Neural Networks
Lecture Notes
Multi-layer networks

Marcel van Gerven

1 Learning Goals

After studying the lecture material you should:

1. be able to explain why backprop learning was a major breakthrough in NN research
2. be able to reproduce the universal approximation theorem
3. know how to compute the output of a multi-layer perceptron
4. know how to compute the derivative of the sigmoid activation function
5. understand what the delta terms are and how they are important in backprop learning
6. know what the weight update looks like when you include momentum

2 Notes

2.1 Multilayer perceptrons

A multilayer perceptron (MLP) is a feedforward network which generalizes the standard perceptron by having one or more hidden layers. Classically, one hidden layer MLPs were used since they already are able to approximate any function [Hor91, Cyb89], as stated by the following theorem.

Theorem 1 (Universal approximation theorem)**.** *A feed-forward network with a single hidden layer containing a finite number of neurons (i.e., a multilayer perceptron), can approximate any continuous function (under mild assumptions on the activation function).*

Here, we will restrict ourselves to the classical MLP with two layers (one hidden layer and one output layer), where we have

$$\mathbf{y} = \mathbf{f}(\mathbf{W}^2\mathbf{f}(\mathbf{W}^1\mathbf{x}))$$

where $\mathbf{W}^1$ denotes the hidden layer and $\mathbf{W}^2$ denotes the output layer.

2.2 Computing the forward sweep

We need to be able to compute the output(s) of an MLP given its inputs. Consider a J-dimensional input vector

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_J \end{bmatrix}$$

and the first layer weight matrix (given I hidden units):

$$\mathbf{W}^1 = \begin{bmatrix} w_{11}^1 & \cdots & w_{1J}^1 \\ \vdots & \ddots & \vdots \\ w_{I1}^1 & \cdots & w_{IJ}^1 \end{bmatrix} .$$

The input activation is then written as

$$\mathbf{W}^1\mathbf{x} = \begin{bmatrix} w_{11}^1 x_1 + w_{12}^1 x_2 + \cdots + w_{1J}^1 x_J \\ \vdots \\ w_{I1}^1 x_1 + w_{I2}^1 x_2 + \cdots + w_{IJ}^1 x_J \end{bmatrix} .$$

and the output of the hidden units is given by

$$\mathbf{h} = \mathbf{f}(\mathbf{W}^1\mathbf{x}) = \begin{bmatrix} f(w_{11}^1 x_1 + w_{12}^1 x_2 + \cdots + w_{1J}^1 x_J) \\ \vdots \\ f(w_{I1}^1 x_1 + w_{I2}^1 x_2 + \cdots + w_{IJ}^1 x_J) \end{bmatrix}$$

where **f** is the activation function applied to all of its inputs. For the output units we have exactly the same computations. That is, assuming K output units, we have

$$\mathbf{y} = \mathbf{f}(\mathbf{W}^2\mathbf{h}) = \begin{bmatrix} f(w_{11}^2 h_1 + w_{12}^2 h_2 + \cdots + w_{1I}^2 h_I) \\ \vdots \\ f(w_{K1}^2 h_1 + w_{K2}^2 h_2 + \cdots + w_{KI}^2 h_I) \end{bmatrix} .$$

2.3 Backprop

The delta rule only worked for single-layer neural networks. How to generalize this to a learning rule for multi-layer neural networks? Such a learning rule was independently discovered by e.g. Bryson (1969), Werbos (1974), Parker (1982), Rumelhart et al. (1986) and LeCun (1986). This learning rule is known as the *generalized delta rule* or *backprop(agation of error)*.

In the following, we develop this learning algorithm. We first introduce some notation for convenience. We use $\mathbf{w} = (\text{vec}(\mathbf{W}^1); \text{vec}(\mathbf{W}^2))$ to collect all weights into a large weight vector. Here, semicolon denotes stacking of column vectors. We define the squared error loss function as

$$E(\mathbf{w}) = \sum_n E^n(\mathbf{w})$$

such that

$$E^n(\mathbf{w}) = \frac{1}{2}\sum_k (y_k^n - t_k^n)^2$$

is the squared loss for a single data point.

