Machine learning

Lecture 4 - Neural Networks

Lecturer: Haim Permuter *Scribe: Nave Algarici*

Throughout this lecture we introduce Neural Netwoks, starting from a single neuron, and ending with the Backpropagation method. Most of the material for this lecture was inspired by the work of Michael Nielsen [1], and the UFLDL Stanford Univversity Wiki [2].

I. Neural Networks

Neural networks are limited imitations of how our own brains work. They've had a big recent resurgence because of advances in computer hardware. There is evidence that the brain uses only one "learning algorithm" for all its different functions. At a very simple level, neurons are basically computational units that take input (dendrites) as electrical input (called "spikes") that are channeled to outputs (axons).

Neural neworks are typically organized in layers. Layers are made up of a number of interconnected 'nodes' which contain an 'activation function'. Patterns are presented to the network via the 'input layer', which communicates to one or more 'hidden layers' where the actual processing is done via a system of weighted 'connections'. The hidden layers then link to an 'output layer', which is the output of the network.

Neural Networks can be applied to many problems, such as: function approximation, classification, data processing, etc.

We will start by looking at a single neuron, define it's model, and combine neurons to a complete network.

A. Single Neuron Model

A neuron takes k inputs, $x_1, x_2, ..., x_k$, a set of weights $w_1, w_2, ..., w_k$ corresponding to the inputs, a bias b and an activation function $\sigma(z)$ and produces a single output y:

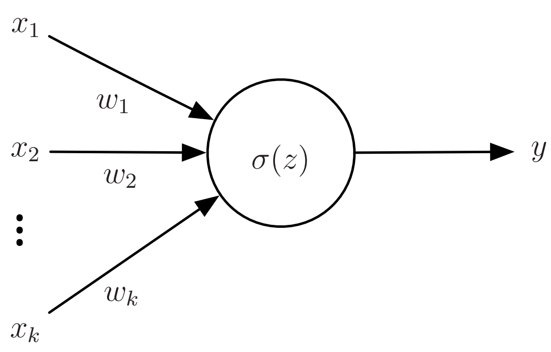

Figure 1. Scheme of a single neuron.

where the output of the neuron is determined by:

$$z = \sum_{i=1}^{k} w_i x_i + b, \tag{1}$$

$$y = \sigma(z). \tag{2}$$

There are many different options for the choice of the activation function. A few of them are:

- $\sigma(z) = \frac{1}{1+e^{-z}}$ - sigmoid function
- $\sigma(z) = sign(z)$ - sign function
- $\sigma(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ - hyperbolic tangent
- $\sigma(z) = \max(0, z)$ - rectified linear unit (RLU)

B. Neural Network Model

As we mentioned before, a Neural Network is organized in layers, where the first layer contains the inputs of the network, the last layer is the output of the network, and the layers in between are called hidden layers. Each layer gets it's inputs from the layer before, and passes it's outputs to the next. We call this step *forward propagation*.

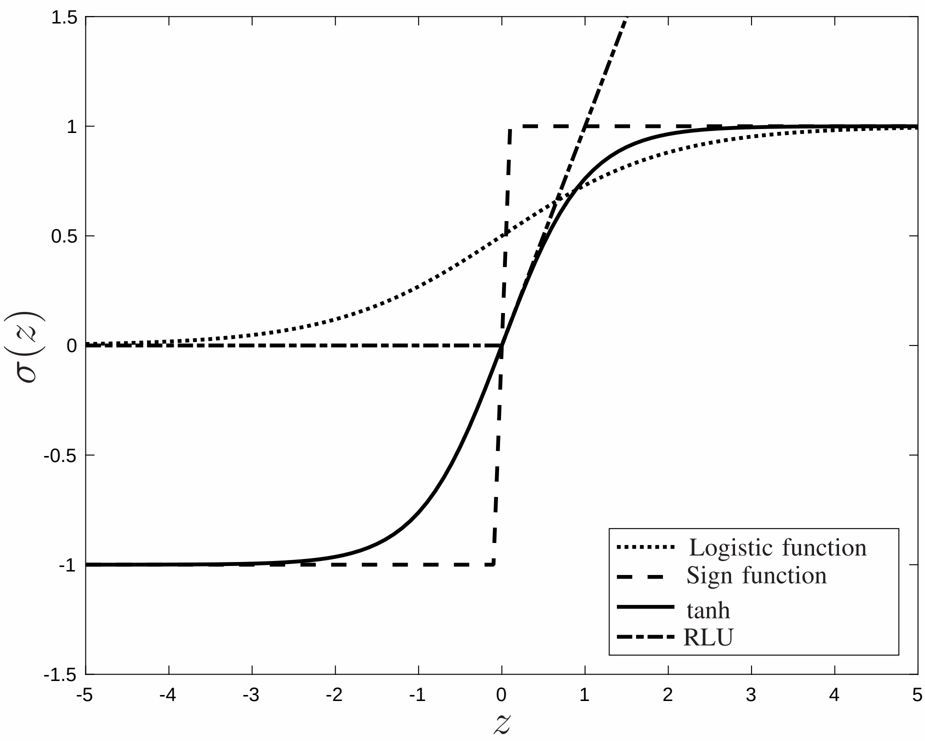

Figure 2. Plot of the activation functions mentioned above.

