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 is based on the online book of Michael Nielsen [1].

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, regression, 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 is depicted in Fig. 1. The Neuron has 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 $\sigma(z)$ that produces a single output y. The output of the neuron y is determined by

$$y = \sigma(z),$$

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

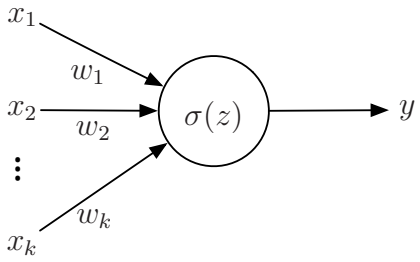

Figure 1. Scheme of a single neuron.

There are many different options for the choice of the activation function $\sigma(z)$. A few of them are given below and a depicted in Fig. 2.

- $\sigma(z) = \frac{1}{1+e^{-z}}$ - 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)

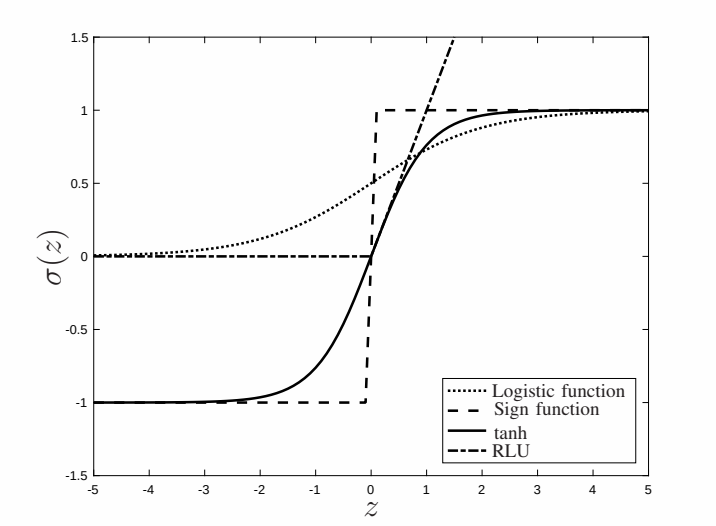

Figure 2. Plot of the activation functions mentioned above.

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*. Fig. 3 depicts a fully connected neural network with input layer, hidden layer and output layer. The reason its called fully connected since the neural in each layer are connected to all the neural in the next layer.

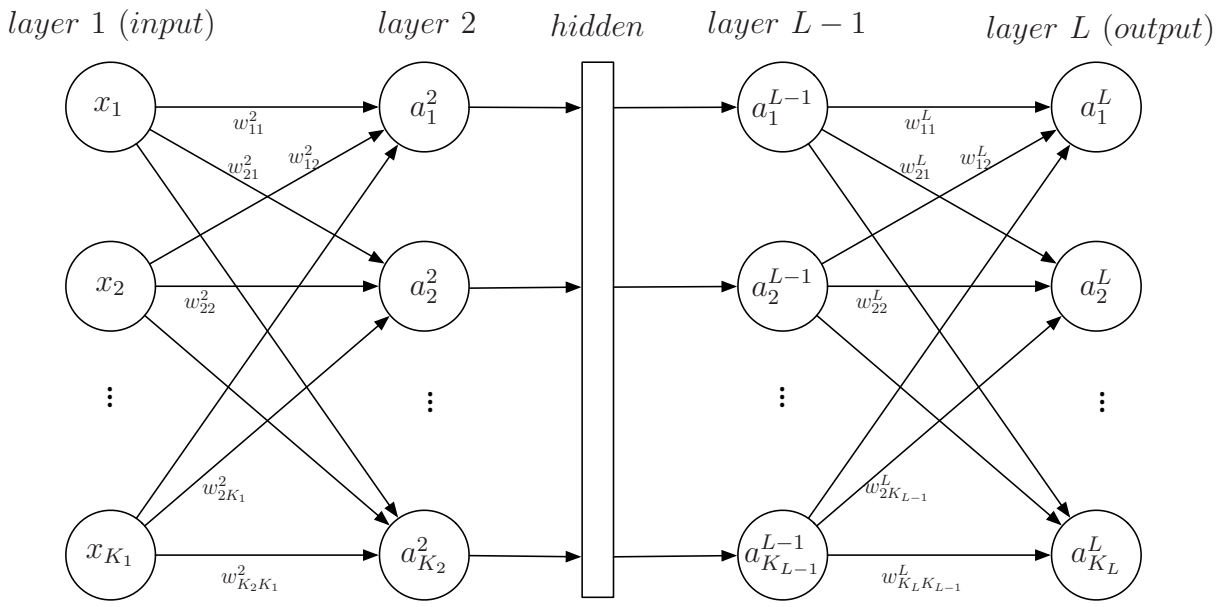

Figure 3. Structure of a general Neural Network

The parameters and variables that define a neural network are the following:

- w_{jk}^l - 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_j^l - 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.
- $z_j^l = \sum_{k=1}^{K_{l-1}} w_{jk}^l a_k^{l-1} + b_j^l$
- $a_j^l = \sigma(z_j^l)$

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.

- $z^l = \left[z_1^l, z_2^l, \ldots, z_{K_l}^l\right]^T$
- $b^l = \left[b_1^l, b_2^l, \ldots, b_{K_l}^l\right]^T$
- $w^l = \begin{bmatrix} w_{11}^l & w_{12}^l & \ldots & w_{1K_{l-1}}^l \\ w_{21}^l & w_{22}^l & \ldots & w_{2K_{l-1}}^l \\ \vdots & \vdots & \ddots & \vdots \\ w_{K_l1}^l & w_{K_l1}^l & \ldots & w_{K_lK_{l-1}}^l \end{bmatrix}$
- $a^l = \left[a_1^l, a_2^l, \ldots, a_{K_l}^l\right]^T$
- $z^l = w^l a^{l-1} + b^l$
- $a^l = \sigma(z^l)$
- $\sigma([x_1, \ldots, x_n]^T) = [\sigma(x_1), \ldots, \sigma(x_n)]^T$

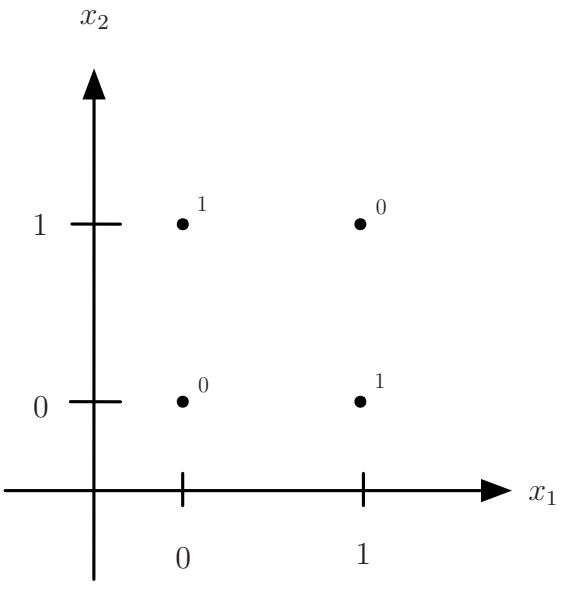

Figure 4. XOR function - illustration of output for binary inputs x_1, x_2

Example 1 (XOR function) One neuron is able to separate two sets only by a linear separation. Now consider the XOR function that we are all familiar with and it is depicted in Fig. 4. It is easy to see, that the XOR function cannot be approximated using a linear

function (there is no line that could separate the two groups of answers). But now we will show, that it is possible to approximate the XOR function using a Neural Network. We build a Network as depicted in Fig. 5 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. And the set the weights that we choose is given in Fig. 5.

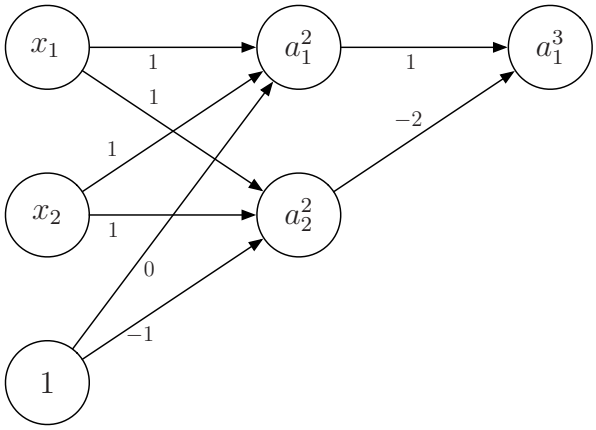

Figure 5. Example: XOR function Neural Network

We choose the activation function to be the RLU function. Let's take the input $[x_1, x_2] = [0, 0]$ and insert it to the network:

$$a_1^2 = max(0, 1 * x_1 + 1 * x_2 + 0 * 1) = 0$$

$$a_2^2 = max(0, 1 * x_1 + 1 * x_2 + -1 * 1) = 0$$

$$a_1^3 = max(0, 1 * a_1^2 - 2 * a_2^2) = 0$$

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

$\boldsymbol{x_1}$	$\boldsymbol{x_2}$	$\boldsymbol{a_1^2}$	$\boldsymbol{a_2^2}$	$\boldsymbol{a_1^3}$
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_1^3 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

