DSR - Introduction to Reinforcement Learning

DSR batch # 17 - February 2019

About me

- DSR participant - batch #15

- Msc in Engineering and Applied Mathematics
- Now working at Ecole Polytechnique in Paris as a Research Engineer in Machine Learning and PhD student (in Reinforcement Learning)

- LinkedIn: https://www.linkedin.com/in/alicemartindonati/
- Email: alice.martindonati.pro@gmail.com

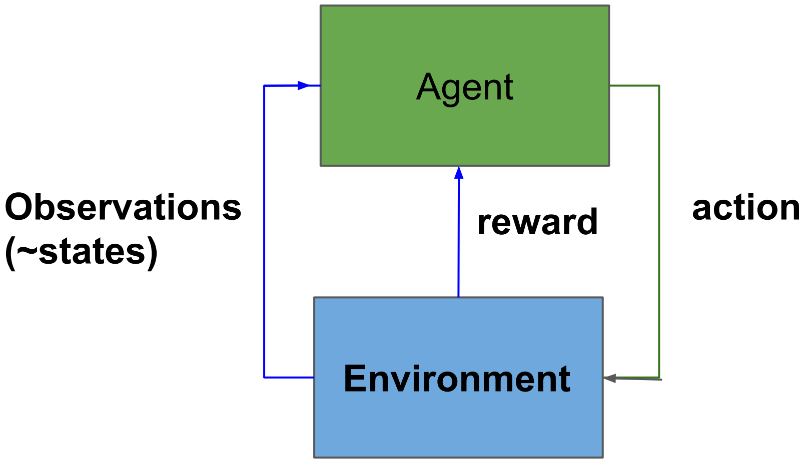
Course outline

- **Introduction: Reinforcement Learning in the Machine Learning Field**
- **Part I: Key concepts in Reinforcement Learning**
- **Part II: Solving a MDP with Dynamic Programming**
- **Implementation of DP algorithms in Python**
- **Part IV: Tabular Q-learning**
- **Implementation of tabular-Q learning in Python**
- **Part V: Introduction to Deep Q networks**
- **Part VI: Introduction to policy gradient algorithms**
- **Conclusion: Future of RL and challenges**

The Reinforcement Learning problem

At each timestep t, the Agent:
-receives an observation O_t and reward R_t from the environment
-performs an action A_t

The environnement:
-receives action A_t,
-emits observations O_{t+1} and scalar reward R_{t+1}



RL: A decision making problem where an agent interacts sequentially in an environnement
Goal: find the sequential set of actions that maximizes his total reward

Machine Learning and Reinforcement Learning

Supervised Learning

Labeled Dataset (X,y)
Learn Relation between X and y
$y=f(X)$
Example: Image Classification

Unsupervised Learning

Unlabelled Dataset X
Learn Hidden Pattern in the Dataset
Example: Clustering

Reinforcement Learning

Agent Learns by interacting with an environment and receiving a signal from this environment

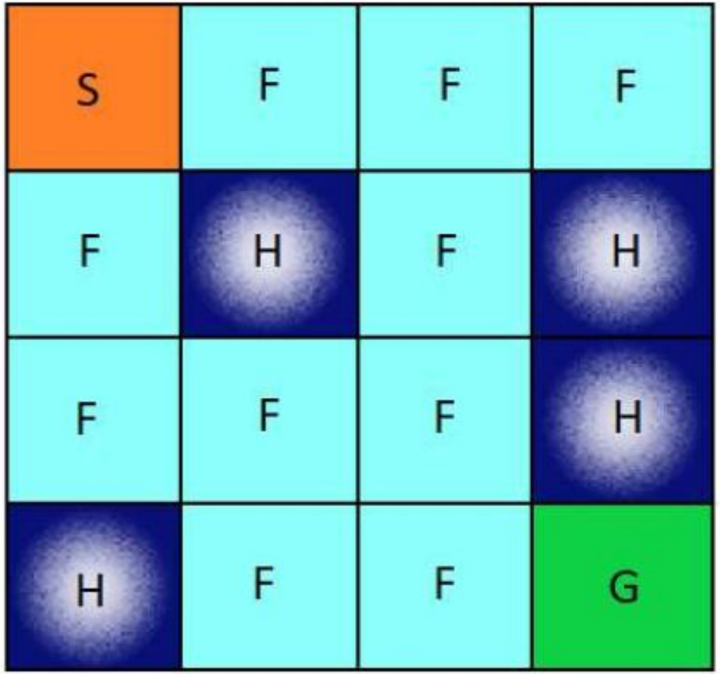
-Trial and Error Process

Characteristics of Reinforcement Learning

- There is no supervisor, only a reward final
- Feedback is delayed, not instantaneous

- Time really matters: sequential decision making problem
- Active learning process: Agent’s actions influence the subsequent data it receives (the reward he gets, his state).

Example : The frozen Lake



An agent has to go from S to G in a frozen lake without falling into the holes.

Actions: move right, left, up, down.

The agent receives +1 if he reaches the goal G, 0 if he falls into a hole.

The episode ends when the agent falls into a hole or reaches the goal

Examples of sequential decision making problems

Fly stunt manoeuvres with an helicopter

Make a humanoid robot walk

Beat humans at Games like Go or BackGammon

Play Atari Games

Energy: Control a power station

Finance: Optimise an investment portfolio

Medicine: Optimize drug dosages

Key Elements in a RL problem

OBSERVATIONS:

- History H_t: sequence of past observations
- State S_t : function of H_t

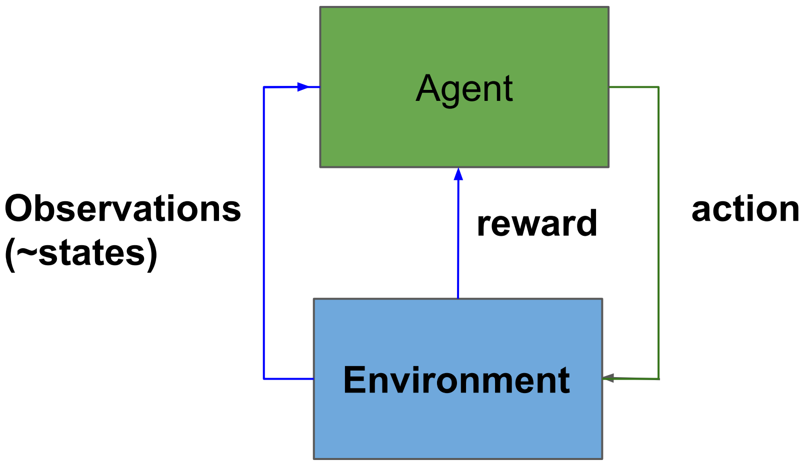
SET OF ACTIONS A

REWARD = function of state S_t and action A_t

DISCOUNT RATE γ: penalisation of the future

MODEL OF THE ENVIRONNEMENT

TERMINAL STATE S_T OR TIME HORIZON H



Key Elements of a RL problem - History, State and the Markov Property

- History: sequence of past observations

$$H_t = (O_1, R_1, A_1), ..., (O_t, R_t, A_t)$$

- The state is a function of the history :

$$S_t = f(H_t)$$

- Markov Property : “The future is independent of the past given the present”

$$P(S_{t+1}|S_t) = P(S_{t+1}|S_t, \ldots, S_1)$$

- Once the state is known, the history can be thrown away

Key Elements of a RL problem - The reward

- A reward R_t is a scalar feedback signal
- It indicates how well the agent is going at step t
- The agent job is to maximise **cumulative rewards**

- Reinforcement Learning is based on the **reward hypothesis**: all goals can be described by the maximisation of expected cumulative reward

Key Elements of a RL problem - The Return

- The return G_t is the total discounted reward from time step t

$$G_t = r_{t+1} + \gamma r_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

- The discount rate γ is the present value of future rewards: $\gamma \in [0, 1]$

Key Elements of a RL problem - The Discounted rate

Why using a discounted rate? **The discount rate penalizes the future**

- Future is more uncertain
- Human Preference for immediate rewards
- If reward = money, discounted rate corresponds to actualisation rate
- Mathematical convergence for environnement with no terminal states:

$$G_t = r_{t+1} + \gamma r_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

Key Elements of a RL problem - The Agent’s Policy

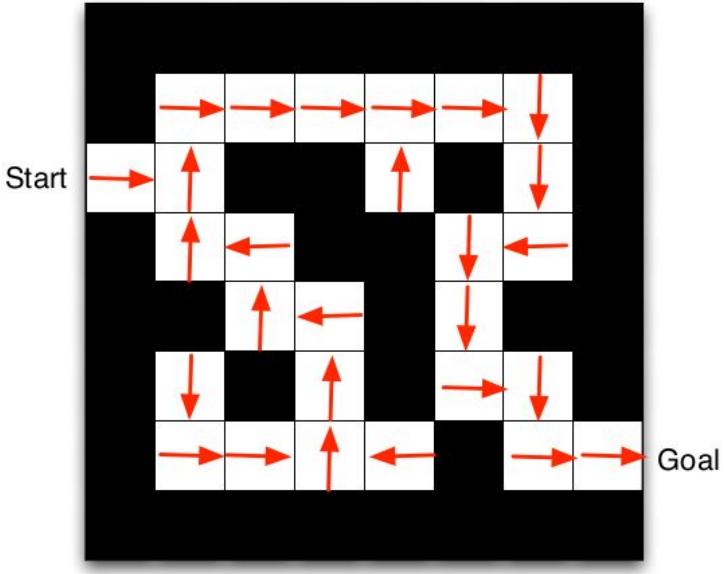
- Model the agent’s behaviour
- Mapping from States to Actions

- Deterministic Policy:

$$a = \pi(s)$$

- Stochastic Policy:

$$\pi(a|s) = P(A_t = a|S_t = s)$$



Key Elements of a RL problem - The Value Functions

- The Value Function is a prediction of future reward
- Used to evaluate the Goodness / Badness of each state
- And therefore to select between actions

2 Value-Functions:

- **Value Function V(s): Measures how good is being in a state s**
- **Action Value Function Q(s,a): Measures how good is being in a state s and taking an action a**

Value-Functions computations are the core of most reinforcement learning algorithms

Key Elements of a RL problem - The model of the environment

A model is the agent’s representation of the environment

A model is made of 2 components:

- **Transition Model P: predicts the next state**
- **Reward Model R: predicts the next immediate reward**

$$p(s'|s,a) = P\left[S_{t+1} = s' \middle| S_t = s, A_t = a\right]$$

$$r(a,s) = \mathbb{E}\left[r_{t+1} \mid S_t = s, A_t = a\right]$$

Key Elements of a RL problem - The model of the environment

Stochastic versus Deterministic Environment:

- Deterministic: The outcome of each action is known (ex chess)
- Stochastic: The outcome of each action is not known (ex:dice game)

Fully Observable versus Partially Observable

- Fully Observable: The observation of the agent is the observation of the environnement (ex: chess)
- Partially Observable: The agent observes indirectly the environment: the agent state is different from the environment state (ex: poker)

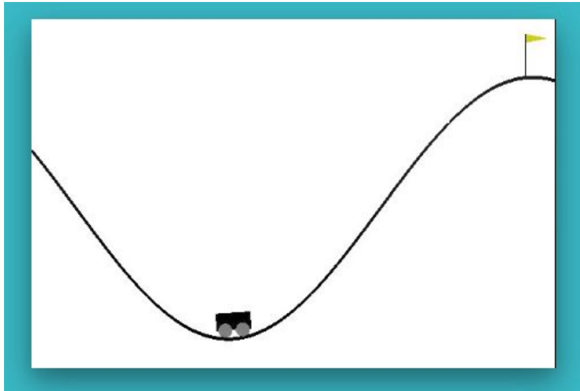
Key Elements of a RL problem - The model of the environment

Episodic versus continuing (non-episodic):

- Episodic: There is a terminal state (ex: game)
- Non-Episodic: There is no terminal state (ex: robotic navigation)

Discrete versus Continuous:

- Discrete: Action Space is a discrete set of actions
- (ex: Frozen Lake)
- Continuous: Action Space is continuous (ex: Mountain Car)



Part I - The Mathematical Framework for Reinforcement Learning: The Markov Decision Process

The Markov Decision Process (MDP)

A Markov Decision Process formally describes an RL environment when:

- The environnement is fully observable
- The current state follows the Markov Property: $P(S_{t+1}|S_t) = P(S_{t+1}|S_t, \ldots, S_1)$

A MDP is a tuple $< S, A, P, R, \gamma >$

- Set of States S
- Set of Actions A
- Transition Probabilities P
- Reward Probabilities R
- Discount rate γ

The Markov Decision Process (MDP)

A MDP is a tuple $< S, A, P, R, \gamma >$

- **Set of States S:** states that the agent can be in.
- **Set of Actions A**: Actions that can be perform by the agent from one state to another
- **Transition Probabilities P:** Probability of moving from one state s to another state s' by performing a:

$$p(s'|s,a) = P\left[S_{t+1} = s' \middle| S_t = s, A_t = a\right]$$

- **Reward Probabilities R:** probability of reward acquired by the agent from being in one state s and performing action a:

$$r(a,s) = \mathbb{E}\left[r_{t+1} \mid S_t = s, A_t = a\right]$$

- **Discount rate γ:** controls the importance of immediate and future rewards

Value Functions in a MDP - Value Function V

How to measure how “good” is a policy?

- **Value Function V(s):** measure how good is being in a state: it is the **expected return of the agent when being in state s**

$$V_\pi(s) = \mathbb{E}_\pi\left(G_t \middle| S_t = s\right) = \mathbb{E}_\pi\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| S_t = s\right)$$

Value Functions in a MDP - Action Value Function Q

How to measure how "good" is a policy?

- **Action-Value Function Q(s,a):** measure how good is being in state s and performing an action a = agent's expected return when being in state s and performing action a

$$Q_\pi(s,a) = \mathbb{E}_\pi\left(G_t \middle| S_t = s, A_t = a\right) = \mathbb{E}\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| S_t = s, A_t = a, \pi\right)$$

Optimal Value Functions of a MDP

Goal of the agent = maximizing its total reward > Maximizing the value functions on all the possible policies he can follow

- Optimal Value Function v_*

$$v_*(s) = \max_{\pi} v_{\pi}(s)$$

- Optimal Action Value Function q_*:

$$q_*(s, a) = \max_{\pi} q_{\pi}(s, a)$$

Optimal Policy of a MDP

Goal of the agent = maximizing its total reward > Finding the optimal policy

- Associated optimal policy **π_***: a policy is said to be optimal when it dominates over any policy in every state:

$$\pi^* \in \arg\max_{\pi} V_\pi$$
$$\pi^* \text{ is optimal} \Leftrightarrow \forall s \in S, \forall \pi,\ \ V_{\pi^*}(s) \geq V_\pi(s)$$

- A MDP is solved when the optimal Value function (q_* or v_*) is reached: it is equivalent to reach the optimal policy

Bellman Equations - Bellman Expectation Equation for V

Decomposition of the Value Function between immediate reward and the Value function of the next state:

$$v_\pi(s) = \mathbb{E}\left[r_{t+1} + \gamma v_\pi(S_{t+1}) \middle| S_t = s\right]$$

$$v_\pi\left(s\right) = \sum_{a \in A} \pi(a|s) q_\pi(s,a) = \sum_{a \in A} \pi(a|s) \left(r\left(s,a\right) + \gamma \sum_{s' \in S} p\left(s'|s,a\right) v_\pi\left(s'\right) \right)$$

Bellman Equations - Bellman Expectation Equation for Q

Decomposition of the State-Action Value Function between immediate reward and the Value function of the next state:

$$q_\pi(s,a) = \mathbb{E}\left[r_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \middle| S_t = s, A_t = a \right]$$

$$q_\pi\left(s,a\right) = r\left(s,a\right) + \gamma \sum_{s' \in S} p\left(s' \middle| s,a\right) \sum_{a' \in A} \pi(a'|s') q_\pi(a', s')$$

Bellman Optimality Equation for V

Decomposition of the **Optimal** Value Function between immediate reward and the Value function of the next state:

$$v_*(s) = \max_{a \in A} q_*(s, a)$$

$$v_*(s) = \max_{a \in A} \left(r(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) v_*(s') \right)$$

To solve a MDP, we will solve the Bellman Optimality Equation with iterative methods.

Bellman Optimality Equation for Q

Decomposition of the **Optimal** State-Action Value Function between immediate reward and the State-Action Value function of the next state:

$$q_*(s,a) = r(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) v_*(s')$$

$$q_*(s,a) = r(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) \max_{a'} q_*(s',a')s$$

To solve a MDP, we will solve the Bellman Optimality Equation with iterative methods.

Solving a MDP : Learning versus Planning

MDP $< S, A, P, R, \gamma >$

- **Case 1: Full knowledge of the MDP:** the transition state matrix P and the reward probabilities R (=the dynamics of the MDP) are known.

  > MDP solved **with Dynamic Programming Methods**: the agent performs computation with its model to improve its policy

- **Case 2: The model of the environment is unknown.**

> MDP solved **with Reinforcement Learning Methods**: the agent interacts with its environment to improve its policy

Part III: Solving a MDP with Dynamic Programming

Dynamic Programming

Technique to solve complex problems.

- The problem is broken into sub-problems: for each sub-problem, we compute and store the solution.
- If the same sub-problem occurs, we use the already computed solution

Dynamic Programming Algorithms to solve a MDP

- The Bellman equation allows to decompose the MDP recursively
- Value Functions allow to store and reuse function

Dynamic Programming

DP algorithms to solve a MDP

- **Value Iteration**
- **Policy Iteration**

Value Iteration Algorithm - Intuition

- Based on updates of the Value Function V(s)
- We start off with a random Value Function
- We improve the Value Function by an iterative application of the Bellman Optimality Equation: at each timestep k, for each state s

$$v_{k+1}(s) = \max_{a \in A} \left\{ r(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) v_k(s') \right\}$$

- Theorem: the value function converges towards the optimal value function v*

Value Iteration - Example

Value iteration

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi \approx \pi_*$, such that
  $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$

Value Iteration Algorithm

Value iteration

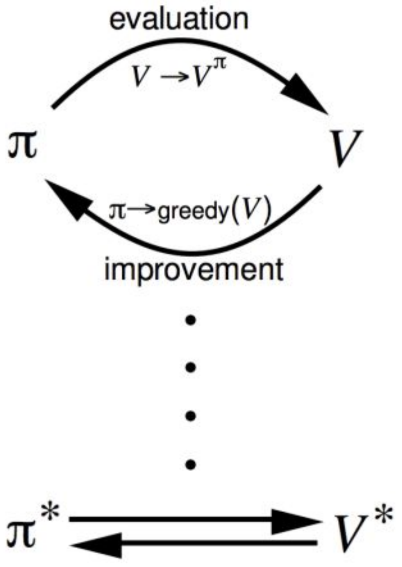
Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi \approx \pi_*$, such that
  $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$

Policy Iteration - Intuition

- Based on iterative update of the Policy π
- We start off with a random policy
- If this policy is not optimal, then the find new improved policy
- We repeat this process until we find the optimal policy
- Algorithm based on two steps:
- **Policy Evaluation:** The policy is evaluated using the value-function.
- **Policy Improvement:** Upon evaluating the value function, if it is not optimal, we find a new improved policy



Policy Iteration - Policy Evaluation and Improvement

1. Policy evaluation for current policy π_k:

- Repeat until convergence:

$$v_{i+1}^{\pi_k}(s) \leftarrow \sum_{s' \in S} p(s'|s,a)[r(s,\pi(s),s') + \gamma v_i^{\pi_k}(s')]$$

2. Policy Improvement: find the best action according to one step look-ahead:

$$\pi_{k+1}(s) \leftarrow \arg\max_a \sum_{s' \in S} p(s'|s,a)[r(s,a,s') + \gamma v^{\pi_k}(s')]$$

Policy Iteration Algorithm

Policy iteration (using iterative policy evaluation)

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
   $\quad \Delta \leftarrow 0$
   $\quad$ For each $s \in \mathcal{S}$:
   $\quad\quad v \leftarrow V(s)$
   $\quad\quad V(s) \leftarrow \sum_{s',r} p(s', r | s, \pi(s)) \big[r + \gamma V(s') \big]$
   $\quad\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   $\quad$ *old-action* $\leftarrow \pi(s)$
   $\quad \pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s', r | s, a) \big[r + \gamma V(s') \big]$
   $\quad$ If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Value Iteration versus Policy Iteration

- Policy Iteration converges usually with less iterations than Value Iteration
- But each step of the Policy Iteration is computationally more expensive

- Overall, Policy Iteration is usually faster than Value Iteration

Part IV: Reinforcement Learning with TD Learning and Monte-Carlo Methods

Limits of Dynamic Programming

- In most RL problems, the dynamics of the environment are not known (transition model and reward probabilities)

- Even if the full MDP is known, computing over all possible states and actions can be too costly

- Model-free algorithms overcome these limitations

Model-Free Prediction and Model-Free Control

- Dynamic Programming : solve a known MDP (probability transition matrix is known)
- Reinforcement Learning solves an unknown MDP with 2 steps:
- **Model free prediction:** Estimate the value function of an unknown MDP: V(s) or Q(s,a)
- **Model-free control:** Optimise the value function of an unknown MDP: find V*(s) or Q*(s,a)

> **2 main reinforcement learning methods: Monte-Carlo and TD Learning**

Monte-Carlo Reinforcement Learning

- Goal: learn v_π from episodes of experience under policy π by using the **empirical mean return**
- Episode= samples of (state, action, rewards) from initial state until terminal state (end of episode):
- Recall the return formula:

$$G_t = r_{t+1} + \gamma r_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

- The value function V(s) is estimated by taking the average return over the episodes for each state s
- 2 variants for Monte-Carlo Policy Evaluation: First-Visit Monte-Carlo and Every-Visit Monte-Carlo

Monte-Carlo Policy Evaluation

For each state s visited during the episode:

- First-visit MC: The first-time the state is visited in the episode, increment counter state: $N(s) \leftarrow N(s) + 1$
- Every-visit MC: Every time the state s is visited in the episode, increment counter state: $N(s) \leftarrow N(s)+1$
- Increment total return for the state: $S(s) \leftarrow S(s) + G_t$
- Value Function V estimated by mean return: $V(s)=S(s)/N(s)$
- Theorem (Law of Large Number): $V(s) \rightarrow v_{\pi(s)}$ as $N(s) \rightarrow \infty$

First-Visit MC Prediction - Algorithm

First-visit MC prediction, for estimating $V \approx v_\pi$

Initialize:

$\pi \leftarrow$ policy to be evaluated

$V \leftarrow$ an arbitrary state-value function

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:

Generate an episode using π

For each state s appearing in the episode:

$G \leftarrow$ the return that follows the first occurrence of s

Append G to $Returns(s)$

$V(s) \leftarrow \text{average}(Returns(s))$

Monte-Carlo Control

- Monte-Carlo Control is based on the Policy Iteration Idea:
- **Policy Evaluation:** Learn v~v_π with Monte-Carlo Policy Iteration Algorithm (instead of solving by dynamic programming)
- **Policy Improvement:** Greedy Policy Improvement
- BUT Greedy-Policy Improvement with V requires a model of the MDP:

$$\pi_{k+1}(s) \leftarrow \arg\max_a \sum_{s' \in S} p(s'|s,a)[r(s,a,s') + \gamma v^{\pi_k}(s')]$$

- Instead, Greedy-Policy Improvement with Q(s,a) is model-free:

$$\pi'(s) = \arg\max_{a \in A} q(s,a)$$

> MC Control uses the State-Value function Q instead of the Value Function V

Monte-Carlo Control: From State Function V(s) to State-Action Value Function Q

- Without a model, it is useful to estimate **<u>action values= the action of state-value pairs</u>** rather than **<u>state values.</u>**
- One must explicitly estimate **the value of each action** in order for the values to be useful in suggesting a policy.
- Recall the Action Value Function Q formula:

$$Q_\pi(s,a) = \mathbb{E}_\pi\left(G_t \middle| S_t = s, A_t = a\right) = \mathbb{E}\left(\sum_{k=0}^{\infty}\gamma^k r_{t+k+1} \middle| S_t = s, A_t = a, \pi\right)$$

> Problem: how to know about the state-action value if we haven't taken this action in that state? **<u>This is the problem of exploration</u>**

Exploration and exploitation tradeoff

Reinforcement Learning is a trial and error process: an agent discover a good policy through experiences with the environment

- **Exploration** finds more information about the environment
- **Exploitation** exploits known information to maximise reward
- It is usually important to explore and exploit

Exploration / Exploitation trade-off examples:

- Exploitation: Go to your favorite restaurant
- Exploration: Try a new restaurant

ϵ-greedy exploration

- The simplest idea to ensure exploration of all possible actions
- ϵ is a small number: $\epsilon \in [0,1]$
- With proba 1-ϵ, the greedy action is chosen
- With proba ϵ, a random action is chosen

$$\pi(a|s) = \begin{cases} 1-\epsilon & \text{if } a^* = \arg\max_{a \in A} q(s,a) \\ \epsilon & \text{otherwise} \end{cases}$$

> **We use a epsilon-greedy improvement for policy improvement in MC Control**

Monte-Carlo Control with ϵ-greedy exploration

On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s, a) \leftarrow$ arbitrary
$Returns(s, a) \leftarrow$ empty list
$\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

Repeat forever:
(a) Generate an episode using π
(b) For each pair s, a appearing in the episode:
$G \leftarrow$ the return that follows the first occurrence of s, a
Append G to $Returns(s, a)$
$Q(s, a) \leftarrow$ average$(Returns(s, a))$
(c) For each s in the episode:
$A^* \leftarrow \arg\max_a Q(s, a)$ (with ties broken arbitrarily)
For all $a \in \mathcal{A}(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(s)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq A^* \end{cases}$$

Model-Free Prediction with Temporal-Difference (TD) Learning

- TD methods learn directly from episodes of experience like MC Methods
- *But* **learn from incomplete episodes by bootstrapping**: update is done every timestep and not after each episode

- TD updates a guess towards a guess.
- The Value $V(S_t)$ is updated toward estimated return: the immediate reward + the estimated reward for the rest of the trajectory:

$$R_{t+1} + \gamma V(S_{t+1})$$

Model-Free Prediction with Temporal-Difference (TD) Learning - The TD update

- The Value $V(S_t)$ is updated toward estimated return: the immediate reward + the estimated reward for the rest of the trajectory $R_{t+1} + \gamma V(S_{t+1})$

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$

$R_{t+1} + \gamma V(S_{t+1})$ is the TD-update

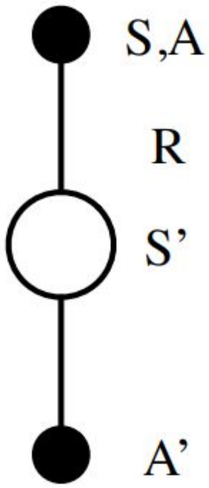
$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is the TD-error

TD Control

- Same idea than to MC Control: Apply TD Prediction to Q(s,a)
- Use a epsilon-greedy improvement
- Update every time-step instead of every episode for Monte-Carlo

- **2 TD learning variants**
- SARSA: On-Policy Control
- Q-Learning: Off-Policy Control

SARSA update



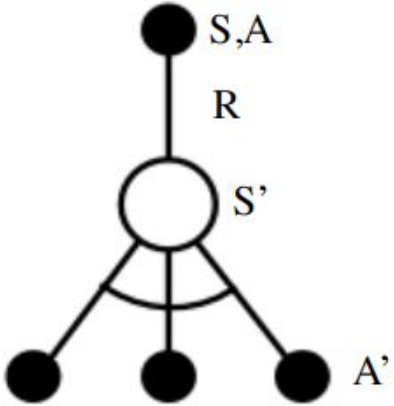
A’ is chosen following a policy
=> On-policy algorithm

$$Q(S,A) \leftarrow Q(S,A) + \alpha \left(R + \gamma Q(S',A') - Q(S,A)\right)$$

SARSA algorithm

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma Q(S', A') - Q(S, A)\right]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Q-learning update



$$Q(S,A) \leftarrow Q(S,A) + \alpha \left(R + \gamma \max_{a'} Q(S',a') - Q(S,A) \right)$$

Q-learning algorithm

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize S
    Repeat (for each step of episode):
        Choose A from S using policy derived from Q (e.g., ε-greedy)
        Take action A, observe R, S'
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
        $S \leftarrow S'$;
    until S is terminal

Monte-Carlo versus TD-Learning

TD can learn before knowing the final outcome:

- TD learns online at every timestep
- MC must wait until the end of the episode before return is known

TD can learn without knowing the final outcome

- TD can learn from incomplete sequences
- TD can learn in continuing environments
- MC can only learn in episodic environments

MC versus TD : Bias/Variance trade-off

- In MC: Return G_t is **unbiased estimate** of $V_\pi(s)$:

$$G_t = r_{t+1} + \gamma r_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

- In TD: TD target is **biased estimate** of $V_\pi(s)$:

$$R_{t+1} + \gamma V(S_{t+1})$$

TD target is much lower variance than the Return:

- Return depends on many random samples of actions, transitions, rewards
- TD update depends on one random action, transition, reward

MC versus TD: advantages and disadvantages

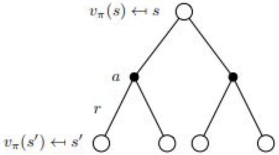
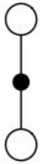
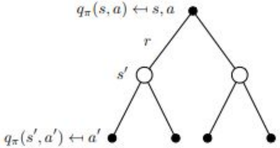
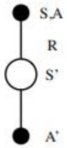
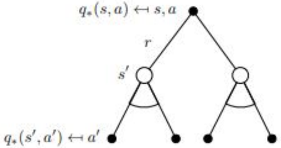
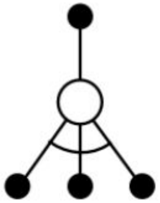
MC has high variance and zero bias:

- Not very sensitive to initial value of state
- Very simple to understand and use
- Only useable in episodic environments

TD has low variance, some bias:

- Usually more sample-efficient than MC (converge with few numbers of episodes)
- More sensitive to initial value of state

Summary: DP versus TD-learning

	Full Backup (DP)	*Sample Backup (TD)*
Bellman Expectation Equation for $v_\pi(s)$	 Iterative Policy Evaluation	 TD Learning
Bellman Expectation Equation for $q_\pi(s,a)$	 Q-Policy Iteration	 Sarsa
Bellman Optimality Equation for $q_*(s,a)$	 Q-Value Iteration	 Q-Learning

Summary: DP versus TD-learning

Full Backup (DP)	*Sample Backup (TD)*
Iterative Policy Evaluation	TD Learning
$V(s) \leftarrow \mathbb{E}\left[R + \gamma V(S') \mid s\right]$	$V(S) \xleftarrow{\alpha} R + \gamma V(S')$
Q-Policy Iteration	Sarsa
$Q(s,a) \leftarrow \mathbb{E}\left[R + \gamma Q(S',A') \mid s,a\right]$	$Q(S,A) \xleftarrow{\alpha} R + \gamma Q(S',A')$
Q-Value Iteration	Q-Learning
$Q(s,a) \leftarrow \mathbb{E}\left[R + \gamma \max_{a' \in \mathcal{A}} Q(S',a') \mid s,a\right]$	$Q(S,A) \xleftarrow{\alpha} R + \gamma \max_{a' \in \mathcal{A}} Q(S',a')$

Part V: Introduction to Deep Reinforcement Learning

Caveats of tabular Q-learning

So far, the algorithms store value functions in a look-up table: For prediction and control, we need to store V(s) and Q(s,a) for every state s and every state-action value pairs

This is a problem for large MDPs:

- There are too many states and/or actions to store in memory
- Learning the value for each state individually is too slow

> Solution: Value Function approximation to generalize from seen states to unseen states

Value-Function approximation

Estimate Q or V with a parameter θ:

$$\hat{v}(s,\theta) \approx v_\pi(s)$$
$$\hat{q}(s,a,\theta) \approx q_\pi(s,a)$$

- The parameter θ is updated using MC or TD update

Parameterization can be any kind of estimator:

- Linear approximation
- Neural networks… > **Deep Reinforcement Learning**

Value Function approximation as Supervised Learning

Goal: finding parameter θ that minimizes the mean square error between true value function $v_\pi(s)$ and value function approximation $v_\theta(s)$

Loss: $$J(\theta) = \mathbb{E}_\pi\left(\left(v_\pi(s) - \hat{v}(s,\theta)\right)^2\right)$$

$v_\pi(s)$ is considered as the label/target like in a supervised learning setting

θ update with gradient descent:

$$\Delta\theta = -1/2\alpha\nabla_\theta J(\theta)$$

$$\theta_{k+1} \leftarrow \theta_k + \alpha(v_\pi(s) - \hat{v}(s,\theta))\nabla_\theta\hat{v}(s,\theta)$$

Value Function approximation as Supervised Learning

- How to get the supervised signal $v_\pi(s)$, i.e the true value function?
- It is the whole goal of RL to find this value function!

In practice we substitute a target for $v_\pi(s)$:

- In monte-Carlo, the target is the return G_t

$$\theta_{k+1} \leftarrow \theta_k + \alpha(G_t(s) - \hat{v}(s,\theta))\nabla_\theta \hat{v}(S_t,\theta)$$

- In TD learning, the target is the TD update

$$\theta_{k+1} \leftarrow \theta_k + \alpha(R_{t+1} + \gamma\hat{v}(S_{t+1},\theta) - \hat{v}(S_t,\theta))\nabla_\theta \hat{v}(s,\theta)$$

Deep Q-network

Learning to play Atari Games with DQN

- In Atari Games, the state space is every screenshot of your current position

- First paper to solve Atari Games with Deep Reinforcement Learning published by DeepMind: [Playing Atari Games with Deep Reinforcement Learning, Minh et al, 2013]
- Reaches super-human level of performance in three games

Atari Games RL model

State at each time-step t

- Last four screens concatenated together
- This allows information about movement
- The screens are pre-processed

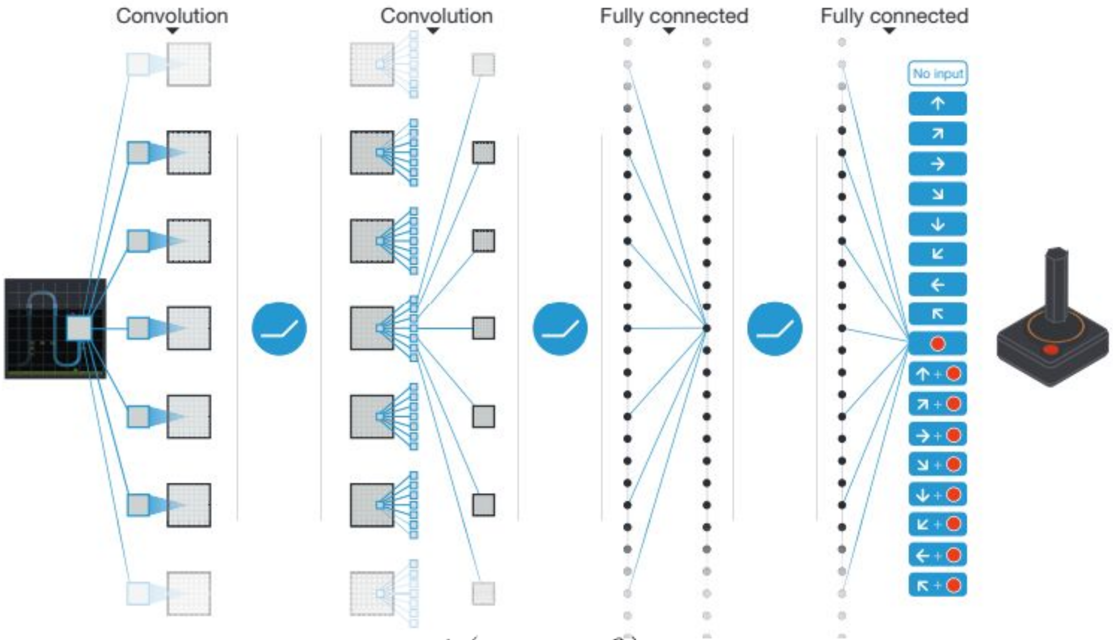
Reward: Score of the game, clipped to [-1,1]

Actions: Joystick buttons

Function approximation in Deep-Q network

- The function Q is approximated with a neural network

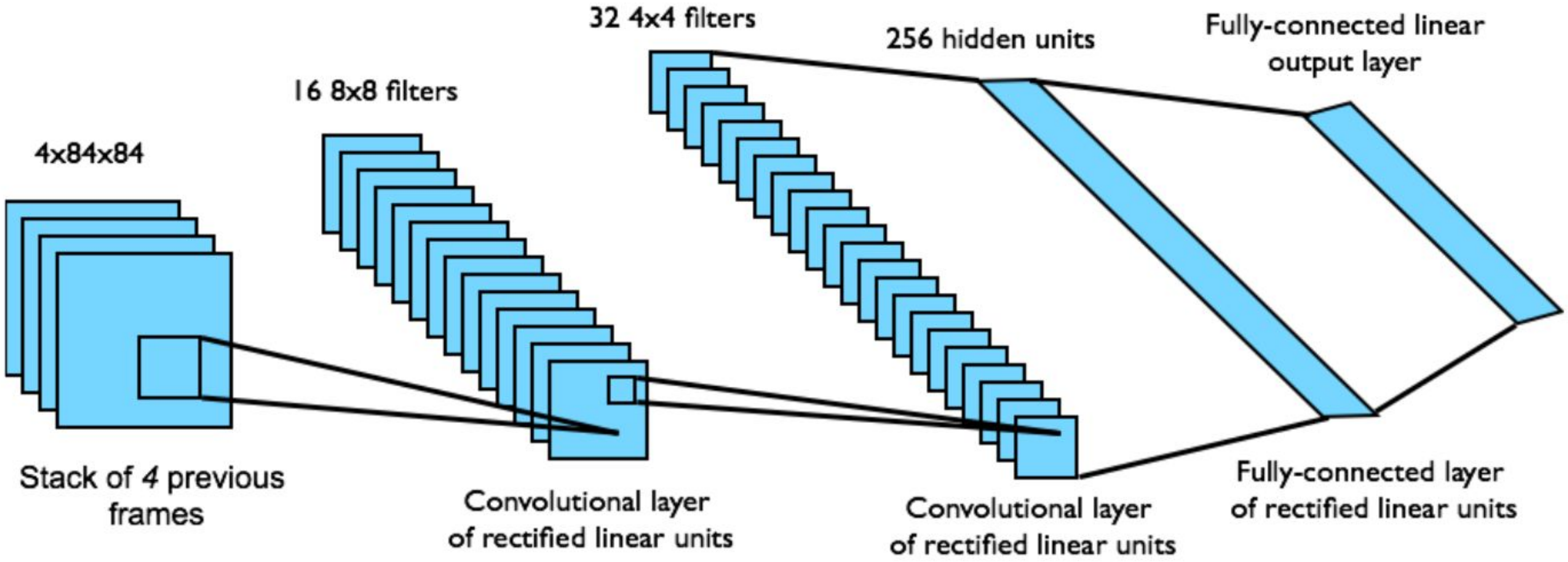
Stack of 4 screens concatenated Corresponding to one state



$$\hat{q}(s, a, \theta)$$

Classifier that outputs the Q value for each possible action

DQN architecture



Learning process of DQN with experience replay

- Idea: Learning $\hat{q}(s, a, \theta)$ like if we were in a supervised learning setting
- Training data = replay buffer with samples of timestep transitions
- "Labels" = Temporal Difference targets

Replay buffer

Sample uniform random

Training targets

$$[(s_0, a_0, r_1, s'_1), (s_1, a_1, r_2, s'_2), (s_2, a_2, r_3, s'_3), \ldots, (s_n, a_n, r_{n+1}, s'_{n+1})]$$

$$[Q(s_0, a_0) = r_1 + \gamma \max_a Q(s_1, a), \ldots, Q(s_2, a_2) = r_3 + \gamma \max_a Q(s_3, a)]$$

Advantages of experience replay

- Randomizes the sampling of training points -> allows to have identically independent distributed (iid) data

- Data Efficiency: we can learn from experience multiple times by sampling randomly from the replay buffer

Target network in DQN algorithm

- Idea: two parametrization of the DQN for the target and the Q function approximation: same architecture with different sets of weights

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[r + \gamma \max_{a'} Q(s',a';\underbrace{\theta_i^-}_{\text{target}}) - Q(s,a;\underbrace{\theta_i}_{\text{approx.}})^2\right]$$

- Set a older set of weights for computing the target (target network): The network is kept “frozen” for a certain number of steps

> **Trick to stabilise training**

Deep-Q network algorithm

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode $= 1, M$ **do**

Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

For $t = 1$,T **do**

With probability ε select a random action a_t

otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

Epsilon-greedy policy

Execute action a_t in emulator and observe reward r_t and image x_{t+1}

Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

Replay buffer

$$\text{Set } y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ \boxed{r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right)} & \text{otherwise} \end{cases}$$

Target network

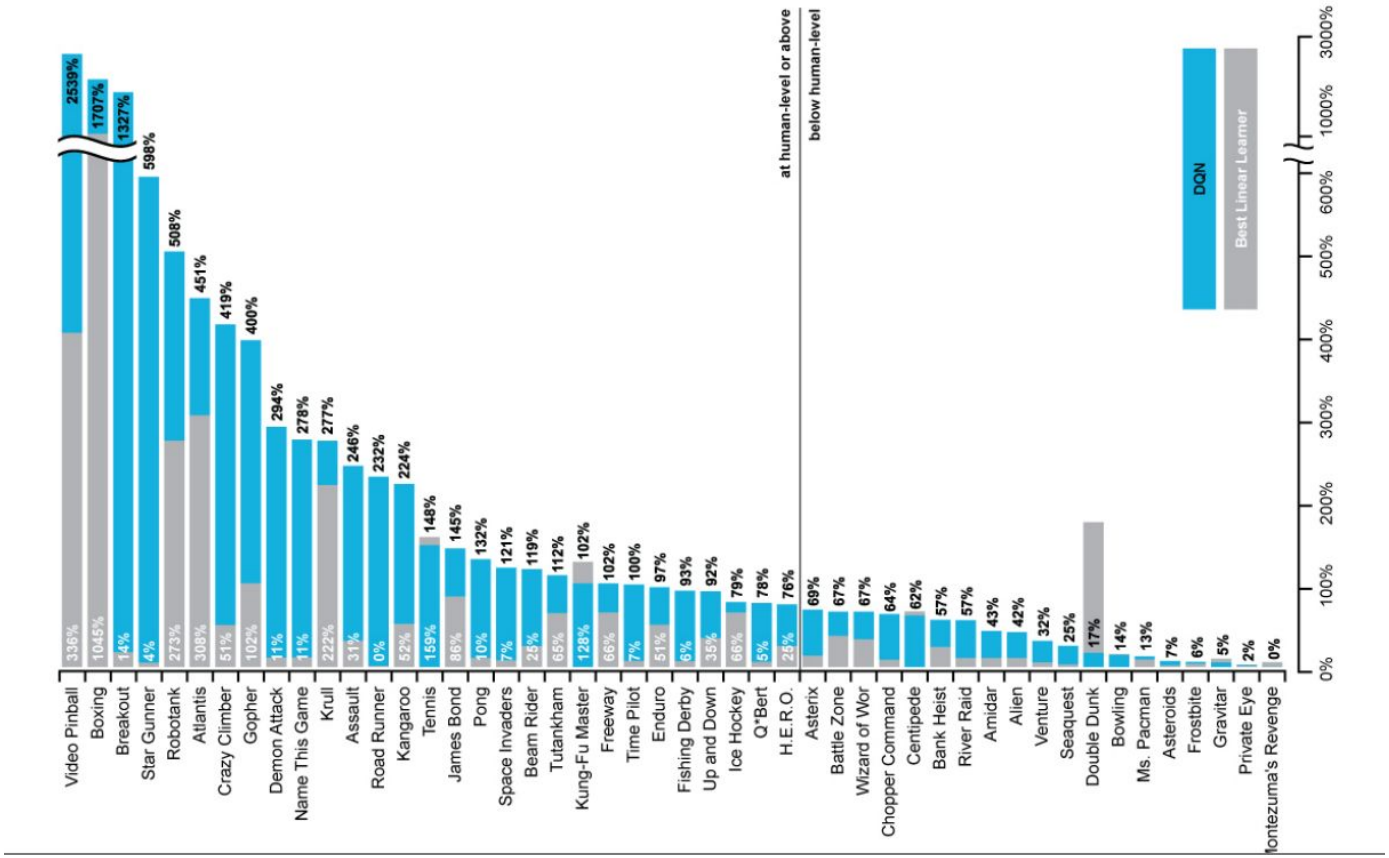
Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ

Every C steps reset $\hat{Q} = Q$

End For

End For

DQN results in Atari*



* From paper:

[Human-level control through reinforcement learning, Mnih et al, 2015]()

Part VI: Policy Gradient Algorithms

Policy Parametrization

- We saw previously function approximation on the value functions:

$$\hat{v}(s, \theta) \approx v_\pi(s)$$
$$\hat{q}(s, a, \theta) \approx q_\pi(s, a)$$

- A policy was then directly derived from the value function (a epsilon-greedy policy)
- In policy Gradient algorithms, we are going to parametrize the policy instead:

$$\pi_\theta(s, a) = \mathbb{P}\left[a \mid s, \theta\right]$$

Value-Based versus Policy-Based RL

Value-Based RL:

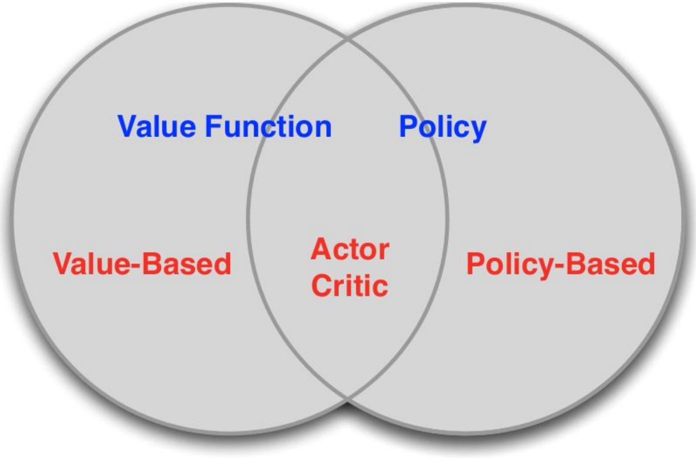
- Learn Value-Function
- Derive an implicit policy

Policy-Based RL:

- Directly learn a policy

Actor-Critic Methods:

- Learn a policy and learn a value-function



Motivation for Policy-Based RL

- Effective in high-dimensional or continuous action space: Value-Based RL methods are limited by the computation of the argmax over all actions

- Can learn optimal stochastic policy (ex: rock,paper, scissors game)

- Has better convergence properties

Policy Objective Function

- Goal of policy gradient algorithm: learn the parametrized policy directly: find the set of $\boldsymbol{\theta}$ that gives the best policy $\boldsymbol{\pi_\theta}$
- How do we measure the quality of a policy?

Policy Objective Function:

- Episodic Environments: Expected return from the initial state

$$J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta}\left[v_1\right]$$

- Continuing Environments: average return per time-step

Policy-Based RL as an optimisation problem

- Policy-Based RL is a optimisation problem: find $\boldsymbol{\theta}$ that maximizes J($\boldsymbol{\theta}$)
- Usually solve with gradient ascent:

$$\Delta\theta = \alpha \nabla_{\theta} J(\theta)$$

□ is the step size,

$\nabla_{\theta} J(\theta)$ is the gradient of the objective function J with respect to θ

Policy Gradient Theorem

- The policy Gradient Theorem gives the formula for the gradient of the objective function

Theorem

For any differentiable policy $\pi_\theta(s, a)$,
for any of the policy objective functions $J = J_1, J_{avR},$ *or* $\frac{1}{1-\gamma} J_{avV}$,
the policy gradient is

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s, a)\; Q^{\pi_\theta}(s, a)\right]$$

Policy Optimisation from the Policy Gradient Theorem

Policy Gradient Theorem: $\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}(\nabla_\theta \log \pi_\theta Q^{\pi_\theta}(s,a))$

Problem: How to estimate $Q^{\pi_\theta}(s,a)$?

- Monte-Carlo Methods: estimation with the return G_t

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}(\nabla_\theta \log \pi_\theta G_t)$$
$$\Delta\theta = \alpha \nabla_\theta \log \pi_\theta G_t$$

- Actor-Critic Methods: $Q^{\pi_\theta}(s,a)$ is parametrized with a function approximation

Policy Gradient with MC - REINFORCE algorithm

REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$
Repeat forever:
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
    For each step of the episode $t = 0, \ldots, T-1$:
        $G \leftarrow$ return from step t
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Actor-Critic Algorithm

Drawback of MC Policy Gradient: has high variance because is based on the total return of each episode

Instead, we can use a **Critic** to estimate the value function: $Q_w(s, a) \approx Q^{\pi_\theta}(s, a)$

Actor-Critic Algorithms are learning two sets of parameters:

- **Critic:** update action-value function with parameters w
- **Actor:** update his policy (towards the direction suggested by the critic) with parameters θ

Actor-Critic Algorithms follow an approximate gradient descent:

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s, a)\ \boxed{Q_w(s, a)}\right]$$

Evaluation of the policy by the critic

$$\Delta\theta = \alpha \nabla_\theta \log \pi_\theta(s, a)\ Q_w(s, a)$$

The actor updates its policy towards the direction suggested by the critic

Value Function Evaluation in Actor-Critic

The critic is solving the policy evaluation problem: how good is a policy π_θ?

To compute the value function evaluating this policy, we can use the classical reinforcement learning methods:

- Monte-Carlo prediction
- Temporal-Difference Learning

In practice, we subtract a baseline B(s) (depending only on the state s) to the action value-function Q(s,a): This baseline can be the value function V(s)

Example of a simple Actor-Critic Algorithm

- Simple actor-critic without baseline with action-value function
- Use a linear function approximation: $Q_w(s, a) = \phi(s, a)^\top w$
    - **CRITIC:** update w with linear TD-learning
    - **ACTOR:** update θ with policy gradient

function QAC
  Initialise s, θ
  Sample $a \sim \pi_\theta$
  for each step **do**
    Sample reward $r = \mathcal{R}_s^a$; sample transition $s' \sim \mathcal{P}_{s,\cdot}^a$
    Sample action $a' \sim \pi_\theta(s', a')$
    $\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$ TD(0) update
    $\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$ policy gradient update
    $w \leftarrow w + \beta \delta \phi(s, a)$
    $a \leftarrow a', s \leftarrow s'$
  end for
end function

Additional resources to dive deeper into Reinforcement Learning

Fundamentals of Reinforcement Learning

- David Silver class (DeepMind, one of the author of the AlphaZero) http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html

- Reference Book: Introduction to Reinforcement Learning, Sutton & Barto, 2017: http://incompleteideas.net/book/bookdraft2017nov5.pdf

On Deep Reinforcement Learning

- Deep Reinforcement Learning Bootcamp from Berkeley:

  https://sites.google.com/view/deep-rl-bootcamp/lectures

- Deep Reinforcement Learning - an overview: https://arxiv.org/pdf/1701.07274.pdf

- A lot of other resources here: https://github.com/AMDonati/RL-ressources-tutos-2018