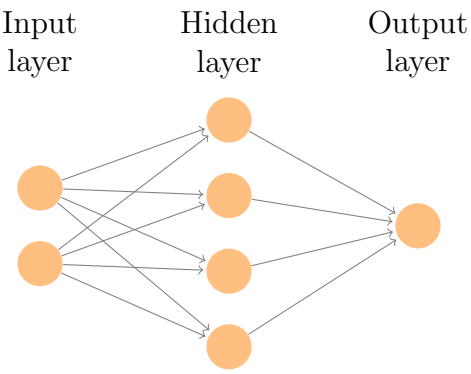
Week 05: Basic Neural Networks – Backpropogation

1 Introduction

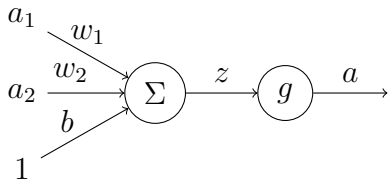
The main motivation for this lesson plan is for you to understand the main proponent to a neural network, defined as **backpropogation**. By understanding the intuition behind backpropogation, you will be able to develop, design, and debug neural networks. On a high level, backpropogation is a way of computing the gradients of expressions using the chain rule. In the following sections, we recap some basics of neural networks, chain rule, and matrix calculus.

1.1 Neural Network Model

A neural network consists of several layers, where the first layer is called the input layer, the last one is called the output layer, and the ones in the middle are called hidden layers. The figure below depicts a neural network consisting of three layers.



Each layer contains several artificial neurons. An artificial neuron is a mathematical function that takes a linear combination of its inputs, adds a bias, and then applies an activation function to the sum. Below, you can see the computational graph of an artificial neuron with two inputs and activation function g.



z is the linear combination of inputs plus the bias, i.e. $z = w_1 a_1 + w_2 a_2 + b$ and the output is given by $a = g(z)$.
We use the following notations throughout this note. We number the layers from 1 to L, with L corresponding to the output layer. We denote by w^l_{ji} the weight of connection between i^{th} neuron in layer $l-1$ and the j^{th} neuron in layer l, w^l_j the vector of weights corresponding to the j^{th} neuron in layer l, and w^l the matrix whose j^{th} row is w^l_j. We also denote by b^l_j the bias of j^{th} neuron in layer l, and b^l the vector of all the biases in layer l. Also, we let z^l_j and a^l_j be the output of the j^{th} neuron in layer l before and after applying the activation function g, respectively. That is $a^l_j = g(z^l_j)$ and $z^l_j = \sum_i w^l_{ji} a^{l-1}_i + b^l_j$. Furthermore, we use z^l and a^l to be the vector of z^l_j and a^l_j, respectively.

1.2 Forward Pass

Forward pass is a procedure to compute the output of a neural network given the weights, biases, and the inputs. We start from the inputs to a neural network and compute the outputs of neurons, layer by layer, till we finally compute neural net's output. By the notations discussed in section 1.1, it is straightforward to see that

$$a^l = g(w^l a^{l-1} + b^l).$$

The equation above gives us a fast matrix-based algorithm for the forward pass. Before each step of backpropagation, which we discuss in section 2, we need to have a forward pass in order to update the outputs of neurons used in the backpropagation algorithm.

1.3 Chain Rule

Chain rule is used to compute the derivative of a composite function. If f and g are two differentiable function, the derivative of $f(g(x))$ is given by

$$\frac{d}{dx} f(g(x)) = g'(x) f'(g(x)).$$

An alternative way to write the chain rule is by using Leibniz's notation. If variable x affects variable z only through variable y, then

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dz}.$$

In the multivariable case, suppose that z is a function of n variables $y_1, \ldots, y_n$, and that each of these variables are in turn functions of m variables $x_1, \ldots, x_m$. Then, the partial derivative of z with respect to any x_i, for $i = 1, \ldots, m$, is given by

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}.$$

For example, Let's compute $\frac{\partial f}{\partial \theta}$, where $f = x^2 + y^2$, $x = r\cos\theta$, and $y = r\sin\theta$. We have

$$\begin{aligned}\frac{\partial f}{\partial \theta} &= \frac{\partial f}{\partial x}\frac{\partial x}{\partial \theta} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial \theta} \\ &= (2x).(-r\sin\theta) + (2y).(r\cos\theta) \\ &= (2r\cos\theta).(-r\sin\theta) + (2r\sin\theta).(r\cos\theta) \\ &= 0.\end{aligned}$$

1.4 The Hadamard Product

The elementwise multiplication of two matrices is called the Hadamard product. If $A = [a_{ij}]_{m\times n}$ and $B = [b_{ij}]_{m\times n}$ are two m by n matrices, then their Hadamard product, shown as $A \circ B$, is given by

$$A \circ B = [a_{ij}b_{ij}]_{m\times n}.$$

2 Backpropagation

Backpropagation is the essence of neural network training, which was originally introduced in 1970s. This algorithm is used to compute the partial derivatives of the error function with respect to the weights and biases.

