Classification 分类

Given: Training data: $(x_1, y_1), \ldots, (x_n, y_n)$, $x_i \in \mathbb{R}^d$ and y_i is discrete (categorical/qualitative), $y_i \in Y$.

Example $Y = \{-1, +1\}$, $Y = \{0, 1\}$

Task: Learn a classification function, $f: \mathbb{R}^d \to Y$

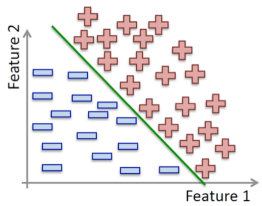
Linear Classification: A classification model is said to be linear if it is represented by a linear function f (linear hyperplane)

Perceptron

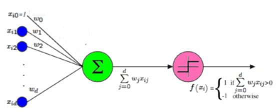
- Belongs to Neural Networks class of algorithms (algorithms that try to mimic how the brain functions).
- The first algorithm used was the Perceptron (Resenblatt 1959).
- Worked extremely well to recognize:
  1. handwritten characters (LeCun et a. 1989),
  2. spoken words (Lang et al. 1990),
  3. faces (Cottrel 1990)
- NN were popular in the 90' s but then lost some of its popularity.
- Now NN back with deep learning.

perceptron也属于NN算法，是NN中第一个使用的算法

- Linear classification method.
- Simplest classification method.
- Simplest neural network.
- For perfectly separated data.



线性分类



perceptron 模拟的是神经的行为

Given n examples and d features.

$$f(x_i) = sign(\sum_{j=0}^{d} w_j x_{ij})$$

- Works perfectly if data is linearly separable. If not, it will not converge.
- Idea: Start with a random hyperplane and adjust it using your training data.
- Iterative method.

从一个超平面开始，不断用数据去修正.

Perceptron Algorithm

Input: A set of examples, $(x_1, y_1), \cdots, (x_n, y_n)$
Output: A perceptron defined by $(w_0, w_1, \cdots, w_d)$

Begin
- Initialize the weights w_j to 0 $\forall j \in \{0, \cdots, d\}$
- Repeat until convergence
  - For each example x_i $\forall i \in \{1, \cdots, n\}$
    - if $y_i f(x_i) \le 0$ #an error?
      - update all w_j with $w_j := w_j + y_i x_{ij}$ #adjust the weights

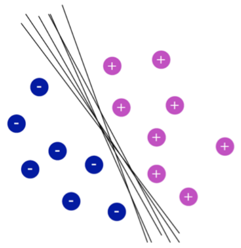
End

Some observations:

- The weights w_1, ..., w_d determine the slope of the decision boundary.
- w_0 determines the offset of the decision boundary (sometimes noted b).
- Line 6 corresponds to:
  - Mistake on positive: add x to weight vector.
  - Mistake on negative: subtract x from weight vector.
  - Some other variants of the algorithm add or subtract ηx or 1.
- Convergence happen when the weights do not change anymore (difference between the last two weight vectors is 0).

- The w_j determine the contribution of x_j to the label.
- $-w_0$ is a quantity that $\sum_{j=1}^{d} w_j x_j$ needs to exceed for the perceptron to output 1.
- Can be used to represent many Boolean functions: AND, OR, NAND, NOR, NOT but not all of them (e.g., XOR).

Choice of the hyperplane



Lots of possible solutions!

Digression: Idea of SVM is to find the optimal solution.

From perceptron to NN

- Neural networks use the ability of the Perceptrons to represent elementary functions and combine them in a network of layers.
- However, a cascade of linear functions is still linear!
- And we want networks that represent highly non-linear functions.

perceptron只能线性分类，对非线性数据效果较差.

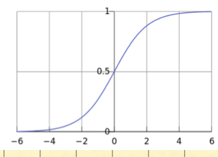
- Also, perceptron used a **step function**, which is non-differentiable and not suitable for gradient descent in case data is not linearly separable.
- We want a function whose output is a differentiable function of the inputs. One possibility is to use the **sigmoid function**:

使用激活函数处理输入

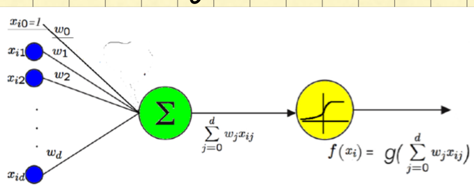
$$g(z) = \frac{e^z}{1+e^z} = \frac{1}{1+e^{-z}}$$

$g(z) \to 1$ when $z \to +\infty$

$g(z) \to 0$ when $z \to -\infty$



Perceptron with Sigmoid



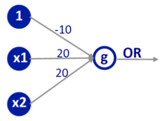
Given n examples and d features.

For an example x_i (the i^{th} line in the matrix of examples)

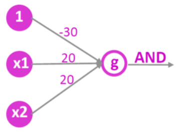
$$f(x_i) = \frac{1}{1+e^{-\sum_{j=0}^{d} w_j x_{ij}}}$$

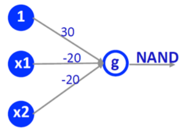
The XOR example

x_1	x_2	x_1 **OR** x_2	$g(z)$
0	0	0	$g(w_0 + w_1x_1 + w_2x_2) = g(-10)$
0	1	1	$g(10)$
1	0	1	$g(10)$
1	1	1	$g(30)$



Similarly, we obtain the perceptrons for the AND and NAND:



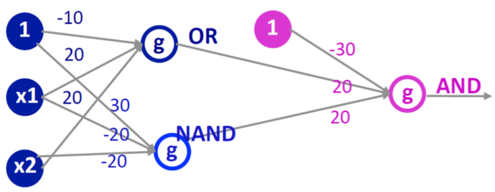


Note: how the weights in the NAND are the inverse weights of the AND.

Let' s try to create a NN for the XOR function using elementary perceptrons.

x_1	x_2	x_1 **XOR** x_2	$(x_1$ **OR** $x_2)$ **AND** $(x_1$ **NAND** $x_2)$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

Let' s put them together...

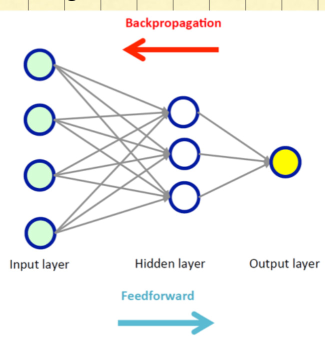


XOR as a combination of 3 basic perceptrons.

Backpropagation algorithm

- Note: Feedforward NN (as opposed to recurrent networks) have no connections that loop.
- Learn the weights for a multilayer network.
- Backpropagation stands for "backward propagation of errors".
- Given a network with a fixed architecture (neurons and interconnections).
- Use Gradient descent to minimize the squared error between the network output value o and the ground truth y.
- We suppose multiple output k.
- Challenge: Search in all possible weight values for all neurons in the network.

Feedforward-Backpropagation



Notations:

- x_{ij}: the i^{th} input to neuron j.
- w_{ij}: the weight associated with the i^{th} input to neuron j.
- $z_j = \Sigma w_{ij} x_{ij}$, weighted sum of inputs for neuron j.
- o_j: output computed by neuron j.
- g is the sigmoid function.
- *outputs*: the set of neurons in the output layer.
- *Succ(j)*: the set of neurons whose immediate inputs include the output of neuron j.

Backpropagation rules

- We consider k outputs
- For an example e defined by (x, y), the error on training example e, summed over all output neurons in the network is:

$$E_e(w) = \frac{1}{2}\sum_k (y_k - o_k)^2$$

- Remember, gradient descent iterates through all the training examples one at a time, descending the gradient of the error w.r.t. this example.

$$\Delta w_{ij} = -\alpha \frac{\partial E_e(w)}{\partial w_{ij}}$$

$$\frac{\partial E_e}{\partial w_{ij}} = \frac{\partial E_e}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}} = \frac{\partial \mathbf{E_e}}{\partial \mathbf{z_j}} x_{ij}$$

$$\Delta w_{ij} = -\alpha \frac{\partial \mathbf{E_e}}{\partial \mathbf{z_j}} x_{ij}$$

We consider two cases in calculating $\frac{\partial E_e}{\partial z_j}$ (let' s abandon the index e):

- **Case 1: Neuron j is an output neuron**
- **Case 2: Neuron j is a hidden neuron**

- **Case 1: Neuron j is an output neuron**

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial z_j}$$

$$\frac{\partial E}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}\sum_k (y_k - o_k)^2$$

$$\frac{\partial E}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}(y_j - o_j)^2$$

$$\frac{\partial E}{\partial o_j} = \frac{1}{2}\, 2\, (y_j - o_j)\frac{\partial (y_j - o_j)}{\partial o_j}$$

$$\frac{\partial E}{\partial o_j} = -(y_j - o_j)$$

We have: $o_j = g(z_j)$

$$\frac{\partial o_j}{\partial z_j} = \frac{\partial g(z_j)}{\partial z_j}$$

$$\frac{\partial o_j}{\partial z_j} = o_j(1 - o_j)$$

$$\frac{\partial E}{\partial z_j} = -(y_j - o_j)o_j(1 - o_j)$$

$$\Delta w_{ij} = \alpha (y_j - o_j)o_j(1 - o_j)x_{ij}$$

We will note

$$\delta_j = -\frac{\partial E}{\partial z_j}$$

$$\Delta w_{ij} = \alpha\ \delta_j\ x_{ij}$$

- **Case 2: Neuron j is a hidden neuron**

$$\frac{\partial E}{\partial z_j} = \sum_{k \in succ\{j\}} \frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial z_j} = \sum_{k \in succ\{j\}} -\delta_k \frac{\partial z_k}{\partial z_j}$$

$$\frac{\partial E}{\partial z_j} = \sum_{k \in succ\{j\}} -\delta_k \frac{\partial z_k}{\partial o_j}\frac{\partial o_j}{\partial z_j}$$

$$\frac{\partial E}{\partial z_j} = \sum_{k \in succ\{j\}} -\delta_k\ w_{jk} \frac{\partial o_j}{\partial z_j}$$

$$\frac{\partial E}{\partial z_j} = \sum_{k \in succ\{j\}} -\delta_k\ w_{jk}\ o_j\ (1 - o_j)$$

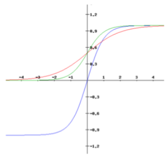
$$\delta_j = -\frac{\partial E}{\partial z_j} = o_j\ (1 - o_j) \sum_{k \in succ\{j\}} \delta_k\ w_{jk}$$

Backpropagation algorithm (BP)

- **Input:** training examples (x, y), learning rate α (e.g., $\alpha = 0.1$), n_i, n_h and n_o.
- **Output:** a neural network with one input layer, one hidden layer and one output layer with n_i, n_h and n_o number of neurons respectively and all its weights.
  1. Create feedforward network (n_i, n_h, n_o)
  2. Initialize all weights to a small random number (e.g., in [−0.2, 0.2])
  3. Repeat until convergence
     For each training example (x, y)
     - **I. Feed forward:** Propagate example x through the network and compute the output o_j from every neuron.
     - **II. Propagate backward:** Propagate the errors backward.
       Case 1 For each output neuron k, calculate its error
       $\delta_k = o_k(1 - o_k)(y_k - o_k)$
       Case 2 For each hidden neuron h, calculate its error
       $\delta_h = o_h(1 - o_h)\sum_{k \in Succ(h)} w_{hk}\delta_k$
     - **III. Update each weight:** $w_{ij} \leftarrow w_{ij} + \alpha\delta_j x_{ij}$

Observations

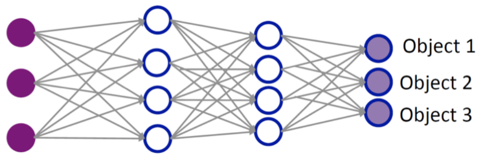
- Convergence: small changes in the weights
- There are other activation functions. Hyperbolic tangent function, is practically better for NN as its outputs range from -1 to 1.



$g(x) = sigmoid(x) = \frac{e^{kx}}{1+e^{kx}}$ for $k = 1$, $k = 2$, etc.

$g(x) = tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$ (It is a rescaling of the logistic/sigmoid function!)

Multi-class case etc.



- Nowadays, networks with more than two layers, a.k.a. deep networks, have proven to be very effective in many domains.
- Examples of deep networks: restricted Boltzman machines, convolutional NN, auto encoders, etc.