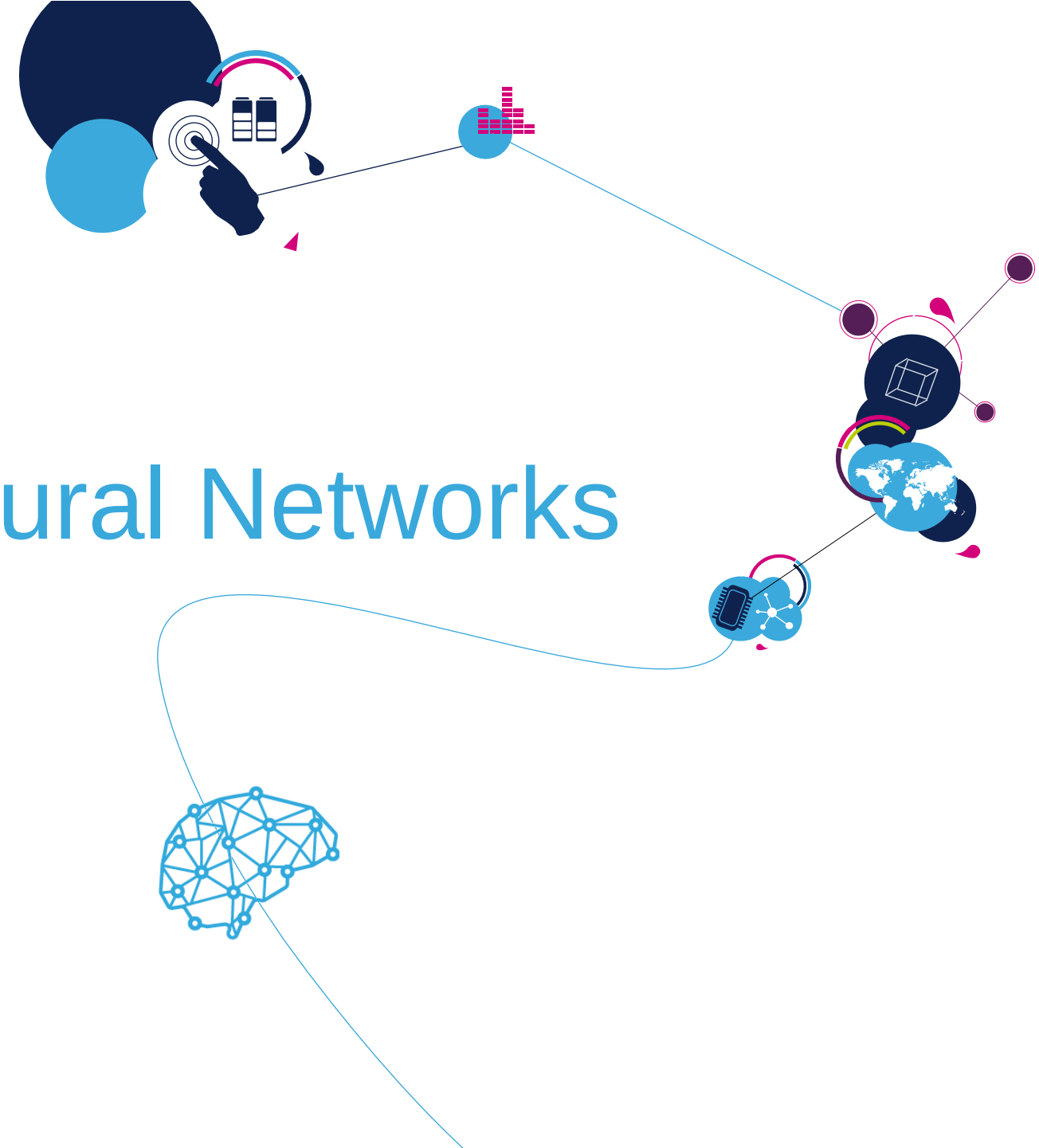


Introduction to Neural Networks

Danilo Pau

Advanced System Technology

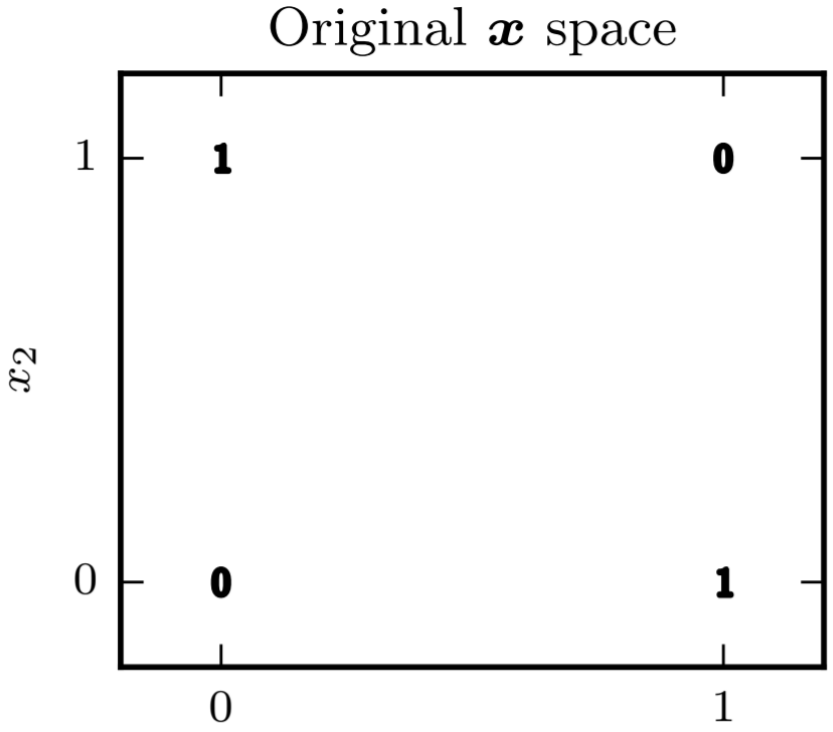
Agrate Brianza

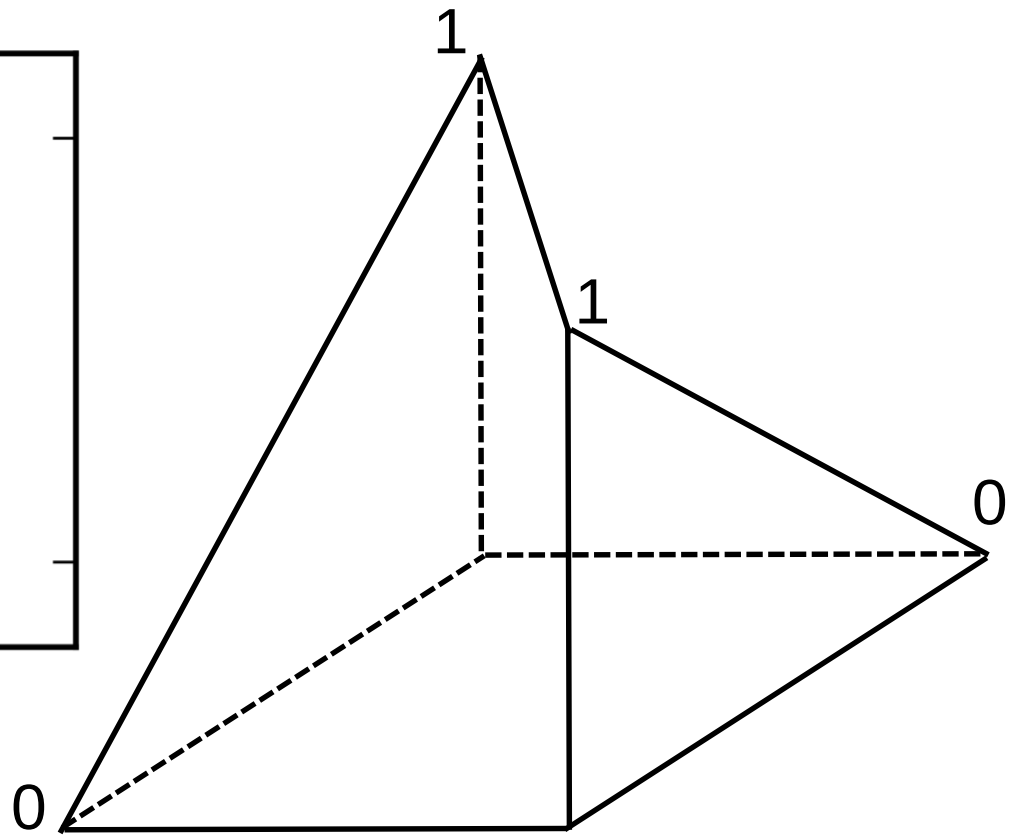
life.augmented

Learning XOR

- Not linearly separable

x_1	x_2	$x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0





Source https://vision.unipv.it/AI/AIRG.html

Learning XOR

- $y = X * \vartheta$

$$\boldsymbol{y} := \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad \boldsymbol{X} := \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \boldsymbol{\vartheta} := \begin{bmatrix} w_1 \\ w_2 \\ c \end{bmatrix}$$

XOR

x_1	x_2	$x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Source https://vision.unipv.it/AI/AIRG.html

Function Approximation: linear combination

- Linear Approximator as 1st attempt

$$\tilde{y} = \boldsymbol{w} \cdot \boldsymbol{x} + c, \quad \boldsymbol{w} \in \mathbb{R}^d, c \in \mathbb{R}$$

For XOR: $\boldsymbol{\vartheta} = (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{y}$

$$\boldsymbol{X}^T \boldsymbol{X} = \begin{bmatrix} 2 & 1 & 2 \\ 1 & 2 & 2 \\ 2 & 2 & 4 \end{bmatrix}$$

squared

$$(\boldsymbol{X}^T \boldsymbol{X})^{-1} = \begin{bmatrix} 1 & 0 & 0.5 \\ 0 & 1 & 0.5 \\ 0.5 & 0.5 & 0.75 \end{bmatrix}$$

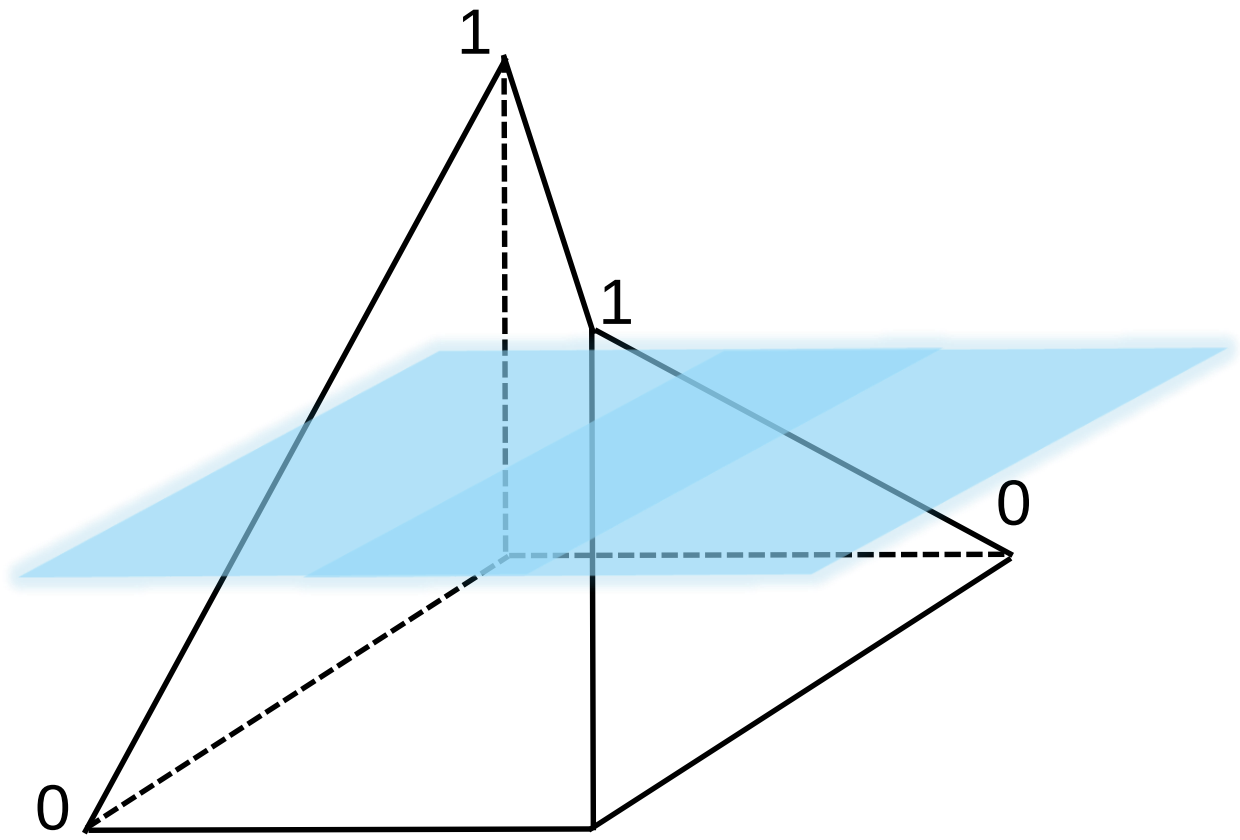
squared

$$(\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{y} = \begin{bmatrix} 0 \\ 0 \\ 0.5 \end{bmatrix}$$

Hence approximator becomes $\tilde{y} = 0.5$

Source https://vision.unipv.it/AI/AIRG.html

Function Approximation: linear combination



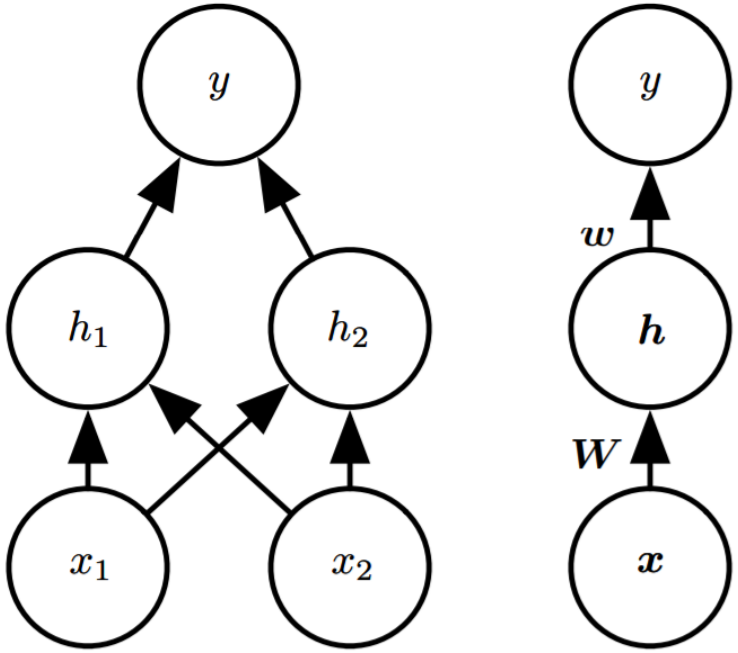
$$\tilde{y} = 0.5$$

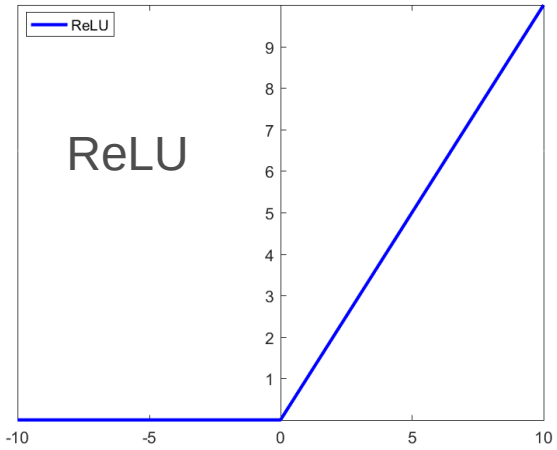
Source https://vision.unipv.it/AI/AIRG.html

Solving XOR

- *(shallow) feed-forward neural network*

$$\tilde{y} = \boldsymbol{w} \cdot \mathrm{ReLU}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{c}) + c$$

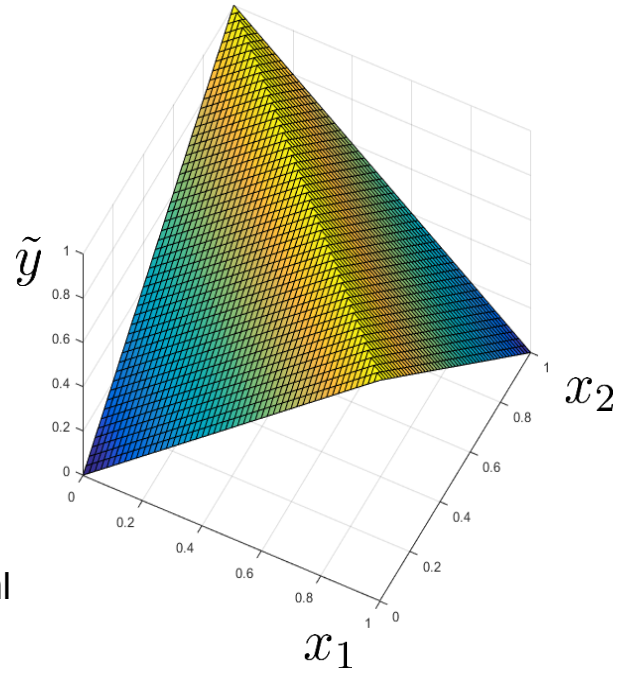




$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix},$$

$$\boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix},$$

$$\boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix},$$



Source https://vision.unipv.it/AI/AIRG.html

Universal approximation theorem (Cybenko, 1989, Hornik, 1991)

For any target function

$$y = f^*(\boldsymbol{x}), \quad \boldsymbol{x} \in \mathbb{R}^d$$

(which is continuous and Borel measurable)

and any $\varepsilon > 0$ there exists parameters

$$h \in \mathbb{Z}^+, \boldsymbol{W} \in \mathbb{R}^{h \times d}, \ \boldsymbol{w}, \boldsymbol{c} \in \mathbb{R}^h, c \in \mathbb{R}$$

this is the dimension of the hidden layer: it is a parameter in the theorem

such that the ***(shallow) feed-forward neural network***

$$\tilde{y} = \boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{c}) + c$$

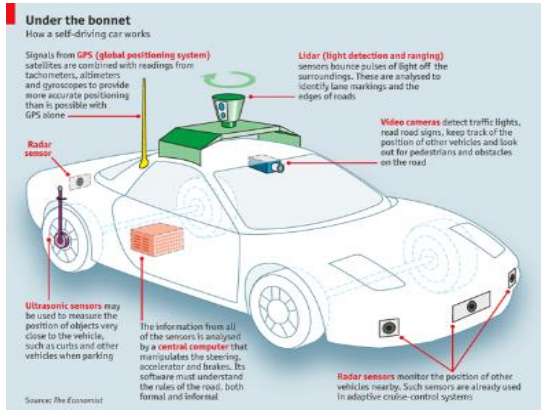
approximates the target function by less than ε

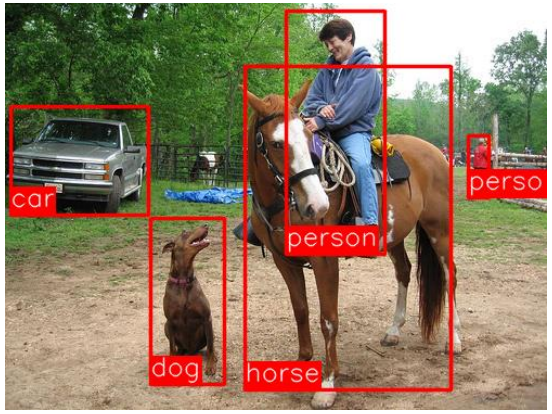
$$\left| f^*(\boldsymbol{x}) - \boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{c}) + c \right| < \varepsilon$$

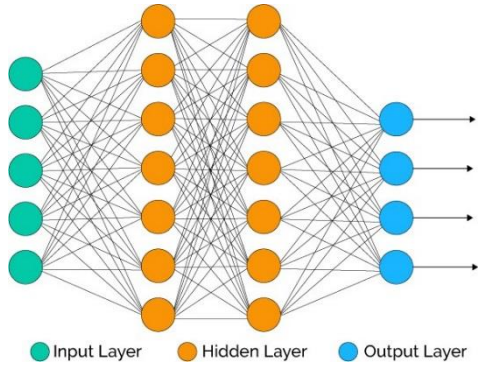
(on a compact subset of $\mathbb{R}^d$)

Source https://vision.unipv.it/AI/AIRG.html

Artificial Neural Network







State of the art

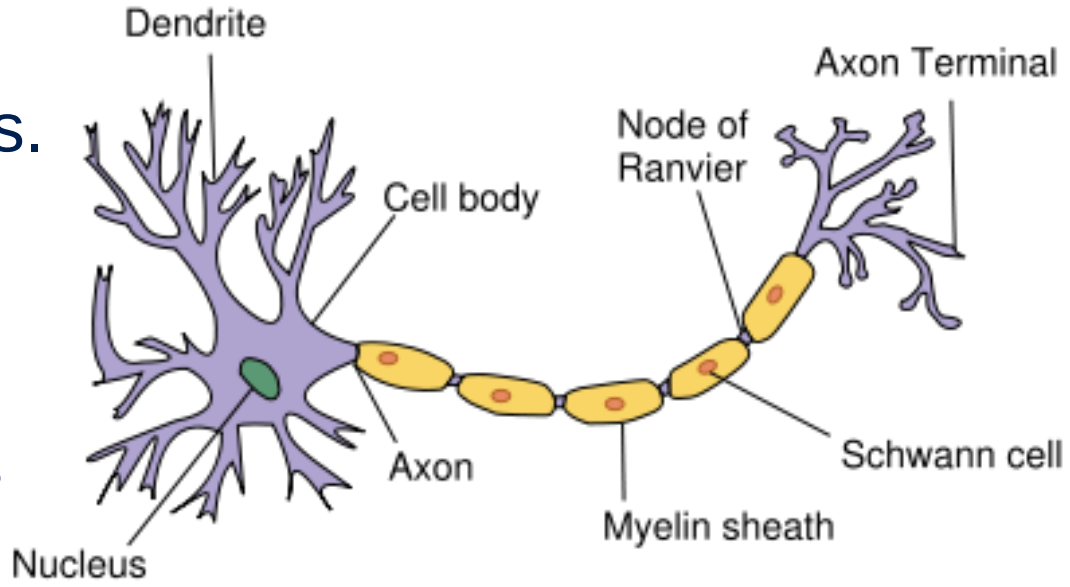
Core of all deep learning applications

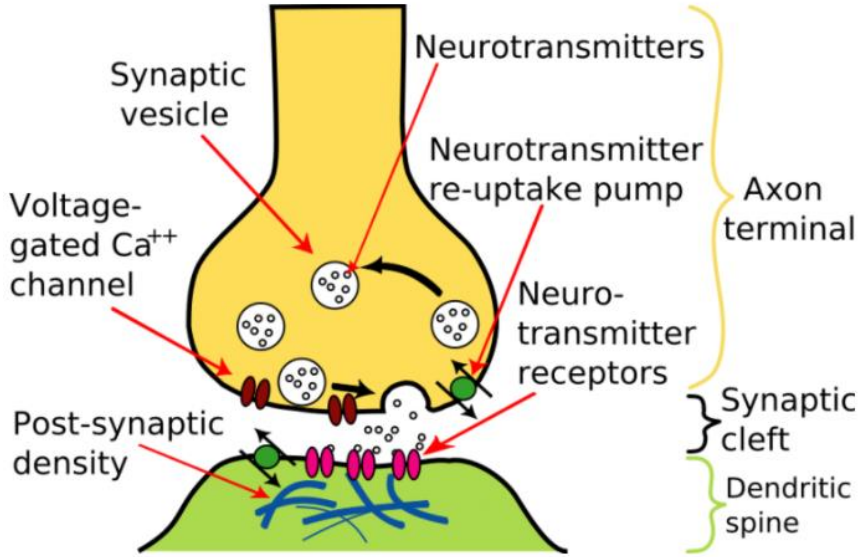
Adaptable

life.augmented

What are Neural Networks?

- Also referred to as Artificial Neural Networks.
- Inspired by human neural system.
- Human neuron has three main components
  - Dendrites
    - Takes inputs from other neurons in terms of electrical pulses.
  - Cell body
    - Makes the inferences and decides the actions to take.
  - Axon terminals
    - Sends the outputs to other neurons in terms of electrical pulses.
  - Synapse
    - Interface between Axons and Dendrites





life.augmented

Michael Jordan: There are no spikes in deep-learning systems. There are no dendrites. And they have bidirectional signals that the brain doesn't have.

We don't know how neurons learn. Is it actually just a small change in the synaptic weight that's responsible for learning? That's what these artificial neural networks are doing. In the brain, we have precious little idea how learning is actually taking place.

Spectrum: I read all the time about engineers describing their new chip designs in what seems to me to be an incredible abuse of language. They talk about the "neurons" or the "synapses" on their chips. But that can't possibly be the case; a neuron is a living, breathing cell of unbelievable complexity. Aren't engineers

appropriating the language of biology to describe structures that have nothing remotely close to the complexity of biological systems?

Michael Jordan: Well, I want to be a little careful here. I think it's important to distinguish two areas where the word *neural* is currently being used.

One of them is in deep learning. And there, each "neuron" is really a cartoon. It's a linear-weighted sum that's passed through a nonlinearity. Anyone in electrical engineering would recognize those kinds of nonlinear systems. Calling that a neuron is clearly, at best, a shorthand. It's really a cartoon. There is a procedure called logistic regression in statistics that dates from the 1950s, which had nothing to do with neurons but which is exactly the same little piece of architecture.

Machine-Learning Maestro Michael Jordan on the Delusions of Big Data and Other Huge Engineering Efforts

Big-data boondoggles and brain-inspired chips are just two of the things we're really getting wrong

By Lee Gomes
Posted 20 Oct 2014 | 19:37 GMT

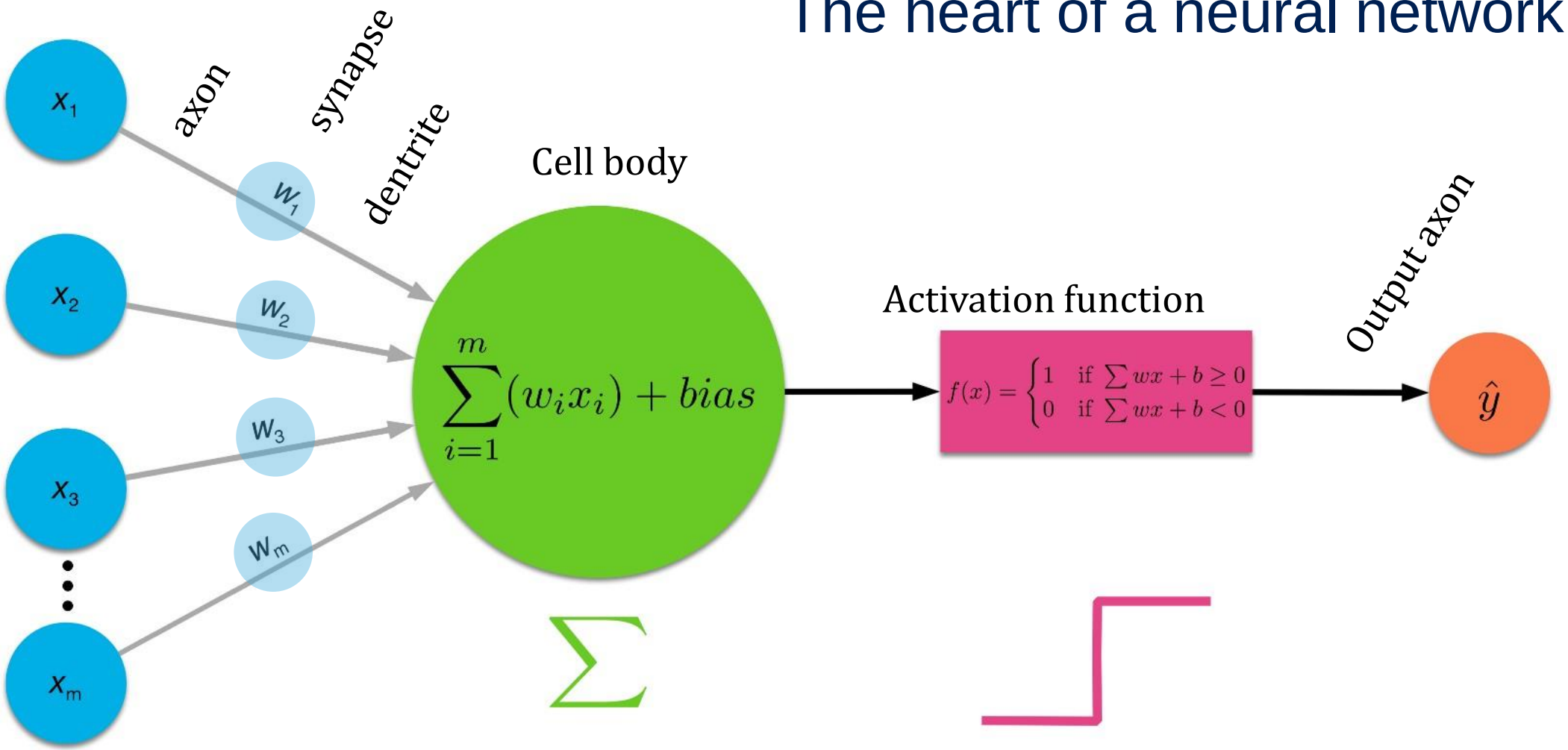
Photo-Illustration: Randi Klett

life.augmented

http://spectrum.ieee.org/robotics/artificial-intelligence/machinelearning-maestro-michael-jordan-on-the-delusions-of-big-data-and-other-huge-engineering-efforts

Artificial Neuron

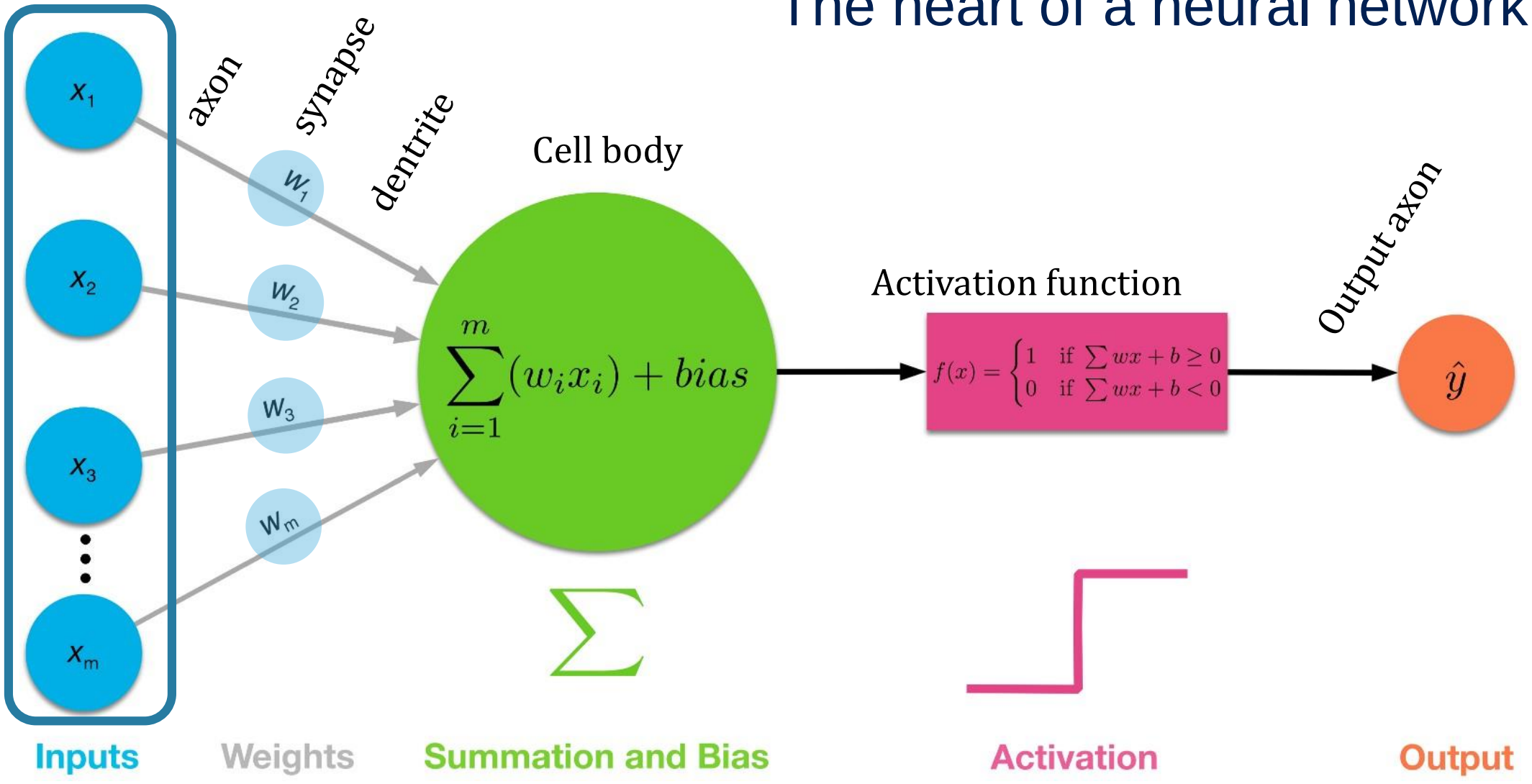
The heart of a neural network



life.augmented

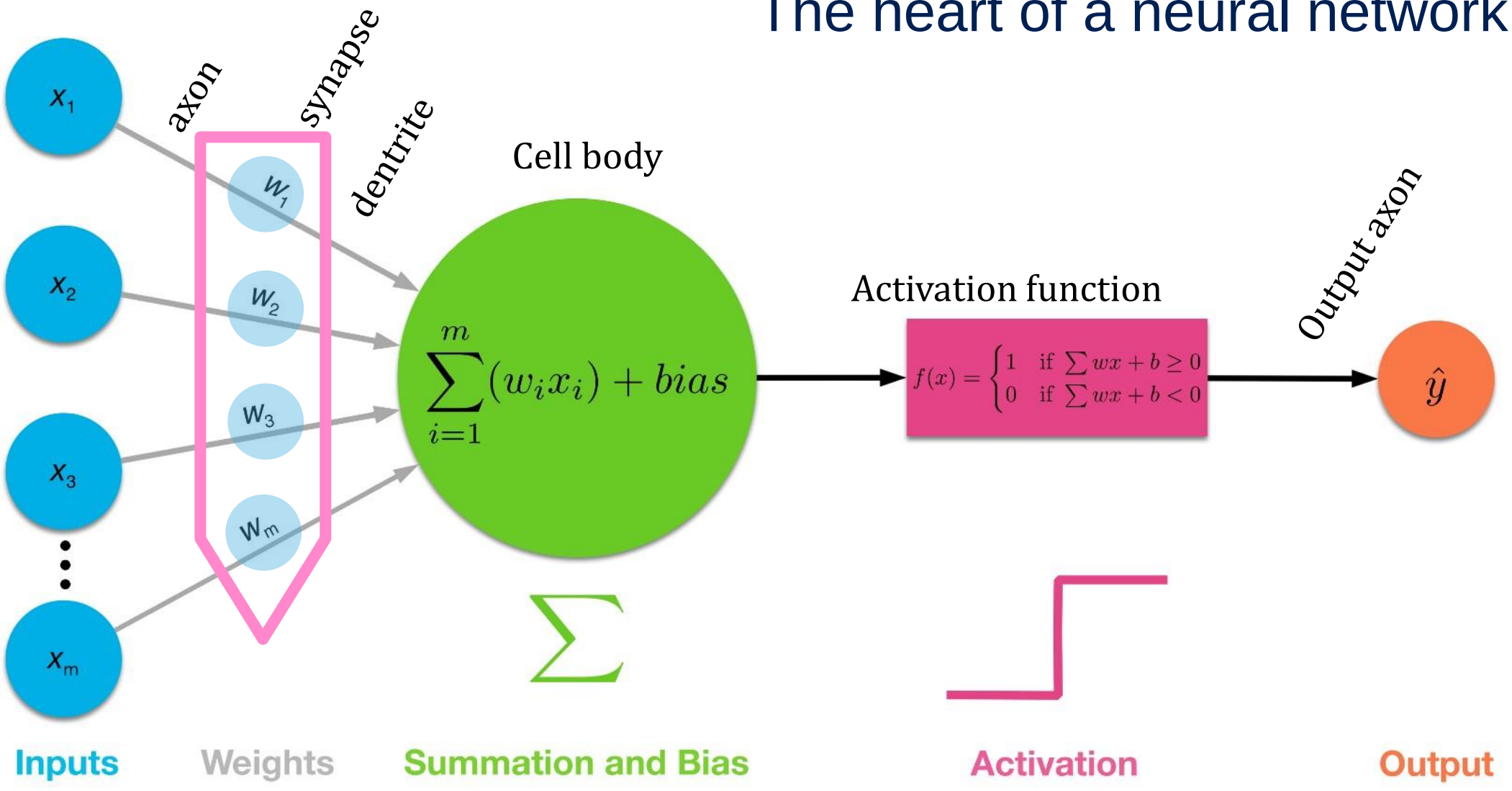
Artificial Neuron

The heart of a neural network



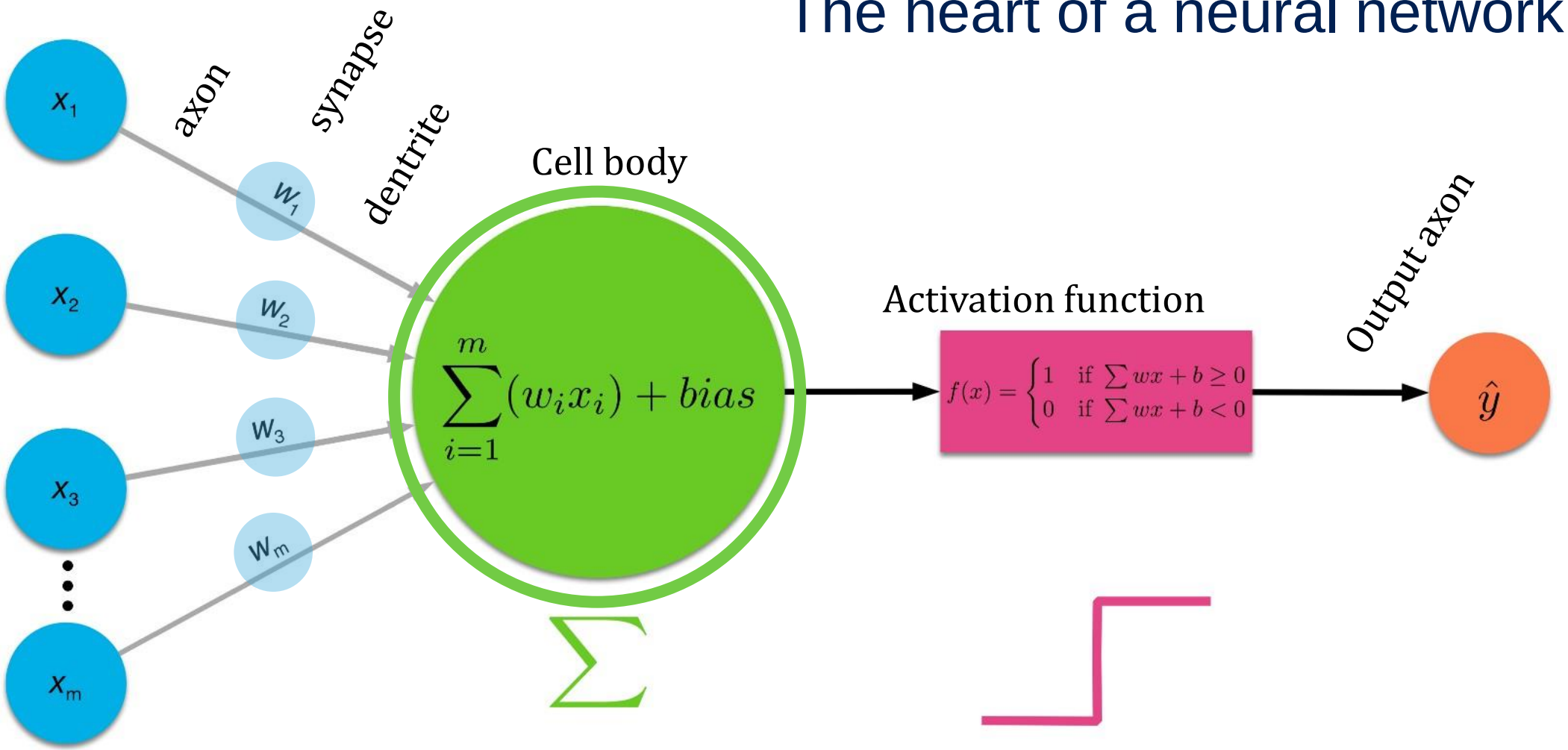
Artificial Neuron

The heart of a neural network



Artificial Neuron

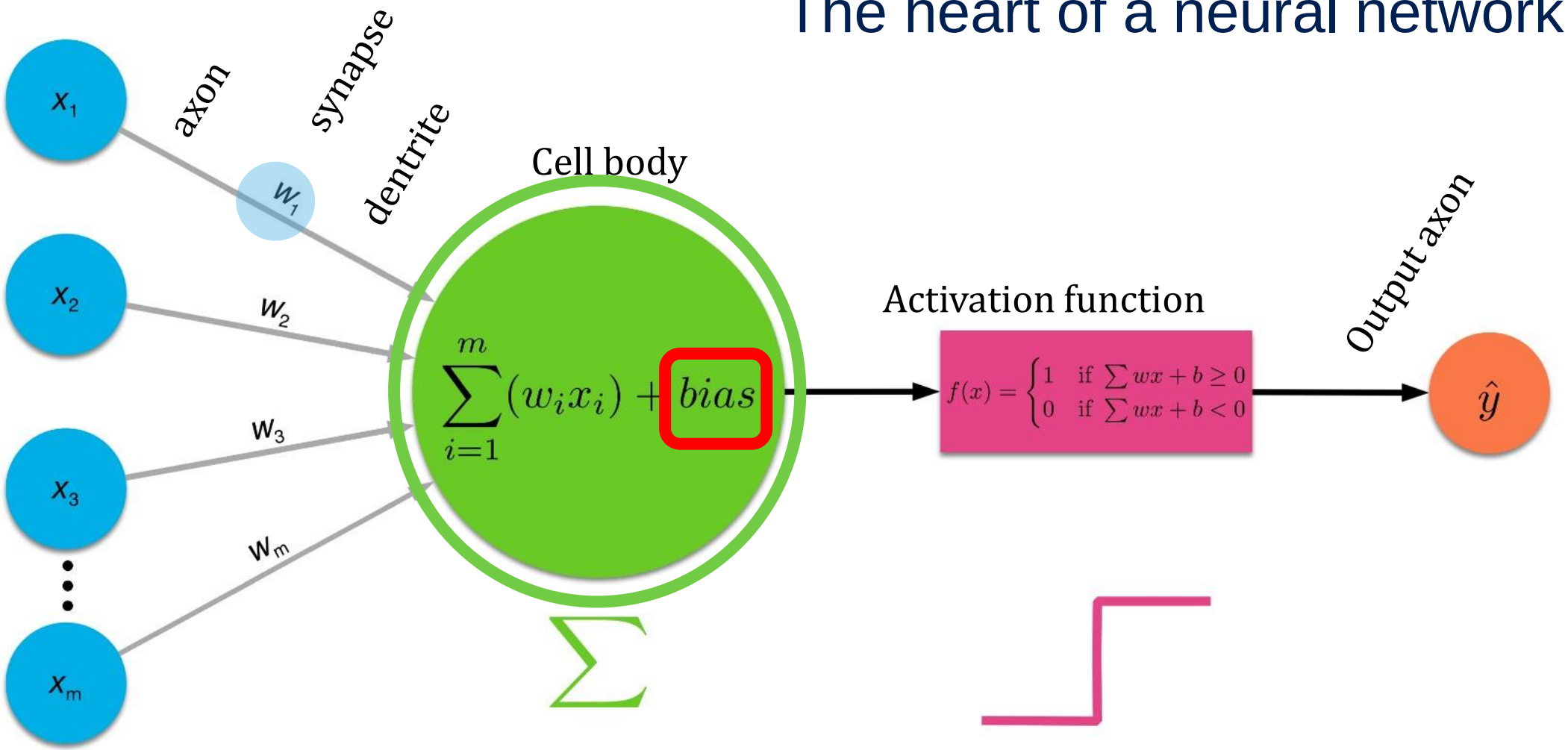
The heart of a neural network



life.augmented

Artificial Neuron

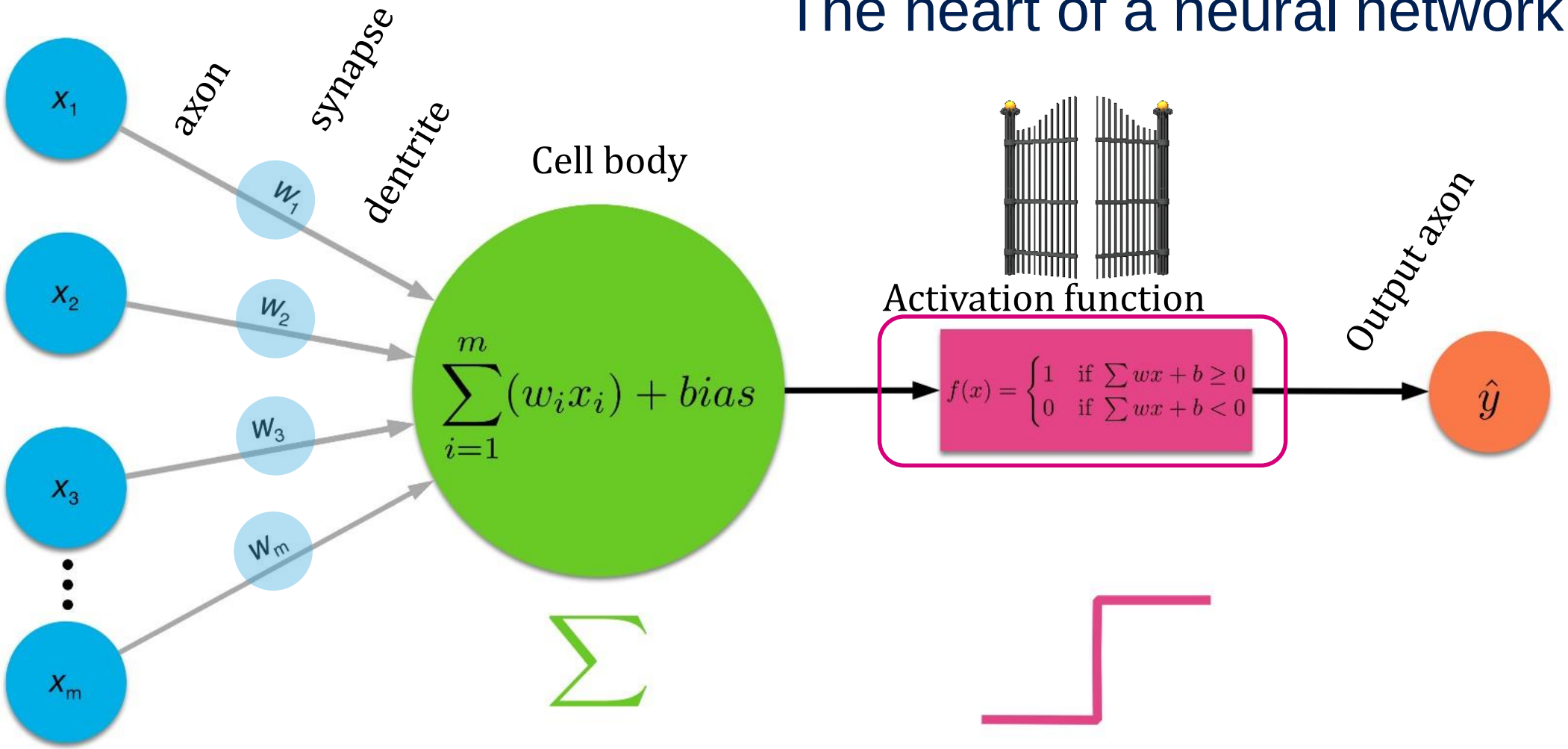
The heart of a neural network



life.augmented

Artificial Neuron

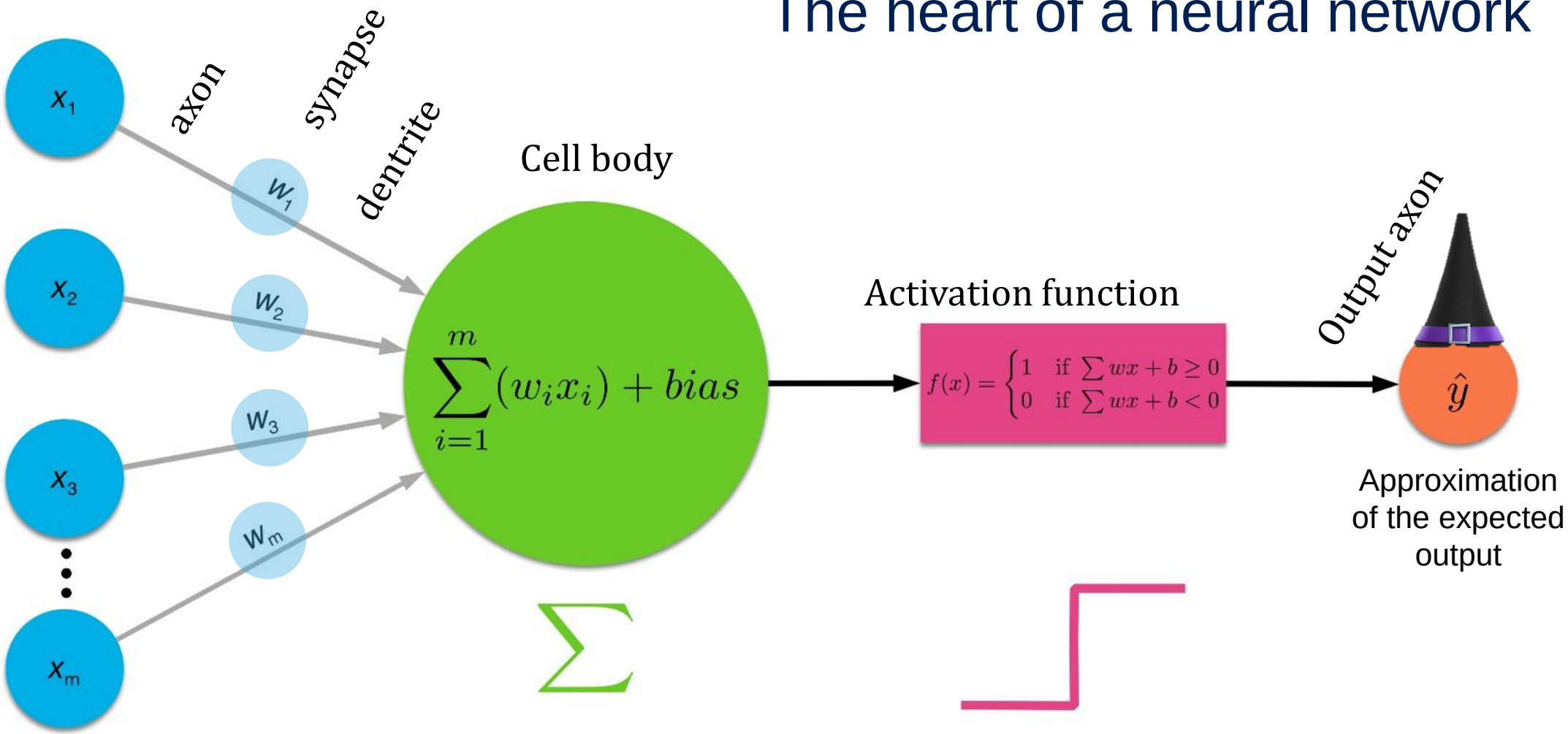
The heart of a neural network



life.augmented

Artificial Neuron

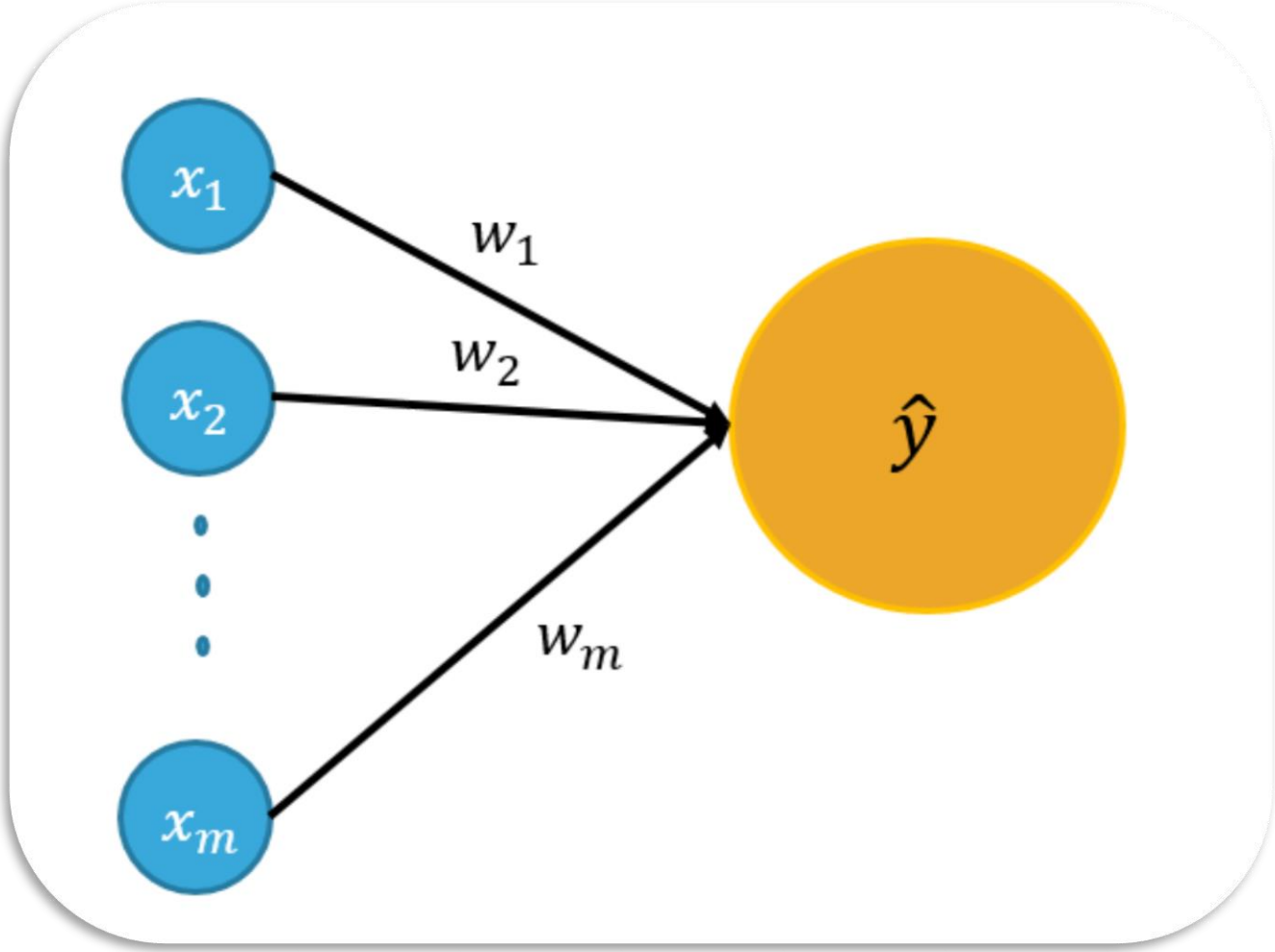
The heart of a neural network



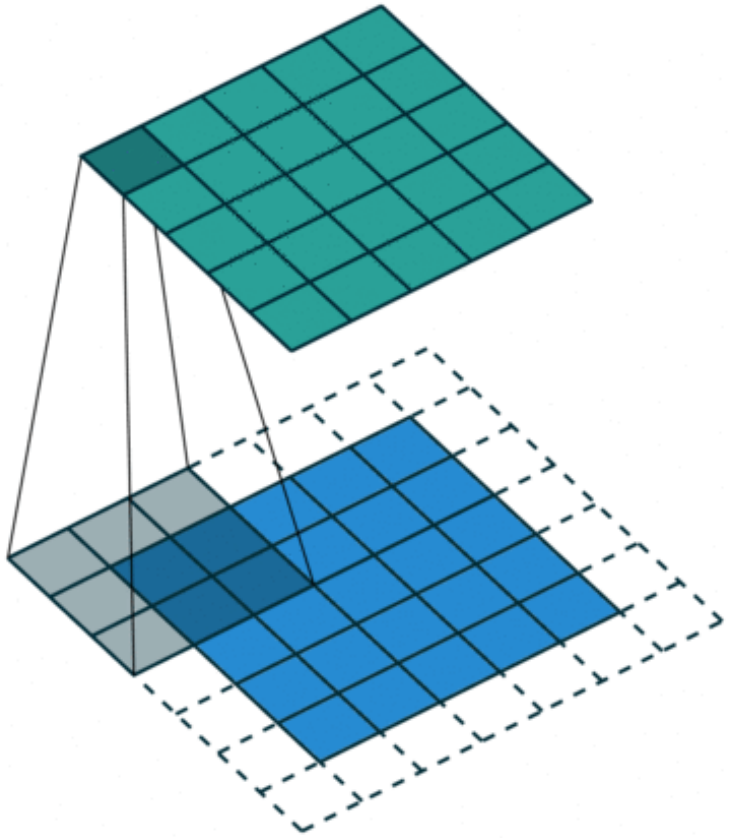
life.augmented

Artificial Neuron

Perceptron: The heart of a neural network

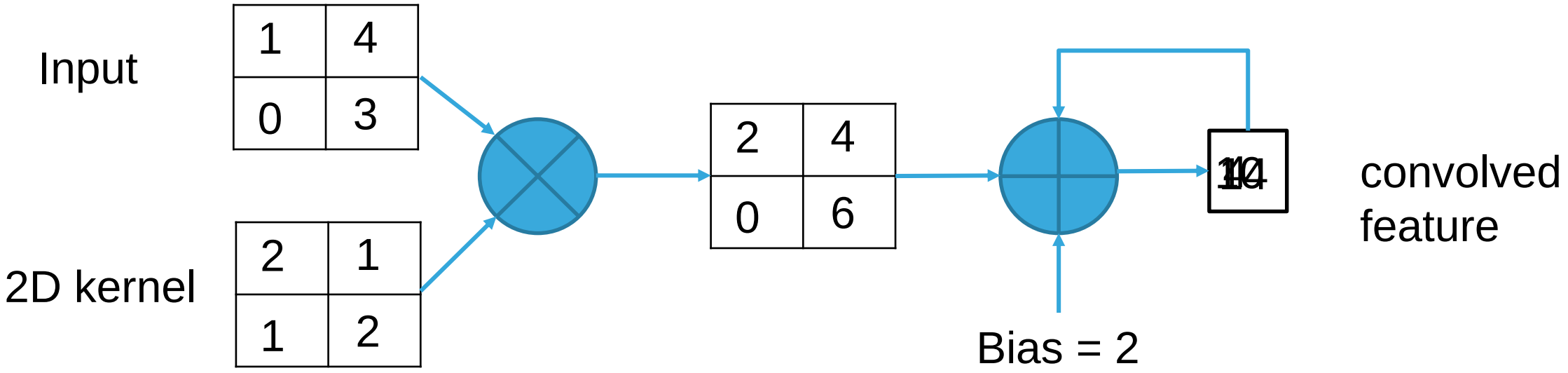


Convolutions



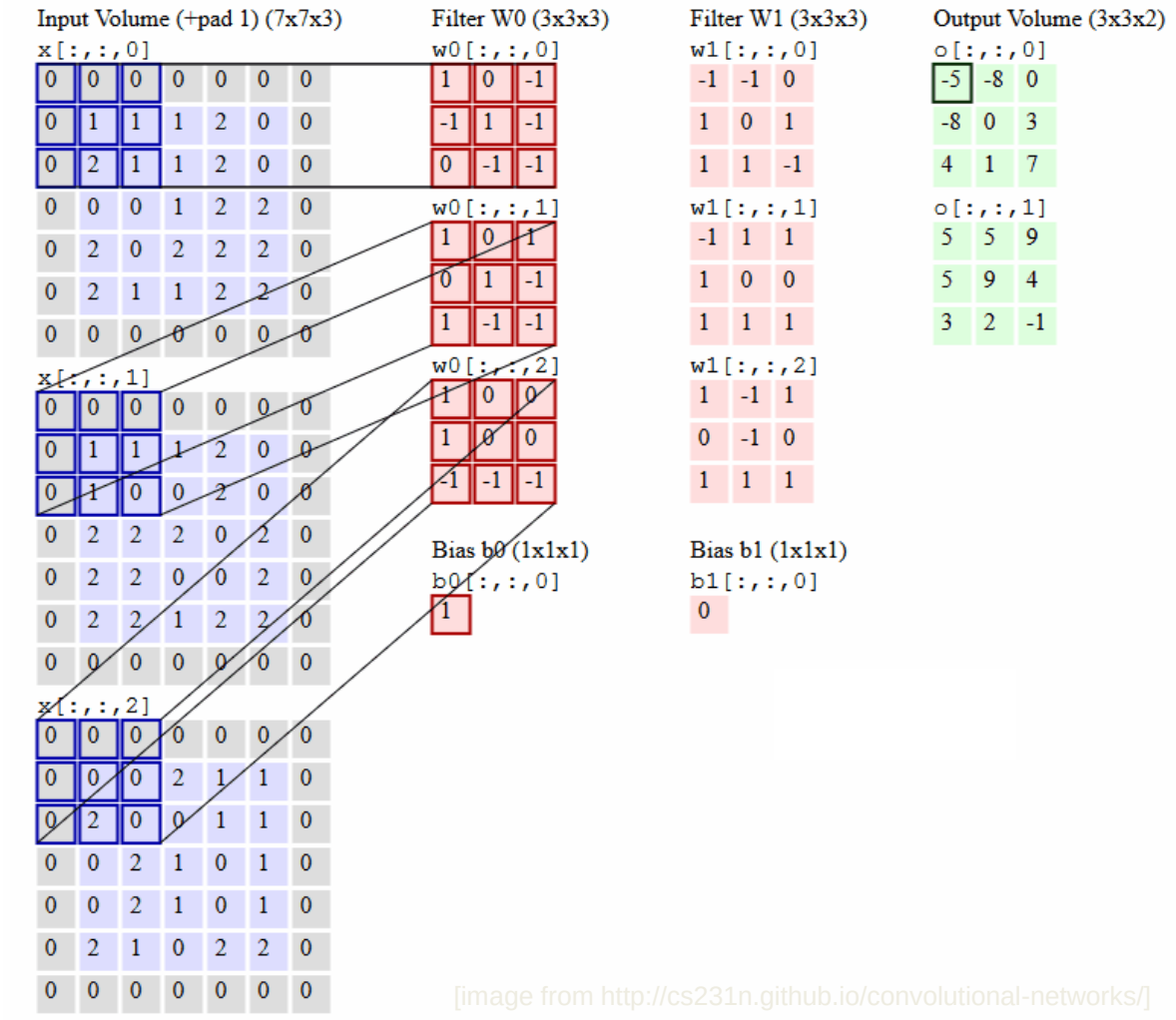
- **Kernel Size**: the field of view of the convolution
- **Stride**: the step size of the kernel when traversing the image.
- **Padding**: defines how the border of a sample is handled.
- **Input & Output Channels**: A convolutional layer takes a certain number of input channels and calculates a specific number of output channels

2D Convolutions



3D Convolutions

- ***Convolution operation***
  - A ***convolution filter*** is a square (or cubic) matrix
    - It is first centered on a pixel of the input image
    - It produces a scalar value: the dot product between the filter and the image region around the pixel
    - By mapping the same procedure on all pixels of the input image,
    - a new image is produced (i.e. a *feature map*)



[image from http://cs231n.github.io/convolutional-networks/]

life.augmented

3D Convolutions

- **Convolution operations (on images)**
  - A ***convolution filter*** is a square (or cubic) matrix
  - In symbols

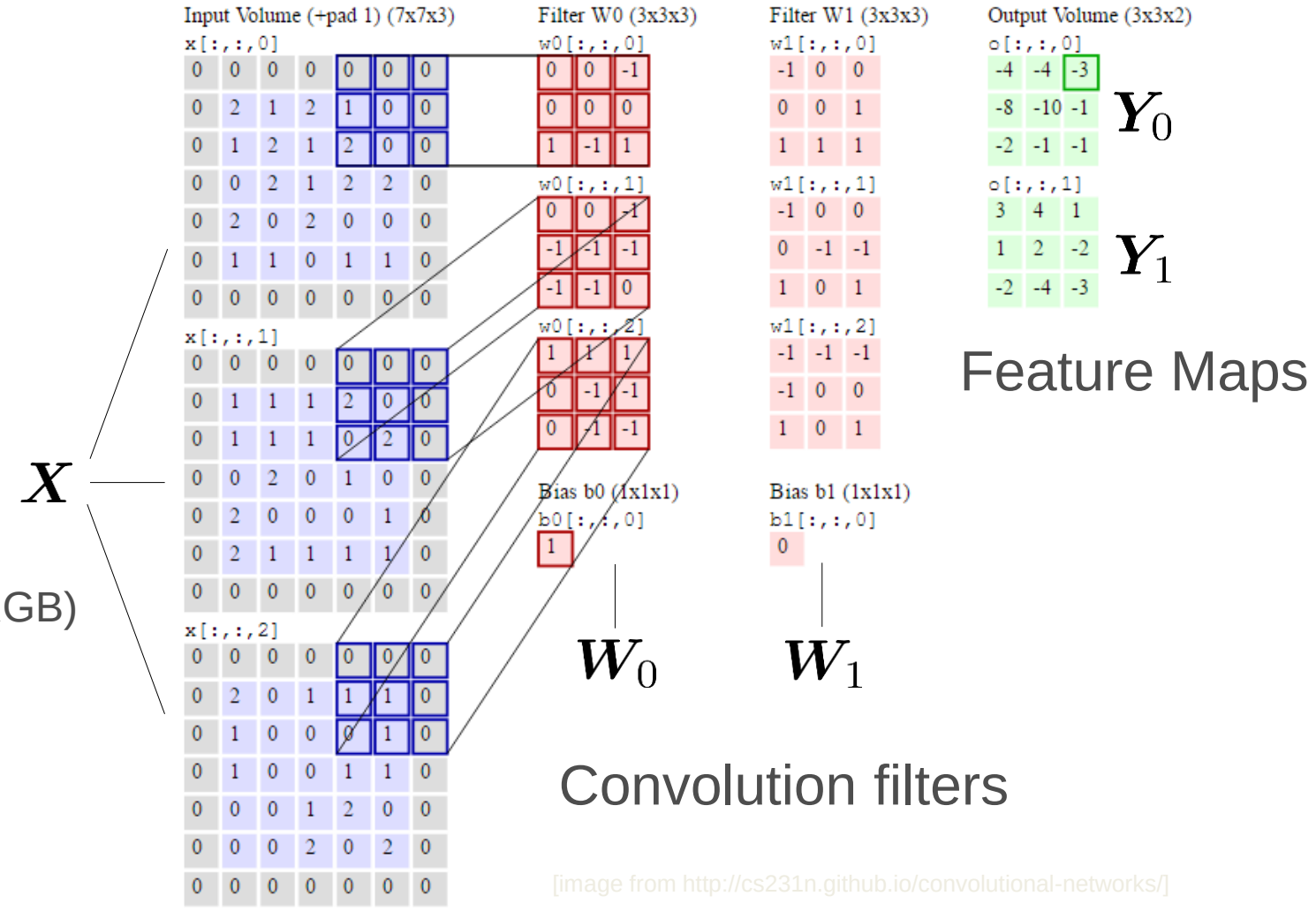
$$\boldsymbol{Y}_i := \boldsymbol{W}_i * \boldsymbol{X}$$

i-th feature map

convolution operator

  - where:

$\boldsymbol{X}$: Input image (e.g. RGB)



Feature Maps

Convolution filters

[image from http://cs231n.github.io/convolutional-networks/]

life.augmented

3D Convolutions

Input Volume (+pad 1) (7x7x3)

x[:,:,0]

0	0	0	0	0	0	0
0	0	0	1	0	2	0
0	1	0	2	0	1	0
0	1	0	2	2	0	0
0	2	0	0	2	0	0
0	2	1	2	2	0	0
0	0	0	0	0	0	0

x[:,:,1]

0	0	0	0	0	0	0
0	2	1	2	1	1	0
0	2	1	2	0	1	0
0	0	2	1	0	1	0
0	1	2	2	2	2	0
0	0	1	2	0	1	0
0	0	0	0	0	0	0

x[:,:,2]

0	0	0	0	0	0	0
0	2	1	1	2	0	0
0	1	0	0	1	0	0
0	0	1	0	0	0	0
0	1	0	2	1	0	0
0	2	2	1	1	1	0
0	0	0	0	0	0	0

Filter W0 (3x3x3)

w0[:,:,0]

-1	0	1
0	0	1
1	-1	1

w0[:,:,1]

-1	0	1
1	-1	1
0	1	0

w0[:,:,2]

-1	1	1
1	1	0
0	-1	0

Bias b0 (1x1x1)

b0[:,:,0]

1

Filter W1 (3x3x3)

w1[:,:,0]

0	1	-1
0	-1	0
0	-1	1

w1[:,:,1]

-1	0	0
1	-1	0
1	-1	0

w1[:,:,2]

-1	1	-1
0	-1	-1
1	0	0

Bias b1 (1x1x1)

b1[:,:,0]

0

Output Volume (3x3x2)

o[:,:,0]

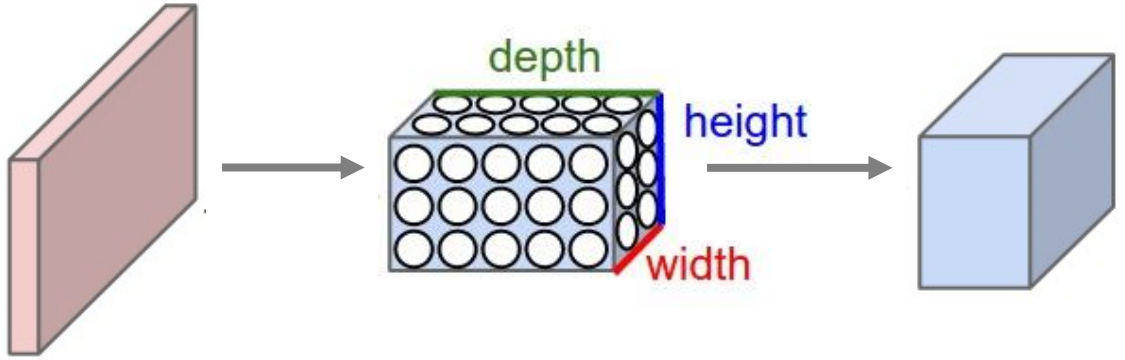
2	3	3
3	7	3
8	10	-3

o[:,:,1]

-8	-8	-3
-3	1	0
-3	-8	-5

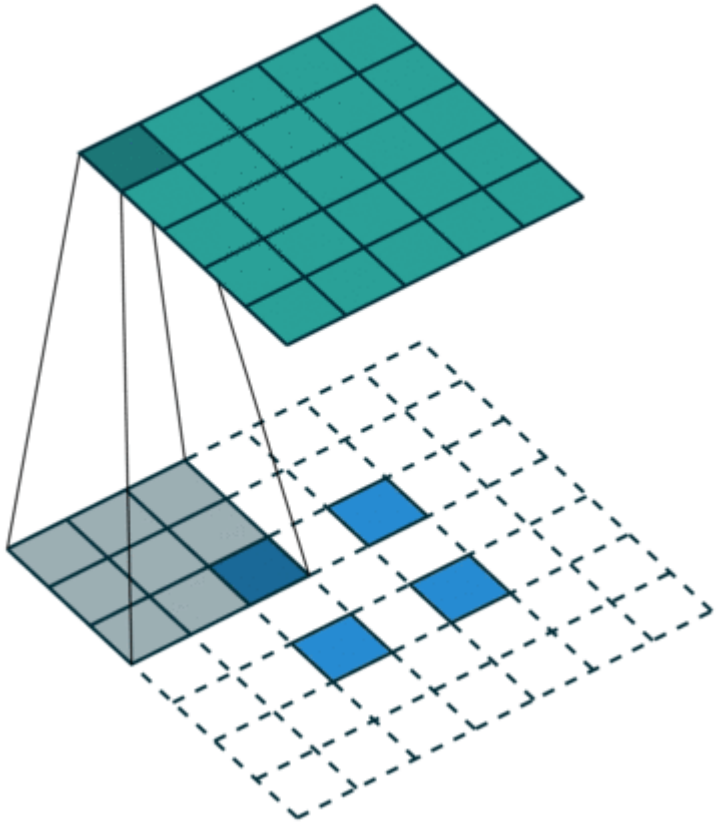
toggle movement

Input image *Convolutional Filters* *Feature Maps*

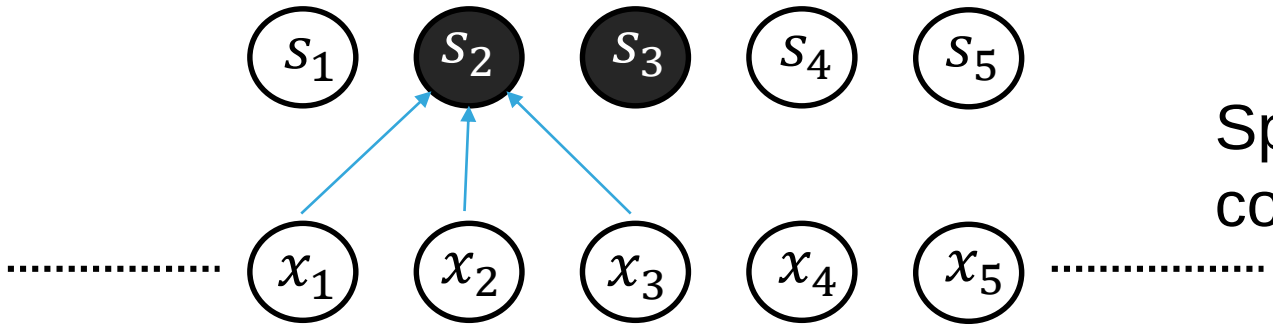


Transposed Convolutions

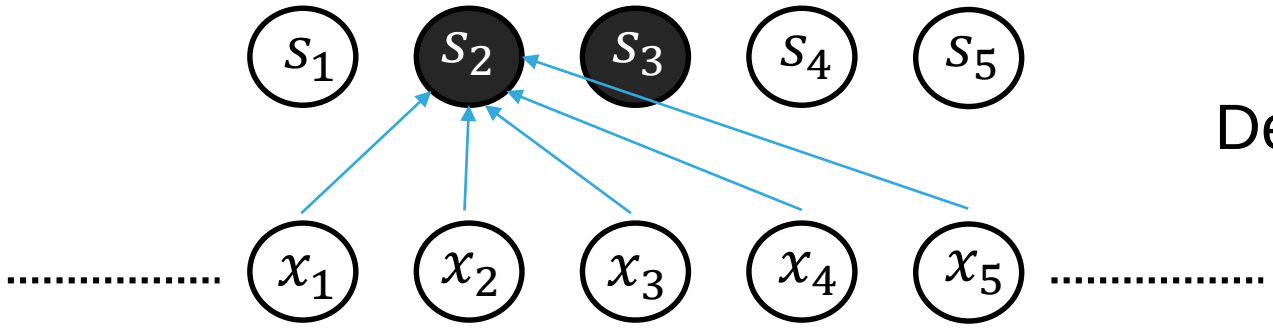
- Also known as Fractionally Strided Convolutions (e.g. 1/2, 1/4 etc)
- Perform some fancy padding on the input
- Not able to numerically reverse convolution followed by down sampling



Sparse Connectivity

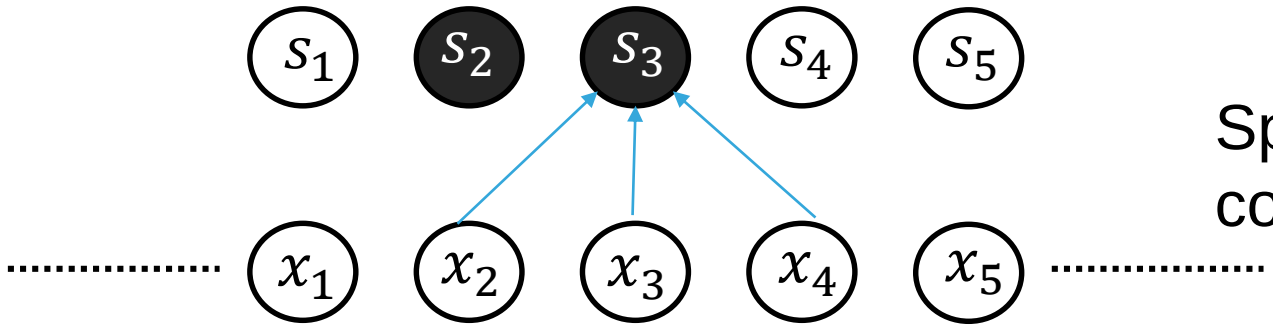


Sparse connections due to small convolution kernel

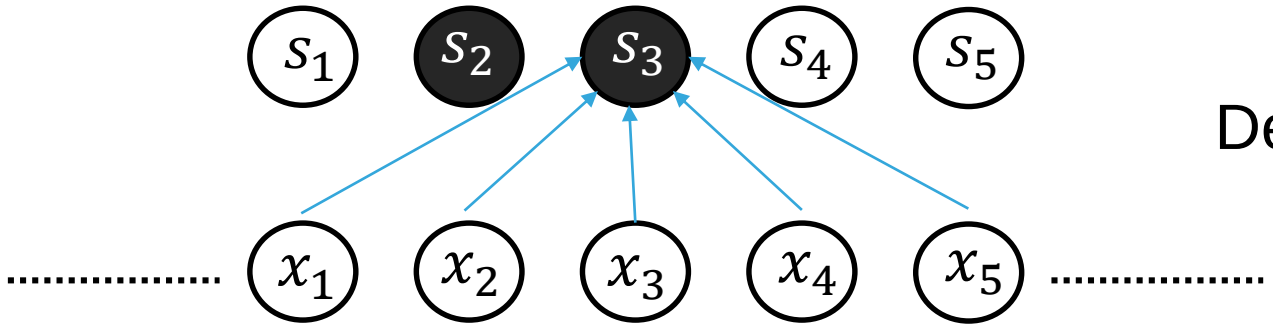


Dense connections

Sparse Connectivity



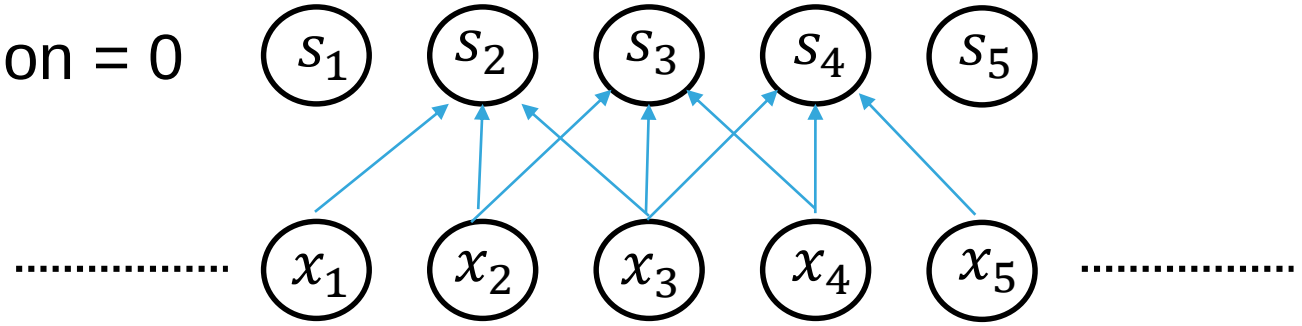
Sparse connections due to small convolution kernel



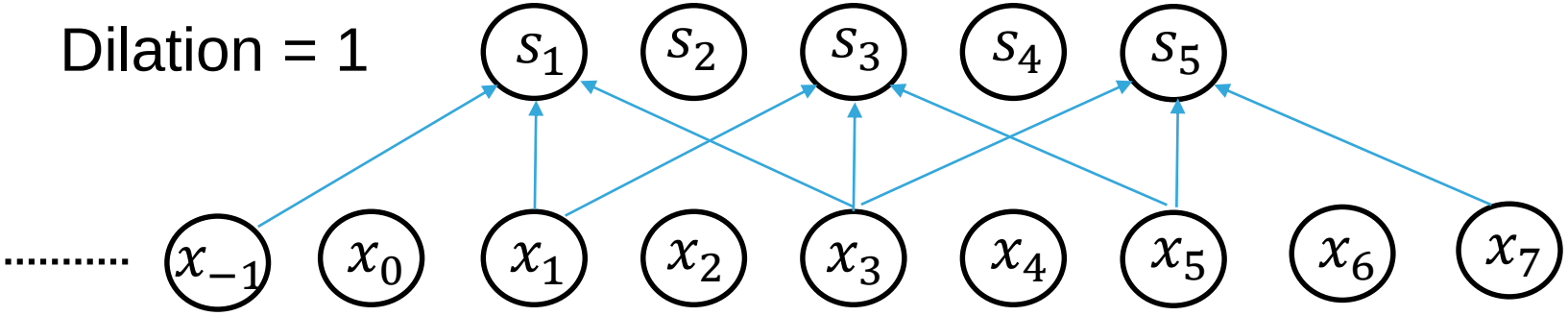
Dense connections

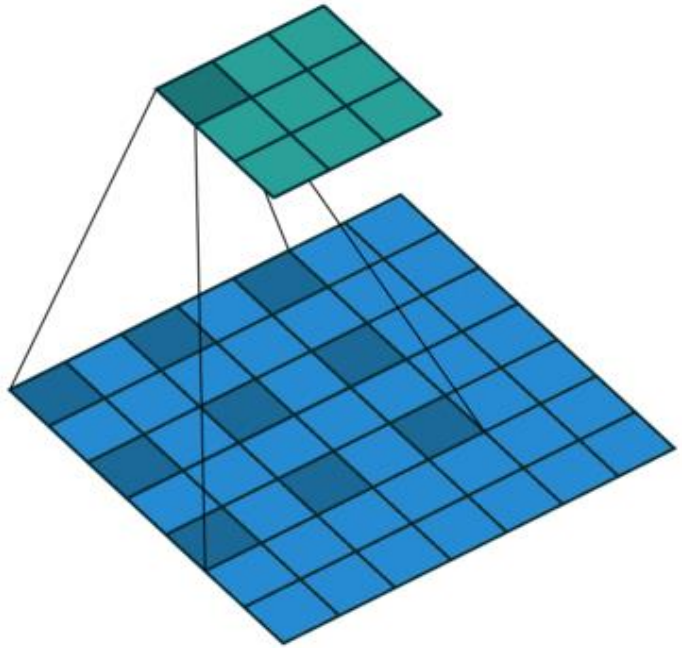
Dilated Convolutions

Dilation = 0



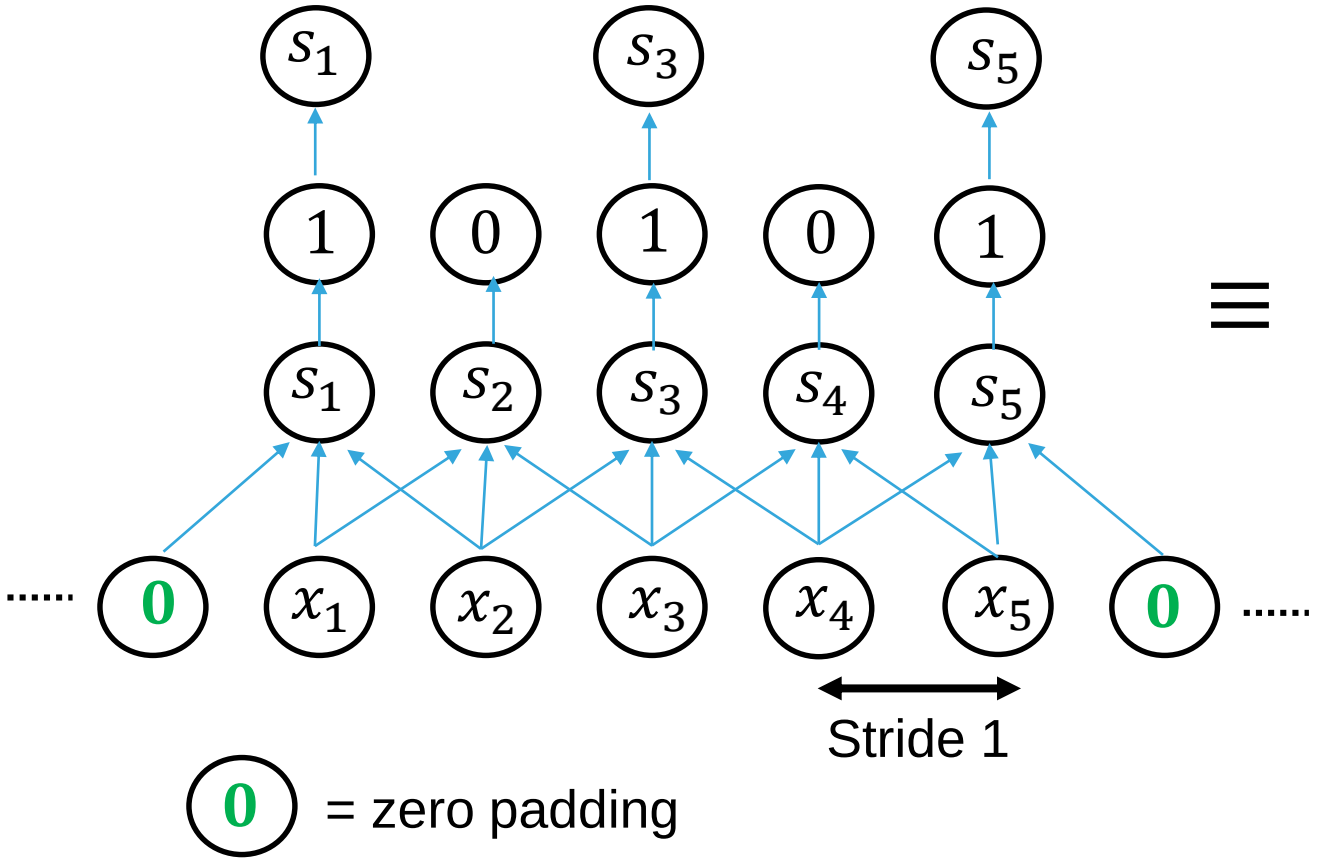
Dilation = 1



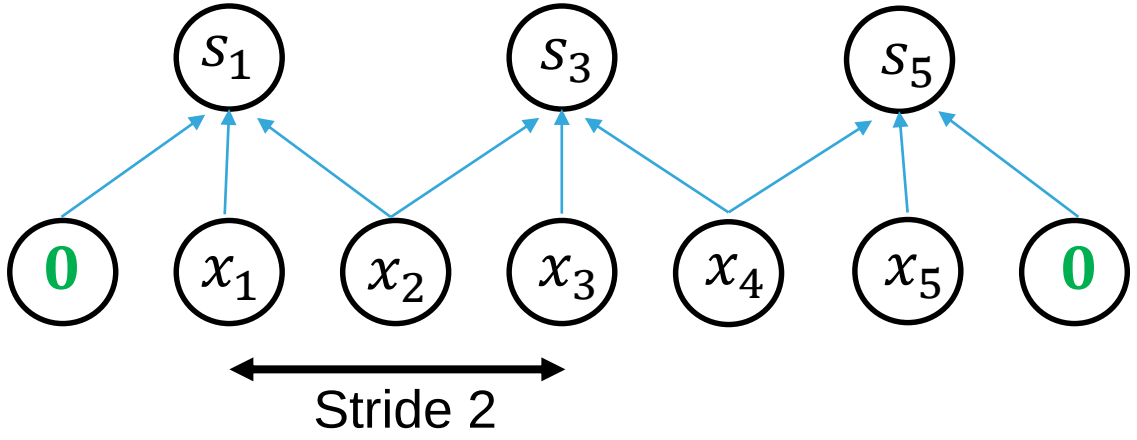


life.augmented

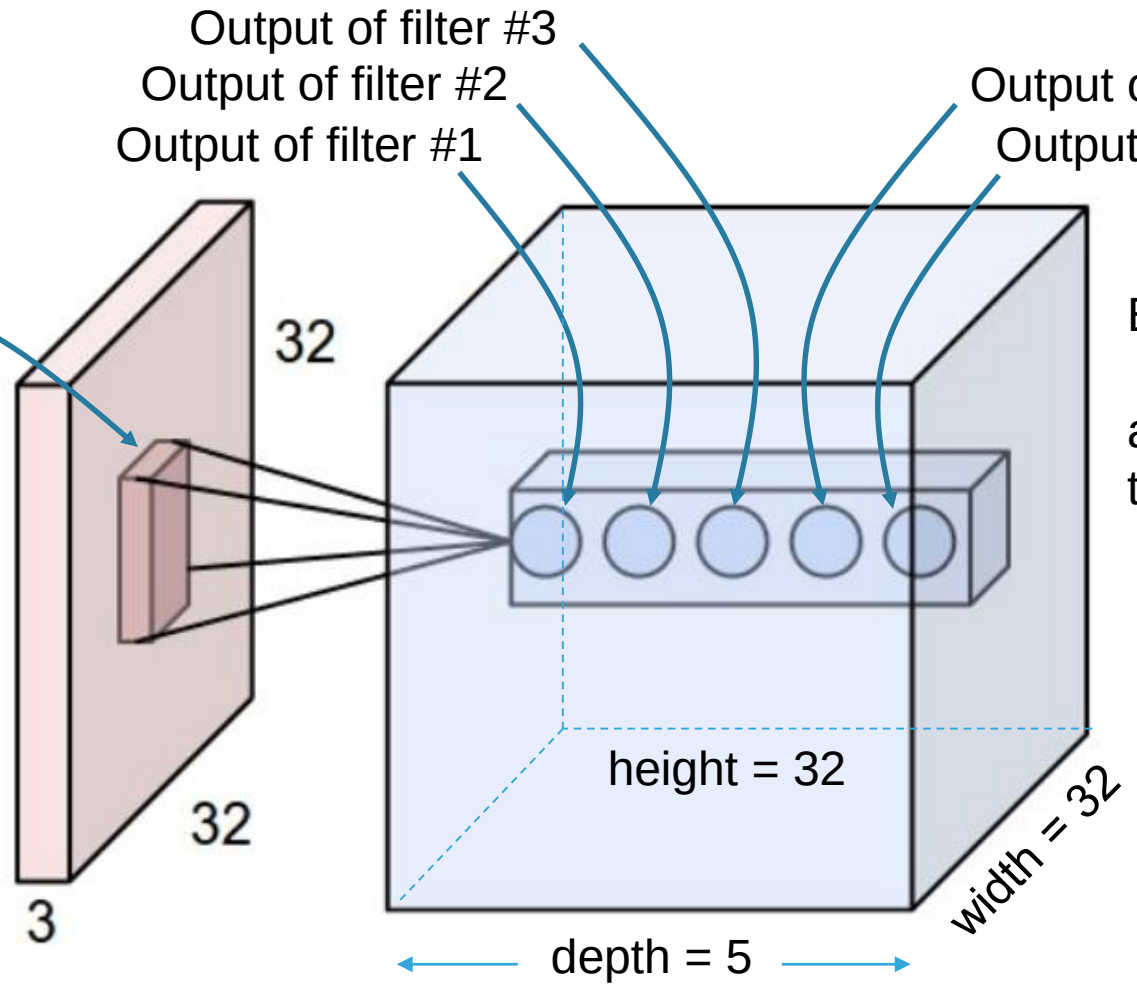
Convolutions with Stride



≡



0 = zero padding



Receptive field or field of view of the filter (filter size, e.g. 5x5x3)

Each filter has $(5 * 5 * 3 + 1) = 76\ weights$ and 75 local (in space= connections to a neuron)

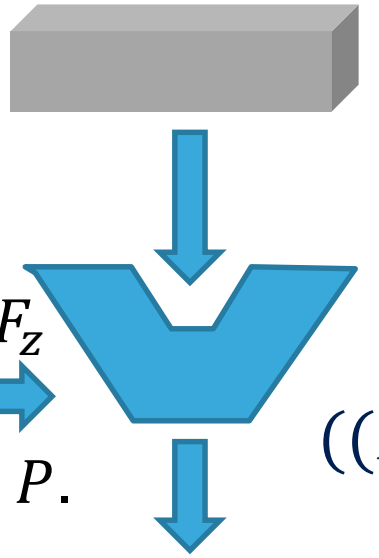
$W = 32\ \#input\ size$
$F = 5\ \#filter\ size$
$P = 2\ \#\ padding$
$S = 1\ \#\ stride$

$$\text{width} = \text{height} = \frac{W-F+2P}{S} + 1 = 32$$

life.augmented

Complexity of 3D Convolutional Layer

$W_1 * H_1 * D_1$

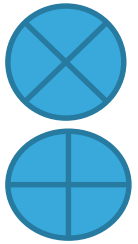


Input feature map

Number of filters K,
Filter spatial extension F_x F_y F_z
The stride S ,
The amount of zero padding P.

$$F_x * F_y * F_z * K * H_2 * W_2$$

$$((F_x * F_y * F_z - 1) + 1) * K * H_2 * W_2$$



$$W_2 = \frac{W_1 - F_x + 2P}{S} + 1$$

$$H_2 = \frac{H_1 - F_y + 2P}{S} + 1$$

$$D_2 = K$$

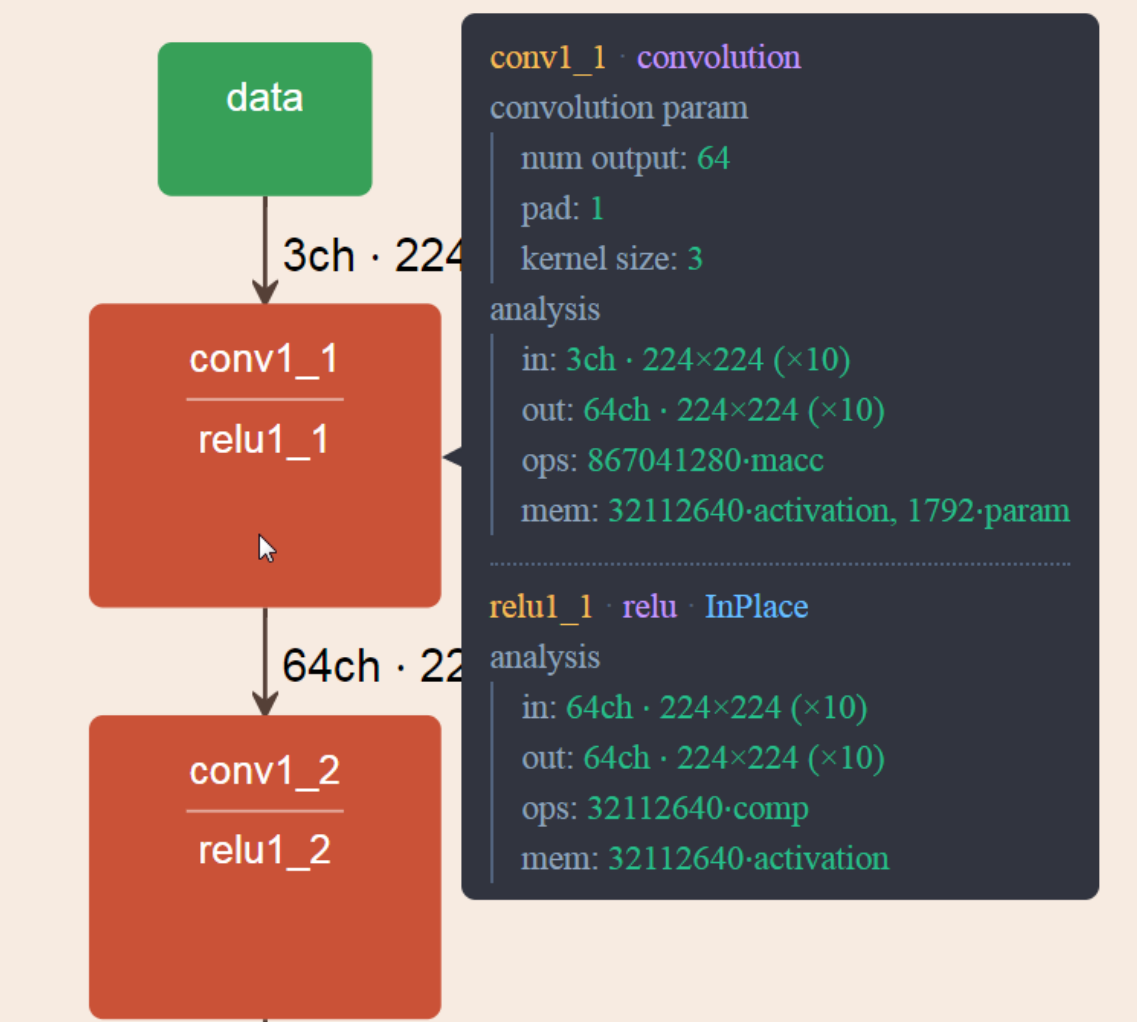
Usually $F_z = D_1$ and no padding is applied on z direction

Output feature map

life.augmented

Netscope CNN Analyzer

- https://dgschwend.github.io/netscope/#/preset/vgg-16
- VGG ILSVRC 16 layers

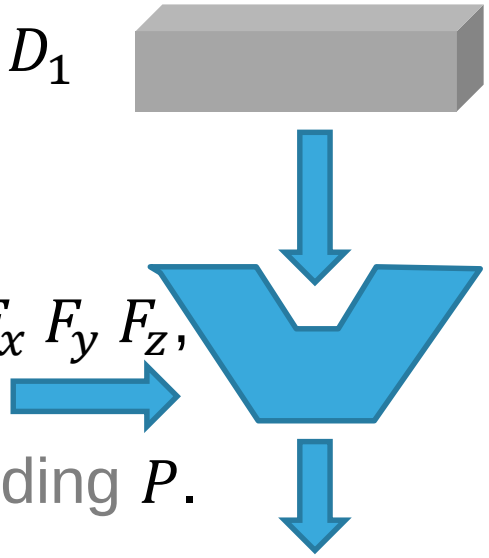


Parameter sharing

- To limit number of parameters in Convolutional Layers.
- Using the example before
  - a volume of size [32x32x5] has 5 depth slices, each of size [32x32]
  - there are 32*32*5 (slide 20) = 5,120 neurons in the Conv Layer
  - each neuron has a own 5*5*3 (filter) = 75 weights and 1 bias.
  - This adds up to 5,120 * 76 = 389,120 parameters for Conv layer itself.
  - **How to reduce it ?**
- **Spatial correlation assumption**: if one feature is useful to compute at some spatial position (x,y), then it should also be useful to compute at a different position (x_2, y_2).
- **Solution**: To constrain the neurons in each depth slice to use the same weights and bias.
- Only 5 unique set of weights, one for each depth slice, for a total of 5 slices*(5*5*3 weights per slice) = 375 unique weights, (+5 biases).

Summary of Convolutional Layer

$W_1 * H_1 * D_1$



Number of filters K,
Filter spatial extension F_x F_y F_z,
The stride S ,
The amount of zero padding P.

With parameter sharing, the layer requires $F_x * F_y * F_z * D_1$ parameters per filter, for a total of $F_x * F_y * F_z * D_1 * K + K$ biases.

$$W_2 = \frac{W_1 - F_x + 2P}{S} + 1$$

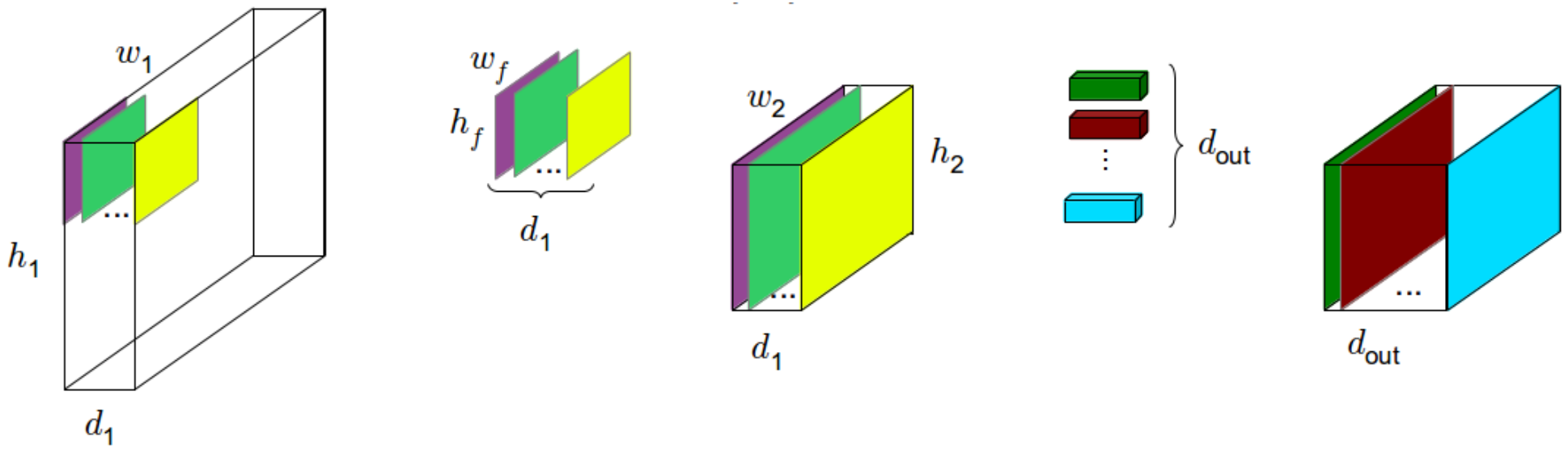
$$H_2 = \frac{H_1 - F_y + 2P}{S} + 1$$

$$D_2 = K$$

Usually $F_z = D_1$ and no padding is applied on z direction

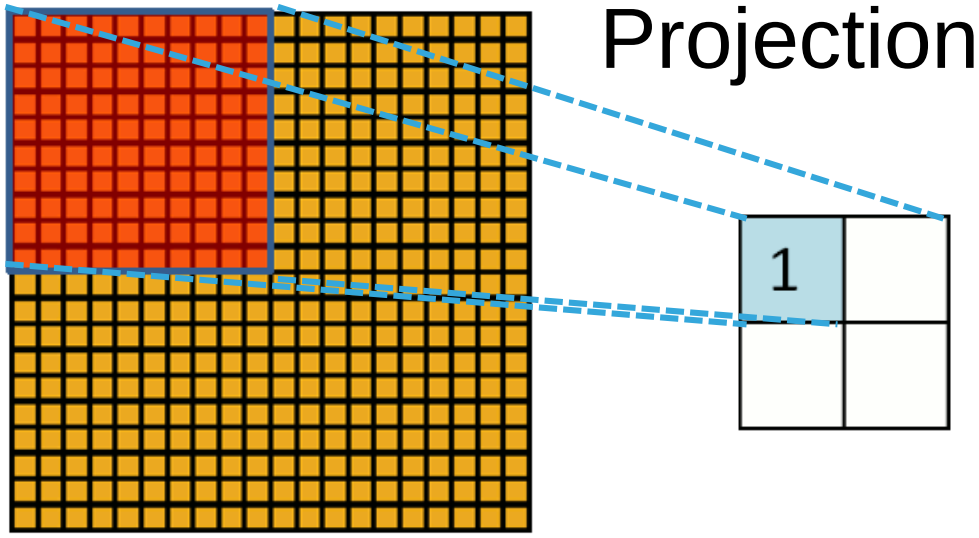
life.augmented

Depth wise separable convolution



- Consider $d_1 = 16$ channels, $w_f = h_f = 3$ each kernel (2D) → 16 feature maps.
- Traverse these 16 feature maps with $d_{out} = 32$, $w_2 = h_2 = 1$ convolutions each
- This results in $656 = [16 * (3 * 3) + 16 * (32 * 1 * 1)]$ parameters opposed to the $4608 = (16 * 32 * 3 * 3)$ parameters from non depth separable filtering.

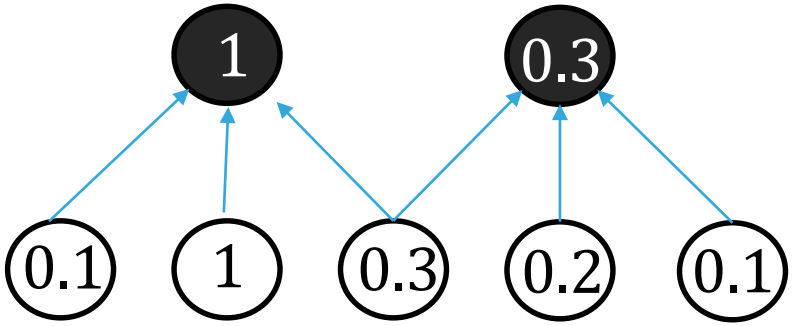
Pooling Features



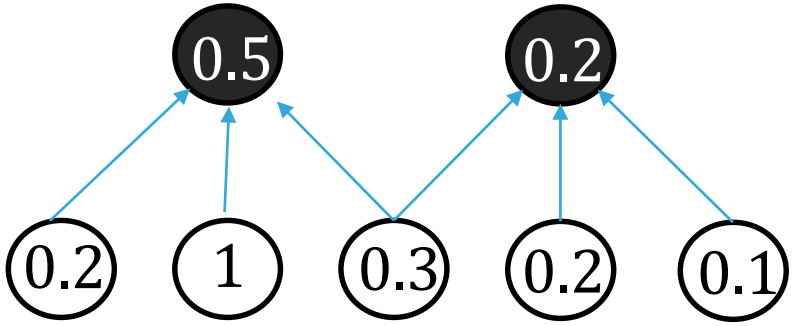
Convolved Feature Map

Pooled Feature Map

Pooling features



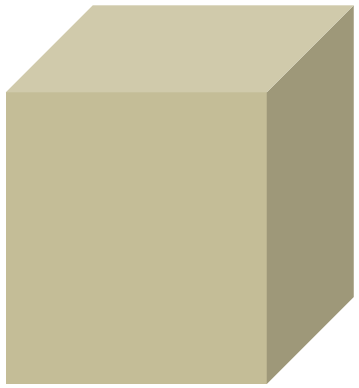
Max Pooling, 3:1



Average Pooling, 3:1

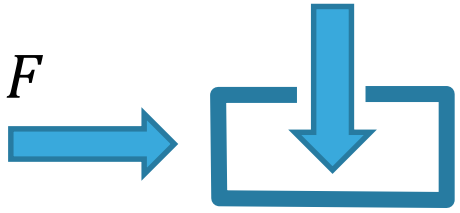
Summary of Pooling Layer

$W_1 * H_1 * D_1$

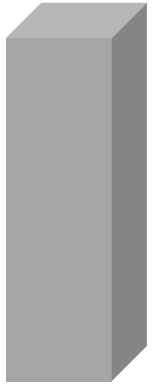


Input Feature Map

Spatial extension F
The stride S



No parameters



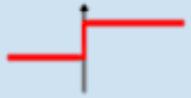
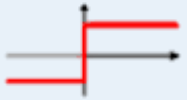
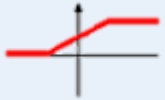
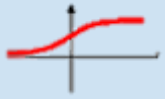
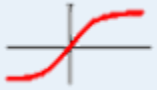
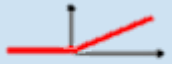
Output Feature Map

$$W_2 = \frac{W_1 - F}{S} + 1$$

$$H_2 = \frac{H_1 - F}{S} + 1$$

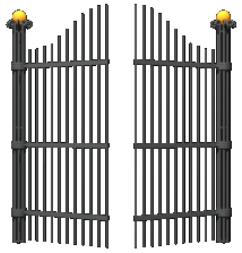
$$D_2 = D_1$$

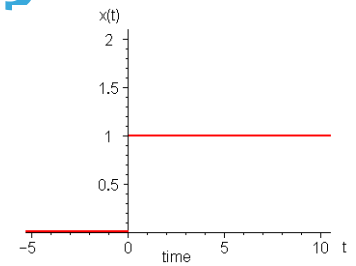
$z = f(\sum_{i=1}^{m} w_i x_i + bias)$ Activation f Functions

Name	Plot	Function	Examples
Unit Step		$\emptyset(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant
Sign (Signum)		$\emptyset(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant
Linear		$\emptyset(z) = z$	Adaline, linear regression
Piece-wise linear		$\emptyset(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z0 < \frac{1}{2}, \\ 1, & z > 0, \end{cases}$	Support vector machine
Logistic (sigmoid)		$\emptyset(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi layer-neural networks
Hyperbolic Tangent		$\emptyset(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi layer neural networks
Rectified Linear Unit (ReLU)		$\emptyset(z) = \begin{cases} z, & z > 0 \\ 0, & z \leq 0 \end{cases}$	Regression, approximation, multi layer neural network

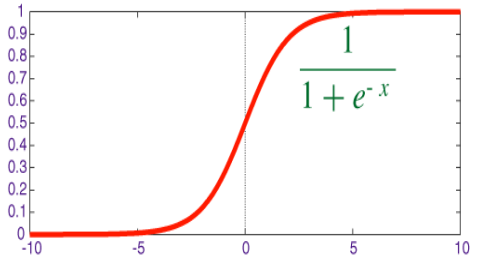
Most widely used activations

- Unit step
  - Threshold

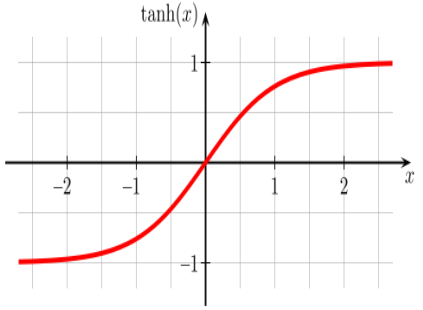




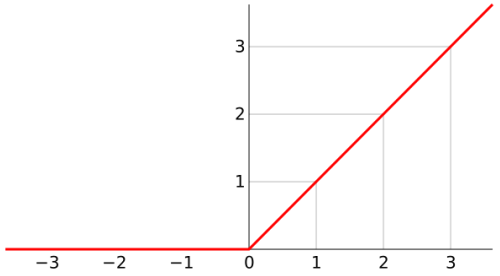
- Sigmoid Function
  - Like a step function but smoother
  - Best to predict probabilities



- Tan hyperbolic
  - Stretched out version of the sigmoid function



- ReLU
  - Computationally efficient



- **Function choice depends on the characteristics of the data.**
- **For example Sigmoid Function works good for classification purposes, resulting in Faster training and convergence.**
- **ReLU is good for approximation. As it is simple so always start from this if you don't know the data characteristics. Helps against gradient vanishing**
- **We can also define custom activations.**

Most widely used activations

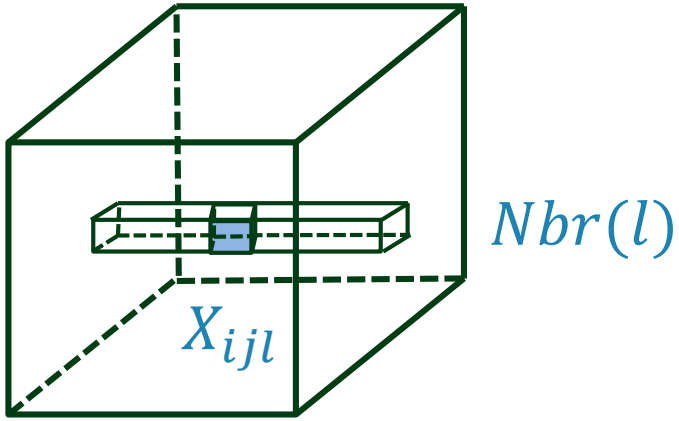
- The Softmax function, or normalized exponential function
- A generalization of the logistic function
- Squeeze the K-dimensional input vector of real values into values in the range [0, 1].

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, for\ j = 1, 2, 3, \ldots, K.$$

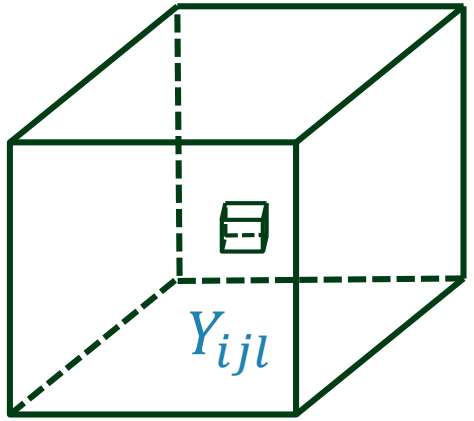
Local Response Normalization

- Compensating the tendency of ReLu to output large values

Activation Feature Map



Normalized Activation Feature Map

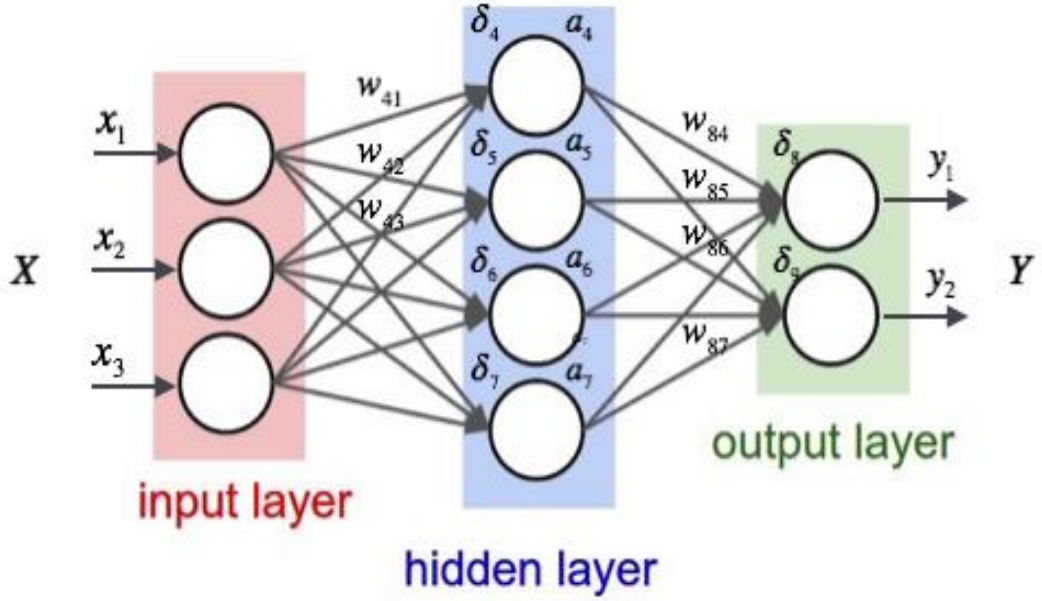


$$Y_{ijl} = \frac{X_{ijl}}{(\alpha + \beta \sum_{k \in Nbr(l)} {X_{ijl}}^2)^{\gamma}}$$

α, β, γ *are hyper-parameters*

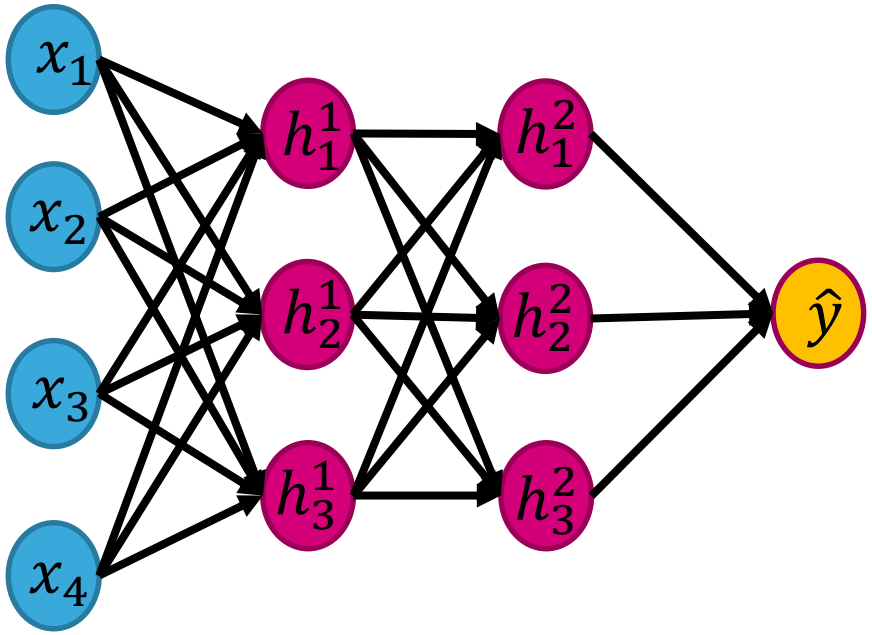
Layers of a Neural Network

- Neural network has three types of layers
- Input layer
  - Can be from other neurons or feature inputs
    - Age, height, weight, pixels in the images etc.
- Hidden layers (one or more)
  - Real power lies here
  - Adding more neurons to the network
- Output layer
  - Gives the output we want to predict
    - Probability of rain
    - Object class
    - Disease is fatal or not…



Neural Networks

- Notations



$$x_n, n = \{1, 2, 3, \dots\}$$

$$h_p^m, m = \{1, 2, 3, \dots\}, p = \{1, 2, 3, \dots\}$$

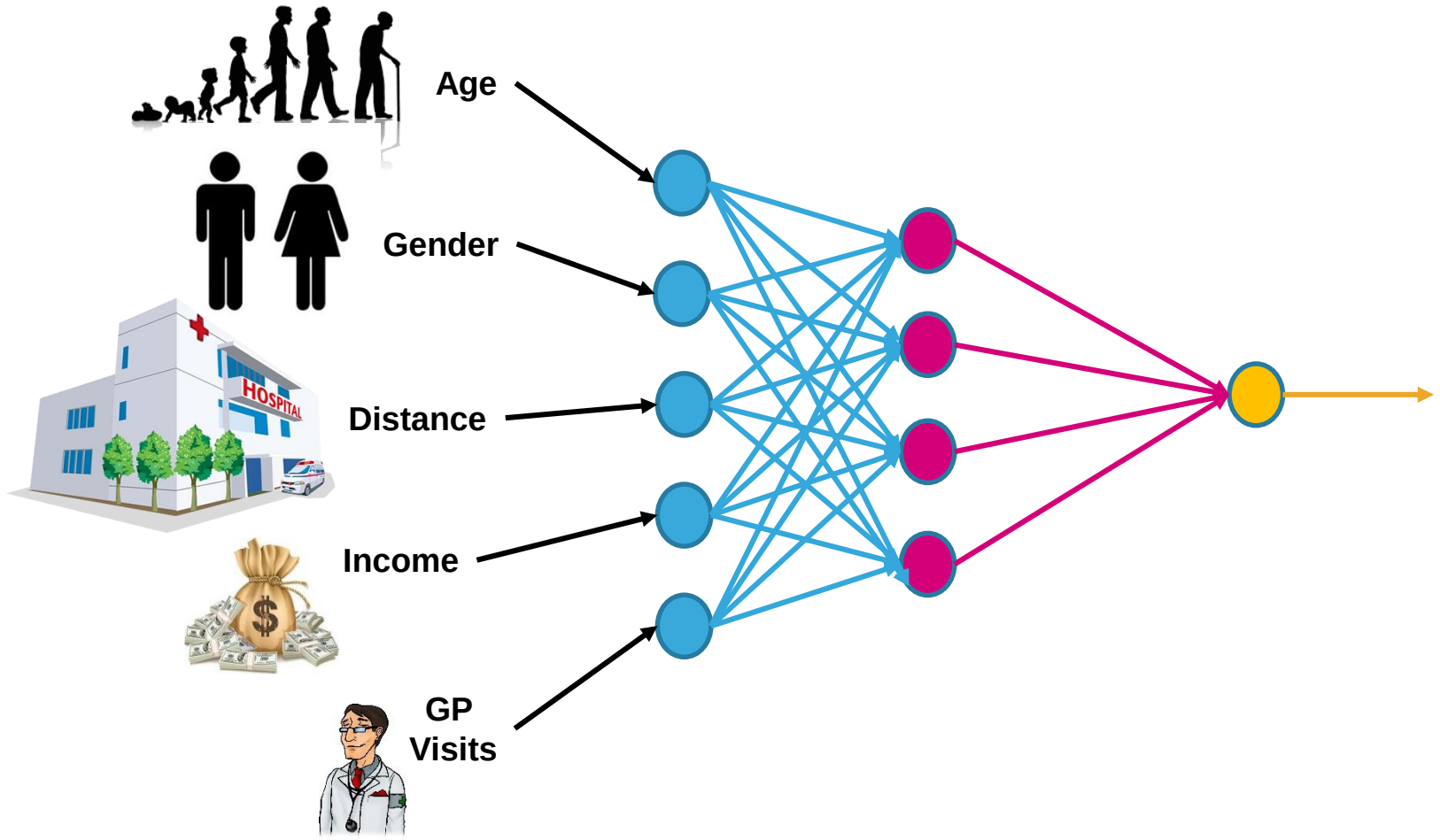
Input Layer

Hidden Layers

Output Layer

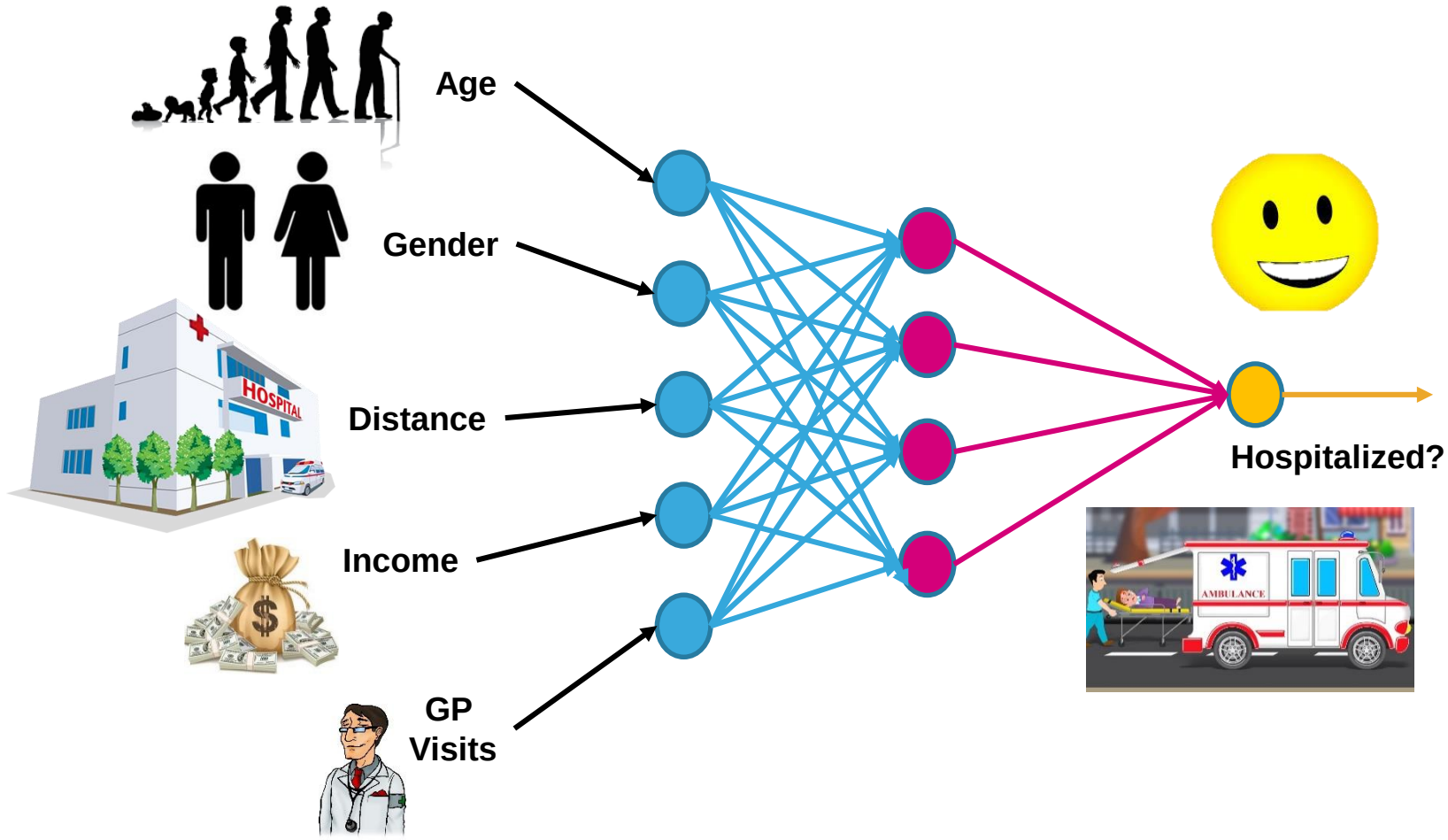
Example of a Neural Network

- To predict if a person has to be hospitalized given
  - Age
  - Gender
  - Distance from hospital
  - Income
  - Number of General Physician (GP) visits
- Let us suppose to train neural network, which means to compute all the weights so that predictions are accurate.
- Consider to have
  - Age = 65
  - Gender = Female
  - Distance, income and GP visit high



Example of a Neural Network

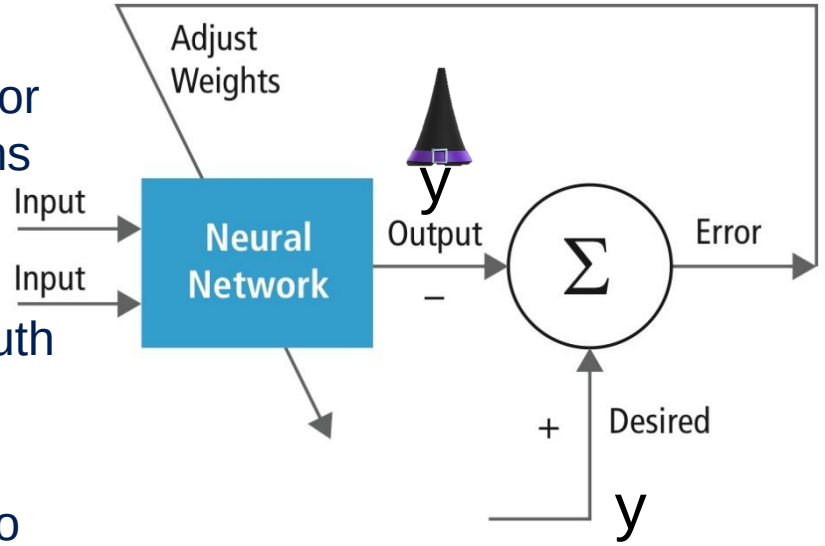
- To predict if a person has to be hospitalized given
  - Age
  - Gender
  - Distance from hospital
  - Income
  - Number of General Physician (GP) visits
- Let us suppose to train neural network, which means to compute all the weights so that predictions are accurate.
- Consider to have
  - Age = 65
  - Gender = Female
  - Distance, income and GP visit high

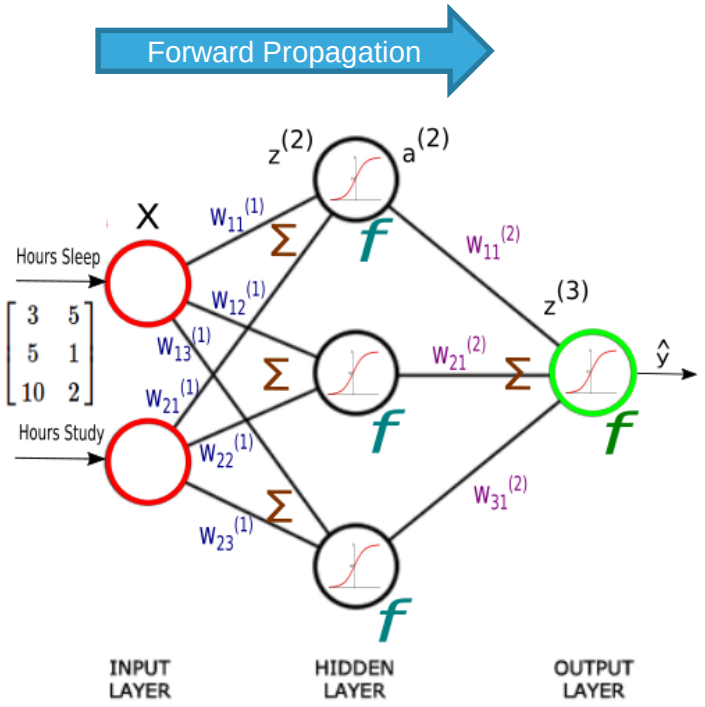


life.augmented

Training neural networks

- In supervised learning an assumption is to have a relatively **large labeled dataset**.
- Feed all the samples as inputs to get an output. Called forward propagation or inference run outputs.
- At start the weights can be randomized or predefined depending on the applications scenario.
- The result $\hat{y}$ is compared with ground truth output y.
- The task is to make the output value $\hat{y}$ to be as close to y as possible reducing the error expressed as Loss functions $L(\tilde{y}, y)$.



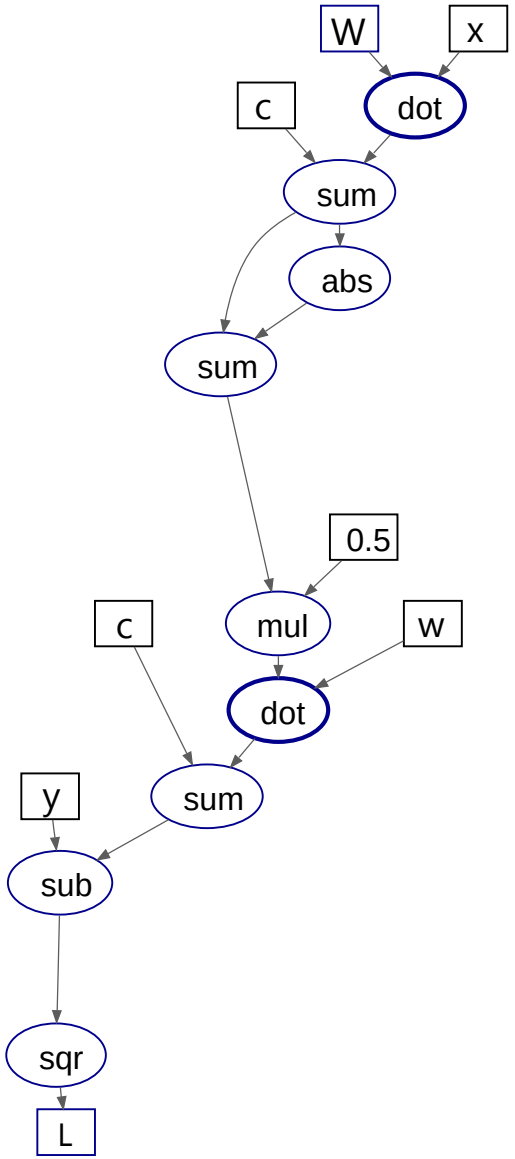


$$L(\tilde{y}, y) = (\boldsymbol{w} \cdot \mathrm{ReLU}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{c}) + c - y)^2 = \text{error}$$

life.augmented

Forward Propagation

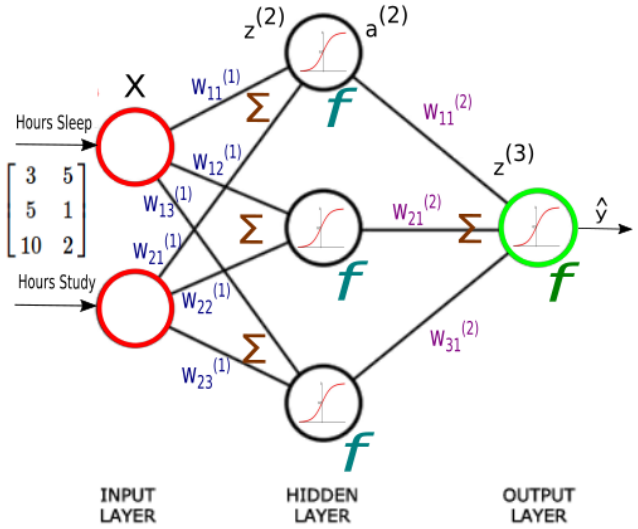
- Let's do it (*in a graphical way*)

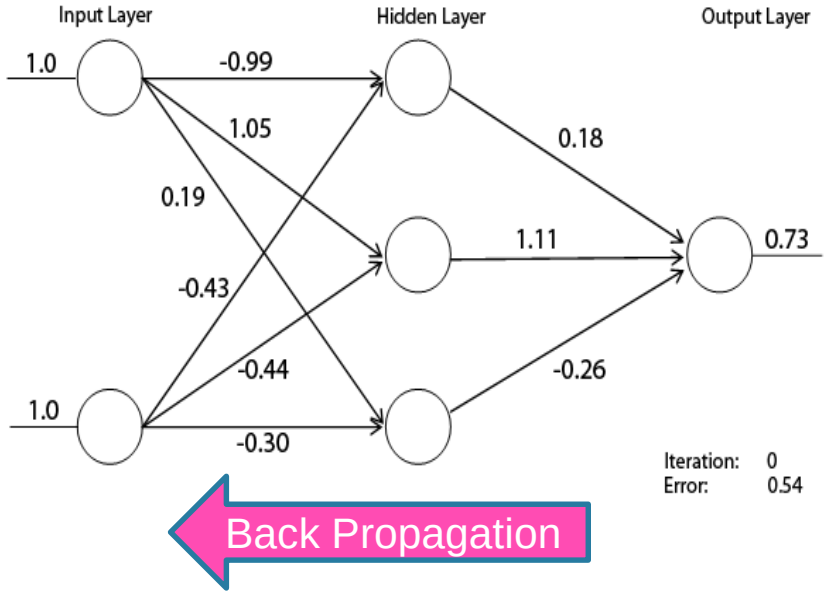


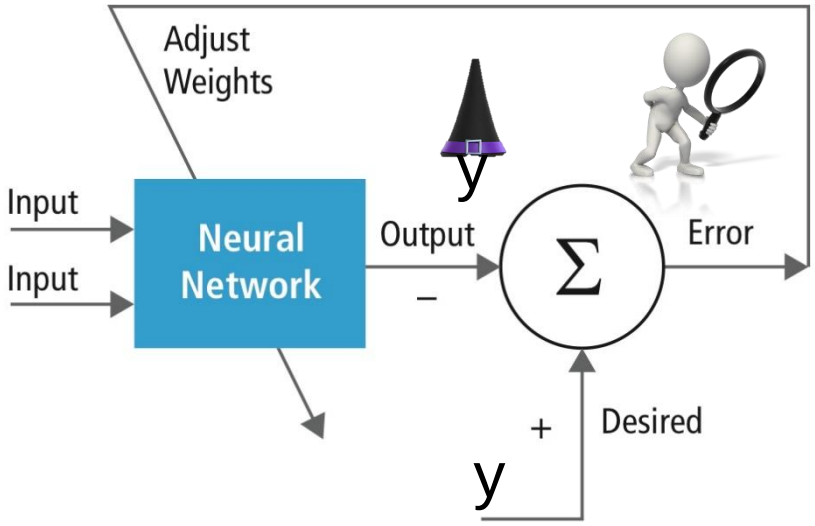
$$L(\tilde{y}, y) = (\boldsymbol{w} \cdot \max(0, \boldsymbol{W}\boldsymbol{x} + \boldsymbol{c}) + c - y)^2$$

Element-wise loss, with ReLU as non-linearity

Training neural networks







- Go back and adjust the weights slowly. Aim is

$$Error_T < Error_{T-1}$$

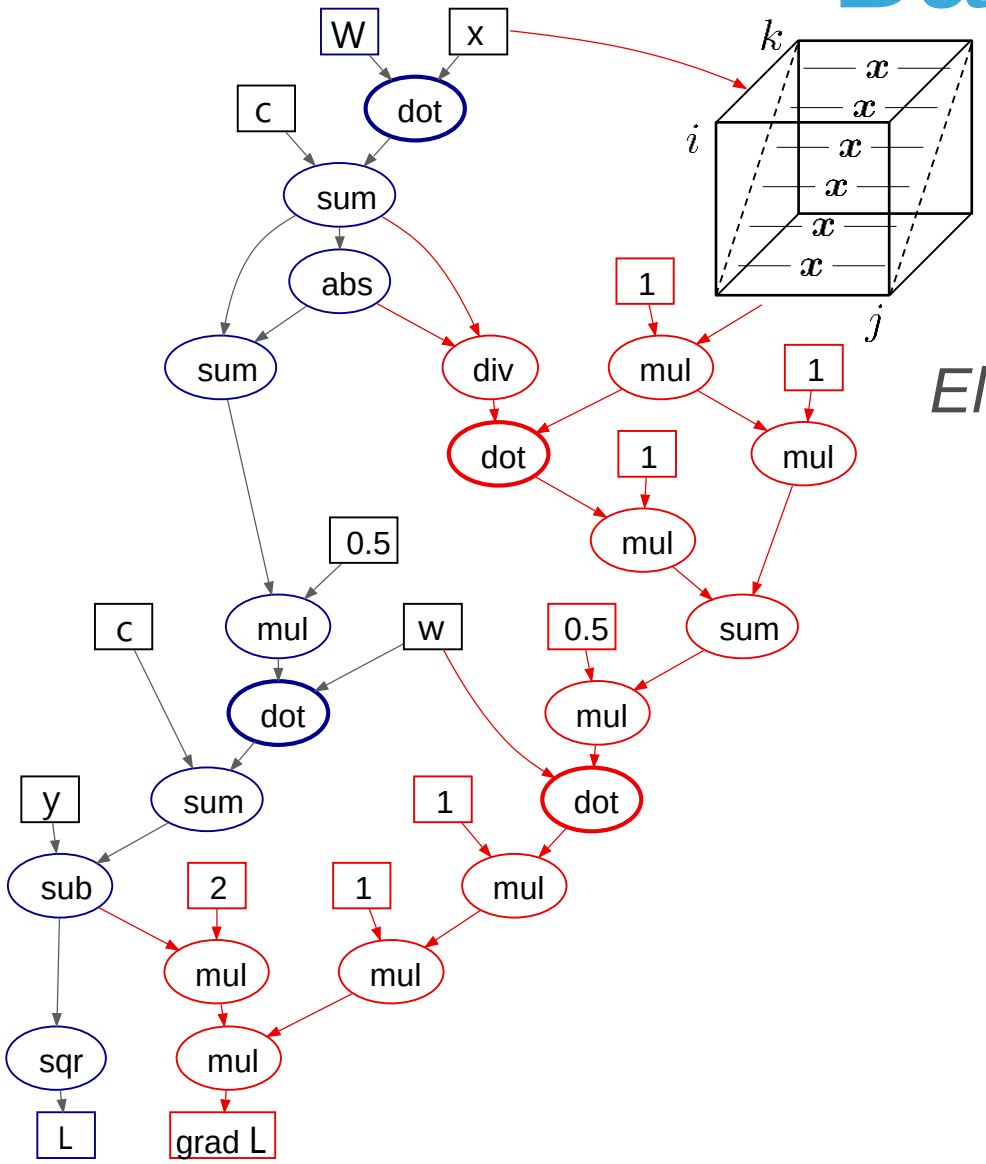
- Repeat this process until the error we get is very small.

$$\lim_{\in \to 0} Error_T < \in$$

life.augmented

Backward Propagation

- Let's do it (*in a graphical way*)



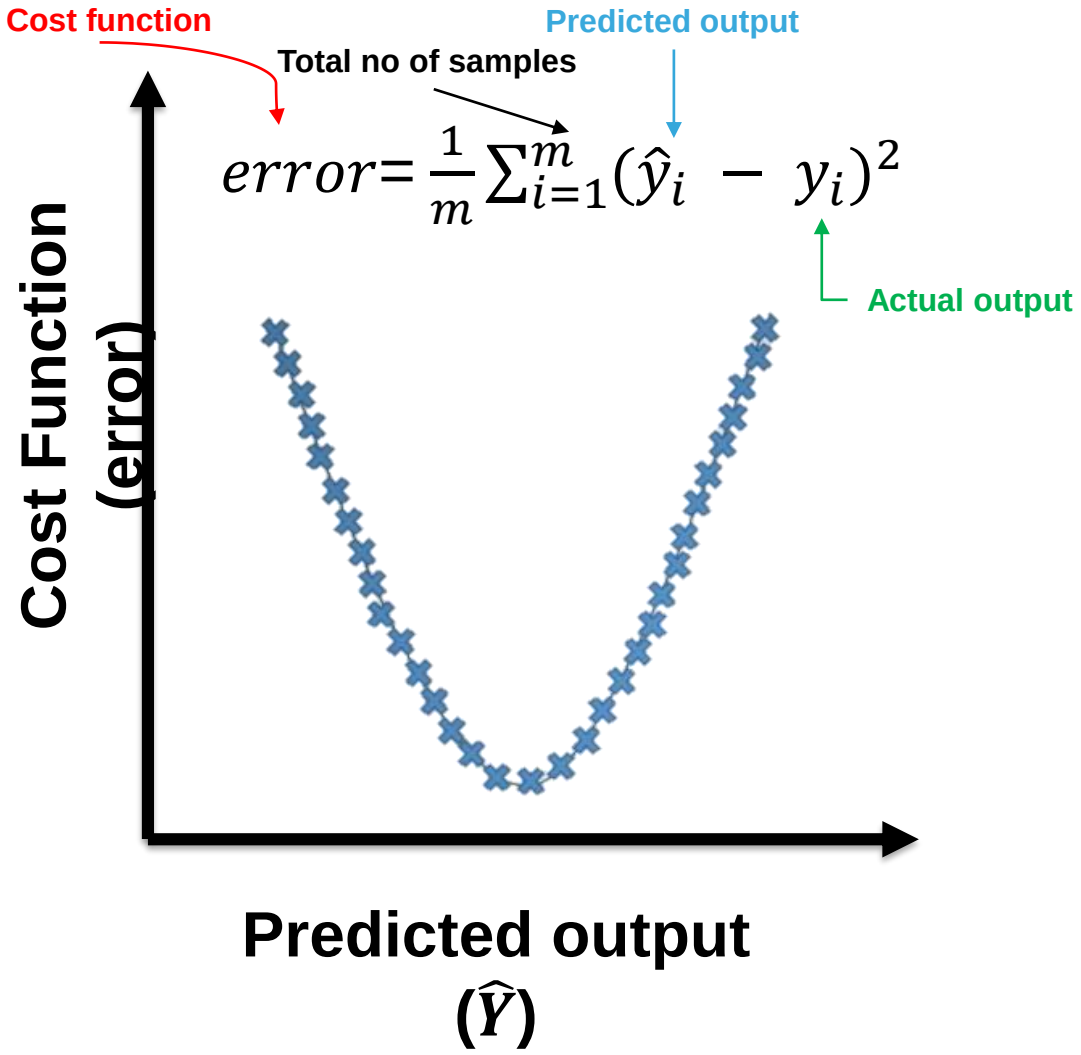
$$L(\tilde{y}, y) = (\boldsymbol{w} \cdot \max(0, \boldsymbol{W}\boldsymbol{x} + \boldsymbol{c}) + c - y)^2$$

Element-wise loss, with ReLU as non-linearity

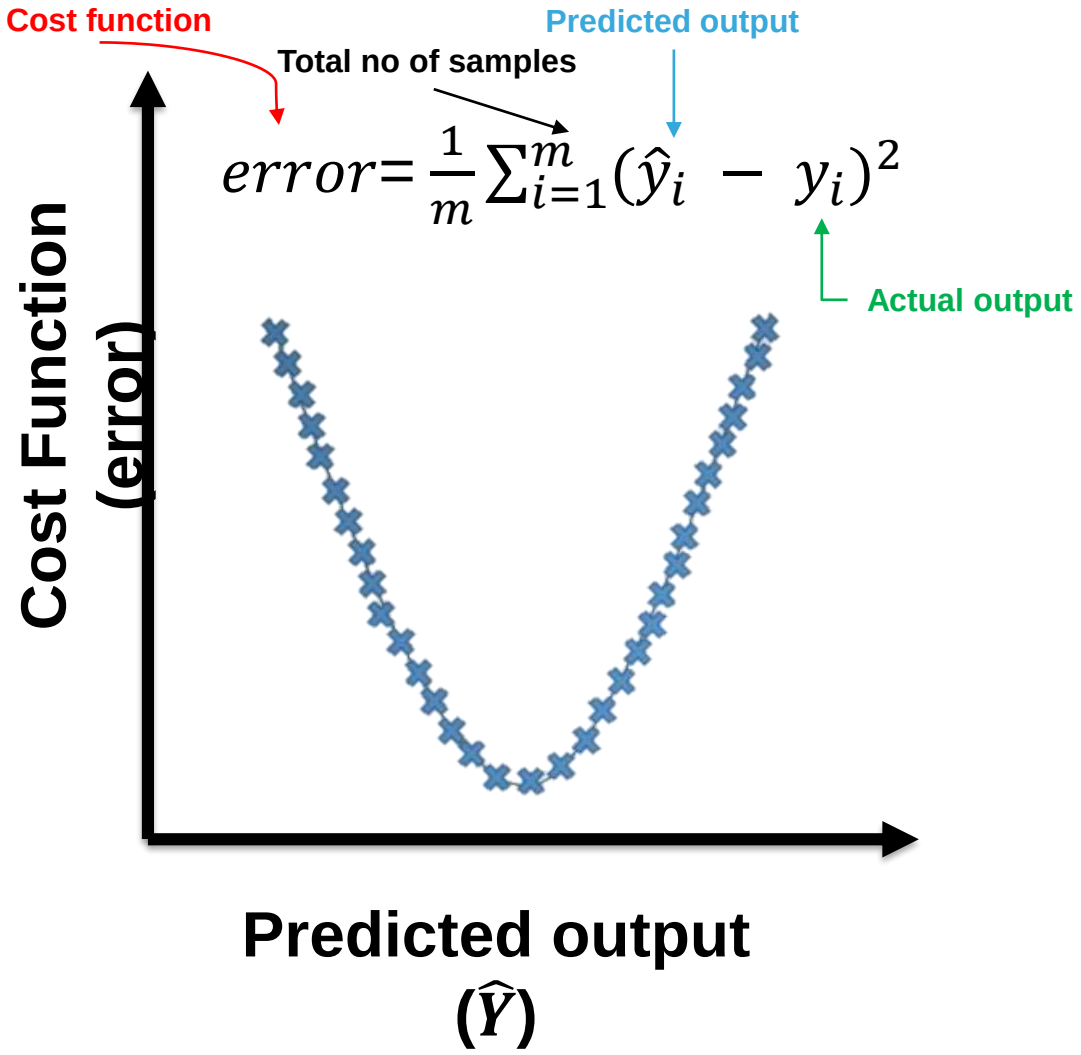
$$\frac{\partial}{\partial \boldsymbol{W}} (\boldsymbol{w} \cdot \mathrm{ReLU}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{c}) + c - y)^2$$

life.augmented

Backpropagation



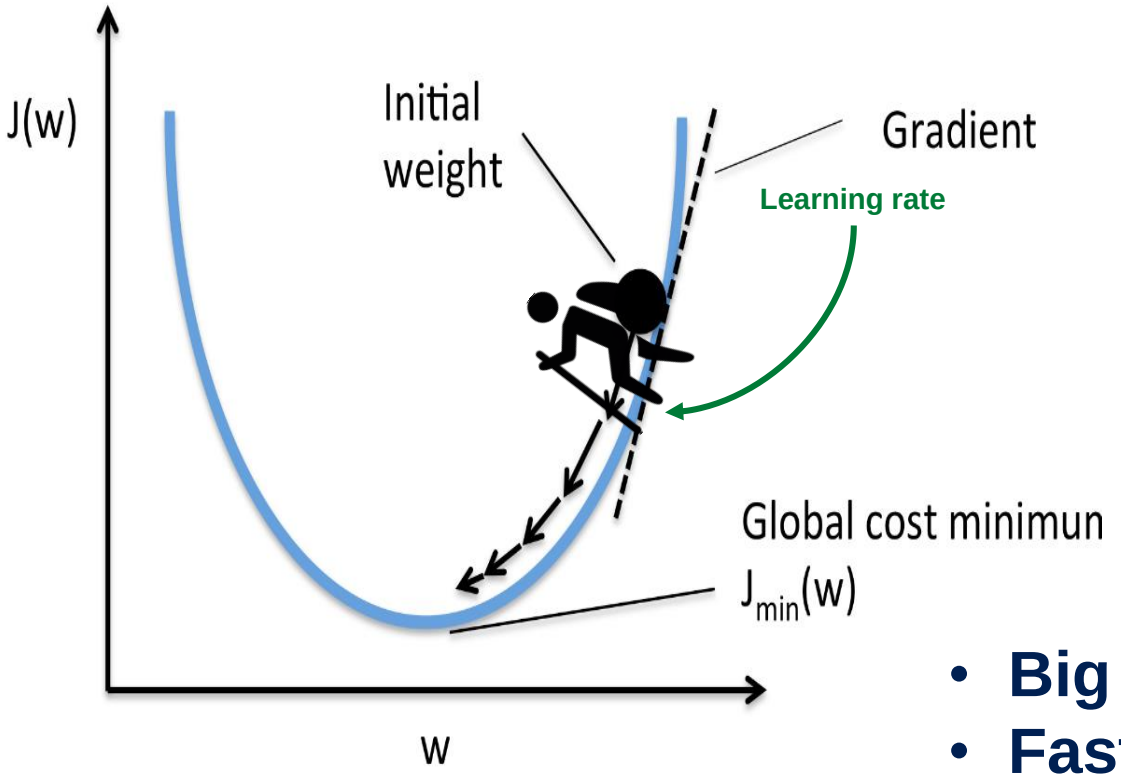
Backpropagation

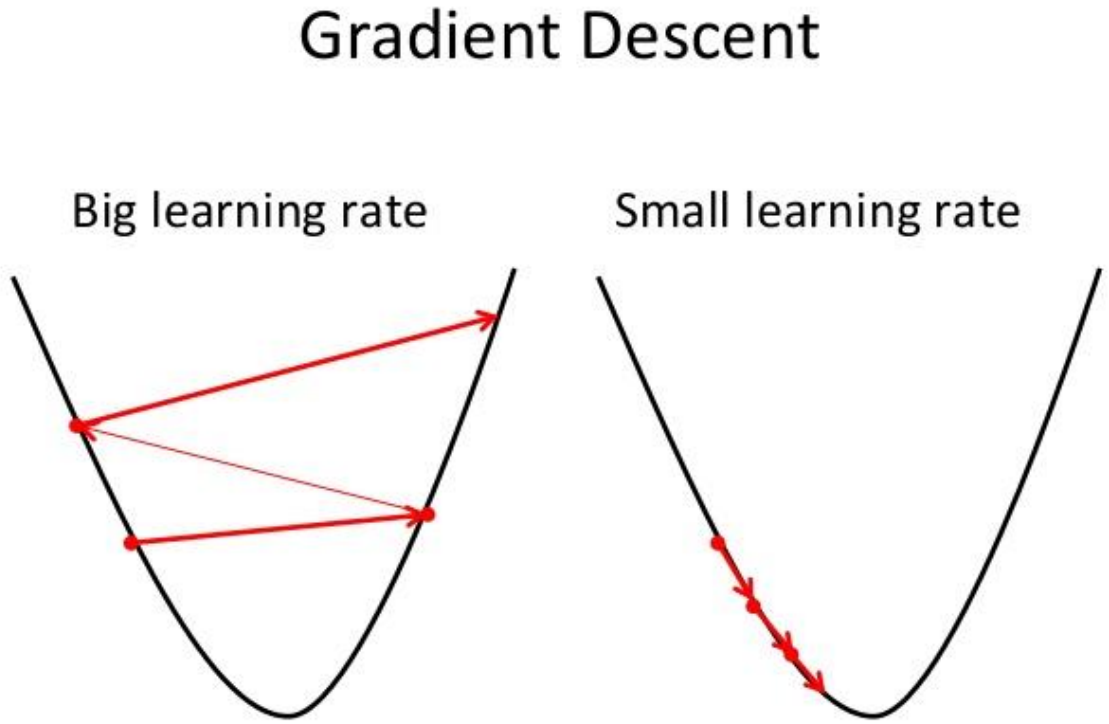


- Brute force
  - Try all the possible combination of weights.
  - Plot the cost function.
  - Use the weights which result in smallest error.
  - Sounds simple but will take too much time !!!

Backpropagation

- Enters the gradient descent





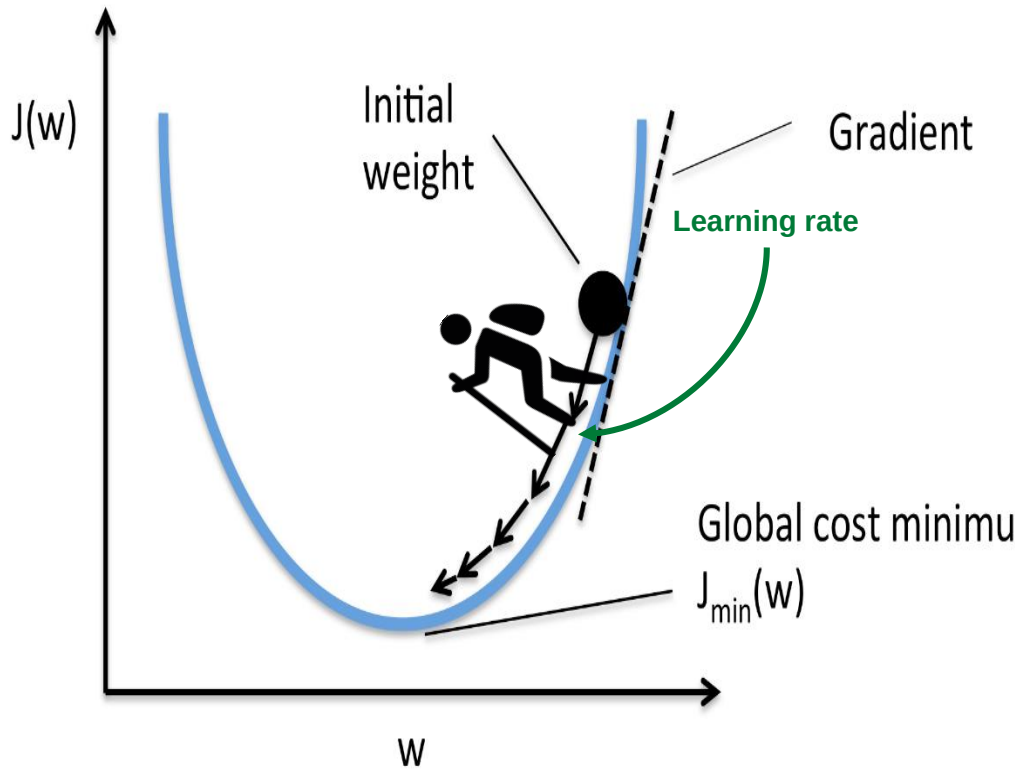
Learning rate compromise

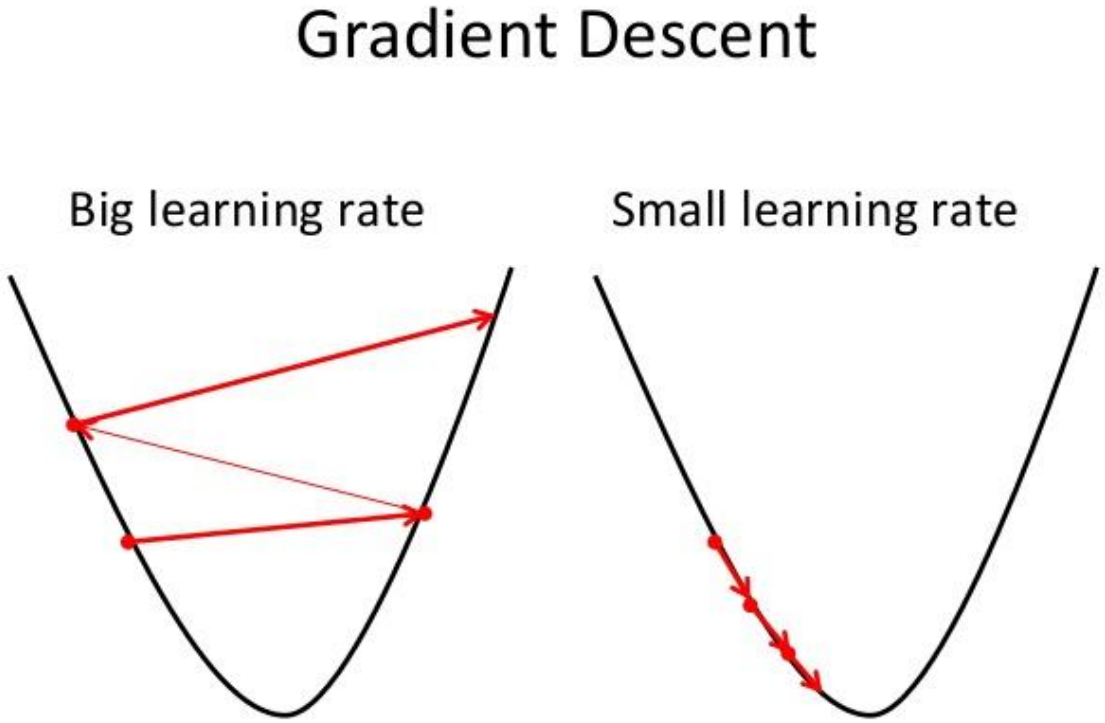
- **Big learning rate**
- **Fast**
- **May never converge.**
- **Small rate.**
- **Lots of small steps**
- **Will converge for sure**

life.augmented

Backpropagation

- Enters the gradient descent

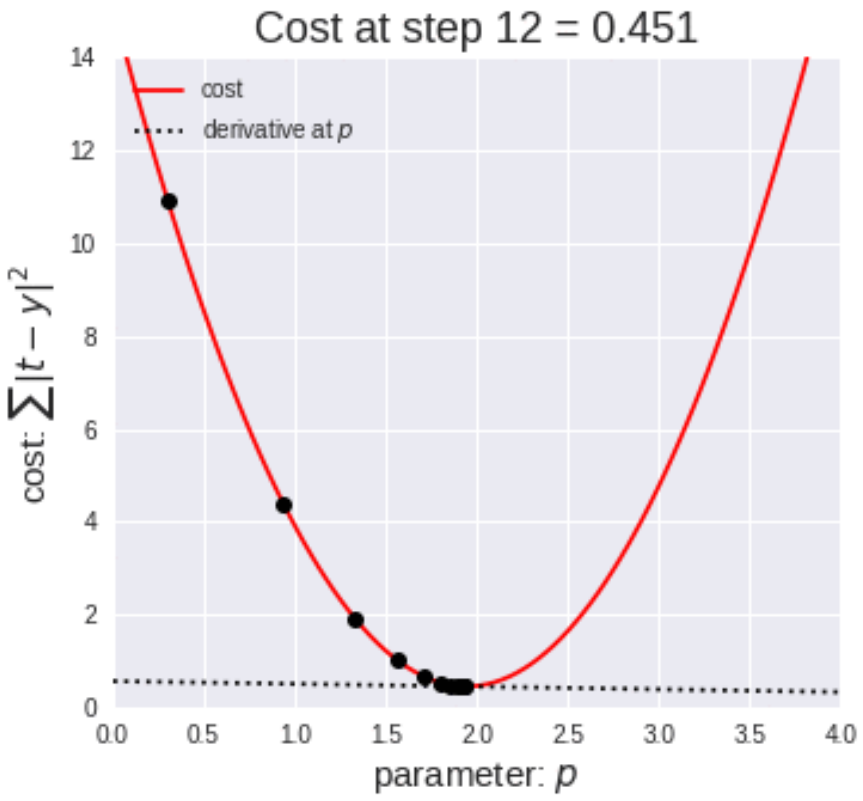


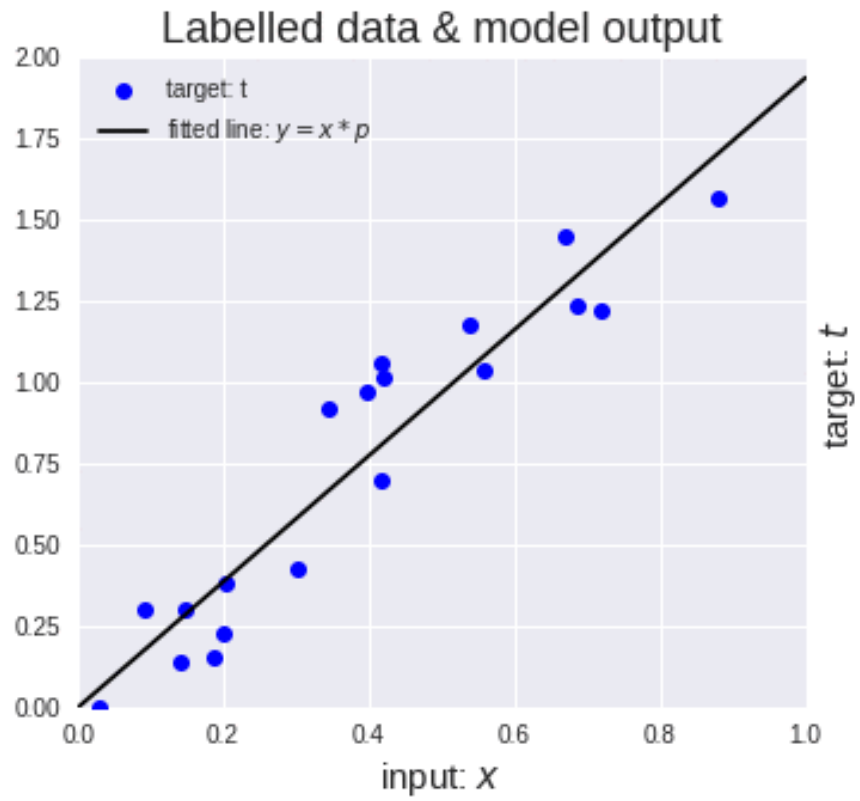


Use adaptive learning rate!

Gradient descent in action

- Example: Finding best linear fit to a set of points.





https://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html

Training Stats

pause

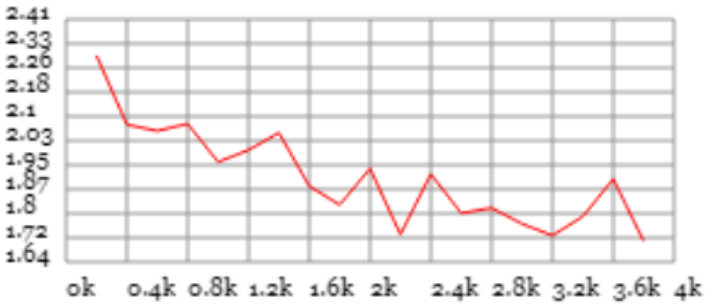
Forward time per example: 12ms
Backprop time per example: 17ms
Classification loss: 1.68352
L2 Weight decay loss: 0.00208
Training accuracy: 0.37
Validation accuracy: 0.27
Examples seen: 3967

Learning rate: 0.01 change
Momentum: 0.9 change
Batch size: 4 change
Weight decay: 0.0001 change

save network snapshot as JSON
init network from JSON snapshot

load a pretrained network (achieves ~80% accuracy)

Loss:



clear graph

Instantiate a Network and Trainer

```
layer_defs = [];
layer_defs.push({type:'input', out_sx:32, out_sy:32, out_depth:3});
layer_defs.push({type:'conv', sx:5, filters:16, stride:1, pad:2, activation:'relu'});
layer_defs.push({type:'pool', sx:2, stride:2});
layer_defs.push({type:'conv', sx:5, filters:20, stride:1, pad:2, activation:'relu'});
layer_defs.push({type:'pool', sx:2, stride:2});
layer_defs.push({type:'conv', sx:5, filters:20, stride:1, pad:2, activation:'relu'});
layer_defs.push({type:'pool', sx:2, stride:2});
layer_defs.push({type:'softmax', num_classes:10});
```