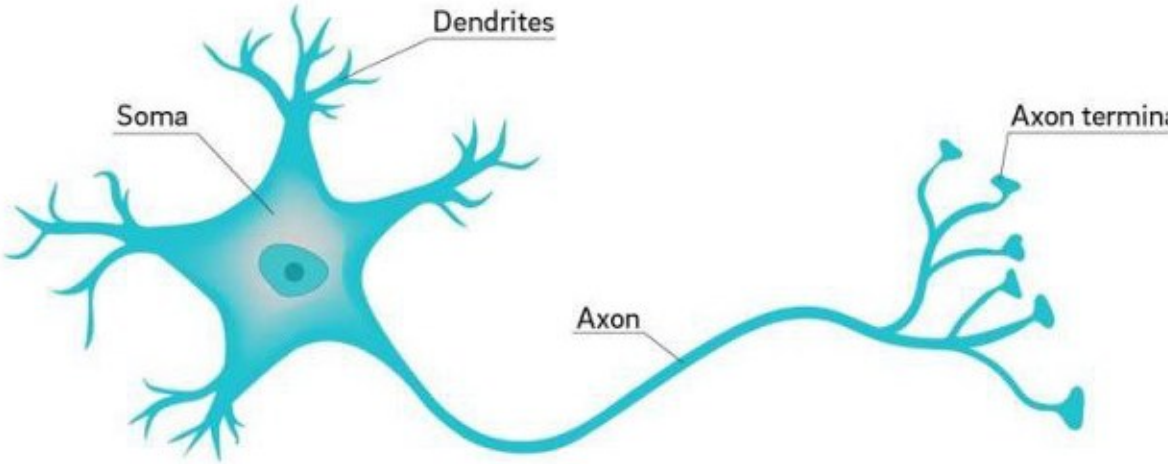
Deep Learning

Module 7

Biological Neuron

- **Why there is need to study biological neuron?**
- **Single Neuron & Single Layer Network**



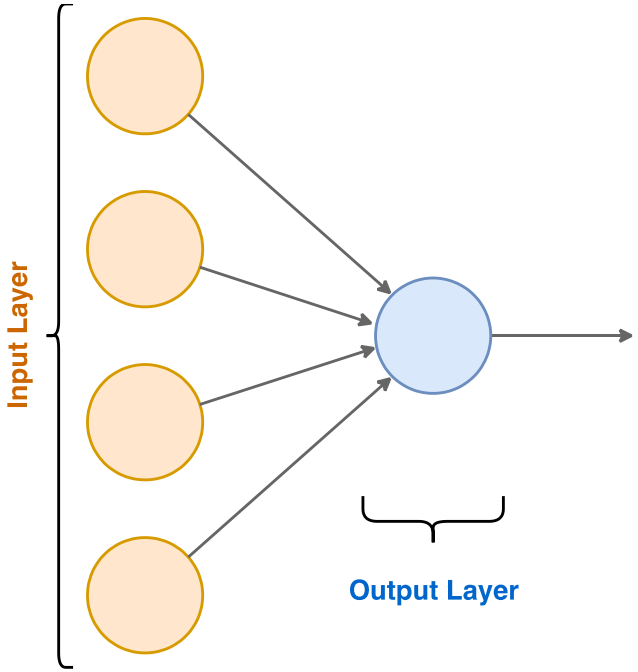
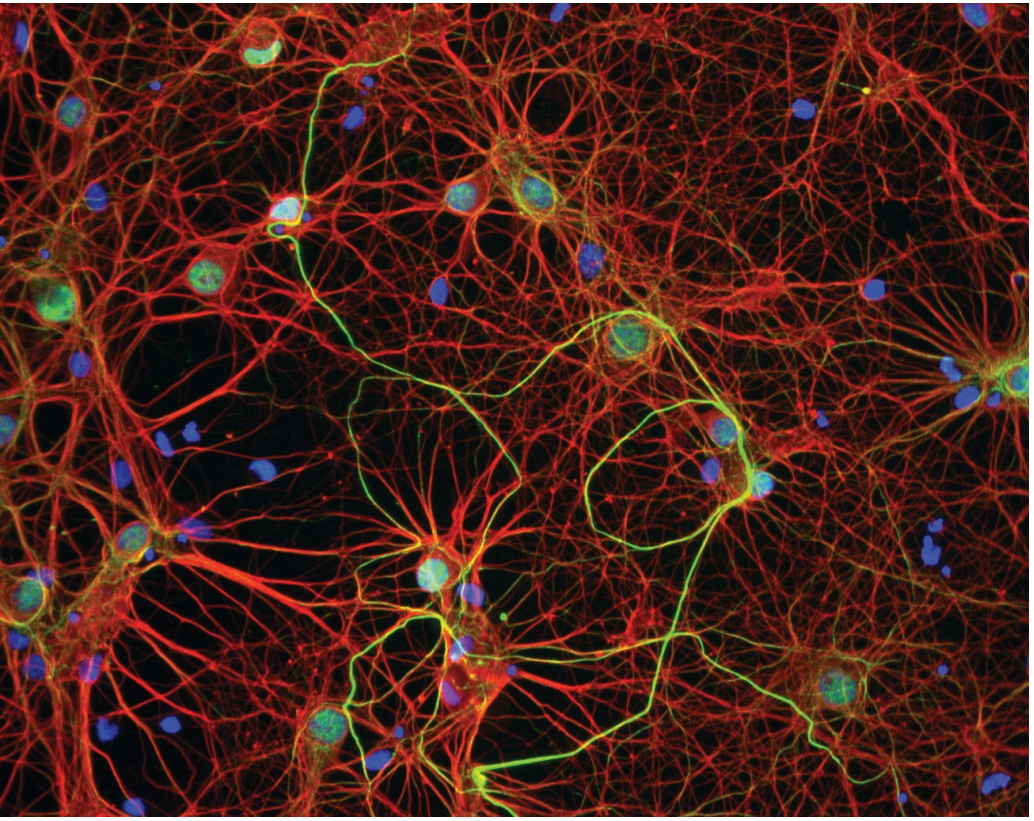
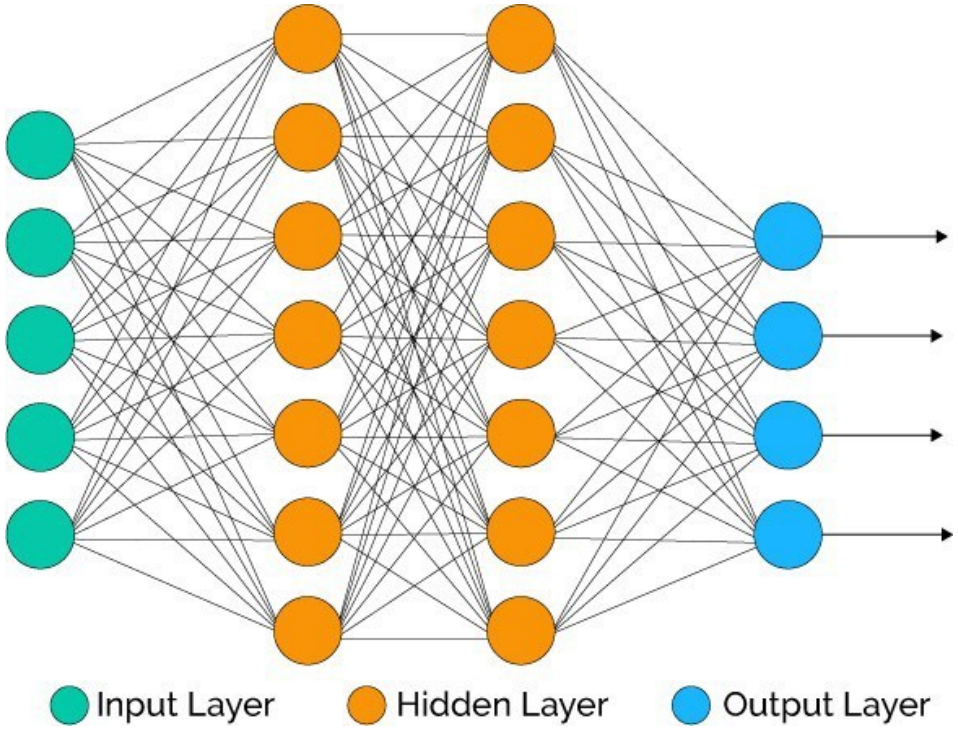


Fig: Single Biological Neuron & Single Layer Neural Network

Biological Neuron

- **Multiple Neurons & Multi Layer Network**





Jennifer Walinga & Charles Stango "*Introduction to Psychology : Brains, Bodies, and Behaviour*"

Types of Layers

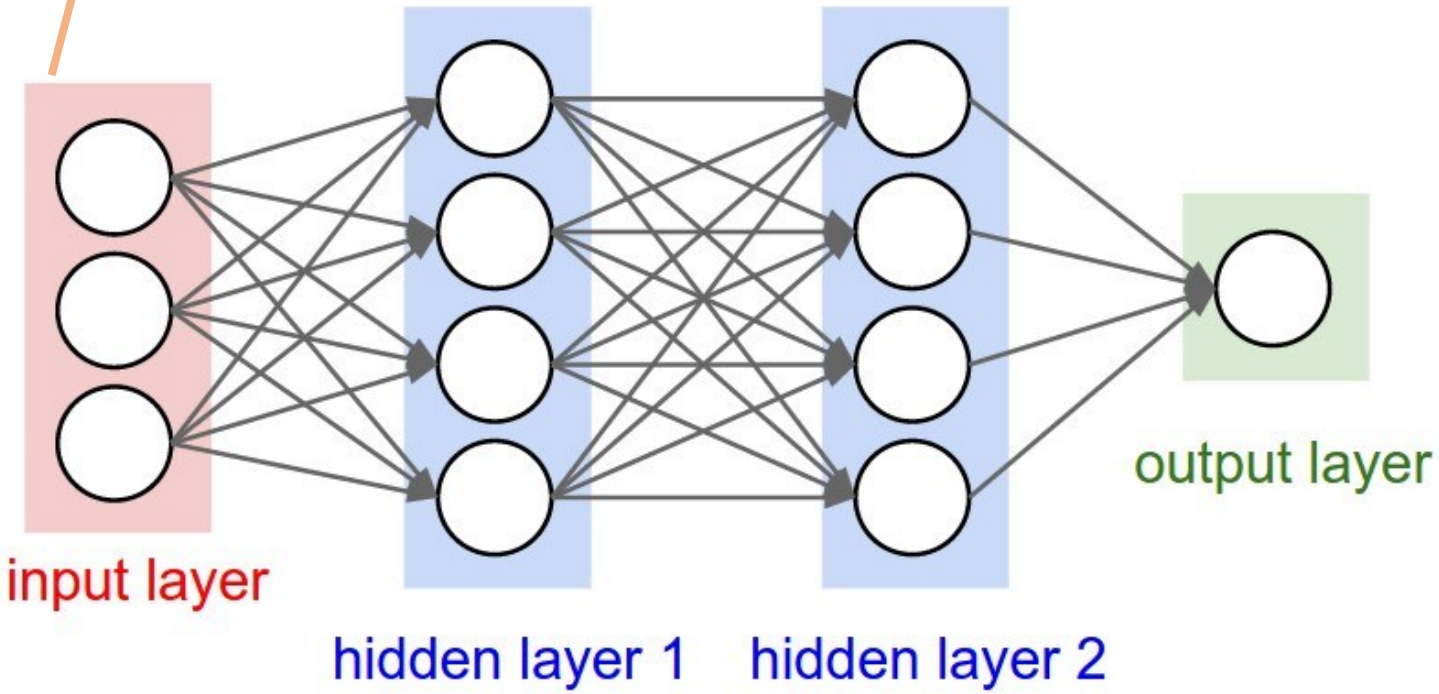


Fig. Multi layer feedforward network.

Input Layer —

- First layer in NN.
- Takes input signals(values) and passes them on to the next layer.
- **Doesn't apply any operations**
- Act as an interface

Types of Layers

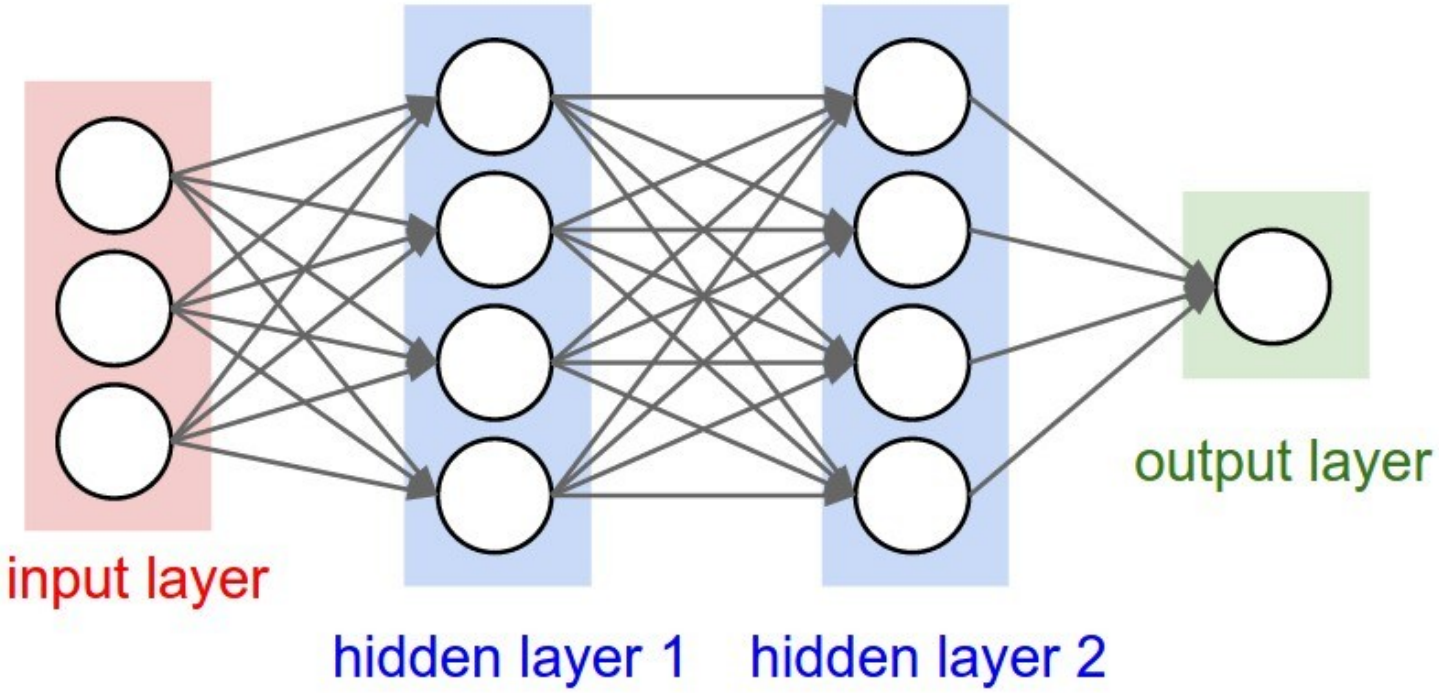
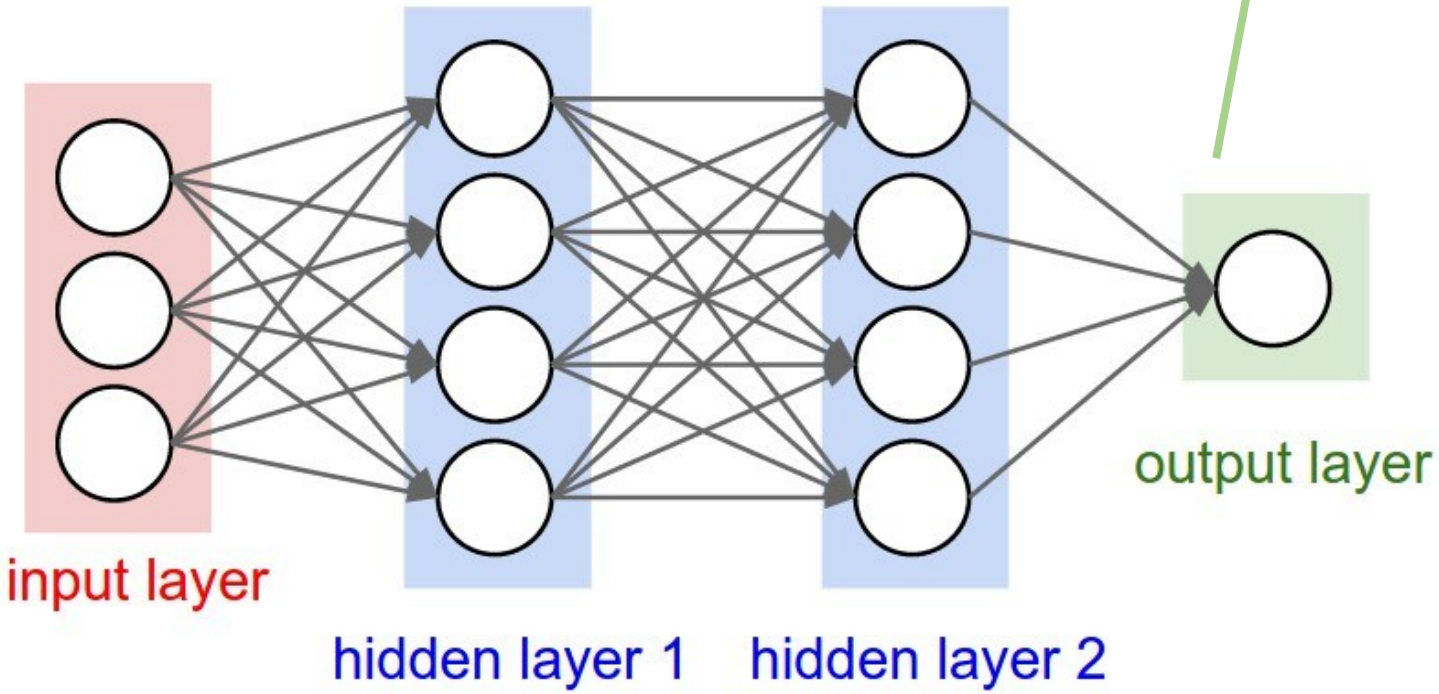


Fig. Multi layer feedforward network.

Hidden Layers —

- Have neurons which **apply different transformations to the input data**.
- One hidden layer is a collection of neurons **stacked vertically.**
- Last hidden layer passes on values to the output layer.
- All the neurons in a hidden layer are connected to each and every neuron in the next layer, hence we have **a fully connected hidden layers**.

