Semestral Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science an Engineering

Hyperparameter tuning of the PPO algorithm for OpenAI's CarRacing

Vojtěch Sýkora

Supervisor: doc. RNDr. Lukáš Chrpa, Ph.D.
Field of study: Open Informatics
Subfield: Artificial Intelligence and Computer Science
January 2023

Contents

Figures

Chapter 1

Introduction

In recent years, Deep Reinforcement Learning has been gaining popularity while being applied to a wide range of complex problems. From playing board games and computer games to teaching humanoid robots to walk, this field has achieved astounding results. Hyperparameter tuning is a critical step in the development of deep reinforcement learning algorithms, as it can significantly affect the performance of the algorithm. In this paper, we will investigate the effects each hyperparameter has on learning with the Proximal Policy Optimization algorithm (PPO) [SWD+17].

PPO has recently gained popularity thanks to its state-of-the-art results and wide range of use cases. We decided it would be the ideal choice for teaching an artificial agent how to drive. Fortunately, OpenAI, one of the leading AI research groups in the world, created the perfect environment for us. Driving a car is a massive task, so it is better to begin with a 2D-controlled environment without all of the moving distractions facing us on the roads.

The contribution this work aims to achieve is exploring each hyperparameter in depth to help in future research easily narrow down the optimal numbers to train on.

Chapter 2

Car Racing Environment

For training agents using Reinforcement Learning, it is recommended to create an environment with which the agent can communicate without human input. We chose a premade environment from the Python library *gym* simulating a car driving on a track in 2D space. This environment intrigued us because of its similarity to real life with its usage of physics and a continuous action space.

2.1 Gym Library

Gym is an open-source Python library for developing and comparing Reinforcement Learning algorithms by providing a standard API to communicate between learning algorithms and environments, as well as a standard set of environments compliant with that API. Since its release, Gym's API has become the field standard for doing this. [BCP$^+$16]. It also implements the usage of parallel/vectorized environments which speed up the training. Its documentation can be found on `https://www.gymlibrary.dev/`.

2.2 Description

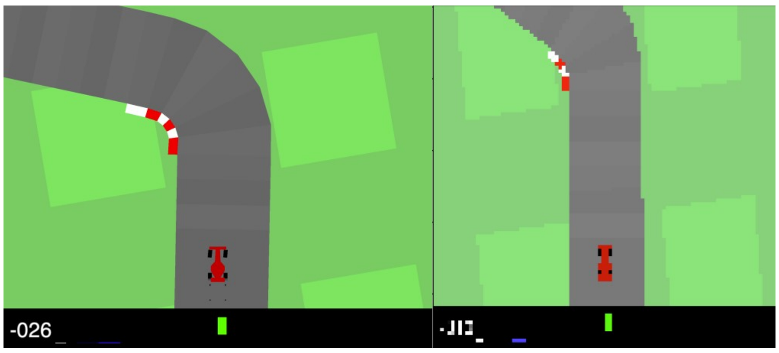


Figure 2.1: Left: Render for humans, Right: 96x96 render for agents

Our chosen environment **CarRacing-v2** is a part of Box2D environments based on real-life physics. A random race track is generated for each episode which can be observed by the agent as an RGB image of size 96x96, which represents a top-down view of the racetrack with the car centered.

Some indicators are shown at the bottom of the window, along with the state RGB buffer. From left to right: actual speed, four ABS sensors, steering wheel position, and gyroscope. It is a powerful rear-wheel drive car which makes it easy to start uncontrollably drifting.

2.2.1 Actions

- If *continuous*: There are 3 actions: steering (-1 is full left, +1 is full right), throttle, and breaking.
- If *discrete*: There are 5 actions: do nothing, steer left, steer right, throttle, brake.

For this project, we used the more challenging continuous action space because it better represents the real world.

2.2.2 Rewards

The reward is -0.1 for every frame and +1000/N for every track tile visited, where N is the total number of tiles visited in the track. For example, if you have finished in 420 frames, your reward is 1000 - 0.1*420 = 958 points. If the car drives completely off the field, it gets -100. This means the only policy for gaining a good positive score is staying on the track while driving fast. The accepted score for solving this environment is 900.

2.2.3 Difficulties

There are an enormous number of different states because the track is generated randomly, so if the agent takes an action, it cannot be known with certainty which state will follow. If the state could be any image of the given size, the number of possible states would be $256^{3*96*96}$ for RGB colors from 0-255, however, they use only some colors, and the image needs to look like a track. Nevertheless, the number would still be enormous.

The last issue is that the rendered view for a human has much more pixels than the observation returned from the environment, as seen in figure 2.1. This negatively affects the training accuracy, but it speeds it up tremendously.

