Part 7: Learning with function approximation and Deep RL

EL 2805 – Reinforcement Learning

Alexandre Proutiere

KTH, The Royal Institute of Technology

Objectives of this part

- What is function approximation?
- How to modify RL algorithms to account for function approximation?
- Deep Q learning algorithm
- An overview of neural nets and on how to implement SGD

References

- Barto-Sutton's book: Chapters 9-10-11 (different than this part)
- DQN algorithm:
  `https://www.nature.com/articles/nature14236`

Part 7: Outline

1. Summary: SARSA, Q-learning with function approximation, and DQN algorithm
2. Function approximation
3. Policy evaluation using function approximation
4. On-policy control with function approximation
5. Off-policy control with function approximation
6. DQN and Deep learning

1. Summary

We present 3 representative algorithms.
DQN is implemented in Lab 2.

- SARSA with function approximation (on-policy control)
- Q-learning with function approximation (off-policy control)
- Deep Q Network (DQN) algorithm (a particular instance of Q-learning with function approximation)

SARSA algorithm with function approximation

Pseudo-code for infinite-horizon discounted RL.

SARSA algorithm with function approximation

1. **Initialization.** θ, initial state s_1
2. **Iterations:** For every $t \geq 1$,
   compute π_t the ϵ-greedy policy w.r.t. Q_θ
   take action a_t according to π_t, and observe r_t, s_{t+1}
   (alternative: select the "a_{t+1}" of the previous step as a_t)
   sample a_{t+1} according to π_t
   update θ as:

$$\theta \leftarrow \theta + \alpha(r_t + \lambda Q_\theta(s_{t+1}, a_{t+1}) - Q_\theta(s_t, a_t))\nabla_\theta Q_\theta(s_t, a_t)$$

Q-learning with function approximation

Pseudo-code for infinite-horizon discounted RL.

Q-learning with function approximation

1. **Initialization.** θ, initial state s_1
2. **Iterations:** For every $t \geq 1$,
   compute π_t the ϵ-greedy policy w.r.t. Q_θ
   take action a_t according to π_t, and observe r_t, s_{t+1}
   update θ as:

$$\theta \leftarrow \theta + \alpha(r_t + \lambda \max_b Q_\theta(s_{t+1}, b) - Q_\theta(s_t, a_t))\nabla_\theta Q_\theta(s_t, a_t)$$

DQN algorithm

1. **Initialization.** θ and ϕ, replay buffer B, initial state s_1
2. **Iterations:** For every $t \geq 1$,
   compute π_t the ϵ-greedy policy w.r.t. Q_θ
   take action a_t according to π_t, and observe r_t, s_{t+1}
   store (s_t, a_t, r_t, s_{t+1}) in B
   sample k experiences (s_i, a_i, r_i, s'_i) from B
   for $i \in [1, k]$: compute using the target net. with weights ϕ

$$y_i = \begin{cases} r_i & \text{if episode stops in } s'_i \\ r_i + \lambda \max_b Q_\phi(s'_i, b) & \text{otherwise} \end{cases}$$

   update the weights θ (using back prop.) as:

$$\theta \leftarrow \theta + \alpha(y_i - Q_\theta(s_i, a_i))\nabla_\theta Q_\theta(s_i, a_i)$$

   every C steps: $\phi \leftarrow \theta$

2. Function approximation

- The best regret and sample complexity scale as $S \times A$
- Continuous action and state spaces

Video games: state = image ($S = ((255)^3)^{250000}$)

RL with function approximation

Idea: restrict our attention to learning functions belonging to a parametrized family of functions. Generally, we approximate state value functions, (state, action) value functions, or policies.

Examples: Value function and Q-function

1. Linear functions: $\mathcal{V} = \{V_\theta, \theta \in \mathbb{R}^M\}$ and $\mathcal{Q} = \{Q_\theta, \theta \in \mathbb{R}^M\}$,

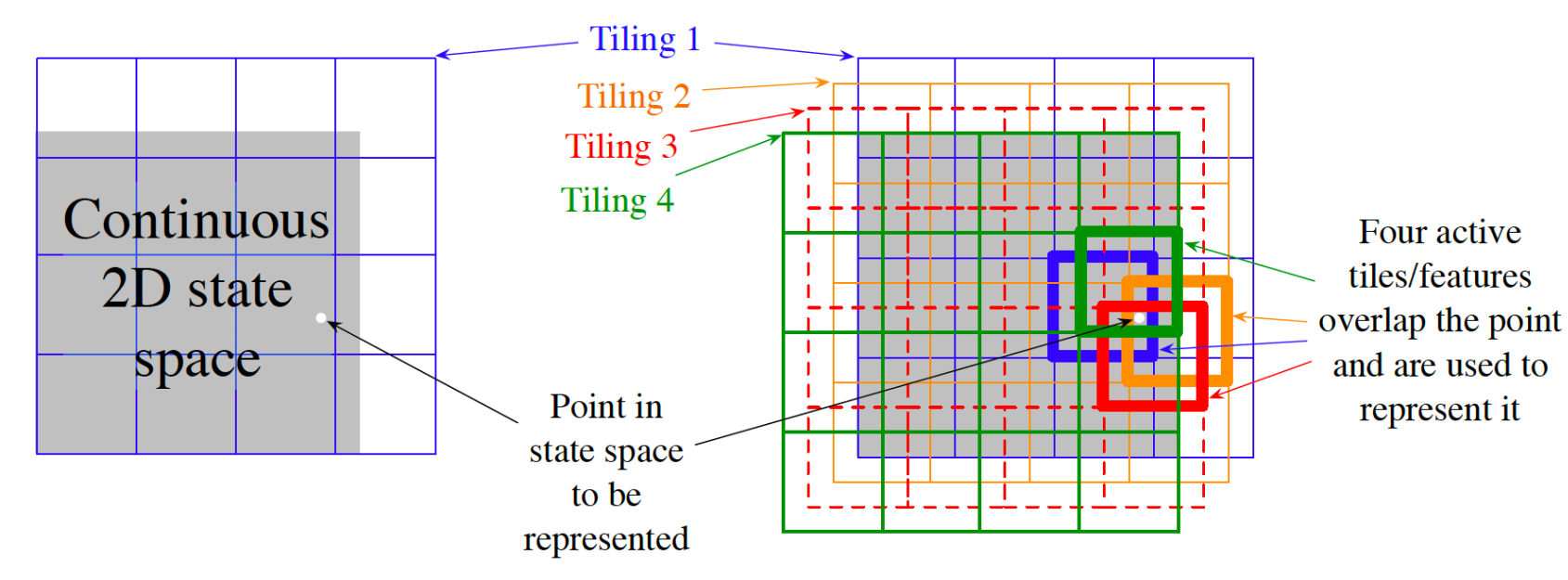
$$V_\theta(s) = \sum_{i=1}^{M} \phi_i(s)\theta_i = \phi(s)^\top \theta, \qquad Q_\theta(s,a) = \sum_{i=1}^{M} \phi_i(s,a)\theta_i = \phi(s,a)^\top \theta$$

   where the ϕ_i's are linearly independent.

2. Deep networks: $\mathcal{V} = \{V_{\mathbf{w}}, \mathbf{w} \in \mathbb{R}^M\}$ and $\mathcal{Q} = \{Q_{\mathbf{w}}, \mathbf{w} \in \mathbb{R}^M\}$. $V_{\mathbf{w}}(s)$ (resp. $Q_{\mathbf{w}}(s,a)$) is given as the output of a neural network with weights $\mathbf{w}$ and inputs s (resp. (s,a)).