Types of Layers

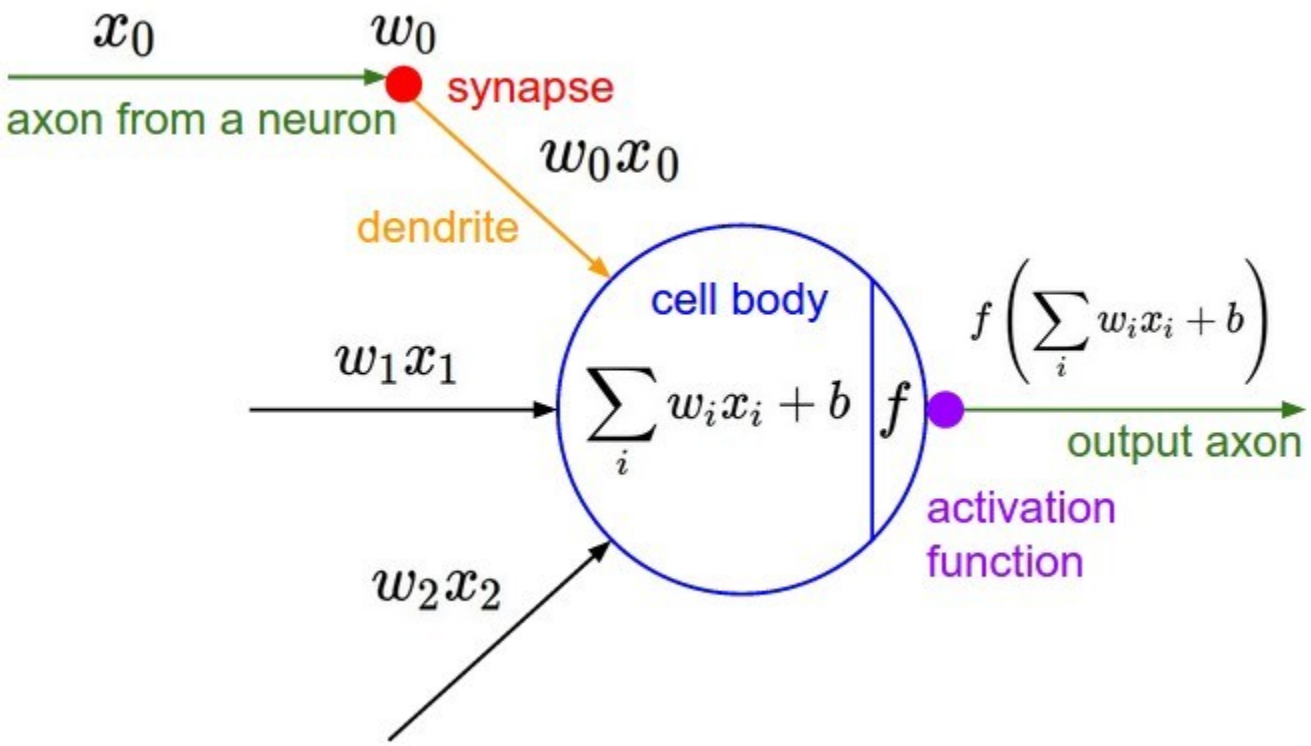


Output Layer —

- Last layer in the network & receives input from the last hidden layer.

Fig. Multi layer feedforward network.

Basic ANN Architecture

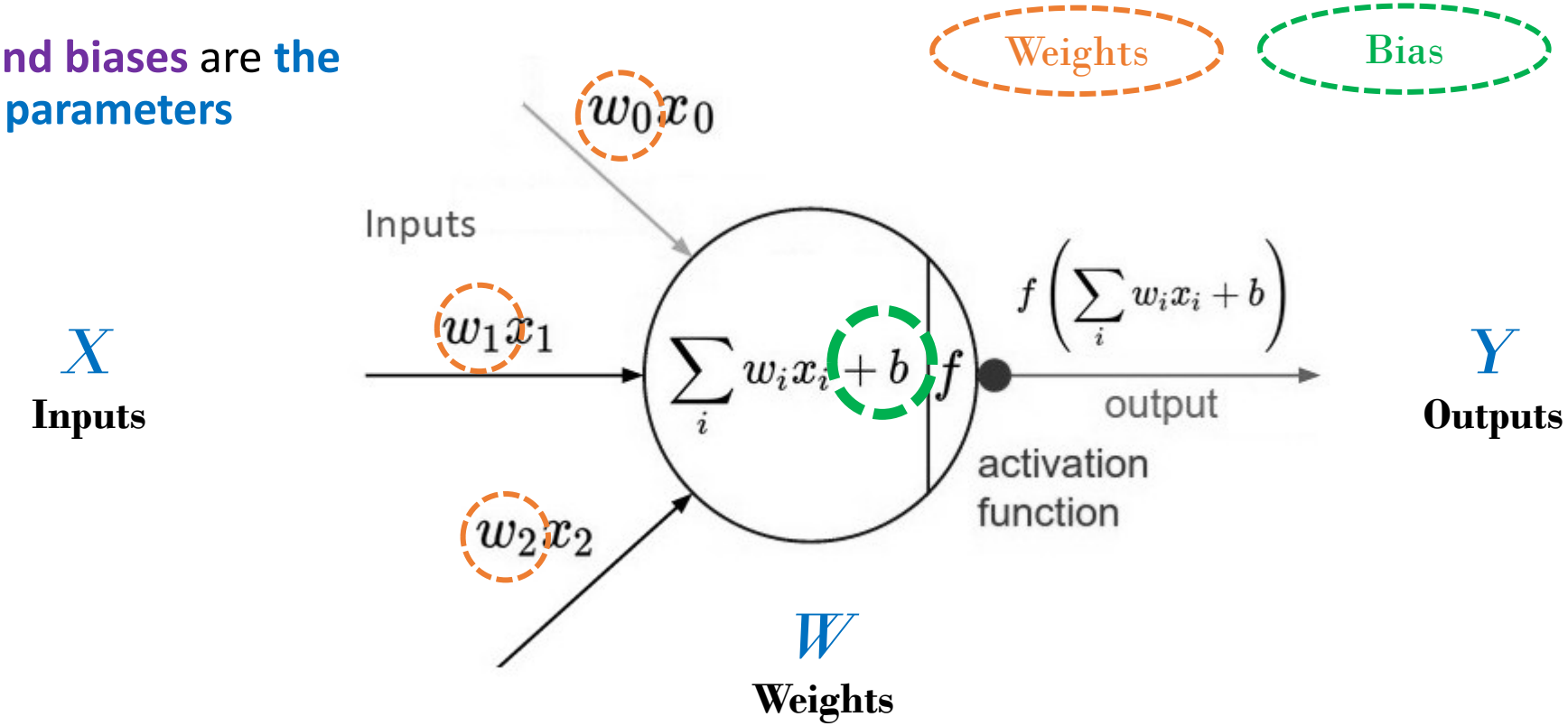


Building Blocks of ANN

- Weights
- Calculation of Net Input
- Activation Functions
- Threshold & Bias

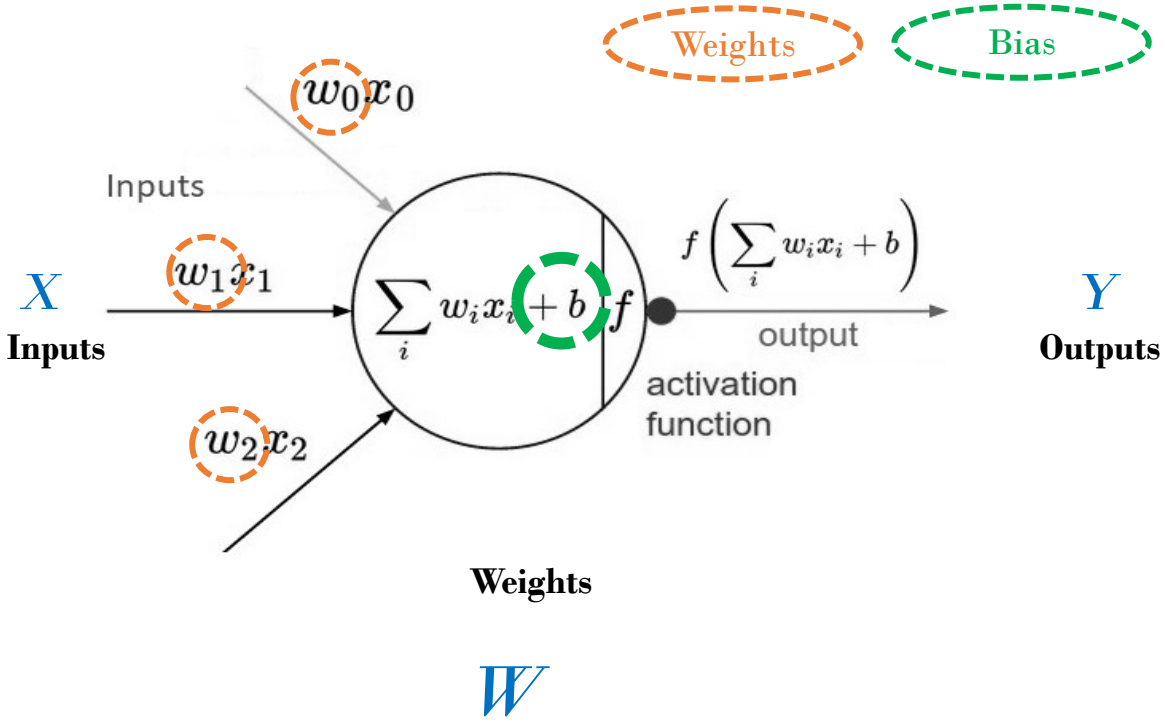
Weights

Weights and biases are **the learnable parameters**



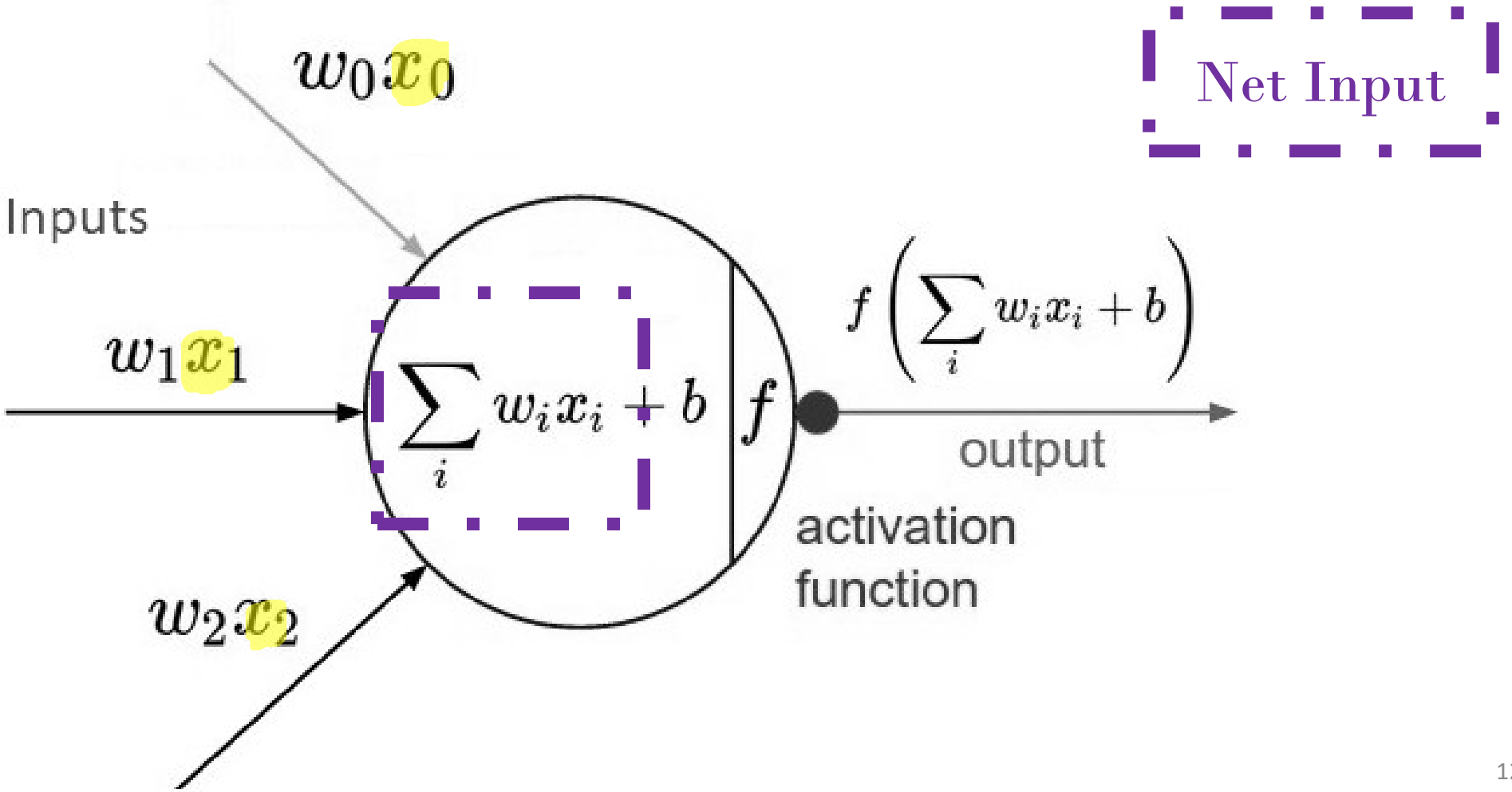
Weights

- **Weights**
  - Control the signal (or the strength of the connection) between two neurons.
  - **A weight decides how much influence the input will have on the output.**
- **Biases**
  - Which are constant, are **<u>an additional input into the next layer that will always have the value of 1.</u>**
  - **Bias units are not influenced by the previous layer** (they do not have any incoming connections) **but they do have outgoing connections with their own weights.**
  - **The bias unit guarantees that even when all the inputs are zeros there will still be an activation in the neuron.**



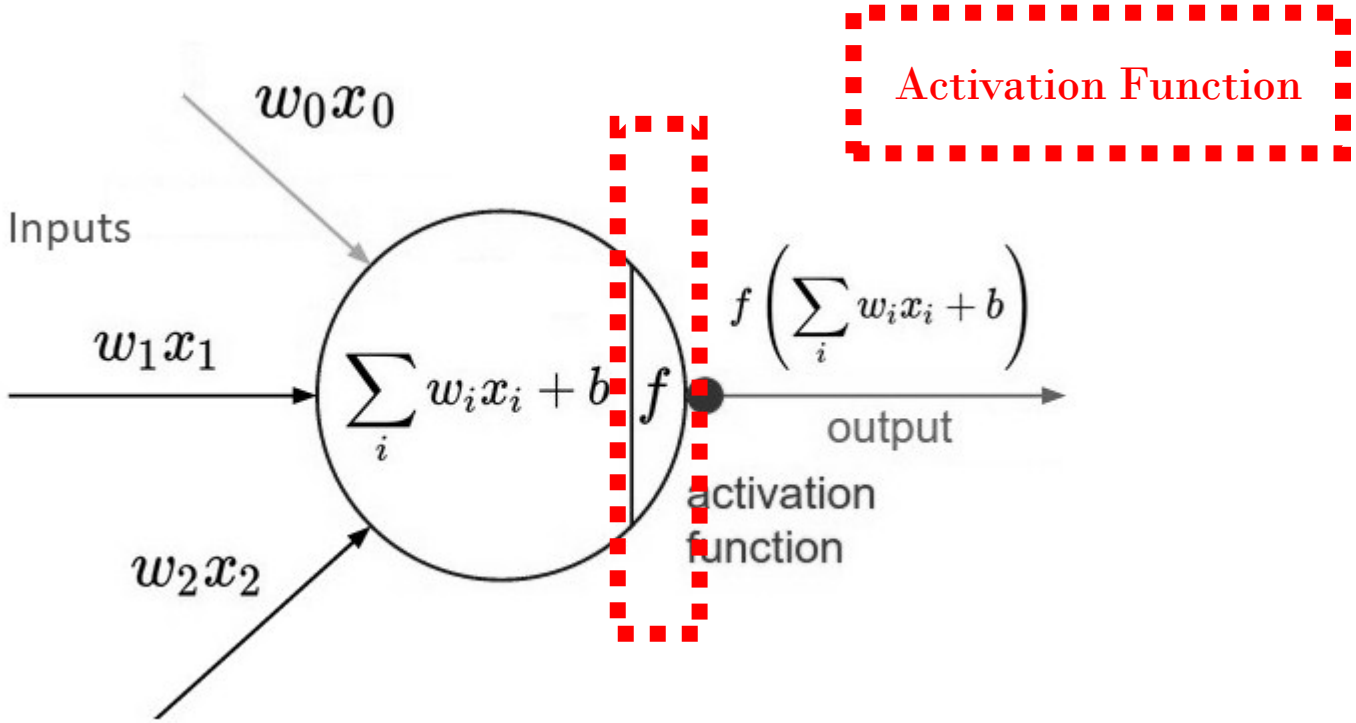
Net Input & Activation Functions

Calculation of Net Input



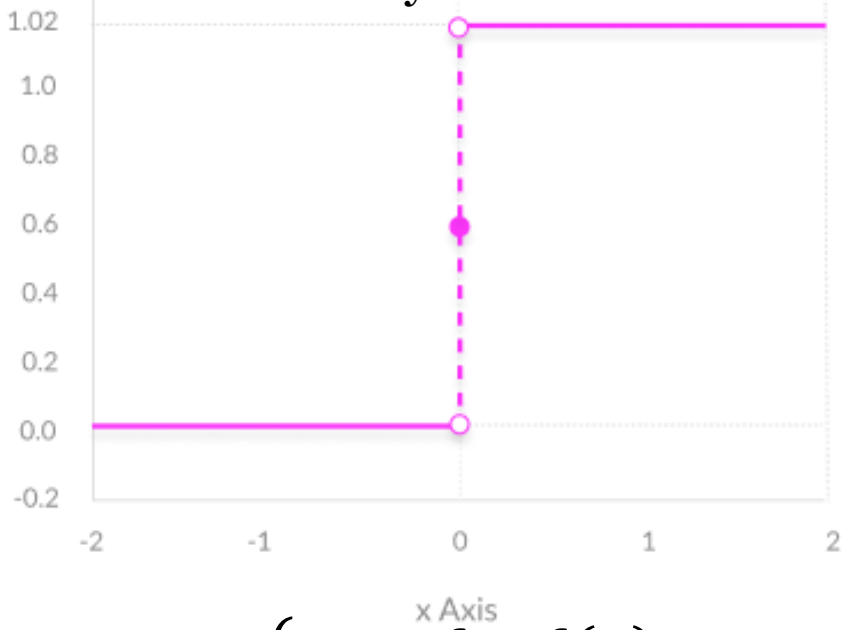
Activation Function

- **Activation Functions**
  - Calculate the output response of a neuron.
  - Mathematical Exp.
- **Types**
  - Identity
  - Binary Step Function
  - Sigmoidal
    - Binary & Bipolar
  - ReLu



Activation Functions … (contd.)

