E.T.S.E.T.

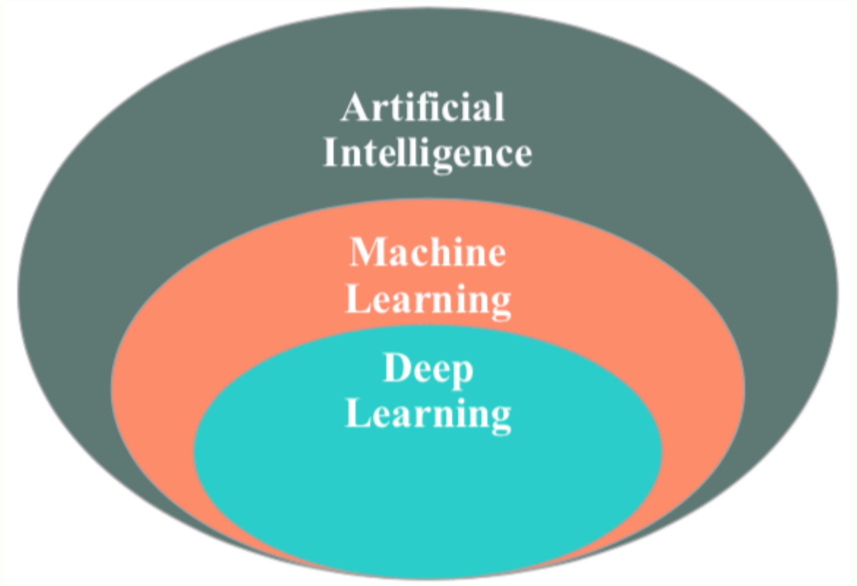
Universidade
deVigo

Reinforcement Learning & Artificial Neural Networks

Author: Juan Carlos Burguillo Rial
Departament Telematic Engineering
Universidade de Vigo
J.C.Burguillo@det.uvigo.es
http://www.det.uvigo.es/~jrial

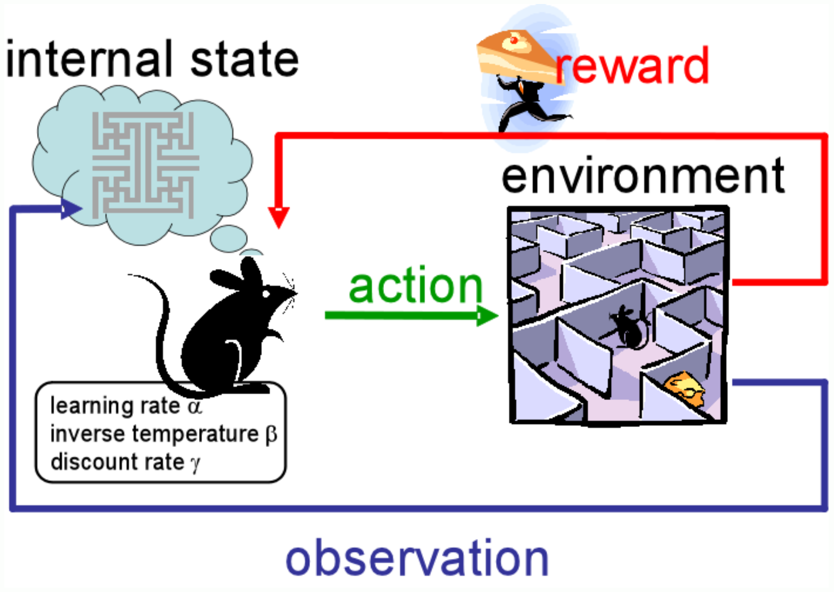
5. AI vs ML vs DL

- Machine learning is a type of artificial intelligence (AI) where computers can essentially learn concepts on their own without being programmed.
- Deep Learning refers to ML algorithms that train deep neural networks to solve problems like: prediction, regression, classification, NLP, etc.



5. Reinforcement Learning (RL)

- Executing an action in a specific state provides the agent with a reward (a real or natural number). The goal of the agent is to maximize its reward.
- It works learning which action is optimal for each state, in the sense of the expected reward for such action in that state.

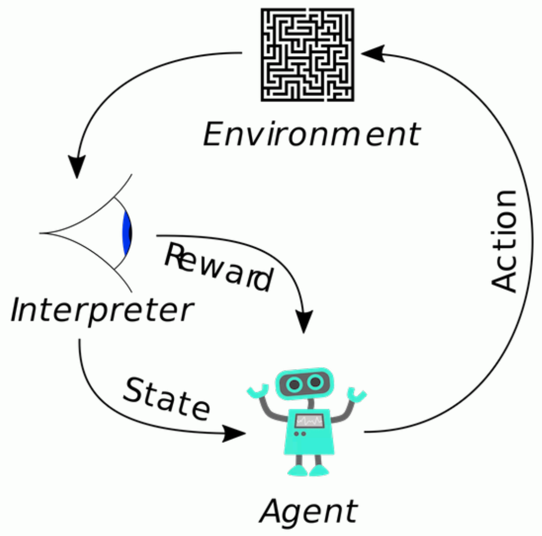


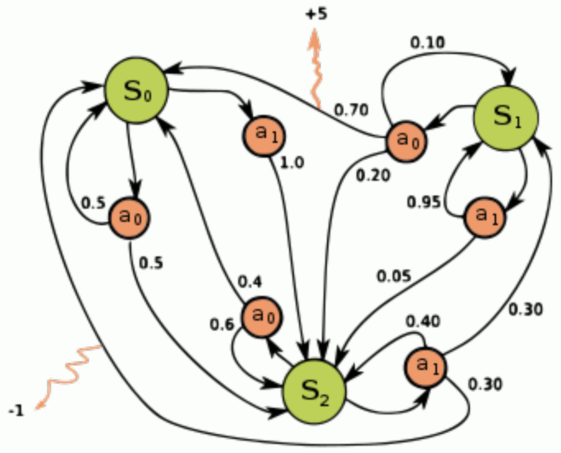
5. RL & MDP

- A **Markov Decision Process** (MDP, from the Russian mathematician Andrey Markov) provides a mathematical framework for modeling decision making in situations where outcomes are partly random, and partly under the control of a decision maker.

- MDPs are useful for studying optimization problems solved via dynamic programming and **reinforcement learning**.

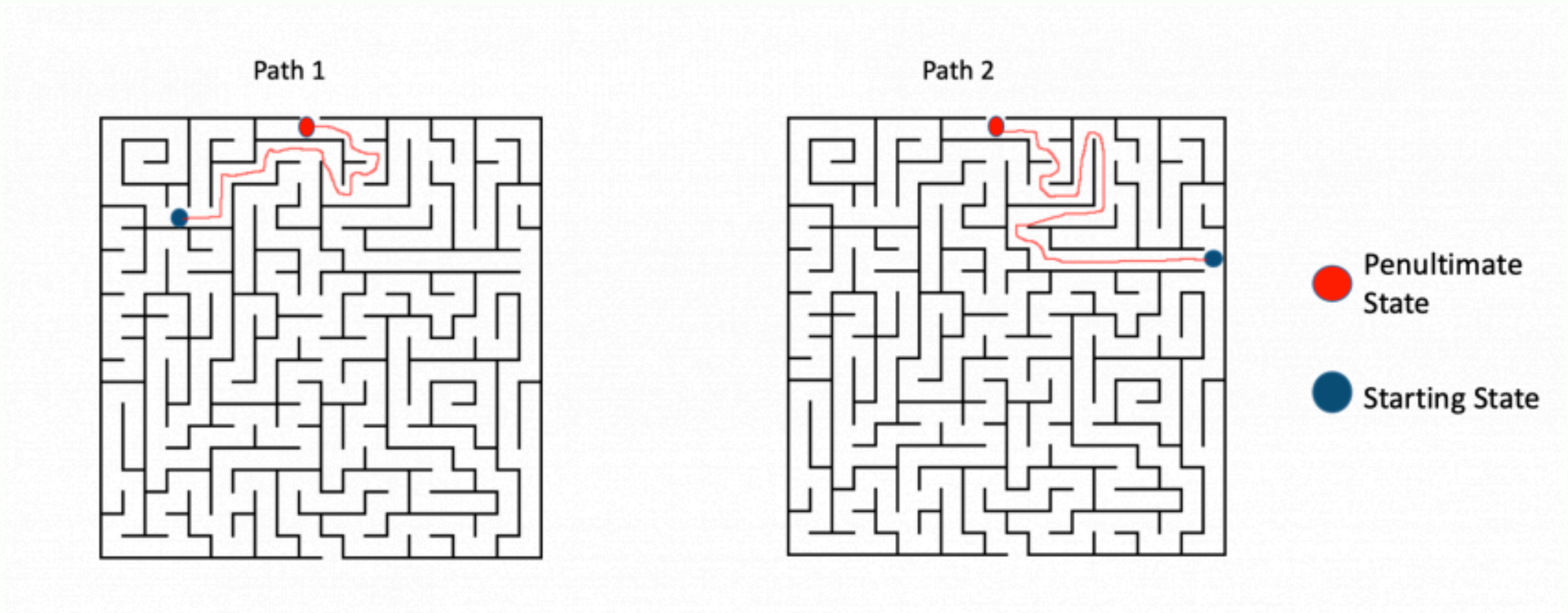
- At each time step, the process is in some state (s), and the decision maker may choose any action (a). Then the process responds at the next time step by moving into a new state (s'), and giving the decision maker a reward $R_a(s,s')$.





