UGE - Ifsttar / Cosys

Reinforcement Learning and Optimal Control - Master 2 SIA

Temporal-Difference, Q-learning & Deep Q-learning (DQN)

Nadir Farhi
chargé de recherche, UGE - Cosys/Grettia

UGE - 17 October 2024

Temporal-Difference (TD) Learning

- TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas.
  - Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics.
  - Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap).

  With the state-value function :

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right], \text{ for all } s \in \mathcal{S},$$

  The Bellman equation :

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_*(s')\Big], \end{aligned}$$

Temporal-Difference (TD) Learning

- Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap).

  With the action-value function :

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t=s, A_t=a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t=s, A_t=a\right].$$

  The Bellman equation :

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \,\middle|\, S_t=s, A_t=a\right] \\ &= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right], \end{aligned}$$

- For the control problem (finding an optimal policy), DP, TD, and Monte Carlo methods all use some variation of generalized policy iteration (GPI).
- The differences in the methods are primarily differences in their approaches to the prediction problem (policy evaluation).

TD prediction - Tabular TD(0)

- TD and MC methods use experience for prediction (policy evaluation).
- Given some experience following π, both methods update their estimate V of v_π for the nonterminal states S_t occurring in that experience.
- MC wait until the return following the visit is known, then use that return as a target for $V(S_t)$.

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

Input: a policy π to be evaluated

Initialize:

$V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$

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

Loop forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:

Append G to $Returns(S_t)$

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

TD prediction - Tabular TD(0)

- Constant-α Monte Carlo :

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[G_t - V(S_t)\Big],$$

- Whereas Monte Carlo methods must wait until the end of the episode to determine the increment to $V(S_t)$ (only then G_t is known),
- TD methods need to wait only until the next time step.
- Simplest TD method :

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

Tabular TD(0) or one-step TD

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

- TD(0) bases its update in part on an existing estimate
- We say that it is a *bootstrapping* method, like DP.

Tabular TD(0) or one-step TD

- Sample updates of DP are based on a complete distribution of all possible successors

$$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a)\big[r + \gamma V(s')\big]$$

- While those of TD(0) are based on a single sample successor.

$$V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$$

- The temporal difference is an error measuring the difference between the estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_{t+1})$:

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t).$$

Advantages of TD Prediction Methods

- TD methods have an advantage over DP methods :
  $\Rightarrow$: Do not require a model of the environment (reward & next-state prob. dist.)

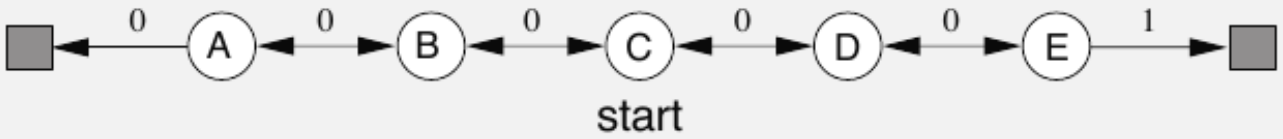
- TD methods have an advantage over Monte Carlo methods :
  $\Rightarrow$: Are naturally implemented in an online, fully incremental fashion.
  - With Monte Carlo methods one must wait until the end of an episode, because only then is the return known,
  - whereas with TD methods one needs to wait only one time step.
- Convergence :
  For any fixed policy π, TD(0) has been proved to converge to v_π.

- Which method TD or MC learns faster ?
  - This is an open question (no mathematical proof).
  - In practice, on stochastic tasks, TD converge faster than constant-α MC.

Exapmple - Random Walk

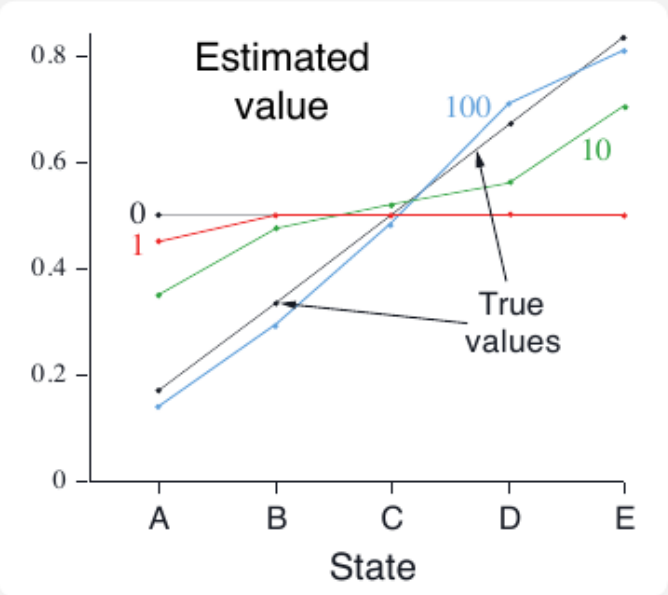
Example 6.2 Random Walk

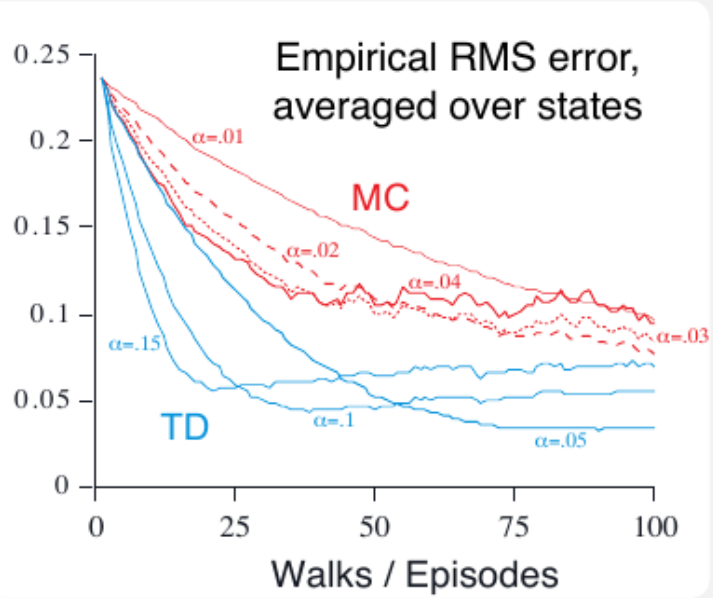
In this example we empirically compare the prediction abilities of TD(0) and constant-α MC when applied to the following Markov reward process:



