Одеський національний університет імені І. І. Мечникова
Факультет математики, фізики та інформаційних технологій
Кафедра оптимального керування та економічної кібернетики

Дипломна робота
бакалавра

на тему: **«Алгоритми керування агентами у стохастичному оточенні»**

«Algorithms of controlling agents in stochastic environments»

Виконав: студент денної форми навчання
спеціальності 113 Прикладна математика
Жуков Павло Петрович

Керівник: канд. фіз.-мат. наук, проф. Мороз В. В.
Рецензент: канд. фіз.-мат. наук, доц. Скрипник Н. В.

Рекомендовано до захисту:
Протокол засідання кафедри
№ ___ від «_____» _____ р.
Завідувач кафедри
_____ _____

Захищено на засіданні ЕК № _____
Протокол № _____ від «_____» _____ р.
Оцінка _____ / _____ / _____
Голова ЕК
_____ _____

Одеса — 2021 р.

Odesa I. I. Mechnikov National University
Faculty of Mathematics, Physics and Information Technology
Department of optimal control and economical cybernetics

Diploma thesis
bachelor

«Algorithms of controlling agents in stochastic environments»

Fulfilled by: full-time student
specialty 113 Applied Mathematics
Zhukov Pavlo

Supervisor: prof. Moroz Volodymyr
Reviewer: docent Skripnik Natalia

Odesa — 2021

Contents

Вступ

Машинне навчання з підкріпленням це облать машинного навчання, яке спеціалізується на навчанні агентів приймати рішення у спецефічних оточеннях. Це може бути щось таке, як класична відеоігра, керування автомобілем, керування роботизованною рукою тощо.

Найцікавіше використання машинного навчання з підкріпленням - автопілот для автомобілей. Вперше вони з'явилися у компанії Tesla, після чого призвели до великої кількости уваги. Насправді, там використовується багато різних технологій та алгоритмів, але без машинного навчання з підкріпленням це було б неможливо. Дуже цікаво, що ця технологія вже врятувала багато людей, втручаючись у важливу хвилину, виконуючи дії замість водія.

Популярність цієї області почалась з презентації DeepMind [6], де вони презентували алгоритм, який зміг грати у Space Invaders набагато краще, ніж це може звичайна людина. Не зважаючи на те, що ця гра досить проста, це був фурор, адже алгоритм грав використовуючи пікселі з екрану, як справжня людина. Сучасні алгоритми вже вміють грати в набагато складніші ігри, такі як шахи, го та СтарКрафт.

У машинному навчанні з підкріпленням є два підходи: Value та Policy. Перший намагається розрахувати оцінку для теперішнього стану агента та обрати таку дію, яка надасть найкращого результату. Другий намагається напряму вирішити, яку дію треба обрати. Ще існує третій тип: Actor-Critic. Він намагається об'єднати обидва алгоритму для того, щоб знайти золоту середину. Ми разглянемо декілька найкращих алгоритмів з кожного типу

Об'єктом дослідження цієї роботи є метод пошуку найкращих моделей машинного навчання з підкріпленням.

Предметом дослідження є найкращі сучасні моделі машинного научная з підкріпленям.

Метою цієї роботи є вивчення ідеї цих алгоритмів і порівняння їх результатів на кількох популярних середовищах.

Завдання цієї роботи:

- Розглянути основні алгоритми машинного навчання і знайти їх недоліки.
- Розглянути різні типи підходів.
- Натренувати деякі моделі Atari games.
- Проаналізувати наскільки розумними є їх дії.
- Порівняти результати різних алгоритмів.

.

Introduction

Reinforcement learning is the special area of machine learning, specialised on learning agents to take actions on specific environments. It could be something like playing a classic video game, driving a car, controlling robotic arms or even playing hide and seek!

One of the most interesting applications of Reinforcement Learning is the autopilot for cars. First introduced in Tesla car, it was a huge success and the technology is still getting better. In fact, it uses lot's of different technology, but RL is the key. And it already saved some lives and some cars for getting crashed on road: nowadays the car can make a fast decision in a dangerous situation, taking control from a human driver without asking.

Popularity of reinforcement learning started with a DeepMind's [6] presentation, where they were able to learn an agent to play Atari [3] Space Invaders game even better, then an average human player could. Despite it being a classic game, it was shocking then. And now, those algorithms are able to play better then even the best static computer programs in Chess and Go. And the latest ones are even able to play much more complex games like StarCraft [13] and show some amazing results there.

In solving a reinforcement learning problem there are two popular ways of approaching the problem, policy methods and value methods. To oversimplify, in policy methods you try to directly optimize the policy, in value methods you try to evaluate the expected future return (learn a value function), and deduce the policy from there. Now a third group of methods are Actor-Critic methods, which, try to combine the previous two in a meaningful way. Actor is the policy which is being optimized, Critic is the value function which is being learned. So A3C means that there are multiple actors performing in parallel (Asynchronous), and use the Advantage function (state*action value - state value) for the optimization of the policy (Actor) which is calculated from the learned value function (Critic). Now, the Actor part can be implemented with many different policy optimization methods (Vanilla PG, TRpO, PPO). Proximal Policy Optimization (PPO) is one such method.

Research object is the method of finding the best reinforcement learning models.

Research subject is the best modern methods of reinforcement learning.

The purpose of the work is studying the ideas of those algorithms and comparing their performance on some popular environments.

Tasks of this work

- Look at some basic RL algorithms and find what problems do they have.
- Look at three different types of algorithm approaches.
- Train some models on different Atari games.
- Analyse how intelligent are their actions.
- Compare results of different algorithms.

1. Overview of the problem area

1.1. What is reinforcement learning?

Reinforcement learning is the training of machine learning models to make a sequence of decisions. The agent learns to achieve a goal in an uncertain, potentially complex environment. In reinforcement learning, an artificial intelligence faces a game-like situation. The computer employs trial and error to come up with a solution to the problem. To get the machine to do what the programmer wants, the artificial intelligence gets either rewards or penalties for the actions it performs. Its goal is to maximize the total reward.

Although the designer sets the reward policy–that is, the rules of the game–he gives the model no hints or suggestions for how to solve the game. It's up to the model to figure out how to perform the task to maximize the reward, starting from totally random trials and finishing with sophisticated tactics and superhuman skills. By leveraging the power of search and many trials, reinforcement learning is currently the most effective way to hint machine's creativity. In contrast to human beings, artificial intelligence can gather experience from thousands of parallel gameplays if a reinforcement learning algorithm is run on a sufficiently powerful computer infrastructure.

1.2. Examples of reinforcement learning

Applications of reinforcement learning were in the past limited by weak computer infrastructure. However, as Gerard Tesauro's backgamon AI superplayer developed in 1990's shows, progress did happen. That early progress is now rapidly changing with powerful new computational technologies opening the way to completely new inspiring applications.

Training the models that control autonomous cars is an excellent example of a potential application of reinforcement learning. In an ideal situation, the computer should get no instructions on driving the car. The programmer would avoid hard-wiring anything connected with the task and allow the machine to learn from its own errors. In a perfect situation, the only hard-wired element would be the reward function.

For example, in usual circumstances we would require an autonomous vehicle to put safety first, minimize ride time, reduce pollution, offer passengers comfort and obey the rules of law. With an autonomous race car, on the other hand, we would emphasize speed much more than the driver's comfort. The programmer cannot predict everything that could happen on the road. Instead of building lengthy "if-then" instructions, the programmer prepares the reinforcement learning agent to be capable of learning from the system of rewards and penalties. The agent (another name for reinforcement learning algorithms performing the task) gets rewards for reaching specific goals.

1.3. Challenges with reinforcement learning

The main challenge in reinforcement learning lays in preparing the simulation environment, which is highly dependant on the task to be performed. When the model has to go superhuman in Chess, Go or Atari games, preparing the simulation environment is relatively simple. When it comes to building a model capable of driving an autonomous car, building a realistic simulator is crucial before letting the car ride on the street. The model has to figure out how to brake or avoid a collision in a safe environment, where sacrificing even a thousand cars comes at a minimal cost. Transferring the model out of the training environment and into to the real world is where things get tricky.

Scaling and tweaking the neural network controlling the agent is another challenge. There is no way to communicate with the network other than through the system of rewards and penalties.This in particular may lead to catastrophic forgetting, where acquiring new knowledge causes some of the old to be erased from the network (to read up on this issue, see this paper, published during the International Conference on Machine Learning). Yet another challenge is reaching a local optimum – that is the agent performs the task as it is, but not in the optimal or required way. A "jumper" jumping like a kangaroo instead of doing the thing that was expected of it-walking-is a great example, and is also one that can be found in our recent blog post.

Finally, there are agents that will optimize the prize without performing the task it was designed for. An interesting example can be found in the OpenAI video below, where the agent learned to gain rewards, but not to complete the race.

1.4. What distinguishes reinforcement learning from deep learning and machine learning?

In fact, there should be no clear divide between machine learning, deep learning and reinforcement learning. It is like a parallelogram – rectangle – square relation, where machine learning is the broadest category and the deep reinforcement learning the most narrow one. In the same way, reinforcement learning is a specialized application of machine and deep learning techniques, designed to solve problems in a particular way.

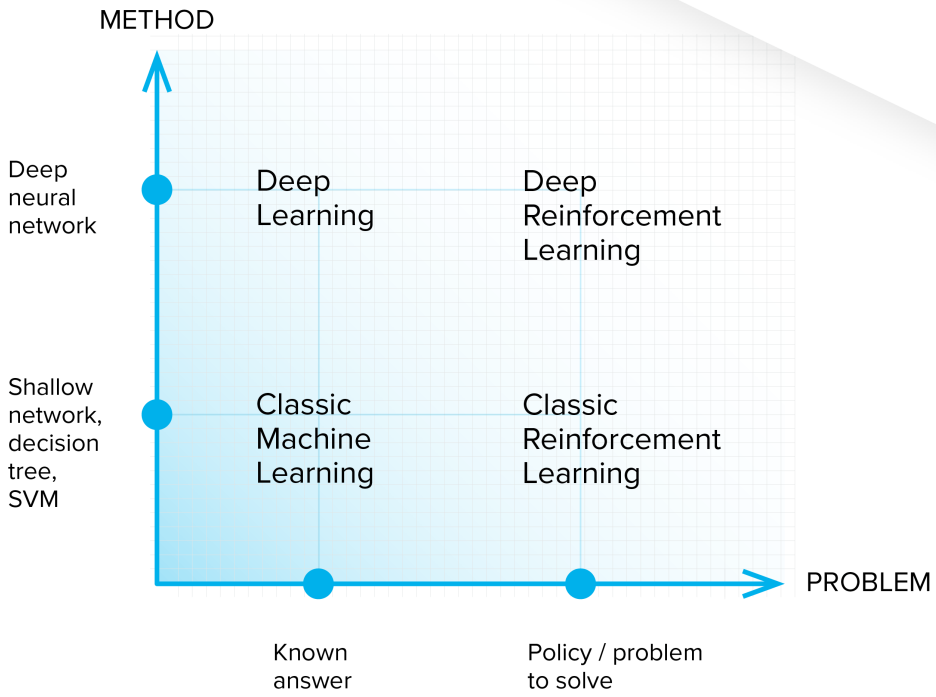


Figure 1.1. ML types

Although the ideas seem to differ, there is no sharp divide between these subtypes. Moreover, they merge within projects, as the models are designed not to stick to a "pure type"

but to perform the task in the most effective way possible. So "what precisely distinguishes machine learning, deep learning and reinforcement learning" is actually a tricky question to answer.

