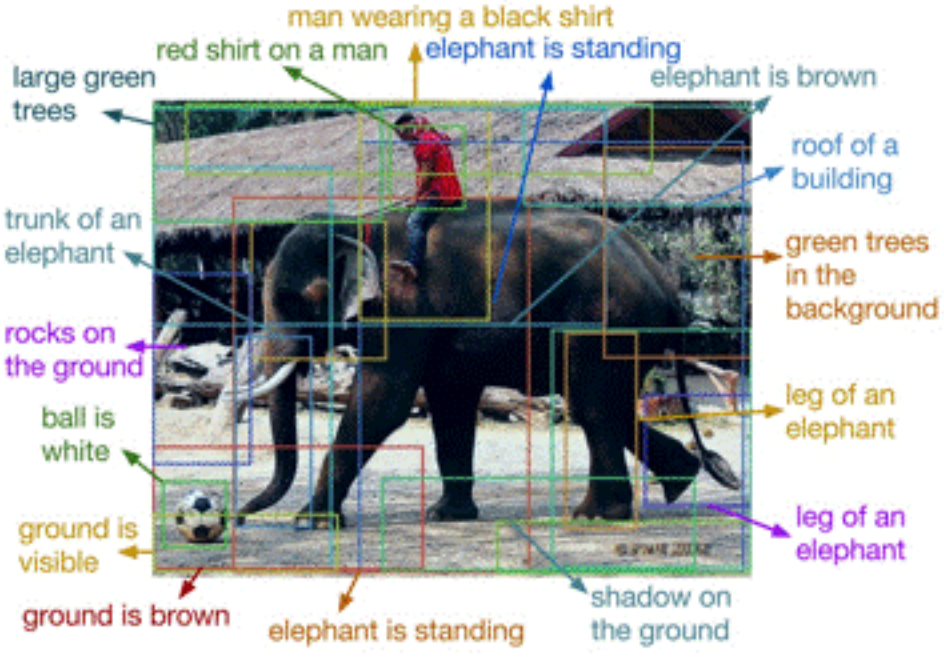
Introduction to Deep Learning

Thierry Pécot
Research Engineer
Biosit SFR UMS CNRS 3480 – Inserm 018
CZI Imaging Scientist

Deep Learning

Speech Recognition (2011)

Computer vision (2012)

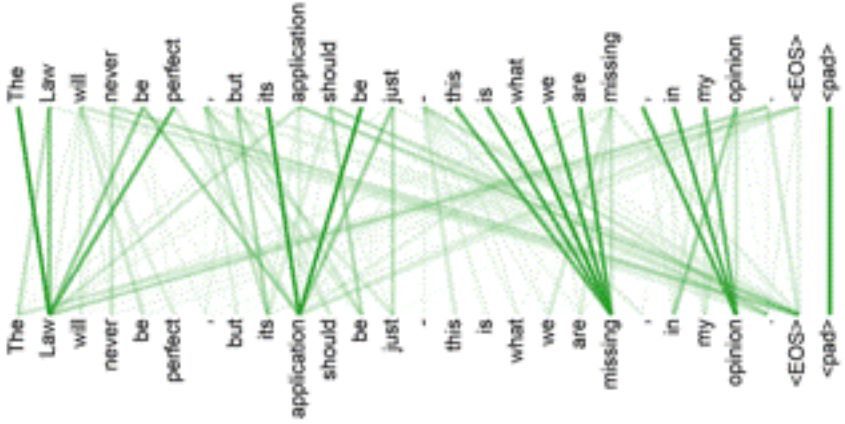


DenseCap, Johnson *et al.*, 2016

Go game (2016)

Mastering the game of Go with deep neural networks and tree search, Silver *et al.*, Nature, 2016

Natural Language Processing for translation and chatbots (transformers 2017)



Attention is all you need, Vaswani *et al.*, 2017

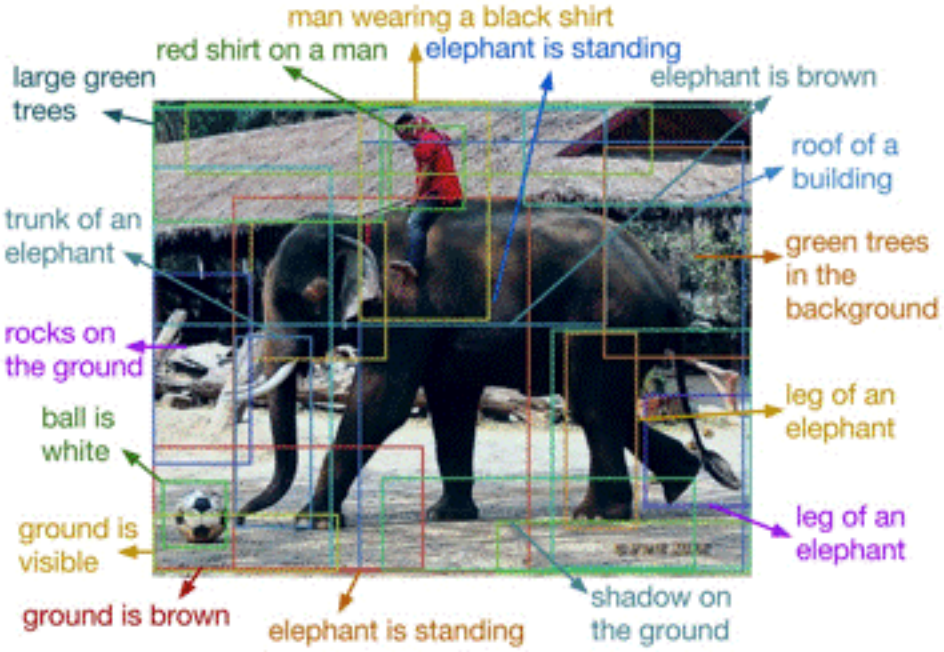
Tech industry

Art, music, journalism, ...

Deep Learning

Speech Recognition (2011)

Computer vision (2012)

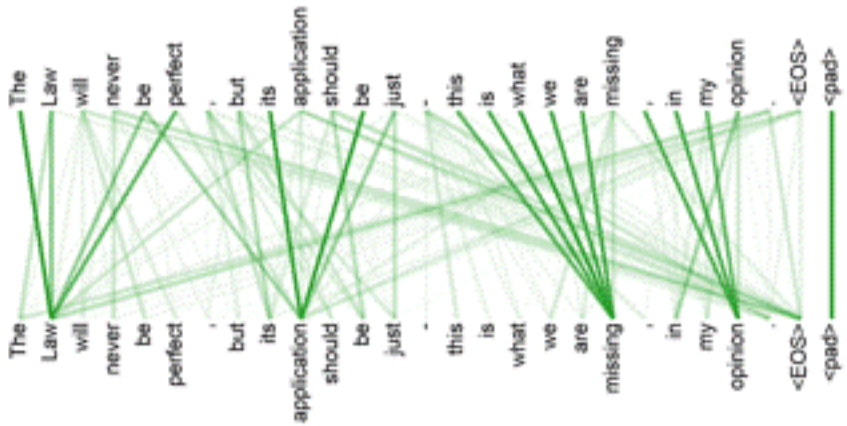


DenseCap, Johnson *et al.*, 2016

Go game (2016)

Mastering the game of Go with deep neural networks and tree search, Silver *et al.*, Nature, 2016

Natural Language Processing for translation and chatbots (transformers 2017)

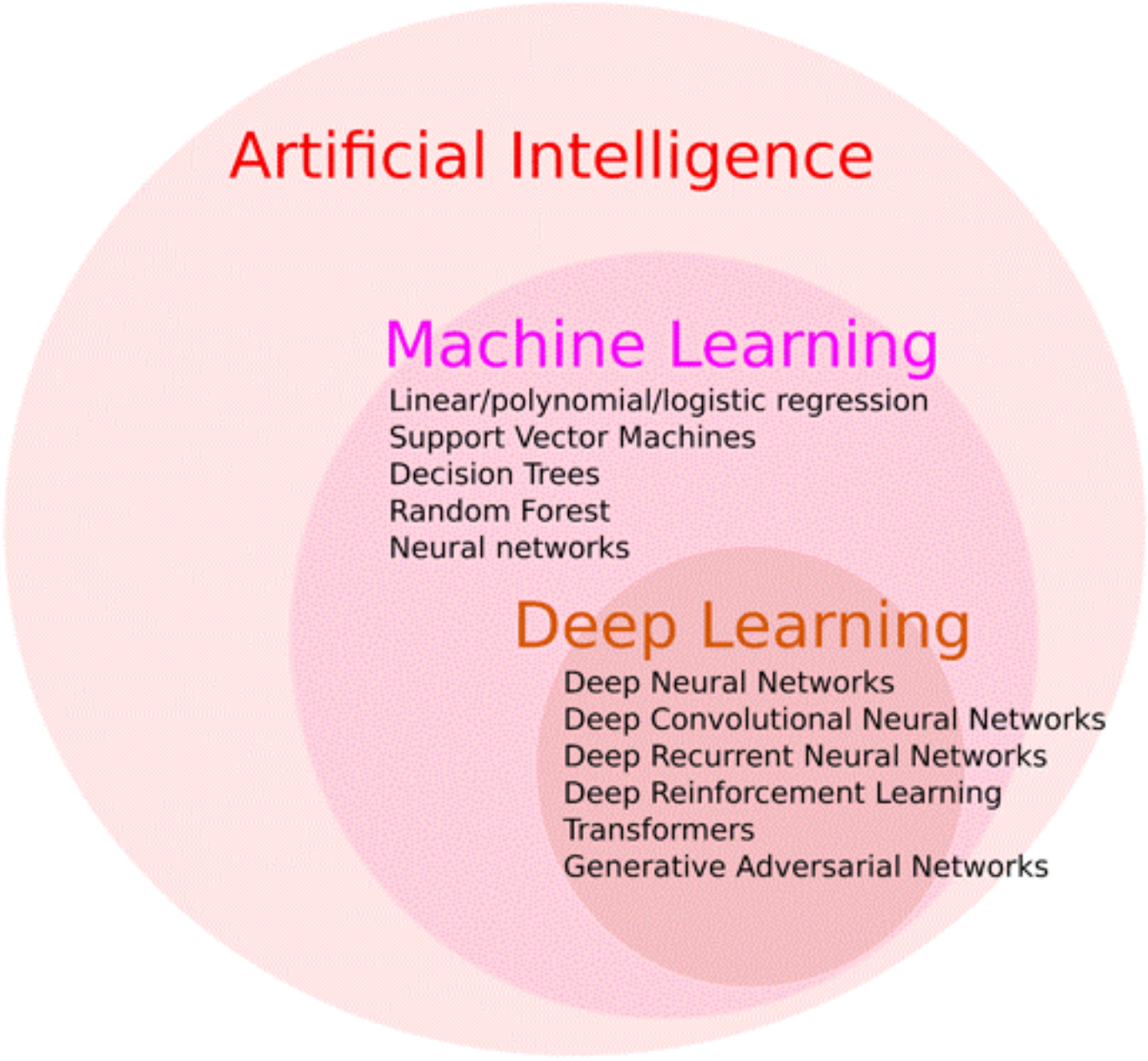


Attention is all you need, Vaswani *et al.*, 2017

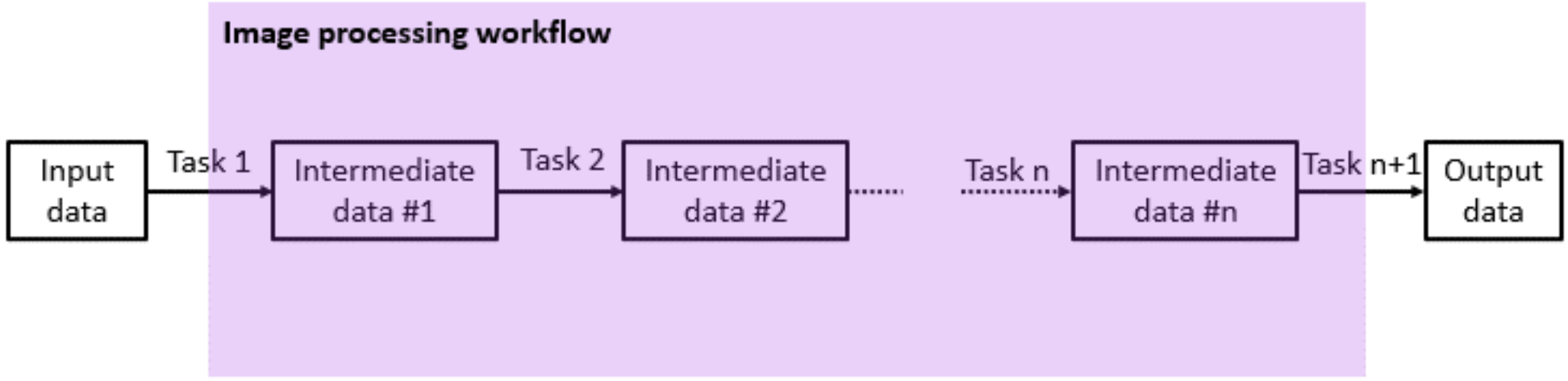
Tech industry

Art, music, journalism, ...

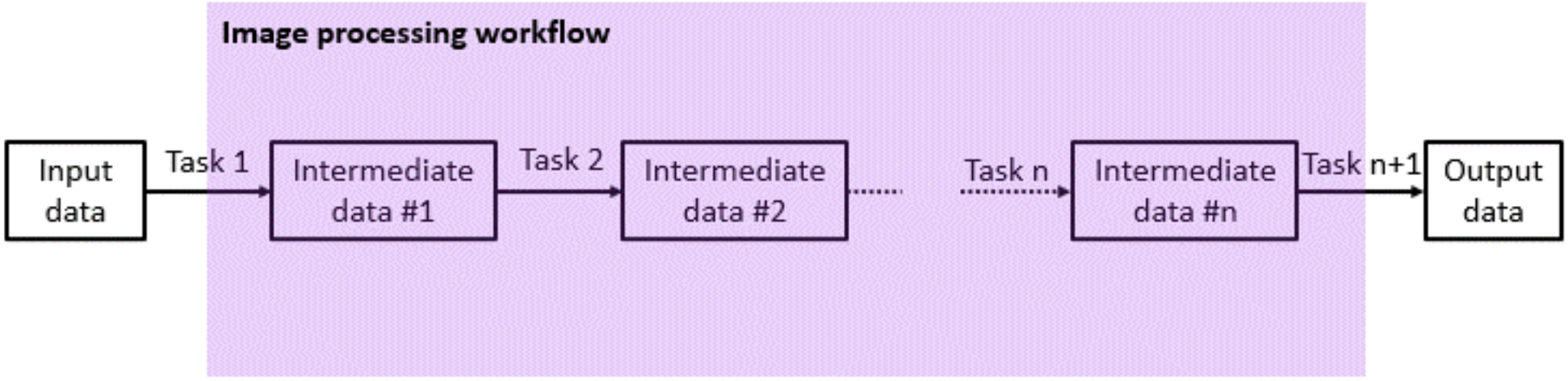
What is AI, more particularly deep learning?

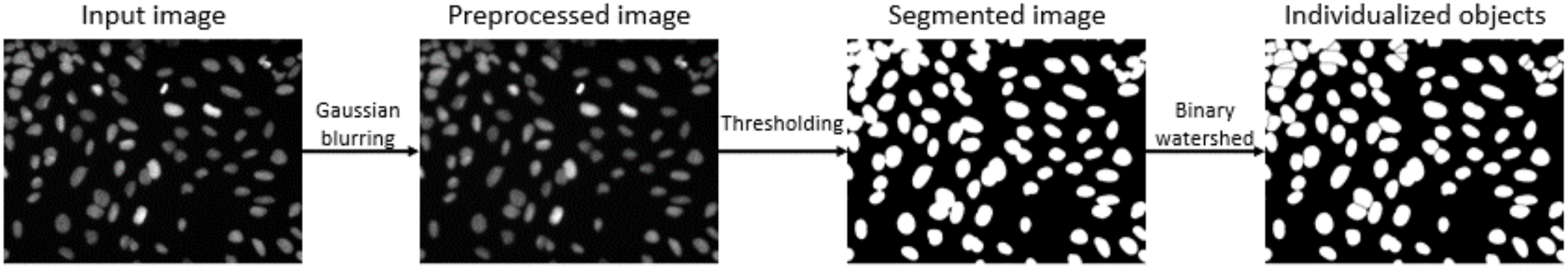

Artificial Intelligence
Machine Learning
Linear/polynomial/logistic regression
Support Vector Machines
Decision Trees
Random Forest
Neural networks
Deep Learning
Deep Neural Networks
Deep Convolutional Neural Networks
Deep Recurrent Neural Networks
Deep Reinforcement Learning
Transformers
Generative Adversarial Networks

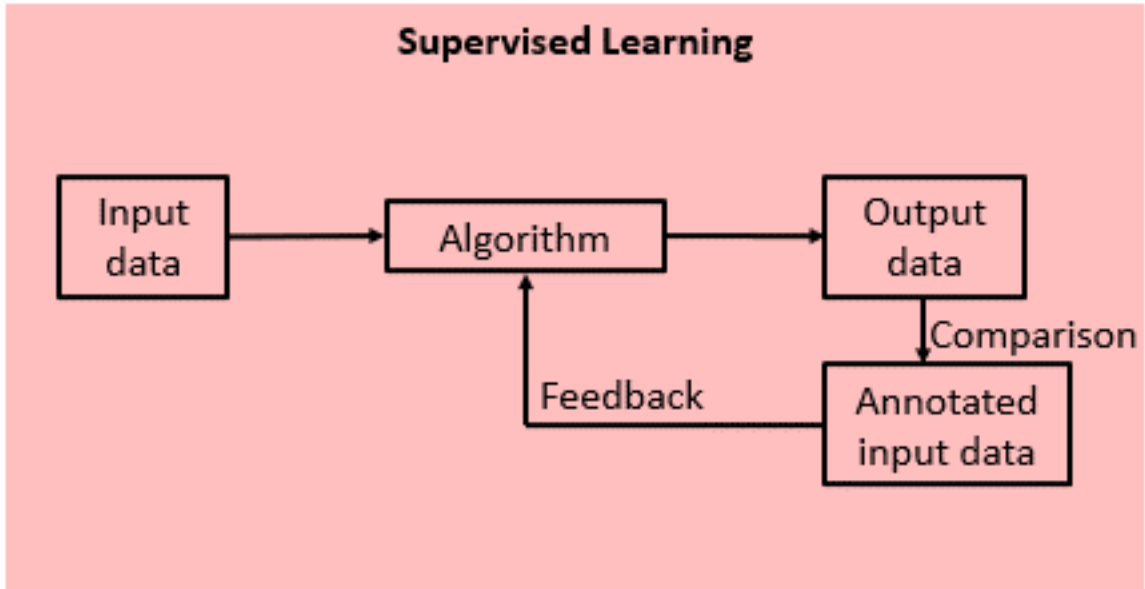
EXPLICIT PROGRAMMING



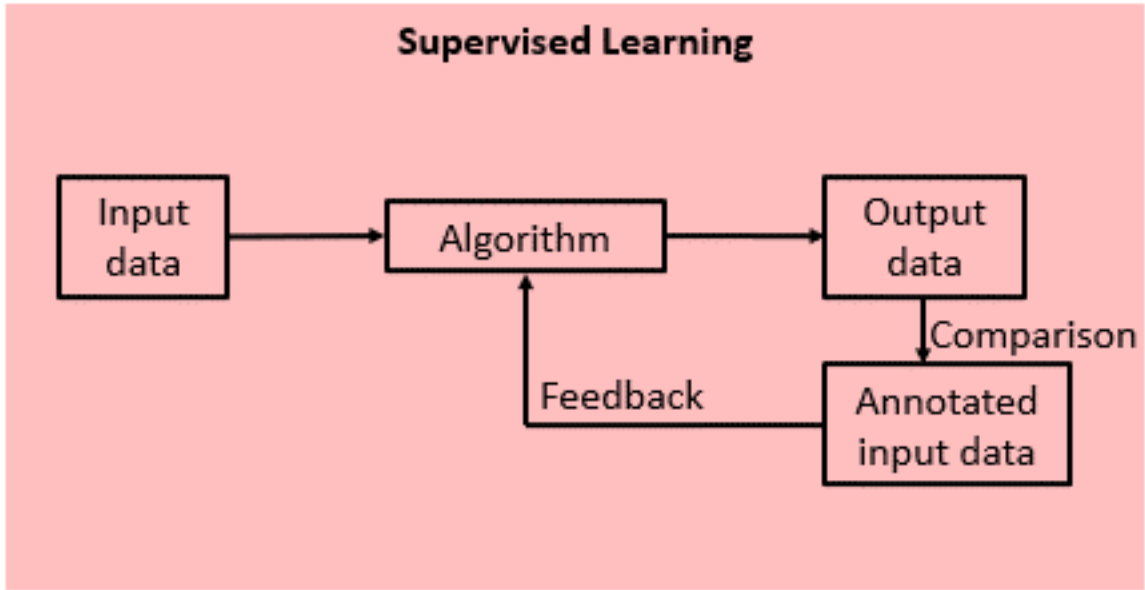
EXPLICIT PROGRAMMING

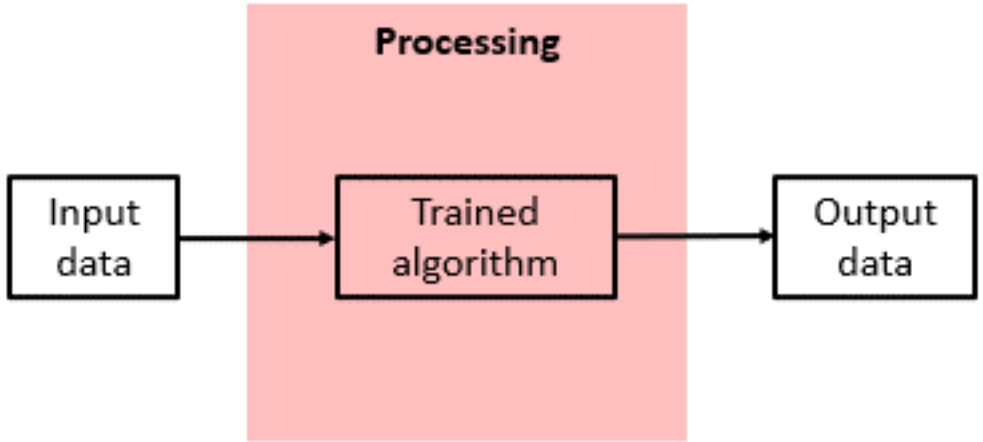




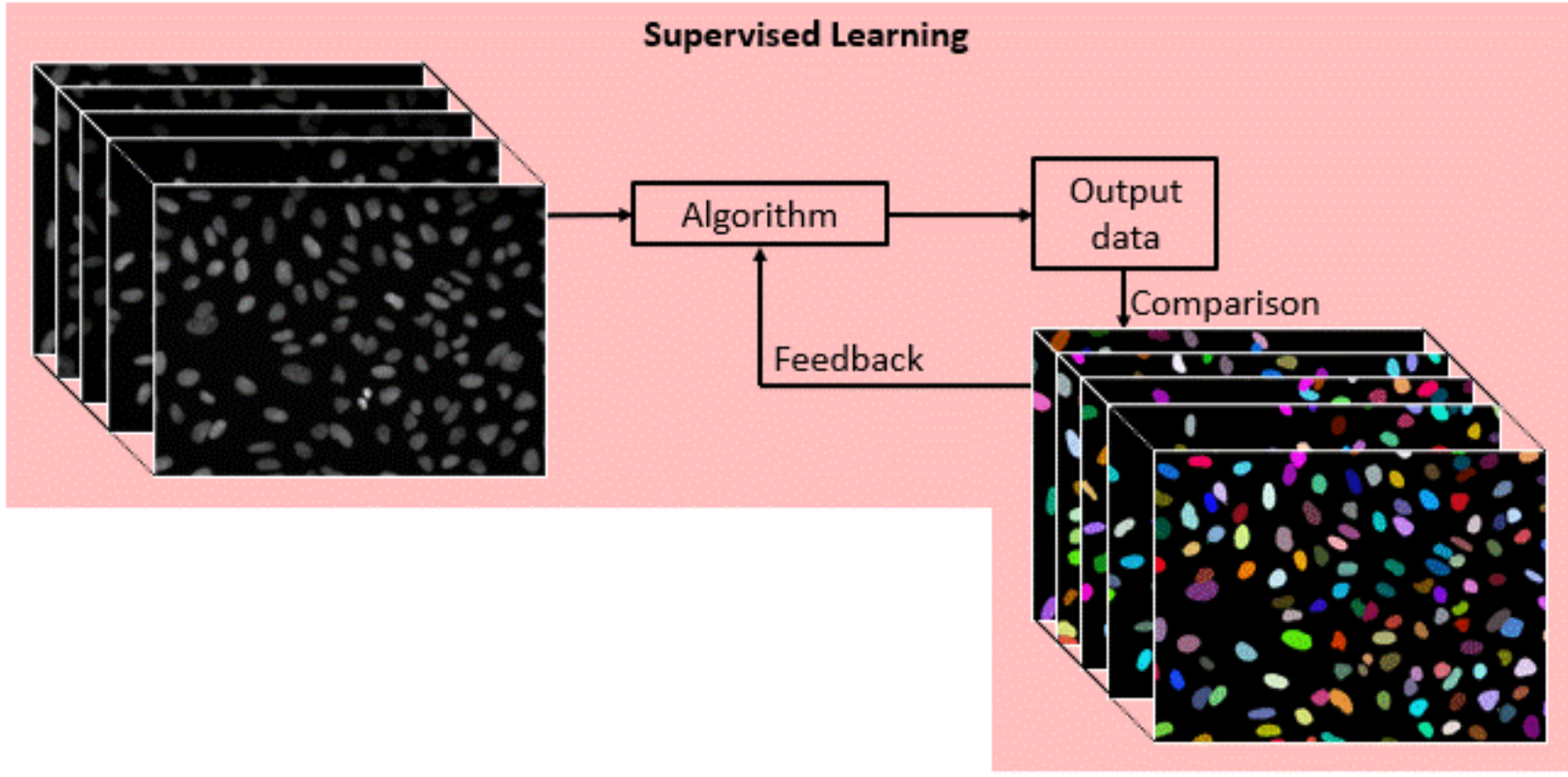

Supervised Learning
Input data
Algorithm
Output data
Comparison
Feedback
Annotated input data

MACHINE LEARNING

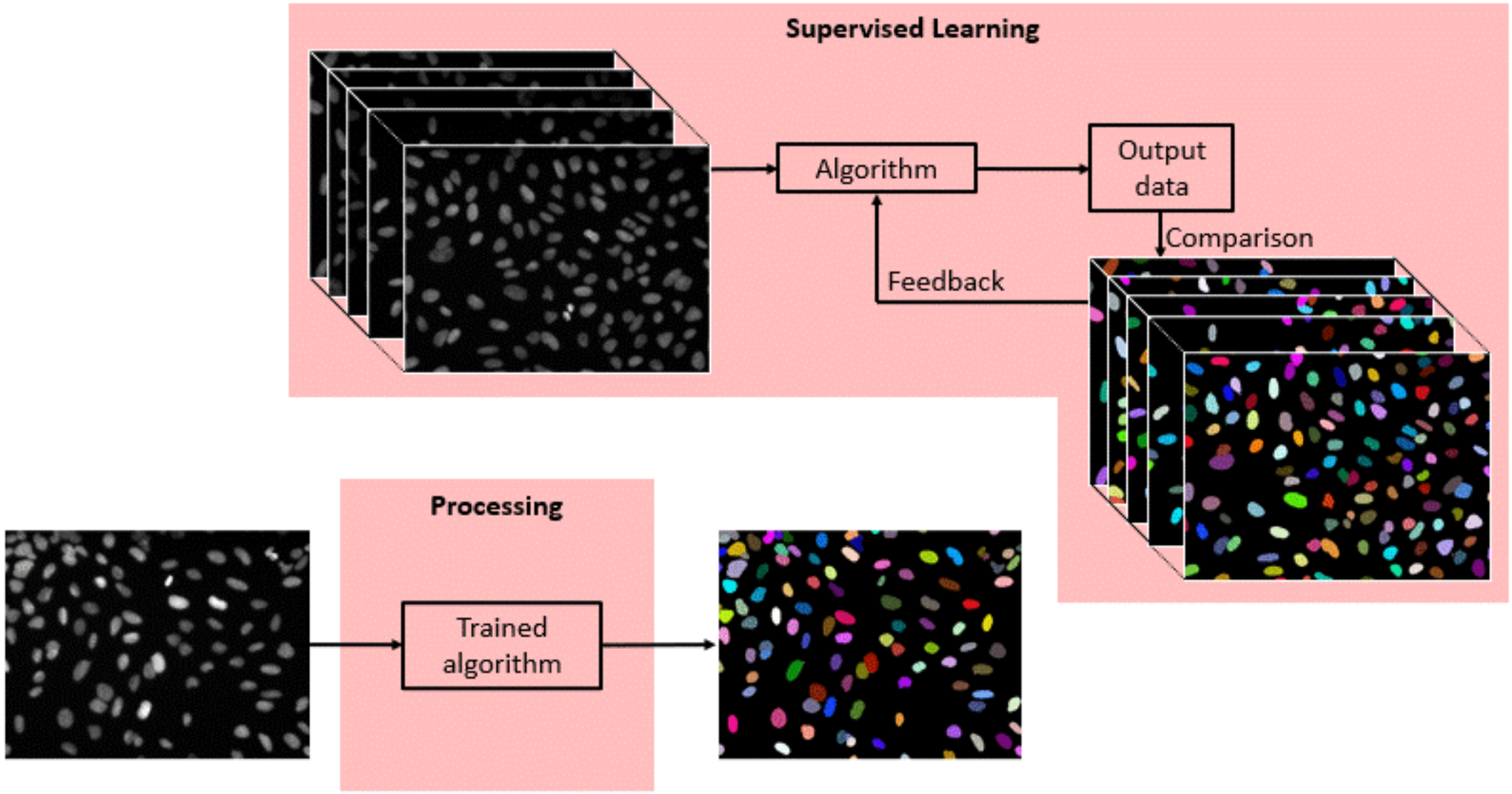




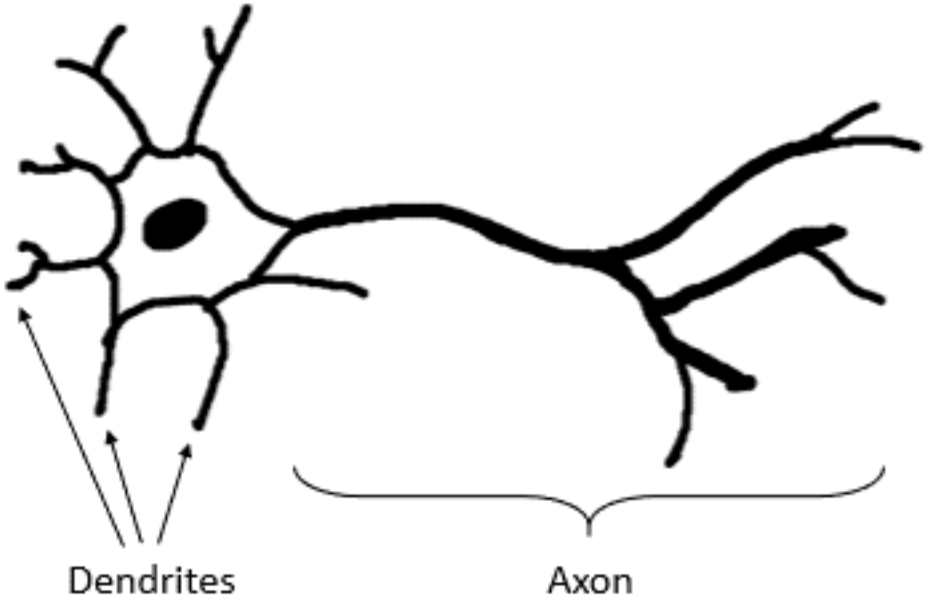
MACHINE LEARNING

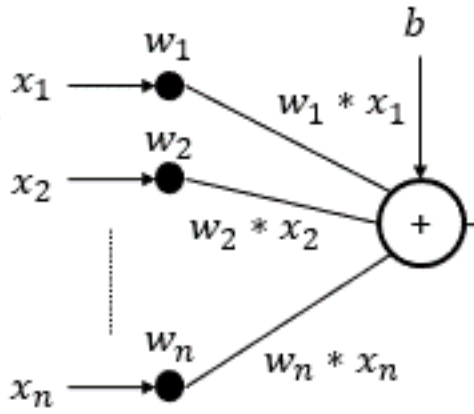


Machine Learning



Artificial Neuron (McCulloch and Pitts, 1943)

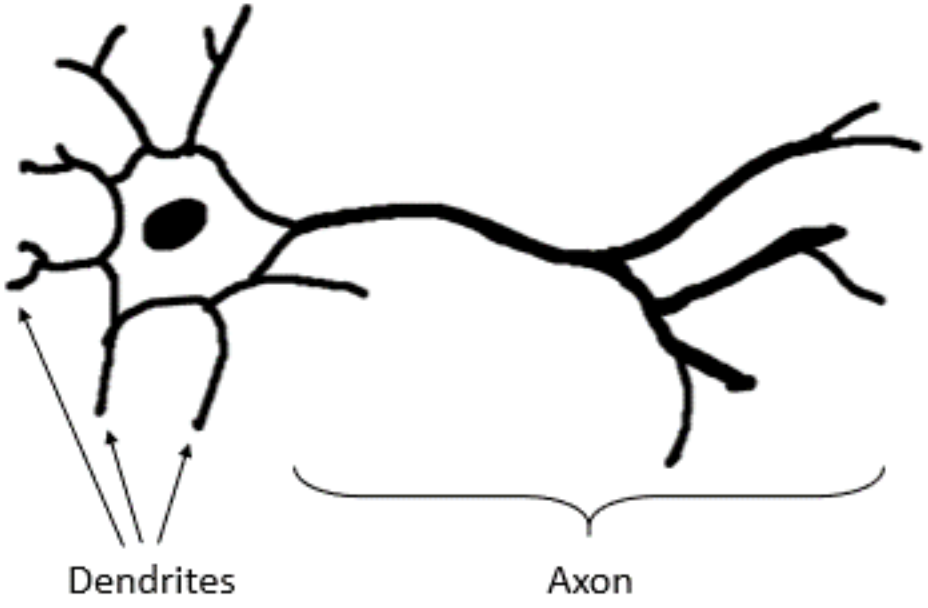


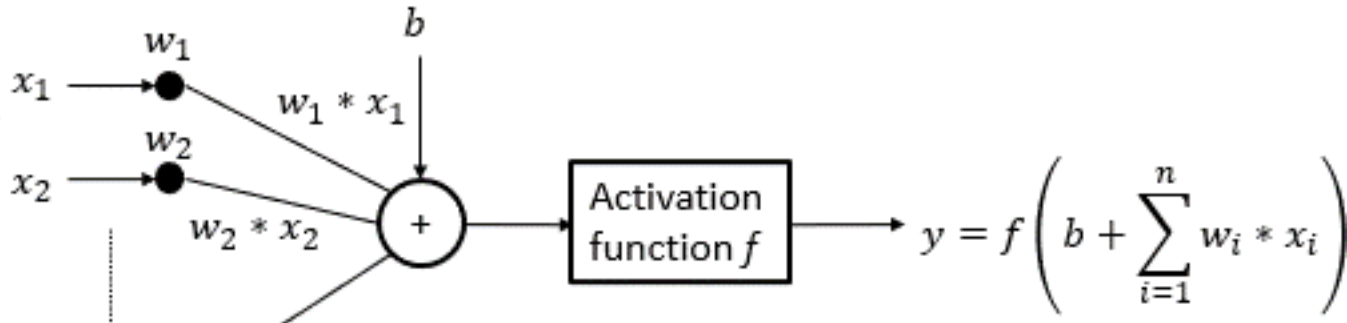


Activation function f

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

Artificial Neuron (McCulloch and Pitts, 1943)

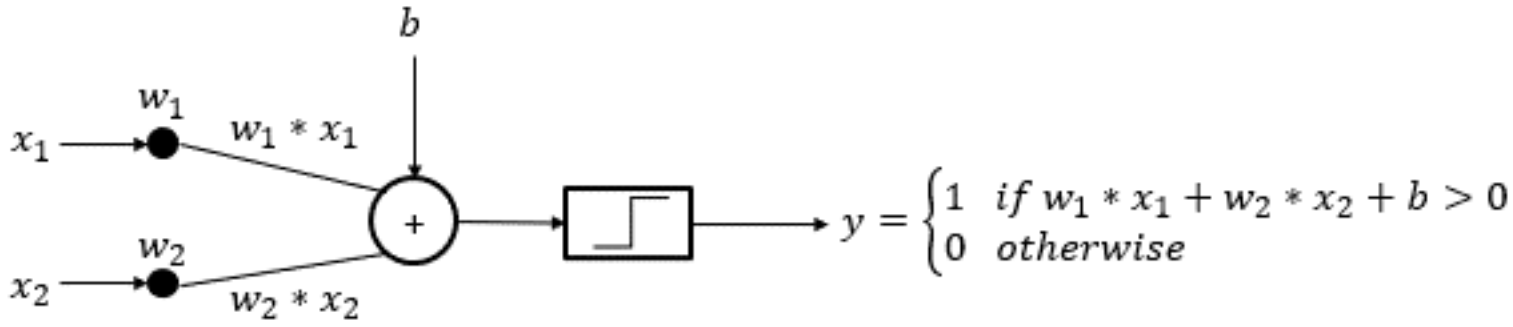




Analogy with biological neuron:

x_i:	axons from other neurons (input axons)
w_i:	dendrites
y:	output axon

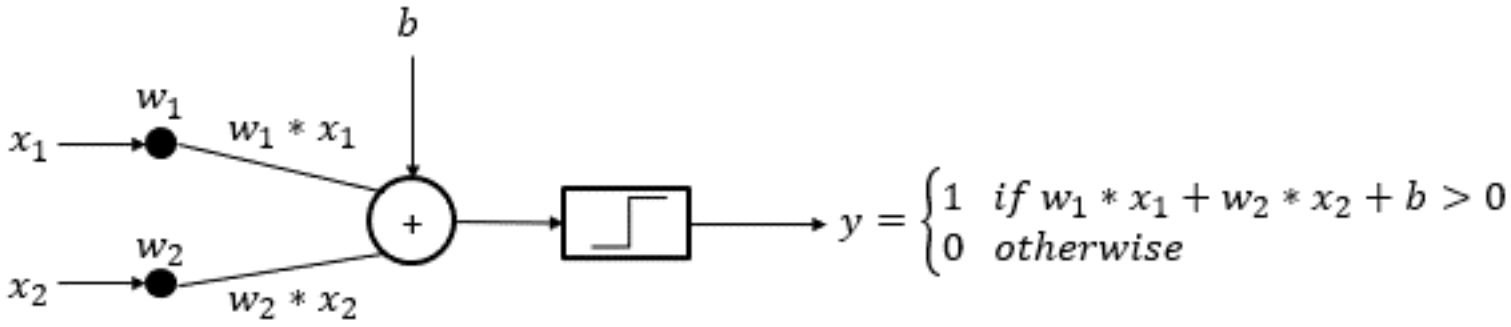
PERCEPTRON (ROSENBLATT, 1958)



PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

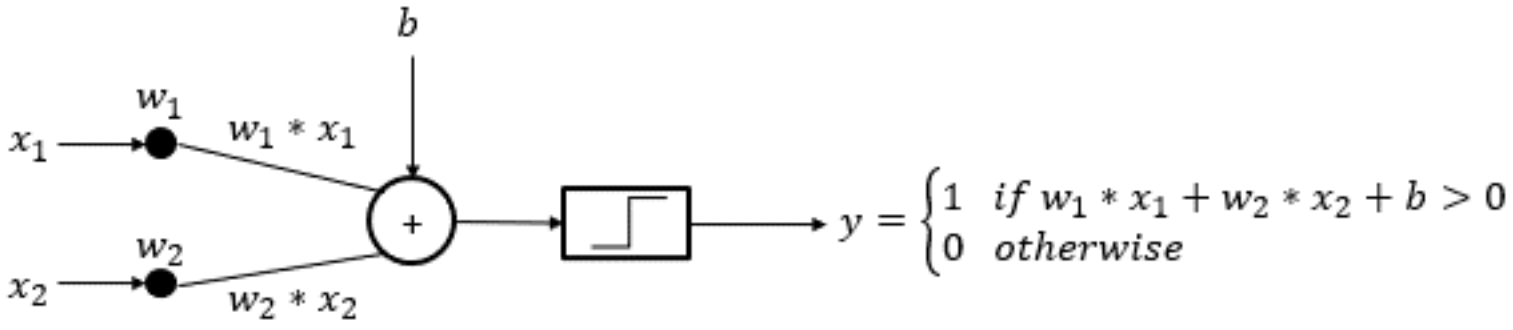
- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is a constant (**learning rate)** and *t* is the **iteration**



PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is a constant (**learning rate**) and *t* is the **iteration**



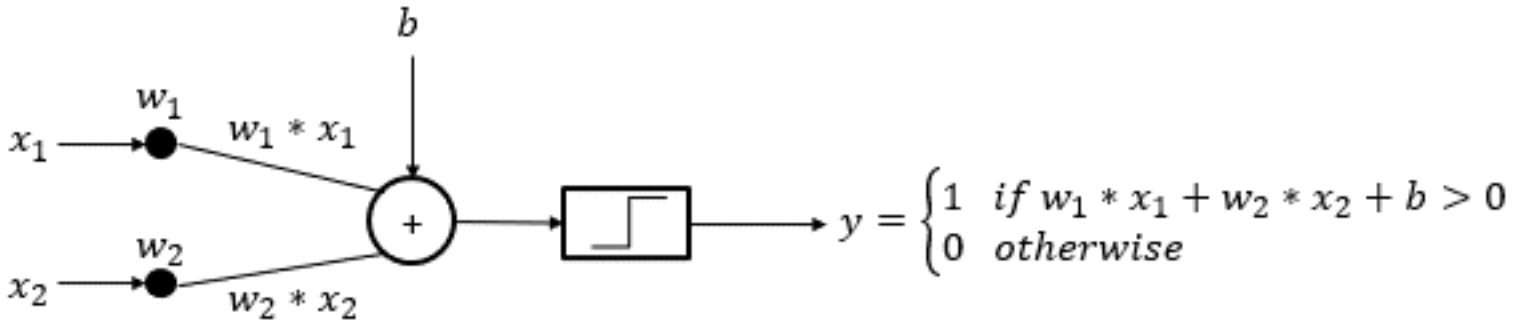
OR gate:

x_1	x_2	y_{real}
0	0	0
1	0	1
0	1	1
1	1	1

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is a constant (**learning rate**) and *t* is the **iteration**



OR gate:

x_1	x_2	y_{real}
0	0	0
1	0	1
0	1	1
1	1	1

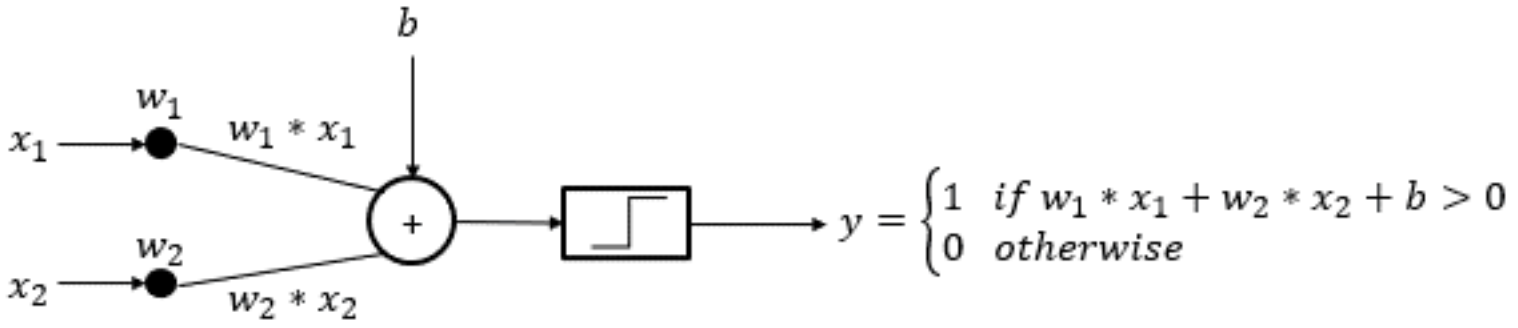
Initialization:

$w_1(0)$	$w_2(0)$	$b(0)$
0	0	0

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is a constant (**learning rate**) and *t* is the **iteration**



OR gate:

x_1	x_2	y_{real}
0	0	0
1	0	1
0	1	1
1	1	1

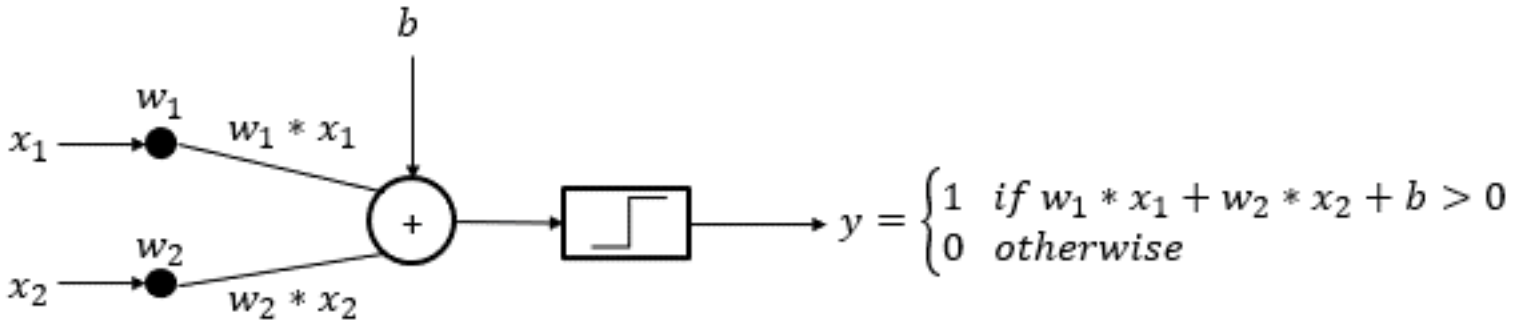
Initialization:

$w_1(0)$	$w_2(0)$	$b(0)$
0	0	0

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is a constant (**learning rate**) and *t* is the **iteration**



OR gate:

x_1	x_2	y_{real}
0	0	0
1	0	1
0	1	1
1	1	1

$$y = f(w_1(0) * x_1 + w_2(0) * x_2 + b(0))$$
$$y = f(0 * 0 + 0 * 0 + 0) = f(0) = 0$$

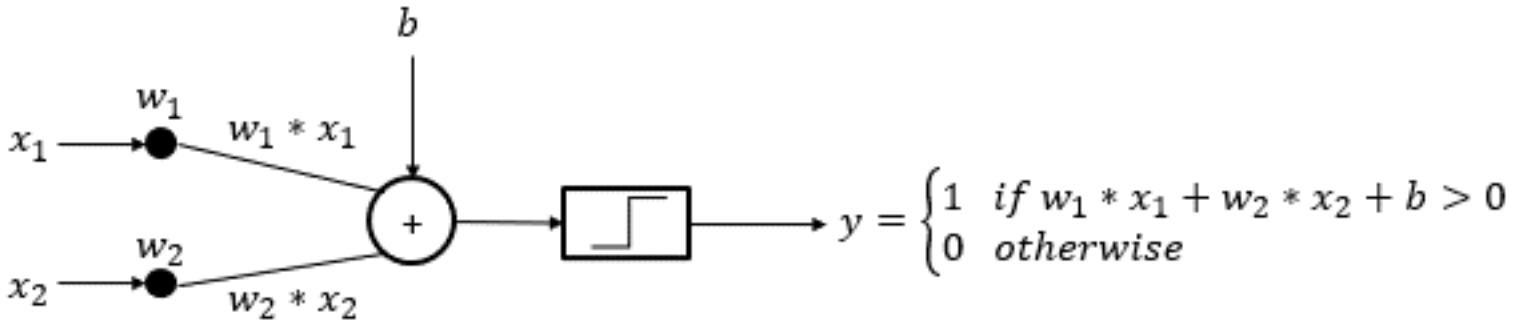
Initialization:

$w_1(0)$	$w_2(0)$	$b(0)$
0	0	0

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is a constant (**learning rate**) and *t* is the **iteration**



OR gate:

x_1	x_2	y_{real}
0	0	0
1	0	1
0	1	1
1	1	1

$y = f(w_1(0) * x_1 + w_2(0) * x_2 + b(0))$
$y = f(0 * 0 + 0 * 0 + 0) = f(0) = 0$

$w_1(1) = w_1(0) + lr * (y_{real} - y) * x_1$
$w_1(1) = 0 + 0.5 * 0 * 0 = 0$

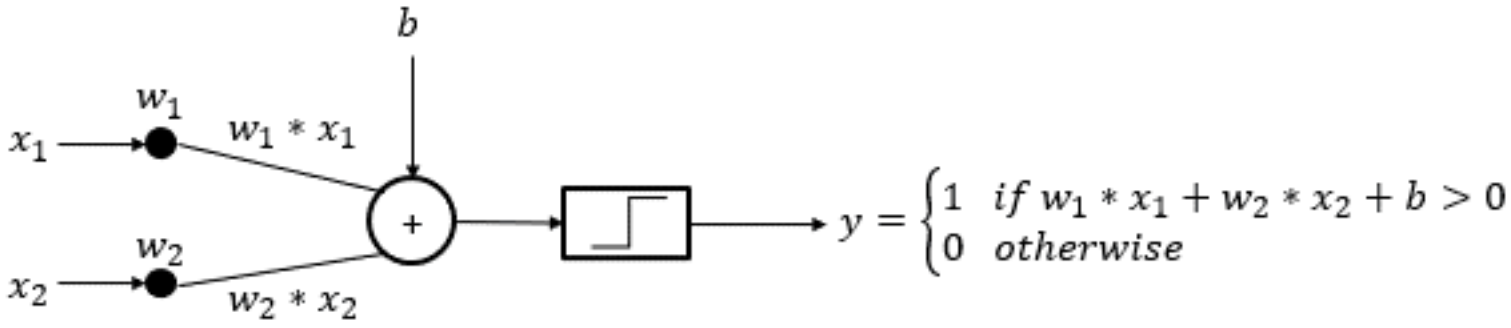
Initialization:

$w_1(0)$	$w_2(0)$	$b(0)$
0	0	0

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is a constant (**learning rate**) and *t* is the **iteration**



OR gate:

x_1	x_2	y_{real}
0	0	0
1	0	1
0	1	1
1	1	1

$y = f(w_1(0) * x_1 + w_2(0) * x_2 + b(0))$
$y = f(0 * 0 + 0 * 0 + 0) = f(0) = 0$

$w_1(1) = w_1(0) + lr * (y_{real} - y) * x_1$
$w_1(1) = 0 + 0.5 * 0 * 0 = 0$

$w_2(1) = w_2(0) + lr * (y_{real} - y) * x_2$
$w_2(1) = 0 + 0.5 * 0 * 0 = 0$

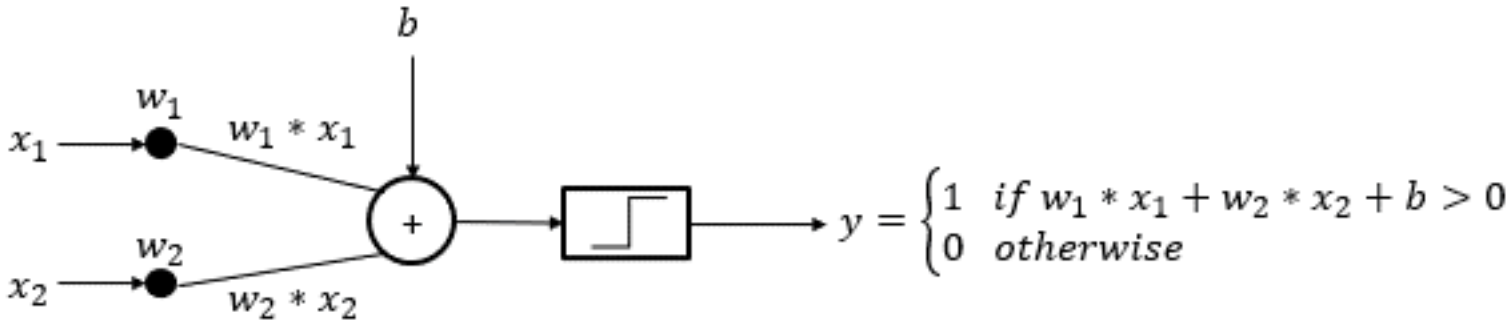
Initialization:

$w_1(0)$	$w_2(0)$	$b(0)$
0	0	0

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is a constant (**learning rate**) and *t* is the **iteration**



OR gate:

x_1	x_2	y_{real}
0	0	0
1	0	1
0	1	1
1	1	1

$y = f(w_1(0) * x_1 + w_2(0) * x_2 + b(0))$
$y = f(0 * 0 + 0 * 0 + 0) = f(0) = 0$

$w_1(1) = w_1(0) + lr * (y_{real} - y) * x_1$
$w_1(1) = 0 + 0.5 * 0 * 0 = 0$

$w_2(1) = w_2(0) + lr * (y_{real} - y) * x_2$
$w_2(1) = 0 + 0.5 * 0 * 0 = 0$

$b(1) = b(0) + lr * (y_{real} - y)$
$b(1) = 0 + 0.5 * 0 = 0$

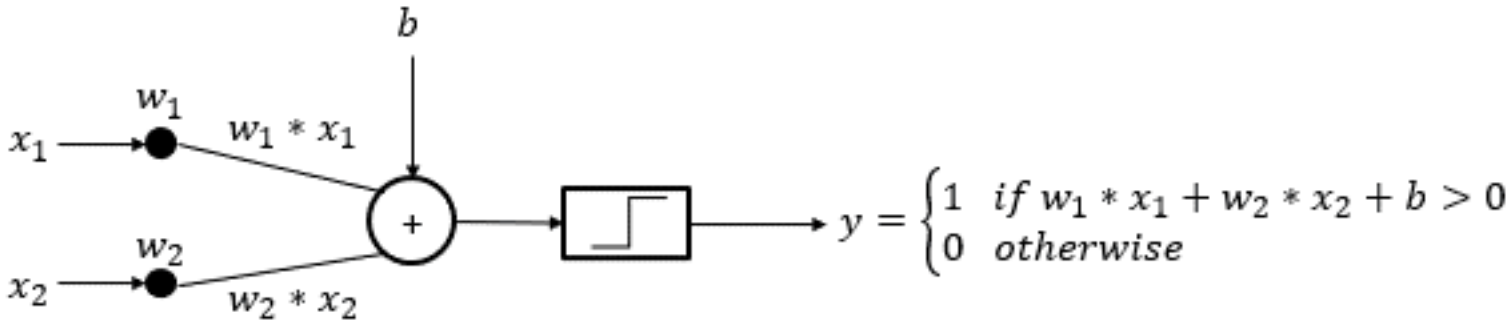
Initialization:

$w_1(0)$	$w_2(0)$	$b(0)$
0	0	0

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is a constant (**learning rate**) and *t* is the **iteration**



OR gate:

x_1	x_2	y_{real}
0	0	0
1	0	1
0	1	1
1	1	1

$y = f(w_1(0) * x_1 + w_2(0) * x_2 + b(0))$
$y = f(0 * 0 + 0 * 0 + 0) = f(0) = 0$

$w_1(1) = w_1(0) + lr * (y_{real} - y) * x_1$
$w_1(1) = 0 + 0.5 * 0 * 0 = 0$

$w_2(1) = w_2(0) + lr * (y_{real} - y) * x_2$
$w_2(1) = 0 + 0.5 * 0 * 0 = 0$

$b(1) = b(0) + lr * (y_{real} - y)$
$b(1) = 0 + 0.5 * 0 = 0$

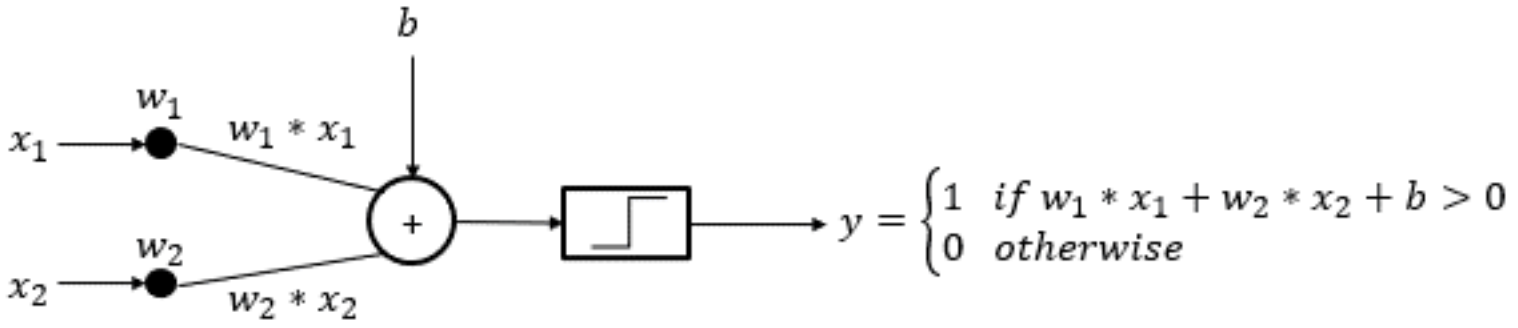
Iteration 1:

$w_1(1)$	$w_2(1)$	$b(1)$
0	0	0

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is a constant (**learning rate**) and *t* is the **iteration**



OR gate:

x_1	x_2	y_{real}
0	0	0
1	0	1
0	1	1
1	1	1

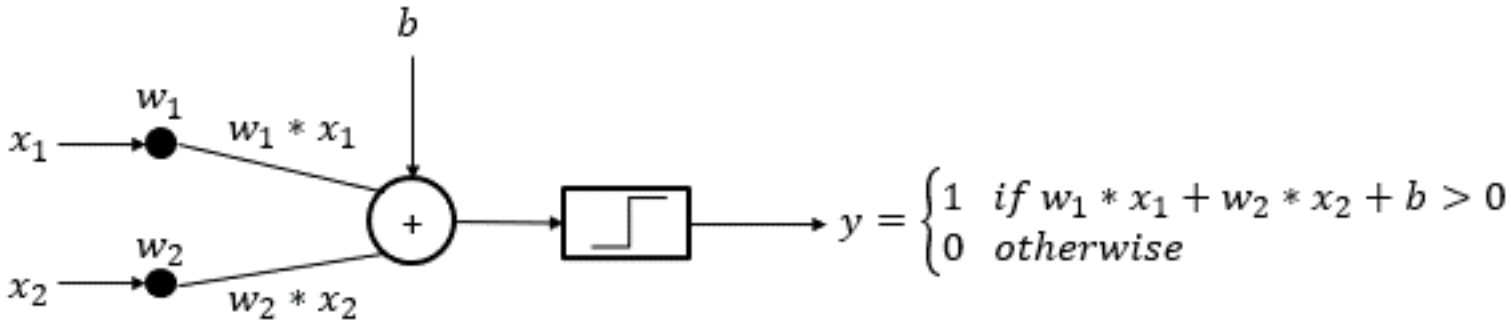
Iteration 1:

$w_1(1)$	$w_2(1)$	$b(1)$
0	0	0

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is a constant (**learning rate)** and *t* is the **iteration**



