Time Series Analysis

Time series, Reinforcement Learning, RNN, LSTM

Nicolas Keriven
CNRS, IRISA, Rennes

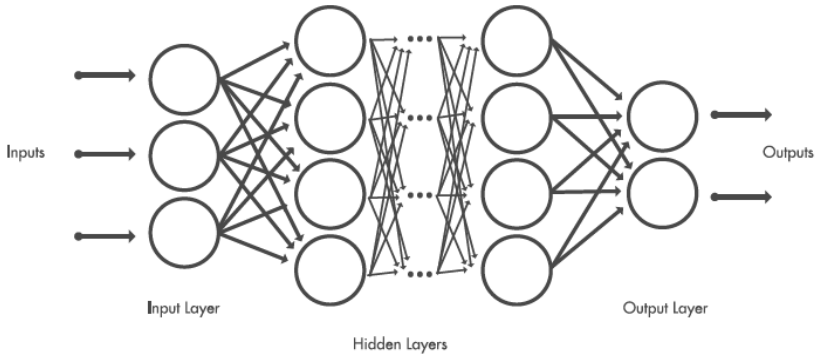
(material from Florent Chatelain, Olivier Michel)

ENSTA 2023

Table of Contents

Time series data

- In classical ML, inputs are assumed iid
- Learning is done by minimizing a Loss function, often by gradient descent (BP for Neural Networks)



But

- Many data come as time series $x_1, \ldots, x_t, \ldots$
- Examples :
  1. Weather forecasting
  2. Natural Language Processing (NLP) : speech recognition, translation, etc.
  3. Games : Chess, Go, Starcraft 3...
  4. Autonomous driving, etc.
- Time series data :
  1. may come online (weather) or all at once (NLP)
  2. are generally correlated with their predecessor (and successor)
  3. may depend on an underlying process with many characteristics

Table of Contents

Recurrent Neural Networks (RNN)

RNNs are Neural Networks for time series. They take many inspirations from control theory, eq. diff. modelization, etc.
Jeffrey Elmann (90) : "Simple Recurrent Neural Network" (one hidden layer) :

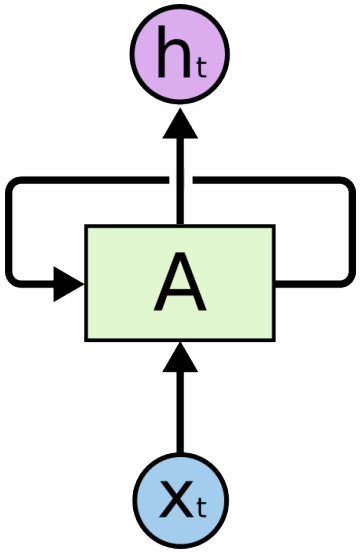
- Introduce some Neuron Internal State (NIS) h_t, depending on the current input, and past NIS :

$$\begin{cases} h_t = \sigma(W_{hx}x_t + W_{hh}h_{t-1} + b_h) & \text{(hidden layer on current } x_t \text{ and previous } h_{t-1}) \\ y_t = f_o(W_{yh}h_t + b_y) & \text{Output at time } t \end{cases}$$

- This introduces a loop (like in many system from control theory), allowing the information to persist (in theory)

Notations : σ= activation function (originally sigmoid) ; h_t=NIS ; x_t=input vector at t ; y_t=output vector ; W_{kl}=weights matrices, $b_\star$=biases, f_o=output function relating y_t to h_t.

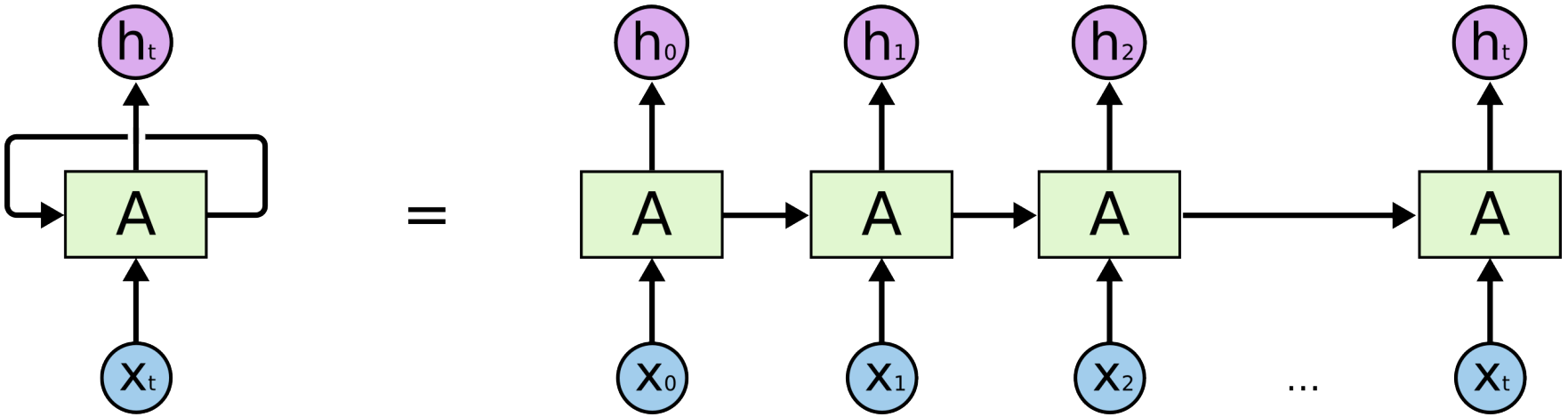
RNN cell :



- The NIS h_t is :
  - outputted and processed to give $y_t = f_o(W_{yh}h_t + b_y)$. May take different forms, depending on the learning task (linear operator for regression or e.g. Softmax for classification)
  - reinjected into the cell at each time step
- Each neuron (or cell) is a MLP with a single hidden layer
- The loop is applied on the hidden layer

Unrolling the RNN

An alternate representation of Elman's equations, accounting for time evolution is :



The RNN can be "thought of" as a Deep Neural Network, with all layers having identical weight matrices and bias, or as multiple copies of the same network, each passing a message (h_{n-1}) to its successor.

