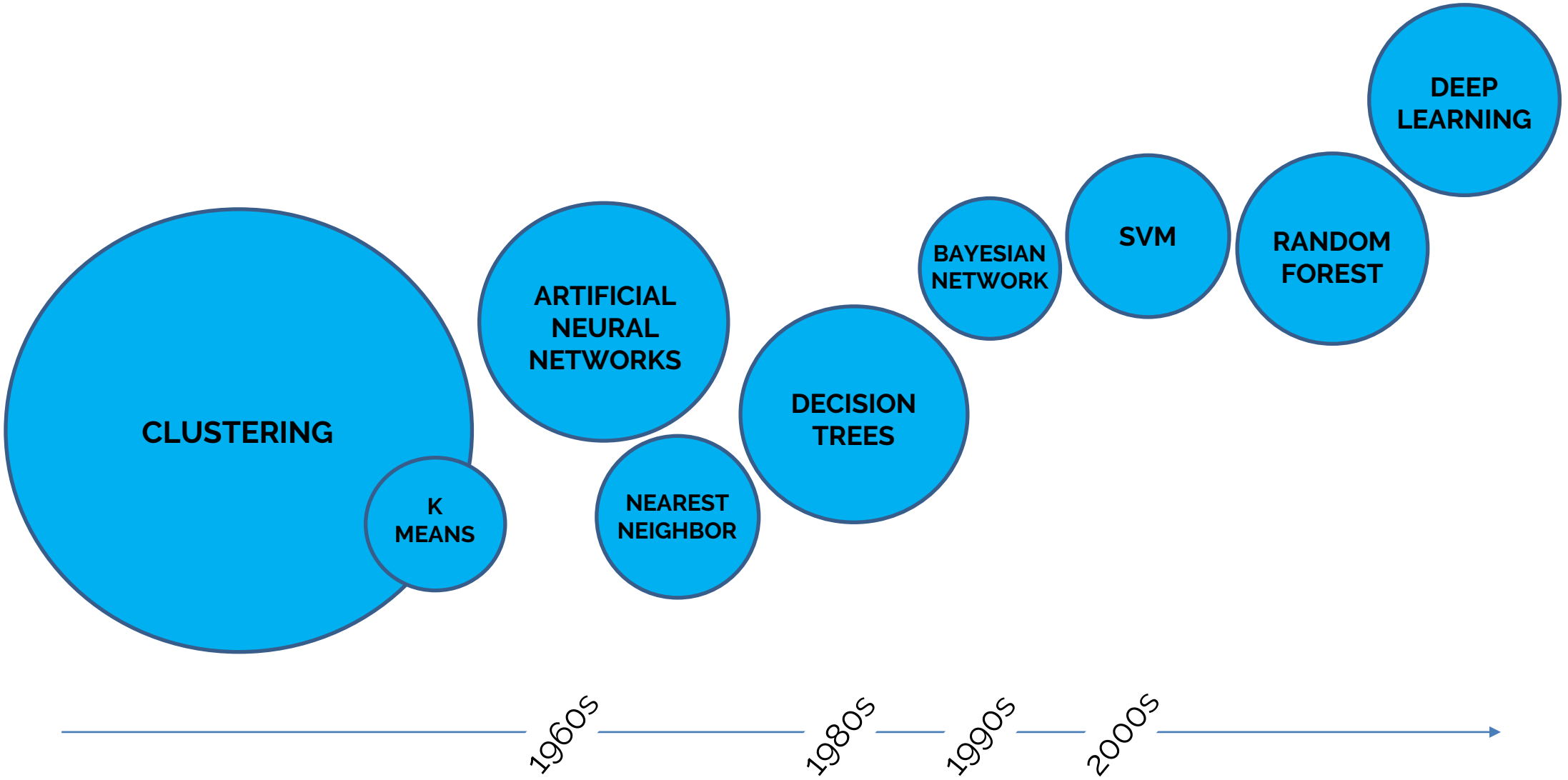
Machine learning e modelli di classificazione di dati biomedici

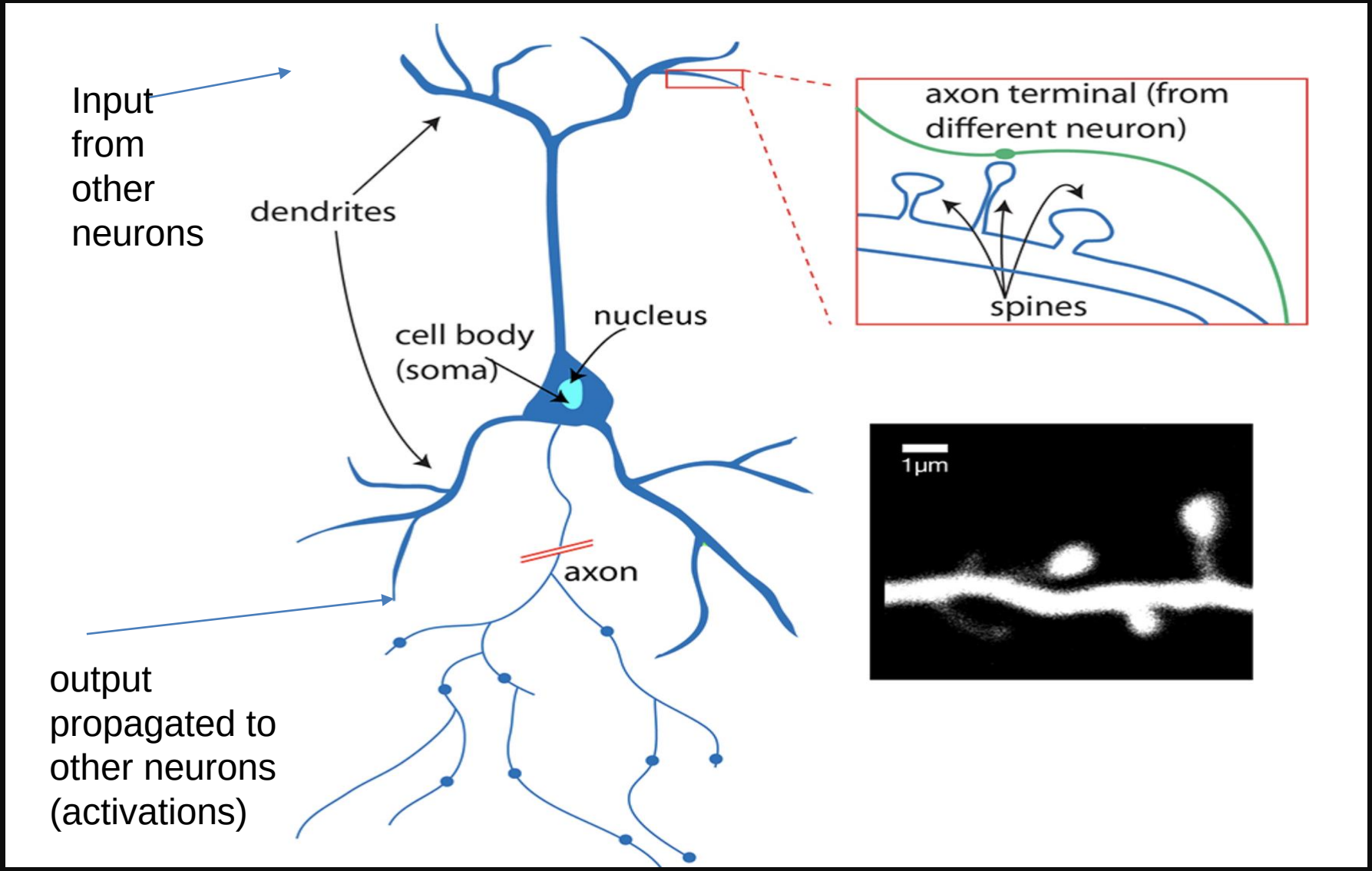
Christian Salvatore
Scuola Universitaria Superiore IUSS Pavia

Machine learning



NEURAL NETWORKs

A Biological Neuron



A Human-Vision Task

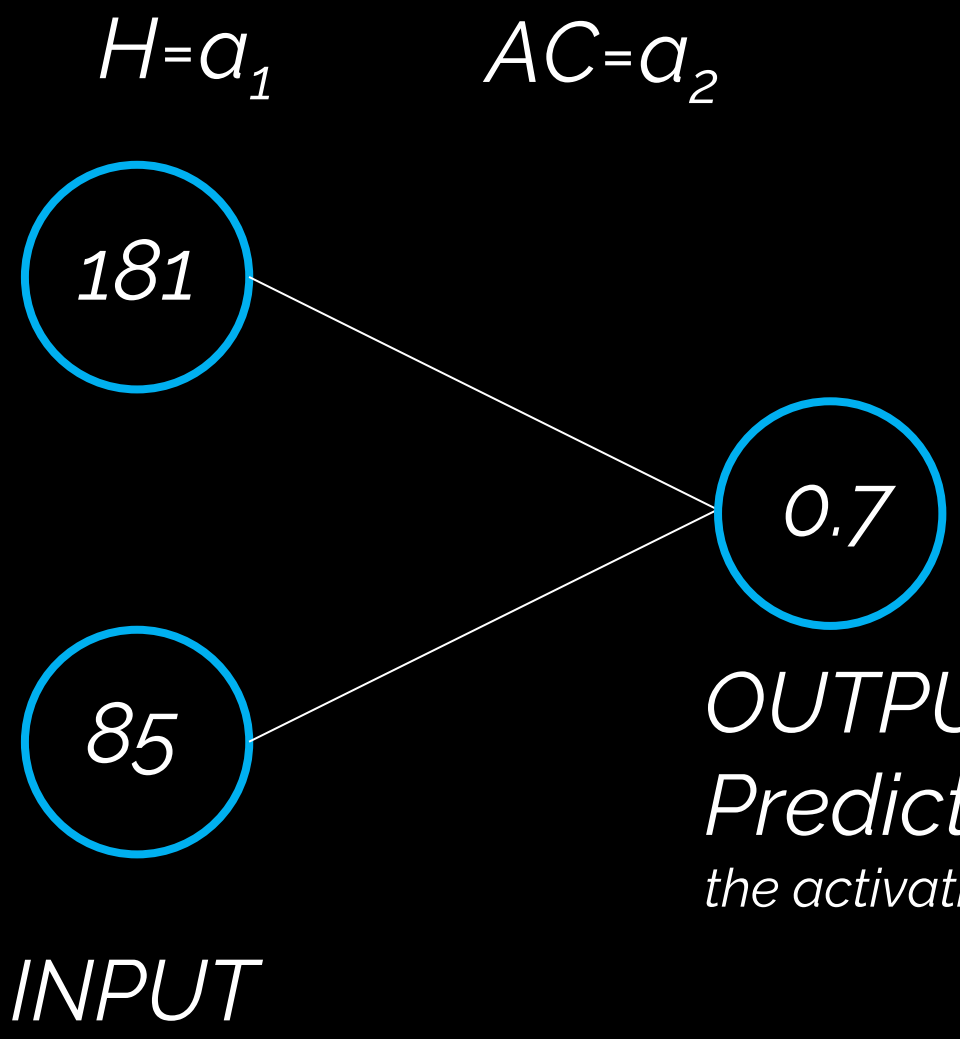
‘healthy’ vs overweighting men

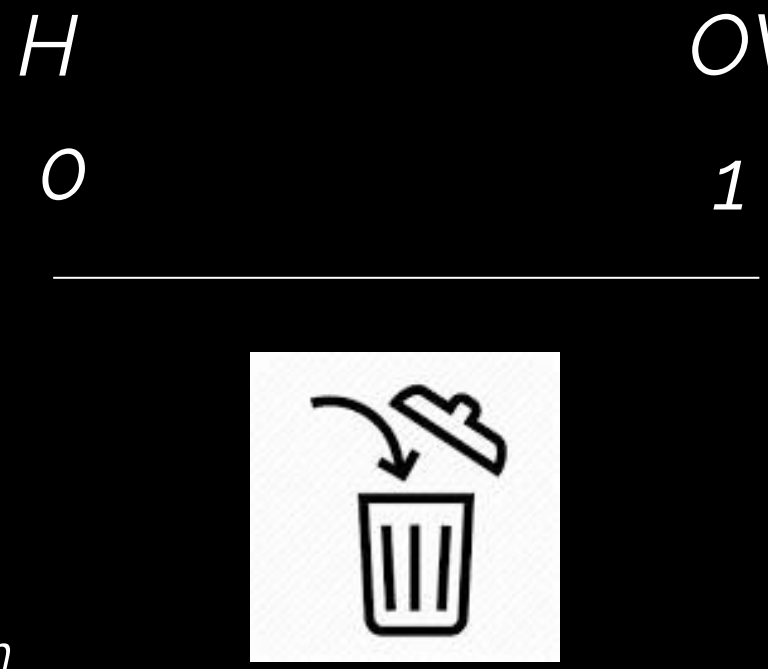
H (cm)	181	184	172	160	170	187	184	176	190
AC (cm)	85	94	102	80	98	110	116	77	84

H *OW*

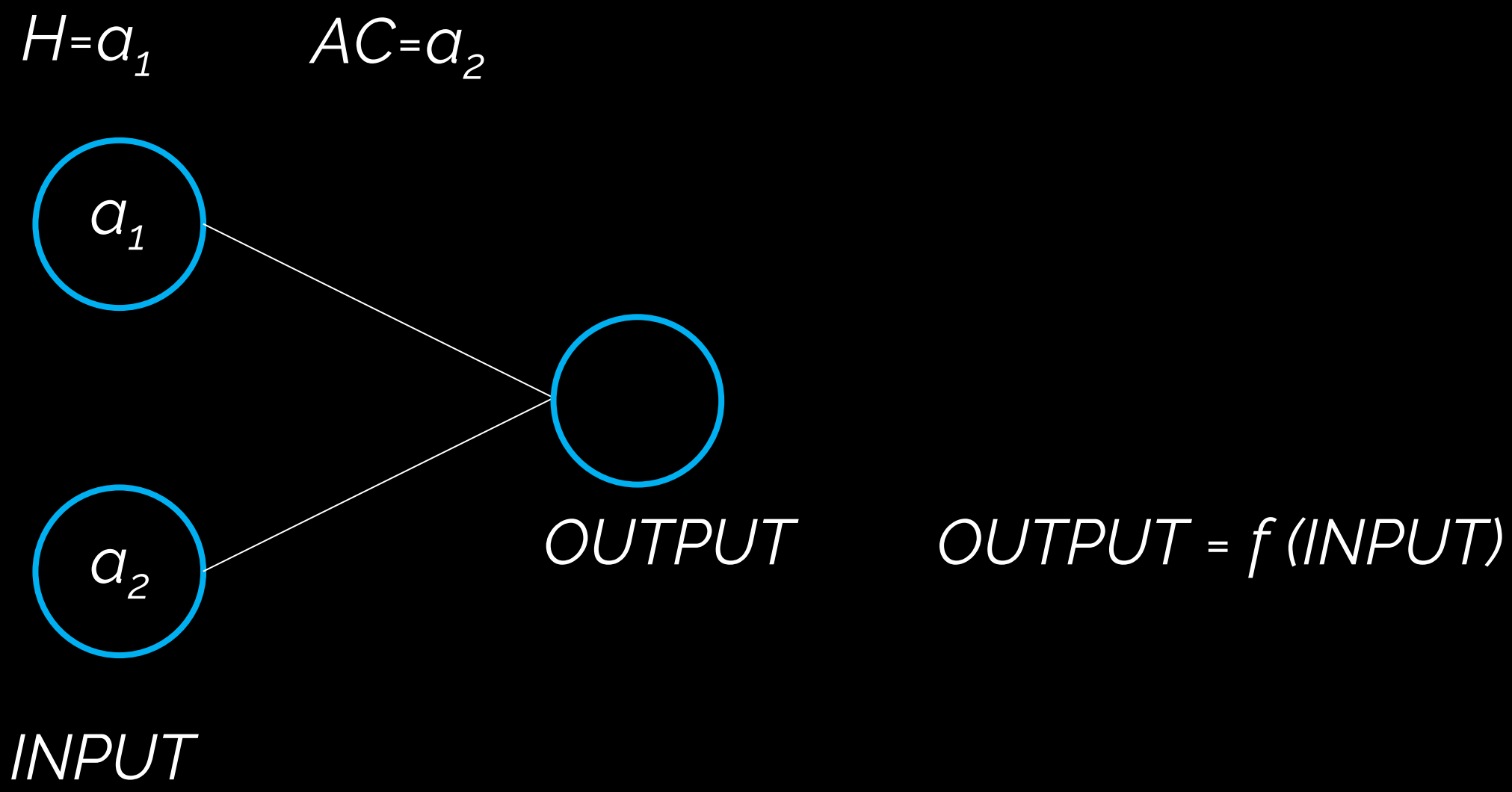
0 *1*

Neural Network





Neural Network

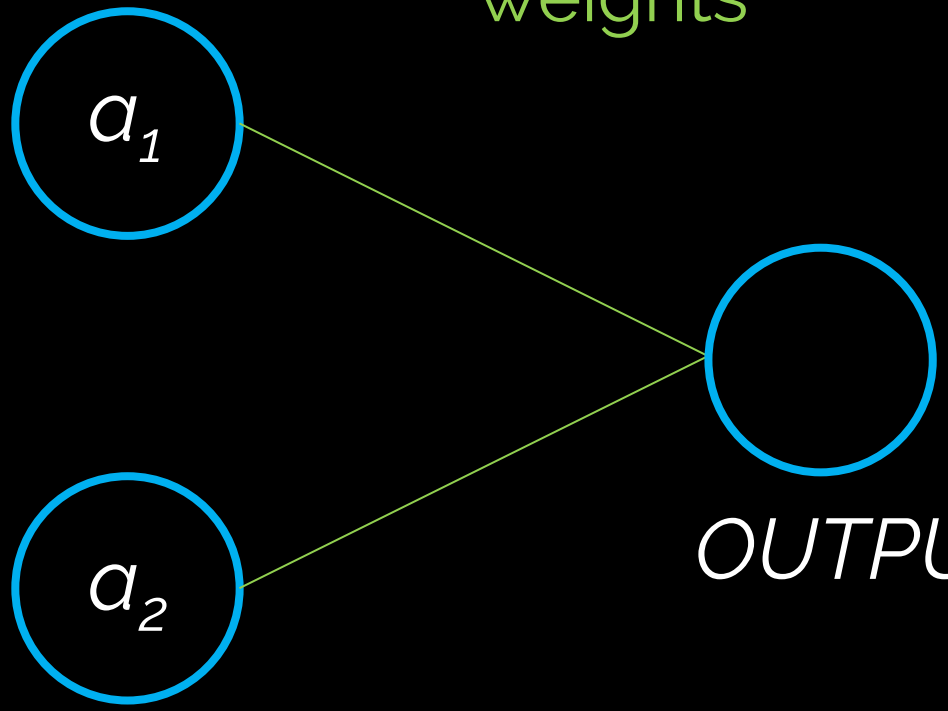


Weights and Bias

Neural Network | Weights and Bias

$H=a_1$ $AC=a_2$

weights



Bias (for inactivity)

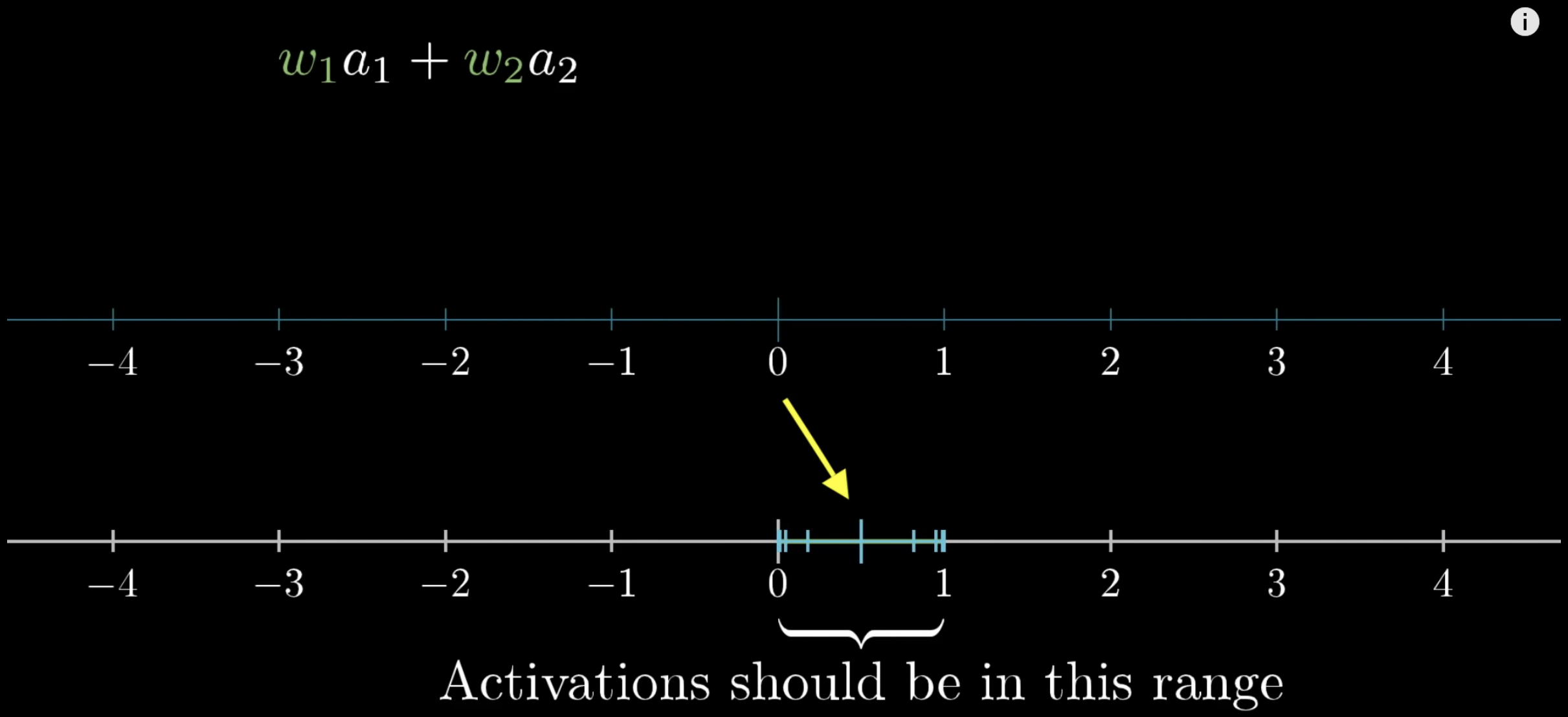
$a_1w_1 + a_2w_2 + b$

OUTPUT *OUTPUT = f (INPUT)*

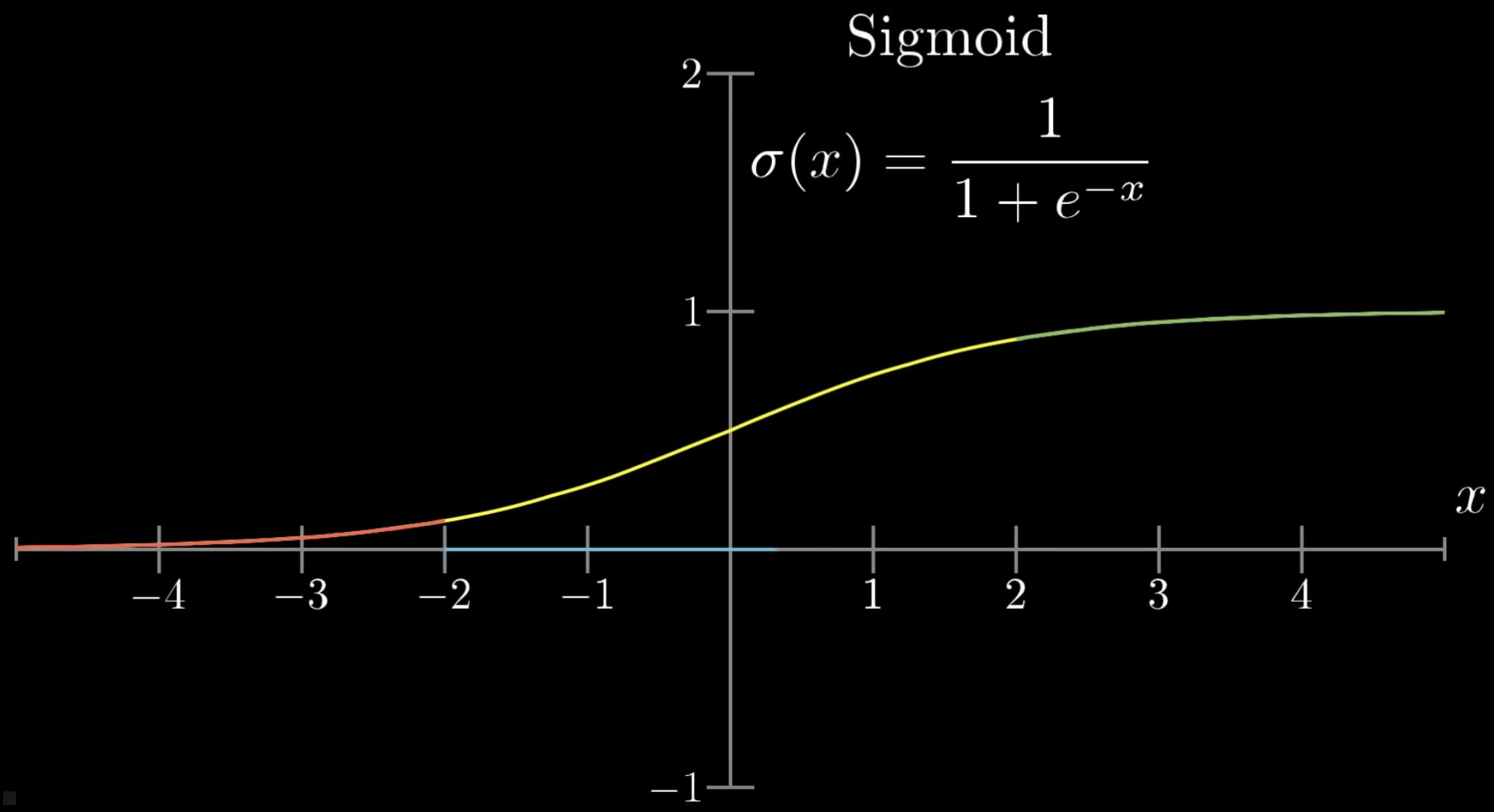
INPUT

The Activation Function

Neural Network



Neural Network



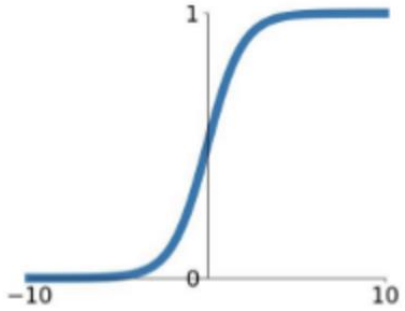
Neural Network

Other functions that progressively change from 0 to 1 with no discontinuity

Hyperbolic tangent funtion

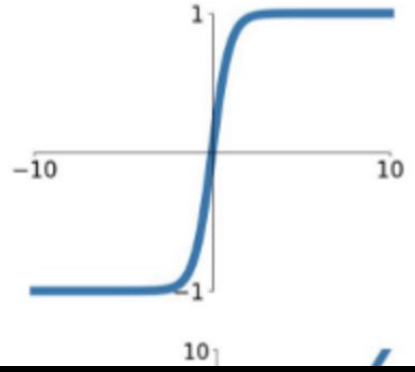
Sigmoid

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



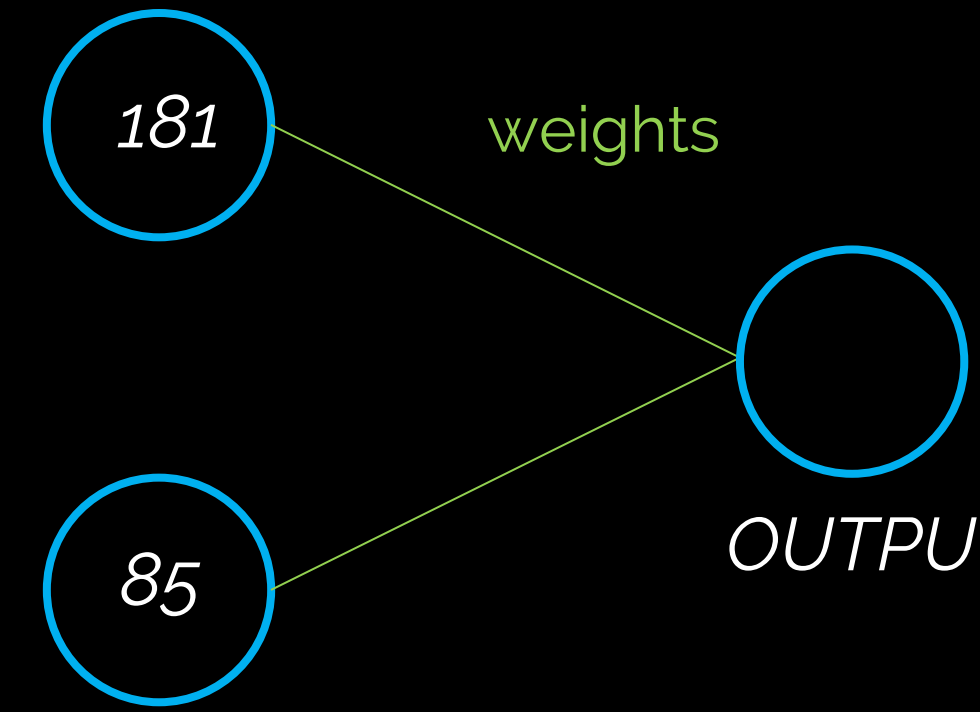
tanh

$$\tanh(x)$$



Neural Network

$H=a_1$ $AC=a_2$



Forward propagation

Activation function

$Sigmoid\ (181w_1 + 85w_2 + b)$

$OUTPUT = f\ (INPUT)$

w_1 ? w_2 ? b ?

Neural Network

INPUT

a_1 a_2

Cost Function?

