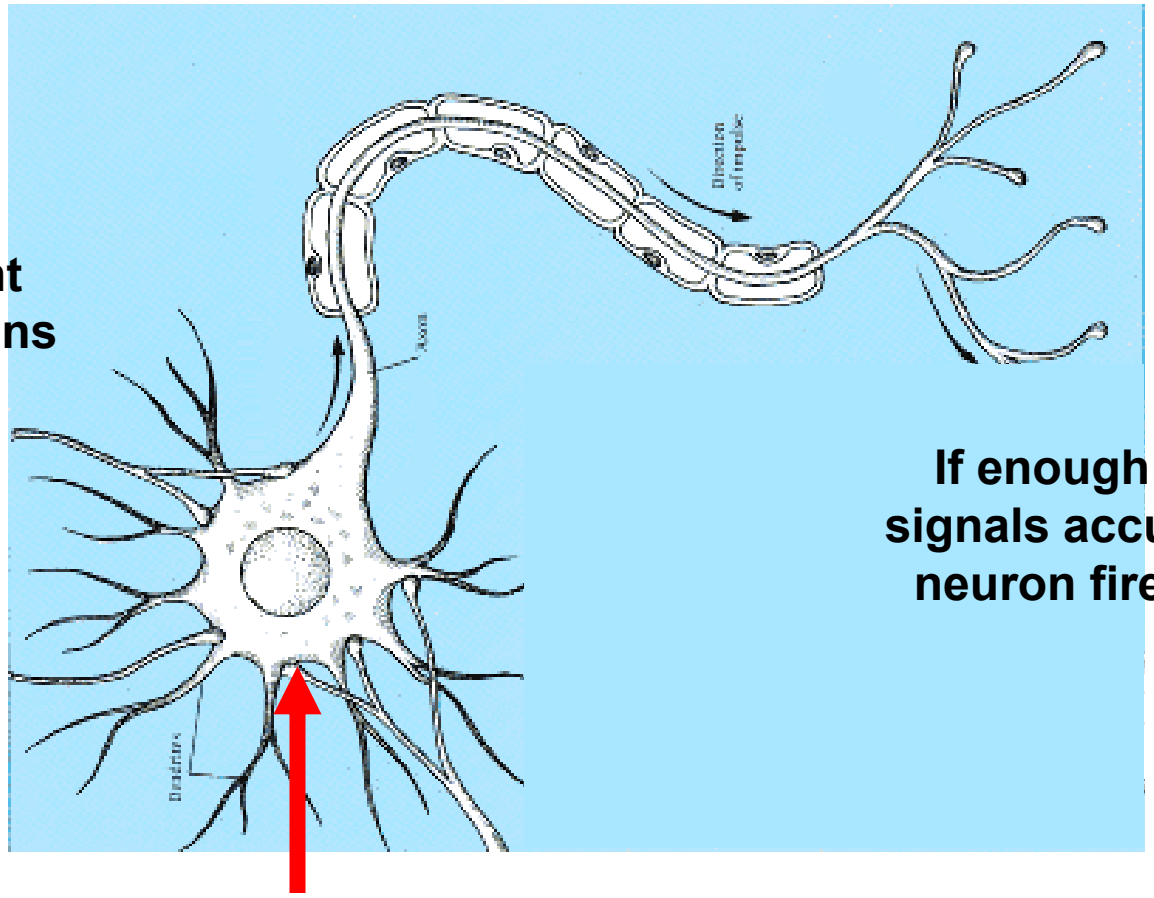
Deep Learning
CS-878

Week-02

Artificial Neural Network - Perceptron

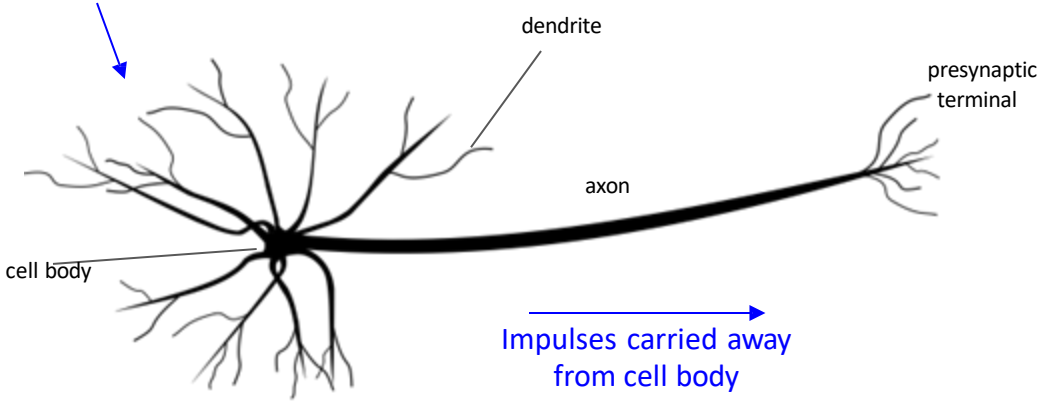
Input signals sent from other neurons



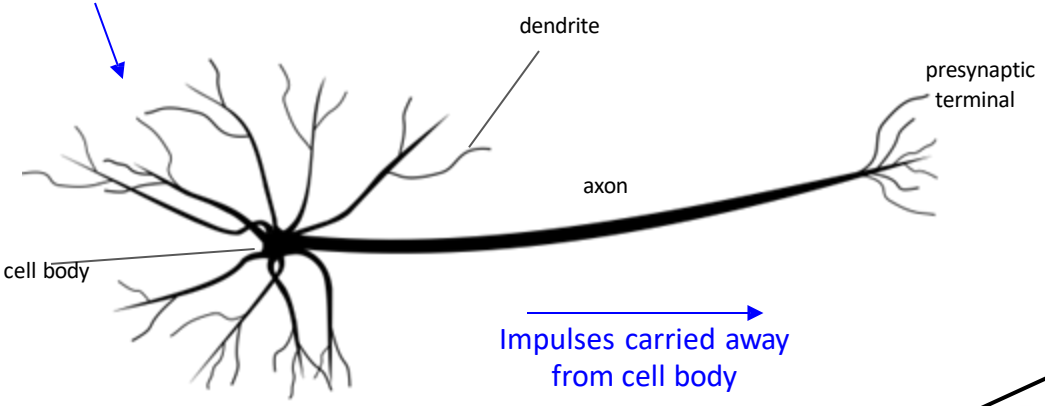
If enough sufficient signals accumulate, the neuron fires a signal.

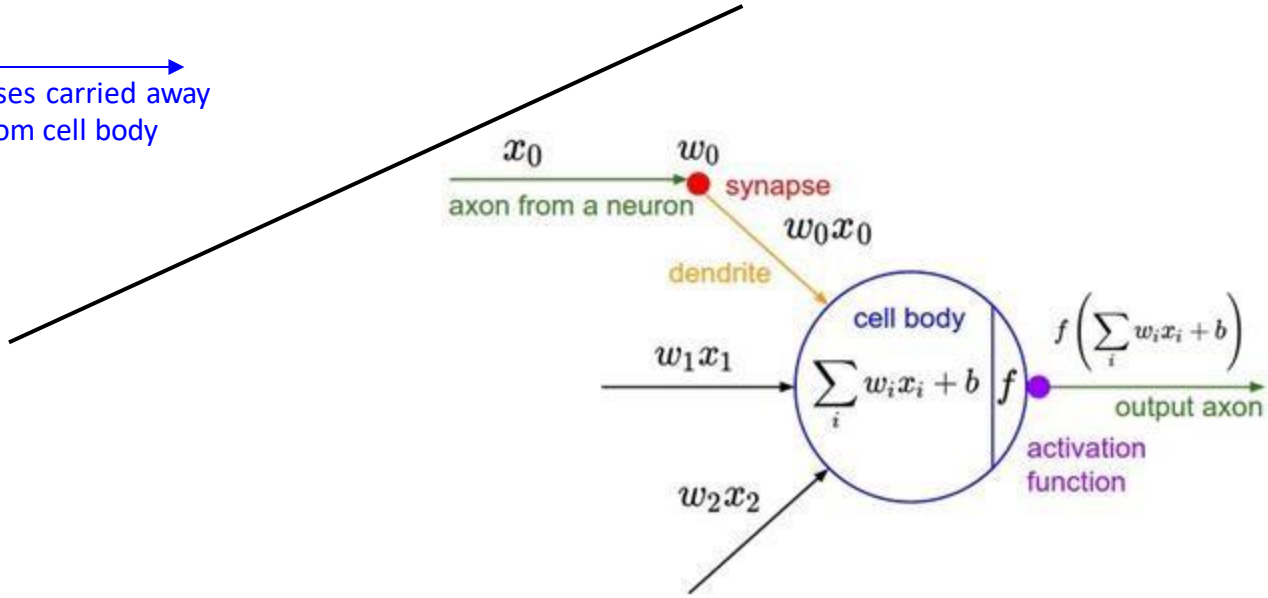
Connection strengths determine how the signals are accumulated

Impulses carried toward cell body

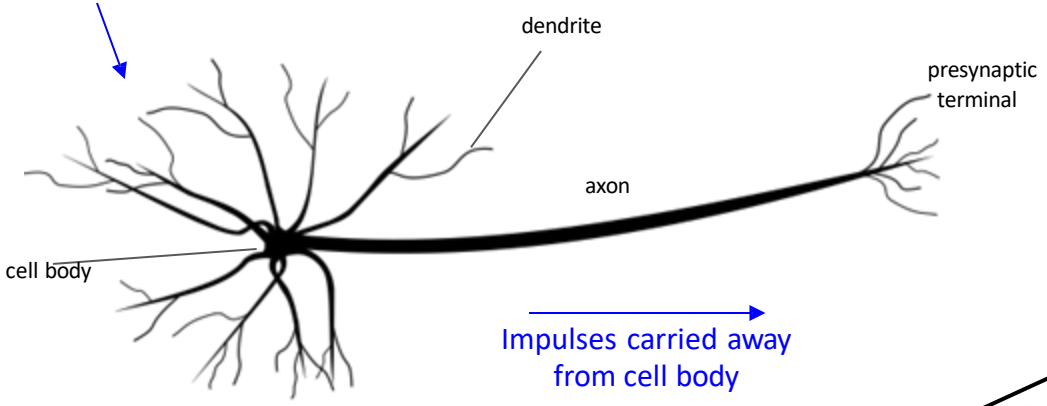


Impulses carried toward cell body

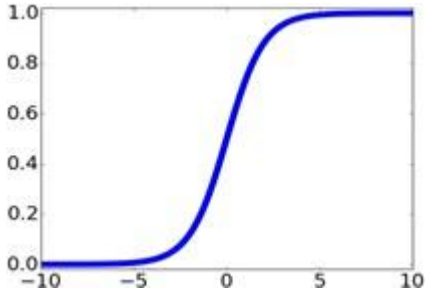




Impulses carried toward cell body

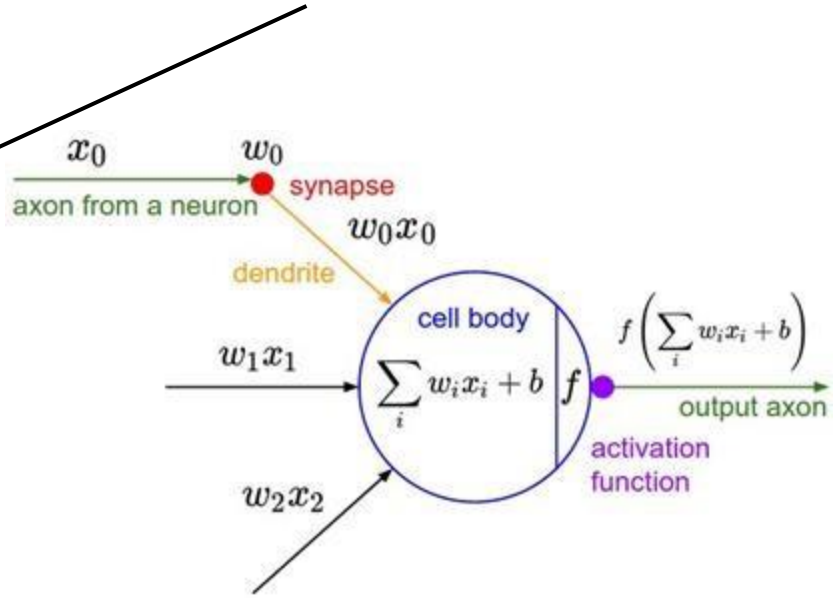


Impulses carried away from cell body

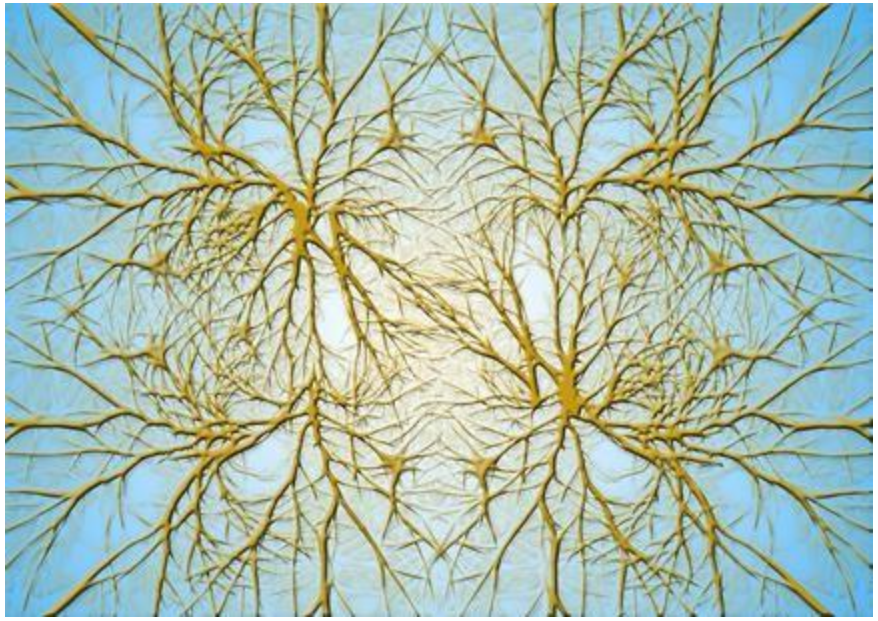


sigmoid activation function

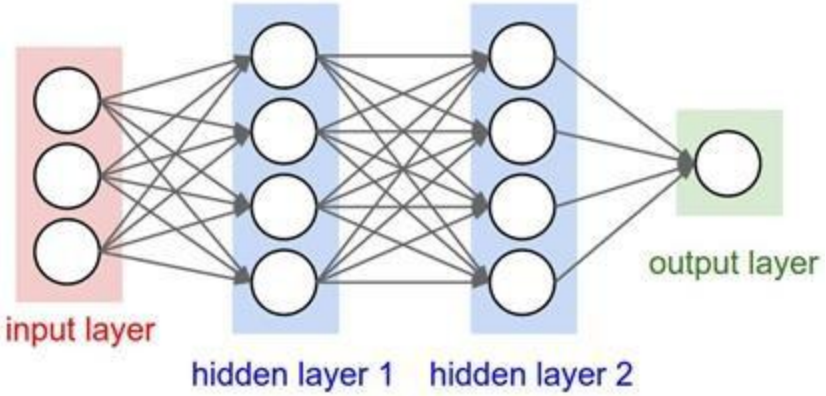
$$\frac{1}{1+e^{-x}}$$



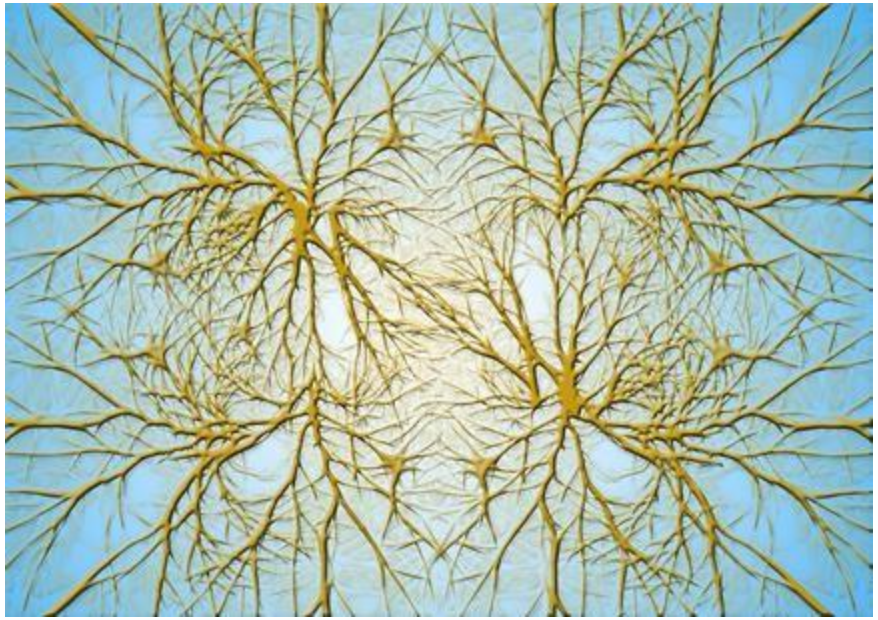
Biological Neurons: Complex connectivity patterns



Neurons in a neural network: Organized into regular layers for computational efficiency

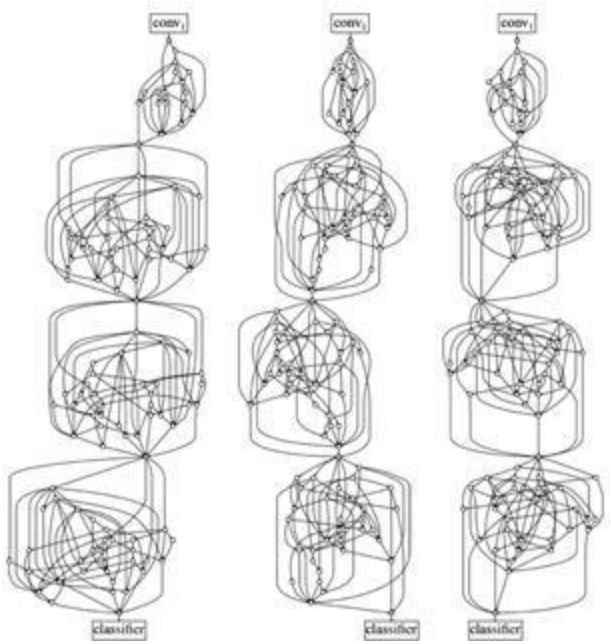


Biological Neurons: Complex connectivity patterns



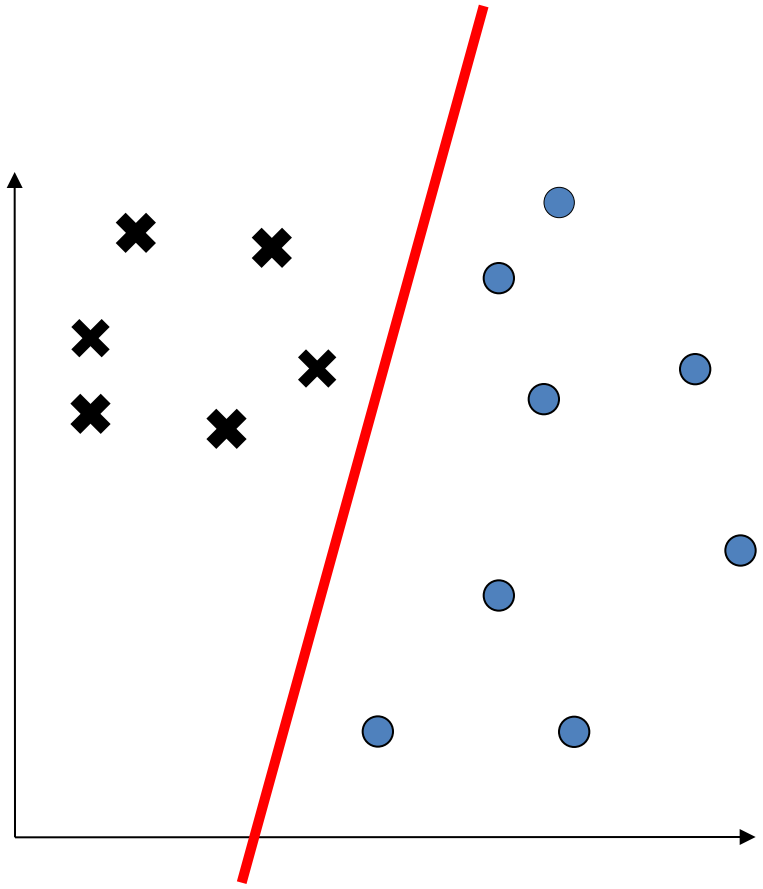
This image is CC0 Public Domain

But neural networks with random connections can work too!



Xie et al, "Exploring Randomly Wired Neural Networks for Image Recognition", IEEE/CVF International Conference on Computer Vision 2019

A (Linear) Decision Boundary



Represented by:

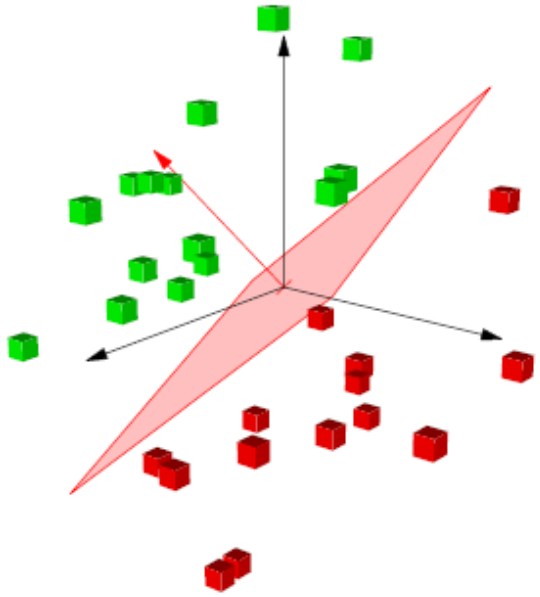
One artificial neuron called a "Perceptron"

Low space complexity

Low time complexity

Perceptron

- The perceptron with a step function performs **classification**
- The perceptron can be 'visualised' as a decision boundary in input space



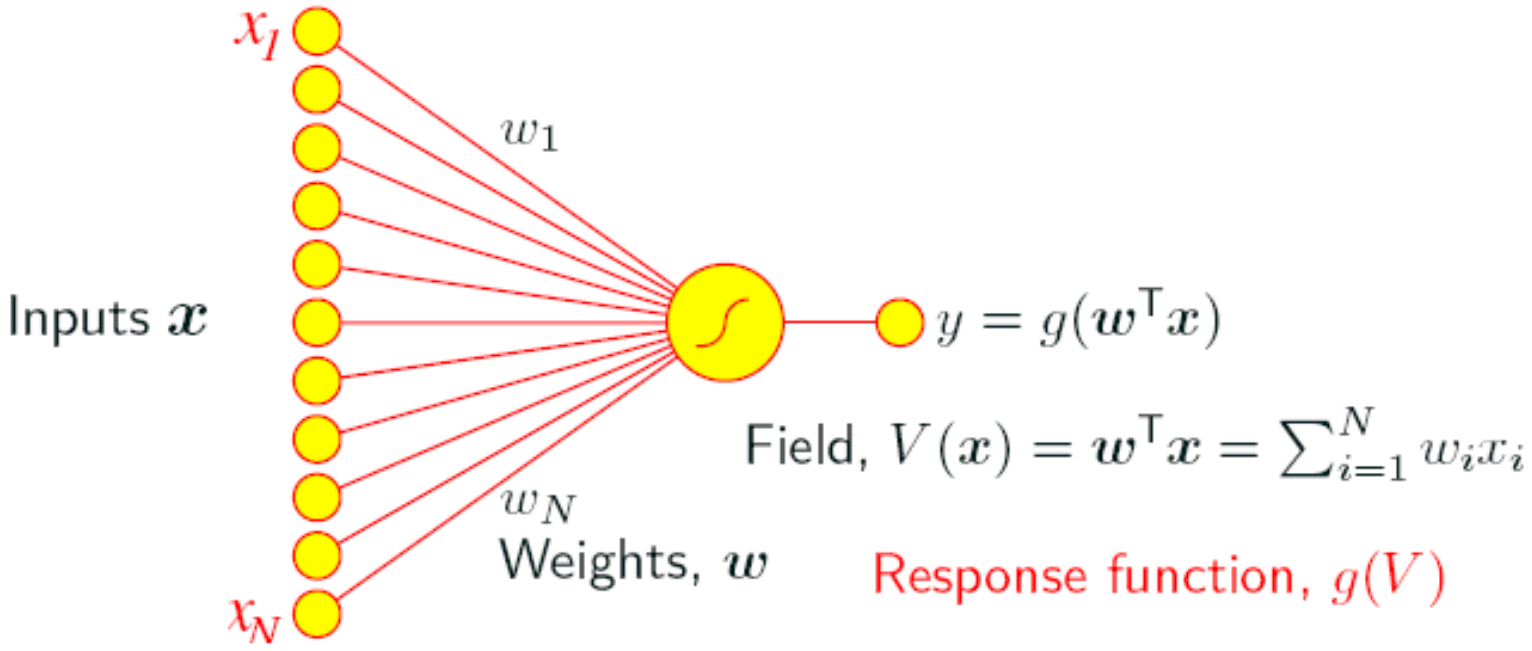
- The perceptron can only separate linear-separable inputs

Perceptron

- Given (numeric) input features $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$
- Prediction given by $f(\boldsymbol{x}; \boldsymbol{w})$
- $\boldsymbol{w}$ are parameters or "weights" that we train
- The **perceptron** provides the classic example of a **parametric** learning algorithm

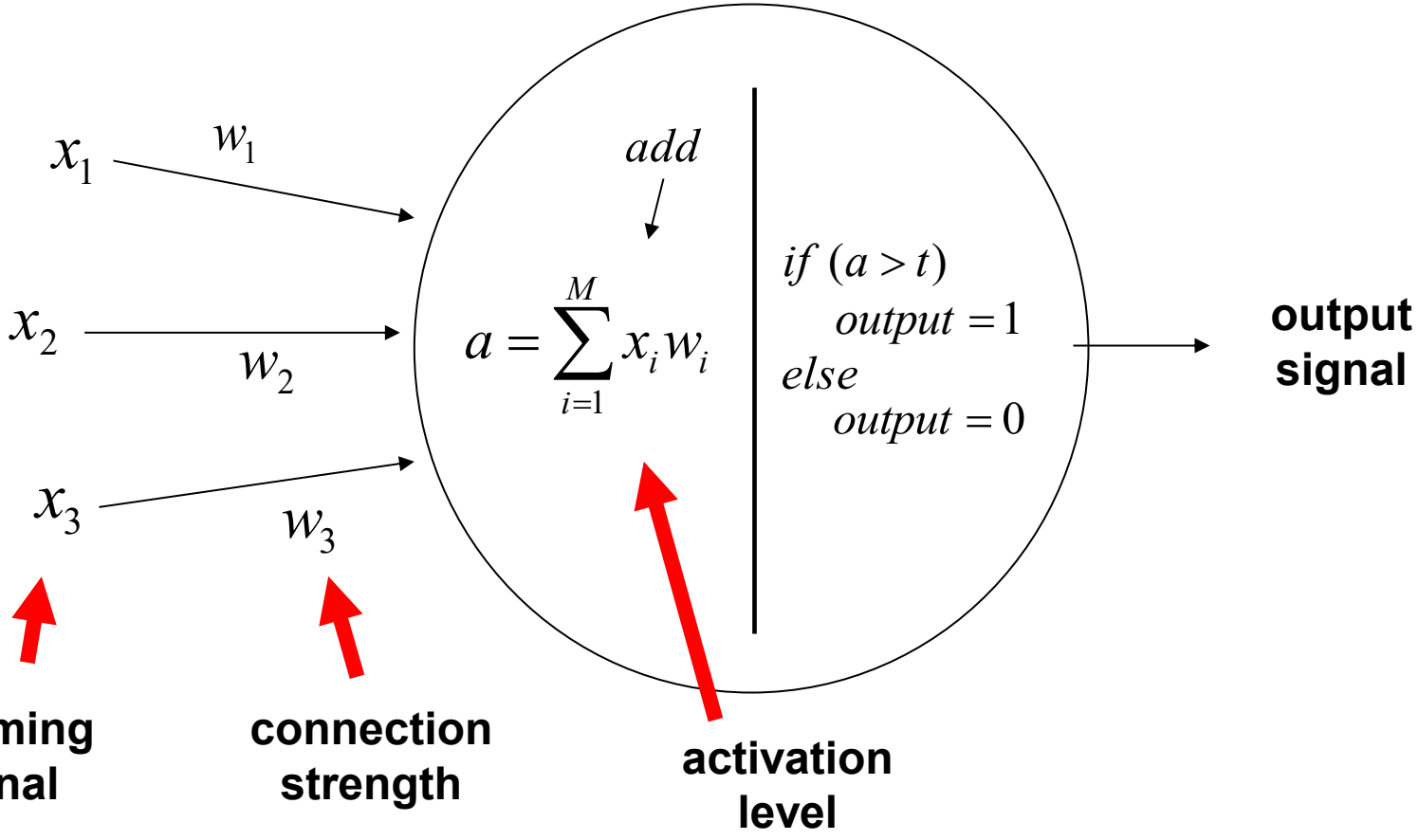
Perceptron

- Proposed by Frank Rosenblatt (1958) (Widrow and Hoff proposed **adaline** at same time)
- Schematic representation



Perceptron

- input signals ‘x’ and weights ‘w’ are multiplied
- weights correspond to connection strengths
- signals are added up – if they are enough, FIRE!