Chapter 3

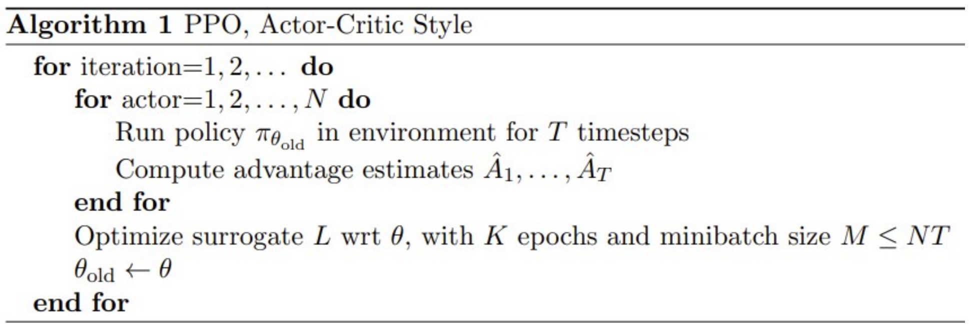
Proximal Policy Optimization (PPO)

The field of Deep Reinforcement Learning has experienced immense growth in recent years, and one of the stable baselines has become the Proximal Policy Optimization algorithm (PPO) [SWD$^+$17]. PPO learns a policy that minimizes its loss and therefore maximizes its reward. It is part of the Policy Gradient algorithms and is a modified superior version of the Trust Region Policy Optimization algorithm (TRPO) [SLM$^+$15]. We will only describe PPO and its objective function; however, for a full understanding of where PPO came from and how it is better than previous algorithms, we recommend the original PPO paper.

The proximal part of the algorithm refers to the fact that the policy is regularized to encourage it to stay close to the previous policy, which helps to improve the stability of the learning process. This regularization can also help to avoid overfitting and improve the overall performance of the algorithm.

3.1 Policy Gradient Method

This algorithm falls in the family of Policy Gradient methods. These algorithms learn online, which is the main difference between them and Deep-Q Networks (DQN). Policy gradient methods don't store past experiences in a replay buffer; instead, they learn directly from what the agent encounters. Once the batch of experiences is used, it is discarded. This is being called *less sample efficient*. DQN uses their experiences multiple times combined with experiences from other episodes, while Policy Gradient methods use them from only the current episode and then delete them. Whereas standard Policy Gradient methods perform one gradient update per data sample, PPO has a novel objective function that enables multiple epochs of minibatch updates, which will be explored in a subsection 4.1.2 of the hyperparameter Tuning chapter.



