Reinforcement Learning Presentation

Giannoutsos Andreas sdi1700021
Apostolopoulou Alexandra sdi1700005
Briakos Spyros sdi1700101

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, where 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 typical Reinforcement Learning Problem .1

In the complete RL problem, **the state changes every time we take an action.**

So the **agent gets the state in which the environment is in**, represented with the letter s.

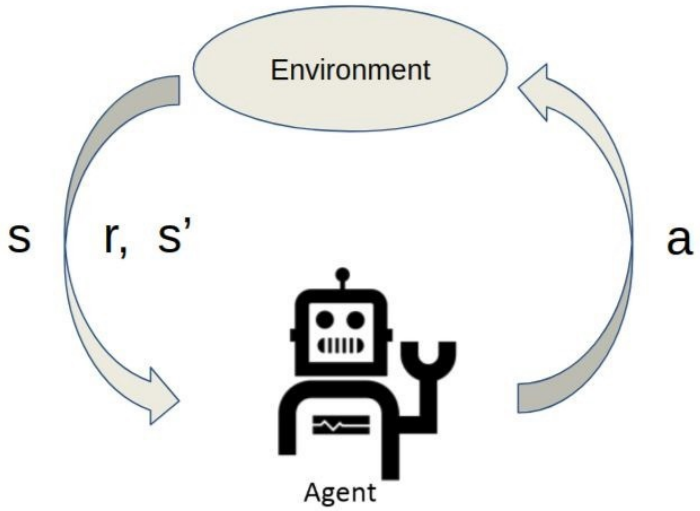
The agent then chooses which action to take, represented with the letter a.

After following that action, **the environment provides a reward**, represented with the letter r

And finally **transitions to a new state**, represented as s'.

This is the main RL cycle and can be observed in above figure.

Quick Representation of typical Reinforcement Learning Problem .2



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:*** *s1, a1,r1; s2, a2, r2; …; sT, aT, rT. Where T indicates the last step in the episode.*

Deep Q-Network (DQN) .1

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 $R_{t0}=\sum_{t=t0}^{\infty}\gamma^{t-t0}r_t$, where R_{t0} is also known as the **return**.

The **discount, γ**, 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.

Deep Q-Network (DQN) .2

The main idea behind Q-learning is that if we had a function $Q*:\text{State}\times\text{Action}\rightarrow R$, **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: $\pi*(s)=\text{argmax}_a\, Q*(s,a)$

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 .1

The intuition behind this 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 γ**, which is a value with range **γ ∈ (0, 1]**. This value γ 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.

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

Bellman's Equation .2

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 **θ**, and it is used **to estimate** the Q-values for the current state **s** and action **a**: **Q(s, a; θ)**.

Bellman's Equation .3

The second one is the **target neural network**, parametrized by the weight vector **θ´**, 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´: Q(s´, a´; θ´).**

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).

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 θ and θ´.

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

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? .1

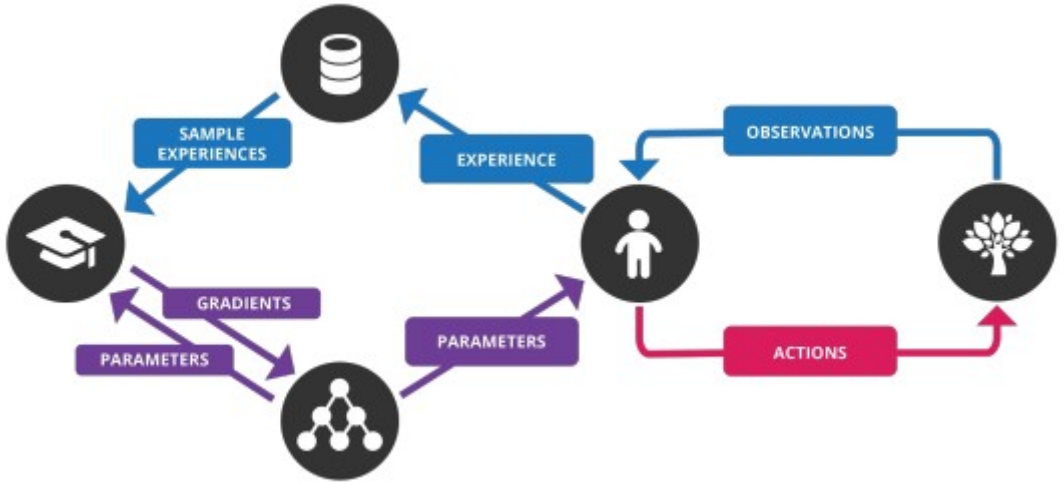
We'll be using experience replay memory for training our DQN. Replay memory ***stores*** *the* ***transitions*** *that the agent observes, allowing us to reuse this data later. By sampling from it* ***randomly****, the transitions that build up a batch are* ***decorrelated*** *and silmutaneously we prevent 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.*

What is Replay Memory? .2

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.*



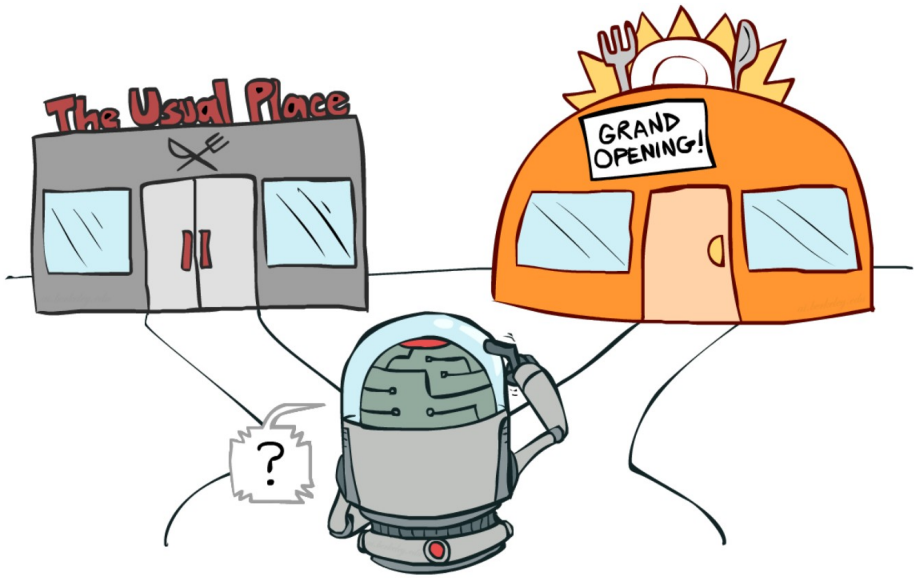
~A more sophisticated sampling strategy might emphasize transitions from which we can learn the most, similar to prioritized sweeping.

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 ϵ 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



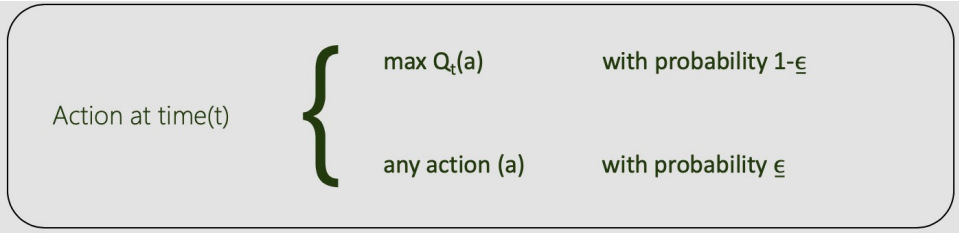
How much of an agent's time should be spent exploiting its existing known-good policy, and how much time should be focused on exploring new, possibility better, actions? We often encounter similar situations in real life too. For example, we face on which restaurant to go to on Saturday night. We all 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 ***ϵ-Greedy approach****, so let's explore it!*