OR gate:

x_1	x_2	y_{real}
0	0	0
1	0	1
0	1	1
1	1	1

$y = f(w_1(1) * x_1 + w_2(1) * x_2 + b(1))$
$y = f(0 * 1 + 0 * 0 + 0) = f(0) = 0$

$w_1(2) = w_1(1) + lr * (y_{real} - y) * x_1$
$w_1(2) = 0 + 0.5 * 1 * 1 = 0.5$

$w_2(2) = w_2(1) + lr * (y_{real} - y) * x_2$
$w_2(2) = 0 + 0.5 * 1 * 0 = 0$

$b(2) = b(1) + lr * (y_{real} - y)$
$b(2) = 0 + 0.5 * 1 = 0.5$

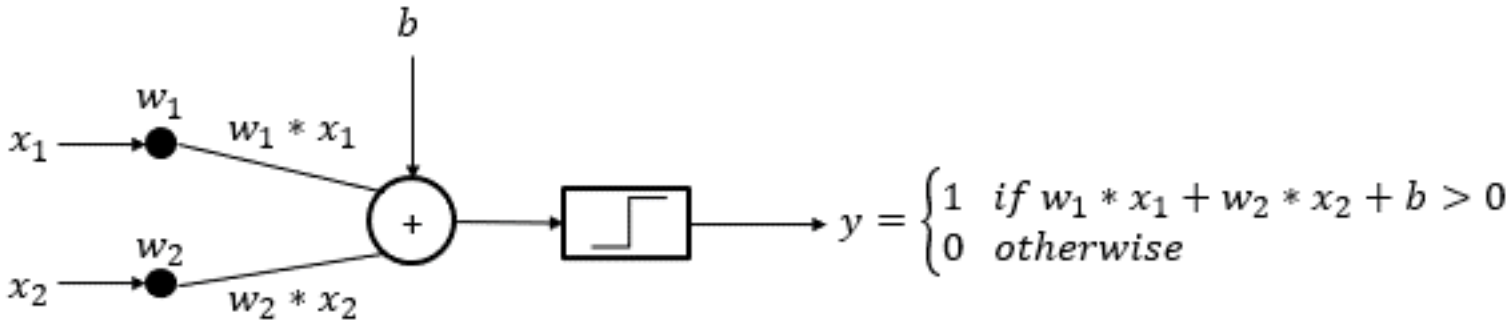
Iteration 1:

$w_1(1)$	$w_2(1)$	$b(1)$
0	0	0

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is a constant (**learning rate)** and *t* is the **iteration**



OR gate:

x_1	x_2	y_{real}
0	0	0
1	0	1
0	1	1
1	1	1

$y = f(w_1(1) * x_1 + w_2(1) * x_2 + b(1))$
$y = f(0 * 1 + 0 * 0 + 0) = f(0) = 0$

$w_1(2) = w_1(1) + lr * (y_{real} - y) * x_1$
$w_1(2) = 0 + 0.5 * 1 * 1 = 0.5$

$w_2(2) = w_2(1) + lr * (y_{real} - y) * x_2$
$w_2(2) = 0 + 0.5 * 1 * 0 = 0$

$b(2) = b(1) + lr * (y_{real} - y)$
$b(2) = 0 + 0.5 * 1 = 0.5$

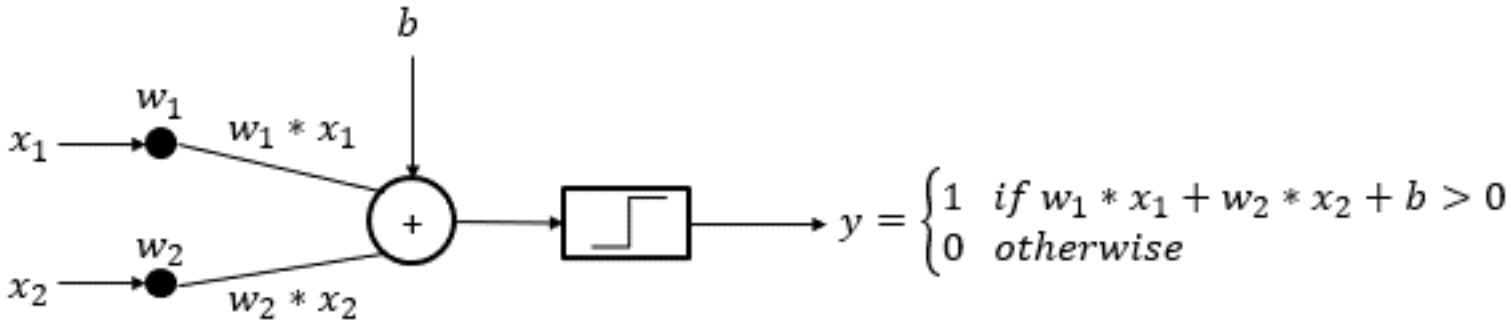
Iteration 2:

$w_1(2)$	$w_2(2)$	$b(2)$
0.5	0	0.5

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is a constant (**learning rate**) and *t* is the **iteration**



OR gate:

x_1	x_2	y_{real}
0	0	0
1	0	1
0	1	1
1	1	1

$y = f(w_1(2) * x_1 + w_2(2) * x_2 + b(2))$
$y = f(0.5 * 0 + 0 * 1 + 0.5) = f(0.5) = 1$

$w_1(3) = w_1(2) + lr * (y_{real} - y) * x_1$
$w_1(3) = 0.5 + 0.5 * 0 * 0 = 0.5$

$w_2(3) = w_2(2) + lr * (y_{real} - y) * x_2$
$w_2(3) = 0 + 0.5 * 0 * 1 = 0$

$b(3) = b(2) + lr * (y_{real} - y)$
$b(3) = 0.5 + 0.5 * 0 = 0.5$

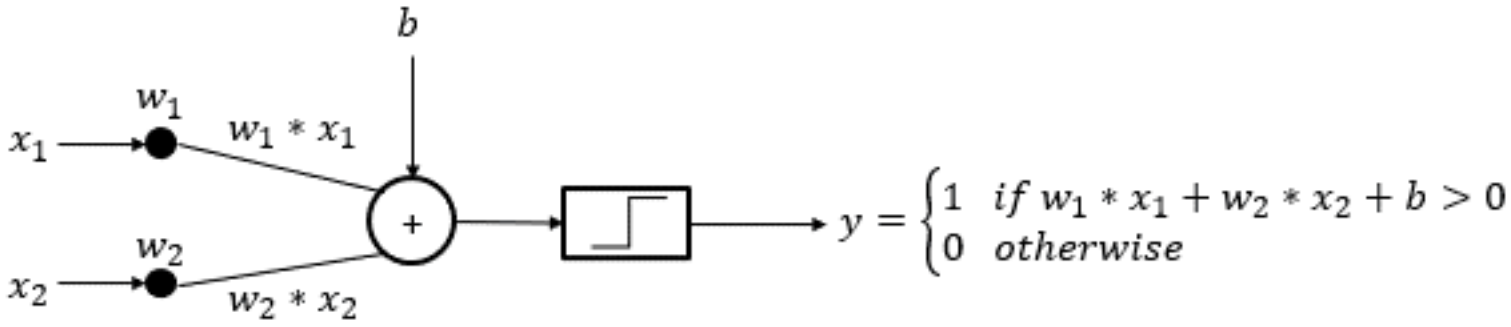
Iteration 2:

$w_1(2)$	$w_2(2)$	$b(2)$
0.5	0	0.5

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is a constant (**learning rate**) and *t* is the **iteration**



OR gate:

x_1	x_2	y_{real}
0	0	0
1	0	1
0	1	1
1	1	1

$y = f(w_1(2) * x_1 + w_2(2) * x_2 + b(2))$
$y = f(0.5 * 0 + 0 * 1 + 0.5) = f(0.5) = 1$

$w_1(3) = w_1(2) + lr * (y_{real} - y) * x_1$
$w_1(3) = 0.5 + 0.5 * 0 * 0 = 0.5$

$w_2(3) = w_2(2) + lr * (y_{real} - y) * x_2$
$w_2(3) = 0 + 0.5 * 0 * 1 = 0$

$b(3) = b(2) + lr * (y_{real} - y)$
$b(3) = 0.5 + 0.5 * 0 = 0.5$

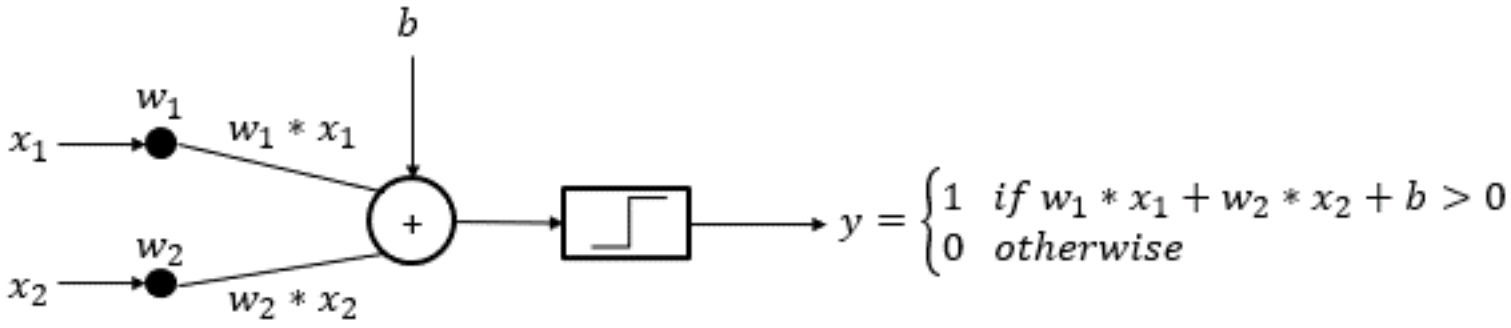
Iteration 3:

$w_1(3)$	$w_2(3)$	$b(3)$
0.5	0	0.5

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is a constant (**learning rate**) and *t* is the **iteration**



OR gate:

x_1	x_2	y_{real}
0	0	0
1	0	1
0	1	1
1	1	1

$y = f(w_1(3) * x_1 + w_2(3) * x_2 + b(3))$
$y = f(0.5 * 1 + 0 * 1 + 0.5) = f(1) = 1$

$w_1(4) = w_1(3) + lr * (y_{real} - y) * x_1$
$w_1(4) = 0.5 + 0.5 * 0 * 1 = 0.5$

$w_2(4) = w_2(3) + lr * (y_{real} - y) * x_2$
$w_2(4) = 0 + 0.5 * 0 * 1 = 0$

$b(4) = b(3) + lr * (y_{real} - y)$
$b(4) = 0.5 + 0.5 * 0 = 0.5$

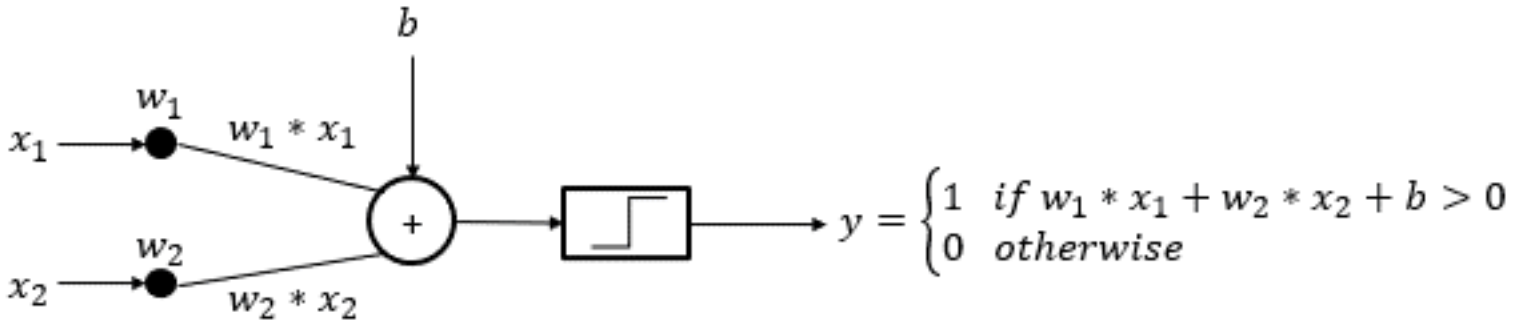
Iteration 3:

$w_1(3)$	$w_2(3)$	$b(3)$
0.5	0	0.5

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is **learning rate** and t is the **iteration**



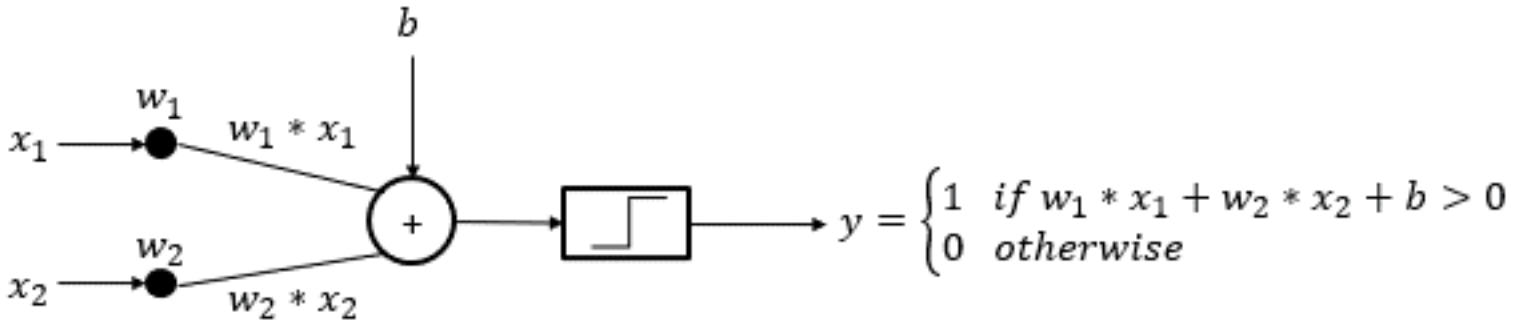
AND gate:

x_1	x_2	y_{real}
0	0	0
1	0	0
0	1	0
1	1	1

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is **learning rate** and t is the **iteration**



AND gate:

x_1	x_2	y_{real}
0	0	0
1	0	0
0	1	0
1	1	1

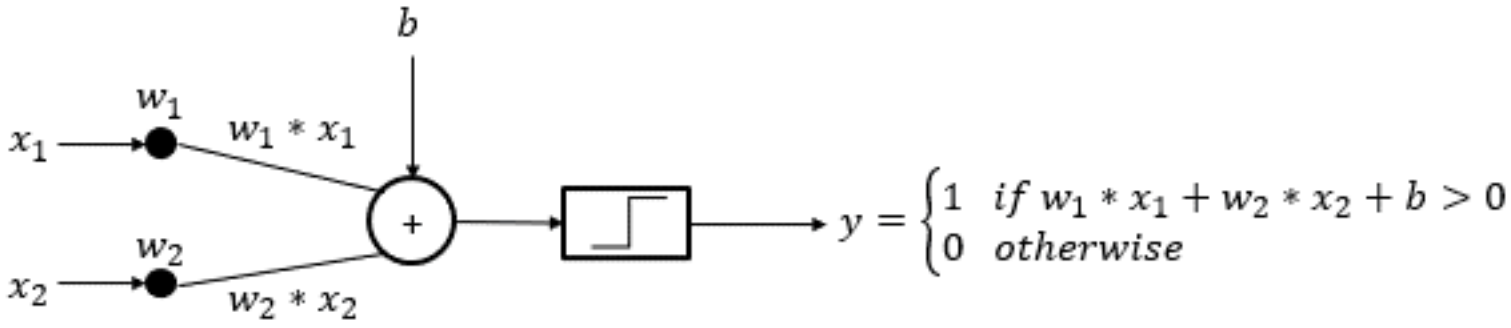
Initialization:

$w_1(0)$	$w_2(0)$	$b(0)$
0	0	0

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is **learning rate** and t is the **iteration**



AND gate:

x_1	x_2	y_{real}
0	0	0
1	0	0
0	1	0
1	1	1

$$y = f(w_1(0) * x_1 + w_2(0) * x_2 + b(0))$$
$$y = f(0 * 0 + 0 * 0 + 0) = f(0) = 0$$

$$w_1(1) = w_1(0) + lr * (y_{real} - y) * x_1$$
$$w_1(1) = 0 + 0.5 * 0 * 0 = 0$$

$$w_2(1) = w_2(0) + lr * (y_{real} - y) * x_2$$
$$w_2(1) = 0 + 0.5 * 0 * 0 = 0$$

$$b(1) = b(0) + lr * (y_{real} - y)$$
$$b(1) = 0 + 0.5 * 0 = 0$$

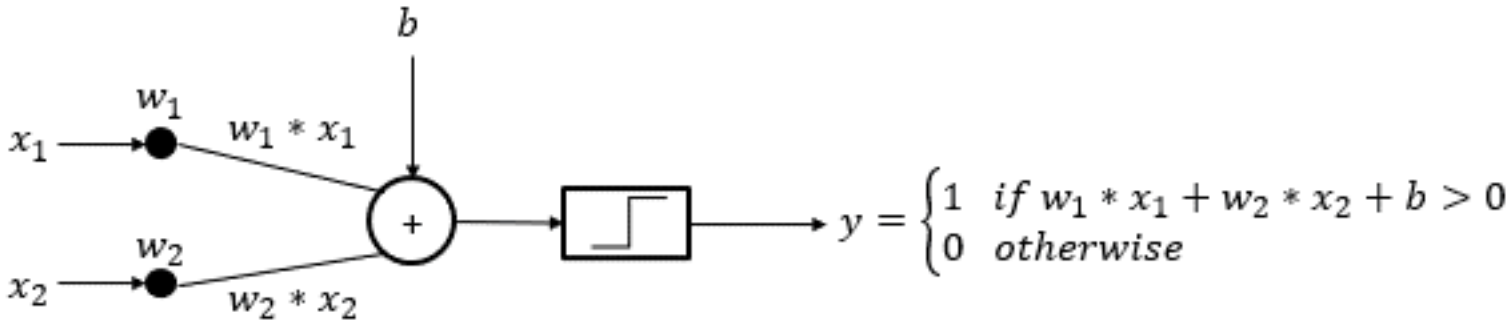
Initialization:

$w_1(0)$	$w_2(0)$	$b(0)$
0	0	0

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is **learning rate** and t is the **iteration**



AND gate:

x_1	x_2	y_{real}
0	0	0
1	0	0
0	1	0
1	1	1

$y = f(w_1(0) * x_1 + w_2(0) * x_2 + b(0))$
$y = f(0 * 0 + 0 * 0 + 0) = f(0) = 0$

$w_1(1) = w_1(0) + lr * (y_{real} - y) * x_1$
$w_1(1) = 0 + 0.5 * 0 * 0 = 0$

$w_2(1) = w_2(0) + lr * (y_{real} - y) * x_2$
$w_2(1) = 0 + 0.5 * 0 * 0 = 0$

$b(1) = b(0) + lr * (y_{real} - y)$
$b(1) = 0 + 0.5 * 0 = 0$

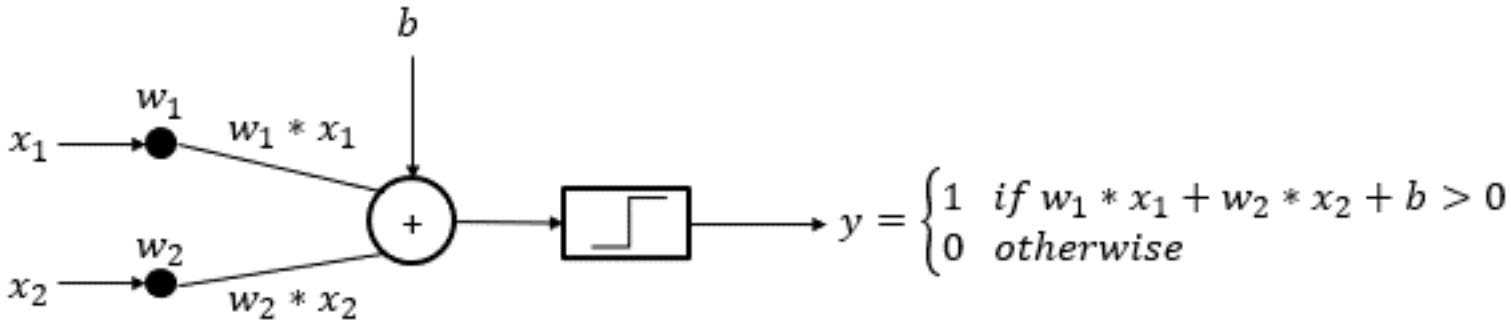
Iteration 1:

$w_1(1)$	$w_2(1)$	$b(1)$
0	0	0

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is **learning rate** and *t* is the **iteration**



AND gate:

x_1	x_2	y_{real}
0	0	0
1	0	0
0	1	0
1	1	1

$y = f(w_1(1) * x_1 + w_2(1) * x_2 + b(1))$
$y = f(0 * 1 + 0 * 0 + 0) = f(0) = 0$

$w_1(2) = w_1(1) + lr * (y_{real} - y) * x_1$
$w_1(2) = 0 + 0.5 * 0 * 1 = 0$

$w_2(2) = w_2(1) + lr * (y_{real} - y) * x_2$
$w_2(2) = 0 + 0.5 * 0 * 0 = 0$

$b(2) = b(1) + lr * (y_{real} - y)$
$b(2) = 0 + 0.5 * 0 = 0$

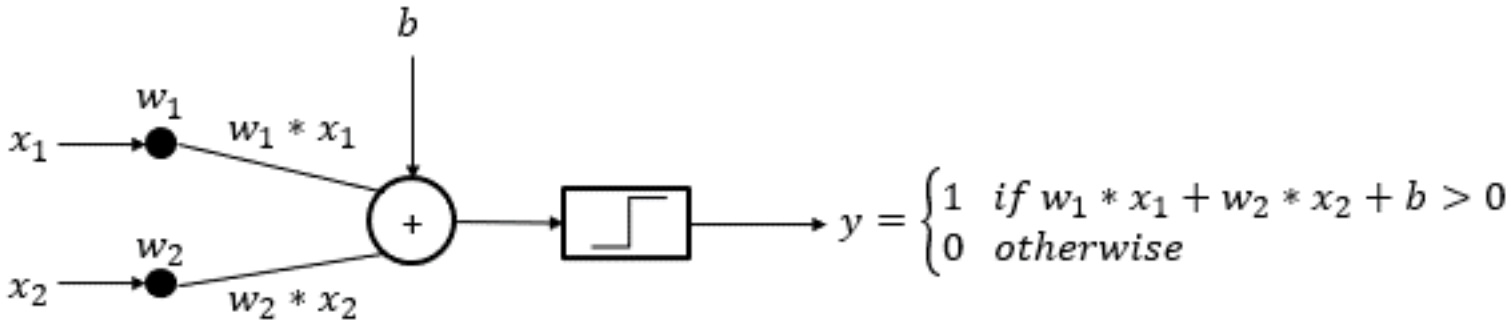
Iteration 1:

$w_1(1)$	$w_2(1)$	$b(1)$
0	0	0

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is **learning rate** and *t* is the **iteration**



AND gate:

x_1	x_2	y_{real}
0	0	0
1	0	0
0	1	0
1	1	1

$y = f(w_1(1) * x_1 + w_2(1) * x_2 + b(1))$
$y = f(0 * 1 + 0 * 0 + 0) = f(0) = 0$

$w_1(2) = w_1(1) + lr * (y_{real} - y) * x_1$
$w_1(2) = 0 + 0.5 * 0 * 1 = 0$

$w_2(2) = w_2(1) + lr * (y_{real} - y) * x_2$
$w_2(2) = 0 + 0.5 * 0 * 0 = 0$

$b(2) = b(1) + lr * (y_{real} - y)$
$b(2) = 0 + 0.5 * 0 = 0$

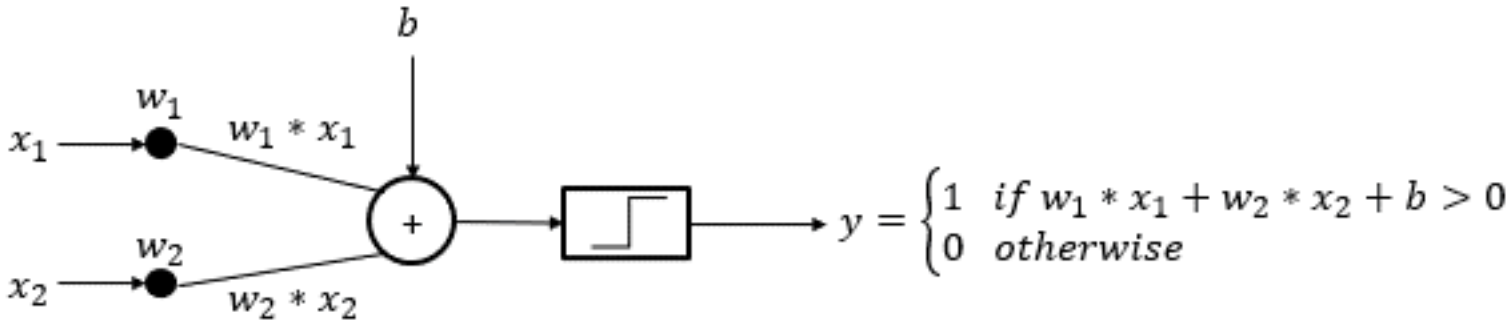
Iteration 2:

$w_1(2)$	$w_2(2)$	$b(2)$
0	0	0

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is **learning rate** and *t* is the **iteration**



AND gate:

x_1	x_2	y_{real}
0	0	0
1	0	0
0	1	0
1	1	1

$y = f(w_1(2) * x_1 + w_2(2) * x_2 + b(2))$
$y = f(0 * 0 + 0 * 1 + 0) = f(0) = 0$

$w_1(3) = w_1(2) + lr * (y_{real} - y) * x_1$
$w_1(3) = 0 + 0.5 * 0 * 0 = 0$

$w_2(3) = w_2(2) + lr * (y_{real} - y) * x_2$
$w_2(3) = 0 + 0.5 * 0 * 1 = 0$

$b(3) = b(2) + lr * (y_{real} - y)$
$b(3) = 0 + 0.5 * 0 = 0$

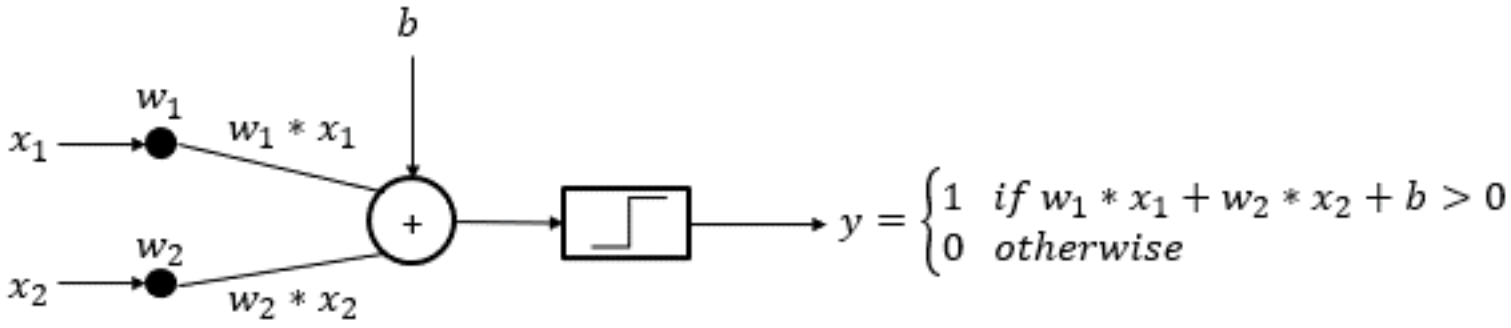
Iteration 2:

$w_1(2)$	$w_2(2)$	$b(2)$
0	0	0

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is **learning rate** and *t* is the **iteration**



AND gate:

x_1	x_2	y_{real}
0	0	0
1	0	0
0	1	0
1	1	1

$y = f(w_1(2) * x_1 + w_2(2) * x_2 + b(2))$
$y = f(0 * 0 + 0 * 1 + 0) = f(0) = 0$

$w_1(3) = w_1(2) + lr * (y_{real} - y) * x_1$
$w_1(3) = 0 + 0.5 * 0 * 0 = 0$

$w_2(3) = w_2(2) + lr * (y_{real} - y) * x_2$
$w_2(3) = 0 + 0.5 * 0 * 1 = 0$

$b(3) = b(2) + lr * (y_{real} - y)$
$b(3) = 0 + 0.5 * 0 = 0$

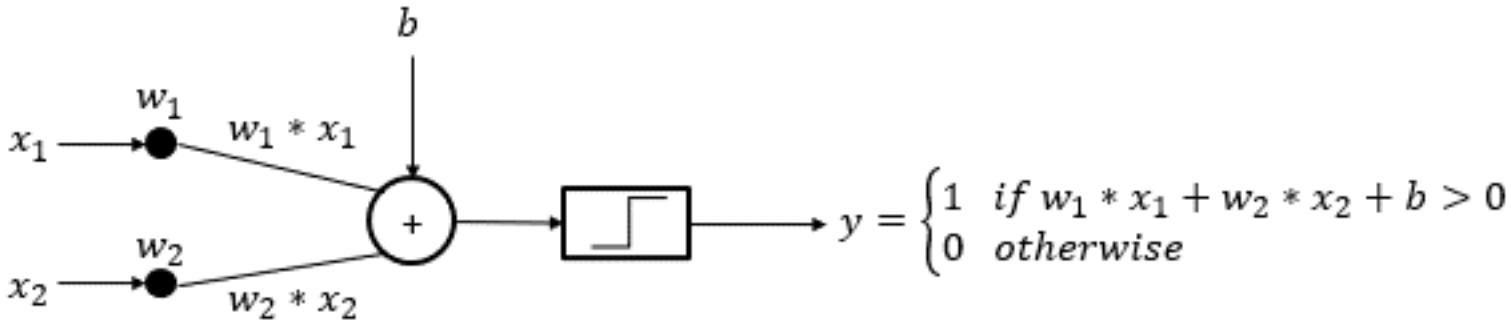
Iteration 3:

$w_1(3)$	$w_2(3)$	$b(3)$
0	0	0

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is **learning rate** and *t* is the **iteration**



AND gate:

x_1	x_2	y_{real}
0	0	0
1	0	0
0	1	0
1	1	1

$y = f(w_1(3) * x_1 + w_2(3) * x_2 + b(3))$
$y = f(0 * 1 + 0 * 1 + 0) = f(0) = 0$

$w_1(4) = w_1(3) + lr * (y_{real} - y) * x_1$
$w_1(4) = 0 + 0.5 * 1 * 1 = 0.5$

$w_2(4) = w_2(3) + lr * (y_{real} - y) * x_2$
$w_2(4) = 0 + 0.5 * 1 * 1 = 0.5$

$b(4) = b(3) + lr * (y_{real} - y)$
$b(4) = 0 + 0.5 * 1 = 0.5$

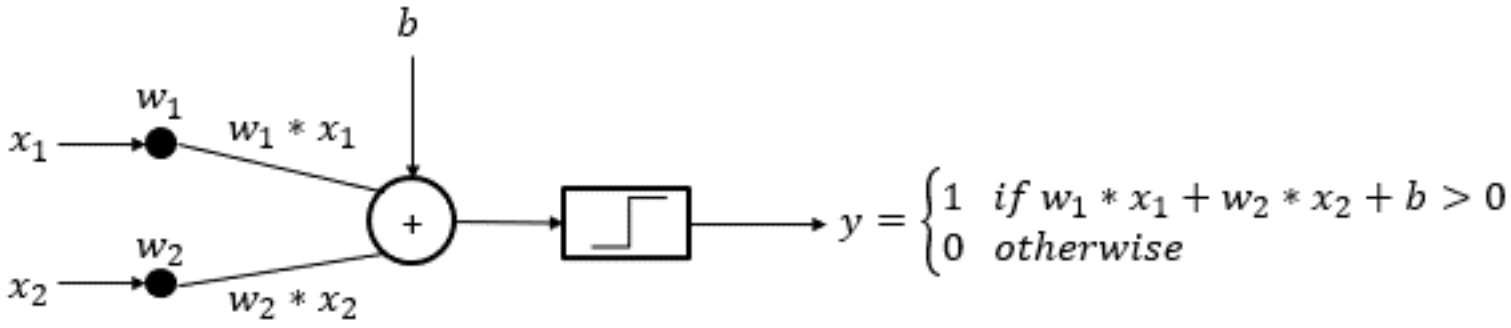
Iteration 3:

$w_1(3)$	$w_2(3)$	$b(3)$
0	0	0

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is **learning rate** and *t* is the **iteration**



AND gate:

x_1	x_2	y_{real}
0	0	0
1	0	0
0	1	0
1	1	1

$y = f(w_1(3) * x_1 + w_2(3) * x_2 + b(3))$
$y = f(0 * 1 + 0 * 1 + 0) = f(0) = 0$

$w_1(4) = w_1(3) + lr * (y_{real} - y) * x_1$
$w_1(4) = 0 + 0.5 * 1 * 1 = 0.5$

$w_2(4) = w_2(3) + lr * (y_{real} - y) * x_2$
$w_2(4) = 0 + 0.5 * 1 * 1 = 0.5$

$b(4) = b(3) + lr * (y_{real} - y)$
$b(4) = 0 + 0.5 * 1 = 0.5$

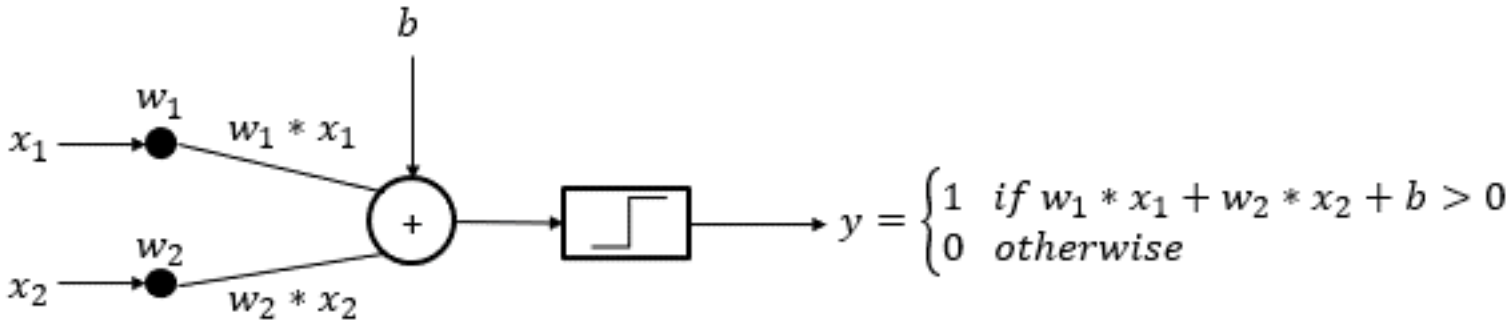
Iteration 4:

$w_1(4)$	$w_2(4)$	$b(4)$
0.5	0.5	0.5

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is **learning rate** and *t* is the **iteration**



AND gate:

x_1	x_2	y_{real}
0	0	0
1	0	0
0	1	0
1	1	1

$y = f(w_1(4) * x_1 + w_2(4) * x_2 + b(4))$
$y = f(0.5 * 0 + 0.5 * 0 + 0.5) = f(0.5) = 1$

$w_1(5) = w_1(4) + lr * (y_{real} - y) * x_1$
$w_1(5) = 0.5 + 0.5 * (-1) * 0 = 0.5$

$w_2(5) = w_2(4) + lr * (y_{real} - y) * x_2$
$w_2(5) = 0.5 + 0.5 * (-1) * 0 = 0.5$

$b(5) = b(4) + lr * (y_{real} - y)$
$b(5) = 0.5 + 0.5 * (-1) = 0$

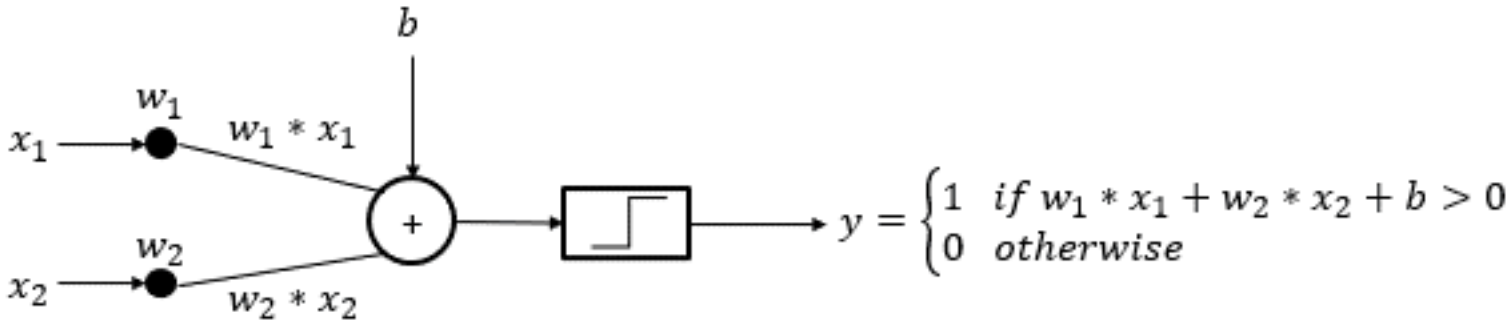
Iteration 4:

$w_1(4)$	$w_2(4)$	$b(4)$
0.5	0.5	0.5

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is **learning rate** and *t* is the **iteration**



AND gate:

x_1	x_2	y_{real}
0	0	0
1	0	0
0	1	0
1	1	1

$y = f(w_1(4) * x_1 + w_2(4) * x_2 + b(4))$
$y = f(0.5 * 0 + 0.5 * 0 + 0.5) = f(0.5) = 1$

$w_1(5) = w_1(4) + lr * (y_{real} - y) * x_1$
$w_1(5) = 0.5 + 0.5 * (-1) * 0 = 0.5$

$w_2(5) = w_2(4) + lr * (y_{real} - y) * x_2$
$w_2(5) = 0.5 + 0.5 * (-1) * 0 = 0.5$

$b(5) = b(4) + lr * (y_{real} - y)$
$b(5) = 0.5 + 0.5 * (-1) = 0$

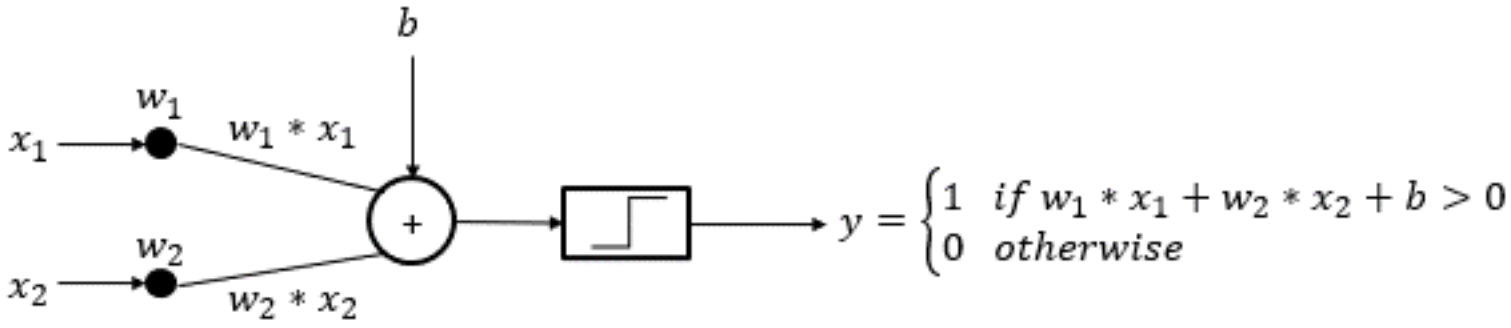
Iteration 5:

$w_1(5)$	$w_2(5)$	$b(5)$
0.5	0.5	0

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is **learning rate** and *t* is the **iteration**



AND gate:

x_1	x_2	y_{real}
0	0	0
1	0	0
0	1	0
1	1	1

$y = f(w_1(5) * x_1 + w_2(5) * x_2 + b(5))$
$y = f(0.5 * 1 + 0.5 * 0 + 0) = f(0.5) = 1$

$w_1(6) = w_1(5) + lr * (y_{real} - y) * x_1$
$w_1(6) = 0.5 + 0.5 * (-1) * 1 = 0$

$w_2(6) = w_2(5) + lr * (y_{real} - y) * x_2$
$w_2(6) = 0.5 + 0.5 * (-1) * 0 = 0.5$

$b(6) = b(5) + lr * (y_{real} - y)$
$b(6) = 0 + 0.5 * (-1) = -0.5$

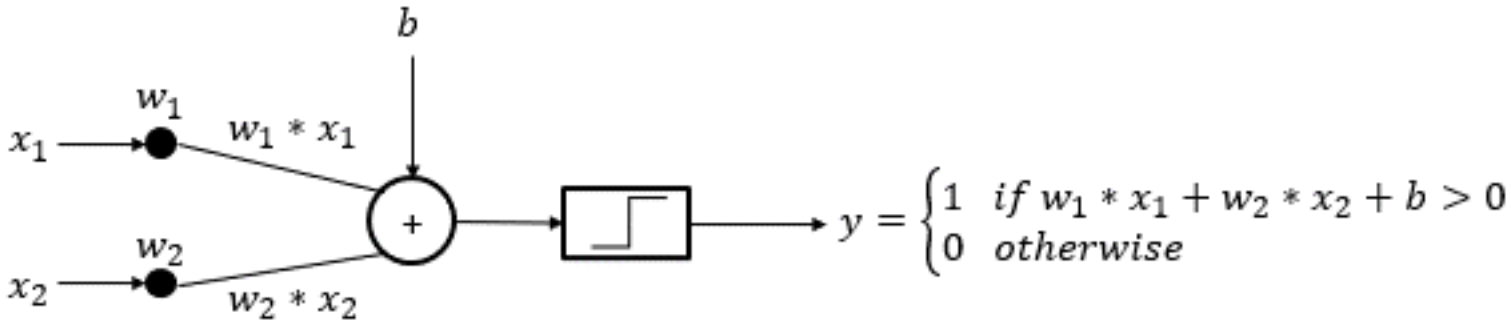
Iteration 5:

$w_1(5)$	$w_2(5)$	$b(5)$
0.5	0.5	0

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is **learning rate** and *t* is the **iteration**



AND gate:

x_1	x_2	y_{real}
0	0	0
1	0	0
0	1	0
1	1	1

$y = f(w_1(5) * x_1 + w_2(5) * x_2 + b(5))$
$y = f(0.5 * 1 + 0.5 * 0 + 0) = f(0.5) = 1$

$w_1(6) = w_1(5) + lr * (y_{real} - y) * x_1$
$w_1(6) = 0.5 + 0.5 * (-1) * 1 = 0$

$w_2(6) = w_2(5) + lr * (y_{real} - y) * x_2$
$w_2(6) = 0.5 + 0.5 * (-1) * 0 = 0.5$

$b(6) = b(5) + lr * (y_{real} - y)$
$b(6) = 0 + 0.5 * (-1) = -0.5$

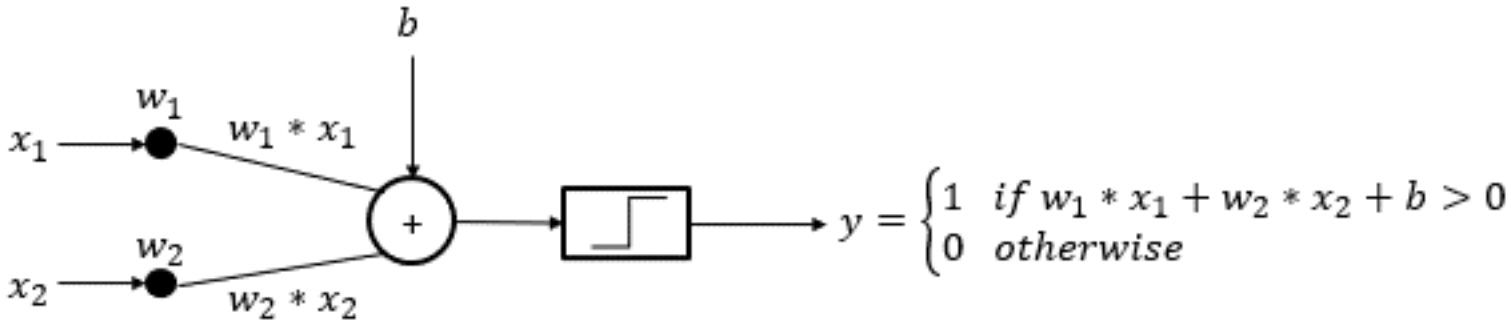
Iteration 6:

$w_1(6)$	$w_2(6)$	$b(6)$
0	0.5	-0.5

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is **learning rate** and *t* is the **iteration**



AND gate:

x_1	x_2	y_{real}
0	0	0
1	0	0
0	1	0
1	1	1

$y = f(w_1(6) * x_1 + w_2(6) * x_2 + b(6))$
$y = f(0 * 0 + 0.5 * 1 - 0.5) = f(0) = 0$

$w_1(7) = w_1(6) + lr * (y_{real} - y) * x_1$
$w_1(7) = 0 + 0.5 * 0 * 0 = 0$

$w_2(7) = w_2(6) + lr * (y_{real} - y) * x_2$
$w_2(7) = 0.5 + 0.5 * 0 * 1 = 0.5$

$b(7) = b(6) + lr * (y_{real} - y)$
$b(7) = -0.5 + 0.5 * 0 = -0.5$

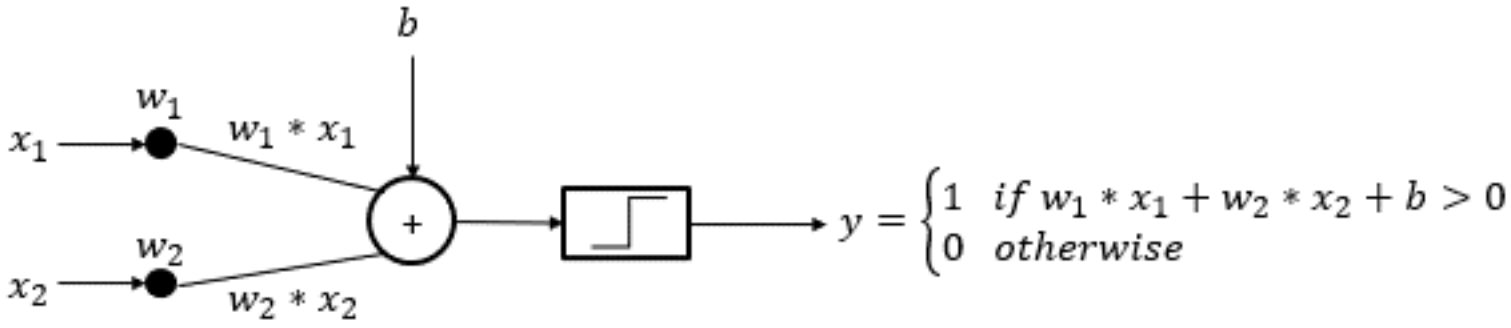
Iteration 6:

$w_1(6)$	$w_2(6)$	$b(6)$
0	0.5	-0.5

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is **learning rate** and *t* is the **iteration**



AND gate:

x_1	x_2	y_{real}
0	0	0
1	0	0
0	1	0
1	1	1

$y = f(w_1(6) * x_1 + w_2(6) * x_2 + b(6))$
$y = f(0 * 0 + 0.5 * 1 - 0.5) = f(0) = 0$

$w_1(7) = w_1(6) + lr * (y_{real} - y) * x_1$
$w_1(7) = 0 + 0.5 * 0 * 0 = 0$

$w_2(7) = w_2(6) + lr * (y_{real} - y) * x_2$
$w_2(7) = 0.5 + 0.5 * 0 * 1 = 0.5$

$b(7) = b(6) + lr * (y_{real} - y)$
$b(7) = -0.5 + 0.5 * 0 = -0.5$

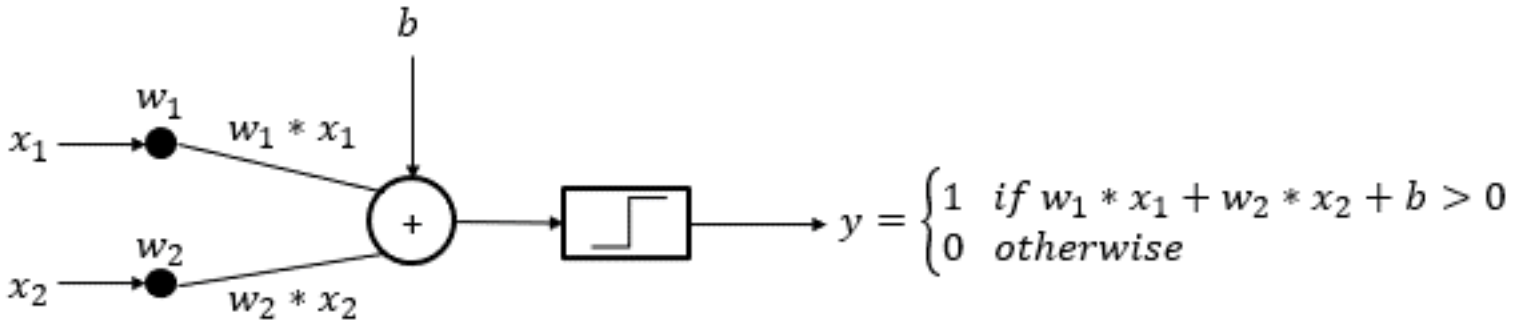
Iteration 7:

$w_1(7)$	$w_2(7)$	$b(7)$
0	0.5	-0.5

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is **learning rate** and t is the **iteration**



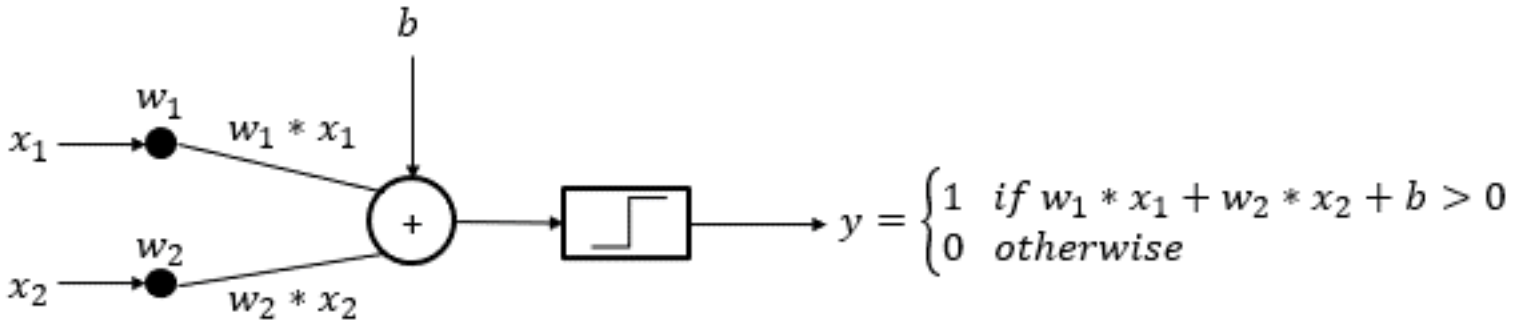
XOR gate:

x_1	x_2	y_{real}
0	0	0
1	0	1
0	1	1
1	1	0

PERCEPTRON (ROSENBLATT, 1958)

Training algorithm:

- Weight **initialization**
- Until **convergence**:
  1. **Compute** y
  2. **Update** weights such that:
     $w_i(t+1) = w_i(t) + lr * (y_{real} - y) * x_i$,
     where *lr* is **learning rate** and t is the **iteration**

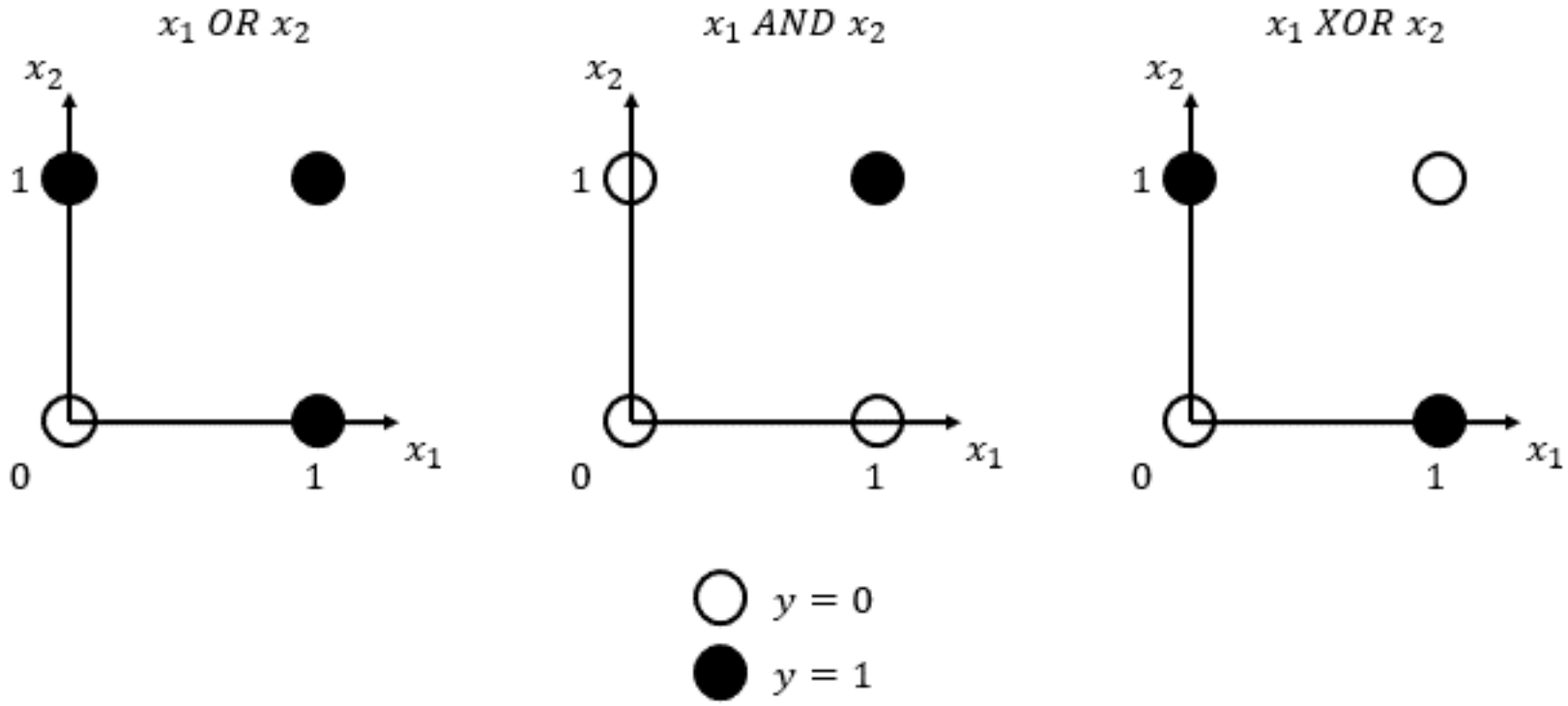


XOR gate:

x_1	x_2	y_{real}
0	0	0
1	0	1
0	1	1
1	1	0

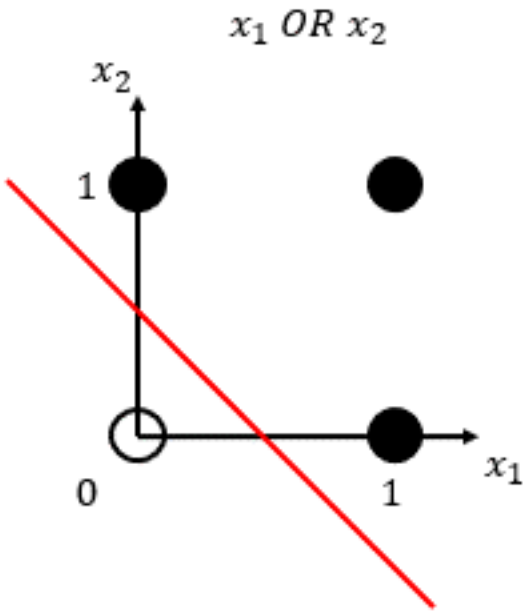
Training does not converge

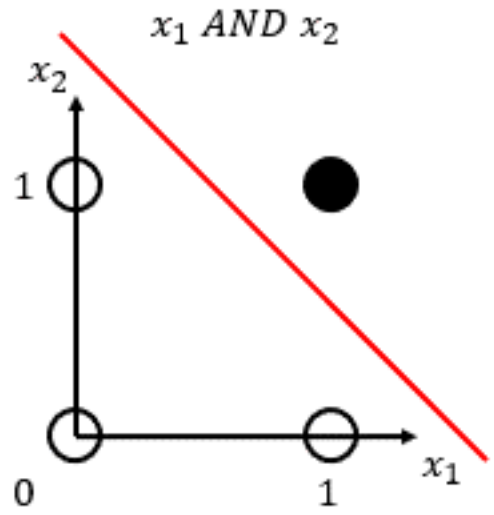
PERCEPTRON (ROSENBLATT, 1958)

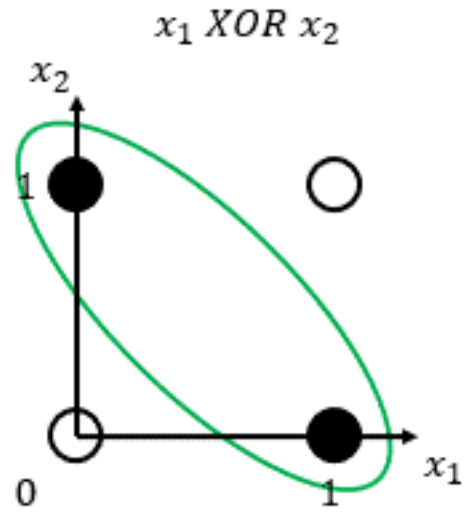


PERCEPTRON (ROSENBLATT, 1958)