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 $\theta$, with $K$ epochs and minibatch size $M \leq NT$
    $\theta_{\text{old}} \leftarrow \theta$
end for
```

Figure 3.1: PPO algorithm [SWD+17]

There are two alternating threads in PPO, the first of which (inner for loop in 3.1) collects experiences from interacting with the environment. If the hardware allows it, multiple actors can collect experiences at once. The second thread then runs gradient descent on the policy network using the saved experiences.

This division of labour can enable large-scale training with hundreds of CPU workers generating experiences and a couple of powerful GPUs learning from these experiences.

3.2 Objective Function

The objective function in the case of PPO is the loss function the model aims to minimize. Before we get to the objective function, let us define some useful formulae. Let $r_t(\theta)$ denote the probability ratio (also called the likelihood ratio) of new to old policy such that $r(\theta_{old}) = 1$.

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{3.1}$$

Where π is the policy with θ parameters (This will be, in our case, a Deep Neural Network.). a_t is the action to be chosen, and s_t is the current state.

TRPO maximizes the surrogate objective, which can be described as a conservative policy iteration.

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t \left[r_t(\theta)\hat{A}_t \right] \tag{3.2}$$

Where $\hat{A}_t$ is an estimate of the advantage function.

This can be calculated by subtracting the baseline estimate (the value output from the critic part of our Neural Network) from the discounted sum of rewards. In our case, we used a truncated version of the generalized advantage estimation, which is defined as follows

$$\begin{aligned} \hat{A}_t &= \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1} \\ \text{where } \delta_t &= r_t + \gamma V_\theta(s_{t+1}) - V_\theta(s_t) \\ t &\in [0, T] \end{aligned} \tag{3.3}$$

where T is our hyperparameter horizon and $V_\theta(s_t)$ is our first output of the Deep Neural Network, also called the Critic (further information in section 3.3). λ is one of our hyperparameters called *GAE lambda* (generalized advantage estimation).

PPO clips the surrogate objective to prevent unreasonably large updates. The following is the clipped loss which is the main part of PPOs objective function.

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right] \tag{3.4}$$

Here ϵ is a hyperparameter which we tune in subsection 4.2.1 under the name *clipping range.*

As can be seen in figure 3.2 when the advantage is positive (meaning the outcome we obtained was better than expected), we clip actions with a high reward to not overly change the policy in one update. When the advantage is negative, our outcome is worse than expected, and we want to undo our previous step by a proportional amount. This represents the linear part of the graph.

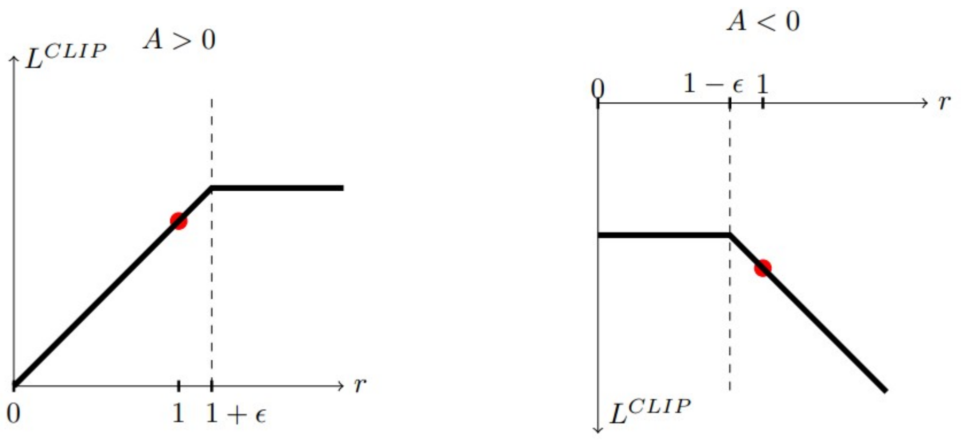


Figure 3.2: Clipping [SWD+17]

Finally, the full equation for the objective function of PPO can be defined as follows

$$L^{PPO}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right] \tag{3.5}$$

where c_1, c_2 are the value function coefficient and entropy coefficient. S denotes the entropy which we obtain from a Beta probability distribution created using our other 2 outputs of the neural network. These other two outputs are called the Actor (further information in section 3.3). L_t^{VF} is the predicted value (from our Neural Network) minus the target value squared.

$$L_t^{VF} = (V_\theta(s_t) - V_t^{\text{target}})^2 \tag{3.6}$$

3.3 Deep Neural Network Structure

