A Step-by-Step Implementation of Gradient Descent and Backpropagation

One example of building a neural network from scratch

Yitong Ren

May 30 · 4 min read

The original intention behind this post was merely me brushing upon mathematics in neural network, as I like to be well versed in the inner workings of algorithms and get to the essence of things. I then think I might as well put together a story rather than just revisiting the formulas on my notepad over and over. Though you might find a number of tutorials for building a simple neural network from scratch. Different people have varied angles of seeing things as well as the emphasis of study. Another way of thinking might in some sense enhance understanding. So let's dive in.

Photo from Unsplash

Neural network in a nutshell

The core of neural network is a big function that maps some input to the desired target value, in the intermediate step does the operation to produce the network, which is by multiplying weights and add bias in a pipeline scenario that does this over and over again. The process of training a neural network is to determine a set of parameters that minimize the difference between expected value and model output. This is done using gradient descent (aka backpropagation), which by definition comprises two steps: calculating gradients of the loss/error function, then updating existing parameters in response to the gradients, which is how the descent is done. This cycle is repeated until reaching the minima of the loss function. This learning process can be described by the simple equation: $W(t+1) = W(t) — dJ(W)/dW(t)$.

The mathematical intuition

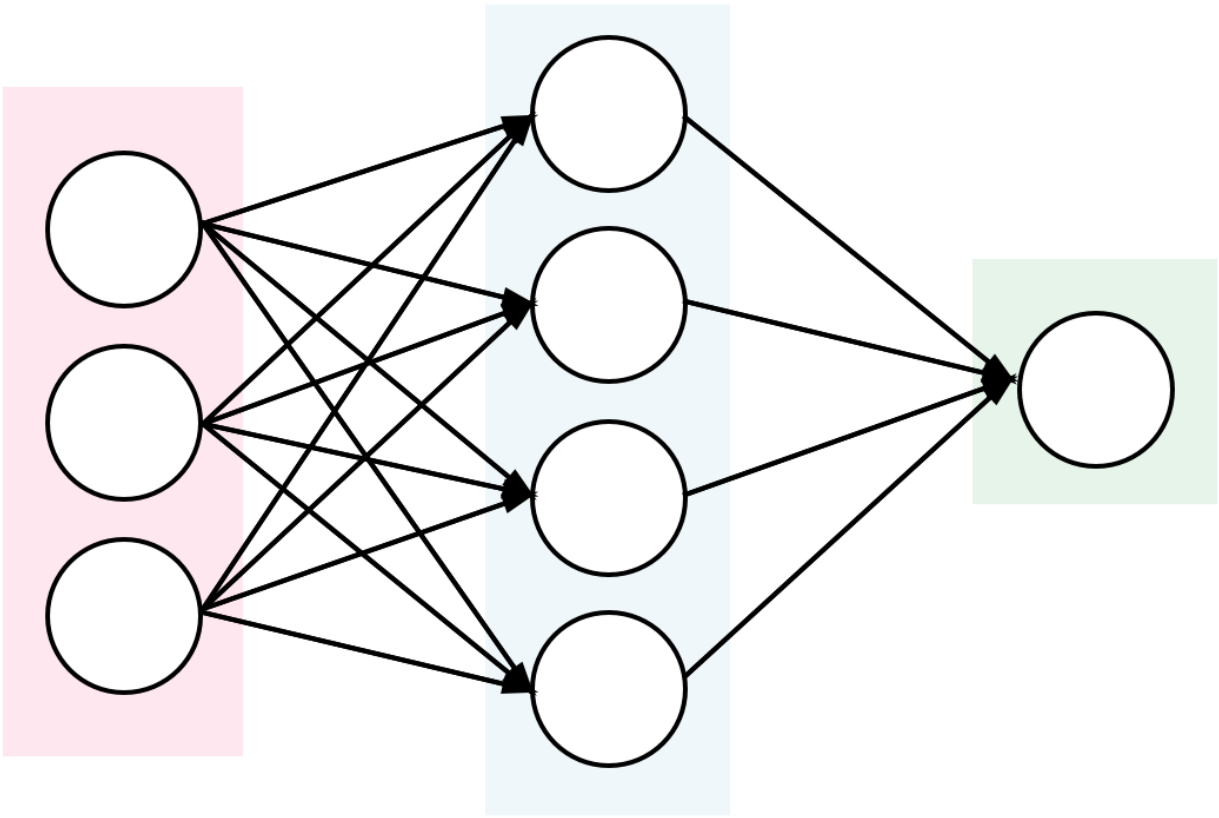


Photo from https://blog.goodaudience.com/artificial-neural-networks-explained-436fcf36e75