Machine learning – is a form of AI in which computers are given the ability to progressively improve the performance of a specific task with data, without being directly programmed (this is Arthur Lee Samuel's definition. He coined the term "machine learning", of which there are two types, supervised and unsupervised machine learning.

Supervised machine learning happens when a programmer can provide a label for every training input into the machine learning system.

Deep learning consists of several layers of neural networks, designed to perform more sophisticated tasks. The construction of deep learning models was inspired by the design of the human brain, but simplified. Deep learning models consist of a few neural network layers which are in principle responsible for gradually learning more abstract features about particular data.

Although deep learning solutions are able to provide marvelous results, in terms of scale they are no match for the human brain. Each layer uses the outcome of a previous one as an input and the whole network is trained as a single whole. The core concept of creating an artificial neural network is not new, but only recently has modern hardware provided enough computational power to effectively train such networks by exposing a sufficient number of examples. Extended adoption has brought about frameworks like TensorFlow, Keras and PyTorch, all of which have made building machine learning models much more convenient.

Reinforcement learning, as stated above employs a system of rewards and penalties to compel the computer to solve a problem by itself. Human involvement is limited to changing the environment and tweaking the system of rewards and penalties. As the computer maximizes the reward, it is prone to seeking unexpected ways of doing it. Human involvement is focused on preventing it from exploiting the system and motivating the machine to perform the task in the way expected. Reinforcement learning is useful when there is no "proper way" to perform a task, yet there are rules the model has to follow to perform its duties correctly. Take the road code, for example.

In particular, if artificial intelligence is going to drive a car, learning to play some Atari classics can be considered a meaningful intermediate milestone. A potential application of reinforcement learning in autonomous vehicles is the following interesting case. A developer is unable to predict all future road situations, so letting the model train itself with a system of penalties and rewards in a varied environment is possibly the most effective way for the AI to broaden the experience it both has and collects.

1.5. Conclusion

The key distinguishing factor of reinforcement learning is how the agent is trained. Instead of inspecting the data provided, the model interacts with the environment, seeking ways to maximize the reward. In the case of deep reinforcement learning, a neural network is in charge of storing the experiences and thus improves the way the task is performed.

2. Reinforcement learning problem

Due to its generality, reinforcement learning is studied in many disciplines, such as game theory, control theory, operations research, information theory, simulation-based optimization, multi-agent systems, swarm intelligence, and statistics. In the operations research and control literature, reinforcement learning is called approximate dynamic programming, or neuro-dynamic programming. The problems of interest in reinforcement learning have also been studied in the theory of optimal control, which is concerned mostly with the existence and characterization of optimal solutions, and algorithms for their exact computation, and less with learning or approximation, particularly in the absence of a mathematical model of the environment. In economics and game theory, reinforcement learning may be used to explain how equilibrium may arise under bounded rationality.

Basic reinforcement is modeled as a Markov decision process (MDP):

- a set of environment and agent states, S
- a set of actions, A, of the agent
- $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability of transition (at time t) from state s to state s' under action a
- $R_a(s, s')$ is the immediate reward after transition from s to s' with action a

The purpose of reinforcement learning is for the agent to learn an optimal, or nearly-optimal, policy that maximizes the "reward function" or other user-provided reinforcement signal that accumulates from the immediate rewards. This is similar to processes that appear to occur in animal psychology. For example, biological brains are hardwired to interpret signals such as pain and hunger as negative reinforcements, and interpret pleasure and food intake as positive reinforcements. In some circumstances, animals can learn to engage in behaviors that optimize these rewards. This suggests that animals are capable of reinforcement learning.

A basic reinforcement learning agent AI interacts with its environment in discrete time steps. At each time t, the agent receives the current state s_t and reward r_t. It then chooses an action a_t from the set of available actions, which is subsequently sent to the environment. The environment moves to a new state s_{t+1} and the reward r_{t+1} associated with the "transition" (s_t, a_t, s_{t+1}) is determined. The goal of a reinforcement learning agent is to learn a "policy": $\pi : A \times S \rightarrow [0, 1]$, $\pi(a, s) = \Pr(a_t = a \mid s_t = s)$ which maximizes the expected cumulative reward.

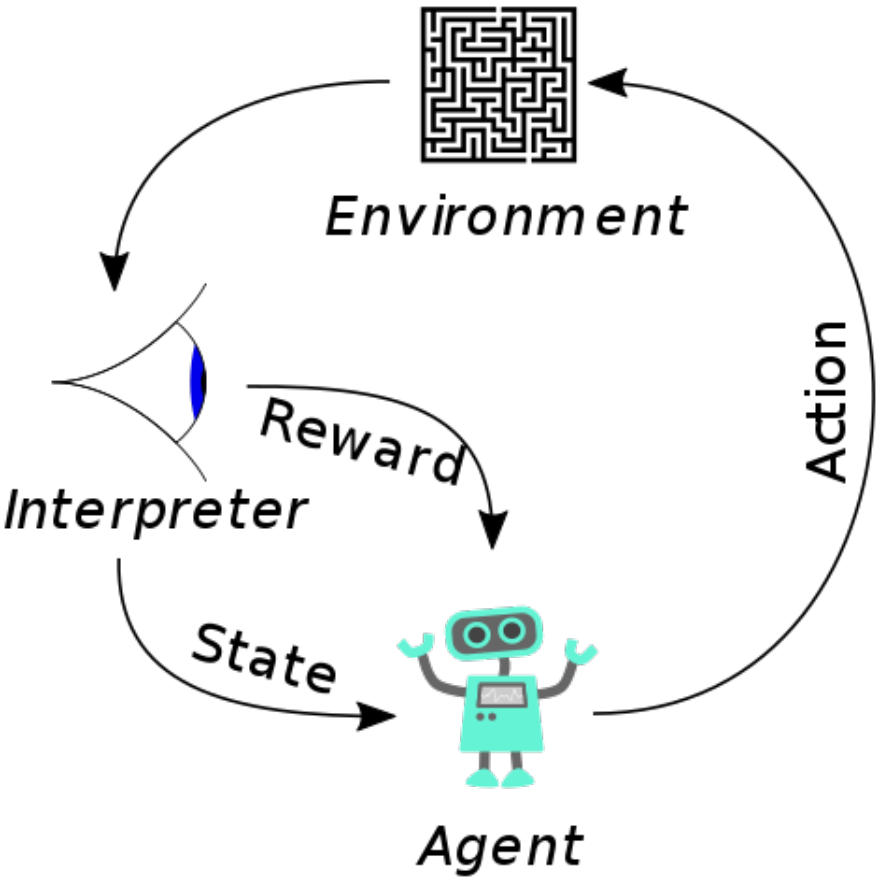


Figure 2.1. The typical framing of a Reinforcement Learning (RL) scenario: an agent takes actions in an environment, which is interpreted into a reward and a representation of the state, which are fed back into the agent.

Formulating the problem as an MDP assumes the agent directly observes the current environmental state; in this case the problem is said to have full observability. If the agent only has access to a subset of states, or if the observed states are corrupted by noise, the agent is said to have partial observability, and formally the problem must be formulated as a Partially observable Markov decision process. In both cases, the set of actions available to the agent can be restricted. For example, the state of an account balance could be restricted to be positive; if the current value of the state is 3 and the state transition attempts to reduce the value by 4, the transition will not be allowed.

When the agent's performance is compared to that of an agent that acts optimally, the difference in performance gives rise to the notion of regret. In order to act near optimally, the agent must reason about the long-term consequences of its actions (i.e., maximize future income), although the immediate reward associated with this might be negative.

Thus, reinforcement learning is particularly well-suited to problems that include a long-term versus short-term reward trade-off. It has been applied successfully to various problems, including robot control, elevator scheduling, telecommunications, backgammon, checkers and Go (AlphaGo).

Two elements make reinforcement learning powerful: the use of samples to optimize performance and the use of function approximation to deal with large environments. Thanks to these two key components, reinforcement learning can be used in large environments in the following situations:

A model of the environment is known, but an analytic solution is not available; Only a

simulation model of the environment is given (the subject of simulation-based optimization); The only way to collect information about the environment is to interact with it. The first two of these problems could be considered planning problems (since some form of model is available), while the last one could be considered to be a genuine learning problem. However, reinforcement learning converts both planning problems to machine learning problems.

2.1. Exploration

The exploration vs. exploitation trade-off has been most thoroughly studied through the multi-armed bandit problem and for finite state space MDPs in Burnetas and Katehakis (1997).

Reinforcement learning requires clever exploration mechanisms; randomly selecting actions, without reference to an estimated probability distribution, shows poor performance. The case of (small) finite Markov decision processes is relatively well understood. However, due to the lack of algorithms that scale well with the number of states (or scale to problems with infinite state spaces), simple exploration methods are the most practical.

One such method is ε -greedy, where $0 < \varepsilon < 1$ is a parameter controlling the amount of exploration vs. exploitation. With probability $1 - \varepsilon$, exploitation is chosen, and the agent chooses the action that it believes has the best long-term effect (ties between actions are broken uniformly at random). Alternatively, with probability ε , exploration is chosen, and the action is chosen uniformly at random. ε is usually a fixed parameter but can be adjusted either according to a schedule (making the agent explore progressively less), or adaptively based on heuristics.

3. Reinforcement learning methods

3.1. Value methods

Lets start by looking at the family of algorithms called a Q-Learning algorithms or Value algorithms. These are different from policy-based algorithms (we will look at them at chapter 2). Instead of starting with a complex and unwieldy deep neural network, we will begin by implementing a simple lookup-table version of the algorithm, and then show how to implement a neural-network equivalent using Tensorflow [15]. This should give an intuition into what is happening in Q-Learning that we can combine the policy gradient and Q-Learning to build state-of-the-art RL agents.

Unlike policy gradient methods, which attempt to learn functions which directly map an observation to an action, Q-Learning attempts to learn the value of being in a given state, and taking a specific action there. While both approaches ultimately allow us to take intelligent actions given a situation, the means of getting to that action differ significantly. Those DeepQ-Networks playing Atari games are just much larger and complex implementation of the basic Q-Learning algorithm we will talk about in this chapter.

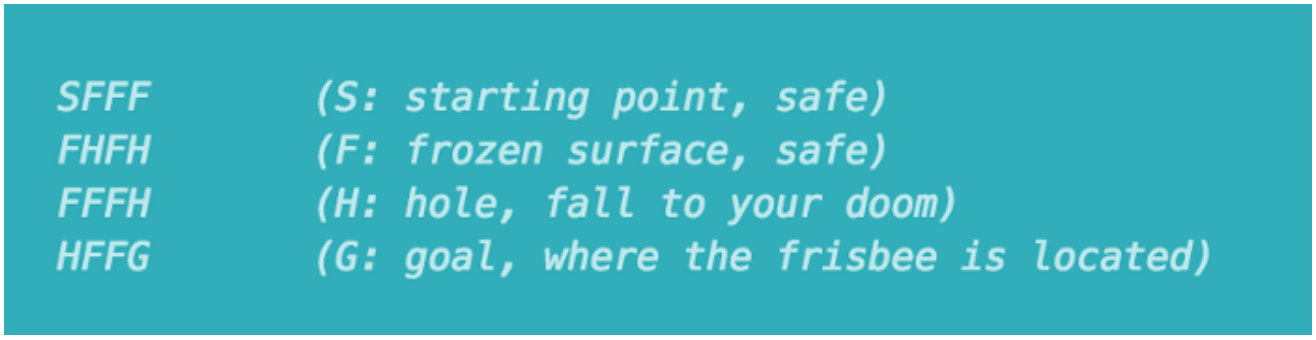


Figure 3.1. The rules of the FrozenLake environment.

For testing, we will attempt to solve the FrozenLake environment from OpenAI [9] gym. OpenAI gym is a python library, that provides an interface for creating your own environments for RL and a big array of implemented game environments, including Atari ones. The FrozenLake environment consists of a 4x4 grid of blocks, each one either being the start block, the goal block, a safe frozen block, or a dangerous hole. The objective is to have an agent learn to navigate from the start to the goal without moving onto a hole. At any given time the agent can choose to move either up, down, left, or right. The catch is that there is a wind which occasionally blows the agent onto a space they didn't choose. As such, perfect performance every time is impossible, but learning to avoid the holes and reach the goal are certainly still doable. The reward at every step is 0, except for entering the goal, which provides a reward of 1. Thus, we will need an algorithm that learns long-term expected rewards. This is exactly what Q-Learning is designed to provide.

3.1.1. Q-Table

In it's simplest implementation, Q-Learning is a table of values for every state (row) and action (column) possible in the environment. Within each cell of the table, we learn a value for how good it is to take a given action within a given state. In the case of the FrozenLake environment, we have 16 possible states (one for each block), and 4 possible actions (the four directions of movement), giving us a 16x4 table of Q-values. We start by initializing the table

to be uniform (all zeros), and then as we observe the rewards we obtain for various actions, we update the table accordingly.

We make updates to our Q-table using something called the Bellman equation [4], which states that the expected long-term reward for a given action is equal to the immediate reward from the current action combined with the expected reward from the best future action taken at the following state. In this way, we reuse our own Q-table when estimating how to update our table for future actions! In equation form, the rule looks like this:

$$Q(s,a) = r + \gamma(\max(Q(s',a')))$$

This says that the Q-value for a given state (s) and action (a) should represent the current reward (r) plus the maximum discounted (γ) future reward expected according to our own table for the next state (s') we would end up in. The discount variable allows us to decide how important the possible future rewards are compared to the present reward. By updating in this way, the table slowly begins to obtain accurate measures of the expected future reward for a given action in a given state. The code you can find in the appendix A.

3.1.2. Q-Learning with Neural Networks

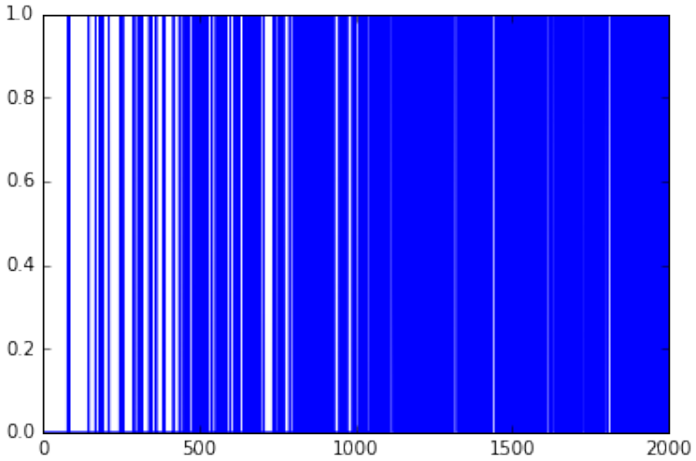
Now, you may be thinking: tables are great, but they don't really scale, do they? While it is easy to have a 16x4 table for a simple grid world, the number of possible states in any modern game or real-world environment is nearly infinitely larger. For most interesting problems, tables simply don't work. We instead need some way to take a description of our state, and produce Q-values for actions without a table: that is where neural networks come in. By acting as a function approximator, we can take any number of possible states that can be represented as a vector and learn to map them to Q-values.

In the case of the FrozenLake example, we will be using a one-layer network which takes the state encoded in a one-hot vector (1x16), and produces a vector of 4 Q-values, one for each action. Such a simple network acts kind of like a glorified table, with the network weights serving as the old cells. The key difference is that we can easily expand the Tensorflow network with added layers, activation functions, and different input types, whereas all that is impossible with a regular table. The method of updating is a little different as well. Instead of directly updating our table, with a network we will be using backpropagation and a loss function. Our loss function will be sum-of-squares loss, where the difference between the current predicted Q-values, and the "target" value is computed and the gradients passed through the network. In this case, our Q-target for the chosen action is the equivalent to the Q-value computed in equation 1 above.

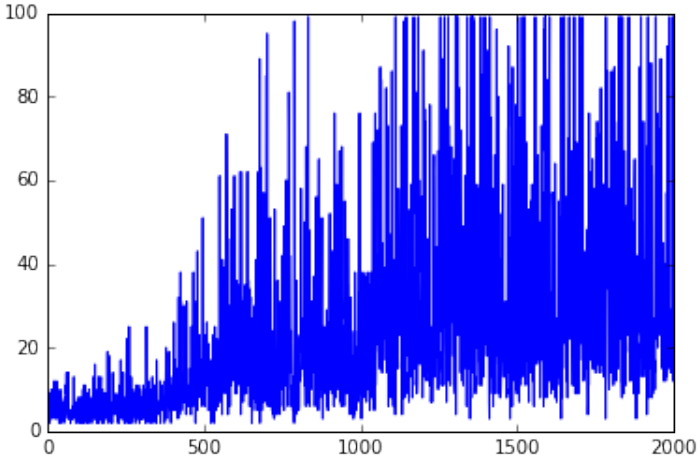
$$L = \sum (Q_{target} - Q)^2$$

where L is the loss function.

We can see that the network begins to consistly reach the goal around the 750 episode mark.



It also begins to progress through the environment for longer than chance around the 750 mark as well. That we also can see from a steps per episode plot:



You can find an implementation in the appendix B.

While the network learns to solve the FrozenLake problem, it turns out it doesn't do so quite as efficiently as the Q-Table. While neural networks allow for greater flexibility, they do so at the cost of stability when it comes to Q-Learning. There are a number of possible extensions to our simple Q-Network which allow for greater performance and more robust learning. Two tricks in particular are referred to as Experience Replay [11] and Freezing Target Networks [5]. Those improvements and other tweaks were the key to getting Atari-playing Deep Q-Networks.

3.2. Policy gradient methods

Different type of methods are Policy gradient methods. Policy gradient methods are fundamental to recent breakthroughs in using deep neural networks for control, from video games, to 3D locomotion [8], to Go. One of the features of this methods is the ability to scale up the hardware. Nowadays Moore's Law [7] is slowly ending and it is harder and harder to speed up the hardware to calculate something faster sequentially. One of the decisions is to go parallel! In Reinforcement Learning the ability to go multithreading is important: you calculate different actors on your environment in parallel and then process all of their gathered knowledge after they are finished. The design of policy gradient algorithms is such that you are able to do that.

Policy gradient methods work by computing an estimator of the policy gradient and plugging it into a stochastic gradient ascent algorithm [14]. The most commonly used gradient estimator has the form:

$$\tilde{g} = \tilde{E}_t[\nabla_\theta \log \pi_\theta(a_t|s_t)\tilde{A}_t]$$

where π_θ is a stochastic policy and $\tilde{A}_t$ is an estimator of the advantage function at timestep t. Here, the expectation $\tilde{E}_t[...]$ indicates the empirical average over a finite batch of samples, in an algorithm that alternates between sampling and optimization. Implementations that use automatic differentiation software work by constructing an objective function whose gradient is the policy gradient estimator; the estimator $\tilde{g}$ is obtained by differentiating the objective

$$L^{PG}(\theta) = \tilde{E}_t[\log \pi_\theta(a_t|s_t)\tilde{A}_t]$$

But getting good results via policy gradient methods is challenging because they are sensitive to the choice of stepsize — too small, and progress is hopelessly slow; too large and the signal is overwhelmed by the noise, or one might see catastrophic drops in performance. They also often have very poor sample efficiency, taking millions (or billions) of timesteps to learn simple tasks.

Researchers have sought to eliminate these flaws with approaches like TRPO [16] and ACER [12], by constraining or otherwise optimizing the size of a policy update. These methods have their own trade-offs — ACER is far more complicated than others, requiring the addition of code for off-policy corrections and a replay buffer, while only doing marginally better on the Atari benchmark; TRPO — though useful for continuous control tasks — isn't easily compatible with algorithms that share parameters between a policy and value function or auxiliary losses, like those used to solve problems in Atari and other domains where the visual input is significant.

3.2.1. Proximal Policy Optimization

With supervised learning, we can easily implement the cost function, run gradient descent on it, and be very confident that we'll get excellent results with relatively little hyperparameter tuning. The route to success in reinforcement learning isn't as obvious — the algorithms have many moving parts that are hard to debug, and they require substantial effort in tuning in order to get good results. OpenAI proposed a new algorithm called Proximal Policy Optimization (PPO) [10]. They compared it to different algorithms and found out that it performs better in average. PPO strikes a balance between ease of implementation, sample complexity, and ease of tuning, trying to compute an update at each step that minimizes the cost function while ensuring the deviation from the previous policy is relatively small.

OpenAI's idea was really simple, but powerfull. They changed the Loss function the way to clip some huge changes, following the same idea as TRPO:

$$L^{\text{CLIP}}(\theta) = \tilde{E}_t[\min(r_t(\theta)\tilde{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\tilde{A}_t)],$$

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

where:

- θ is the policy parameter
- $\tilde{E}_t$ denotes the empirical expectation over timesteps
- π_θ is a stochastic policy
- r_t is the ratio of the probability under the new and old policies, respectively
- $\tilde{A}_t$ is the estimated advantage at time t
- ϵ is a hyperparameter, usually 0.1 or 0.2

This objective implements a way to do a Trust Region update which is compatible with Stochastic Gradient Descent, and simplifies the algorithm by removing the KL penalty and need to make adaptive updates. In tests, this algorithm has displayed the best performance on continuous control tasks and almost matches ACER's performance on Atari, despite being far simpler to implement.

The PPO2 - is the improved version of this algorithm, that allows to use a GPU for calculations. It is also created and published by OpenAI and it runs approximately 3X faster than the first PPO on Atari.

3.3. Actor-Critic methods

In search of perfection the third type of algorithms was developed: Actor-Critic. It combined an idea of Q-Learning with a Policy gradient. It uses a mixed approach attempting to get the best of two.

3.3.1. Asynchronous Advantage Actor-Critic

The A3C algorithm [2] was released by Google's DeepMind group and it made a splash by essentially obsoleting DQN. It was faster, simpler, more robust, and able to achieve much better scores on the standard battery of Deep RL tasks. On top of all that it could work in continuous as well as discrete action spaces. Given this, it has become the go-to Deep RL algorithm for new challenging problems with complex state and action spaces.

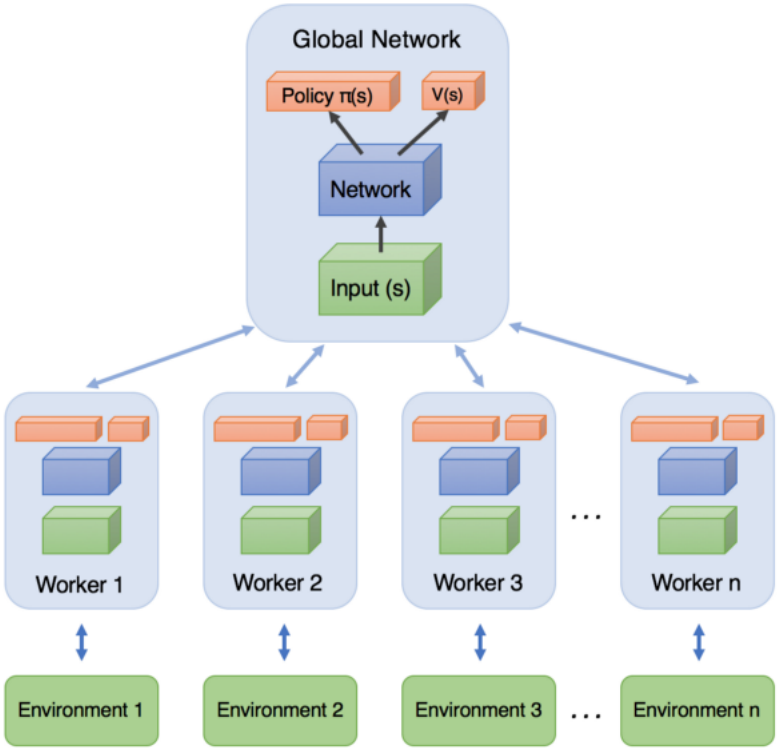


Figure 3.2. Diagram of A3C high-level architecture.

Asynchronous Advantage Actor-Critic is quite a mouthful. Let's start by unpacking the name, and from there, begin to unpack the mechanics of the algorithm itself.

Asynchronous: Unlike DQN, where a single agent represented by a single neural network interacts with a single environment, A3C utilizes multiple incarnations of the above in order to learn more efficiently. In A3C there is a global network, and multiple worker agents which each have their own set of network parameters. Each of these agents interacts with it's own copy of the environment at the same time (the same as in PPO) as the other agents are interacting with their environments. The reason this works better than having a single agent (beyond the speedup of getting more work done), is that the experience of each agent is independent of the experience of the others. In this way the overall experience available for training becomes more diverse.

Actor-Critic: So far we looked at Value methods such as Q-learning and Policy methods such as PPO. Actor-Critic combines the benefits of both approaches. In the case of A3C, our network will estimate both a value function $V(s)$ (how good a certain state is to be in) and a policy $\pi(s)$ (a set of action probability outputs). These will each be separate fully-connected layers sitting at the top of the network. Critically, the agent uses the value estimate (the critic) to update the policy (the actor) more intelligently than traditional policy gradient methods.

Advantage: In Policy Gradient, the update rule uses the discounted returns from a set of experiences in order to tell the agent which of its actions were "good" and which were "bad":

$$R = \gamma(r)$$

The network was then updated in order to encourage and discourage actions appropriately.

The insight of using advantage estimates rather than just discounted returns is to allow the agent to determine not just how good its actions were, but how much better they turned out to be than expected. Intuitively, this allows the algorithm to focus on where the network's predictions were lacking. In the Dueling Q-Network architecture [17], the advantage function is as follow:

$$A = Q(s, a) - V(s)$$

Since we won't be determining the Q values directly in A3C, we can use the discounted returns (R) as an estimate of $Q(s, a)$ to allow us to generate an estimate of the advantage:

$$A = R - V(s)$$

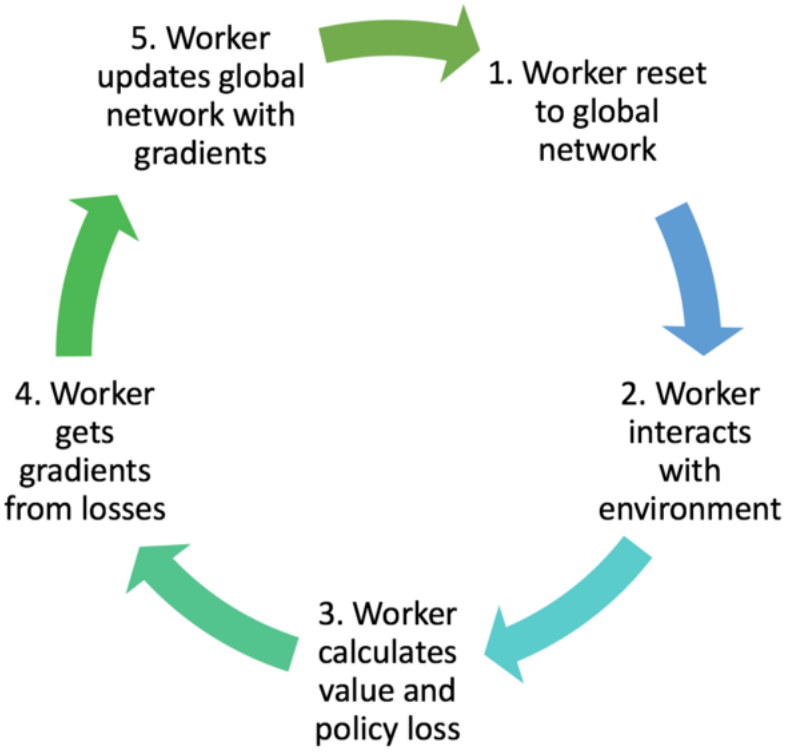


Figure 3.3. Training workflow of each worker agent in A3C.

You can find an example of A3C in appendix C

4. Comparative analysis of models

In Reinforcement Learning community it is common to test your algorithm on a set of environments, that are not too complicated for a machine, but still can be a problem for a human being to master. Mostly, such environment are presented in a OpenAI's GYM open source interface, that is used widely, as implementing this interface would mean that it is possible for their algorithms to use your custom environment. We will use some of their stochastic environments to train our algorithms to solve them. Algorithm implementations will be taken from Stable Baselines 3 library.

4.1. Space Invaders

The most classic environment is, for sure, an Atari game called ”Space Invaders”. It is not that easy even for people to get high scores, but it is possible for a RL algorithm to solve it using nothing but a screen image. Let’s first look into the rules of the game and how to control it.

4.1.1. Environment rules

The game is about defending from invading aliens, that shoot and move closer and closer to the ground. The player is green object on a ground, that is capable of moving left and right and shooting. From the beginning, there are 6 rows and 6 columns of alien entities, that repeatedly move from left to right, then from right to left, moving one step closer to the ground each time they hit a wall. The less aliens are left alive, the faster they move. Near the ground there are three protective entities, that get partially destroyed on each contact with a projectile and get totally destroyed if any alien get close enough to the earth. Aliens shoot randomly with some consistency. Hitting the player results in a player dying and loosing one of his three lives. If at any point in time an alien reaches the ground, the game is automatically finished, as well as if player looses all lives. The goal is to maximize the score, which is gained by destroying aliens and destroying a purple ”mystery” ship, that appears randomly in the top of the screen and moves from one direction to the other. If all of the aliens are destroyed, the game starts again, saving current amount of lives and points.

Figure 4.1. Space Invaders Screen

4.1.2. Preprocessing

The algorithm gets an input of all pixels from the game screen and is required to perform one of the many discreet actions, that contain moving left, right, shooting and some combinations of those. It is possible to decrease the input space at least in three times by converting an image to a gray-scale, and also it is possible to decrease it a few more times by scaling down an image. We will do the next preprocessing:

- Frame skip
- Max-pooling
- Grayscale
- Resize
- Clip reward

First, we will perform a frame skip. What it basicaly means, we will throw away some frames, performing actions only on n-th one. For our case we will use 4 frame skip, what means we will only use each 4th frame, doing nothing on 1st, 2nd and 3rd. This will increase training speed and learning speed with no significant drop in algorithm performance. Learning speed increases because the information density will become larger, as such games are designed for human players, that can not perform actions instantaneously, but the RL model can. So making actions only each 4th frame will cause no problem.

After that, we will perform max-pooling. It means, that the model will receive not only current frame, but also the last one. It allows the model to have a human-like feeling of motion and is actually a really important preprocessing step. It is also logical, because to be able to shoot a target, that is moving, there should be a concept of movement.

Then, as we said already, we will grayscale the observed image. This is an obvious step to decrease the input space at least 3 times without loosing any information, as there is no additional information about a game state in colors and the same amount of information will be conserved.

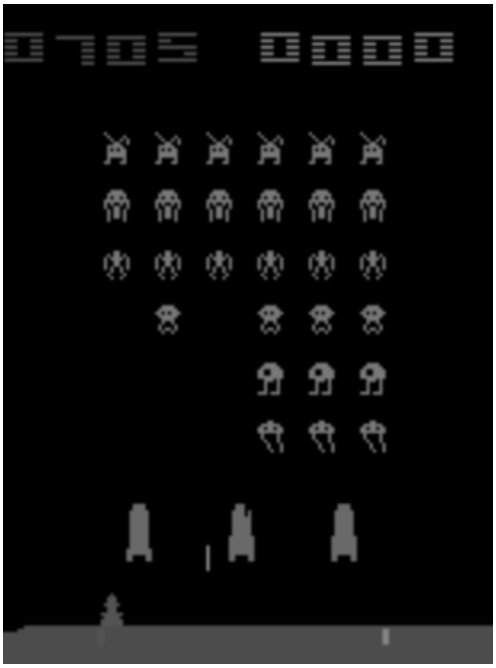


Figure 4.2. Space Invaders Screen Grayscaled

As we can see, it is still possible to understand everything that is going on and a model should have no problem understanding it.

Resize also helps to decrease the input space. Atari 2600 games (as well as emulated environments) use 160 x 192 pixels resolution screen. We will scale it down to 84 x 84 resolution.

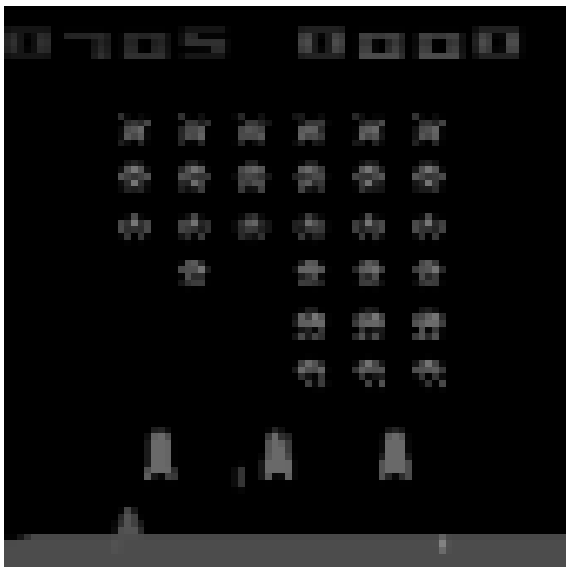


Figure 4.3. Space Invaders Screen Grayscaled and Resized

Despite now it is being a little harder for a human being to understand what is going on, it is still objectively possible to recognise shapes and no helpful information has been lost.

The last step is to do something with the reward function. In environment it works like that: each step reward is equal to amount of points, that a player received this frame, equals minus one if the player died and equals zero otherwise. We will clip it down, so any positive reward will give +1, any negative - -1 and 0 otherwise. So it will normalise the reward function.

Because all the algorithms will learn from the image data, the Convolutional Neural Network will be used.

4.1.3. DQN

The first step before trying to train the best model is to optimize hyperparameters. Hyperparameters are different algorithm-related variables. There is no way to find the best hyperparameters, but it is possible to make a value-based optimization based on training a model on an environment for a few iterations and comparing how good it performed. The best tool, that is used for such optimization in python is called Optuna. Optuna is a whole framework made for optimizing hyperparameters. It creates different studies, and randomly generate a point (values for hyperparameters), then calculates the value in this point and makes decisions based on it. It is also storing all data in a database, so you can analyse it later.

After a few days of Optuna working it found those parameters for DQN network:

- batch_size = 32
- buffer_size = 10000
- exploration_final_eps = 0.01
- exploration_fraction = 0.1
- gradient_steps = 1
- learning_rate = 0.0001

- learning_starts = 100000
- optimize_memory_usage = true
- target_update_interval = 1000
- train_freq = 4

Now, using those hyperparameters we will train a model. To make comparing of all algorithms fair, we will train all of them using 10 million training steps. Each training step, in our case, is a frame of the game. Each model's actions we will describe objectively and will calculate their mean score ($\overline{x}$) and it's standard deviation (σ). So, the trained model evaluations are:

$$\begin{cases} \overline{x} = 619.831 \\ \sigma = 202.824 \end{cases}$$

On the descriptive part, we can see, that model's actions are looking intelligent, that it learned to avoid aliens' shots and aiming on hitting it's shots. It can be seen, that model uses a strategy to destroy aliens by rows from the beginning, mostly leaving the farthest target for later. This can be considered logical, as for the farthest aliens it will take the most to reach the ground.

But there are also some problems. At the point, when model destroys most of the aliens it can be seen, that it has some problems hitting the last target and, mostly, the model just leaves the player standing still on one place, shooting all the possible time, just randomly trying to shoot it. This can be easily be considered not a logical strategy, as it will be much better to aim and hit the target on a fly. This can be caused by an increased speed of a target, that differs a lot of the speed of aliens on the beginning of the game. The thing is that the part of the game, where there is only one alien is left requires an algorithm to pass the whole previous stage of the round. This will surely be fixed by training for more iterations.

4.1.4. PPO2

As we already told in the first part, PPO2, on practice, is the most powerful algorithm of them all and from the most of the tasks it should perform better then others. So, we should expect a better result from it, but still, a result on the same level or below should not be a surprise. The same amount of Optuna optimisation gives us next hyperparameters:

- batch_size = 256
- clip_range = 0.1
- learning_rate = 2.5e-4
- n_envs = 8
- n_epochs = 4
- n_steps = 128
- vf_coef = 0.5

Now to train the model on 10 million steps and evaluate it:

$$\begin{cases} \overline{x} = 960.331 \\ \sigma = 425.355 \end{cases}$$

As we can see, $\overline{x}$ is greater for PPO, then for DQN. With that, σ is also higher, but the result can be definitely considered higher.

Objectively, the strategy of this algorithm is looking more advanced, it has much less problems on the latest stages of the game destroying last aliens. It has much more chances of clearing the first wave of the game, but still, it rarely reaches the third one. But, in comparison to the DQN, PPO2 was able to get to a higher level of playing the game in the same amount of training.

4.1.5. A3C

Now, let's also do the same steps for A3C. Optuna gives us those hyperparameters:

- ent_coef = 0.01
- n_envs = 16
- eps = 1e-5
- vf_coef = 0.25

After that training it for 10 millions steps, evaluation gives next metrics:

$$\begin{cases} \overline{x} = 627.160 \\ \sigma = 201.974 \end{cases}$$

By the metrics, it is a little bit better then DQN and can be considered almost identical result.

Subjectively, analysing it's gameplay, it feels a little bit better than DQN, but objectively it has a little bit more problems avoiding shots, but less problems in later stages. In total, as seen by the metrics, results are really similar.

4.1.6. Training Results

After 10 million frames of training, we have the next data:

Table 4.1

Algorithms on Space Invaders 10 million frames

Algorithm	$\overline{x}$	σ
DQN	619.831	202.824
PPO2	960.331	425.355
A3C	627.160	201.974

From this result, we can make conclusions, that PPO2 is the best algorithm compared. After it, A3C is the next best with almost similar results to DQN

4.2. Seaquest

Seaquest is an Atari 2600 video game designed by Steve Cartwright and published by Activision in 1983. The game is an underwater shooter in which the player controls a submarine.

4.2.1. Environment rules

The player uses a submarine to shoot at enemies and rescue divers. Enemies include sharks and submarines, which shoot missiles at the player's submarine . The player must ward off the enemies by firing an unlimited supply of missiles while trying to rescue divers swimming through the water. The points awarded to the player for shooting an enemy starts at 10 points each, and increases as the game advances. The sub can hold up to six divers at a time. Each time the player resurfaces prior to having a full load of six divers, one of the divers is removed.

Figure 4.4. Seaquest Screen

The submarine has a limited amount of oxygen. The player must surface often in order to replenish the oxygen, but if the player resurfaces without any rescued divers, they will lose a life. If the player resurfaces with the maximum amount of divers, they will gain bonus points for the sub's remaining oxygen. Each time the player surfaces, the game's difficulty

increases; enemies increase in number and speed. Eventually an enemy sub begins patrolling the surface, leaving the player without a safe haven.

The player starts the game with 3 extra lives, and is awarded an additional extra life each time the player scores 10,000 points. Also, we will use the exact same methods of preprocessing, as in previous chapter.

4.2.2. DQN

Optuna gives us next hyperparameters:

- batch_size = 32
- buffer_size = 10000
- exploration_final_eps = 0.01
- exploration_fraction = 0.1
- gradient_steps = 1
- learning_rate = 0.0001
- learning_starts = 100000
- optimize_memory_usage = true
- target_update_interval = 1000
- train_freq = 4

After 10 million steps, evaluation is:

$$\begin{cases} \overline{x} = 2000.290 \\ \sigma = 606.644 \end{cases}$$

4.2.3. PPO2

Optuna gives us next hyperparameters:

- batch_size = 256
- clip_range = 0.1
- learning_rate = 2.5e-4
- n_envs = 8
- n_epochs = 4
- n_steps = 128
- vf_coef = 0.5

After 10 million steps, evaluation is:

$$\begin{cases} \overline{x} = 1783.636 \\ \sigma = 34.096 \end{cases}$$

4.2.4. A3C

Optuna gives us next hyperparameters:

- ent_coef = 0.01
- n_envs = 16
- eps = 1e-5
- vf_coef = 0.25

After 10 million steps, evaluation is:

$$\begin{cases} \overline{x} = 1721.493 \\ \sigma = 105.339 \end{cases}$$

4.2.5. Results

After 10 million frames of training, we have the next data:

Table 4.2

Algorithms on Seaquest 10 million frames

Algorithm	$\overline{x}$	σ
DQN	2000.290	606.644
PPO2	1783.636	34.096
A3C	1721.493	105.339

4.3. QBert

4.3.1. Environment rules

Q*bert is an action game with puzzle elements played from an axonometric third-person perspective to convey a three-dimensional look. The game is played using a single, diagonally mounted four-way joystick. The player controls Q*bert, who starts each game at the top of a pyramid made of 28 cubes, and moves by hopping diagonally from cube to cube. Landing on a cube causes it to change color, and changing every cube to the target color allows the player to progress to the next stage.

At the beginning, jumping on every cube once is enough to advance. In later stages, each cube must be hit twice to reach the target color. Other times, cubes change color every time Q*bert lands on them, instead of remaining on the target color once they reach it. Both elements are then combined in subsequent stages. Jumping off the pyramid results in the character's death.

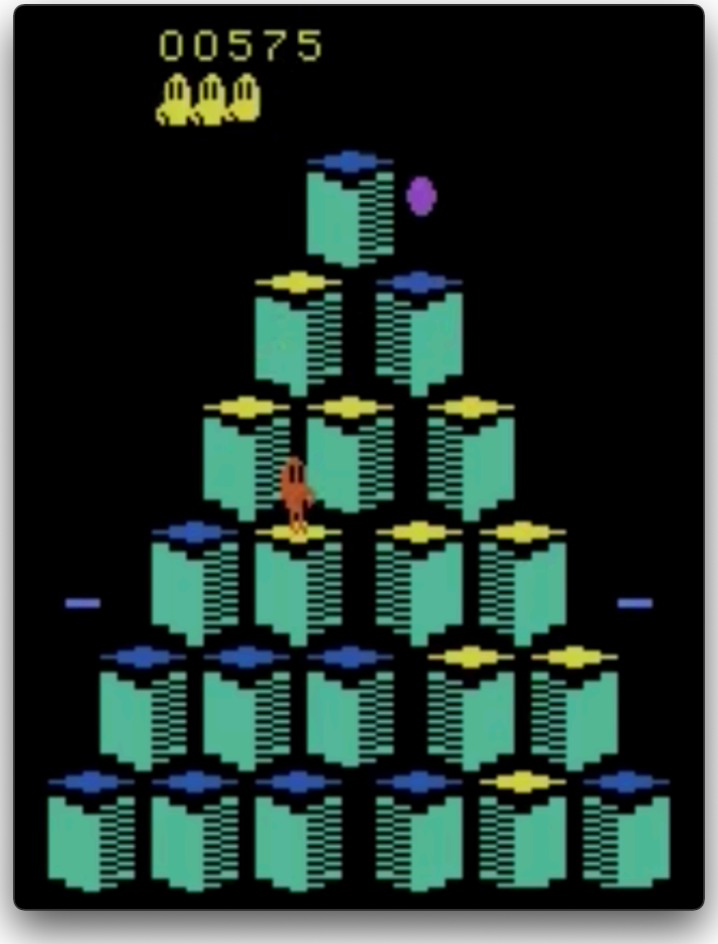


Figure 4.5. QBert Screen

The player is impeded by several enemies, introduced gradually to the game:

- Coily – Coily first appears as a purple egg that bounces to the bottom of the pyramid and then transforms into a snake that chases after Q*bert. He is often considered the main antagonist and Q*bert's arch-nemesis.

- Ugg and Wrongway – Two purple creatures that hop along the sides of the cubes in an Escheresque manner. Starting at either the bottom left or bottom right corner, they keep moving toward the top right or top left side of the pyramid respectively and fall off the pyramid when they reach the end.
- Slick and Sam – Two green creatures that descend down the pyramid and revert cubes whose color has already been changed.

A collision with purple enemies is fatal to the character, whereas the green enemies are removed from the board upon contact. Colored balls occasionally appear at the second row of cubes and bounce downward; contact with a red ball is lethal to Q*bert, while contact with a green one immobilizes the on-screen enemies for a limited time. Multicolored floating discs on either side of the pyramid serve as an escape from danger, particularly Coily. When Q*bert jumps on a disc, it transports him to the top of the pyramid. If Coily is in close pursuit of the character, he will jump after Q*bert and fall to his death, awarding bonus points. This causes all enemies and balls on the screen to disappear, though they start to return after a few seconds.

Points are awarded for each color change (25), defeating Coily with a flying disc (500), remaining discs at the end of a stage (at higher stages, 50 or 100) and catching green balls (100) or Slick and Sam (300 each). Bonus points are also awarded for completing a screen, starting at 1,000 for the first screen of Level 1 and increasing by 250 for each subsequent completion, up to 5,000 after Level 4. Extra lives are granted for reaching certain scores, which are set by the machine operator.

4.3.2. DQN

Optuna gives us next hyperparameters:

- batch_size = 32
- buffer_size = 10000
- exploration_final_eps = 0.01
- exploration_fraction = 0.1
- gradient_steps = 1
- learning_rate = 0.0001
- learning_starts = 100000
- optimize_memory_usage = true
- target_update_interval = 1000
- train_freq = 4

After 10 million steps, evaluation is:

$$\begin{cases} \overline{x} = 9496.774 \\ \sigma = 5399.633 \end{cases}$$

4.3.3. PPO2

Optuna gives us next hyperparameters:

- batch_size = 256
- clip_range = 0.1
- learning_rate = 2.5e-4
- n_envs = 8
- n_epochs = 4
- n_steps = 128
- vf_coef = 0.5

After 10 million steps, evaluation is:

$$\begin{cases} \overline{x} = 15627.108 \\ \sigma = 3313.538 \end{cases}$$

4.3.4. A3C

Optuna gives us next hyperparameters:

- ent_coef = 0.01
- n_envs = 16
- eps = 1e-5
- vf_coef = 0.25

After 10 million steps, evaluation is:

$$\begin{cases} \overline{x} = 3882.345 \\ \sigma = 1223.327 \end{cases}$$

4.3.5. Results

After 10 million frames of training, we have the next data:

Table 4.3

Algorithms on Qbert 10 million frames

Algorithm	$\overline{x}$	σ
DQN	9496.774	5399.633
PPO2	15627.108	3313.538
A3C	3882.345	1223.327

Висновки

Отже, ми розглянули 3 різні класи алгоритмів навчання з підкріпленням:

- Значення алгоритмів
- Алгоритми політики
- Актор-критичні алгоритми

На практиці два-три алгоритми, як правило, використовуються більше, ніж інші, оскільки в більшості проблем вони, як правило, дають найкращі результати. У цій роботі ми розглянули деякі з найкращих: PPO2 та A3C. Це дійсно потужні алгоритми, які можна використовувати для гри в шахи або навіть StarCraft.

Крім того, ми розглянули найбільш основні: найпростіший Q-Table та більш потужний Q-Learning. Незважаючи на те, що він був потужнішим, це призвело до гіршої поведінки: Q-Table отримував більше очок у грі FrozenLake. Але це лише через те, що сама гра дійсно примітивна і проста. Для інших завдань Q-таблиці марні: змусити його працювати стає занадто важко.

Але навіть коли йдеться про реальну проблему, "срібної кулі" не існує. Якщо подивитись на ефективність Atari Games (додаток B у [10]), ви зможете побачити, що порівняння між A2C (синхронізована версія A3C), ACER та PPO дає цікаві результати: ACER, будучи найскладніший з усіх них, він займає трохи більше перших місць, ніж PPO, будучи простим. А A2C веде лише в одній грі. Але якщо ви хочете отримати найкращу продуктивність, спробуйте підійти до своєї проблеми, використовуючи різні алгоритми. Для цього існує кілька бібліотек. Вони містять купу попередньо записаних алгоритмів, а також деякі інші інструменти для RL. Звідти ви можете просто вибрати свій алгоритм, налаштувати деякі налаштування та зробити спробу, не маючи величезної купи коду.

Наступним кроком має бути використання алгоритмів для рішення нових та/або набагато складніших задач.

Робота була частиною дослідницького проекту [1], і описані методи використовувались для спроби торгівлі та прогнозування ціни криптовалюти. Незважаючи на те, вони не дали достатньо хороших результатів, тому не були включені в остаточну статтю.

Conclusion

So, we looked at 3 different classes of Reinforcement Learning algorithms:

- Value algorithms
- Policy algorithms
- Actor-Critic algorithms

On practice, two or three algorithms tend to be used more then the others, because in most of the problems they tend to give the best results. In this work we looked at some of the best ones: PPO2 and A3C. Those are really powerful algorithms that could be used to play chess or even StarCraft.

Also, we looked at the more basic ones: the most simple Q-Table and more powerful Q-Learning. Despite being more powerful, it resulted in a worse behavior: Q-Table was getting more points in FrozenLake game. But, this is just because of the game itself being really primitive and simple. On other tasks Q-Tables are useless: it starts to be too hard to get it to work.

But even when it comes to a real problem, there is no "Silver bullet". If you look at the performance on Atari Games (B appendix in [10]) you can see, that comparison between A2C (sync version of A3C), ACER and PPO gives interesting results: ACER, being the most complex from them all, it taking a little bit more first places, than a PPO, being the easy one. And A2C leads in only one game. But if you want to get a best performance you should try approaching your problem using different algorithms. For that, there exist some libraries. They contain a bunch of pre-written algorithms as well as some other instruments for RL. From there, you can just select your algorithm, configure some settings and make an attempt, without a huge bunch of code.

The next step should be using those algorithms for solving new or/and much harder problems.

The work was a part of a research project [1] and methods described were used to try to trade and predict cryptocurrency price. Despite, they did not gave good enough results, so were not included in a final article.

References

1. Analysis of neural network models LSTM and GMDH for cryptocurrency forecasting / M. Volodimyr [et al.] // Bulletin of the National Technical University Kharkiv Polytechnic Institute. — 2020.
2. Asynchronous Methods for Deep Reinforcement Learning / V. Mnih [et al.] // CoRR. — 2016. — Vol. abs/1602.01783. — arXiv: 1602.01783. — URL: http://arxiv.org/abs/1602.01783.
3. Atari — creator of the classic computer games. — URL: https://en.wikipedia.org/wiki/Atari.
4. Bellman equation. — URL: https://en.wikipedia.org/wiki/Bellman_equation.
5. Deep Q-Network (DQN) article from towardsdatascience.com. — URL: https://towardsdatascience.com/deep-q-network-dqn-ii-b6bf911b6b2c.
6. DeepMind — Google's company that specializes on AI. — URL: https://en.wikipedia.org/wiki/DeepMind.
7. Moore's law. — URL: https://en.wikipedia.org/wiki/Moore%5C%27s_law.
8. Multi-Joint dynamics with Contact. — URL: http://www.mujoco.org/.
9. OpenAI — Elon Musk's AI research company. — URL: https://openai.com/.
10. Proximal Policy Optimization Algorithms / J. Schulman [et al.] // CoRR. — 2017. — Vol. abs/1707.06347. — arXiv: 1707.06347. — URL: http://arxiv.org/abs/1707.06347.
11. Revisiting Fundamentals of Experience Replay. — URL: https://arxiv.org/abs/2007.06700.
12. Sample Efficient Actor-Critic with Experience Replay / Z. Wang [et al.] // CoRR. — 2016. — Vol. abs/1611.01224. — arXiv: 1611.01224. — URL: http://arxiv.org/abs/1611.01224.
13. StarCraft — really popular strategy game. — URL: https://en.wikipedia.org/wiki/StarCraft.
14. Stochastic Gradient Descent article from towardsdatascience.com. — URL: https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31.
15. TensorFlow — a Python framework for machine learning. — URL: https://www.tensorflow.org/.
16. Trust Region Policy Optimization / J. Schulman [et al.] // CoRR. — 2015. — Vol. abs/1502.05477. — arXiv: 1502.05477. — URL: http://arxiv.org/abs/1502.05477.
17. *Wang Z.*, *Freitas N. de*, *Lanctot M.* Dueling Network Architectures for Deep Reinforcement Learning // CoRR. — 2015. — Vol. abs/1511.06581. — arXiv: 1511.06581. — URL: http://arxiv.org/abs/1511.06581.
18. *Wydmuch M.*, *Kempka M.*, *Jaśkowski W.* ViZDoom Competitions: Playing Doom from Pixels // IEEE Transactions on Games. — 2018.

A. Q-Table for FrozenLake

```python
import gym
import numpy as np

env = gym.make("FrozenLake-v0")

# Initialize table with all zeros
Q = np.zeros([env.observation_space.n, env.action_space.n])
# Set learning parameters
lr = 0.8
y = 0.95
num_episodes = 2000
# create lists to contain total rewards and steps per episode
# jList = []
rList = []
for i in range(num_episodes):
    # Reset environment and get first new observation
    s = env.reset()
    rAll = 0
    d = False
    j = 0
    # The Q-Table learning algorithm
    while j < 99:
        j += 1
        # Choose an action by greedily (with noise) picking from Q table
        a = np.argmax(
            Q[s, :] + np.random.randn(1, env.action_space.n) * (1.0 / (i
            ↪  + 1))
        )
        # Get new state and reward from environment
        s1, r, d, _ = env.step(a)
        # Update Q-Table with new knowledge
        Q[s, a] = Q[s, a] + lr * (r + y * np.max(Q[s1, :]) - Q[s, a])
        rAll += r
        s = s1
        if d == True:
            break
    # jList.append(j)
    rList.append(rAll)

print("Score over time: " + str(sum(rList) / num_episodes))

```

Listing 1: FrozenLake attempt using a Q-Table

B. Q-Network for FrozenLake

```
import gym
import numpy as np
import random
import tensorflow as tf
import matplotlib.pyplot as plt

env = gym.make("FrozenLake-v0")

tf.reset_default_graph()

# These lines establish the feed-forward part of the network used to
↪ choose actions
inputs1 = tf.placeholder(shape=[1, 16], dtype=tf.float32)
W = tf.Variable(tf.random_uniform([16, 4], 0, 0.01))
Qout = tf.matmul(inputs1, W)
predict = tf.argmax(Qout, 1)

# Below we obtain the loss by taking the sum of squares difference
↪ between the target and prediction Q values.
nextQ = tf.placeholder(shape=[1, 4], dtype=tf.float32)
loss = tf.reduce_sum(tf.square(nextQ - Qout))
trainer = tf.train.GradientDescentOptimizer(learning_rate=0.1)
updateModel = trainer.minimize(loss)

init = tf.initialize_all_variables()

# Set learning parameters
y = 0.99
e = 0.1
num_episodes = 2000
# create lists to contain total rewards and steps per episode
jList = []
rList = []
with tf.Session() as sess:
    sess.run(init)
    for i in range(num_episodes):
        # Reset environment and get first new observation
        s = env.reset()
        rAll = 0
        d = False
        j = 0
        # The Q-Network
        while j < 99:
            j += 1
            # Choose an action by greedily (with e chance of random
            ↪ action) from the Q-network
            a, allQ = sess.run(
                [predict, Qout], feed_dict={inputs1: np.identity(16)[s :
                ↪ s + 1]}
            )
            if np.random.rand(1) < e:
                a[0] = env.action_space.sample()
            # Get new state and reward from environment
            s1, r, d, _ = env.step(a[0])
            # Obtain the Q' values by feeding the new state through our
            ↪ network
```

```
            Q1 = sess.run(Qout, feed_dict={inputs1: np.identity(16)[s1 :
            ↪  s1 + 1]})
            # Obtain maxQ' and set our target value for chosen action.
            maxQ1 = np.max(Q1)
            targetQ = allQ
            targetQ[0, a[0]] = r + y * maxQ1
            # Train our network using target and predicted Q values
            _, W1 = sess.run(
                [updateModel, W],
                feed_dict={inputs1: np.identity(16)[s : s + 1], nextQ:
                ↪  targetQ},
            )
            rAll += r
            s = s1
            if d == True:
                # Reduce chance of random action as we train the model.
                e = 1.0 / ((i / 50) + 10)
                break
        jList.append(j)
        rList.append(rAll)

print(f"Percent of succesful episodes: {sum(rList)/num_episodes}%")
```

Listing 2: FrozenLake attempt using a Q-Network

C. A3C for VisDoom [18]

The general outline of the code architecture is:

- AC_Network — This class contains all the Tensorflow ops to create the networks themselves.
- Worker — This class contains a copy of AC_Network, an environment class, as well as all the logic for interacting with the environment, and updating the global network.
- High-level code for establishing the Worker instances and running them in parallel.

The next example is the A3C implementation for learning VisDoom environment on python. VisDoom is an environment, that allows your algorithms to play a classic Doom game. Let's start with importing all we need:

```
import multiprocessing
import os
import threading
from random import choice
from time import sleep, time

import numpy as np
import matplotlib.pyplot as plt
import scipy.signal
import tensorflow as tf
import tensorflow.contrib.slim as slim
from vizdoom import *
```

Also we will define some helper functions that we will use later:

```
# Copies one set of variables to another.
# Used to set worker network parameters to those of global network.
def update_target_graph(from_scope, to_scope):
    from_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
    ↪  from_scope)
    to_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
    ↪  to_scope)

    op_holder = []
    for from_var, to_var in zip(from_vars, to_vars):
        op_holder.append(to_var.assign(from_var))
    return op_holder


# Processes Doom screen image to produce cropped and resized image.
def process_frame(frame):
    s = frame[10:-10, 30:-30]
    s = scipy.misc.imresize(s, [84, 84])
    s = np.reshape(s, [np.prod(s.shape)]) / 255.0
    return s


# Discounting function used to calculate discounted returns.
def discount(x, gamma):
    return scipy.signal.lfilter([1], [1, -gamma], x[::-1], axis=0)[::-1]


# Used to initialize weights for policy and value output layers
def normalized_columns_initializer(std=1.0):
    def _initializer(shape, dtype=None, partition_info=None):
        out = np.random.randn(*shape).astype(np.float32)
        out *= std / np.sqrt(np.square(out).sum(axis=0, keepdims=True))
        return tf.constant(out)

    return _initializer
```

The A3C algorithm begins by constructing the global network. This network will consist of convolutional layers to process spatial dependencies, followed by an LSTM layer to process temporal dependencies, and finally, value and policy output layers. Below is example code for establishing the network graph itself.

```
class AC_Network:
    def __init__(self, s_size, a_size, scope, trainer):
        with tf.variable_scope(scope):
            # Input and visual encoding layers
            self.inputs = tf.placeholder(shape=[None, s_size],
            ↪  dtype=tf.float32)
            self.imageIn = tf.reshape(self.inputs, shape=[-1, 84, 84, 1])
            self.conv1 = slim.conv2d(
                activation_fn=tf.nn.elu,
                inputs=self.imageIn,
                num_outputs=16,
                kernel_size=[8, 8],
                stride=[4, 4],
                padding="VALID",
```

```
            )
            self.conv2 = slim.conv2d(
                activation_fn=tf.nn.elu,
                inputs=self.conv1,
                num_outputs=32,
                kernel_size=[4, 4],
                stride=[2, 2],
                padding="VALID",
            )
            hidden = slim.fully_connected(
                slim.flatten(self.conv2), 256, activation_fn=tf.nn.elu
            )

            # Recurrent network for temporal dependencies
            lstm_cell = tf.contrib.rnn.BasicLSTMCell(256,
             ↪  state_is_tuple=True)
            c_init = np.zeros((1, lstm_cell.state_size.c), np.float32)
            h_init = np.zeros((1, lstm_cell.state_size.h), np.float32)
            self.state_init = [c_init, h_init]
            c_in = tf.placeholder(tf.float32, [1,
             ↪  lstm_cell.state_size.c])
            h_in = tf.placeholder(tf.float32, [1,
             ↪  lstm_cell.state_size.h])
            self.state_in = (c_in, h_in)
            rnn_in = tf.expand_dims(hidden, [0])
            step_size = tf.shape(self.imageIn)[:1]
            state_in = tf.contrib.rnn.LSTMStateTuple(c_in, h_in)
            lstm_outputs, lstm_state = tf.nn.dynamic_rnn(
                lstm_cell,
                rnn_in,
                initial_state=state_in,
                sequence_length=step_size,
                time_major=False,
            )
            lstm_c, lstm_h = lstm_state
            self.state_out = (lstm_c[:1, :], lstm_h[:1, :])
            rnn_out = tf.reshape(lstm_outputs, [-1, 256])

            # Output layers for policy and value estimations
            self.policy = slim.fully_connected(
                rnn_out,
                a_size,
                activation_fn=tf.nn.softmax,
                weights_initializer=normalized_columns_initializer(0.01),
                biases_initializer=None,
            )
            self.value = slim.fully_connected(
                rnn_out,
                1,
                activation_fn=None,
                weights_initializer=normalized_columns_initializer(1.0),
                biases_initializer=None,
            )

            # Only the worker network need ops for loss functions and
             ↪  gradient updating.
            if scope != "global":
                self.actions = tf.placeholder(shape=[None],
                 ↪  dtype=tf.int32)
```

```
                self.actions_onehot = tf.one_hot(self.actions, a_size,
                ↪  dtype=tf.float32)
                self.target_v = tf.placeholder(shape=[None],
                ↪  dtype=tf.float32)
                self.advantages = tf.placeholder(shape=[None],
                ↪  dtype=tf.float32)

                self.responsible_outputs = tf.reduce_sum(
                    self.policy * self.actions_onehot, [1]
                )

                # Loss functions
                self.value_loss = 0.5 * tf.reduce_sum(
                    tf.square(self.target_v - tf.reshape(self.value,
                    ↪  [-1]))
                )
                self.entropy = -tf.reduce_sum(self.policy *
                ↪  tf.log(self.policy))
                self.policy_loss = -tf.reduce_sum(
                    tf.log(self.responsible_outputs) * self.advantages
                )
                self.loss = (
                    0.5 * self.value_loss + self.policy_loss -
                    ↪  self.entropy * 0.01
                )

                # Get gradients from local network using local losses
                local_vars =
                ↪  tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                ↪  scope)
                self.gradients = tf.gradients(self.loss, local_vars)
                self.var_norms = tf.global_norm(local_vars)
                grads, self.grad_norms =
                ↪  tf.clip_by_global_norm(self.gradients, 40.0)

                # Apply local gradients to global network
                global_vars = tf.get_collection(
                    tf.GraphKeys.TRAINABLE_VARIABLES, "global"
                )
                self.apply_grads = trainer.apply_gradients(zip(grads,
                ↪  global_vars))
```

Next, a set of worker agents, each with their own network and environment are created. Each of these workers are run on a separate processor thread, so there should be no more workers than there are threads on your CPU. Each worker begins by setting its network parameters to those of the global network. We can do this by constructing a Tensorflow op which sets each variable in the local worker network to the equivalent variable value in the global network. Each worker then interacts with its own copy of the environment and collects experience. Each keeps a list of experience tuples (observation, action, reward, done, value) that is constantly added to from interactions with the environment. Once the worker's experience history is large enough, we use it to determine discounted return and advantage, and use those to calculate value and policy losses. We also calculate an entropy (H) of the policy. This corresponds to the spread of action probabilities. If the policy outputs actions with relatively similar probabilities, then entropy will be high, but if the policy suggests a single action with a large probability then entropy will be low. We use the entropy as a means of improving exploration, by encouraging the model to be conservative regarding its sureness

of the correct action.

$$L^{\text{Value}} = \sum (R - V(s))^2$$

$$L^{\text{Policy}} = -\log(\pi(s)) \cdot A(s) - \beta \cdot H(\pi)$$

A worker then uses these losses to obtain gradients with respect to its network parameters. Each of these gradients are typically clipped in order to prevent overly-large parameter updates which can destabilize the policy. Then it uses the gradients to update the global network parameters. In this way, the global network is constantly being updated by each of the agents, as they interact with their environment. Once a successful update is made to the global network, the whole process repeats! The worker then resets its own network parameters to those of the global network, and the process begins again.

```
class Worker:
    def __init__(
        self, game, name, s_size, a_size, trainer, model_path,
        ↪  global_episodes
    ):
        self.name = "worker_" + str(name)
        self.number = name
        self.model_path = model_path
        self.trainer = trainer
        self.global_episodes = global_episodes
        self.increment = self.global_episodes.assign_add(1)
        self.episode_rewards = []
        self.episode_lengths = []
        self.episode_mean_values = []
        self.summary_writer = tf.summary.FileWriter("train_" +
        ↪  str(self.number))

        # Create the local copy of the network and the tensorflow op to
        ↪  copy global paramters to local network
        self.local_AC = AC_Network(s_size, a_size, self.name, trainer)
        self.update_local_ops = update_target_graph("global", self.name)

        # The Below code is related to setting up the Doom environment
        game.set_doom_scenario_path(
            "basic.wad"
        )  # This corresponds to the simple task we will pose our agent
        game.set_doom_map("map01")
        game.set_screen_resolution(ScreenResolution.RES_160X120)
        game.set_screen_format(ScreenFormat.GRAY8)
        game.set_render_hud(False)
        game.set_render_crosshair(False)
        game.set_render_weapon(True)
        game.set_render_decals(False)
        game.set_render_particles(False)
        game.add_available_button(Button.MOVE_LEFT)
        game.add_available_button(Button.MOVE_RIGHT)
        game.add_available_button(Button.ATTACK)
        game.add_available_game_variable(GameVariable.AMMO2)
        game.add_available_game_variable(GameVariable.POSITION_X)
        game.add_available_game_variable(GameVariable.POSITION_Y)
        game.set_episode_timeout(300)
        game.set_episode_start_time(10)
        game.set_window_visible(False)
        game.set_sound_enabled(False)
```

```
            game.set_living_reward(-1)
            game.set_mode(Mode.PLAYER)
            game.init()
            self.actions = self.actions = np.identity(a_size,
            ↪  dtype=bool).tolist()
            # End Doom set-up
            self.env = game

        def train(self, rollout, sess, gamma, bootstrap_value):
            rollout = np.array(rollout)
            observations = rollout[:, 0]
            actions = rollout[:, 1]
            rewards = rollout[:, 2]
            next_observations = rollout[:, 3]
            values = rollout[:, 5]

            # Here we take the rewards and values from the rollout, and use
            ↪  them to
            # generate the advantage and discounted returns.
            # The advantage function uses "Generalized Advantage Estimation"
            self.rewards_plus = np.asarray(rewards.tolist() +
            ↪  [bootstrap_value])
            discounted_rewards = discount(self.rewards_plus, gamma)[:-1]
            self.value_plus = np.asarray(values.tolist() + [bootstrap_value])
            advantages = rewards + gamma * self.value_plus[1:] -
            ↪  self.value_plus[:-1]
            advantages = discount(advantages, gamma)

            # Update the global network using gradients from loss
            # Generate network statistics to periodically save
            feed_dict = {
                self.local_AC.target_v: discounted_rewards,
                self.local_AC.inputs: np.vstack(observations),
                self.local_AC.actions: actions,
                self.local_AC.advantages: advantages,
                self.local_AC.state_in[0]: self.batch_rnn_state[0],
                self.local_AC.state_in[1]: self.batch_rnn_state[1],
            }
            v_l, p_l, e_l, g_n, v_n, self.batch_rnn_state, _ = sess.run(
                [
                    self.local_AC.value_loss,
                    self.local_AC.policy_loss,
                    self.local_AC.entropy,
                    self.local_AC.grad_norms,
                    self.local_AC.var_norms,
                    self.local_AC.state_out,
                    self.local_AC.apply_grads,
                ],
                feed_dict=feed_dict,
            )
            return v_l / len(rollout), p_l / len(rollout), e_l /
            ↪  len(rollout), g_n, v_n

        def work(self, max_episode_length, gamma, sess, coord, saver):
            episode_count = sess.run(self.global_episodes)
            total_steps = 0
            print("Starting worker " + str(self.number))
            with sess.as_default(), sess.graph.as_default():
                while not coord.should_stop():
                    sess.run(self.update_local_ops)
```

```
                episode_buffer = []
                episode_values = []
                episode_frames = []
                episode_reward = 0
                episode_step_count = 0
                d = False

                self.env.new_episode()
                s = self.env.get_state().screen_buffer
                episode_frames.append(s)
                s = process_frame(s)
                rnn_state = self.local_AC.state_init
                self.batch_rnn_state = rnn_state
                while self.env.is_episode_finished() == False:
                    # Take an action using probabilities from policy
                    ↪  network output.
                    a_dist, v, rnn_state = sess.run(
                        [
                            self.local_AC.policy,
                            self.local_AC.value,
                            self.local_AC.state_out,
                        ],
                        feed_dict={
                            self.local_AC.inputs: [s],
                            self.local_AC.state_in[0]: rnn_state[0],
                            self.local_AC.state_in[1]: rnn_state[1],
                        },
                    )
                    a = np.random.choice(a_dist[0], p=a_dist[0])
                    a = np.argmax(a_dist == a)

                    r = self.env.make_action(self.actions[a]) / 100.0
                    d = self.env.is_episode_finished()
                    if d == False:
                        s1 = self.env.get_state().screen_buffer
                        episode_frames.append(s1)
                        s1 = process_frame(s1)
                    else:
                        s1 = s

                    episode_buffer.append([s, a, r, s1, d, v[0, 0]])
                    episode_values.append(v[0, 0])

                    episode_reward += r
                    s = s1
                    total_steps += 1
                    episode_step_count += 1

                    # If the episode hasn't ended, but the experience
                    ↪  buffer is full, then we
                    # make an update step using that experience rollout.
                    if (
                        len(episode_buffer) == 30
                        and d != True
                        and episode_step_count != max_episode_length - 1
                    ):
                        # Since we don't know what the true final return
                        ↪  is, we "bootstrap" from our current
                        # value estimation.
                        v1 = sess.run(
```

```
                    self.local_AC.value,
                    feed_dict={
                        self.local_AC.inputs: [s],
                        self.local_AC.state_in[0]: rnn_state[0],
                        self.local_AC.state_in[1]: rnn_state[1],
                    },
                )[0, 0]
                v_l, p_l, e_l, g_n, v_n = self.train(
                    episode_buffer, sess, gamma, v1
                )
                episode_buffer = []
                sess.run(self.update_local_ops)
            if d == True:
                break

        self.episode_rewards.append(episode_reward)
        self.episode_lengths.append(episode_step_count)
        self.episode_mean_values.append(np.mean(episode_values))

        # Update the network using the episode buffer at the end
        ↪  of the episode.
        if len(episode_buffer) != 0:
            v_l, p_l, e_l, g_n, v_n = self.train(
                episode_buffer, sess, gamma, 0.0
            )

        # Periodically save gifs of episodes, model parameters,
        ↪  and summary statistics.
        if episode_count % 5 == 0 and episode_count != 0:
            if self.name == "worker_0" and episode_count % 25 ==
            ↪  0:
                time_per_step = 0.05
                images = np.array(episode_frames)
                make_gif(
                    images,
                    "./frames/image" + str(episode_count) +
                    ↪  ".gif",
                    duration=len(images) * time_per_step,
                    true_image=True,
                    salience=False,
                )
            if episode_count % 250 == 0 and self.name ==
            ↪  "worker_0":
                saver.save(
                    sess,
                    self.model_path + "/model-" +
                    ↪  str(episode_count) + ".cptk",
                )
                print("Saved Model")

            mean_reward = np.mean(self.episode_rewards[-5:])
            mean_length = np.mean(self.episode_lengths[-5:])
            mean_value = np.mean(self.episode_mean_values[-5:])
            summary = tf.Summary()
            summary.value.add(
                tag="Perf/Reward",
                ↪  simple_value=float(mean_reward)
            )
            summary.value.add(
```

```
                        tag="Perf/Length",
                        ↪  simple_value=float(mean_length)
                    )
                    summary.value.add(tag="Perf/Value",
                    ↪  simple_value=float(mean_value))
                    summary.value.add(tag="Losses/Value Loss",
                    ↪  simple_value=float(v_l))
                    summary.value.add(tag="Losses/Policy Loss",
                    ↪  simple_value=float(p_l))
                    summary.value.add(tag="Losses/Entropy",
                    ↪  simple_value=float(e_l))
                    summary.value.add(tag="Losses/Grad Norm",
                    ↪  simple_value=float(g_n))
                    summary.value.add(tag="Losses/Var Norm",
                    ↪  simple_value=float(v_n))
                    self.summary_writer.add_summary(summary,
                    ↪  episode_count)

                    self.summary_writer.flush()
                if self.name == "worker_0":
                    sess.run(self.increment)
                episode_count += 1
```

Now let's finish with the code that calls the corresponding functions and starts the learning:

```
max_episode_length = 300
gamma = 0.99  # discount rate for advantage estimation and reward
↪  discounting
s_size = 7056  # Observations are greyscale frames of 84 * 84 * 1
a_size = 3  # Agent can move Left, Right, or Fire
load_model = False
model_path = "./model"

tf.reset_default_graph()

if not os.path.exists(model_path):
    os.makedirs(model_path)

# Create a directory to save episode playback gifs to
if not os.path.exists("./frames"):
    os.makedirs("./frames")

with tf.device("/cpu:0"):
    global_episodes = tf.Variable(
        0, dtype=tf.int32, name="global_episodes", trainable=False
    )
    trainer = tf.train.AdamOptimizer(learning_rate=1e-4)
    master_network = AC_Network(
        s_size, a_size, "global", None
    )  # Generate global network
    num_workers = (
        multiprocessing.cpu_count()
    )  # Set workers to number of available CPU threads
    workers = []
    # Create worker classes
    for i in range(num_workers):
```

```
        workers.append(
            Worker(DoomGame(), i, s_size, a_size, trainer, model_path,
            ↪  global_episodes)
        )
    saver = tf.train.Saver(max_to_keep=5)

with tf.Session() as sess:
    coord = tf.train.Coordinator()
    if load_model == True:
        print("Loading Model...")
        ckpt = tf.train.get_checkpoint_state(model_path)
        saver.restore(sess, ckpt.model_checkpoint_path)
    else:
        sess.run(tf.global_variables_initializer())

    # This is where the asynchronous magic happens.
    # Start the "work" process for each worker in a separate threat.
    worker_threads = []
    for worker in workers:
        worker_work = lambda: worker.work(max_episode_length, gamma,
        ↪  sess, coord, saver)
        t = threading.Thread(target=(worker_work))
        t.start()
        sleep(0.5)
        worker_threads.append(t)
    coord.join(worker_threads)
```