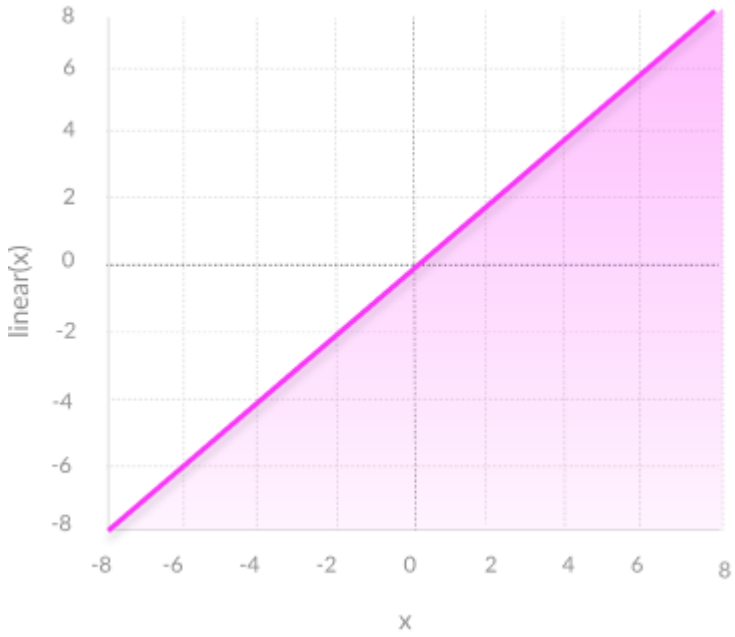
- Binary Step Activation Function
  - Threshold based function
  - **Problem**: O/p is always 1; so no multi-value o/p
  - Limited to **Binary Classification**



$$f(x) = \begin{cases} 1; & if \quad f(x) > 0 \\ 0; & if \quad f(x) < 0 \end{cases}$$

Linear Activation Function

- **Output signal proportional to the input**
- **A linear regression model**

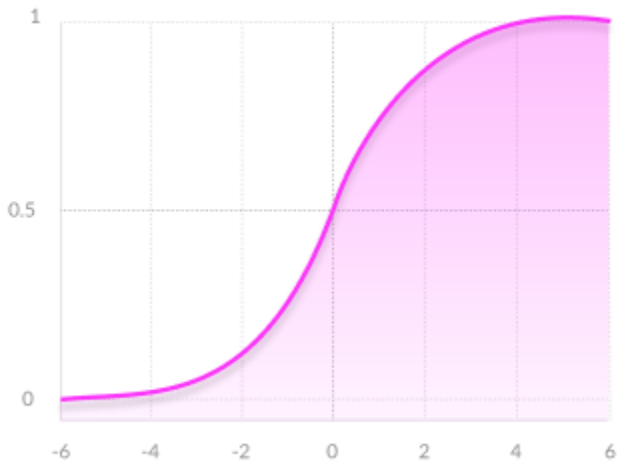


$$f(x) = x$$

Activation Functions … (contd.)

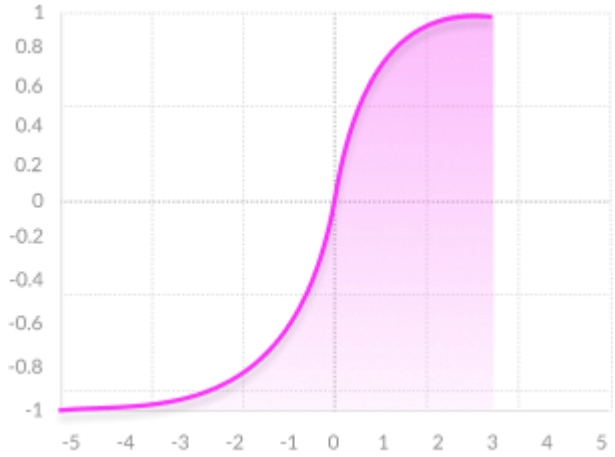
- Sigmoidal Activation Function

Binary



$$f(x) = \frac{1}{1 + exp(-\sigma x)}$$

Bipolar



$$f(x) = \frac{1 - exp(-\sigma x)}{1 + exp(-\sigma x)}$$

- Non Linear Activation Functions
- Create complex mappings between the network's inputs and outputs
- Function is **differentiable**

Threshold & Bias

- **Output of Net**

$$Net = b + \sum x_i w_i$$

- **Output of AF if *Bias* is included**

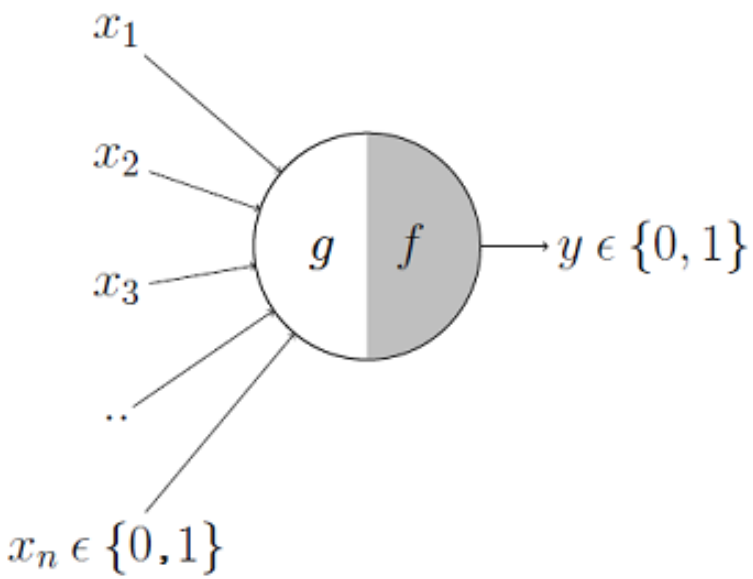
$$y = f(Net) = \begin{cases} +1; & if \quad net \geq 0 \\ -1; & if \quad Net < 0 \end{cases}$$

- **Output of AF if *threshold θ* is considered**

$$y = f(Net) = \begin{cases} +1; & if \quad net \geq \theta \\ -1; & if \quad Net < \theta \end{cases}$$

McCulloch-Pitts Model

- In 1943 by Warren McCulloch (neuroscientist) and Walter Pitts (logician) proposed an **computational model**:
  - By mimicking the functionality of a biological neuron



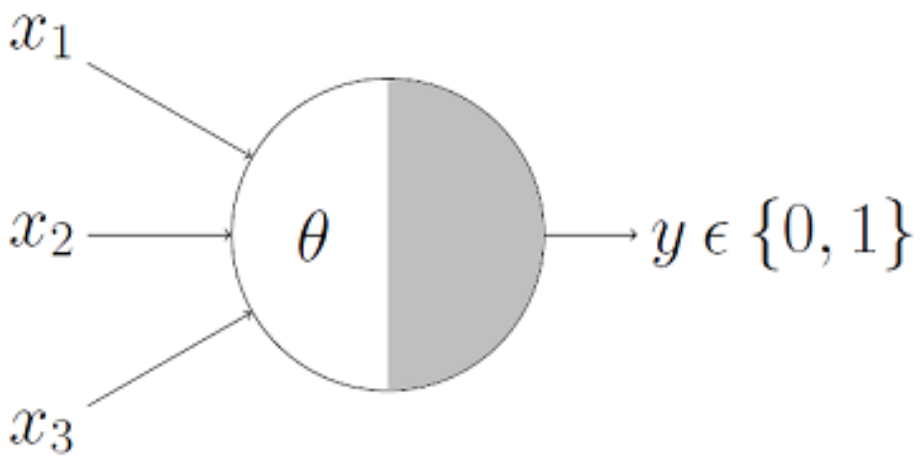
- **Divided 2 parts.**
  - The first part, ***g*** takes an input, <u>performs an aggregation</u>
  - The second part, ***f*** <u>makes a decision</u> based on the aggregated value.

- Now **g(x)** is just doing a sum of the inputs
- ***Theta*** here is called thresholding parameter.
- The ***output y*** is the function of q(x) i.e. it's a function of the sum of the inputs,
- The inputs in case of McCullochs Pitts model are Boolean.

$$g(x_1, x_2, x_3, ..., x_n) = g(\mathbf{x}) = \sum_{i=1}^{n} x_i$$

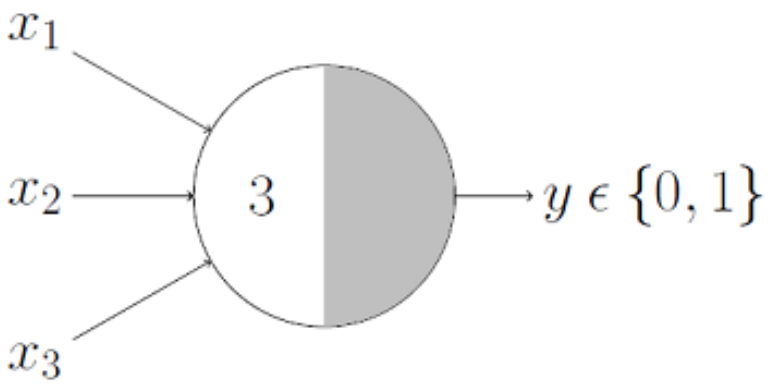
$$\begin{aligned} y = f(g(\mathbf{x})) &= 1 \quad if \quad g(\mathbf{x}) \geq \theta \\ &= 0 \quad if \quad g(\mathbf{x}) < \theta \end{aligned}$$

Boolean Functions using McCulloch Neuron

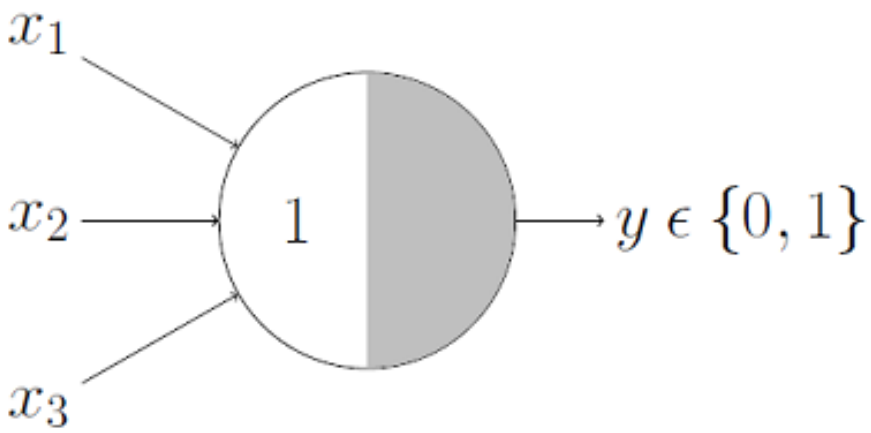


Boolean Functions using McCulloch Neuron … (Contd)

- **AND Logic**



OR Logic



Perceptron : Introduction

- Introduced by ***Frank Rosenblatt*** in 1957. [Link]
  - "***Principles of Neuro-dynamics: Perceptrons and the Theory of Brain Mechanisms***" 1962.
  - Work was criticized by **Marvin Minksy and Seymour Papert**
    - **Arguing that neural networks were flawed and could only solve linear separation problem**.

- **Perceptron is the simplest form of neural network**
  - Used for the classification of patterns said to be ***linearly separable***

- Algorithm for ***Supervised Learning*** of binary classifiers.
- Enables neurons to ***learn*** **&** ***processes*** elements in the training set one at a time.
- ***The Perceptron Algorithm***
  - Learns the weights for the input signals in order to draw a linear decision boundary.

Perceptron Learning Rule

- States that:
  "The algorithm would automatically learn the optimal weight coefficients."
- Learning signal:
  *"Difference between the **desired** and the **actual** neurons response:*

- **Rule States that:**
  - For a finite 'n' number of training input vectors x(n) and each with associated target value t(n):

    $\boldsymbol{x(n)}$ **where n=1 to N**

    $\boldsymbol{t(n)}$ **where n=1 to N**
  - and an activation function $\boldsymbol{y = f(y_{in})}$

$$f(y_{in}) = \begin{cases} 1 & if \quad y_{in} > \theta \\ 0 & if \quad -\theta \leq y_{in} \leq \theta \\ -1 & if \quad y_{in} < -\theta \end{cases}$$

Perceptron Activation Functions

- In perceptron model; the chosen activation function is a ***step function***

- 2 Possible Choices:
  a. Heaviside function
  b. Sign functions

$$f(y_{in}) = \begin{cases} 1 & if \quad y_{in} \geq 0 \\ 0 & if \quad y_{in} < 0 \end{cases}$$

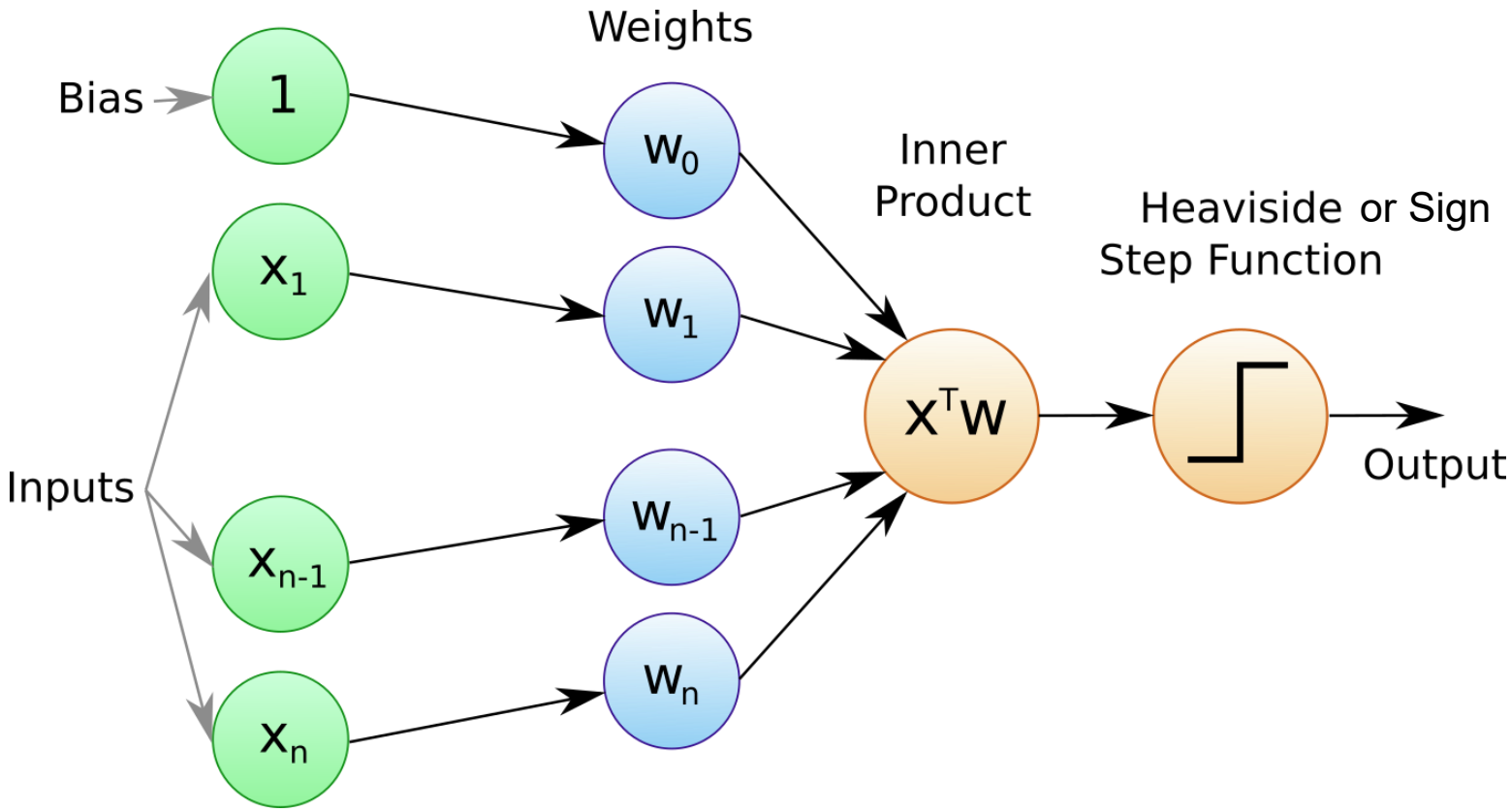
Heaviside function

$$f(y_{in}) = \begin{cases} 1 & if \quad y_{in} > \theta \\ 0 & if \quad -\theta \leq y_{in} \leq \theta \\ -1 & if \quad y_{in} < -\theta \end{cases}$$

Sign function

Simple Perceptron Architecture

- Single Layer Perceptron Model



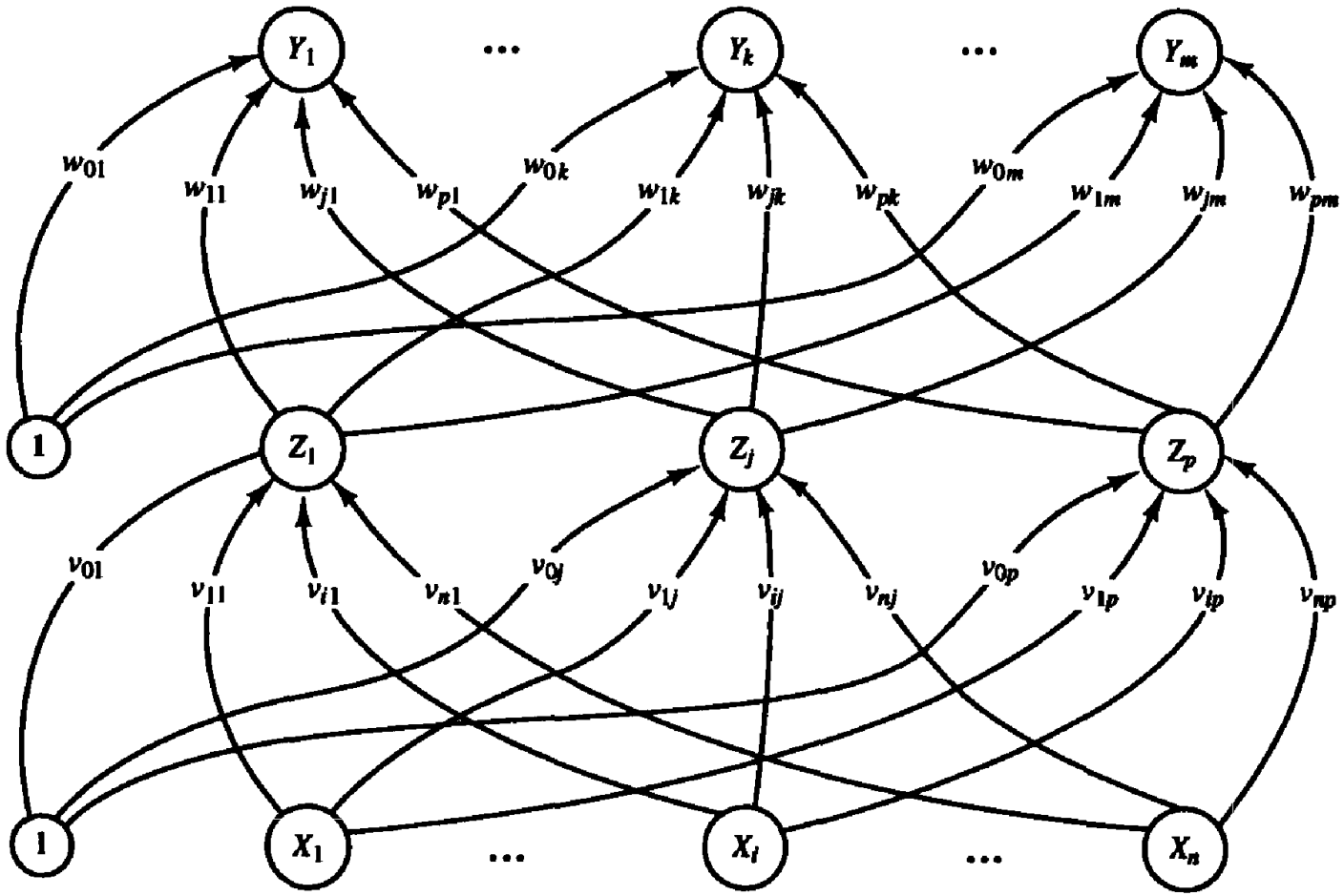
Perceptron Algorithm

Step 0	**Initialize all weights**		
	$\boldsymbol{w_i = 0}$ **, b = 0**	(i = 1 … n)	n is the no. of input neurons
	Set Learning Rate (0 < $\boldsymbol{\alpha \leq 1}$)	**(for simplicity set $\boldsymbol{\alpha = 1}$)**	
Step 1	While Stopping Criteria is FALSE, Do Steps 2 - 6		
Step 2	For each : Input Training Vector (s) & Target Output Pairs (t) ***Do Steps : 3 - 5***		
	Step 3 : Set activations for **input** units		
	$\boldsymbol{x_i = s_i}$	(i = 1 … n)	
	Step 4 : Find response of **output** units		
	$\boldsymbol{y} = \begin{cases} 1 & if \quad y_{in} > \theta \\ 0 & if \quad -\theta \leq y_{in} \leq \theta \\ -1 & if \quad y_{in} < -\theta \end{cases}$		
	Step 5 : Update the **Weights & Bias** if an error has occurred for this pattern		
	If $\boldsymbol{y \neq t}$ $\boldsymbol{w_i(new) = w_i(old) + \alpha t x_i}$ $\boldsymbol{b(new) = b(old) + \alpha t}$		(i = 1 … n)
	Else $\boldsymbol{w_i(new) = w_i(old)}$ $\boldsymbol{b(new) = b(old)}$		
	Step 6 : Check Stopping Criteria		
	If **no weights changed** in Step 2, then **STOP**; else continue.		

