**CSE 446: Machine Learning** **Lecture**

# Multi-layer Perceptrons & the Back-propagation Algorithm

*Instructor: Sham Kakade*

## 1 Terminology

- non-linear decision boundaries and the XOR function
- multi-layer neural networks & multi-layer perceptrons
- # of layers (definitions sometimes not consistent)
- input layer is $x$. output layer is $y$. hidden layers.
- activation function or transfer function or link function.
- forward propagation
- back propagation

Issues related to training are:

- non-convexity
- initialization
- weight symmetries and "symmetry breaking"
- saddle points & local optima & global optima
- vanishing gradients

## 2 Backprop (for MLPs)

### 2.1 MLPs

We can specify an $L$-hidden layer network as follows: given outputs $\{z_j^{(l)}\}$ from layer $l-1$, the *input* activations are:

$$a_j^{(l)} = \left( \sum_{i=1}^{d^{(l-1)}} w_{ji}^{(l)} z_i^{(l-1)} \right) + w_{j0}^{(l)}$$

where $w_{j0}^{(l)}$ is a "bias" term. For ease of exposition, we drop the bias term and proceed by assuming that:

$$a_j^{(l)} = \sum_{i=1}^{d^{(l-1)}} w_{ji}^{(l)} z_i^{(l-1)} \, .$$

The *output* activation of each node is:

$$z_j^{(l)} = h(a_j^{(l)})$$

**Remark:** The terminology of the "activation" is not necessarily used consistently. Sometimes the activation is used to refer to only the input $a$'s , as defined above, or only the input $z$'s, as defined above . Sometimes the literature uses the terminology input activation and output activation. We will use the latter terminology as it less confusing. Or we just use the terminology "inputs" and "outputs" of nodes.

The target function/output, after we go through $L$-hidden layers, is then:

$$\widehat{y}(x) = a^{(L+1)} = \sum_{i=1}^{d^{(L)}} w_i^{(L+1)} z_i^{(L)},$$

where saying the output is the activation at level $L + 1$. If we have more than one output than,

$$\widehat{y}_j(x) = a_j^{(L+1)} = \sum_{i=1}^{d^{(L)}} w_{ji}^{(L+1)} z_i^{(L)},$$

for $j \in \{1, \dots K\}$, if we had $K$ outputs (e.g. if we had $K$ classes). It is straightforward to generalize this to force $\widehat{y}(x)$ to be bounded between 0 and 1 (using a sigmoid transfer function) or having multiple outputs. Let us also use the convention that:

$$z_i^{(0)} = x[i]$$

The parameters of the model are all the weights

$$w^{(L+1)}, w^{(L)}, \dots w^{(1)} .$$

## 2.2 The Back-Propagation Algorithm for MLPs

The back-propagation algorithm was introduced in [1].

In general, the loss function in an $L$-hidden layer network is:

$$L(w^{(1)}, w^{(2)}, \dots, w^{(L+1)}) = \frac{1}{N} \sum_n \ell(y, \widehat{y}(x))$$

and for the special case of the square loss we have:

$$\frac{1}{N} \sum_n \ell(y_n, \widehat{y}(x_n)) = \frac{1}{2} \frac{1}{N} \sum_n (y_n - \widehat{y}(x_n))^2$$

Again, we seek to compute:

$$\nabla \ell(y_n, \widehat{y}(x_n))$$

where the gradient is with respect to all the parameters.

## The Forward Pass

Starting with the input $x$, go forward (from the input to the output layer), compute and store in memory the variables $a^{(1)}, z^{(1)}, a^{(2)}, z^{(2)}, \dots a^{(L)}, z^{(L)}, a^{(L+1)}$

## The Backward Pass

Note $\ell(y, \widehat{y})$ depends on all the parameters and we will not write out this functional dependency (e.g. $\widehat{y}$ depends on $x$ and all the weights).

We will compute the derivates by recursion. It useful to do recursion by computing the derivates with respect to the input activations and proceeding "backwards" (from the output layer to the input layer). Define:

$$\delta_j^{(l)} := \frac{\partial \ell(y, \widehat{y})}{\partial a_j^{(l)}}$$

First, let us see that if we had all the $\delta_j^{(l)}$'s then we are able to obtain the derivatives with respect to all of our parameters:

$$\frac{\partial \ell(y, \widehat{y})}{\partial w_{ji}^{(l)}} = \frac{\partial \ell(y, \widehat{y})}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} z_i^{(l-1)}$$

where we have used the chain rule and that $\frac{\partial a_j^{(l)}}{\partial w_{ji}^{(l)}} = z_i^{(l-1)}$. To see the latter claim is true, note

$$a_j^{(l)} = \sum_{c=1}^{d^{(l-1)}} w_{jc}^{(l)} z_c^{(l-1)}$$

(the sum is over all nodes $c$ in layer $l-1$). This expression implies $\frac{\partial \partial a_j^{(l)}}{\partial w_{ji}^{(l)}} = z_i^{(l-1)}$

Now let us understand how to start the recursion, i.e. to compute the $\delta$'s for the output layer, if there is only one node and $\widehat{y} = a^{(L+1)}$,

$$\delta^{(L+1)} = \frac{\partial \ell(y, \widehat{y})}{\partial a^{(L+1)}} = -(y - a^{(L+1)}) = -(y - \widehat{y})$$

(so we don't need a subscript of $j$ since there is only one node). Hence, for the output layer,

$$\frac{\partial \ell(y, \widehat{y})}{\partial w_j^{(L+1)}} = \delta^{(L+1)} z_j^{(L)} = -(y - a^{(L+1)}) z_j^{(L)}$$

where we have used our expression for $\delta^{(L+1)}$.