A very important part of defining the Neural Network is the goal that it is designed to achieve. Hence, one need to define a cost function that quantify how close we're to achieving the goal. The cost function is a measure of how close is the output of the neural network to the desired label. For instance a mean square error function (or a quadratic cost function) is define as a cost function as follows:

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

where *x* are the input vectors and *y* are the corresponding labels. The input and the label are determined by the problem setting, hence are fixed. The parameters *w* and biases *b* are determined by the neural network, hence we can consider that the cost function is a function of the weights *w* and biases *b*.

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. It says that to find a local minimum of a function, one should 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 $\frac{\partial C}{\partial w_{jk}^l}$ and $\frac{\partial C}{\partial b_j^l}$. The main idea of the backpropagation is using the chain-rule of derivative. To compute the derivative, we first introduce an intermediate quantity, δ_j^l, which we call the error in the j^{th} neuron in the l^{th} layer. Backpropagation provide a procedure to compute the error δ_j^l, and then will relate δ_j^l to $\frac{\partial C}{\partial w_{jk}^l}$ and $\frac{\partial C}{\partial b_j^l}$.

Our goal is to minimize *C* as a function of w and b. To train our neural network, we initialize each parameter w_{jk}^l and each b_j^l 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:

$$w_{jk}^l = w_{jk}^l - \alpha \frac{\partial C}{\partial w_{jk}^l}, \tag{3}$$

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

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

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}, \tag{5}$$

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

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}. \tag{6}$$

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

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L}, \tag{7}$$

where the sum is over all neurons *k* in the output layer. Of course, the output activation a_k^L of the k^th neuron depends only on the input weight z_j^L for the j^th 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

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L}. \tag{8}$$

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

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

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

$$\delta^L = \nabla_a C \odot \sigma'(z^L). \tag{10}$$

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:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \tag{11}$$

$$= \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \tag{12}$$

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

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

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1}. \tag{14}$$

Differentiating, we obtain

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l). \tag{15}$$

Substituting back into (13) we obtain

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l). \tag{16}$$

In a matrix-based form,

$$\delta^l = \left(\left(w^{l+1}\right)^T \delta^{l+1} \right) \odot \sigma'(z^l), \tag{17}$$

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 (17) with (10) we can copmute the error δ^l for any layer in the network. We start by using (10) to compute δ^L, then apply Equation (17) to compute δ^{L-1}, then Equation (17) again to compute δ^{L-1}, and so on, all the way back through the network.

Now that we have the errors δ^l_j of all the layers of the network, we can compute the partial derivatives $\frac{\partial C}{\partial w^l_{jk}}$ and $\frac{\partial C}{\partial b^l_j}$ as a function of δ^l_j:

$$\frac{\partial C}{\partial w^l_{jk}} = \frac{\partial C}{\partial z^l_j}\frac{\partial z^l_j}{\partial w^l_{jk}} = \delta^l_j a^{l-1}_k, \tag{18}$$

$$\frac{\partial C}{\partial b^l_j} = \frac{\partial C}{\partial z^l_j}\frac{\partial z^l_j}{\partial b^l_j} = \delta^l_j. \tag{19}$$

For each iteration we use (3) and (4) 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

$$\delta^L_j = \frac{\partial C}{\partial a^L_j}\sigma'(z^L_j). \tag{20}$$

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

$$\delta^l_j = \sum_k w^{l+1}_{kj}\delta^{l+1}_k \sigma'(z^l_j). \tag{21}$$

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

$$\frac{\partial C}{\partial w^l_{jk}} = \delta^l_j a^{l-1}_k, \tag{22}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j. \tag{23}$$

5) Update the weights and biases of the network:

$$w^l_{jk} = w^l_{jk} - \alpha\frac{\partial C}{\partial w^l_{jk}}, \tag{24}$$

$$b^l_j = b^l_j - \alpha\frac{\partial C}{\partial b^l_j}, \tag{25}$$

REFERENCES

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