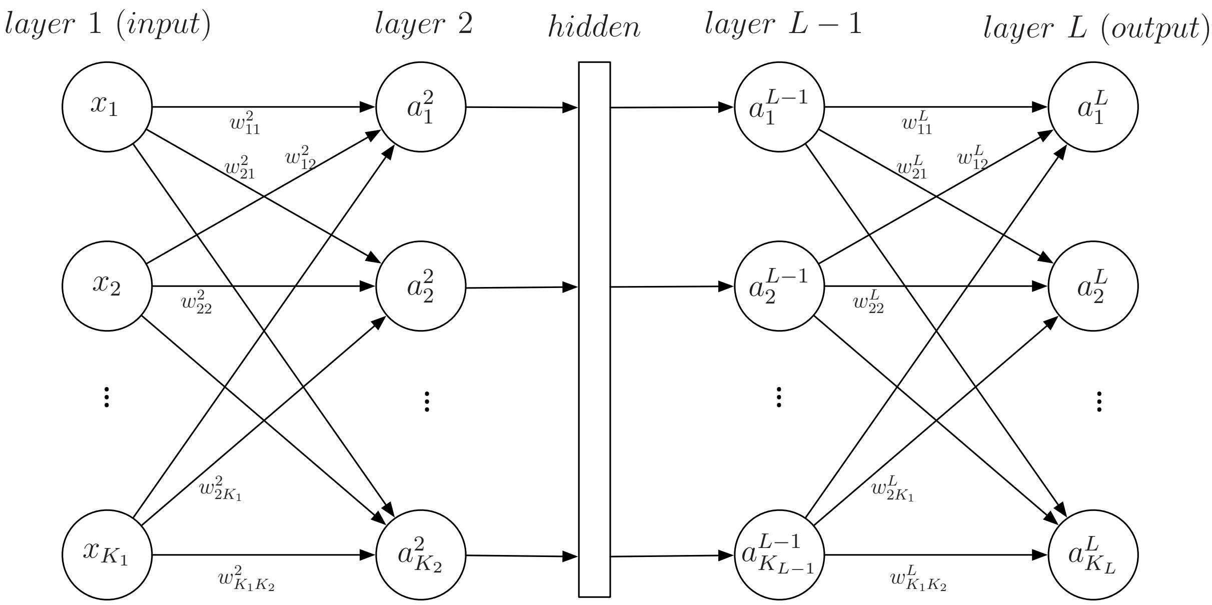

Figure 3. Structure of a general Neural Network

where:

- w^l_{jk} - the weight for the connection from the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer.
- b^l_j - the coefficient we add to the j^{th} neuron in the l^{th} layer.
- K_l - the number of neurons in the l^{th} layer.
- $z^l_j = \sum_{k=1}^{K_l} w^l_{jk} a^{l-1}_k + b^l_j$
- $a^l_j = \sigma(z^l_j)$

By organizing our parameters in matrices and using matrix-vector operations, we can take advantage of fast linear algebra routines to quickly perform calculations in our network.

- $z^l = \left[z^l_1, z^l_2, \ldots, z^l_{K_l}\right]^T$
- $b^l = \left[b^l_1, b^l_2, \ldots, z^l_{K_l}\right]^T$
- $w^l = \begin{bmatrix} w^l_{11} & w^l_{12} & \ldots & w^l_{1K_{l-1}} \\ w^l_{21} & w^l_{22} & \ldots & w^l_{2K_{l-1}} \\ \vdots & \vdots & \ddots & \vdots \\ w^l_{K_l 1} & w^l_{K_l 1} & \ldots & w^l_{K_l K_{l-1}} \end{bmatrix}$
- $a^l = \left[a^l_1, a^l_2, \ldots, a^l_{K_l}\right]^T$
- $z^l = w^l a^{l-1} + b^l$
- $a^l = \sigma(z^l)$
- $\sigma([x_1, \ldots, x_n]^T) = [\sigma(x_1), \ldots, \sigma(x_n)]^T$

Example 1 (XOR function) Consider a simple two input XOR function:

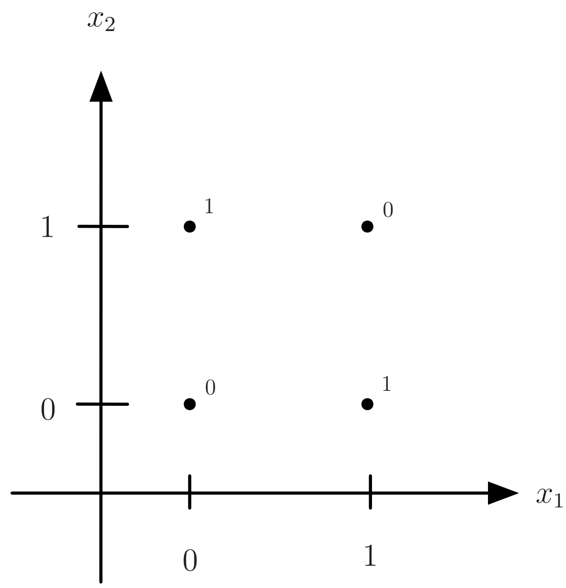

Figure 4. XOR function - illustration of output for binary inputs x_1, x_2

It is easy to see, that the XOR function cannot be approximated using a linear function (there is no line that could seperate the two groups of answers). We now will show, that it is possible to approximate the XOR function using a Neural Network. We will build a Network with three inputs, x_1, x_2, and the third set to '1', a hidden layer of two neurons, and a single neuron output layer. All biases are set to zero. Set the weights of the network as such:

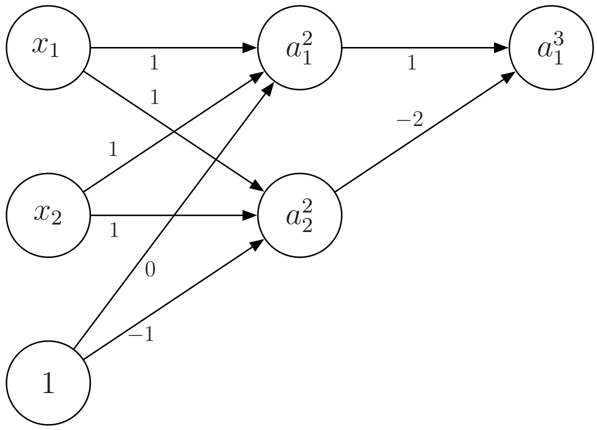

Figure 5. Example: XOR function Neural Network

For this example, the activation function will be the RLU function.

Let's take the input $[x_1, x_2] = [0, 0]$ and insert it to the network:

$$a^2_1 = max(0, 1 * x_1 + 1 * x_2 + 0 * 1) = 0$$

$$a^2_2 = max(0, 1 * x_1 + 1 * x_2 + -1 * 1) = 0$$

$$a^3_1 = max(0, 1 * a^2_1 - 2 * a^2_2) = 0$$

The same goes for the other options for the inputs: $[0, 1], [1, 0], [1, 1]$:

$\boldsymbol{x_1}$	$\boldsymbol{x_2}$	$\boldsymbol{a^2_1}$	$\boldsymbol{a^2_2}$	$\boldsymbol{a^3_1}$
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	2	1	0

We can see that the output a^3_1 fits the XOR function perfectly. Note that the output of each layer is not dependant pof previous layers, only it's own inputs and weights.

C. *Cost Function*

To quantify how well we're achieving this goal we define a cost function:

$$C(w,b) = \frac{1}{2N}\sum_{i=1}^{N}\|a(x_i) - y_i\|^2, \tag{3}$$

where *x* are the input vectors and *y* are there corresponding labels, both are detirmined. So the cost function chagnes it's value depending on the weigts *w* and biases *b*. The cost function given in (3) is the quadratic cost function.

Our main goal is to minimize the cost function, so that the the output from the network will be close to the desired output as possible. To minimize the cost function, we will use a method called *Gradient Decent*.

Gradient descent is a iterative optimization algorithm. To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient (or of the approximate gradient) of the function at the current point.

D. *Backpropagation*

Backpropagation is about understanding how changing the weights and biases in a network changes the cost function. Ultimately, this means computing the partial derivatives $\frac{\partial C}{\partial w^l_{jk}}$ and $\frac{\partial C}{\partial b^l_j}$. But to compute those, we first introduce an intermediate quantity, δ^l_j, which we call the error in the j^{th} neuron in the l^{th} layer.

Backpropagation will give us a procedure to compute the error δ^l_j, and then will relate δ^l_j to $\frac{\partial C}{\partial w^l_{jk}}$ and $\frac{\partial C}{\partial b^l_j}$.