The PPO algorithm falls in the family of Actor-Critic methods, which use a Deep Neural Network structure [RN10] oriented around having an Actor network and a Critic network. Our network has both the Actor and the Critic in a single network because they share the layers processing the observed game state (RGB image 96x96 pixels).

The Critic is responsible for estimating the expected value of a state. The value is the sum of all rewards it expects to receive in the future. This value is a scalar, and we obtain it from a single dense layer with a linear activation because it can be any number. This layer can be seen on the bottom right of figure 3.3.

The Actor is responsible for generating a probability value for each possible action from the observed state. Since we used a continuous action space, our probability had to be expressed using a probability distribution. For this experiment, we used the Beta probability distribution [16], whose probability density function is defined as

$$f(x, \alpha, \beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)} \tag{3.7}$$

We obtain the alpha and beta variables from the bottom right dense layers with a slight readjusting using the Lambda layer.

Both Actor and Critic share a convolution network that processes the inputted observation to extract some features and lower the number of total parameters. The current number of parameters of the whole network is 632 999, all being trainable.

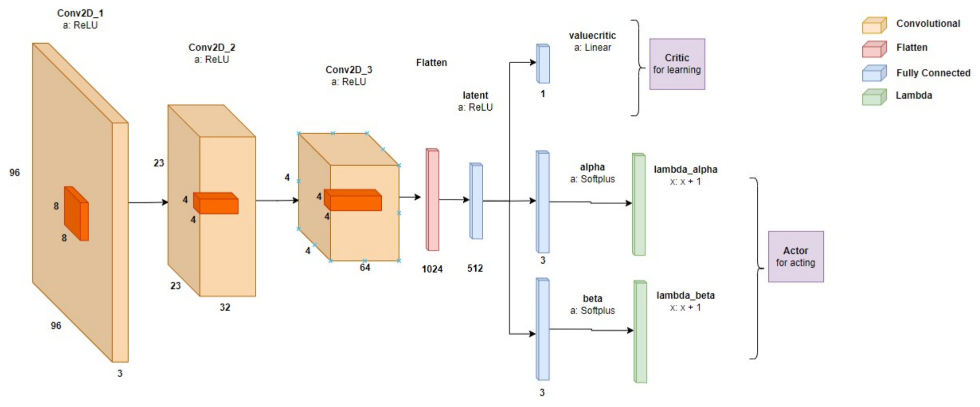


Figure 3.3: Our Deep Convolutional Neural Network

Chapter 4

Hyperparameter Tuning

Each deep reinforcement learning program has one issue, which is called hyperparameters. One can perfectly write the algorithm, however, for it to be effective, a good set of hyperparameters has to be found. Where to even begin? Is random initialization a good idea?

Since we wanted to study how the hyperparameters change the rewards, we decided to initialize them using our previous knowledge [NMK22] combined with the knowledge gained from the original PPO paper [SWD+17]. We wanted our graphs to focus solely on the studied parameter without being corrupted because of a different parameter being set poorly.

Our initial hyperparameters were

$$\begin{aligned} \text{horizon} &= 128 \\ \text{mini-batch size} &= 256 \\ \text{epochs per episode} &= 3 \\ \text{gamma} &= 0.99 \\ \text{clipping range} &= 0.2 \\ \text{gae lambda} &= 0.95 \\ \text{value function coefficient} &= 1 \\ \text{entropy coefficient} &= 0.01 \\ \text{learning rate} &= 2.5e\text{–}4 \end{aligned}$$

This chapter will include graphs, all of which have mean cumulative return from the last 300 runs on the y-axis, and the number of timesteps on the x-axis. All graphs will be generated using the TensorFlow library for Python and visualized using Tensorboard, which is included in TensorFlow [AAB+15].

4.1 Experience Collection

We divided our experimentation into four blocks with a similar focus. The first one is hyperparameters influencing the experience collection. These control how long we train, how we divide the obtained experiences, and how many times we learn from one episode.

4.1.1 Horizon

Horizon, in our case, is the number of steps in each episode. Since we are running multiple environments in parallel, we don't end when the environment says it is done. We need to hardcode how many steps we want to explore in each episode. If one of the parallel environments ends its run, it automatically resets.

With this information, we can deduce that at the start of training, a lower horizon would result in the car exploring only the beginning of the track in one run while a large horizon could even explore some turns. This would mean that a car with a lower horizon learns the track in smaller sections mastering each before moving further in the track. A large horizon would result in the car exploring exponentially more space which would lengthen its initial training time. Both these facts can be seen in the graph 4.1 in the first 400k steps.