Linearly separable





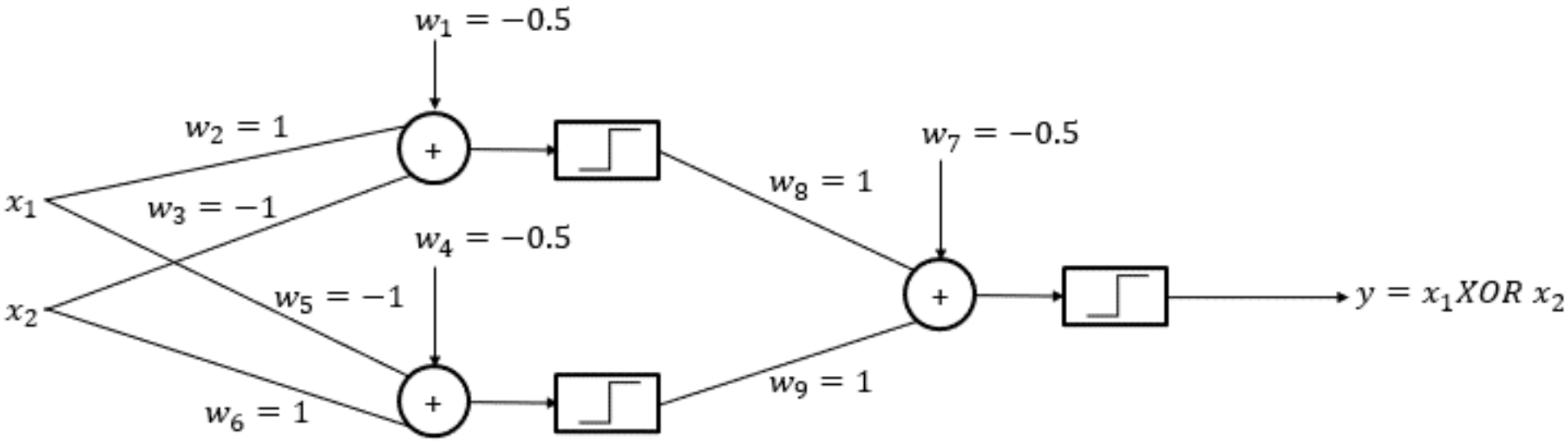


$y = 0$

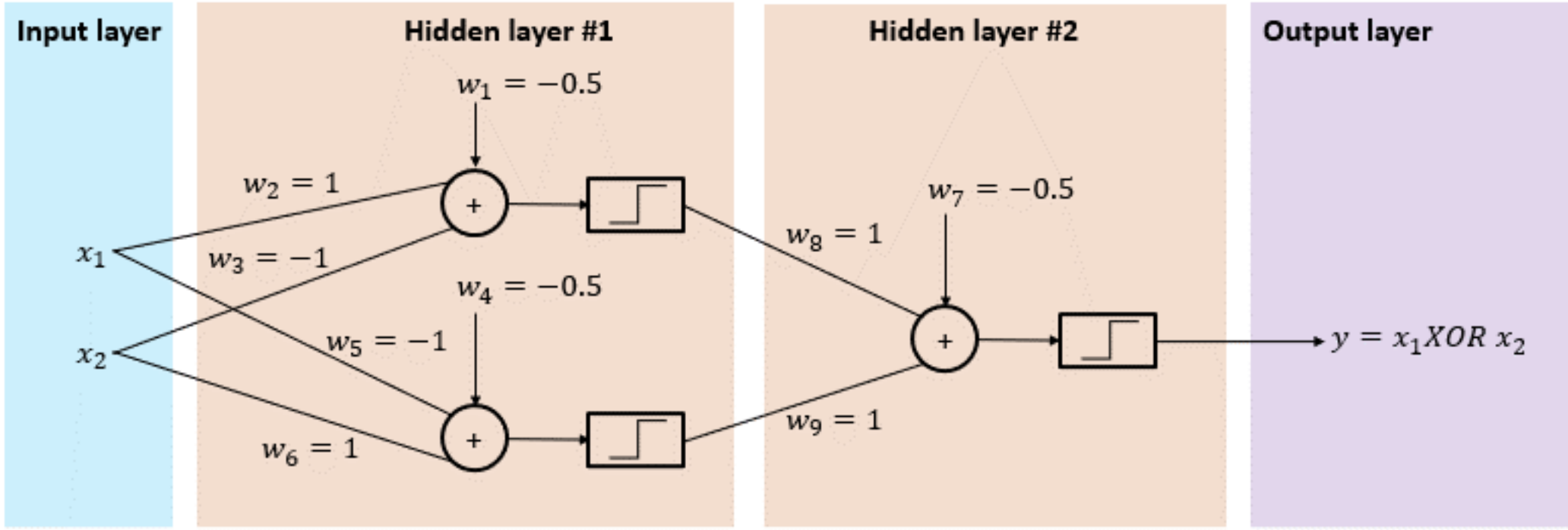
$y = 1$

Non linearly separable

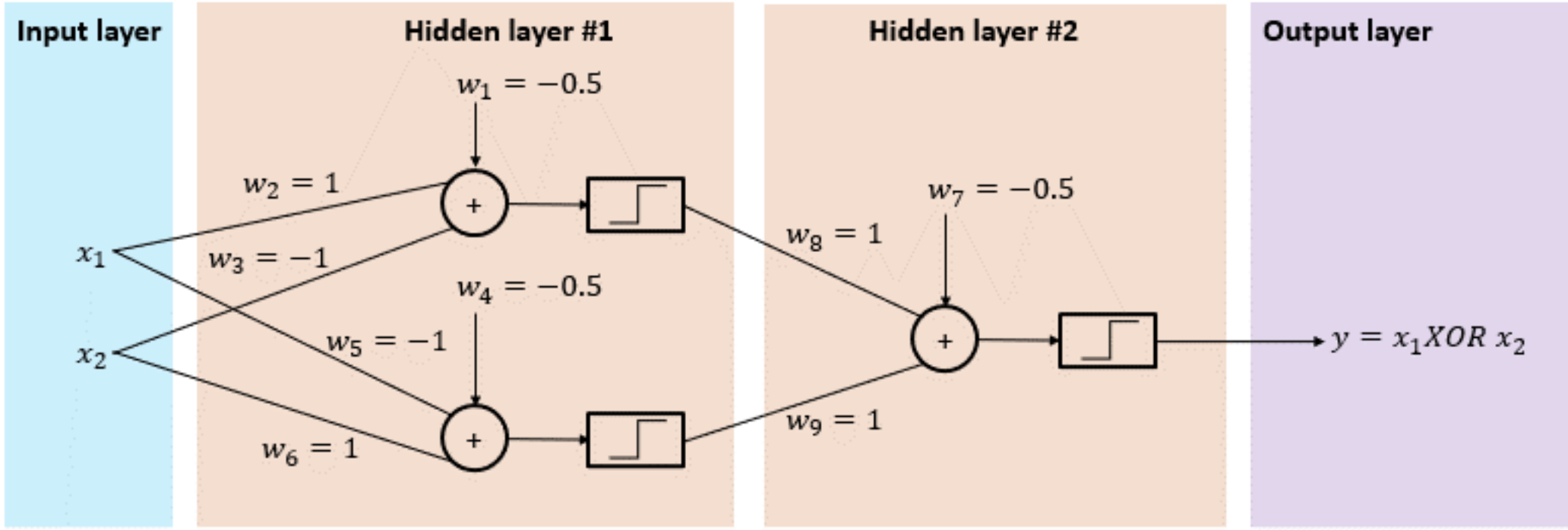
PERCEPTRON (ROSENBLATT, 1958)



PERCEPTRON (ROSENBLATT, 1958)

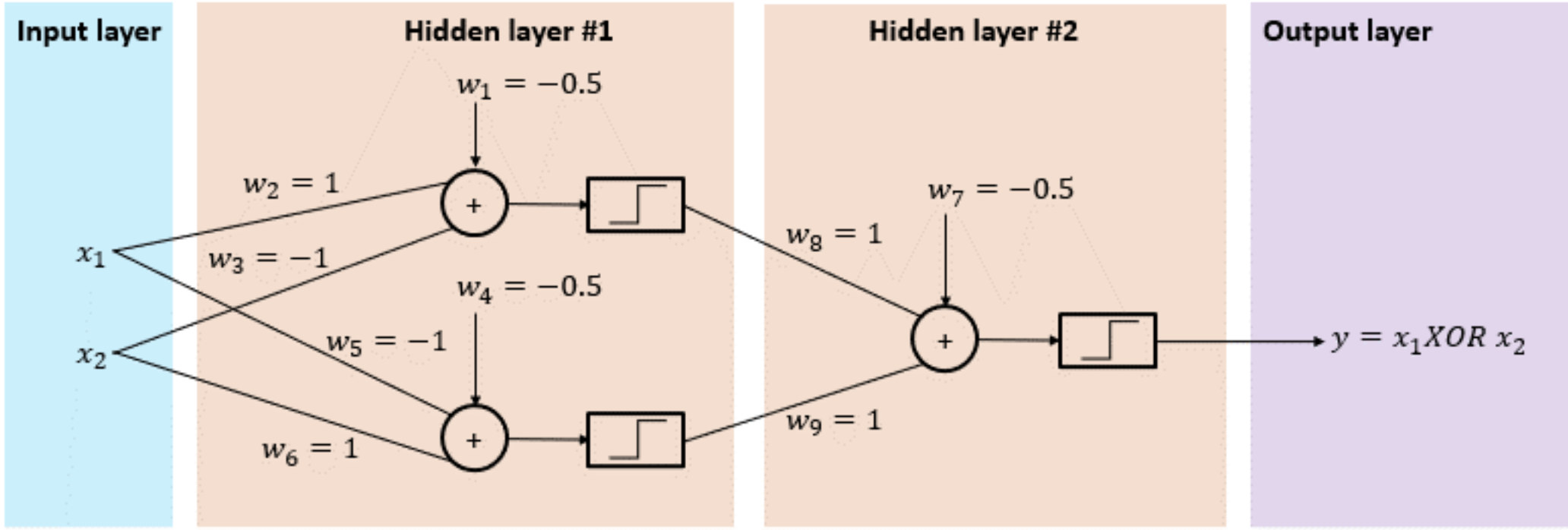


PERCEPTRON (ROSENBLATT, 1958)



2-hidden layer perceptron can **solve non linear** problems

PERCEPTRON (ROSENBLATT, 1958)

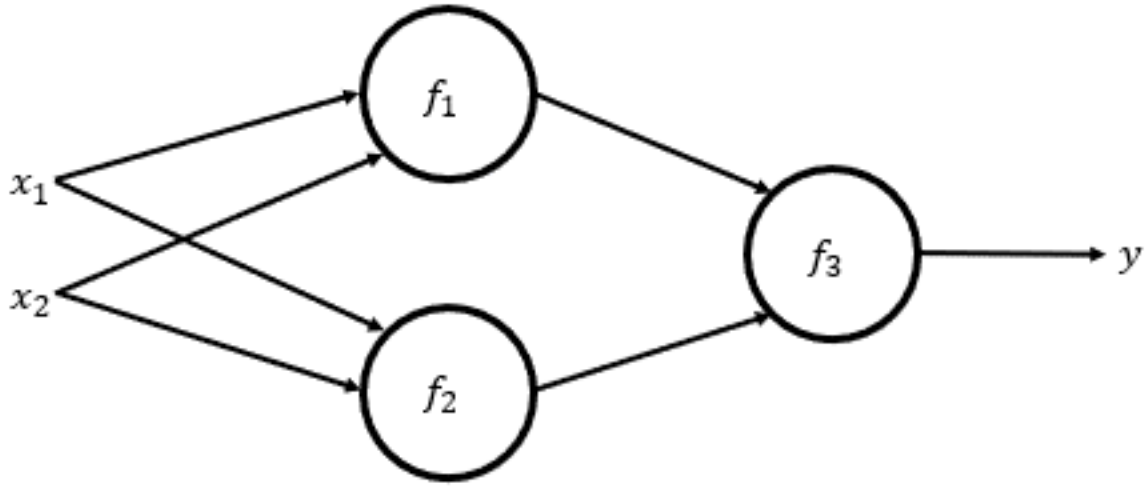


2-hidden layer perceptron can **solve non linear** problems

How to train such network?

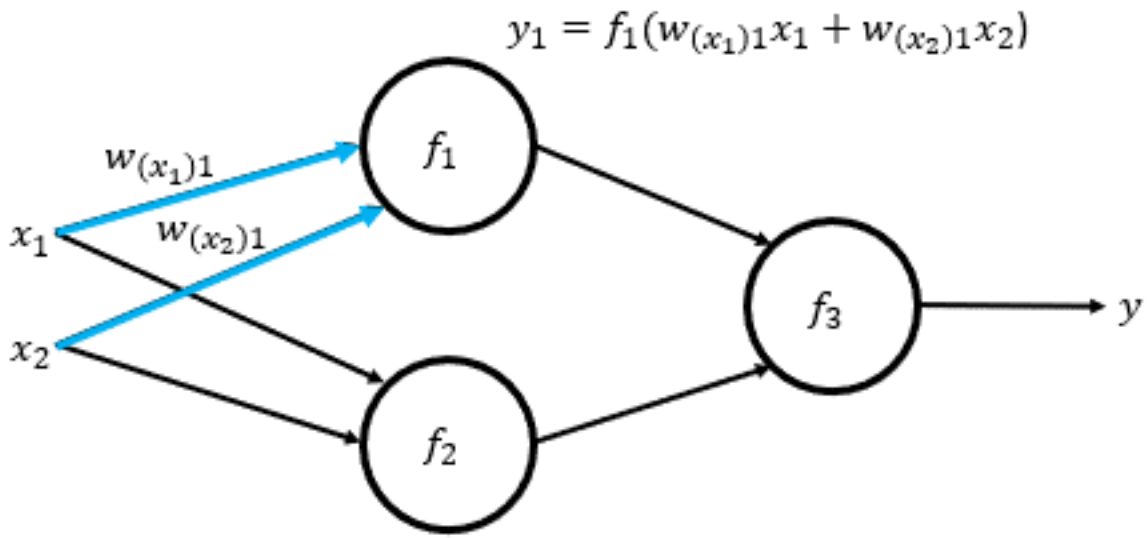
BACKPROPAGATION

Forward propagation:



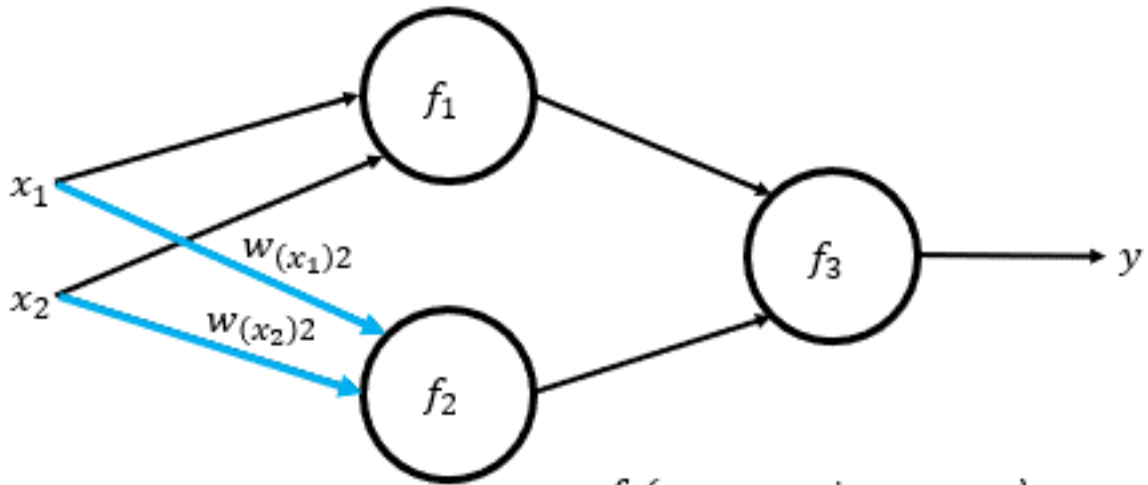
BACKPROPAGATION

Forward propagation:



BACKPROPAGATION

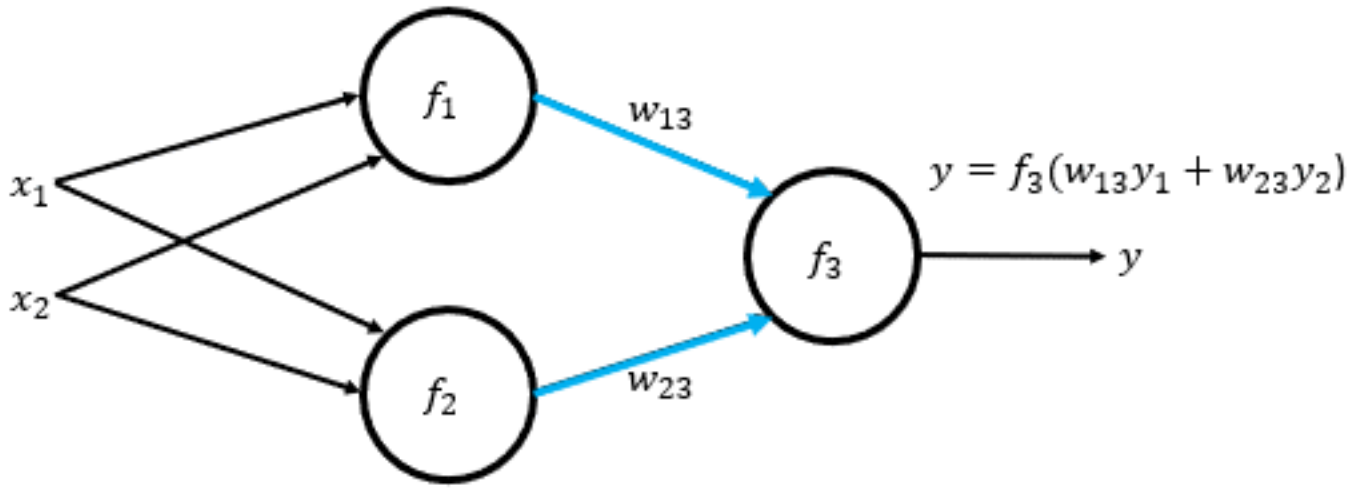
Forward propagation:



$$y_2 = f_2(w_{(x_1)2}x_1 + w_{(x_2)2}x_2)$$

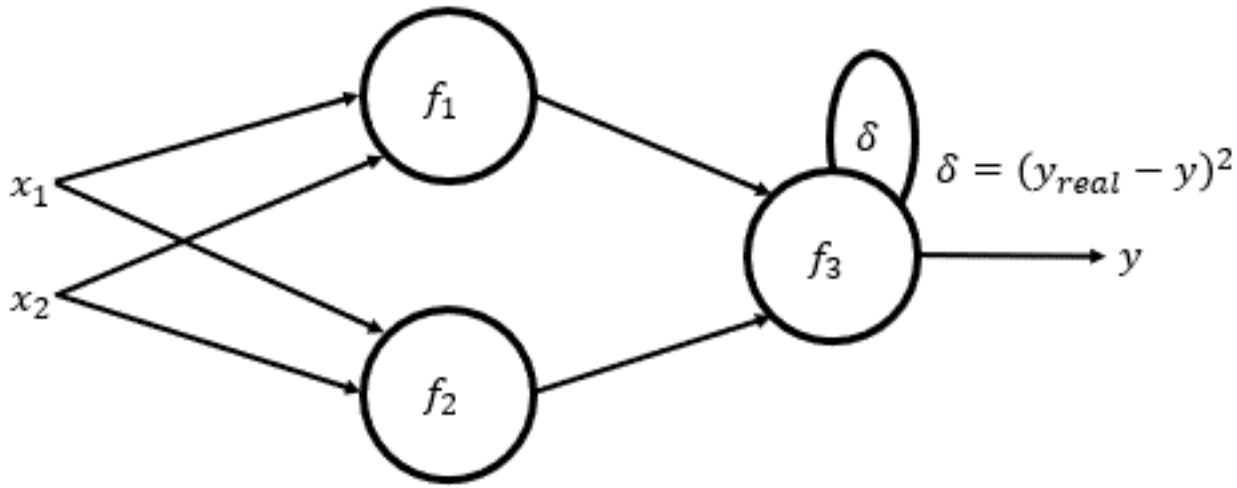
BACKPROPAGATION

Forward propagation:



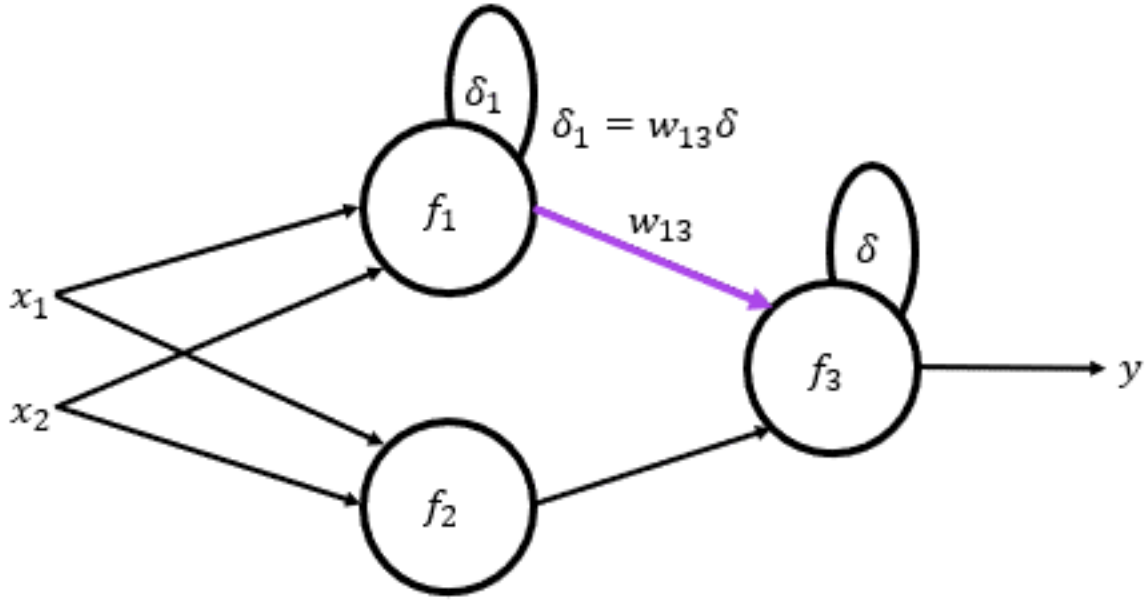
BACKPROPAGATION

Forward propagation:



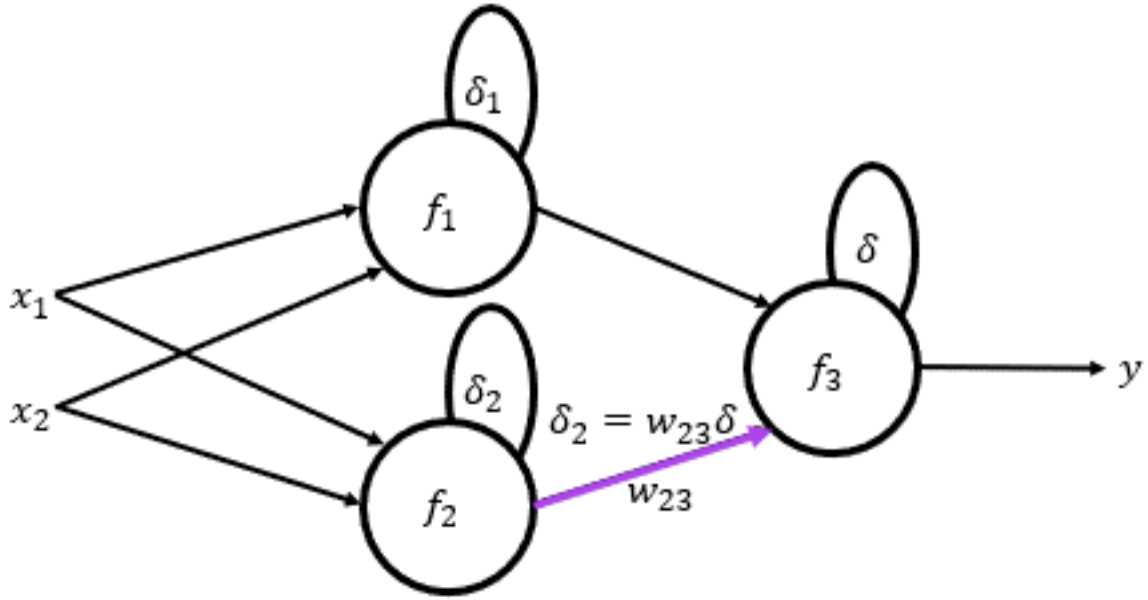
BACKPROPAGATION

Backpropagation:



BACKPROPAGATION

Backpropagation:

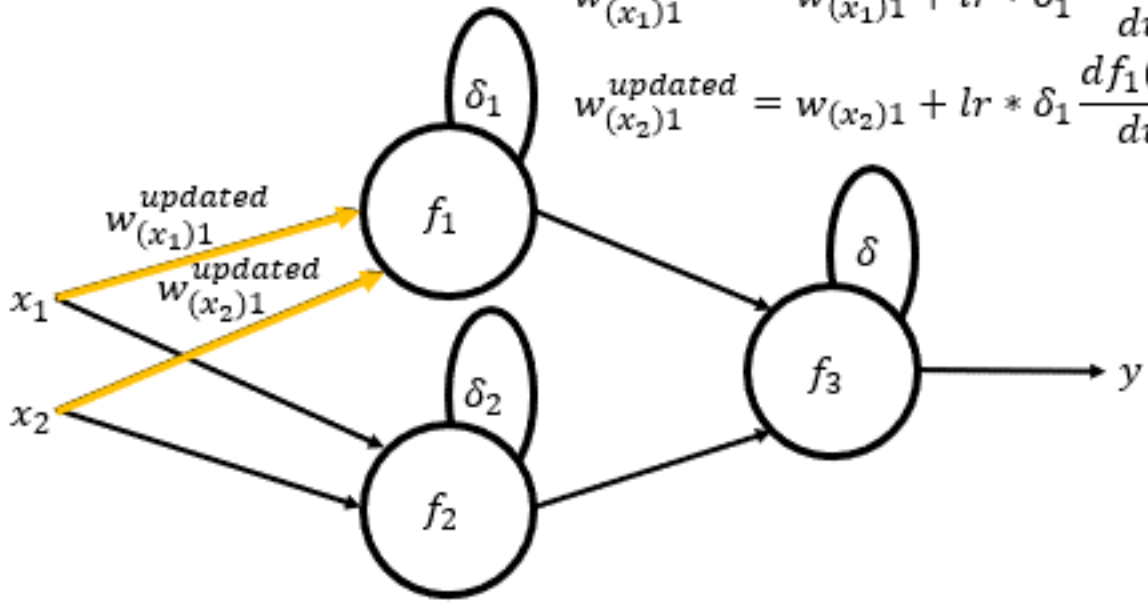


BACKPROPAGATION

Backpropagation:

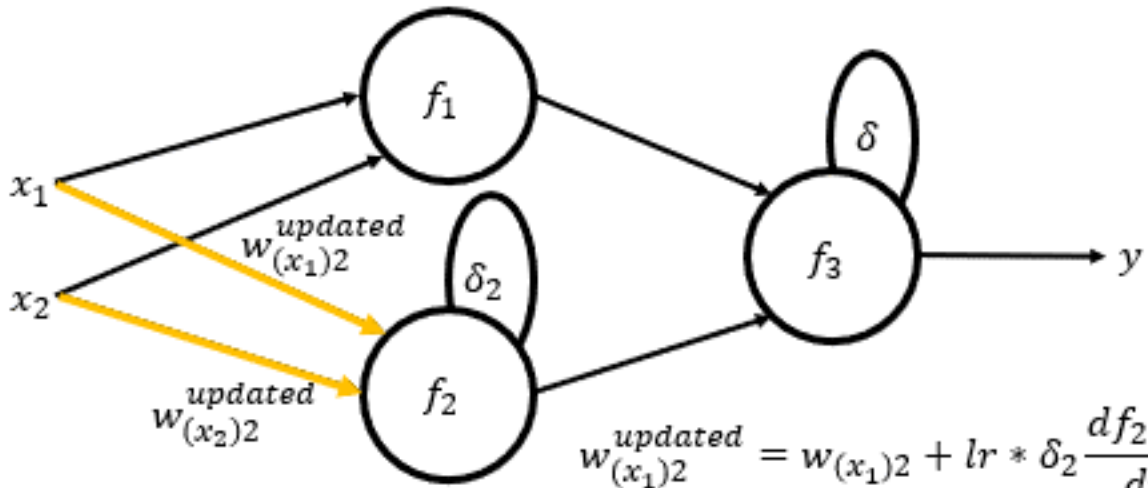
$$w_{(x_1)1}^{updated} = w_{(x_1)1} + lr * \delta_1 \frac{df_1(u)}{du} * x_1$$

$$w_{(x_2)1}^{updated} = w_{(x_2)1} + lr * \delta_1 \frac{df_1(u)}{du} * x_2$$



BACKPROPAGATION

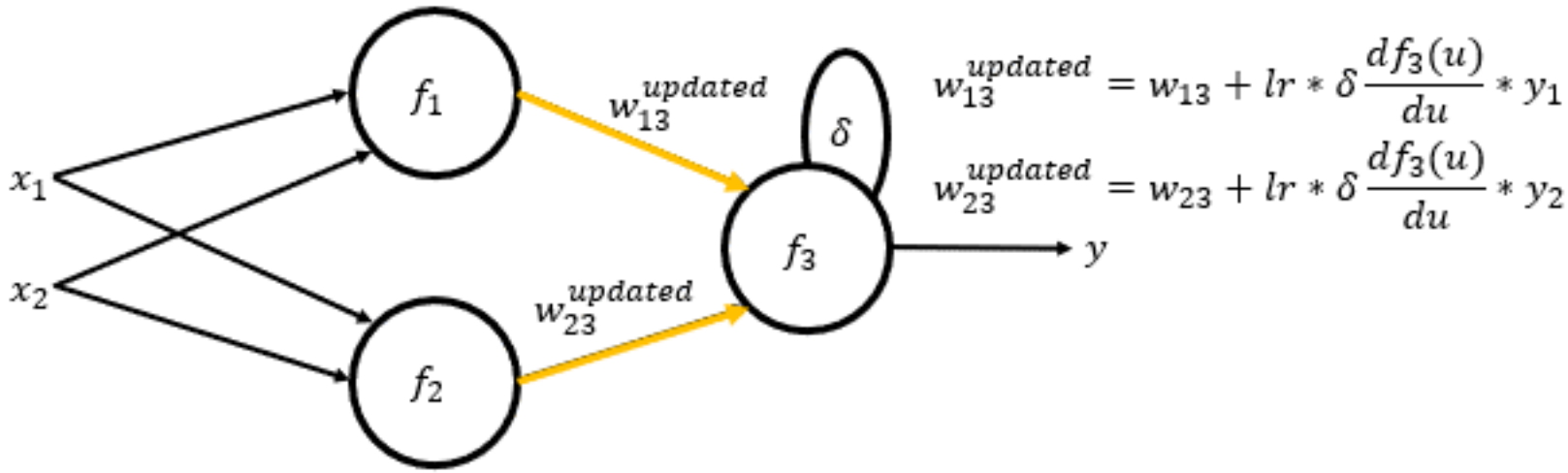
Backpropagation:



$$w_{(x_1)2}^{updated} = w_{(x_1)2} + lr * \delta_2 \frac{df_2(u)}{du} * x_1$$

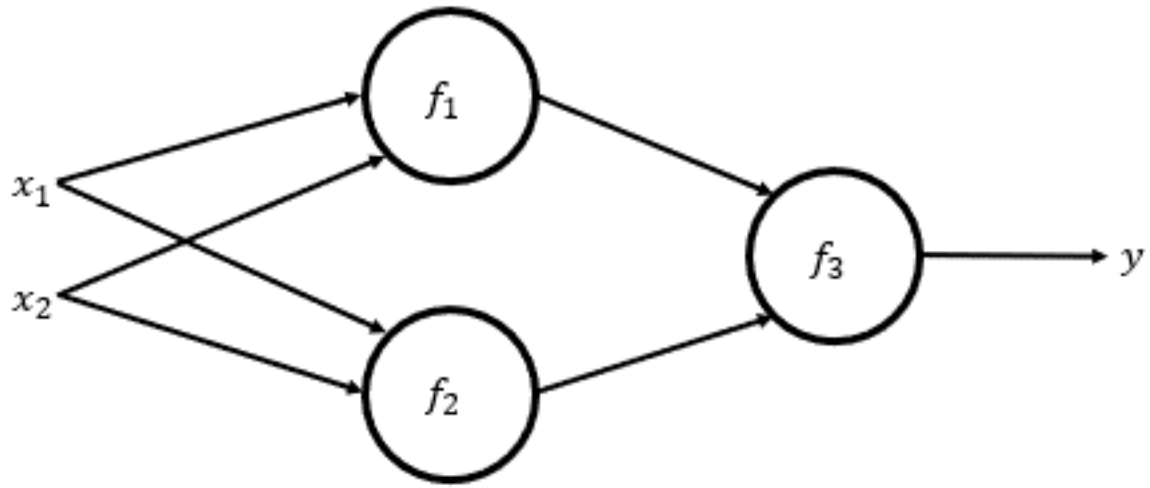
$$w_{(x_2)2}^{updated} = w_{(x_2)2} + lr * \delta_2 \frac{df_2(u)}{du} * x_2$$

Backpropagation:



BACKPROPAGATION

Backpropagation:



Critical point: $\frac{df(u)}{du}$ is known while the Heaviside function is **not differentiable**

Examples of **activation functions**: sigmoid, hyperbolic tangent, rectified linear unit (ReLU), softmax

OPTIMIZATION

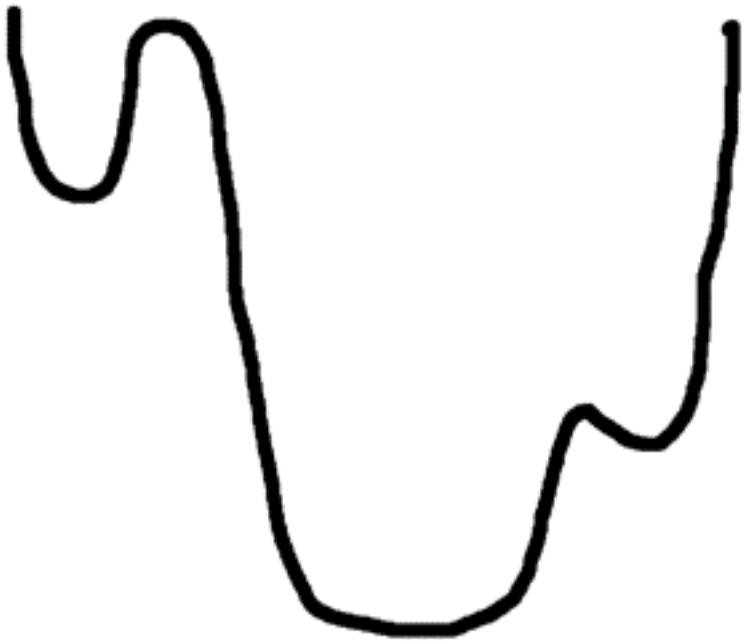
- The goal is to **minimize the loss (cost) function** $Loss(y_{real}, y)$ (Mean Squared, error, Cross-Entropy, …)

OPTIMIZATION

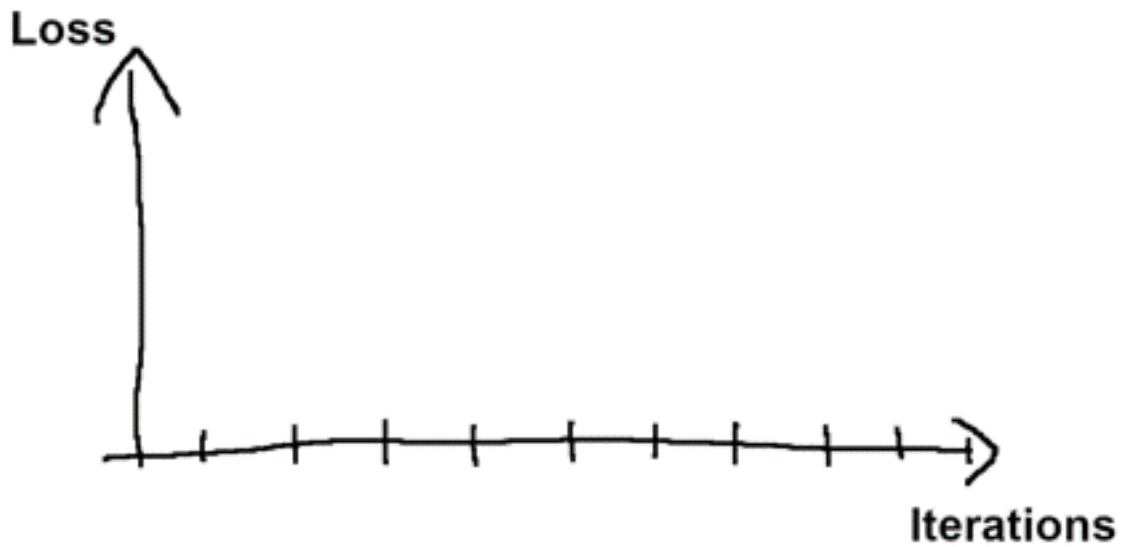
- The goal is to **minimize the loss (cost) function** $Loss(y_{real}, y)$ (Mean Squared, error, Cross-Entropy, …)
- This is achieved via a **gradient descent** (stochastic gradient descent, RMSprop, ADAM, …)

OPTIMIZATION

- The goal is to **minimize the loss (cost) function** $Loss(y_{real}, y)$ (Mean Squared, error, Cross-Entropy, …)
- This is achieved via a **gradient descent** (stochastic gradient descent, RMSprop, ADAM, …)

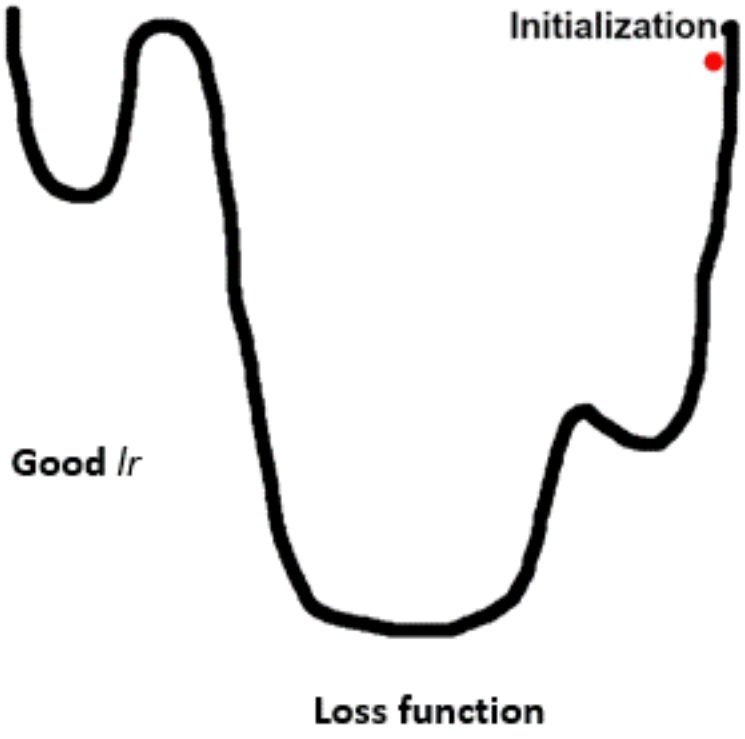


Loss function



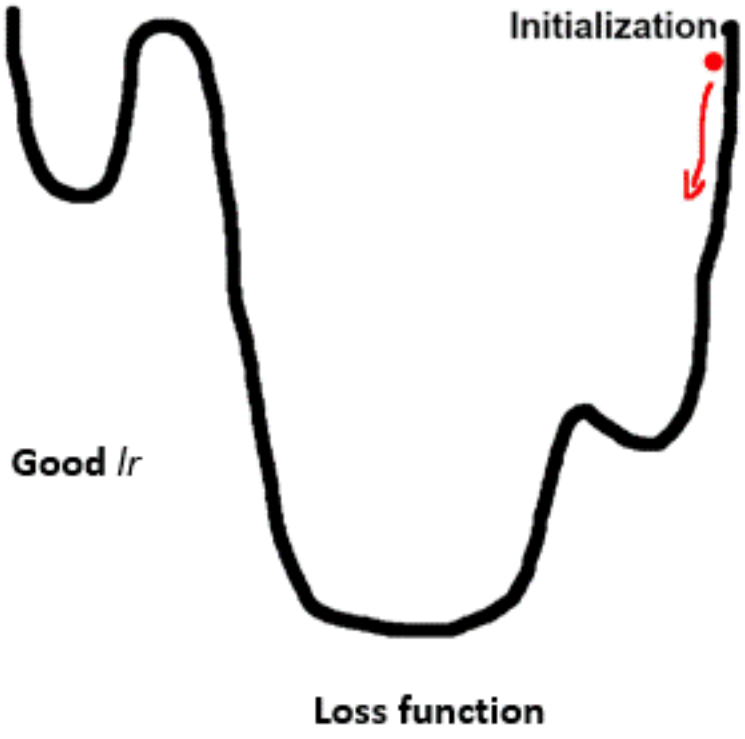
OPTIMIZATION

- The goal is to **minimize the loss (cost) function** $Loss(y_{real}, y)$ (Mean Squared, error, Cross-Entropy, …)
- This is achieved via a **gradient descent** (stochastic gradient descent, RMSprop, ADAM, …)



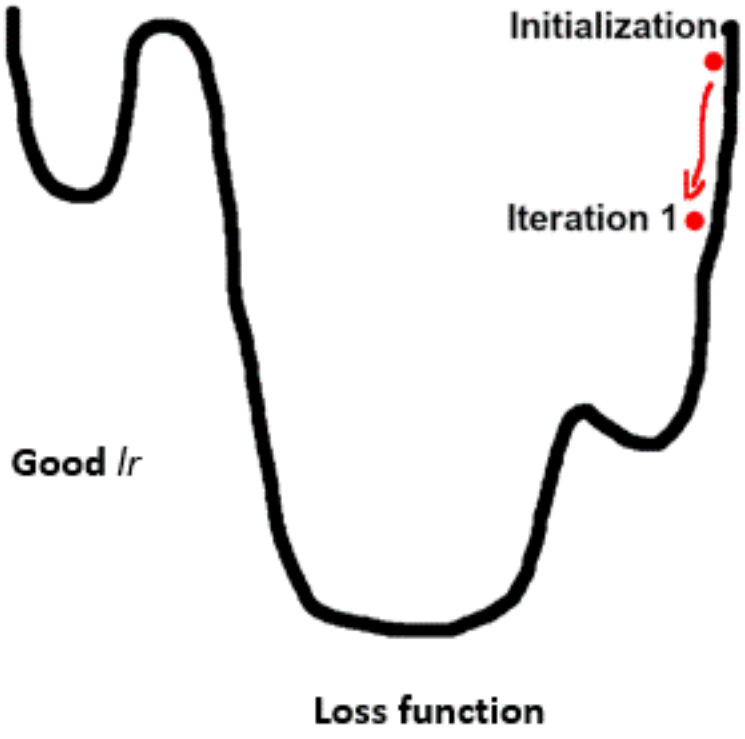
OPTIMIZATION

- The goal is to **minimize the loss (cost) function** $Loss(y_{real}, y)$ (Mean Squared, error, Cross-Entropy, …)
- This is achieved via a **gradient descent** (stochastic gradient descent, RMSprop, ADAM, …)



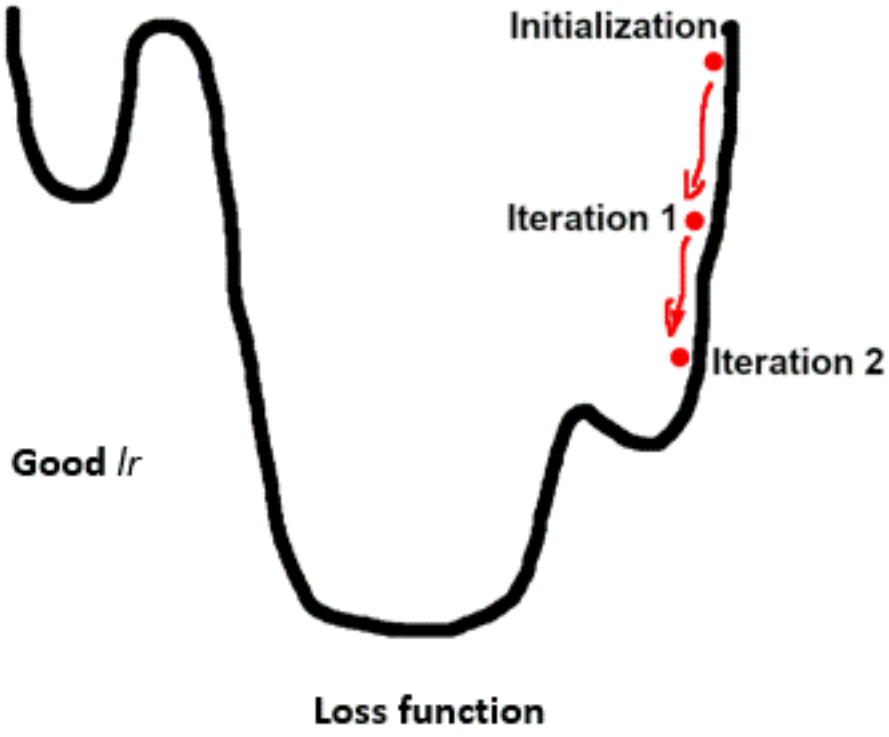
OPTIMIZATION

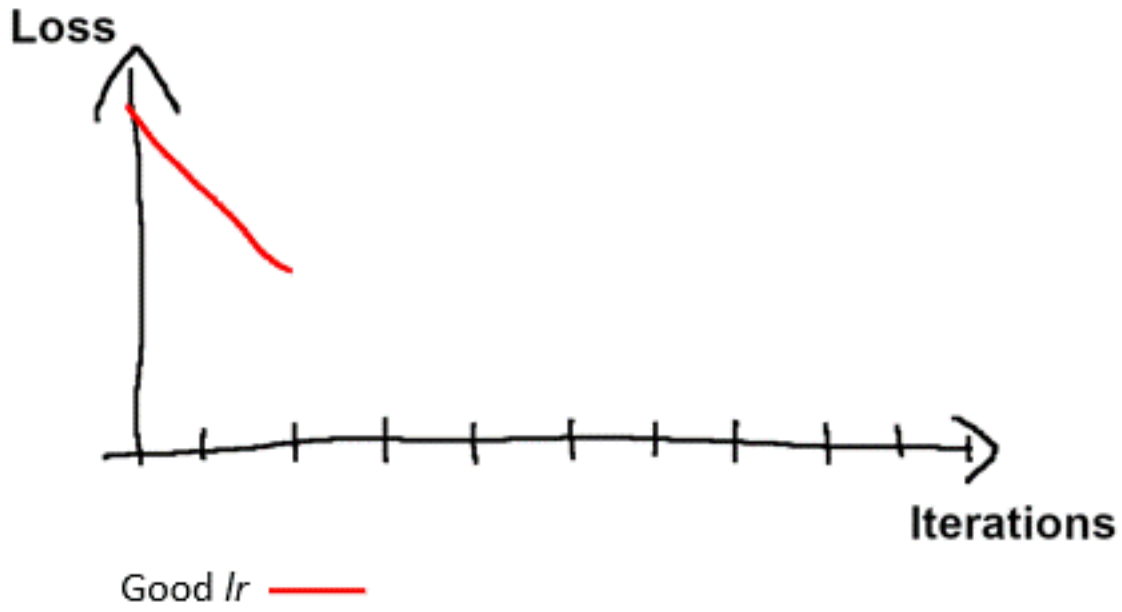
- The goal is to **minimize the loss (cost) function** $Loss(y_{real}, y)$ (Mean Squared, error, Cross-Entropy, …)
- This is achieved via a **gradient descent** (stochastic gradient descent, RMSprop, ADAM, …)



OPTIMIZATION

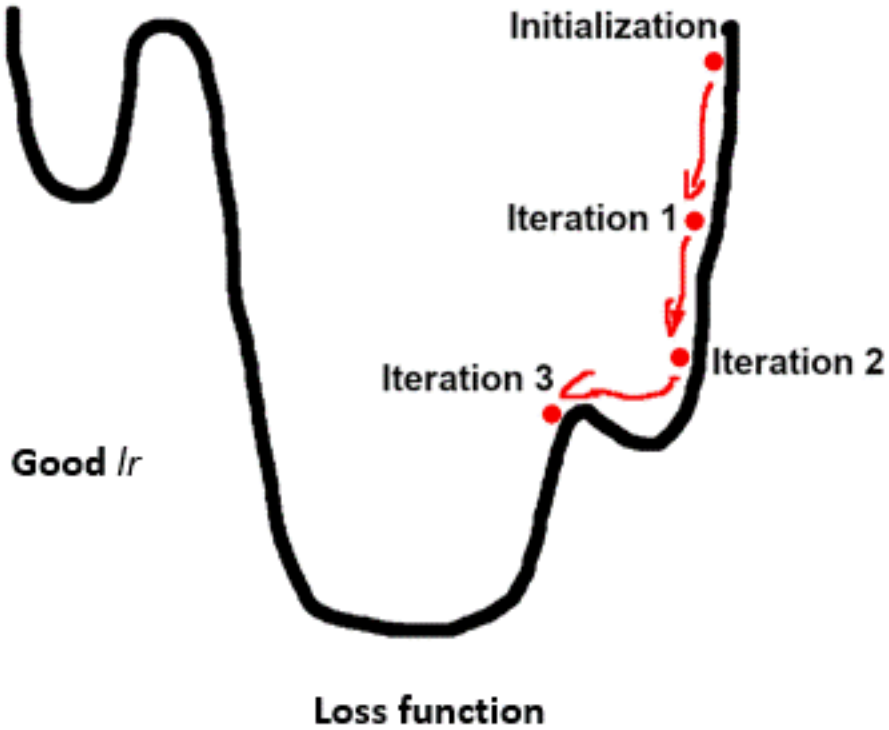
- The goal is to **minimize the loss (cost) function** $Loss(y_{real}, y)$ (Mean Squared, error, Cross-Entropy, …)
- This is achieved via a **gradient descent** (stochastic gradient descent, RMSprop, ADAM, …)

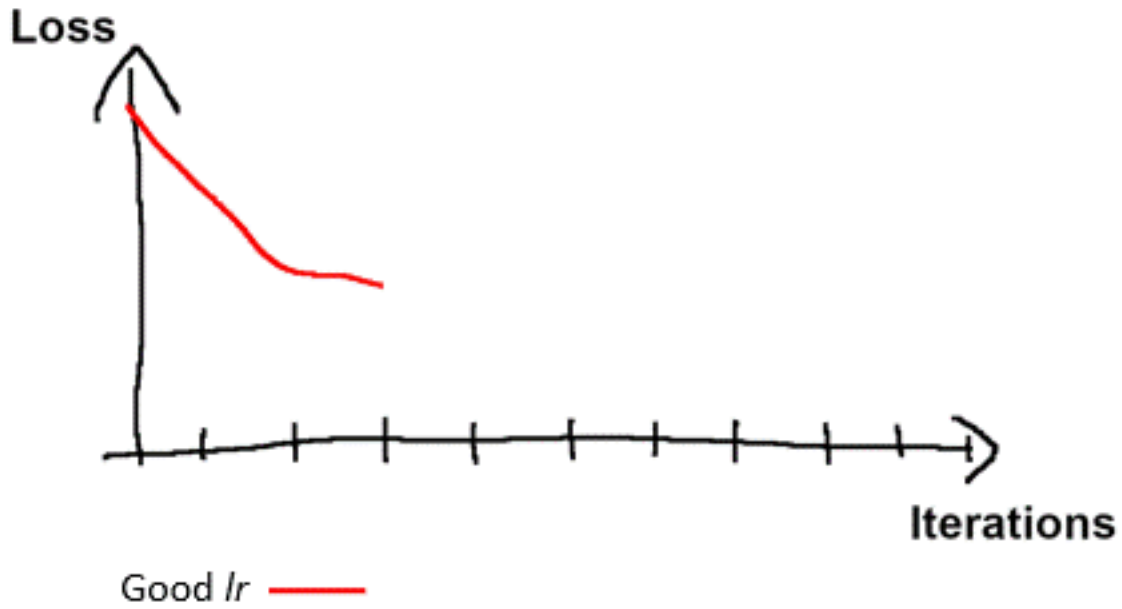




OPTIMIZATION

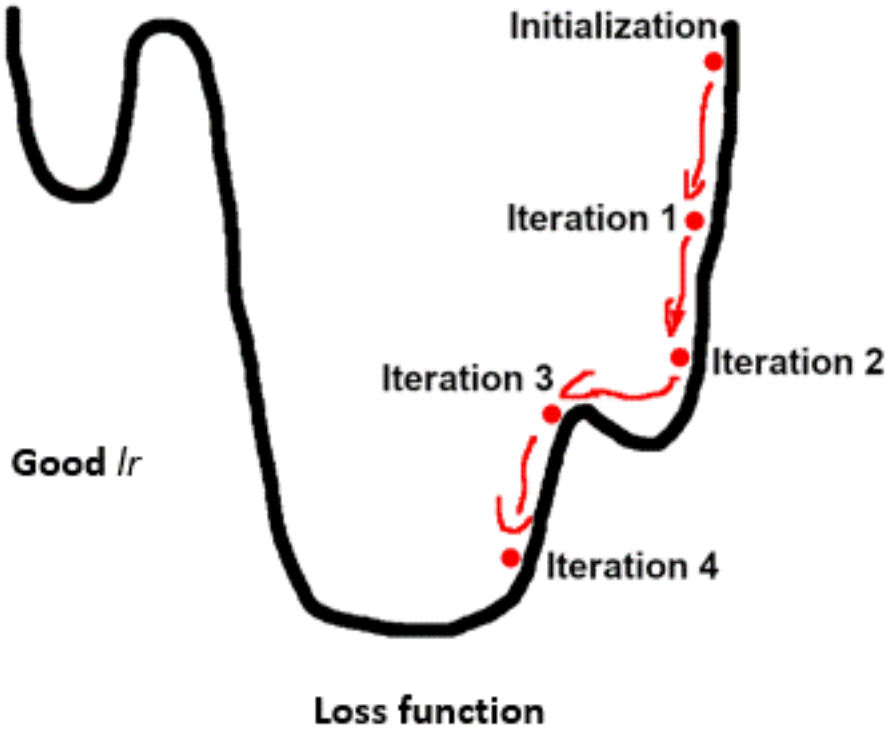
- The goal is to **minimize the loss (cost) function** $Loss(y_{real}, y)$ (Mean Squared, error, Cross-Entropy, …)
- This is achieved via a **gradient descent** (stochastic gradient descent, RMSprop, ADAM, …)

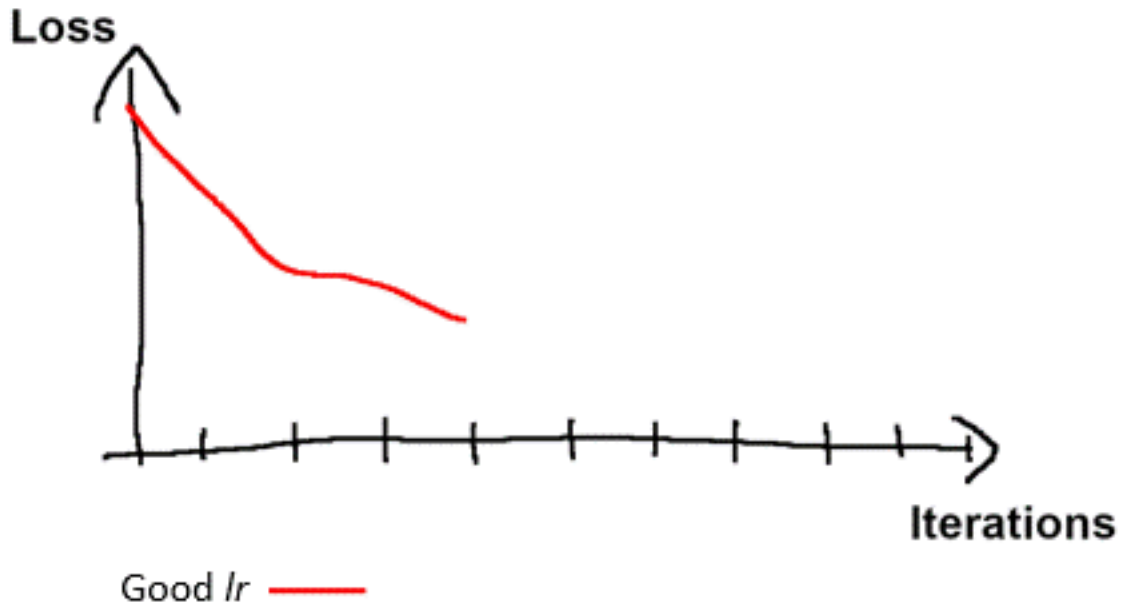




OPTIMIZATION

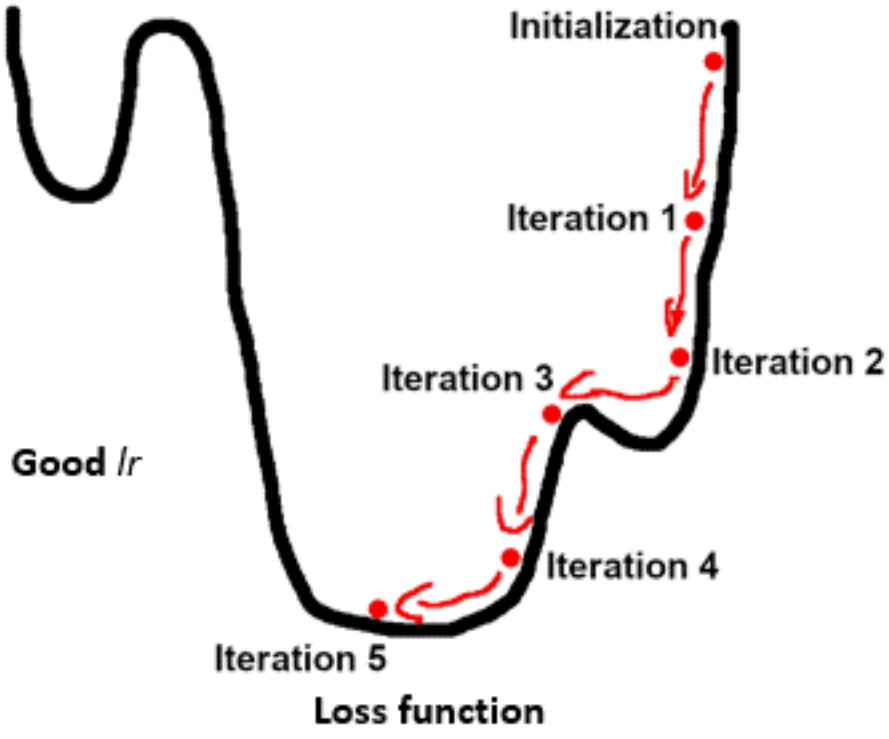
- The goal is to **minimize the loss (cost) function** $Loss(y_{real}, y)$ (Mean Squared, error, Cross-Entropy, …)
- This is achieved via a **gradient descent** (stochastic gradient descent, RMSprop, ADAM, …)

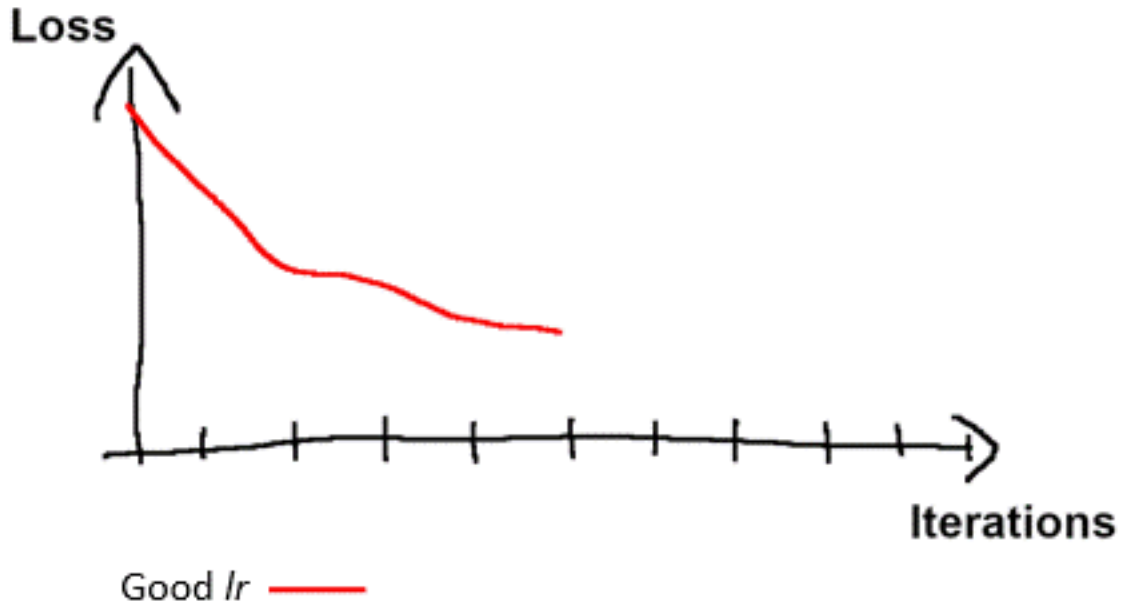




OPTIMIZATION

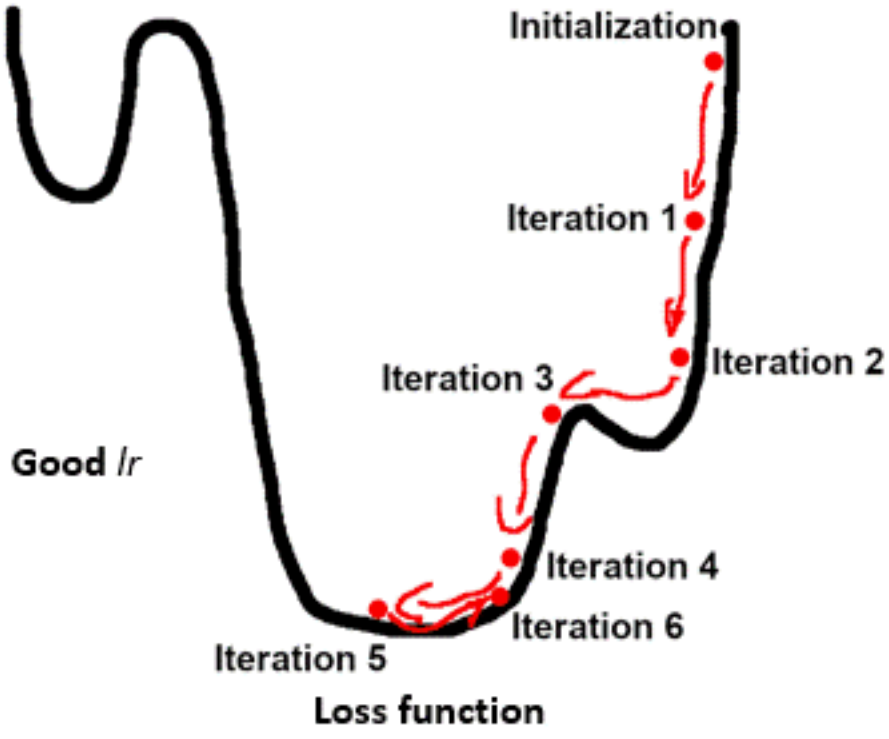
- The goal is to **minimize the loss (cost) function** $Loss(y_{real}, y)$ (Mean Squared, error, Cross-Entropy, …)
- This is achieved via a **gradient descent** (stochastic gradient descent, RMSprop, ADAM, …)

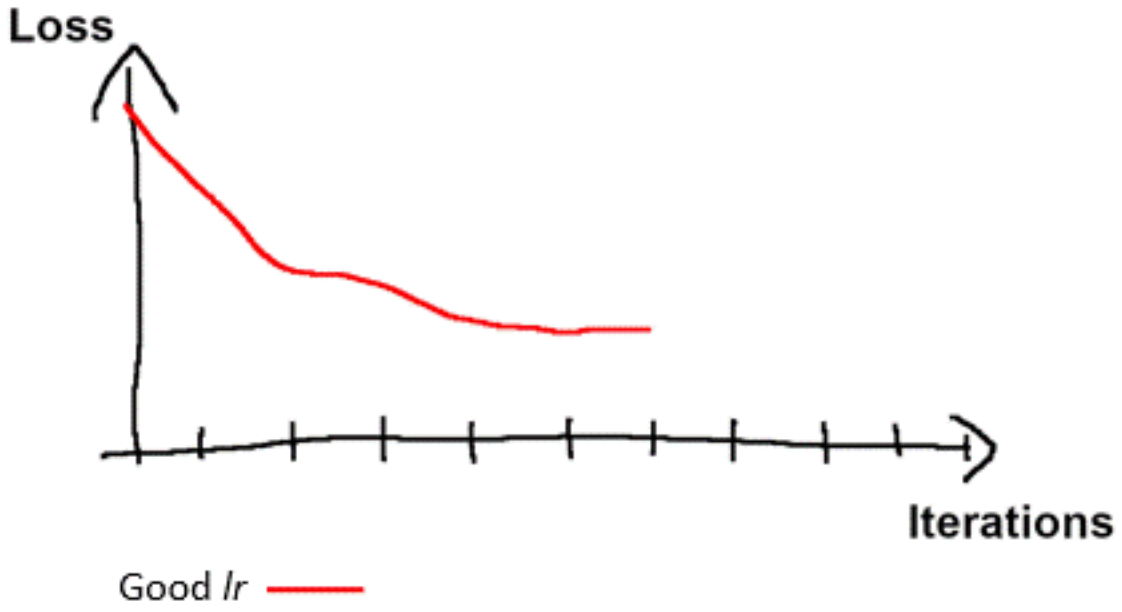




OPTIMIZATION

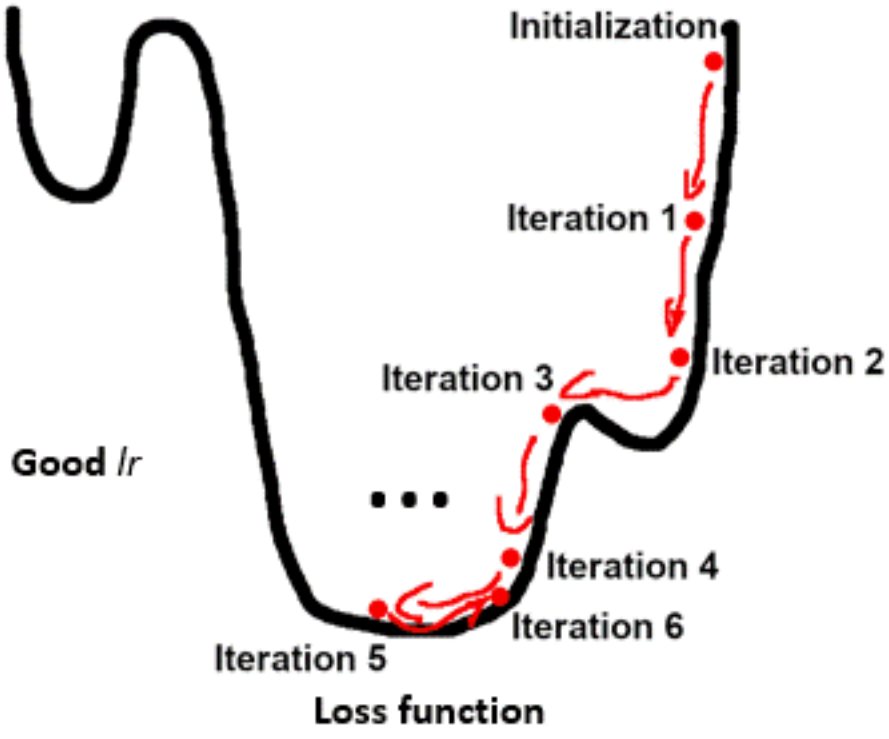
- The goal is to **minimize the loss (cost) function** $Loss(y_{real}, y)$ (Mean Squared, error, Cross-Entropy, …)
- This is achieved via a **gradient descent** (stochastic gradient descent, RMSprop, ADAM, …)

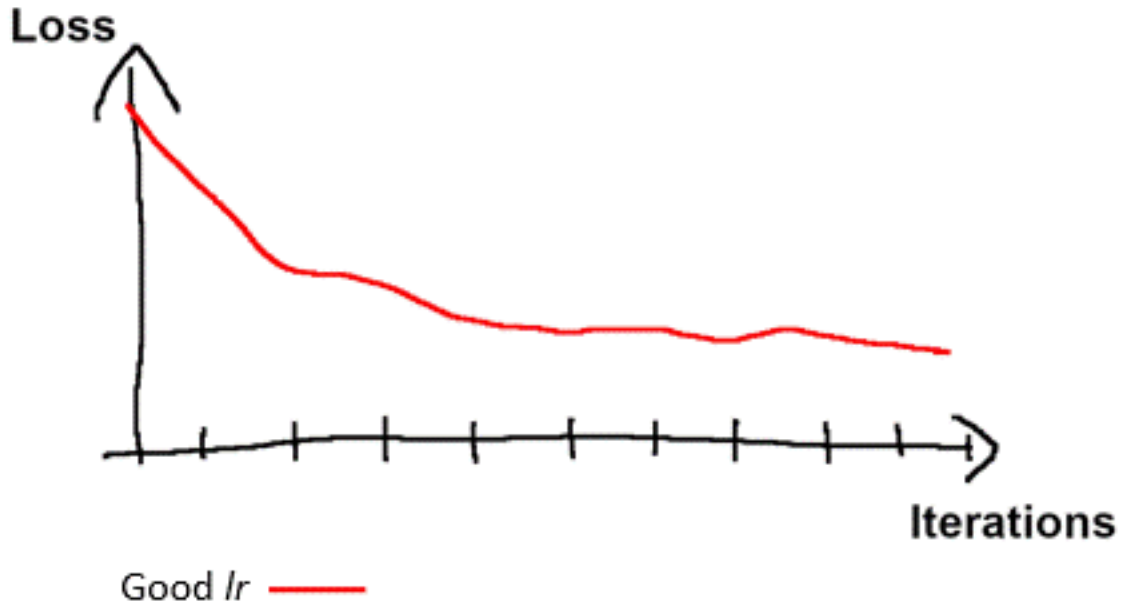




OPTIMIZATION

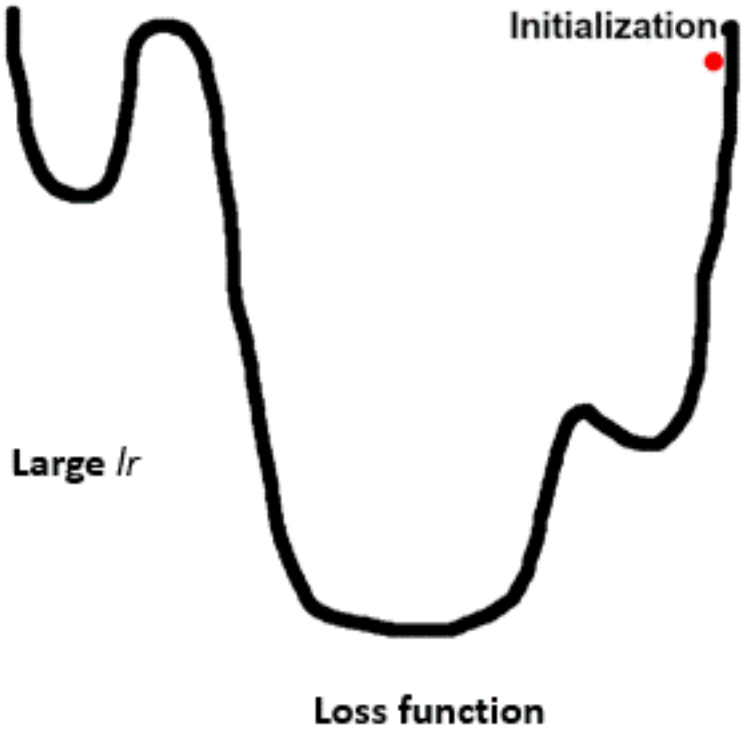
- The goal is to **minimize the loss (cost) function** $Loss(y_{real}, y)$ (Mean Squared, error, Cross-Entropy, …)
- This is achieved via a **gradient descent** (stochastic gradient descent, RMSprop, ADAM, …)

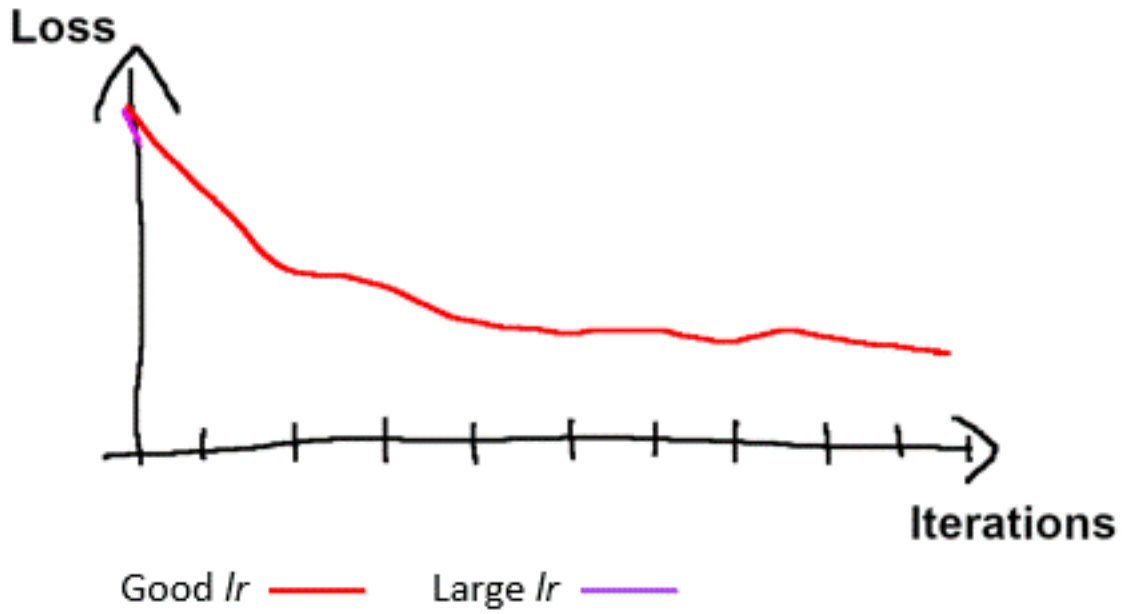




OPTIMIZATION

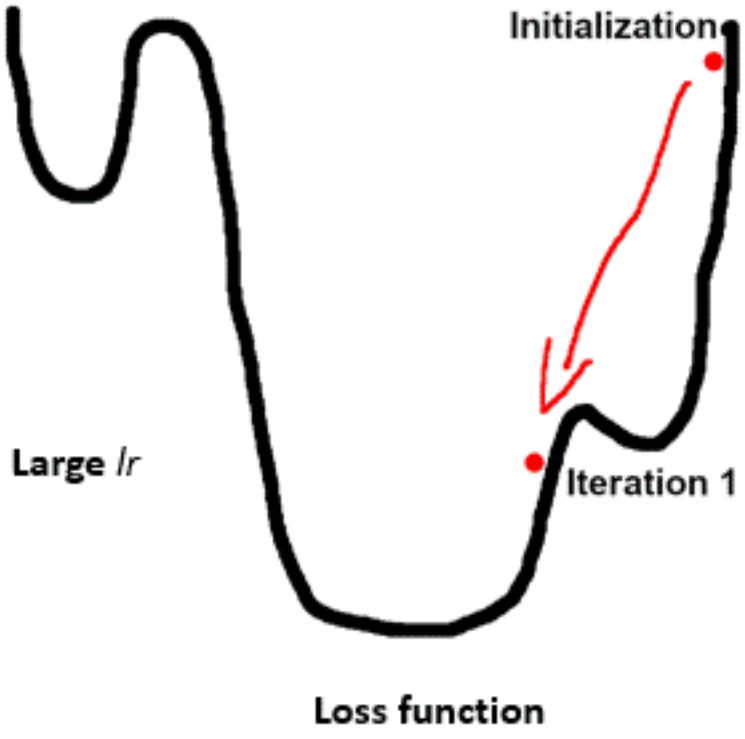
- The goal is to **minimize the loss (cost) function** $Loss(y_{real}, y)$ (Mean Squared, error, Cross-Entropy, …)
- This is achieved via a **gradient descent** (stochastic gradient descent, RMSprop, ADAM, …)

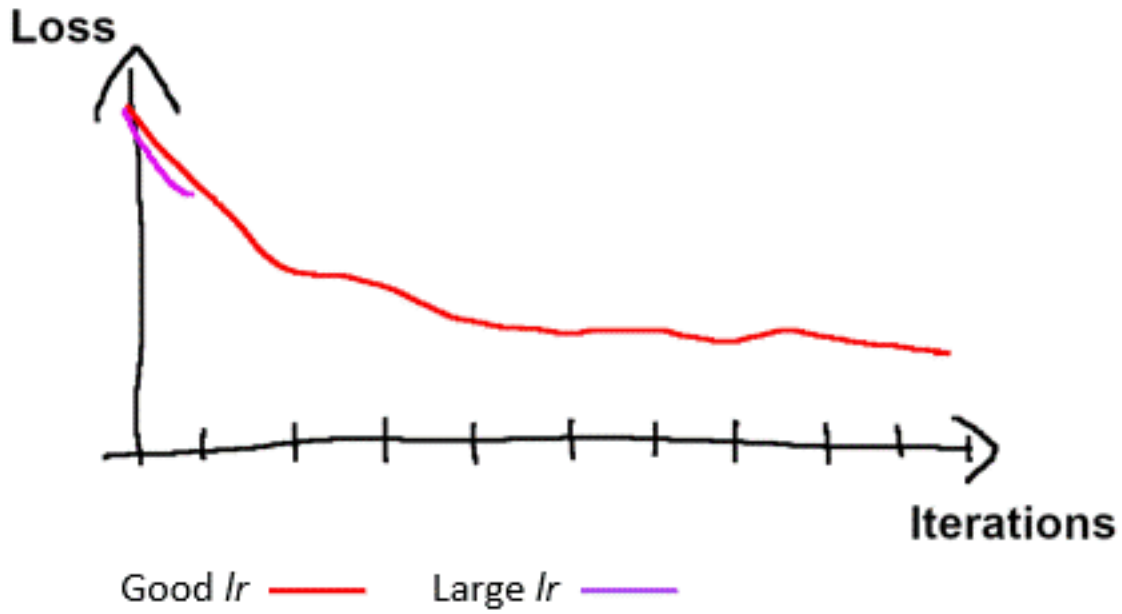




OPTIMIZATION

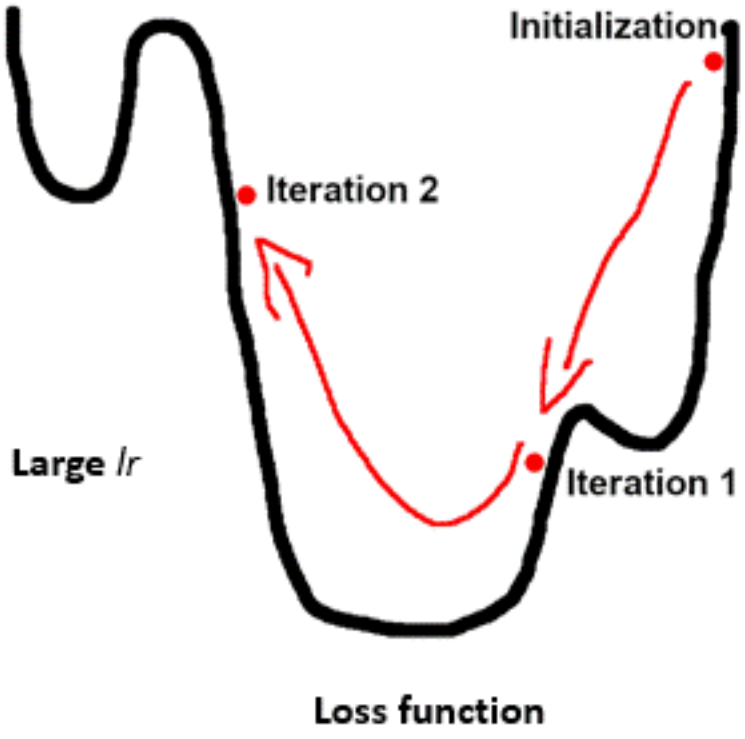
- The goal is to **minimize the loss (cost) function** $Loss(y_{real}, y)$ (Mean Squared, error, Cross-Entropy, …)
- This is achieved via a **gradient descent** (stochastic gradient descent, RMSprop, ADAM, …)

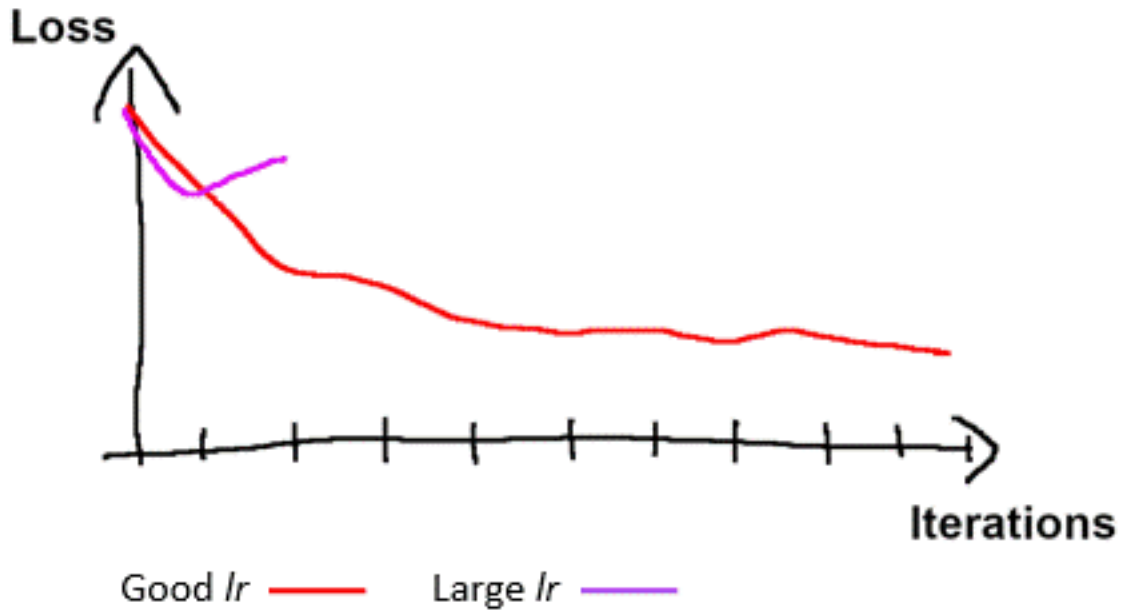




OPTIMIZATION

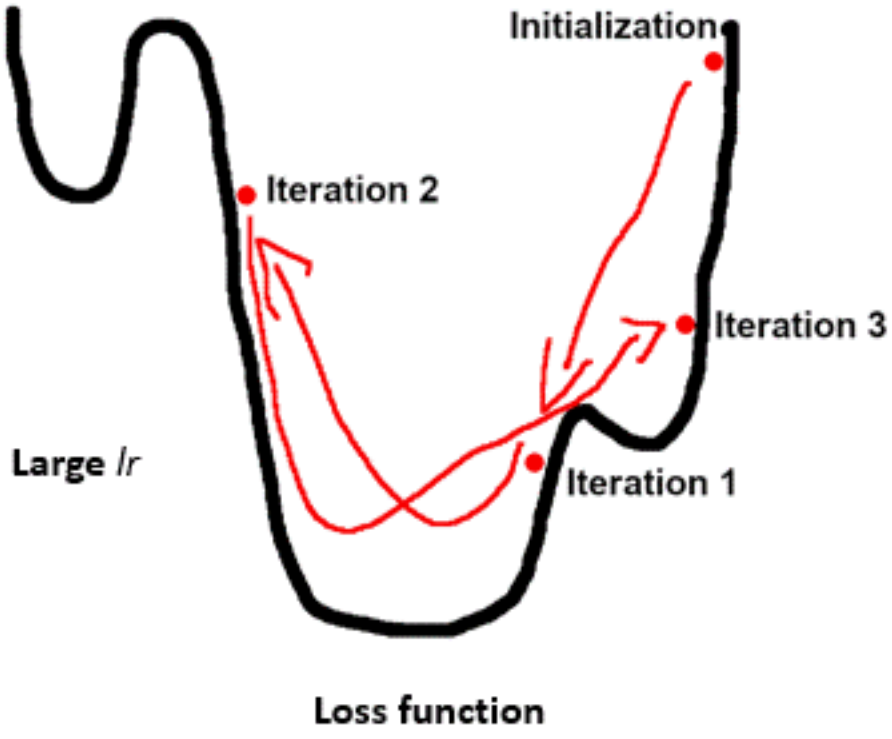
- The goal is to **minimize the loss (cost) function** $Loss(y_{real}, y)$ (Mean Squared, error, Cross-Entropy, …)
- This is achieved via a **gradient descent** (stochastic gradient descent, RMSprop, ADAM, …)

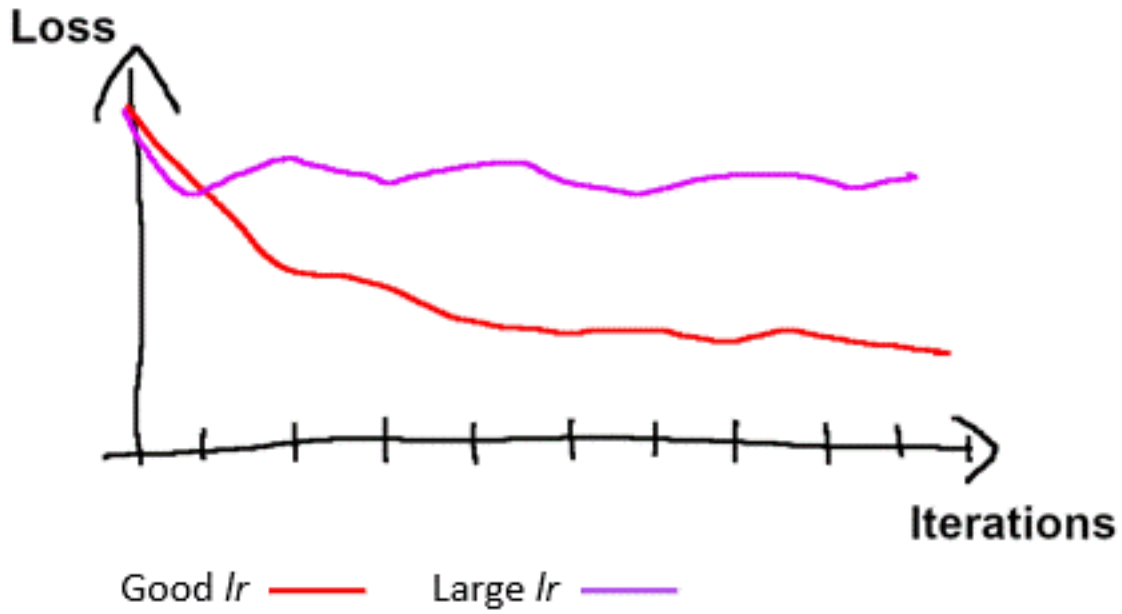




OPTIMIZATION

- The goal is to **minimize the loss (cost) function** $Loss(y_{real}, y)$ (Mean Squared, error, Cross-Entropy, …)
- This is achieved via a **gradient descent** (stochastic gradient descent, RMSprop, ADAM, …)



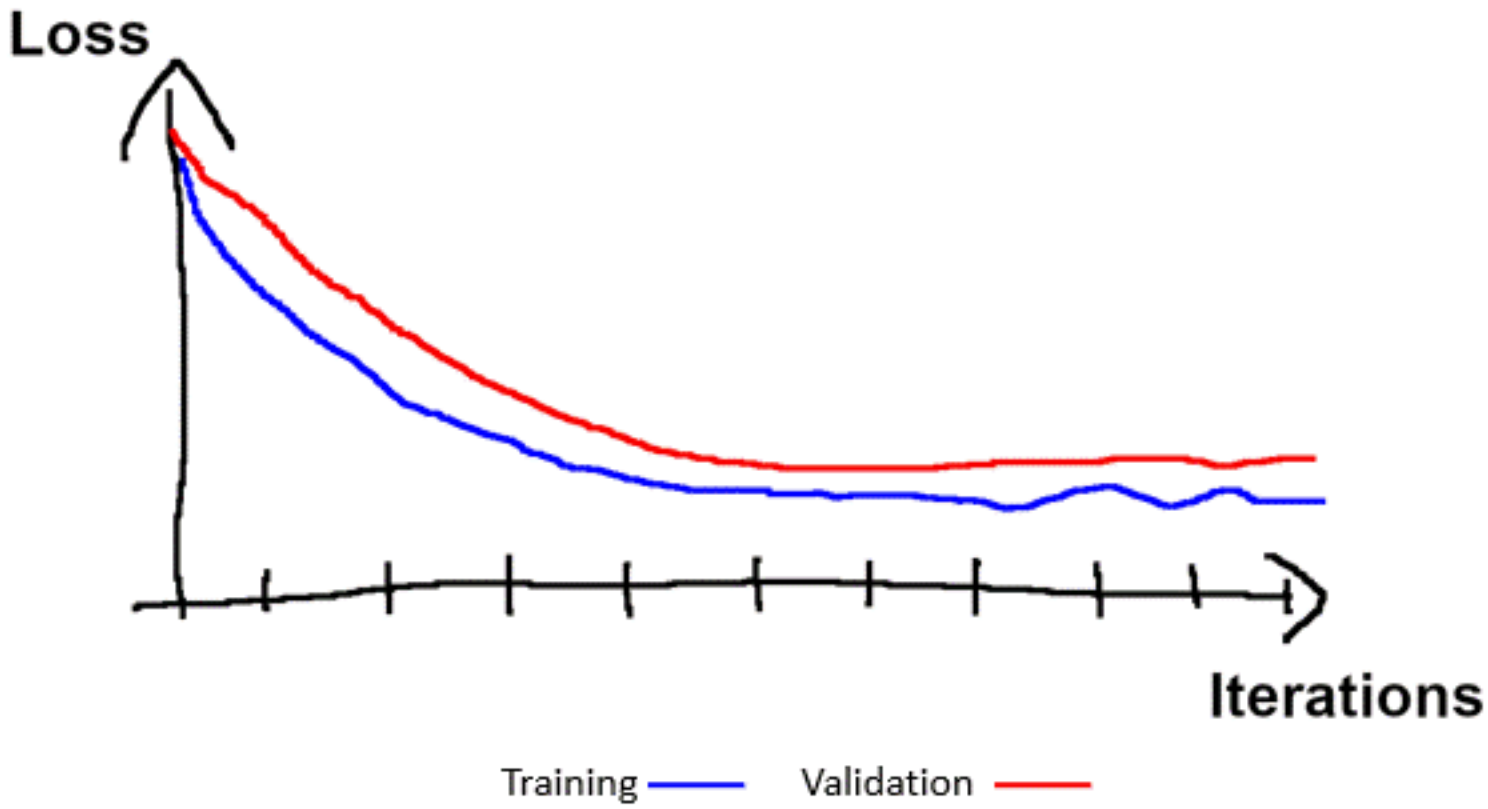


Training and Validation

Training dataset is divided into two parts: **training** and **validation**

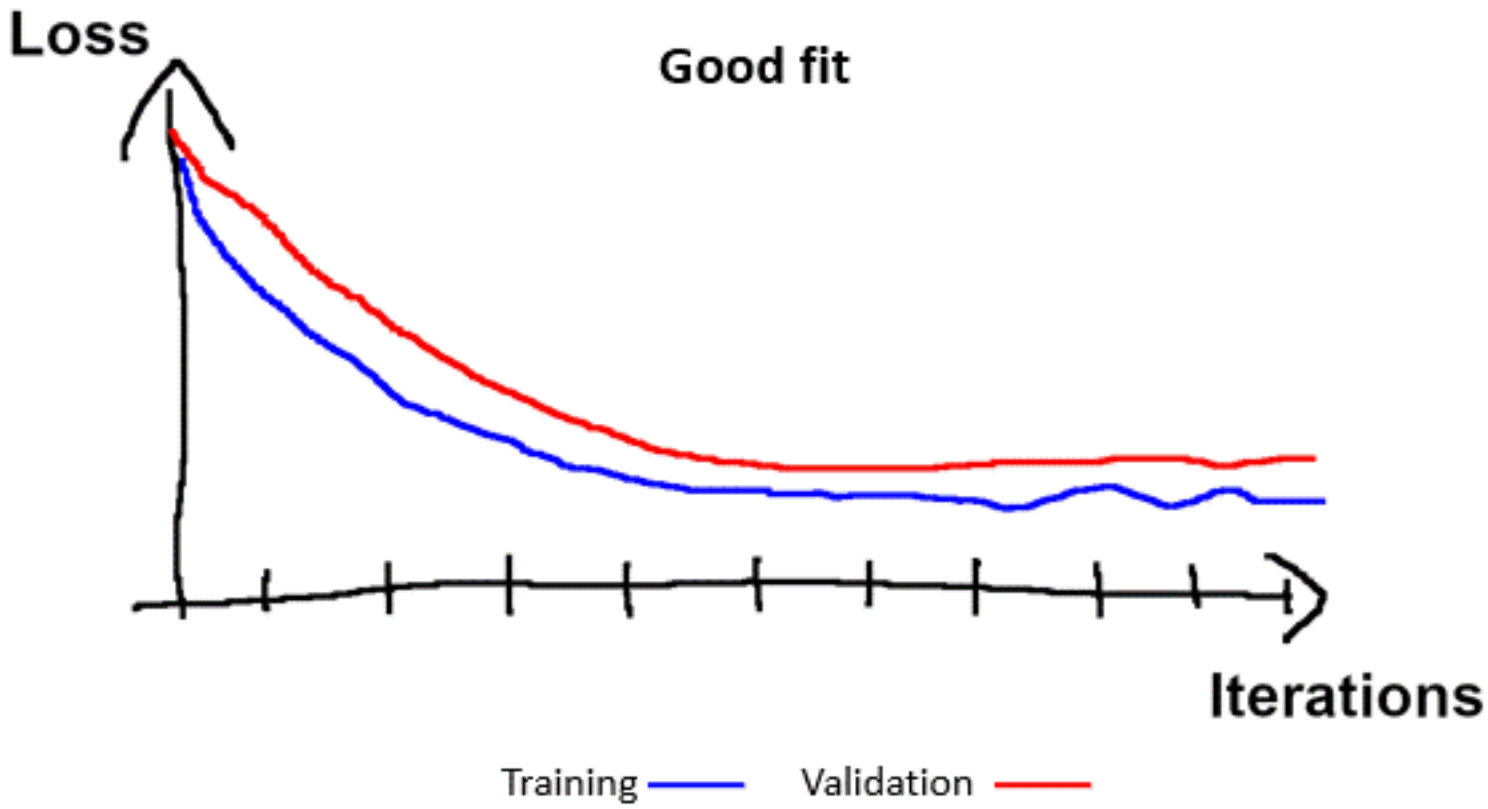
TRAINING AND VALIDATION

Training dataset is divided into two parts: **training** and **validation**



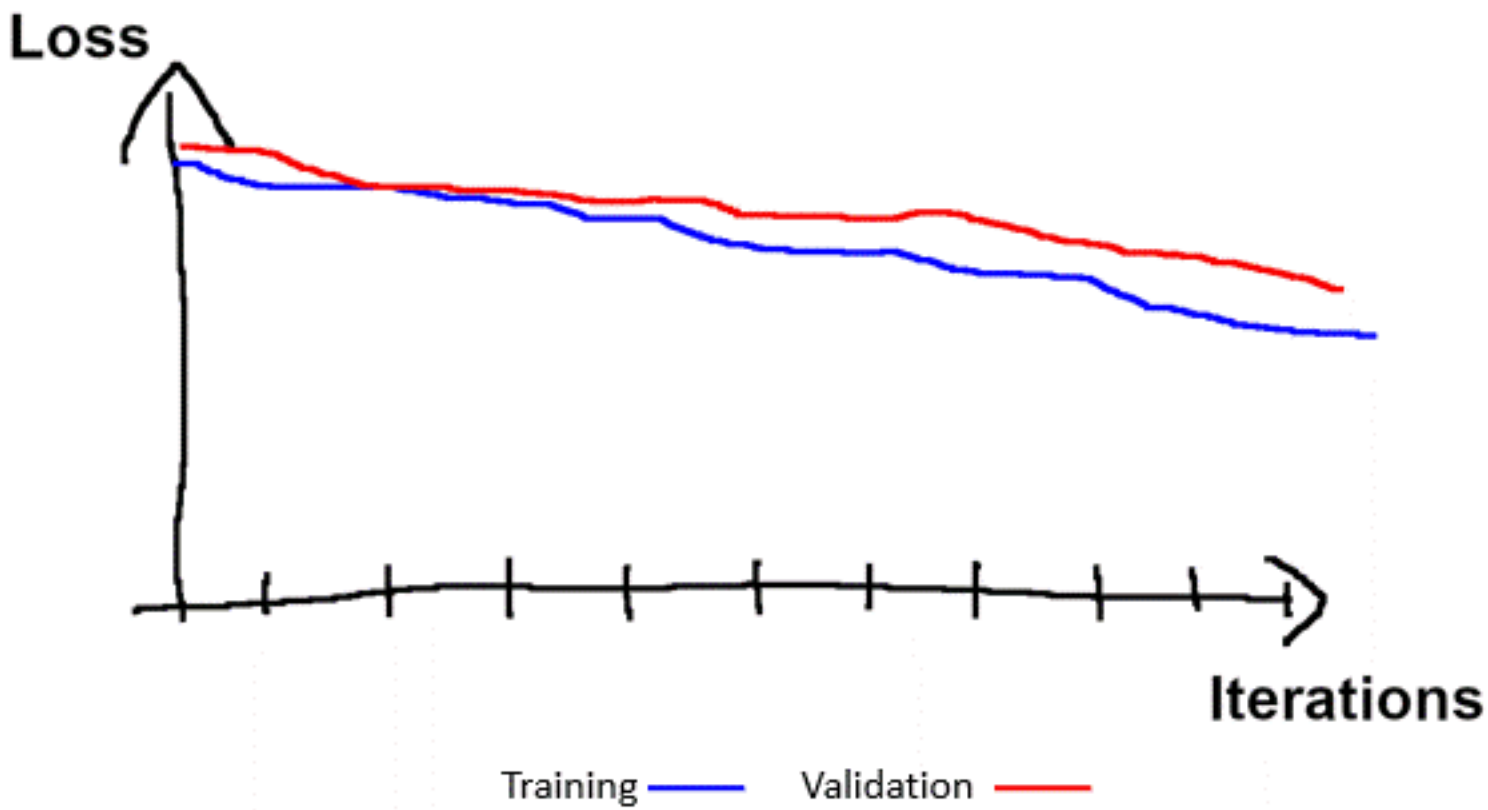
TRAINING AND VALIDATION

Training dataset is divided into two parts: **training** and **validation**



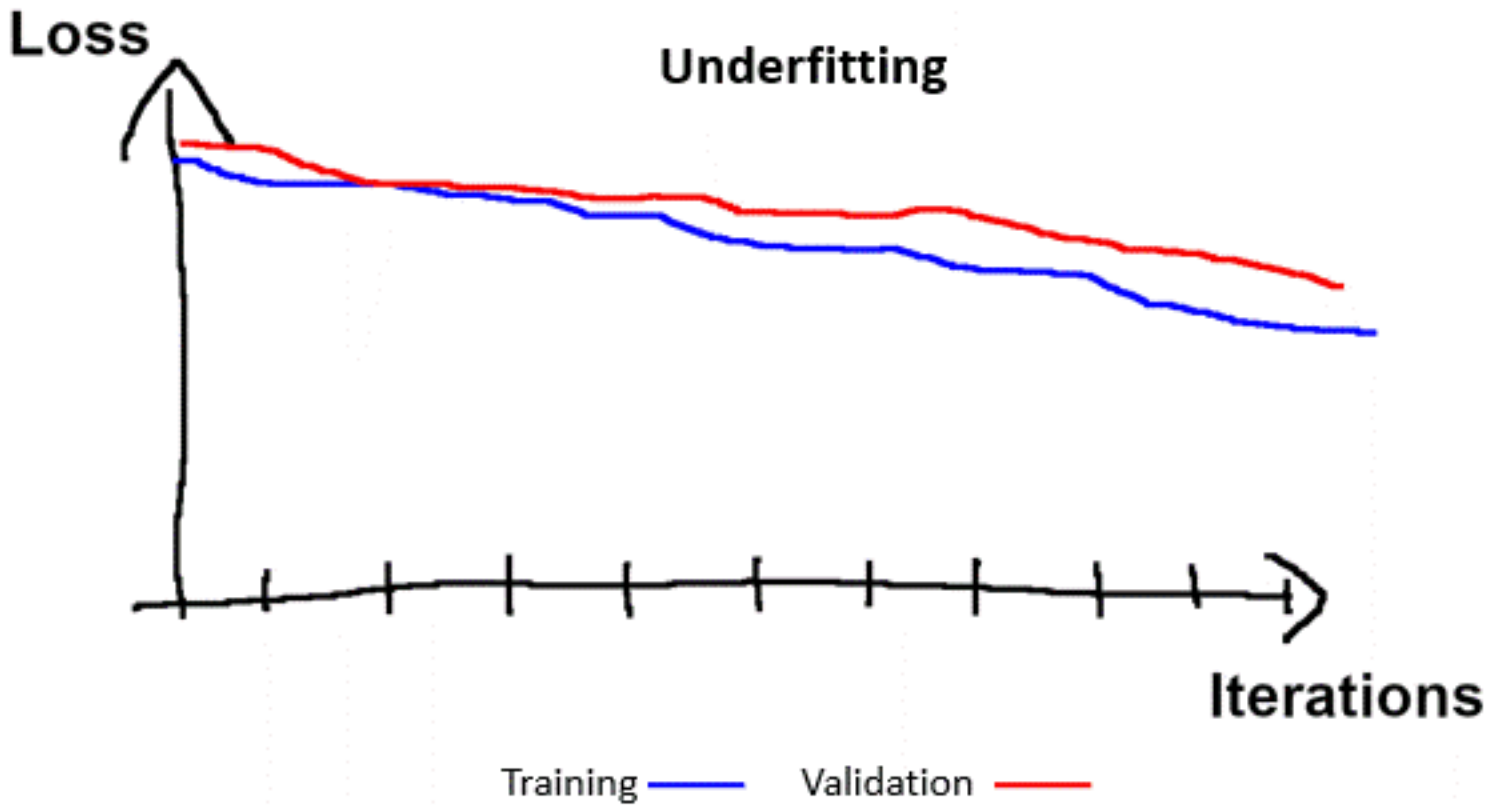
Training and Validation

Training dataset is divided into two parts: **training** and **validation**



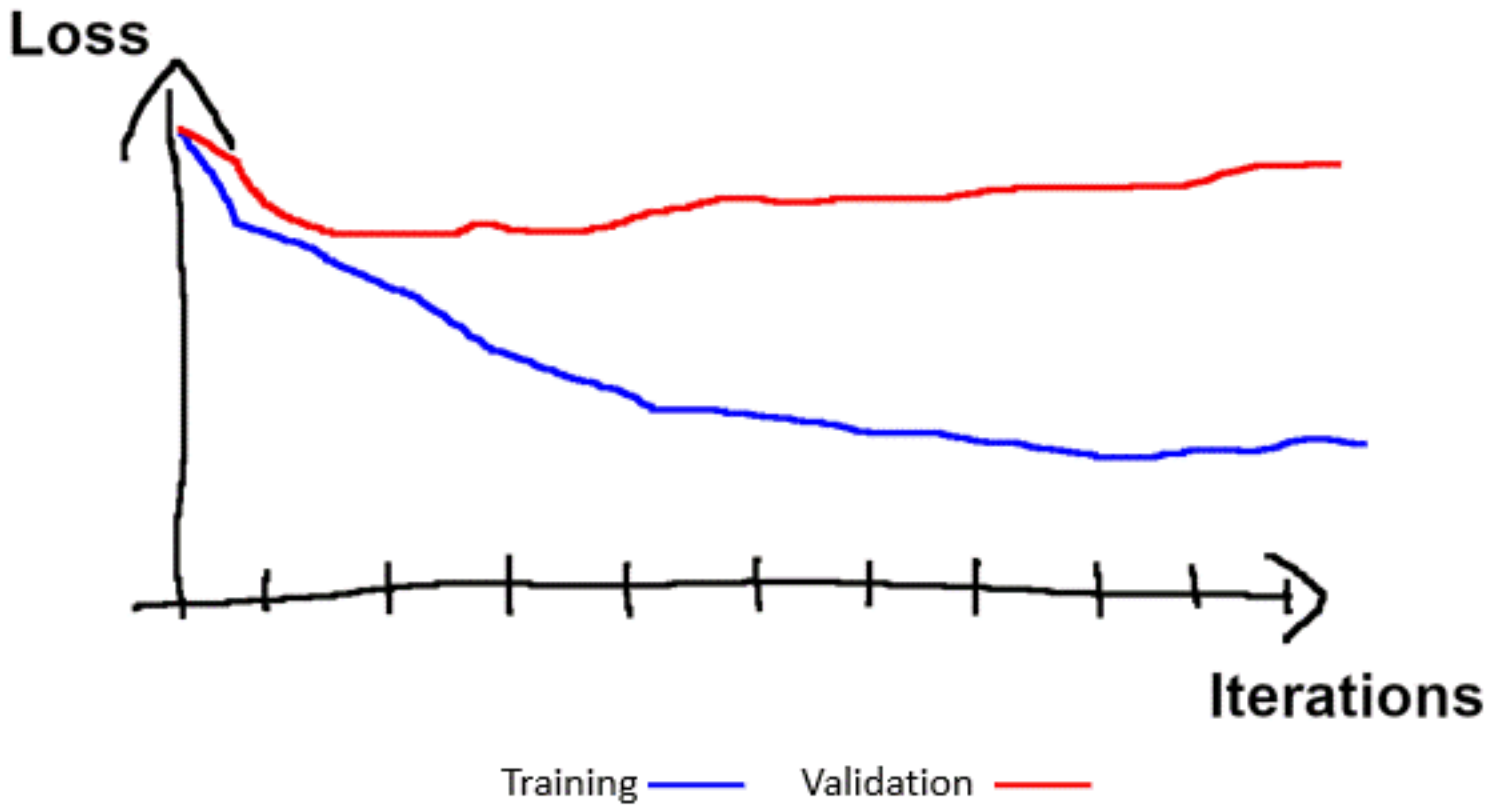
Training and Validation

Training dataset is divided into two parts: **training** and **validation**



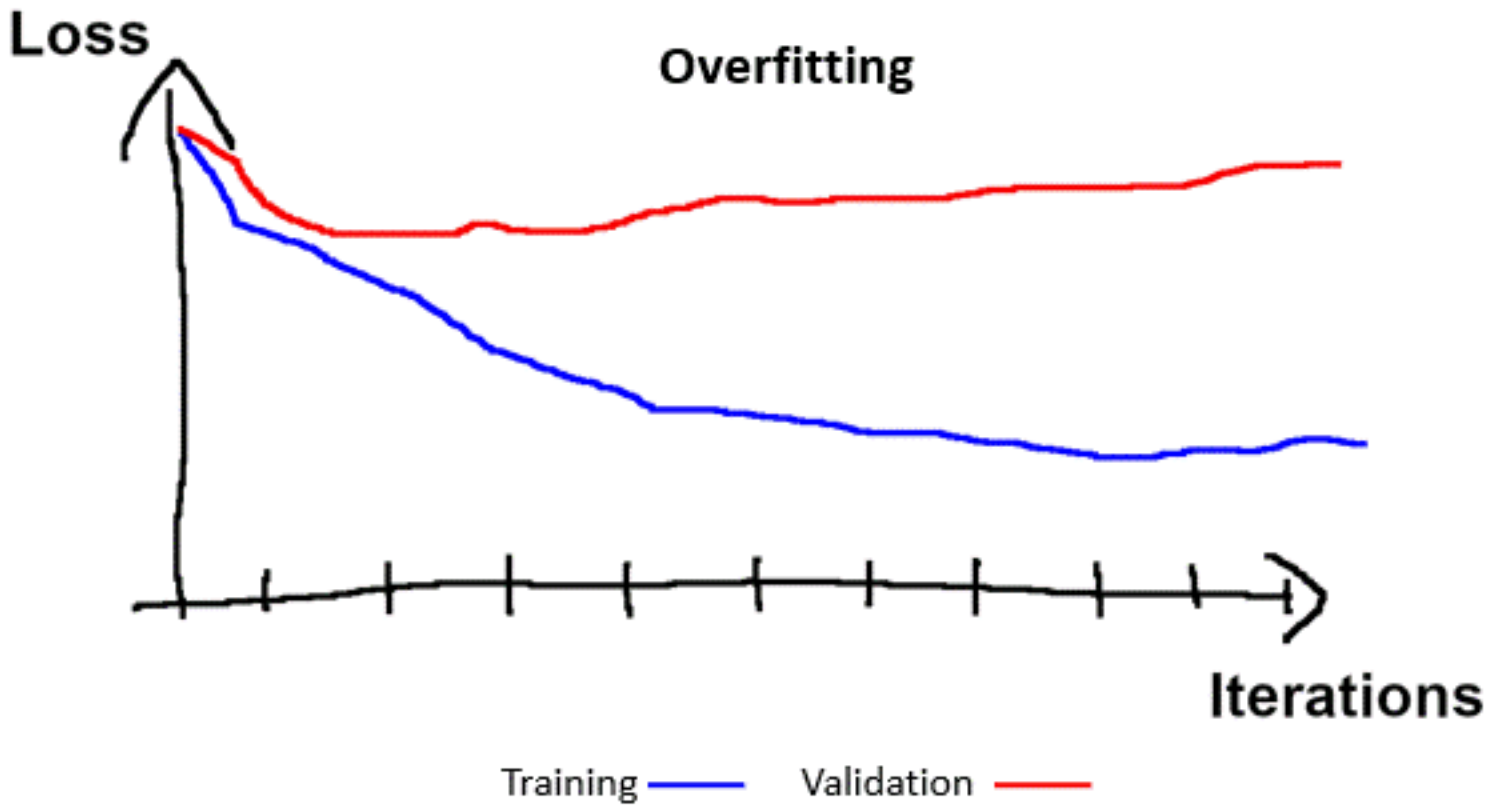
Training and Validation

Training dataset is divided into two parts: **training** and **validation**



Training and Validation

Training dataset is divided into two parts: **training** and **validation**



Fully Connected Neurons

Input layer:

5x5 image patch

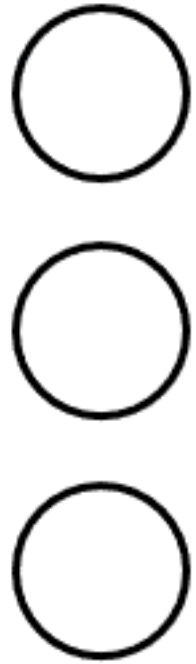
10	15	22	18	16
19	77	76	74	17
24	79	98	72	19
19	74	81	68	18
13	14	18	17	16

Fully Connected Neurons

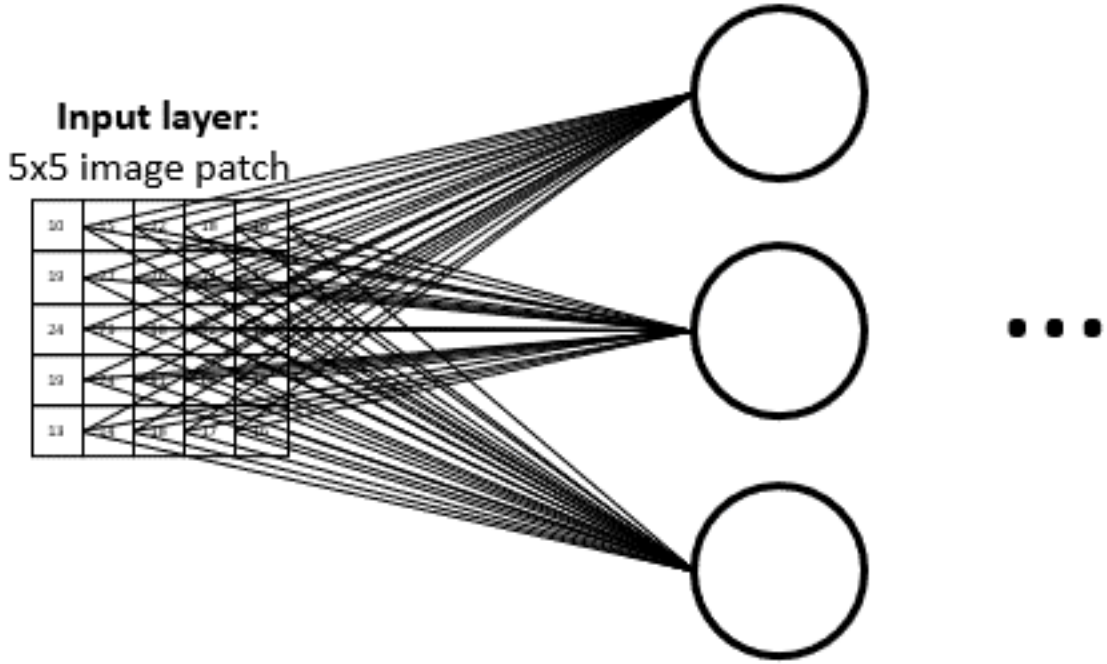
Input layer:

5x5 image patch

10	15	22	18	16
10	77	76	74	17
24	79	98	72	19
10	74	81	68	18
13	14	18	17	16



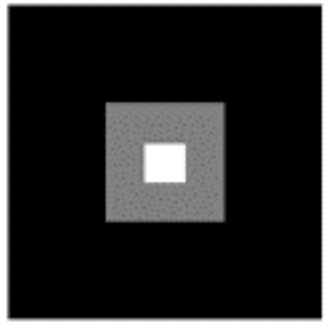
Fully Connected Neurons



CONVOLUTION

Image

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	5	5	5	0	0
0	0	5	10	5	0	0
0	0	5	5	5	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0



Filter

1	1	1
1	1	1
1	1	1

Filtered image

CONVOLUTION

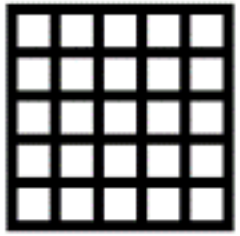
Image

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	5	5	5	0	0
0	0	5	10	5	0	0
0	0	5	5	5	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

