AIBT Reinforcement Learning session

Chapter 1

Introduction

1.0.1 Reinforcement learning

Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards [7].

One of the challenges that arise in reinforcement learning, is the trade-off between exploration and exploitation. To obtain a reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward. But to discover such actions, it has to try actions that it has not selected before. The agent has to exploit what it has already experienced in order to obtain reward, but it also has to explore in order to make better action selections in the future.

Markov Decision Processes

The problem of reinforcement learning using ideas from dynamical systems theory,specifically, as the optimal control of incompletely-known Markov decision processes (MDP). MDPs are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards. Thus MDPs involve delayed reward and the need to tradeoff immediate and delayed reward. In Markov Decision Processes we estimate $q_*(s, a)$ is the value value of taking action a in state s under the optimal policy.

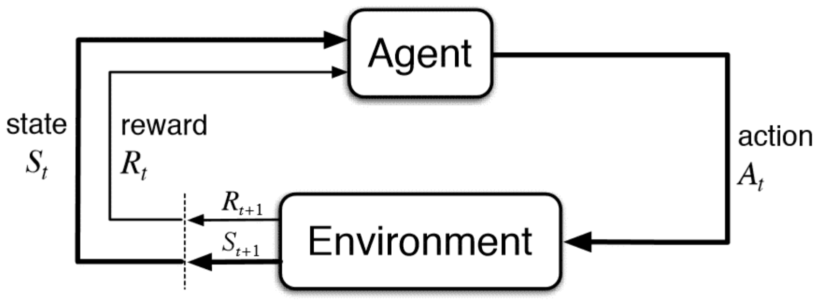


Figure 1.1: The agent–environment interaction in a Markov decision process [7]

The MDPs aim to provide a simple approach to the problem of learning from interaction to achieve the following objectives a goal. The learner and the decision-maker are called the agent. The thing he interacts with is called the environment. These interact continuously, with the agent selecting the actions and the environment responding to these actions and presenting new situations to the agent. The environment also return a reward values that the agent try to maximize.

At each time step $t \in 1, T$, the agent will take action s_t and the environment will return an state s_{t+1} and a reward $r_{t+1} \in \mathbb{R}$.

The MDP and the agent together give a trajectory that begins as follows :
$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, \ldots$

In a finite MDP, the sets of states, actions, and rewards (S, A, and R) all have a finite number of elements. In this case, the random variables R_t and S_t have defined discrete probability distributions dependent only on the preceding state and action. That is, for particular values of these random variables $s' \in S$ and $r \in R$, there is a probability of those values occurring at time t, given particular values of the preceding state and action:

$$p\left(s', r | s, a\right) \doteq \operatorname{Pr}\left\{S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a\right\} \tag{1.1}$$

By the law of total probability.

$$\textstyle\sum_{s' \in S} \sum_{r \in R} p\left(s', r | s, a\right) = 1$$

We can then calculate the reward expectation for each pair, state, action:

$$r(s,a) \doteq \mathbb{E}\left[R_t|S_{t-1}=s, A_{t-1}=a\right] = \sum_{r\in} r \sum_{s'\in S} p\left(s', r|s,a\right) \tag{1.2}$$

Or we can also calculate the expectation of the reward for the triplet state, action and next-state:

$$r\left(s,a,s'\right) \doteq \mathbb{E}\left[R_t|S_{t-1}=s, A_{t-1}=a, S_t=s'\right] = \sum_{r\in R} r \frac{p\left(s', r|s,a\right)}{p\left(s'|s,a\right)} \tag{1.3}$$

Reward and Gain

In reinforcement learning, the purpose or goal of the agent is formalized in terms of reward, passing from the environment to the agent. At each time step, the reward is a simple scalar, $R_t \in R$. Informally, the agent's goal is to maximize the total amount of reward it receives. This means maximizing not immediate reward, but cumulative reward in the episode.

In general, we seek to maximize expected returns, where the return, called Gt, is defined as a specific function of the reward sequence. In the simplest case is the sum of rewards :

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T = \sum_{k=1}^{T-k} R_{t+k} \tag{1.4}$$

Where T is the final step of the episode.

However, this version of the goal does not allow more weight to be given to shorter-term rewards and less to long-term rewards. To make an analogy, it is more interesting to earn 5 $ tomorrow than to earn 5 $ in a week.

To do this, you have to put a discount at each term and the next objective is largely preferred :

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^T R_T = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

(1.5)

$$0 < \gamma \leqslant 1$$

Policies and Value Functions

Some of the reinforcement learning algorithms must estimate the *value function*. This function estimates for each action and a given state how good the action is in terms of future rewards.

The policy is the decision-making function (control strategy) of the agent, which represents a mapping from situations to actions.

Now we can define for Markovian decision processes the value of a state s under a certain policy.

$$v_\pi(s) \doteq \mathbb{E}_\pi\left[G_t|S_t=s\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t=s\right], \forall s \in S \tag{1.6}$$

For any policy π and any state s, the following consistency condition holds between the value of s and the value of its possible successor states:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi\left[G_t|S_t=s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1}|S_t=s\right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p\left(s', r|s,a\right)\left[r + \gamma \mathbb{E}_\pi\left[G_{t+1}|S_{t+1}=s'\right]\right] \\ &= \sum_a \pi(a|s) \sum_{s',r} p\left(s', r|s,a\right)\left[r + \gamma v_\pi\left(s'\right)\right], \quad \forall s \in S \end{aligned} \tag{1.7}$$

We can also define the value of a pair of states, action under a certain policy s.

$$q_\pi(s,a) \doteq \mathbb{E}_\pi\left[G_t|S_t=s, A_t=a\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t=s, A_t=a\right], \forall s \in S \wedge a \in A \tag{1.8}$$

We call q_π the action-value function for policy π.