For my own practice purpose, I like to use a small network with a single hidden layer as in the diagram. In this layout, X represents input, subscripts i, j, k denote the number of units in the input, hidden and output layers respectively; w_ij represents the weights connecting input to hidden layer, and w_jk is the weights connecting hidden to output layer.

The model output calculation, in this case, would be:

$$\hat{y} = \delta(w_{jk} \cdot \underbrace{\delta(\underbrace{w_{ij} x_i + b_j}_{z_j})}_{x_j} + b_k)$$

z_k

Often the choice of the loss function is the sum of squared error. Here I use sigmoid activation function and assume bias b is 0 for simplicity, meaning weights are the only variables that affect model output. Let's derive the formula for calculating gradients of hidden to output weights w_jk.

$$\frac{\partial J(w)}{\partial w_{jk}} = \frac{\partial \frac{1}{2} \sum_{k \in K} (y - \hat{y})^2}{\partial w_{jk}}$$

$$= (y - \hat{y}) \frac{\partial (y - \hat{y})}{\partial w_{jk}}$$

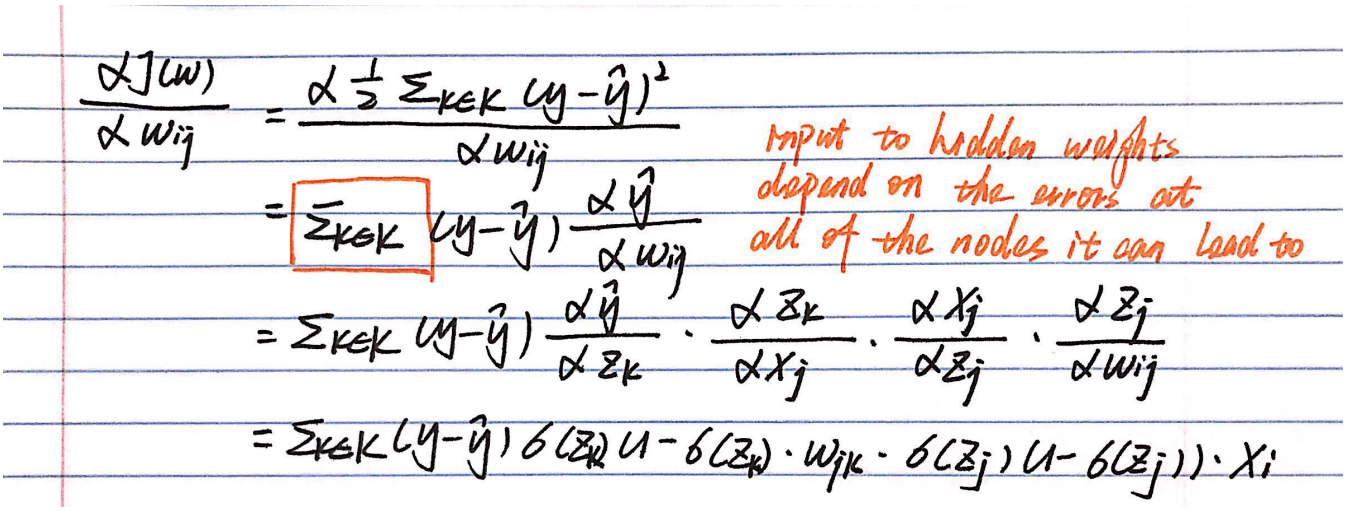
$$= (y - \hat{y}) \frac{\partial \hat{y}}{\partial w_{jk}}$$

$$= (y - \hat{y}) \boxed{\frac{\partial \hat{y}}{\partial z_k} \cdot \frac{\partial z_k}{\partial w_{jk}}} \rightarrow \text{chain rule}$$

$$= (y - \hat{y}) \boxed{\delta(z_k)(1 - \delta(z_k))} \cdot x_j$$

derivative of sigmoid function $\delta(z)$

The complexity of determining input to hidden weights is that it affects output error indirectly. Each hidden unit output affects model output, thus input to hidden weights w_ij depend on the errors at all of the units it is connected to. The derivation starts the same, just to expand the chain rule at z_k to the subfunction.



