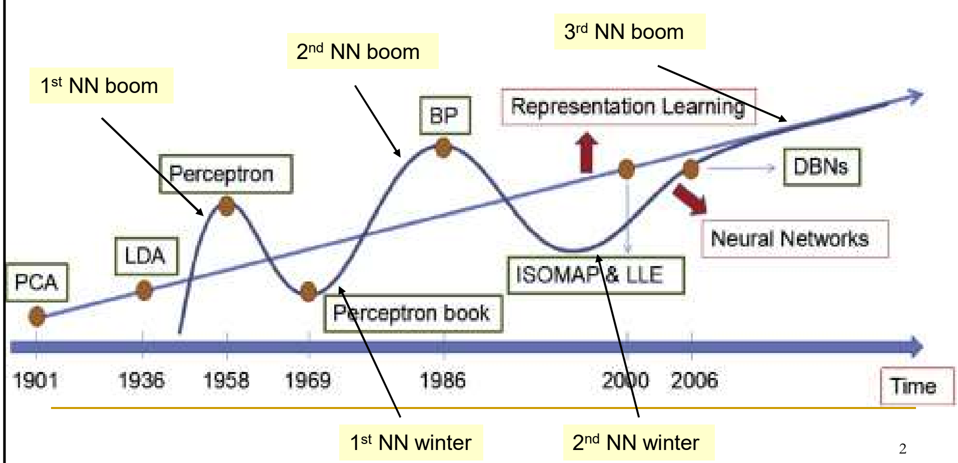
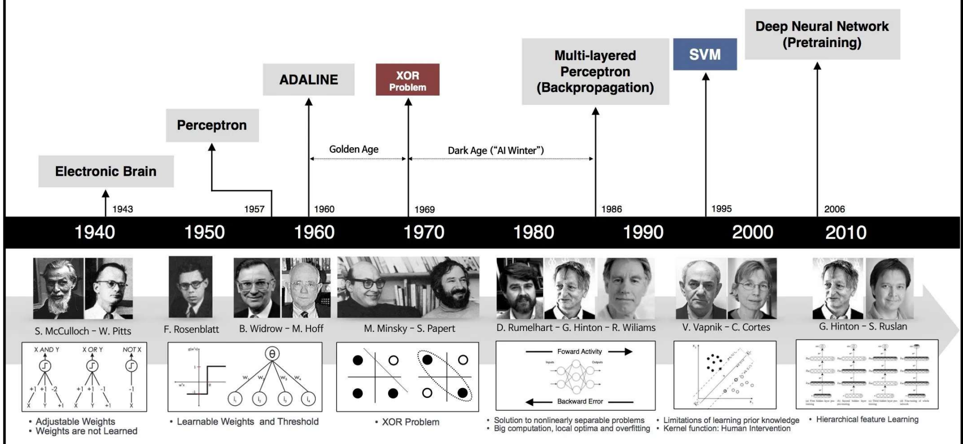
Perceptron Learning

Neural Network History (1/2)

Neural network as well as AI experienced several hype cycles.

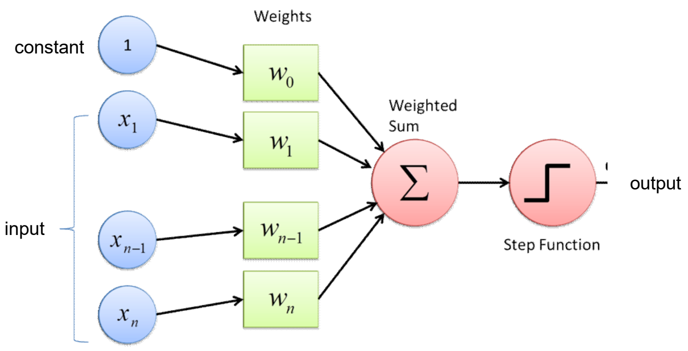


Neural Network History (2/2)



3

Perceptron Learning Rule



Perceptron learning tries to learn the weights for the network with many inputs, where it is impossible to visualize the decision boundaries. But the perceptron learning can be seen as a kind of error correction learning.

