

Artificial Intelligence (CS311)

Lecture 10: Perceptron & Neural Networks

Credit: Ansa Salleb-Aouissi, and “Artificial Intelligence: A Modern Approach”, Stuart Russell and Peter Norvig, and “The Elements of Statistical Learning”, Trevor Hastie, Robert Tibshirani, and Jerome Friedman, and “Machine Learning”, Tom Mitchell.

Perceptron

Classification

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

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

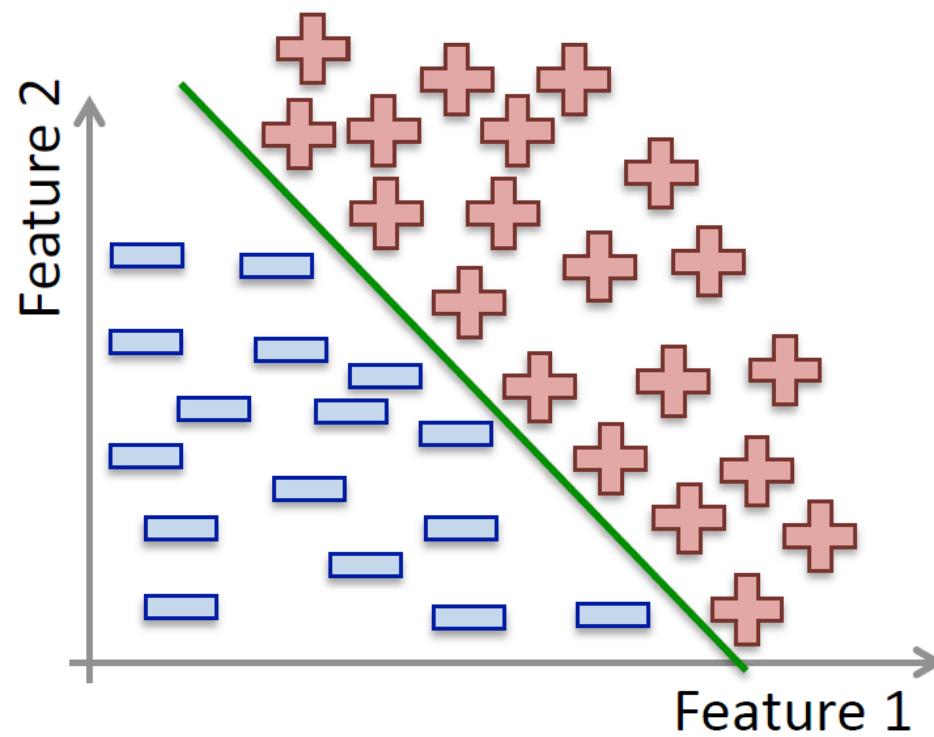
Task: Learn a classification function, $f: \mathbb{R}^d \rightarrow Y$

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

Perceptron

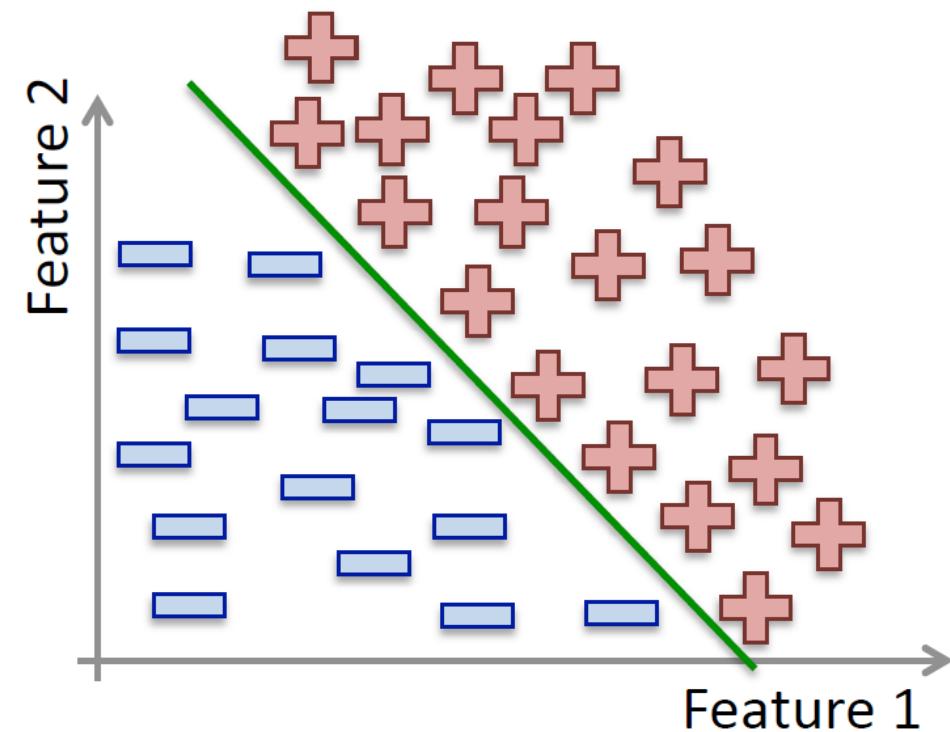
- Belongs to Neural Networks class of algorithms (algorithms that try to mimic how the brain functions).
- The first algorithm used was the Perceptron (Resenblatt 1959).
- Worked extremely well to recognize:
 1. handwritten characters (LeCun et a. 1989),
 2. spoken words (Lang et al. 1990),
 3. faces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

Perfectly separable data

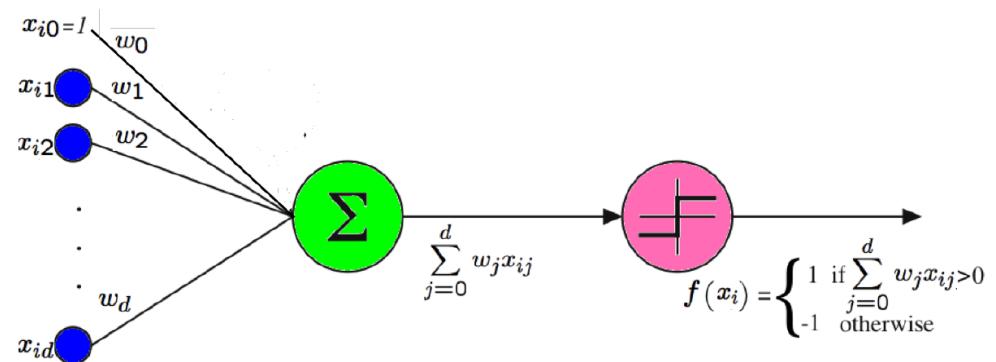
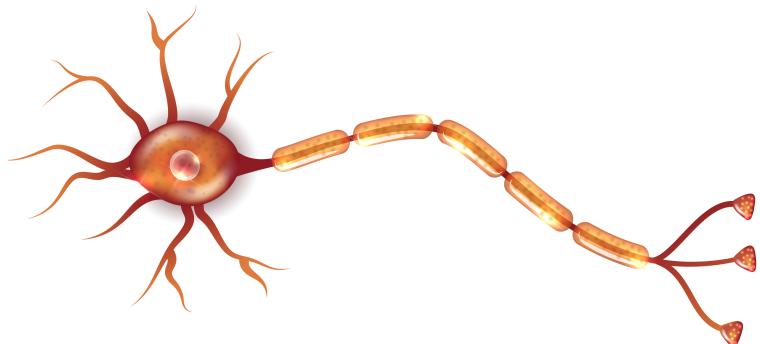


Perceptron

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



Perceptron



Given n examples and d features.

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

Perceptron

- 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.

Perceptron

Perceptron Algorithm

Input: A set of examples, $(x_1, y_1), \dots, (x_n, y_n)$

Output: A perceptron defined by (w_0, w_1, \dots, w_d)

Begin

2. Initialize the weights w_j to 0 $\forall j \in \{0, \dots, d\}$
3. Repeat until convergence
 4. For each example $x_i \quad \forall i \in \{1, \dots, n\}$
 5. if $y_i f(x_i) \leq 0$ #an error?
 6. update all w_j with $w_j := w_j + y_i x_{ij}$ #adjust the weights

