Overview of Reinforcement Learning

1. Markov Decision Processes

1.1. MDPs

- The definition of Markov Decision processes
- The Value functions:
    - Value function of a policy V^{π}
    - State-action Value function of a policy Q^{π}
    - The advantage function A^{π}
- The Bellman equation

1.2. Dynamic Programming

- Policy evaluation
- The two main dynamic programming algorithms:
    - Policy Iteration
    - Value iteration

2. Tabular Methods

2.1. Model Free Prediction

- Monte-Carlo policy evaluation
- Temporal Difference Learning (TD)
- Eligibility traces

2.2. Model Free Control

- On policy vs off-policy
- Generalized Policy Iteration
- Exploration Policies
    - ϵ-Greedy
    - Boltzmann
- Importance sampling
- Monte-Carlo control
- SARSA and SARSA(λ)
- Q-Learning

3. Approximate Value-based methods

3.1. Function Approximation

- Continuous MDPs and the curse of dimensionality
- Function approximation
- The Mean Squared Value Error
- Features:
    - Polynomial
    - Fourier Features
    - Gaussian Radial Basis Function
    - Coarse coding and Tile coding
- Gradient Monte Carlo
- Semi-Gradient methods
- The deadly triad
    - Off-policy
    - Bootstrapping
    - Function approximation
- Batch RL approaches (LSPI, Fitted Q-iterations)

3.2. Deep Reinforcement Learning

- The issues with Deep neural networks
    - The deadly triad
    - the importance of online learning
    - Catastrophic forgetting
- The DQN solutions:
    - Target networks
    - Replay Buffers
    - Minibatch Updates
    - Reward and target clipping
- The DQN variants:
    - Double DQN
    - Prioritized Replay Buffer
    - Noisy DQN
    - Distributional DQN
    - Rainbow

4. Policy gradient and Actor-critic

4.1. Policy Gradient

- Critic-only, Actor-only and Actor-Critic
- Parametric policies
- The policy gradient with the likelihood ratio trick
    - REINFORCE
    - GPOMDP
- The Fisher information matrix and the natural gradient
- The Policy Gradient Theorem
- Compatible function approximation
- The Episodic Natural Actor-Critic algorithm (eNAC)

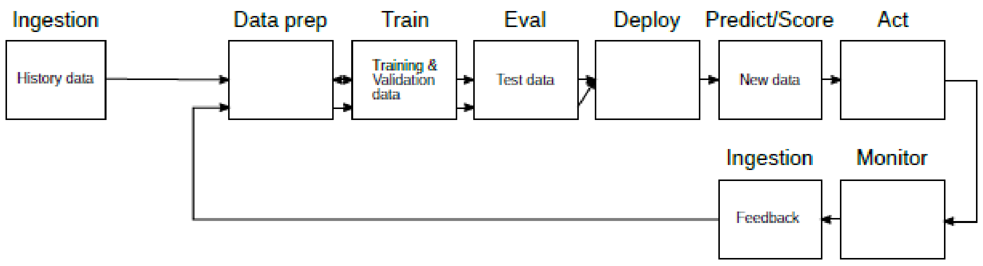
4.2. Actor-Critic

- Entropy and Relative Entropy
- The issues of the PGT
- The surrogate losses
- the Advantage Actor-Critic algorithm (A2C)
- Trust region optimization
- Monte-Carlo advantage estimation
- Generalized Advantage Estimation (GAE)
- The Trust Region Policy Optimization algorithm (TRPO)
    - The conjugate gradient
    - Efficient Fisher-vector product
- The Proximal Policy Optimization algorithm (PPO)
    - the clipped loss

Introduction to Reinforcement Learning

1. What is Reinforcement Learning?

According to E.Brunskill “*The fundamental challenge in artificial intelligence and machine learning is learning to make good decisions under uncertainty”.* Reinforcement Learning (RL) is agent-oriented learning: learning by interacting with an environment to achieve a goal. Every AI problem can be phrased this way, or all data science work loops as illustrated in figure below are in fact reinforcement learning.



An agent has repeated interactions with the world, gets rewards for sequences of decisions and does not know in advance how the world works. It learns by trial and error, with only delayed evaluative feedback (reward). This kind of machine learning is like natural learning
Which can tell for itself when it is right or wrong.

The typical setting

v

2. What’s Special about RL?

3. Introduction to Markov Decision Process

4. Flavors of the RL Problem

5. Wrap-Up

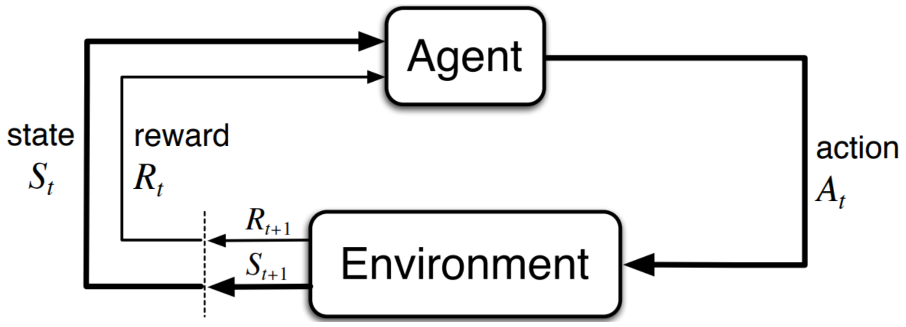
Why RL is crucial for AI and why all other approaches are probably doomed!
Background and characteristics of RL

Classifications of RL problems
Core components of RL algorithms

Markov Decision Processes

1. Markov Decision Processes (MDPs)

Markov Decision Processes formally describes an environment for RL. The environment is fully observable, and the current state completely characterizes the process. Let's explain this in detail using the following Agent-Environment representation:



At each time step for a sequence of discrete time steps $t = 0,1,2, \ldots$; the agent first receives a representation of state s_t, executes an action a_t and then obtains a reward r_t as well as reaching a new state s_{t+1}. The prolonged (sequential) interaction between agent and environment generates a trajectory:

$$s_0, a_0, r_1, s_1, a_1, s_2, \ldots$$

A Markov Decision Process (MDP) can be summarized as a tuple $M =< S, A, R, P, \iota, \gamma >$ where:

- S is the set containing all states
- A is the set containing all action
- R is the reward function
- P is the transition function/dynamics
- ι is the probability distribution over initial states
- $\gamma \in [0,1)$ is the discount factor

There are also different types of MDPs summarized in the following table

Discrete (or finite) MDP	**Continuous MDP**	**Deterministic MDP**	**Stochastic MDP**
- Discrete state space S - Discrete action space A	- Continuous state space S - Discrete or continuous action space A	- Deterministic transition function $P: SxA \to S$ - Reward function is $R: SxA \to \mathbb{R}$	- Stochastic transition function $P: SxAxS \to$ a probability - Reward function is $R: SxAxS \to \mathbb{R}$

1.1. Transition Dynamics of an MDPs

The transition dynamics of an MDP is defined by a probability

$$P(s', r|s, a) = P(S_t = s', R_t = \ r|S_{t-1} = s, A_{t-1} = a)$$

for all $s \in S, a \in A, r \in R$ and $s' \epsilon \ S$. Being a probability the sum over all next states as well as rewards needs to be 1:

$$\sum_{s' \in S} \sum_{r \epsilon R} P(s', r|s, a) = 1 \, , \forall s \epsilon S, a \epsilon A P(s', r|s, a) = P(S_t = s', R_t = \ r|S_{t-1} = s, A_{t-1} = a)$$

Note that for deterministic MDPs an action $a \epsilon A$ executed in a state $s \epsilon S$ leads to a reward $r \epsilon R$ and transition to the next state $s' \epsilon S$ with:

$$P(s', r, |s, a) = 1$$

The **Markov property** states that the transition dynamics only depends on the current state $s \epsilon S$ and the executed action $a \epsilon A$. This property enables to obtain all information about the environment:

- The *state-transition probability function:*

$$P(s', r|s, a) = P(S_t = s', R_t = \ r|S_{t-1} = s, A_{t-1} = a)$$

- *The expected reward for state-action pairs:*

$$R(s, a) = E[R_t|S_{t-1} = s, A_{t-1} = a] = \textstyle\sum_{r \in R} r \sum_{s' \in S} P(s', r|s, a)$$

- *The expected rewards for state-action-next-state triples:*

$$R(s, a, s') = E[R_t|S_{t-1} = s, A_{t-1} = a, S_t = s'] = \frac{\sum_{r \epsilon R} r P(s', r|s, a)}{P(s', r|s, a)}$$

1.2. Episodes

The interaction between an agent and the environment is modelled through episodes. The agent starts an episode in one of the possible initial states sampled according to the initial state distribution ι.

The episode ends when the agent has performed an arbitrary maximum number of steps, known as horizon, or has reached an absorbing state, i.e., a state where all actions lead to itself, and the reward is always 0. Episodes are used to train agents and evaluate their behaviour.

1.3. Returns

The sum of rewards collected after T steps is called return, which is usually, computed over a whole episode:

$$J_t = \sum_{k=0}^{\infty} R_{t+k+1}$$

A discounted return is defined as:

$$J_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

The discount factor $0 < \gamma < 1$ trades off immediate and long-term reward. E.g., $\gamma = 0$ means that the agent only cares about immediate rewards (myopic) and $\gamma = 1$ means that the agent cares equally about all rewards, even the ones collected after infinite steps. For $T \to \infty$ and $R > 0$ the return turns to a geometric series and converges to $\frac{R}{1-\gamma}$. Most MDPs are discounted because

- Its mathematically convenient to discount rewards
- It avoids infinite returns in cyclic MDPs
- the uncertainty about the future may not be fully represented
- animal/human behaviour shows preference for immediate reward

If the reward is for example financial then immediate rewards may earn
more interest than delayed rewards. It is sometimes possible to use undiscounted MDPs (i.e., $\gamma = 1$) e.g., if all sequences terminate.

1.4. Rewards

The (discounted) return has a recursive nature:

$$J_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} = R_{t+1} + \gamma \sum_{k=1}^{\infty} \gamma^{k-1} R_{t+k+1} = R_{t+1} + \gamma J_{t+1}$$

The goal of Reinforcement Learning is obtaining a behaviour that maximizes the discounted return J_t starting from any time step t. Thus, the reward function expresses desired (or undesired) behaviour that should be reinforced (or avoided). The reward function is designed by a human expert according to what they want to obtain. Reward functions where the reward often changes are called dense, e.g., the distance from a goal state. Reward functions where the reward is almost constant are called sparse, e.g., they are always 0 except for 1 upon reaching a goal state.

The question that arises now is if a scalar reward is an adequate notion of a purpose?
According to the Sutton hypothesis, all of what we mean by goals and purposes can be well thought of as the maximization of the cumulative sum of a received scalar signal (reward). But this is still an open problem, in which its current solution is so simple and flexible, that we must disprove it before considering anything different.
A goal should specify what we want to achieve, not how we want to achieve it. The same goal can be specified by (infinite!) different reward functions. A goal must also be outside the agent's direct control and thus outside the agent who must be able to measure success explicitly and frequently during its lifespan.

2. Policies and value functions

What do we mean by behaviour of an agent? Agents interact with the environment by executing actions at each state. A policy is a probability distribution that, given a state $s \in S$, it computes the probability of executing any action $a \epsilon A$ from that state. Policies are commonly denoted as π, e.g., $\pi(a|s)$ is the probability of executing action a in state s. Reinforcement Learning aims at obtaining the policy π that maximizes the return J_t obtained from any state $s \in S$. The policy that maximizes the return from every state is called optimal policy and is often denoted as π^*.

2.1. Value Functions for non-optimal policies

The *value function* $V^\pi(s)$ of a state s under a policy π is the expected discounted return when starting in s and following π thereafter (How good is it to be in a state s?):

$$V^\pi(s) = E_\pi[J_t|S_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,|S_t = s]$$

Note that since the reward r is lower bounded by $R_{min}$and upper bounded by R_{max}we have the following value function bounds:

$$V^\pi(s) \,\epsilon \left[\frac{R_{min}}{1-\gamma}, \frac{R_{max}}{1-\gamma}\right]$$

The *action-value function* $Q^\pi(s,a)$ of taking an action a in a state s under a policy π is the expected discounted return when starting in s, executing action a, and following π thereafter (How good is it to take action a in state s?):

$$Q^\pi(s,a) = E_\pi[J_t|S_t = s, A_t = a] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,|S_t = s, A_t = a]$$

In some contexts, it is beneficial to focus on the relative performance of an action w.r.t. the policy behaviour. For that we can use the *advantage function* $A^\pi(s,a)$ which is the expected advantage in terms of return in state s, taking action a, w.r.t. the policy π (How good is an action a compared to the policy's behaviour?).

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$$

The *relationships* between action-value function and value function when an action is sampled from the (not optimal) policy:

$$V^\pi(s) = E_{a\sim\pi}[Q^\pi(s,a)]$$

The *expected advantage* when a is sampled from (not optimal) policy π is 0:

$$E_{a\sim\pi}[A^\pi(s,a)] = E_{a\sim\pi}[Q^\pi(s,a)] - V^\pi(s)$$

2.2. Value Functions for optimal policies

The value functions induced by an optimal policy π^* are called optimal value functions.

- Optimal Value function: $V^*(s) = \max_\pi V^\pi(s)$
- Optimal Action-Value Function: $Q^*(s,a) = \max_\pi Q^\pi(s,a)$
- Relationship between them: : $V^*(s) = \max_{a\in A} Q^*(s,a)$

- Optimal value functions can be used to compute optimal policies, by selecting:

$$\pi^* = argmax_{a \in \pi} Q^*(s, a), \forall s \in S$$

In the following some important properties of optimal policies:

- Value functions define a partial ordering over policies
- There exists an optimal policy π∗ that is better than or equal to all
- other policies
- All optimal policies induce the optimal value function
- All optimal policies induce the optimal action-value function
- There is always a deterministic optimal policy for any MDP, which can be found by maximizing over

3. Bellman equations

4. Wrap-up

Dynamic Programming

5. Intro

According to Sutton and Barto "Dynamic programming refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as an MDP". The word **dynamic** translates to a sequential or temporal component to the problem meanwhile **programming** is about optimizing a "program", i.e., a policy. The general idea is to break the overall problem down into smaller sub-problems which we then solve optimally. By combining optimal sub-solutions, we get an optimal global solution, if *Bellman's principle of optimality* applies and that the decomposition is possible.

5.1. Requirements

Dynamic Programming is a class of algorithmic solutions suitable for problems which have two properties (or requirements) as already mentioned above:

- *Optimal substructure*: The Principle of optimality applies, and the optimal solution can be decomposed into sub-problems.

- *Overlapping sub-problems:* The Sub-problems recur many times; and Solutions can be cached and reused.

MDPs satisfy both properties with the **Bellman equation** which enables a recursive decomposition and the **value function** which stores and reuses solutions.

5.2. Planning

Dynamic programming assumes full knowledge of the MDPs transition function P and reward function R. It is used for planning in an MDP. Note that we have two central problems we can solve in an MDP: *prediction* and *control.* In prediction, we just want to predict the MDPs behaviour, i.e., measure its value function given a policy. In control, we want to find the optimal value function and policy. As these problems are closely related and we need the former for the latter, we will discuss them jointly. So, to summarize:

- **Prediction** (policy evaluation)
  - Input: MDP $M =< S, A, R, P, \iota, \gamma >$ and policy π
  - Output: value function V^{π}

- **Control** (policy improvement)
  - Input: MDP $M =< S, A, R, P, \iota, \gamma >$
  - Output: optimal value function V^{*} and optimal policy π^{*}

5.3. Finite-Horizon

For finite-horizon MDPs, DP is straightforward by starting from $k = T - 1$ and iterating backwards, $k = T - 1, T, \ldots, 0$, reusing the sub-solutions. The value function and policy are then k-dependent and we get the optimal value function and policy for $k = 0$ (when the information was able to "flow" through the MDP). Compared to brute-force policy search, we get an exponential speedup! As DP for finite-horizon problems is straightforward, we will now stick to infinite-horizon problems.

6. Policy Iteration

Policy iteration (PI) is an algorithm for solving infinite-horizon MDPs by repeating the following steps:

1. Policy Evaluation: estimate V^{π}
2. Policy Improvement: find a $\pi' \geq \pi$

The following sections describe these steps in detail and discuss convergence.

6.1. Policy Evaluation

For a given policy π compute

7. Wrap-Up

8. Questions

8.1. Policy Iteration

Suboptimal policy:

$$V^\pi(s) = E_\pi[Q^\pi(s,a)|a] = \sum_a \pi(a|s) * Q^\pi(s,a)$$

$$Q^\pi(s,a) = r(s,a) + \gamma E_{s' \sim p(\cdot|s,a)}[V^\pi(s')] = \mathrm{r}(s,a) + \gamma \sum_{s'} p(s'|s,a) V^\pi(s')$$

Optimal policy

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

$$Q^*(s,a) = r(s,a) + \gamma E_{s' \sim p(\cdot|s,a)}[V^*(s')] = r(s,a) + \gamma \sum_{s'} p(s'|s,a) V^*(s')$$

Policy Improvement

In policy improvement*,* we want to use find a better policy $\pi' \geq \pi$ using Q^π. We do this by acting greedily, i.e.:

$$\pi'(s) = \ argmax_{a \epsilon A} Q^\pi(s,a).$$

With this definition, we have the following inequality which improves the value from any state s over one step:

$$Q^\pi\big(s, \pi'(s)\big) = \max_{a \in A} Q^\pi(s,a) \geq Q^\pi(s, \pi(s)) \ = \ V^\pi(s).$$

Theorem 1 (Policy Improvement Theorem). Let π and π' be policies with $Q^\pi\big(s, \pi'(s)\big) \geq V^\pi(s)$. Then $\pi' \geq \pi$ and therefore $V^{\pi'}(s) \geq \ V^\pi(s)$.