A *Markov reward process*, or MRP, is a Markov decision process without actions. We will often use MRPs when focusing on the prediction problem, in which there is no need to distinguish the dynamics due to the environment from those due to the agent. In this MRP, all episodes start in the center state, C, then proceed either left or right by one state on each step, with equal probability. Episodes terminate either on the extreme left or the extreme right. When an episode terminates on the right, a reward of +1 occurs; all other rewards are zero. For example, a typical episode might consist of the following state-and-reward sequence: C, 0, B, 0, C, 0, D, 0, E, 1. Because this task is undiscounted, the true value of each state is the probability of terminating on the right if starting from that state. Thus, the true value of the center state is $v_\pi(\mathsf{C}) = 0.5$. The true values of all the states, A through E, are $\frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}$, and $\frac{5}{6}$.

Exapmple - Random Walk





The left graph above shows the values learned after various numbers of episodes on a single run of TD(0). The estimates after 100 episodes are about as close as they ever come to the true values—with a constant step-size parameter ($\alpha = 0.1$ in this example), the values fluctuate indefinitely in response to the outcomes of the most recent episodes. The right graph shows learning curves for the two methods for various values of α. The performance measure shown is the root mean square (RMS) error between the value function learned and the true value function, averaged over the five states, then averaged over 100 runs. In all cases the approximate value function was initialized to the intermediate value $V(s) = 0.5$, for all s. The TD method was consistently better than the MC method on this task.

Batch updating MC and TD(0)

- Suppose only a finite amount of experience is available.
- A common approach is to present the experience repeatedly until the method converges upon an answer.
- In batch updating, the value function is changed only once, by the sum of all the increments.
- Then all the available experience is processed again with the new value function to produce a new overall increment, and so on, until the value function converges.
- Under batch updating, both MC and TD(0) converge deterministically, as long as α is chosen to be sufficiently small.

Batch updating TD(0)

$\mathbb{D}$ is a dataset of experience

Batch updating:
Collect a dataset $\mathbb{D}$ of experience (somehow)
Initialize V arbitrarily
Repeat until V converged:

$$V' = V$$

For all $(s, a, s', r) \in \mathbb{D}$:

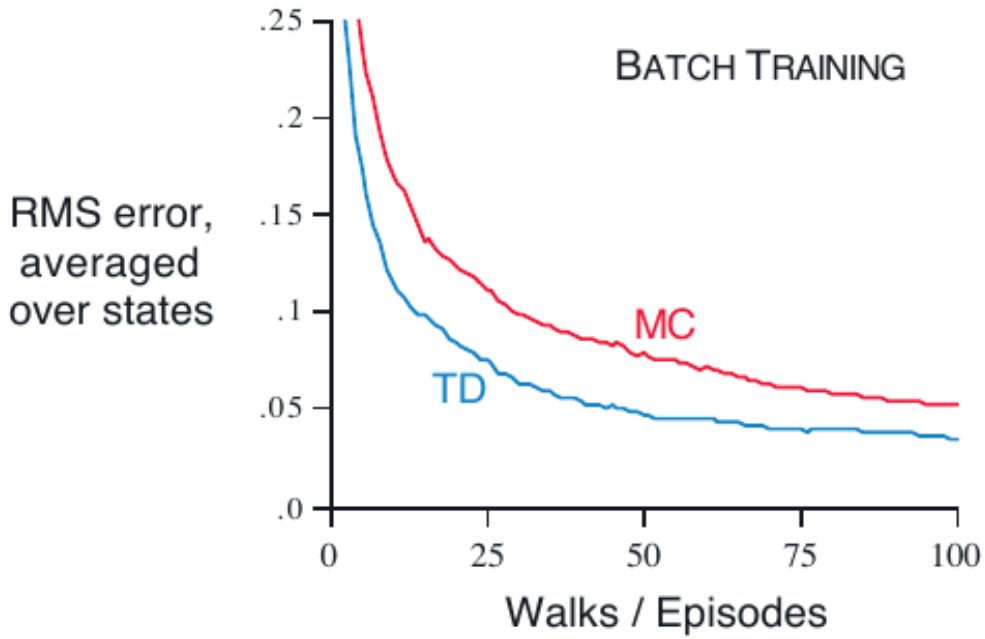
$$V'(s) = V(s) + \alpha[r + \gamma V(s') - V(s)]$$

$$V = V'$$

This integrates a bunch of TD steps into one update

Random walk under batch updating

- After each new episode, all episodes seen so far were treated as a batch.
- They were repeatedly presented to the algorithm, with small α.
- The resulting value function was then compared with v_π, and the average root mean square error across the five states (and across 100 independent repetitions of the whole experiment) was plotted.



- The batch TD method was consistently better than the batch MC method.

Batch MC vs Batch TD(0)

- Suppose we like to predict returns for an unknown Markov reward process, by observing the following 8 episods :

A, 0, B, 0	B, 1
B, 1	B, 1
B, 1	B, 1
B, 1	B, 0

- E.g. the 1st episode started in state A, transitioned to B with a reward of 0, and then terminated from B with a reward of 0.
- The optimal value for $V(B)$ should be 3/4.
- For $V(A)$:
  - Batch TD(0) gives $V(A) = 3/4$, since we have one obseravtion starting with A which transits to B with a reward 0 (takes into account the transitions).
  - Batch MC gives 0, since we have one observation starting with A which gives a reward of 0 (takes into account the data).

———————-

- Monte Carlo answer is better on the existing data.
- However, if the process is Markov, we expect that the TD(0) answer will produce lower error on future data.

