Notes on Back Propagation in 4 Lines

Lili Mou
moull12@sei.pku.edu.cn

March, 2015

Congratulations! You are reading the clearest explanation of forward and backward propagation I have ever seen. In this note, we are going to delve deep into maximal math details of neural networks with minimal notations and concepts. We do not define unnecessary terms like *correction*, *error propagation*, *local gradient*, *delta rule*, odd notations like *net* containing 3 letters for one variable, or excessive sub-/super-scripts like $w_{ij}^{(l)}$ associating with l-th layer or $(l+1)$-th (?). We derive backpropagation by pure calculus in 4 lines, which is much more clearer. The appendix of matrix-vector representations also serves as a cheat sheet when implementing neural networks.

We assume readers have basic knowledge about calculus, especially the chain rule for composition functions' derivatives. We also assume readers know about basic concepts in machine learning, e.g., classification, gradient descent, even though some materials will also be introduced for self-containedness.

1 Perceptions, Neurons, Logits, Units, etc

Consider the simplest classification problem: two dimensional data with two target classes. We would like to use a linear classifier[1] to distinguish the two classes. The process is much like cutting an uneven cake using a knife, cherries for you and chocolate for me. See Figure 1. The incision is a line in the 2-d plane, called *decision boundary*. Generally, a decision boundary of n-dimensional space can be represented by a hyperplane

$$\boldsymbol{w}^T\boldsymbol{x} + b$$

where $\boldsymbol{w} \in \mathbb{R}^n$ and $b \in \mathbb{R}$ are parameters; $\boldsymbol{x} \in \mathbb{R}^n$ is a data sample.

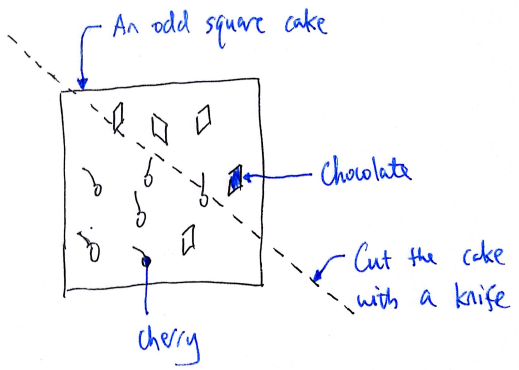


Figure 1: Cutting the cake as a 2-d binary classification problem.

Let us assume $\boldsymbol{w} = (1, 1)$. Then the upper-right half space satisfies $\boldsymbol{w}^T\boldsymbol{x} + b > 0$; the lower-left satisfies $\boldsymbol{w}^T\boldsymbol{x} + b < 0$ (b is a constant). To easily determine whether a cherry (with target label 0) or

[1] Don't ask me why linear. If you like it, you use it; if you don't like it, don't use it.

a piece of chocolate (with target label 1) is correctly classified, we define an indicating variable y to be the output of the binary classifier as

$$y = \begin{cases} 1, & \text{if } \boldsymbol{w}^T\boldsymbol{x} + b \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

or more compactly,

$$y = \text{sgn}(\boldsymbol{w}^T\boldsymbol{x} + b)$$

The above *perception* model was invented by Rosenblatts in 1958, which is surprisingly valid even today. The learning algorithm of perceptions is rather simple, whose objective is to find $\boldsymbol{w}$ and b which minimize the training error. The iterative scheme and its convergence theorem can be found in [Haykin, 2009].

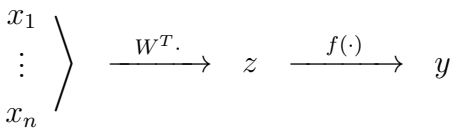
Even though it is the empirical error rate (training error), from a learning theory prospective, that counts most—which the perception learning algorithm (or its variations) maximizes exactly—the sgn causes trouble because

- The derivative of sgn is either 0 or infinity, preventing it from gradient-based optimization.
- The hard decision is not robust: data samples near or far away from the boundary make no difference in computing the boundary.