Sigmoid ($a_1 w_1 + a_2 w_2 + b$)

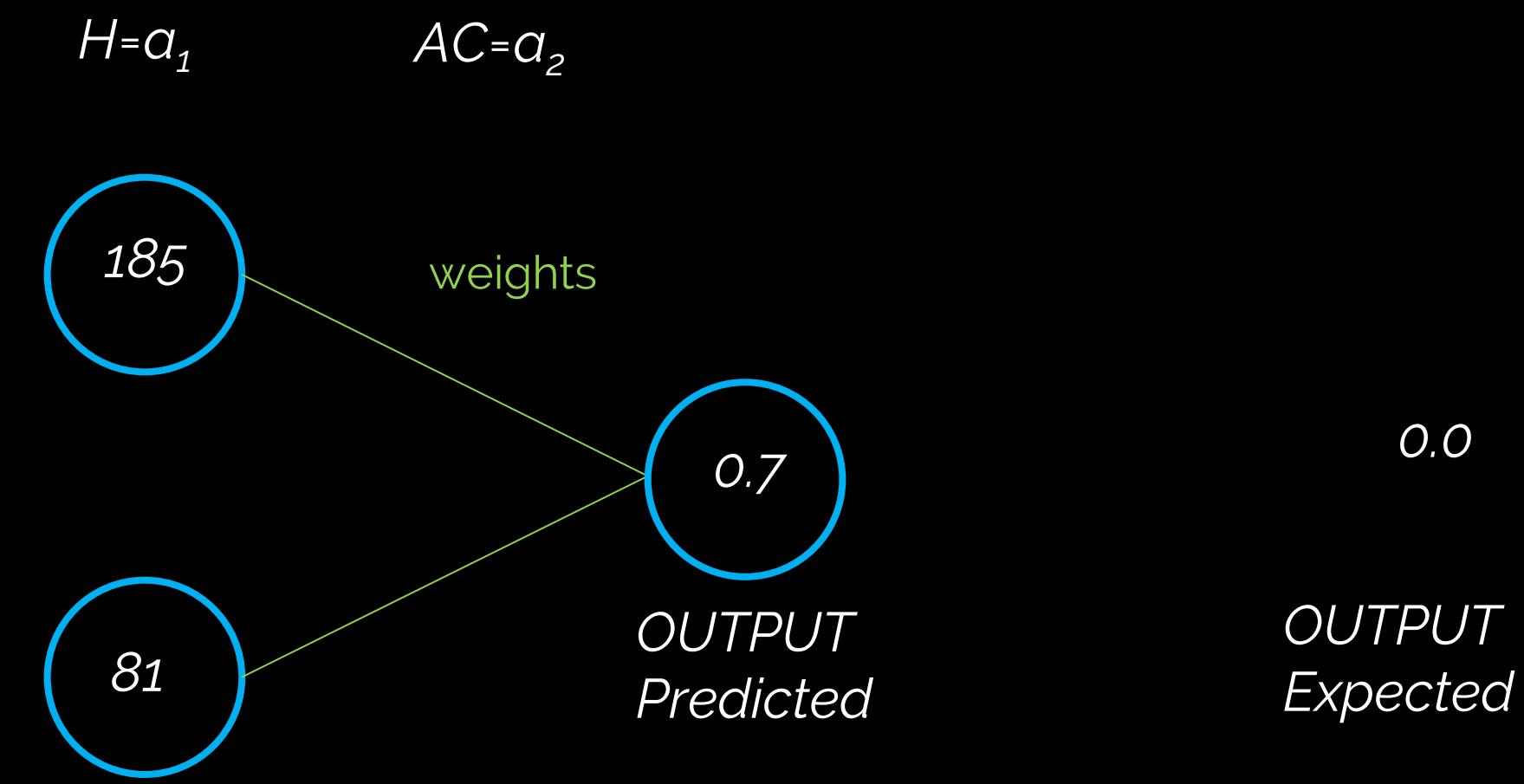
OUTPUT

Neural Network

$$\text{Cost Function} = (\text{Predicted}-\text{Expected})^2$$

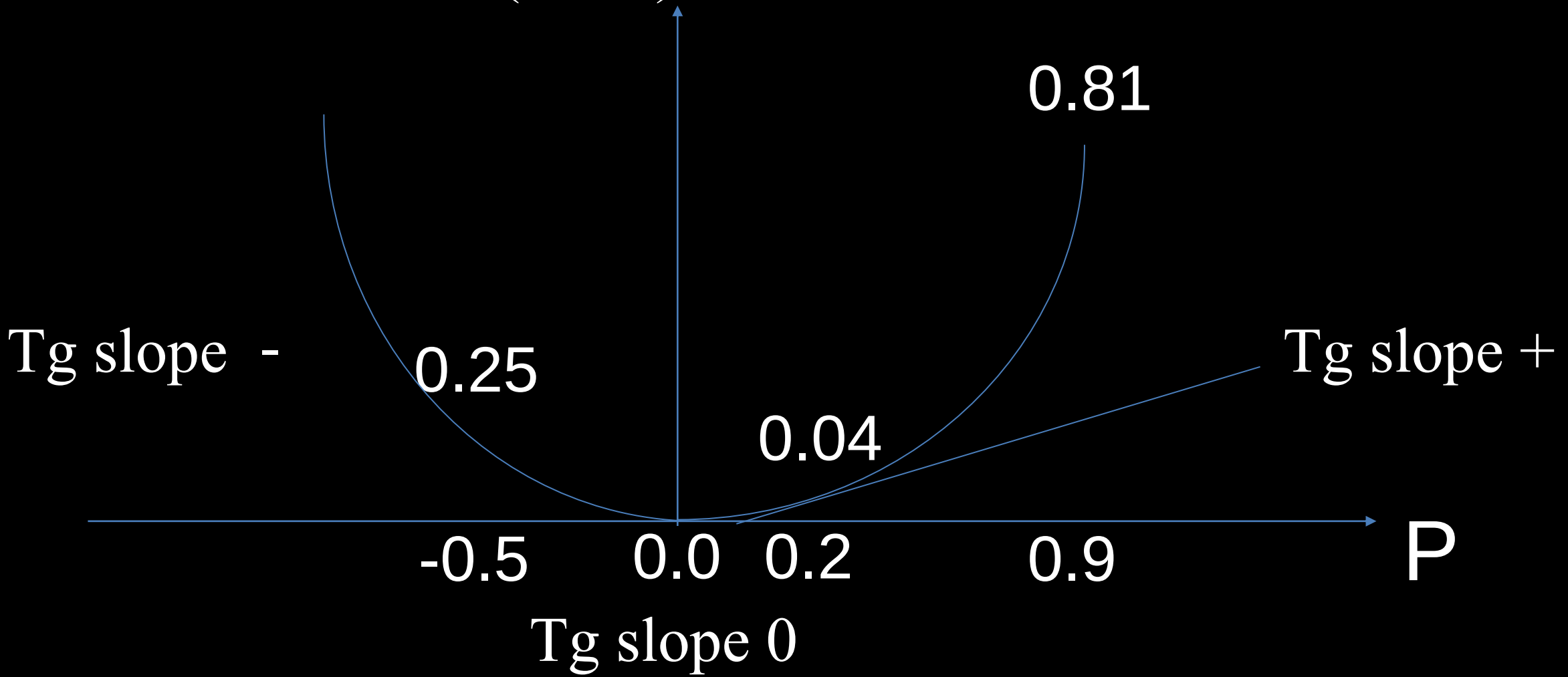
Squared error cost

Neural Network



Cost Function = $(0.7-0.0)^2=0.49$

The Cost Function = $(P-0.0)^2$



Neural Network

If the slope is + we must decrease P of a fraction of the slope

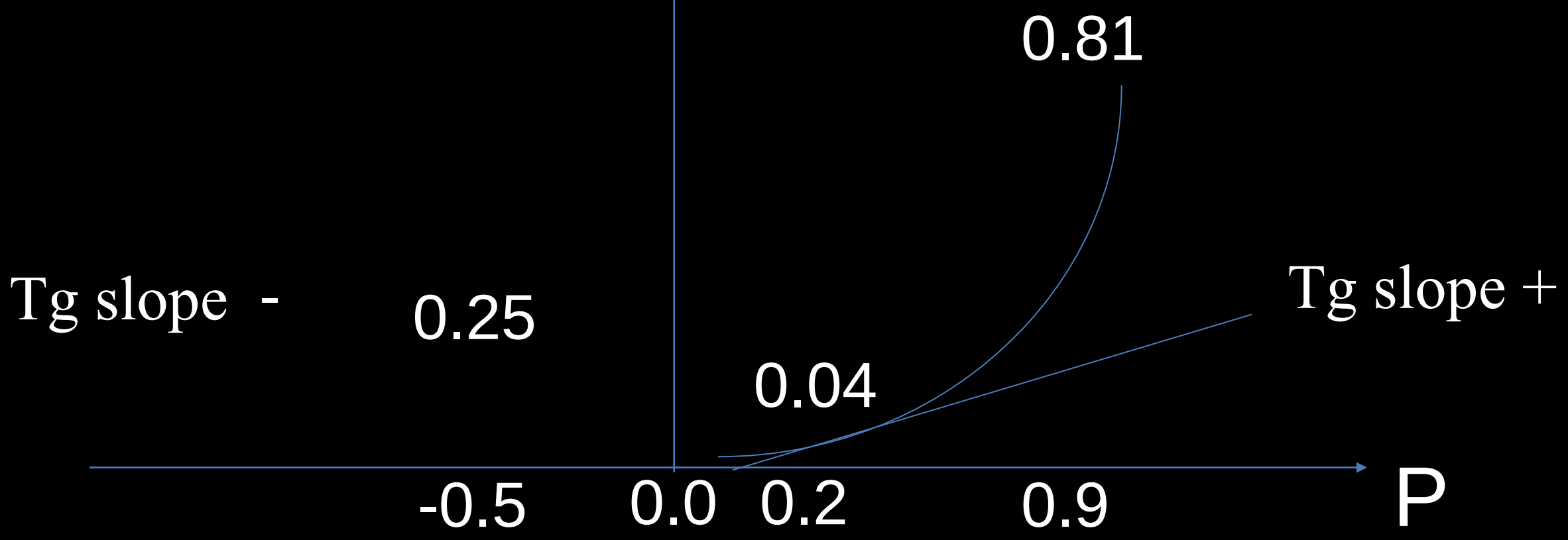
If the slope is - we must increase P of a fraction of the slope

If the slope is 0 we have the solution

The Learning Rate

Neural Network

The Cost Function = $(P-0.0)^2$



Next P = P – LR*(Tg slope)P LR*=Learning Rate

Neural Network

Next P = P – LR* (slope Tg)P

Slope Tg= derivative of the Cost Function vs P= 2(P-E)

In our case

Neural Network

LR determines how much weights are changed every time
Too high→ output wanders around the expected solutions
Too low → output fails to converge to acceptable solution

The Training

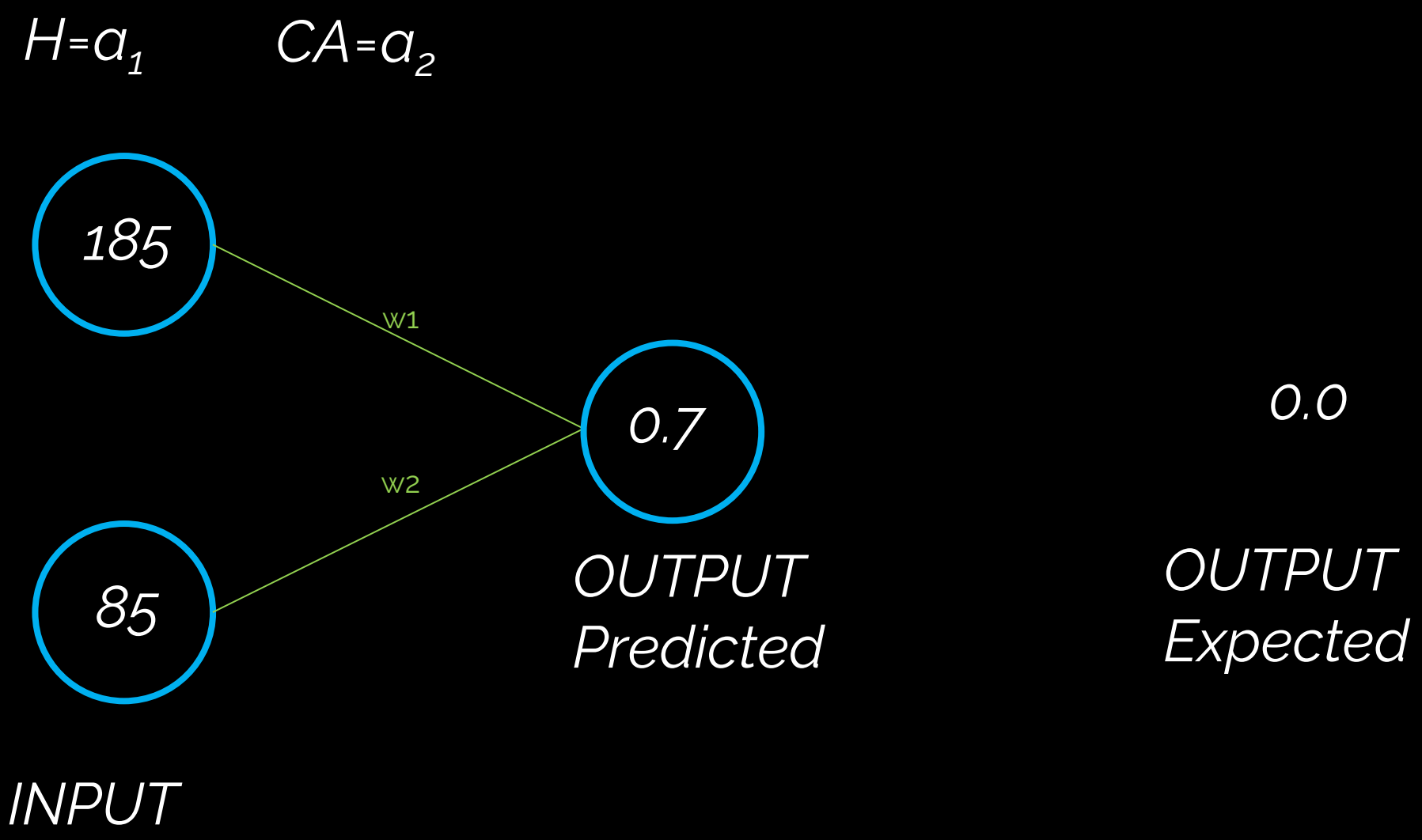
Neural Network

Different training methods

Supervised

learning rule that trains the neural network on already known correct output

Neural Network



Neural Network

w1 = 0.0006

w2 = 0.0002
b=1

0.7 =Sigmoid (185x0.0006+ 85x0.0002+1) =
Sigmoid (0.111+0.281+1)= Sigmoid (1.281)

the back propagation

Neural Network

Computation of the error (STEP 2)

The Cost Function = $(0.7-0.0)^2=0.49$

= $(Sigmoid\ (185x0.0006 + 85x0.0002 + 1) - 0.0)^2$

Weights and Bias Adjustment: the Back Propagation

Neural Network

Adjust weights and b to reduce the error (STEP 3)

$0.0005 = 0.0006 - 0.0001$

$0.0001 = 0.0002 - 0.0001$

$0.0008 = 1 - 0.0002$

$w_1 = w_1 - LR^* \text{ slope} = w_1 - LR^* \text{ derivative}_{w1} \text{ of the Cost Function}$

$w_2 = w_2 - LR^* \text{ slope} = w_2 - LR^* \text{ derivative}_{w2} \text{ of the Cost Function}$

$b = b - LR^* \text{ slope} = b - LR^* \text{ derivative}_{b} \text{ of the Cost Function}$

Neural Network

$$w1 = w1 - LR^* \frac{\partial costo}{\partial w1}$$

$$w2 = w2 - LR^* \frac{\partial costo}{\partial w2}$$

$$b = b - LR^* \frac{\partial costo}{\partial b}$$

Neural Network

$$\frac{\partial \text{costo}}{\partial w1} = \frac{\partial \text{costo}}{\partial p} \times \frac{\partial p}{\partial t}$$

$$\frac{\partial}{\partial t}\,\text{sigmoide}(t) = \text{sigmoide}(t)(1 - \text{sigmoide}(t))$$

$$\frac{\partial \text{costo}}{\partial w1} = \frac{\partial \text{costo}}{\partial p} \times \frac{\partial p}{\partial t} \times \frac{\partial t}{\partial w1}$$

$$\frac{\partial \text{costo}}{\partial w1} = 2(\text{sigmoide}(2w1+5w2+b) - 1) \times \text{sigmoide}(2w1+5w2+b)(1-\text{sigmoide}(2w1+5w2+b)) \times 2$$

Neural Network

$$\frac{\partial costo}{\partial w1} = 2(sigmoide(2w1+5w2+b) - 1) \times sigmoide(2w1+5w2+b)(1-sigmoide(2w1+5w2+b)) \times 2$$

$$\frac{\partial costo}{\partial w2} = 2(sigmoide(2w1+5w2+b) - 1) \times sigmoide(2w1+5w2+b)(1-sigmoide(2w1+5w2+b)) \times 5$$

$$\frac{\partial costo}{\partial b} = 2(sigmoide(2w1+5w2+b) - 1) \times sigmoide(2w1+5w2+b)(1-sigmoide(2w1+5w2+b)) \times 1$$

Neural Network

$$\frac{\partial costo}{\partial w1} = \frac{\partial costo}{\partial p} \times \frac{\partial p}{\partial t} \times \frac{\partial t}{\partial w1}$$

$$\frac{\partial costo}{\partial w1} = 2(sigmoide(2w1+5w2+b) - 1) \times sigmoide(2w1+5w2+b)(1-sigmoide(2w1+5w2+b)) \times 2$$

$$\frac{\partial costo}{\partial w2} = \frac{\partial costo}{\partial p} \times \frac{\partial p}{\partial t} \times \frac{\partial t}{\partial w2}$$

$$\frac{\partial costo}{\partial w2} = 2(sigmoide(2w1+5w2+b) - 1) \times sigmoide(2w1+5w2+b)(1-sigmoide(2w1+5w2+b)) \times 5$$

$$\frac{\partial costo}{\partial wb} = \frac{\partial costo}{\partial p} \times \frac{\partial p}{\partial t} \times \frac{\partial t}{\partial b}$$

$$\frac{\partial costo}{\partial b} = 2(sigmoide(2w1+5w2+b) - 1) \times sigmoide(2w1+5w2+b)(1-sigmoide(2w1+5w2+b)) \times 1$$

Neural Network

$w_1 = 0.0006$ $w_2 = 0.0002$ $b = +1$

The Cost Function (w_1 w_2 b) = (Sigmoid-0.0)^2=0.49

Back propagation

Weights and bias adjustment

$w_1 = 0.0005$ $w_2 = 0.0001$ $b = 0.0008$

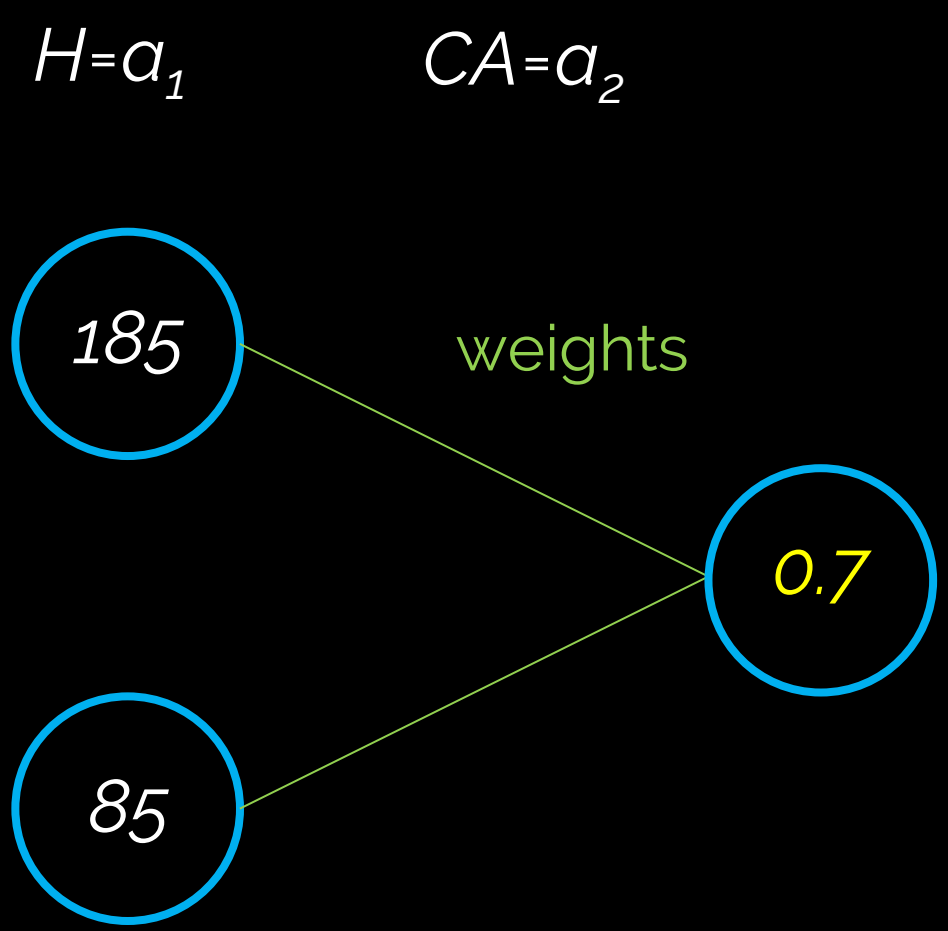
The Cost Function (w_1 w_2 b) = (Sigmoid-0.0)^2=0.43

Neural Network

Sigmoid (185x0.0005+ 85x0.0001+1) = Sigmoid (0.0925+0.085+0.008)= Sigmoid (1.178)=0.65

the back propagation

Neural Network



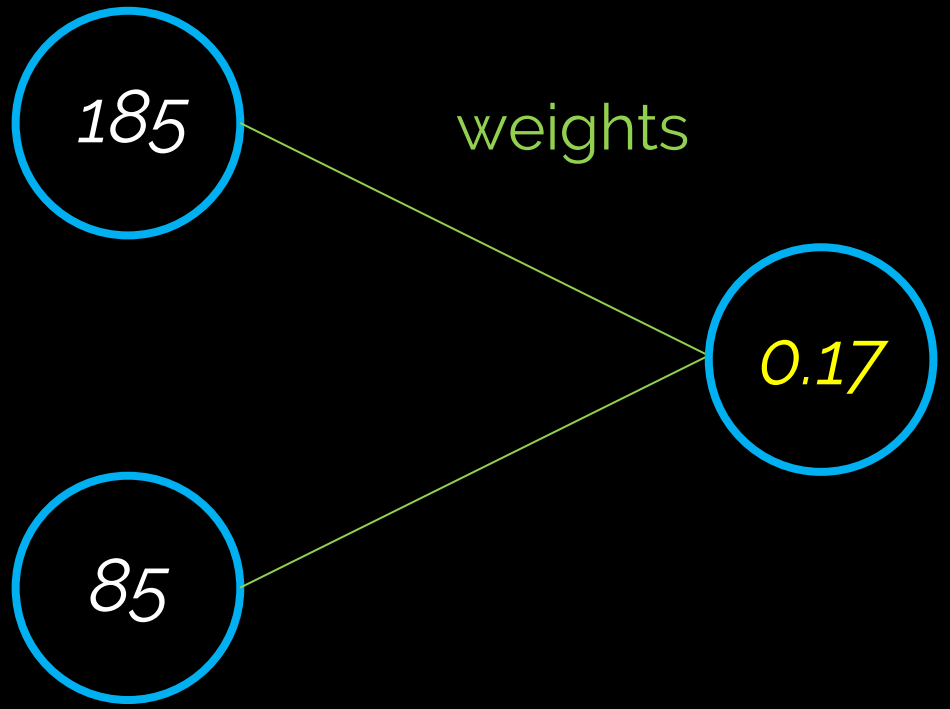
Forward propagation

Activation function

$$Sigmoid\,(185w_1 + 85w_2 + b)$$

Neural Network

$H = a_1$ $CA = a_2$



Back propagation

Cost function

$$(Sigmoid\ (185w_1 + 85w_2 + b) - 0.0)^2$$

Training set: ‘healthy’ vs overweighting men

H	185	178	184	190	169	188	185	192	175	H	168	179	170	180	169	189	188	190	178	H	185	192	175	185	192	168	190	186	168
C	85	94	102	80	98	110	116	77	84	C	88	90	68	120	112	109	89	92	94	C	77	103	119	98	99	100	98	96	78

H	188	190	178	169	188	185	192	190	188	H	185	192	168	179	188	185	192	175	180	H	169	189	188	190	169	188	185	192	188
C	90	79	68	120	98	99	102	104	89	C	87	90	102	98	96	78	90	103	110	C	90	89	90	79	68	120	96	99	79

H	168	179	170	182	168	178	190	188	177	H	175	188	178	169	176	168	179	170	180	H	178	193	190	169	188	185	177	186	165
C	113	90	90	79	68	120	98	96	78	C	119	98	106	116	77	103	119	98	77	C	90	79	68	120	112	109	89	92	78

H	190	170	179	170	180	179	177	185	159	H	176	168	179	188	185	177	168	179	188	H	176	181	174	185	192	175	185	192	169
C	68	120	98	96	77	103	119	98	89	C	104	77	98	96	78	90	79	68	120	C	102	98	96	78	82	95	99	100	96

Neural Network

Calculate the error (STEP 2)

Adjust weights and b to reduce the error (STEP 3)

Repeat step 2 and step 3 for all training data until the error is within acceptable level

These steps are similar to supervised machine learning (model adjusting vs learning rules adjusting) (model adjusting vs weights and bias adjusting)

EPOCH (1 Trainig iteration)

Neural Network

$Sigmoid\ (185 \times 0.0006 + 85 \times 0.0002 + 1) = 0.7$

$Sigmoid\ (178 \times 0.0004 + 94 \times 0.0003 + 0.008) = 0.65$

$Sigmoid\ (184 \times 0.0003 + 102 \times 0.0001 + 0.007) = 0.64$

$Sigmoid\ (100 \times 0.0005 + 80 \times 0.0002 + 0.008) = 0.55$

$(Sigmoid - 0.0)^2 = 0.49$

$(Sigmoid - 0.0)^2 = 0.43$

$(Sigmoid - 0.0)^2 = 0.41$

$(Sigmoid - 0.0)^2 = 0.30$

Generalized delta rule

SGD (Stochastic Gradient Descent) method (error is calculated for each training data, weights updated immediately)

Batch method (error is calculated for all training data, each of the weight update are calculated but the avarage of all weight updates are used only once in each epoch)

Mini-Batch method (mix) →

- Small values give a learning process that converges quickly at the cost of noise in the training process.
- Large values give a learning process that converges slowly with accurate estimates of the error gradient.

Neural Network

Mini-Batch method

Training data 1

Training data 2

Training data 3

Training data N

Training using batch method

Part of the training data is selected

Mini-Batch method

1-10
11-20
21-40
41-60
61-80

Batch method applied

5 weight update will be performed to complete the training process

Robusteness of SGD
Efficiency of batch

SOFTWARE CODES

Matlab: essential functions and scripts

Matlab: simple examples

Neural Network

Sigmoid.m

```
1   function y = Sigmoid(x)
2 -   y = 1/(1+exp(-x));
3 -   end
4
5
```

Neural Network

Sigmoid.m | SGD_method.m

```
1   function Weight = SGD_method(Weight, input, correct_Output)
2   alpha = 0.9;
3
4   N = 4;
5   for k = 1:N
6       transposed_Input = input(k, :)';
7       d = correct_Output(k);
8   weighted_Sum = Weight*transposed_Input;
9   output = Sigmoid(weighted_Sum);
10
11  error      = d - output;
12  delta = output*(1-output)*error;
13
14  dWeight = alpha*delta*transposed_Input;
15
16  Weight(1) = Weight(1) + dWeight(1);
17  Weight(2) = Weight(2) + dWeight(2);
```

Neural Network

Sigmoid.m | SGD_method.m*

```
3
4 -     N = 4;
5 -     for k = 1:N
6 -         transposed_Input = input(k, :)';
7 -         d = correct_Output(k);
8 -     weighted_Sum = Weight*transposed_Input;
9 -     output = Sigmoid(weighted_Sum);
10
11 -    error      = d - output;
12 -    delta = output*(1-output)*error;
13
14 -    dWeight = alpha*delta*transposed_Input;
15
16 -    Weight(1) = Weight(1) + dWeight(1);
17 -    Weight(2) = Weight(2) + dWeight(2);
18 -    Weight(3) = Weight(3) + dWeight(3);
19 -    end
20 -    end
21
```

Neural Network

Sigmoid.m | SGD_method.m | Training.m

```
input = [ 0 0 1;
          0 1 1;
          1 0 1;
          1 1 1;
         ];
correct_Output = [0
                  0
                  1
                  1
                  ];
Weight = 2*rand(1, 3) - 1;
for epoch = 1:10000
    Weight = SGD_method(Weight, input, correct_Output);
end

save('Trained_Network.mat')
```

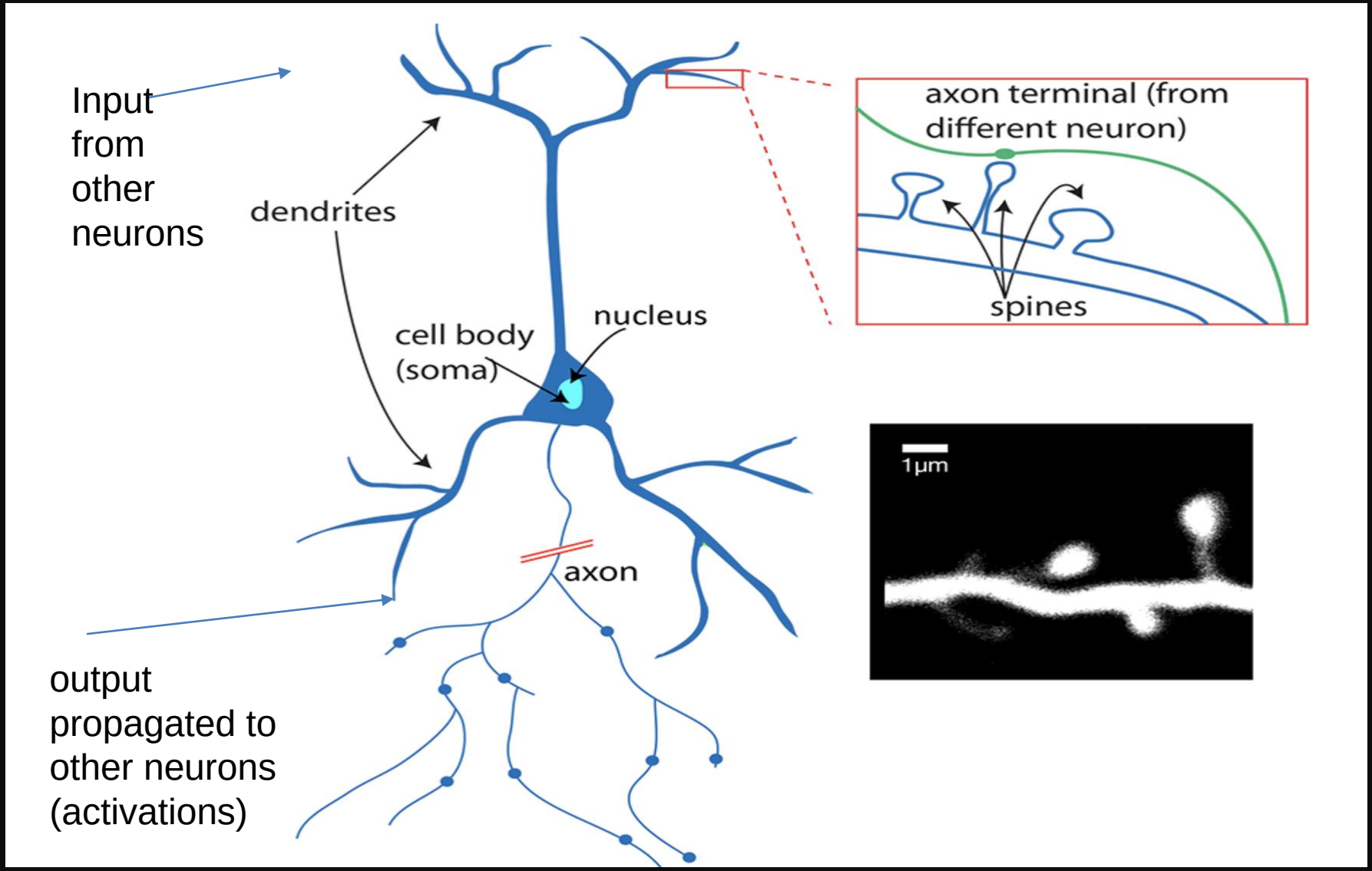
Neural Network

Sigmoid.m | SGD_method.m | Training.m | testing.m*

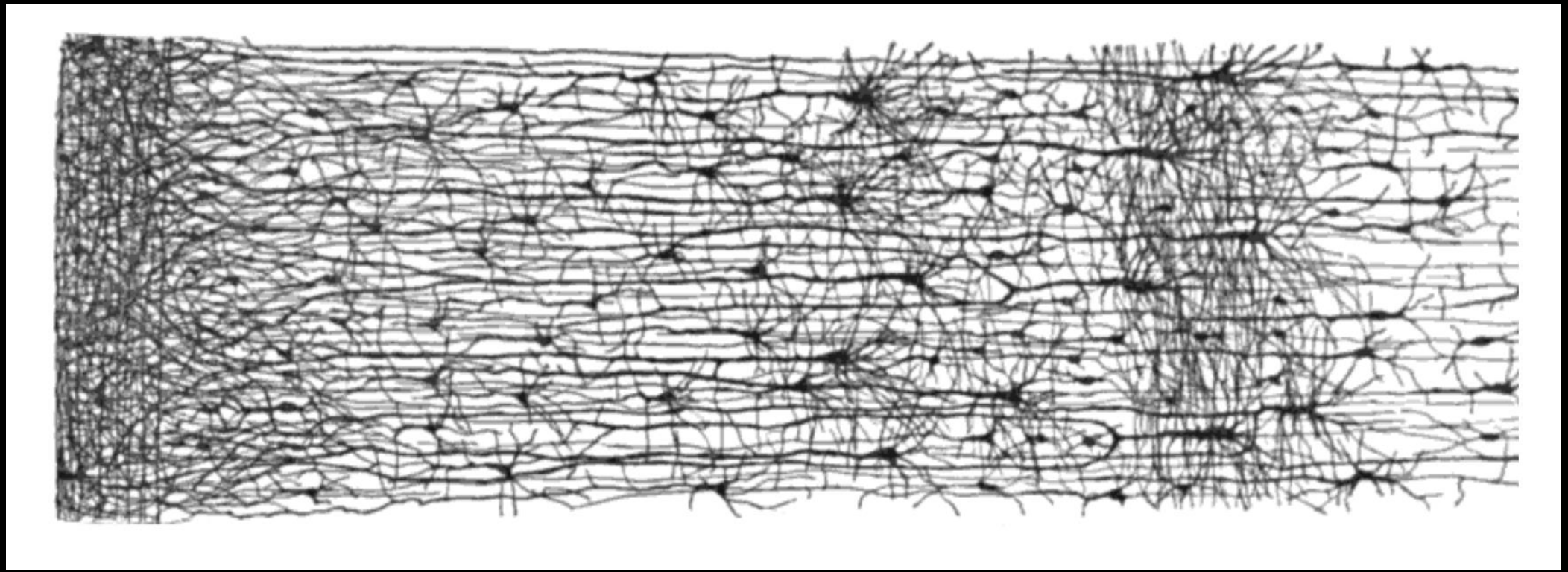
```
load('Trained_Network.mat');
input = [ 0 0 1;
          0 1 1;
          1 0 1;
          1 1 1;
        ];
N = 4;
for k = 1:N
    transposed_Input = input(k, :)';
    weighted_Sum = Weight*transposed_Input;
    output = Sigmoid(weighted_Sum)
end
```