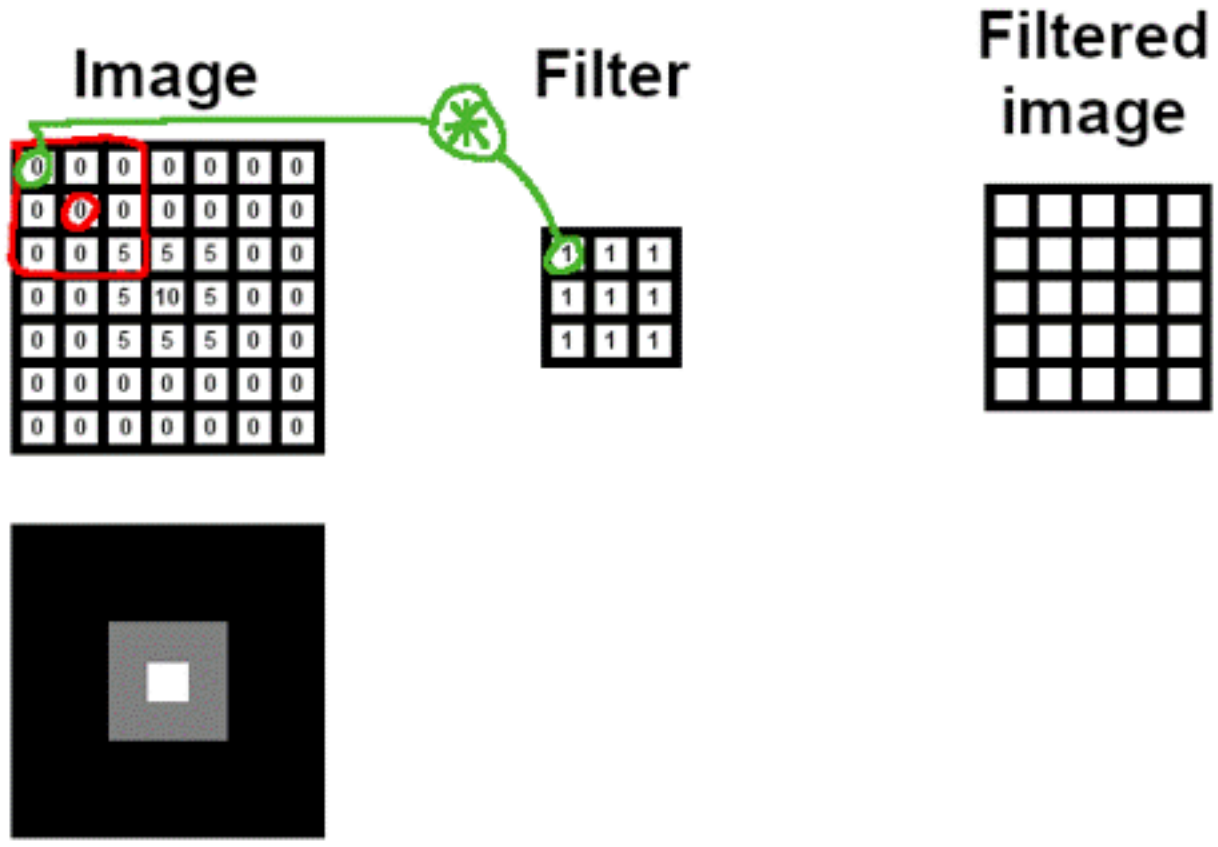
Filter

1	1	1
1	1	1
1	1	1

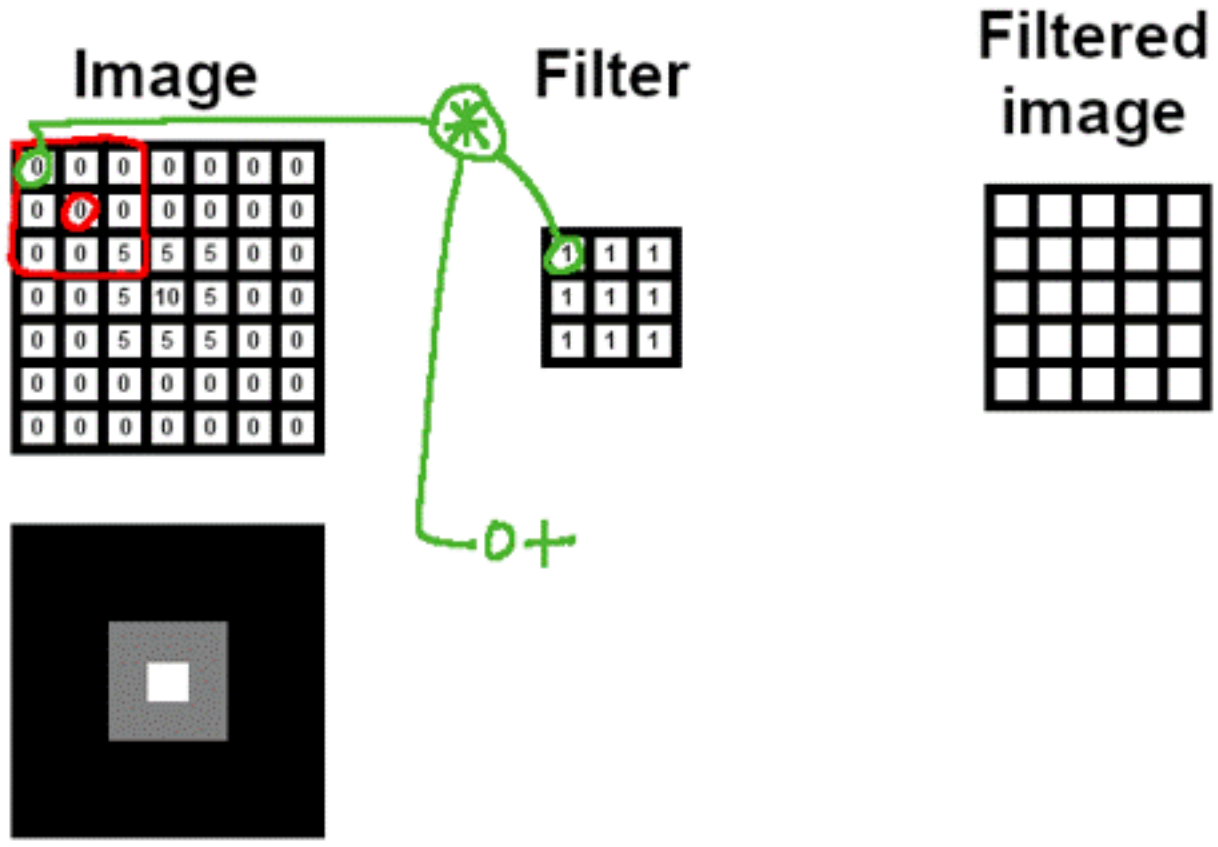
Filtered image



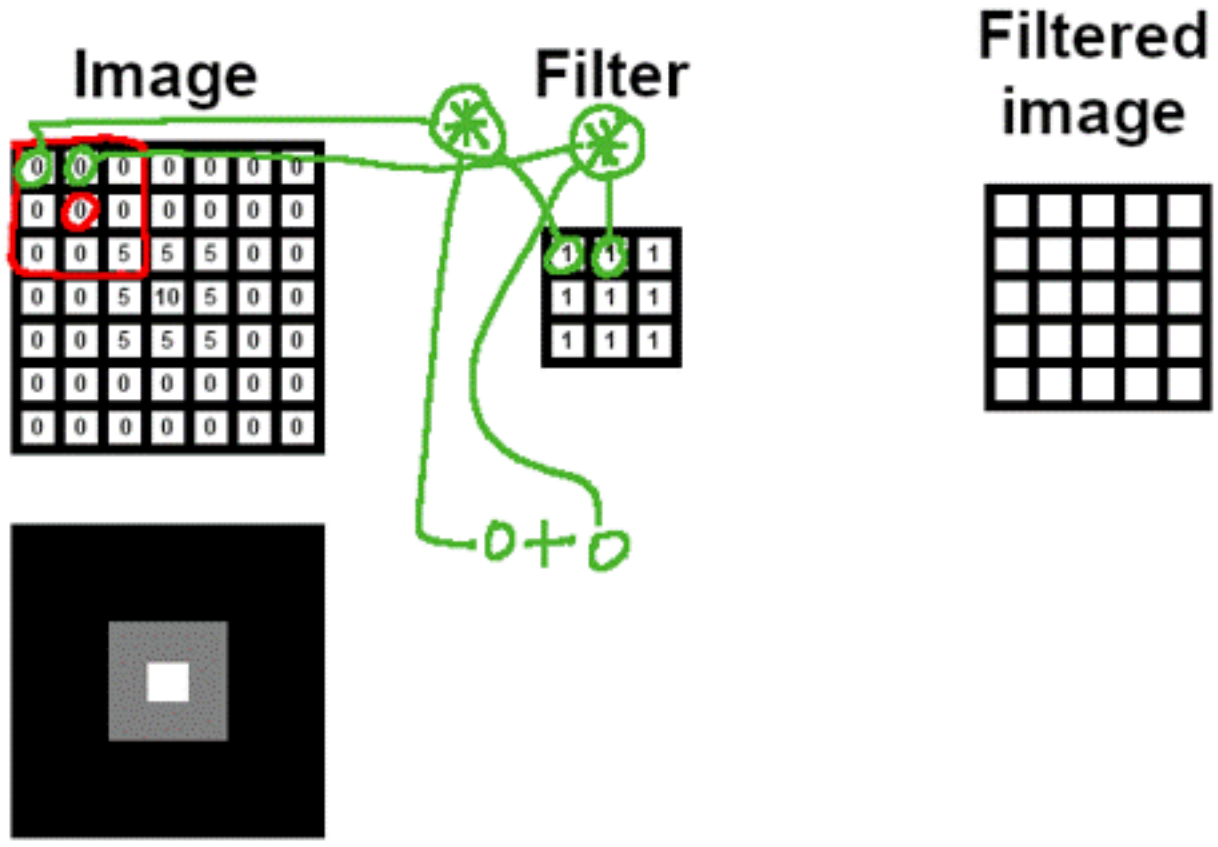
CONVOLUTION



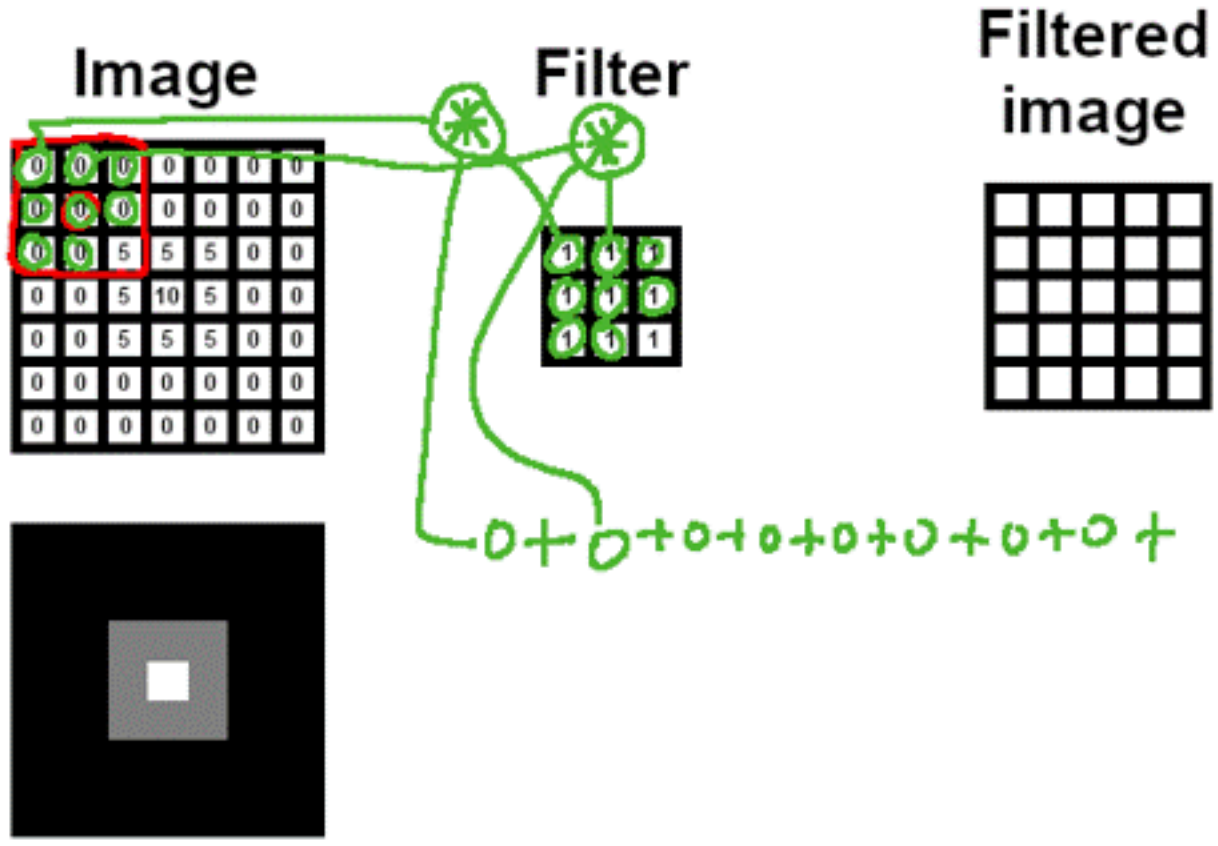
CONVOLUTION



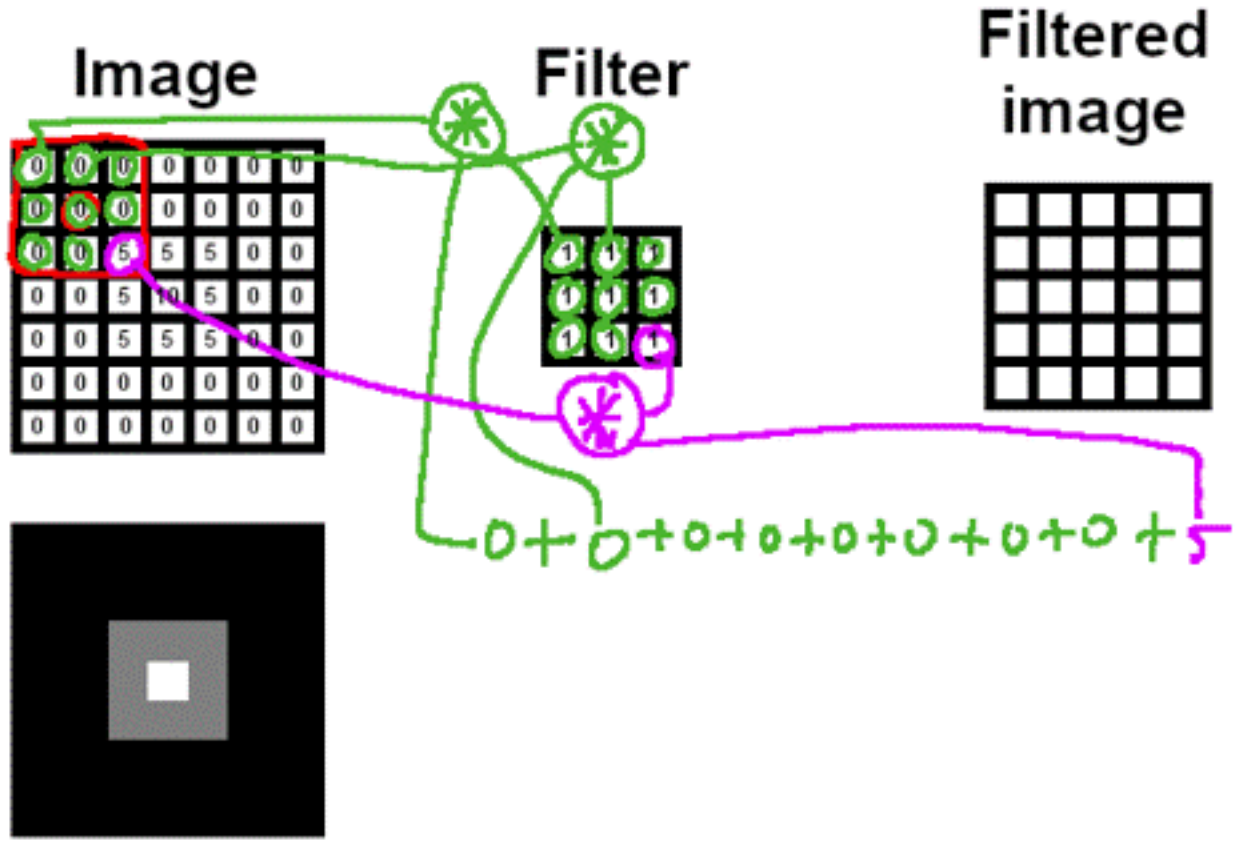
CONVOLUTION



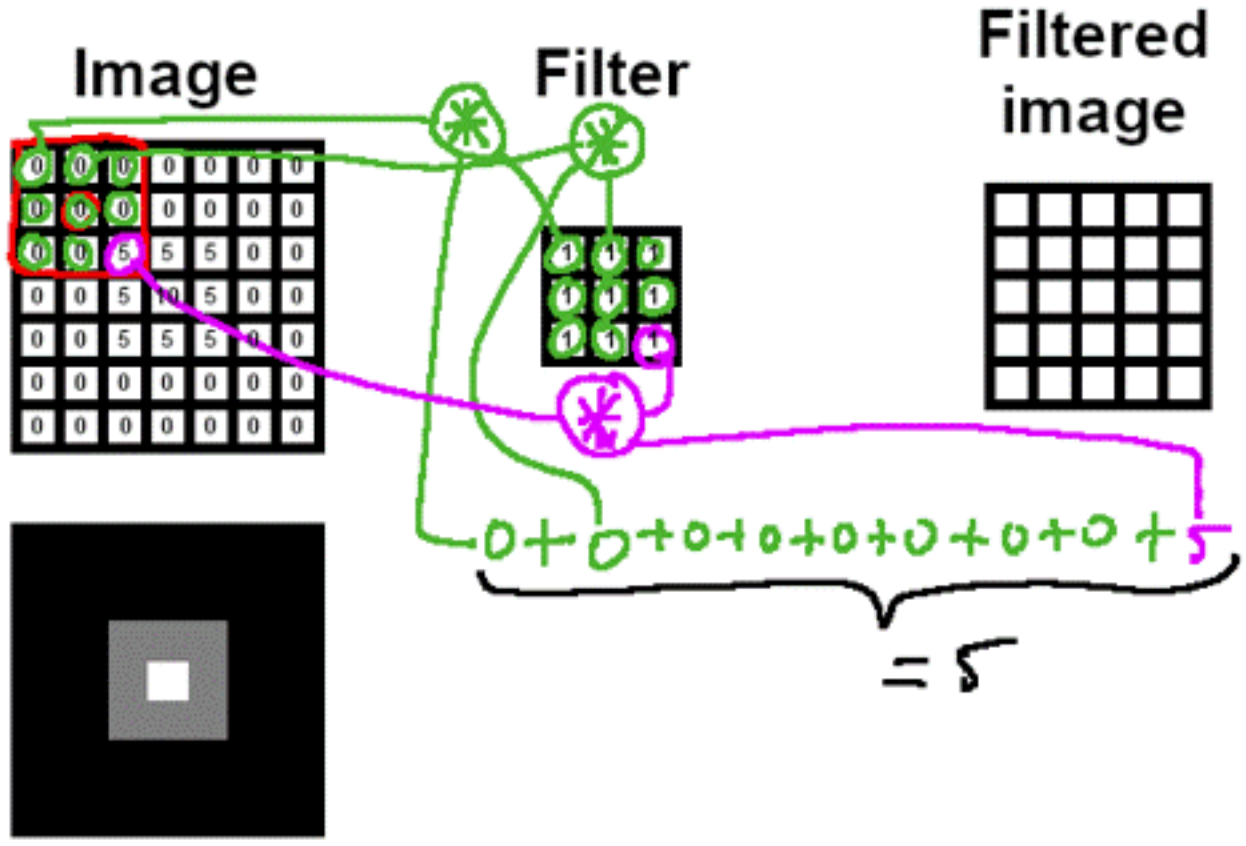
CONVOLUTION



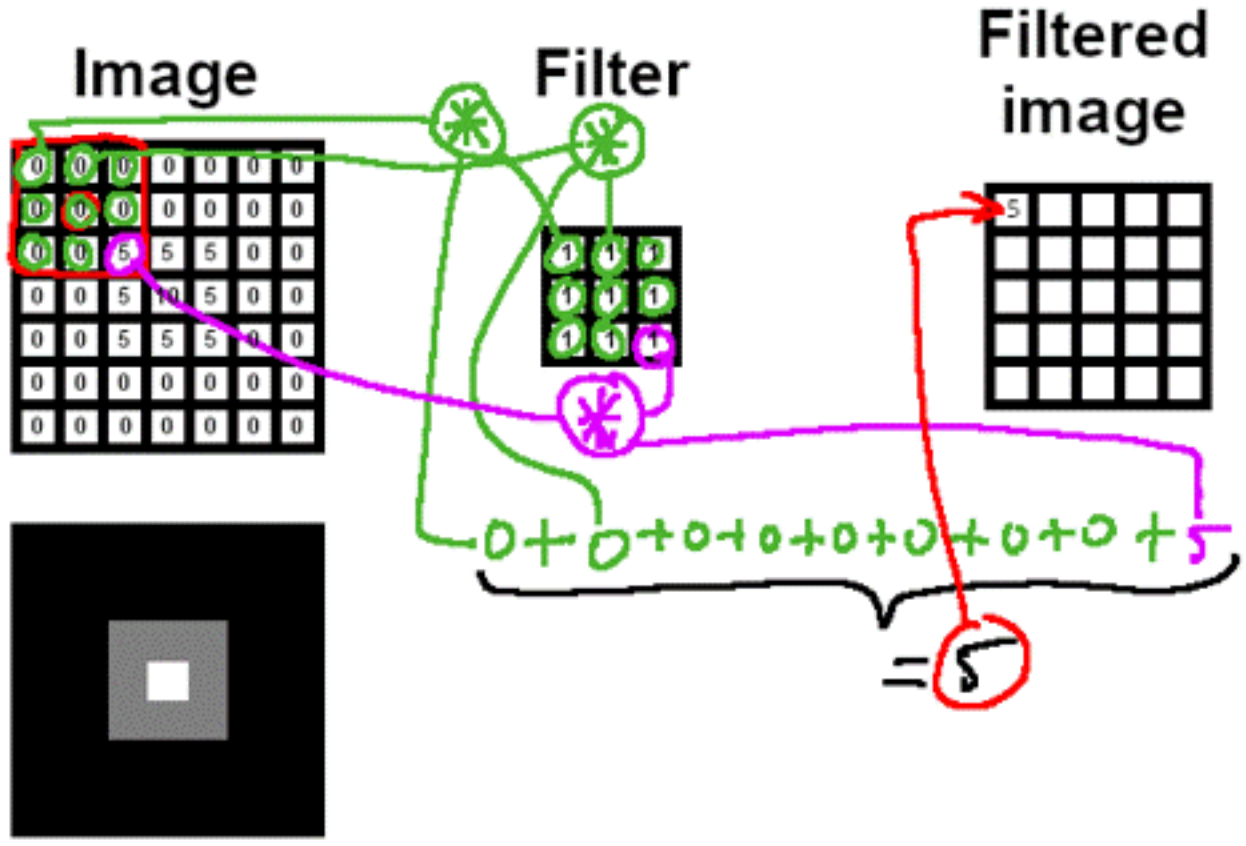
CONVOLUTION



CONVOLUTION



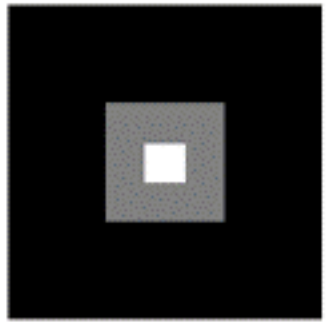
CONVOLUTION



CONVOLUTION

Image

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	5	5	5	0	0
0	0	5	10	5	0	0
0	0	5	5	5	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0



Filter

1	1	1
1	1	1
1	1	1

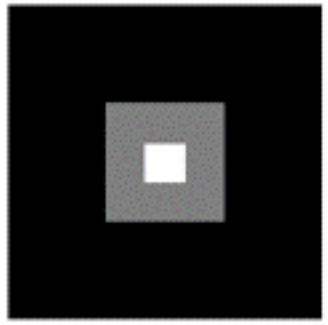
Filtered image

5				

CONVOLUTION

Image

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	5	5	5	0	0
0	0	5	10	5	0	0
0	0	5	5	5	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0



Filter

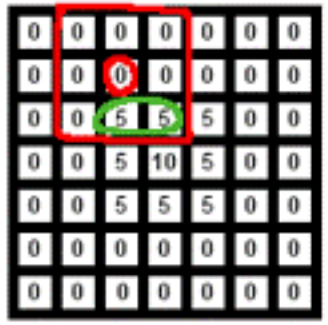
1	1	1
1	1	1
1	1	1

Filtered image

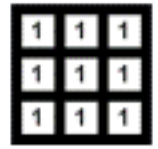
5				

CONVOLUTION

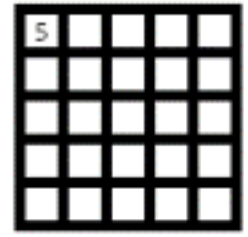
Image



Filter

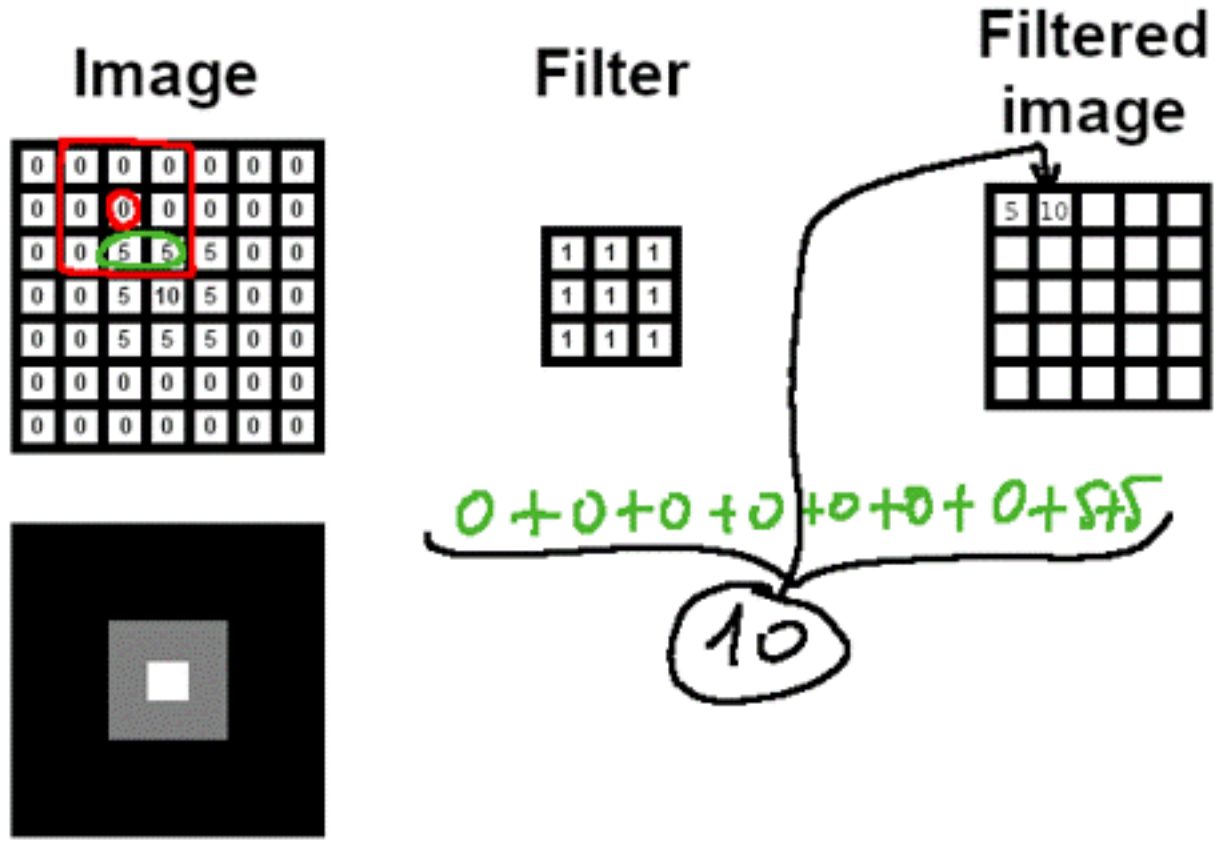


Filtered image

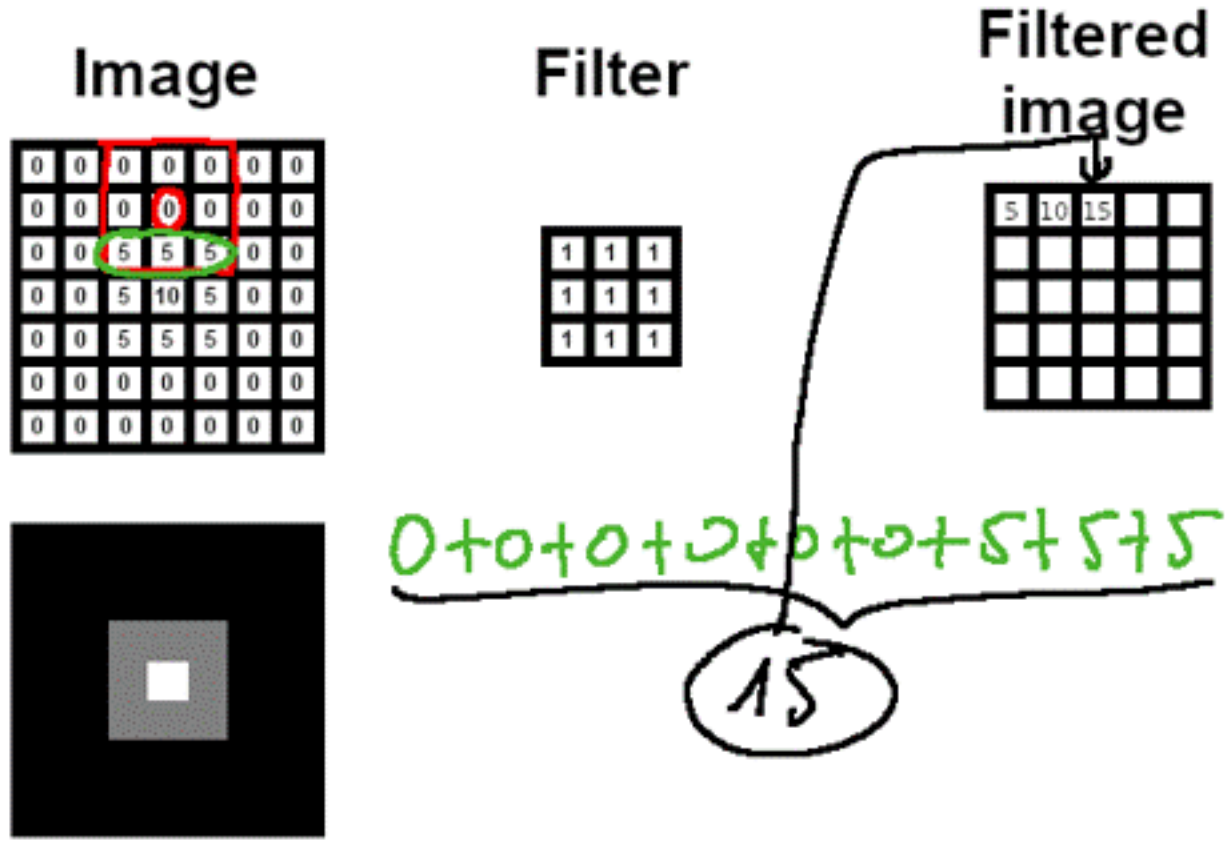


0+0+0+0+0+0+0+5=5

CONVOLUTION



CONVOLUTION



CONVOLUTION

Image

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	5	5	5	0	0
0	0	5	10	5	0	0
0	0	5	5	5	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

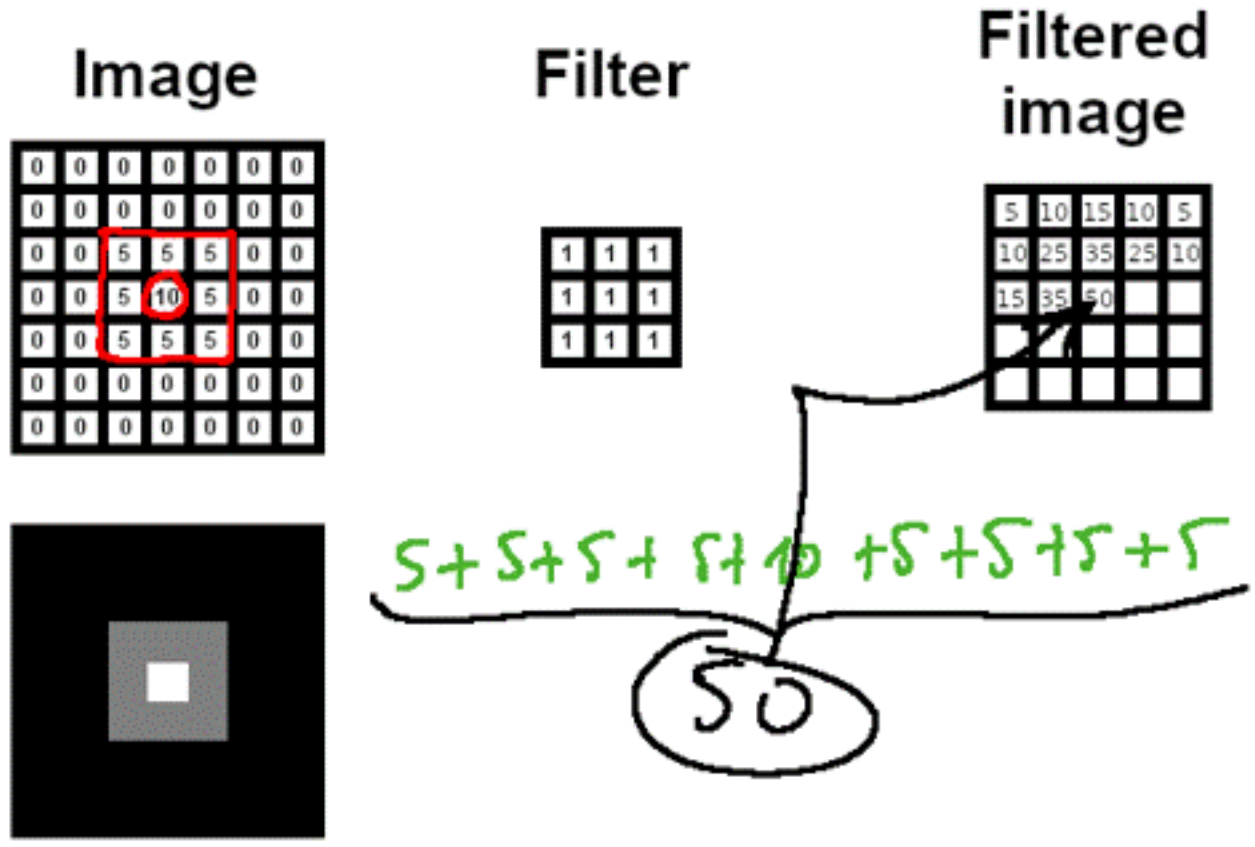
Filter

1	1	1
1	1	1
1	1	1

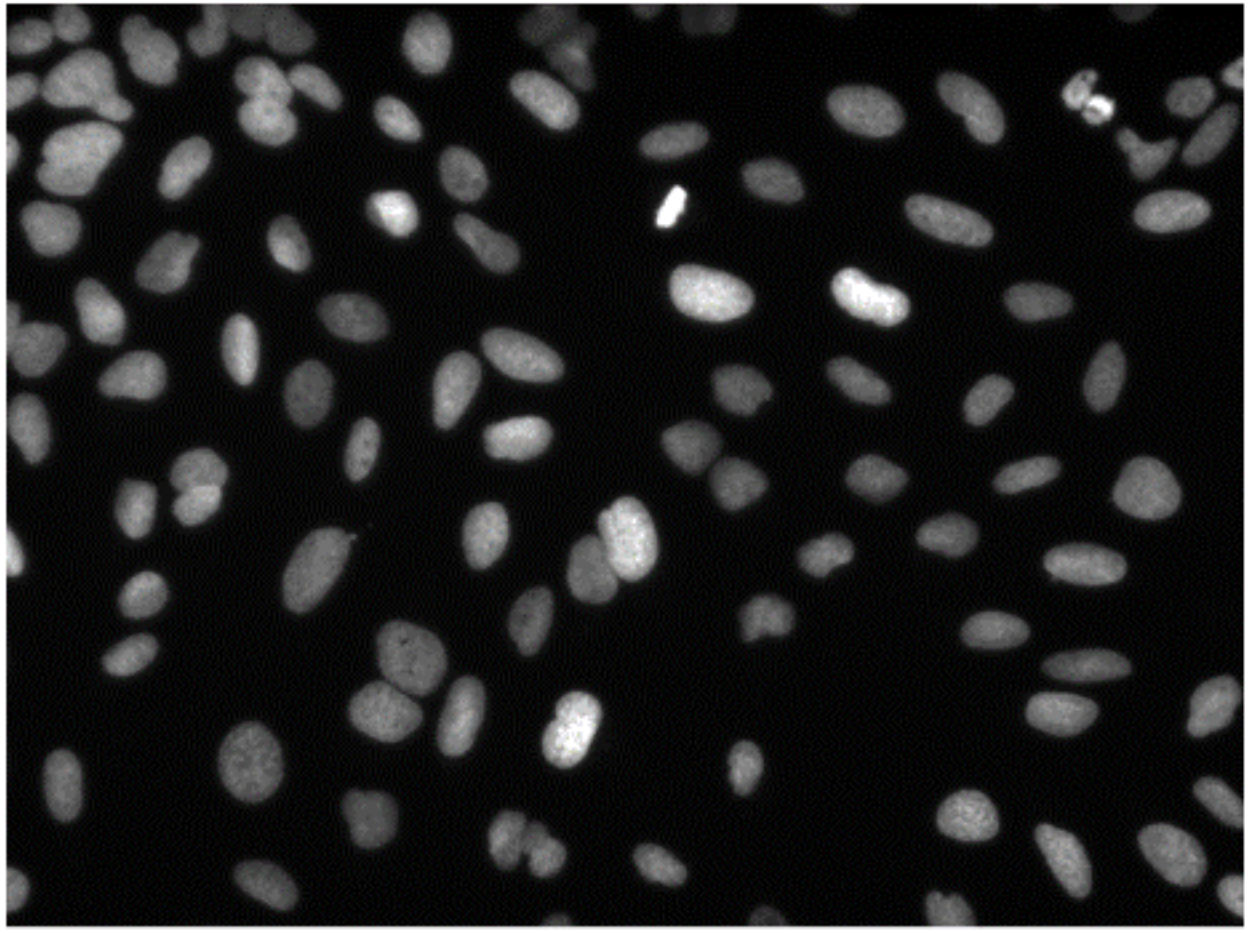
Filtered image

5	10	15	10	5
10	25	35	25	10
15	35			

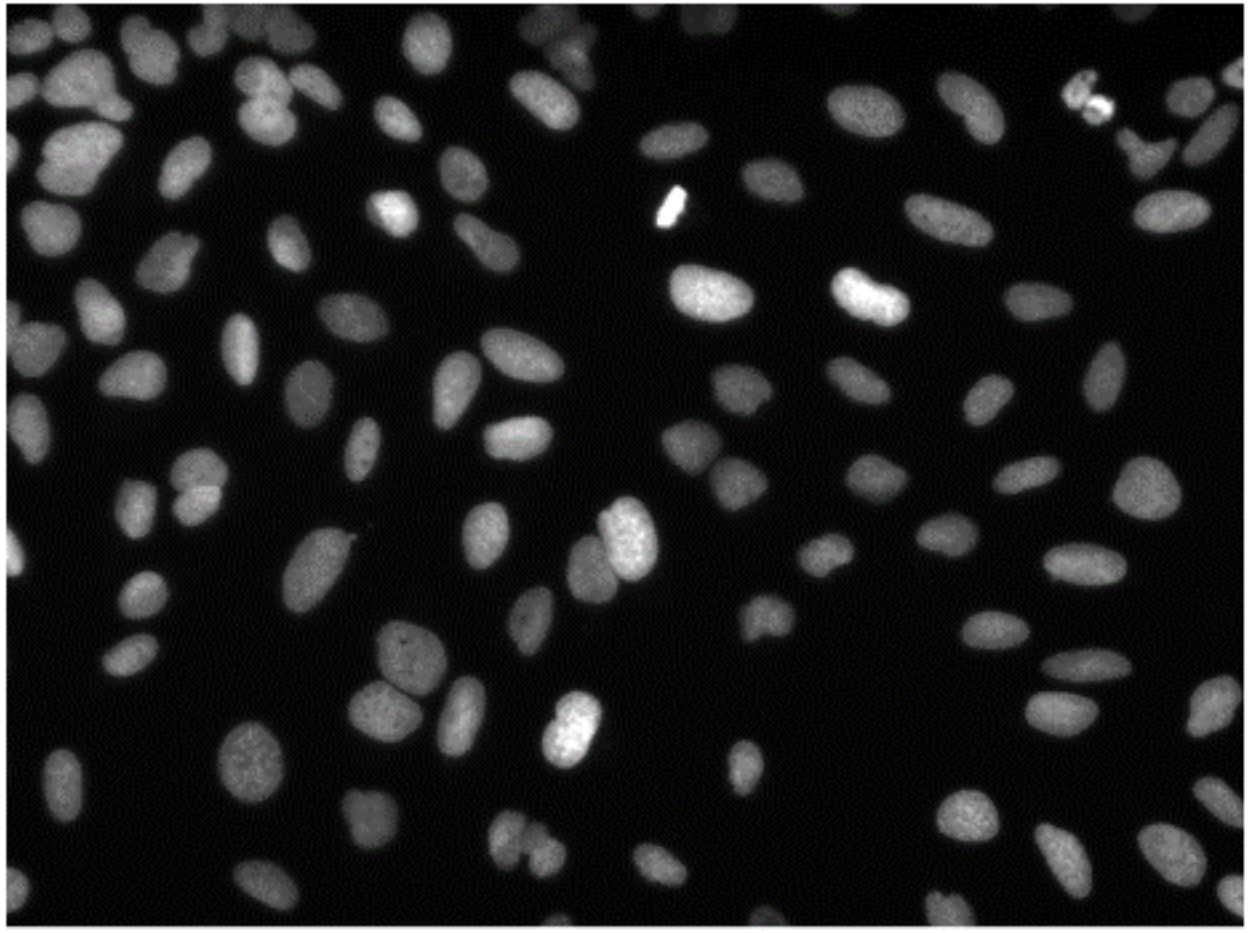
CONVOLUTION

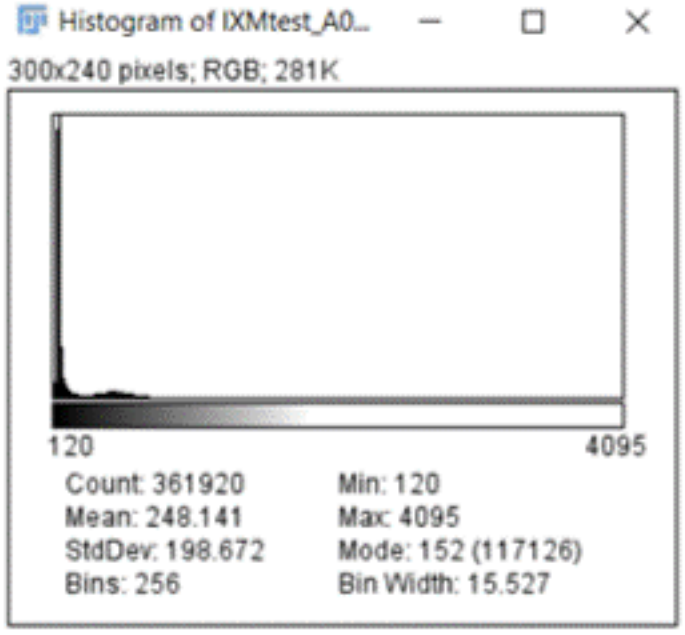


Normalization

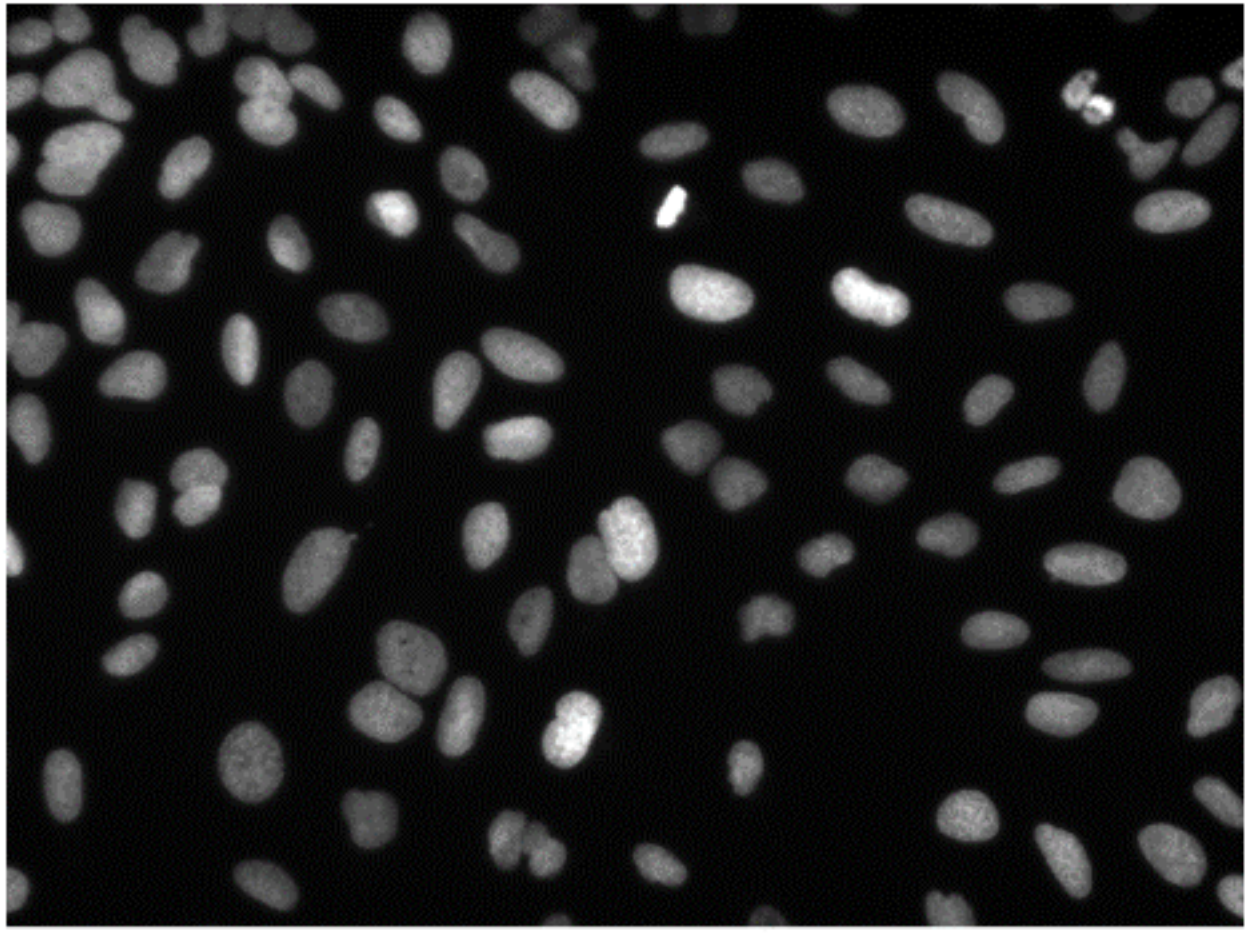


NORMALIZATION

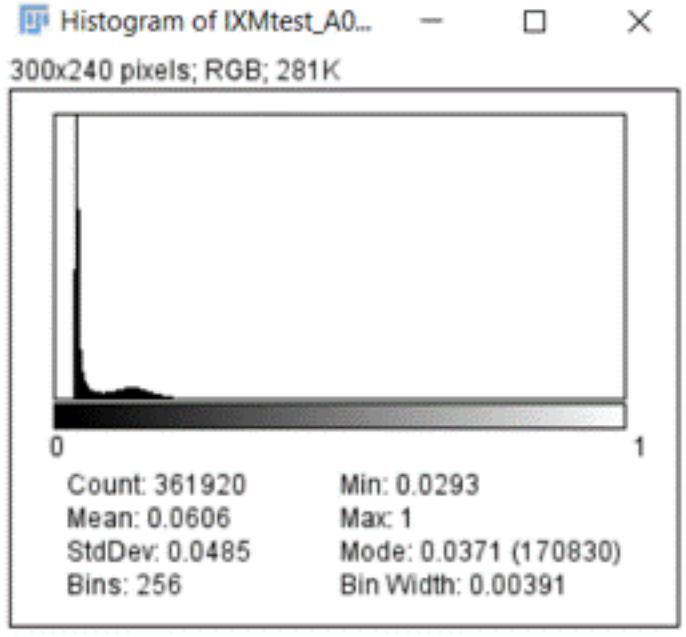




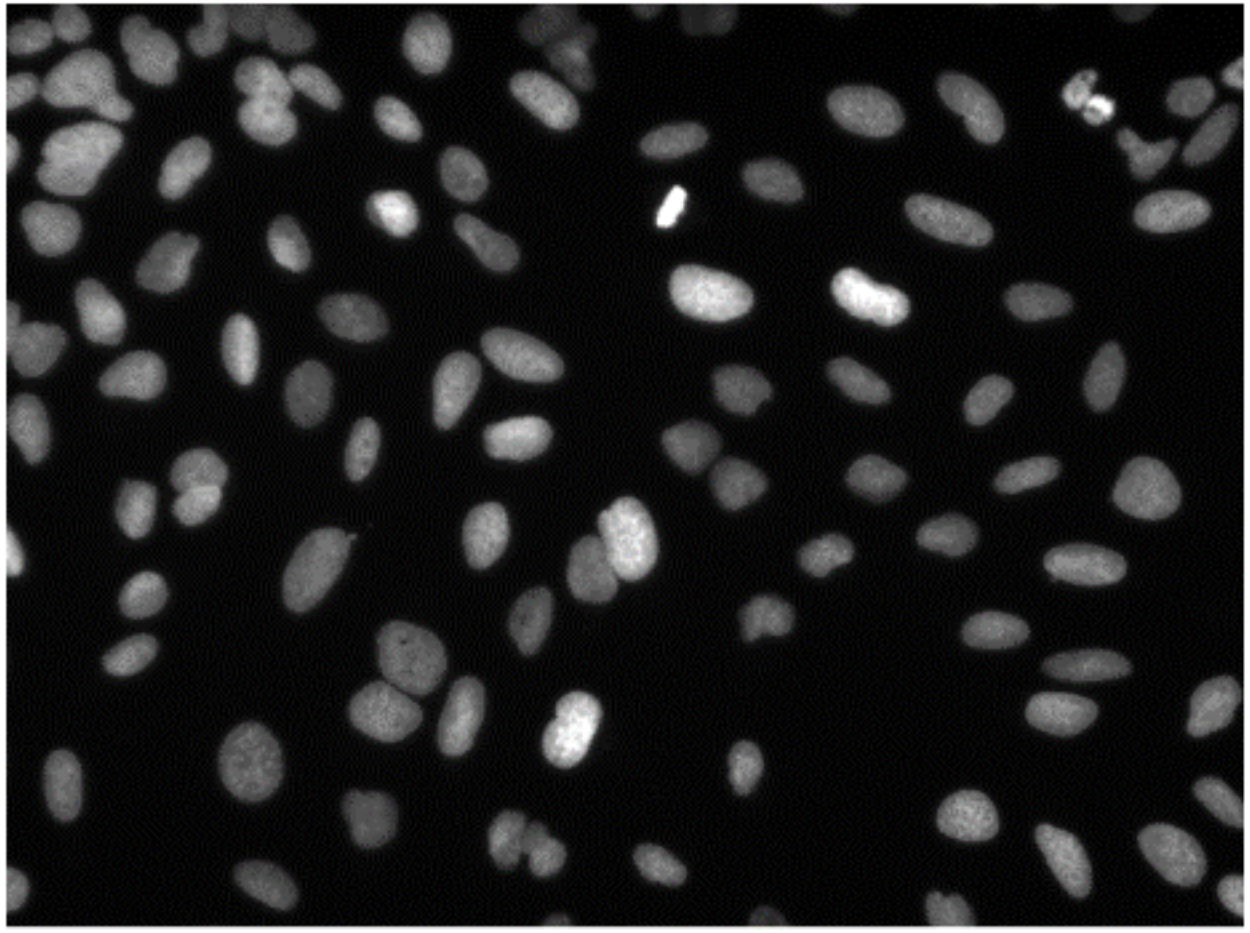
NORMALIZATION



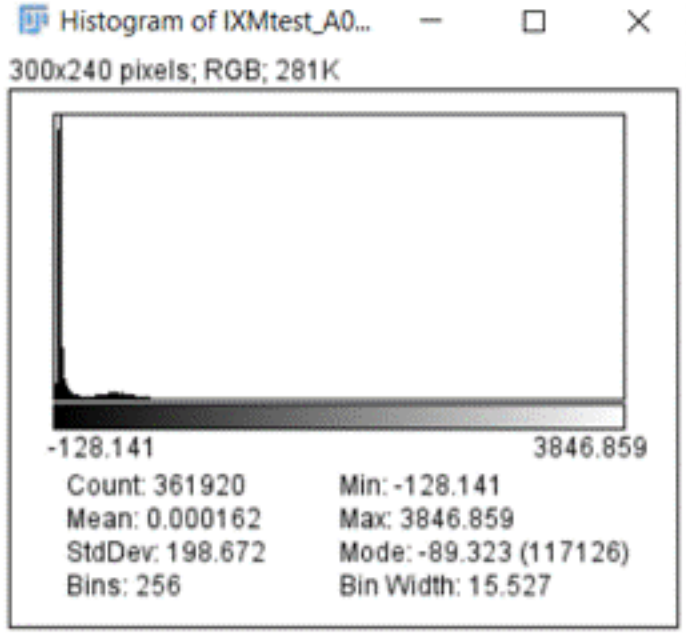
Pixel normalization



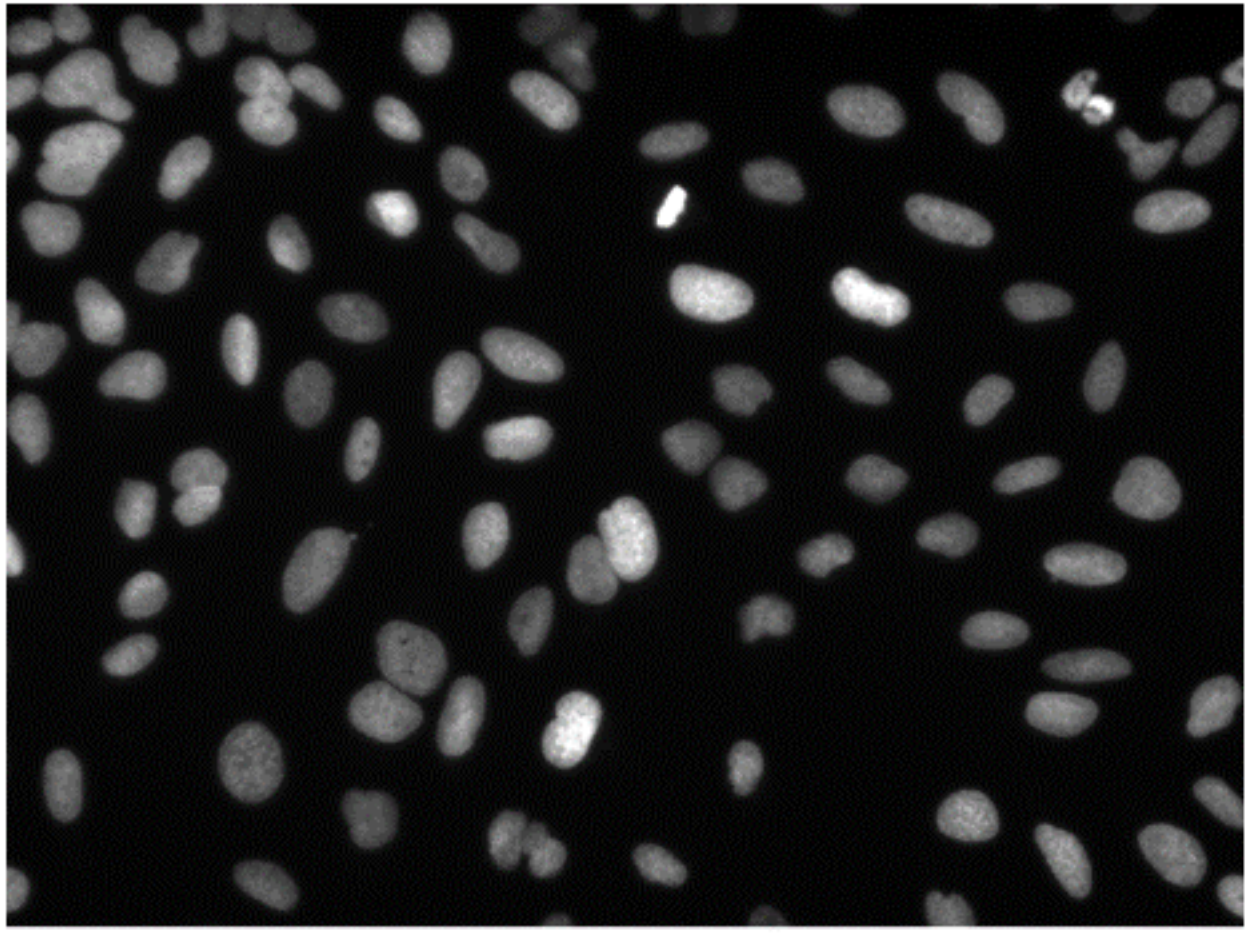
NORMALIZATION



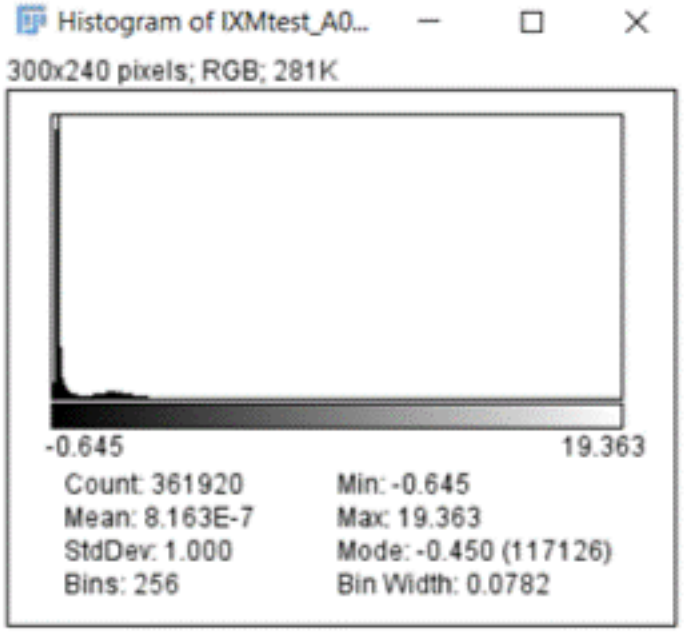
Pixel centering



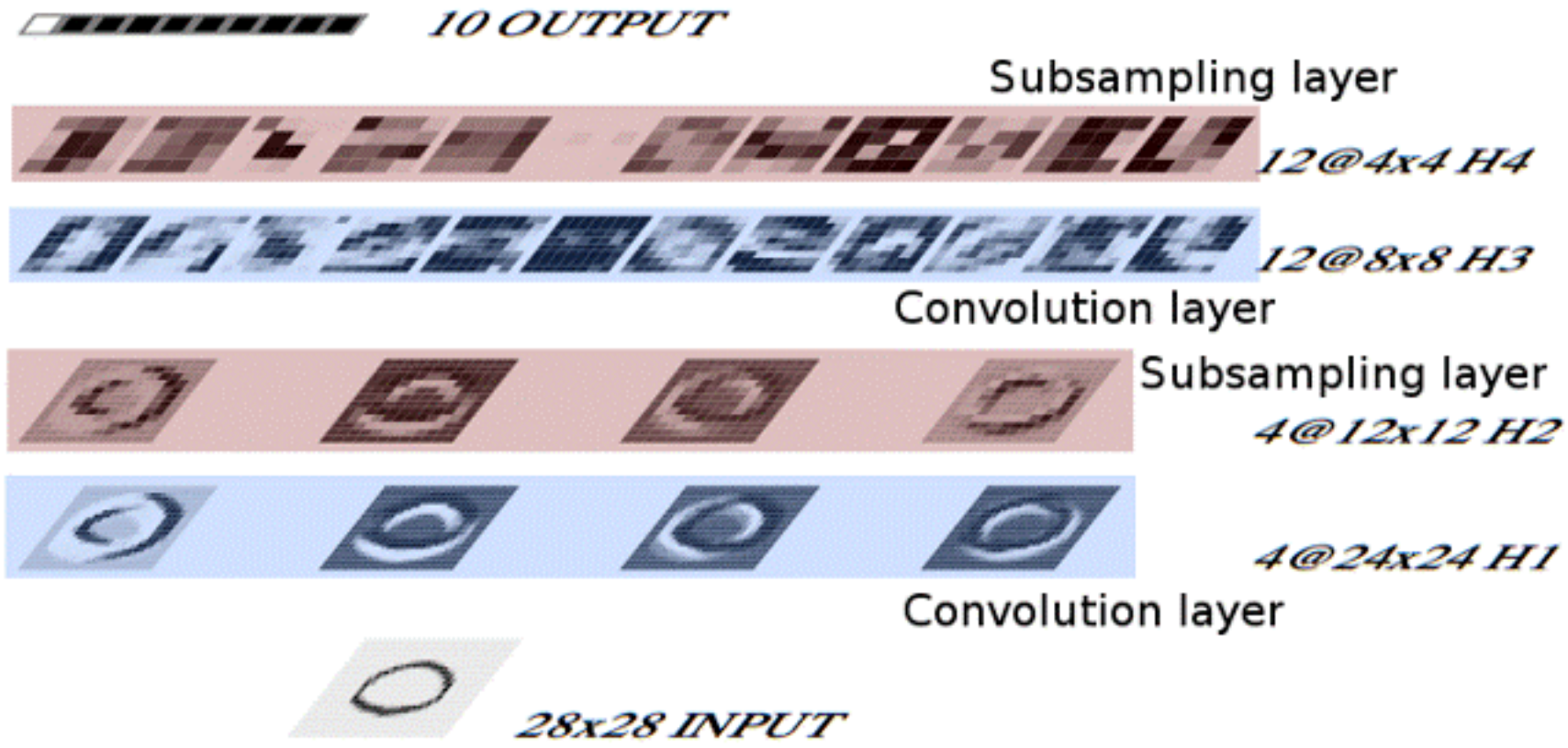
NORMALIZATION



Pixel standardization



CONVOLUTIONAL NEURAL NETWORK (1989)



LeCun *et al.*, Handwritten digit recognition with a back-propagation network, *Neural Information Processing Systems, 1989*

4,635 pixels and 2,578 parameters (much lower than fully connected network)

Deep Learning Requires

1) High computational power

4.4 Speedup factor of GPU code

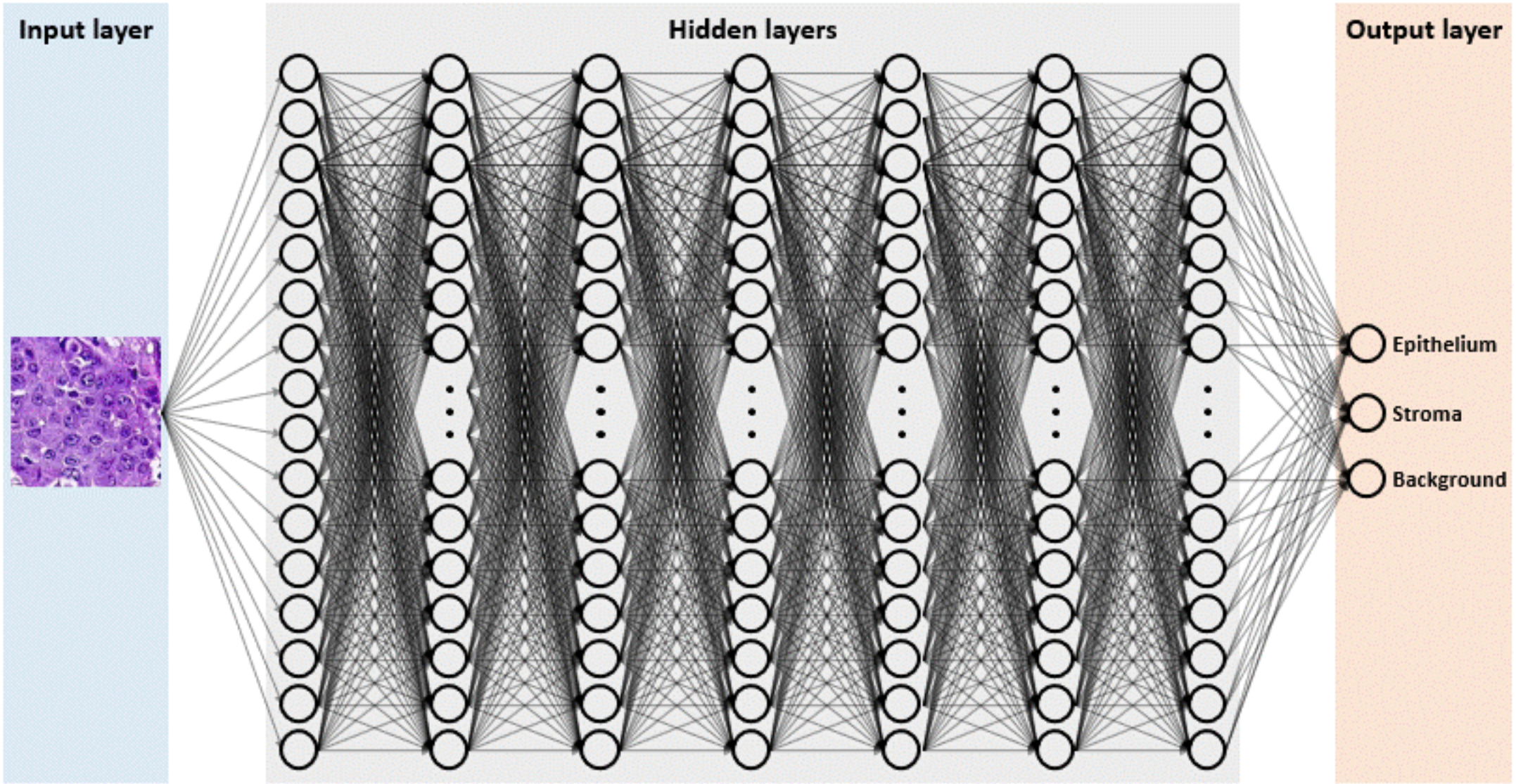
The GPU code scales well with network size. For small nets the speedup is small (but still over 10) since they fit better inside the CPU cache, and GPU resources are underutilized. For huge nets (ex: Table 2) the GPU implementation is more than 60 times faster than a compiler-optimized CPU version. Given the flexibility of our GPU version, this is a significant speedup. One epoch takes 35 GPU minutes but more than 35 CPU hours.