Backpropagation NN

- A multilayer, feedforward net trained by backpropagation
  - *This is the step where we send the error backwards to update the weights.*
- Aim is to train the NNet to achieve a balance between the
  - *Ability to respond correctly to the input patterns (memorization) and*
- *Ability to give reasonable (good) responses to input that is similar (generalization).*
- Training of a NN by Backpropagation involves three stages:
  - The **feedforward** of the input training pattern,
  - The **calculation and backpropagation** of the **associated error**, and
  - The **adjustment of the weights**.
- After training, **application** of the net involves only **the computations of the feedforward phase**.

BPNN : Architecture



BPNN : ALGORITHM

- Training a network by backpropagation involves three stages:
  - The **feedforward** of the input training pattern,
  - The **calculation and backpropagation** of the **associated error**, and
  - The **adjustment of the weights**.

- **During feedforward,**
  - Each input unit (X_i) receives an input signal and broadcasts this signal to the each of the hidden units $Z_1, \ldots, Z_p$.
  - Each hidden unit then computes its activation and sends its signal (z_i) to each output unit.
  - Each output unit (Y_k) computes its activation (y_k) to form the response of the net for the given input pattern.
  - During training, each output unit compares its computed activation y_k with its target value t_k to determine the associated error for that pattern with that unit.

BPNN : ALGORITHM

- **Based on this error**, the factor δ_k (k = 1, . . . , m) is computed.
  - δ_k is used to distribute the error at output unit Y_k back to all units in the previous layer.
  - In a similar manner, the factor δ_j (j = 1, . . . , p) is computed for each hidden unit

  - After all of the δ factors have been determined,
    - The weights for all layers are adjusted simultaneously
    - The adjustment of weights w_{jk} (from Hidden Unit Z_j to Output Unit Y_k) is based on the factor δ_k and the activation z_j of the hidden unit

  - The adjustment to the weight v_{ij} ; (from input unit X_i, to hidden unit Z_j) is based on the factor δ_j and the activation x_i of the input unit.

BPNN : ALGORITHM

Nomenclature:

x	:	Input training vector: $x = (x_1, \ldots, x_i, \ldots, x_n)$
t	:	Output target vector: $t = (t_1, \ldots, t_k, \ldots, t_m)$
$\boldsymbol{\delta_k}$	:	Portion of error correction weight adjustment for w_{jk} that is due to an error at output unit Y_k; also, the information about the error at unit Y_k that is propagated back to the hidden units that feed into unit Y_k.
$\boldsymbol{\delta_j}$	:	Portion of error correction weight adjustment for that is due to the backpropagation of error information from the output layer to the hidden unit
a	:	Learning rate.
$\boldsymbol{X_i}$	:	Input unit *1:* For an input unit, the input signal and output signal are the same, namely, xi.
$\boldsymbol{v_{0j}}$	:	Bias on hidden unit j.

BPNN : ALGORITHM

Nomenclature:

$\boldsymbol{Z_j}$: Hidden unit *j:*

The net input to *Zj* is denoted

$z_{IN_j} = v_{0j} + \sum_i x_i v_{ij}$

The output signal (activation) of Z; is denoted

$z_j = f\left(z_{IN_j}\right)$

$\boldsymbol{w_{0k}}$: Bias on output unit *k*.

$\boldsymbol{Y_k}$: Output unit *k:*

The net input to *Yk* is denoted *y_in k:*

$y_{IN_k} = w_{0k} + \sum_k z_j w_{jk}$

The output signal (activation) of *Yk* is denoted *Yk:*

$y_k = f\left(y_{IN_k}\right)$

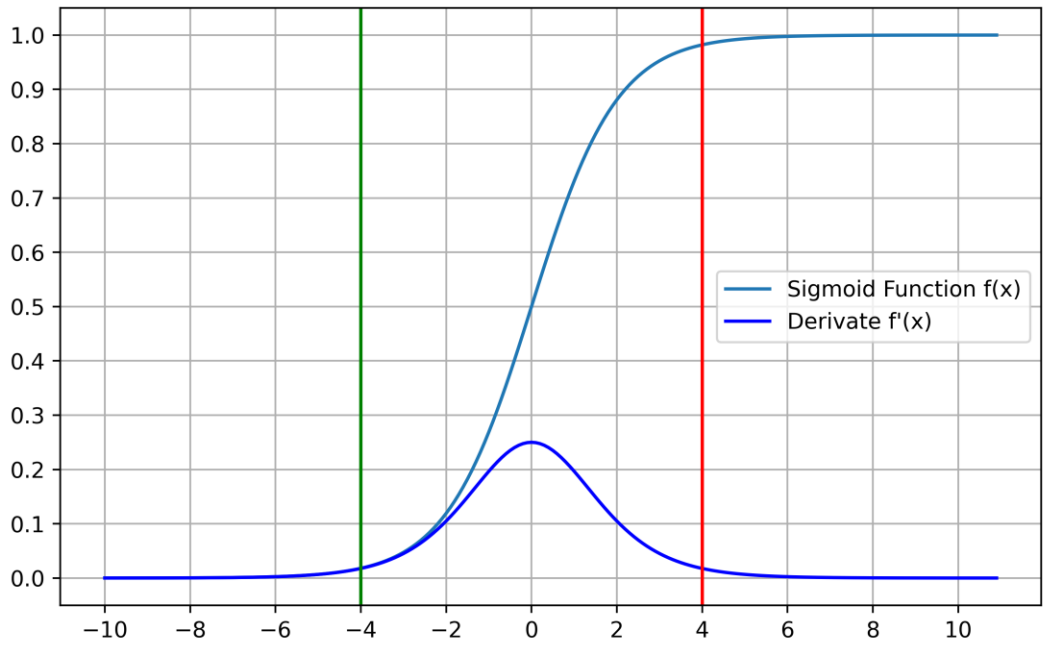
BPNN : Activation Function

Activation Function

- It should be continuous, differentiable, and monotonically non-decreasing.
- For computational efficiency, it is desirable that its derivative be easy to compute.
- Usually, the function is expected to saturate, i.e., approach finite maximum and minimum values asymptotically.
- One of the most typical activation functions is the binary sigmoid function, which has range of (0, 1) and is defined as

$$f_1(x) = \frac{1}{1 + \exp(-x)}$$

$$f_1{}'(x) = f_1(x)[1 - f_1(x)]$$



BPNN : Training Algorithm

Step 0 Initialize weights.
(Set to small random values).

Step 1 While stopping condition is false, do Steps 2-9.

Step 2 For each training pair, do Steps 3-8.

Feedforward:

Step 3 Each input unit ($\boldsymbol{X_i}$, i = 1, . . . , n) receives input signal xi and broadcasts this signal to all units in the layer above (the hidden units).

Step 4 Each hidden unit ($\boldsymbol{Z_j}$, j = 1, . . . , p) sums its weighted input signals,

$$z_{IN_j} = v_{0j} + \sum_i x_i v_{ij}$$

applies its activation function to compute its output signal,

$$z_j = f\left(z_{IN_j}\right)$$

and sends this signal to all units in the layer above (output units).

BPNN : Training Algorithm

Step 5 Each output unit ($\boldsymbol{Y_k}$, k = 1, . . . , m) sums its weighted input signals,

$$y_{IN_k} = w_{0k} + \sum_k z_j w_{jk}$$

and applies its activation function to compute its output signal,

$$y_k = f\left(y_{IN_k}\right)$$

Backpropagation of error:

Step 6 Each output unit ($\boldsymbol{Y_k}$, k = 1, . . . , m) receives a target pattern corresponding to the input training pattern, computes its error information term,

$$\delta_k = (t_k - y_k)\acute{f}\left(y_{IN_k}\right)$$

calculates its weight correction term (used to update $\boldsymbol{w_{ik}}$ later),

$$\Delta w_{jk} = \alpha \delta_k z_j$$

calculates its bias correction term (used to update $\boldsymbol{w_{0k}}$ later),

$$\Delta w_{0k} = \alpha \delta_k$$

and sends $\boldsymbol{\delta_k}$ to units in the layer below.

BPNN : Training Algorithm

Step 7 Each hidden unit ($\boldsymbol{Z_i}$, j = 1, . . . , p) sums its delta inputs (from units in the layer above),

$$\delta_{IN_j} = \sum_{k=1}^{m} \delta_k w_{jk}$$

multiplies by the derivative of its activation function to calculate its information term,

$$\delta_j = \delta_{IN_j} \acute{f}\left(z_{IN_j}\right)$$

calculates its weight correction term (used to update $\boldsymbol{v_{ij}}$ later),

$$\Delta v_{ij} = \alpha \delta_j x_i$$

and calculates its bias correction term (used to update $\boldsymbol{v_{0j}}$ later),

$$\Delta v_{0j} = \alpha \delta_j$$

BPNN : Training Algorithm

Update weights and biases:

Step 8 Each output unit ($\boldsymbol{Y_k}$, k = 1, . . . , m) updates its bias and weights (j = 0, …, p):

$$w_{jk}(new) = w_{jk}(old) + \Delta w_{jk}$$

Each hidden unit ($\boldsymbol{Z_j}$, j = 1, . . . , p) updates its bias and weights (i = 0, . . . , n):

$$v_{ij}(new) = v_{ij}(old) + \Delta v_{ij}$$

Step 9. Test stopping condition.

BPNN : Application

After training, a backpropagation neural net is applied by using only the feedforward phase of the training algorithm. The application procedure is as follows:

Step 0 Initialize weights (from training algorithm).

Step 1 For each input vector, do Steps 2-4.

Step 2 For i = 1, . . , n: set activation of input uni

$$x_i$$

Step 3 For j = 1, . . . , p:

$$z_{IN_j} = v_{0j} + \sum_{i=1} x_i v_{ij}$$

$$z_j = f\left(z_{IN_j}\right)$$

Step 4 For k = 1, . . . , m:

$$y_{IN_k} = w_{0k} + \sum_{i=1} z_i w_{jk}$$

$$y_k = f\left(y_{IN_j}\right)$$

Convolutional Neural Networks

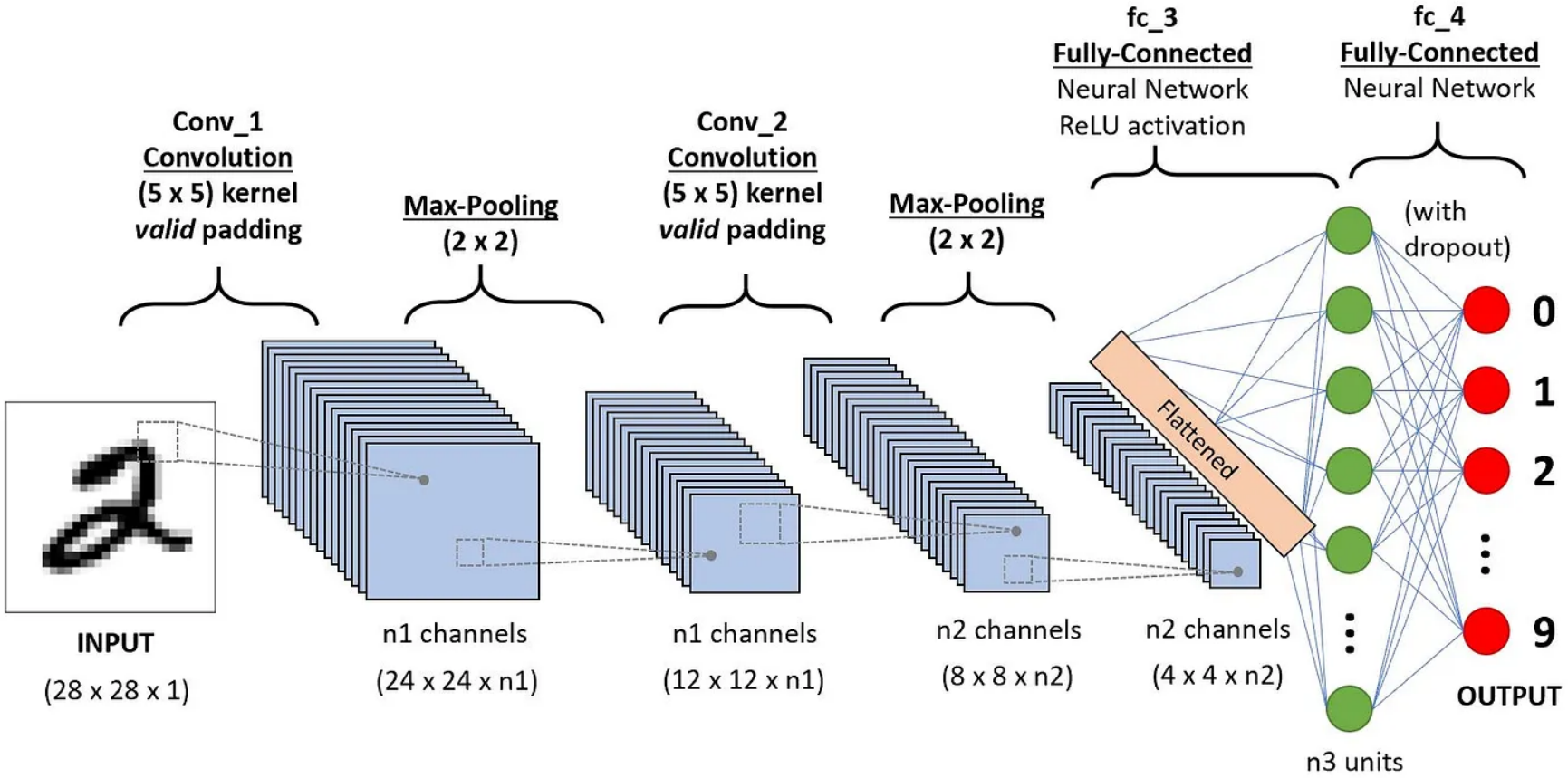
- Convolutional Neural Network (CNN), also known as ***ConvNet***
- A specialized type of deep learning algorithm mainly designed for tasks need object recognition including:
  - Image classification,
  - Detection, and
  - Segmentation.

- **Importance:**
  - CNNs have ability to **autonomously extract features** at a **large scale**, **bypassing** the **need** for **manual feature engineering** and thereby enhancing efficiency.
  - The convolutional layers grant CNNs their *translation-invariant characteristics*, empowering them to *identify* and *extract patterns* and *features* from *data irrespective of variations in position*, *orientation*, *scale*, or *translation*.

Convolutional Neural Networks

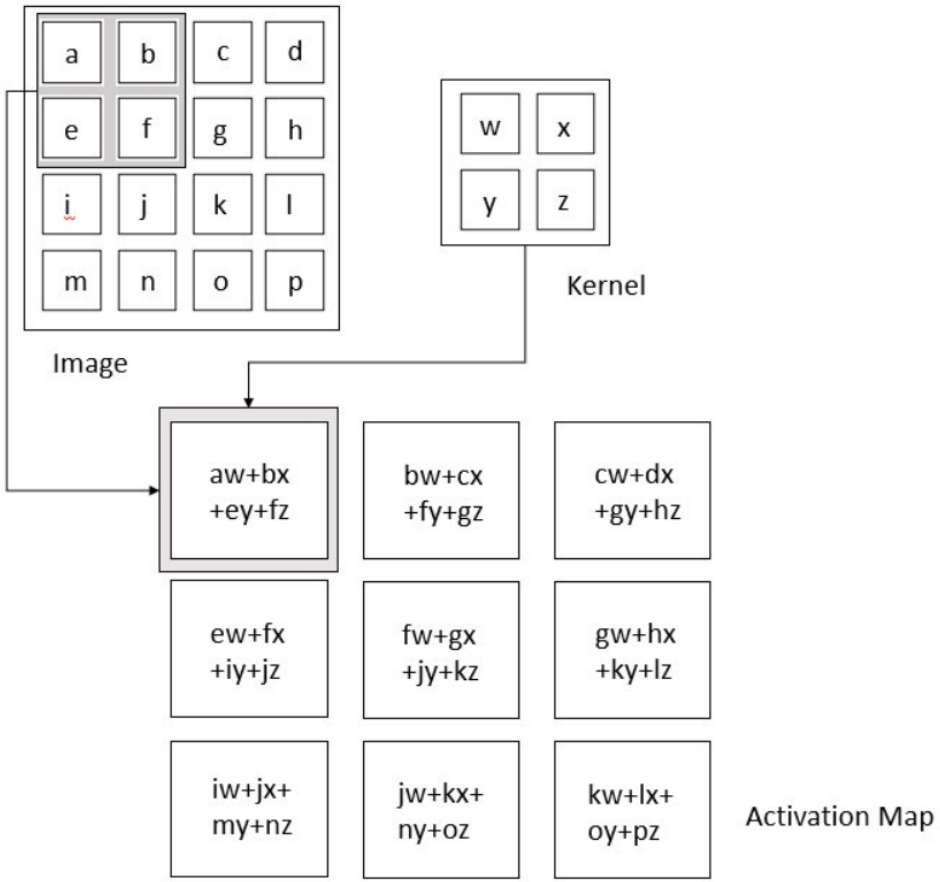
- **Convolution Neural Networks (CNN), a Deep learning algorithm which can adaptively learn spatial hierarchies of features through a back propagation algorithm.**

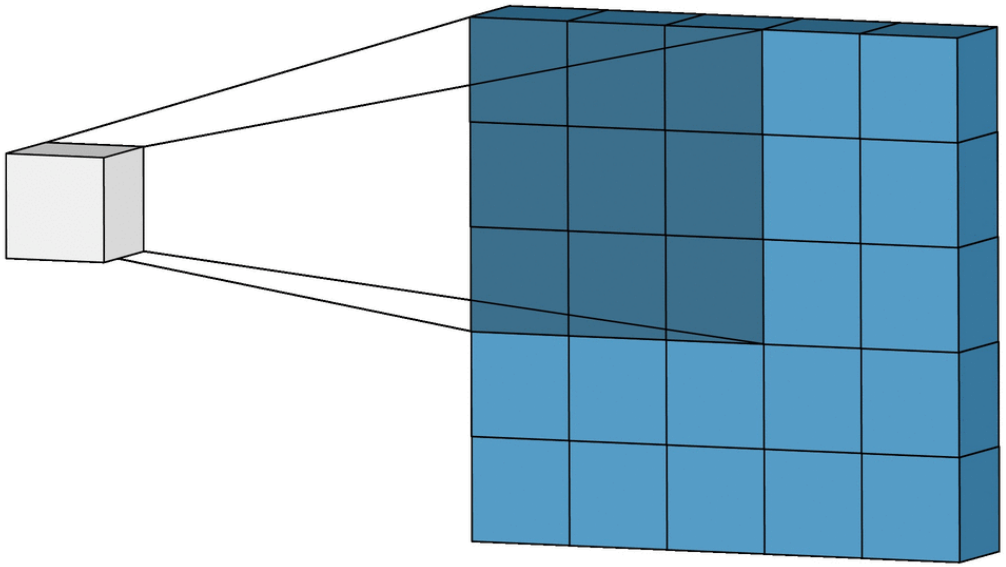
- Key components of the architecture:
  - *Convolutional layers*
  - *Rectified Linear Unit (ReLU for short)*
  - *Pooling layers*
  - *Fully connected layers*