DEEP LEARNING

A Biological Neuron



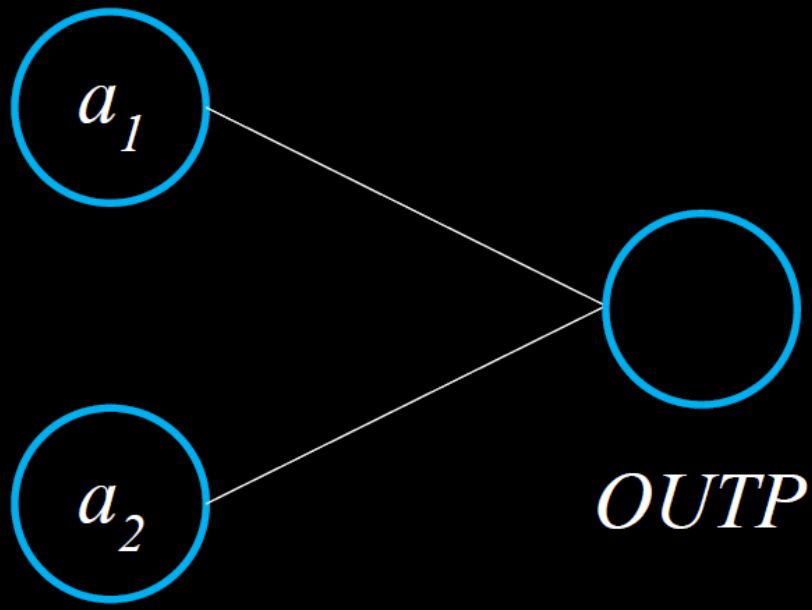
A Biological Neuron Network



100 B neurons

Deep Learning

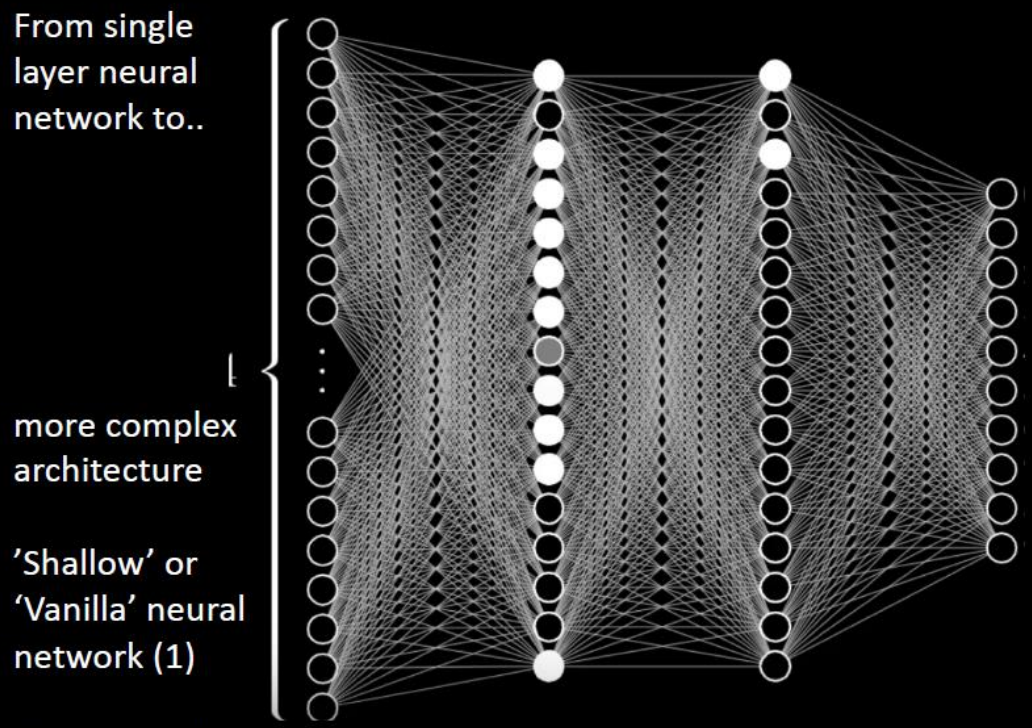
$H=a_1$ $AC=a_2$



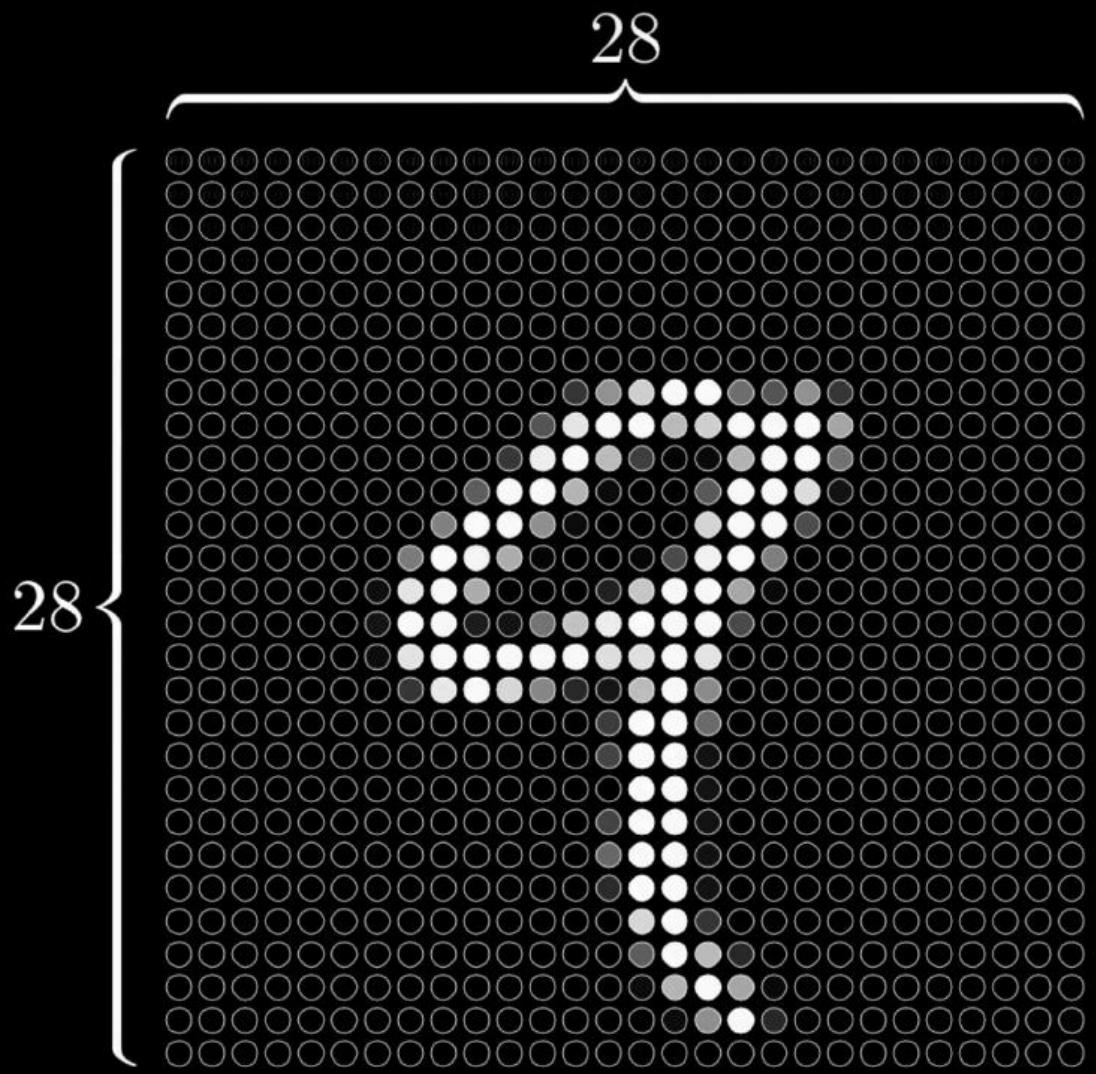
$OUTPUT$ $OUTPUT = f(INPUT)$

$INPUT$

Deep Learning



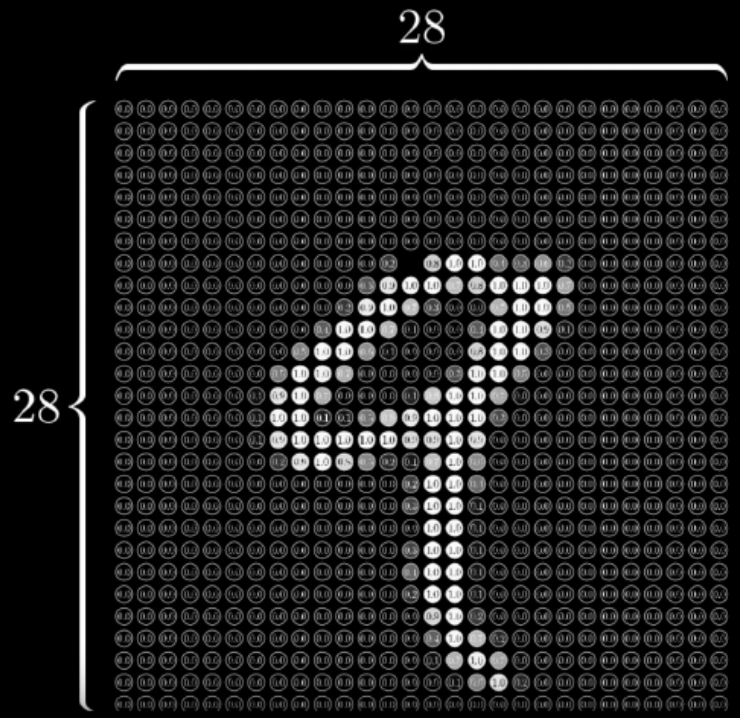
Deep Learning



$$28 \times 28 = 784$$

neurons

Deep Learning



$28 \times 28 = 784$ neurons

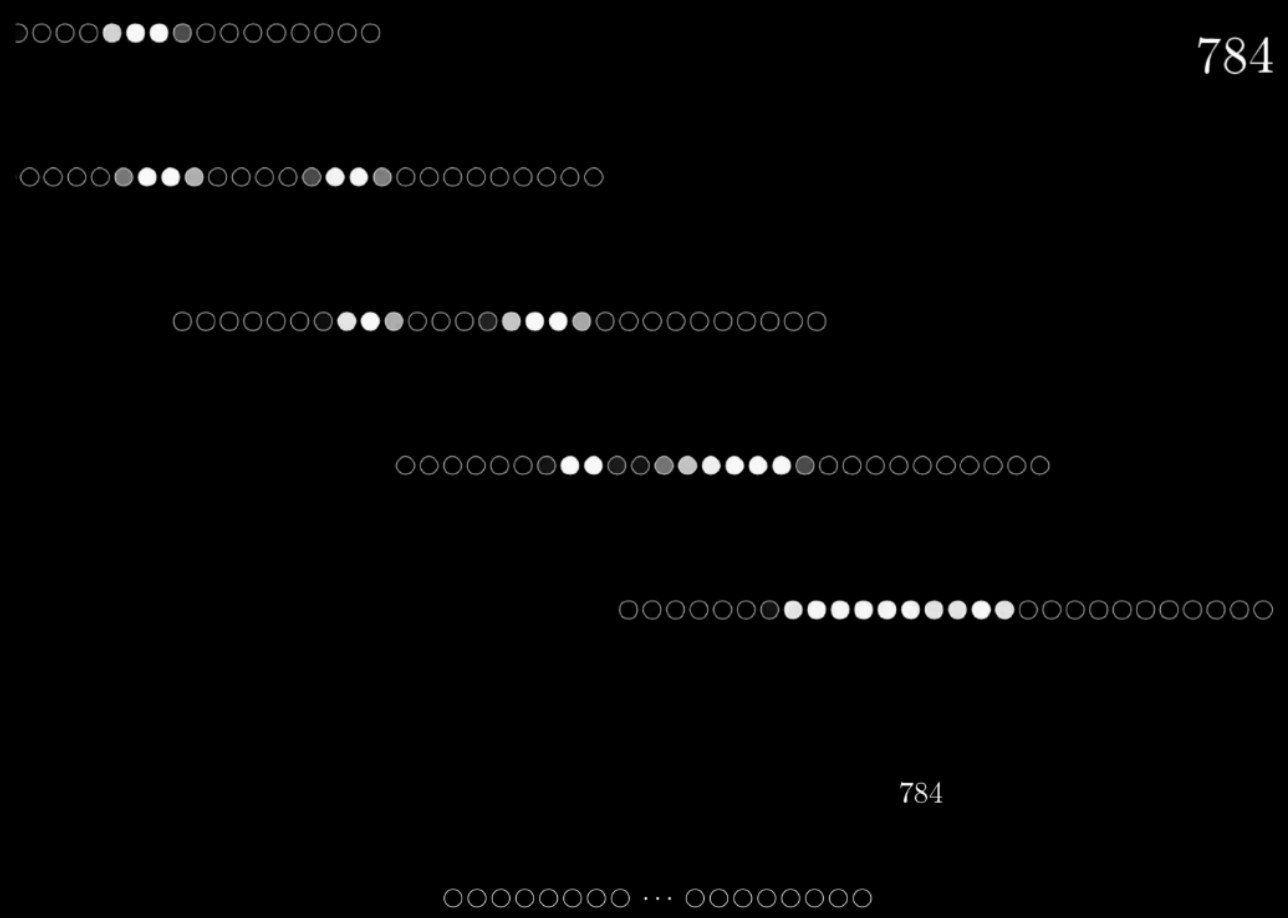
→ They make the 1° layer of the network

0.29

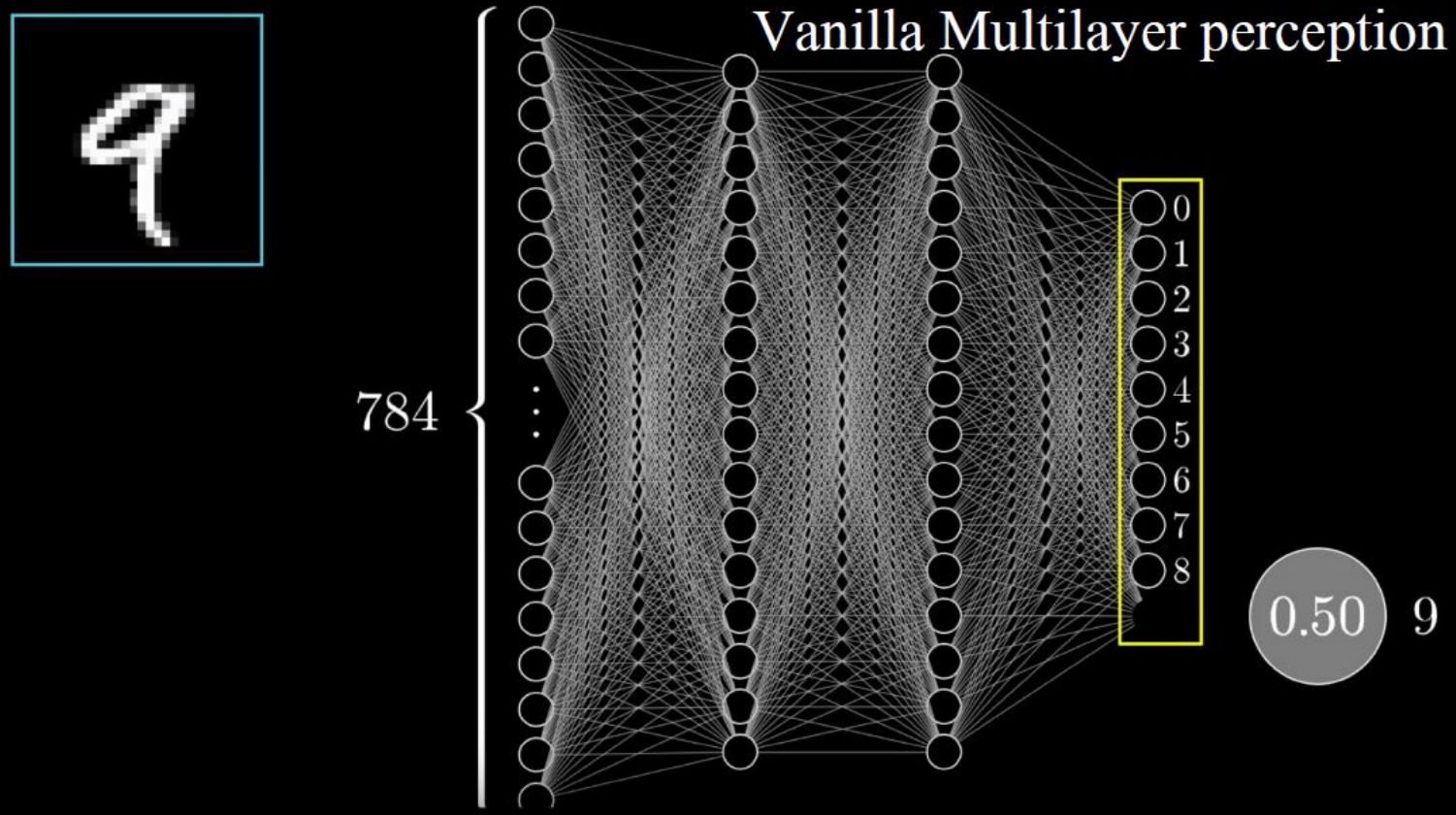
«Activation» of the neuron (input)

Grey scale value of each image pixel

Deep Learning



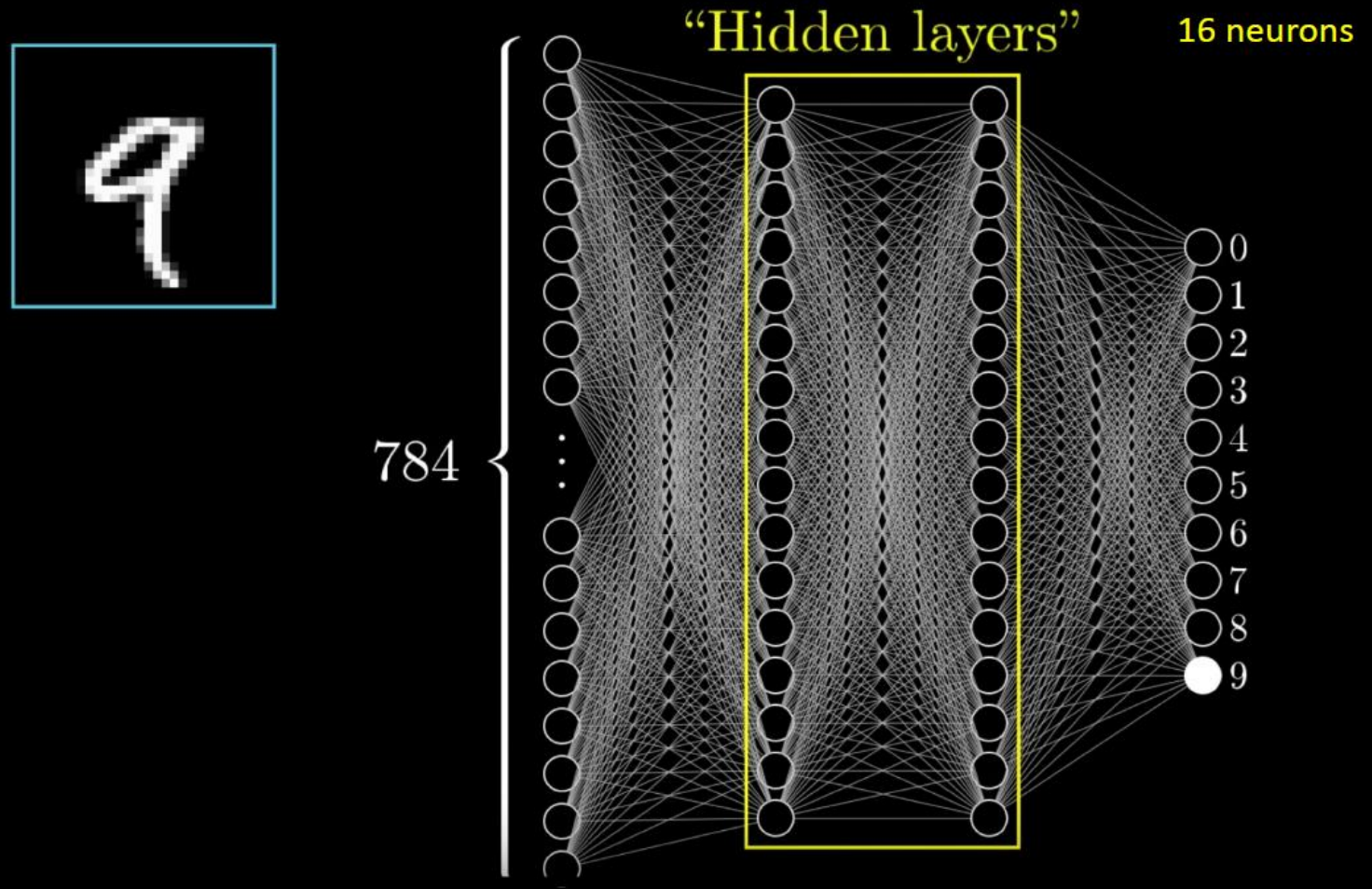
Deep Learning



«Activation» of the neuron (output)

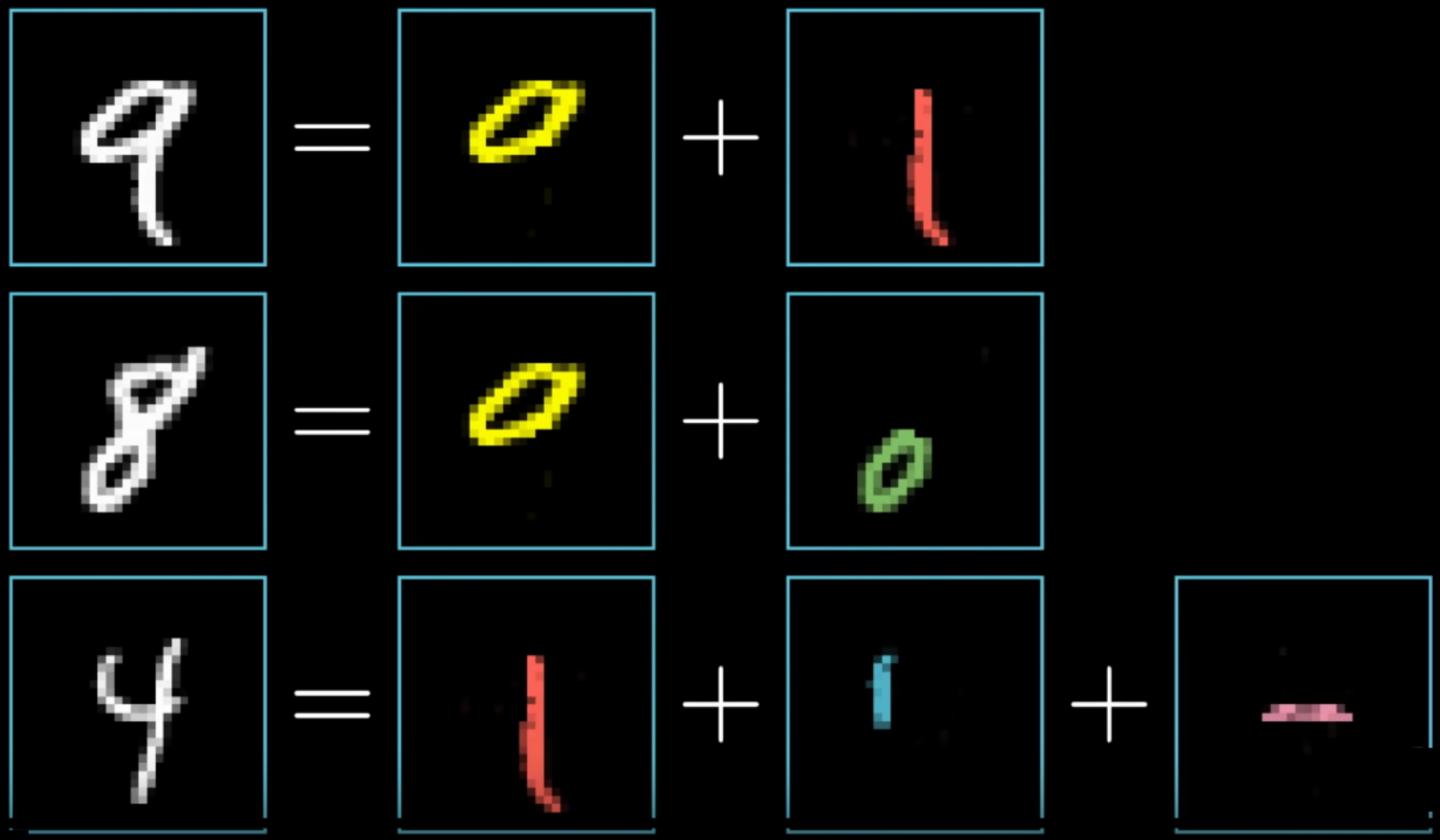
How the system think the given image corresponds to the given digit

Deep Learning



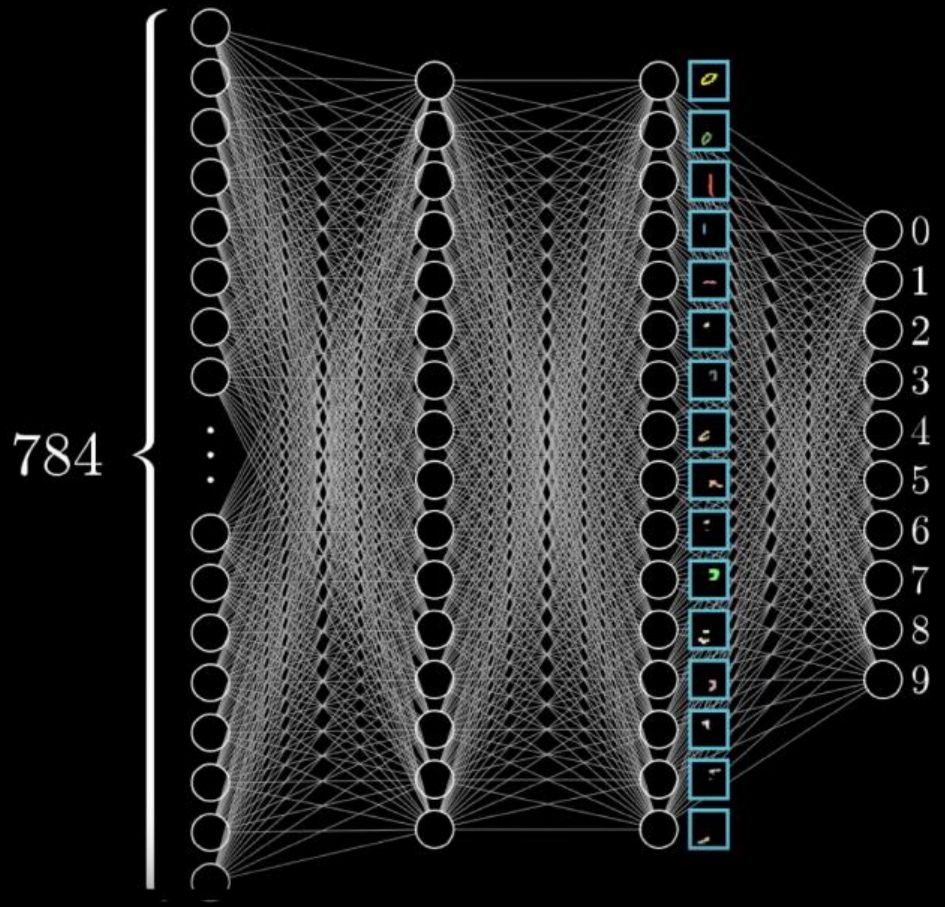
ACTIVATION of a layer determines activation in the next layers (cascade influence)

Deep Learning



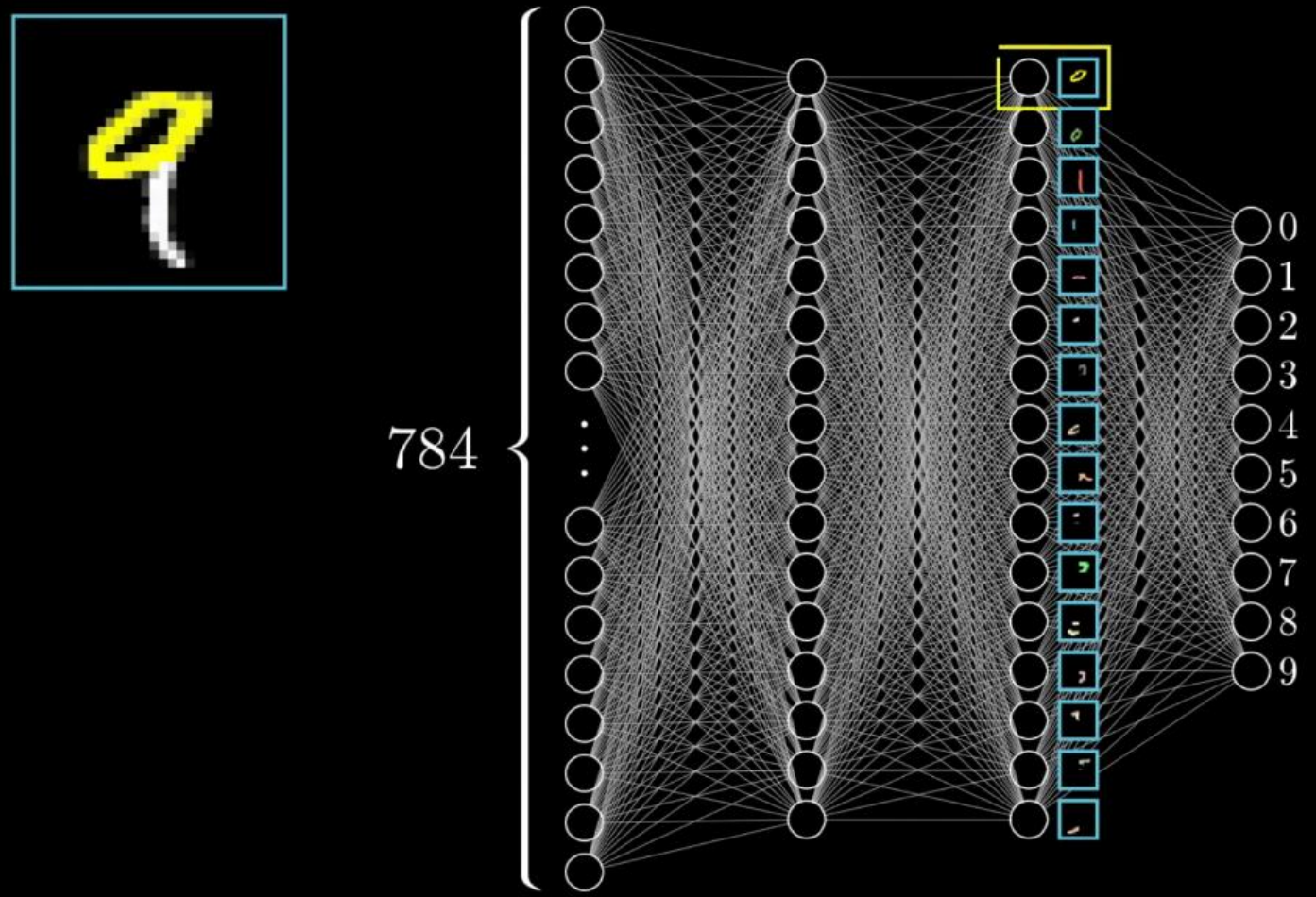
Possible subcomponents recognized by our visual-cortex neurons

Deep Learning



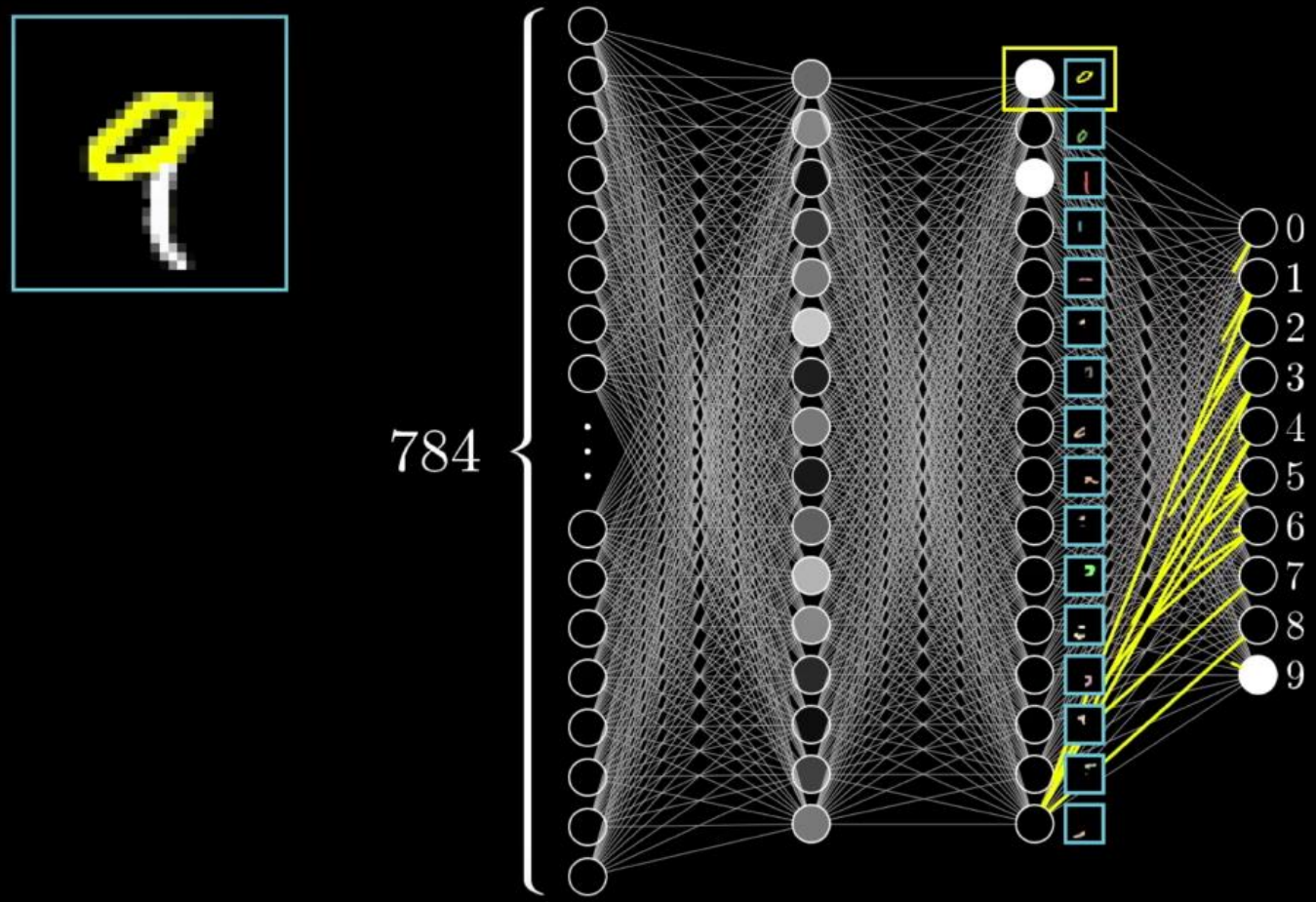
In the optimal situations each neurons of the middle layer correpond to each of the possible sub-components