More thoughts:

Notice that the gradients of the two weights have a similar form. The error is backpropagated via the derivative of activation function, then weighted by the input (the activation output) from the previous layer. In the second formula, the backpropagated error from the output layer is further projected onto w_jk, then repeat the same way of backpropagation and weighted by the input. This backpropagating process is iterated all the way back to the very first layer in an arbitrary-layer neural network. ***"The gradients with respect to each parameter are thus considered to be the contribution of the parameter to the error and should be negated during learning."***

Putting the above process into code:

```
import numpy as np

class NeuralNetwork:
    def __init__(self):
        np.random.seed(10) # for generating the same results
        self.wij   = np.random.rand(3,4) # input to hidden layer weights
        self.wjk   = np.random.rand(4,1) # hidden layer to output weights

    def sigmoid(self, x, w):
        z = np.dot(x, w)
        return 1/(1 + np.exp(-z))

    def sigmoid_derivative(self, x, w):
        return self.sigmoid(x, w) * (1 - self.sigmoid(x, w))
```

```
    def gradient_descent(self, x, y, iterations):
        for i in range(iterations):
            Xi = x
            Xj = self.sigmoid(Xi, self.wij)
            yhat = self.sigmoid(Xj, self.wjk)
            # gradients for hidden to output weights
            g_wjk = np.dot(Xj.T, (y - yhat) * self.sigmoid_derivative(Xj, self.wjk))
            # gradients for input to hidden weights
            g_wij = np.dot(Xi.T, np.dot((y - yhat) * self.sigmoid_derivative(Xj, self.wjk),
                                        self.wjk.T) * self.sigmoid_derivative(Xi, self.wij))
            # update weights
            self.wij += g_wij
            self.wjk += g_wjk
        print('The final prediction from neural network are: ')
        print(yhat)
```

Below is the complete example:

```
import numpy as np

class NeuralNetwork:
    def __init__(self):
        np.random.seed(10) # for generating the same results
        self.wij   = np.random.rand(3,4) # input to hidden layer
weights
        self.wjk   = np.random.rand(4,1) # hidden layer to output
weights

    def sigmoid(self, x, w):
        z = np.dot(x, w)
        return 1/(1 + np.exp(-z))

    def sigmoid_derivative(self, x, w):
        return self.sigmoid(x, w) * (1 - self.sigmoid(x, w))

    def gradient_descent(self, x, y, iterations):
        for i in range(iterations):
            Xi = x
            Xj = self.sigmoid(Xi, self.wij)
            yhat = self.sigmoid(Xj, self.wjk)
            # gradients for hidden to output weights
            g_wjk = np.dot(Xj.T, (y - yhat) *
self.sigmoid_derivative(Xj, self.wjk))
            # gradients for input to hidden weights
            g_wij = np.dot(Xi.T, np.dot((y - yhat) *
self.sigmoid_derivative(Xj, self.wjk), self.wjk.T) *
self.sigmoid_derivative(Xi, self.wij))
            # update weights
            self.wij += g_wij
            self.wjk += g_wjk
        print('The final prediction from neural network are: ')
        print(yhat)

if __name__ == '__main__':
    neural_network = NeuralNetwork()
    print('Random starting input to hidden weights: ')
    print(neural_network.wij)
```

```
print('Random starting hidden to output weights: ')
print(neural_network.wjk)
X = np.array([[0, 0, 1], [1, 1, 1], [1, 0, 1], [0, 1, 1]])
y = np.array([[0, 1, 1, 0]]).T
neural_network.gradient_descent(X, y, 10000)
```

References:

1. https://theclevermachine.wordpress.com/2014/09/06/derivation-error-backpropagation-gradient-descent-for-neural-networks/
2. https://towardsdatascience.com/how-to-build-your-own-neural-network-from-scratch-in-python-68998a08e4f6