Convolution Filter

- *Main mathematical task*: **Convolution**
  - Application of a **sliding window function** to a **matrix of pixels** representing an image.
  - The sliding function applied to the matrix is called ***kernel*** or ***filter*.**



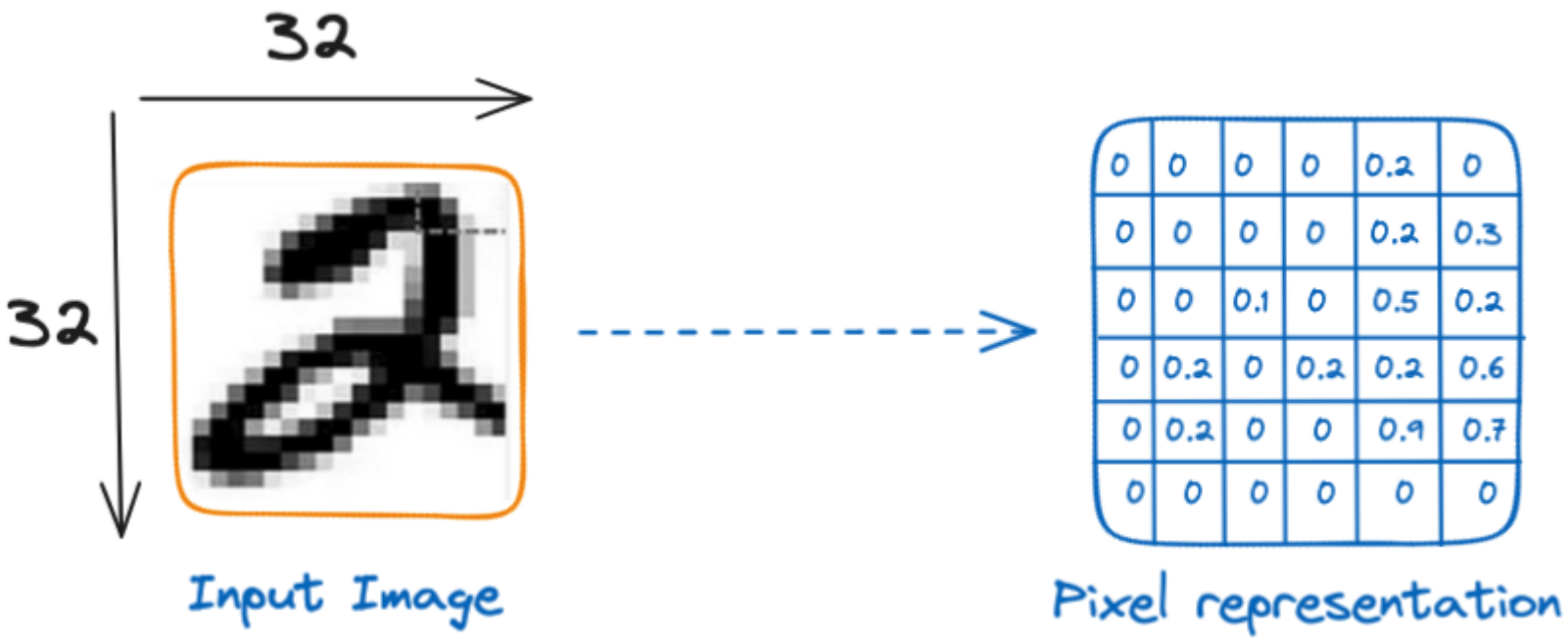


This layer performs a dot product between two matrices, where one matrix is the set of learnable parameters otherwise known as a kernel, and the other matrix is the restricted portion of the receptive field.

Convolution Filter

- **Several filters** of **equal size** are applied
- Each filter is used to recognize a specific pattern from the image, such as
  - Curving of the digits,
  - Edges,
  - Whole shape of the digits, and more.
- Example

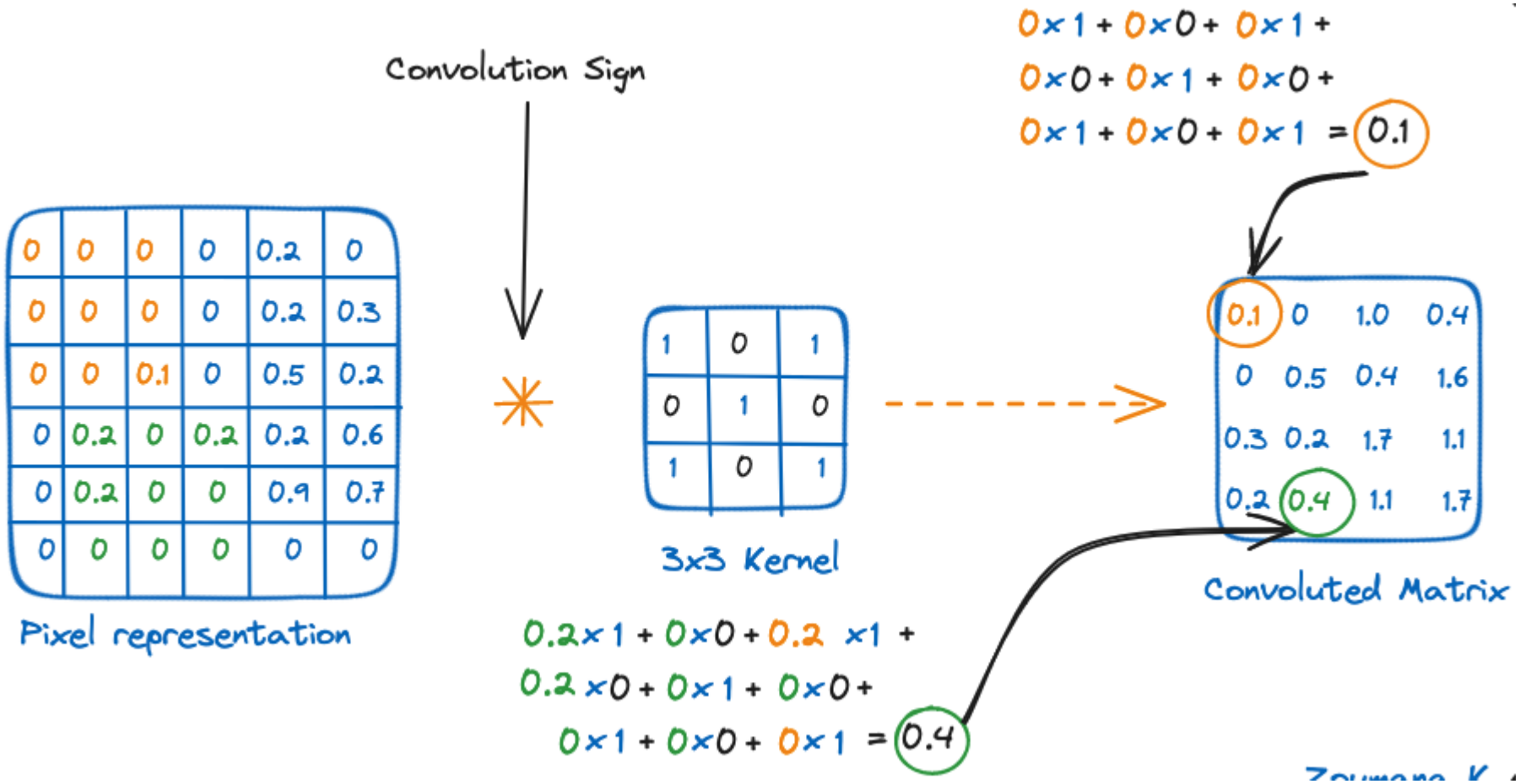
Illustration of the input image and its pixel representation



Convolution Filter

- Example: *Application of convolution using a stride of 1 with a 3 x 3 kernel*

1. **Apply kernel matrix** from the ***top-left corner*** to the ***right***.
2. Perform **element-wise multiplication**.
3. **Sum** the **values** of the products.
4. **Resulting value** corresponds to the **first value** (top-left corner) in **the convoluted matrix.**
5. Move the **kernel down** with **respect** to the **size** of the **sliding window**.
6. Repeat steps 1 to 5 until the image matrix is fully covered.



Convolution Filter

- **Why Convolution filter?**
  - *Parameter sharing:* a feature detector (such as a vertical edge detector) useful in one part of the image.
  - *Sparsity of connections:* Each output value depends only on small number of inputs.
- Division of image into small blocks helps in extracting **locally correlated pixel** values : ***Features***

❑ **For Instance:**

❑ **Averaging** neighboring pixels kernel can be used to ***blur the input image***.

❑ **Subtracting** neighboring kernel is used to ***perform edge detection***.

Pooling Layer

- *Goal of the pooling layer*: *Pull* the *most significant features* from the *convoluted matrix*.
  - Done by applying some *aggregation* operations,
  - Which *reduce the dimension* of the *feature map* (convoluted matrix),
  - hence *reducing the memory* used while training the network.
  - Pooling is also relevant for *mitigating overfitting*.

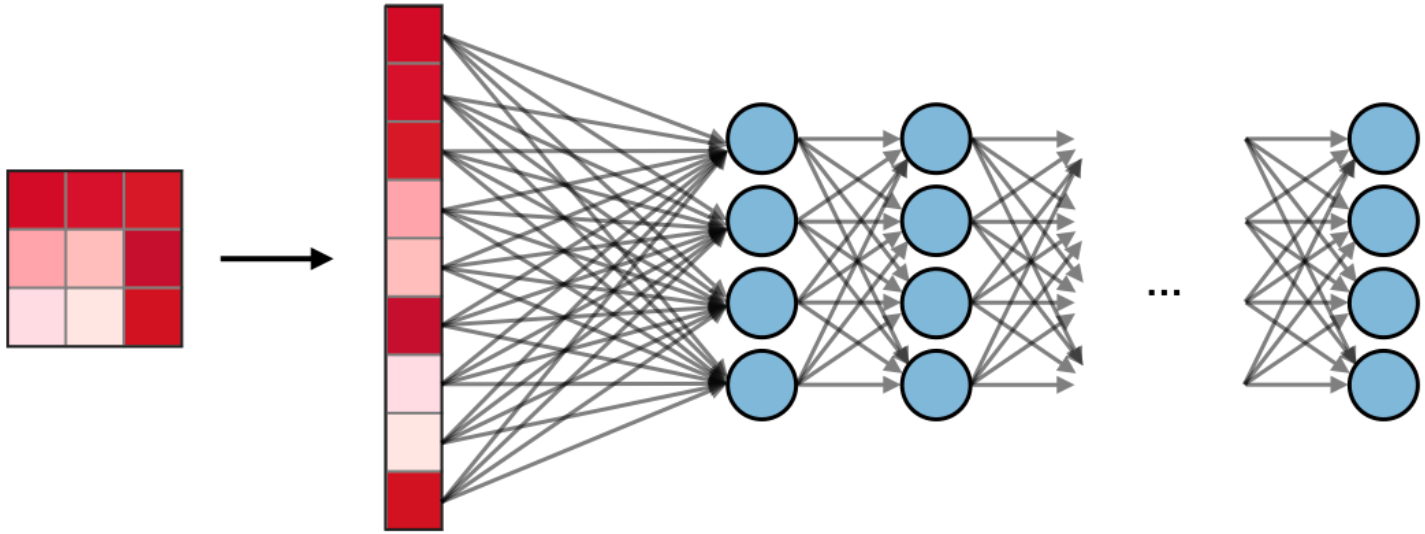
- Most common aggregation functions :
  - *Max pooling*: Maximum value of the feature map
  - *Sum pooling*: *S*um of all the values of the feature map
  - *Average pooling*: Average of all the values.

Pooling Layer

- *Application of a max pooling with a stride of 2 using a 2 x 2 filter*

Fully Connected Layer

- Neurons in this layer have full connectivity with all neurons in the preceding and succeeding layer as seen in regular FCNN.
- This is why it can be computed as usual by a matrix multiplication followed by a bias effect.
- The FC layer helps to map the representation between the input and the output.



Non-Linearity Layers

- Since convolution is a linear operation and images are far from linear,
- Non-linearity layers are often placed directly after the convolutional layer to *introduce non-linearity to the activation map*.
- Several types of non-linear operations, the popular ones being:
  - *Sigmoid*: The sigmoid non-linearity has the mathematical form $\sigma(\kappa) = \frac{1}{1+e^{-\kappa}}$. It takes a real-valued number and "squashes" it into a range between 0 and 1.
  - *Tanh*: Tanh squashes a real-valued number to the range [-1, 1]. Like sigmoid, the activation saturates, but — unlike the sigmoid neurons — its output is zero centered.
  - *ReLU*: Rectified Linear Unit (ReLU) has become very popular in the last few years. It computes the function ƒ(κ)=max (0,κ). In other words, the activation is simply threshold at zero.

Recurrent Neural Networks

- Recurrent neural network or RNN is a deep neural network trained on **sequential** or **time series data** to create a machine learning (ML) model that can **make sequential predictions** or **conclusions based on sequential inputs**.
- **Can solve problems pertaining to:**
  - *Language translation,*
  - *Natural language processing (NLP),*
  - *Sentiment analysis, speech recognition and*
  - *Image captioning.*
- RNNs uses **past information** to **improve** the **performance** of the network on **current and future inputs**.
- RNN architecture characterizes **hidden states** and **loops**.
  - The looping structure allows the network to store past information in the hidden state and operate on sequences.

RNN : Architecture

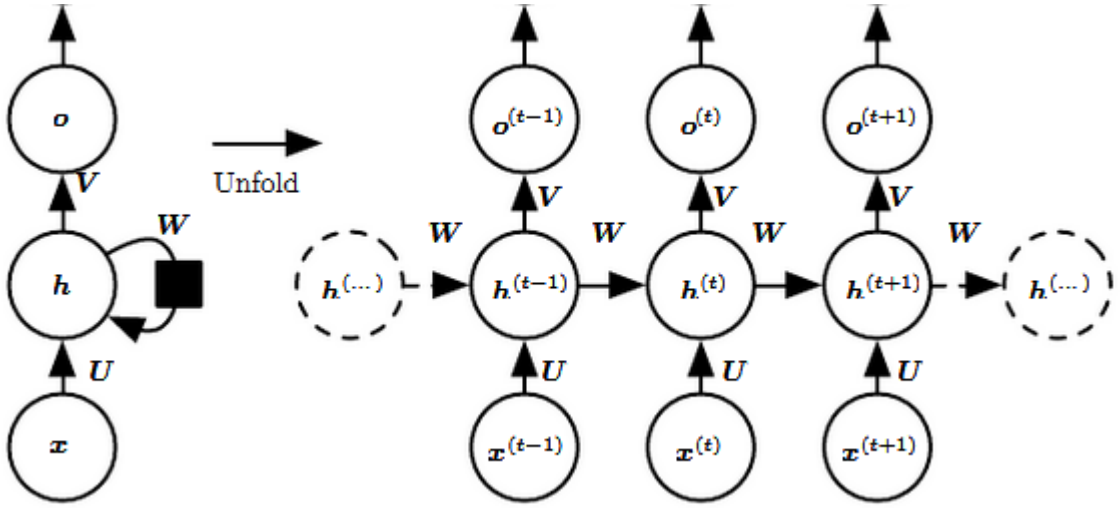
- **Input:** $x(t)$ is taken as the input to the network at time step t. For example, x_1,could be a one-hot vector corresponding to a word of a sentence.

- **Hidden state:** $h(t)$ represents a hidden state at time t and acts as "**memory**" of the network.

- *$h(t)$ is calculated based on the current input and the previous time step's hidden state:*

- $h(t) = f(U\,x(t) + W\,h(t-1))$.

- The function f is taken to be a non-linear transformation such as tanh, ReLU.

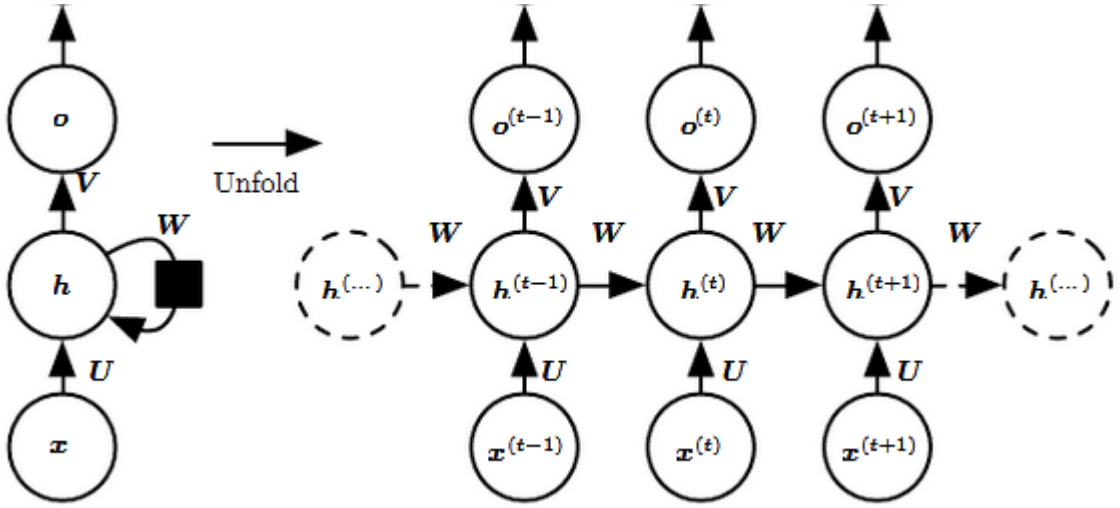


- ❑ **Recurrent connections** are depicted via cyclic edges.

- ❑ We **unfold the RNN over time steps**. Here, ***recurrent edges span adjacent time steps***, while conventional connections are computed synchronously.¶

RNN : Architecture

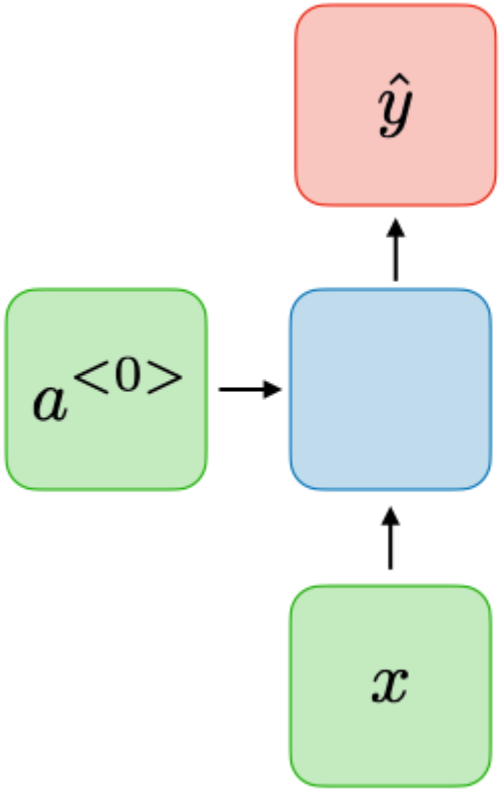
- **Weights**: RNN has *input* to *hidden connections* parameterized by a *weight matrix **U***, *hidden-to-hidden recurrent* connections parameterized by a *weight matrix **W***, and *hidden-to-output connections* parameterized by a *weight matrix **V*** and all these weights (U, V, W) are shared across time.
- **Output**: $o(t)$ illustrates the ***output of the network***.
  - In the figure I just put an arrow after $o(t)$ which is also often subjected to non-linearity, especially when the network contains further layers downstream.



