

# Reinforcement Learning

A deep insight in Q-Learning  
and Policy Gradient  
Algorithms

*while playing CarRacing-v0...*



**NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS  
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS**

**Title of Project:**

«A deep insight in Q-Learning and Policy Gradient Algorithms while playing  
CarRacing-v0»

**Course:** Reinforcement Machine Learning and theory games

***Full names:***

Giannoutsos Andreas (1115201700021)

Apostolopoulou Alexandra (1115201700005)

Briakos Spyros (11152017000101)

*Athens, 2021*

# Contents:

<b>Abstract.....</b>	<b>4</b>
<b>Deep Q-Network.....</b>	<b>5</b>
1. Introduction to Reinforcement Learning.....	5
2. Deep Q-Network (DQN).....	5
3. Experiment Setup.....	9
4. Experiments.....	12
<b>Actor-Critic.....</b>	<b>15</b>
1. Introduction to policy gradient.....	15
2. The math behind policy gradient.....	15
2.1 On-policy exploration problem.....	16
2.2 Policy Gradient Theorem.....	17
2.3 Monte-Carlo policy gradient.....	17
3. Actor-Critic model.....	19
3.1 Advantage Actor-Critic.....	21
4. Experiments and results.....	22
<b>Proximal Policy Optimization.....</b>	<b>24</b>
1. Introduction.....	24
2. Background.....	24
2.1 Policy Gradient Methods.....	25
2.2 Trust Region Methods.....	26
3. Clipped Surrogate Objective.....	26
4. Algorithm.....	28
5. Experiments with PPO.....	29
<b>Conclusion.....</b>	<b>33</b>
<b>Sources.....</b>	<b>34</b>

## ***Abstract***

Here we represent you our work for the Project in different *Q-Learning Methods* and *Policy Gradient Methods*. More specifically, we select the game of *CarRacing-v0* of *OpenAI*, in which we implemented three algorithms, with purpose to make it stay on the road, get the maximum rewards and in the end to win. The three models that we tried are the *Deep Q-Network*, the *Proximal Policy Optimization* and the *Actor-Critic*. After we did an extensive research and report about these three algorithms and we understood the way they act, in depth, we experimented and train them for many days, in order to see their behavior. The results didn't satisfy us to the maximum degree, i.e. some behaviours weren't as expected from theory, but we attributed it to the fact that we hadn't as much computing power that required (e.g. for parallel environments or RAM space for buffer size), but also we hadn't the required time, as our *Google Colab* was disconnected from much use all the time. Nevertheless, the results were pretty decent, in general. Our team did its best to overcome all the technical problems that existed and with a lot of patience and passion reached a very satisfactory degree of results.

# Deep Q-Network

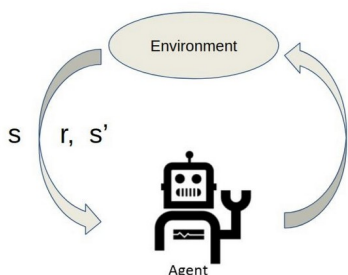
## 1. Introduction to Reinforcement Learning

### So what is Reinforcement Learning?

Reinforcement Learning is when an algorithm gives feedback about specific actions, where there is a desired end state, yet typically there is no clear best path. In Reinforcement Learning, a model is asked to make **a series of decisions**. Each decision that is acted upon will **provide a reward**, which will either encourage the model to keep performing said actions or the opposite.

### Quick Representation of Reinforcement Learning Problem

In the complete RL problem, the state changes every time we take an action. This is how we can represent it. The agent gets the state in which the environment is in, represented with the letter  $s$ . The agent then chooses which action to take, represented with the letter  $a$ . After following that action, the environment provides a reward, represented with the letter  $r$ , and transitions to a new state, represented as  $s'$ . This is the main RL cycle and can be observed in above figure.



**As a result of this cycle**, the action the agent chooses to take at each time-step must not maximize the immediate reward at that step. It must choose the action that will lead to the highest possible sum of future rewards (in other words, the return) until the end of the episode. **This cycle results in a sequence of states, actions and rewards, from the beginning of the episode, until the end:**  $s_1, a_1, r_1; s_2, a_2, r_2; \dots; s_T, a_T, r_T$ . Where  $T$  indicates the last step in the episode.

## 2. Deep Q-Network (DQN)

### Introduction to DQN

Our environment is deterministic, so all equations presented here are also formulated deterministically for the sake of simplicity. In the Reinforcement Learning literature, they would also contain expectations over stochastic transitions in the environment.

Our **aim** will be to train a policy that tries to maximize the discounted, cumulative reward  $R_t = \sum_{t=0}^{\infty} \gamma^t (r_t - \gamma R_t)$ , where  $R_t$  is also known as the **return**. The **discount,  $\gamma$** , should be a constant between 0 and 1 that ensures the sum converges. It makes rewards from the uncertain far future less important for our agent than the ones in the near future that it can be fairly confident about.

The main idea behind Q-learning is that if we had a function  $Q^* : \text{State} \times \text{Action} \rightarrow \mathbb{R}$ , that could tell us what our return would be, if we were to take an action in a given state, then we could easily construct a policy that maximizes our rewards:  $\pi^*(s) = \arg\max_a Q^*(s, a)$

However, we don't know everything about the world, so we don't have access to  $Q^*$ . But, since neural networks are universal function approximators, we can simply create one and train it to resemble  $Q^*$ . For our training update rule, we'll use a fact that every Q function for some policy obeys the Bellman equation:  $Q(\pi(s, a)) = r + \gamma Q(\pi(s'), \pi(s'))$

### ***Bellman's Equation***

The intuition behind this equation is the following. The Q-value for state  $s$  and action  $a$  ( $Q(s, a)$ ) must be equal to the immediate reward  $r$  obtained as a result of that action, plus the Q-value of the best possible next action  $a'$  taken from the next state  $s'$ , multiplied by a **discount factor**  $\gamma$ , which is a value with range  $\gamma \in (0, 1]$ . This value  $\gamma$  is used to decide how much we want to weight the short and long-term rewards, and it is a hyper-parameter we need to decide.

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

When there are billions of possible unique states and hundreds of available actions for each of them, the table becomes too big, and tabular methods become impractical. The **Deep Q-Networks (DQN)** algorithm was invented by Mnih in order to solve this. This algorithm combines the Q-Learning algorithm with **deep neural networks (DNNs)**. As it is well known in the field of AI, DNNs are great non-linear function approximators. Thus, DNNs are used to approximate the Q-function, replacing the need for a table to store the Q-values. In reality, this algorithm uses two DNNs to stabilize the learning process.

The first one is called the **main neural network**, represented by the weight vector  $\theta$ , and it is used to estimate the Q-values for the current state  $s$  and action  $a$ :  $Q(s, a; \theta)$ .

The second one is the **target neural network**, parametrized by the weight vector  $\theta'$ , and it will have the exact same architecture as the main network, but it will be used to estimate the Q-values of the next state  $s'$  and action  $a'$ .

All the learning takes place in the main network. The target network is frozen (its parameters are left unchanged) for a few iterations (usually around 10000) and then **the weights of the main network are copied into the target network**, thus transferring the learned knowledge from one to the other. This makes the estimations produced by the target network more stable and accurate after the copying has occurred.

## Bellman's Equation for DQN

Bellman's equation has this shape now, where the Q functions are parametrized by the network weights  $\theta$  and  $\theta'$ .

$$Q(s, a; \theta) = r + \gamma \max_{a'} Q(s', a'; \theta')$$

In order to train a neural network, we need a **loss function**, which is defined as the squared difference between the two sides of the bellman equation, in the case of the DQN algorithm.

$$L(\theta) = \mathbb{E}[(r + \gamma \max_{a'} Q(s', a'; \theta') - Q(s, a; \theta))^2]$$

This is the function we will minimize using **gradient descent**, which can be calculated automatically using a **Deep Learning library such as TensorFlow or PyTorch**.

**Learning directly from consecutive samples is inefficient**, due to the strong correlations between the samples; randomizing the samples breaks these correlations and therefore reduces the variance of the updates. When learning on-policy the current parameters determine the next data sample that the parameters are trained on.

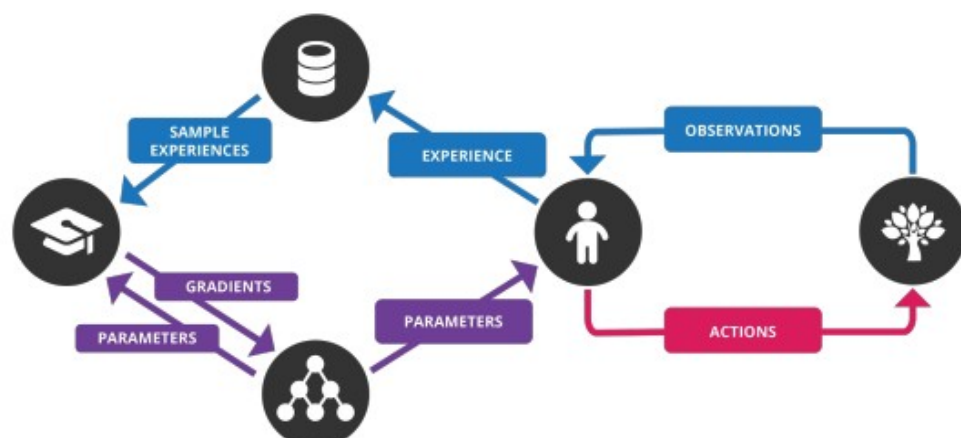
For example, if the maximizing action is to move left then the training samples will be dominated by samples from the left-hand side; if the maximizing action then switches to the right then the training distribution will also switch. It is easy to see how unwanted feedback loops may arise and the parameters could **get stuck in a poor local minimum**, or even **diverge catastrophically**. We can apply a technique, called Replay Memory or Experience Replay, which aims to avoid oscillations or divergence in the parameters.

- **What is Replay Memory?**

We'll be using experience replay memory for training our DQN. Replay memory **stores the transitions** that the agent observes, allowing us to reuse this data later. By sampling from it randomly, the transitions that build up a batch are decorrelated and simultaneously we prevent the network from only learning about what it is immediately doing in the environment, and allow it to learn from a more varied array of past experiences. It has been shown that this greatly **stabilizes** and **improves** the DQN training procedure.

Transition: a named tuple representing a single transition in our environment. It essentially maps (state, action) pairs to their (next\_state, reward) result. Each of these experiences are stored as a tuple of `<state, action, reward, next state>`.

Replay Memory: a cyclic buffer of bounded size that holds the transitions observed recently. It also implements a `.sample()` method for selecting a random batch of transitions for training.