To solve this problem, we can substitute sgn with other functions, e.g., sigmoid, tanh, ReLU, etc, shown in Figure 2. This model is called a *neuron*, a *logit*, or just simply a *unit*. Sometimes, a *perception* is also referred to interchangeably. Formally, a neuron takes vector $\boldsymbol{x} \in \mathbb{R}^n$ as input, sums with weight w, and outputs through an activation function. Let z be a scalar, we have

$$z = \boldsymbol{w}^T\boldsymbol{x} + b$$
$$y = f(z)$$

illustrated as follows.



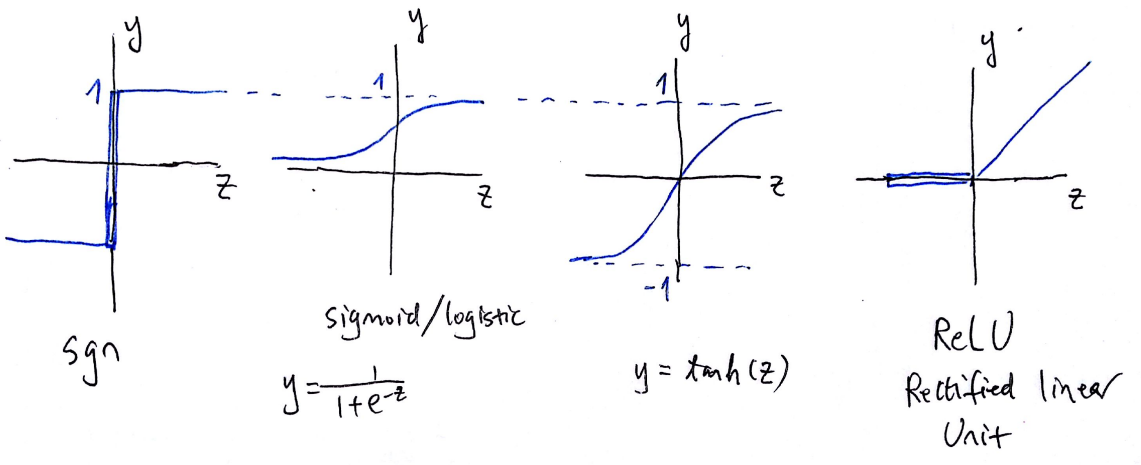


Figure 2: Activation functions.

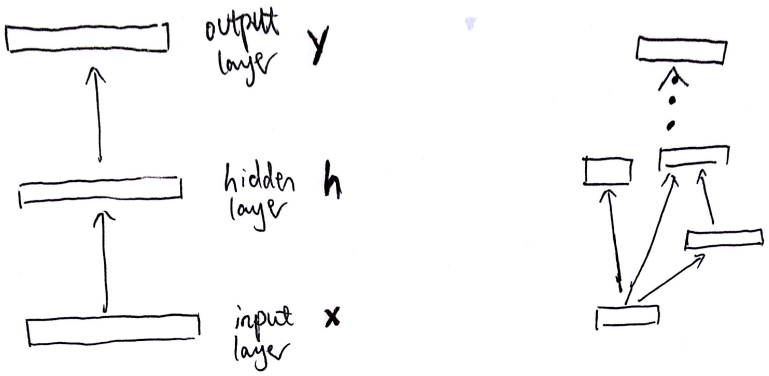


Figure 3: Multi-layer neural networks. The left is a classical architecture with one hidden layer. However, we can design an arbitrary structures as long as no loop exists. We call them feed-forward neural networks. If loops exists, it becomes a recurrent neural network, which is actually equivalent to feed-forward neural network after being unrolled along time.

If we need multiple outputs, e.g., cherry, cream, etc., the output becomes a vector $\boldsymbol{y} \in \mathbb{R}^m$. The set of m neurons, called a *neural layer*, acts as

$$\boldsymbol{z} = W^T \boldsymbol{x} + \boldsymbol{b} \tag{1}$$

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{z}) \tag{2}$$

where $W \in \mathbb{R}^{m \times n}$ and $\boldsymbol{b}, \boldsymbol{z} \in \mathbb{R}^m$. In fact, the process is exactly the same—transforming input by a matrix, adding a bias term, and activating through $\boldsymbol{f}$. Here, $\boldsymbol{f} : \mathbb{R}^m \to \mathbb{R}^m$ is usually a pointwise function, except for softmax.