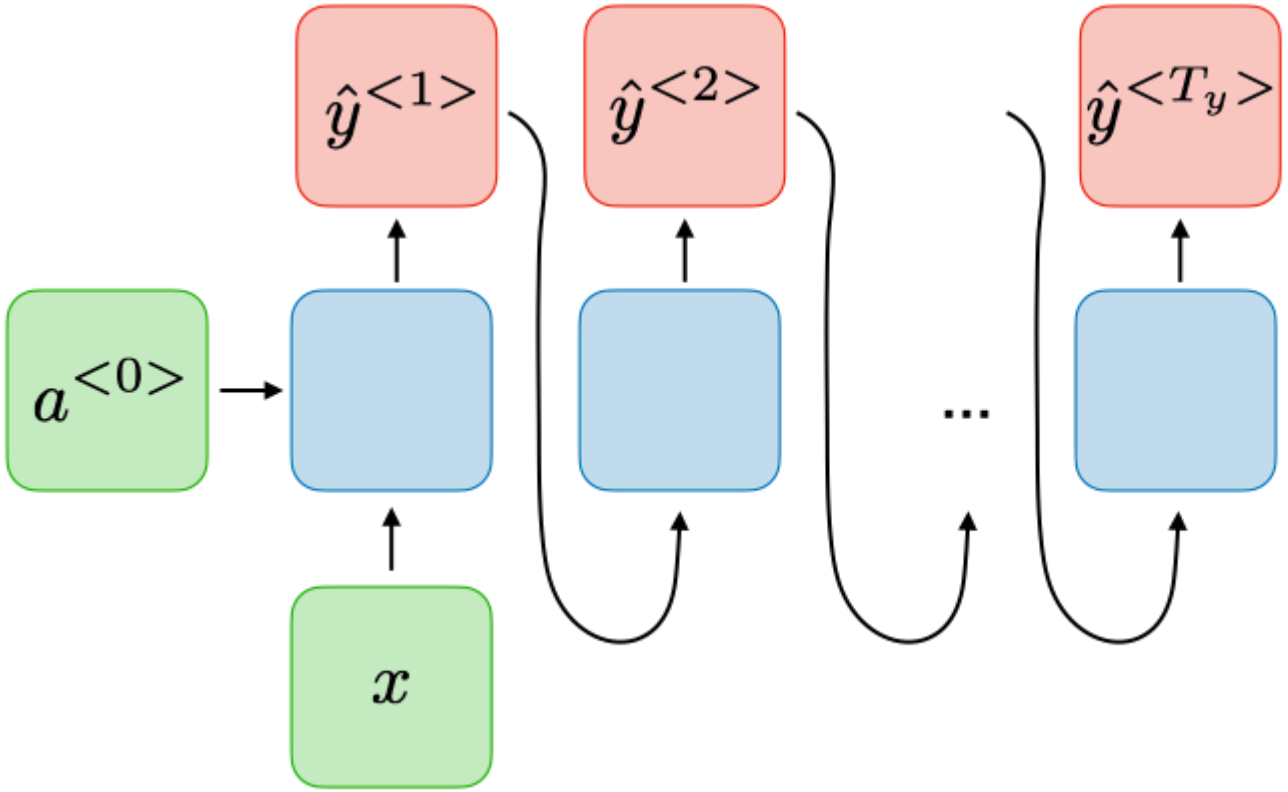
- ❑ **Recurrent connections** are depicted via cyclic edges.
- ❑ We **unfold the RNN over time steps**. Here, ***recurrent edges span adjacent time steps***, while conventional connections are computed synchronously.¶

RNN : Types

- **One-to-one**
- $Tx = Ty = 1$
- Traditional neural networks

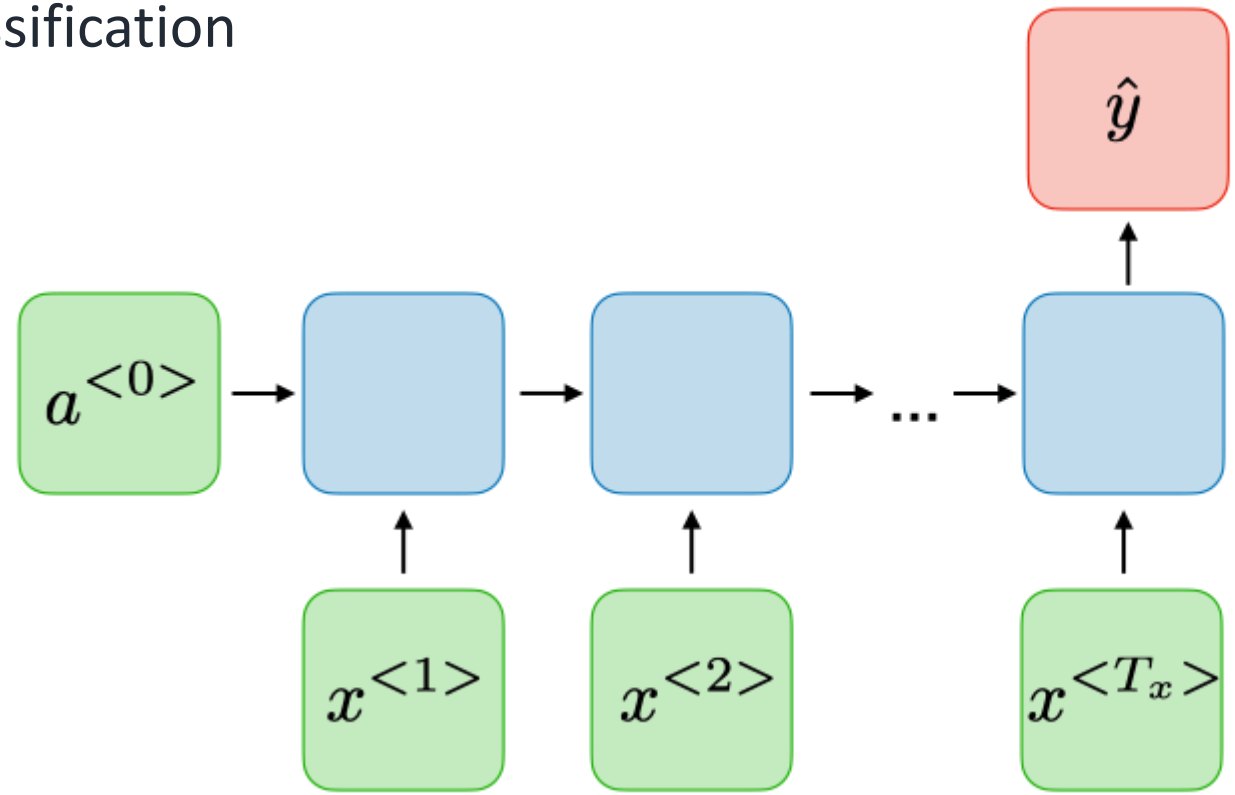


- **One-to-many**
- $Tx = 1, Ty > 1$
- Music generation



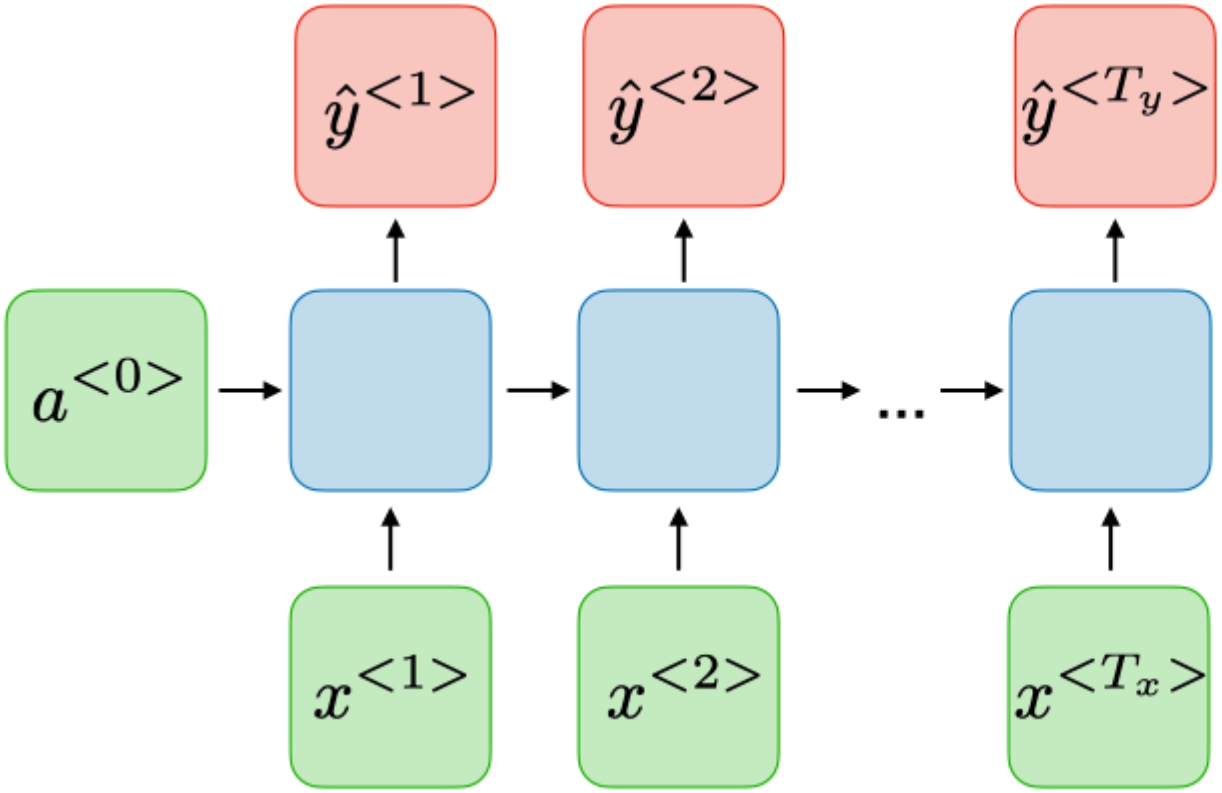
RNN : Types

- **Many-to-one**
- $Tx > 1, Ty = 1$
- Sentiment classification



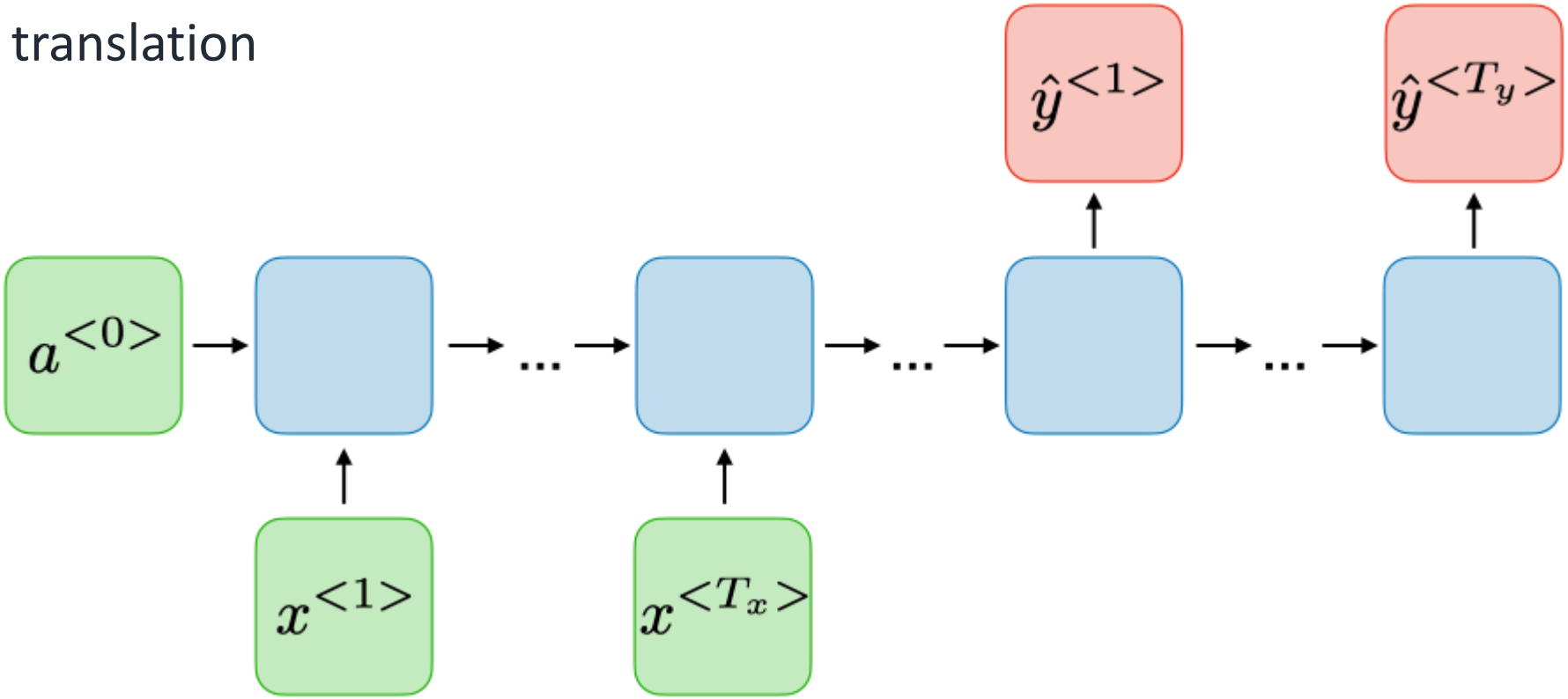
RNN : Types

- **Many-to-Many**
- $Tx = Ty$
- Name entity recognition



RNN : Types

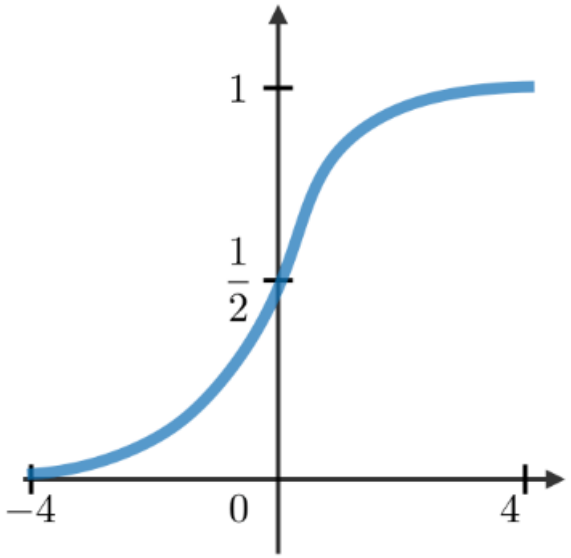
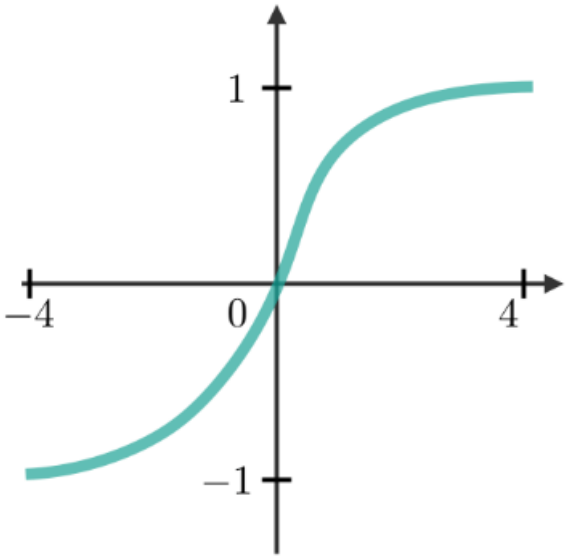
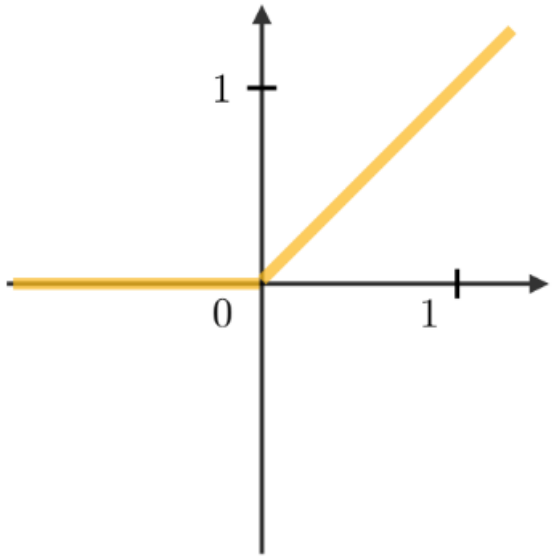
- **Many-to-Many**
- $Tx \neq Ty$
- Machine translation



RNN

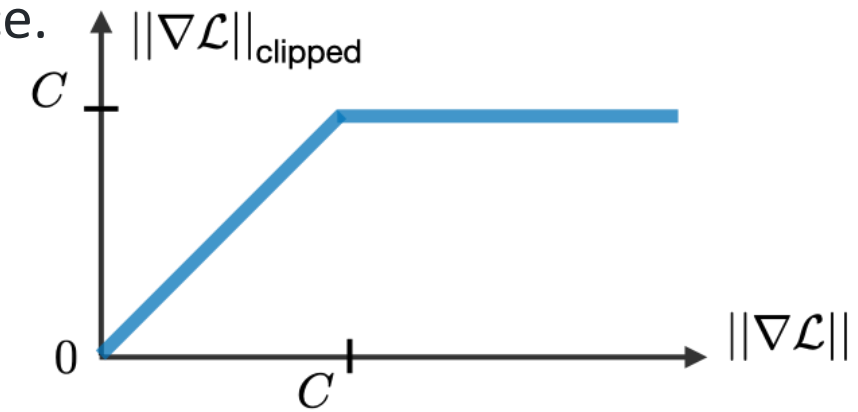
Advantages	Drawbacks
• Possibility of processing input of any length • Model size not increasing with size of input • Computation takes into account historical information • Weights are shared across time	• Computation being slow • Difficulty of accessing information from a long time ago • Cannot consider any future input for the current state

RNN : Activation Function

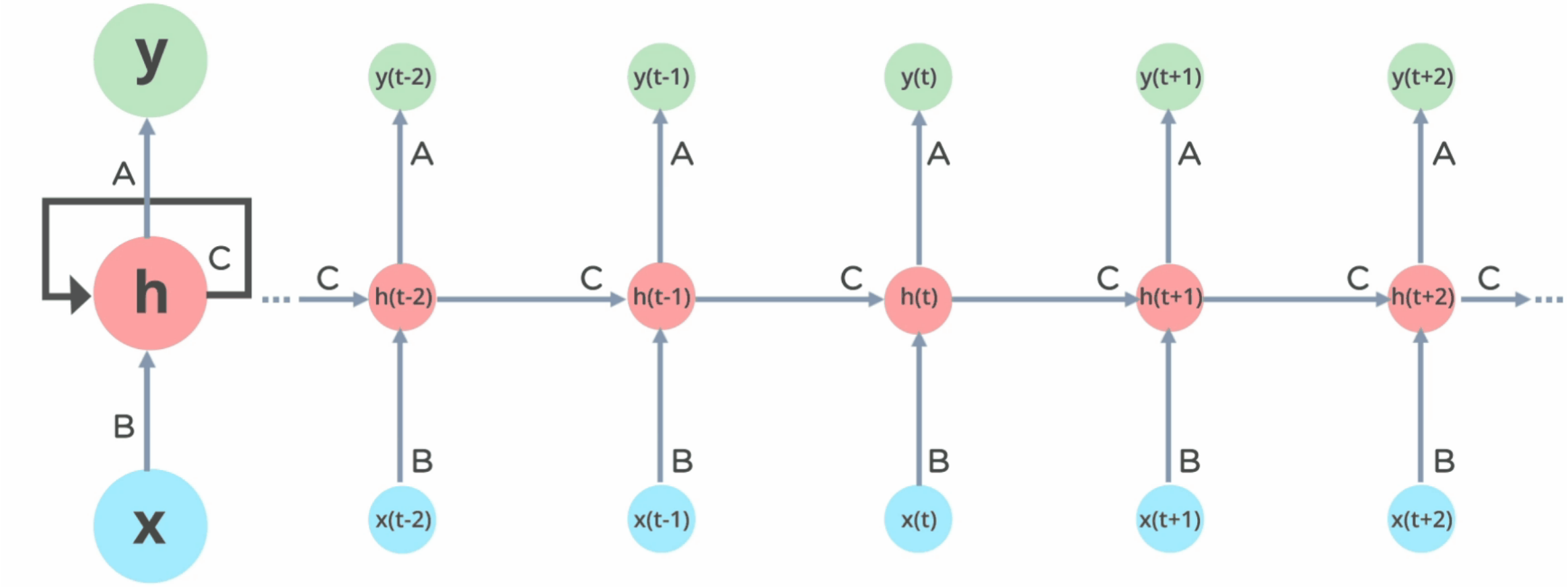
Sigmoid	Tanh	RELU
$g(z) = \frac{1}{1+e^{-z}}$	$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0, z)$
		

RNN : Activation Function

- **Vanishing/exploding gradient**
  - The vanishing and exploding gradient phenomena are often encountered in the context of RNNs.
  - The reason why they happen is that it is **difficult to capture long term dependencies** because of **multiplicative gradient** that can be **exponentially decreasing/increasing** with respect to the **number of layers**.
- **Gradient clipping**
  - It is a technique used to cope with the exploding gradient problem sometimes encountered when performing backpropagation. By **capping the maximum value for the gradient**, this phenomenon is controlled in practice.