In practice, our algorithm only stores the last  $N$  experience tuples in the replay memory, and samples uniformly at random from  $D$  when performing updates. This approach is in some respects limited since the memory buffer **does not differentiate important transitions** and always **overwrites** with **recent transitions** due to the finite memory size  $N$ . Similarly, the uniform sampling gives equal importance to all transitions in the replay memory.

~A more sophisticated sampling strategy might emphasize transitions from which we can learn the most, similar to prioritized sweeping.

## DQN Algorithm

---

### Algorithm 1 Deep Q-learning with Experience Replay

---

```

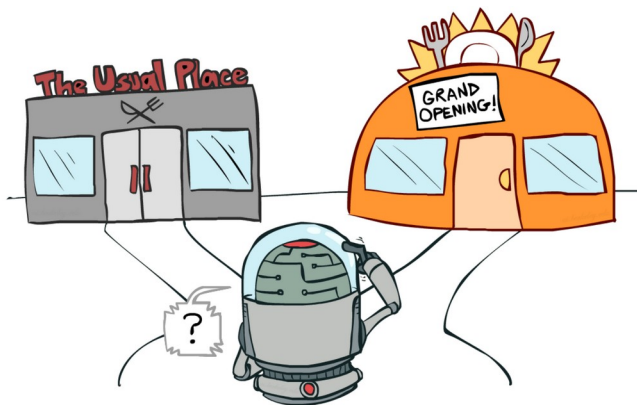
Initialize replay memory  $\mathcal{D}$  to capacity  $N$ 
Initialize action-value function  $Q$  with random weights
for episode = 1,  $M$  do
  Initialise sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$ 
  for  $t = 1, T$  do
    With probability  $\epsilon$  select a random action  $a_t$ 
    otherwise select  $a_t = \max_a Q^*(\phi(s_t), a; \theta)$ 
    Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
    Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
    Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $\mathcal{D}$ 
    Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $\mathcal{D}$ 
    Set  $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$ 
    Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  according to equation 3
  end for
end for

```

---

## Exploration vs Exploitation

How much of an agent's time should be spent exploiting its existing known-good policy, and how much time should be focused on exploring new, possibility better, actions? We often encounter similar situations in real life too. For example, we face on which restaurant to go to on Saturday night. We all have a set of restaurants that we prefer, based on our policy/strategy book  $Q(s, a)$ . If we stick to our normal preference, there is a strong probability that we'll pick a good restaurant.

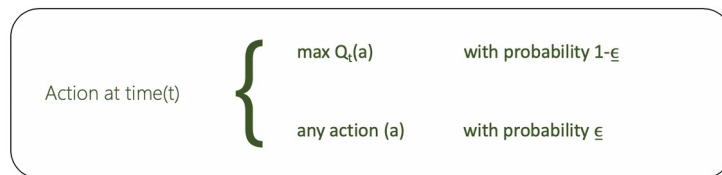


However, sometimes, we occasionally like to try new restaurants to see if they are better. The RL agents face the same problem. In order to **maximize future reward**, they need to **balance** the amount of time that they follow their **current policy** (this is called being “greedy”), and the time they spend **exploring** new possibilities that might be better. A popular approach is called  $\epsilon$  greedy approach, so let's explore it!



## $\epsilon$ -Greedy Approach

**Explanation:** A simple **combination** of the **greedy and random** approaches yields one of the most used exploration strategies:  $\epsilon$ -greedy. In this approach the agent chooses what it believes to be the optimal action most of the time, but occasionally acts randomly. This way the agent takes actions which it may not estimate to be ideal, but may provide new information to the agent. The  $\epsilon$  in  $\epsilon$ -



greedy is an adjustable parameter which determines the probability of taking a random, rather than principled, action. Due to its simplicity and surprising power, this approach has become the defacto technique

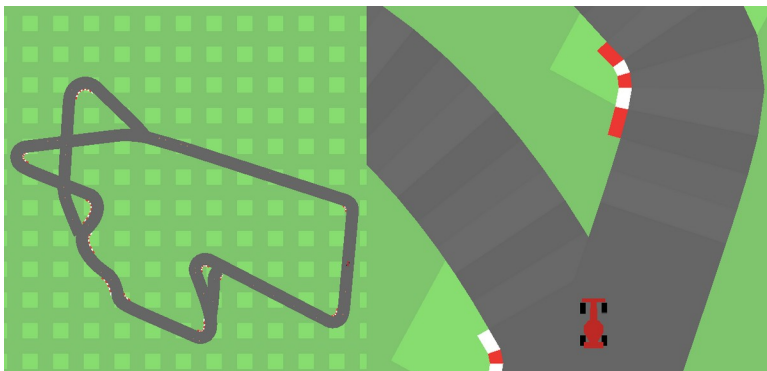
for most recent reinforcement learning algorithms, including DQN and its variants.

**Adjusting during training:** At the start of the training process the  $\epsilon$  value is often initialized to a large probability, to **encourage exploration** in the face of knowing little about the environment. The value is **then annealed down** to a small constant (often 0.1), as the agent is assumed to learn most of what it needs about the environment.

## 3. Experiment Setup

### CarRacing-v0 Environment

The classic CarRacing-v0 environment is both simple and straightforward. Without any external modifications, the state consists of **96x96 pixels**, starting off with a **classical RGB** environment as well. The reward is equal to -0.1 for each frame and  $+1000/N$  for every track tile visited, where  $N$  is represented by the total number of tiles throughout the entirety of the track. To be considered a



**successful run**, the agent must achieve a **reward of 900** consistently, thus meaning that the maximum time the agent has to be on the track is 1000 frames. Furthermore, there is a barrier outside of the track, which results in a -100 penalty and an immediate finish of the episode if crossed over. Outside the track consists of grass, which does

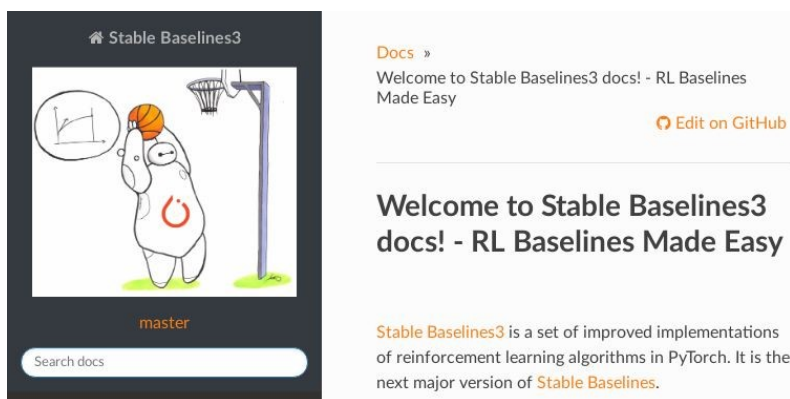
not give rewards but due to the friction of the environment, results in a struggle for the vehicle to move back onto the track. Overall, this environment is a classic 2D environment which is significantly simpler than that of 3D environments, making OpenAI's CarRacing-v0 much simpler.

## Google Colab GPU (Training Time)

Training time is an essential part of deep learning, due to the fact that learning **takes lots of time to be able to learn** whatever process the model is being applied upon. For many situations, **GPUs are a must** as they exponentially increase the speed of training, thus training the model quicker, allocating the additional training time for perfecting the model, and tweaking the extra changes.



## Stable Baselines3 Library



We are going to utilize brand new version of library Stable Baselines3, which contains improved implementations of Reinforcement Learning algorithms, in native Pytorch.

## On-Policy vs Off Policy Reinforcement Learning

On-Policy Reinforcement Learning	Off-Policy Reinforcement Learning
<u>Policy <math>\pi</math> is updated with data collected by <math>\pi</math> itself.</u> We optimize the current policy $\pi$ and use it to determine what spaces and actions to explore and sample next. That means we will try to improve the same policy that the agent is <b>already using</b> for action selection.	Allows the use of older samples (collected using the older policies) in the calculation. To update the policy, experiences are sampled from a buffer which comprises experiences/interactions that are collected <b>from its own predecessor policies</b> . This improves sample efficiency since we don't need to recollect samples whenever a policy is changed.
Behaviour policy == Policy used for action selection	Behaviour policy $\neq$ Policy used for action selection
Examples: Policy Iteration, Sarsa, PPO, TRPO	Examples: Q- learning, DQN, DDQN, DDPG

So as we can easily understand that DQN belongs to family of Off-Policy algorithms. Indeed, we can observe that class [DQN](#) inherits from class [OffPolicyAlgorithm](#)!

#### Source code for stable\_baselines3.dqn.dqn

```
from typing import Any, Dict, List, Optional, Tuple, Type, Union

import gym
import numpy as np
import torch as th
from torch.nn import functional as F

from stable_baselines3.common import logger
from stable_baselines3.common.off_policy_algorithm import OffPolicyAlgorithm
from stable_baselines3.common.type_aliases import GymEnv, MaybeCallback, Schedule
from stable_baselines3.common.utils import get_linear_fn, is_vectorized_observation, polyak_update
from stable_baselines3.dqn.policies import DQNPoly

class DQN(OffPolicyAlgorithm):
    """
    Deep Q-Network (DQN)
    """
```

#### Source code for stable\_baselines3.common.off\_policy\_algorithm

```
import io
import pathlib
import time
import warnings
from typing import Any, Dict, Optional, Tuple, Type, Union

import gym
import numpy as np
import torch as th

from stable_baselines3.common import logger
from stable_baselines3.common.base_class import BaseAlgorithm
from stable_baselines3.common.buffers import ReplayBuffer
from stable_baselines3.common.callbacks import BaseCallback
from stable_baselines3.common.noise import ActionNoise
from stable_baselines3.common.policies import BasePolicy
from stable_baselines3.common.save_util import load_from_pk, save_to_pk
from stable_baselines3.common.type_aliases import GymEnv, MaybeCallback, RolloutReturn, Schedule
from stable_baselines3.common.utils import safe_mean
from stable_baselines3.common.vec_env import VecEnv

class OffPolicyAlgorithm(BaseAlgorithm):
    """
    The base for Off-Policy algorithms (ex: SAC/TDS)
    """
```



#### Can I use?

- Recurrent policies: ✗
- Multi processing: ✗
- Gym spaces:

Space	Action	Observation
Discrete	✓	✓
Box	✗	✓
MultiDiscrete	✗	✓
MultiBinary	✗	✓

So...stable\_baselines3 library has a set of restrictions for DQN class. As we can see on leftside, **actions must be discrete!** The problem, here is that car\_racing, from gym.ai environment, agent has continuous actions. How we manage to solve it?