Sarsa : On-policy TD Control

State-Action-Reward-State-Action

- Off-policy learning considers two policies :
  - Target policy : the one being learned and becomes the optimal policy.
  - Behavior policy : which is more eploratory and used to generate behavior.
- On-policy learning considers only one policy used to generate behavior and is learned and becomes the optimal pilicy.
- With off-policy methods, because the data is not due to the target policy, we have greater variance, and slow convergence.
- SARSA learns an action-value function rather than a state-value function.
- After every transition from a nonterminal state S_t, the following update is done :

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \Big].$$

- If S_t is terminal then $Q(S_t, A_t) = 0$.

Sarsa : On-policy TD Control

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Q-learning : Off-policy TD Control (Watkins, 1989)

Off-policy learning considers two policies :

- Target policy : the one being learned and becomes the optimal policy.
- Behavior policy : which is more eploratory and used to generate behavior.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop 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

Expected Sarsa

- Like Q-learning except that instead of the maximum over next state-action pairs it uses the expected value.

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \mathbb{E}_\pi[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t)\Big] \\ &= Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\Big], \end{aligned}$$

- π : a current policy derived from Q.
- It takes into account how likely each action is under the current policy.
- Given the next state S_{t+1}, the algorithm moves deterministically in the same direction as SARSA moves in expectation.
- Expected SARSA is more complex computationally, but, in return, it eliminates the variance due to the random selection of A_{t+1}.
- Expected SARSA performs slightly better than SARSA.

Double Q-learning (Maximization Bias)

- In the algorithms above, a maximum over estimated values is used implicitly as an estimate of the maximum value, which can lead to a significant positive bias.
- Example :
  - Consider s with many actions a whose true values $q(s, a)$ are all zero.
  - but whose estimated values, $Q(s, a)$, are uncertain and thus distributed some above and some below zero.
  - The maximum of the true values is zero,
  - while the maximum of the estimates is positive, a positive bias.
- The problem is the use of the same samples (plays) both to determine the maximizing action and to estimate its value.
- Suppose we divide the plays in two sets and used them to learn two independent estimates, $Q_1(a)$ and $Q_2(a)$, of the true value $q(a)$:
  - We use estimate Q_1 to determine the maximizing action $A_* = arg\max_a Q_1(a)$,
  - and use Q_2 to estimate its value $Q_2(A_*) = Q_2(arg\max_a Q_1(a))$.

Double Q-learning algorithm

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probabilility:

$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \operatorname{argmax}_a Q_1(S', a)\big) - Q_1(S, A)\Big)$$

    else:

$$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \operatorname{argmax}_a Q_2(S', a)\big) - Q_2(S, A)\Big)$$

    $S \leftarrow S'$
  until S is terminal

Neural Network based Q-learning

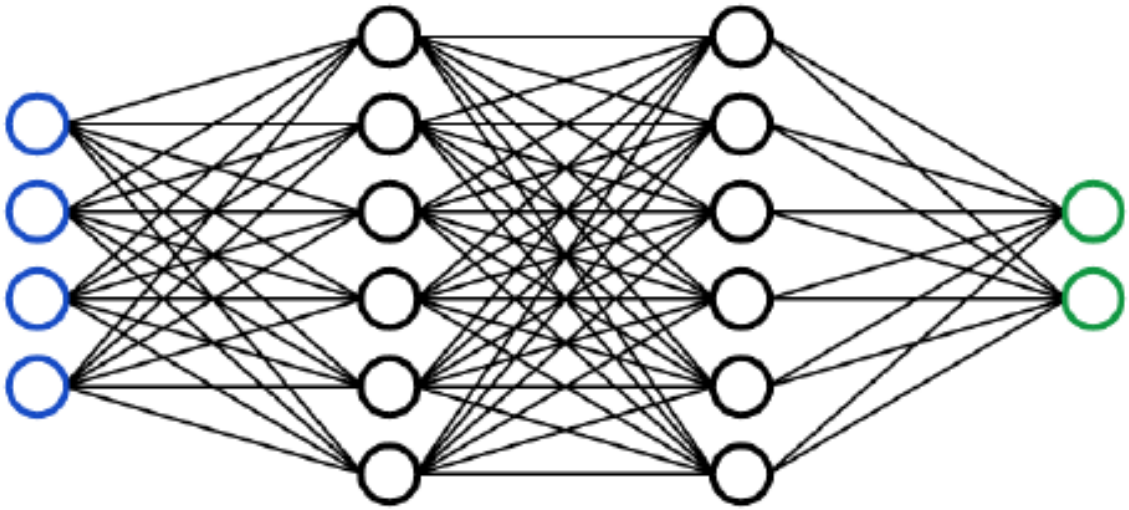


Figure – Multi-layered perceptron (MLP)

- Neural networks have been proposed as a solution for approximating the Q-function.
- However, it was found that non-linear approximators make Q-learning unstable and diverge in practice.

- Neural network based Q-learning was thought to be an insoluble problem for decades.
- The rise of deep learning in the recent years allowed researchers at Google DeepMind to came up in 2015 with the deep Q-network, or DQN.

Instability of NN based Q-learning

Main ideas :

- It introduces a Neural Network for the approximation of the action value function $Q(s, a)$.
- The Q network with weights Θ is a Q function approximator such that $Q_\pi(s, a, \Theta) \rightarrow Q^*_\pi(s, a)$.
- The Neural network maps from one state s to all the Q values $Q_\pi(s, a, \Theta), \forall a \in A$, for this state.
- The input layer is an s dimensional vector, in the continuous state space S.
- The output layer is an $|A|$ - dimensional vector, for the number $|A|$ of $Q_\pi(s, a, \Theta)$ of the $|A|$ possible actions in the discrete space A.
- $a = arg\max_a Q_\pi(s, a, \Theta)$: the optimal action following the policy $\pi^*(s, a)$.

continuous space state :

- In a Q-table, each value is estimated separately, and must be updated several times for optimization.
- A Q-table grow exponentially with the state space dimension.
- One solution is to introduce a neural network.
- However, non-linear approximations make Q-learning unstable, and it diverges in practice.
- 2015 : Researchers at Google DeepMind propose stable DQN.

Instability :

1. Moving targets : A neural network updates all its weights at each back propagation pass, thus updating the Q-values $Q_\pi(s, a)$ of the Q function.
   - Then any update can deeply alter the policy.
2. Successive correlated data.

Deep Q-learning

Issue 1 : Moving targets

- In supervised learning, target values are labeled beforehand, and are therefore the exact and definitive values of optimal hypotheses.
- In Q-learning however, the targets are constructed dynamically by temporal difference from the Q-values.
- With a Q-table, only the specific Q-value $Q_\pi(s, a)$ is updated at the given transition,
- While a neural network updates all its weights at each back propagation pass, thus updating all the Q-values $Q_\pi(s, a)$ of the Q-function.
- Consequently, the estimated temporal difference targets are constantly changing, or moving, and any update can deeply alter the policy.
- In order to solve the instability issue caused by this fact, it has been proved that creating a **second network** for the targets is quite efficient (Mnih et al. 2015).
- This second network, also called the **target network**, is a copy of the main network but not updated as often.

Issue 2 : Successively correlated data

- Another issue is the correlation between the consecutive samples of experience within a trajectory of transitions in the environment.
- The correlation causes catastrophic forgetting, where the neural network over-fits to new experiences and unlearns old training, after having significantly changed the weights.
- For this reason, a technique called **experience replay** is used during training.
- Using experience replay means that the agent's experience, meaning tuples of state-action-reward-next state at each time step is stored in a **replay memory** to be used later as training data.
- Sampling a **random mini batch** from this memory and use them for training helps to have a better representation of the data distribution.

Because of issues 1 and 2, NN-based Q-learning has been known to be unstable and diverge.

Deep Q-learning

DQN combines Q-learning with Deep learning.

1 - The target network addresses the moving targets issue.

- The target $\hat{Q}$-network $\hat{Q}$ with weights Θ^- is a second neural network, copied from the online Q-network periodically every C steps, such that $\Theta_t^- = \Theta_{\lceil t/C \rceil C}$, with C the update target frequency.
- The target $\hat{Q}$-network is never updated by gradient optimization.
- Its weights Θ^- are held fixed and used for generating fixed temporal difference targets y_t for the C next updates of the online Q-network, introducing a delay that reduces divergence.

2 - Replay memory addresses the successively correlated data issue

- The replay memory D is a data buffer storing the N last transitions experienced by the agent, $D = \{e_{t_1}, e_{t_2}, \ldots, e_{t_N}\}$, $e_t = (s_t, a_t, s_{t+1}, r_{t+1})$.
- At each time step t, the agent stores the experienced transition e_t in the replay memory, without updating the Q-network.
- It then draws uniformly at random M past transitions from the replay memory, and performs an update of the Q-network by back propagation over this batch.
- Before the beginning of learning, the replay memory is filled with random transitions.
- Hence, the bigger N the size of the replay memory, the lower the correlation probability.

Vanilla Algorithm

The target $y_t^{(m)}$, the temporal difference target by forward propagation through the target $\hat{Q}$-network $\hat{Q}$, and the hypothesis $h_{\Theta,t}^{(m)}$, the Q-value by forward propagation through the online Q-network Q, are computed for each transition $(s,a,s',r')^{(m)}$ in the batch $U(D)$:

$$y_t^{(m)} = r'^{(m)} + \gamma \cdot \max_{a'} Q_{\pi,t}(s'^{(m)}, a', \Theta_t^-),\ h_{\Theta,t}^{(m)} = Q_{\pi,t}(s^{(m)}, a^{(m)}, \Theta_t)$$

The online Q-network Q is updated by back propagation with a gradient descent step on the gradient Δ_t derived from the MSE loss over the temporal differences $\delta_t^{(m)} = y_t^{(m)} - h_{\Theta,t}^{(m)}$:

$$\begin{aligned} L_t(y_t, h_{\Theta,t}) &= \frac{1}{2M}\sum_{m=1}^{M}(y_t^{(m)} - h_{\Theta,t}^{(m)})^2 = L_t(\delta_t) = \frac{1}{2M}\sum_{m=1}^{M}(\delta_t^{(m)})^2 \\ &= \frac{1}{2M}\sum_{m=1}^{M}(r'^{(m)} + \gamma \cdot \max_{a'} Q_{\pi,t}(s'^{(m)}, a', \Theta_t^-) - Q_{\pi,t}(s^{(m)}, a^{(m)}, \Theta_t))^2 \end{aligned}$$

Finally, the weights Θ^- of the target $\hat{Q}$-network $\hat{Q}$ are reset by copy of the weights Θ of the online Q-network Q every C steps, such that $\Theta_t^- = \Theta_t$ and $\hat{Q}_t = Q_t$ if $t \equiv 0 \pmod{C}$.

Vanilla Algorithm

Vanilla DQN algorithm

Initialize step $t = 0$;
Initialize the online Q-network Q with random weights Θ_0;
Initialize the target $\hat{Q}$-network $\hat{Q}$ with weights $\Theta_0^- = \Theta_0$;
Initialize the replay memory buffer D to capacity N
with $Nmin$ random transitions $(s, \text{rand } a \in A, s', r')$;
for episode e = 1:E **do**
  Initialize sequence, observe initial state $s_t = \phi(x_t)$;
  while s_t not terminal **do**
    With probability ϵ select a random action $a_t \in A$
    otherwise select action $a_t = arg\max_a Q_{\pi,t}(s_t, a, \Theta_t)$;
    Execute a_t in emulator ε and observe reward r_{t+1} and next state $s_{t+1} = \phi(x_{t+1})$;
    Store transition $(s_t, a_t, s_{t+1}, r_{t+1})$ in D;
    Sample random batch $U(D)$ of M transitions $(s, a, s', r')^{(m)}$ in D;
    for m = 1:M **do**
      Set TD error $\delta_t^{(m)} = r'^{(m)} + \gamma \cdot \max_{a'} Q_{\pi,t}(s'^{(m)}, a', \Theta_t^-) - Q_{\pi,t}(s^{(m)}, a^{(m)}, \Theta_t)$;
    end for
    Perform a gradient descent step on MSE loss $L_t(\delta_t)$ w.r.t. Θ_t, with Δ_t, α;
    if $t \equiv 0 \pmod{C}$ **then**
      Reset target $\hat{Q}$-network $\hat{Q}$ = online Q-network Q, with $\Theta_t^- = \Theta_t$;
    end if
    Decay ϵ with linear decay;
    Increment step $t = t + 1$;
  end while
end for