Linear function approximation: Examples

- State aggregation: M clusters of states $V_1, \ldots, V_M$. $\phi_i(s) = 1_{\{s \in V_i\}}$. $V^\pi(s) \approx \sum_i \theta_i \phi_i(s)$.
- Tile coding:



- Fourier basis: $\phi_i(s) = \cos(i\pi s)$

3. Policy evaluation using function approximation

Discounted MDP with terminal state

- MDP: $(\lambda, S, A_s, p(\cdot|s,a), r(s,a), a \in A_s, s \in S)$.
- **Terminal state:** There is a state s_{end} after which no reward is collected.
- **Episode:** It starts at time $t = 1$ in state s_1 and finishes after a random time when s_{end} is reached.
- Assumption: under any policy, episodes finish in finite time almost surely.
- State value function of π: for any $s \in \mathcal{S}$,

$$V^\pi(s) = \mathbb{E}_\pi[\sum_{t \geq 1} \lambda^{t-1} r(s_t, a_t) | s_1 = s].$$

On-policy evaluation using Monte Carlo methods

Let π be a deterministic stationary policy.

We wish to estimate its state value function V^π using MC methods and function approximation, i.e., we want to find θ such that V_θ is the best approximation of V^π within the set $\mathcal{V} = \{V_\theta, \theta \in \mathbb{R}^M\}$.

Objective: find θ minimizing:

$$J(\theta) = \frac{1}{2}\mathbb{E}_{s\sim\mu}[(V^\pi(s) - V_\theta(s))^2] = \sum_s \mu(s)(V^\pi(s) - V_\theta(s))^2,$$

where μ denotes the state stationary distribution under π.

Evaluating π through SGD

$$\nabla J(\theta) = \mathbb{E}_{s\sim\mu}[-(V^\pi(s) - V_\theta(s))\nabla_\theta V_\theta(s)]$$

The *target* $V^\pi(s)$ is unknown, and can be estimated using the MC prediction algorithm:

- Simulate π and generate episodes
- Let $G(s)$ be the return obtained when s is visited for the first time in an episode, then

$$\mathbb{E}_{s\sim\mu}[G(s)] = \mathbb{E}_{s\sim\mu}[V^\pi(s)].$$

Hence the algorithm presented in the next slide in a SGD algorithm to minimize J.

MC prediction with function approximation

Monte Carlo prediction algorithm:

1. **Initialization:** θ
2. **Iterations:** for episode $i = 1, \ldots, n$

   generate $\tau_i = (s_{1,i}, a_{1,i}, r_{1,i}, \ldots, s_{T_i,i}, a_{T_i,i}, r_{T_i,i})$ under π

   $G = 0$

   for $t = T_i, T_i - 1, \ldots, 1$:

   a. $G = r_{t,i} + \lambda G$

   b. Unless $s_{t,i}$ appears in $\{s_{1,i}, \ldots, s_{t-1,i}\}$

   $$\theta \leftarrow \theta + \alpha(G - V_\theta(s_{t,i}))\nabla_\theta V_\theta(s_{t,i})$$

TD algorithms

For infinite-horizon discounted RL problems, we need a bootstrap method, since V^π can not be estimated directly.

Solution 1. (cf Barto-Sutton's book) Replace the target by $r(s, \pi(s)) + \lambda V_\theta(s')$ where s' is the next observed state. Observe in step t: s_t, a_t, r_t, s_{t+1} under π, the update becomes:

$$\theta \leftarrow \theta + \alpha(r_t + \lambda V_\theta(s_{t+1}) - V_\theta(s_t))\nabla_\theta V_\theta(s_t)$$

This is not a SGD algorithm. A *semi-gradient* algorithm.

Solution 2. Minimize the TD error $J(\theta) = \frac{1}{2}\mathbb{E}_{s\sim\mu}[(r(s, \pi(s)) + \lambda V_\theta(s') - V_\theta(s))^2]$. Leading to the update:

$$\theta \leftarrow \theta + \alpha(r_t + \lambda V_\theta(s_{t+1}) - V_\theta(s_t))(\nabla_\theta V_\theta(s_t) - \lambda\nabla_\theta V_\theta(s_{t+1}))$$

TD(0) algorithm with function approximation

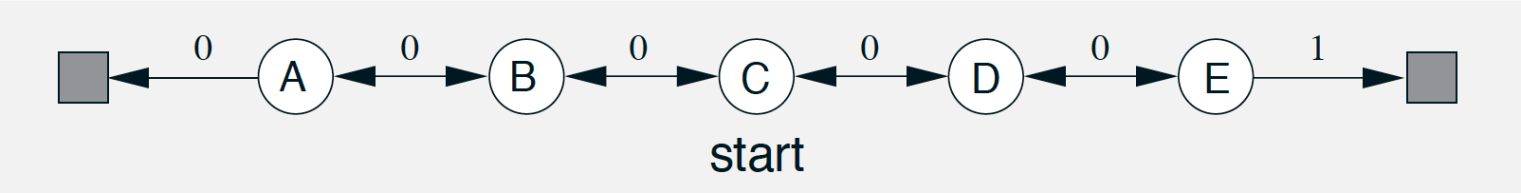
Pseudo-code for infinite-horizon discounted RL.

TD(0) algorithm

1. **Initialization.** θ, initial state s_1
2. **Iterations:** For every $t \geq 1$, observe s_t, a_t, r_t, s_{t+1} under π. Update θ as:

$$\theta \leftarrow \theta + \alpha(r_t + \lambda V_\theta(s_{t+1}) - V_\theta(s_t))\nabla_\theta V_\theta(s_t)$$

