Vanilla SGD Neural Network Implementation Comparison

Gradient Descent Algorithm Implementation

June 23, 2025

1 Overview

This document provides a comparative analysis of vanilla SGD (Stochastic Gradient Descent) implementations using NumPy and PyTorch. Both implementations train a simple neural network to fit a sine function.

2 Problem Definition

2.1 Dataset

- Input: $X \in \mathbb{R}^{100\times 1}$, randomly generated in $[0, 1]$ interval
- Target: $y = \sin(X) + \epsilon$, where $\epsilon \sim \mathcal{N}(0, 0.1^2)$
- Task: Use neural network to fit sine function

2.2 Network Architecture

- Input layer: 1 neuron
- Hidden layer: 1000 neurons (ReLU activation)
- Output layer: 1 neuron

3 Algorithm Implementation

3.1 Vanilla SGD Algorithm Flow

The core idea of vanilla SGD is to randomly select one sample for training in each iteration:

1. Randomly select a sample (x_i, y_i)
2. Forward propagation to compute prediction $\hat{y}_i$
3. Calculate loss function $L = \frac{1}{2}(\hat{y}_i - y_i)^2$
4. Backward propagation to compute gradient $\nabla_\theta L$
5. Update parameters: $\theta \leftarrow \theta - \alpha \nabla_\theta L$

3.2 NumPy Implementation Logic

3.2.1 Network Parameter Initialization

```
# Weight matrices
W1 = np.random.randn(input_size, hidden_size)  # (1, 1000)
W2 = np.random.randn(hidden_size, output_size) # (1000, 1)

# Bias vectors
b1 = np.zeros((input_size, hidden_size))       # (1, 1000)
b2 = np.zeros((input_size, output_size))       # (1, 1)
```

3.2.2 Forward Propagation

$$z_1 = X \cdot W_1 + b_1 \quad \text{(linear transformation)} \tag{1}$$
$$a_1 = \mathrm{ReLU}(z_1) \quad \text{(activation function)} \tag{2}$$
$$z_2 = a_1 \cdot W_2 + b_2 \quad \text{(output layer)} \tag{3}$$
$$\hat{y} = z_2 \tag{4}$$

3.2.3 Backward Propagation Gradient Calculation

$$\frac{\partial L}{\partial z_2} = \hat{y} - y \quad \text{(output layer gradient)} \tag{5}$$
$$\frac{\partial L}{\partial W_2} = a_1^T \cdot \frac{\partial L}{\partial z_2} \tag{6}$$
$$\frac{\partial L}{\partial b_2} = \sum \frac{\partial L}{\partial z_2} \tag{7}$$
$$\frac{\partial L}{\partial a_1} = \frac{\partial L}{\partial z_2} \cdot W_2^T \tag{8}$$
$$\frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial a_1} \odot (z_1 > 0) \quad \text{(ReLU gradient)} \tag{9}$$
$$\frac{\partial L}{\partial W_1} = x_i^T \cdot \frac{\partial L}{\partial z_1} \tag{10}$$
$$\frac{\partial L}{\partial b_1} = \sum \frac{\partial L}{\partial z_1} \tag{11}$$

Step-by-step explanation:

1. **Output layer gradient $\frac{\partial L}{\partial z_2} = \hat{y} - y$:**
   - Since $L = \frac{1}{2}(\hat{y} - y)^2$ and $\hat{y} = z_2$
   - $\frac{\partial L}{\partial z_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} = (\hat{y} - y) \cdot 1 = \hat{y} - y$
2. **Weight gradient $\frac{\partial L}{\partial W_2} = a_1^T \cdot \frac{\partial L}{\partial z_2}$:**
   - Since $z_2 = a_1 \cdot W_2 + b_2$
   - $\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2} = \frac{\partial L}{\partial z_2} \cdot a_1^T$

3. **Bias gradient** $\frac{\partial L}{\partial b_2} = \sum \frac{\partial L}{\partial z_2}$:
   - Since $z_2 = a_1 \cdot W_2 + b_2$
   - $\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial z_2} \cdot \frac{\partial z_2}{\partial b_2} = \frac{\partial L}{\partial z_2} \cdot 1$
   - For batch processing, we sum over all samples
4. **Activation gradient** $\frac{\partial L}{\partial a_1} = \frac{\partial L}{\partial z_2} \cdot W_2^T$:
   - This propagates the error back to the hidden layer
   - W_2^T is the transpose of the weight matrix
5. **Hidden layer gradient** $\frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial a_1} \odot (z_1 > 0)$:
   - Since $a_1 = \mathrm{ReLU}(z_1) = \max(0, z_1)$
   - $\frac{\partial \mathrm{ReLU}(z_1)}{\partial z_1} = \begin{cases} 1 & \text{if } z_1 > 0 \\ 0 & \text{if } z_1 \leq 0 \end{cases}$
   - $\odot$ represents element-wise multiplication
6. **Input weight gradient** $\frac{\partial L}{\partial W_1} = x_i^T \cdot \frac{\partial L}{\partial z_1}$:
   - Since $z_1 = x_i \cdot W_1 + b_1$
   - $\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial z_1} \cdot \frac{\partial z_1}{\partial W_1} = \frac{\partial L}{\partial z_1} \cdot x_i^T$
7. **Input bias gradient** $\frac{\partial L}{\partial b_1} = \sum \frac{\partial L}{\partial z_1}$:
   - Similar to output bias, but for the hidden layer

Key concepts:

- **Chain rule**: $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial x}$
- **ReLU derivative**: $\frac{d}{dx} \max(0, x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$
- **Matrix derivatives**: For $z = x \cdot W + b$, we have $\frac{\partial z}{\partial W} = x^T$ and $\frac{\partial z}{\partial x} = W^T$

3.3 PyTorch Implementation Logic

3.3.1 Network Definition

```
class SimpleNN(nn.Module):
    def __init__(self, input_size=1, hidden_size=1000, output_size=1):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

3.3.2 Training Loop

```
for iteration in range(n_iterations):
    # Randomly select sample
    random_index = torch.randint(0, m, (1,)).item()
    xi = X[random_index:random_index+1]
    yi = y[random_index:random_index+1]

    # Forward propagation
    y_hat = model(xi)
    loss = criterion(y_hat, yi)

    # Backward propagation
    model.zero_grad()
    loss.backward()

    # Manual parameter update (vanilla SGD)
    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
```

4 Key Differences Analysis

4.1 Implementation Comparison

Aspect	**NumPy Version**	**PyTorch Version**
Gradient computation	Manual implementation	Automatic differentiation
Parameter update	Explicit update of W1, W2, b1, b2	Iterate through model.parameters()
Memory management	Manual management	Automatic management
Code complexity	Higher	Lower
Debugging difficulty	More difficult	Easier

Table 1: Implementation Comparison

4.2 Performance Characteristics

4.2.1 NumPy Version Advantages

- Fully controllable parameter update process
- No framework dependency, lightweight
- Better for understanding underlying mathematical principles

4.2.2 PyTorch Version Advantages

- Automatic differentiation, reducing errors
- Modular design, concise code
- Easy to extend and debug

5 Training Process Analysis

5.1 Loss Function

Both versions use MSE loss function:

$$L = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \tag{12}$$

5.2 Some Settings

- Learning rate: $\alpha = 0.01$
- Training steps: 1000 iterations
- Output loss every 100 steps

6 Summary

Both implementations successfully demonstrate the core ideas of vanilla SGD algorithm:

- Randomly select samples for training
- Compute gradients and update parameters
- Gradually optimize network weights

The NumPy version is more suitable for learning algorithm principles, while the PyTorch version is better for practical application development.

Both implementations can effectively fit the sine function, validating the effectiveness of the vanilla SGD algorithm.