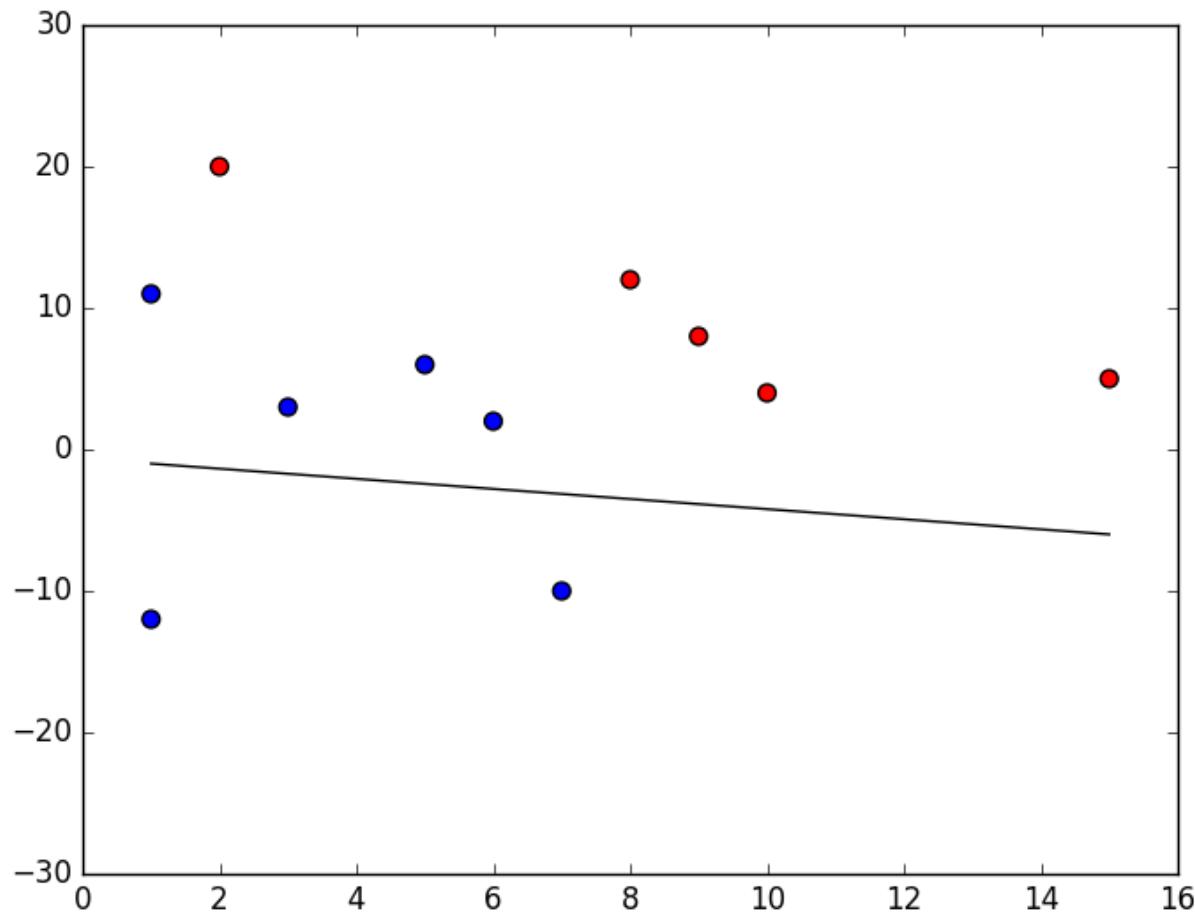
End

Perceptron

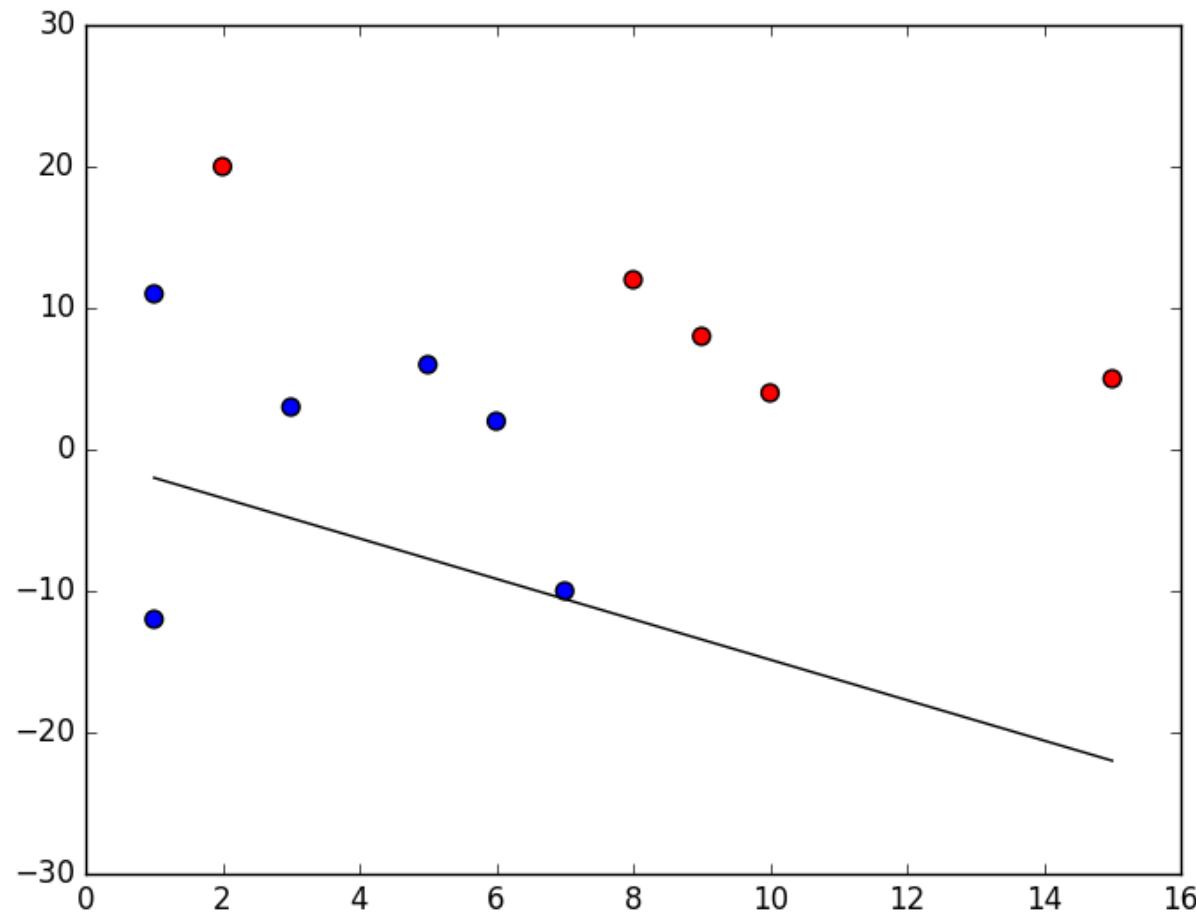
Some observations:

- The weights w_1, \dots, 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 1.
- Convergence happens when the weights do not change anymore (difference between the last two weight vectors is 0).