- RNNs can be trained by BP (usually called Backpropagation Through Time (BPTT) in this case, but same algorithm)
- One of the historical, most successful variant of RNNs are called Long-Short Term Memory (LSTM, Hochreiter and Schmidhuber 97)

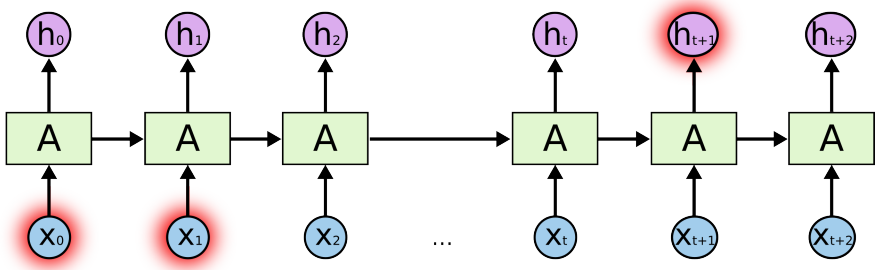
Table of Contents

Motivation

Two main motivations to go beyond "vanilla" RNNs : vanishing/exploding gradients and catastrophic forgetting.

Catastrophic forgetting

- In practice, RNNs have a hard time modelling far-ranged interactions between distant time steps, even though this is perfectly possible. This is crucial in some applications ! This is the so-called catastrophic forgetting phenomenon.



Vanishing/exploding gradient

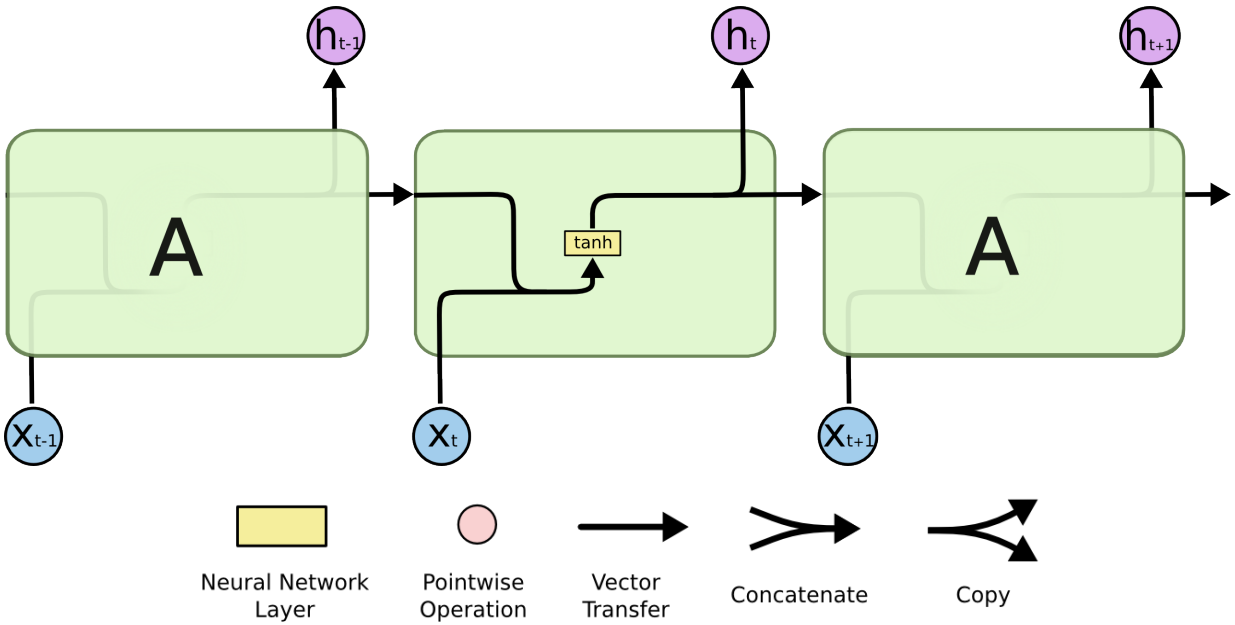
- Vanishing or exploding gradients are a problem for all DNNs, but particularly for RNNs, since, when "unrolled", the number of layers is the size of the time series !

