Linear Classifier, Perceptron, and MLP

Deep Learning 2021

E. Fatemizadeh

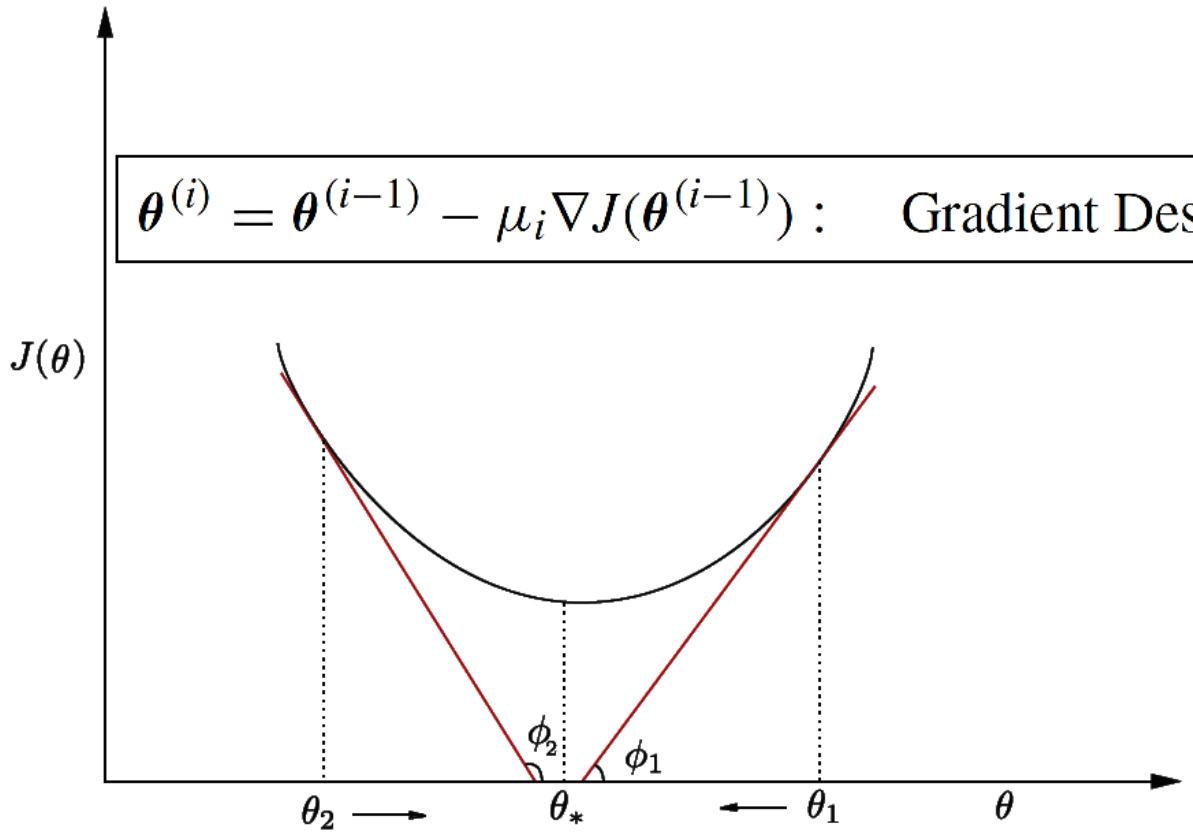
Steepest (Gradient) Descent Algorithm

› A way to solve unconstraint convex optimization (minimization):

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

i: *iteration step*
μ_i: *step size*

$$\boldsymbol{\theta}^{(i)} = \boldsymbol{\theta}^{(i-1)} - \mu_i \nabla J(\boldsymbol{\theta}^{(i-1)}) : \quad \text{Gradient Descent Scheme.}$$



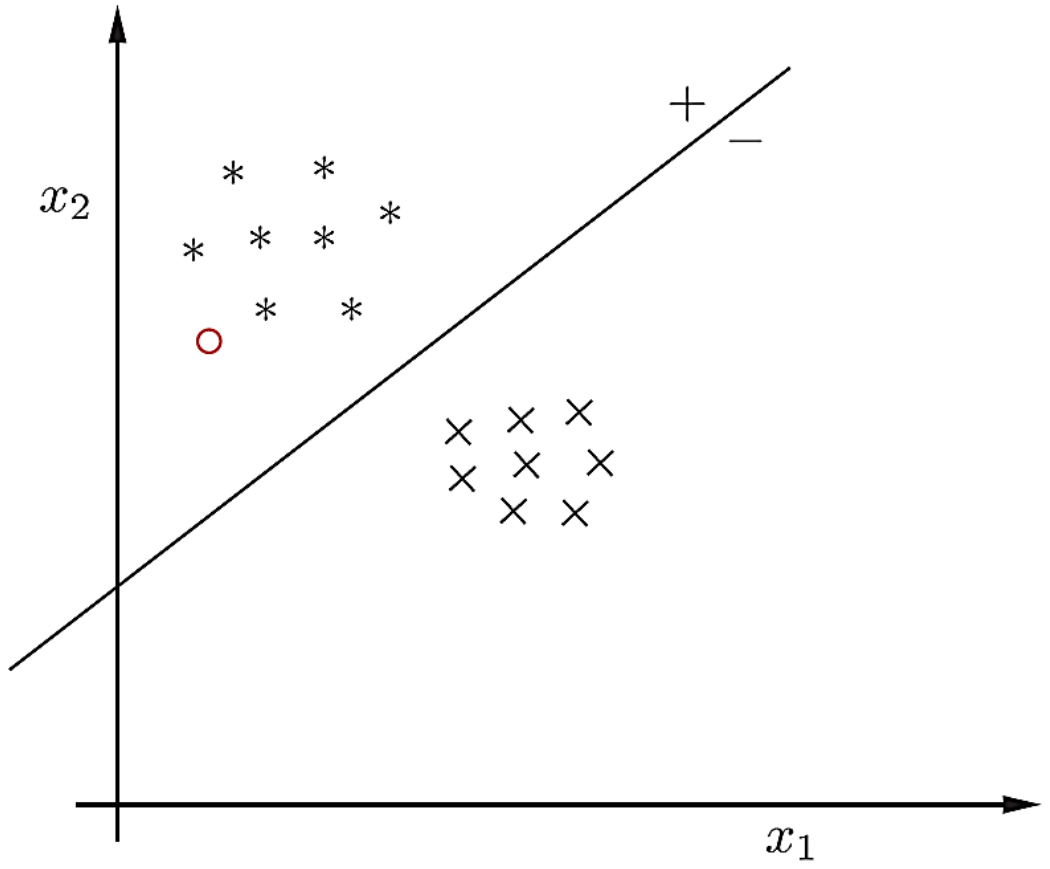
Steepest (Gradient) Descent Algorithm

- GD variations:
- $\mu_i = \mu_0$
- $\mu_i = \frac{\mu_0}{i}$
- $\mu_i = \frac{\mu_0}{1+i/T}$
- $\mu_i = \mu_0 \frac{|\nabla J(\theta^{(i-1)})|}{1+|\nabla J(\theta^{(i-1)})|^2}$
- …. (We will discuss more)

$$\theta^{(i)} = \theta^{(i-1)} - \mu_i \nabla J(\theta^{(i-1)}) : \quad \text{Gradient Descent Scheme.}$$

Perceptron Algorithm

- First consider linear separable problem (two classes):
- Data: $\{x_n, y_n\}_{n=1}^{N}$
- $x_n \in \mathbb{R}^l, (for\ example\ l = 2)$
- $y_n \in \{-1, +1\}$
- $g(x) = \theta_1 x_1 + \theta_2 x_2 + \theta_0$
- $g(x) = w_1 x_1 + w_2 x_2 + w_0$
- $g(x) = \theta_1 x_1 + \theta_2 x_2 + \theta_0 \times 1$



Perceptron Algorithm

- General case:
- Data: $\{x_n, y_n\}_{n=1}^{N}$
- $g(x) = \sum_{i=1}^{l} \theta_i x_i + \theta_0 = \boldsymbol{\theta}^T \tilde{\boldsymbol{x}}$
- $\tilde{\boldsymbol{x}} = (x_1 \quad x_2 \quad \dots \quad x_D \quad 1)^T : extened\ data\ vector$
- $\boldsymbol{\theta} = (\theta_1 \quad \theta_2 \quad \dots \quad \theta_D \quad \theta_0)^T$
- $\{\theta_i\}_{i=1}^{l} : weights$
- $\theta_0 = bias\ or\ threshold$

Perceptron Algorithm

› General formulation:

$$g(x) = \boldsymbol{\theta}^T \boldsymbol{x} \text{ or } g(x) = \boldsymbol{w}^T \boldsymbol{x}$$

› Optimal Classifier is a hyperplane:

$$\boldsymbol{\theta}_*^T \boldsymbol{x} > 0, \quad \text{if } \boldsymbol{x} \in \omega_1,$$
$$\boldsymbol{\theta}_*^T \boldsymbol{x} < 0, \quad \text{if } \boldsymbol{x} \in \omega_2.$$

› Our Goal: Developing an algorithm that iteratively computes a hyperplane that classifies correctly all the patterns from both classes.

Perceptron Algorithm – Cost Function

- Input Data: $\{x_n, y_n\}_{n=1}^{N},\ y_n \in \{-1, +1\}$
- Assume $\mathcal{Y}$ be the set of all *misclassified* samples.
- Cost is defined as:

$$J(\boldsymbol{\theta}) = -\sum_{n:x_n \in \mathcal{Y}} y_n \boldsymbol{\theta}^T \boldsymbol{x}_n : \quad \text{Perceptron Cost,}$$

$$y_n = \begin{cases} +1, & \text{if } \boldsymbol{x} \in \omega_1, \\ -1, & \text{if } \boldsymbol{x} \in \omega_2. \end{cases}$$

- Observe that (for summation over $\mathcal{Y}$) the cost function is nonnegative.
- The cost function becomes zero, if there are no misclassified points, that is, $\mathcal{Y} = \emptyset$, which corresponds to a perfect solution.

Perceptron Algorithm – Cost Function

› Perceptron Algorithm (Gradient Descent):

$$\frac{\partial\left(-\sum_{n;x_n\in\mathcal{Y}} y_n \boldsymbol{\theta}^T \boldsymbol{x}_n\right)}{\partial \boldsymbol{\theta}} = -\sum_{n;x_n\in\mathcal{Y}} y_n \boldsymbol{x}_n$$

› Then:

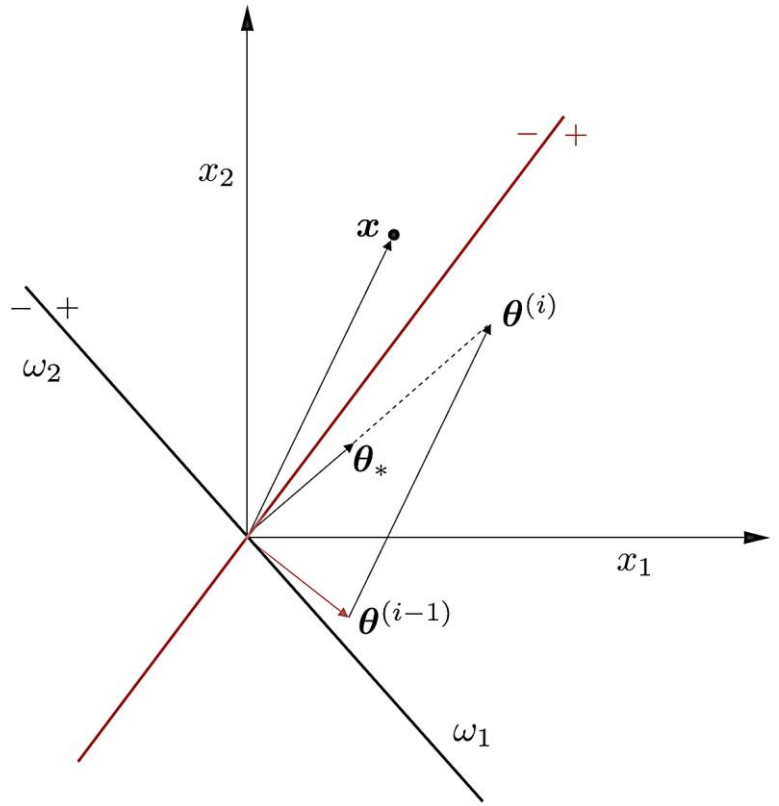
$$\boldsymbol{\theta}^{(i)} = \boldsymbol{\theta}^{(i-1)} + \mu_i \sum_{n:\boldsymbol{x}_n\in\mathcal{Y}} y_n \boldsymbol{x}_n : \quad \text{The Perceptron Rule,}$$

Perceptron Algorithm – Batch Mode!

- Choose $\boldsymbol{\theta}^{(0)}$ randomly or $\mathbf{0}$
- Choose $\mu_{(0)}$
- set $i = 0$
- Repeat
  - $\mathcal{Y} = \emptyset$
  - for $n = 1$ to N, if $y_n \boldsymbol{\theta}^T \boldsymbol{x}_n < 0$ then $\mathcal{Y} = \mathcal{Y} \cup \{\boldsymbol{x}_n\}$, end;
  - $i \leftarrow i + 1$
  - $\boldsymbol{\theta}^{(i)} = \boldsymbol{\theta}^{(i-1)} + \mu_{(i)} \sum_{n; x_n \in \mathcal{Y}} y_n \boldsymbol{x}_n$
  - Update $\mu_{(i)}$
- Until $\mathcal{Y} = \emptyset$

Perceptron Algorithm – Batch Mode!

› How it works! Assume one point ($\boldsymbol{x}$) misclassified:



Perceptron Algorithm – Convergence!

› It can be shown that starting from an arbitrary point, $\boldsymbol{\theta}^{(0)}$, the iterative update formula converges after a finite number of steps.

Perceptron Algorithm – Sample/Pattern Mode

› Considers one sample per iteration in a cyclic fashion, until the algorithm converges:

$$\boldsymbol{\theta}^{(i)} = \begin{cases} \boldsymbol{\theta}^{(i-1)} + \mu_i y_{(i)} \boldsymbol{x}_{(i)}, & \text{if } \boldsymbol{x}_{(i)} \text{ is misclassified by } \boldsymbol{\theta}^{(i-1)}, \\ \boldsymbol{\theta}^{(i-1)}, & \text{otherwise.} \end{cases}$$

› Starting from initial condition, $\boldsymbol{\theta}^{(0)}$

› Test each one of the samples, $n = 1..N$

› Every time a sample is misclassified, action is taken for a correction,

› Once all samples have been considered, we say that one ***epoch*** has been completed.

› If no convergence has been attained, all samples are reconsidered in a second epoch and so on.

Perceptron Algorithm – Sample/Pattern Mode

- Initialization
  - $\boldsymbol{\theta}^{(0)} = \mathbf{0}$.
  - Select μ; usually it is set equal to one.
  - $i = 0$.
- **Repeat**; Each iteration corresponds to an epoch.
  - counter = 0; Counts the number of updates per epoch.
  - **For** $n = 1, 2, \ldots, N$, **Do**; For each epoch, all samples are presented.
    - **If** $(y_n \boldsymbol{x}_n^T \boldsymbol{\theta}^{(i-1)} \leq 0)$ **Then**
      - $i = i + 1$
      - $\boldsymbol{\theta}^{(i)} = \boldsymbol{\theta}^{(i-1)} + \mu y_n \boldsymbol{x}_n$
      - counter=counter+1
  - **End For**
- **Until** counter=0

Sum of Error Squares Estimation

› Consider least square criterion:

$$J(\boldsymbol{\theta}) = \sum_{n=1}^{N} (\boldsymbol{\theta}^T \boldsymbol{x}_n - y_n)^2 = \sum_{n=1}^{N} (\boldsymbol{x}_n^T \boldsymbol{\theta} - y_n)^2 = \sum_{n=1}^{N} e_n^2$$

› Solve via batch mode:

$$\boldsymbol{\theta}^{(i)} = \boldsymbol{\theta}^{(i-1)} - \mu_{(i)} \sum_{n=1}^{N} \boldsymbol{x}_n \left(\boldsymbol{x}_n^T \boldsymbol{\theta}^{(i-1)} - y_n \right)$$

› Solve via sample mode:

$$\boldsymbol{\theta}^{(i)} = \boldsymbol{\theta}^{(i-1)} - \mu_{(i)} \boldsymbol{x}_i \left(\boldsymbol{x}_i^T \boldsymbol{\theta}^{(i-1)} - y_i \right) = \boldsymbol{\theta}^{(i-1)} - \mu_{(i)} \boldsymbol{x}_i e_i$$

› LMS Widrow-Hopf algorithm

› There is a closed form solution! How?

Sum of Error Squares Estimation

› Under mild condition the iterative scheme converge to stochastic approximation on LMS:

$$\sum_{i=1}^{\infty} \mu_i \to \infty \;\; and \;\; \sum_{i=1}^{\infty} \mu_i^2 < \infty$$

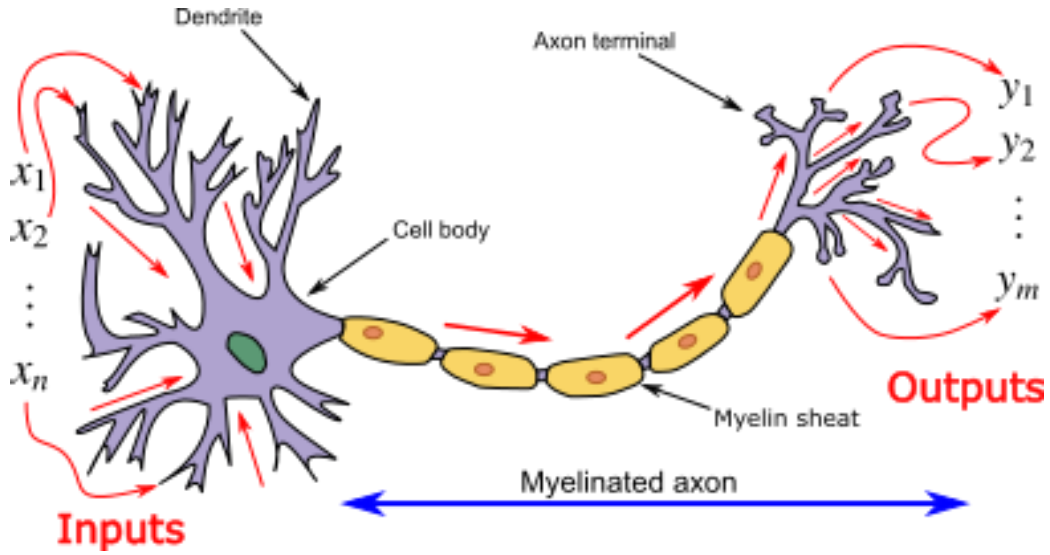
› Example: $\mu_i = \frac{1}{i}$

› Stochastic approximation of LMS

$$\min_{\boldsymbol{\theta}} E\{(\boldsymbol{x}^T \boldsymbol{\theta} - y)^2\}$$

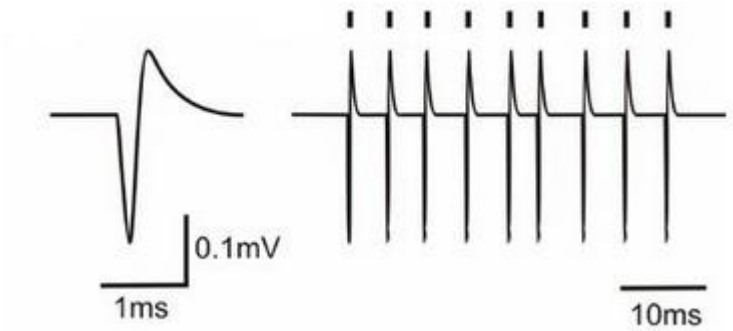
Linear Classifier

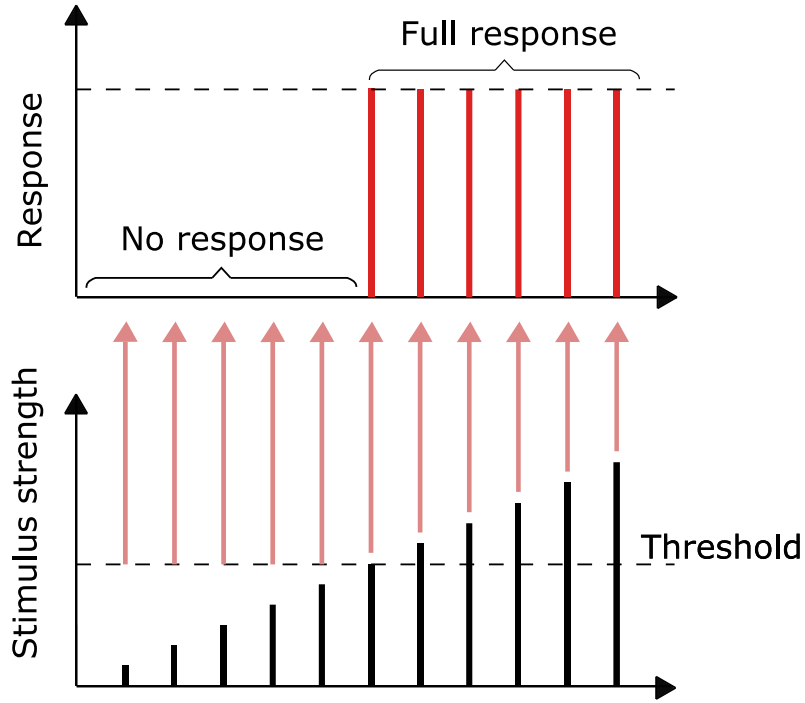
- Perceptron (Single Layer Perceptron , SLP):
  - Neurons (Biologically Inspiration)
    - Inputs
    - Weights (Synapses)
    - Connection
    - Functionality:
      - All-or-None Law
      - Threshold
      - Frequency Modulated Response



Perceptron (Single Layer Perceptron: SLP)

- All-or-None Law
  - Threshold
  - Frequency Modulated Response

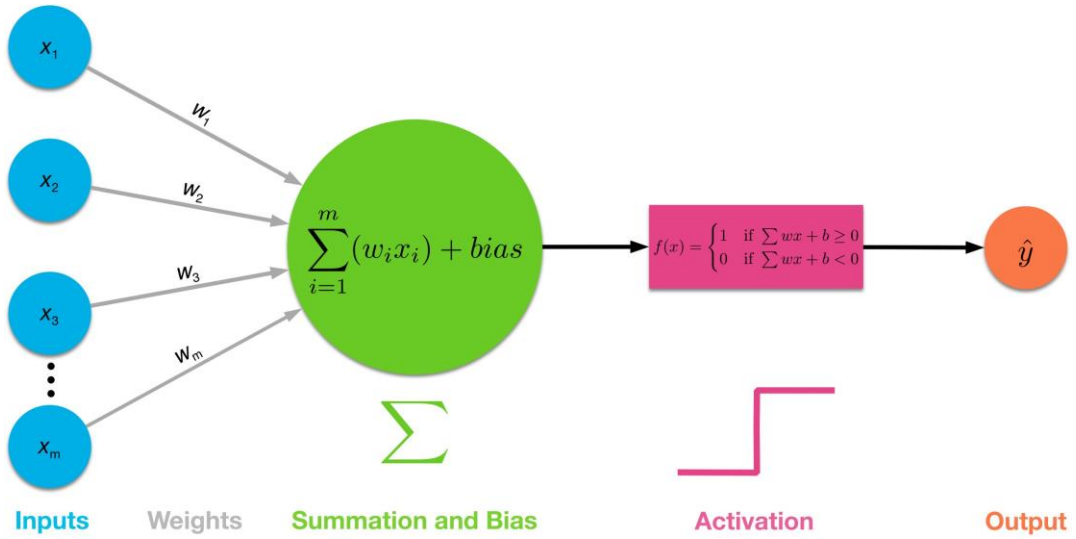




Perceptron (Single Layer Perceptron: SLP)

› Rosenblatt Model or Adaptive Linear Element (ADALINE):
  – Arbitrary (real) inputs
  – Arbitrary (real) weights
  – Arbitrary (real) bias (threshold)
  – Step or Sign activation function!

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0, \\ 0 & \text{otherwise} \end{cases}$$



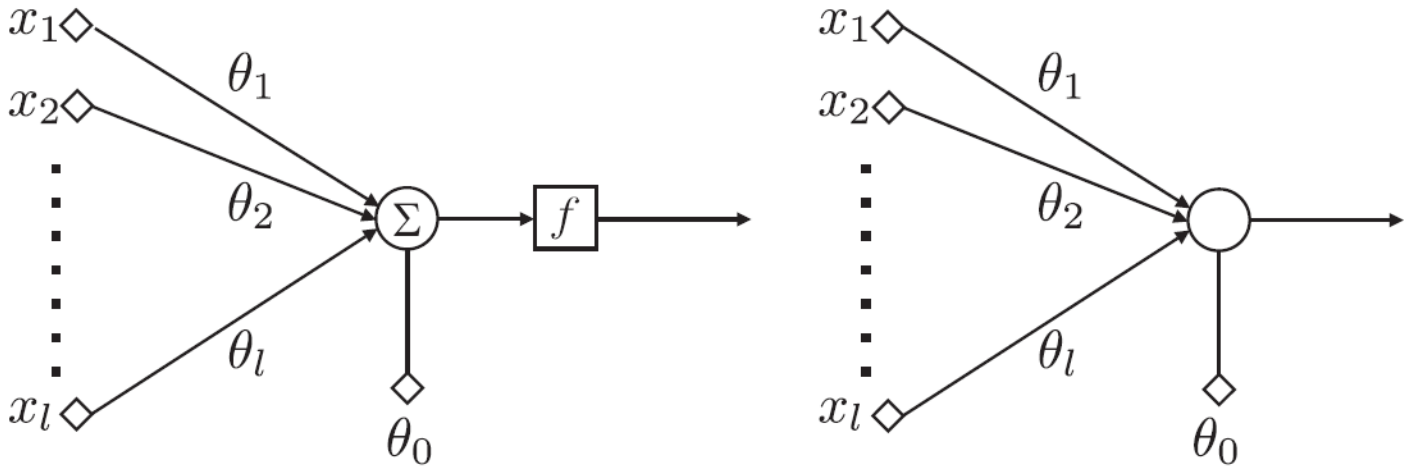
Perceptron with nonlinear activation

› Assume we have a differentiable activation function!

$$\boldsymbol{\theta}^{(i)} = \boldsymbol{\theta}^{(i-1)} - \mu_{(i)} \boldsymbol{x}_i e_i f'(e_i)$$

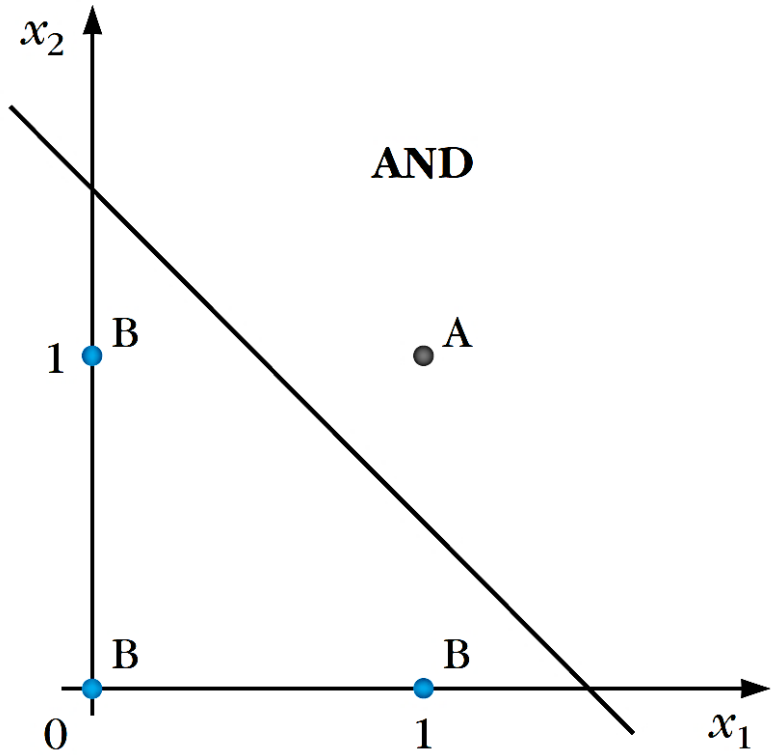
› Neuron activity: $\boldsymbol{\theta}^T \boldsymbol{x}$

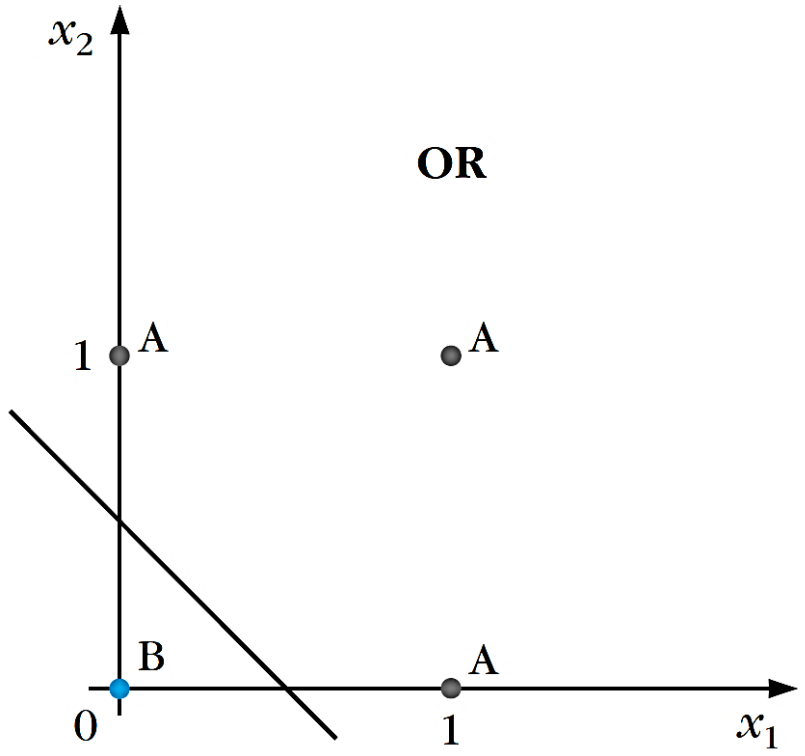
› Neuron output: $f(\boldsymbol{\theta}^T \boldsymbol{x})$



SLP Capability!

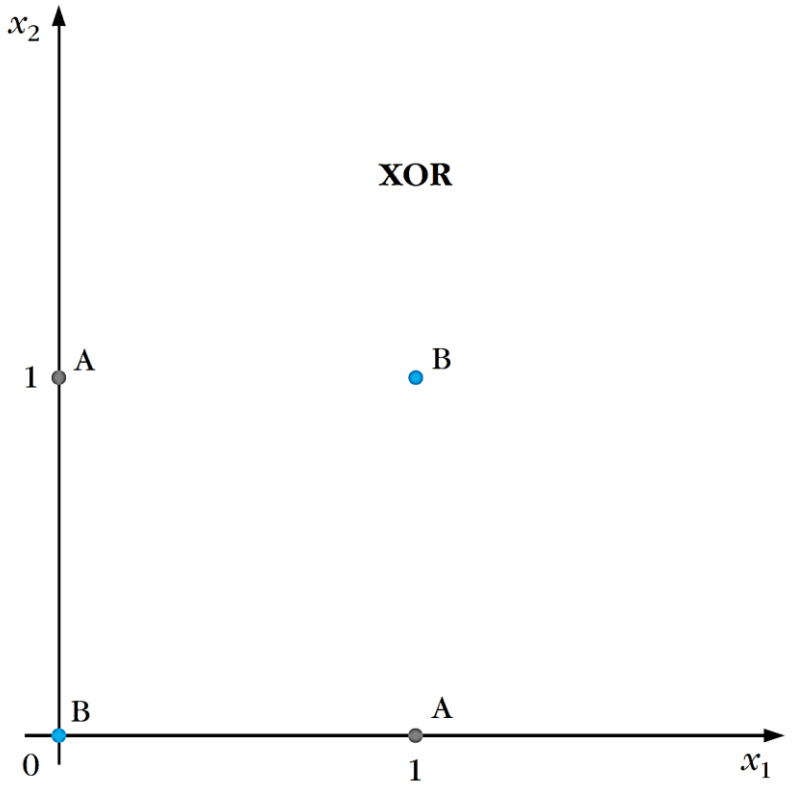
› Let's solve AND and OR problem: it is possible!





SLP Capability!

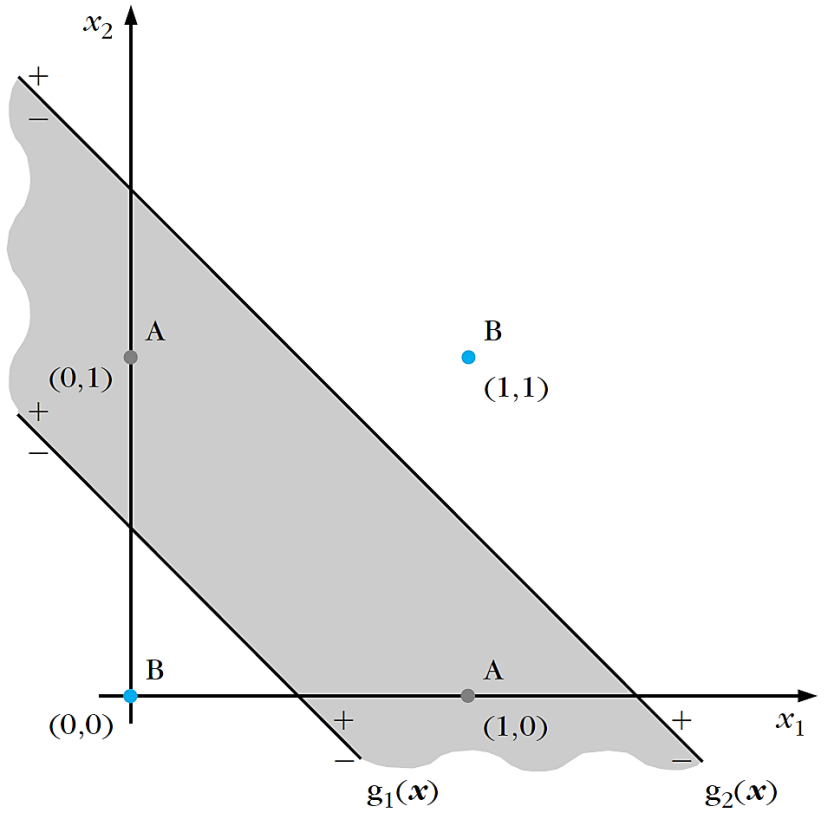
› Now Let's solve XOR problem: it is impossible!



Two-Layer Perceptron

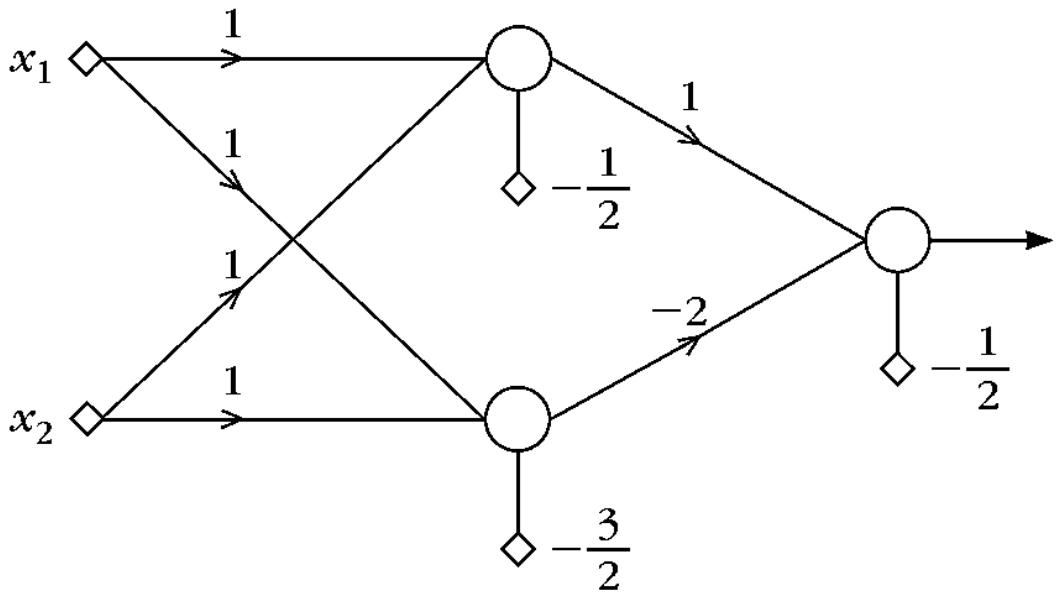
› Now try two-layer perceptron!

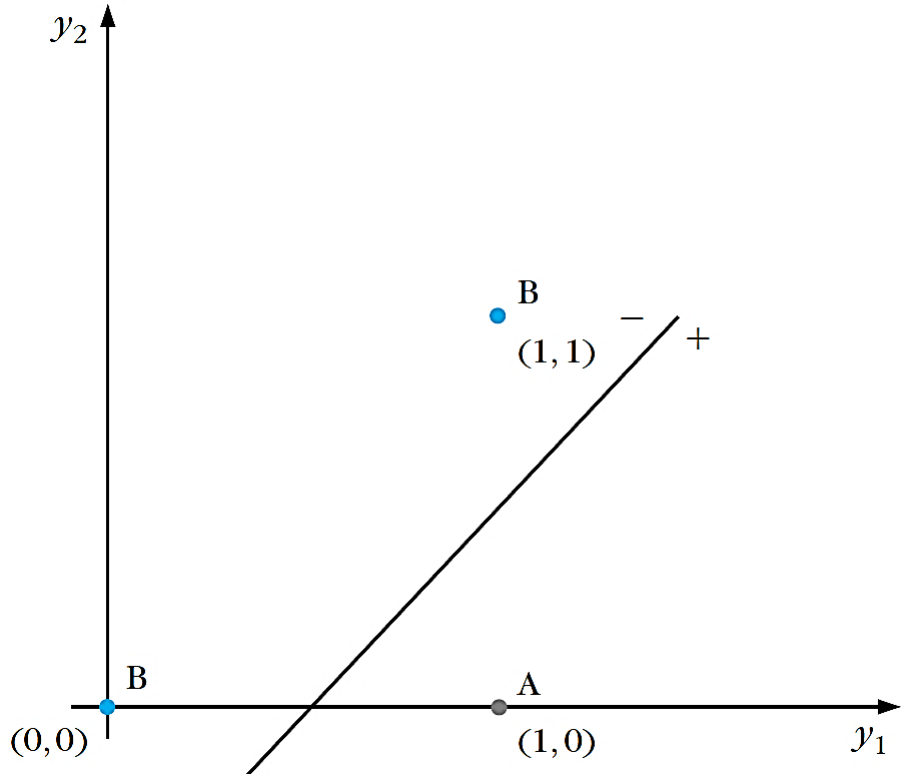
1st Phase				2nd Phase
x_1	x_2	y_1	y_2	
0	0	0 (−)	0 (−)	B (0)
0	1	1 (+)	0 (−)	A (1)
1	0	1 (+)	0 (−)	A (1)
1	1	1 (+)	1 (+)	B (0)



Two-Layer Perceptron

› Now try two-layer perceptron!

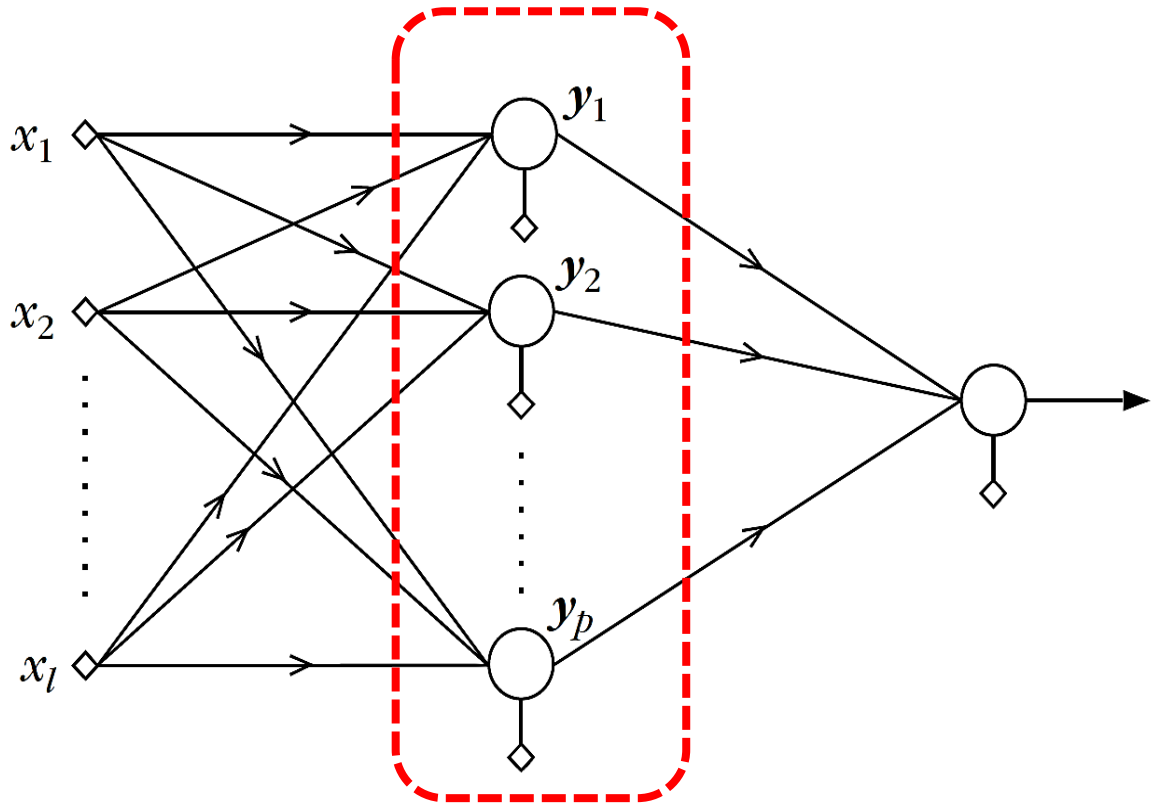




Two-Layer Perceptron Capability

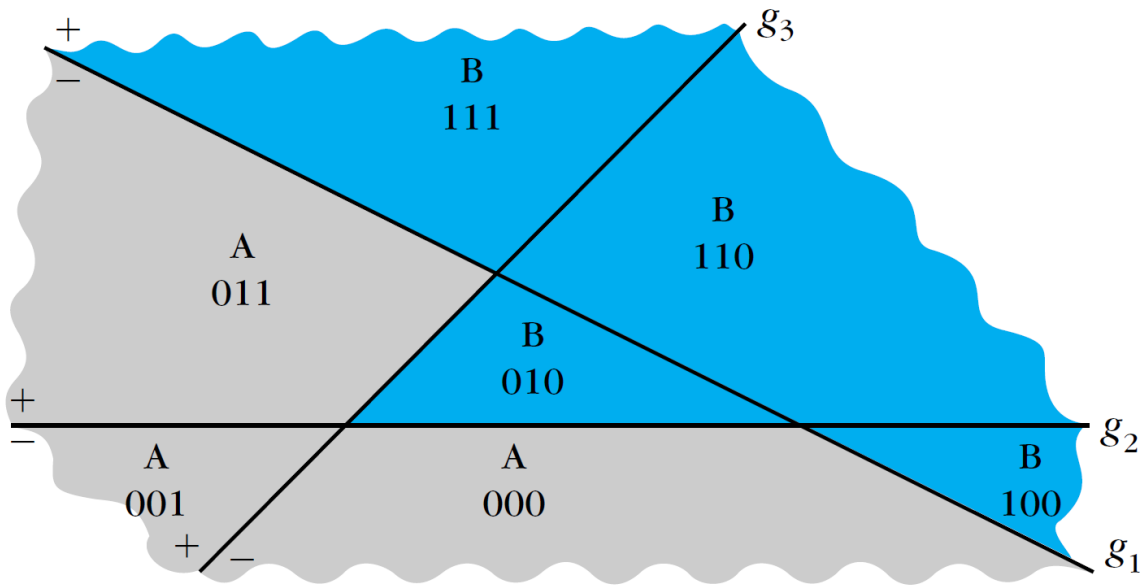
› Two-Layer perceptron:

- Input layer (1): 1
- Output layer (1): 1
- Hidden layer(≥ 0): 1



Two-Layer Perceptron Capability

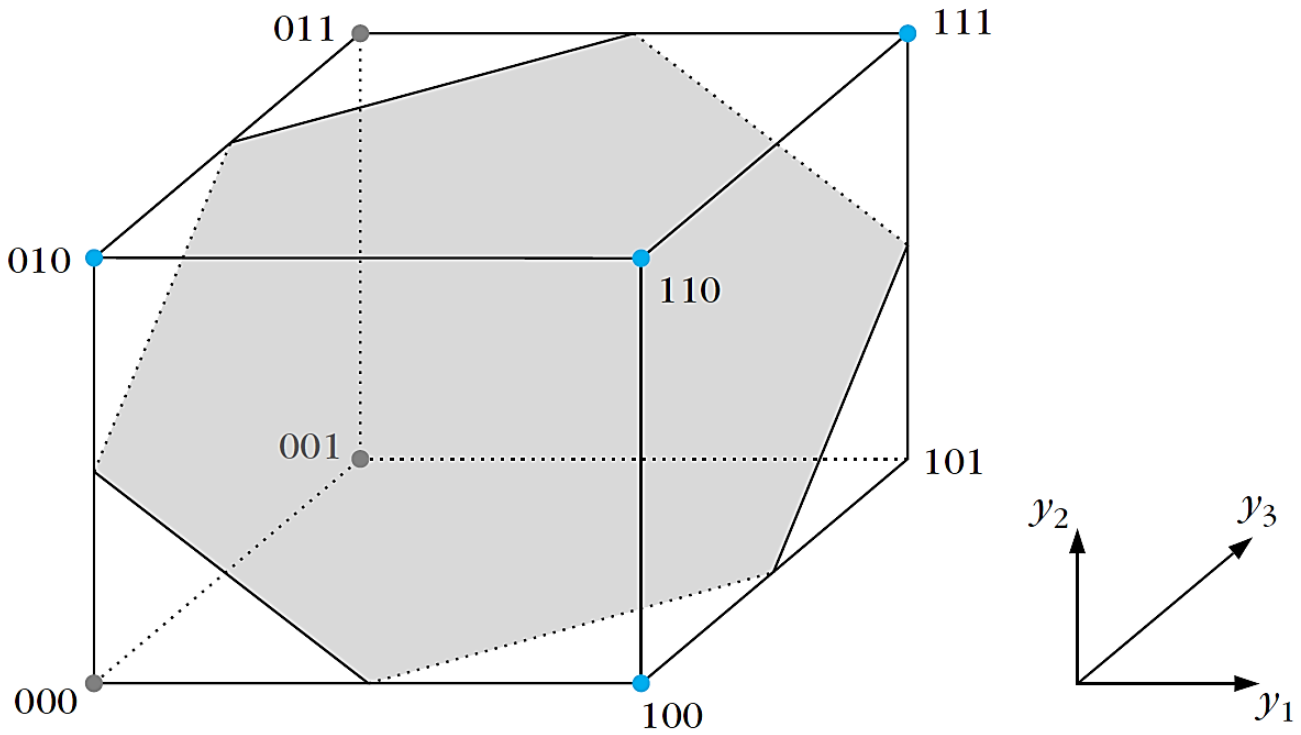
› Two-Layer perceptron (p=3)



› A two-layer perceptron can separate classes each consisting of unions of polyhedral regions but not any union of such regions. (check 000 ∪ 111 ∪ 110 as A and the rest as B)

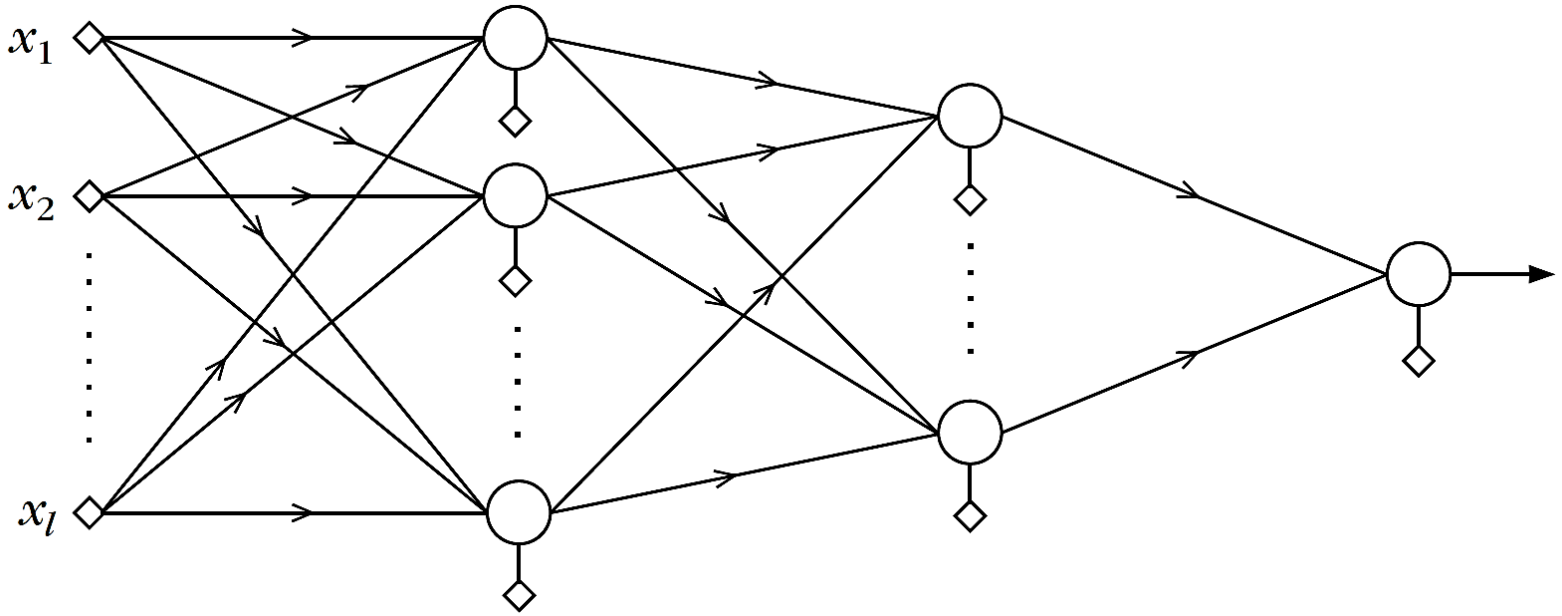
Two-Layer Perceptron Capability

› Two-Layer perceptron (p=3):



Three-Layer Perceptron

› Let have three-layer perceptron!

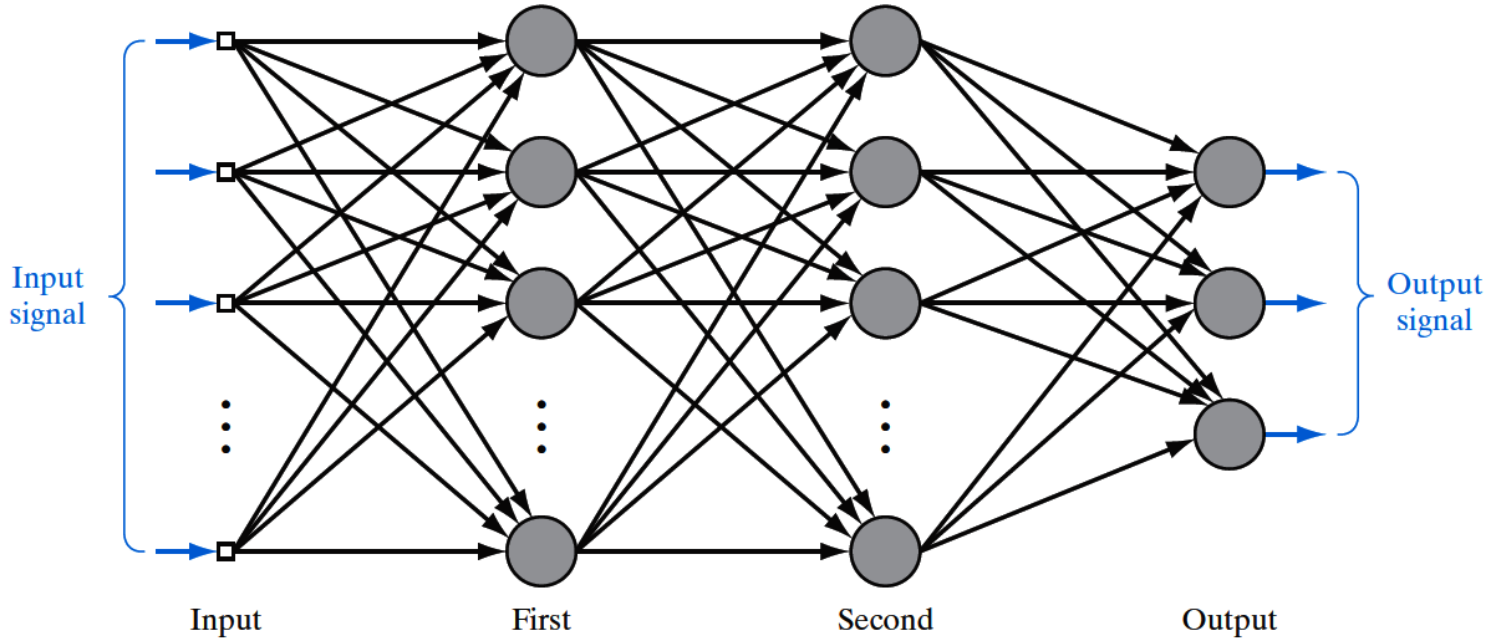


› Can separate classes resulting from any union of polyhedral regions

MADALINE and Multi-Layer Perceptron (MLP)

› MADALINE and MLP (*Shallow* Neural Network):

› Architectural graph of a MLP with *two hidden* layers. (*Dense* or *Fully Connected*)



MLP Transfer Function

› MLP Transfer function is: Highly Nonlinear Mapping from "*Input*" to "*Output*"

$$Outputs = \mathcal{F}(Inputs; \boldsymbol{W})$$

› where $\boldsymbol{W}$ is set of free parameters.

› Hyperparameters (# of layers, # of neurons in each layer)

› MLP is general solution for:

- Classification
- Regression (function approximation)

Training

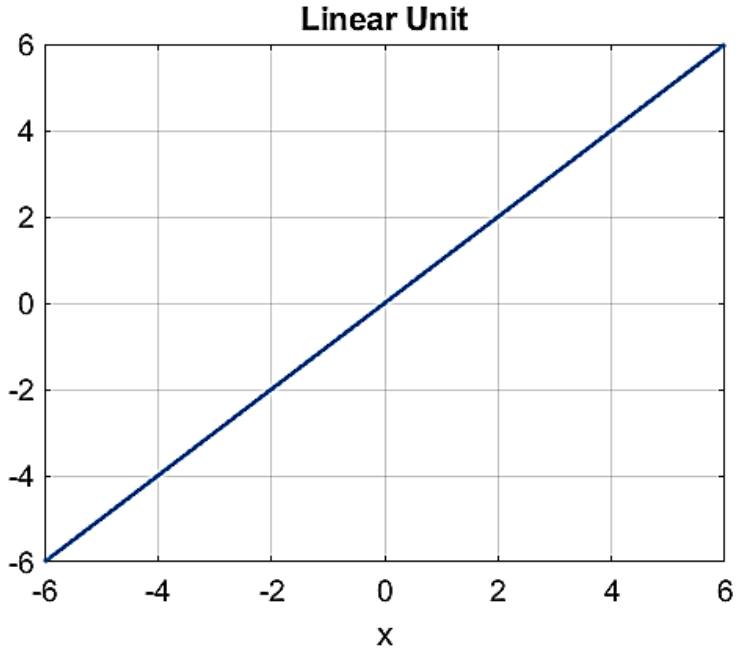
- There are old fashion constructive algorithms to build **M**ulti **L**ayer **P**erceptron (MLP) for classification tasks → Huge architecture
- Error minimization techniques needs differentiability of cost function!
- Solution: smooth step/sign function!

Activation Functions

1) Linear function:

$$f(x) = x$$

$$f'(x) = 1$$



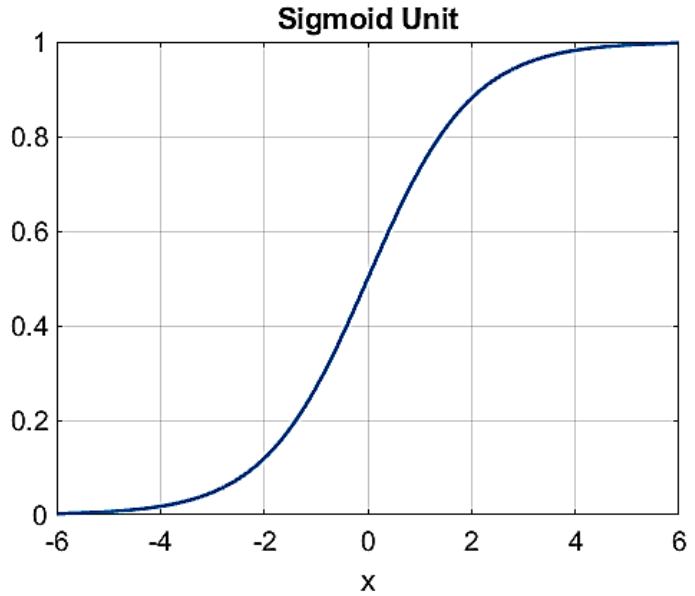
Activation Functions

2) Sigmoid function

$$f(x) = \begin{cases} 1, & x > 0 \\ 0, & x < 0 \end{cases} \Rightarrow f(x) = \frac{1}{1 + e^{-x}}$$

$$f'(x) = f(x)\big(1 - f(x)\big)$$

- Smooth gradient
- Output values bounded
- Vanishing gradient
- Non-Zero at origin
- Computationally expensive



Activation Functions

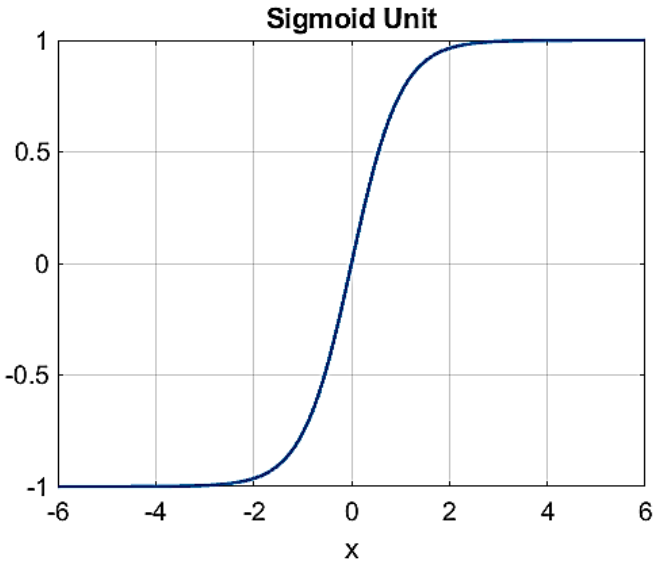
3) tanh function

$$f(x) = \begin{cases} 1, & x > 0 \\ -1, & x < 0 \end{cases} \Rightarrow f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$f'(x) = 1 - f^2(x)$$

› Same as sigmoid

› Zero at origin

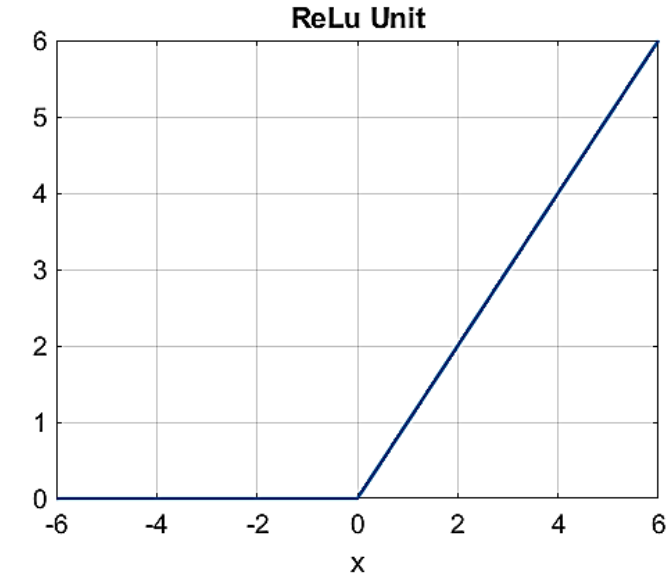


Activation Functions

4) ReLu (Rectified Linear)

$$f(x) = max(x, 0) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

$$f'(x) = u(x)$$



- Non-linear
- Computationally efficient
- Inactive near origin or negative

Activation Functions

5) Leaky ReLu:

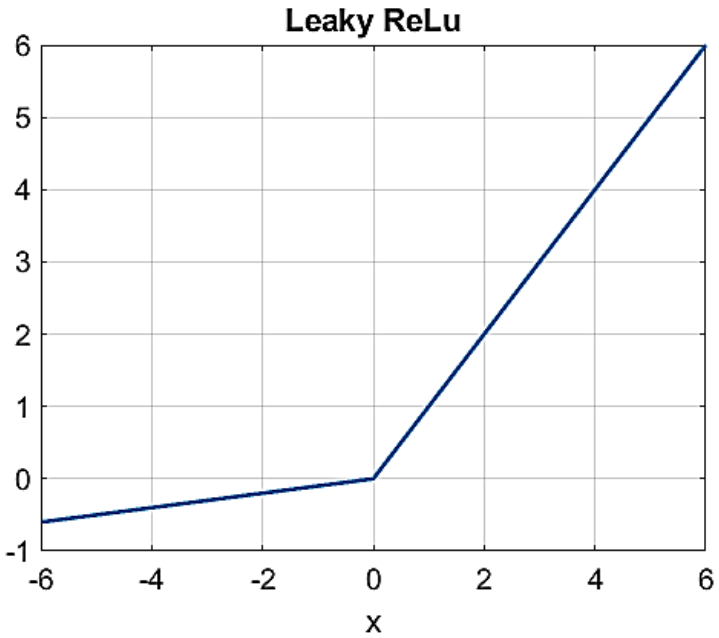
$$f(x) = max(0.01x, x)$$

› Non-linear

› Computationally efficient

› Active near origin or negative

› Problem with negative inputs



Activation Functions

6) Parametric ReLu:

$$f(x) = max(\boldsymbol{w}x, x)$$

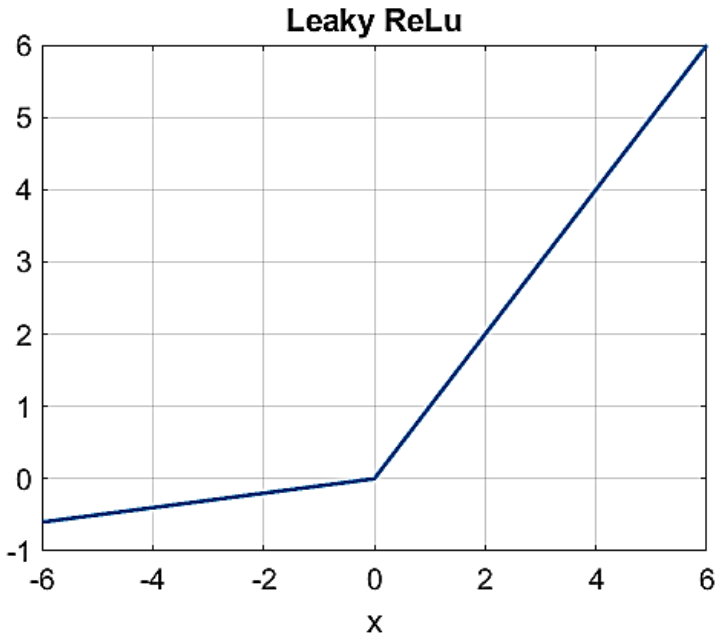
› Non-linear

› Computationally efficient

› Active near origin or negative

› Problem with negative inputs

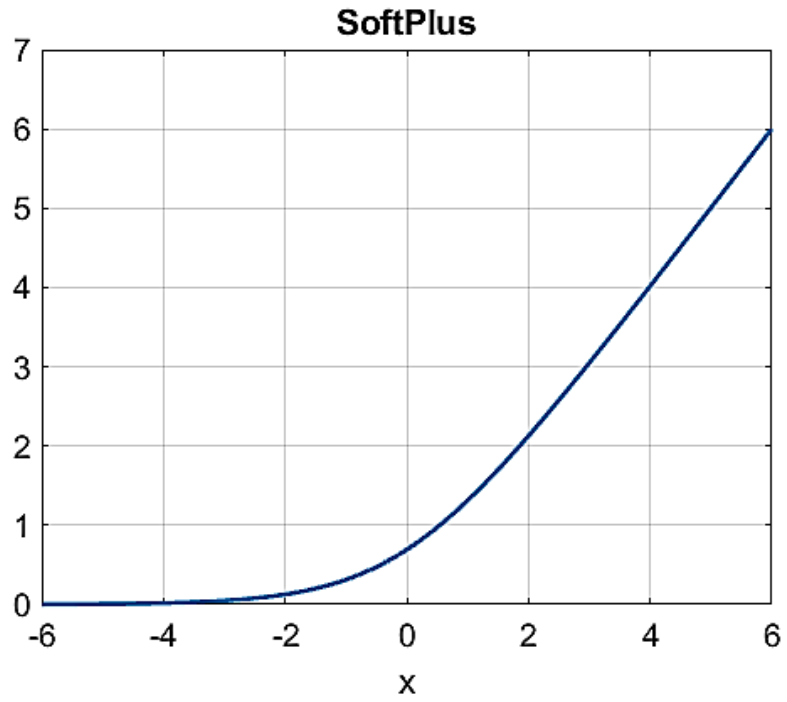
› Learnable for negative values



Activation Functions

7) Softplus (smooth ReLu):

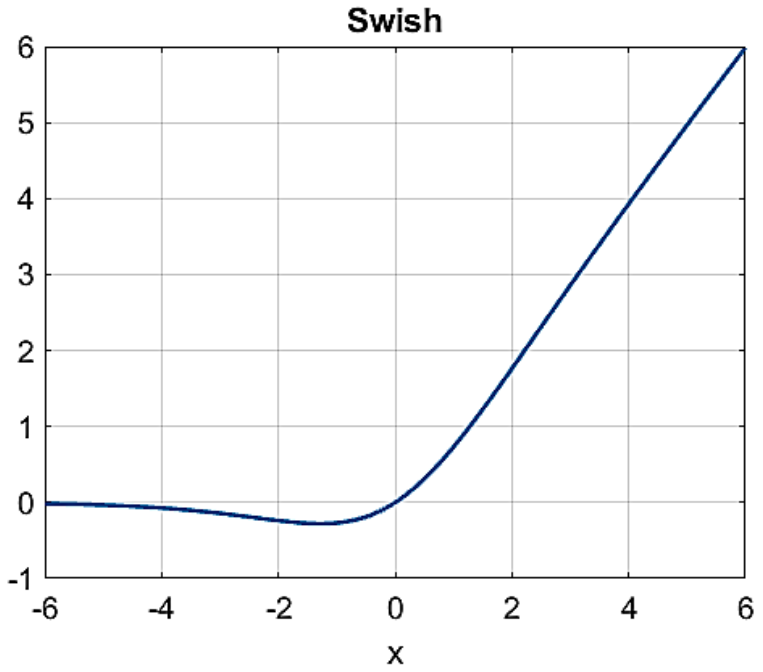
$$f(x) = \log(1 + e^x)$$



Activation Functions

8) Swish:

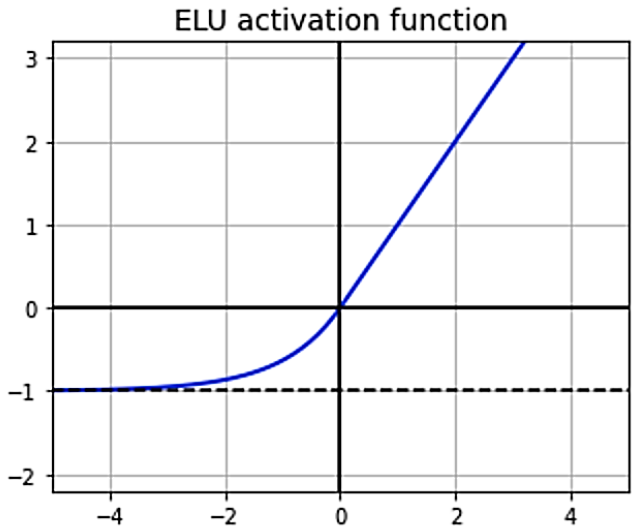
$$f(x) = \frac{x}{1 + e^{-x}}$$



Activation Functions

9) ELU and SELU (Scaled Exponential ReLU):

$$f(x) = \begin{cases} \beta x, & x > 0 \\ \beta\alpha(e^x - 1), & x \leq 0 \end{cases}$$



Activation Functions

10) Softmax (multioutput network):

$$f_i(x) = \frac{e^{x_i}}{\sum_{k=1}^{m} e^{x_k}}$$

› $0 < f_i(x) < 1$

› $\sum_{k=1}^{m} f_k(x) = 1$

“Loss/Cost” function and “output” activation function

- Regression (function approximation) tasks:
- Definition:
  - $N: \#\ of\ samples$
  - $\boldsymbol{y}_i \subset \mathcal{R}^D: Desired\ output\ (target)$
  - $\hat{\boldsymbol{y}}_i \subset \mathcal{R}^D: Actual\ output$
  - $\boldsymbol{e}_i \subset \mathcal{R}^D: (\boldsymbol{y}_i - \hat{\boldsymbol{y}}_i)\ error$

Regression Loss/Cost Function

› Mean Square Error (MSE or L_2 norm):

$$MSE = \frac{1}{N}\sum_{i=1}^{N} \|\boldsymbol{e}_i\|_2^2$$

› Mean Absolute Error (MAE or L_1 norm):

$$MAE = \frac{1}{N}\sum_{i=1}^{N} \|\boldsymbol{e_i}\|_1$$

› Mean Huber Error (MHE):

$$MHE = \frac{1}{N}\sum_{i=1}^{N} h(e_i;\delta)\,, \qquad h(e;\delta) = \begin{cases} e^2, & |e| < \delta \\ \delta(2|e| - \delta), & |e| \geq \delta \end{cases}$$

Regression Loss/Cost Function and Output Function

› Pseudo Mean Huber Error (SMHE):

$$MHE = \frac{1}{N}\sum_{i=1}^{N} \hat{h}(e_i;\delta)\,, \hat{h}(e;\delta) = 2\delta^2\left(\sqrt{1+\left(\frac{e}{\delta}\right)^2}-1\right)$$

› Output activation function:

- Linear
- ReLu

Binary Classification (one binary output) Task

- Definition:
  - N: $\#\ of\ samples$
  - $y_i \in \{0,1\}$: $Desired\ output\ (target)$
  - $\hat{y}_i \in \mathcal{R}$: $Actual\ output$

Binary Classification (one binary output) Task

› Binary Cross Entropy (BCE or log-loss):

$$BCE = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i log\hat{y}_i + (1-y_i)log(1-\hat{y}_i)\right), \quad y_i \in \{0,1\}$$

› Weighted Binary Cross Entropy (WBCE for class imbalance):

$$WBCE = -\frac{1}{N}\sum_{i=1}^{N}\left(\beta y_i log\hat{y}_i + (1-y_i)log(1-\hat{y}_i)\right), \quad y_i \in \{0,1\}$$

- $\beta > 1$: class 1 is weighted higher, meaning the network is less likely to ignore it (lesser FN).
- $\beta > 1$ class 0 is weighted higher, meaning there will be lesser FP.

› Balanced Cross Entropy (BCE

$$BCE = -\frac{1}{N}\sum_{i=1}^{N}\left(\beta y_i log\hat{y}_i + (1-\beta)(1-y_i)log(1-\hat{y}_i)\right), \beta = 1 - \frac{\sum_{i=1}^{N} y_i}{N} \quad y_i \in \{0,1\}$$

Binary Classification (one binary output) Task

› Kullback–Leibler (KL):

$$KL = \frac{1}{N}\sum_{i=1}^{N} y_i log\frac{y_i}{\hat{y}_i}, \quad y_i \in \{0,1\}$$

› Mean Hing Loss (from SVM literature):

$$MHL = \frac{1}{N}\sum_{i=1}^{N} max(0,1 - y_i\hat{y}_i) \quad y_i \in \{-1,1\}$$

› Smoothed Hing Loss (from SVM literature):

$$MSHL = \frac{1}{N}\sum_{i=1}^{N} SHL_i, \quad SHL_i = \begin{cases} 0.5 - y_i\hat{y}_i, & y_i\hat{y}_i < 0 \\ 0.5(1 - y_i\hat{y}_i)^2, & 0 < y_i\hat{y}_i < 1 \\ 0, & y_i\hat{y}_i > 1 \end{cases}$$

Binary Classification (one binary output) Task

› Output activation function:

- sigmoid, $\sigma(z) \in [0,1]$
- tanh, $tanh(z) \in [-1,1]$

Multi-Class Classification(Single Label or Categorical) Tasks

- Definition:
  - N: $\#\ of\ samples$
  - $\boldsymbol{y}_i \in \{0,1\}^M$: $Desired\ output\ (target)$
  - $\boldsymbol{y}_i$ is *one-hot* vector, example (for M=5):

$$\boldsymbol{y}_i = (0 \quad 0 \quad 1 \quad 0 \quad 0)^T$$

  - $\hat{\boldsymbol{y}}_i \in \mathcal{R}^M$: $Actual\ outputs$

Multi-Class Classification(Single Label or Categorical) Tasks

› Cross Entropy (CE):

$$CE = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{M} y_{i,k} log\hat{y}_{i,k}\,, \quad y_{i,k} \in \{0,1\}$$

› Binary Cross Entropy (BCE):

$$BCE = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{M} \left(y_{i,k} log\hat{y}_{i,k} + \left(1 - y_{i,k}\right) log\left(1 - \hat{y}_{i,k}\right)\right), \quad y_{i,k} \in \{0,1\}$$

Multi-Class Classification(Single Label or Categorical) Tasks

› Kullback–Leibler (KL):

$$KL = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{M} y_{i,k} log \frac{y_{i,k}}{\hat{y}_{i,k}}, \quad y_{i,k} \in \{0,1\}$$

› where $y_{i,k}$ is k^{th} element of target vector, and $\hat{y}_{i,k}$ is k^{th} element of output vector.

› Others: Focal Loss, Dice Loss, Tversky Loss

Multi-Class Classification(Single Label or Categorical) Tasks

› Most used output activation function is *softmax*:

$$\hat{y}_{i,k} = softmax(z_{i,k}) = \frac{e^{z_{i,k}}}{\sum_{l=1}^{M} e^{z_{i,l}}}, \qquad 1 \leq k \leq M$$

› where $z_{i,k}$ is output of previous layer

› Taylor *softmax:*

$$\hat{y}_{i,k} = Tsm(z_{i,k}) = \frac{1 + z_{i,k} + 0.5z_{i,k}^2}{\sum_{l=1}^{M} 1 + z_{i,l} + 0.5z_{i,l}^2}$$

Multi-Class Classification(Multiple Label) tasks

- Definition:
  - N: $\#\ of\ samples$
  - $\boldsymbol{y}_i \in \{0,1\}^M$: $Desired\ output\ (target)$
  - $\boldsymbol{y}_i$ is *multi-hot* vector, example (for M=5):

$$\boldsymbol{y}_i = (0 \quad 1 \quad 1 \quad 0 \quad 1)^T$$

  - $\hat{\boldsymbol{y}}_i \in \mathcal{R}^M$: $Actual\ output$

Multi-Class Classification(Multiple Label) tasks:

- Binary Cross Entropy (BCE):

$$BCE = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{M}\left(y_{i,k}log\hat{y}_{i,k} + \left(1-y_{i,k}\right)log\left(1-\hat{y}_{i,k}\right)\right), \quad y_{i,k} \in \{0,1\}$$

- Output activation function:

- *sigmoid*, $\sigma(z) \in [0,1]$

Error Back Propagation Algorithm - EBP

- Definitions:
- $\{\boldsymbol{x_n}, \boldsymbol{y}_n\}_{n=1}^{N}$: set of training samples
- L layers MLP network (L-1 hidden layer and one output layer)
- $\{k_r\}_{r=1}^{L}$: # of neurons in r-th layer
- $\boldsymbol{y}_n = \left(y_{n_1}, y_{n_2}, \dots, y_{n_{k_L}}\right)^T \in \mathbb{R}^{k_L}, n = 1,2, \dots, N$
- $k_0 = l$: input dimension, $\boldsymbol{x}_n \in \mathbb{R}^l$ or $\boldsymbol{x}_n \in \mathbb{R}^{k_0}$
- $\boldsymbol{\theta}_j^r$: weights (vector) associated with the j-th neuron in the r-th layer ($j = 1,2, \dots k_r, r = 1,2, \dots L$)
- $\boldsymbol{\theta}_j^r = \left(\theta_{j_0}^r, \theta_{j_1}^r, \dots, \theta_{j_{k_{r-1}}}^r\right)^T$, The synaptic weights link the respective neuron to all neurons in layer k_{r-1}, *zero index considered for bias*

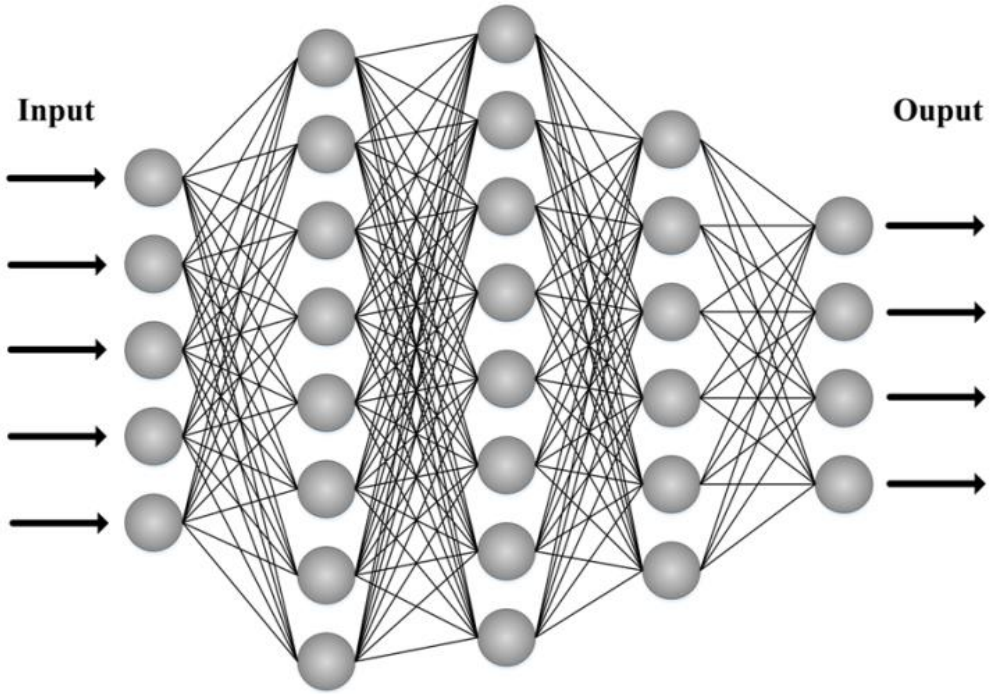
Error Back Propagation Algorithm - EBP

› Illustration:

› $L = 4$

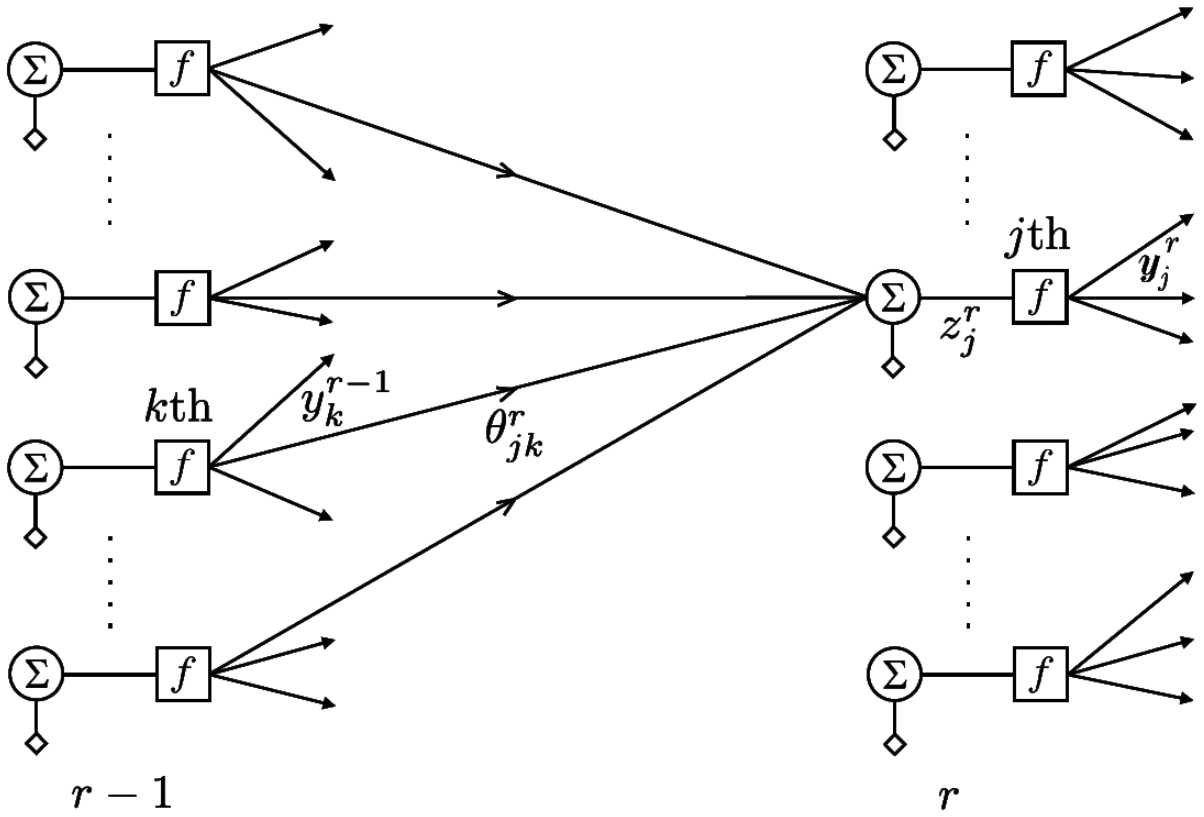
› $k_0 = l = 5$

› $k_1 = 8, k_2 = 8, k_3 = 6, k_4 = 4$



Error Back Propagation Algorithm - EBP

› Illustration:



EBP – Loss Funtion

› Sample Loss:

$$J_n(\boldsymbol{\theta}) = \frac{1}{2}\sum_{k=1}^{k_L}\left(\hat{y}_{nk} - y_{nk}\right)^2$$

› Batch Loss:

$$J(\boldsymbol{\theta}) = \sum_{n=1}^{N} J_n(\boldsymbol{\theta})$$

Error Back Propagation Algorithm - EBP

› The basic update equation using gradient descent algorithm:

$$\boldsymbol{\theta}_j^r(new) = \boldsymbol{\theta}_j^r(old) + \Delta\boldsymbol{\theta}_j^r \qquad \Delta\boldsymbol{\theta}_j^r = -\mu \frac{\partial J}{\partial \boldsymbol{\theta}_j^r}\bigg|_{\boldsymbol{\theta}_j^r(old)}$$

› Computation of the gradients:

› z_{nj}^r: output of the *linear combiner* of the *j-th* neuron in the *r-th* layer for sample n, when the pattern $\boldsymbol{x}_n$ is applied at the input MLP.

$$z_{nj}^r = \left(\boldsymbol{\theta}_j^r\right)^T \boldsymbol{y}_n^{r-1}$$

› $\boldsymbol{y}_n^{r-1}$: output vector from previous layer ($\boldsymbol{y}_n^0 = \boldsymbol{x}_n, \boldsymbol{y}_n^L = \hat{\boldsymbol{y}}_n$)

Error Back Propagation Algorithm - EBP

› Using chain rule:

$$\frac{\partial J_n}{\partial \boldsymbol{\theta}_j^r} = \frac{\partial J_n}{\partial z_{nj}^r} \frac{\partial z_{nj}^r}{\partial \boldsymbol{\theta}_j^r} = \frac{\partial J_n}{\partial z_{nj}^r} \mathbf{y}_n^{r-1}$$

› Remember that z_{nj}^r is output of the of the *j-th* neuron in the *r-th* layer, before activation function.

› Let define:

$$\delta_{nj}^r := \frac{\partial J_n}{\partial z_{nj}^r}$$

Error Back Propagation Algorithm - EBP

› Thus:

$$\Delta\boldsymbol{\theta}_j^r = -\mu \sum_{n=1}^{N} \delta_{nj}^r \mathbf{y}_n^{r-1}, \quad r = 1, 2, \ldots, L.$$

› How to compute δ_{nj}^r (the *backbone* of EBP algorithm!)

› Start from last layer (r=L) and proceeds backwards toward r=*1*

› This philosophy justifies the name given to the algorithm (E**B**P).

