

Neural Networks



شما در حال حاضر، در حال استفاده از یک سیستم عصبی بیولوژیکی پیچیده هستید، جهت فهم مطالب هستید.

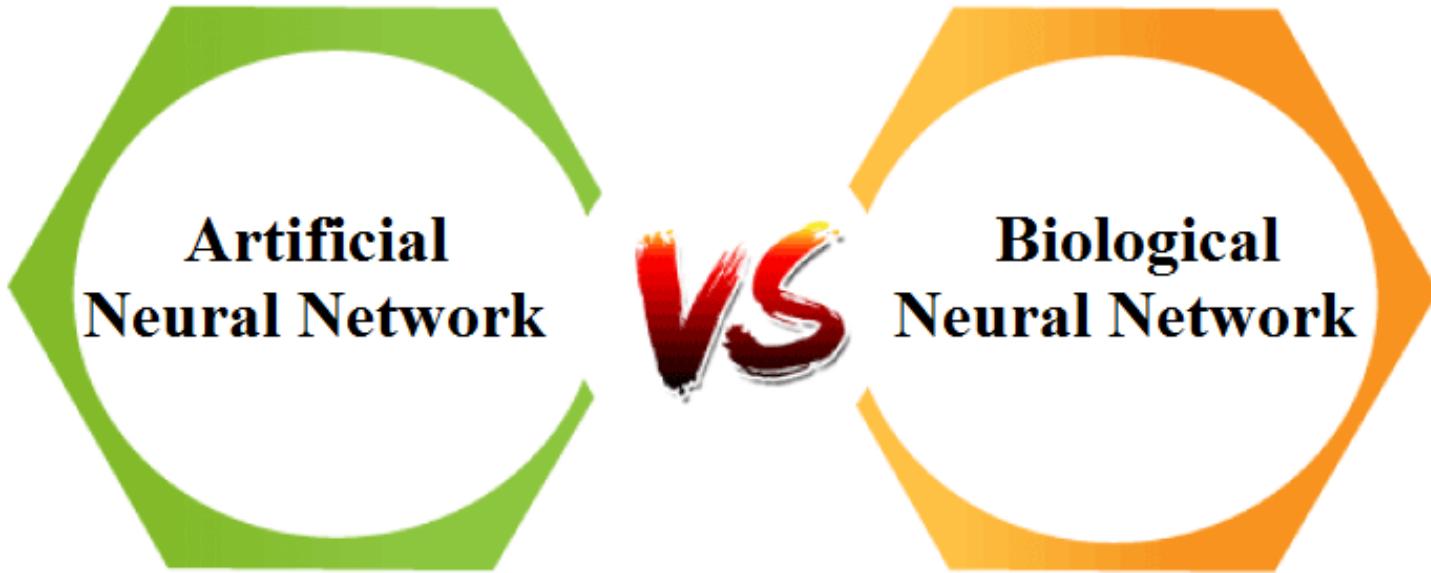
این سیستم دو درصد از وزن بدن شما را تشکیل می‌دهد و بیش از ۲۰ درصد کل اکسیژن بدن را مصرف می‌کند.



سیستم عصبی شما یکی از سریع ترین و دقیق ترین سیستم های محاسباتی دنیاست.

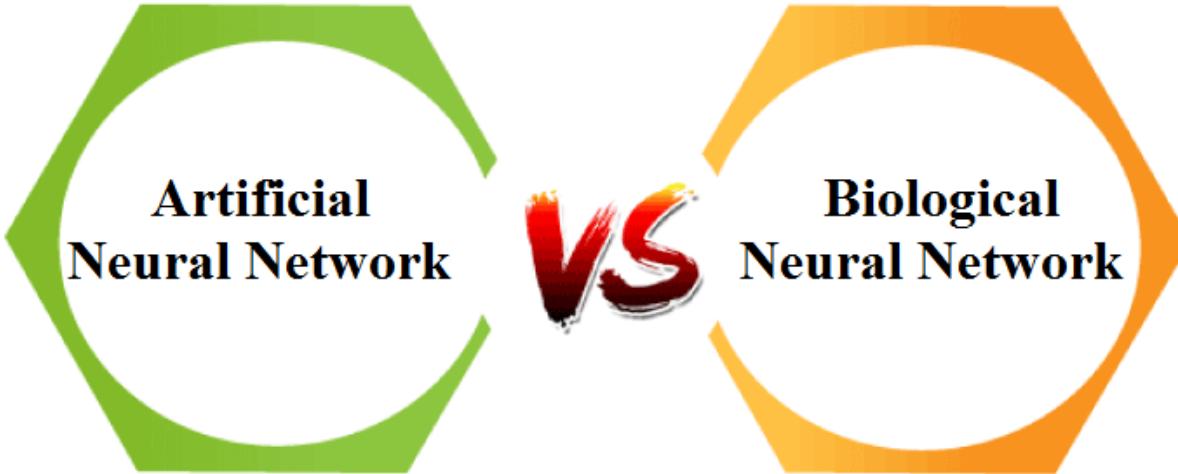


حجم عظیمی از اطلاعات و سیگنال‌ها جهت این کار و در طی زمانی کمتر از چند صدم ثانیه بایستی جمع آوری و محاسبه شود.



پیاده سازی ویژگی های شگفت انگیز مغز در یک سیستم عصبی مصنوعی همیشه وسوسه انگیز و مطلوب بوده است.

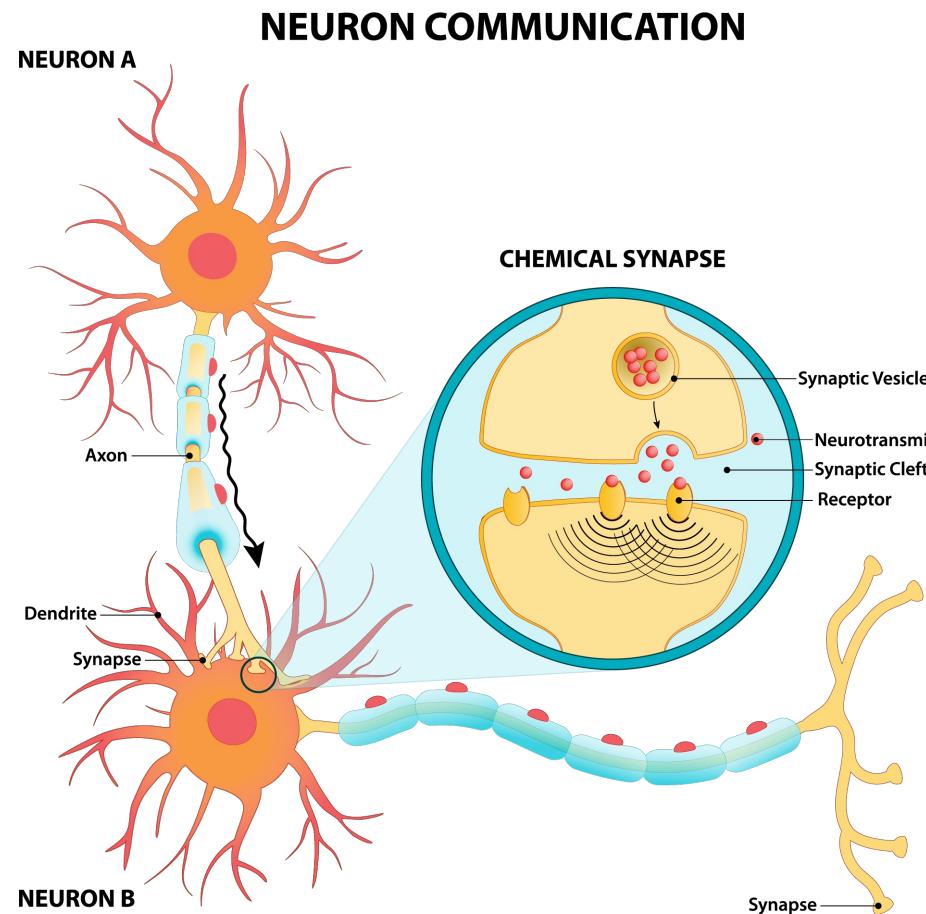
اما سالها تلاش محققین از سال ۱۹۱۱ ثابت می کند که **مغز بشر دست نیافتنی** است.



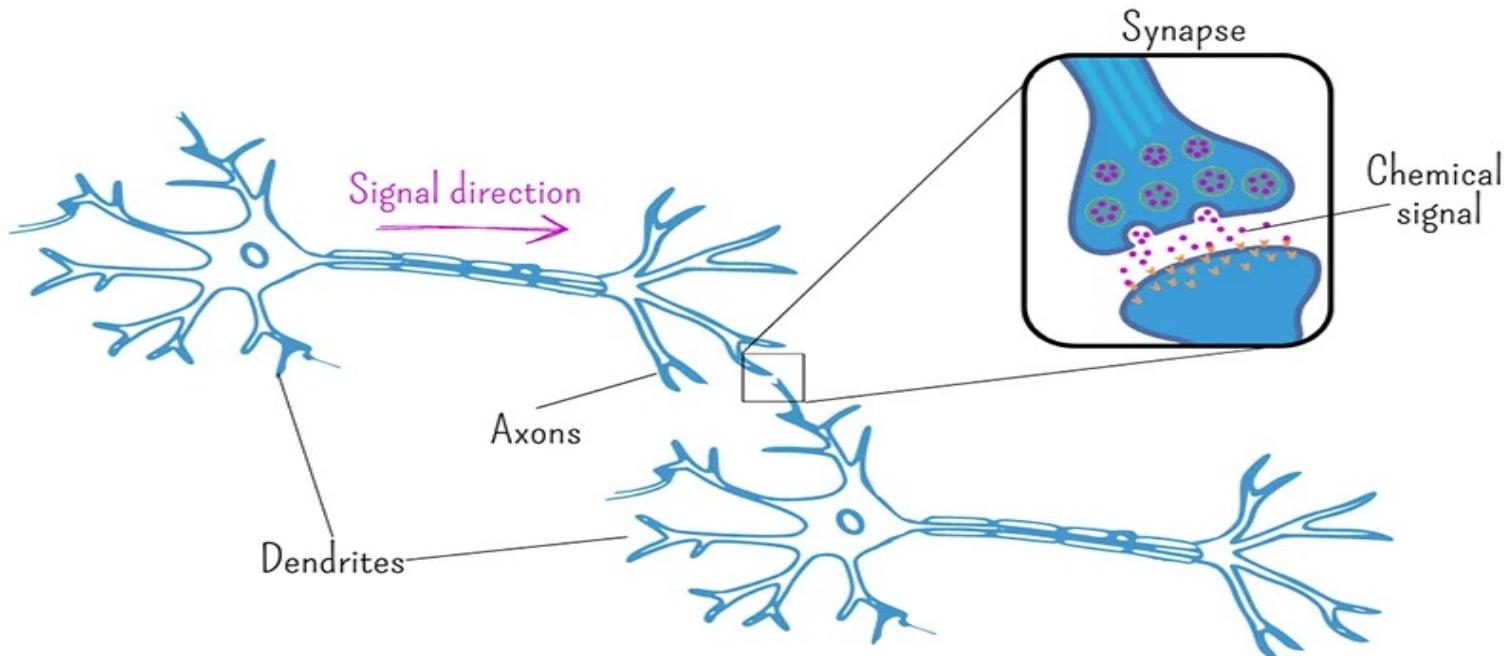
اگر چه نرون های بیولوژیکی از نرون های مصنوعی که توسط های مدار های الکتریکی ساخته می شوند، بسیار کندتر هستند (یک میلیون)، اما عملکرد مغز خیلی سریعتر از یک کامپیوتر معمولی است.

علت این پدیده بیشتر به دلیل ساختار کاملاً موازی نرون های بیولوژیکی است. در سیستم های بیولوژیکی همه نرون های بطور همزمان کار می کنند و پاسخ می دهند.

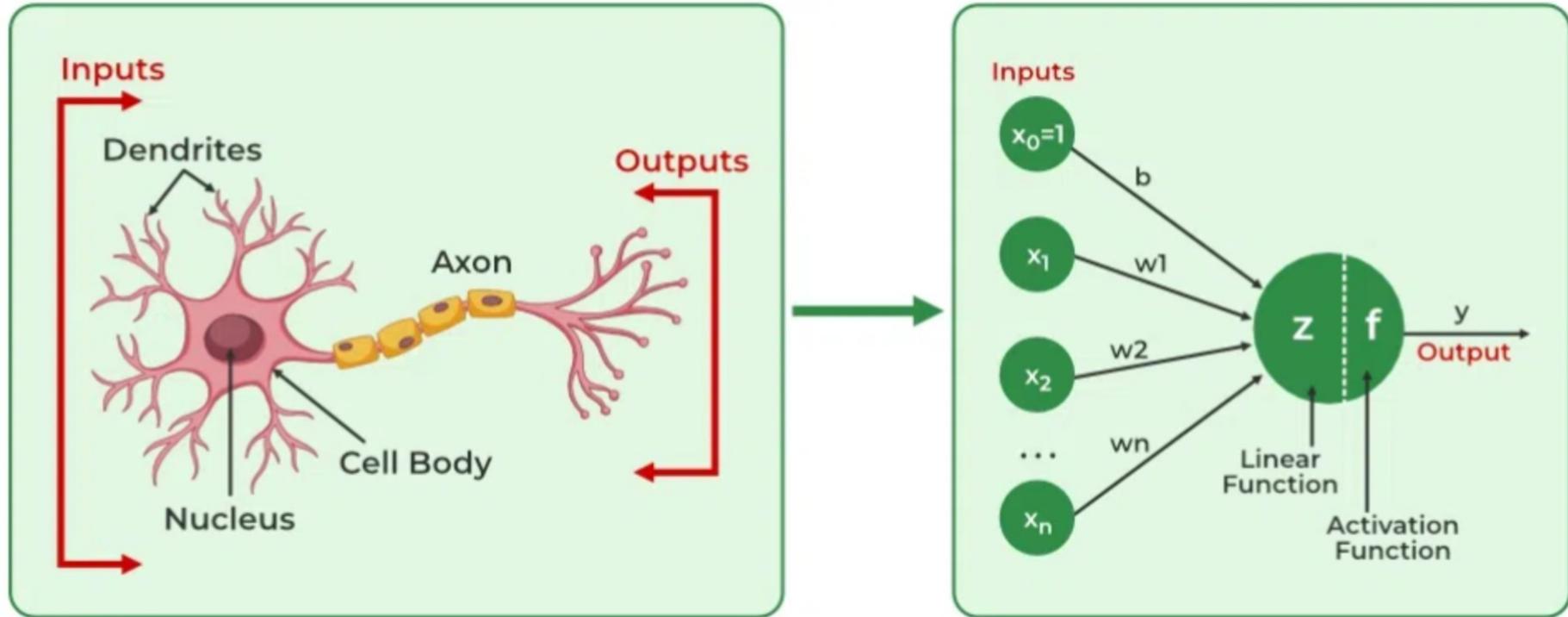
ساختار نرون



- ▶ هسته : انرژی لازم برای فعالیت های نرون را فراهم می کند.
- ▶ دندریت : دریافت کننده سیگنال های الکتروشیمیایی
- ▶ اکسون : سیگنال های الکتروشیمیایی نرون را به سایر نرون ها منتقل می کند.
- ▶ سیناپس : محل تلاقي اکسون یک نرون به دندریت نرون دیگر



مغز شما یک سیستم پردازش اطلاعات با ساختار موازی است که از 10^0 تریلیون (10^{11}) نرون به هم متصل با 10^{16} ارتباط تشکیل شده است.



شبکه های عصبی مصنوعی الگوریتم هایی هستند که سعی می کنند از ساختار شبکه ای ،

نحوه پردازش موازی و شیوه یادگیری مغز الگو بگیرند.



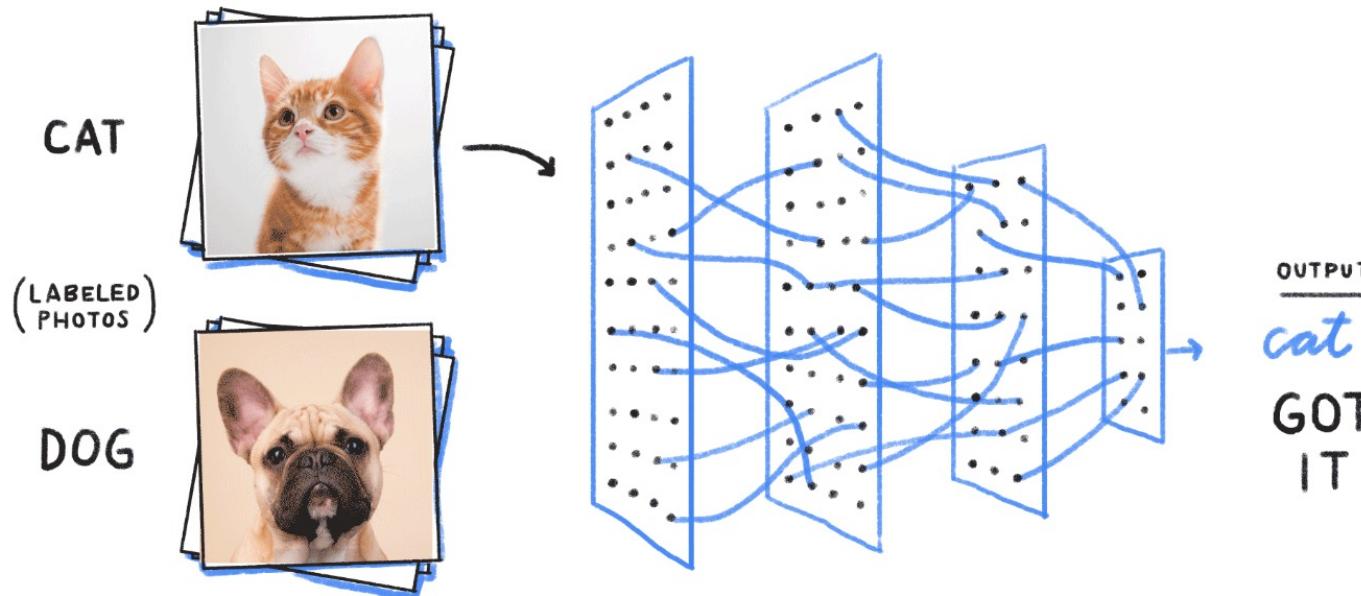
یادگیری به معنای ایجاد ارتباطات جدید بین نرون ها یا تنظیم مجدد ارتباطات موجود بین آنهاست.

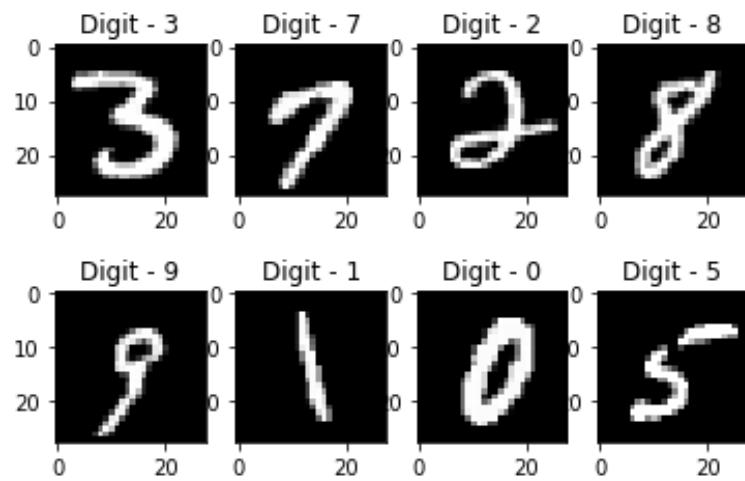
سوال:

▶ آیا می توان یک شبکه کوچک از نرون های مصنوعی ساخت،
به طوری که جهت حل مسائل پیچیده (که همان یادگیری
نگاشت هاست) آموزش پذیر باشد؟

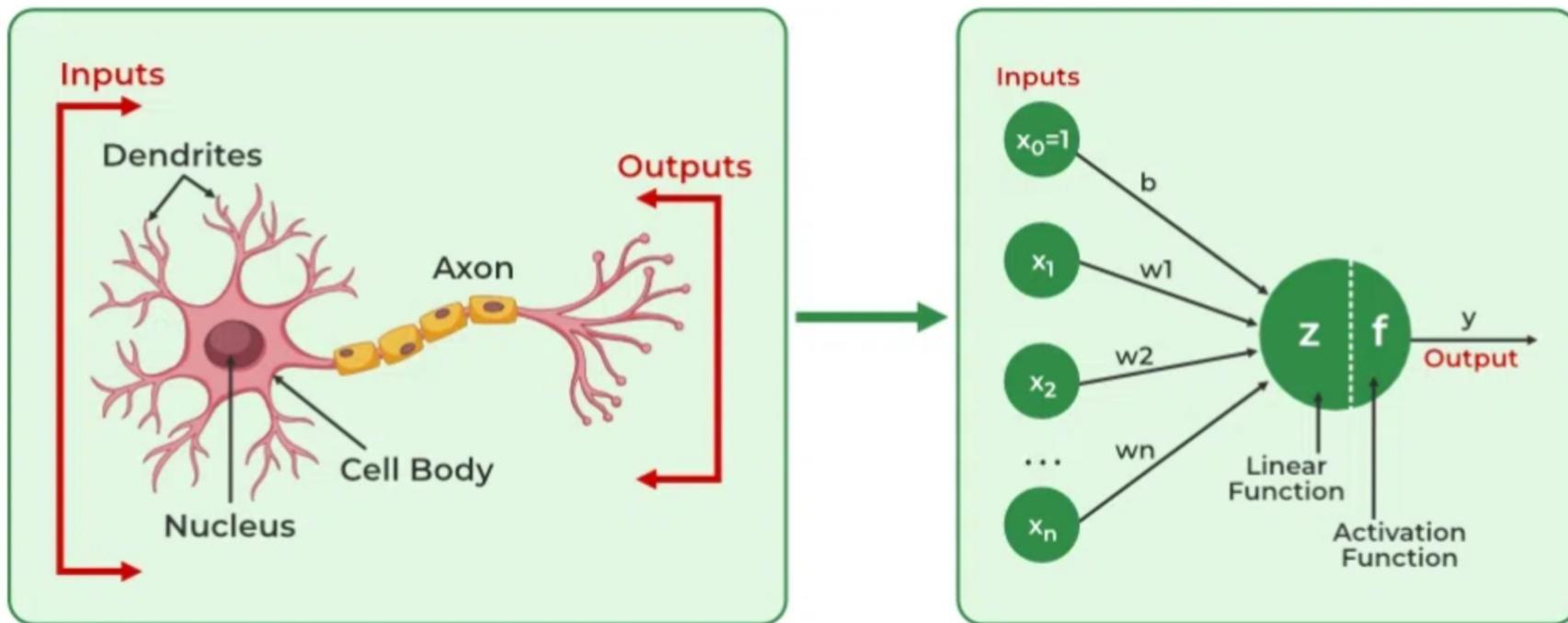
بهترین کاربرد شبکه های عصبی مصنوعی

زمانی است که تعداد feature های شما زیاد باشد و به همین دلیل سایر مدل های یادگیری ماشین روی داده شما عملکرد خوبی نداشته باشند.

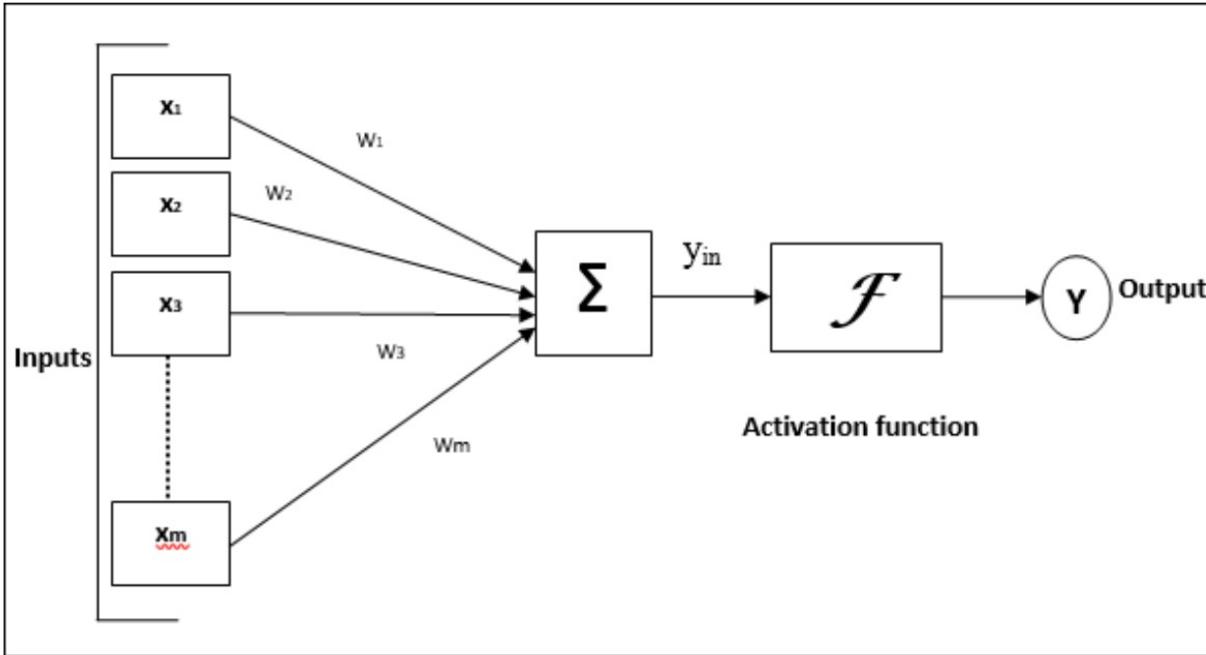




Neuron model



Neuron model



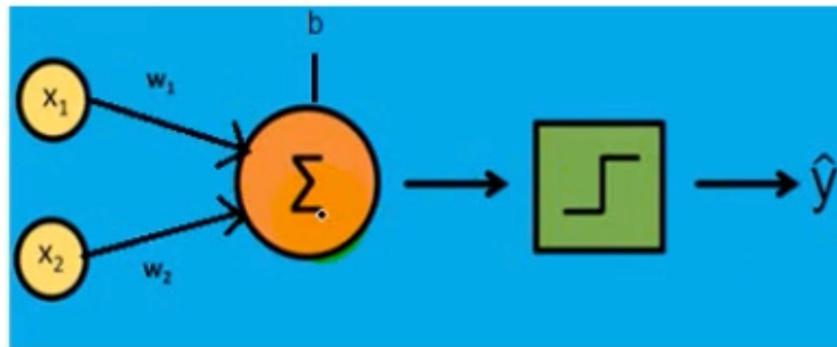
$$y = f(\sum_{i=1}^n w_i x_i + b)$$

Here f is a sign function

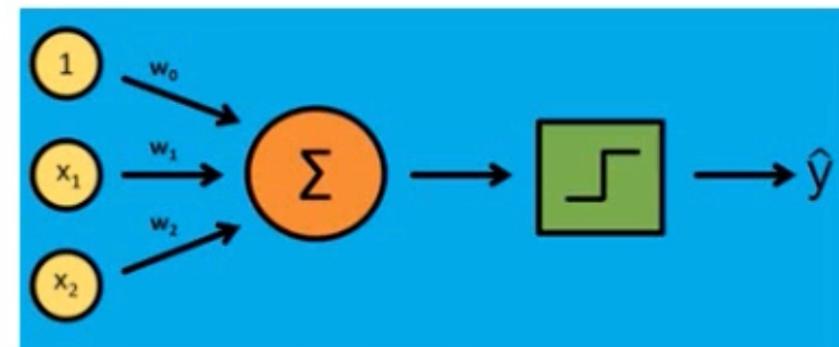
As we see single neuron is a linear classifier

Perceptron

شبکه های پرسپترون در سال ۱۹۵۸ توسط روزنبلات معرفی شد.
پرسپترون یک نرون ساده است که داده ها را به صورت خطی به دو کلاس تعریف می کند.

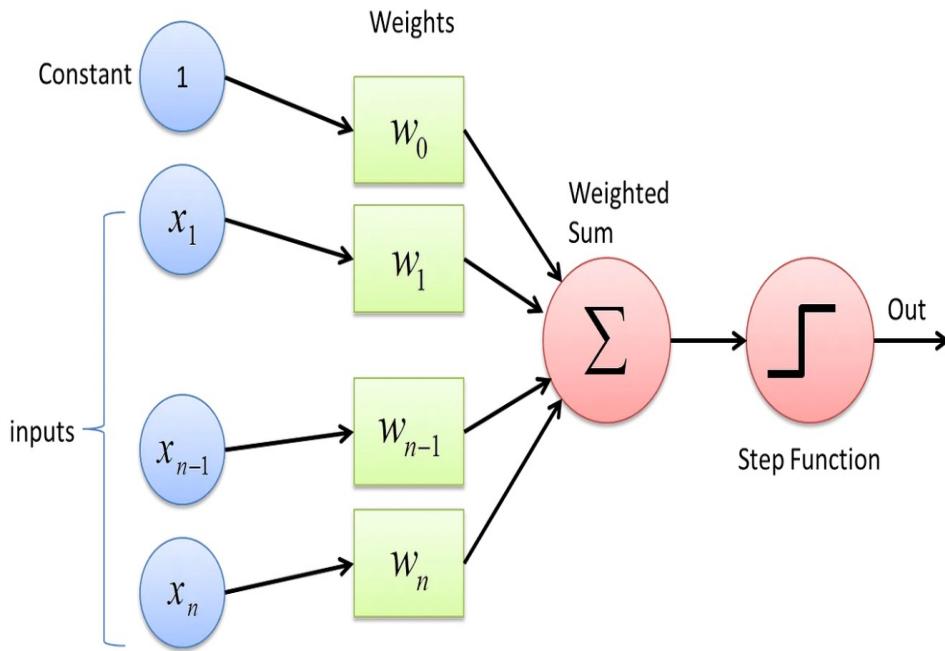


$$y = f(\sum_{i=1}^n w_i x_i + b)$$



$$y = f(\sum_{i=0}^n w_i x_i)$$

Single layer perceptron



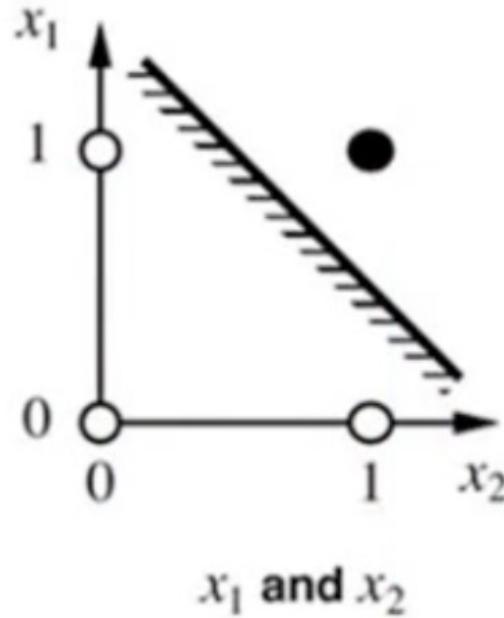
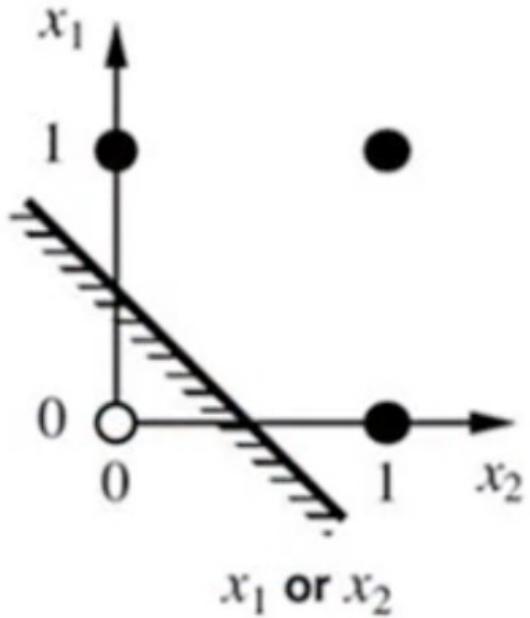
The perceptron consists of 4 parts.

1. Input values or One input layer
2. Weights and Bias
3. Net sum
4. Activation Function

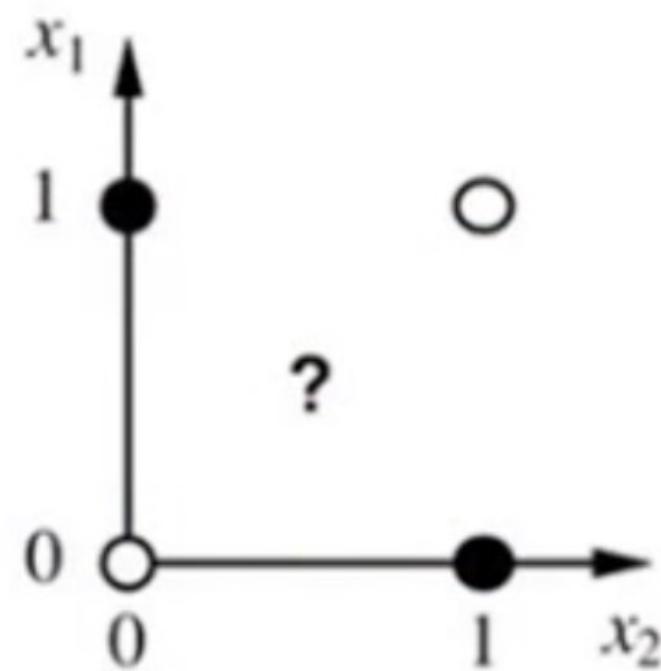
$$\begin{cases} a = 0 & n < 0 \\ a = 1 & n > 0 \end{cases}$$

تابع محرک در پرسپترون تابع پله ای یا sign است

پیاده سازی گیت های منطقی با پر سپترون



پیاده سازی گیت های منطقی با پر سپترون



$x_1 \text{ xor } x_2$

قانون آموزش پرسپترون (قانون دلتا)

برای پیدا کردن وزن های پرسپترون ، از یک مقدار تصادفی اولیه شروع می کنیم و بعد با استفاده از قانون دلتا وزن هارا بروزرسانی می کنیم.

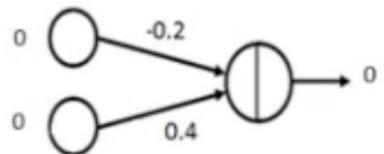
$$w_{i\text{ جدید}} = w_{i\text{ قدیم}} + \alpha e x_i$$

$$e = t - y$$

مقدار واقعی : t

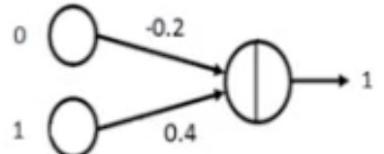
مقدار پیش بینی شده : y

پیاده سازی OR با پر سپترون تک لایه



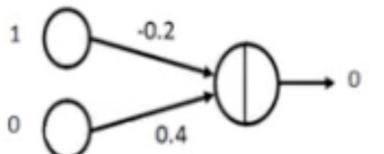
$$y = f(0 \times -0.2 + 0 \times 0.4) = f(0) = 0$$

$$e = 0 - 0 = 0$$



$$y = f(0 \times -0.2 + 1 \times 0.4) = f(0.4) = 1$$

$$e = 0 - 0 = 0$$

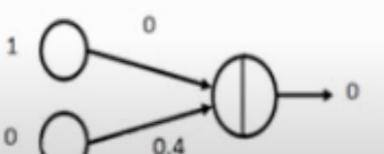


$$y = f(1 \times -0.2 + 0 \times 0.4) = f(-0.2) = 0$$

$$e = 1 - 0 = 1$$

$$w_1 = w_1 + 1 \times 0.2 \times x_1 = -0.2 + 0.2 = 0$$

$$w_2 = w_2 + 1 \times 0.2 \times x_2 = 0.4 + 0 = 0.4$$



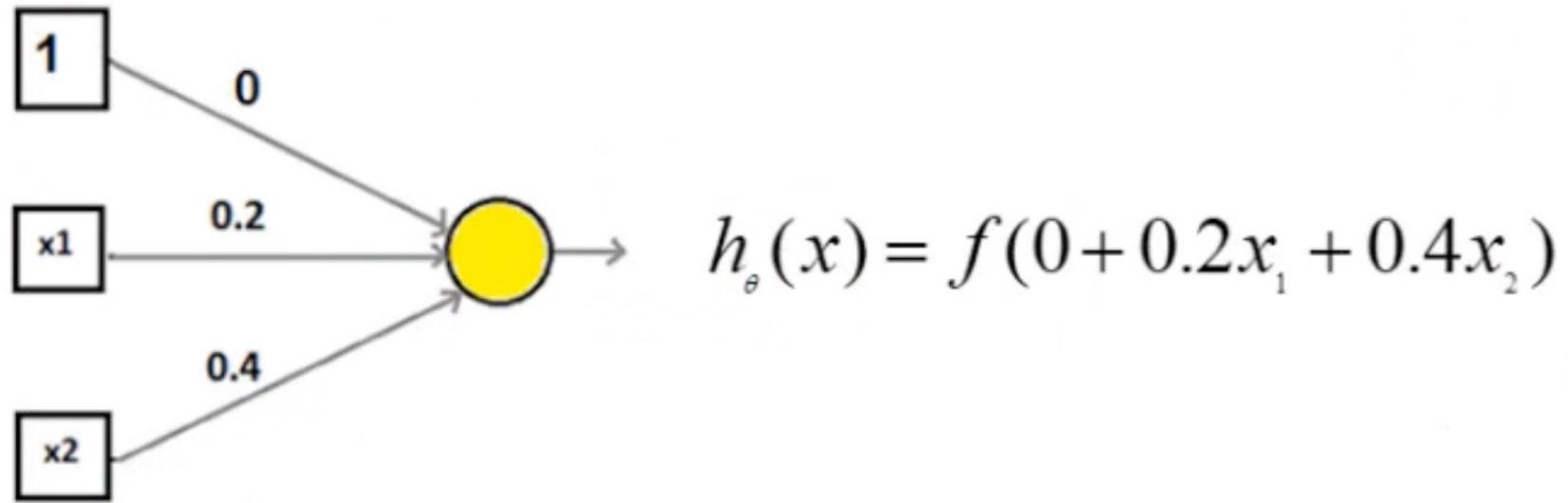
$$y = f(1 \times 0 + 1 \times 0.4) = f(0.4) = 1$$

$$e = 1 - 1 = 0$$

پیاده سازی OR با پرسپترون تک لایه

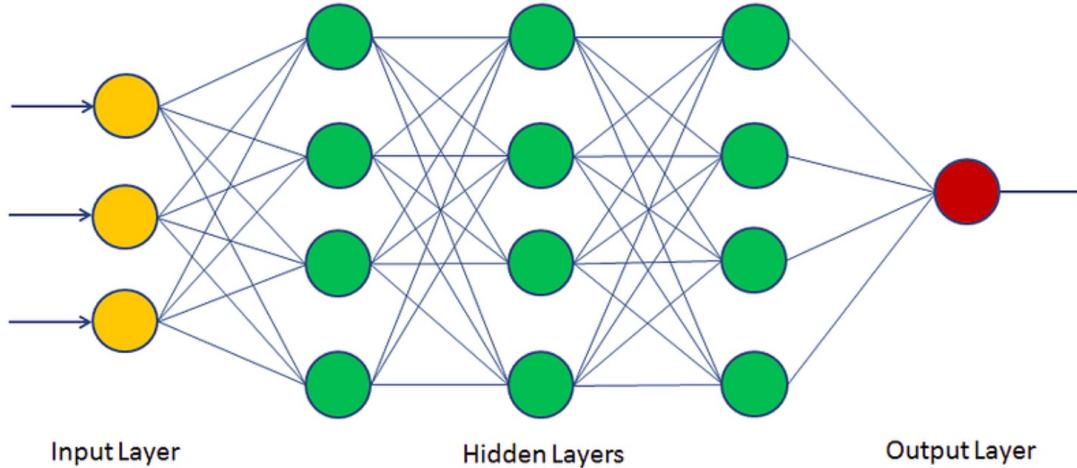
epoch	x1	x2	t	y	e	w1	w2
1	0	0	0	0	0	-0.2	0.4
	0	1	1	1	0	-0.2	0.4
	1	0	1	0	1	0	0.4
	1	1	1	1	0	0	0.4
2	0	0	0	0	0	0	0.4
	0	1	1	1	0	0	0.4
	1	0	1	0	1	0.2	0.4
	1	1	1	1	0	0.2	0.4
3	0	0	0	0	0	0.2	0.4
	0	1	1	1	0	0.2	0.4
	1	0	1	1	0	0.2	0.4
	1	1	1	1	0	0.2	0.4

پیاده سازی OR با پرسپترون تک لایه

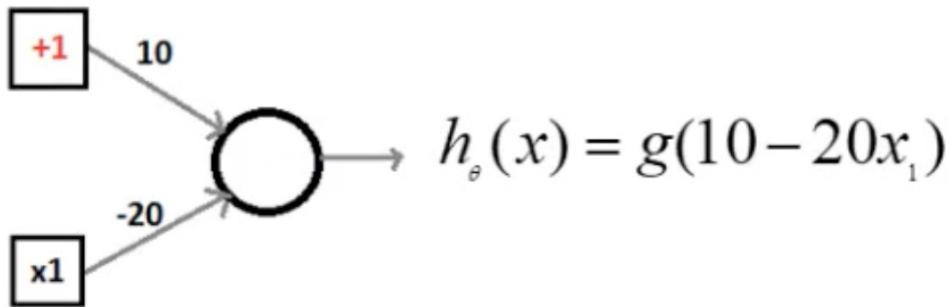


Multi layer perceptron (MLP)

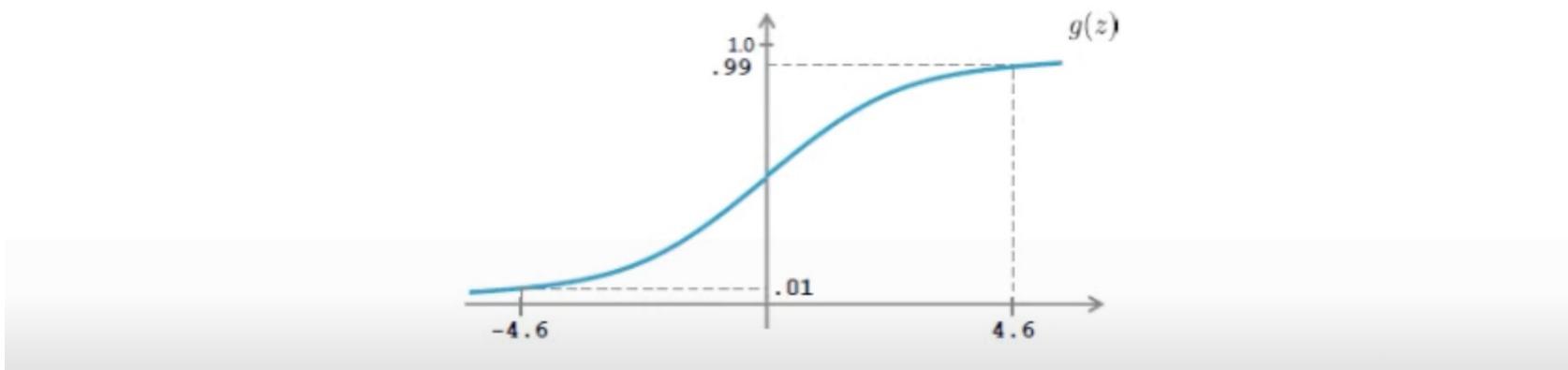
- همان طور که دیدیم شبکه های عصبی پرسپترون تک لایه فقط قادر به تفکیک کلاس هایی هستند که به صورت خطی جدایی پذیر هستند.
- اما می دانیم که بیشتر مسایل دنیای واقعی به صورت خطی جدایی پذیر نیستند به همین دلیل از الگوریتم پرسپترون چند لایه برای آنها استفاده می کنیم.



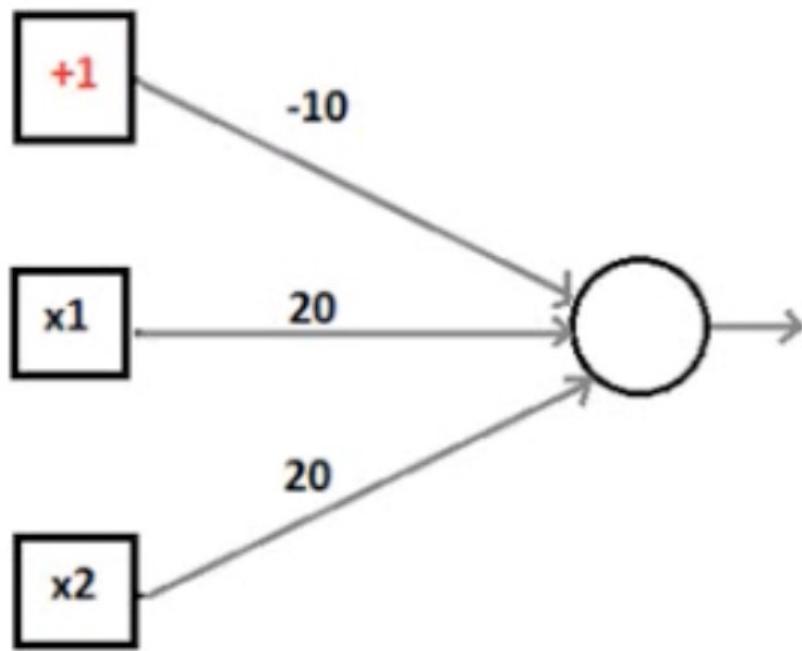
پیاده سازی گیت NOT



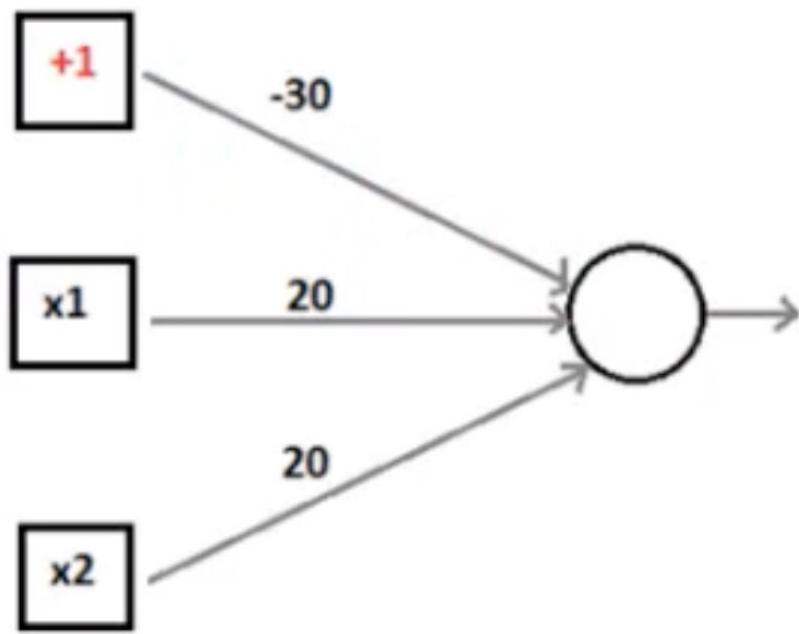
x_1	$h(x)$
0	$g(10)=1$
1	$g(-10)=0$



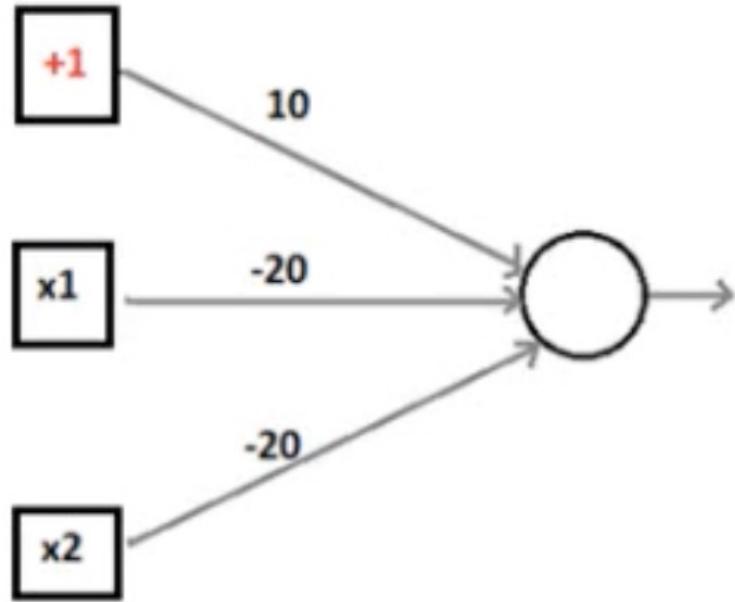
پیاده سازی گیت OR



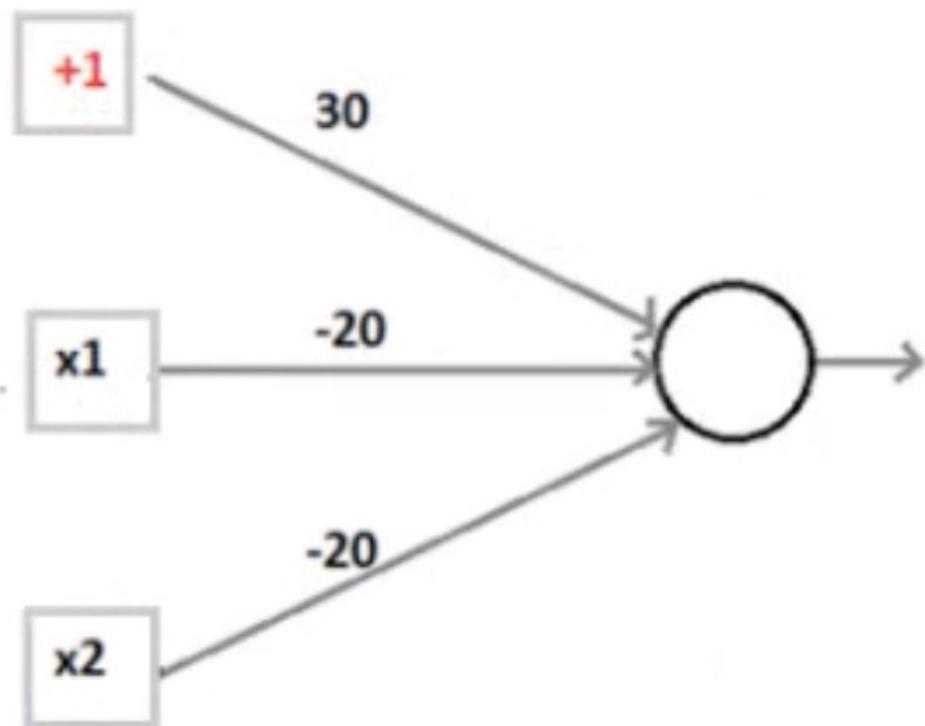
پیاده سازی گیت AND



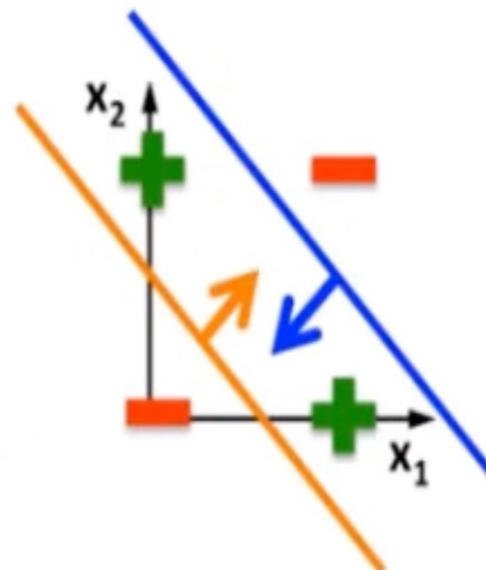
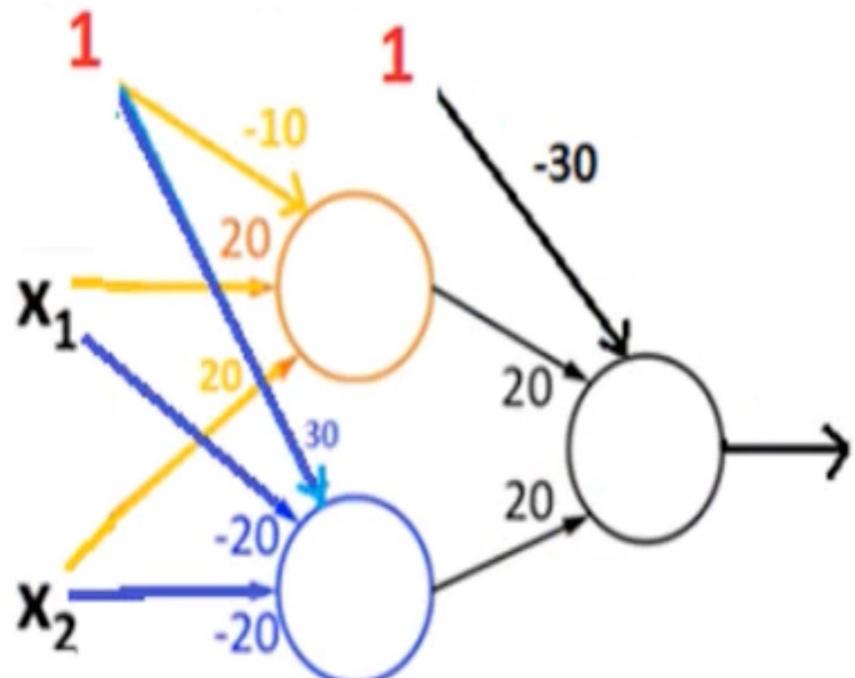
$(not\ x_1) \ AND \ (not\ x_2)$



(not x_1) AND (not x_2)



پیاده سازی گیت XOR

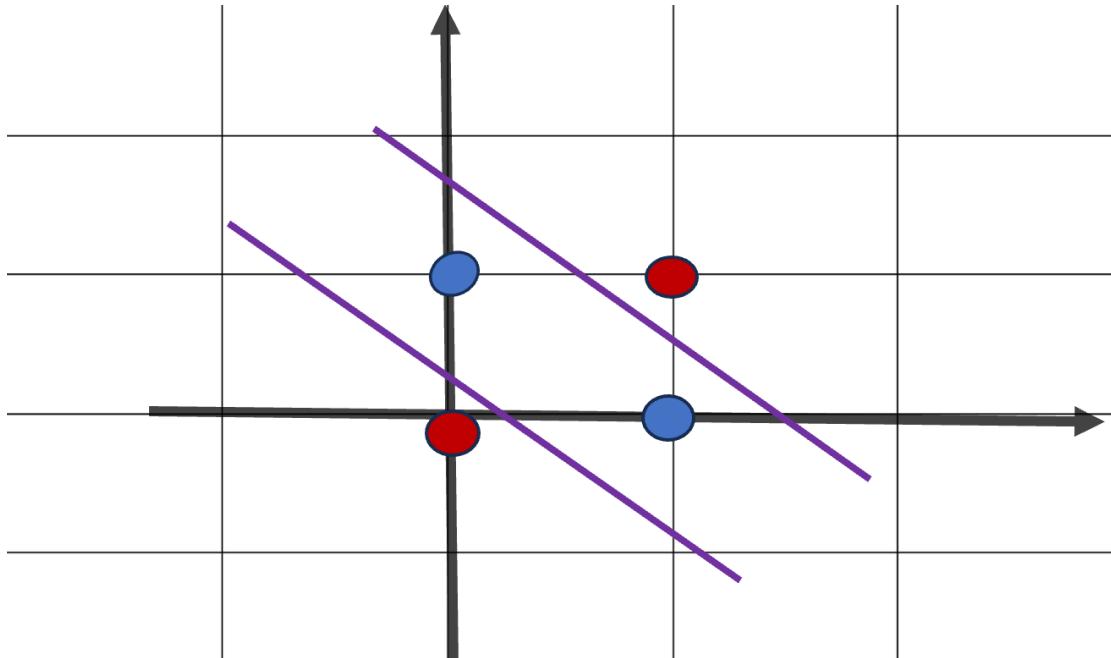


30

پیاده سازی گیت XNOR

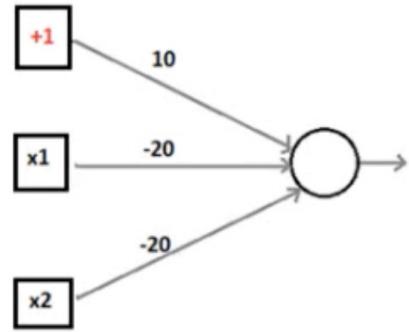
X1	X2	XNOR
0	0	1
0	1	0
1	0	0
1	1	1

$$\overline{X_1} \cdot \overline{X_2} + X_1 \cdot X_2$$

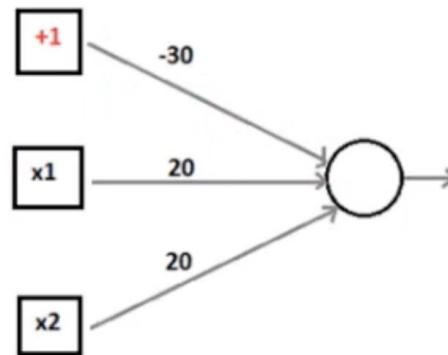


پیاده سازی گیت XNOR

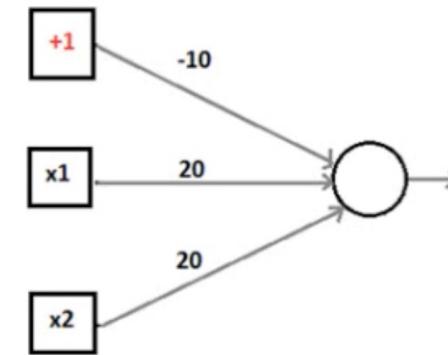
$(not\ x_1) \ AND \ (not\ x_2)$



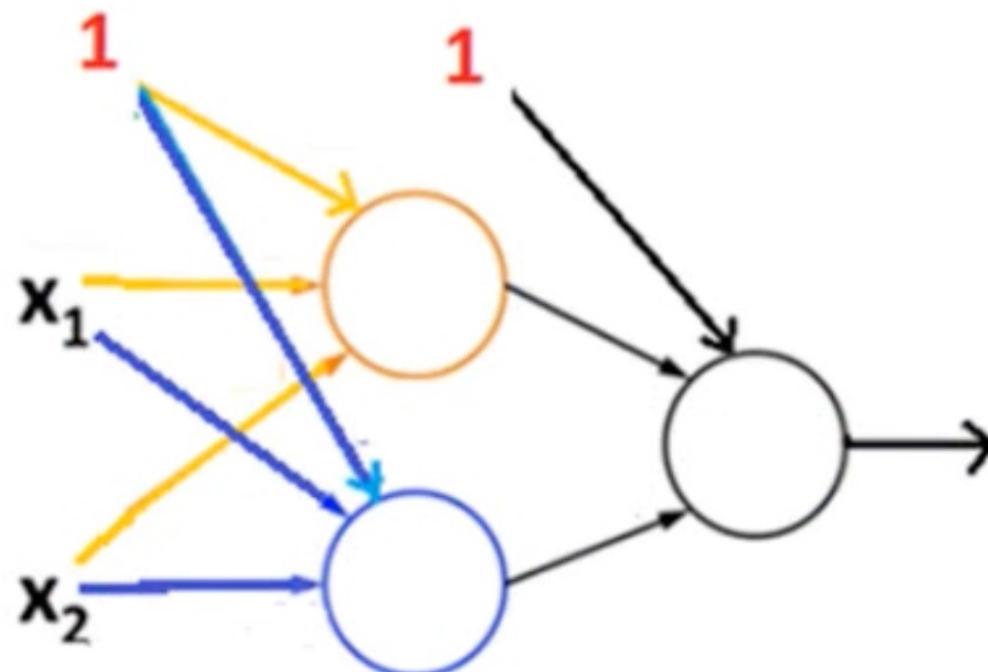
گیت AND



گیت OR



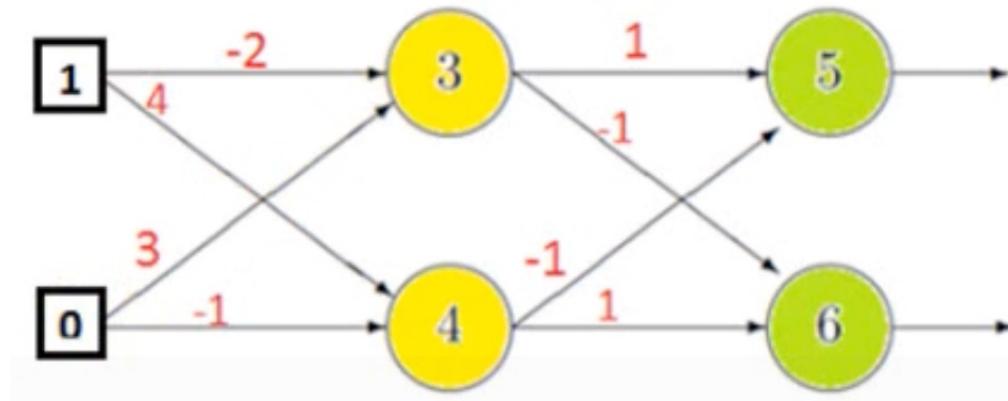
پیاده سازی گیت XNOR



مثال

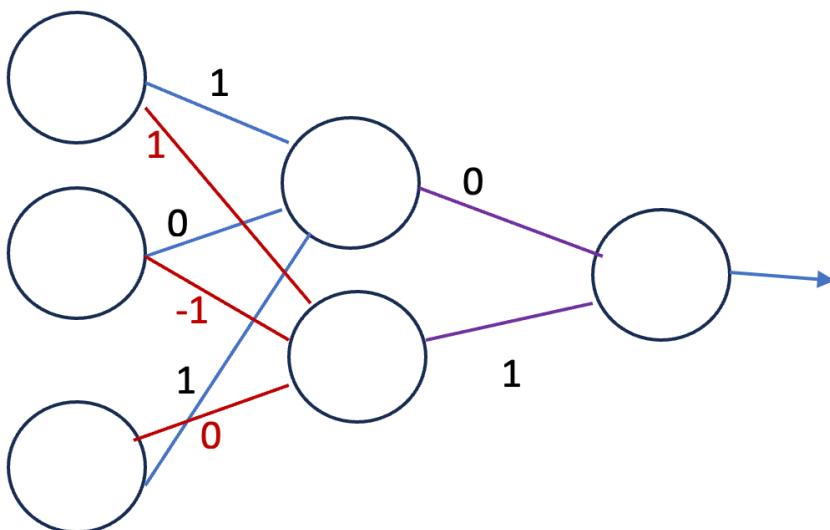
خروجی شبکه عصبی زیر با یک لایه مخفی را مشخص کنید. فرض کنید Activation function به شکل رو برو است.

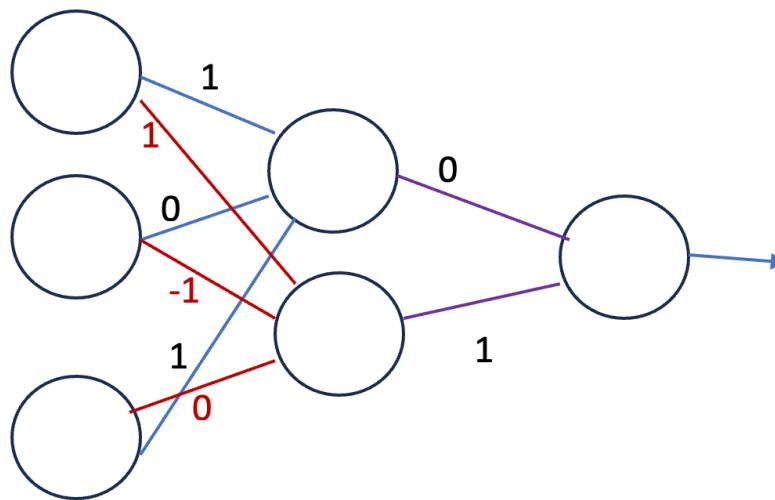
$$f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$



مثال

فرض کنید یک شبکه MLP با سه لایه داریم که در لایه اول سه نرون، در لایه دوم ۲ نرون و در لایه سوم یا خروجی فقط یک نرون دارد. اگر تابع محرک بصورت $\sigma(z) = \max(0, z)$ باشد خروجی را به از ورودی $[1 \ 2 \ 1]$ محاسبه کنید.



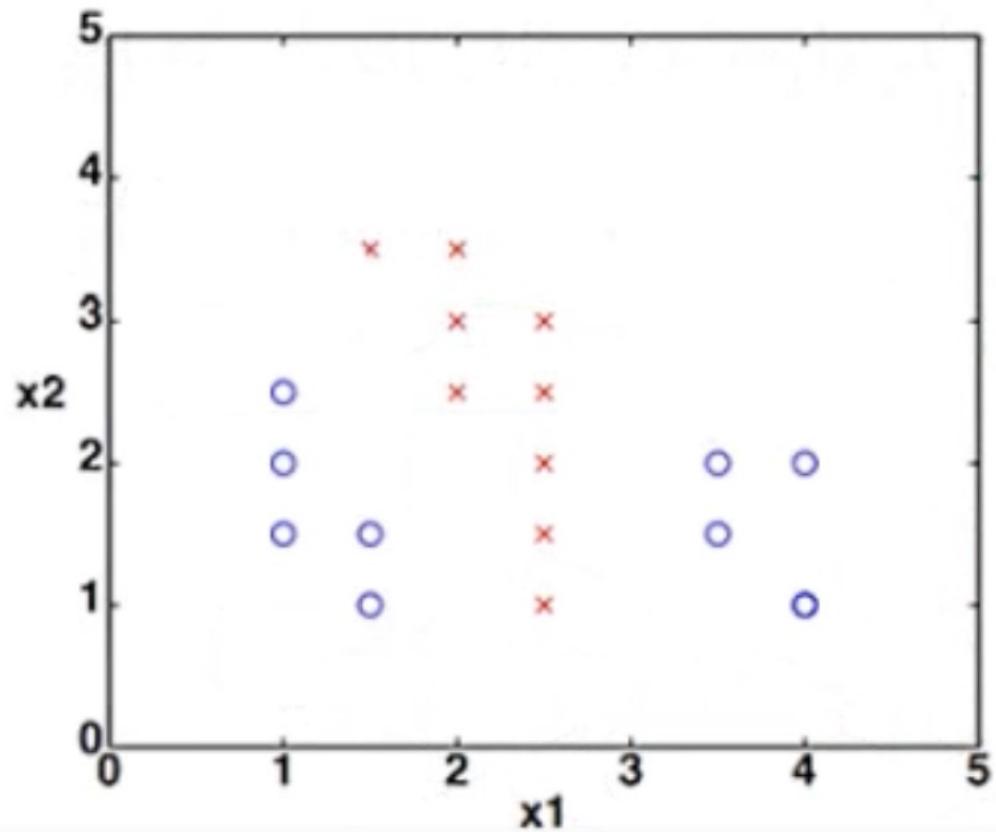


$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \end{bmatrix} \longrightarrow \sigma \left(\begin{bmatrix} 2 \\ -1 \end{bmatrix} \right) = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$$

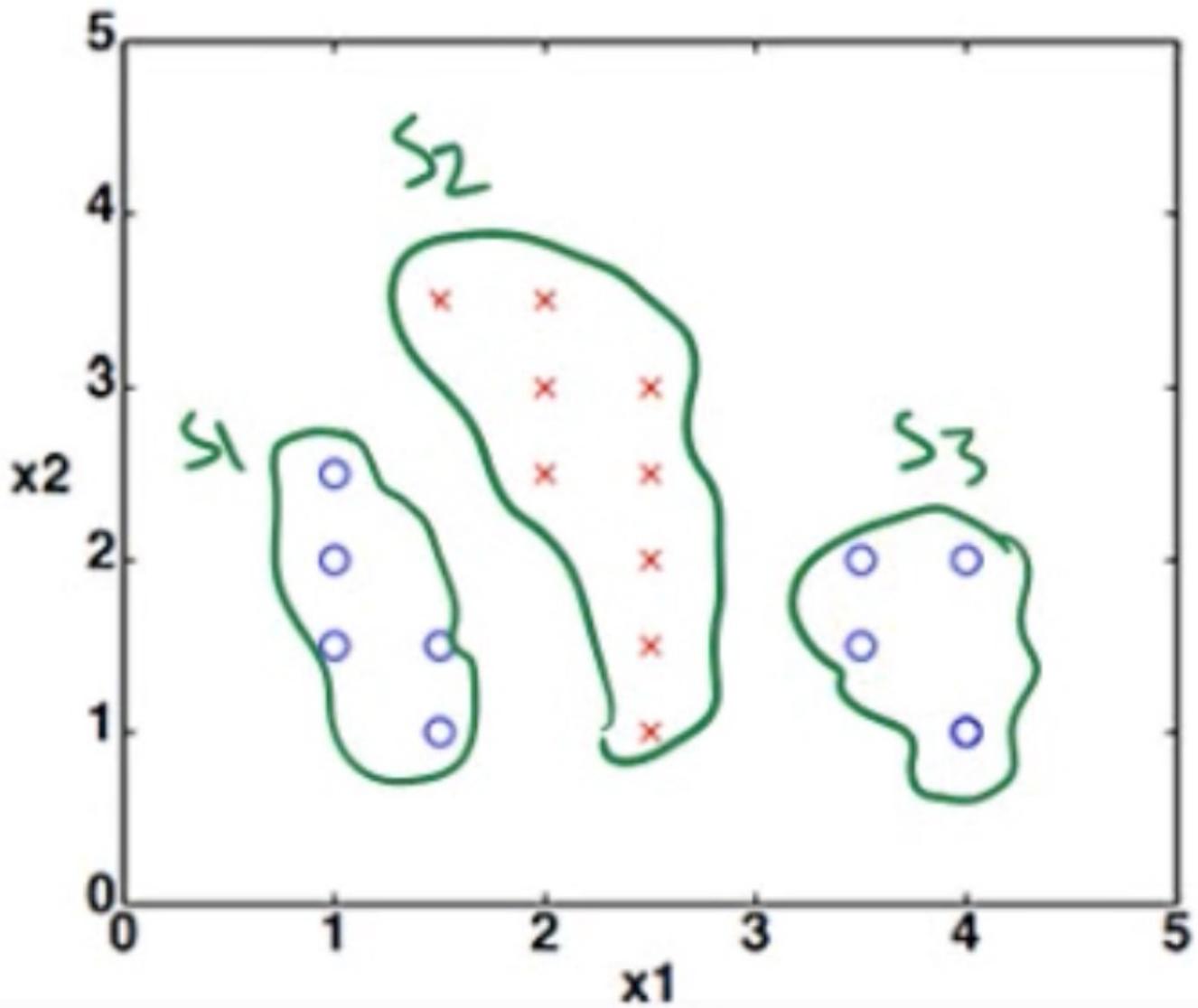
$$\begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \end{bmatrix} = 0 \longrightarrow \sigma(0) = 0$$

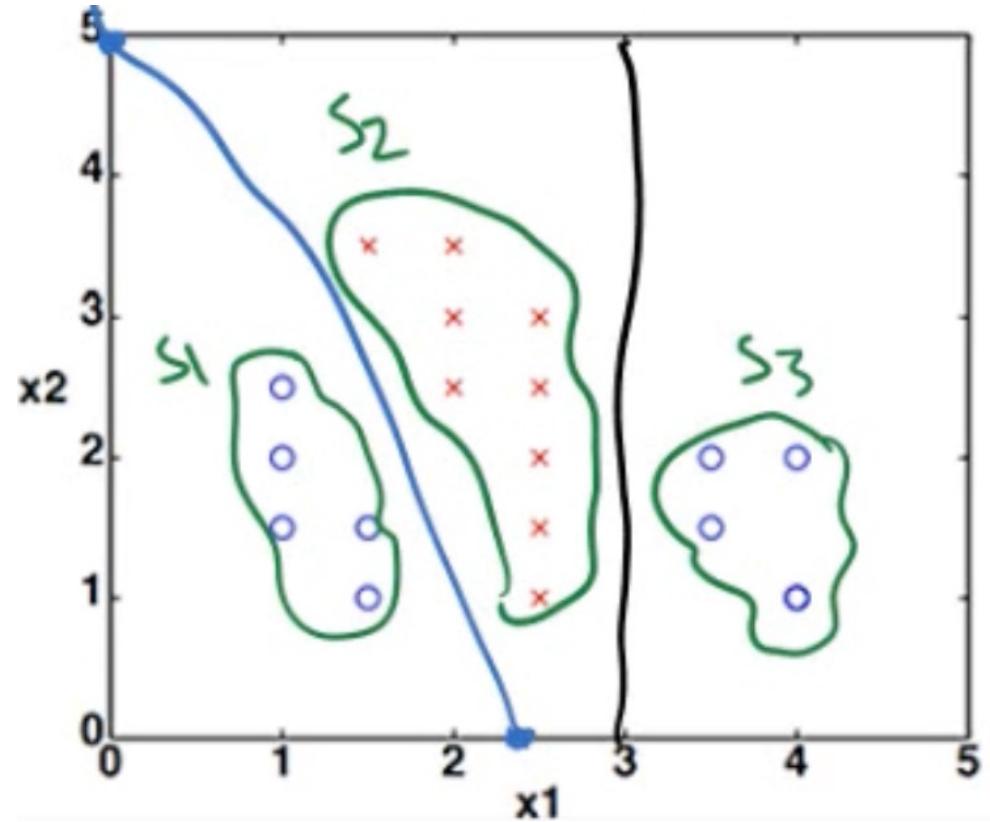
مثال

به کمک یک شبکه MLP داده‌های مقابل را دسته‌بندی (classify) کنید.



$$\phi(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$$





$$h_1(x) = \phi(-x_1 + 3)$$

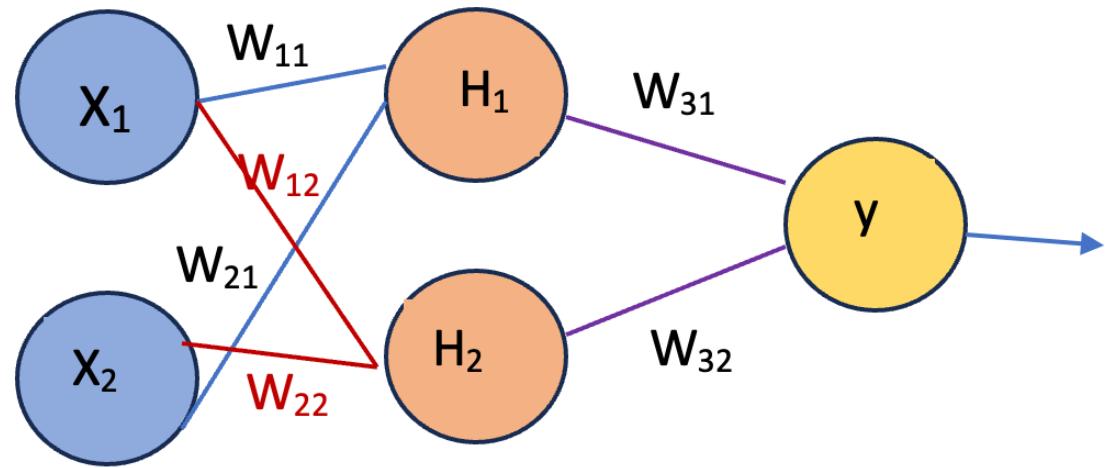
$$h_2(x) = \phi(2x_1 + x_2 - 5)$$

$$y = \phi(h_1(x) + h_2(x) - 1.5)$$

$(1,1) \rightarrow 0$

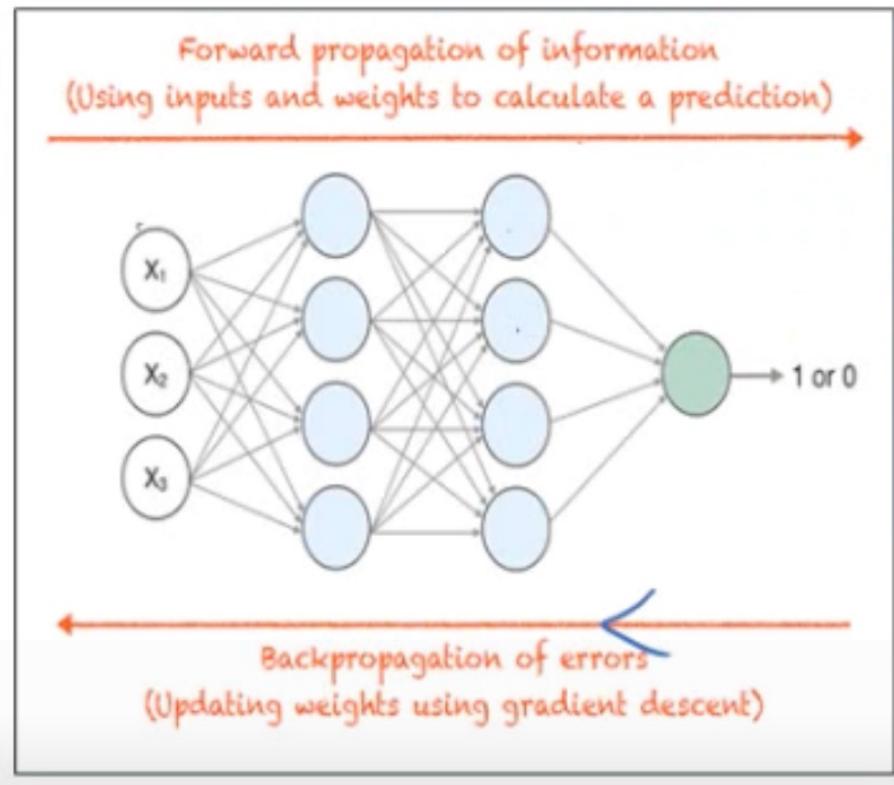
$(4,1) \rightarrow 0$

$(2,3) \rightarrow 1$



الگوریتم error backpropagation

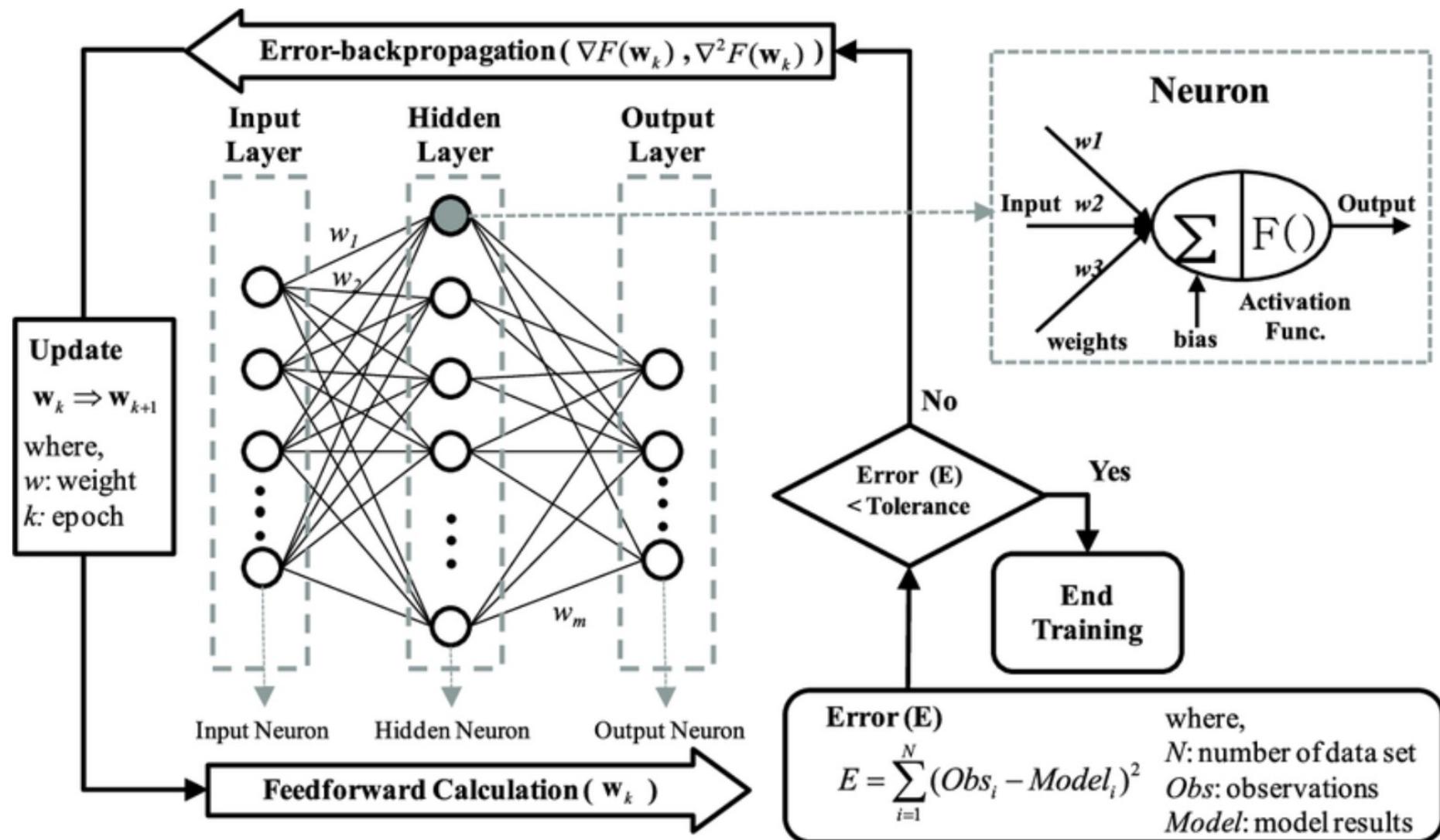
هنگامی که خطاپی در شبکه های عصبی چند لایه feed-forward رخ می دهد. این خطا باید به سمت ورودی برگشت داده شود و باعث اصلاح وزن ها گردد. پس این الگوریتم در سه مرحله اتفاق



می افتد.

- ۱- پیش خور کردن (Feed-forward) ورودی
- ۲- محاسبه و پس انتشار خطای مربوطه
- ۳- تنظیم وزن

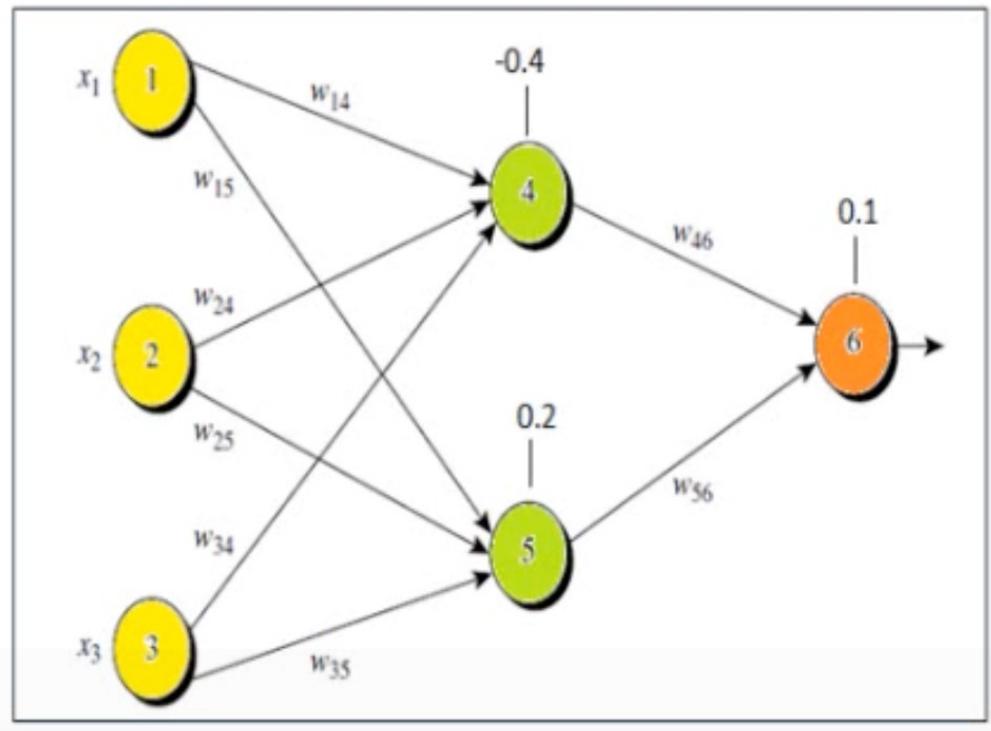
Backpropagation



روشی است برای کاهش گرادیان، جهت حداقل کردن کل مربعات خطای خروجی

مثال

فرض کنید برای ورودی $(1, 0, 1)$ مقدار هدف برابر ۱ است.



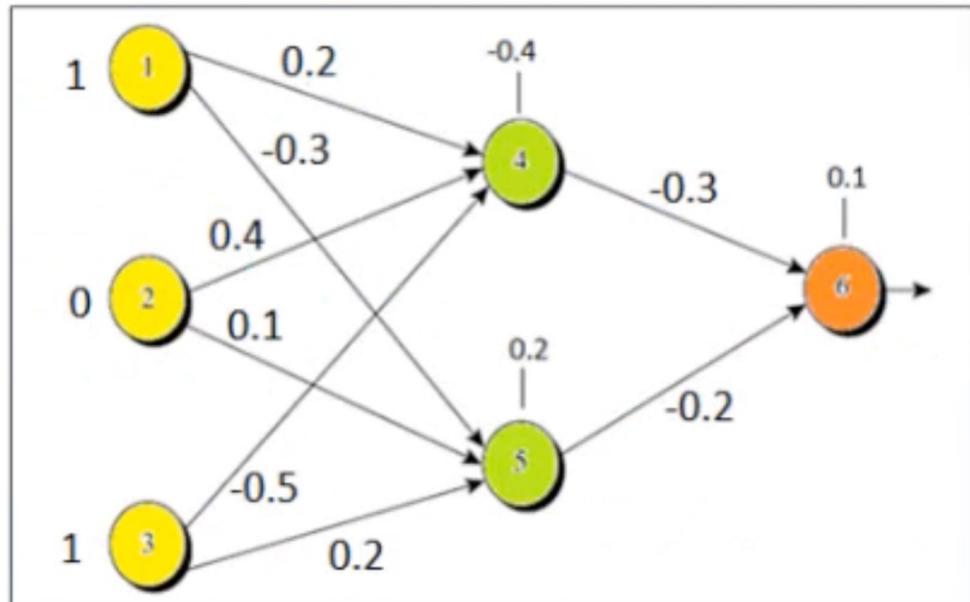
w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{56}
0.2	-0.3	0.4	0.1	-0.5	0.2	-0.3	-0.2

$$y = \frac{1}{1+e^{-x}}$$

$$y' = y(1-y)$$

تابع تحریک: سیگموئید

Feed Forward (Forward pass)



$$0.2 + 0 - 0.5 - 0.4 = -0.7$$

$$1 / (1 + e^{0.7}) = 0.332$$

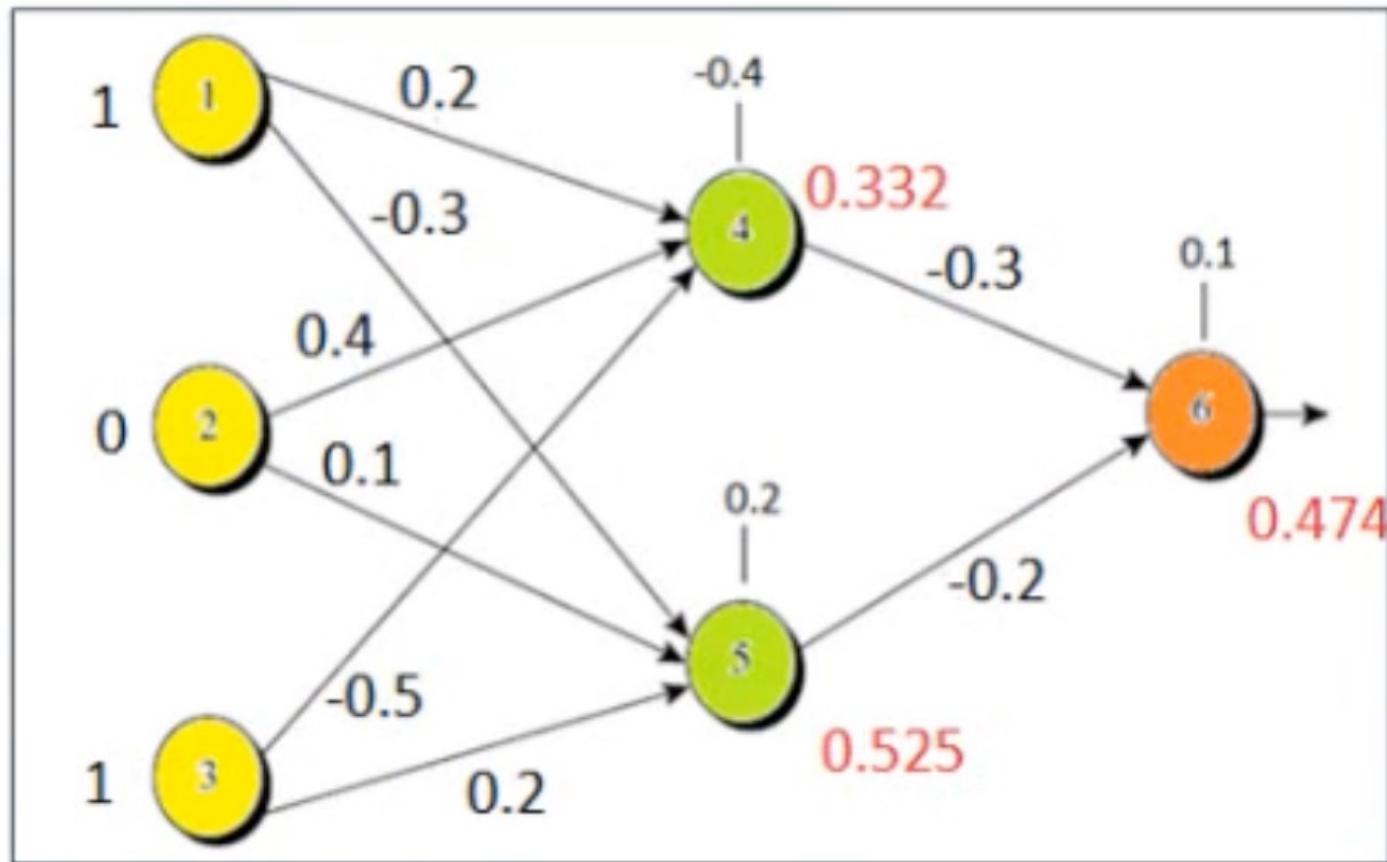
$$-0.3 + 0 + 0.2 + 0.2 = 0.1$$

$$1 / (1 + e^{-0.1}) = 0.525$$

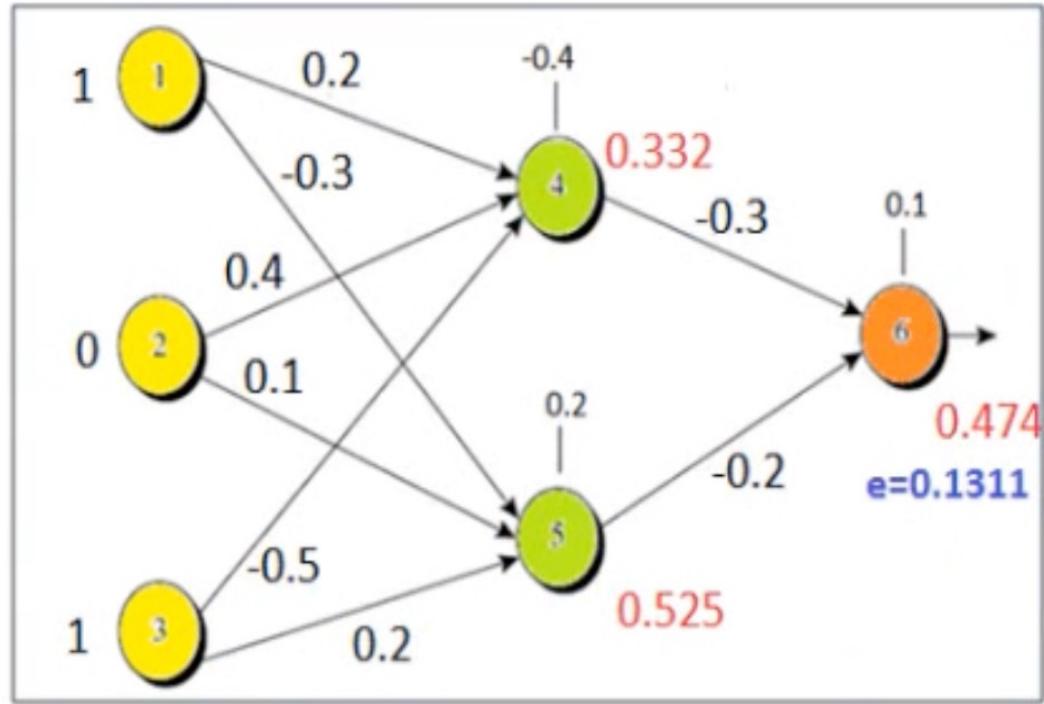
$$(-0.3)(0.332) - (0.2)(0.525) + 0.1 = -0.105$$

$$1 / (1 + e^{0.105}) = 0.474$$

Feed forward



Back propagation (Backward pass)



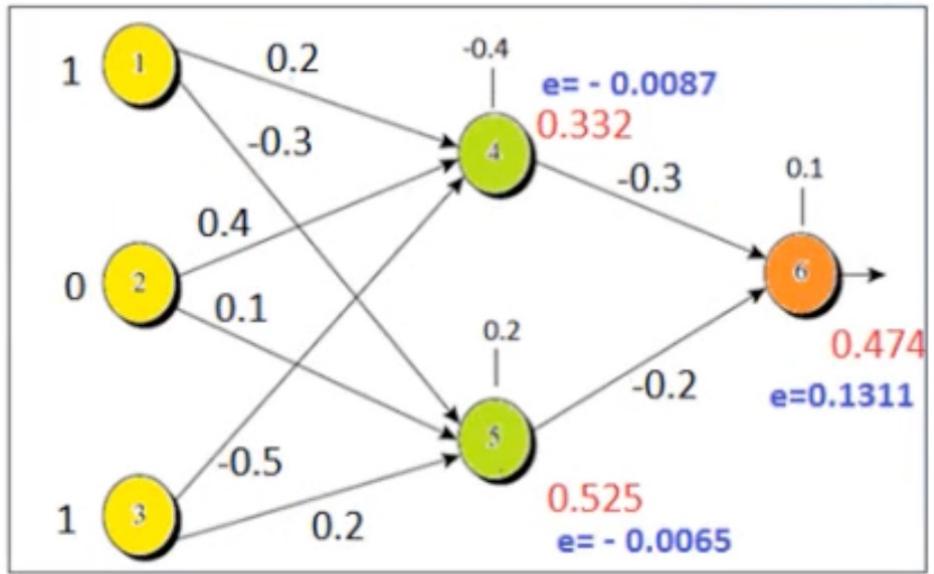
$$e_y = (1 - 0.474) \times (0.474) \times (1 - 0.474) = 0.1311$$

محاسبه خطای نورون خروجی :

$$e_y = (t - y)f'(y)$$

$$e_y = (t - y)y(1 - y)$$

Back propagation (Backward pass)



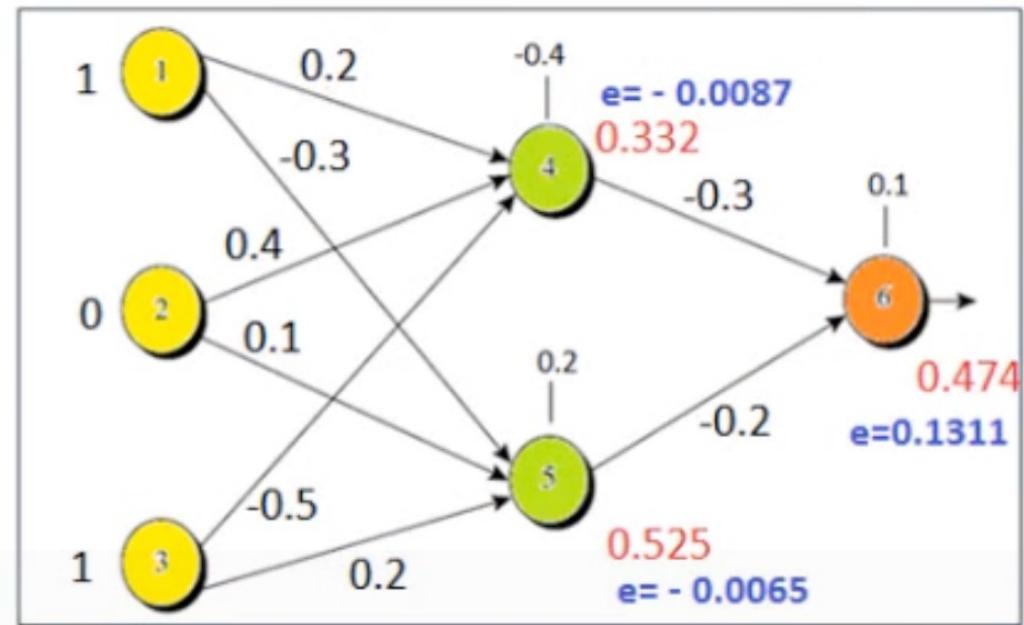
محاسبه خطای نورون‌های مخفی :

$$e_j = (e_y \cdot w_j) h_j (1 - h_j)$$

$$e_5 = ((0.1311)(-0.2)) \times (0.525) \times (1 - 0.525) = -0.0065$$

$$e_4 = ((0.1311)(-0.3)) \times (0.332) \times (1 - 0.332) = -0.0087$$

Calculating new weights



نحوه یادگیری

خروجی نرون i

خطای نرون j

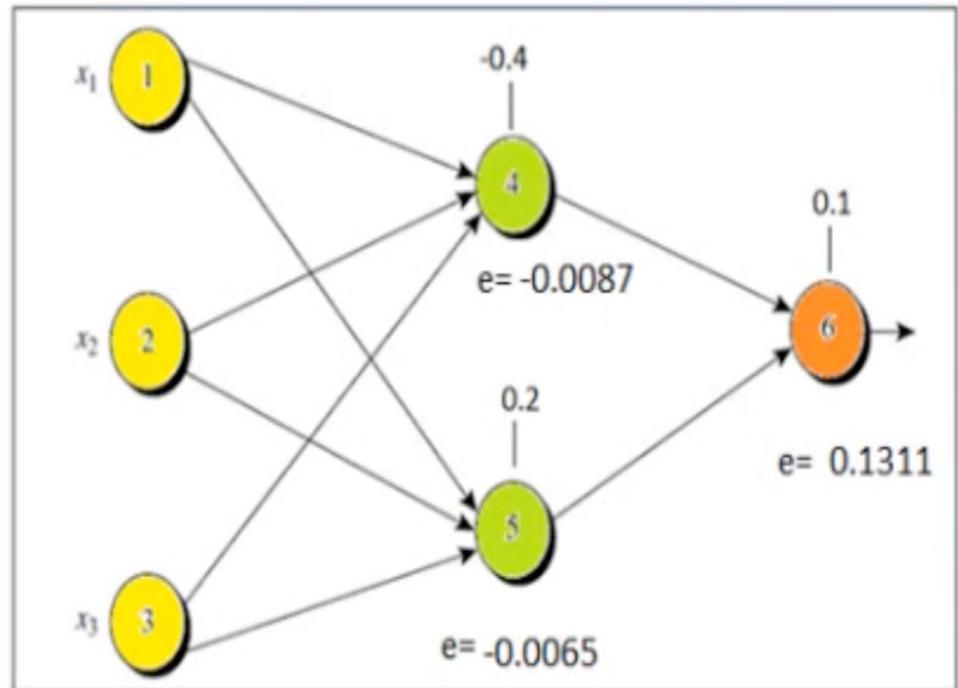
قدیم

جديد

$$w_{ij} \leftarrow w_{ij} + \alpha * e_j * o_i$$

Weight	New Value
w_{46}	$-0.3 + (0.9)(0.1311)(0.332) = -0.261$
w_{56}	$-0.2 + (0.9)(0.1311)(0.525) = -0.138$
w_{14}	$0.2 + (0.9)(-0.0087)(1) = 0.192$
w_{15}	$-0.3 + (0.9)(-0.0065)(1) = -0.306$
w_{24}	$0.4 + (0.9)(-0.0087)(0) = 0.4$
w_{25}	$0.1 + (0.9)(-0.0065)(0) = 0.1$
w_{34}	$-0.5 + (0.9)(-0.0087)(1) = -0.508$
w_{35}	$0.2 + (0.9)(-0.0065)(1) = 0.194$

Calculating new biases



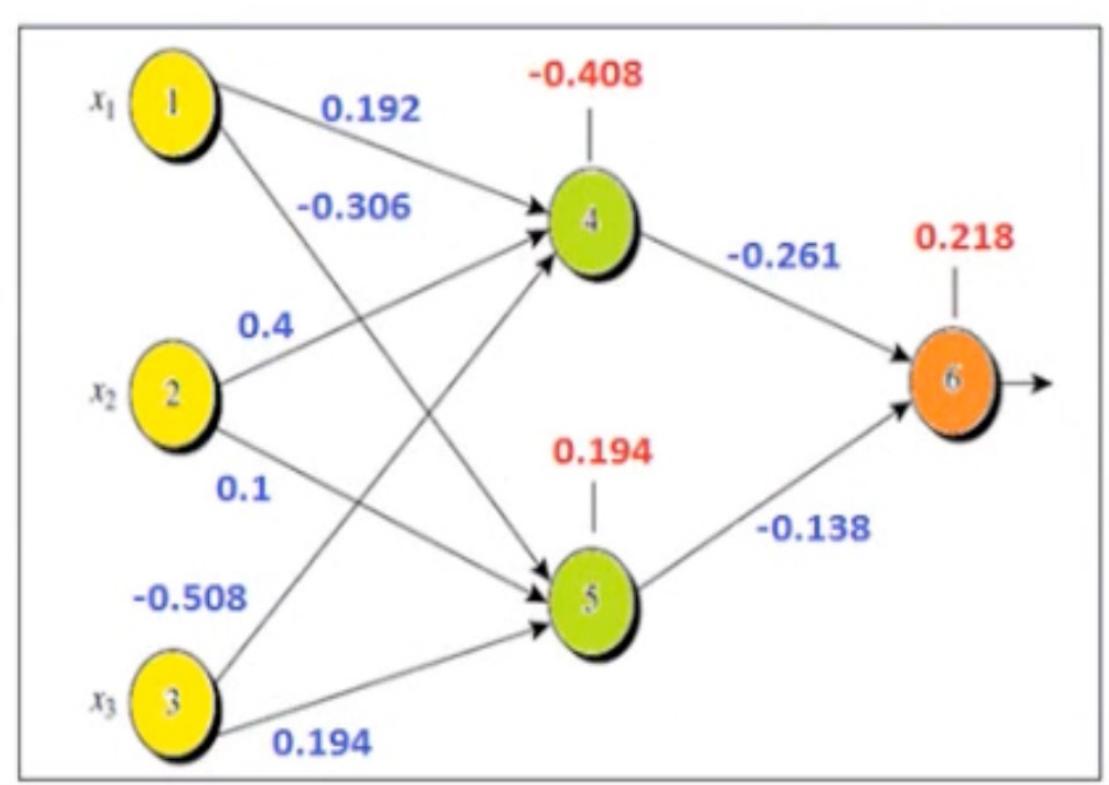
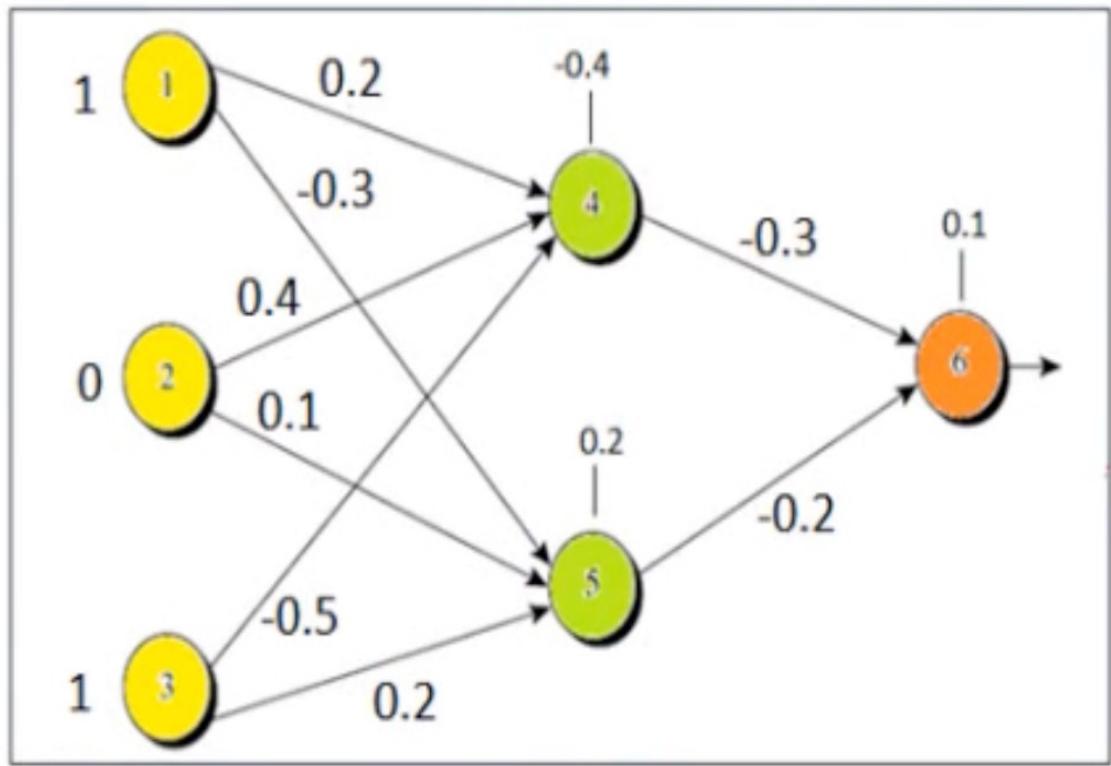
قديم \downarrow
جديد $\rightarrow b_i \leftarrow b_i + \alpha * e_i$

$$b_4 = -0.4 + (0.9) \times (-0.0087) = -0.408$$

$$b_5 = 0.2 + (0.9) \times (-0.0065) = 0.194$$

$$b_6 = 0.1 + (0.9) \times (0.1311) = 0.218$$

New MLP



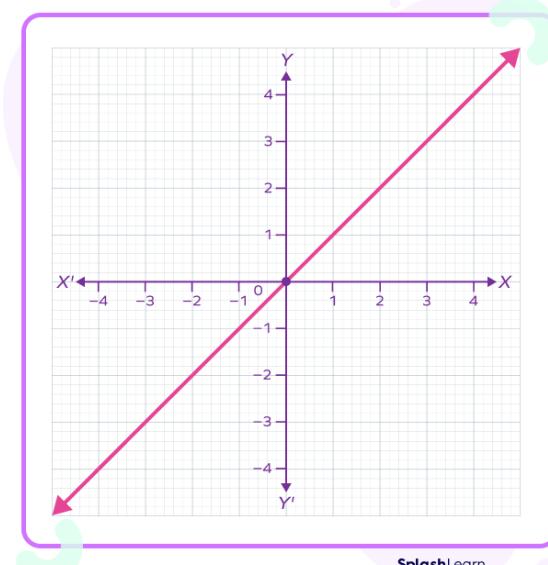
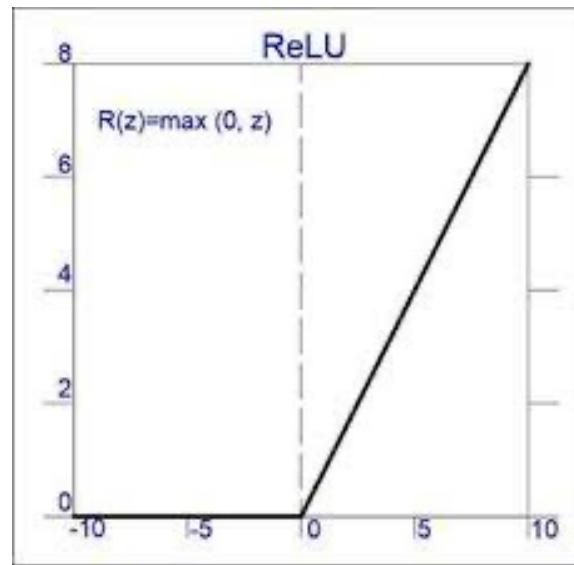
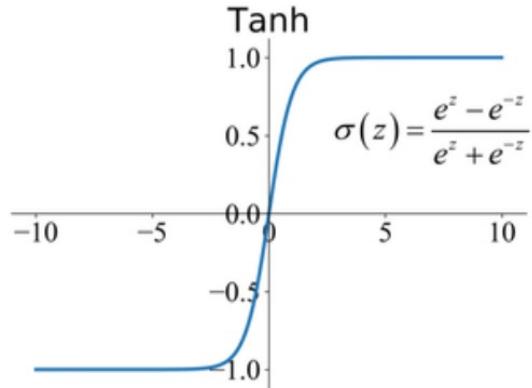
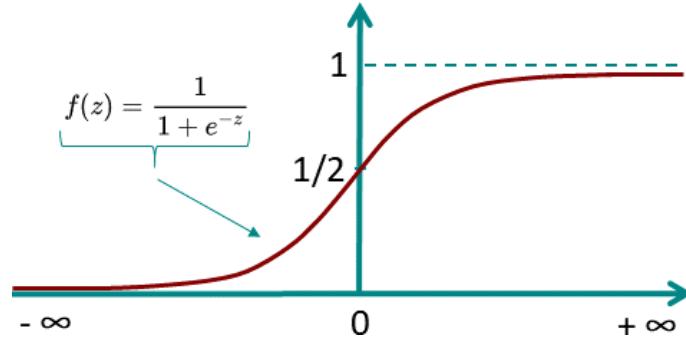
sklearn.neural_network

```
from sklearn.neural_network import MLPClassifier
```

```
model = MLPClassifier(hidden_layer_sizes=(50, 20, 30))
```

activation : {‘identity’, ‘logistic’, ‘tanh’, ‘relu’}, default=‘relu’

Activation Function



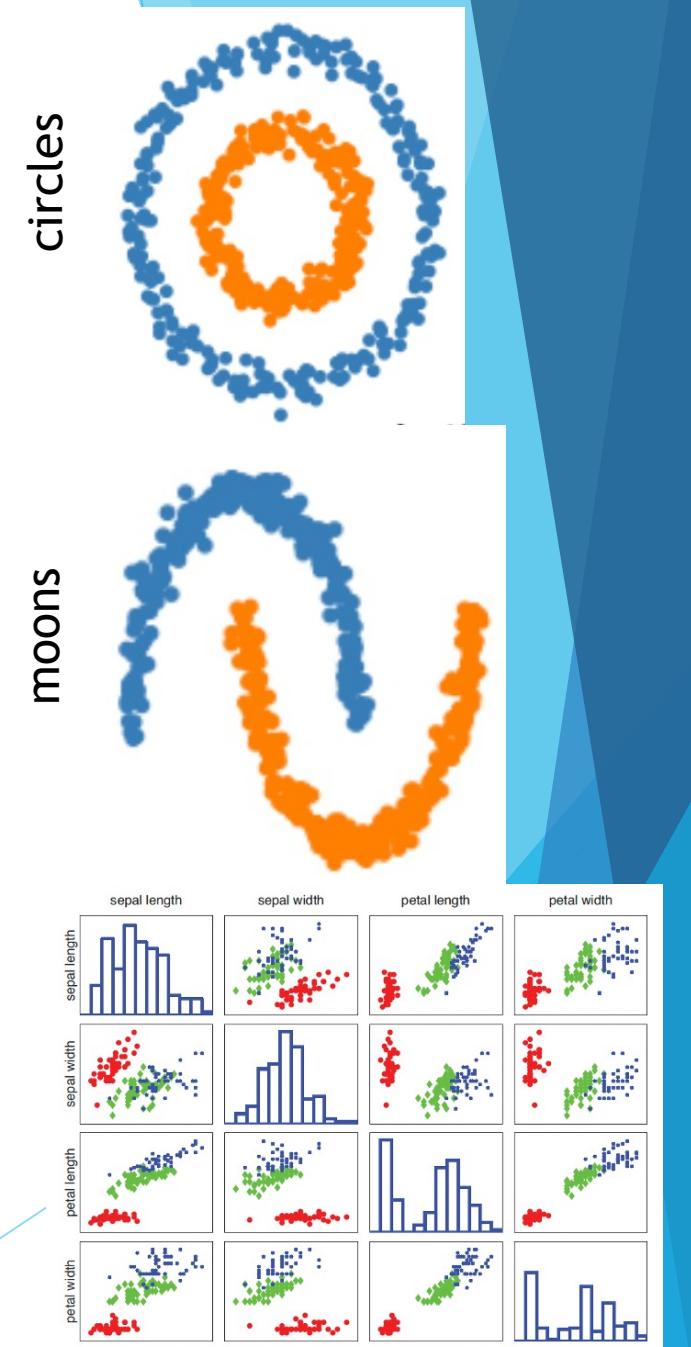
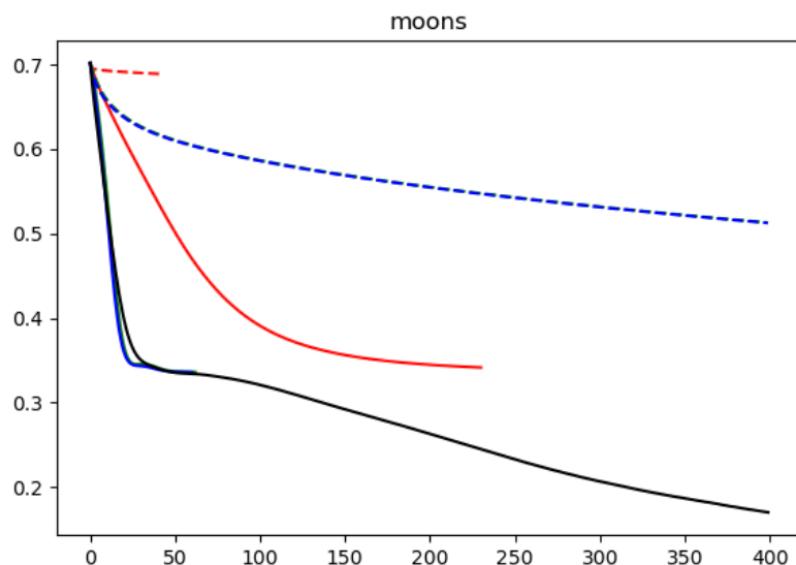
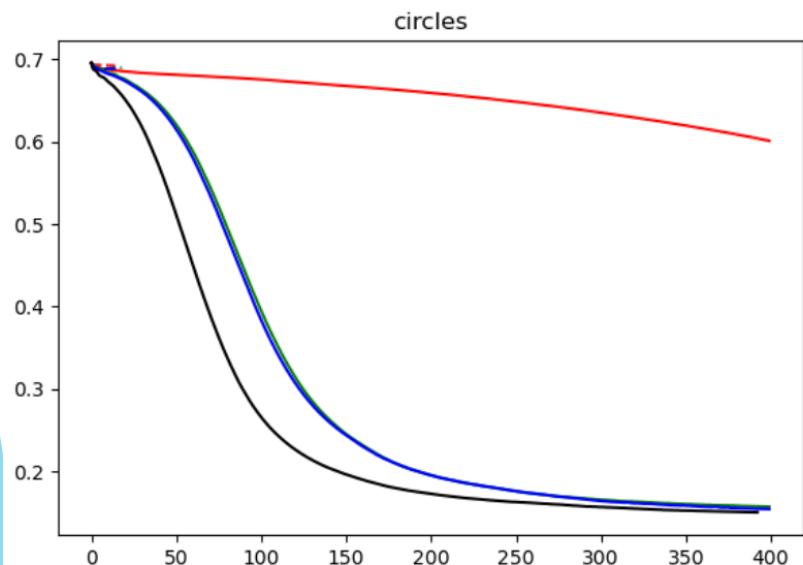
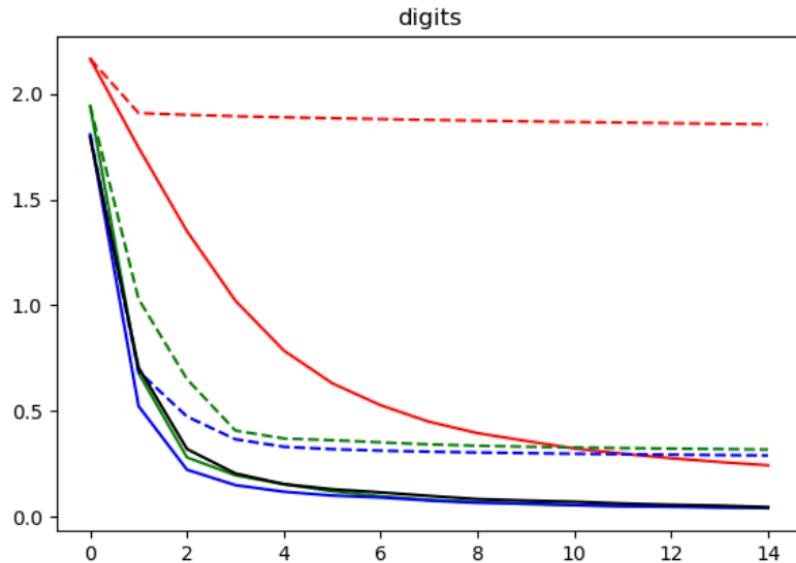
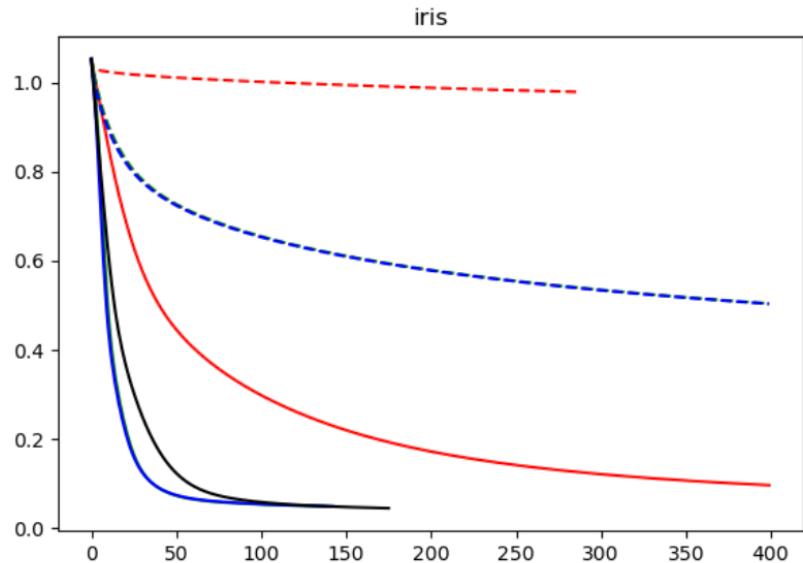
weight optimization (Solver)

solver : {'lbfgs', 'sgd', 'adam'}, default='adam'

The solver for weight optimization.

- 'lbfgs' is an optimizer in the family of quasi-Newton methods.
- 'sgd' refers to stochastic gradient descent.
- 'adam' refers to a stochastic gradient-based optimizer proposed by Kingma, Diederik, and Jimmy Ba

constant learning-rate	inv-scaling learning-rate	inv-scaling with Nesterov's momentum
constant with momentum	inv-scaling with momentum	adam
constant with Nesterov's momentum		



Stoping Criteria

1) Number of iteration

max_iter : *int, default=200*

2) Until convergence

n_iter_no_change : *int, default=10*

learning_rate : {'constant', 'invscaling', 'adaptive'}, default='constant'

Learning rate schedule for weight updates.

- 'constant' is a constant learning rate given by 'learning_rate_init'.
- 'invscaling' gradually decreases the learning rate at each time step 't' using an inverse scaling exponent of 'power_t'. $\text{effective_learning_rate} = \text{learning_rate_init} / \text{pow}(t, \text{power_t})$
- 'adaptive' keeps the learning rate constant to 'learning_rate_init' as long as training loss keeps decreasing. Each time two consecutive epochs fail to decrease training loss by at least tol, or fail to increase validation score by at least tol if 'early_stopping' is on, the current learning rate is divided by 5.

Only used when `solver='sgd'`.

momentum : float, default=0.9

Momentum for gradient descent update. Should be between 0 and 1. Only used when `solver='sgd'`.

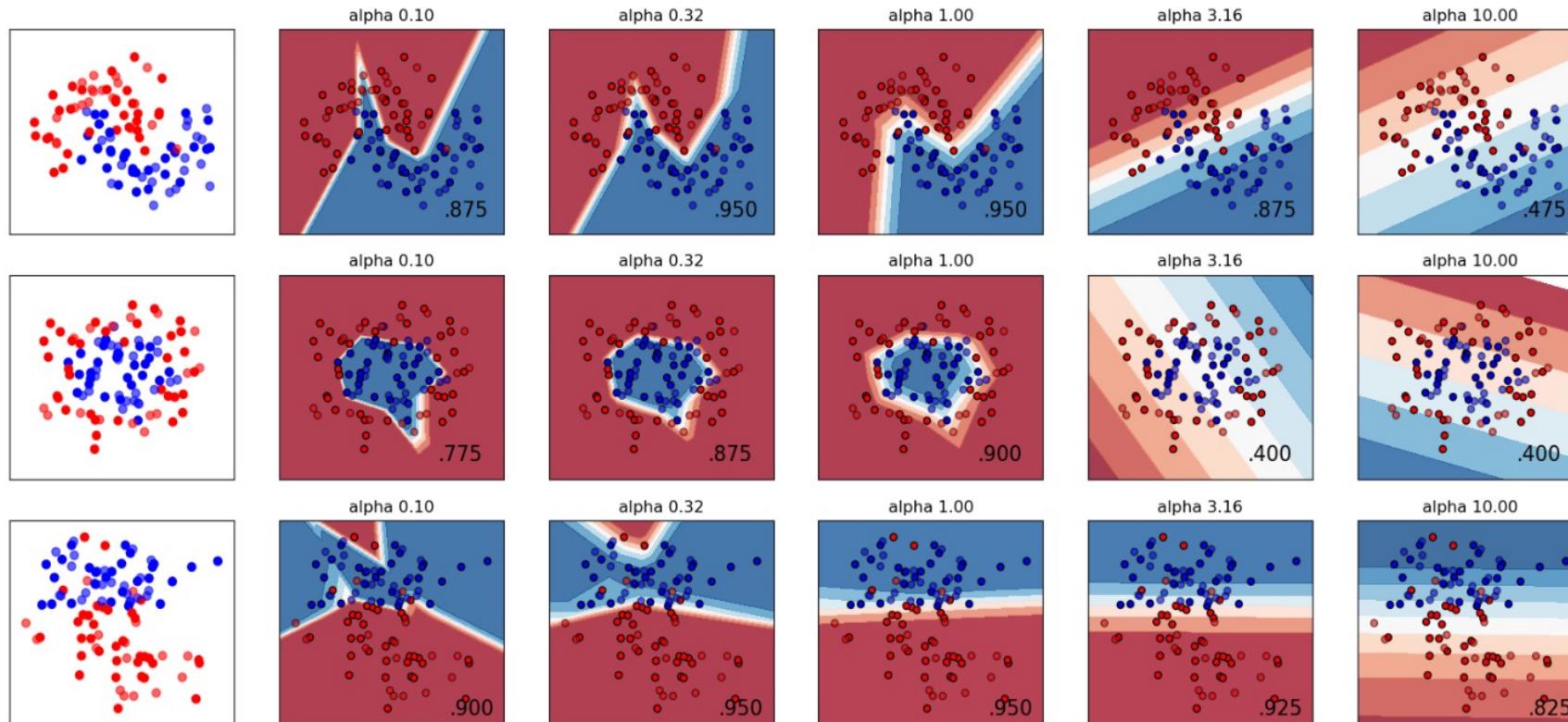
nesterovs_momentum : bool, default=True

Whether to use Nesterov's momentum. Only used when `solver='sgd'` and `momentum > 0`.

Regularization term

alpha : float, default=0.0001

Strength of the L2 regularization term. The L2 regularization term is divided by the sample size when added to the loss.



verbose : bool, default=False

Whether to print progress messages to stdout.

```
model = MLPClassifier(hidden_layer_sizes= 10, activation='relu',solver = 'adam',verbose= True)

model.fit(X_train,y_train)

Iteration 1, loss = 0.76499540
Iteration 2, loss = 0.75486773
Iteration 3, loss = 0.74480880
Iteration 4, loss = 0.73482088
Iteration 5, loss = 0.72490837
Iteration 6, loss = 0.71507200
Iteration 7, loss = 0.70531408
Iteration 8, loss = 0.69563704
Iteration 9, loss = 0.68604332
Iteration 10, loss = 0.67653537
Iteration 11, loss = 0.66711568
Iteration 12, loss = 0.65778678
Iteration 13, loss = 0.64855118
Iteration 14, loss = 0.63941144
Iteration 15, loss = 0.63037011
Iteration 16, loss = 0.62142978
Iteration 17, loss = 0.61259303
Iteration 18, loss = 0.60386247
Iteration 19, loss = 0.59524071
Iteration 20, loss = 0.58673037
Iteration 21, loss = 0.57833409
Iteration 22, loss = 0.57005450
Iteration 23, loss = 0.56189426
Iteration 24, loss = 0.55385601
Iteration 25, loss = 0.54594240
Iteration 26, loss = 0.53815610
Iteration 27, loss = 0.53049976
Iteration 28, loss = 0.52207603
```

Epoch vs Iteration

Epoch:

An epoch is one complete pass through the entire training dataset.

Batch:

A batch is a subset of the training dataset used in one iteration of the optimization algorithm.

Iteration:

An iteration is one update of the model's weights, typically after processing one batch.

Epoch vs Iteration

- ✓ In each epoch, you go through several batches, and in each batch, you perform multiple iterations.
- ✓ The number of iterations in an epoch depends on the size of your dataset and the batch size. For example, if you have 1000 examples and use a batch size of 100, you'd have 10 iterations per epoch.

Key Considerations:

- ✓ **Too Many Epochs:** Training for too many epochs can lead to overfitting, where the model memorizes the training data instead of learning general patterns.
- ✓ **Batch Size:** Smaller batch sizes introduce more noise but allow for more frequent updates. Larger batch sizes provide more stable updates but might require more memory.
- ✓ **Computational Efficiency:** Batching and iterations help make the training process computationally efficient, especially when dealing with large datasets.

MNIST dataset

It is a collection of handwritten digit widely used for training and testing.

It contains 70,000 images of handwritten digits from 0 to 9



MNIST dataset

MNIST is a starter dataset used for machine learning for several reasons:

1. **Benchmarking:** It provides a straightforward dataset to test and benchmark machine learning models, particularly in image recognition algorithms.
2. **Learning Tool:** Due to its simplicity and small size, MNIST is an excellent dataset for beginners to learn the basics of machine learning and pattern recognition.
3. **Research:** It continues to be a reference data set for evaluating new machine learning techniques.

Applications of MNIST

- ▶ **Banking Sector:**

Recognizing Handwritten Numbers on Checks

- ▶ **Postal Services**

Automating Postal Code Reading Document Management

- ▶ **Digitizing**

Written Documents and Recognizing Numbers