Neural Networks

Matteo Manca
matteomanca@gmail.com

Contents

The following notes represent a personal interpretation of the Neural Networks classes of the Coursera machine learning course (https://www.coursera.org/learn/machine-learning/) presented by Professor Andrew Ng .

0.1 Artificial Neural Networks

Neural Network is a learning algorithm based on how human brain works and it is very useful to approach different machine learning problems. In order to better understand why we need another learning algorithm although we already study linear regression and logistic regression, let's see a an example of machine learning problem where ne need complex non-linear hypotheses.

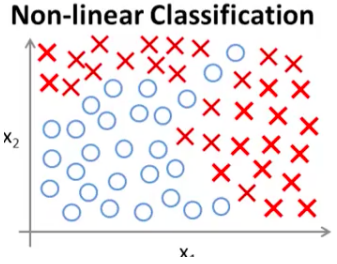


Figure 1: Example of a non-linear classification problem

If we want to apply logistic regression to this problem we have to use a lot of non linear features like in equation 1 **sigmoid function** (see logistic regression for more explanations about that):

Equation 1 Sigmoid function with many terms

$$g(\Theta_0 + \Theta_1 x_1 + \Theta_2 x_2 + \Theta_3 x_1 x_2 + \Theta_4 x_1^2 x_2 + \dots) \tag{1}$$

So, to get a good classification we need to include a high number of polynomial terms. Logistic regression method works well when we have only two input features, x_1 and x_2 because in that case we can add all polynomial terms of x_1 and x_2. In many machine learning problems we will have a lot more feature than just two.

Let's think at the price house prediction problem we saw in previous lectures and suppose to have a classification problem rather than a regression problem. For instance we can imagine to have different features of a house and we want to predict what are the odds that the house will be hold within the next six months:
x_1 = size
x_2 = # bedrooms
$\vdots$

$x_{100} = \dots$

In a problem like that reported above, with $n = 100$ features, if we want to include the polynomial terms there would be a very huge number of terms. For instance, if we think to include even just the quadratic terms (second order terms, that are the terms obtained as the product of two of these terms - for instance $x_1 x_2$) in a problem with $n = 100$ we obtain about 5000 features.
Asymptotically the number of quadratic terms grows as order of $O(n^2)$, and it is closer to $n^2/2$, in fact $100^2/2 = 5000$.

We demonstrated that also considering just the quadratic terms the number of features is very high, consequently there is a high risk of **overfitting** (see previous lectures for the overfitting problem) the training set and the computation would be too much expensive. One idea to avoid this high number of features could be to consider just a subset of the terms, for instance $x_1^2, x_2^2, \dots, x_{100}^2$. In this way the number of features would be smaller but these features would not be enough to represent the dataset represented in figure 1. In fact if we include only the quadratic features with the original features $x_1, x_2, \dots, x_n$ we can obtain just lines of the ellipse (Figure 2) but we cannot fit more complex datasets.

Figure 2: Example of a quadratic function

If we think to include the third order (or even higher) polynomial terms the number of features will increase again so we are going to face the same problem described above. Let's see an example to better understand how the number of features n can be very high.
Consider an example of computer vision problem where we want to train a classifier able to examine an image and tell us if the image is a car or not (Remember that an image is a matrix of pixels intensity values). In our example we have a labeled training set with few label example of cars and a few label of other objects (not cars) , see figure 3.

Figure 3: Computer Vision example dataset

We can imagine to plot a object of the dataset. Let's pick a couple of pixel locations (pixel 1 and pixel 2) in our image and plot the object depending on the intensity of pixel 1 and pixel 2 (see figure 4)

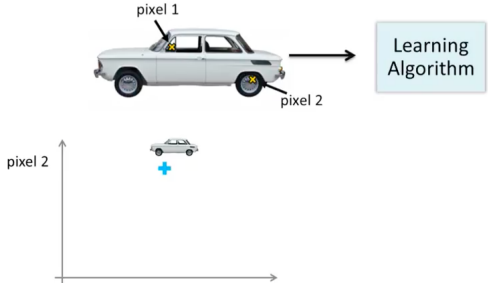


Figure 4: Objects plot

Plotting all objects of the dataset we can notice that cars and not-cars end up lying in different regions of the plot (see figure 5) so we need a non-linear hypothesis to try to separate the two classes {cars, not-cars}.

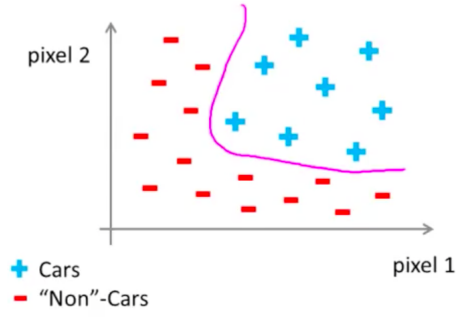


Figure 5: Plot with all objects of the dataset

in the above example we consider just two pixel, but if we use a 50×50 pixel image the dimension of the features space is $n = 50 * 50 = 2500$, so we have a vector of 2500 pixel intensities. If we try to learn a non-linear hypothesis by including all the quadratic terms, we obtain about 3 million features and it would be too large.

So,we demonstrated that logistic regression with the addition of quadratic (or cubic) features is not a good idea to learn a complex non-linear hypothesis when n is large because we end up with too many features.

0.1.1 Model Representation

Neural networks were developed simulating neurons or network of neurons in the human brain. Let's look how a single neuron looks like:

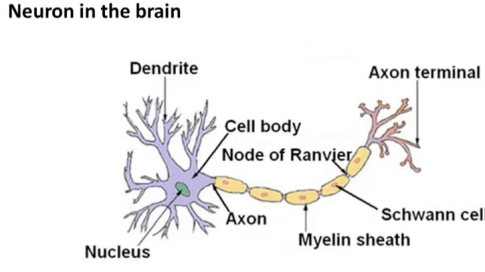


Figure 6: A single neuron

Neurons are cells in the brain; some things to draw attention to are the **Cell body**, a number of input wires called **dendrites** and a number of

output wires called **axons**. We can think to a neuron as a computational unit that receives inputs through dendrites, does some computations and sends an output signal via its axons to other nodes. Neurons comunicate by means of little pulse of electricity

Neural Model: Logistic Unit

In an Artificial Neural Network implemented in a computer, we use a very simple model of what a neuron does. We are going to model a neuron as a simple logistic unit (figure 7):

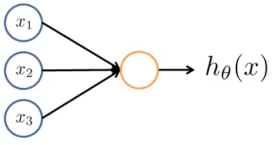


Figure 7: A simple logistic unit

where $h_{\theta(x)}$ represent the computation of:

Equation 2 Sigmoid function

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

and θ and x are the parameter vectors:

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix} \quad x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{3}$$

Note that we add x_0 that is the **bias unit** and we set $x_0 = 1$.
In neural networks the sigmoid function (or logistic function) is also called **Activation function** and the parameters θ are called **weights** of the model. Till now we see just a single neuron, but a neural network is a proof of *different neurons models that interact each other* .