Error Correction Learning

Prepare the input signals and the teacher signal

$x_{p1}, x_{p2}, \ldots, x_{pm}$: the pth input signals

y_p : the pth teacher signal

where *m* is the number of inputs, *P* is the number of data set.

- For the cases of OR function (or AND function), *m* = 2, and *P*=4.
- For such cases, the teacher signal is given. It is called supervised learning.
- If the teacher signal is not given, it is called unsupervised learning.

Error Correction Learning (cont'd)

For a given input signal, the weight and the threshold are adjusted, based on the error between the teacher signal and the output of the neuron, by

$$w_i^{new} = w_i^{old} + \eta(y_p - out)x_{pi} \quad (1 \le i \le m)$$

where, η<1 is learning rate with a small positive value, and (y_p - *out*) is usually given by

$$(y_p - out) = \begin{cases} 1 & \text{when} \quad y_p = 1, out = 0 \\ -1 & \text{when} \quad y_p = 0, out = 1 \\ 0 & \text{when} \quad y_p = out \end{cases}$$

Only when the output and the teacher signal are different, the weight is corrected.

Error Correction Learning (an example)

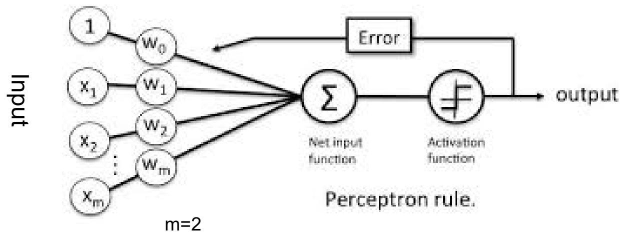
AND function

x_1	x_2	y_p
0	0	0
0	1	0
1	0	0
1	1	1

For linear separable problem, perceptron learning rule guarantees to obtain the solution.

Perceptron

$$w_i^{new} = w_i^{old} + \eta(y_p - out)x_{pi} \quad (1 \le i \le m)$$



m=2

Gradient Descent Method

Loss Optimization

We want to find the network weights that ***achieve the lowest loss***

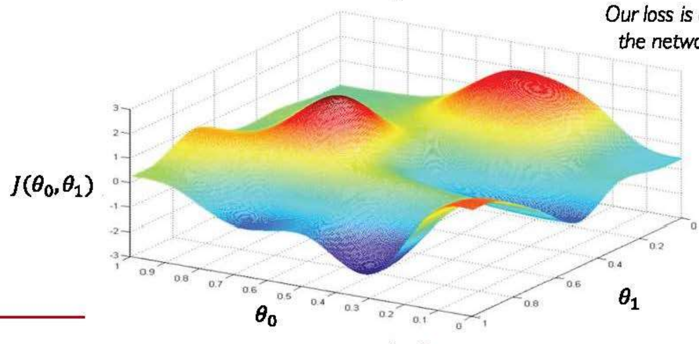
$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{\theta}\right), y^{(i)}\right)$$

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, J(\boldsymbol{\theta})$$

Loss Optimization

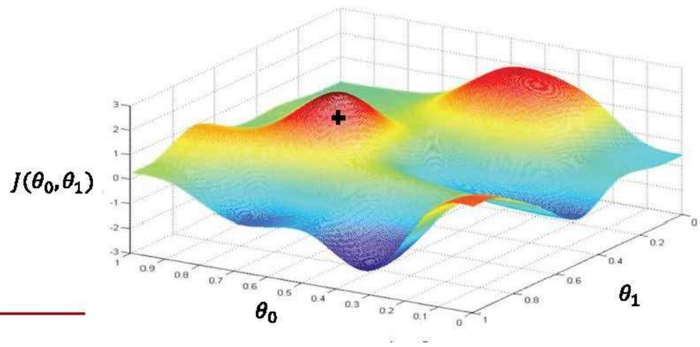
$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, J(\boldsymbol{\theta})$$

Remember:
Our loss is a function of the network weights!



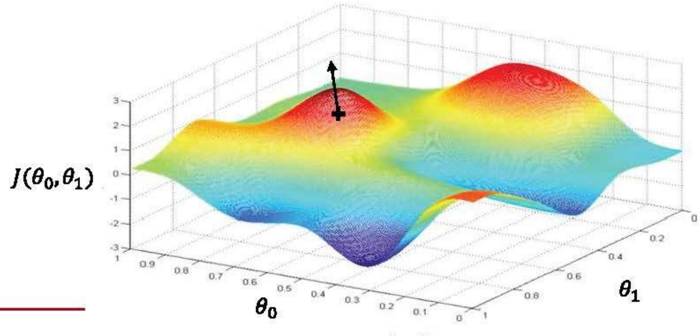
Loss Optimization

Randomly pick an initial (θ_0, θ_1)



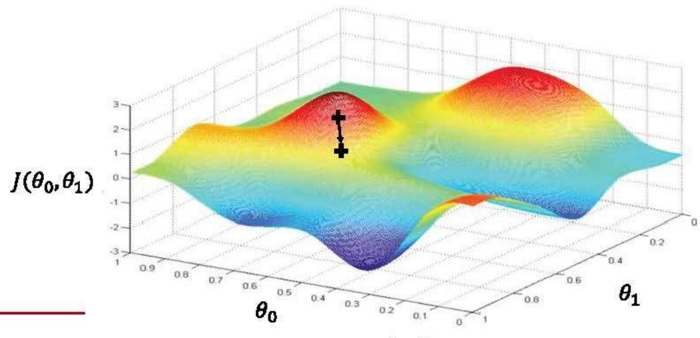
Loss Optimization

Compute gradient, $\frac{\partial J(\theta)}{\partial \theta}$



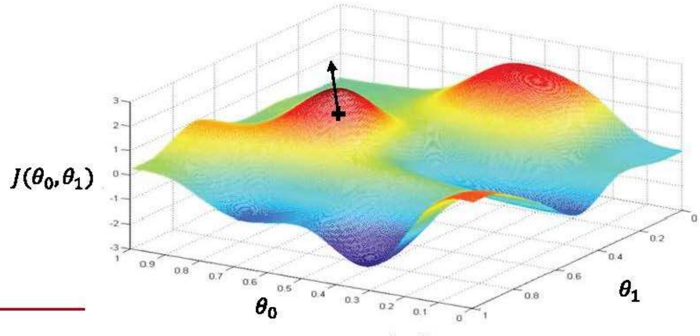
Loss Optimization

Take small step in opposite direction of gradient



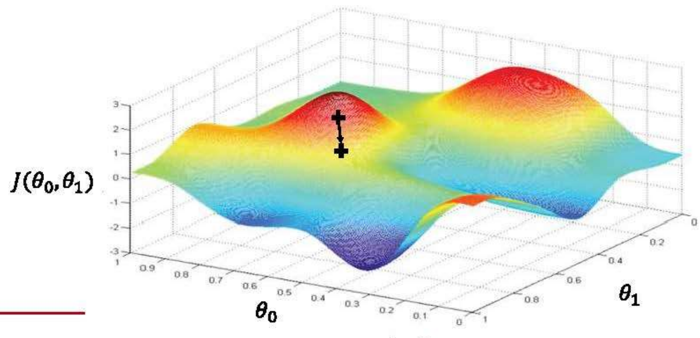
Loss Optimization

Compute gradient, $\frac{\partial J(\theta)}{\partial \theta}$



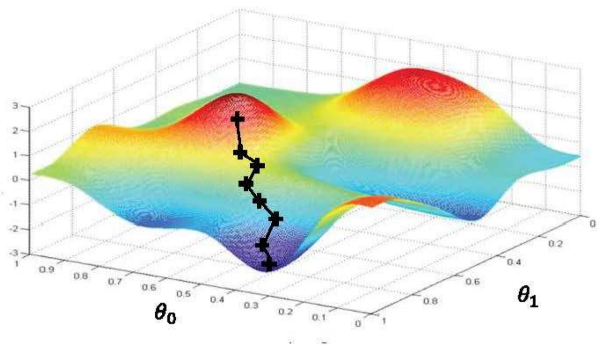
Loss Optimization

Take small step in opposite direction of gradient



Gradient Descent

Repeat until convergence



Gradient Descent

Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\theta)}{\partial \theta}$
4. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$
5. Return weights