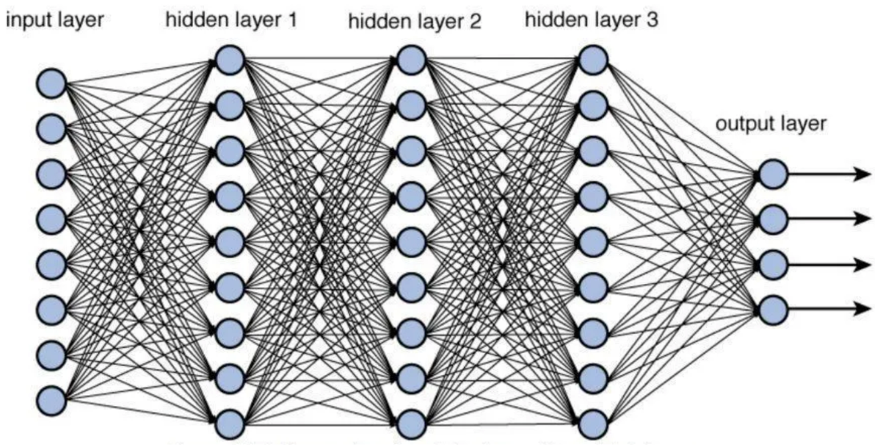
CHAPTER 1: BUILDING NEURAL NETWORK FOR LINEAR REGRESSION



HEMANT THAPA

In [1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import math
import statistics as st
import random
import yfinance as yf
import tensorflow as tf
import warnings
warnings.filterwarnings("ignore")
```

In [2]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

Linear regression is a fundamental and widely used statistical technique in machine learning and deep learning. While it may not be considered a deep learning model itself, it serves as a foundational concept that underlies more complex neural network architectures.

At its core, linear regression is a supervised learning algorithm used for predicting a continuous output variable (also known as the dependent or target variable) based on one or more input features (independent variables). It assumes a linear relationship between the input features and the output variable.

Mathematical Representation:

In simple linear regression, you have a single input feature (X) and a single output variable (Y). The relationship between them is represented as Y = b0 + b1 * X

Y represents the output variable.

X represents the input feature.

b0 is the y-intercept (bias).

b1 is the slope (weight) of the linear relationship.

In [3]:
```
class stock:
    def __init__(self, ticker, period):
        self.ticker = ticker
        self.period = period

    def chart(self):
        # Download historical stock data using yfinance and return it
        return yf.download(self.ticker, period=self.period)

#class for plotting linear regression predictions
class Plot:
    def __init__(self, X ,y, y_pred):
        self.X = X          # Independent variables
        self.y = y          # Actual dependent variables
        self.y_pred = y_pred  # Predicted dependent variables

    def linear_predict(self):
        # Create a scatter plot of actual data points
        plt.figure(figsize=(8,6))
        plt.scatter(self.X, self.y, alpha=0.5)

        # Plot the linear regression predictions in red
        plt.plot(self.X, self.y_pred, color="red")
        plt.grid(True, linestyle="--", color="grey", alpha=0.5)
        plt.xlabel("Dependent Variables")
        plt.ylabel("Independent Variables")
        plt.show()
```

In [4]:
```
# Define the time period for downloading financial data (5 years)
years = "5Y"

# Download historical data for the S&P 500 index (^GSPC) for the specified peric
X = yf.download("^GSPC", period=years)

# Download historical data for Microsoft (MSFT) stock prices for the same perioa
y = yf.download("MSFT", period=years)
```

```
[*********************100%%**********************]  1 of 1 completed
[*********************100%%**********************]  1 of 1 completed
```

In [5]:
```
#SNP500
X[:5]
```

Out[5]:

Date	Open	High	Low	Close	Adj Close	Volume
2018-10-01	2926.290039	2937.060059	2917.909912	2924.590088	2924.590088	3375890000
2018-10-02	2923.800049	2931.419922	2919.370117	2923.429932	2923.429932	3432620000
2018-10-03	2931.689941	2939.860107	2921.360107	2925.510010	2925.510010	3625510000
2018-10-04	2919.350098	2919.780029	2883.919922	2901.610107	2901.610107	3510370000
2018-10-05	2902.540039	2909.639893	2869.290039	2885.570068	2885.570068	3340820000

In [6]:

```
X_mean_value = np.mean(X.Close)
X_mode_value = X.Close.mode().values[0]
X_median_value = np.median(X.Close)
```

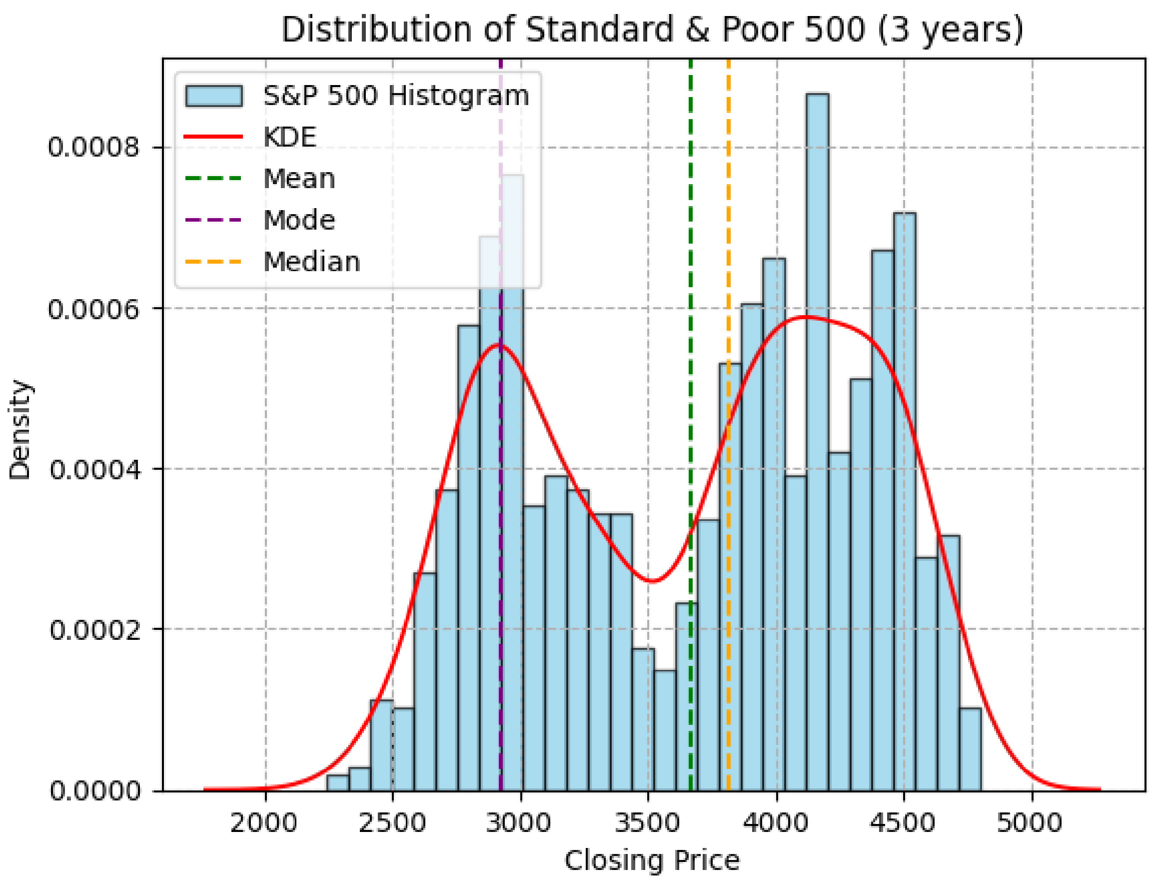
In [7]:

```
print(f'Mean: {X_mean_value:.2f}')
print(f'Median: {X_median_value:.2f}')
print(f'Mode: {X_mode_value:.2f}')
```

```
Mean: 3667.39
Median: 3818.82
Mode: 2926.46
```

In [8]:

```
# Assuming X.Close is your data for the distribution plot
plt.hist(X.Close, bins=30, density=True, alpha=0.7, color='skyblue', edgecolor='

# Overlay a KDE plot using Seaborn
sns.kdeplot(X.Close, color='red', label='KDE')
plt.axvline(X_mean_value, color='green', linestyle='--', label='Mean')
plt.axvline(X_mode_value, color='purple', linestyle='--', label='Mode')
plt.axvline(X_median_value, color='orange', linestyle='--', label='Median')
plt.xlabel("Closing Price")
plt.ylabel("Density")
plt.title("Distribution of Standard & Poor 500 (3 years)")
plt.grid(True, ls='--')
plt.legend()
plt.show()
```

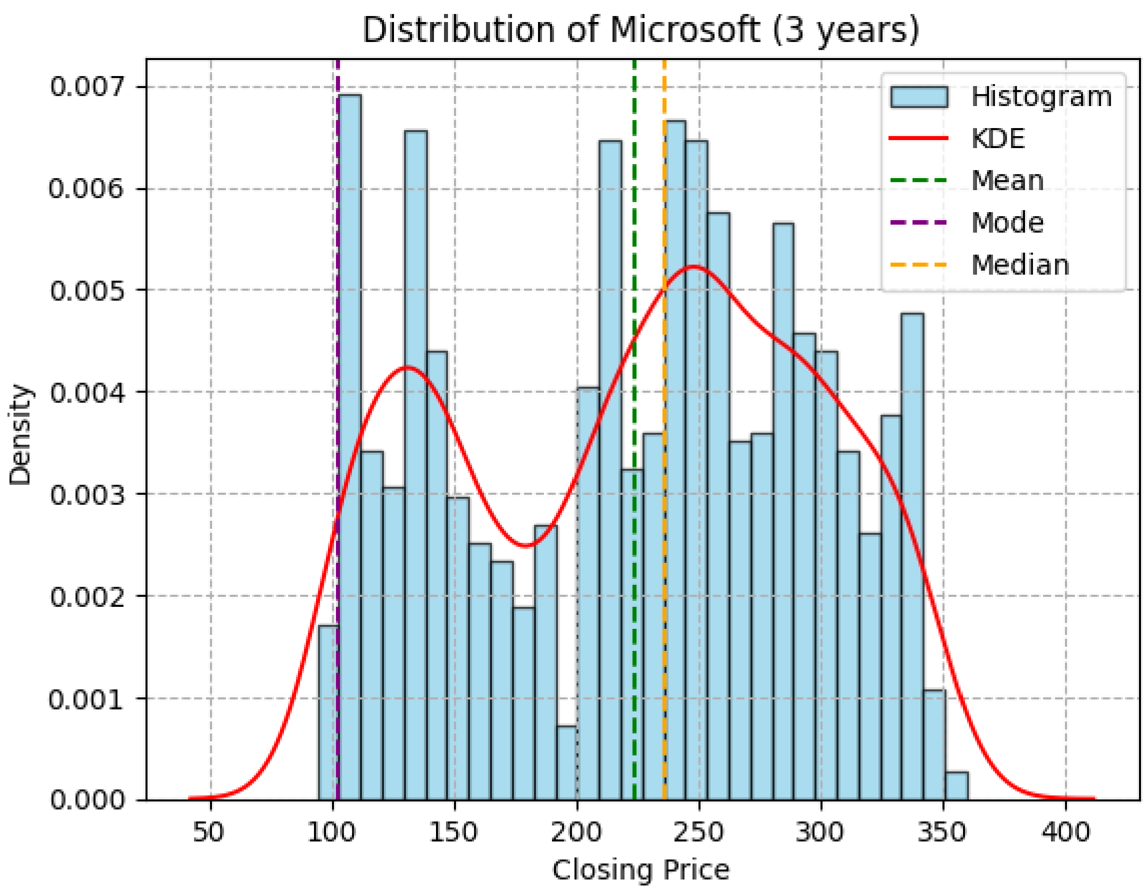


In [9]:
```
y_mean_value = np.mean(y.Close)
y_mode_value = y.Close.mode().values[0]
y_median_value = np.median(y.Close)
```

In [10]:
```
print(f'Mean: {y_mean_value:.2f}')
print(f'Median: {y_median_value:.2f}')
print(f'Mode: {y_mode_value:.2f}')
```