Calculation…

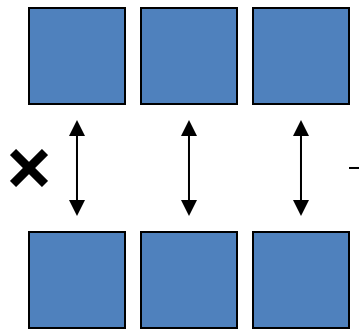
$$a = \sum_{i=1}^{M} x_i w_i$$

Sum notation

(just like a loop from 1 to M)

`double[] x =`

`double[] w =`



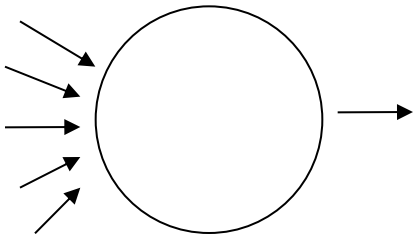
Multiple corresponding elements and add them up

(activation)

if (activation > threshold) FIRE !

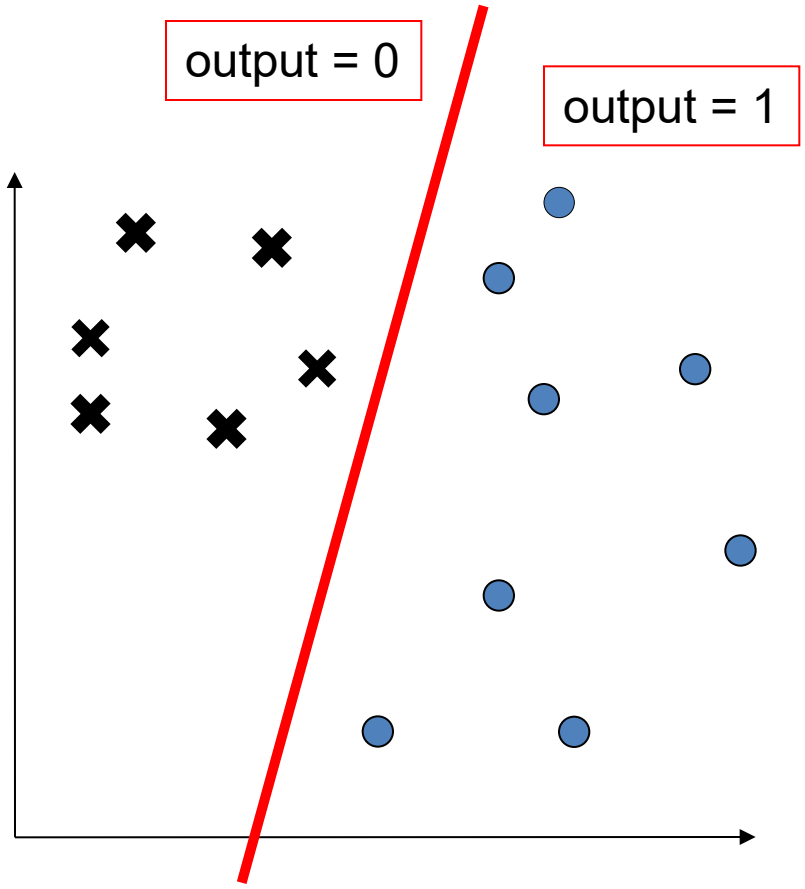
Perceptron Decision Rule

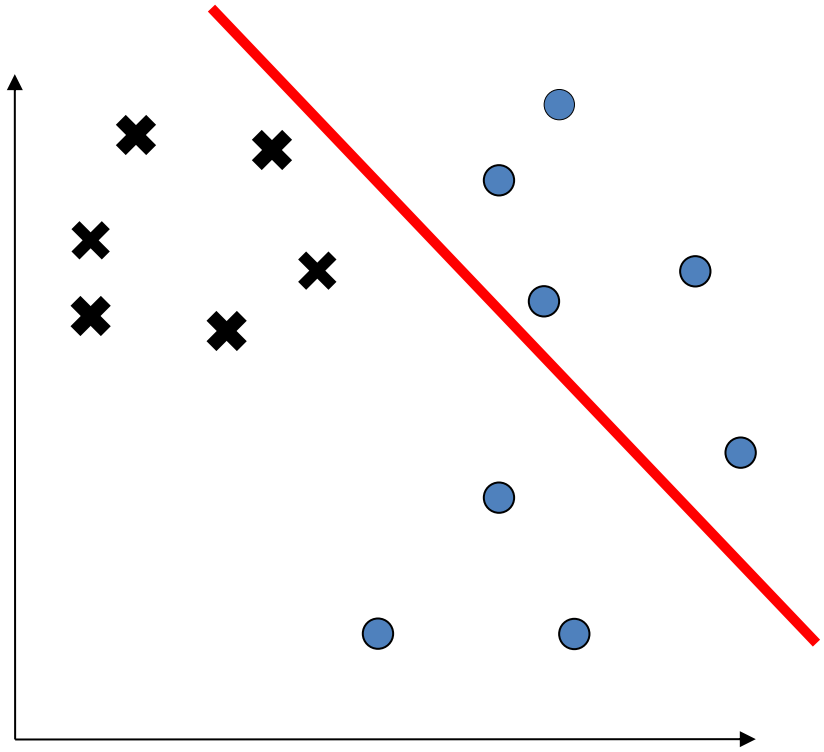
$$\textit{if} \quad \left(\sum_{i=1}^{M} x_i w_i \right) \quad > \quad t \qquad \text{then } output = 1, \text{ else } output = 0$$



$$\textit{if} \quad \left(\sum_{i=1}^{M} x_i w_i \right) > \quad t$$

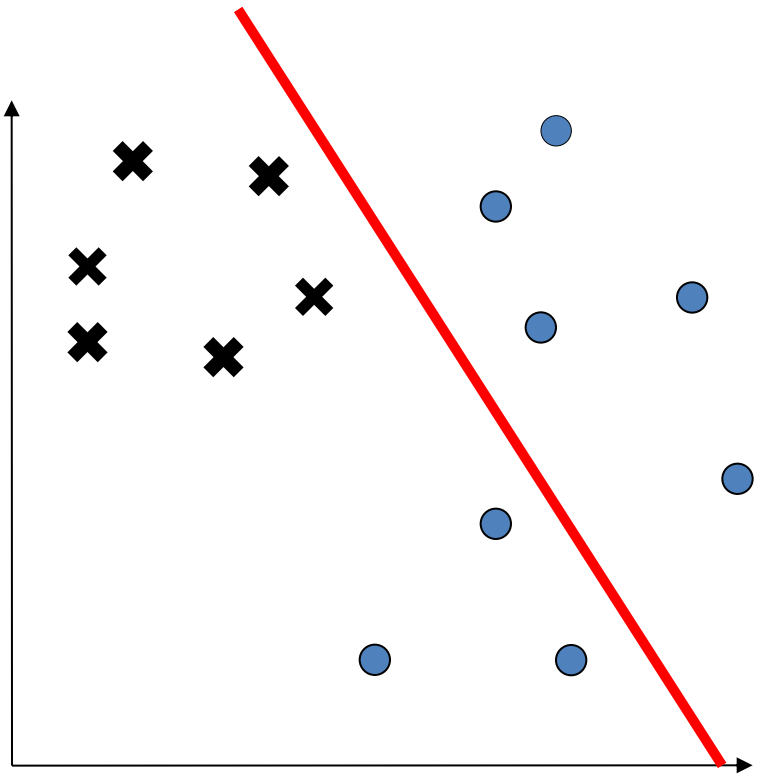
$$\text{then } output = 1, \text{ else } output = 0$$





Is this a good decision boundary?

$$\textit{if} \quad \left(\sum_{i=1}^{M} x_i w_i \right) \quad > \quad t \qquad \text{then } output = 1, \text{ else } output = 0$$

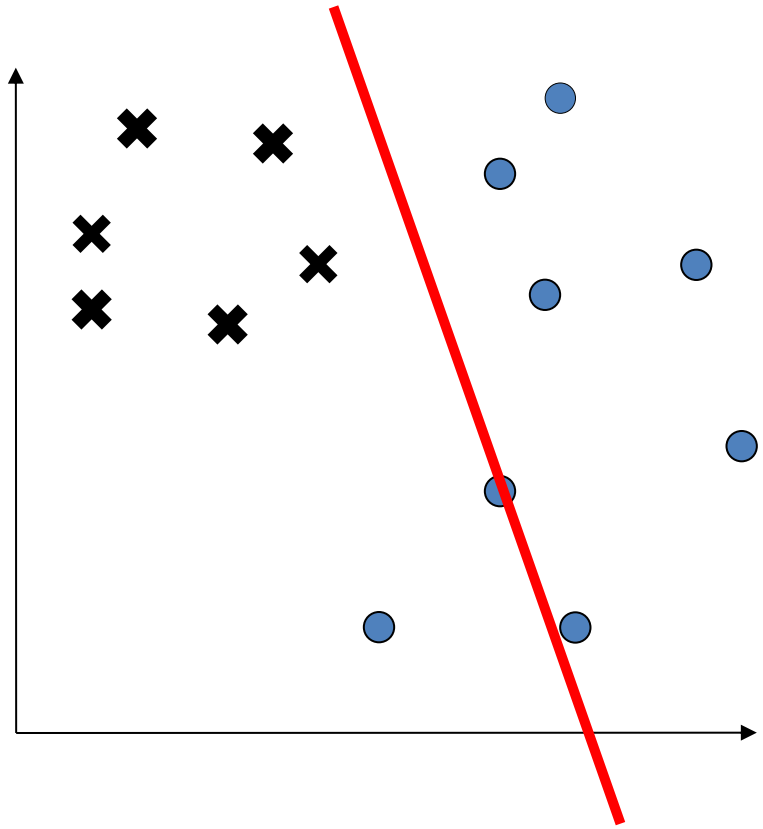


$w_1 = 1.0$

$w_2 = 0.2$

$t = 0.05$

$$\textit{if} \quad \left(\sum_{i=1}^{M} x_i w_i \right) \quad > \quad t \qquad \text{then } output = 1, \text{ else } output = 0$$

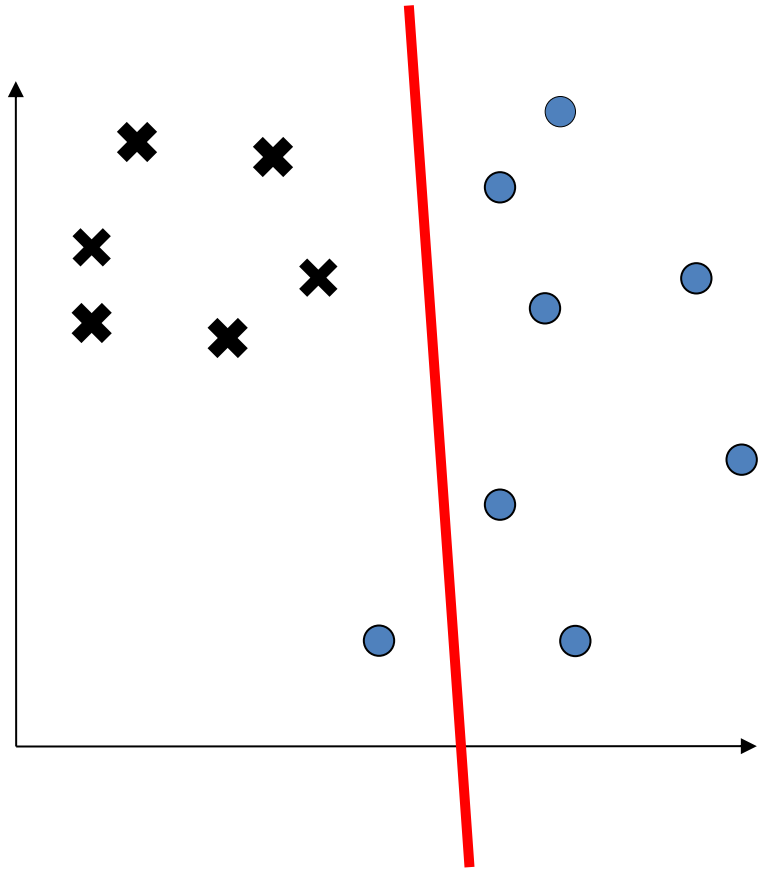


$w_1 = 2.1$

$w_2 = 0.2$

$t = 0.05$

$$\text{if} \quad \left(\sum_{i=1}^{M} x_i w_i \right) \quad > \quad t \qquad \text{then } output = 1, \text{ else } output = 0$$

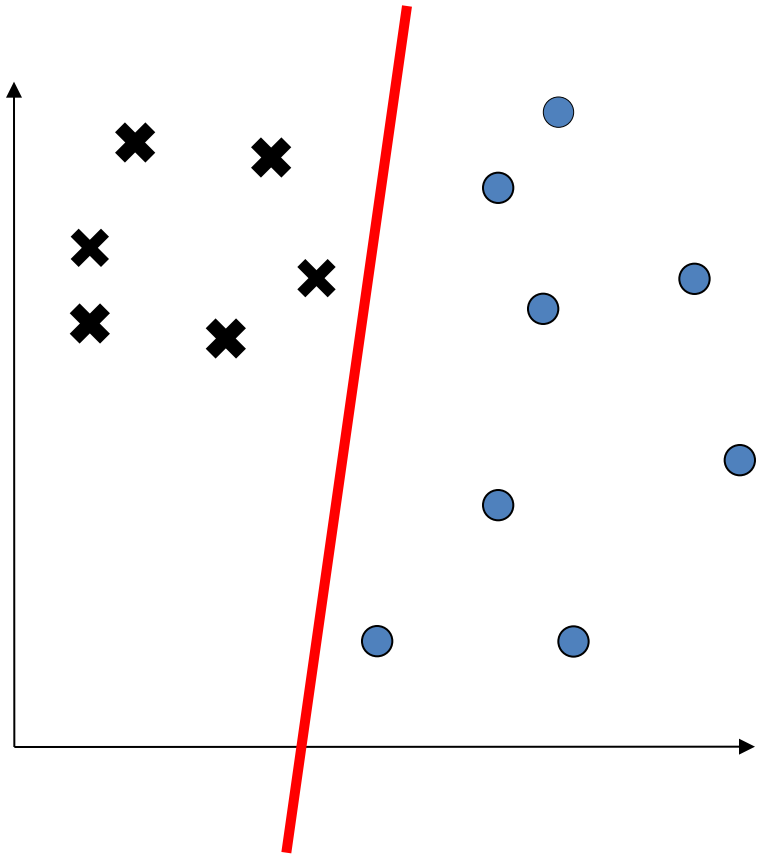


$w_1 = 1.9$

$w_2 = 0.02$

$t = 0.05$

$$\textit{if} \quad \left(\sum_{i=1}^{M} x_i w_i \right) \quad > \quad t \qquad \text{then } output = 1, \text{ else } output = 0$$



$w_1 = -0.8$

$w_2 = 0.03$

$t = 0.05$

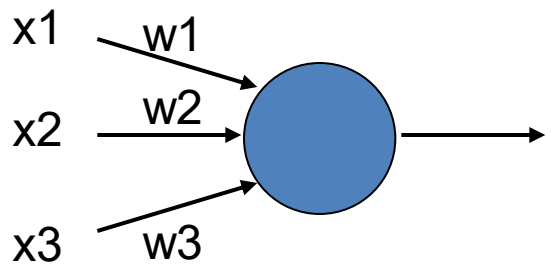
Changing the weights/threshold makes the decision boundary move.

$$x = [\ 1.0,\ 0.5,\ 2.0\ ]$$

$$w = [\ 0.2,\ 0.5,\ 0.5\ ]$$

$$t = 1.0$$

$$a = \sum_{i=1}^{M} x_i w_i$$



Q1. What is the activation, ***a***, of the neuron?

Q2. Does the neuron fire?

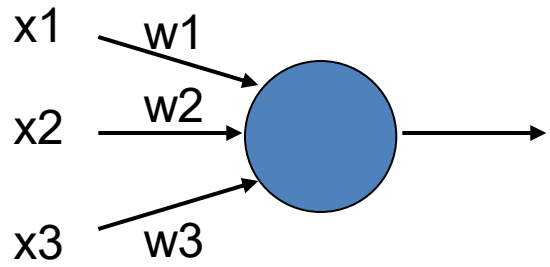
Q3. What if we set threshold at 0.5 and weight #3 to zero?

$$x = [\ 1.0,\ 0.5,\ 2.0\ ]$$

$$w = [\ 0.2,\ 0.5,\ 0.5\ ]$$

$$t = 1.0$$

$$a = \sum_{i=1}^{M} x_i w_i$$



Q1. What is the activation, *a*, of the neuron?

$$a = \sum_{i=1}^{M} x_i w_i = (1.0 \times 0.2) + (0.5 \times 0.5) + (2.0 \times 0.5) = 1.45$$

Q2. Does the neuron fire?

if (activation > threshold) output=1 else output=0

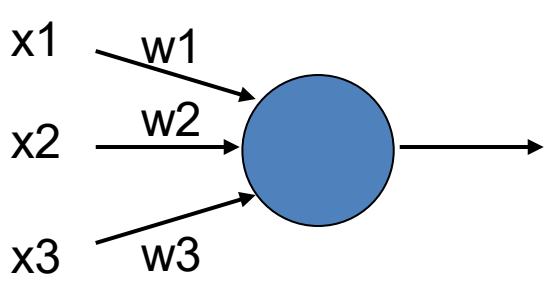
…. So yes, it fires.

$$x = [\ 1.0,\ 0.5,\ 2.0\ ]$$

$$w = [\ 0.2,\ 0.5,\ 0.5\ ]$$

$$t = 1.0$$

$$a = \sum_{i=1}^{M} x_i w_i$$



Q3. What if we set threshold at 0.5 and weight #3 to zero?

$$a = \sum_{i=1}^{M} x_i w_i = (1.0 \times 0.2) + (0.5 \times 0.5) + (2.0 \times 0.0) = 0.45$$

if (activation > threshold) output=1 else output=0

…. So no, it does not fire..

We can rearrange the decision rule….

$$\textit{if} \quad \left(\sum_{i=1}^{M} x_i w_i\right) \quad > \quad t \quad \text{then } output = 1, \text{ else } output = 0$$

$$\textit{if} \quad \left(\sum_{i=1}^{M} x_i w_i\right) - t \quad > \quad 0 \quad \text{then } output = 1, \text{ else } output = 0$$

$$\textit{if} \quad \left(\sum_{i=1}^{M} x_i w_i\right) + (-1 \times t) \quad > \quad 0 \quad \text{then } output = 1, \text{ else } output = 0$$

$$\textit{if} \quad \left(\sum_{i=1}^{M} x_i w_i\right) + (x_0 \times w_0) \quad > \quad 0 \quad \text{then } output = 1, \text{ else } output = 0$$

$$\textit{if} \quad \left(\sum_{i=0}^{M} x_i w_i\right) \quad > \quad 0 \quad \text{then } output = 1, \text{ else } output = 0$$

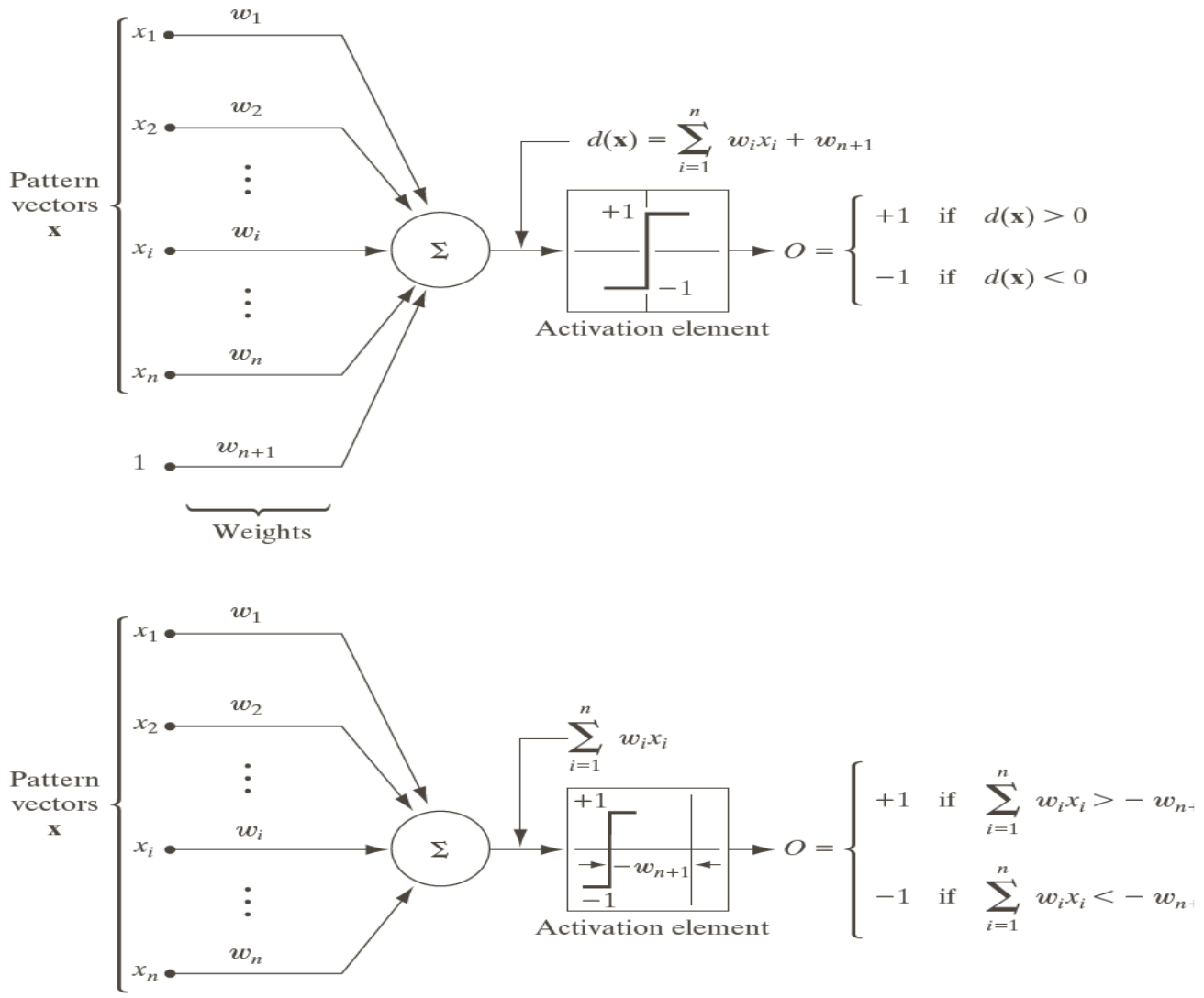
We now treat the threshold like any other weight with a permanent input of -1

Perceptron Learning Algorithm

```
initialise weights (w)
Repeat until all points are correctly classified
        Repeat for each point
                Calculate margin (yiwXi) for point i
                If margin > 0, point is correctly classified
                Else change the weights to increase margin such
                that Δw = ηyiXi and wnew = wold + Δw
        end
end
```

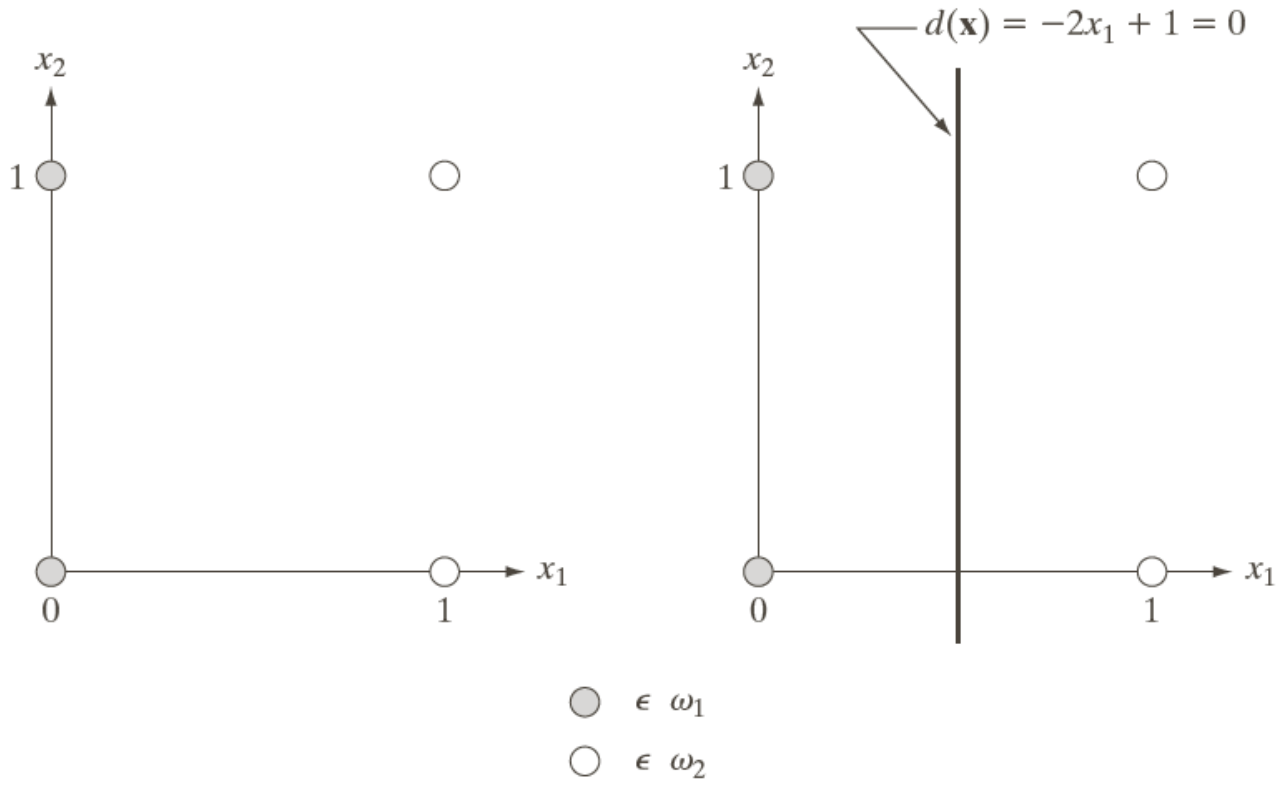
Perceptron convergence theorem:

If the data is linearly separable, then application of the Perceptron learning rule will find a separating decision boundary, within a finite number of iterations



a
b **FIGURE 12.14** Two equivalent representations of the perceptron model for two pattern classes.

Decision Boundary Using Perceptron

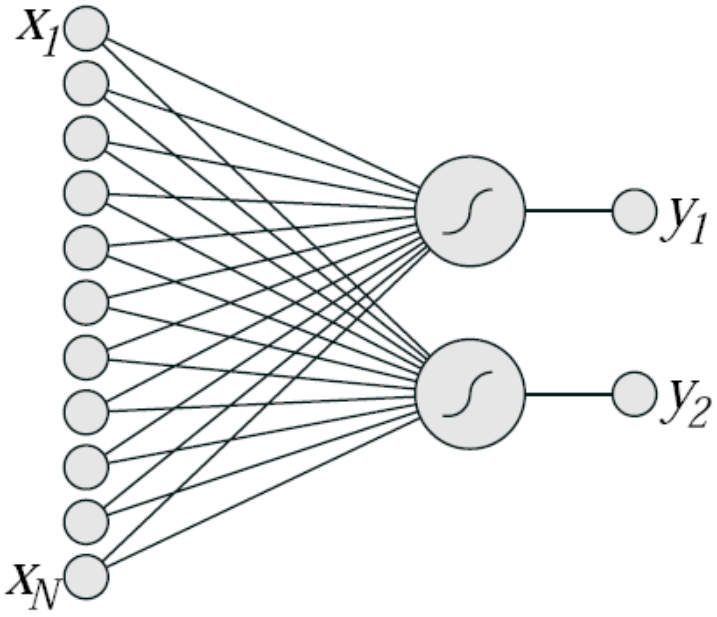


a b

FIGURE 12.15
(a) Patterns belonging to two classes.
(b) Decision boundary determined by training.

Multiple Outputs

- We can produce more complicated machines by using several perceptrons



- Treat each perceptron independently
- Consider just single perceptron

Criticize on Perceptron

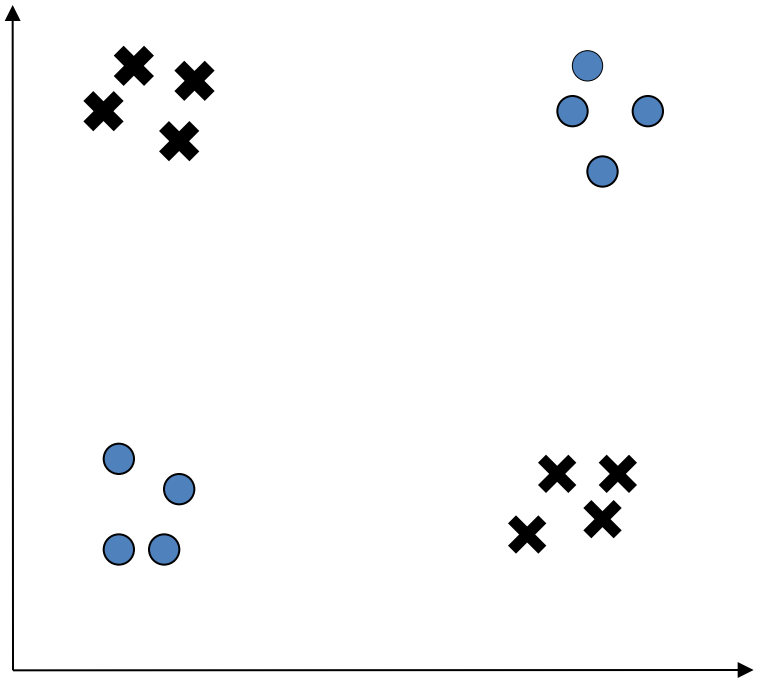
Minsky and Papert criticise the perceptron (1969)

Minsky and Papert's criticism was partly right

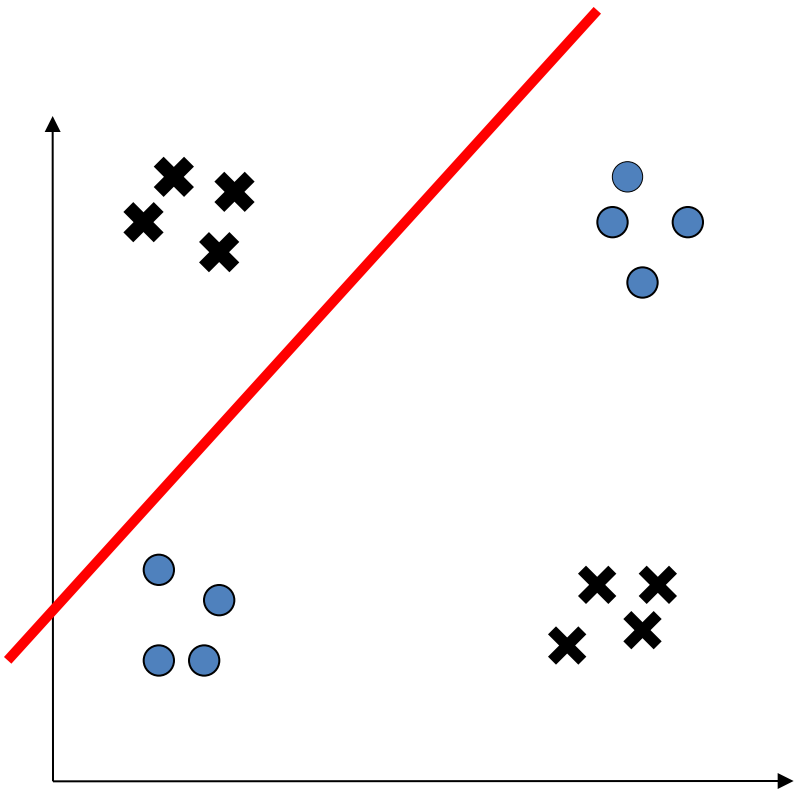
We have no a priori reason to think that problems should be linear separable

However, it turns out the world is full of problems that are a least close to linear separable

Can a Perceptron solve this problem?



Can a Perceptron solve this problem? ….. NO.



Perceptrons only solve

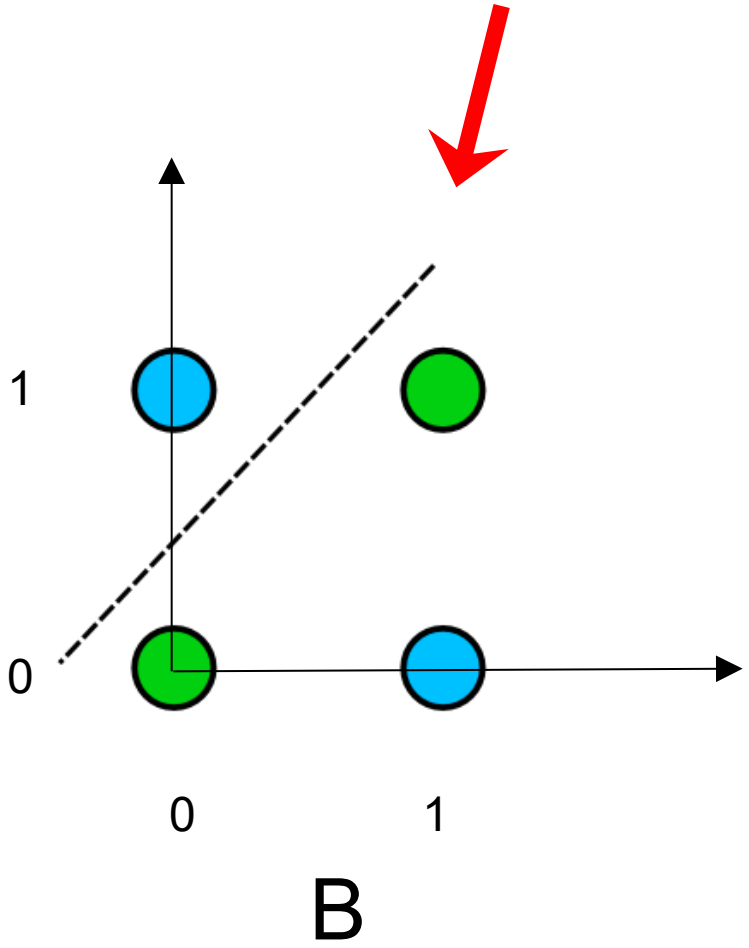
LINEARLY SEPARABLE

problems

Exclusive-OR gate

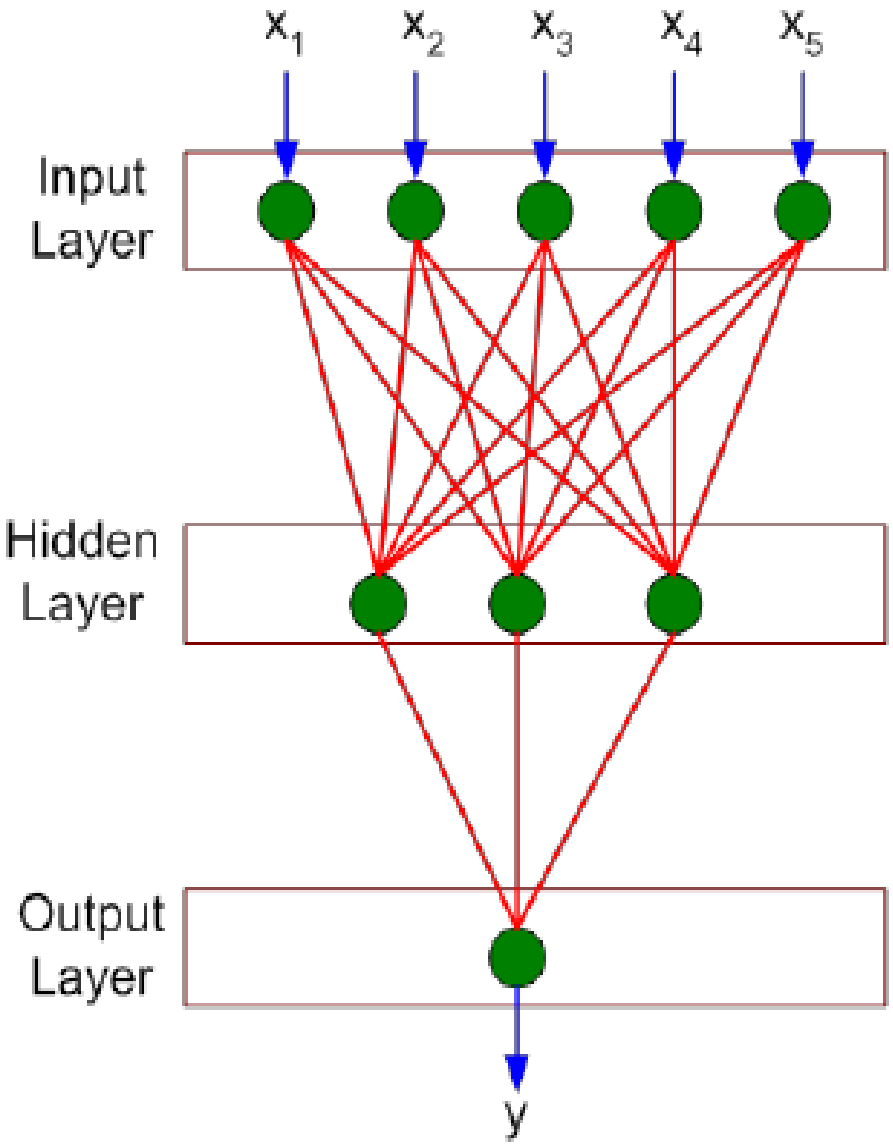
A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

With a perceptron…
the decision boundary is
LINEAR



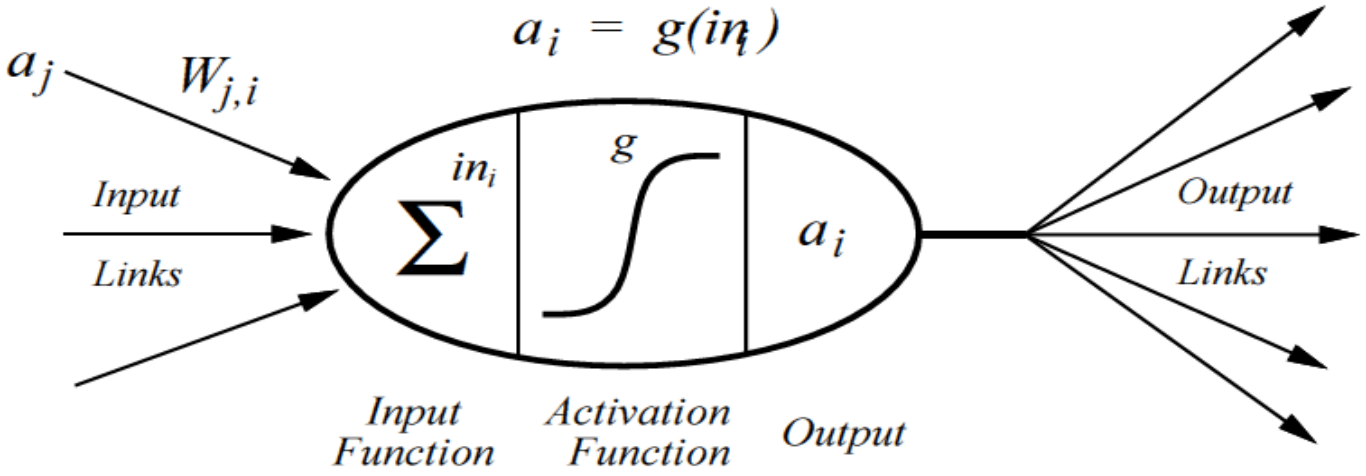
Artificial Neural Networks (ANN)

- Neural computing requires a number of neurons, to be connected together into a neural network.
- A neural network consists of:
  - layers
  - links between layers
- The links are weighted.
- There are three kinds of layers:
  1. input layer
  2. Hidden layer
  3. output layer



A simple neuron

At each neuron, every input has an associated weight which modifies the strength of each input.
The neuron simply adds together all the inputs and calculates an output to be passed on.

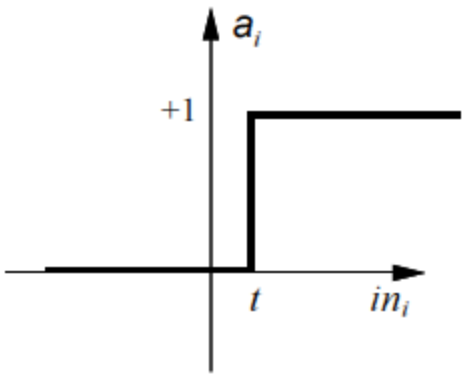


Activation function

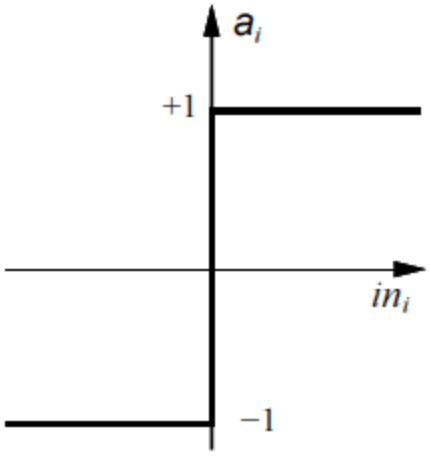
The **activation function** g calculates the output a_i (from the inputs) which will be transferred to other units via output-links:

$$a_i := g(in_i)$$

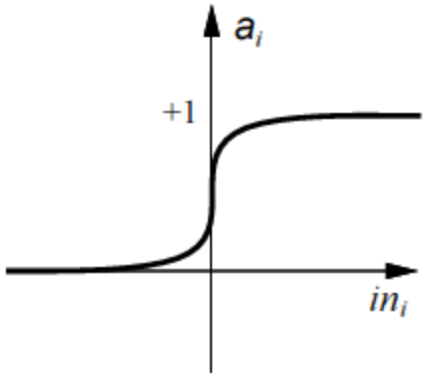
Examples:



(a) Step function



(b) Sign function



(c) Sigmoid function

MultiLayer Perceptron (MLP)

Motivation

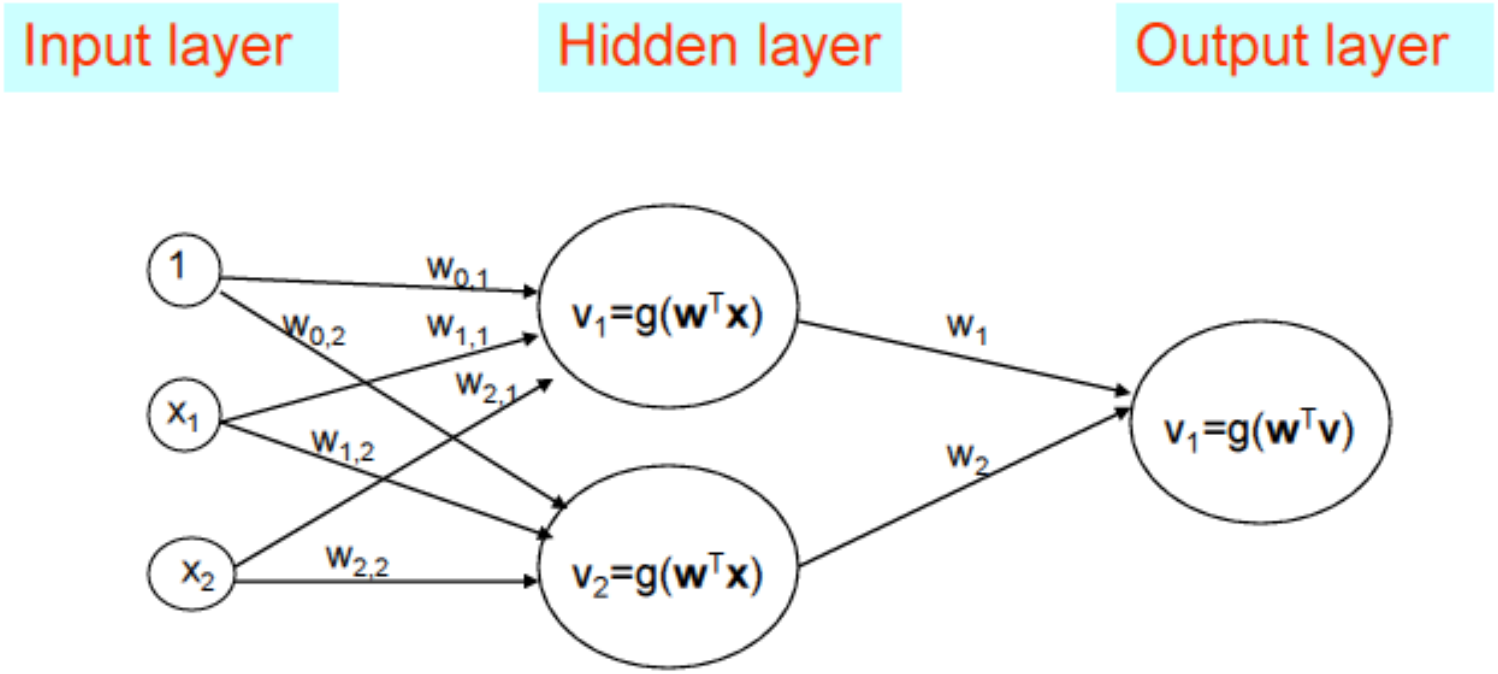
Perceptrons are limited because they can only solve problems that are linearly separable

We would like to build more complicated learning machines to model our data

One way to do this is to build a multiple layers of perceptrons

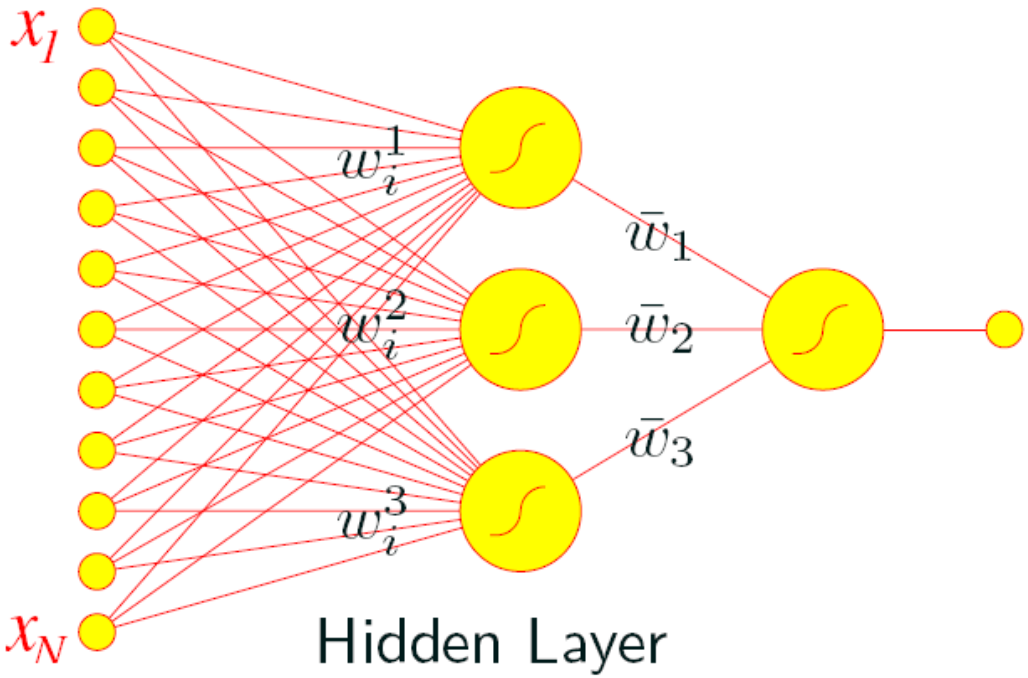
Multi layer networks

So far, we discussed networks with one layer.
But these networks can be extended to combine several layers, increasing the set of functions that can be represented using a NN



MLP

- E.g. A multi-layer perceptron (MLP)



$$z = f(\boldsymbol{x}|\boldsymbol{w}) = g\left(\sum_i \bar{w}_i \, g\left(\boldsymbol{x}^\mathsf{T}\boldsymbol{w}^i\right)\right)$$

- Note that $\boldsymbol{w}$ in $f(\boldsymbol{x}|\boldsymbol{w})$ includes all weights. $\boldsymbol{w}$ is no longer the same dimension as the inputs $\boldsymbol{x}$

MLP

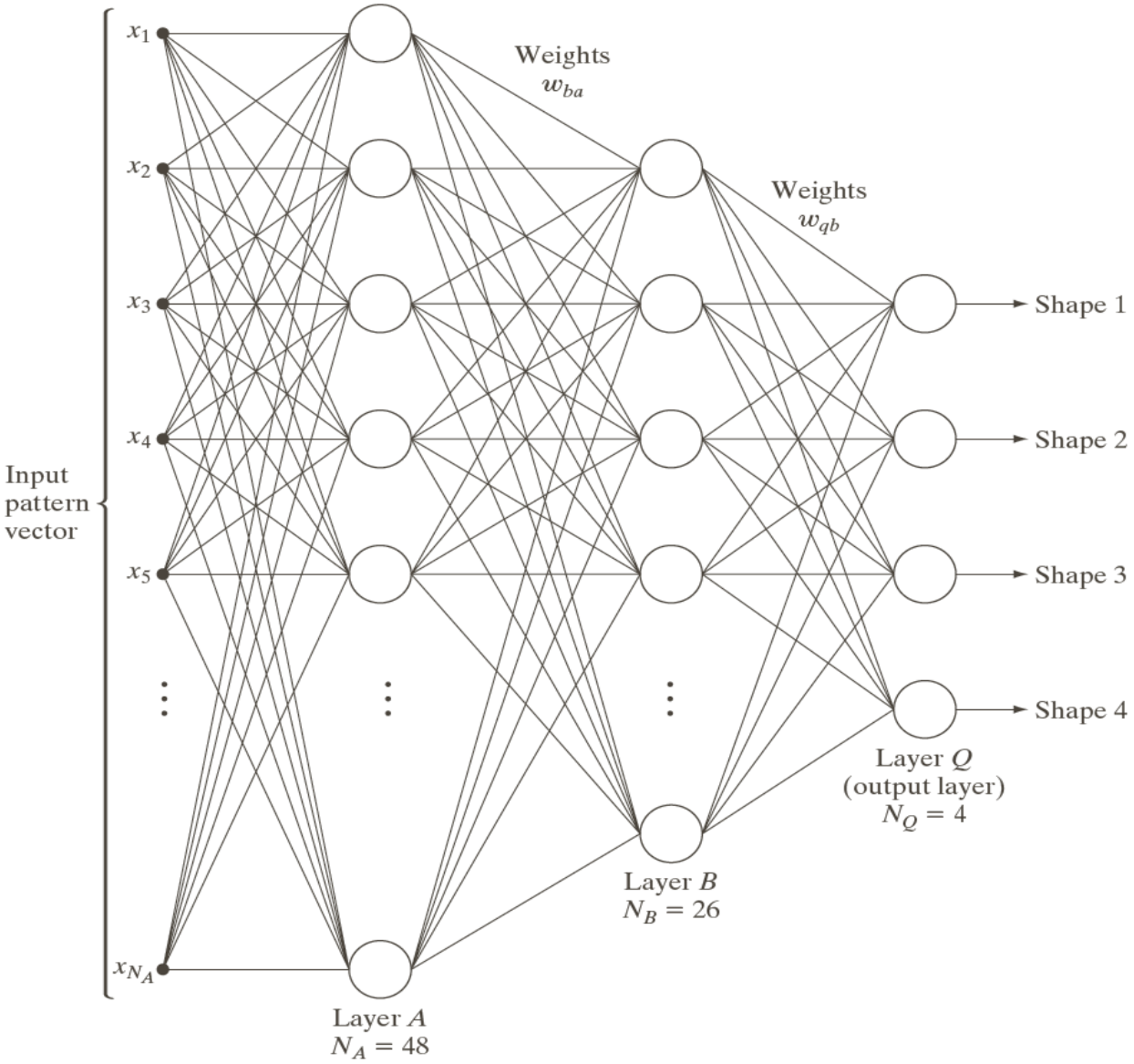


FIGURE 12.19 Three-layer neural network used to recognize the shapes in Fig. 12.18. (Courtesy of Dr. Lalit Gupta, ECE Department, Southern Illinois University.)

Multilayer Neural Network

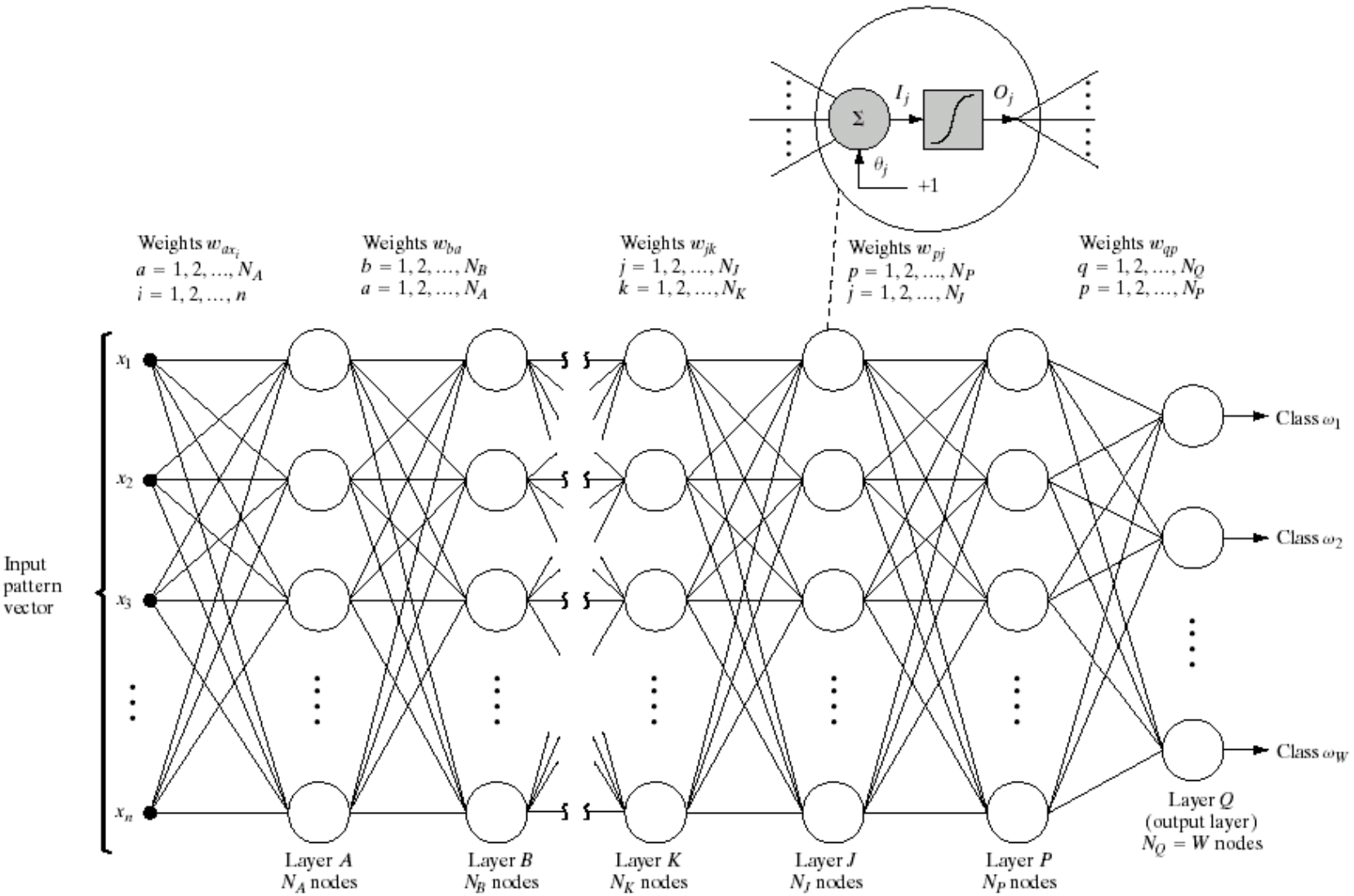


FIGURE 12.16 Multilayer feedforward neural network model. The blowup shows the basic structure of each neuron element throughout the network. The offset, θ_j, is treated as just another weight.

Sigmoid Response Functions

- To obtain a more powerful learning machine we have to use non-linear response functions
- The two most used functions are the sigmoidal (squashing) functions:
- A logistic function

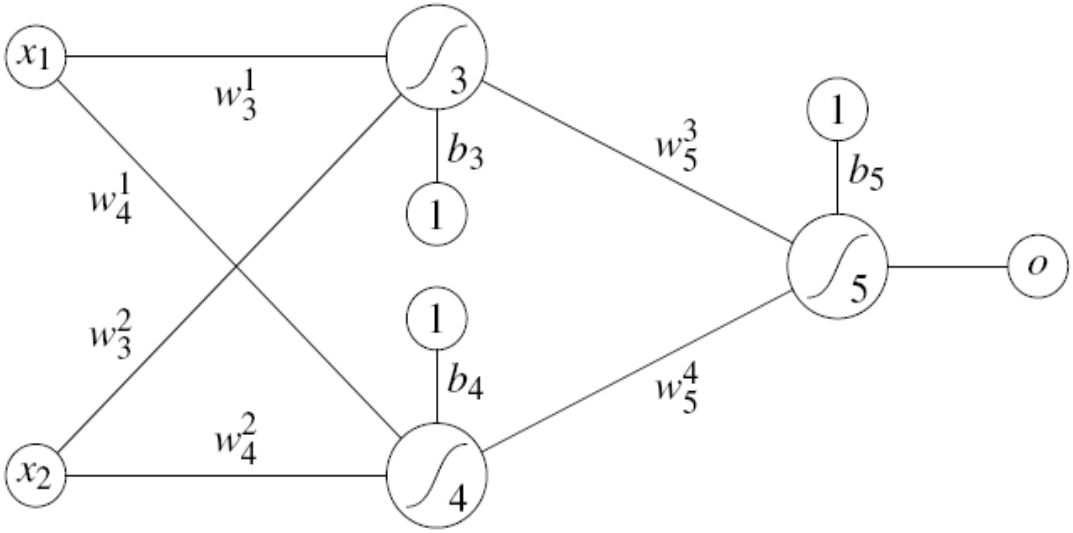
$$g(V) = \frac{1}{1 + \mathrm{e}^{-(V)}}$$

- A tanh function

$$g(V) = \tanh(V)$$

MLP

- A diagram for an neural network such as an MLP

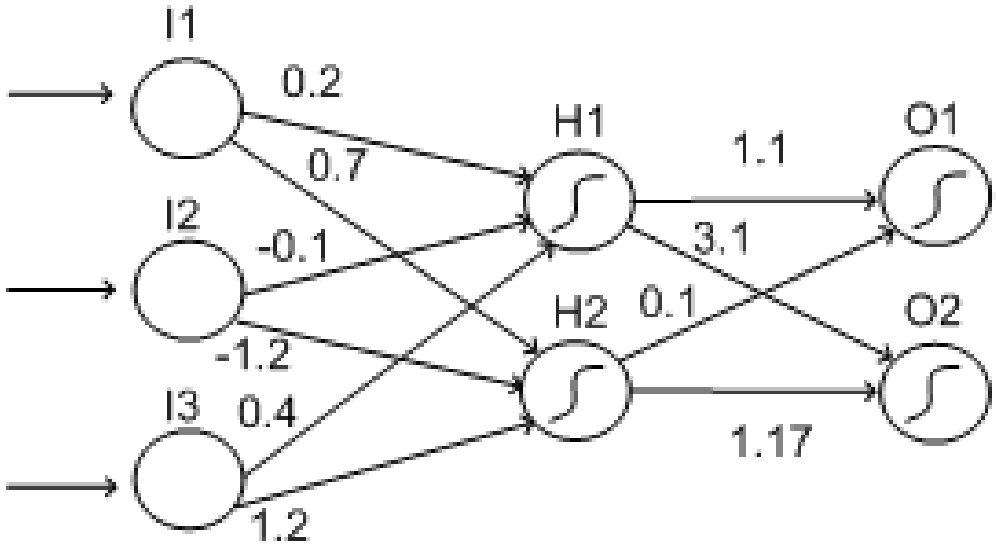


- Stands for the function $(o = f(\boldsymbol{x}|\boldsymbol{w}))$

$$o = g\big(w_5^3\, g(w_3^1\, x_1 + w_3^2\, x_2 + b_3) + w_5^4\, g(w_4^1\, x_1 + w_4^2\, x_2 + b_4) + b_5\big)$$

where, for example, $g(V) = \frac{1}{1+\mathrm{e}^{-V}}$

Example of multilayer neural network



Suppose input values are 10, 30, 20

The weighted sum coming into H1

$$S_{H1} = (0.2 * 10) + (-0.1 * 30) + (0.4 * 20)$$
$$= 2 - 3 + 8 = 7.$$

The σ function is applied to S_{H1}:

$$\sigma(S_{H1}) = 1/(1+e^{-7}) = 1/(1+0.000912) = 0.999$$

Similarly, the weighted sum coming into H2:

$$S_{H2} = (0.7 * 10) + (-1.2 * 30) + (1.2 * 20)$$
$$= 7 - 36 + 24 = -5$$

σ applied to S_{H2}:

$$\sigma(S_{H2}) = 1/(1+e^{5}) = 1/(1+148.4) = 0.0067$$

Now the weighted sum to output unit O1 :

$$S_{O1} = (1.1 * 0.999) + (0.1*0.0067) = 1.0996$$

The weighted sum to output unit O2:

$$S_{O2} = (3.1 * 0.999) + (1.17*0.0067) = 3.1047$$

The output sigmoid unit in O1:

$$\sigma(S_{O1}) = 1/(1+e^{-1.0996}) = 1/(1+0.333) = 0.750$$

The output from the network for O2:

$$\sigma(S_{O2}) = 1/(1+e^{-3.1047}) = 1/(1+0.045) = 0.957$$

The input triple (10,30,20) would be categorised with O2, because this has the larger output.

Neural Networks

Neural networks: the original linear classifier

(Before) Linear score function:

$$f = Wx$$

$$x \in \mathbb{R}^D, W \in \mathbb{R}^{C \times D}$$

Neural networks: 2 layers

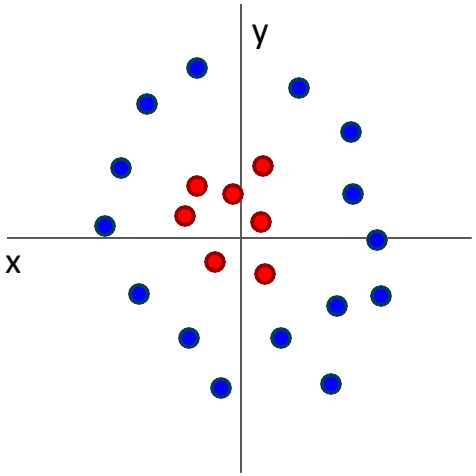
(Before) Linear score function: $f = Wx$

(Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$

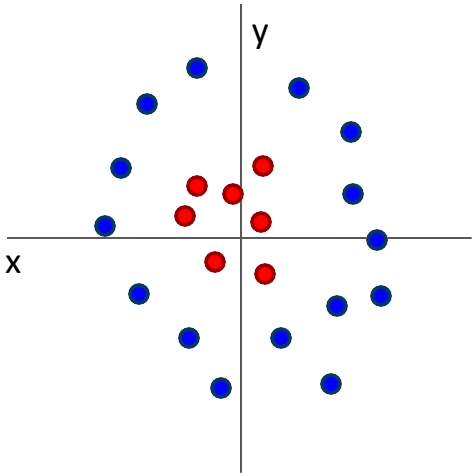
(In practice we will usually add a learnable bias at each layer as well)

Why do we want non-linearity?



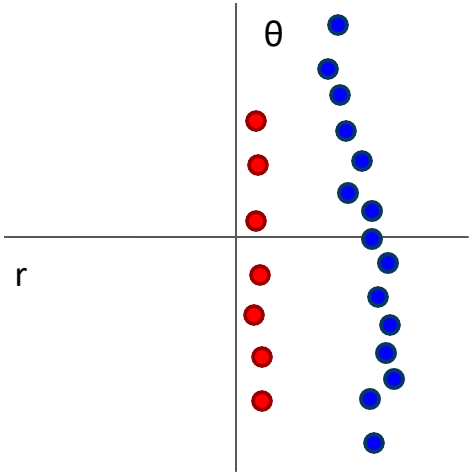
Cannot separate red and blue points with linear classifier

Why do we want non-linearity?



Cannot separate red and blue points with linear classifier

$$f(x, y) = (r(x, y), \theta(x, y))$$



After applying feature transform, points can be separated by linear classifier

Neural networks: also called fully connected network

(Before) Linear score function: $f = Wx$

(Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

"Neural Network" is a very broad term; these are more accurately called "fully-connected networks" or sometimes "multi-layer perceptrons" (MLP)

(In practice we will usually add a learnable bias at each layer as well)

Neural networks: 3 layers

(Before) Linear score function: $f = Wx$

(Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

or 3-layer Neural Network $f = W_3 \max(0, W_2 \max(0, W_1 x))$

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H_1 \times D}, W_2 \in \mathbb{R}^{H_2 \times H_1}, W_3 \in \mathbb{R}^{C \times H_2}$$

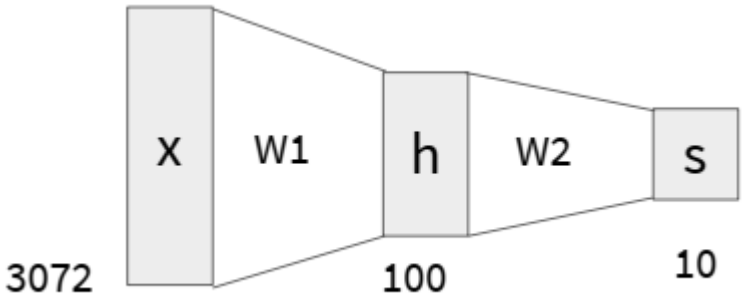
(In practice we will usually add a learnable bias at each layer as well)

Neural networks: hierarchical computation

Neural networks: hierarchical computation

(Before) Linear score function: $f = Wx$

(Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

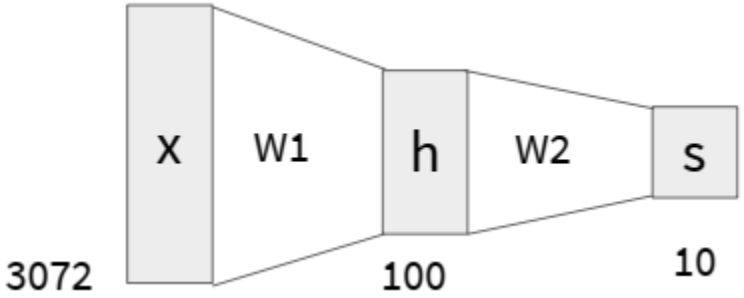


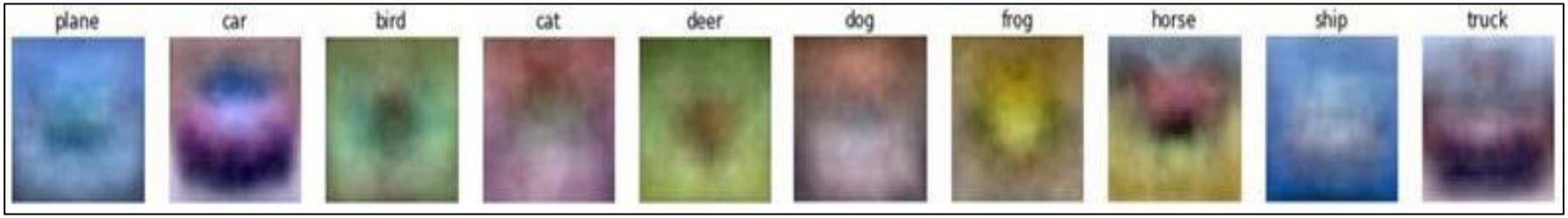
$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$

Neural networks: learning 100s of templates

(Before) Linear score function: $f = Wx$

(Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$





Learn 100 templates instead of 10.

Share templates between classes

Neural networks: why is max operator important?

(Before) Linear score function: $f = Wx$

(Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

The function $\max(0, z)$ is called the activation function.

Q: What if we try to build a neural network without one?

$$f = W_2 W_1 x$$

Neural networks: why is max operator important?

(Before) Linear score function: $f = Wx$

(Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

The function $\max(0, z)$ is called the activation function.

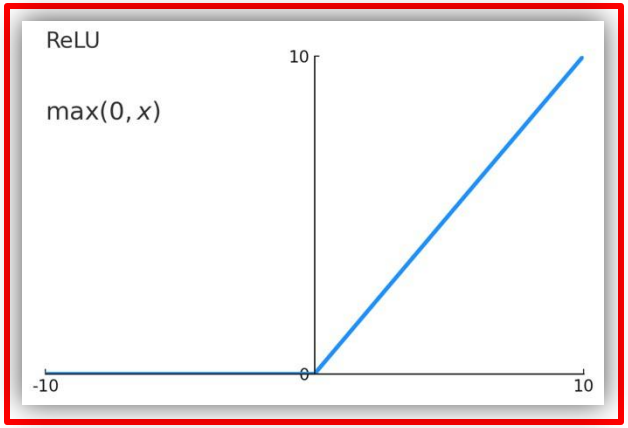
Q: What if we try to build a neural network without one?

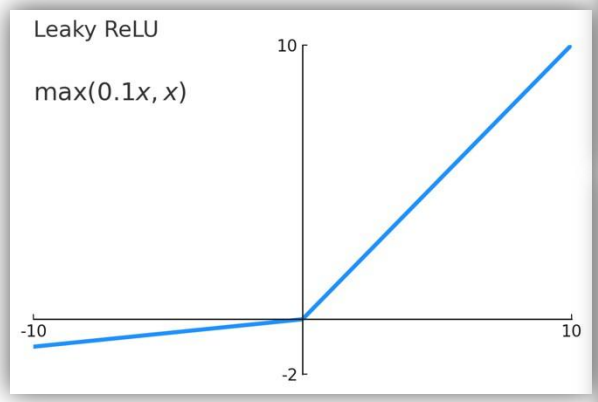
$$f = W_2 W_1 x \qquad W_3 = W_2 W_1 \in \mathbb{R}^{C \times H}, f = W_3 x$$

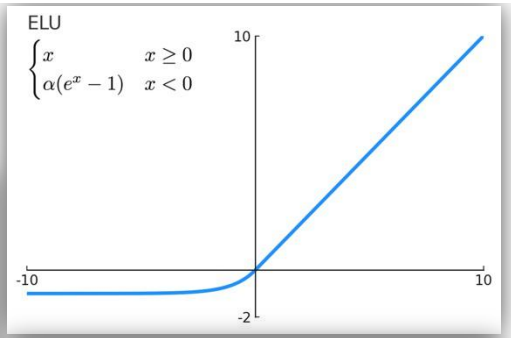
A: We end up with a linear classifier again!

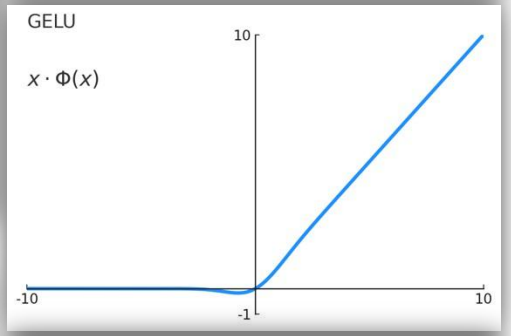
Activation functions

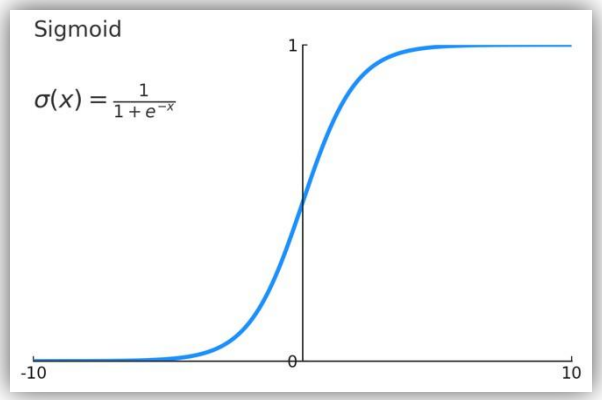
ReLU is a good default choice for most problems

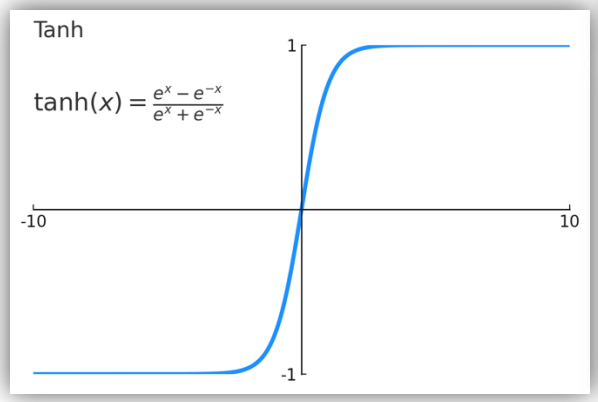


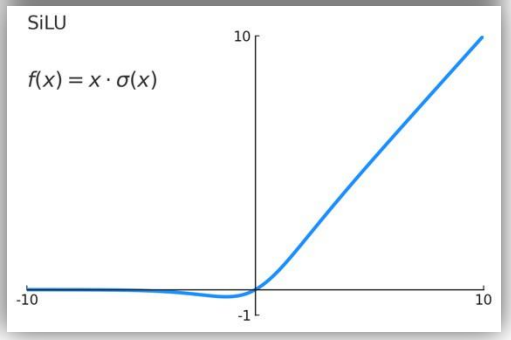




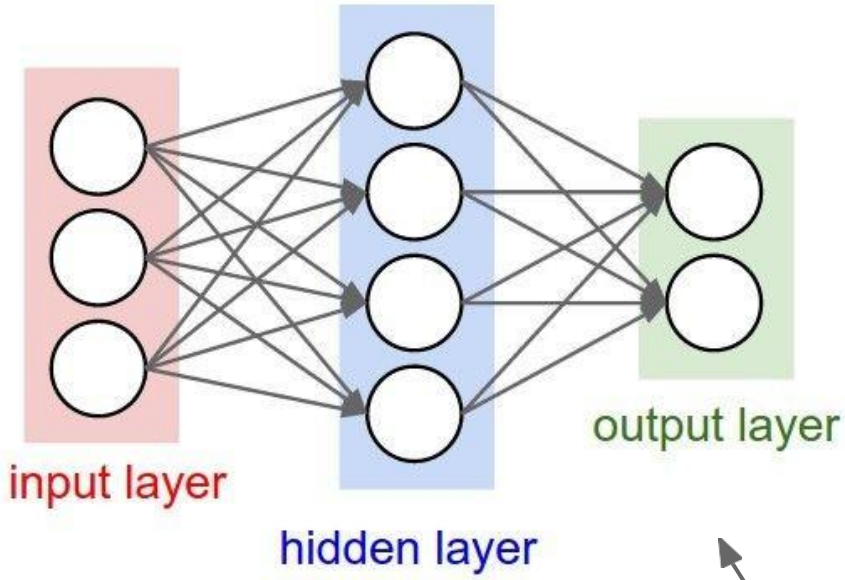


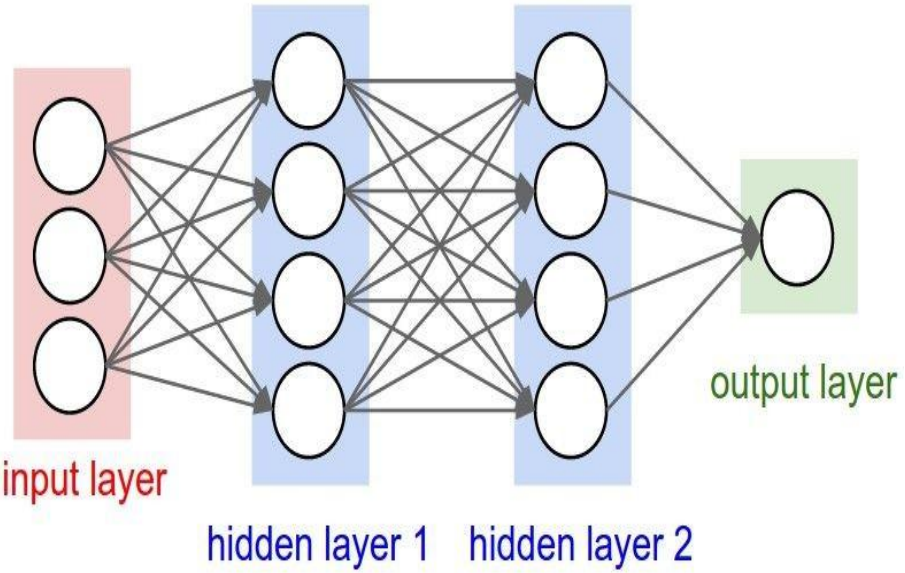






Neural networks: Architectures



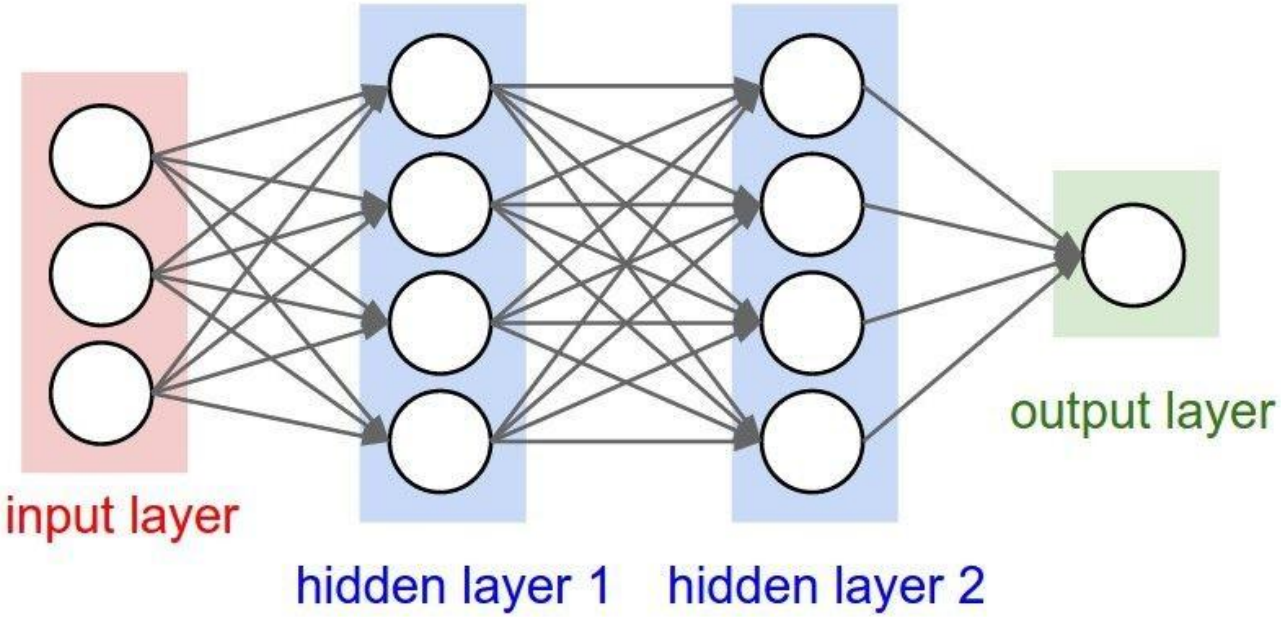


"2-layer Neural Net", or
"1-hidden-layer Neural Net"

"3-layer Neural Net", or
"2-hidden-layer Neural Net"

"Fully-connected" layers

Example feed-forward computation of a neural network




```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
import numpy as np
from numpy.random import randn

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = randn(N, D_in), randn(N, D_out)
w1, w2 = randn(D_in, H), randn(H, D_out)

for t in range(2000):
  h = 1 / (1 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
  print(t, loss)

  grad_y_pred = 2.0 * (y_pred - y)
  grad_w2 = h.T.dot(grad_y_pred)
  grad_h = grad_y_pred.dot(w2.T)
  grad_w1 = x.T.dot(grad_h * h * (1 - h))

  w1 -= 1e-4 * grad_w1
  w2 -= 1e-4 * grad_w2
```

Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
import numpy as np
from numpy.random import randn

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = randn(N, D_in), randn(N, D_out)
w1, w2 = randn(D_in, H), randn(H, D_out)

for t in range(2000):
  h = 1 / (1 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
  print(t, loss)

  grad_y_pred = 2.0 * (y_pred - y)
  grad_w2 = h.T.dot(grad_y_pred)
  grad_h = grad_y_pred.dot(w2.T)
  grad_w1 = x.T.dot(grad_h * h * (1 - h))

  w1 -= 1e-4 * grad_w1
  w2 -= 1e-4 * grad_w2
```

Define the network

Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1  import numpy as np
2  from numpy.random import randn
3
4  N, D_in, H, D_out = 64, 1000, 100, 10
5  x, y = randn(N, D_in), randn(N, D_out)
6  w1, w2 = randn(D_in, H), randn(H, D_out)
7
8  for t in range(2000):
9    h = 1 / (1 + np.exp(-x.dot(w1)))
10   y_pred = h.dot(w2)
11   loss = np.square(y_pred - y).sum()
12   print(t, loss)
13
14   grad_y_pred = 2.0 * (y_pred - y)
15   grad_w2 = h.T.dot(grad_y_pred)
16   grad_h = grad_y_pred.dot(w2.T)
17   grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19   w1 -= 1e-4 * grad_w1
20   w2 -= 1e-4 * grad_w2
```

Define the network

Forward pass

Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1  import numpy as np
2  from numpy.random import randn
3
4  N, D_in, H, D_out = 64, 1000, 100, 10
5  x, y = randn(N, D_in), randn(N, D_out)
6  w1, w2 = randn(D_in, H), randn(H, D_out)
7
8  for t in range(2000):
9    h = 1 / (1 + np.exp(-x.dot(w1)))
10   y_pred = h.dot(w2)
11   loss = np.square(y_pred - y).sum()
12   print(t, loss)
13
14   grad_y_pred = 2.0 * (y_pred - y)
15   grad_w2 = h.T.dot(grad_y_pred)
16   grad_h = grad_y_pred.dot(w2.T)
17   grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19   w1 -= 1e-4 * grad_w1
20   w2 -= 1e-4 * grad_w2
```

Define the network

Forward pass

Calculate the analytical gradients

Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1  import numpy as np
2  from numpy.random import randn
3
4  N, D_in, H, D_out = 64, 1000, 100, 10
5  x, y = randn(N, D_in), randn(N, D_out)
6  w1, w2 = randn(D_in, H), randn(H, D_out)
7
8  for t in range(2000):
9    h = 1 / (1 + np.exp(-x.dot(w1)))
10   y_pred = h.dot(w2)
11   loss = np.square(y_pred - y).sum()
12   print(t, loss)
13
14   grad_y_pred = 2.0 * (y_pred - y)
15   grad_w2 = h.T.dot(grad_y_pred)
16   grad_h = grad_y_pred.dot(w2.T)
17   grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19   w1 -= 1e-4 * grad_w1
20   w2 -= 1e-4 * grad_w2
```

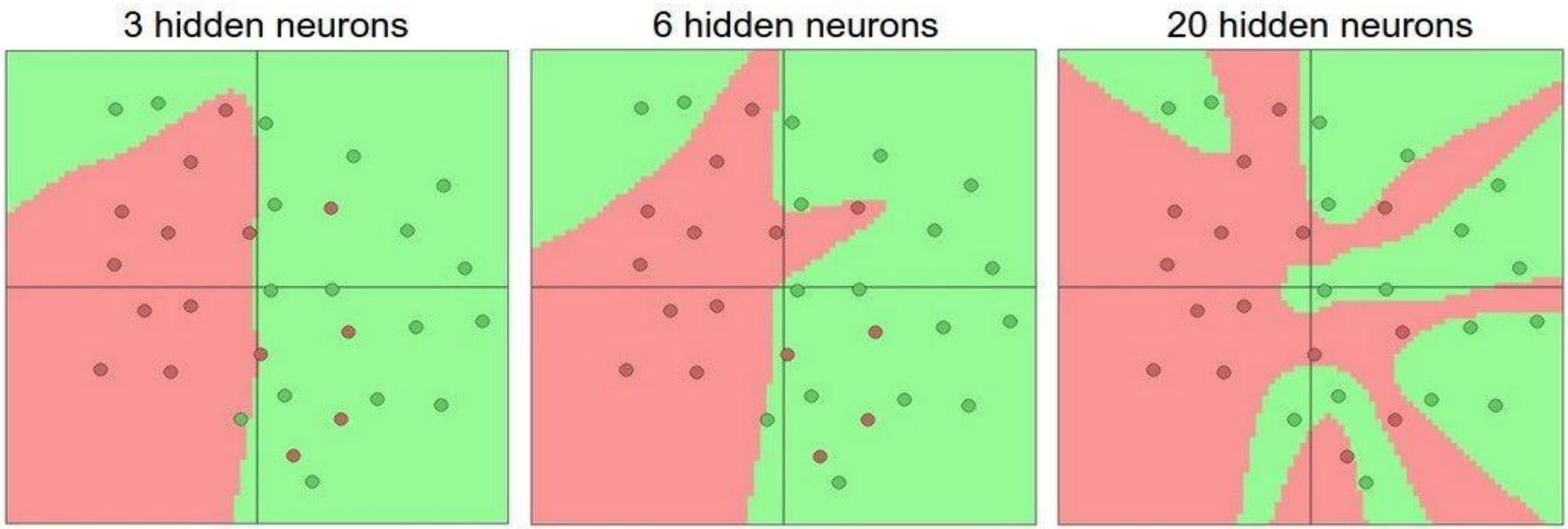
Define the network

Forward pass

Calculate the analytical gradients

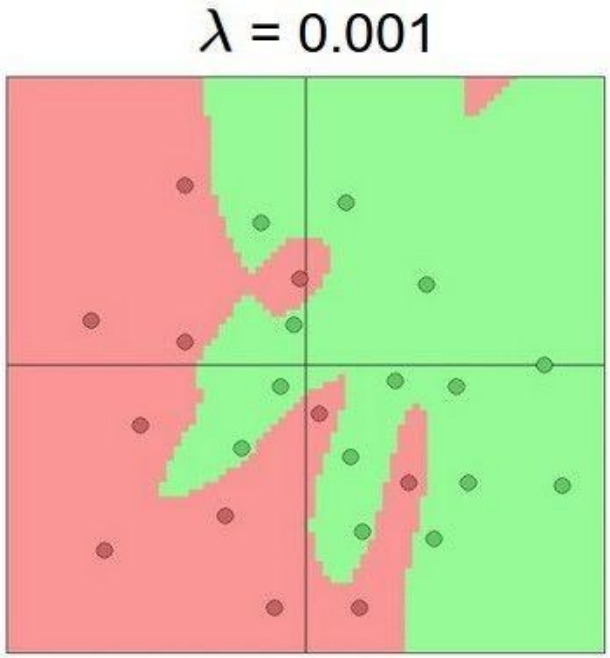
Gradient descent

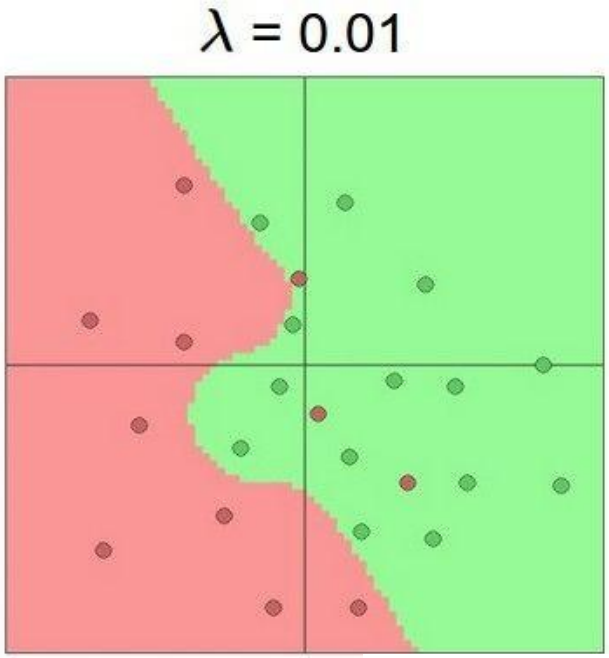
Setting the number of layers and their sizes

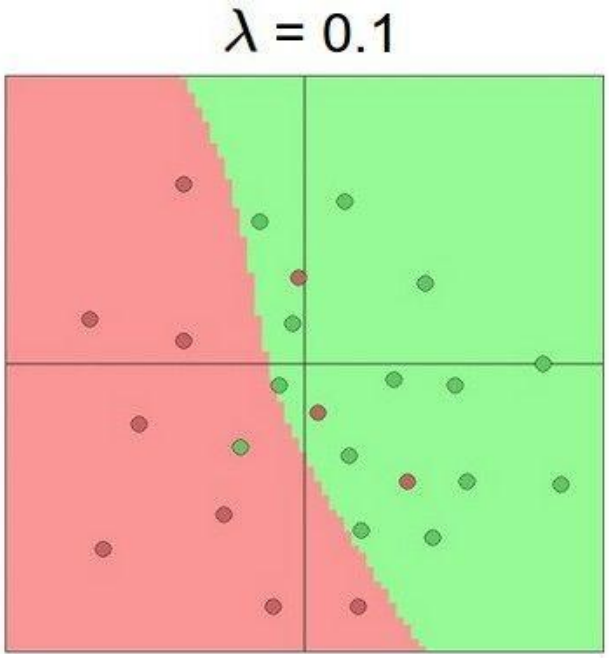


more neurons = more capacity

Do not use size of neural network as a regularizer. Use stronger regularization instead:







$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W)$$

(Web demo with ConvNetJS: http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html)

Plugging in neural networks with loss functions

- Nonlinear score function
- SVM Loss on predictions

$$s = f(x; W_1, W_2) = W_2 \max(0, W_1 x)$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

- Regularization

$$R(W) = \sum_k W_k^2$$

- Total loss: data loss + regularization

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda R(W_1) + \lambda R(W_2)$$

Problem: How to compute gradients?

- Nonlinear score function $s = f(x; W_1, W_2) = W_2 \max(0, W_1 x)$
- SVM Loss on predictions $L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$
- Regularization $R(W) = \sum_k W_k^2$
- Total loss: data loss + regularization $L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda R(W_1) + \lambda R(W_2)$

If we can compute $\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_2}$ then we can learn W_1 and W_2

(Bad) Idea: Derive on paper

$$\nabla_W L$$

Problem: Very tedious: Lots of matrix calculus, need lots of paper

Problem: What if we want to change loss? E.g. use softmax instead of SVM? Need to re-derive from scratch =(

Problem: Not feasible for very complex models!

$$s = f(x; W) = Wx$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$= \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1)$$

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda \sum_k W_k^2$$

$$= \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2$$

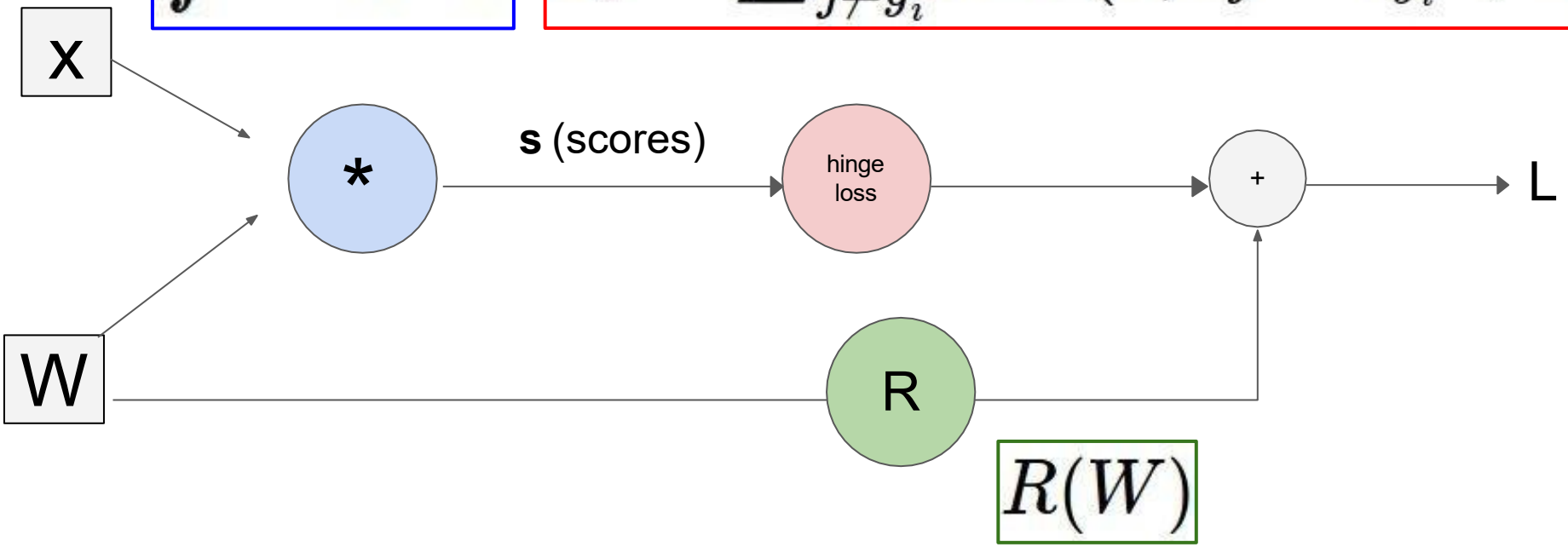
$$\nabla_W L = \nabla_W \left(\frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2 \right)$$

Better Idea: Computational graphs + Backpropagation

Hinge Loss:

$$f = Wx$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

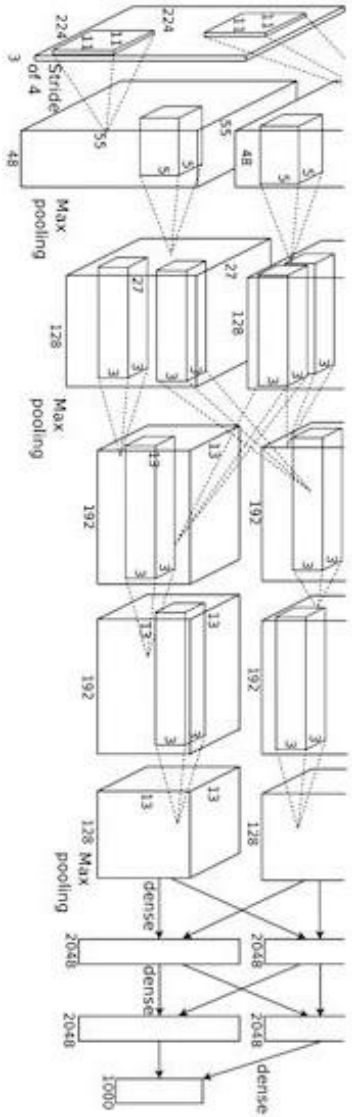


Convolutional network (AlexNet)

input image

weights

loss



Really complex neural networks!!

input image

loss

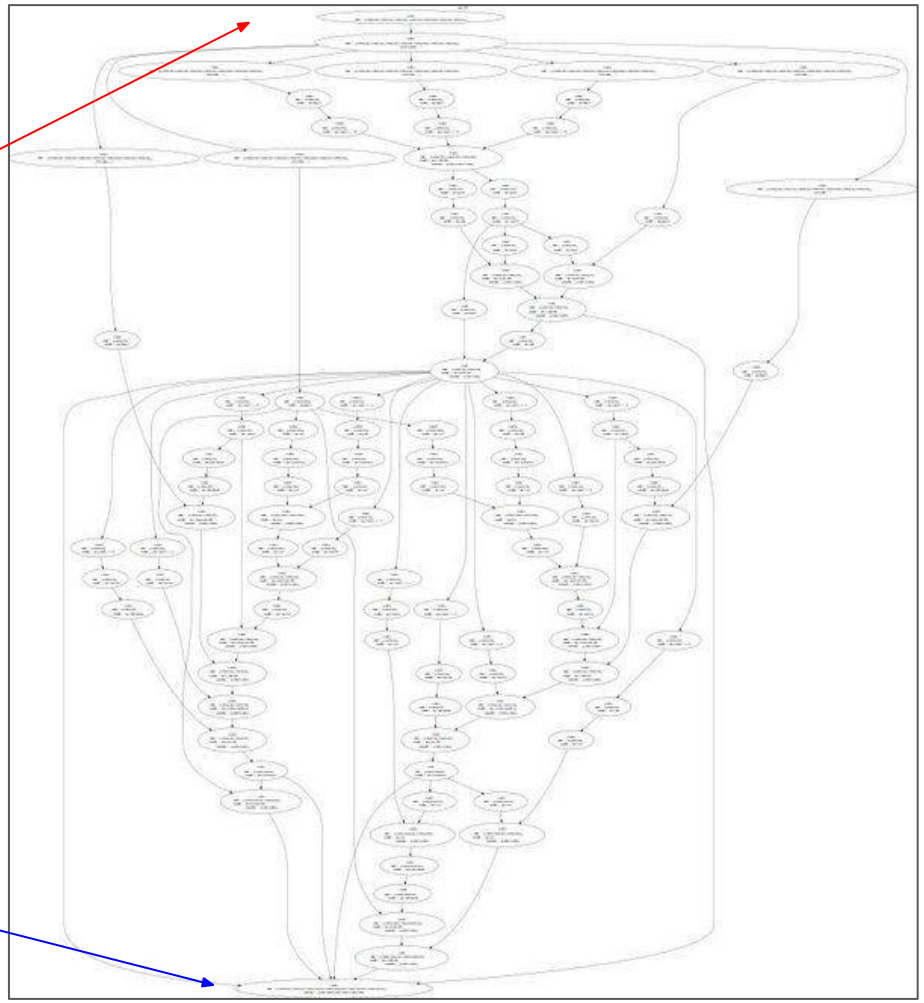


Figure reproduced with permission from a Twitter post by Andrej Karpathy.

Neural Turing Machine

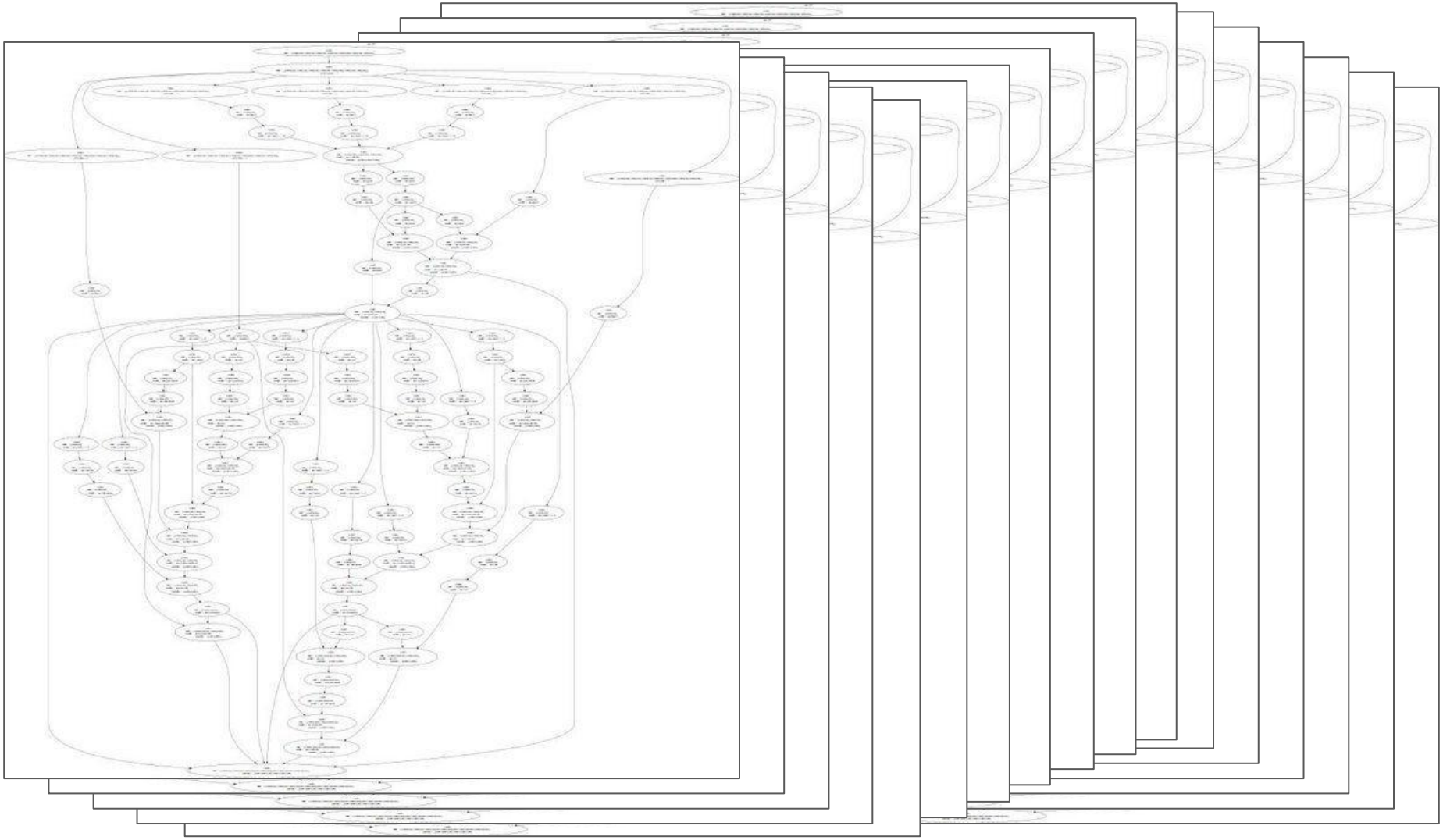


Figure reproduced with permission from a Twitter post by Andrej Karpathy.

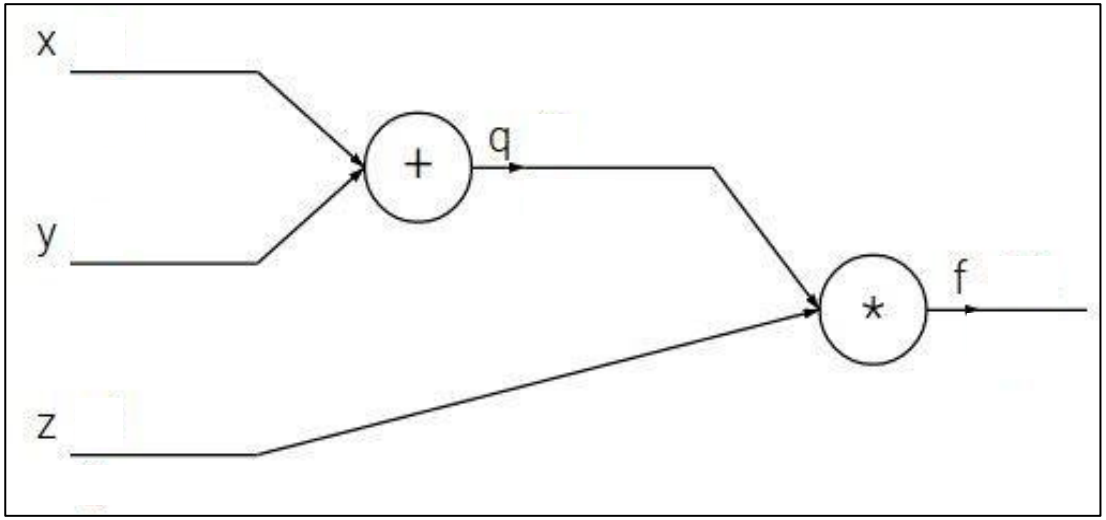
SOLUTION: BACKPROPAGATION

Backpropagation: a simple example

$$f(x, y, z) = (x + y)z$$

Backpropagation: a simple example

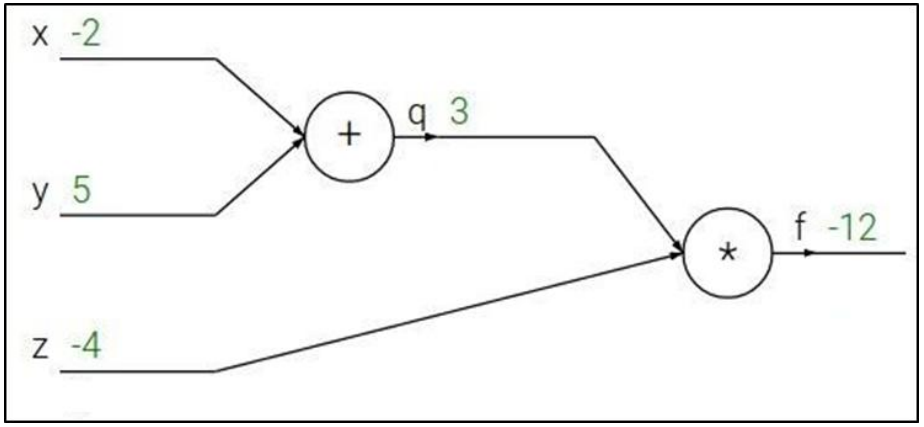
$$f(x, y, z) = (x + y)z$$



Backpropagation: a simple example

$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

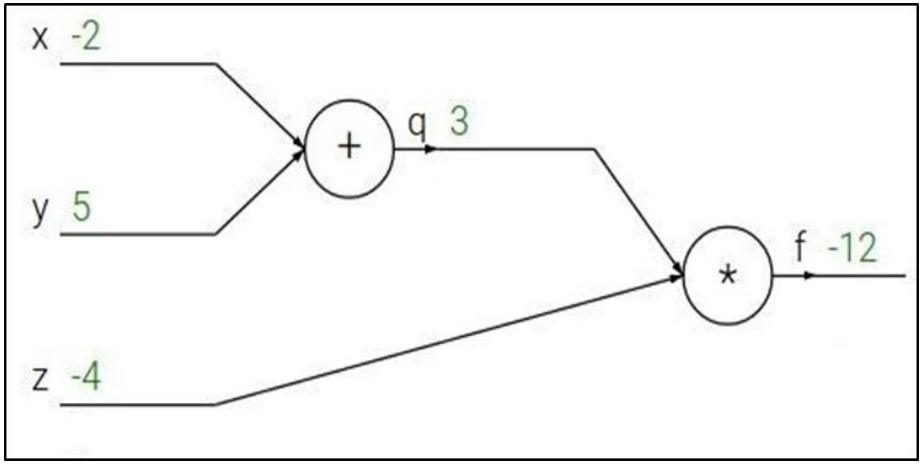


Backpropagation: a simple example

$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

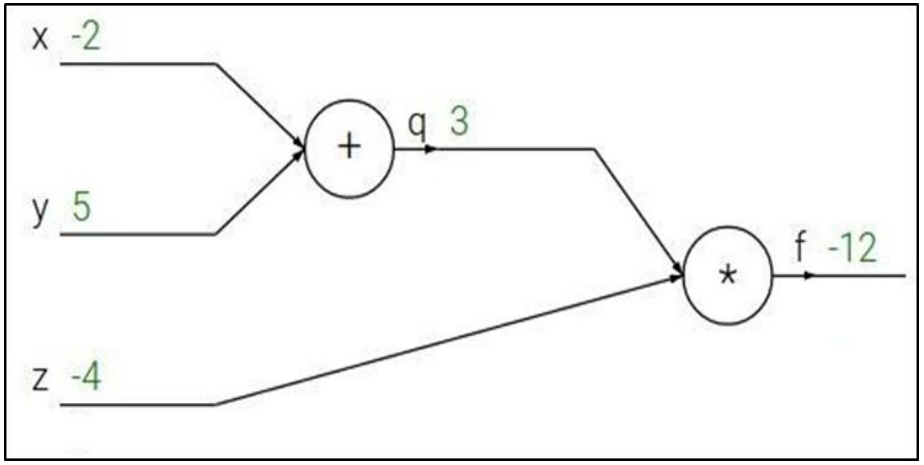
$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$



Backpropagation: a simple example

$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4



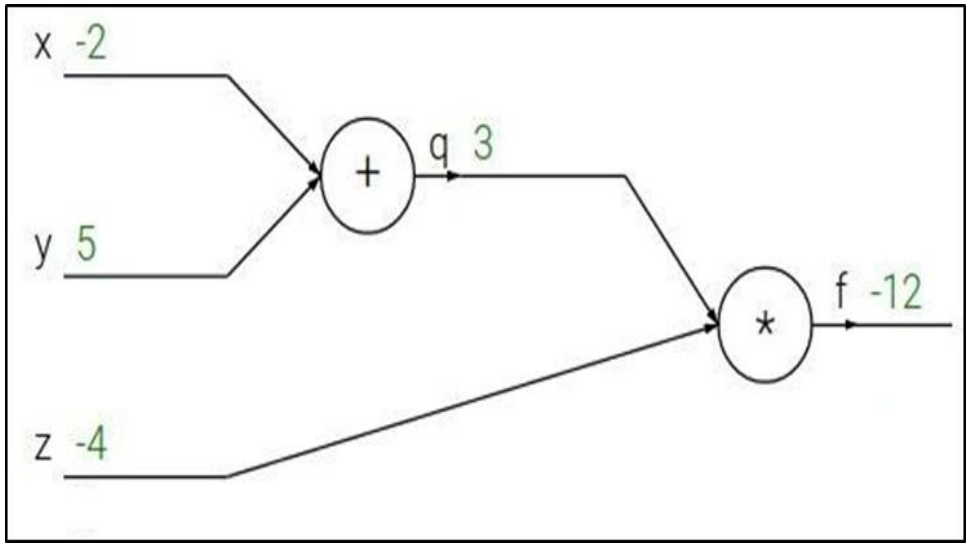
$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Backpropagation: a simple example

$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4



$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

Backpropagation: a simple example

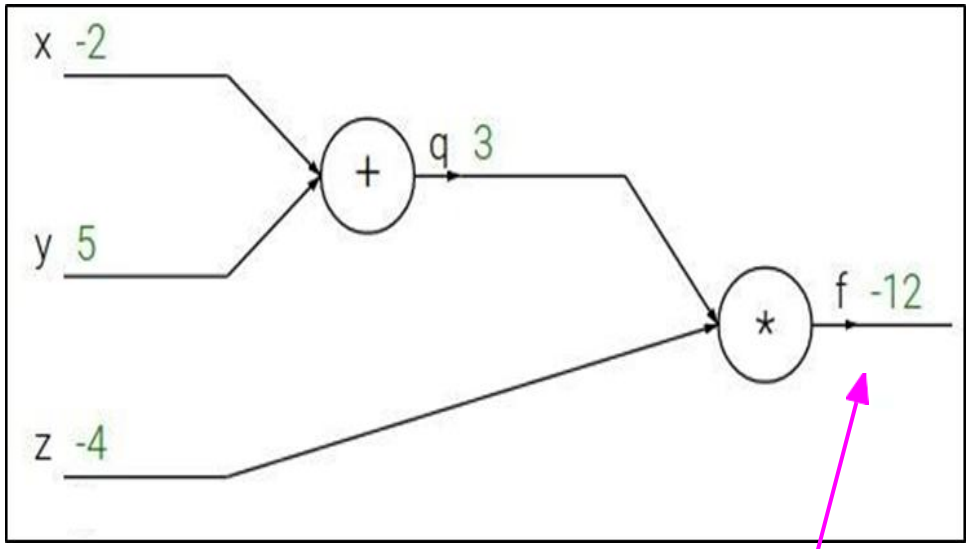
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$$\frac{\partial f}{\partial f}$$

Backpropagation: a simple example

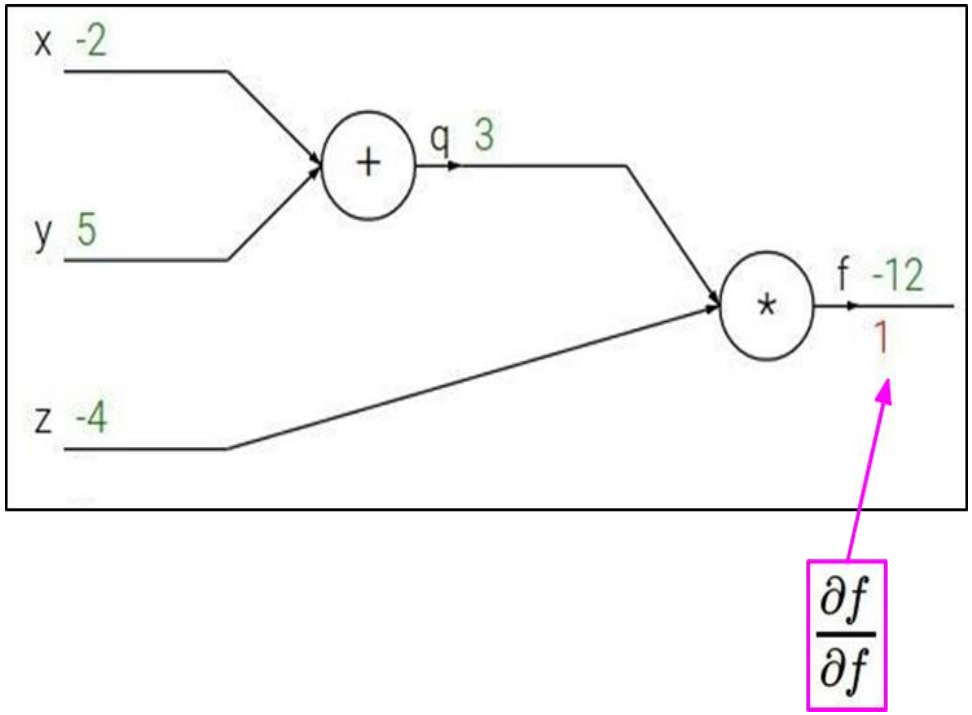
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backpropagation: a simple example

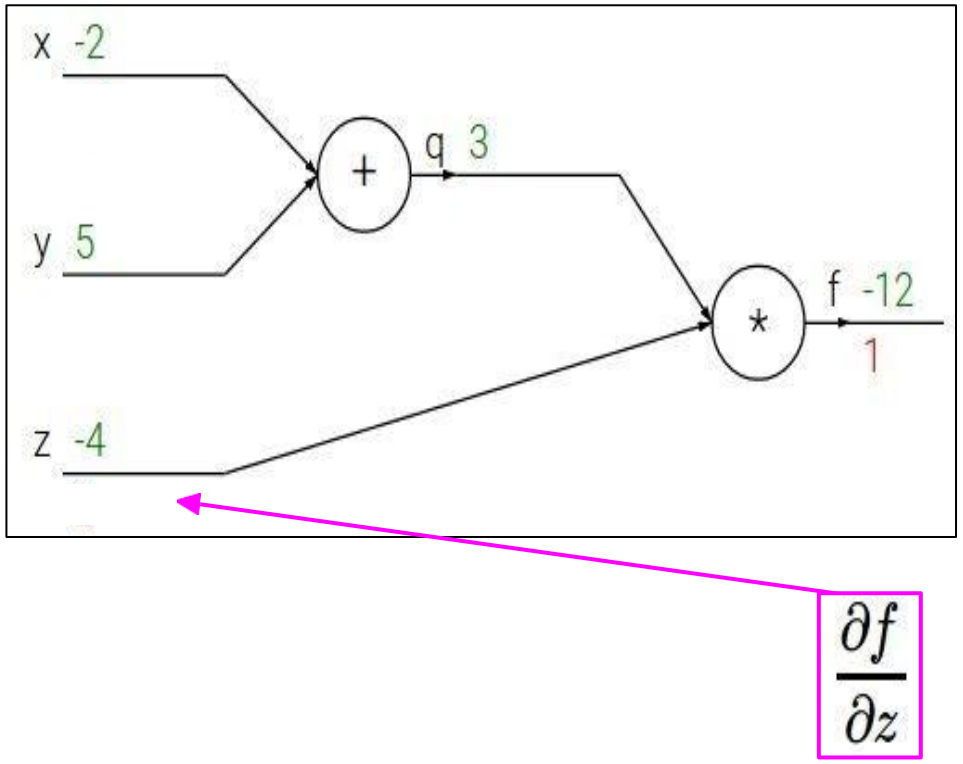
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backpropagation: a simple example

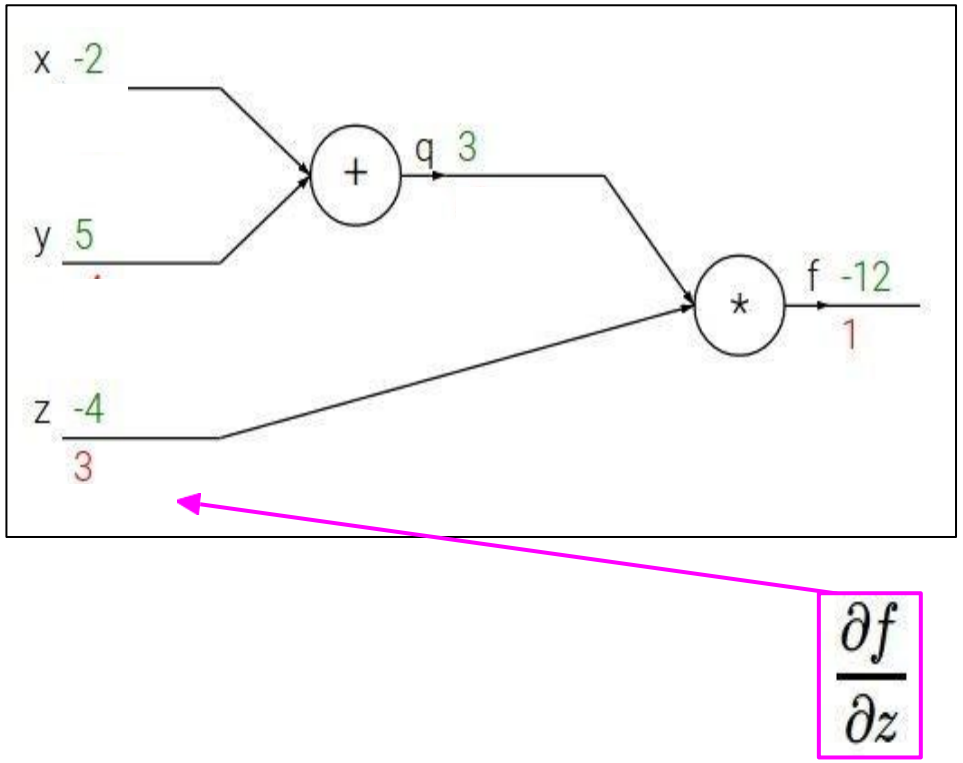
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backpropagation: a simple example

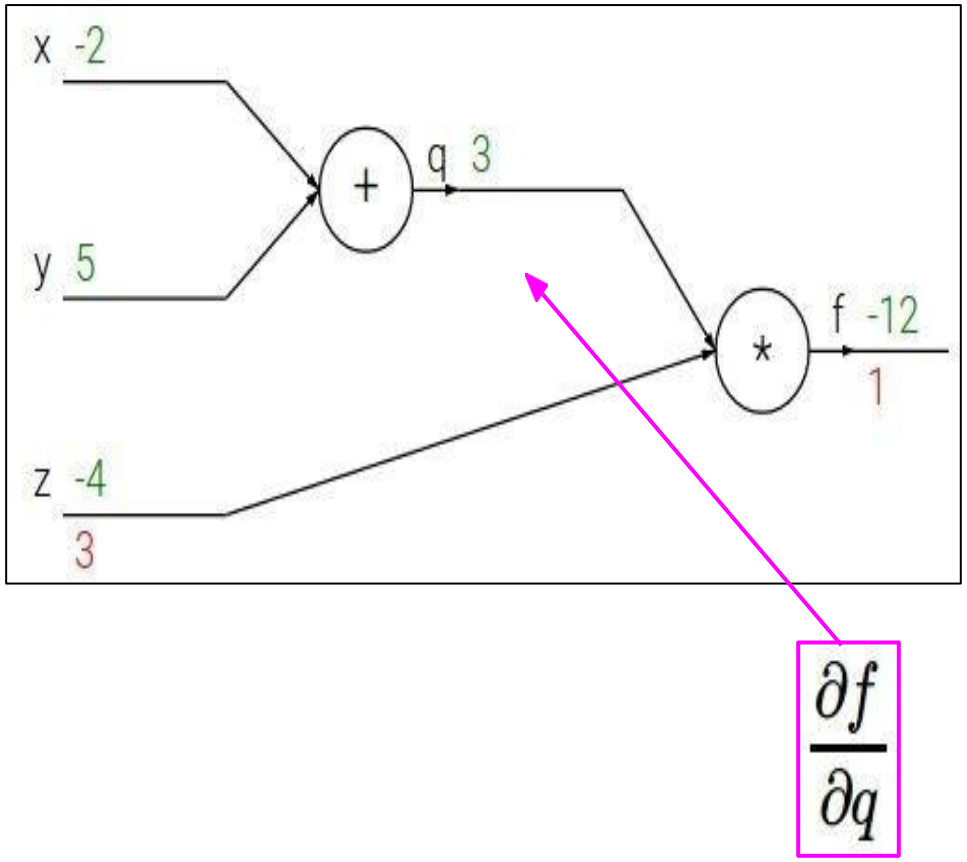
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backpropagation: a simple example

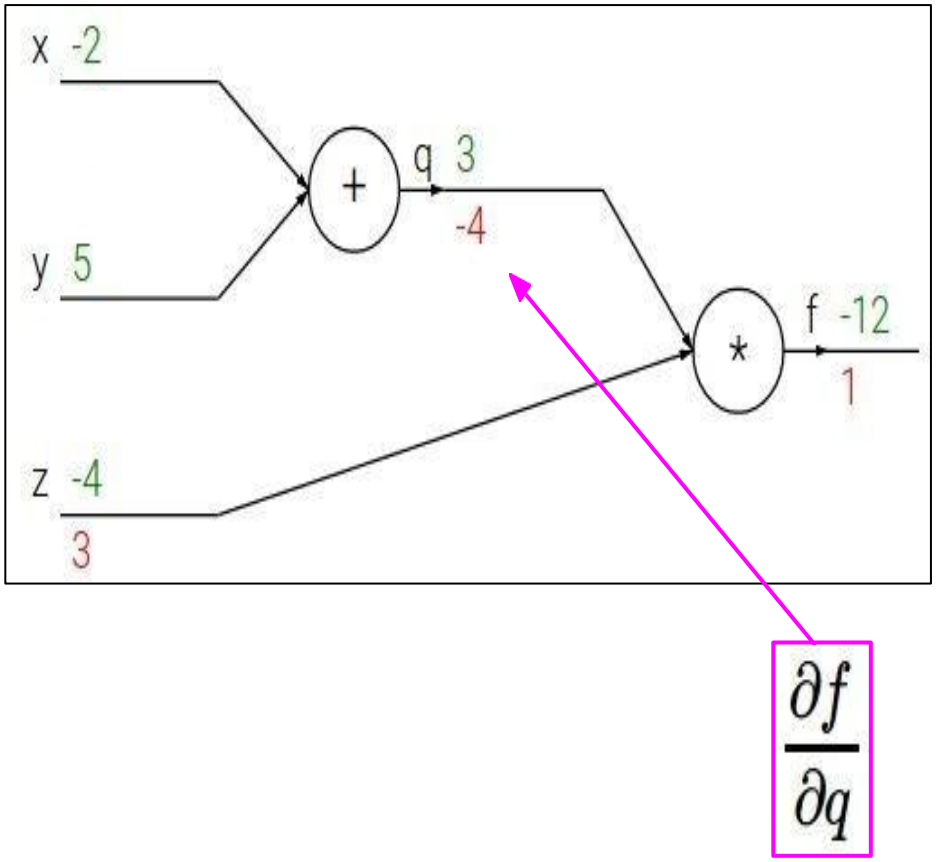
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backpropagation: a simple example

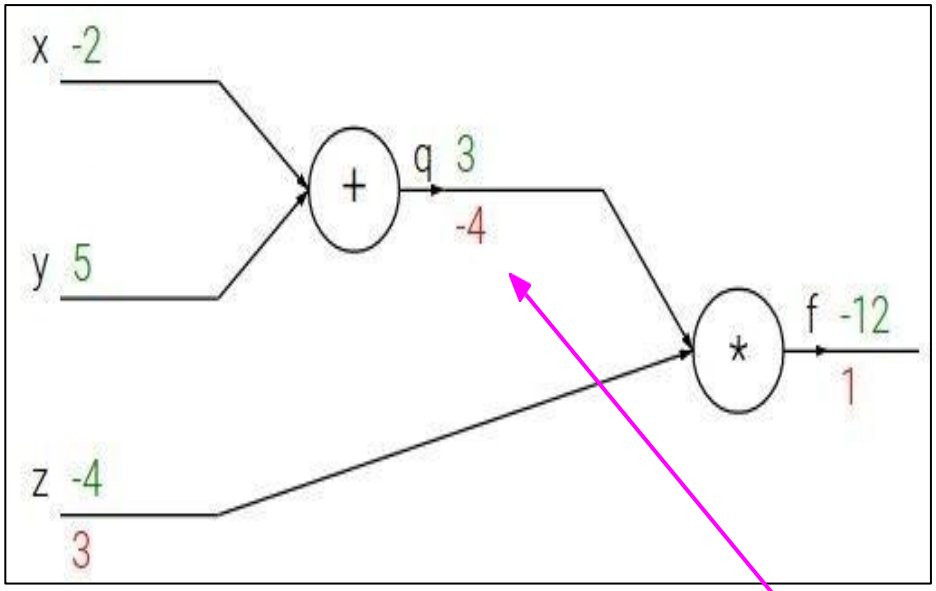
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Chain rule:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y}$$

$\frac{\partial f}{\partial q}$

Upstream gradient

Local gradient

Backpropagation: a simple example

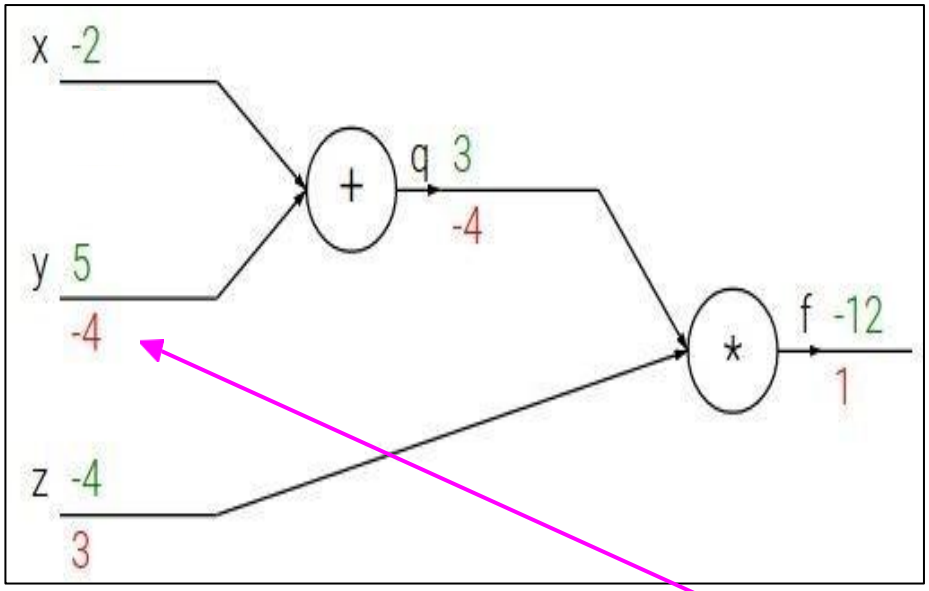
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Chain rule:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y}$$

$\frac{\partial f}{\partial y}$

Upstream gradient

Local gradient

Backpropagation: a simple example

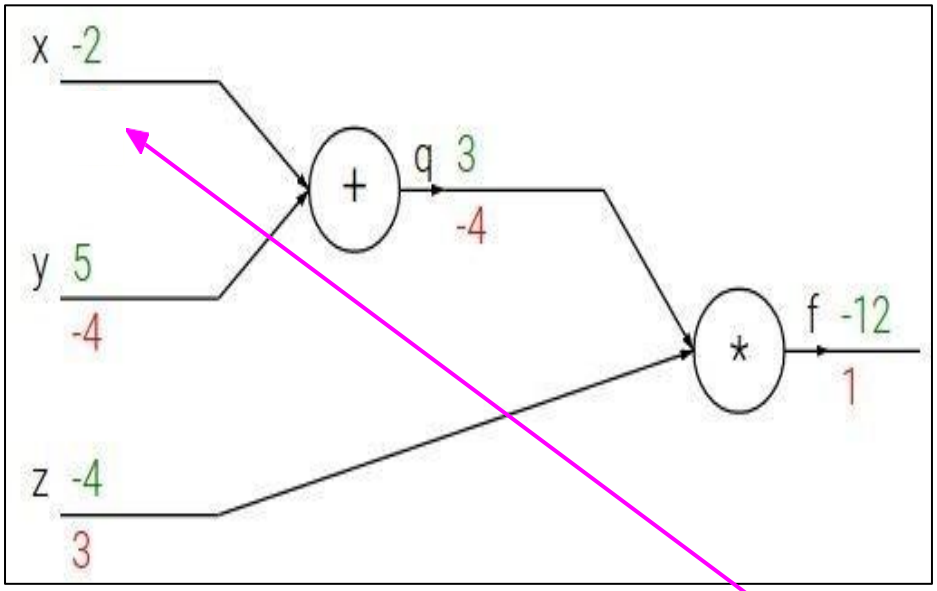
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Chain rule:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y}$$

Upstream gradient

Local gradient

$\frac{\partial f}{\partial x}$

Backpropagation: a simple example

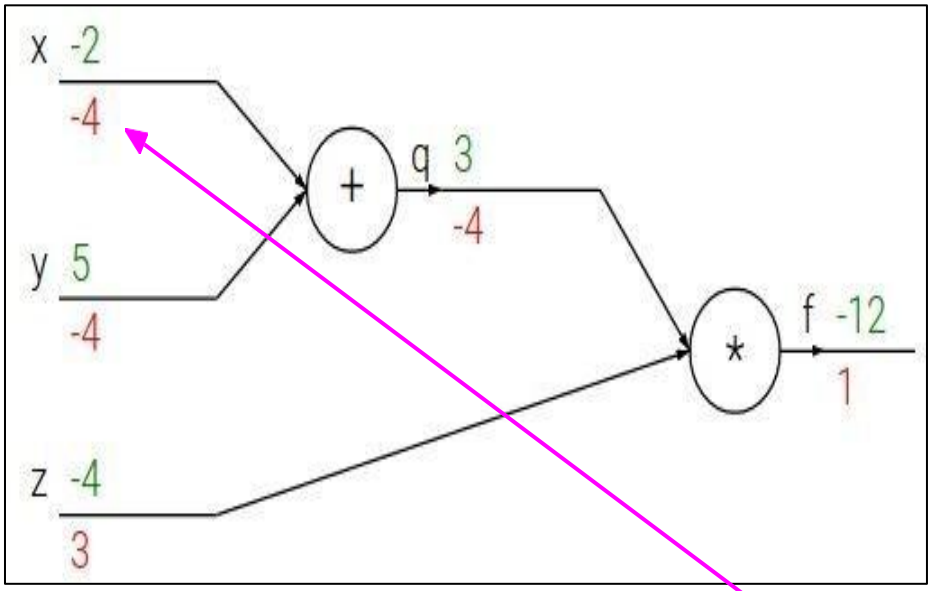
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



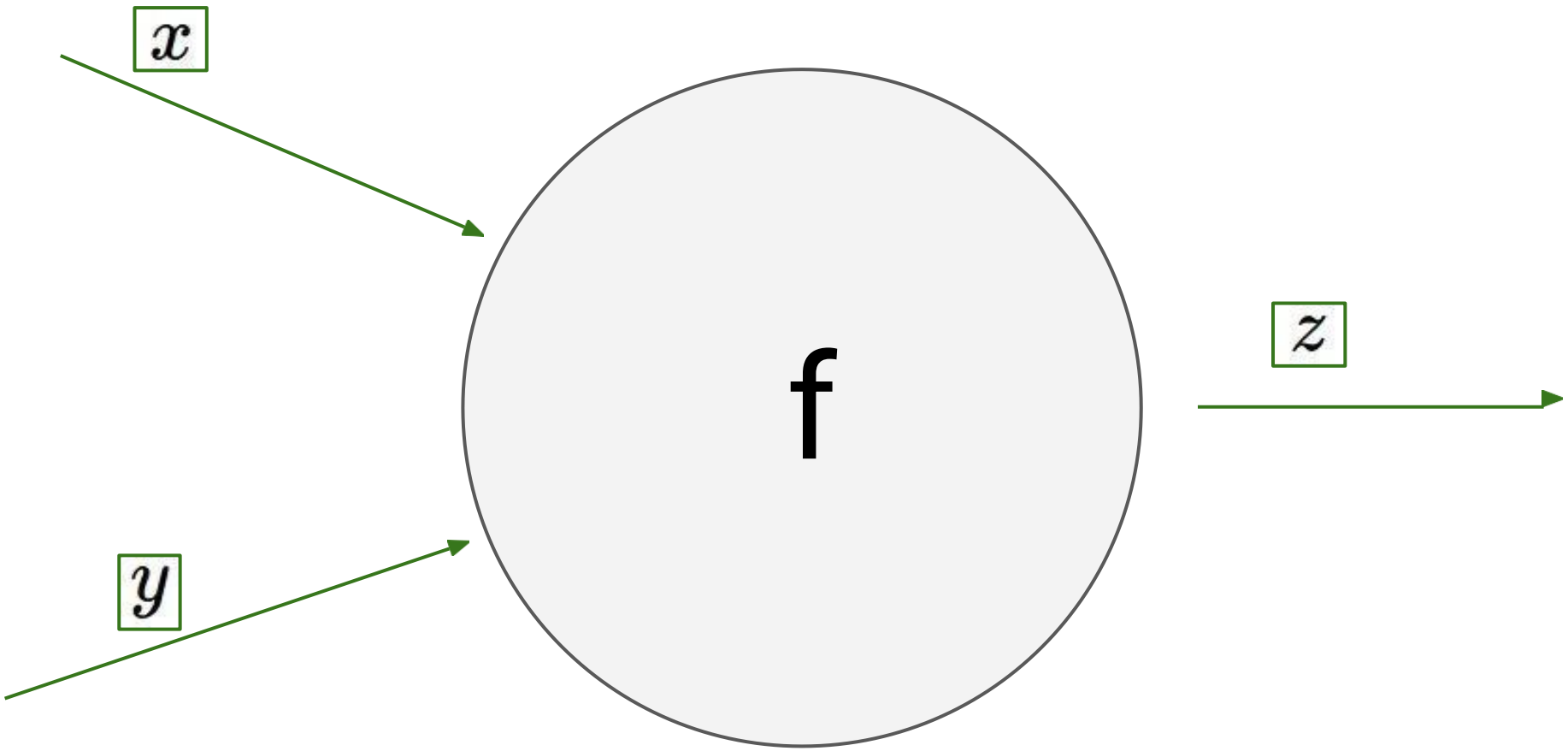
Chain rule:

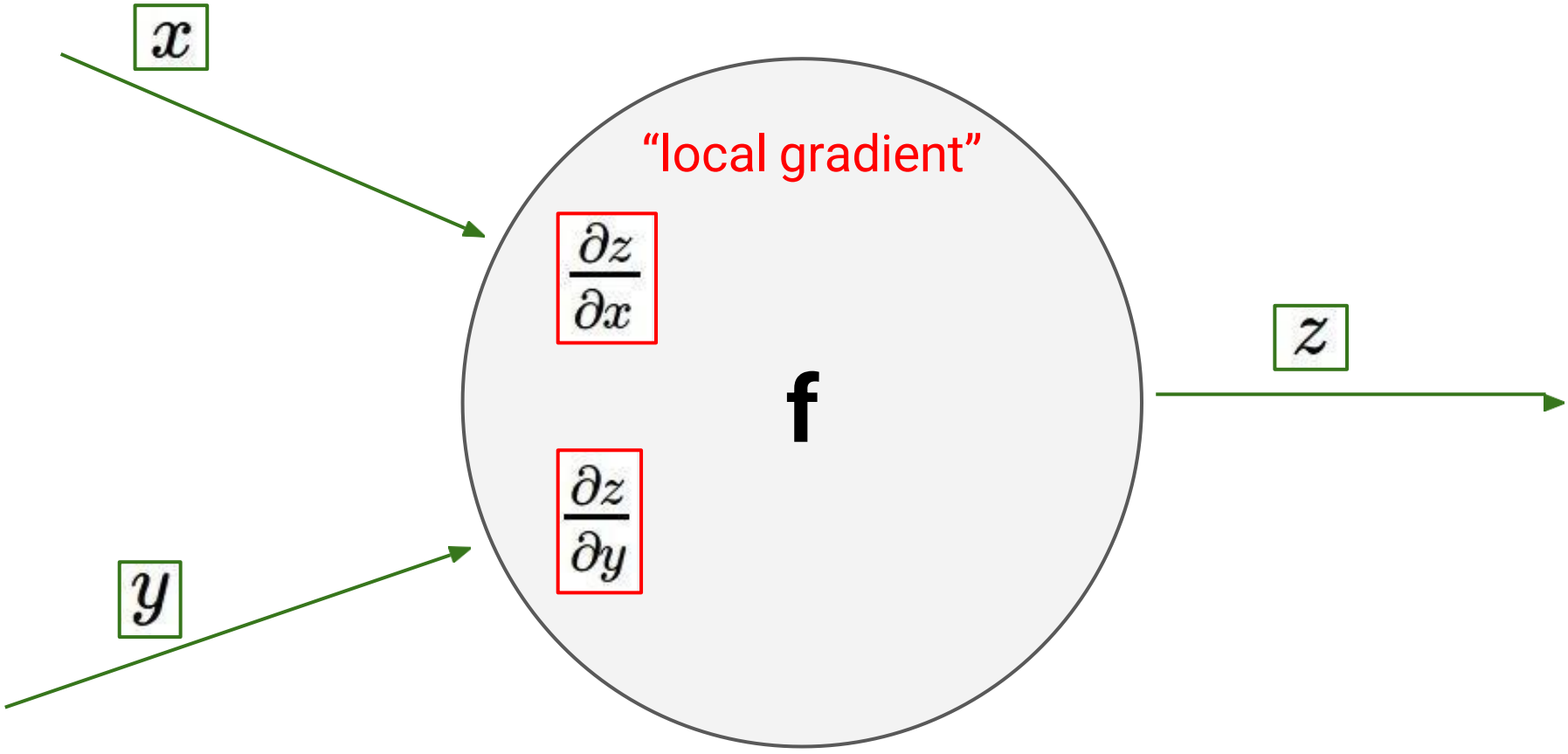
$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y}$$

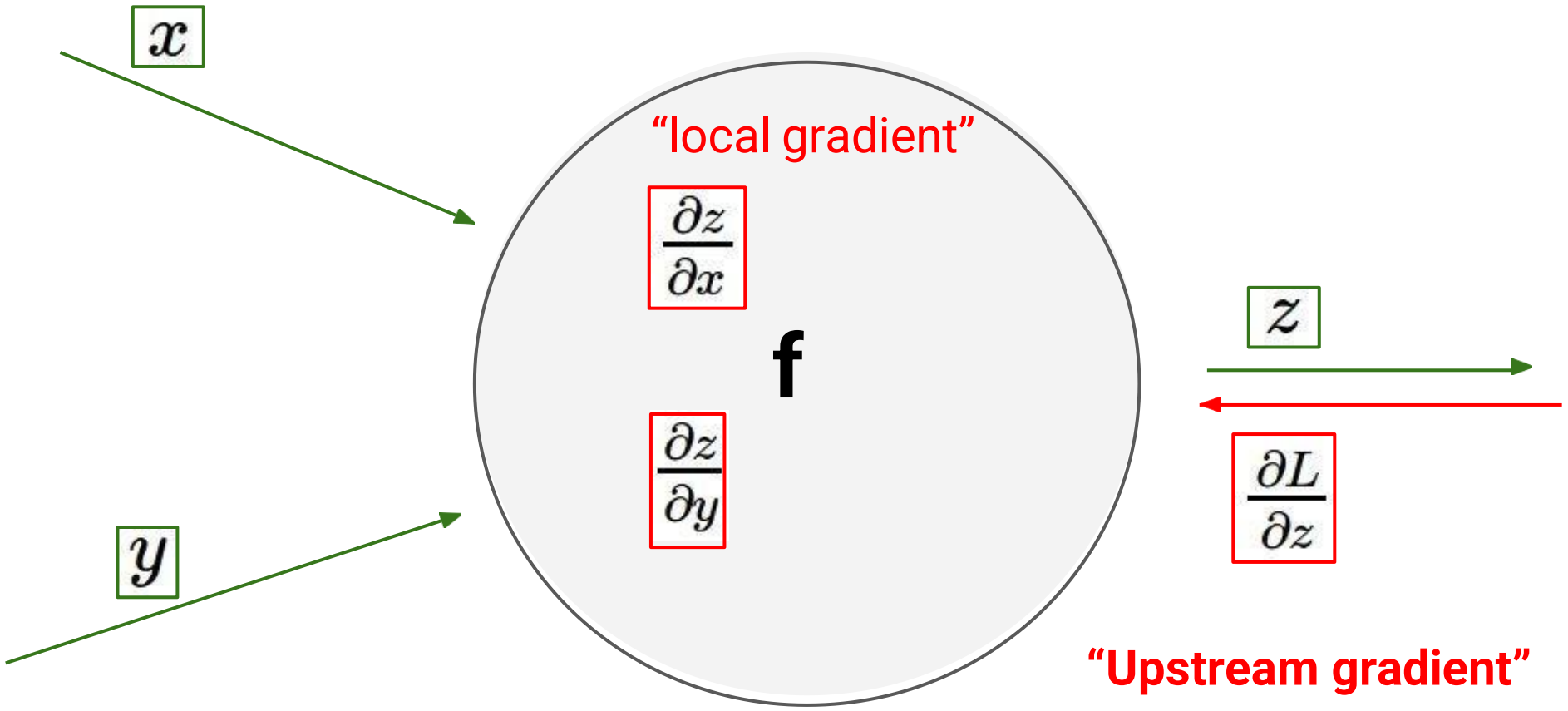
Upstream gradient

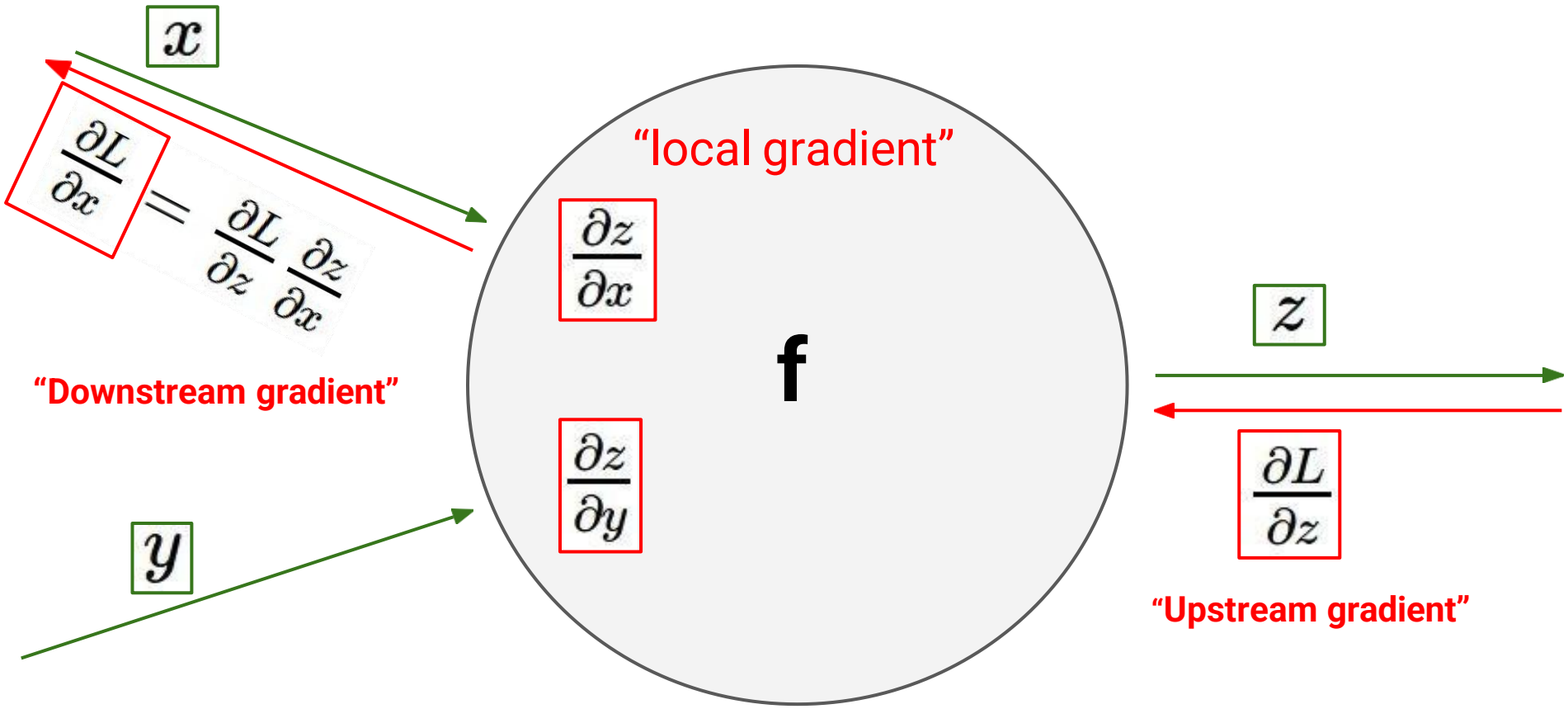
Local gradient

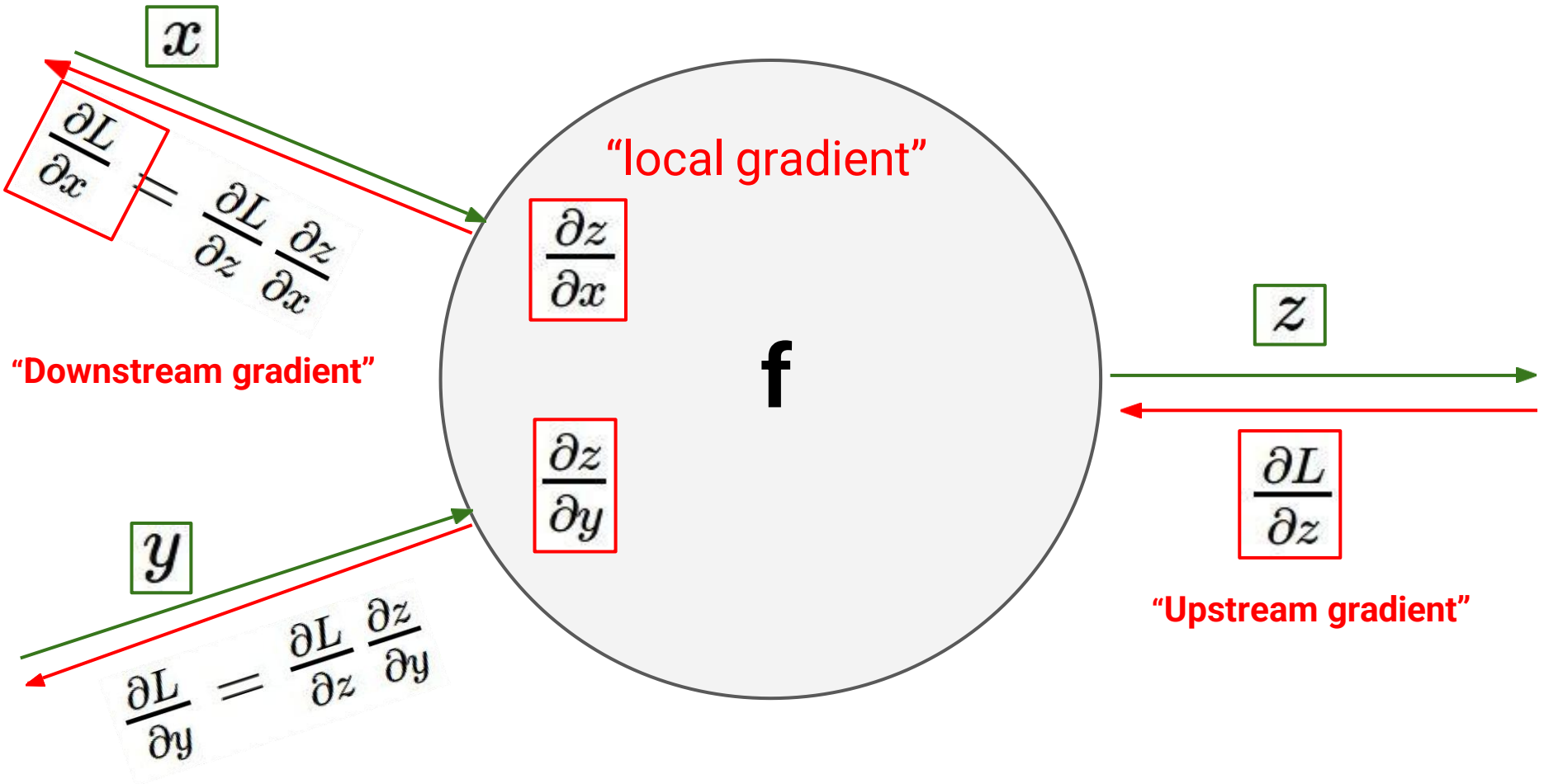
$\frac{\partial f}{\partial x}$

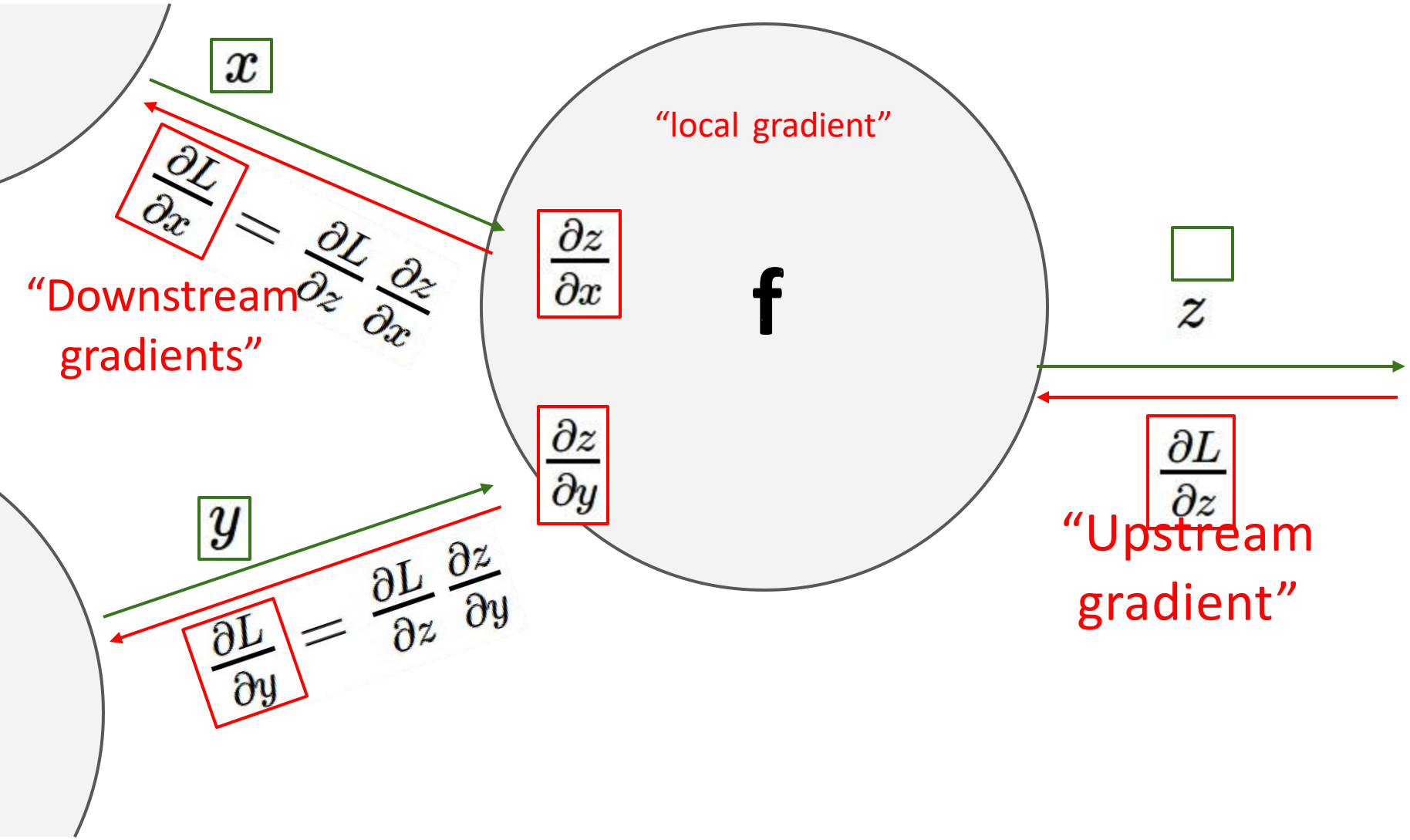

x
y
f
z


x
y
“local gradient”
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
f
z


x
y
"local gradient"
$\frac{\partial z}{\partial x}$
f
$\frac{\partial z}{\partial y}$
z
$\frac{\partial L}{\partial z}$
"Upstream gradient"


x
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"Downstream gradient"
"local gradient"
$\frac{\partial z}{\partial x}$
f
$\frac{\partial z}{\partial y}$
y
z
$\frac{\partial L}{\partial z}$
"Upstream gradient"


x
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"Downstream gradient"
"local gradient"
$\frac{\partial z}{\partial x}$
f
$\frac{\partial z}{\partial y}$
z
$\frac{\partial L}{\partial z}$
"Upstream gradient"
y
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$

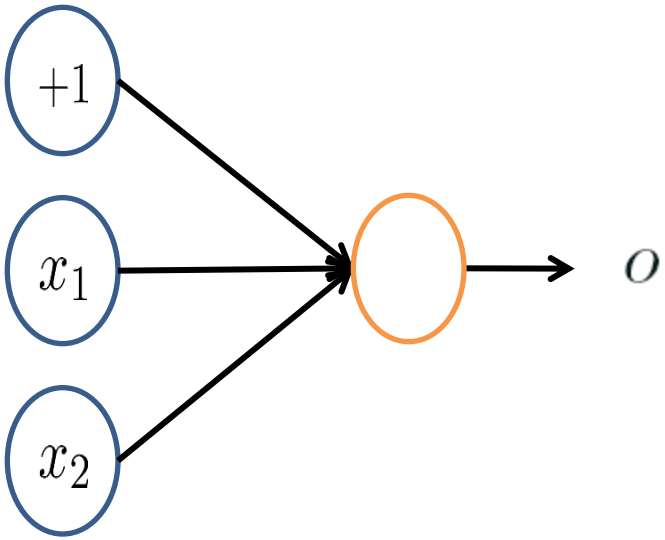

x
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"Downstream gradients"
"local gradient"
$\frac{\partial z}{\partial x}$
f
$\frac{\partial z}{\partial y}$
z
$\frac{\partial L}{\partial z}$
"Upstream gradient"
y
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$

Self Study

Simple example: AND

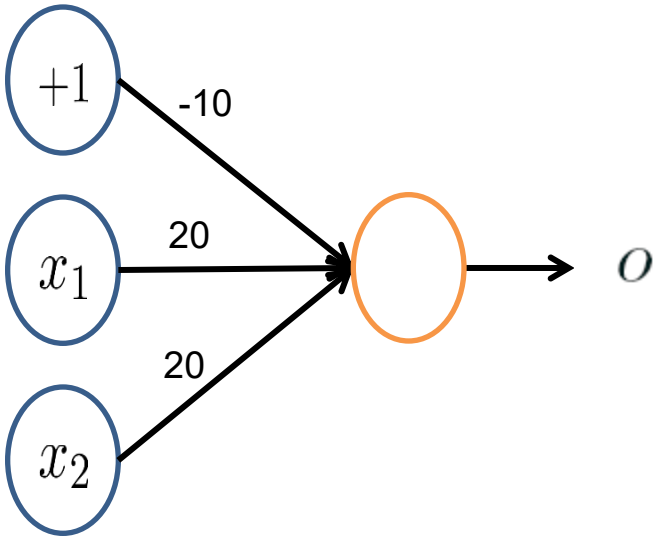
$x_1, x_2 \in \{0, 1\}$

$y = x_1 \text{ AND } x_2$



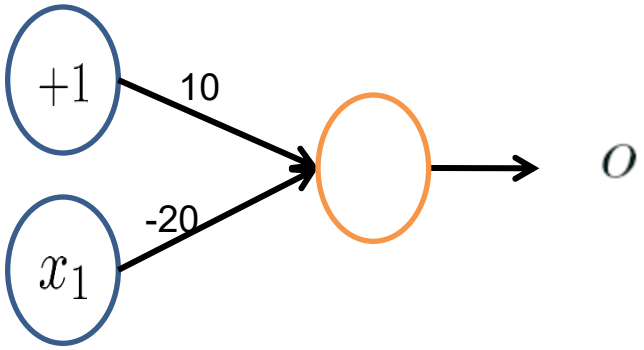
x_1	x_2	o
0	0	
0	1	
1	0	
1	1	

Example: OR function



x_1	x_2	o
0	0	
0	1	
1	0	
1	1	

Negation:

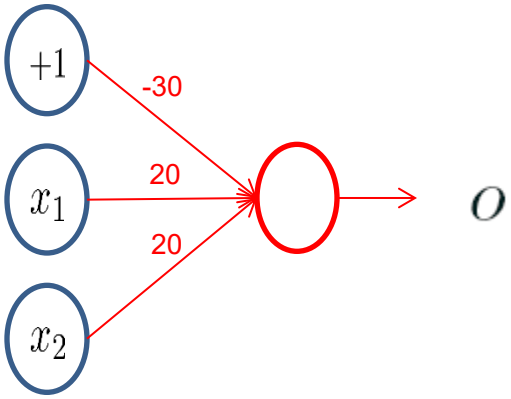


x_1	o
0	
1	

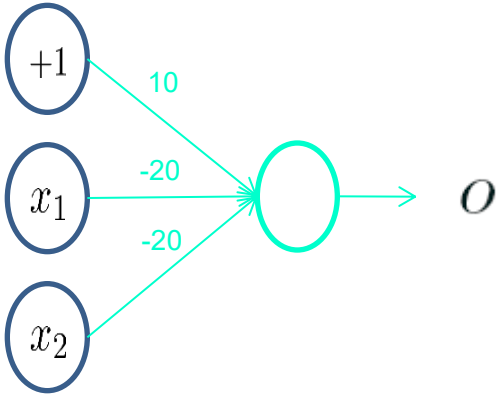
$o = g(10 - 20x_1)$

$(\text{NOT } x_1) \text{ AND } (\text{NOT } x_2)$

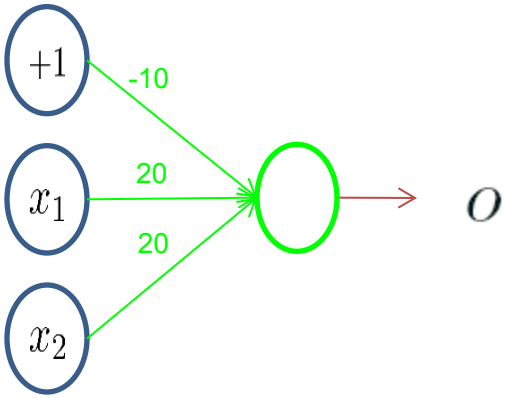
Putting it together: x_1 XNOR x_2



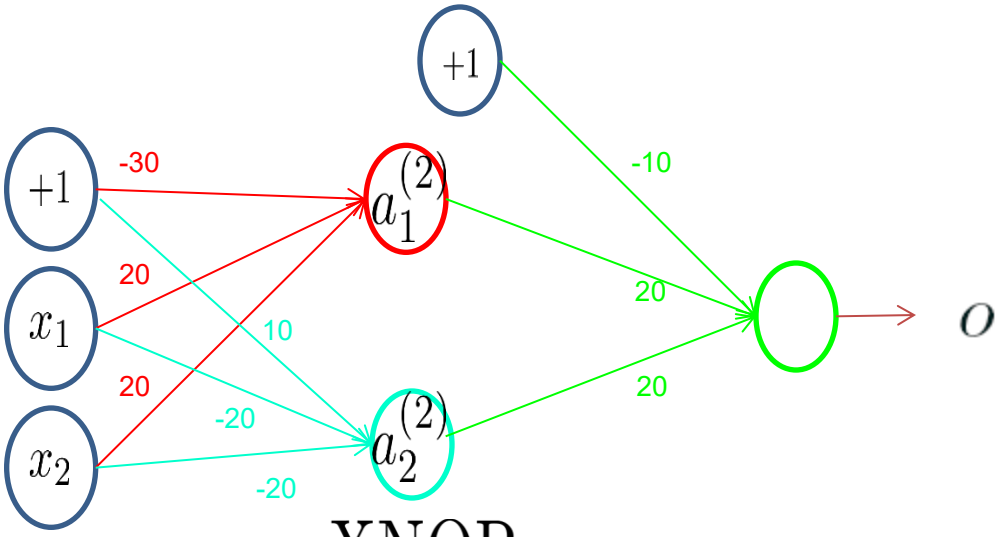
x_1 AND x_2



(NOT x_1) AND (NOT x_2)



x_1 OR x_2



x_1 XNOR x_2

x_1	x_2	$a_1^{(2)}$	$a_2^{(2)}$	o
0	0			
0	1			
1	0			
1	1			

Training Parametric Model

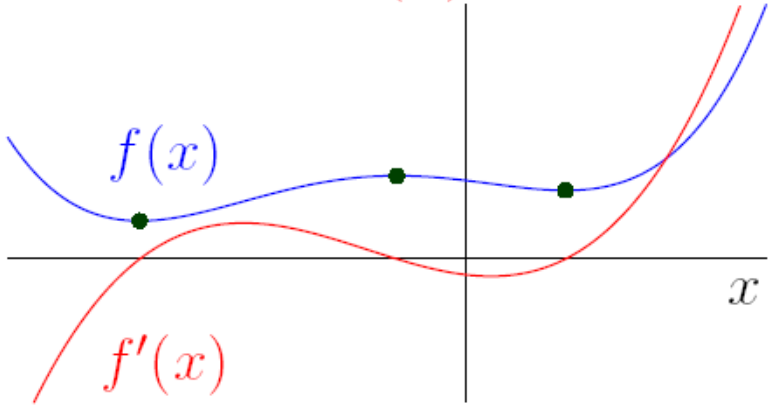
- We have seen how to train single layer step perceptron and linear perceptrons
- But how do we train multilayer non-linear perceptrons?
- More generally how do we train any parametric model $\boldsymbol{f}(\boldsymbol{x}|\boldsymbol{w})$?
- Given data $\mathcal{D} = \left\{(\boldsymbol{x}_k, \boldsymbol{y}_k)\right\}_{k=1}^{P}$ we can seek to minimise the mean squared error

$$E(\boldsymbol{w}|\mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{(\boldsymbol{x}_k, y_k) \in \mathcal{D}} \left(y_k - f(\boldsymbol{x}_k|\boldsymbol{w})\right)^2$$

Minimizing Error

- To minimise $E(\boldsymbol{w}|\mathcal{D})$, we want to move in the direction that decreases $E(\boldsymbol{w}|\mathcal{D})$
- The gradient $\nabla E(\boldsymbol{w}|\mathcal{D})$ is the direction of steepest increase of $E(\boldsymbol{w}|\mathcal{D})$
- Given a one dimensional function $f(x)$ the gradient is given by

$$f'(x) = \frac{\mathrm{d}f(x)}{\mathrm{d}x}$$



Least Squares Gradient

- Recall, the mean squared error is

$$E(\boldsymbol{w}|\mathcal{D}) = \frac{1}{P}\sum_{k=1}^{P}\left(f(\boldsymbol{x}_k|\boldsymbol{w}) - y_k\right)^2$$

- Its gradient is

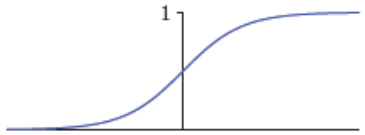
$$\boldsymbol{\nabla} E(\boldsymbol{w}|\mathcal{D}) = \frac{2}{P}\sum_{k=1}^{P}\left(f(\boldsymbol{x}_k|\boldsymbol{w}) - y_k\right)\,\boldsymbol{\nabla} f(\boldsymbol{x}_k|\boldsymbol{w}) = \frac{2}{P}\sum_{k=1}^{P}\delta_k\,\boldsymbol{\nabla} f(\boldsymbol{x}_k|\boldsymbol{w})$$

- where $f(\boldsymbol{x}_k|\boldsymbol{w}) - y_k = \delta_k$ is the error for example k

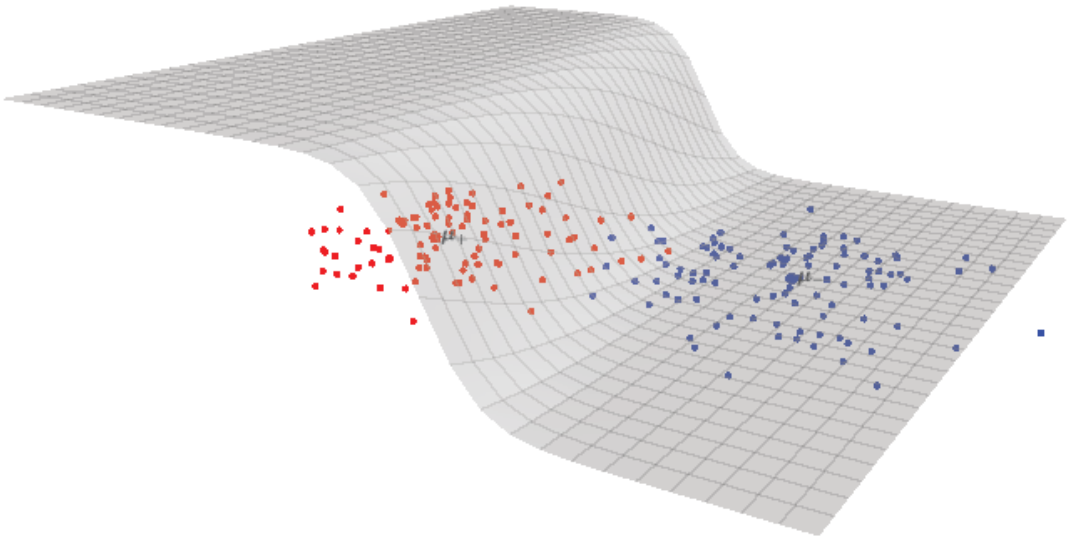
Single Layer Perceptron

- Before considering a MLP let us first consider the simpler case of a single layer perceptron, e.g.

$$f(\boldsymbol{x}|\boldsymbol{w}) = g\left(\boldsymbol{x}^{\mathsf{T}}\boldsymbol{w}\right), \qquad g(V) = \frac{1}{1+\mathrm{e}^{-V}}$$



- In the input space



Single layer Perceptrons

- For a single-layer perceptron $f(\boldsymbol{x}|\boldsymbol{w}) = g(\boldsymbol{x}^\mathsf{T}\boldsymbol{w})$

$$\begin{aligned}\boldsymbol{\nabla}_w g(\boldsymbol{x}^\mathsf{T}\boldsymbol{w}) &= g'(\boldsymbol{x}^\mathsf{T}\boldsymbol{w})\,\boldsymbol{\nabla}_w\left(\boldsymbol{x}^\mathsf{T}\boldsymbol{w}\right)\\ &= g'(\boldsymbol{x}^\mathsf{T}\boldsymbol{w})\,\boldsymbol{x}\end{aligned}$$

(using the chain rule)

- The gradient of the mean squared error for a **single-layer** perceptron is

$$\boldsymbol{\nabla}_w E(\boldsymbol{w}|\mathcal{D}) = \frac{2}{P}\sum_{k=1}^{P}\delta_k\; g'(\boldsymbol{x}_k^\mathsf{T}\boldsymbol{w})\,\boldsymbol{x}_k$$

Different Response Functions

- For the linear perceptron $g(V) = V$ so $g'(V) = 1$
- For the logistic perceptron

$$g(V) = \frac{1}{1 + \mathrm{e}^{-V}}$$

$$g'(V) = \frac{\mathrm{e}^{-V}}{(1 + \mathrm{e}^{-V})^2} = g(V)\,(1 - g(V))$$

- For the $g(V) = \tanh(V)$

$$g'(V) = 1 - \tanh^2(V) = 1 - g^2(V)$$

Learning a Logistic Perceptron

- For a single-layer *logistic* perceptron the gradient in the error is

$$\nabla_{\boldsymbol{w}} E(\boldsymbol{w}|\mathcal{D}) = \frac{2}{P} \sum_{k=1}^{P} \delta_k \, o_k (1 - o_k) \, \boldsymbol{x}_k$$

$o_k = g(\boldsymbol{x}_k^{\mathsf{T}} \boldsymbol{w}) = f(\boldsymbol{x}_k^{\mathsf{T}} | \boldsymbol{w})$

- We can iteratively reduce the error by changing the weights in the direction opposite the gradient

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \eta \nabla_{\boldsymbol{w}} E(\boldsymbol{w}^{(k)} | \mathcal{D})$$

where $\boldsymbol{w}^{(k)}$ are weights at iteration k

Back Propagation

- To train a MLP we have to adjust the weights of the nodes at each layer
- We can update the weights in the direction opposite the gradient

$$\boldsymbol{w}^{new} = \boldsymbol{w} - \eta \boldsymbol{\nabla} E(\boldsymbol{w}|\mathcal{D})$$

- To compute the gradient we just use the chain rule
- The update is always proportional to the error so this is called

back-propagation of errors

Back Propagation

- Gradient of error function

$$\nabla E(\boldsymbol{w}|\mathcal{D}) = \frac{2}{P}\sum_{k=1}^{P}\delta_k \nabla f(\boldsymbol{x}_k|\boldsymbol{w})$$

$$f(\boldsymbol{x}_k|\boldsymbol{w}) = g\left(\sum_i \bar{w}_i g(\boldsymbol{w}_i^{\mathsf{T}}\boldsymbol{x})\right)$$

- Gradients of MLP output $f(\boldsymbol{x}_k|\boldsymbol{w})$

$$\nabla_{\bar{\boldsymbol{w}}} f(\boldsymbol{x}_k|\boldsymbol{w}) = g'\left(\sum_i \bar{w}_i g(\boldsymbol{w}_i^{\mathsf{T}}\boldsymbol{x})\right)\ \boldsymbol{y}, \qquad y_i = g(\boldsymbol{w}_i^{\mathsf{T}}\boldsymbol{x})$$

$$\nabla_{\boldsymbol{w}_i} f(\boldsymbol{x}_k|\boldsymbol{w}) = g'\left(\sum_j \bar{w}_j g(\boldsymbol{w}_i^{\mathsf{T}}\boldsymbol{x})\right)\ \bar{w}_i\ g'(\boldsymbol{w}_i^{\mathsf{T}}\boldsymbol{x})\ \boldsymbol{x}$$

Back Propagation

Initialize all weights to small random numbers
Until convergence, Do
For each training example, Do

1. Input it to network and compute network outputs
2. For each output unit k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

3. For each hidden unit h

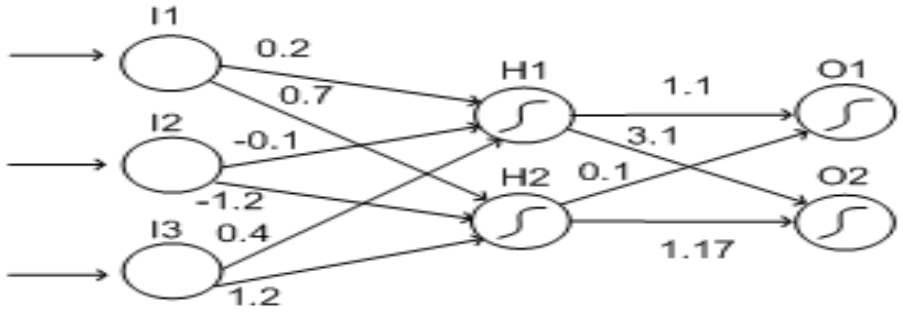
$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k}\delta_k$$

4. Update each network weight $w_{i,j}$

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

where $\Delta w_{i,j} = \eta \delta_j x_{i,j}$

A Worked Example:



Input units		Hidden units			Output units		
Unit	Output	Unit	Weighted Sum Input	Output	Unit	Weighted Sum Input	Output
I1	10	H1	7	0.999	O1	1.0996	0.750
I2	30	H2	-5	0.0067	O2	3.1047	0.957
I3	20						

Propagated the values (10,30,20) through the network

Suppose now that the target categorization for the example was the one associated with O1(using a learning rate of $\eta = 0.1$)

the target output for O1 was 1, and the target output for O2 was 0

$t1(E) = 1;$ $t2(E) = 0;$ $o1(E) = 0.750;$ $o2(E) = 0.957$

error values for the output units O1 and O2

$\delta O1 = o1(E)(1 - o1(E))(t1(E) - o1(E)) = 0.750(1-0.750)(1-0.750) = 0.0469$

$\delta O2 = o2(E)(1 - o2(E))(t2(E) - o2(E)) = 0.957(1-0.957)(0-0.957) = -0.0394$

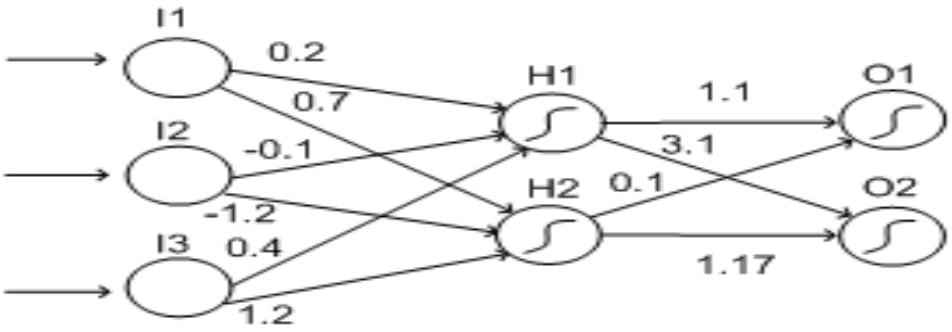
A Worked Example:

- To propagate this information backwards to the hidden nodes H1 and H2
  - Multiply the error term for O1 by the weight from H1 to O1, then add this to the multiplication of the error term for O2 and the weight between H1 and O2, (1.1*0.0469) + (3.1*-0.0394) = -0.0706

  - δ_{H1} = -0.0706*(0.999 * (1-0.999)) = -0.0000705
  - Similarly for H2: (0.1*0.0469)+(1.17*-0.0394) = -0.0414
  - δ_{H2} -0.0414 * (0.067 * (1-0.067)) = -0.00259

$$\delta_{H_k} = h_k(E)(1 - h_k(E)) \sum_{i \in outputs} w_{ki}\delta_{O_i}$$

A Worked Example:



Input unit	Hidden unit	η	δ_H	x_i	$\Delta = \eta * \delta_H * x_i$	Old weight	New weight
I1	H1	0.1	-0.0000705	10	-0.0000705	0.2	0.1999295
I1	H2	0.1	-0.00259	10	-0.00259	0.7	0.69741
I2	H1	0.1	-0.0000705	30	-0.0002115	-0.1	-0.1002115
I2	H2	0.1	-0.00259	30	-0.00777	-1.2	-1.20777
I3	H1	0.1	-0.0000705	20	-0.000141	0.4	0.39999
I3	H2	0.1	-0.00259	20	-0.00518	1.2	1.1948

Hidden unit	Output unit	η	δ_O	$h_i(E)$	$\Delta = \eta * \delta_O * h_i(E)$	Old weight	New weight
H1	O1	0.1	0.0469	0.999	0.000469	1.1	1.100469
H1	O2	0.1	-0.0394	0.999	-0.00394	3.1	3.0961
H2	O1	0.1	0.0469	0.0067	0.00314	0.1	0.10314
H2	O2	0.1	-0.0394	0.0067	-0.0000264	1.17	1.16998

When to Learn

- There are two classic alternative strategies to learning
- **On-line learning**
  - ⋆ learn after every input pattern $(\boldsymbol{x}_k, y_k)$
  - ⋆ minimise error for single inputs
- **Batch learning**
  - ⋆ learn after seeing all the inputs
  - ⋆ minimise error on all inputs

Online Learning

- Here we learn after each input example $(\boldsymbol{x}_k, y_k)$

$$\boldsymbol{w}^{new} = \boldsymbol{w} - \eta \nabla E(\boldsymbol{w}|(\boldsymbol{x}_k, y_k))$$

- Where the learning error is now one example

$$E(\boldsymbol{w}|(\boldsymbol{x}_k, y_k)) = \big(f(\boldsymbol{x}_k|\boldsymbol{w}) - y_k\big)^2$$

- The input pattern are chosen in a random order
- Learning goes through many rounds of re-presenting the data

Batch Learning

- In batch learning we use all the data at once
- The full learning error is

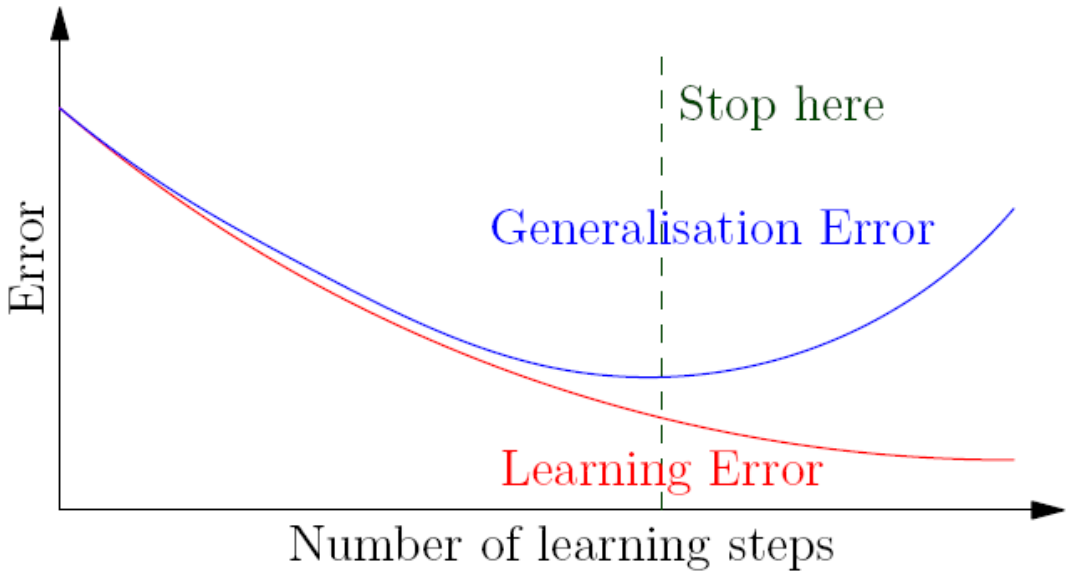
$$E(\boldsymbol{w}|\mathcal{D}) = \sum_{(\boldsymbol{x}_k, y_k) \in \mathcal{D}} E(\boldsymbol{w}|(\boldsymbol{x}_k, y_k))$$

- The gradient is computed for all training examples

$$\boldsymbol{\nabla} E(\boldsymbol{w}|\mathcal{D}) = \sum_{k=1}^{P} \boldsymbol{\nabla} E(\boldsymbol{w}|(\boldsymbol{x}_k, y_k))$$

Early Stopping

- As the number of learning steps increase the learning error falls but after some time the generalisation error tends to rise due to over-fitting



Self Study Examples

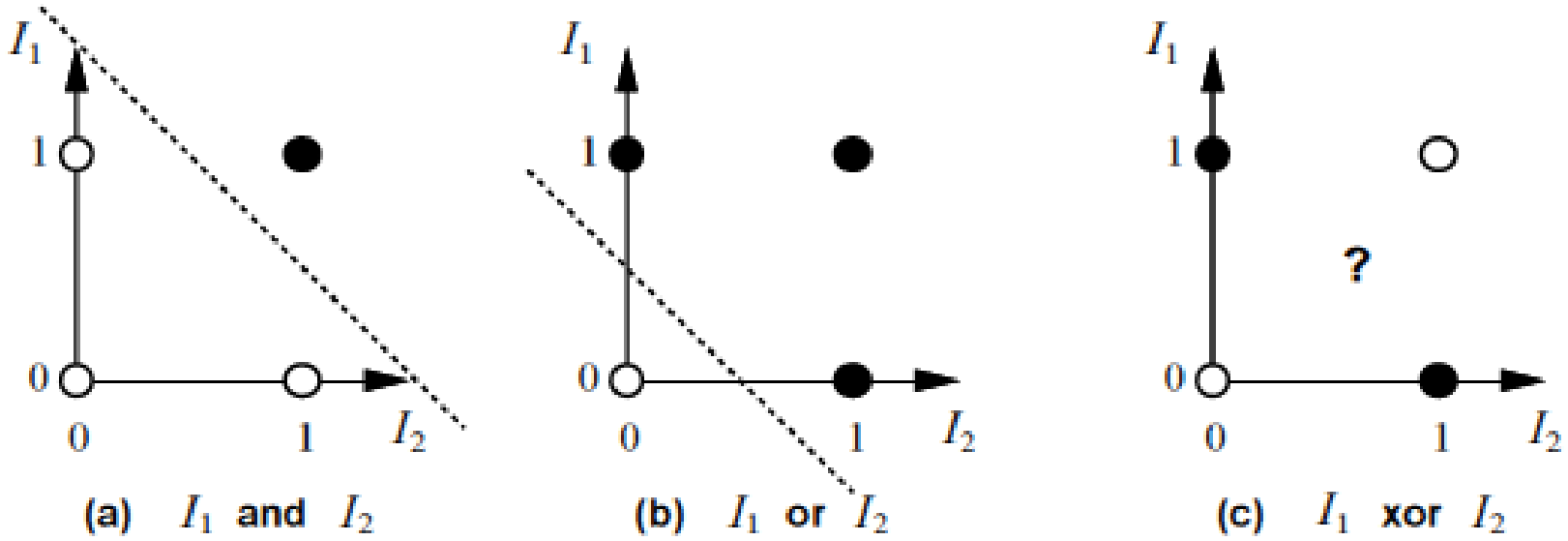
XOR Example

Linear Separation

Can AND, OR and NOT be represented?
Is it possible to represent every boolean function by simply combining these?

Every Boolean function can be composed using AND, OR and NOT (or even only NAND).

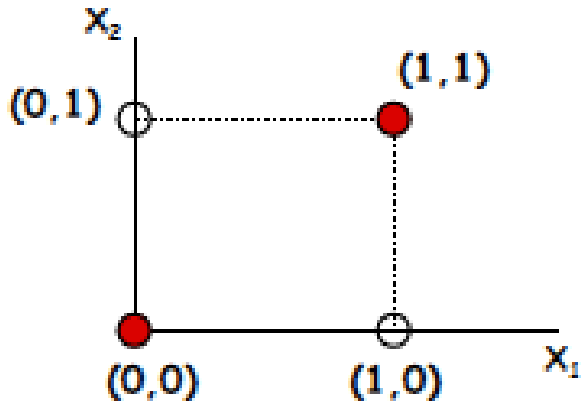
Linear Separation



How can we learn XOR function?

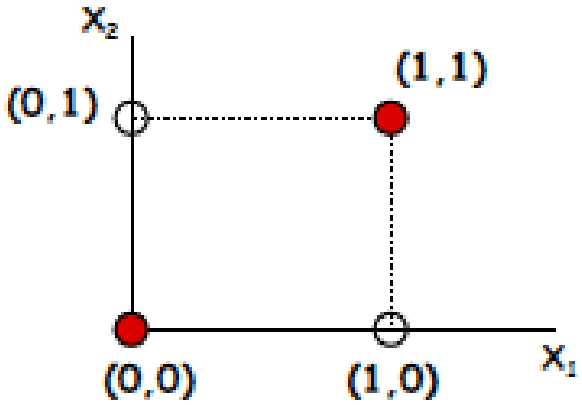
Linear Separation

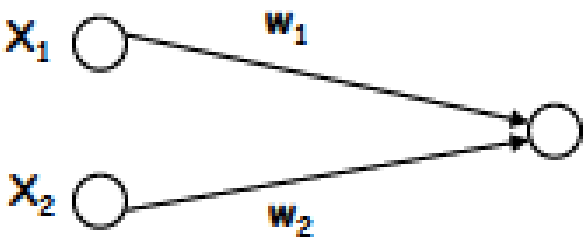
X1	X2	XOR
0	0	0
1	0	1
0	1	1
1	1	0



Linear Separation

X1	X2	XOR
0	0	0
1	0	1
0	1	1
1	1	0





$(1,0) \Rightarrow 1 : 1*w_1+0*w_2=1 \Rightarrow w_1=1$

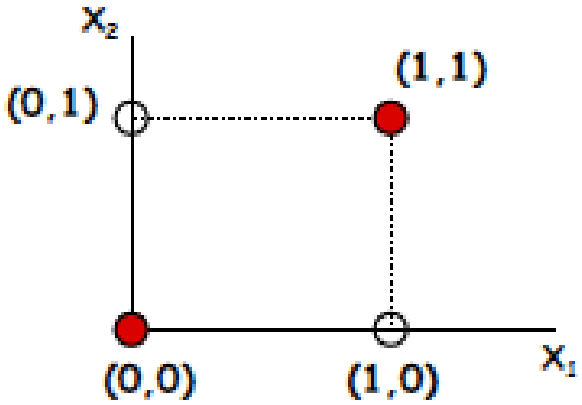
$(0,1) \Rightarrow 1 : 0*w_1+1*w_2=1 \Rightarrow w_2=1$

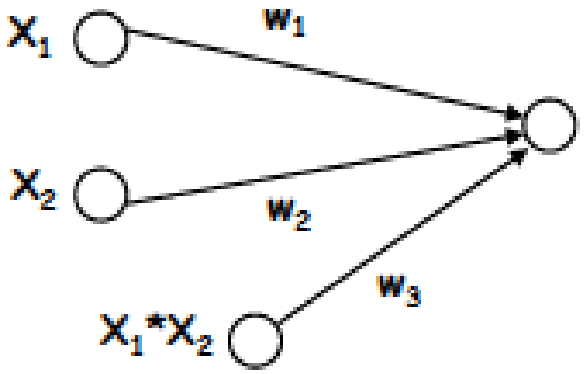
$(1,1) \Rightarrow 0 : 1*w_1+1*w_2=0 \Rightarrow$???

It is impossible to find the value of W_i to learn XOR

Linear Separation

X1	X2	X1*X2	XOR
0	0		0
1	0		1
0	1		1
1	1		0





$1*w_1+0*w_2+0*w_3=1 \Rightarrow w_1=1$
$0*w_1+1*w_2+0*w_3=1 \Rightarrow w_2=1$
$1*w_1+1*w_2+1*w_3=0 \Rightarrow w_3=-2$
$0*w_1+0*w_2+0*w_3=0$ vérification : ok

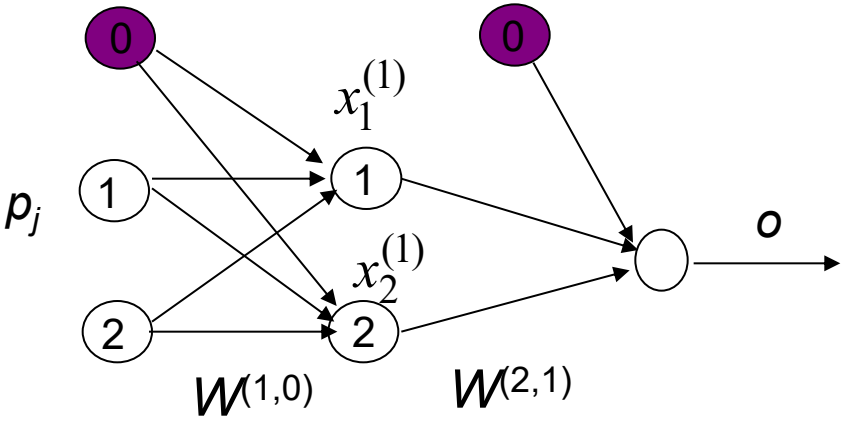
So, we learned W1, W2 and W3

Example, Back Propogation learning function XOR

Training samples (bipolar)

	in_1	in_2	d
P0	-1	-1	-1
P1	-1	1	1
P2	1	-1	1
P3	1	1	1

Network: 2-2-1 with thresholds (fixed output 1)



- Initial weights W(0)

$$w_1^{(1,0)} : (-0.5,\ 0.5,\ -0.5)$$
$$w_2^{(1,0)} : (-0.5,\ -0.5,\ 0.5)$$
$$w^{(2,1)} : (-1,\ 1,\ 1)$$

- Learning rate = 0.2
- Node function: hyperbolic tangent

$$g(x) = \tanh(x) = \frac{1 - e^{-x}}{1 + e^{-x}};$$
$$\lim_{x \to \pm\infty} g(x) = \pm 1$$
$$s(x) = \frac{1}{1 + e^{-x}};$$
$$g(x) = 2s(x) - 1$$
$$s'(x) = s(x)(1 - s(x))$$
$$g'(x) = 0.5(1 + g(x))(1 - g(x))$$

Present $P_0 = (1, -1, -1)$: $d_0 = -1$

Forward computing

$$net_1 = w_1^{(1,0)} p_0 = (-0.5,\ 0.5, -0.5)\ (1, -1, -1) = -0.5$$

$$net_2 = w_2^{(1,0)} p_0 = (-0.5, -0.5, 0.5)\ (1, -1, -1) = -0.5$$

$$x_1^{(1)} = g(net_1) = 2/(1 + e^{0.5}) - 1 = -0.24492$$

$$x_1^{(1)} = g(net_2) = 2/(1 + e^{0.5}) - 1 = -0.24492$$

$$net_o = w^{(2,1)} x^{(1)} = (-1, 1, 1)(1, -0.24492, -0.24492) = -1.48984$$

$$o = g(net_o) = -0.63211$$

Error back propogating

$$l = d - o = -1 - (-0.63211) = -0.36789$$

$$\delta = l \cdot g'(net_o) = l \cdot (1 + g(net_o))(1 - g(net_o))$$
$$= -0.3679 \cdot (1 - 0.6321)(1 + 0.6321) = -0.2209$$

$$\mu_1 = \delta \cdot w_1^{(2,1)} \cdot g'(net_1)$$
$$= -0.2209 \cdot 1 \cdot (1 - 0.24492) \cdot (1 + 0.24492) = -0.20765$$

$$\mu_2 = \delta \cdot w_2^{(2,1)} \cdot g'(net_2)$$
$$= -0.2209 \cdot 1 \cdot (1 - 0.24492) \cdot (1 + 0.24492) = -0.20765$$

Weight update

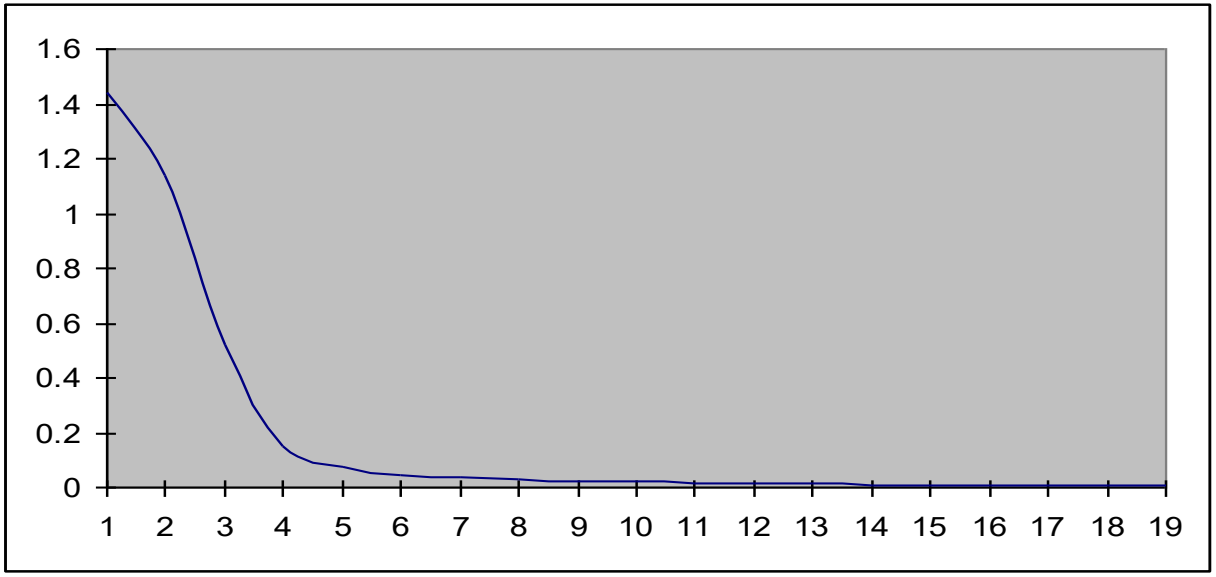
$$\Delta w^{(2,1)} = \eta \cdot \delta \cdot x^{(1)}$$
$$= 0.2 \cdot (-0.2209) \cdot (1, -0.2449, -0.2449) = (-0.0442, 0.0108, 0.0108)$$

$$w^{(2,1)} = w^{(2,1)} + \Delta w^{(2,1)} = (-1, 1, 1) + (-0.0442, 0.0108, 0.0108)$$
$$= (-0.5415, 1.0108, 1.0108)$$

$$\Delta w_1^{(1,0)} = \eta \cdot \mu_1 \cdot p_0 = 0.2 \cdot (-0.2077) \cdot (1, -1, -1) = (-0.0415, 0.0415, 0.0415)$$
$$\Delta w_2^{(1,0)} = \eta \cdot \mu_2 \cdot p_0 = 0.2 \cdot (-0.2077) \cdot (1, -1, -1) = (-0.0415, 0.0415, 0.0415)$$
$$w_1^{(1,0)} = w_1^{(1,0)} + \Delta w_1^{(1,0)} = (-0.5, 0.5, -0.5) + (-0.0415, 0.0415, 0.0415)$$
$$= (-0.5415, 0.5415, -0.4585)$$
$$w_2^{(1,0)} = w_2^{(1,0)} + \Delta w_2^{(1,0)} = (-0.5, -0.5, 0.5) + (-0.0415, 0.0415, 0.0415)$$
$$= (-0.5415, -0.4585, 0.5415)$$

$$\text{Error for } P_0 = l^2 \text{ reduced from } 0.135345 \text{ to } 0.102823$$

MSE reduction:
every 10 epochs



Output: every 10 epochs

epoch	1	10	20	40	90	140	190	d
P0	-0.63	-0.05	-0.38	-0.77	-0.89	-0.92	-0.93	**-1**
P1	-0.63	-0.08	0.23	0.68	0.85	0.89	0.90	**1**
P2	-0.62	-0.16	0.15	0.68	0.85	0.89	0.90	**1**
p3	-0.38	0.03	-0.37	-0.77	-0.89	-0.92	-0.93	**-1**
MSE	**1.44**	**1.12**	**0.52**	**0.074**	**0.019**	**0.010**	**0.007**	

After epoch 1

	$w_1^{(1,0)}$	$w_2^{(1,0)}$	$w^{(2,1)}$
init	(-0.5, 0.5, -0.5)	(-0.5, -0.5, 0.5)	(-1, 1, 1)
p0	-0.5415, 0.5415, -0.4585	-0.5415, -0.45845, 0.5415	-1.0442, 1.0108, 1.0108
p1	-0.5732, 0.5732, -0.4266	-0.5732, -0.4268, 0.5732	-1.0787, 1.0213, 1.0213
p2	-0.3858, 0.7607, -0.6142	-0.4617, -0.3152, 0.4617	-0.8867, 1.0616, 0.8952
p3	-0.4591, 0.6874, -0.6875	-0.5228, -0.3763, 0.4005	-0.9567, 1.0699, 0.9061

Epoch

13	-1.4018, 1.4177, -1.6290	-1.5219, -1.8368, 1.6367	0.6917, 1.1440, 1.1693
40	-2.2827, 2.5563, -2.5987	-2.3627, -2.6817, 2.6417	1.9870, 2.4841, 2.4580
90	-2.6416, 2.9562, -2.9679	-2.7002, -3.0275, 3.0159	2.7061, 3.1776, 3.1667
190	-2.8594, 3.18739, -3.1921	-2.9080, -3.2403, 3.2356	3.1995, 3.6531, 3.6468

Acknowledgements

Various contents in this presentation have been taken from different books, lecture notes, and the web. These solely belong to their owners and are here used only for clarifying various educational concepts. Any copyright infringement is *not* intended.