2.1 The Intuition Behind Backpropagation

To see how this algorithm works, let's consider a simple example. Assume that we want to compute the partial derivative of $f = (x + y)z$ with respect to x. The effect of x on f is only through $t = x + y$. Therefore, using the chain rule, one can first compute the partial derivative of f with respect to t, and then multiply it by the partial derivative of t with respect to x, i.e.

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial t}\frac{\partial t}{\partial x} = z.$$

This basic procedure is what backpropagation is built upon. To find the partial derivative of the error function with respect to one of the weights, say w, in a neural network, we first need to find the partial derivative of the error with respect to the output of the neuron connected to w, and then multiply it by the partial derivative of that specific neuron's output with respect to w. Furthermore, in the exact same manner, the partial derivative of the error with respect to one of the neurons' output could be computed using the partial derivatives of the

error with respect to the outputs of neurons in the next layer. Consequently, we may start from the output layer and go backward to compute the corresponding partial derivatives. In the next section, you will be walked through the mathematical details of backpropagation.

2.2 Implementing Backpropagation

In backpropagation, we want to see how changes in weights and biases affect the error function. Indeed, we want to find $\frac{\partial e}{\partial w_{ji}^l}$ and $\frac{\partial e}{b_j^l}$. Note that w_{ji}^l and b_j^l affect e only through z_j^l, and hence it is helpful to define $\delta_j^l = \frac{\partial e}{\partial z_j^l}$. Let's denote by δ^l the vector of such partial derivatives in layer l. Now, we derive four equations that backpropagtion algorithm relies on.

First, we start from the output layer to compute δ^L. Note that since this is the last layer, we can compute the partial derivative of e with respect to each neuron's output, a_j^L, directly and use the chain rule to get our first equation:

$$\delta_j^L = \frac{\partial e}{\partial a_j^L} g'(z_j^L). \tag{1}$$

Notice that (1) depends on the choice of error function e, so we may not be able to further simply this expression unless we are given a specific error function. We can also write (1) in a matrix-based form as follows:

$$\delta^L = \nabla_{a^L} e \circ g'(z^L).$$

Second, we derive δ^l provided we have access to δ^{l+1}. Note that

$$\begin{aligned}
\delta_j^l &= \frac{\partial e}{\partial z_j^l} \\
&= \sum_k \frac{\partial e}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \\
&= \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l} \\
&= \sum_k \delta_k^{l+1} \frac{\partial (\sum_s w_{ks}^{l+1} a_s^l + b_k^{l+1})}{\partial z_j^l} \\
&= \sum_k \delta_k^{l+1} \frac{\partial (w_{kj}^{l+1} a_j^l)}{\partial z_j^l} \\
&= \sum_k \delta_k^{l+1} w_{kj}^{l+1} \frac{\partial a_j^l}{\partial z_j^l} \\
&= \sum_k \delta_k^{l+1} w_{kj}^{l+1} g'(z_j^l).
\end{aligned}$$

Thus, we have found that

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} g'(z_j^l), \tag{2}$$

which could also be written in a matrix-based form as follows:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \circ g'(z^l).$$

By the first two equations, we can compute all δ^l. We just need to see how to derive the partial derivatives with respect to weights and biases using δ^l.
Note that $z_j^l = \sum_i w_{ji}^l a_i^{l-1} + b_j^l$ implying that $\frac{\partial z_j^l}{\partial b_j^l} = 1$. Therefore, we have

$$\frac{\partial e}{\partial b_j^l} = \frac{\partial e}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \frac{\partial e}{\partial z_j^l} = \delta_j^l. \tag{3}$$

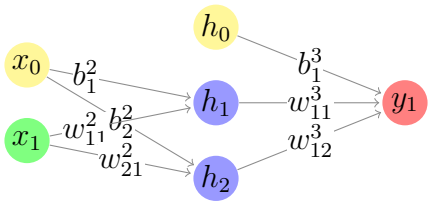
To compute $\frac{\partial e}{\partial w_{ji}^l}$, note that $\frac{\partial z_j^l}{\partial w_{ji}^l} = a_i^{l-1}$. Hence,

$$\frac{\partial e}{\partial w_{ji}^l} = \frac{\partial e}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{ji}^l} = a_i^{l-1} \frac{\partial e}{\partial z_j^l} = a_i^{l-1} \delta_j^l. \tag{4}$$

Equations (1), (2), (3), and (4) are the ones that backpropagation algorithm uses to compute the partial derivatives of the error function with respect to the weights and biases.

2.3 Backpropagation In A Small Neural Network

In this section, we implement backpropagation on a small neural network with just one hidden layer.



The input layer has only one neuron x_1, and the neuron x_0 is for the bias, so it always outputs 1. Similarly, there are only two neurons in the hidden layer and h_0 is the bias for the neuron y_1 in the output layer.
Assume that the error function is given by the squared error. That is, if the output of the neural net is $\hat{y}$ and the true output is y, then $e = (y - \hat{y})^2$. Given a dataset $\{(x_1, y_1), \ldots, (x_n, y_n)\}$,

the incurred error would be $e = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$. We would like to find the partial derivatives of e with respect to the weights and biases of the network. Since the error function is the some of errors incurred at each data point, we can just restrict our attention to one data point, compute the corresponding partial derivatives, and then sum it over all the data points.

Suppose that the input is x and we have initialized the weights and biases randomly. Using the forward pass, we first compute the output of each neuron (this part is skipped here). Let $\hat{y}$ be the output of the network, so $e = (y - \hat{y})^2$. We then have

$$\begin{aligned}\delta^3 &= \nabla_{a^3} e \circ g'(z^3) \\ &= \nabla_{\hat{y}} e \circ g'(z^3) \\ &= 2(\hat{y} - y)g'(z^3).\end{aligned}$$

Therefore,

$$\begin{aligned}\delta^2 &= ((w^3)^T \delta^3) \circ g'(z^2) \\ &= (\begin{bmatrix} w_{11}^3 \\ w_{12}^3 \end{bmatrix} \delta^3) \circ \begin{bmatrix} g'(z_1^2) \\ g'(z_2^2) \end{bmatrix} \\ &= \begin{bmatrix} \delta^3 w_{11}^3 g'(z_1^2) \\ \delta^3 w_{12}^3 g'(z_2^2) \end{bmatrix}.\end{aligned}$$

Now that we have computed δ for each layer (we do not need it for the input layer), it is easy to get the partial derivatives with respect to the weights and the biases. Using (3) and (4), we have the following:

$$\frac{\partial e}{\partial b_1^3} = \delta^3 = 2(\hat{y} - y)g'(z^3),$$

$$\frac{\partial e}{\partial b_1^2} = \delta_1^2 = \delta^3 w_{11}^3 g'(z_1^2) = 2(\hat{y} - y)w_{11}^3 g'(z^3)g'(z_1^2),$$

$$\frac{\partial e}{\partial b_2^2} = \delta_2^2 = \delta^3 w_{12}^3 g'(z_2^2) = 2(\hat{y} - y)w_{12}^3 g'(z^3)g'(z_2^2),$$

$$\frac{\partial e}{\partial w_{11}^3} = a_1^2 \delta^3 = 2(\hat{y} - y)a_1^2 g'(z^3),$$

$$\frac{\partial e}{\partial w_{12}^3} = a_2^2 \delta^3 = 2(\hat{y} - y)a_2^2 g'(z^3),$$

$$\frac{\partial e}{\partial w_{11}^2} = a_1^1 \delta_1^2 = 2(\hat{y} - y)w_{11}^3 g'(z^3)g'(z_1^2)x,$$

$$\frac{\partial e}{\partial w_{21}^2} = a_1^1 \delta_2^2 = 2(\hat{y} - y)w_{12}^3 g'(z^3)g'(z_2^2)x.$$

2.4 What Are The Benefits of Backpropagation?

There are some other alternatives for computing the gradient such as finite differences, but they are far more computationally expensive than backpropagation, specifically when it comes to neural nets with millions of neurons. In finite differences, in order to compute the partial derivative of the error function with respect to a particular weight, say w_{ji}^l, we fix all the other weights and biases, perturb w_{ij}^l and make one forward pass. Consequently, we need to make many forward passes depending on the number of weights and biases of the network making this procedure very computationally expensive, whereas in the backpropagation algorithm, we just need one forward and one backward pass.

2.4.1 Exploding Gradients

Finite differences is not accurate, as we use an approximation for partial derivatives instead of their exact value. In deep neural networks, this error in gradient computation can accumulate during an update and result in very large gradients. These in turn result in large updates to the network weights, and in turn, an unstable network. At an extreme, the values of weights can become so large as to overflow and result in NaN values. This phenomenon is called the exploding gradients. By using backpropagation instead of finite differences, one can compute the exact value for partial derivatives, which results in less computational error in gradients. This reduction in the error helps preventing the exploding gradients.

3 Conclusion

In backpropagation, we start at the end of the network, backpropagate or feed the errors back, recursively apply chain rule to compute gradients all the way to the inputs of the network and then update the weights. This provides us with a cheap way to exactly compute the gradient of the error function. Next week, you will get familiar with PyTorch and TensorFlow, packages that can speed up constructing neural networks and implementing backpropagation.

References

[1] https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd

[2] http://neuralnetworksanddeeplearning.com/chap2.html

[3] http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture4.pdf