RNN : Working


y
A
h
C
B
x
y(t-2)
y(t-1)
y(t)
y(t+1)
y(t+2)
A
C
h(t-2)
h(t-1)
h(t)
h(t+1)
h(t+2)
B
x(t-2)
x(t-1)
x(t)
x(t+1)
x(t+2)

RNN : Gates

- In order to **remedy** the **vanishing gradient problem**, specific gates are used in some types of RNNs and usually have a well-defined purpose.
- They are usually noted Γ and are equal to:

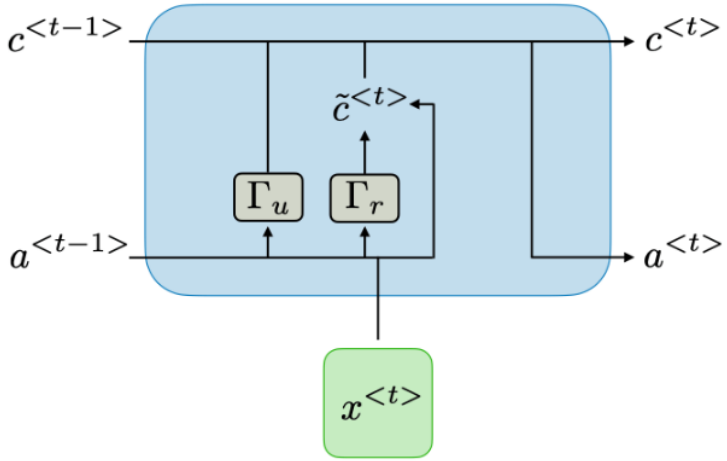
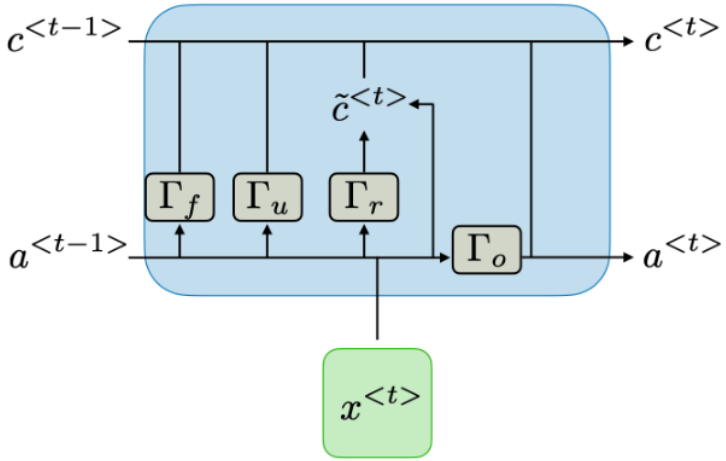
$$\Gamma = \sigma(Wx^{<t>} + Ua^{<t-1>} + b)$$

where ***W, U, b*** are coefficients specific to the gate and ***σ*** is the sigmoid function.

Type of gate	Role	Used in
Update gate Γ_u	How much past should matter now?	GRU, LSTM
Relevance gate Γ_r	Drop previous information?	GRU, LSTM
Forget gate Γ_f	Erase a cell or not?	LSTM
Output gate Γ_o	How much to reveal of a cell?	LSTM

RNN : GRU / LSTM

- ***Gated Recurrent Unit (GRU)*** and ***Long Short-Term Memory units (LSTM)*** deal with the vanishing gradient problem encountered by traditional RNNs, with LSTM being a generalization of GRU.

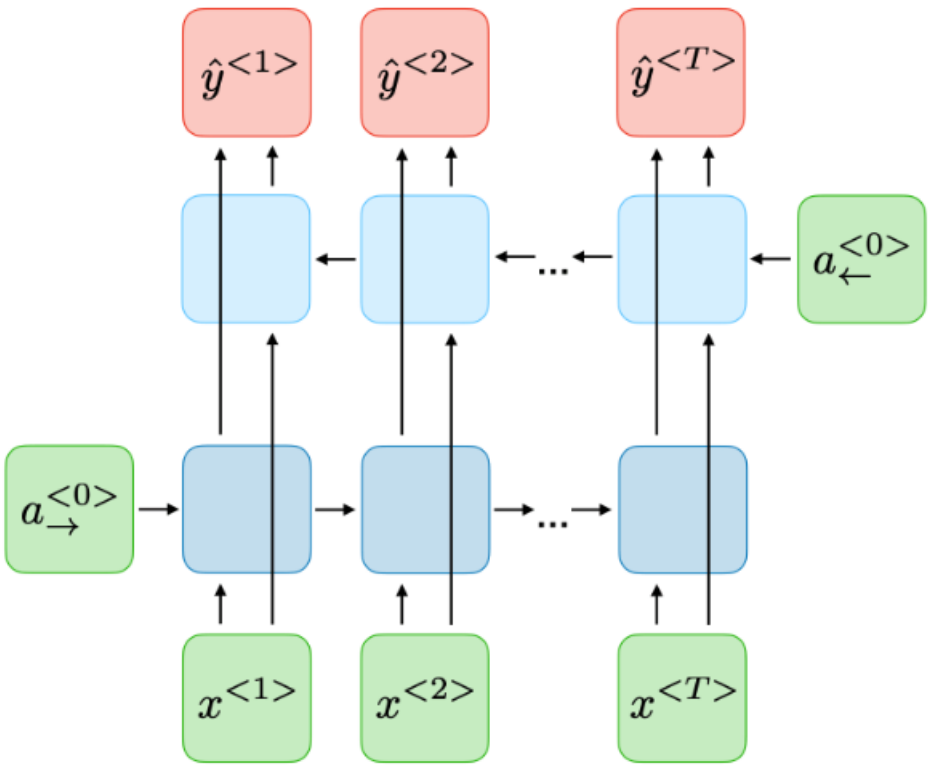
Characterization	**Gated Recurrent Unit** (GRU)	**Long Short-Term Memory** (LSTM)
$\tilde{c}^{<t>}$	$\tanh(W_c[\Gamma_r \star a^{<t-1>}, x^{<t>}] + b_c)$	$\tanh(W_c[\Gamma_r \star a^{<t-1>}, x^{<t>}] + b_c)$
$c^{<t>}$	$\Gamma_u \star \tilde{c}^{<t>} + (1 - \Gamma_u) \star c^{<t-1>}$	$\Gamma_u \star \tilde{c}^{<t>} + \Gamma_f \star c^{<t-1>}$
$a^{<t>}$	$c^{<t>}$	$\Gamma_o \star c^{<t>}$
Dependencies		

Remark: the sign $\star$ denotes the element-wise multiplication between two vectors.

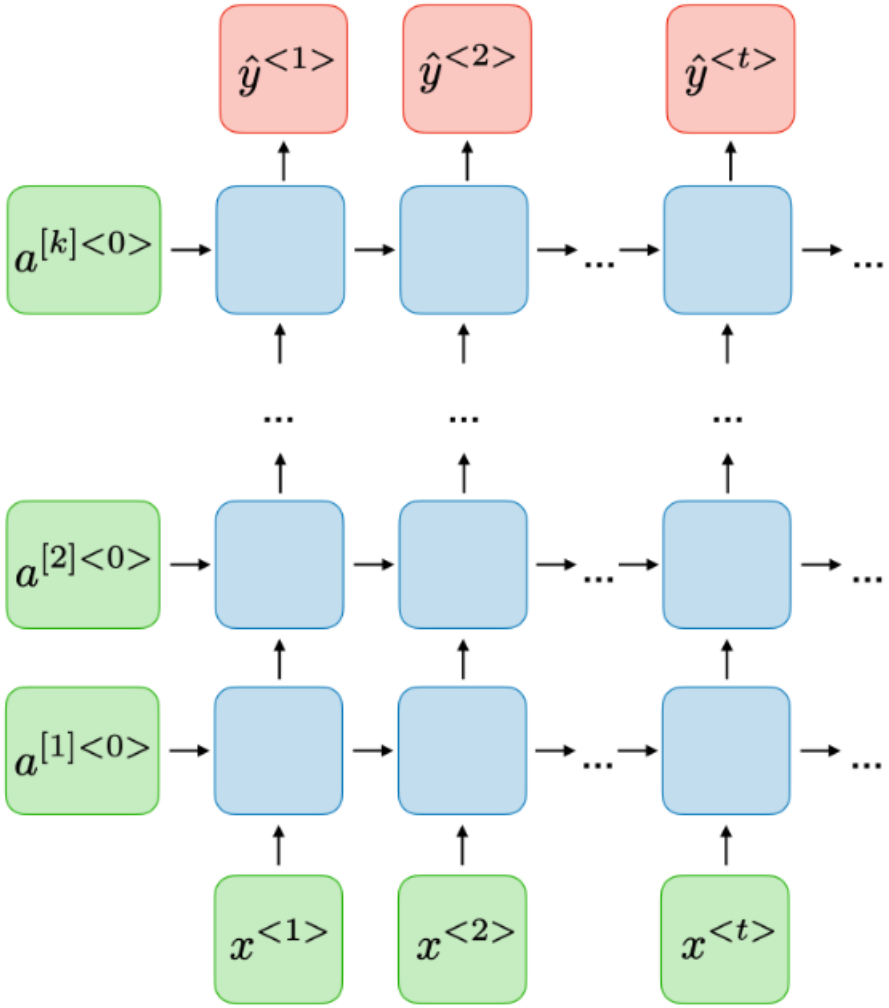
RNN : Variants

- *Ga*
  gra

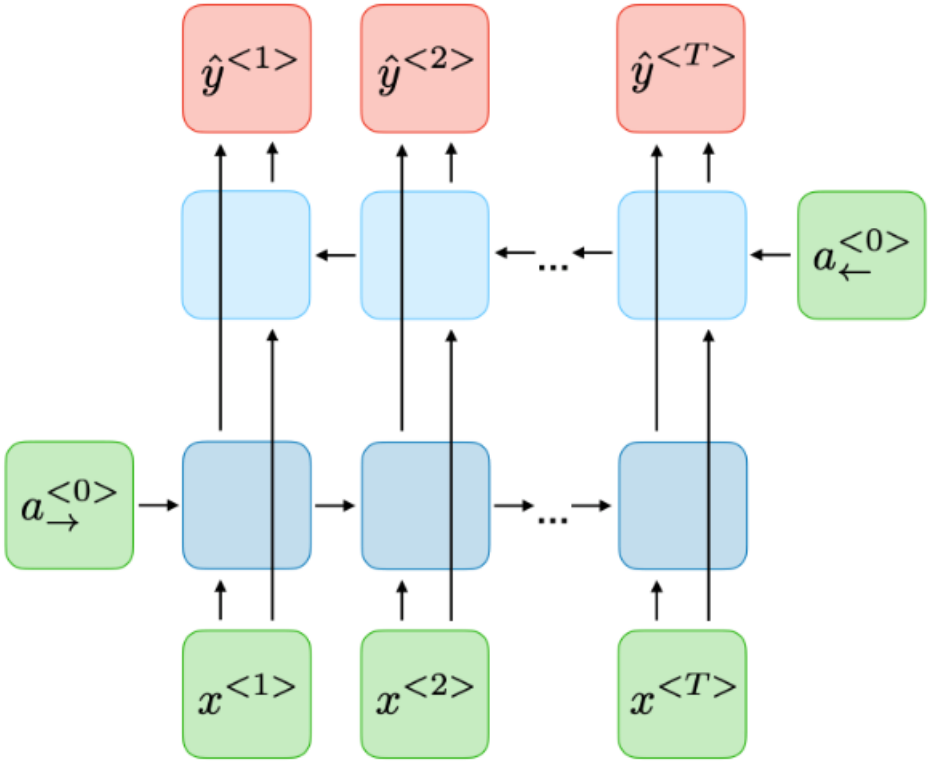
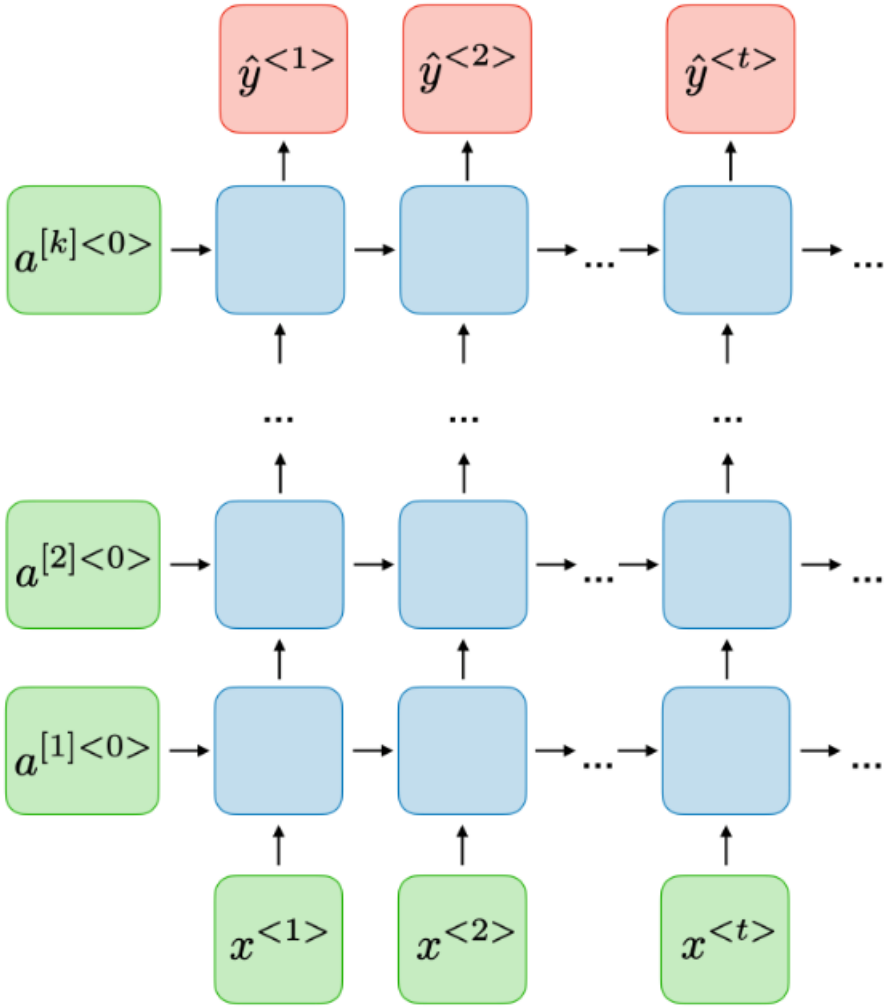
Bidirectional (BRNN)



Deep (DRNN)



RNN : Variants

Bidirectional (BRNN)	**Deep** (DRNN)
	

RNN : Learning word representation

* Note V the vocabulary and $|V|$ its size.