Proof. By repeatedly applying the premise $(*)$, the Bellman expectation equation (†), and the definition of the value function (‡), we find the following inequality chain:

$$V^\pi(s) \leq Q^\pi\big(s, \pi'(s)\big) = E_{\pi',P}[r_{t+1} + \gamma V^\pi(s_{t+1})|s_t = s]$$

$$\leq E_{\pi',P}\big[r_{t+1} \ + \ \gamma Q^\pi\big(s_{t+1}, \pi'(s_{t+1})\big)\big|s_t = s\big]$$

$$\leq \ E_{\pi',P}[r_{t+1} \ + \ \gamma r_{t+2} + \ \gamma^2 Q^\pi(s_{t+2}, \pi'(s_{t+2}))|s_t = s]$$

$$\leq \ E_{\pi',P}[r_{t+1} \ + \ \gamma r_{t+2} + \cdots |s_t = s] = V^{\pi'}(s)$$

Model-Free Policy Eval./Prediction

1. Recap

In Dynamical Programming as we know there are two classes of algorithms:

a) Policy Iteration

   i. **Iterative Policy Evaluation:** Uses Bellman Equation to compute the Value Function under the evaluated policy π
   ii. **Policy Improvement:** Checks if the greedy policy (the one maximizing the Q-function) is better than the current policy π, and if yes it updates it.

b) Value Iteration:

   i. **One single iteration:** Performs one policy evaluation followed by one policy improvement. It uses Bellman Optimality Equation and doesn’t extract intermediate policies. But intermediate value functions per iteration may not represent anything meaningful.

Dynamical Programming is about Model-based prediction and control. Planning occurs inside **known** MDPs.

2. Monte-Carlo Algorithms

In this chapter we consider our first learning methods for estimating value functions and discovering optimal policies. Unlike the previous chapter, here we do not assume complete knowledge of the environment. Monte Carlo methods require only **experience** - sample sequences of states, actions, and rewards from actual or simulated interaction with an environment.

Although a model is required, the model need only generate sample transitions, not the complete probability distributions of all possible transitions that is required for dynamic programming (DP).

Monte Carlo methods are **Model-free Prediction** and **Model-free Control** methods. The overall goal is to estimate value functions and optimizing policies in **unknown** MDPs assuming finite MDP problems (or problems close to that).

In Model-free Prediction the objective is to estimate the value function of an **unknown MRP** (MDP + Policy). This is also called **Policy Evaluation**.

In Model-free Control the objective is to optimize the value function of an **unknown MDP**.

In general, they are a broad class of computational algorithms relying on repeated random sampling to obtain numerical results. MC methods learn directly from episodes of experience. An episode is the recording of actions and states that an agent performed from a start state to an end state.

They are model-free as already explained they have no knowledge of an MDPs **transitions** and **rewards**. Learning occurs from complete episodes with no bootstrapping. MC uses the simplest possible idea that **a value equals the mean return**. According to Caveat one can only apply MC to episodic MDPs in which all episodes must terminate! To summarize:

2.1. Mathematical Background

The goal is to estimate mean for Monte-Carlo methods. Let X be a random variable with $\mu = E[x]$ and Variance $\sigma^2 = Var[X]$. Let $x_i \sim X, i = 1, \dots, n$ be i.i.d. realiztions of X. Then the empirical mean of X is:

$$\hat{\mu}_n = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Consider we have $E[\hat{\mu}_n] = \mu,\ Var[\hat{\mu}_n] = \frac{Var[x]}{n}$

Strong law of large numbers: $\hat{\mu}_n \overset{P}{\rightarrow} \mu$ bzw. $P(\lim_{n\rightarrow\infty} \hat{\mu}_n = \mu) = 1$

Weak law of large numbers: $\hat{\mu}_n \overset{a.s.}{\longrightarrow} \mu$ bzw. $P(|\hat{\mu}_n - \mu| > \in) = 0$

Central limit theorem: $\sqrt{n}\,(\hat{\mu}_n - \mu) \overset{D}{\rightarrow} N(0,\ Var[x])$

If samples are i.i.d. by the law of large numbers, the estimate of the mean converges to the expected value. Each average is itself an unbiased estimate, and the standard deviation of its error falls as $\frac{1}{\sqrt{n}}$ where n is the number of samples averaged.

2.2. Deep Dive Monte Carlo Policy Evaluation

Input: Episodes of experience $\{s_1,\ a_1,\ r_2,\ .\ .\ .\ ,\ s_T\}$ generated by following the policy π in given MDP or generated by MRP.

Output/Goal: Learn $\boldsymbol{V^\pi}$ from experience under policy $\boldsymbol{\pi}$: $s_1, a_1, r_2, \dots, s_T \sim \pi$

Solution:

The return is the cumulative discounted reward.

$$J_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{T-1} r_{t+T}$$

and the value function is the expected return.

$$\boldsymbol{V^\pi(s) = E_\pi[J_t | s_t = s]}.$$

Instead of using the expected return we are using the empirical mean return. Estimate $V^\pi(s)$, the value of a state $\boldsymbol{s}$ under policy $\boldsymbol{\pi}$, given a set of episodes obtained by following $\boldsymbol{\pi}$ and passing through $\boldsymbol{s}$. Each occurrence of state $\boldsymbol{s}$ in an episode is called a visit to $\boldsymbol{s}$. Of course, $\boldsymbol{s}$ may be visited multiple times in the same episode. There are two methods in general:

1. **First-Visit Monte Carlo Policy Evaluation:** Average returns only for the first-time s is visited (unbiased estimator). See algorithm 1.

1. **Every-Visit Monte Carlo Policy Evaluation:** Average returns for every time s is visited (biased but consistent estimator). Extends more naturally to function approximation and eligibility traces. Every-visit MC would be the same except without the check for $\boldsymbol{s_t}$ having occurred earlier in the episode. See algorithm 2.

Algorithm 1: First Visit Monte Carlo Policy Evaluation

```
    Input: policy $\pi$
    Output: approximate $V^{\pi}$
 1: initialize $V^{(k)}$ arbitrarily.
 2: initialize $Returns(s)$ as an empty list for all $s \in S$.
 3: repeat
 4:     $(s_0; \mathrm{r}_1, \mathrm{s}_1; \mathrm{r}_2,\ s_2, \ldots, \mathrm{r}_{\mathrm{T}-1}, \mathrm{s}_{\mathrm{T}-1},\ r_T)$ ← generate episode
 5:     foreach $t = 0,1,\ldots,T$ do
 6:         if $s_t$ is visited for the first time then
 7:           $J_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$        // cummulative discounted reward
 8:           append $J_t$ to $Returns(s)$
 9:           $V(s_t)$ ← average($Returns(s)$)       //uodate value estimate
10: until convergence
11: return $V$
```

Algorithm 2: Every Visit Monte Carlo Policy Evaluation

```
    Input: policy $\pi$
    Output: approximate $V^{\pi}$
 1: initialize $V^{(k)}$ arbitrarily.
 2: initialize $Returns(s)$ as an empty list for all $s \in S$.
 3: repeat
 4:     $(s_0; \mathrm{r}_1, \mathrm{s}_1; \mathrm{r}_2,\ s_2, \ldots, \mathrm{r}_{\mathrm{T}-1}, \mathrm{s}_{\mathrm{T}-1},\ r_T)$ ← generate episode
 5:     foreach $t = 0,1,\ldots,T$ do
 7:         $J_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$          // cummulative discounted reward
 8:         append $J_t$ to $Returns(s)$
 9:         $V(s_t)$ ← average($Returns(s)$)          //update value estimate
10: until convergence
11: return $V$
```

3. Temporal-Difference Learning

We will now turn to methods that feel more like “reinforcement learning” rather than DP and control theory: temporal difference (TD) learning. Like MC methods, TD learning learns directly from experience and is also model-free, i.e., it has no knowledge of the MDP dynamics. However, TD methods can learn from incomplete episodes and therefore **do not require episodic MDPs**, making them readily applicable to all kinds of problems. The core idea is to update an estimate towards a better estimate.
Consider incremental every-visit MC policy evaluation,

$$V(s_t) \leftarrow V(s_t) + \alpha\big(J_t - V(s_t)\big) = (1-\alpha)V(s_t) + \alpha J_t$$

where $J_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k = r_{t+1} + \gamma V(s_{t+1})$ is the return following s_t and α is a trade-off between the two estimates. Note that for $\alpha = \frac{1}{K+1}$ where K is the number of samples collected before this sample, this update reduces to plain every-visit MC. The idea of TD learning is to replace the return J_t which requires all future samples with an estimate of the return using the immediate reward r_{t+1} together with an estimate of the value function V_{t+1}. This process of using an estimate to update another estimate is called bootstrapping. This yields the simplest TD learning algorithm, TD(0), with the following update:

$$V(s_t) \leftarrow V(s_t) + \alpha\big(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\big) = V(s_t) + \alpha\delta_t$$

We call $r_{t+1} + \gamma V(s_{t+1})$ the **TD target** and $\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ the TD error. Again, α is a trade-off between the two estimates where $\alpha \approx 0$ just sticks to the previous estimate and $\alpha \approx 1$ directly jumps to the new (bootstrapped) estimate.

Algorithm 3: TD(0) (One step Prediction)

```
   Input: environment
   Output: value function
 1: repeat
 4:     initialize $s$
 5:     while $s$ is not terminated do
 7:         take action $a \sim \pi(\cdot|s)$, observe reward $r$ and next state $s'$
 8:         $\delta = r + \gamma V(s') - V(s)$        // TD-Error
 9:         $V(s_t) \leftarrow V(s) + \alpha\delta$
10:         $s \leftarrow s'$
10: until convergence
11: return $V$
```

4. TD(λ)

Looking at TD(0) one might ask why we only look *one* step into the future and not, for instance, three steps. In fact, this is possible, and we can consider the n-*step return:*

$$J_t^{(\mathrm{n})} = \mathrm{r}_{\mathrm{t+1}} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V(s_{t+n})$$

with n-step TD learning being:

$$V(s_t) \leftarrow V(s_t) + \alpha \left(J_t^{(\mathrm{n})} - V(s_t) \right)$$

Increasing n has the advantage of reducing the estimate's bias for the cost of increasing its variance (as with a larger n, more random factors come into play). With $n \to \infty$, this update approaches the MC update.

4.1. Forward View

We can average n-step returns over different n to combine information from different timesteps. For example, averaging 2-step and 4-step returns $\tilde{J} = \frac{1}{2} J^{(2)} + \frac{1}{2} J^{(4)}$. But can we come up with a principled way of weighing different n-step returns? The λ-return J_t^λ combines all n-step returns $J_t^{(n)}$ by using a a weight function $(1-\lambda)\lambda^{n-1}, \lambda \in [0,1)$:

$$J_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} J_t^{(n)}$$

The factor $(1-\lambda)$ is necessary for the sum to be convex, i.e., $(1-\lambda)$

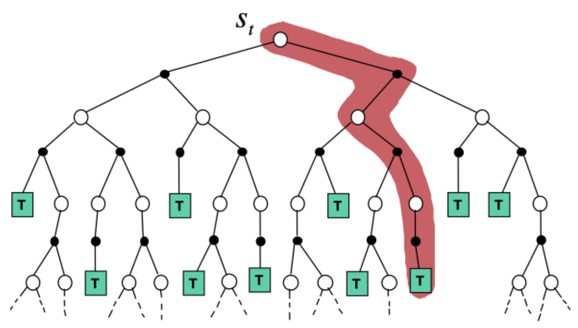
5. Monte Carlo vs. Temporal Differences (vs. Dynamic Programming)

Even though MC and TD are model-free there are still major differences.

Learning

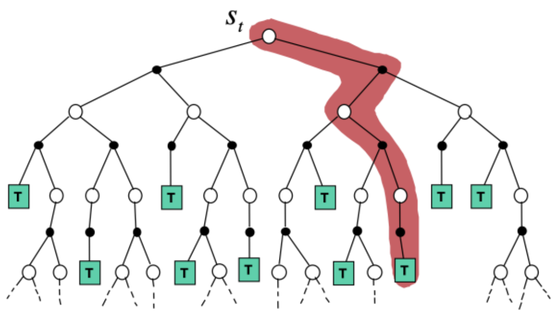
In TD an agent can learn before knowing the final outcome or without the final outcome, because it learns online after every step. This enables learning from incomplete sequences and works in continuing (non-terminating) environments.

$$V(s_t) \leftarrow V(s_t) + \alpha\big(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\big) = V(s_t) + \alpha \delta_{\mathrm{t}}$$



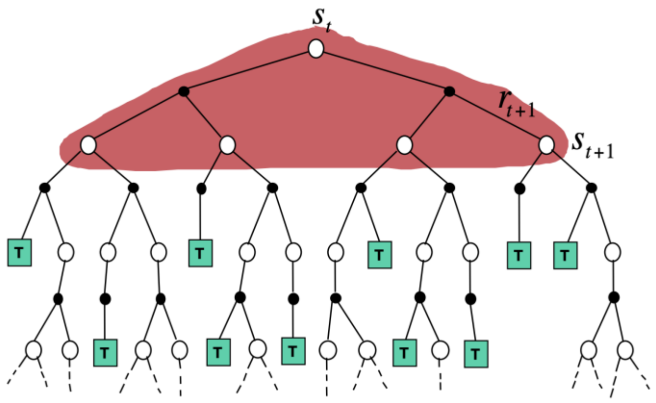
In MC the agent must wait until the end of an episode before the return is known and only learns from complete sequences. Therefore, MC works only for episodic (terminating environments).

$$V(s_t) \leftarrow V(s_t) + \alpha\big(J_t - V(s_t)\big)$$



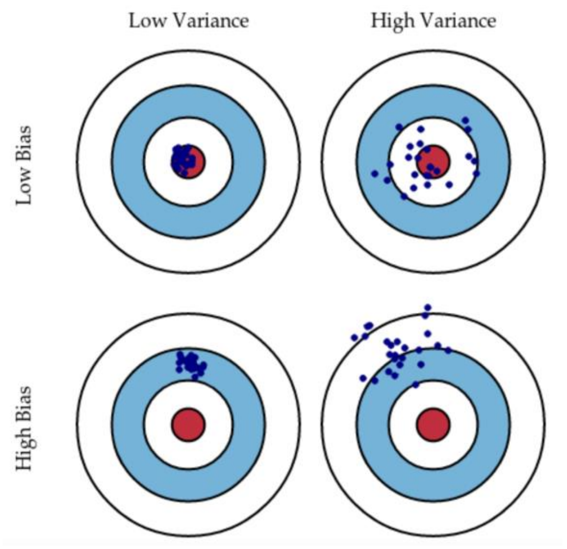
In Dynamic programming the agent considers all possible actions of a state.

$$V(s_t) \leftarrow E_\pi[R_{t+1} + \gamma V(s_{t+1})]$$



Bias–Variance Trade–Off

Error due to bias is the difference between the expected prediction of our model and the actual value we want to predict. Error due to variance is the variability of a model prediction for a given data point. See the following picture:



MC has *high variance* (the return depends on **many** random actions, transitions, rewards) and *zero bias* (J_t is an unbiased estimator of $V^{\pi}(s_t)$). This results in:

- Good convergence properties
- Works well with function approximation.
- Not very sensitive to initial value
- Very simple to understand and use.

TD has *low variance* (TD target depends on **one** random action, transition, reward) and *some bias* (TD target is a biased estimator of $V^{\pi}(s_t)$ unless $V(s_{t+1}) = V^{\pi}(s_t)$).

- Usually more efficient than MC
- TD(0) converges to $V^{\pi}(s_t)$
- Problem with function approximation
- More sensitive to initial values

Markov Property

TD exploits the Markov property and is usually more efficient in Markov environments.
MC does not exploit the Markov property and usually more effective in non-Markov environments.

Bootstrapping and Sampling

Bootstrapping means that the update involves an estimate and sampling on the other hand updates the samples with an expectation.

Method	**Uses Bootstrapping**	**Uses Sampling**
Dynamic Programming	Yes	No
Monte-Carlo	No	Yes
Temporal Differences	Yes	Yes

6. Wrap-Up

We know how to approximate value functions when we don’t have the dynamics (transition function) and the differences between Dynamic Programming vs Monte-Carlo vs. Temporal Difference methods. We also know what the eligibility traces are and how to compute TD(λ).

Additional reading material:
- Book: “Introduction to Reinforcement Learning” (Sutton and Barto, 2018), Chapters 6, 7, 12

7. Questions

How to approximate value functions with unknown dynamics?

What are the differences, advantages, and disadvantages of MC methods compared to DP and TD methods?

What are eligibility traces?

How to compute TD(λ)?

What are the differences, advantages, and disadvantages of TD methods compared to DP and MC methods?

RoLe: What is TD learning? How to derive it?

RoLe: What does on- and off-policy mean?

Model-Free Control

1. Model-free Reinforcement Learning

In Model-free Control the goal is to optimize the value function of an unknown MDP

$$M = < S, A, R, P, \iota, \gamma >$$

which means we don't know P and R. We can only experience them through exploration (experience collected) by acting on the MDP. In the end the goal is as already said to obtain the optimal value function V^* and optimal policy π^*.

Some example problems that can be modelled as MDPs are Robocup Soccer, Protein Folding, Airplane Logistics, Bioreactor, Helicopter, Elevator etc. For most of these problems either the MDP is unknown, but experience can be sampled or the MDP is known but it is too big to use except by samples. Model-free Control can solve these problems.

We will cover methods from *tabular* RL where the state-action-space is small enough such that we can represent the action-value function as a table. We distinguish two categories of methods: on- and off-policy learning.

On-policy learning refers to "learning on the job". In detail learn about policy π from experience sampled from π and evaluate as well as improve the same policy that the agent is already using for action selection.

Off-policy learning refers to "look someone over the shoulder". In detail learn about policy π from experience sampled from a different policy q then the one that is being trained.

2. On-Policy Methods

In this section we discuss tabular on-policy methods.