Cireşan *et al.*, High-Performance Neural Networks for Visual Object Classification. Tech Report, 2011.

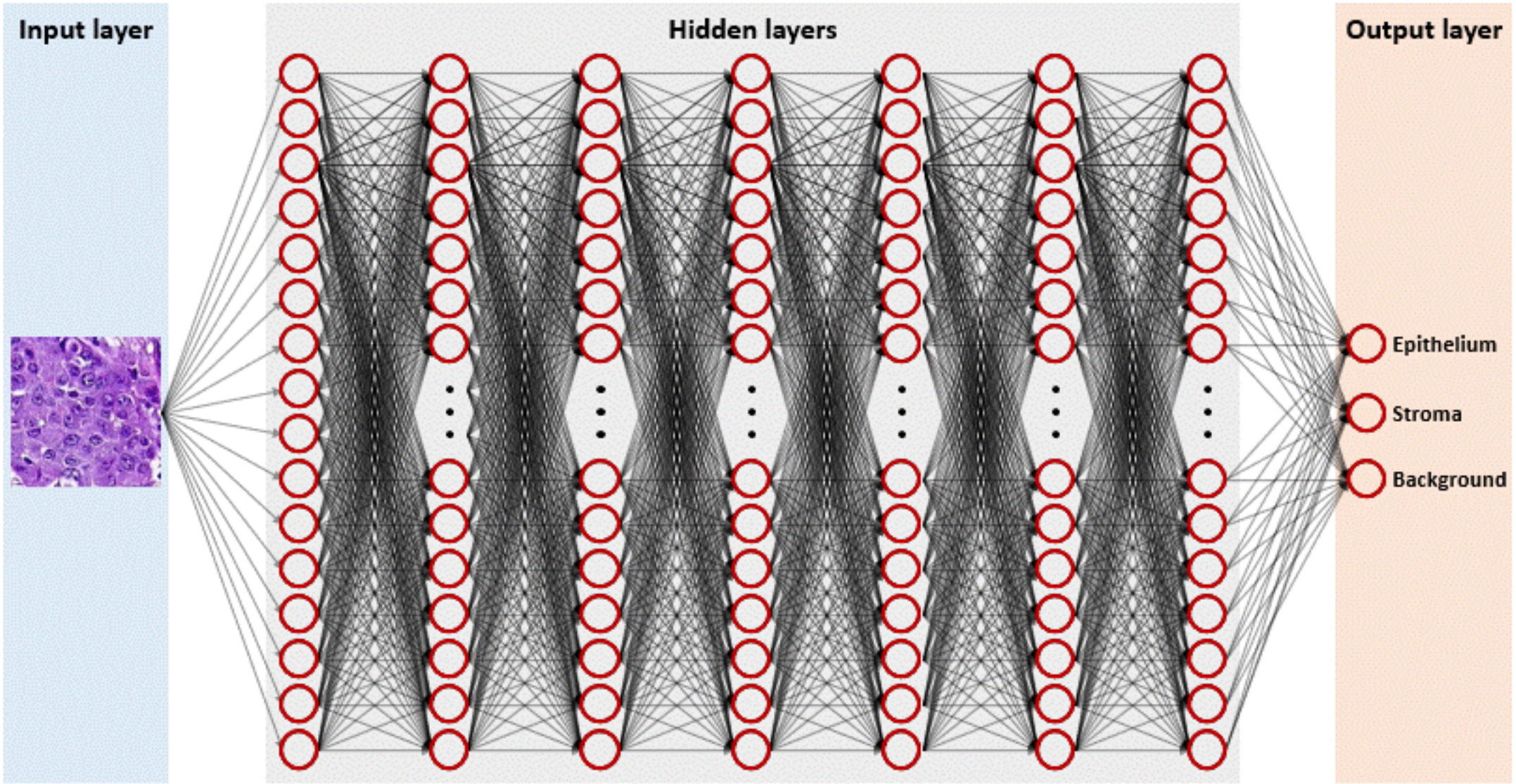
2) Large amounts of training data

ImageNet Large Scale Visual Recognition Challenge (2010 – 2017)
1000 classes – 10,000,000 images

