Feedforward Backpropagation

Karim Boustany

Abstract

We provide an elementary derivation of the backpropagation equations for feedforward neural networks.

1 Feedforward Networks

We define the class of neural networks in which we will be interested. We begin with a (usually fixed) $n \times (p+1)$ input matrix X, whose first column we assume to be comprised entirely of ones. A **feedforward neural network** with L **layers** is given by $Y = X^L + \epsilon$, with $\epsilon \sim N(0, \sigma^2 I)$ and X^l defined recursively through

$$X^0 = X, \qquad X^l = a^l(X^{l-1}W^l), \qquad l = 1, \ldots, L.$$

Here, for $l = 1, \ldots, L$, the constituent elements are given by:

1. An $n \times d_L$ matrix Y of independent random variables.
2. An **activation function** $a^l \colon \mathbb{R} \to \mathbb{R}$ which is applied to each individual entry of a matrix.
3. A $p_l \times d_l$ matrix of **weights** W^l, where we require $p_1 = p+1$ and $p_l = d_{l-1}$ for $l = 2, \ldots, L$ in order for matrix multiplication to be well defined at each layer.

The entries of the various matrices and vectors above are interchangeably called **units**, **nodes** or **neurons** of the network. The matrix $X = X^0$ is called the **input layer** of the neural network, and the matrix X^L is called the **output layer**. The intermediate layers X^l for $l = 1, \ldots, L-1$ are called **hidden layers**. When $L = 1$, the network is called **shallow**, and is called **deep** otherwise.

2 Backpropagation

As in linear regression, given a set of observations (Y, X) to which we would like to fit our neural network, the first step is to learn the parameters of such a network, a process which is also referred to as **training**. This proceeds in the usual way of initializing guesses for the matrices $W^1, \ldots, W^L$, and then

updating these guesses via some version of gradient descent. In this case, we are trying to minimize a chosen **cost function**

$$C(Y, \widehat{Y}) = C(Y, \widehat{Y}(W^1, \ldots, W^L)),$$

which we will require to satisfy

$$C(Y, \widehat{Y}) = \frac{1}{n} \sum_{i=1}^{n} C_i(Y_i, \widehat{Y}_i),$$

so that techniques like stochastic gradient descent can be applied. Observe that we are implicitly assuming the ith component C_i depends only on the corresponding rows of Y and $\widehat{Y}$. One common example of a cost function is the **mean square error**, which is given by

$$C(Y, \widehat{Y}) = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \widehat{Y}_i)^2.$$

Here we assume $d_L = 1$, so that Y and $\widehat{Y}$ are simply n-dimensional column vectors. A common example where this is not the case is the **softmax cross entropy**, whose ith term is given by

$$C_i(Y_i, \widehat{Y}_i) = -\sum_{j=1}^{d_L} Y_{ij} \log(P_{ij}),$$

and where we have set

$$P_{ij} = \frac{\exp(\widehat{Y}_{ij})}{\sum_{k=1}^{d_L} \exp(\widehat{Y}_{ik})}.$$

Because of the high complexity of deep neural networks, implementing gradient descent requires a computationally expensive number of differentiations using the chain rule. However, there is an algorithm called **backpropagation** which allows these computations to be carried out with minimal redundancy. Before describing it, we will fix some notational conventions.

2.1 Conventions

When f is a scalar-valued function of the entries a_{ij} of a matrix A, we define the **gradient** of f with respect to A to be the matrix $\nabla_A f$ whose (i, j)th entry is $\partial_{a_{ij}} f$. In particular, the matrices $\nabla_A f$ and A must always have the same dimensions.