The second interesting occurrence which can be explained is the initial flat graph followed by a steep rise. This could be because the first turn is quite close to the beginning, which results in the initial random exploration ending mostly outside of the track until it overcomes a threshold and learns to drive through turns. We can see a slight increase in the slope at the beginning because of the initial short straight track; however, to successfully navigate through the first turn, the car needs to fully learn how to turn. This occurrence, in particular, will repeat in every graph because of the reasons stated in this paragraph.

The final choice of horizon won 2250 over 2000 because its curve rises earlier and is relatively stable on the rest of the graph.

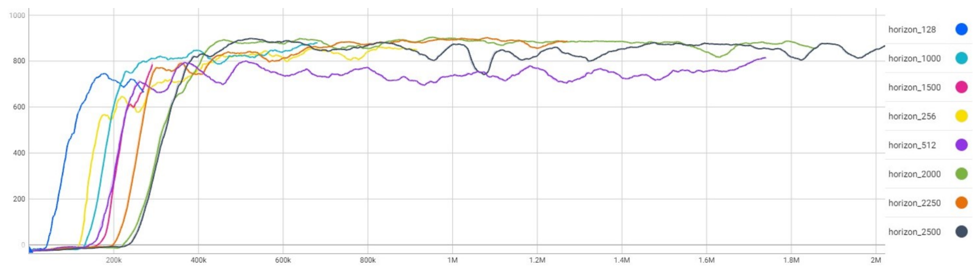


Figure 4.1: Horizon: mean score per step

4.1.2 Mini-batch Size

The experience we collect during the episode can grow to a number in the higher tens of thousands, and even with great hardware, we cannot optimize our Neural Network with all of the experiences at once. This is where mini-batches come in. We optimize using gradient descent using a single batch of experiences at one time. Smaller batch sizes are often preferred because they are noisy, which offers a regularizing effect and lowers generalization error. And even though Deep Neural Networks are often trained on GPU clusters, it is still much easier to fit only one small batch in the memory.

The graphs from the training show some interesting features. Firstly, the most used and praised mini-batch size of 32 seems to fluctuate quite a lot which is quite the mystery. It cannot be said that it was a strike of bad luck on the random generation part of the tracks because the largest drop occurred during 200k steps, which is much larger than usual drops.

In the end, we ran the experiment again with mini-batch sizes 32,64,128,1024, and the most consistent one with the best score came out to be the largest. For now, we are changing the mini-batch size to 1024.

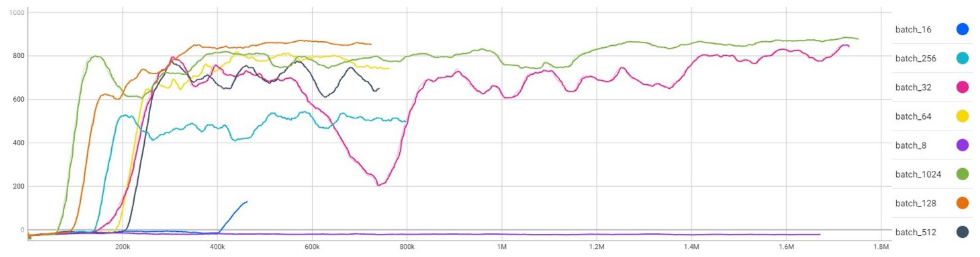


Figure 4.2: Mini-batch size: mean score per step

4.1.3 Epochs

An epoch refers to one cycle through the full training dataset. Our training dataset is the experience buffer from one episode. It is recommended to train with more than one epoch, each having randomized mini-batches because it leads to better generalization with not yet seen batches of experiences. In theory, the more epochs we have, the more information can be learned from one experience buffer, however, in practise, it is not the best strategy. Too many epochs can cause overfitting, which may, in extreme cases, lead to almost no increase in rewards, as can be seen in graph 4.3 on curves showing 15–30 epochs. The yellow curve of 10 epochs supports this argument, being slightly overfitted, but still managing to learn.

The results show that smaller epoch numbers are much more desirable and any number from 3 to 5 would be a good choice. We decided on 3.

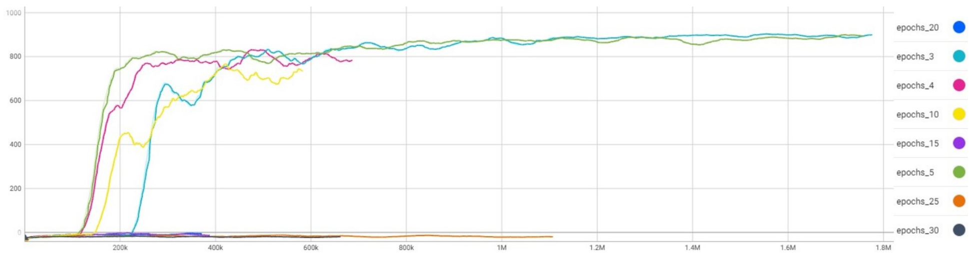


Figure 4.3: Epochs: mean score per step

4.2 Policy Updating

In this section, we focus on the hyperparameters influencing how we update the old policy. If the policy is updated in too large a step, policy performance can collapse drastically and never recover. PPO uses a clipped surrogate loss function to keep the update in a safe range so that the new policy is not drastically different from the old policy.