Thus, we know how to start our recursion. Now let us proceed recursively, computing $\delta_j^{(l)}$ using $\delta_j^{(l+1)}$. Observe that all the functional dependencies on the activations at layer $l$ goes through the activations at $l+1$. This implies, using the chain rule,

$$\begin{aligned}
\delta_j^{(l)} &= \frac{\partial \ell(y, \widehat{y})}{\partial a_j^{(l)}} \\
&= \sum_{k=1}^{d^{(l+1)}} \frac{\partial \ell(y, \widehat{y})}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial a_j^{(l)}} \\
&= \sum_{k=1}^{d^{(l+1)}} \delta_k^{(l+1)} \frac{\partial a_k^{(l+1)}}{\partial a_j^{(l)}}
\end{aligned}$$

To complete the recursion we need to evaluate $\frac{\partial a_k^{(l+1)}}{\partial a_j^{(l)}}$. By definition,

$$a_k^{(l+1)} = \sum_{c=1}^{d^{(l)}} w_{kc}^{(l+1)} z_c^{(l)} = \sum_{c=1}^{d^{(l)}} w_{kc}^{(l+1)} h(a_c^{(l)})$$

This implies:

$$\frac{\partial a_k^{(l+1)}}{\partial a_j^{(l)}} = w_{kj}^{(l+1)} h'(a_j^{(l)}) \,,$$

and, by substitution,

$$\delta_j^{(l)} = h'(a_j^{(l)}) \sum_{k=1}^{d^{(l+1)}} w_{kj}^{(l+1)} \delta_k^{(l+1)}$$

which completes our recursion.

## 2.3 The Algorithm

We are now ready to state the algorithm.

**The Forward Pass:**

1. Starting with the input $x$, go forward (from the input to the output layer), compute and store in memory the variables $a^{(1)}, z^{(1)}, a^{(2)}, z^{(2)}, \ldots a^{(L)}, z^{(L)}, a^{(L+1)}$

**The Backward Pass:**

1. Initialize as follows:
$$\delta^{(L+1)} = -(y - \widehat{y}) = -(y - a^{(L+1)})$$
and compute the derivatives at the output layer:
$$\frac{\partial \ell(y, \widehat{y})}{\partial w_j^{(L+1)}} = -(y - \widehat{y}) z_j^{(L)}$$
2. Recursively, compute
$$\delta_j^{(l)} = h'(a_j^{(l)}) \sum_{k=1}^{d^{(l+1)}} w_{kj}^{(l+1)} \delta_k^{(l+1)}$$
and also compute our derivatives at layer $l$:
$$\frac{\partial \ell(y, \widehat{y})}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} z_i^{(l-1)}$$

# 3 Time Complexity

Assume that addition, subtraction, multiplication, and division take "one unit" of time (and let us ignore numerical precision issues).

Let us say the time complexity to compute $\ell(y, \widehat{y}(x))$ is based on the naive algorithm (which explicitly just computes all the sums in the forward pass).

**Theorem 3.1.** *Suppose that we can compute the derivative $h'(\cdot)$ in an amount of time that is within a constant factor of the time it takes to compute $h(\cdot)$ itself (and suppose that constant is $5$). Using the backprop algorithm, the (total) time to compute* both *$\ell(y, \widehat{y}(x))$ and $\nabla \ell(y, \widehat{y}(x))$ — note that $\nabla \ell(y, \widehat{y}(x))$ contains # parameter partial derivatives — is within a factor of $5$ of computing just the scalar value $\ell(y, \widehat{y}(x))$.*

*Proof.* The number of transfer function evaluations in the forward pass is just the number of nodes. In the backward pass, the number of times we need to evaluate the derivative of the transfer function is also just the number of nodes. In the forward pass, note that the total computation time is a constant factor (actually $2$) times the number weights. To see this, note that to compute any activation, it costs us (one addition and one multiplication) associated with each weight (and that the every weight is only involved in the computation of just one activation). Similarly, by examining the recursion in the backward pass, we see that the total compute time to get all the $\delta$'s is a constant factor times the number of weights (again there is only one multiplication and one addition associated with each weight). Also, the compute time for obtaining the partial derivatives from the $\delta$'s is equal to the number of weights. $\square$

**Remark:** That we defined the time complexity to be under the "naive" algorithm is irrelevant. If we had a faster algorithm to compute $\ell(y, \widehat{y}(x))$ (say through a faster matrix multiplication algorithm or possibly other tricks), then the above theorem would still hold. This is the remarkable Baur-Strassen theorem [2] (also independently stated by [3]).

## References

[1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In David E. Rumelhart, James L. McClelland, and PDP Research Group, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1*, pages 318–362. MIT Press, Cambridge, MA, USA, 1986.

[2] Walter Baur and Volker Strassen. The complexity of partial derivatives. *Theoretical Computer Science*, 22:317–330, 1983.

[3] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2008.