```
Mean: 223.80
Median: 235.61
Mode: 102.80
```

In [11]:
```
plt.hist(y.Close, bins=30, density=True, alpha=0.7, color='skyblue', edgecolor='
sns.kdeplot(y.Close, color='red', label='KDE')
mean_value = np.mean(y.Close)
mode_value = y.Close.mode().values[0]
median_value = np.median(y.Close)
plt.axvline(y_mean_value, color='green', linestyle='--', label='Mean')
plt.axvline(y_mode_value, color='purple', linestyle='--', label='Mode')
plt.axvline(y_median_value, color='orange', linestyle='--', label='Median')
plt.xlabel("Closing Price")
plt.ylabel("Density")
plt.title("Distribution of Microsoft (3 years)")
plt.grid(True, ls='--')
plt.legend()
plt.show()
```



In [12]:
```
import scipy.stats as stats
```

In [13]:
```
#skewness for X.Close (Standard & Poor 500)
X_skewness = stats.skew(X.Close)

#skewness for y.Close (Microsoft)
y_skewness = stats.skew(y.Close)

print(f"Skewness for Standard & Poor 500 (X.Close): {X_skewness:.2f}")
print(f"Skewness for Microsoft (y.Close): {y_skewness:.2f}")
```

```
Skewness for Standard & Poor 500 (X.Close): -0.16
Skewness for Microsoft (y.Close): -0.15
```

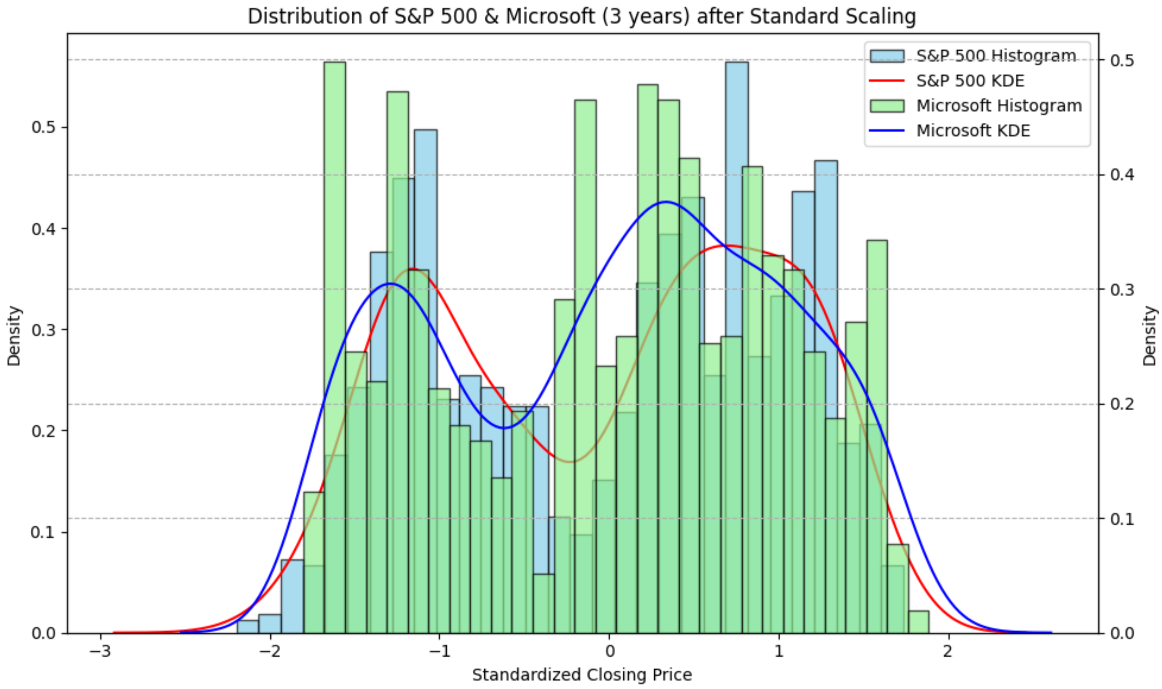
Comparing Distribution Plot

In [14]:
```
# Scale the data using StandardScaler
scaler = StandardScaler()
scaled_X = scaler.fit_transform(X.Close.values.reshape(-1, 1))
```

In [15]:
```
# Scale the data for the second plot (Microsoft)
scaled_y = scaler.fit_transform(y.Close.values.reshape(-1, 1))
```

In [16]:
```
fig, ax1 = plt.subplots(figsize=(10, 6))
ax1.hist(scaled_X, bins=30, density=True, alpha=0.7, color='skyblue', edgecolor=
sns.kdeplot(scaled_X[:, 0], color='red', label='S&P 500 KDE')
ax1.set_xlabel("Standardized Closing Price")
ax1.set_ylabel("Density")
ax1.set_title("Distribution of S&P 500 & Microsoft (3 years) after Standard Scal

ax2 = ax1.twinx()
```

```
ax2.hist(scaled_y, bins=30, density=True, alpha=0.7, color='lightgreen', edgecol
sns.kdeplot(scaled_y[:, 0], color='blue', label='Microsoft KDE')
ax2.set_ylabel("Density")

lines1, labels1 = ax1.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
lines = lines1 + lines2
labels = labels1 + labels2
ax1.legend(lines, labels, loc='upper right')
plt.tight_layout()
plt.grid(True, ls='--')
plt.show()
```



Our skewness do not get affected during featuring scaling. When we standardize data (subtract the mean and divide by the standard deviation), we are essentially linearly transforming the data. This transformation do not change the relative ordering of the data points. It only shifts and scales the data along the same distribution shape. Therefore, if your data had a positive or negative skewness before standardization, it will maintain the same skewness after standardization

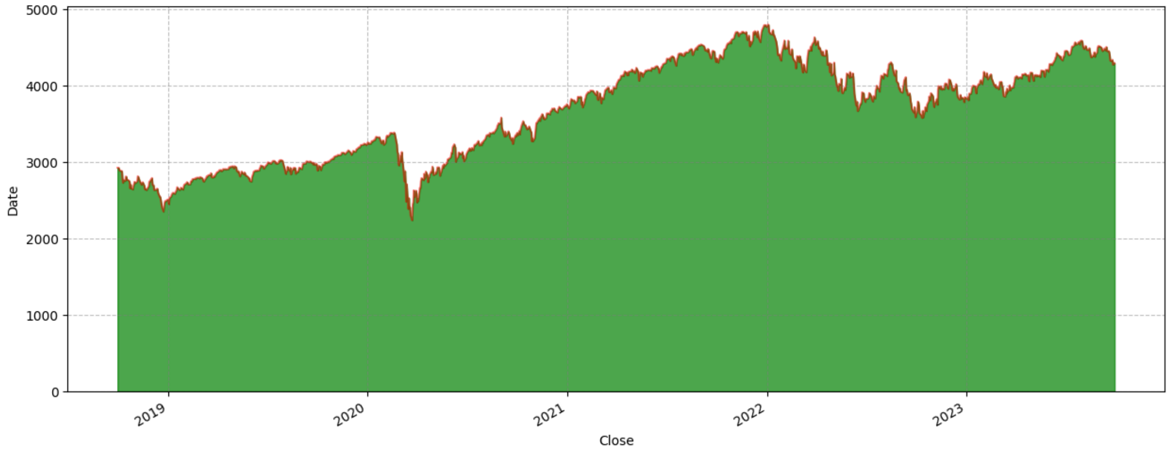
Linear transformations include addition, subtraction, multiplication, and division, all of which are involved in standardization.

In [17]:
```
# Skewness for X.Close (Standard & Poor 500)
X_scaled_skewness = stats.skew(scaled_X, axis=None)
# Skewness for y.Close (Microsoft)
y_scaled_skewness = stats.skew(scaled_y, axis=None)
print(f"Skewness for Standard & Poor 500 (X.Close): {X_scaled_skewness:.2f}")
print(f"Skewness for Microsoft (y.Close): {y_scaled_skewness:.2f}")
```

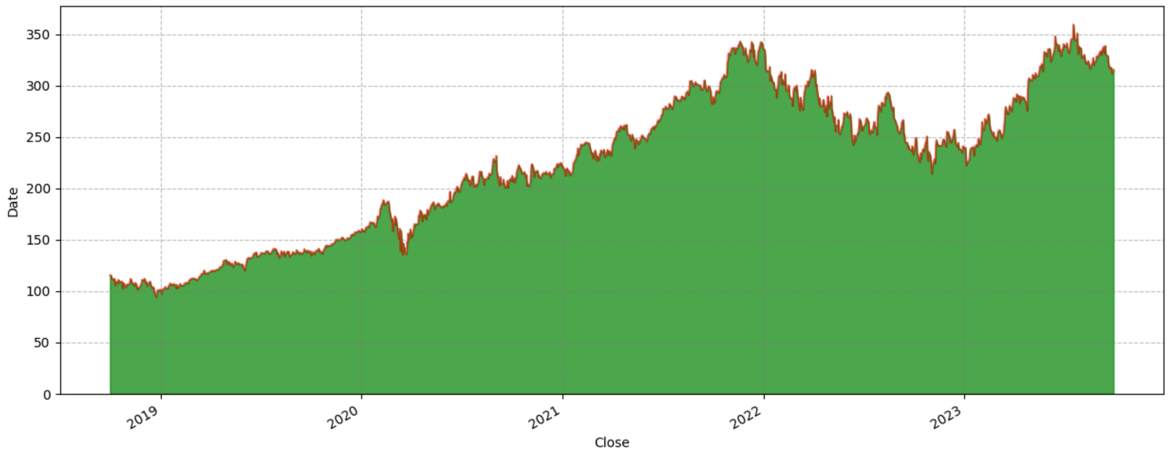
```
Skewness for Standard & Poor 500 (X.Close): -0.16
Skewness for Microsoft (y.Close): -0.15
```

Standard & Poor 500 has karl pearson skewness of -0.16 and Microsoft has -0.16, This means that the distribution of the dataset is slightly right-skewed (positively skewed). In a right-skewed distribution, the tail on the right side (toward larger values) is longer or more pronounced than the tail on the left side.

In [18]:
```
#Snp500 plot
plt.figure(figsize=(15,6))
X.Close.plot(color="red", alpha=0.5)
plt.stackplot(X.index, X.Close, alpha=0.7, color="green")
plt.grid(True, linestyle="--", color="grey", alpha=0.5)
plt.xlabel("Close")
plt.ylabel("Date")
plt.show()
```



In [19]:
```
#microsoft plot
plt.figure(figsize=(15,6))
y.Close.plot(color="red", alpha=0.5)
plt.stackplot(y.index, y.Close, alpha=0.7, color="green")
plt.grid(True, linestyle="--", color="grey", alpha=0.5)
plt.xlabel("Close")
plt.ylabel("Date")
plt.show()
```



In [20]:
```
#creating data set, where X is snp500 and y is micrsoft
dataset = {"X":X.Close.values, "y":y.Close.values}
```

In [21]:
```
print(len(dataset['X']))
print(len(dataset['y']))
```

```
1258
1258
```

In [22]:
```
df = pd.DataFrame(dataset)
```

$$x_{\text{stand}} = \frac{x - \text{mean}(x)}{\text{standard deviation }(x)}$$

In [23]:
```
#standardisation
df_standard_scale = (df - df.mean()) / df.std()
```

In [24]:
```
df_standard_scale[:5]
```

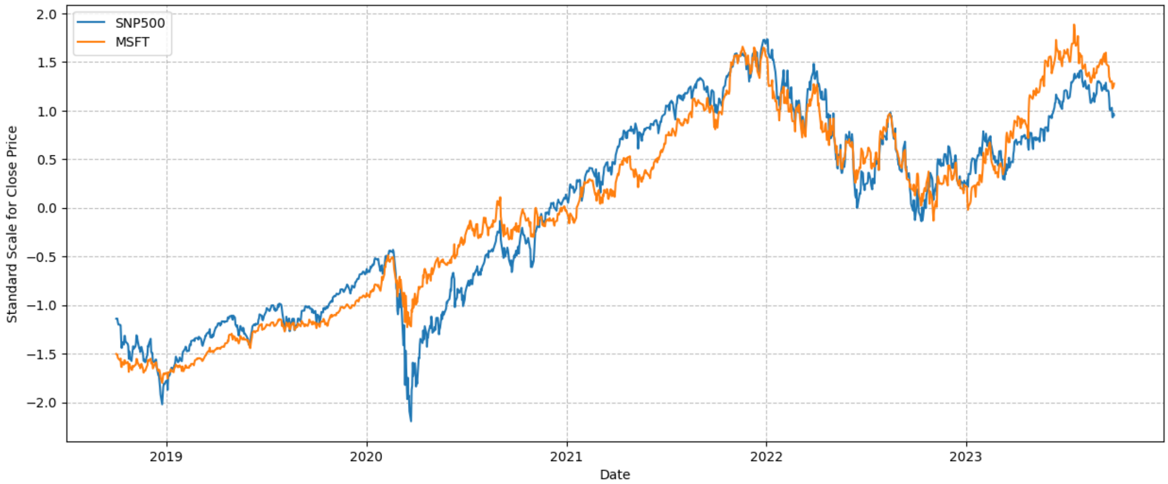
Out[24]:

	X	y
0	-1.140822	-1.503019
1	-1.142604	-1.509410
2	-1.139410	-1.509132
3	-1.176116	-1.542196
4	-1.200751	-1.551365

STANDARD SCALER

StandardScaler, also known as Z-score normalization, transforms the data such that it has a mean (average) of 0 and a standard deviation of 1. This scaling technique is particularly useful when your data follows a Gaussian (normal) distribution. It subtracts the mean from each data point and divides by the standard deviation. The result is that the scaled data has a distribution with a mean of 0 and a standard deviation of 1.

In [25]:
```
#plotting scaler values for time series analysis
plt.figure(figsize=(15,6))
plt.plot(X.index, df_standard_scale['X'], label="SNP500")
plt.plot(y.index, df_standard_scale['y'], label="MSFT")
plt.grid(True, linestyle="--", color="grey", alpha=0.5)
plt.xlabel("Date")
plt.ylabel("Standard Scale for Close Price")
plt.legend()
plt.show()
```



MIN MAX SCALER

MinMaxScaler, on the other hand, scales the data to a specific range, typically between 0 and 1. It does this by subtracting the minimum value of the feature and then dividing by the range (the difference between the maximum and minimum values). The scaled data is constrained to the specified range.

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

In [26]:

```
#Min Max scaler
data_mim_max_scaled = (df - df.min()) / (df.max() - df.min())
```

In [27]:

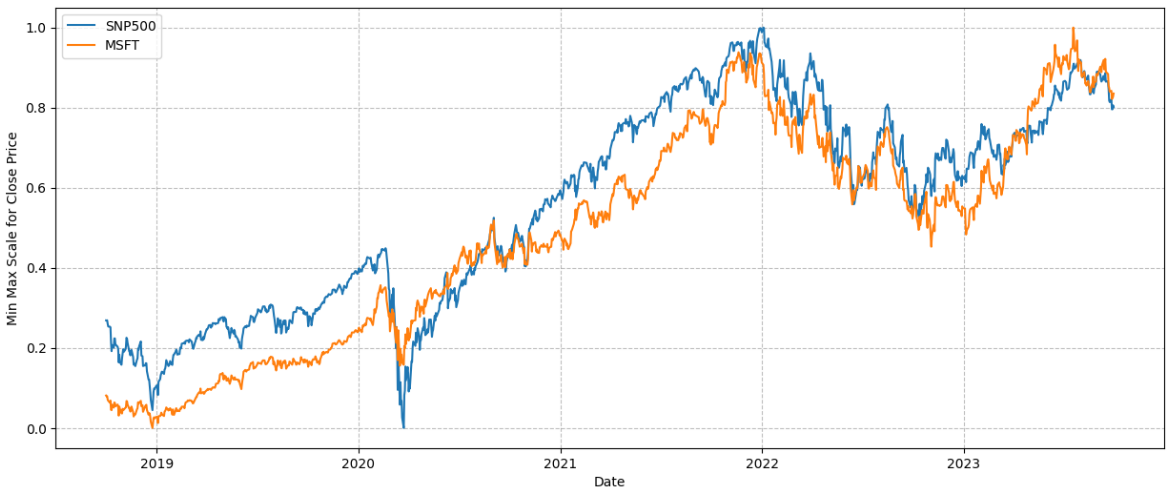
```
data_mim_max_scaled [:5]
```

Out[27]:

	X	y
0	0.268522	0.080947
1	0.268068	0.079213
2	0.268881	0.079289
3	0.259542	0.070320
4	0.253275	0.067832

In [28]:

```
plt.figure(figsize=(15,6))
plt.plot(X.index, data_mim_max_scaled ['X'], label="SNP500")
plt.plot(y.index, data_mim_max_scaled ['y'], label="MSFT")
plt.grid(True, linestyle="--", color="grey", alpha=0.5)
plt.xlabel("Date")
plt.ylabel("Min Max Scale for Close Price")
plt.legend()
plt.show()
```



SNP500 and Microsoft are showing some types of similar pattern during time series analysis.

LINEAR REGRESSION

In [29]:
```
# Extract the 'X' column data from the DataFrame and convert it to a 2D NumPy ar
X = df['X'].values.reshape(-1, 1)