LSTM

Material is from `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

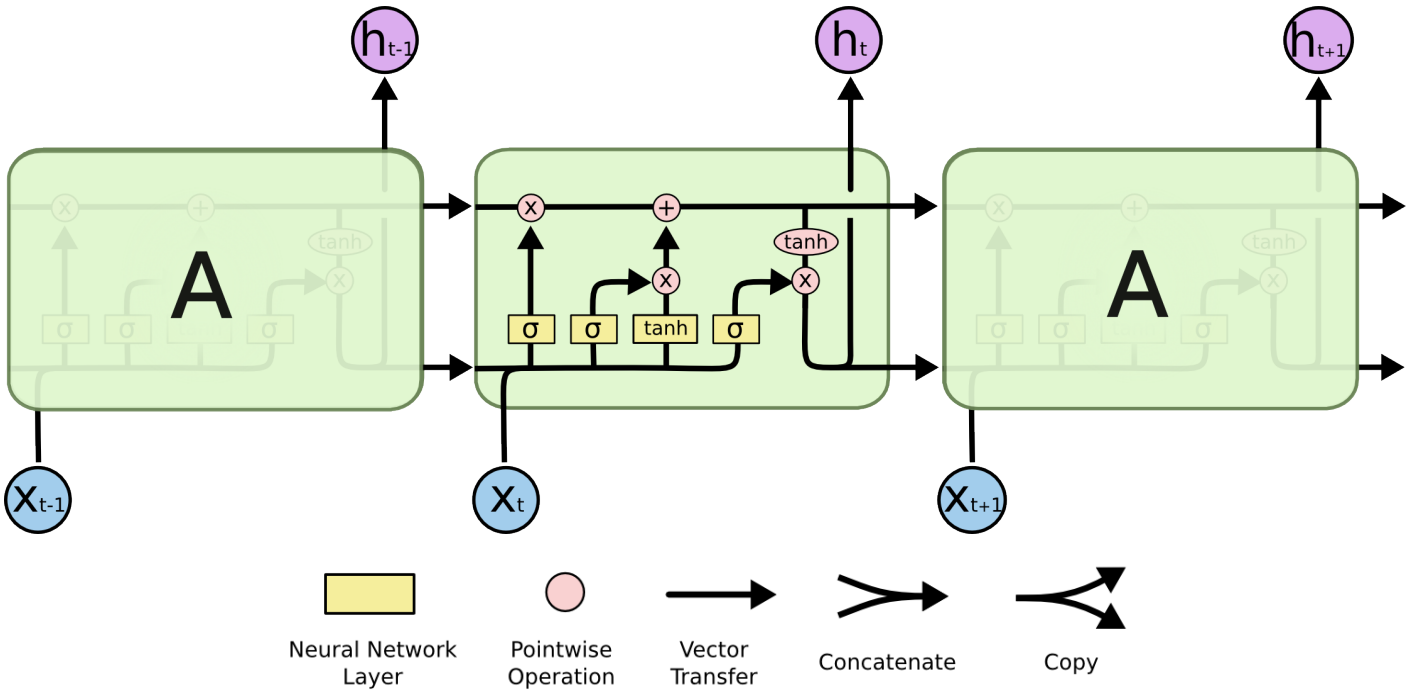
- Just a particular case of RNNs : keeps the form of a repeating chain of similar modules

- Recall the structure of a simple RNN : each cell is an MLP with 1 hidden layer



LSTM

- Instead of a single hidden layer MLP, a LSTM will count four (4) layer interacting in a special way. This increase in complexity has the unique purpose of avoiding vanishing gradient problems!



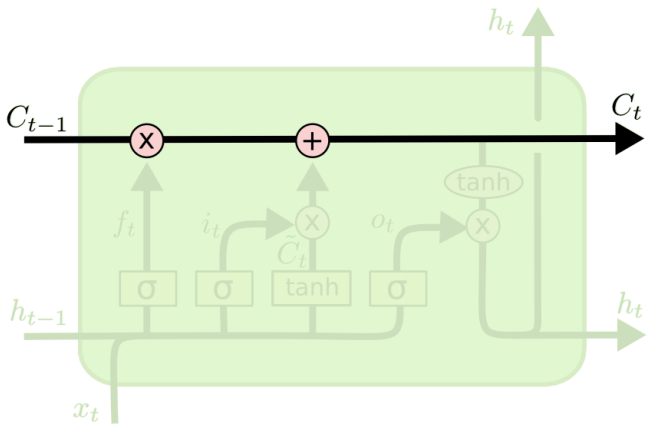
Note that each line in the diagram above carries an entire vector.

- Each of the yellow boxes represents a learned neural network

LSTM as an information conveyor belt

The key point of LSTMs is the presence of a cell state C_t
(in addition to h_t)

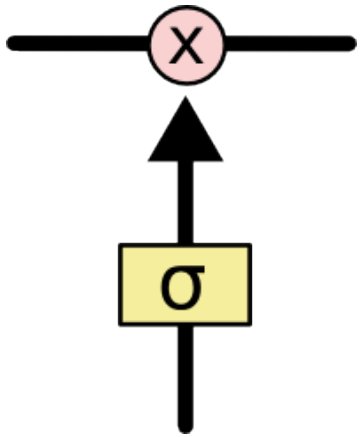
- only lightly modified at each layer, facilitating long-range interactions
- does not go through any non-linearity σ, to mitigate vanishing/exploding gradients.



- The first operator is a point wise multiplication : this will allow to cancel (remove via a multiplication w/ a very small value) irrelevant information.
- The second operator is a poinwise addition : this allows to add information to the cell state
- These operations are regulated by structures called **GATES**

Gates

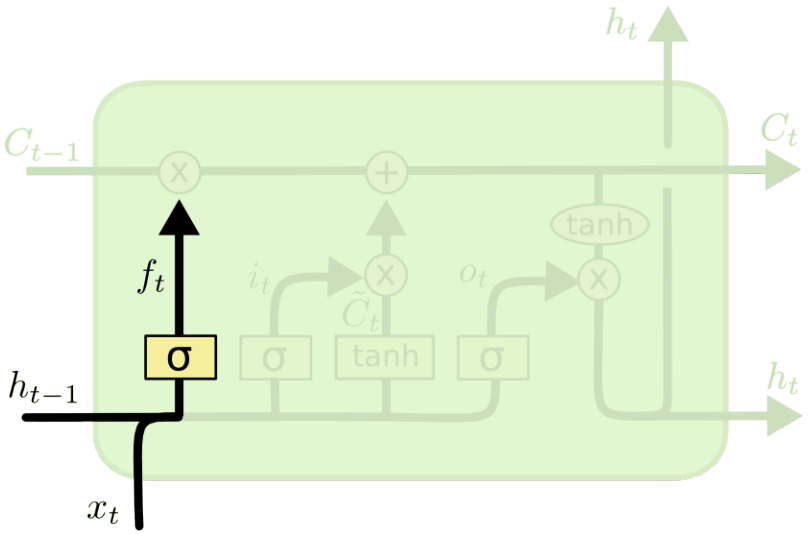
- Gates regulate how information is either cancelled or let through. Gates composed out of sigmoid neural net layer (whose outputs component values are between 0 and 1).



- A gate output component value of 0 means "stop this information"
- A gate output component value of 1 means "let this information through"

Forget gate

The purpose of the forget gate is to decide what information must be thrown away from the cell state, by considering both the new input x_t and the previous time internal state h_{t-1} :

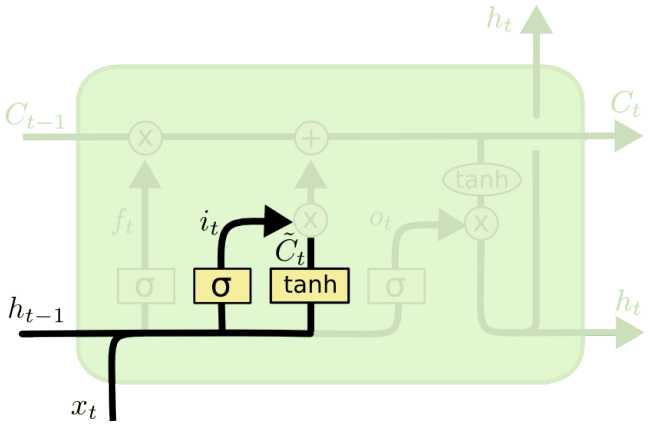


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Input gate

The input gate decides which information to add to the cell state.

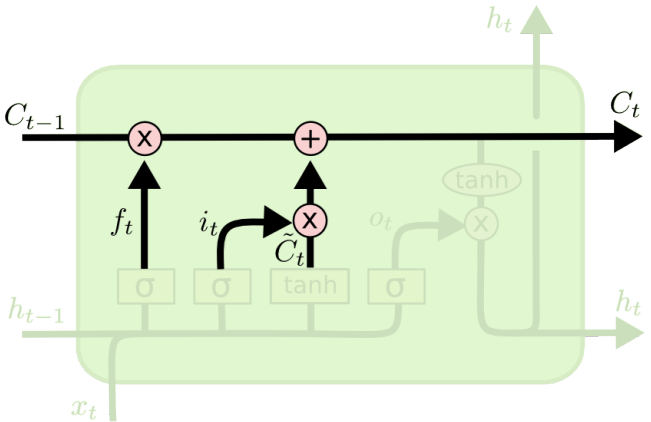
- First decide which values should be updated by a sigmoid gate i_t
- Next, a 'tanh' layer creates a new candidate $\tilde{C}_t$ to add to the state



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

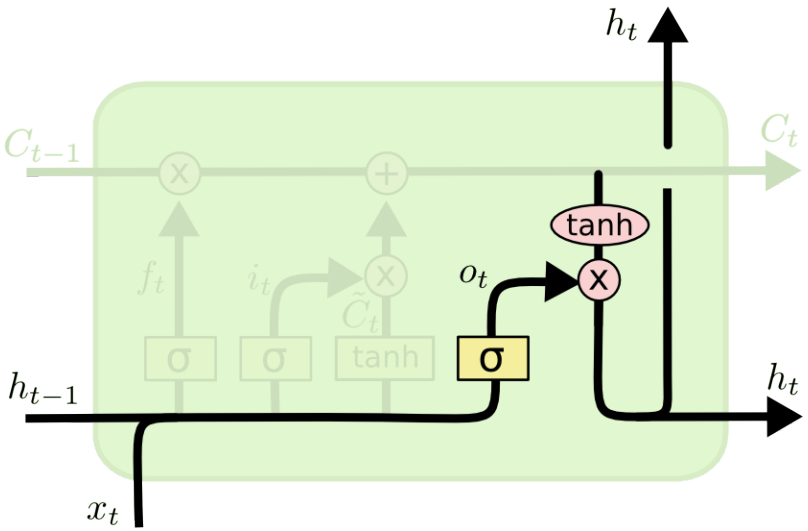
- Third, combine the two previous results to create the state update :



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Output gate

The output gate goes the other way : its goal is to compute a value h_t with information from the current input x_t, the cell state C_t and the previous hidden state h_{t-1} :



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

LSTM : Summary

- A LSTM network is a recurrent network or equivalently a chain of identical modules or cells
- In comparison to simple RNN, LSTM contains not only a hidden state but also a cell state
- The cell state is transferred between modules with no non-linearity, thus avoiding vanishing gradient problems
- The cell state is only lightly modified, allowing long term memory storing
- The structure of each cell is based on three gates :
  1. A forget gate, deciding which internal information to cancel
  2. An input gate, deciding which information to store in the internal cell state
  3. An output gate, combining last input, previous hidden state and updated cell state to provide the new hidden state

LSTM variants

As always, many variants of the preceding structure may be found in the literature, among which some popular are just cited here :

- Adding "Peephole connections" : all three or only a subset of the NNs (σ) also take C_{t-1} as inputs.
- Some authors proposed to couple forget and input gates : all forget info is also replaced at the input stage.
- Gated Recurrent Units (GRU) -2014- : merge input and ouput to make an 'update gate', merge C_t and h_t

Table of Contents

Reinforcement learning (RL)

- RL is the "third" type of ML problems / algorithms, with supervised and unsupervised learning
- RL treats of time series data
- In RL, an agent interact with an environment, and receive rewards
- RL aims to maximize the total sum of rewards
- Extremely *hot topic* right now ! Many state-of-the-art outstanding challenges are RL problems

high reward

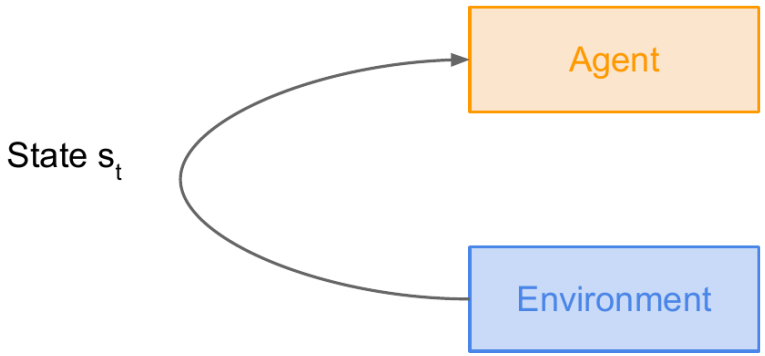
low reward

Basis of RL

- An RL problem is formed by sets of :
  - states $s \in \mathcal{S}$
  - actions $a \in \mathcal{A}$
  - rewards $r \in \mathcal{R}$
- At each time step t :
  - The *agent* observe the state of the *environment* s_t
  - The agent takes an action a_t
  - The agent receives a reward r_t and observe the next state s_{t+1}

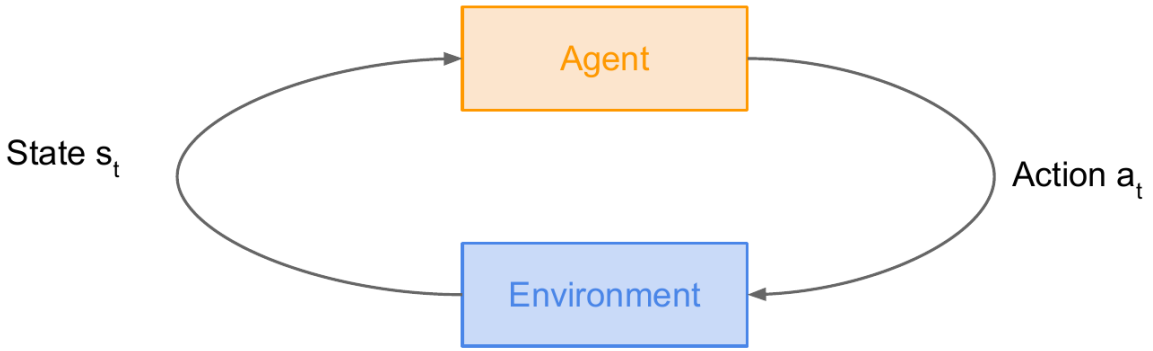
Basis of RL

- An RL problem is formed by sets of :
  - states $s \in \mathcal{S}$
  - actions $a \in \mathcal{A}$
  - rewards $r \in \mathcal{R}$
- At each time step t :
  - The *agent* observe the state of the *environment* s_t
  - The agent takes an action a_t
  - The agent receives a reward r_t and observe the next state s_{t+1}



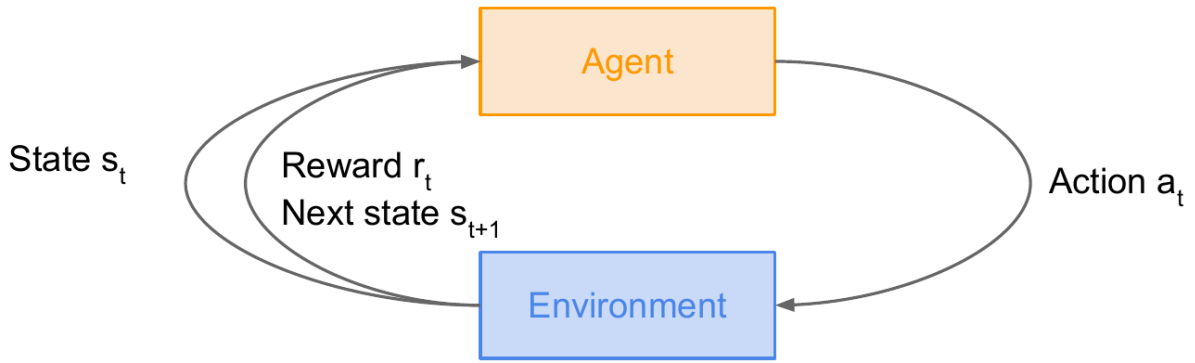
Basis of RL

- An RL problem is formed by sets of :
  - states $s \in \mathcal{S}$
  - actions $a \in \mathcal{A}$
  - rewards $r \in \mathcal{R}$
- At each time step t :
  - The *agent* observe the state of the *environment* s_t
  - The agent takes an action a_t
  - The agent receives a reward r_t and observe the next state s_{t+1}