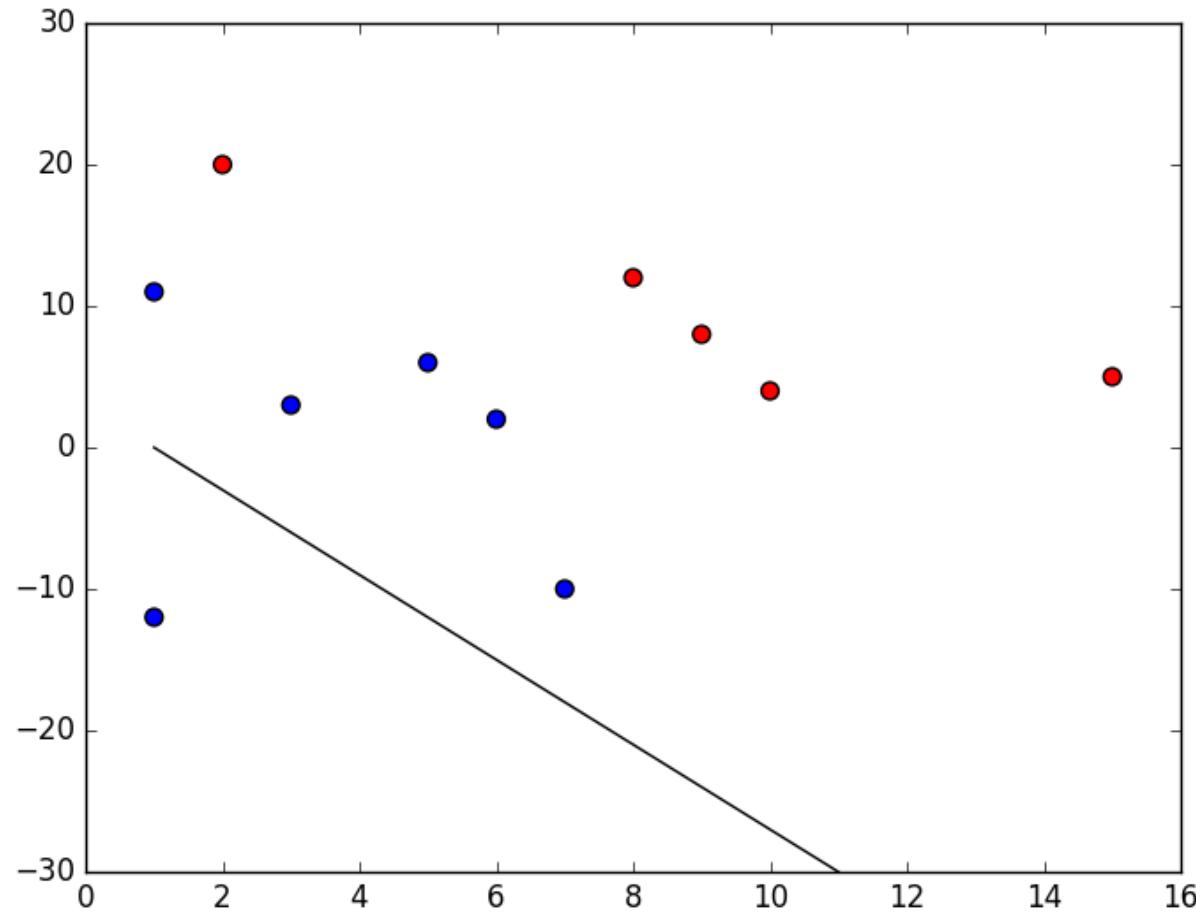
Perceptron: Example



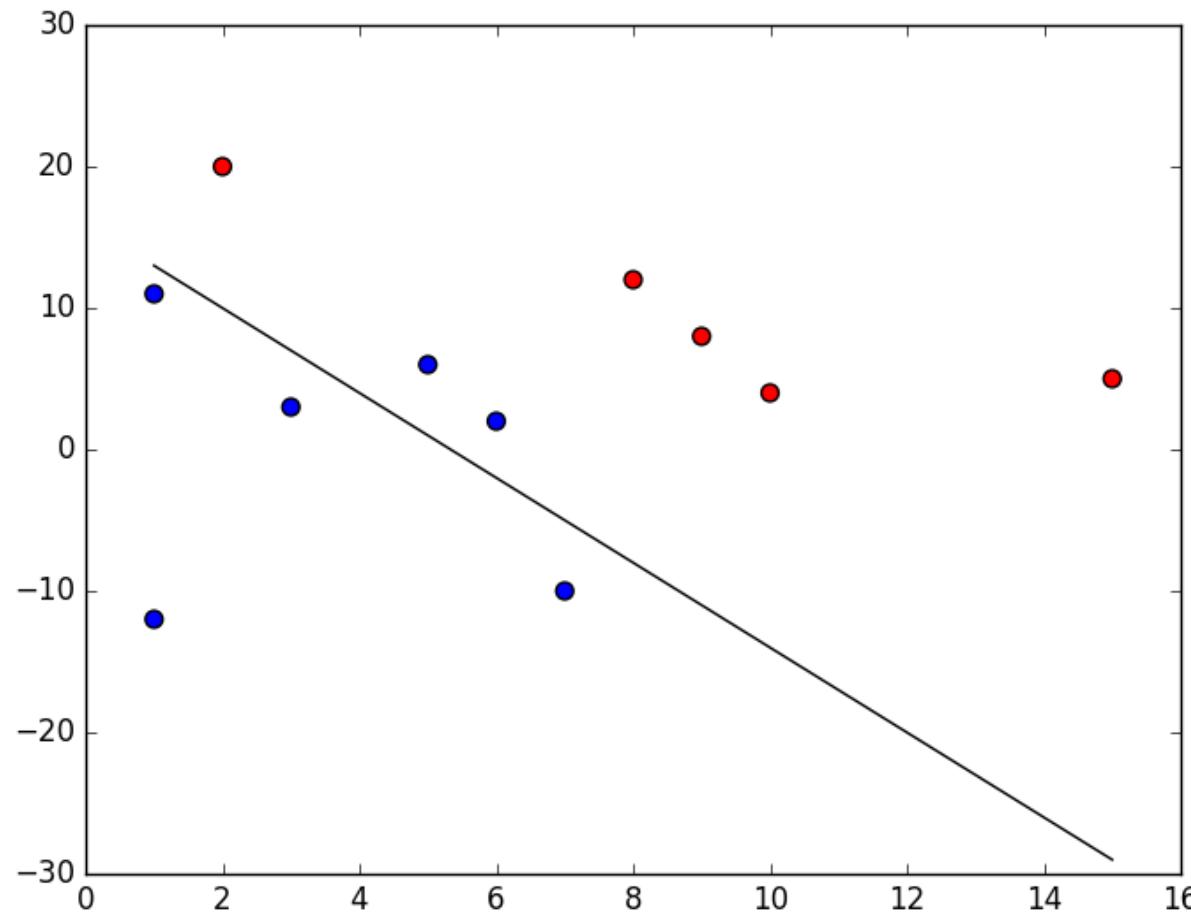
Perceptron: Example



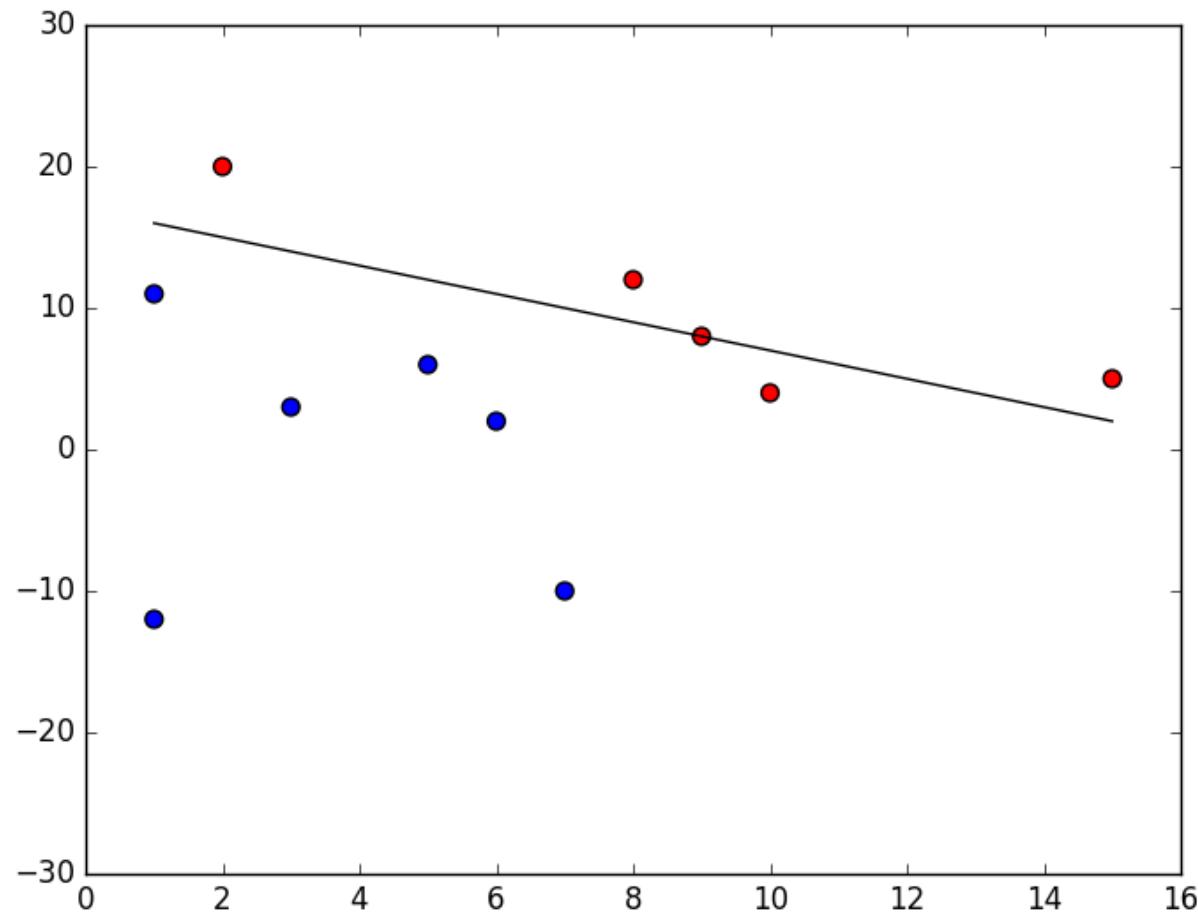
Perceptron: Example



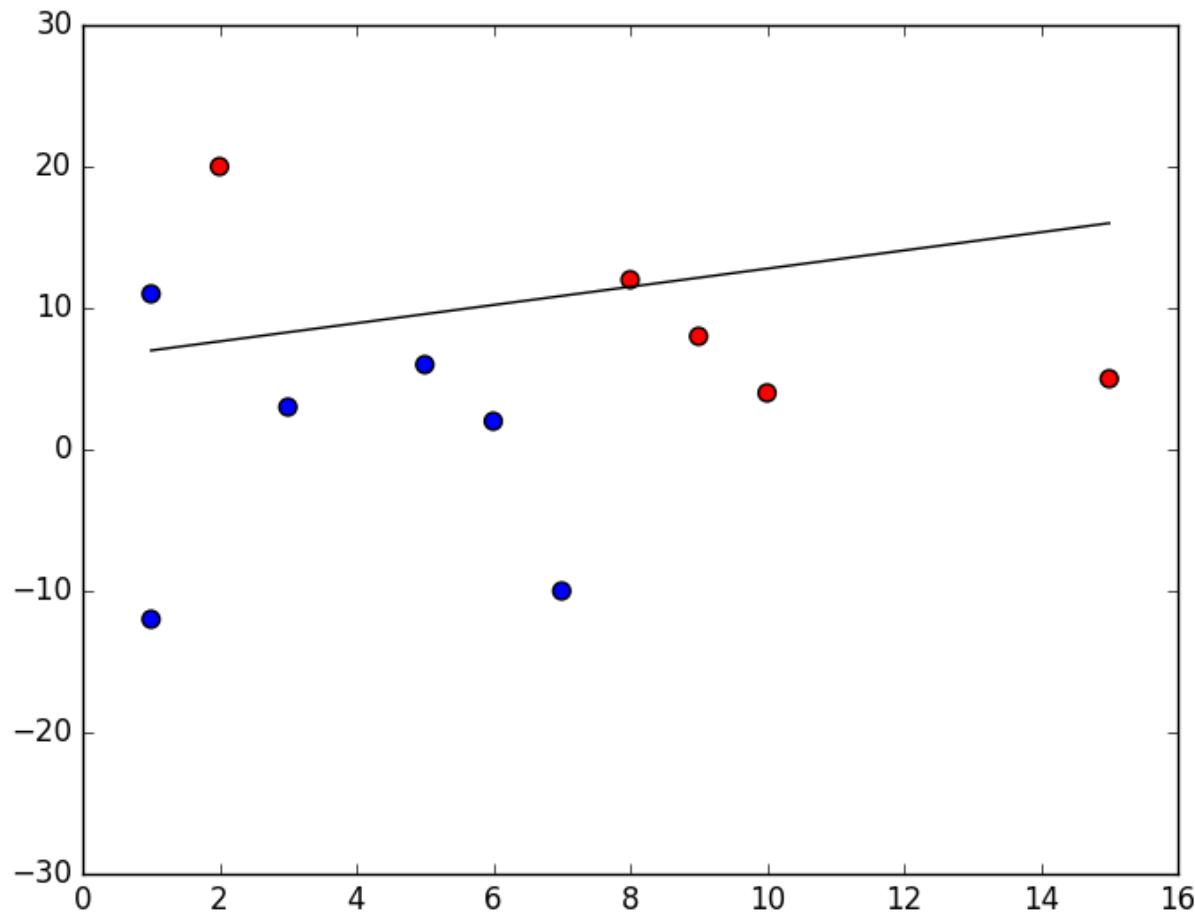
Perceptron: Example



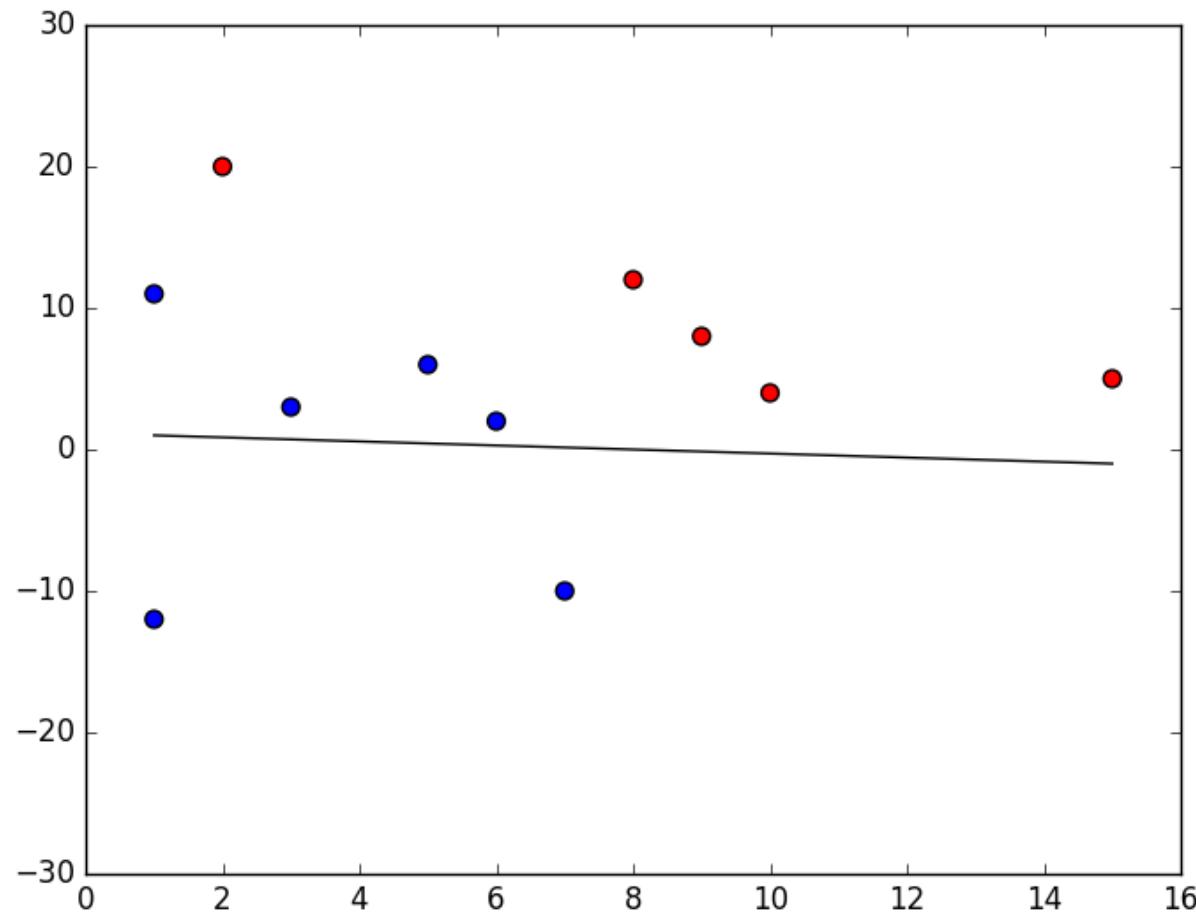
Perceptron: Example



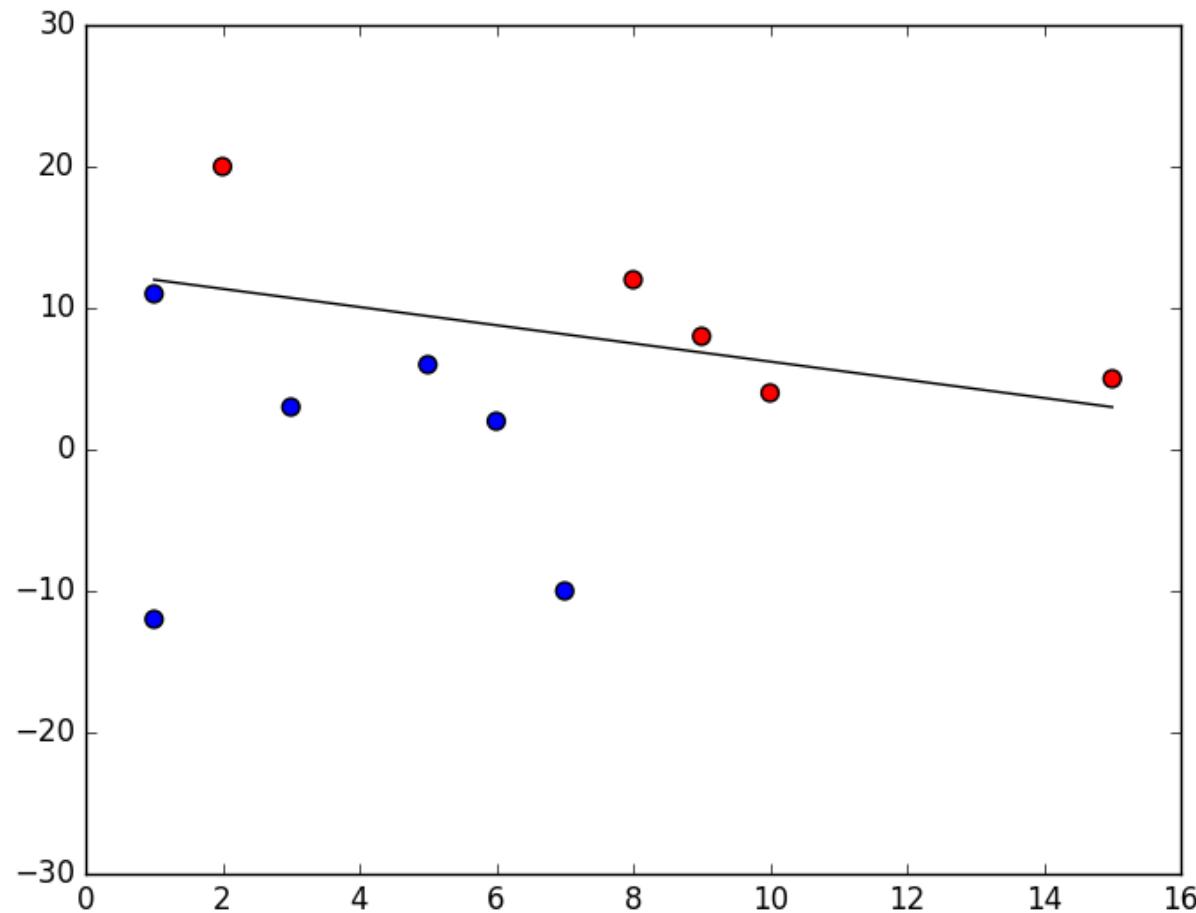
Perceptron: Example