4.2.1 Clipping Range

Clipping range is the range where we deem the surrogate objective acceptable. It is used to ensure that the policy update is not excessively large. A more detailed description was in section 3.2. Shortly, the higher the clipping range, the larger the policy update can be done, which could result in a drastic change in the policy. To keep the policy stable, a smaller number is often used.

In our case, we can see that none of the tested numbers ended in an unsatisfactory result which is quite interesting given that the recommended clipping range is between 0.1 and 0.2. What can be seen in figure 4.4 on the orange curve (0.35) is that in the beginning, there were large fluctuations, which strengthens our point that a large clipping range may experience larger updates.

To make the graph more readable, we only left the best curves, corresponding to 0.15, 0.2, and 0.3. Now we can see that the larger fluctuations happened also with the black (0.3) curve, again reinforcing the argument above. The final two curves are almost identical, so our choice was 0.15, which is the average of the recommended ranges.

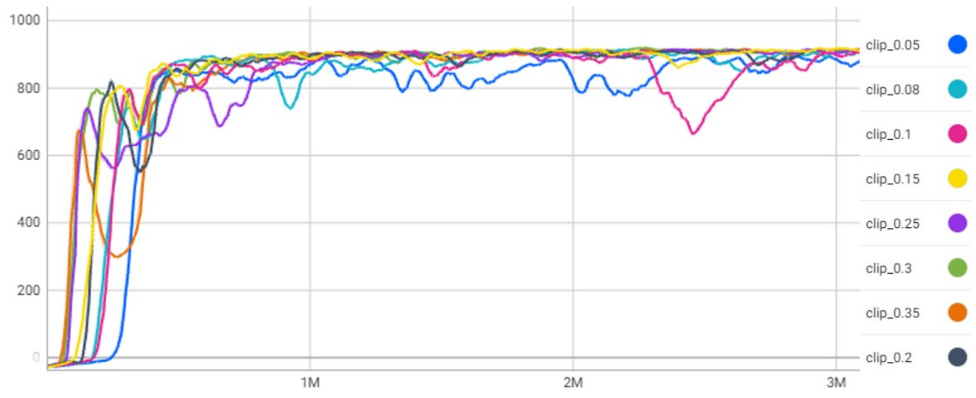


Figure 4.4: Clipping range: mean score per step

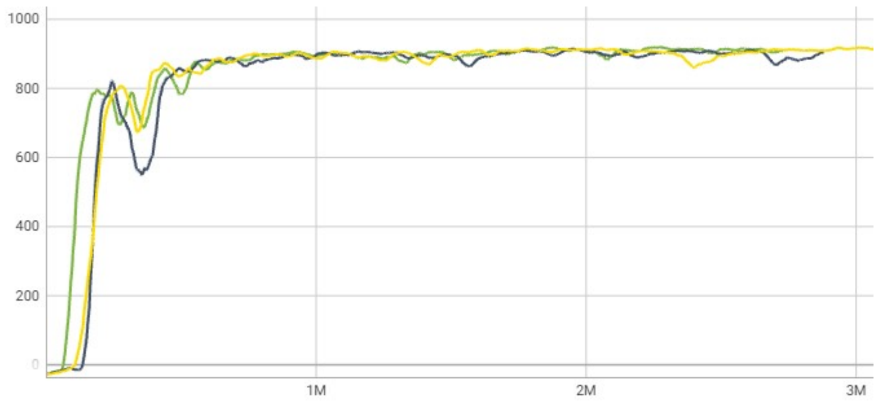


Figure 4.5: Clipping range: mean score per step (same legend as fig 4.4)

4.2.2 Gamma (γ)

The discount factor represented by the Greek letter gamma (γ) accounts for the fact that our agent prefers rewards that it will receive now rather than the same reward further down the line. This can be compared to interest with finances, as a person would rather get the same amount of money now rather than in a year's time. To explain this further, if we have gamma=0.9, the reward in 6 steps is half as important as the immediate reward, whereas, with gamma=0.99, the reward in 60 steps is half as important as the immediate reward.

It is recommended to leave the discount factor between 0.9 and 0.99; however, we wanted to explore a much wider range. As can be seen from figure 4.6, it is truly important how gamma is set. The other hyperparameters had overall similar curves, but this figure shows that a slightly different gamma may ruin the whole training. As we can see, gammas lower than 0.9 are extremely shortsighted and value only the short-term reward, which means that the agent would rather sacrifice the long-term return for one good reward. This can mean that when a car is closing on a turn at a fast speed, it would rather stay on the track and fail to turn, whereas a car with a higher gamma would see the long-term value in cutting the corner over a grass patch.

We decided to stick with the most often used discount factor, 0.99, which showed the best rewards and a stable curve.

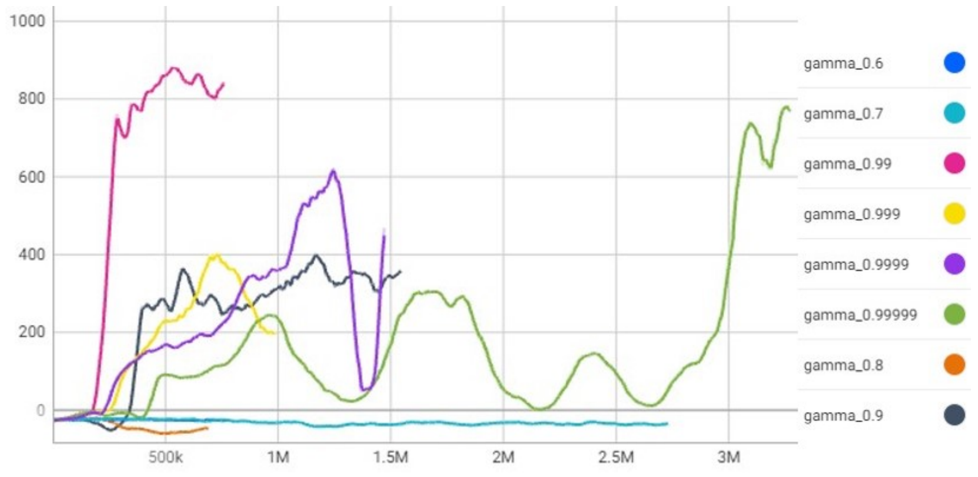


Figure 4.6: Gamma: mean score per steps

4.2.3 GAE Lambda (λ)

GAE lambda is used to control the bias-variance trade-off. If you want to have a smoother training curve corresponding to training being more stable, choose a λ close to zero. A number close to zero means high bias and low variance, while a number close to 1 means the opposite.

In the original PPO paper [SWD$^+$17] they use $\lambda = 0.95$ which is quite large, however, having a high variance in training is beneficial if one does not want to get stuck in a local minimum. Getting stuck in a local minimum is what happened to the yellow curve representing $\lambda = 0.1$ in figure 4.7.

When looking at the graph, we can see that for our environment, the best options would be either 0.8 or 0.9, being the most stable ones. We decided to continue with 0.9, the purple curve, because of its quicker and steeper rise. This would save time in further training.

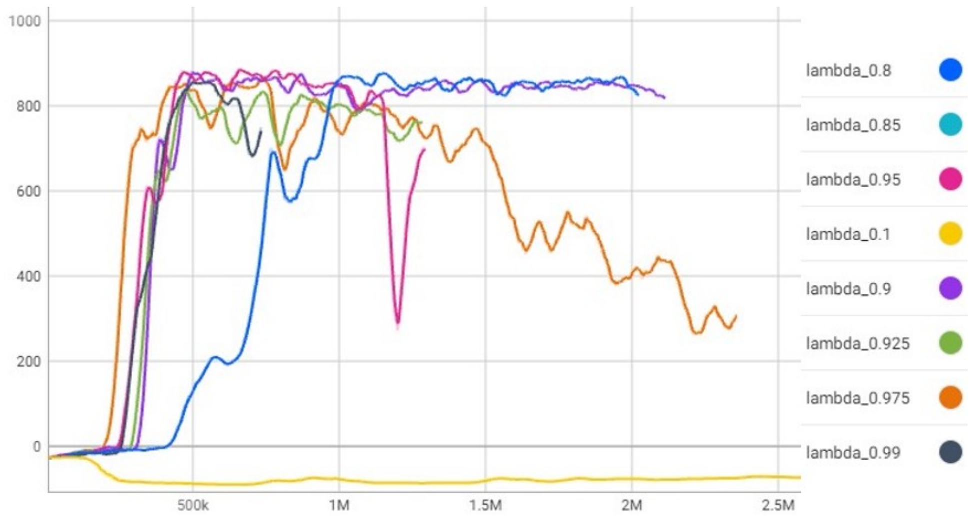


Figure 4.7: GAE lambda: mean score per step

4.3 Loss Function Coefficients

These are the two coefficients c_1 and c_2 in the PPO objective function, which was shown in equation 3.5.

4.3.1 Value Function Coefficient

The value function controls the impact of the value function loss on PPO's objective function. It decides how influential should our prediction, of the value of a state, be.

We tried linearly spaced numbers between 0.5 and 1, and as can be seen in figure 4.8 none of them was a misstep. If we look closer, at figure 4.9, we can see the highest point on the whole graph which was score 919 around timestep 2.26 million. This was achieved by the purple curve representing a coefficient of 0.64. A close second was the pink curve (0.5) which, however, has a slightly more unstable beginning. Our final choice was a value coefficient of 0.64.

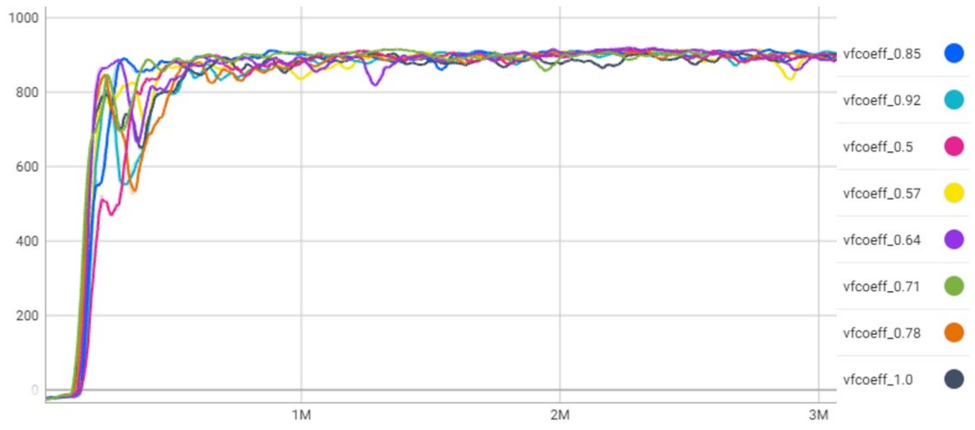


Figure 4.8: Value function coefficient: mean score per step

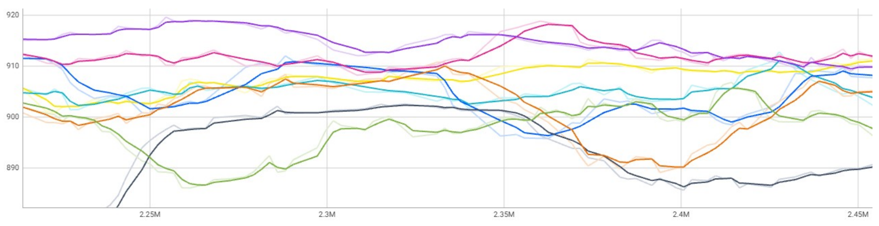


Figure 4.9: Value function coefficient: zoom in on fig 4.8

4.3.2 Entropy Coefficient

The entropy coefficient can be called a regularizer because it helps prevent premature dominance of one action probability over the policy which could prevent exploration. A policy has minimum entropy when a single action has an overly dominant probability. This means that if we always wanted to be greedy and choose the current best action, we would have entropy as low as possible, and a high entropy if we wanted to explore the state space.

In our case, a small entropy coefficient was the best choice since one wrong action could lead to an uncontrollable drift out of the track. If we look at figure 4.10, we can see that any coefficient between 0 and 0.01 is a valid option and if we zoom in on the graph and look at figure 4.11, we can see that two curves even managed to leap over the 920 score mark. In the end, the choice was between pink 0.0043 and purple 0.0071 and since purple had a much more stable beginning, we went with 0.0071.

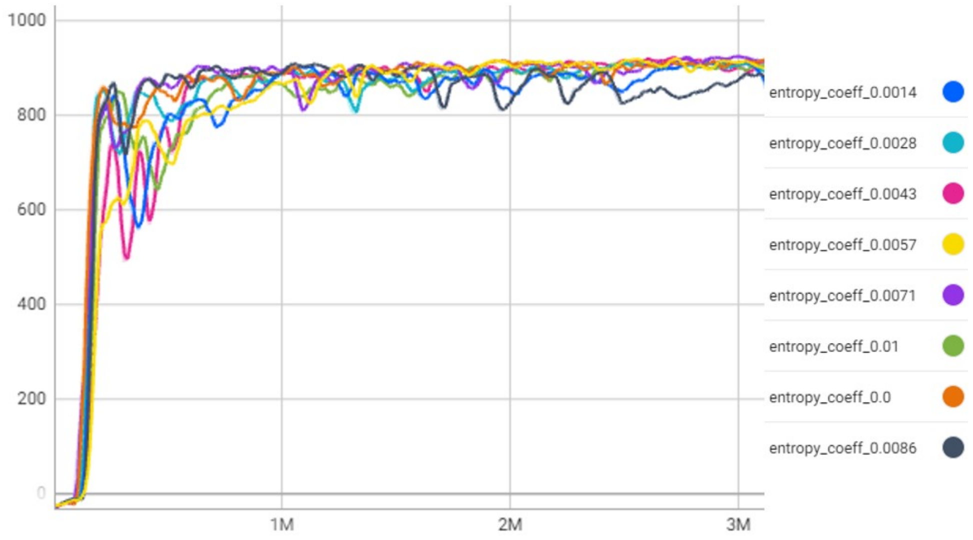


Figure 4.10: Entropy coefficient: mean score per step

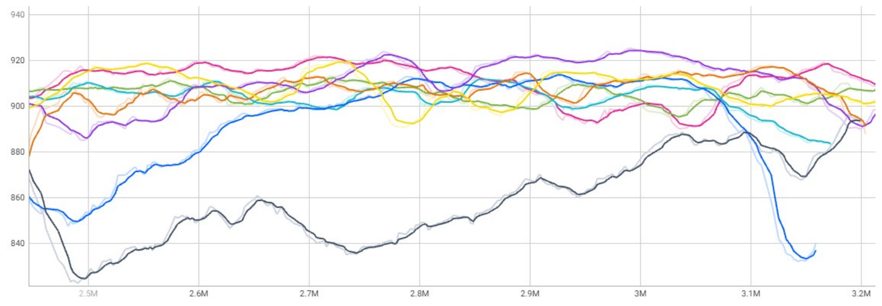


Figure 4.11: Entropy coefficient:zoom in on fig 4.10

4.4 General

Lastly, there are some parameters that appear in all deep Reinforcement Learning algorithms. The learning rate of the optimizer and the condition which controls when to end the training.

4.4.1 Optimizer Learning Rate

The learning rate is how large of an impact should the optimizer have during a single update. For our experiment we chose the Adam optimizer [KB14].

The original PPO paper described using a discounted learning rate [SWD+17], however, we wanted to experiment with both discounted and constant learning rates. After experience collection and before our agent started learning we applied a function to the learning rate to get a new one for each episode. If we wanted a discounted learning rate we multiplied the initial learning rate by a decreasing number $(1 - \frac{\text{current episode number}}{\text{final episode number}})$ which fell linearly from 1 to 0.

A decreasing learning rate is used because at the beginning of training it is useful to explore and be able to escape some local minima. As the agent learns and becomes better at gaining a positive reward, it is much less desirable to change the policy significantly in a single update.

The experiments showed that a learning rate of 0.0003 won in both constant and discounted runs, and if compared, the difference between discounted and constant learning rates was nonexistent. For our final learning rate, we took the discounted 0.0003 because of the theory.

