50.021 -AI

Alex

Week 03: Basics of neural networks

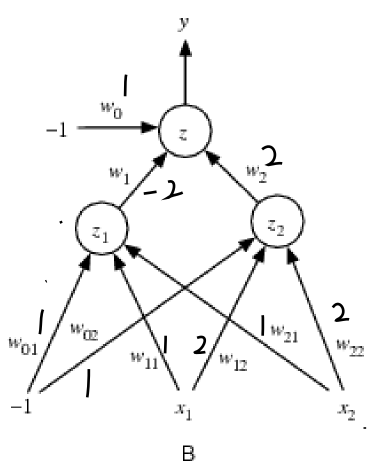
[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.]

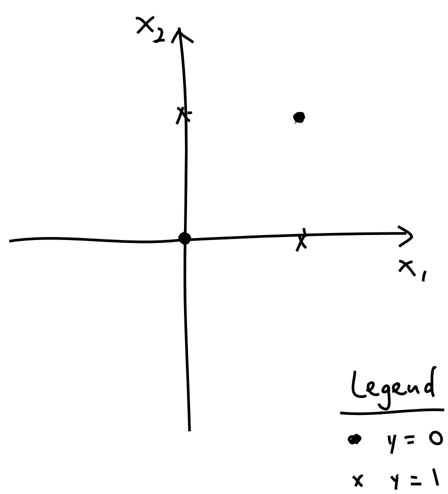
in class Problem 1 - XOR

Consider the following dataset (xor problem)

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

For the following network, assume that each unit outputs 1 if $\sum_i x_i w_i > 0$ and a 0 otherwise.





1. Pick the weights for the units so that the desired outputs are predicted correctly.
2. Plot the decision boundaries of each of the cells in the x_1,x_2 space.

in class Coding - tanh and linear

In the file modules.py look at the implementation of Sigmoid layer, make sure you understand how it works.

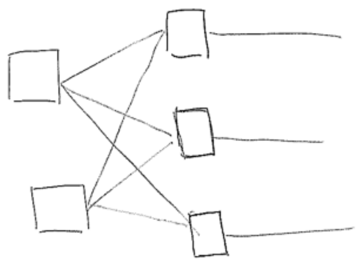
1. Complete the class definitions in `modules.py` that will allow us to add Tanh layers to our networks.
2. Also, complete the class definitions in modules.py for Linear. This requires filling in the backward method.
3. Use `testNN()` in `learn.py` your code on some simple examples, such as xor.

Some helpful hints about the code In general, a linear layer has m inputs and n outputs. The non-linear layers, Sigmoid , etc., have $m = n$. We'll describe the behavior of the code when batchsize is 1, which is how it is used in testNN.

- `forward(X)` computes the unit output Y for input X. X is a $1 \times m$ matrix and the output is a $1 \times n$ matrix. This function generally also remembers (if needed in the backward method) its input and output, that is, it sets self.X and self.Y.
- `backward(DY)` computes the "error" for the next layer given the error, DY from the previous layer. This return value is the dX in the description below. In the Linear unit, this also sets self.dW and self.dB, the gradient of the error with respect to the weights (see below).

Linear is the most complicated layer. The following scan explains the components in the computations done by Linear.

One tricky bit is that `dB` is expected to be a vector in the code (its shape is $(n,)$) but `dY` is a matrix (its shape is $(1, n)$). So, `self.dB = DY.sum(axis=0)` does this mapping.



X $\quad$ Y

$1 \times m$ $\quad$ $1 \times n$

$W : m \times n$

$B : 1 \times n$

Forward

$$Y = X \cdot W + B \qquad Y_j = \left(\sum_i X_i \cdot W_{ij} \right) + B_j$$

Backward

dY coming in is $\frac{\partial E}{\partial Y} : 1 \times n \qquad dY_j = \frac{\partial E}{\partial Y_j}$

dW is $\frac{\partial E}{\partial W} = \frac{\partial Y}{\partial W} \cdot \frac{\partial E}{\partial Y} : m \times n \qquad dW_{ij} = \frac{\partial E}{\partial W_{ij}}$

$= X^T \cdot dY$

dB is $\partial E / \partial B = \frac{\partial Y}{\partial B} \cdot \frac{\partial E}{\partial Y} : 1 \times n \qquad dB_j = \frac{\partial E}{\partial B_j}$

$= dY$

Now need to compute, to pass backward

$dX = \frac{\partial E}{\partial X} : 1 \times m \qquad dX_i = \frac{\partial E}{\partial X_i} = \sum_j W_{ij} \cdot dY_j$

$$\frac{\partial E}{\partial X} = \frac{\partial Y}{\partial X} \cdot \frac{\partial E}{\partial Y}$$

$$= \underbrace{\underset{1 \times n}{dY} \cdot \underset{n \times m}{W^T}}_{1 \times m}$$