Lecture 10: Neural Networks

Presentation · July 2019

1 author:

Alaa Tharwat
Fachhochschule Bielefeld

Lecture 10: Neural Networks

Alaa Tharwat

Lecture 10: Neural Networks

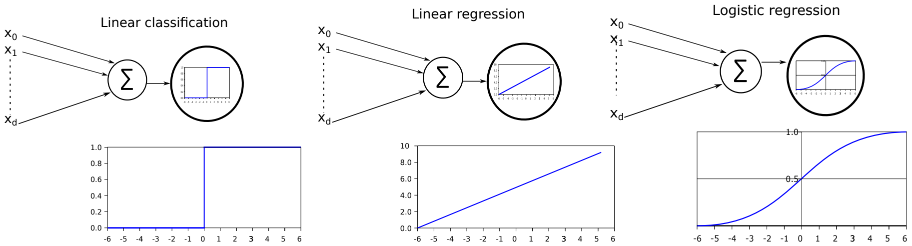
- Review of Lecture 9
- Neural networks model
- Backpropagation algorithm

Lecture 10: Neural Networks

- Review of Lecture 9
- Neural networks model
- Backpropagation algorithm

Nonlinear transformation

- $d_{VC} \uparrow$ Higher $\tilde{d}$: better chance of being linearly separable ($E_{in} \downarrow$)
- $d_{VC} \downarrow$ Higher $\tilde{d}$: possibly not linearly separable ($E_{in} \uparrow$)



Sigmoid function

$$h(x) = \theta(s) = \frac{e^s}{e^s + 1} = \frac{1}{1 + e^{-s}} = \frac{1}{1 + e^{-(\mathbf{w}.\mathbf{x})}}$$

$$P(y|\mathbf{x}) = \begin{cases} h(\mathbf{x}) & \text{for } y = +1 \\ 1 - h(\mathbf{x}) & \text{for } y = -1 \end{cases}$$

$$E_{in} = \frac{1}{N} \sum_{n=1}^{N} ln(1 + e^{-y_n \mathbf{w}^T \mathbf{x}_n}) \rightarrow \textbf{Cross-entropy error}$$

The weights are adjusted using Gradient descent algorithm

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \bigtriangledown E_{in}$$

- In Batch gradient descent or simply GD, the gradient is computed for the error on the whole data

$$E_{in}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} e(h(\mathbf{x}_i), y_i)$$

- The weights are updated as follows, $\Delta\mathbf{w} = -\eta \bigtriangledown E_{in}(\mathbf{w})$
- SGD is a randomized version of GD. In SGD, we pick one example (e.g. $(\mathbf{x}_n, y_n)$) and apply GD

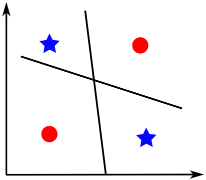
$$E_n[- \bigtriangledown e(h(\mathbf{x}_n), y_n)] = \frac{1}{N}\sum_{i=1}^{N} - \bigtriangledown e(h(\mathbf{x}_i), y_i) = - \bigtriangledown E_{in}$$

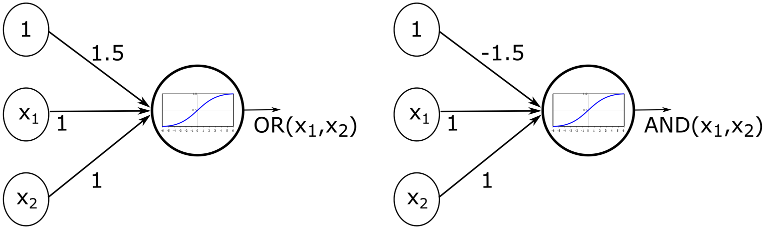
- SGD is
  - simple
  - computationally cheaper
  - random

Lecture 10: Neural Networks

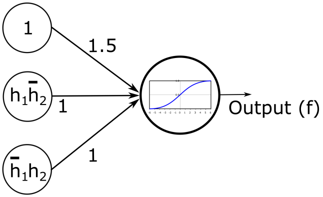
- Review of Lecture 9
- Neural networks model
- Backpropagation algorithm

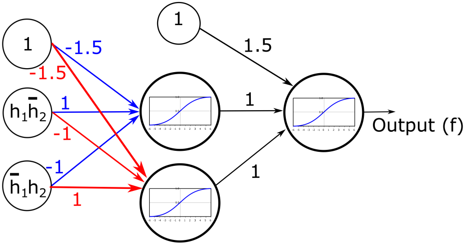
Combining perceptrons

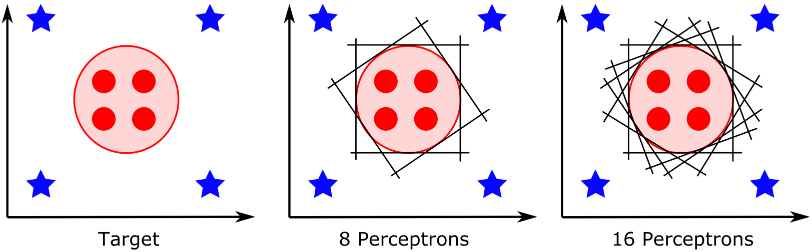




How to create layers?

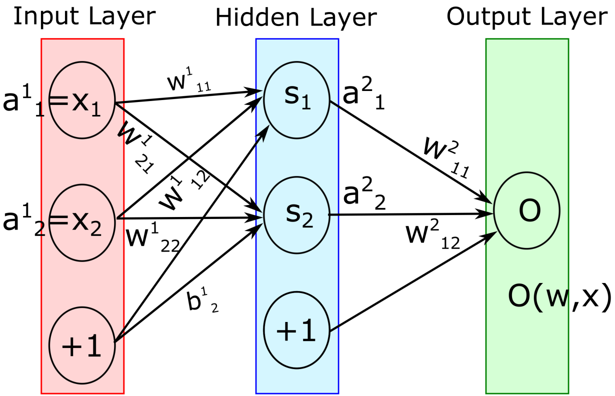





Target
8 Perceptrons
16 Perceptrons

- Input layer ($l = 0$): inputs $(x_1, x_2, \ldots, x_m)$
- Hidden layers ($1 \leq l \leq L$), L is the number of layers
- Output layer ($l = L$)

$$a = \theta(s) = \theta(\mathbf{w}.\mathbf{x} + b)$$



$$a = \theta(s) = \theta(\mathbf{w}.\mathbf{x} + b)$$

where

- $b = -$threshold and the threshold and weights are parameters of the neuron. Therefore, small changes in weights or bias cause a small change in the output

$$\begin{aligned} \mathbf{x} &= [x_1, x_2, \dots, x_n, +1] \\ \mathbf{w} &= [w_1, w_2, \dots, w_n, w_{n+1}] \end{aligned} \tag{1}$$

- $s = \sum \text{weight . input} + \text{bias} = \mathbf{w}^T.\mathbf{x} + b = \sum_{i=1}^{n+1} w_i x_i = \mathbf{w}^T\mathbf{x}$ is the weighted sum of the neuron's inputs plus the bias term. Hence, the value of s ranged from $-\infty$ to $+\infty$.
- θ is the *activation function*. There are different functions that can be used as activation functions, these activation functions are used for calculating the output of the perceptron.

$$w_{ij}^{l} = \begin{cases} 1 \leq l \leq L & \text{layers} \\ 0 \leq i \leq d^{(l-1)} & \text{inputs} \\ 1 \leq j \leq d^{(l)} & \text{outputs} \end{cases}$$

$$x_j^{(l)} = \theta(a_j^{(l)}) = \theta(\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)})$$

Activation functions

- **Step function**: the simplest activation function and the neuron is fired if s exceeds a certain threshold, however, getting intermediate values for the activation output makes the learning process smoother.

$$\theta(s) = \theta(\mathbf{w}.\mathbf{x} + b) = \begin{cases} 0 & \text{if} \sum_i w_i x_i + b \leq 0 \\ 1 & \text{if} \sum_i w_i x_i + b > 0 \end{cases} \quad (2)$$

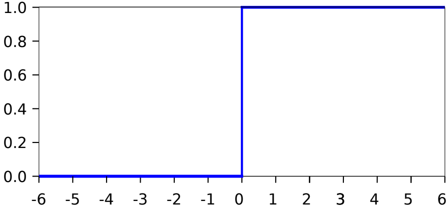
- **Linear function**: In this function ($s = \mathbf{w}.\mathbf{x}$), the activation function is proportional with the inputs. Adjusting the weights and bias depends on $\theta'(s)$ which is in this function is a constant (**w**). Hence, the gradient has no relationship with the inputs and the gradient will be a constant gradient.

- **Sigmoid function**: sigmoid function produces zero or one depending on the value of $\mathbf{w}.\mathbf{x} + b$. A small change in weights or bias will produce a small change in the output. Towards either end of the sigmoid function, the value of s makes a very small change in $s = \theta(s)$ and this is called *vanishing gradients*.

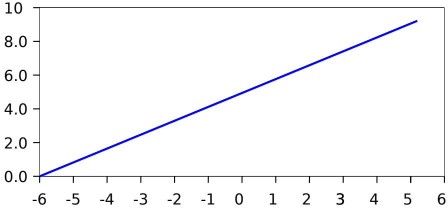
$$\theta(s) = \frac{e^s}{e^s + 1} = \frac{1}{1 + e^{-s}} = \frac{1}{1 + e^{-(\mathbf{w}.\mathbf{x}+b)}} \tag{3}$$

- **Tanh function**: the *Tanh* function is very similar to the sigmoid function, but, the gradient is stronger for Tanh function than sigmoid. However, the Tanh function still has the vanishing gradient problem.

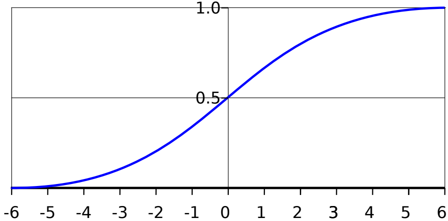
$$\theta(s) = tanh(s) = \frac{2}{1 + e^{-2s}} - 1 = 2\text{sigmoid}(2s) - 1 \tag{4}$$



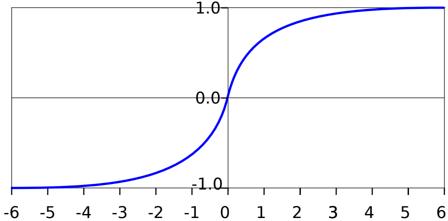
(a) Step function



(b) Linear function



(c) Sigmoid function



(d) Tanh function

Lecture 10: Neural Networks

- Review of Lecture 9
- Neural networks model
- Backpropagation algorithm

- In BPNN, SGD is used, and the gradient is

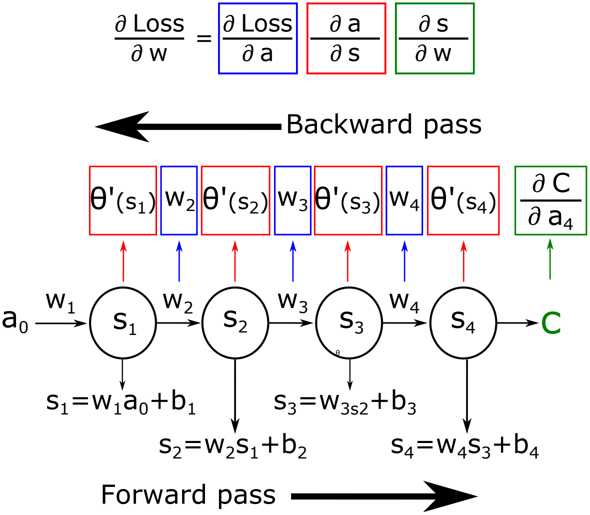
$$\triangledown e(\mathbf{w}) = \frac{\partial e(\mathbf{w})}{\partial w_{ij}^{(l)}}$$

where $\mathbf{w} = \{w_{ij}^{(l)}\}$

$$\frac{\partial e(\mathbf{w})}{\partial w_{ij}^{(l)}} = \frac{\partial e(\mathbf{w})}{\partial s_j^{(l)}} \times \frac{\partial s_j^{(l)}}{\partial w_{ij}^{(l)}}$$

$$\frac{\partial s_j^{(l)}}{\partial w_{ij}^{(l)}} = x_i^{(l-1)}$$

The error in the final layer ($l = L$ and $j = 1$) is $\delta_1^{(L)} = \frac{\partial e(\mathbf{w})}{\partial s_1^{(L)}}$, where $e(\mathbf{w}) = (x_1^{(L)} - y_n)^2$, $x_1^{(L)} = \theta(s_1^{(L)})$, and $\theta'(s) = 1 - \theta^2(s)$ (tanh function)



$$
\begin{aligned}
\delta_i^{(l-1)} &= \frac{\partial e(\mathbf{w})}{\partial s_i^{(l-1)}} \\
&= \sum_{j=1}^{d(l)} \frac{\partial e(\mathbf{w})}{\partial s_j^{(l)}} \times \frac{\partial s_j^{(l)}}{\partial x_i^{(l-1)}} \times \frac{\partial x_i^{(l-1)}}{\partial s_i^{(l-1)}} \\
&= \sum_{j=1}^{d(l)} \delta_j^{(l)} \times w_{ij}^{(l)} \times \theta'(s_i^{(l-1)}) \\
&= (1 - (x_i^{(l-1)})^2) \sum_{j=1}^{d(l)} w_{ij}^{(l)} \delta_j^{(l)}
\end{aligned}
$$

Algorithm 1 : The backpropagation algorithm.

```
Initialize all weights (w_ij^(l)) randomly
for all t = 0, 1, ... do
    Pick n ∈ {1, 2, ..., N}
    Forward: compute all x_j^(l)
    Backward: compute all δ_j^(l)
    Update weights w_ij^(l) ← w_ij^(l) − η x_i^(l−1) δ_j^(l)
    Iterate until reaching stopping conditions
end for
```

Initialize all weights $(w_{ij}^{(l)})$ randomly

for all $t = 0, 1, \ldots$ **do**

- Pick $n \in \{1, 2, \ldots, N\}$
- Forward: compute all $x_j^{(l)}$
- Backward: compute all $\delta_j^{(l)}$
- Update weights $w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta x_i^{(l-1)} \delta_j^{(l)}$
- Iterate until reaching stopping conditions

end for