Deep Learning



Anytime we feed an image with a loop (9, 8) on top there is a specific neuron in the third layer whose activation is near to 1

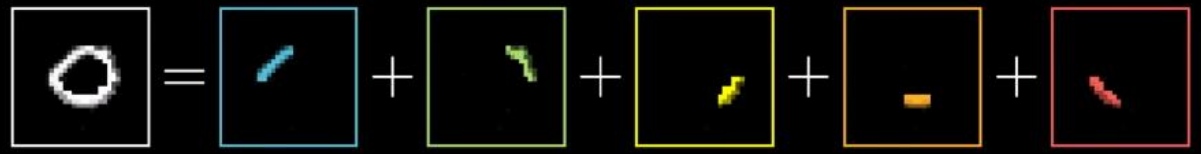
Deep Learning

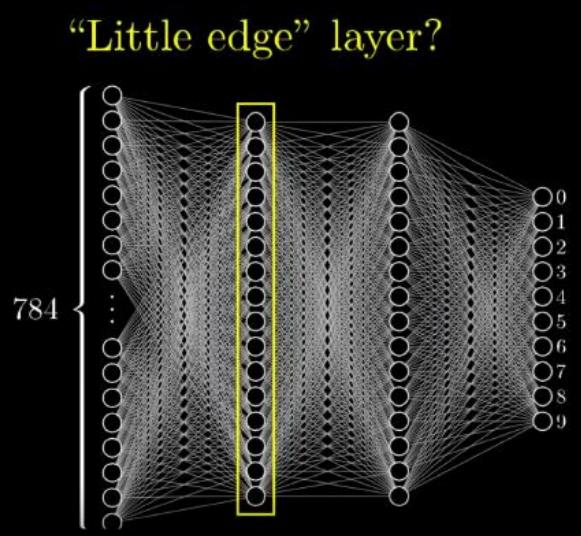


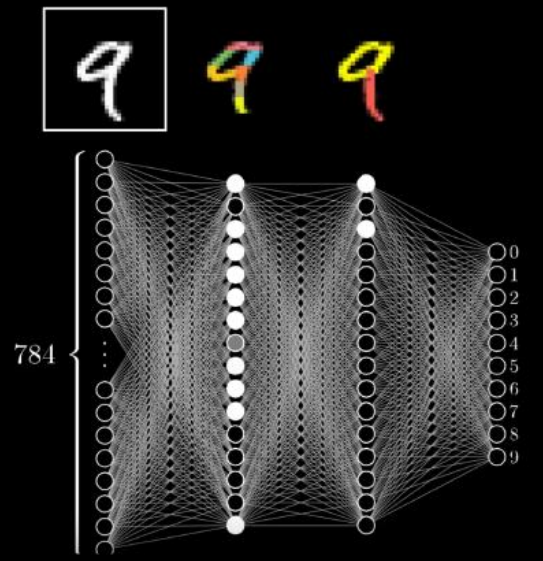
Going from the third layer to the last one just require learning which combination of sub-components corresponds to the given digits

Deep Learning

Break down the task into sub-problems
(e.g., recognize different edges from pixels)



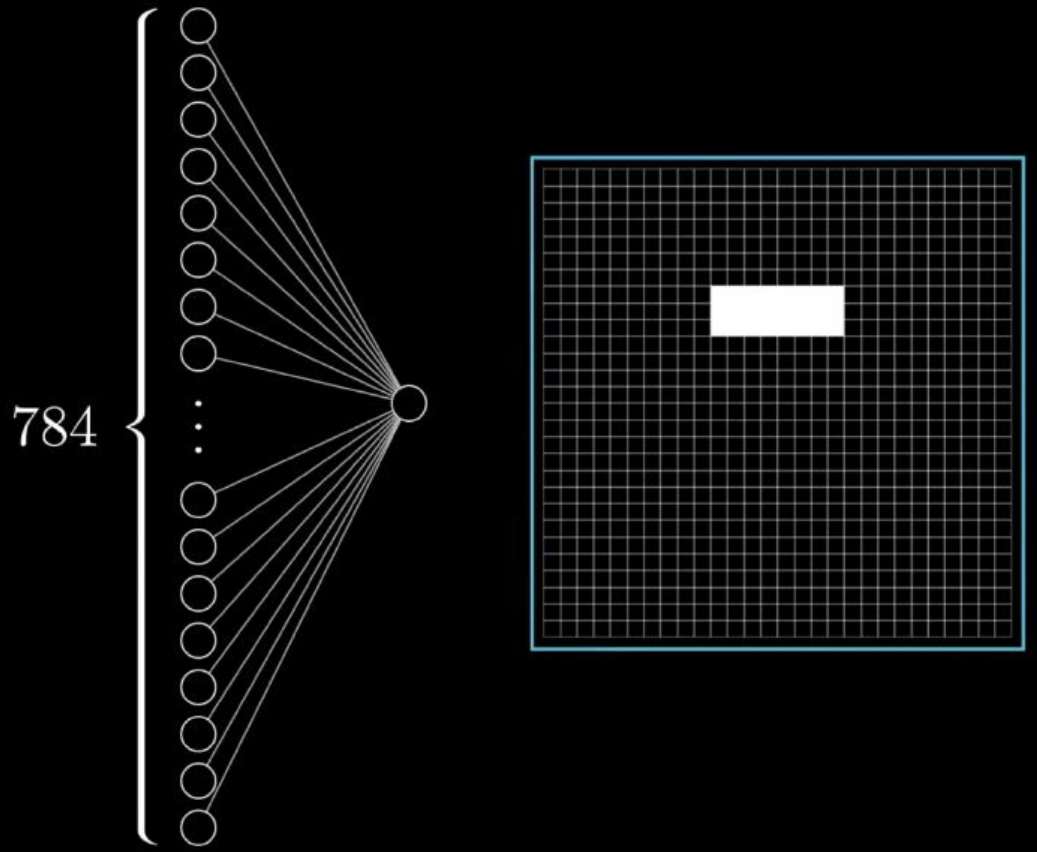




(recognize different
patterns from edges)

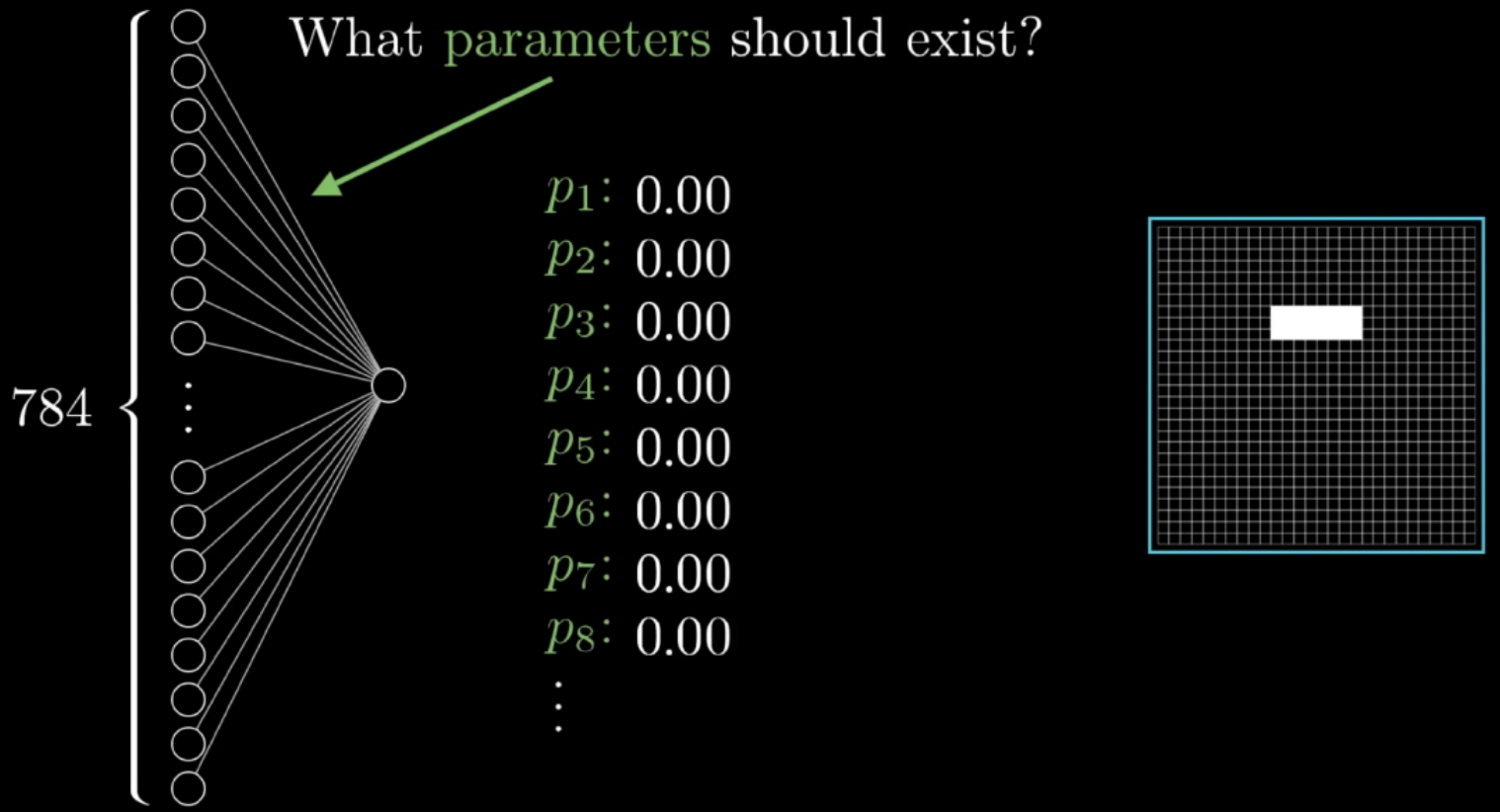
Deep Learning

Consider one specific neuron in the second layer, to “identify” whether an image has an edge in a given region or not



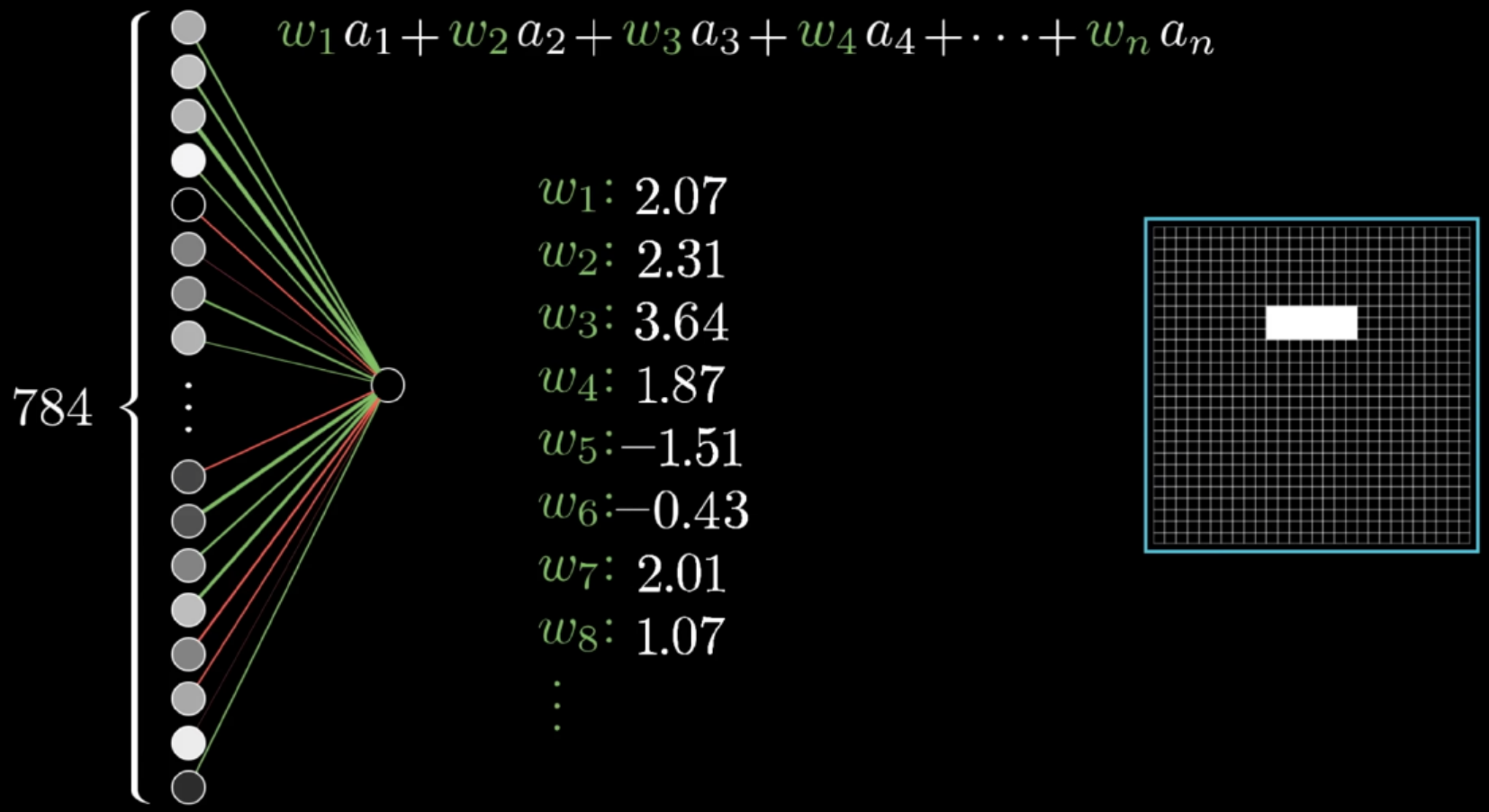
Deep Learning

Which are the best network parameters to capture this pattern?



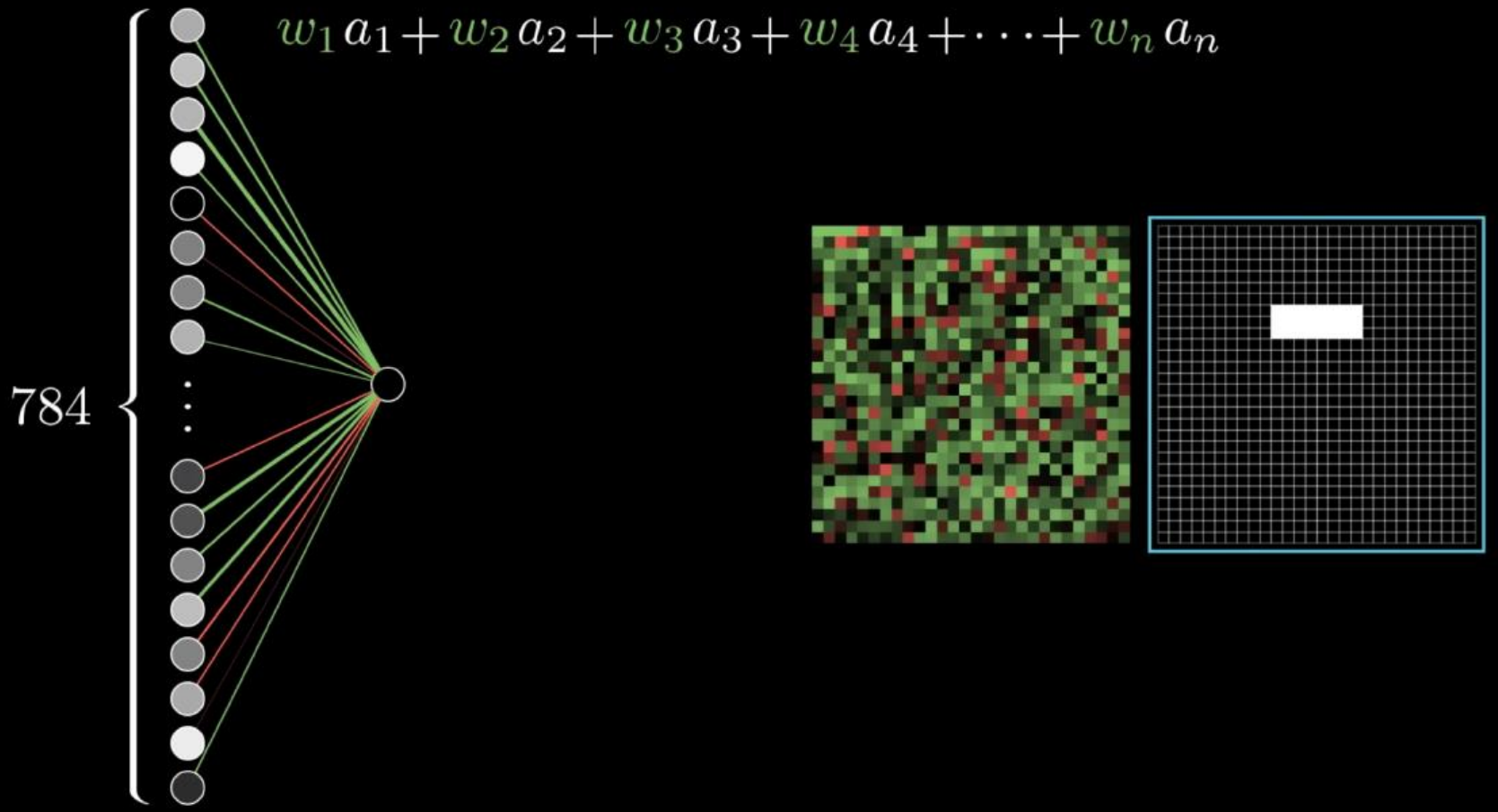
Deep Learning

Algorithm: assign (learn) a weight to each connection between our neuron and all the neuron in the previous (first) layer



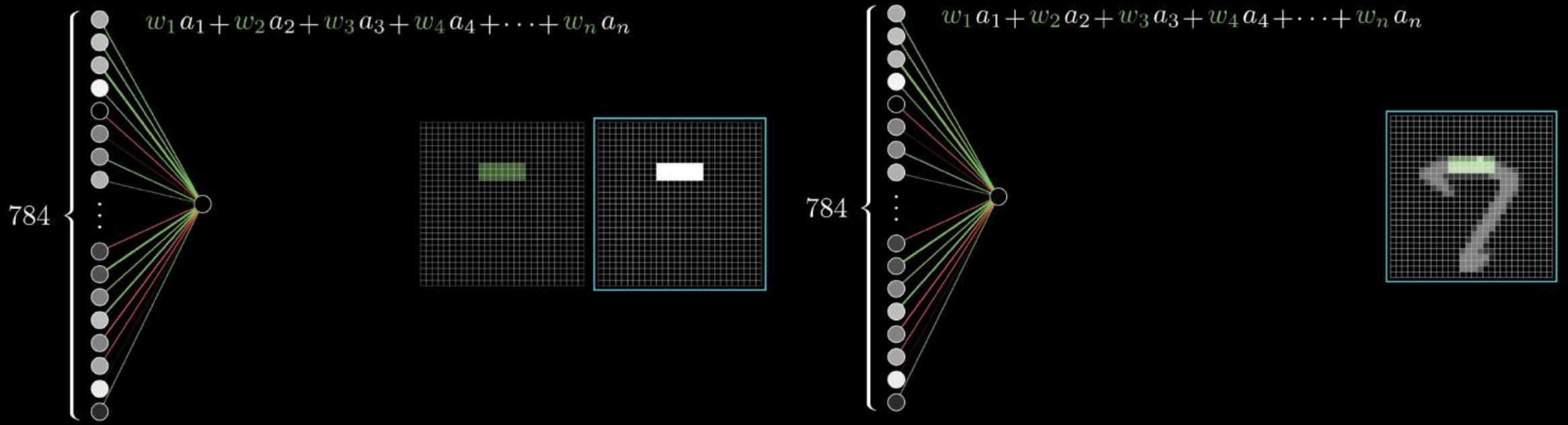
Deep Learning

Organize a grid of weights (green +, red -)



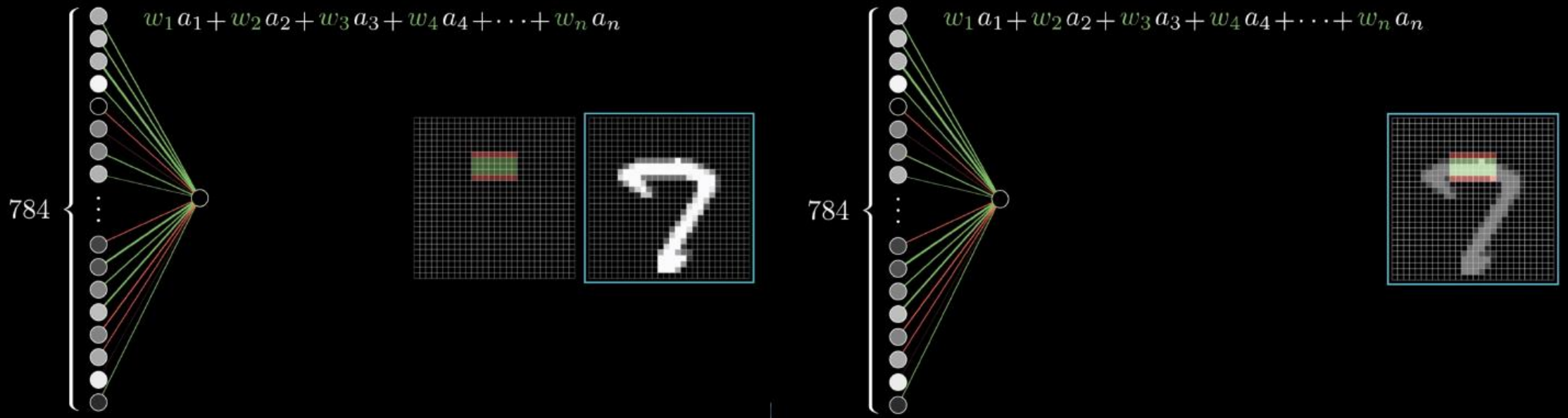
Deep Learning

Weighted sum of all pixel values = Sum only in the “region of interest”

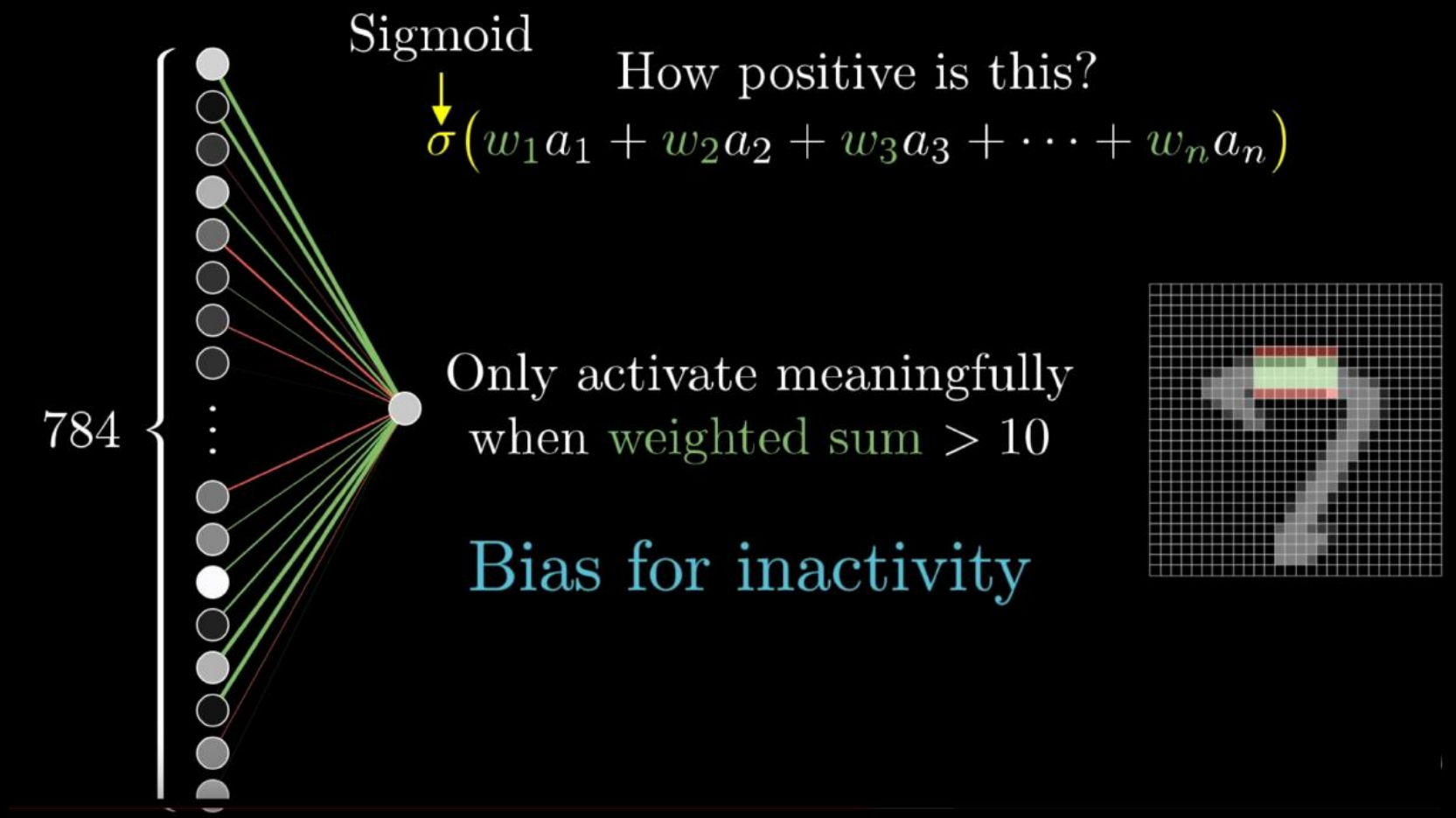


Deep Learning

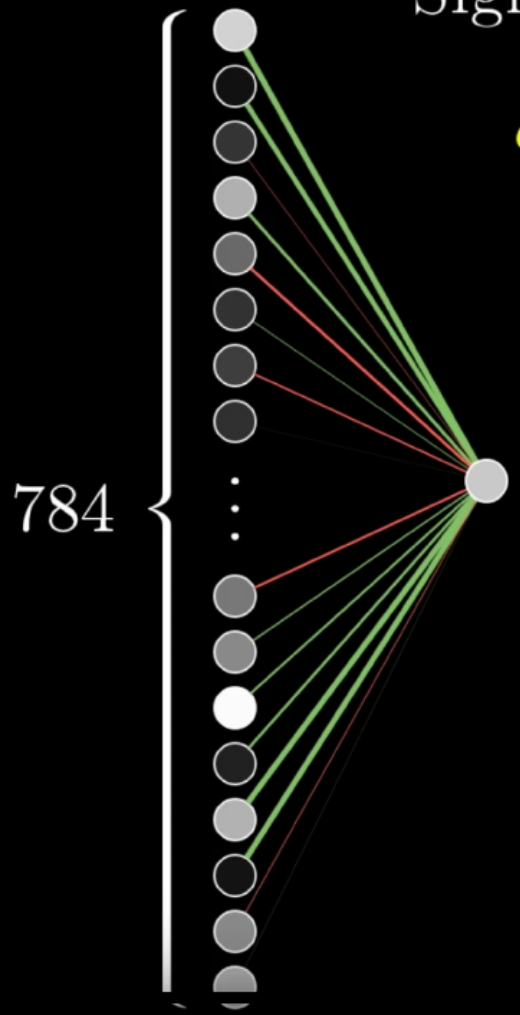
Are there any negative weights associated with the “surrounding” pixels?



Activation of a neuron is a measure of “how positive” the relevant weighted sum is



Deep Learning

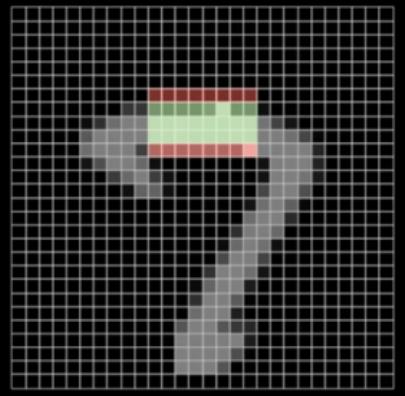


Sigmoid

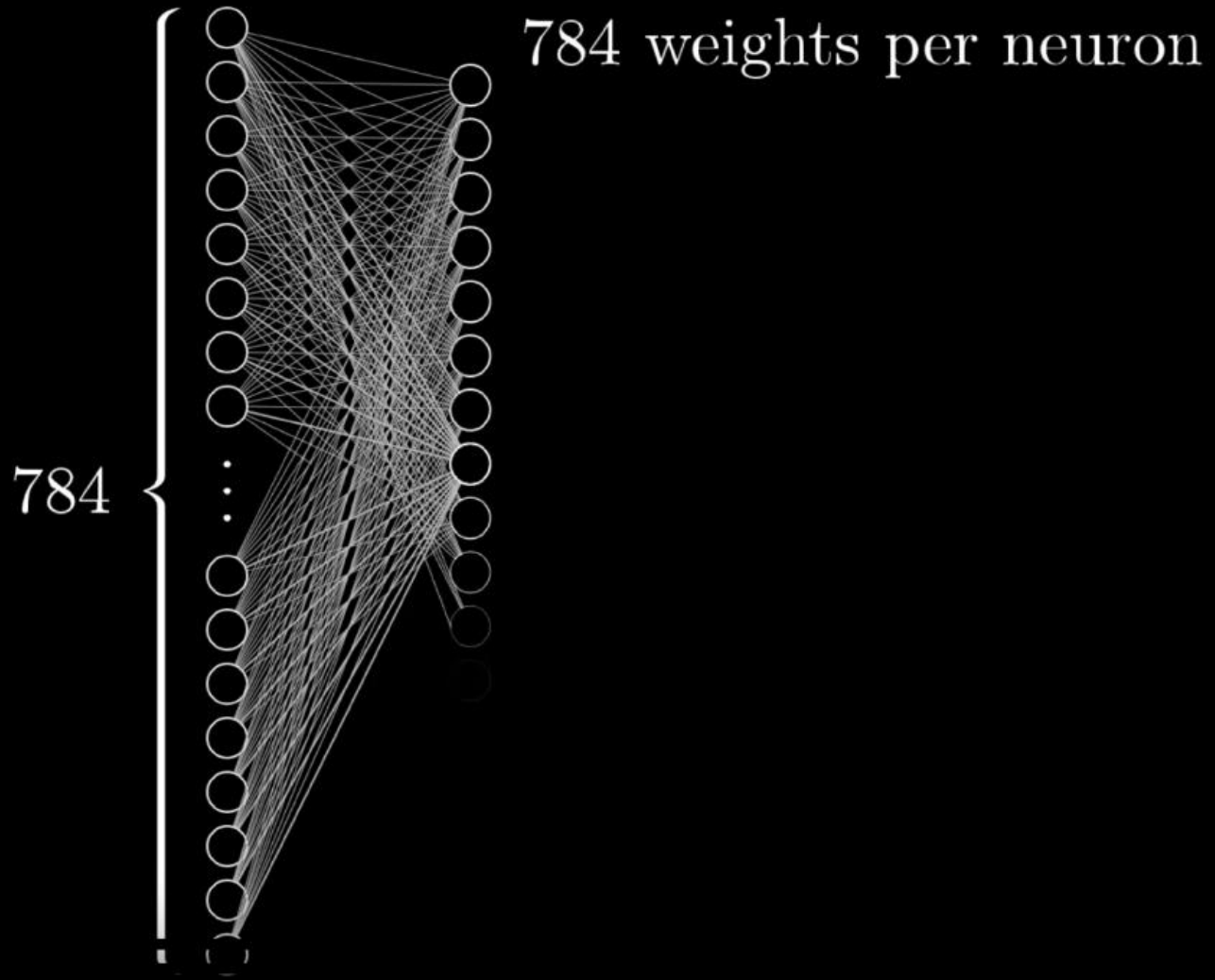
How positive is this?

$$\sigma\left(w_1 a_1 + w_2 a_2 + w_3 a_3 + \cdots + w_n a_n \boxed{-10}\right)$$

