Backpropagation

Roland Kwitt

December 11, 2018

This document is meant to serve as supplementary material to the lecture on learning via *backpropagation* in the context of neural networks. We will derive the backprop. rule(s) on the example of a *logistic regression classifier*. First, let us define the class of affine functions as

$$L_d = \{h_{\mathbf{w},b} : \mathbf{w} \in \mathbb{R}^d, b \in \mathbb{R}\} \tag{1}$$

where

$$h_{\mathbf{w},b}(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + b$$

Typically, we incorporate the bias term b into $\boldsymbol{\theta} = [b, \mathbf{w}]$ and let $\mathbf{x} = [1, \mathbf{x}]$. If not further specified, we will not explicitly mention that we add the bias term and simply consider $\mathbf{x}$ and $\boldsymbol{\theta}$.

In logistic regression, the classifier is a composition of a sigmoid, i.e.,

$$\mathrm{sigm}(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

over the class of affine functions L_d, see Eq. (1). In other words, these are functions of the form

$$h_{\boldsymbol{\theta}} : \mathbb{R}^d \to [0, 1], \quad \mathbf{x} \mapsto h_{\boldsymbol{\theta}}(\mathbf{x}) = \mathrm{sigm}(\langle \boldsymbol{\theta}, \mathbf{x} \rangle)$$

We interpret the output of $h_{\boldsymbol{\theta}}(\mathbf{x})$ as the *posterior probability* that the label of $\mathbf{x}$ is 1. More formally

$$P(Y = 1 | X = \mathbf{x}; \boldsymbol{\theta}) := h_{\boldsymbol{\theta}}(\mathbf{x}) \ \ .$$

The output in our example can take on only two values, 0 and 1. If we model this output via a random variable, the proper choice is a Bernoulli random variable with parameter $\gamma \in [0, 1]$. In detail, we have the Bernoulli distribution $p(y; \gamma)$ given by

$$p(y; \gamma) = \begin{cases} \gamma^y (1 - \gamma)^{1-y}, & \text{if } y \in \{0, 1\} \\ 0, & \text{else} \end{cases}$$

Under this model, and given an i.i.d. dataset $\mathbf{x}_1, \ldots, \mathbf{x}_n$ where each sample has an associated label $y_i \in \{0, 1\}$, we can write the likelihood $l(\boldsymbol{\theta})$ as

$$l(\boldsymbol{\theta}) = \prod_{i=1}^{n} \left[\underbrace{\frac{1}{1 + e^{-\mathbf{x}_i^\top \boldsymbol{\theta}}}}_{\pi_i} \right]^{y_i} \left[1 - \frac{1}{1 + e^{-\mathbf{x}_i^\top \boldsymbol{\theta}}} \right]^{1-y_i} \tag{3}$$

Note that γ in the Bernoulli distribution is exactly $h_{\boldsymbol{\theta}}(\mathbf{x}) = \pi_i$ here. Switching to the log-domain, we get

$$-\log l(\boldsymbol{\theta}) = -\sum_{i=1}^{n} y_i \log(\pi_i) + (1 - y_i) \log(1 - \pi_i) \tag{4}$$

This is called the *binary cross-entropy (BCE)*. As $\mathbf{x}_1, \ldots, \mathbf{x}_n$ as well as $y_1, \ldots, y_n$ (the corresponding labels) are given to us for training and $\boldsymbol{\theta}$ is our parameter we wish to optimize, we write this as

$$C(\boldsymbol{\theta}) = -\log l(\boldsymbol{\theta}) \tag{5}$$

and call $C(\boldsymbol{\theta})$ our *BCE cost function*. Cost minimization is therefor equivalent to maximizing the likelihood (as we took the negative log-likelihood as our cost function). In the multiclass case, i.e., when we do not just have binary outputs, we have

$$l(\boldsymbol{\theta}) = \sum_{i=1}^{n} \log p_{g_i}(\mathbf{x}_i; \boldsymbol{\theta})$$

with $p_k(\mathbf{x}_i; \boldsymbol{\theta}) = P(Y = k | X = \mathbf{x}_i; \boldsymbol{\theta})$, i.e., a multinomial distribution.

When we look at our two-class case more closely, we see that logistic regression can easily be implemented via simple neural network architecture:

- Input: $\mathbf{x}_i$
- Linear layer: $z_i = \langle \boldsymbol{\theta}, \mathbf{x}_i \rangle$
- Sigmoid layer: $\pi_i = \mathrm{sigm}(\mathbf{z}_i)$
- BCE loss

We will see next that this can be done in an alternative manner that also allows for multiclass output. Specifically, we model the posterior probability of label $Y = k$, given $X = \mathbf{x}$ as a *softmax function*

$$P(Y = k | X = \mathbf{x}; \boldsymbol{\Theta}) = \frac{e^{\mathbf{x}^\top \boldsymbol{\theta}_k}}{\sum_j e^{\mathbf{x}^\top \boldsymbol{\theta}_j}}$$

where $\boldsymbol{\Theta} = \{\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_k\}$. By letting

$$P(Y = j | X = \mathbf{x}_i; \boldsymbol{\Theta}) =: \pi_{ij}$$

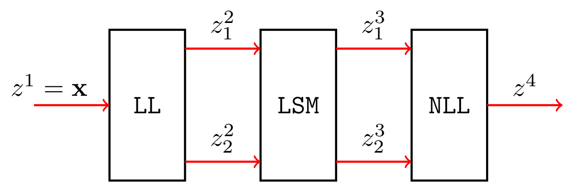

Figure 1: Logistic regression network architecture.

