Linear Classifier, Perceptron, and MLP

DEEP LEARNING 2023

E. FATEMIZADEH

Contents

- Single and Layer Perceptron (SLP)
- Multilayer Perceptron (MLP)
- Activation Function
- Cost-Loss Functions
- MLP Training and Error Back Propagation
- Generalization Theorem
- Deep Architecture

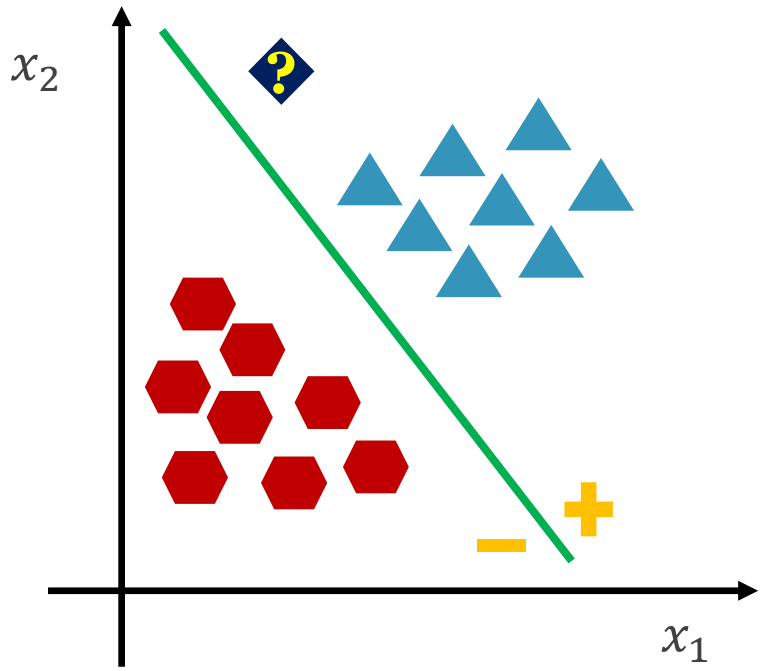
Single Layer Perceptron (SLP)

Contents

- Linear Separable Classification Problems
- Mathematical Formulation
- Cost Function and Optimization
- Batch and Stochastic Learning
- Least Square Approach
- ADALINE and MADALINE

Perceptron Algorithm

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



Perceptron Algorithm

- Problem formulation for D-dimensional data
- Data: $\{\boldsymbol{x}_n, y_n\}_{n=1}^{N}$
- $\boldsymbol{x}_n \in \mathbb{R}^{\mathrm{D}},\ y_n \in \{-1, +1\}$
- $g(\boldsymbol{x}) = \sum_{i=1}^{D} \theta_i x_i + \theta_0 = \theta^T \tilde{\boldsymbol{x}}$
- $\tilde{\boldsymbol{x}} = (x_1 \quad x_2 \quad \dots \quad x_D \quad 1)^T : extened\ data\ vector$
- $\boldsymbol{\theta} = (\theta_1 \quad \theta_2 \quad \dots \quad \theta_D \quad \theta_0)^T$
- $\{\theta_i\}_{i=1}^{\mathrm{D}} : weights$
- $\theta_0 = bias\ or\ threshold$

Perceptron Algorithm

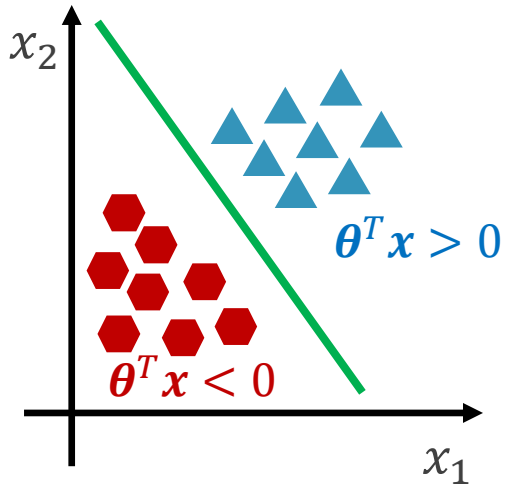
- General formulation:

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

- Optimal Classifier is a hyperplane:

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

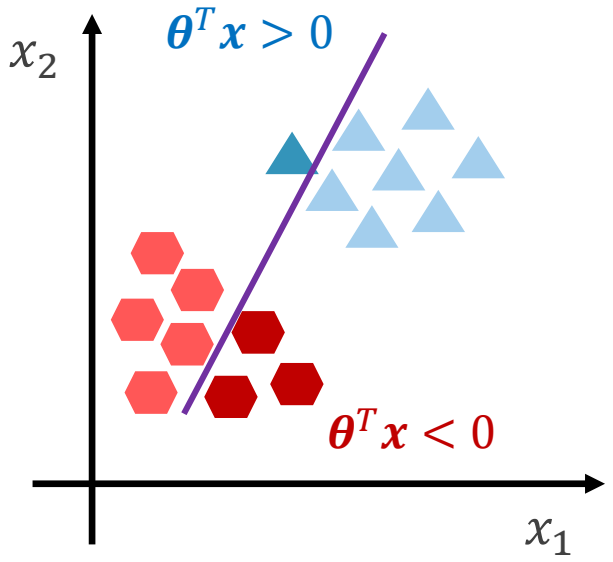
$$\boldsymbol{\theta}_*^T \boldsymbol{x} < 0, \text{ if } \boldsymbol{x} \in \omega_2$$



- 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: $\{\boldsymbol{x}_n, y_n\}_{n=1}^{N},\ y_n \in \{-1, +1\}$
- Assume $\mathcal{Y}$ be the set of all *misclassified* samples.



Perceptron Algorithm – Cost Function

- Cost/Loss function is defined as:

$$J(\boldsymbol{\theta}) = -\sum_{n:\boldsymbol{x}_n \in \mathcal{Y}} y_n \boldsymbol{\theta}^T \boldsymbol{x}_n\,, \qquad y_n = \begin{cases} +1, & \boldsymbol{x} \in \omega_1 \\ -1, & \boldsymbol{x} \in \omega_2 \end{cases}, \qquad (\boldsymbol{\theta}^T \boldsymbol{x}_n)_{n:\boldsymbol{x}_n \in \mathcal{Y}} = \begin{cases} < 0, & \boldsymbol{x} \in \omega_1 \\ \geq 0, & \boldsymbol{x} \in \omega_2 \end{cases}$$

- Observe that (for summation over $\mathcal{Y}$) the cost function is *non-negative*.
- 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 (batch perceptron rule):

$$\boldsymbol{\theta}^{(i)} = \boldsymbol{\theta}^{(i-1)} + \mu_i \sum_{n;x_n\in\mathcal{Y}} y_n\boldsymbol{x}_n$$

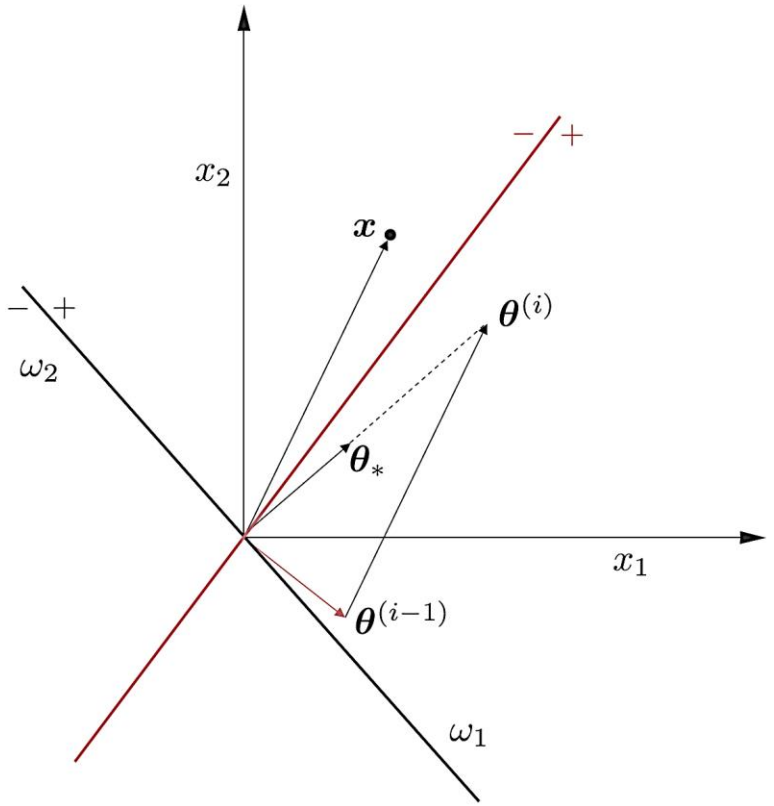
Batch Perceptron Algorithm

- Batch Mode Algorithm

- Hyperparameter selection (here learning rate), μ_0
- Random or zero initialization, $\boldsymbol{\theta}^{(0)}$
- 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 $\mathcal{y} = \emptyset$

How it works

- Assume one point ($\boldsymbol{x}$) misclassified:



Perceptron Algorithm – Convergence

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

$$\lim_{t\to\infty} \sum_{i=0}^{t} \mu_i = \infty$$

$$\lim_{t\to\infty} \sum_{i=0}^{t} \mu_i^2 < \infty$$

Online Perceptron Algorithm

- 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.

Online Perceptron Algorithm

- Sample Mode Perceptron Algorithm:

- Hyperparameter selection (here learning rate), μ_0
- Random or zero initialization, $\boldsymbol{\theta}^{(0)}$
- $i \leftarrow 0$
- Repeat; *Each iteration correspond to an epoch.*
  - $counter \leftarrow 0$; *Counts the number of updates per epoch.*
  - for $n = 1$ to N; *For each epoch, all samples are presented.*
    - if $\left(y_n \boldsymbol{x}_n^T \boldsymbol{\theta}^{(i-1)} \leq 0\right)$ then
      - $i \leftarrow i + 1$
      - $\boldsymbol{\theta}^{(i)} = \boldsymbol{\theta}^{(i-1)} + \mu_0 y_n \boldsymbol{x}_n$
      - $counter \leftarrow counter + 1$
    - end if
  - end for
- until $counter = 0$

Least Mean Square Error Approach

- Consider least square criterion:

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

- Solve via *batch* mode:

$$\boldsymbol{\theta}^{(i)} = \boldsymbol{\theta}^{(i-1)} - \epsilon_{(i)}\sum_{n=1}^{N}\boldsymbol{x}_n\left(\boldsymbol{x}_n^T\boldsymbol{\theta}^{(i-1)} - y_n\right), \qquad \epsilon_{(i)} \leftarrow \frac{\mu_{(i)}}{0.5N}$$

- Solve via *sample* mode:

$$\boldsymbol{\theta}^{(i)} = \boldsymbol{\theta}^{(i-1)} - \epsilon_{(i)}\boldsymbol{x}_i\left(\boldsymbol{x}_i^T\boldsymbol{\theta}^{(i-1)} - y_i\right) = \boldsymbol{\theta}^{(i-1)} - \mu_{(i)}\boldsymbol{x}_i e_i, \qquad \epsilon_{(i)} \leftarrow \frac{\mu_{(i)}}{0.5N}$$

- LMS Widrow-Hopf algorithm
- There is a closed form solution! How?

Least Mean Square Error Approach

- 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

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

LMS vs Perceptron Algorithm

- Perceptron:

$$J(\boldsymbol{\theta}) = -\sum_{n:x_n \in Y} y_n \boldsymbol{\theta}^T \boldsymbol{x}_n\,, \qquad y_n = \begin{cases} +1, & \boldsymbol{x} \in \omega_1 \\ -1, & \boldsymbol{x} \in \omega_2 \end{cases}, \qquad (\boldsymbol{\theta}^T \boldsymbol{x}_n)_{n:x_n \in Y} = \begin{cases} < 0, & \boldsymbol{x} \in \omega_1 \\ \geq 0, & \boldsymbol{x} \in \omega_2 \end{cases}$$

- LMS:

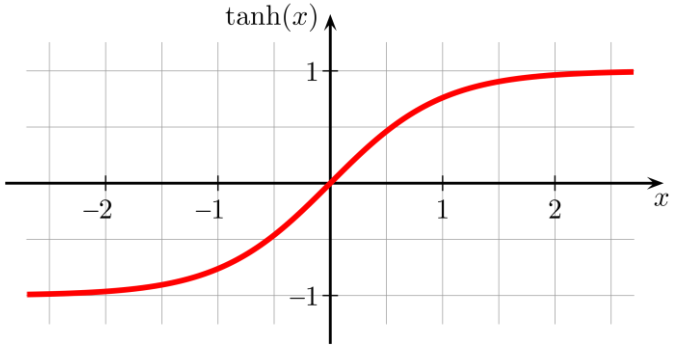
$$J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N} (\boldsymbol{\theta}^T \boldsymbol{x}_n - y_n)^2$$

- $\boldsymbol{\theta}^T \boldsymbol{x}_n$ and y_n have significantly different dynamic ranges.
- Merge two approaches:

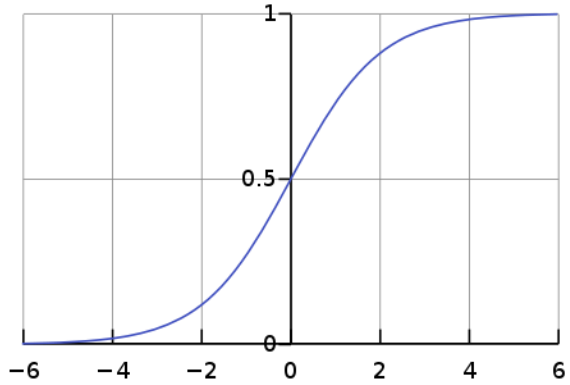
$$J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N} (f(\boldsymbol{\theta}^T \boldsymbol{x}_n) - y_n)^2\,, \qquad f: smoothed\ sign\ function$$

LMS vs Perceptron Algorithm

- Smoothed sign function:

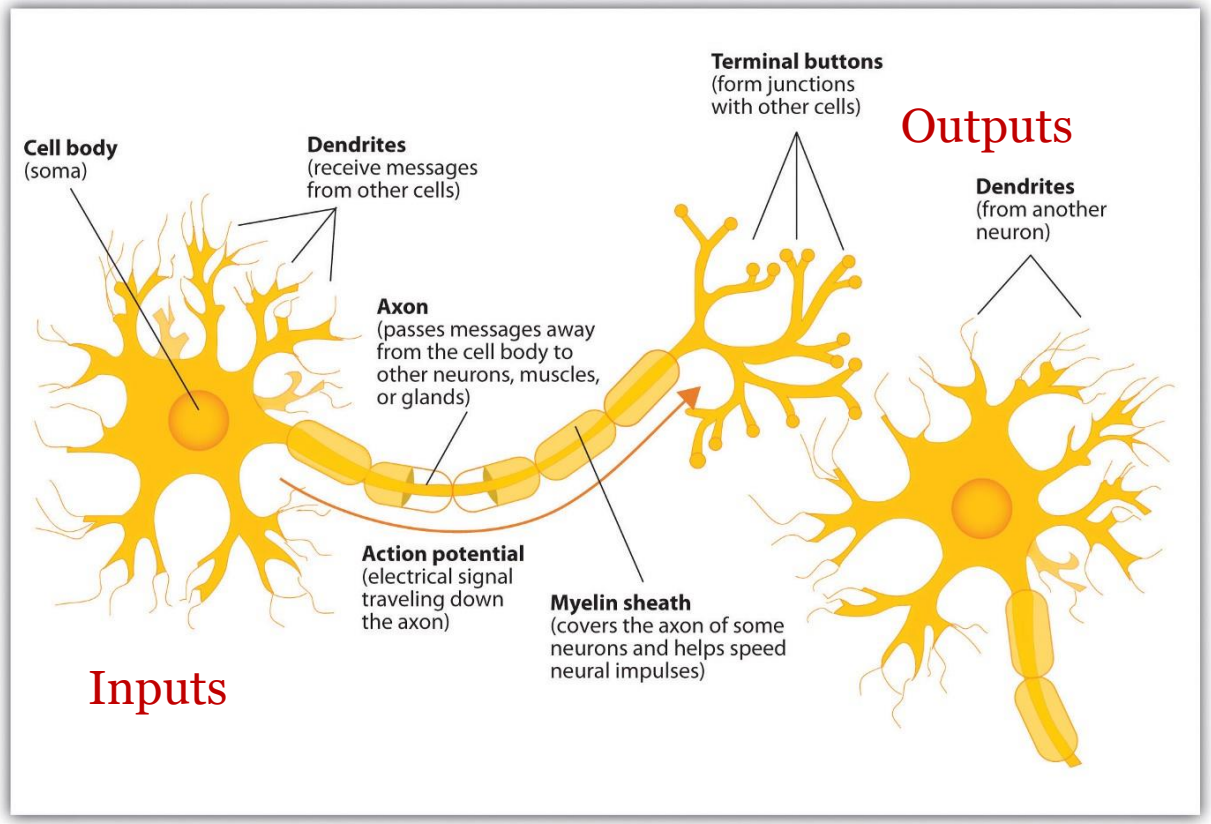


- For $y_n \in \{0, 1\}$ we may use smoothed step (Heaviside) function:



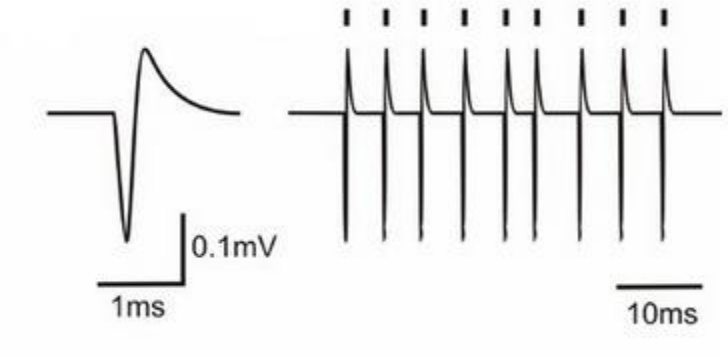
Single Layer Perceptron (SLP)

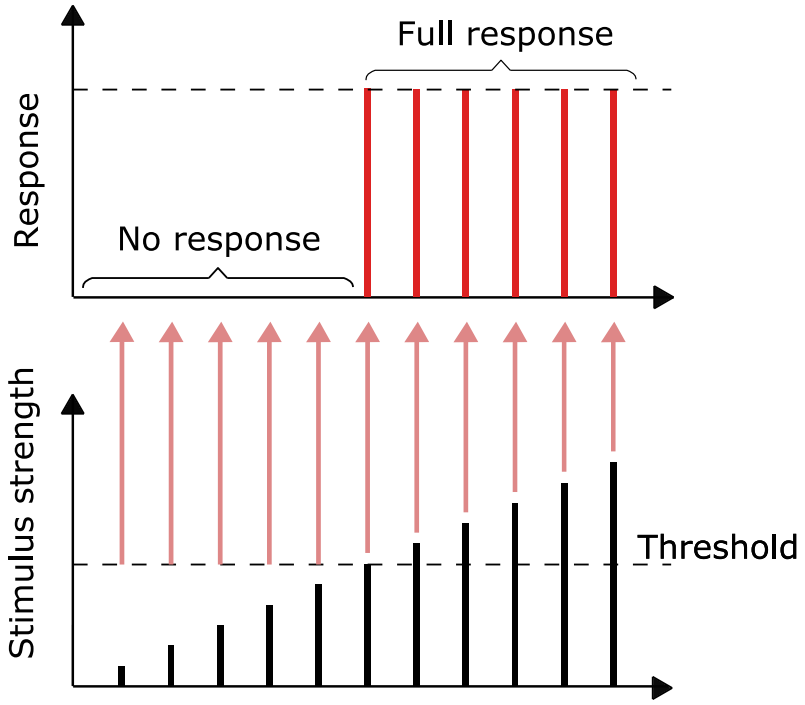
- Biologically Inspired
- Neurons
  - Inputs
  - Weights (Synapses)
  - Connection
  - Functionality:
    - All-or-None Law
    - Threshold
    - Frequency Modulated Response



Single Layer Perceptron (SLP)

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

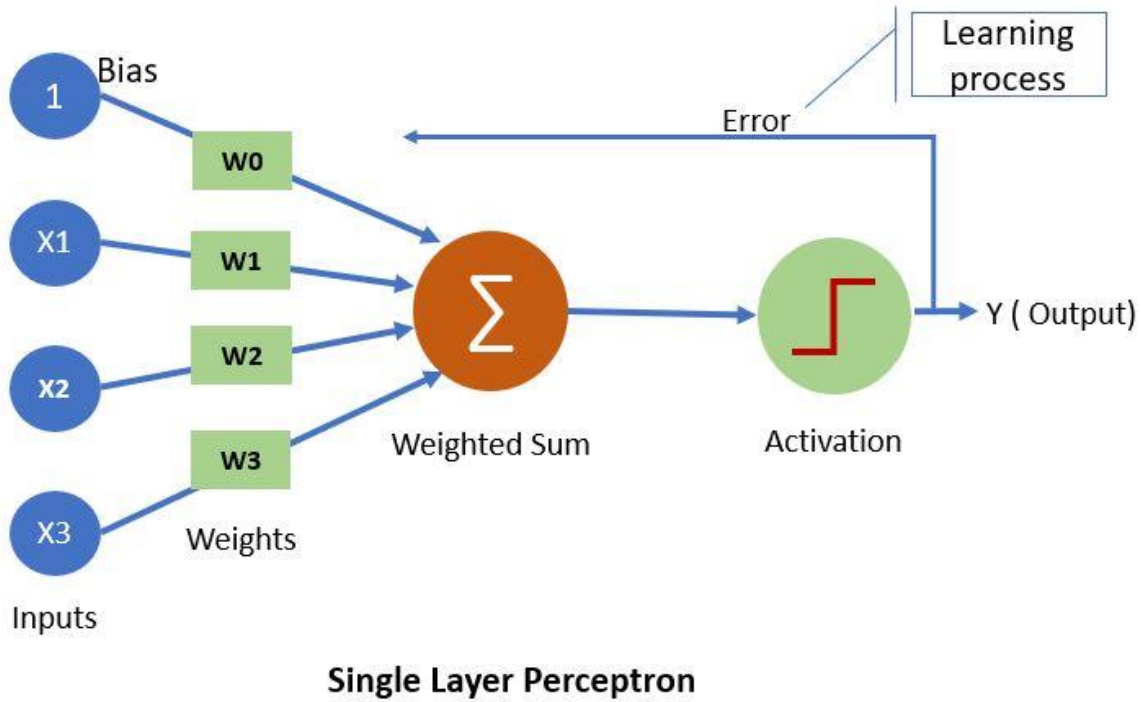




Rosenblatt Model or Adaptive Linear Element (ADALINE)

- Arbitrary (real) inputs
- Arbitrary (real) weights
- Arbitrary (real) bias (threshold)
- Step or Sign activation function

$$f(x) = \begin{cases} 1\ (+1), & \boldsymbol{w}^T x + b \geq 0 \\ 0\ (-1), & \boldsymbol{w}^T x + b < 0 \end{cases}$$



Single Layer Perceptron

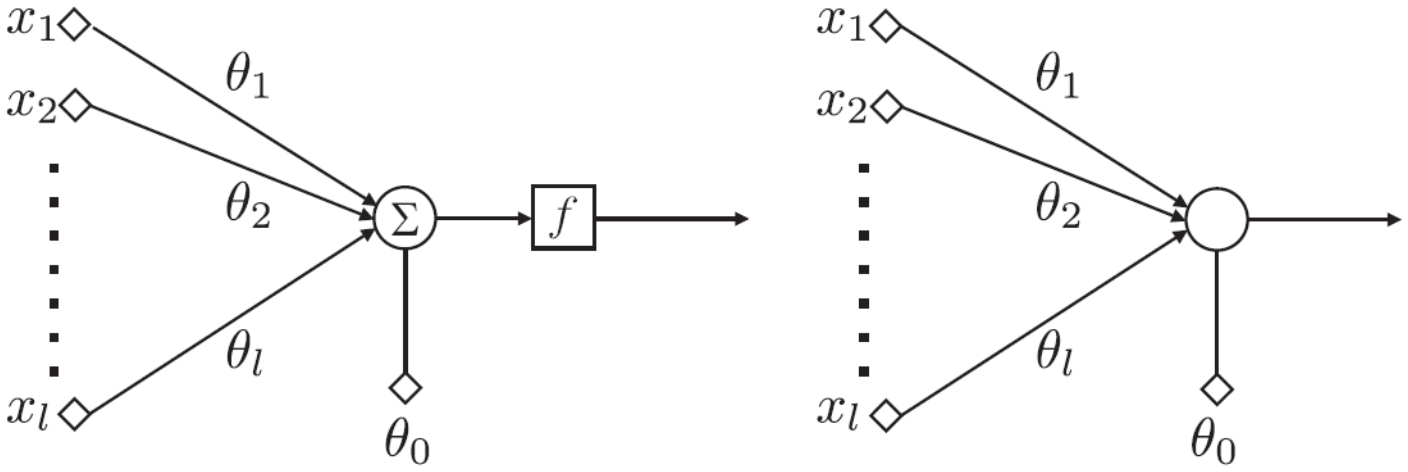
- Alternative name: ADAptive LInear Elements (ADALINE)

Perceptron with nonlinear activation

- Assume we have a differentiable activation function!

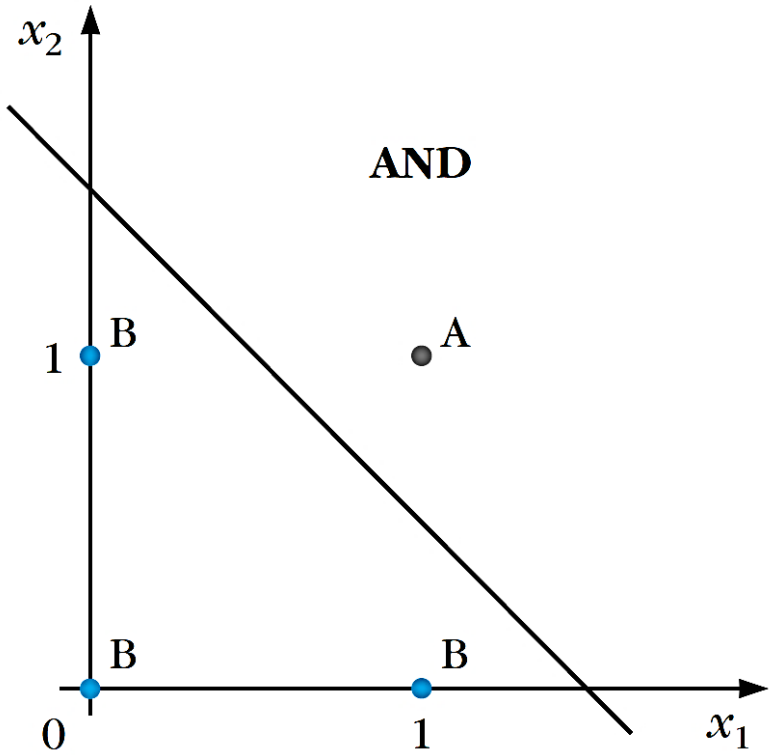
$$J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N}(f(\boldsymbol{x}_n^T\boldsymbol{\theta}) - y_n)^2 \xrightarrow{sample\ mode} \boldsymbol{\theta}^{(i)} = \boldsymbol{\theta}^{(i-1)} - \epsilon_{(i)}\boldsymbol{x}_i e_i f'\left(\boldsymbol{x}_i^T\boldsymbol{\theta}^{(i-1)}\right)$$

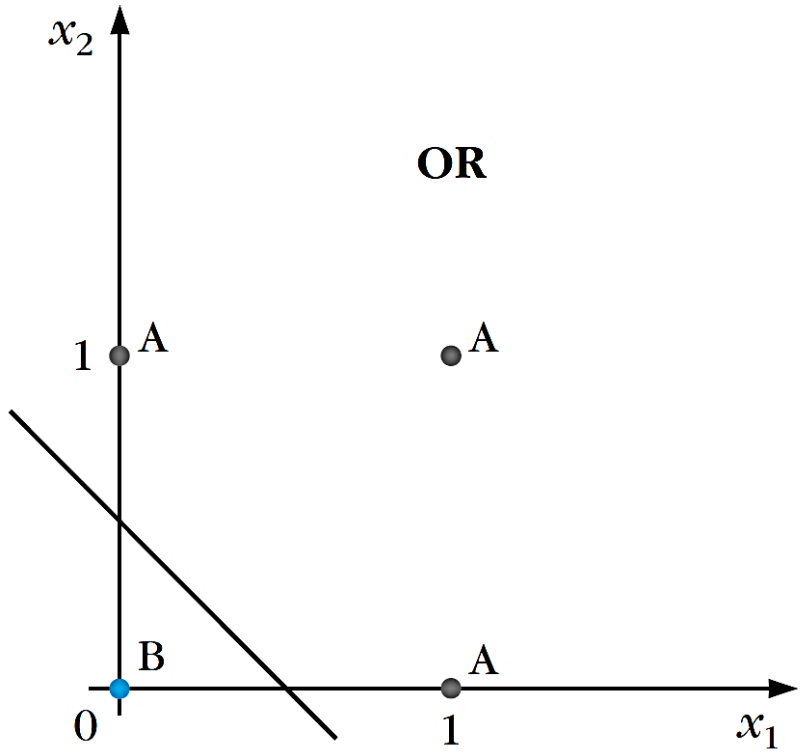
- $e_i = f(\boldsymbol{x}_n^T\boldsymbol{\theta}) - y_n$
- Neuron activity: $\boldsymbol{\theta}^T\boldsymbol{x}$
- Neuron output: $f(\boldsymbol{\theta}^T\boldsymbol{x})$



SLP Capacity

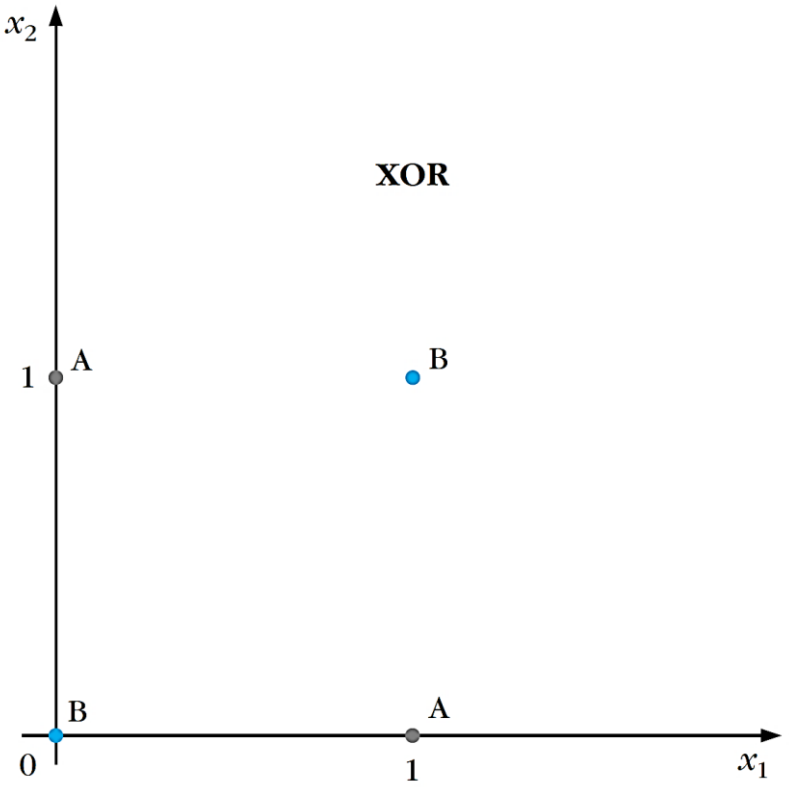
- Let's solve AND and OR problem: it is possible!





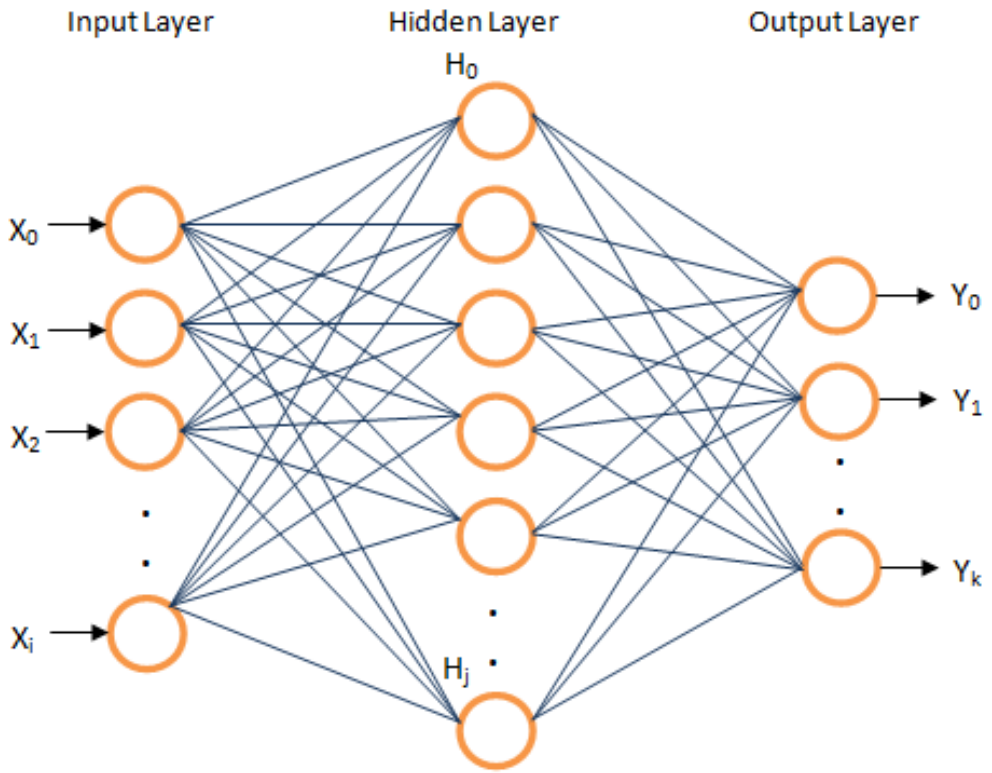
SLP Capacity

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



Multioutput SLP (MADALINE)

- Many ADALINE (MADALINE):



Multilayer Perceptron (MLP)

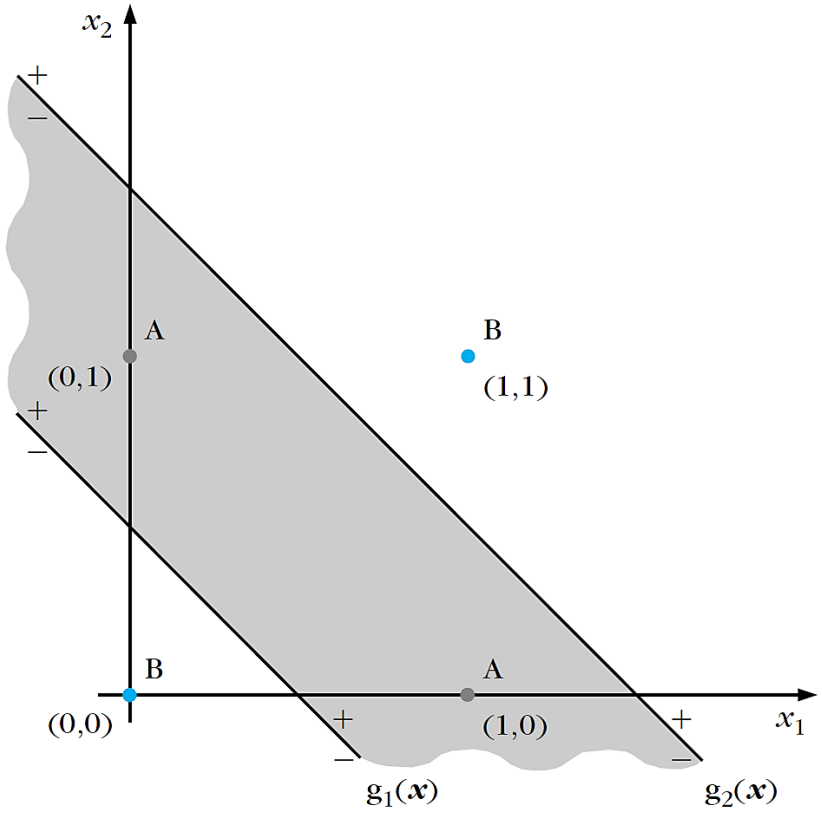
Contents

- Two Layer Perceptron and Capacity
- Three Layer Perceptron and Capacity
- Multi Layer Perceptron and Capacity
- MLP Design
- Activation Function
- Loss Function and Activation Function in Regression Task
- Loss Function and Activation Function in Classification Task
- Error Back Propagation (EBP)
- Generalization Theorem

Two-Layer Perceptron

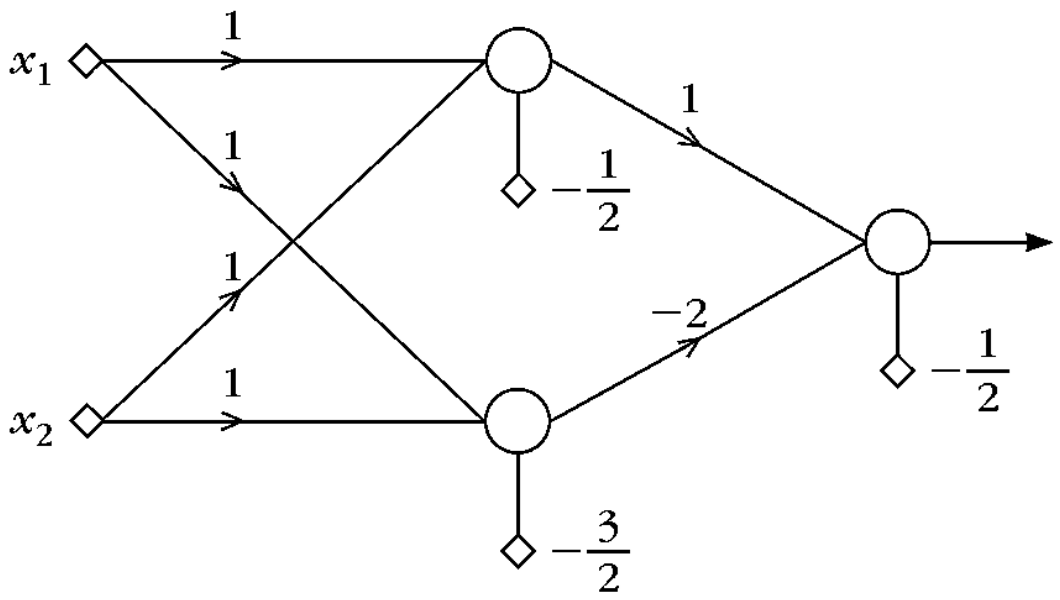
- Now try two-layer perceptron!

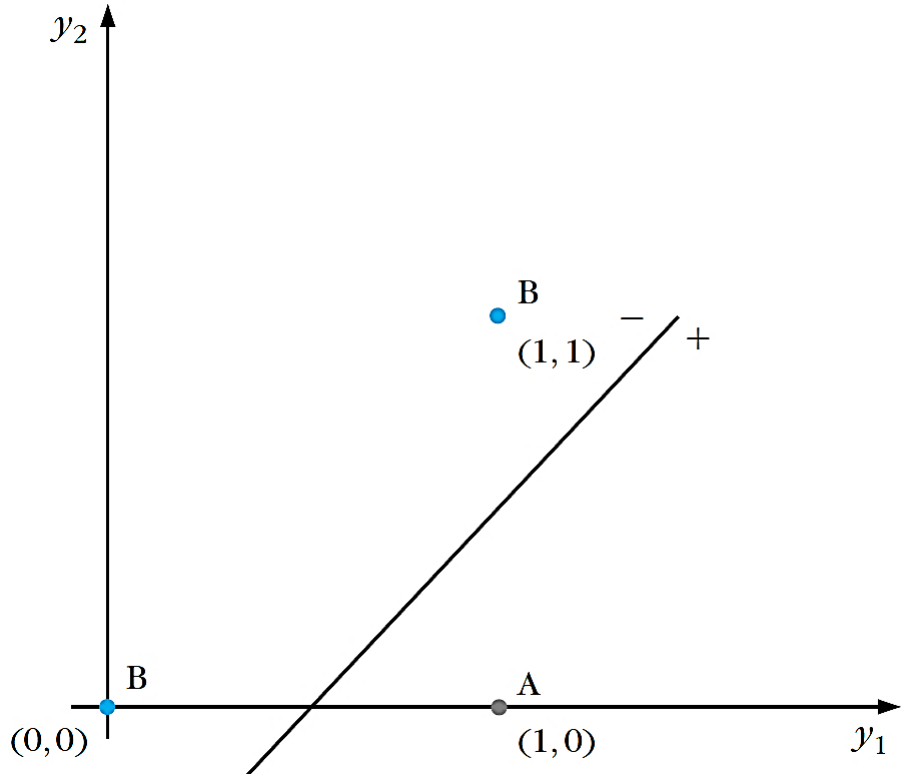
1st Phase				2nd Phase
x_1	x_2	y_1	y_2	
0	0	0 (−)	0 (−)	B (0)
0	1	1 (+)	0 (−)	A (1)
1	0	1 (+)	0 (−)	A (1)
1	1	1 (+)	1 (+)	B (0)



Two-Layer Perceptron

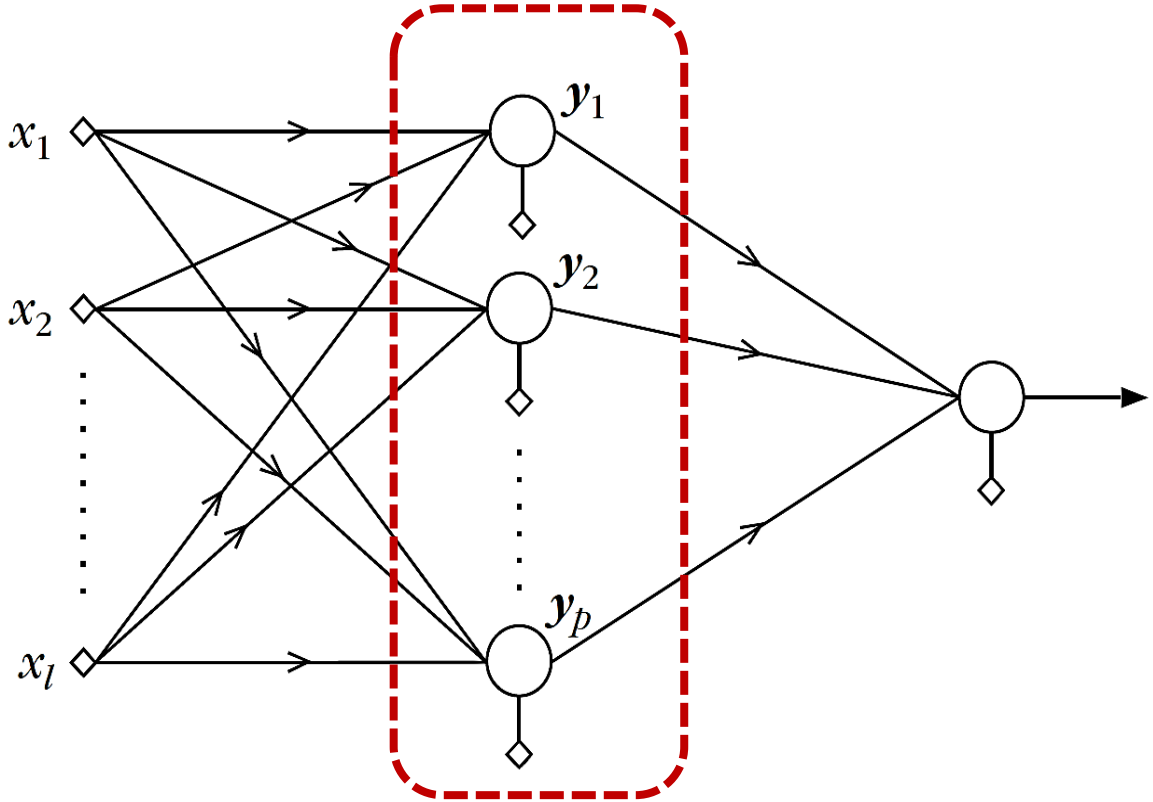
- Now try two-layer perceptron!





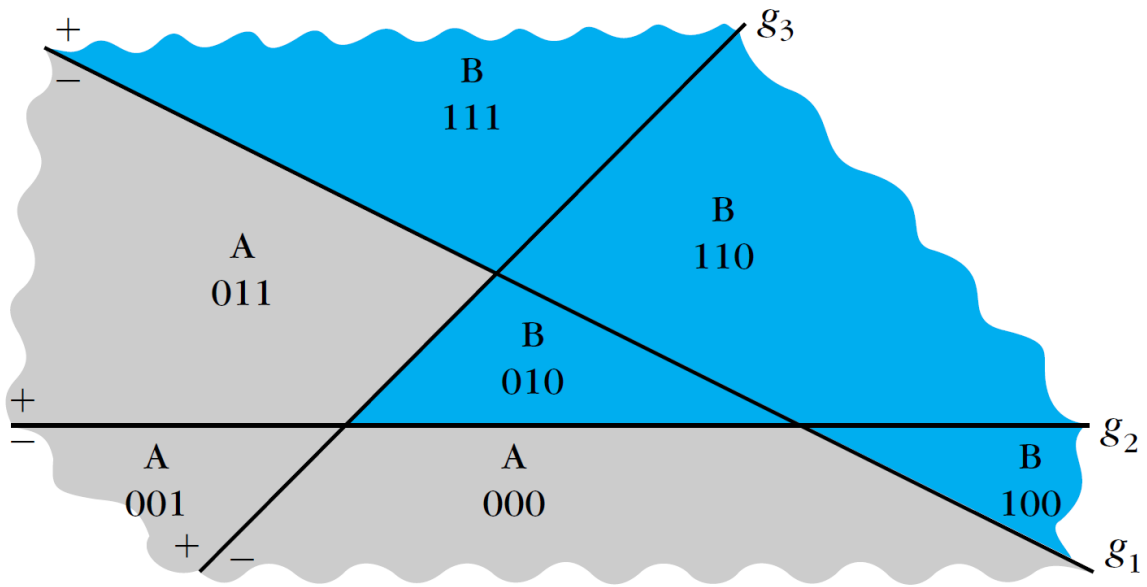
Two-Layer Perceptron

- Two-Layer perceptron:
  - Input layer (1): 1
  - Output layer (1): 1
  - Hidden layer(≥ 0): 1



Two-Layer Perceptron Capacity

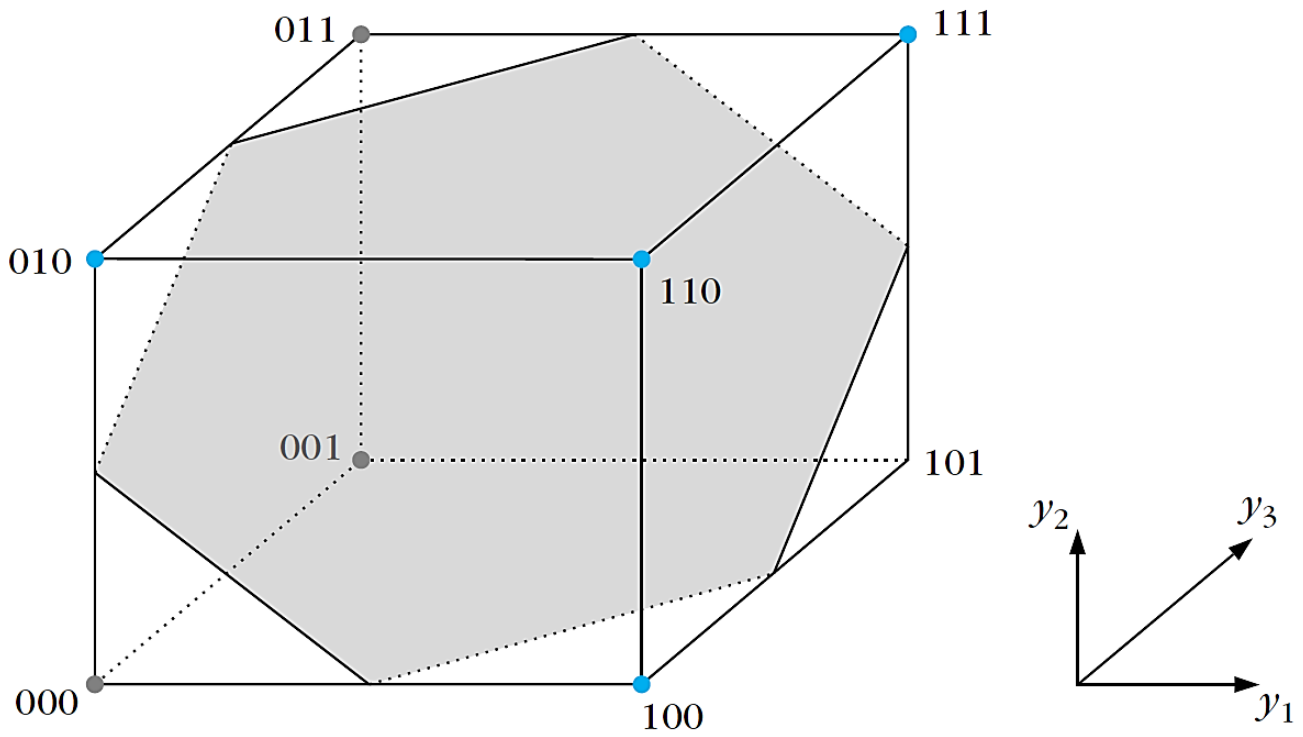
- Two-Layer perceptron (p=3)



- 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)

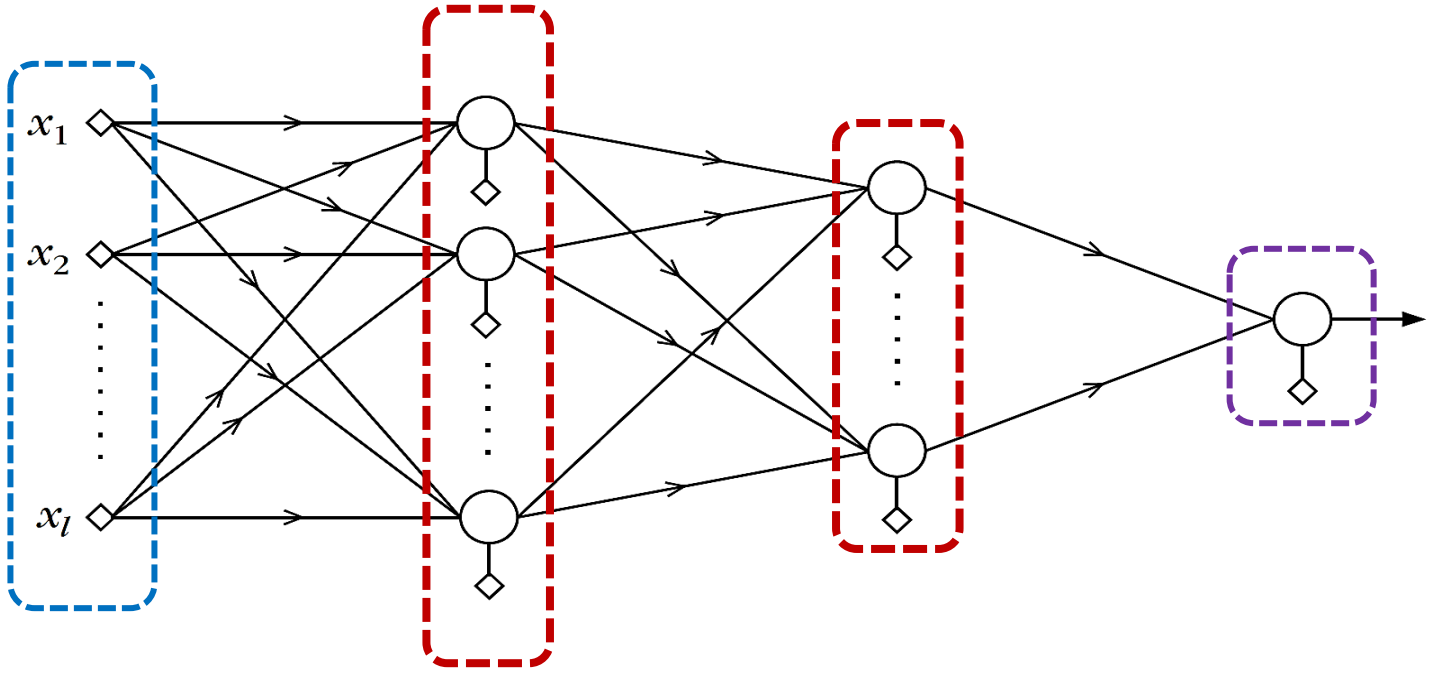
Two-Layer Perceptron Capacity

- Two-Layer perceptron (p=3):



Three-Layer Perceptron

- Let have three-layer perceptron!
- Input Layer (1)
- Hidden Layer (2)
- Output Layer(1)



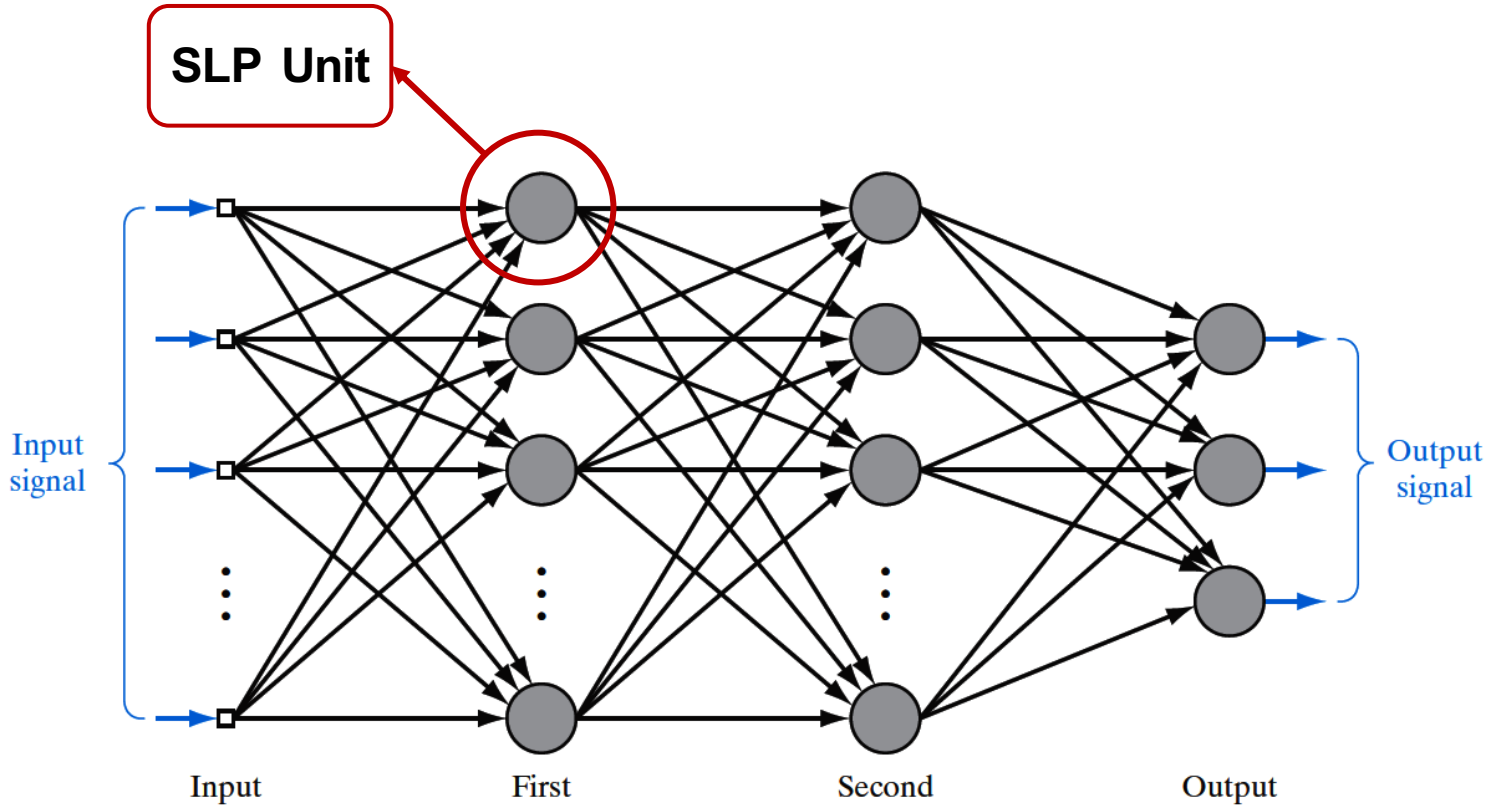
- Can separate classes resulting from any union of polyhedral regions

Multi-Layer Perceptron (MLP)

- Vocabulary
  - Multi Layer Perceptron (MLP)
  - Shallow Neural Network
  - Fully Connected (FC) Layer
  - Dense Layer
  - Feed Forward Artificial Neural Network

Multi-Layer Perceptron (MLP)

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



MLP Transfer Function

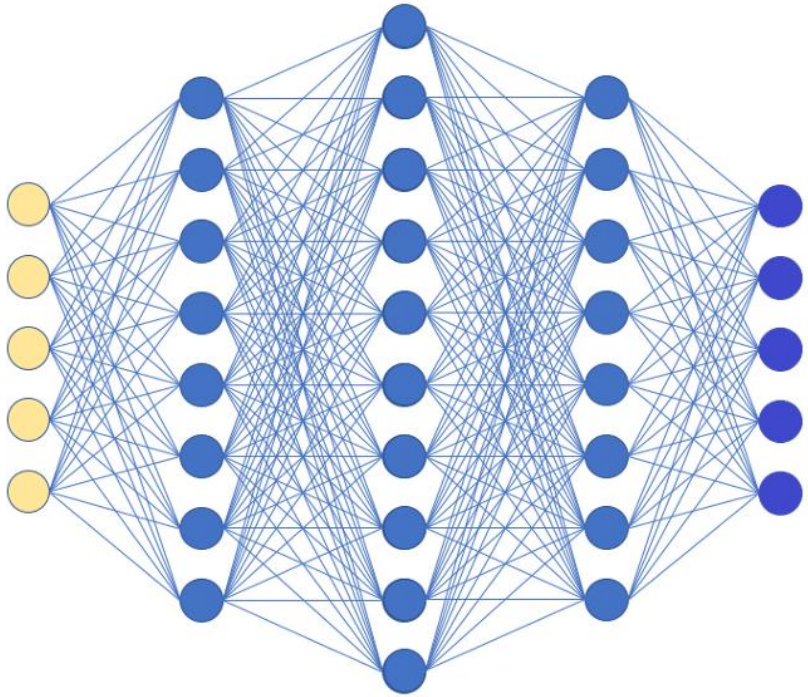
- MLP Transfer function is Nonlinear Mapping from «Input» to «Output»:

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

- where $\boldsymbol{W}$ is set of free parameters.
- Hyperparameters: # of layers, # of neurons in each hidden layer
- MLP is general solution for:
  - Classification
  - Regression (function approximation)

MLP Design

- We should select:
  - Activation function (hidden layer and output layer)
  - Loss function to reach the goal
  - Number of hidden layer (here 3)
  - Number of neuron in each hidden layer (here 8-10-8)



MLP Training

- There are old fashion constructive algorithms to build Multi Layer Perceptron (MLP) for classification tasks → Huge architecture
- Error minimization techniques needs differentiability of cost function

$$Loss(\boldsymbol{W}) = \sum_{n} Loss(\mathcal{F}(\boldsymbol{x}_n; \boldsymbol{W}), \boldsymbol{y}_n)$$

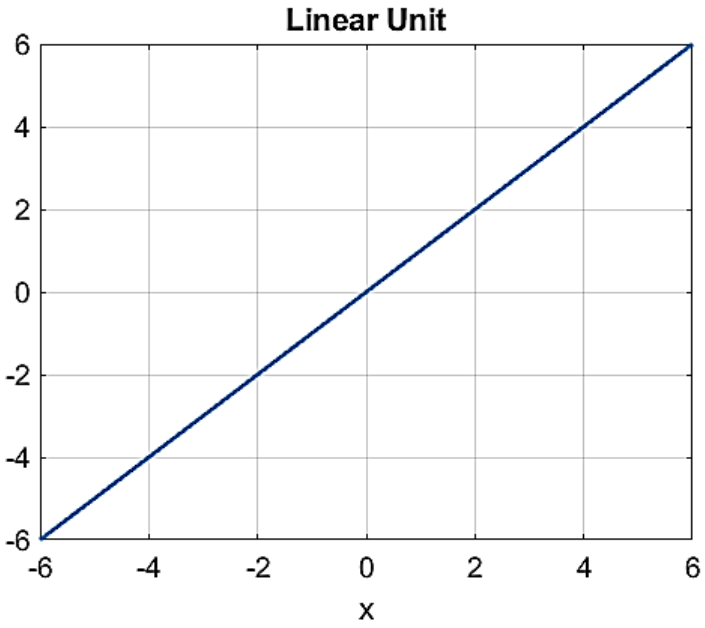
- Differentiability of loss function is related to differentiability of activation functions.
- Solution: smooth step/sign function

Activation Functions - Linear

- Linear function:

$$f(x) = x$$
$$f'(x) = 1$$

- Unlimited output
- Positive and Negative output
- Zero at origin

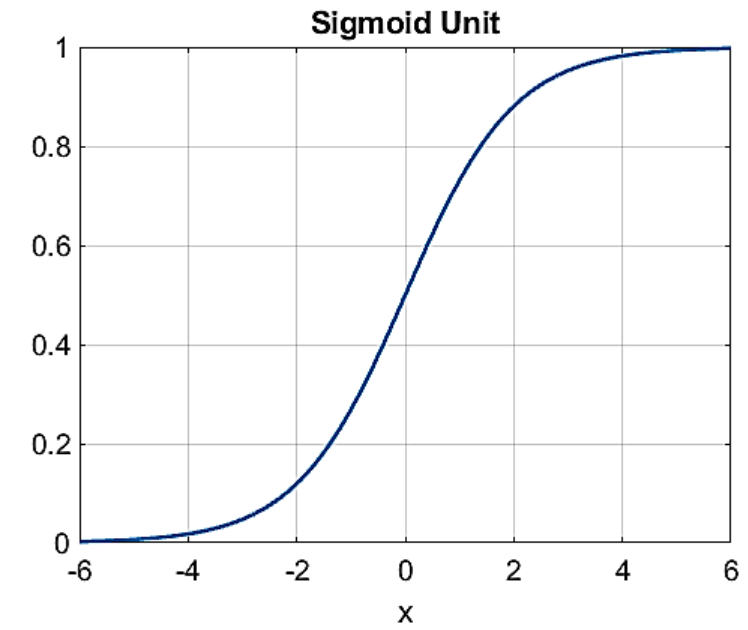


Activation Functions - Sigmoid

- Sigmoid function (soft step)

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

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



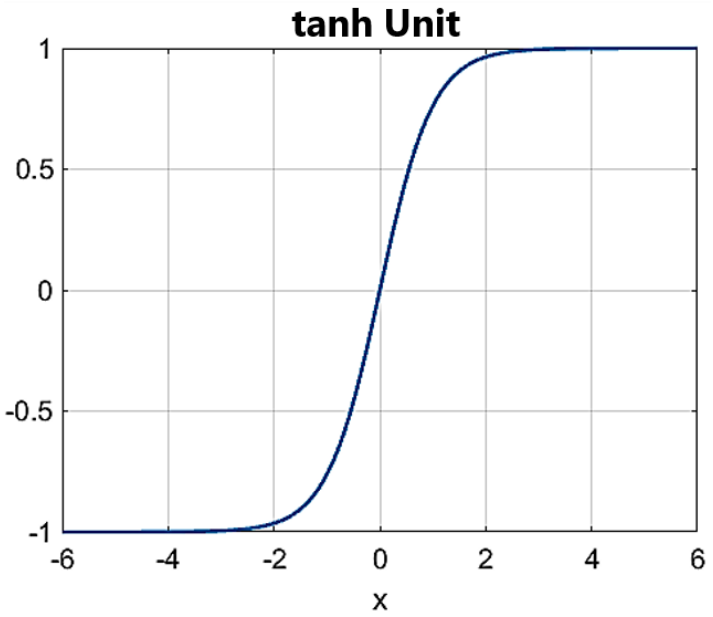
- Smooth gradient
- Output values bounded
- Vanishing gradient
- Non-Zero at origin
- Computationally expensive

Activation Functions - tanh

- Hyperbolic tangent function (soft sign):

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

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



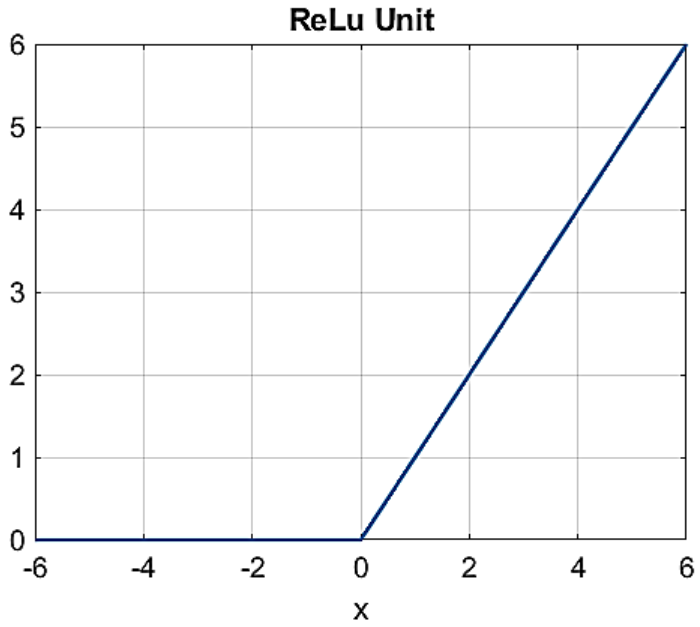
- Smooth gradient
- Output values bounded
- Vanishing gradient
- Zero at origin
- Computationally expensive

Activation Functions - ReLU

- 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
- Unit gradient
- Computationally efficient
- Inactive near origin or negative
- Problem with negative inputs

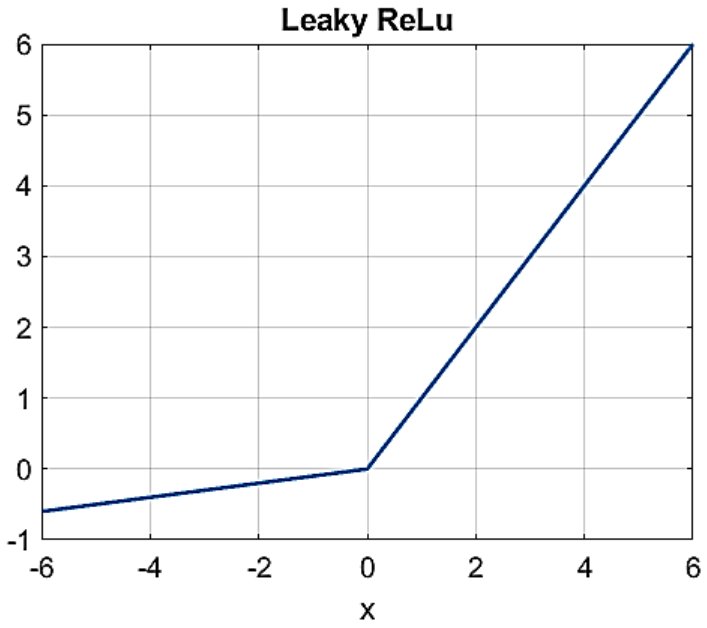
Activation Functions – Leaky ReLu

- Leaky ReLu:

$$f(x) = max(ax, x), \qquad a \propto 0.01$$

$$f'(x) = u(x) + \alpha u(-x)$$

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

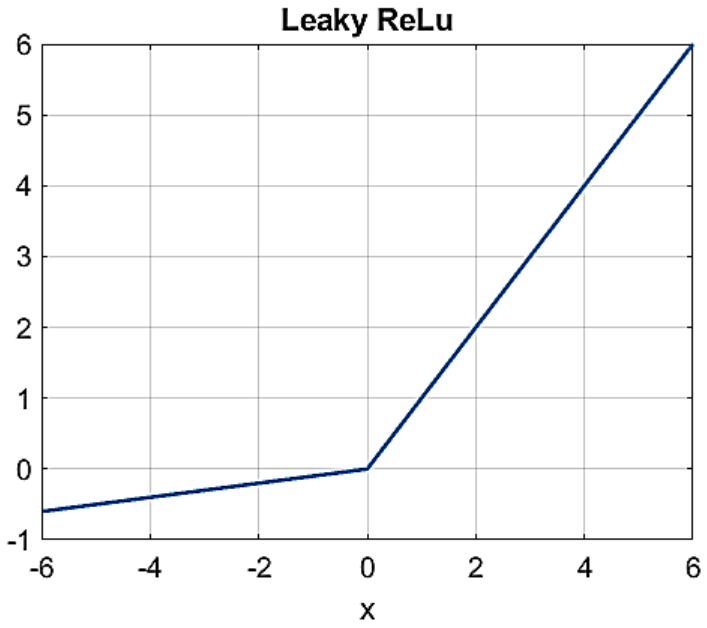


Activation Functions – Parametric ReLu

- Parametric ReLu:

$$f(x) = max(wx, x)$$
$$f'(x) = u(x) + wu(-x)$$

- Non-linear
- Computationally efficient
- Active near origin or negative
- Problem with negative inputs
- *Learnable* for negative values

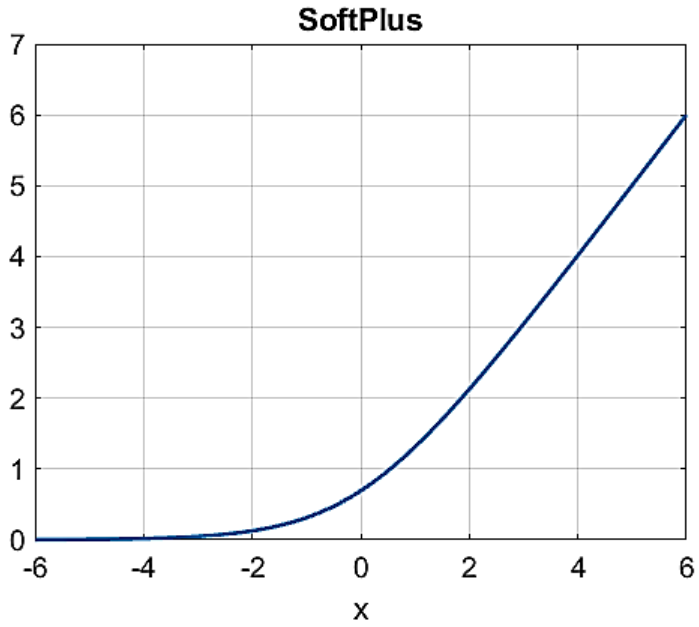


Activation Functions - SoftPlus

- Softplus (smooth ReLu):

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

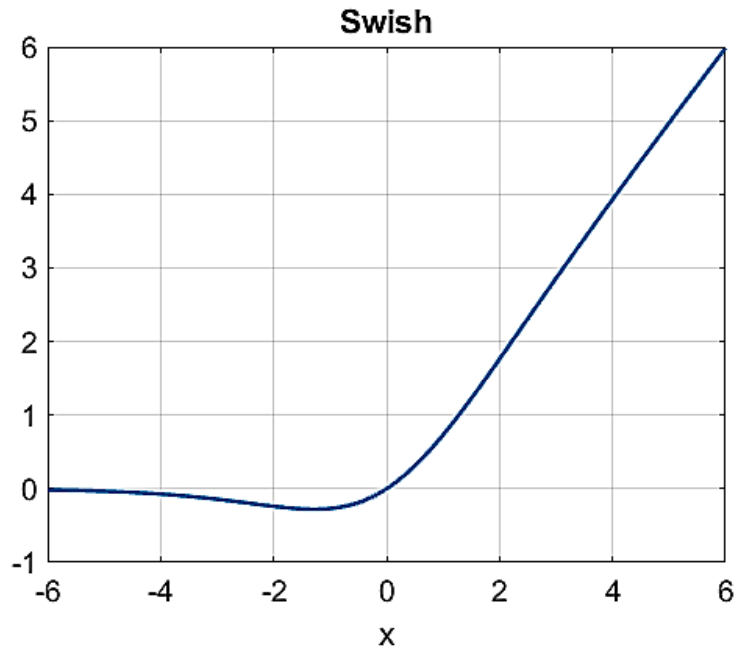
$$f'(x) = \frac{1}{1 + e^{-x}} = \sigma(x)$$



Activation Functions – SiLU and Parametric SilU

- Sigmoid Linear Units (SiLU) or swish and trainable SiLU:

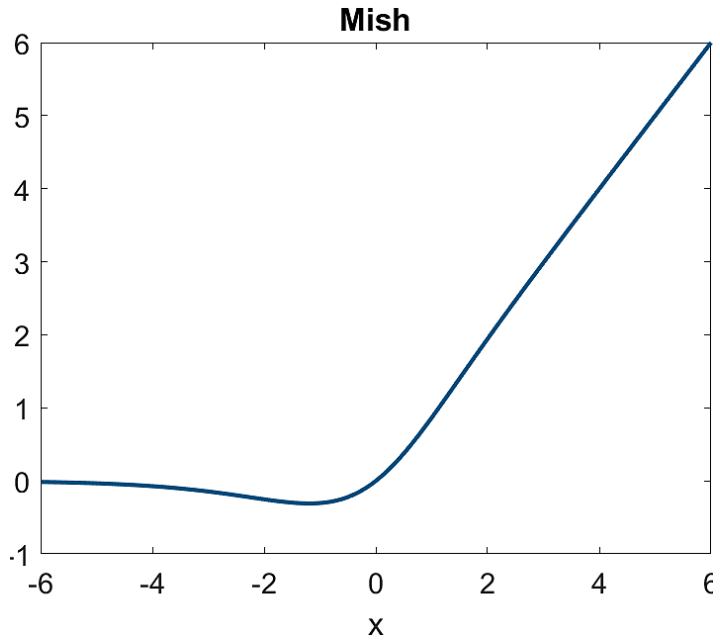
$$f(x) = x\sigma(\beta x) = \frac{x}{1 + e^{-\beta x}}$$



Activation Functions – Mish

- Mish is also smoothed ReLU:

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

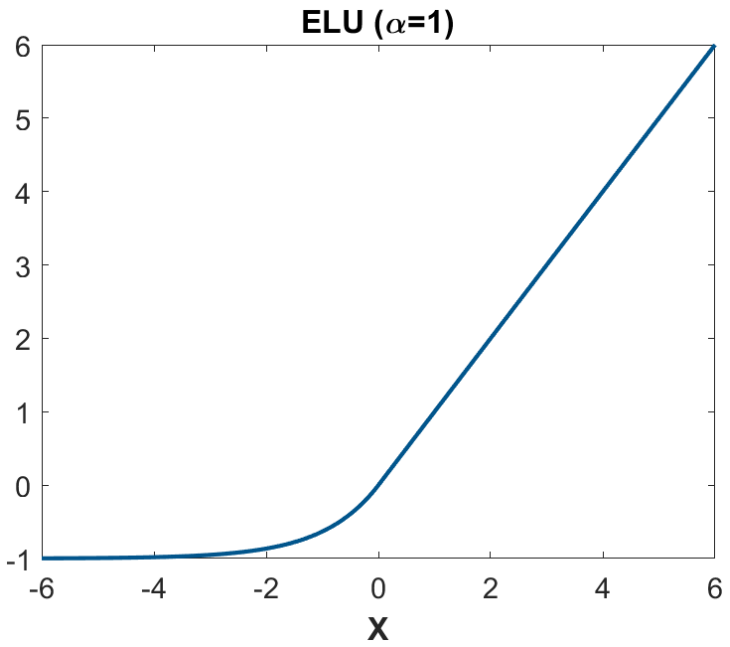


Activation Functions – ELU and SELU

- Exponential Linear Unit (ELU) and Scaled Exponential ReLU (SELU):

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

- ELU: $\beta = 1$
- SELU: $\beta \neq 1$



Activation Functions – GELU and ISRLU

- Gaussian Error Linear Unit (GELU) :

$$f(x) = 0.5x\left(1 + \tanh\left(\sqrt{2/\pi}(x + 0.044715x^3)\right)\right)$$

- Inverse Square Root Linear Unit (ISRLU):

$$f(x) = \begin{cases} x, & x \geq 0 \\ \dfrac{x}{\sqrt{1 + \alpha x^2}}, & x \leq 0 \end{cases}$$

Activation Functions – SoftMax

- Softmax (classification, single label and multiclass task):

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

- $0 < f_i(x) < 1$
- $\sum_{k=1}^{m} f_k(x) = 1$
- β may be fixed or trainable

«Loss/Cost» functions and «output» activation functions

- 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» functions

- Mean Square Error (MSE or L2 norm):

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

- Mean Absolute Error (MAE or L1 norm):

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

- Mean Huber Error (MHE):

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

Regression «Loss/Cost» functions

- Pseudo Mean Huber Error (SMHE):

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

Regression «output» activation functions

- There are two choices:
  - Linear (real output), $output \in \mathcal{R}$
  - ReLu (positive real output) , $output \in \mathcal{R}^{+}$

Binary Classification (one binary output) Task

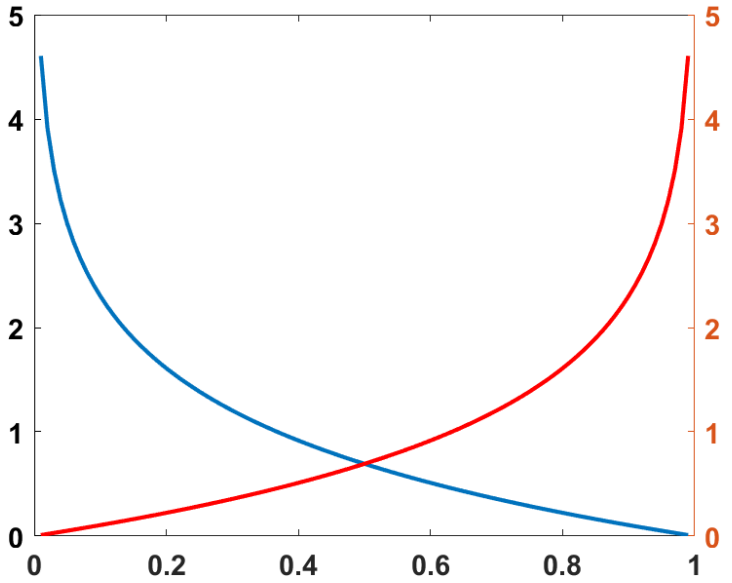
- Definition:
  - $N: \#\ of\ samples$
  - $y_i \in \{0,1\}: Desired\ output\ (target)$
  - $\hat{y}_i \in [0\ 1] \subset \mathbb{R}: Actual\ output$

Binary Classification «Loss/Cost» Function

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

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

- BCE plot for $y_i = 0$ and $y_i = 1$



Binary Classification «Loss/Cost» Function

- 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):

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

Binary Classification «Loss/Cost» Function

- 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):

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

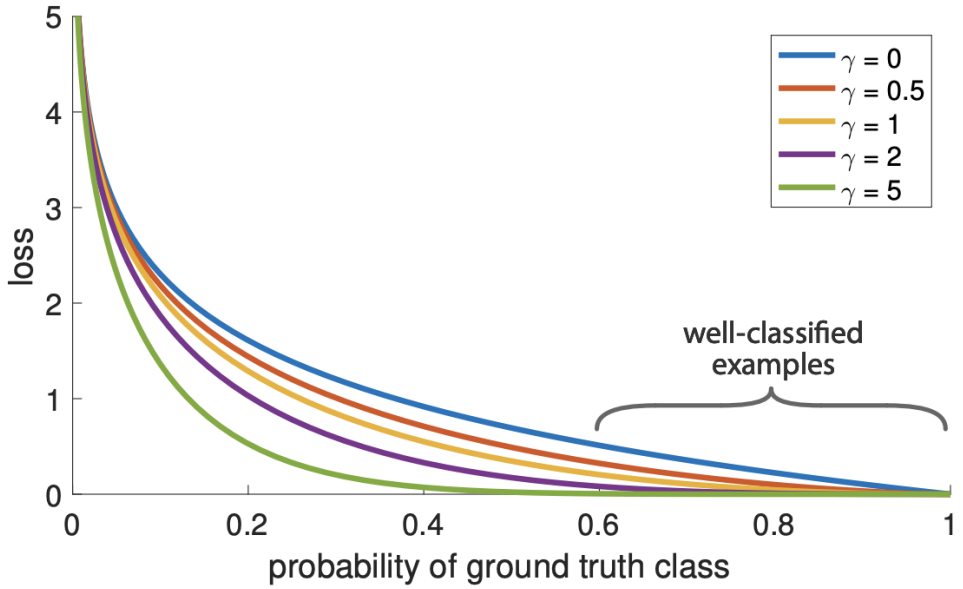
- Smoothed Hing Loss (from SVM literature):

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

Others - Focal Loss

- Focal Loss: Addresses *class imbalance* during training in tasks like object detection. Focal loss applies a modulating term to the cross entropy loss in order to focus learning on *hard misclassified examples*. It is a dynamically scaled cross entropy loss, where the scaling factor decays to zero as confidence in the correct class increases

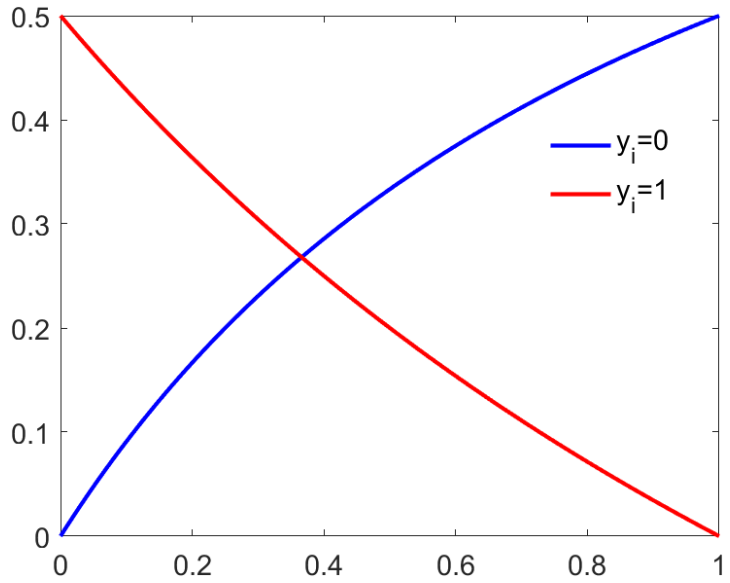
$$FL(\hat{y}_i;\gamma) = -(1-\hat{y}_i)^{\gamma} log(\hat{y}_i)$$
$$FL(\hat{y}_i;0) = CE(\hat{y}_i)$$



Others – Dice Loss

- Dice Loss: Widely used in medical image segmentation tasks to address the data imbalance problem.

$$DL(y_i, \hat{y}_i) = 1 - \frac{2y_i\hat{y}_i + 1}{y_i + \hat{y}_i + 1}$$



Others – Tversky Loss

- Tversky Loss: Can be seen as an generalization of Dices coefficient. It adds a weight to FP (false positives) and FN (false negatives) with the help of β coefficient.

$$TL(y_i, \hat{y}_i) = 1 - \frac{y_i\hat{y}_i + 1}{y_i\hat{y}_i + \beta(1 - y_i)\hat{y}_i + (1 - \beta)y_i(1 - \hat{y}_i) + 1}$$

Binary Classification «output» activation Function

- There are two choices:
  - 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):

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

  - $\hat{\boldsymbol{y}}_i \in [0\ 1]^M$: $Actual\ outputs$

Multi-Class Classification «Loss/Cost» Function

- 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):

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

- Kullback–Leibler (KL):

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

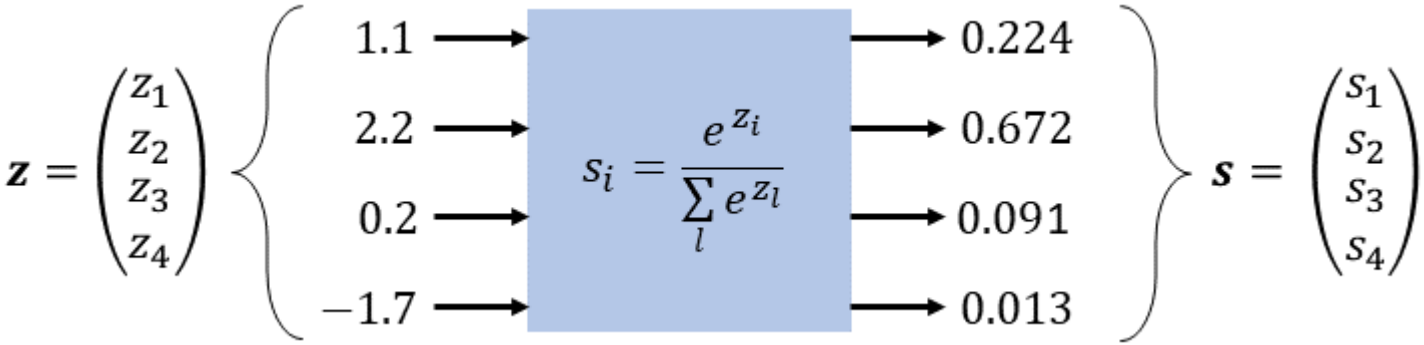
- where $y_{i,k}$ is kth element of target vector, and $\hat{y}_{i,k}$ is kth element of output vector.

Multi-Class Classification «output» Activation Function

- Most used output activation function is softmax:

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

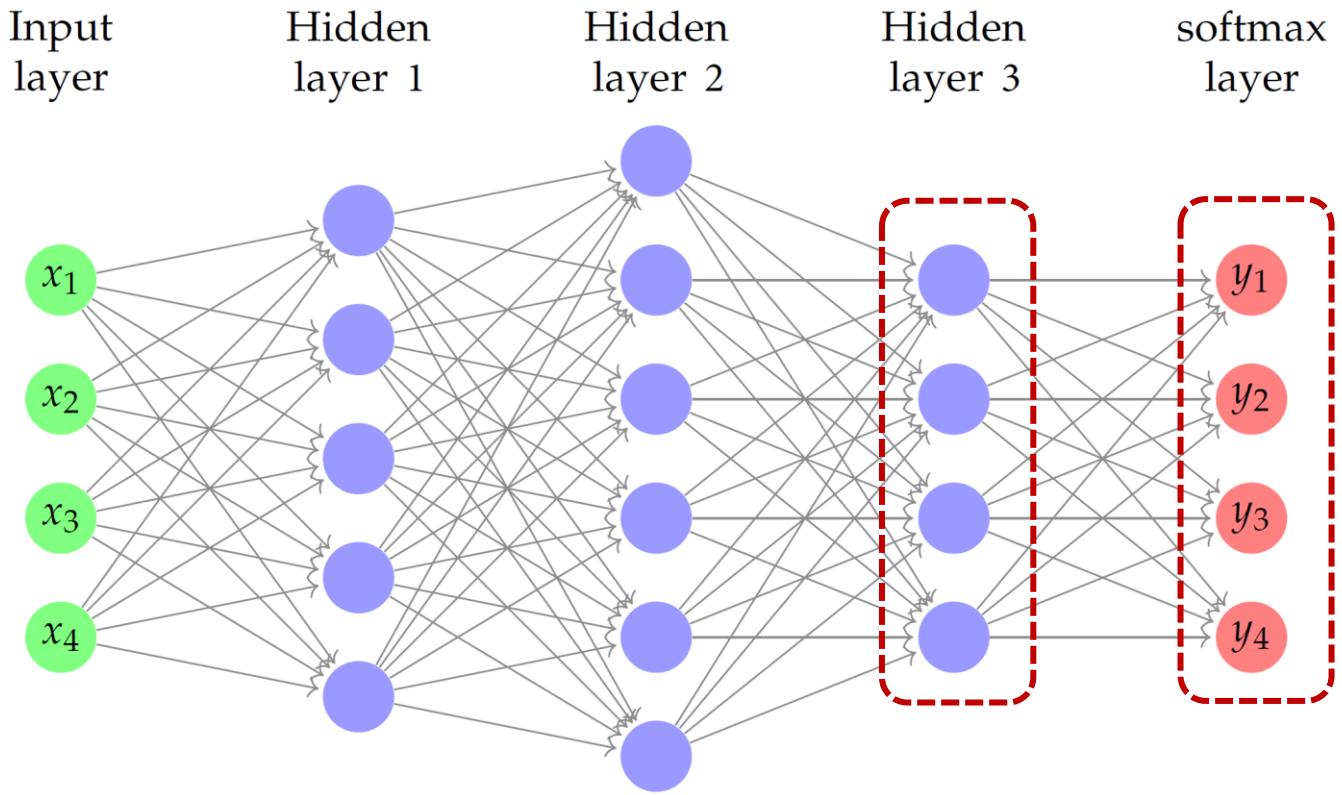
- where $z_{i,k}$ is output of previous layer.
- Example:



$$-1.7 < 0.2 < 1.1 < 2.2 \Rightarrow 0.013 < 0.091 < 0.224 < 0.672$$

Multi-Class Classification «output» Activation Function

- Softmax layer:



Multi-Class Classification «output» Activation Function

- 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):

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

  - $\hat{\boldsymbol{y}}_i \in [0\ 1]^M$: $Actual\ output$

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

MLP Training

- Input-Target pairs (training samples):

$$\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^{N}$$

- Input-Output relationship:

$$\hat{\boldsymbol{y}}_i = \mathcal{F}(\boldsymbol{x}_i; \boldsymbol{\theta})$$

- Loss functions (batch, minibatch, sample):
  - $Loss_{Batch}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} L(\boldsymbol{y}_i, \hat{\boldsymbol{y}}_i) = \frac{1}{N}\sum_{i=1}^{N} L\big(\boldsymbol{y}_i, \mathcal{F}(\boldsymbol{x}_i; \boldsymbol{\theta})\big)$
  - $Loss_{Minibatch}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{y}_i, \hat{\boldsymbol{y}}_i) = \frac{1}{m}\sum_{i=1}^{m} L\big(\boldsymbol{y}_i, \mathcal{F}(\boldsymbol{x}_i; \boldsymbol{\theta})\big)$
  - $Loss_{Sample}(\boldsymbol{\theta}) = L(\boldsymbol{y}_i, \hat{\boldsymbol{y}}_i) = L\big(\boldsymbol{y}_i, \mathcal{F}(\boldsymbol{x}_i; \boldsymbol{\theta})\big)$
- where $L(\boldsymbol{y}_i, \hat{\boldsymbol{y}}_i)$ is sample loss (BCE, SD, AD, ...)

MLP Training

- Solution (GD or SGD):

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \epsilon^{(t)} \left. \frac{\partial Loss(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right|_{\boldsymbol{\theta}=\boldsymbol{\theta}^{(t)}}$$

- Need a systematic way to compute loss derivatives with respect to all weights in all layers, it is:

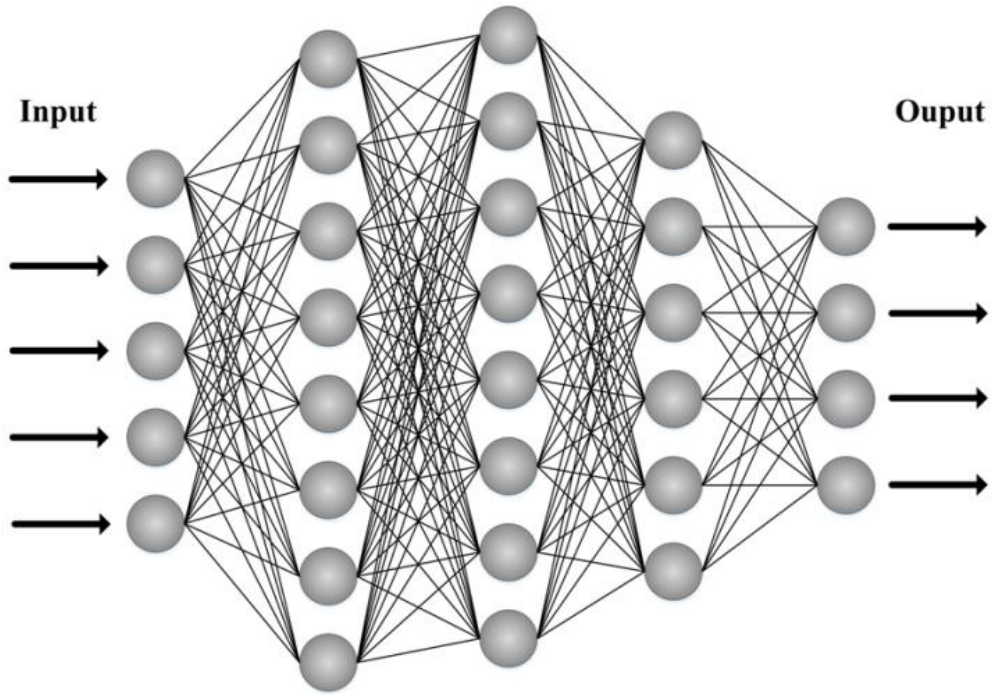
Error Back Propagation (EBP)

Error Back Propagation Algorithm - EBP

- $\{\boldsymbol{x_n}, \boldsymbol{y}_n\}_{n=1}^{N}$: 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 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.

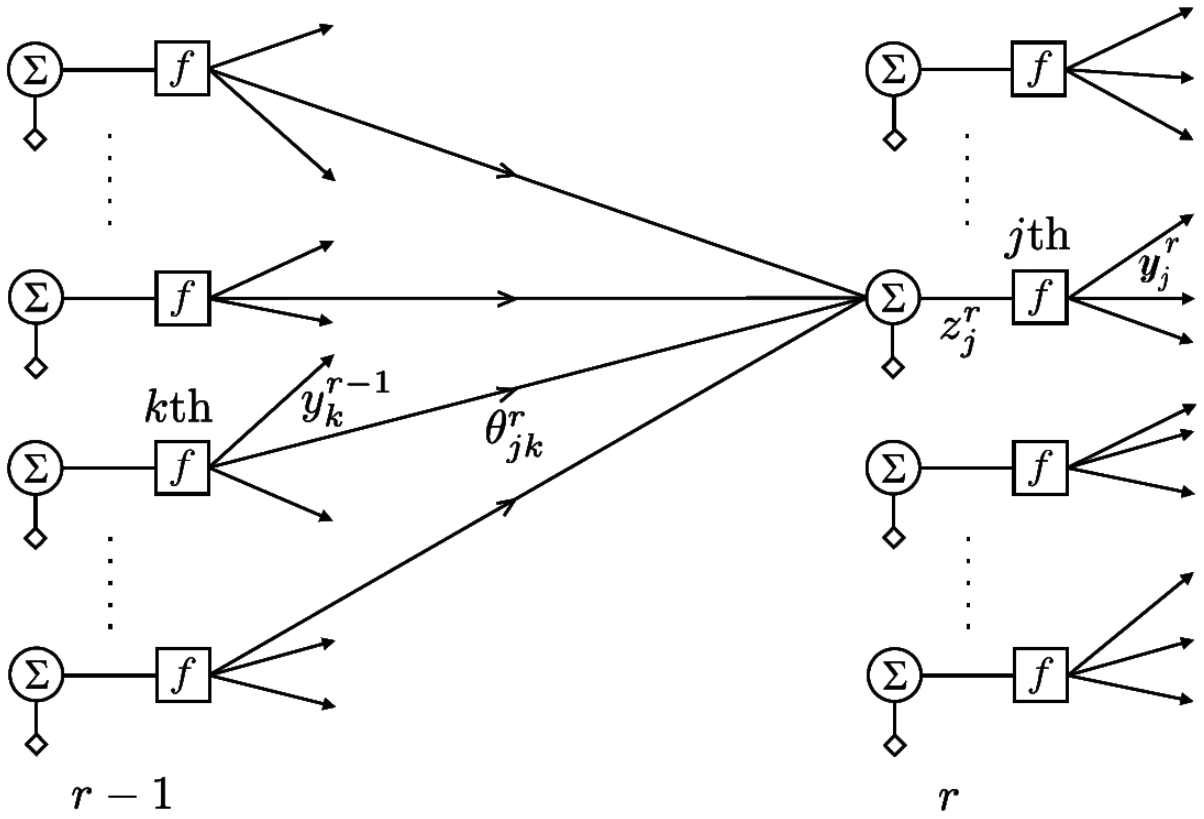
Error Back Propagation Algorithm - EBP

- Example:
- $L = 4$
- $k_0 = l = 5$
- $k_1 = 8$
- $k_2 = 8$
- $k_3 = 6$
- $k_4 = 4$



Error Back Propagation Algorithm - EBP

- Illustration:



Error Back Propagation Algorithm - EBP

- The basic update equation using gradient descent algorithm:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \epsilon^{(t)} \left. \frac{\partial Loss(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right|_{\boldsymbol{\theta}=\boldsymbol{\theta}^{(t)}}$$

- 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:

$$\frac{\partial Loss(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_j^r} = \frac{\partial Loss(\boldsymbol{\theta})}{\partial z_{nj}^r} \frac{\partial z_{nj}^r}{\partial \boldsymbol{\theta}_j^r} = \frac{\partial Loss}{\partial z_{nj}^r} \boldsymbol{y}_n^{r-1}$$

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

- Let define:

$$\delta_{nj}^r \triangleq \frac{\partial Loss(\boldsymbol{\theta})}{\partial z_{nj}^r}$$

Error Back Propagation Algorithm - EBP

- How to compute δ_{nj}^{r} (the backbone of EBP algorithm!)
- Start from last layer (r=L) and proceeds backwards toward first hidden layer (r=*1*)
- This philosophy justifies the name given to the algorithm (EBP).
- $r = L$ (assume simple SGD, sample-mode):

$$\frac{\partial Loss(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_j^L} = \frac{\partial Loss(\boldsymbol{\theta})}{\partial z_{nj}^L}\frac{\partial z_{nj}^L}{\partial \boldsymbol{\theta}_j^r} = \frac{\partial Loss(\boldsymbol{\theta})}{\partial z_{nj}^L}\boldsymbol{y}_n^{r-1}$$

$$\delta_{nj}^L = \frac{\partial Loss(\boldsymbol{\theta})}{\partial z_{nj}^L} = \frac{\partial}{\partial z_{nj}^L}\sum_{k=1}^{k_L} L\left(y_{nk}, f\left(z_{nk}^L\right)\right) = f'\left(z_{nj}^L\right)L'\left(y_{nj}, f\left(z_{nj}^L\right)\right)$$

- Easy to compute.

Error Back Propagation Algorithm - EBP

- $r < L$ (hidden/input layer) :
- Using chain rule:

$$\delta_{nj}^{r-1} = \frac{\partial Loss(\boldsymbol{\theta})}{\partial z_{nj}^{r-1}} \overset{Chain\ Rule}{=} \sum_{k=1}^{k_r} \frac{\partial Loss}{\partial z_{nk}^{r}} \frac{\partial z_{nk}^{r}}{\partial z_{nj}^{r-1}} = \sum_{k=1}^{k_r} \delta_{nk}^{r} \frac{\partial z_{nk}^{r}}{\partial z_{nj}^{r-1}}$$

$$\frac{\partial z_{nk}^{r}}{\partial z_{nj}^{r-1}} = \frac{\partial \left(\left(\boldsymbol{\theta}_j^r\right)^T \boldsymbol{y}_n^{r-1} \right)}{\partial z_{nj}^{r-1}} = \frac{\partial \left(\sum_{m=0}^{k_{r-1}} \theta_{km}^r y_{nm}^{r-1} \right)}{\partial z_{nj}^{r-1}} = \theta_{kj}^r f'\left(z_{nj}^{r-1}\right), y_{nm}^{r-1} = f(z_{nm}^{r-1})$$

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

Error Back Propagation Algorithm - EBP

- $r < L$ (hidden/input layer) :

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

- Write this recursion for two successive layer:

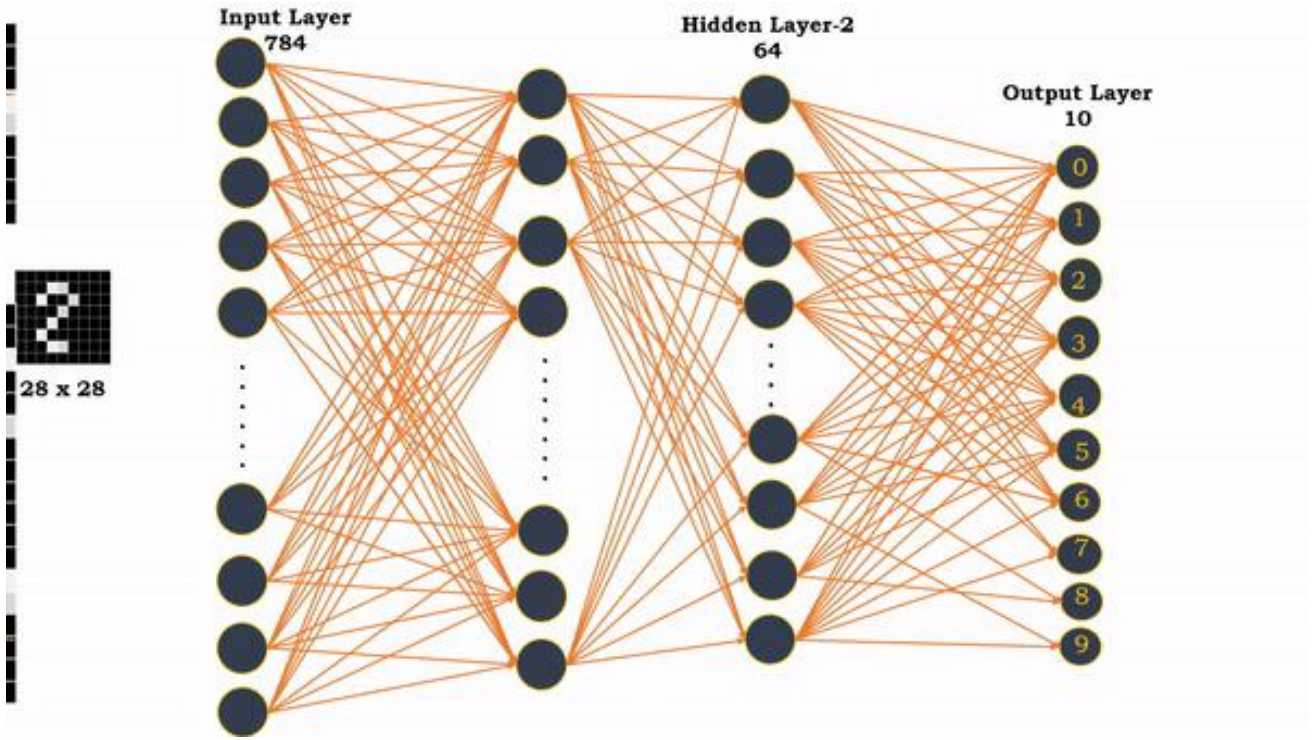
$$\delta_{nj}^{r-2} = \left(\sum_{s=1}^{k_{r-1}} \delta_{ns}^{r-1} \theta_{sj}^{r-1}\right) f'\left(z_{nj}^{r-2}\right),\ j = 1,2,\dots,k_{r-2} \,|\, r \leftarrow r-1, s \leftarrow k$$

$$\delta_{nj}^{r-2} = \left(\sum_{s=1}^{k_{r-1}} \left(\left(\sum_{k=1}^{k_r} \delta_{nk}^{r} \theta_{ks}^{r}\right) f'(z_{ns}^{r-1})\right) \theta_{sj}^{r-1}\right) f'\left(z_{nj}^{r-2}\right),\ j = 1,2,\dots,k_{r-2}$$

- The derivatives of the nonlinearities, $f'(\cdot)$, as well as the *weights* are *multiplied* and the number of the involved products grows:
  - Gradient Vanishing and Exploding!

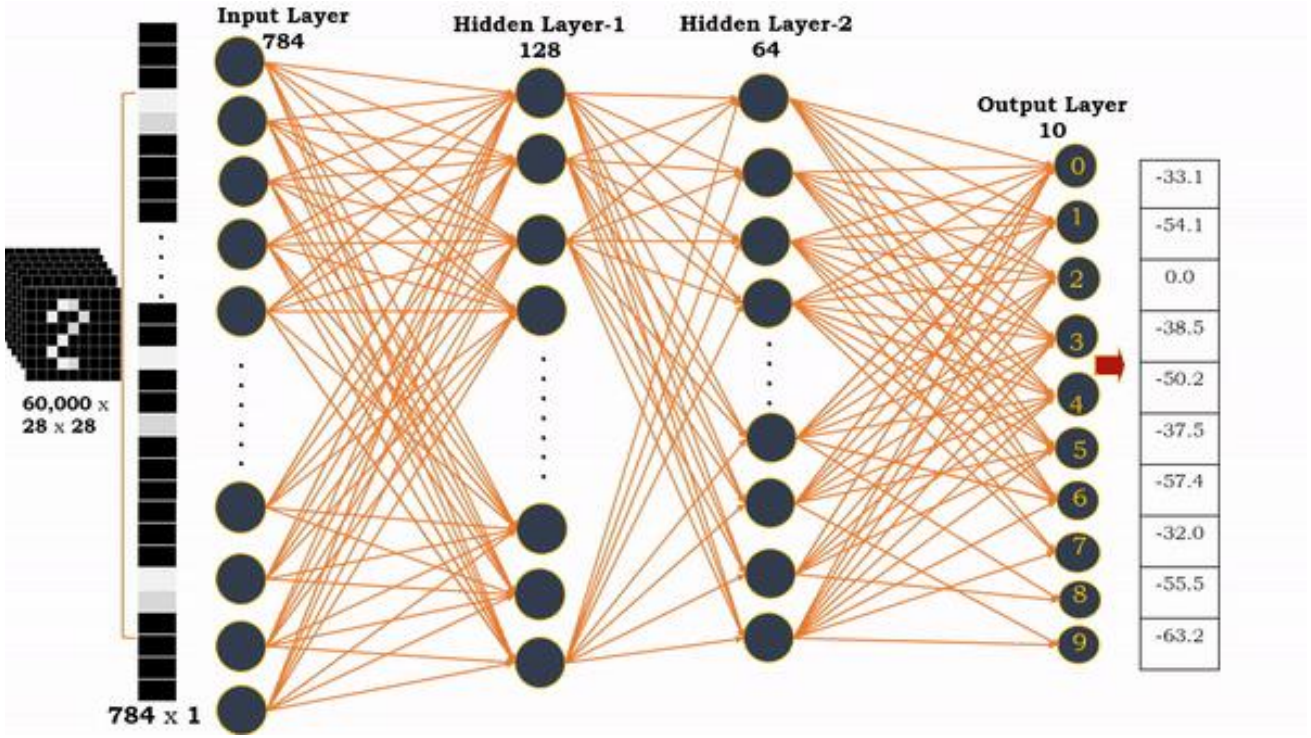
Error Back Propagation Algorithm - EBP

- Forward Calculation:



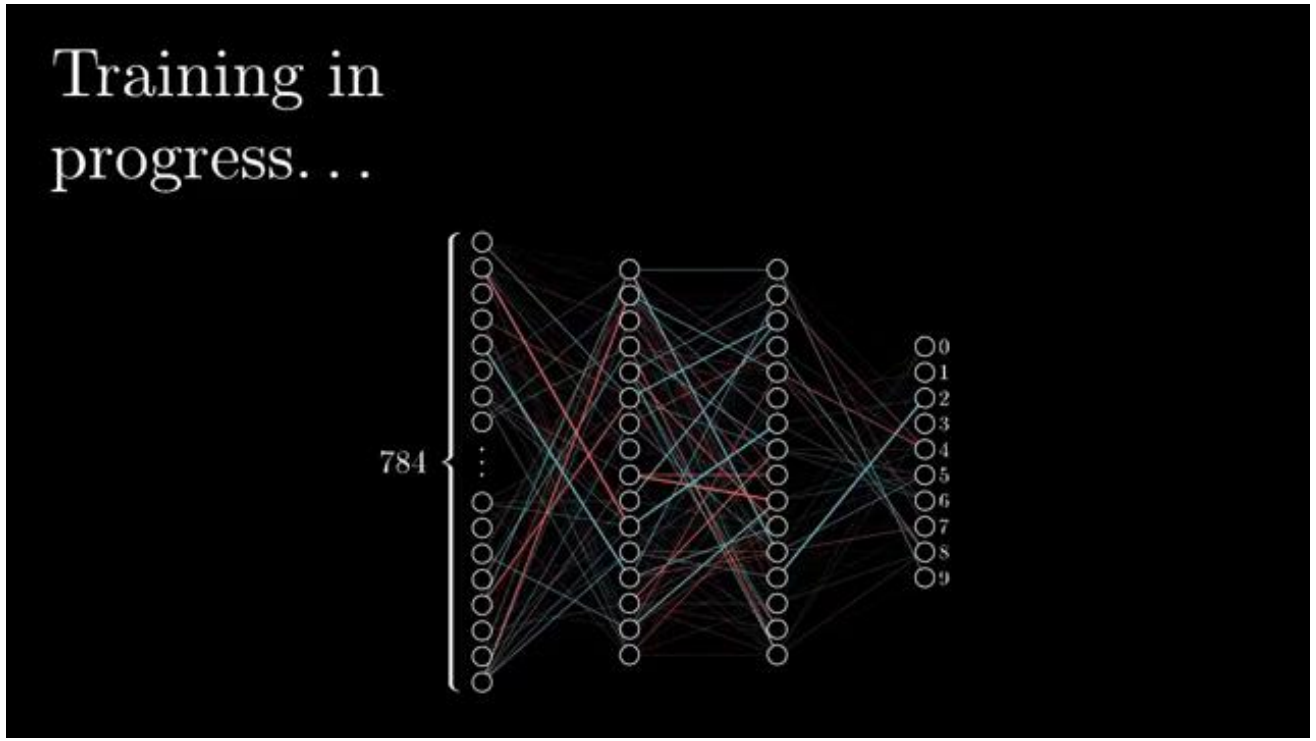
Error Back Propagation Algorithm - EBP

- Backward Update:



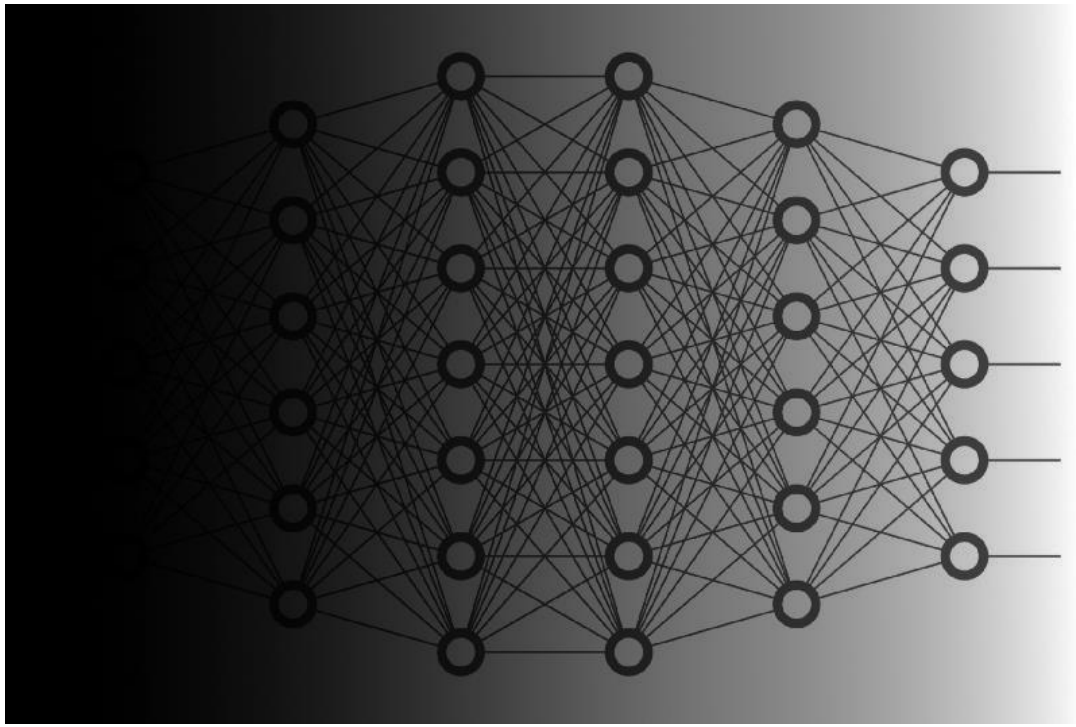
Error Back Propagation Algorithm - EBP

- Optimization:



Error Back Propagation Algorithm - EBP

- Gradient Vanishing:



Heuristics for Making the EBP Algorithm Perform Better

- Stochastic versus batch update:
  - Sample/pattern mode or minibatch 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) in hidden layer, Note: $tanh\big(Utanh(V\boldsymbol{x})\big) \approx UV\boldsymbol{x}$

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:

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

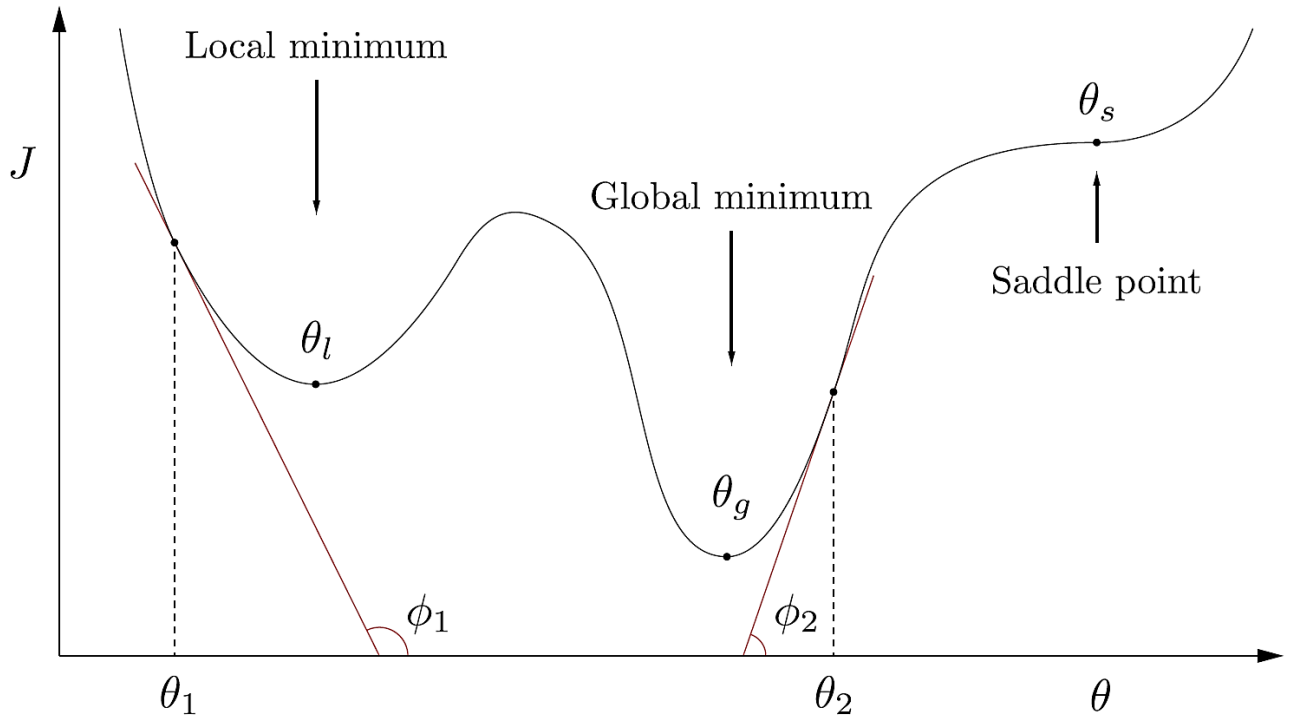
- Input Normalization:
  - Un-correlate input with zero mean and equal variance (using PCA) over the entire training sample.

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 $ReLu$

Heuristics for Making the EBP Algorithm Perform Better

- Local minima and saddle points challenge:



Heuristics for Making the EBP Algorithm Perform Better

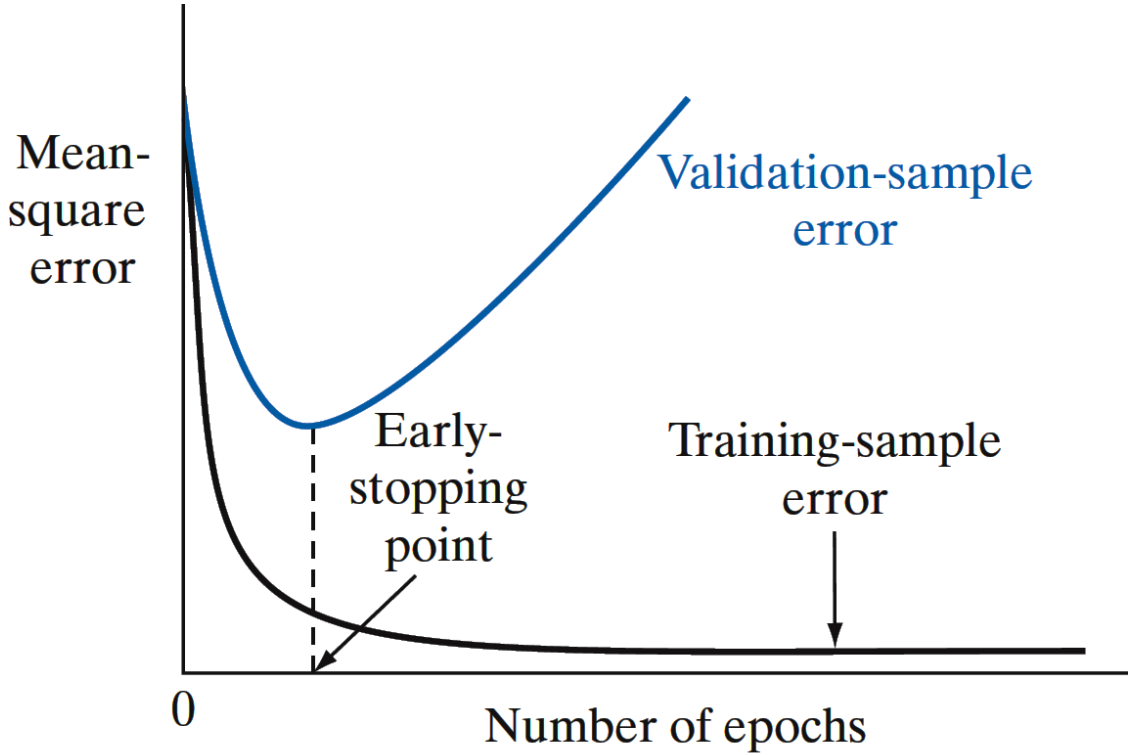
- Local minima and saddle points challenge:
  - Re-Initialization:
    - Re-train using different random initial weights to avoid local minima and saddle point.
  - Noise Injection:
    A. Add noise to input,
    B. Add noise to the parameters as they are being estimated during training,
    C. Add noise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

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

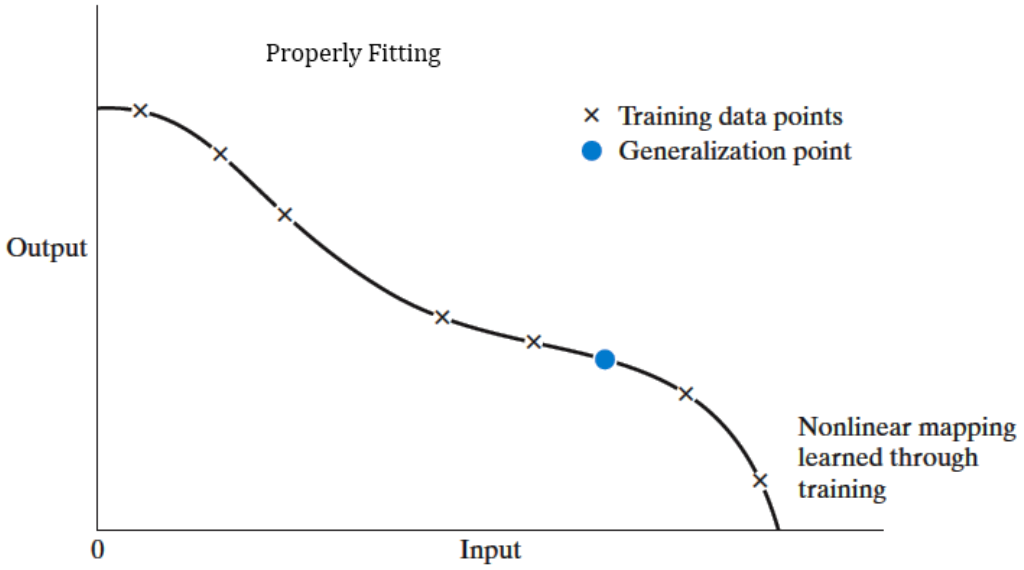
Heuristics for Making the EBP Algorithm Perform Better

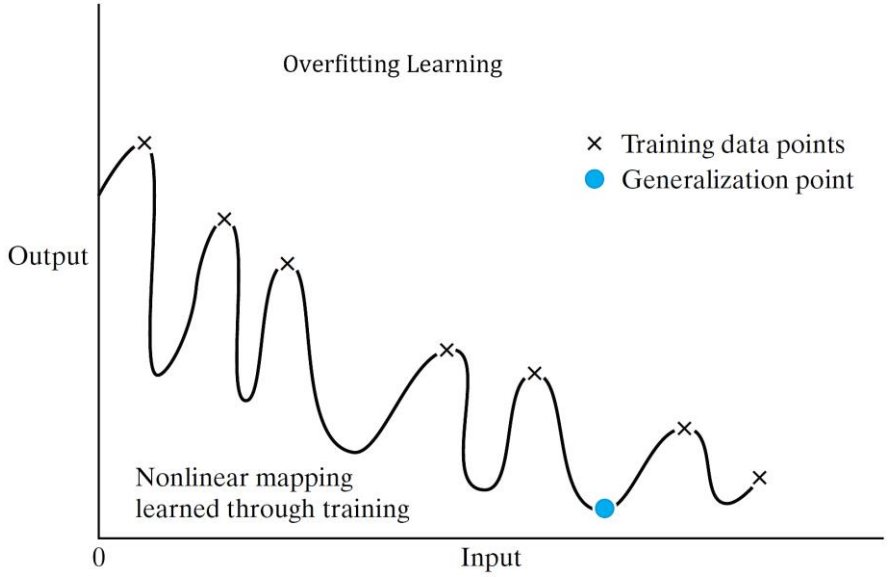
- Early-Stopping:



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.

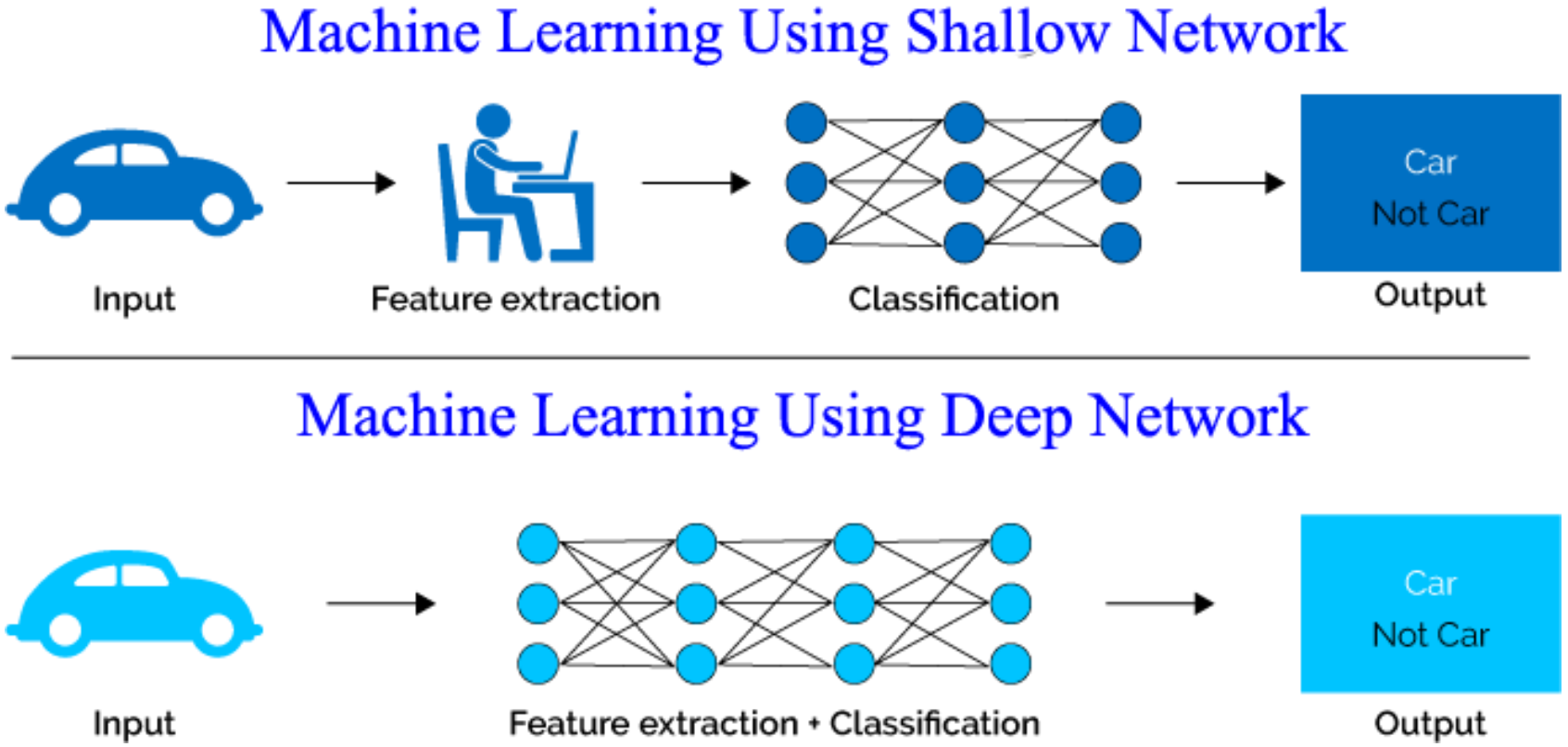
Deep Structure

Contents

- Why Deep Not Shallow
- Why Deep Not Wide
- On DNN Structure Optimization
- On DNN Generalization Power

Why «Deep» Not «Shallow» or Why «Deep» Not «Deep»

- Deep architecture vs Shallow Architecture:



Why «Deep» Not «Shallow» or Why «Deep» Not «Deep»

- 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.

- References:
  - 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 «Deep»

- 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.

- Reference:
  - 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 «Deep»

- 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.
- Reference:
  - 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 «Deep»

- 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.
- Reference:
  - 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 «Deep»

- 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.
- Reference:
  - 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
- Reference:
  - 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.

- Reference:
  - 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.

- Reference:
  - 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.
- Reference:
  - 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!

Any Question