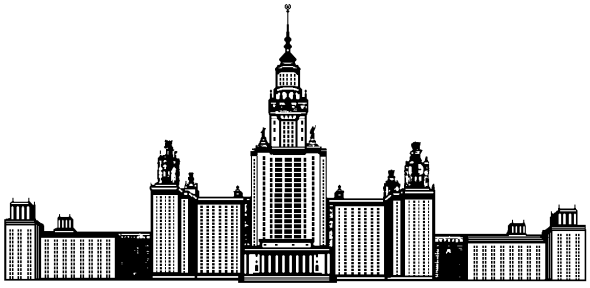

Moscow State University
Faculty of Computational Mathematics and Cybernetics
Department of Mathematical Methods of Forecasting

Modern Deep Reinforcement Learning Algorithms

Made by:
student of 517 group
Sergey Ivanov

Scientific advisor:
Alexander D'yakonov

Moscow, 2019

Contents

1 Introduction **4**

2 Reinforcement Learning problem setup **5**
2.1 Assumptions of RL setting . . . 5
2.2 Environment model . . . 6
2.3 Objective . . . 6
2.4 Value functions . . . 8
2.5 Classes of algorithms . . . 9
2.6 Measurements of performance . . . 9

3 Value-based algorithms **10**
3.1 Temporal Difference learning . . . 10
3.2 Deep Q-learning (DQN) . . . 12
3.3 Double DQN . . . 14
3.4 Dueling DQN . . . 15
3.5 Noisy DQN . . . 16
3.6 Prioritized experience replay . . . 17
3.7 Multi-step DQN . . . 18

4 Distributional approach for value-based methods **20**
4.1 Theoretical foundations . . . 20
4.2 Categorical DQN . . . 22
4.3 Quantile Regression DQN (QR-DQN) . . . 24
4.4 Rainbow DQN . . . 27

5 Policy Gradient algorithms **29**
5.1 Policy Gradient theorem . . . 29
5.2 REINFORCE . . . 30
5.3 Advantage Actor-Critic (A2C) . . . 31
5.4 Generalized Advantage Estimation (GAE) . . . 33
5.5 Natural Policy Gradient (NPG) . . . 34
5.6 Trust-Region Policy Optimization (TRPO) . . . 36
5.7 Proximal Policy Optimization (PPO) . . . 39

6 Experiments **41**
6.1 Setup . . . 41
6.2 Cartpole . . . 41
6.3 Pong . . . 42
6.4 Interaction-training trade-off in value-based algorithms . . . 43
6.5 Results . . . 44

7 Discussion **47**

A Implementation details **50**

B Hyperparameters **51**

C Training statistics on Pong **52**

D Playing Pong behaviour **54**

Abstract

Recent advances in Reinforcement Learning, grounded on combining classical theoretical results with Deep Learning paradigm, led to breakthroughs in many artificial intelligence tasks and gave birth to Deep Reinforcement Learning (DRL) as a field of research. In this work latest DRL algorithms are reviewed with a focus on their theoretical justification, practical limitations and observed empirical properties.

1. Introduction

During the last several years Deep Reinforcement Learning proved to be a fruitful approach to many artificial intelligence tasks of diverse domains. Breakthrough achievements include reaching human-level performance in such complex games as Go [20], multiplayer Dota [14] and real-time strategy StarCraft II [24]. The generality of DRL framework allows its application in both discrete and continuous domains to solve tasks in robotics and simulated environments [12].

Reinforcement Learning (RL) is usually viewed as general formalization of decision-making task and is deeply connected to dynamic programming, optimal control and game theory. [21] Yet its problem setting makes almost no assumptions about world model or its structure and usually supposes that environment is given to agent in a form of black-box. This allows to apply RL practically in all settings and forces designed algorithms to be adaptive to many kinds of challenges. Latest RL algorithms are usually reported to be transferable from one task to another with no task-specific changes and little to no hyperparameters tuning.

As an object of desire is a strategy, i. e. a function mapping agent's observations to possible actions, reinforcement learning is considered to be a subfiled of machine learning. But instead of learning from data, as it is established in classical supervised and unsupervised learning problems, the agent learns from experience of interacting with environment. Being more "natural" model of learning, this setting causes new challenges, peculiar only to reinforcement learning, such as necessity of exploration integration and the problem of delayed and sparse rewards. The full setup and essential notation are introduced in section 2.

Classical Reinforcement Learning research in the last third of previous century developed an extensive theoretical core for modern algorithms to ground on. Several algorithms are known ever since and are able to solve small-scale problems when either environment states can be enumerated (and stored in the memory) or optimal policy can be searched in the space of linear or quadratic functions of state representation features. Although these restrictions are extremely limiting, foundations of classical RL theory underlie modern approaches. These theoretical fundamentals are discussed in sections 3.1 and 5.1–5.2.

Combining this framework with Deep Learning [5] was popularized by Deep Q-Learning algorithm, introduced in [13], which was able to play any of 57 Atari console games without tweaking network architecture or algorithm hyperparameters. This novel approach was extensively researched and significantly improved in the following years. The principles of value-based direction in deep reinforcement learning are presented in section 3.

One of the key ideas in the recent value-based DRL research is distributional approach, proposed in [1]. Further extending classical theoretical foundations and coming with practical DRL algorithms, it gave birth to distributional reinforcement learning paradigm, which potential is now being actively investigated. Its ideas are described in section 4.

Second main direction of DRL research is policy gradient methods, which attempt to directly optimize the objective function, explicitly present in the problem setup. Their application to neural networks involve a series of particular obstacles, which requested specialized optimization techniques. Today they represent a competitive and scalable approach in deep reinforcement learning due to their enormous parallelization potential and continuous domain applicability. Policy gradient methods are discussed in section 5.

Despite the wide range of successes, current state-of-art DRL methods still face a number of significant drawbacks. As training of neural networks requires huge amounts of data, DRL demonstrates unsatisfying results in settings where data generation is expensive. Even in cases where interaction is nearly free (e. g. in simulated environments), DRL algorithms tend to require excessive amounts of iterations, which raise their computational and wall-clock time cost. Furthermore, DRL suffers from random initialization and hyperparameters sensitivity, and its optimization process is known to be uncomfortably unstable [9]. Especially embarrassing consequence of these DRL features turned out to be low reproducibility of empirical observations from different research groups [6]. In section 6, we attempt to launch state-of-art DRL algorithms on several standard testbed environments and discuss practical nuances of their application.

2. Reinforcement Learning problem setup

2.1. Assumptions of RL setting

Informally, the process of sequential decision-making proceeds as follows. The ***agent*** is provided with some initial observation of environment and is required to choose some action from the given set of possibilities. The ***environment*** responds by transitioning to another state and generating a ***reward signal*** (scalar number), which is considered to be a ground-truth estimation of agent's performance. The process continues repeatedly with agent making choices of actions from observations and environment responding with next states and reward signals. The only goal of agent is to maximize the cumulative reward.

This description of learning process model already introduces several key assumptions. Firstly, the time space is considered to be discrete, as agent interacts with environment sequentially. Secondly, it is assumed that provided environment incorporates some reward function as supervised indicator of success. This is an embodiment of the ***reward hypothesis***, also referred to as ***Reinforcement Learning hypothesis***:

Proposition 1. ***(Reward Hypothesis)*** *[21]*
«All of what we mean by goals and purposes can be well thought of as maximization of the expected value of the cumulative sum of a received scalar signal (reward).»

Exploitation of this hypothesis draws a line between reinforcement learning and classical machine learning settings, supervised and unsupervised learning. Unlike unsupervised learning, RL assumes supervision, which, similar to labels in data for supervised learning, has a stochastic nature and represents a key source of knowledge. At the same time, no data or «right answer» is provided to training procedure, which distinguishes RL from standard supervised learning. Moreover, RL is the only machine learning task providing explicit objective function (cumulative reward signal) to maximize, while in supervised and unsupervised setting optimized loss function is usually constructed by engineer and is not «included» in data. The fact that reward signal is incorporated in the environment is considered to be one of the weakest points of RL paradigm, as for many real-life human goals introduction of this scalar reward signal is at the very least unobvious.

For practical applications it is also natural to assume that agent's observations can be represented by some feature vectors, i. e. elements of $\mathbb{R}^{\boldsymbol{d}}$. The set of possible actions in most practical applications is usually uncomplicated and is either discrete (number of possible actions is finite) or can be represented as subset of $\mathbb{R}^{\boldsymbol{m}}$ (almost always $[-\mathbf{1}, \mathbf{1}]^{\boldsymbol{m}}$ or can be reduced to this case)[1]. RL algorithms are usually restricted to these two cases, but the mix of two (agent is required to choose both discrete and continuous quantities) can also be considered.

The final assumption of RL paradigm is a ***Markovian property***:

Proposition 2. ***(Markovian property)***
Transitions depend solely on previous state and the last chosen action and are independent of all previous interaction history.

Although this assumption may seem overly strong, it actually formalizes the fact that the world modeled by considered environment obeys some general laws. Giving that the agent knows the current state of the world and the laws, it is assumed that it is able to predict the consequences of his actions up to the internal stochasticity of these laws. In practice, both laws and complete state representation is unavailable to agent, which limits its forecasting capability.

In the sequel we will work within the setting with one more assumption of ***full observability***. This simplification supposes that agent can observe complete world state, while in many real-life tasks only a part of observations is actually available. This restriction of RL theory can be removed by considering ***Partially observable Markov Decision Processes (PoMDP)***, which basically forces learning algorithms to have some kind of memory mechanism to store previously received observations. Further on we will stick to fully observable case.

[1] this set is considered to be permanent for all states of environment without any loss of generality as if agent chooses invalid action the world may remain in the same state with zero or negative reward signal or stochastically select some valid action for him.

2.2. Environment model

Though the definition of ***Markov Decision Process (MDP)*** varies from source to source, its essential meaning remains the same. The definition below utilizes several simplifications without loss of generality.[2]

Definition 1. ***Markov Decision Process*** (MDP) is a tuple $(\mathcal{S}, \mathcal{A}, \mathbb{T}, r, s_0)$, where:

- $\mathcal{S} \subseteq \mathbb{R}^d$ — arbitrary set, called the ***state space***.
- $\mathcal{A}$ — a set, called the ***action space***, either
  - ***discrete***: $|\mathcal{A}| < +\infty$, or
  - ***continuous domain***: $\mathcal{A} = [-1, 1]^m$.
- $\mathbb{T}$ — ***transition probability*** $p(s' \mid s, a)$, where $s, s' \in \mathcal{S}, a \in \mathcal{A}$.
- $r : \mathcal{S} \to \mathbb{R}$ — ***reward function***.
- $s_0 \in \mathcal{S}$ — starting state.

It is important to notice that in the most general case the only things available for RL algorithm beforehand are d (dimension of state space) and action space $\mathcal{A}$. The only possible way of collecting more information for agent is to interact with provided environment and observe s_0. It is obvious that the first choice of action a_0 will be probably random. While the environment responds by sampling $s_1 \sim p(s_1 \mid s_0, a_0)$, this distribution, defined in $\mathbb{T}$ and considered to be a part of MDP, may be unavailable to agent's learning procedure. What agent does observe is s_1 and reward signal $r_1 := r(s_1)$ and it is the key information gathered by agent from interaction experience.

Definition 2. The tuple $(s_t, a_t, r_{t+1}, s_{t+1})$ is called **transition** . Several sequential transitions are usually referred to as ***roll-out***. Full track of observed quantities

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, a_3 \dots$$

is called a ***trajectory***.

In general case, the trajectory is infinite which means that the interaction process is neverending. However, in most practical cases the ***episodic property*** holds, which basically means that the interaction will eventually come to some sort of an end[3]. Formally, it can be simulated by the environment stucking in the last state with zero probability of transitioning to any other state and zero reward signal. Then it is convenient to reset the environment back to s_0 to initiate new interaction. One such interaction cycle from s_0 till reset, spawning one trajectory of some finite length T, is called an ***episode***. Without loss of generality, it can be considered that there exists a set of ***terminal states*** $\mathcal{S}^+$, which mark the ends of interactions. By convention, transitions $(s_t, a_t, r_{t+1}, s_{t+1})$ are accompanied with binary flag $\mathbf{done}_{t+1} \in \{0, 1\}$, whether s_{t+1} belongs to $\mathcal{S}^+$. As timestep t at which the transition was gathered is usually of no importance, transitions are often denoted as $(s, a, r', s', \mathbf{done})$ with primes marking the «next timestep».

Note that the length of episode T may vary between different interactions, but the episodic property holds if interaction is guaranteed to end after some finite time $T^{\max}$. If this is not the case, the task is called ***continuing***.

2.3. Objective

In reinforcement learning, the agent's goal is to maximize a cumulative reward. In episodic case, this reward can be expressed as a summation of all received reward signals during one episode and

[2] the reward function is often introduced as stochastic and dependent on action a, i. e. $R(r \mid s, a) : \mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathbb{R})$, while instead of fixed s_0 a distribution over $\mathcal{S}$ is given. Both extensions can be taken into account in terms of presented definition by extending the state space and incorporating all the uncertainty into transition probability $\mathbb{T}$.

[3] natural examples include the end of the game or agent's failure/success in completing some task.

is called ***the return***:

$$R \coloneqq \sum_{t=1}^{T} r_t \tag{1}$$

Note that this quantity is formally a random variable, which depends on agent's choices and the outcomes of environment transitions. As this stochasticity is an inevitable part of interaction process, the underlying distribution from which r_t is sampled must be properly introduced to set rigorously the task of return maximization.

Definition 3. Agent's algorithm for choosing a by given current state s, which in general can be viewed as distribution $\pi(a \mid s)$ on domain $\mathcal{A}$, is called a ***policy*** (strategy).

Deterministic policy, when the policy is represented by deterministic function $\pi\colon \mathcal{S} \to \mathcal{A}$, can be viewed as a particular case of ***stochastic policy*** with degenerated policy $\pi(a \mid s)$, when agent's output is still a distribution with zero probability to choose an action other than $\pi(s)$. In both cases it is considered that agent sends to environment a sample $a \sim \pi(a \mid s)$.

Note that given some policy $\pi(a \mid s)$ and transition probabilities $\mathbb{T}$, the complete interaction process becomes defined from probabilistic point of view:

Definition 4. For given MDP and policy π, the probability of observing

$$s_0, a_0, s_1, a_1, s_2, a_2 \dots$$

is called ***trajectory distribution*** and is denoted as $\mathcal{T}_\pi$:

$$\mathcal{T}_\pi \coloneqq \prod_{t=0} p(s_{t+1} \mid s_t, a_t)\pi(a_t \mid s_t)$$

It is always substantial to keep track of what policy was used to collect certain transitions (roll-outs and episodes) during the learning procedure, as they are essentially samples from corresponding trajectory distribution. If the policy is modified in any way, the trajectory distribution changes either.

Now when a policy induces a trajectory distribution, it is possible to formulate a task of ***expected reward*** maximization:

$$\mathbb{E}_{\mathcal{T}_\pi} \sum_{t=1}^{T} r_t \to \max_\pi$$

To ensure the finiteness of this expectation and avoid the case when agent is allowed to gather infinite reward, limit on absolute value of r_t can be assumed:

$$|r_t| \leq R^{\max}$$

Together with the limit on episode length $T^{\max}$ this restriction guarantees finiteness of optimal (maximal) expected reward.

To extend this intuition to continuing tasks, the reward for each next interaction step is multiplied on some discount coefficient $\gamma \in [0, 1)$, which is often introduced as part of MDP. This corresponds to the logic that with probability $1 - \gamma$ agent "dies" and does not gain any additional reward, which models the paradigm «better now than later». In practice, this discount factor is set very close to 1.

Definition 5. For given MDP and policy π the ***discounted expected reward*** is defined as

$$J(\pi) \coloneqq \mathbb{E}_{\mathcal{T}_\pi} \sum_{t=0} \gamma^t r_{t+1}$$

Reinforcement learning task is to find an ***optimal policy*** π^*, which maximizes the discounted expected reward:

$$J(\pi) \to \max_\pi \tag{2}$$

2.4. Value functions

Solving reinforcement learning task (2) usually leads to a policy, that maximizes the expected reward not only for starting state s_0, but for any state $s \in \mathcal{S}$. This follows from the Markov property: the reward which is yet to be collected from some step t does not depend on previous history and for agent staying at state s the task of behaving optimal is equivalent to maximization of expected reward with current state s as a starting state. This is the particular reason why many reinforcement learning algorithms do not seek only optimal policy, but additional information about usefulness of each state.

Definition 6. For given MDP and policy π the ***value function under policy*** π is defined as

$$V^{\pi}(s) \coloneqq \mathbb{E}_{\mathcal{T}_{\pi}|s_0=s} \sum_{t=0} \gamma^t r_{t+1}$$

This value function estimates how good it is for agent utilizing strategy π to visit state s and generalizes the notion of discounted expected reward $J(\pi)$ that corresponds to $V^{\pi}(s_0)$.

As value function can be induced by any policy, value function $V^{\pi^*}(s)$ under optimal policy π^* can also be considered. By convention[4], it is denoted as $V^*(s)$ and is called an ***optimal value function***.

Obtaining optimal value function $V^*(s)$ doesn't provide enough information to reconstruct some optimal policy π^* due to unknown world dynamics, i. e. transition probabilities. In other words, being blind to what state s may be the environment's response on certain action in a given state makes knowing optimal value function unhelpful. This intuition suggests to introduce a similar notion comprising more information:

Definition 7. For given MDP and policy π the ***quality function (Q-function) under policy*** π is defined as

$$Q^{\pi}(s, a) \coloneqq \mathbb{E}_{\mathcal{T}_{\pi}|s_0=s, a_0=a} \sum_{t=0} \gamma^t r_{t+1}$$

It directly follows from the definitions that these two functions are deeply interconnected:

$$Q^{\pi}(s, a) = \mathbb{E}_{s' \sim p(s'|s,a)} \left[r(s') + \gamma V^{\pi}(s') \right] \tag{3}$$

$$V^{\pi}(s) = \mathbb{E}_{a \sim \pi(a|s)} Q^{\pi}(s, a) \tag{4}$$

The notion of ***optimal Q-function*** $Q^*(s, a)$ can be introduced analogically. But, unlike value function, obtaining $Q^*(s, a)$ actually means solving a reinforcement learning task: indeed,

Proposition 3. *If $Q^*(s, a)$ is a quality function under some optimal policy, then*

$$\pi^*(s) = \underset{a}{\boldsymbol{argmax}}\, Q^*(s, a)$$

is an optimal policy.

This result implies that instead of searching for optimal policy π^*, an agent can search for optimal Q-function and derive the policy from it.

Proposition 4. *For any MDP existence of optimal policy leads to existence of deterministic optimal policy.*

[4] though optimal policy may not be unique, the value functions under any optimal policy that behaves optimally from any given state (not only s_0) coincide. Yet, optimal policy may not know optimal behaviour for some states if it knows how to avoid them with probability 1.

2.5. Classes of algorithms

Reinforcement learning algorithms are presented in a form of computational procedures specifying a strategy of collecting interaction experience and obtaining a policy with as higher $J(\pi)$ as possible. They rarely include a stopping criterion like in classic optimization methods as the stochasticity of given setting prevents any reasonable verification of optimality; usually the number of iterations to perform is determined by the amount of computational resources. All reinforcement learning algorithms can be roughly divided into four[5] classes:

- ***meta-heuristics***: this class of algorithms treats the task as black-box optimization with zeroth-order oracle. They usually generate a set of policies $\pi_1 \dots \pi_P$ and launch several episodes of interaction for each to determine best and worst policies according to average return. After that they try to construct more optimal policies using evolutionary or advanced random search techniques [15].
- ***policy gradient***: these algorithms directly optimize (2), trying to obtain π^* and no additional information about MDP, using approximate estimations of gradient with respect to policy parameters. They consider RL task as an optimization with stochastic first-order oracle and make use of interaction structure to lower the variance of gradient estimations. They will be discussed in sec. 5.
- ***value-based*** algorithms construct optimal policy implicitly by gaining an approximation of optimal Q-function $Q^*(s, a)$ using dynamic programming. In DRL, Q-function is represented with neural network and an approximate dynamic programming is performed using reduction to supervised learning. This framework will be discussed in sec. 3 and 4.
- ***model-based*** algorithms exploit learned or given world dynamics, i. e. distributions $p(s' \mid s, a)$ from $\mathbb{T}$. The class of algorithms to work with when the model is explicitly provided is represented by such algorithms as Monte-Carlo Tree Search; if not, it is possible to imitate the world dynamics by learning the outputs of black box from interaction experience [10].

2.6. Measurements of performance

Achieved performance (***score***) from the point of average cumulative reward is not the only one measure of RL algorithm quality. When speaking of real-life robots, the required number of simulated episodes is always the biggest concern. It is usually measured in terms of interaction steps (where step is one transition performed by environment) and is referred to as ***sample efficiency***.

When the simulation is more or less cheap, RL algorithms can be viewed as a special kind of optimization procedures. In this case, the final performance of the found policy is opposed to required computational resources, measured by ***wall-clock time***. In most cases RL algorithms can be expected to find better policy after more iterations, but the amount of these iterations tend to be unjustified.

The ratio between amount of interactions and required wall-clock time for one update of policy varies significantly for different algorithms. It is well-known that model-based algorithms tend to have the greatest sample-efficiency at the cost of expensive update iterations, while evolutionary algorithms require excessive amounts of interactions while providing massive resources for parallelization and reduction of wall-clock time. Value-based and policy gradient algorithms, which will be the focus of our further discussion, are known to lie somewhere in between.

[5] in many sources evolutionary algorithms are bypassed in discussion as they do not utilize the structure of RL task in any way.

3. Value-based algorithms

3.1. Temporal Difference learning

In this section we consider temporal difference learning algorithm [21, Chapter 6], which is a classical Reinforcement Learning method in the base of modern value-based approach in DRL.

The first idea behind this algorithm is to search for optimal Q-function $Q^*(s,a)$ by solving a system of recursive equations which can be derived by recalling interconnection between Q-function and value function (3):

$$\begin{aligned} Q^\pi(s,a) = \mathbb{E}_{s'\sim p(s'|s,a)}\left[r(s') + \gamma V^\pi(s')\right] = \\ = \{\text{using (4)}\} = \mathbb{E}_{s'\sim p(s'|s,a)}\left[r(s') + \gamma \mathbb{E}_{a'\sim\pi(a'|s')} Q^\pi(s',a')\right] \end{aligned}$$

This equation, named ***Bellman equation***, remains true for value functions under any policies including optimal policy π^*:

$$Q^*(s,a) = \mathbb{E}_{s'\sim p(s'|s,a)}\left[r(s') + \gamma \mathbb{E}_{a'\sim\pi(a'|s')} Q^*(s',a')\right] \tag{5}$$

Recalling proposition 3, optimal (deterministic) policy can be represented as $\pi^*(s) = \underset{a}{argmax}\, Q^*(s,a)$. Substituting this for $\pi^*(s)$ in (5), we obtain fundamental ***Bellman optimality equation***:

Proposition 5. ***(Bellman optimality equation)***

$$Q^*(s,a) = \mathbb{E}_{s'\sim p(s'|s,a)}\left[r(s') + \gamma \max_{a'} Q^*(s',a')\right] \tag{6}$$

The straightforward utilization of this result is as follows. Consider the ***tabular case***, when both state space $\mathcal{S}$ and action space $\mathcal{A}$ are finite (and small enough to be listed in computer memory). Let us also assume for now that transition probabilities are available to training procedure. Then $Q^*(s,a) : \mathcal{S}\times\mathcal{A}\to\mathbb{R}$ can be represented as a finite table with $|\mathcal{S}||\mathcal{A}|$ numbers. In this case (6) just gives a set of $|\mathcal{S}||\mathcal{A}|$ equations for this table to satisfy.

Addressing the values of the table as unknown variables, this system of equations can be solved using basic ***point iteration*** method: let $Q_0^*(s,a)$ be initial arbitrary values of table (with the only exception that for terminal states $s\in\mathcal{S}^+$, if any, $Q_0^*(s,a) = 0$ for all actions a). On each iteration t the table is updated by substituting current values of the table to the right side of equation until the process converges:

$$Q_{t+1}^*(s,a) = \mathbb{E}_{s'\sim p(s'|s,a)}\left[r(s') + \gamma \max_{a'} Q_t^*(s',a')\right] \tag{7}$$

This straightforward approach of learning the optimal Q-function, named ***Q-learning***, has been extensively studied in classical Reinforcement Learning. One of the central results is presented in the following convergence theorem:

Proposition 6. *Let by $\mathcal{B}$ denote an operator $(\mathcal{S}\times\mathcal{A}\to\mathbb{R})\to(\mathcal{S}\times\mathcal{A}\to\mathbb{R})$, updating Q_t^* as in (7):*

$$Q_{t+1}^* = \mathcal{B}Q_t^*$$

for all state-action pairs s, a.

Then $\mathcal{B}$ is a **contraction mapping***, i. .e. for any two tables $Q_1, Q_2 \in (\mathcal{S}\times\mathcal{A}\to\mathbb{R})$*

$$\|\mathcal{B}Q_1 - \mathcal{B}Q_2\|_\infty \le \gamma\|Q_1 - Q_2\|_\infty$$

Therefore, there is a unique fixed point of the system of equations (7) and the point iteration method converges to it.

The contraction mapping property is actually of high importance. It demonstrates that the point iteration algorithm converges with exponential speed and requires small amount of iterations. As the true Q^* is a fixed point of (6), the algorithm is guaranteed to yield a correct answer. The trick is

that each iteration demands full pass across all state-action pairs and exact computation of expectations over transition probabilities.

In general case, these expectations can't be explicitly computed. Instead, agent is restricted to samples from transition probabilities gained during some interaction experience. **Temporal Difference** (TD)[6] algorithm proposes to collect this data using $\pi_t = \underset{a}{argmax}\, Q^*_t(s,a) \approx \pi^*$ and after each gathered transition $(s_t, a_t, r_{t+1}, s_{t+1})$ update only one cell of the table:

$$Q^*_{t+1}(s,a) = \begin{cases} (1-\alpha_t)Q^*_t(s,a) + \alpha_t \left[r_{t+1} + \gamma \max\limits_{a'} Q^*_t(s_{t+1}, a')\right] & \text{if } s = s_t, a = a_t \\ Q^*_t(s,a) & \text{else} \end{cases} \tag{8}$$

where $\alpha_t \in (0,1)$ plays the role of exponential smoothing parameter for estimating expectation $\mathbb{E}_{s' \sim p(s'|s_t,a_t)}(\cdot)$ from samples.

Two key ideas are introduced in the update formula (8): exponential smoothing instead of exact expectation computation and cell by cell updates instead of updating full table at once. Both are required to settle Q-learning algorithm for online application.

As the set $\mathcal{S}^+$ of terminal states in online setting is usually unknown beforehand, a slight modification of update (8) is used. If observed next state s' turns out to be terminal (recall the convention to denote this by flag $\mathbf{done}$), its value function is known to be equal to zero:

$$V^*(s') = \max_{a'} Q^*(s', a') = 0$$

This knowledge is embedded in the update rule (8) by multiplying $\max\limits_{a'} Q^*_t(s_{t+1}, a')$ on $(1 - \mathbf{done}_{t+1})$. For the sake of shortness, this factor is often omitted but should be always present in implementations.

Second important note about formula (8) is that it can be rewritten in the following equivalent way:

$$Q^*_{t+1}(s,a) = \begin{cases} Q^*_t(s,a) + \alpha_t \left[r_{t+1} + \gamma \max\limits_{a'} Q^*_t(s_{t+1}, a') - Q^*_t(s,a)\right] & \text{if } s = s_t, a = a_t \\ Q^*_t(s,a) & \text{else} \end{cases} \tag{9}$$

The expression in the brackets, referred to as ***temporal difference***, represents a difference between Q-value $Q^*_t(s,a)$ and its one-step approximation $r_{t+1} + \gamma \max\limits_{a'} Q^*_t(s_{t+1}, a')$, which must be zero in expectation for true optimal Q-function.

The idea of exponential smoothing allows us to formulate first practical algorithm which can work in the tabular case with unknown world dynamics:

Algorithm 1: Temporal Difference algorithm

Hyperparameters: $\alpha_t \in (0,1)$

Initialize $Q_*(s,a)$ arbitrary
On each interaction step:

1. select $a = \underset{a}{argmax}\, Q^*(s,a)$
2. observe transition $(s, a, r', s', \mathbf{done})$
3. update table:

$$Q^*(s,a) \leftarrow Q^*(s,a) + \alpha_t \left[r' + (1 - \mathbf{done})\gamma \max_{a'} Q^*(s', a') - Q^*(s,a)\right]$$

It turns out that under several assumptions on state visitation during interaction process this procedure holds similar properties in terms of convergence guarantees, which are stated by the following theorem:

[6] also known as TD(0) due to theoretical generalizations

Proposition 7. *[26] Let's define*

$$e_t(s,a) = \begin{cases} \alpha_t & (s,a) \text{ is updated on step } t \\ 0 & \text{otherwise} \end{cases}$$

Then if for every state-action pair (s,a)

$$\sum_t^{+\infty} e_t(s,a) = \infty \quad \sum_t^{+\infty} e_t(s,a)^2 < \infty$$

the algorithm 1 converges to optimal Q^* *with probability 1.*

This theorem states that basic policy iteration method can be actually applied online in the way proposed by TD algorithm, but demands «enough exploration» from the strategy of interacting with MDP during training. Satisfying this demand remains a unique and common problem of reinforcement learning.

The widespread kludge is ***ε-greedy strategy*** which basically suggests to choose random action instead of $a = \underset{a}{argmax}\, Q^*(s,a)$ with probability ε_t. The probability ε_t is usually set close to 1 during first interaction iterations and scheduled to decrease to a constant close to 0. This heuristic makes agent visit all states with non-zero probabilities independent of what current approximation $Q^*(s,a)$ suggests.

The main practical issue with Temporal Difference algorithm is that it requires table $Q^*(s,a)$ to be explicitly stored in memory, which is impossible for MDP with high state space complexity. This limitation substantially restricted its applicability until its combination with deep neural network was proposed.

3.2. Deep Q-learning (DQN)

Utilization of neural nets to model either a policy or a Q-function frees from constructing task-specific features and opens possibilities of applying RL algorithms to complex tasks, e. g. tasks with images as input. Video games are classical example of such tasks where raw pixels of screen are provided as state representation and, correspondingly, as input to either policy or Q-function.

Main idea of Deep Q-learning [13] is to adapt Temporal Difference algorithm so that update formula (9) would be equivalent to gradient descent step for training a neural network to solve a certain regression task. Indeed, it can be noticed that the exponential smoothing parameter α_t resembles learning rate of first-order gradient optimization procedures, while the exploration conditions from theorem 7 look identical to restrictions on learning rate of stochastic gradient descent.

The key hint is that (9) is actually a gradient descent step in the parameter space of the table functions family:

$$Q(s,a,\theta) = \theta^{s,a}$$

where all $\theta^{s,a}$ form a vector of parameters $\theta \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$.

To unravel this fact, it is convenient to introduce some notation from regression tasks. First, let's denote by y the **target** of our regression task, i. e. the quantity that our model is trying to predict:

$$y(s,a) := r(s') + \gamma \max_{a'} Q^*(s',a',\theta) \tag{10}$$

where s' is a sample from $p(s' \mid s,a)$ and s,a is input data. In this notation (9) is equivalent to:

$$\theta_{t+1} = \theta_t + \alpha_t \left[y(s,a) - Q^*(s,a,\theta_t) \right] e^{s,a}$$

where we multiplied scalar value $\alpha_t \left[y(s,a) - Q^*(s,a,\theta_t) \right]$ on the following vector $e^{s,a}$

$$e^{s,a}_{i,j} := \begin{cases} 1 & (i,j) = (s,a) \\ 0 & (i,j) \neq (s,a) \end{cases}$$

to formulate an update of only one component of θ in a vector form. By this we transitioned to update in parameter space using $Q^*(s,a) = \theta^{s,a}$. Notice that for table functions family the derivative

of $Q(s, a, \theta)$ by θ for given input s, a is its one-hot encoding, i. e. exactly $e^{s,a}$:

$$\frac{\partial Q^*(s, a, \theta)}{\partial \theta} = e^{s,a} \tag{11}$$

The statement now is that this formula is a gradient descent update for regression with input s, a, target $y(s, a)$ and MSE loss function:

$$\mathbf{Loss}(y(s, a), Q^*(s, a, \theta_t)) = (Q^*(s, a, \theta_t) - y(s, a))^2 \tag{12}$$

Indeed:

$$\theta_{t+1} = \theta_t + \alpha_t \left[y(s, a) - Q^*(s, a, \theta_t)\right] e^{s,a} =$$
$$\{12\} = \theta_t - \alpha_t \frac{\partial \mathbf{Loss}(y, Q^*(s, a, \theta_t))}{\partial Q} e^{s,a}$$
$$\{11\} = \theta_t - \alpha_t \frac{\partial \mathbf{Loss}(y, Q^*(s, a, \theta_t))}{\partial Q^*} \frac{\partial Q^*(s, a, \theta_t)}{\partial \theta} =$$
$$\{\text{chain rule}\} = \theta_t - \alpha_t \frac{\partial \mathbf{Loss}(y, Q^*(s, a, \theta_t))}{\partial \theta}$$

The obtained result is evidently a gradient descent step formula to minimize MSE loss function with target (10):

$$\theta_{t+1} = \theta_t - \alpha_t \frac{\partial \mathbf{Loss}(y, Q^*(s, a, \theta_t))}{\partial \theta} \tag{13}$$

It is important that dependence of y from θ is ignored during gradient computation (otherwise the chain rule application with y being dependent on θ is incorrect). On each step of temporal difference algorithm new target y is constructed using current Q-function approximation, and a new regression task with this target is set. For this fixed target one MSE optimization step is done according to (13), and on the next step a new regression task is defined. Though during each step the target is considered to represent some ground truth like it is in supervised learning, here it provides a direction of optimization and because of this reason is sometimes called a ***guess***.

Notice that representation (13) is equivalent to standard TD update (9) with all theoretical results remaining while the parametric family Q_θ is a table functions family. At the same time, (13) can be formally applied to any parametric function family including neural networks. It must be taken into account that this transition is not rigorous and all theoretical guarantees provided by theorem 7 are lost at this moment.

Further on we assume that optimal Q-function is approximated with neural network $Q^*_\theta(s, a)$ with parameters θ. Note that for discrete action space case this network may take only s as input and output $|\mathcal{A}|$ numbers representing $Q^*_\theta(s, a_1) \dots Q^*_\theta(s, a_{|\mathcal{A}|})$, which allows to find an optimal action in a given state s with a single forward pass through the net. Therefore target y for given transition $(s, a, r', s', \mathbf{done})$ can be computed with one forward pass and optimization step can be performed in one more forward[7] and one backward pass.

Small issue with this straightforward approach is that, of course, it is impractical to train neural networks with batches of size 1. In [13] it is proposed to use ***experience replay*** to store all collected transitions $(s, a, r', s', \mathbf{done})$ as data samples and on each iteration sample a batch of standard for neural networks training size. As usual, the loss function is assumed to be an average of losses for each transition from the batch. This utilization of previously experienced transitions is legit because TD algorithm is known to be an ***off-policy*** algorithm, which means it can work with arbitrary transitions gathered by any agent's interaction experience. One more important benefit from experience replay is ***sample decorrelation*** as consecutive transitions from interaction are often similar to each other since agent usually locates at the particular part of MDP.

Though empirical results of described algorithm turned out to be promising, the behaviour of Q^*_θ values indicated the instability of learning process. Reconstruction of target after each optimization step led to so-called ***compound error*** when approximation error propagated from the close-to-terminal states to the starting in avalanche manner and could lead to guess being $\mathbf{10^6}$ and more times bigger than the true Q^* value. To address this problem, [13] introduced a kludge known as ***target network***, which basic idea is to solve fixed regression problem for $K > 1$ steps, i. .e. recompute target every K-th step instead of each.

[7] in implementations it is possible to combine s and s' in one batch and perform these two forward passes «at once».

To avoid target recomputation for the whole experience replay, the copy of neural network Q^*_θ is stored, called the target network. Its architecture is the same while weights θ^- are a copy of Q^*_θ from the moment of last target recomputation[8] and its main purpose is to generate targets y for given current batch.

Combining all things together and adding ε-greedy strategy to facilitate exploration, we obtain classic DQN algorithm:

Algorithm 2: Deep Q-learning (DQN)

Hyperparameters: B — batch size, K — target network update frequency, $\varepsilon(t) \in (0, 1]$ — greedy exploration parameter, Q^*_θ — neural network, SGD optimizer.

Initialize weights of θ arbitrary
Initialize $\theta^- \leftarrow \theta$
On each interaction step:

1. select a randomly with probability $\varepsilon(t)$, else $a = \underset{a}{argmax}\, Q^*_\theta(s, a)$
2. observe transition $(s, a, r', s', \mathbf{done})$
3. add observed transition to experience replay
4. sample batch of size B from experience replay
5. for each transition T from the batch compute target:
$$y(T) = r(s') + \gamma \max_{a'} Q^*(s', a', \theta^-)$$
6. compute loss:
$$\mathbf{Loss} = \frac{1}{B} \sum_T \left(Q^*(s, a, \theta) - y(T)\right)^2$$
7. make a step of gradient descent using $\frac{\partial\, \mathbf{Loss}}{\partial \theta}$
8. if $t \mod K = 0$: $\theta^- \leftarrow \theta$

3.3. Double DQN

Although target network successfully prevented Q^*_θ from unbounded growth and empirically stabilized learning process, the values of Q^*_θ on many domains were evident to tend to overestimation. The problem is presumed to reside in max operation in target construction formula (10):

$$y = r(s') + \gamma \max_{a'} Q^*(s', a', \theta^-)$$

During this estimation $\mathbf{max}$ shifts Q-value estimation towards either to those actions that led to high reward due to luck or to the actions with overestimating approximation error.

The solution proposed in [23] is based on idea of separating ***action selection*** and ***action evaluation*** to carry out each of these operations using its own approximation of Q^*:

$$\max_{a'} Q^*(s', a', \theta^-) = Q^*(s', \underset{a'}{argmax}\, Q^*(s', a', \theta^-), \theta^-) \approx$$
$$\approx Q^*(s', \underset{a'}{argmax}\, Q^*(s', a', \theta_1^-), \theta_2^-)$$

The simplest, but expensive, implementation of this idea is to run two independent DQN («Twin DQN») algorithms and use the twin network to evaluate actions:

$$y_1 = r(s') + \gamma Q_1^*(s', \underset{a'}{argmax}\, Q_2^*(s', a', \theta_2^-), \theta_1^-)$$

[8] alternative, but more computationally expensive option, is to update target network weights on each step using exponential smoothing

$$y_2 = r(s') + \gamma Q_2^*(s', \underset{a'}{argmax}\, Q_1^*(s', a', \theta_1^-), \theta_2^-)$$

Intuitively, each Q-function here may prefer lucky or overestimated actions, but the other Q-function judges them according to its own luck and approximation error, which may be as underestimating as overestimating. Ideally these two DQNs should not share interaction experience to achieve that, which makes such algorithm twice as expensive both in terms of computational cost and sample efficiency.

Double DQN [23] is more compromised option which suggests to use current weights of network $\boldsymbol{\theta}$ for action selection and target network weights $\boldsymbol{\theta}^-$ for action evaluation, assuming that when the target network update frequency $\boldsymbol{K}$ is big enough these two networks are sufficiently different:

$$y = r(s') + \gamma Q^*(s', \underset{a'}{argmax}\, Q^*(s', a', \theta), \theta^-)$$

3.4. Dueling DQN

Another issue with DQN algorithm 2 emerges when a huge part of considered MDP consists of states of low optimal value $V^*(s)$, which is an often case. The problem is that when the agent visits unpromising state instead of lowering its value $V^*(s)$ it remembers only low pay-off for performing some action a in it by updating $Q^*(s, a)$. This leads to regular returns to this state during future interactions until all actions prove to be unpromising and all $Q^*(s, a)$ are updated. The problem gets worse when the cardinality of action space is high or there are many similar actions in action space.

One benefit of deep reinforcement learning is that we are able to facilitate generalization across actions by specifying the architecture of neural network. To do so, we need to encourage the learning of $V^*(s)$ from updates of $Q^*(s, a)$. The idea of ***dueling architecture*** [25] is to incorporate approximation of $V^*(s)$ explicitly in computational graph. For that purpose we need the definition of advantage function:

Definition 8. For given MDP and policy π the ***advantage function under policy*** π is defined as

$$A^\pi(s, a) \coloneqq Q^\pi(s, a) - V^\pi(s) \tag{14}$$

Advantage function is evidently interconnected with Q-function and value function and actually shows the relative advantage of selecting action a comparing to average performance of the policy. If for some state $A^\pi(s, a) > 0$, then modifying π to select a more often in this particular state will lead to better policy as its average return will become bigger than initial $V^\pi(s)$. This follows from the following property of arbitrary advantage function:

$$\begin{aligned} \mathbb{E}_{a\sim\pi(a|s)} A^\pi(s, a) &= \mathbb{E}_{a\sim\pi(a|s)} \left[Q^\pi(s, a) - V^\pi(s)\right] = \\ &= \mathbb{E}_{a\sim\pi(a|s)} Q^\pi(s, a) - V^\pi(s) = \\ \{\text{using (4)}\} &= V^\pi(s) - V^\pi(s) = 0 \end{aligned} \tag{15}$$

Definition of optimal advantage function $A^*(s, a)$ is analogous and allows us to reformulate $Q^*(s, a)$ in terms of $V^*(s)$ and $A^*(s, a)$:

$$Q^*(s, a) = V^*(s) + A^*(s, a) \tag{16}$$

Straightforward utilization of this decomposition is following: after several feature extracting layers the network is joined with two heads, one outputting single scalar $V^*(s)$ and one outputting $|\mathcal{A}|$ numbers $A^*(s, a)$ like it was done in DQN for Q-function. After that this scalar value estimation is added to all components of $A^*(s, a)$ in order to obtain $Q^*(s, a)$ according to (16). The problem with this naive approach is that due to (15) advantage function can not be arbitrary and must hold the property (15) for $Q^*(s, a)$ to be identifiable.

This restriction (15) on advantage function can be simplified for the case when optimal policy is

induced by optimal Q-function:

$$\begin{aligned}
\mathbf{0} &= \mathbb{E}_{a\sim\pi^*(a|s)} Q^*(s,a) - V^*(s) = \\
&= Q^*(s, \underset{a}{argmax}\, Q^*(s,a)) - V^*(s) = \\
&= \max_a Q^*(s,a) - V^*(s) = \\
&= \max_a [Q^*(s,a) - V^*(s)] = \\
&= \max_a A^*(s,a)
\end{aligned}$$

This condition can be easily satisfied in computational graph by subtracting $\max_a A^*(s,a)$ from advantage head. This will be equivalent to the following formula of dueling DQN:

$$Q^*(s,a) = V^*(s) + A^*(s,a) - \max_a A^*(s,a) \tag{17}$$

The interesting nuance of this improvement is that after evaluation on Atari-57 authors discovered that substituting max operation in (17) with averaging across actions led to better results (while usage of unidentifiable formula (16) led to poor performance). Although gradients can be backpropagated through both operation and formula (17) seems theoretically justified, in practical implementations averaging instead of maximum is widespread.

3.5. Noisy DQN

By default, DQN algorithm does not concern the exploration problem and is always augmented with ε-greedy strategy to force agent to discover new states. This baseline exploration strategy suffers from being extremely hyperparameter-sensitive as early decrease of $\varepsilon(t)$ to close to zero values may lead to stucking in local optima, when agent is unable to explore new options due to imperfect Q^*, while high values of $\varepsilon(t)$ force agent to behave randomly for excessive amount of episodes, which slows down learning. In other words, ε-greedy strategy transfers responsibility to solve ***exploration-exploitation*** trade-off on engineer.

The key reason why ε-greedy exploration strategy is relatively primitive is that exploration priority does not depend on current state. Intuitively, the choice whether to exploit knowledge by selecting approximately optimal action or to explore MDP by selecting some other depends on how explored the current state s is. Discovering a new part of state space after any amount of interaction probably indicates that random actions are good to try there, while close-to-initial states will probably be sufficiently explored after several first episodes.

In ε-greedy strategy agent selects action using deterministic $Q^*(s,a,\theta)$ and only afterwards injects state-independent noise in a form of $\varepsilon(t)$ probability of choosing random action. ***Noisy networks*** [4] were proposed as a simple extension of DQN to provide state-dependent and parameter-free exploration by injecting noise of trainable volume to all (or most[9]) nodes in computational graph.

Let a linear layer with m inputs and n outputs in q-network perform the following computation:

$$y(x) = Wx + b$$

where $x \in \mathbb{R}^m$ is input, $W \in \mathbb{R}^{n\times m}$ — weights matrix, $b \in \mathbb{R}^m$ — bias. In noisy layers it is proposed to substitute deterministic parameters with samples from $\mathcal{N}(\mu, \sigma)$ where μ, σ are trained with gradient descent[10]. On the forward pass through the noisy layer we sample $\varepsilon_W \sim \mathcal{N}(0, I_{nm\times nm}), \varepsilon_b \sim \mathcal{N}(0, I_{n\times n})$ and then compute

$$\begin{aligned}
W &= (\mu_W + \sigma_W \odot \varepsilon_W) \\
b &= (\mu_b + \sigma_b \odot \varepsilon_b) \\
y(x) &= Wx + b
\end{aligned}$$

where $\odot$ denotes element-wise multiplication, $\mu_W, \sigma_W \in \mathbb{R}^{n\times m}, \mu_b, \sigma_b \in \mathbb{R}^n$ — trainable parameters of the layer. Note that the number of parameters for such layers is doubled comparing to ordinary layers.

[9] usually it is not injected in very first layers responsible for feature extraction like convolutional layers in networks for images as input.

[10] using standard reparametrization trick

As the output of q-network now becomes a random variable, loss value becomes a random variable too. Like in similar models for supervised learning, on each step an expectation of loss function over noise is minimized:

$$\mathbb{E}_{\varepsilon}\,\mathbf{Loss}(\boldsymbol{\theta}, \varepsilon) \to \min_{\boldsymbol{\theta}}$$

The gradient in this setting can be estimated using Monte-Carlo:

$$\nabla_{\boldsymbol{\theta}}\mathbb{E}_{\varepsilon}\,\mathbf{Loss}(\boldsymbol{\theta}, \varepsilon) = \mathbb{E}_{\varepsilon}\nabla_{\boldsymbol{\theta}}\,\mathbf{Loss}(\boldsymbol{\theta}, \varepsilon) \approx \nabla_{\boldsymbol{\theta}}\,\mathbf{Loss}(\boldsymbol{\theta}, \varepsilon) \quad \varepsilon \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$$

It can be seen that amount of noise actually inflicting output of network may vary for different inputs, i. e. for different states. There are no guarantees that this amount will reduce as the interaction proceeds; the behaviour of average magnitude of noise injected in the network with time is reported to be extremely sensitive to initialization of $\boldsymbol{\sigma_W}, \boldsymbol{\sigma_b}$ and vary from MDP to MDP.

One technical issue with noisy layers is that on each pass an excessive amount (by the number of network parameters) of noise samples is required. This may substantially reduce computational efficiency of forward pass through the network. For optimization purposes it is proposed to obtain noise for weights matrices in the following way: sample just $\boldsymbol{n} + \boldsymbol{m}$ noise samples $\varepsilon^{\mathbf{1}}_{\boldsymbol{W}} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}_{m\times m}), \varepsilon^{\mathbf{2}}_{\boldsymbol{W}} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}_{n\times n})$ and acquire matrix noise in a factorized form:

$$\varepsilon_{\boldsymbol{W}} = \boldsymbol{f}(\varepsilon^{\mathbf{1}}_{\boldsymbol{W}})\boldsymbol{f}(\varepsilon^{\mathbf{2}}_{\boldsymbol{W}})^{T}$$

where $\boldsymbol{f}$ is a scaling function, e. g. $\boldsymbol{f}(\boldsymbol{x}) = \mathbf{sign}(\boldsymbol{x})\sqrt{|\boldsymbol{x}|}$. The benefit of this procedure is that it requires $\boldsymbol{m} + \boldsymbol{n}$ samples instead of $\boldsymbol{mn}$, but sacrifices the interlayer independence of noise.

3.6. Prioritized experience replay

In DQN each batch of transitions is sampled from experience replay using uniform distribution, treating collected data as equally prioritized. In such scheme states for each update come from the same distribution as they come from interaction experience (except that they become decorellated), which agrees with TD algorithm as the basement of DQN.

Intuitively observed transitions vary in their importance. At the beginning of training most guesses tend to be more or less random as they rely on arbitrarily initialized $\boldsymbol{Q^*_\theta}$ and the only source of trusted information are transitions with non-zero received reward, especially near terminal states where $\boldsymbol{V^*_\theta}(\boldsymbol{s'})$ is known to be equal to 0. In the midway of training, most of experience replay is filled with the memory of interaction within well-learned part of MDP while the most crucial information is contained in transitions where agent explored new promising areas and gained novel reward yet to be propagated through Bellman equation. All these significant transitions are drowned in collected data and rarely appear in sampled batches.

The central idea of prioritized experience replay [16] is that priority of some transition $\boldsymbol{T} = (\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{r'}, \boldsymbol{s'}, \mathbf{done})$ is proportional to temporal difference:

$$\rho(\boldsymbol{T}) \coloneqq \boldsymbol{y}(\boldsymbol{T}) - \boldsymbol{Q^*}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{\theta}) = \sqrt{\mathbf{Loss}(\boldsymbol{y}(\boldsymbol{T}), \boldsymbol{Q^*}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{\theta}))} \tag{18}$$

Using these priorities as proxy of transition importances, sampling from experience replay proceeds using following probabilities:

$$\mathcal{P}(\boldsymbol{T}) \propto \rho(\boldsymbol{T})^{\alpha}$$

where hyperparameter $\boldsymbol{\alpha} \in \mathbb{R}^{+}$ controls the degree to which the sampling weights are sparsified: the case $\boldsymbol{\alpha} = \mathbf{0}$ corresponds to uniform sampling distribution while $\boldsymbol{\alpha} = +\boldsymbol{\infty}$ is equivalent to greedy sampling of transitions with highest priority.

The problem with (18) claim is that each transition's priority changes after each network update. As it is impractical to recalculate loss for the whole data after each step, some simplifications must be put up with. The straightforward option is to update priority only for sampled transitions in the current batch. New transitions can be added to experience replay with highest priority, i. e. $\max_{\boldsymbol{T}} \rho(\boldsymbol{T})$[11].

Second debatable issue of prioritized replay is that it actually substitutes loss function of DQN updates, which assumed uniform sampling of visited states to ensure they come from state visitation distribution:

$$\mathbb{E}_{\boldsymbol{T}\sim\mathbf{Uniform}}\,\mathbf{Loss}(\boldsymbol{T}) \to \min_{\boldsymbol{\theta}}$$

[11] which can be computed online with $\mathcal{O}(\mathbf{1})$ complexity

While it is not clear what distribution is better to sample from to ensure exploration restrictions of theorem 7, prioritized experienced replay changes this distribution in uncontrollable way. Despite its fruitfulness at the beginning and midway of training process, this distribution shift may destabilize learning close to the end and make algorithm stuck with locally optimal policy. Since formally this issue is about estimating an expectation over one probability with preference to sample from another one, the standard technique called ***importance sampling*** can be used as countermeasure:

$$\begin{aligned}\mathbb{E}_{T\sim\mathbf{Uniform}}\,\mathbf{Loss}(T) &= \sum_{i=0}^{M}\frac{1}{M}\,\mathbf{Loss}(T_i) = \\ &= \sum_{i=0}^{M}\mathcal{P}(T_i)\frac{1}{M\mathcal{P}(T_i)}\,\mathbf{Loss}(T_i) = \\ &= \mathbb{E}_{T\sim\mathcal{P}(T)}\frac{1}{M\mathcal{P}(T)}\,\mathbf{Loss}(T)\end{aligned}$$

where M is a number of transitions stored in experience replay memory. Importance sampling implies that we can avoid distribution shift that introduces undesired bias by making smaller gradient updates for significant transitions which now appear in the batches with higher frequency. The price for bias elimination is that importance sampling weights lower prioritization effect by slowing down learning of highlighted new information.

This duality resembles trade-off between bias and variance, but important moment here is that distribution shift does not cause any seeming issues at the beginning of training when agent behaves close to random and do not produce valid state visitation distribution anyway. The idea proposed in [16] based on this intuition is to anneal the importance sampling weights so they correct bias properly only towards the end of training procedure.

$$\mathbf{Loss}^{\mathbf{prioritizedER}} = \mathbb{E}_{T\sim\mathcal{P}(T)}\left(\frac{1}{B\mathcal{P}(T)}\right)^{\beta(t)}\mathbf{Loss}(T)$$

where $\beta(t)\in[0,1]$ and approaches 1[12] as more interaction steps are executed. If $\beta(t)$ is set to 0, no bias correction is held, while $\beta(t)=1$ corresponds to unbiased loss function, i. e. equivalent to sampling from uniform distribution.

The most significant and obvious drawback of prioritized experience replay approach is that it introduces additional hyperparameters. Although α represents one number, algorithm's behaviour may turn out to be sensitive to its choosing, and $\beta(t)$ must be designed by engineer as some scheduled motion from something near 0 to 1, and its well-turned selection may require inaccessible knowledge about how many steps it will take for algorithm to «warm up».

3.7. Multi-step DQN

One more widespread modification of Q-learning in RL community is substituting one-step approximation present in Bellman optimality equation (6) with N-step:

Proposition 8. ***(N-step Bellman optimality equation)***

$$Q^*(s_0,a_0) = \mathbb{E}_{\mathcal{T}_{\pi^*}|s_0,a_0}\left[\sum_{t=1}^{N}\gamma^{t-1}r(s_t) + \gamma^N\max_{a_N}Q^*(s_N,a_N)\right] \tag{19}$$

Indeed, definition of $Q^*(s,a)$ consists of average return and can be viewed as making $T^{\max}$ steps from state s_0 after selecting action a_0, while vanilla Bellman optimality equation represents $Q^*(s,a)$ as reward from one next step in the environment and estimation of the rest of trajectory reward recursively. N-step Bellman equation (19) generalizes these two opposites.

All the same reasoning as for DQN can be applied to N-step Bellman equation to obtain N-step DQN algorithm, which only modification appears in target computation:

$$y(s_0,a_0) = \sum_{t=1}^{N}\gamma^{t-1}r(s_t) + \gamma^N\max_{a_N}Q^*(s_N,a_N,\theta) \tag{20}$$

[12] often it is initialized by a constant close to 0 and is linearly increased until it reaches 1

To perform this computation, we are required to obtain for given state s and a not only one next step, but N steps. To do so, instead of transitions N-step roll-outs are stored, which can be done by precomputing following tuples:

$$T = \left(s,\, a,\, \sum_{n=1}^{N} \gamma^{n-1} r^{(n)},\, s^{(N)},\, \mathbf{done} \right)$$

where $r^{(n)}$ is the reward received in n steps after visitation of considered state s, $s^{(N)}$ is state visited in N steps, and $\mathbf{done}$ is a flag whether the episode ended during N-step roll-out[13]. All other aspects of algorithm remain the same in practical implementations, and the case $N = 1$ corresponds to standard DQN.

The goal of using $N > 1$ is to accelerate propagation of reward from terminal states backwards through visited states to s_0 as less update steps will be required to take into account freshly observed reward and optimize behaviour at the beginning of episodes. The price is that formula (20) includes an important trick: to calculate such target, for second (and following) step action a' must be sampled from π^* for Bellman equation (19) to remain true. In other words, application of N-step Q-learning is theoretically improper when behaviour policy differs from π^*. Note that we do not face this problem in the case $N = 1$ in which we are required to sample only from transition probability $p(s' \mid s, a)$ for given state-action pair s, a.

Even considering $\pi^* \approx \underset{a}{argmax}\, Q^*(s, a, \theta)$, where Q^* is our current approximation of π^*, makes N-step DQN an ***on-policy*** algorithm when for every state-action pair s, a it is preferable to sample target using the closest approximation of π^* available. This questions usage of experience replay or at the very least encourages to limit its capacity to store only $M^{\max}$ newest transitions with $M^{\max}$ being relatively not very big.

To see the negative effect of N-step DQN, consider the following toy example. Suppose agent makes a mistake on the second step after s and ends episode with huge negative reward. Then in the case $N > 2$ each time the roll-out starting with this s is sampled in the batch, the value of $Q^*(s, a, \theta)$ will be updated with this received negative reward even if $Q^*(s', \cdot, \theta)$ already learned not to repeat this mistake again.

Yet empirical results in many domains demonstrate that raising N from 1 to 2-3 may result in substantial acceleration of training and positively affect the final performance. On the contrary, the theoretical groundlessness of this approach explains its negative effects when N is set too big.

[13] all N-step roll-outs must be considered including those terminated at k-th step for $k < N$.

4. Distributional approach for value-based methods

4.1. Theoretical foundations

The setting of RL task inherently carries internal stochasticity of which agent has no substantial control. Sometimes intelligent behaviour implies taking risks with severe chance of low episode return. All this information resides in the distribution of return $\boldsymbol{R}$ (1) as random variable.

While value-based methods aim at learning expectation of this random variable as it is the quantity we actually care about, in distributional approach [1] it is proposed to learn the whole distribution of returns. It further extends the information gathered by algorithm about MDP towards model-based case in which the whole MDP is imitated by learning both reward function $r(s)$ and transitions $\mathbb{T}$, but still restricts itself only to reward and doesn't intend to learn world model.

In this section we discuss some theoretical extensions of temporal difference ideas in the case when expectations on both sides of Bellman equation (5) and Bellman optimality equation (6) are taken away.

The central object of study in Q-learning was Q-function, which for given state and action returns the expectation of reward. To rewrite Bellman equation not in terms of expectations, but in terms of the whole distributions, we require a corresponding notation.

Definition 9. For given MDP and policy π the ***value distribution of policy*** π is a random variable defined as

$$Z^{\pi}(s,a) := \sum_{t=0} \gamma^t r_{t+1} \;\Big|\; s_0 = s, a_0 = a$$

Note that Z^{π} just represents a random variable which is taken expectation of in definition of Q-function:

$$Q^{\pi}(s,a) = \mathbb{E}_{\mathcal{T}_{\pi}} Z^{\pi}(s,a)$$

Using this definition of value distribution, Bellman equation can be rewritten to extend the recursive connection between adjacent states from expectations of returns to the whole distributions of returns:

Proposition 9. ***(Distributional Bellman Equation)*** *[1]*

$$Z^{\pi}(s,a) \overset{\text{c.d.f.}}{=} r(s') + \gamma Z^{\pi}(s',a') \;\Big|\; s' \sim p(s' \mid s,a), a' \sim \pi(a' \mid s') \tag{21}$$

Here we used some auxiliary notation: by $\overset{\text{c.d.f.}}{=}$ we mean that cumulative distribution functions of two random variables to the right and left are equal almost everywhere. Such equations are called ***recursive distributional equations*** and are well-known in theoretical probability theory[14]. By using $\mid$ we describe a sampling procedure for the random variable to the right side of equation: for given s, a next state s' is sampled from transition probability, then a' is sampled from given policy, then random variable $Z^{\pi}(s',a')$ is sampled to calculate a resulting sample $r(s') + \gamma Z^{\pi}(s',a')$.

While the space of Q-functions $Q^{\pi}(s,a) \in \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is finite, the space of value distributions is a space of mappings from state-action pair to continuous distributions:

$$Z^{\pi}(s,a) \in \mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathbb{R})$$

and it is important to notice that even in the table-case when state and action spaces are finite, the space of value distributions is essentially infinite. Crucial moment for us will be that convergence properties now depend on chosen metric[15].

The choice of metric in $\mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathbb{R})$ represents the same issue as in the space of continuous random variables $\mathcal{P}(\mathbb{R})$: if we choose a metric in the latter, we can construct one in the former:

[14] to get familiar with this notion, consider this basic example:

$$X_1 \overset{\text{c.d.f.}}{=} \frac{X_2}{\sqrt{2}} + \frac{X_3}{\sqrt{2}}$$

where X_1, X_2, X_3 are random variables coming from $\mathcal{N}(0, \sigma^2)$.

[15] in finite spaces it is true that convergence in one metric guarantees convergence to the same point for any other metric.

Proposition 10. *If $d(X, Y)$ is a metric in the space $\mathcal{P}(\mathbb{R})$, then*

$$\overline{d}(Z_1, Z_2) \coloneqq \sup_{s \in \mathcal{S}, a \in \mathcal{A}} d(Z_1(s,a), Z_2(s,a))$$

is a metric in the space $\mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathbb{R})$.

The particularly interesting for us example of metric in $\mathcal{P}(\mathbb{R})$ will be Wasserstein metric, which concerns only random variables with bounded moments, so we will additionally assume that for all state-action pairs s, a

$$\mathbb{E} Z^\pi(s,a)^p \leq +\infty$$

are finite for $p \geq 1$.

Proposition 11. *For $1 \leq p \leq +\infty$ for two random variables X, Y on continuous domain with p-th bounded moments and cumulative distribution functions F_X and F_Y correspondingly a* **Wasserstein distance**

$$W_p(X, Y) \coloneqq \left(\int_0^1 \left| F_X^{-1}(\omega) - F_Y^{-1}(\omega) \right|^p d\omega \right)^{\frac{1}{p}}$$

$$W_\infty(X, Y) \coloneqq \sup_{\omega \in [0,1]} \left| F_X^{-1}(\omega) - F_Y^{-1}(\omega) \right|$$

is a metric in the space of random variables with p-th bounded moments.

Thus we can conclude from proposition 10 that maximal form of Wasserstein metric

$$\overline{W}_p(Z_1, Z_2) = \sup_{s \in \mathcal{S}, a \in \mathcal{A}} W_p(Z_1(s,a), Z_2(s,a)) \tag{22}$$

is a metric in the space of value distributions.

We now concern convergence properties of point iteration method to solve (21) in order to obtain Z^π for given policy π, i. e. solve the task of ***policy evaluation***. For that purpose we initialize $Z_0^\pi(s,a)$ arbitrarily[16] and perform the following updates for all state-action pairs s, a:

$$Z_{t+1}^\pi(s,a) \overset{\text{c.d.f.}}{\coloneqq} r(s') + \gamma Z_t^\pi(s', a') \tag{23}$$

Here we assume that we are able to compute the distribution of random variable on the right side knowing π, all transition probabilities $\mathbb{T}$, the distribution of Z_t^π and reward function. The question whether the sequence $\{Z_t^\pi\}$ converges to Z^π can be given a detailed answer:

Proposition 12. *[1] Denote by $\mathcal{B}$ the following operator $(\mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathbb{R})) \to (\mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathbb{R}))$, updating Z_t^π as in (23):*

$$Z_{t+1}^\pi = \mathcal{B} Z_t^\pi$$

for all state-action pairs s, a.

Then $\mathcal{B}$ is a contraction mapping in $\overline{W}_p$ (22) for $1 \leq p \leq +\infty$, i.e. for any two value distributions Z_1, Z_2

$$\overline{W}_p(\mathcal{B} Z_1, \mathcal{B} Z_2) \leq \gamma \overline{W}_p(Z_1, Z_2)$$

Hence there is a unique fixed point of system of equations (21) and the point iteration method converges to it.

One more curious theoretical result is that $\mathcal{B}$ is in general not a contraction mapping for such distances as Kullback-Leibler divergence, Total Variation distance and Kolmogorov distance[17]. It shows

[16] here we consider value distributions from theoretical point of view, assuming that we are able to explicitly store a table of $|\mathcal{S}||\mathcal{A}|$ continuous distributions without any approximations.

[17] one more metric for which the contraction property was shown is Cramer metric:

$$l_2(X, Y) = \left(\int_{\mathbb{R}} (F_X(\omega) - F_Y(\omega))^2 \, d\omega \right)^{\frac{1}{2}}$$

where F_X, F_Y are c.d.f. of random variables X, Y correspondingly.

that metric selection indeed influences convergence rate.

Similar to traditional value functions, we can define ***optimal value distribution*** $Z^*(s,a)$. Substituting[18] $\pi^*(s) = \underset{a}{argmax}\, \mathbb{E}_{\mathcal{T}_{\pi^*}} Z^*(s,a)$ into (21), we obtain distributional Bellman optimality equation:

Proposition 13. ***(Distributional Bellman optimality equation)***

$$Z^*(s,a) \overset{\text{c.d.f.}}{=} r(s') + \gamma Z^*(s', \underset{a'}{argmax}\, \mathbb{E}_{\mathcal{T}_{\pi^*}} Z^*(s',a')) \;\mid\; s' \sim p(s' \mid s,a) \tag{24}$$

Now we concern the same question whether the point iteration method of solving (24) leads to solution Z^* and whether it is a contraction mapping for some metric. The answer turns out to be negative.

Proposition 14. *[1] Point iteration for solving (24) may diverge.*

Level of impact of this result is not completely clear. Point iteration for (24) preserves means of distributions, i. e. it will eventually converge to $Q^*(s,a)$ with all theoretical guarantees from classical Q-learning. The reason behind divergence theorems hides in the rest of distributions like other moments and situations when equivalent (in terms of average return) actions may lead to different higher moments.

4.2. Categorical DQN

There are obvious obstacles for practical application of distributional Q-learning following from complication of working with arbitrary continuous distributions. Usually we are restricted to approximations inside some family of parametric distributions, so we have to perform a projection step on each iteration.

Second matter in combining distributional Q-learning with deep neural networks is to take into account that only samples from $p(s' \mid s,a)$ are available for each update. To provide a distributional analog of temporal difference algorithm 9, some analog of exponential smoothing for distributional setting must be proposed.

Categorical DQN [1] (also referred as c51) provides straightforward design of practical distributional algorithm. While DQN was a resemblance of temporal difference algorithm, Categorical DQN attempts to follow the logic of DQN.

The concept is as following. The neural network with parameters θ in this setting takes as input $s \in \mathcal{S}$ and for each action a outputs parameters $\zeta_\theta(s,a)$ of distributions of random variable $Z^*_\theta(s,a)$. As in DQN, experience replay can be used to collect observed transitions and sample a batch for each update step. For each transition $T = (s,a,r',s',\mathbf{done})$ in the batch a guess is computed:

$$y(T) \overset{\text{c.d.f.}}{:=} r' + (1-\mathbf{done})\gamma Z^*_\theta \left(s', \underset{a'}{argmax}\, \mathbb{E} Z^*_\theta(s',a')\right) \tag{25}$$

Note that expectation of $Z^*_\theta(s',a')$ is computed explicitly using the form of chosen parametric family of distributions and outputted parameters $\zeta_\theta(s',a')$, as is the distribution of random variable $r' + (1-\mathbf{done})\gamma Z^*_\theta(s',a')$. In other words, in this setting guess $y(T)$ is also a continuous random variable, distribution of which can be constructed only approximately. As both target and model output are distributions, it is reasonable to design loss function in a form of some divergence $\mathcal{D}$ between $y(T)$ and $Z^*_\theta(s,a)$:

$$\mathbf{Loss}(\theta) = \mathbb{E}_T \mathcal{D}\left(y(T) \parallel Z^*_\theta(s,a)\right) \tag{26}$$

$$\theta_{t+1} = \theta_t - \alpha \frac{\partial\, \mathbf{Loss}(\theta_t)}{\partial \theta}$$

[18] to perform this step validly, a clarification concerning $argmax$ operator definition must be given. The choice of action a returned by this operator in the cases when several actions lead to the same maximal average returns must not depend on Z, as this choice affects higher moments of resulted distribution. To overcome this issue, for example, in the case of finite action space all actions can be enumerated and the optimal action with the lowest index is returned by operator.

The particular choice of this divergence must be made with concern that $y(T)$ is a «sample» from a full one-step approximation of Z^*_θ which includes transition probabilities:

$$y^{\text{full}}(s,a) \overset{\text{c.d.f.}}{\coloneqq} \sum_{s' \in \mathcal{S}} p(s' \mid s,a) y(s,a,r(s'),s',\text{done}(s')) \tag{27}$$

This form is precisely the right side of distributional Bellman optimality equation as we just incorporated intermediate sampling of s' into the value of random variable. In other words, if transition probabilities $\mathbb{T}$ were known, the update could be made using distribution of y^{full} as a target.

$$\text{Loss}^{\text{full}}(\theta) = \mathbb{E}_{s,a} \mathcal{D}(y^{\text{full}}(s,a) \parallel Z^*_\theta(s,a))$$

This motivates to choose $\text{KL}(y(T) \parallel Z^*_\theta(s,a))$ (specifically with this order of arguments) as $\mathcal{D}$ to exploit the following property (we denote by p_X a p.d.f. pf random variable X):

$$\begin{aligned}
\nabla_\theta \mathbb{E}_T \,\text{KL}(y^{\text{full}}(s,a) \parallel Z^*_\theta(s,a)) &= \nabla_\theta \left[\mathbb{E}_T \int_{\mathbb{R}} -p_{y^{\text{full}}(s,a)}(\omega) \log p_{Z^*_\theta(s,a))}(\omega) d\omega + \text{const}(\theta) \right] = \\
\{\text{using (27)}\} &= \nabla_\theta \mathbb{E}_T \int_{\mathbb{R}} \mathbb{E}_{s' \sim p(s'|s,a)} - p_{y(T)}(\omega) \log p_{Z^*_\theta(s,a))}(\omega) d\omega = \\
\{\text{taking expectation out}\} &= \nabla_\theta \mathbb{E}_T \mathbb{E}_{s' \sim p(s'|s,a)} \int_{\mathbb{R}} -p_{y(T)}(\omega) \log p_{Z^*_\theta(s,a))}(\omega) d\omega = \\
&= \nabla_\theta \mathbb{E}_T \mathbb{E}_{s' \sim p(s'|s,a)} \,\text{KL}\left(y(T) \parallel Z^*_\theta(s,a)\right)
\end{aligned}$$

This property basically states that gradient of loss function (26) with KL as $\mathcal{D}$ is an unbiased (Monte-Carlo) estimation of gradient of KL-divergence for «full» distribution (27), which resembles the employment of exponential smoothing in temporal difference learning. For many other divergences, including Wasserstein metric, same statement is not true, so their utilization in described online setting will lead to biased gradients and all theory-grounded intuition that algorithm moves in the right direction becomes distinctively lost. Moreover, KL-divergence is known to be one of the easiest divergences to work with due to its nice smoothness properties and wide prevalence in many deep learning pipelines.

Described above motivation to choose KL-divergence as an actual objective for minimization is contradictory. Theoretical analysis of distributional Q-learning, specifically theorem 12, though concerning policy evaluation other than optimal Z^* search, explicitly hints that the process converges exponentially fast for Wasserstein metric, while even for precisely made updates in terms of KL-divergence we are not guaranteed to get any closer to true solution.

More «practical» defect of KL-divergence is that it demands two comparable distributions to share the same domain. This means that by choosing KL-divergence we pledge to guarantee that $y(T)$ and $Z^*_\theta(s,a)$ in (26) have coinciding support. This emerging restriction seems limiting even beforehand as for episodic MDP value distribution in terminal states is obviously degenerated (their support consists of one point $r(s)$ which is given all probability mass) which means that our value distribution approximation is basically ensured to never be precise.

In Categorical DQN, as follows from the name, the family of distributions is chosen to be categorical on the fixed support $\{z_0, z_1 \dots z_{A-1}\}$ where A is number of ***atoms***. As no prior information about MDP is given, the basic choice of this support is uniform grid from some $V_{\min} \in \mathbb{R}$ to $V^{\max} \in \mathbb{R}$:

$$z_i = V_{\min} + \frac{i}{A-1}(V_{\max} - V_{\min}), \quad i \in 0, 1, \dots A-1$$

These bounds, though, must be chosen carefully as they implicitly assume

$$V_{\min} \le Z^*(s,a) \le V_{\max}$$

and if these inequalities are not tight, the approximation will obviously become poor.

Therefore the neural network outputs A numbers, summing into 1, to represent arbitrary distribution on this support:

$$\zeta_i(s,a,\theta) \coloneqq \mathcal{P}(Z^*_\theta(s,a) = z_i)$$

Within this family of distributions, computation of expectation, greedy action selection and KL-divergence is trivial. One problem hides in target formula (25): while we can compute distribution $y(T)$, its support may in general differ from $\{z_0 \dots z_{A-1}\}$. To avoid the issue of disjoint supports,

a ***projection step*** must be done to find the closest to target distribution within the chosen family[19]. Therefore the resulting target used in the loss function is

$$y(T) \overset{\text{c.d.f.}}{:=} \mathbf{\Pi}_C \left[r' + (1 - \mathbf{done})\gamma Z^*_\theta \left(s', \underset{a'}{argmax}\, \mathbb{E} Z^*_\theta(s', a') \right) \right]$$

where $\mathbf{\Pi}_C$ is projection operator.

The resulting practical algorithm, named c51 after categorical distributions with $A = 51$ atoms, inherits ideas of experience replay, ε-greedy exploration and target network from DQN. Empirically, though, usage of target network remains an open question as the chosen family of distributions restricts value approximation from unbounded growth by «clipping» predictions at z_{A-1} and z_0, yet it is still considered slightly improving performance.

Algorithm 3: Categorical DQN (c51)

Hyperparameters: B — batch size, $V_{\max}, V_{\min}, A$ — parameters of support, K — target network update frequency, $\varepsilon(t) \in (0, 1]$ — greedy exploration parameter, ζ^* — neural network, SGD optimizer.

Initialize weights θ of neural net ζ^* arbitrary
Initialize $\theta^- \leftarrow \theta$
Precompute support grid $z_i = V_{\min} + \frac{i}{A-1}(V_{\max} - V_{\min})$
On each interaction step:

1. select a randomly with probability $\varepsilon(t)$, else $a = \underset{a}{argmax} \sum_i z_i \zeta^*_i(s, a, \theta)$
2. observe transition $(s, a, r', s', \mathbf{done})$
3. add observed transition to experience replay
4. sample batch of size B from experience replay
5. for each transition T from the batch compute target:

$$\mathcal{P}(y(T) = r' + \gamma z_i) = \zeta^*_i \left(s', \underset{a'}{argmax} \sum_i z_i \zeta^*_i(s', a', \theta^-), \theta^- \right)$$

6. project $y(T)$ on support $\{z_0, z_1 \dots z_{A-1}\}$
7. compute loss:

$$\mathbf{Loss} = \frac{1}{B} \sum_T \mathbf{KL}(y(T) \parallel Z^*(s, a, \theta))$$

8. make a step of gradient descent using $\frac{\partial \mathbf{Loss}}{\partial \theta}$
9. if $t \mod K = 0$: $\theta^- \leftarrow \theta$

4.3. Quantile Regression DQN (QR-DQN)

Categorical DQN discovered a gap between theory and practice as $\mathbf{KL}$-divergence, used in practical algorithm, is theoretically unjustified. Theorem 12 hints that the true divergence we should care about is actually Wasserstein metric, but it remained unclear how it could be optimized using only samples from transition probabilities $\mathbb{T}$.

In [3] it was discovered that selecting another family of distributions to approximate $Z^*_\theta(s, a)$ will reduce Wasserstein minimization task to the search for quantiles of specific distributions. The

[19] to project a categorical distribution with support $\{v_0, v_1 \dots v_{A-1}\}$ on categorical distributions with support $\{z_0, z_1 \dots z_{A-1}\}$ one can just find for each v_i the closest two atoms $z_j \le v_i \le z_{j+1}$ and split all probability mass for v_i between z_j and z_{j+1} proportional to closeness. If $v_i < z_0$, then all its probability mass is given to z_0, same with upper bound.

latter can be done in online setting using ***quantile regression*** technique. This led to alternative distributional Q-learning algorithm named Quantile Regression DQN (QR-DQN).

The basic idea is to «swap» fixed support and learned probabilities of Categorical DQN. We will now consider the family with fixed probabilities for A-atomed categorical distribution with arbitrary support $\{\zeta_0^*(s,a,\theta), \zeta_1^*(s,a,\theta), \dots, \zeta_{A-1}^*(s,a,\theta)\}$. Again, we will assume all probabilities to be equal given the absence of any prior knowledge; namely, our distribution family is now

$$Z_\theta^*(s,a) \sim \mathbf{Uniform}\left(\zeta_0^*(s,a,\theta), \dots, \zeta_{A-1}^*(s,a,\theta)\right)$$

In this setting neural network outputs A arbitrary real numbers that represent the support of uniform categorical distribution[20], where A is the number of atoms and the only hyperparameter to select.

For table-case setting, on each step of point iteration we desire to update the cell for given state-action pair s, a with full distribution of random variable to the right side of (24). If we are limited to store only A atoms of the support, the true distribution must be projected on the space of A-atomed categorical distributions. Consider now this task of projecting some given random variable with c.d.f. $F(\omega)$ in terms of Wasserstein distance. Specifically, we will be interested in minimizing $\mathcal{W}_1$-distance for $p = 1$ as the theorem 12 states the contraction property for all $1 \le p \le +\infty$ and we are free to choose any:

$$\int_0^1 \left|F^{-1}(\omega) - U^{-1}_{z_0, z_1 \dots z_{A-1}}(\omega)\right| d\omega \to \min_{z_0, z_1 \dots z_{A-1}} \tag{28}$$

where $U_{z_0, z_1 \dots z_{A-1}}$ is c.d.f. for uniform categorical distribution on given support. Its inverse, also known as ***quantile function***, has a following simple form:

$$U^{-1}_{z_0, z_1 \dots z_{A-1}}(\omega) = \begin{cases} z_0 & 0 \le \omega < \frac{1}{A} \\ z_1 & \frac{1}{A} \le \omega < \frac{2}{A} \\ \vdots & \\ z_{A-1} & \frac{A-1}{A} \le \omega < 1 \end{cases}$$

Substituting this into (28)

$$\sum_{i=0}^{A-1} \int_{\frac{i}{A}}^{\frac{i+1}{A}} \left|F^{-1}(\omega) - z_i\right| d\omega \to \min_{z_0, z_1 \dots z_{A-1}}$$

splits the optimization of Wasserstein into A independent tasks that can be solved separately:

$$\int_{\frac{i}{A}}^{\frac{i+1}{A}} \left|F^{-1}(\omega) - z_i\right| d\omega \to \min_{z_i} \tag{29}$$

Proposition 15. *[3] Let's denote*

$$\tau_i := \frac{\frac{i}{A} + \frac{i+1}{A}}{2}$$

Then every solution for (29) satisfies $F(z_i) = \tau_i$, i. e. it is τ_i-th quantile of c. d. f. F.

The result 15 states that we require only A specific quantiles of random variable to the right side of Bellman equation[21]. Hence the last thing to do to design a practical algorithm is to develop a procedure of unbiased estimation of quantiles for the random variable on the right side of distribution Bellman optimality equation (24).

[20] Note that target distribution is now guaranteed to remain within this distribution family as multiplying on γ just shrinks the support and adding r' just shifts it. We assume that if some atoms of the support coincide, the distribution is still A-atomed categorical; for example, for degenerated distribution (like in the case of terminal states) $\zeta_0^*(s,a,\theta) = \zeta_1^*(s,a,\theta) = \dots = \zeta_{A-1}^*(s,a,\theta)$. This shows that projection step heuristic is not needed for this particular choice of distribution family.

[21] It can be proved that for table-case policy evaluation algorithm which stores in each cell not expectations of reward (as in Q-learning) but A quantiles updated according to distributional Bellman equation (21) using theorem 15 converges to quantiles of $Z^*(s,a)$ in Wasserstein metric for $1 \le p \le +\infty$ and its update operator is a contraction mapping in $\mathcal{W}_\infty$.

Quantile regression is the standard technique to estimate the quantiles of ***empirical distribution*** (i. .e. distribution that is represented by finite amount of i. i. d. samples from it). Recall from machine learning that the constant solution optimizing l1-loss is median, i. .e. $\frac{1}{2}$-th quantile. This fact can be generalized to arbitrary quantiles:

Proposition 16. ***(Quantile Regression)*** *[11] Let's define loss as*

$$\mathbf{Loss}(c, X) = \begin{cases} \tau(c - X) & c \geq X \\ (1 - \tau)(X - c) & c < X \end{cases}$$

Then solution for

$$\mathbb{E}_X \mathbf{Loss}(c, X) \to \min_{c \in \mathbb{R}} \tag{30}$$

is τ-th quantile of distribution of X.

As usual in the case of neural networks, it is impractical to optimize (30) until convergence on each iteration for each of A desired quantiles τ_i. Instead just one step of gradient optimization is made and the outputs of neural network $\zeta_i^*(s, a, \theta)$, which play the role of c in formula (30), are moved towards the quantile estimation via backpropagation. In other words, (30) sets a loss function for network outputs; the losses for different quantiles are summed up. The resulting loss is

$$\mathbf{Loss}^{\mathbf{QR}}(s, a, \theta) = \sum_{i=0}^{A-1} \mathbb{E}_{s' \sim p(s'|s,a)} \mathbb{E}_{y \sim y(T)} \left(\tau - \mathbb{I}[\zeta_i^*(s, a, \theta) < y]\right) \left(\zeta_i^*(s, a, \theta) - y\right) \tag{31}$$

where $\mathbb{I}$ denotes an indicator function. The expectation over $y \sim y(T)$ for given transition can be computed in closed form: indeed, $y(T)$ is also an A-atomed categorical distribution with support $\{r' + \gamma\zeta_0^*(s', a'), \ldots, r' + \gamma\zeta_{A-1}^*(s', a')\}$, where

$$a' = \underset{a'}{argmax}\, \mathbb{E}Z^*(s', a', \theta) = \underset{a'}{argmax}\, \frac{1}{A} \sum_i \zeta_i^*(s', a', \theta)$$

and expectation over transition probabilities, as always, is estimated using Monte-Carlo by sampling transitions from experience replay.

Algorithm 4: Quantile Regression DQN (QR-DQN)

Hyperparameters: B — batch size, A — number of atoms, K — target network update frequency, $\varepsilon(t) \in (0, 1]$ — greedy exploration parameter, ζ^* — neural network, SGD optimizer.

Initialize weights θ of neural net ζ^* arbitrary
Initialize $\theta^- \leftarrow \theta$
Precompute mid-quantiles $\tau_i = \frac{\frac{i}{A} + \frac{i+1}{A}}{2}$
On each interaction step:

1. select a randomly with probability $\varepsilon(t)$, else $a = \underset{a}{argmax}\, \frac{1}{A} \sum_i \zeta_i^*(s, a, \theta)$
2. observe transition $(s, a, r', s', \mathbf{done})$
3. add observed transition to experience replay
4. sample batch of size B from experience replay
5. for each transition T from the batch compute the support of target distribution:

$$y(T)_j = r' + \gamma\zeta_j^* \left(s', \underset{a'}{argmax}\, \frac{1}{A} \sum_i \zeta_i^*(s', a', \theta^-), \theta^-\right)$$

6. compute loss:

$$\mathbf{Loss} = \frac{1}{\boldsymbol{BA}} \sum_{\boldsymbol{T}} \sum_{\boldsymbol{i}} \sum_{\boldsymbol{j}} \left(\tau_i - \mathbb{I}[\zeta_i^*(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{\theta}) < \boldsymbol{y}(\boldsymbol{T})_j]\right) \left(\zeta_i^*(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{\theta}) - \boldsymbol{y}(\boldsymbol{T})_j\right)$$

7. make a step of gradient descent using $\frac{\partial \mathbf{Loss}}{\partial \boldsymbol{\theta}}$
8. if $\boldsymbol{t} \mod \boldsymbol{K} = \mathbf{0}$: $\boldsymbol{\theta}^- \leftarrow \boldsymbol{\theta}$

4.4. Rainbow DQN

Success of Deep Q-learning encouraged a full-scale research of value-based deep reinforcement learning by studying various drawbacks of DQN and developing auxiliary extensions. In many articles some extensions from previous research were already considered and embedded in compared algorithms during empirical studies.

In Rainbow DQN [7], seven Q-learning-based ideas are united in one procedure with ablation studies held whether all these incorporated extensions are essentially necessary for resulted RL algorithm:

- DQN (sec. 3.2)
- Double DQN (sec. 3.3)
- Dueling DQN (sec. 3.4)
- Noisy DQN (sec. 3.5)
- Prioritized Experience Replay (sec. 3.6)
- Multi-step DQN (sec. 3.7)
- Categorical[22] DQN (sec. 4.2)

There is little ambiguity on how these ideas can be combined; we will discuss several non-straightforward circumstances and provide the full algorithm description after.

To apply prioritized experience replay in distributional setting, the measure of transition importance must be provided. The main idea is inherited from ordinary DQN where priority is just loss for this transition:

$$\rho(\boldsymbol{T}) \coloneqq \mathbf{Loss}(\boldsymbol{y}(\boldsymbol{T}), \boldsymbol{Z}^*(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{\theta})) = \mathbf{KL}(\boldsymbol{y}(\boldsymbol{T}) \parallel \boldsymbol{Z}^*(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{\theta}))$$

To combine noisy networks with double DQN heuristic, it is proposed to resample noise on each forward pass through the network and through its copy for target computation. This decision implies that action selection, action evaluation and network utilization are independent and stochastic (for exploration cultivation) steps.

The one snagging combination here is categorical DQN and dueling DQN. To merge these ideas, we need to model advantage $\boldsymbol{A}^*(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{\theta})$ in distributional setting. In Rainbow this is done straightforwardly: the network has two heads, value stream $\boldsymbol{v}(\boldsymbol{s}, \boldsymbol{\theta})$ outputting $\boldsymbol{A}$ real values and advantage stream $\boldsymbol{a}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{\theta})$ outputting $\boldsymbol{A} \times |\boldsymbol{\mathcal{A}}|$ real values. Then these streams are integrated using the same formula (17) with the only exception being softmax applied across atoms dimension to guarantee that output is categorical distribution:

$$\zeta_i^*(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{\theta}) \propto \exp\left(\boldsymbol{v}(\boldsymbol{s}, \boldsymbol{\theta})_i + \boldsymbol{a}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{\theta})_i - \frac{1}{|\boldsymbol{\mathcal{A}}|} \sum_{\boldsymbol{a}} \boldsymbol{a}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{\theta})_i\right) \tag{32}$$

Combining lack of intuition behind this integration formula with usage of mean instead of theoretically justified max makes this element of Rainbow the most questionable. During the ablation studies it was discovered that dueling architecture is the only component that can be removed without noticeable loss of performance. All other ingredients are believed to be crucial for resulting algorithm as they address different problems.

[22] Quantile Regression can be considered instead

Algorithm 5: Rainbow DQN

Hyperparameters: B — batch size, $V_{\max}, V_{\min}, A$ — parameters of support, K — target network update frequency, N — multi-step size, α — degree of prioritized experience replay, $\beta(t)$ — importance sampling bias correction for prioritized experience replay, ζ^* — neural network, SGD optimizer.

Initialize weights θ of neural net ζ^* arbitrary
Initialize $\theta^- \leftarrow \theta$
Precompute support grid $z_i = V_{\min} + \frac{i}{A-1}(V_{\max} - V_{\min})$
On each interaction step:

1. select $a = \underset{a}{argmax} \sum_i z_i \zeta_i^*(s, a, \theta, \varepsilon), \varepsilon \sim \mathcal{N}(0, I)$
2. observe transition $(s, a, r', s', \text{done})$
3. construct N-step transition $T = \left(s, a, \sum_{n=0}^{N} \gamma^n r^{(n+1)}, s^{(N)}, \text{done}\right)$ and add it to experience replay with priority $\max_T \rho(T)$
4. sample batch of size B from experience replay using probabilities $\mathcal{P}(T) \propto \rho(T)^\alpha$
5. compute weights for the batch (where M is the size of experience replay memory)

$$w(T) = \left(\frac{1}{M\mathcal{P}(T)}\right)^{\beta(t)}$$

6. for each transition $T = (s, a, \bar{r}, \bar{s}, \text{done})$ from the batch compute target (detached from computational graph to prevent backpropagation):

$$\varepsilon_1, \varepsilon_2 \sim \mathcal{N}(0, I)$$

$$\mathcal{P}(y(T) = \bar{r} + \gamma^N z_i) = \zeta_i^* \left(\bar{s}, \underset{\bar{a}}{argmax} \sum_i z_i \zeta_i^*(\bar{s}, \bar{a}, \theta, \varepsilon_1), \theta^-, \varepsilon_2\right)$$

7. project $y(T)$ on support $\{z_0, z_1 \dots z_{A-1}\}$
8. update transition priorities

$$\rho(T) \leftarrow \text{KL}(y(T) \parallel Z^*(s, a, \theta, \varepsilon)), \varepsilon \sim \mathcal{N}(0, I)$$

9. compute loss:

$$\text{Loss} = \frac{1}{B} \sum_T w(T)\rho(T)$$

10. make a step of gradient descent using $\frac{\partial \text{Loss}}{\partial \theta}$
11. if $t \mod K = 0$: $\theta^- \leftarrow \theta$

5. Policy Gradient algorithms

5.1. Policy Gradient theorem

Alternative approach to solving RL task is direct optimization of objective

$$J(\theta) = \mathbb{E}_{\mathcal{T}\sim\pi_\theta} \sum_{t=1} \gamma^{t-1} r_t \to \max_\theta \tag{33}$$

as a function of θ. Policy gradient methods provide a framework how to construct an efficient optimization procedure based on stochastic first-order optimization within RL setting.

We will assume that $\pi_\theta(a \mid s)$ is a stochastic policy parameterized with $\theta \in \Theta$. It turns out, that if π is differentiable by θ, then so is our goal (33). We now proceed to discuss the technique of derivative calculation which is based on employment of ***log-derivative trick***:

Proposition 17. *For arbitrary distribution $\pi(a)$ parameterized by θ:*

$$\nabla_\theta \pi(a) = \pi(a) \nabla_\theta \log \pi(a) \tag{34}$$

In most general form, this trick allows us to derive the gradient of expectation of an arbitrary function $f(a, \theta) : \mathcal{A} \times \Theta \to \mathbb{R}$, differentiable by θ, with respect to some distribution $\pi_\theta(a)$, also parameterized by θ:

$$\begin{aligned}
\nabla_\theta \mathbb{E}_{a\sim\pi_\theta(a)} f(a,\theta) &= \nabla_\theta \int_{\mathcal{A}} \pi_\theta(a) f(a,\theta) da = \\
&= \int_{\mathcal{A}} \nabla_\theta \left[\pi_\theta(a) f(a,\theta)\right] da = \\
\{\text{product rule}\} &= \int_{\mathcal{A}} \left[\nabla_\theta \pi_\theta(a) f(a,\theta) + \pi_\theta(a) \nabla_\theta f(a,\theta)\right] da = \\
&= \int_{\mathcal{A}} \nabla_\theta \pi_\theta(a) f(a,\theta) da + \mathbb{E}_{\pi_\theta(a)} \nabla_\theta f(a,\theta) = \\
\{\text{log-derivative trick (34)}\} &= \int_{\mathcal{A}} \pi_\theta(a) \nabla_\theta \log \pi_\theta(a) f(a,\theta) da + \mathbb{E}_{\pi_\theta(a)} \nabla_\theta f(a,\theta) = \\
&= \mathbb{E}_{\pi_\theta(a)} \nabla_\theta \log \pi_\theta(a) f(a,\theta) + \mathbb{E}_{\pi_\theta(a)} \nabla_\theta f(a,\theta)
\end{aligned}$$

This technique can be applied sequentially (to expectations over $\pi_\theta(a_0 \mid s_0)$, $\pi_\theta(a_1 \mid s_1)$ and so on) to obtain the gradient $\nabla_\theta J(\pi_\theta)$.

Proposition 18. ***(Policy Gradient Theorem)*** *[22] For any MDP and differentiable policy π_θ the gradient of objective (33) is*

$$\nabla_\theta J(\theta) = \mathbb{E}_{\mathcal{T}\sim\pi_\theta} \sum_{t=0} \nabla_\theta \log \pi_\theta(a_t \mid s_t) Q^\pi(s_t, a_t) \tag{35}$$

For future references, we require another form of formula (35), which provides another point of view. For this purpose, let us define a state visitation frequency:

Definition 10. For given MDP and given policy π its ***state visitation frequency*** is defined by

$$d_\pi(s) \coloneqq \sum_{t=0} \mathcal{P}(s_t = s)$$

where s_t are taken from trajectories $\mathcal{T}$ sampled using given policy π.

State visitation frequencies, if normalized, represent a marginalized probability for agent to land in a given state s. It is rarely attempted to be learned, but it assists theoretical study as it allows us to rewrite expectations over trajectories with separated intrinsic and extrinsic randomness of the decision making process:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim d_\pi(s)} \mathbb{E}_{a\sim\pi(a|s)} \nabla_\theta \log \pi_\theta(a \mid s) Q^\pi(s, a) \tag{36}$$

This form is equivalent to (35) as sampling a trajectory and going through all visited states induces the same distribution as defined in $d_\pi(s)$.

Now, although we acquired an explicit form of objective's gradient, we are able to compute it only approximately, using Monte-Carlo estimation for expectations via sampling one or several trajectories. Second form of gradient (36) reveals that it is possible to use roll-outs of trajectories without waiting for episode ending, as the states for the roll-outs come from the same distribution as they would for complete episode trajectories. The essential thing is that exactly the policy $\pi(\theta)$ must be used for sampling to obtain unbiased Monte-Carlo estimation (otherwise state visitation frequency $d_\pi(s)$ is different). These features are commonly underlined by notation $\mathbb{E}_\pi$, which is a shorter form of $\mathbb{E}_{s\sim d_\pi(s)}\mathbb{E}_{a\sim\pi(a|s)}$. When convenient, we will use it to reduce the gradient to a shorter form:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi(\theta)} \nabla_\theta \log \pi_\theta(a \mid s) Q^\pi(s,a) \tag{37}$$

Second important thing worth mentioning is that $Q^\pi(s,a)$ is essentially present in the gradient. Remark that it is never available to the algorithm and must also be somehow estimated.

5.2. REINFORCE

REINFORCE [27] provides a straightforward approach to approximately calculate the gradient (35) in episodic case using Monte-Carlo estimation: N games are played and Q-function under policy π is approximated with corresponding return:

$$Q^\pi(s,a) = \mathbb{E}_{\mathcal{T}\sim\pi_\theta|s,a} R(\mathcal{T}) \approx R(\mathcal{T}), \quad \mathcal{T} \sim \pi_\theta \mid s,a$$

The resulting formula is therefore the following:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{\mathcal{T}}^{N} \sum_{t=0} \left[\nabla_\theta \log \pi_\theta(a_t \mid s_t) \left(\sum_{t'=t} \gamma^{t'-t} r_{t'+1} \right) \right] \tag{38}$$

This estimation is unbiased as both approximation of Q^π and approximation of expectation over trajectories are done using Monte-Carlo. Given that estimation of gradient is unbiased, stochastic gradient ascent or more advanced stochastic optimization techniques are known to converge to local optimum.

From theoretical point of view REINFORCE can be applied straightforwardly for any parametric family $\pi_\theta(a \mid s)$ including neural networks. Yet the enormous time required for convergence and the problem of stucking in local optimums make this naive approach completely impractical.

The main source of problems is believed to be the ***high variance*** of gradient estimation (38), as the convergence rate of stochastic gradient descent directly depends on the variance of gradient estimation.

The standard technique of variance reduction is an introduction of ***baseline***. The idea is to add some term that will not affect the expectation, but may affect the variance. One such baseline can be derived using following reasoning: for any distribution it is true that $\int_{\mathcal{A}} \pi_\theta(a \mid s) da = 1$. Taking the gradient ∇_θ from both sides, we obtain:

$$\begin{aligned} 0 = \int_{\mathcal{A}} \nabla_\theta \pi_\theta(a \mid s) da = \\ \{\text{log-derivative trick (34)}\} = \int_{\mathcal{A}} \pi_\theta(a \mid s) \nabla_\theta \log \pi_\theta(a \mid s) da = \\ = \mathbb{E}_{\pi_\theta(a|s)} \nabla_\theta \log \pi_\theta(a \mid s) \end{aligned}$$

Multiplying this expression on some constant, we can scale this baseline:

$$\mathbb{E}_{\pi_\theta(a|s)} \operatorname{const}(a) \nabla_\theta \log \pi_\theta(a \mid s) = 0$$

Notice that the constant here must be independent of a, but may depend on s. Application of this technique to our case provides the following result[23]:

[23] this result can be generalized by introducing different baselines for estimation of different components of $\nabla_\theta J(\theta)$.

Proposition 19. *For any arbitrary function $b(s): \mathcal{S} \to \mathbb{R}$, called* **baseline***:*

$$\nabla_\theta J(\theta) = \mathbb{E}_{\mathcal{T}\sim\pi_\theta} \sum_{t=0} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \left(Q^\pi(s_t, a_t) - b(s_t)\right)$$

Selection of the baseline is up to us as long as it does not depend on actions a_t. The intent is to choose it in a way that minimizes the variance.

It is believed that high variance of (38) originates from multiplication of $Q^\pi(s,a)$, which may have arbitrary scale (e. .g. in a range $[100, 200]$) while $\nabla_\theta \log \pi_\theta(a_t \mid s_t)$ naturally has varying signs[24]. To reduce the variance, the baseline must be chosen so that absolute values of expression inside the expectation are shifted towards zero. Wherein the optimal baseline is provided by the following theorem:

Proposition 20. *The solution for*

$$\mathbb{V}_{\mathcal{T}\sim\pi_\theta} \sum_{t=0} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \left(Q^\pi(s_t, a_t) - b(s_t)\right) \to \min_{b(s)}$$

is given by

$$b(s) = \frac{\mathbb{E}_{a\sim\pi_\theta(a|s)} \|\nabla_\theta \log \pi_\theta(a \mid s)\|_2^2 Q^\pi(s,a)}{\mathbb{E}_{a\sim\pi_\theta(a|s)} \|\nabla_\theta \log \pi_\theta(a \mid s)\|_2^2} \tag{39}$$

As can be seen, optimal baseline calculation involves expectations which again can only be computed (in most cases) using Monte-Carlo (both for numerator and denominator). For that purpose, for every visited state s estimations of $Q^\pi(s,a)$ are needed for all (or some) actions a, as otherwise estimation of baseline will coincide with estimation of $Q^\pi(s,a)$ and collapse gradient to zero. Practical utilization of result (39) is to consider a constant baseline independent of s with similar optimal form:

$$b = \frac{\mathbb{E}_{\mathcal{T}\sim\pi_\theta} \sum_{t=0} \|\nabla_\theta \log \pi_\theta(a_t \mid s_t)\|_2^2 Q^\pi(s_t, a_t)}{\mathbb{E}_{\mathcal{T}\sim\pi_\theta} \sum_{t=0} \|\nabla_\theta \log \pi_\theta(a_t \mid s_t)\|_2^2}$$

which can be profitably estimated via Monte-Carlo.

Utilization of some kind of baseline, not necessarily optimal, is known to significantly reduce the variance of gradient estimation and is an essential part of any policy gradient method. The final step to make this family of algorithms applicable when using deep neural networks is to reduce variance of Q^π estimation by employing RL task structure like it was done in value-based methods.

5.3. Advantage Actor-Critic (A2C)

Suppose that in optimal baseline formula (39) it happens that $\|\nabla_\theta \log \pi_\theta(a \mid s)\|_2^2 = \mathbf{const}(a)$. Though in reality this is actually not true, under this circumstance the optimal baseline formula significantly reduces and unravels a close-to-optimal but simple form of baseline:

$$b(s) = \mathbb{E}_{a\sim\pi_\theta(a|s)} Q^\pi(s,a) = V^\pi(s)$$

Substituting this baseline into gradient formula (37) and recalling the definition of advantage function (14), the gradient can now be rewritten as follows:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi(\theta)} \nabla_\theta \log \pi_\theta(a \mid s) A^\pi(s,a) \tag{40}$$

This representation of gradient is used as the basement for most policy gradient algorithms as it offers lower variance while selecting the baseline expressed in terms of value functions which can be efficiently learned similar to how it was done in value-based methods. Such algorithms are usually named Actor-Critic as they consist of two neural networks: $\pi_\theta(a \mid s)$, representing a policy, called an ***actor***, and $V^\pi_\phi(s)$ with parameters ϕ, approximately estimating actor's performance, called a ***critic***. Note that the choice of value function to learn can be arbitrary; it is possible to learn Q^π or A^π instead, as all of them are deeply interconnected. Value function V^π is chosen as the simplest one since it depends only on state and thus is hoped to be easier to learn.

[24] this follows, for example, from baseline derivation

Having a critic $V_\phi^\pi(s)$, Q-function can be approximated in a following way:

$$Q^\pi(s,a) \approx r' + \gamma V^\pi(s') \approx r' + \gamma V_\phi^\pi(s')$$

First approximation is done using Monte-Carlo, while second approximation inevitably introduces bias. Important thing to notice is that at this moment our gradient estimation stops being unbiased and all theoretical guarantees of converging are once again lost.

Advantage function therefore can be obtained according to the definition:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s) \approx r' + \gamma V_\phi^\pi(s') - V_\phi^\pi(s) \tag{41}$$

Note that biased estimation of baseline doesn't make gradient estimation biased by itself, as baseline can be an arbitrary function of state. All bias introduction happens inside the approximation of Q^π. It is possible to use critic only for baseline, which allows complete avoidance of bias, but then the only way to estimate Q^π is via playing several games and using corresponding returns, which suffers from higher variance and low sample efficiency.

The logic behind training procedure for the critic is taken from value-based methods: for given policy π its value function can be obtained using point iteration for solving

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(a|s)}\mathbb{E}_{s'\sim p(s'|s,a)}\left[r' + \gamma V^\pi(s')\right]$$

Similar to DQN, on each update a target is computed using current approximation

$$y = r' + \gamma V_\phi^\pi(s')$$

and then MSE is minimized to move values of $V_\phi^\pi(s)$ towards the guess.

Notice that to compute the target for critic we require samples from the policy π which is being evaluated. Although actor evolves throughout optimization process, we assume that one update of policy π does not lead to significant change of true V^π and thus our critic, which approximates value function for older version of policy, is close enough to construct the target. But if samples from, for example, old policy are used to compute the guess, the step of critic update will correspond to learning the value function for old policy other than current. Essentially, this means that both actor and critic training procedures require samples from current policy π, making Actor-Critic algorithm ***on-policy*** by design. Consequently, samples that were collected on previous update iterations become useless and can be forgotten. This is the key reason why policy gradient algorithms are usually less sample-efficient than value-based.

Now as we have an approximation of value function, advantage estimation can be done using one-step transitions (41). As the procedure of training an actor, i. .e. gradient estimation (40), also does not demand sampling the whole trajectory, each update now requires only a small roll-out to be sampled. The amount of transitions in the roll-out corresponds to the size of mini-batch.

The problem with roll-outs is that the data is obviously not i. i. d., which is crucial for training networks. In value-based methods, this problem was solved with experience replay, but in policy gradient algorithms it is essential to collect samples from scratch after each update of the networks parameters. The practical solution for simulated environments is to launch several instances of environment (for example, on different cores of multiprocessor) in parallel threads and have several parallel interactions. After several steps in each environment, the batch for update is collected by uniting transitions from all instances and one synchronous[25] update of networks parameters θ and ϕ is performed.

One more optimization that can be done is to partially share weights of networks θ and ϕ. It is justified as first layers of both networks correspond to basic features extraction and these features are likely to be the same for optimal policy and value function. While it reduces the number of training parameters almost twice, it might destabilize learning process as the scales of gradient (40) and gradient of critic's MSE loss may be significantly different, so they should be balanced with additional hyperparameter.

[25] there is also an asynchronous modification of advantage actor critic algorithm (A3C) which accelerates the training process by storing a copy of network for each thread and performing weights synchronization from time to time.

Algorithm 6: Advantage Actor-Critic (A2C)

Hyperparameters: B — batch size, V_ϕ^* — critic neural network, π_θ — actor neural network, α — critic loss scaling, SGD optimizer.

Initialize weights θ, ϕ arbitrary
On each step:

1. obtain a roll-out of size B using policy $\pi(\theta)$
2. for each transition T from the roll-out compute advantage estimation:
$$A^\pi(T) = r' + \gamma V_\phi^\pi(s') - V_\phi^\pi$$
3. compute target (detached from computational graph to prevent backpropagation):
$$y(T) = r' + \gamma V_\phi^\pi(s')$$
4. compute critic loss:
$$\text{Loss} = \frac{1}{B}\sum_T \left(y(T) - V_\phi^\pi\right)^2$$
5. compute critic gradients:
$$\nabla^{\text{critic}} = \frac{\partial \text{Loss}}{\partial \phi}$$
6. compute actor gradient:
$$\nabla^{\text{actor}} = \frac{1}{B}\sum_T \nabla_\theta \log \pi_\theta(a \mid s) A^\pi(T)$$
7. make a step of gradient descent using $\nabla^{\text{actor}} + \alpha\nabla^{\text{critic}}$

5.4. Generalized Advantage Estimation (GAE)

There is a design dilemma in Advantage Actor Critic algorithm concerning the choice whether to use the critic to estimate $Q^\pi(s,a)$ and introduce bias into gradient estimation or to restrict critic employment only for baseline and cause higher variance with necessity of playing the whole episodes for each update step.

Actually, the range of possibilities is wider. Since Actor-Critic is an on-policy algorithm by design, we are free to use N-step approximations instead of one-step: using

$$Q^\pi(s,a) \approx \sum_{n=0}^{N-1} \gamma^n r^{(n+1)} + \gamma^N V^\pi\left(s^{(N)}\right)$$

we can define N-step advantage estimator as

$$A^\pi_{(N)}(s,a) \coloneqq \sum_{n=0}^{N-1} \gamma^n r^{(n+1)} + \gamma^N V_\phi^\pi\left(s^{(N)}\right) - V_\phi^\pi(s)$$

For $N = 1$ this estimation corresponds to Actor-Critic one-step estimation with high bias and low variance. For $N = \infty$ it yields the estimator with critic used only for baseline with no bias and high variance. Intermediate values correspond to something in between. Note that to use N-step advantage estimation we have to perform N steps of interaction after given state-action pair.

Usually finding a good value for N as hyperparameter is difficult as its «optimal» value may float throughout the learning process. In Generalized Advantage Estimation (GAE) [18] it is proposed to construct an ensemble out of different N-step advantage estimators using exponential smoothing with some hyperparameter λ:

$$A^\pi_{\text{GAE}}(s,a) \coloneqq (1-\lambda)\left(A^\pi_{(1)}(s,a) + \lambda A^\pi_{(2)}(s,a) + \lambda^2 A^\pi_{(3)}(s,a) + \dots\right) \tag{42}$$

Here the parameter $\lambda \in [0, 1]$ allows smooth control over bias-variance trade-off: $\lambda = 0$ corresponds to Actor-Critic with higher bias and lower variance while $\lambda \to 1$ corresponds to REINFORCE with no bias and high variance. But unlike N as hyperparameter, it uses mix of different estimators in intermediate case.

GAE proved to be a convenient way how more information can be obtained from collected roll-out in practice. Instead of waiting for episode termination to compute (42) we may use «truncated» GAE which ensembles only those N-step advantage estimators that are available:

$$A^{\pi}_{\mathbf{trunc.GAE}}(s, a) \coloneqq \frac{A^{\pi}_{(1)}(s, a) + \lambda A^{\pi}_{(2)}(s, a) + \lambda^2 A^{\pi}_{(3)}(s, a) + \cdots + \lambda^{N-1} A^{\pi}_{(N)}(s, a)}{1 + \lambda + \lambda^2 + \cdots + \lambda^{N-1}}$$

Note that the amount N of available estimators may be different for different transitions from roll-out: if we performed K steps of interaction in some instance of environment starting from some state-action pair s, a, we can use $N = K$ step estimators; for next state-action pair s', a' we have only $N = K - 1$ transitions and so on, while the last state-action pair s^{N-1}, a^{N-1} can be estimated only using $A^{\pi}_{(1)}$ as only $N = 1$ following transition is available. Although different transitions are estimated with different precision (leading to different bias and variance), this approach allows to use all available information for each transition and utilize multi-step approximations without dropping last transitions of roll-outs used only for target computation.

5.5. Natural Policy Gradient (NPG)

In this section we discuss the motivation and basic principles behind the idea of natural gradient descent, which we will require for future references.

The standard gradient descent optimization method is known to be extremely sensitive to the choice of parametrization. Suppose we attempt to solve the following optimization task:

$$f(q) \to \min_{q}$$

where q is a distribution and F is arbitrary differentiable function. We often restrict q to some parametric family and optimize similar objective, but with respect to some vector of parameters θ as unknown variable:

$$f(q_\theta) \to \min_{\theta}$$

Classic example of such problem is maximum likelihood task when we try to fit the parameters of our model to some observed data. The problem is that when using standard gradient descent both the convergence rate and overall performance of optimization method substantially depend on the choice of parametrization q_θ. The problem holds even if we fix specific distribution family as many distribution families allow different parametrizations.

To see why gradient descent is parametrization-sensitive, consider the model which is used at some current point θ_k to determine the direction of next optimization step:

$$\begin{cases} f(q_{\theta_k}) + \langle \nabla_\theta f(q_{\theta_k}), \delta\theta \rangle \to \min\limits_{\delta\theta} \\ \|\delta\theta\|_2^2 < \alpha_k \end{cases}$$

where α_k is learning rate at step k. Being first-order method, gradient descent constructs a «model» which approximates F locally around θ_k using first-order Taylor expansion and employs standard Euclidean metric to determine a region of trust for this model. Then this surrogate task is solved analytically to obtain well-known update formula:

$$\delta\theta \propto -\nabla_\theta f(q_{\theta_k})$$

The issue arises from reliance on Eucliden metric in the space of parameters. In most parametrizations, small changes in parameters space do not guarantee small change in distribution space and vice versa: some small changes in distribution may demand big steps in parameters space[26].

Natural gradient proposes to use another metric, which achieves invariance to parametrization of distribution q using the properties of Fisher matrix:

[26] classic example is that $\mathcal{N}(0, 100)$ is similar to $\mathcal{N}(1, 100)$ while $\mathcal{N}(0, 0.1)$ is completely different from $\mathcal{N}(1, 0.1)$, although Euclidean distance in parameter space is the same for both pairs.

Definition 11. For distribution q_θ ***Fisher matrix*** $F_q(\theta)$ is defined as

$$F_q(\theta) \coloneqq \mathbb{E}_{x\sim q} \nabla_\theta \log q_\theta(x) (\nabla_\theta \log q_\theta(x))^T$$

Note that Fisher matrix depends on parametrization. Yet for any parametrization it is guaranteed to be positive semi-definite by definition. Moreover, it induces a so-called ***Riemannian metric***[27] in the space of parameters:

$$d(\theta_1, \theta_2)^2 \coloneqq (\theta_2 - \theta_1)^T F_q(\theta_1)(\theta_2 - \theta_1)$$

In natural gradient descent it is proposed to use this metric instead of Euclidean:

$$\begin{cases} f(q_{\theta_k}) + \langle \nabla_\theta f(q_{\theta_k}), \delta\theta \rangle \to \min\limits_{\delta\theta} \\ \delta\theta^T F_q(\theta_k) \delta\theta < \alpha_k \end{cases}$$

This surrogate task can be solved analytically to obtain the following optimization direction:

$$\delta\theta \propto -F_q(\theta_k)^{-1} \nabla_\theta f(q_{\theta_k}) \tag{43}$$

The direction of gradient descent is corrected by Fisher matrix which concerns the scale across different axes. This direction, specified by $F_q(\theta_k)^{-1}\nabla_\theta f(q_{\theta_k})$, is called ***natural gradient***.

Let's discuss why this new metric really provides us invariance to distribution parametrization. We already obtained natural gradient for q being parameterized by θ (43). Assume that we have another parametrization q_ν. These new parameters ν are somehow related to θ; we suppose there is some functional dependency $\theta(\nu)$, which we assume to be differentiable with jacobian J. In this notation:

$$\delta\theta = J\delta\nu, \qquad J_{ij} \coloneqq \frac{\partial\theta_i}{\partial\nu_j} \tag{44}$$

The central property of Fisher matrix, which provides the desired invariance, is the following:

Proposition 21. *If $\theta = \theta(\nu)$ with jacobian J, then reparametrization formula for Fisher matrix is*

$$F_q(\nu) = J^T F_q(\theta) J \tag{45}$$

Now it can be derived that natural gradient for parametrization with ν is the same as for θ. If we want to calculate natural gradient in terms of ν, then our step is, according to (44):

$$\begin{aligned} \delta\theta = J\delta\nu = \\ \{\text{natural gradient in terms of } \nu\} &\propto J F_q(\nu_k)^{-1} \nabla_\nu f(q_{\nu_k}) = \\ \{\text{Fisher matrix reparametrization (45)}\} &= J \left(J^T F_q(\theta_k) J\right)^{-1} \nabla_\nu f(q_{\nu_k}) \\ \{\text{chain rule}\} &= J \left(J^T F_q(\theta_k) J\right)^{-1} \nabla_\nu \theta(\nu_k)^T \nabla_\theta f(q_{\theta_k}) = \\ &= J J^{-1} F_q(\theta_k)^{-1} J^{-T} J^T \nabla_\theta f(q_{\theta_k}) = \\ &= F_q(\theta_k)^{-1} \nabla_\theta f(q_{\theta_k}) \end{aligned}$$

which can be seen to be the same as in (43).

Application of natural gradient descent in DRL setting is complicated in practice. Theoretically, the only change that must be done is scaling of gradient using inverse Fisher matrix (43). Yet, Fisher matrix requires n^2 memory and $\mathcal{O}(n^3)$ computational costs for inversion where n is the number of parameters. For neural networks this causes the same complications as the application of second-order optimization methods.

K-FAC optimization method provides a specific approximation form of Fisher matrix for neural networks with linear layers which can be efficiently computed, stored and inverted. Usage of K-FAC approximation allows to compute natural gradient directly using (43).

[27] in Euclidean space the general form of scalar product is $\langle x, y \rangle \coloneqq x^T G y$, where G is fixed positive semi-definite matrix. The metric induced by this scalar product is correspondingly $d(x, y)^2 \coloneqq (y-x)^T G (y-x)$. The difference in Riemannian space is that G, called ***metric tensor***, depends on x, so the relative distance may vary for different points. It is used to describe the distances between points on manifolds and holds important properties which Fisher matrix inherits as metric tensor for distribution space.

5.6. Trust-Region Policy Optimization (TRPO)

The main drawback of Actor-Critic algorithm is believed to be the abandonment of experience that was used for previous updates. As the number of updates required is usually huge, this is considered to be a substantial loss of information. Yet, it is not clear how this information can be effectively used for newer updates.

Suppose we want to make an update of $\pi(\theta)$, but using samples collected by some π^{old}. The straightforward approach is importance sampling technique, which naive application to gradient formula (40) yields the following result:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\mathcal{T}\sim\pi^{\text{old}}} \frac{\mathcal{P}(\mathcal{T} \mid \pi(\theta))}{\mathcal{P}(\mathcal{T} \mid \pi^{\text{old}})} \sum_{t=0} \nabla_\theta \log \pi_\theta(a_t \mid s_t) A^\pi(s_t, a_t)$$

The emerged importance sampling weight is actually computable as transition probabilities cross out:

$$\frac{\mathcal{P}(\mathcal{T} \mid \pi(\theta))}{\mathcal{P}(\mathcal{T} \mid \pi^{\text{old}})} = \frac{\prod_{t=1} \pi_\theta(a_t \mid s_t)}{\prod_{t=1} \pi^{\text{old}}(a_t \mid s_t)}$$

The problem with this coefficient is that it tends either to be exponentially small or to explode. Even with some heuristic normalization of coefficients the batch gradient would become dominated by one or several transitions and destabilize the training procedure by introducing even more variance.

Notice that application of importance sampling to another representation of gradient (37) yields seemingly different result:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi^{\text{old}}} \frac{d_{\pi(\theta)}(s)}{d_{\pi^{\text{old}}}(s)} \frac{\pi_\theta(a \mid s)}{\pi^{\text{old}}(a \mid s)} \nabla_\theta \log \pi_\theta(a \mid s) A^\pi(s, a) \tag{46}$$

Here we avoided common for the whole trajectories importance sampling weights by using the definition of state visitation frequencies. But this result is even less practical as these frequencies are unknown to us.

The first key idea behind the theory concerning this problem is that may be these importance sampling coefficients behave more stable if the policies π^{old} and $\pi(\theta)$ are in some terms «close». Intuitively, in this case $\frac{d_{\pi(\theta)}(s)}{d_{\pi^{\text{old}}}(s)}$ of formula (46) is close to 1 as state visitation frequencies are similar, and the remained importance sampling coefficient becomes acceptable in practice. And if some two policies are similar, their values of our objective (2) are probably close too.

For any two policies, π and π^{old}:

$$\begin{aligned}
J(\pi) - J(\pi^{\text{old}}) &= \mathbb{E}_{\mathcal{T}\sim\pi} \sum_{t=0} \gamma^t r(s_t) - J(\pi^{\text{old}}) = \\
&= \mathbb{E}_{\mathcal{T}\sim\pi} \sum_{t=0} \gamma^t r(s_t) - V^{\pi^{\text{old}}}(s_0) = \\
&= \mathbb{E}_{\mathcal{T}\sim\pi} \left[\sum_{t=0} \gamma^t r(s_t) - V^{\pi^{\text{old}}}(s_0) \right] = \\
\{\text{trick } \textstyle\sum_{t=0}^{\infty} (a_{t+1} - a_t) = -a_0\text{[28]}\} &= \mathbb{E}_{\mathcal{T}\sim\pi} \left[\sum_{t=0} \gamma^t r(s_t) + \sum_{t=0} \left[\gamma^{t+1} V^{\pi^{\text{old}}}(s_{t+1}) - \gamma^t V^{\pi^{\text{old}}}(s_t) \right] \right] = \\
\{\text{regroup}\} &= \mathbb{E}_{\mathcal{T}\sim\pi} \sum_{t=0} \gamma^t \left(r(s_t) + \gamma V^{\pi^{\text{old}}}(s_{t+1}) - V^{\pi^{\text{old}}}(s_t) \right) = \\
\{\text{by definition (3)}\} &= \mathbb{E}_{\mathcal{T}\sim\pi} \sum_{t=0} \gamma^t \left(Q^{\pi^{\text{old}}}(s_t, a_t) - V^{\pi^{\text{old}}}(s_t) \right) \\
\{\text{by definition (14)}\} &= \mathbb{E}_{\mathcal{T}\sim\pi} \sum_{t=0} \gamma^t A^{\pi^{\text{old}}}(s_t, a_t)
\end{aligned}$$

The result obtained above is often referred to as ***relative policy performance identity*** and is actually very interesting: it states that we can substitute reward with advantage function of arbitrary policy and that will shift the objective by the constant.

We will require this identity rewritten in terms of state visitation frequencies. To do so, it is convenient to define discounted version of state visitations distribution:

[28] and if MDP is episodic, for terminal states $V^{\pi^{\text{old}}}(s_T) = 0$ by definition.

Definition 12. For given MDP and given policy π its ***discounted state visitation frequency*** is defined by

$$d(s \mid \pi) \coloneqq (1-\gamma)\sum_{t=0}\gamma^t \mathcal{P}(s_t = s)$$

where s_t are taken from trajectories $\mathcal{T}$ sampled using given policy π.

Using frequency as unnormalized state visitation distribution, relative policy performance identity can be rewritten as

$$J(\pi) - J(\pi^{\text{old}}) = \frac{1}{1-\gamma}\mathbb{E}_{s\sim d(s|\pi)}\mathbb{E}_{a\sim\pi(a|s)}A^{\pi^{\text{old}}}(s,a)$$

Now assume we want to optimize parameters θ of policy π while using data collected by π^{old}: applying importance sampling in the same manner:

$$J(\pi_\theta) - J(\pi^{\text{old}}) = \frac{1}{1-\gamma}\mathbb{E}_{s\sim d(s|\pi^{\text{old}})}\frac{d(s \mid \pi_\theta)}{d(s \mid \pi^{\text{old}})}\mathbb{E}_{a\sim\pi^{\text{old}}(a|s)}\frac{\pi_\theta(a \mid s)}{\pi^{\text{old}}(a \mid s)}A^{\pi^{\text{old}}}(s,a)$$

As we have in mind the idea of π^{old} being close to π_θ, the question is how well this identity can be approximated if we assume $d(s \mid \pi_\theta) = d(s \mid \pi^{\text{old}})$. Under this assumption:

$$J(\pi_\theta) - J(\pi^{\text{old}}) \approx L_{\pi^{\text{old}}}(\theta) \coloneqq \frac{1}{1-\gamma}\mathbb{E}_{s\sim d(s|\pi^{\text{old}})}\mathbb{E}_{a\sim\pi^{\text{old}}(a|s)}\frac{\pi_\theta(a \mid s)}{\pi^{\text{old}}(a \mid s)}A^{\pi^{\text{old}}}(s,a)$$

The point is that interaction using π^{old} corresponds to sampling from the expectations presented in $L_{\pi^{\text{old}}}(\theta)$:

$$L_{\pi^{\text{old}}}(\theta) = \mathbb{E}_{\pi^{\text{old}}}\frac{\pi_\theta(a \mid s)}{\pi^{\text{old}}(a \mid s)}A^{\pi^{\text{old}}}(s,a)$$

The approximation quality of $L_{\pi^{\text{old}}}(\theta)$ can be described by the following theorem:

Proposition 22.

[17] $\left|J(\pi_\theta) - J(\pi^{\text{old}}) - L_{\pi^{\text{old}}}(\theta)\right| \le C\max_s \mathbf{KL}(\pi^{\text{old}} \parallel \pi_\theta)[s]$

where C is some constant and $\mathbf{KL}(\pi^{\text{old}} \parallel \pi_\theta)[s]$ is a shorten notation for $\mathbf{KL}(\pi^{\text{old}}(a \mid s) \parallel \pi_\theta(a \mid s))$.

There is an important corollary of proposition 22:

$$J(\pi_\theta) - J(\pi^{\text{old}}) \ge L_{\pi^{\text{old}}}(\theta) - C\max_s \mathbf{KL}(\pi^{\text{old}} \parallel \pi_\theta)[s]$$

which not only states that expression on the right side represents a ***lower bound***, but also that the optimization procedure

$$\theta_{k+1} = \underset{\theta}{argmax}\left[L_{\pi_{\theta_k}}(\theta) - C\max_s \mathbf{KL}(\pi_{\theta_k} \parallel \pi_\theta)[s]\right] \tag{47}$$

will yield a policy with guaranteed monotonic improvement[29].

In practice there are several obstacles which preserve us from obtaining such procedure. First of all, our advantage function estimation is never precise. Secondly, it is hard to estimate precise value of constant C. One last obstacle is that it is not clear how to calculate $\mathbf{KL}$-divergence in its maximal form (with $\mathbf{max}$ taken across all states).

In Trust-Region policy optimization [17] the idea of practical algorithm, approximating procedure (47), is analyzed. To address the last issue, the naive approximation is proposed to substitute $\mathbf{max}$ with averaging across states[30]:

$$\max_s \mathbf{KL}(\pi^{\text{old}} \parallel \pi_\theta)[s] \approx \mathbb{E}_{s\sim d(s|\pi^{\text{old}})}\mathbf{KL}(\pi^{\text{old}} \parallel \pi_\theta)[s]$$

[29] the maximum of lower bound is non-negative as its value for $\theta = \theta_k$ equals zero, which causes $J(\pi_{k+1}) - J(\pi_k) \ge 0$

[30] the distribution from which the states come is set to be $d(s \mid \pi^{\text{old}})$ for convenience as this is the distribution from which they come in $L_{\pi^{\text{old}}}(\theta)$.

The second step of TRPO is to rewrite the task of unconstrained minimization (47) in equivalent constrained («trust-region») form[31] to incorporate the unknown constant C into learning rate:

$$\begin{cases} L_{\pi^{\text{old}}}(\theta) \to \max\limits_{\theta} \\ \mathbb{E}_{s\sim d(s|\pi^{\text{old}})}\, \mathbf{KL}(\pi^{\text{old}} \parallel \pi_\theta)[s] < C \end{cases} \tag{48}$$

Note that this rewrites an update iteration in terms of optimization methods: while $L_{\pi^{\text{old}}}(\theta)$ is an approximation of true objective $J(\pi_\theta) - J(\pi^{\text{old}})$, the constraint sets the region of trust to the surrogate. Remark that constraint is actually a divergence in policy space, i. e. it is very similar to a metric in the space of distributions while the surrogate is a function of the policy and depends on parameters θ only through π_θ.

To solve the constrained problem (48), the technique from convex optimization is used. Assume that π^{old} is a current policy and we want to update its parameters θ_k. Then the objective of (48) is modeled using first-order Taylor expansion around θ_k while constraint is modeled using second-order [32] Taylor approximation:

$$\begin{cases} L_{\pi^{\text{old}}}(\theta_k + \delta\theta) \approx \langle \nabla_\theta\, L_{\pi^{\text{old}}}(\theta)|_{\theta_k}, \delta\theta \rangle \to \max\limits_{\delta\theta} \\ \mathbb{E}_{s\sim d(s|\pi^{\text{old}})}\, \mathbf{KL}(\pi^{\text{old}} \parallel \pi_{\theta_k+\delta\theta}) \approx \frac{1}{2}\mathbb{E}_{s\sim d(s|\pi^{\text{old}})} \delta\theta^T\, \nabla^2_\theta\, \mathbf{KL}(\pi^{\text{old}} \parallel \pi_\theta)\big|_{\theta_k}\, \delta\theta < C \end{cases}$$

It turns out, that this model is equivalent to natural policy gradient, discussed in sec. 5.5:

Proposition 23.

$$\nabla^2_\theta\, \mathbf{KL}(\pi_\theta \parallel \pi^{\text{old}})[s]\big|_{\theta_k} = F_{\pi(a|s)}(\theta)$$

so $\mathbf{KL}$-divergence constraint can be approximated with metric induced by Fisher matrix. Moreover, the gradient of surrogate function is

$$\nabla_\theta L_{\pi^{\text{old}}}(\theta)|_{\theta_k} = \mathbb{E}_{\pi^{\text{old}}} \frac{\nabla_\theta \pi_\theta(a \mid s)|_{\theta_k}}{\pi^{\text{old}}(a \mid s)} A^{\pi^{\text{old}}}(s,a) =$$

$$\{\pi^{\text{old}} = \pi_{\theta_k}\} = \mathbb{E}_{\pi^{\text{old}}} \nabla_\theta \log \pi_{\theta_k}(a \mid s) A^{\pi^{\text{old}}}(s,a)$$

which is exactly an Actor-Critic gradient. Therefore the formula of update step is given by

$$\delta\theta \propto -F_\pi(\theta)^{-1} \nabla_\theta L_{\pi^{\text{old}}}(\theta)$$

where $\nabla_\theta L_{\pi^{\text{old}}}(\theta)$ coincides with standard policy gradient, and $F_\pi(\theta)$ is hessian of $\mathbf{KL}$-divergence:

$$F_\pi(\theta) := \mathbb{E}_{s\sim d(s|\pi^{\text{old}})}\, \nabla^2_\theta\, \mathbf{KL}(\pi^{\text{old}} \parallel \pi_\theta)\big|_{\theta_k}$$

In practical implementations $\mathbf{KL}$-divergence can be Monte-Carlo estimated using collected roll-out. The size of roll-out must be significantly bigger than in Actor-Critic to achieve sufficient precision of hessian estimation. Then to obtain a direction of optimization step the following system of linear equations

$$F_\pi(\theta)\delta\theta = -\nabla_\theta L_{\pi^{\text{old}}}(\theta)$$

is solved using a ***conjugate gradients*** method which is able to work with Hessian-vector multiplication procedure instead of requiring to calculate $F_\pi(\theta)$ explicitly.

TRPO also accompanies the update step with a ***line-search*** procedure which dynamically adjusts step length using standard backtracking heuristic. As TRPO intuitively seeks for policy improvement on each step, the idea is to check whether the lower bound (47) is positive after the biggest step allowed according to $\mathbf{KL}$-constraint and reduce the step size until it becomes positive.

Unlike Actor-Critic, TRPO performs extremely expensive complicated update steps but requires relatively small number of iterations in return. Of course, due to many approximations done, the overall procedure is only a resemblance of theoretically-justified iterations (47) providing improvement guarantees.

[31] the unconstrained objective is Lagrange function for constrained form

[32] as first-order term is zero

5.7. Proximal Policy Optimization (PPO)

Proximal Policy Optimization [19] proposes alternative heuristic way of performing lower bound (47) optimization which demonstrated encouraging empirical results.

PPO still substitutes $\max\limits_{s} \mathbf{KL}$ on average, but leaves the surrogate in unconstrained form, suggesting to treat unknown constant C as a hyperparameter:

$$\mathbb{E}_{\pi^{\text{old}}}\left[\frac{\pi_\theta(a \mid s)}{\pi^{\text{old}}(a \mid s)} A^{\pi^{\text{old}}}(s,a) - C\,\mathbf{KL}(\pi^{\text{old}} \parallel \pi_\theta)[s]\right] \to \max_\theta \tag{49}$$

The naive idea would be to straightforwardly optimize (49) as it is equivalent to solving the constraint trust-region task (48). To avoid Hessian-involved computations, one possible option is just to perform one step of first-order gradient optimization of (49). Such algorithm was empirically discovered to perform poorly as importance sampling coefficients $\frac{\pi_\theta(a|s)}{\pi^{\text{old}}(a|s)}$ tend to unbounded growth.

In PPO it is proposed to cope with this problem in a simple old-fashioned way: by clipping. Let's denote by

$$r(\theta) := \frac{\pi_\theta(a \mid s)}{\pi^{\text{old}}(a \mid s)}$$

an importance sampling weight and by

$$r^{\text{clip}}(\theta) := clip(r(\theta), 1-\epsilon, 1+\epsilon)$$

its clipped version where $\epsilon \in (0,1)$ is a hyperparameter. Then the clipped version of lower bound is:

$$\mathbb{E}_{\pi^{\text{old}}}\left[\min\left(r(\theta)A^{\pi^{\text{old}}}(s,a), r^{\text{clip}}(\theta)A^{\pi^{\text{old}}}(s,a)\right) - C\,\mathbf{KL}(\pi^{\text{old}} \parallel \pi_\theta)[s]\right] \to \max_\theta \tag{50}$$

Here the minimum operation is introduced to guarantee that the surrogate objective remains a lower bound. Thus the clipping at $1+\epsilon$ may occur only in the case if advantage is positive while clipping at $1-\epsilon$ may occur if advantage is negative. In both cases, clipping represents a penalty for importance sampling weight $r(\theta)$ being too far from 1.

The overall procedure suggested by PPO to optimize the «stabilized» version of lower bound (50) is the following. A roll-out is collected using current policy π^{old} with some parameters θ. Then the batches of typical size (as for Actor-Critic methods) are sampled from collected roll-out and several steps of SGD optimization of (50) proceed with respect to policy parameters θ. During this process the policy π^{old} is considered to be fixed and new interaction steps are not performed, while in implementations there is no need to store old weights θ_k since everything required from π^{old} is to collect transitions and remember the probabilities $\pi^{\text{old}}(a \mid s)$. The idea is that during these several steps we may use transitions from the collected roll-out several times. Similar alternative is to perform several epochs of training by passing through roll-out several times, as it is often done in deep learning.

Interesting fact discovered by the authors of PPO during ablation studies is that removing $\mathbf{KL}$-penalty term doesn't affect the overall empirical performance. That is why in many implementations PPO does not include $\mathbf{KL}$-term at all, making the final surrogate objective have a following form:

$$\mathbb{E}_{\pi^{\text{old}}} \min\left(r(\theta)A^{\pi^{\text{old}}}(s,a), r^{\text{clip}}(\theta)A^{\pi^{\text{old}}}(s,a)\right) \to \max_\theta \tag{51}$$

Note that in this form the surrogate is not generally a lower bound and «improvement guarantees» intuition is lost.

Algorithm 7: Proximal Policy Optimization (PPO)

Hyperparameters: B — batch size, R — rollout size, $\mathbf{n\_epochs}$ — number of epochs, ε — clipping parameter, V_ϕ^* — critic neural network, π_θ — actor neural network, α — critic loss scaling, SGD optimizer.

Initialize weights θ, ϕ arbitrary
On each step:

1. obtain a roll-out of size $\boldsymbol{R}$ using policy $\boldsymbol{\pi(\theta)}$, storing action probabilities as $\boldsymbol{\pi^{\text{old}}(a \mid s)}$.
2. for each transition $\boldsymbol{T}$ from the roll-out compute advantage estimation (detached from computational graph to prevent backpropagation):

$$\boldsymbol{A^{\pi}(T) = r' + \gamma V_{\phi}^{\pi}(s') - V_{\phi}^{\pi}}$$

3. perform $\mathbf{n\_epochs}$ passes through roll-out using batches of size $\boldsymbol{B}$; **for each batch:**
   - compute critic target (detached from computational graph to prevent backpropagation):

$$\boldsymbol{y(T) = r' + \gamma V_{\phi}^{\pi}(s')}$$

   - compute critic loss:

$$\mathbf{Loss} = \frac{\mathbf{1}}{\boldsymbol{B}} \sum_{\boldsymbol{T}} \left(\boldsymbol{y(T) - V_{\phi}^{\pi}} \right)^{\mathbf{2}}$$

   - compute critic gradients:

$$\boldsymbol{\nabla^{\text{critic}}} = \frac{\boldsymbol{\partial}\, \mathbf{Loss}}{\boldsymbol{\partial \phi}}$$

   - compute importance sampling weights:

$$\boldsymbol{r_{\theta}(T)} = \frac{\boldsymbol{\pi_{\theta}(a \mid s)}}{\boldsymbol{\pi^{\text{old}}(a \mid s)}}$$

   - compute clipped importance sampling weights:

$$\boldsymbol{r_{\theta}^{\text{clip}}(T) = clip(r_{\theta}(T), 1 - \epsilon, 1 + \epsilon)}$$

   - compute actor gradient:

$$\boldsymbol{\nabla^{\text{actor}}} = \frac{\mathbf{1}}{\boldsymbol{B}} \sum_{\boldsymbol{T}} \boldsymbol{\nabla_{\theta}} \, \mathbf{min} \left(\boldsymbol{r_{\theta}(T) A^{\pi}(T), r_{\theta}^{\text{clip}}(T) A^{\pi}(T)} \right)$$

   - make a step of gradient descent using $\boldsymbol{\nabla^{\text{actor}} + \alpha \nabla^{\text{critic}}}$

6. Experiments

6.1. Setup

We performed our experiments using custom implementation of discussed algorithms attempting to incorporate best features from different official and unofficial sources and unifying all algorithms in a single library interface. The full code is available at our github.

While custom implementation might not be the most efficient, it hinted us several ambiguities in algorithms which are resolved differently in different sources. We describe these nuances and the choices made for our experiments in appendix A.

For each environment we launch several algorithms to train the network with the same architecture with the only exception being the head which is specified by the algorithm (see table 1).

DQN	Linear transformation to $\lvert\mathcal{A}\rvert$ arbitrary real values
Dueling	First head: linear transformation to $\lvert\mathcal{A}\rvert$ arbitrary real values Second head: linear transformations to an arbitrary scalar Aggregated using dueling architecture formula (17)
Categorical	$\lvert\mathcal{A}\rvert$ linear transformations with softmax to $\boldsymbol{A}$ values
Dueling Categorical	First head: linear transformation to $\lvert\mathcal{A}\rvert$ arbitrary real values Second head: $\lvert\mathcal{A}\rvert$ linear transformations to $\boldsymbol{A}$ arbitrary real values Aggregated using dueling architecture formula (32)
Quantile	$\lvert\mathcal{A}\rvert$ linear transformations to $\boldsymbol{A}$ arbitrary real values
Dueling Quantile	First head: linear transformation to $\lvert\mathcal{A}\rvert$ arbitrary real values Second head: $\lvert\mathcal{A}\rvert$ linear transformations to $\boldsymbol{A}$ arbitrary real values Aggregated using dueling architecture formula (32) without softmax
Policy Gradient	Actor head: linear transformation with softmax to $\lvert\mathcal{A}\rvert$ values Critic head: linear transformation to scalar value

Table 1: Heads used for different algorithms. Here $\lvert\mathcal{A}\rvert$ is the number of actions and $\boldsymbol{A}$ is the chosen number of atoms.

For noisy networks all fully-connected layers in the feature extractor and in the head are substituted with noisy layers, doubling the number of their trained parameters. Both usage of noisy layers and the choice of the head influences the total number of parameters trained by the algorithm.

As practical tuning of hyperparameters is computationally consuming activity, we set all hyperparameters to their recommended values while trying to share the values of common hyperparameters among algorithms without affecting overall performance.

We choose to give each algorithm same amount of interaction steps to provide the fair comparison of their sample efficiency. Thus the wall-clock time, number of episodes played and the number of network parameters updates varies for different algorithms.

6.2. Cartpole

Cartpole from OpenAI Gym [2] is considered to be one of the simplest environments for deep RL algorithms testing. The state is described with 4 real numbers while action space is two-dimensional discrete.

The environment rewards agent with +1 each tick until the episode ends. Poor action choices lead to early termination. The game is considered solved if agent holds for 200 ticks, therefore 200 is maximum reward in this environment.

In our first experiment we launch algorithms for 10 000 interaction steps to train a neural network on the Cartpole environment. The network consists of two fully-connected hidden layers with 128 neurons and an algorithm-specific head. We used ReLU for activations. The results of a single launch are provided in table 2.

	Reached 200	Average reward	Average FPS
Double DQN	23.0	126.17	95.78
Dueling Double DQN	27.0	121.78	62.65
DQN	33.0	116.27	101.53
Categorical DQN	28.0	110.87	74.95
Prioritized Double DQN	37.0	110.52	85.58
Categorical Prioritized Double DQN	46.0	104.86	66.00
Quantile Prioritized Double DQN	42.0	100.76	68.62
Categorical DQN with target network	44.0	96.08	73.92
Quantile Double DQN	54.0	93.14	75.40
Quantile DQN	70.0	88.12	77.93
Categorical Double DQN	42.0	81.25	70.90
Noisy Quantile Prioritized Dueling DQN	86.0	74.13	21.41
Twin DQN	57.0	71.14	52.51
Noisy Double DQN	67.0	71.06	31.81
Noisy Prioritized Double DQN	94.0	67.34	30.72
Quantile Regression Rainbow	106.0	67.11	21.54
Rainbow	91.0	64.01	20.35
Noisy Quantile Prioritized Double DQN	127.0	63.01	28.27
Noisy Categorical Prioritized Double DQN	63.0	62.04	27.81
PPO with GAE	144.0	53.06	390.53
Noisy Prioritized Dueling Double DQN	180.0	47.52	22.56
PPO	184.0	45.19	412.88
Noisy Categorical Prioritized Dueling Double DQN	428.0	22.09	20.63
A2C	-	12.30	1048.64
A2C with GAE	-	11.50	978.00

Table 2: Results on Cartpole for different algorithms: number of episode when the highest score of 200 was reached, average reward across all played episodes and average number of frames processed in a second (FPS).

6.3. Pong

We used Atari Pong environment from OpenAI Gym [2] as our main testbed to study the behaviour of the following algorithms:

- DQN — Deep Q-learning (sec. 3.2)
- c51 — Categorical DQN (sec. 4.2)
- QR-DQN — Quantile Regression DQN (sec. 4.3)
- Rainbow (sec. 4.4)
- A2C — Advantage Actor Critic (sec. 5.3) extended with GAE (sec. 5.4)
- PPO — Proximal Policy Optimization (sec. 5.7) extended with GAE (sec. 5.4)

In Pong, each episode is split into rounds. Each round ends with player either winning or loosing. The episode ends when the player wins or looses 21 rounds. The reward is given after each round and is +1 for winning and -1 for loosing. Therefore the maximum total reward is 21 and the minimum is -21. Note that the flag $\mathbf{done}$ indicating episode ending is not provided to the agent after each round but only at the end of full game (consisting of 21-41 rounds).

The standard preprocessing for Atari games proposed in DQN [13] was applied to the environment (see table 3). Thus, state space is represented by $(\mathbf{84}, \mathbf{84})$ grayscale pixels input (1 channel with domain $[\mathbf{0}, \mathbf{255}]$). Action space is discrete with $|\mathcal{A}| = \mathbf{6}$ actions.

All algorithms were given 1 000 000 interaction steps to train the network with the same feature extractor presented on fig. 1. The number of trained parameters is presented in table 4. All used hyperparameters are listed in table 7 in appendix B.

NoopResetEnv	Do nothing first 30 frames of games to imitate the pause between game start and real player reaction.
MaxAndSkipEnv	Each interaction steps takes 4 frames of the game to allow less frequent switch of action. Max is taken over 4 passed frames to obtain an observation.
FireResetEnv	Presses «Fire» button at first frame to launch the game, otherwise screen remains frozen.
WarpFrame	Turns observation to grayscale image of size 84x84.

Table 3: Atari Pong preprocessing

Algorithm	Number of trained parameters
DQN	1 681 062
c51	1 834 962
QR-DQN	1 834 962
Rainbow	3 650 410
A2C	1 681 575
PPO	1 681 575

Table 4: Number of trained parameters in Pong experiment.

6.4. Interaction-training trade-off in value-based algorithms

There is a common belief that policy gradient algorithms are much faster in terms of computational costs while value-based algorithms are preferable when simulation is expensive because of their sample efficiency. This follows from the nature of algorithms, as the fraction «observations per network updates» is extremely different for these two families: indeed, in DQN it is often assumed to perform one network update after each new transitions, while A2C collects about 32-40 observations for only one update. That makes the number of network updates performed during 1M steps interaction process substantially different and is the main reason of policy gradients speed rate.

Also policy gradient algorithms use several threads for parallel simulations (8 in our experiments) while value-based algorithms are formally single-threaded. Yet they can also enjoy multi-threaded interaction, in the simplest form by playing 1 step in all instances of environment and then performing $\boldsymbol{L}$ steps of network optimization [8]. For consistency with single-threaded case it is reasonable to set the value of $\boldsymbol{L}$ to be equal to the number of threads to maintain the same fraction «observations per network updates».

However it has been reported that lowering value of $\boldsymbol{L}$ in two or four times can positively affect wall-clock time with some loss of sample efficiency, while raising batch size may mitigate this downgrade. The overall impact of such acceleration of value-based algorithms on performance properties is not well studied and may alter their behaviour.

In our experiments on Pong it became evident that value-based algorithms perform extensive amount of redundant network optimization steps, absorbing knowledge faster than novel information from new transitions comes in. This reasoning in particular follows from the success of PPO on Pong task which performs more than 10 times less network updates.

	Vanilla algorithm	Accelerated version
Threads	1	8
Batch size	32	128
$\boldsymbol{L}$	1	2
Interactions per update	1	4

Table 5: Setup for value-based acceleration experiment

We compared two versions of value-based algorithms: ***vanilla version***, which is single-threaded with standard batch size (32) and $\boldsymbol{L} = \mathbf{1}$ meaning that each observed transition is followed with one network optimization step, and ***accelerated version***, where 1 interaction step is performed in 8 parallel instances of environment and $\boldsymbol{L}$ is set to be 2 instead of 8 which raises the fraction «ob-

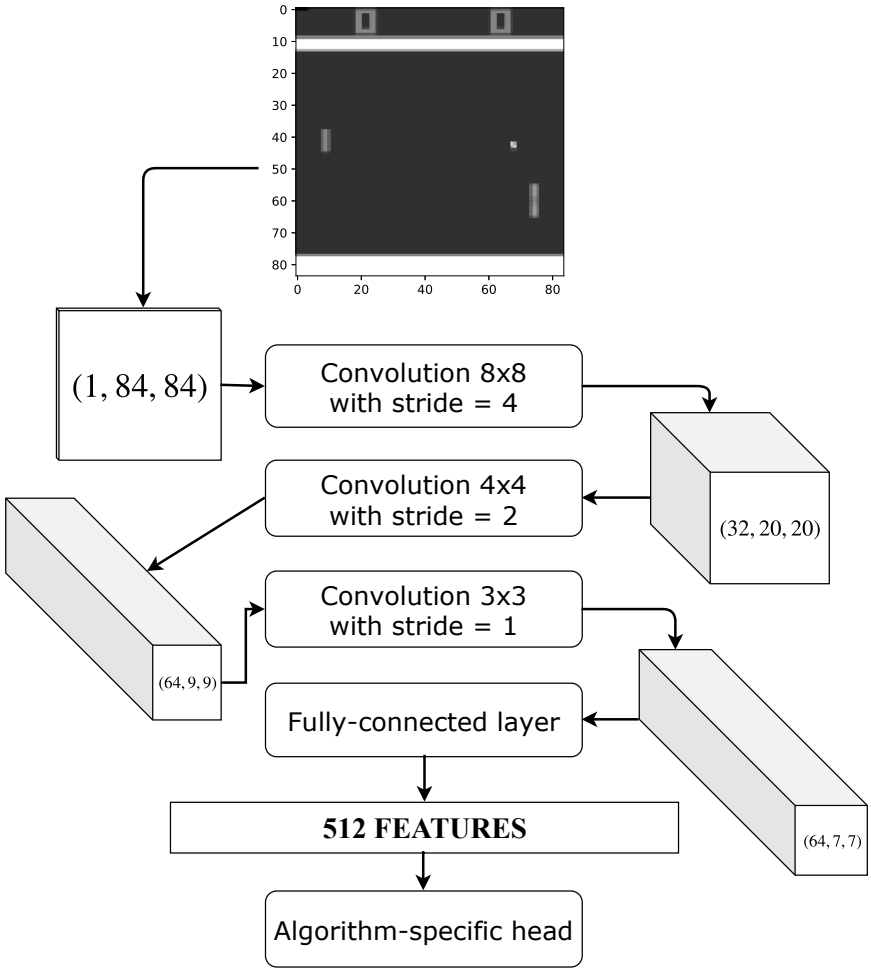

Figure 1: Network used for Atari Pong. All activation functions are ReLU. For Rainbow the fully-connected layer and all dense layers in the algorithm-specific head are substituted with noisy layers.

servations per training step» in four times. To compensate this change we raised batch size in four times.

As expected, average speed of algorithms increases in approximately 3.5 times (see table 6). We provide training curves with respect to 1M performed interaction steps on fig. 2 and with respect to wall-clock time on fig. 3. The only vanilla algorithm that achieved better final score comparing to its accelerated rival is QR-DQN, while other three algorithms demonstrated both acceleration and performance improvement. The latter is probably caused by randomness as relaunch of algorithms within the same setting and hyperparameters can be strongly influenced by random seed.

It can be assumed that fraction «observations per updates» is an important hyperparameter of value-based algorithms which can control the trade-off between wall-clock time and sample efficiency. From our results it follows that low fraction leads to excessive network updates and may slow down learning in several times. Yet this hyperparameter can barely be tuned universally for all kinds of tasks opposed to many other hyperparameters that usually have their recommended default values.

We stick further to the accelerated version and use its results in final comparisons.

6.5. Results

We compare the results of launch of six algorithms on Pong from two perspectives: sample efficiency (fig. 4) and wall-clock time (fig. 5). We do not compare final performance of these algorithms as all six algorithms are capable to reach near-maximum final score on Pong given more iterations, while results after 1M iterations on a single launch significantly depend on chance.

All algorithms start with a warm-up session during which they try to explore the environment and

Algorithm	Interactions per update		Average transitions per second	
	vanilla	accelerated	vanilla	accelerated
DQN	1	4	55.74	168.43
c51	1	4	44.08	148.76
QR-DQN	1	4	47.46	155.97
Rainbow	1	4	19.30	70.22
A2C	40		656.25	
PPO	10.33		327.13	

Table 6: Computational efficiency of vanilla and accelerated versions.

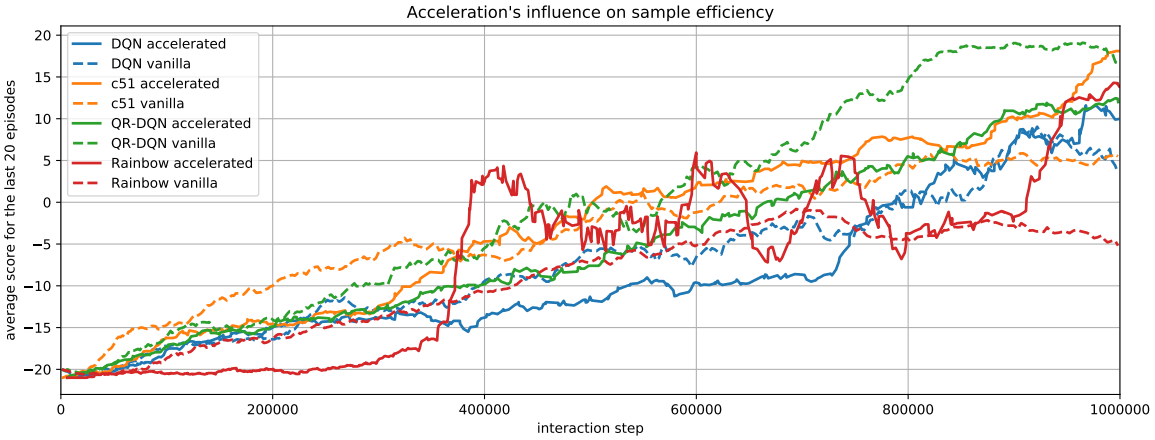

Figure 2: Training curves of vanilla and accelerated version of value-based algorithms on 1M steps of Pong. Although accelerated versions perform network updates four times less frequent, the performance degradation is not observed.

learn first dependencies how the result of random behaviour can be surpassed. Epsilon-greedy with tuned parameters provides sufficient amount of exploration for DQN, c51 and QR-DQN whithout slowing down further learning while hyperparameter-free noisy networks are the main reason why Rainbow has substantially longer warm-up.

Policy gradient algorithms incorporate exploration strategy in stochasticity of learned policy but underutilization of observed samples leads to almost 1M-frames warm-up for A2C. It can be observed that PPO successfully mitigates this problem by reusing samples thrice. Nevertheless, both PPO and A2C solve Pong relatively quickly after the warm-up stage is over.

Value-based algorithm proved to be more computationally costly. QR-DQN and categorical DQN introduce more complicated loss computation, yet their slowdown compared to standard DQN is moderate. On the contrary, Rainbow is substantially slower mainly because of noise generation involvement. Furthermore, combination of noisy networks and prioritized replay results in even less stable training process.

We provide loss curves for all six algorithms and statistics for noise magnitude and prioritized replay for Rainbow in appendix C; some additional visualizations of trained algorithms playing episodes of Pong are presented in appendix D.

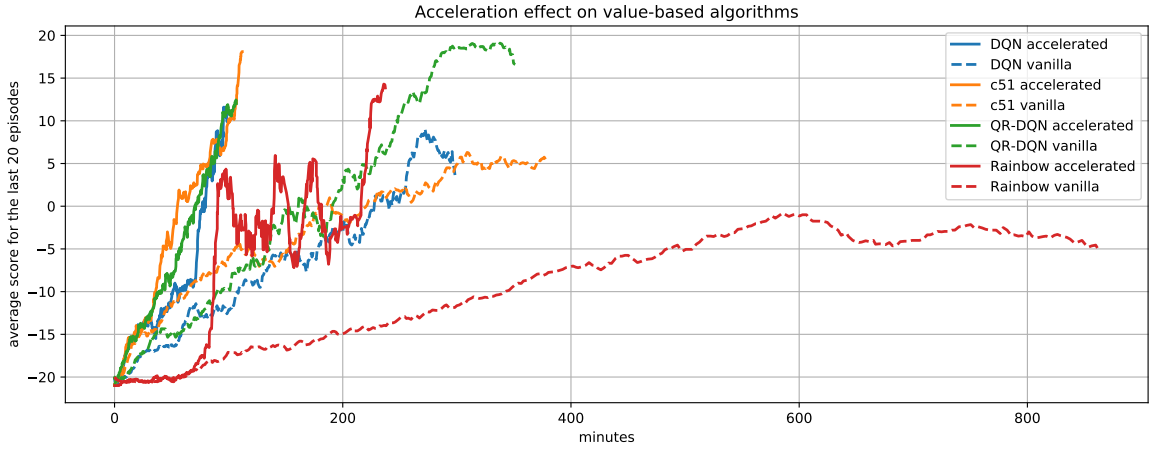

Figure 3: Training curves of vanilla and accelerated version of value-based algorithms on 1M steps of Pong from wall-clock time.

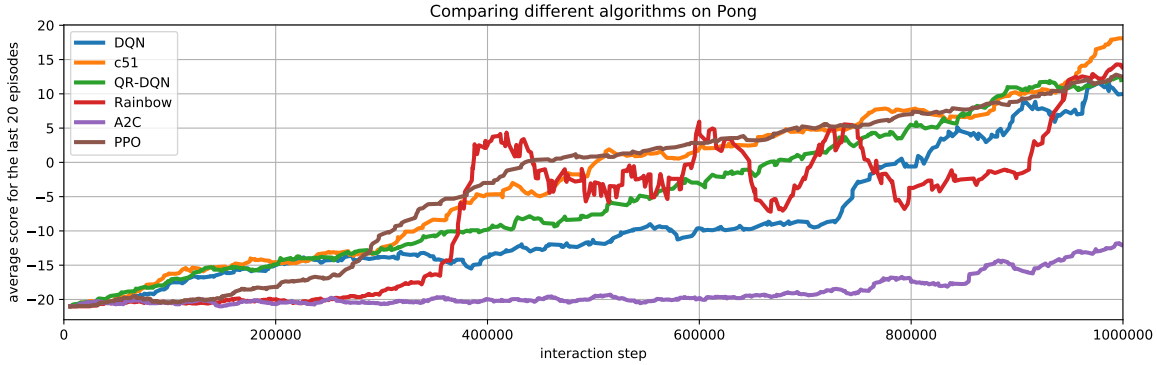

Figure 4: Training curves of all algorithms on 1M steps of Pong.

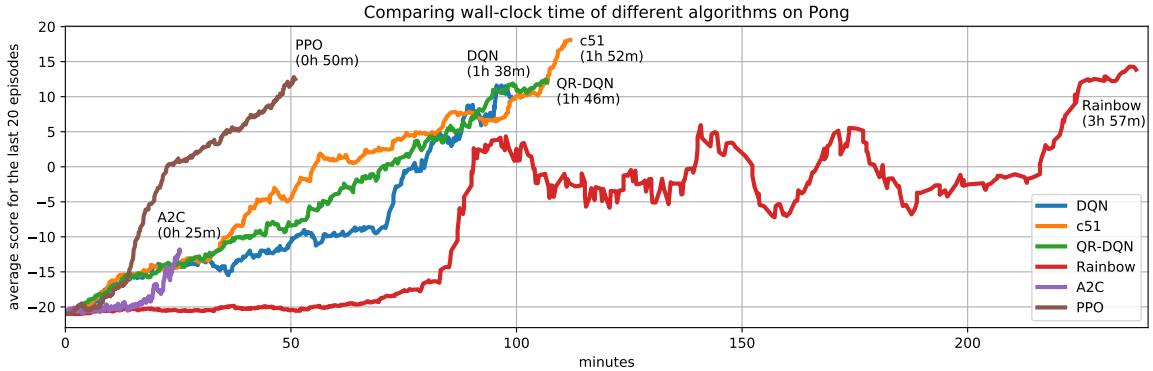

Figure 5: Training curves of all algorithms on 1M steps of Pong from wall-clock time.

7. Discussion

We have concerned two main directions of universal model-free RL algorithm design and attempted to recreate several state-of-art pipelines.

While the extensions of DQN are reasonable solutions of evident DQN problems, their effect is not clearly seen on simple tasks like Pong[33]. Current state-of-art in single-threaded value-based approach, Rainbow DQN, is full of «glue and tape» decisions that might be not the most effective way of training process stabilization.

Policy gradient algorithms are aimed at direct optimization of objective and currently beat value-based approach in terms of computational costs. They tend to have less hyperparameters but are extremely sensitive to the choice of optimizer parameters and especially learning rate. We have affirmed the effectiveness of state-of-art algorithm PPO, which succeeded to solve Pong within an hour without hyperparameter tuning. Though on the one hand this algorithm was derived from TRPO theory, it essentially deviates from it and substitutes trust region updates with heuristic clipping.

It can be observed in our results that PPO provides better gradients to the same network than DQN-based algorithms despite the absence of experience replay. While it is fair to assume that forgetting experienced transitions leads to information loss, it is also true that most observations stored in replay memory are already learned or contain no useful information. The latter makes most transitions in the sampled mini-batches insignificant, and, while prioritized replay attacks this issue, it might still be the case that current experience replay management techniques are imperfect.

There are still a lot of deviations of empirical results from theoretical perspectives. It is yet unclear which techniques are of the highest potential and what explanation lies behind many heuristic elements composing current state-of-art results. Possibly essential elements of modeling human-like reinforcement learning are yet to be unraveled as active research in this area promises substantial acceleration, generalization and stabilization of DRL algorithms.

[33] although it takes several hours to train, Pong is considered to be the easiest of 57 Atari games and one of the most basic testbeds for RL algorithms.

References

[1] M. G. Bellemare, W. Dabney, and R. Munos. A distributional perspective on reinforcement learning. In ***Proceedings of the 34th International Conference on Machine Learning-Volume 70***, pages 449–458. JMLR. org, 2017.

[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. ***arXiv preprint arXiv:1606.01540***, 2016.

[3] W. Dabney, M. Rowland, M. G. Bellemare, and R. Munos. Distributional reinforcement learning with quantile regression. In ***Thirty-Second AAAI Conference on Artificial Intelligence***, 2018.

[4] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, et al. Noisy networks for exploration. ***arXiv preprint arXiv:1706.10295***, 2017.

[5] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. ***Deep learning***, volume 1. MIT press Cambridge, 2016.

[6] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. Deep reinforcement learning that matters. In ***Thirty-Second AAAI Conference on Artificial Intelligence***, 2018.

[7] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. In ***Thirty-Second AAAI Conference on Artificial Intelligence***, 2018.

[8] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. Van Hasselt, and D. Silver. Distributed prioritized experience replay. ***arXiv preprint arXiv:1803.00933***, 2018.

[9] A. Irpan. Deep reinforcement learning doesnвЃЄt work yet. ***Online (Feb. 14): https://www. alexirpan. com/2018/02/14/rl-hard. html***, 2018.

[10] L. Kaiser, M. Babaeizadeh, P. Milos, B. Osinski, R. H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine, et al. Model-based reinforcement learning for atari. ***arXiv preprint arXiv:1903.00374***, 2019.

[11] R. Koenker and G. Bassett Jr. Regression quantiles. ***Econometrica: journal of the Econometric Society***, pages 33–50, 1978.

[12] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. ***arXiv preprint arXiv:1509.02971***, 2015.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. ***arXiv preprint arXiv:1312.5602***, 2013.

[14] OpenAI. Openai five. https://blog.openai.com/openai-five/, 2018.

[15] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. ***arXiv preprint arXiv:1703.03864***, 2017.

[16] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. ***arXiv preprint arXiv:1511.05952***, 2015.

[17] J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz. Trust region policy optimization. In ***Icml***, volume 37, pages 1889–1897, 2015.

[18] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. ***arXiv preprint arXiv:1506.02438***, 2015.

[19] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. ***arXiv preprint arXiv:1707.06347***, 2017.

[20] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. ***arXiv preprint arXiv:1712.01815***, 2017.

[21] R. S. Sutton and A. G. Barto. ***Reinforcement learning: An introduction***. MIT press, 2018.

[22] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In ***Advances in neural information processing systems***, pages 1057–1063, 2000.

[23] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In ***Thirtieth AAAI Conference on Artificial Intelligence***, 2016.

[24] O. Vinyals, I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. M. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell, T. Ewalds, D. Horgan, M. Kroiss, I. Danihelka, J. Agapiou, J. Oh, V. Dalibard, D. Choi, L. Sifre, Y. Sulsky, S. Vezhnevets, J. Molloy, T. Cai, D. Budden, T. Paine, C. Gulcehre, Z. Wang, T. Pfaff, T. Pohlen, Y. Wu, D. Yogatama, J. Cohen, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, C. Apps, K. Kavukcuoglu, D. Hassabis, and D. Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/, 2019.

[25] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas. Dueling network architectures for deep reinforcement learning. ***arXiv preprint arXiv:1511.06581***, 2015.

[26] C. J. Watkins and P. Dayan. Q-learning. ***Machine learning***, 8(3-4):279–292, 1992.

[27] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. ***Machine learning***, 8(3-4):229–256, 1992.

Appendix A. Implementation details

Here we describe several technical details of our implementation which may potentially influence the obtained results.

In most papers on value-based algorithms hyperparameters recommended for Atari games assume raw input in the range $[\mathbf{0}, \mathbf{255}]$, while in various implementations of policy gradient algorithms normalized input in the range $[\mathbf{0}, \mathbf{1}]$ is considered. Stepping aside from these agreements may damage the convergence speed both for value-based and policy gradient algorithms as the change of input domain requires hyperparameters retuning.

We use MSE loss emerged in theoretical intuition for DQN while in many sources it is recommended to use Huber loss[34] instead to stabilize learning.

In all value-based algorithms except c51 we update target network each $\boldsymbol{K}$-th frame instead of exponential smoothing of its parameters as it is computationally cheaper. For c51 we remove target network heuristic as apriori limited domain prevents unbounded growth of predictions.

We do not architecturally force quantiles outputted by the network in Quantile Regression DQN to satisfy $\zeta_0 \leq \zeta_1 \leq \cdots \leq \zeta_{A-1}$. As in the original paper, we assume that all $\boldsymbol{A}$ outputs of network are arbitrary real values and use a standard linear transformation as our last layer.

In dueling architectures we subtract mean of $\boldsymbol{A}(s, a)$ across actions instead of theoretically assumed maximum as proposed by original paper authors.

We implement sampling from prioritized replay using SumTree data structure and in informal experiments affirmed the acceleration it provides. The importance sampling weight annealing $\beta(t)$ is represented by initial value $\beta(0) = \beta$ which is then linearly annealed to 1 during first T_β frames; both β and T_β are hyperparameters.

We do not allow priorities $\mathcal{P}(T)$ to be greater than 1 by clipping as suggested in the original paper. This may mitigate the effect of prioritization replay but stabilizes the process.

As importance sampling weights $w(T) = \frac{1}{B\mathcal{P}(T)}$ are potentially very close to zero, in original article it was proposed to normalize them on $\max w(T)$. In some implementations the maximum is taken over the whole experience replay while in others maximum is taken over current batch, which is not theoretically justified but computationally much faster. We stick to the latter option.

For noisy layers we use factorized noise sampling: for layer with m inputs and n outputs we sample $\varepsilon_1 \in \mathbb{R}^n, \varepsilon_2 \in \mathbb{R}^m$ from standard normal distributions and scale both using $f(\varepsilon) = \mathbf{sign}(\varepsilon)\sqrt{\varepsilon}$. Thus we use $f(\varepsilon_1)f(\varepsilon_2)^T$ as our noise sample for weights matrix and $f(\varepsilon_2)$ as noise sample for bias. All noise is shared across mini-batch. Noise is resampled on each forward pass through the network and thus is independent between evaluation, selection and interaction. Despite all these simplifications, we found noisy layers to be the most computationally expensive modification of DQN leading to substantial degradation of wall-clock time.

For policy gradient algorithms we add additional policy entropy term to the loss to force exploration. We also define actor loss as a scalar function that yields the same gradients as in the corresponding gradient estimation (40) for A2C to compute it using PyTorch mechanics. For PPO objective (51) provides analogous «actor loss»; thus, in both policy gradient algorithms the full loss is defined as summation of actor, critic and entropy losses, with the two latter being scaled using scalar hyperparameters.

We use shared network architecture for policy gradient algorithms with one feature extractor and two heads, one for policy and one for critic.

$\mathbf{KL}$-penalty is not used in our PPO implementation. Also we do not normalize advantage estimations across the roll-out to zero mean and unit standard deviation as additionally done in some implementations.

We use PyTorch default initialization for linear and convolutional layers although orthogonal initialization of all layers is reported to be beneficial for policy gradient algorithms. Initial values of sigmas for noisy layers is set to be constant and equal to $\frac{\sigma_{\text{init}}}{m}$ where σ_{init} is a hyperparameter and m is the number of inputs in accordance with original paper.

We use Adam as our optimizer with default $\beta_1 = \mathbf{0.9}, \beta_2 = \mathbf{0.999}, \varepsilon = \mathbf{1e{-}8}$. No gradient clipping is performed.

[34] Huber loss is defined as

$$\mathbf{Loss}(y, \hat{y}) = \begin{cases} (y - \hat{y})^2 & \text{if } |y - \hat{y}| < 1 \\ |y - \hat{y}| & \text{else} \end{cases}$$

Appendix B. Hyperparameters

	DQN	QR-DQN	c51	Rainbow	A2C	PPO
Reward discount factor γ	0.99					
$\varepsilon(t)$-greedy strategy	$0.01 + 0.99e^{-\frac{t}{30\,000}}$			-	-	
Interactions per training step	4				-	
Batch size B	128				-	32
Rollout capacity	-				40	1024
PPO number of epochs	-					3
Replay buffer initialization size[35]	10 000 transitions				-	
Replay buffer capacity M	1 000 000 transitions				-	
Target network updates K	each 1000-th step				-	
Number of atoms A	-	51			-	
$V_{\mathrm{min}}, V_{\mathrm{max}}$	-	-	$[-10, 10]$		-	
Noisy layers std initialization	-	-	-	0.5	-	
Multistep N	-	-	-	3	-	
Prioritization degree α	-	-	-	0.5	-	
Prioritization bias correction β	-	-	-	0.4	-	
Unbiased prioritization after	-	-	-	100 000 steps	-	
GAE coeff. λ	-				0.95	
Critic loss weight	-				0.5	
Entropy loss weight	-				0.01	
PPO clip ϵ	-					0.1
Optimizer	Adam					
Learning rate	0.0001					

Table 7: Selected hyperparameters for Atari Pong

[35] number of transitions to collect in replay memory before starting network optimization using mini-batch sampling.

Appendix C. Training statistics on Pong

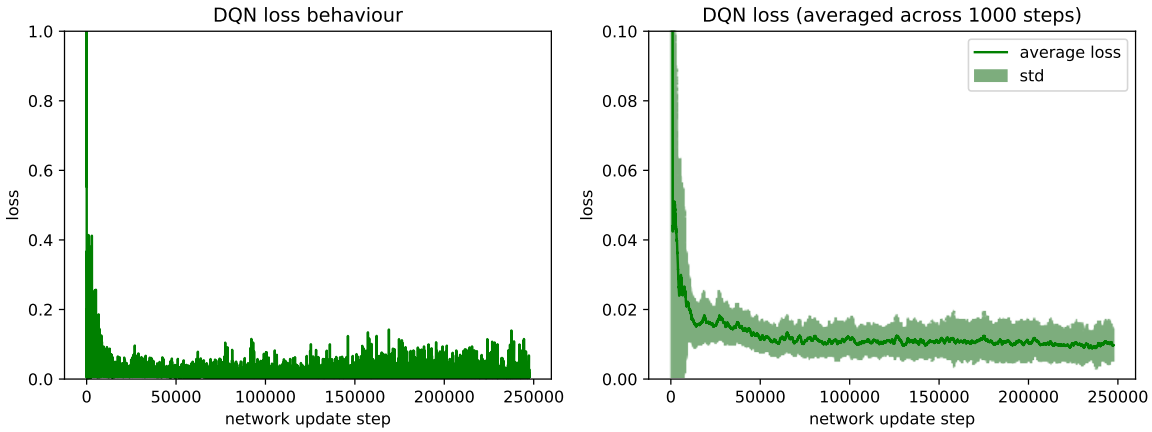

Figure 6: DQN loss behaviour during training on Pong.

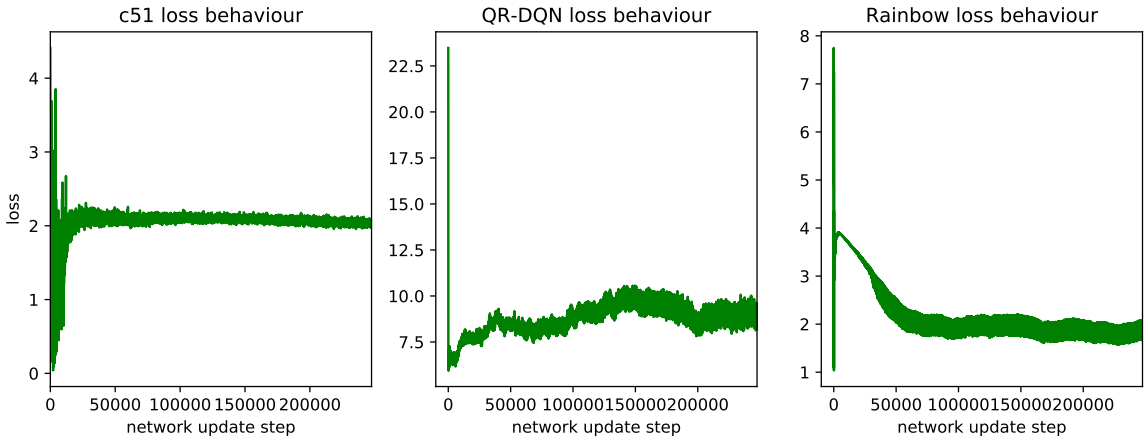

Figure 7: Loss behaviours of c51, QR-DQN and Rainbow during training on Pong.

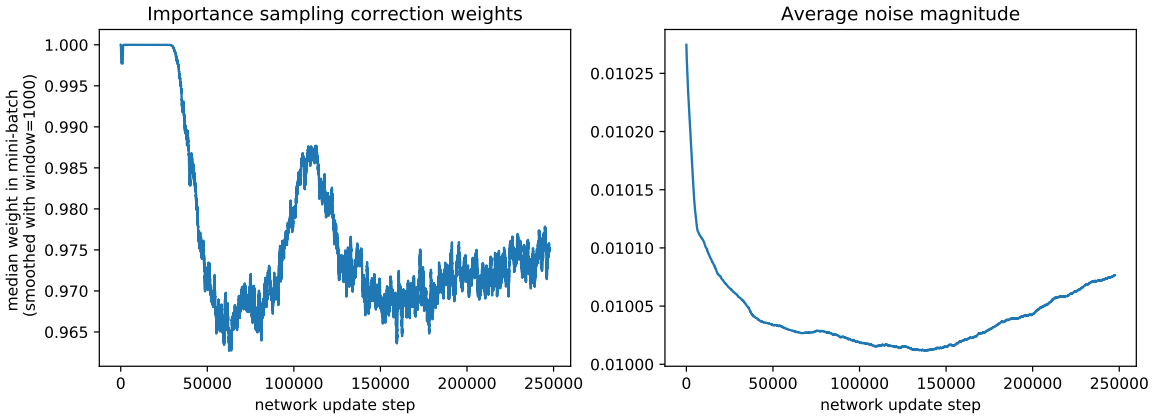

Figure 8: Rainbow statistics during training. Left: smoothed with window 1000 median of importance sampling weights from sampled mini-batches. Right: average noise magnitude logged at each 20-th step of training.

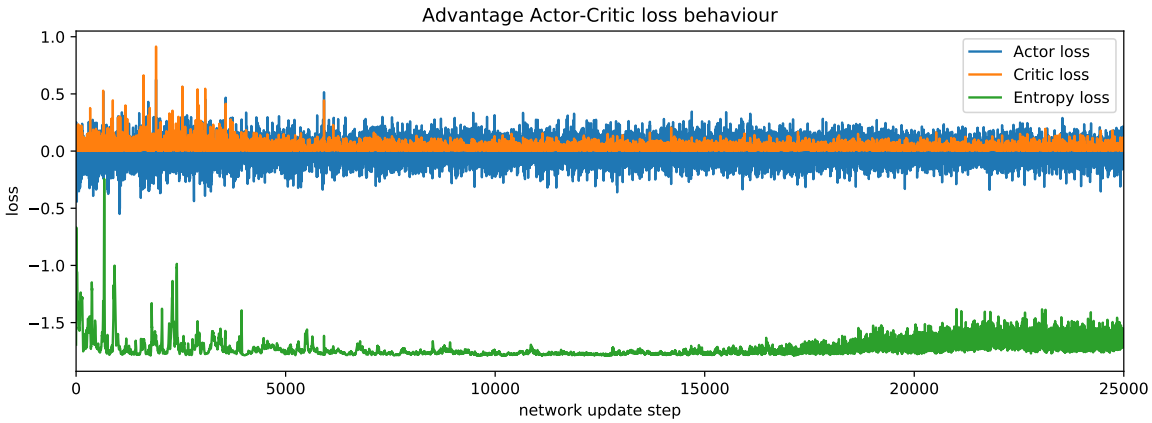

Figure 9: A2C loss behaviour during training.

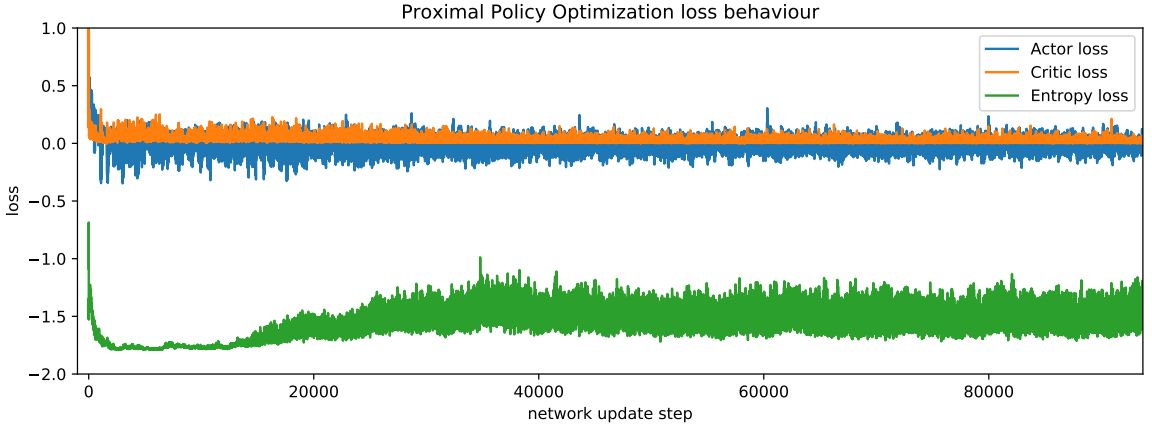

Figure 10: PPO loss behaviour during training.

Appendix D. Playing Pong behaviour

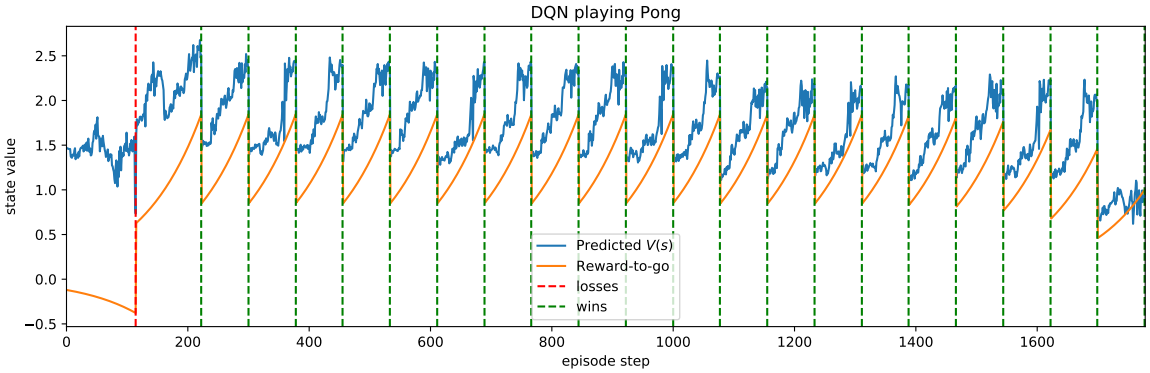

Figure 11: DQN playing one episode of Pong.

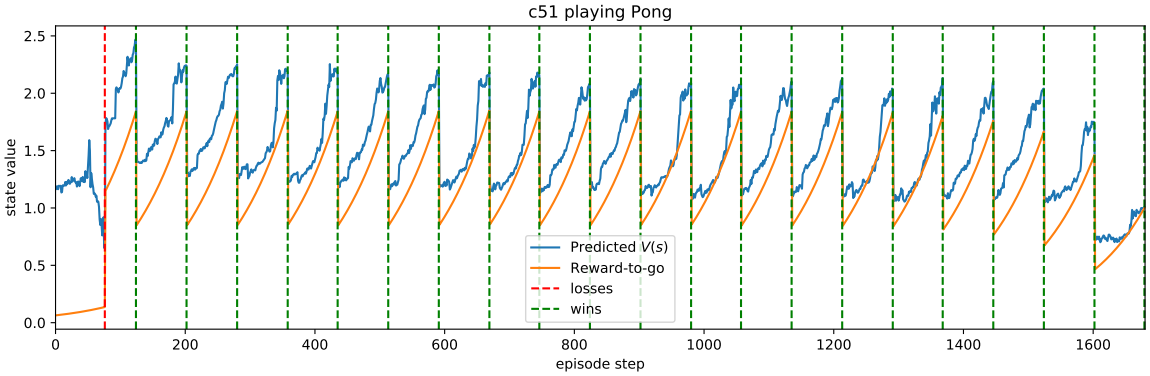

Figure 12: c51 playing one episode of Pong.

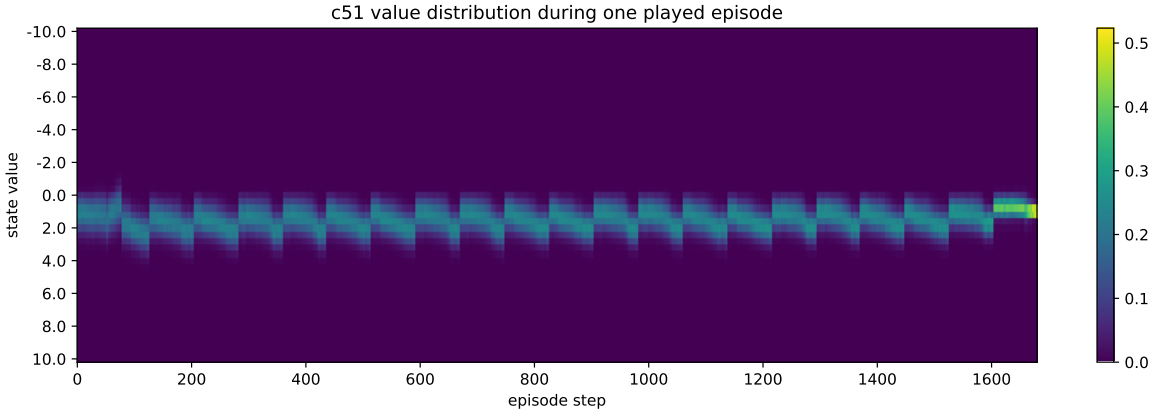

Figure 13: c51 value distribution prediction during one episode of Pong.

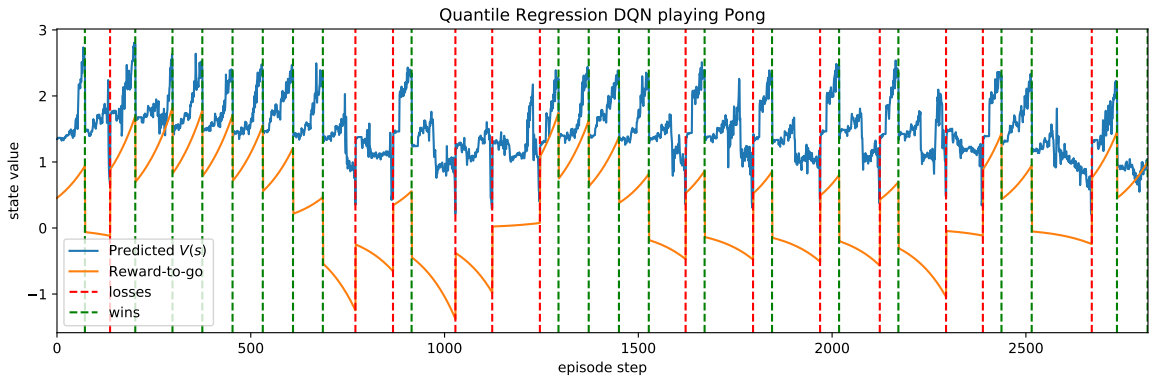

Figure 14: Quantile Regression DQN playing one episode of Pong.

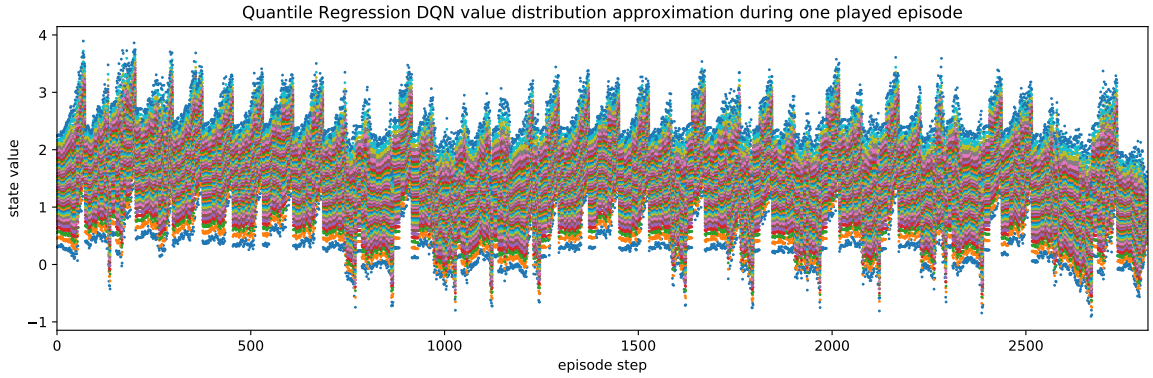

Figure 15: Quantile Regression DQN value distribution prediction during one episode of Pong.

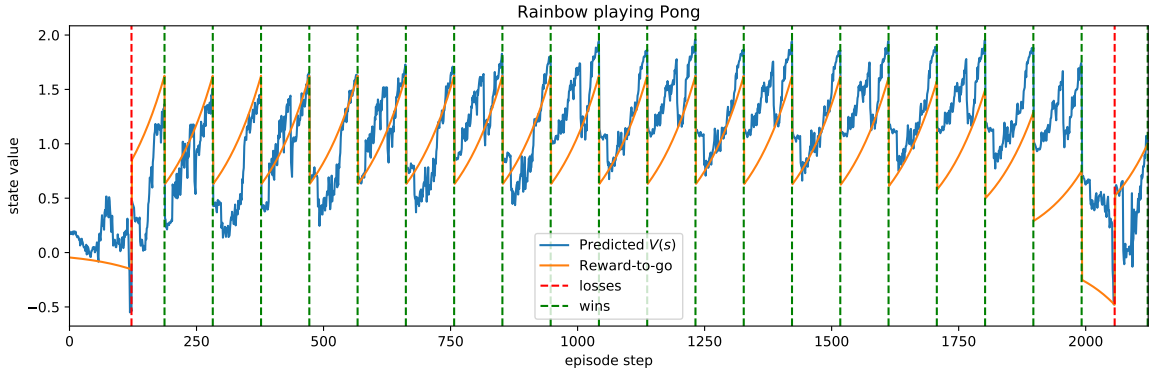

Figure 16: Rainbow playing one episode of Pong (exploration turned off, i.e. all noise samples are zero).

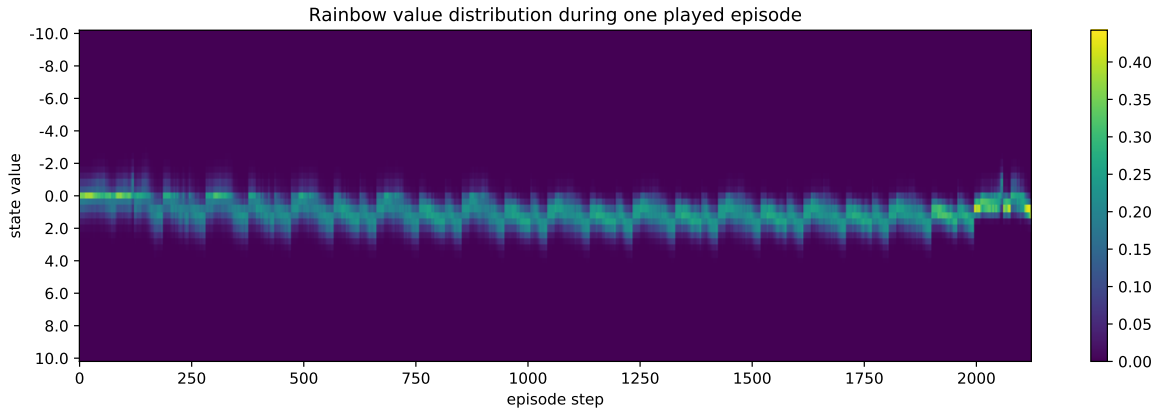

Figure 17: Rainbow value distribution prediction during one episode of Pong (exploration turned off, i.e. all noise samples are zero).

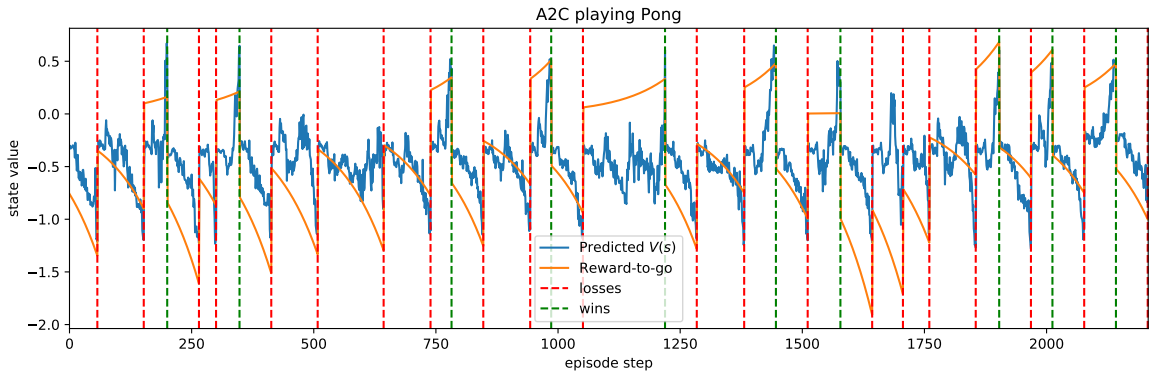

Figure 18: A2C playing one episode of Pong.

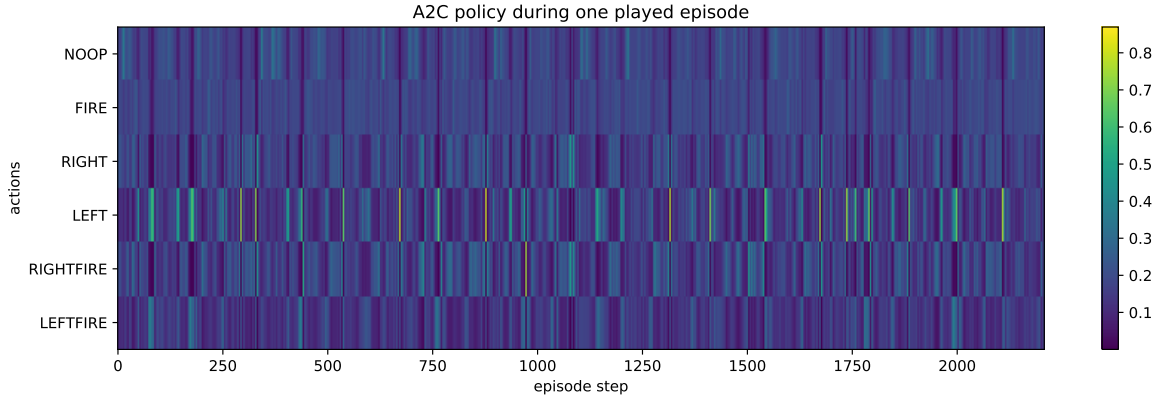

Figure 19: A2C policy distribution during one episode of Pong.

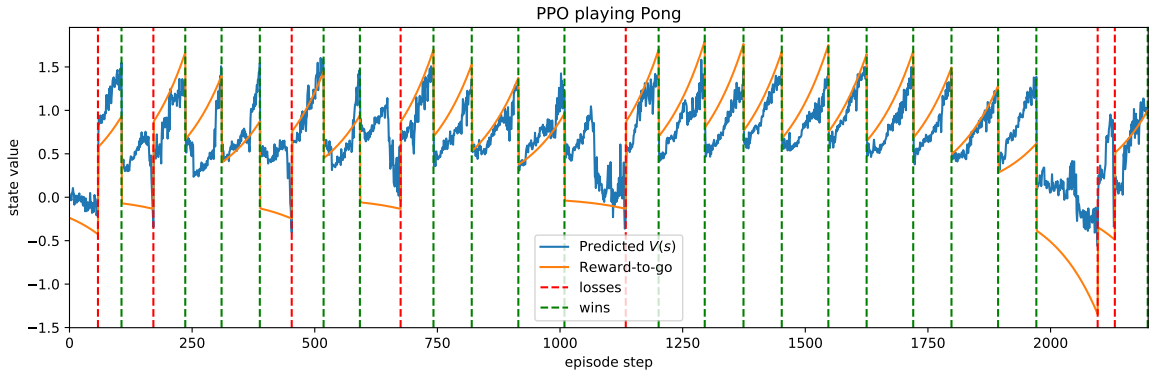

Figure 20: PPO playing one episode of Pong.

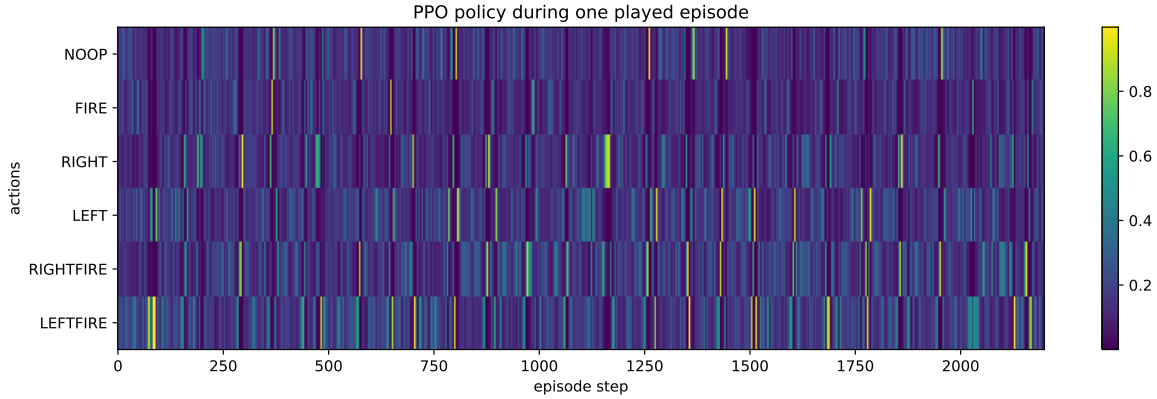

Figure 21: PPO policy distribution during one episode of Pong.