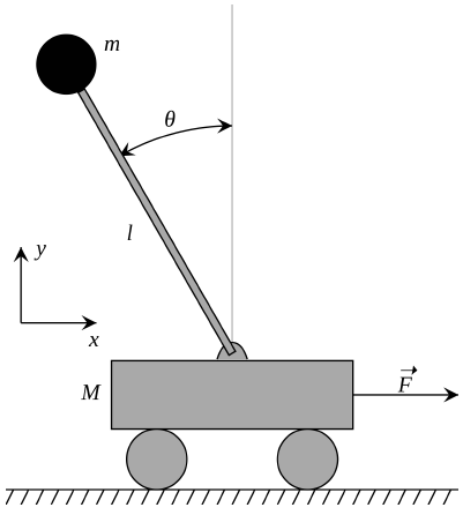
Basis of RL

- An RL problem is formed by sets of :
  - states $s \in \mathcal{S}$
  - actions $a \in \mathcal{A}$
  - rewards $r \in \mathcal{R}$
- At each time step t :
  - The *agent* observe the state of the *environment* s_t
  - The agent takes an action a_t
  - The agent receives a reward r_t and observe the next state s_{t+1}



RL : examples

Cart-Pole Problem

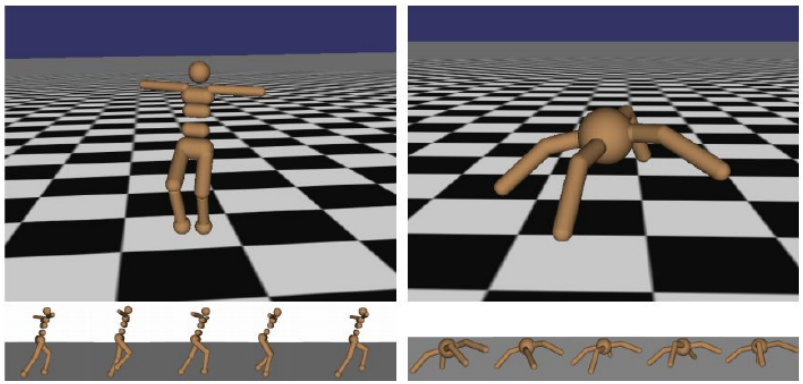


Objective: Balance a pole on top of a movable cart

State: angle, angular speed, position, horizontal velocity
Action: horizontal force applied on the cart
Reward: 1 at each time step if the pole is upright

RL : examples

Robot Locomotion



Objective: Make the robot move forward

State: Angle and position of the joints
Action: Torques applied on joints
Reward: 1 at each time step upright + forward movement

RL : examples

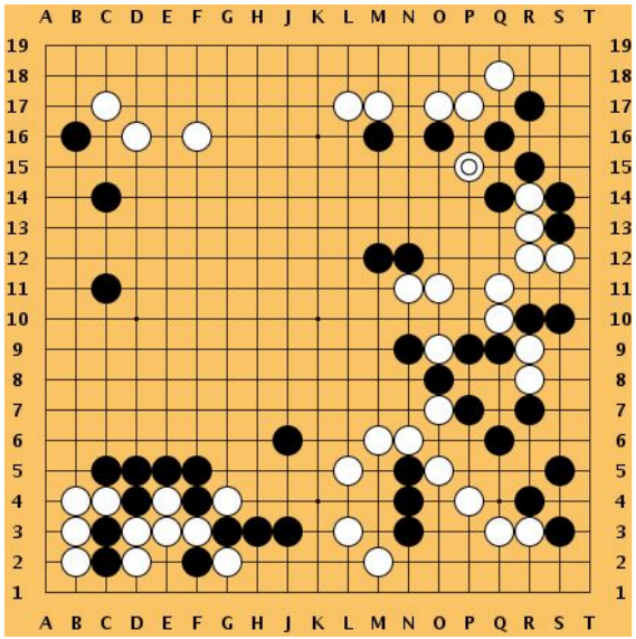
Atari Games

Objective: Complete the game with the highest score

State: Raw pixel inputs of the game state
Action: Game controls e.g. Left, Right, Up, Down
Reward: Score increase/decrease at each time step

RL : examples

Go



Objective: Win the game!

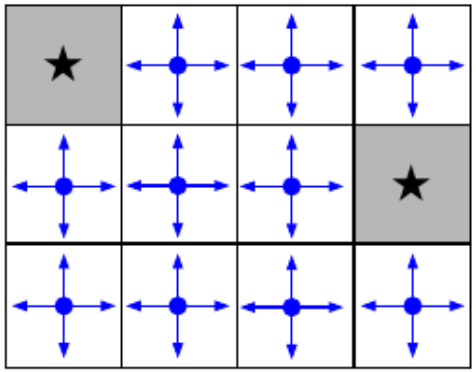
State: Position of all pieces
Action: Where to put the next piece down
Reward: 1 if win at the end of the game, 0 otherwise

RL : mathematical modelization

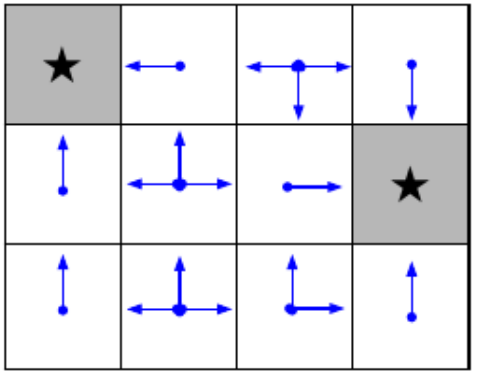
Markov Decision Process (MDP)

- Sets of states $\mathcal{S}$, actions $\mathcal{A}$, rewards $\mathcal{R}$
- Transition probability $\mathbb{P}$ of observing the next state
  - Markov hypothesis : s_{t+1} only depends on s_t and a_t : $s_{t+1} \sim \mathbb{P}(\cdot|s_t, a_t)$
- Reward probability distribution $r_t \sim \mathbb{P}_R(\cdot|s_t, a_t, s_{t+1})$ (may not depend on s_{t+1})
- The objective is to learn a policy π to define the action to take in each state $a_t \sim \pi(\cdot|s_t)$
- The goal is (usually) to maximize an expected cumulative discounted reward : $\max_\pi \mathbb{E} \sum_{t=0}^{\infty} r_t \gamma_t$, eg $\gamma_t = 1_{t \leq T}$ or $\gamma_t = \gamma^t$ (finite or infinite *horizon*)

MDP



Random Policy



Optimal Policy

- MDP are simplified : in practice s_{t+1} often depends on all that happened before, and π *should* depend on more than the current state s_t
  - More computationnally / theoretically tractable
  - Less parameters = less data-hungry
- As computers become more powerful and data more available, bigger models emerged (see eg Transformers in next session)

Frameworks

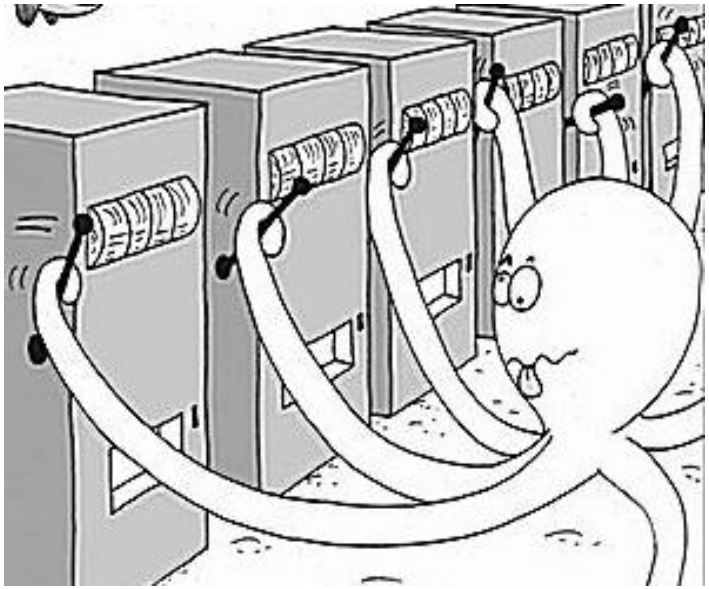
Different frameworks :

- Nothing is known in advance : the optimal policy must be learned *at the same time* that the cumulative reward must be optimized
  - eg multi-armed bandit (see next), finance...
  - trade-off : exploration (learning the policy) and exploitation (collecting rewards)

- The training / inference are separated : the policy is learned, then exploited
  - eg playing games
  - training is pure exploration, inference is pure exploitation
- Often a bit of both ! Model is pre-trained in a generic context, then adjusted during application
  - eg, playing games, adjusting for our opponent

Multi-armed bandit

- A simplified and historic example of reinforcement learning.
- A multi-armed bandit has K arms. Pulling an arm gives a random reward. Some arms will give higher expected rewards, but they are unknown in advance.
- Formally, an MDP with no state space
- The goal is to maximize the cumulative reward, knowing nothing in advance : trade-off between exploration (estimating the expected rewards) and exploitation (pulling the best arm up until now)



Multi-armed bandit : two simple algorithms

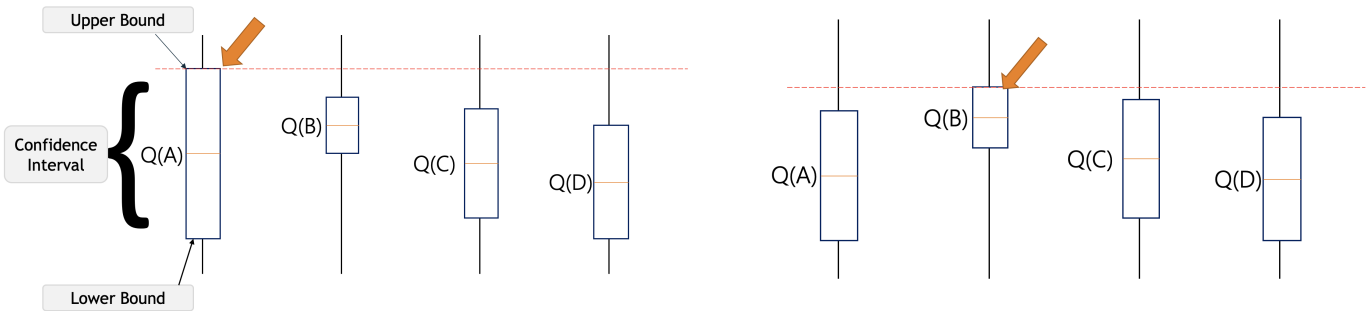
ϵ-greedy algorithm (Sutton, Barto 98)