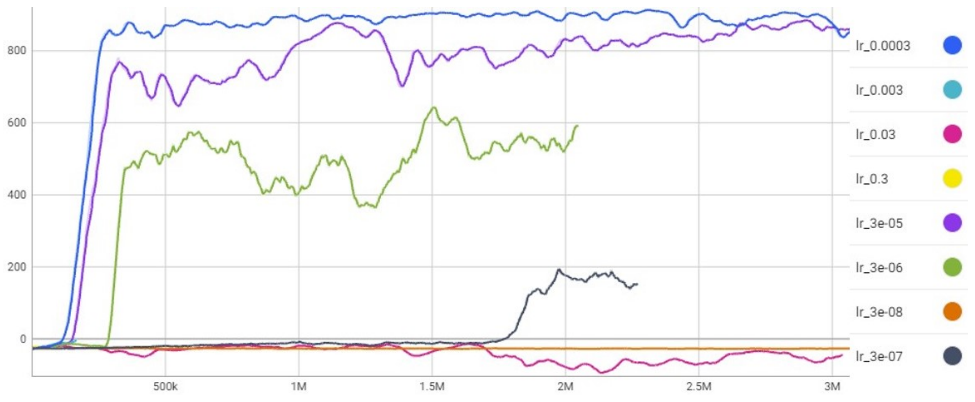


Figure 4.12: Learning rate (constant): mean score per steps

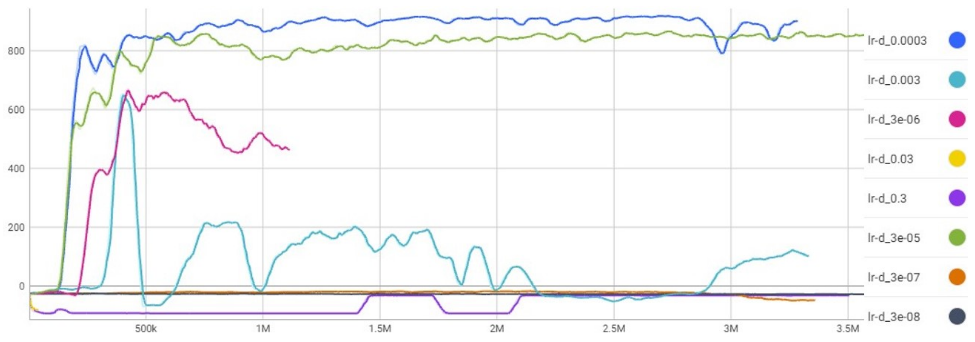


Figure 4.13: Learning rate (discount): mean score per steps

4.4.2 Terminating Condition

Even though the environment has a respected score of 900 which resembles solving the environment, we wanted to explore the hyperparameters and that is why we had no true terminating condition. There was only a placeholder of 4000 episodes which was only meant to be high enough for the experiment to run for over a day. The GPU cluster we used had a one-day limit on the node we were using and we wanted to get as much out of it as possible.

Chapter 5

Conclusion

In conclusion, this project delved deep into the algorithm known as Proximal Policy Optimization. From understanding how it works and how it can be modified to implementing it in OpenAI's Car Racing environment. This racing environment, which uses real-world physics, continuous actions, and a random track generation, showed the full potential of PPO.

Having explored ten hyperparameters, we can say that some had a larger impact than others. With some, the best one could be immediately observed from the graph, while with others, it was not so straightforward. An occurrence that could be seen on all at least mildly successful runs was an initial flat graph followed by a steep rise which we explained as the initial struggle to drive through a turn followed by breaking through the threshold and suddenly learning much faster. It is important that such occurrences can be explained by understanding the theory and the functioning of the environment. AI explainability is currently a leading issue in the field.

In the end, we solved the environment by reaching a score of over 900, which means we successfully solved the environment. For further projects, we would like to explore autonomous cars in more challenging environments and try different algorithms to see how they hold up. The car racing environment could be modified with a random wind affecting the chosen actions or by adding obstacles to the track.

Bibliography

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015, Software available from tensorflow.org.

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba, *Openai gym*, arXiv preprint arXiv:1606.01540 (2016).

[KB14] Diederik P. Kingma and Jimmy Ba, *Adam: A method for stochastic optimization*, 2014.

[NMK22] Andrew Ng, Younes Bensouda Mourri, and Kian Katanforoosh, 2022.

[RN10] Stuart J. Russell and Peter Norvig, *Artificial intelligence: a modern approach*, 3 ed., Pearson, 2010.

[SLM+15] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel, *Trust region policy optimization*, 2015.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov, *Proximal policy optimization algorithms*, CoRR **abs/1707.06347** (2017).

[16] Şenol Çelik and Mehmet Korkmaz, *Beta distribution and inferences about the beta functions*, Asian Journal of Science and Technolog **7** (2016), 2960–2970.

Appendix A

Hardware used for Training

We are extremely grateful for the access The Research Center for Informatics of the Czech Technical University has given us to their GPU cluster.

We ran our experiments with a maximum number of 128 parallel threads set in TensorFlow. We found out that the optimal number of parallel environments was 6. With these numbers set, we ran experiments using 32GB per CPU and 1 CPU per task.

Information about the Research Center for Informatics:
`http://rci.cvut.cz/`

Appendix B

Code

The link to the GitHub repository of this project is:

`https://github.com/sykoravojtech/PPOCarRacing`