Our goal is to minimize *C* as a function of w and b. To train our neural network, we initialize each parameter w^l_{jk} and each b^l_j to a small random value near zero, and then apply an optimization algorithm such as batch gradient descent. Since *C* is a non-convex function, gradient descent is susceptible to local optima. However, in practice gradient descent usually works fairly well. Note that it is important to initialize the parameters randomly, rather than to all 0's. If all the parameters start off at identical values, then all the hidden layer units will end up learning the same function of the input. The random initialization serves the purpose of *symmetry breaking*.

One iteration of gradient descent updates the parameters w, b as follows:

$$w_{jk}^l = w_{jk}^l - \alpha \frac{\partial C}{\partial w_{jk}^l}, \tag{4}$$

$$b_j^l = b_j^l - \alpha \frac{\partial C}{\partial b_j^l}, \tag{5}$$

where α is the learning rate. The key step is computing the partial derivatives above. We will now describe the backpropagation algorithm, which gives an efficient way to compute these partial derivatives.

We define the error δ_j^l of neuron j in layer l by

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}, \tag{6}$$

starting with the L^{th} layer, we get

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}. \tag{7}$$

Applying the chain rule, we can re-express the partial derivative above in terms of partial derivatives with respect to the output activations

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L}, \tag{8}$$

where the sum is over all neurons k in the output layer. Of course, the output activation a_k^L of the $k^t h$ neuron depends only on the input weight z_j^L for the $j^t h$ neuron when $k = j$. And so $\frac{\partial a_k^L}{\partial z_j^L}$ vanishes when $k \neq j$. As a result we can simplify the previous equation to

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L}. \tag{9}$$

Recalling that $a_j^L = \sigma(z_j^L)$, the second term on the right can be written as $\sigma'(z_j^L)$, and the equation becomes

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L). \tag{10}$$

We can rewrite the equation in a matrix-based form, as

$$\delta^L = \nabla_a C \odot \sigma'(z^L). \tag{11}$$

Here, $\nabla_a C$ is defined to be a vector whose components are the partial derivatives $\frac{\partial C}{\partial a_j^L}$. We use $\odot$ to denote the elementwise product of the two vectors.

Next, we'll develop the equation for the error δ^l in terms of the error in the next layer, δ^{l+1}. To do this, we want to rewrite $\delta_j^l = \frac{\partial C}{\partial z_j^l}$ in terms of $\delta_k^{l+1} = \frac{\partial C}{\partial z_k^{l+1}}$. We can do this using the chain rule:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \tag{12}$$

$$= \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \tag{13}$$

$$= \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1}, \tag{14}$$

where in the last line we have interchanged the two terms on the right-hand side, and substituted the definition of δ_k^{l+1}. To evaluate the first term on the last line, note that

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1}. \tag{15}$$

Differentiating, we obtain

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l). \tag{16}$$

Substituting back into (14) we obtain

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l). \tag{17}$$

In a matrix-based form,

$$\delta^l = \left(\left(w^{l+1} \right)^T \delta^{l+1} \right) \odot \sigma'(z^l), \tag{18}$$

where $\left(w^{l+1}\right)^T$ is the transpose of the weight matrix w^{l+1} for the $(l+1)^{th}$ layer.

By combining (18) with (11) we can copmute the error δ^l for any layer in the network. We start by using (11) to compute δ^L, then apply Equation (18) to compute δ^{L-1}, then Equation (18) again to compute δ^{L-1}, and so on, all the way back through the network.

Now that we have the errors δ_j^l of all the layers of the network, we can compute the partial derivatives $\frac{\partial C}{\partial w_{jk}^l}$ and $\frac{\partial C}{\partial b_j^l}$ as a function of δ_j^l:

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1}, \tag{19}$$

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l. \tag{20}$$

For each iteration we use (4) and (5) and compute the new values of the parameters.

To summerize the backpropagation algorithm:

1) Perform a feedforward pass, computing the activations for layers 2,3, and so on up to the output layer L.
2) For each output unit j in layer L (the output layer), set

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L). \tag{21}$$

3) For layers $l = L-1, L-1, ..., 2$, for each node j in layer l, set

$$\delta_j^l = \sum_k w_{kj}^{l+1}\delta_k^{l+1}\sigma'(z_j^l). \tag{22}$$

4) Compute the desired partial derivatives, which are given as:

$$\frac{\partial C}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1}, \tag{23}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \tag{24}$$

5) Update the weights and biases of the network:

$$w_{jk}^l = w_{jk}^l - \alpha\frac{\partial C}{\partial w_{jk}^l}, \tag{25}$$

$$b_j^l = b_j^l - \alpha\frac{\partial C}{\partial b_j^l}, \tag{26}$$

REFERENCES

[1] Michael A. Nielsen. *Neural Networks and Deep Learning.* Determination Press, 2015.

[2] Andrew Ng, Jiquan Ngiam, Chuan Yu Foo, Yifan Mai, Caroline Suen. *UFLDL Stanford Tutorial.* http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial.