Deep learning:
an introduction

Deep learning in a nutshell

- Deep learning is a machine learning methodology that aims at solving (modeling) problems by building layer-wise models with several (many) levels of increasing abstraction

- Layers of these models capture discriminative/descriptive information from raw data

- Can be used for: supervised/unsupervised learning, reinforcement learning, feature extraction, …

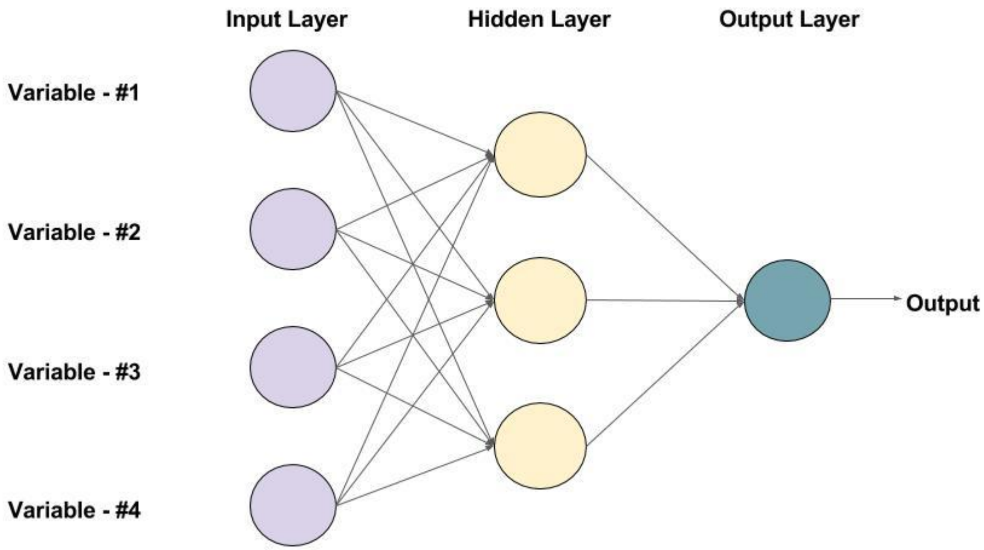
- Examples: multi-layer perceptrons, deep neural networks, convolutional neural networks, deep belief nets, auto encoders, etc.

Deep neural networks

- Deep feedforward networks are the ``essential" deep learning models
- Conventionally, a neural network is said to be deep if it has at least 2 hidden layers
- Hence, feedforward neural networks comprise the fundamentals of deep learning
- How much do you know about NNs?

Neural networks – recap.

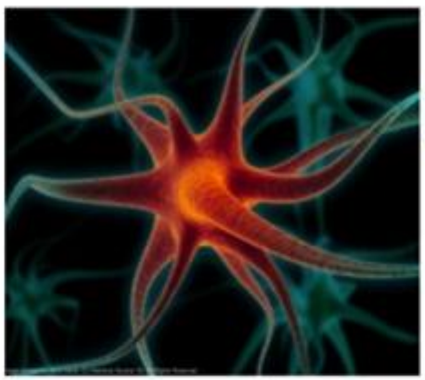
- A feedforward neural network is a model:
  - That approximates functions of the form $y = f(x; \Theta)$
  - Is formed by multiple (nonlinear) functions arranged in layers
  - Layers form a network
  - In which information flows in a single (forward) direction

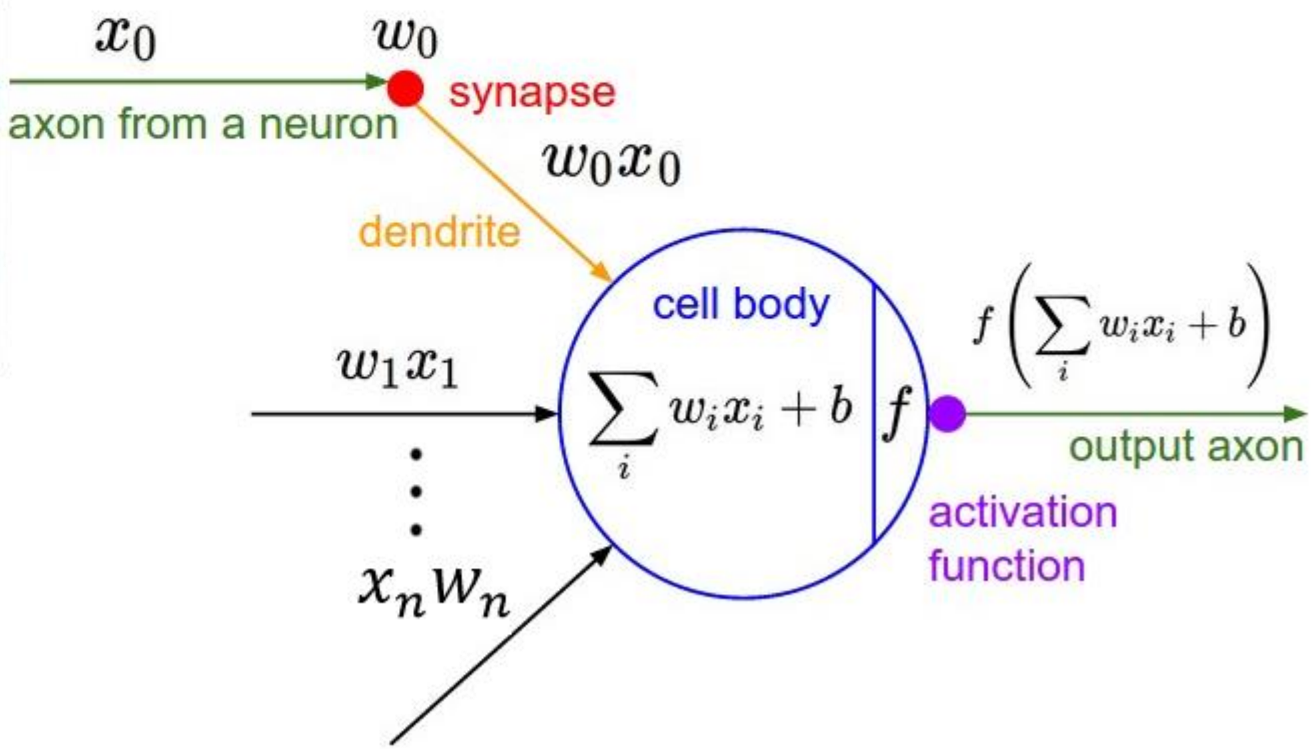


An example of a Feed-forward Neural Network with one hidden layer (with 3 neurons)

Neural networks – recap.

- Neuron analogy

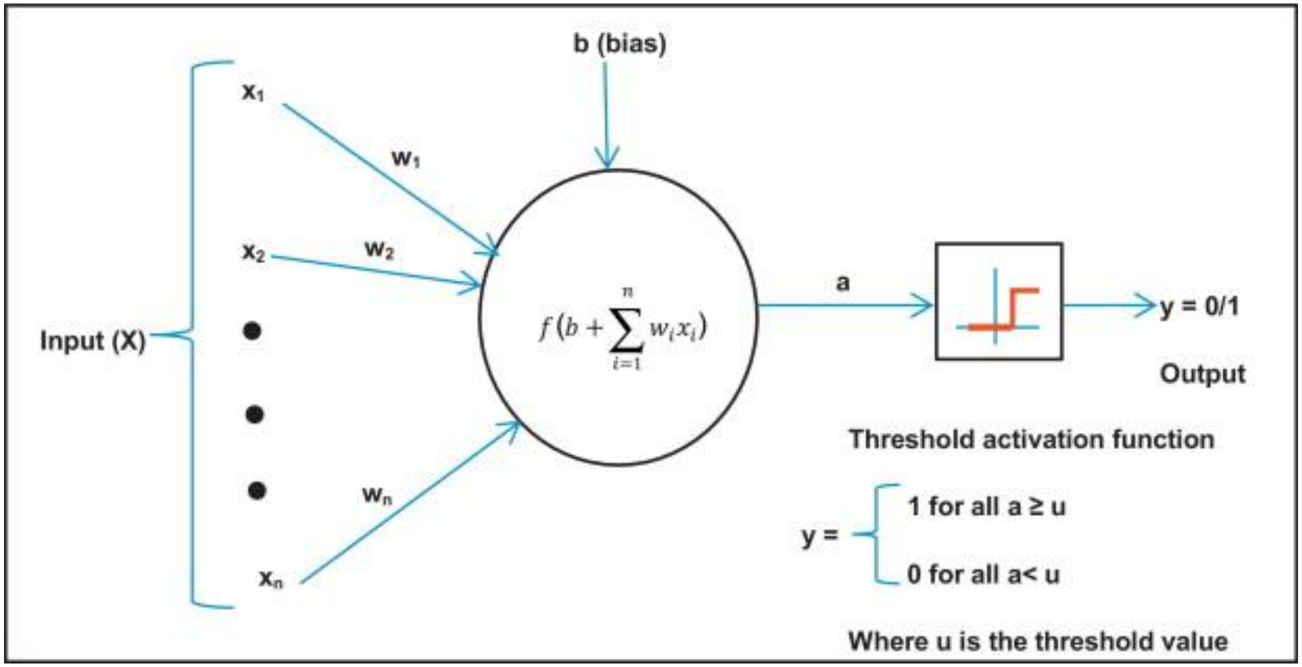


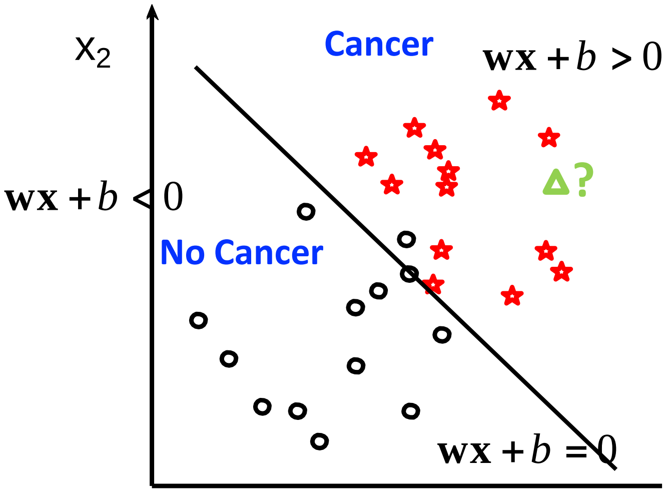


Neural networks –from perceptrons to DNNs)

- In general neural networks are built of units that resemble the perceptron (linear units activated by a differentiable function)

- **Perceptron -** A simple, linear classifier that can solve linearly separable classification (grandparent of NNs and the SVM) problems, learnable weights





Neural networks – recap. (from perceptrons to DNNs)

- How to determine the weights **w**?

- The Perceptron learning algorithm

  1. $\mathbf{w}$ ← randomly initialize weights
  2. Repeat until stop criterion meet
     I. For each $\mathbf{x}_i \in D$
        a) $o_i \leftarrow \mathbf{w}\mathbf{x}_i + b$ // estimate perceptron's prediction
        b) $\Delta\mathbf{w} \leftarrow \eta(y_i - o_i)$ // estimate the rate of change
        c) $\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w}$ //Update *w*
  3. Return **w**

- Convergence guaranteed (linearly separable problems)

What if the problem is non linearly separable?

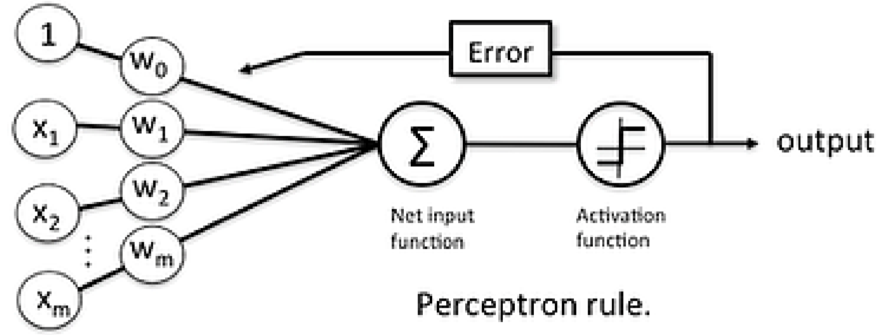
Neural networks – recap. (from perceptrons to DNNs)

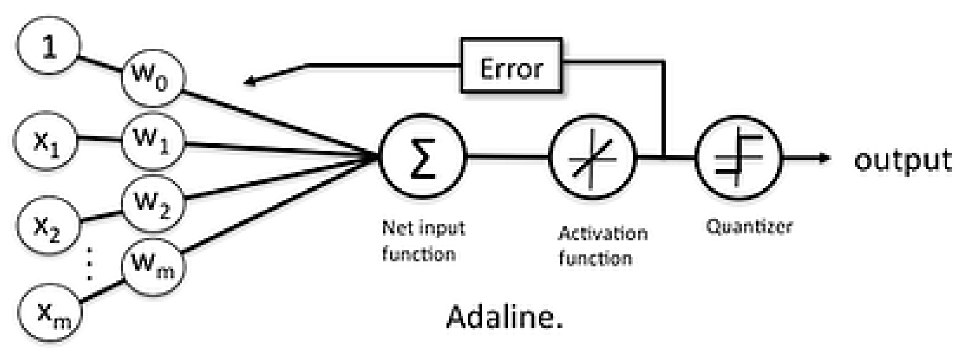
- For non-linearly separable problems, weights for a *similar* unit can be optimized with gradient descent

- Suppose we want to learn weights for a perceptron without threshold
  - $f(\mathbf{x}) = (\mathbf{wx} + b)$, with $\mathbf{w} \in \mathbb{R}^d$
  - $f(\mathbf{x}) = (\mathbf{wx})$, if we augment the input space with a 1, and include b into w

- And suppose we want to minimize:
  - $E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}(y_i - o_i)^2$

- How to learn these weights?



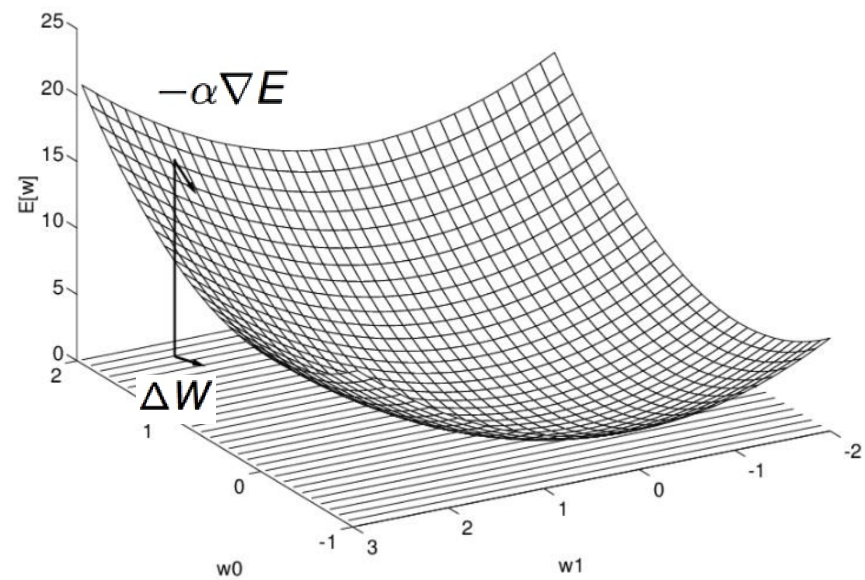


- perceptron은 0, +1의 출력을 가지는데 반해 Adaline은 -1 +1의 bipolar 출력을 가진다.
- adaptive signal processing

Neural networks – recap. (from perceptrons to DNNs)

- Problem:
  - Minimize $E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}(y_i - o_i)^2$ w.r.t **w**

- Idea: to explore the space of possible values that **w** can take. Starting with an initial w and updating it in the direction that decreases the error



Neural networks – recap. (from perceptrons to DNNs)

- The math:

$$W \leftarrow W + \Delta W$$

$$\Delta W = -\alpha \nabla E$$

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
&= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d) \\
&= \sum_{d \in D} (t_d - o_d)(-x_{i,d})
\end{aligned}$$

$$\Delta w_i = \alpha \sum_{d \in D} (t_d - o_d) x_{i,d}$$

Neural networks – recap. (from perceptrons to DNNs)

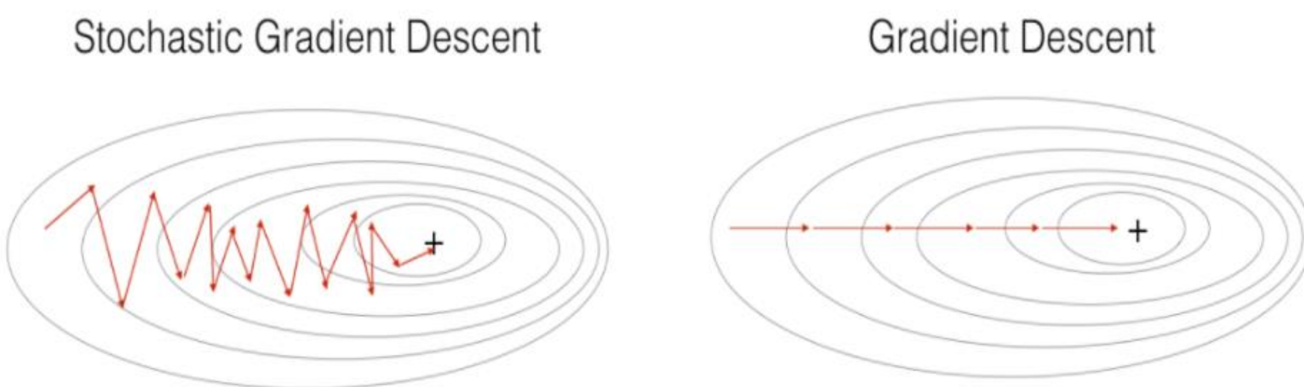
- The delta rule learning algorithm (gradient descend)

1. $\mathbf{w} \leftarrow$ randomly initialize weights
2. Repeat until stop criterion meet
   I. $\Delta\mathbf{w} \leftarrow$ initialize to 0
   II. For each $\mathbf{x}_i \in D$
      a) $o_i \leftarrow \mathbf{w}\mathbf{x}_i + b$ // estimate perceptron's prediction
      b) For each weight j estimate
         1. $\Delta\mathbf{w}_j \leftarrow \Delta\mathbf{w}_j + \eta(y_i - o_i)\, x_{\boldsymbol{i},j}$ // estimate the rate of change
   III. $\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w}$ //Update *w*
3. *Return* $\mathbf{w}$

Neural networks – recap. (from perceptrons to DNNs)

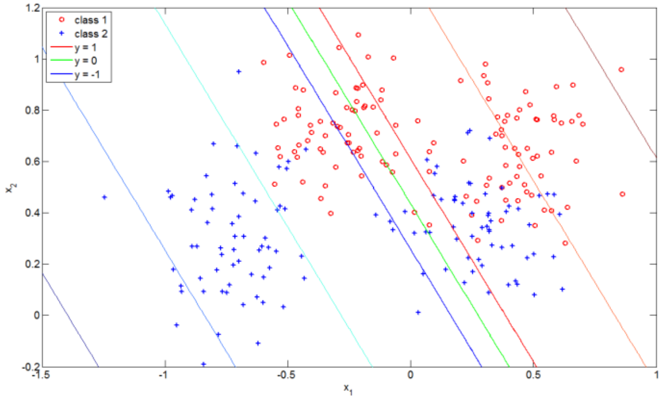
- SGD: in practice, a stochastic version of the algorithm is used, in which weights are updated after processing each input (*batch)

1. $\mathbf{w} \leftarrow$ randomly initialize weights
2. Repeat until stop criterion meet
   I. $\Delta\mathbf{w} \leftarrow$ initialize to 0
   II. For each $\mathbf{x}_i \in D$
      a) $o_i \leftarrow \mathbf{w}\mathbf{x}_i + b$ // estimate perceptron's prediction
      b) For each weight j estimate
         1. $\Delta\mathbf{w}_j \leftarrow \Delta\mathbf{w}_j + \eta(y_i - o_i)\, x_{i,j}$ // estimate the rate of change
      c) $\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w}$ //Update *w*
3. *Return* $\mathbf{w}$

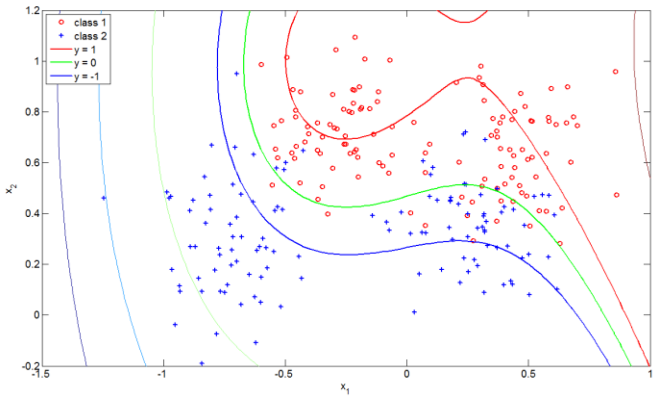


Neural networks – recap. (from perceptrons to DNNs)

- The problem with perceptrons et al.: They can only learn linear functions. When the data is not linearly separable the best one can do is to expect to have a *good fit*



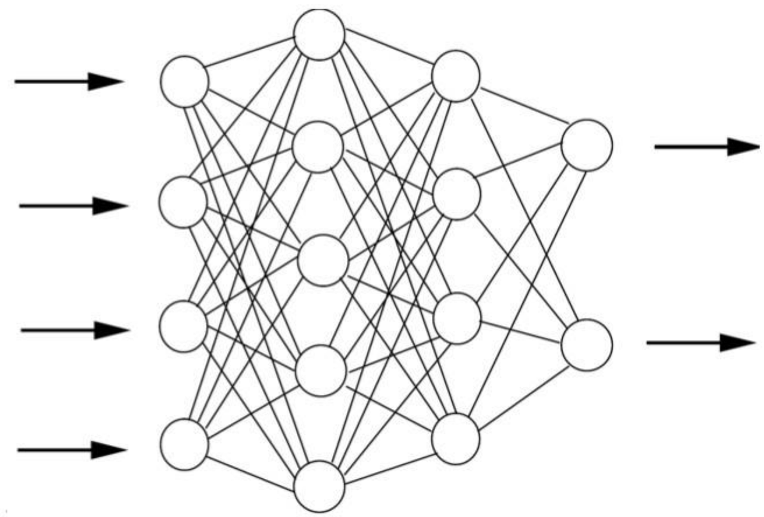
- Solutions?
  - To map the data into a non linear feature space in which the problem can become linearly separable
    - How?



Neural networks – recap. (from perceptrons to DNNs)

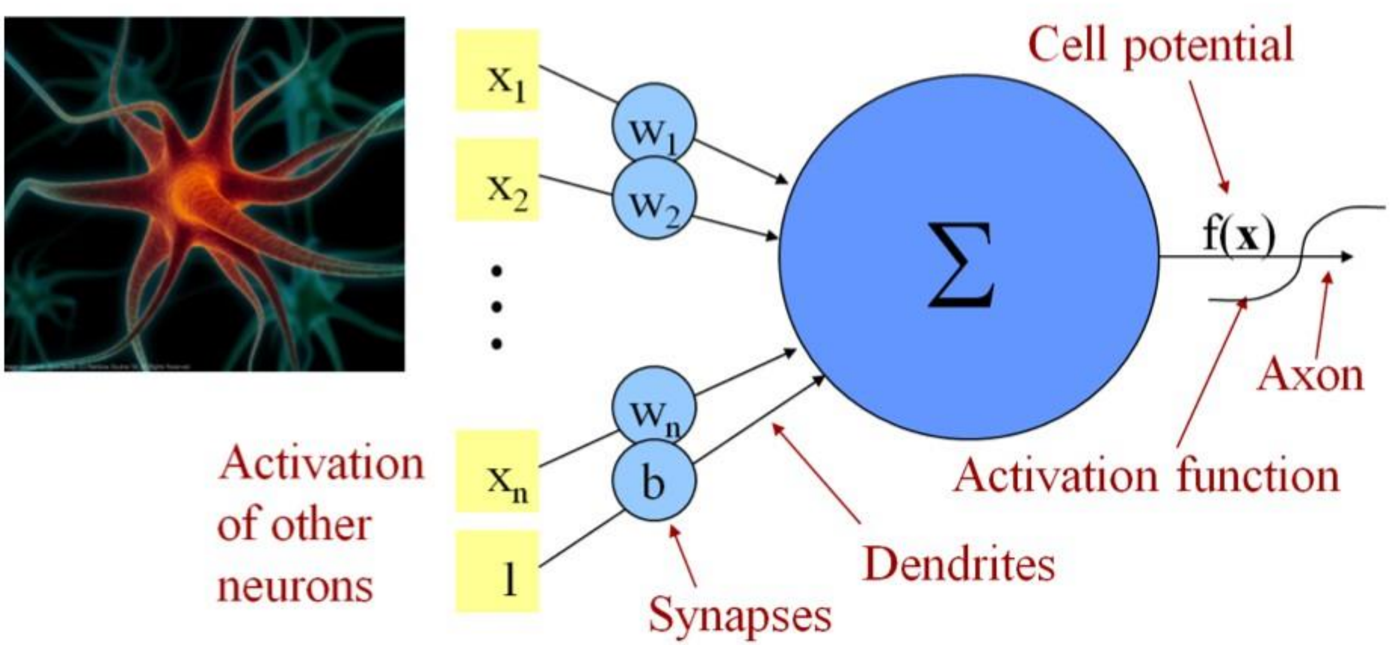
- What about stacking multiple layers of linear units?
  - Still will produce only linear functions

- Idea: stacking multiple layers of linear units *activated* with non linear functions



Neural networks – recap. (from perceptrons to DNNs)

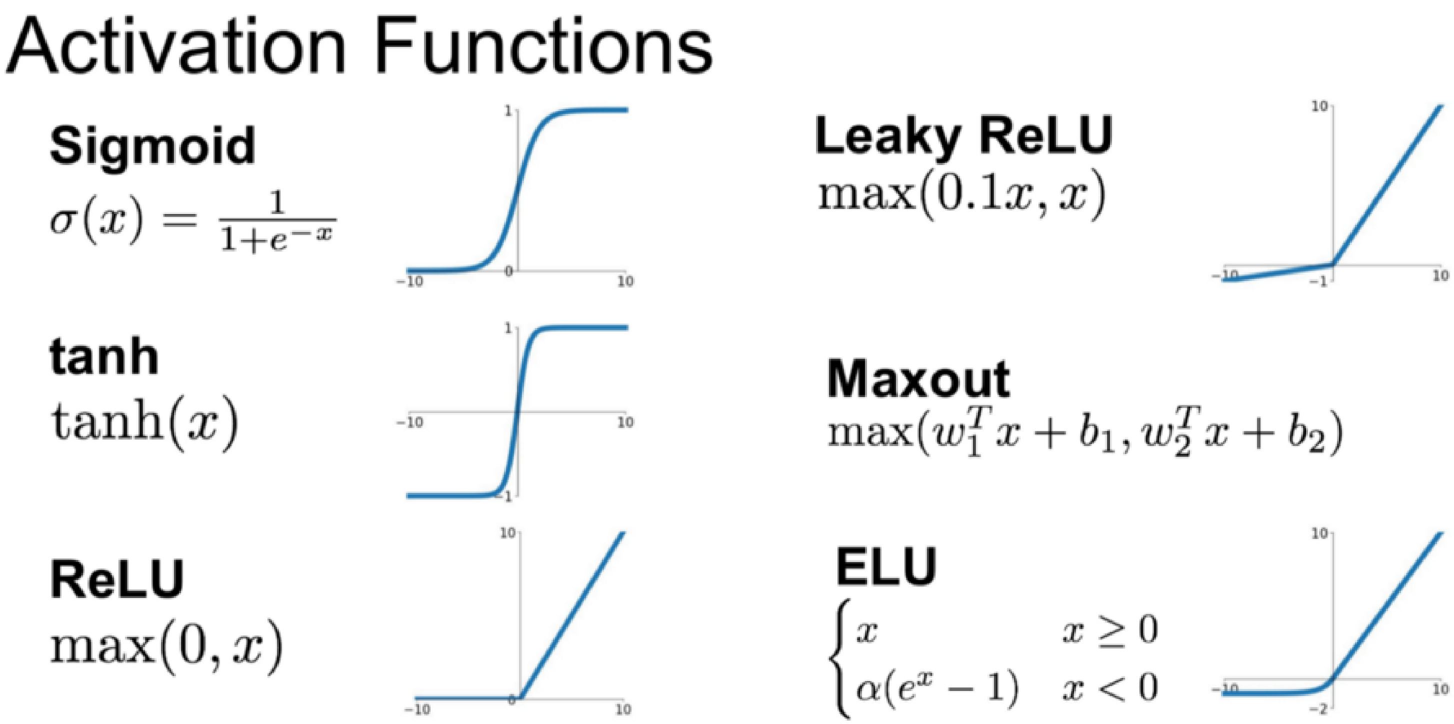
- Introducing non linearities in units



Slide from I. Guyon

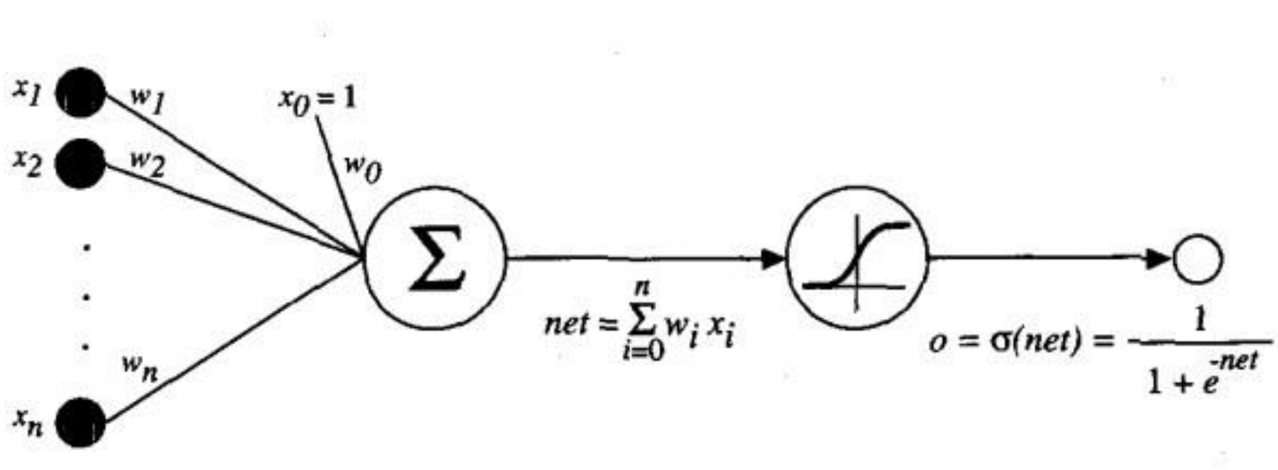
Neural networks – recap. (from perceptrons to DNNs)

- Introducing non linearities in units



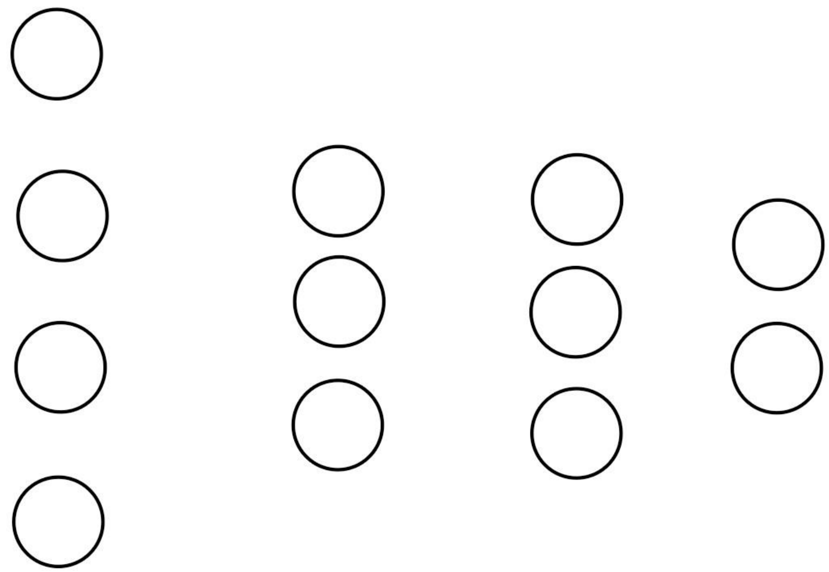
Neural networks – recap. (from perceptrons to DNNs)

- Introducing non linearities in units



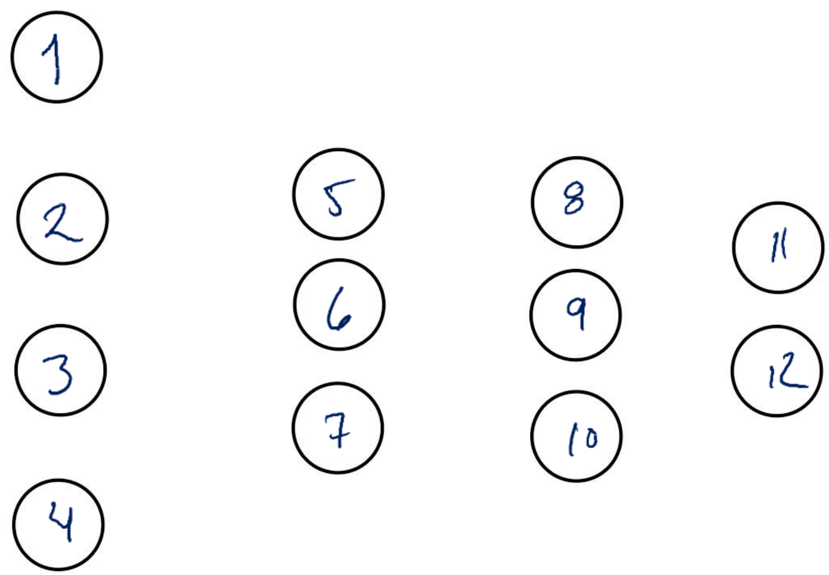
T. Mitchell. **Machine Learning**, McGrawHill 1997,

Neural networks – recap. (from perceptrons to DNNs)



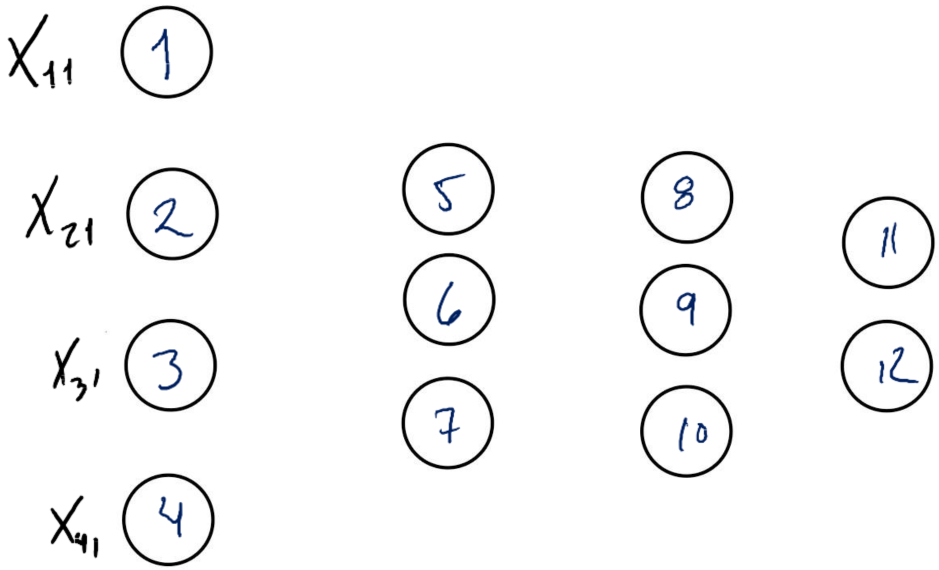
- Disentangling NNs

Neural networks – recap. (from perceptrons to DNNs)



- Disentangling NNs

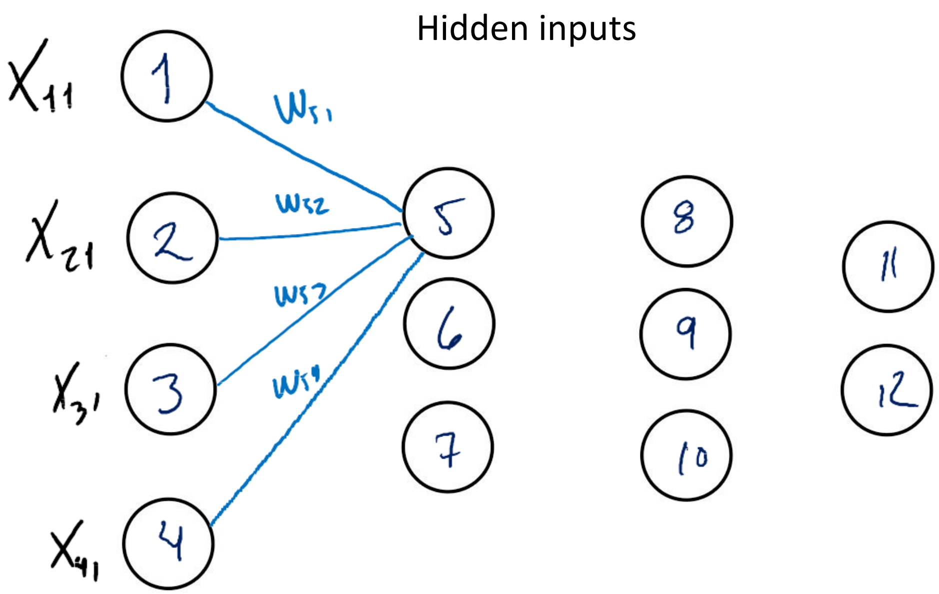
Neural networks – recap. (from perceptrons to DNNs)



Input units

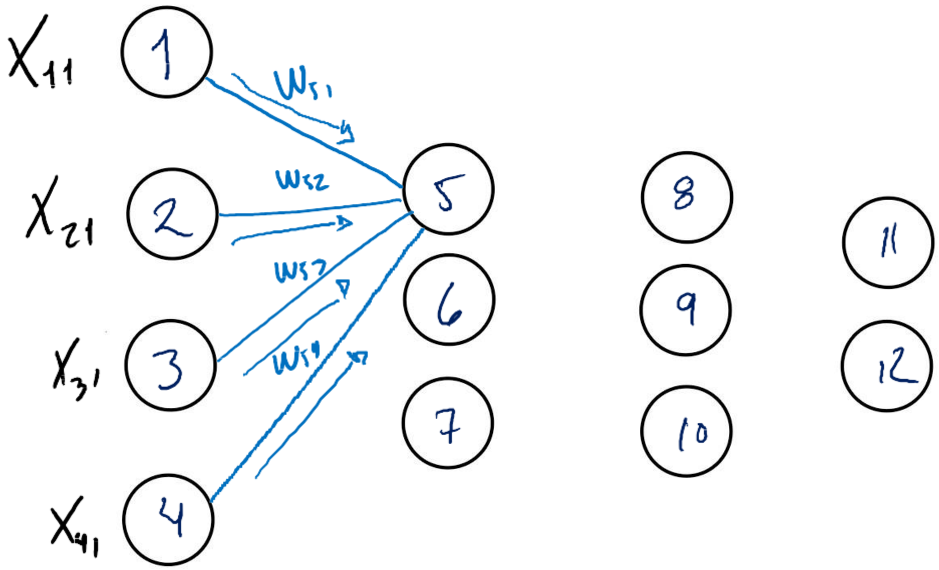
- Disentangling NNs

Neural networks – recap. (from perceptrons to DNNs)



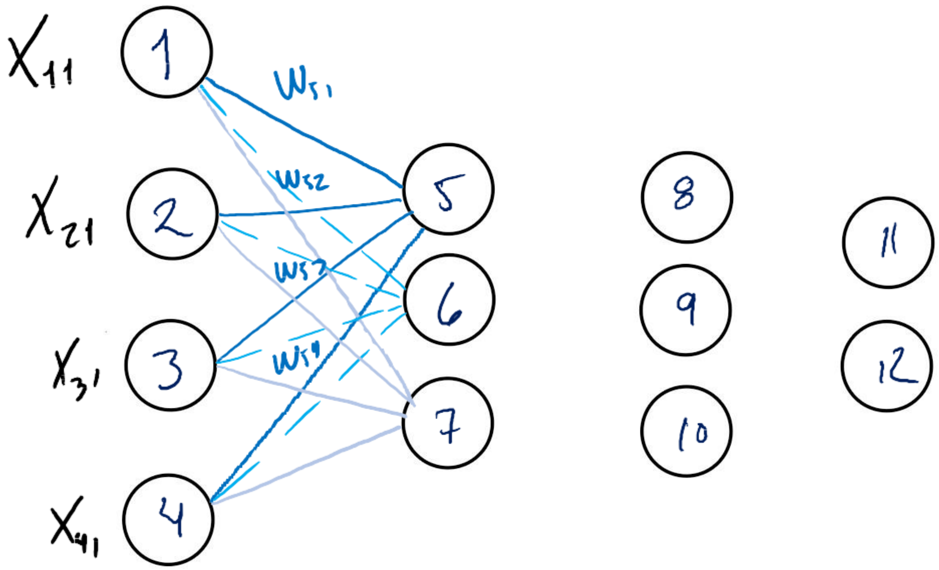
- Disentangling NNs

Neural networks – recap. (from perceptrons to DNNs)



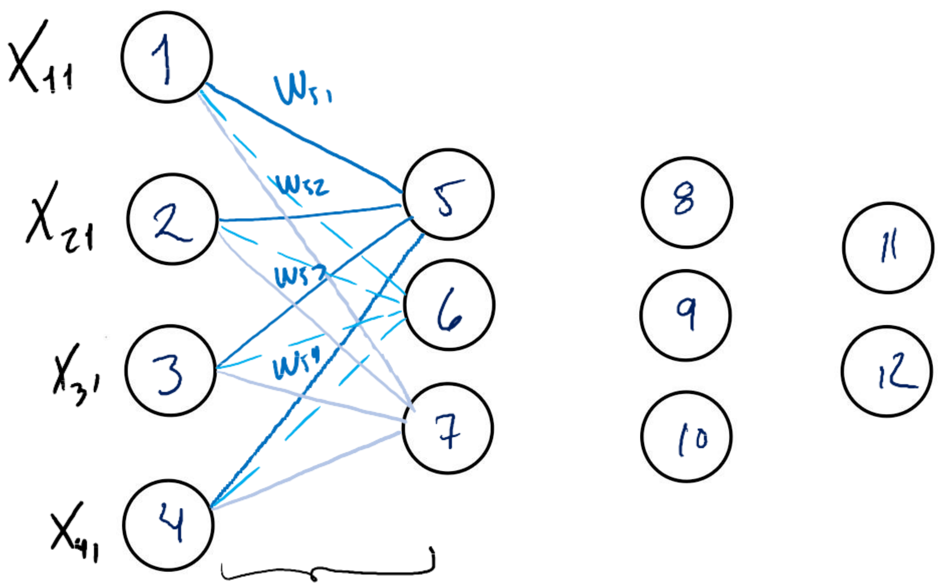
- Disentangling NNs

Neural networks – recap. (from perceptrons to DNNs)



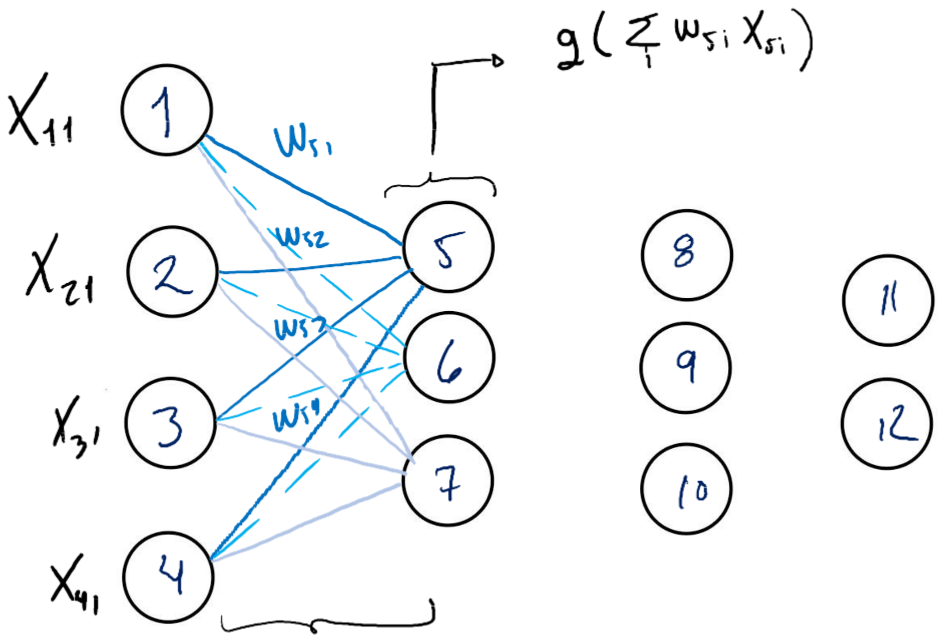
- Disentangling NNs

Neural networks – recap. (from perceptrons to DNNs)



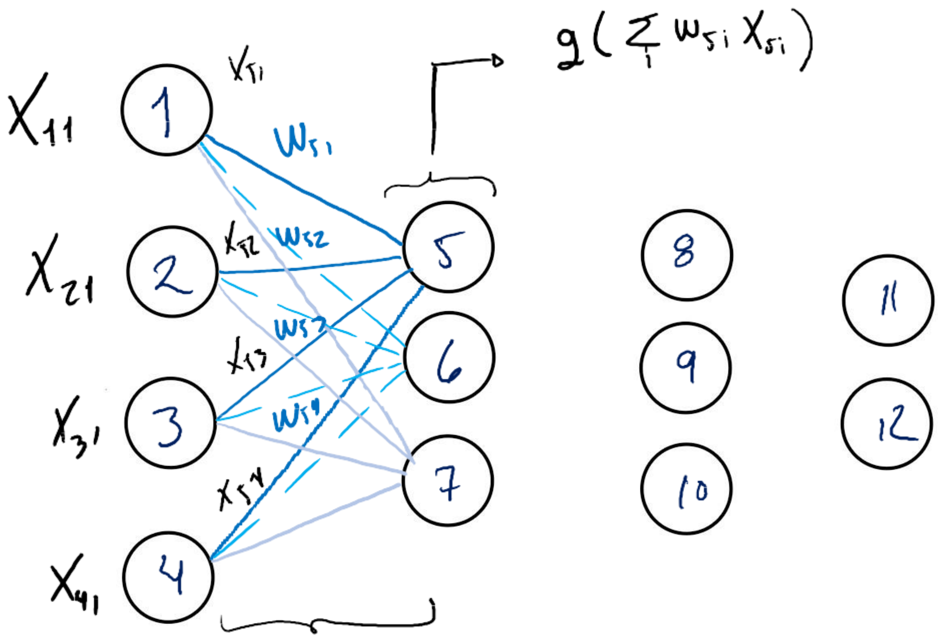
Weight matrix (4x3)

Neural networks – recap. (from perceptrons to DNNs)



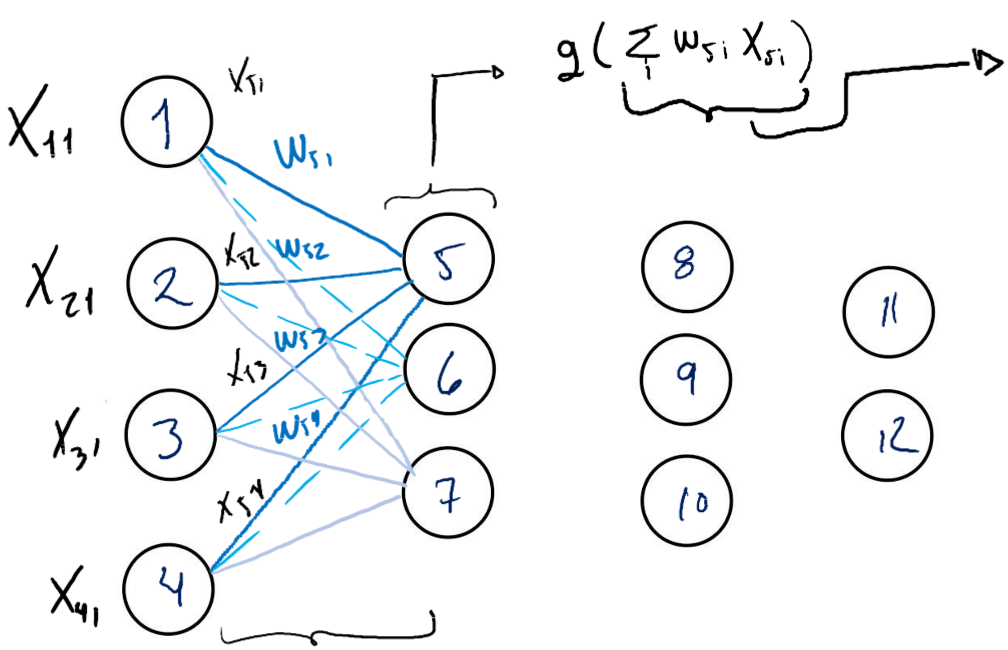
Weight matrix (4x3)

