Multi-layer Perceptron Networks

Our task is to tackle a class of problems that the simple perceptron cannot solve. Complex perceptron networks not only carry some important information for neuroscience, but are also useful in medical diagnostics, robotics, and the US Postal Service, where they decode (even handwritten) mailing addresses.

We recall the binary output structure of the single-layer perceptron:

$$o^j = f(\sum_{i=1}^{n} w_i x_i) \text{ where } f(\theta) = \begin{cases} 1, & \text{if } \theta > 0. \\ 0, & \text{if } \theta \leqslant 0. \end{cases} \tag{1}$$

and first ask whether the thresholding action of $f(\theta)$ could be replaced with a more accomodating function. We then outfit the network with a hidden layer that allows it to solve more difficult classification problems.

Sigmoidal Neurons

Instead of returing a 1 for input > 0 and a 0 for input $\leqslant 0$, a sigmoidal threshold returns non-binary output for inputs in a particular range. In the sigmoidal functions we will consider, the parameter β determines the steepness of the sigmoid, or in other words, the width of the range for which the output will lie somehwere between 1 and 0.

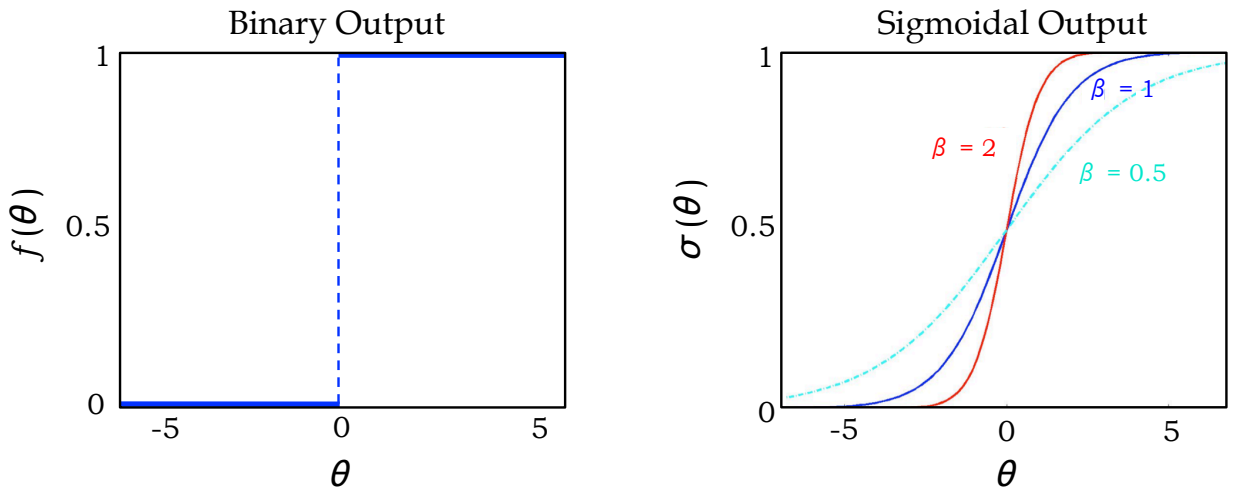

Figure 1: Comparison of binary and sigmoidal thresholding functions. Notice in the sigmoidal case that the range of inputs for which the output is not 1 or 0 is smaller for larger β.

$$\sigma(\theta, \beta) = e^{\beta\theta}/(1 + e^{\beta\theta}) \tag{2}$$

Is one example of a sigmoid, and produces the sigmoidal thresholding function illustrated above.

Hidden Layer

We now add a hidden layer to our perceptron. Notice that the hidden neurons and final output neuron are sigmoidal and not binary.

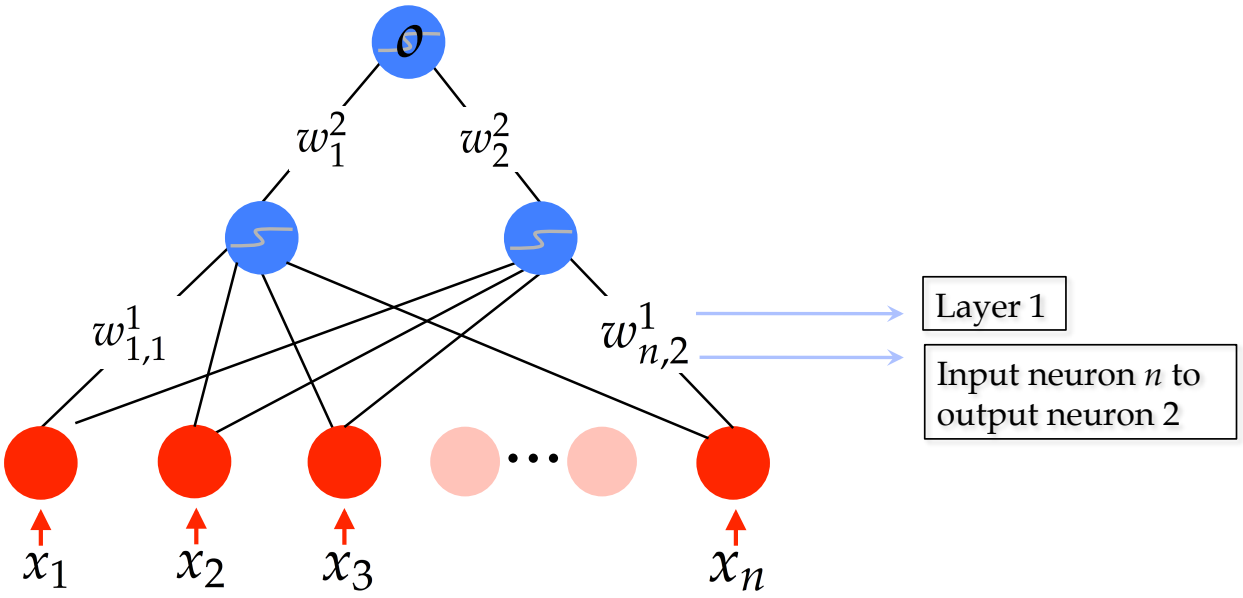

Figure 2: Two layer perceptron with (sigmoidal) hidden layer and sigmoidal output

The output at each layer (hidden and output) is calculated by passing a weighted sum of inputs through the sigmoid function. Each hidden neuron sums its inputs multiplied by their weights ($\theta_j = \sum_{i=1}^{n} w_{i,j}^1 x_i$ for hidden neuron j), then returns $\sigma(\theta_j)$ as its output. The process is repeated at the next layer: the output neuron sums the contributions of the hidden neurons (outputs calculated at last step) multiplied by their weights, and passes that sum through the sigmoid.

Next we ask how the weights are updated at each iteration of training. Recall that we are working under the umbrella of supervised learning, where desired outputs are known and we use the difference between current and desired outputs to update weight values. The process in a multilayer network is more difficult than in a single layer network. With a single layer, we could compare the actual output o to the desired output y, which is known. In this more complicated scenario, we can compare y to the guess of the final output neuron, but we cannot perform this comparison at the hidden layer because we do not know what the output of each hidden neuron should be.

How do we handle the hidden neurons and layer 1 weights? We employ the method of **gradient descent**. As this technique allows us to consider the effect of changing weights in layer 1 on the output of layer 2, it is known in this context as **back propagation**. The mathematics of the derivation using gradient descent on the square error are omitted from the algorithm you will find on the following page. Inquire for details.

Step One: for input $x = [x_1....x_n]$, calculate output h_j at each hidden layer neuron

$$\boldsymbol{h_j = \sigma(\sum_{i=1}^{n} w_{i,j}^1 x_i)} \text{ where } \sigma(\theta) = \frac{1}{1 + e^{-\beta\theta}} \tag{3}$$

Use $\beta = 1$.

Step Two: for input $h = [h_1...h_j]$ from hidden layer, calculate output of output neuron

$$\boldsymbol{o = \sigma(\sum_{j} w_j^2 h_j),} \tag{4}$$

where $\sigma(\theta)$ is the same as above. Note that $\boldsymbol{w_j^2}$ indicates weights in layer 2, not squares.

Step Three: calculate $\boldsymbol{\delta^2}$, which will be used to update layer 2 weights

$$\boldsymbol{\delta^2 = (o - y)(o)(1 - o)} \tag{5}$$

because there is one output, δ^2 is a scalar.

Step Four: calculate $\boldsymbol{\delta^1}$, which will be used to update layer 1 weights

$$\boldsymbol{\delta_j^1 = (\delta^2)(w_j^2 h_j(1 - h_j))} \tag{6}$$

because there are j hidden neuron outputs, δ^1 is a j-vector.

Step Five: calculate Δw^2

$$\boldsymbol{\Delta w_j^2 = \ell \delta^2 h_j} \tag{7}$$

where ℓ is the learning rate. Use $\ell = 0.1$.

Step Six: calculate Δw^1

$$\boldsymbol{\Delta w_{i,j}^1 = \ell \delta_j^1 x_i} \tag{8}$$

$\Delta w_{i,j}^1$ can be conveniently stored in matrix form. If δ^1 is a j by 1 vector and x is a 1 by i vector, $\delta^1 * x$ will produce a matrix with j rows and i columns.

Step Seven: update layer 2 weights

$$\boldsymbol{w_j^2 = w_j^2 - \Delta w_j^2} \tag{9}$$

Step Eight: update layer 1 weights

$$\boldsymbol{w_{i,j}^1 = w_{i,j}^1 - \Delta w_{i,j}^1} \tag{10}$$

Project: Boolean Automator

You will train a perceptron with a sigmoidal hidden layer to solve all 16 Boolean operations on two input arguments. With a single-layer perceptron, we were able to solve AND and OR problems, but not the nonlinearly separable XOR problem. Recall the structure of XOR:

x_1	x_2	y
1	1	0
1	0	1
0	1	1
0	0	0

Table 1: XOR with two inputs

In their 1969 book *Perceptrons: An Introduction to Computational Geometry*, Marvin Minsky and Seymour Papert prove that the addition of a hidden layer allows perceptrons to solve such problems. In fact, the multi-layer perceptron described here can solve all possible Boolean operations, with inputs $x_{1,2}$ and desired output y_i for $i = 1...16$:

x_1	x_2	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}	y_{16}
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Table 2: All Boolean operations on two inputs

You will write two functions:

`booletron`: a driver that runs `multiperceptron` on all 16 Boolean operations and checks its accuracy by comparing its output `[guess]` to the desired output `[y]`.

`guess = multiperceptron(x,y)`, where input matrix `[x]` remains the same for all 16 operations and `[y]` varies. `multiperceptron` trains the perceptron to perform the Boolean operation specified by `[y]` using the algorithm detailed previously and returns the perceptron's output after training.

A network with two input neurons, a single hidden layer containing two neurons, and a single output neuron can solve the problem. However, you again need to add a "bias bit" in the form of another input neuron that always receives an input of 1. This can be done by adding another column of ones to the matrix `x`.

You may want to keep track of the weights for layers 1 and 2 in two separate matrices. You should initialize the weights using

```
W1 = 1 1 1                    W2 = 1 -1.
     1 1 1
```

Implementing the training algorithm for 10^5 iterations will suffice.

Because we are using a non-binary thresholding function and a reasonable number of iterations to train, we do not expect the perceptron's guess to perfectly match desired (binary) output y. In application, outputs at a particular level of accuracy could be rounded to match y. To check the accuracy of `multiperceptron` use Mean Squared Error, via the function `MSE`. `mse(guess-y)` will do the trick. For each Boolean operator, display the perceptron's output, the desired output, and the Mean Squared Error. For example, for the XOR output, booletron could print:

```
Perceptron output= 0.064756   0.94630   0.94387   0.068749
Desired output= 0   1   1   0
MSE = 0.0037385
```

Furthermore, for only the XOR problem, you will keep track of the perceptron's accuracy at each iteration of training. This can be achieved by calculating the absolute error of the perceptron's output (read: absolute value of the difference between the final output and desired output). This will be a scalar value: remember that the desired output changes based on the inputs at each iteration. After the final trial, create a figure like that below displaying the convergence of the perceptron's guess to the actual output over the training period.

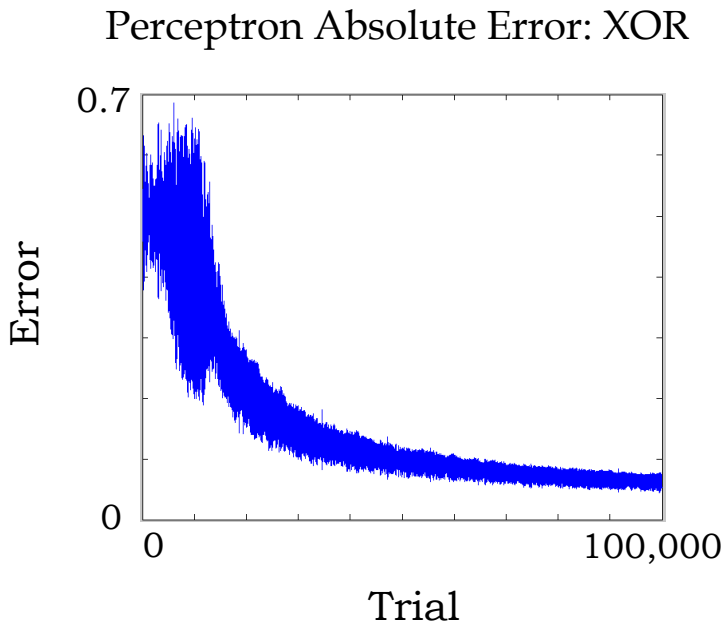

Figure 3: Absolute error of perceptron's output over XOR training

Your work will be graded as follows:

```
 5 pts for header containing detailed usage of each function
 5 pts for further comments in code
 2 pts for indentation
 8 pts for correct driver booletron
10 pts for display of correct outputs and errors
15 pts for correct multiperceptron function
 5 pts for correct error figure for XOR
```