For gradient descent, we use

$$\mathbf{w} \leftarrow \mathbf{w} - \gamma \nabla E .$$

and for each element of the gradient we need partial derivatives

$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E^n}{\partial w_{ji}}$$

So, if we can compute the right hand side terms $\frac{\partial E^n}{\partial w_{ji}}$ for weights in the hidden layer and the output layer then we are done. To this end, we use backpropagation. To minimize E by gradient descent, the partial derivative of E^n with respect to each weight within the network needs to be computed. For a given data point, n, this partial derivative is computed in two passes:

1. a forward pass through the network
2. a backward pass which propagates derivatives back through the layers.

Hence, backpropagation (of error).

A more formal development is as follows. First, in a forward pass compute

$$\mathbf{y}^{(n)} = \mathbf{f}(\mathbf{W}^2\mathbf{f}(\mathbf{W}^1\mathbf{x}^{(n)}))$$

where superscript (n) denotes the nth example.

Gradient descent requires us to know how the error changes as a function of the weights (here w.r.t. one data point). Using the chain rule, we may write:

$$\frac{\partial E^n}{\partial w_{ji}} = \frac{\partial E^n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}$$

where we ignore indices for data point (n) and network layer (k). Recall that activation is the net input to a unit z_j which can be either a hidden unit or an output unit.

We rewrite the preceding as:

$$\frac{\partial E^n}{\partial w_{ji}} = \delta_j \frac{\partial a_j}{\partial w_{ji}}$$

where we define the *error* $\delta_j = \frac{\partial E^n}{\partial a_j}$. The error quantifies the change in E as we change the activation. Since $a_j = \sum_k w_{jk} z_k$, it holds that:

$$\frac{\partial a_j}{\partial w_{ji}} = \frac{\partial \sum_k w_{jk} z_k}{\partial w_{ji}} = z_i\,.$$

Hence, the change in E as a function of a weight change is given by:

$$\frac{\partial E^n}{\partial w_{ji}} = \delta_j z_i\,.$$

So if we can write down the error (δ) with respect to an arbitrary weight (hidden or output weight), then we can run (stochastic) gradient descent!

2.3.1 Error for the output units

We first compute the error for the output units:

$$\begin{aligned}\delta_j &= \frac{\partial E^n}{\partial a_j}\\ &= \frac{\partial E^n}{\partial y_j}\frac{\partial y_j}{\partial a_j}\\ &= \frac{\partial E^n}{\partial y_j}\frac{\partial f(a_j)}{\partial a_j}\end{aligned}$$

where we again applied the chain rule and use $y_j = f(a_j)$.

Recall that $\frac{\partial f(a_j)}{\partial a_j}$ is just the derivative of the activation function. Let's call it $f'(a_j)$. Also note that

$$\frac{\partial E^n}{\partial y^n} = \frac{\partial \frac{1}{2}(y^n - t^n)^2}{\partial y^n} = (y^n - t^n)\,.$$

Hence,

$$\delta_j = (y^n - t^n) f'(a_j)\,.$$

2.3.2 Error for the hidden units

The problem is that, in contrast to the output units, we don't know what the hidden units ought to do... But we can compute how fast the error changes as we change a hidden activity. Instead of using desired activities to train the hidden units, use error derivatives w.r.t. hidden activities. Each hidden activity can affect many output units and can therefore have many separate effects on the error. These effects must be combined. Once we have the error derivatives for the hidden activities, its easy to get the error derivatives for the weights going into a hidden unit. A more formal development is as follows.

We use the chain rule in a smart way and write

$$\delta_j = \frac{\partial E^n}{\partial a_j} = \sum_k \frac{\partial E^n}{\partial a_k} \frac{\partial a_k}{\partial a_j}$$

where the sum runs over all output units k to which hidden unit j sends connections.Variations in a_j give rise to variations in the error function (difference between predicted and actual output) through variations in the a_k.