RL Policy-based methods (or Policy Gradient Methods)

- RL value-based methods learn the values of actions and then select actions based on their estimated action values.
- Their policies would not even exist without the action-value estimates.
- RL Policy-based methods learn a parameterized policy that can select actions without consulting a value function.
- Parameterized policy : $\pi(a \mid s, \theta) = P(A_t = a \mid S_t = s, \theta_t = \theta)$.
- Methods can also learn a parameterized value function : $v(s, w)$.
- In general, policy-based methods learn the policy parameter based on the gradient of some scalar performance measure $J(\theta)$, with respect to the policy parameter θ.
- They maximize performance by updating θ with gradient ascent in J :

$$\theta_{t+1} = \theta_t + \alpha \hat{\nabla} J(\theta_t).$$

- $\hat{\nabla} J(\theta_t)$: stochastic estimate of the gradient of J w.r.t. θ_t.

Policy approximation

- In practice, to ensure exploration we generally require that the policy never becomes deterministic i.e. $\pi(a|s,\theta) \in (0,1), \forall s, a, \theta)$.
- Parameterization of discrete action spaces has some advantages over action-value methods.
- Policy-based methods also offer useful ways of dealing with continuous action spaces.
- Discrete and not too large action space :
  - Define a parameterized numerical preferences $h(s,a,\theta)$.
  - Then consider an exponential soft-max distribution :

$$\pi(a|s,\theta) := \frac{e^{h(s,a,\theta)}}{\sum_b e^{h(s,b,\theta)}}.$$

  - $h(s,a,\theta)$ can be parameterized arbitrarly, e.g. linear in features $x(s,a)$:

$$h(s,a,\theta) := \theta^T x(s,a).$$

  - They may also be computed by a deep artificial neural network (ANN), where θ is the vector of all the connection weights of the network.

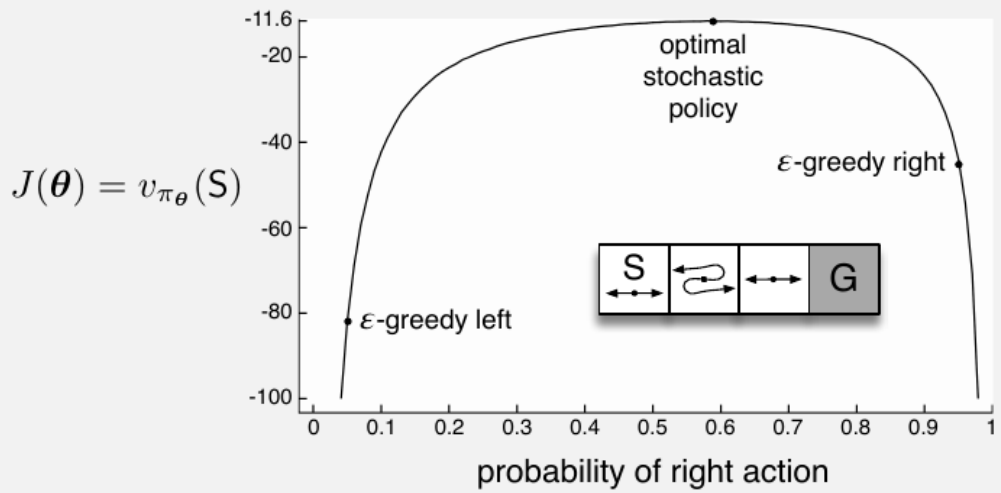
Advantages of policy approximation

- Approach a deterministic policy.
- Enable the selection of actions with arbitrary probabilities.
- The best approximate policy may be stochastic, e.g. in card games with imperfect information the optimal play is often to do two different things with specific probabilities, such as when bluffing in Poker.
- Action-value methods have no natural way of finding stochastic optimal policies.
- which is possible with policy approximating methods.
- With continuous policy parameterization the action probabilities change smoothly as a function of the learned parameter,
- Thus, stronger convergence guarantees are available for policy-gradient methods than for action-value methods.

Find stochastic optimal policies

Example 13.1 Short corridor with switched actions

Consider the small corridor gridworld shown inset in the graph below. The reward is -1 per step, as usual. In each of the three nonterminal states there are only two actions, right and left. These actions have their usual consequences in the first and third states (left causes no movement in the first state), but in the second state they are reversed, so that right moves to the left and left moves to the right. The problem is difficult because all the states appear identical under the function approximation. In particular, we define $\mathbf{x}(s, \text{right}) = [1, 0]^\top$ and $\mathbf{x}(s, \text{left}) = [0, 1]^\top$, for all s. An action-value method with ε-greedy action selection is forced to choose between just two policies: choosing right with high probability $1 - \varepsilon/2$ on all steps or choosing left with the same high probability on all time steps. If $\varepsilon = 0.1$, then these two policies achieve a value (at the start state) of less than -44 and -82, respectively, as shown in the graph. A method can do significantly better if it can learn a specific probability with which to select right. The best probability is about 0.59, which achieves a value of about -11.6.



Policy Gradient

- Episodic case : the performance measure is defined as the value of the start state of the episode :

$$J(\theta) := v_{\pi_\theta}(s_0).$$

- Policy gradient theorem for the episodic case :

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a \mid s, \theta)$$

$\propto$: proportional to

- Episodic case : constant of proportionality is the average length of an episode.
- Continuing case : constant of proportionality is 1 (equality).
- The distribution μ is the on-policy distribution under π (see Next slide).

Policy Gradient

The on-policy distribution in episodic tasks

In an episodic task, the on-policy distribution is a little different in that it depends on how the initial states of episodes are chosen. Let $h(s)$ denote the probability that an episode begins in each state s, and let $\eta(s)$ denote the number of time steps spent, on average, in state s in a single episode. Time is spent in a state s if episodes start in s, or if transitions are made into s from a preceding state $\bar{s}$ in which time is spent:

$$\eta(s) = h(s) + \sum_{\bar{s}} \eta(\bar{s}) \sum_{a} \pi(a|\bar{s}) p(s\,|\,\bar{s}, a), \quad \text{for all } s \in \mathcal{S}. \tag{9.2}$$

This system of equations can be solved for the expected number of visits $\eta(s)$. The on-policy distribution is then the fraction of time spent in each state normalized to sum to one:

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')}, \quad \text{for all } s \in \mathcal{S}. \tag{9.3}$$

This is the natural choice without discounting. If there is discounting ($\gamma < 1$) it should be treated as a form of termination, which can be done simply by including a factor of γ in the second term of (9.2).

REINFORCE : Monte Carlo Policy Gradient

- The later result gives an exact expression proportional to the gradient.
- Now we need to define some way of sampling whose expectation equals or approximates this expression.
- The right-hand side of the policy gradient theorem is a sum over states weighted by how often the states occur under the target policy π :

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a \mid s, \theta)$$

- If π is followed, then states will be encountered in these proportions :

$$\begin{aligned} \nabla J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}) \\ &= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\boldsymbol{\theta}) \right] . \end{aligned}$$

- All-actions algorithm :

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \sum_a \hat{q}(S_t,a,\mathbf{w}) \nabla \pi(a|S_t,\boldsymbol{\theta})$$

where $\hat{q}$ is some learned approximation to q_π.

REINFORCE : Monte Carlo Policy Gradient

- We introduce A_t in the same way as we ntroduced S_t.
- We replace a sum over the random variable's possible values by an expectation under π,
- and then we sample the expectation.

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &\propto \mathbb{E}_\pi\left[\sum_a \pi(a|S_t,\boldsymbol{\theta}) q_\pi(S_t,a) \frac{\nabla\pi(a|S_t,\boldsymbol{\theta})}{\pi(a|S_t,\boldsymbol{\theta})}\right] \\
&= \mathbb{E}_\pi\left[q_\pi(S_t,A_t)\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}\right] && \text{(replacing } a \text{ by the sample } A_t\sim\pi) \\
&= \mathbb{E}_\pi\left[G_t\,\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}\right], && \text{(because } \mathbb{E}_\pi[G_t|S_t,A_t]=q_\pi(S_t,A_t))
\end{aligned}$$

REINFORCE (Willams, 1992)

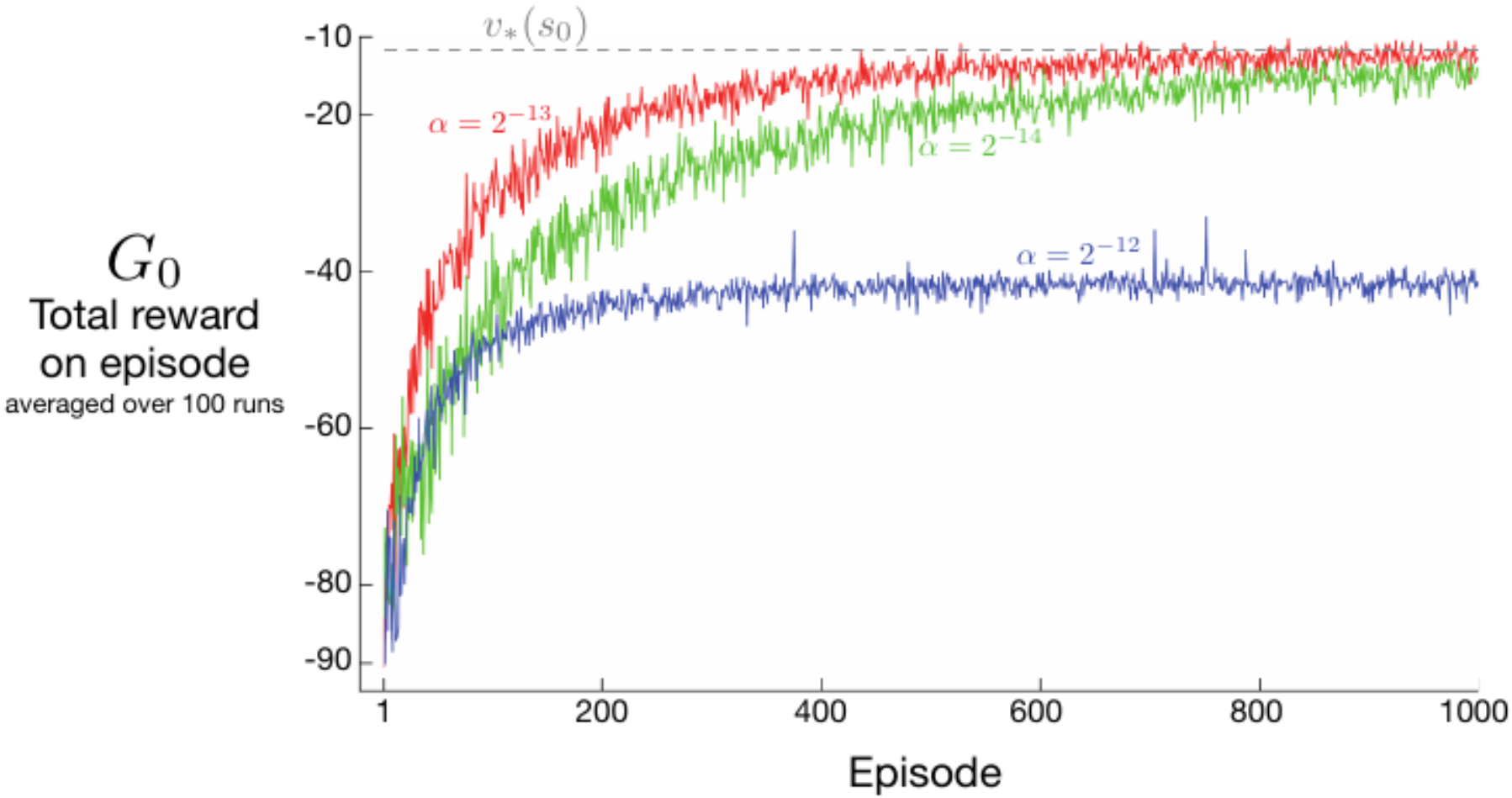
REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

- The update increases the parameter vector proportional to the return.
  (move most in the directions that favor actions that yield the highest return).
- and inversely proportional to the action probability.
  (otherwise actions that are selected frequently are at an advantage).

REINFORCE on the short-corridor gridworld (slide 30)



REINFORCE with baseline

- REINFORCE is generalized to include a comparison of the action value to an arbitrary baseline $b(s)$:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a \Big(q_\pi(s,a) - b(s) \Big) \nabla \pi(a|s,\boldsymbol{\theta}).$$

- As long as the baseline does not vary with a, the equation remains valid :

$$\sum_a b(s) \nabla \pi(a|s,\boldsymbol{\theta}) \;=\; b(s) \nabla \sum_a \pi(a|s,\boldsymbol{\theta}) \;=\; b(s) \nabla 1 \;=\; 0.$$

- New update rule :

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \Big(G_t - b(S_t) \Big) \frac{\nabla \pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}.$$

- In general, the baseline leaves the expected value of the update unchanged, but it can have a large effect on its variance.

REINFORCE with baseline

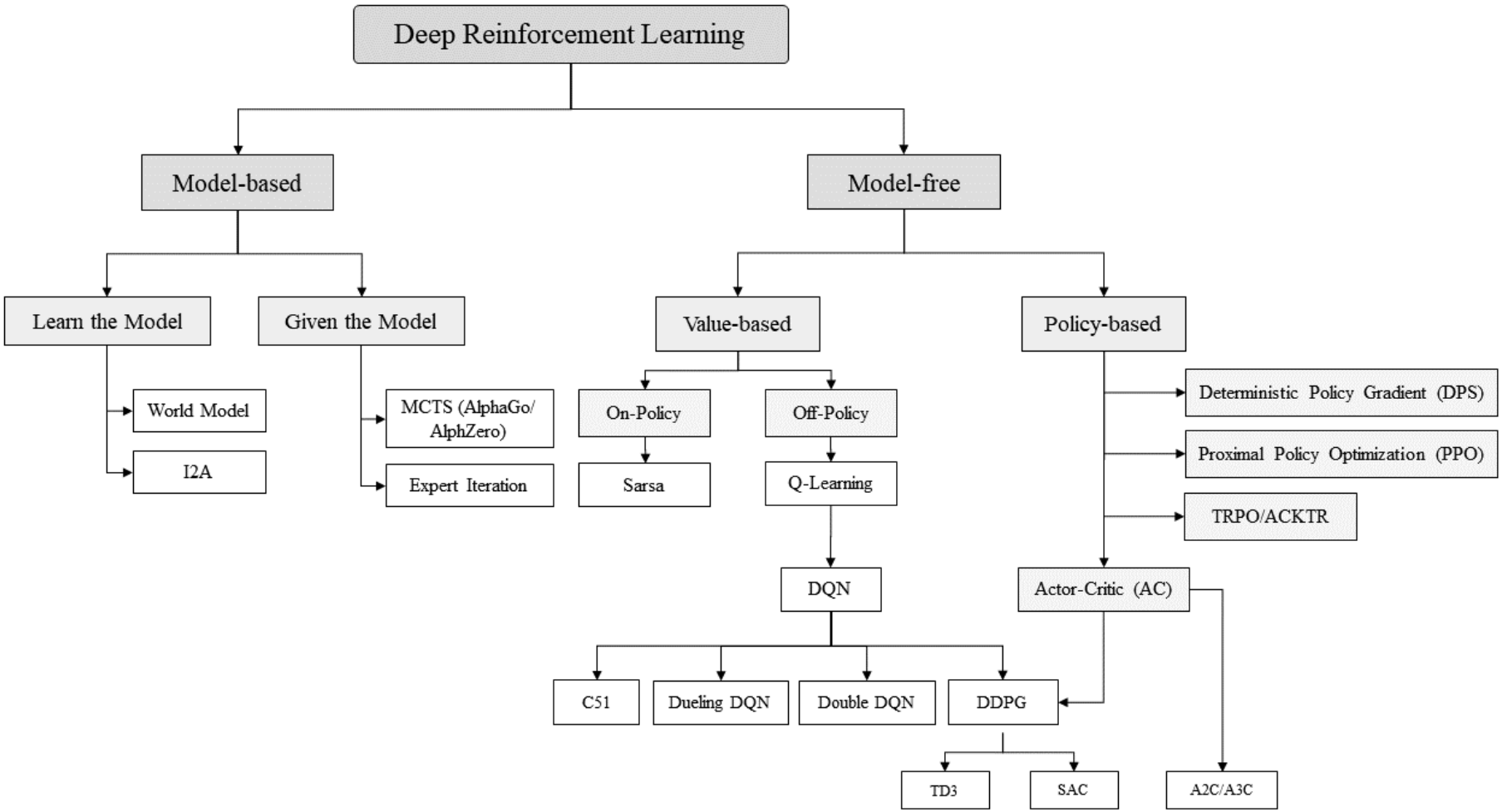
- One natural choice for the baseline is an estimate of the state value, $\hat{v}(S_t, w)$.
- It is natural to also use a Monte Carlo method to learn the state-value weights, w.

REINFORCE with Baseline (episodic), for estimating $\pi_\theta \approx \pi_*$

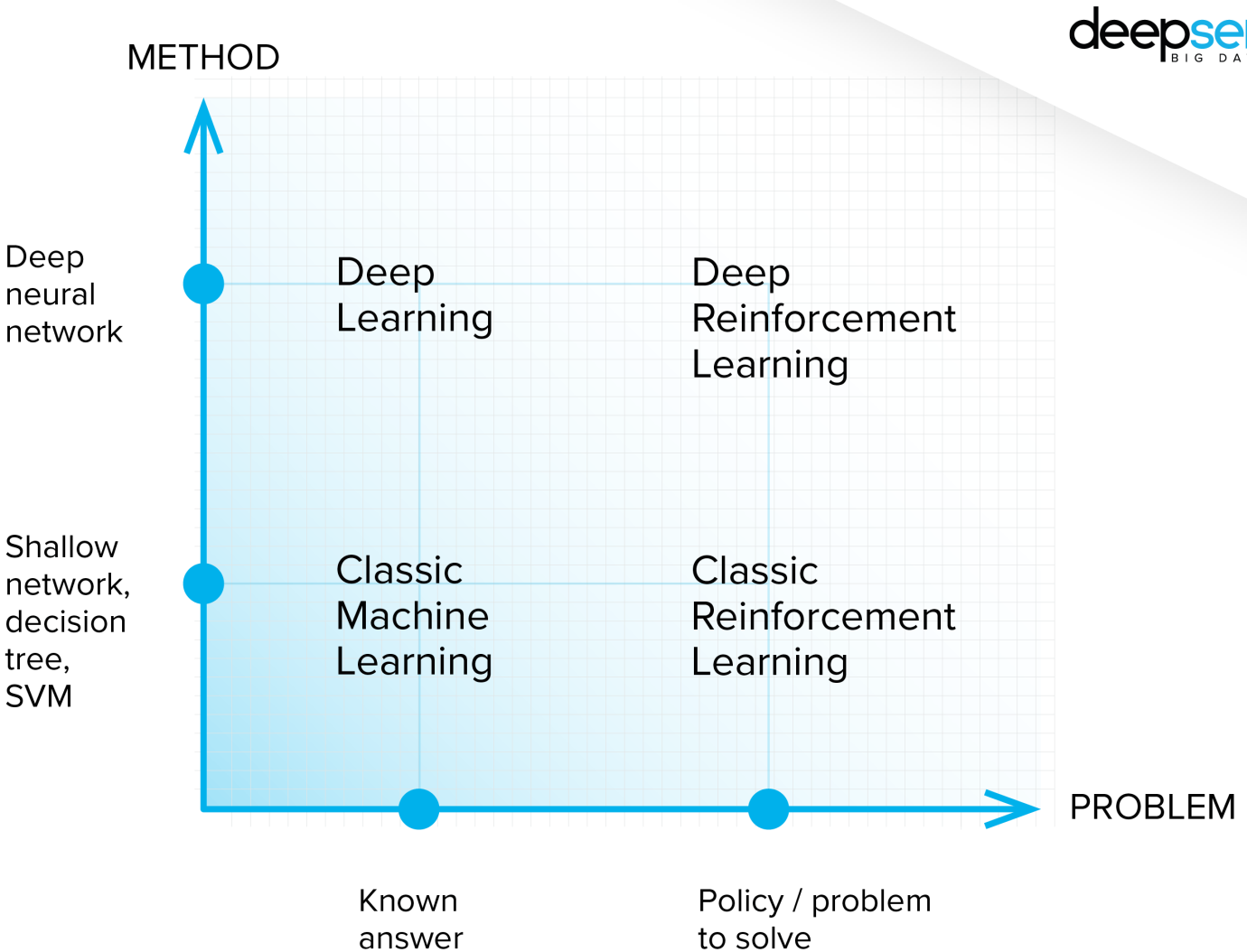
Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\qquad (G_t)$
    $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Deep Reinforcement Learning



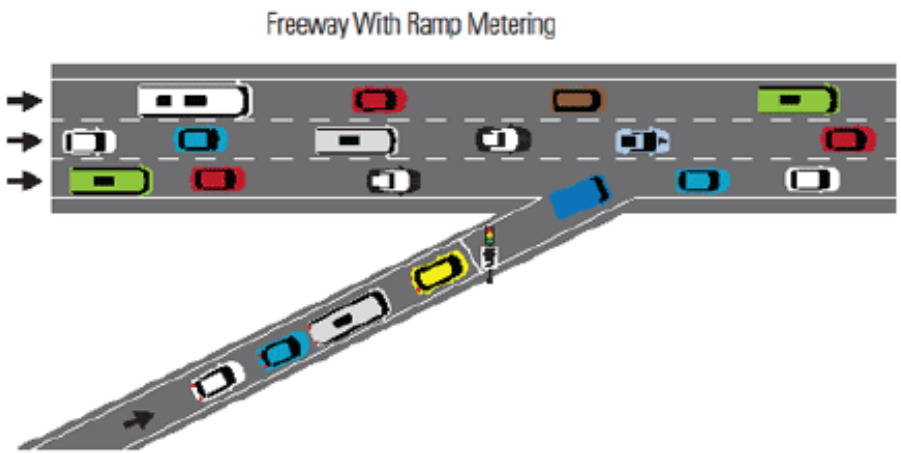
NN-based methods



Updates about the Project

Project 2023-2024 - Reinforcement learning for ramp metring on highways.

The objective of this project is to apply algorithms of Q-learning and Deep Q-learning to learn by numerical simulation the ramp metering control on a highway. We consider a stretch of highway with a given number of lanes (which is a parameter here), with an entering ramp controlled with a traffic light. We use the traffic simulator SUMO (Simulation of Urban Mobility) to simulate the car-following and lane change of all cars. The Q-learning algorithm should control the traffic light at the ramp, in a way that it optimizes the traffic, both on the highway stretch and on the ramp.



https://ops.fhwa.dot.gov

Updates about the Project

Possibility of usign the work of Romain Ducrocq :
- Projet 1: Framework DQN: https://github.com/romainducrocq/frameworQ/
- Projet 2: DQN for Intelligent Traffic Signal Control with Partial Detection: https://github.com/romainducrocq/DQN-ITSCwPD/
- Article sur Arxiv: https://arxiv.org/abs/2109.14337

State variables :
The state variable representation should reflect the state of traffic on both the highway, and the entering ramp.

Action variables :
The action variable can be for example the proportion of the green light in predefined cycle of the traffic light (cyclic control).

Reward :
The reward modeling should reflect the optimization of traffic on both the highway and the ramp.

Scenarios
Vary the values of the following parameters, in order to cover a maximum number of scenarios

- Length of the highway stretch
- Speed limits on the highway and on the ramp.
- Initial car-density on the highway stretch and on the ramp.
- The car-flow (veh./h.) on the highway, and the inflow from the input ramp.
- Number of lanes
- etc.