Neural networks – recap. (from perceptrons to DNNs)



Weight matrix (4x3)

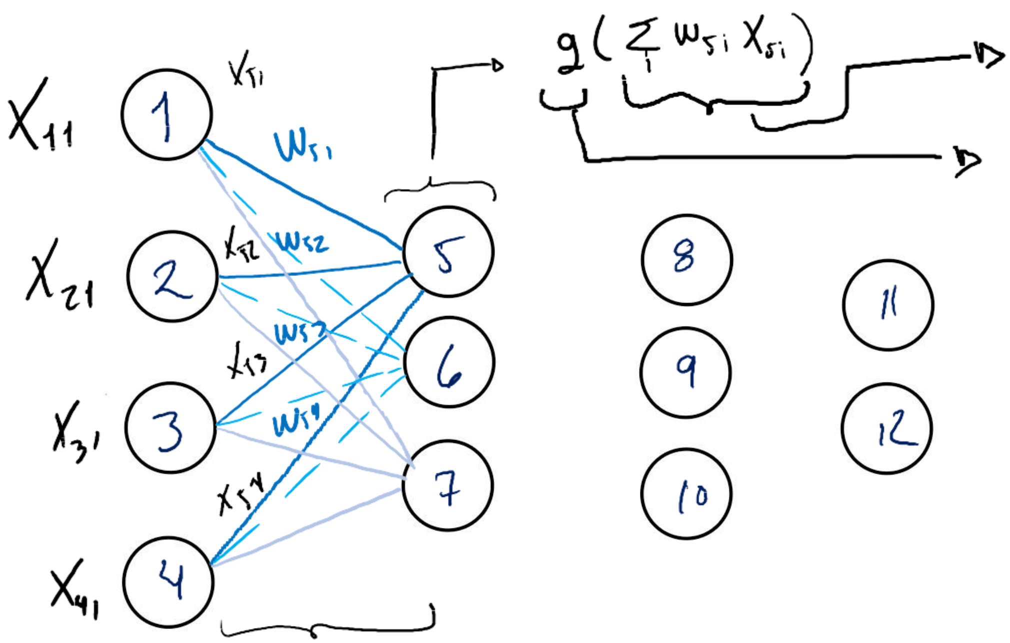
Neural networks – recap. (from perceptrons to DNNs)



Linear combination (perceptrón unit)

Weight matrix (4x3)

Neural networks – recap. (from perceptrons to DNNs)

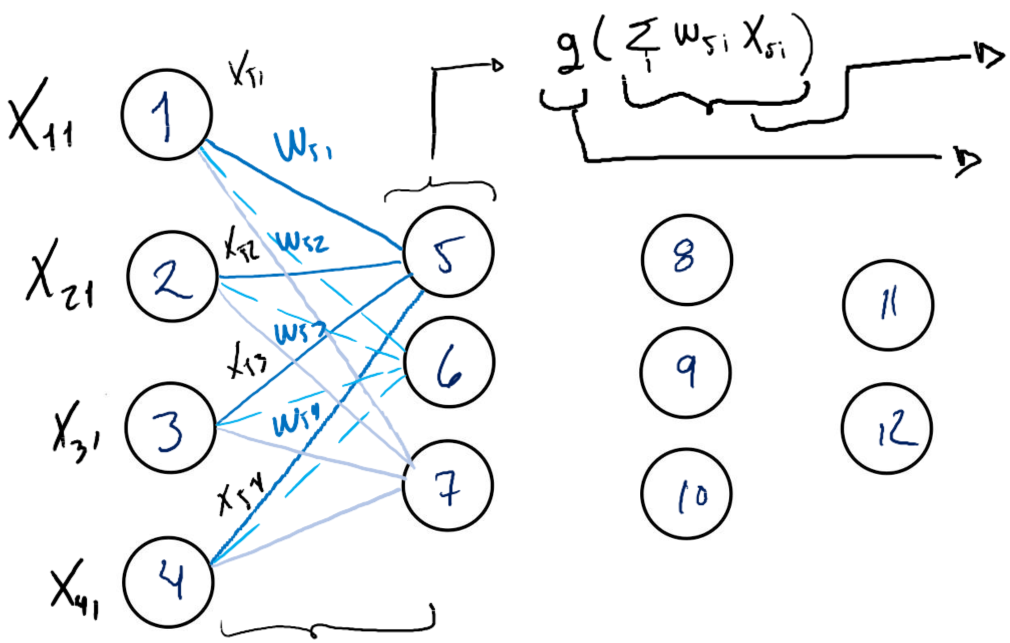


Linear combination (perceptrón unit)

Non linear activation

Weight matrix (4x3)

Neural networks – recap. (from perceptrons to DNNs)

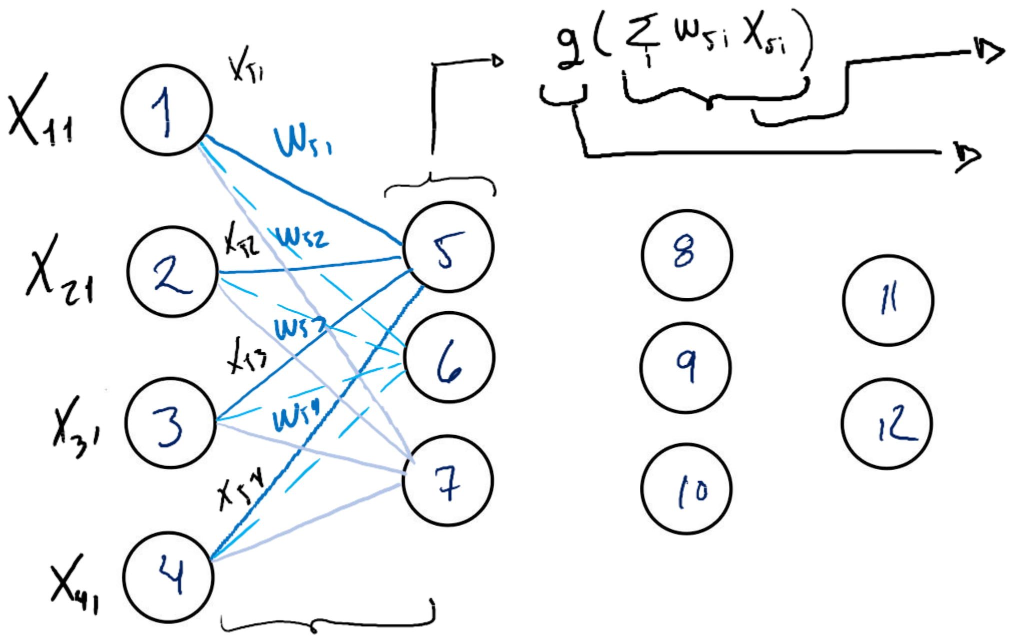


Linear combination (perceptrón unit)

Non linear activation

Weight matrix (4x3)

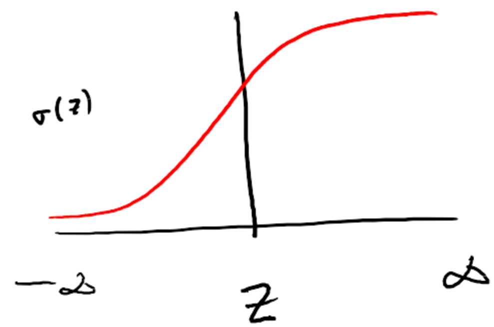
Neural networks – recap. (from perceptrons to DNNs)



Linear combination
(perceptrón unit)

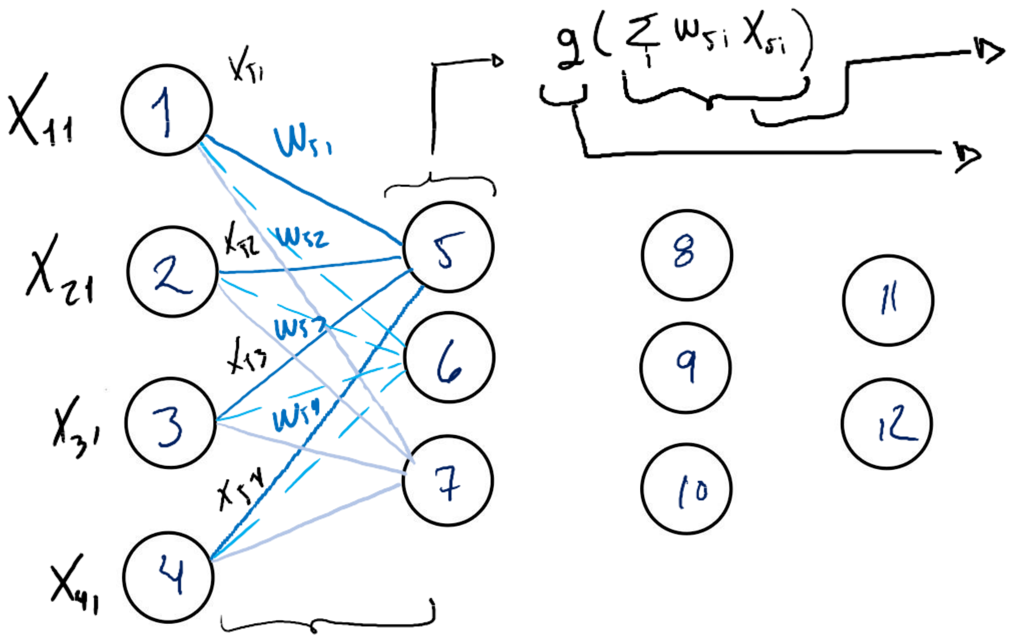
Non linear
activation

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



Weight matrix (4x3)

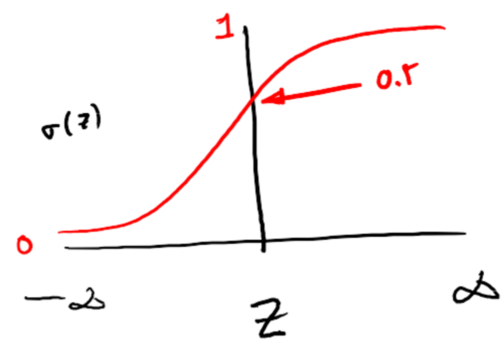
Neural networks – recap. (from perceptrons to DNNs)



Linear combination (perceptrón unit)

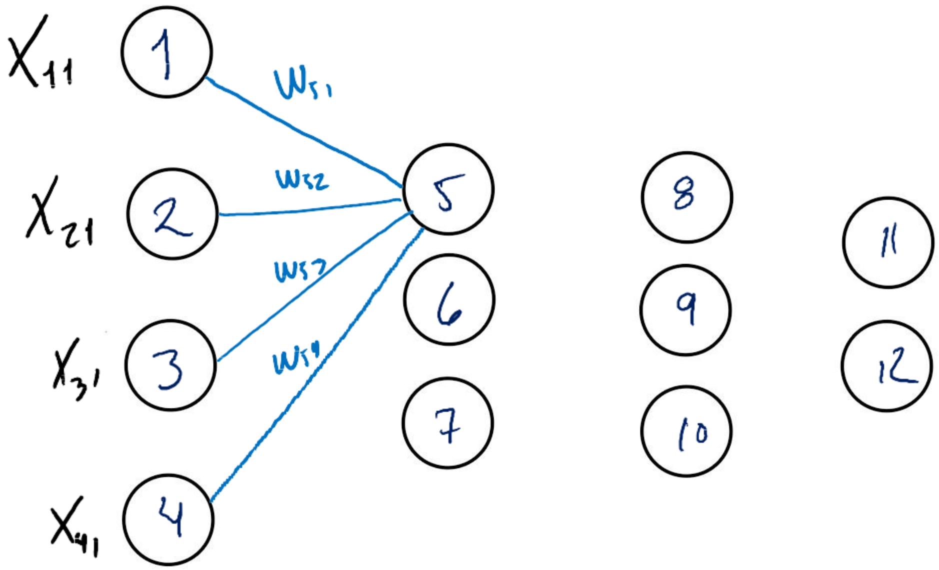
Non linear activation

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

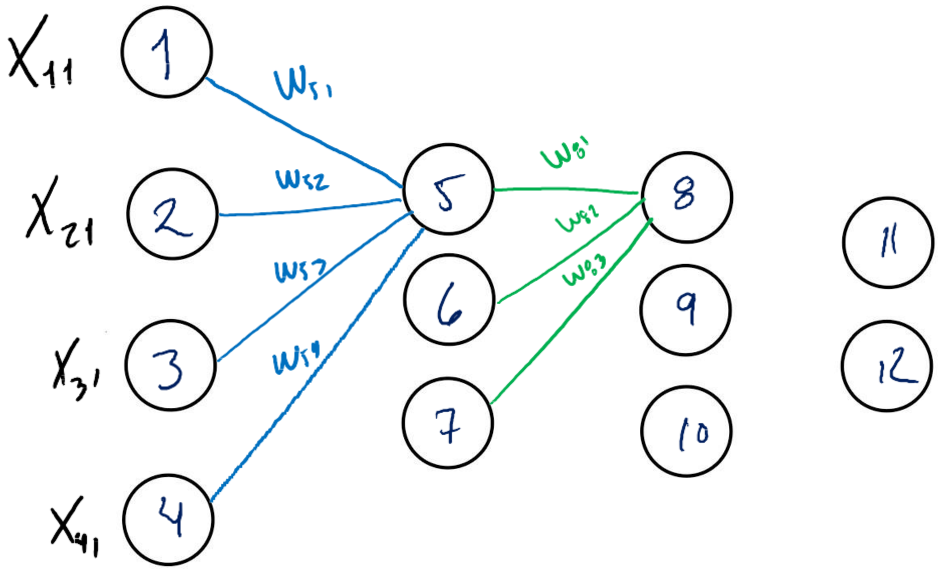


Weight matrix (4x3)

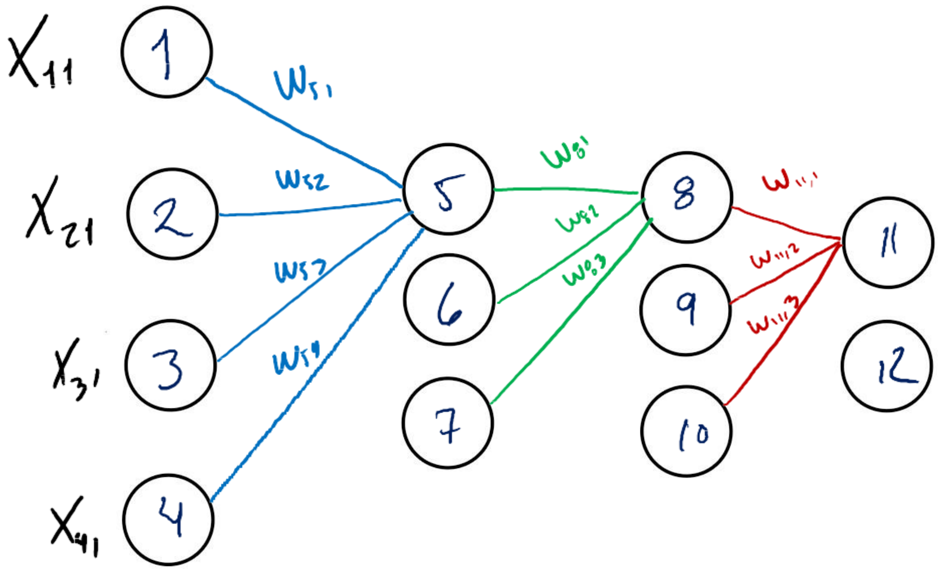
Neural networks – recap. (from perceptrons to DNNs)



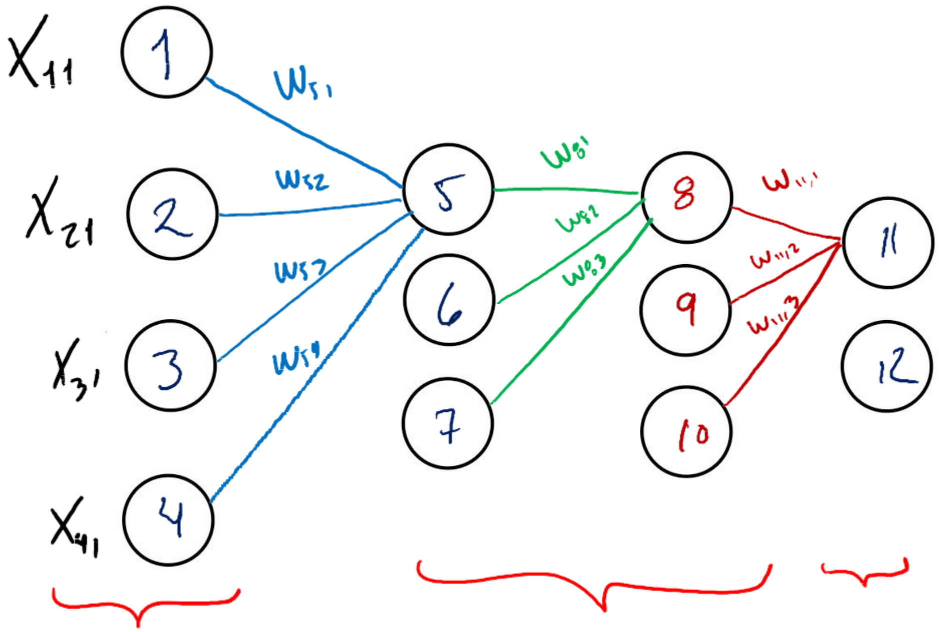
Neural networks – recap. (from perceptrons to DNNs)



Neural networks – recap. (from perceptrons to DNNs)



Neural networks – recap. (from perceptrons to DNNs)



Input units

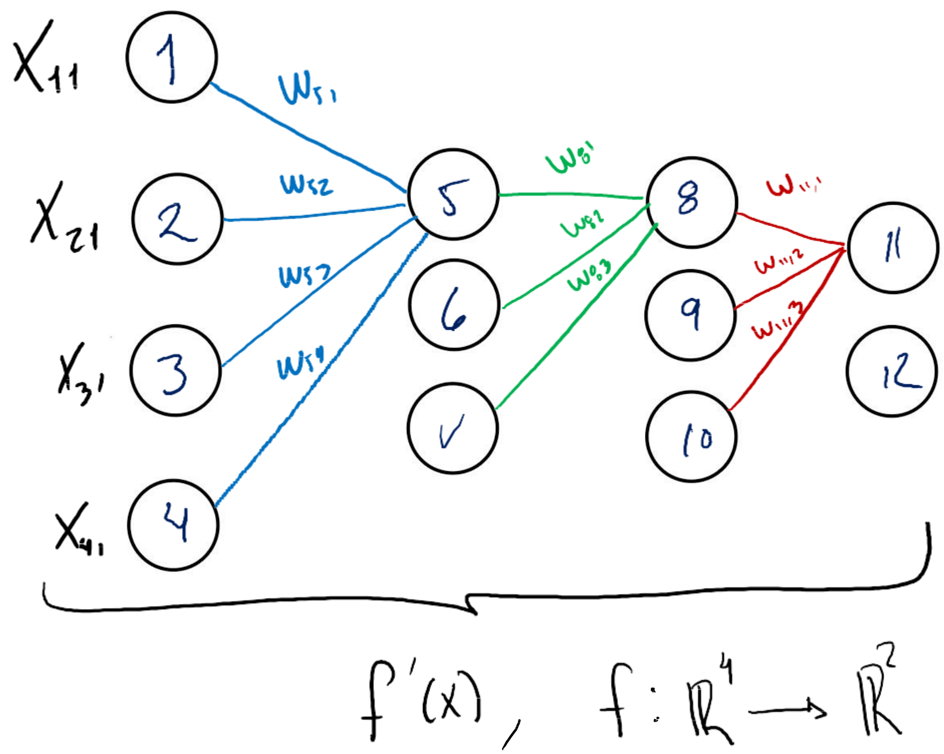
hidden units

output units

Neural networks – recap. (from perceptrons to DNNs)

- As other learning algorithms, NNs aim to learn a function mapping inputs to outputs
- Training a NN reduces to learning the weights in the network that minimize an error estimate
  - How many parameters?
  - How to adjust/determine their values?
  - What criterion to adopt?

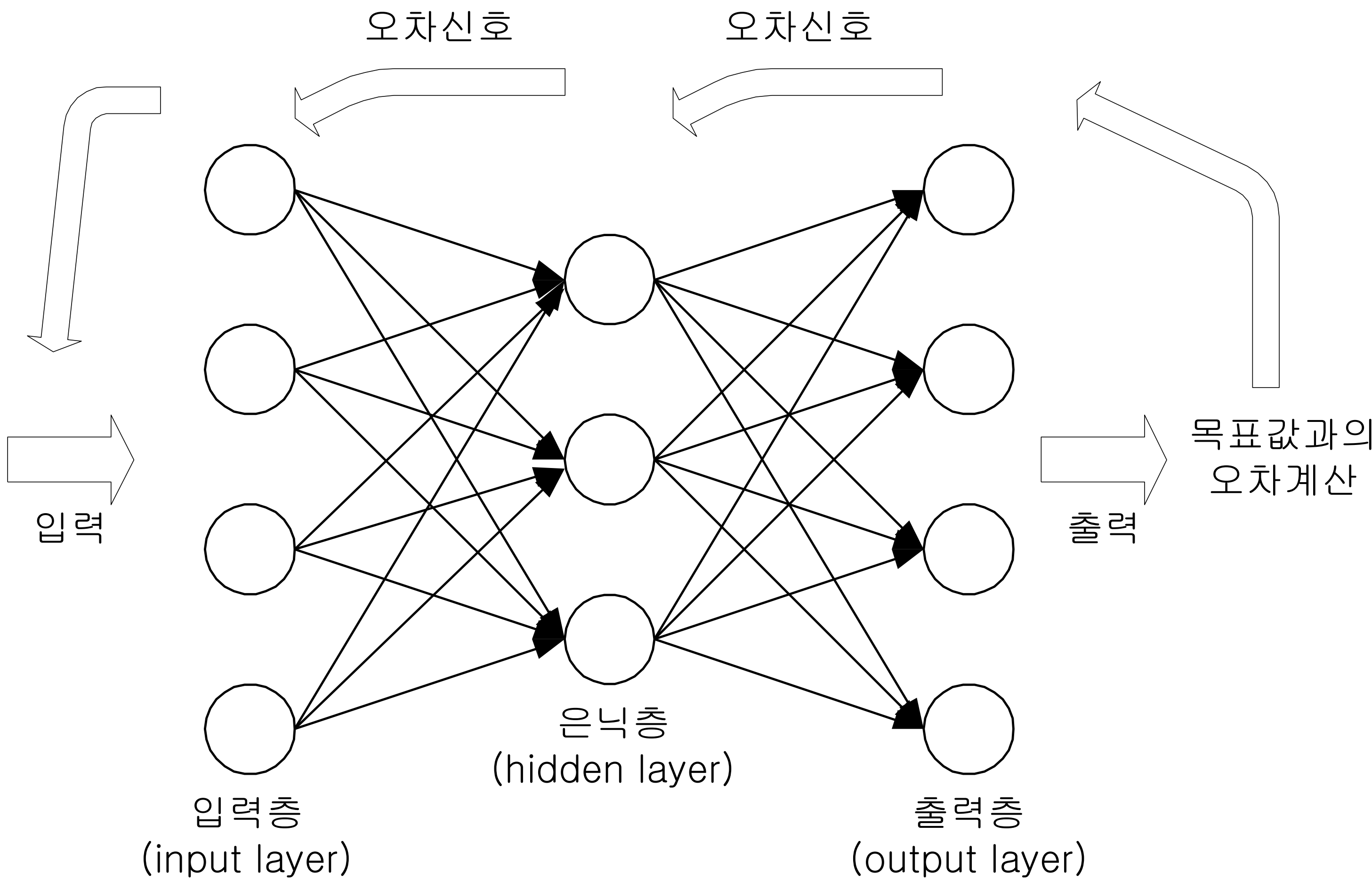
Neural networks – recap. (from perceptrons to DNNs)



Neural networks – **Backpropagation(BP) 신경망**

- 지도학습(supervised learning) 패턴의 대표적인 신경망 모델
- 오차 역전(error back propagation)기법을 processing unit에 적용한 것
- 각 processing unit 간의 연결강도(weights)를 수정함으로써 다음 학습 시 목표 값에 더욱 접근된 출력값을 갖게 한다.
- 출력값이 목표값과 유사하게 될 때까지 학습을 반복하게 되며, 학습이 끝나면 학습한대로 출력을 하게 된다.

Backpropagation 신경망 구조



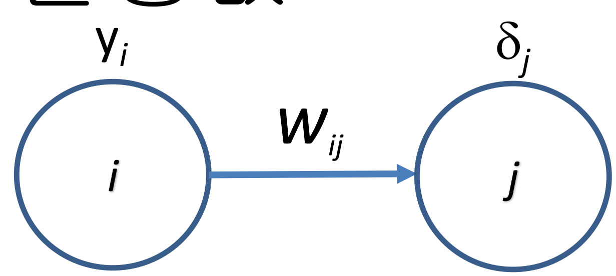
- **BP 신경망의 3 단계 학습**

1. 제 1 단계(전방향 단계) : 입력층에서 입력을 받아 출력층으로 출력하는 과정
   활성값(activations)이 출력됨
2. 제 2 단계 : 오차를 구하는 단계
   오차 - 목표값과 활성값과의 차
   목표값 - 출력층의 각 신경세포들에게 학습을 위해 미리 설정한 값
3. 제 3 단계 : 오차를 이용하여 오차신호를 계산하여 은닉층과 입력층에 역방향으로 되돌리면서 신경세포들 간의 연결강도들(weights)을 조율

신경세포 *i*와 *j* 간 연결강도(w_{ij}) 변화

- 목표값과 활성값과의 차인 오차를 구하여 오차신호를 계산
- 은닉층과 입력층에 역방향으로 되돌리면서 신경세포 *i*와 *j* 간의 연결강도(w_{ij}) 변화
- η: 학습률(learn rate, 0.1로 가정)
- δ_j: 신경세포 *j*의 오차신호(error signal)
- y_i: 신경세포 *i*의 Sigmoid 활성값

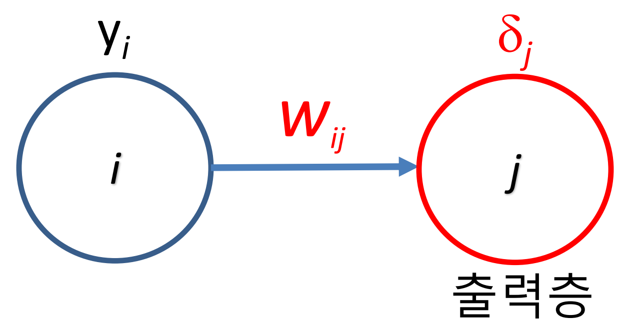
$$\Delta w_{ij} = \eta \delta_j y_i$$



* 출력층의 오차신호 δ_j

◈ 출력층의 오차 값에 y_j의 미분값을 곱

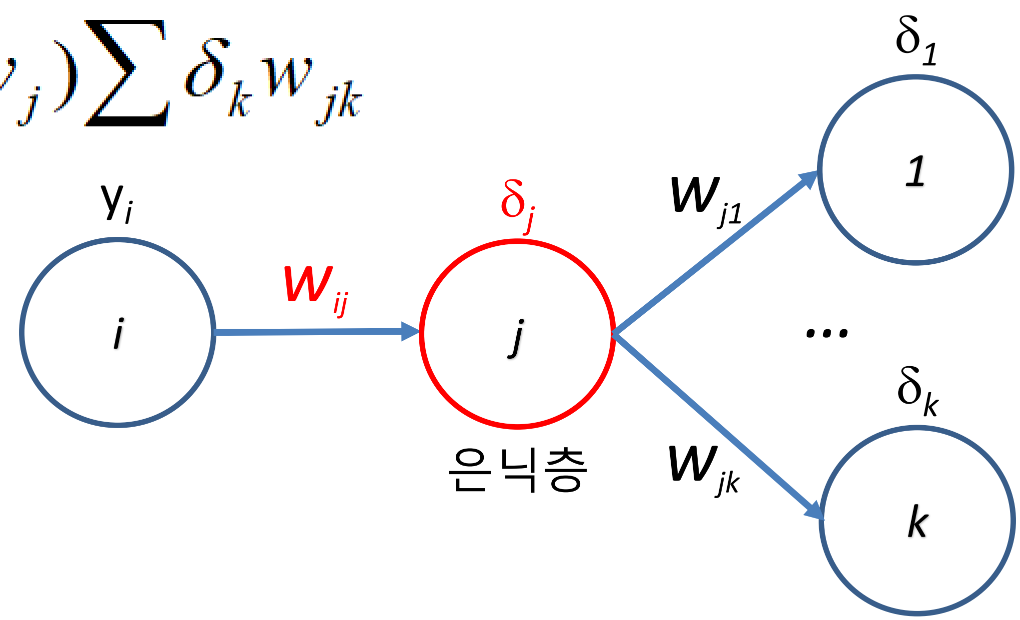
$$\delta_j = (t_j - y_j)y'_j = (t_j - y_j)y_j(1 - y_j)$$



* 은닉층 y_j에 대한 오차신호 (δ_j)

◈ 은닉층의 오차 값에 y_j의 미분 값을 곱

$$\delta_j = y'_j \sum \delta_k w_{jk} = y_j(1 - y_j) \sum \delta_k w_{jk}$$



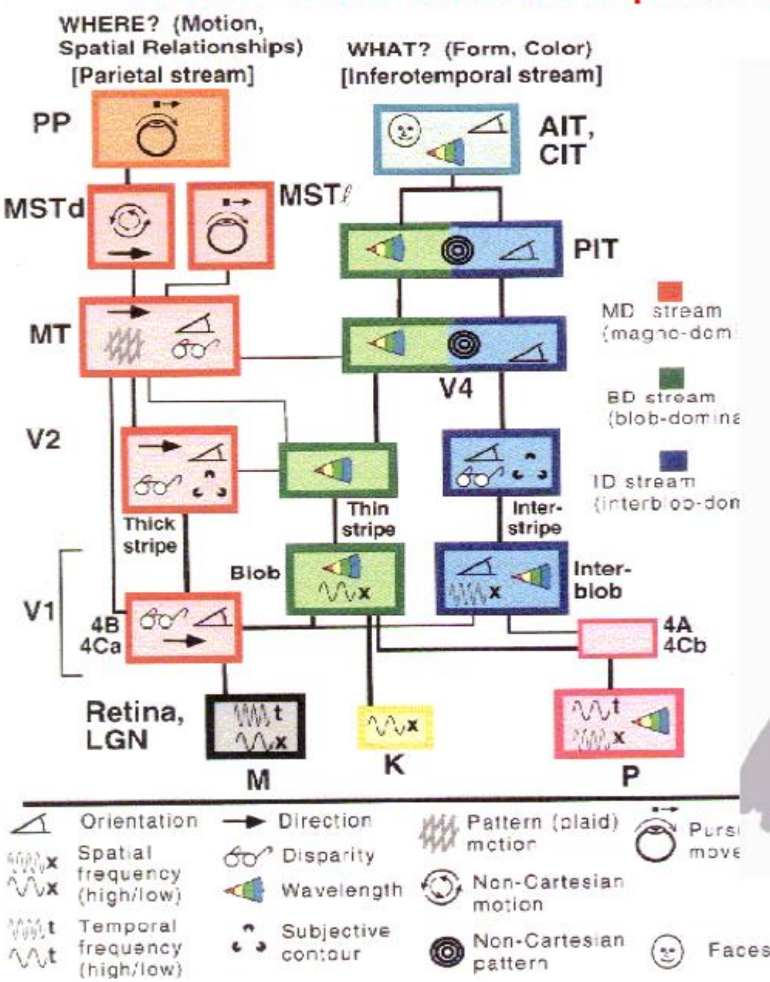
* n+1번째 학습 시 연결강도 변화

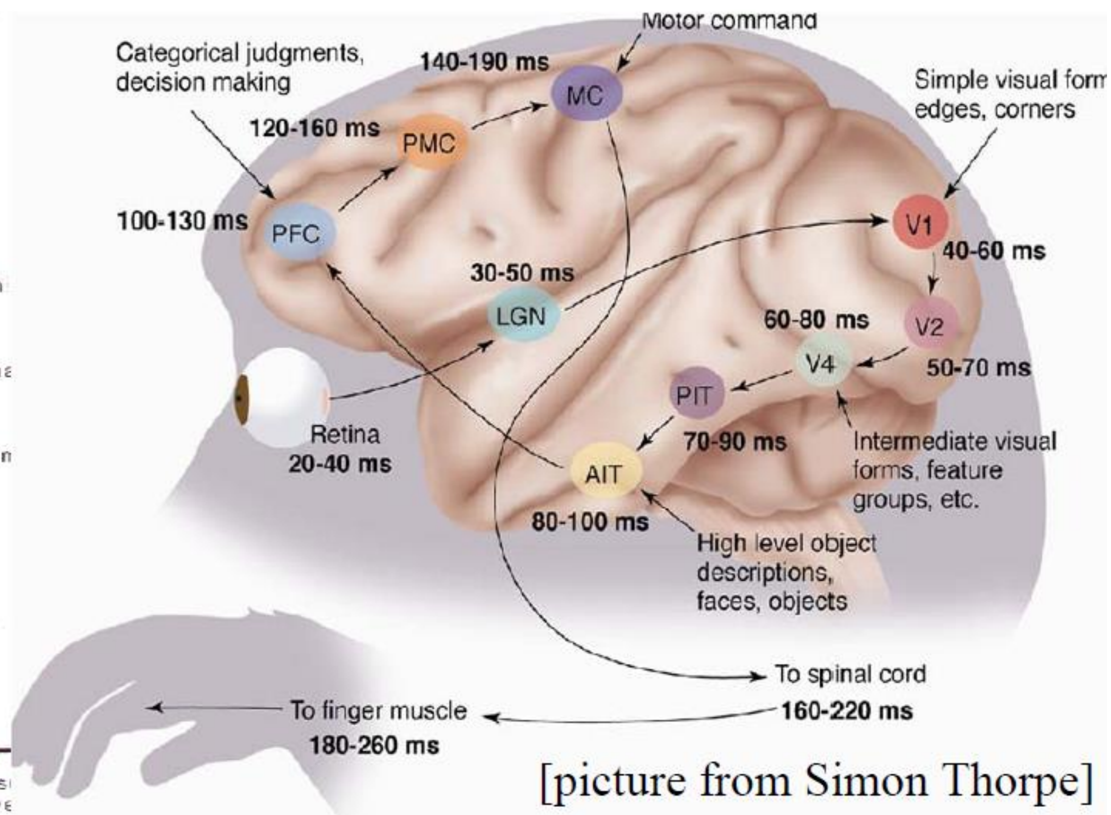
- 현재 연결강도의 변화량을 구한 후
- 바로 전 단계 학습 시 연결강도의 변화량에 타성(momentum) α를 곱한 값을 더해준다.

$$\Delta w_{ij}(n+1) = \eta \delta_j y_i + \alpha \Delta w_{ij}(n)$$

Convolutional neural networks

- The ventral (recognition) pathway in the visual cortex has multiple stages
- Retina - LGN - V1 - V2 - V4 - PIT - AIT ....
- Lots of intermediate representations



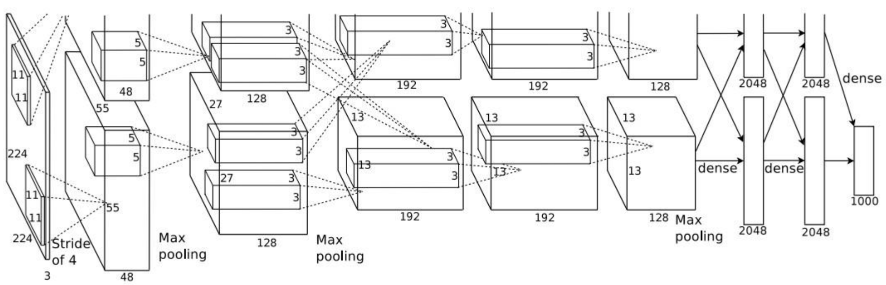


[picture from Simon Thorpe]

[Gallant & Van Essen]

Convolutional neural networks

- Type of neural network for processing data having grid-like topology
  - Time series (1D grid)
  - Images (2D grid)
  - Video (3D grid)
- Components: convolutional layers, activation of units, pooling layers,
- Weights are learned with backpropagation



Convolutional neural networks

- Convolution : 신호와 필터간의 element-wise 곱 연산 후 더함

$$\underset{\text{신호}}{\begin{bmatrix} 2 & 2 & 2 \\ 10 & 10 & 10 \\ 2 & 2 & 2 \end{bmatrix}} \cdot \underset{\text{필터}}{\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}} = 30$$

$$= 2\times0+2\times0+2\times0+10\times1+10\times1+10\times1 +2\times0+2\times0+2\times0$$

$$\underset{\text{신호}}{\begin{bmatrix} 2 & 10 & 2 \\ 2 & 10 & 2 \\ 2 & 10 & 2 \end{bmatrix}} \cdot \underset{\text{필터}}{\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}} = 14$$

$$= 2\times0+10\times0+2\times0+2\times1+10\times1+2\times1 +2\times0+10\times0+2\times0$$