2.1. On-Policy Monte-Carlo Control

Let’s recap and look back at *generalized policy iteration*, which is the strategy used mostly in tabular methods to obtain an optimal value function and policy. This is done in two steps:

a) Policy Evaluation: estimate V^{π}
b) Policy Improvement: Greedy Policy Improvement -> Generate $\pi' \geq \pi$

In *generalized Policy Iteration*, we do not evaluate the state-value function $V(s_t)$, because a greedy policy improvement over $V(s_t)$ needs the transition dynamics P:

$$\pi'(s) = argmax_{a \epsilon A} R_s^a + P_{ss'}^a V(s')$$

Instead, we are evaluating the action-value function, or we are doing a greedy policy improvement over $Q(s, a)$ which is model-free and does not require an MDP:

$$\pi'(s) = argmax_{a \epsilon A} Q(s, a)$$

However, just using the deterministic policy found during policy improvement for MC policy evaluation means that we do not have exploration (no “new” actions are tried)! This brings us to the *exploration vs. exploitation dilemma.* During decision-making, we have two strategies: *exploit* the current knowledge to make the best-known decision or *explore* and gather more information. In other words, the best long-term strategy (which we want to find) might involve short-term sacrifices. This dilemma is a fundamental problem in RL, and we do not have a satisfying solution yet. Two common approaches are ϵ-*greedy*,

$$a = \{ \begin{matrix} a_t^* \; with \, probability \; 1 - \epsilon \\ random \; action \; with \, probability \; \epsilon \end{matrix}$$

and softmax: A softmax policy biases the action selection towards exploration to find promising actions. Softmax action selection methods **grade action probabilities** by estimated values. The most common softmax uses a Gibbs (or Boltzmann) distribution:

$$\pi(a|s) = \frac{e^{\frac{Q(s,a)}{\tau}}}{e^{\sum_{a' \in A} \frac{Q(s,a')}{\tau}}}$$

$\boldsymbol{\pi(a|s)}$ represents the probability of taking action a given state s using the Softmax policy. We are choosing the action with the highest probability. $\boldsymbol{Q(s, a)}$ is the expected reward (or score) for taking action a in state s. $\boldsymbol{\tau}$ is the *computational* temperature parameter that controls the exploration-exploitation trade-off. Higher values make the policy more exploratory, while lower values make it greedier:

$$\tau \rightarrow \infty : P = \frac{1}{|A|}$$

$$\tau \rightarrow 0 : greedy$$

∈-Greedy Exploration and Policy Improvement

∈-Greedy Exploration is the simplest idea for ensuring continual exploration. All m actions are tried with non-zero probability. Choose the greed action with probability $1-\epsilon$ and a random action with probability ϵ. This can be formulated as a distribution:

$$\pi(a|s) = \{ \begin{matrix} \frac{\epsilon}{m} + 1 - \epsilon \; if \; a^* = argmax_{a\epsilon A} Q(s,a) \\ \frac{\epsilon}{m} \; otherwise \end{matrix}$$

Interestingly, the ϵ-greedy policy still causes monotonic improvements as for any ϵ-greedy policy π, the ϵ-greedy policy w.r.t. $Q\pi$ fulfils the premise of the policy improvement theorem ($V^{\pi'}(s) \geq V^{\pi}(s)$):

$$\boldsymbol{Q^{\pi}(s, \pi'(s))} = \sum_{a\epsilon A} \pi(a|s) * Q^{\pi}(s,a)$$

$$= \sum_{a\epsilon A} Q^{\pi}(s,a) \{ \begin{matrix} \frac{\epsilon}{m} + 1 - \epsilon \; if \; a^* = argmax_{a\epsilon A} Q(s,a) \\ \frac{\epsilon}{m} \; otherwise \end{matrix} \quad (*)$$

$$= \frac{\epsilon}{m} \sum_{a\epsilon A} Q^{\pi}(s,a) + (1-\epsilon) \max_{a\epsilon A} Q^{\pi}(s,a)$$

$$\geq \frac{\epsilon}{m} \sum_{a\epsilon A} Q^{\pi}(s,a) + (1-\epsilon) \sum_{a\epsilon A} \frac{\pi(a|s) - \frac{\epsilon}{m}}{1-\epsilon} Q^{\pi}(s,a) \quad (\dagger)$$

$$= \sum_{a\epsilon A} \pi(a|s) Q^{\pi}(s,a) \quad (\ddagger)$$

$$= \boldsymbol{V^{\pi}(s)}$$

In step $(*)$ we explicitly plugged in the policy π' using that its first case is applied exactly once for the maximum of $Q^{\pi}(s,a)$. Hence, we can pull it out of the sum. In step (†), we "inverse plugged in" an ϵ-greedy policy using the relations:

$$\frac{\pi(a|s) - \frac{\epsilon}{m}}{1-\epsilon} = \frac{\frac{\epsilon}{m} - \frac{\epsilon}{m}}{1-\epsilon} = 0 \qquad \frac{\pi(a|s) - \frac{\epsilon}{m}}{1-\epsilon} = \frac{\frac{\epsilon}{m} + 1 - \epsilon - \epsilon - \frac{\epsilon}{m}}{1-\epsilon} = 1$$

For the "maximum" a and all others, respectively. Note that this maximum is not w.r.t. $Q^{\pi}(s,a)$ and hence the sum over all actions must be less than or equal to the maximum value of $Q^{\pi}(s,a)$. Finally, in step (‡), we used the definition of π and that $Q^{\pi}(s,a)$ is its action-value function. Using ϵ-greedy exploration therefore gives monotonic improvement while not shutting cancelling exploration during policy evaluation!

GLIE Monte-Carlo

Definition 1: GLIE (Greedy in the Limit of Infinite Exploration). A policy π is greedy in the limit of infinite exploration (GLIE) if all state-action pairs are explored infinitely many times:

$$\lim_{k\to\infty} N_k(s,a) = \infty$$

and the policy converges to a greedy policy:

$$\lim_{k\to\infty} \pi_k(a|s) = \mathbf{1}(a = argmax_{a'\in A} Q'_k(s,a'))$$

Here k is the policy iteration index e.g., k-th episode using π: $\{s_1, a_1, r_2, \dots, s_T\} \sim \pi$. The policy is improved based on the new action-value function by $\epsilon \leftarrow \frac{1}{k}$, $\pi \leftarrow \in -greedy(Q)$. We then also have the following theorem.

Theorem 2 (Convergence of GLIE Monte-Carlo). GLIE Monte-Carlo control converges to the optimal action-value function:

$$Q(s,a) \to Q^*(s,a)$$

2.2. On Policy TD-Learning: SARSA

As already discussed Temporal-differece learning has several advantages over Monte-Carlo like lower varincace, online learning, learning with incomplete sequence etc. The Idea now is to use TD instead of MC in our control loop by applying TD to $Q^\pi(s,a)$ using ϵ-greedy policy improvemt to update every time step.

In SARSA (for "state, action,reward, state, action") we use the policy's proposed action during the TD update i.e.:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

Where $a_{t+1} \sim \pi^Q(\cdot\, | s_{t+1})$ is sampled froma policy derived from Q. Hence SARSA is an on-policy algoritm. The pseudocode is decipted in algorithm 4. If we choose the step sizes α wisely, we also have convergens guarantees!

To summarize Policy Iteration then results to:

a) Policy evalaution: SARSA, $Q \approx Q^\pi$
b) Policy Improvement: ϵ-greedy policy improvement

Theorem 3 (Convergence of SARSA). SARSA converges to the optimal action-value function $Q(s,a) \to Q^*(s,a)$ under the following conditions that the policy $\pi_t(s,a)$ constitues a GLIE sequence and constitues a Robbins-Monro sequence of step-sizes α_t: $\sum_{t=1}^{\infty} \alpha_t = \infty, \sum_{t=1}^{\infty} \alpha_t^2 < \infty$.

Algorithm 4: SARSA Algorithm for On-Policy Control

```
   Input: environment
   Output: optimal action-value function
1: initialize $Q(s,a)$ arbitrarily. Except that $Q(terminal,\cdot) = 0$
```

```
 2: repeat for each episode
 3:     initialize $s$
 4:     choose $a \sim \pi^Q(\cdot\,|s)$
 5:     while s is not terminal do
 6:         take action $a$, observe reward $r$ and next state $s'$
 7:         choose $a \sim \pi^Q(\cdot\,|s')$
 8:         $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$ // TD-Error
 9:         $Q(s, a) \leftarrow Q(s, a) + \alpha\delta$
10:         $s \leftarrow s'$
11:         $a \leftarrow a'$
12: until converged
13: return $Q$
```

3. Off-Policy Methods

4. Questions

What is the difference between on-policy and off-policy learning?

How model-free control is related to the Generalized Policy iteration?

What are the sufficient conditions for an effective exploration strategy?

How can we use ϵ-greedy for exploration?

What is Sarsa and how we can do on-policy control?

How can you incorporate λ-returns in TD-control?

How you can do off-policy learning with importance sampling

How to do off-policy control with Q-learning without the need for importance sampling?

What is the relationship of the Bellman equations and the TD targets?

RoLe: What is the difference between Q-learning and SARSA?

RoLe: When do value function methods work well?

Sutton (2.1): In ε-greedy action selection, for the case of two actions and $\varepsilon = 0.5$, what is the probability that the greedy action is selected?

Sutton (6.11): Why is Q-learning considered an off-policy control method?

Sutton (6.12): Suppose action selection is greedy. Is Q-learning then exactly the same algorithm as SARSA? Will they make exactly the same action selections and weight updates?

Approximate Value-based methods

1. Continuous MDPs

Not every RL problem can be formalized as a discrete (and finite) MDP where we could represent the value functions as tables. In continuous MDPs the state -action-space $\boldsymbol{S}, \boldsymbol{A}$ is not discrete (and finite) ($S = \{1, \ldots, N_s\} \subseteq \mathbb{N}, A = \{1, \ldots, N_a\} \subseteq \mathbb{N}$) but continuous (and infinite) ($S \subseteq \mathbb{R}^d, A \subseteq \mathbb{R}^m$). To handle these kinds of environments, we need to use *function approximation* for the value functions (and policies). In this chapter we will also focus only on discrete action spaces and future lectures we will then go to continuous action spaces.

2. Approximated On-Policy Methods

The value function cannot be computed as a table because we have many states (and actions) to store in memory. This results in slow updates for each state. Hence, we use function approximations. The standard choice are parametric value function estimators where the parameters are expressed as the weight vector $w \in \mathbb{R}^d$:

$$\hat{V}_w(s) \approx V^\pi(s), \hat{Q}_w(s) \approx Q^\pi(s.a)$$

Mostly Linear functions, Neural Networks, Regression Trees and gaussian processes are used. The parameters w are much fewer than the states (otherwise the approximation would not make sense). Changing weight affects the accuracy of the estimate of multiple states. Improving the accuracy of the value-function estimate of one state. May decrease the accuracy of the estimate of others. The accuracy can be measured with the *Mean Squared Value Error* (MSVE):

$$\overline{VE}(s; \boldsymbol{w}) \triangleq \sum_{s \in S} \mu(s) \left[V^\pi(s) - \hat{V}_w(s)\right]^2$$

With the state distribution $\mu(s) \geq 0, \sum_s \mu(s) = 1$ which weights the importance of the estimate error for each state s. For on-policy algorithms $\mu(s)$ is the fraction of time spent in state s while following policy π,i.e.:

$$\mu(s) = \frac{1}{T+1} \sum_{t=0}^{T} \pi(s_t = s)$$

2.1. Value function estimation with stochastic gradient descent

Assume the exact value-function $V^\pi(s)$ is known for $\forall s \in S$. The goal is to find an approximation with a differentiable estimator $\hat{V}_w(s)$ with the weight vector $\boldsymbol{w_t} \triangleq \left(w_{1_t}, w_{2_t}, \cdots, w_{d_t}\right)^T$ by stochastic gradient descent (SGD). For each time step, we can then locally update the weights using the update rule:

$$w_{t+1} \leftarrow w_t - \frac{1}{2} \alpha \nabla \left[V^\pi(s_t) - \hat{V}_{w_t}(s_t)\right]^2 = w_t + \alpha \left[V^\pi(s_t) - \hat{V}_{w_t}(s_t)\right] \nabla \hat{V}_{w_t}(s_t)$$

where $\alpha > 0$ is the learning rate and $\nabla \hat{V}_{w_t}(s_t) \triangleq \left(\frac{\partial \hat{V}_{w_t}(s_t)}{\partial \mathrm{w}_1}, \frac{\partial \hat{V}_{w_t}(s_t)}{\partial \mathrm{w}_1}, \dots, \frac{\partial \hat{V}_{w_t}(s_t)}{\partial \mathrm{w}_\mathrm{d}}\right), w \in \mathbb{R}^d$. Note that we update the value function for each step in the trajectory, hence we use a "time" index here. For a proper decay of the learning rate, this is guaranteed to converge to a local optimum. However, we usually do not have access to the real value function-this is why we ultimately need learning!

2.2. Value function estimation with Gradient Monte-Carlo

In most cases the exact value-function $V^\pi(s)$ is not known. Hence, we replace $V^\pi(s)$ with an arbitrary value estimate U_t:

$$w_{t+1} \leftarrow w_t + \alpha\big[U_t - \hat{V}_{w_t}(s_t)\big]\, \nabla \hat{V}_{w_t}(s_t)$$

If U_t is an unbiased estimate of $V^\pi(s_t)$: $\mathrm{E}[\mathrm{U_t}] = \mathrm{V}^\pi(\mathrm{s_t}), \forall \mathrm{t}$, then convergence is guaranteed with properly decaying α. One suitable (unbiased) estimate is the **Monte Carlo Target** $U_t = J_t = \sum_{i=t}^{T} \gamma^{i-t} r_{i+1}$ for which $E[J_t] = V^\pi(s_t)$ (Note that this expectation holds for a specific t, but $E[J_t] = V^\pi(s)$ does **not** hold, hence every-visit MC prediction is biased but gradient MC is not.) holds by defintion of $V^\pi(s_t)$. This approach is summarized in algorithm 10.

Linear Approximator

Consider a linear function approximator (which is linear in the weights $\boldsymbol{w}$) with the feature vector $\phi(s)$:

$$\hat{V}_w(s) = w^T \phi(s) = \sum_{i=1}^{d} w_i\, \phi_i(s)$$

The gradient of a linear value-function $\hat{V}_w(s) = w^T\phi(s)$ is the feature vector $\phi(s)$: $\nabla \hat{V}_{\boldsymbol{w}}(s_t) = \nabla(w^T\phi(s)) = \phi(s)$. When used in gradient Monte-Carlo, the weight update becomes:

$$w_{t+1} \leftarrow w_t + \alpha\big[U_t - \hat{V}_{w_t}(s_t)\big]\, \phi(s_t)$$

Algorithm 5: Gradient Monte Carlo

```
   Input: policy $\pi$, differentiable approximator $\hat{V}: S \to \mathbb{R}$
   Output: estimated parameters $w$
1: initialize value-function $\boldsymbol{w} \in \mathbb{R}^d$ arbitrarily, e.g. $\boldsymbol{w} = \boldsymbol{0}$
2: repeat
3:     $(s_0; \mathrm{r}_1, \mathrm{s}_1; \mathrm{r}_2,\ s_2, \ldots, \mathrm{r}_{\mathrm{T}-1}, \mathrm{s}_{\mathrm{T}-1},\ r_T) \leftarrow$ generate an episode using $\pi$
4:     foreach $t = 0,1,\dots,T$ do
5:         $J_t \leftarrow \sum_{i=t}^{T} \gamma^{i-t} r_{i+1}$    // compute the Monte-Carlo targe
6:         $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\big[J_t - \hat{V}_{\boldsymbol{w}}(s_t)\big]\, \nabla \hat{V}_{\boldsymbol{w}}(s_t)$    //update the function approximator
7: until convergence
8: return $\boldsymbol{w}$
```

2.3. Value function estimation with Semi-Gradient Methods

MC methods are not the only way for estimating V^{π} and indeed, we could also use DP or bootstrapped TD targets, allowing step-based updates and learning from incomplete episodes. The DP and n-step TD targets are:

$$U_t = E_{a_{t+1},s_{t+1}}\left[r(s_t,a) + \gamma\, \hat{V}_w(s_{t+1})\right] = \sum\nolimits_{a,s',r} \pi(a|s_t)p(s_{t+1},r|s_t,a_t)\left[r + \gamma\, \hat{V}_w(s_{t+1})\right]$$

$$U_t = \sum\nolimits_{i=t}^{t+n-1} \gamma^{i-t} r_{i+1} + \gamma^n\, \hat{V}_w(s_{t+n})$$

Note, however, that the targets themselves depend on w! As we just ignore this dependence, the result algorithms are called *semi-gradient* TD methods. Semi-gradient methods are typically faster to converge than gradient MC methods and enable online learning. However, the semi-gradient is a biased estimate of the true gradient convergence is not always guaranteed. Semi-gradient methods only **use one part** of the gradient of the mean squared value error and they **discard** the gradient coming from the **bootstrapped target**:

$$\frac{1}{2}\nabla\left[r + \gamma\, \hat{V}_w(s') - \hat{V}_w(s\,)\right]^2 = \gamma\,\left[r + \gamma\, \hat{V}_w(s') - \hat{V}_w(s\,)\right]\nabla\hat{V}_w(s') - \left[r + \gamma\, \hat{V}_w(s') - \hat{V}_w(s\,)\right]\nabla\hat{V}_w(s)$$

Linear Approximator

Consider a linear function approximator (which is linear in the weights $\boldsymbol{w}$) with the feature vector $\phi(s)$:

$$\hat{V}_w(s) = w^T\phi(s) = \sum_{i=1}^{d} w_i\, \phi_i(s)$$

The gradient of a linear value-function $\hat{V}_w(s) = w^T\phi(s)$ is the feature vector $\phi(s)$: $\nabla\hat{V}_w(s_t) = \nabla(w^T\phi(s)) = \phi(s)$. When used in semi-gradient TD(0), the weight update becomes:

$$\begin{aligned} w_{t+1} &\leftarrow w_t + \alpha\left[\, r_{t+1} + \gamma w_t^T\phi(s_{t+1}) - w_t^T\phi(s_t)\right]\phi(s_t) \\ &= w_t + \alpha\left[\, r_{t+1}\,\phi(s_t) + \gamma w_t^T\phi(s_{t+1})\phi(s_t) - w_t^T\phi(s_t)\phi(s_t)\,\right] \\ &= w_t + \alpha\left[\, r_{t+1}\,\phi(s_t) + \gamma\phi(s_t)\phi^T(s_{t+1})w_t - \phi(s_t)\phi^T(s_t)w_t\right]\ (*) \\ &= w_t + \alpha\left[\, r_{t+1}\,\phi(s_t) + \phi(s_t)\left(\gamma\phi\,(s_{t+1}) - \phi\,(s_t)\right)^T w_t\right] \\ &= w_t + \alpha\left[\, r_{t+1}\,\phi(s_t) - \phi(s_t)\left(\phi\,(s_t) - \gamma\phi\,(s_{t+1})\right)^T w_t\right] \end{aligned}$$

In $(*)$ we used $v_1^T v_2 v_3 = v_3 v_1^T v_2$, where $v_1^T v_2$ is a scalar, so we can rearrange vectors here!

Algorithm 6: Semi Gradient TD(0) (One step Prediction)

Input: policy π, differentiable approximator $\hat{V}: S \to \mathbb{R}$
Output: estimated parameters w

```
 1: initialize value-function $\boldsymbol{w} \in \mathbb{R}^d$ arbitrarily, e.g. $\boldsymbol{w} = \boldsymbol{0}$
 2: **repeat**
 3:     initialize $s = s_0$
 4:     **while** $s$ is not terminal **do**
 5:         take action $a \sim \pi(\cdot\,|s)$, observe reward $r$ and next state $s'$
 8:         $\delta = r + \gamma \hat{V}_w(s') - \hat{V}_{\boldsymbol{w}}(s)$     // TD-Error
 9:         $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\delta\nabla\hat{V}_{\boldsymbol{w}}(s)$
10:         $s \leftarrow s'$
10: **until** convergence
11: **return** $\boldsymbol{w}$
```

2.4. Value function estimation with Semi-Gradient SARSA

Like semi-gradient TD(0), we can also approximate the action-value function instead of the state-value function using SARSA. The one-step SARSA update is then:

$$w_{t+1} \leftarrow w_t + \alpha\left[U_t - \hat{Q}_{w_t}(s_t, a_t)\right] \nabla\hat{Q}_{w_t}(s_t, a_t)$$

$$= w_t + \alpha\left[r_{t+1} + \gamma\, \hat{Q}_{w_t}(s_{t+1}, a_{t+1}) - \hat{Q}_{w_t}(s_t, a_t)\right]\nabla\hat{Q}_{w_t}(s_t, a_t)$$

Algorithm 7: Semi Gradient SARSA

```
    **Input:** policy $\pi$, differentiable approximator $\hat{Q}: S \to \mathbb{R}$ with parameters $w$
    **Output:** estimated parameters $w$
 1: initialize $\boldsymbol{w} \in \mathbb{R}^d$ arbitrarily, e.g. $\boldsymbol{w} = \boldsymbol{0}$
 2: **repeat**
 3:     initialize $s = s_0$
 4:     choose $a \sim \pi^Q(\cdot\,|s)$
 5:     **while** true **do**
 6:         take action $a \sim \pi(\cdot\,|s)$, observe reward $r$ and next state $s'$
 7:         **if** $s'$ is terminal **then**
 8:             **break**
 9:         choose $a' \sim \pi^Q(\cdot\,|s')$
10:         $\delta = r + \gamma\hat{Q}_w(s', a') - \hat{Q}_{\boldsymbol{w}}(s, a)$     // TD-Error
11:         $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\delta\nabla\hat{Q}_{\boldsymbol{w}}(s, a)$
12:         $s \leftarrow s'$
13:         $a \leftarrow a'$
14: **until** convergence
15: **return** $\boldsymbol{w}$
```

2.5. Feature Construction of linear methods

The State vector contains representative features of the problem such the position and angular velocity for balancing a pendulum, the pixels for playing a videogame or even the stock price for finance applications. Basic features may not be representative enough to capture complex behaviour. Let's look at the pendulum example with the state space $s = (s_1, s_2)$ in which $s_1 \in R$ being the angular position and $s_2 \in R$ being the angular velocity. A feature vector $\phi(s) = (s_1, s_2)^T$

is a poor representation of the problem because **interactions** between the state dimensions are not considered! Representative feature vectors consider all dimensions of the state and their (potentially complex) interactions though. How do we obtain good features now?

Polynominal Features

Polynomial features capture interaction among state dimensions by multiplication:

- 1st-order: $\phi(s) = (1, s_1, s_2, s_1 s_2)^T$
- 2nd -order: $\phi(s) = (1, s_1, s_2, s_1 s_2, s_1^2, s_2^2, s_1^2 s_2, s_1 s_2^2, s_1^2 s_2^2)^T$
- …

The number of features grows **exponentially** with the number of dimensions of the state. Note that the approximation is still linear in the weights.

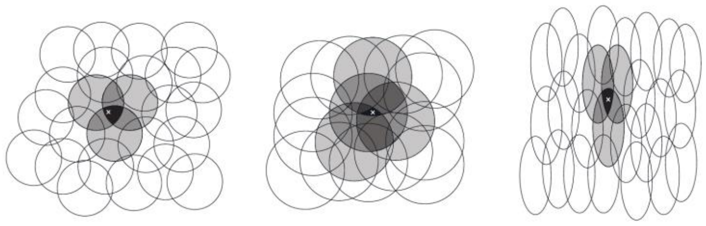
Fourier Basis

The Fourier series expresses periodic functions $f\ (x) = f\ (x + \tau\ )$ as a weighted sum of sine and cosine. It can be used also in RL to approximate aperiodic functions bounded in a certain interval. Use Fourier features with τ set to the length of the interval. Alternatively, it is possible to use only cosine features by considering τ as twice the length of the interval. For example, it is possible to represent any even function (symmetric about the origin) using only cosine features. This means it is possible to approximate any function in the half interval using only cosine features! The one-dimensional n-th order Fourier cosine feature consists of $n + 1$ features:

$$\phi_i(s) \ = \ cos(i\pi s); \ s \in [0, 1], i \ = \ 0, \dots, n$$

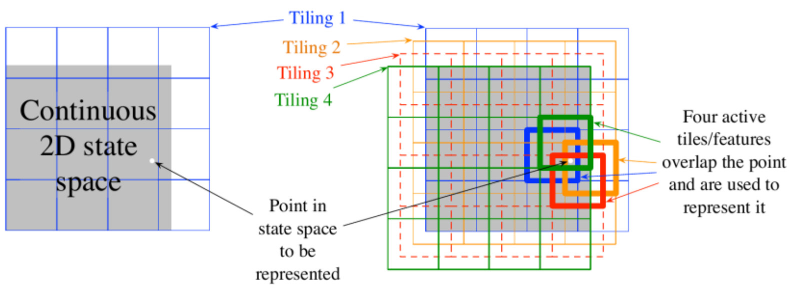
Corse Coding

In Coarse Coding we divide the state space in M different regions. The feature vector then has M binary values. Given a state s, for each region, assign feature value 1 if the state is inside the region, 0 otherwise. The 0-1 features are also called *sparse.*



Tile Coding

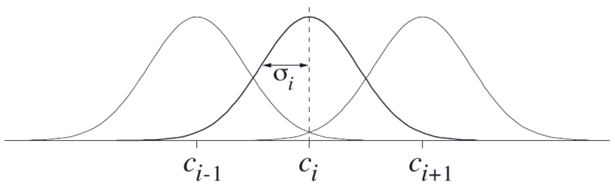
Flexible and computationally efficient form of coarse coding. Use N tiling's, each one composed of M tiles. The feature vector is a $N \times M$ matrix. The feature value is 1 if the state is inside a tile, 0 otherwise. Every state has the same number of active features.


Continuous 2D state space
Tiling 1
Tiling 2
Tiling 3
Tiling 4
Point in state space to be represented
Four active tiles/features overlap the point and are used to represent it

Radial Basis Functions

RBFs are a generalization of coarse coding. Feature values are real numbers between $[0,1]$.
The (non-normalized) Gaussian kernel is a typical RBF with the mean c_i and variance σ_i^2:

$$\phi_i(s) = e^{\left(-\frac{\|s-c_i\|^2}{2\sigma_i^2}\right)}$$



(Deep) Neural Networks

What if constructing features by hand is difficult or impractical?
Use (deep) neural networks! Automatically extract features in hidden layers and enable processing high-dimensional data.

3. Approximated Off-Policy Methods

Of course, we can apply function approximation not only in on-, but also in off-policy methods.

3.1. Semi Gradient TD(0)

For off-policy methods we are performing an importance sampling to change the expectation from a target distribution π to the behavioural distribution q. For that we define the importance sampling ratio (measuring the occupancy measure):

$$\rho_t = \frac{\pi(a_t|s_t)}{q(a_t|s_t)}$$

The one step semi gradient off-policy TD(0) update is then:

$$w_{t+1} \leftarrow w_t + \alpha\rho_t \delta_t \nabla\hat{V}_{w_t}(s_t) \; with \; \delta_t = r_{t+1} + \gamma\hat{V}_{w_t}(s_{t+1}) - \hat{V}_{w_t}(s_t)$$

While off-policy methods might allow better exploration, they have a major flaw: updating the value function *on-policy* is important for convergence. Hence, off-policy methods with approximations can *diverge!* Divergence happens because the distribution of update is different from the on-policy Distribution. *w* is updated only during the given transition, not after. Let’s look at an example:

4. Offline Methods

So far, we only looked at *online RL* methods, i.e., methods that only use the current transitions and which can be applied “on the job.” *Offline* or *batch RL* methods, on the other hand, can use data (transitions and trajectories) that was collected previously:

$$D =< s_i, a_i, r_i, s_i' >$$

4.1. Least Squares Temporal Differences (LSTD)

Gradient and semi gradient methods with linear approximation converge to a point near to the local optimum. The solution found is called fixed point w_{TD}:

$$\widetilde{\boldsymbol{w}} = \widetilde{\boldsymbol{w}} + \alpha\left(\boldsymbol{b} - \boldsymbol{A}_{\boldsymbol{w}_t}\right)$$

where $\boldsymbol{b} = r_{t+1}\phi\,(s_t)\;\epsilon R^d$, $\boldsymbol{A} = \phi\,(s_t)\;(\phi\,(s_t) - \gamma\phi\,(s_{t+1})\,)^T\;\epsilon R^d x R^d$. The fixed point is therefore $\boldsymbol{w_{TD}} = \boldsymbol{A^{-1}b}$. The value error at the fixed point is bounded as

$$\overline{VE}(\boldsymbol{w_{TD}}) \leq \frac{1}{1-\gamma}\min_{w}\overline{VE}(\boldsymbol{w})$$

If we directly find the fixed point by solving the equation above, we arrive at *least-squares TD (LSTD)*, an *offline* variant of semi-gradient TD(0). Given a set of transitions, LSTD first computes the weight matrix and vector.

$$\widehat{\boldsymbol{A}}_{\boldsymbol{t}} = \sum_{k=0}^{t-1} \phi\,(s_k)\;(\phi\,(s_k) - \gamma\phi\,(s_{k+1})\,)^T + \varepsilon\boldsymbol{I}$$

$$\widehat{\boldsymbol{b}}_{\boldsymbol{t}} = \sum_{k=0}^{t-1} r_{k+1}\phi\,(s_k)$$

and then solves then computes the fixed point **using** $\boldsymbol{w_t} = \left(\widehat{\boldsymbol{A}}_{\boldsymbol{t}}\right)^{\boldsymbol{-1}}\widehat{\boldsymbol{b}}_{\boldsymbol{t}}$ where ε is a small regularization constant. *Least-squares policy iteration (LSPI)* extends the idea of LSTD to learning the action-value function, forming LSTDQ and combines it with greedy policy improvement. A sketch is shown in algorithm 8.

4.2. The deadly triad

Instability and divergence arises for methods based on the following elements: function approximation, bootstrapping and off-policy training. However, we need all of these and cannot get rid one of them without having a disadvantage!

Function approximation is necessary to **scale** to large problems, bootstrapping is important for **data efficiency**, off-policy training is essential for learning from **heterogeneous** experience.

5. Wrap-Up/ Questions

What are continuous problems in RL?

Why we need function approximation?

How function approximation can be used in RL

What are the consequences for using function approximation in RL?

How to improve the meaning of states by extracting features?

What are the challenges of off-policy training with function approximation?

Explain the Deadly Triad!

RoLe: What are the problems of DP?

RoLe: Can we use function approximation?

RoLe: How do batch methods work?

RoLe: How to derive LSTD?

RoLe: Why do value function methods often fail for high-dimensional continuous actions?

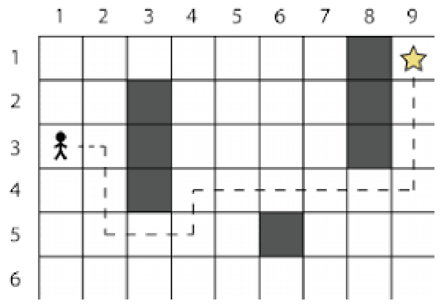
Deep Q-Learning

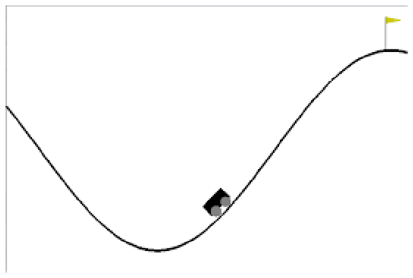
1. Recap

So far, we only discussed RL methods relying on classical non-deep ML for continuous MDPs (continuous states and discrete actions) such as On-policy, Off-policy, and Offline methods. even though it is theoretically possible to plug in a NN function approximator into, for example, SARSA. However, as we will see in this chapter, employing NNs requires a bunch of tricks and hacks to make them work. While this might be annoying, it allows us to bring RL to high-dimensional problems and even learning from images! The methods for feature extraction we have seen so far are done by “hand”, are only suitable for linear regression and low dimensional problems, hence NNs are very important! In this chapter we focus on value-based deep RL and the later we will look at actor-critic methods, which will allow us to work with a continuous action space.

2. Scaling RL to high-dimensional problems

The dimension of the state and action space is critical in RL. The curse of dimensionality is a problem in which theoretical and practical issues arising from high-dimensional problems. The RL methods we discussed can only handle low-dimensional problems like the examples below:

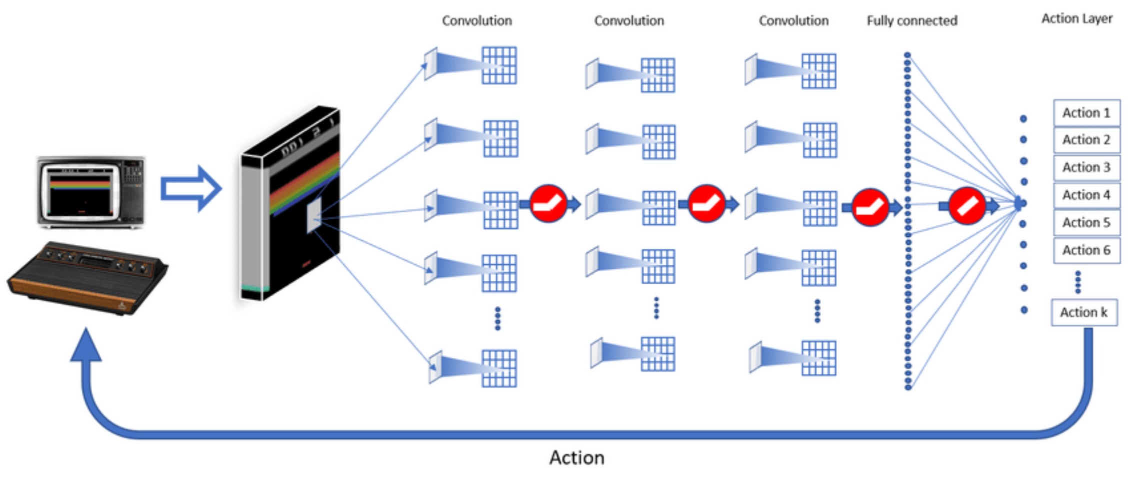




High dimensional problems have state space S with impractically large number of discrete values (e.g., each pixel of an image has an integer value $\in [0,\ 255]$). More than 8-10 dimensions (e.g., postion and velocity of joints for a robot). The action space A could be either discrete (e.g., commands to a videogame) or continuous (e.g., torque to joints of a robot).

Question: But how do we enable RL to solve more complex problems?

Solution: High-dimensional state/action spaces can be handled with deep neural networks and the use of deep learning techniques.



3. Value-based Deep RL – Deep Q-Learning (DQN)

Recall the definition of action-value function and suppose the state space S is high dimensional:

$$Q_\pi(s,a) = E_\pi\left[\sum_t \gamma^t r_{t+1} | s_0 = s,\ a_0 = a\right]$$

The goal in deep Q-Learning is to approximate the action-value function using a deep neural network with parameters w, $\hat{Q}_{\boldsymbol{w}}(s,a) \approx Q^\pi(s,a)$. Hence, this method is also called the *deep Q-network (DQN).* In *tabular* Q-learning, we just set the new action-value to a new estimate. Of course, this is not possible when the action-value function is a NN, and we therefore instead minimize the expected squared TD error:

$$L_i(\boldsymbol{w}) = E_{(s,a,r,s')\sim D}\left[\left(r + \gamma \max_{a' \in A} \hat{Q}_{\boldsymbol{w}}(s',a') - \hat{Q}_{\boldsymbol{w}}(s,a)\right)^2\right]$$

where the expectation is over a dataset D of trajectories. With this approach, we have a variety of problems:

1. The loss contains bootstrapping, is off-policy and of course we are doing a function approximation. That's the *deadly triad* we talked about in the previous chapter.

2. We cannot use offline algorithms, because an offline dataset of transitions is assumed unavailable due to the complexity of the problems (e.g., FQI).

3. The data must be collected online but training a neural network in online RL can lead to catastrophic forgetting.

We now go over some established methods for tackling these problems and making DQN work in the first place. The complete deep Q-learning with all essential tricks (table below) employed is summarized in algorithm 9.

Type	Name	Tackled Issue	Approach
Essential	Replay Buffer Target Network Minibatch Updates Reward-Clipping	distribution shift reuse instability inefficiency unstable optimization	reuse old transitions use a target network for the TD error sample small minibatch clip the reward
Enhance	Double DQN Prioritized Replay Buffer Dueling DQN Noisy DQN Distributional DQN	Overestimation sample inefficiency recovering V and A exploration stochastic rewards	max. over target, evaluate with current bias sampling towards large TD error explicitly split NN output into V and A add noisy linear layers model return distribution, not expectation

While DQN is extremely powerful, it comes at high cost! Learning requires many samples, the algorithm is highly sensitive to hyperparameter tuning (e.g., the learning rate), and computation times are enormous. The following table summarizes each

Algorithm 9: Deep Q-Learning

```
     Input: environment, differentiable approximator $\hat{Q}: S \to \mathbb{R}$ with parameters $\boldsymbol{w}$
     Output: estimated parameters $\boldsymbol{w}$
  1: initialize replay buffer $D$ to capacity $N$
  2: initialize action-value function $\hat{Q}$ with random weights $\boldsymbol{w} \in \mathbb{R}^d$
  3: initialize target action-value function weights $w' \leftarrow w$
  4: repeat
  5:     initialize $s = s_0$
  6:     while $s$ is not terminal do
             // Execute the policy.
  7:         take action $a \sim \epsilon - greedy(\hat{Q}_w)$, observe reward $r$ and next state $s'$
  8:         store transition $< s, a, r, s' >$ in $D$
             // Sample a minibatch of transitions from the replay buffer.
  9:         $< s_i, a_i, r_i, s_i' >_{i=1}^{M} \sim D$
             // Compute TD targets.
 10:         $y_i \leftarrow \begin{cases} r_i & if\ s_i'\ is\ terminal \\ r_i + \gamma \max_{a' \in A} \hat{Q}_{w'}(s_i', a') & otherwise \end{cases}$
             // Perform the optimization
 11:         perform gradient descent step on $\left(y_i - \hat{Q}_w(s_i, a_i)\right)^2$ w.r.t. $\boldsymbol{w}$
 12:         every C steps update target $\boldsymbol{w}' \leftarrow \boldsymbol{w}$
 13: until convergence
 14: return $\boldsymbol{w}$
```

3.1. Replay Buffer

In replay buffer past transitions are collected and stored in a queue a.k.a. replay buffer of finite capacity P to reuse them for updates:

$$< s_1, a_1, r_2, s_2 > ,1$$
$$< s_2, a_2, r_3, s_3 > ,2$$
$$< s_t, a_t, r_{t+1}, s_{t+1} >, P$$

Off-policy updates allow to reuse transitions out of the sampling distribution**. This reduces the negative impact of distribution shift.**

3.2. Target network

We are keeping a copy of the neural network to update it periodically (e.g., every C steps). The weights $\boldsymbol{w}$ of the online network are copied in the target network as $\boldsymbol{w}'$:

$$\boldsymbol{w} \rightarrow \boldsymbol{w}'$$

and kept fixed for the next C steps. The copy of the neural network, the target network is used to compute the target of the mean squared TD-error:

$$L_i(\boldsymbol{w}) = E_{(s,a,r,s')\sim D}\left[\left(r + \gamma \max_{a' \in A} \hat{Q}_{\boldsymbol{w}'}(s', a') - \hat{Q}_{\boldsymbol{w}}(s, a)\right)^2\right]$$

This avoids instability due to function approximation as the TD target does not change as frequent as before.

3.3. Minibatch Updates

At each step we uniformly sample a minibatch of N transitions from the replay buffer:

$$L_i(\boldsymbol{w}) = \sum_{i=1}^{N}\left[\left(r_i + \gamma \max_{a' \in A} \hat{Q}_{\boldsymbol{w}'}(s_i{}', a') - \hat{Q}_{\boldsymbol{w}}(s_i, a_i)\right)^2\right]$$

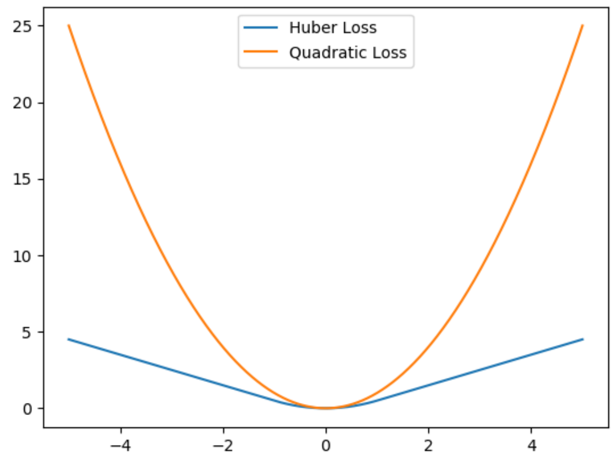
This improves efficiency with respect to training on all transitions. Also, random samples are closer to fulfilling the i.i.d.-property assumed by SGD compared to the temporally correlated trajectories.

3.4. Reward and Target Clipping

Instead of using the “real” reward, we clip the values between $[-1, 1]$. This also clips the TD error:

$$r_i + \gamma \max_{a' \in A} \hat{Q}_{\boldsymbol{w}'}(s_i{}', a') - \hat{Q}_{\boldsymbol{w}}(s_i, a_i) \in [-1,1]$$

as it turns out to be sufficient to use the Huber loss as seen in figure below instead of the quadratic loss. This improves the stability of the optimization as values stay reasonably small.



4. DQN enhancements

Even though DQN is already powerful as is, it still has some flaws we want to and can overcome. In this section we discuss some of these flaws and approaches to fixing them. All enhancements are also summarized in Table 1.

4.1. Overestimation and Double Deep Q-Learning

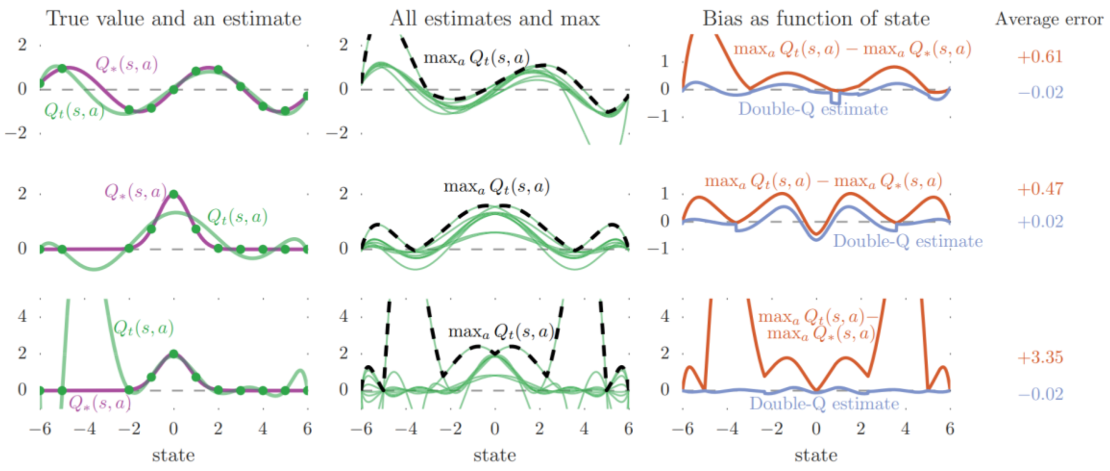
Let’s revisit the DQN loss (mean squared TD error):

$$L_i(\boldsymbol{w}) = E_{(s,a,r,s')\sim D}\left[\left(r_i + \gamma \max_{a'\in A} \hat{Q}_{\boldsymbol{w}'}(s_i{}', a') - \hat{Q}_{\boldsymbol{w}}(s_i, a_i)\right)^2\right]$$

It is known that the maximum operator in DQN (and Q-Learning) leads to overestimating the real action value. That’s not necessarily bad, but it could lead to suboptimal performance in highly stochastic problems. So how do we solve this problem? By applying the idea of Double Q-Learning to DQN. Instead of maximizing over $\hat{Q}_{\boldsymbol{w}'}(s_i', a')$, we find the action that maximizes the online policy $\hat{Q}_{\boldsymbol{w}}(s_i', a_i')$ and evaluate it using the target value function:

$$L_i(\boldsymbol{w}) = E_{(s,a,r,s')\sim D}\left[\left(r_i + \gamma \hat{Q}_{\boldsymbol{w}'}(s_i', argmax_{a'\in A} \hat{Q}_{\boldsymbol{w}'}(s_i', a')) - \hat{Q}_{\boldsymbol{w}}(s_i, a_i)\right)^2\right]$$

Notice that the parameters/weights of the action-value functions being used for maximization are different! We call this approach *double DQN (DDQN)*.



4.2. Prioritized replay buffer

The replay buffer can contain hundreds of thousands of transitions but not all are equally useful. Transition resulting high TD error δ are more informative:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

where $p_i = |\delta| + \varepsilon$ in which ε is a small positive constant to avoid degenerate cases. The variable α regulates the prioritization ($\alpha = 0$ is uniform sampling). Prioritized sampling introduces bias in the estimation which we can correct by using importance sampling:

$$w = \left(\frac{1}{NP(i)}\right)^\beta$$

Where β regulates the strength of importance sampling ($\beta = 1$ fully compensates for the non-uniform probabilities). Importance sampling weights w_i are normalized by $\frac{1}{\max\limits_i w_i}$ for stability reasons. In practice the β coefficient is annealed from a small initial value $\beta_0 < 1$ to 1.

4.3. Dueling DQN

It may be helpful from time to time to recover the state-value and advantage from the action-value function. However, given a $Q = V + A$, we cannot recover the latter. The idea of *duelling DQN* is to split the output of the Q-network into two streams $\hat{V}_{w,\beta}, \hat{A}_{w,\alpha}$ i.e., a network with two regression heads parameterized by β and α, respectively. However, instead of just adding up the two outputs to get the action-value, duelling DQN uses:

$$\hat{Q}_{w,\alpha,\beta}(s,a) = V_{w,\beta}(s) + (\hat{A}_{w,\alpha}(s,a) - \max_{a' \epsilon A} \hat{A}_{w,\alpha}(s,a'))$$

forcing $\max\limits_{a \epsilon A} \hat{Q}_{w,\alpha,\beta}(s,a) = V_{w,\beta}(s)$.

4.4. Noisy DQN

Usually, exploration is performed directly on policy level using a ε −greedy policy. That is, we perturb ("stören") the action given an action-value function. However, we can also achieve exploration by perturbing the action-value function directly by adding noisy parameters ε to the NN's layers (weights and bias):

$$w, b = \mu + \Sigma\, \sigma \odot \varepsilon \qquad // \odot = \text{component-wise multiplication for vectors/matrices}$$

Where $\varsigma = (\mu, \sigma)$ are learnable parameters and ε is a fixed zero-mean noise. Optimization of the parameters is then w.r.t. the expected loss over the noise ε:

$$\bar{L}(\varsigma) = E_{\varepsilon}[L(\varsigma)]$$

For example, we can replace the usual linear layer $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$ with input $x \in R^{p}$, weights $W \in R^{qxb}$ and biases $b \in R^{q}$ with noise using:

$$\tilde{\boldsymbol{y}} = (\boldsymbol{\mu_W} + \boldsymbol{\sigma_W} \odot \boldsymbol{\varepsilon_W})\boldsymbol{x} + (\boldsymbol{\mu_b} + \boldsymbol{\sigma_b} \odot \boldsymbol{\varepsilon_b})$$

where $\boldsymbol{\mu}_W \in R^{qxp}, \boldsymbol{\sigma}_W \in R^{qxp}, \boldsymbol{\mu}_b \in R^{q}$ and $\boldsymbol{\Sigma}_b \in \mathrm{R}^q$ are learnable parameters and $\varepsilon_W \sim \mathrm{N}(\mathbf{0}, \mathbf{I}) \in R^{qxp}$ and $\varepsilon_b \sim \mathrm{N}(\mathbf{0}, \mathbf{I}) \in R^{q}$ are noise random variables. In practice we only make the last layer of the Q network noisy.

4.5. Distributional DQN

Recall the Q function is the expected cumulative discounted return:

$$Q^{\pi}(s, a) = E_{\pi}[J_t | s_t = s, a_t = a] = E_{\pi}\left[\sum_{i=0}^{\infty} \gamma^{i} r_{i+1} | s_t = s, a_t = a\right]$$

The idea of distributional DQN is to model the whole return distribution. Note that we do not model the epistemic uncertainty (arises due to a lack of data/information/knowledge) of our model, but the intrinsic aleatoric uncertainty (arises from inherent variability and cannot be reduced even with additional information) of the environment! For that we define the distributional value function $Z^{\pi}(s, a)$ and the reward $R(s, a)$ to be a distribution over rewards. To reason about how the distributional value transitions, let P^{π} be the transition operator such that

$$(P^{\pi} Z^{\pi})(s, a) = Z^{\pi}(s', a')$$

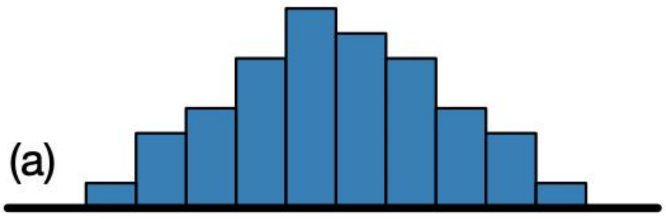
with $s' \sim P(\cdot\, | s, a)$ and $a' \sim \pi(\cdot\, | s')$. Also, we define the distributional Bellman operator T such that:

$$(T^{\pi} Z^{\pi})(s, a) = R(s, a) + \gamma (P^{\pi} Z^{\pi})(s, a)$$

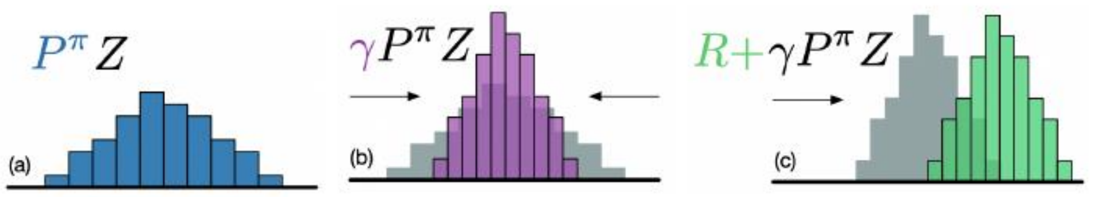
Notice that we have 3 sources of randomness here:

a) randomness in the reward $R(s, a)$
b) randomness in the transition $P^{\pi}(s, a)$
c) randomness in the next-state value distribution $Z^{\pi}(s', a')$

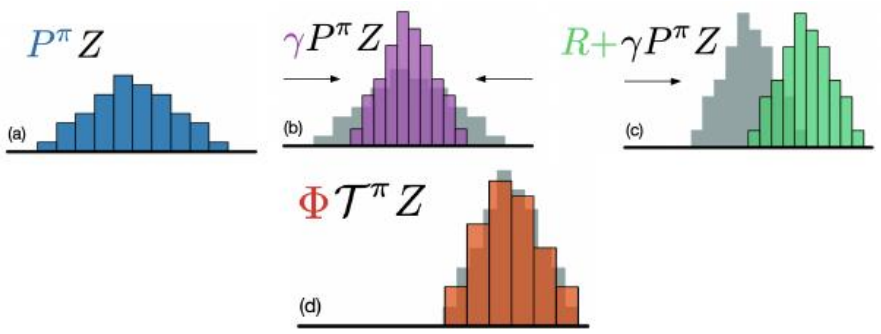
The distributional value function is approximated or modelled as a discrete distribution which contains N elements called atoms. Each atom represents a canonical return as seen in the figure below. Hence, we can now write the categorical value distribution approximation $Z_w(s, a) \approx Z^{\pi}(s, a)$



First, we apply the distributional Bellmann operator $(T^{\pi}Z_w)(s,a) = R(s,a) + \gamma(P^{\pi}Z_w)(s,a)$ to the distributional value function



While we can now explicitly compute the update, we have a problem: The **shrinking** (discount factor) and **translation** (reward) change the **support** of the distribution such that $T^{\pi}Z_w$ and Z_w have close to disjoint support. Hence, we apply a projection Φ to project the distribution back to the original support.



The distribution Z_w is then updated (To find new parameters w_{new}) by minimizing the KL-Divergence:

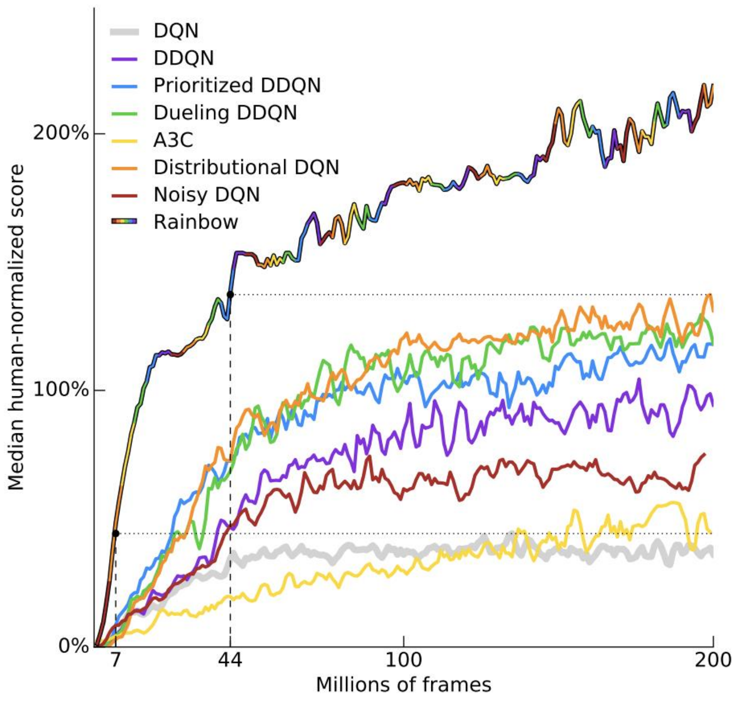
$$D_{KL}(\Phi T Z_{w'}(s,a) || Z_w(s,a))$$

4.6. Rainbow DQN

Rainbow is ahigh-performing variant of DQN that combines the previously described methods:

- Estimation bias (Double DQN, Dueling DQN)
- Sample-efficiency (Prioritized replay)
- Exploration (Noisy DQN)
- Dealing with uncertainty of the return (Distributional DQN).

in a single algorithm. This is possible as luckily; all these methods are compatible. With a correct implementation, rainbow is extremely powerful, although hard to implement. Additionally, rainbow uses n-step return to estimate the Q-function. The following figure compares each method with each other:



5. Wrap-Up

- curse of dimensionality and its effect in RL
- deep learning in RL for handling high-dimensional problems
- problems of deep RL and some techniques addressing them
- the DQN algorithm and how to set up experiments with it
- enhancing DQN by improving function approximation and sample usage
- improving key problems of RL, e.g., exploration, by combining deep learning techniques and DQN
- Additional reading material:
    - Paper: “Playing Atari with Deep Reinforcement Learning” (Mnih et al., 2013)
    - Paper: “Deep Reinforcement Learning with Double Q-Learning” (Hasselt et al., 2016)
    - Paper: “A Distributional Perspective on Reinforcement Learning” (Bellemare et al., 2017)
    - Paper: “Curiosity-Driven Exploration by Self-Supervised Prediction” (Pathak et al., 2017)

6. Questions

What is the curse of dimensionality? How does it affect RL?

How can deep learning methods be used in RL to handle high-dimensional problems?

What are the problems of deep RL and what are some techniques to address them?

What is the DQN algorithm and how can we setup experiments with It?

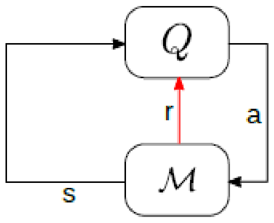
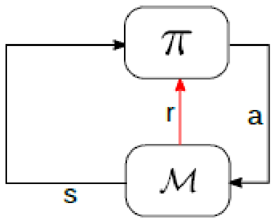
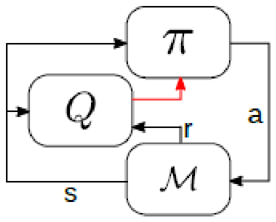
How to enhance DQN by improving function estimation and use of samples?

How can we combine techniques from deep learning with DQN to improve key problems of RL, e.g., exploration?

Policy Search and Policy Gradient Methods

1. Policy Search

There are three main classes of Reinforcement Learning algorithms as shown in the table:

Value Based	Policy Search	Actor-Critic
		
• Also known as Critic-only methods • All the methods we have seen so far are value-based • The policy is implicitly defined by the value function	• Also known as Actor-only methods • The value function is not used • The policy is explicitly defined	• A combination of the previous two methods • Both the policy (actor) and the value function (critic) are explicitly defined

So far, we always learned/estimated a value function and extracted a policy from it. However, why bother to learn a value function and not directly learn a policy which is what we are interested in? This is the idea of *policy search.* Given an MDP $M = < S, A, R, P, \iota, \gamma >$, policy search can be formalized as an explicit optimization problem over the expected reward:

$$\pi^* = argmax_\pi J_\pi = argmax_\pi E_{\tau\sim\iota,P,\pi}[J(\tau)]$$

where $J(\tau)$ is the discounted return of the trajectory τ and the expectation is performed by sampling trajectories on the environment using the policy π. But why should we use a policy search algorithm? There are many benefits but also some drawbacks as listed in the following table:

Benefits	Drawbacks
• Learning a value function may be more difficult than directly learning the policy • Domain knowledge can be encoded in the policy • Demonstrations can be used to initialize the policy (e.g., with a (suboptimal) result from imitation learning) • For continuous action spaces, computing the action that maximize the value function may be difficult	• No guarantee of convergence to the optimal policy: optimization may converge to a local optimum though • The policy evaluation is inefficient and may have high variance: the policy must be evaluated by Monte-Carlo rollouts in the environment

Policy Search algorithms can be used to solve both discrete and continuous control actions. In the following, for simplicity we will assume that the control variable is a vector of continuous variables $a = [a_0, \dots, a_i, \dots a_N]^T$. However, the same can be applied to discrete action policies! Note that in approximate value-based methods we used discrete control variables!

Like before exploration is crucial in policy search! The most common exploration strategy is to use a Gaussian policy:

$$\pi_\theta(a|s) = N(a|\mu_\theta(s), \Sigma_\theta(s))$$

where the mean (μ_θ) and covariance (Σ_θ) matrix is given by a (usually learnable) state-dependent function approximator with parameters θ. This can, for instance, be a NN. For finite discrete action policies, the Boltzmann policy is often used:

$$\pi_\theta(a|s) = \frac{e^{\frac{f_\theta(a,s)}{\tau}}}{\sum_{i=0}^{|A|} e^{\frac{f_\theta(a,s)}{\tau}}}$$

In fact, we will also confine ourselves to *parametric* policies, i.e., policies π_θ (probability to take action a given current state s and parameters θ. We are choosing the action which has the highest probability in that state. Probability distribution over actions a defined by θ! These parameters θ define how the policy assigns probabilities to different actions a given a particular state s, thereby determining the agent's behavior in the environment.) with parameters θ. The optimization problem is then:

$$\pi^* = argmax_{\pi \epsilon \{\pi_{\theta_i}\}} J_\pi = argmax_\theta J_{\pi_\theta}$$

This optimization is equivalent to the optimization of the objective J w.r.t. the parameter vector θ. Thus, we can use standard gradient optimization techniques using the Policy Gradient $\nabla_\theta J_{\pi_\theta}$

2. Policy Gradient

In *policy gradient (PG)* methods, we maximize the objective J_θ directly by taking its gradient w.r.t. to the policy parameters θ the so-called policy gradient $\nabla_\theta J_\theta$. Algorithm 10 shows a prescription on how this works.

Algorithm 1: Policy Search using Policy Gradient

```
   Input: environment, differentiable parametric policy $\pi_\theta$ with parameters $\theta$
   Output: optimized policy $\pi$
1: initialize $\theta$ arbitrarily
2: repeat
       // Collect data and perform gradient update
3:     $D \leftarrow$ genrate rollouts using $\pi_\theta$
4:     $\nabla_\theta J_\theta \leftarrow$ compute gradient descent using D
5:     $\theta \leftarrow \theta + \alpha \nabla_\theta J_\theta$
6: until convergence
7: return $\pi_\theta(\infty)$
```

2.1. Likelihood Ratio Trick

The most pressing question of PG methods is how we compute the gradient as it goes through an expectation. The policy gradient for stochastic policies can be written as:

$$\nabla_\theta J_\theta = \nabla_\theta \int p(\tau|\theta) J(\tau) d\tau = \int \nabla_\theta p(\tau|\theta) J(\tau) d\tau$$

By the Markov property, the probability $p(\tau|\theta)$ of a trajectory τ can be decomposed into:

$$p(\tau|\theta) = \iota(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t) \pi_\theta(a_t|s_t)$$

where $\iota(s_0)$ is the initial state distribution. To compute $\nabla_\theta p(\tau|\theta)$ we need to know the transition function P! So how can we compute the gradient without knowing the transition model? By exploiting the likelihood ratio trick, we can compute the gradient of a MC estimate of J_θ exactly!

$$\nabla_\theta logp(\tau|\theta) = \frac{\nabla_\theta p(\tau|\theta)}{p(\tau|\theta)} \xleftrightarrow{p(\tau|\theta) \neq 0} \nabla_\theta p(\tau|\theta) = p(\tau|\theta) \nabla_\theta logp(\tau|\theta)$$

Applying the likelihood ratio trick to our gradient estimator simplifies it to an expectation over the gradient of the log-likelihood:

$$\nabla_\theta J_\theta = \int p(\tau|\theta) \nabla_\theta logp(\tau|\theta) J(\tau) d\tau = E_\tau[\nabla_\theta logp(\tau|\theta) J(\tau)]$$

And this expectation of the MC estimate can be estimated by sampling (approximated with a sample mean):

$$E_x[f(s)] \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i) \leftrightarrow E_\tau[\nabla_\theta logp(\tau|\theta)J(\tau)] \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta logp\left(\tau^{[i]}\middle|\theta\right)J\left(\tau^{[i]}\right)$$

Hence, **we must only compute** $\nabla_\theta logp(\tau|\theta)$ efficiently! We can apply logarithmic rules and rewrite $logp(\tau|\theta)$ as:

$$logp(\tau|\theta) = log\iota(s_0) + \sum_{t=0}^{T-1}\left(logP(s_{t+1}|s_t, a_t) + log\,\pi_\theta(a_t|s_t)\right)$$
$$= \sum_{t=0}^{T-1} log\,\pi_\theta(a_t|s_t) + constant$$

and everything except for the log-policy (which we model explicitly) is constant w.r.t. θ! Hence the gradient only depends on the policy:

$$\nabla_\theta logp(\tau|\theta) = \sum_{t=0}^{T-1} log\,\pi_\theta(a_t|s_t)$$

This is the simplest likelihood-ratio estimator we can imagine! But this can be biased!

2.2. REINFORCE Algorithm and Baselines

The simplest likelihood ratio approach is called *REINFORCE.* If we combine the simple policy gradient (derived with logarithmic rules) with the MC estimate (derived with likelihood ratio trick) the policy gradient can be written as:

$$\nabla_\theta J_\theta = E_\tau\left[\left(\sum_{t=0}^{T-1} log\,\pi_\theta(a_t|s_t)\right)J(\tau)\right]$$

The expectation of the MC estimate can be estimated by sampling (approximated with a sample mean). This results in the *REINFORCE* gradient estimator:

$$\nabla_\theta^{\mathrm{RF}} J_\theta = \frac{1}{N}\sum_{i=0}^{T-1}\left[\left(\sum_{t=0}^{T-1} \nabla_\theta log\,\pi_\theta\left(a_t^{[i]}\middle|s_t^{[i]}\right)\right)J\left(\tau^{[i]}\right)\right]$$

While this estimator is nice, simple, and *unbiased,* it has *very high variance*! In fact, a specific estimate is useless and causes divergence quickly. However, we can introduce a *baseline b* to reduce the variance by subtracting it from the return $J(\tau)$. This leads to the *REINFORCE* policy gradient with baseline b

$$\nabla_\theta^{\mathrm{RF}} J_\theta = \frac{1}{N}\sum_{i=0}^{T-1}\left[\left(\sum_{t=0}^{T-1} \nabla_\theta log\,\pi_\theta\left(a_t^{[i]}\middle|s_t^{[i]}\right)\right)\left(J\left(\tau^{[i]}\right) - b\right)\right]$$

If a baseline b is not dependent on state and actions, adding a baseline will not add a bias to the estimate as its expectation vanishes but it still reduces the variance:

$$E_\tau[\nabla_\theta logp(\tau|\theta)b] = \int p(\tau|\theta)\,\nabla_\theta logp(\tau|\theta)bd\tau = b\nabla_\theta \int p(\tau|\theta)\,d\tau = b\nabla_\theta(1) = 0$$

The same result holds for vector baselines. We have different baselines for every gradient element. In each iteration, we can also find an optimal baseline:

$$b^* = argmin_b Var_\tau[\nabla_\theta logp(\tau|\theta)(J(\tau) - b)]$$

The optimal (vectorial) baseline for REINFORCE is (where the power and the division are element-wise):

$$b^{RF} = \frac{E_\tau[(\sum_{t=0}^{T-1} \nabla_\theta log\, \pi_\theta(a_t|s_t))^2 J(\tau)]}{E_\tau[(\sum_{t=0}^{T-1} \nabla_\theta log\, \pi_\theta(a_t|s_t))^2]}$$

Algorithm 2: REINFORCE Gradient Estimation with Optimal Baseline

```
    **Input:** transition dataset $D$, differentiable parametric policy $\pi_\theta$ with parameters $\theta$
    **Output:** $\nabla_\theta^{\mathrm{RF}} J_\theta$
     // Compute returns for all $\tau^{[i]} \epsilon D$
1:  $J^{[i]} \leftarrow J(\tau^{[i]}) =$
     // Compute optimal baseline
2:  $b^{RF} \leftarrow \frac{\sum_{i=1}^{N-1}(\sum_{t=0}^{T-1} \nabla_\theta log\, \pi_\theta(a_t|s_t))^2 J^{[i]}}{\sum_{i=1}^{N-1}(\sum_{t=0}^{T-1} \nabla_\theta log\, \pi_\theta(a_t|s_t))^2}$
     // Estimate the Policy Gradient
3:  $\nabla_\theta^{\mathrm{RF}} J_\theta \leftarrow \frac{1}{N}\sum_{i=0}^{T-1} \left[\left(\sum_{t=0}^{T-1} \nabla_\theta log\, \pi_\theta\left(a_t^{[i]}\middle|s_t^{[i]}\right)\right)\left(J(\tau^{[i]}) - b\right)\right]$
4: **return** $\nabla_\theta^{\mathrm{RF}} J_\theta$
```

2.3. GPOMDP Algorithm

Even when using the optimal baseline, the REINFORCE algorithm has high variance. The main reason for that is that the Monte-Carlo estimates of trajectory returns are extremely noisy! A possible solution is the GPOMDP (Gradient for partially observed MDP) algorithm. We can further reduce the variance by not considering the total return of a trajectory:

$$J(\tau) = \sum_{t=0}^{T-1} \gamma^t r_{t+1}$$

but instead decompose the return in step-based rewards and observe that the rewards from the past do not depend on actions in the future (We can neglect the derivatives of futures actions):

$$E[\nabla_\theta log\, \pi_\theta(a_t|s_t)\gamma^k r_{k+1}], \forall k \leq t$$

Plugging this in the with a time-dependent baseline in $\nabla_\theta^{\mathrm{RF}} J_\theta$, yields the GPOMDP gradient:

$$\nabla_\theta^{\mathrm{GP}} J_\theta = E_\tau[\sum_{k=0}^{T-1} \sum_{t=0}^{k} \nabla_\theta log\, \pi_\theta\left(a_t^{[i]}\middle|s_t^{[i]}\right)\left(\gamma^k r_{k+1} - b_k\right)$$

with the optimal baseline b_k:

$$b_k = \frac{E_\tau\left[\left(\sum_{t=0}^{k} \nabla_\theta log\, \pi_\theta(a_t|s_t)\right)^2 \gamma^k r_{k+1}\right]}{E_\tau\left[\left(\sum_{t=0}^{k} \nabla_\theta log\, \pi_\theta(a_t|s_t)\right)^2\right]}$$

The derivation of this baseline is analogous to the REINFORCE case. The approach is summarized in algorithm 3. Both the gradient and baselines can be estimated from trajectory rollouts!

Algorithm 3: GPOMDP

```
   Input: transition dataset $D$, differentiable parametric policy $\pi_\theta$ with parameters $\theta$
   Output: $\nabla_\theta^{\text{GP}} J_\theta$
    //Compute optimal time-dependant baseline for each timestep $k$
1: $b_k \leftarrow$
    // Estimate the Policy Gradient
3: $\nabla_\theta^{\text{RF}} J_\theta \leftarrow \frac{1}{N}\sum_{i=0}^{T-1} \left[\left(\sum_{t=0}^{T-1} \nabla_\theta log\, \pi_\theta\left(a_t^{[i]}\middle|s_t^{[i]}\right)\right)\left(J(\tau^{[i]}) - b\right)\right]$
4: return $\nabla_\theta^{\text{RF}} J_\theta$
```

3. The Natural Gradient

So far, we used classical gradient methods that assume that the metric of the space being optimized is Euclidean. However, parameters can have a different effects on the probability distribution! To measure the "distance" between two probability distributions we can use the Kullback–Leibler divergence (KL). However, the KL is not a metric (is not symmetric) and is difficult to compute/use.

Instead, we can use the Fisher Information Matrix:

$$F_\theta = Var_\tau[\nabla_\theta p_\theta(\tau)] = Var_\tau\left[\sum_{t=0}^{T-1} log\, \pi_\theta(a_t|s_t)\right]$$

For small variations $\delta\theta$ of the parameters it holds:

$$KL(p(\tau|\theta + \delta\theta)||p(\tau|\theta)) \approx \delta\theta^T F_\theta \delta\theta$$

The Fisher information matrix is the second-order approximation of the KL. The core idea of the *natural* PG is to update the policy under a KL constraint ε (using a 2° order approximation). For a given “vanilla” update $\delta\theta^{VG}$, we find the natural gradient update $\delta\theta^{NG}$ by solving the following optimization problem:

$$\delta\theta^{NG} = argmax_{\delta\theta}\delta\theta^T\delta\theta^{VG}$$
$$s.t.\ \delta\theta^T F_\theta \delta\theta \leq \varepsilon$$

The solution is:

$$\delta\theta^{NG} \propto \mathrm{F}_\theta^{-1}\delta\theta^{VG}$$

This solution leads us to the natural policy gradient:

$$\nabla_\theta^{\mathrm{NG}} J_\theta = F_\theta^{-1} \nabla_\theta J_\theta$$

Fon any KL constraint ε we can find a learning rate such that the KL update is bounded. The KL bound holds only for sufficiently small parameter updates. The natural gradient is independent from policy parametrizations.

