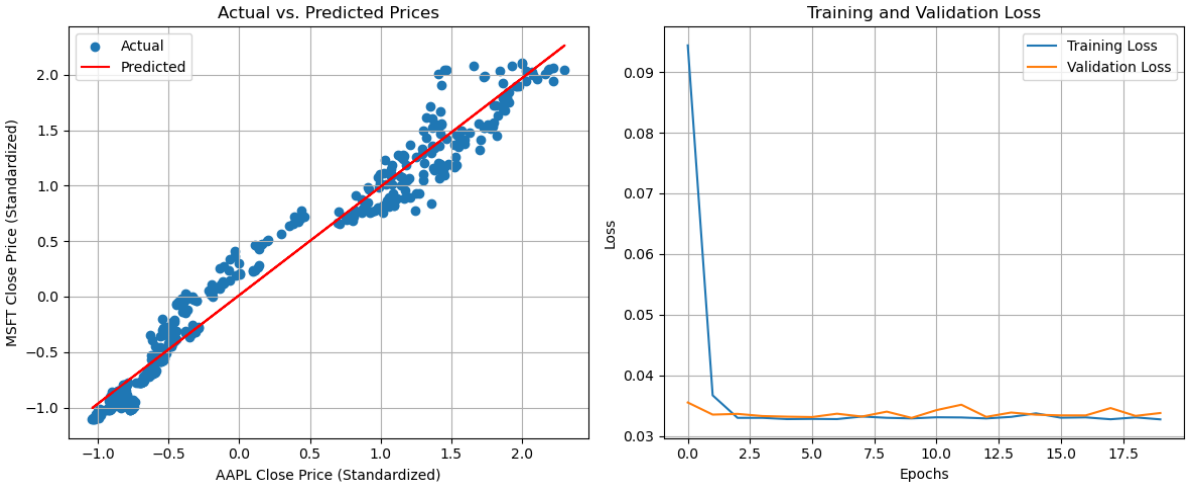
In [55]:
```
model.compile(optimizer='adam', loss='mean_squared_error')
early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patien
```

In [56]:
```
history = model.fit(aapl_train, msft_train, epochs=100, batch_size=64,
                    validation_split=0.2, callbacks=[early_stopping], verbos
```

In [57]:
```
predictions = model.predict(aapl_test)
r2 = r2_score(msft_test, predictions)
print(f"R-squared Score on Test Set: {r2:.4f}")
```

```
16/16 [==============================] - 0s 2ms/step
R-squared Score on Test Set: 0.9675
```

In [58]:
```
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.scatter(aapl_test, msft_test, label='Actual')
plt.plot(aapl_test, predictions, color='red', label='Predicted')
plt.xlabel('AAPL Close Price (Standardized)')
plt.ylabel('MSFT Close Price (Standardized)')
plt.title('Actual vs. Predicted Prices')
plt.legend()
plt.grid()

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.title('Training and Validation Loss')
plt.legend()
plt.grid()
plt.tight_layout()
plt.show()
```



GOOD FIT LEARNING CURVES

A good fit is the goal of the learning algorithm and exists between an overfit and underfit model.

A good fit is identified by a training and validation loss that decreases to a point of stability with a minimal gap between the two final loss values.

The loss of the model will almost always be lower on the training dataset than the validation dataset. This means that we should expect some gap between the train and validation loss learning curves. This gap is referred to as the “generalization gap.”

A PLOT OF LEARNING CURVES SHOWS A GOOD FIT IF

The plot of training loss decreases to a point of stability.

The plot of validation loss decreases to a point of stability and has a small gap with the training loss.

Continued training of a good fit will likely lead to an overfit.

In [59]:
```
predict_point = 200
predict_point = np.array([predict_point])
predictionThree = model.predict(predict_point)
```

```
1/1 [==============================] - 0s 91ms/step
```

In [60]:
```
predictionThree
```

Out[60]:
```
array([[195.6323]], dtype=float32)
```

In [61]:
```
print(predictionOne)
print(predictionTwo)
print(predictionThree)
```

```
[[200.9107]]
[[196.58853]]
[[195.6323]]
```

REFERENCES:

Machine Learning Mastery. (n.d.). Learning Curves for Diagnosing Machine Learning Model Performance. Retrieved from https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/

GeeksforGeeks. (n.d.). Artificial Neural Networks and Its Applications. Retrieved from https://www.geeksforgeeks.org/artificial-neural-networks-and-its-applications/

University of New South Wales. (n.d.). BackPropagation. Retrieved from https://www.cse.unsw.edu.au/~cs9417ml/MLP2/BackPropagation.html

Towards Data Science. (n.d.). Training Deep Neural Networks. Retrieved from https://towardsdatascience.com/training-deep-neural-networks-9fdb1964b964