Simple Neural Network (Linear Regression) from Scratch

Overview

This project demonstrates how to build a simple neural network from scratch using only Python and NumPy. The model attempts to learn the linear relationship y = 2x from a small dataset using **gradient descent** to minimize the **mean squared error (MSE)**.

The code implements a single-layer perceptron that models a linear function of the form y = wx + b, where:

- x is the input,
- w is the weight,
- b is the bias,
- y is the predicted output.

The goal is to train the neuron to minimize the difference between predicted outputs and actual outputs from a given dataset. The dataset provided consists of input-output pairs representing a linear relationship (e.g., y = 2x). The neuron adjusts its parameters (w and b) using gradient descent to minimize the mean squared error (MSE) loss.

The code includes:

- A loss_function to compute the MSE.
- A train function to optimize the weight and bias.
- A main block to define the training data and execute the training.

Code

1. Imports

```
import numpy as np
```

2. Loss Function

```
def loss_function(w, b, train_data):
  result = 0
  count = len(train_data)
  for i in range(count):
    x = train_data[i][0]
    y = x*w + b
    d = y - train_data[i][1]
    result += d * d
  result /= count
  return result
```

The loss_function calculates the mean squared error (MSE) between the predicted outputs and the actual outputs for the training data.

Parameters

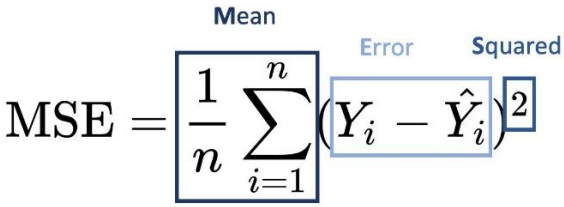
- w: The weight of the neuron (a scalar).
- b: The bias of the neuron (a scalar).
- train_data: A list of [x, y] pairs, where x is the input and y is the expected output.

Logic

1. Initialize result to store the sum of squared errors.
2. Get the number of data points (count).
3. For each data point:
    - Extract the input x (train_data[i][0]) and expected output y (train_data[i][1]).
    - Compute the predicted output: y_pred = x * w + b.
    - Calculate the error: d = y_pred - y.
    - Add the squared error (d * d) to result.
4. Compute the mean by dividing result by count.
5. Return the MSE.

Mathematics

The MSE is defined as:



where:

- $y\_pred_i = w * x_i + b$
- y_i is the actual output.
- n is the number of data points.

This measures how far the predictions are from the actual values, guiding the optimization process.

3. Training Function

```
def train(train_data):
  np.random.seed(0)
```

```
w = np.random.uniform(0, 10, 1)
b = np.random.uniform(0, 5, 1)

epsilon = 0.001
learning_rate = 0.01
print("loss: ", loss_function(w, b, train_data))

epochs = 200
for i in range(epochs):
    c = loss_function(w, b, train_data)
    cost_distance = (loss_function(w+epsilon, b, train_data) - c) / epsilon
    bias_distance = (loss_function(w, b+epsilon, train_data) - c) / epsilon

    w -= learning_rate * cost_distance
    b -= learning_rate * bias_distance
    if i % 10 == 0:
        print(loss_function(w, b, train_data))

print("------------------")
print(w)
print(b)
```

Purpose

The train function optimizes the weight (w) and bias (b) to minimize the loss function using gradient descent.

Parameters

- train_data: The dataset of input-output pairs.

Logic

1. **Initialization**:
    - Set a random seed (np.random.seed(0)) for reproducibility.
    - Initialize w with a random value between 0 and 10.
    - Initialize b with a random value between 0 and 5.
    - Both w and b are NumPy arrays of size 1 for consistency.

2. **Hyperparameters**:
    - epsilon = 0.001: A small value used to approximate gradients.
    - learning_rate = 0.01: Controls the step size of parameter updates.
    - epochs = 200: Number of training iterations.
3. **Initial Loss**:
    - Compute and print the initial loss using loss_function(w, b, train_data).
4. **Training Loop**:
    - For each epoch:
        - Compute the current loss (c): Calculate the mean squared error (c) using the current weight (w) and bias (b) by calling loss_function(w, b, train_data).
        - Approximate the gradient for the weight (w): Compute the change in loss when w is slightly increased by epsilon (0.001). The gradient is approximated as:
            - cost_distance = (loss_function(w+epsilon, b, train_data) - c) / epsilon
        - Approximate the gradient for the bias (b): Compute the change in loss when b is slightly increased by epsilon. The gradient is approximated as:
            - bias_distance = (loss_function(w, b+epsilon, train_data) - c) / epsilon
        - Update parameters using gradient descent: Adjust w and b in the direction that reduces the loss, scaled by the learning rate (0.01):
            - w -= learning_rate * cost_distance

              b -= learning_rate * bias_distance

            This moves w and b toward values that minimize the loss.
        - Every 10 epochs, print the current loss to monitor progress.
5. **Output**:
    - Print the final values of w and b.

Mathematics

The training process uses gradient descent to minimize the loss function. The gradient of the loss with respect to each parameter (w and b) shows how much the loss increases if that parameter is slightly increased. To reduce the loss, we move in the opposite direction of the gradient (i.e., subtract the gradient scaled by the learning rate). The gradient is approximated using a numerical method called finite difference:

For the weight w, the gradient is calculated as the difference in loss when w is increased by a small value (epsilon), divided by epsilon:

(loss_function(w + epsilon, b, train_data) - loss_function(w, b, train_data)) / epsilon

Similarly, for the bias b, the gradient is:

(loss_function(w, b + epsilon, train_data) - loss_function(w, b, train_data)) / epsilon

4. Main Block

```
if __name__ == "__main__":
  train_data = [[0,0],
                [1,2],
                [2,4],
                [4,8],
                [8,16]]
  train_example = train(train_data)
```

The main block defines the training dataset and initiates the training process.

Dataset

The train_data is a list of [x, y] pairs:

- [[0,0], [1,2], [2,4], [4,8], [8,16]]
- This represents a linear relationship where $y = 2x$. For example:
    - When $x = 1$, $y = 2$.
    - When $x = 2$, $y = 4$.

The train function is called with train_data, starting the optimization process. The variable train_example is assigned the return value of train, but since train does not return anything, train_example is None.

Expected Output

loss: [some_initial_value]

[loss every 10 epochs]

[final_w]

[final_b]

- The initial loss depends on the random w and b.
- The loss decreases over epochs as w approaches 2 and b approaches 0, reflecting the true relationship $y = 2x$.
- The final w should be close to 2, and b close to 0, indicating the neuron has learned the linear function.

Conclusion

This simple neural network implementation serves as an excellent introduction to ML concepts. It demonstrates how a single neuron can learn a linear relationship through iterative optimization. By understanding each component—loss function, gradient descent, and parameter updates—learners gain a solid foundation for exploring more complex neural networks.