and noting

$$\sum_j \pi_{ij} = 1$$

we can write

$$l(\boldsymbol{\Theta}) = \prod_{i=1}^{n} \pi_{i0}^{\mathbb{I}_0(y_i)} \pi_{i1}^{\mathbb{I}_1(y_i)}$$

in the two-class case. Note that $\mathbb{I}_0(y_i)$ is the indicator function that returns 1 if $y_i = 0$ and 0 otherwise. Switching to log-domain again gives

$$C(\boldsymbol{\Theta}) = -\log l(\boldsymbol{\Theta}) = -\sum_{i=1}^{n} \mathbb{I}_0(y_i)\log(\pi_{i0}) + \mathbb{I}_1(y_i)\log(\pi_{i1})$$

which can again be easily implemented using a neural network:

- Input: $\mathbf{x}_i$
- Linear layer (k output units, e.g., $k = 2$): $z_k = \langle \boldsymbol{\theta}_k, \mathbf{x}_i \rangle$
- Log-softmax layer: computes $z_{ik} = \log(\pi_{ik})$, i.e., (log) class probabilities
- Negative log-likelihood loss: simply $-z_{ik}$ if $\mathbb{I}_k(y_i) \neq 0$.

Alternatively, you can implement this using the *cross-entropy loss* which combines a log-softmax layer with the negative log-likelihood loss.

Now, we can write the output z^4 as a function of z_1^3 and z_2^3, and z_1^3 as a function of z_1^2, etc. Overall, we get

$$C(\boldsymbol{\Theta}) = z^4\{z_1^3[z_1^2(z^1, \boldsymbol{\theta}_1), z_1^2(z^1, \boldsymbol{\theta}_2)], z_2^3[z_1^2(z^1, \boldsymbol{\theta}_1), z_1^2(z^1, \boldsymbol{\theta}_2)]\}$$

Lets actually perform the computation for

$$\frac{\partial C}{\boldsymbol{\theta}_1} =?$$

Using the *chain-rule*, we get

$$\begin{aligned}\frac{\partial C}{\boldsymbol{\theta}_1} = &\frac{\partial z^4}{\partial z_1^3}\frac{\partial z_1^3}{\partial z_1^2}\frac{\partial z_1^2}{\partial \boldsymbol{\theta}_1} + \frac{\partial z^4}{\partial z_1^3}\frac{\partial z_1^3}{\partial z_2^2}\frac{\partial z_2^2}{\partial \boldsymbol{\theta}_1} \\ &\frac{\partial z^4}{\partial z_2^3}\frac{\partial z_2^3}{\partial z_1^2}\frac{\partial z_1^2}{\partial \boldsymbol{\theta}_1} + \frac{\partial z^4}{\partial z_2^3}\frac{\partial z_2^3}{\partial z_2^2}\frac{\partial z_2^2}{\partial \boldsymbol{\theta}_1}\end{aligned}$$

As we can clearly see, several terms are computed multiple times. The following recursive pattern emerges:

$$\frac{\partial C}{\partial z_{i}^{L}} = \sum_{j} \frac{\partial C}{\partial z_j^{L+1}} \frac{\partial z_j^{L+1}}{\partial z_{i}^{L}}$$

In principle, we need to be able to compute the derivative of the output of each layer (here: z_j^{L+1}) w.r.t. all its input (here: z_i^L). If we set

$$\delta_i^L = \frac{\partial C}{\partial z_i^L}$$

then the recursion can be written as

$$\frac{\partial C}{\partial z_{i}^{L}} = \sum_{j} \delta_j^{L+1} \frac{\partial z_j^{L+1}}{\partial z_{i}^{L}}$$

which now shows that we essentially only need the derivative of the output of a layer w.r.t. its inputs. The other gradients are provided by the other layers, multiplied together and summed up.

In our example,

$$\frac{\partial C}{\partial z_i^L} \equiv \frac{\partial z^4}{\partial z_i^L}$$

so

$$\frac{\partial C}{\partial z_1^3} \equiv \frac{\partial z^4}{\partial z_1^3} = \sum_{j=1}^{1} \underbrace{\frac{\partial z^4}{\partial z^4}}_{=1} \frac{\partial z^4}{\partial z_1^3}$$

showing that $\delta^4 = 1$ and we get the following backward sequence for the gradients

$$\delta^1 \leftarrow \delta^2 \leftarrow \delta^3 \leftarrow 1$$

Equivalently, if a layer has parameters (as in our illustration for $\boldsymbol{\theta}_1$), we get

$$\frac{\partial C}{\partial \boldsymbol{\theta}^L} = \sum_j \frac{\partial C}{\partial z_j^{L+1}} \frac{\partial z_j^{L+1}}{\partial \boldsymbol{\theta}^L} = \sum_j \delta_j^{L+1} \frac{\partial z_j^{L+1}}{\partial \boldsymbol{\theta}^L}$$

and we see that it is also enough to simply have the derivative of the layer's output w.r.t. its parameters ready.

Example

In this example, we build a rectified linear unit (ReLU) layer. Given input x, we have

$$\text{relu}(x) = \max\{0, x\}$$

Setting $z = \mathrm{relu}(x)$ we get

$$\frac{\partial z}{\partial x} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{else.} \end{cases}$$

This is all we need, as the ReLU activation layer *does not* have any additional parameters.