This last equation is Bellman's equation for v_π averages over all the possibilities, weighting each by its probability of occurring. It states that the value of the start state must equal the (discounted) value of the expected next state, plus the reward expected along the way. The value function v_π is the unique solution to its Bellman equation.

Optimal Policies and Optimal Value Functions

Solving a reinforcement learning problem means finding a policy that will give you as much reward as possible in the long term. We can also compare two policies to determine which is the best. In fact, a policy π is better than policy π' if for all states if the expected reward of π is greater than that of π'.
Formalisation:

$$\pi \geq \pi' \text{ if and only if } v_{\pi}(s) \geq v_{\pi'}(s), \forall s \in S \tag{1.9}$$

The policy with no policy greater than it is called the optimal policy denoted by π_*. The optimal state-value function, denoted v_*, and defined as:

$$v_*(s) \doteq \max_{\pi} v_{\pi}(s), \forall s \in S. \tag{1.10}$$

The optimal action-value function, denoted q_*, and defined as:

$$q_*(s,a) \doteq \max_{\pi} q_{\pi}(s,a) = \mathbb{E}\left[R_{t+1} + \gamma v_*\left(S_{t+1}\right) | S_t = s, A_t = a\right] \tag{1.11}$$

Temporal-Difference Learning

Temporal difference (TD) learning refers to a class of model-free reinforcement learning methods which learn by bootstrapping from the current estimate of the value function. These methods sample from the environment, like Monte Carlo methods, and perform updates based on current estimates, like dynamic programming methods [7].

Unlike Monte Carlo methods, time difference methods do not need to wait until the end of the episode in order to update the value function.

For Monte Carlo method:

$$V\left(S_t\right) \leftarrow V\left(S_t\right) + \alpha\left[G_t - V\left(S_t\right)\right] \tag{1.12}$$

For Temporal-Difference Learning:

$$V\left(S_t\right) \leftarrow V\left(S_t\right) + \alpha\left[R_{t+1} + \gamma V\left(S_{t+1}\right) - V\left(S_t\right)\right] \tag{1.13}$$

Second difference, Monte Carlo methods use an estimate of $v_{\pi}(s) = \mathbb{E}_{\pi}\left[R_{t+1} + \gamma G_{t+1} | S_t = s\right]$ as a target, whereas DP methods use an estimate of $v_{\pi}(s) = \mathbb{E}_{\pi}\left[R_{t+1} + \gamma v_{\pi}\left(S_{t+1}\right)\right]$ as a target.

We will accept as equals the one listed below:

$$\begin{aligned} v_{\pi}(s) &\doteq \mathbb{E}_{\pi}\left[G_t | S_t = s\right] \\ &= \mathbb{E}_{\pi}\left[R_{t+1} + \gamma G_{t+1} | S_t = s\right] \\ &= \mathbb{E}_{\pi}\left[R_{t+1} + \gamma v_{\pi}\left(S_{t+1}\right) | S_t = s\right] \end{aligned} \tag{1.14}$$

The main advantage of TD methods is that they can be done incrementally.

Q-Learning

Learn an action-value function rather than a state-value function. In particular, for an on-policy method we must estimate $q_{\pi}(s,a)$ for the current behavior policy π and for all states s and actions a. In contrast to the methods where we learned the state-state value now we consider an pair of state, action. Theoretically it comes back to the same thing, and the theorem and convergence is the same so we can apply the same algorithm as the TD method but now for a pair state, action.

$$Q\left(S_t, A_t\right) \leftarrow Q\left(S_t, A_t\right) + \alpha\left[R_{t+1} + \gamma Q\left(S_{t+1}, A_{t+1}\right) - Q\left(S_t, A_t\right)\right] \tag{1.15}$$

But one of the greatest advances in reinforcement learning has been the development of the Q-learning algorithm.

$$Q\left(S_t, A_t\right) \leftarrow Q\left(S_t, A_t\right) + \alpha\left[R_{t+1} + \gamma \max_{a} Q\left(S_{t+1}, a\right) - Q\left(S_t, A_t\right)\right] \tag{1.16}$$

In this case the algorithm an action value function that will directly estimate q_*.

Algorithm 1 Q-Learning algorithm

```
Initialize: $Q(s,a), \forall s \in S, a \in A(s), arbitrarily, \wedge Q(terminal - state, .) = 0$
foreach episode do
    Initialize: $S$
    while episode not done do
        Choose $A$ from $S$ using policy derived from $Q$( e.g. , e-greedy )   Take action $A$, observe $R, S'$
        $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q\left(S', a\right) - Q(S,A)\right]$
        $S \leftarrow S'$
    end
end
```

Q-learning involve maximization in the construction of their target policies. In these algorithms, a maximum over estimated values is used implicitly as an estimate of the maximum value, which can lead to a significant positive bias.

To overcome this problem there is the Double Q-Learning algorithm. The key idea is to learn two value functions so that we no longer use the same function to select and evaluate an action [1].

Algorithm 2 Double Q-Learning algorithm

```
Initialize: $Q_1(s,a) \wedge Q_2(s,a), \forall s \in S, a \in A(s)$, arbitrarily
Initialize : $Q_1(terminal-state,.) = Q_1(terminal-state,.) = 0$
foreach episode do
    Initialize: $S$
    while episode not done do
        Choose $A$ from $S$ using policy derived from $Q_1$ and $Q_2$ ( e.g., $\varepsilon$ -greedy in $Q_1+Q_2$)
        Take action $A$, observe $R, S'$
        if $random.uniform(0,1) > 0.5$ then
            $Q_1(S,A) \leftarrow Q_1(S,A) + \alpha\left(R + \gamma Q_2\left(S', \arg\max_a Q_1\left(S',a\right)\right) - Q_1(S,A)\right)$
        else
            $Q_2(S,A) \leftarrow Q_2(S,A) + \alpha\left(R + \gamma Q_1\left(S', \arg\max_a Q_2\left(S',a\right)\right) - Q_2(S,A)\right)$
        end
        $S \leftarrow S'$
    end
end
```

Deep Q-Learning and variation

The reinforcement learning algorithms presented above are not effective when the observation space is high and the great successes by reinforcement learning were because there were handmade features.

In 2013 a team of DeepMind researchers succeeded in reaching a human level on Atari games using raw pixels as input. This incredible achievement came about with the clever combination of Q-learning to learn a control strategy and deep learning to deal with high dimensional data such as images.
Also the second advantage of using deep learning, is that the algorithm itself learns the features so there is no longer any need to manually design them
This method is the Deep Q-Learning Network (DQN) [3].
The basic idea behind many reinforcement learning algorithms is to estimate the action-value function, by using the Bellman equation as an iterative update

$$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i\left(s',a'\right)|s,a\right] \tag{1.17}$$

In practice, this approach is impractical, because the action-value function is estimated separately for each sequence, without any generalisation. Instead, it is common to use a function approximator to estimate the action-value function $Q(s,a;\theta) \approx Q^*(s,a)$ where θ is the weights of the neural network. A Q-network can be trained by minimising a sequence of loss functions $L_i\left(\theta_i\right)$ that changes at each iteration i,

$$L_i\left(\theta_i\right) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[\left(y_i - Q\left(s,a;\theta_i\right)\right)^2\right] \tag{1.18}$$

where $y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q\left(s',a';\theta_{i-1}\right)|s,a\right]$
Differentiating the loss function with respect to the weights we arrive at the following gradient :

$$\nabla_{\theta_i} L_i\left(\theta_i\right) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q\left(s',a';\theta_{i-1}\right) - Q\left(s,a;\theta_i\right)\right)\nabla_{\theta_i} Q\left(s,a;\theta_i\right)\right] \tag{1.19}$$

Transitions are stored in a buffer and then randomly buffer to be presented to the Q-network. This is because sequential experiences are highly correlated (temporally) with each other. In statistical learning and optimization tasks, we want our data to be independently distributed. That is, we don't want the data we are feeding to be correlated with each other in any way. Random sampling of experiences breaks this temporal correlation of behavior and distributes/averages it over many of its previous states. By doing so, we avoid significant oscillations or divergence in our model—problems that can arise from correlated data.

Algorithm 3 Deep Q-Learning algorithm

```
Initialize: Replay memory D to capacity N
Initialize: Initialize action-value function Q with random weights
foreach episode do
    Initialize:  sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi\left(s_1\right)$
    while episode not done do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*\left(\phi\left(s_t\right), a; \theta\right)$
        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 $\phi_{t+1} = \phi\left(s_{t+1}\right)$
        Store transition $\left(\phi_t, a_t, r_t, \phi_{t+1}\right)$ in $D$
        Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from $D$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q\left(\phi_{j+1}, a'; \theta\right) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$
    end
end
```

Policy gradient

Another way of solving reinforcement learning problem is to learn an optimal policy π^* function which associates a probability distribution on the action space $\pi^*(.|s_t)$ with each state s_t. These algorithms, which consist in finding a probability distribution on the possible actions for each state, are called **policy-based** algorithms [8].

First, we define an agent trajectory τ_θ by :

$$\tau_\theta = (s_0, a_0, s_1, a_1, \dots, s_{T+1}) \tag{1.20}$$

it's the sequence of states and actions that the agent follows in an episode under a certain policy π_θ.

Next, we need to determine how well the agent performed in an episode. To do this, we will use a criterion named **the finite-horizon discounted return** which is defined as :

$$G(\tau_\theta) = \sum_{t=0}^{T} \gamma^t r_t \tag{1.21}$$

This is the sum of the discounted rewards that the agent receives in the current episode by following policy π_θ. The following derivation is the same for infinite horizon $T = \infty$.

We can then define a policy performance metric for how good the policy is as :

$$J(\pi_\theta) = \mathbb{E}[G(\tau_\theta)] \tag{1.22}$$

This is the expected return at the end of each episode by following policy π_θ. If we randomly initialize θ, run a few episodes, and store the return at the end of each episode, we will most likely find that the average of the returns is low, that is, the policy is bad. However, from calculus, we know that the gradient of a function always points in the direction of the greatest rate of increase. **If we can calculate** the gradient of $J(\pi_\theta)$ with respect to the policy parameters, θ, we can update θ according to :

$$\theta = \theta + \alpha \nabla_\theta J(\pi_\theta) \tag{1.23}$$

We can approximate the gradient of $J(\pi_\theta)$ using the **policy gradient theorem**. A short proof is given here. We want to calculate the gradient of $J(\pi_\theta)$ define as bellow :

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \nabla_\theta \mathbb{E}[G(\tau_\theta)] \\
&= \nabla_\theta \int_\tau P(\tau \mid \theta) G(\tau) \\
&= \int_\tau \nabla_\theta P(\tau \mid \theta) G(\tau)
\end{aligned}$$

Now we will use a famous trick in reinforcement learning called the log-derivative trick :

$$\nabla_\theta \log P(\tau \mid \theta) = \frac{\nabla_\theta P(\tau \mid \theta)}{P(\tau \mid \theta)} \Leftrightarrow \nabla_\theta P(\tau \mid \theta) = \nabla_\theta \log P(\tau \mid \theta) P(\tau \mid \theta) \tag{1.24}$$

Because for all f positive and differentiable :

$$\nabla \log f = \frac{\nabla f}{f}$$

We can now express $\nabla_\theta J(\pi_\theta)$ using $\log P(\tau \mid \theta)$:

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \int_\tau P(\tau \mid \theta) \nabla_\theta \log P(\tau \mid \theta) G(\tau) \\
&= \mathbb{E}[\nabla_\theta \log P(\tau \mid \theta) G(\tau_\theta)]
\end{aligned}$$

The probability of following trajectory τ under the current policy is the product of the probability of starting in state s_0, the probability of taking action a_0 under the current policy, and the probability of transitioning to state s_1 after taking action a_0, and so on.

$$P(\tau \mid \theta) = \rho_0(s_0) \prod_{t=0}^{T} P(s_{t+1} \mid s_t, a_t) \pi_\theta(a_t \mid s_t) \Leftrightarrow \log(P(\tau \mid \theta)) = \log\left(\rho_0(s_0) \prod_{t=0}^{T} P(s_{t+1} \mid s_t, a_t) \pi_\theta(a_t \mid s_t)\right)$$

By the logarithm product group :

$$\begin{aligned}
P(\tau \mid \theta) &= \log(\rho_0(s_0) P(s_1 \mid s_0, a_0) \pi_\theta(a_0 \mid s_0) \dots P(s_{T+1} \mid s_T, a_T) \pi_\theta(a_T \mid s_T)) \\
&= \log \rho_0(s_0) + \log P(s_1 \mid s_0, a_0) + \log \pi_\theta(a_0 \mid s_0) + \dots + \log P(s_{T+1} \mid s_T, a_T) \\
&= \log \rho_0(s_0) + \sum_{t=0}^{T} (\log P(s_{t+1} \mid s_t, a_t) + \log \pi_\theta(a_t \mid s_t))
\end{aligned}$$

For all $i \in \{0, \dots, T\}$, $\log(\rho_0(s_0))$ and $P(s_{i+1} \mid s_i, a_0)$ doesn't depend on θ :

$$\begin{aligned}
\nabla_\theta P(\tau \mid \theta) &= \nabla_\theta \left(\log \rho_0(s_0) + \sum_{t=0}^{T} (\log P(s_{t+1} \mid s_t, a_t) + \log \pi_\theta(a_t \mid s_t))\right) \\
&= \nabla_\theta \sum_{t=0}^{T} \log \pi_\theta(a_t \mid s_t) \\
&= \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)
\end{aligned}$$

Now we can rewrite $\nabla_\theta J(\pi_\theta)$ as follow :

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, G(\tau_\theta)\right] \tag{1.25}$$

When we use $G(\tau_\theta)$ to evaluate how good an action was, we call that algorithm **REINFORCE** :

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \sum_{t'=t}^{T} \gamma^{t'} r_{t'}\right] \tag{1.26}$$

Algorithm 4 REINFORCE algorithm

```
1 : Initialize the policy parameter $\theta$ at random
2 : Generate one trajectory on policy $\{S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T\} \sim \pi_\theta$
for $t = 1, 2, \ldots, N$ do
    3 : Estimate the return $G_t$
    4 : Update policy parameter: $\theta \leftarrow \theta + \alpha G_t \nabla_\theta \ln \pi_\theta(A_t \mid S_t)$
end
```

Actor-Critic

As seen before, there are two classes of reinforcement learning algorithms :

- Value based: they try to find or approximate the optimal value function, which is a mapping between an action and a value. The higher the value, the better the action (Q-learning, Deep Q-learning, TD learning)
- Policy based: policy based algorithms like policy Gradients try to find the optimal policy directly without the Q -value as a middleman.

Each method has advantages. For example, policy-based are better for continuous and stochastic environments, have a faster convergence, while value-based are more sample efficient and steady.

The idea behind the actor-critic algorithm [2] is to merge that these classes of algorithms. The actor-critic algorithm aims to take advantage of the benefits from both value-based and policy-based while eliminating their drawbacks.

More precisely, in a vanilla policy gradient such as REINFORCE, we generally use a softmax based policy. Instead of estimating the Q values of the actions, we assign arbitrary preferences $H(a)$ to the action. What's interesting is that they don't correspond to anything. A higher preference means a higher probability to be selected. But unlike Q values, preferences cannot be interpreted. The policy gradient theorem tells you in which direction you should update your preferences from a state given an action and return. But, again the preferences are not explicitly trying to compute anything. It's just saying "if that action did good, raise its probability" (therefore its preference, which decreases the probability of all other actions, but not their preferences); if the action did bad, lower its probability (decreasing its preference, therefore increasing the probability of all other actions without increasing preference).

Now if you think about it, increasing or decreasing the action probabilities based on the full return without comparing it to previous returns means that if an action got a positive return, you're still going to increase its preference even though other actions do much better. This is super naive and it just makes more sense to compare it to how much you could've gotten: the idea of a baseline. The best thing to use as a baseline is the best estimate of the return from a state you can get - so by definition, the value function. This is the first place where value and policy based methods meet."

The main idea is to split the model in two: one for computing an action based on a state and another one to produce the Q values of the action.

The actor takes as input the state and outputs the best action. It essentially controls how the agent behaves by learning the optimal policy (policy-based). The critic, on the other hand, evaluates the action by computing the value function (value-based). These two models participate in a game where they both get better in their respective role as time passes. The result is that the overall architecture will learn to play the game more efficiently than the two methods separately.

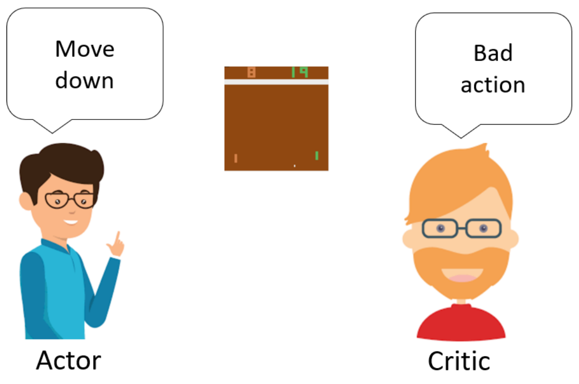


Figure 1.2: Philosophy of actor-critic architectures : the actor network takes actions, the critic network quantifies the value of the actions taken by the actor and enables them to be adjusted by optimisation

The following loss functions are an example of the policy gradient function and actor-critic function

- Vanilla policy gradient REINFORCE : $\nabla_\theta J(\pi_\theta) = \mathbb{E}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \sum_{t'=t}^{T} \gamma^{t'} r_{t'}\right]$. Only one network is involved, the actor network π_θ. The probability of selecting the action a_t in the state s_t is increased if the discounted return $\sum_{t'=t}^{T} \gamma^{t'} r_{t'}$ is positive.

- Baseline policy gradient : $\nabla_\theta J(\pi_\theta) = \mathbb{E}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \sum_{t'=t}^{T} \gamma^{t'} r_{t'} - b(s_t)\right]$. Only one network is involved, the actor network π_θ. The probability of selecting the action a_t in the state s_t is increased if the discounted return $\sum_{t'=t}^{T} \gamma^{t'} r_{t'}$ is higher that a given baseline $b(s_t)$.

- Q-value actor-critic : $\nabla_\theta J(\pi_\theta) = \mathbb{E}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) Q^\pi(s_t, a_t)\right]$. Two networks are involved, the actor network π_θ and the critic network Q^π . The probability of selecting the action a_t in the state s_t is increased if the Q-value of the state-action pair $Q^\pi(s_t, a_t)$ is positive.

- TD residual actor-critic : $\nabla_\theta J(\pi_\theta) = \mathbb{E}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)(r_t + V^\pi(s_{t+1}) - V^\pi(s_t))\right]$. Two networks are involved, the actor network π_θ and the critic network V^π . The probability of selecting the action a_t in the state s_t is increased if the TD-error $r_t + V^\pi(s_{t+1}) - V^\pi(s_t)$ is positive. If this error is positive, it means that the state s_{t+1} reached by the action a_t is a better state than the previous state s_t, so the action a_t was a good action. In this case, we increased the probability of selecting a_t in s_t.

- Advantage actor-critic : $\nabla_\theta J(\pi_\theta) = \mathbb{E}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)(Q^\pi(s_t, a_t) - V^\pi(s_t))\right]$. Two networks are involved, the actor network π_θ and the critic network V^π . The probability of selecting the action a_t in the state s_t is increased if the advantage $Q^\pi(s_t, a_t) - V^\pi(s_t)$ is positive. We have that $V^\pi(s_t) = \sum_{a \in \mathcal{A}} \pi_\theta(a \mid s_t) Q^\pi(s_t, a)$ so if the advantage is positive, it means that the Q-value of state-action pair at time t is higher than the average Q-value at time t. So in this case we increased the probability of selecting a_t in s_t.

Algorithm 5 Actor critic using ψ as advantage algorithm

```
1 : Initialize the policy parameter $\theta$and the value parameter$\phi$ at random
2 : Generate one trajectory on policy $\{S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T\} \sim \pi_\theta$
for $t = 1, 2, \ldots, N$ do
    3 : Estimate the return $G_t$
    4 : Update policy parameter: $\theta \leftarrow \theta + \alpha\psi(V^\pi(S_t))\nabla_\theta \ln \pi_\theta(A_t \mid S_t)$
end
```

Proximal Policy Optimization

During the project the algorithm mainly used to optimise the strategy of the agents is the **PPO (Proximal Policy Optimization)** algorithm [5]. This algorithm is considered the best performing optimisation algorithm in the vast majority of cases.

PPO is the evolution of an algorithm called **TRPO (Trust Region Policy Optimisation)** [6]. The philosophy behind these two algorithms is the same: to optimise the strategy of the agents in a stable way and thus to avoid sudden changes in strategy which could lead to a collapse in performance.

To understand TRPO and PPO we first need to define, the **surrogate objectif** $r_t(\theta)$:

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \tag{1.27}$$

Given a state s_t and an action a_t , $r_t(\theta)$ will be greater than 1 if the probability of selecting a_t is higher for the current policy than it is for the old policy. It will be between 0 and 1 when a_t is less probable for our current policy.

The TRPO objective is the following :

$$\begin{aligned} \underset{\theta}{\operatorname{maximize}} \quad & \hat{\mathbb{E}}_t\left[r_t(\theta)\hat{A}_t\right] \\ \text{subject to} \quad & \hat{\mathbb{E}}_t\left[\mathrm{KL}\left[\pi_{\theta_{\text{old}}} \cdot \mid s_t), \pi_\theta(\cdot \mid s_t)\right]\right] \leq \delta \end{aligned} \tag{1.28}$$

The probability of selecting the action a_t in the state s_t is changed in proportion to the sign and value of the advantage. However, this change is controlled by a KL-divergence constraint. We search for a policy in a neighbourhood of radius δ around the initial policy (in the sense of the Kullback-Leibler divergence).

However, optimization under constraint is costly and requires that the implementation uses exact Hessian-vector product (Conjugate gradient) or finite differences approximation.

One way around the problem is to use a penalty instead of a constraint to get back to an unconstrained optimisation problem :

$$\underset{\theta}{\operatorname{maximize}}\ \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}\hat{A}_t - \beta \mathrm{KL}\left[\pi_{\theta_{\text{old}}} \cdot \cdot \mid s_t), \pi_\theta(\cdot \mid s_t)\right]\right] \tag{1.29}$$

However, this formulation is problematic because the β parameter is very hard to choose and depends strongly on the problem.

The idea of PPO is to get rid of this KL divergence constraint.

We need define the most important part of the PPO loss, the **clipped surrogate objectif** $L^{CLIP}(\theta)$:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, \operatorname{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right] \tag{1.30}$$

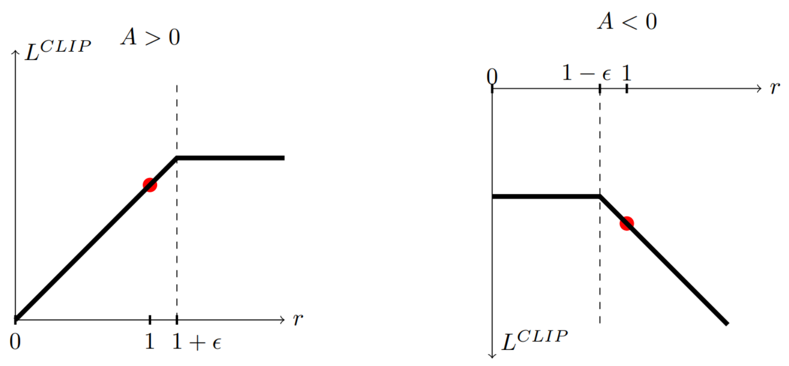


Figure 1.3: Plots showing one term (i.e., a single time step) of the surrogate function ^{CLIP} as a function of the probability ratio r, for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimization, i.e. , $r = 1$. Note that ^{CLIP} sums many of these terms

- When the advantage is positive : then the selected action had a better-than-expected effect on the outcome. On the right, the loss function flattens out when r gets too high i.e. when an action is a lot more likely under current policy than it was under the old policy. We do not want to overdo the action update by taking a step too far, so we clip the objective to prevent this as well as blocking the gradient with a flat line.
- When the advantage is negative : the loss function would flatten out when r goes near zero, meaning a particular action is much less likely on current policy.

Two other terms are added to the clipped surrogate objective :

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S\left[\pi_\theta\right](s_t) \right] \tag{1.31}$$

The critic regression term :

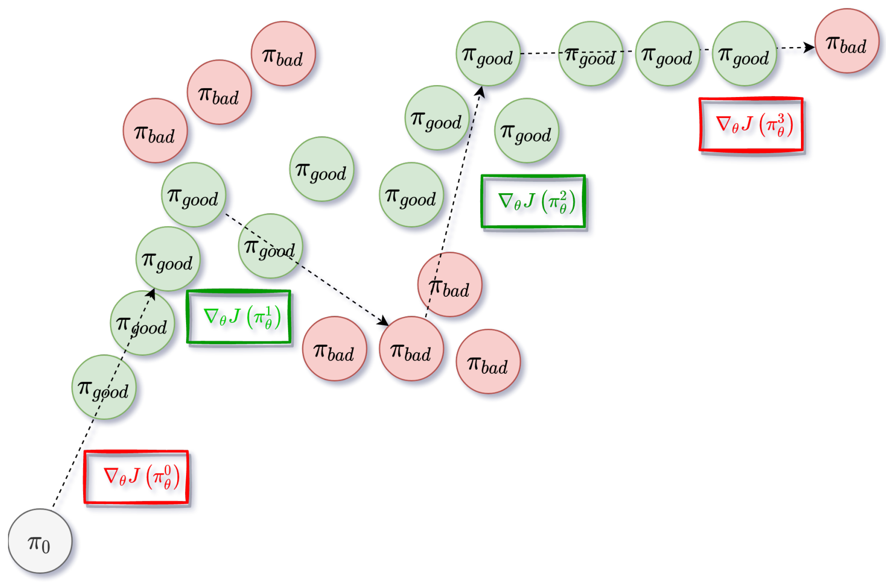
$$L_t^{VF} = \left(V^\pi\left(s_t\right) - G_t\right)^2 \tag{1.32}$$

This term is to update the critic network. We recall that the role of the critical network is to estimate G_t. A good estimation of G_t allows to have a good estimation of the advantage function and consequently a better optimization.

The entropy term :

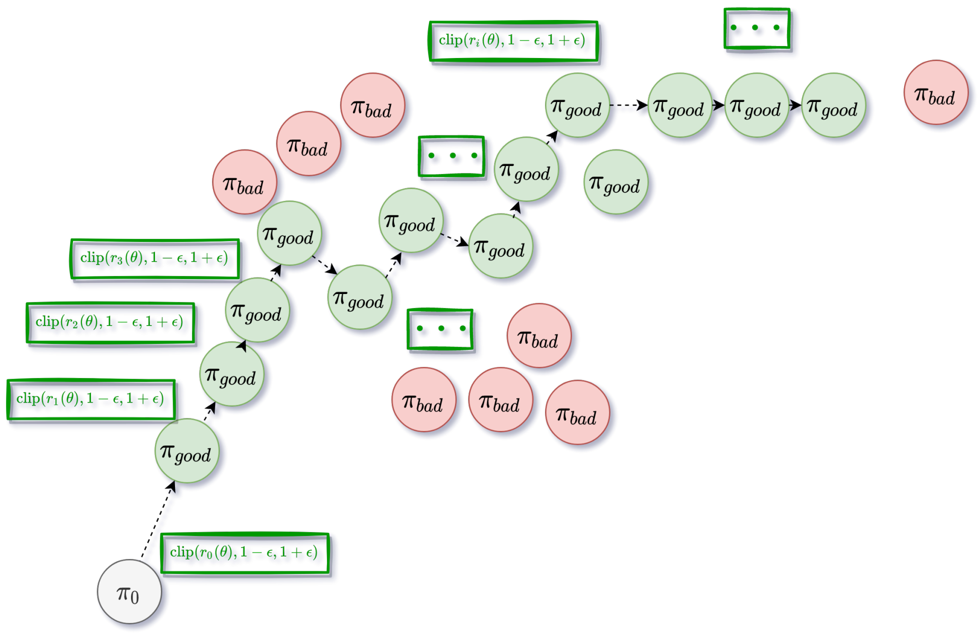
$$S\left[\pi_\theta\right](s_t) = \mathcal{H}(\pi_\theta(s_t)) = -\sum_{a \in \mathcal{A}} \pi_\theta^a(s_t) \log \pi_\theta^a(s_t) \tag{1.33}$$

with $\mathcal{H}$ an entropy operator (Shannon entropy). This term is the entropy term on the actor's policy. By maximising this term, the agent is encouraged to take innovative actions. This term allows the exploration of new actions and consequently allows to get out of a local optima.



π_{bad}
π_{good}
$\nabla_\theta J(\pi_\theta^0)$
$\nabla_\theta J(\pi_\theta^1)$
$\nabla_\theta J(\pi_\theta^2)$
$\nabla_\theta J(\pi_\theta^3)$
π_0

Figure 1.4: Scheme for forming an intuition : space of policies explored by optimising a classical policy gradient criterion



π_0
$\text{clip}(r_0(\theta), 1-\epsilon, 1+\epsilon)$
$\text{clip}(r_1(\theta), 1-\epsilon, 1+\epsilon)$
$\text{clip}(r_2(\theta), 1-\epsilon, 1+\epsilon)$
$\text{clip}(r_3(\theta), 1-\epsilon, 1+\epsilon)$
$\text{clip}(r_i(\theta), 1-\epsilon, 1+\epsilon)$
π_{good}
π_{bad}
$\cdots$

Figure 1.5: Scheme for forming an intuition : space of policies explored by optimising a PPO or TRPO criterion

Generalized Advantage Estimation

Previously, it was seen that the classical actor-critic architectures use a certain advantage function to estimate the direction of the gradient. Several of these advantage functions have been presented. One can generalise the writing of a large part of these advantage functions in the following way :

$$\begin{aligned}
\hat{A}_t^{(1)} &= r_t + \gamma V\left(s_{t+1}\right) - V\left(s_t\right) \\
\hat{A}_t^{(2)} &= r_t + \gamma r_{t+1} + \gamma^2 V\left(s_{t+2}\right) - V\left(s_t\right) \\
\cdots &= \cdots \\
\hat{A}_t^{(\infty)} &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots - V\left(s_t\right)
\end{aligned} \tag{1.34}$$

The trade-off here is that the estimators $A_t^{(k)}$ with small k have low variance but high bias, whereas those with large k have low bias but high variance. A simple intuition behind this is that with a small k, we have fewer terms to sum over (which means low variance). However, the bias is relatively large because it does not make use of extra "exact" information with r_K for $K > k$. Moreover $V(s_t)$ is constant among the estimator class, so it does not affect the relative bias or variance among the estimators: differences arise entirely due to the k-step returns.

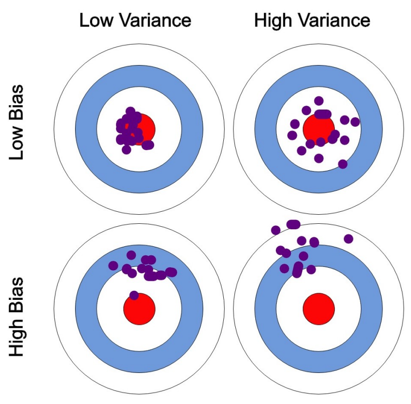


Figure 1.6: Bias-variance illustration : a perfect estimator should have a minimal variance and a minimal bias

The idea of GAE [4] is to use all these estimators to find a bias-variance compromise. The GAE estimator is therefore of the form :

$$\hat{A}_t^{GAE(\gamma,\lambda)} = \Gamma(A_t^{(1)}, A_t^{(2)}, \ldots, A_t^{(k)})$$

More concretely, the estimator is defined as follows:

$$\begin{aligned}
\hat{A}_t^{GAE(\gamma,\lambda)} &= (1-\lambda)\left(\hat{A}_t^{(1)} + \lambda \hat{A}_t^{(2)} + \lambda^2 \hat{A}_t^{(3)} + \cdots\right) \\
&= (1-\lambda)\left(\delta_t^V + \lambda\left(\delta_t^V + \gamma\delta_{t+1}^V\right) + \lambda^2\left(\delta_t^V + \gamma\delta_{t+1}^V + \gamma^2\delta_{t+2}^V\right) + \cdots\right) \\
&= (1-\lambda)\left(\delta_t^V\left(1+\lambda+\lambda^2+\cdots\right) + \gamma\delta_{t+1}^V\left(\lambda+\lambda^2+\cdots\right) + \cdots\right) \\
&= (1-\lambda)\left(\delta_t^V \frac{1}{1-\lambda} + \gamma\delta_{t+1}^V \frac{\lambda}{1-\lambda} + \cdots\right) \\
&= \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V
\end{aligned} \tag{1.35}$$

with $\delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t)$ the **temporal difference residual** presented in the actor-critic section and $(\lambda,\gamma) \in [0,1]^2$. Therefore, GAE can be seen as an exponentially-decayed sum of residual terms.

- When $\gamma = 0$ the GAE $\hat{A}_t^{GAE(\gamma,\lambda)}$ estimator is equal to the TD residual estimator δ_t^V, **(high bias, low variance)**.
- When $\gamma = 1$ the GAE $\hat{A}_t^{GAE(\gamma,\lambda)}$ estimator is equivalent to choose $\hat{A}_t^{(\infty)}$ as advantage estimator, **(low bias, high variance)**.

Example : David Silver Maze

The aim is to find an optimal strategy to reach the goal from the start by reinforcement (successive interactions with the environment).

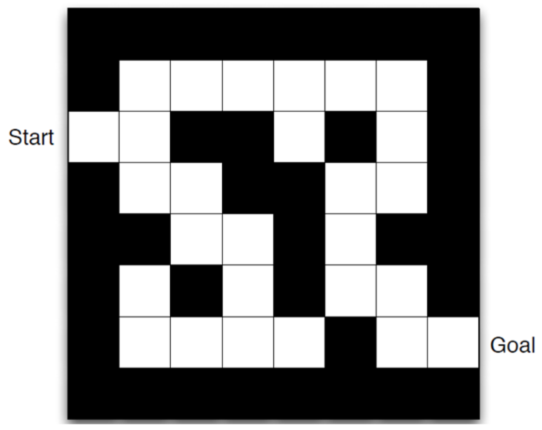


Figure 1.7: Maze example

The environment is as follows :

- The current state s_t is the agent position
- The possible actions are : North, East, South, West.
- The reward is -1 at each step.

One way of solving the problem is to learn an optimal V^* function which associates a value $V^*(s_t)$ to each state s_t. Or, similarly, a function Q^* which associates a value to each state-action pair (s_t, a_t). These algorithms consist in finding a value for each state are called **value-based** algorithms (Q-Learning, TD learning ...)

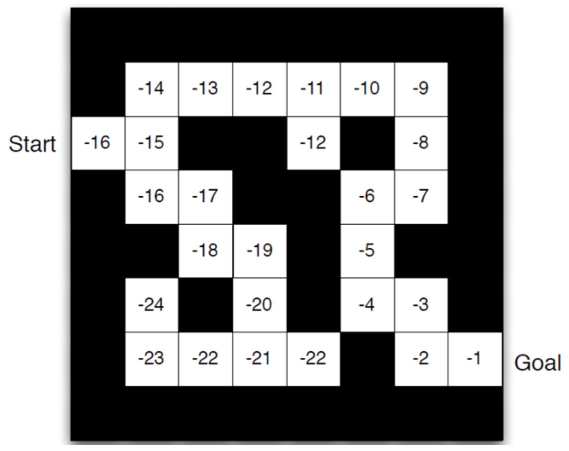


Figure 1.8: Optimal value function for the maze problem. This function associates to each state its distance to the objective point.

Another way of solving this problem is to learn an optimal π^* function which associates a probability distribution on the action space $\pi^*(.|s_t)$ with each state s_t. These algorithms, which consist in finding a probability distribution on the possible actions for each state, are called "policy-based" algorithms (Vanilla policy gradient, TRPO, PPO).

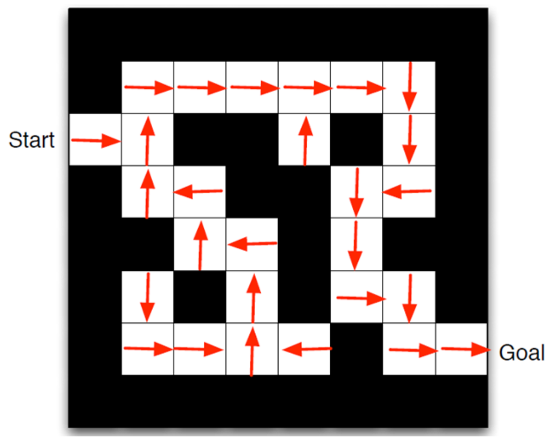


Figure 1.9: Optimal policy function for the maze problem. This function associates to each state the optimal probability distribution (Dirac on the best action to take according to the current state.)