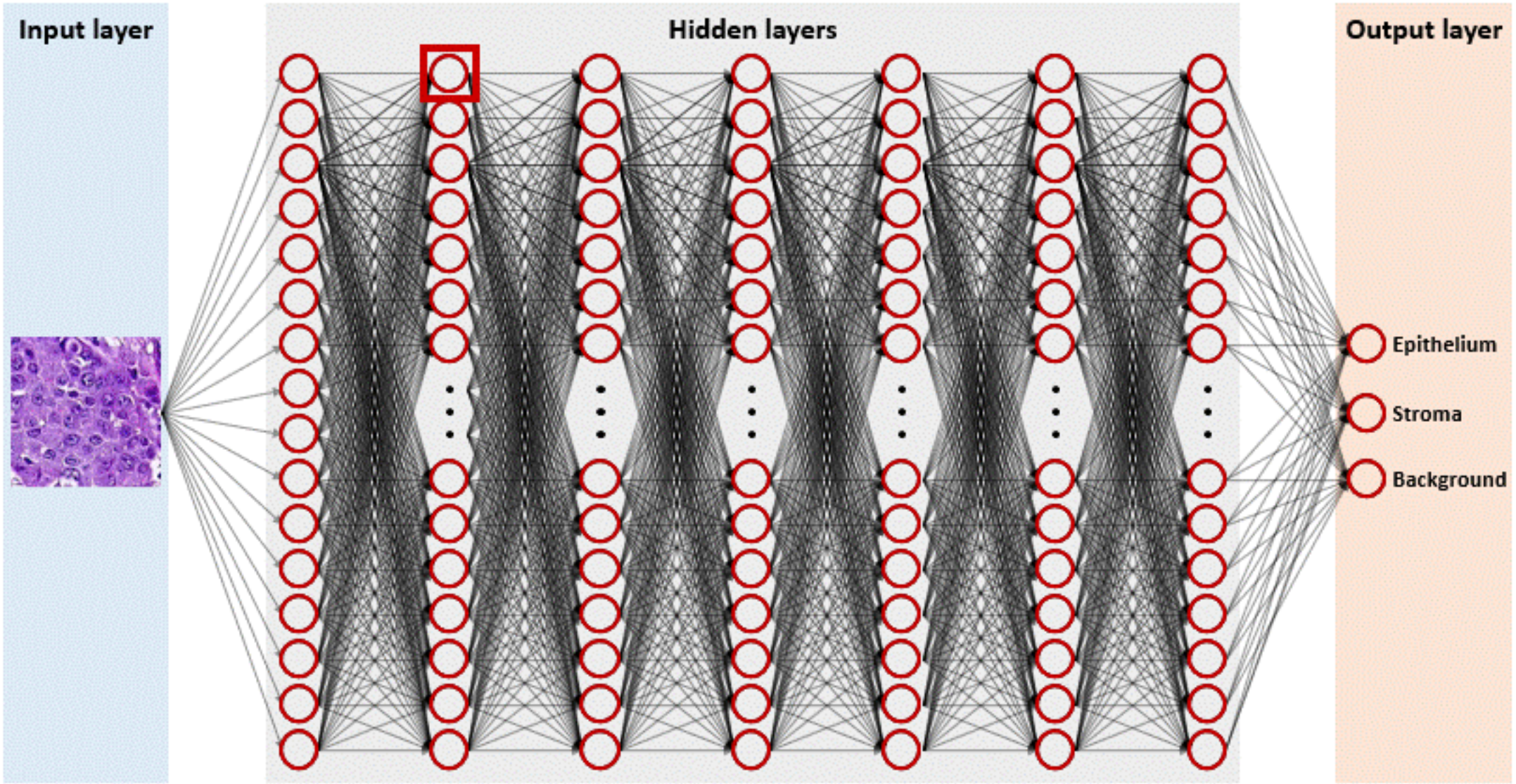
Deep Convolutional Neural Networks



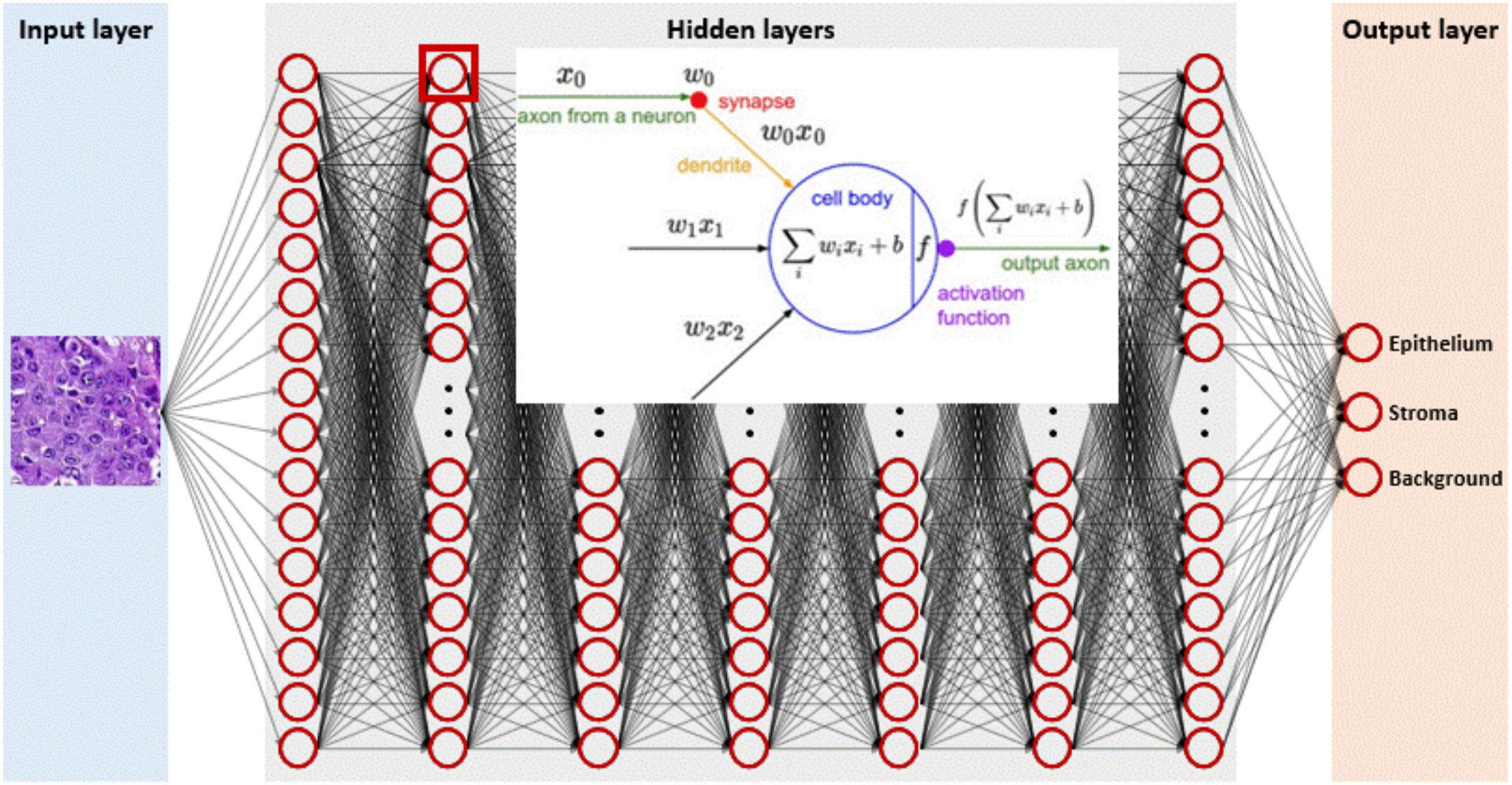
Deep Convolutional Neural Networks



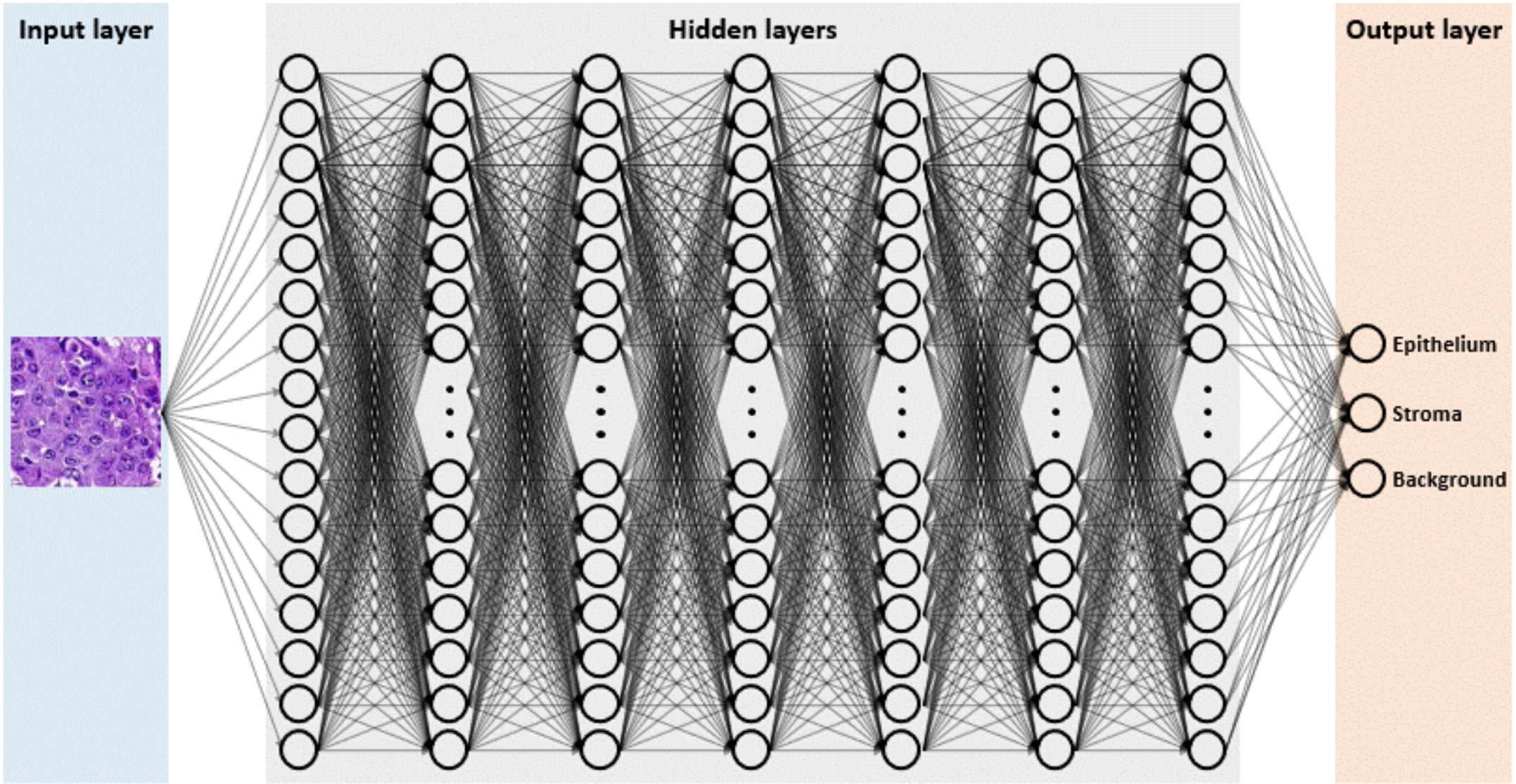
Deep Convolutional Neural Networks



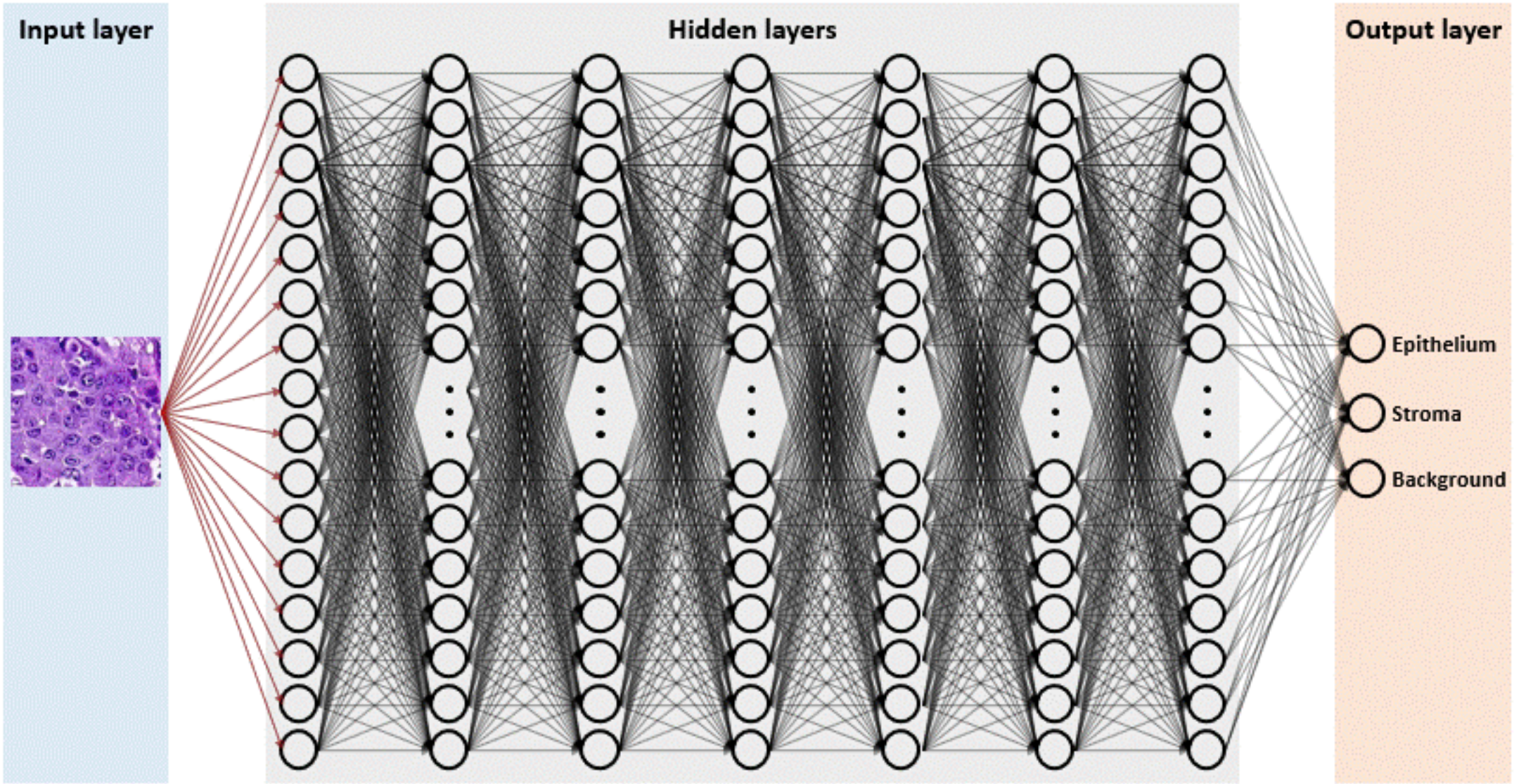
Deep Convolutional Neural Networks



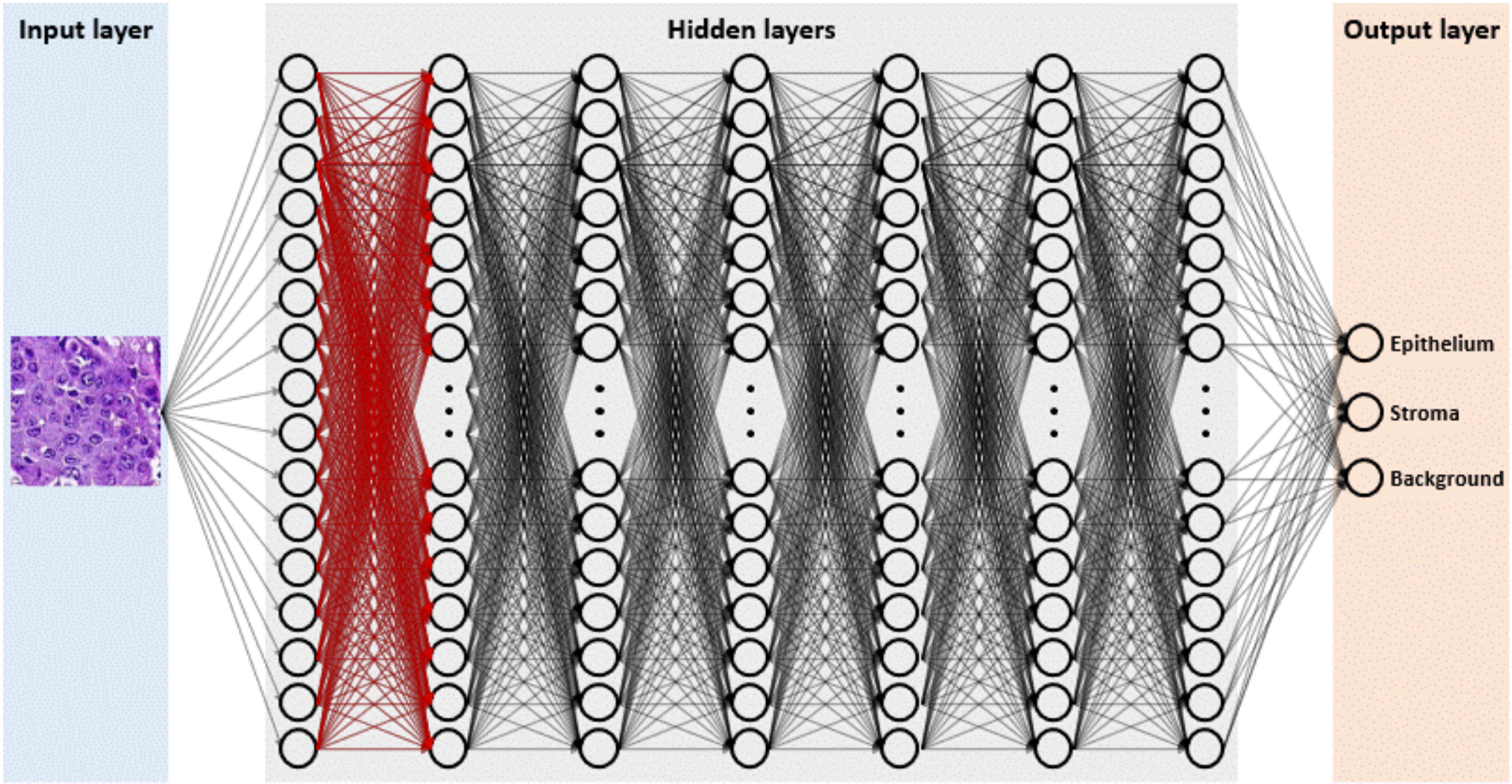
Deep Convolutional Neural Networks



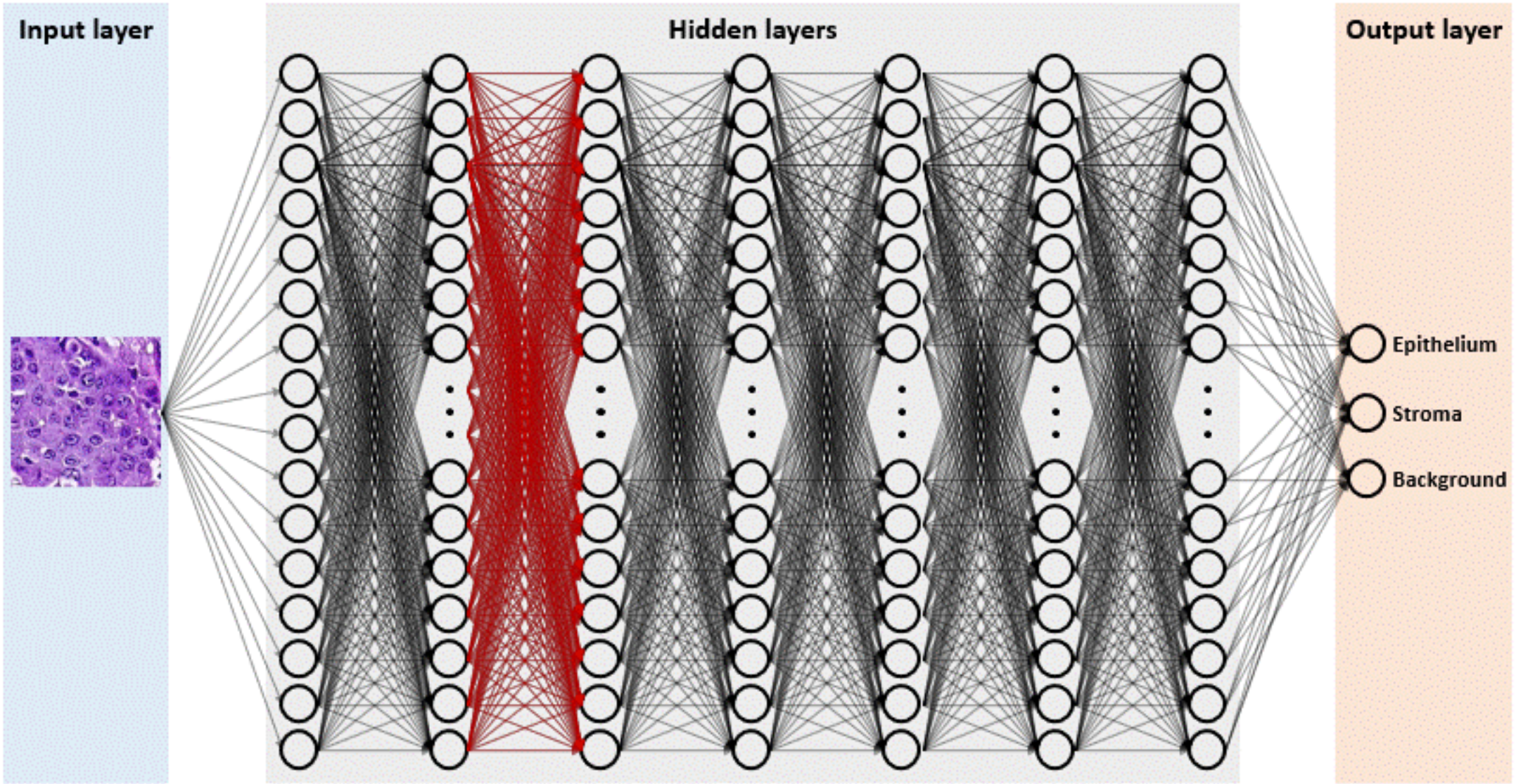
Deep Convolutional Neural Networks



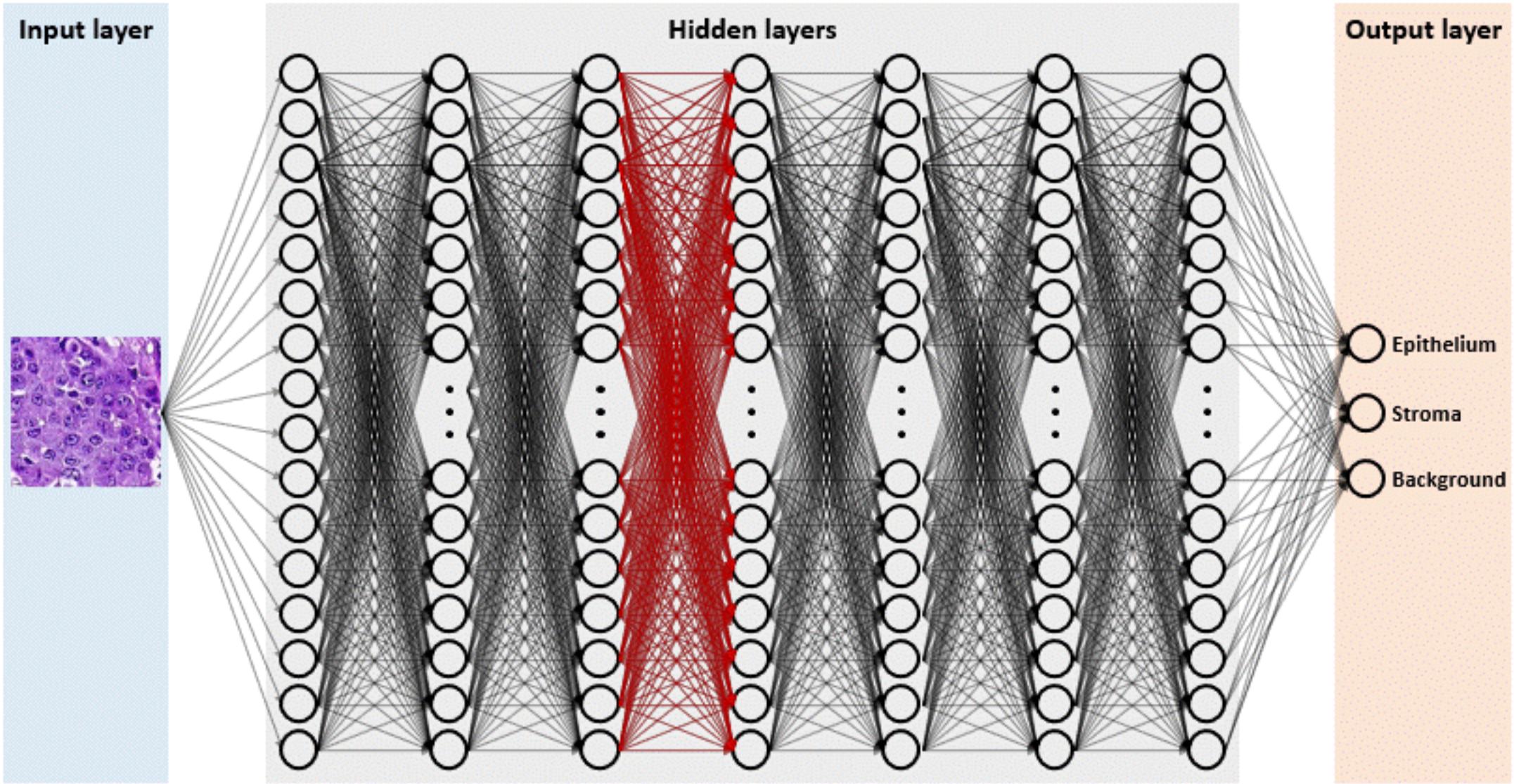
Deep Convolutional Neural Networks



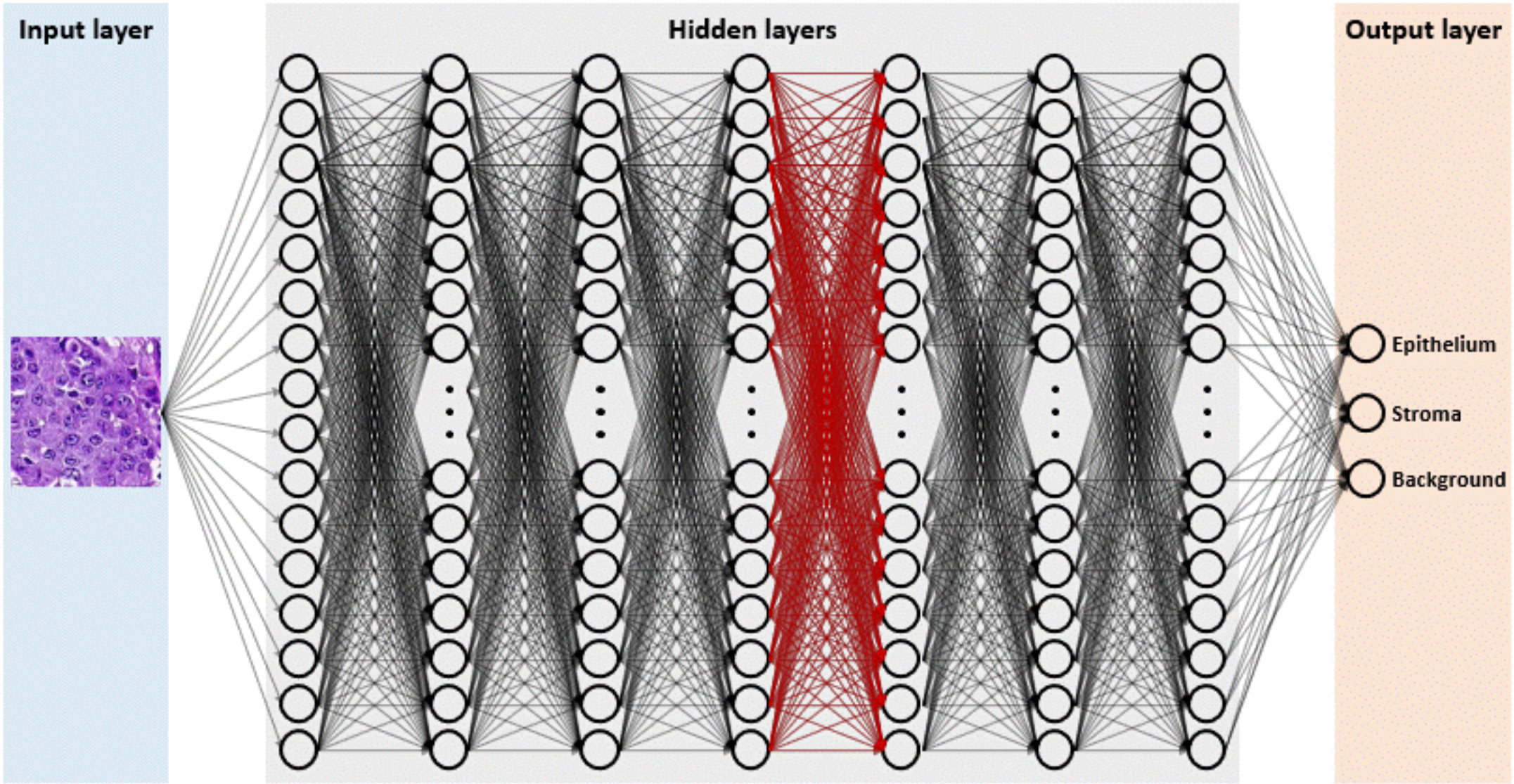
Deep Convolutional Neural Networks



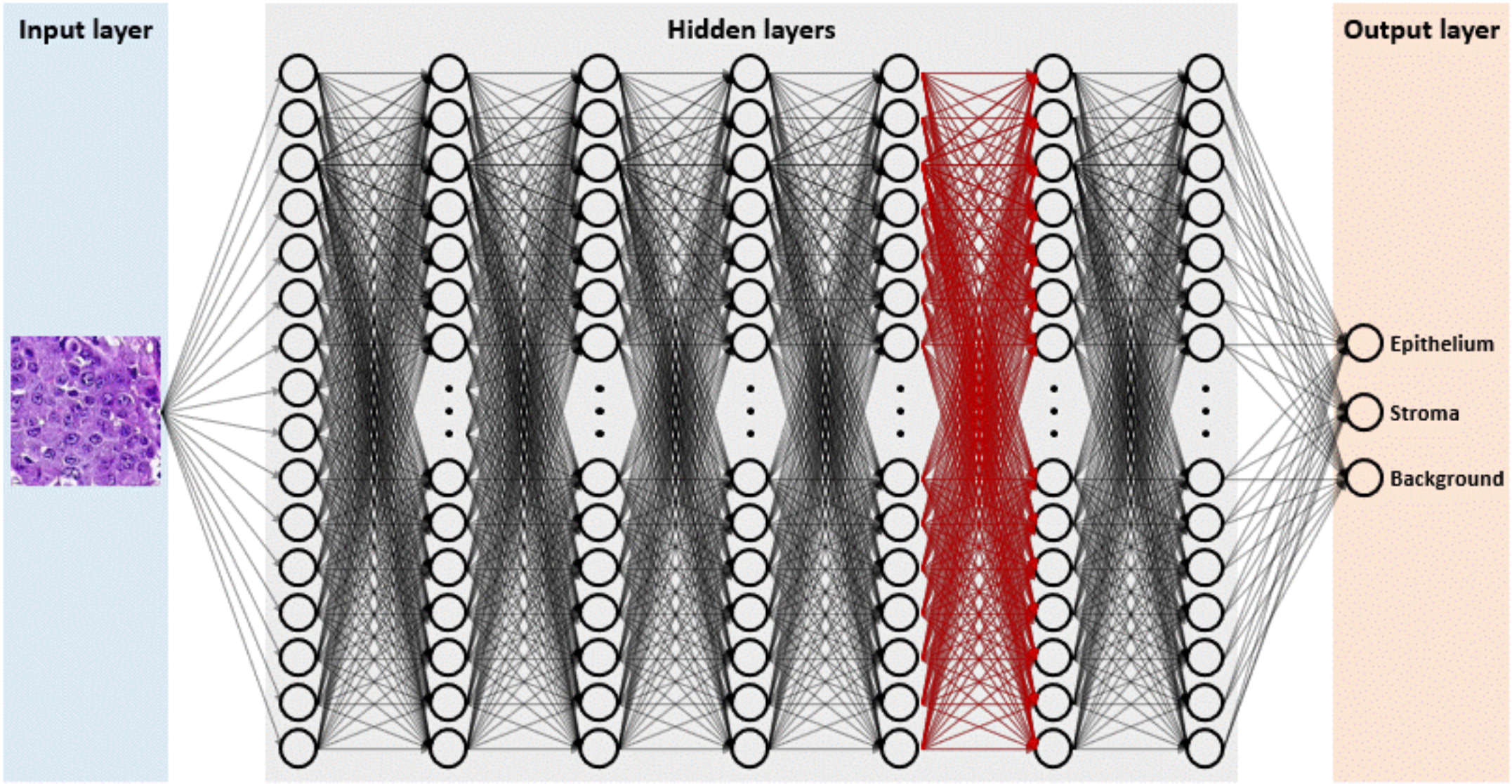
Deep Convolutional Neural Networks



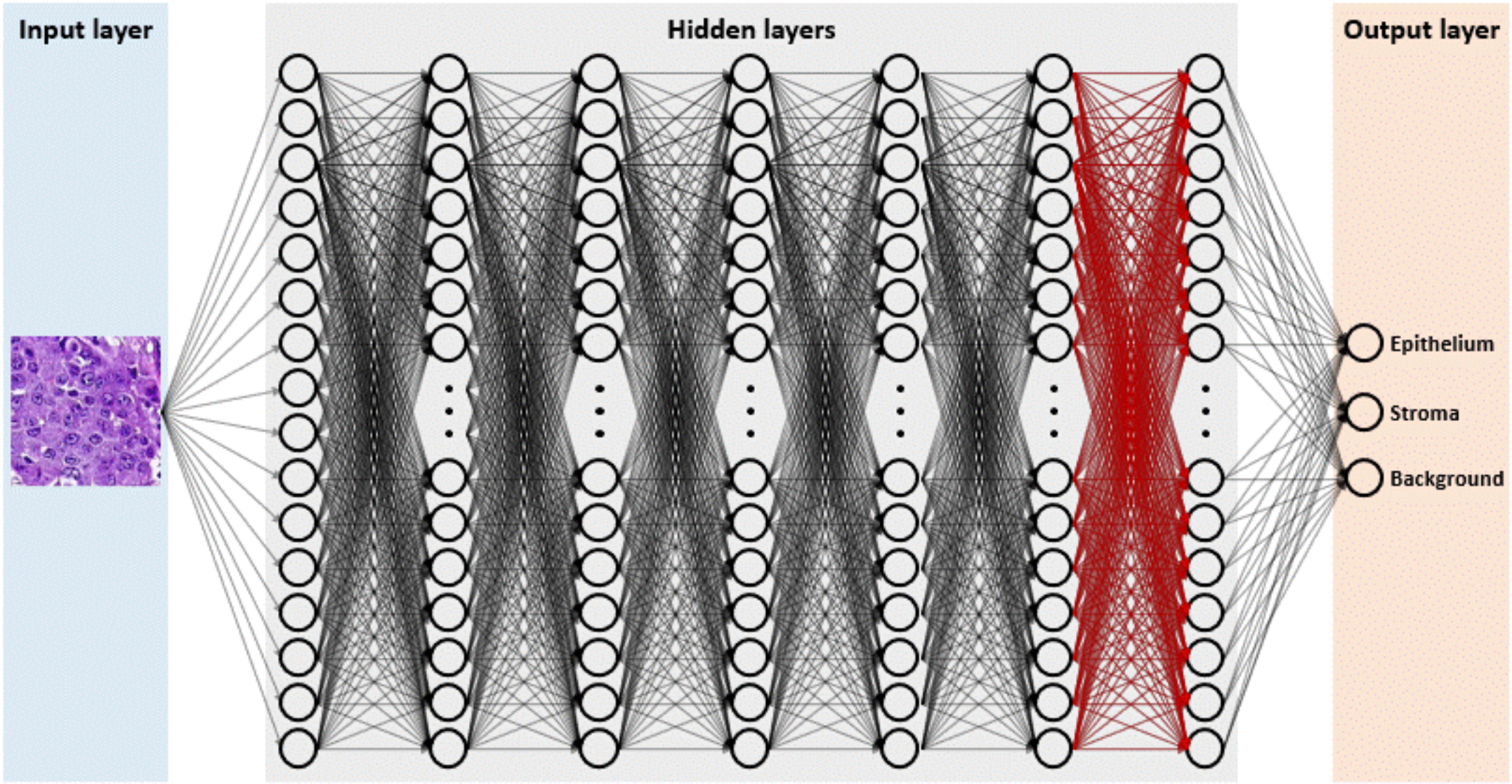
Deep Convolutional Neural Networks



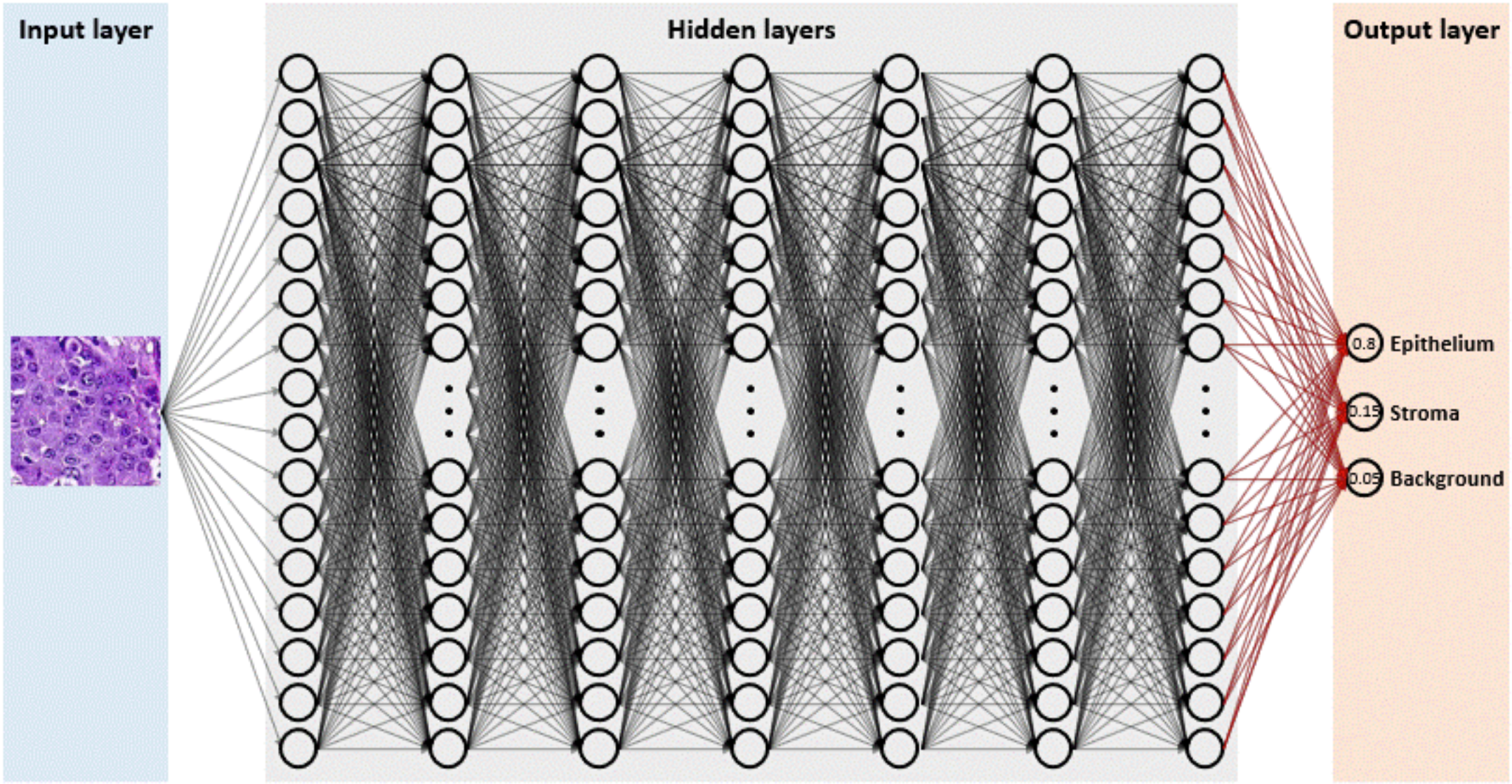
Deep Convolutional Neural Networks



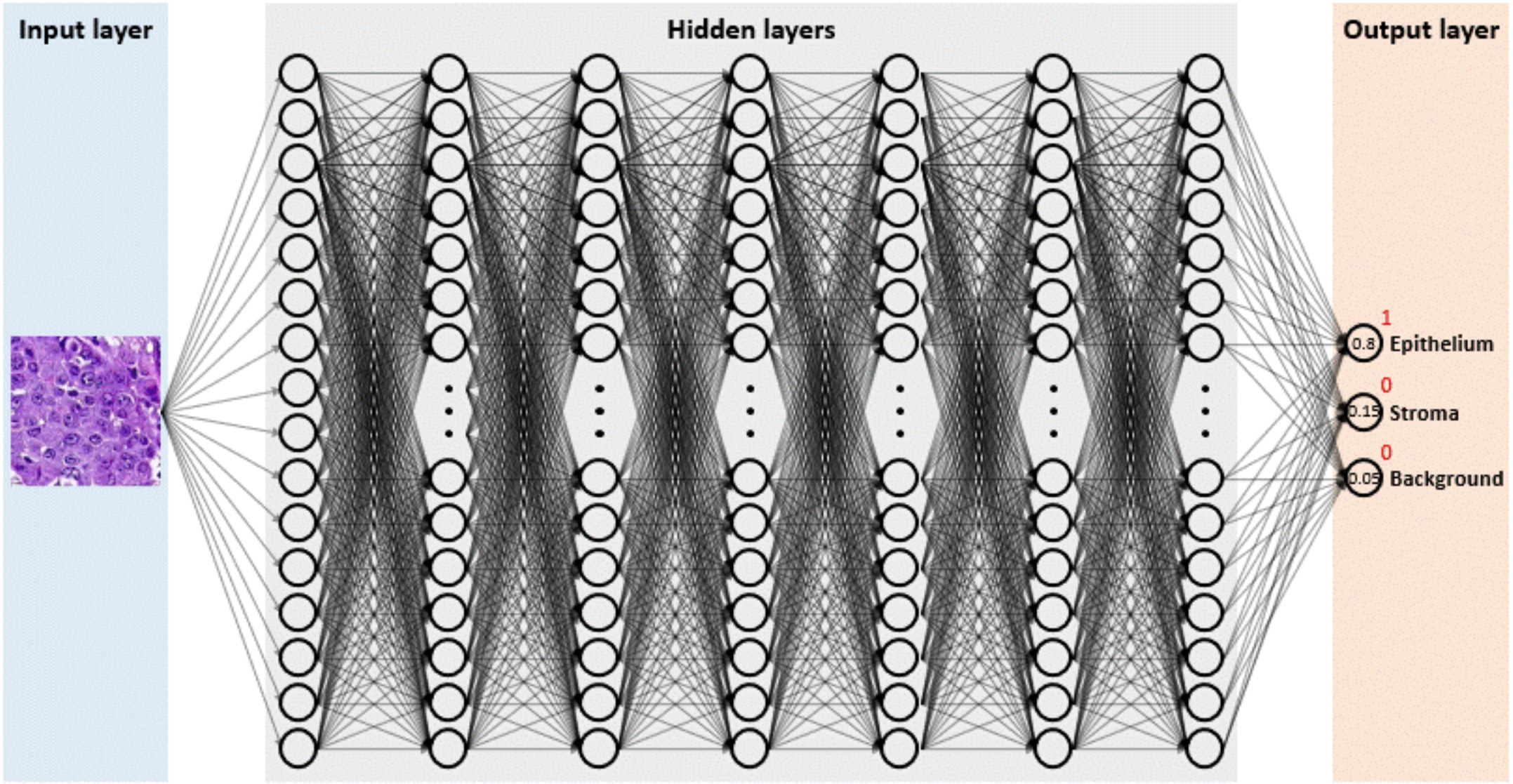
Deep Convolutional Neural Networks



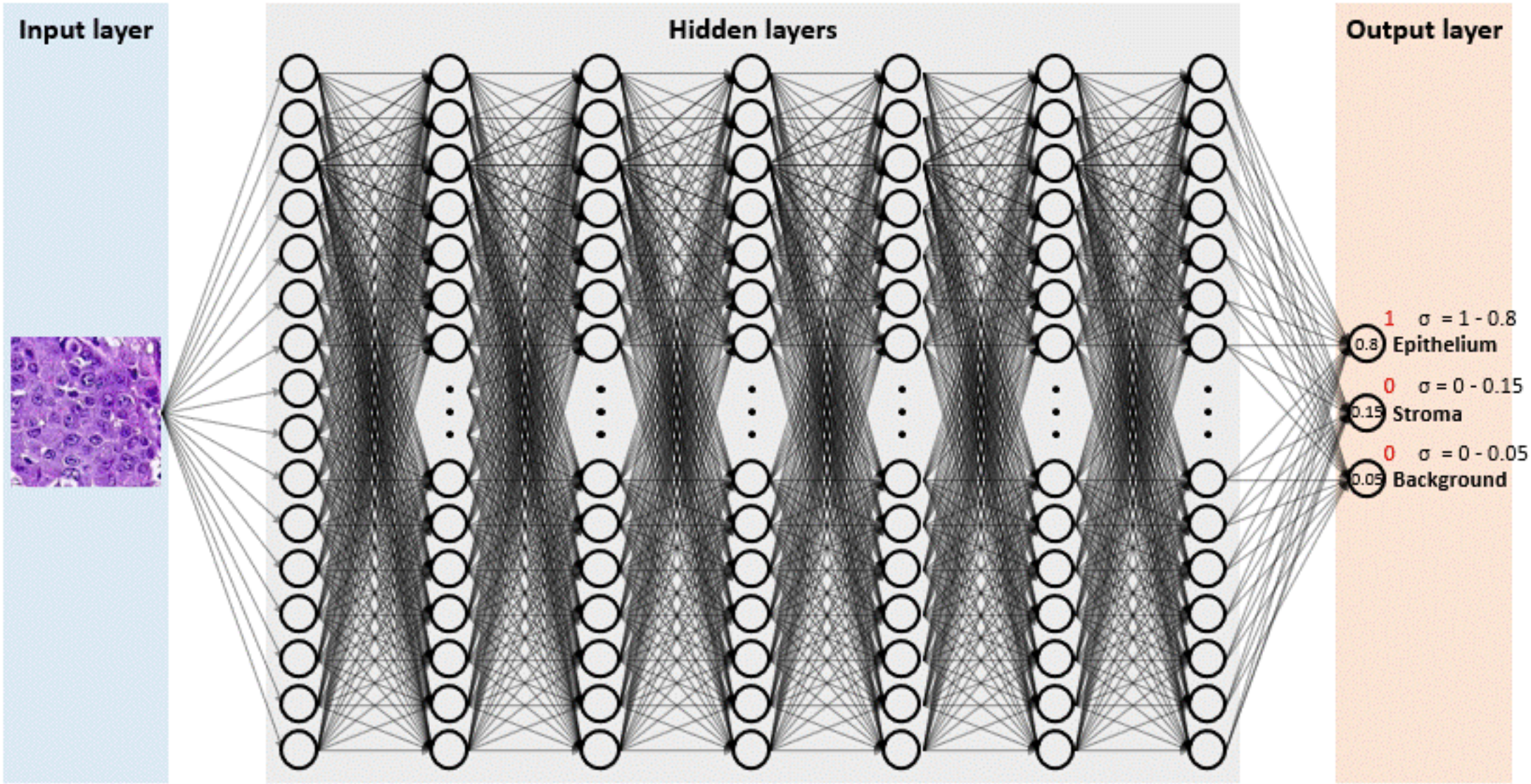
Deep Convolutional Neural Networks



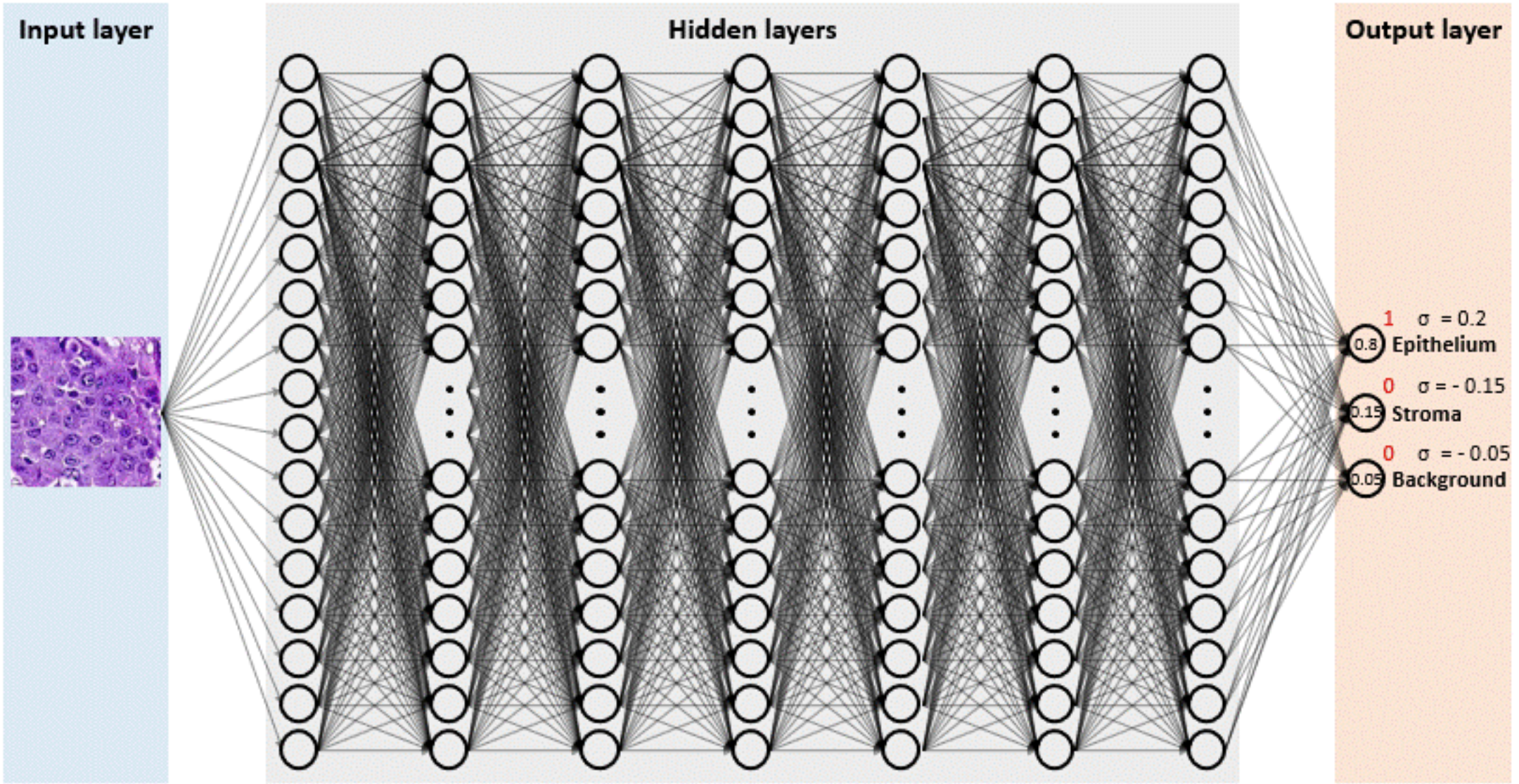
Deep Convolutional Neural Networks



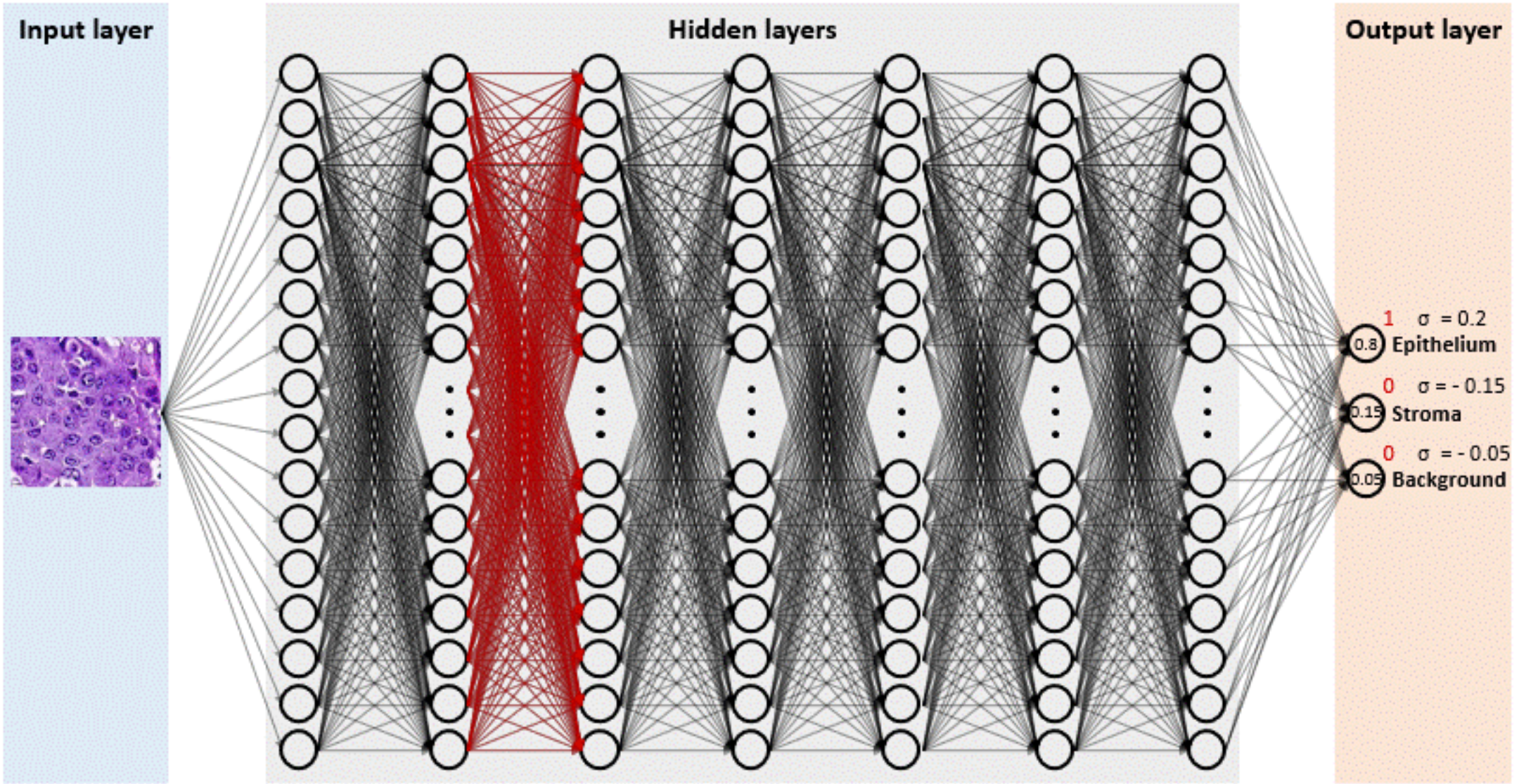
Deep Convolutional Neural Networks



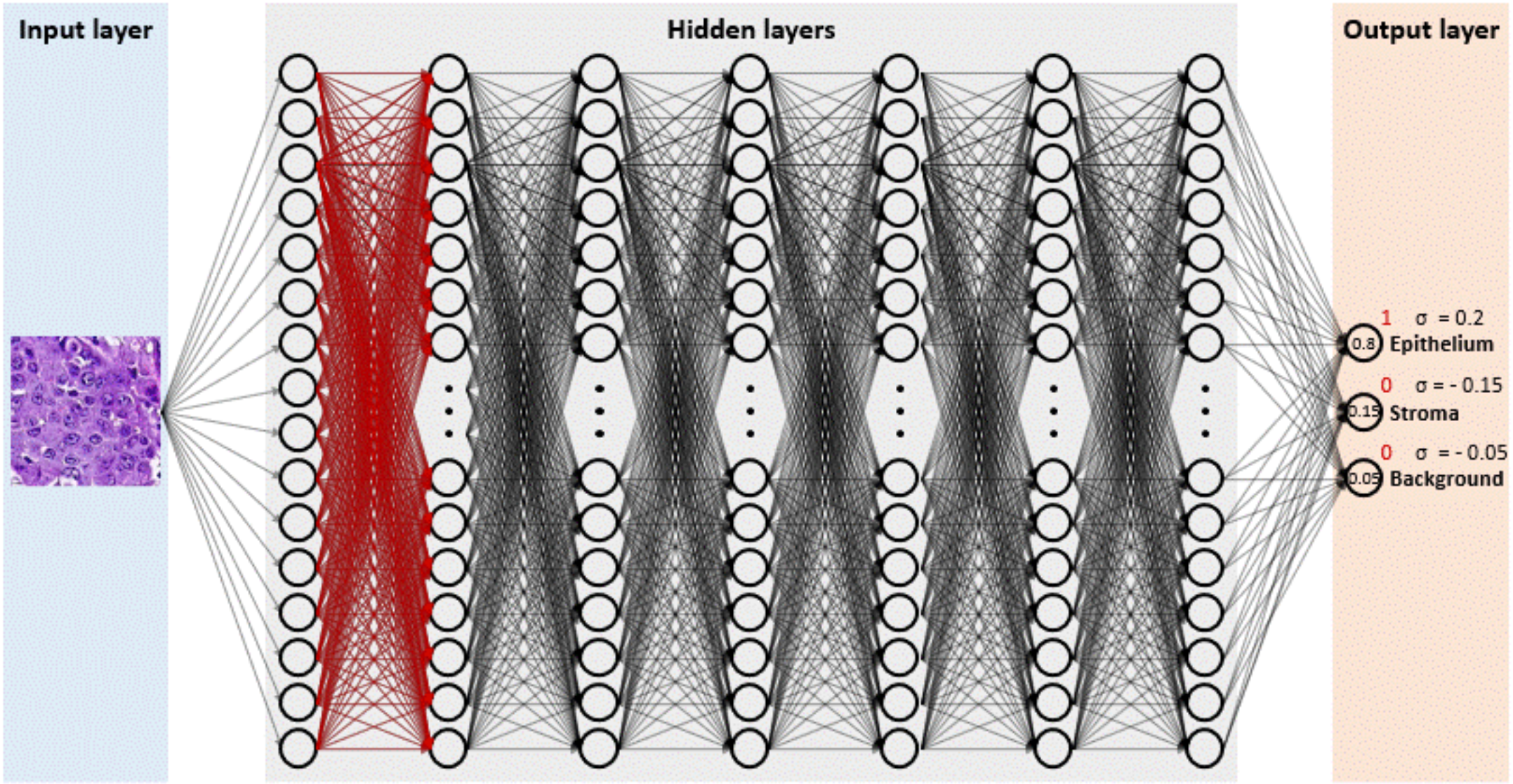
Deep Convolutional Neural Networks



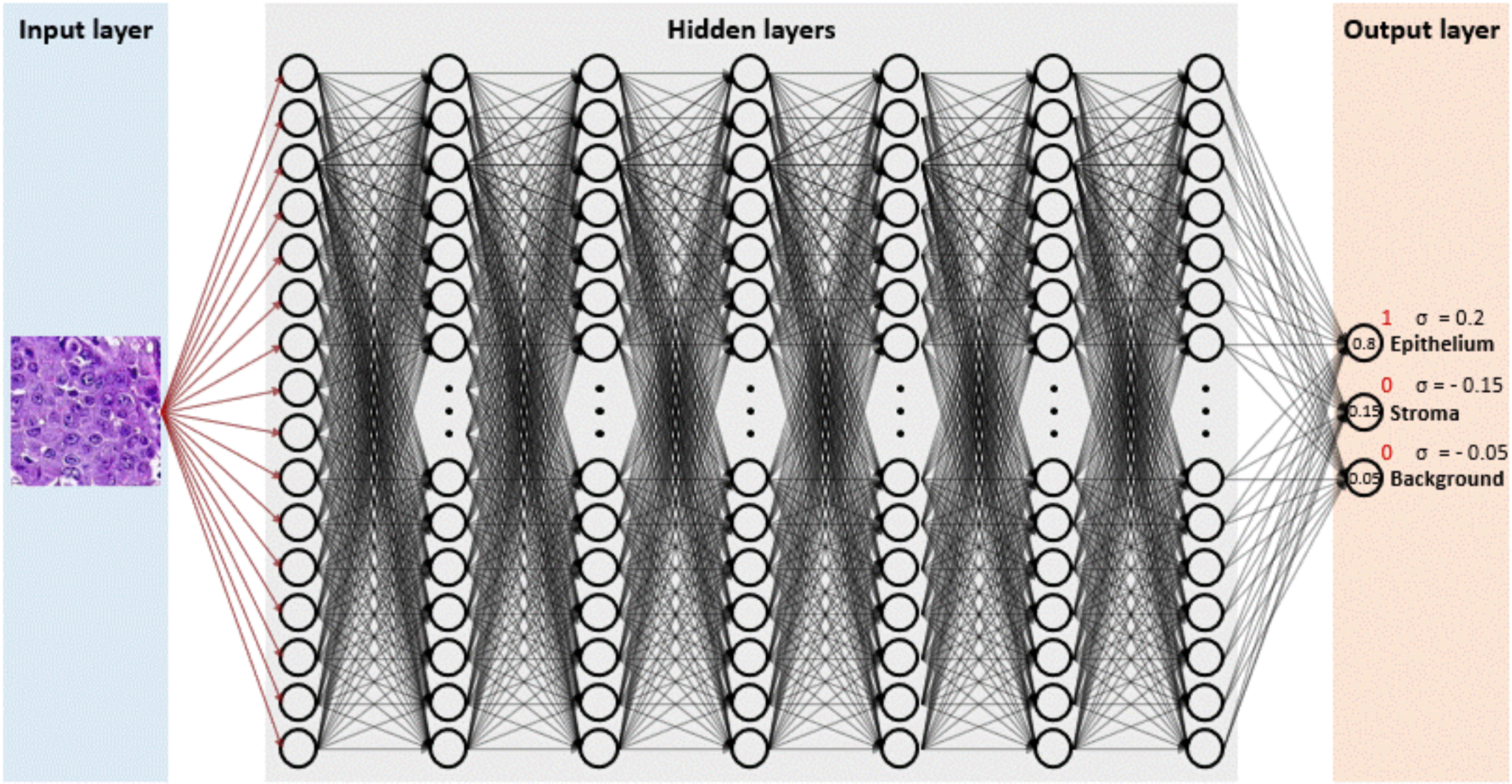
Deep Convolutional Neural Networks



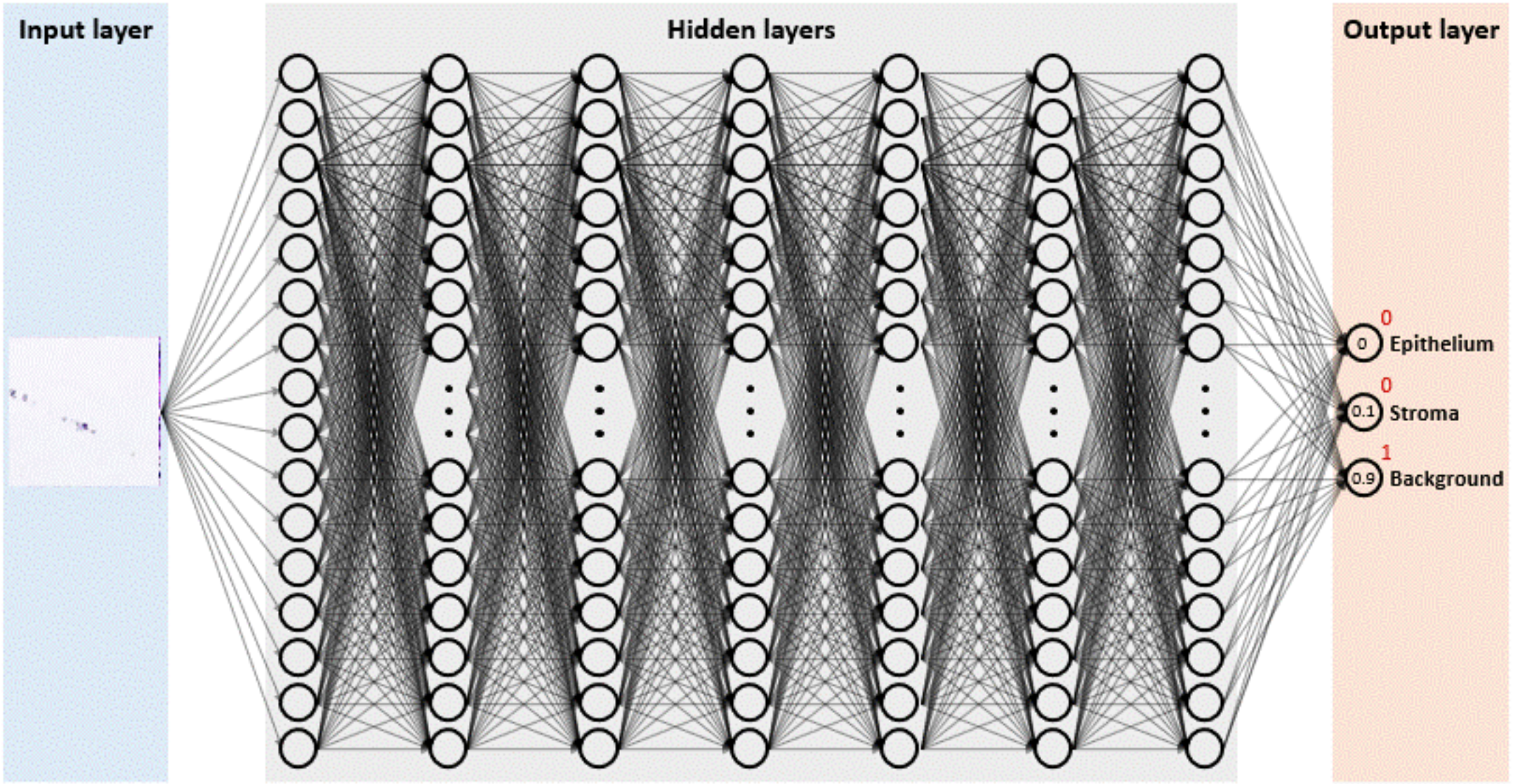
Deep Convolutional Neural Networks



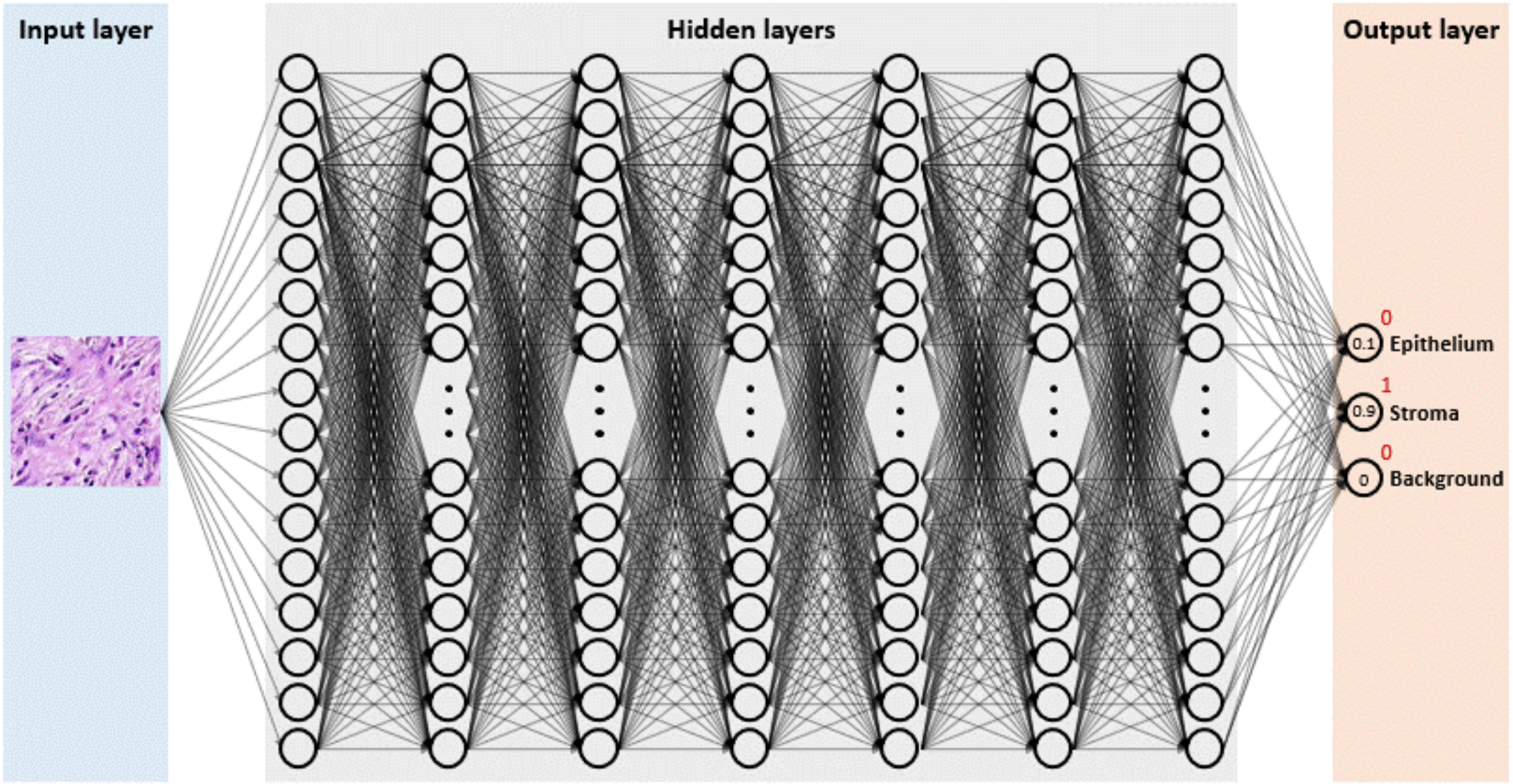
Deep Convolutional Neural Networks



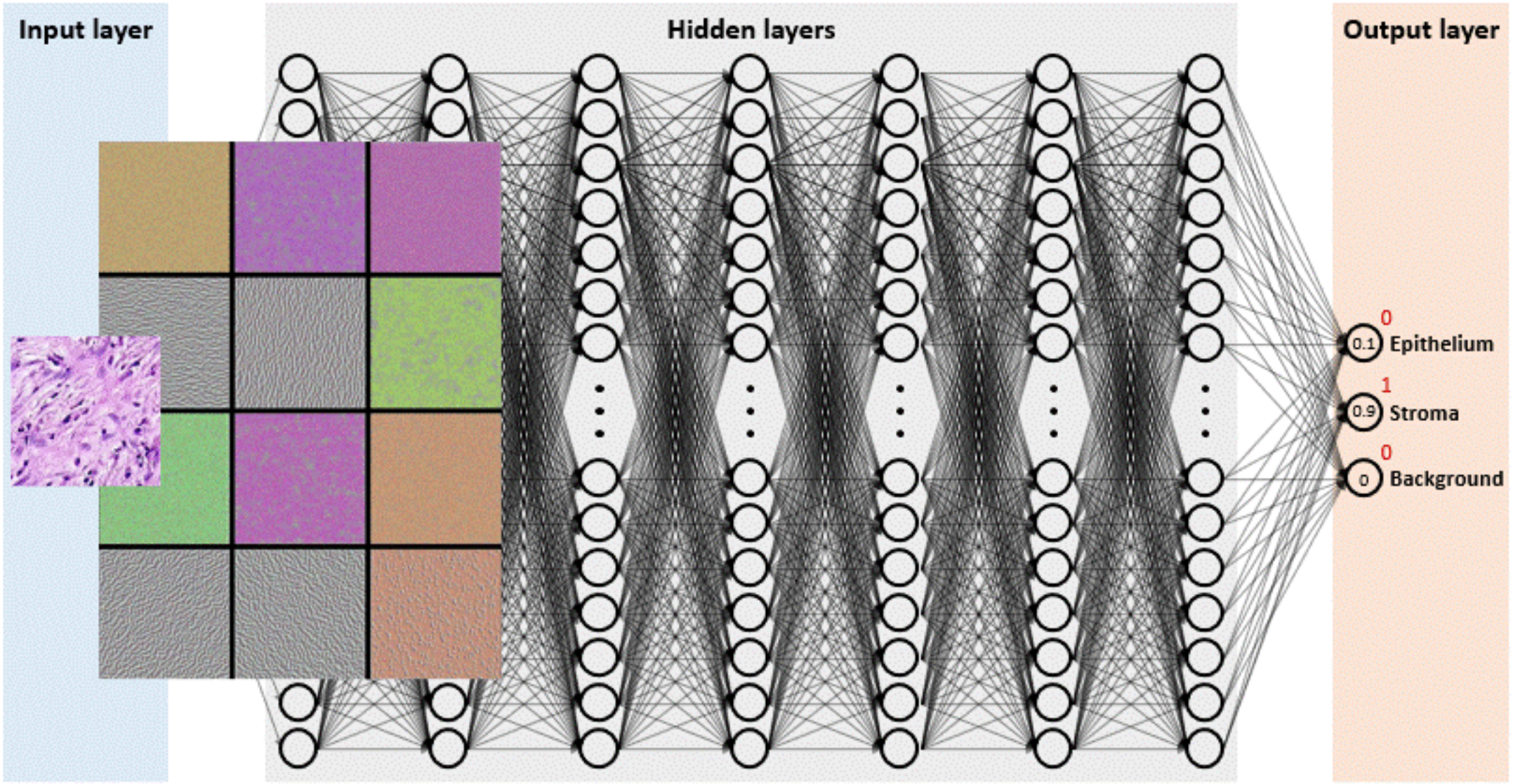
Deep Convolutional Neural Networks



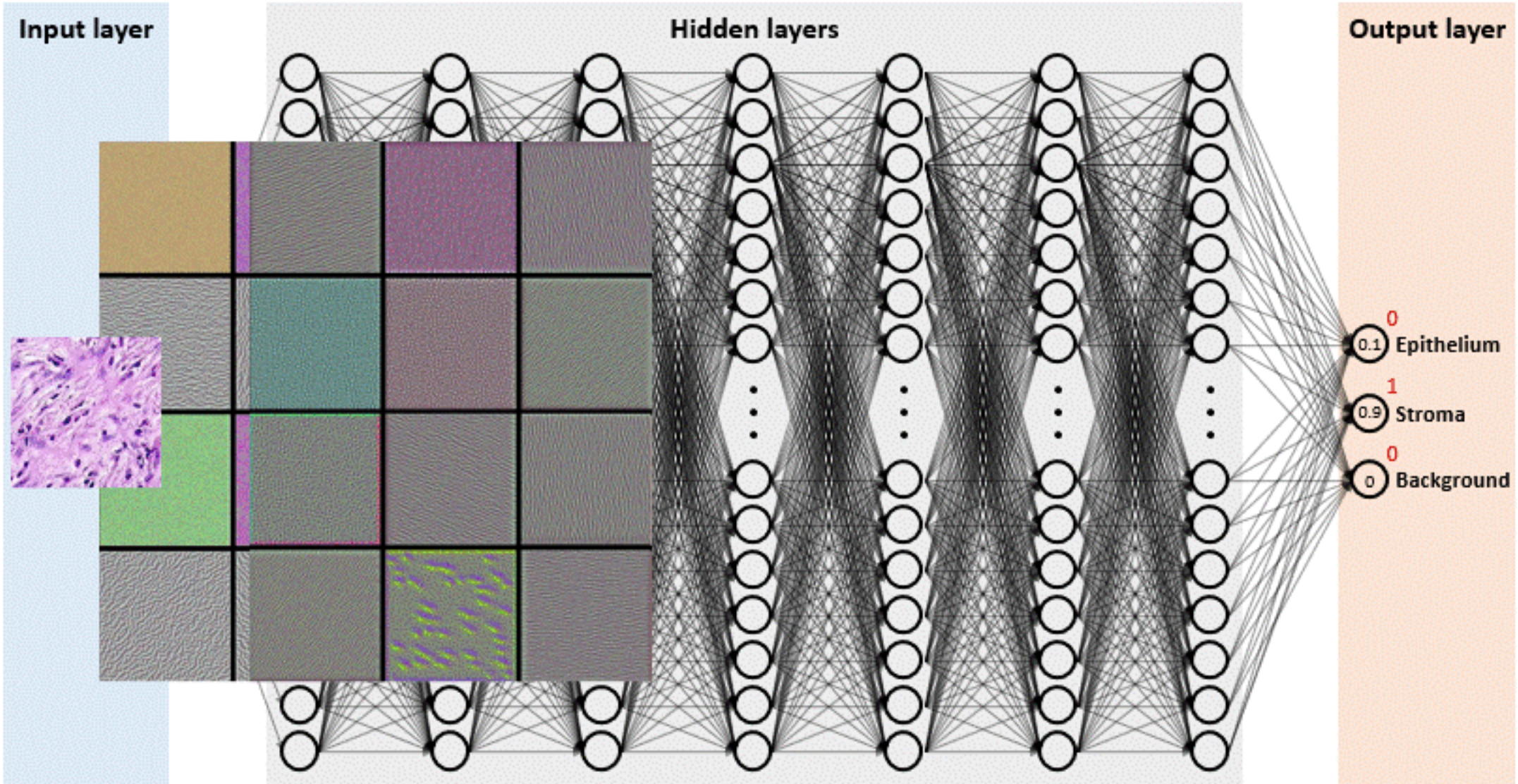
Deep Convolutional Neural Networks



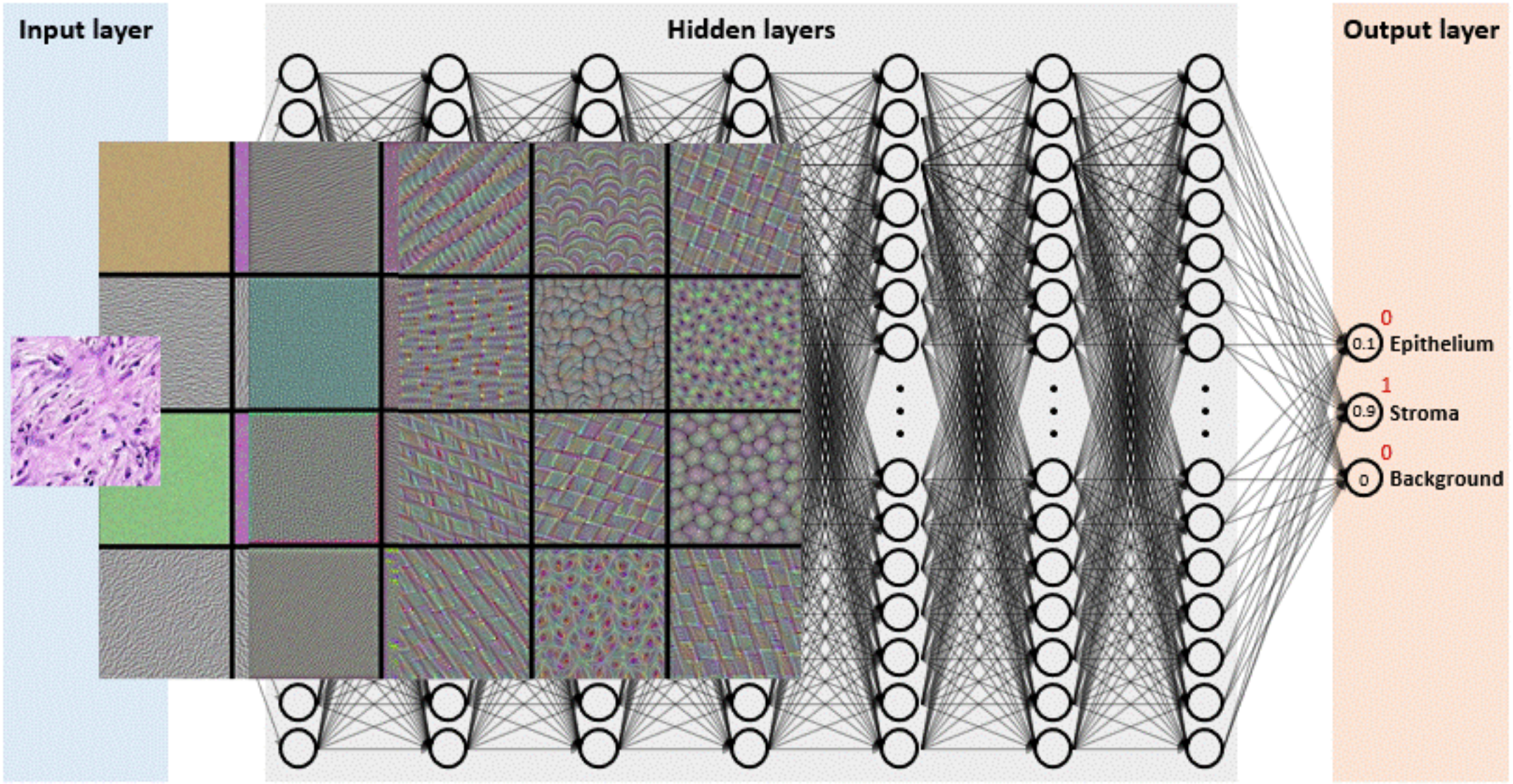
Deep Convolutional Neural Networks



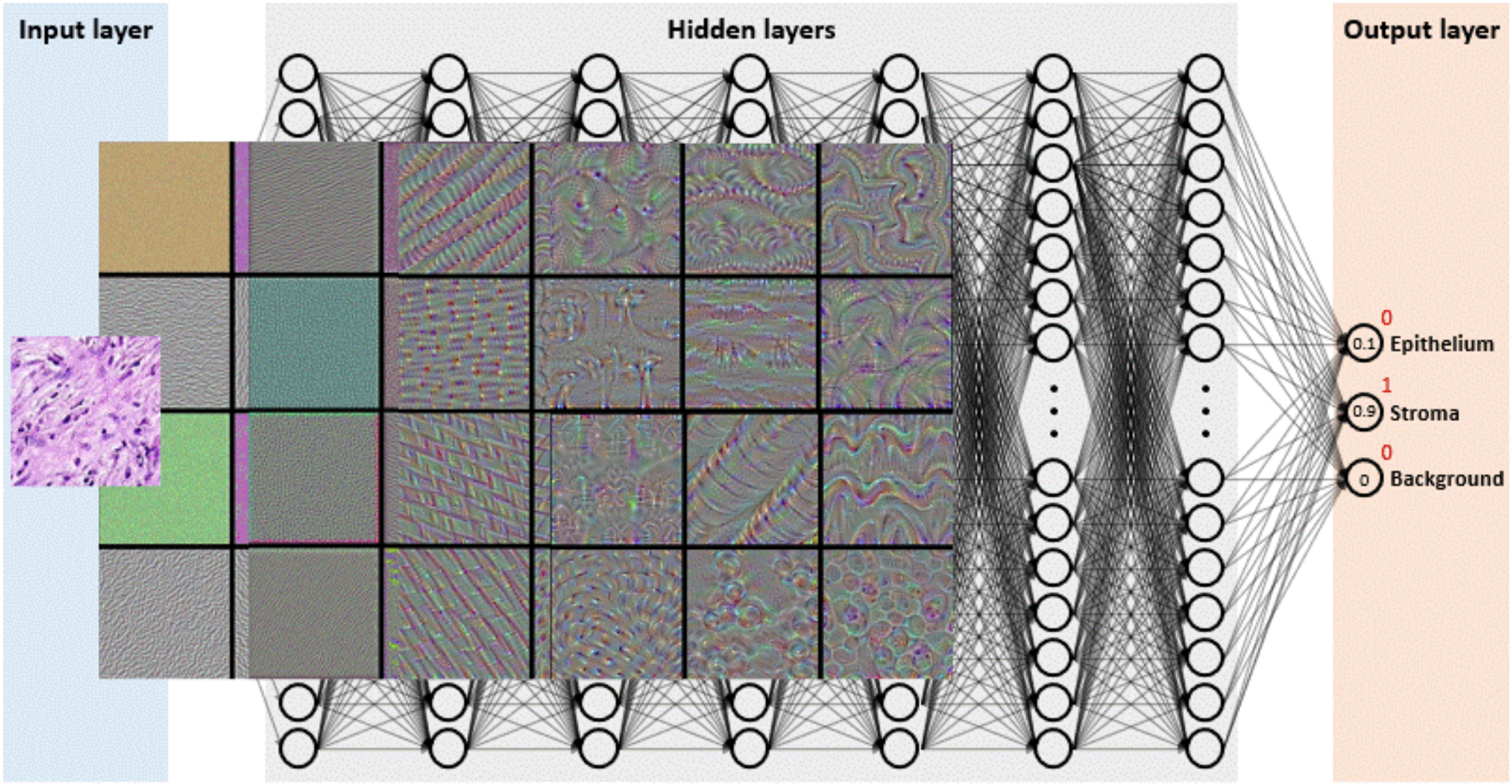
Deep Convolutional Neural Networks



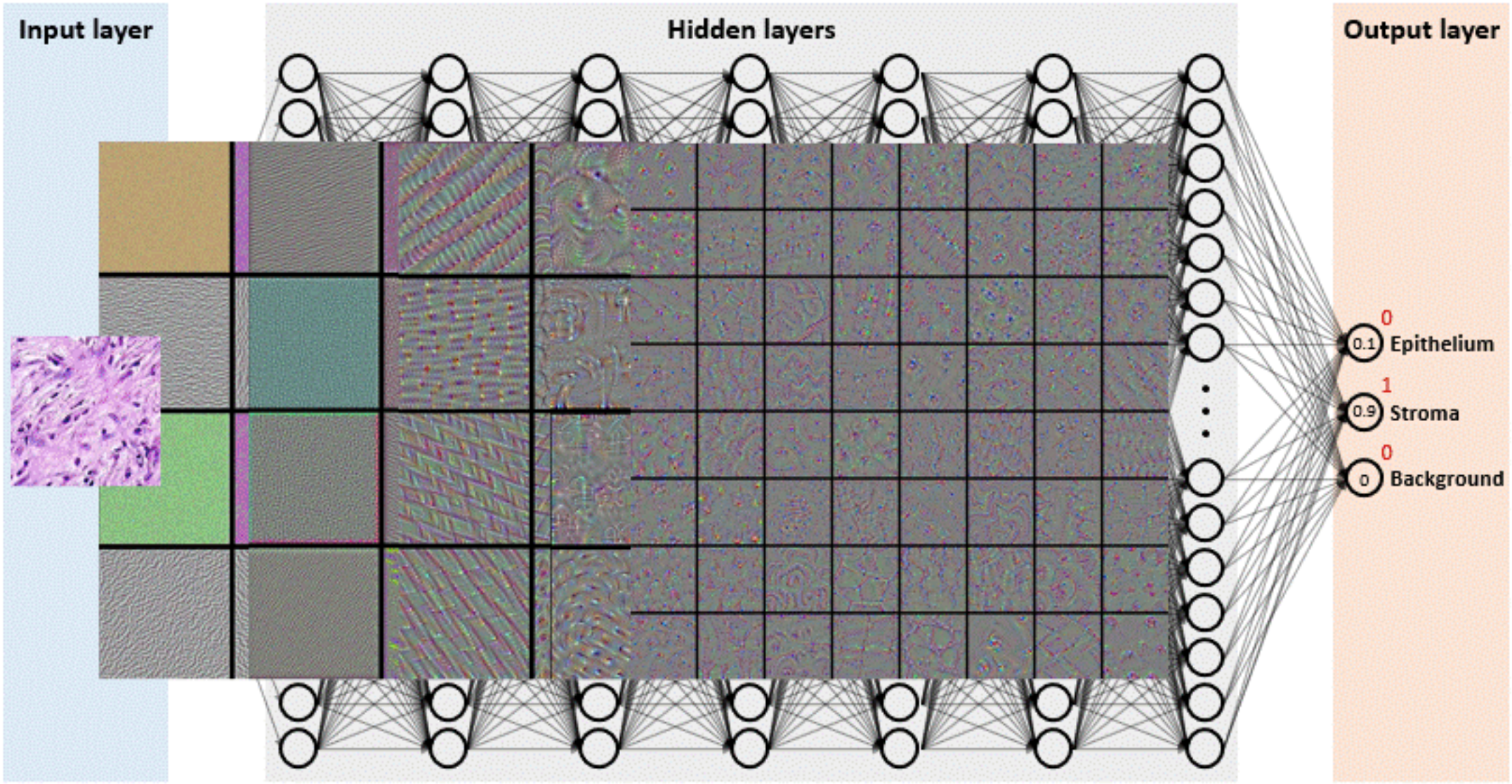
Deep Convolutional Neural Networks



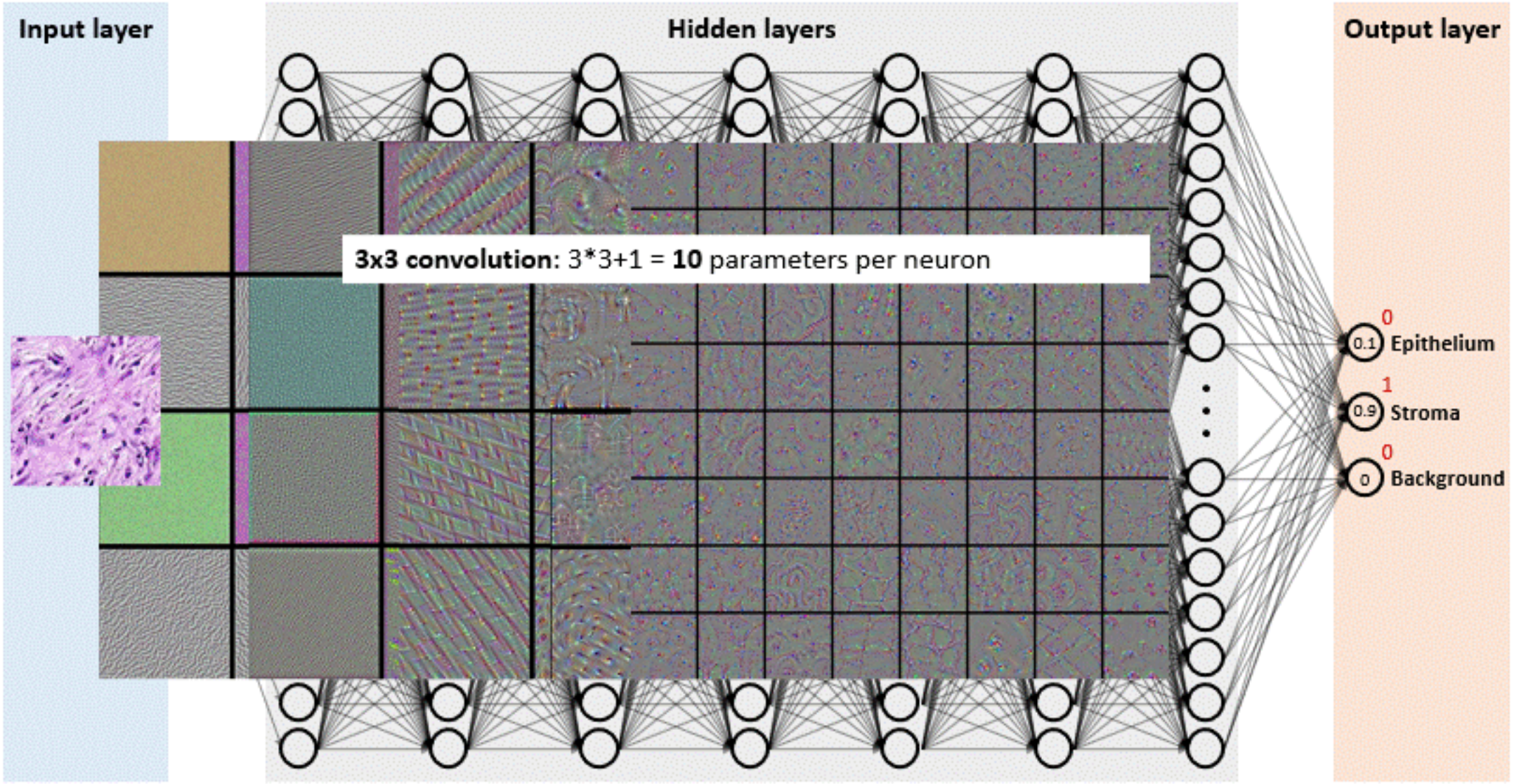
Deep Convolutional Neural Networks



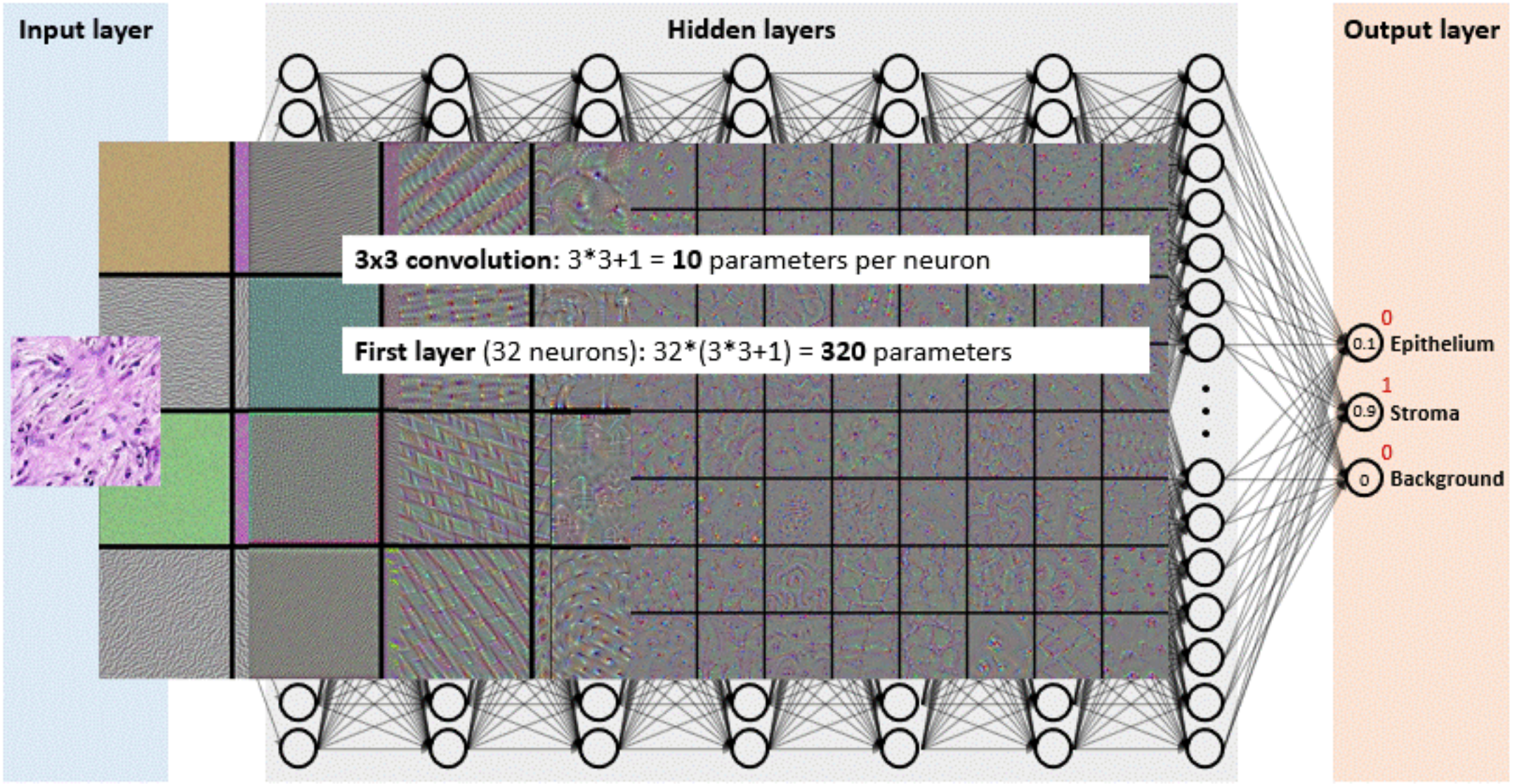
Deep Convolutional Neural Networks



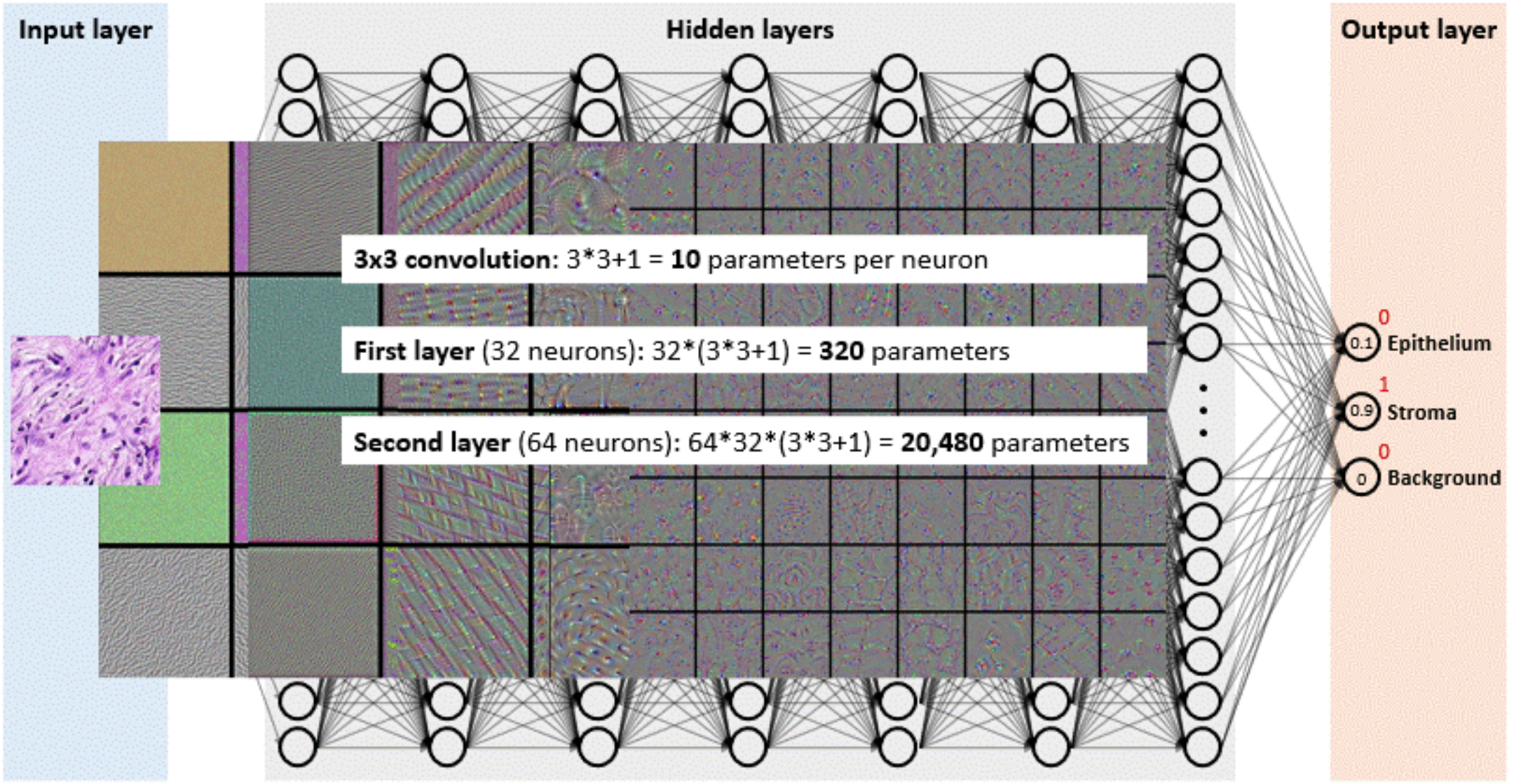
Deep Convolutional Neural Networks



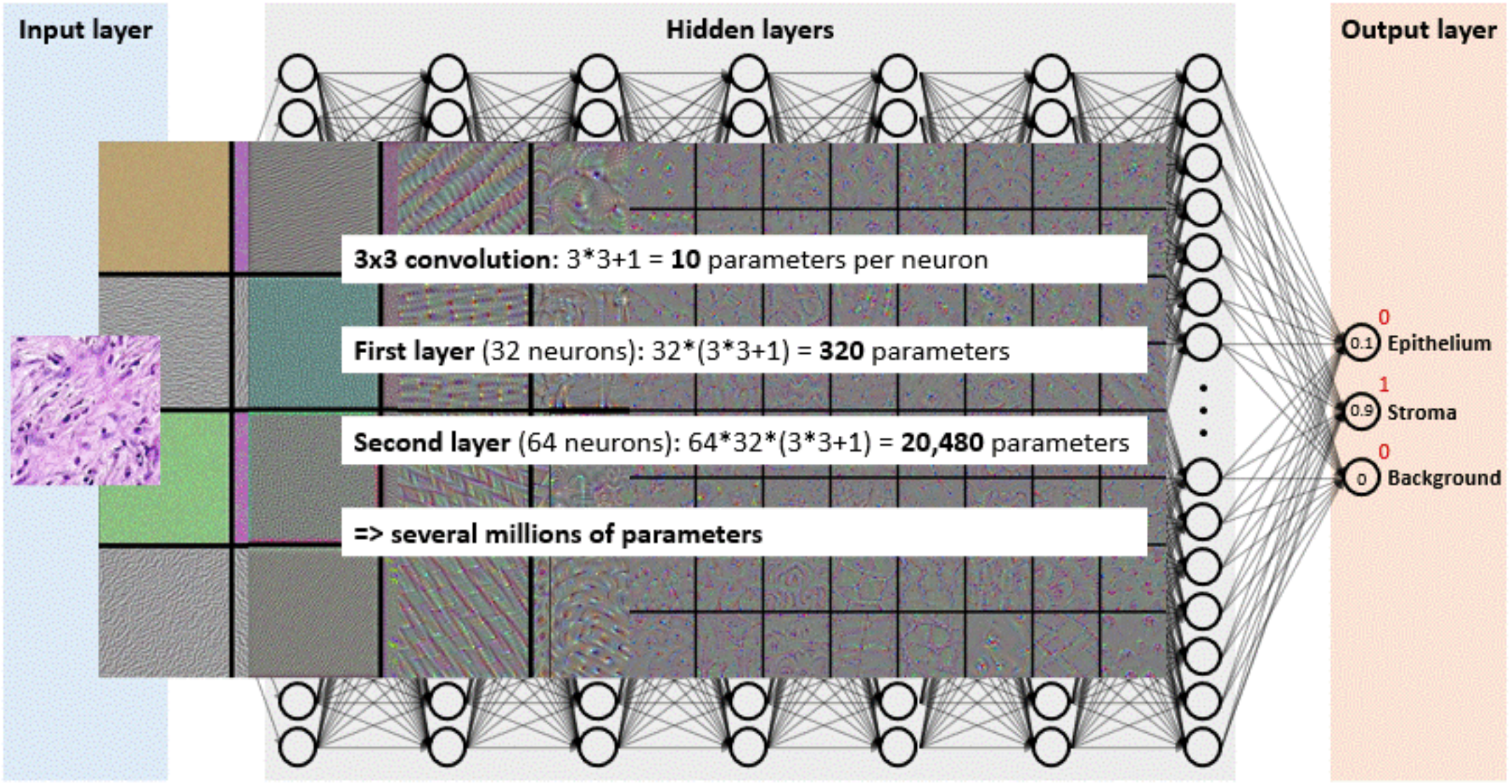
Deep Convolutional Neural Networks



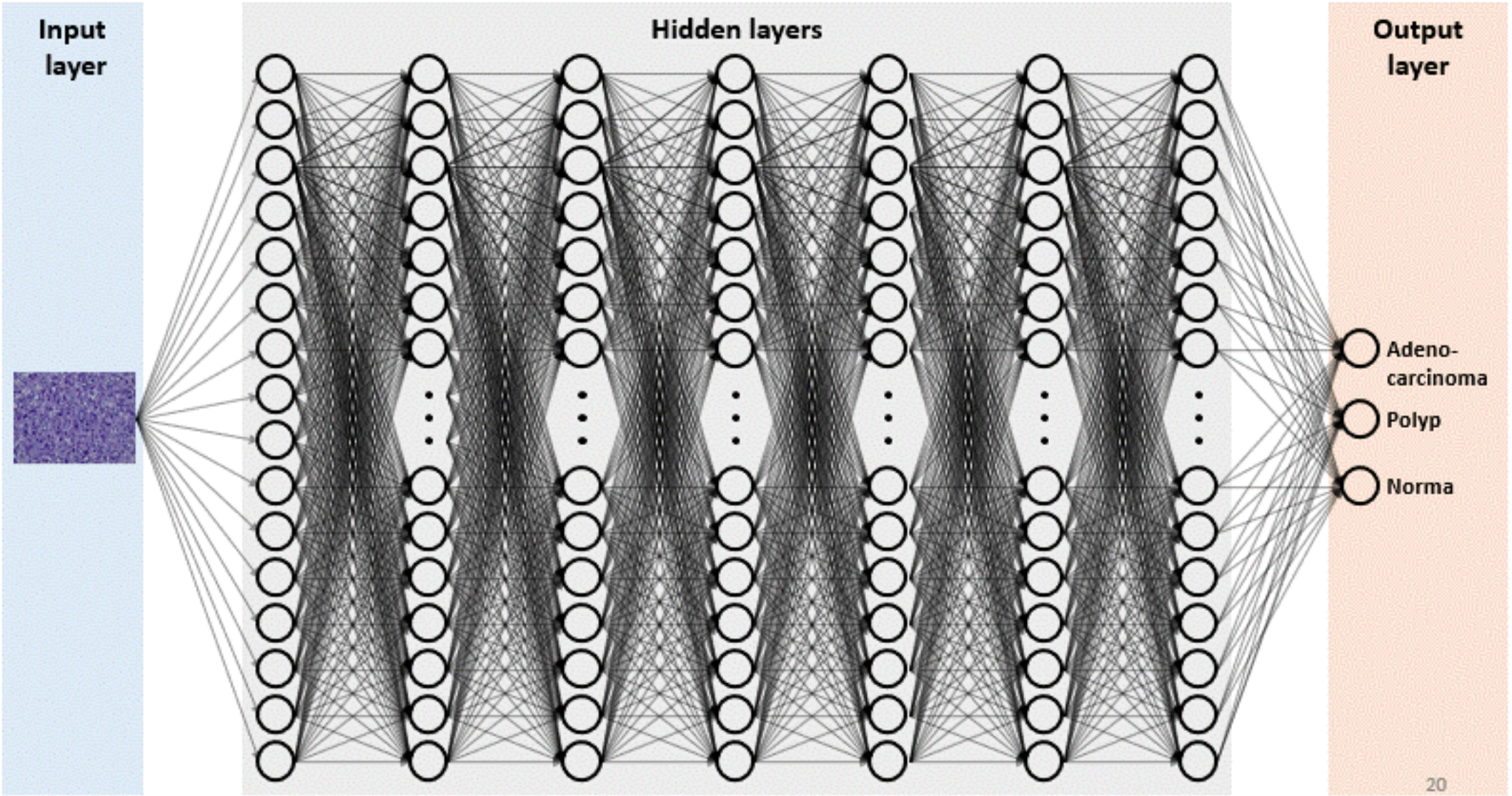
Deep Convolutional Neural Networks



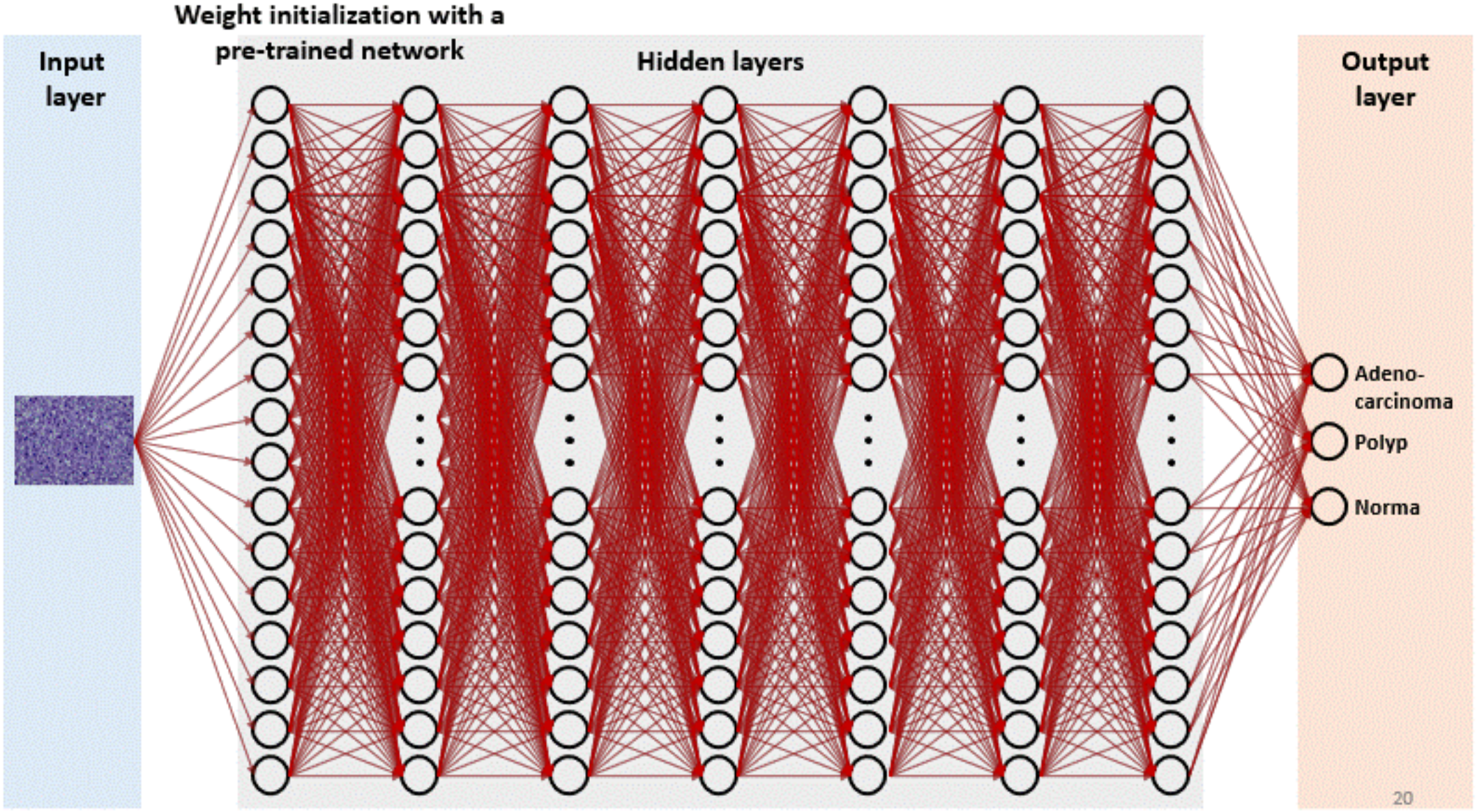
Deep Convolutional Neural Networks



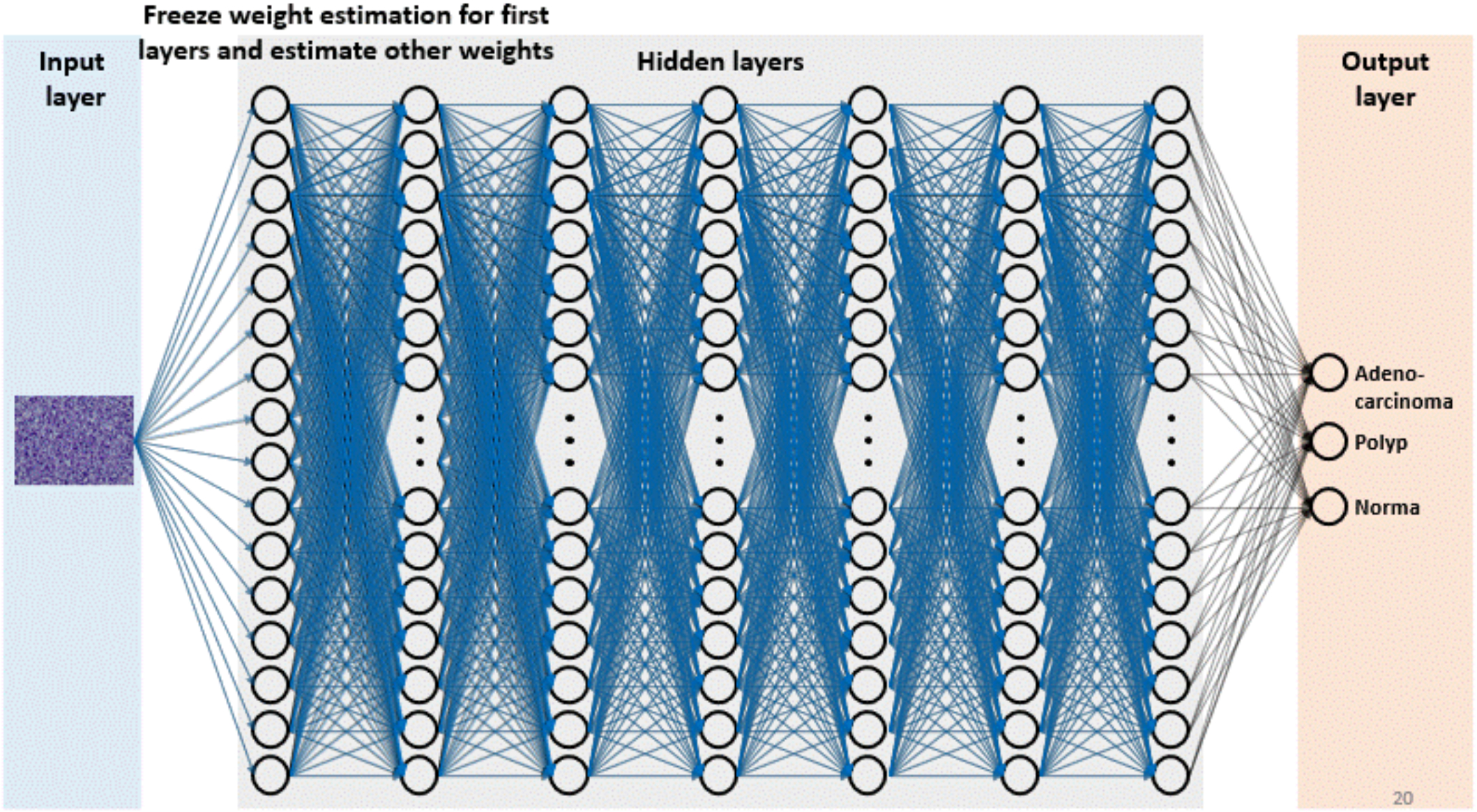
TRANSFER LEARNING



TRANSFER LEARNING

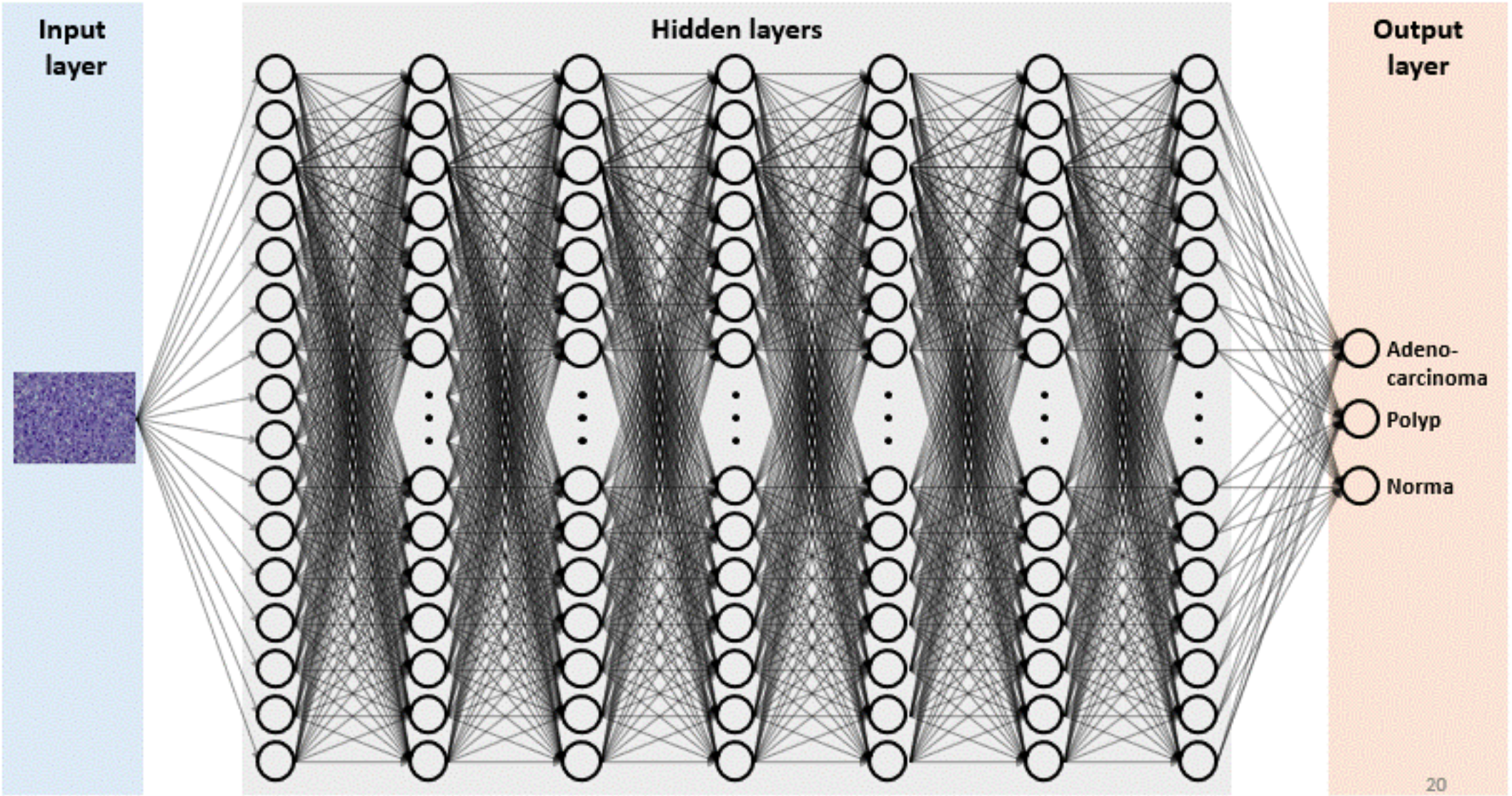


TRANSFER LEARNING



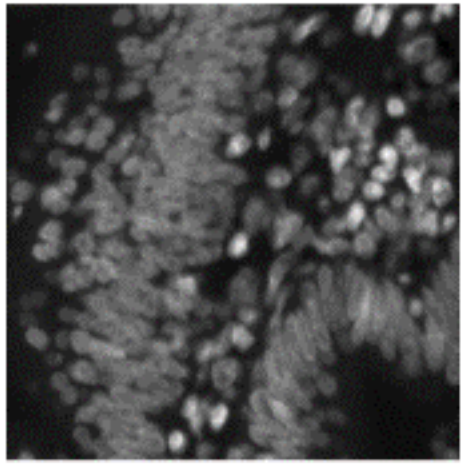
TRANSFER LEARNING

Estimate all weights

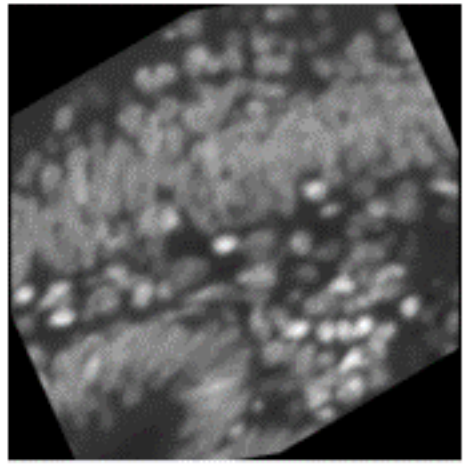


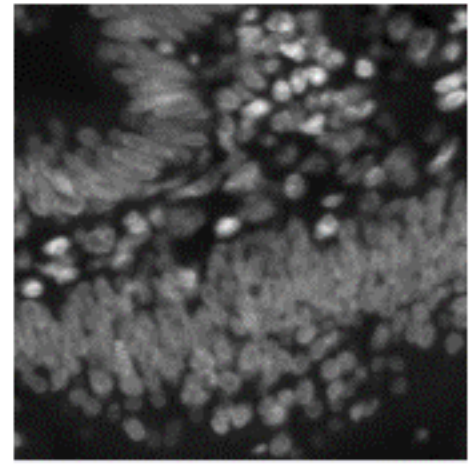
Data Augmentation

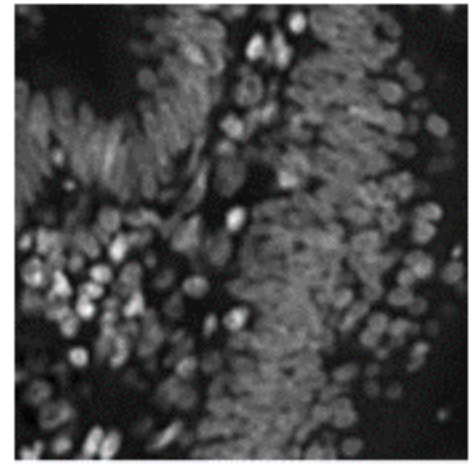
Original image

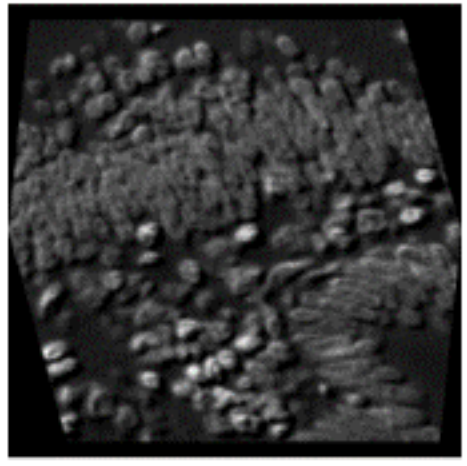


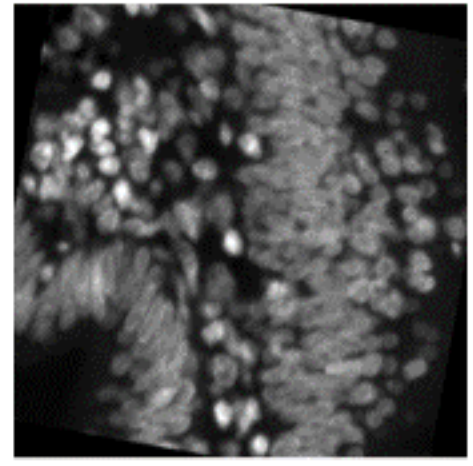
Augmented images

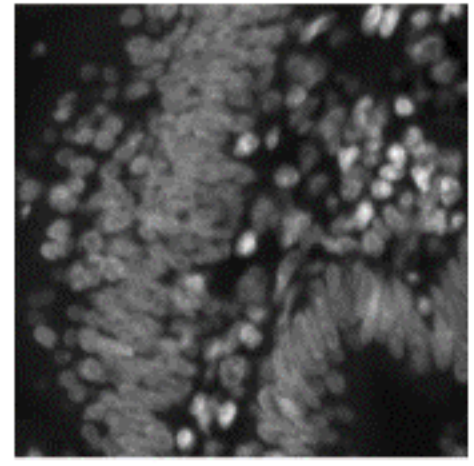




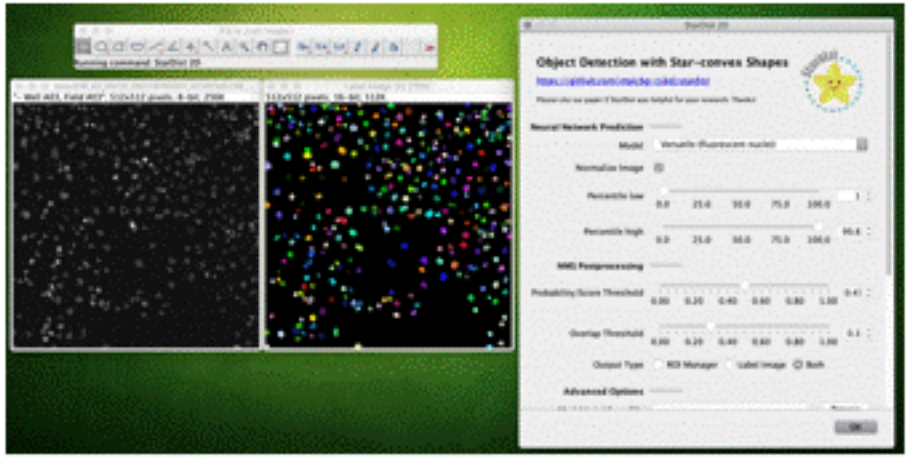


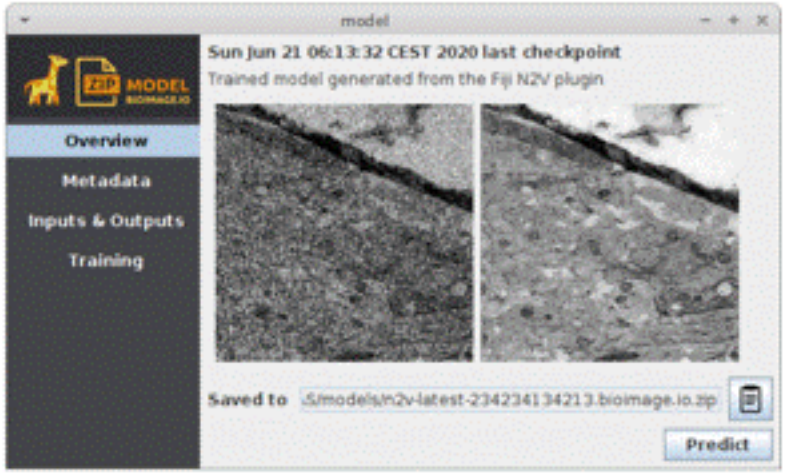






Pre-Trained Models





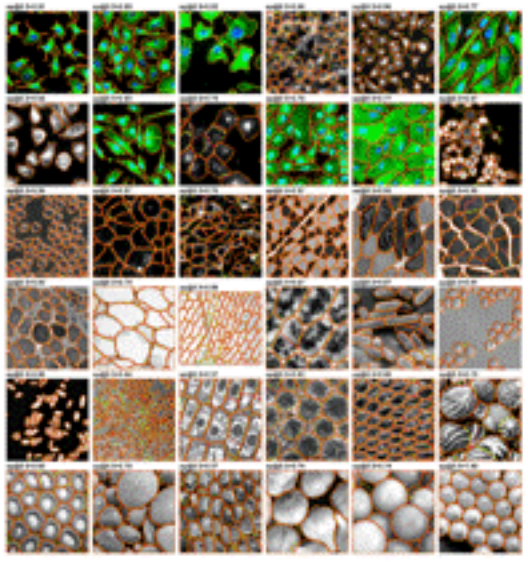
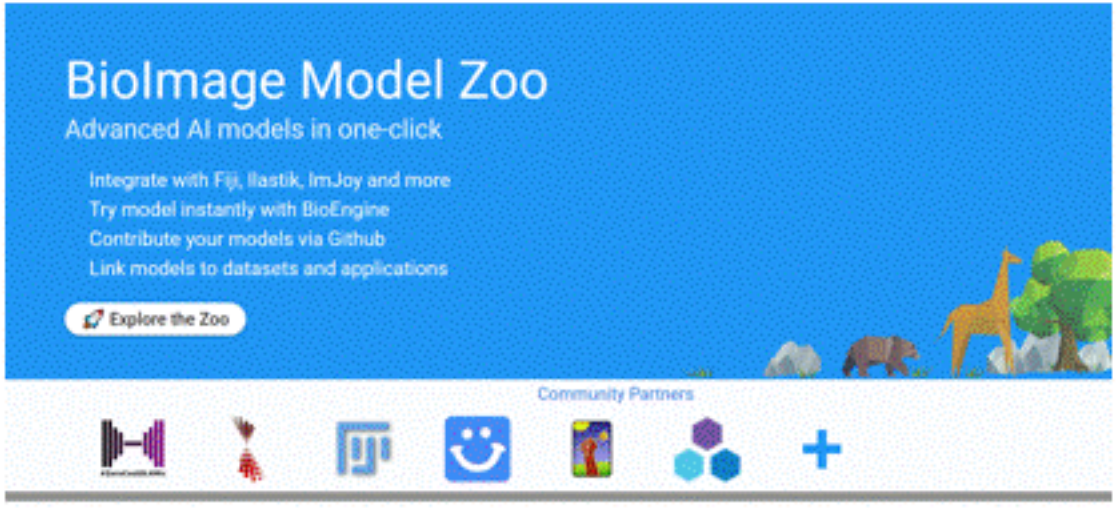
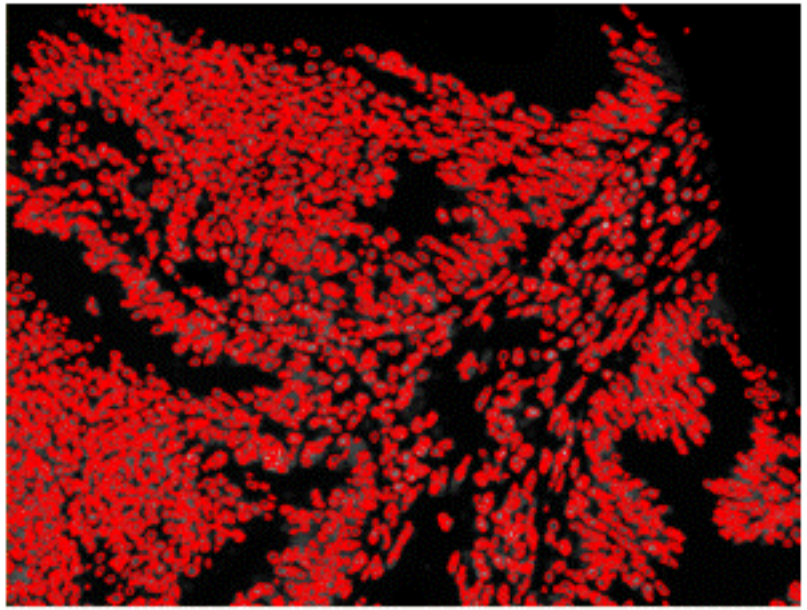
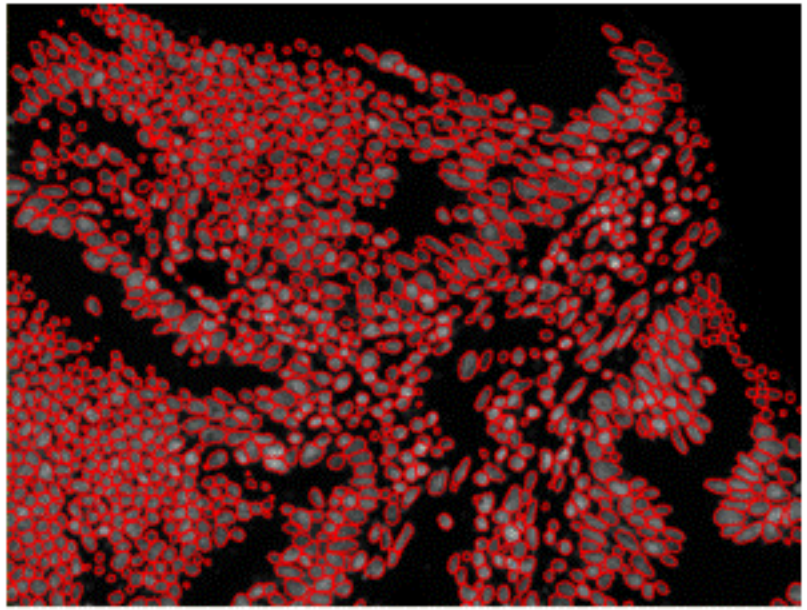


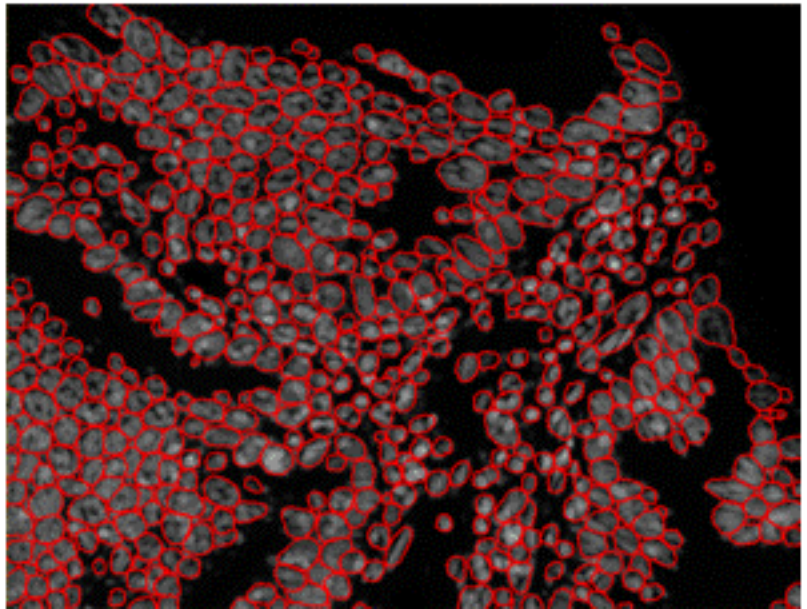
Figure 4: **Example cellpose segmentations for 36 test images.** The ground truth masks segmented by a human operator are shown in yellow, and the predicted masks are shown in dotted red line. Compare to Stardist and Mask-RCNN in Figure S3 and Figure S4.

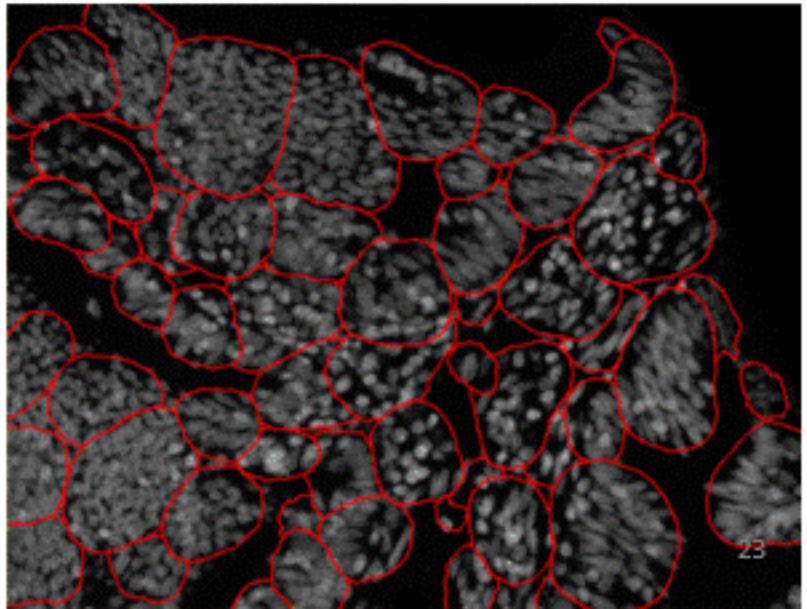


WARNINGS

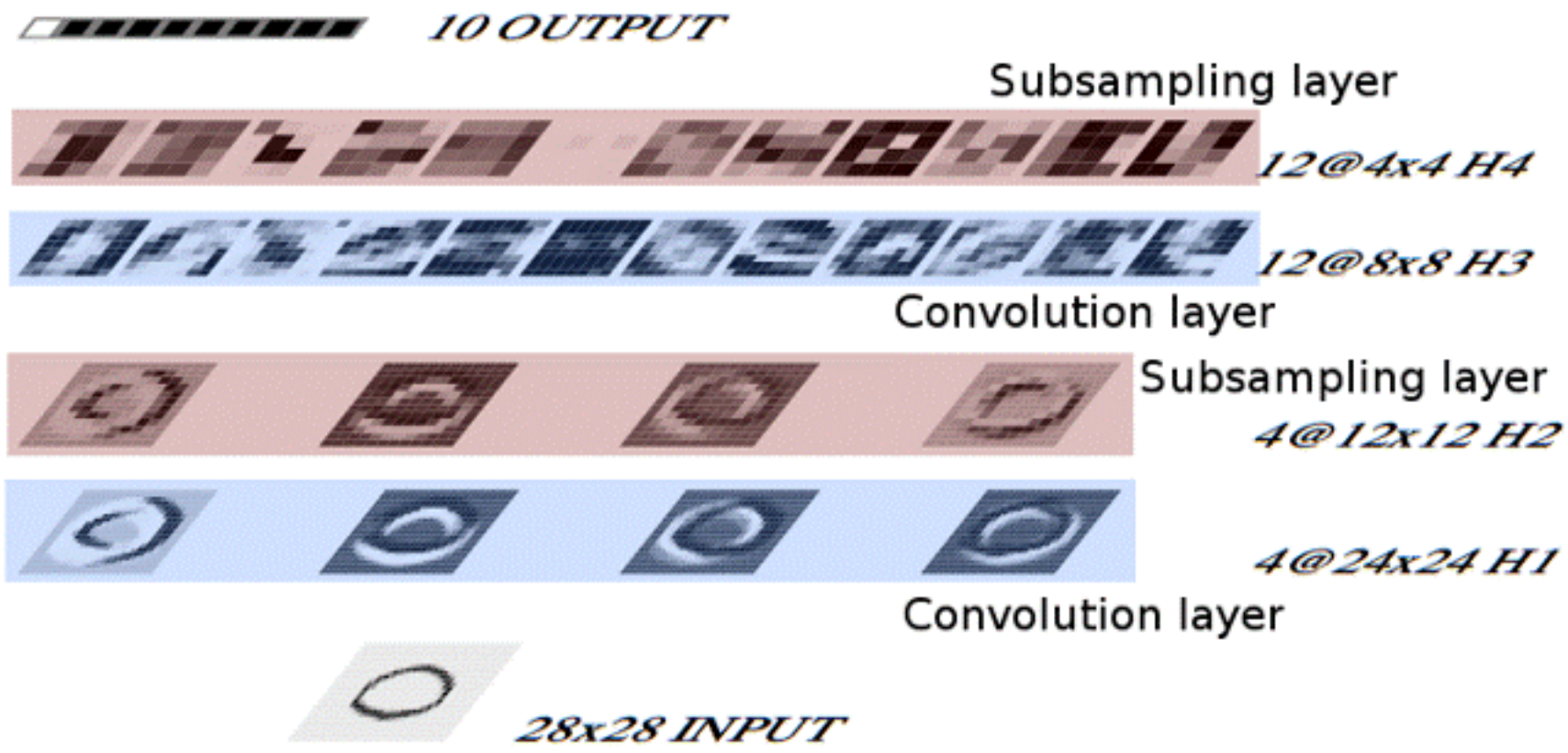








CONVOLUTIONAL NEURAL NETWORK



LeCun *et al.*, Handwritten digit recognition with a back-propagation network, *Neural Information Processing Systems, 1989*

4,635 pixels and 2,578 parameters (much lower than fully connected network)

MOBILENETV2

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

Table 2: MobileNetV2 : Each line describes a sequence of 1 or more identical (modulo stride) layers, repeated n times. All layers in the same sequence have the same number c of output channels. The first layer of each sequence has a stride s and all others use stride 1. All spatial convolutions use 3×3 kernels. The expansion factor t is always applied to the input size as described in Table 1.

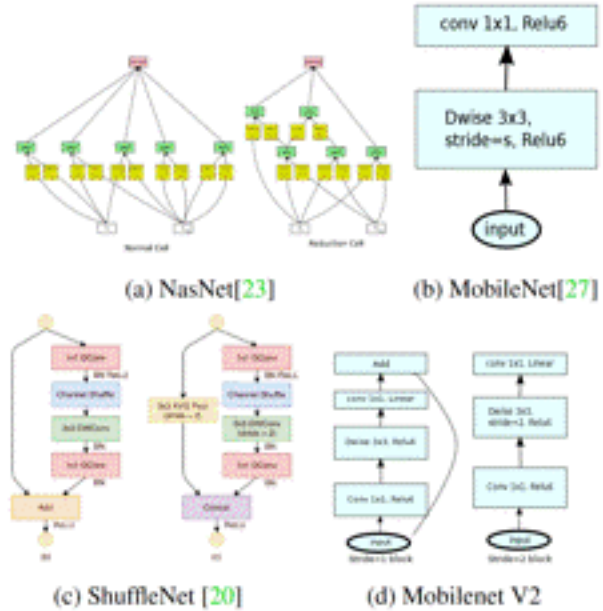


Figure 4: Comparison of convolutional blocks for different architectures. ShuffleNet uses Group Convolutions [20] and shuffling, it also uses conventional residual approach where inner blocks are narrower than output. ShuffleNet and NasNet illustrations are from respective papers.

Sandler *et al.*, MobileNetV2: Inverted Residuals and Linear Bottlenecks, *arXiv, 2019*

UNET

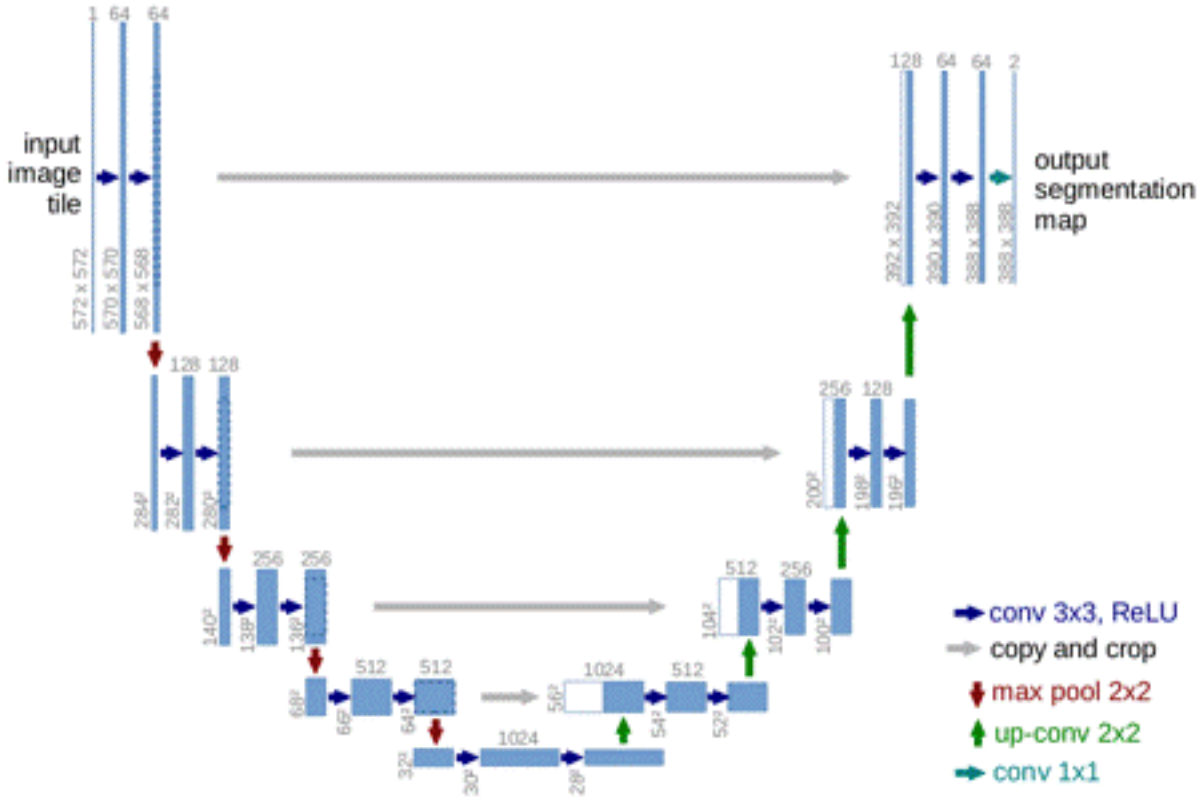
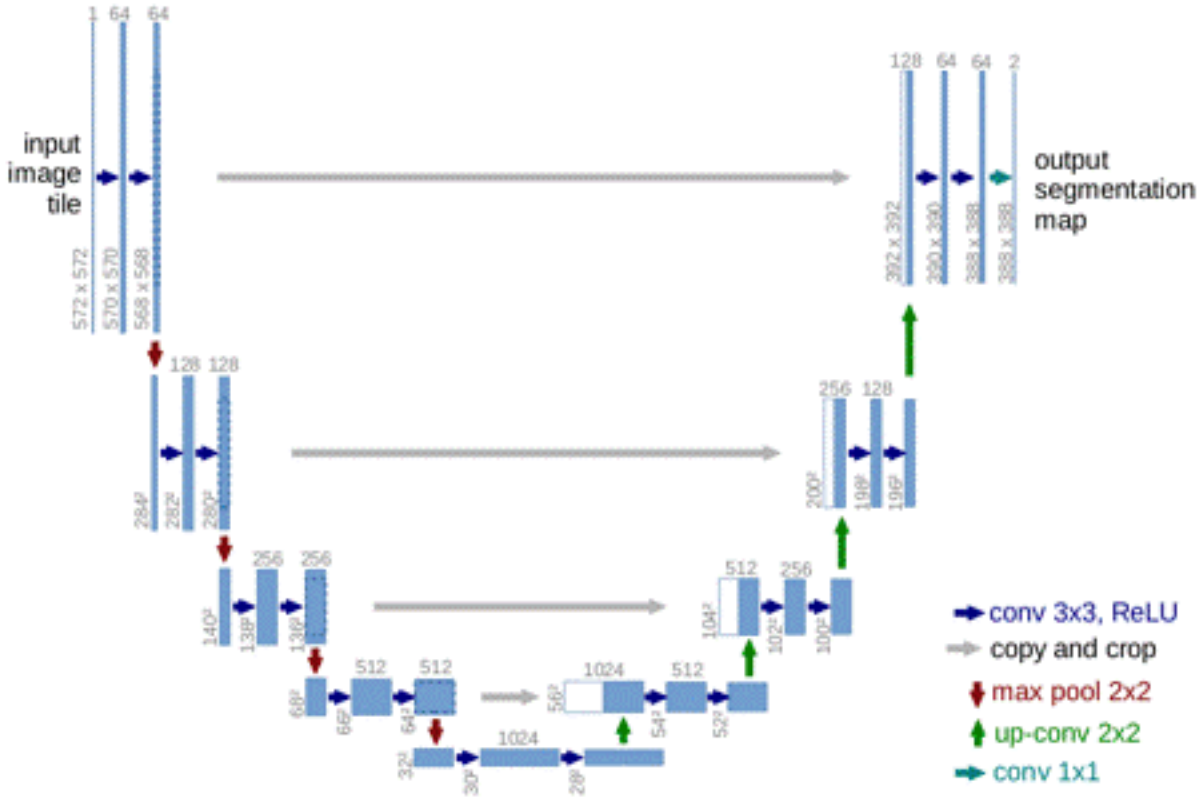


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

Ronneberger *et al.* U-Net: Convolutional Networks for Biomedical Image Segmentation. *Proceedings of MICCAI*, 2015.

UNET



7,700,000 parameters

Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

Ronneberger *et al.* U-Net: Convolutional Networks for Biomedical Image Segmentation. *Proceedings of MICCAI*, 2015.

StarDist – 2D

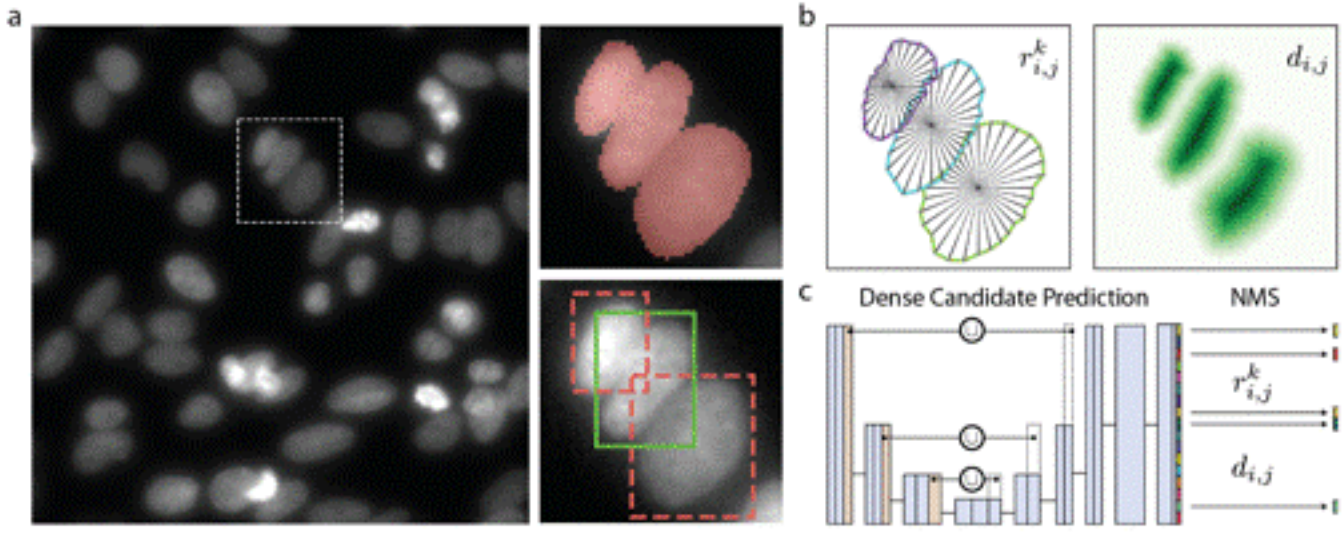
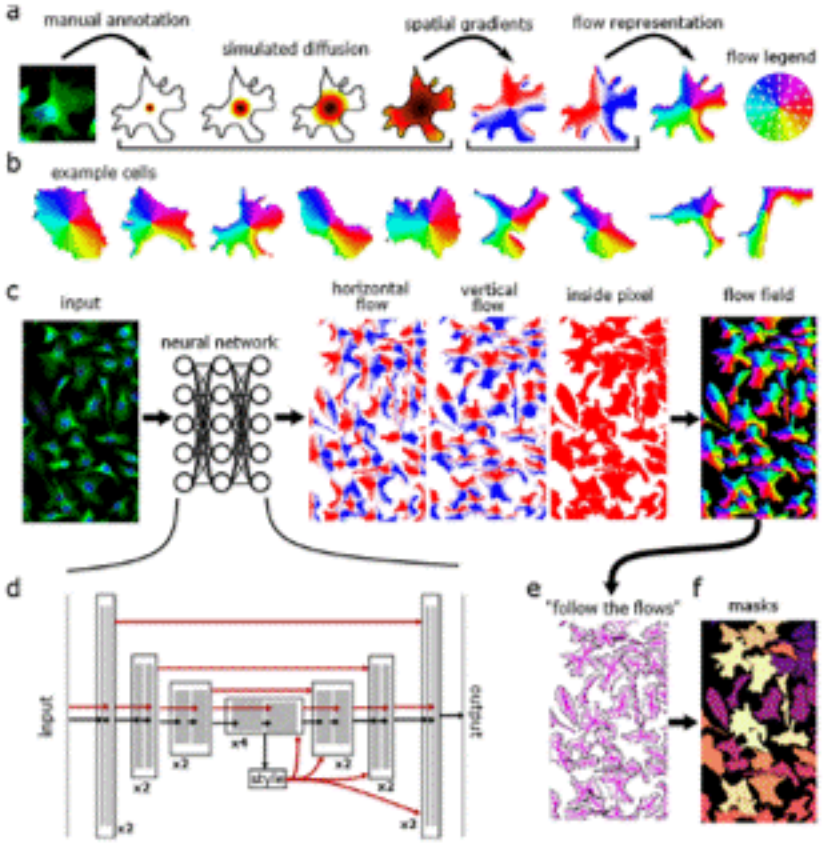
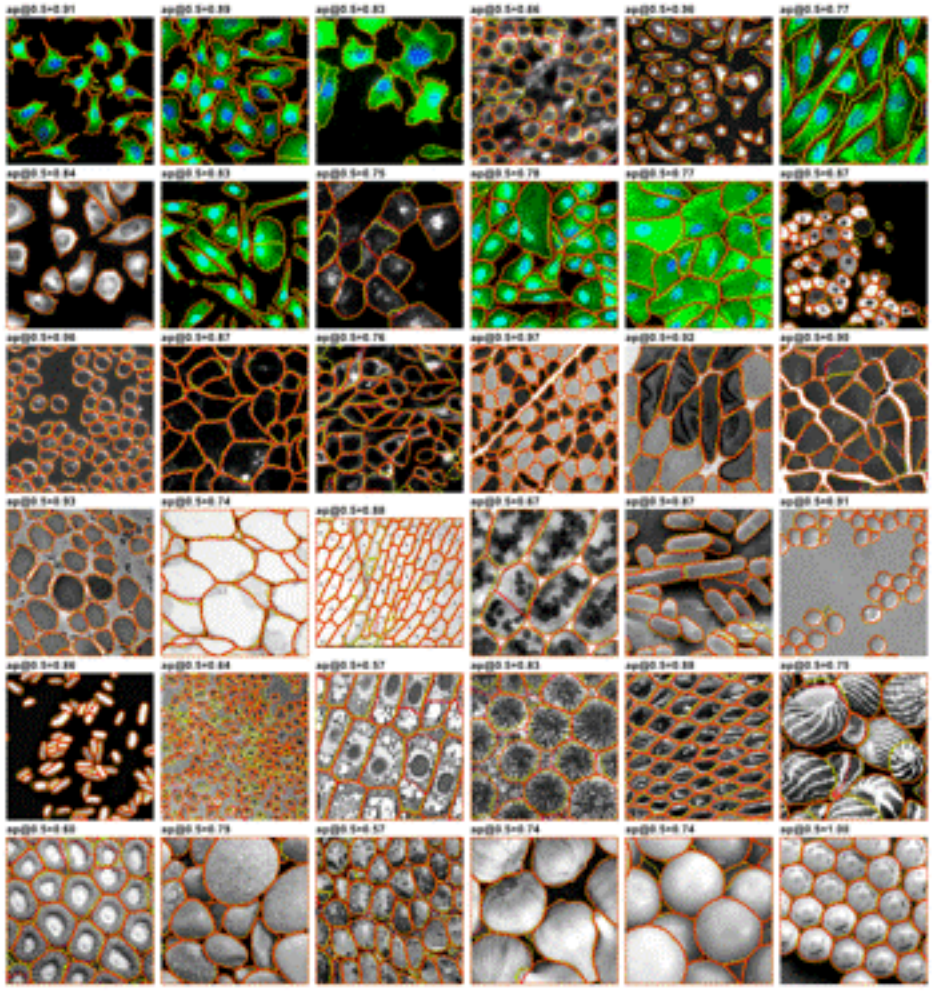


Fig. 1: *(a)* Potential segmentation errors for images with crowded nuclei: Merging of touching cells (upper right) or suppression of valid cell instances due to large overlap of bounding box localization (lower right). *(b)* The proposed StarDist method predicts object probabilities $d_{i,j}$ and star-convex polygons parameterized by the radial distances $r^k_{i,j}$. *(c)* We densely predict $r^k_{i,j}$ and $d_{i,j}$ using a simple U-Net architecture [15] and then select the final instances via non-maximum suppression (NMS).

CELLPOSE



Stringer *et al.,* Cellpose: a generalist algorithm for cellular segmentation. *Nature Methods*, 2021