Error Back Propagation Algorithm - EBP

› $r=L$ (output layer) :

$$\delta_{nj}^{L} = \frac{\partial J_n}{\partial z_{nj}^{L}} \qquad J_n = \frac{1}{2}\sum_{k=1}^{k_L}\left(f(z_{nk}^{L}) - y_{nk}\right)^2$$

› where f is activation function

› Hence:

$$\begin{aligned}\delta_{nj}^{L} &= (\hat{y}_{nj} - y_{nj}) f'(z_{nj}^{L}), \\ &= e_{nj} f'(z_{nj}^{L}), \quad j = 1, 2, \ldots, k_L,\end{aligned}$$

› Easy to compute!

Error Back Propagation Algorithm - EBP

› $r<L$ (hidden/input layer) :

› z_{nj}^{r-1} has effect on all $\{z_{nk}^{r}\}_{k=1}^{k_r}$(next layer), employing chain rule:

$$\delta_{nj}^{r-1} = \frac{\partial J_n}{\partial z_{nj}^{r-1}} = \sum_{k=1}^{k_r} \frac{\partial J_n}{\partial z_{nk}^{r}} \frac{\partial z_{nk}^{r}}{\partial z_{nj}^{r-1}} \Rightarrow \frac{\partial J_n}{\partial z_{nj}^{r-1}} = \sum_{k=1}^{k_r} \delta_{nk}^{r} \frac{\partial z_{nk}^{r}}{\partial z_{nj}^{r-1}}$$

› Now:

$$\frac{\partial z_{nk}^{r}}{\partial z_{nj}^{r-1}} = \frac{\partial \left(\sum_{m=0}^{k_{r-1}} \theta_{km}^{r} y_{nm}^{r-1} \right)}{\partial z_{nj}^{r-1}} \qquad y_{nm}^{r-1} = f(z_{nm}^{r-1})$$

Error Back Propagation Algorithm - EBP

› $r<L$ (hidden/input layer) :

$$\frac{\partial z_{nk}^{r}}{\partial z_{nj}^{r-1}} = \frac{\partial \left(\sum_{m=0}^{k_{r-1}} \theta_{km}^{r} y_{nm}^{r-1} \right)}{\partial z_{nj}^{r-1}} \qquad y_{nm}^{r-1} = f(z_{nm}^{r-1})$$

› Which lead:

$$\frac{\partial z_{nk}^{r}}{\partial z_{nj}^{r-1}} = \theta_{kj}^{r} \, f'(z_{nj}^{r-1})$$

Error Back Propagation Algorithm - EBP

› $r<L$ (hidden/input layer) :

› Using:

$$\delta_{nj}^{r-1} = \frac{\partial J_n}{\partial z_{nj}^{r-1}} = \sum_{k=1}^{k_r} \frac{\partial J_n}{\partial z_{nk}^{r}} \frac{\partial z_{nk}^{r}}{\partial z_{nj}^{r-1}}$$

$$\frac{\partial z_{nk}^{r}}{\partial z_{nj}^{r-1}} = \theta_{kj}^{r} \, f'(z_{nj}^{r-1})$$

$$\frac{\partial J_n}{\partial z_{nj}^{r-1}} = \sum_{k=1}^{k_r} \delta_{nk}^{r} \frac{\partial z_{nk}^{r}}{\partial z_{nj}^{r-1}}$$

› We have:

$$\delta_{nj}^{r-1} = \left(\sum_{k=1}^{k_r} \delta_{nk}^{r} \theta_{kj}^{r} \right) f'(z_{nj}^{r-1}), \quad j = 1, 2, \ldots, k_{r-1}$$

Error Back Propagation Algorithm - EBP

› $r<L$ (hidden/input layer) :

› For uniformity with "r=L" case, we define:

$$e_{nj}^{r-1} := \sum_{k=1}^{k_r} \delta_{nk}^{r} \theta_{k}^{r} \quad \Longrightarrow \quad \delta_{nj}^{r-1} = e_{nj}^{r-1} f'(z_{nj}^{r-1})$$

› Finally:

$$\frac{\partial J_n}{\partial \boldsymbol{\theta}_j^r} = \frac{\partial J_n}{\partial z_{nj}^r} \frac{\partial z_{nj}^r}{\partial \boldsymbol{\theta}_j^r} = \frac{\partial J_n}{\partial z_{nj}^r} \mathbf{y}_n^{r-1} \qquad \delta_{nj}^r := \frac{\partial J_n}{\partial z_{nj}^r}$$

Error Back Propagation Algorithm - EBP

› For classification tasks:

$$\hat{y}_{nk} = \frac{\exp(z_{nk}^{L})}{\sum_{m=1}^{k_L} \exp(z_{nm}^{L})} : \quad \text{softmax activation function}$$

Error Back Propagation Algorithm - EBP

› An important observation, Recall that:

$$\delta_{nj}^{r-1} = \left(\sum_{k=1}^{k_r} \delta_{nk}^{r} \theta_{kj}^{r} \right) f'(z_{nj}^{r-1}), \quad j = 1, 2, \ldots, k_{r-1}$$

› Write this recursion for two successive layer:

$$\delta_{j}^{r-1} = \left(\sum_{k=1}^{k_r} \left(\sum_{i=1}^{k_{r+1}} \delta_{i}^{r+1} \theta_{ik}^{r+1} \right) f'(z_{k}^{r}) \theta_{kj}^{r} \right) f'(z_{j}^{r-1}), \quad j = 1, 2, \ldots, k_{r-1}$$

The derivatives of the nonlinearities as well as the weights are *multiplied* and the number of the involved products grows.

Error Back Propagation Algorithm - EBP

- Heuristics for Making the EBP Algorithm Perform Better:
  - Stochastic versus batch update:
    - Sample/pattern mode with random shuffle order in each epoch
  - Maximizing information content:
    - Use samples that results in the largest training error (or radically different)
    - Sample/pattern mode with random shuffle order in each epoch
    - Duplicate hard-to-learn samples in each epoch
  - Activation function:
    - Use symmetric-odd function (*tanh*) for fast convergence (in comparison with *sigmoid*), Note: $tanh\big(Utanh(V\boldsymbol{x})\big) \approx UV\boldsymbol{x}$

Error Back Propagation Algorithm - EBP

› Heuristics for Making the EBP Algorithm Perform Better:

– Target Value:

› Target values be chosen within the range of the activation function (with proper margin), for example ± 0.7 instead of ± 1 for $tanh$, or $(0.3\ ,0.7)$ instead of $(0\ ,1)$ for $sigmoid$.

› A recommendation by LeCun:

$$\varphi(z) = 1.7159 tanh\left(\frac{2}{3}z\right), \qquad d_i \in \{-1, +1\}$$

– Input Normalization:

› Un-correlate input with zero mean and equal variance (using PCA) over the *entire* training sample.

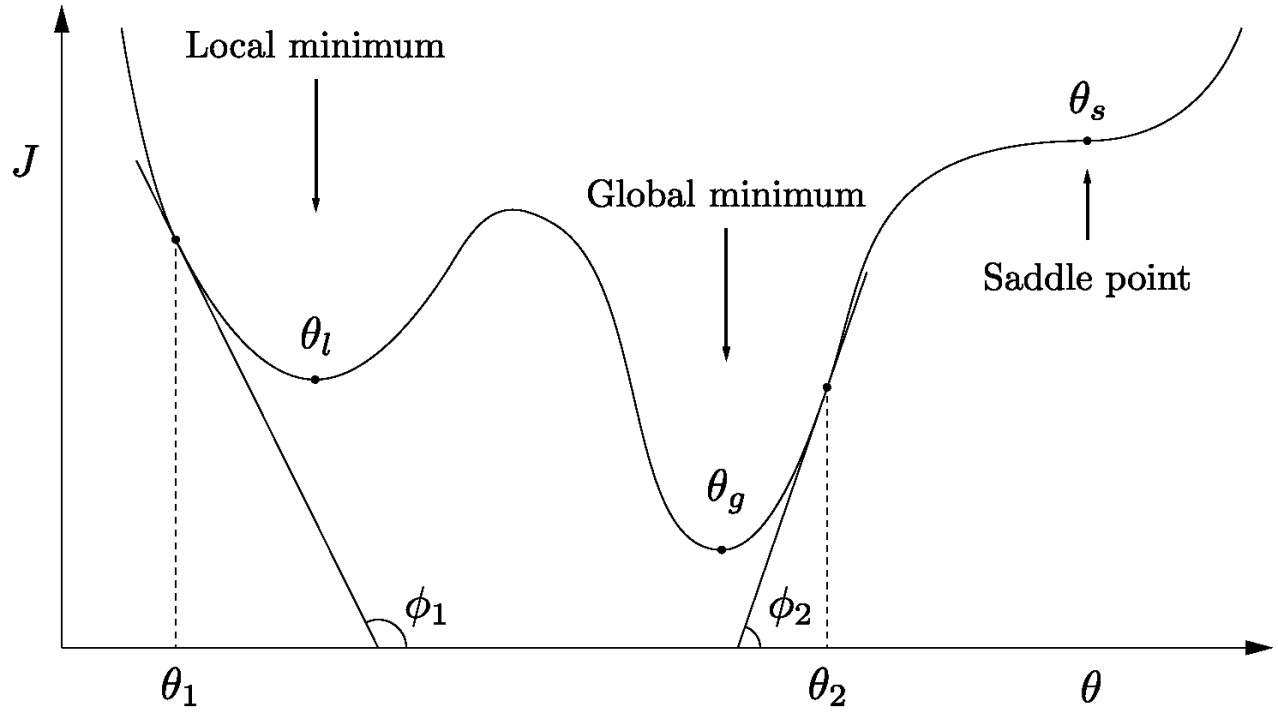
Error Back Propagation Algorithm - EBP

- Heuristics for Making the EBP Algorithm Perform Better:
  - Weight Initialization:
    - General assumption: input to neuron is zero mean and unity variance (uniform or gaussian), $v_j = \sum_{i=1}^{m} w_{ij} y_i, m_{y_i} = 0, var(y_i) = 1$
    - Goal: Neuron output be zero mean and unity variance (same distribution), $m_{v_j} = 0, var(v_j) = 1$
  - Some results (Xavier/He/Glorot initialization):
    - $Uni\left[-\sqrt{\frac{6}{m_{in}+m_{out}}}, \sqrt{\frac{6}{m_{in}+m_{out}}}\right]$ or $N\left(0, \frac{2}{m_{in}+m_{out}}\right)$ for $tanh$
    - $Uni\left[-\sqrt{\frac{12}{m_{in}+m_{out}}}, \sqrt{\frac{12}{m_{in}+m_{out}}}\right]$ or $N\left(0, \frac{4}{m_{in}+m_{out}}\right)$ for $ReLu$

Error Back Propagation Algorithm - EBP

› Heuristics for Making the EBP Algorithm Perform Better:

– Local minima and saddle points challenge:



Error Back Propagation Algorithm - EBP

› Heuristics for Making the EBP Algorithm Perform Better:

- Local minima and saddle points challenge:
- **Re-Initialization:** Re-train using different random initial weights to avoid local minima and saddle point.
- **Noise Injection**:
- a) Add noise to input,
- b) Add noise to the parameters as they are being estimated during training,
- c) Add noise to output target (*label smoothing* for classification)

Error Back Propagation Algorithm - EBP

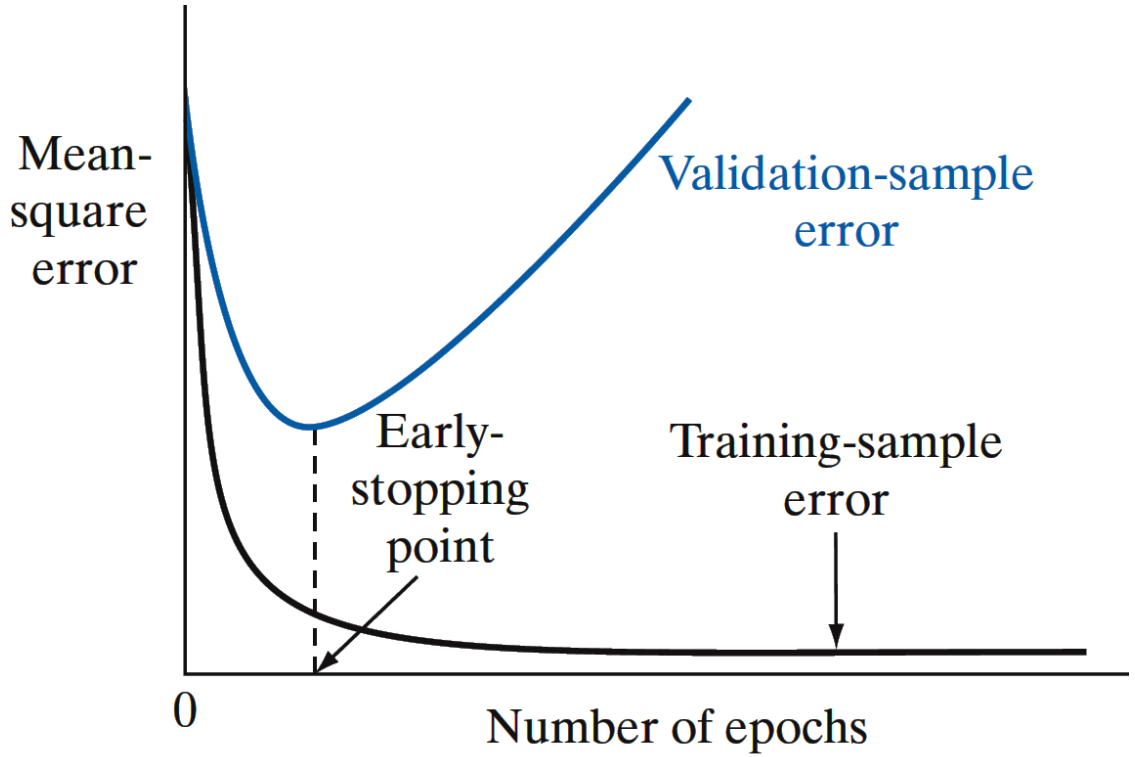
› Heuristics for Making the EBP Algorithm Perform Better:

- Database:
- **Training data:** Used during the learning process and is used to find the parameters of model.
- **Validation data:** Used to tune *hyperparameters* (i.e. the network architecture: # of hidden layer, # of hidden units in each layer), avoid *overfitting*, or *early-stopping* policy.
- **Test data:** Used for final evaluation

Error Back Propagation Algorithm - EBP

› Heuristics for Making the EBP Algorithm Perform Better:

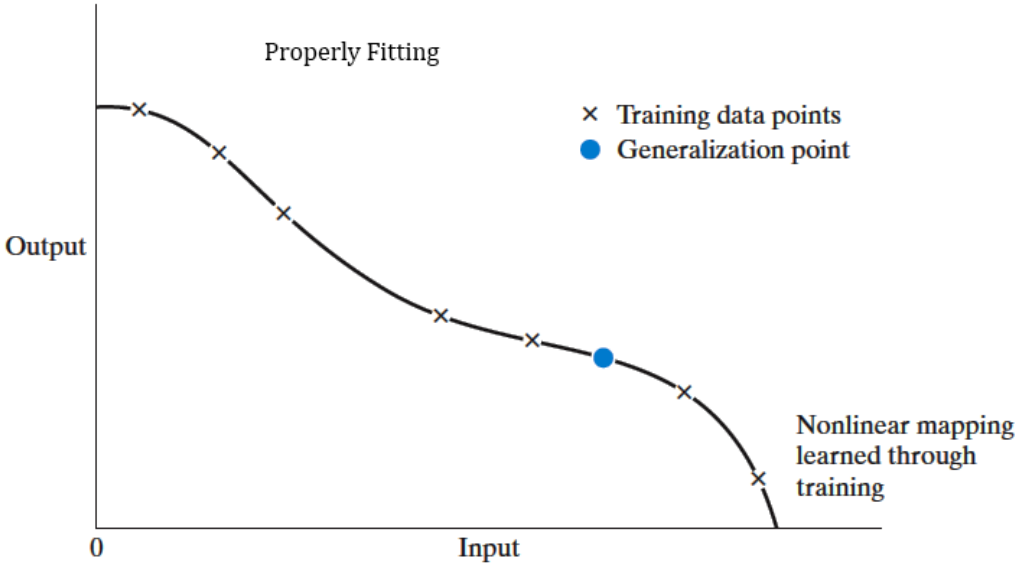
– Early-Stopping:

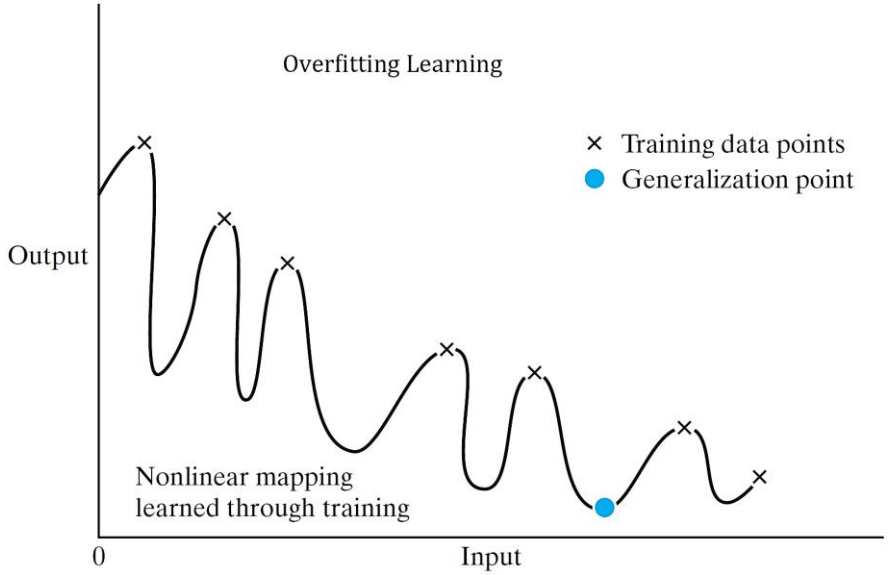


Network Generalization

› Generalization:

– A network is said to generalize well when the input–output mapping computed by the network is *correct* (or nearly so) for *test data never used* in creating or training the network.





Network Generalization

› Universal Approximation Theorem:

Let $\varphi(\cdot)$ be a nonconstant, bounded, and monotone-increasing continuous function. Let I_{m_0} denote the m_0-dimensional unit hypercube $[0, 1]^{m_0}$. The space of continuous functions on I_{m_0} is denoted by $C(I_{m_0})$. Then, given any function $f \ni C(I_{m_0})$ and $\varepsilon > 0$, there exist an integer m_1 and sets of real constants α_i, b_i, and w_{ij}, where $i = 1, \ldots, m_1$ and $j = 1, \ldots, m_0$ such that we may define

$$F(x_1, \ldots, x_{m_0}) = \sum_{i=1}^{m_1} \alpha_i \varphi\left(\sum_{j=1}^{m_0} w_{ij} x_j + b_i \right) \tag{4.88}$$

as an approximate realization of the function $f(\cdot)$; that is,

$$|F(x_1, \ldots, x_{m_0}) - f(x_1, \ldots, x_{m_0})| < \varepsilon$$

for all $x_1, x_2, \ldots, x_{m_0}$ that lie in the input space.

Network Generalization

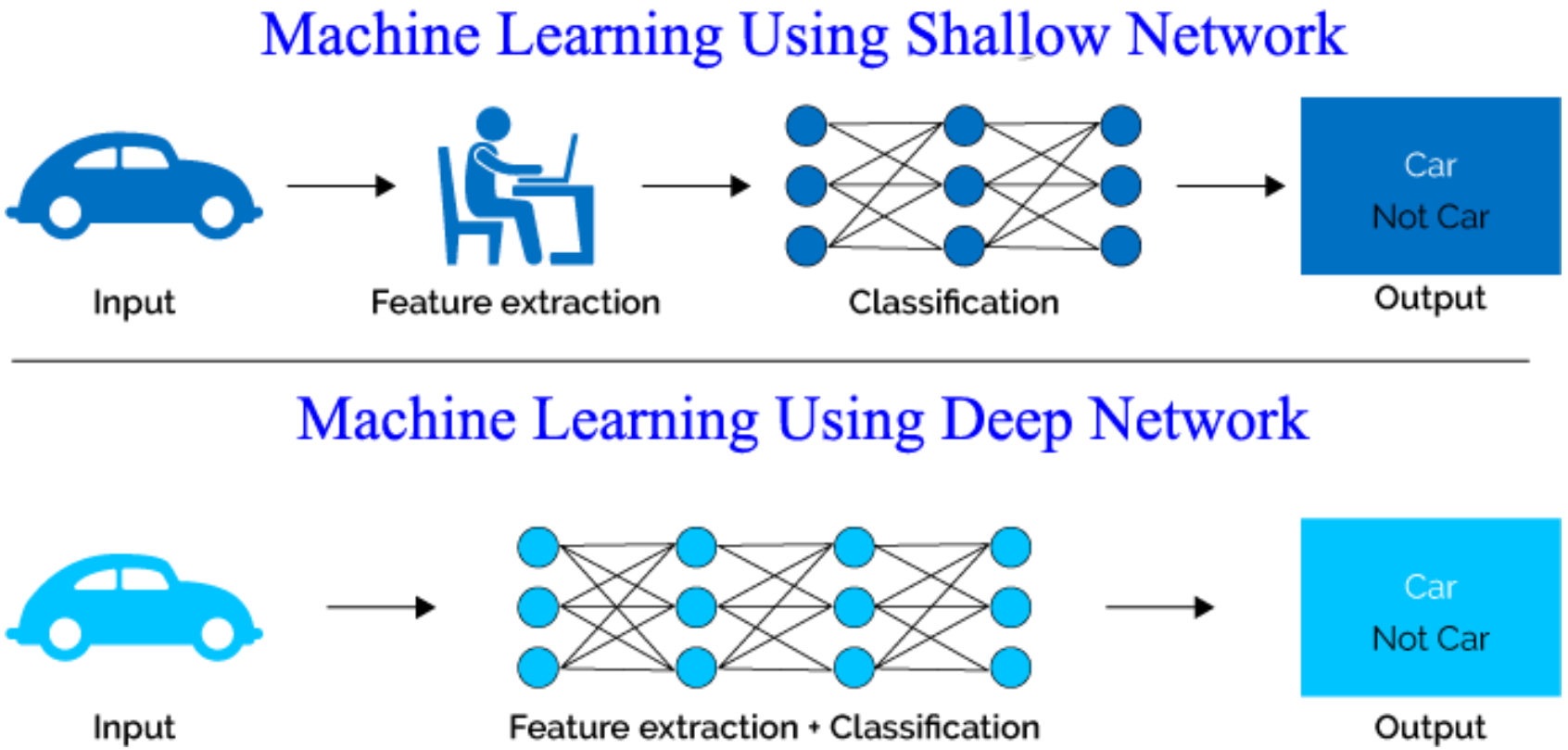
› Universal Approximation Theorem:

  – This function is a MLP with one hidden layer (sigmoid/tanh activation function) and linear output layer. *This is existence theorem!*

› A similar theorem proved for *ReLU* activation function.

“Why Deep Not Shallow” or “Why Deep Not Wide”

› Deep architecture vs Shallow Architecture:



“Why Deep Not Shallow” or “Why Deep Not Wide”

- It is shown that for a special class of *deep* networks and target outputs, one needs a substantially *smaller number of nodes* to achieve a predefined accuracy compared to a *shallow network.*
- From:
  - *H. Mhaskar, T. Poggio, Deep vs shallow networks: an approximation theory perspective, Anal. Appl. 14 (6) (2016) 829–848.*
  - *T. Poggio, H. Mhaskar, L. Rosasco, B. Miranda, Q. Liao, Why and when can deep but not shallow networks avoid the curse of dimensionality: a review, Int. J. Autom. Comput. 14 (5) (2017) 503–519.*

“Why Deep Not Shallow” or “Why Deep Not Wide”

- It is shown that there is a simple function in $\mathbb{R}^l$, which is expressive by a *small three-layer feed-forward* neural network, while it cannot be adequately approximated by *any two-layer network*, unless the number of nodes is exponentially large with respect to the dimension.
- From:
  - *R. Eldan, O. Shamir, The power of depth for feed-forward neural networks, arXiv:1512.03965v4 [cs.LG], 9 May 2016.*

“Why Deep Not Shallow” or “Why Deep Not Wide”

- Formally, these results demonstrate that *depth* - even if increased by *one* - can be exponentially more valuable than width (number of nodes per layer) for standard feed-forward neural networks.
- From:
  - *R. Eldan, O. Shamir, The power of depth for feed-forward neural networks, arXiv:1512.03965v4 [cs.LG], 9 May 2016.*

“Why Deep Not Shallow” or “Why Deep Not Wide”

- It is shown that besides a negligible set, *all functions* that can be realized by a *deep CNN network* of *polynomial* size require *exponential* size in order to be realized, or even approximated, by a *shallow* network.
- From:
  - *N. Cohen, O. Sharir, A. Shashua, On the expressive power of deep learning: a tensor analysis, arXiv:1509.05009v3 [cs.NE], 27 May 2016.*

“Why Deep Not Shallow” or “Why Deep Not Wide”

- It is shown that there exists a family of ReLU networks that *cannot* be approximated by *narrower* networks whose *depth* increase is no more than polynomial.
- The *theoretical* and the *experimental* evidence in the paper points out that *depth* may be more effective than *width* for the expressiveness of ReLU networks.
- From:
  - *Z. Lu, H. Pu, F. Wang, Z. Hu, L. Wang, The expressive power of neural networks: a view from the width, in: Advances in Neural Information Processing Systems, NIPS, 2017.*

On the Optimization of Deep Networks: Some Theoretical Highlights

- The *general belief* 1980s and 1990s was that because the number of the parameters becomes *very large*, the cost function in the parameter space becomes complicated and the probability of getting stuck in a *local minimum* significantly increases.

- This belief has been seriously *challenged* after 2010. At that time, it was discovered that one can *train large networks*, provided that *enough training data* were used. It was around this time that large data sets were built and could be used for training in parallel with the advances in computer technology that offered the necessary *computational power*.

On the Optimization of Deep Networks: Some Theoretical Highlights

- Crucial factors for the comeback of MLP-NN:
  - The availability of large data sets
  - Computer technology (GPU)
- Some important (with secondary contribution)
  - ReLU nonlinearity,
  - Regularization techniques (Dropout and etc.),

On the Optimization of Deep Networks: Some Theoretical Highlights

- A more profound *difficulty*, especially in high-dimensional problems, originates from the proliferation of *saddle points*. The existence of such points can *slow down* the convergence of the training algorithm dramatically
- From:
  - *Y. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, Y. Bengio, Identifying and attacking the saddle point problem in high-dimensional nonconvex optimization, arXiv:1406.2572v1 [cs.LG], 10 June 2014.*

On the Optimization of Deep Networks: Some Theoretical Highlights

- It is claimed that in *large-size* networks, most of the *local minima* yield *low cost* values and result in similar performance on a test set. Moreover, the probability of finding a *poor* (high cost value) *local* minimum decreases fast as the network *size increases*.
- From:
  - *A. Choromanska, M. Henaff, M. Mathieu, G.B. Arous, Y. Le Cun, The loss surfaces of multilayer networks, in: Proceedings of the 18th International Conference on Artificial Intelligence and Statistics, AISTATS, 2015.*

On the Optimization of Deep Networks: Some Theoretical Highlights

- For the case of the *squared error loss* function, it is shown that *gradient descent* finds a *global* minimum in training deep neural networks (CNN/Residual Net, and FC); this is in spite of the fact that the cost function is a nonconvex one with respect to the involved parameters.
- From:
  - *S. Du, J. Lee, H. Li, L.Wang, X. Zhai, Gradient descent finds global minima of deep neural networks, arXiv:1811.03804v1 [cs.LG], 9 November 2018.*

On the Generalization Power of Deep Networks

- Generalization Power of Deep Networks is still an open problem, is a very active area of research.
- There are many *global minima* of the training objective, most of which will *not generalize well*, but the optimization algorithm (e.g., gradient descent) biases the solution *toward* a particular minimum that does *generalize well*.
- From:
  - *D. Soudry, E. Hoffer, M.S. Nason, N. Srebro, The implicit bias of gradient descent on separable data, in: Proceedings International Conference on Learning Representations, ICLR, 2018.*

On the Generalization Power of Deep Networks

- Training very large overparameterized networks, where the number of parameters is larger than the size of the training set, even without regularization, often (not always) the resulting network exhibits good generalization performance.
- Concluded from several works and research, most of them experimentally.

On the Generalization Power of Deep Networks

- Two Good sources to understand the difficulty of generalization analysis:
  - *C. Zhang, S. Bengio, M. Hardt, B. Recht, O. Vinyals, Understanding deep learning requires rethinking generalization, arXiv:1611.03530v2 [cs.LG], 26 February 2017.*
  - *K. Kawaguchi, L. Kaelbling, Y. Bengio, Generalization in deep networks, arXiv:1710.05468v3 [stat.ML], 22 February 2018.*

Final words:

› Theoretical aspects of Deep Learning (optimization, generalization, design, …) is an open problem and ongoing, but it works!