Following list two notes.

1. $\boldsymbol{f}$ is typically nonlinear. It makes no difference for a single neuron, but empowers the model if multiple neural layers are stacked, as we will do in the next section.

2. $\boldsymbol{f}$ is typically monotonic (maybe not strictly). As the classifier is linear currently, it makes little sense to apply bell-shaped, say, activation functions.

2 Multi-layer Perceptions, or Neural Networks

As we see, a perception, or a single neural does not suffice for most real-world problems due to the linearity. One direct approach is to stack multiple layers of neurons, also known as a *neural network*, shown in Figure 3. It can be approved that, for any boolean or continuous function, a 2-layer neural network, with sufficient hidden units, can approximate it arbitrarily; for any function, a 3-layer neural network suffices.

Figure 4 illustrates an example of XOR. Though XOR cannot be represented by a single neural, multi-layer neural networks have no difficulty in distinguishing the two classes: hidden units h_1 and h_2 map original input $\boldsymbol{x} = (1, 0)$ and $(0, 1)$ to $\boldsymbol{h} = (1, 0)$, accomplished by the non-linear transformation sgn; then the problem becomes linear separable in $\boldsymbol{h}$'s space.

As we see in Figure 3, the (once) most widely applied multi-layer neural network is such that we stack two layers with full connections between successive two layers. However, it is not necessarily the case: the architecture, in fact, can be arbitrarily large or complex. (Designing neural architectures remains a $\$64,000,000$ question.) Hence, when representing the forward propagation, we do not need, and also we cannot unroll the propagation process. What we shall do is to provide the iterative

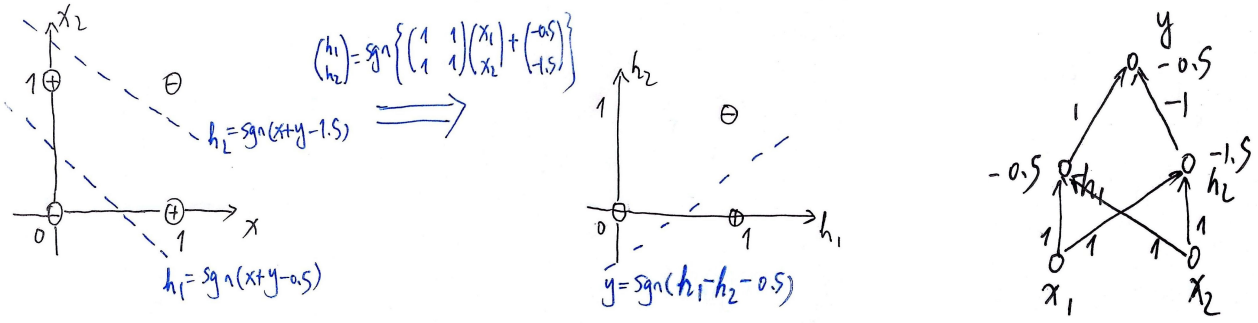


Figure 4: The XOR problem, and the neural network solving it.

scheme. In this way, we eliminate unnecessary superscripts ambiguously indicating the layer, as in many other tutorials.

Let $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{y} \in \mathbb{R}^m$ be the output of two successive layers with full connections from $\boldsymbol{x}$ to $\boldsymbol{y}$. Obviously, the lower layer's output $\boldsymbol{x}$ is also the upper layer's input. Hence, notations $\boldsymbol{x}$ and $\boldsymbol{y}$ are quite meaningful. The propagation is exactly Equation 1 and 2, not repeated in detail here.

