Neural Network from Scratch

Your Name

1 Introduction

In this document I am going to go over my implementation of a very simple Neural Network and the math behind it.

2 Network Architecture

My simple neural network consists of:

1. **Input Layer**: 3-dimensional input
2. **Hidden Layer**: One neuron
3. **Output Layer**: 1-dimensional output

3 Initialization

We generate/initialize the weights using the Xavier Initialization:

$$W \sim N(0, \frac{2}{n_{in} + n_{out}}) \tag{1}$$

Where:

- n_{in} is the number of input features
- n_{out} is the number of neurons in the layer
- N is the normal distribution function

In our Python implementation:

```
n_in = 3
n_out = 1
w = np.random.normal(0, np.sqrt(2 / (n_in + n_out)), size=(n_in,
    n_out))
b = 0
```

4 Forward Propagation

In the Forward Propagation we pass the input data through the Neural Network to generate an output.

1. **Linear Combination**: Calculate the weighted sum of inputs plus a bias:

$$Z = W \cdot X + b \tag{2}$$

2. **Activation Function**: Apply a non-linear function, in this case Sigmoid, to introduce some non-linearity:

$$A = \frac{1}{1 + e^{-Z}} \tag{3}$$

In our code:

```
z = np.dot(w.T, x) + b
a = sigmoid(z)
```

Where `sigmoid` is defined as:

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

5 Loss Function

We use the Mean Squared Error (MSE) to calculate the loss of this epoch and evaluate how the model is doing:

$$L = \frac{1}{2}(A - Y)^2 \tag{4}$$

Where:

- A is the predicted output
- Y is the actual target value

6 Backpropagation

In the Backpropagation process we calculate the gradients of the loss function with respect to the weights and the biases to then be able to update them accordingly.

6.1 Derivatives

We compute the following partial derivatives:

6.1.1 Derivative of Loss with respect to Activation

We start with the Mean Squared Error loss function:

$$L = \frac{1}{2}(A - Y)^2 \tag{5}$$

To find $\frac{\partial L}{\partial A}$, we apply the chain rule:

$$\begin{aligned}
\frac{\partial L}{\partial A} &= \frac{\partial}{\partial A}[\frac{1}{2}(A - Y)^2] && (6)\\
&= \frac{1}{2} \cdot 2(A - Y) \cdot \frac{\partial}{\partial A}(A - Y) && (7)\\
&= (A - Y) \cdot 1 && (8)\\
&= A - Y && (9)
\end{aligned}$$

Thus we derive: $\frac{\partial L}{\partial A} = (A - Y)$

6.1.2 Derivative of Activation with respect to Linear Combination

We use the sigmoid function as our activation function:

$$A = \frac{1}{1 + e^{-Z}} = \sigma(Z) \tag{10}$$

To find $\frac{\partial A}{\partial Z}$, we use the derivative of the sigmoid function:

$$\begin{aligned}
\frac{\partial A}{\partial Z} &= \frac{\partial}{\partial Z}\sigma(Z) && (11)\\
&= \sigma(Z)(1 - \sigma(Z)) && (12)\\
&= A(1 - A) && (13)
\end{aligned}$$

$A = \sigma(Z)$ Thus, we derive: $\frac{\partial A}{\partial Z} = A(1 - A)$

6.1.3 Derivative of Linear Combination with respect to Weights

Our linear combination is:

$$Z = W \cdot X + b \tag{14}$$

To find $\frac{\partial Z}{\partial W}$, we differentiate with respect to W:

$$\begin{aligned}
\frac{\partial Z}{\partial W} &= \frac{\partial}{\partial W}(W \cdot X + b) && (15)\\
&= \frac{\partial}{\partial W}(W \cdot X) + \frac{\partial}{\partial W}b && (16)\\
&= X + 0 && (17)\\
&= X && (18)
\end{aligned}$$

Thus, we derive: $\frac{\partial Z}{\partial W} = X$

6.1.4 Derivative of Linear Combination with respect to Bias

$$Z = W \cdot X + b \tag{19}$$

To find $\frac{\partial Z}{\partial b}$, we differentiate with respect to b:

$$\begin{aligned}
\frac{\partial Z}{\partial b} &= \frac{\partial}{\partial b}(W \cdot X + b) && (20)\\
&= \frac{\partial}{\partial b}(W \cdot X) + \frac{\partial}{\partial b} b && (21)\\
&= 0 + 1 && (22)\\
&= 1 && (23)
\end{aligned}$$

Thus, we derive: $\frac{\partial Z}{\partial b} = 1$

6.2 Chain Rule

We use the chain rule to calculate the gradients of the loss with respect to the weights and bias:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial A} \cdot \frac{\partial A}{\partial Z} \cdot \frac{\partial Z}{\partial W} = (A - Y) \cdot A(1 - A) \cdot X \tag{24}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial A} \cdot \frac{\partial A}{\partial Z} \cdot \frac{\partial Z}{\partial b} = (A - Y) \cdot A(1 - A) \tag{25}$$

In our code:

```
dl_dw = (a - y) * a * (1 - a) * x
dl_db = (a - y) * a * (1 - a)
```

7 Gradient Descent

In the last step we apply an algorithm called Gradient Descent to find the direction for which the weights and biases will be minimal. Afterwards we update them to minimize the loss.

$$W = W - \eta \cdot \frac{\partial L}{\partial W} \tag{26}$$

$$b = b - \eta \cdot \frac{\partial L}{\partial b} \tag{27}$$

Where η is the learning rate, which controls the step size of each update.
In our implementation:

```
learning_rate = 0.005
w = w - learning_rate * dl_dw
b = b - learning_rate * dl_db
```

8 Training Loop

In the full code this process gets repeated multiple times to hopefully minimize the loss and get as accurate as possible. In the full code it looks like this:

```
def feedforward_nn(epochs, x, y):
    # ... (initialization code)

    for epoch in range(epochs):
        # Forward propagation
        z = np.dot(w.T, x) + b
        a = sigmoid(z)

        # Backpropagation
        dl_dw = (a - y) * a * (1 - a) * x
        dl_db = (a - y) * a * (1 - a)

        # Gradient descent
        w = w - learning_rate * dl_dw
        b = b - learning_rate * dl_db

        # Print progress (optional)
        if epoch % 100 == 0:
            print(f"Epoch {epoch}: w = {list(map('{:.3f}%'.format,
    w.flatten()))}, b = {list(map('{:.3f}%'.format,b))}, loss = {
    list(map('{:.3f}%'.format,(a - y) ** 2))}")

    return w, b
```

Hope you enjoyed my implementation:)