Convolutional neural networks

- Pooling : 신호의 크기를 축소시키는 (Subsampling) 연산

0	1	2	4
5	6	6	8
3	2	1	0
1	2	3	4

신호

필터 종류 : max pooling
필터 크기 : 2×2
Stride : 2

6	8
3	4

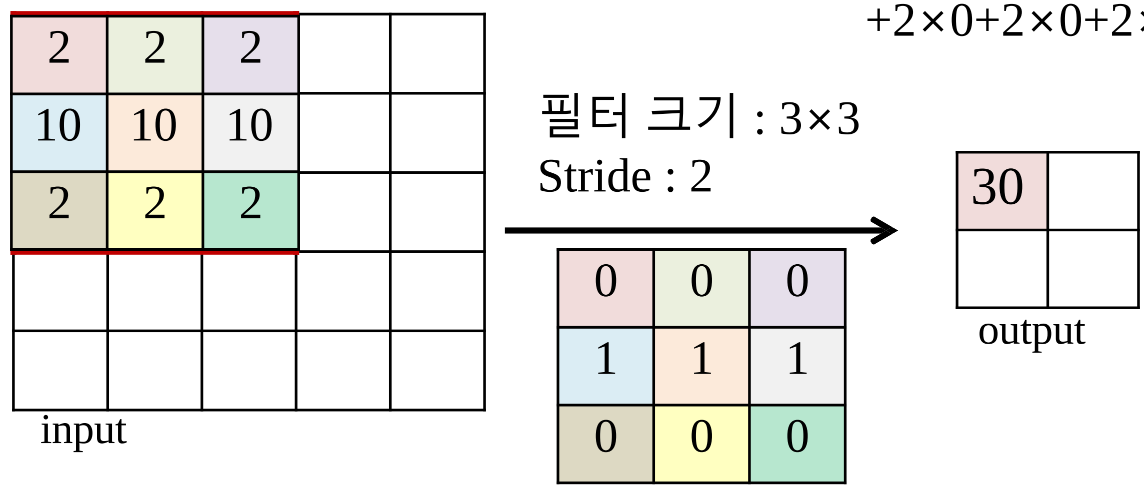
필터 종류 : average pooling
필터 크기 : 2×2
Stride : 2

3	5
2	2

Convolutional neural networks

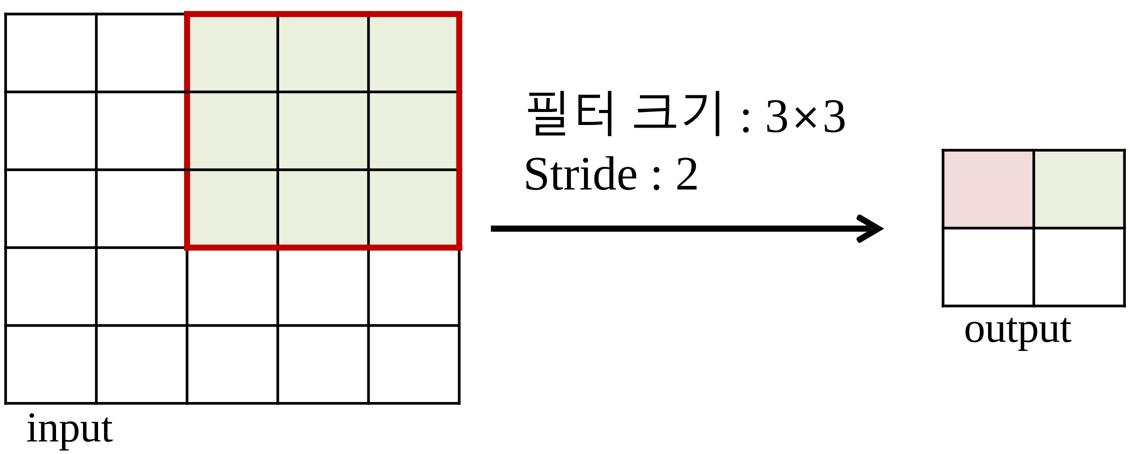
- Stride : 필터 연산의 수행간격
  - 예) stride 2

$$= 2\times0+2\times0+2\times0+10\times1+10\times1+10\times1$$
$$+2\times0+2\times0+2\times0$$



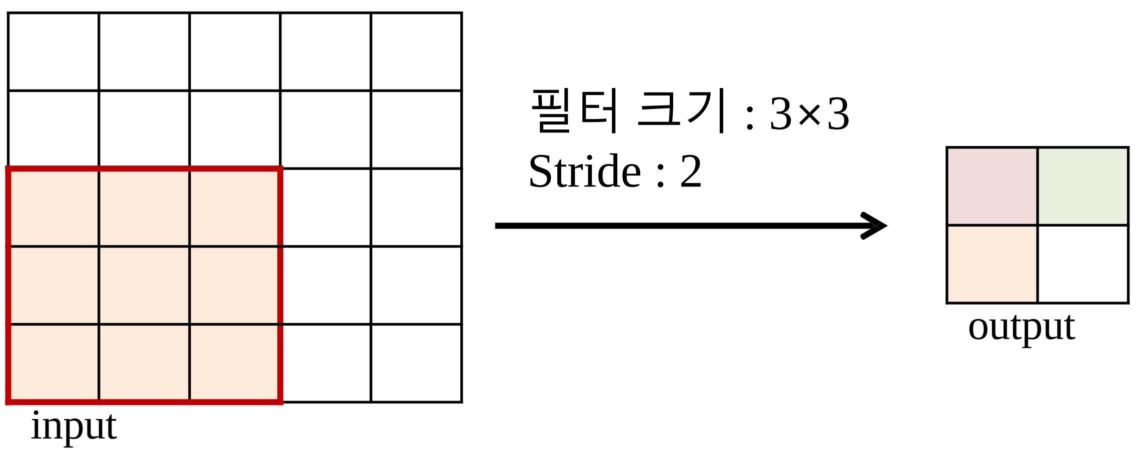
Convolutional neural networks

- Stride : 필터 연산의 수행간격
  - 예) stride 2



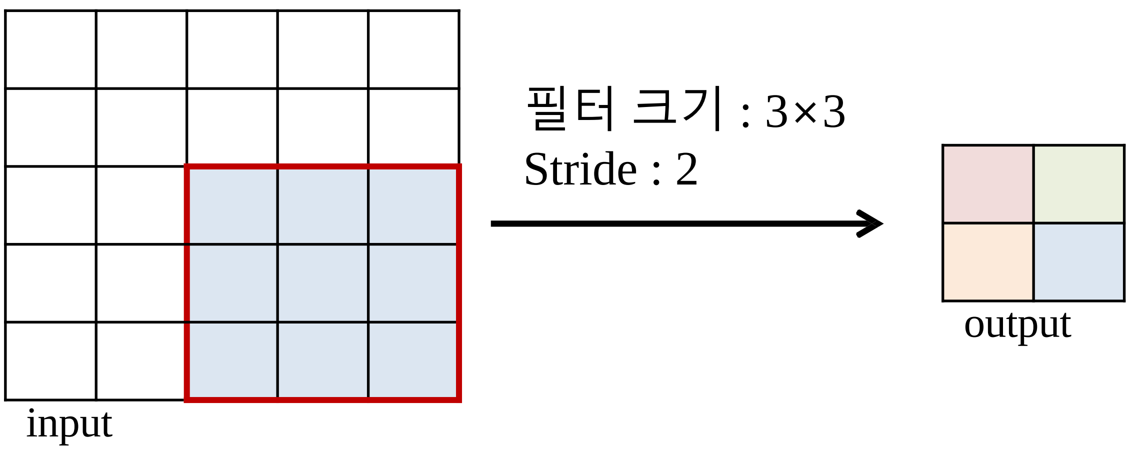
Convolutional neural networks

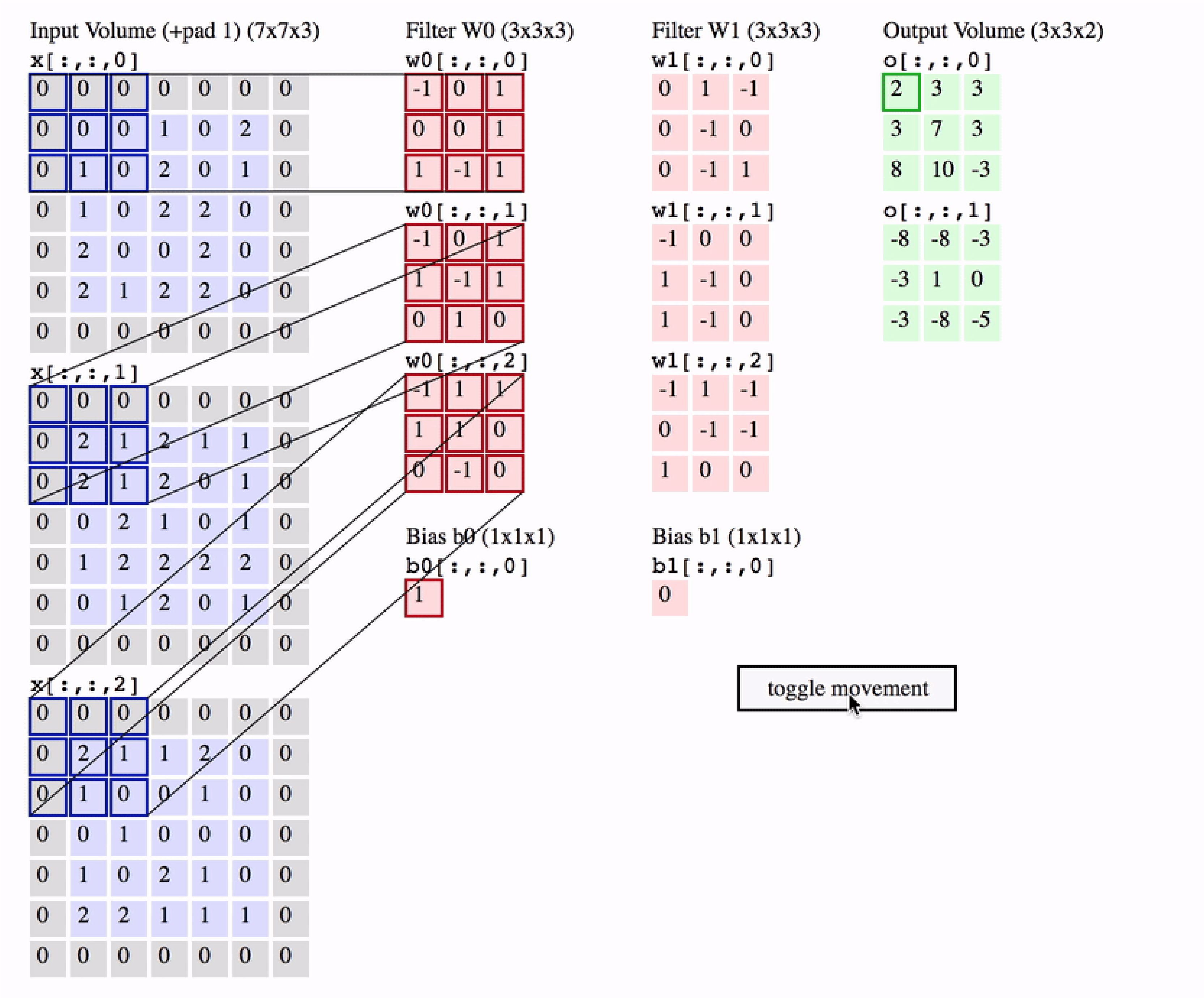
- Stride : 필터 연산의 수행간격
  - 예) stride 2



Convolutional neural networks

- Stride : 필터 연산의 수행간격
  - 예) stride 2




Input Volume (+pad 1) (7x7x3)
x[:,:,0]
x[:,:,1]
x[:,:,2]
Filter W0 (3x3x3)
w0[:,:,0]
w0[:,:,1]
w0[:,:,2]
Bias b0 (1x1x1)
b0[:,:,0]
Filter W1 (3x3x3)
w1[:,:,0]
w1[:,:,1]
w1[:,:,2]
Bias b1 (1x1x1)
b1[:,:,0]
Output Volume (3x3x2)
o[:,:,0]
o[:,:,1]
toggle movement

Convolutional neural networks

- How do the filters look like?