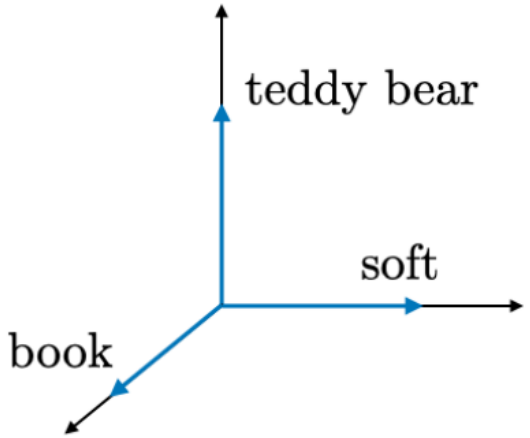
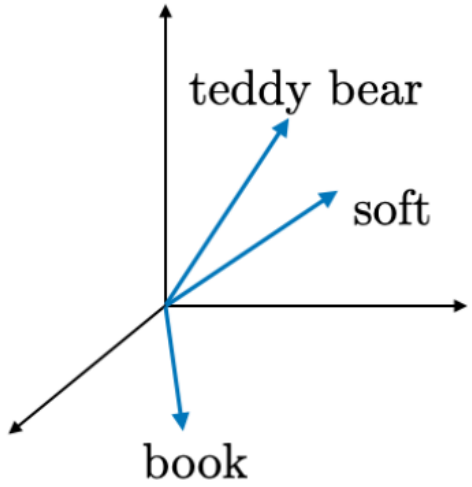
Representation techniques: The two main ways of representing words are summed up in the table below:

Embedding Matrix

For a given **word w**, the **embedding matrix E** is a matrix that maps its **1-hot representation o_w** to its **embedding e_w** as follows:

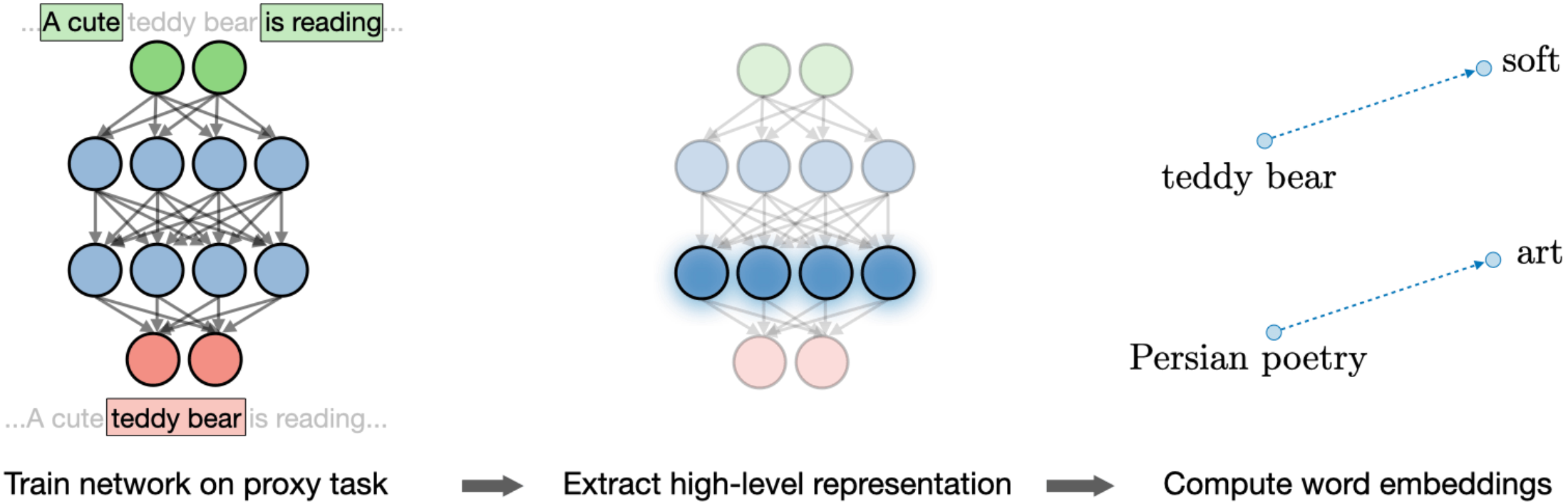
$$e_w = E o_w$$

Remark: learning the embedding matrix can be done using target/context likelihood models.

1-hot representation	Word embedding
	
• Noted o_w • Naive approach, no similarity information	• Noted e_w • Takes into account words similarity

RNN : Word2Vec

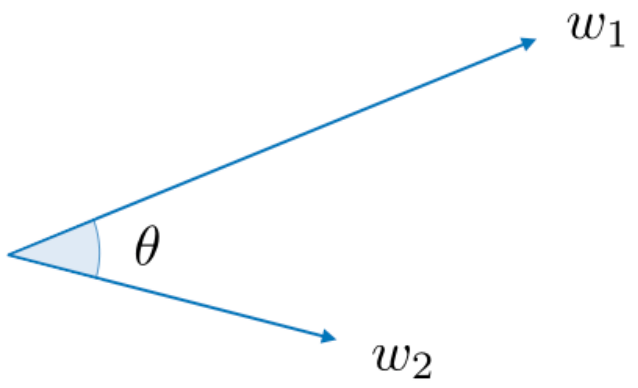
- **Word2vec** is a framework aimed at ***learning word embeddings*** by ***estimating the likelihood*** that a given ***word is surrounded by other words***. Popular models include skip-gram, negative sampling and CBOW.



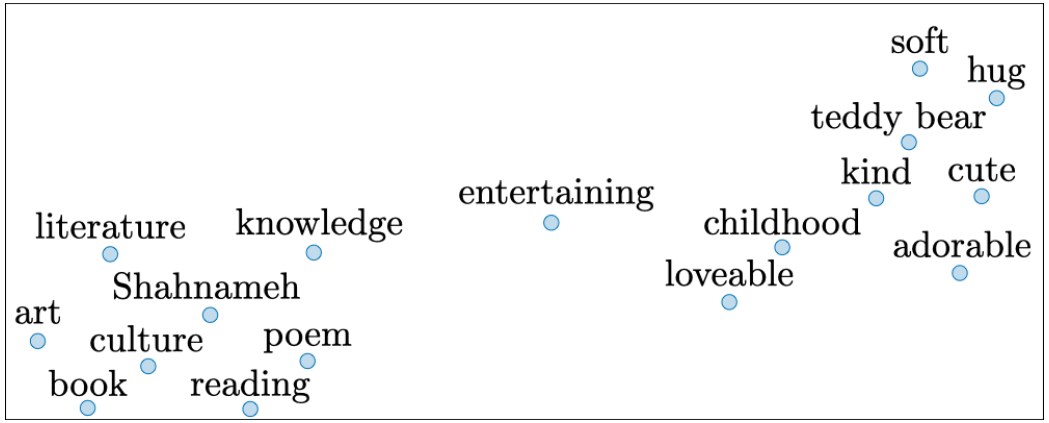
RNN : Similarity

- **Cosine similarity:** The cosine similarity between words w_1 and w_2 is expressed as follows:

$$\text{similarity} = \frac{w_1 \cdot w_2}{||w_1||\ ||w_2||} = \cos(\theta)$$



- ***t*-SNE**: (*t*-distributed Stochastic Neighbor Embedding) is a technique aimed at *reducing high-dimensional embeddings* into a *lower dimensional space*. In practice, it is commonly used to visualize word vectors in the 2D space.



RNN : Machine Translation

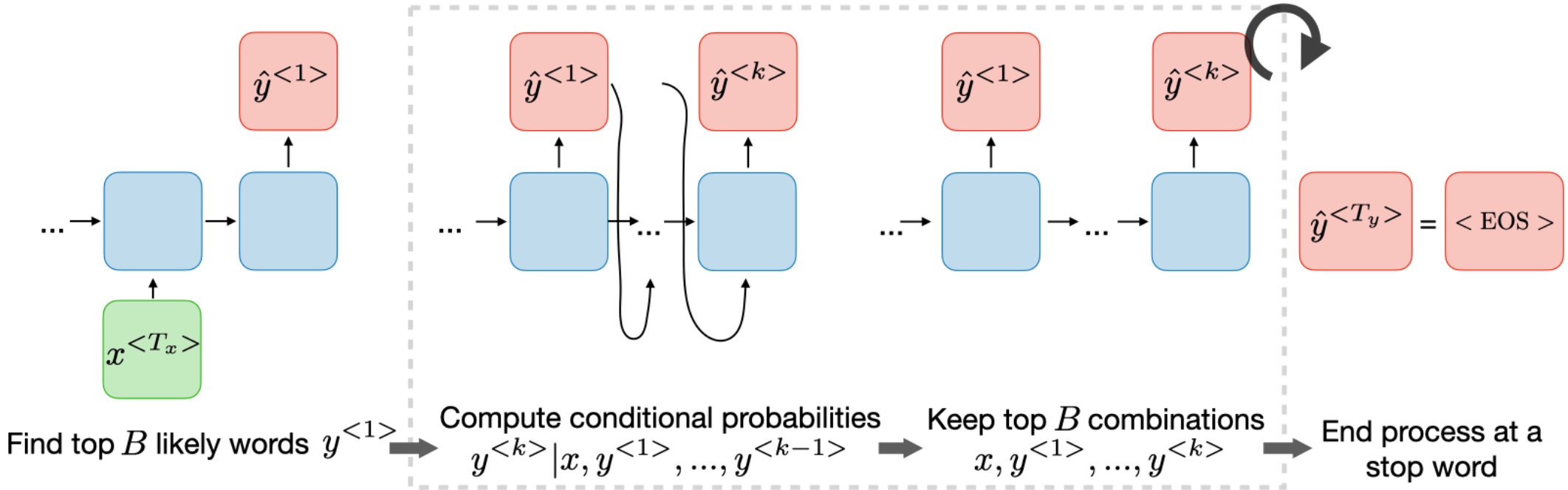
- A machine translation model is **similar** to a **language model** except it has an **encoder network placed before**.
- For this reason, it is sometimes referred as a **conditional language model**.
- The goal is to find a sentence *y* such that:

$$y = \underset{y^{<1>},\ldots,y^{<T_y>}}{\arg\max}\ P(y^{<1>},\ldots,y^{<T_y>}|x)$$

RNN : Machine Translation

Beam search: It is a heuristic search algorithm used in machine translation and speech recognition to find the likeliest sentence *y* given an input *x*.

- Step 1: Find top B likely words $y^{<1>}$
- Step 2: Compute conditional probabilities $y^{<k>}|x, y^{<1>}, ..., y^{<k-1>}$
- Step 3: Keep top B combinations $x, y^{<1>}, ..., y^{<k>}$



RNN : Machine Translation

- **Beam width :**
  - The beam width *B* is a parameter for beam search.
  - **Large values of *B* yield to better result** but with **slower performance** and **increased memory**.
  - **Small values of *B*** lead to **worse results** but is **less computationally intensive**.
  - A standard value for *B* is around 10.

Error analysis
When obtaining a predicted translation $\hat{y}$ that is bad, one can wonder why we did not get a good translation $y*$ by performing the following error analysis:

Case	$P(y^*\|x) > P(\hat{y}\|x)$	$P(y^*\|x) \leqslant P(\hat{y}\|x)$
Root cause	Beam search faulty	RNN faulty
Remedies	Increase beam width	• Try different architecture • Regularize • Get more data

RNN : Machine Translation

Bleu score

The bilingual evaluation understudy (bleu) score quantifies **how good a machine translation is** by computing a **similarity score** based on ***n*-gram precision**.
It is defined as follows:

$$\boxed{\text{bleu score} = \exp\left(\frac{1}{n}\sum_{k=1}^{n} p_k\right)}$$

where p_n is the bleu score on n-gram only defined as follows:

$$p_n = \frac{\sum_{\text{n-gram}\in\hat{y}} \text{count}_{\text{clip}}(\text{n-gram})}{\sum_{\text{n-gram}\in\hat{y}} \text{count}(\text{n-gram})}$$

Remark: a brevity penalty may be applied to short predicted translations to prevent an artificially inflated bleu score.

RNN : Attention

Attention model: This model **allows an RNN** to **pay attention to specific parts** of the **input** that is **considered** as **being important**, which **improves the performance** of the resulting model in practice. By noting $\alpha < t, t' >$ the amount of attention that the output $y < t >$ should pay to the activation $a < t' >$ and $c < t >$ the context at time t, we have:

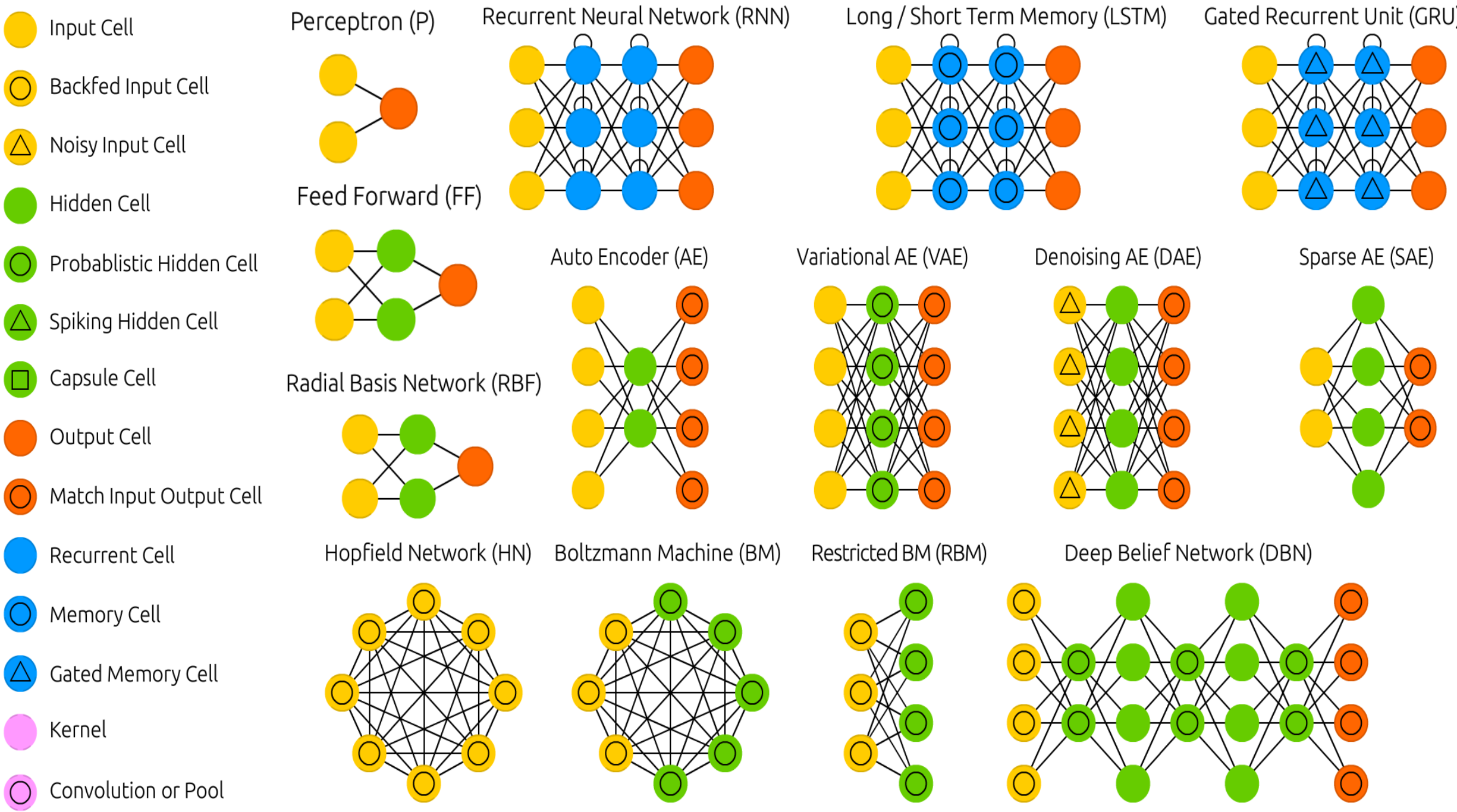
$$\boxed{c^{<t>} = \sum_{t'} \alpha^{<t,t'>} a^{<t'>}} \quad \text{with} \quad \sum_{t'} \alpha^{<t,t'>} = 1$$

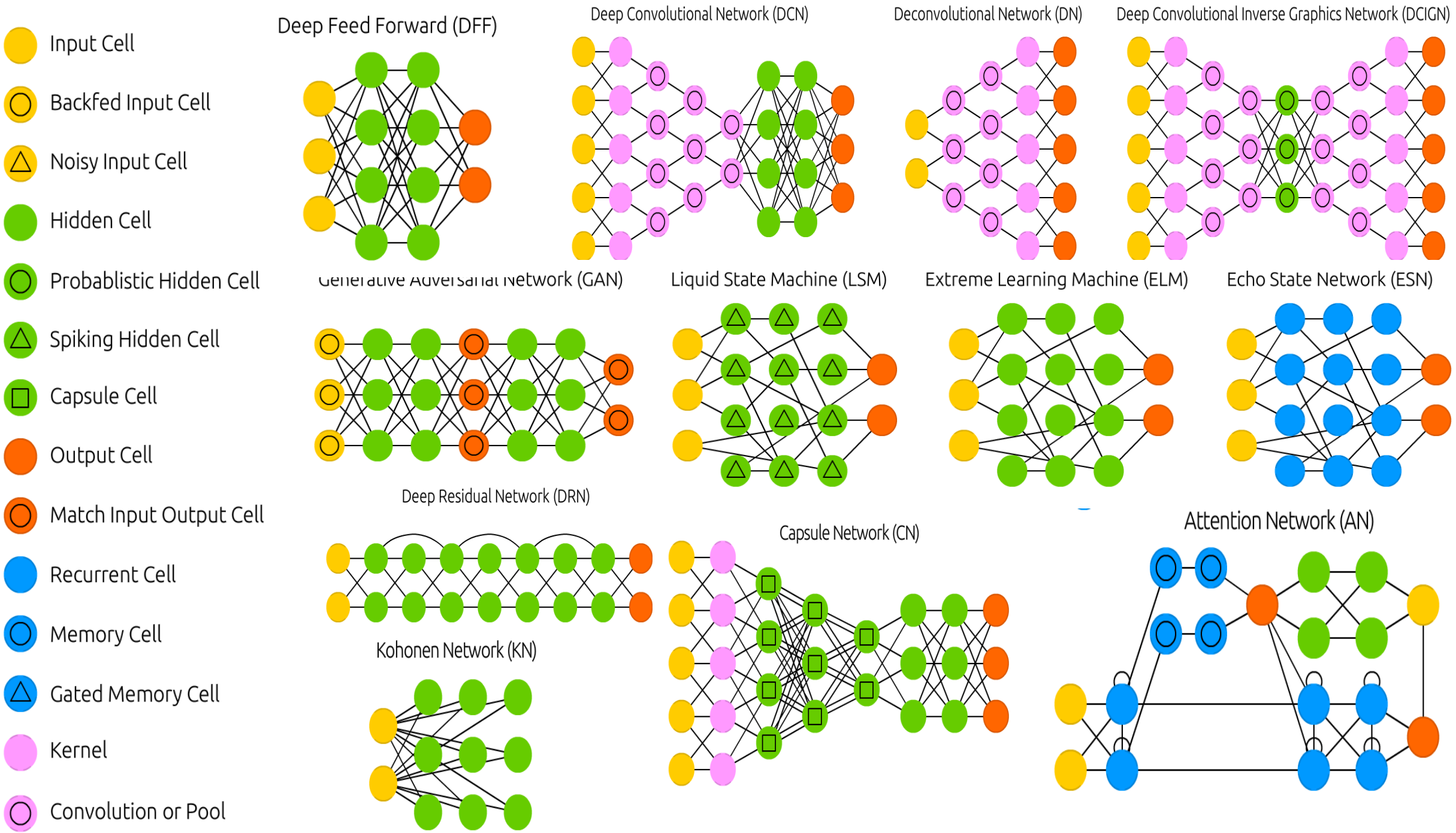
Attention weight: The amount of attention that the output *y*<*t*> should pay to the activation *a*<*t'*> is given by *α*<*t*,*t'*> computed as follows:

$$\alpha^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum_{t''=1}^{T_x} \exp(e^{<t,t''>})}$$

A cute teddy bear is reading Persian literature

A cute teddy bear is reading Persian literature


Input Cell
Backfed Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Capsule Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Gated Memory Cell
Kernel
Convolution or Pool
Perceptron (P)
Feed Forward (FF)
Radial Basis Network (RBF)
Recurrent Neural Network (RNN)
Long / Short Term Memory (LSTM)
Gated Recurrent Unit (GRU)
Auto Encoder (AE)
Variational AE (VAE)
Denoising AE (DAE)
Sparse AE (SAE)
Hopfield Network (HN)
Boltzmann Machine (BM)
Restricted BM (RBM)
Deep Belief Network (DBN)


Input Cell
Backfed Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Capsule Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Gated Memory Cell
Kernel
Convolution or Pool
Deep Feed Forward (DFF)
Deep Convolutional Network (DCN)
Deconvolutional Network (DN)
Deep Convolutional Inverse Graphics Network (DCIGN)
Generative Adversarial Network (GAN)
Liquid State Machine (LSM)
Extreme Learning Machine (ELM)
Echo State Network (ESN)
Deep Residual Network (DRN)
Kohonen Network (KN)
Capsule Network (CN)
Attention Network (AN)

Reference:

- https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks
- https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks