Introduction to Reinforcement Learning

AI Frameworks

Brendan Guillouet

2019

Institut National des Sciences Appliquées

ML Python Libraries

scikit learn

Keras

Gym

TensorFlow

gensim

Viz' Python

matplotlib Version 3.1.1

seaborn

plotly

Python Environme

PC

jupyter

ANACONDA

Other Tools &

docker

APACHE Spark™

Google Cloud Platform

git

DATA SCIENCE TOOLS

ML Python Libraries

Viz' Python

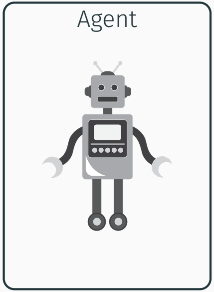
Python Environme

Other Tools &

Table of contents

Introduction RL

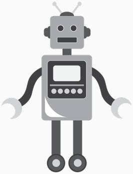
In reinforcement Learning, an agent makes observations of the states of an environment.



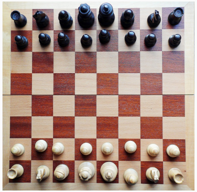
Definition

In reinforcement Learning, an **agent** makes **observations** of the **states** of an **environment**.

Agent

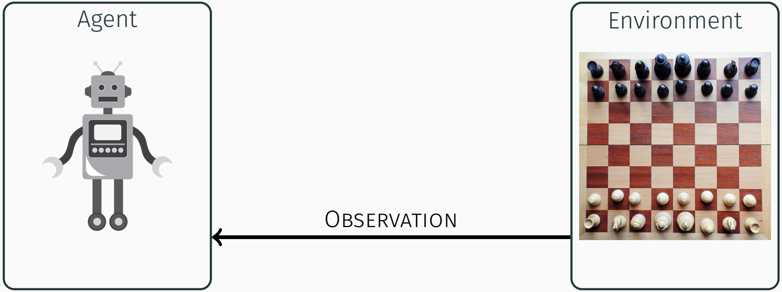


Environment



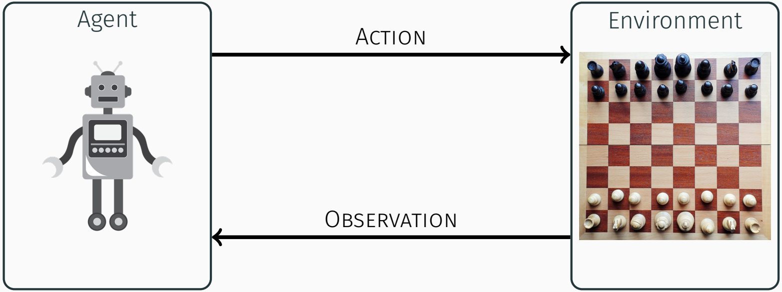
Definition

In reinforcement Learning, an **agent** makes **observations** of the **states** of an **environment**.



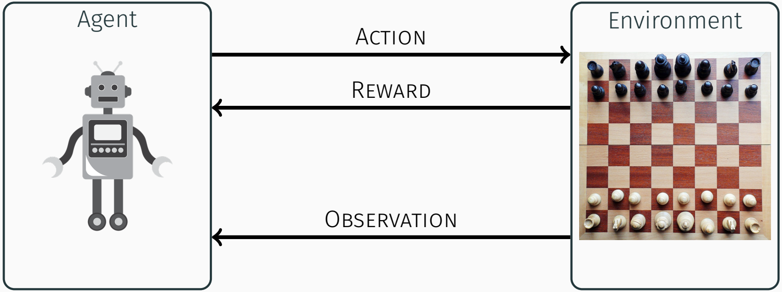
Observation Locations of each pieces on the chessboard. $(32 \times (id, x, y))$.

In reinforcement Learning, an **agent** makes **observations** of the **states** of an **environment**.
It takes **actions** within it,



Observation	Locations of each pieces on the chessboard. $(32 \times (id, x, y))$.
Action	Move piece p from $(x_p, y_p), (x_q, y_q)$

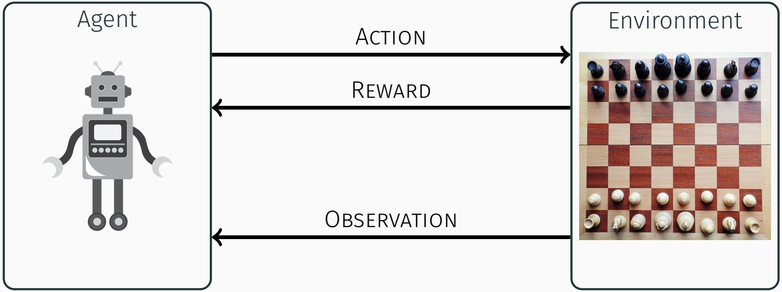
In reinforcement Learning, an **agent** makes **observations** of the **states** of an **environment**. It takes **actions** within it, and in return it receives **rewards**.



Observation	Locations of each pieces on the chessboard. $(32 \times (id, x, y))$.
Action	Move piece p from $(x_p, y_p), (x_q, y_q)$
Reward	positive if a pieces has been captured (for example)

Definition

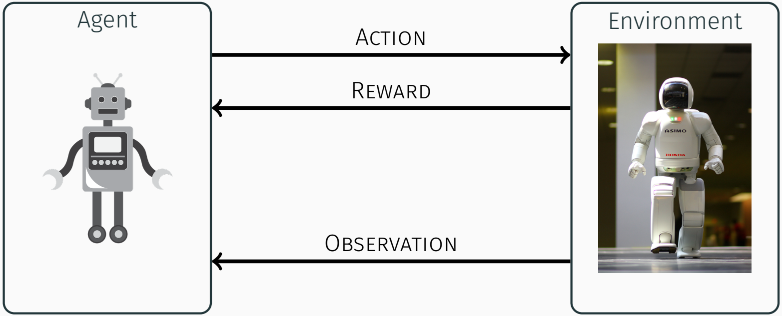
In reinforcement Learning, an **agent** makes **observations** of the **states** of an **environment**.
It takes **actions** within it, and in return it receives **rewards**.



Observation	Locations of each pieces on the chessboard. $(32 \times (id, x, y))$.
Action	Move piece p from $(x_p, y_p), (x_q, y_q)$
Reward	positive if a pieces has been captured (for example)
New Observation	New Locations of each pieces on the chessboard.

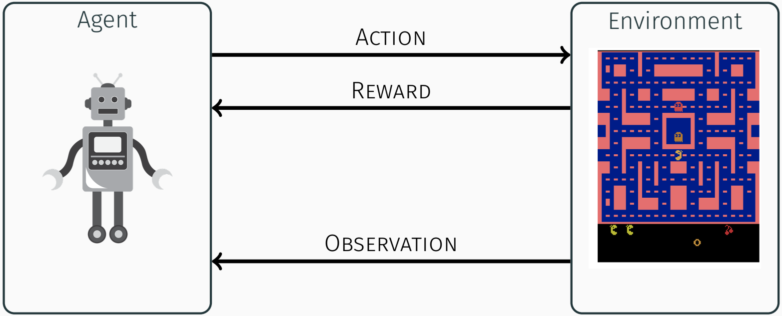
Examples

Walking robot



Observation	Measurement of different sensor.
Action	Move a member (continous)
Reward	• positive when it approaches the target destination, • negative when it wastes time, falls down etc..
New Observation	New measurement of sensors.

Pac Man



Observation	The image itself.
Action	left/right/up/down/center
Reward	game point
New Observation	The New image

When using reinforcement learning you need to defined the following objects :

- An AGENT : The one who will takes **action** within the **environment**.
- An ENVIRONMENT : Where the **action** will be taken.
  - The **state** of the environment defines it after each **action**.
  - The **observation** is what we will used from the state to decide the next **action** to take.
- Actions : A list of possible actions that will affect the **environment** and produce new **state**.
- Reward : A numerical value which reveal how positive or negative the **action** taken is.

When using reinforcement learning you need to defined the following objects :

- An AGENT : The one who will takes **action** within the **environment**.
- An ENVIRONMENT : Where the **action** will be taken.
  - The **state** of the environment defines it after each **action**.
  - The **observation** is what we will used from the state to decide the next **action** to take.
- ACTIONS : A list of possible actions that will affect the **environment** and produce new **state**.
- REWARD : A numerical value which reveal how positive or negative the **action** taken is.

$\Longrightarrow$ You can't apply reinforcements learning if you're not able to define these objects properly !

Objective

- Maximize the **long-term** rewards.
  - *Do not pull all the effort on capturing the queen if it means losing all your pieces.*

How? There are two main approaches.

- Policy-Based : Look for the optimal **Policy**,
  - *i.e the best action to take for each observation.*
- Value-Based : Look for the optimal **Reward**
  - *learns to estimate the expected rewards for each action in each state, then uses this knowledge to decide how to act.*

NB : Methods like **Actor-Critic**, try to optimize both policy and rewards.

Objective

- Maximize the **long-term** rewards.
  - *Do not pull all the effort on capturing the queen if it means losing all your pieces.*

How? There are two main approaches.

- **Policy-Based** : Look for the optimal **Policy**,
  - *i.e the best action to take for each observation.*
- Value-Based : Look for the optimal **Reward**
  - *learns to estimate the expected rewards for each action in each state, then uses this knowledge to decide how to act.*

NB : Methods like **Actor-Critic**, try to optimize both policy and rewards.

Policy-Based Methods

Policy

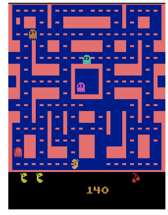
Policy : The algorithm used by the agent to determine its actions.

$$\Pi = P(a/s)$$

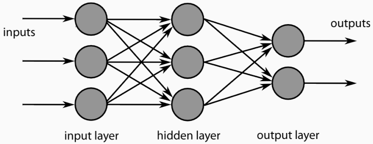
It Can be whatever you want :

- Rules
  - *Example : Move to the opposite direction of the closest enemy.*
- Learning Algorithm
  - *Example : A CNN where the input is the image of the screen*

Input



Policy



Output

GO UP!

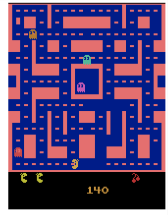
POLICY : The algorithm used by the agent to determine its actions.

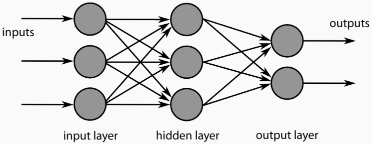
$$\Pi = P(a/s)$$

It Can be whatever you want :

- Rules
    - *Example : Move to the opposite direction of the closest enemy.*
- Learning Algorithm
    - *Example : A CNN where the input is the image of the screen*

Input Policy Output





GO UP!

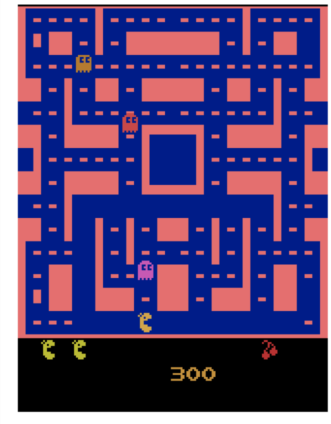
How do you train the policy?

- Random Search
  - *Does not work when the space its too big, which is often the case.*
- Genetic algorithm
  1. *Try a set of N policies.*
  2. *Keep the n best policies.*
  3. *Generate N new policies that are random deviation of these n policies.*
  4. *Iterate*
- Policy gradient
  1. *Evaluate the gradients of the rewards with regards to the policy parameters.*
  2. *update these parameters with gradient descent.*

How do you train the policy?

- Random Search
  - *Does not work when the space its too big, which is often the case.*
- Genetic algorithm
  1. *Try a set of N policies.*
  2. *Keep the n best policies.*
  3. *Generate N new policies that are random deviation of these n policies.*
  4. *Iterate*
- **Policy gradient**
  1. *Evaluate the gradients of the rewards with regards to the policy parameters.*
  2. *update these parameters with gradient descent.*

How to choose the value on which to train the policy ?

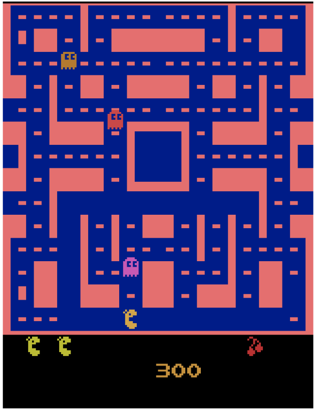


Proposition : The imediate reward.

Problem : we do not know the influence on the *long-term reward* of an action.

Example : Going up is obviously not the best action.

How to choose the value on which to train the policy ?



Proposition : The imediate reward.

Problem : we do not know the influence on the ***long-term reward*** of an action.

Example : Going up is obviously not the best action.

$\Longrightarrow$ The discounted cumulative expected reward

SOLUTION : evaluate an action based on the sum of all the rewards that come after it.

$$R_t = \sum_{i=t}^{\infty} \gamma^t r_i$$

where γ is the discounted rate and r_t is the reward at step t.

PACMAN EXAMPLE :

The agent decides to go up three times in a row. It gets +10 reward after the first step, 0 after the second step, and finally –50 after the third step (by being killed).

Discounted Rate	step	Discounted cumulated expected reward	Imedate reward
$\gamma = 0.8$	0	$R_0 = 10 + 0 \times 0.8 + (-50) \times (0.8)^2 = -22$	10
	1	$R_1 = 0 + (50) \times 0.8 = 40$	0
	2	$R_2 = 50$	-50

Tips : To get fairly reliable action scores, we must run many episodes and normalize all the action scores before training.

We want to train a model M_θ that learn the policy.

We define an objective function that represent how good our policy is :

$$J(\theta) = \mathbb{E}[R_1]$$

Update gradient with :

$$\theta = \theta + \alpha \nabla J(\theta)$$

Policy gradient algorithm as variant. Here is the one popular algorithm called REINFOCE algorithm, introduce in 1992 by Ronald Williams

1. Initiate the policy randomly
2. Let the policy play the game several times. and at each step compute the gradients that would make the chosen action even, don't apply these gradients yet.
3. Compute each action' discounted cumulative expected reward
4. Multiply each gradient vector by the corresponding action's score.
5. Compute the mean of all the resulting gradient vectors, and use it to perform a Gradient Descent step.

1. Initiate the policy randomly.
2. Let the policy play the game several times. At each game played compute and store the **state**, **actions** and the **discounted rewards**.
3. Train the model where :
   - X = state
   - y = action
   - loss = weighted cross/binary-entropy where the weight are the discounted rewards.

Exploration VS Exploitation

- **Exploitation mode** : Pick the best action according to the policy.
  $\Longrightarrow$ Problem : Being stuck in a non-optimal solution.
- **Exploration mode** : Takes random action to explore the space.
  $\Longrightarrow$ Problem : Can take a lot of time.

- **Exploitation mode** : Pick the best action according to the policy.
  $\Longrightarrow$ PROBLEM : Being stuck in a non-optimal solution.
- **Exploration mode** : Takes random action to explore the space.
  $\Longrightarrow$ PROBLEM : Can take a lot of time.

$\Longrightarrow$ SOLUTION : Mix both !

- *ϵ-greedy strategy* : Take the best action (1-ϵ)% of the time and a random action ϵ% of the time.
- *Stochastic strategy* : Use the probability to take an action to choose to act randomly or not :
  - action = 0 if random.uniform(0, 1) < predict_proba else 1

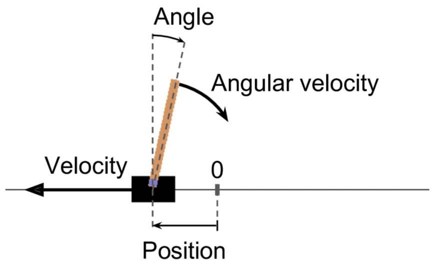
Pseudo Code

Initiate the model to train *M*.

- Generate episodes :
  - Choose action randomly (exploration) or by following the model prediction (exploitation) according to the strategy you choose.
  - Store all actions, states, discounted rewards generated etc..
- At learning time (when you have run enough episode) :
  - Normalise all discounted rewards
  - Train the model with :
    - X = (States X Discounted rewards)
    - y = actions

Environments : Cart-Pole

State : The image itself.



Observations : [Velocity, Angle, Angular velocity, Position]

Actions : Push to the Left, Push To the right.

Rewards : +1 if the game does not end.

One Notebook : *PG.ipynb*

Objectives :

- Discover Gym environment.
- Hard-Coded policy.
- Neural network policy.
- Learn Neural Network policy with Policy Gradient algorithm.

Value-Based Methods

- Policy-Based : Look for the optimal **Policy**,
    - *i.e the best action to take for each observation.*
- Value-Based : Look for the optimal **Reward**
    - *learns to estimate the expected rewards for each action in each state, then uses this knowledge to decide how to act.*

- POLICY-BASED : Look for the optimal **Policy**,
  - *i.e the best action to take for each observation.*
- VALUE-BASED : Look for the optimal **Reward**
  - *learns to estimate the expected rewards for each action in each state, then uses this knowledge to decide how to act.*

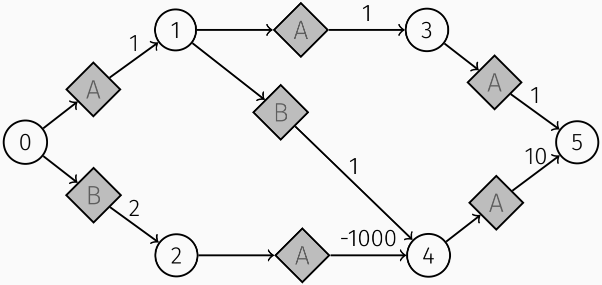
$\Longrightarrow$ Let's see how It works for a Markov Process

Formally, an MDP is composed of :

- a set of state : $S = \{s_1, s_2, .., s_n\}$.
- a set of action : $A = \{a_1, b_2, .., a_m\}$.
- a reward function : $R = S \times A \times S \to \mathcal{R}$.
- a transition function : $P_{ji}^{a} = P(s_i|s_j, a_k)$.

$\{s_1, s_2, ..\}$

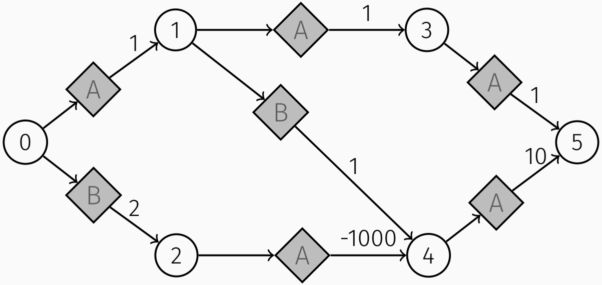
- a set of state : $S = \{0, 1, 2, 3, 4, 5\}$
- a set of action : $A = \{A, B\}$
- a reward function : $R = \{0, A, 1\} = 1, \{0, B, 2\} = 2, \ldots$
- a transition function : $P^a_{ji} = 1, \forall(i, j, a)$



Policies

There are 3 policies for this MDP

- $\pi_1 = \{0 \rightarrow 1 \rightarrow 3 \rightarrow 5\}$
- $\pi_2 = \{0 \rightarrow 1 \rightarrow 4 \rightarrow 5\}$
- $\pi_3 = \{0 \rightarrow 2 \rightarrow 3 \rightarrow 5\}$

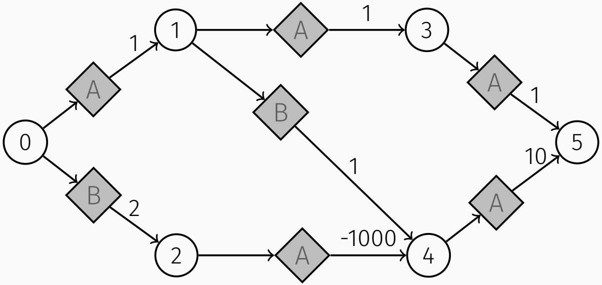


Policies

There are 3 policies for this MDP

- $\pi_1 = \{0 \rightarrow 1 \rightarrow 3 \rightarrow 5\}$
- $\pi_2 = \{0 \rightarrow 1 \rightarrow 4 \rightarrow 5\}$
- $\pi_3 = \{0 \rightarrow 2 \rightarrow 3 \rightarrow 5\}$

Which one i s the best ?

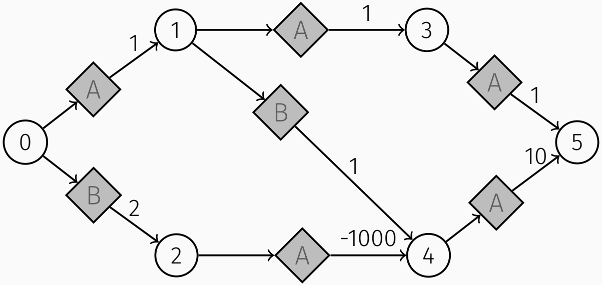


Policies

There are 3 policies for this MDP

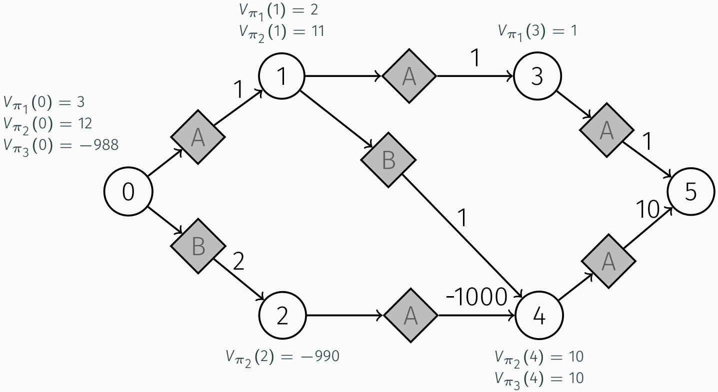
- $\pi_1 = \{0 \rightarrow 1 \rightarrow 3 \rightarrow 5\}, R = 3$
- $\pi_2 = \{0 \rightarrow 1 \rightarrow 4 \rightarrow 5\}, R = 12 \Longrightarrow$ Best Policies.
- $\pi_3 = \{0 \rightarrow 2 \rightarrow 3 \rightarrow 5\}, R = -988$

Which one i s the best ? $\Longrightarrow$ Compute their total reward !!



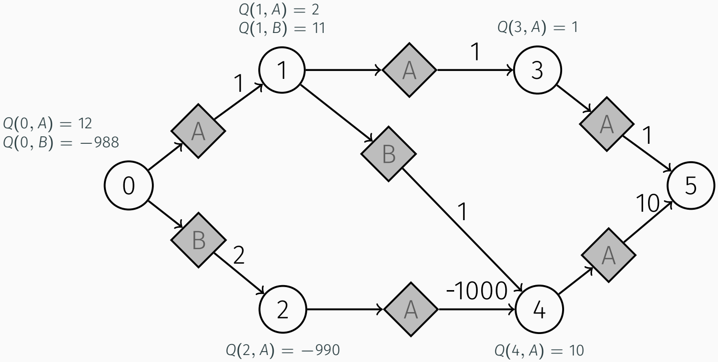
A state value $V_\pi(s)$ describes how good is it to run policy π from state s .

$\Longrightarrow$ You can then define which policy to follow from each state.

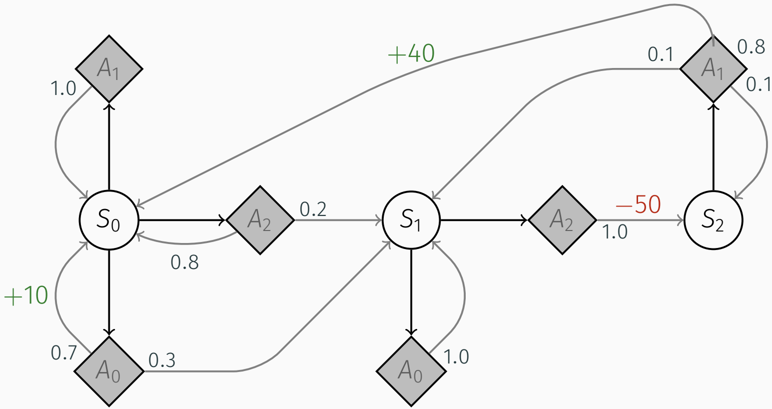


A action-state value $Q(s, a)$ describes the value of taking an action a from state s .

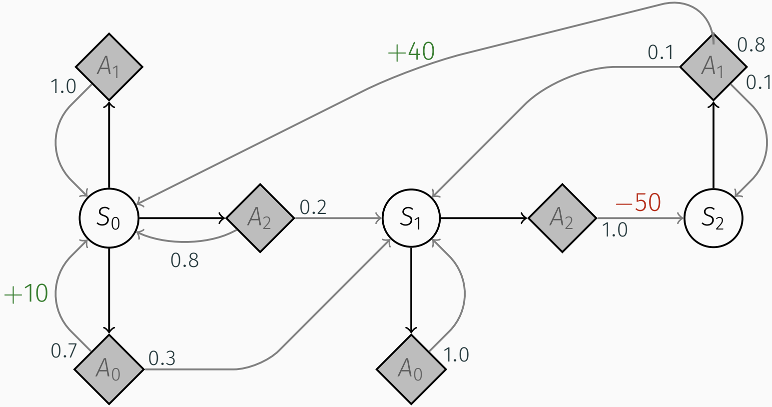
$\Longrightarrow$ You can then define an action to take at each state without knowing the policy.



What if your action can lead you do different state ?



What if your action can lead you do different state ?



Solution : The bellman equation

For all s :

$$V^*(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma . V^*(s')]$$

- $T(s, a, s')$ is the is the transition probability from state s to state s', given that the agent chose action a.
- $R(s, a, s')$ is the reward that the agent gets when it goes from state s to state s', given that the agent chose action a.
- γ is the discount rate

This equation leads directly to an algorithm that can precisely estimate the optimal state-value of every possible state : The **value iteration algorithm**.

Value Iteration algorithm :

- Initialize : $V(s) = 0, \forall s$
- Iterate, $\forall$s :

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma.V_k(s')]$$

  - $V_k(s)$ is the estimated value of state s at the k^{th} iteration of the algorithm.
- Guaranteed to converge to the optimal state values (given enough time).

Value Iteration algorithm :

- Initialize : $V(s) = 0, \forall s$
- Iterate, $\forall s$:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma . V_k(s')]$$

  - $V_k(s)$ is the estimated value of state s at the k^{th} iteration of the algorithm.
- Guaranteed to converge to the optimal state values (given enough time).

$\Longrightarrow$ We can know evaluate the optimal policy...

Value Iteration algorithm :

- Initialize : $V(s) = 0, \forall s$
- Iterate, $\forall$s :

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma . V_k(s')]$$

    - $V_k(s)$ is the estimated value of state s at the k^{th} iteration of the algorithm.
- Guaranteed to converge to the optimal state values (given enough time).

$\Longrightarrow$ We can know evaluate the optimal policy...

... but we don't know which action to take at each state!

Q-Value Iteration algorithm :

- Initialize : $Q(s,a) = 0, \forall\ (s,a)$
- Iterate, $\forall\ (s,a)$:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma . \max_{a'} Q_k(s',a')]$$

Once you have the optimal *Q-Values*, the optimal policy, $\Pi^*(s)$, is defined as :

$$\Pi^*(s) = \arg\max_a Q^*(s,a)$$

Value-Based Methods

Q-Learning

The Q-Value iteration enables to compute all Q-value of *Markov Decision Process*.

Problem : How it works in real case ?

The agent has only *partial* knowledge of the *Markov Decision Process*, i.e. :

- The agent does not know $T(s, a, s')$ and $R(s, a, s')$.
- It only knows the list of possible states $s \in S$ and actions $a \in A$.

Real Application

The Q-Value iteration enables to compute all Q-value of *Markov Decision Process*.

Problem : How it works in real case ?

The agent has only *partial* knowledge of the *Markov Decision Process*, i.e. :

- The agent does not know $T(s, a, s')$ and $R(s, a, s')$.
- It only knows the list of possible states $s \in S$ and actions $a \in A$.

Proposition : *Estimate* those values :

- Estimate $R(s, a, s')$ need to see each transition at least once.
- Estimate $T(s, a, s')$ need to experience it multiple times !

Real Application

The Q-Value iteration enables to compute all Q-value of *Markov Decision Process*.

PROBLEM : How it works in real case ?

The agent has only *partial* knowledge of the *Markov Decision Process*, i.e. :

- The agent does not know $T(s, a, s')$ and $R(s, a, s')$.
- It only knows the list of possible states $s \in S$ and actions $a \in A$.

PROPOSITION : *Estimate* those values :

- Estimate $R(s, a, s')$ need to see each transition at least once.
- Estimate $T(s, a, s')$ need to experience it multiple times !

SOLUTION : TEMPORAL DIFFERENCE LEARNING.

At iteration k, we have an estimation of the optimal state value : $V_k(s),\ \forall s$.

The Agent explore the MDP from a state s :

- Compute an estimation of the next state-value $V^*_{k+1}(s) = R(s, a, s') + \gamma . V_k(s')$
  - $R(s, a, s')$ is the immediate rewards.
  - $V_k(s')$ is the rewards it expects to get later
- Update $V_{k+1}(s)$ from previous version $V_k(s)$ and estimation $V^*_{k+1}(s)$

TD-Learning iteration algorithm :

$$V_{k+1}(s) \leftarrow (1 - \alpha)V_k(s) + \alpha V^*_{k+1}(s)$$

TD-Learning iteration algorithm :

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha[R(s,a,s') + \gamma.V_k(s')]$$

- α is the learning rate.
    - The importance or confidence we give to our estimation.
- γ is the discount factor.
    - The importance we give to future rewards.

The Q-Value Iteration algorithm can be adapted the same way.

$\Longrightarrow$ The Q-Learning algorithm

Q-Learning iteration algorithm :

$$Q_{k+1}(s,a) \leftarrow (1-\alpha)Q_k(s,a) + \alpha[R(s,a,s') + \gamma . \max_{a'} Q_k(s',a')]$$

- We build a memory table to store Q-values for **For all possible combinations of** s **and** a.

Q-LEARNING ITERATION ALGORITHM :

$$Q_{k+1}(s,a) \leftarrow (1-\alpha)Q_k(s,a) + \alpha[R(s,a,s') + \gamma . \max_{a'} Q_k(s',a')]$$

- We build a memory table to store Q-values for **For all possible combinations of** s **and** a.
- As we keep playing we update this table.

Q-Learning iteration algorithm :

$$Q_{k+1}(s,a) \leftarrow (1-\alpha)Q_k(s,a) + \alpha[R(s,a,s') + \gamma . \max_{a'} Q_k(s',a')]$$

- We build a memory table to store Q-values for **For all possible combinations of s and a.**
- As we keep playing we update this table.
- And it will converge...

Q-Learning iteration algorithm :

$$Q_{k+1}(s,a) \leftarrow (1-\alpha)Q_k(s,a) + \alpha[R(s,a,s') + \gamma . \max_{a'} Q_k(s',a')]$$

- We build a memory table to store Q-values for For all possible combinations of s and a.
- As we keep playing we update this table.
- And it will converge...
- ...with some tricks

The **TD-Learning** and **Q-Learning** iteration algorithm used with random policy are called **OFF-POLICY ALGORITHM** :

- The policy you're learning :π_l (by updating either Q-value or state value V) is not the one you're execute (which is random) π_r .
- You're learning how to act by observing someone doing random action.
- Q-Learning can work only if the exploration policy explores the MDP thoroughly enough.
- It may take an extremely long time to do so.

The **TD-Learning** and **Q-Learning** iteration algorithm used with random policy are called **OFF-POLICY ALGORITHM :**

- The policy you're learning :π_l (by updating either Q-value or state value V) is not the one you're execute (which is random) π_r .
- You're learning how to act by observing someone doing random action.
- Q-Learning can work only if the exploration policy explores the MDP thoroughly enough.
- It may take an extremely long time to do so.

$\Longrightarrow$ Can't we do better ?

At each step you will choose either :

- the random policy π_r with a probability ϵ
- or the learned policy π_l with a probability $1 - \epsilon$

You start with $\epsilon = 1$ purely random exploration and the environment, and you decrease the value of epsilon over iteration in order to spend more and more time exploring the interesting parts of the environment.

Value-Based Methods

DeepQ-Learning (DQN)

Approximate Q-Learning

Problem : Q-Learning still does not scale to large MDPSs !

Example : Pacman.

- 240 pellets that can be eaten and are present of absent.
- $2^240 \approx 10^72$ which is approximately the number of atoms in the universe.
- and we need to consider pacman and ghosts' position.

We can't compute Q-value for every (s, a) couple (neither store it) !

Problem : Q-Learning still does not scale to large MDPSs!

Example : Pacman.

- 240 pellets that can be eaten and are present of absent.
- $2^240 \approx 10^72$ which is approximately the number of atoms in the universe.
- and we need to consider pacman and ghosts' position.

We can't compute Q-value for every (s, a) couple (neither store it)!

$\Longrightarrow$ **Approximate Q-learning** : use a function that evaluate it.

Q-Learning

We remembering the Q-Values in a big table and we pick value from it.

Q-Learning

We remembering the Q-Values in a big table and we pick value from it.

Approximate Q-learning

We train a function Q that will generalise the approximation of the Q-value and use this function to estimate value.

- Compute hand-crafted features (localisation of pacman, distance to the ghost, localisation of remaining pellets…)

Q-Learning

We remembering the Q-Values in a big table and we pick value from it.

Approximate Q-learning

We train a function Q that will generalise the approximation of the Q-value and use this function to estimate value.

- Compute hand-crafted features (localisation of pacman, distance to the ghost, localisation of remaining pellets...)

Deep Q-learning

- Used only the images as features and use deep-Learning to estimate the Q-function.

$\Longrightarrow$ DeepMind make it worth (very well!) on various Atari Game in 2014

Q Value Iteration Algorithm

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma . \max_{a'} Q_k(s',a')]$$

Q-learning

$$target = R(s,a,s') + \gamma \max_{a'} Q_k(s',a')$$

$$Q_{k+1}(s,a) \leftarrow (1-a)Q_k(s,a) + \alpha[target]$$

Deep Q-learning

$$target = R(s,a,s') + \gamma \max_{a'} Q_k(s',a')$$

$$\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_\theta \mathbb{E}_{s \sim' P(s'|s,a)}[(Q_\theta(s,a) - target(s'))^2]_{\theta=\theta_k}$$

- The learning rate α of the optimization can be interpreted the same way.

Pseudo Code

Initiate Q, a *convolutional network*.

- Generate episodes from Q (or randomly) : s,a,s'
- At learning time :
  - Create target from Q, s, a, s' (also generate with Q) :

$$Target = [R(s, a, s') + \gamma. \max_{a'} Q(s', a')]$$

  - Train the model :

$$\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_\theta \mathbb{E}_{s \sim' P(s'|s,a)}[(Q_\theta(s, a) - target(s'))^2]_{\theta=\theta_k}$$

- The train model Q actually take only s as an input and generate all $Q(s, a)$ value.

$$Q \mapsto [Q(s, a_1), Q(s, a_2), Q(s, a_3), Q(s, a_4)]$$

  - During training we fixed the value for not tested a, i.e if we generate a target for (s, a_2, s'), the real train target is :

$$Target = [Q(s, a_1), R(s, a_2, s') + \gamma . \max_{a'} Q(s', a'), Q(s, a_3), Q(s, a_4)]$$

- We train Q as a classical *Supervised learning problem*.

PROBLEM : The input of the supervised learning problem are not i.i.d

The input from the same episode are not independent.

SOLUTION : Experience Replay

Sample **randomly** some state from different episode previously generated.

Problem : The targets are unstable :

$$Target = [R(s, a, s') + \gamma . \max_{a'} Q(s', a')]$$

- The target value depends of Q itself.
- At each train iteration :
  - our Q values shift to get closer to the target.
  - which shift the same way!
- we are chasing a **non-stationary target**.

Solution : Use a different Q function to generate the target!

Peusdo-Code

- Create Q_{main} and Q_{target}.
- Initiate the replay memory M.
- Generate some train data from Q_{main} : s, a, s' (or randomly) and store it in M
- At learning time :
    - Sample a batch of data (s, a, s') from replay memory M.
    - Create *targets* with Q_{target}

$$Target = [R(s, a, s') + \gamma . \max_{a'} Q_{target}(s, a)]$$

    - Train the Q_{main} Network
- Update Q_{target} weight with Q_{main} weight from time to time.
    - With a rate τ

Number of improvements of DQN algorithm (mainly by DeepMind) have allowed greater performance and stability.

- Dueling DQN Wang et al. [2015] .
- Double QDN. Van Hasselt et al. [2016] .
- Prioritized Experience Replay Schaul et al. [2015]

Number of improvements of DQN algorithm (mainly by DeepMind) have allowed greater performance and stability.

- Dueling DQN Wang et al. [2015].
- Double QDN. Van Hasselt et al. [2016].
- Prioritized Experience Replay Schaul et al. [2015]

Main intuition :

- Q-value represent how good it is to take an action a at state s.
- It can be decomposed into two notions :
  - The **value function** $V(s)$: How good it is to be in this state ?
  - The **advantage function** $A(s, a)$: How much better is it to take this action at that state
- The formally :

$$Q(s, a) = V(s) + A(s, a)$$

- Actually computed this way :

$$Q(s, a) = V(s) + A(s, a) - \frac{1}{\mathcal{A}} \sum_{1}^{\mathcal{A}} A(s, a)$$

Main intuition :

- Q-value represent how good it is to take an action a at state s.
- It can be decomposed into two notions :
  - The **value function** $V(s)$: How good it is to be in this state ?
  - The **advantage function** $A(s, a)$: How much better is it to take this action at that state
- The formally :

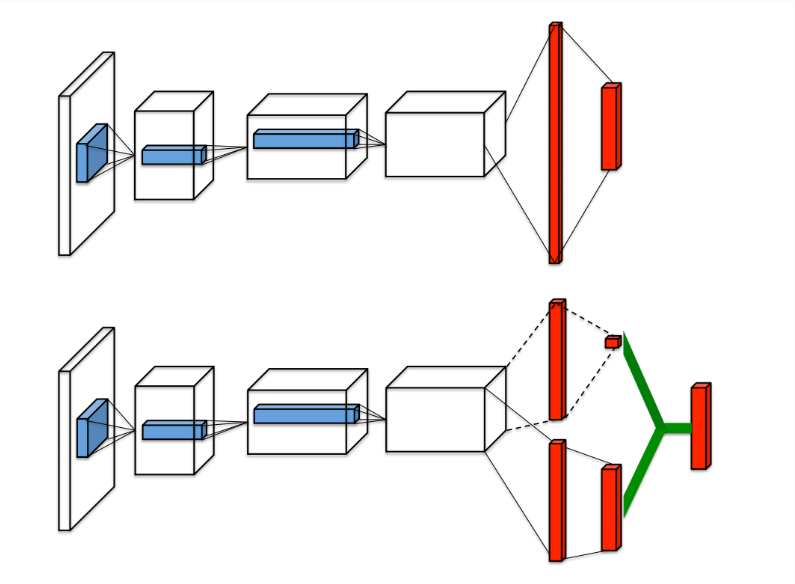
$$Q(s, a) = V(s) + A(s, a)$$

- Actually computed this way :

$$Q(s, a) = V(s) + A(s, a) - \frac{1}{\mathcal{A}} \sum_{1}^{\mathcal{A}} A(s, a)$$

Let's build a network that represent it !

Dueling - Illustration



Main intuition :

- DQN overestimates the Q-values of the potential actions to take.

Separated Target Network Take the max over Q_{target} value.

$$target = R(s, a, s') + \gamma \max_{a'} Q_{target}(s', a')$$

Double DQN : Use Q_{main} to chose the action.

$$target = R(s, a, s') + \gamma Q_{target}(s', argmax_a Q_{main}(s', a))$$

Second Notebook : *DQN.ipynb*

Objectives :

- Q-Learning on simple Markov Decision Process
- DQN on *pacman-like* game.
- Double-DQN with dueling on *pacman-like* game.
- Try on real pacman !

Reference

https ://medium.com/@awjuliani/simple-reinforcement-learning-with-tensorflow-part-4-deep-q-networks-and-beyond-8438a3e2b8df

https ://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-q-learning-with-tables-and-neural-networks-d195264329d0

https ://towardsdatascience.com/reinforcement-learning-tutorial-part-3-basic-deep-q-learning-186164c3bf4

https ://sergioskar.github.io/Reinforcement_learning/

http ://www2.econ.iastate.edu/tesfatsi/RLUsersGuide.ICAC2005.pdf

Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems, Géron [2017]

Références

Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.", 2017.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540) :529, 2015.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv :1511.05952*, 2015.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv :1511.06581*, 2015.