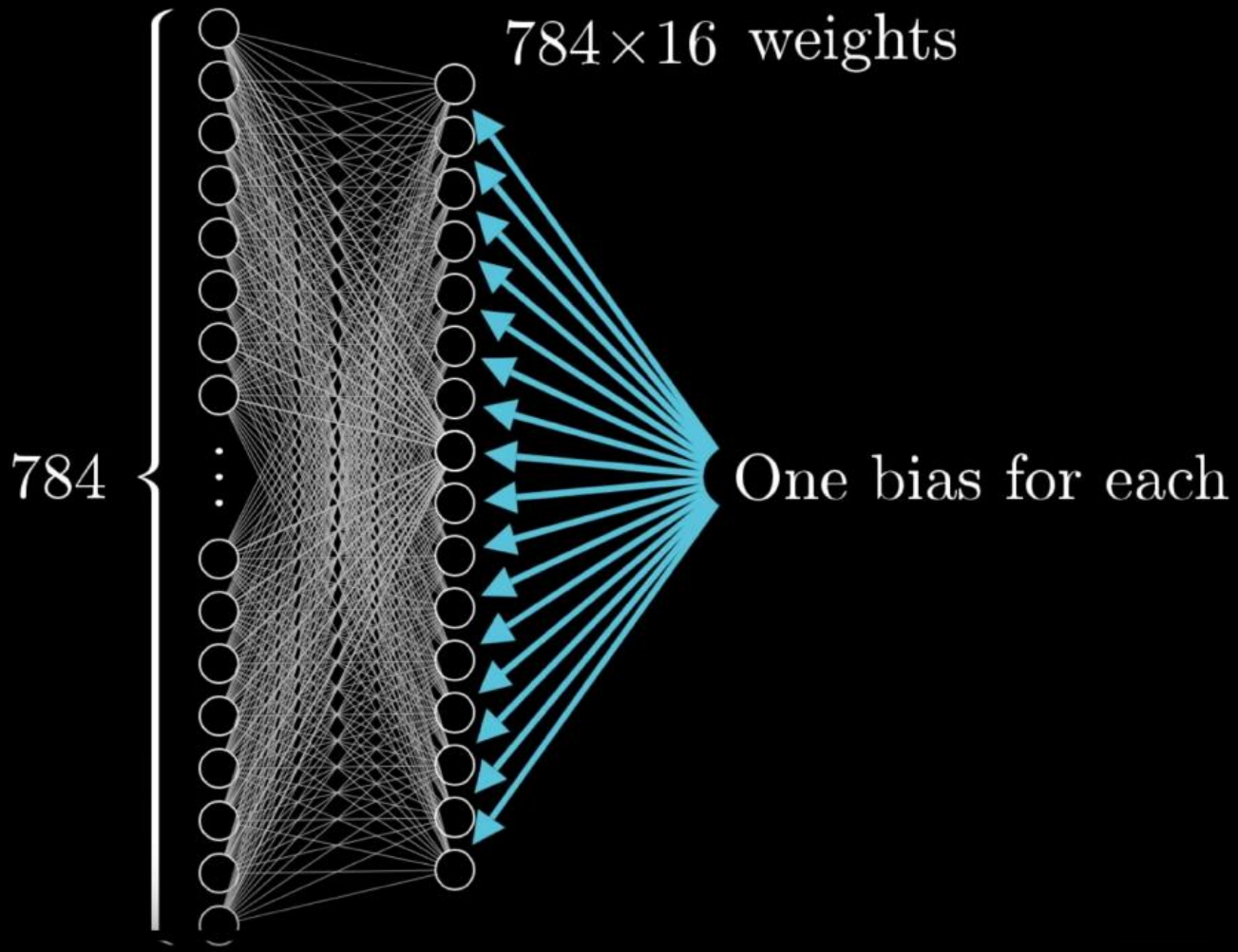
"bias"



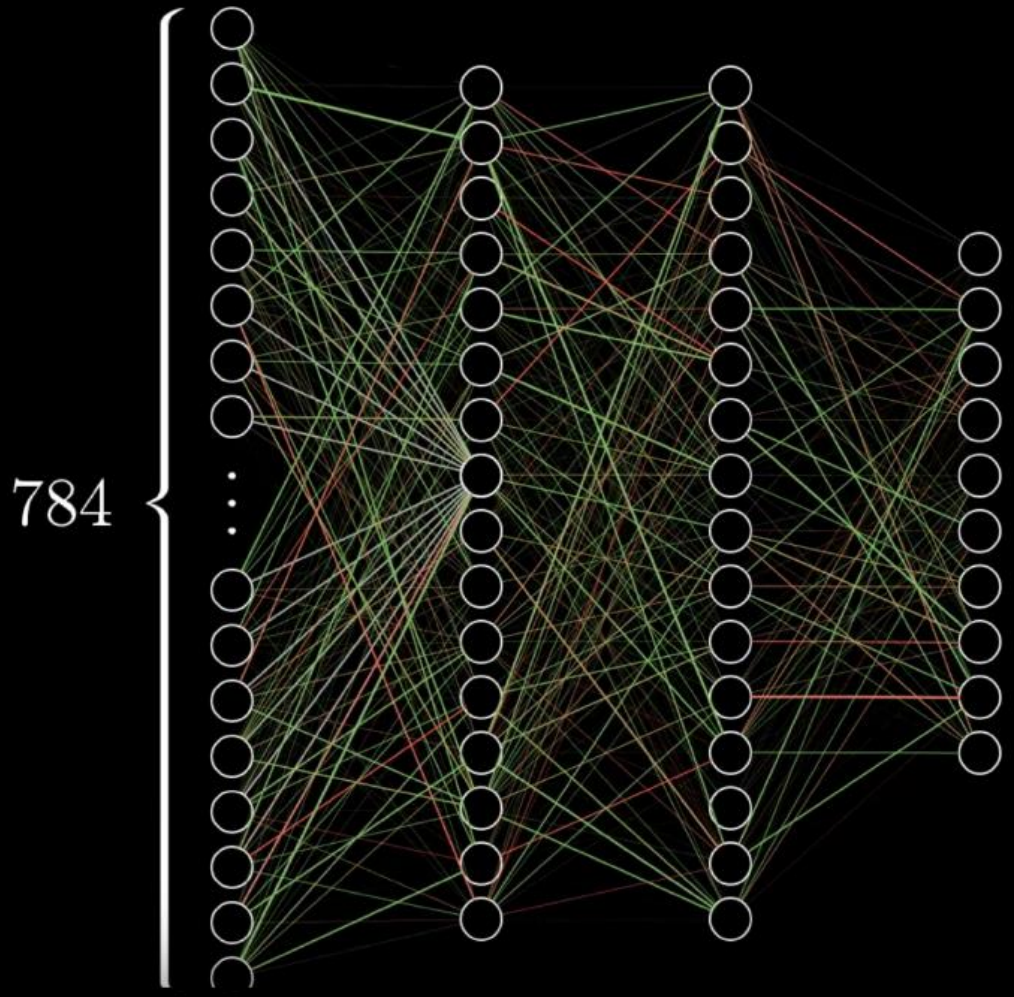
Deep Learning



Deep Learning



Deep Learning

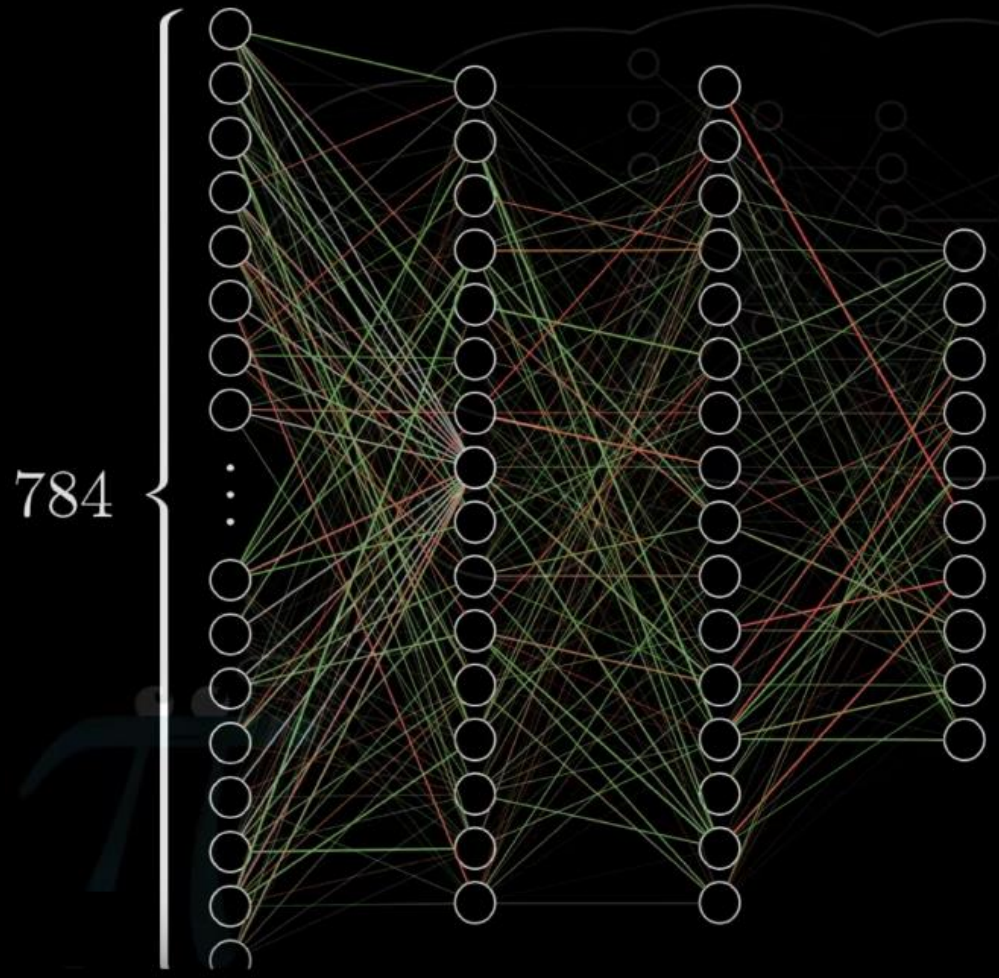


$$784 \times 16 + 16 \times 16 + 16 \times 10$$
weights

$$16 + 16 + 10$$
biases

$$13{,}002$$

Deep Learning



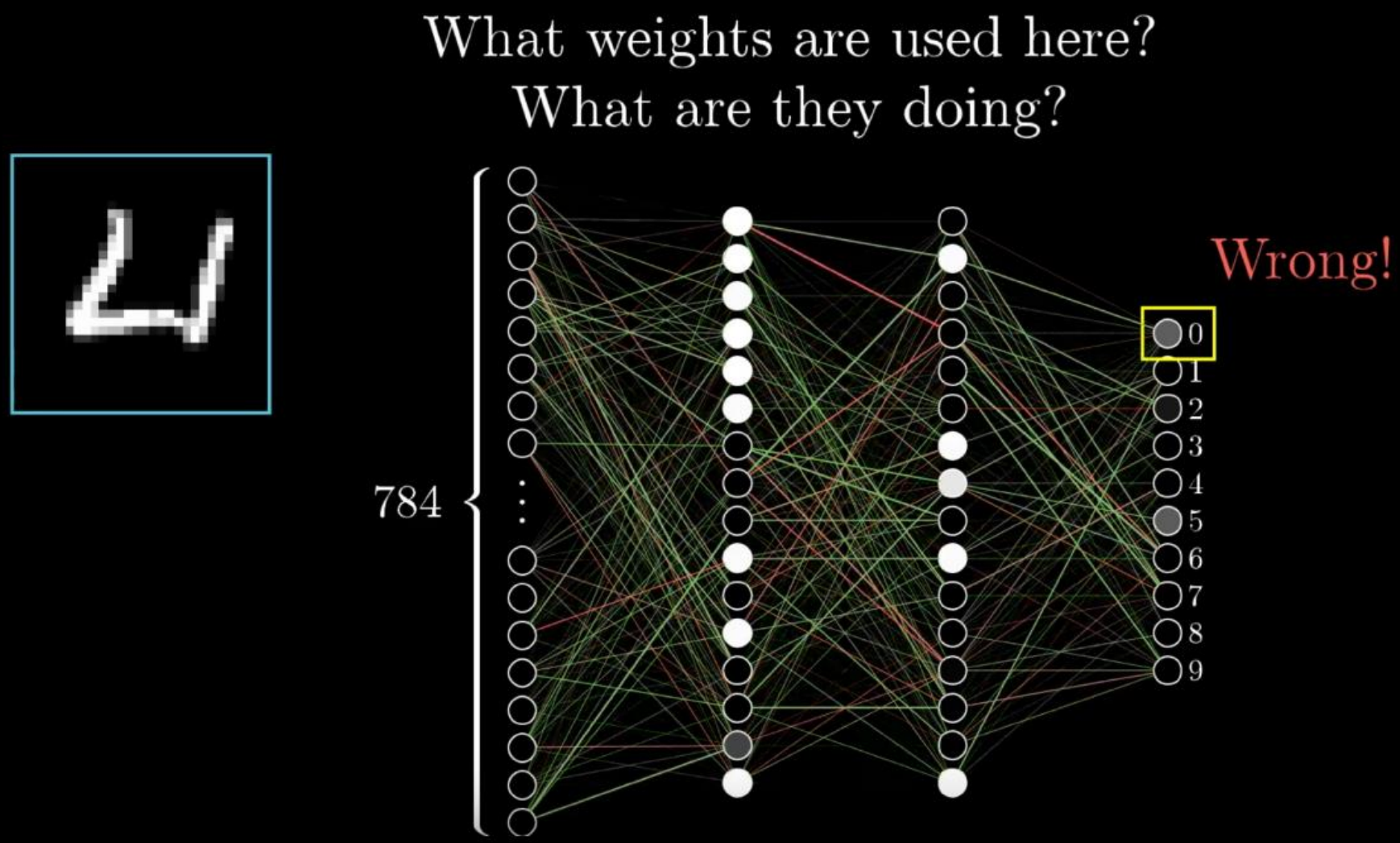
$784 \times 16 + 16 \times 16 + 16 \times 10$
weights

$16 + 16 + 10$
biases

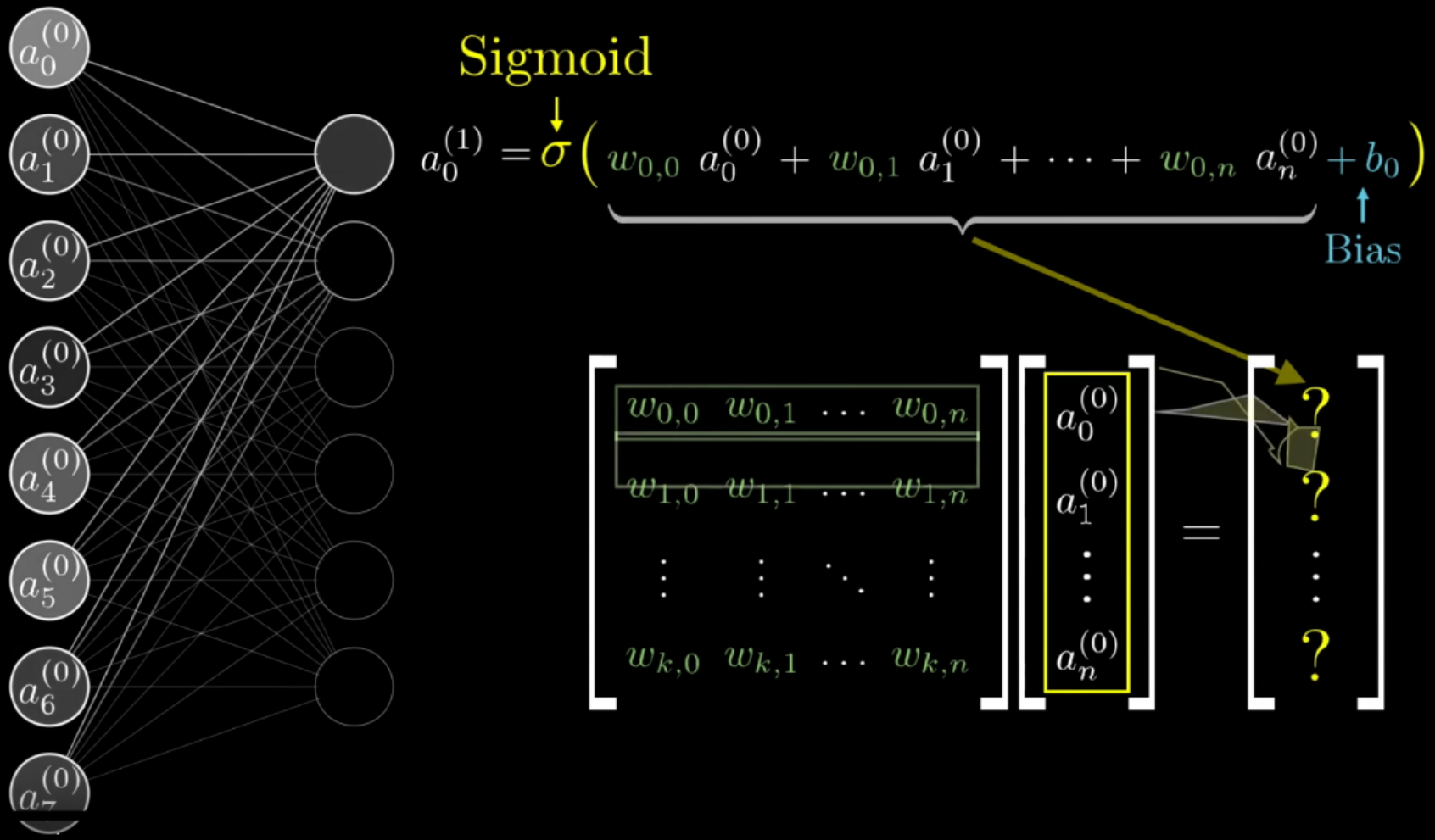
13,002

Learning $\rightarrow$ Finding the right weights and biases

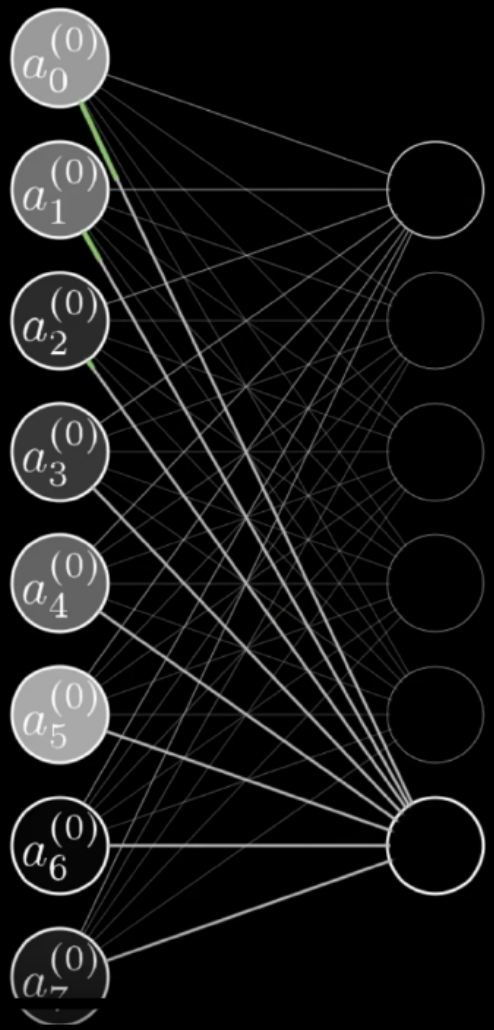
Deep Learning



Deep Learning



Deep Learning



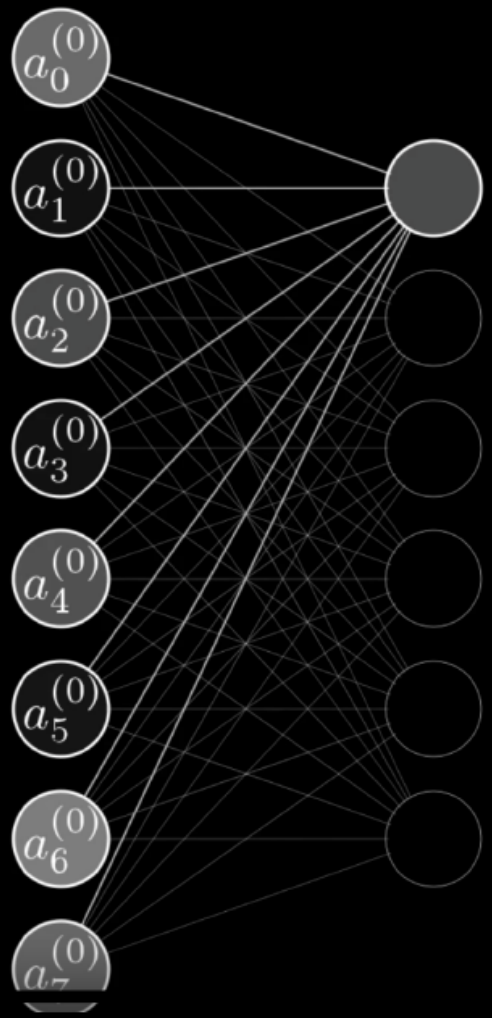
Sigmoid

$$a_0^{(1)} = \sigma\left(w_{0,0}\, a_0^{(0)} + w_{0,1}\, a_1^{(0)} + \cdots + w_{0,n}\, a_n^{(0)} + b_0 \right)$$

Bias

$$\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} = \begin{bmatrix} ? \\ ? \\ \vdots \\ ? \end{bmatrix}$$

Deep Learning



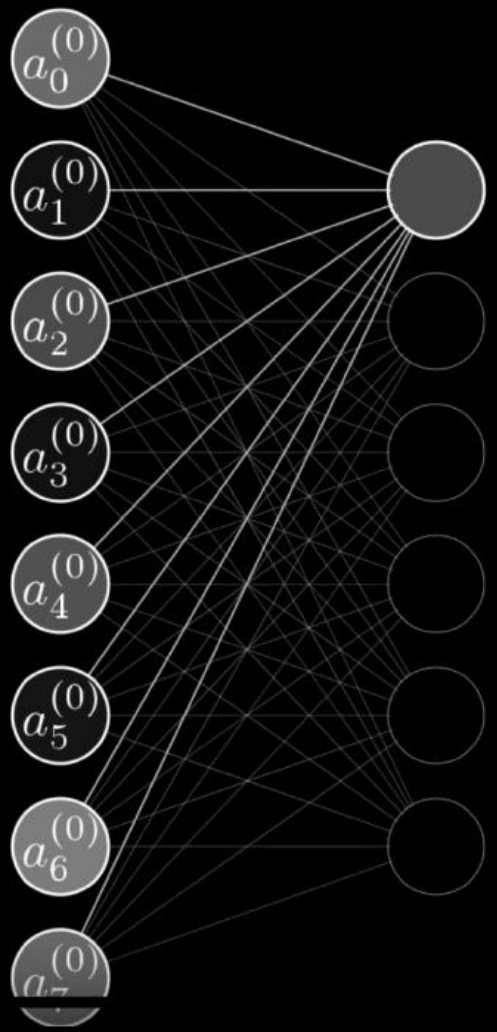
Sigmoid

$$a_0^{(1)} = \sigma\left(w_{0,0}\, a_0^{(0)} + w_{0,1}\, a_1^{(0)} + \cdots + w_{0,n}\, a_n^{(0)} + b_0 \right)$$

Bias

$$\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix}$$

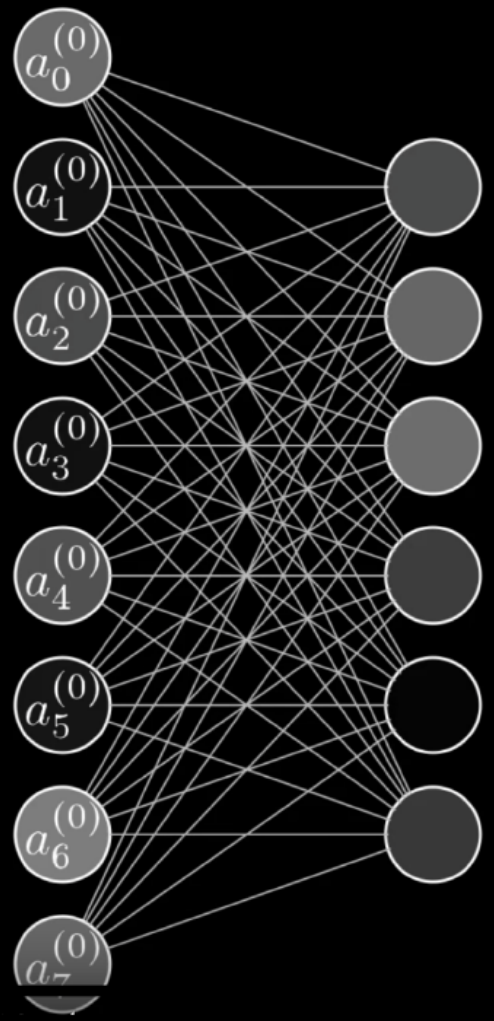
Deep Learning



$$\sigma\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix}\right) = \begin{bmatrix} \sigma(x) \\ \sigma(y) \\ \sigma(z) \end{bmatrix}$$

$$\sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix}\right)$$

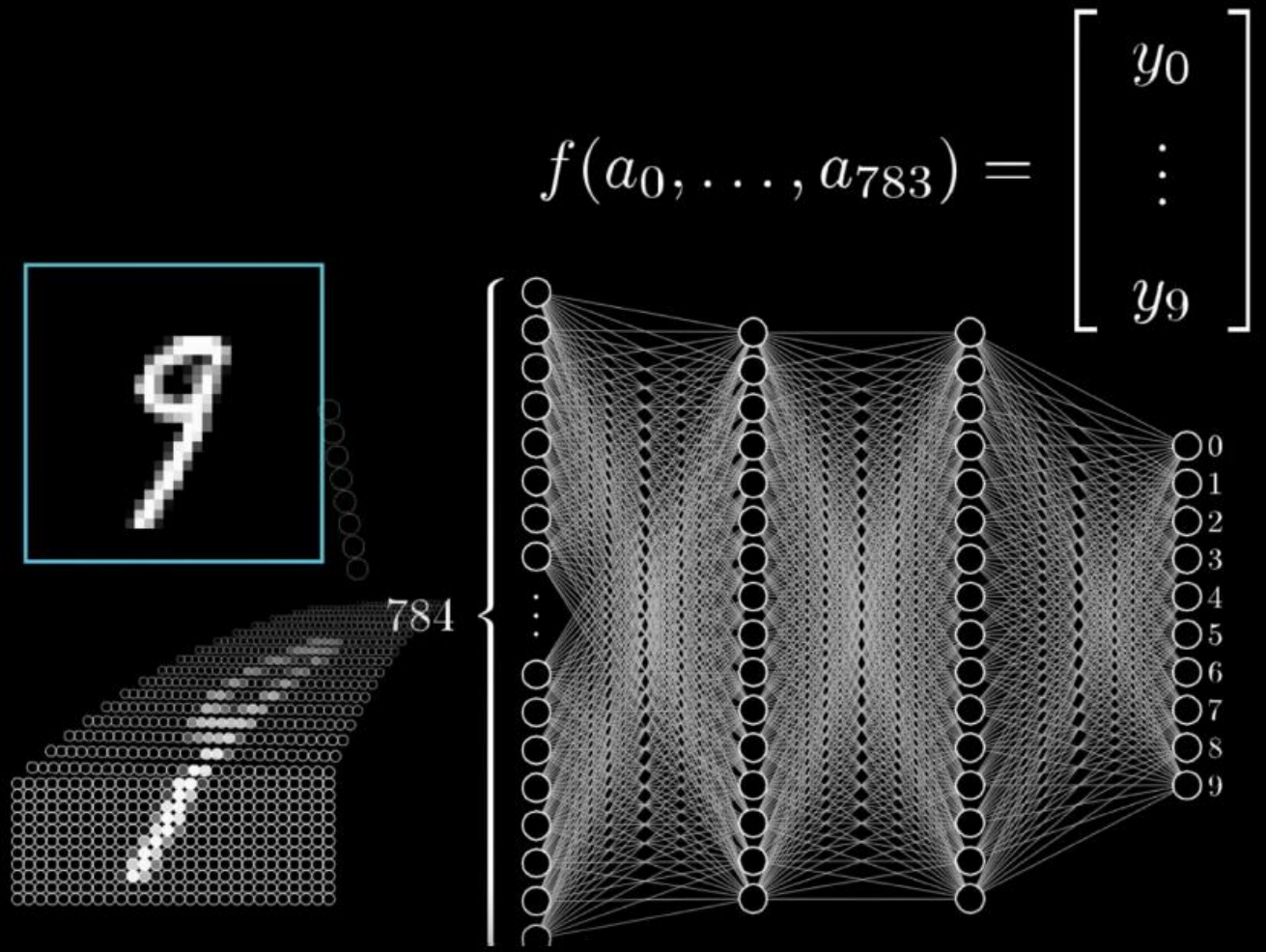
Deep Learning



$$\mathbf{a}^{(1)} = \sigma\left(\mathbf{W}\mathbf{a}^{(0)} + \mathbf{b}\right)$$

$$\sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix}\right)$$

Deep Learning



Deep Learning

Training multiple hidden layers by back propagation algorithm

30 years to implement Learning Rules

20 years to implement training performance

Problems in training performance

-Vanishing Gradient

-Overfitting

-Computational load

Vanishing gradients

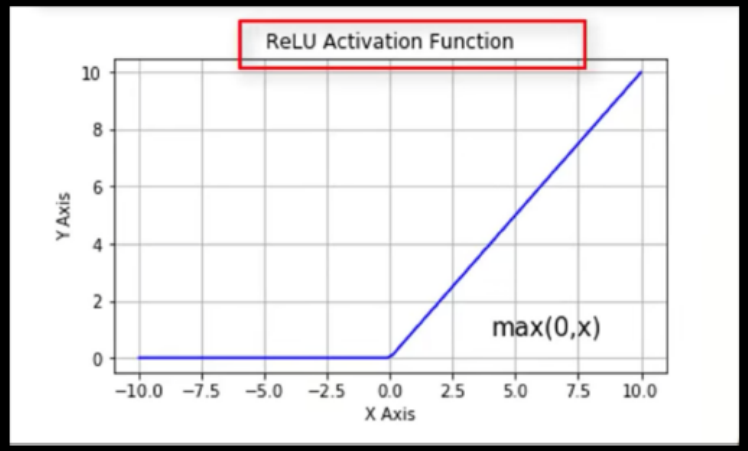
During training each weight receives an update proportional to the partial derivative of the error function with respect to the current weight in each iteration of training.

In some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training.

As one example of the problem cause, traditional activation functions such as the hyperbolic tangent function have gradients in the range (-1, 1), and backpropagation computes gradients by the chain rule. This has the effect of multiplying n of these small numbers to compute gradients of the "front" layers in an n-layer network, meaning that the gradient (error signal) decreases exponentially with n while the front layers train very slowly.

Vanishing gradients

Can be solved using Rectified Linear Unit function (ReLU) and its derivative as activation function

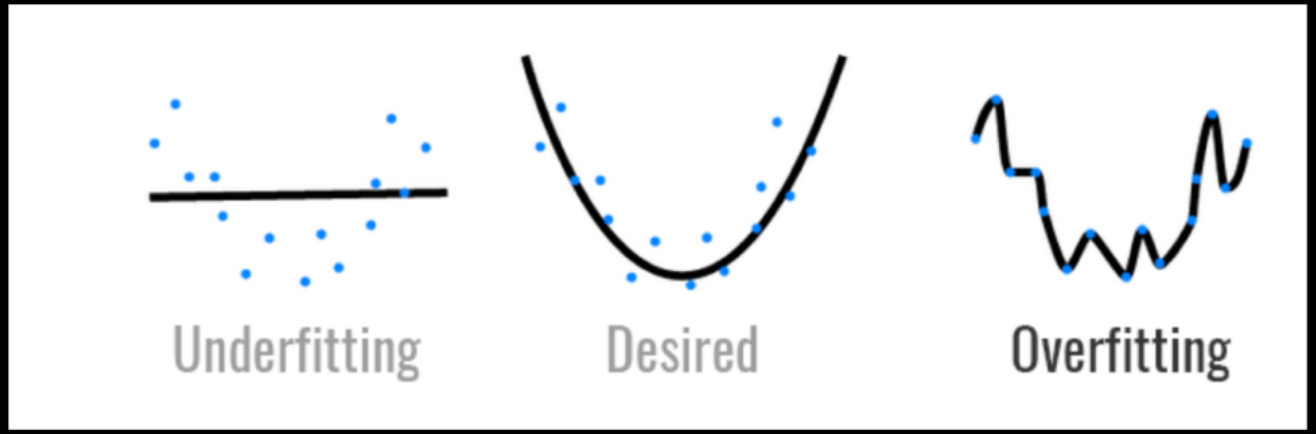


$$\varphi(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases}$$
$$= \max(0, x)$$

$$\varphi'(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

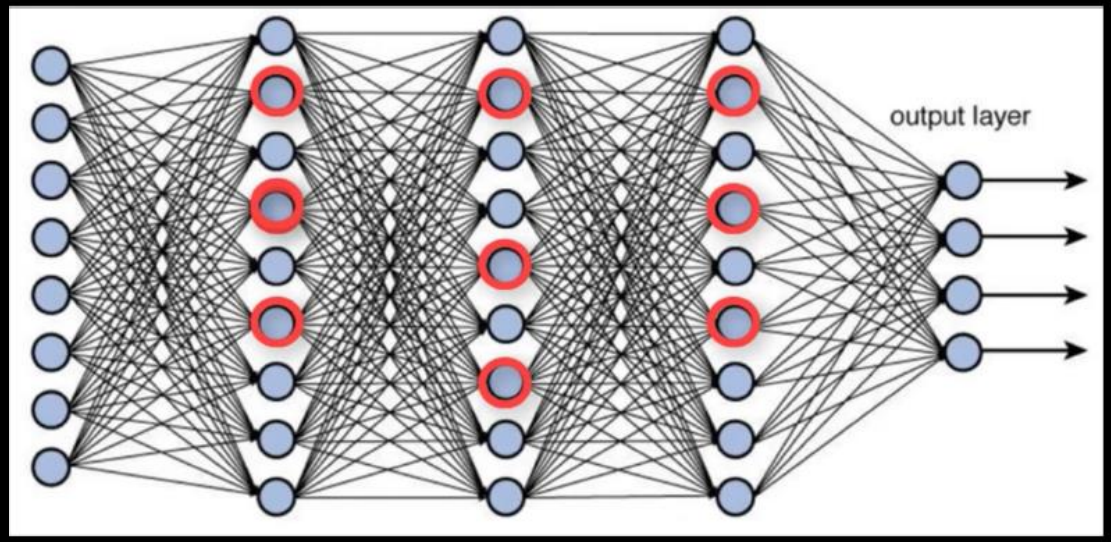
Overfitting

It refers to a model that models the training data too well. Instead of learning the genral **distribution** of the data, the model learns the *expected output* for every data point.

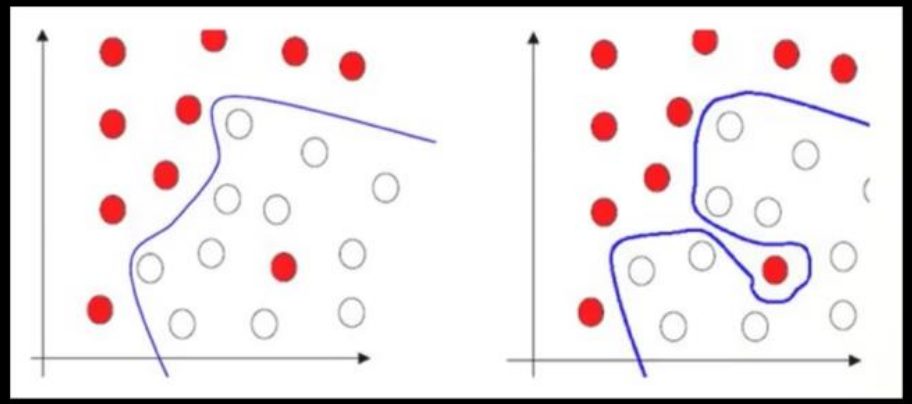


Overfitting

Can be solved using Dropout or Regularization



Dropout



Regulariztion

Deep Learning

Computational load

Can be solved by GPU, Batch Normalization method

SOFTWARE CODES

Matlab: essential functions and scripts

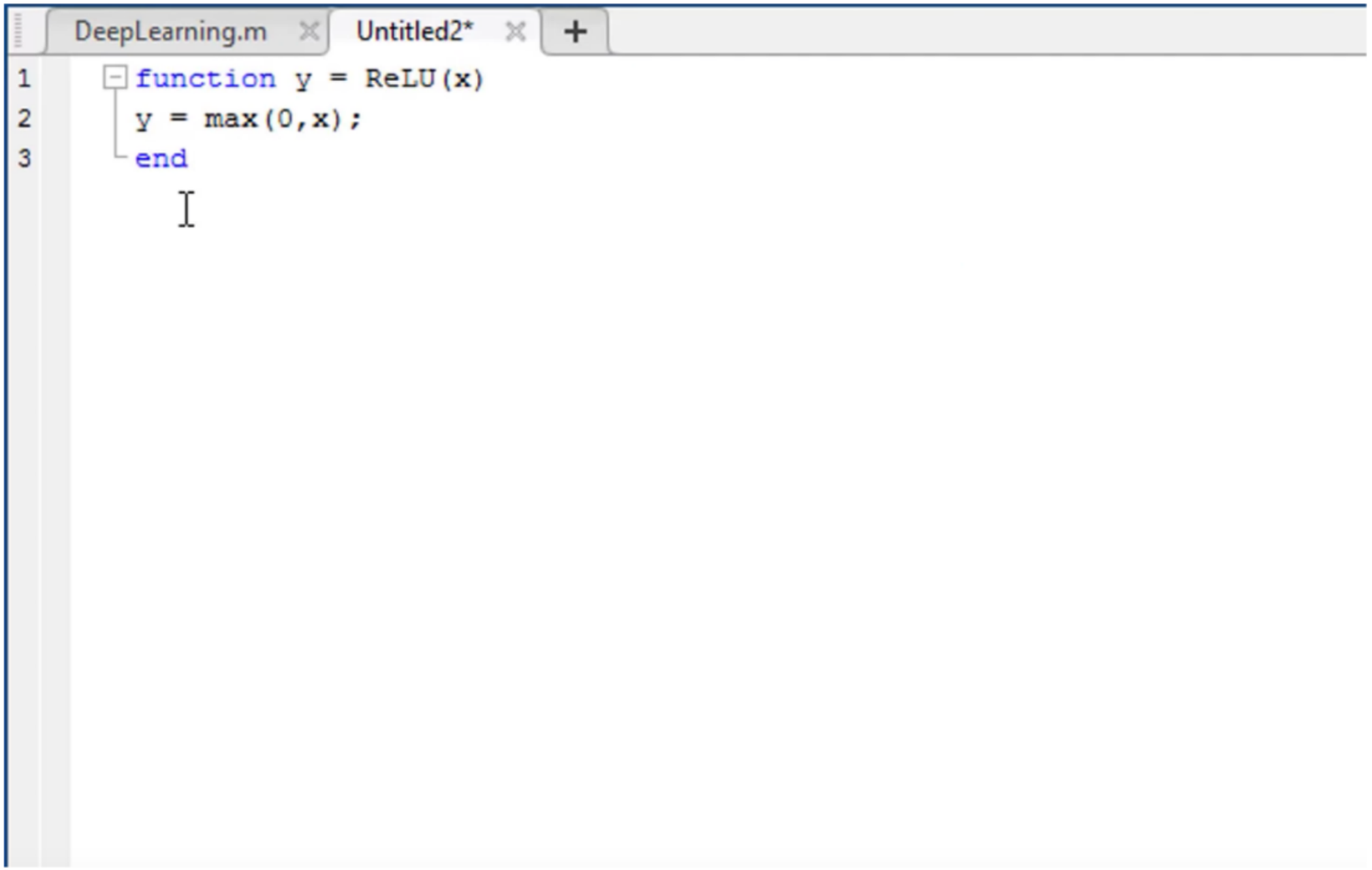
Matlab: simple examples

Deep Learning

DeepLearning.m* | Untitled2*

```
1   function [w1, w2, w3, w4] = DeepLearning(w1, w2, w3, w4, input_Image, correct_Output)
2   alpha = 0.01; %to control the learning rate
3
4   N = 5;
5   for k = 1:N
6       reshaped_input_Image = reshape(input_Image(:,:,k), 25, 1);
7
8       input_of_hidden_layer1 = w1*reshaped_input_Image;
9       output_of_hidden_layer1 = ReLU(input_of_hidden_layer1);
10
11      input_of_hidden_layer2 = w2* output_of_hidden_layer1;
12      output_of_hidden_layer2 = ReLU(input_of_hidden_layer2);
13
14      input_of_hidden_layer3 = w3* output_of_hidden_layer2;
15      output_of_hidden_layer3 = ReLU(input_of_hidden_layer3);
16
17      input_of_output_node = w4* output_of_hidden_layer3;
18      final_output = Softmax(input_of_output_node)
```

Deep Learning



Deep Learning

DeepLearning.m | ReLU.m | Untitled3*

```
1   function y = Softmax(x)
2       ex = exp(x);
3       y = ex/sum(ex);
4   end
```

Deep Learning

DeepLearning.m | ReLU.m | Softmax.m | TrainingNetwork.m*

```
input_Image = zeros(5,5,5);

input_Image(:,:,1) = [1 0 0 1 1;
                      1 1 0 1 1;
                      1 1 0 1 1;
                      1 1 0 1 1;
                      1 0 0 0 1;
                      ];
input_Image(:,:,2) = [0 0 0 0 1;
                      1 1 1 1 0;
                      1 0 0 0 1;
                      0 1 1 1 1;
                      0 0 0 0 0;
                      ];
input_Image(:,:,3) = [
```

Deep Learning

DeepLearning.m | ReLU.m | Softmax.m | TrainingNetwork.m*

```
                          1 1 1 1 0;
                          0 0 0 0 1;
                          ];
    correct_Output = [1 0 0 0 0;
                      0 1 0 0 0;
                      0 0 1 0 0;
                      0 0 0 1 0;
                      0 0 0 0 1;
                      ];
    w1 = 2*rand(20,25)-1;
    w2 = 2*rand(20,20)-1;
    w3 = 2*rand(20,20)-1;
    w4 = 2*rand(5,20)-1;

    for epoch = 1:10000
        [w1, w2, w3, w4] = DeepLearning(w1, w2, w3, w4, input_Image, correct_Output);
    end
    save('DeepNeuralNetwork.mat')
```

Deep Learning

DeepLearning.m | ReLU.m | Softmax.m | TrainingNetwork.m | TestDeepLearning.m

```
1 -    load('DeepNeuralNetwork.mat');
2
3 -    input_Image = [1 0 0 1 1;
4                     1 1 0 1 1;
5                     1 1 0 1 1;
6                     1 1 0 1 1;
7                     1 0 0 0 1;
8                     ];
9
10 -   input_Image = reshape(input_Image, 25, 1);
11
12 -   input_of_hidden_layer1 = w1*input_Image;
13 -   output_of_hidden_layer1 = ReLU(input_of_hidden_layer1);
```

Command Window

New to MATLAB? See resources for Getting Started.

```
>> TestDeepLearning

final_output =

    1.0000
    0.0000
    0.0000
    0.0000
    0.0000

fx >>
```

Deep Learning

DeepLearning.m | ReLU.m | Softmax.m | TrainingNetwork.m | TestDeepLearning.m*

```
8                    ];
9
10 -    input_Image = reshape(input_Image, 25, 1);
11
12 -    input_of_hidden_layer1 = w1*input_Image;
13 -    output_of_hidden_layer1 = ReLU(input_of_hidden_layer);
14
15 -    input_of_hidden_layer2 = w2*output_of_hidden_layer1;
16 -    output_of_hidden_layer2 = ReLU(input_of_hidden_layer2);
17
18 -    input_of_hidden_layer3 = w3*output_of_hidden_layer2;
19 -    output_of_hidden_layer3 = ReLU(input_of_hidden_layer3);
20
21      input_of_output_node = w4*output_of_h
```

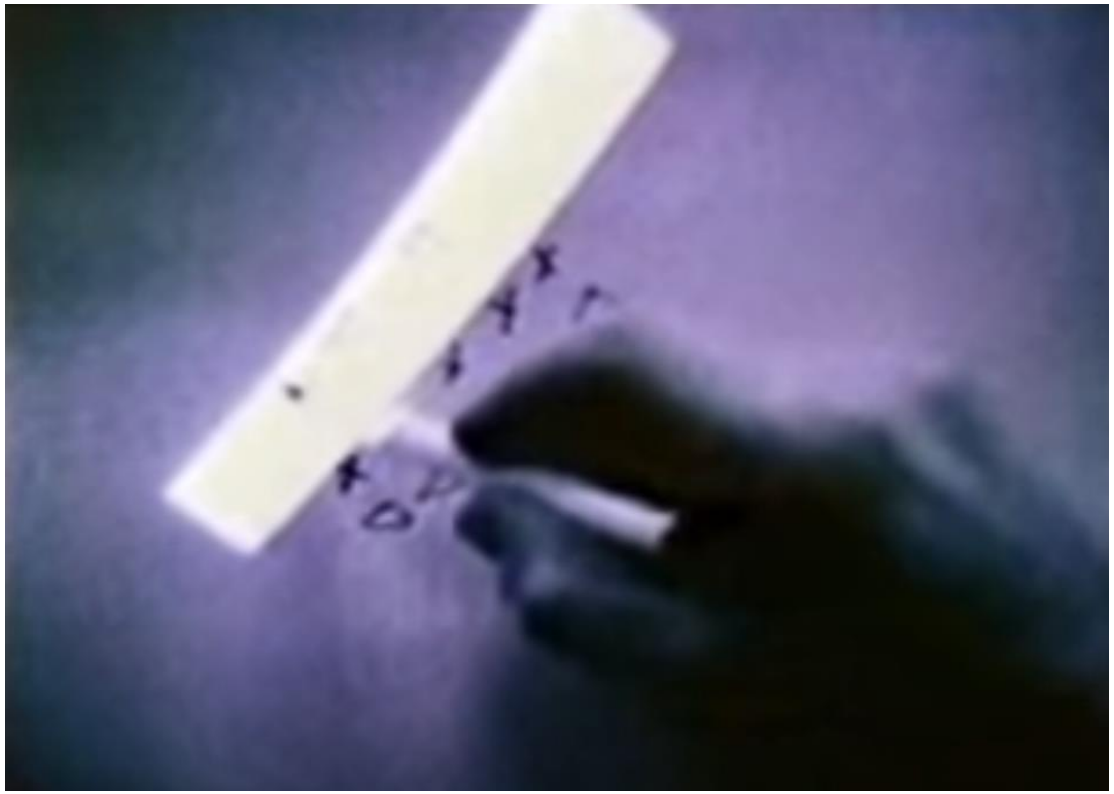
Command Window

New to MATLAB? See resources for Getting Started.

```
>> TestDeepLearning
```

CONVOLUTIONAL NEURAL NETWORKs

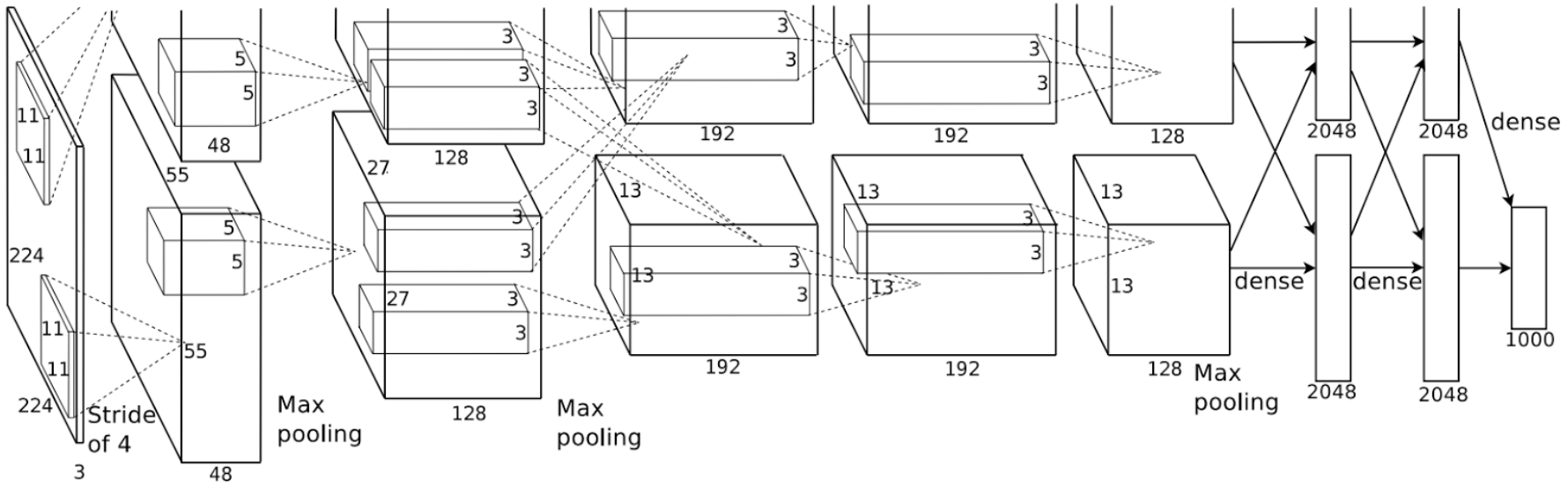
Convolutional Neural Networks



Hubel and Wiesel

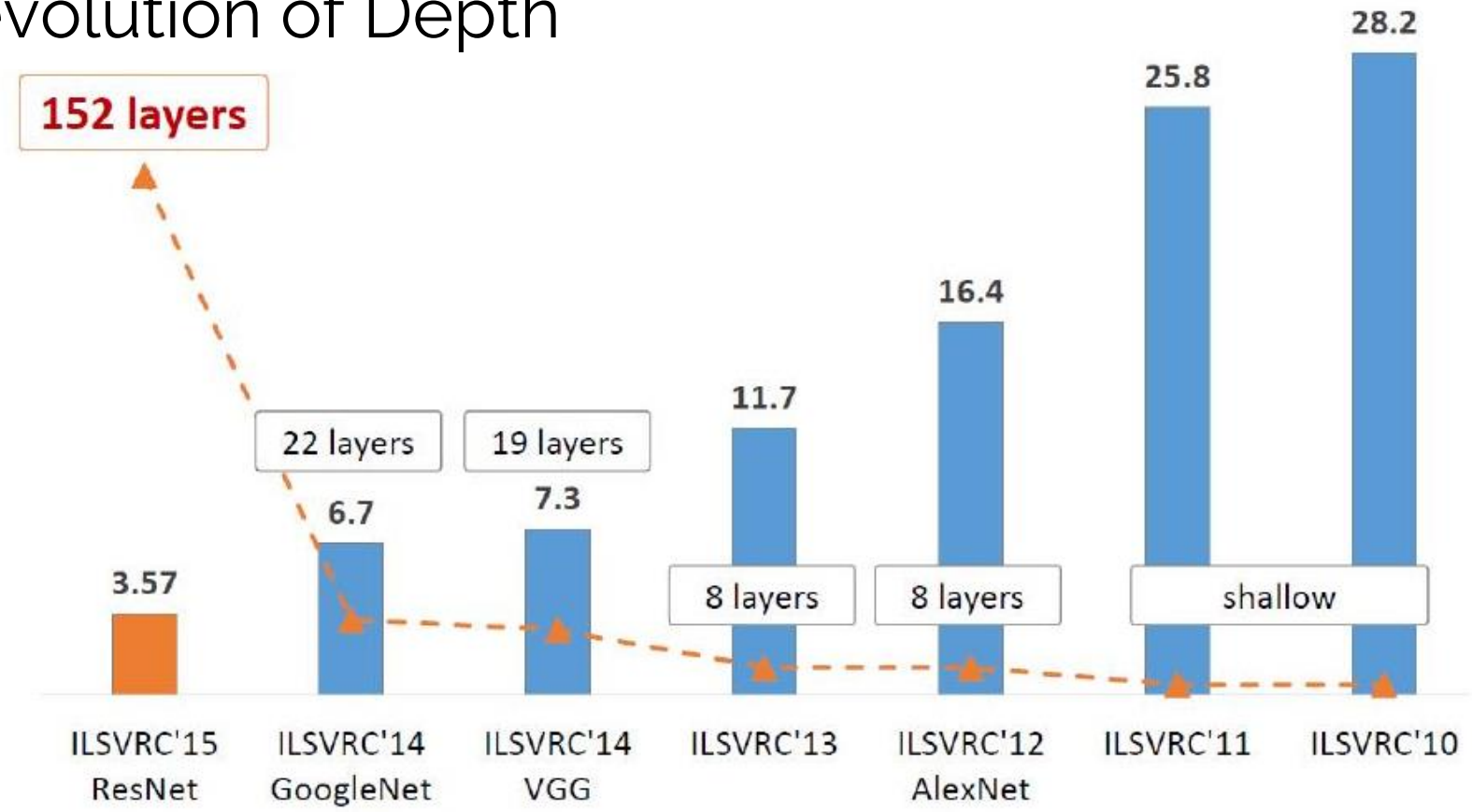
https://www.youtube.com/watch?v=Cw5PKV9Rj30

Convolutional Neural Networks



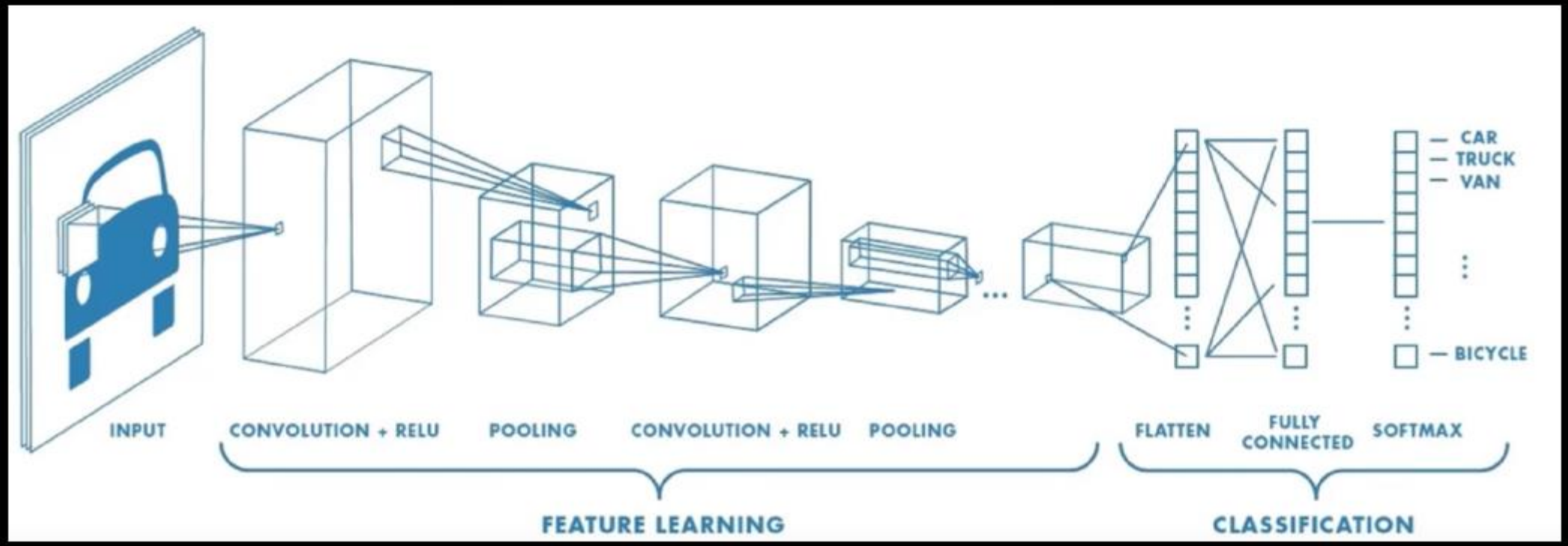
Krizhevsky, Sutskever, and Hinton, 2012

Revolution of Depth



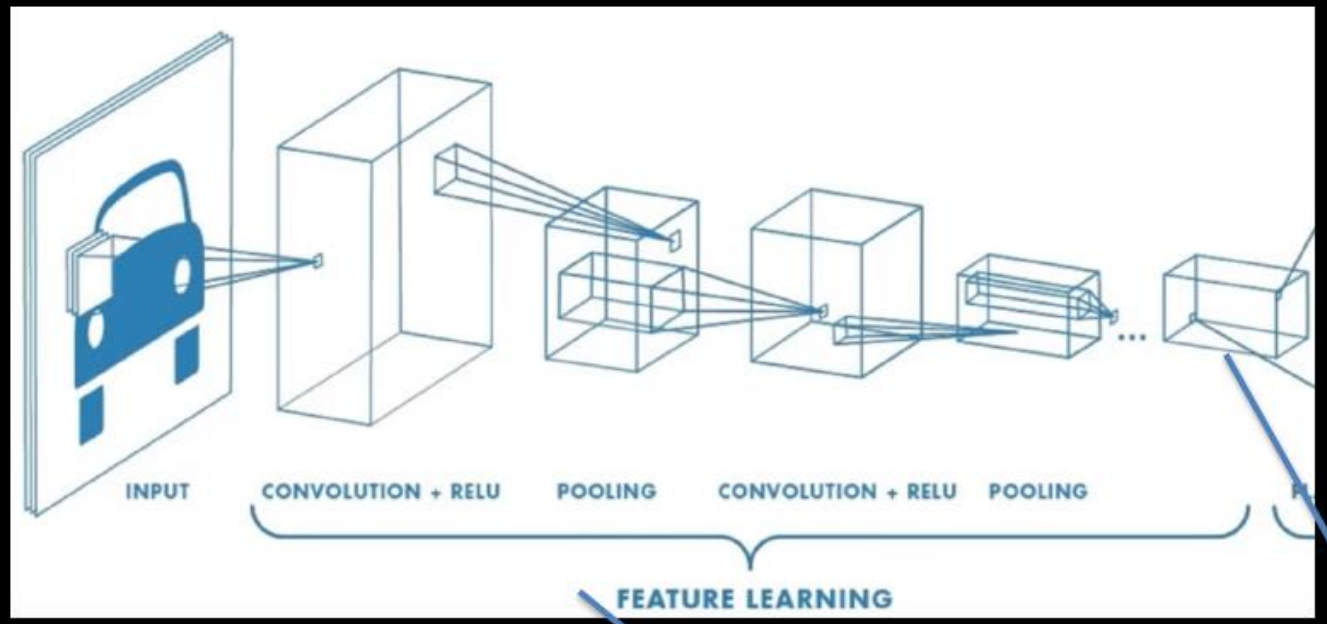
ImageNet Classification top-5 error (%)

Convolutional Neural Networks



Convolutional Neural Networks

Feature Learning



Feature extraction network

Convolutional layers

Paired pooling layers

Extracted features

Convolutional Neural Networks

Convolution operation: a collection of digital filters (2D)

Pooling operation combines adjacent pixels into 1 pixel (2D) (reduce the size of the image)

Convolutional Neural Networks

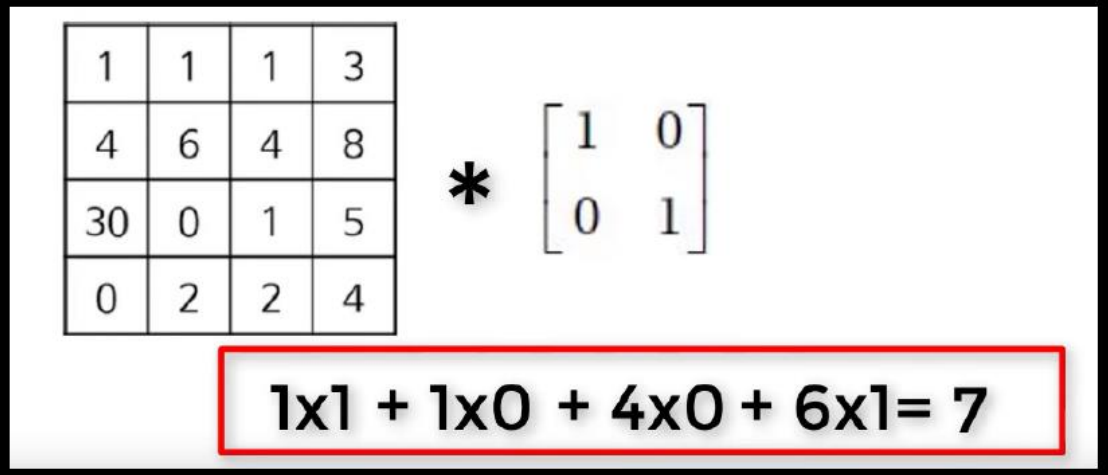
Convolution layers converts images into features map

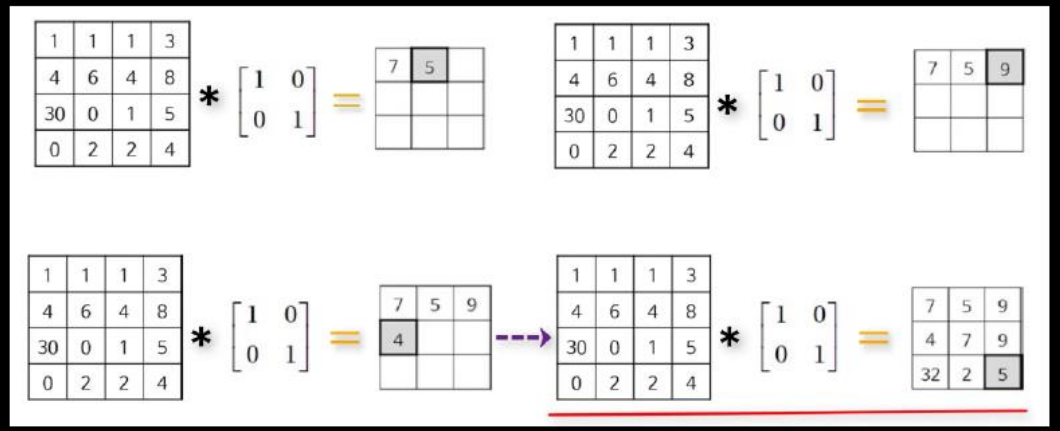
Different filters (convolutional filters) convert images into different feature maps