ϵ-Greedy Approach

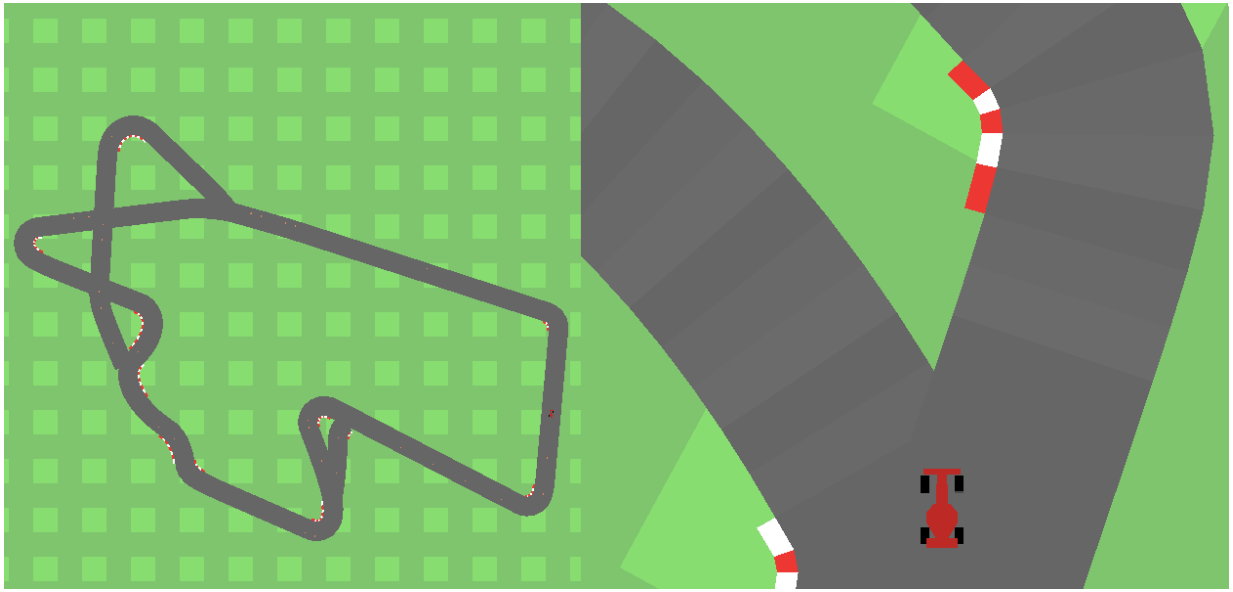
***Explanation**: A simple **combination** of the **greedy and random** approaches yields one of the most used exploration strategies: ϵ-greedy. In this approach the agent chooses what it believes to be the optimal action most of the time, but occasionally acts randomly. This way the agent takes actions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

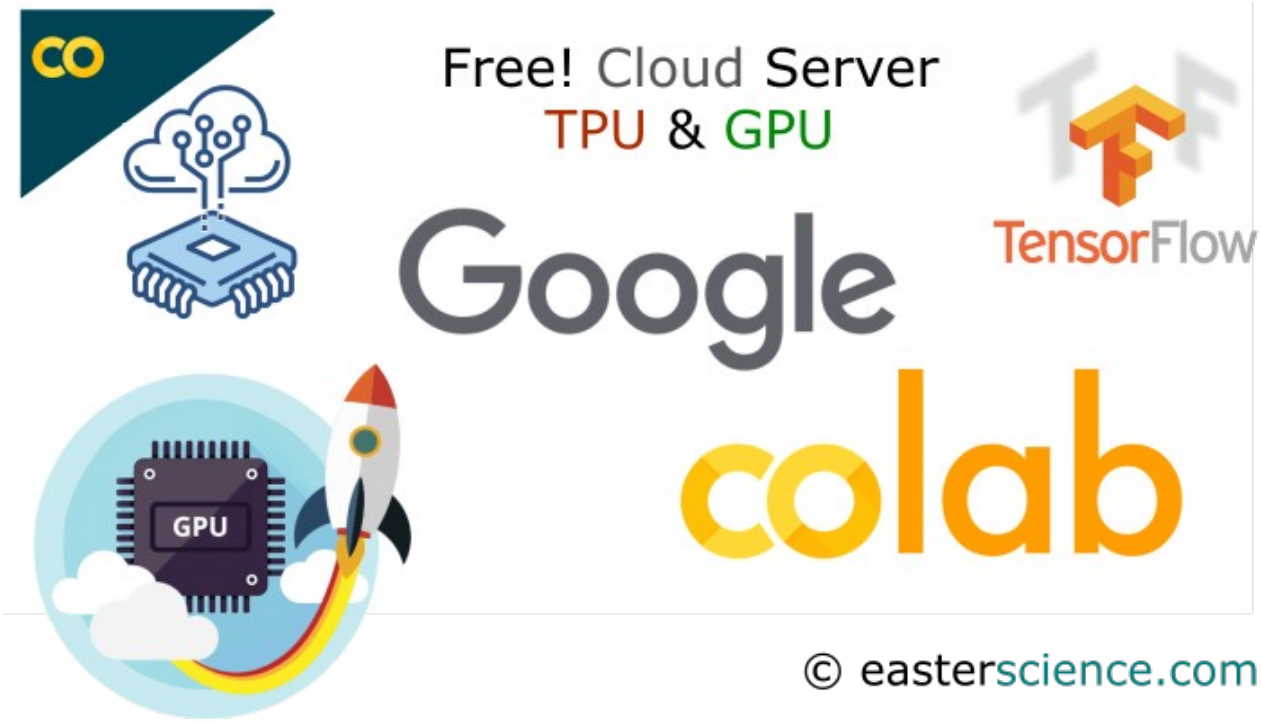
CarRacing-v0 Environment

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)

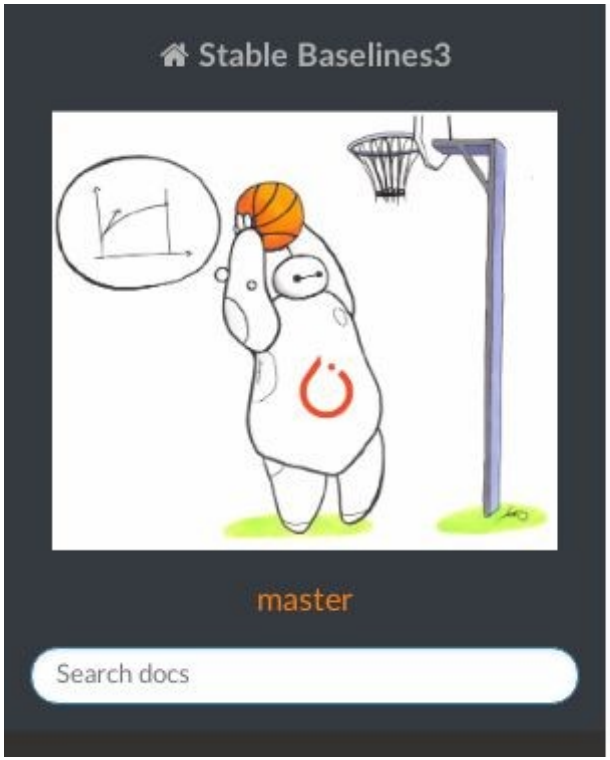
Training time is an essential part of deep learning, due to the fact that learning **takes lots of time to be able to learn** whatever process the model is being applied upon. For many situations, **GPUs are a must** as they exponentially increase the speed of training, thus training the model quicker,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



On-Policy vs Off Policy Reinforcement Learning

On-Policy Reinforcement Learning	*Off-Policy Reinforcement Learning*
Policy π *is updated with data collected by* π *itself*. We optimize the current policy π 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 already using 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 **from its own predecessor policies**. 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 ≠ Policy used for action selection
Examples: Policy Iteration, Sarsa, PPO, TRPO	Examples: Q- learning, DQN, DDQN, DDPG

Stable Baselines3 Library .1



Docs »

Welcome to Stable Baselines3 docs! - RL Baselines Made Easy

Edit on GitHub

Welcome to Stable Baselines3 docs! - RL Baselines Made Easy

Stable Baselines3 is a set of improved implementations of reinforcement learning algorithms in PyTorch. It is the next major version of Stable Baselines.

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

Stable Baselines3 Library .2

Source code for stable_baselines3.dqn.dqn

```python
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_upd
from stable_baselines3.dqn.policies import DQNPolicy


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

Source code for stable_baselines3.common.off_policy_algorithm

```python
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_pkl, save_to_pkl
from stable_baselines3.common.type_aliases import GymEnv, MaybeCallback, RolloutReturn, Schedul
from stable_baselines3.common.utils import safe_mean
from stable_baselines3.common.vec_env import VecEnv


class OffPolicyAlgorithm(BaseAlgorithm):                                          [docs]
    """
    The base for Off-Policy algorithms (ex: SAC/TD3)
```

So as we can easily understand that DQN belongs to family of Off-Policy algortihms. Indeed, we can observe that class DQN inherits from class OffPolicyAlgorithm!

Stable Baselines3 Library .3

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?

Modifications

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

Hyperparameters

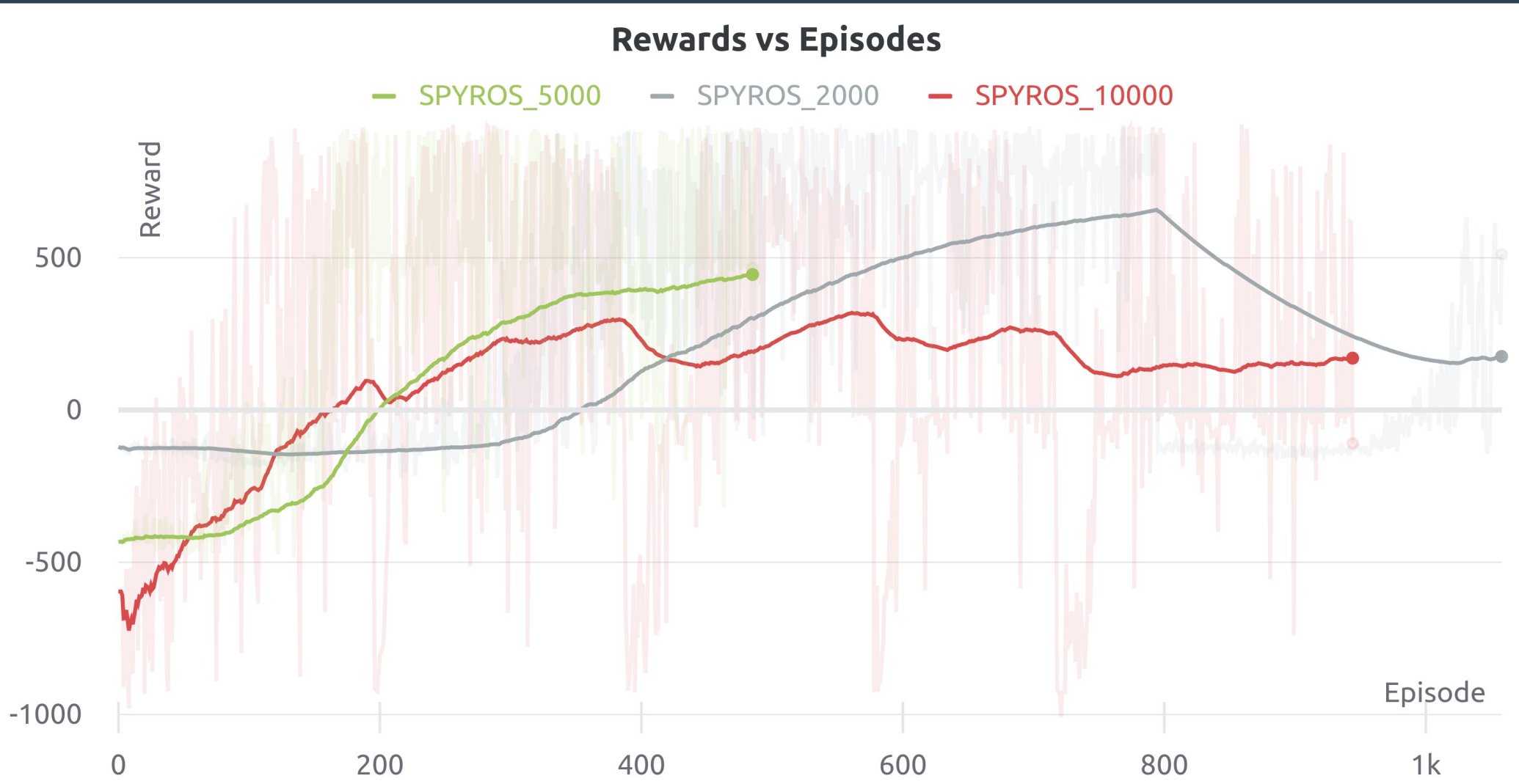
policy: **CNNPolicy**
learning_rate: **1e-4**
learning_starts: **5.000**
batch_size: **32**
gamma: **0.99**
exploration_fraction: **0.1**
exploration_initial_eps: **1.0**
exploration_final_eps = **0.05**

- So...with this DQN model we managed to try three big seperate 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!
- 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, respectivelythat 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.

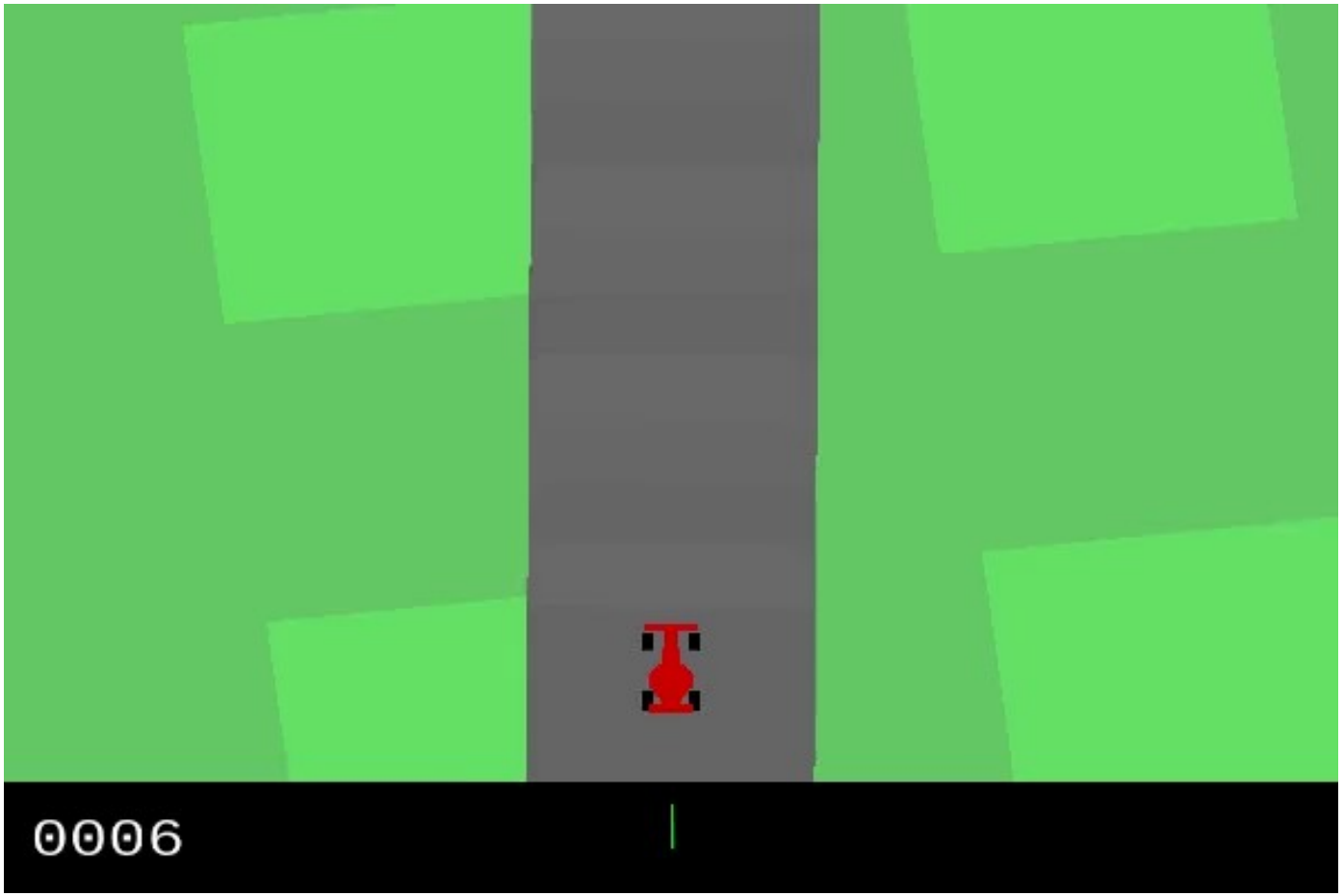
Differences of experiments

First Experiment	*Second Experiment*	*Third Experiment*
Buffer_size: 100.000, ~6GB	Buffer_size: 150.000, ~10GB	Buffer_size: 150.000, ~10GB
Maximum_steps_per_episode: 10.000 (3’20’’)	Maximum_steps_per_episode: 2.000 (40’’)	Maximum_steps_per_episode: 5.000 (1’40’’)
Episodes: 1.000	Episodes: 1.000	Episodes: 500

Experiments



Evaluation of our best DQN model!



Catastrophic Forgetting .1

- 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.

Catastrophic Forgetting .2

- ***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.

Reinforcement Learning

Policy Gradient, Actor-Critic (A2C)

Why policy gradient

- **Stochastic policies can not be learned, which can be useful in some environments in Q-learning**
- **Continuous action space**
- **It would be more beneficial to improve the strategy directly and not indireclty as Q-learning**

Why stochastic policy matters

By mapping certain states to actions, the opponent could easily exploit these patterns. Therefore a uniform random policy is optimal.

The math behind policy gradient

Policy-based reinforcement learning is an optimization problem

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

Let’s call πθ(a|s) the probability of taking action given a state s

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

However, we are not aware of the best action.
We propagete information about the reward of an action

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

On-policy exploration problem

The algorithm picks its next action based on the assumption that it has made from all the previous actions.

This may be good or bad

We need to regulate the update function

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

Policy Gradient Theorem

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

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

In order to adjust the policy, to get a particular action, we need to adjust the policy in the direction that does more of the good things and less of the bad things.

Monte-Carlo policy gradient (Reinforce)

1. Initialize the policy parameter θ at random.
2. Generate one trajectory on policy πθ: S1,A1,R2,S2,A2,…,ST
3. For t=1, 2, … , T:
4. Update policy parameters: $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$

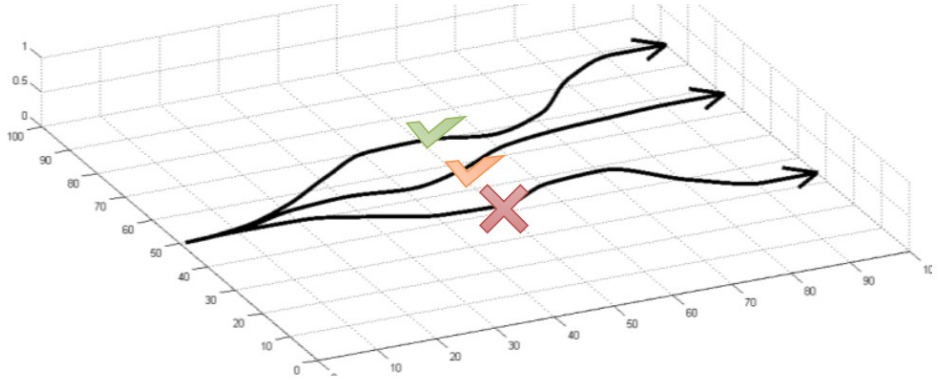
We get a set of trajectories, and update θ using the **empirical expectation**

This will be **too slow** and **unreliable**

due to **high variance** on the gradient estimates.

Why there is so much variance

When we want to compute the gradient step we typically **sample** data from multiple trajectories **based** on the **current policy π**



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

Why there is so much variance

In **supervised learning** we know that after every iteration the loss will be **less** than the previous loss.

This is **not true** in reinforcment learning and we need to solve this

Advantage Baseline

A common baseline is to subtract state-value from action-value

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

2 neural networks for the Q and V values?

Bellman optimality equation, we can use the relationship between the Q and V

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

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

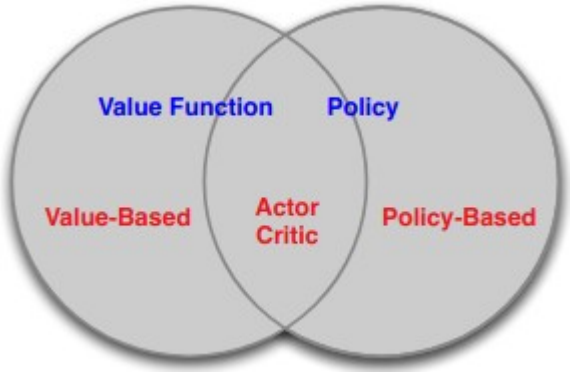
Advantage Baseline

In a nutshell this means **how much better** is to take an action **compared** to the **average general** action at the given state.

Actor-Critic model

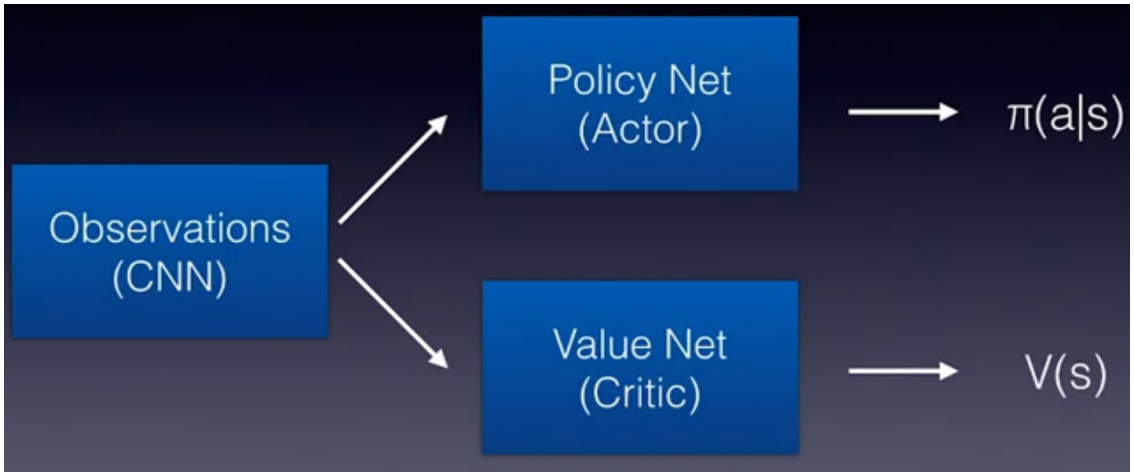
Policy model and **Value function** are **2 principal components** of the policy gradient

In addition to policy it makes a lot of sense to learn the **value function**, as learning the value function will **improve** the policy update



Actor-Critic model

- **Critic** updates the value function parameters w
- **Actor** updates the policy parameters θ for πθ(a|s), in the direction suggested by the critic



Actor-Critic model

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

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

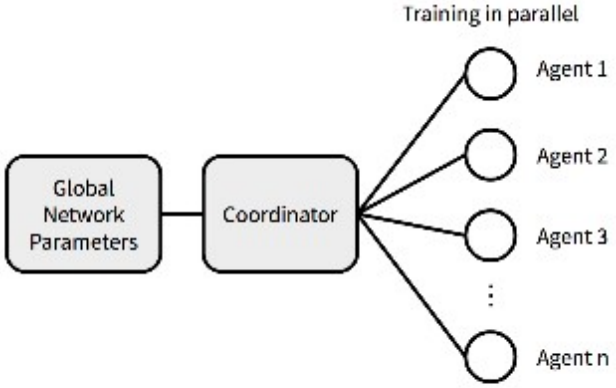
1. Initialize s,θ,w at random
2. For t=1…T
3. Pick an action A from πθ(a|s)
4. Sample reward Rt∼R(s,a) and next state s′∼π(s′|s,a)
5. Update the policy parameters:
6. Compute the correction (TD error) for action-value at the $\delta_t = r_t + \gamma Q_w(s',a') - Q_w(s,a)$ and use it to update the parameters of the action-value function: $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s,a)$
7. Update a ← a′ and s ← s′.

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.

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

It benefits from **parallel environments**



A2C (Sync)

Advantage Actor-Critic

1. Initialize s,θ,w at random
2. For t=1…T
3. Pick an action A from πθ(a|s)
4. Sample reward Rt∼R(s,a) and next state s′∼π(s′|s,a)
5. Dicrease the reward with gamma R = γR + Ri
6. Update the parameters θ and w

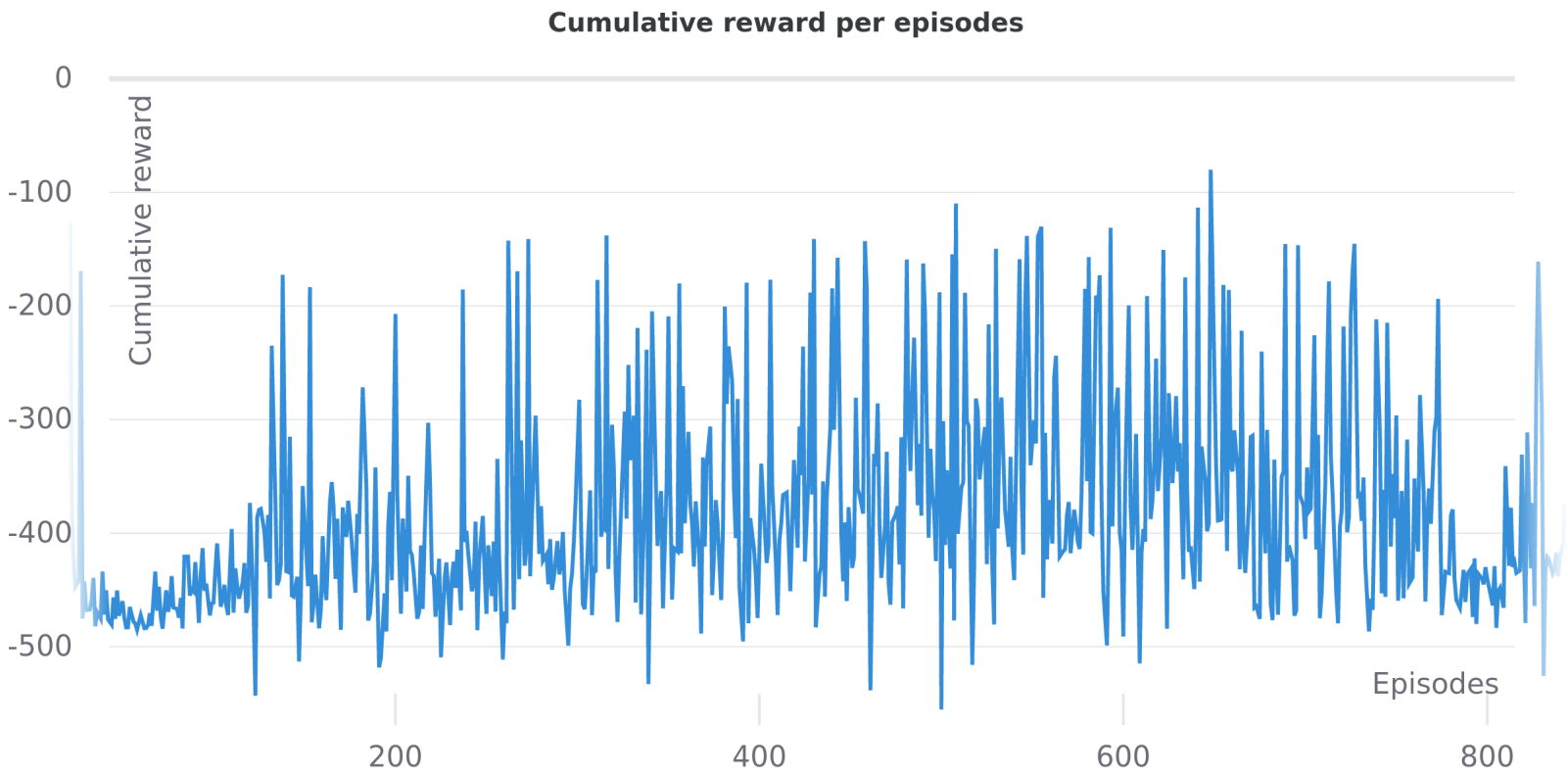
$$\theta' : d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi_{\theta'}(a_i|s_i)(R - V_{w'}(s_i));$$

$$w' : dw \leftarrow dw + 2(R - V_{w'}(s_i))\nabla_{w'}(R - V_{w'}(s_i))$$

Experiments with A2C

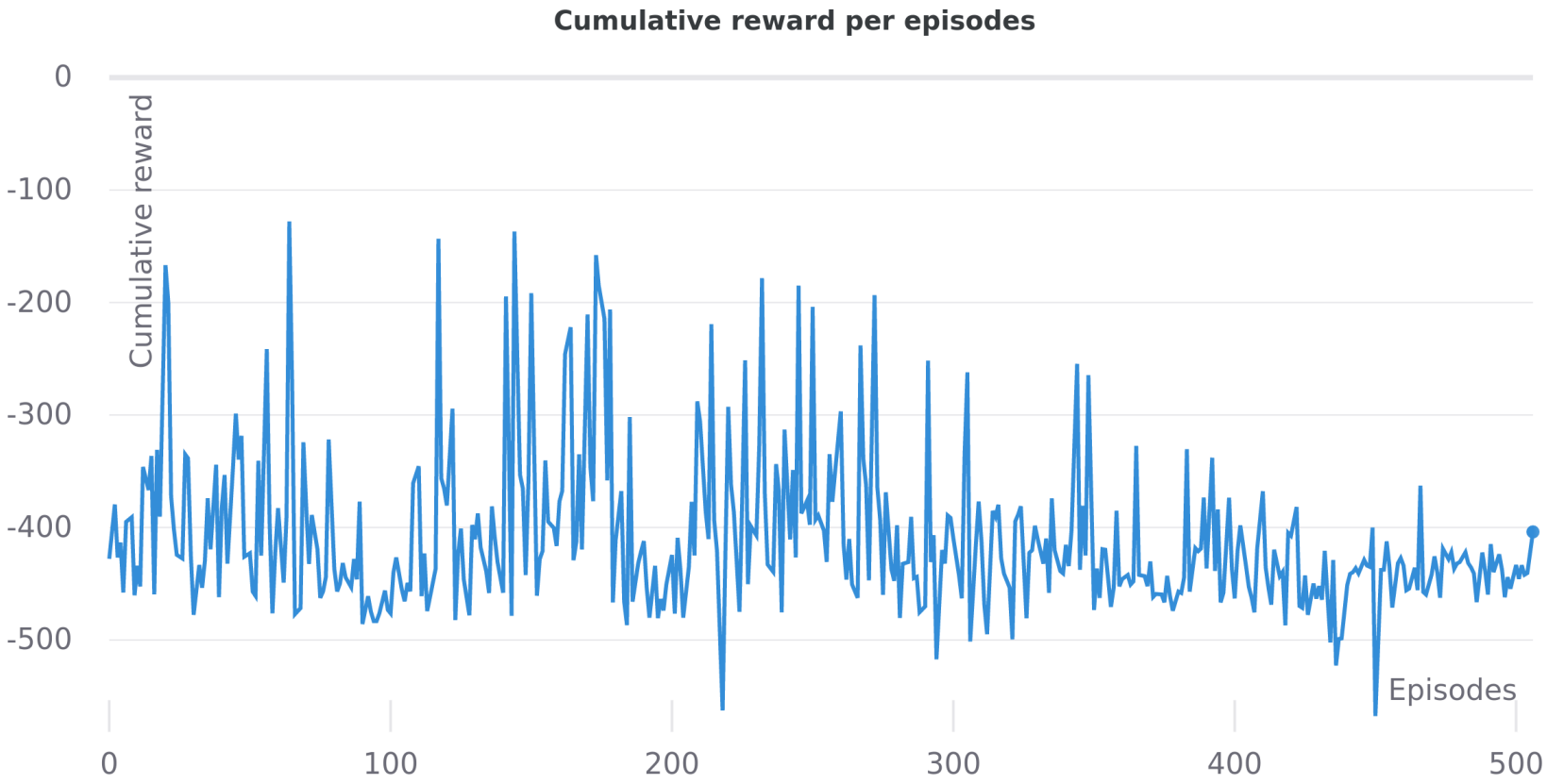
- We used the Open Source Library, **Stable Baselines** instead of Stable Baselines 3.
- The most important hyperparameter, n_step size we used 32 and 16
- Training for 10 hours with only one environment due to hardware limitations in google colab

1st Experiment



Here is our first attempt with n_step size 16

2nd Experiment



Here is our next experiment with n_step size 32

Why not so good

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 **hardware limitations** of our training.

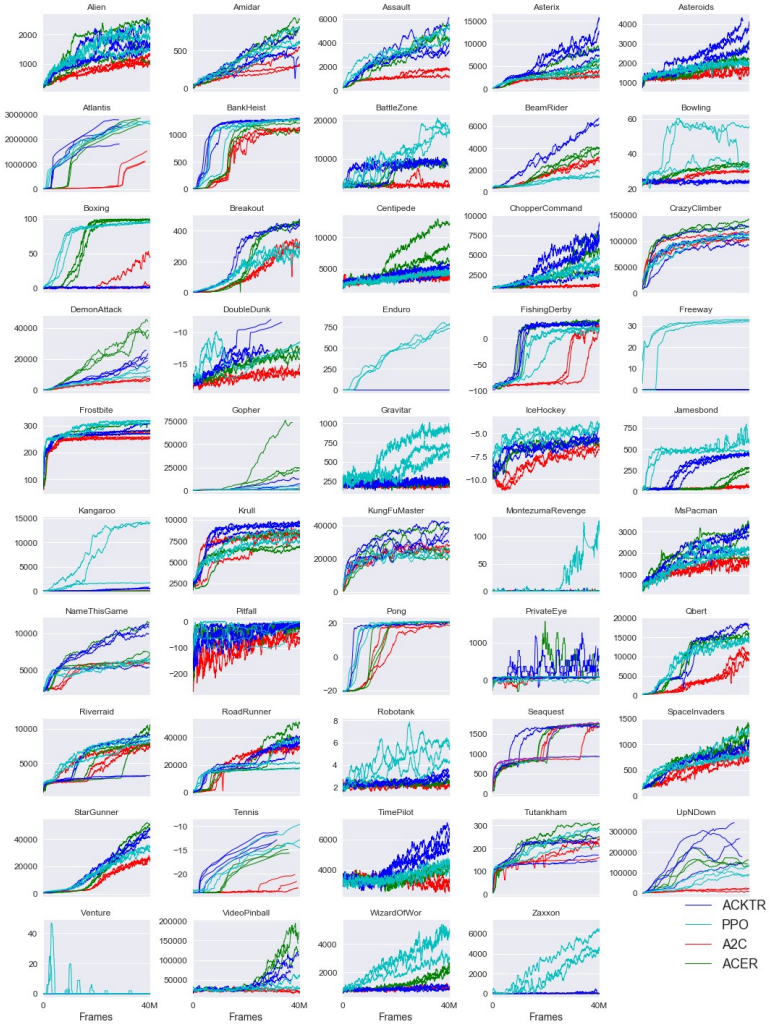
Why not so good

In the paper they run 16 parallel environments on a 16 core or more CPU

Google Colab only provides 1 CPU core!

Why not so good

A2C may not perform well in this environment



Reinforcement Learning

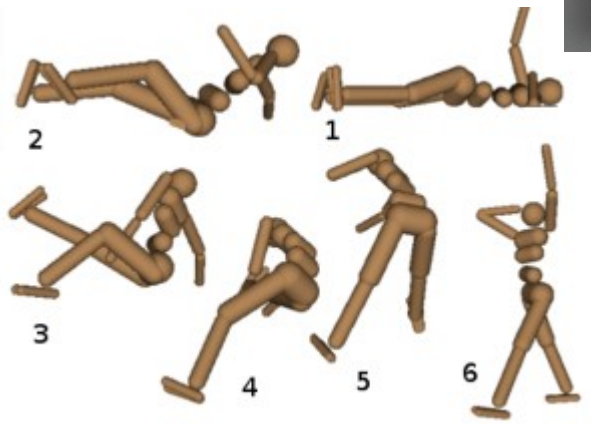
Proximal Policy Optimization (PPO)

A quote from OpenAI on PPO:

"Proximal Policy Optimization (PPO), which perform comparably or better than state-of-the-art approaches while being much simpler to implement and tune."

So what is PPO?

PPO is an effective reinforcement learning approach, that belongs to Policy Gradient Methods, and solves mostly control tasks, video games, 3D locomotion and Go.



Why would anyone choose PPO?

Solves problems like:

1. *Unstable Policy Update*
2. *Data Inefficiency*

Advantages:

- *Ease of Implementation*
- *Sample Efficiency*
- *Ease of Tune*

Why would anyone choose PPO?

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.

Background 1: Policy Gradient Methods

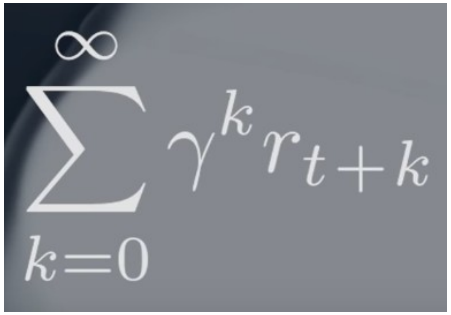
General ***Policy Optimization Methods*** usually start by defining the policy gradient loss, as the expectation over the log of the policy actions multiplied by an estimate of the advantage function.

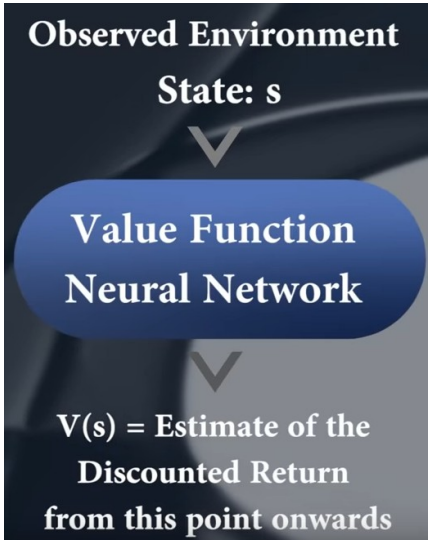
$$L^{PG}(\theta) = \hat{\mathbb{E}}_t \left[\log \pi_\theta(a_t \mid s_t) \hat{A}_t \right].$$

The first term π_θ 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.

Advantage Function

In order to compute the advantage we need two things: we need to ***discounted the sum of rewards*** and a ***Baseline 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?

Background 2: Trust Region Methods

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 $\pi_{\theta\ \text{old}}$.

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

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. So, in a sense, we just want to stick close to the region, where we know everything works fine.

Clipped Surrogate Objective

This is the central objective function that is used in PPO:

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

- **r(θ)** is just a probability ratio between the new updated policy outputs and the outputs of the previous old version of the policy network
    - **r(θ) > 1**, if the action is more likely now than the old version of the policy
    - **0 < r(θ) < 1**, if the action is less likely now than it was before the last gradient step
- $\hat{\mathbf{A}}_t$ is the Advantage Function

Clipped Surrogate Objective

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.

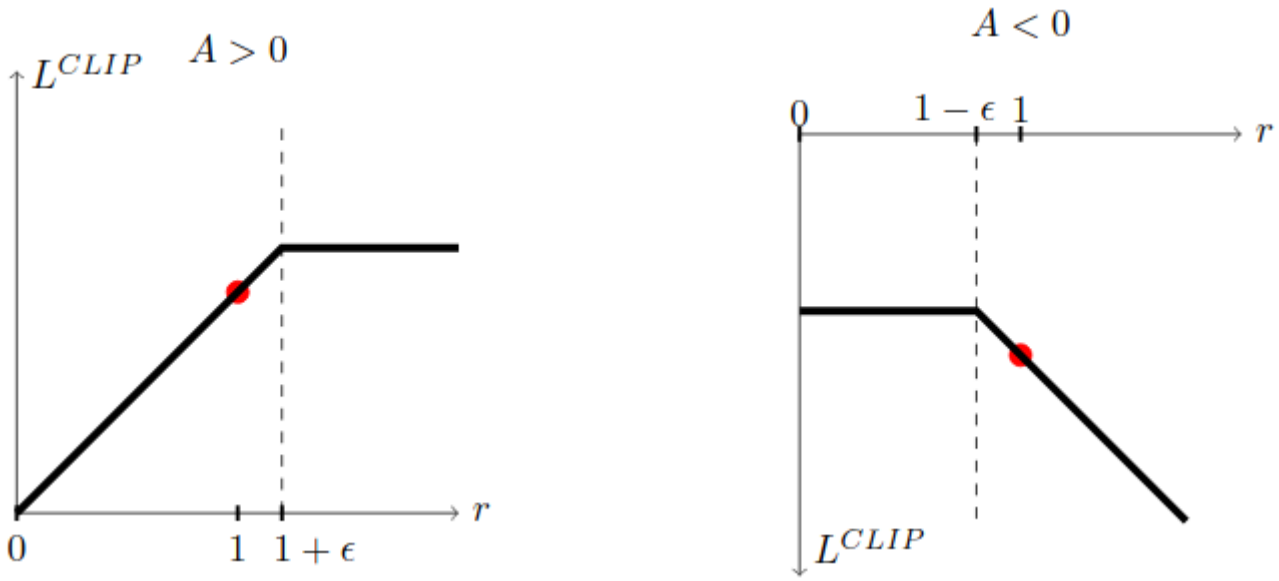
$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right]$$

The **first** of these terms is **r(θ)** multiplied by the **advantage estimate**.

The second term is very similar to the first one, except that it contains a truncated version of this r(θ) ratio, by applying a **clipping** operation between **1- ε** and **1+ ε**, where epsilon is usually something like **0.2**. Then, the min operator is applied to the two terms to get the final result**.**

Clipped Surrogate Objective

The advantage estimate can be both positive and negative and this changes the effect of the min operator.



If the probability ratio between the new policy and the old policy falls outside the range (1 — ε) and (1 + ε), the advantage function will be clipped. Effectively, this discourages large policy change if it is outside our comfortable zone.

Clipped Surrogate Objective

TRPO Objective* vs *PPO Objective:

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

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

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.**

Final Loss Function

The final loss function, that is used to train an agent, is the sum of this clips PPO objective, plus two additional terms.

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

- ***Loss function***, is basically in charge of updating the baseline network.
- ***Entropy term*** is responsible of making sure that our agent does enough exploration during training.
- 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

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

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.

Algorithm

Algorithm 1 PPO, Actor-Critic Style

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

Thoughts

PPO adds a soft constraint that can be optimized by a first-order optimizer. We may make some bad decisions once a while but it strikes a good balance on the speed of the optimization. Experimental results prove that this kind of balance achieves the ***best performance with the most simplicity.***

Experiments with PPO

- We used the Open Source Library, **Stable Baselines** instead of Stable Baselines 3.

- We used the **PPO2** model, instead of PPO1, as it is the implementation of OpenAI made for GPU.

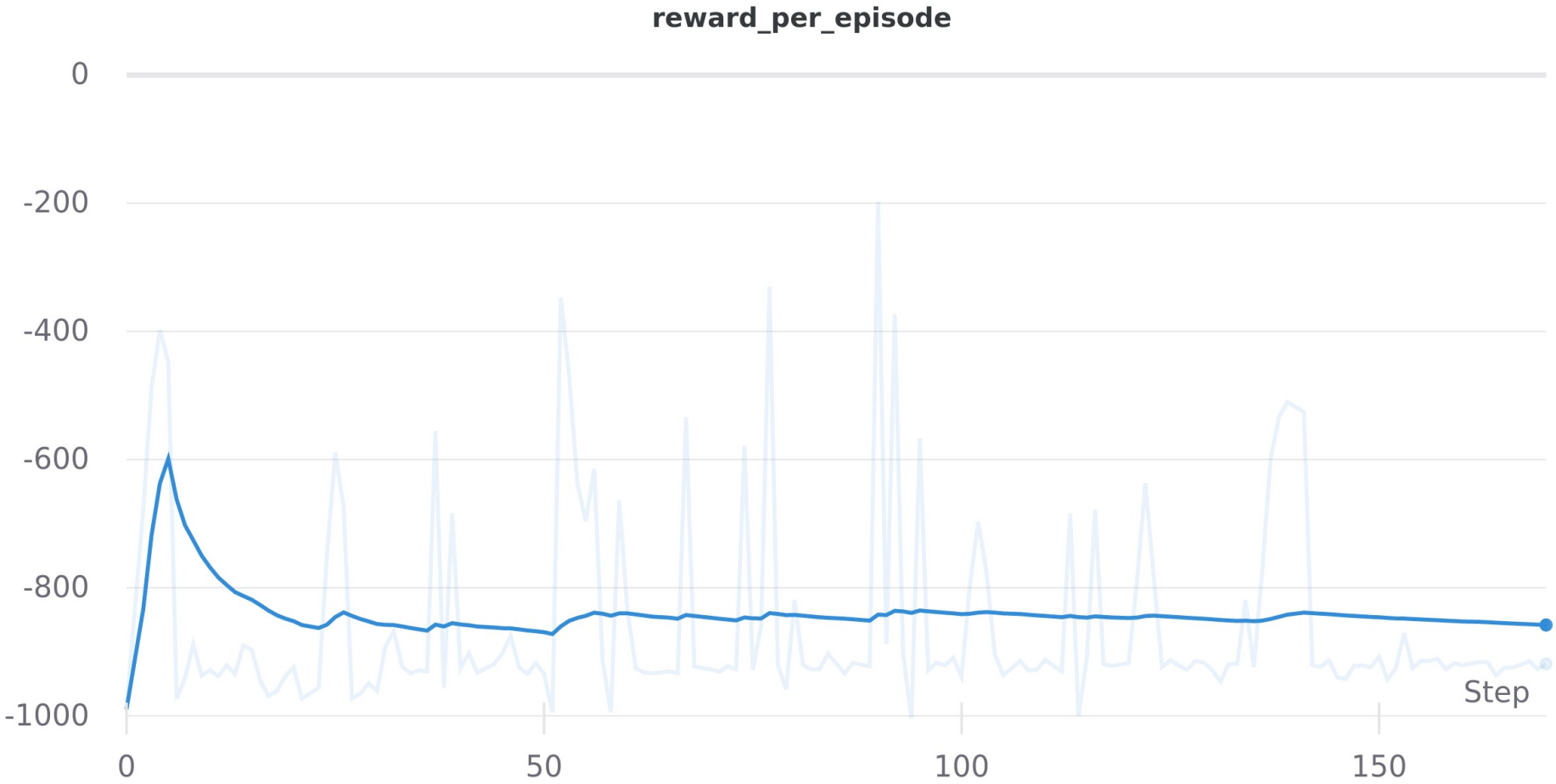
- PPO2 version is **3 times faster** than the implementation of PPO1.

Hyperparameters

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

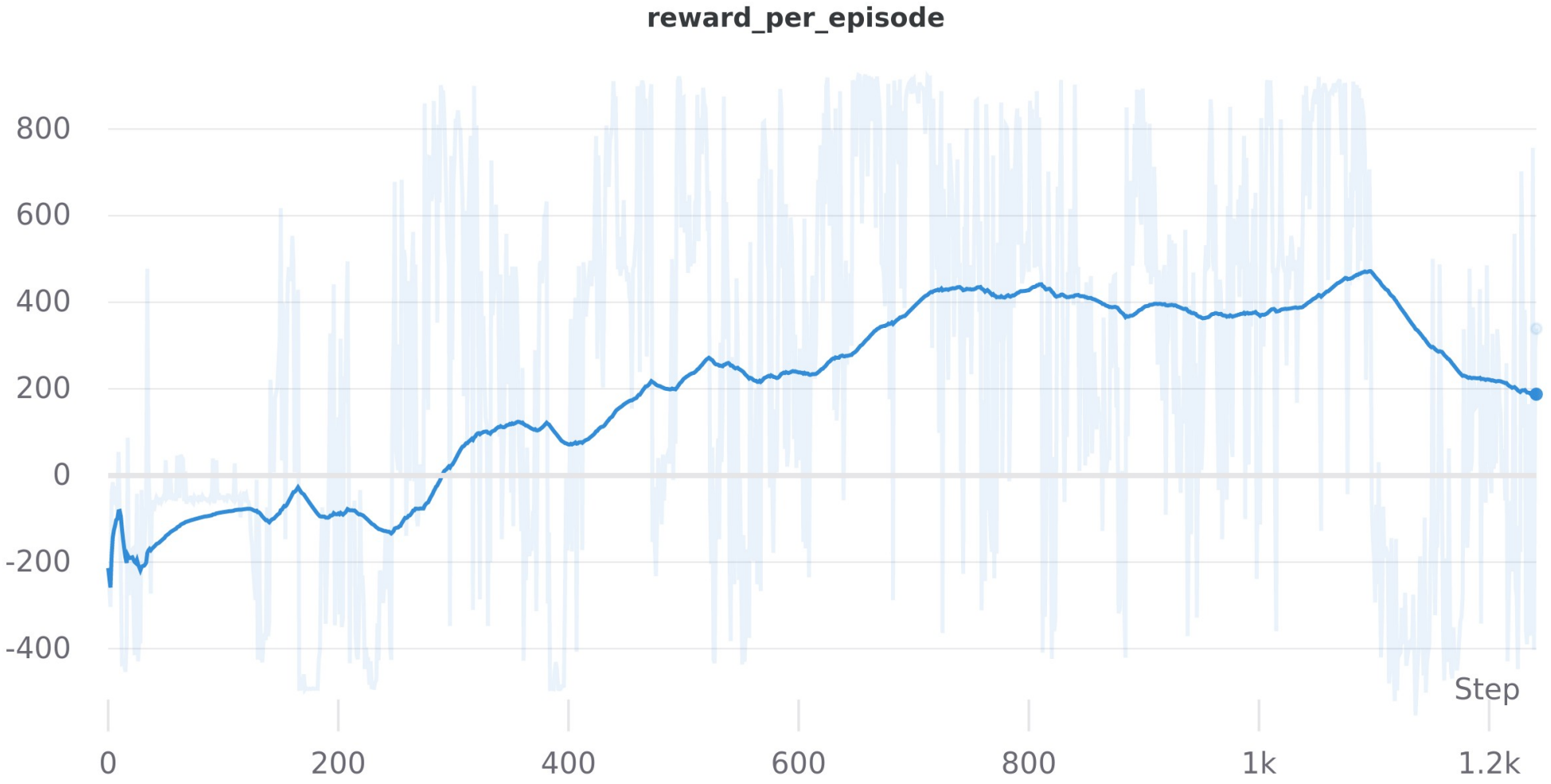
- ***gamma*** = 0.99
- ***n_steps*** = *64*
- **ent_coef** = 0.01
- **learning_rate** = 0.00025
- ***vf_coef*** = *0.5*
- ***max_grad_norm*** = 0.5
- ***lam*** = 0.95
- ***nminibatches*** = 4
- ***noptepochs*** = 4
- ***cliprange*** = 0.2

1st Experiment



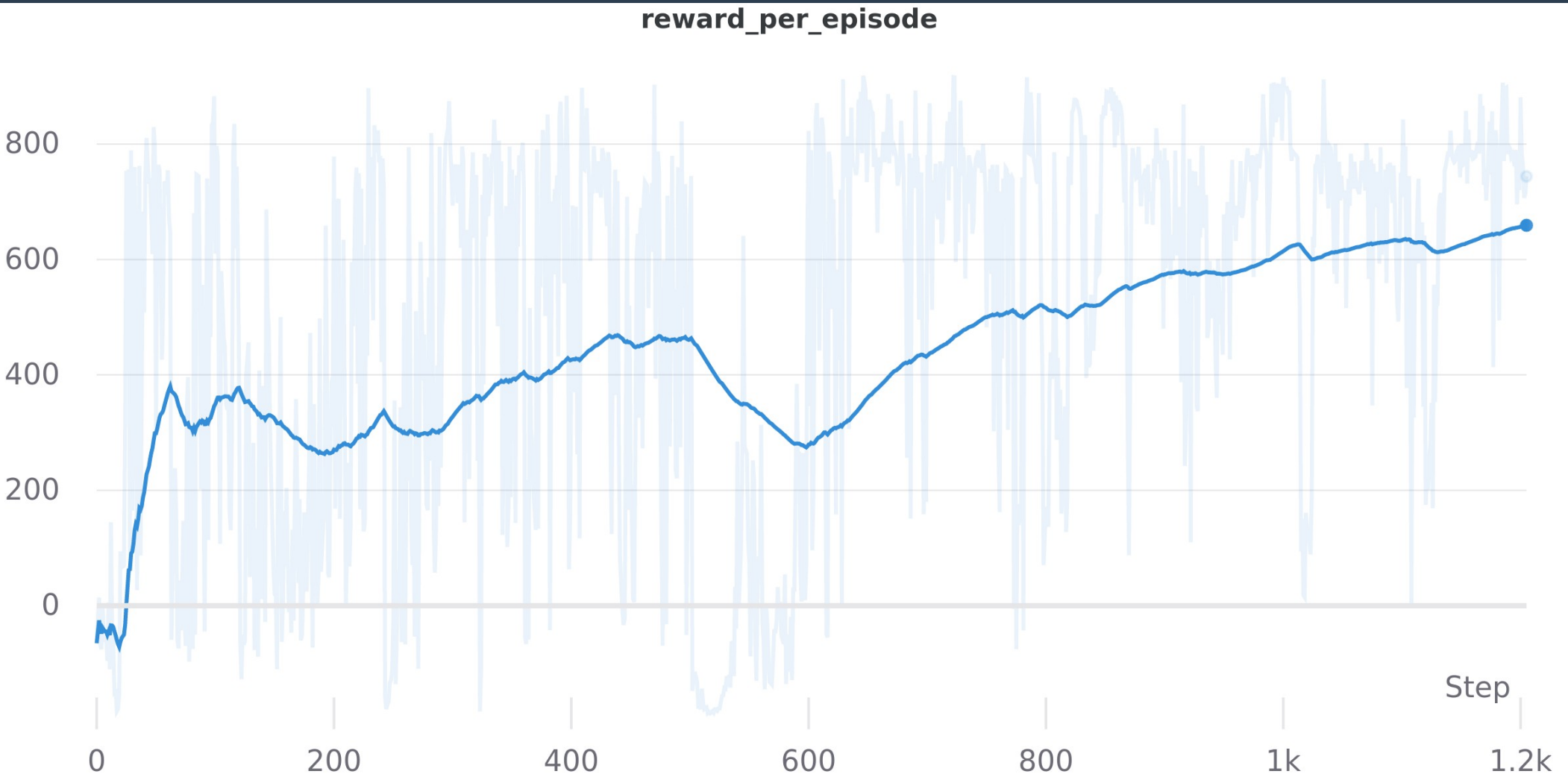
Here is our first attempt with 10.000 timesteps and 150 episodes

Best Experiment



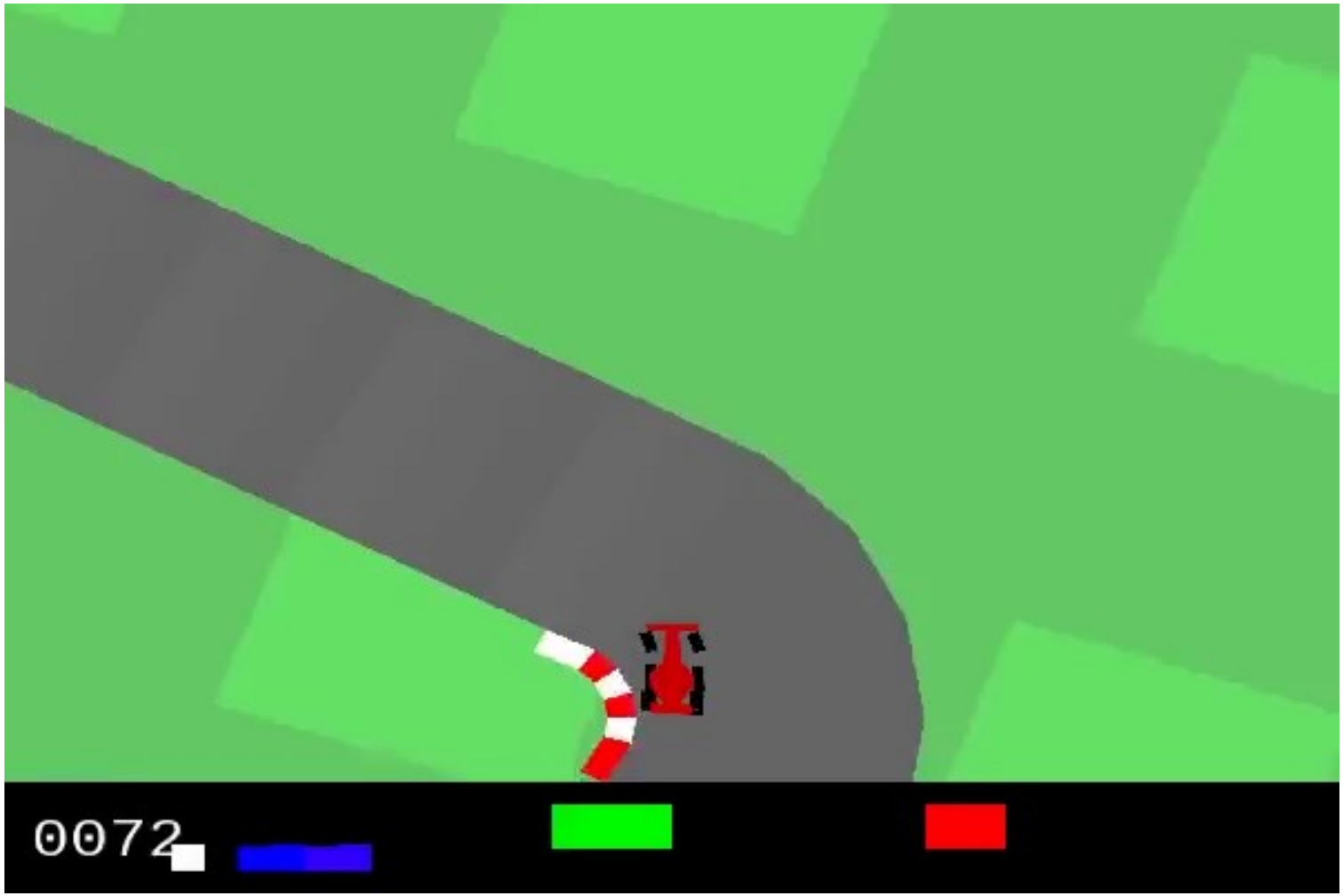
Here is our best attempt with 5000 timesteps and 1.2k episodes

3rd Experiment

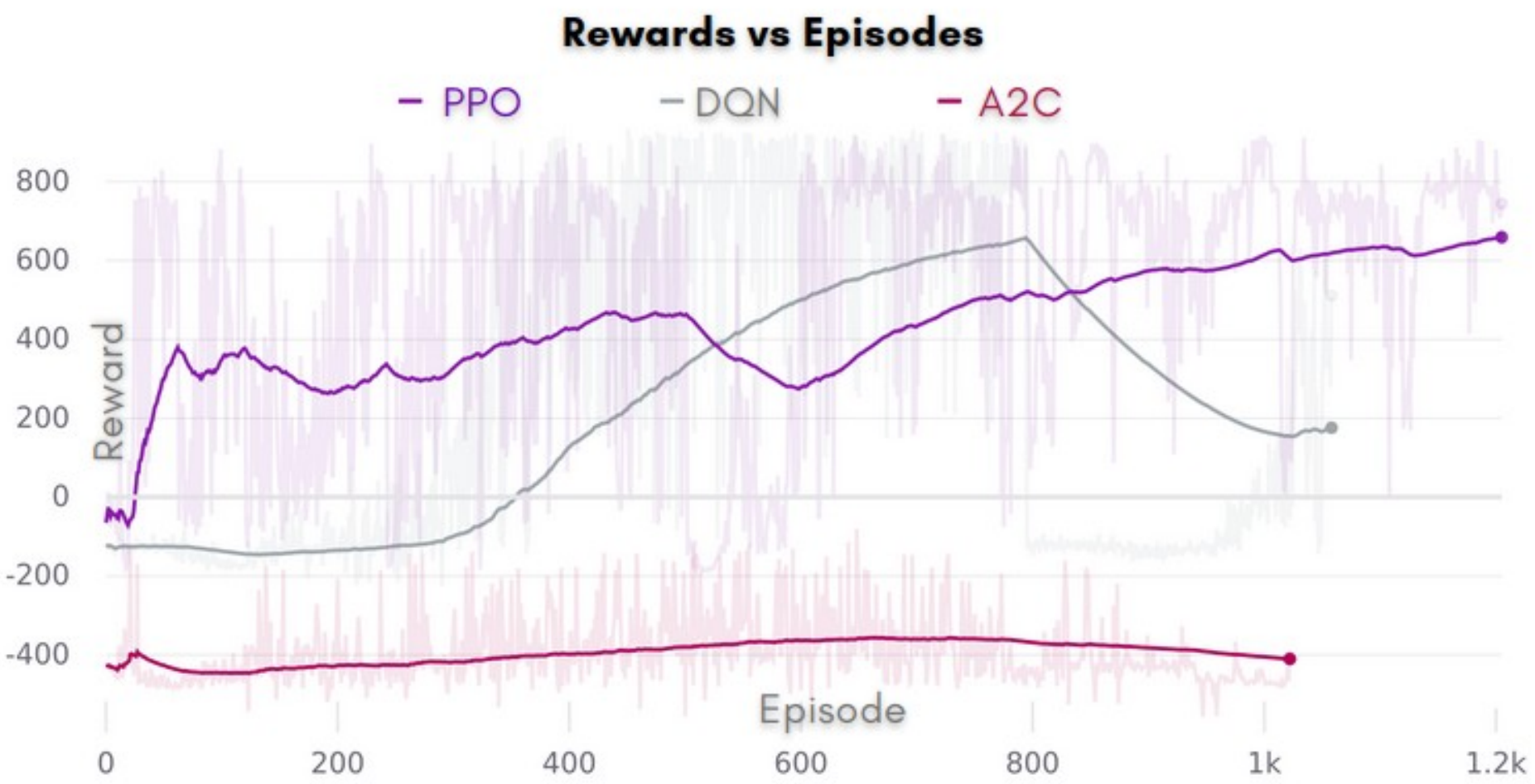


Here is our latest attempt with 2000 timesteps and 1.2k episodes

Evaluation of PPO model’s performance



Conclusions



RL Agents, indeed, learned on their own!