Deriving the Backpropagation Equations

Jayanth Koushik
Carnegie Mellon University
jkoushik@cs.cmu.edu

1 Introduction

An intuitive understanding of neural networks and backpropagation can be gained by considering artificial neural networks (ANNs) to be real-valued circuits of simple gates. In the forward phase, each gate receives some inputs, performs an operation, and generates an output. In the backward phase, the gates receive the gradient with respect to their output, and propagate it back to their inputs. This propagation takes the simple form of a product of the local gradient and the received gradient. In this tutorial we will use this view to derive the backpropagation equations for a fully connected neural network. The final goal is to get expressions for the gradient of the cost function with respect to the weights and biases of a neural network.

2 Notation

w^l_{jk} represents the weight from unit k of level $l-1$ to unit j of level l. $w^l_{\cdot k}$ (note the dot before k) represents the vector of all weights to level l from unit k of level $l-1$. b^j_l is the bias of unit j of level l. The activation function of the hidden units is represented by σ. Note that we do not make any assumptions about the form of the activation function (sigmoid, tanh etc.). It is however assumed that we can compute its gradient. a^l_j is the activation of unit j of level l. And finally, z^l_j is the biased weighted sum of inputs i.e. $z^l_j = \sum_k w^l_{jk} a^{l-1}_k + b^l_j$. Clearly, $a^l_j = \sigma(z^l_j)$. We do not specify the number of units in the hidden layers. Sums over quantities in a layer are over all units in the layer.

3 Building the circuit

The neural network in consideration is a fully connected network with L layers. Let us start by designing the circuit that represents this network. We will focus on a particular unit j of level l. Each unit just computes a weighted sum of its inputs, adds a bias and applies the activation function. $a^l_j = \sigma(\sum_k w^l_{jk} a^{l-1}_k + b^l_j)$. The atomic operations here are just multiplication, addition, and application of the activation function; so we can represent a unit using gates corresponding to

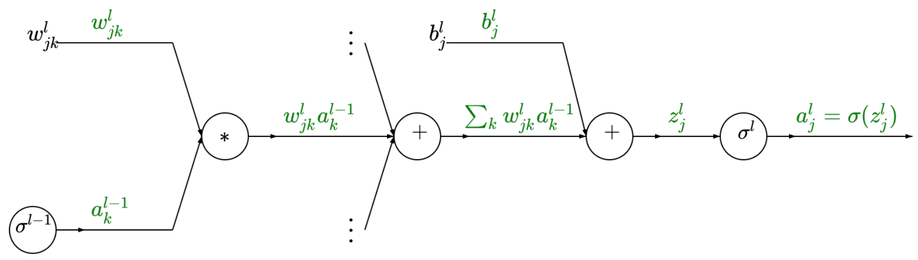

Figure 1: Circuit representing a single unit in the neural network. Gate outputs are shown above the connecting "wires" in green. The activation gates are superscripted with their corresponding levels so they can be referred to unambiguously.

these operations as shown if Figure 1. Let us analyze this circuit. The whole circuit corresponds to a single perceptron unit in the conventional model of a neural network. The multiply gate multiplies a single activation from the previous level with a corresponding weight. All such weighted activations are summed up by the first add gate. The second add gate biases this weighted sum to produce z^l_j. Finally the activation gate applies the activation function on this to produce a^l_j which will act as input to the next level.

4 Recursive computation of gradient

Since gates behave locally, ignorant of everything they are not connected to, we can compute gradients across a chain through backpropagation, provided we know the gradient with respect to the final output. However, in the circuit we just constructed, we don't know any gradients to use as a starting point and we seem to be stuck in a loop. Let us break this loop arbitrarily by giving a name to the gradient with respect to z^l_j: δ^l_j. If the final cost for the neural network circuit is C, then $\delta^l_j = {}^{\partial C}/_{\partial z^l_j}$. We will eventually need to find a way of computing δ^l_j, but for now let us use it to compute the other gradients.

For any f, z, x, y, with $z = x + y$, we have from the chain rule

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial z}\frac{\partial z}{\partial x} = \frac{\partial f}{\partial z} \qquad\qquad \frac{\partial f}{\partial y} = \frac{\partial f}{\partial z}\frac{\partial z}{\partial y} = \frac{\partial f}{\partial z}$$

This shows that the add gate simply distributes the gradient of its output to all its inputs. So δ^l_j passes unchanged to b^l_j, $w^l_{jk}a^{l-1}_k$ and we have ${}^{\partial C}/_{\partial b^l_j} = \delta^l_j$, ${}^{\partial C}/_{\partial w^l_{jk}a^{l-1}_k} = \delta^l_j$. Similarly for f, z, x, y, with $z = xy$, we have

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial z}\frac{\partial z}{\partial x} = y\frac{\partial f}{\partial z} \qquad\qquad \frac{\partial f}{\partial y} = \frac{\partial f}{\partial z}\frac{\partial z}{\partial y} = x\frac{\partial f}{\partial z}$$

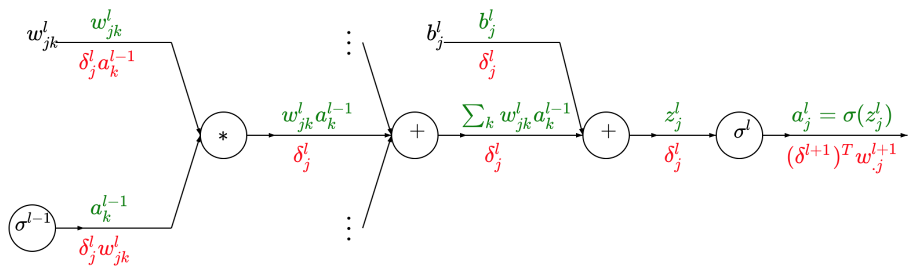

Figure 2: Gradients (shown below the wires in red) corresponding to a unit computed in terms of δ^l_j. For σ^{l-1}, only the gradient received from the shown multiply gate is indicated.

So the multiply gate switches its inputs and multiplies them with the output gradient. It sends $\delta^l_j a^{l-1}_k$ back to w^l_{jk} which gives us $\partial C/\partial w^l_{jk} = \delta^l_j a^{l-1}_k$. To the σ^{l-1} gate it sends back $\delta^l_j w^l_{jk}$. Now the activation gate output is connected to all the multiply gates of the next level. It receives gradients (as we've just calculated) from all the multiply gates and adds them up giving $\partial C/\partial a^{l-1}_k = \sum_j \delta^l_j w^l_{jk}$. This can be written concisely in the form of a dot product: $(\delta^l)^{\mathrm{T}} w^l_{.k}$ where δ^l is a vector of all the δ^l_js. By the same logic, we can compute the gradient with respect to a^l_j as $(\delta^{l+1})^{\mathrm{T}} w^{l+1}_{.j}$. These values are shown in Figure 2. Looking at the circuit, we can see that δ^l_j can be expressed using the gradient with respect to a^l_j. First for any f, z, g, x with $z = g(x)$ we have

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial z}\frac{\partial z}{\partial x} = g'(x)\frac{\partial f}{\partial z}$$

From this we have that the σ^l gate multiplies its local gradient $\sigma'(z^l_j)$ with the gradient of its output $(\delta^{l+1})^{\mathrm{T}} w^{l+1}_{.j}$ and passes it backwards. So $\delta^l_j = \sigma'(z^l_j)(\delta^{l+1})^{\mathrm{T}} w^{l+1}_{.j}$. Note that this depends only on values from the next level in the network allowing us to compute δ^l_j recursively, and based on it, the other gradients corresponding to a unit.

5 Recursion base case

To complete our backpropagation algorithm, we need to solve the base case of the recursion. First, we need to find an expression for δ^L_j. We will stick to the convention of not applying the activation function on the final layer. So the output from the neural network is the vector of z^L_js which can be used to compute a cost C (like cross-entropy or multiclass SVM loss). Since $\delta^L_j = \partial C/\partial z^L_j$, this is readily computed as the form of C is known. Using the δ^L_js as the starting point, all the δ^l_js can be computed recursively.

The gradient with respect to the weights, $\delta_j^l a_k^{l-1}$ depends on the activation from the previous level. So we need to consider the special case when $l = 1$ i.e. the first hidden layer. However, this “issue” is remedied quite simply by adopting the convention that a^0 corresponds to the input vector x of the neural network. The correctness follows readily from the local nature of the gates: it does not matter if the inputs are constants or outputs of other gates.

6 Conclusion

We have derived the gradient of a neural network cost function with respect to its weights and biases. The gradient takes the form of recursive equations which can be evaluated by a forward and backward pass over the network. The equations are summarized here.

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} \tag{1}$$

$$\delta_j^l = \sigma'(z_j^l)(\delta^{l+1})^{\mathrm{T}} w_{\cdot j}^{l+1} \text{ for } l = 1 \ldots L-1 \tag{2}$$

$$\frac{\partial C}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1} \text{ for } l = 1 \ldots L \tag{3}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \text{ for } l = 1 \ldots L \tag{4}$$

References

[1] Andrej Karpathy. *Hacker’s guide to Neural Networks*. URL: http://karpathy.github.io/neuralnets/.

[2] Michael Nielsen. *Neural Networks and Deep Learning*. URL: http://neuralnetworksanddeeplearning.com/index.html.