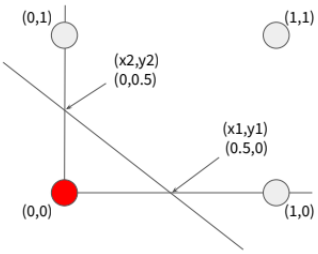
Contents

Contents

McCulloch-Pitts Neuron

- The McCulloch-Pitts model was proposed by **Warren MuCulloch** (neuroscientist) and **Walter Pitts** (logician) in 1943.
- McCulloch and Pitts modeled computationally able to **emulate the behavior of a few boolean functions or *logical gates***, like the **AND gate** and the **OR gate**.
- **Neurons can be seen as biological computational devices**, in the sense that they can ***receive inputs***, ***apply calculations over those inputs algorithmically***, *and then* ***produce outputs***.
- The McCulloch-Pitts model is **the first computational model** of a neuron and it was an extremely simple artificial neuron. ***The inputs could be either a zero or a one. And the output was a zero or a one.***

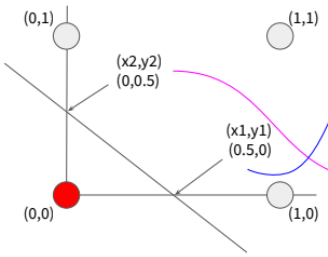
Linearly separable



$$y = mx + c$$

$$\begin{aligned} m &= \frac{y_2 - y_1}{x_2 - x_1} \\ &= \frac{0.5 - 0}{0 - 0.5} \\ &= -1 \end{aligned}$$

Linearly separable

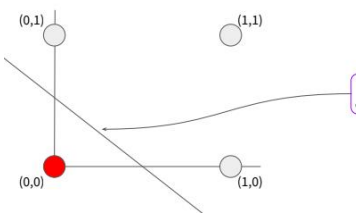


$$y = mx + c$$

$$\begin{aligned} c &= y_1 - mx_1 \\ &= 0 - (-1) \times 0.5 \\ &= 0.5 \end{aligned}$$

$$\begin{aligned} c &= y_2 - mx_2 \\ &= 0.5 - (-1) \times 0 \\ &= 0.5 \end{aligned}$$

Linearly separable

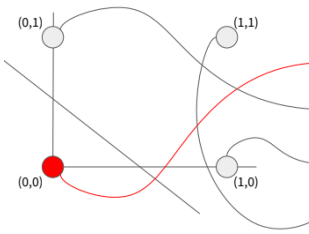


$$y = mx + c$$

$$y = (-1)x + 0.5$$

$$f(x, y) = -x - y + 0.5$$

Linearly separable



$$f(x, y) = -x - y + 0.5$$

$$f(0,0) = +0.5 \quad \text{POSITIVE}$$

$$f(0,1) = -0.5$$

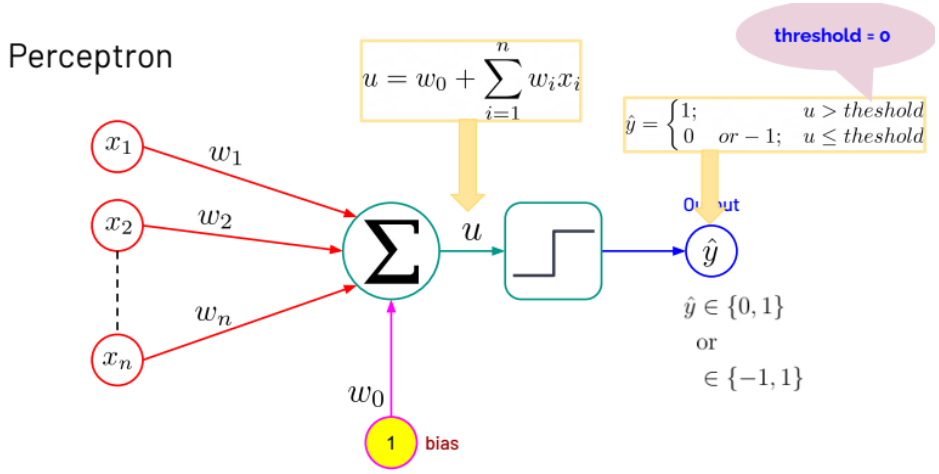
$$f(1,0) = -0.5 \quad \text{NEGATIVE}$$

$$f(1,1) = -1.5$$

Perceptron Learning

Learning rule

- Neural networks learn by mimicking the human brain's learning process, which is capable of ***adapting and changing behavior*** *in response to environmental stimuli*.
- Neural networks learning rules can be defined as the algorithm called ***Learning Algorithm***.



Perceptron Learning Rule

- **Frank Rosenblatt** published the first concept of the **perceptron learning rule** based on the McCulloch-Pitts neuron model in 1957
- In the perceptron learning rule, **weight adjustments** are made through iterative operations.
- According to ***convergence theory***, every round, **the weights of the perceptron must be adjusted to the proper weights;** that is, **the adjusted weights should produce an output that is as close to the actual value as possible.**

Perceptron Learning Rule

- In Perceptron Learning, when the **learning rate** η is defined, the weights are adjusted as

$$w_i(t+1) = w_i(t) + \eta(y - \hat{y})x_i$$

- where $0 < \eta \leq 1$

- According to **convergence theory**, **if the input vector and target values *can be linearly separated***, then when using the perceptron learning algorithm, **the weights should be obtained within a finite number of rounds**.

Perceptron Learning Algorithm

Step 1:

- Initially, **random** the weights with small value
- Define the value of **learning rate** $\eta = (0, 1]$
- Define the stopping criteria i.e. *number of round*

Step 2:

- Check the stopping criteria
  - If meet the criteria, then stop
  - If far from the criteria, go to **step 3**

Perceptron Learning Algorithm

Step 3: Train model

- For each data point (**x**)
  - **Step 3.1:** Calculate **sum-of-product** between input and weight $u = w_0 + \sum_{i=1}^{n} w_i x_i$
  - **Step 3.2:** Calculate the **output** of model $\hat{y} = \begin{cases} 1; & u > theshold \\ 0 \quad or - 1; & u \leq theshold \end{cases}$
  - **Step 3.3: Update Weights** $w_i(t+1) = w_i(t) + \eta(y - \hat{y})x_i$

Step 4:

- Go to **step 2**

AND Gate via Perceptron

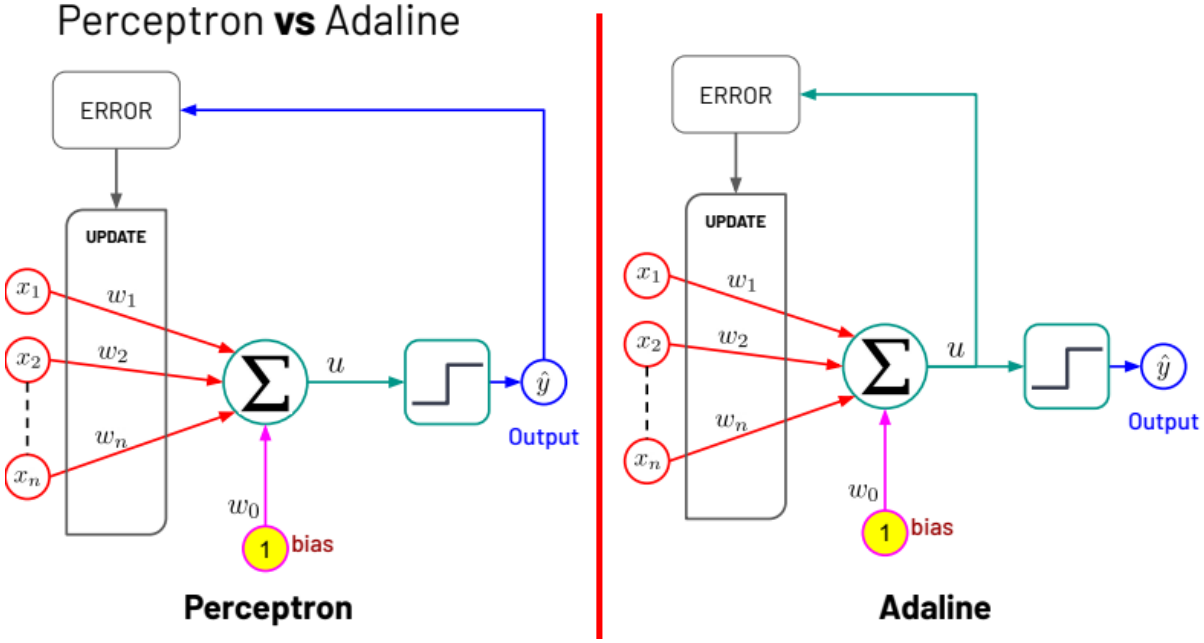
Round 1(Train model)

			Step 3.1	Step 3.2	Step 3.3		
					$w_i(t+1) = w_i(t) + \eta(y - \hat{y})x_i$		
x_1	x_2	y	$u = w_0 + \sum_{i=1}^{n} w_i x_i$	$\hat{y} = \begin{cases} 1; & u > 0 \\ -1; & u \leq 0 \end{cases}$	w_0	w_1	w_2
1	1	1	**0** + (**0***1) + (**0***1) = 0	-1	**0**+ 1*(1-(-1))*1 = **2**	**0**+ 1*(1-(-1))*1 = **2**	**0**+ 1*(1-(-1))*1 = **2**
1	-1	-1	**2** + (**2***1) + (**2***-1) = 2	1	**2**+ 1*(-1-1)*1 = **0**	**2**+ 1*(-1-1)*1 = **0**	**2**+ 1*(-1-1)*-1 = **4**
-1	1	-1	**0** + (**0***-1) + (**4***1) = 4	1	**0**+ 1*(-1-1)*1=**-2**	**0**+ 1*(-1-1)*-1 = **2**	**4**+ 1*(-1-1)*1 = **2**

Adaline

- Adaline was proposed by **Bernard Widrow** and **Ted Hoff** in 1960.
- Adaline was developed based on the McCulloch-Pitts neuron.
- In the Adaline learning, the sum of product (wx) is passed to the *linear activation function* $\sigma(u)$ that is then used to make a prediction using a step function (threshold) as with the perceptron.
- A *key* difference with the perceptron is that
  - The *linear activation function* is used for learning the weights, whilst
  - The step function (threshold) is only used for making the prediction at the end.
  - The linear activation function is *differentiable* whilst the step function is not!

Perceptron **vs** Adaline



Adaline Learning

- In Adaline Learning, when the **learning rate** η is defined, the weights are adjusted as

$$w_i(t+1) = w_i(t) - \eta(2(u-y))x_i$$

- where $0 < \eta \leq 1$

Adaline Learning Algorithm

Step 1:

- Initially, **random** the weights with small value
- Define the value of **learning rate** $\eta = (0, 1]$
- Define the stopping criteria i.e. *number of round*

Step 2:

- Check the stopping criteria
  - If meet the criteria, then stop
  - If far from the criteria, go to **step 3**

Adaline Learning Algorithm

Step 3: Train model

- For each data point ($\mathbf{x}$)
  - **Step 3.1:** Calculate **sum-of-product** between input and weight $u = w_0 + \sum_{i=1}^{n} w_i x_i$
  - **Step 3.2:** Calculate the **output** of model $\hat{y} = \begin{cases} 1; & u > theshold \\ 0 \quad or -1; & u \leq theshold \end{cases}$
  - **Step 3.3: Update Weights** $w_i(t+1) = w_i(t) - \eta(2(u-y))x_i$

Step 4:

- Go to **step 2**

Function Approximation

- The function to approximate the desire (target) value y can be modeled as
  - $y = \hat{y} = f(x, w)$ (target = $\hat{y}$, function = $f(x, w)$)
- This parameters of this function includes data x and weight w
- The objective of this function is **to compute the weight w that can make the minimum error of the function $f(w, x)$**, then the objective function can be represented as
  - $J = \|y - f(x, w)\|$
  - So, the optimum value of w must produce the minimum value for J

Delta Learning Rule

- *Delta Learning Rule* was developed by Widrow and Hoff in the early 1960s.
- Also known as the the *Widrow & Hoff Learning Rule* or the *Least Mean Square (LMS) algorithm*.
- The delta rule is used in Adaline networks for training.
- The delta rule uses the difference between target output and obtained activation to drive learning, which aims to minimize that difference (error) by using the gradient descent to minimize the error from an adaline network's weights.
- The delta rule employs the *error function* for **Gradient Descent learning**, which involves *the modification of weights along the most direct path in weight-space to minimize error*.

Delta Learning Rule

- Given

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

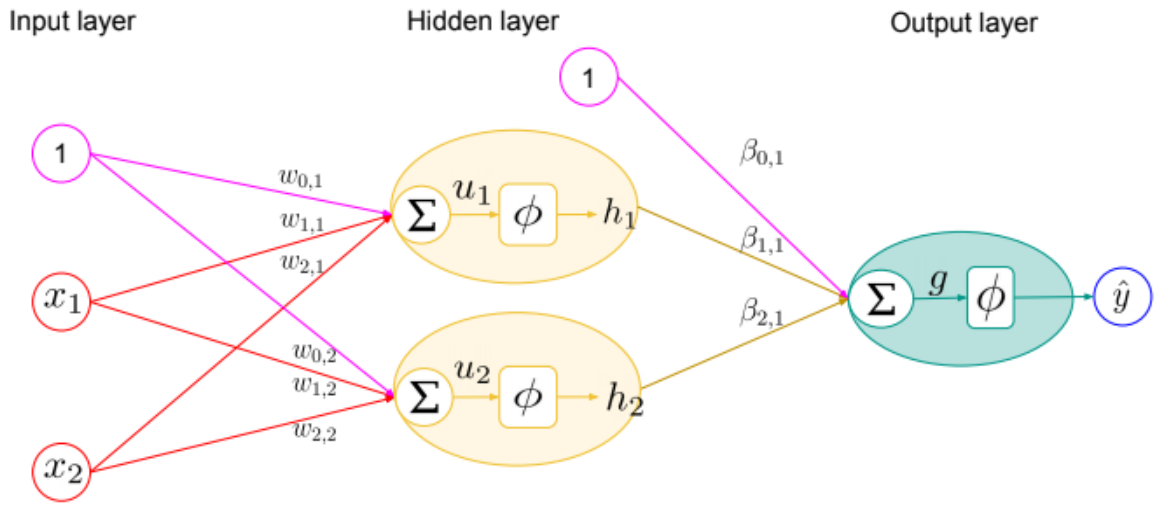
$$\hat{y}_j = \phi\left(w_0 + \sum_{i=1}^{n} w_i x_{j,i}\right)$$ is the **output** for the j-th sample

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

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

$$w_i \leftarrow w_i - \eta \frac{\partial J_j}{\partial w_i}$$

Multi-Layer Perceptron (MLP)



How to train MLP?

Step 1: parameter setting

- Define network architecture
  - number of hidden layers
  - number of hidden nodes
- Set the initial weights
- Define Activation function
- Define the value of learning rate
- Define the stopping criteria
  - (i.e.) number of round

How to train MLP?

Step 2: Train Model by *Backpropagation Learning Algorithm*

- For each data point (**x**)

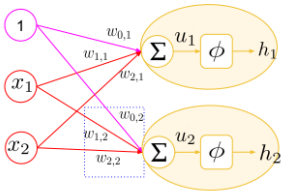
Feed-Forward

  - **Step 2.1:** Compute *outputs of hidden layer (h)*
  - **Step 2.2:** Compute *outputs of output layer (y_hat)*

Backpropagation

  - **Step 2.3:** Adjust the *weights of output layer*
  - **Step 2.4:** Adjust the *weights of input (hidden) layer*

Feed-Forward

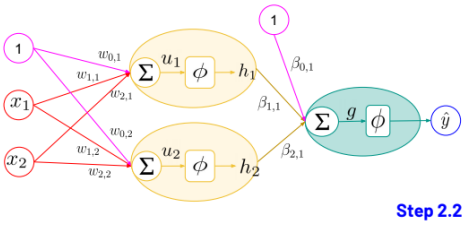


Step 2.1

$$u_1 = w_{0,1} + \sum_{j=1}^{n} w_{j,1} x_j \longrightarrow h_1 = \phi(u_1)$$

$$u_2 = w_{0,2} + \sum_{j=1}^{n} w_{j,2} x_j \longrightarrow h_2 = \phi(u_2)$$

Feed-Forward



Step 2.1

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

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

$$h_1 = \phi(u_1)$$

$$h_2 = \phi(u_2)$$

Step 2.2

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

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

Backpropagation

Weight adjustment

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

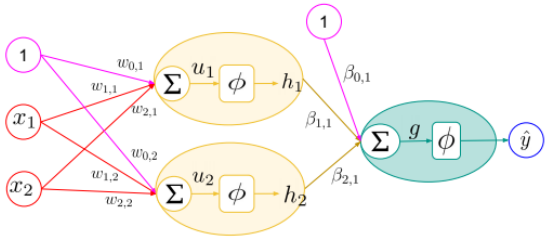
$$w = w + \eta e(\hat{y}(1-\hat{y}))\beta(h(1-h))x$$

Train MLP for XOR

Step 1:

- Define network architecture: 2-2-1
- Set the initial weights
- Define Activation function : "sigmoid" $\frac{1}{1+e^{-u(t)}}$
- Define the value of learning rate : 0.0001
- Define the stopping criteria i.e. *number of round* : 3

Train MLP for XOR



$$W = \begin{bmatrix} w_{0,1}=1 & w_{0,2}=2 \\ w_{1,1}=1 & w_{1,2}=2 \\ w_{2,1}=-2 & w_{2,2}=-1 \end{bmatrix} \qquad \beta = \begin{bmatrix} \beta_{0,1}=2 & \beta_{1,1}=-1 & \beta_{2,1}=2 \end{bmatrix}$$

Train MLP for XOR

Step 2: Train model

- For each data point (**x**)

Feed-forward

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

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

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

$$h_1 = \phi(u_1)$$

$$h_2 = \phi(u_2)$$

Activation function: Sigmoid $\frac{1}{1+e^{-u(t)}}$

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

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

Backpropagation

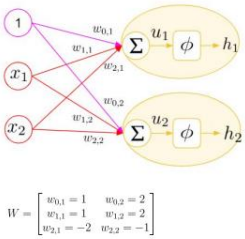
- **Step 2.3:** Adjust the weights of output layer $\beta = \beta - \eta \nabla_{\beta} E$
- **Step 2.4:** Adjust the weights of input (hidden) layer $w = w - \eta \nabla_{w} E$

Feed-Forward

Step 2: Train Model

Round: 1
Learn with data row: 1

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



$$W = \begin{bmatrix} w_{0,1}=1 & w_{0,2}=2 \\ w_{1,1}=1 & w_{1,2}=2 \\ w_{2,1}=-2 & w_{2,2}=-1 \end{bmatrix}$$

Step 2.1: Compute outputs of hidden layer

$$u_1 = w_{0,1} + w_{1,1}x_1 + w_{2,1}x_2$$
$$u_2 = w_{0,2} + w_{1,2}x_1 + w_{2,2}x_2$$

$$u_1 = 1 + (1 \times 0) + (-2 \times 0) = 1$$
$$u_2 = 2 + (2 \times 0) + (-1 \times 0) = 2$$

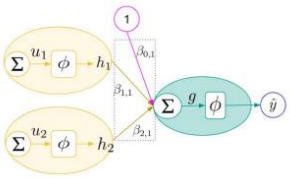
$$h_1 = \frac{1}{1+e^{-u_1}} = \frac{1}{1+e^{-1}} = 0.73$$
$$h_2 = \frac{1}{1+e^{-u_2}} = \frac{1}{1+e^{-2}} = 0.88$$

Feed-Forward

Step 2: Train Model

Round: 1
Learn with data row: 1

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



$$\beta = [\beta_{0,1}=2 \quad \beta_{1,1}=-1 \quad \beta_{2,1}=2]$$

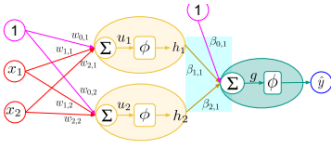
Step 2.2: Compute outputs of output layer

$$g = \beta_{0,1} + \beta_{1,1}h_1 + \beta_{2,1}h_2$$

$$g = 2 + (-1 \times 0.73) + (2 \times 0.88) = 3.03$$

$$\hat{y} = \frac{1}{1+e^{-g}} = \frac{1}{1+e^{-3.03}} = 0.95$$

Backpropagation



Step 2: Train Model

Round: 1
Learn with data row: 1

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Output weights tuning

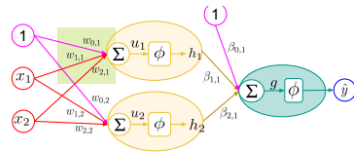
$$\beta = \beta + \eta e(\hat{y}(1-\hat{y}))h$$

$$\beta_{0,1} = \beta_{0,1} + (\eta)(y-\hat{y})(\hat{y}(1-\hat{y}))(1)$$
$$\beta_{0,1} = 2 + (0.0001)(0-0.95)(0.95(1-0.95))(1) \simeq 1.99$$

$$\beta_{1,1} = \beta_{1,1} + (\eta)(y-\hat{y})(\hat{y}(1-\hat{y}))(h_1)$$
$$\beta_{1,1} = -1 + (0.0001)(0-0.95)(0.95(1-0.95))(0.73) \simeq -1$$

$$\beta_{2,1} = \beta_{2,1} + (\eta)(y-\hat{y})(\hat{y}(1-\hat{y}))(h_2)$$
$$\beta_{2,1} = 2 + (0.0001)(0-0.95)(0.95(1-0.95))(0.88) \simeq 1.99$$

Backpropagation



Step 2: Train Model

Round: 1
Learn with data row: 1

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Input weights tuning

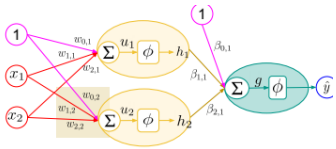
$$w = w + \eta e(\hat{y}(1-\hat{y}))\beta(h(1-h))x$$

$$w_{0,1} = w_{0,1} + \eta(y-\hat{y})(\hat{y}(1-\hat{y}))\beta_{1,1}(h_1(1-h_1))(1)$$
$$w_{0,1} = 1 + (0.0001)(0-0.95)(0.95(1-0.95))(-1)(0.73(1-0.73))(1) \simeq 1$$

$$w_{1,1} = w_{1,1} + \eta(y-\hat{y})(\hat{y}(1-\hat{y}))\beta_{1,1}(h_1(1-h_1))x_1$$
$$w_{1,1} = 1 + (0.0001)(0-0.95)(0.95(1-0.95))(-1)(0.73(1-0.73))0 \simeq 1$$

$$w_{2,1} = w_{2,1} + \eta(y-\hat{y})(\hat{y}(1-\hat{y}))\beta_{1,1}(h_1(1-h_1))x_2$$
$$w_{2,1} = -2 + (0.0001)(0-0.95)(0.95(1-0.95))(-1)(0.73(1-0.73))0 \simeq -2$$

Backpropagation



Step 2: Train Model

Round: 1
Learn with data row: 1

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Input weights tuning

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

$$w_{0,2} = w_{0,2} + \eta(y-\hat{y})(\hat{y}(1-\hat{y}))\beta_{1,2}(h_2(1-h_2))(1)$$
$$w_{0,2} = 2 + (0.0001)(0-0.95)(0.95(1-0.95))(2)(0.88(1-0.88))(1) \simeq 1.99$$

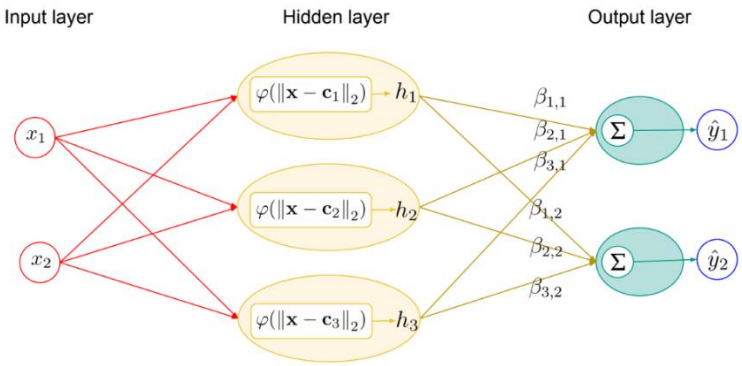
$$w_{1,2} = w_{1,2} + \eta(y-\hat{y})(\hat{y}(1-\hat{y}))\beta_{1,2}(h_2(1-h_2))x_1$$
$$w_{1,2} = 2 + (0.0001)(0-0.95)(0.95(1-0.95))(2)(0.88(1-0.88))0 \simeq 2$$

$$w_{2,2} = w_{2,2} + \eta(y-\hat{y})(\hat{y}(1-\hat{y}))\beta_{1,2}(h_2(1-h_2))x_2$$
$$w_{2,2} = -1 + (0.0001)(0-0.95)(0.95(1-0.95))(2)(0.88(1-0.88))0 \simeq -1$$

RBF Neural Networks

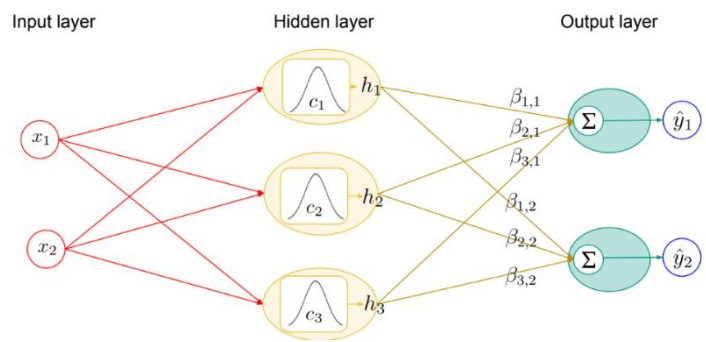
- RBF network is the single hidden layer feedforward neural network which consists of an input layer, a single hidden layer, and an output layer.
- There are no input weights on the lines from the input nodes to the hidden nodes.
- The **Radial-Basis Function** is used as the activation function in hidden layer.
- Each hidden node stores a "prototype" vector which also often called "**center**" vector because it is the value at the center of the bell shaped radial basis function.
- The leaning method for tuning the **output weights** can be:
  - LMS, Gradient descent
  - Generalized pseudo inverse

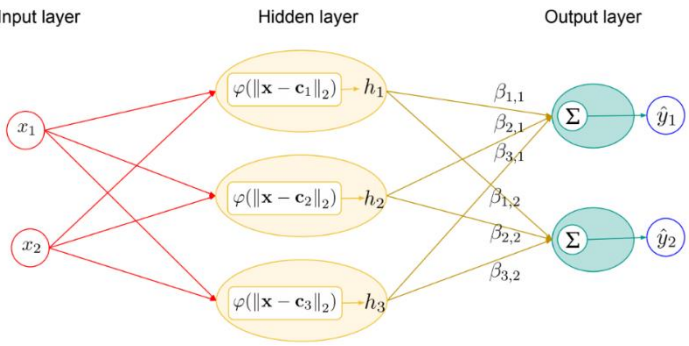
RBF neural network



RBF neural network

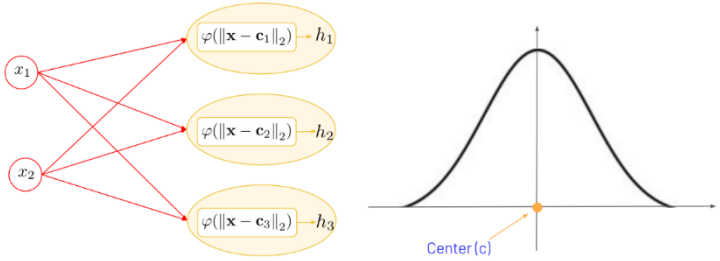
RBF neural network





RBF Activation Function

- The **Radial-Basis Function** is used as the activation function

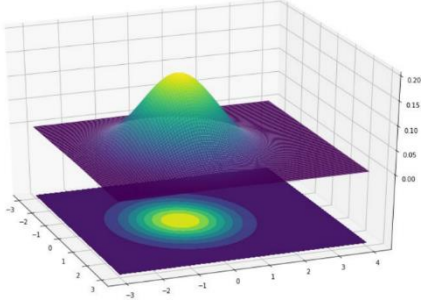


Gaussian Function

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

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

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



RBF Neural Network Learning

1. **Hidden layer learning**
   - Learn to **find the center** vectors of hidden nodes
   - Compute the output of hidden layer
2. **Output weights learning**
   - Any learning algorithms can be used such as
     - *LMS,*
     - *Generalized pseudo inverse,*
     - *etc.*

Hidden layer learning

Selecting the centers vectors

- The number of center vectors relate to the number of hidden nodes.
- There are many possible learning strategies that can be used to select the center vectors of RBFN
  a. The center vectors are randomly selected from training set
  b. Using the "K-means" clustering algorithm to set center vectors

Output weights learning

Compute the weight between hidden layer and output layer

- To adjust the output weights of RBF network, any algorithm can be used such as
  a. The *gradient descent* (similar to MLP neural network learning algorithm)

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

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_{1,1} & \cdots & \beta_{1,M} \\ \vdots & \ddots & \vdots \\ \beta_{L,1} & \cdots & \beta_{L,M} \end{bmatrix}$$

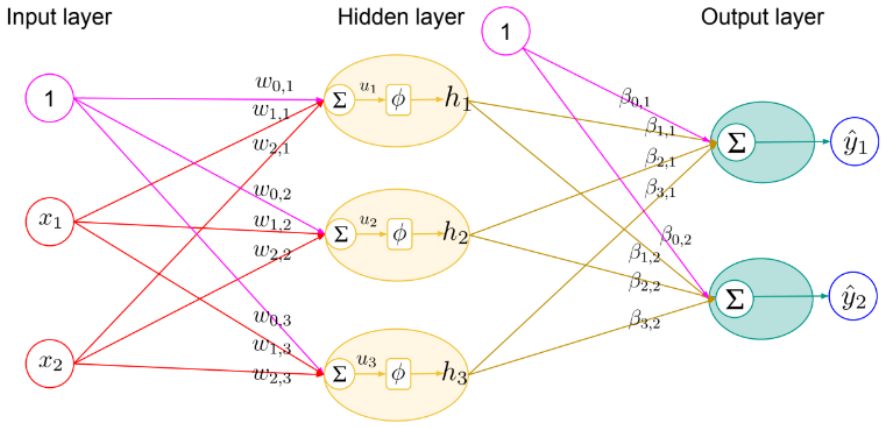
  b. The generalized pseudo inverse

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

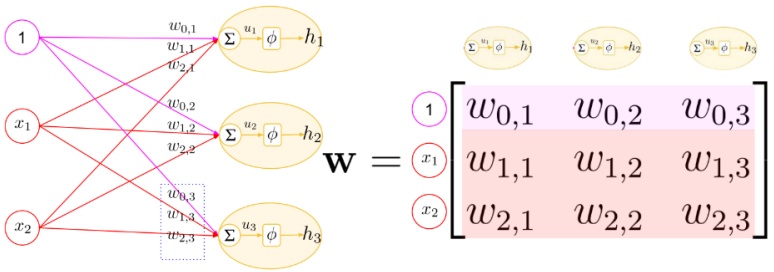
Random Neural Network

- Single hidden layer feedforward neural network
- **Input weights** are randomly chosen
- **Output weights** are analytically computed by the generalized Pseudo inverse matrix.
- No iterative tuning
- Fast learning model

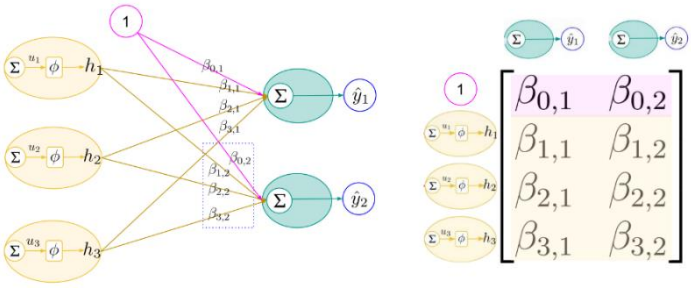
Single hidden layer feedforward neural network



Input Weights



Output Weights

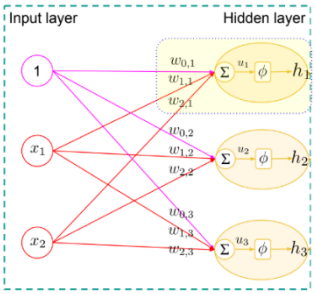


Learning Algorithm

Feed-forward Learning

- Hidden layer calculation
  - Random input weights
  - Compute Hidden layer output
- Output layer calculation
  - Compute output weight using the **generalized Pseudo inverse**

Learning Algorithm

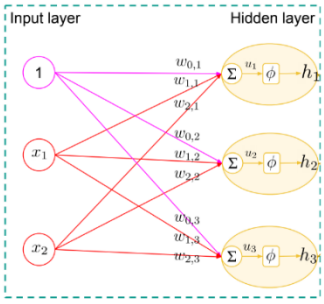


$$u_1 = w_{0,1} + \sum_{j=1}^{n} w_{j,1} x_{i,j}$$
$$= \mathbf{x}_{(i,:)} \cdot \mathbf{w}_{(:,1)}$$
$$= \begin{bmatrix} 1 & x_{i,1} & x_{i,2} \end{bmatrix} \cdot \begin{bmatrix} w_{0,1} \\ w_{1,1} \\ w_{2,1} \end{bmatrix}$$

$$h_{(i,1)} = \phi(\mathbf{x}_{(i,:)} \cdot \mathbf{w}_{(:,1)})$$

$i \in [1, \cdots, N]$

Learning Algorithm

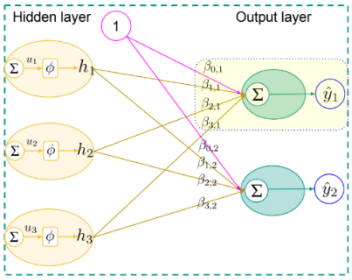


$$\mathbf{H} = \phi(\mathbf{X} \cdot \mathbf{W})$$

$$\mathbf{H} = \begin{bmatrix} h_{1,1} & h_{1,2} & h_{1,3} \\ \vdots & \vdots & \\ h_{N,1} & h_{N,2} & h_{N,3} \end{bmatrix}$$

$i \in [1, \cdots, N]$

Learning Algorithm

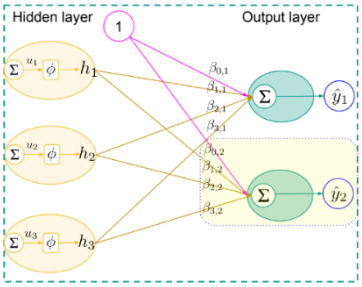


$$\hat{y}_{i,1} = \beta_{0,1} + \sum_{l=1}^{L} h_{i,l} \beta_{l,1}$$
$$= \mathbf{h}_{(i,:)} \cdot \beta_{(:,1)}$$
$$= \begin{bmatrix} 1 & h_{i,1} & h_{i,2} & h_{i,3} \end{bmatrix} \cdot \begin{bmatrix} \beta_{0,1} \\ \beta_{1,1} \\ \beta_{2,1} \\ \beta_{3,1} \end{bmatrix}$$

$$\hat{y}_{(i,1)} = \mathbf{h}_{(i,:)} \cdot \beta_{(:,1)}$$

$i \in [1, \cdots, N]$

Learning Algorithm

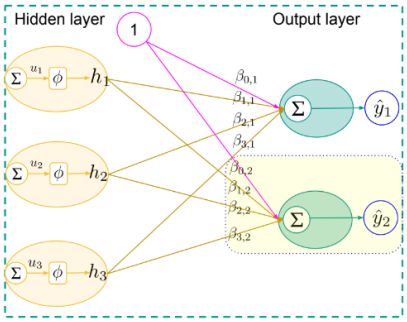


$$\hat{y}_{i,2} = \beta_{0,2} + \sum_{l=1}^{L} h_{i,l} \beta_{l,2}$$
$$= \mathbf{h}_{(i,:)} \cdot \beta_{(:,2)}$$
$$= \begin{bmatrix} 1 & h_{i,1} & h_{i,2} & h_{i,3} \end{bmatrix} \cdot \begin{bmatrix} \beta_{0,2} \\ \beta_{1,2} \\ \beta_{2,2} \\ \beta_{3,2} \end{bmatrix}$$

$$\hat{y}_{(i,2)} = \mathbf{h}_{(i,:)} \cdot \beta_{(:,2)}$$

$i \in [1, \cdots, N]$

Learning Algorithm



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

$$\boldsymbol{\beta} = ?$$

Generalized Pseudo inverse

Compute output weight using the **Generalized Pseudo inverse**

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

numpy.linalg.pinv( )

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