N convolution layers x M convolutional filters→ NxM feature maps

Convolutional Neural Networks

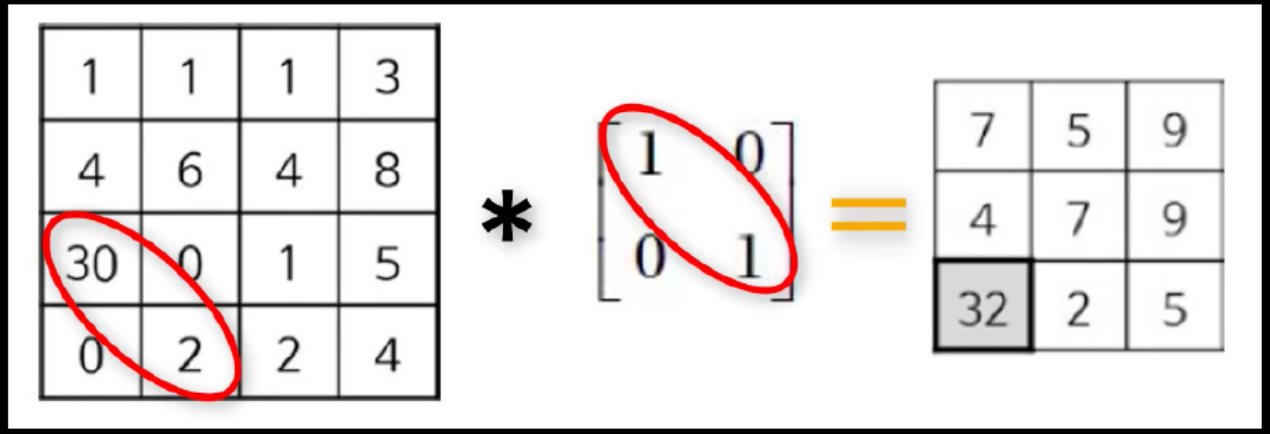
Convolution filter



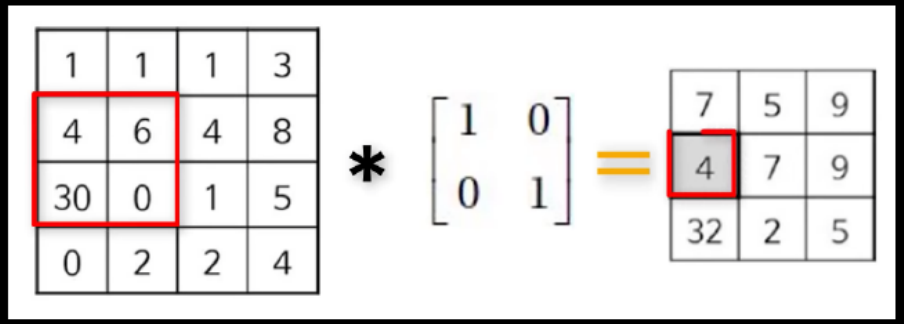


Convolutional Neural Networks

Convolution filter: first example



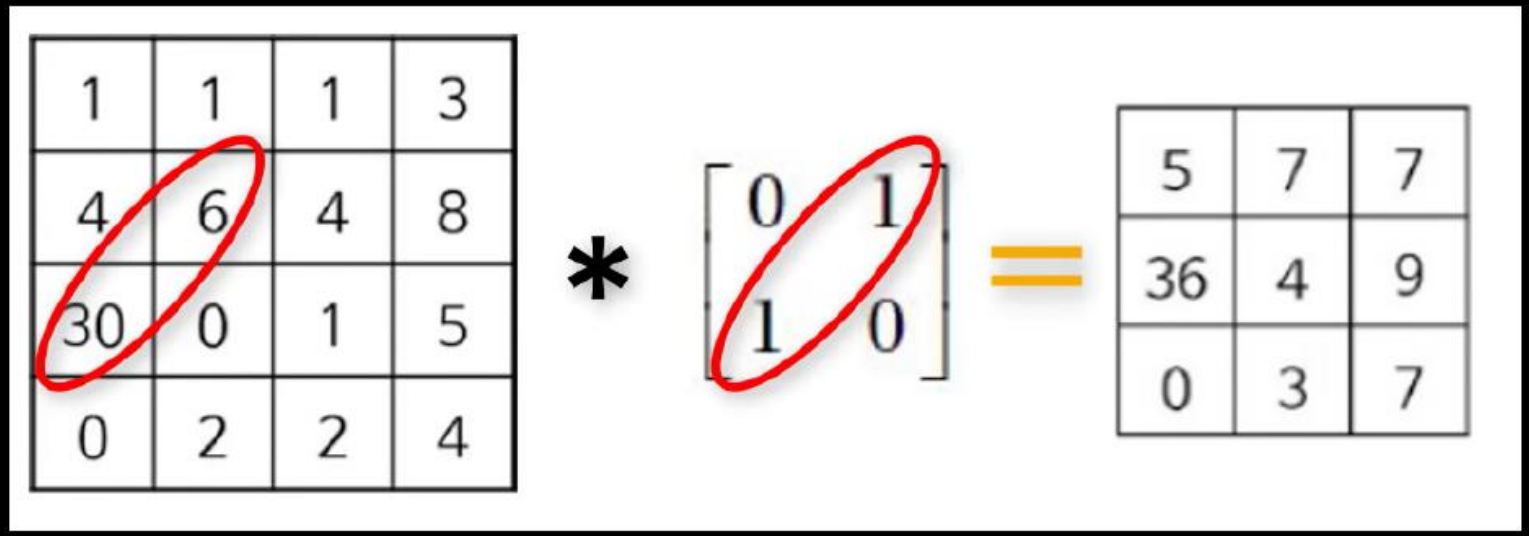
High value when filter mach image patch



Low value when filter does not mach image patch

Convolutional Neural Networks

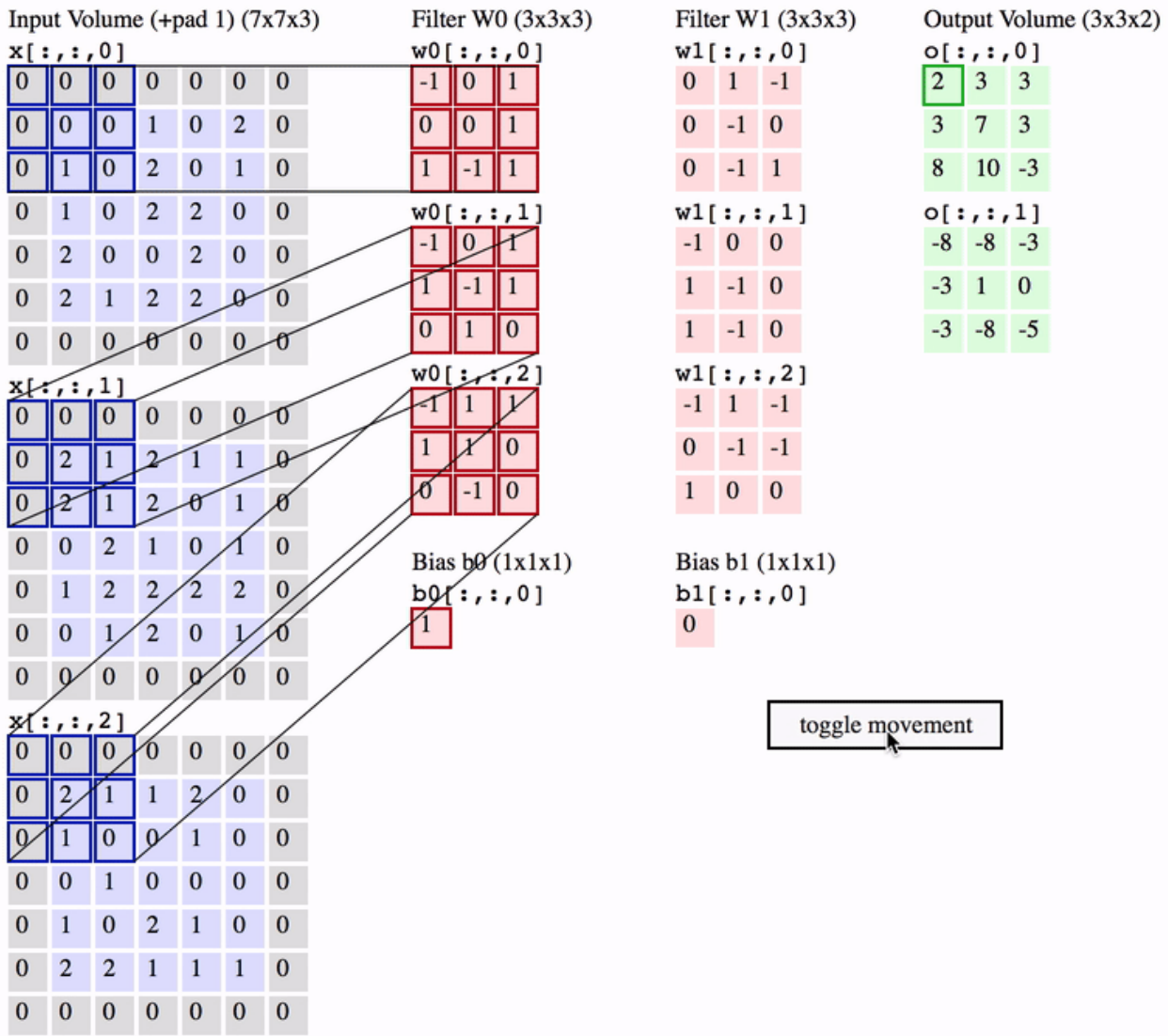
Convolution filter: second example



High value when filter mach image patch

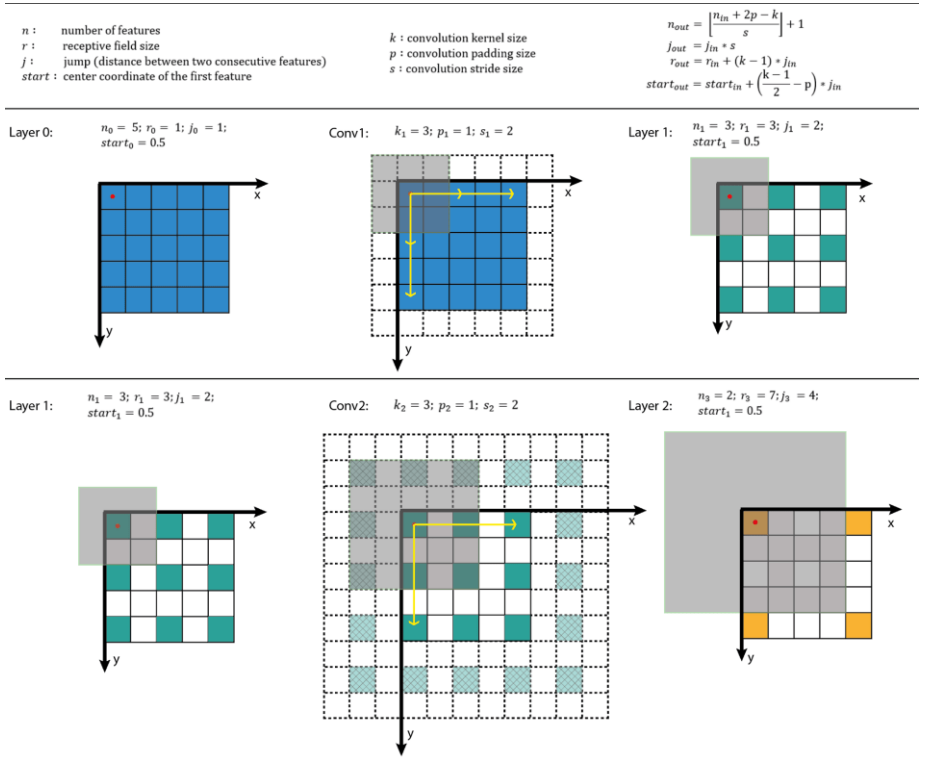
Low value when filter does not mach image patch

Convolutional Neural Networks



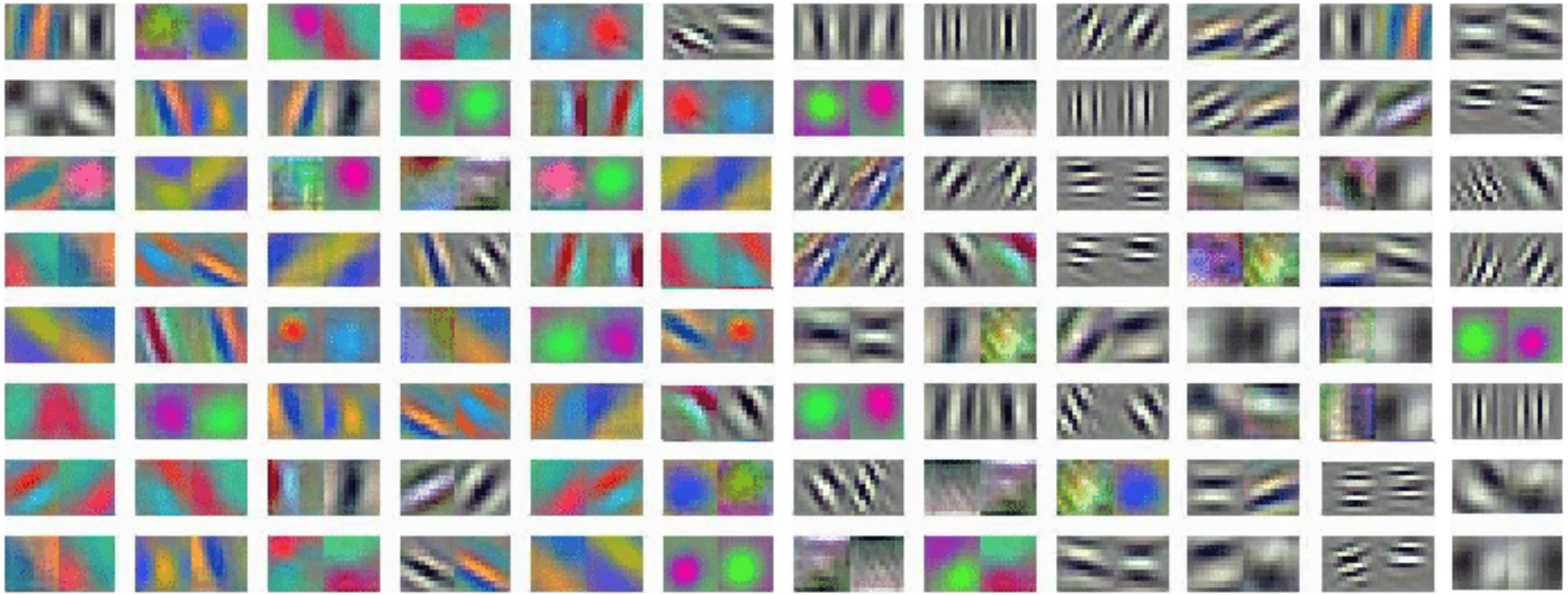
CONVOLUTION HYPERPARAMETERS

DEPTH STRIDE ZERO-PADDING

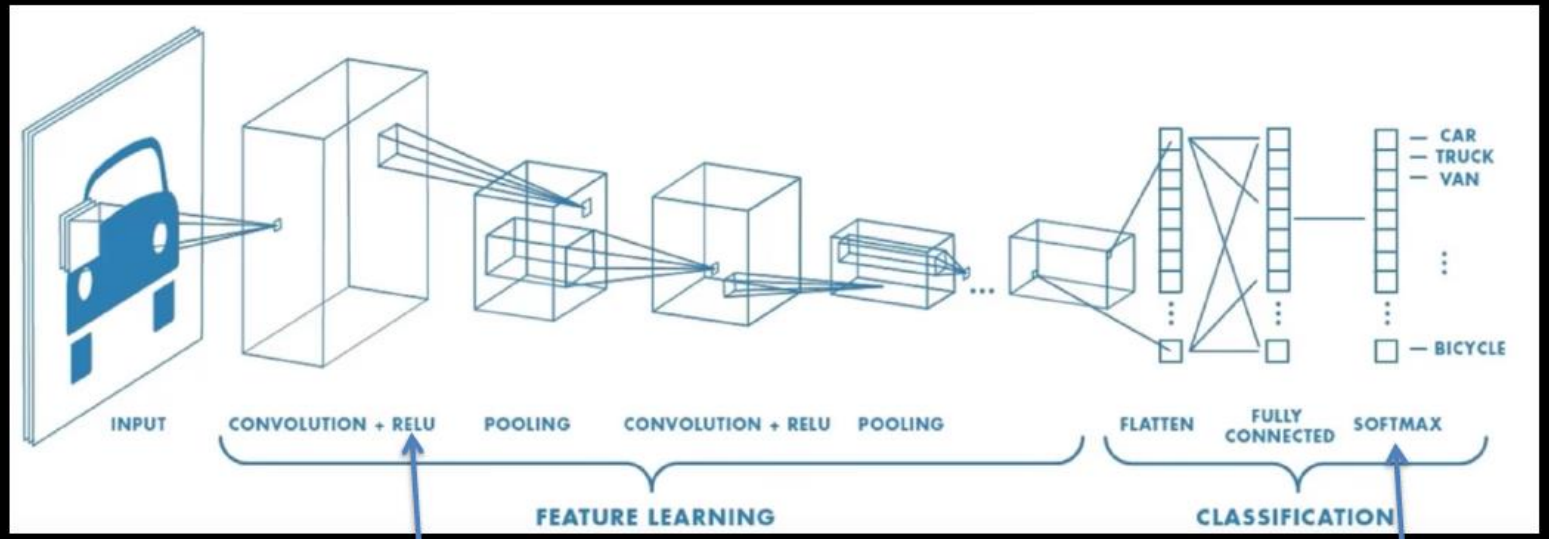


Convolutional Neural Networks

ALEXNET CONV1 LEARNED FILTERS



Convolutional Neural Networks



Activation function

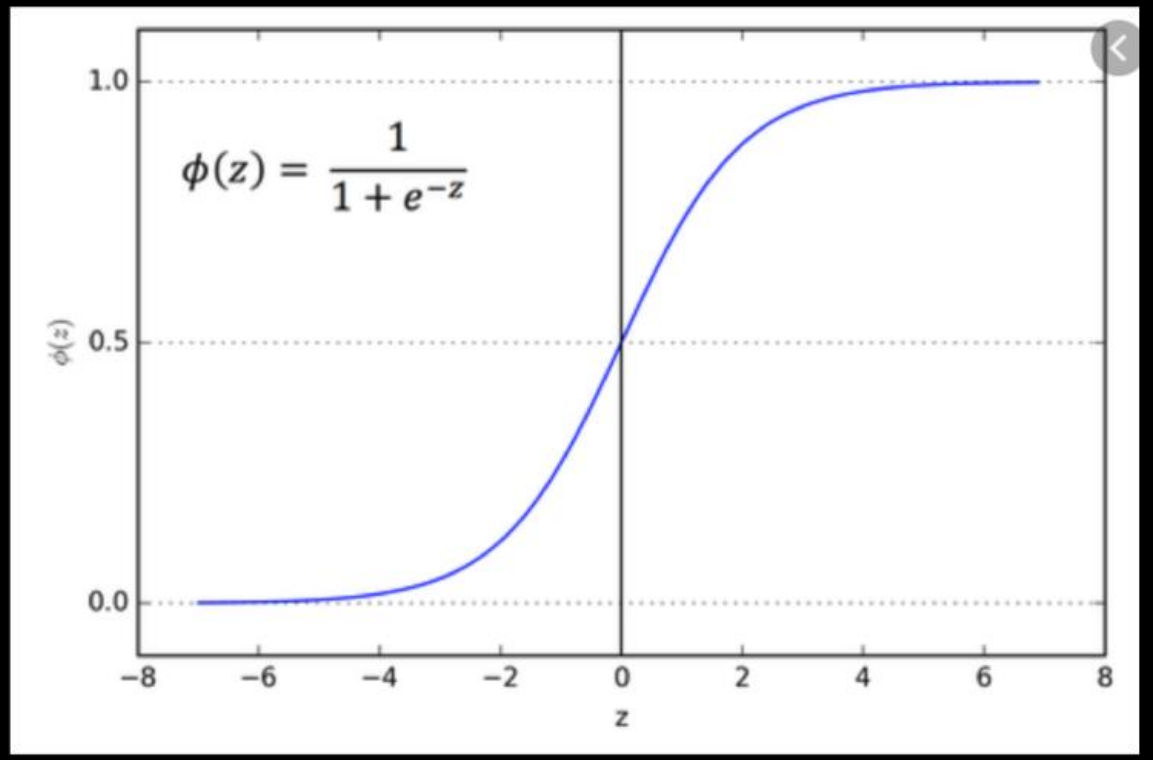
RELU
SIGMOID
TANH

Activation function

SOFTMAX

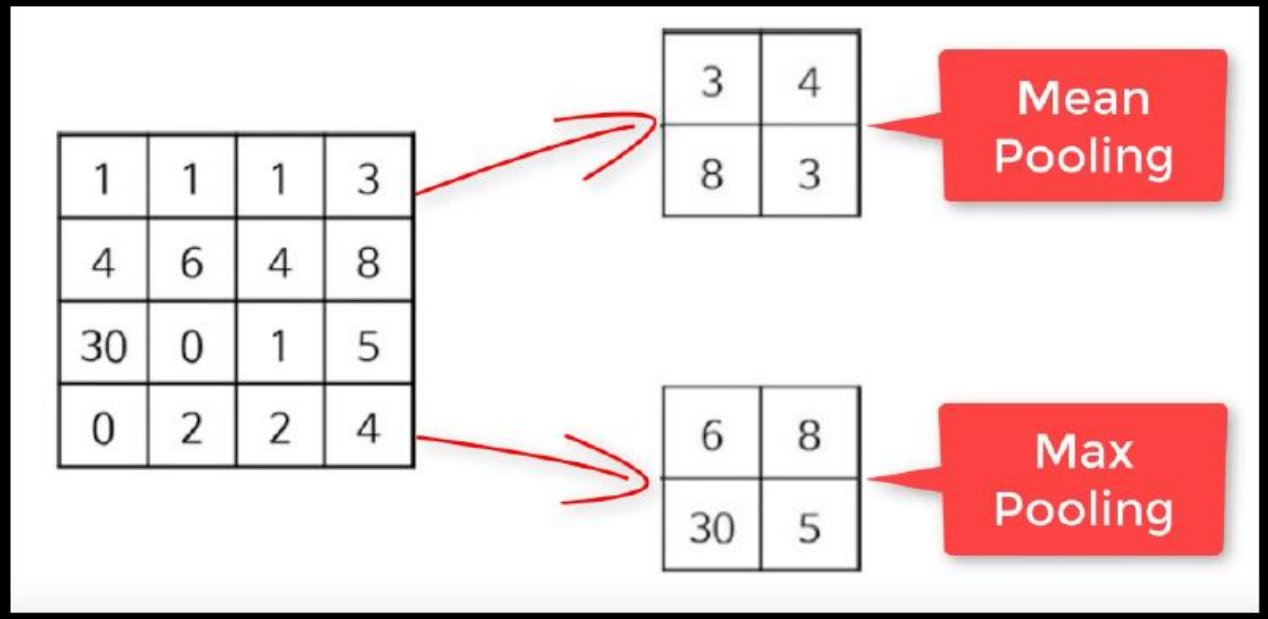
Convolutional Neural Networks

SOFTMAX



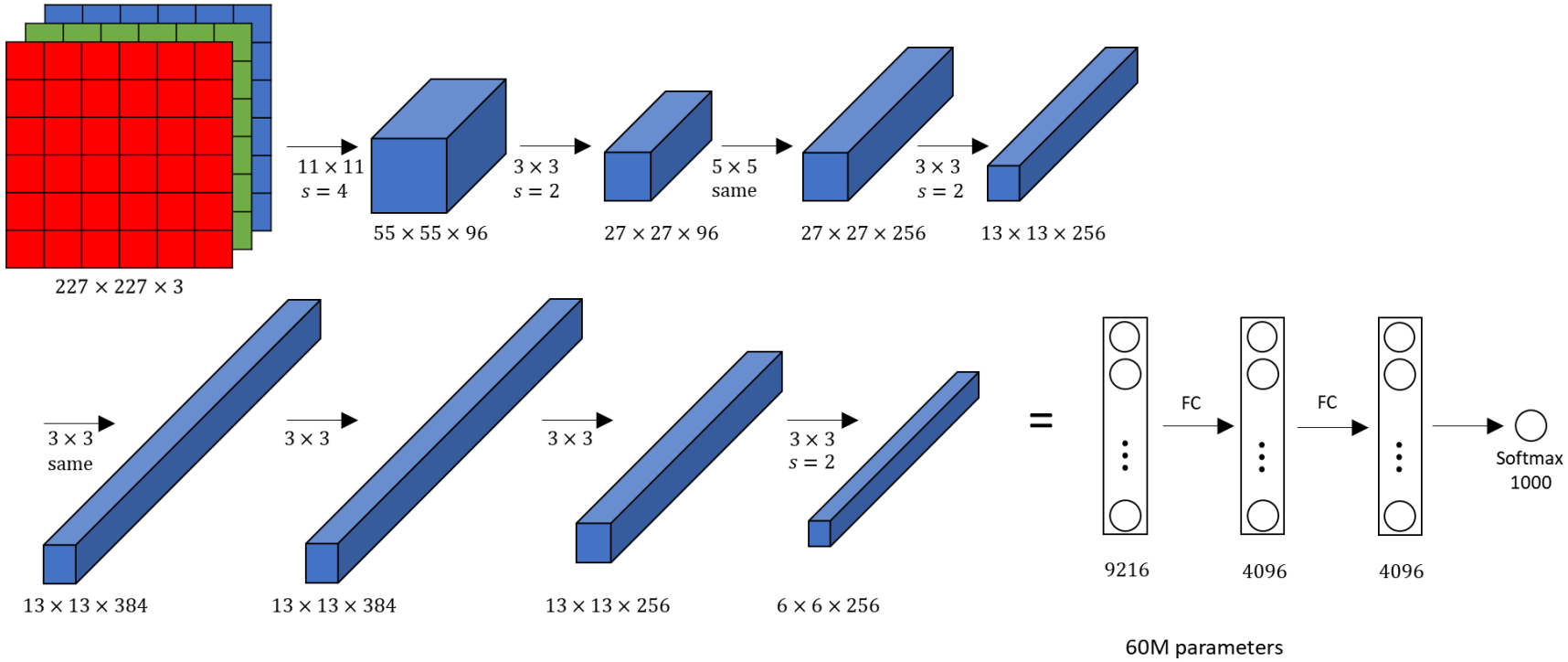
It converts the output of the last layer into a probability distribution

Pooling layer



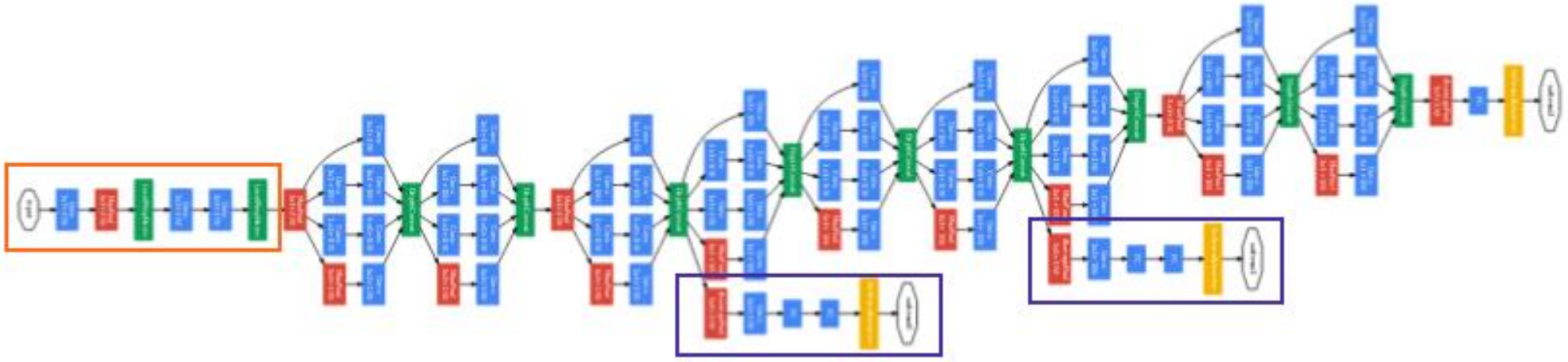
Convolutional Neural Networks | From LeNet to . . .

AlexNet [winner of the ImageNet ILSVRS challenge in 2012; (17)] is composed of both stacked and connected layers and includes five convolutional layers followed by three fully-connected layers, with max-pooling layers in between. A rectified linear unit nonlinearity is applied to each convolutional layer along with a fully-connected layer to enable faster training.



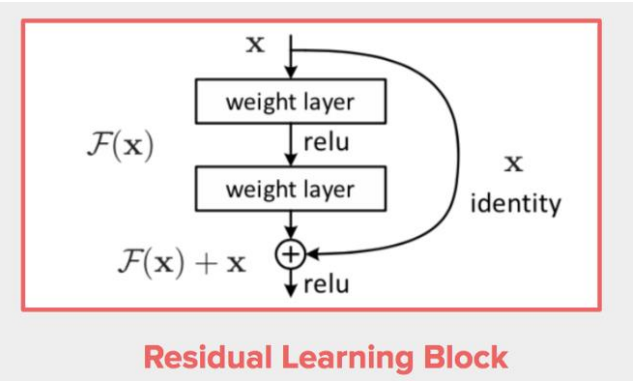
Convolutional Neural Networks | From LeNet to . . .

GoogleNet [winner of the ImageNet ILSVRS challenge in 2014; (41)] is the deep-learning algorithm whose design introduced the so-called Inception module, a subnetwork consisting of parallel convolutional filters whose outputs are concatenated. Inception greatly reduces the number of required parameters. GoogleNet is composed by 22 layers that require training (for a total of 27 layers when including the pooling layers).

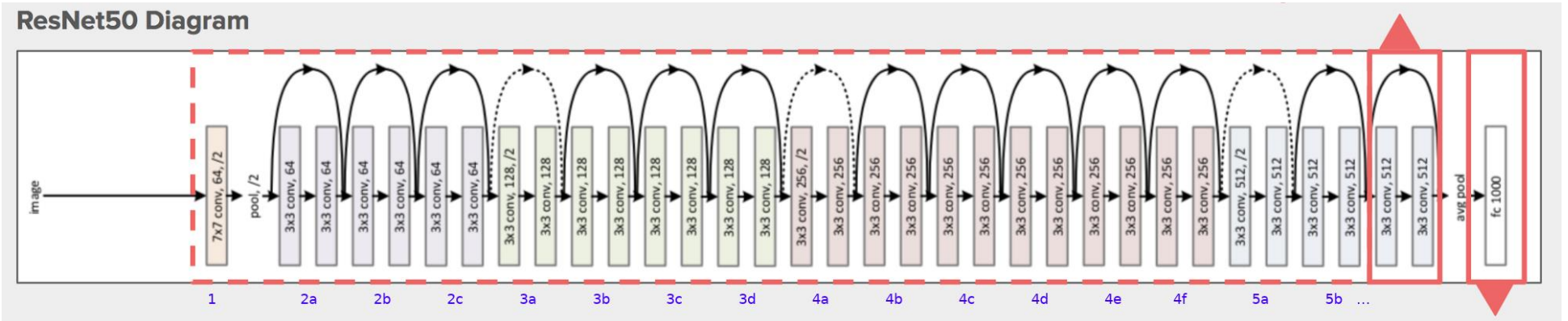


Convolutional Neural Networks | From LeNet to . . .

ResNet [winner of ILSVRC 2015; (42)], an architecture that is approximately twenty times deeper than AlexNet; its main novelty is the introduction of residual layers, a kind of network-in-network architecture that forms building blocks to construct the network. ResNet uses special skip connections and batch normalization, and the fully-connected layers at the end of the network are substituted by global average pooling. Instead of learning unreferenced functions, ResNet explicitly reformulates layers as learning residual functions with reference to the input layer, which makes the model smaller in size and thus easier to optimize than other architectures.

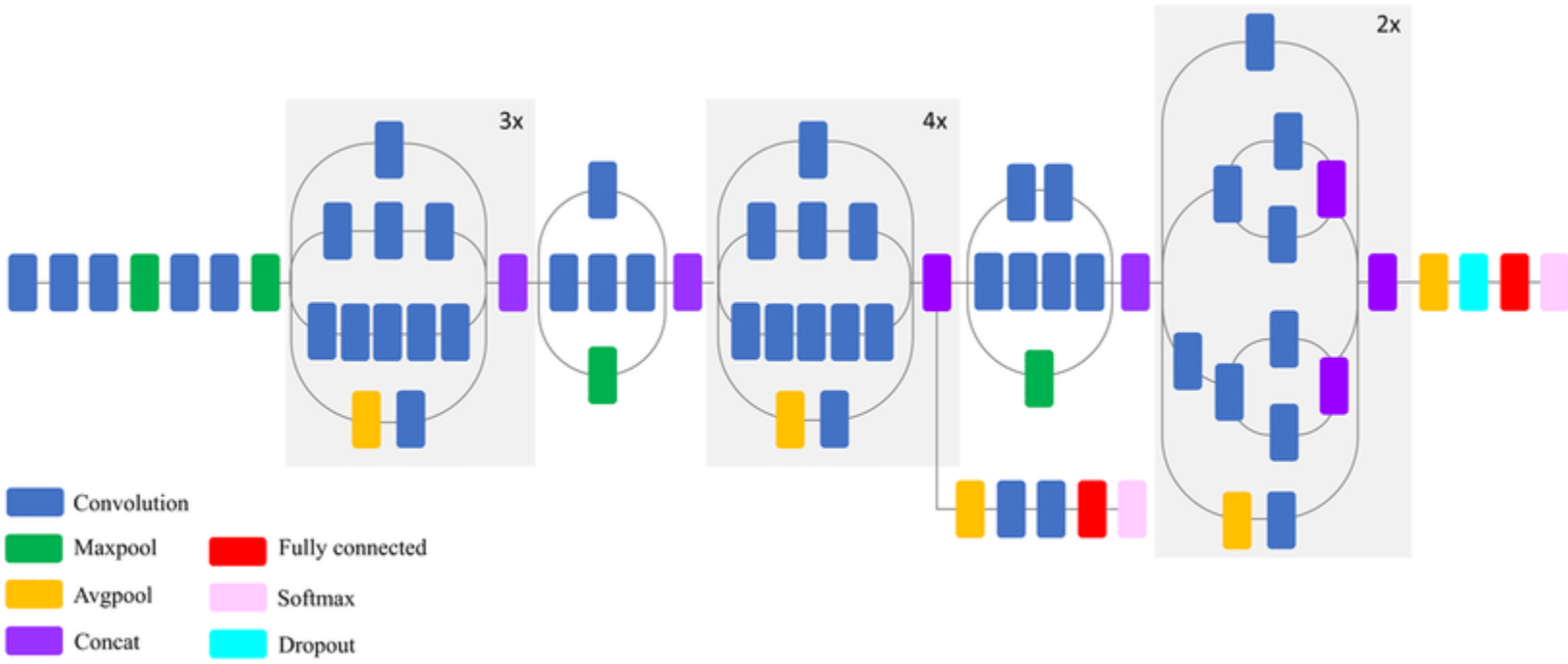


Residual Learning Block



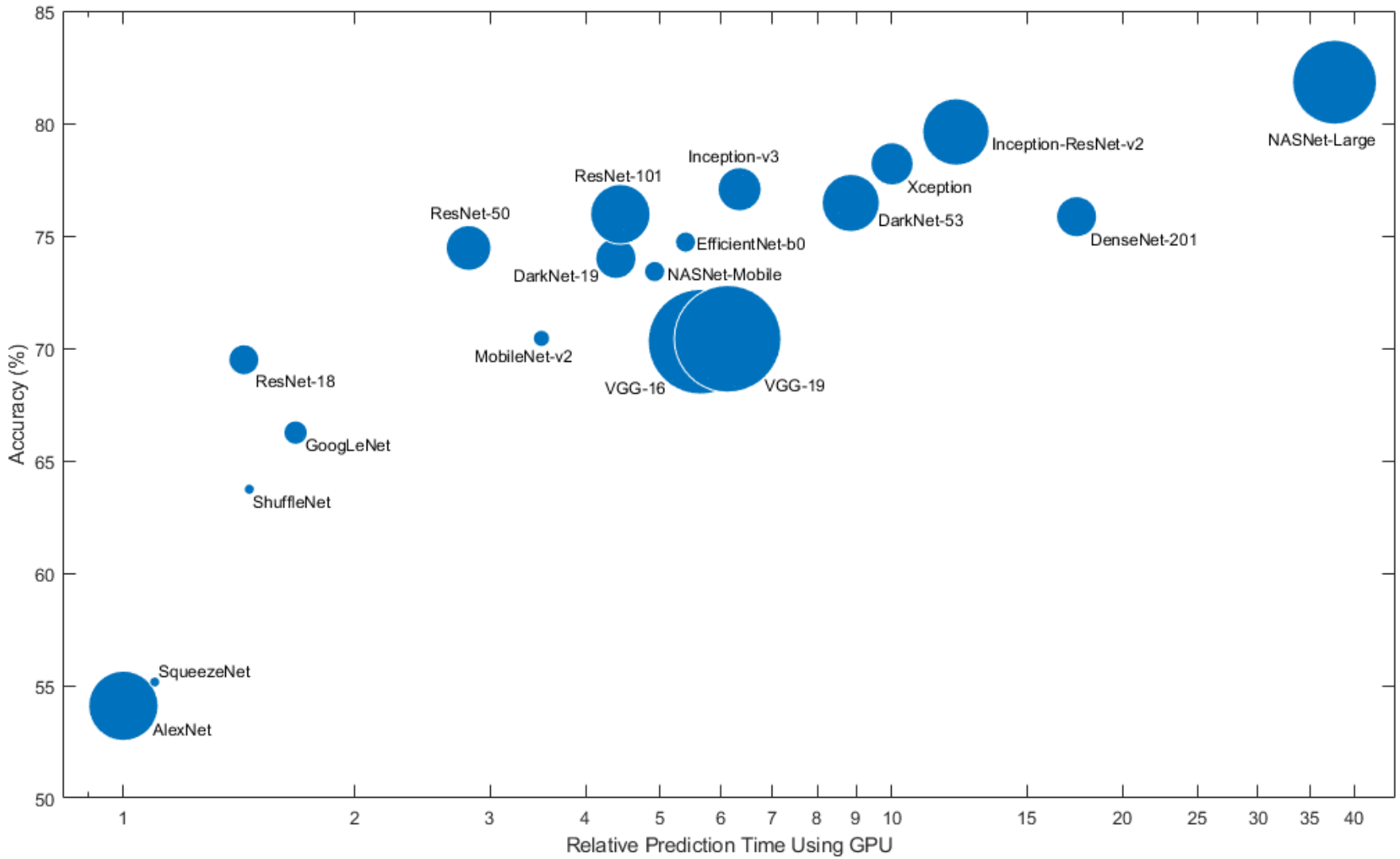
Convolutional Neural Networks | From LeNet to . . .

Inception-v3 (43), a deep architecture of 48 layers able to classify images into 1,000 object categories; the net was trained on more than a million images obtained from the ImageNet database (resulting in a rich feature representation for a wide range of images). Inception-v3 classified as the first runner up for the ImageNet ILSVRC challenge in 2015.



Convolutional Neural Networks | From LeNet to . . .

https://it.mathworks.com/help/deeplearning/ug/pretrained-convolutional-neural-networks.html



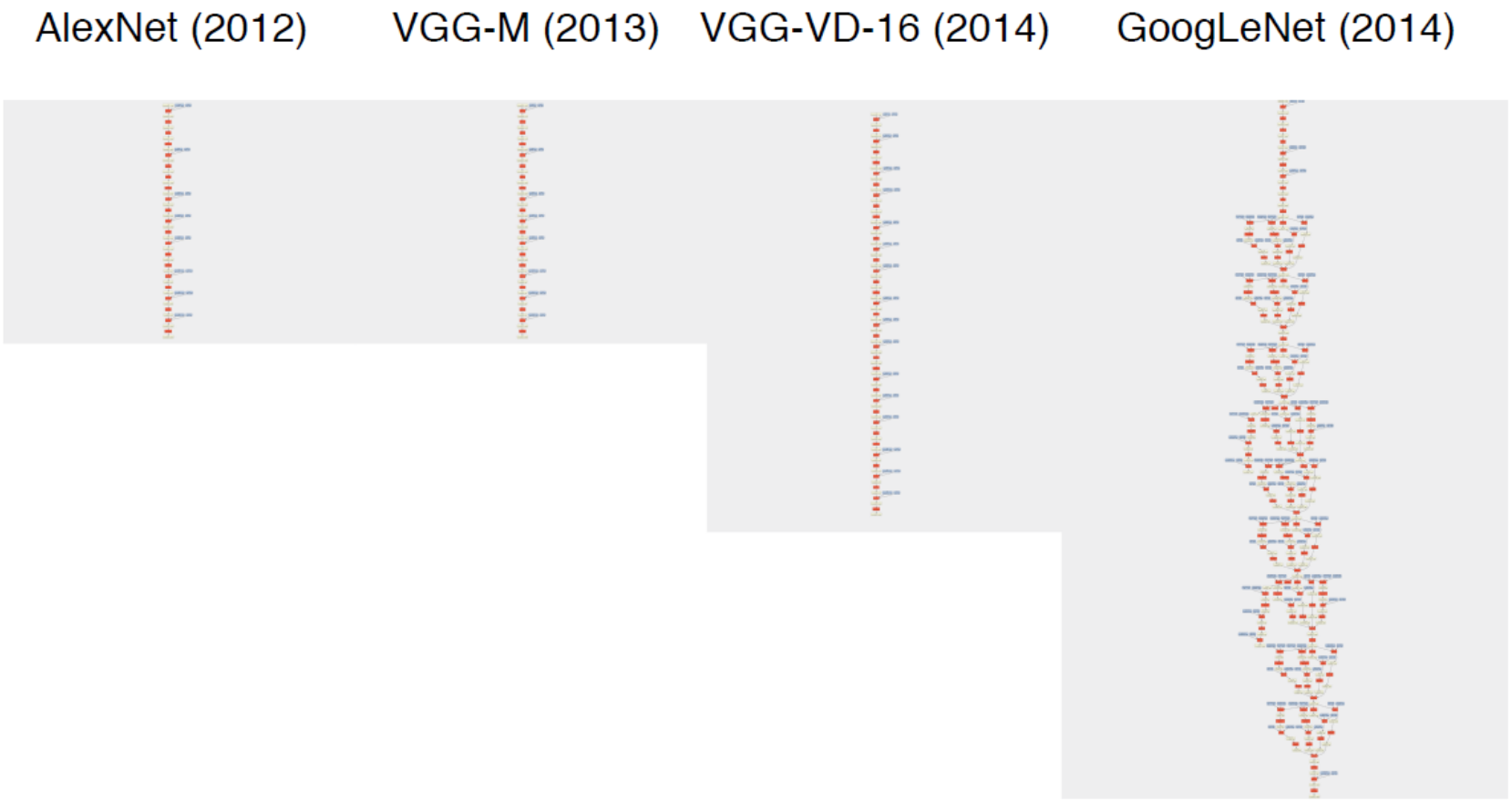
Convolutional Neural Networks | From LeNet to . . .

Network	Depth	Size	Parameters (Millions)	Image Input Size
squeezenet	18	5.2 MB	1.24	227-by-227
googlenet	22	27 MB	7.0	224-by-224
inceptionv3	48	89 MB	23.9	299-by-299
densenet201	201	77 MB	20.0	224-by-224
mobilenetv2	53	13 MB	3.5	224-by-224
resnet18	18	44 MB	11.7	224-by-224
resnet50	50	96 MB	25.6	224-by-224
resnet101	101	167 MB	44.6	224-by-224
xception	71	85 MB	22.9	299-by-299
inceptionresnetv2	164	209 MB	55.9	299-by-299
shufflenet	50	5.4 MB	1.4	224-by-224
nasnetmobile	*	20 MB	5.3	224-by-224
nasnetlarge	*	332 MB	88.9	331-by-331
darknet19	19	78 MB	20.8	256-by-256
darknet53	53	155 MB	41.6	256-by-256
efficientnetb0	82	20 MB	5.3	224-by-224
alexnet	8	227 MB	61.0	227-by-227
vgg16	16	515 MB	138	224-by-224
vgg19	19	535 MB	144	224-by-224

Convolutional Neural Networks | From LeNet to . . .

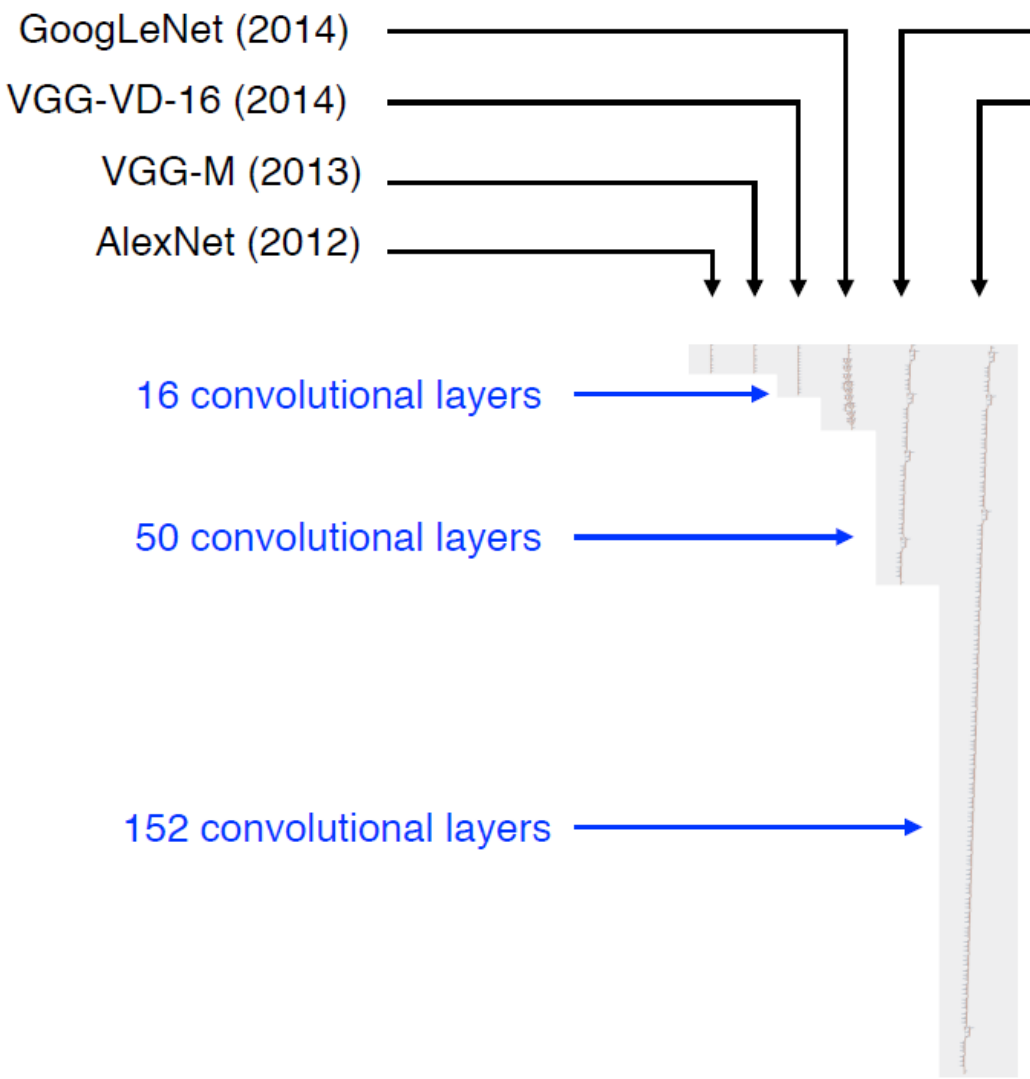
FROM A. VEDALDI

Convolutional Neural Networks | From LeNet to . . .



FROM A. VEDALDI

Convolutional Neural Networks | From LeNet to . . .



Krizhevsky, I. Sutskever, and G. E. Hinton. *ImageNet classification with deep convolutional neural networks*. In Proc. NIPS, 2012.

C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. *Going deeper with convolutions*. In Proc. CVPR, 2015.

K. Simonyan and A. Zisserman. *Very deep convolutional networks for large-scale image recognition*. In Proc. ICLR, 2015.

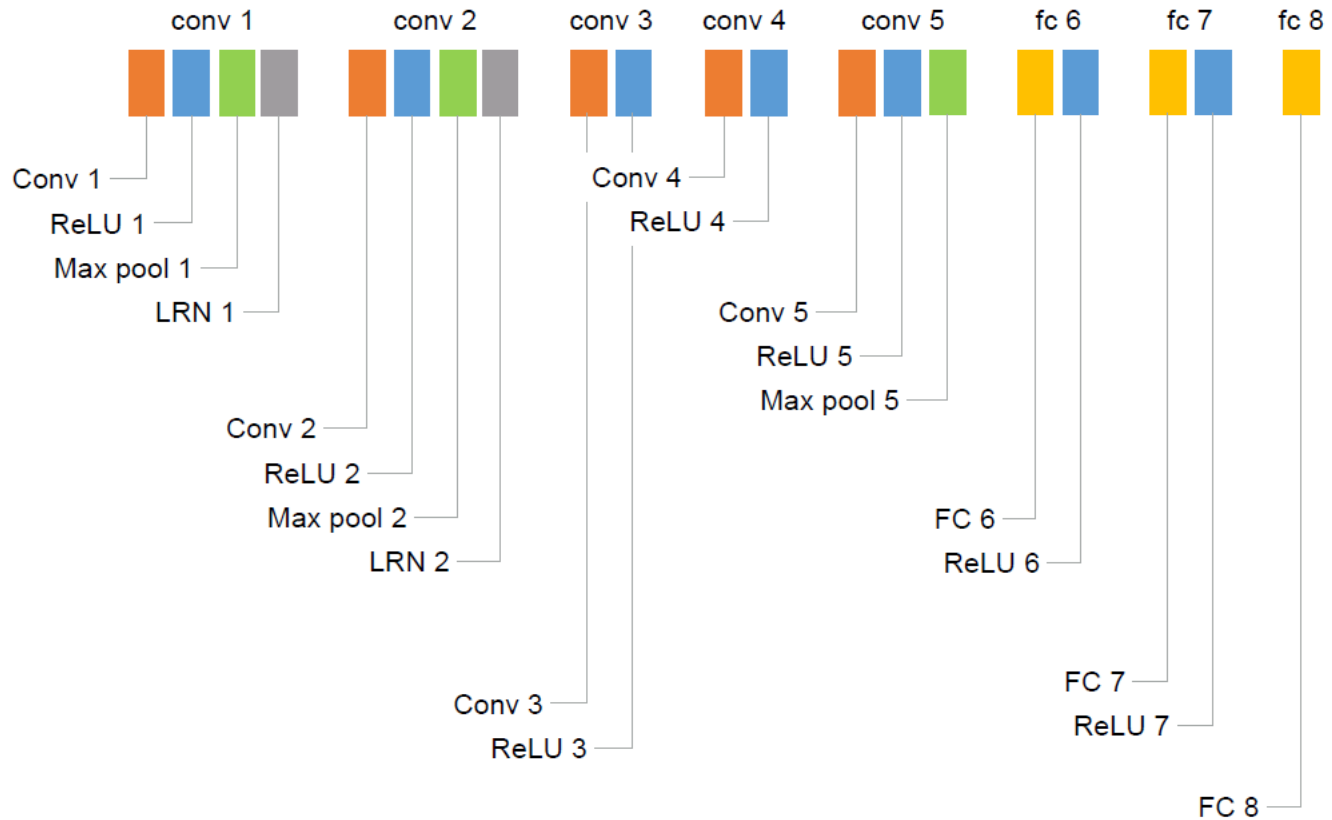
K. He, X. Zhang, S. Ren, and J. Sun. *Deep residual learning for image recognition*. In Proc. CVPR, 2016.

FROM A. VEDALDI

A CONSIDERATION ON EXPLAINABILITY AND (OVER)FITTING

UNDERFITTING, OVERFITTING AND BEST FITTING

Inversion



AlexNet
[Krizhevsky et al. 2012]

UNDERFITTING, OVERFITTING AND BEST FITTING

FROM A. VEDALDI

Inversion

11

Original Image

UNDERFITTING, OVERFITTING AND BEST FITTING

FROM A. VEDALDI

Inversion

12

Original Image

UNDERFITTING, OVERFITTING AND BEST FITTING

FROM A. VEDALDI

Inversion

13

Original Image

UNDERFITTING, OVERFITTING AND BEST FITTING

FROM A. VEDALDI

Inversion

Original Image

UNDERFITTING, OVERFITTING AND BEST FITTING

FROM A. VEDALDI

Inversion

16

Original Image

UNDERFITTING, OVERFITTING AND BEST FITTING

Inversion

17

Original Image

FROM A. VEDALDI

UNDERFITTING, OVERFITTING AND BEST FITTING

Inversion

18

Original Image

FROM A. VEDALDI

UNDERFITTING, OVERFITTING AND BEST FITTING

FROM A. VEDALDI

Inversion

19

Original Image

UNDERFITTING, OVERFITTING AND BEST FITTING

Inversion

Original Image

FROM A. VEDALDI

UNDERFITTING, OVERFITTING AND BEST FITTING

Inversion

Original Image

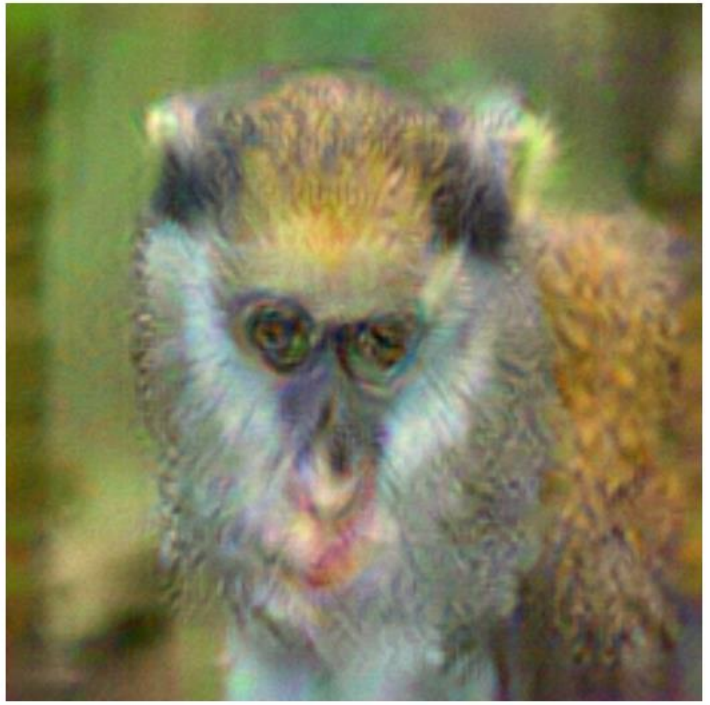
FROM A. VEDALDI

UNDERFITTING, OVERFITTING AND BEST FITTING

FROM A. VEDALDI

Inversion

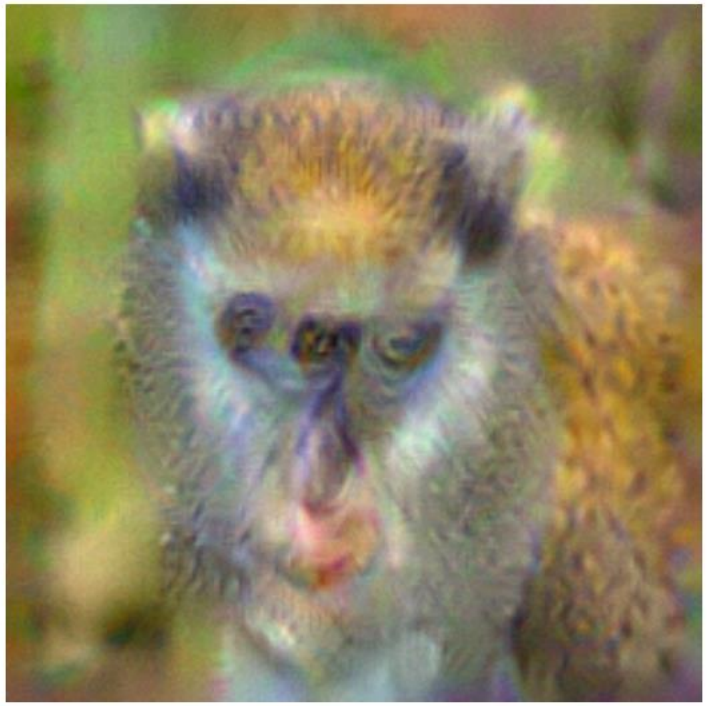
Original Image



UNDERFITTING, OVERFITTING AND BEST FITTING

Inversion

Original Image



FROM A. VEDALDI

UNDERFITTING, OVERFITTING AND BEST FITTING

Inversion

Original Image

FROM A. VEDALDI

UNDERFITTING, OVERFITTING AND BEST FITTING

Inversion

Original Image

FROM A. VEDALDI

UNDERFITTING, OVERFITTING AND BEST FITTING

Inversion

Original Image

FROM A. VEDALDI

UNDERFITTING, OVERFITTING AND BEST FITTING

Inversion

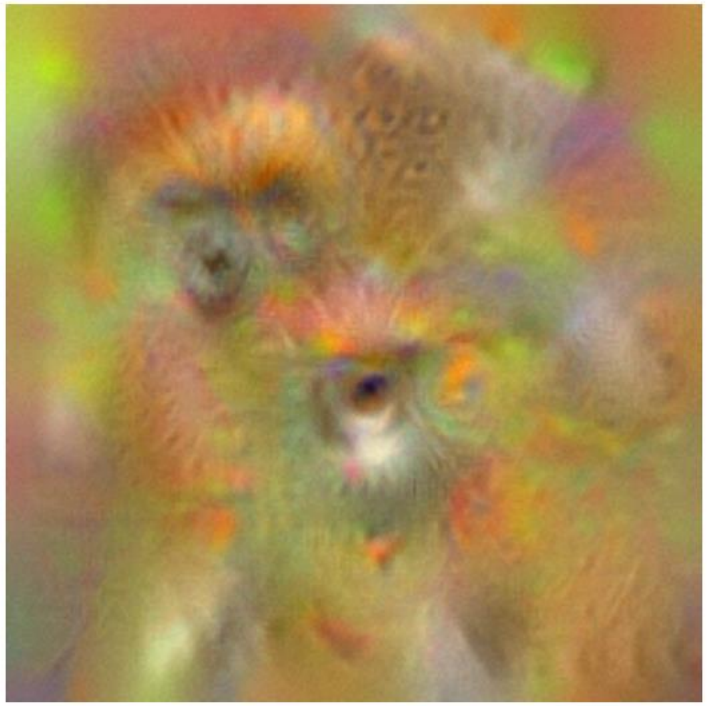
Original Image

FROM A. VEDALDI

UNDERFITTING, OVERFITTING AND BEST FITTING

Inversion

Original Image



FROM A. VEDALDI

UNDERFITTING, OVERFITTING AND BEST FITTING

Inversion

29

Original Image

FROM A. VEDALDI

UNDERFITTING, OVERFITTING AND BEST FITTING

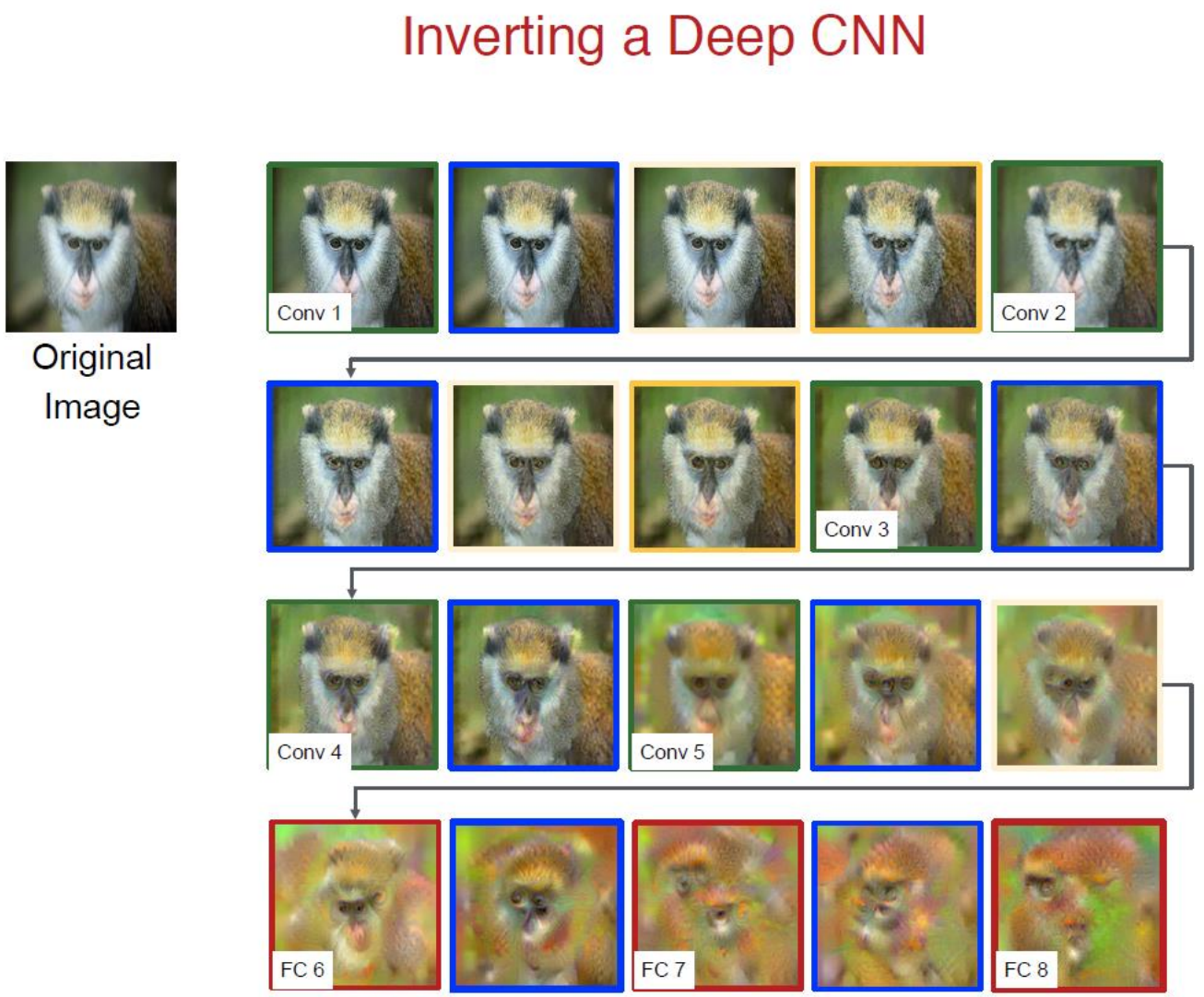
Inversion

Original Image

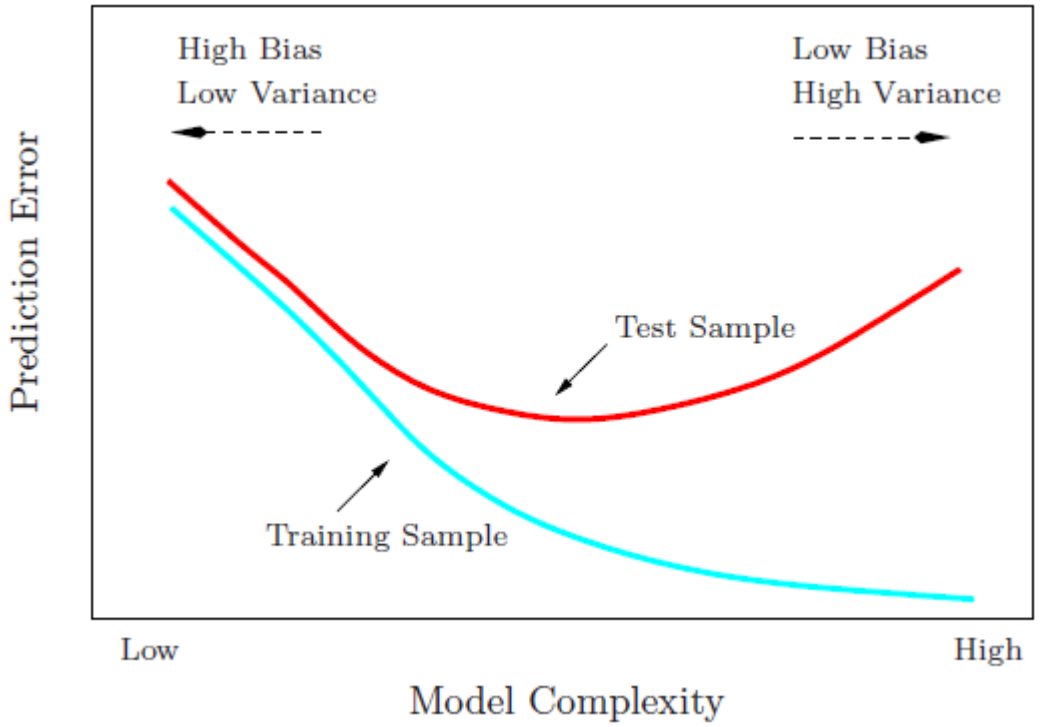
FROM A. VEDALDI

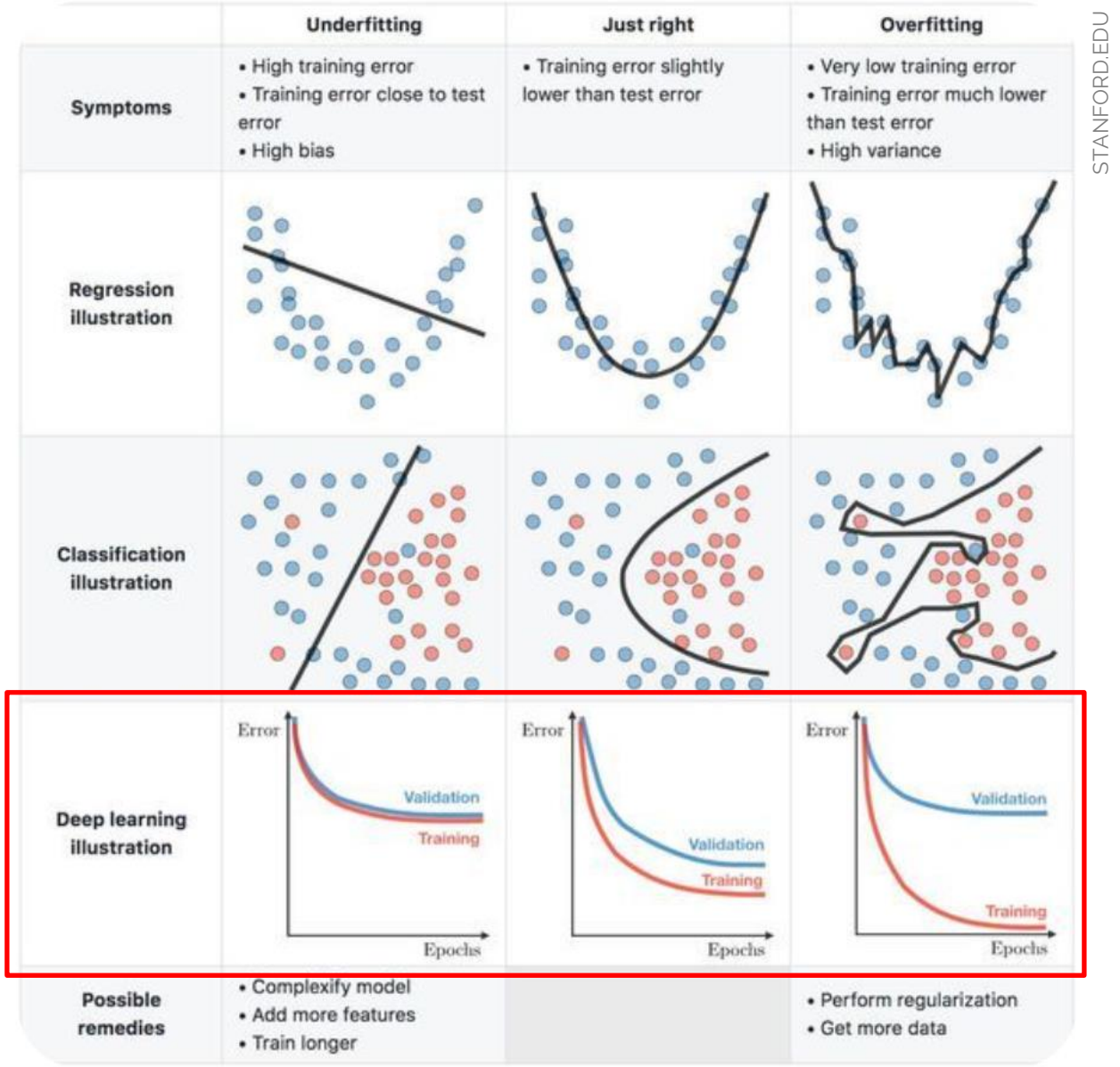
UNDERFITTING, OVERFITTING AND BEST FITTING

FROM A. VEDALDI



UNDERFITTING, OVERFITTING AND BEST FITTING

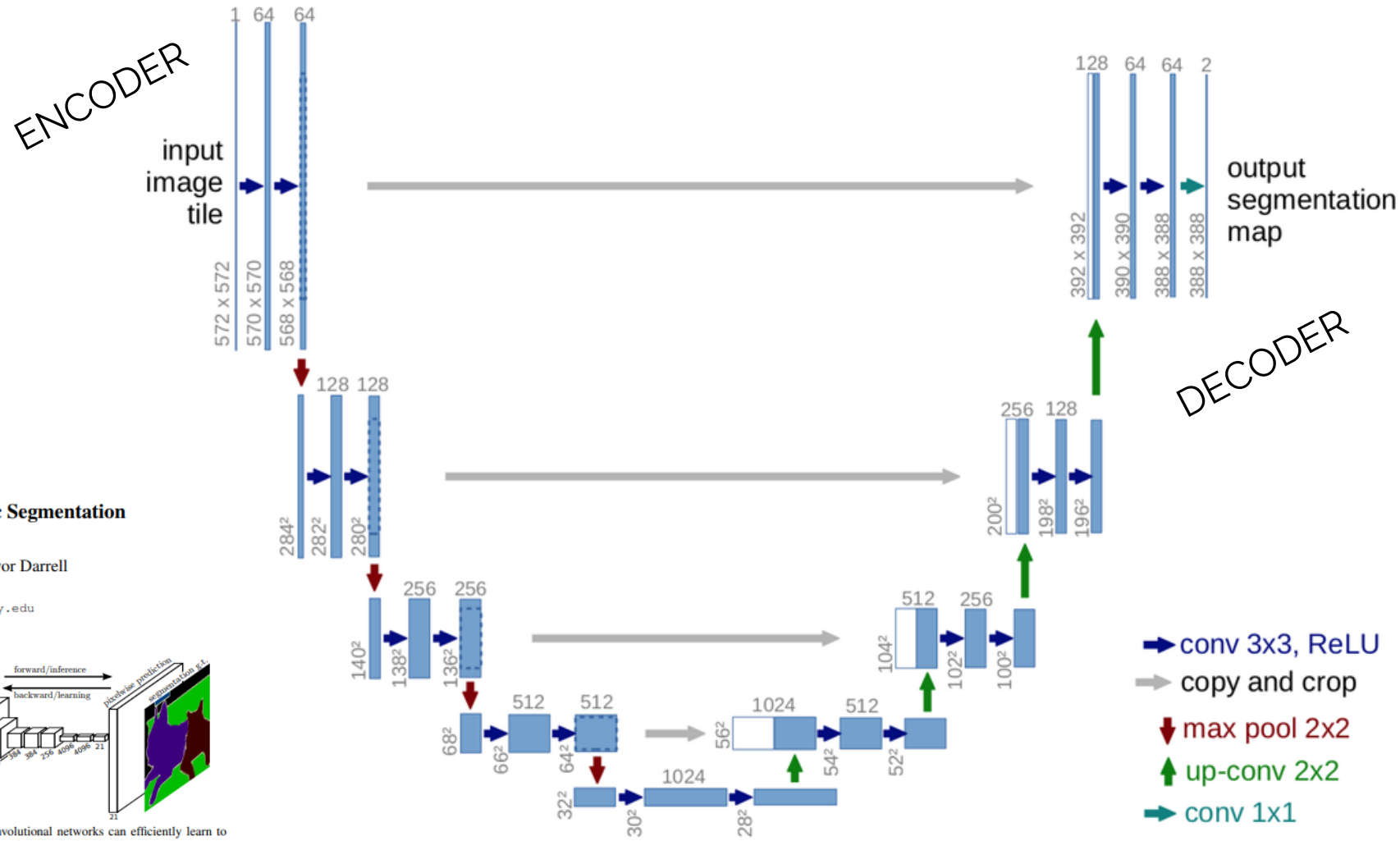




Convolutional Neural Networks | U-Net

U-Net

Suited for (medical) image segmentation



Fully Convolutional Networks for Semantic Segmentation

Jonathan Long* Evan Shelhamer* Trevor Darrell
UC Berkeley
{jonlong,shelhamer,trevor}@cs.berkeley.edu

Abstract

Convolutional networks are powerful visual models that yield hierarchies of features. We show that convolutional networks by themselves, trained end-to-end, pixels-to-pixels, exceed the state-of-the-art in semantic segmentation. Our key insight is to build "fully convolutional" networks that take input of arbitrary size and produce correspondingly-sized output with efficient inference and learning. We define and detail the space of fully convolutional networks, explain their application to spatially dense prediction tasks, and draw connections to prior models. We adapt contemporary classification networks (AlexNet [19],

Figure 1. Fully convolutional networks can efficiently learn to make dense predictions for per-pixel tasks like semantic segmentation.

christian.salvatore@iusspavia.it

https://christiansalvatore.github.io/machinelearning-iuss/