# Extract the 'y' column data from the DataFrame and convert it to a 2D NumPy ar
y = df['y'].values.reshape(-1, 1)
```

In [30]:
```
X.ndim
```

Out[30]: 2

In [31]:
```
y.ndim
```

Out[31]: 2

In [32]:
```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.80, randon
```

In [33]:
```
from sklearn.linear_model import LinearRegression
```

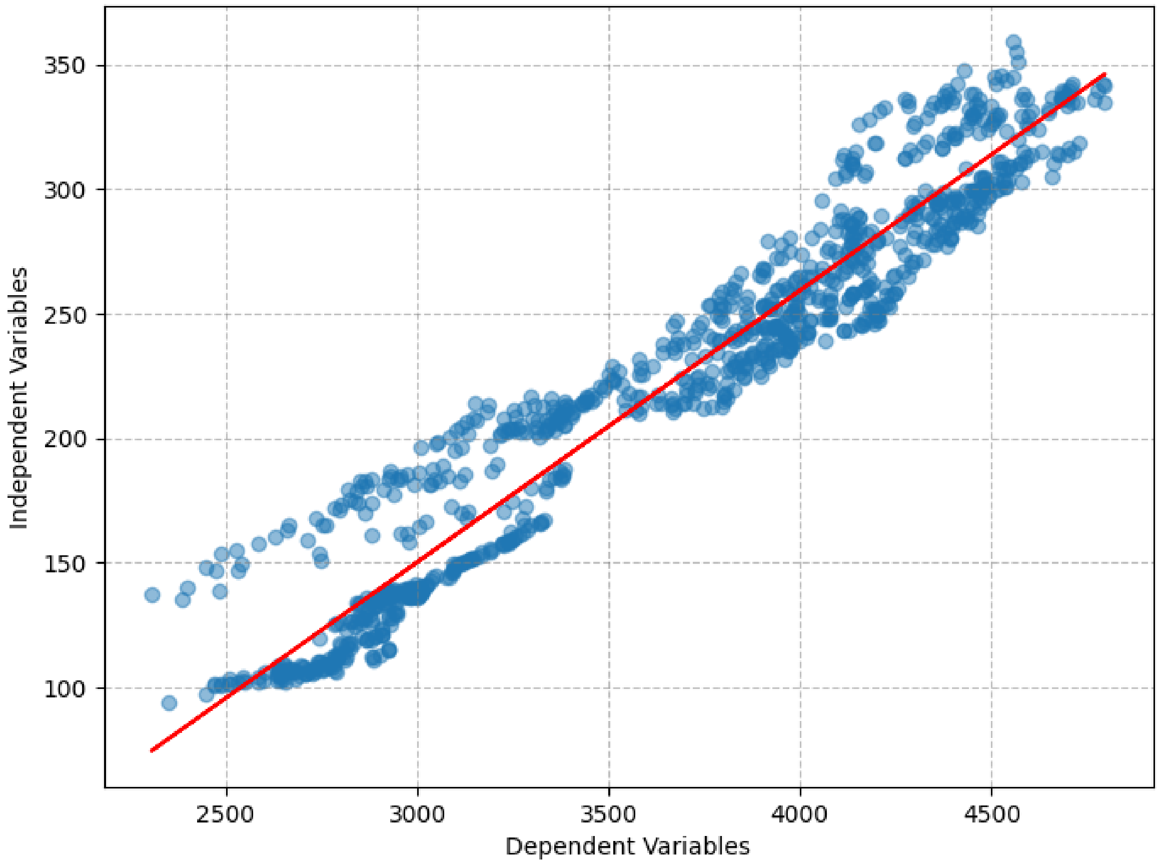
In [34]:
```
# Create an instance of the LinearRegression model
model = LinearRegression()
```

In [35]:
```
#Training model
model.fit(X_train,y_train)
```

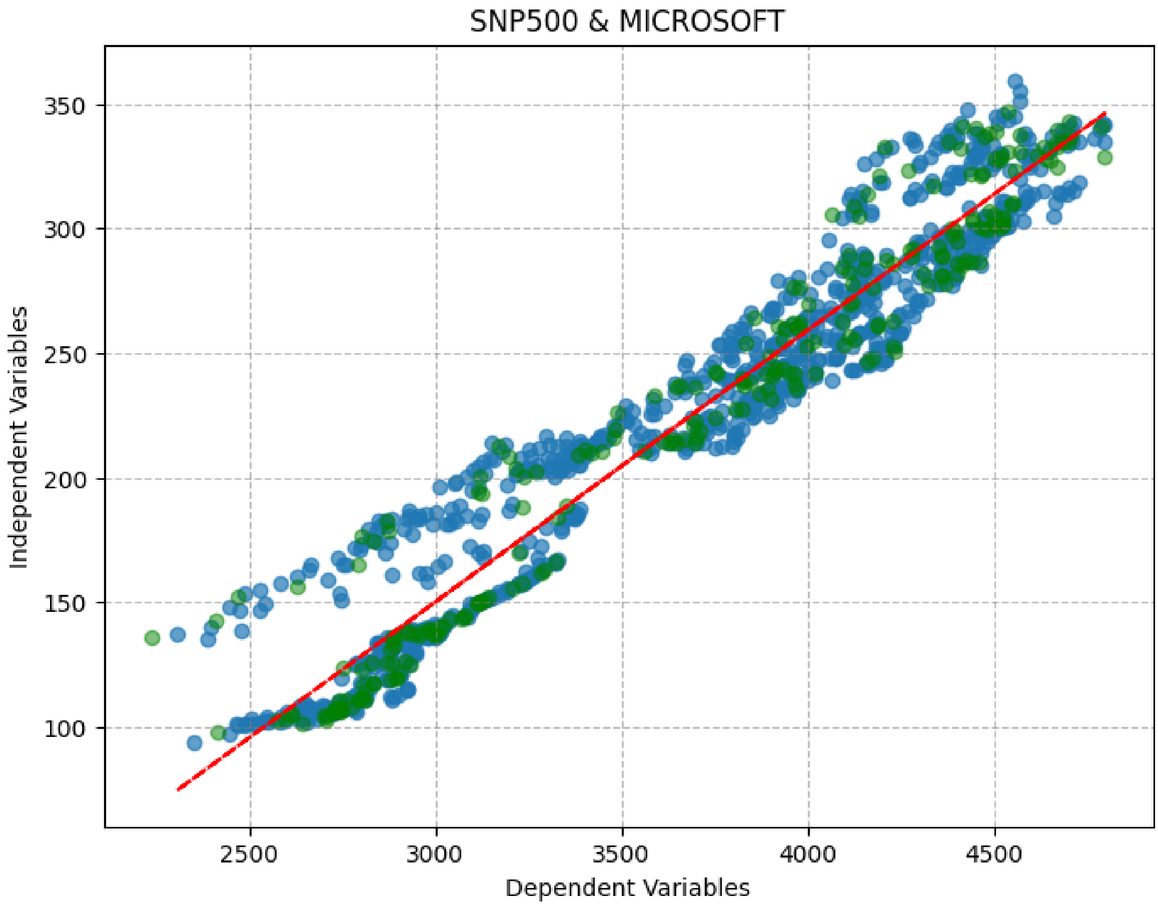
Out[35]:
```
▾ LinearRegression
LinearRegression()
```

In [36]:
```
y_pred = model.predict(X_test)
```

In [37]:
```
predictions = Plot(X_test, y_test, y_pred)
predictions.linear_predict()
```



In [38]:
```
plt.figure(figsize=(8,6))
plt.scatter(X_test, y_test, alpha = 0.7)
plt.scatter(X_train, y_train, color='green', alpha=0.5)
plt.plot(X_test, y_pred, color='red', ls='--')
plt.grid(True, linestyle="--", color="grey", alpha=0.5)
plt.xlabel("Dependent Variables")
plt.ylabel("Independent Variables")
plt.title('SNP500 & MICROSOFT')
plt.show()
```



In [39]:
```
print("Number of coefficients:", len(model.coef_))

print("Estimated coefficients: {}".format(model.coef_[0][0]))
```

```
Number of coefficients: 1
Estimated coefficients: 0.10909012446548745
```

In [40]:
```
print("Estimated intercept: {}".format(model.intercept_[0]))
```

```
Estimated intercept: -177.00969367081865
```

R SQAURE

$$R^2 = 1 - \frac{SS_{RES}}{SS_{TOT}} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \overline{y})^2}$$

In [41]:
```
#Calcaulting R square
rsquare = r2_score(y_test, y_pred)
```

In [42]:
```
print(f'R Sqaure : {round(rsquare,4)}')
```

```
R Sqaure : 0.9234
```

MEAN ABSOLUTE ERROR

$$\mathrm{MAE} = \underbrace{\frac{1}{n}\sum_{i=1}^{n}}_{\text{test set}} |\underbrace{y_i}_{\text{predicted vaue}} - \underbrace{\hat{y}_i}_{\text{actual value}}|$$

In [43]:
```
# Calculate Mean Absolute Error (MAE)
mae = mean_absolute_error(y_test, y_pred)
print(f"MAE: {mae:.2f}")
```

```
MAE: 15.87
```

MEAN SQUARE ERROR

$$\mathrm{MSE} = \underbrace{\frac{1}{n}\sum_{i=1}^{n}}_{\text{Mean}} \underbrace{(\overset{\substack{\text{Actual}\\ \text{output}\\ \downarrow}}{Y_i} - \overset{\substack{\text{Predicted}\\ \text{output}\\ \downarrow}}{\hat{Y}_i})^2}_{\text{Squares of the errors}}$$

In [44]:
```
# Calculate Mean Squared Error (MSE)
mse = mean_squared_error(y_test, y_pred)
print(f"MSE: {mse:.2f}")
```

```
MSE: 384.22
```

ROOT MEAN SQUARE

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

In [45]:
```
# Calculate Root Mean Squared Error (RMSE)
rmse = np.sqrt(mse)
print(f"RMSE: {rmse:.2f}")
```

```
RMSE: 19.60
```

CROSS VALIDATION

Cross Validation on Train set

In [46]:
```
from sklearn.model_selection import cross_val_score
```

In [47]:
```
# Compute 5-fold cross-validation scores: cv_scores
cv_scores_five_fold = cross_val_score(model, X_train, y_train, cv=5, scoring='r2
```

In [48]:
```
# The R-squared scores for each fold
for i, r2 in enumerate(cv_scores_five_fold, 1):
    print(f"Fold {i} R-squared Score: {r2:.4f}")
```

```
Fold 1 R-squared Score: 0.9459
Fold 2 R-squared Score: 0.9572
Fold 3 R-squared Score: 0.9465
Fold 4 R-squared Score: 0.8931
Fold 5 R-squared Score: 0.9447
```

In [49]:
```
print("Average 5-Fold CV Score: {}".format(np.mean(cv_scores_five_fold).round(2)
```

```
Average 5-Fold CV Score: 0.94
```

Cross Validation on Test set

In [50]:
```
# Compute 5-fold cross-validation scores: cv_scores
cv_scores_five_fold = cross_val_score(model, X_test, y_test, cv=5, scoring='r2')
```

In [51]:
```
# The R-squared scores for each fold
for i, r2 in enumerate(cv_scores_five_fold, 1):
    print(f"Fold {i} R-squared Score: {r2:.4f}")
```

```
Fold 1 R-squared Score: 0.9155
Fold 2 R-squared Score: 0.9276
Fold 3 R-squared Score: 0.9376
Fold 4 R-squared Score: 0.9169
Fold 5 R-squared Score: 0.9173
```

In [52]:
```
print("Average 5-Fold CV Score: {}".format(np.mean(cv_scores_five_fold).round(2)
```

```
Average 5-Fold CV Score: 0.92
```

Evaluating Train & Test Performance

In [53]:
```
# Evaluating training set performance
print("Training set score: {:.2}".format(model.score(X_train, y_train)))
# Evaluating test set performance
print("Test set score: {:.2}".format(model.score(X_test, y_test)))
```

```
Training set score: 0.94
Test set score: 0.92
```

DEEP LEARNING MODEL

In [54]:
```
#Independent and Dependent Variable
X = df['X'].values.reshape(-1,1)
y = df['y'].values.reshape(-1,1)
```

In [55]:
```
# Split the dataset into three sets: training, validation, and testing

# Step 1: Split the original dataset into a temporary training set and a test se
X_train_temp, X_test_tf, y_train_temp, y_test_tf = train_test_split(X, y, test_s
```

```
# Step 2: Split the temporary training set into the final training set and a val
X_train_tf, X_val, y_train_tf, y_val = train_test_split(X_train_temp, y_train_te

# Now you have three sets:
# - X_train_tf and y_train_tf: The final training set used for model training.
# - X_val and y_val: The validation set used for tuning hyperparameters and moni
# - X_test_tf and y_test_tf: The test set used for evaluating the final model's
```

In [56]:
```
scaler = StandardScaler()
scaler
```

Out[56]:
```
▾ StandardScaler
StandardScaler()
```

In [57]:
```
X_train[:5]
```

Out[57]:
```
array([[3962.70996094],
       [3963.93994141],
       [3655.04003906],
       [3557.54003906],
       [3325.54003906]])
```

In [58]:
```
X_test[:5]
```

Out[58]:
```
array([[2884.42993164],
       [3357.01000977],
       [2599.94995117],
       [3986.15991211],
       [2941.76000977]])
```

In [59]:
```
# Scale the input features for all three sets (training, validation, and testing

# Step 1: Scale the features in the final training set
X_train_scaled = scaler.fit_transform(X_train_tf.reshape(-1, 1))

# Step 2: Scale the features in the validation set using the same scaler
X_val_scaled = scaler.transform(X_val.reshape(-1, 1))

# Step 3: Scale the features in the test set using the same scaler
X_test_scaled = scaler.transform(X_test_tf.reshape(-1, 1))

# Scaling is important to ensure that all input features have similar scales,
# which can help improve the performance of machine learning models.
# The `fit_transform` method is used to fit the scaler to the training data and
# then transform both the training and validation/test data based on the scaling
# parameters learned from the training data.
```

In [60]:
```
# Defining a Sequential Keras model
# Create a Sequential model object
model = Sequential([
    #Add a Dense layer to the model
    Dense(units=1,        # Number of output units (1 for regression)
          input_shape=(1,),  # Input shape (1 feature)
          activation='linear',  # Linear activation for regression
          use_bias=True)       # Use bias terms in the layer
])
```

```python
#Compile the model
model.compile(
    optimizer='sgd',          # Stochastic Gradient Descent optimizer
    loss='mean_squared_error'  # Mean Squared Error loss function for regression
)

# The defined model is a simple linear regression model with one input feature,
# one output unit, and linear activation. It uses stochastic gradient descent (S
# as the optimizer and mean squared error (MSE) as the loss function for trainin
```

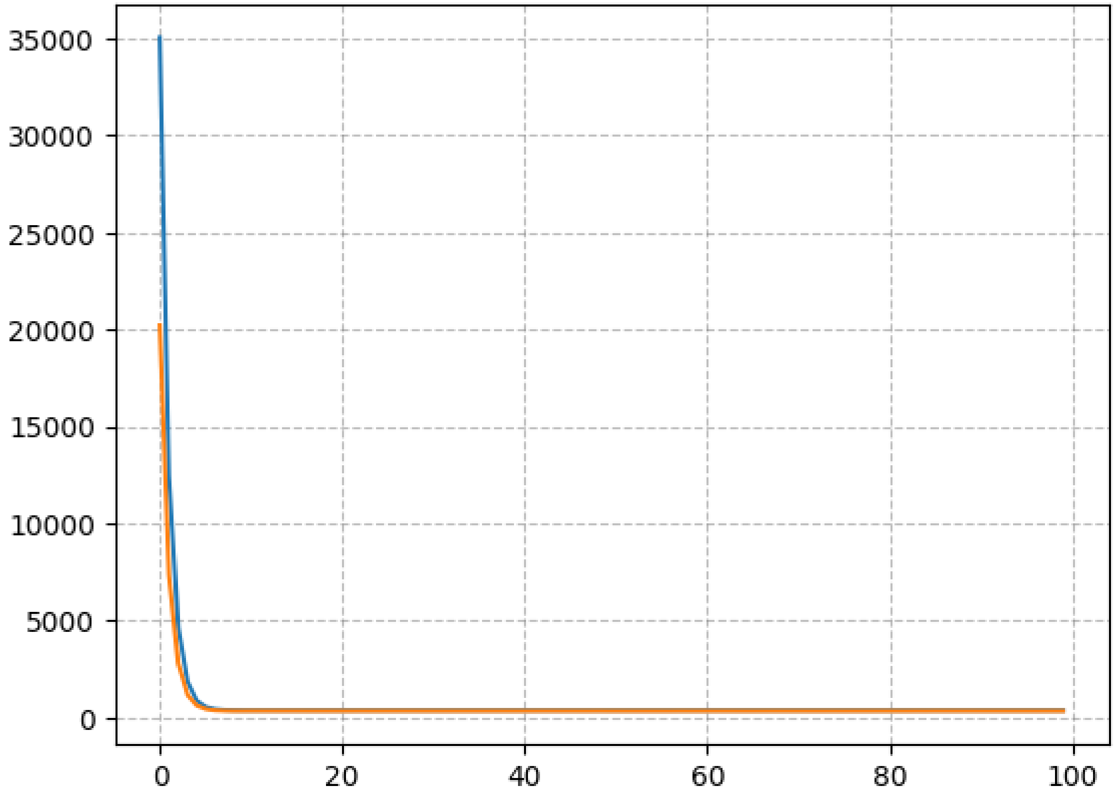
In [61]:
```python
# Train the Keras model
history = model.fit(
    X_train_scaled,          # Training features (scaled)
    y_train_tf,              # Training target
    validation_data=(X_val_scaled, y_val),  # Validation data
    epochs=100,              # Number of training epochs
    verbose=0                # Set to 0 for no training progress output
)

# The `fit` method trains the Keras model using the specified training data
# (features and target) for a specified number of training epochs. It also
# uses the validation data to monitor the model's performance during training.
# Setting `verbose` to 0 suppresses training progress output.
```

In [62]:
```python
# Make predictions using the trained Keras model
y_pred_tf = model.predict(X_train_scaled)
y_pred_tf[:5]
```

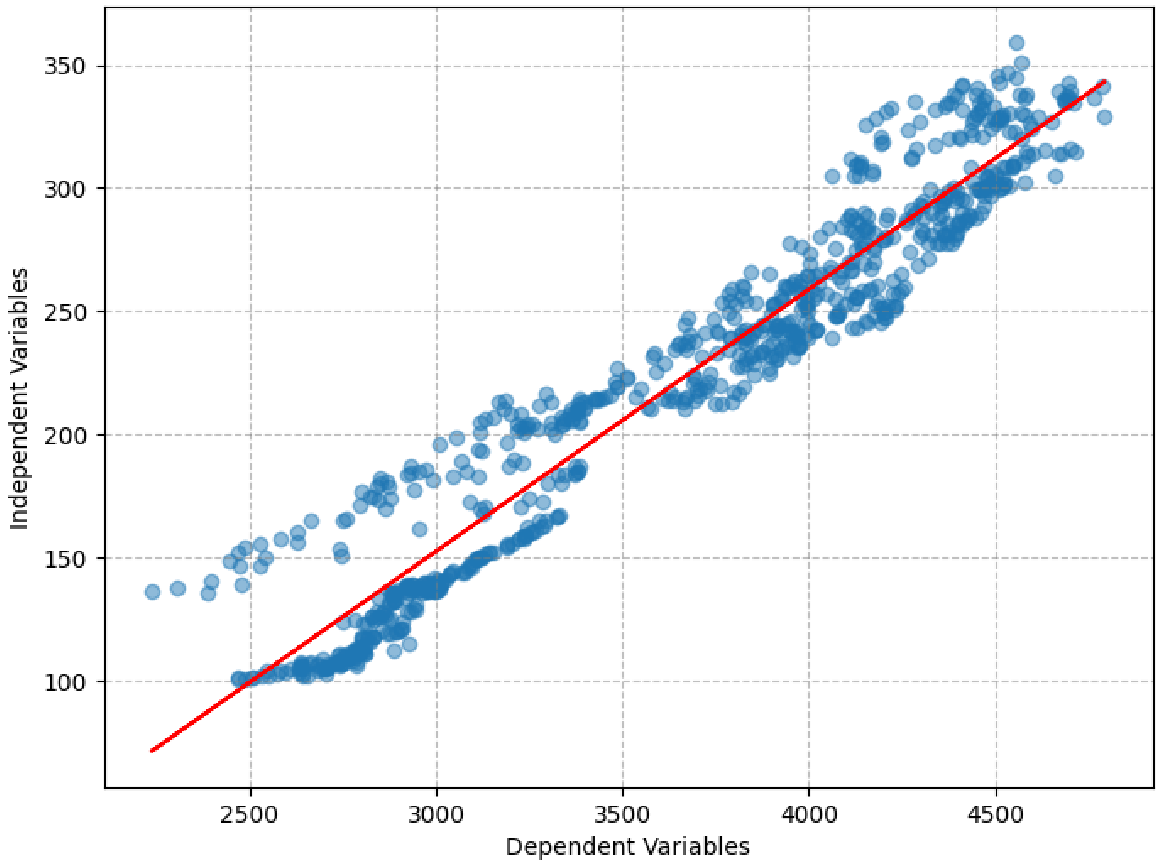
```
26/26 [==============================] - 0s 1ms/step
```

Out[62]:
```
array([[298.18542],
       [245.09006],
       [270.50543],
       [300.99817],
       [131.01462]], dtype=float32)
```

In [63]:
```python
# Plot the training and validation loss curves
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.grid(True, linestyle="--", color="grey", alpha=0.5)
```

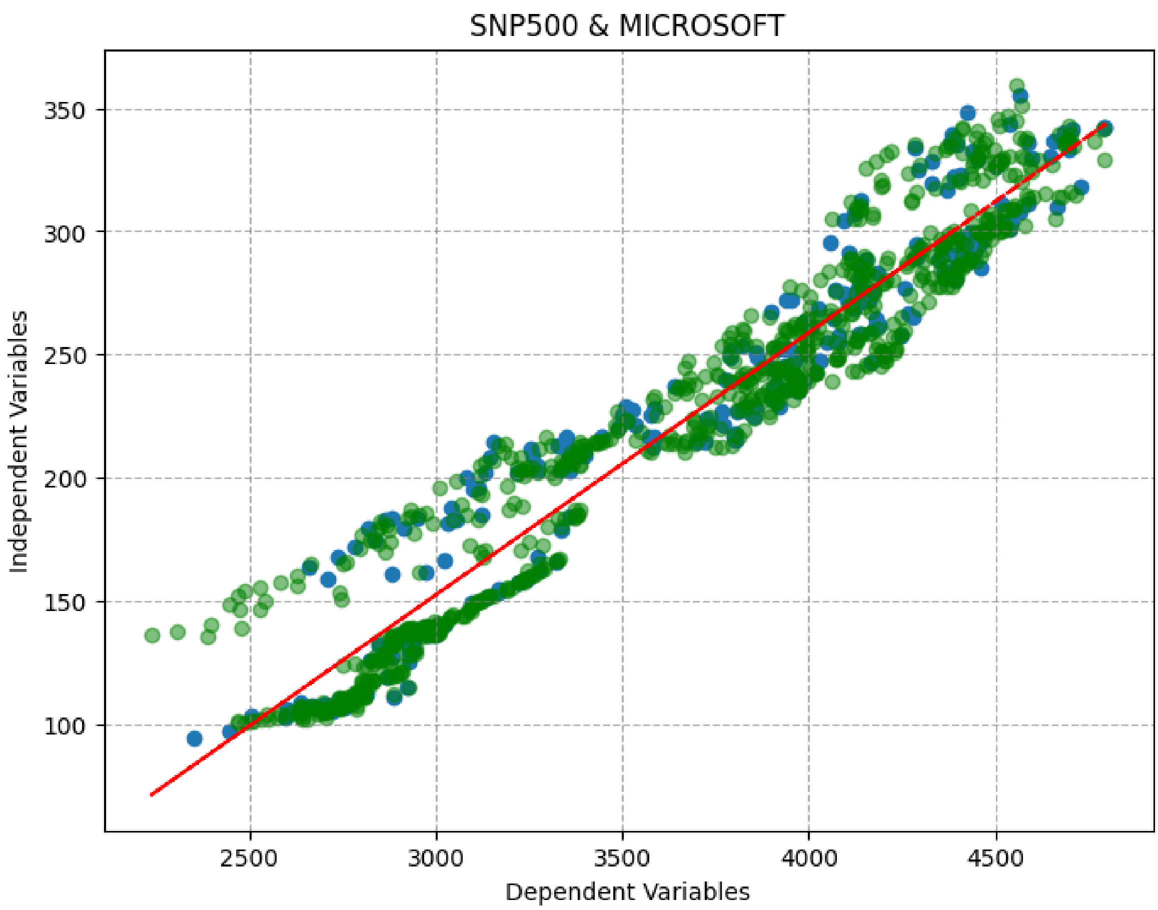


In [64]:
```
#plotting best fit line
predictions_tf = Plot(X_train_tf,y_train_tf, y_pred_tf)
predictions_tf.linear_predict()
```



In [65]:
```
#plotting best fit line
#plottinf scatter plot of training and test set
plt.figure(figsize=(8,6))
```

```
plt.scatter(X_test_tf, y_test_tf)
plt.scatter(X_train_tf, y_train_tf, color='green', alpha=0.5)
plt.plot(X_train_tf, y_pred_tf, color='red', ls='--')
plt.grid(True, linestyle="--", color="grey", alpha=0.6)
plt.xlabel("Dependent Variables")
plt.ylabel("Independent Variables")
plt.title('SNP500 & MICROSOFT')
plt.show()
```



In [66]:
```
#calculating R sqaure
rsquare = r2_score(y_train_tf, y_pred_tf)
```

In [67]:
```
print(f"R Square: {rsquare}")
```

```
R Square: 0.9267797980703649
```

In [68]:
```
# Calculating Mean Absolute Error (MAE)
mae = mean_absolute_error(y_train_tf, y_pred_tf)
print(f"MAE: {mae:.2f}")
```

```
MAE: 15.87
```

In [69]:
```
# Calculating Mean Squared Error (MSE)
mse = mean_squared_error(y_train_tf, y_pred_tf)
print(f"MSE: {mse:.2f}")
```

```
MSE: 377.38
```

In [70]:
```
# Calculating Root Mean Squared Error (RMSE)
rmse = np.sqrt(mse)
print(f"RMSE: {rmse:.2f}")
```

```
RMSE: 19.43
```

KFOLD CROSS VALIDATION WITH FIVE FOLDS

In [71]:
```
X = (X - X.mean())/X.std()
y = (y - y.mean())/y.std()
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.80, randon
```

In [72]:
```
from sklearn.model_selection import KFold
```

In [73]:
```
def r_squared(y_true, y_pred):
    # Calculating the sum of squared residuals (SSR)
    ssr = np.sum((y_true - y_pred) ** 2)
    # Calculating the total sum of squares (SST)
    sst = np.sum((y_true - np.mean(y_true)) ** 2)
    # Calculating R-squared as 1 - (SSR / SST)
    r_squared = 1 - (ssr / sst)
    # Return the R-squared value
    return r_squared
```

In [74]:
```
#a list to store R-squared scores
r2_scores = []

#the number of folds
n_splits = 5
kf = KFold(n_splits=n_splits)
```

In [77]:
```
# Perform k-fold cross-validation
for train_indices, val_indices in kf.split(X_train):
    X_train_fold, X_val_fold = X_train[train_indices], X_train[val_indices]
    y_train_fold, y_val_fold = y_train[train_indices], y_train[val_indices]

    # Create a new model for each fold
    model = Sequential([
        Dense(units=1, input_shape=(1,), activation='linear', use_bias=True)
    ])
    model.compile(optimizer='sgd', loss='mean_squared_error')
    # Train the model on the current fold
    model.fit(X_train_fold, y_train_fold, epochs=100, verbose=0)
    # Make predictions on the validation fold
    y_pred = model.predict(X_val_fold)
    #  R-squared for this fold
    r2 = r_squared(y_val_fold, y_pred)
    r2_scores.append(r2)
```

```
2/2 [==============================] - 0s 3ms/step
2/2 [==============================] - 0s 4ms/step
2/2 [==============================] - 0s 2ms/step
2/2 [==============================] - 0s 3ms/step
2/2 [==============================] - 0s 2ms/step
```

In [78]:
```
#R-squared scores for each fold
for i, r2 in enumerate(r2_scores):
    print(f"Fold {i + 1} R-squared: {r2}")
```

```
Fold 1 R-squared: 0.9458133169669112
Fold 2 R-squared: 0.9569775586047692
Fold 3 R-squared: 0.9466770487728365
Fold 4 R-squared: 0.8942041889327921
Fold 5 R-squared: 0.943779062384023
Fold 6 R-squared: 0.9455893687286455
Fold 7 R-squared: 0.9572795226624567
Fold 8 R-squared: 0.94666217616754
Fold 9 R-squared: 0.8946943421162997
Fold 10 R-squared: 0.9439804118820051
```

In [79]:

```
# Mean R-squared score
mean_r2 = np.mean(r2_scores)
print(f"Mean R-squared: {mean_r2}")
```

```
Mean R-squared: 0.9375656997218279
```

KFOLD CROSS VALIDATION WITH TEN FOLDS

In [80]:

```
# Define the number of folds
n_splits = 10
kf = KFold(n_splits=n_splits)
```

In [81]:

```
#list to store R-squared scores
r2_scores = []

# Perform 10-fold cross-validation
for train_indices, val_indices in kf.split(X_train):
    X_train_fold, X_val_fold = X_train[train_indices], X_train[val_indices]
    y_train_fold, y_val_fold = y_train[train_indices], y_train[val_indices]

    # Create a new model for each fold
    model = Sequential([
        Dense(units=1, input_shape=(1,), activation='linear', use_bias=True)
    ])
    model.compile(optimizer='sgd', loss='mean_squared_error')
    # Train the model on the current fold
    model.fit(X_train_fold, y_train_fold, epochs=100, verbose=0)
    # Make predictions on the validation fold
    y_pred = model.predict(X_val_fold)
    # Calculate R-squared for this fold
    r2 = r_squared(y_val_fold, y_pred)
    r2_scores.append(r2)
```

```
1/1 [==============================] - 0s 33ms/step
1/1 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 55ms/step
1/1 [==============================] - 0s 36ms/step
1/1 [==============================] - 0s 51ms/step
1/1 [==============================] - 0s 35ms/step
1/1 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 39ms/step
1/1 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 38ms/step
```

In [82]:

```
#R-squared scores for each fold
for i, r2 in enumerate(r2_scores):
    print(f"Fold {i + 1} R-squared: {r2}")
```

```
Fold 1 R-squared: 0.9133532763792127
Fold 2 R-squared: 0.9593670302874157
Fold 3 R-squared: 0.9513068675772078
Fold 4 R-squared: 0.9585514618652844
Fold 5 R-squared: 0.9291713576643994
Fold 6 R-squared: 0.9571188657850933
Fold 7 R-squared: 0.8513725309435806
Fold 8 R-squared: 0.9448098465734979
Fold 9 R-squared: 0.933576167964511
Fold 10 R-squared: 0.9605080125990481
```

In [83]:
```
# mean R-squared score
mean_r2 = np.mean(r2_scores)
print(f"Mean R-squared: {mean_r2}")
```

```
Mean R-squared: 0.9359135417639249
```

MANUAL CALCULATION

In [84]:
```
df[:5]
```

Out[84]:

	X	y
0	2924.590088	115.610001
1	2923.429932	115.150002
2	2925.510010	115.169998
3	2901.610107	112.790001
4	2885.570068	112.129997

In [85]:
```
# new column 'XY' by multiplying values in columns 'X' and 'y'
df['XY'] = df['X'] * df['y']
# the mean of the 'X' column
X_mean = sum(df['X']) / len(df['X'])
# the mean of the 'y' column
y_mean = sum(df['y']) / len(df['y'])
#  new column 'X square' by squaring values in the 'X' column
df['X square'] = df['X']**2
# new column 'y_square' by squaring values in the 'y' column
df['y_square'] = df['y']**2
```

In [86]:
```
X_mean
```

Out[86]: 3667.392121671304

In [87]:
```
y_mean
```

Out[87]: 223.7995229821137

In [88]:
```
# Calculating the deviation of each 'X' value from its mean
df['Xi - X mean'] = df['X'] - X_mean
# Calculating the deviation of each 'y' value from its mean
df['yi - y mean'] = df['y'] - y_mean
# Calculating the product of the deviations of 'X' and 'y' from their respective
df['(Xi - X mean)(yi - y mean)'] = df['Xi - X mean'] * df['yi - y mean']
```

```
# Calculating the square of the deviation of 'X' from its mean
df['(Xi - X mean)**2'] = df['Xi - X mean'] * df['Xi - X mean']
```

In [89]:
```
df[:5]
```

Out[89]:

	X	y	XY	X square	y_square	Xi - X mean	yi
0	2924.590088	115.610001	338111.861846	8.553227e+06	13365.672241	-742.802034	-1
1	2923.429932	115.150002	336632.961089	8.546443e+06	13259.522851	-743.962190	-1
2	2925.510010	115.169998	336930.982468	8.558609e+06	13264.128478	-741.882112	-1
3	2901.610107	112.790001	327272.606673	8.419341e+06	12721.584307	-765.782014	-1
4	2885.570068	112.129997	323558.963840	8.326515e+06	12573.136284	-781.822053	-1

In [90]:
```
# Calculating the coefficient b1 using the formula for simple linear regression
b1 = sum(df['(Xi - X mean)(yi - y mean)']) / sum(df['(Xi - X mean)**2'])
print(f'Coefficient: {round(b1,4)}')
```

```
Coefficient: 0.1065
```

In [91]:
```
# Calculatin the intercept bo using the formula for simple linear regression
bo = y_mean - b1 * X_mean
print(f'Intercept: {round(bo,4)}')
```

```
Intercept: -166.7956
```

In [92]:
```
# Extracting the values from the 'X' column in the DataFrame
X_value = df['X']
# Converting the extracted values to a list
X_value = X_value.tolist()
```

In [93]:
```
#empty list to store the predicted values
predicted_values = []

# predicted values for each X_value
for i in X_value:
    y = bo + b1 * i
    predicted_values.append(y)
```

In [94]:
```
# 'y_pred' to the DataFrame and assign it the predicted values
df['y_pred'] = predicted_values
```

In [95]:
```
# 'residual' to the DataFrame and calculating residuals
df['residual'] = df['y'] - df['y_pred']
```

In [96]:
```
df[:5]
```

Out[96]:

	X	y	XY	X square	y_square	Xi - X mean	yi
0	2924.590088	115.610001	338111.861846	8.553227e+06	13365.672241	-742.802034	-1
1	2923.429932	115.150002	336632.961089	8.546443e+06	13259.522851	-743.962190	-1
2	2925.510010	115.169998	336930.982468	8.558609e+06	13264.128478	-741.882112	-1
3	2901.610107	112.790001	327272.606673	8.419341e+06	12721.584307	-765.782014	-1
4	2885.570068	112.129997	323558.963840	8.326515e+06	12573.136284	-781.822053	-1

SST

In [97]:

```
#  mean of 'y'
y_mean = sum(df['y']) / len(df['y'])
# mean of 'X'
X_mean = sum(df['X']) / len(df['X'])
#  Total Sum of Squares (SST)
sst = sum([(y - y_mean)**2 for y in df['y']])
#  SST value
print(f'Total Sum of Square (SST): {sst}')
```

```
Total Sum of Square (SST): 6512931.814715496
```

SSE

In [98]:

```
# Sum of Squares of Residuals (SSE)
sse = sum((df['residual'])**2)
# SSE value
print(f'Sum of Square Residual Error (SSE): {sse}')
```

```
Sum of Square Residual Error (SSE): 468098.16543555376
```

SSR

In [99]:

```
# the squared differences between predicted 'y' values and the mean of 'y'
df['(y_pred - y_mean)**2'] = (df['y_pred'] - y_mean)**2
# Total Sum of Squares of Regression (SSR)
ssr = sum(df['(y_pred - y_mean)**2'])
# SSR value
print(f'Total Sum of Square Regression (SSR): {ssr}')
```

```
Total Sum of Square Regression (SSR): 6044833.649279933
```

R SQUARE

In [100…

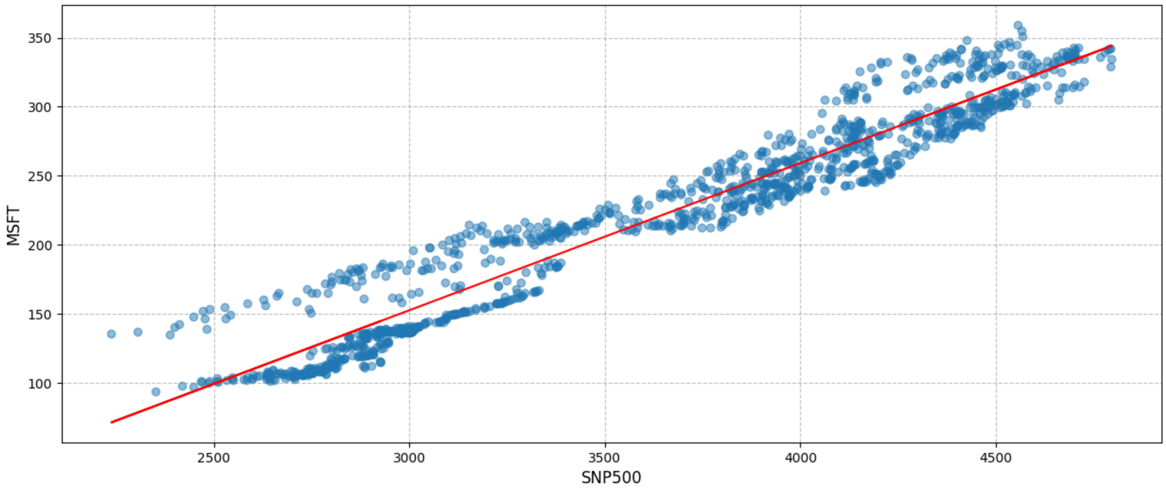
```
#R-squared (coefficient of determination)
r_square = ssr/sst
f"R square: {r_square}"
```

Out[100…
```
'R square: 0.9281278879078806'
```

In [101…
```
#R-squared (coefficient of determination) using an alternative formula
r_square = 1 - sse/sst
f"R square: {r_square}"
```

Out[101…
```
'R square: 0.928127887907882'
```

In [102…
```
plt.figure(figsize=(15,6))
plt.scatter(df['X'], df['y'], alpha=0.5)
plt.plot(df['X'], df['y_pred'], color="red")
plt.grid(True, linestyle="--", color="grey", alpha=0.5)
plt.xlabel("SNP500", fontsize=12)
plt.ylabel("MSFT", fontsize=12)
plt.show()
```



BUILDING NEURAL NETWORK

In [103…
```
# Define the time period for downloading financial data (5 years)
years = "5Y"

# Download historical data for the S&P 500 index (^GSPC) for the specified peric
X = yf.download("^GSPC", period=years)

# Download historical data for Microsoft (MSFT) stock prices for the same period
y = yf.download("MSFT", period=years)
```

```
[*********************100%%**********************]  1 of 1 completed
[*********************100%%**********************]  1 of 1 completed
```

In [104…
```
X[:5]
```

Out[104…

Date	Open	High	Low	Close	Adj Close	Volume
2018-10-01	2926.290039	2937.060059	2917.909912	2924.590088	2924.590088	3375890000
2018-10-02	2923.800049	2931.419922	2919.370117	2923.429932	2923.429932	3432620000
2018-10-03	2931.689941	2939.860107	2921.360107	2925.510010	2925.510010	3625510000
2018-10-04	2919.350098	2919.780029	2883.919922	2901.610107	2901.610107	3510370000
2018-10-05	2902.540039	2909.639893	2869.290039	2885.570068	2885.570068	3340820000

In [105…

```
y[:5]
```

Out[105…

Date	Open	High	Low	Close	Adj Close	Volume
2018-10-01	114.750000	115.680000	114.730003	115.610001	109.532570	18883100
2018-10-02	115.300003	115.839996	114.440002	115.150002	109.096733	20787200
2018-10-03	115.419998	116.180000	114.930000	115.169998	109.115700	16648000
2018-10-04	114.610001	114.760002	111.629997	112.790001	106.860794	34821700
2018-10-05	112.629997	113.169998	110.639999	112.129997	106.235504	29068900

In [106…

```
# Select the "Close" column from the S&P 500 index data and convert it to a NumP
X = X["Close"].to_numpy()
# Reshape the S&P 500 data to a 2D array with a single column
X = X.reshape(-1, 1)
# Select the "Close" column from the Microsoft (MSFT) stock price data and conve
y = y["Close"].to_numpy()
# Reshape the Microsoft data to a 2D array with a single column
y = y.reshape(-1, 1)
```

In [107…

```
#standardscaler
scaler = StandardScaler()
X = scaler.fit_transform(X)
y = scaler.fit_transform(y)
```

In [108…

```
# Split the data into a temporary training set (X_train_temp and y_train_temp) a
# The test set size is 20% of the entire dataset, and no random seed (random_sta
X_train_temp, X_test, y_train_temp, y_test = train_test_split(X, y, test_size=0.

# Further split the temporary training set into a training set (X_train and y_tr
# and a validation set (X_val and y_val)
# The validation set size is 20% of the temporary training set, and no random se
X_train, X_val, y_train, y_val = train_test_split(X_train_temp, y_train_temp, te
```

In [109…

```
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Initialize weights and biases with small random values
        self.weights_input_hidden = np.random.randn(input_size, hidden_size)
        self.bias_hidden = np.zeros((1, hidden_size))
        self.weights_hidden_output = np.random.randn(hidden_size, output_size)
        self.bias_output = np.zeros((1, output_size))

        self.loss_history = []

    def forward(self, input_data):
        # Hidden layer
        self.hidden_input = np.dot(input_data, self.weights_input_hidden) + self
        self.hidden_output = self.hidden_input

        # Output layer
        self.output_input = np.dot(self.hidden_output, self.weights_hidden_outpu
        self.output = self.output_input

    def mean_squared_error(self, y_true, y_pred):
        return np.mean((y_true - y_pred) ** 2)

    def backward(self, input_data, target, learning_rate):
        error = target - self.output

        # Derivatives for the output layer
        delta_output = -error
        d_weights_hidden_output = np.dot(self.hidden_output.T, delta_output)
        d_bias_output = np.sum(delta_output, axis=0, keepdims=True)

        # Derivatives for the hidden layer
        delta_hidden = delta_output.dot(self.weights_hidden_output.T)
        d_weights_input_hidden = np.dot(input_data.T, delta_hidden)
        d_bias_hidden = np.sum(delta_hidden, axis=0, keepdims=True)

        # Update weights and biases
        self.weights_hidden_output -= d_weights_hidden_output * learning_rate
        self.bias_output -= d_bias_output * learning_rate
        self.weights_input_hidden -= d_weights_input_hidden * learning_rate
        self.bias_hidden -= d_bias_hidden * learning_rate

    def train(self, X, y, epochs, learning_rate):
        for epoch in range(epochs):
            self.forward(X)
            loss = self.mean_squared_error(y, self.output)
            self.backward(X, y, learning_rate)

            self.loss_history.append(loss)

            if epoch % 100 == 0:
                print(f"Epoch {epoch}, Loss: {loss:.4f}")

    def r_squared(self, y_true, y_pred):
         # Sum of squared residuals
        ssr = np.sum((y_true - y_pred) ** 2)
```

```
        # Total sum of squares
        sst = np.sum((y_true - np.mean(y_true)) ** 2)
        return 1 - (ssr / sst)

    def predict(self, X):
        self.forward(X)
        return self.output
```

In [144…

```
# size of the input layer
input_size = 1
# size of the hidden layer
hidden_size = 1
# size of the output layer
output_size = 1
# learning rate for the neural network's optimizer
learning_rate = 0.001
# number of training epochs (iterations) for training the neural network
epochs = 25
```

In [145…

```
#an instance of the NeuralNetwork class with the specified hyperparameters
model = NeuralNetwork(input_size, hidden_size, output_size)
# Train the model using the training data with the specified number of epochs an
model.train(X_train, y_train, epochs, learning_rate)
```

```
Epoch 0, Loss: 0.1550
```

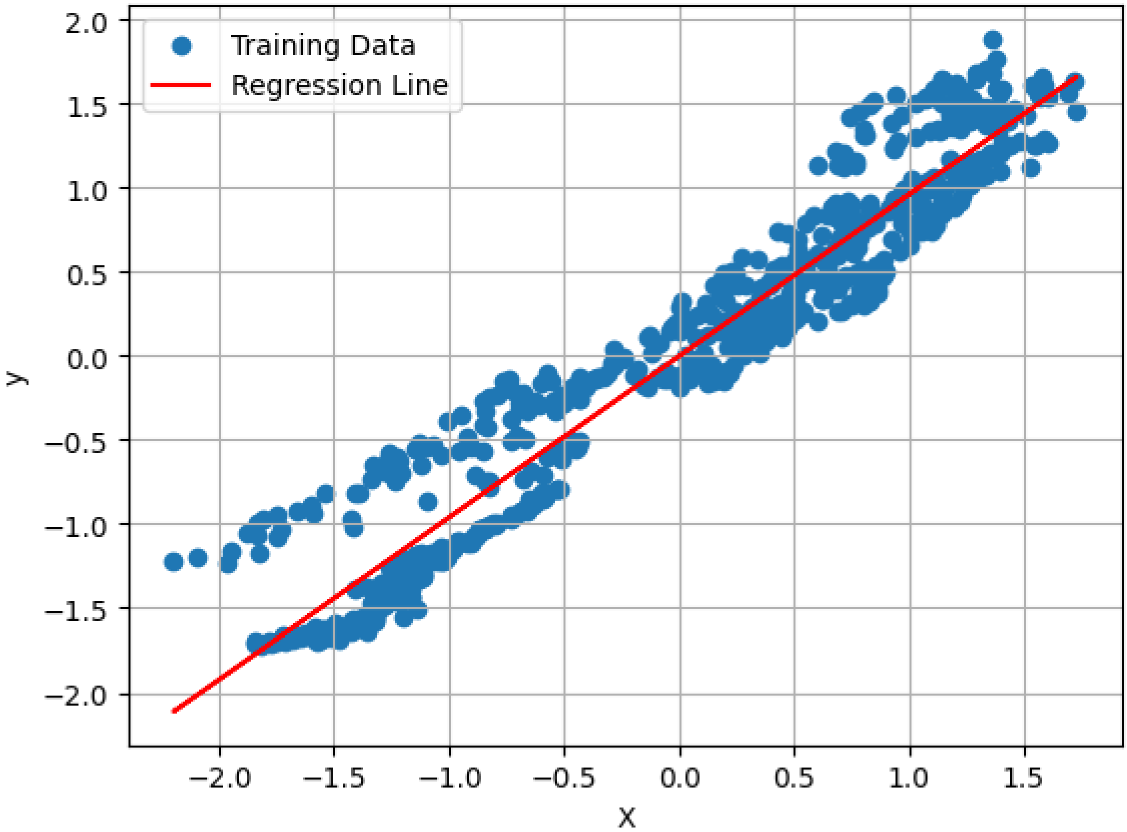
In [146…

```
# trained model to make predictions on the training data
y_pred = model.predict(X_train)
print(y_pred[:5])
```

```
[[ 1.03390274]
 [ 0.29766927]
 [ 0.65008484]
 [ 1.07290455]
 [-1.28412904]]
```

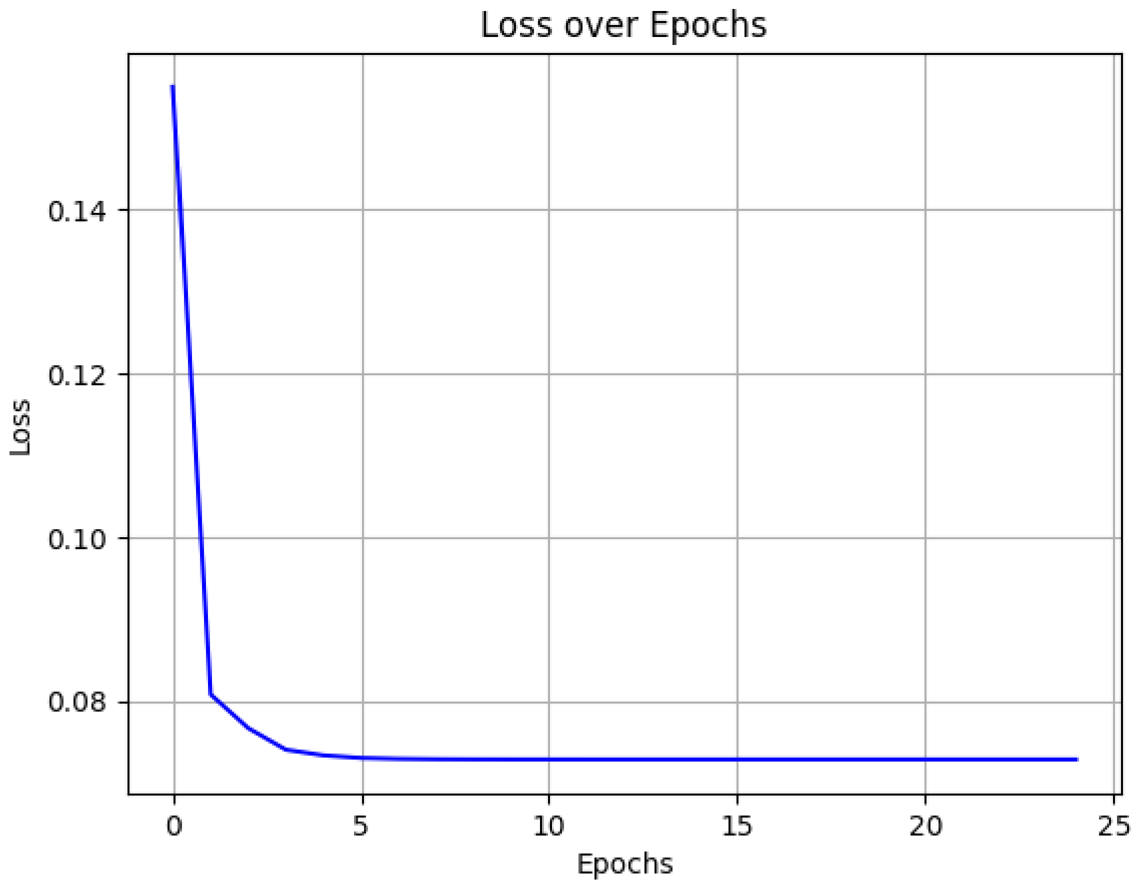
In [147…

```
#best fit line
plt.scatter(X_train, y_train, label="Training Data")
plt.plot(X_train, y_pred, color='red', label="Regression Line")
plt.xlabel("X")
plt.ylabel("y")
plt.grid(True)
plt.legend()
plt.show()
```



In [148…

```
# Plot the loss over training epochs
plt.plot(range(epochs), model.loss_history, linestyle='-', color='b')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss over Epochs')
plt.grid()
plt.show()
```



In [149…

```
y_pred = model.predict(X_train)
```

In [150…

```
r2_train = model.r_squared(y_train, y_pred)
print(f"R-squared (Training): {r2_train:.4f}")

# Predict on the validation data
y_pred_val = model.predict(X_val)

# Calculate R-squared for the validation data
r2_val = model.r_squared(y_val, y_pred_val)
print(f"R-squared (Validation): {r2_val:.4f}")
```

```
R-squared (Training): 0.9268
R-squared (Validation): 0.9380
```

REFERENCES:

Neural Networks: Forward Pass and Backpropagation URL: https://towardsdatascience.com/neural-networks-forward-pass-and-backpropagation-be3b75a1cfcc

Everything You Need to Know About Activation Functions in Deep Learning Models URL: https://towardsdatascience.com/everything-you-need-to-know-about-activation-functions-in-deep-learning-models-84ba9f82c253