The first partial derivative on the right-hand side is just the error for the output units k to which j connects. I.e. $\frac{\partial E^n}{\partial a_k} = \delta_k$, which we already know how to compute (see previous section). So, we have

$$\delta_j = \sum_k \delta_k \frac{\partial a_k}{\partial a_j} .$$

The second partial derivative expresses how the activity of output unit k changes as a function of a change in hidden unit j:

$$\frac{\partial a_k}{\partial a_j} = \frac{\partial \sum_i w_{ki} f(a_i)}{\partial a_j} = w_{kj} \frac{\partial f(a_j)}{\partial a_j} = w_{kj} f'(a_j) .$$

By plugging this back in, we obtain

$$\delta_j = f'(a_j) \left[\sum_k \delta_k w_{kj} \right] .$$

2.3.3 Recipe

The full backprop algorithm becomes:

- Apply an input vector and propagate it forward to get the activations of hidden and output units:
$$\mathbf{y}^{(n)} = \mathbf{f}(\mathbf{W}^2 \mathbf{f}(\mathbf{W}^1 \mathbf{x}^{(n)}))$$
- Evaluate the delta terms for all the output units:
$$\delta_k = (y^n - t^n) f'(a_k) .$$
- Propagate back the deltas to obtain the delta terms for the hidden units:
$$\delta_j = f'(a_j) \left[\sum_k \delta_k w_{kj} \right] .$$
- Evaluate the required derivatives for the output units:
$$\frac{\partial E^n}{\partial w_{ji}} = (y^n - t^n) f'(a_j) h_i$$

- Evaluate the required derivatives for the hidden units:

$$\frac{\partial E^n}{\partial w_{ji}} = f'(a_j) \left[\sum_k \delta_k w_{kj} \right] x_i$$

Note that this backpropagation strategy works for networks of arbitrary depth (though other problems emerge in very deep neural networks).

Recall that an error surface for the quadratic loss and a monotonic activation function is a bowl with a unique global minimum. Unfortunately this does not hold for MLPs that use non-linear activation functions. Hence, we are only guaranteed to obtain a local optimum. Often this optimum is good enough.

2.4 Batch learning versus online learning

If the dataset is highly redundant, the gradient on the first half is similar to the gradient on the second half. Instead update weights using the gradient on the first half and then get a gradient for the new weights on the second half. An extreme version of this approach updates weights after each case (known as online learning). Mini-batches are usually better than online learning. Here, less computation is used for updating the weights. Computing the gradient for many cases simultaneously uses matrix-matrix multiplications which are very efficient, especially on GPUs. Note though that mini-batches need to be balanced for different output classes

2.5 Momentum

Going downhill reduces the error, but the direction of steepest descent does not point at the minimum unless the ellipse is a circle. In other words, the gradient is big in the direction in which we only want to travel a small distance and the gradient is small in the direction in which we want to travel a large distance. Solutions to this problem are to (1) use fancy optimization methods (e.g. conjugate gradient, quasi-newton, LM, L-BFGS) or (2) use *momentum*.

Let $\Delta w(t)$ denote the weight change at the tth iteration datapoint. Momentum proposes the following way to compute the weight change:

$$\Delta w(t) = -\gamma \frac{\partial E(\mathbf{w})}{\partial w} + \alpha \Delta w(t-1)$$

where α is the momentum parameter.

At the beginning of learning there may be very large gradients. So it pays to use a small momentum (e.g. 0.5). Once the large gradients have disappeared and the weights are stuck in a ravine the momentum can be smoothly raised to its final value (e.g. 0.9 or even 0.99). This allows us to learn at a rate that would cause divergent oscillations without the momentum.

3 Reading material

Study parts II. Multivariate non-linear mappings and III. The multilayer perceptron of [Bis94] which can be downloaded here. Karpathy's blog is also an excellent resource. For a detailed example on how backpropagation works, consult Chapter 2 of neuralnetworksanddeeplearning.

References

[Bis94] Christopher Bishop. Neural Networks and their applications, 1994.

[Cyb89] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Math. Control. Signals, Syst.*, 2:303–314, 1989.

[Hor91] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.