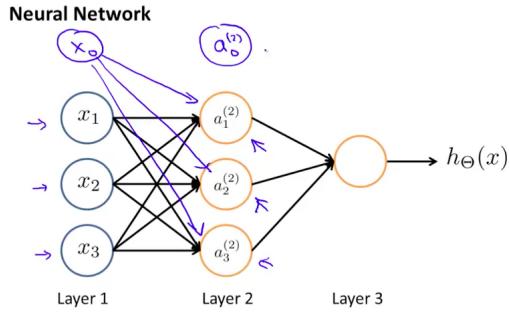


Figure 8: An example of neural network

In figure 8 we have a neural network composed by three neurons a_1^2, a_2^2, a_3^2 more the bias unit a_0^2; moreover in the above neural network we have multiple layers:

- **input layer** (layer 1): in this layer we input the features x_1, x_2 and x_3 (of our training set);
- **hidden layer** (layer 2): it contains values that we don't observe in the training set;
- **final layer** (layer 3): it has a neuron / units (it can have more units) that output the final value computed by the hypothesis.

In order to explain a specific computation of a neural network, in the following we report a some definitions:

- $a_i^{(j)}$: **Activation** of unit i in layer j (i.e., the value that is computed by and that is the output of a specify unit);
- $\Theta^{(j)}$: **matrix of weights** that controls the function mapping from layer j to layer $j+1$.

The computation of the diagram in figure 8 is reported in equation 4.

Equation 4 Neural Network computation

$$\begin{aligned} a_1^{(2)} &= g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3) \\ a_2^{(2)} &= g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3) \\ a_3^{(2)} &= g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3) \\ h_\Theta(x) = a_1^{(3)} &= g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} x_3^{(2)}) \end{aligned} \tag{4}$$

Summarizing, in our neural network (figure 8) we have:

- 3 input units (without consider the bias unit);
- 3 hidden units (without consider the bias unit);

$\Theta^{(1)}$ is 3×4 matrix, so $\Theta^{(1)} \in \mathbb{R}^{3\times 4}$.

Definition 1 *If a network has S_j units in layer j, S_{j+1} units in layer $j+1$, then $\Theta^{(j)}$ is a $S_{j+1} \times S_j + 1$ matrix:*

$$\Theta^{(j)} \in \mathbb{R}^{S_{j+1}\times S_j+1} (5)$$

0.1.2 Feedforward propagation

In this section we'll see how to carry out the computation of the neural network (in figure 9) efficiently using a **vectorized implementation** and how a neural network can help to learn complex non-linear hypotheses.

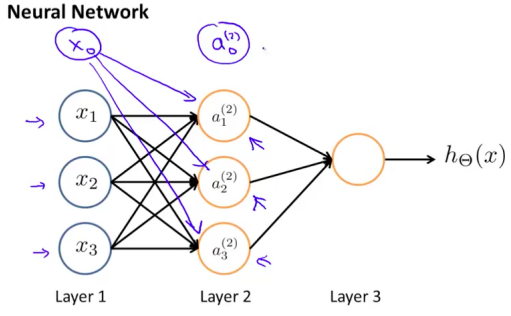


Figure 9: An example of neural network

Equation 6 Neural Network computation

$$\begin{aligned} a_1^{(2)} &= g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3) \\ a_2^{(2)} &= g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3) \\ a_3^{(2)} &= g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3) \\ h_\Theta(x) = a_1^{(3)} &= g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} x_3^{(2)}) \end{aligned} \quad (6)$$

Starting from equation 6 we define some new notation:

$$
\begin{aligned}
a_1^{(2)} = g(z_1^{(2)}) &\implies z_1^{(2)} = \Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3 \\
a_2^{(2)} = g(z_2^{(2)}) &\implies z_2^{(2)} = \Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3 \\
a_3^{(2)} = g(z_3^{(2)}) &\implies z_3^{(2)} = \Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3
\end{aligned} \tag{7}
$$

Observing z_1, z_2 and z_3, can be noticed that this block corresponds to the matrix-vector multiplication $\Theta^{(1)} x$.

Let be x and z be:

$$
\left(x = \begin{bmatrix} x_0 = 1 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix} \right) \tag{8}
$$

we can vectorize the computation as follow:

Equation 9 Vectorized computation

$$
\begin{aligned}
z^{(2)} &= \Theta^{(1)} x \\
a^{(2)} &= g(z^{(2)})
\end{aligned} \tag{9}
$$

where $z^{(2)} \in \mathbb{R}^3$ and $a^{(2)} \in \mathbb{R}^3$. In order to use a more clear notation we also define:

$$
\begin{aligned}
a^{(1)} &= x \quad \text{so:} \\
z^{(2)} &= \Theta^{(1)} a^{(1)}
\end{aligned} \tag{10}
$$

In equation 10 we define $a^{(1)}$ to be the activation function in the input layer .

Since now we have seen how to compute $a_1^{(2)}$, $a_2^{(2)}$ and $a_3^{(2)}$; now we need to add in the hidden layer of our model the bias unit $a_0^{(2)} = 1$; in this way $a^{(2)} \in \mathbb{R}^4$, so it become a $4 - d$ vector.

To compute the output value of the hypothesis $h_\Theta(x)$, we simply need to compute $z^{(3)}$:

Equation 11 Activation function of the output layer. $a^{(3)} \in \mathbb{R}$

$$
h_\Theta(x) = a^{(3)} = g(z^{(3)}) \tag{11}
$$

The computation of $h_\Theta(x)$ is called **forward propagation** because it starts with the activation of the input units, it propagates that to the hidden layer computing the activation function of this layer and the propagation follows with the activation of the output layer (in Matlab to make that computation we use a vector wise implementation procedure).

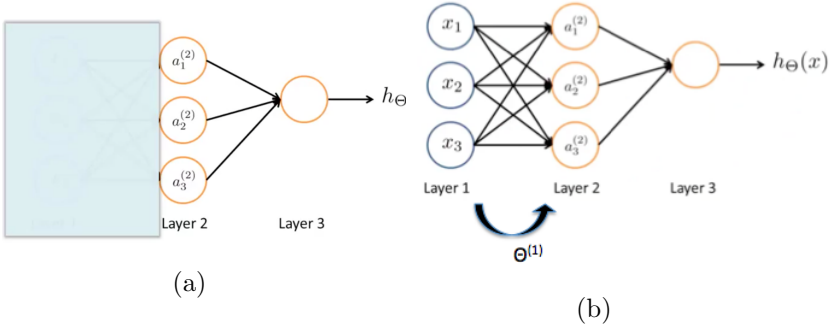


Figure 10: An example of neural network

If we look at figure 10-a it looks like a logistic regression unit able to predict $h_\Theta(x)$, where:

Equation 12 Hypothesis function

$$h_\Theta(x) = g(\Theta_0 a_0^{(2)} + \Theta_1 a_1^{(2)} + \Theta_2 a_2^{(2)} + \Theta_3 a_3^{(2)}) \tag{12}$$

