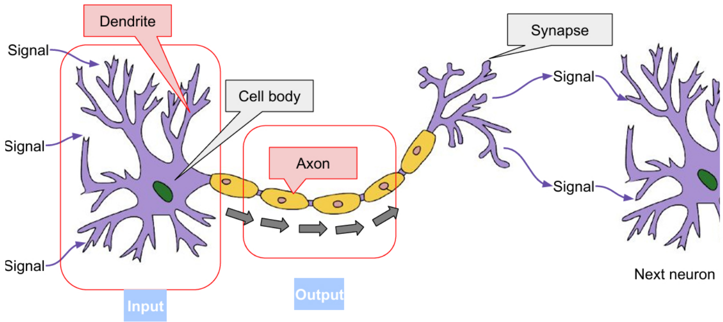
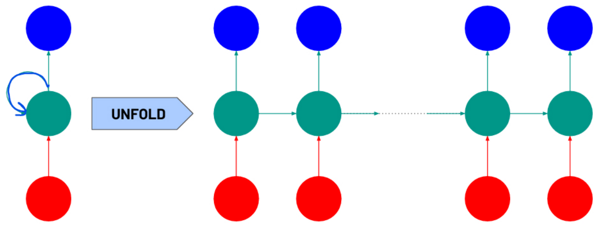
Biological neuron

McCulloch-Pitts (MCP) neuron, in 1943
McCulloch and Pitts described a nerve cell as a simple logic gate with binary outputs;

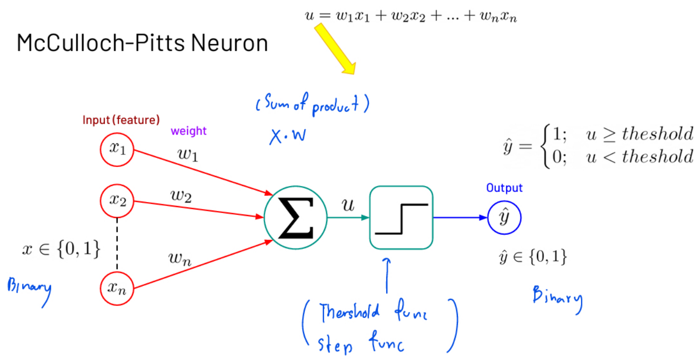
- multiple signals arrive at the dendrites,
- they are then integrated into the cell body, and,
- if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.



Recurrent Neural Network



McCulloch-Pitts Neuron



- โมเดล neural ตัวแรก
- มาจาก Warren McCulloch (neuro-sci)
  Walter Pitts (logician)
- เลียนแบบการทำงานของเซลล์ประสาท [illegible] ไฟเปิด-ปิด
- มี Input / output เป็น 0 กับ 1

วิธีใช้

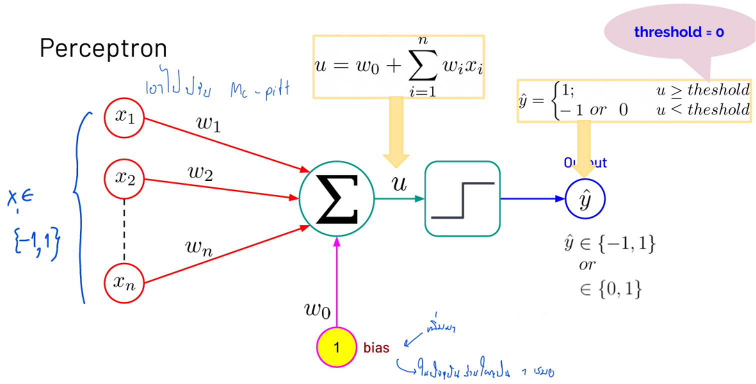
1 เลือก logic gate มา 1 ตัว
2 กำหนดค่า weight เอง (สุ่ม)
3 คำนวณค่า Σ แต่ละ instance
4 ส่งค่าไปผ่าน threshold func

* ข้อจำกัด *

ต้อง set weight & threshold เอง } model จะไม่มี [illegible]
set ไม่ได้ [illegible]

Linearly separable
เส้นตรงนั้น Mac-pitt ต้อง tune weight, threshold
$f(x,y) = -y + mx + c$

Perceptron



- Frank Rosenblatt based on McCulloch-Pitts
  ↳ ยัง Model เดิม
- weight จะถูกปรับ [illegible] ทำให้ [illegible] output [illegible] (convergence theory)
- มี learning rate (η-eta) $\eta \in (0,1]$

