Feed Forward Neural Network from Scratch

Bhanu Prasanna Koppolu

Table of contents

Source Code

Full implementation available at DL_from_scratch/ffn.py

Feed Forward Neural Network

Starting with a feed forward neural network to predict MNIST dataset accurately. The MNIST consists of images with handwritten numbers from 0 - 9 = 10 classes. Each image is 28×28 size, so 28×28 = 784 pixels.

Network Architecture

I will be building the following network:

1. **Input Layer** - 784 neurons (flattened image)
2. **Hidden Layer 1** - 512 neurons
3. **Hidden Layer 2** - 256 neurons
4. **Hidden Layer 3** - 128 neurons
5. **Output Layer** - 10 neurons (Softmax for classification)

Activation Functions

ReLU (Rectified Linear Unit):

$$\text{ReLU}(x) = \max(0, x)$$

$$\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Softmax:

$$\text{Softmax}(x_i) = \frac{e^{x_i - \max(x)}}{\sum_j e^{x_j - \max(x)}}$$

Loss Function

Sparse Categorical Cross-Entropy:

$$L(y, \hat{y}) = -\log(\hat{y}_{y_{true}})$$

where $\hat{y}_{y_{true}}$ is the predicted probability for the true class.

Weight Initialization

Using **He initialization** for ReLU networks:

$$W \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{in}}}\right)$$

Implementation

```
def relu(x):
    """ReLU activation: max(0, x)"""
    return np.where(x > 0, x, 0)

def relu_backward(dout, x):
    """Gradient of ReLU: 1 if x > 0, else 0"""
    dz = dout.copy()
    dz[x <= 0] = 0
    return dz

def softmax(x):
    """Softmax with numerical stability"""
    exp_x = np.exp(x - np.max(x))
    return exp_x / exp_x.sum()

def sparse_categorical_crossentropy(y_true, y_pred):
    """Cross entropy loss for integer labels"""
    y_pred = np.clip(y_pred, 1e-15, 1 - 1e-15)
    return -np.log(y_pred[y_true])
```

```python
class FFN_MNIST:
    """Feed Forward Neural Network for MNIST classification.

    Architecture: 784 -> 512 -> 256 -> 128 -> 10
    Activations: ReLU for hidden layers, Softmax for output
    """

    def __init__(self, learning_rate=0.01):
        # He initialization for ReLU networks: std = sqrt(2/n_in)

        # First hidden layer: 784 -> 512
        self.w1 = np.random.randn(784, 512) * np.sqrt(2.0 / 784)
        self.b1 = np.zeros(512)

        # Second hidden layer: 512 -> 256
        self.w2 = np.random.randn(512, 256) * np.sqrt(2.0 / 512)
        self.b2 = np.zeros(256)

        # Third hidden layer: 256 -> 128
        self.w3 = np.random.randn(256, 128) * np.sqrt(2.0 / 256)
        self.b3 = np.zeros(128)

        # Output layer: 128 -> 10
        self.w4 = np.random.randn(128, 10) * np.sqrt(2.0 / 128)
        self.b4 = np.zeros(10)

    def fit(self, X, y, epochs=10, subset_size=5000):
        """Train the network using SGD.

        For each sample:
        1. Forward pass to compute predictions
        2. Compute loss
        3. Backward pass to compute gradients
        4. Update weights using gradient descent
        """
        ...

    def _forward(self, X):
        """Forward pass through all layers.

        Layer operations:
        Z1 = X @ W1 + b1 -> H1 = ReLU(Z1)
```

```
        Z2 = H1 @ W2 + b2 -> H2 = ReLU(Z2)
        Z3 = H2 @ W3 + b3 -> H3 = ReLU(Z3)
        Z4 = H3 @ W4 + b4 -> out = Softmax(Z4)
        """
        ...

    def _backward(self, X, y_true, y_pred, loss):
        """Backward pass using chain rule.

        Gradient flow:
        1. dZ4 = y_pred - y_true_onehot (softmax + cross-entropy combined)
        2. dW4 = H3.T @ dZ4, dH3 = dZ4 @ W4.T
        3. dZ3 = dH3 * ReLU'(Z3), then continue...

        Uses outer product for weight gradients when processing
        single samples.
        """
        ...

    def _update_grads(self):
        """Update weights using gradient descent.
        W = W - lr * dW
        """
        ...

    def predict(self, X):
        """Predict class labels for samples."""
        ...

    def evaluate(self, X, y):
        """Compute accuracy on dataset."""
        ...
```

Backpropagation Details

The backward pass computes gradients using the chain rule:

1. **Output layer gradient** (softmax + cross-entropy combined):

$$\frac{\partial L}{\partial Z_4} = \hat{y} - y_{onehot}$$

2. **Weight gradients** (using outer product for single samples):

$$\frac{\partial L}{\partial W_4} = H_3^T \cdot \frac{\partial L}{\partial Z_4}$$

3. **Hidden layer gradients** (propagating through ReLU):

$$\frac{\partial L}{\partial H_3} = \frac{\partial L}{\partial Z_4} \cdot W_4^T$$

$$\frac{\partial L}{\partial Z_3} = \frac{\partial L}{\partial H_3} \odot \text{ReLU}'(Z_3)$$

4. **Repeat for each layer** going backwards through the network.