5. RL & MDP (2)

- The probability that the process moves into a new state s' is influenced by the chosen action a, and given by the state transition function P_a(s.s').
- The next state (s') depends on the current state (s) and the decision maker's action (a). But given (s) and (a), it is independent of all previous states and actions, i.e., it has no memory → **Markov property**.



5. RL: Learning Automata (LA)

- Learning Automata is a simple RL algorithm that tries to estimate the probability for every action that the state has available. The model consists of an agent with a set of states S and a set of actions per state A. By performing an action a in A, the agent moves from state to state.
- All action probabilities are updated using two simple equations:
  - P_{t+1} (Successful_action) = P_t (Successful_action) + α . (1 – P_t (Successful_action))
  - P_{t+1} (Other_action) = (1 – α) . P_t (Other_action)
- Where α in [0,1] is a small learning factor that should decrease over time.
- The first equation is used to reinforce the best possible action in that state. At the same time, we apply the 2nd equation to the rest of the actions in such state, decreasing their probability.
- Next time, the agent will chose its new action using the updated probabilities.

5. RL: Quality Learning (QL)

- Q-Learning is similar to Learning Automata, but **Q-Learning guarantees convergence**, as the goal of the agent is to **maximize its total reward**, not just the immediate reward resulting from the present action-state pair.

- This is achieved by **learning which action is optimal for each state**, in the sense of the expected value of the total reward over all future steps starting from the current one.

- Therefore the algorithm has a function which calculates the **Quality of a state-action** combination.

$$Q : S \times A \to \mathbb{R}$$

- Before learning has started, Q returns an arbitrary fixed value, chosen by the designer.

5. RL: Q-Learning

- Each time the agent selects an action, it gets a reward and moves to a new state that depends on: the previous state and the selected action.

- The core of the algorithm is a value iteration update. It assumes the old value and makes a correction based on the new information.

$$R = \begin{array}{c|cccccc} \text{State} \backslash \text{Action} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & -1 & -1 & -1 & -1 & 0 & -1 \\ 1 & -1 & -1 & -1 & 0 & -1 & 100 \\ 2 & -1 & -1 & -1 & 0 & -1 & -1 \\ 3 & -1 & 0 & 0 & -1 & 0 & -1 \\ 4 & 0 & -1 & -1 & 0 & -1 & 100 \\ 5 & -1 & 0 & -1 & -1 & 0 & 100 \end{array}$$

$$Q_{t+1}(s_t,a_t) = \underbrace{Q_t(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha_t(s_t,a_t)}_{\text{learning rate}} \times \left[\overbrace{\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q_t(s_{t+1},a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q_t(s_t,a_t)}_{\text{old value}} \right]$$

$\alpha_t(s, a)$ with $(0 < \alpha \leq 1)$ is the **learning rate** (usually the same for all pairs)

R_{t+1} is the **reward** observed after performing a_t in s_t

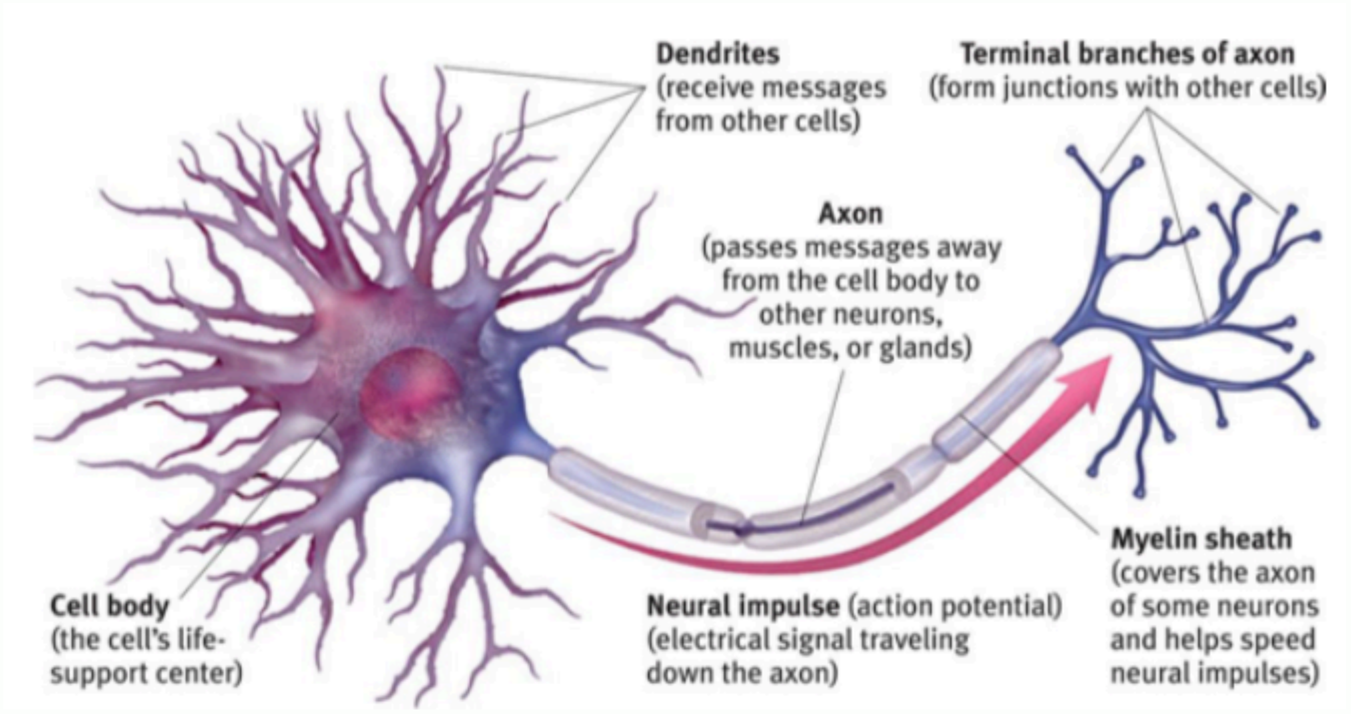
γ with $(0 \leq \gamma \leq 1)$ is the **discount factor** that weights future rewards.

5. Reinforcement Learning Issues

- As said, Q-learning usually handles a matrix, with as many rows as possible states, and as many columns as possible actions.
- **Unfortunately**, the number of possible states in any modern game or real-world environment is often unmanageable. Moreover, if any state variable can take values in a continuous range, discretization is a must; increasing the matrix size.
- Therefore, tables are not feasible for most real world problems, so we need some way to take a description of a state (not a explicit representation), and select actions without a table.
- **Neural networks** help to manage these issues, as they provide function approximations that can:
  a) work with real numbers natively, and
  b) interpolate from non-exhaustive observations.

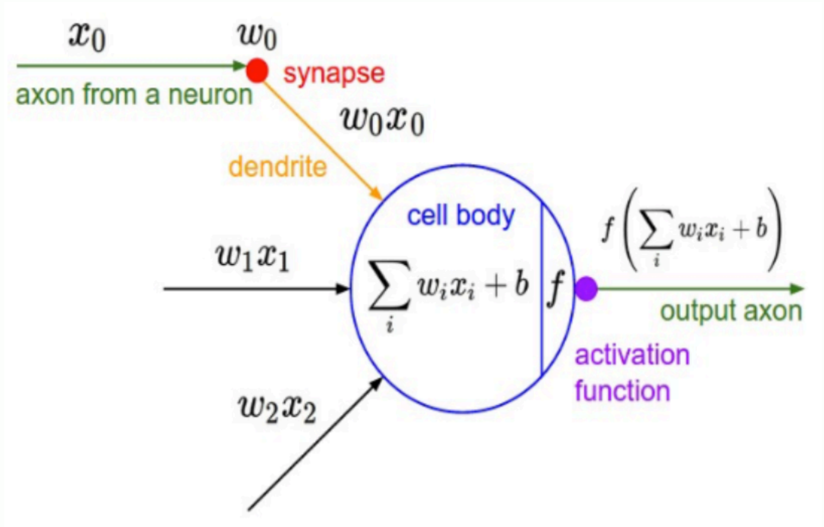
5. Neurons and the Brain

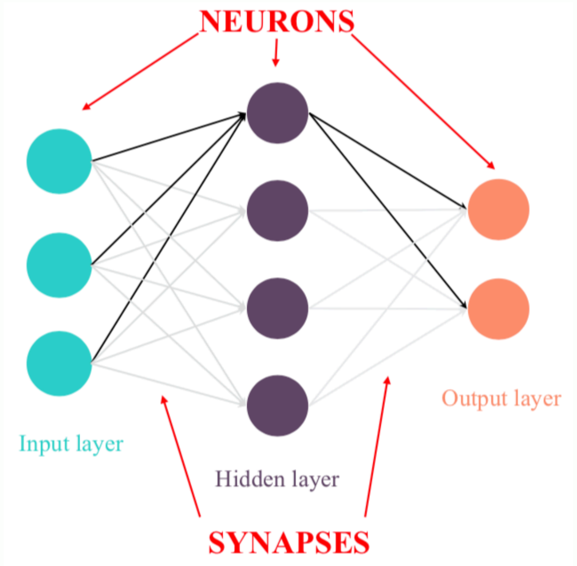
- The neuron is the basic computational unit of the brain (≈ 8 x 10^{10} neurons).
- Each neuron is connected to other 7.000 neurons (≈ 10^{14} synapses), receiving input signals from dendrites, and producing output signals along the axon.
- Synaptic weights are learnable to control the influence weight.


Dendrites (receive messages from other cells)
Terminal branches of axon (form junctions with other cells)
Axon (passes messages away from the cell body to other neurons, muscles, or glands)
Myelin sheath (covers the axon of some neurons and helps speed neural impulses)
Cell body (the cell's life-support center)
Neural impulse (action potential) (electrical signal traveling down the axon)

5. Artificial Neurons

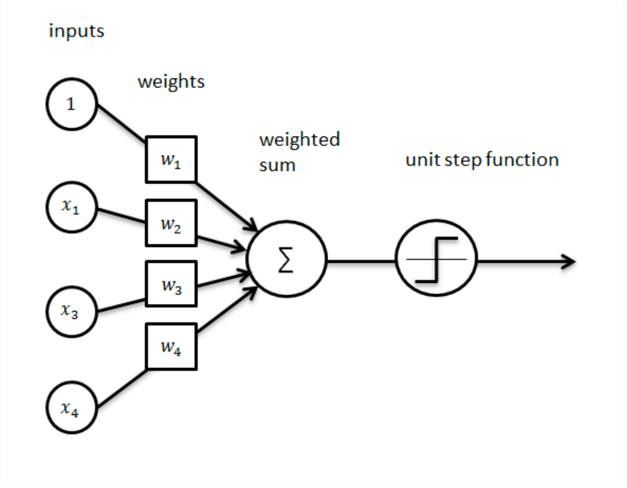
- Artificial neurons resemble such architecture using connections to other neurons, a weighted sum and a non-linear activation function.
- Connections are organized by layers: input layer, hidden layer(s) and output one.
- When we have several hidden layers we get a **deep learning model**.

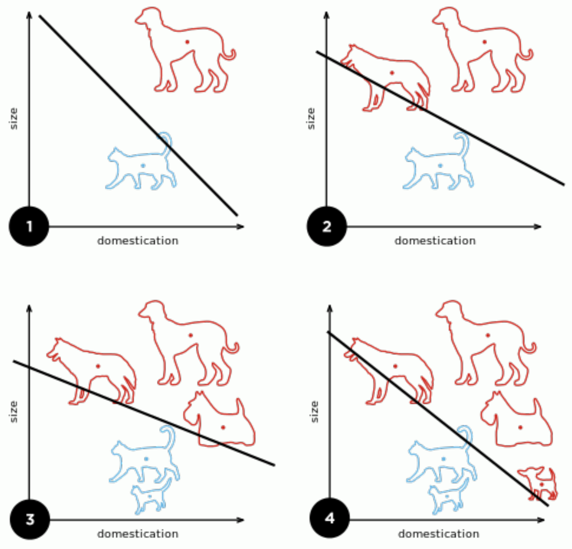




5. Perceptron: the simplest AN

- The perceptron is one of the simplest neural network, and dates back to the 1950s.
- The name also refers to the algorithm for supervised learning of binary classifiers.
- The perceptron algorithm processes input samples from the training set one at a time.
- It is a type of linear classifier, and its predictions are based on a linear predictor function combining a set of weights with the feature vector.





5. Perceptron

- Therefore, the perceptron is an algorithm for learning a binary classifier: a function that maps its input *x* (a real-valued vector) to a single binary output value *f(x)*:

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

- where *w* is a vector of real-valued weights and *b* is the bias that shifts the decision boundary away from the origin.

- The value of f(x) (usually 0 or 1) is used to classify x as either a positive or a negative instance, in the case of a binary classification problem.

- However, **the perceptron learning algorithm does not converge if the learning set is not linearly separable**.

5. Perceptron: Learning

Assume: the bias *b* is denoted as w_0, and we set $x_{j0}=1$ in the input vector.

1. Initialize the weights *W* to small random values.

2. For each input sample x_j and desired output d_j from the training set *D*, perform the following steps:

   a) Calculate the actual output:

   $$\begin{aligned} y_j(t) &= f[\mathbf{w}(t)\cdot\mathbf{x}_j] \\ &= f[w_0(t)x_{j,0} + w_1(t)x_{j,1} + w_2(t)x_{j,2} + \cdots + w_n(t)x_{j,n}] \end{aligned}$$

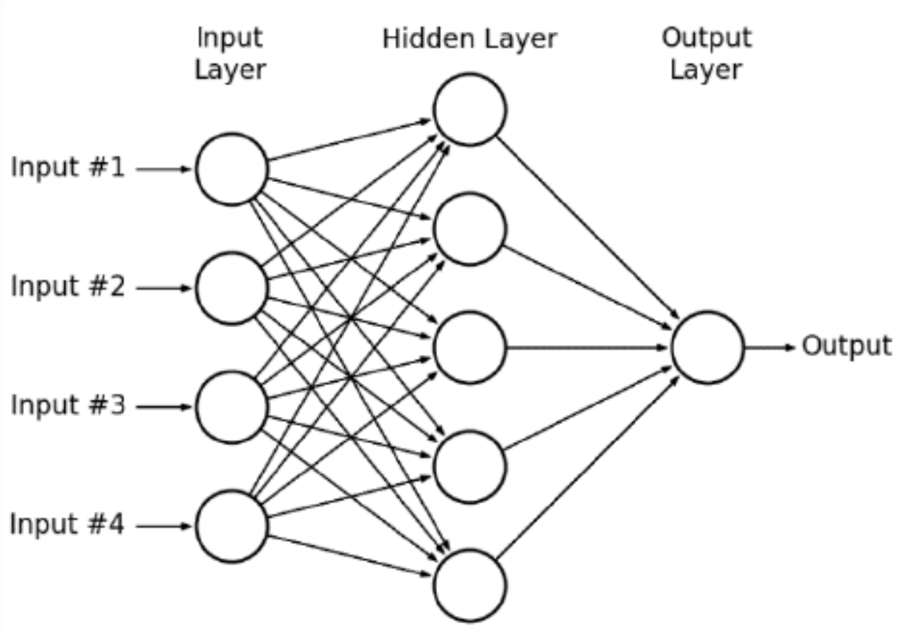
   b) Update the weights:

   $$w_i(t+1) = w_i(t) + (d_j - y_j(t))x_{j,i}$$

3. The step 2 is repeated until the iteration error = $\frac{1}{s}\sum_{j=1}^{s}|d_j - y_j(t)|$ is small or any other stopping criteria.

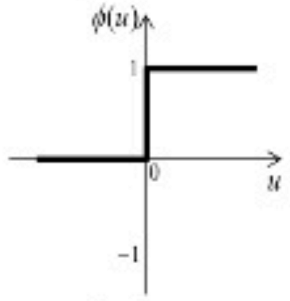
5. Multilayer Perceptron (MLP)

- A multilayer perceptron (MLP) is a class of **feed-forward** artificial neural network.
- An MLP consists of at least three layers of nodes, and **except for the ones on the input layer**, each node is a neuron that uses a **nonlinear activation function**.
- MLP uses a supervised learning technique called back-propagation for training.
- As a difference with the linear perceptron, **MLP can distinguish data that is not linearly separable**.



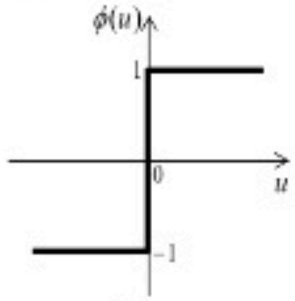
5. MLP: Activation Functions

step function



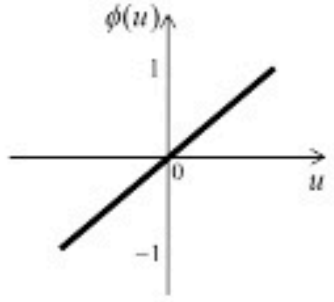
$$\phi_{step}(u) = \begin{cases} 1 & \text{if} \quad u \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

sign function

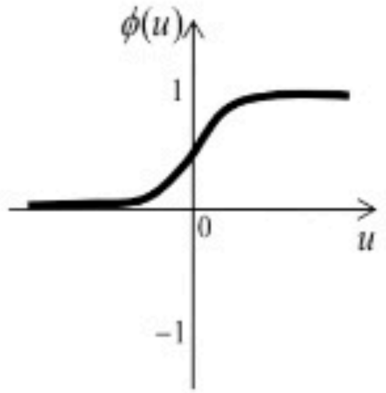


$$\phi_{sign}(u) = \begin{cases} 1 & \text{if} \quad u \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

identity function

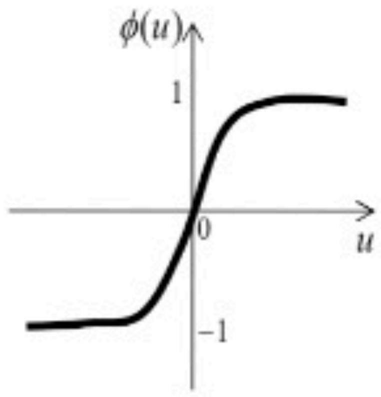


$$\phi_{id}(u) = u$$



$$\phi_{sig}(u) = \frac{1}{1+e^{-u}}$$

sigmoid function



$$\phi_h = \frac{e^u - 1}{e^u + 1}$$

hyper tangent function

5. MLP: Learning

The learning algorithm is divided into two phases: propagation and weight update.

Phase 1: propagation

1. Propagation forward through the network to generate the output value(s).
2. Calculation of the error.
3. Propagation of the output activations back through the network using the training pattern target in order to obtain the error, i.e., the difference between the targeted and actual output values in all output and hidden neurons.

Phase 2: weight update

1. The weights' and the input activation are used to find the **gradient** of the weights.
2. A certain **ratio** (percentage) of the weights' gradient is subtracted from the weight.

This ratio (percentage) influences the speed and quality of learning; and it is called the **learning rate**. The greater the ratio, the faster the net is trained, but the lower the ratio, the more accurate the training is.

Learning is repeated (over the input samples) until the network performs adequately.

5. MLP: Learning II

Pseudocode for training a three-layer network (only one hidden layer) using a stochastic gradient descent algorithm:

```
initialize_network (weights)              // often small random values

do

  forEach training_input_sample named input
    pred_value = get_neural_net_output (network, input)   // forward
    real_value = get_real-output (input)
    compute_error (pred_value – real_value)
    compute $\Delta w_h$   // for weights from hidden to output layer (backward)
    compute $\Delta w_i$   // for weights from input to hidden layer  (backward)
    update_network (weights)               // input layer not modified

until (error is below a threshold or any other stopping criteria)

return network
```

5. MLP: BackPropagation

- The perceptron learns by changing connection weights, after each input sample is processed, based on the amount of error in the network output compared to the expected result.

- MLP uses supervised learning, generalizing the least mean squares algorithm from the linear perceptron.

- Let's denote by *X* the neuron input vector, *W* the weights vector, *Y* the output vector, and *Z* the right output vector. We represent the error in one of the output neurons *j* in the *n*-th input sample by:

$$\delta_j(n) = z_j(n) - y_j(n)$$

- The node weights must be adjusted based on corrections that minimize the error for the whole set of output neurons *J*, given by:

$$E(n) = \frac{1}{2}\sum_j \delta_j^2(n)$$

5. MLP: BackPropagation

- Using gradient descent, the change in each weight is:

$$\Delta w_{ij}(n) = -\eta \frac{\partial E(n)}{\partial v_j(n)} x_i(n)$$

where x_i is the output of the previous neuron, $v_i = \sum_i x_i . w_i$ and η is the learning rate, which is selected to ensure that the weights converge quickly but without oscillations.

- The derivative to be calculated depends on the induced local field v_j, which itself varies, but **for an output node this derivative can be simplified to**:

$$-\frac{\partial E(n)}{\partial v_j(n)} = \delta_j(n) f'(v_j(n))$$

where f ′ is the derivative of the activation function, which does not change, and can be substituted in the upper formulae.

5. MLP: BackPropagation

- In the case of the weights **for a hidden node**, the relevant derivative is:

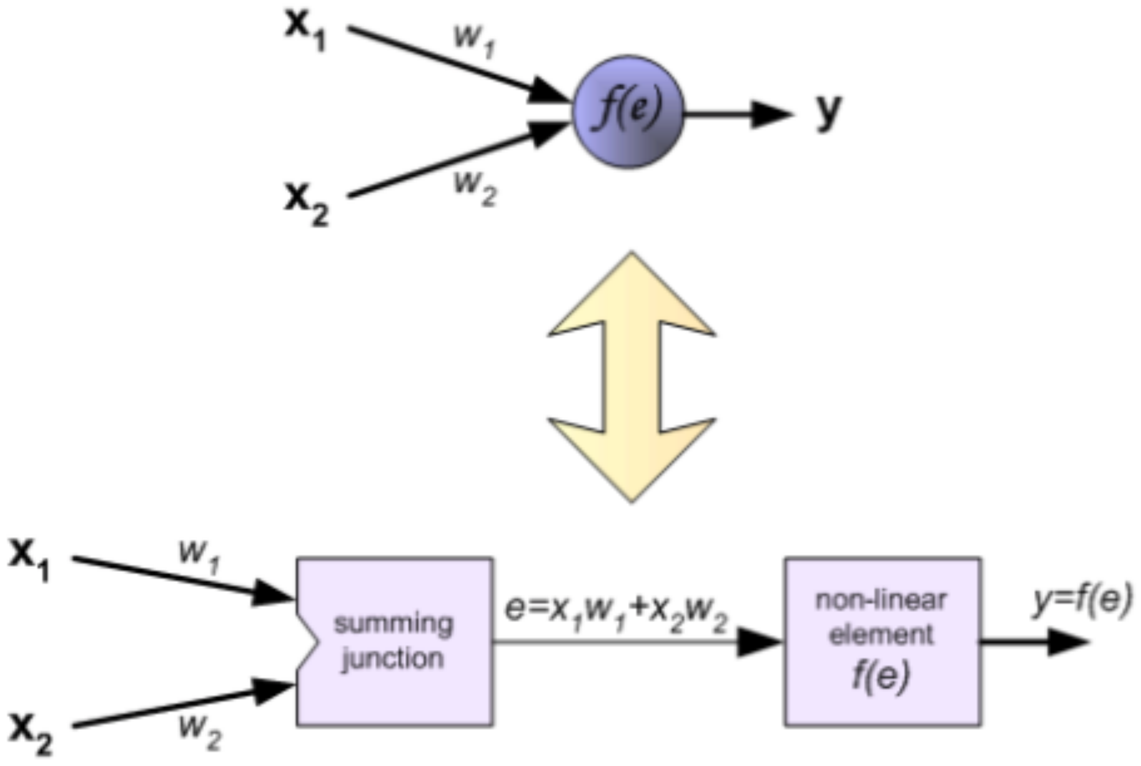
$$-\frac{\partial E(n)}{\partial v_j(n)} = f'(v_j(n))\sum_k -\frac{\partial E(n)}{\partial v_k(n)} w_{kj}(n)$$

- This depends on the weights of the *j*-th nodes, which represent the output layer.

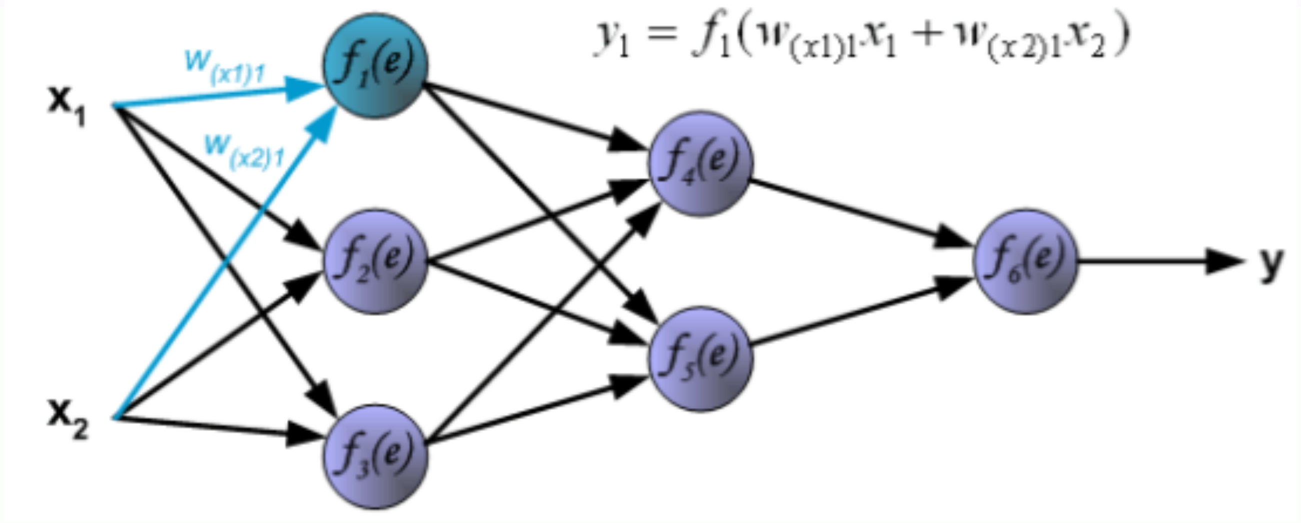
- Thus, **to change the hidden layer weights, we use the output layer weights, that also have changed according to the derivative of the activation function**, i.e., we are backpropagating the weight variations.

5. BackPropagation: Example

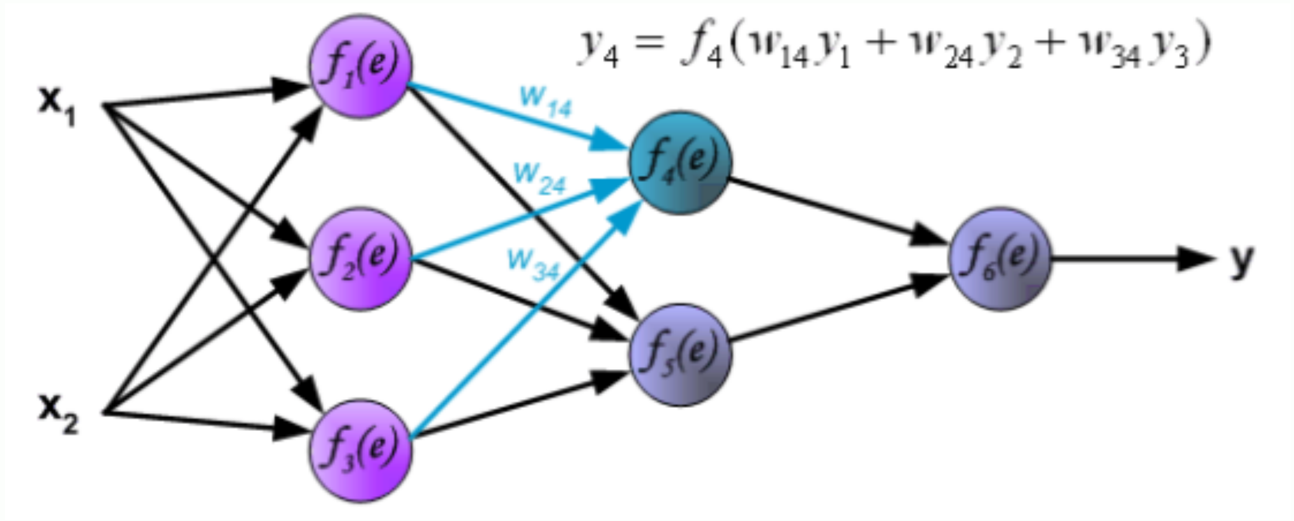
- Let's see an example with two hidden layers and using y = f(e) with e = $\Sigma_i\, w_i.x_i$
- We shall see only the feedforward part of the propagation affecting the upper part of the network, and the same along the backpropagation.
- **Note that 'e' is not the error; that is denoted by δ !!!**



5. BackPropagation: Example



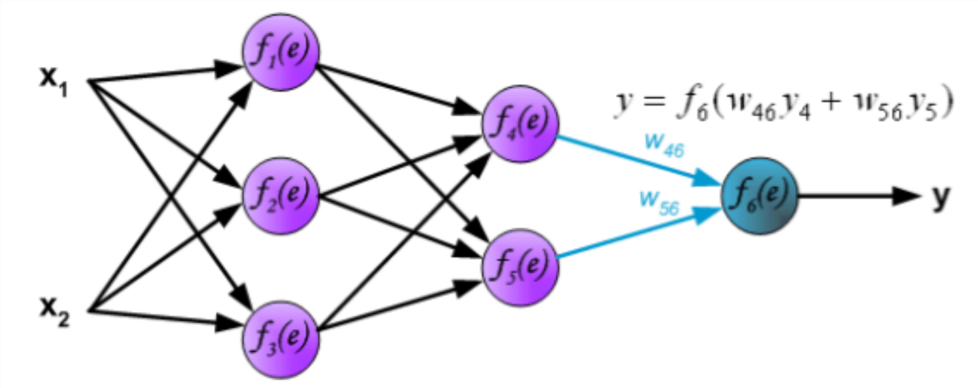
$$y_1 = f_1(w_{(x1)1}x_1 + w_{(x2)1}x_2)$$



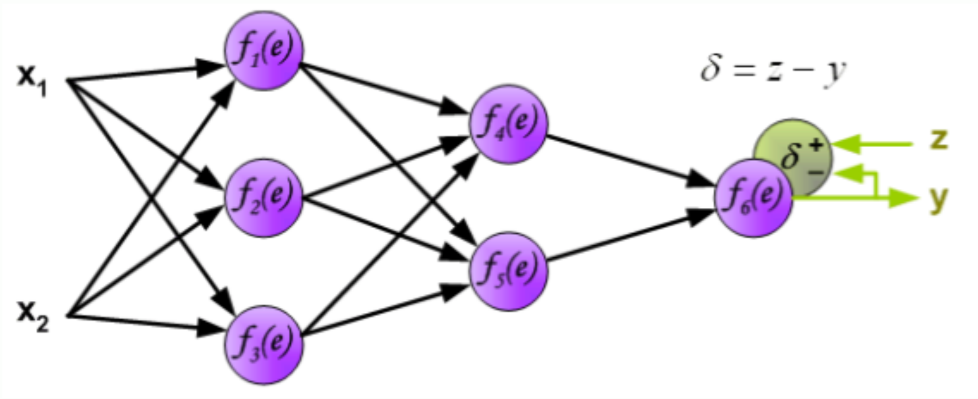
$$y_4 = f_4(w_{14}y_1 + w_{24}y_2 + w_{34}y_3)$$

5. BackPropagation: Example

- Here we get y as the output value…

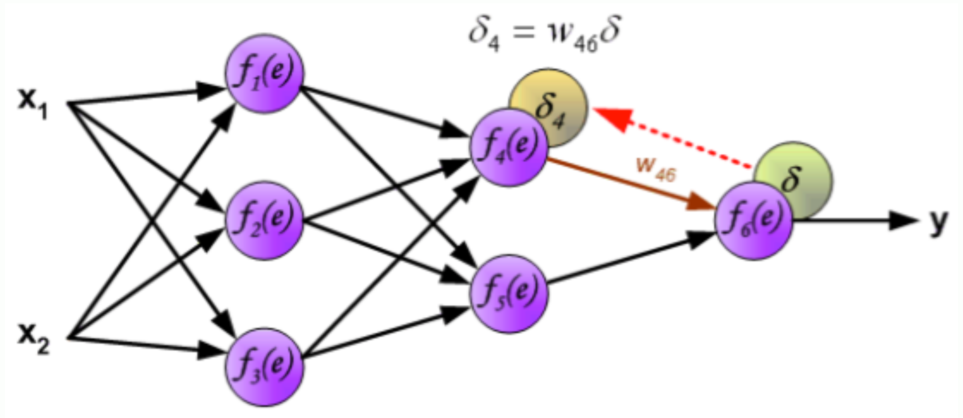


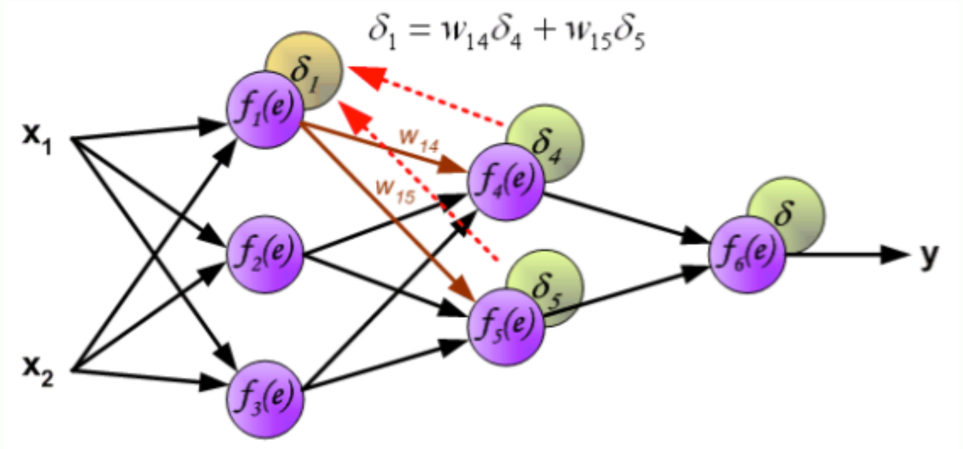
- … and we calculate **the error δ**.



5. BackPropagation: Example

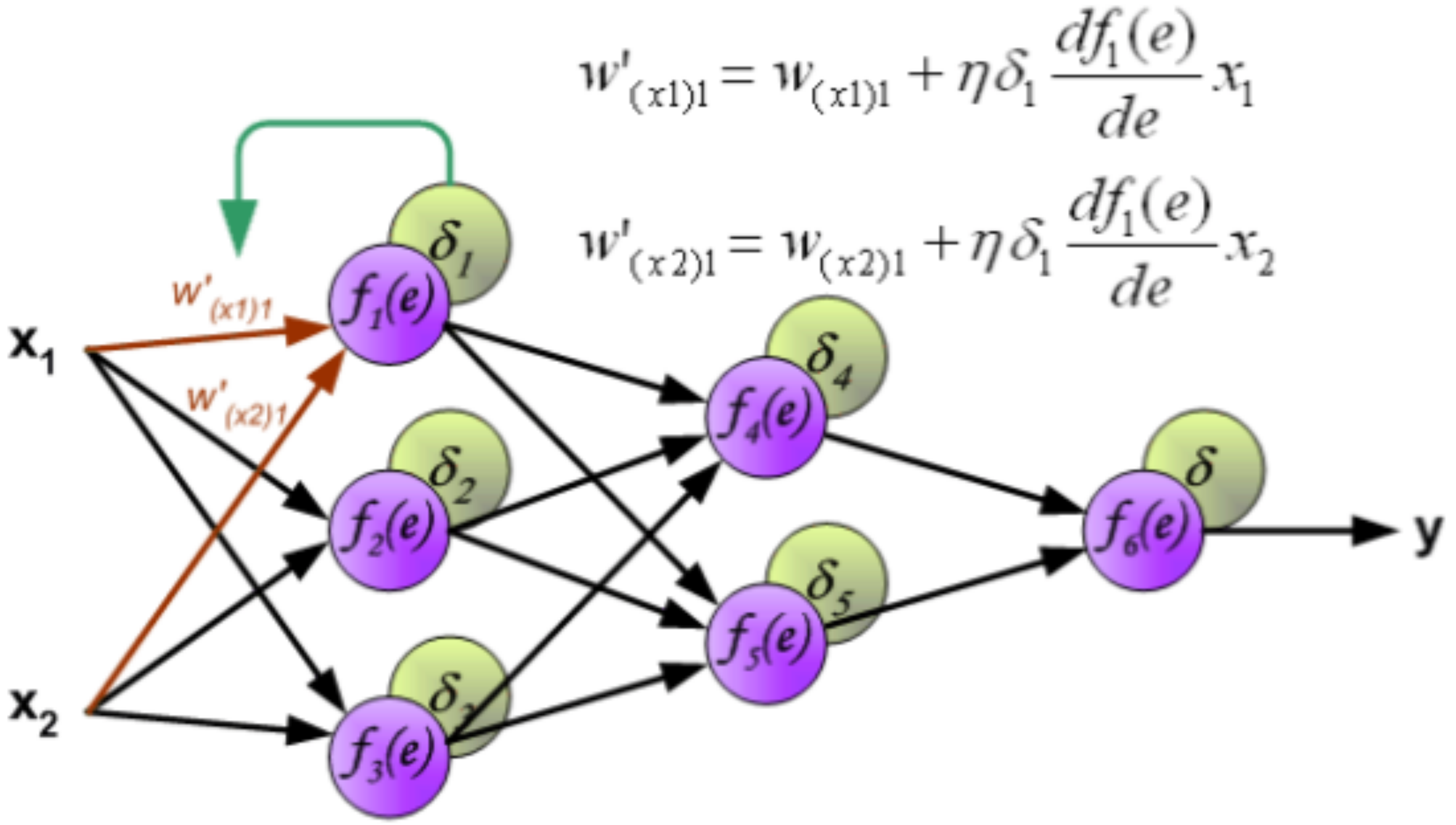
- Errors are calculated in the output layer (here 1 neuron) and backpropagated to the hidden layers.





5. BackPropagation: Example

- Now we can calculate the change in the weights; that, in this single output neuron case, depends on the single derivative of the activation function.



$$w'_{(x1)1} = w_{(x1)1} + \eta \delta_1 \frac{df_1(e)}{de} x_1$$

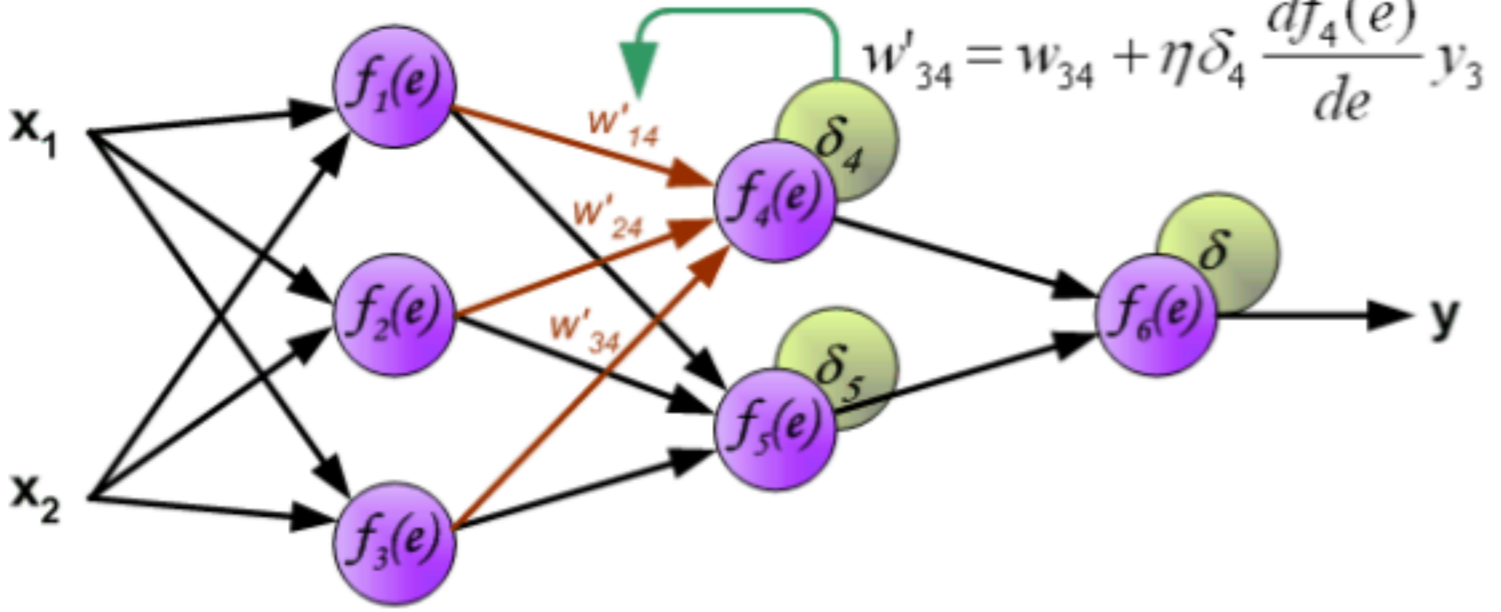
$$w'_{(x2)1} = w_{(x2)1} + \eta \delta_1 \frac{df_1(e)}{de} x_2$$

5. BackPropagation: Example

$$w'_{14} = w_{14} + \eta \delta_4 \frac{df_4(e)}{de} y_1$$

$$w'_{24} = w_{24} + \eta \delta_4 \frac{df_4(e)}{de} y_2$$

$$w'_{34} = w_{34} + \eta \delta_4 \frac{df_4(e)}{de} y_3$$

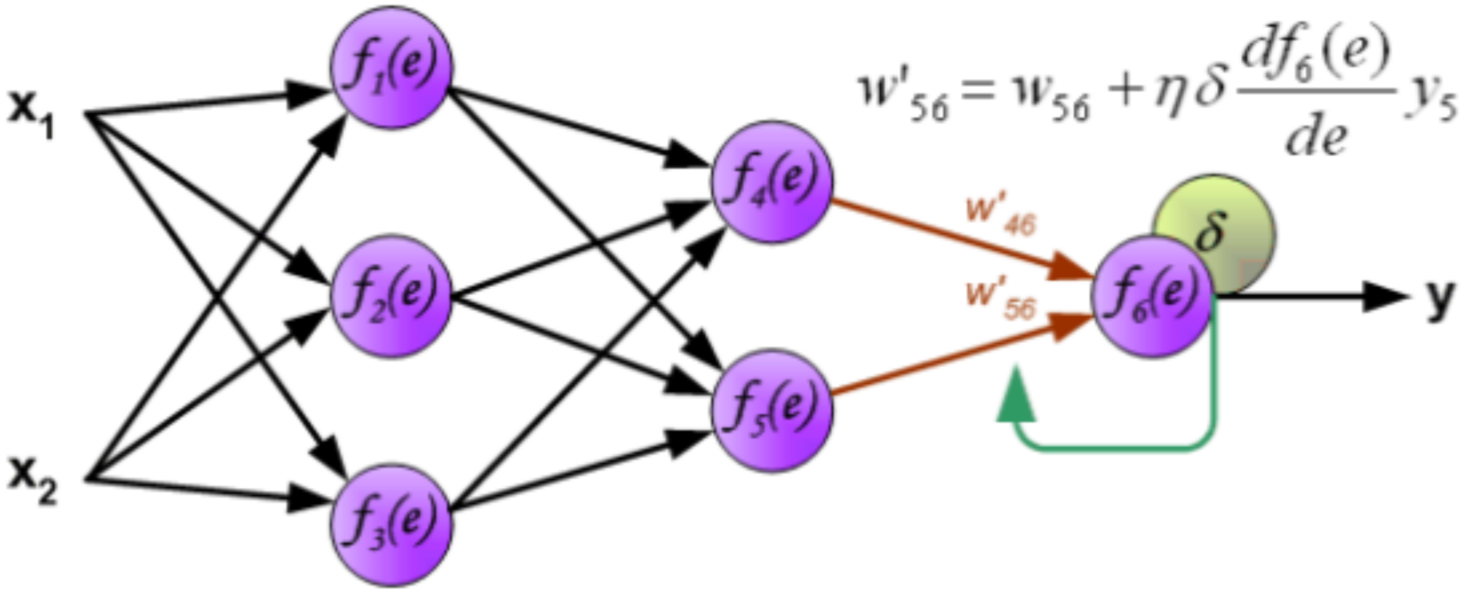


5. BackPropagation: Example

- And we have finished modifying the weights for the upper neurons of the network. We should do the same for the rest of them.

$$w'_{46} = w_{46} + \eta \delta \frac{df_6(e)}{de} y_4$$

$$w'_{56} = w_{56} + \eta \delta \frac{df_6(e)}{de} y_5$$



5. Self-Organizing MAPs

- Self-organizing Maps (SOM or Kohonen network) are neuron arrays, that self-organize themselves using **unsupervised learning**, based on the input patterns,

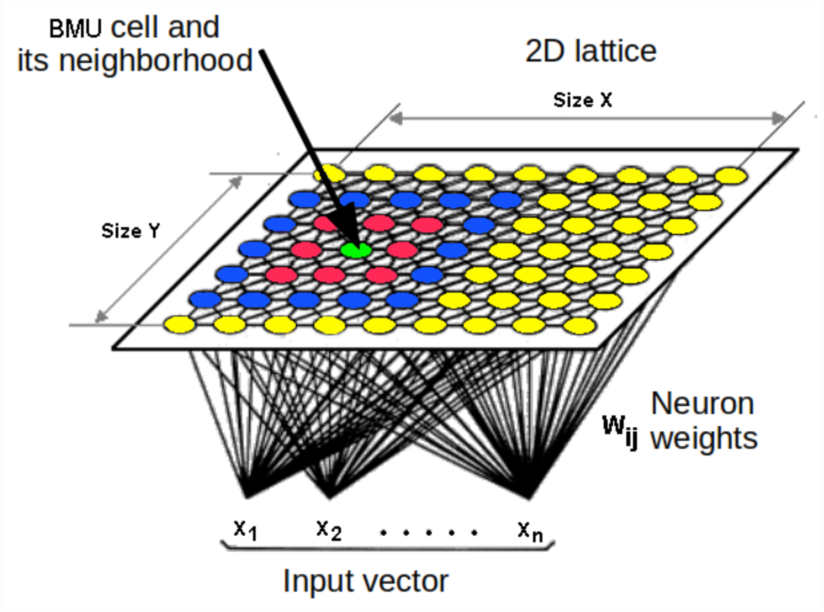
- From a topological point of view, a SOM is a **single layer neural network**, where the neurons are set along a N-dimensional grid. In most applications this grid is 2D and rectangular.

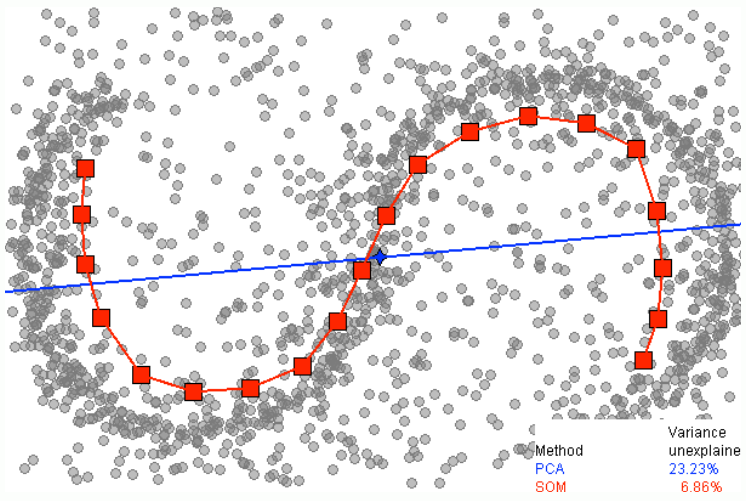
- Associated with each neuron is a weight vector of the same dimension as the input vectors (patterns) and a position in the map space.

- The self-organizing map describes a mapping from a higher dimensional input space to a lower dimensional map space.

5. Self-Organizing MAPs

- After several iterations, the result is an ordered network in which nearby neurons will share certain similarities.





5. Self-Organizing MAPs

- SOM neurons may be initialized randomly, and operate in two modes (like most of the ANNs):
  1. **Training** builds the map using input examples (it is a competitive process, also called vector quantization).
  2. **Mapping** automatically classifies a new input vector.

- The number of neurons m in the map depends on the dataset, but usually: *m = 0,2 * Sqrt (Number_of_samples)*

- The procedure for placing a vector from the input space onto the map is to find the neuron with the closest weight vector, referred as the **Best Matching Unit (BMU)**:

$$BMU\ (x(t)) = argmin\ ||x(t) - w_i(t)||,\ \ \forall i \in \{1, ... , m\},$$

- where $||\cdot||$ denotes the Euclidean distance, and t is the discrete time step associated with the iterations of the algorithm.

5. Self-Organizing MAPs

- The weight vectors $w_j(t)$ in the map are trained according to a competitive-cooperative learning rule, where the weight vector of the winning neuron b and its neighbors are updated with this formula:

$$w_i(t + 1) = w_i(t) + \alpha(t)h_{bi}(t)[x(t) - w_i(t)], \ \forall i \in \{1, . . . , m\},$$

- where $0 < \alpha(t) < 1$ is the learning rate and $h_{bi}(t)$ is a weighting function which limits the neighborhood of the BMU.

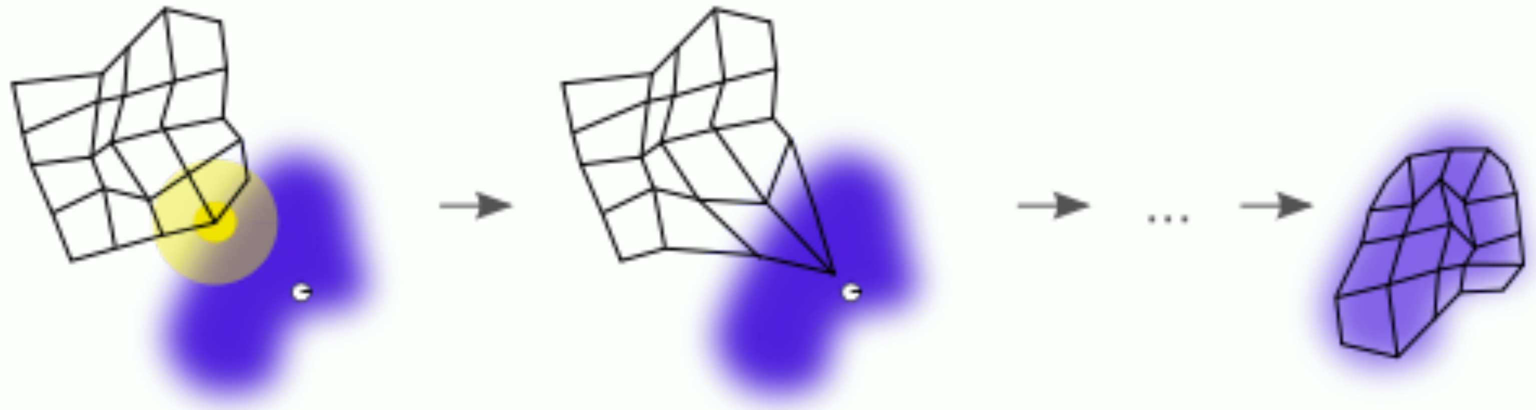
- This neighborhood function assumes values in [0, 1], and is bigger for neurons closer to the BMU.

- The neighborhood radio and α(t) usually decays with t to guarantee convergence of the weight vectors in the map to stable steady states.

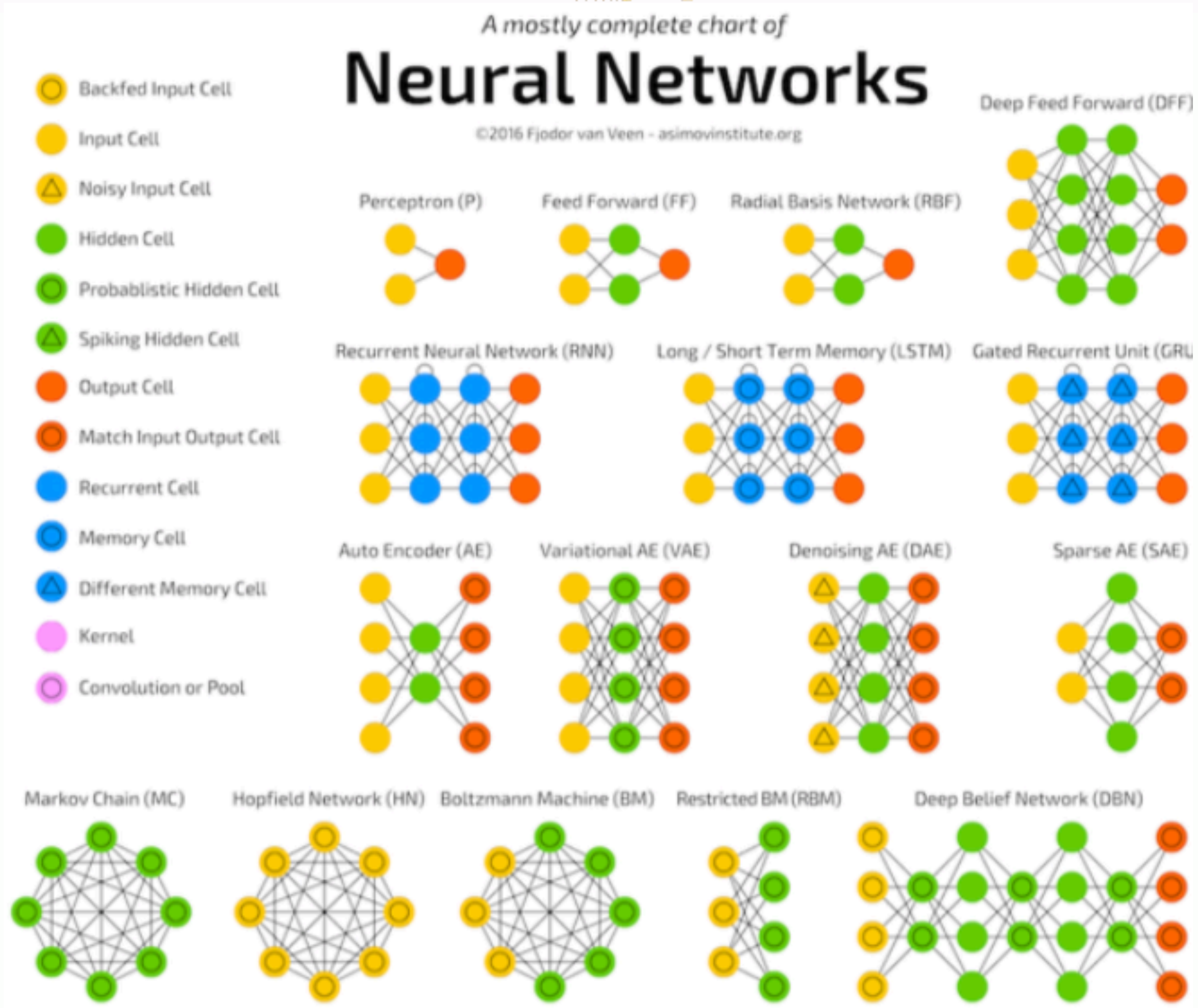
5. Self-Organizing MAPs

- Dense regions of the input space attract more neurons, and the distribution of neurons in the weight space reflects the distribution of the input data in the input space.

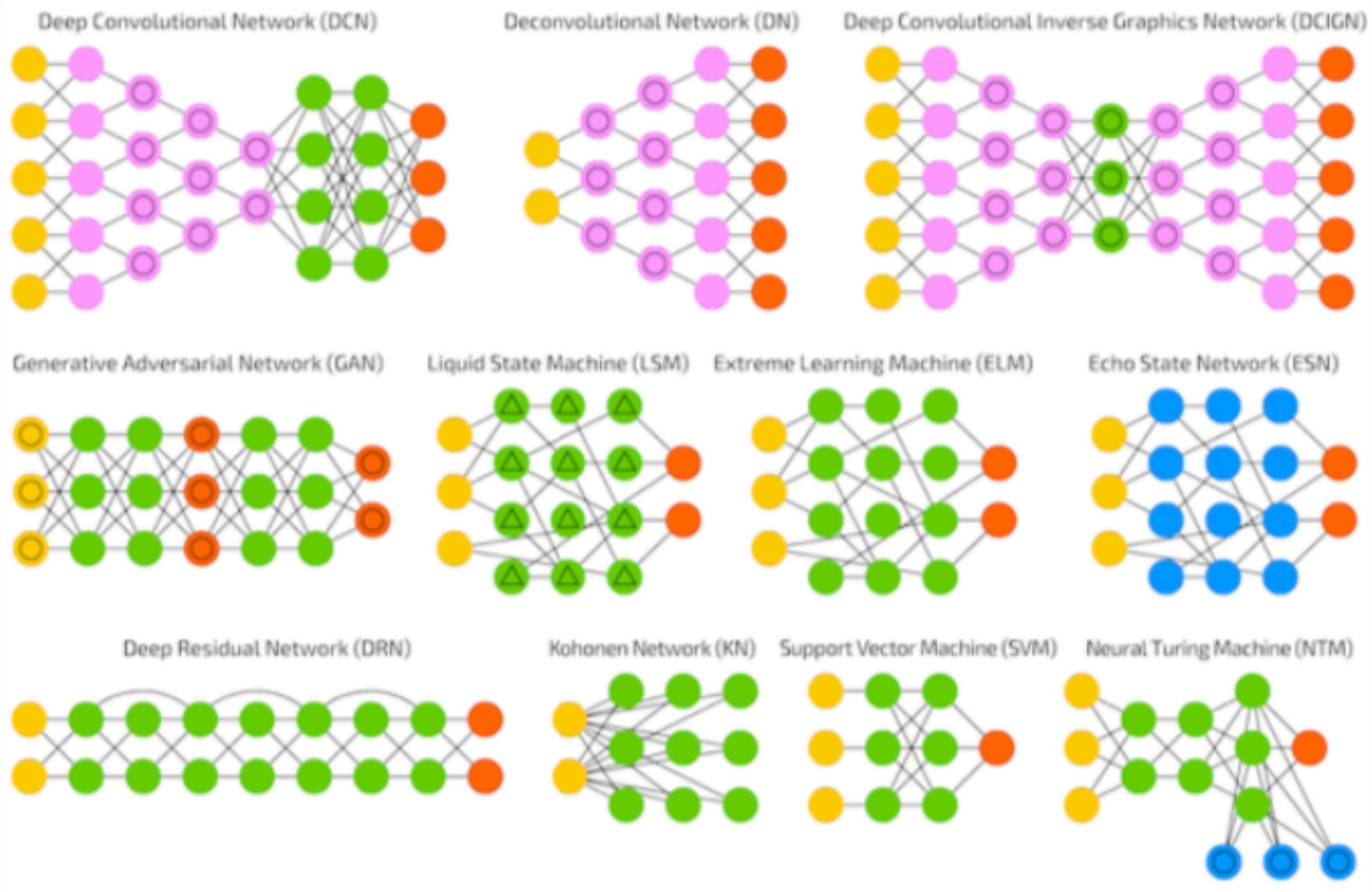
- Therefore similar input patterns activate similar areas in the SOM producing a local specialization in the global self-organized network.



5. Zoology of Neural Networks 1/2



5. Zoology of Neural Networks 2/2


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)
Support Vector Machine (SVM)
Neural Turing Machine (NTM)