4. The Policy Gradient Theorem

Let $\rho^{\pi_\theta}(s)$ be the occupancy measure under the policy π_θ:

$$\rho^{\pi_\theta}(s) = \sum_{t=0}^{\infty} \gamma^t P(s_t = s|\pi_\theta)$$

Where the discount factor γ can be viewed as a termination probability. Then for any MDP the policy gradient can be computed as:

$$\nabla_\theta J_\theta = \int_S \rho^{\pi_\theta}(s) \int_A \nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a) da ds$$

The occupancy measure is not a distribution but can be interpreted as the expected number of visits of s in a trajectory τ:

$$\int_S \rho^{\pi_\theta}(s) = \int_S \sum_{t=0}^{\infty} \gamma^t P(s_t = s|\pi_\theta)\, ds$$

$$= \sum_{t=0}^{\infty} \gamma^t \int_S P(s_t = s|\pi_\theta)\, ds$$

$$= \sum_{t=0}^{\infty} \gamma^t = \frac{1}{1-\gamma} \neq 1$$

We can define the discounted state distribution $d^{\pi_\theta}(s)$ where $(1-\gamma)$ ensures that the agent's initial visit to the state at time step 0 is not discounted:

$$d^{\pi_\theta}(s) = (1-\gamma)\rho^{\pi_\theta}(s) = \rho^{\pi_\theta}(s) - \gamma\rho^{\pi_\theta}(s)$$

Thus, we can rewrite the Policy Gradient Theorem as:

$$\nabla_\theta J_\theta = \frac{1}{1-\gamma} \int_S d^{\pi_\theta}(s) \int_A \nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a) da ds$$

$$= \frac{1}{1-\gamma} \int_S d^{\pi_\theta}(s) \int_A \pi_\theta(a|s) \nabla_\theta log\, \pi_\theta(a\,|s\,) Q^{\pi_\theta}(s,a) da ds$$

$$= \frac{1}{1-\gamma} E_{(s,a)\sim d^{\pi_\theta}, \pi_\theta}[\nabla_\theta \log \pi_\theta(a\,|s\,) Q^{\pi_\theta}(s,a)]$$

$$\propto E_{(s,a)\sim d^{\pi_\theta},\pi_\theta}[\nabla_\theta \log \pi_\theta(a\,|s\,)Q^{\pi_\theta}(s,a)]$$

and we can readily apply MC-based techniques if we know the true action-value function. Note how nicely the Policy Gradient Theorem connects policy search and gradient estimation with value functions. However, we need the action-value function to compute the gradient. If we estimate the Q-function using Monte Carlo rollouts the calculation is equivalent to GPOMDP. If we estimate the Q-function using TD learning (which is much more robust than MC), we have an *actor-critic* algorithm which are the current state of the art (SOTA) in RL.

4.1. Compatible function approximation

But now we have a major issue the policy gradient theorem requires the knowledge of the true value function or to estimate it by Monte-Carlo sampling. If we use an estimated Q-function, the gradient estimation is biased. The solution is to use a compatible function approximation. Let's look at the compatible function approximation theorem. If the following conditions hold:

1. the value function approximation is compatible to the policy:

$$\nabla_\omega \hat{Q}_\omega(s,a) = \nabla_\theta\, log\, \pi_\theta(a\,|s\,)$$

2. Value function parameters ω minimize the mean–squared value error

$$\omega^* = argmin\, E_{(s,a)\sim d^{\pi_\theta},\pi_\theta}[Q^{\pi_\theta}(s,a) - \hat{Q}_\omega(s,a)]$$

Then the policy gradient estimate is unbiased:

$$\nabla_\theta J_\theta \approx E_{(s,a)\sim d^{\pi_\theta},\pi_\theta}[\nabla_\theta\, log\, \pi_\theta(a\,|s\,)\, \hat{Q}_{\omega^*}(s,a)]$$

We can write the compatible function approximation (CFA) as a linear combination of the policy log-gradient:

$$\hat{Q}_\omega(s,a) = \nabla_\theta\, log\, \pi_\theta(a\,|s\,)^T\, \boldsymbol{\omega}$$

Unfortunately, we cannot use TD learning and have an unbiased gradient estimate. The compatible function approximation has zero mean:

$$E_{a\sim\pi_\theta}[\nabla_\theta\, log\, \pi_\theta(a\,|s\,)^T\, \boldsymbol{\omega}] = \int_{\boldsymbol{A}} \nabla_\theta \pi_\theta(a\,|s\,)^T da\boldsymbol{\omega} = 0$$

Plugging the CFA in the policy gradient estimate equation above yields:

$$\nabla_\theta^{CFA} J_\theta = E_{\tau\sim,\pi_\theta.P}\left[\sum_{t=0}^{T-1} \nabla_\theta\, log\, \pi_\theta(a_t\,|s_t\,)\ \nabla_\theta\, log\, \pi_\theta(a_t\,|s_t\,)^T\right]\boldsymbol{\omega} = \boldsymbol{F_\theta \omega}$$

4.2. The eNAC algorithm

Let's recall the definition of the advantage function. The advantage function measures how good an action a is compared to the policy's behaviour:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

The Episodic Natural Actor-Critic (eNAC) algorithm combines both compatible function approximation and the natural gradient. The key idea is to estimate the advantage function with a compatible function approximation (as it usually has lower variance than the action-value function):

$$\hat{A}(s,a) = \nabla_\theta \log \pi_\theta(a \mid s)^T \boldsymbol{\omega}$$

As the CFA and NG gradient are given by:

$$\nabla_\theta^{\text{CFA}} J_\theta = \mathbf{F}_{\boldsymbol{\theta}} \boldsymbol{\omega}$$

$$\nabla_\theta^{\text{NG}} J_\theta = \boldsymbol{F}_{\boldsymbol{\theta}}^{-1} \nabla_\theta J_\theta$$

Combining them together we have the eNAC gradient:

$$\nabla_\theta^{\text{eNAC}} J_\theta = \boldsymbol{F}_{\boldsymbol{\theta}}^{-1} \mathbf{F}_{\boldsymbol{\theta}} \boldsymbol{\omega} = \boldsymbol{\omega}$$

However, we still must compute ω! For this, we also need an approximation of the value function (as a baseline) for the initial state. If the initial state does not change, this is just a constant and if it does, we choose a linear approximation with some features $\phi(s)$ of the state s:

$$\hat{V}_v(s) = \phi(s)^T \boldsymbol{v}$$

Now we can write:

$$J(\tau) = \hat{V}_v(s_0) + \sum_{t=0}^{T-1} \gamma^t \hat{A}_\omega(s_t, a_t)$$

which is a linear equation in terms of v and ω as we know $J(\tau)$ from sampling. For a set of N trajectories $\tau = [\tau^{[1]}, \dots, \tau^{[N]}]$, we can formulate this as a matrix-vector linear equation:

$$J(\tau) = \boldsymbol{\Psi} \begin{bmatrix} \boldsymbol{\omega} \\ \boldsymbol{v} \end{bmatrix}$$

with

$$\boldsymbol{J} = \begin{bmatrix} J(\tau^{[1]}) \\ J(\tau^{[2]}) \\ \vdots \\ J(\tau^{[N]}) \end{bmatrix}, \boldsymbol{\Psi} = \begin{bmatrix} \psi^{[1]} \\ \psi^{[2]} \\ \vdots \\ \psi^{[N]} \end{bmatrix} \; with \; \boldsymbol{\psi}^{[i]} = \begin{bmatrix} \sum_{t=0}^{T-1} \gamma^t \nabla_\theta \, log \, \pi_\theta \left(a_t^{[i]} \middle| s_t^{[i]} \right) \\ \phi\left(s_0^{[i]} \right) \end{bmatrix}$$

As this system is linear and overdetermined, we can evaluate ω and v using least squares. Note that this is an *episodic* algorithm as the calculation of $J(\tau)$ requires a complete episode. Algorithm 4 summarizes this approach.

Algorithm 4: Episodic Natural Actor-Critic

```
    Input: transition dataset $D$, differentiable parametric policy $\pi_\theta$ with parameters $\theta$
    Output: $\nabla_\theta^{\mathrm{eNAC}} J_\theta$
    // Compute for all $\{\tau^{[i]}\}_{i=1}^{N}$
1:  $J(\tau^{[i]})$
    // Compute the state features for the value function
2:  $\boldsymbol{\psi}^{[i]} = \begin{bmatrix} \sum_{t=0}^{T_i-1} \gamma^t \nabla_\theta \log \pi_\theta \left(a_t^{[i]} \middle| s_t^{[i]}\right) \\ \Phi\left(s_0^{[i]}\right) \end{bmatrix}$
    // Fit the advantage and value function (for the initial state).
3:  $\boldsymbol{J} \leftarrow \begin{bmatrix} J(\tau^{[1]}) \\ J(\tau^{[2]}) \\ \vdots \\ J(\tau^{[N]}) \end{bmatrix}$
4:  $\boldsymbol{\Psi} = \begin{bmatrix} \psi^{[1]} \\ \psi^{[2]} \\ \vdots \\ \psi^{[N]} \end{bmatrix}$
5:  $\begin{bmatrix} \boldsymbol{\omega} \\ \boldsymbol{v} \end{bmatrix} \leftarrow \left( \boldsymbol{\Psi}^{\boldsymbol{T}} \boldsymbol{\Psi} \right)^{-1} \boldsymbol{\Psi J}$
6:  return $\boldsymbol{\omega}$
```

5. Wrap-up

Now you know:

- The difference between Value-based, policy search, and Actor-Critic.
- The importance of exploration in policy search scenarios and why we use Gaussian policies for continuous control.
- The policy gradient approaches:
  - Reinforce, and the importance of reducing the variance by using a baseline
  - GPOMDP, derived from the fact that the rewards from the past do not depend on the actions in the future.
- How we can use the Fisher information matrix to derive the natural
- gradient.
- The policy gradient theorem and the connection between policy search, value-based, and actor-critic.
- How we can derive the eNAC algorithm by combining Compatible function approximation and the natural gradient.

6. Questions

What are the differences between value-based, policy search, and actor-critic methods?

In value-based (critic-only) methods the policy is defined implicitly by the value function. In Policy Search (actor-only) we are learning the policy directly and no value-function is used. Actor-Critic methods are a combination of the two previously mentioned methods.

Why is exploration important in policy search? Why do we use Gaussian policies (for continuous control)?

We need to see where we can do something good. The most common exploration strategy is to use a Gaussian policy:

$$\pi_\theta(a|s) = N(a|\mu_\theta(s), \Sigma_\theta(s))$$

where the mean (μ_θ) and covariance (Σ_θ) matrix is given by a (usually learnable) state-dependent function approximator with parameters θ. This can, for instance, be a NN. Gaussian policies are a nice way of introducing randomness by just learning the mean and covariance. The policy is then able to control exploration their self.

What are the three big approaches for computing policy gradients? What is their core idea?

This is how policy search works as Algorithm 1 shows:

Algorithm 1: Policy Search using Policy Gradient

```
   Input: environment, differentiable parametric policy $\pi_\theta$ with parameters $\theta$
   Output: optimized policy $\pi_\theta$
1: initialize $\theta$ arbitrarily
2: repeat
       // Collect data and perform gradient update
3:     $D \leftarrow$ genrate rollouts using $\pi_\theta$
4:     $\nabla_\theta J_\theta \leftarrow$ compute gradient descent using D
5:     $\theta \leftarrow \theta + \alpha \nabla_\theta J_\theta$
6: until convergence
7: return $\pi_\theta(\infty)$
```

To compute $\nabla_\theta J_\theta$ we can use 2 different methods:

1. REINFORCE with Optimal Baselines

   Rewrite the Gradient as an Integral, use likelihood ration trick to solve the gradient of transition dynamics

   1) GPOMDP (Gradient for partially observed MDP)

How can we use the FIM to compute the NG?

Parameters θ can have different effect on probability distribution. KL Divergence measures the distance between two probability distribution, but this is difficult to compute. Instead, approximate KL divergence with Fisher Information matrix. FIM is a second order approximation of KL:

$$F_\theta = Var_\tau[\nabla_\theta p_\theta(\tau)] = Var_\tau\left[\sum_{t=0}^{T-1} log\, \pi_\theta(a_t|s_t)\right]$$

What is the PGT and its connection to value-based and actor-critic methods?

Policy Gradient Theorem connects policy search and gradient estimation with value functions. However, we need the action-value function to compute the gradient.

If we estimate the Q-function using Monte Carlo rollouts the calculation is equivalent to GPOMDP.

If we estimate the Q-function using TD learning (which is much more robust than MC), we have an *actor-critic* algorithm which are the current state of the art (SOTA) in RL.

Derivation:

PGT is a theorem where we rewrite the gradient using the occupancy measure (expected number of visits of s in a generated trajectory τ by differentiable parametric policy π_θ):

$$\rho^{\pi_\theta}(s) = \sum_{t=0}^{\infty} \gamma^t P(s_t = s|\pi_\theta)$$

$$\nabla_\theta J_\theta = \int_S \rho^{\pi_\theta}(s) \int_A \nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a) da ds$$

Defining the so-called discounted state distribution where $(1-\gamma)$ ensures that the agent's initial visit to the state at time step 0 is not discounted:

$$d^{\pi_\theta}(s) = (1-\gamma)\rho^{\pi_\theta}(s)$$

Plug this in PGT and we will get:

$$\nabla_\theta J_\theta = \frac{1}{1-\gamma} E_{(s,a)\sim d^{\pi_\theta},\pi_\theta}[\nabla_\theta \log \pi_\theta(a\,|s\,) Q^{\pi_\theta}(s,a)]$$

How to derive the eNAC algorithm?

RoLe: What is likelihood-ratio gradient estimators?

RoLe: Why do baselines lower the variance of the gradient estimate?

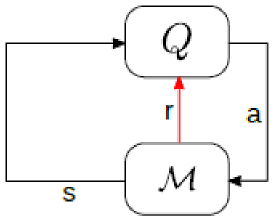
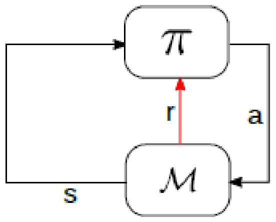
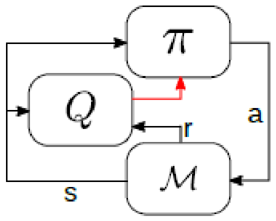
RoLe: Why is the FIM so important? How does it relate to the KL?

RoLe: What is the NG? Why is the NG invariant to reparameterization?

RoLe: What is the CFA? How is it connected to the NG?

Deep Actor-Critic On-Policy

In this chapter we move from “just” value-function methods to the SOTA of deep RL: deep actor-critic methods. These methods explicitly model both a policy (the “actor”) and the value function (the “critic”). Remember the table from the last chapter:

Value Based	Policy Search	Actor-Critic
		
• Also known as Critic-only methods • All the methods we have seen so far are value-based • The policy is implicitly defined by the value function	• Also known as Actor-only methods • The value function is not used • The policy is explicitly defined	• A combination of the previous two methods • Both the policy (actor) and the value function (critic) are explicitly defined

1. Entropy and Relative Entropy

The Entropy of a policy π in a state s is defined as:

$$H(\pi(\cdot|s)) = -\int_A \pi(a|s)\log\pi(a|s)\,da = E_{a\sim\pi}[\log\pi(a|s)]$$

This quantity measures the amount of "randomness" of the policy. For Gaussian policies, it can be computed in closed form as:

$$H(\pi(\cdot|\mu(s),\Sigma(s))) = \frac{1}{2}\log[\det(2\pi e\Sigma(s))]$$

The Relative Entropy or Kullback-Leibeler divergence (KL) between two policies π and q in a state s is:

$$KL(\pi(\cdot|s)||q(\cdot|s)) = \int_A \pi(a|s)\log\frac{\pi(a|s)}{q(a|s)}da$$

The KL measures the "distance" between the probability density of the actions at a given state s. Remember that the KL is not a proper distance:

$$KL(\pi||q) \neq KL(q||\pi)$$

For Gaussian distributions, the KL can be computed in closed form:

$$KL(\pi(\cdot|s)||q(\cdot|s)) = \frac{1}{2}[\left(\mu_q(s) - \mu_\pi(s)\right)^T \Sigma_q^{-1}(s)\left(\mu_q(s) - \mu_\pi(s)\right) + \log\frac{det\Sigma_q(s)}{det\Sigma_\pi(s)}$$
$$+tr\left(\Sigma_q^{-1}(s)\Sigma_\pi(s)\right) - d)]$$

where d is the dimensionality of the action space. The above entropy can be decomposed in three terms: mean, entropy and rotation terms.

2. Deep Actor-Critic Methods

The Deep Actor-Critic methods are Actor-Critic because they learn both an Actor (the policy) and a Critic (the value function). Deep, because they use Deep Neural Networks to model both the Actor and the Critic. These methods are approximate, and the gradient estimation will be biased. We replace the true objective J_θ with different surrogate losses L_θ that are easier to compute.

Before, we must clarify some nomenclature: while in TD learning the difference between on- and off-policy algorithms is whether the samples come from the actual policy or a behavioural policy, respectively, in deep actor-critic methods we use these terms differently.

On-policy methods updates the policy only with samples from the previous policy while *off-policy* methods use a replay buffer (note that as long as the policy does not change much, we still reuse old samples in on-policy methods). We broaden this distinction as with the TD definition, almost all deep actor-critic methods would be *on-policy.*

2.1. Surrogate Loss and the issue with the PGT

We start our discussion of deep actor-critic methods with the important *surrogate objective.* First, why do we need the surrogate objective in the first place?

The **first issue** is that the policy gradient theorem requires the **true** Q-function of the policy being optimized. This does not work well for off-policy training! Only one update is possible per Q-function estimate!

The **second issue** is the policy gradient theorem is an expectation under the discounted state distribution induced by the policy:

$$\nabla_\theta J_\theta = \frac{1}{1-\gamma} E_{(s,a)\sim d^{\pi_\theta},\pi_\theta}[\nabla_\theta \log \pi_\theta(a\,|s\,)Q^{\pi_\theta}(s,a)]$$
$$d^{\pi_\theta}(s) = (1-\gamma)\sum_{t=0}^{\infty}\gamma^t P(s_t = s|s_0,\pi)$$

at each step, terminate the trajectory with probability $1 - \gamma$. This is not a very efficient usage of environment samples!

A possible solution is to use the **undiscounted policy distribution** for the expectation but **discount the gradient at every step**. This still has some drawbacks e.g., issues with long trajectories where the gradient w.r.t. states close to the end will be very small. It is also very difficult to implement this in off-policy settings. Deep Actor-Critic methods trade-off rigorous mathematical formulation for performance!

The original loss function is very complex to optimize, as the state distribution depends on the current policy. To derive a simplified loss, we proceed as follows: Let π_θ and q be two arbitrary policies. Using the *Performance Difference Lemma*, we can write $J(\pi_\theta)$ as:

$$J(\pi_\theta) = J(q) + E_{\tau\sim\pi_\theta.P}\left[\sum_{t=0}^{\infty}\gamma^t A^q(s_t, a_t)\right] = J(q) + E_{s\sim d^{\pi_\theta},a\sim\pi_\theta}[A^q(s, a)]$$

where d^{π_θ} is the discounted state distribution. This formulation is equivalent to the normal objective function, only with an additional baseline policy q. But it contains d^{π_θ} i.e., the (discounted) state distribution. Thus, it's difficult to compute the gradient of this objective!

Instead, we replace d^{π_θ} with the discounted state distribution w.r.t. q, d^q. These yields the surrogate objective:

$$L_q(\pi_\theta) = J(q) + E_{s\sim d^q,a\sim\pi_\theta}[A^q(s, a)]$$

Notice that if $q = \pi_\theta$ then $L_q(\pi_\theta) = J(\pi_\theta), \nabla_\theta L_q(\pi_\theta) = \nabla_\theta J(\pi_\theta)$

In practice We drop the term $J(q)$ as it does not depend on the policy parameters i.e., it is a constant term: $L_q(\pi_\theta) = E_{s\sim d^q,a\sim\pi_\theta}[A^q(s, a)]$.

Also, we approximate the advantage function $\hat{A}(s, a) \approx A(s, a)$.

We select q to be the previous policy π_{θ_k} and π_θ to be the policy we are currently optimizing, i.e., taking the gradient w.r.t. its parameter θ.

Additionally, most of the algorithms we discuss introduce another approximation by not using the discounted state distribution d^q but using the undiscounted one, u^q.

No vanishing gradient with long trajectories but wrong gradient estimate!

3. A2C Algorithm

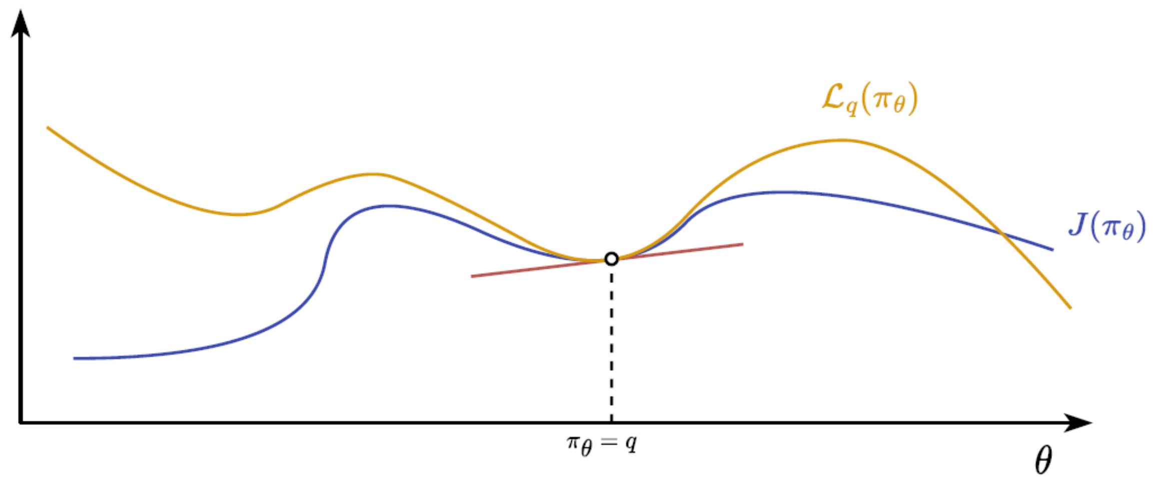
Advantage actor-critic (A2C) is the simplest deep actor-critic methods and just uses MC estimates (short trajectory chunks) of the value and advantage function ($V^{\pi_\theta}(s)$ and $A^{\pi_\theta}(s, a)$). Subsequently, it just follows the gradient of the surrogate loss:

$$\nabla_\theta L_\theta = E_{s\sim d^q}\left[\int_A \nabla_\theta \pi_\theta(a|s)\, \hat{A}(s, a) da\right]$$

4. Trust-region Methods

In this section we discuss a variety of on-policy methods which, in the deep actor-critic sense, are on-policy as they do not use a replay buffer. We will cover trust region methods!

The Surrogate Loss is close to the true objective only if we stay close to the sampling distribution as illustrated in the following picture:



The Solution is to limit policy updates using a Trust Region!

4.1. TRPO Algorithm

The Trust Region Policy Optimization (TRPO) algorithm is defined as the following optimization:

$$\theta^* = argmax_\theta L(\theta)$$
$$s.t. E_{s\sim u^q}[KL(\pi(a|s)||q(s|a))] \leq \varepsilon$$

In this approach, we rewrite the surrogate loss using *importance sampling*:

$$L(\theta) = E_{s,a\sim q}\left[\frac{\pi_\theta(a|s)}{q(a|s)}\hat{A}(s,a)\right]$$

The TRPO algorithm Optimizes the policy in a "Trust Region" where we have info about the policy performance. Remember that q is the policy of the previous iteration and π_θ is the policy we are currently optimizing. With this approach, we get an *off-policy objective,* but *on-policy data.* The advantage is estimated using *generalized advantage estimation (GAE):*

$$\hat{A}^{GAE(\lambda)}(s_t, a_t) = \sum_{l=0}^{\infty}(\gamma\lambda)^l \delta_{t+l}^V$$

with the TD error $\delta_{t+l}^t = r_t + \gamma V(s_{t+1}) - V(s_t)$. To implement TRPO, we perform a few simplifications to make the problem tractable, assuming $\pi_\theta \approx q$:

1. Approximate the KL divergence with the Fisher information matrix $\boldsymbol{F}_\theta$:

$$KL(\pi(a|s)||q(a|s)) \approx \delta$$

2. Compute the natural gradient as $\nabla^{NG} L_\theta = \boldsymbol{F}_\theta^{-1}\nabla_\theta L_\theta$

3. Use line search to find the optimal step size along the natural gradient direction

5. Wrap-up

- Why it is difficult to use the PGT in practical approaches.
- How Deep RL simplifies the problem using surrogate losses.
- The On-policy approaches that use samples coming from the current policy:
  - A2C, the simplest extension
  - TRPO, using a trust-region to optimize the policy
  - PPO, a simpler way to implement a trust region approach
- How to use Conjugate Gradient algorithm to compute the natural gradient for large Neural Networks.

6. Questions

Write down the formula for the Entropy and Relative Entropy and explain them!

The Entropy of a policy π in a state s is defined as:

$$H(\pi(\cdot|s)) = -\int_A \pi(a|s)\log\pi(a|s)\,da = E_{a\sim\pi}[\log\pi(a|s)]$$

This quantity measures the amount of "randomness" of the policy

The Relative Entropy or Kullback-Leibler divergence (KL) between two policies π and q in a state s is:

$$KL(\pi(\cdot|s)||q(\cdot|s)) = \int_A \pi(a|s)\log\frac{\pi(a|s)}{q(a|s)}da$$

The KL measures the "distance" between the probability density of the actions at a given state s. Remember that the KL is not a proper distance:

$$KL(\pi||q) \neq KL(q||\pi)$$

What are the difficulties of using PGT in practice?

The **first issue** is that the policy gradient theorem requires the **true** Q-function of the policy being optimized. This does not work well for off-policy training! Only one update is possible per Q-function estimate!

The **second issue** is the policy gradient theorem is an expectation under the discounted state distribution induced by the policy:

$$\nabla_\theta J_\theta = \frac{1}{1-\gamma}E_{(s,a)\sim d^{\pi_\theta},\pi_\theta}[\nabla_\theta \log\pi_\theta(a\,|s\,)Q^{\pi_\theta}(s,a)]$$

$$d^{\pi_\theta}(s) = (1-\gamma)\sum_{t=0}^{\infty} \gamma^t P(s_t = s | s_0, \pi)$$

at each step, terminate the trajectory with probability $1 - \gamma$. This is not a very efficient usage of environment samples!

How does deep RL simplify the problem using surrogate objectives?

We replace d^{π_θ} with the discounted state distribution w.r.t. q, d^q. These yields the surrogate objective:

$$L_q(\pi_\theta) = J(q) + E_{s\sim d^q, a\sim\pi_\theta}[A^q(s, a)]$$

In practice We drop the term $J(q)$ as it does not depend on the policy parameters i.e., it is a constant term:

$$L_q(\pi_\theta) = E_{s\sim d^q, a\sim\pi_\theta}[A^q(s, a)]$$

What are the three big on-policy approaches we discussed and what are their core features?

How can we compute the NG for large NNs?

How do we use Conjugate Gradient algorithm to compute the natural gradient for large Neural Networks?

Model-Based RL & Intrinsic motivation

All this course covered *model-free* RL (except for DP which uses a known model). In *model-based RL (MBRL)*, we learn a model of the environment and its transitions. We can then use this model to efficiently solve the task. To apply planning methods, we still must learn the model though as listed here:

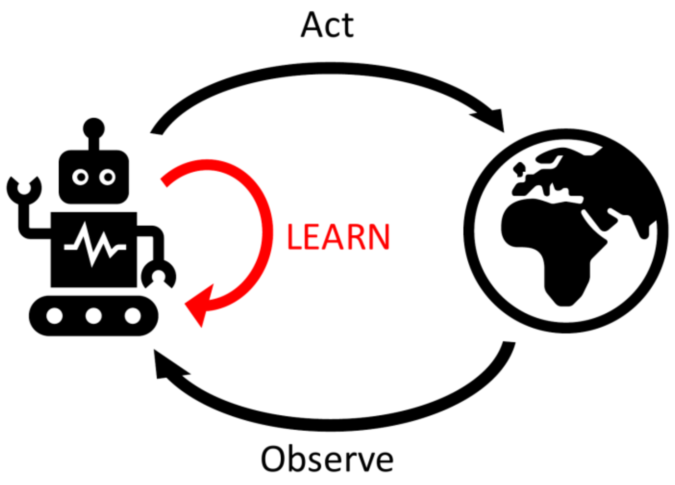
1. Learn models from samples: Reward function and transition dynamics
2. Use learned models to plan actions

But why even use MBRL? A model enables you to plan (and therefore you can add additional constraints, penalize uncertainty, etc.). MBRL is sample efficient and ensures transferability as well as generality. Therefore, a model can be reused for achieving different tasks. In the next section we will cover how to even learn models.

1. Learning models to plan

Let’s start with Craik famous quote: “If the organism carries a ‘small-scale model’ of external reality and of its own possible actions within its head, it is able to try out various alternatives, conclude which is the best of them, react to future situations before they arise, utilise the knowledge of past events in dealing with the present and future, and in every way to react in a much fuller, safer, and more competent manner to the emergencies which face it.”

Now consider a MDP $M =< S, A, R, P, \iota, \gamma >$:



What is even a model? A model is a representation that explicitly encodes knowledge about the structure of the environment and task. But what knowledge to explicitly encode? Here are the most common used models:

- Transition/dynamics model: $s' = f_s(s, a)$
- Reward model: $r = f_r(s, a)$
- Inverse Dynamics Model: $a = f_s^{-1}(s, s')$
- Distance model: $d_{ij} = f_d(s_i . s_j)$

And what do we learn now in MBRL? A full model $\widehat{M}$ consists of an approximated reward function $\hat{R} \approx R$ and approximated transition dynamics $\hat{P} \approx P$. Models can be either deterministic or stochastic.

After learning the
 it is used to search through the state space to find an optimal policy. This is also called planning.

1.1. Dyna-Q

What happens when learning the model online, i.e., while planning? The model can change and affect the planned actions. One common Algorithm for that is called Dyna-Q which combines model-free Q-learning with model-based planning as shown in Algorithm 1.

Algorithm 1: Dyna Q

```
    Input: Environment, Model $\widehat{M}: \hat{R}, \hat{P}$
    Output: Optimal Policy and Q-function
 1: Initialize $Q(s,a)$ and model $\widehat{M}$ for all $s \in S, a \in A$
 2: repeat
 3:     $s \leftarrow$ current non absorbing state
 4:     $a \sim \varepsilon$-greedy using $Q$
 5:     Take action $a$, observe reward $r$ and state $s'$
 6:     $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \max_{a'} Q(s',a') - Q(s,a)]$
        // Store which pairs you visited to replay them when doing the update of the Q
 7:     $\widehat{M}(s,a) \leftarrow r, s'$ (assuming deterministic environment)
 8:     for $n = 1, \dots, N$ do:
 9:         $s \leftarrow$ random previously observed state
10:         $a \leftarrow$ random action previously taken in state $s$
11:         $r, s' \leftarrow \widehat{M}(s,a)$
12:         $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \max_{a'} Q(s',a') - Q(s,a)]$
13: until convergence
14: return $Q^*(s,a)$
```

1.2. Model Based Policy Optimization

But in Dyna-Q we have assumed a deterministic environment. The model of a transition can be learned by visiting it just once! What about stochastic models? We have two kinds of uncertainties:

- *Aleatoric:* Uncertainty of the output given an input, i.e., stochastic environment
- *Epistemic:* Uncertainty in the accuracy of the model, due to inaccurate approximation of the model.

Solutions for these uncertainties are to use a probabilistic model to capture aleatoric uncertainty and improve the accuracy of the model to account for epistemic uncertainty.

2. Intrinsic Motivation

There are two key Exploration Problems in Deep RL:

- The "Hard-Exploration" Problem: Exploration in environments with very sparse (limited feedback (rewards) from the environment) or even deceptive rewards. Random exploration is prone to failure as it will rarely find successful states or obtain meaningful feedback from the environment. Montezuma's Revenge is one of such examples.
- The Noisy-TV Problem: Initially proposed by Burda et al. where he states that an agent seeks for novelty in the environment and finds a TV with uncontrollable & unpredictable output (e.g., Gaussian noise). This will attract the agent's attention forever! The agent obtains new rewards from the noisy TV but fails to make any meaningful progress and becomes a "couch potato"!

To solve these exploration problems, we can instead of only providing an extrinsic task-specific $r^e(s,a)$ provide an additional signal (intrinsic) $r^i(s,a)$ that drives (motivates) the agent's exploration:

$$r(s,a) = r^e(s,a) + \beta r^i(s,a)$$

where β is a hyperparameter that balances exploitation and exploration. Intrinsic rewards act as exploration bonuses. The intrinsic reward is/can be inspired by intrinsic motivation and
we can transfer those findings to RL too: Discovery of novel (unfamiliar, new) states and improvement of the agent's knowledge about the environment. Let's look now at some methods for this.

2.1. Count-based exploration

Some problems are difficult to be solved e.g., if we have sparse rewards, complex dynamics, and a high probability of failures. The idea of count-based exploration is to explore states the agent has visited less and where the agent is the most uncertain. To do this, we are using the approach of *Optimism in the face of uncertainty (OFU)*. This is a heuristic approach of assigning optimistic values to uncertain or unexplored states or actions. It encourages exploration and learning by assuming that unexplored options have a higher potential for positive rewards, driving the agent to actively seek out new strategies and gather more information to maximize long-term gains. It can be formally studied for providing approximately optimal decision making (see bandits).

Add an intrinsic reward:

$$r^i(s,a) = \beta(N_n(s) + 0.01)^{-\frac{1}{2}} = \frac{\beta}{\sqrt{N_n(s) + 0.01}}$$

To the extrinsic reward $r^e(s,a)$. The total reward is then:

$$r^{tot}(s,a) = r^e(s,a) + r^i(s,a)$$

Counting states is possible in discrete state spaces but it is not in continuous state spaces.
Still, we can consider some states similar to others! We can then use a pseudo-count that computes a count for similar states. Pseudo-counts are typically computed with unsupervised learning methods:

- clustering
- kernel density estimation
- autoencoders

2.2. Curiosity-driven exploration

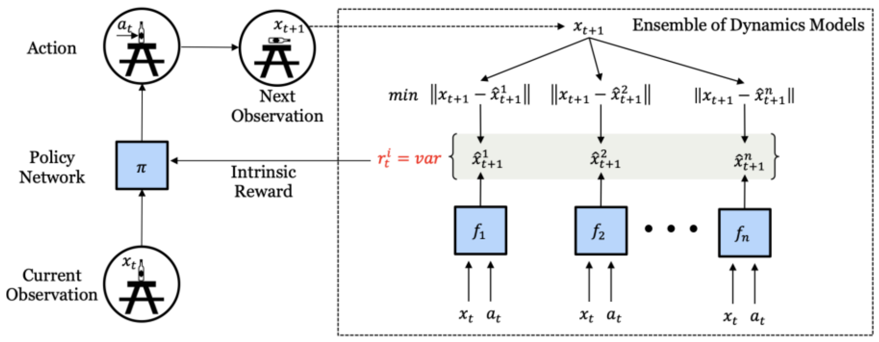
Optimism in the face of uncertainty can be pursued in other ways instead of just exploring states the agent has visited less and where the agent is the most uncertain. An intrinsic reward can have different forms. In curiosity-driven exploration, the difference between the (features of) predicted next state and the actual next state is computed. The greater the difference, the more interesting the next state is. Thus, the intrinsic reward is:

$$r^i(s,a) = \frac{\eta}{2}\left|\left|\hat{\phi}(s') - \phi(s')\right|\right|_2^2$$

2.3. Self-Supervised Exploration via Disagreement

Use an ensemble of prediction models and use their disagreement as bonus:

- High disagreement → low confidence → needs more exploration
- r^i is differentiable → intrinsic reward can be directly optimized
- very efficient differentiable approach



2.4. More Methods for Driving Exploration

- Information gain, based on some measure of uncertainty
- Physics-aware exploration (e.g., for robotics)
- Competence measures
- Distillation and use of Random Networks (DORA the explorer)
- Distributional methods

3. Wrap-up

Now you know:

- Now you know how to learn transition models of MDPs
- You know how to use models to perform sample-efficient policy learning
- You know how to use the model to plan actions
- How to use uncertainty during model-based planning
- The idea behind intrinsic motivation
- Different approaches, both exploiting state visitation, and model-based ones to drive the agent’s motivation to learn

4. Questions