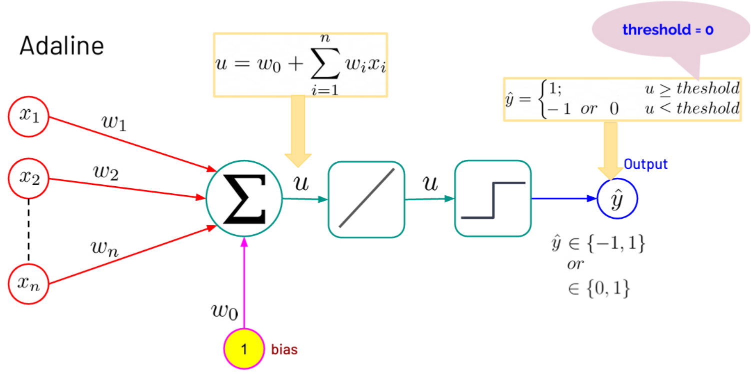
$[w_{ใหม่} = w_{เก่า} + \eta \cdot (y - \hat{y}) \cdot x_i]$

วิธีการใช้

1 (สุ่ม) weight เริ่มต้น
กำหนด Learning rate $\eta = (0,1]$
กำหนดรอบ ทำซ้ำ

2 สำหรับ แต่ละ instance
คำนวณ u แล้วส่งไปผ่าน Threshold $y \in \{-1,1\}$
อัปเดต weights

Adaline

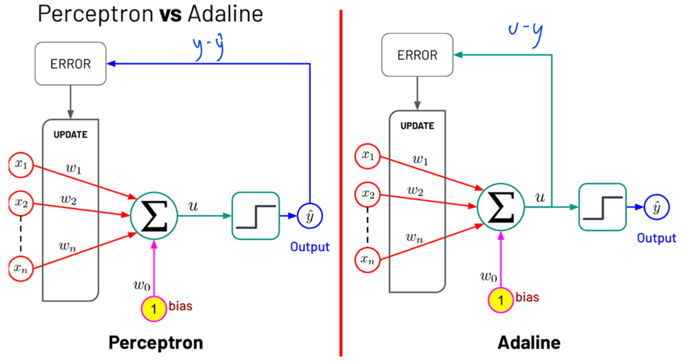


- มาจาก Bernard Widrow & Ted Hoff
- Based on McCulloch-Pitts neuron
  โดย output จะผ่าน linear activation func ($f(x) = x$) ก่อนไปยัง step func
- ต่างจาก Perceptron คือ
  step func ใช้สำหรับ predict ในขั้นตอนสุดท้าย

วิธีใช้ เหมือน Perceptron

ตอนขั้นตอน ... u แล้ว ไม่ต้องนำไปใช้ [illegible] [illegible] threshold func
ปรับ weight
เมื่อ [illegible] ผ่าน threshold func

Perceptron vs Adaline



Behind the Adaline Learning

Function Approximation

- เป็น func ที่ใช้ในการหาค่า weight ที่สามารถลด error ได้น้อยที่สุด
- target function
  $y = \hat{y} = f(x, w)$ x : data, w : weight
- $J = \| y - f(x, w) \|$ ยิ่งน้อยยิ่งดี

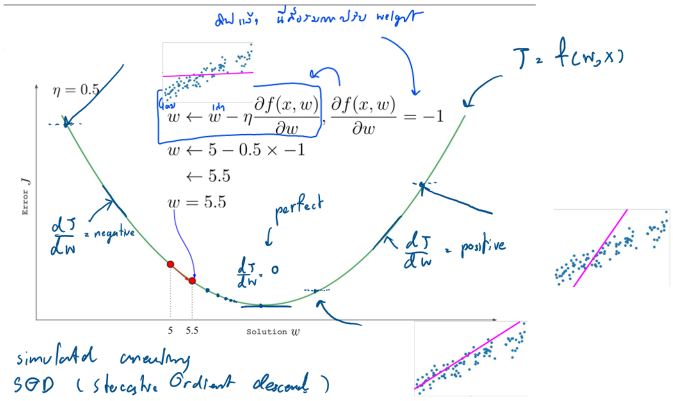
1 Gradient - descent

$w_i \leftarrow w_i - \eta \frac{\partial J_i}{\partial w_i}$ { ความชันเป็นลบ จะเพิ่ม w_i จะความชันเป็นบวก จะลด w_i }

