Implement a linear regression model with a single neuron model

Linear regression is one of the simplest machine learning models used for predicting a continuous target variable based on one or more predictor variables. The simplest form of a linear regression model, which uses a single feature, can be visualized as a straight line fitted to a scatter plot of the data.

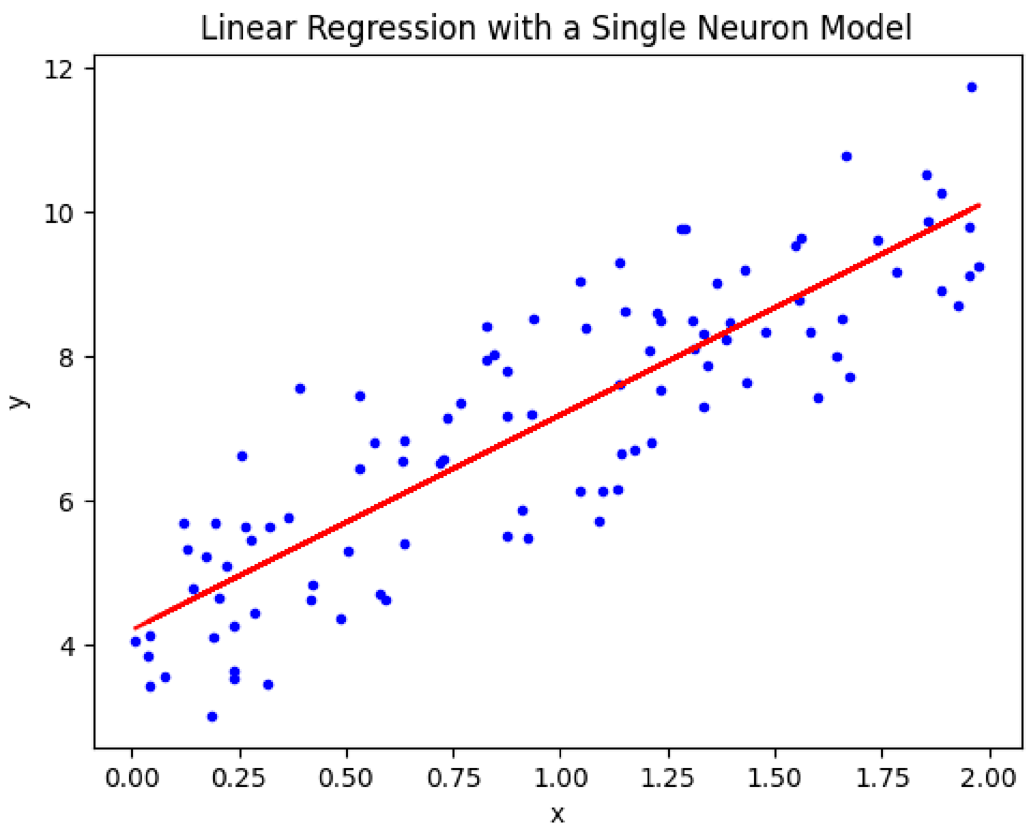
Single Neuron Model

A single neuron model, also known as a perceptron, can be used to implement linear regression. This model consists of the following components:

- **Inputs (features)**: The data points.
- **Weights**: Coefficients that determine the importance of each input.
- **Bias**: An additional parameter that allows the model to fit the data better.
- **Activation Function**: For linear regression, this is typically a linear function (i.e., no activation function).
-

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

# Generate some synthetic data
np.random.seed(0)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Function to perform linear regression using a single neuron model
def linear_regression(X, y, learning_rate=0.01, epochs=1000):
    m, n = X.shape
    X_b = np.c_[np.ones((m, 1)), X]  # add bias term (x0 = 1)
    theta = np.random.randn(n + 1, 1)  # initialize weights randomly

    for epoch in range(epochs):
        gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
        theta -= learning_rate * gradients
    return theta

# Train the model
theta = linear_regression(X, y)

# Plot the results
plt.plot(X, y, "b.")
plt.plot(X, theta[0] + theta[1] * X, "r-")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Linear Regression with a Single Neuron Model")
plt.show()

# Display the learned parameters
print(f"Intercept (bias): {theta[0][0]}")
print(f"Slope (weight): {theta[1][0]}")
```

Output:



```
Intercept (bias): 4.20607718142562
Slope (weight): 2.9827303563323175
```

Explanation of the Code

1. **Data Generation**: We generate some synthetic data for testing the model.
2. **Function Definition**: The `linear_regression` function performs linear regression using gradient descent.
    - `X_b` is the input matrix with an additional bias term.
    - `theta` is the parameter vector (weights and bias) initialized randomly.
    - In each epoch, we compute the gradients and update the parameters using the gradient descent algorithm.
3. **Training the Model**: We call the `linear_regression` function to train the model on the synthetic data.
4. **Plotting**: We plot the original data points and the fitted line to visualize the results.
5. **Parameter Display**: We print the learned parameters (intercept and slope).

The `np.random.seed(0)` function is used to set the seed for NumPy's random number generator. By setting a seed, you ensure that the sequence of random numbers generated is the same every time you run the code. This is essential for reproducibility, especially when debugging or comparing results across different runs.

How It Works

1. **Random Number Generation**: Random number generators (RNGs) in computers are actually deterministic algorithms that produce sequences of numbers that only appear random. These sequences are determined by an initial value called the "seed."
2. **Seeding the RNG**: By setting the seed with `np.random.seed(0)`, you initialize the RNG to a specific starting point. This means that every time you set the seed to the same value and then generate random numbers, you'll get the same sequence of numbers.
3. **Reproducibility**: This is particularly useful when you want to ensure that your experiments or simulations are reproducible. Other people (or you, at a later time) can run the same code and get the same results.

`y = 4 + 3 * X + np.random.randn(100, 1)`

1. **`4 + 3 * X`**:
   - This creates a **linear relationship between `X` and `y`**.
   - `4` is the intercept (bias term), and `3` is the slope (weight).
   - For each value in `X`, it computes the corresponding `y` value based on the equation y=4+3xy

X_b = np.c_[np.ones((m, 1)), X

`np.ones((m, 1))`:

- This creates an array of shape `(m, 1)` **filled with ones.**
- Here, `m` is the **number of samples** in `X`.
- Each element in this array represents the bias term (intercept) for each sample.

`np.c_[...]`:

- `np.c_` is a NumPy function that **concatenates arrays** along the second axis (columns).
- It is used to horizontally stack the arrays passed to it.

`np.c_[np.ones((m, 1)), X]`:

- This **concatenates the column of ones (bias terms) with the original feature matrix `X`.**
- The result is a new matrix `X_b` that has an additional column of ones at the beginning.