We simply obtained code from [official code for car racing.py](#) and we convert a small part of code, in order to have five discrete actions.

```
class CarRacingDiscrete(gym.Env, EzPickle):
```

Declaring, a brand new class called CarRacingDiscrete, with which, from now on, our agent can pick an action from five choices {"left","right","gas", "brake","do nothing"}. Each action represented by a unique vector, with size=3.

```
# discrete car racer
# self.action_space = spaces.Box( np.array([-1,0,0]), np.array([+1,+1,+1]), dtype=np.float32) # steer, gas, brake
self.action_space = spaces.Discrete(5)
self.actions = [np.array([-1,0,0], dtype=np.float32),
                np.array([1,0,0], dtype=np.float32),
                np.array([0,1,0], dtype=np.float32),
                np.array([0,0,0.8], dtype=np.float32),
                np.array([0,0,0], dtype=np.float32)] # left right, gas, brake, nothing
```

You are able to check our conversion from continuous to discrete here: [discrete car racing.py](#)

## 4. Experiments

So...with this DQN model we managed to try three big separate experiments! Here are the hyperparameters, which we maintain in both of them and we choosed from this fundamental paper [Playing Atari with Deep Reinforcement Learning](#)!

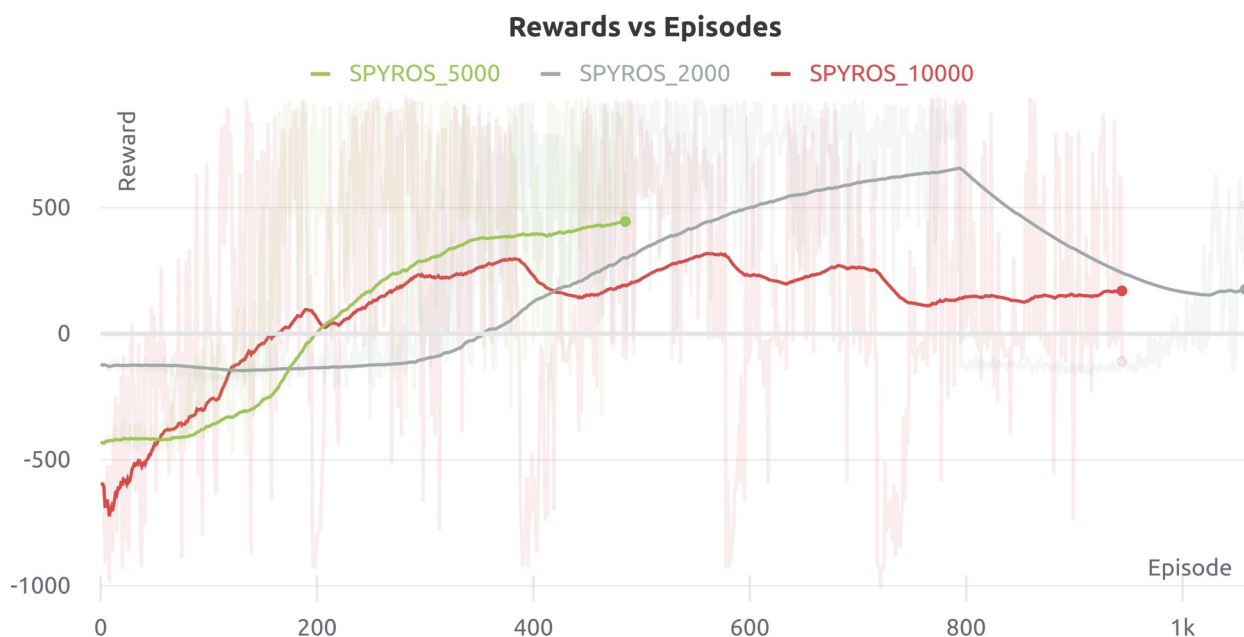
policy: **CNNPolicy**  
learning\_rate: **1e-4**  
learning\_starts: **5.000**  
batch\_size: **32**  
gamma: **0.99**  
exploration\_fraction: **0.1**  
exploration\_initial\_eps: **1.0**  
exploration\_final\_eps = **0.05**

- Stable Baselines3 Library provides two choices about policy and for our problem more suitable was CNNPolicy, cause we were using images (frames of our car into grid) as input to our Neural Net.
- Hyperparameter learning\_starts was set to 5.000, which means that when buffer obtain 5.000 frames-states (timesteps), then our gradient descent of NN will start to work.
- Last three hyperparameters indicate, respectively that epsilon will begin initially with 1.0 value and gradually decrease with 0.1=10% rate-rhythm until it finally get at final value of 0.05.

### *Differences of experiments*

First Experiment	Second Experiment	Second Experiment
Buffer_size: 100.000	Buffer_size: 150.000	Buffer_size: 150.000
Maximum_steps_per_episode: 10.000 (3'20")	Maximum_steps_per_episode: 2.000 (40")	Maximum_steps_per_episode: 5.000 (1'40")
Log_interval: 20	Log_interval: 50	Log_interval: 50
Episodes: 1.000	Episodes: 1.000	Episodes: 500

## Charts of Experiments



DQN models, with different number of steps, which we can observe right above, seem to learn through process of Q-Learning, during episodes, but we can observe that in some points of chart mean reward of each experiment decreases rapidly! This phenomenon is called catastrophic forgetting and we are going to explain it! However, we managed to have a pretty descent performance at some checkpoints of our three big experiments. *Note, that you can check videos and seperate charts of our experiment's model's evaluation inside folder DQN\_Results!*

We aimed to gather some significant information about DQN's performance, from this paper [Implementing Deep Q-Network \(Brown University\)](#), which are going to verify our experiment's results.

A common belief for new users of DQN is that performance should fairly **stably** improve as more training time is given. Indeed, average Q-learning learning curves in tabular settings are typically fairly stable improvements and supervised deep-learning problems also tend have fairly steady average improvement as more data becomes available.

However, it is not uncommon in DQN to have “**catastrophic forgetting**” in which the agent’s performance can drastically drop after a period of learning. For example, the DQN agent may reach a point of averaging a high score over 400, and then, after another large batch of learning, it might be averaging a score of only around 200.

- **Why catastrophic forgetting occurs?**

One of the reasons this forgetting occurs is the inherent **instability of approximating the Q-function** over a large state-space using these Bellman updates.

- **How did Mnih manage to overcome this instability?**

One of the main contributions of Mnih was fighting this instability using **experience replay** and **stale network parameters**. Additionally, Mnih found that **clipping the gradient** of the error term to be between  $-1.0$  and  $1.0$  further improved the stability of the algorithm by not allowing any single mini-batch update to change the parameters drastically. These additions, and others, to the DQN algorithm *improve* its stability *significantly*, but the network **still** experiences catastrophic forgetting.

Another reason this catastrophic forgetting occurs is that the algorithm is learning a proxy, the Q-values, for a policy instead of approximating the policy directly. A side effect of this method of policy generation is a learning update could increase the accuracy of a Q-function approximator, while decreasing the performance of the resulting policy.

A quick example so as to understand this!

Say the true Q-value for some state,  $s$ , and actions,  $a1$  and  $a2$ , are

$$Q^*(s, a1) = 2 \text{ and } Q^*(s, a2) = 3$$

So the optimal policy at state  $s$  would be to choose action  $a2$ .

Now say the Q-function approximator for these values using current parameters,  $\theta$ , estimates

$$\hat{Q}(s, a1; \theta) = 0 \text{ and } \hat{Q}(s, a2; \theta) = 1$$

so the policy chosen by this approximator will also be  $a2$ . But, after some learning updates we arrive at a set of parameters  $\theta$ , where

$$\hat{Q}(s, a1; \theta) = 2 \text{ and } \hat{Q}(s, a2; \theta) = 1$$

These learning updates **clearly decreased** the error of the Q-function approximator, but now the agent **will not** choose the optimal action at state  $s$ . Furthermore Q-values for different actions of the same state can be very similar if any of these actions does not have a significant effect on near-term reward.

### **Conclusion**

The consequence of trying to learn an approximator for this type of function is that:  
*"Very small errors in the Q-values can result in very different policies, making it difficult to learn long-term policies".*

# Actor-Critic

## 1. *Introduction to policy gradient*

The Q-Learning algorithm can have several drawbacks. Deterministic is the policy that Q-Learning follows. This implies that through Q-Learning, stochastic policies can not be learned, which can be useful in some environments. It also means that we need to establish our own exploration plan because, following the policy, no exploration can be carried out. With the  $\epsilon$ -greedy discovery, which can be very inefficient, we normally do this.

Q-Learning algorithm also picks the action that has the maximum value outcome. This is done by picking the maximum variable out of a discrete set of actions. In Q-Learning, there is no clear way of treating continuous acts. Under the policy gradient, it is reasonably straightforward to manage continuous actions.

Finally, in the policy gradient, we follow gradients with respect to the policy itself, which means that we are continually improving the policy. By comparison, we are improving our estimates of the values of various behaviors in Q-Learning, which only indirectly improves the policy. It would be more beneficial to improve the strategy directly.

### Example of stochastic policy



An example about a stochastic policy. In a two-player game of rock-paper-scissors, a deterministic policy could be highly exploitable. By mapping certain states to actions, the opponent could easily exploit these patterns. Therefore a uniform random policy is optimal.

## 2. *The math behind policy gradient*

Policy-based reinforcement learning is an optimization problem. We need to find the parameters of an approximation function that maximizes a function of  $J(\theta)$ .

$$\Delta\theta = \alpha \nabla_{\theta} J(\theta)$$

For our policy function, we may consider the following. The letter  $\pi$  will symbolize a policy. Let's call  $\pi_{\theta}(a|s)$  the probability of taking action given a state  $s$ .  $\theta$  represents the parameters of our policy which in deep reinforcement learning are the parameters of our neural network.



We need clearly every time to pick the best and optimal action given a state  $s$ . That indicates that the probability of the action  $\pi_\theta(a^*|s)$  must be close to 1. To achieve that we update our weights using gradient ascent.

$$\theta_{t+1} = \theta_t + \alpha \nabla \pi_{\theta_t}(a^*|s)$$

However, in a model-free prediction environment, we are not aware of the best action. We need to propagate to our policy estimator information about the reward of an action. In the case of a one-step MDP, we can simply multiply the reward of an action at the end of the derivative. If an action contributes to better rewards, the gradient ascent multiplied by the higher rewards will take care of our parameters.

$$\theta_{t+1} = \theta_t + \alpha \hat{Q}(s, a) \nabla \pi_{\theta_t}(a|s)$$

The Q function gives us the reward of action in a certain state. From the previous chapter, we know that in a complex environment Q function can be replaced by a neural network and approximate the value function. For now, let's just assume that this Q function is a given without needing further optimization.

## 2.1 On-policy exploration problem

During on-policy learning, the algorithm learns to optimize its parameters on its own without needing to feed it data of how the policy should behave in certain scenarios. This is beneficial because we can accumulate more rewards even during training. However, training on policy might give us a biased exploration set of data. The algorithm picks its next action based on the assumption that it has made from all the previous actions. This behavior can turn into a more positive situation if all the previous actions were truly the most optimal. Nevertheless, if the previous actions were not optimal, which in many cases seems more likely, the algorithm could be stuck in a local optimal by considering that it made the best decisions and based on that false perception of the world will pick the next actions.

This implies that we need to compensate for the fact that more likely acts will be done more often. We need to change our value function and divide the derivative of our policy by the probability value of our policy. This means that when a more likely action will be taken, the updated parameters will be reduced times the likelihood of that action.

$$\theta_{t+1} = \theta_t + \alpha \frac{\hat{Q}(s, a) \nabla \pi_{\theta_t}(a|s)}{\pi_{\theta_t}(a|s)}$$

## 2.2 Policy Gradient Theorem

All of that implies a one-step MDP. In order to scale to a multi-step MDP we just need to replace the immediate reward with the value function of the long-term reward. That actually turns out to be the true gradient of the policy and the policy gradient theorem tells us that if you start with some policy the average value of the policy gradient is basically given by this type.

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(s, a) Q^{\pi_{\theta}}(s, a)]$$

The expectation over the policy gradient is multiplied by the action-value function  $Q$ . More specifically this expression tells us that in order to adjust the policy, to get a particular action multiplied by how good that particular action was, we need to adjust the policy in the direction that does more of the good things and less of the bad things.

## 2.3 Monte-Carlo policy gradient

One of the earliest policy gradient algorithms is the Reinforce algorithm, which executes a stochastic gradient descent over the simplified policy gradient expectation. The algorithm uses episode samples to update the policy parameter  $\theta$ .

The process is pretty straightforward:

1. Initialize the policy parameter  $\theta$  at random.
2. Generate one trajectory on policy  $\pi_{\theta}$ :  $S_1, A_1, R_2, S_2, A_2, \dots, S_T$
3. For  $t=1, 2, \dots, T$ :
4. Update policy parameters:  $\theta \leftarrow \theta + \alpha \nabla_{\theta} \log \pi_{\theta}(s_t, a_t) v_t$

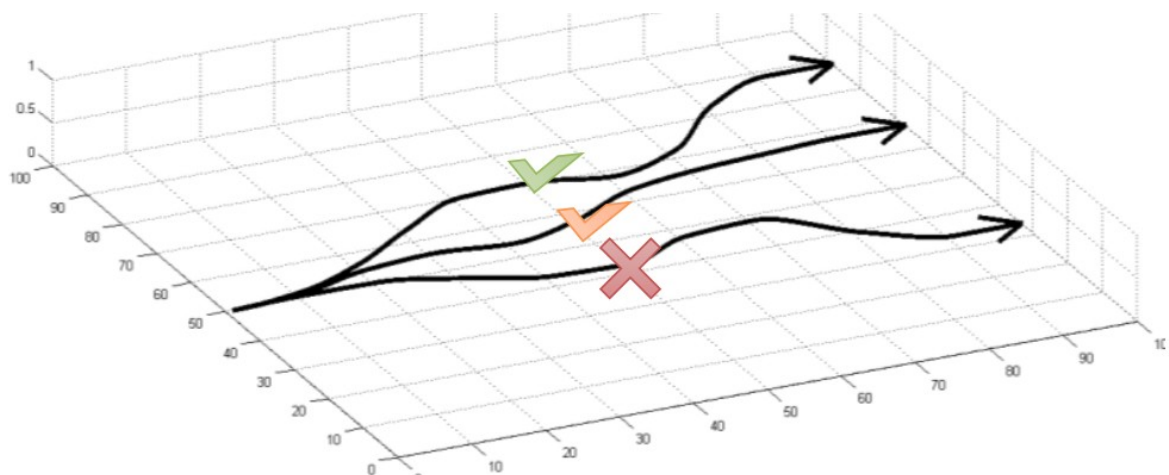
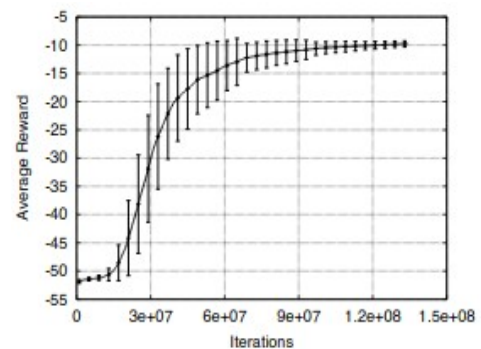
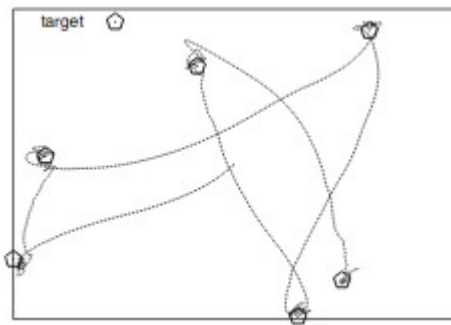
So that's the reinforce gradient policy optimization. Unfortunately, we're not quite done yet. The naive way is to run the agent on a batch of episodes or on every episode if it's stochastic, get a set of trajectories, and update  $\theta$  using the empirical expectation, but this will be too slow and unreliable due to high variance on the gradient estimates.

### Why there is so much variance

When we talk about high variance in the policy gradient method, we're specifically talking about the fact that the variance of the gradients is high, namely that  $\text{Var}(\nabla_{\theta} J(\theta))$  is big. When we want to compute the gradient step we typically sample data from multiple trajectories based on the current policy  $\pi$ . Then we average the values and we update our policy parameters by them. This means that the gradient is dependent on the randomly selected samples from the policy that we want

to update. As a result, we are going to have high variance since the update parameters depend on the data they propose to sample.

Furthermore, after one batch of training data, we may encounter a wide range of results. Some of them might have better, equal, or worse performance. To find an optimal solution due to this high variance we need to run for millions of iterations, our update function. The high variance of these gradient optimizers is why there have been so many variance reduction techniques. Phenomenally good actions might not serve good results. This is what provokes the variance problem.



## Advantage Baseline

A method to reduce the variance of the policy gradient method is to use a baseline. A common baseline is to subtract state-value from action-value, and if applied, we would use the advantage  $A$ .

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

Does this suggest that, in addition to the policy network, we'll need to construct two neural networks for the  $Q$  and  $V$  values? That would be incredibly inefficient. Instead, from the Bellman optimality equation, we can use the relationship between the  $Q$  and  $V$ .

$$Q(s_t, a_t) = \mathbb{E}[r_{t+1} + \gamma V(s_{t+1})]$$

So by using the Bellman optimality equation we can rewrite the advantage as:

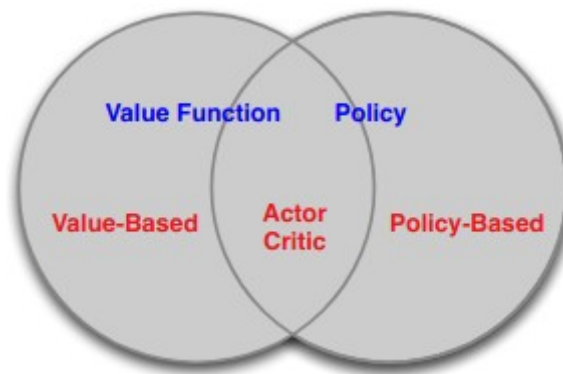
$$A(s_t, a_t) = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)$$

From that we only need to use one neural network and the final policy gradient function will be written as:

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(s, a) A^{\pi_{\theta}}(s, a)]$$

This doesn't actually change the expected value of the gradient at all but will reduce the variance if you set the right baseline. So, we deduct the  $Q$  value term with the  $V$  value using the  $V$  function as the baseline function. In a nutshell, this means how much better is to take an action compared to the average general average action at the given state.

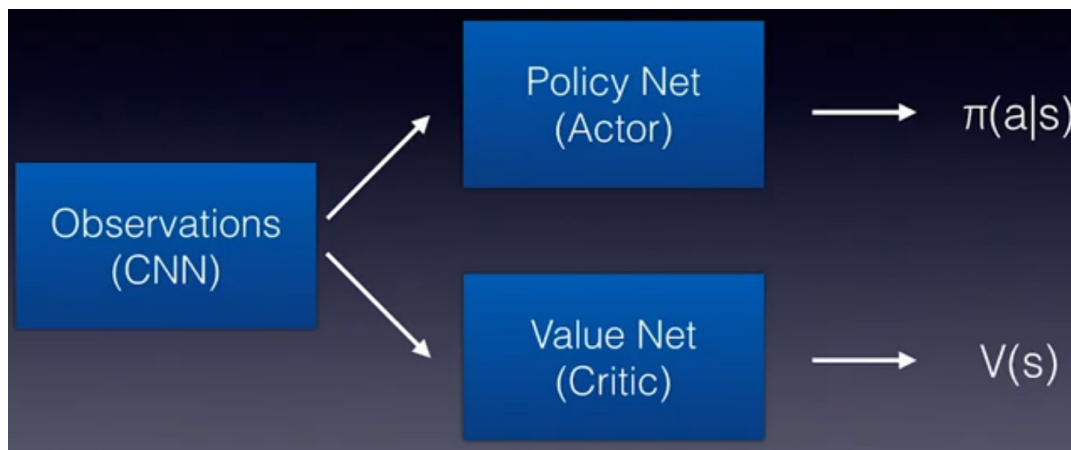
## 3. Actor-Critic model



The policy model and the value function are two principal components of the policy gradient. In addition to the policy, it makes a lot of sense to learn the value function, as learning the value function will improve the policy update, such as reducing gradient variation in vanilla policy gradients, and that's exactly what the Actor-Critic technique does.

Actor-critic methods consist of two models, which may optionally share parameters:

- Critic updates the value function parameters  $w$  and depending on the algorithm it could be action-value  $Q_w(a|s)$  or state-value  $V_w(s)$
- Actor updates the policy parameters  $\theta$  for  $\pi_\theta(a|s)$ , in the direction suggested by the critic.



Thus the can be written as:

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(s, a) Q_w(s, a)]$$

Let's see how it works in a simple action-value actor-critic algorithm.

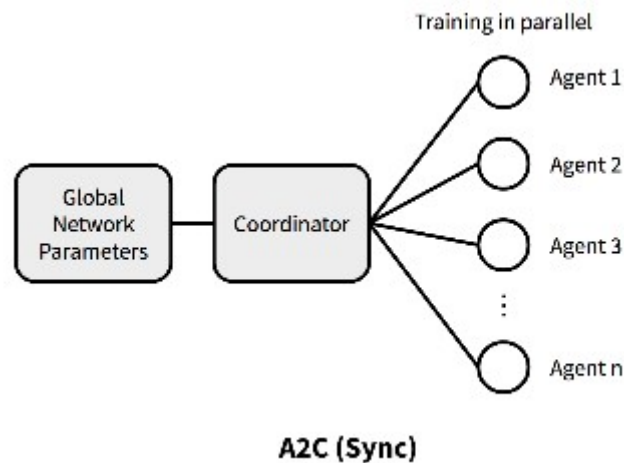
1. Initialize  $s, \theta, w$  at random
2. For  $t=1 \dots T$ 
  3. Pick an action  $A$  from  $\pi_{\theta}(a|s)$
  4. Sample reward  $R_t \sim R(s, a)$  and next state  $s' \sim \pi(s'|s, a)$
  5. Update the policy parameters:  $\theta \leftarrow \theta + \alpha_{\theta} Q_w(s, a) \nabla_{\theta} \ln \pi_{\theta}(a|s)$
  6. Compute the correction (TD error) for action-value at the time
$$\delta_t = r_t + \gamma Q_w(s', a') - Q_w(s, a)$$
and use it to update the parameters of the action-value function:
$$w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s, a)$$
7. Update  $a \leftarrow a'$  and  $s \leftarrow s'$ .

Both Reinforce and the vanilla version of the actor-critic method are on-policy: training samples are collected according to the target policy, the very same policy that we try to optimize for. However, this method suffers from high variance problems such as the Reinforce algorithm. A baseline needs to be introduced to the actor-critic model to alleviate this problem.

### 3.1 Advantage Actor-Critic

Advantage Actor-Critic (Mnih et al., 2016), short for A2C, is a classic actor-critic, policy gradient method that uses the advantage baseline to reduce the variance. This method also benefits from parallel training. In A2C, the critics learn the value function while multiple actors are trained in parallel and get synced every time to update the shared parameters. As an example, let's use the state-value function. The loss function for state value is  $J_v(w) = (G_t - V_w(s))^2$ , and gradient descent can be used to find the best  $w$ . In the policy gradient update, this state-value function acts as the baseline.

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



Here a simple algorithm outline:

1. Initialize  $s, \theta, w$  at random
2. For  $t=1 \dots T$ 
  3. Pick an action  $A$  from  $\pi_{\theta}(a|s)$
  4. Sample reward  $R_t \sim R(s, a)$  and next state  $s' \sim \pi(s'|s, a)$
  5. Decrease the reward with gamma  $R = \gamma R + R_t$
  6. Update the parameters  $\theta$  and  $w$ 
    - a.  $\theta': d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi_{\theta'}(a_i | s_i) (R - V_{w'}(s_i));$
    - b.  $w': dw \leftarrow dw + 2(R - V_{w'}(s_i)) \nabla_{w'} (R - V_{w'}(s_i))$

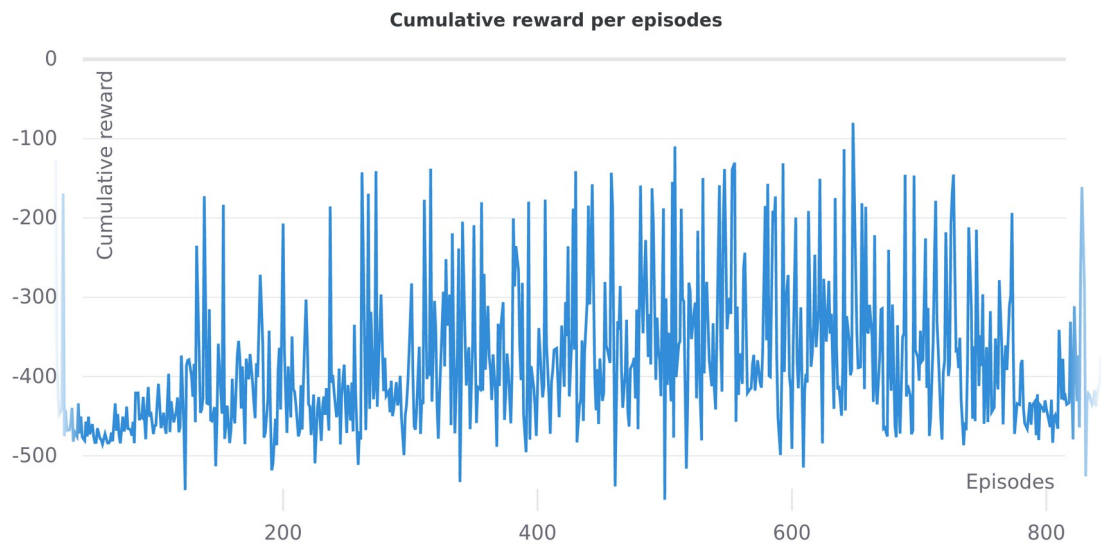
A2C algorithm relies on the parallelization of the environments which will produce every time multiple sets of states, rewards and actions. However parallel environments provide the agent

with sets of actions and observations irregularly. To overcome the inconsistency, an A2C coordinator waits until all parallel actors have completed their work before updating the global parameters, and then parallel actors start from the same policy in the next iteration. The synchronized gradient update helps to make the training more coherent and can even speed up convergence.

## 4. Experiments and results

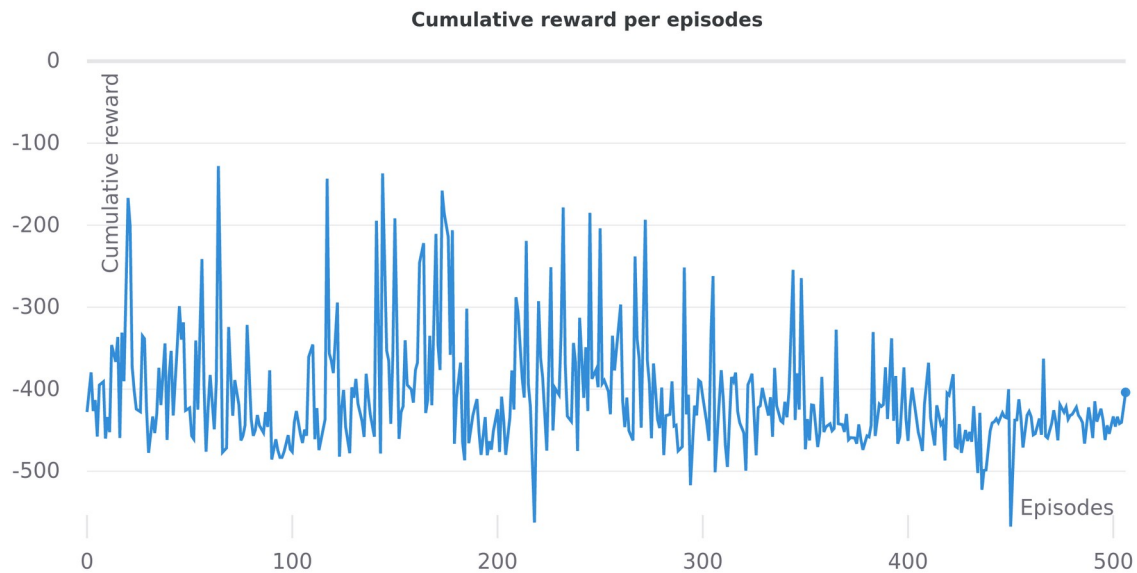
For the experiments on the A2C algorithm we used the library of stable-baselines. We carried experiments on the A2C algorithm with different parameters. The one that contributed the most was the  $n$  step size. The  $n$  step size is the interval in which the agent will update its parameters and also its the interval when observations are made. Smaller step size may help the agent to learn better short term actions but it lacks in long term policies. Bigger step size may lead to a better policy but the agent might struggle on simple short term actions.

Below there is the graph of the cumulative reward per episode of a training sequence with step size 16. The training took about 10 hours with only one environment due to hardware limitations.



As we have mentioned before, the policy gradient models show high variance. This experiment comes to verify our previous analysis. The model not only has high variance but it also finds it very difficult to converge. From another experiment that took another 10 hours with step size 32 we can see no difference.

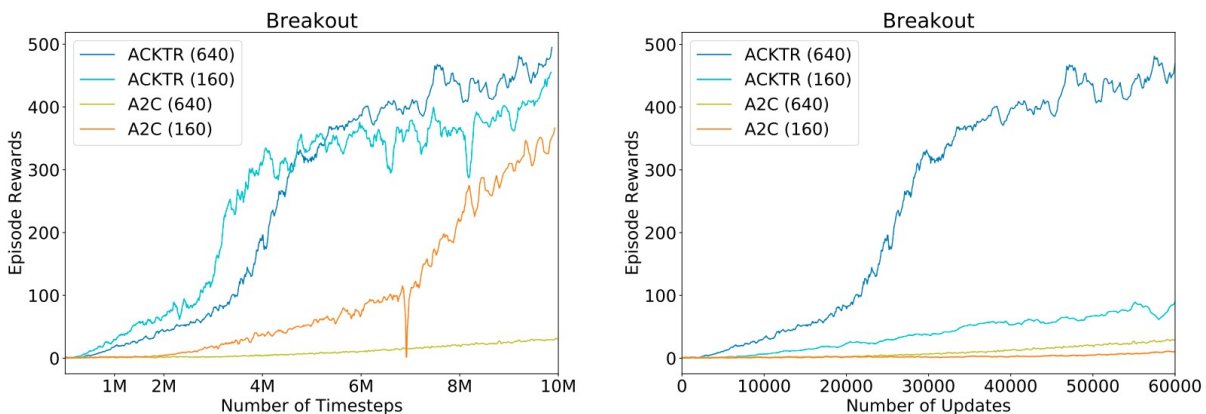




The results of our experiments are not satisfactory. Many factors can be responsible for these results. Although the most important seem to be the model itself and the physical limitations of our training.

Firstly, the A2C model benefits from parallel environments. By having that large number of data the sampled data could reach independent and identically distributed random variable status. Our experiments we run in google colab which can offer for free a very useful and expensive GPU on the other hand offers only 1 CPU core and this does not help in the parallelization and the simultaneous execution of the experiments. On the other hand we would be able to run the experiments for a longer time but this would not be realistic as already 10 hours was too much for one runtime.

Secondly and finally, the A2C model may not be capable of reducing the variance to an acceptable level in this complex environment. The purpose of the advantage baseline to address this problem. However this baseline is very simple and in many may environments might not work that well. Evidence of its inefficiency can be found on a Open-ai article with policy gradient models benchmarked on several environments.



A2C model seems to need millions of timesteps with multiple parallel environments to reach a better reward. That proves to be insufficient. That is the reason why we need better policy gradient methods to beat the problem of high variance with elegant mathematical solutions.

# Proximal Policy Optimization

## 1. Introduction

I took over trying the class of reinforcement learning algorithm, *Proximal Policy Optimization (PPO)*, which perform comparably or better than state-of-the-art approaches while being much simpler to implement and tune. PPO has become the default reinforcement learning algorithm at *OpenAI* because of its ease of use and good performance.

### Why would anyone choose PPO?

This is a real good question that deserves a good explanation. Policy gradient methods are fundamental to recent breakthroughs in using deep neural networks for control, from video games, to 3D locomotion and Go. However, getting good results via policy gradient methods is challenging because of:

1. **Unstable Policy Update:** In many Policy Gradient Methods, policy updates are unstable because of larger step size, which leads to bad policy updates and when this new bad policy is used for learning then it leads to even worse policy. And if steps are small then it leads to slower learning.
2. **Data Inefficiency:** Many learning methods learn from current experience and discard the experiences after gradient updates. This makes the learning process slow as a neural network takes lots of data to learn.

PPO comes to overcome the above issues. It has some of the benefits of ***Trust Region Policy Optimization (TRPO)***, but it is much simpler to implement, more general, and have better sample complexity (empirically).

## 2. Background

Firstly, I have to emphasize that PPO belongs to the *On-Policy* learning algorithms. On-Policy learning algorithms are the algorithms that evaluate and improve the same policy which is being used to select actions. That means we will try to evaluate and improve the same policy that the agent is already using for action selection. In short, Target Policy = Behavior Policy. The main purpose of PPO is to strike a balance between

1. Ease of Implementation
2. Sample Efficiency
3. Ease of Tuning

Unlike popular Q learning approaches, like DQN, that can learn from stored offline data, PPO learns online. This means that it doesn't use a replay buffer to store past experiences, but instead it learns directly from whatever its agent encounters in the environment. Once a batch of experience has been used to do a gradient update, the experience then is discarded and the policy moves on. This also means that Policy Gradient Methods are typically less sample efficient than Q learning methods because they only use the collected experience once for doing an update.

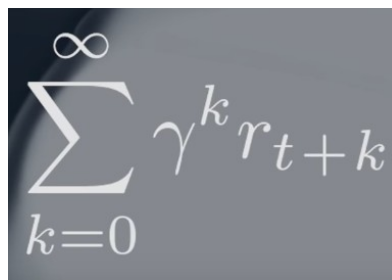
## 2.1 Policy Gradient Methods

General Policy Optimization Methods usually start by defining the policy gradient loss as the expectation over the logarithm of the policy actions multiplied by an estimate of the advantage function.

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t \left[ \log \pi_{\theta}(a_t | s_t) \hat{A}_t \right].$$

So, what are all these mean? The first term  $\pi_{\theta}$  is our policy. It's a neural network that takes the observed states from the environment as an input and suggests actions to take as an output. The second term is the **Advantage Function A**, which basically tries to estimate what the relative value is of the selected action in the current state. In order to compute the advantage we need two things: we need to discounted the sum of rewards and a Baseline Estimate.

So the first part is the discounted sum of rewards. This is basically a weighted sum of all the rewards the agent have during each time step in the current episode. Then the discount factor gamma which is usually somewhere between 0.9 and 0.99, accounts for the fact that your agent cares more about reward that is going to get very quickly, versus the same reward it would get a hundred times that for now. This is exactly the same idea as interest in the financial world in the sense that getting money tomorrow is usually more valuable than getting the same amount of money e.g in a year from now.


$$\sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

Also, I have to mention that the advantage is calculated after the episode sequence was collected from the environment, so in other words we know all the rewards. There is no guessing involved in computing the discount because we actually know what happened.

The second part of the Advantage Function A, is the Baseline or the Value Function. Basically, the value function tries to give an estimate of the discounted sum of rewards from this point on-wards. In other words, it's trying to guess what the final return is going to be in this episode starting from the current state. During training this neural network that's representing the value function is going to be frequently updated using the experience that our agent collects in the environment because this is basically a supervised learning problem. You're taking states as an input and your neural net is trying to predict what the discounted sum of rewards is going to be from this state on-wards. So basic supervised learning. Notice that because this value is the output of a neural network, this is gonna be a noisy estimate. There's gonna be some variance because our network is not going to always predict the exact value of that states, so basically we're going to end up with a noisy estimate of the value function.

So now we have the two terms that we need. We have the discounted sum of rewards, that we computed from our episode roll-out and we have an expectation, an estimate of that value given the state that we're in. If we subtract the baseline estimate from the actual return we got, we get what we call the advantage estimate. So basically the advantage estimate is answering the question: "How much better was the action that I took based on the expectation of what would normally happen in the state that I was in?". So basically the action that our agent took was it better than

expected or was it worse?. Then by multiplying the log probabilities of your policy actions with this advantage function, we get the final optimization objective that is used in Policy Gradient Methods.

At this point, there are two cases. If the advantage estimate was positive, means that the actions that the agent took in the sample trajectory, resulted in better than the average return. In this case, we will increase the probability of selecting them again in the future, when we encounter in the same state. On the other hand, if the advantage function was negative, then we will reduce the likelihood of the selected actions.

## 2.2 Trust Region Methods

As I've already mentioned, one of the problems is that if you simply keep running gradient descent on one batch of collected experience, what will happen is that you'll update the parameters in your network so far outside of the range where this data were collected. For example, the advantage function, which is in principle a noisy estimate of the real advantage, is going to be completely wrong. So, in a sense, you are just going to destroy your policy, if you keep running gradient descent on a single batch of collected experience.

In order to solve this issue, one successful approach is to make sure that if you're updating the policy, you are never going to move too far away from the old policy. This idea is called *Trust Region Policy Optimization* or *TRPO*, which is actually the whole basis on which PPO was built. So here is the objective function that was used in TRPO and if you compare this with the previous objective function of the Vanilla Policy Gradients, what you can see is that the only thing that changed in this formula is that the log operator is replaced with the division by  $\pi_{\theta_{old}}$ .

$$\begin{aligned} \underset{\theta}{\text{maximize}} \quad & \hat{\mathbb{E}}_t \left[ \frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{old}}(a_t | s_t)} \hat{A}_t \right] \\ \text{subject to} \quad & \hat{\mathbb{E}}_t [\text{KL}[\pi_{\theta_{old}}(\cdot | s_t), \pi_{\theta}(\cdot | s_t)]] \leq \delta. \end{aligned}$$

The above formula shows that optimizing this TRPO objective, is in fact identical to vanilla policy gradients. Now to make sure that the updated policy doesn't move too far away from the current policy, TRPO adds a KL constraint to the optimization objective. What this KL constraint effectively does is that it's just going to make sure that the new updated policy doesn't move too far away from the old policy. So, in a sense, we just want to stick close to the region, where we know everything works fine. The problem is that this KL constraint adds additional overhead to our optimization process and can sometimes lead to very undesirable training behavior. So, wouldn't it be nice if we can somehow include this extra constraint directly into our optimization objective? Well as you might have guessed, that is exactly what PPO does.

## 3. Clipped Surrogate Objective

Let  $r_t(\theta)$  denote the probability ratio  $r_t(\theta) = \frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{old}}(a_t | s_t)}$ , so  $r(\theta_{old}) = 1$ . TRPO maximizes a “surrogate” objective

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t \left[ \frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{old}}(a_t | s_t)} \hat{A}_t \right] = \hat{\mathbb{E}}_t [r_t(\theta) \hat{A}_t]. \quad (6)$$

So now that we have a little bit of surroundings, let's dive into the crux of the algorithm the central optimization objective behind PPO. Firstly, let's define a variable  $r(\theta)$ , which is just a probability ratio between the new updated policy outputs and the outputs of the previous old version of the policy network. Given a sequence of sampled actions and states, this  $r(\theta)$  value will be larger than 1, if the action is more likely now than it was in the old version of the policy. Moreover, it will be somewhere between 0 and 1, if the action is less likely now than it was before the last gradient step. Then, if we multiply this ratio  $r(\theta)$  with the advantage function, we get the normal TRPO objective in a more readable form. With this notation we can finally write down the central objective function that is used in PPO. Here it is,

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

At first sight, looks surprisingly simple. Well first of all, you can see that the objective function that PPO optimizes, is an expectation operator. So this means that we're going to compute this over batches of trajectories and this expectation operator is taken over the minimum of two terms. The first of these terms is  $r(\theta)$  multiplied by the advantage estimate. So this is the default objective for normal policy gradients, which pushes the policy towards actions that yield a high positive advantage over the baseline.

Now the second term is very similar to the first one, except that it contains a truncated version of this  $r(\theta)$  ratio, by applying a clipping operation between  $1 - \epsilon$  and  $1 + \epsilon$ , where epsilon is usually something like 0.2. Then, the min operator is applied to the two terms to get the final result. While this function looks rather simple at first sight, it takes a little bit more effort.

Firstly it's important to note that the advantage estimate can be both positive and negative and this changes the effect of the min operator. Here is a plot of the objective function, for both positive and negative values of the advantage estimate.

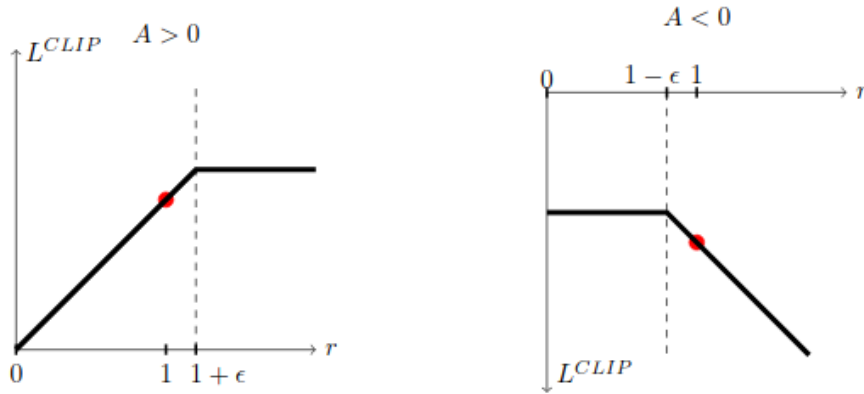


Figure 1: Plots showing one term (i.e., a single timestep) of the surrogate function  $L^{CLIP}$  as a function of the probability ratio  $r$ , for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimization, i.e.,  $r = 1$ . Note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

On the left half of the diagram, the advantage function is positive or there all the cases where the selected action had a better-than-expected effect on the outcome. On the right half of the diagram we can find situations, where the action had an estimated negative effect on the outcome. On the left side, notice how the loss function flattens out, when  $r$  gets too high. This happens when the action is a lot more likely under the current policy, than it was under the old policy. In this case, we don't want to overdo the action update too much, so the objective function gets clipped here to limit the effect of the gradient update.

On the right side, where the action had an estimated negative value, the objective flattens when  $r$  goes near zero. This corresponds to actions that are much less likely now than in the old policy and it will have the same effect of not overdoing a similar update, which might otherwise reduce these action probabilities to zero. Remember, the advantage function is noisy, so we don't want to destroy a policy based on a single estimate.

Finally, what about the very right hand side? Well, the objective function only ends up in this region, when the last gradient step made the selected action a lot more probable, so  $r$  is big, while also making our policy worse, since the advantage is negative here. If that's the case then, we would really want to undo the last gradient step and this is what the objective function in PPO allows us to do. The function is negative here, so the gradient will tell us to walk the other direction and make the action less probable by an amount proportional to how much we screwed it up in the first place. Also notice that this is the only region where the unclipped part of the objective function has a lower value than the clipped version and those gets returned by the minimization operator.

As we can understand, the PPO does the same as a TRPO, which forces the policy updates to be conservative, if they move very far away from the current policy. The only difference is that PPO does this with a very simple objective function, that doesn't require to calculate all these additional constraints or KL divergences. In fact, it turns out that the simple PPO objective function, often outperforms the more complicated variant that we have in TRPO. Simplicity often wins.

#### *TRPO Objective vs PPO Objective:*

$$\begin{aligned} &\underset{\theta}{\text{maximize}} \quad \hat{\mathbb{E}}_t \left[ \frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)} \hat{A}_t \right] \\ &\text{subject to} \quad \hat{\mathbb{E}}_t [\text{KL}[\pi_{\theta_{\text{old}}}(\cdot | s_t), \pi_{\theta}(\cdot | s_t)]] \leq \delta. \end{aligned}$$

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

## 4. Algorithm

Now that we've seen the central objective function behind PPO, let's take a look at the entire algorithm end to end. As mentioned before, there are two alternating loops in PPO. In the first one, the current policy is interacting with the environment, generating episode sequences for which we immediately calculate the advantage function using our fitted baseline estimate for the state values. Then, every so many episodes, a second loop is going to collect all that experience and run gradient descent on the policy network, using the clips PPO.

---

### Algorithm 1 PPO, Actor-Critic Style

---

```

for iteration=1, 2, ... do
  for actor=1, 2, ..., N do
    Run policy  $\pi_{\theta_{\text{old}}}$  in environment for  $T$  timesteps
    Compute advantage estimates  $\hat{A}_1, \dots, \hat{A}_T$ 
  end for
  Optimize surrogate  $L$  wrt  $\theta$ , with  $K$  epochs and minibatch size  $M \leq NT$ 
   $\theta_{\text{old}} \leftarrow \theta$ 
end for

```

---



I shouldn't skip the fact that the final loss function that is used to train an agent, is the sum of this clips PPO objective, that we just saw, plus two additional terms. The first additional term of the loss function, is basically in charge of updating the baseline network. The last term in the objective function, is called the entropy term and this term is in charge of making sure that our agent does enough exploration during training. So in contrast to discrete action policies, that output the action choice probabilities, the PPO policy head outputs the parameters of a Gaussian distribution for each available action type. When running the agent and training mode, the policy will then sample from these distributions to get a continuous output value for each action head.

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

Basically the entropy of a stochastic variable, which is driven by an underlying probability distribution, is the average amount of bits that is needed to represent its outcome. It is a measure of how unpredictable an outcome of this variable really is and maximizing its entropy, will force it to have a wide spread over all the possible options resulting in the most unpredictable outcome. So this gives some intuition as to why adding an entropy term will push the policy to behave a little bit more randomly until the other parts of the objective start dominating. As always we have a couple of hyperparameters  $c_1$  and  $c_2$ , that wave the contributions of these different parts in the loss function.

#### *Entropy Function:*

$$H(X) = - \sum_{i=1}^n P(x_i) \log_b P(x_i)$$

The important thing to remember though, is that PPO wasn't specifically designed for sample efficiency, but rather to address the really complicated code that was needed for a lot of other algorithms and making it relatively easy to tune in terms of hyperparameters. Because PPO achieves both of those objectives, while also yielding close to or above state-of-the-art performance on a wide range of tasks, it has become one of the benchmarks in Deep Reinforcement Learning.

In summary, PPO is a state of the art policy gradient method. The algorithm has the stability and reliability of TRPO, but is much simpler to implement, requiring only a few tweaks to vanilla policy gradient methods and it can be used for a wide range of reinforcement learning tasks.

## **5. Experiments with PPO**

For our experiments, we took advantage of the Open Source Library, that we have already mention, of Stable Baselines. At the beginning, we selected the Pytorch implementation of the new updated library of Stable Baselines 3 PPO, but the results weren't so satisfactory (maybe caused by some potential bug of the new library). So we switched this with the PPO2 model of the previous version of Stable Baselines, that it is implemented in TensorFlow. Specifically, PPO2 is the



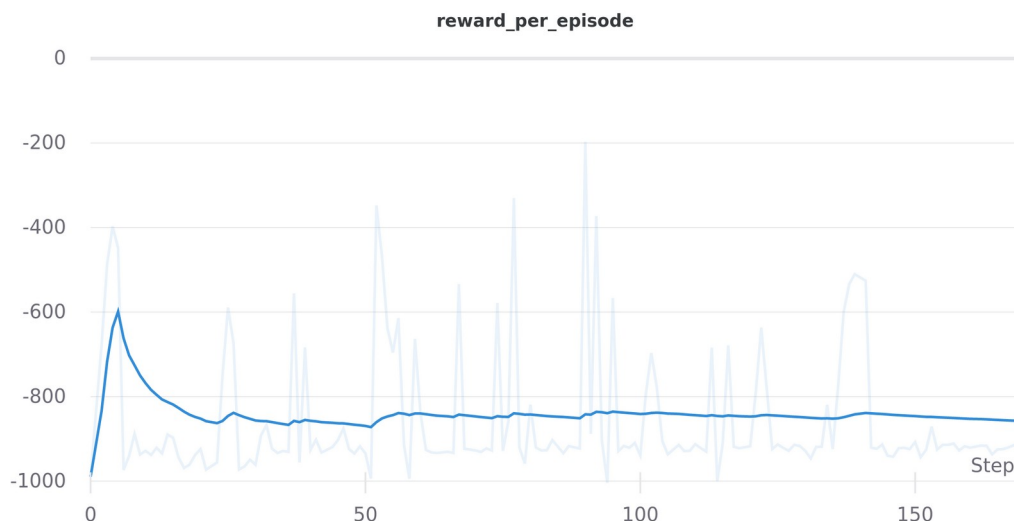
implementation of OpenAI made for GPU. They claim that the PPO2 version is 3 times faster than the implementation of PPO1, because it used the GPU, is the reason we were selected it.

Here are the hyperparameters that we used, while we train the model:

- ***gamma***=0.99
- ***n\_steps***=64
- ***ent\_coef***=0.01
- ***learning\_rate***=0.00025
- ***vf\_coef***=0.5
- ***max\_grad\_norm***=0.5
- ***lam***=0.95
- ***nminibatches***=4
- ***noptepochs***=4
- ***cliprange***=0.2

We have to mention that we didn't experimented so much with the hyperparameters, due to the fact that the default ones gives us already pretty descent performance. Besides, this is the good with PPO, that is an already easy to fine tuned algorithm, because of its simplicity.

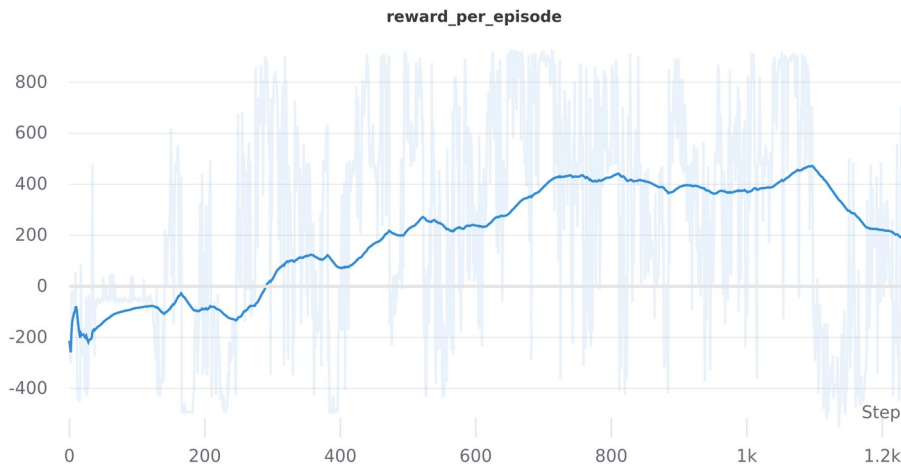
However, it is worth to mention that we experiment a lot with the number of timesteps and the number of episodes. At our first experiments, we tried mainly big numbers of timesteps and episodes (10000\*100). We observed that, in first videos of our Car Racing, in which the car didn't knew the environment well, it went out of the road and it stayed on the grass very long time. So, we decided that it needn't so much time for it to learn that the grass isn't a good thing and we reduced the amount of timesteps in the half (5000\*100). So now, every time that it went in the grass section, it didn't stayed so much time on it and started from the beginning again.



**Example 1:** 10.000 timesteps \* 150 episodes

When we found the appropriate hyperparameters, we trained it for almost two days in Google Colab. Training these kind of Reinforcement Learning models is a very time consuming procedure. So we increased the number of episodes until 1.3k episodes. After almost 1100 episodes,

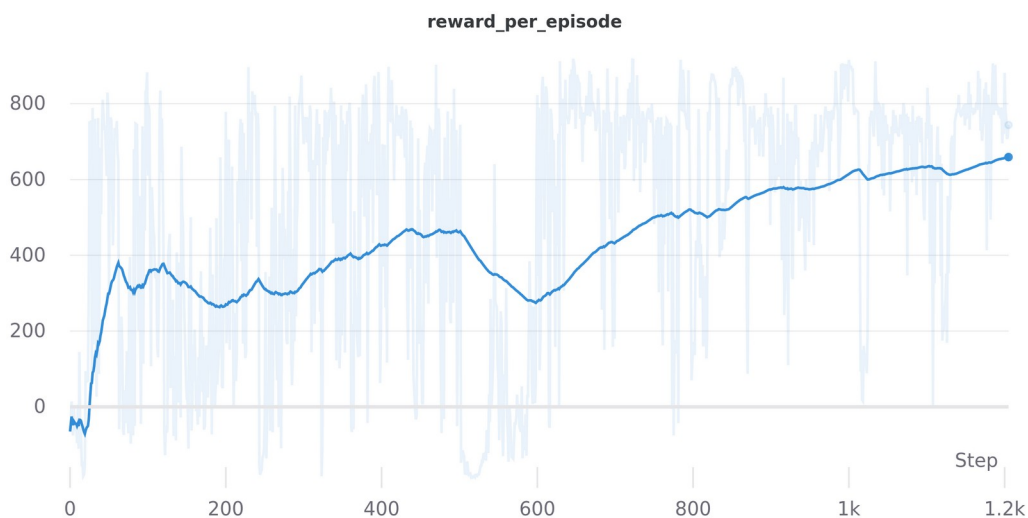
we observed that the reward per episode was decreased dramatically. So we decide to stop the training procedure after 2 days of training.



**Example 2:** 5.000 timesteps \* 1.200 episodes

The car takes a lot of time to learn how to stay on the road and get good and positive rewards. But when, for example, it learned to take one turn very well, then in the next episodes we notice that the car increased its speed in significant level. This is the reason that sometimes went off-road fast, but because of the negative value of advantage function, the car has the potential to “undo” this very big mistake and change direction quickly.

We did many experiments with other values of timesteps and episodes, but they aren’t worth to mention. One very interesting phenomenon that we observed, is that the more we reduced the timesteps, the faster the car got better rewards. If you see the above diagram, that we run it for 2000 timesteps, you will notice that the rewards are very high from the very first episodes. It needed only 30 episodes to take a mean reward of ~360, in contrast to the previous experiment, that needed ~500 episodes to take the same high score! This leads to the result that, the shorter the timesteps, the bigger the rewards it gets in less time.

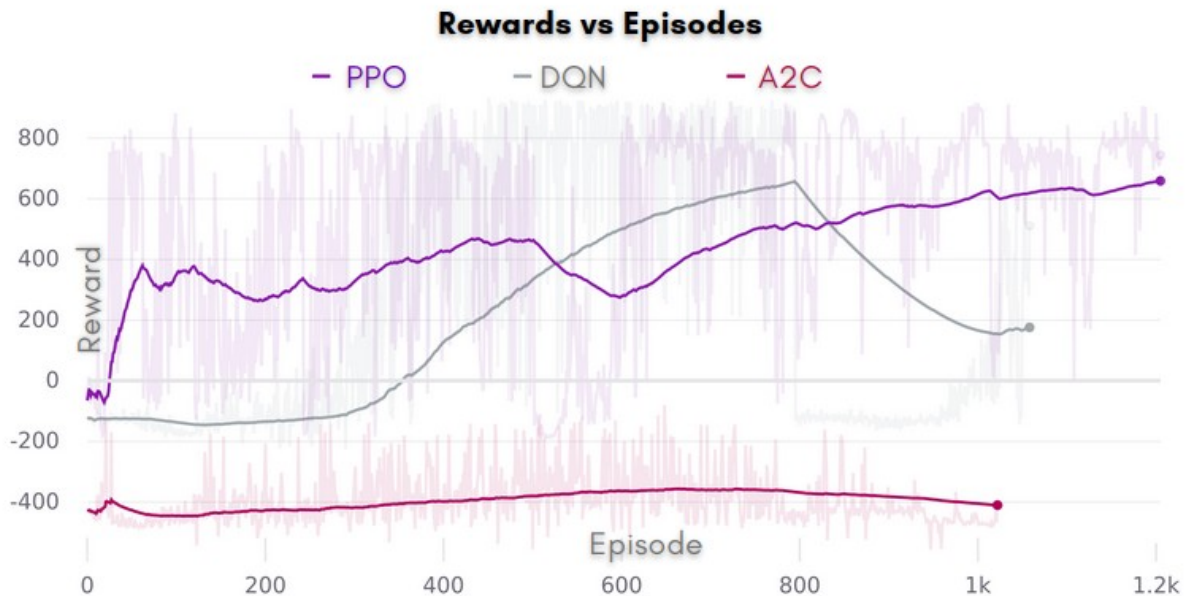


**Example 3:** 2.000 timesteps \* 1.200 episodes

If we continue the training, we are sure that the rewards would increased even more. However, because of the small amount of timesteps, the video has very little duration and the car didn't manage to explore the track in big level. The video is only 40 seconds and it is very logic that the car learn a little piece of track very easily and hence the great rewards. So, this is the reason that we didn't pick it as the best model. Instead of this model, we pick as the best the model with 5000, because we thought that the duration of 1:40 minutes is a decent time to learn to the car a big amount of the track and allow it to explore it more.

**Note:** Inside the file, you will find the videos of these three experiments.

## Conclusion



Deep Reinforcement Learning has the potential to revolutionize AI and is a step toward creating autonomous systems, that have a higher-level understanding of the visual environment directly from pixels. Our experimentation with the Car-racer environment helped us understand the benefits and problems that arise from training deep reinforcement learning models. Starting from Q-learning, we saw that when we borrow elements from the classic supervised learning, we can achieve very good results in a field that has nothing to do with supervision. This clever way of sampling replayed data, contributed to the models learning effectively from almost independent and identically distributed data. But this replay memory comes with a computational cost. The Policy Gradient methods help to partially solve this problem, as the agent learns only from the data. This method also offers us the flexibility to use stochastic policies, which may be particularly useful in some environments. Actor-critic algorithms, by utilizing efficient neural networks can build better insight into the environment. However, these methods suffer from a high variance problem. Assuming without knowing that your policy is the best, can lead you to complicated paths that add up the overall variance.

We need an elegant solution to this problem that won't misuse computational resources. PPO was developed to fix the above issues. By using the mathematical model of Trust Region, PPO archives state-of-the-art performance, while keeping it simple. This method pushes the agent to pick better actions that are meaningful to the policy, thus reducing the variance and accelerating the learning process. Even though, PPO is currently the state-of-the-art reinforcement learning algorithm, many challenges remain towards to Artificial General Intelligence's goal. The environment was a simplified version of an autonomous vehicle. Questions like, if the model has enough capacity to capture the complexity of the real world or concerns about security, still remain. Although in a couple of hours of training on a laptop, we could train a simplified autonomous car to drive by itself. The future seems promising for Reinforcement Learning.

## Sources

Github Repository – Proximal Policy Optimizer Implementation

[elsheikh21/car-racing-ppo: Implementation of a Deep Reinforcement Learning algorithm, Proximal Policy Optimization \(SOTA\), on a continuous action space openai gym \(Box2D/Car Racing v0\)](#)

Github Repository – World-Models-Tensorflow

[dariocazzani/World-Models-TensorFlow](#)

Paper – Reinforcement Learning for a Simple Racing Game

[2D Racing game using reinforcement learning and supervised learning Abstract](#)  
[Reinforcement Learning for a Simple Racing Game](#)

Paper – Challenging On Car Racing Problem from OpenAI gym

[https://arxiv.org/pdf/1911.04868.pdf](#)

Paper – Deep Reinforcement Learning for End-to-End autonomous driving

[Deep Reinforcement Learning for End-to-End autonomous driving](#)

Github Repository – CarRacing-v1

[https://github.com/NotAnyMike/gym](#)

Kaggle's Notebook – Learn by example Reinforcement Learning with Gym

[https://www.kaggle.com/charel/learn-by-example-reinforcement-learning-with-gym](#)

TensorFlow's Introduction to RL and Deep Q Networks

[https://www.tensorflow.org/agents/tutorials/0\\_intro\\_rl](#)

DNQBook's Introduction to RL

[https://www.dqnbook.com/2/introduction](#)

Article – A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python

[https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/](#)

Article – Introduction to Various Reinforcement Learning Algorithms

[https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287](#)

Article – Deep Reinforcement Learning. Introduction. DQN Algorithm.

[https://medium.com/@markus.x.buchholz/deep-reinforcement-learning-introduction-deep-q-network-dqn-algorithm-fb74bf4d6862](#)

Library – Stable Baselines

[https://stable-baselines.readthedocs.io/en/master/](#)

Article – Applying a Deep Q Network for OpenAI's Car Racing Game

[https://towardsdatascience.com/applying-a-deep-q-network-for-openais-car-racing-game-a642daf58fc9](#)

Paper – Playing Atari with Deep Reinforcement Learning

<https://arxiv.org/pdf/1312.5602v1.pdf>

Article – RL-DQN Deep Q-Network

<https://jonathan-hui.medium.com/rl-dqn-deep-q-network-e207751f7ae4>

Article – Simple Reinforcement Learning with Tensorflow Part 4

<https://awjuliani.medium.com/simple-reinforcement-learning-with-tensorflow-part-4-deep-q-networks-and-beyond-8438a3e2b8df#.y5m9i5d8w>

Article – Welcome to Deep Reinforcement Learning Part 1: DQN

<https://towardsdatascience.com/welcome-to-deep-reinforcement-learning-part-1-dqn-c3cab4d41b6b>

Article – Introduction to Reinforcement Learning Part 2: Q-Learning

<https://markelsanz14.medium.com/introduction-to-reinforcement-learning-part-2-q-learning-4d93f9f37e3e>

Article – Introduction to Reinforcement Learning Part 2: Q-Learning with Neural Networks, Algorithm DQN

<https://markelsanz14.medium.com/introduction-to-reinforcement-learning-part-3-q-learning-with-neural-networks-algorithm-dqn-1e22ee928ecd>

Article – Using Keras and Deep Q-Network to Play FlappyBird

<https://yanpanlau.github.io/2016/07/10/FlappyBird-Keras.html>

Article – Going Deeper Into Reinforcement Learning: Understanding DQN

<https://danieltakeshi.github.io/2016/12/01/going-deeper-into-reinforcement-learning-understanding-dqn/>

Article – On-Policy vs Off-Policy Reinforcement Learning

<https://analyticsindiamag.com/reinforcement-learning-policy/>

Article - On-Policy vs Off-Policy vs Offline Reinforcement Learning

<https://towardsdatascience.com/off-policy-vs-on-policy-vs-offline-reinforcement-learning-demystified-f7f87e275b48>

Article – On-Policy vs Off-Policy Learning

<https://towardsdatascience.com/on-policy-vs-off-policy-learning-75089916bc2f>

Article – An Intuitive Explanation of Policy Gradient

<https://towardsdatascience.com/an-intuitive-explanation-of-policy-gradient-part-1-reinforce-aa4392cbfd3c>

Article – Understanding Actor Critic Methods and A2C

<https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f>

David Silver's Lectures

<https://www.davidsilver.uk/teaching/>

Paper – Asynchronous Methods for Deep Reinforcement Learning

<https://arxiv.org/pdf/1602.01783.pdf>

Article – Reinforcement Learning models

<https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html#a2c>

Article – Going Deeper Into Reinforcement Learning: Fundamentals of Policy Gradients

<https://danieltakeshi.github.io/2017/03/28/going-deeper-into-reinforcement-learning-fundamentals-of-policy-gradients/>

Article - High Variance's Explanation

<https://www.quora.com/Why-does-the-policy-gradient-method-have-a-high-variance?share=1>

Blog – OpenAI A2C

<https://openai.com/blog/baselines-acktr-a2c/>

Paper - Proximal Policy Optimization Algorithms

<https://arxiv.org/pdf/1707.06347.pdf>