If in equation 12 we do not consider the superscript (2), it looks like the standard logistic regression equation except that now we are using Θ instead of θ; in this case the input features of the logistic regression are the values computed by the hidden layer. In fact, the represented neural network is doing a logistic regression computation using $a_1^{(2)}$, $a_2^{(2)}$, $a_3^{(2)}$ rather than use the original features x_1, x_2, x_3. The interesting thing is that the features $a_1^{(2)}$, $a_2^{(2)}$, $a_3^{(2)}$ are themselves learned as functions of the input x_1, x_2, x_3. Concretely the function mapping from layer 1 to layer 2 is determined by other set of parameters $\Theta^{(1)}$. So, that shows how a neural network instead of being constrained to feed the features x_1, x_2, x_3 to logistic regression, it gets to learn its own features $a_1^{(2)}$, $a_2^{(2)}$, $a_3^{(2)}$ to feed into logistic regression and depending on what parameters it chooses for Θ_1 it can learn some pretty interesting and complex features and therefore we can obtain a better hypothesis than if we were constrained to use the raw features x_1, x_2, x_3.

This algorithm has the flexibility to try to learn whatever features at once using $a_1^{(2)}$, $a_2^{(2)}$, $a_3^{(2)}$ in order to feed the output unit $h_\Theta(x)$. This example is described at high level, so to better understand the intuition beyond the neural networks, in the following we'll see some examples.

Definition 2 Architecture of a Neural Network: *how the different neurons of a network are connected each others.*

0.1.3 Examples and Intuitions

In the following some examples how neural networks can compute complex non-linear hypotheses are reported. Consider the following example with input features x_1 and x_2 that are binary values $(0, 1)$.

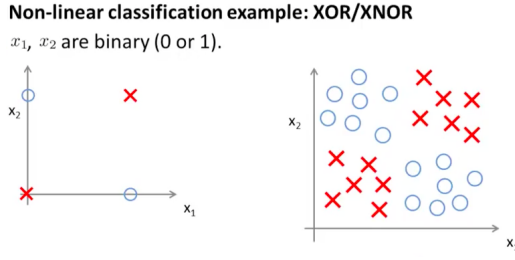


Figure 11: Non-linear Classification example: XOR/XNOR

In figure 11, on the left we have a simplified version of a more complex (on the right) learning problem. We would like to learn a non-linear function (decision boundary) to separate positive and negative examples. Looking at the simplified version we can consider positive examples ($y = 1$, the X symbol) if both x_1 and x_2 are equal to zero or if $x_1 = x_2 = 1$. We want to figure out if we can get a neural network to fit the training set of the example. Before to build the neural network that fit the above example (that is a XNOR function), let's start building a neural network that fits the AND logic function at first and then some other logic functions.

Example1: AND Neural Network

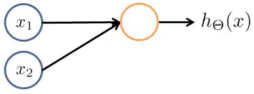


Figure 12: Non-linear Classification example: AND

In figure 12 we have a unit network to compute the **logical AND**:

- $x_1, x_2 \in \{0, 1\}$
- $y = x_1 \wedge x_2$

Let's add the bias unit and assign some values to the weights (parameters) of the network (figure 13):

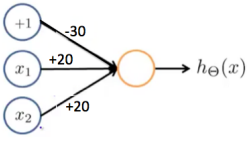


Figure 13: Non-linear Classification example with bias unit and weights: AND

Given the neural network represented in figure 13, the hypothesis $h_\Theta(x)$ will be:

Equation 13 AND Hypothesis function

$$h_\Theta(x) = g(-30 + 20x_1 + 20x_2) \tag{13}$$

where:

- $\Theta_{10}^{(1)} = -30$
- $\Theta_{11}^{(1)} = +20$
- $\Theta_{12}^{(1)} = +20$

Taking into account the sigmoid function $g(z)$ represented in figure 14, look at the four possible input values for x_1 and x_2

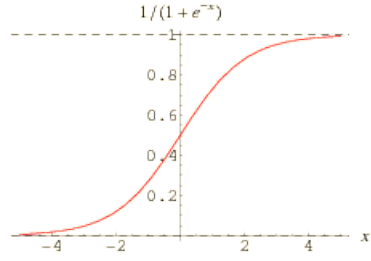


Figure 14: Sigmoid function $g(z)$

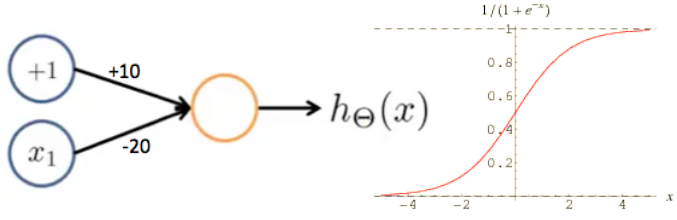
Table 1: Table related to neural network in figure 13 (Logic AND)

x_1	x_2	$h_\Theta(x) = g(-30 + 20x_1 + 20x_2)$
0	0	$g(-30) \approx 0$
0	1	$g(-30 + 20) \approx 0$
1	0	$g(-30 + 20) \approx 0$
1	1	$g(-30 + 20 + 20) \approx 1$

Example2: NOT Neural Network

Equation 14 NOT Hypothesis function

$$\begin{aligned} x_1 &\in \{0,1\} \\ h_\Theta(x) &= g(10 - 20x_1) \end{aligned} \tag{14}$$



(a) (b)

Figure 15: Non-linear Classification example: NOT(a) and Sigmoid function $g(z)$ (b)

Table 2: Table related to neural network in figure 15-a (Logic NOT)

x_1	$h_\Theta(x) = g(10 - 20x_1)$
0	$g(10) \approx 1$
1	$g(+10 - 20) \approx 0$

Example3: (NOT) AND (NOT) Neural Network

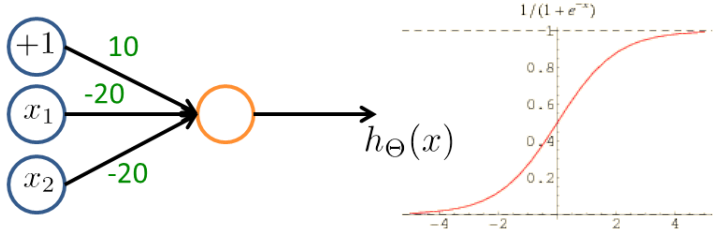


Figure 16: (NOT) AND (NOT) Neural Network example

Table 3: Table related to neural network in figure 17 ((NOT x_1) AND (NOT x_2))

x_1	x_2	$h_\Theta(x) = g(10 - 20x_1 - 20x_2)$
0	0	$g(10) \approx 1$
0	1	$g(+10 - 20) \approx 0$
1	0	$g(+10 - 20) \approx 0$
1	1	$g(+10 - 20 - 20) \approx 0$

Example 4: OR Neural Network

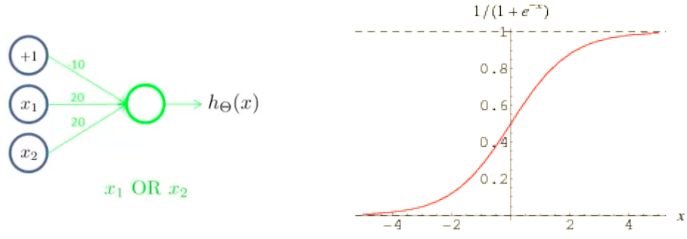


Figure 17: OR Neural Network example

Table 4: Table related to neural network in figure 17 x_1 OR x_2

x_1	x_2	$h_\Theta(x) = g(-10 + 20x_1 + 20x_2)$
0	0	$g(-10) \approx 0$
0	1	$g(-10 + 20) \approx 1$
1	0	$g(-10 + 20) \approx 1$
1	1	$g(-10 + 20 + 20) \approx 1$

XNOR Neural Network

Now we want to put together the neural networks we have seen previously in order to compute x_1 XNOR x_2. So, starting from the training example in figure 18, we need a decision boundary in order to separate positive and negative examples. In figure 20 the related neural network is represented.

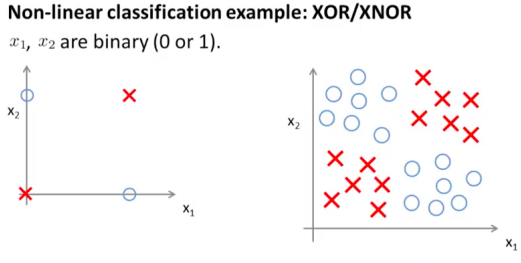


Figure 18: Non-linear Classification example: XOR/XNOR

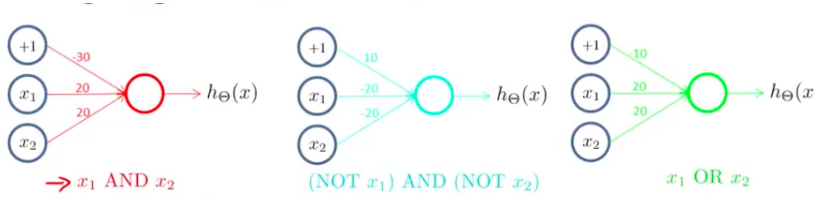
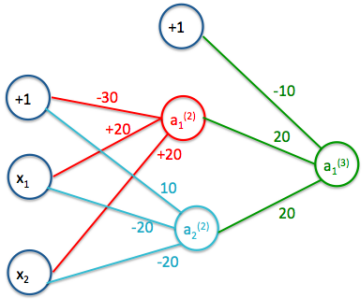


Figure 19: Non-linear Classification examples



x1	x2	a1(2)	a2(2)	h(x)
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1

Figure 20: Non-linear Classification example: XNOR

So, we can notice that $h_\Theta(x) = 1$ if $x_1 = x_2 = 0$ or if $x_1 = x_2 = 1$ (see figure 20); in this way we found a non-linear decision boundary that computes the nor function.
The general intuition is that in the input layer we have the raw inputs, then a hidden layer is added in order to compute some slightly more complex functions (and, or, etc.) of the inputs and, finally, the output layer computes an even more complex non-linear function.

In general we can state that the deeper the layer is, more complicated the computed non-linear function is.

0.1.4 Multiclass classification

This section explains how to use neural networks to do multiples classification by means of an extension of the one-vs-all method.

Let's consider a computer vision example where we want to recognize four categories of objects (given an image): pedestrian, car, motorcycle, truck. We would build a a neural network with four output units, so that

the output is a vector of four numbers ($h_\Theta(x) \in \mathbb{R}^4$); so, in this example the network has four output units where:

- output unit 1: predict if the image is a pedestrian or not;
- output unit 2: predict if the image is a car or not;
- output unit 3: predict if the image is a motorcycle or not;
- output unit 4: predict if the image is a truck or not;

The four output units could be seen as four logistic regression classifier.

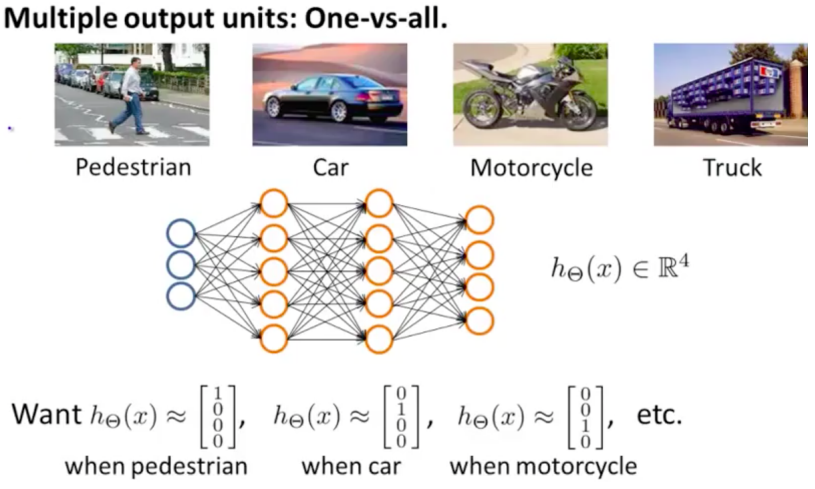


Figure 21: Multi-class classification example: one-vs-all

Exercise

Suppose you have a multi-class classification problem with 10 classes. Your neural network has 3 layers, and the hidden layer (layer 2) has 5 units. Using the one-vs-all method how many elements does $\Theta^{(2)}$ has?

Solution

In general :
$\Theta^{(j)} \in (S_{j+1}) \times (S_j + 1)$;
where S_j is the number of units in layer j.

We have:

- layer 2 $\Rightarrow$ 5 units
- output layer (layer 3) $\Rightarrow$ 10 units

So,
$\Theta^{(2)} \in (S_3) \times (S_2 + 1) = 10 \times 6$

0.1.5 Neural Networks: learning. Cost function

This section studies how to learn the parameters of a neural network given a training set. Suppose to have the following neural network:

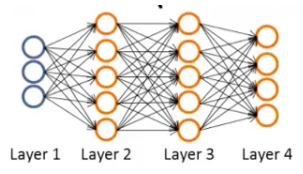


Figure 22: Multi-class classification neural network example

and that:

- $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$ is the training set;

- L is the total number of layer in the considered network ($L = 4$ in the above example);
- S_l is the number of units in layer l.

In the example two types of classification problems are considered:

- *Binary classification*: $y \in \{0, 1\}$
- *Multi-class classification*(K-classes):
  - $y \in \mathbb{R}^k$;
  - k output units;

The cost function for the above neural network can be defined as a generalization of the logistic regression cost function:

Equation 15 Logistic Regression Cost function.

$$J(\Theta) = -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} \log h_\Theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\Theta(x^{(i)}))] + \frac{\lambda}{2m} \sum_{j=1}^{n} \Theta_j^2 \tag{15}$$

Note that the regularization is not applied to Θ_0.

In the case of neural networks the cost function is a generalization of the logistic regression one (equation 15), where instead of having just one logistic regression output unit, there may be k output units, so the cost function is:

Equation 16 Neural Networks Cost function.

$$J(\Theta) = -\frac{1}{m}[\sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)] + \frac{\lambda}{2m} \sum_{l=1}^{L} \sum_{i=1}^{S_l} \sum_{j=1}^{S_{l+1}} (\Theta_{ij}^{(l)})^2 \tag{16}$$

Where :

- $h_\Theta(x) \in \mathbb{R}^K$;
- $(h_\Theta(x))_i = i^{th}$ output.

Observing the regularization term $\frac{\lambda}{2m}\sum_{l=1}^{L}\sum_{i=1}^{S_l}\sum_{j=1}^{S_{l+1}}(\Theta_i j^{(l)})^2$ can be noticed that the sum is built by summing over $\Theta_{ij}^{(l)}$ for all i, j and l except that for the terms that correspond to the bias values; Concretely, the sum does not consider the case of $i = 0$. The reason why that terms are not considered is that they would be multiplied with the bias units.

0.1.6 Backpropagation algorithm

Now, the **backpropagation algorithm** that aims to minimize the cost function is presented. In order to minimize the cost function through the **gradient descent** or some other **optimization algorithms**, we need to implement the code that take the parameter Θ as input and compute $J(\Theta)$ and the partial derivatives terms (remember that the gradient descent is an algorithm that uses the partial derivatives in order to find the local minimum of a function):

Equation 17 Partial Derivatives terms.

$$\frac{\partial}{\partial\Theta_{ij}^{(l)}}J(\Theta) \tag{17}$$

Now, let's focus on the computation of the partial derivatives. Consider the case where only one training example (x, y) is available. First of all we have to apply the forward propagation in order to compute what the hypothesis outputs given an input feature x:

Equation 18 Forward propagation application - vectorized implementation

$$\begin{aligned} a^{(1)} &= x \\ z^{(2)} &= \Theta^{(1)}a^{(1)} \\ a^{(2)} &= g(z^{(2)}) \text{ add } a_0^{(2)} \\ z^{(3)} &= \Theta^{(2)}a^{(2)} \\ a^{(3)} &= g(z^{(3)}) \text{ add } a_0^{(3)} \\ z^{(4)} &= \Theta^{(3)}a^{(3)} \\ a^{(4)} &= g(z^{(4)}) = h_\Theta(x) \end{aligned} \tag{18}$$

The vectorized implementation of the forward propagation algorithm allows us to compute all **activation values** for all neurons of the neural network. In order to compute the derivatives we use the **Backpropagation algorithm**.

Gradient computation: Backpropagation algorithm

Before to see the backpropagation algorithm, let's define δ:

Definition 3 $\delta_j^{(l)}$: *Error of node j in layer l (it can be defined as the difference between the hypothesis output $h_\Theta(x)$ and the real output value y)*

The backpropagation algorithm is based on the intuition that for each node j of the neural network, the value $\delta_j^{(l)}$, that represents the error of the node j in layer l is computed.

Remember that :
$a_j^{(j)}$ is the activation of unit j in layer l.

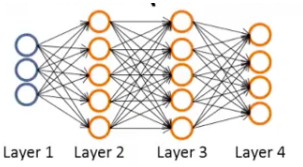


Figure 23: Multi-class classification neural network example

Consider the neural network in figure 23; we have four layers ($L = 4$) and for each output unit we have to compute: $\delta_j^4 = a_j^{(4)} - y_j$ where $a_j^{(4)} = (h_\Theta(x))_j$. If we think of δ, a and y as vectors, we can use the vectorized implementation:

Equation 19 Error computation with vectorized implementation

$$\delta^{(4)} = a^{(4)} - y \qquad \delta^4, a^{(4)}, y \in \mathbb{R}^L \tag{19}$$

where L is the number of output units.

The next step is the computation of the δ terms for the earlier layers in the network:

Equation 20 Error computation with vectorized implementation for the other layers of the network

$$\begin{aligned}\delta^{(3)} &= (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)}) \\ \delta^{(2)} &= (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})\end{aligned} \tag{20}$$

where $g'(z^{(l)})$ is the derivative of the activation function g evaluated at the input value given by $z^{(l)}$, and it is computes as follows:

Equation 21 Error computation with vectorized implementation for the other layers of the network

$$g'(z^{(l)}) = a^{(l)} .* (1 - a^{(l)}) \tag{21}$$

where:

- $a^{(l)}$ is a vector of activations;
- 1 is a vector of 1_s

Definition 4 *The symbol ".* " is the notation for the element wise multiplication, also called Hadamard product. It is a binary operation that takes two matrices of the same dimensions, and produces another matrix where each element ij is the product of elements ij of the original two matrices.*

Note that there is not $\delta^{(1)}$ term because the first layer corresponds to the input layer where the values are the features observed in the training set, so there isn't any prediction error associated to them.

It is possible to proof that the partial derivative terms is given by

Equation 22 Partial Derivatives terms computation (case of one training examples)

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(j)} \delta_i^{(l+1)} \tag{22}$$

Till now we considered the case of a single training example; in the following we will put all together to see how to implement the **backpropagation algorithm** to compute the derivatives with respect to the parameters and for those cases characterized by a large training set.

Suppose to have a training set of m examples:

$$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$$

we can implement the backpropagation algorithm as follows:

Algorithm 1 Backpropagation algorithm

Set $\Delta_{ij}^{(l)} = 0$ for all i, j, l;
for $i = 1 \dots m$ **do**
    Set $a^{(1)} = x^{(i)}$
    Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \dots, L$
    Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$
    Compute $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$
    $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$
end for
$D_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)} + \frac{\lambda \Theta_{ij}^{(l)}}{m}$ if $j \neq 0$
$D_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)}$ if $j = 0$

Note that $j = 0$ corresponds to the bias term, that is why we are omitting the regularization term. The formal proof is quite complicate (and we omit that), but we can state that:

Equation 23 Partial Derivatives terms computation (case of multiple training examples)

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} \tag{23}$$

We can use equation 23 with either gradient descent or in one of the advanced optimization algorithms.
Now we are going to analyze the mechanical steps of the backpropagation algorithm, but to better understand this algorithm, let's take a look at what forward propagation does (see figure 24), particularly we take into account the computation of the $z_1^{(3)}$ activation.