We introduce the symbol $Z^l = X^{l-1} W^l$ for $l = 1, \ldots, L$, so that by definition we have $X^l = a^l(Z^l)$. Because the function a^l already has a superscript, we will denote its derivative by $\dot{a}^l$, and when we apply this derivative to a matrix, it is understood that we are applying it to each entry of that matrix. Furthermore, when computing derivatives of the cost function C with respect to any one of

its variables, it is sufficient to do so for each individual component C_i. However, this component only depends on Y_i and $\widehat{Y}_i$, which in turn only depend on the ith row of each of the matrices X^l and Z^l for $l = 1, \ldots, L$, owing to how matrix multiplication is defined. Consequently, when computing the derivative of C_i with respect to w^l_{jk}, the (j,k)th entry of W^l, we will only ever need to consider the ith rows of the matrices X^l in the course of applying the chain rule.

Because of this, we will now fix the ith component component C_i and drop its subscript. We will also fix the ith rows of $\widehat{Y}$, X^l and Z^l, again dropping their subscripts and denoting them by the lowercase letters $\hat{y}$, x^l and z^l. We then use subscripts to denote *entries* of these rows. Thus, when we write something like x^l_i, we are referring to the ith scalar entry of the d_l-dimensional row vector x^l. We will also make use of the **Kronecker delta** symbol, which is defined as

$$\delta_{ij} = \begin{cases} 1, & \text{if } i = j, \\ 0, & \text{if } i \neq j. \end{cases}$$

Finally, in what follows we will denote ordinary matrix multiplication by $*$ and entrywise (Hadamard) multiplication by $\circ$.

2.2 Computations

By the chain rule, we have that

$$\partial_{w^l_{jk}} C = \sum_{i=1}^{d_L} \partial_{\hat{y}_i} C \cdot \partial_{w^l_{jk}} \hat{y}_i.$$

It will be convenient to introduce the function f^l_i, which is the composition of the activation function a^l with all subsequent layers of the neural network until $\hat{y}_i = x^L_i$. This allows us to write

$$\hat{y}_i(W^l) = f^l_i(z^l_1(W^l), \ldots, z^l_{d_l}(W^l)),$$

and the chain rule yields

$$\partial_{w^l_{jk}} \hat{y}_i = \sum_{r=1}^{d_l} \partial_{z^l_r} f^l_i \cdot \partial_{w^l_{jk}} z^l_r.$$

But now observe that for $r = 1, \ldots, d_l$ we have

$$z^l_r(W^l) = \sum_{s=1}^{d_{l-1}} x^{l-1}_s \cdot w^l_{sr},$$

so that $\partial_{w^l_{jk}} z^l_r = x^{l-1}_j \cdot \delta_{kr}$. This in turn implies that

$$\partial_{w^l_{jk}} \hat{y}_i = \partial_{z^l_k} f^l_i \cdot x^{l-1}_j.$$

Inserting this derivative into our original formula gives

$$\partial_{w^l_{jk}} C = \sum_{i=1}^{d_L} \partial_{\hat{y}_i} C \cdot \partial_{z^l_k} f^l_i \cdot x^{l-1}_j.$$

If we define the row vector δ^l_i to have kth entry equal to $\partial_{z^l_k} f^l_i$, then it is easy to check that the above translates in matrix notation to

$$\nabla_{W^l} C = \sum_{i=1}^{d_L} (\partial_{\hat{y}_i} C \cdot (x^{l-1})^T * \delta^l_i).$$

We are therefore reduced to determining the vectors δ^l_i for $i = 1, \ldots, d_L$ and $l = 1, \ldots, L$. It turns out that this can be done using a backwards recursion, from which the backpropagation algorithm derives its name.