3 Back propagation

Parameter estimation, playing a vital role in all machine learning models—that is why we say "learning"—has a same scheme: we define an objective function either by supervised error, unsupervised measure (e.g., likelihood), or the gain/penalty in reinforcement learning; then we optimize the objective. Oftentimes, there is no closed form solution to the objective, and thus we need to apply numerical methods. Gradient descent (or maybe ascent), can be regarded as a greedy algorithm, rushing to the (local) optimum according to the current gradient. Let Θ be the parameters, J be the objective to minimize, and α be the learning rate. The iterative scheme is

$$\Theta \leftarrow \Theta - \alpha \cdot \nabla_\Theta J(\Theta)$$

Now, the only problem for multi-layer neural networks is how to compute the derivatives. As calculus had been developed for 300 years, it took only(?) 5 years before people realized the efficient analytic solution to gradient calculation. The algorithm is now well-known as *back propagation* without magic.[2]

Back propagation computes $\frac{\partial J}{\partial \boldsymbol{y}}$ or $\frac{\partial J}{\partial \boldsymbol{x}}$ in a recursive manner, where J is the cost function we define, and $\boldsymbol{y}$ or $\boldsymbol{x}$ is the output of a neural layer, $\boldsymbol{y}$ representing the upper layer and $\boldsymbol{x}$ representing the lower layer. Recall that $\boldsymbol{y}$ to the upper layer is as $\boldsymbol{x}$ to the lower layer.

- **Initialization,** where $\boldsymbol{y}$ is output layer. $\frac{\partial J}{\partial \boldsymbol{y}}$ is known because we define the cost function.
- **Recursion.** If it is not the output layer, we denote it as $\boldsymbol{x}$ rather than $\boldsymbol{y}$, assuming its parent's partial derivative, i.e., $\frac{\partial J}{\partial \boldsymbol{y}}$ is known by recursion. We write again the forward propagation

$$z_j = w_{j1}x_1 + w_{j2}x_2 + \cdots + w_{jn}x_n + b_j, \quad i = 1, 2, \cdots, m \tag{3}$$

$$y_j = f(z_j) \tag{4}$$

$$J = J(\boldsymbol{y}) = J(y_1, y_2, \cdots, y_m) \tag{5}$$

[2]What is trivial today was not trivial at all in the history.

The observation is that $\boldsymbol{y}$ layer has multiple units, $y_1, y_2, \cdots, y_m$; also, J is independent of either $\boldsymbol{z}$ or $\boldsymbol{x}$ given $\boldsymbol{y}$, the so called "Markov blanket." Back propagation is then a 4-line derivation.

$$\frac{\partial J}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial J}{\partial y_j} \cdot \frac{\partial y_j}{\partial x_i} \tag{6}$$

$$= \sum_{j=1}^{m} \frac{\partial J}{\partial y_j} \cdot \frac{\partial y_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial x_i} \tag{7}$$

Equation 6 holds because of the chain rule for composition functions. Note that J is related to $y_1, \cdots, y_m$, each of which contains x_i. Hence, we have to sum over these m partial derivatives. Equation 7 is computable because $\frac{\partial J}{\partial y_i}$ is known either derives directly from cost function or computed recursively; $\frac{\partial y_i}{\partial z_j} = f'\big|_{z_j}$ derived from Equation 4; $\frac{\partial z_j}{\partial x_i} = w_{ji}$ from Equation 3. Finally, we compute the partial derivatives of the cost function with respect to model parameters according to Equation 3.

$$\frac{\partial J}{\partial b_j} = \frac{\partial J}{\partial z_j} \cdot \frac{\partial z_j}{\partial b_j} = \frac{\partial J_j}{\partial z_j} \tag{8}$$

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial z_j} \cdot \frac{\partial z_j}{w_{ji}} = \frac{\partial J}{\partial z_j} \cdot x_i \tag{9}$$

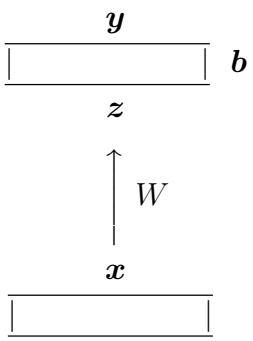
To wrap up, back propagation computes $\frac{\partial J}{\partial \boldsymbol{y}}$ recursively (2 lines), and then computing $\frac{\partial J}{\partial \boldsymbol{w}}$ and $\frac{\partial J}{\partial \boldsymbol{b}}$ is another 2-line derivations.

Appendix: Matrix-Vector Representation

If the forward and backward propagations are represented as a matrix-vector product, it will be much easier to memorize, and much faster in implementation. Various existing numerical packages are available, e.g., `MATLAB`, `numpy` for python, or even `blas`, a basic linear algebra subprogram, written in `fortran` with the interface `cblas` for `C/C++`.

Let n_x be the lower layer's unit number, n_y be the upper layer's unit number, and n_{data} be the number of data samples (in one batch).

Let $\boldsymbol{x} \in \mathbb{R}^{n_x \times n_{\text{data}}}$ be the output of the lower layer (or the input of the upper layer), each column in $\boldsymbol{x}$ being a data sample.[3] Let $W \in \mathbb{R}^{n_y \times n_x}$ and $b \in \mathbb{R}^{n_y}$ be the parameters of the current layer, depicted as follows.



- **Forward propagation.**

$$\boldsymbol{z} = W \cdot \boldsymbol{x} + \text{repmat}(b, 1, n_{\text{data}}) \qquad \in \mathbb{R}^{n_y \times n_{\text{data}}}$$

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{z}) \qquad \in \mathbb{R}^{n_y \times n_x}$$

[3] We represented $\boldsymbol{x}$ in a vector fashion, because n_{data} does not lead to trouble.

- **Back propagation.** Let $\dfrac{\partial E}{\partial y} \in \mathbb{R}^{n_y \times n_{\text{data}}}$ and $\dfrac{\partial y}{\partial z} \in \mathbb{R}^{n_y \times n_{\text{data}}}$ be known.

$$\begin{aligned}
\frac{\partial J}{\partial \boldsymbol{z}} &= \frac{\partial J}{\partial \boldsymbol{y}} .* \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{z}} && \in \mathbb{R}^{n_y \times n_data} \\
\frac{\partial J}{\partial \boldsymbol{x}} &= W^T \cdot \frac{\partial J}{\partial \boldsymbol{z}} && \in \mathbb{R}^{n_y \times n_data} \\
\frac{\partial J}{\partial W} &= \frac{\partial J}{\partial \boldsymbol{z}} \cdot \boldsymbol{x}^T \Big/ n_{\text{data}} && \in \mathbb{R}^{n_y \times n_x} \\
\frac{\partial J}{\partial \boldsymbol{b}} &= W^T \cdot \frac{\partial J}{\partial \boldsymbol{z}} \cdot \mathbf{1}_{n_{\text{data}}} \Big/ n_{\text{data}} && \in \mathbb{R}^{n_y}
\end{aligned}$$

We have some final notes.

- Here, we use `MATLAB` slang, `repmat` representing "repeat matrix," and .* representing pointwise product for matrices or vectors.
- Back propagation is a linear process, one virtue of which is versatility in computing. For example, if we have different cost functions, $J = J_1 + J_2 + \cdots + J_j$, we only need to sum over these partial derivatives. Further, it can be efficiently accomplished by a degraded *dynamic programming* algorithm, where for each layer we compute $\frac{\partial J}{\partial \boldsymbol{y}}$, and then propagate. Another example is weight association. If we impose a constraint $W_1 = W_2 \triangleq W$ for two layers, we only need to pretend that W_1 and W_2 are not equivalent, then $\frac{\partial J}{\partial W} = \frac{\partial J}{\partial W_1} + \frac{\partial J}{\partial W_2}$ due to the linearity inferred from the chain rule of composition functions.

Reference

[Haykin, 2009] Simon Haykin, *Neural Networks and Learning Machines (3rd Edition)*, Pearson Education, Inc.