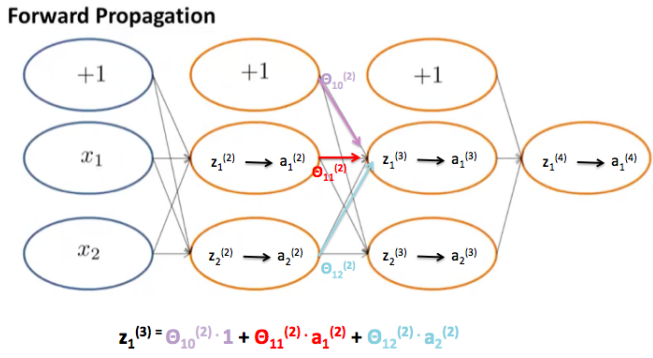


Figure 24: Forward propagation computation

What backpropagation does is very similar to what we have seen for forward propagation, except that instead of perform the computation flowing from left to the right, it flows from the right to the left. In order to understand the backpropagation algorithm, let's observe the cost function for the case where we have just one output unit (equation 24):

Equation 24 Neural Networks Cost function.

$$J(\Theta) = -\frac{1}{m}[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)] + \frac{\lambda}{2m}\sum_{l=1}^{L}\sum_{i=1}^{S_l}\sum_{j=1}^{S_{l+1}}(\Theta_{ij}^{(l)})^2 \tag{24}$$

If we focus on the case of a single training example $(x^{(i)}, y^{(i)})$, one output unit and ignore the regularization term $(\lambda = 0)$, we obtain the following cost function:

Equation 25 Neural Networks Cost function, one single example, one out unit and no regularization

$$cost(i) = y^{(i)} \log h_\Theta(x^{(i)}) + (1 - y^{(i)}) \log h_\Theta(x^{(i)}) \tag{25}$$

$cost(i)$ is the cost associated with the i^{th} example and we can think to that as: $cost(i) \simeq (h_\Theta(x^{(i)}) - y^{(i)})^2$, so it plays a role similar to the square error.

As in the case of logistic regression, in neural networks we use a slightly more complicated function using the logarithm. So, the cost function $cost(i)$ measures how well the neural network is doing on correctly predicting the example i.

An intuition is that the **backpropagation algorithm** computes the error $\delta_j^{(l)}$ that is the error of the activation value of the unit $a_j^{(l)}$ (unit j of the layer l). More formally:

Equation 26 Error of the activation value of unit j in layer l

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} cost(i) \qquad \text{for } j >= 0 \tag{26}$$

where:

Equation 27 Cost function of the activation unit

$$cost(i) = y^{(i)} \log h_\Theta(x^{(i)}) + (1 - y^{(i)}) \log h_\Theta(x^{(i)}) \tag{27}$$

In the following, more details about what the backpropagation algorithm does are given:

- For the output layer the error is given by $\delta_1^{(4)} = y^{(i)} - a_1^{(4)}$, which is the error measured as the difference between the actual value $y^{(i)}$ and the predicted value $a_1^{(4)}$.
- Next, the computed value $\delta_1^{(4)}$ is propagated backward in previous layer and the delta terms for the previous layer ($\delta_1^{(3)}$,$\delta_2^{(3)}$) are computed.
- The algorithm follows by propagating backward in the previous layers and computing $\delta_1^{(2)}$,$\delta_2^{(2)}$.

The backpropagation computation is like running the forward propagation algorithm, but doing it backwards.

0.1.7 Neural networks: implementation notes

In this section we will see some Matlab implementation details about **unrolling** the parameters from matrices into vectors, in order to use the advances optimization routines.

In neural networks, parameters are no longer vectors (as in logistic regression), but are matrices:

Neural Network L=4:

- $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ - Matrices (Theta1, Theta2, Theta3);
- $D^{(1)}, D^{(2)}, D^{(3)}$ - Matrices (D1, D2, D3);

We want to **unroll** these matrices into vectors so that they end up being in a format suitable for passing as parameters in the implementation of the cost function:

```
function [jVal, gradient] = costFunction(theta)
...
optTheta = fminunc(@costFunction, initialTheta,
    options)
```

Note that $theta \in \mathbb{R}^{n+1}$ and $initialTheta \in \mathbb{R}^{n+1}$ are vectors.

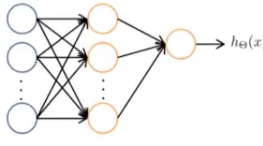


Figure 25: Forward propagation computation

Concretely, if we have a neural network with a input layer that has 10 units, one hidden layer with 10 units and one output layer with 1 unit (like that represented in figure 25), we have the following parameters:

- $S_1 = 10, S_2 = 10, S_3 = 1$
- $\Theta^{(1)} \in \mathbb{R}^{10\times 11}, \Theta^{(2)} \in \mathbb{R}^{10\times 11}, \Theta^{(3)} \in \mathbb{R}^{1\times 11}$
- $D^{(1)} \in \mathbb{R}^{10\times 11}, D^{(2)} \in \mathbb{R}^{10\times 11}, D^{(3)} \in \mathbb{R}^{1\times 11}$

In Matlab, in order to obtain the vectors starting from the matrices we have to run the following code:

```
thetaVec = [Theta1(:) ; Theta2(:) ; Theta3(:)];
DVec = [D1(:) ; D2(:) ; D3(:)];
```

The above code will unroll all the element of the matrices **Theta1, Theta2, Theta3** in a big long vector called **thetaVec**. In the same way we created the vector **DVec**.

If we want to go back from the vector representation, to the original matrix representation we have to run the following code:

```
Theta1 = reshape(thetaVec(1:110),10,11)
Theta2 = reshape(thetaVec(111:220),10,11)
Theta1 = reshape(thetaVec(221:231),1,11)
```

In the following all steps needed to implement a learning algorithm are summarized:

Algorithm 2 Learning algorithm

1: Suppose to have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$;
2: Unroll $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ to get **initialTheta** to pass to:

```
fminunc(@costFunction, initialTheta, options)
```

3: the other thing to do is the implementation of the **cost function**:

```
function [jVal, gradient] = costFunction(thetaVec)
```

4: from **thetaVec**, get $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ ▷ Using the **reshape function**
5: Use **forward prop** / **back prop** to compute $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\Theta)$.
6: unroll $D^{(1)}, D^{(2)}, D^{(3)}$ to get **gradientVec**.

Summarizing:

- parameters in **matrix representation**: more convenient to do back-propagation and forward propagation;
- parameters in **vector representation**: more convenient for the advanced optimization algorithms.

0.1.8 Gradient Checking

Sometimes we could implement a back propagation algorithm that led to a decreasing of the cost function $J(\Theta)$ but that contains some errors. Given the decreasing of the cost function, it could be difficult to see the error, so we end up to implement a neural network that contain a high level of error.

In order to avoid the just described problem we can use the **gradient checking** algorithm. In fact, once we implement the numerical gradient checking we are able to verify if our back propagation implementation is really computing the derivatives of the cost function $J(\Theta)$.

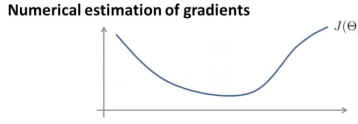


Figure 26: Cost function example

Suppose to have a function $J(\Theta)$ (figure 26) and some values Θ (assume Θ is a real number $\Theta \in \mathbb{R}$) and we want to estimate the derivatives of this function in the Θ point (remember that the partial derivative in a given point is equal to the slop of the tangent line in that point).

Now we are going to numerically approximate The derivative by:

- computing $\Theta + \varepsilon$;
- computing $\Theta - \varepsilon$;
- Connecting $\Theta + \varepsilon$ and $\Theta - \varepsilon$ by a straight line and using the slope of this straight-line as approximation of the derivative (see figure 27).

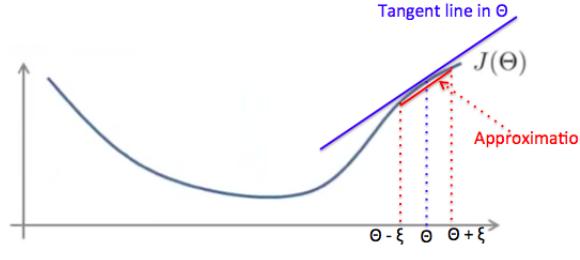


Figure 27: Gradient checking approximation

Looking at figure 27, the true derivative is the slope of the tangent line in point Θ (the blue line), but the line that connects the points $(\Theta - \varepsilon, \Theta + \varepsilon)$ (the red line) seem to be a very good approximation.

Mathematically, the slope of the "approximation" line is given by the vertical height b, divided by the horizontal width a(see figure 28).

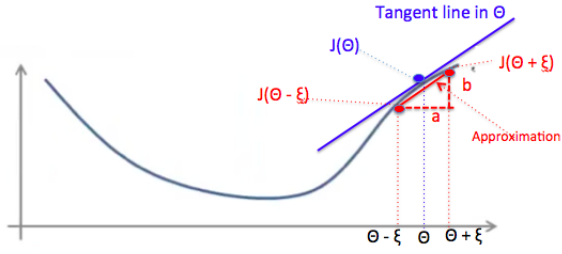


Figure 28: Gradient checking approximation computation

Looking at figure 28 we can define equation 28:

Equation 28 Gradient checking approximation computation

$$\begin{aligned} a &= 2\varepsilon \\ b &= J(\Theta+\varepsilon) - J(\Theta-\varepsilon) \\ \frac{\partial}{\partial\Theta}J(\Theta) &\approx \frac{J(\Theta+\varepsilon) - J(\Theta-\varepsilon)}{2\varepsilon} \end{aligned} \tag{28}$$

So, $\frac{\partial}{\partial\Theta}J(\Theta) \approx \frac{J(\Theta+\varepsilon)-J(\Theta-\varepsilon)}{2\varepsilon}$ is the approximation of the derivative.

Usually ε is a very small value (like $\varepsilon = 10^{-4}$). If we use a value of ε excessively small we'll obtain actually the derivative and not the approximation (looking at figure 28 we can notice that for values of ε excessively small the approximation line -red line- will correspond to the tangent line in Θ-the blue line- so the derivative will be not an approximation but the real value).

In Matlab we will implement the computation of the approximation as follows:

```
gradApprox = (J(theta + EPSILON) - J(theta -
    EPSILON) ) / (2 * EPSILON)
```

Exercise

Let $J(\Theta) = \Theta^3$. Furthermore, let $\Theta = 1$ and $\varepsilon = 0,01$. You use the formula $\frac{J(\Theta+\varepsilon)-J(\Theta-\varepsilon)}{2\varepsilon}$ to approximate the derivative. What value do you get using this approximation? (When using $\Theta = 1$ the true exact derivative is $\frac{\partial}{\partial\Theta}J(\Theta) = 3$)

Solution

- $J(\Theta) = \Theta^3$
- $\Theta = 1$
- $\varepsilon = 0,01$
- $\Theta + \varepsilon = 1 + 0,01 = 1,01$
- $\Theta - \varepsilon = 1 - 0,01 = 0,99$
- $J(\Theta + \varepsilon) = 1,01^3 = 1,030301$
- $J(\Theta - \varepsilon) = 0,99^3 = 0,970299$
- $2 * \varepsilon = 2 * 0,01 = 0,02$
- $\frac{J(\Theta+\varepsilon)-J(\Theta-\varepsilon)}{2\varepsilon} = \frac{1,030301-0,970299}{0,02} = 3,001$

Till now we considered the case of $\Theta \in \mathbb{R}$. Now consider the case of where Θ is a vector parameter, so:

- $\Theta \in \mathbb{R}^n$ (e.g., Θ is the *unrolled* version of $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$)
- $\Theta = \Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$

Starting from the approximation just described, for the case of where Θ is a vector we can use a similar idea to approximate all the partial derivative terms :

Equation 29 Gradient checking approximation computation, case $\Theta \in \mathbb{R}^n$

$$\begin{aligned}
\frac{\partial}{\partial\Theta_1}J(\Theta) &\approx \frac{J(\Theta_1+\varepsilon,\Theta_2,\Theta_3,\dots,\Theta_n)-J(\Theta_1-\varepsilon,\Theta_2,\Theta_3,\dots,\Theta_n)}{2\varepsilon}\\
\frac{\partial}{\partial\Theta_2}J(\Theta) &\approx \frac{J(\Theta_1,\Theta_2+\varepsilon,\Theta_3,\dots,\Theta_n)-J(\Theta_1,\Theta_2-\varepsilon,\Theta_3,\dots,\Theta_n)}{2\varepsilon}\\
&\vdots\\
\frac{\partial}{\partial\Theta_n}J(\Theta) &\approx \frac{J(\Theta_1,\Theta_2,\Theta_3,\dots,\Theta_n+\varepsilon)-J(\Theta_1,\Theta_2,\Theta_3,\dots,\Theta_n-\varepsilon)}{2\varepsilon}
\end{aligned} \tag{29}$$

Equation 29 give us way to numerically approximate the partial derivative of J with respect to anyone of the parameters.

Concretely in Matlab we have to implement the following code:

```
for i=1:n,
        thetaPlus = theta;
        thetaPlus(i) = thetaPlus(i) + EPSILON;
        thetaMinus = theta;
        thetaMinus(i) = thetaMinus(i) - EPSILON
            ;
        gradApprox(i) = (J(thetaPlus) - J(
            thetaMinus)) / (2 * EPSILON)
end;
```

The above code computes the partial derivatives of the cost function (or a good approximation of the derivatives) with respect to every parameter in the neural network; in the "for" loop, n is the dimension of the parameter vector Θ. Usually it is better to run that code with the *unrolled* version of the parameters. So, Θ is a long vector of all the parameters in the neural network.

At this point we can compare the derivative numerically computed with the gradient checking *gradApprox* with the derivatives we obtained with the backpropagation algorithm $DVec$ (remember that the backpropagation

is an efficient algorithm to compute the derivatives,or partial derivatives, of the cost function with respect to all of our parameters).