By the very definition of the function f^l_i, we have the relation

$$f^l_i(z^l) = \hat{y}_i(x^l_1(z^l), \ldots, x^l_{d_l}(z^l)),$$

and taking derivatives, this gives us

$$\begin{aligned} \partial_{z^l_j} f^l_i &= \sum_{k=1}^{d_L} \partial_{x^l_k} \hat{y}_i \cdot \partial_{z^l_j} x^l_k \\ &= \sum_{k=1}^{d_L} \partial_{x^l_k} \hat{y}_i \cdot \dot{a}^l(z^l_j) \cdot \delta_{jk} \\ &= \partial_{x^l_j} \hat{y}_i \cdot \dot{a}^l(z^l_j). \end{aligned}$$

In terms of matrix operations, this says that δ^l_i is given by

$$\delta^l_i = \nabla_{x^l} \hat{y}_i \circ \dot{a}^l(z^l).$$

In particular, the row vector δ^L_i has all entries equal to zero except the ith entry, which is $\dot{a}^L(z^L_i)$. But now consider the relation

$$\hat{y}_i(x^l_1, \ldots, x^l_{d_{l-1}}) = f^l_i(z^l_1(x^{l-1}), \ldots, z^l_{d_l}(x^{l-1})),$$

and take derivatives to get

$$\partial_{x^{l-1}_j} \hat{y}_i = \sum_{k=1}^{d_l} \partial_{z^l_k} f^l_i \cdot w^l_{jk}.$$

In terms of matrix operations, this is equivalent to

$$\nabla_{x^{l-1}} \hat{y}_i = \delta^l_i * (W^l)^T.$$

Applying the entrywise product with $\dot{a}^{l-1}(z^{l-1})$ to both sides of this equation and using our previous equation for δ^l_i finally gives us

$$\delta^{l-1}_i = (\delta^l_i * (W^l)^T) \circ \dot{a}^{l-1}(z^{l-1}).$$

We have now derived the following **backpropagation equations**:

$$\nabla_{W^l} C = \sum_{i=1}^{d_L} (\partial_{\hat{y}_i} C \cdot (x^{l-1})^T * \delta_i^l), \qquad \delta^{l-1} = (\delta^l * (W^l)^T) \circ \dot{a}^{l-1}(z^{l-1}).$$

Here we are denoting by δ^l the matrix whose ith row is δ_i^l. Let us examine the equation for $\nabla_{W^l} C$ in the special cases of the mean squared error and softmax cross entropy. In the first case, we have $d_L = 1$, and a straightforward calculation yields

$$\nabla_{W^l} C = -2(y - \hat{y}) \cdot (x^{l-1})^T * \delta^l.$$

For the softmax cross entropy, the calculation is a little more involved, and ultimately yields

$$\nabla_{W^l} C = \sum_{i=1}^{d_L} ((p_i - y_i) \cdot (x^{l-1})^T * \delta_i^l),$$

where we recall that

$$p_i = \frac{\exp(\hat{y}_i)}{\sum_{j=1}^{d_L} \exp(\hat{y}_j)}.$$

However, the reader is cautioned *not* to deduce from this formula that

$$\partial_{\hat{y}_i} C = p_i - y_i.$$

Indeed, the formula above falls out of the specific computations involved in differentiating the softmax cross entropy, and not from naive substitution.

3 Summary of Backpropagation

Up until now, we have been using C to denote only a single component of the full cost function. To obtain the gradients for the total cost function, one uses the backpropagation equations for each component and then averages the resulting component gradients. We will assume this to be understood going forward, and now revert to using the letter C for the full cost function in the following description of the complete backpropagation algorithm:

1. Initialize guesses for the weights $W^1, \ldots, W^l$.
2. Randomly shuffle the observations in the given training set.
3. Use the given parameters to compute X^l and Z^l for $l = 1, \ldots, L$. This is called the **forward pass**.
4. Using the backpropagation equations, recursively compute the matrices δ^l, starting with
$$\text{diag}(\dot{a}^L(z_1^L), \ldots, \dot{a}^L(z_{d_L}^L)),$$
then use these to compute the gradients $\nabla_{W^l} C$ for $l = 1, \ldots, L$. This is called the **backward pass**.

5. Use the previously computed gradients to perform gradient descent and update the weights $W^1, \ldots, W^L$.

6. Repeat steps 2 through 5 until the cost function C is as small as desired.