Gradient-descent

1 Gradient - descent

$w_i \leftarrow w_i - \eta \frac{\partial J_i}{\partial w_i}$ { ความชันเป็นลบ จะเพิ่ม w_i จะความชันเป็นบวก จะลด w_i }



2 simulated annealing
3 SGD (Stocastic Gradient descent)

Delta Learning Rule

- ถูกค้นพบโดย Widrow & Hoff (คนเดียวกับ Adaline)
  - ↳ Delta Learning Rule, Widrow & Hoff Learning Rule, Least Mean Square (LMS) algorithm คือตัวเดียวกัน
- เป็นพื้นฐานของ Adaline ในการ train
- ใช้ความต่างของ target (y) และ output ที่ได้จาก activation function ($\hat{y}$) → error
  โดยพยายาม minimize error ให้น้อย ด้วยหลักการ gradient descent เพื่อปรับ weight

  error function, loss function → MSE, RMSE, MAE etc.

  Adaline → ใช้ Delta Learning → ใช้ Gradient descent → derivative of error ---→ ต้องใช้ derivative of activation function

Delta Learning Rule

- Given

$E_j = y_j - \hat{y}_j$ is the **error** for the j-th sample

$\hat{y}_j = \phi\left(w_0 + \sum_{i=1}^{n} w_i x_{j,i}\right)$ is the **output** for the j-th sample

$J_j = E_j^2$ is the **squared error** for the j-th sample

- The learning of delta rule use the ***Gradient Descent Learning*** of E_j

$w_i \leftarrow w_i - \eta \frac{\partial J_j}{\partial w_i}$ ← ↳ เรือก loss function

Loss function เช่น: Square Error

Compute the derivative

$\phi(u) = \frac{1}{1 + e^{-u_j}}$

$\frac{\partial \phi(u_j)}{\partial u_j} = \phi(u_j)\left(1 - \phi(u_j)\right)$

$\frac{\partial J_j}{\partial w_i} = -\left(y_j - \hat{y}_j\right) \frac{\partial \phi(u)}{\partial u} x_{j,i}$

- $w_i \leftarrow w_i + \eta \left(y_j - \hat{y}_j\right) \frac{\partial \phi(u)}{\partial u} x_{j,i}$
- when η is the learning rate which it value is $\eta = (0, 1]$
- $\frac{\partial \phi(u)}{\partial u}$ is **the derivative of activation function** used in the neural model

Weight adjusting

$$w_i \leftarrow w_i - \eta \frac{\partial J_j}{\partial w_i}$$
$$\leftarrow w_i + \eta \left(y_j - \hat{y}_j\right) \frac{\partial \phi(u)}{\partial u} x_{j,i}$$
$$\leftarrow w_i + \eta \left(y_j - \hat{y}_j\right) \phi(u_j)\left(1 - \phi(u_j)\right) x_{j,i}$$

Multi-Layer Perceptron (MLP)

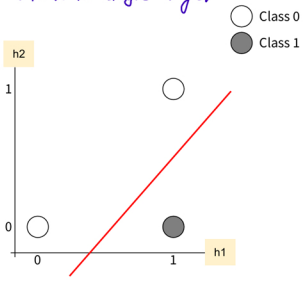
Feed forward

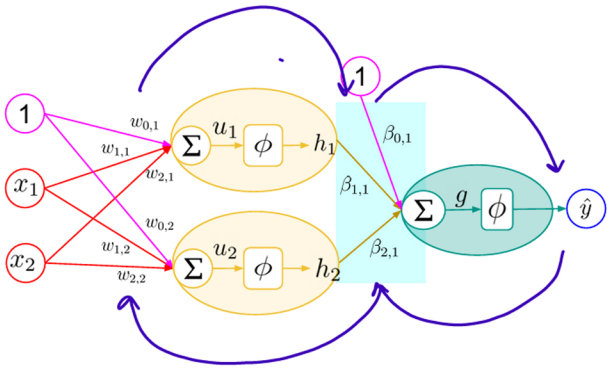
Map the original space (x_1, x_2) to the new space (h_1, h_2)

Feature mapping ทำให้ข้อมูล stayle layer

XOR

x_1	x_2	h_1	h_2	y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0





Back propagation

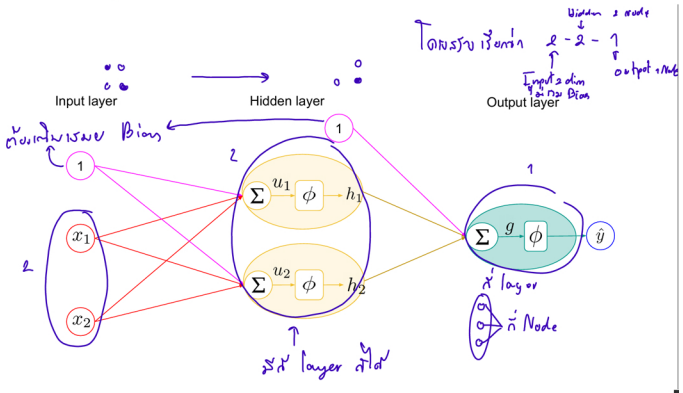
1 โครงข่ายประสาท 2 - 2 - 1

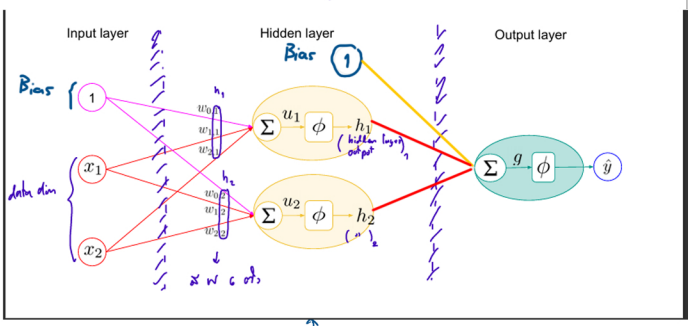
คือ Node Hidden layer

2 - 2 - 1

Input: 2 dimension ไม่รวม Bias

Output 1 Node





→ Feed-Forward (compute output)

← Backpropagation (weight tuning) เอา Error ย้อนกลับจาก output → ไปหาต้น → layer แรก

Recall:

$$E(t) = \frac{1}{2}e^2(t)$$

$$e = y - \hat{y}$$

$$\hat{y} = \phi(g)$$

$$g = \beta_{0,1} + \sum_{k=1}^{l} \beta_{k,1} h_k$$

$$u_1 = w_{0,1} + \sum_{j=1}^{n} w_{j,1} x_j$$

$$u_2 = w_{0,2} + \sum_{j=1}^{n} w_{j,2} x_j$$

$$h_1 = \phi(u_1)$$

$$h_2 = \phi(u_2)$$

$$\beta = \beta + \eta e(\hat{y}(1-\hat{y}))h$$

$$w = w + \eta e(\hat{y}(1-\hat{y}))\beta(h(1-h))x$$

Step 2: Train model

- For each data point (x)

train

- **Feed-forward**
  - **Step 2.1:** Compute outputs of hidden layer
  - **Step 2.2:** Compute outputs of output layer

predict

$$u_1 = w_{0,1} + \sum_{j=1}^{n} w_{j,1} x_j$$

$$u_2 = w_{0,2} + \sum_{j=1}^{n} w_{j,2} x_j$$

$$h_1 = \phi(u_1)$$

$$h_2 = \phi(u_2)$$

$$g = \beta_{0,1} + \sum_{k=1}^{l} \beta_{k,1} h_k$$

Activation function: Sigmoid

$$\frac{1}{1+e^{-u(t)}}$$

$$\hat{y} = \phi(g)$$

- **Backpropagation**
  - **Step 2.3:** Adjust the weights of output layer — $\beta = \beta - \eta \nabla_\beta E$
  - **Step 2.4:** Adjust the weights of input (hidden) layer — $w = w - \eta \nabla_w E$

Step 1 : ตั้งค่า Parameter

- กำหนด Network architecture
  - จำนวน hidden layer
  - จำนวน hidden node
- ตั้ง Weight เริ่มต้น
- กำหนด activation function
- กำหนด learning rate
- จำนวน epoch

Step 2 :

Feed - Forward

2.1 คำนวณ output of hidden layer (h_1, h_2)

2.2 คำนวณ output of output layer (y_hat)

Back propagation

2.3 ปรับ weight output layer

2.4 ปรับ weight input layer

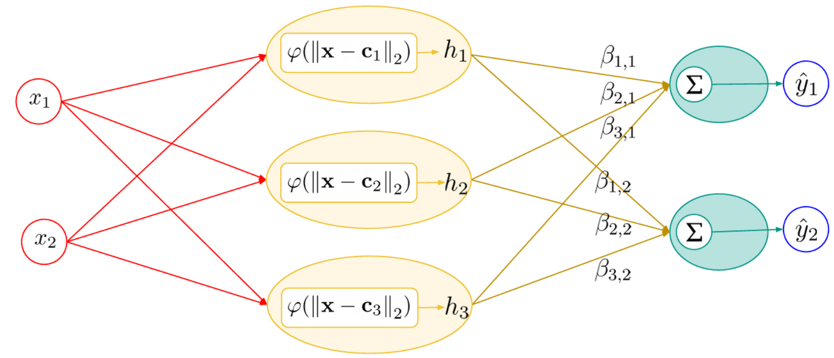
Radial Basis Function (RBF) Neural Network

RBF มีเพียง hidden layer 1 layer
↳ ใช้แนวทาง จากการวัดระยะทาง

- มี output layer, 1 Hidden Layer, output Layer
- ไม่มี input weight
- activation function ใช้ Radial-Basis function
- จำนวนโหนด Node ใน Hidden node จะเก็บค่า "Prototype" หรือ "center" เป็น vector เพื่อหาเป็นค่ากึ่งกลางของ bell shape ของ RBF
- tuning output by { LMS, GD / Generalized pseudo inverse

center คือ จุดศูนย์กลางในข้อมูล
↓
$center_n = C_n$ ประจำ Hidden node ที่ n
↳ user กำหนด, k-mean

ไม่มี φ ที่ output Layer

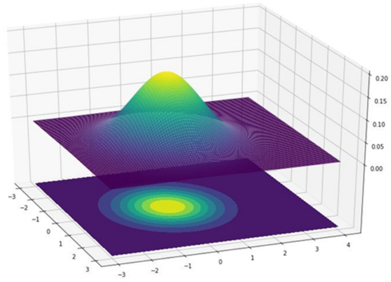


The **Gaussian Functions** are generally used for Radial Basis Function.

Euclidean distance ↓

Given : $r = \|\mathbf{x}-\mathbf{c}\|_2$

$$\varphi(r) = exp(-\frac{r^2}{2\sigma^2})$$



Gaussian RBF:

$$\varphi(\|\mathbf{x}-\mathbf{c}\|_2) = exp(-\frac{\|\mathbf{x}-\mathbf{c}\|_2^2}{\sigma^2})$$

1. Hidden layer

เลือก center vector โดยจะมีจำนวนเท่ากับ Hidden Node
↳ - ถูกสุ่มจาก training set
- ใช้ k-mean clustering ...

2. Output weight tuning

- gradient descent $J = \frac{1}{2}(y-\hat{y})^2$ = square error

$$\beta_{l,m} \leftarrow \beta_{l,m} - \eta \frac{\partial J_i}{\partial \beta_{l,m}}$$
$$\leftarrow \beta_{l,m} + \eta\,(y_{i,m} - \hat{y}_{i,m})\,h_{i,l}$$

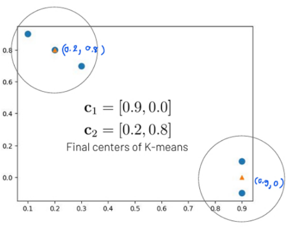
- generalized pseudo inverse

$$\boldsymbol{\beta} = (\mathbf{H}^\top\mathbf{H})^{-1}\mathbf{H}^\top\mathbf{Y}$$

k-mean →

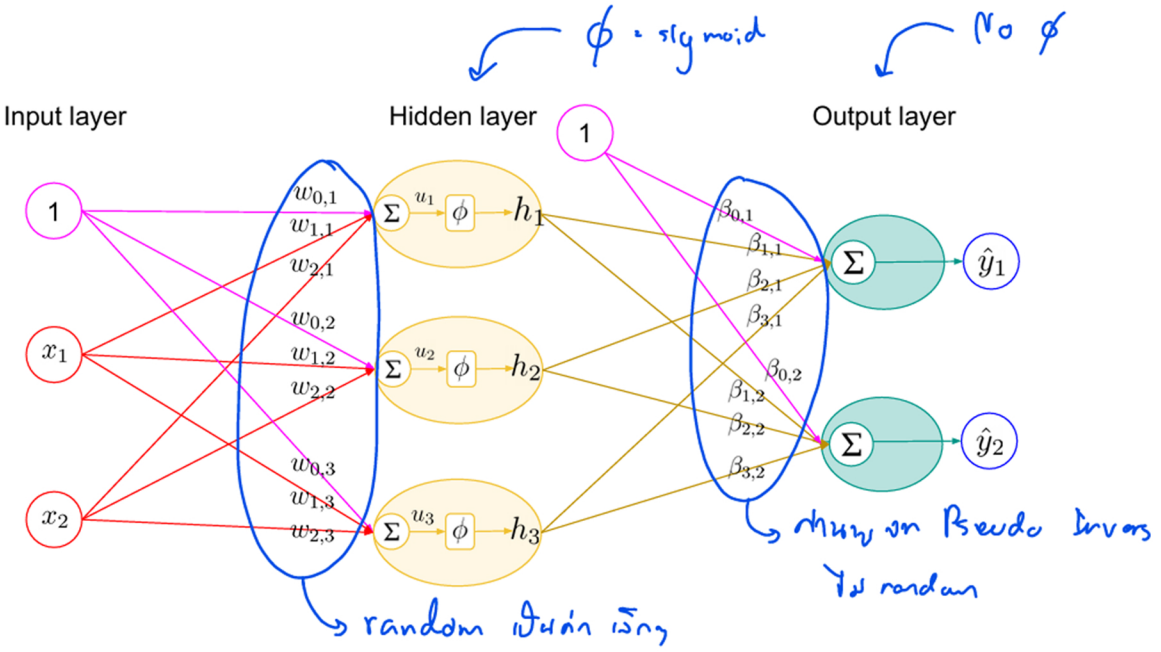
Example

x_1	x_2	y
0.1	0.9	-1
0.2	0.8	-1
0.3	0.7	-1
0.9	0.1	1
0.9	-0.1	1



Random Neural Network

- 1 hidden layer ไม่มี Backprob แต่ทำ Pseudo Inverse แทน
- สุ่ม input weight
- Out weight ได้จาก Pseudo inverse
- Fast learning model (ไม่สิ้นเปลือง)



---

data

X = bias + data

$$N \begin{bmatrix} 1 & x & \cdots & x \\ 1 & x & \cdots & x \\ 1 & x & \cdots & x \end{bmatrix}_{d+1}$$

$N \times (d+1)$

Single Hidden Node = L

W

$$d+1 \begin{bmatrix} 1 & 2 & \cdots & L \\ w & w & \cdots & w \\ w & w & \cdots & w \\ w & w & \cdots & w \end{bmatrix}$$

$(d+1) \times L$

$H = \phi(X \cdot W)$ =

$$N \begin{bmatrix} 1 & h & h & \cdots & h \\ 1 & h & h & \cdots & h \\ 1 & h & h & \cdots & h \end{bmatrix}_{L+1}$$

$N \times (L+1)$

Output Weight

Output Node = m

$$L+1 \begin{bmatrix} \beta & \beta & \cdots & \beta \\ \beta & \beta & \cdots & \beta \\ \beta & \beta & \cdots & \beta \\ \beta & \beta & \cdots & \beta \end{bmatrix}$$

$(L+1) \times m$

$\hat{Y} = H \cdot \beta$

$$N \begin{bmatrix} y & y & y \\ y & y & y \\ y & y & y \\ y & y & y \end{bmatrix}_{m}$$

$N \times m$

β

$$\mathbf{H}\beta = \mathbf{Y}$$

$$\mathbf{H}^\top\mathbf{H}\beta = \mathbf{H}^\top\mathbf{Y}$$

$$(\mathbf{H}^\top\mathbf{H})^{-1}\mathbf{H}^\top\mathbf{H}\beta = (\mathbf{H}^\top\mathbf{H})^{-1}\mathbf{H}^\top\mathbf{Y}$$

$$\mathbf{I}\beta = (\mathbf{H}^\top\mathbf{H})^{-1}\mathbf{H}^\top\mathbf{Y}$$

numpy.linalg.pinv( ) $\beta = (\mathbf{H}^\top\mathbf{H})^{-1}\mathbf{H}^\top\mathbf{Y}$ ←

Deep Learning Timeline