OMNIPOSE

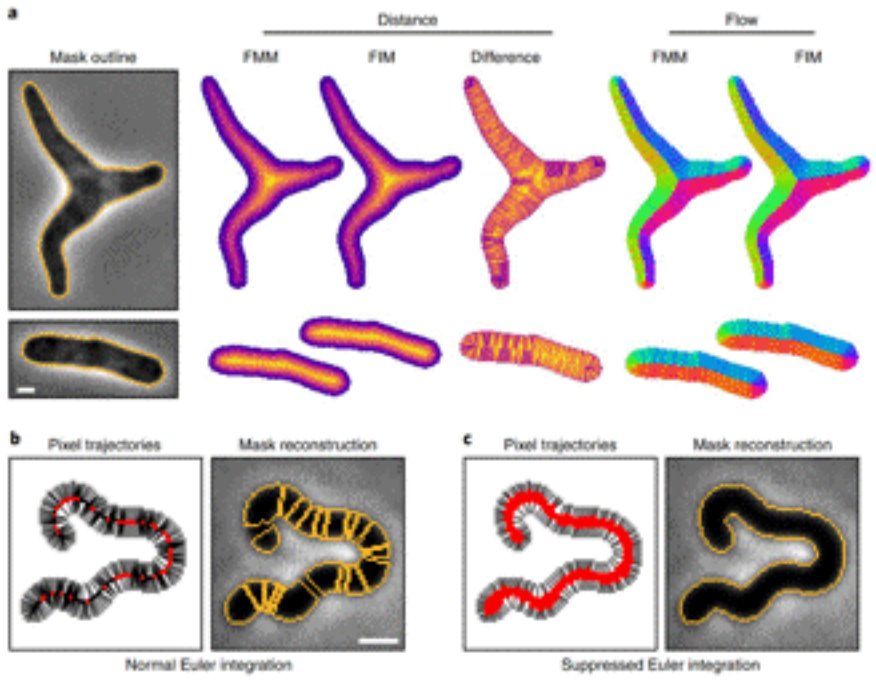


Fig. 2 | Core innovations of Omnipose. a, Comparison of distance field algorithms and corresponding flow fields on ground-truth masks. FMM produces ridges in the distance field resulting from pixelation on the cell mask boundary. Our smooth FIM algorithm minimizes these features. The difference image (FIM − FMM) highlights artifacts in the FMM method. Flow fields are calculated as the normalized gradient of the distance field. Boundary pixelation affects the FMM flow field deep into the cell, regardless of cell size. **b**,**c**, Comparison of mask reconstruction algorithms on a smooth flow field. Boundary pixel trajectories and resulting mask outlines from standard Euler integration (**b**). Trajectories and mask outlines under suppressed Euler integration (**c**). Red dots indicate the final positions of all cell pixels, not only the boundary pixels for which trajectories are displayed. Bacteria displayed are *E. coli* CS703-1 (**a**) and *H. pylori* (**b**,**c**) both treated with aztreonam. Scale bars, 1 μm. Images are representative of 1,299 *E. coli* and 701 *H. pylori* cells in the total ground-truth dataset, respectively.

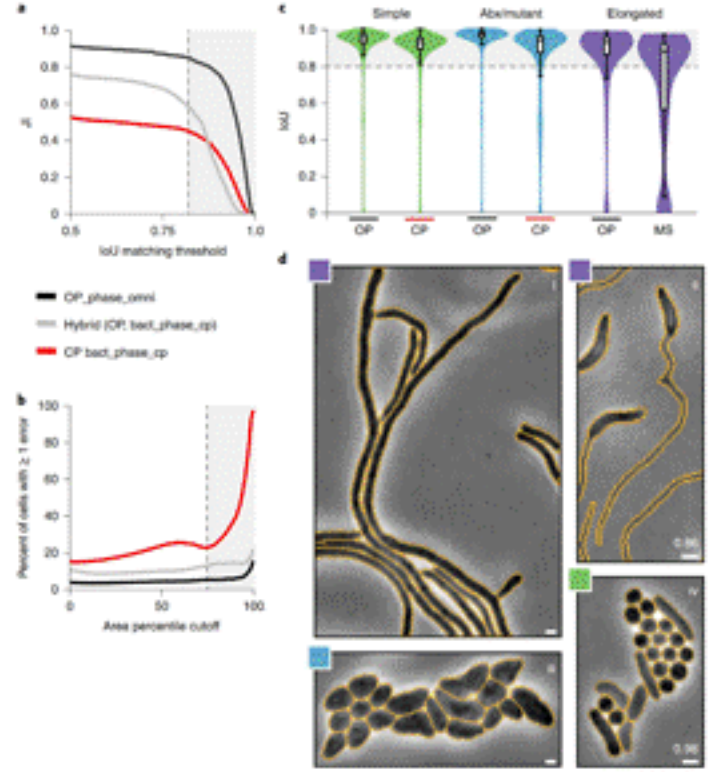


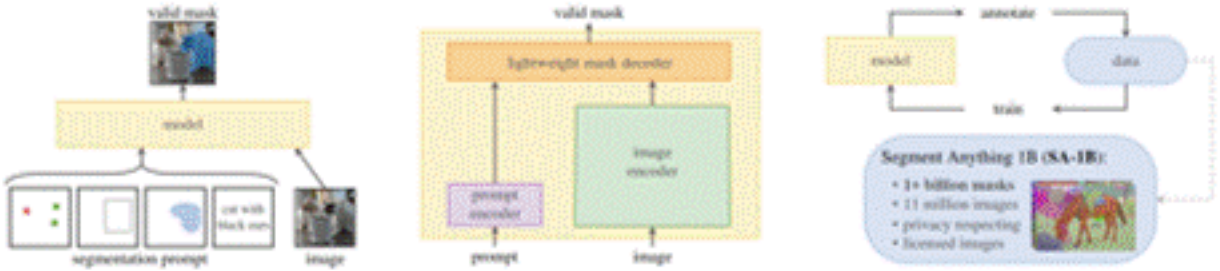
Fig. 3 | Omnipose substantially outperforms Cellpose on elongated cells. a, Overall performance of Omnipose (OP) (bact_phase_omni) and Cellpose (CP) (bact_phase_cp) measured by IJI. The hybrid method (gray) uses the original center-seeking flow output of bact_phase_cp and the mask reconstruction of Omnipose. Gray box represents IoU ≥ 0.8. $n = 19{,}570$ cells in the test set. **b**, Quantification of segmentation performance by cell size. The percent of cells with at least one segmentation error is computed for cells in each area percentile group from 1 to 100. Gray box represents the top quartile. **c**, Omnipose IoU distribution on the bact_phase dataset compared to the next highest performing algorithm in each of three cell categories (simple, $n = 12{,}869$; Abx/mutant, $n = 6{,}138$; and elongated, $n = 531$). Boxes centered on medians from Q1 to Q3, whiskers from Q1 − 1.5 IQR to Q3 + 1.5 IQR. **d**, Example micrographs and Omnipose segmentation. Mean matched IoU values shown. Bacteria displayed are *Streptomyces pristinaespiralis* (i), *Caulobacter crescentus* grown in HBGG medium (ii), *Shigella flexneri* treated with A22 (iii) and a mix of *Pseudomonas aeruginosa*, *Staphylococcus aureus*, *V. cholerae*, and *Bacillus subtilis* (iv). HBGG, Hutner base–imidazole-buffered glucose–glutamate. Scale bars, 1 μm.

Cutler *et al.,* Omnipose: a high-precision morphology-independent solution for bacterial cell segmentation. *Nature Methods*, 2022

FOUNDATION MODEL FOR INSTANCE SEGMENTATION

Segment Anything

Alexander Kirillov[1,2,4] Eric Mintun[2] Nikhila Ravi[1,2] Hanzi Mao[2] Chloe Rolland[3] Laura Gustafson[3]
Tete Xiao[3] Spencer Whitehead Alexander C. Berg Wan-Yen Lo Piotr Dollár[4] Ross Girshick[4]
[1]project lead [2]joint first author [3]equal contribution [4]directional lead
Meta AI Research, FAIR



(a) **Task**: promptable segmentation (b) **Model**: Segment Anything Model (**SAM**) (c) **Data**: data engine (top) & dataset (bottom)

Figure 1: We aim to build a foundation model for segmentation by introducing three interconnected components: a prompt-able segmentation *task*, a segmentation *model* (SAM) that powers data annotation and enables zero-shot transfer to a range of tasks via prompt engineering, and a *data* engine for collecting SA-1B, our dataset of over 1 billion masks.

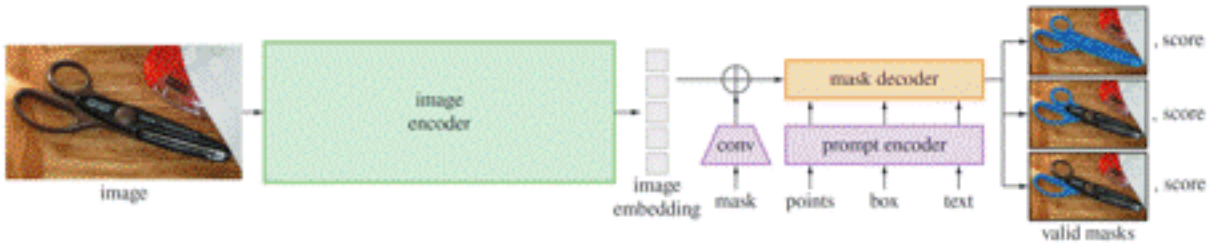


Figure 4: Segment Anything Model (SAM) overview. A heavyweight image encoder outputs an image embedding that can then be efficiently queried by a variety of input prompts to produce object masks at amortized real-time speed. For ambiguous prompts corresponding to more than one object, SAM can output multiple valid masks and associated confidence scores.

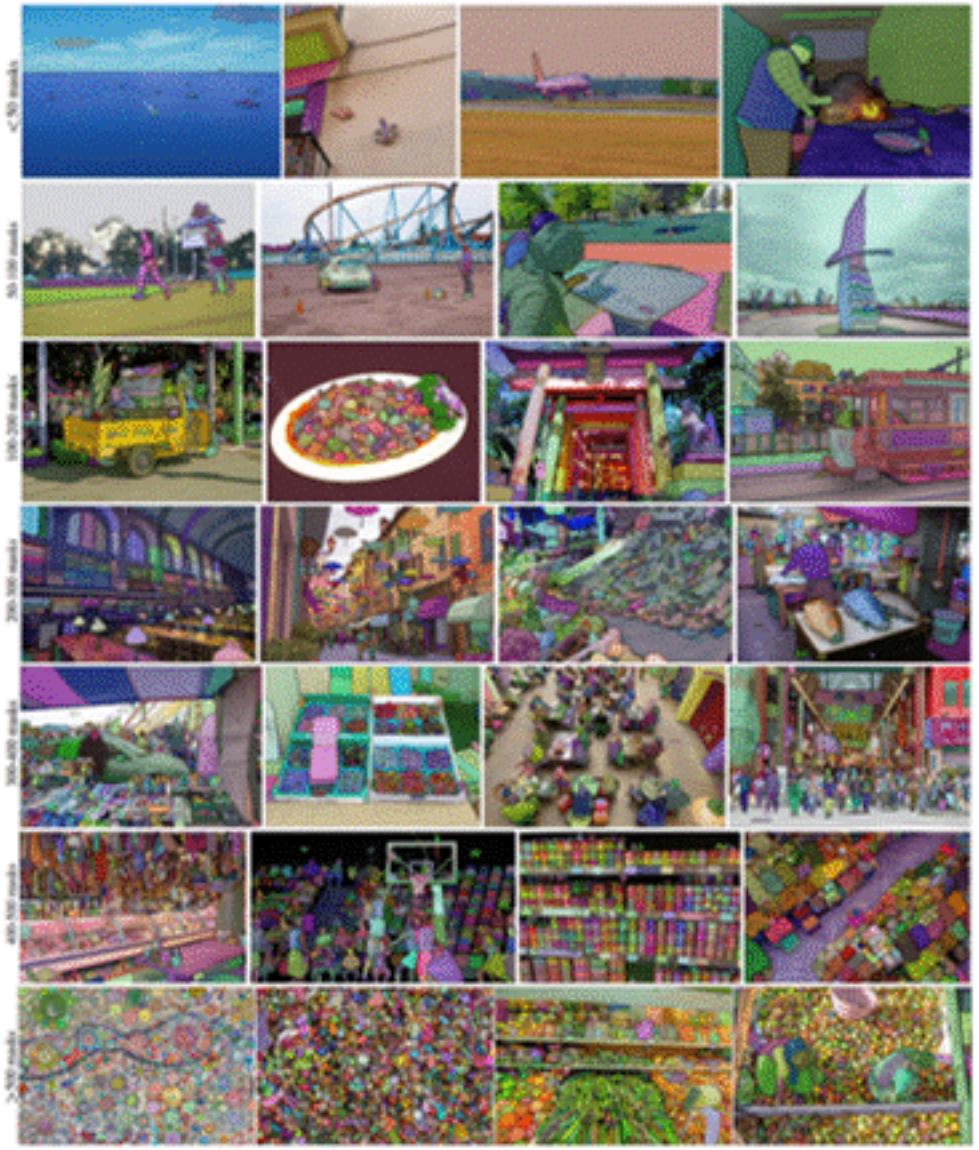


Figure 2: Example images with overlaid masks from our newly introduced dataset, **SA-1B**. SA-1B contains 11M diverse, high-resolution, licensed, and privacy protecting images and 1.1B high-quality segmentation masks. These masks were annotated *fully automatically* by SAM, and as we verify by human ratings and numerous experiments, are of high quality and diversity. We group images by number of masks per image for visualization (there are ~100 masks per image on average).

FOUNDATION MODEL FOR INSTANCE SEGMENTATION

Segment Anything

Alexander Kirillov[1,2,4] Eric Mintun[2] Nikhila Ravi[1,2] Hanzi Mao[2] Chloe Rolland[3] Laura Gustafson[3]
Tete Xiao[3] Spencer Whitehead Alexander C. Berg Wan-Yen Lo Piotr Dollár[4] Ross Girshick[4]
[1]project lead [2]joint first author [3]equal contribution [4]directional lead
Meta AI Research, FAIR

(a) **Task**: promptable segmentation (b) **Model**: Segment Anything Model (**SAM**) (c) **Data**: data engine (top) & dataset (bottom)

Figure 1: We aim to build a foundation model for segmentation by introducing three interconnected components: a promptable segmentation *task*, a segmentation *model* (SAM) that powers data annotation and enables zero-shot transfer to a range of tasks via prompt engineering, and a *data* engine for collecting SA-1B, our dataset of over 1 billion masks.

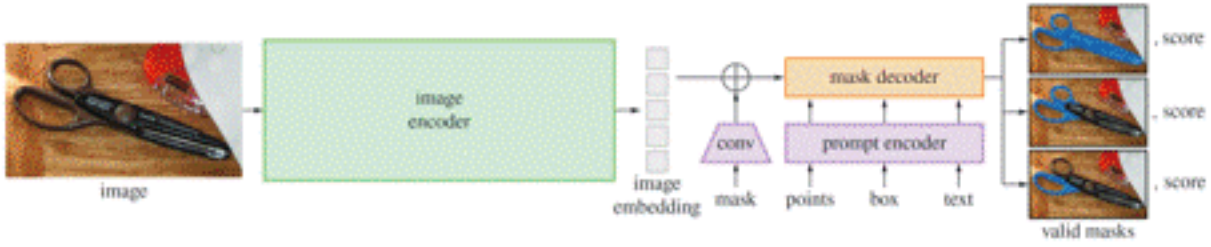


Figure 4: Segment Anything Model (SAM) overview. A heavyweight image encoder outputs an image embedding that can then be efficiently queried by a variety of input prompts to produce object masks at amortized real-time speed. For ambiguous prompts corresponding to more than one object, SAM can output multiple valid masks and associated confidence scores.

Figure 3: Each column shows 3 valid masks generated by SAM from a single ambiguous point prompt (green circle).