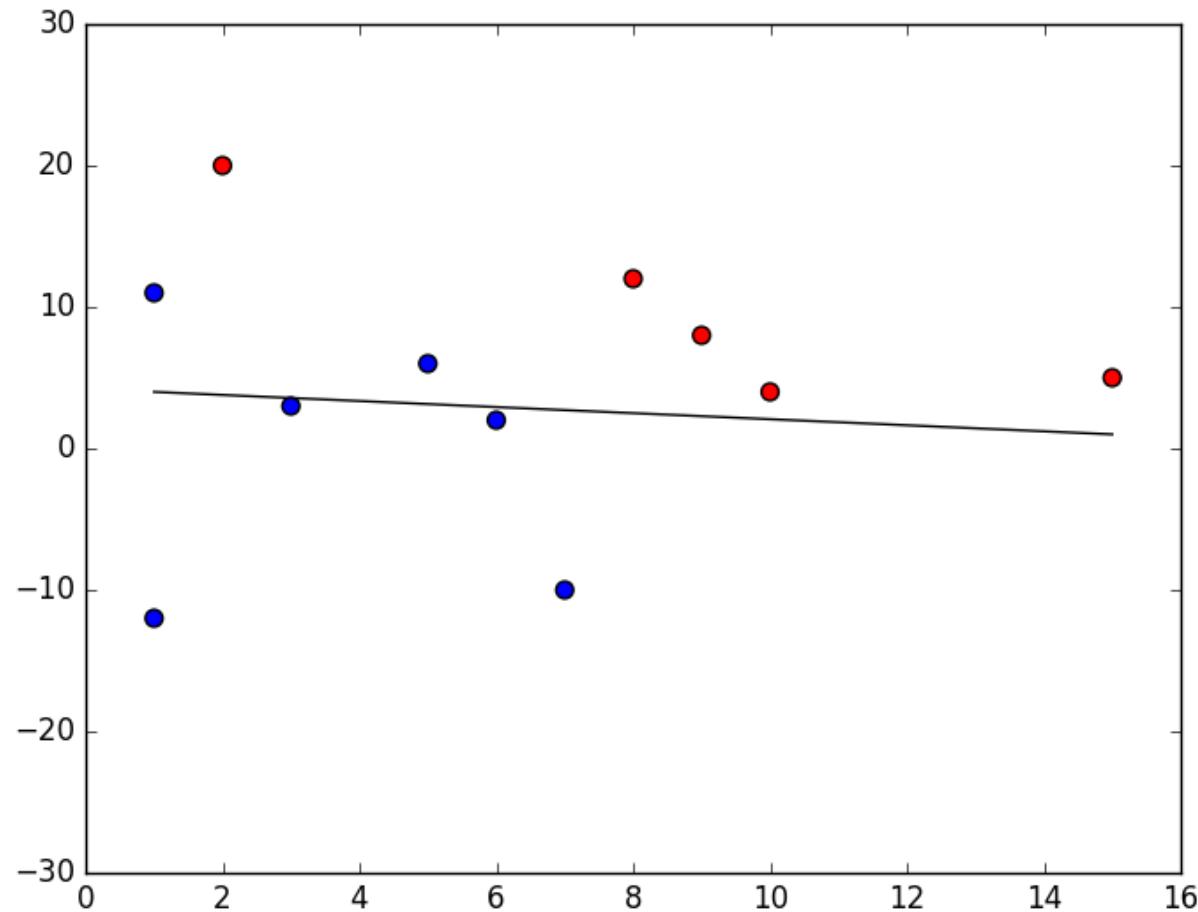
Perceptron: Example



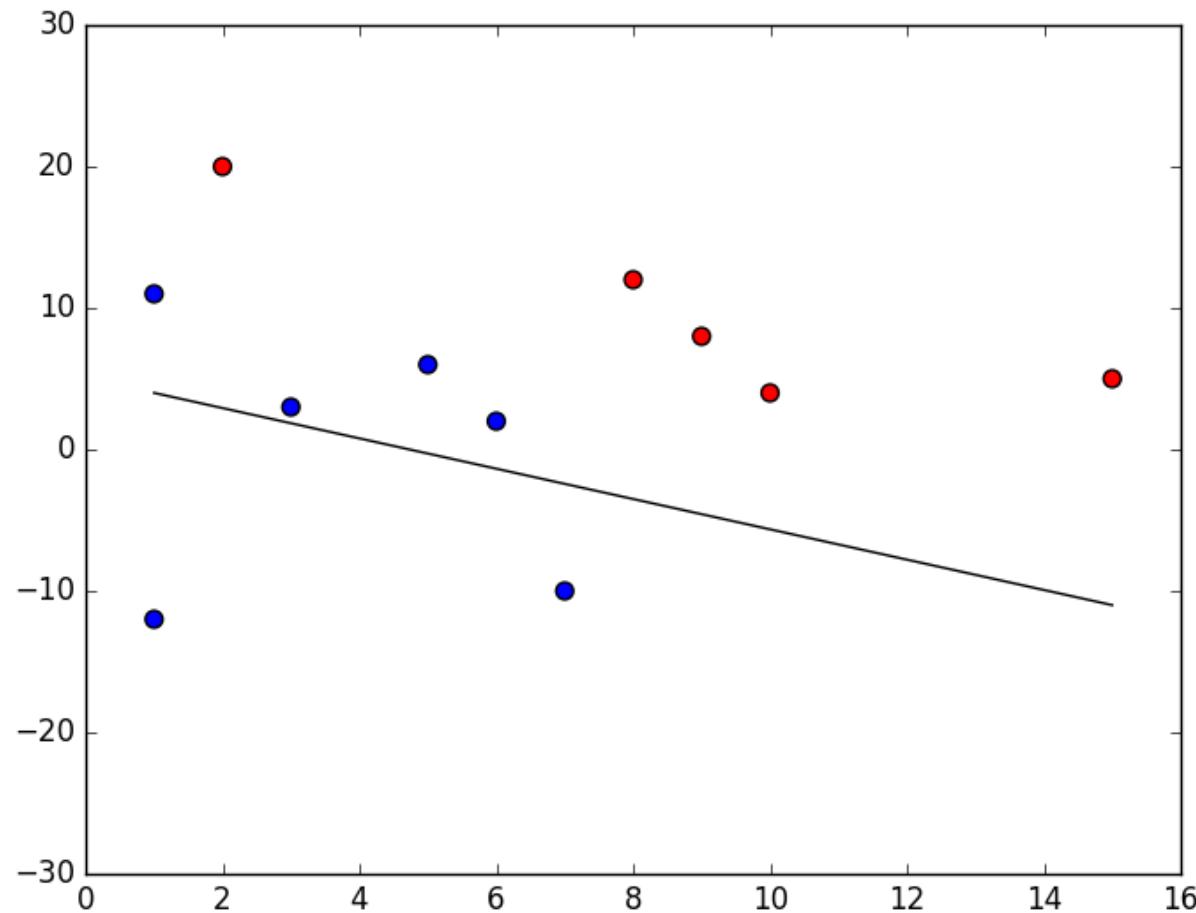
Perceptron: Example



Perceptron: Example

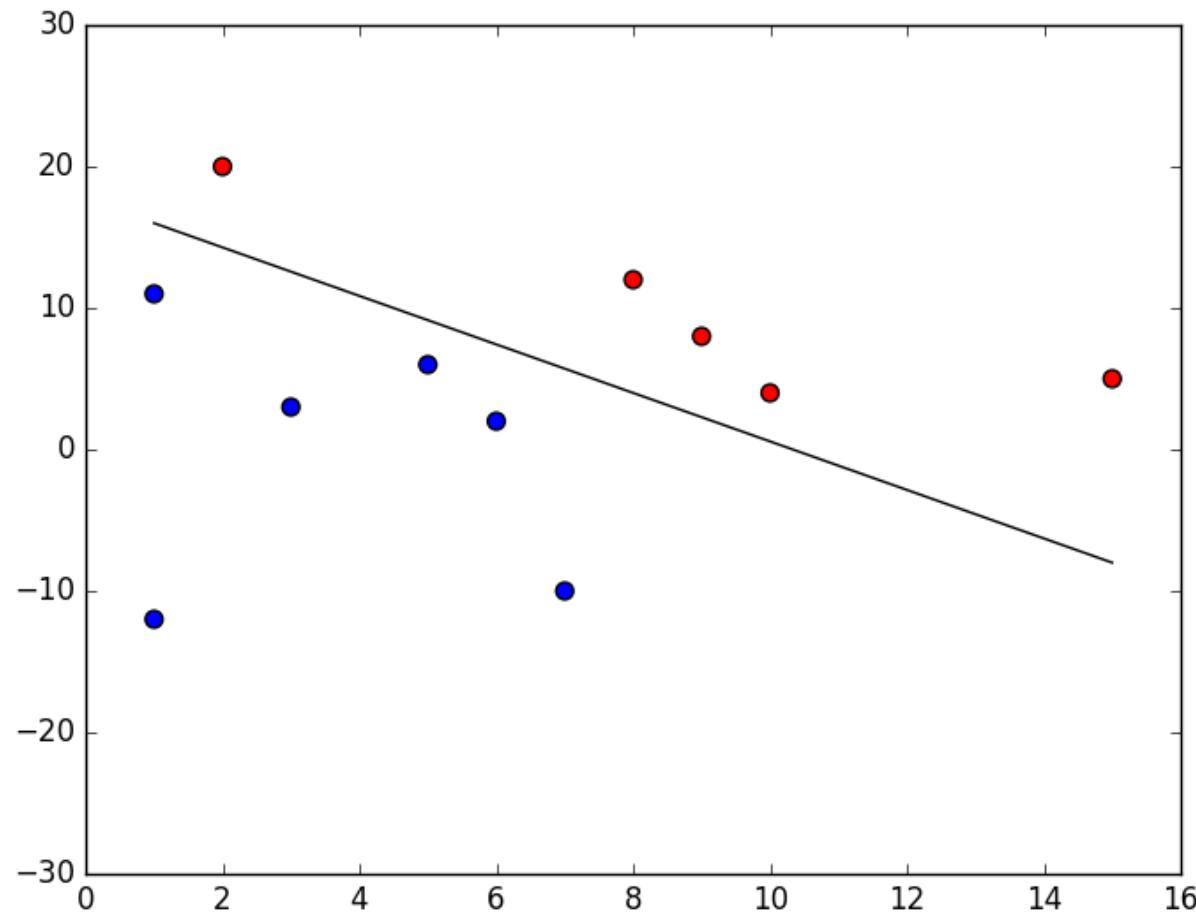


Perceptron: Example



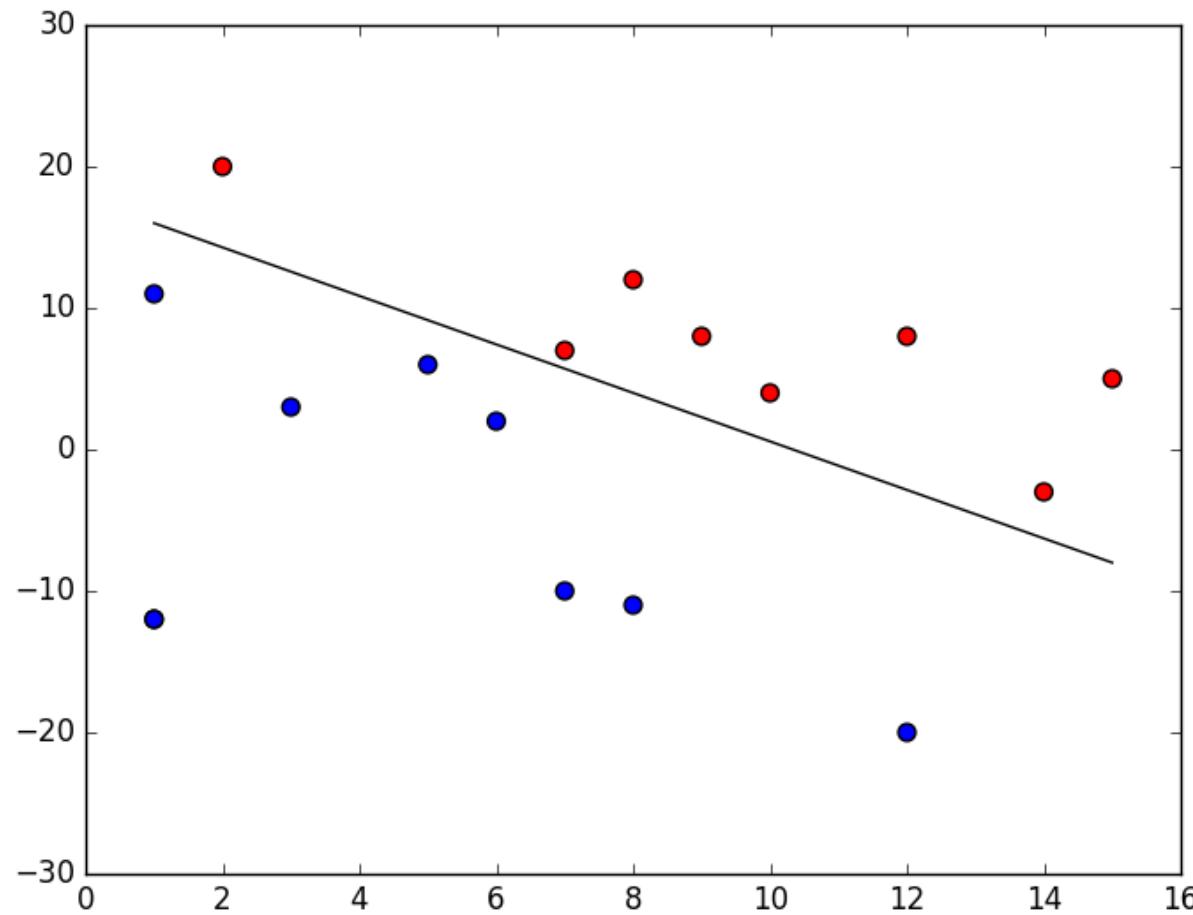
Perceptron: Example

Finally converged!



Perceptron: Example

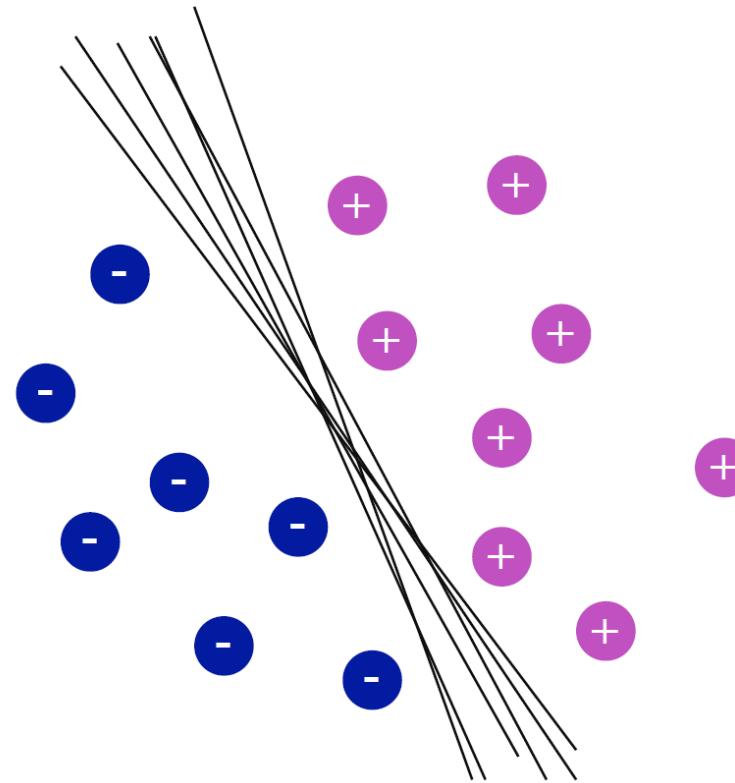
With some test data:



Perceptron

- 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.

Neural Networks

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.

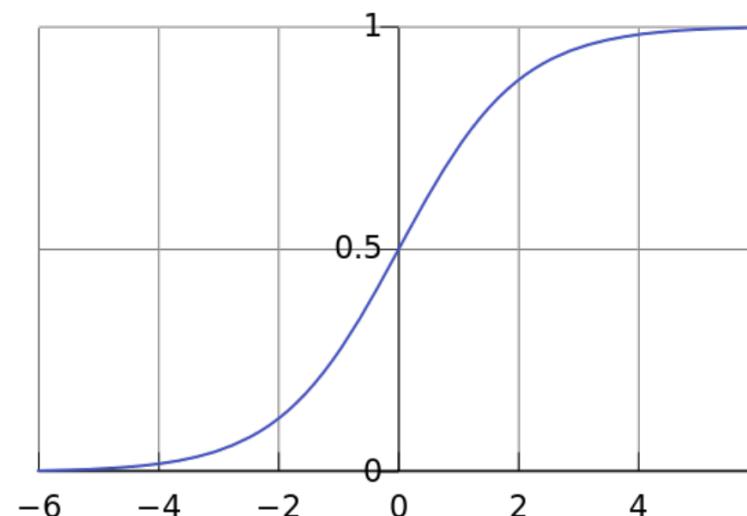
From perceptron to NN

- 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**:

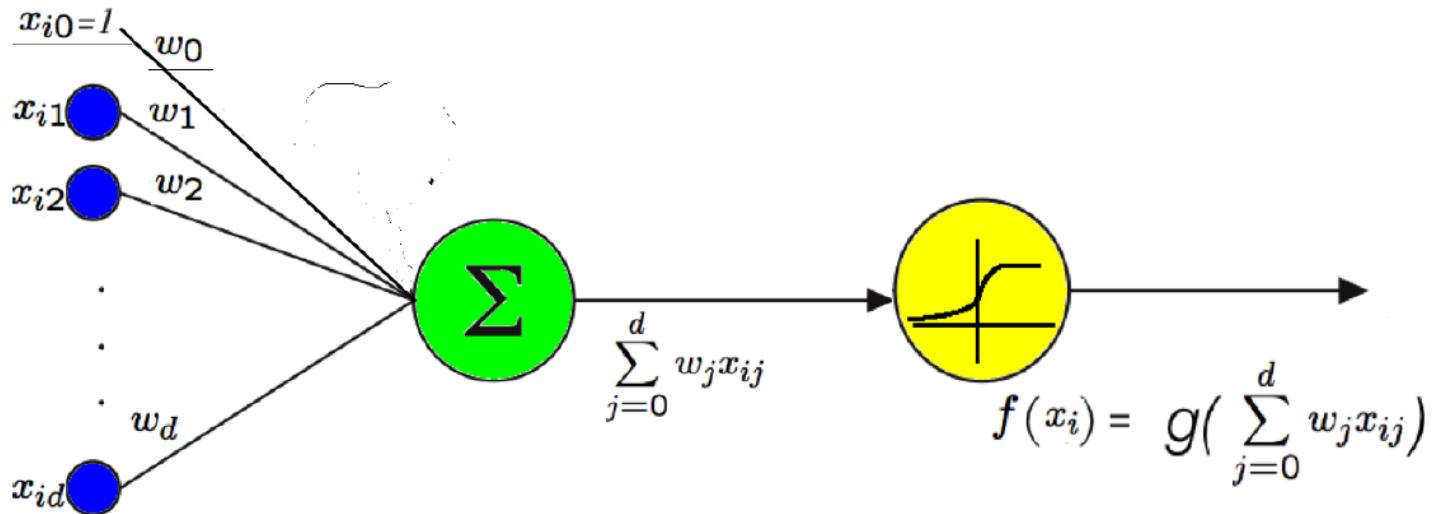
$$g(z) = \frac{e^z}{1 + e^z} = \frac{1}{1 + e^{-z}}$$

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

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



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

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

First observe:

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

$$g(10) = 0.99995$$

$$g(-10) = 0.00004$$

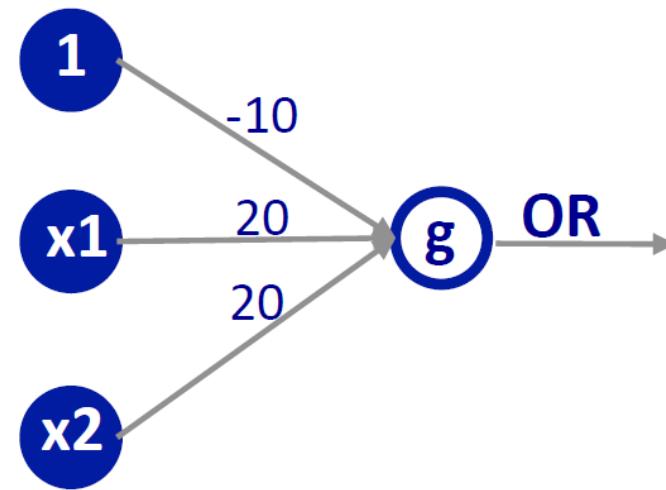
Let's consider that: For $z \geq 10$, $g(z) \rightarrow 1$. For $z \leq -10$, $g(z) \rightarrow 0$.

The XOR example

First what is the perceptron of the OR?

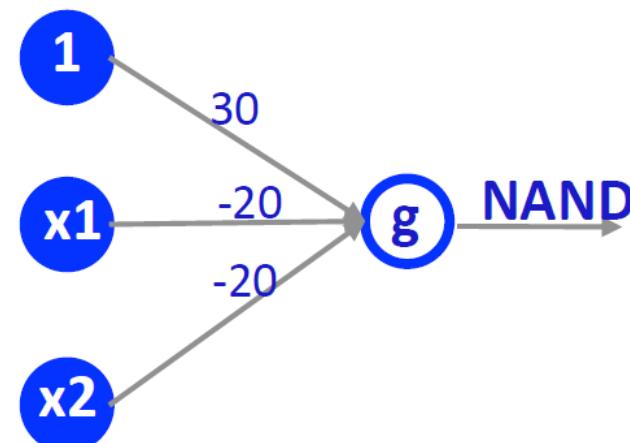
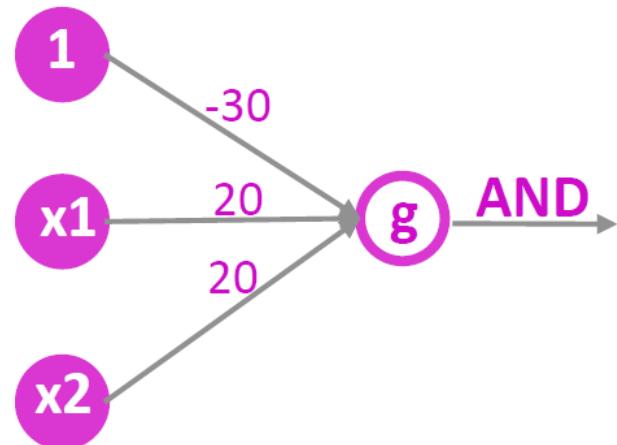
The XOR example

x_1	x_2	$x_1 \text{ OR } x_2$	$g(z)$
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	$g(30)$



The XOR example

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



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

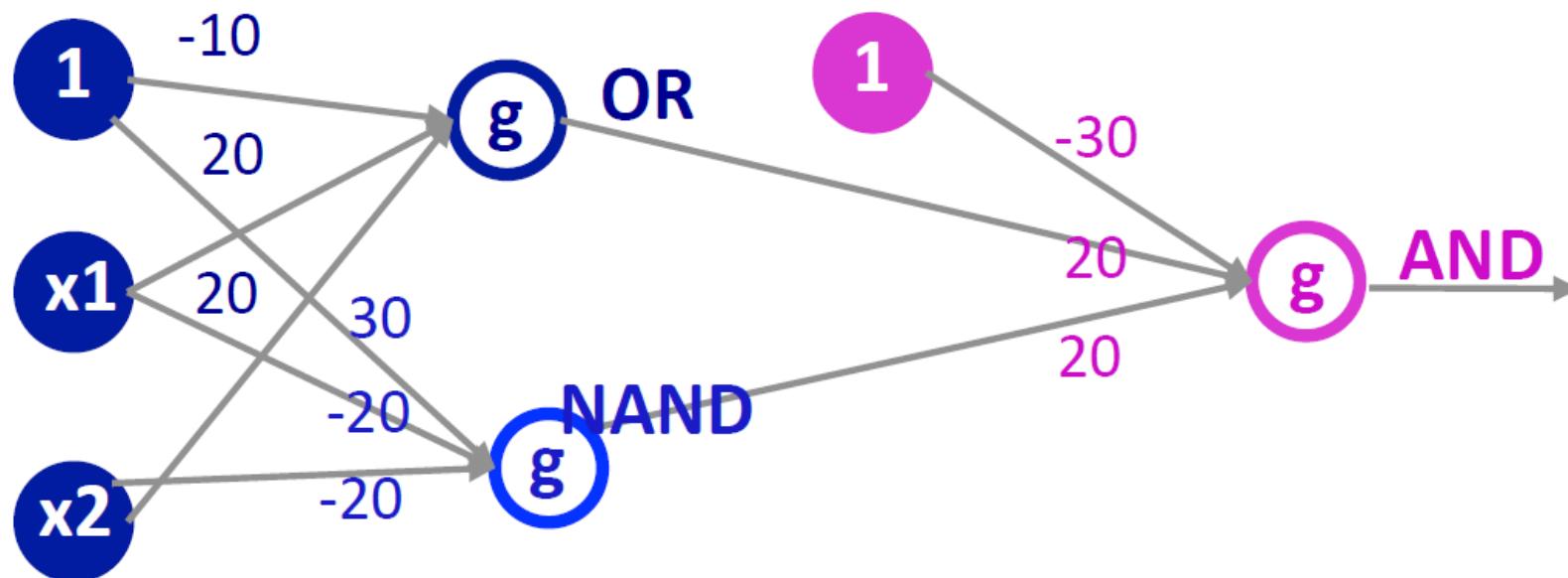
The XOR example

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

x_1	x_2	$x_1 \text{ XOR } x_2$	$(x_1 \text{ OR } x_2) \text{ AND } (x_1 \text{ NAND } x_2)$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

The XOR example

Let's put them together...

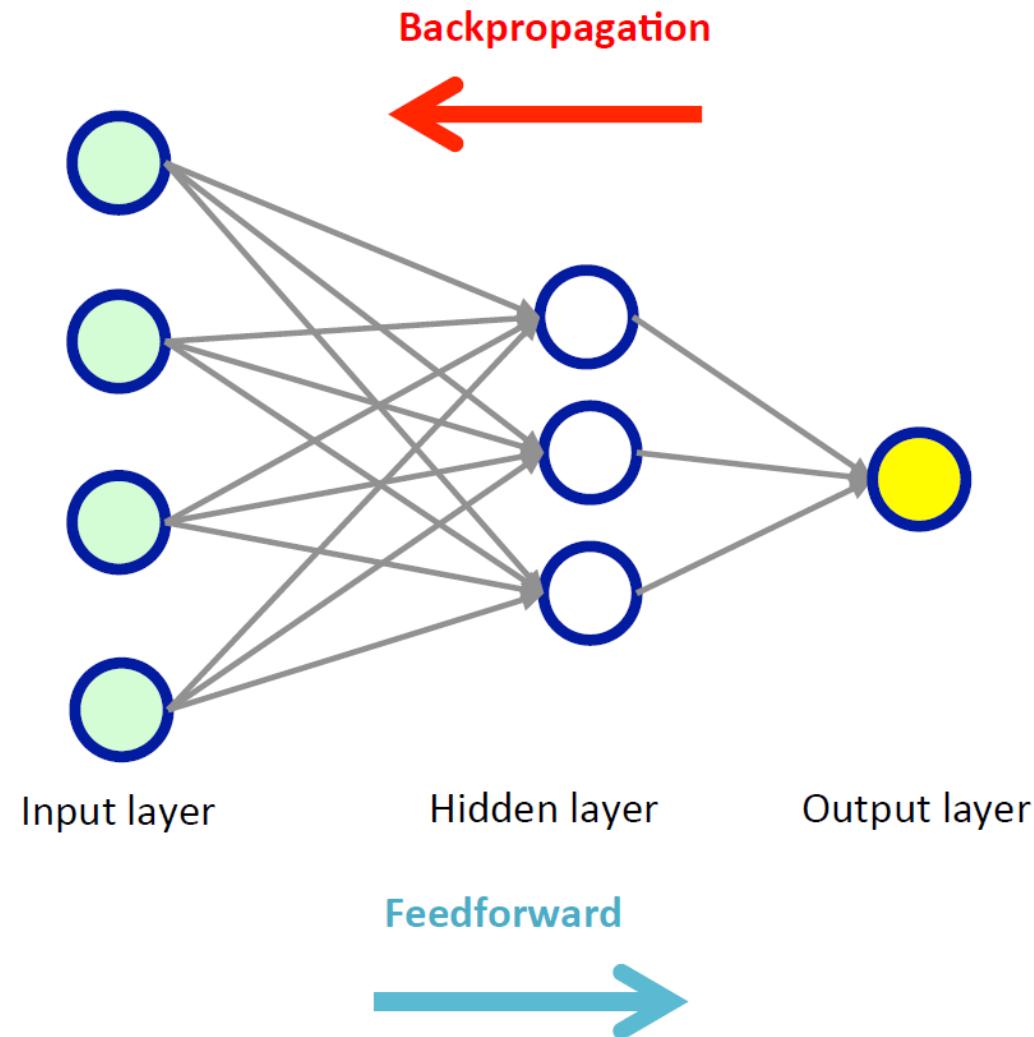


XOR as a combination of 3 basic perceptrons.

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



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.

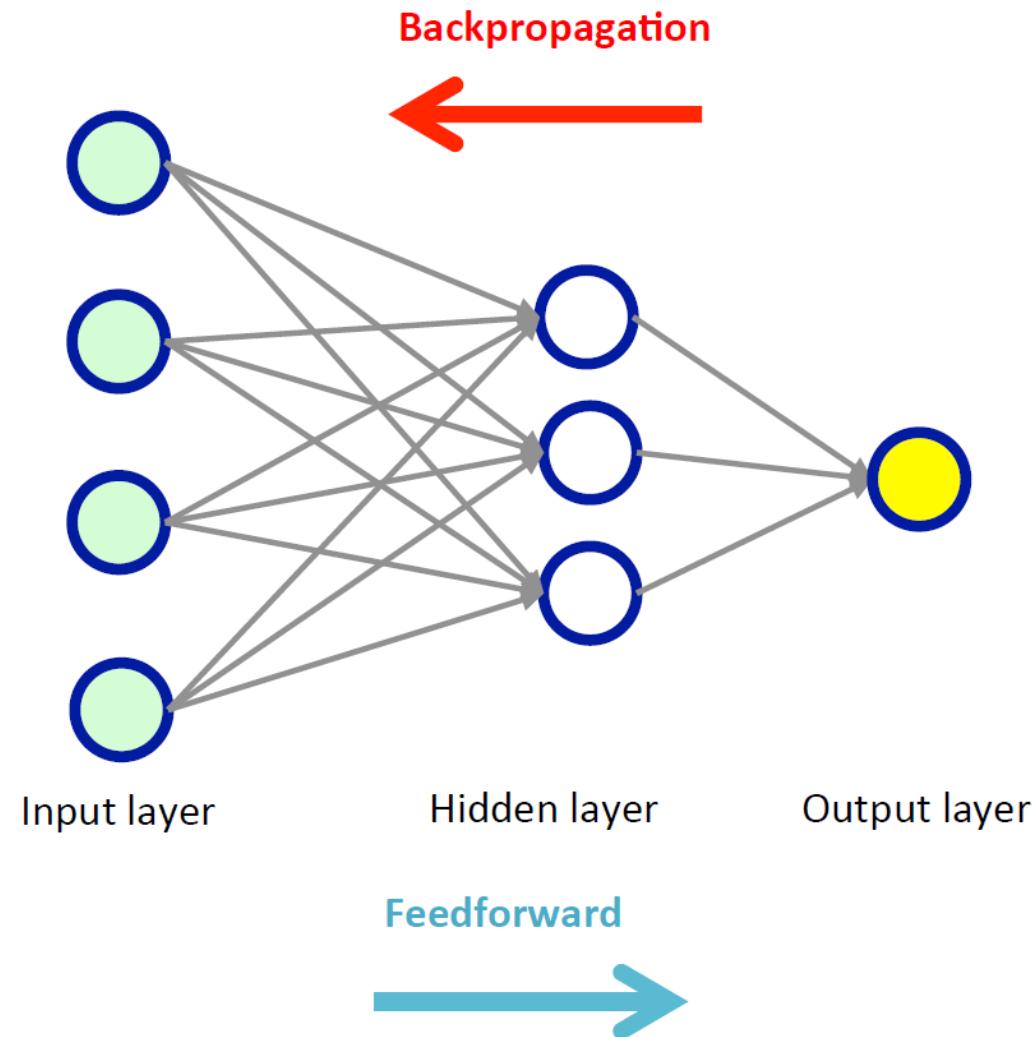
$$\Delta w_{ij} = -\alpha \frac{\partial E_e(w)}{\partial w_{ij}}$$

Backpropagation notations

Notations:

- x_{ji} : the i^{th} input to neuron j .
- w_{ji} : the weight associated with the i^{th} input to neuron j .
- $z_j = \sum w_{ji} x_{ji}$, 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 notations



Backpropagation rules

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

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

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**

Backpropagation rules

- Case 1: Neuron j is an output neuron

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

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

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

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

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

We have: $o_j = g(z_j)$

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

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

Backpropagation rules

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

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

We will note

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

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

Backpropagation rules

- Case 2: Neuron j is a hidden neuron

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

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

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

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

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

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

$$\delta_k = o_k(1 - o_k)(y_k - o_k)$$

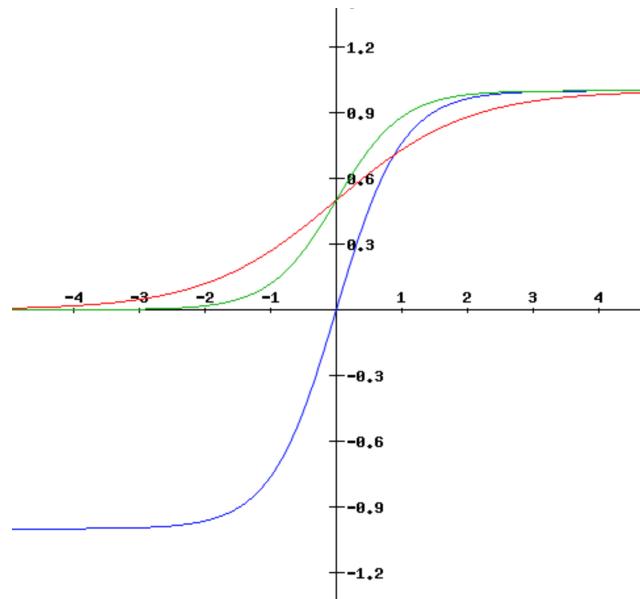
Case 2 For each hidden neuron h , calculate its error

$$\delta_h = o_h(1 - o_h) \sum_{k \in \text{Succ}(h)} w_{hk} \delta_k$$

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.

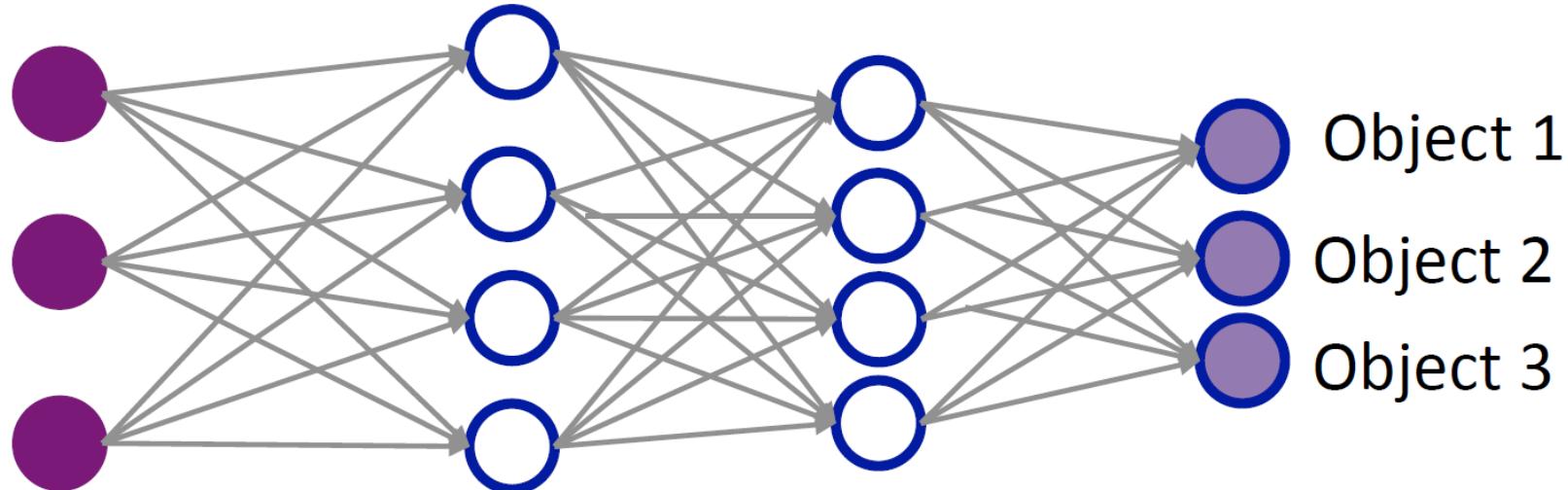


$$g(x) = \text{sigmoid}(x) = \frac{e^{kx}}{1+e^{kx}} \quad \text{for } k = 1, k = 2, \text{ etc.}$$

$$g(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

(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.

MNIST database



- The MNIST database of handwritten digits
- Training set of 60,000 examples, test set of 10,000 examples
- Vectors in \mathbb{R}^{784} (28x28 images)
- Labels are the digits they represent
- Various methods have been tested with this training set and test set
- Linear models: 7% - 12% error
- KNN: 0.5%- 5% error
- Neural networks: 0.35% - 4.7% error
- Convolutional NN: 0.23% - 1.7% error

To be continued