If $gradApprox \approx DVec$ we are more confident that the implementation of the backpropagation algorithm is correct. Summarizing, in the following all steps needed in order to do a correct implementation of the back propagation algorithm are reported:

- Implement backpropagation to compute **DVec** (unrolled $D^{(1)}, D^{(2)}, D^{(3)}$).
- Implement numerical gradient checking to compute **gradApprox**.
- Make sure they give very similar values.
- Turn off gradient checking and use backpropagation for learning.

Note: It is very important to disable the gradient checking code before to train the classifier. If we run the numerical gradient computation on every iteration of gradient descent (or in the inner loop of **costFunction(...)**) the code will be very slow.

0.1.9 Neural networks: Random initialization

In previous lectures we have seen that for gradient descent and the advance optimization methods we needed some initialization values for Θ:

```
optTheta = fminunc(@costFunction, initialTheta,
    options)
```

Considering the gradient descent it is possible to initialize Θ to a vector of zeros (as we did in logistic regression):

```
initialTheta = zeros(n,1)
```

The above initialization works well with the logistic regression, but **it does not work** if we are training a neural network.

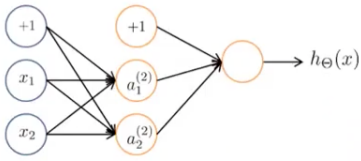


Figure 29: Neural Network initialization

Consider the neural network in figure 29. If we initialize all parameters Θ to zero, that means that both the hidden unit $a_1^{(2)}$ and $a_2^{(2)}$ are going to compute the same function:

$$\begin{aligned} a_1^{(2)} = g(z_1^{(2)}) \implies z_1^{(2)} = \Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 = 0 \\ a_2^{(2)} = g(z_2^{(2)}) \implies z_2^{(2)} = \Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 = 0 \\ a_1^{(2)} = g(z_1^{(2)}) = g(0) = 0.5 \\ a_2^{(2)} = g(z_2^{(2)}) = g(0) = 0.5 \end{aligned} \tag{30}$$

Also the computation of the δ values (in the backpropagation algorithm) presents the same problem, so $\delta_1^{(2)} = \delta_2^{(2)}$.

What we can show is that the partial derivatives with respect to the parameters will satisfy the following equation:

$$\frac{\partial}{\partial \Theta_{01}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{02}^{(1)}} J(\Theta) \tag{31}$$

From equation 31 we can infer that, even after a gradient descent update, we will obtain that $\Theta_{01}^{(1)} = \Theta_{02}^{(1)}$, so they will be some non-zero values but they will be equal.

After each update, parameters corresponding to inputs going into each of the hidden units are identical; consequently also the different hidden units will compute the same values that the input. With these parameters the neural network is not able to compute an interesting function.

In order to avoid the just described problem (called **Symmetry Breaking**), we initialize the parameters of a neural network with a **random initialization**; so each value of Θ will be initialed to a random value between $[-\varepsilon, \varepsilon]$ ($-\varepsilon \leq \Theta_{ij}^{(l)} \leq \varepsilon$). In the following the Matlab code to make a random initialization of Θ:

```
Theta1 = rand(10,11) * (2 * INIT_EPSILON) -
    INIT_EPSILON
Theta2 = rand(1,11) * (2 * INIT_EPSILON) -
    INIT_EPSILON
```

Note that $rand(10, 11)$ will create a 10×11 matrix of random values between $[0, 1]$, so a value x ($0 \leq x \leq 1$) multiplied by $(2\varepsilon) - \varepsilon$ give us a real number between $[-\varepsilon, \varepsilon]$ (note: the ε used in random initialization is **unrelated** from the ε used in gradient checking).

0.1.10 Putting it together

In this section we'll see all steps needed to train a neural network. The first step consist of choosing a neural network architecture (the architecture is the connectivity pattern between the neural network neurons).
How do we choose the neural network architecture?

- **Number of input units**: Dimension of features $x^{(i)}$.
- **Number of output units**: Number of classes.
- **Number of hidden layer**: A reasonable default is to use a single hidden layer (it is the most common). If you use more than one hidden layer, a reasonable default is to use the same number of hidden units in every single hidden layer.
- **Number of hidden units**: usually, the more hidden units, the better. On the other hand, consider that a neural network with a lot of hidden units, can became computationally expensive. Usually the number of hidden units in each layer may be comparable to the dimension of the input features x; it is common to choose a number slightly higher than the number of the input features.

Following the above guideline, usually lead to a neural network that works very well.

Now we'll see all steps we need to implement to **train a neural network**:

1. Randomly **initialize weights**.
2. Implement **forward propagation** to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$.

3. Implement code to compute **cost function** $J(\Theta)$.

4. Implement **backpropagation algorithm** to compute partial derivatives $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$

5. Use **gradient checking** to compare $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$ computed using backpropagation with that computed using numerical estimate of gradient of $J(\Theta)$ (gradient checking).

6. **disable gradient checking** code.

7. Use gradient descent or any advanced optimization method with back propagation to try to **minimize** $J(\Theta)$ as a function of parameter Θ.

Concretely, to implement the backpropagation algorithm, we use a *for* loop over the training examples (there are also other methods but they are more complicated):

for 1=1:m
perform forward propagation and back propagation using example $(x^{(i)}, y^{(i)})$
(get activation $a^{(l)}$ and delta terms $\delta^{(l)}$ for $l = 2, \ldots, L$)

Within the *for* loop we compute the **accumulation term**:

Equation 32 Accumulation term

$$\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T \tag{32}$$

Outside the *for* loop we compute the **partial derivative terms** $\frac{\partial}{\partial \Theta_{uk}^{(l)}} J(\Theta)$ taking into account the regularization term λ.

How gradient descent works in a neural network might seem a little bit magical. Let's see figure 30 to try to understand the intuition about what gradient descent for a neural network do.

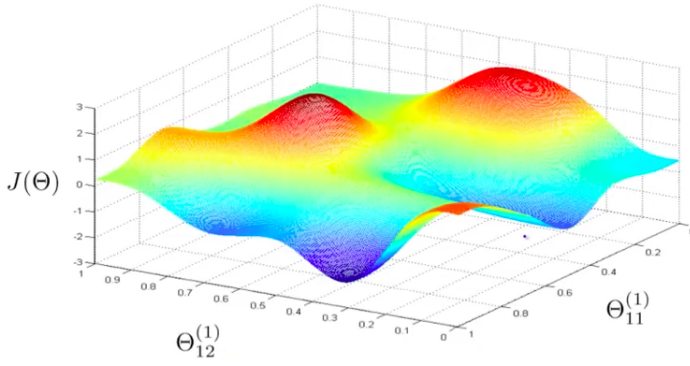


Figure 30: Gradient descent

In figure 30 the cost function $J(\Theta)$ measures how well the neural network fits the training data. The values of $J(\Theta)$ in the figure that are very low corresponds to a setting of the parameters that makes the output $h_\Theta(x^{(i)}) \approx y^{(i)}$.

Gradient descent starts from some random initial points and it repeatedly go down; so, what backpropagation does is to compute the direction of the gradient.