- For each arm k, maintain an empirical average reward $\hat{\mu}_k = \frac{1}{N_k} \sum_{i=1}^{N_k} r_{ki}$, where N_k is the number of times arm k has been pulled and r_{ki} are the rewards observed from arm k,
- At each step :
  - with probability ϵ : pull a **random arm** (*explore*)
  - with probability $1 - \epsilon$: pull the arm with the **highest** $\hat{\mu}_k$ (*exploit*)
- Pbm : continues to explore even when the arms are very-well estimated ! (some fix : decreasing ϵ)

Multi-armed bandit : two simple algorithms

Upper Confidence Bound (UCB) (Auer et al., 02)

- for each arm, maintain $\hat{\mu}_k$ as well as a confidence interval $[\hat{\mu}_k - \sigma_k, \hat{\mu}_k + \sigma_k]$, where $\sigma_k = O(\sqrt{\frac{\log(t)}{N_k}})$ based on probabilistic modelling assumptions
- At each step, draw the arm with the highest upper bound of the confidence interval $\hat{\mu}_k + \sigma_k$
- Explanation : as we pull arms, their expected reward will be better estimated. Arms with a poor $\hat{\mu}_k$ but high σ_k (that has been not pulled enough to be well-estimated) will still be explored.



Generic approach : Q-learning

- Consider the expected cumulative reward with exponential weights $\mathbb{E}[\sum_{t\geq 0}\gamma^t r_t]$

- The Q-value function starts at state s and initial action a, then follow policy π :

$$Q^\pi(s,a) = \mathbb{E}[\sum_{t\geq 0}\gamma^t r_t \mid s_0 = s, a_0 = a, \pi]$$

  Denoting $Q^\star(s,a) = Q^{\pi^\star}(s,a)$, we have $\pi^\star(s) = \arg\max_a Q^\star(s,a)$.

- Key : $Q^\star$ satisfy the Bellman equation, a fixed-point equation :

$$Q^\star(s,a) = \mathbb{E}_{s',r\sim\mathbb{P}(\cdot|s,a)}[r + \gamma \max_{a'} Q^\star(s',a')]$$

Q-learning

- Idea of Q-learning : maintain a table of Q-values. At each step, compute an approximate "gradient" (finite difference) and use gradient ascent :

$$\begin{cases} \text{grad} = \underbrace{r_t + \gamma \overbrace{\max_a Q(s_{t+1}, a)}^{\text{using known values}}}_{\text{target value}} - Q(s_t, a_t) \\ Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot \text{grad} \end{cases}$$

- Problems :
  - very high dimensional ! Many steps are required to even visit $Q(s, a)$ once...
  - ☞ Sol : using a function approximation Q_θ and perform gradient updates on θ instead (eg a deep NN : deep Q-learning !)
  - successive samples are *correlated*, and we use r_t, s_{t+1} as a (poor !) proxy for the expectation $\mathbb{E}$ in Bellman's equation. Can lead to bad feedback loop.
  - ☞ Sol : use experience replay, a (random or not) sample of many previous actions instead of only the most recent one.

Policy gradient

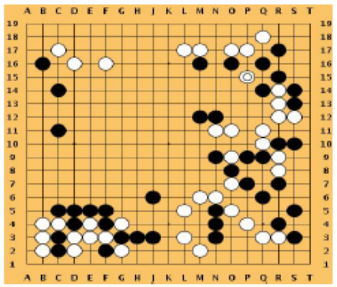
- Instead of parametrizing Q, directly parametrize the policy π_θ
- We must **differentiate an expectation**... Use a classic trick :

$$\begin{aligned}\nabla_\theta \mathbb{E}_{p_\theta} f(x) &= \int f(x) \nabla_\theta p_\theta(x) dx \\ &= \int f(x) p_\theta(x) \frac{\nabla_\theta p_\theta(x)}{p_\theta(x)} dx \\ &= \int f(x) p_\theta(x) \nabla_\theta [\log p_\theta(x)] dx = \mathbb{E}[f(x) \nabla_\theta \log p_\theta(x)]\end{aligned}$$

the final expectation can be estimated by sampling many trajectories (Monte-Carlo sampling).

Many, many other tricks. Big modern systems use a combination of everything !

More policy gradients: AlphaGo



Overview:

- Mix of supervised learning and reinforcement learning
- Mix of old methods (Monte Carlo Tree Search) and recent ones (deep RL)

How to beat the Go world champion:

- Featurize the board (stone color, move legality, bias, ...)
- Initialize policy network with supervised training from professional go games, then continue training using policy gradient (play against itself from random previous iterations, +1 / -1 reward for winning / losing)
- Also learn value network (critic)
- Finally, combine combine policy and value networks in a Monte Carlo Tree Search algorithm to select actions by lookahead search

[Silver et al., Nature 2016]