Cairo University
Faculty of Engineering
Computer Engineering Department

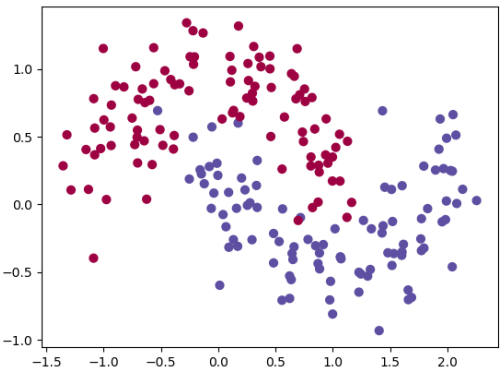
CMPN450
Fall 2018

# Pattern Recognition and Neural Networks.

## Lab 7 – Neural Networks

In this exercise we will implement a simple 3-layer neural network from scratch. We won't derive all the math that's required, but we will try to give you an intuitive explanation of what we are doing and will point to resources to read up on the details.

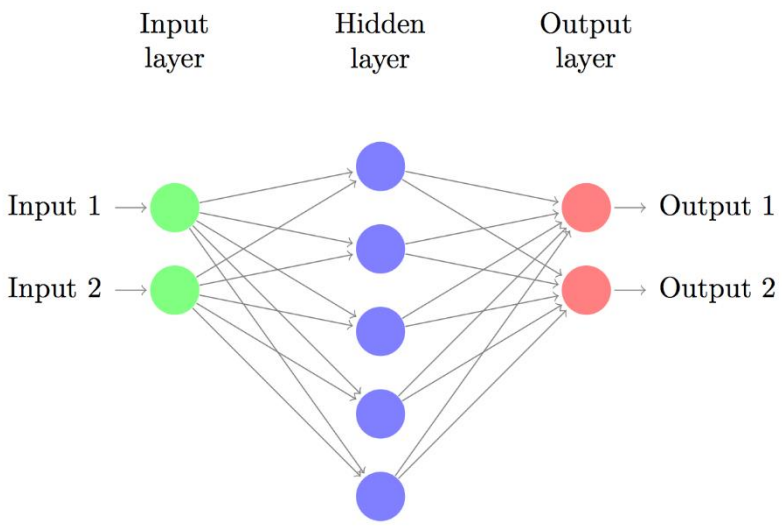
The dataset we generated has two classes, plotted as red and blue points. You can think of the blue dots as male patients and the red dots as female patients, with the x- and y- axis being medical measurements.




Our goal is to train a Machine Learning classifier that predicts the correct class (male or female) given the x- and y- coordinates. Note that the data is not linearly separable, we can't draw a straight line that separates the two classes. This means that linear classifiers, such as Logistic Regression, won't be able to fit the data unless you hand-engineer non-linear features (such as polynomials) that work well for the given dataset.
In fact, that's one of the major advantages of Neural Networks. You don't need to worry about feature engineering. The hidden layer of a neural network will learn features for you.

## Training a Neural Network:

Let's now build a 3-layer neural network with one input layer, one hidden layer, and one output layer. The number of nodes in the input layer is determined by the dimensionality of our data (two input features in our case). Similarly, the number of nodes in the output layer is determined by the number of classes we have, also (two classes in our case). Note that, because we only have 2 classes we could actually get away with only one output node predicting 0 or 1, but having 2 makes it easier to extend the network to more classes later on. The input to the network will be x- and y- coordinates and its output will be two probabilities, one for class 0 ("female") and one for class 1 ("male"). It looks something like this:




We can choose the dimensionality (the number of nodes) of the hidden layer. The **more nodes we put into the hidden layer the more complex functions we will be able fit**. But higher dimensionality comes at a cost. First, more computation is required to make predictions and learn the network parameters. A bigger number of parameters also means we become more prone to **overfitting** our data.

## How to choose the size of the hidden layer?

While there are some general guidelines and recommendations, it always depends on your specific problem and is more of an art than a science. We will play with the number of nodes in the hidden layer later on and see how it affects our output.

We also need to pick an ***activation function*** for our hidden layer. The activation function transforms the inputs of the layer into its outputs. A nonlinear activation function is what allows us to fit nonlinear hypotheses. Common choices for activation functions are **tanh**, **the sigmoid function**, or **ReLUs**. We will use **tanh**, which performs quite well in many scenarios. A nice property of these functions is that their derivate can be computed using the original function value. **(How is that?)**

This is useful because it allows us to compute the activation function once and re-use its value later on to get the derivative.

**Because we want our network to output probabilities the activation function for the output layer will be the softmax, which is simply a way to convert raw scores to probabilities**. *(If you're familiar with the logistic function you can think of softmax as its generalization to multiple classes.)*

## How our network makes predictions

Our network makes predictions using *forward propagation*, which is just a bunch of matrix multiplications and the application of the activation function(s) we defined above.

If $x$ is the 2-dimensional input to our network then we calculate our prediction $\hat{y}$ (also two-dimensional) as follows:

$$z_1 = xW_1 + b_1$$
$$a_1 = \tanh(z_1)$$
$$z_2 = a_1W_2 + b_2$$

$$a_2 = \hat{y} = \text{softmax}(z_2)$$

$z_i$ is the weighted sum of inputs of layer $i$ (bias included) and $a_i$ is the output of layer $i$ after applying the activation function. $W_1, b_1, W_2, b_2$ are the parameters of our network, which we need to learn from our training data.

## Backpropagation and learning the parameters:

Remember that our goal is to find the **parameters** that minimize our loss function. The loss function is equal to $L = -[ylog(\hat{y}) + (1-y)\log(1-\hat{y})$. This function is called cross entropy loss function. We can use gradient descent to find its minimum.

As an input, gradient descent needs the gradients (vector of derivatives) of the loss function with respect to our parameters: $\frac{\partial L}{\partial W_1}$, $\frac{\partial L}{\partial W_2}$, $\frac{\partial L}{\partial b_1}$, $\frac{\partial L}{\partial b_2}$To calculate these gradients we use the **backpropagation algorithm**, which is a way to efficiently calculate the gradients starting from the output.

It's required to derive the four partial derivatives $\frac{\partial L}{\partial W_1}$, $\frac{\partial L}{\partial W_2}$, $\frac{\partial L}{\partial b_1}$, $\frac{\partial L}{\partial b_2}$ given that you know $L = -[ylog(\hat{y}) + (1-y)\log(1-\hat{y})$. We already computed the derivative $\frac{\partial L}{\partial z_2}$ which represents the derivative of the loss function with respect to $z_2$(i.e. the value of the output before applying the activation function). You will find the derivative computed in the two-dimensional vector **delta3** in the code.

## Requirements:

1. Fill in the missing functions in the code for feedforwarding and backpropagation. Derive the four partial derivatives $\frac{\partial L}{\partial W_1}$, $\frac{\partial L}{\partial W_2}$, $\frac{\partial L}{\partial b_1}$, $\frac{\partial L}{\partial b_2}$ and plug them in the backpropagation algorithm.

2. Try changing the **number of hidden layer neurons**. What do you conclude?

3. Try changing the **value of the learning rate.** What do you conclude?

4. Change the activation function to be **sigmoid function** instead of **tanh.** What will need to be modified in your code?

5. Partition the dataset into **training** and **validation** sets. Using the **validation** set, determine the best value for the **learning rate** (try values from 0-1 with step 0.05) and the number of hidden layer neurons (try values from 2 to 50 with step 1).