Example: random walk

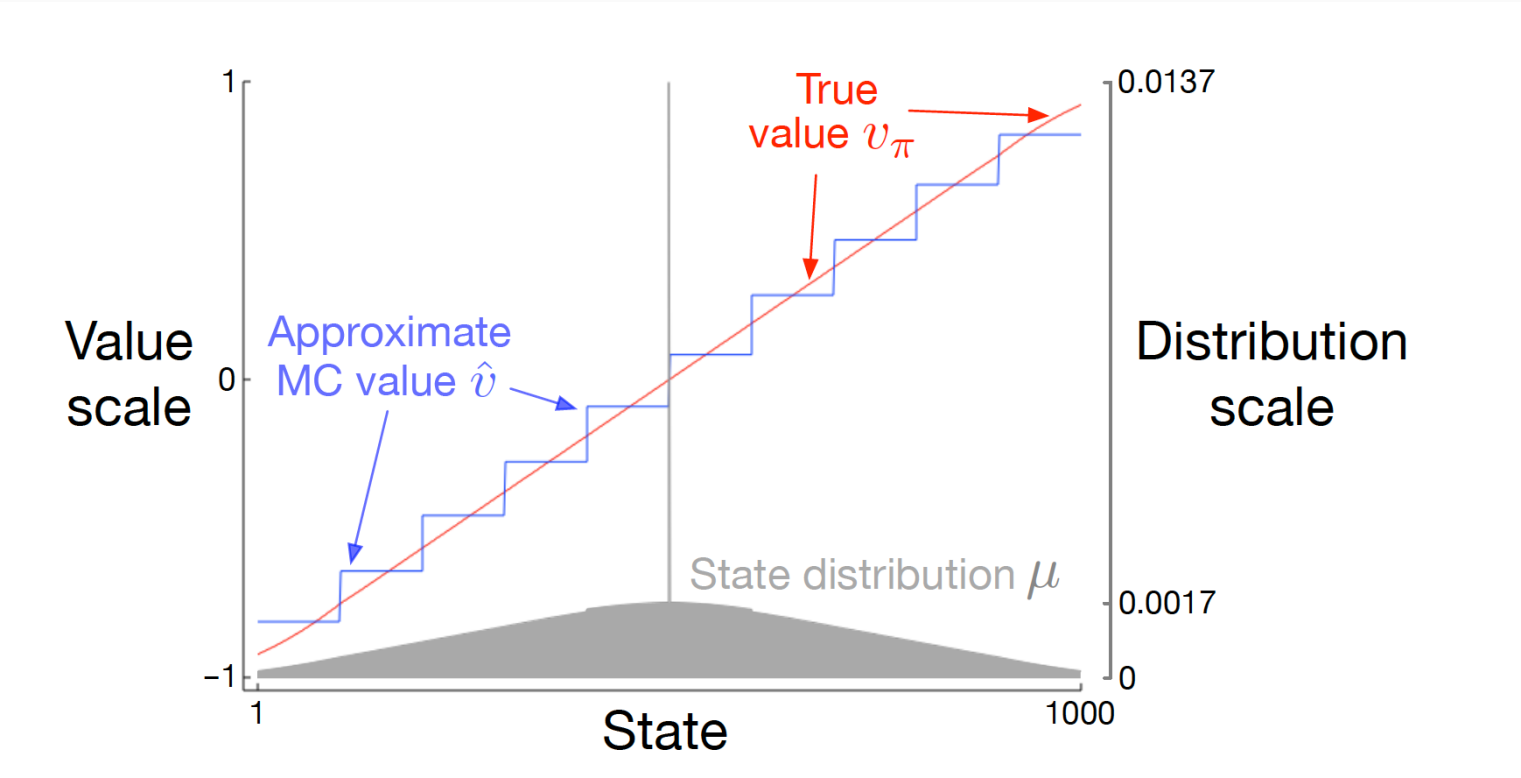


The random walk example[1]. 1000 states, 1 to 1000 from left to right.
Episodes start in state 500.
Policy π: move right and left with equal probability. When moving right (resp. left), the next state is one of the 100 right (resp. left) neighbors with equal probability.

[1] See Sutton-Barto's book

Example: random walk

State aggregation: each cluster has 100 consecutive states.



4. On-policy control with function approximation

The TD algorithms with function approximation can be applied to provide an approximation of the (state, action) value function of a given policy π.

$$\forall s, a, \quad Q^{\pi}(s, a) \approx Q_{\theta}(s, a)$$

When the (state, action) value function of π is estimated, we can improve the policy by e.g. taking the ϵ-greedy policy w.r.t. Q^{π}. This is the SARSA algorithm with function approximation.

SARSA algorithm with function approximation

Pseudo-code for infinite-horizon discounted RL.

SARSA algorithm with function approximation

1. **Initialization.** θ, initial state s_1
2. **Iterations:** For every $t \geq 1$,
   compute π_t the ϵ-greedy policy w.r.t. Q_θ
   take action a_t according to π_t, and observe r_t, s_{t+1}
   (alternative: select the "a_{t+1}" of the previous step as a_t)
   sample a_{t+1} according to π_t
   update θ as:

$$\theta \leftarrow \theta + \alpha(r_t + \lambda Q_\theta(s_{t+1}, a_{t+1}) - Q_\theta(s_t, a_t))\nabla_\theta Q_\theta(s_t, a_t)$$

5. Off-policy control with function approximation

Off-policy control with function approximation is not a well-understood topic. We present below heuristics, without convergence guarantees.

Bellman Error of a (state, action) function $\tilde{Q}$:

$$BE(s,a) = r(s,a) + \lambda \sum_j p(j|s,a) \max_b \tilde{Q}(j,b) - \tilde{Q}(s,a)$$

A possible objective: minimize $J(\theta)$ the Mean Square Bellman Error:

$$\begin{aligned} J(\theta) &= \frac{1}{2}\mathbb{E}_{(s,a)\sim\mu_b}[BE(s,a)^2] \\ &= \frac{1}{2}\mathbb{E}_{(s,a)\sim\mu_b}[(r(s,a) + \lambda \sum_j p(j|s,a) \max_b Q_\theta(j,b) - Q_\theta(s,a))^2] \end{aligned}$$

μ_b is the stationary distribution of (s,a) under π_b.

Q-learning algorithm: Tabular case

Q-learning "looks" like a SGD of J in the tabular case ($\theta = (Q(s,a))_{s,a}$)

Parameter. Step sizes (α_t)

1. **Initialization.** Select a Q-function $Q^{(0)} \in \mathbb{R}^{S \times A}$
2. **Observations.** (s_t, a_t, r_t, s_{t+1}) under the behavior policy π_b
3. Q**-function improvement.** For $t \geq 0$. Update the estimated Q-function as follows: $\forall s, a,$

$$Q^{(t+1)}(s,a) = Q^{(t)}(s,a)$$
$$+ 1_{(s_t,a_t)=(s,a)} \alpha_{n^{(t)}(s_t,a_t)} \left[r_t + \lambda \max_{b \in \mathcal{A}} Q^{(t)}(s_{t+1}, b) - Q^{(t)}(s_t, a_t) \right]$$

where $n^{(t)}(s,a) := \sum_{m=1}^{t} 1[(s,a) = (s_m, a_m)]$.

Semi-gradient methods

$$J(\theta) = \frac{1}{2}\mathbb{E}_{(s,a)\sim\mu_b}[(r(s,a) + \lambda \sum_j p(j|s,a) \max_b Q_\theta(j,b) - Q_\theta(s,a))^2]$$

The semi-gradient:

$$-\mathbb{E}_{(s,a)\sim\mu_b}\Big[\underbrace{(r(s,a) + \lambda \sum_j p(j|s,a) \max_b Q_\theta(j,b)}_{\text{target}} - Q_\theta(s,a))\nabla_\theta Q_\theta(s,a)\Big]$$

Semi-gradient estimate: observation (s, a, r, s')

$$-(r + \lambda \max_b Q_\theta(s', b) - Q_\theta(s,a))\nabla_\theta Q_\theta(s,a))$$

Q-learning with function approximation

One possible implementation is as follows. No guarantees (actually does not seem to work with neural nets approximation)

Q-learning with function approximation

1. **Initialization.** θ, initial state s_1
2. **Iterations:** For every $t \geq 1$,
   compute π_t the ϵ-greedy policy w.r.t. Q_θ
   take action a_t according to π_t, and observe r_t, s_{t+1}
   update θ as:

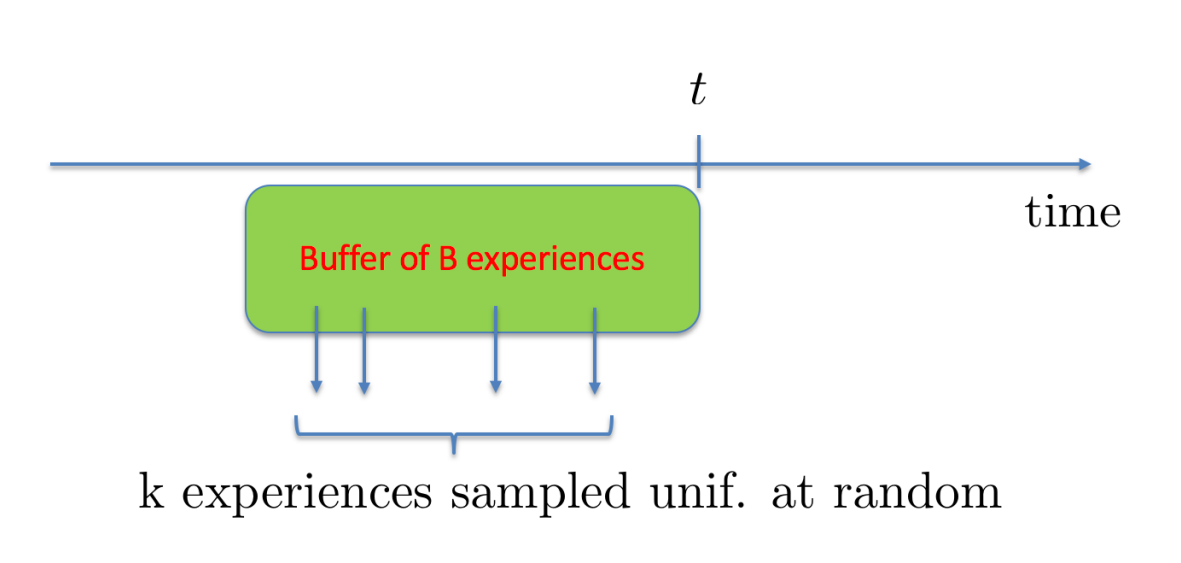
$$\theta \leftarrow \theta + \alpha(r_t + \lambda \max_b Q_\theta(s_{t+1}, b) - Q_\theta(s_t, a_t))\nabla_\theta Q_\theta(s_t, a_t)$$

Experience Replay

$$\theta \leftarrow \theta + \alpha(r_t + \lambda \max_b Q_\theta(s_{t+1}, b) - Q_\theta(s_t, a_t))\nabla_\theta Q_\theta(s_t, a_t)$$

Successive updates are strongly correlated (following a particular trajectory) – affect the convergence rate.

Experience replay: maintain a buffer B of previous experiences (s, a, r, s'). Sample mini-batches of fixed size k from B uniformly at random, and update θ accordingly.



Experience Replay

Updates with ER and mini-batches:

Sample k experiences from the buffer B

For $i = 1, \ldots, k$, experience (s_i, a_i, r_i, s'_i)

$$\theta \leftarrow \theta + \alpha(r_i + \lambda \max_b Q_\theta(s'_i, b) - Q_\theta(s_i, a_i))\nabla_\theta Q_\theta(s_i, a_i)$$

Fixed targets

$$\theta \leftarrow \theta + \alpha(\underbrace{r_t + \lambda \max_b Q_\theta(s_{t+1}, b)}_{\text{non-stationary target}} - Q_\theta(s_t, a_t))\nabla_\theta Q_\theta(s_t, a_t)$$

The target evolves as θ is constantly updated – it moves too fast to get tracked.

Solution: fix the target for C successive steps.
For every step:

$$\theta \leftarrow \theta + \alpha(r_t + \lambda \max_b Q_\phi(s_{t+1}, b) - Q_\theta(s_t, a_t))\nabla_\theta Q_\theta(s_t, a_t)$$

Every C steps, update the target: $\phi \leftarrow \theta$

Q-learning with ER and fixed targets

1. **Initialization.** θ and ϕ, replay buffer B, initial state s_1
2. **Iterations:** For every $t \geq 1$,
   compute π_t the ϵ-greedy policy w.r.t. Q_θ
   take action a_t according to π_t, and observe r_t, s_{t+1}
   store (s_t, a_t, r_t, s_{t+1}) in B
   sample k experiences (s_i, a_i, r_i, s'_i) from B
   for $i = 1, \ldots, k$:

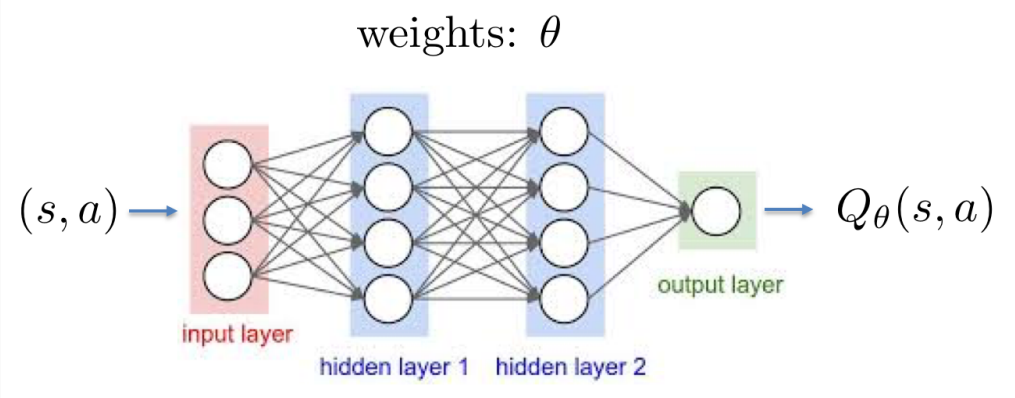
$$y_i = \begin{cases} r_i & \text{if episode stops in } s'_i \\ r_i + \lambda \max_b Q_\phi(s'_i, b) & \text{otherwise} \end{cases}$$

   update θ as:

$$\theta \leftarrow \theta + \alpha(y_i - Q_\theta(s_i, a_i))\nabla_\theta Q_\theta(s_i, a_i)$$

   every C steps: $\phi \leftarrow \theta$

DQN: Deep Q networks



With deep neural networks as function approximator, we get DQN:
`https://www.nature.com/articles/nature14236`
`https://www.youtube.com/watch?v=fevMOp5TDQs&t=2066s`

Tested on 49 atari games ...

Design of DQN: with fixed targets, we have a simple least square regression on the (state, action) value. Using classical Deep Learning training techniques to update θ.

DQN on atari games

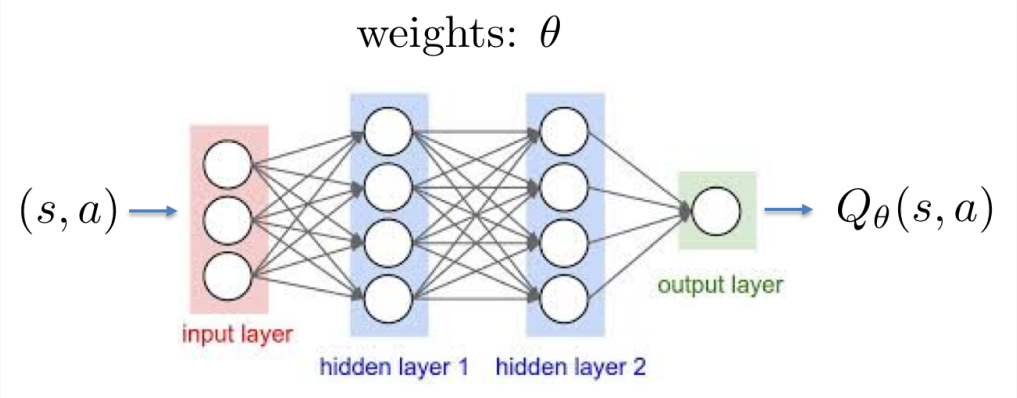
https://www.youtube.com/watch?v=V1eYniJ0Rnk

6. DQN and Deep Learning

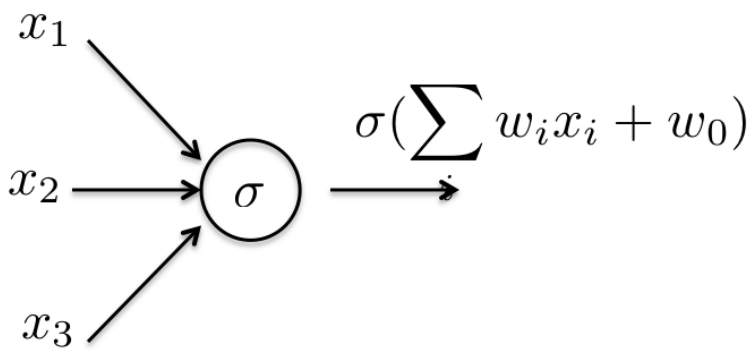
In DQN with ER and fixed targets, the "dataset" is the replay buffer B. Each data sample i from B has an example of input $X_i = (s_i, a_i)$ with the corresponding output $Y_i = r_i + \lambda \max_b Q_\phi(s'_i, b)$ computed from the target network.

We train the network to fit the data, i.e., to find θ minimizing the empirical risk:

$$\frac{1}{2}\sum_i (Y_i - Q_\theta(s_i, a_i))^2$$



Neural networks

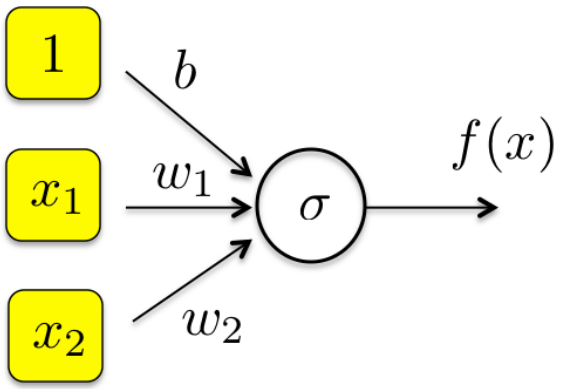


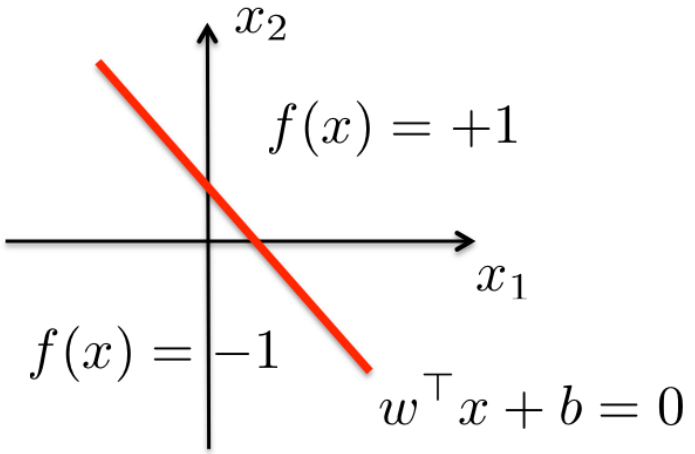
Loosely inspired by how the brain works[2]. Construct a network of simplified neurones, with the hope of approximating and learning any possible function

[2] Mc Culloch-Pitts, 1943

The perceptron

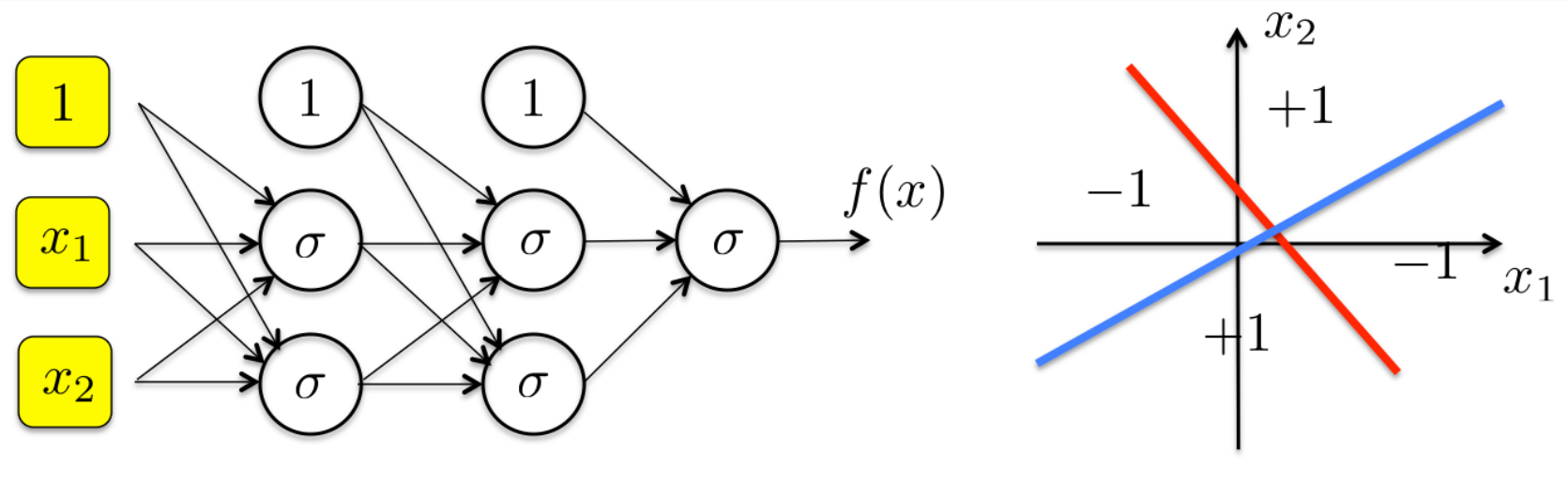
The first artificial neural network with one layer, and $\sigma(x) = \text{sgn}(x)$ (classification)
Input $x \in \mathbb{R}^d$, output in $\{-1, 1\}$. Can represent separating hyperplanes.





Multilayer perceptrons

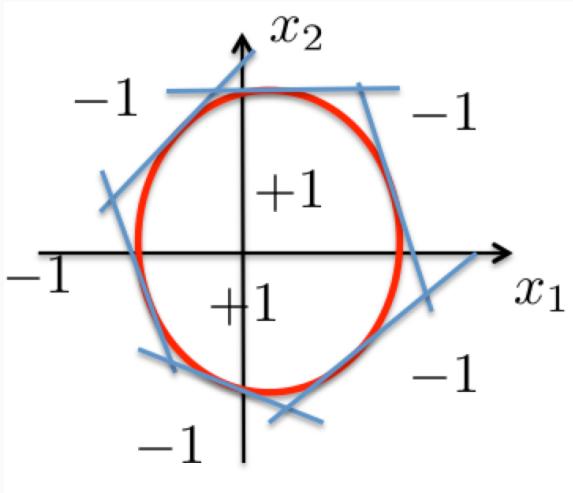
They can represent any function of $\mathbb{R}^d$ to $\{-1, 1\}$



... but the structure depends on the **unknown** target function f, and is difficult to optimise

From perceptrons to neural networks

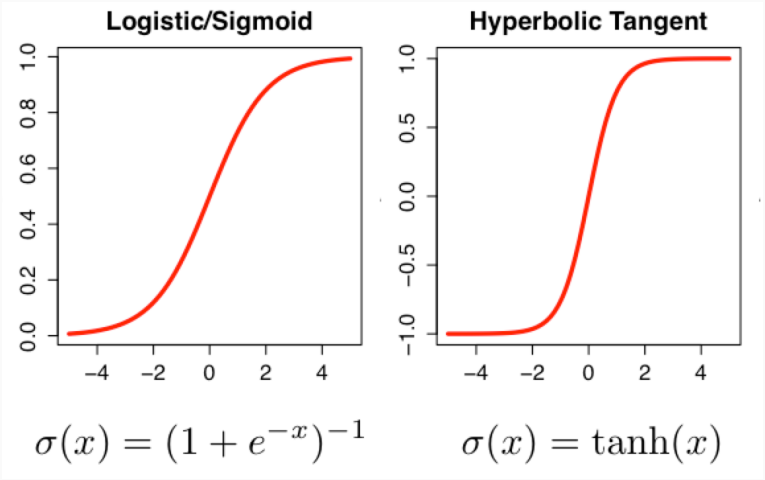
... and the number of layers can rapidly grow with the complexity of the function



A key idea to make neural networks practical: **soft-thresholding** ...

Soft-thresholding

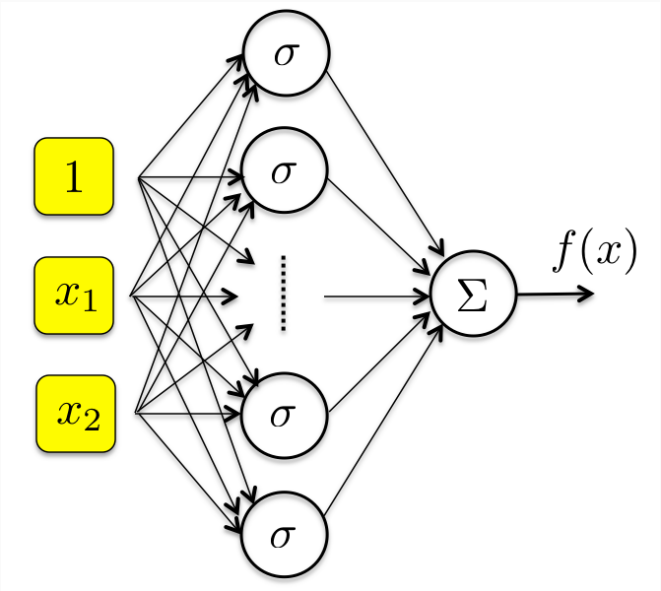
Replace hard-thresholding function σ by smoother functions



Theorem (Cybenko 1989) *Any continuous function f from $[0,1]^d$ to $\mathbb{R}$ can be approximated as a function of the form: $\sum_{j=1}^{N} \alpha_j \sigma(w_j^\top x + b_j)$, where σ is any sigmoid function.*

Soft-thresholding

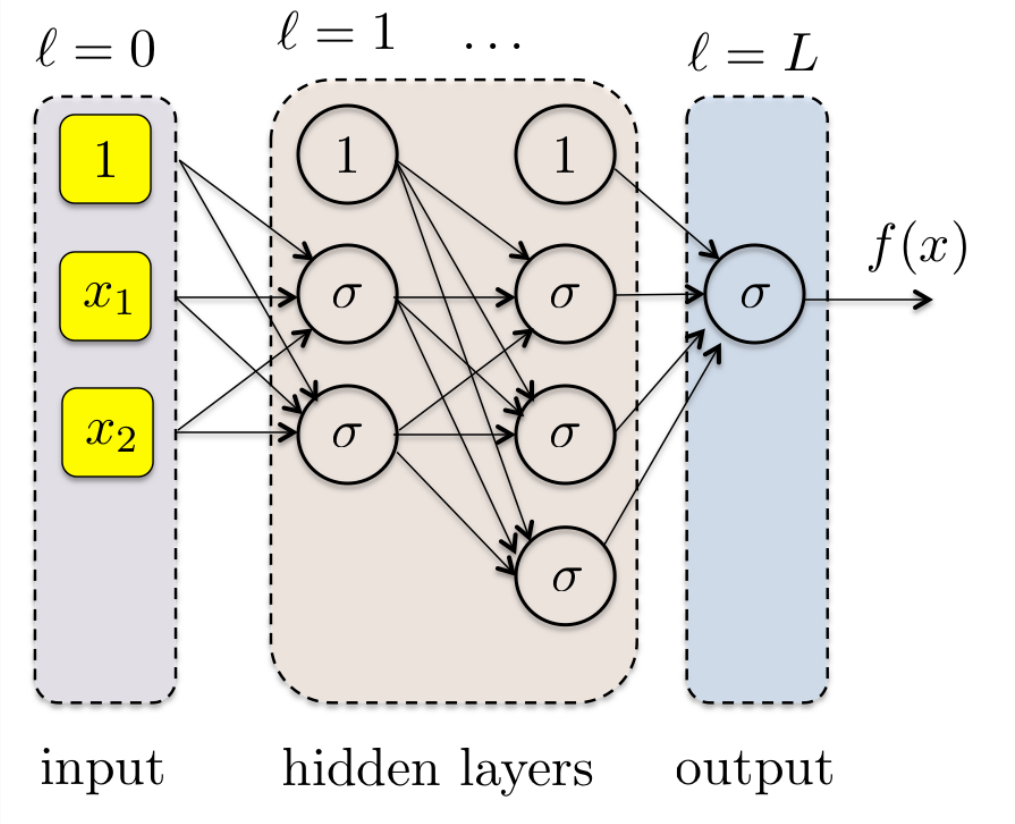
Cybenko's theorem tells us that f can be represented using a single hidden layer network ...



A non-constructive proof: how many neurones do we need? Might depend on f ...

Neural networks

A feedforward layered network (deep learning = enough layers)



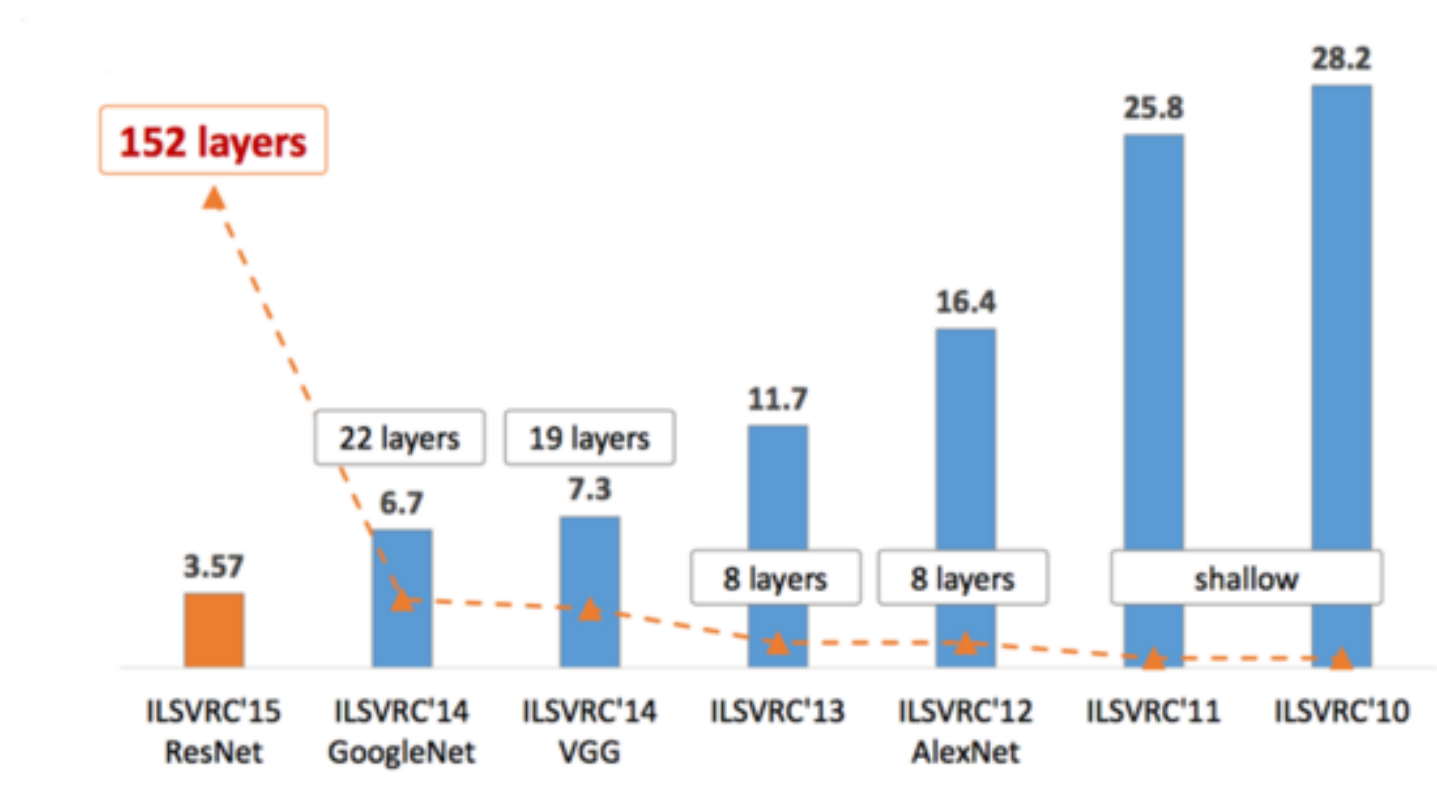
Deep Learning and the ILSVR challenge

Deep learning outperformed any other techniques in all major machine learning competitions (image classification, speech recognition and natural language processing)

The ImageNet Large Scale Visual Recognition Challenge (ILSVRC).

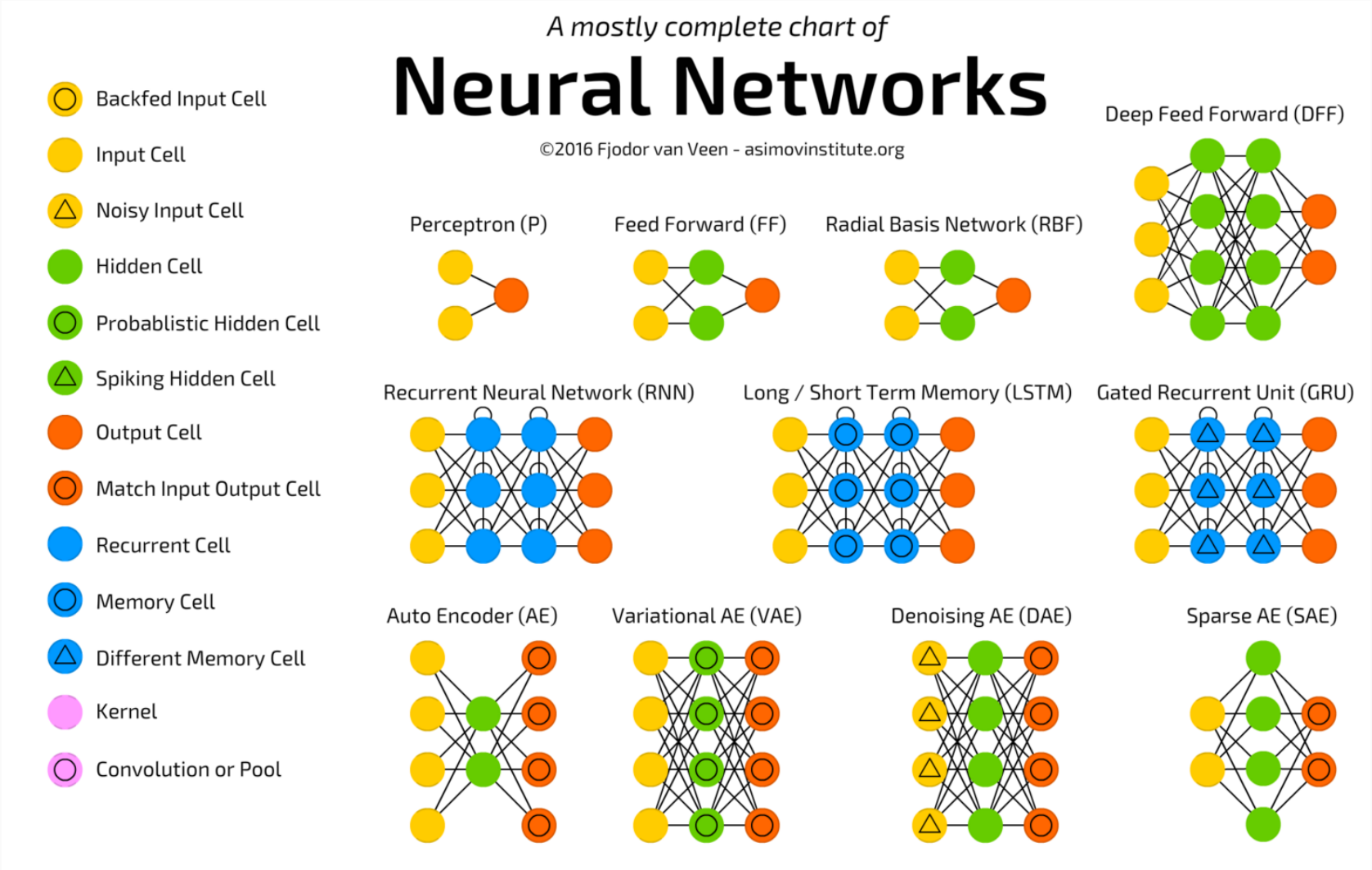
1. Training: 1.2 million images (227×227), labeled one out of 1000 categories
2. Test: 100.000 images (227×227)
3. Error measure: The teams have to predict 5 (out of 1000) classes and an image is considered to be correct if at least one of the predictions is the ground truth. [3]

ILSVR challenge

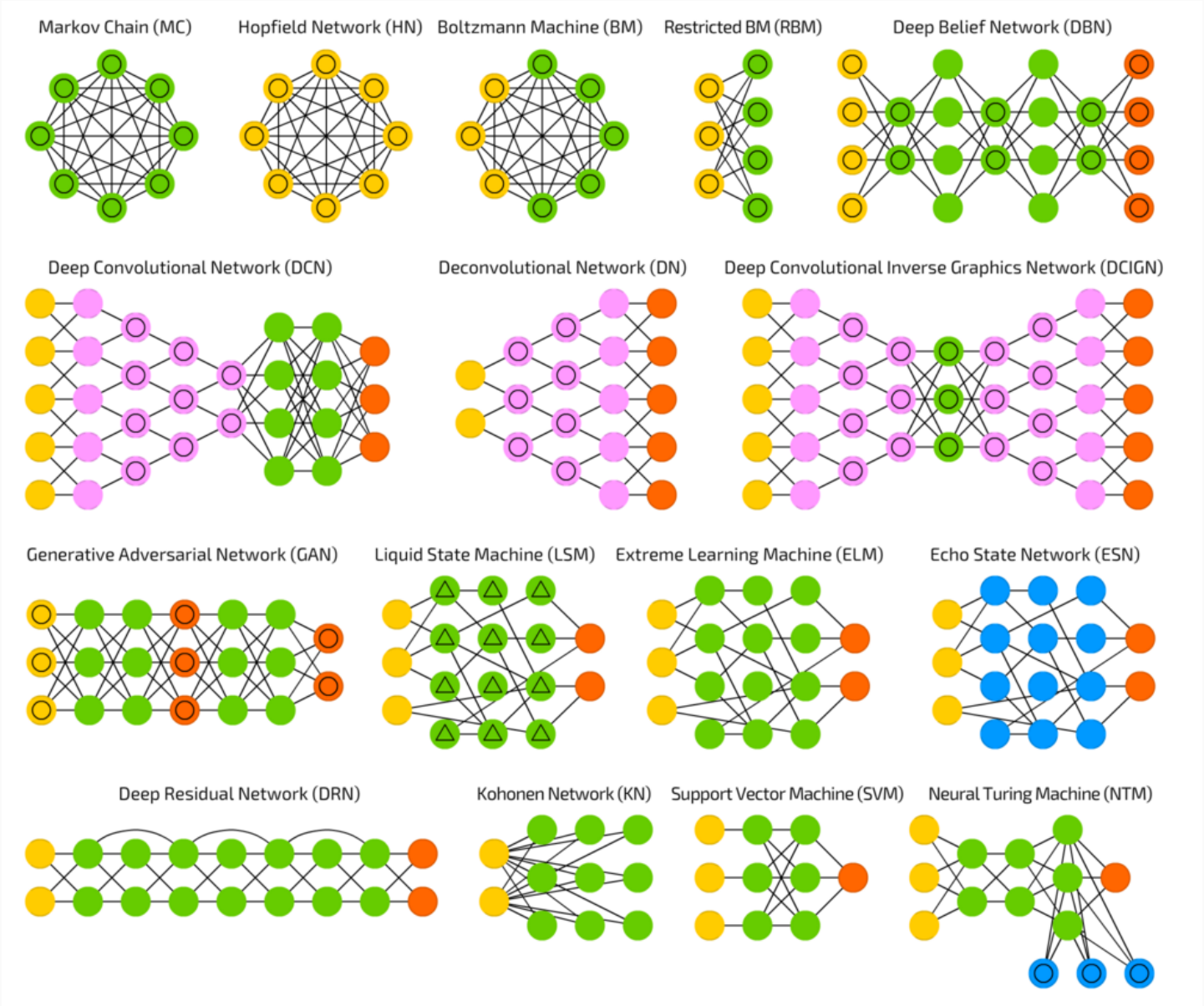


[1] From Stanford CS231n lecture notes

Architectures

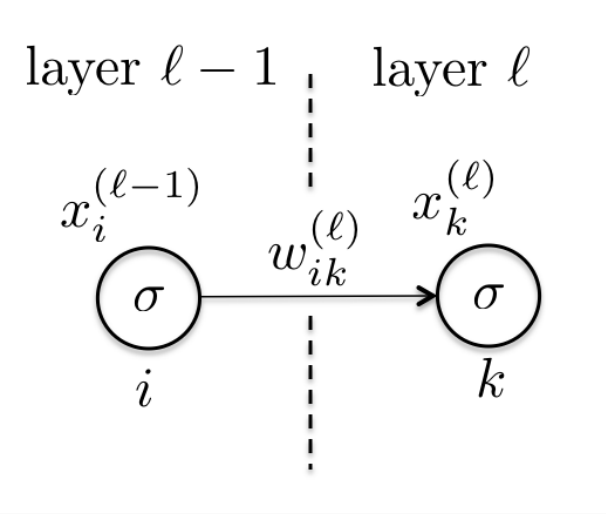


Architectures



Computing with neural networks

- Layer 0: inputs $x = (x_1^{(0)}, \ldots, x_d^{(0)})$ and $x_0^{(0)} = 1$
- Layer $1, \ldots, L-1$: hidden layer ℓ, $d^{(\ell)} + 1$ nodes, state of node i, $x_i^{(\ell)}$ with $x_0^{(\ell)} = 1$
- Layer L: output $y = x_1^{(L)}$



Signal at k: $s_k^{(\ell)} = \sum_{i=0}^{d^{(\ell-1)}} w_{ik}^{(\ell)} x_i^{(\ell-1)}$

State at k: $x_k^{(\ell)} = \sigma(s_k^{(\ell)})$

Output: the state of $y = x_1^{(L)}$

Training neural networks

The output of the network is a function of $\mathbf{w} = (w_{ij}^{(\ell)})_{i,j,\ell}$: $y = f_{\mathbf{w}}(x)$
We wish to optimise over $\mathbf{w}$ to find the most accurate estimation of the target function

Training data: $(X_1, Y_1), \ldots, (X_n, Y_n) \in \mathbb{R}^d \times \{-1, 1\}$

Objective: find $\mathbf{w}$ minimising the empirical risk:

$$E(\mathbf{w}) := R(f_{\mathbf{w}}) = \frac{1}{2n} \sum_{l=1}^{n} |f_{\mathbf{w}}(X_l) - Y_l|^2$$

Stochastic Gradient Descent

$E(\mathbf{w}) = \frac{1}{2n} \sum_{l=1}^{n} E_l(\mathbf{w})$ where $E_l(\mathbf{w}) := |f_{\mathbf{w}}(X_l) - Y_l|^2$
In each iteration of the SGD algorithm, only one function E_l is considered ...

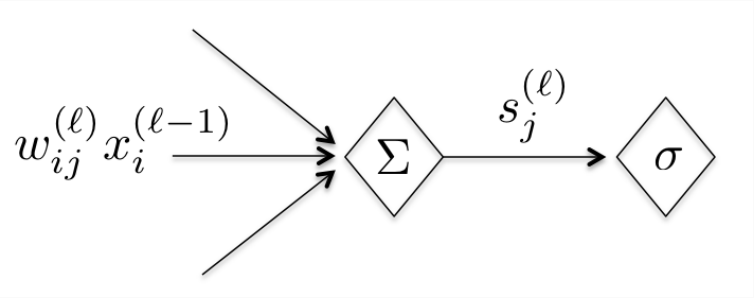
Parameter. learning rate $\alpha > 0$

1. **Initialization.** $\mathbf{w} := \mathbf{w}_0$
2. **Sample selection.** Select l uniformly at random in $\{1, \ldots, n\}$
3. **GD iteration.** $\mathbf{w} := \mathbf{w} - \alpha \nabla E_l(\mathbf{w})$, go to 2.

Is there an efficient way of computing $E_l(\mathbf{w})$?

Backpropagation

We fix l, and introduce $e(\mathbf{w}) = E_l(\mathbf{w})$.
Let us compute $\nabla e(\mathbf{w})$:



$$\frac{\partial e}{\partial w_{ij}^{(\ell)}} = \underbrace{\frac{\partial e}{\partial s_j^{(\ell)}}}_{:=\delta_j^{(\ell)}} \times \underbrace{\frac{\partial s_j^{(\ell)}}{\partial w_{ij}^{(\ell)}}}_{=x_i^{(\ell-1)}}$$

The sensitivity of the error w.r.t. the signal at node j can be computed recursively ...

Backward recursion

Output layer. $\delta_1^{(L)} := \frac{\partial e}{\partial s_1^{(L)}}$ and $e(\mathbf{w}) = (\sigma(s_1^{(L)}) - Y_l)^2$

$$\delta_1^{(L)} = 2(x_1^{(L)} - Y_l)\sigma'(s_1^{(L)})$$

From layer ℓ to layer $\ell - 1$.

$$\delta_i^{(\ell-1)} := \frac{\partial e}{\partial s_i^{(\ell-1)}} = \sum_{j=1}^{d^{(\ell)}} \underbrace{\frac{\partial e}{\partial s_j^{(\ell)}}}_{:=\delta_j^{(\ell)}} \times \underbrace{\frac{\partial s_j^{(\ell)}}{\partial x_i^{(\ell-1)}}}_{=w_{ij}^{(\ell)}} \times \underbrace{\frac{\partial x_i^{(\ell-1)}}{\partial s_i^{(\ell-1)}}}_{=\sigma'(s_i^{(\ell-1)})}$$

Summary.

$$\frac{\partial E_l}{\partial w_{ij}^{(\ell)}} = \delta_j^{(\ell)} x_i^{(\ell-1)}, \quad \delta_i^{(\ell-1)} = \sum_{j=1}^{d^{(\ell)}} \delta_j^{(\ell)} w_{ij}^{(\ell)} \sigma'(s_i^{(\ell-1)})$$

Backpropagation algorithm

Parameter. Learning rate $\alpha > 0$

Input. $(X_1, Y_1), \ldots, (X_n, Y_n) \in \mathbb{R}^d \times \{-1, 1\}$

1. **Initialization.** $\mathbf{w} := \mathbf{w}_0$
2. **Sample selection.** Select l uniformly at random in $\{1, \ldots, n\}$
3. **Gradient of E_l.**
   - $x_i^{(0)} := X_{li}$ for all $i = 1, \ldots d$
   - Forward propagation: compute the state and signal at each node $(x_i^{(\ell)}, s_i^{(\ell)})$
   - Backward propagation: propagate back Y_l to compute $\delta_i^{(\ell)}$ at each node and the partial derivative $\frac{\partial E_l}{\partial w_{ij}^{(\ell)}}$
4. **GD iteration.** $\mathbf{w} := \mathbf{w} - \alpha \nabla E_l(\mathbf{w})$, go to 2.