Project Report

Reinforcement Learning: Training a virtual dog to fetch a stick

Vixra Keo
Attoumassa Samake
Antoine Bedouch

November 26, 2023

Contents

1 Introduction

Artificial Intelligence (AI) has come a long way in the past decades, in particular thanks to the rise of Machine Learning and Deep Learning. Typically, these methods use large quantities of data in order to perform predictions, each data point nudging the model algorithms to be closer to the truth. This works great when the problem at hand is well defined like for classification or regression, and when the available data is of good quality. There is however, more than just classification and regressions, some problems do not fit in these two categories.

Take for instance the game of chess, how can an algorithm know what the best move is for a board state? Traditionally, chess algorithms would perform minimax computations which consist of looking moves ahead and picking the one that has the best consequences for the AI. This, however, quickly becomes inefficient as the complexity of a game of chess grows exponentially with the number of turns looked ahead. Thanks to Deep Learning however, a new method emerged: Reinforcement Learning. This consists of training an Artificial Neural Network to decide the best next move to take within an environment. Because winning a game of chess is abstract for an algorithm, it is necessary formalise what it means to be in a good position and to win the game. To do that we use what is called a Reward Function: a function that will quantify how well the AI is doing for each taken action. The cumulative reward is a quantifiable metric the model can maximise. We want the cumulative reward to be a proxy of our end goal (for instance, winning a game of chess). Thus, we must create a reward function reflecting our goal: for chess for instance, capturing a strong piece can yield a strong reward, losing a piece can yield a negative reward, and check mating the adversary can yield an enormous positive reward. An algorithm maximising the cumulative reward will have to take decisions minimising losing pieces while maximising capturing pieces, all the while trying to check mate.

To illustrate reinforcement learning, we will try to train a simulated dog to fetch a simulated stick. Reinforcement learning is perfect in this example because we want to train an agent (Huggy the dog) to perform an abstract task that is to fetch a stick. This would be very hard to code explicitly because we need to coordinate the movement of each limb, adapt the movement to the specific dog and stick position. If we find the right reward function and training algorithm, it should be possible to have Huggy learn how to move efficiently towards a target (here the stick).

For this project, we will be using the python library ml-agents (Juliani et al. [2018]) which works in pair with unity environments. Unity is a game and simulation engine in which the physical simulation of Huggy, the stick and the environment, is coded. ml-agents uses Pytorch to train the agents created in unity.

2 Theory

2.1 Reinforcement learning

Reinforcement learning (RL) is an area of machine learning concerned with how intelligent agents ought to take actions in an environment in order to maximise the notion of cumulative reward. Reinforcement learning is one of three basic machine learning paradigms, alongside supervised learning, and unsupervised learning.

The main characters of RL are the agent and the environment. The agent is the learner and decision maker; the environment is the world that agent lives in and interacts with by performing actions.
The agent perceives a reward signal from the environment, a number that tells it how good or bad the current state is. The goal of the agent is to maximise its cumulative reward. We can therefore define RL methods as the ways that the agent can learn behaviours to achieve its goal.

Here we can see that RL algorithms are divided in two big families: Model free RL and Model based RL.

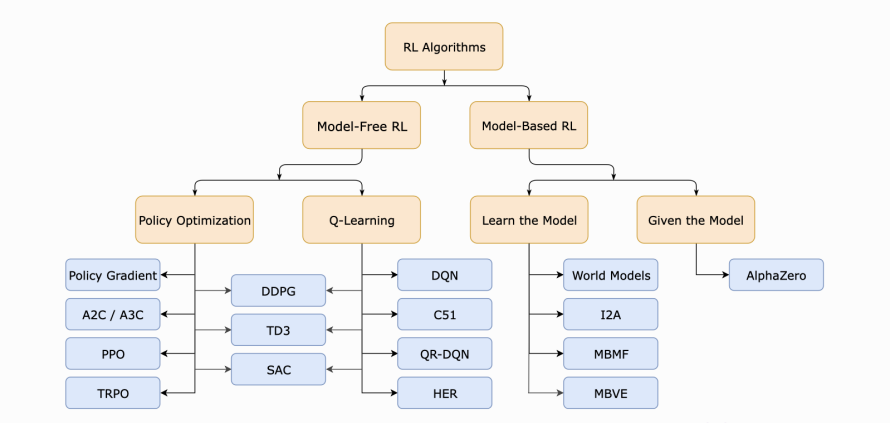

Figure 1: Reinforcement Learning taxonomy as defined by OpenAI

Model-based learning attempts to model the environment then choose the optimal policy (a rule used by an agent to decide what actions to take when in a given state and given environment) based on its learned model: i.e a function which predicts state transitions and rewards. In Model-free learning the agent relies on trial-and-error experience for setting up the optimal policy.

In this project, we will specifically focus on the Proximal Policy Optimization (PPO) algorithm, which is a model free method.

2.2 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (Schulman et al. [2017]) is a policy optimization method for reinforcement learning based on policy gradient methods, and Trust Region Policy Optimization.

Policy gradient methods aim to maximize the expected reward by directly adjusting the policy parameters. To achieve this, they use back-propagation to estimate the gradient of performance with respect to the policy parameters. However, they may suffer from high variability in updates, leading to unstable learning.

TRPO addresses this instability issue by imposing a trust region constraint on policy modifications during each iteration. It employs a trust region approach to ensure that updates are not too aggressive, thereby avoiding abrupt changes that could render the learning process unstable. Nevertheless, TRPO has its drawbacks, as it can be computationally expensive due to the need to solve an optimization problem with constraints.

On the other hand, PPO simplifies the constraint problem by employing a more straightforward and easily optimized formulation. It introduces a new objective function that measures the proximity between the proposed new policy and the old policy while avoiding excessive updates. Consequently, PPO is designed to be more straightforward than TRPO while maintaining a certain level of stability in learning.

The objective function of PPO is defined as follows:

$$L(\theta) = E\left[\min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}A^{\text{adv}}(s,a), \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}, 1-\varepsilon, 1+\varepsilon\right)A^{\text{adv}}(s,a)\right)\right]$$

Where:

- θ represents the parameters of the current policy.
- $\pi_\theta(a|s)$ is the probability of action a given state s with parameters θ.
- θ_{old} are the parameters of the previous policy.
- $A^{\text{adv}}(s,a)$ is the estimated advantage of taking action a in state s.
- ε is a small constant to stabilize updates.

The "clip" refers to the limitation of policy updates. It is a technique aimed at preventing overly large updates that could compromise the stability of learning.

3 Methodology

Huggy the dog project was an experiment developed by Thomas Simonini in the Hugging face platform based on a game called "Puppo the Corgi". The game was first created by a Unity's team developer and introduced at the https://youtu.be/nntQ3QuWyuM. The developer showed the ML-agents framework capabilities applied in the gaming industry for coding the behaviour of Non-Player Characters.
https://blog.unity.com/engine-platform/puppo-the-corgi-cuteness-overload-with-the-unity-ml-agents-toolkit.
They provided the Puppo demo source codes to let people try it. The original project is available at the following link:
https://s3.amazonaws.com/unity-ml-agents/demos/v0.5/PuppoDemo.zip

The Simonini's Jupyter notebook was our starting point. We used Google Colab to run the notebook and take advantage of the compute power of the CPU available to shorten the trainings. Then we changed PPO's hyper-parameters involved in the training to see if there is an improvement in the learning process. The creation of the configuration files as well as the training process has been automated and ran in a row. Finally, we tried to launch the Unity PuppoDemo project in the Unity game software to modify the reward function or change the policy, but we faced various difficulties we will talk about later.

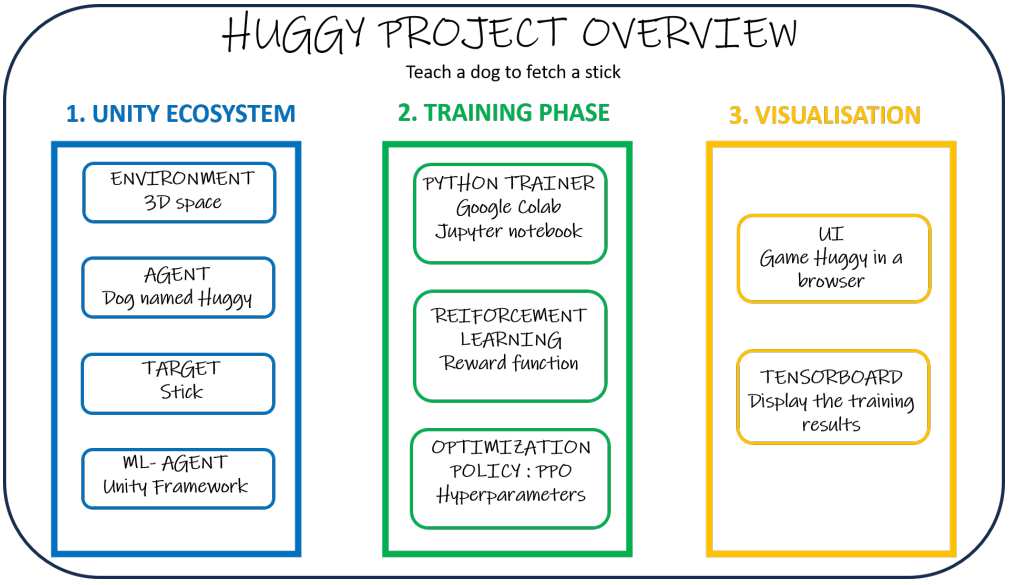

Figure 2: Huggy project overview

3.1 Unity environment

In this section we will introduce, in a nutshells, the technologies and components used to train Huggy fetch the stick.

3.1.1 Unity

Unity is a cross-platform game engine developed by Unity Technologies. They released an open-source project that enables games and simulations to serve as environment for training intelligent agents called "Unity Machine Learning Toolkit" or ML-agents (Juliani et al. [2018]). The Unity game engine provide the graphical interface, with a custom scene, in which the agent can learn.

3.1.2 ML-Agents

The ML agents toolkit contains different components such that :

- Learning Environment : which contains the Unity scene and the game characters. The Unity scene provides the environment where the agent can observe, act and learn.
- Python API : contains a python interface for interacting and manipulating a learning environment.
- External Communicator : connects the learning environment with the python API.
- Python Trainers : contains all the machine learning algorithms that enable training agents.
- Agents : attached to a Unity Game Object.
- Behaviour : define the specific actions the agent can do.

ML-agent is very versatile as it implements various training strategies (sparse reward, exploration / exploitation compromise, imitation learning...)

3.1.3 About Huggy

Huggy is in an environment with complete physics simulation. Its legs are driven by joint motors which have associated 3D coordinates. To get the target Huggy must learn how to rotate the joint motors of each of his legs correctly to go toward the stick.

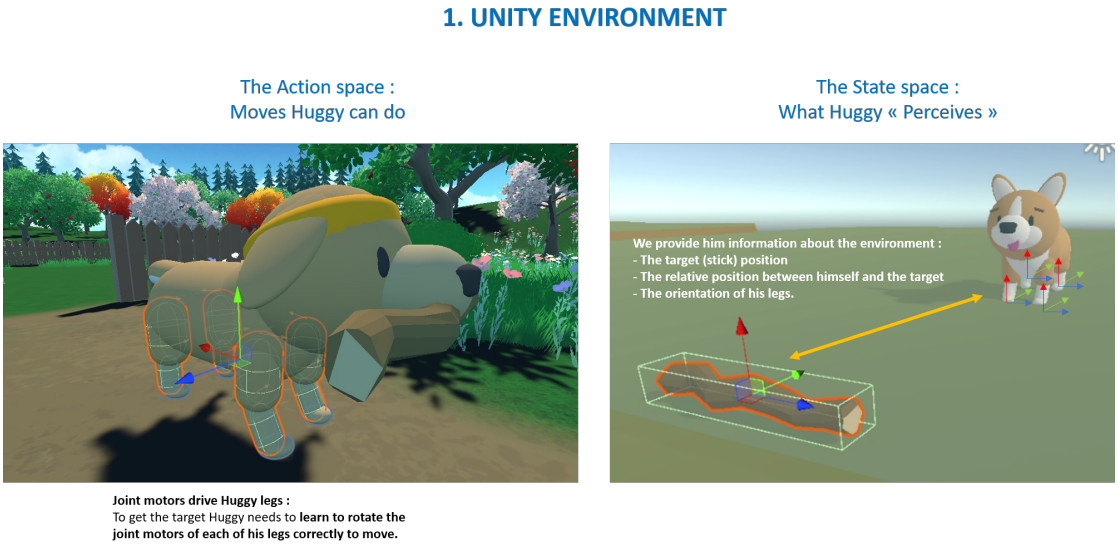

Figure 3: Unity environment

Huggy's observations : The State Space (input data):

- Target position
- Relative position to target
- position of his legs
- Whether Huggy is holding the stick or not

Huggy's actions : The Action Space (output data):

- new position of the legs

As stated earlier, the learning is driven by maximising a cumulative reward. Huggy's reward function is defined as follow:

- Positive reward for getting closer to target (each frame)
- Big positive reward for reaching the target (one time)
- Small negative reward for taking time
- Negative reward for spinning too much or too quickly

The algorithm used to train Huggy is PPO provided by the ML-agents toolkit.

3.2 Training step

3.2.1 Training description

We have described how Huggy evolves in the environment and how the reward function is calculated. Now for training to occur, our optimiser musts maximise the reward function. In order to do so, it measures the cumulative reward function at the end of the simulation and updates the Artificial Neural Network (ANN), i.e. Huggy's brain. After each training, PPO is going to tweak Huggy's brain (the ANN's weights and biases) which define how Huggy's chooses the next best move. Finally, the hyper-parameters of PPO are given by the PPO configuration file.

This process is summarised in the figure 4.

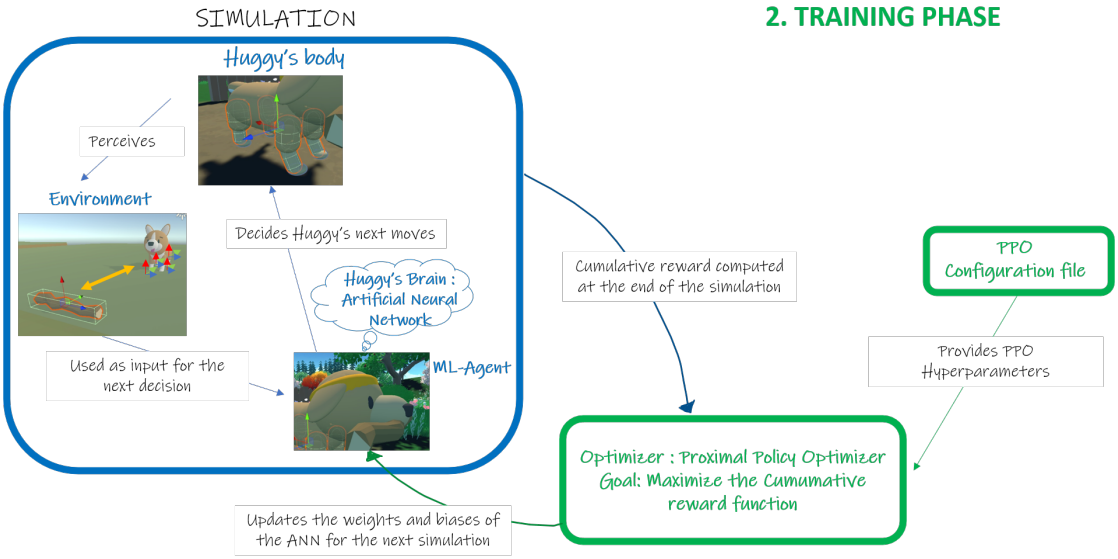

Figure 4: Training step

3.2.2 PPO configuration files

PPO configuration files consist of .yaml files containing the chosen value for each one of PPO's hyper parameter. The Huggy project comes with a default PPO training configuration. we have decided to test different parameters to see how it impacts Huggy's training. We will call the model trained with the default configuration the baseline. For every configuration file, we kept the default configuration except for one value which we changed.

Baseline configuration file:

```
behaviors:
```

```
Huggy:
  even_checkpoints: true
  hyperparameters:
    batch_size: 2048
    beta: 0.005
    buffer_size: 20480
    epsilon: 0.2
    lambd: 0.95
    learning_rate: 0.0003
    learning_rate_schedule: linear
    num_epoch: 3
  keep_checkpoints: 10
  max_steps: 2000000
  network_settings:
    hidden_units: 512
    normalize: true
    num_layers: 3
    vis_encode_type: simple
  reward_signals:
    extrinsic:
      gamma: 0.995
      strength: 1.0
  summary_freq: 10000
  threaded: true
  time_horizon: 1000
  trainer_type: ppo
```


The table below lists each configuration file used for this project, including for each file the one parameter that is different from the baseline.

All the configuration files can be found on the github repository under `src/configuration_files` .

There are different categories of hyperparameters:

- Those linked to the PPO value function (e.g. beta, lambda, gamma)
- Those linked to the artificial neural network structure (hidden_units, num_layers)
- Those linked to the training process (learning_rate, buffer_size)

For more details on the parameters, please refer to the ml-agents documentation.

Configuration filename	Modified parameter name	Modified parameter value
ppo_learning_rate_0.001.yaml	learning_rate	0.001
ppo_learning_rate_1e-05.yaml	learning_rate	1e-05
ppo_learning_rate_schedule_constant.yaml	learning_rate_schedule	constant
ppo_hidden_units_32.yaml	hidden_units	32
ppo_hidden_units_128.yaml	hidden_units	128
ppo_num_layers_1.yaml	num_layers	1
ppo_num_layers_2.yaml	num_layers	2
ppo_normalize_False.yaml	normalize	False
ppo_buffer_size_409600.yaml	buffer_size	409600
ppo_buffer_size_2048.yaml	buffer_size	2048
ppo_beta_0.0001.yaml	beta	0.0001
ppo_beta_0.01.yaml	beta	0.01
ppo_lambd_0.9.yaml	lambd	0.9
ppo_gamma_0.8.yaml	gamma	0.8
ppo_epsilon_0.3.yaml	epsilon	0.3
ppo_epsilon_0.1.yaml	epsilon	0.1

Table 1: Table listing all the PPO configuration files trained for this project

3.3 Visualisation

3.3.1 Play with Huggy

On the Hugging Face platform, user Thomas Simoni has created a web page that simulates the Unity environment of Huggy. This allows interaction with our Huggy agent using the .onnx file obtained after training the model. Here is the link that allows you to play with huggy. You will need to load our model by selecting :"cereale/trained_huggy_models" in the list and then select the .onnx model file you want to perform.

3.3.2 Show training data

During the training process, ML-Agents logs various metrics and statistics related to the training session. These metrics can include things like rewards, episode lengths, and any custom metrics you define in your training script. ML-Agents logs include a TensorBoard integration. This integration involves writing the logged metrics to TensorFlow event files. Once the training is underway, you can start TensorBoard and point it to the directory where the event files are stored. TensorBoard will then read these files and provide interactive visualisations of the logged metrics. You can view graphs, charts, and other visual representations of your training progress.

3.4 Note about tweaking the reward function

One could note that in our planned methodology, we are only talking about modifying the PPO configuration file. However, as we have seen previously and as is illustrated in figure 4, the ppo configuration file is only one of the tools needed for training. In fact the most interesting part would be changing the how the reward function is calculated and see if we could find a better reward function that would either train a better model, or an equivalent one in less training time.

This is something we wanted to do originally, however, in the way the Huggy project has been made, it is not possible to modify the reward function. The reward function seems to be embedded in the `Huggy.x86\_64` executable which we cannot uncompile. We did look into Huggy's parent project: Puppo the dog. In Puppo the dog, we found where the reward function was coded: in a C# script attached to the scene (see appendix A.1).

Seeing this we thought we would compile and train Puppo the dog with various PPO configuration and various reward functions. This however proved to be difficult:

- The scripts to code the scene, the agent, the target and the behaviour are in C#. We did manage to understand what the codes were doing, thanks to the code documentation, but rewriting and compiling the code in a language we weren't familiar with was too challenging for the time frame we had.
- Getting familiar with the Unity ecosystem and tools take time and we decided to focus on what we could easily have access to, in term of knowledge regarding our programming experience.
- Puppo The Corgi uses old version of libraries. Even if we installed the versions required in the requirements file, the debug was difficult due to the incompatibility of version of Unity and the ml-agents libraries.

So while Huggy the dog's black box lacks flexibility, it is convenient for us to use as it requires minimal configuration. This however means that for the rest of the project we have kept the reward function provided by the Huggy project and focused on seeing how different PPO configurations affected the training.

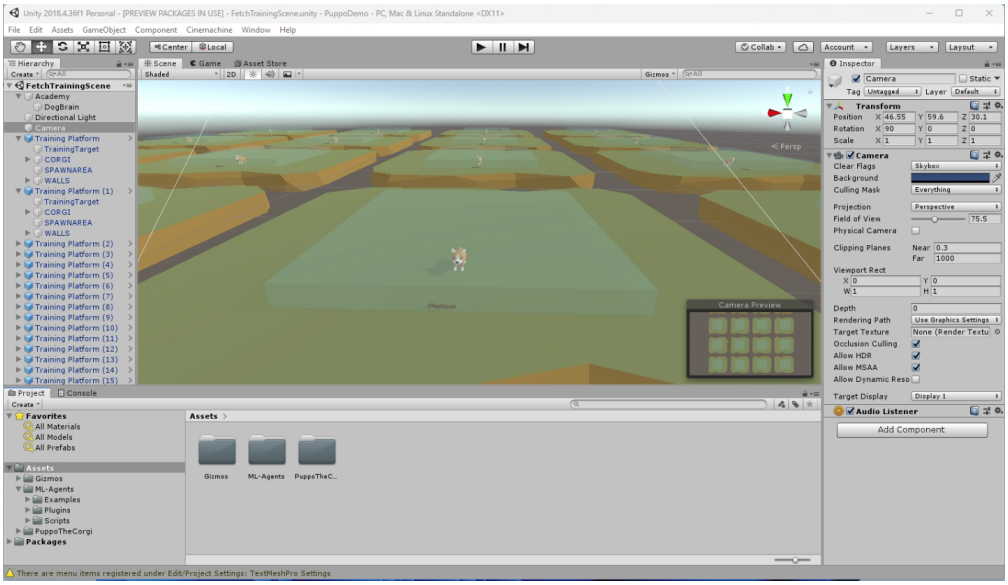

Figure 5: Unity IDE

4 Results

4.1 Result presentation

Now that we have trained our models, we want to analyse the results. The goal of course is to have the dog fetching the stick as well as possible. This however is very hard to quantify. We could just load every model in the web demo and assess how well each Huggy is performing, but this would be not only tedious, but also very subjective and imprecise.

While there is no precise and objective way to measure how well a dog fetches a stick, we could imagine creating a benchmark measuring precisely quantitatively how well Huggy fetches the stick. Well we are in luck because this is exactly what the reward function is: a quantitative measure of our desired behaviour.

So in order to compare our trained models, we will compare the evolution of the reward function over the training iterations. In order to plot this, we used Tensorboard which provides exactly that plot. In figure 6 we have plotted all the ppo cumulative reward functions as a function of training iteration. This plot is really hard to read due to the number of curves. For reading convenience, we have hosted a tensorboard instance showing these exact results where one can interactively select which plot to display. If the server is still running, this hosted board can be found here: `http://matrix-antoine-bdc.duckdns.org:6006`.

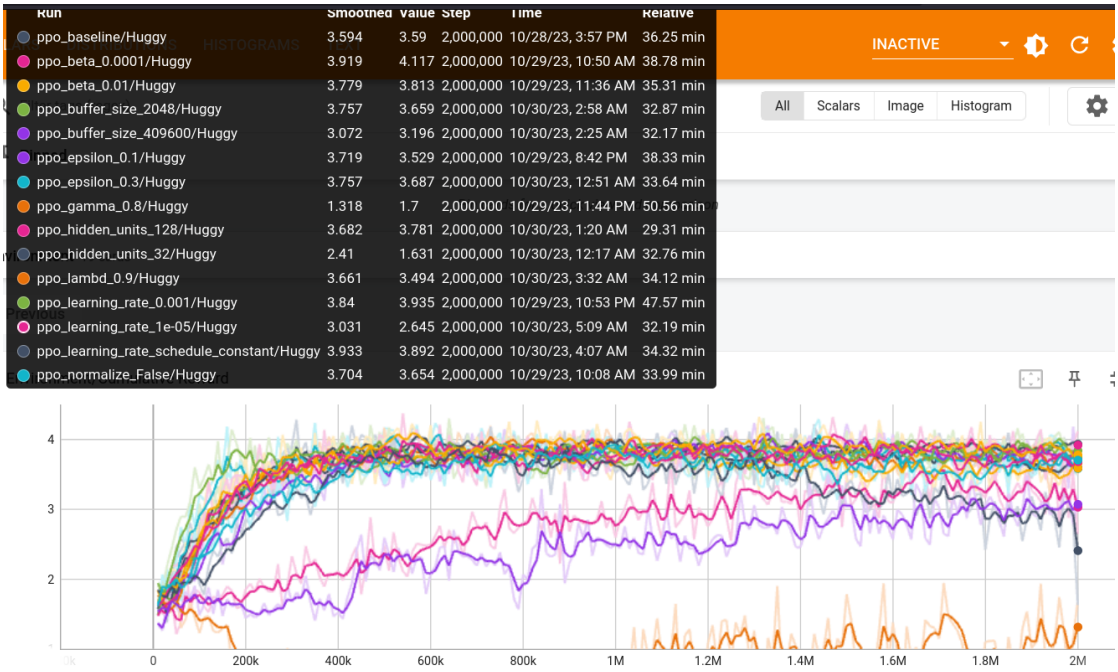

Figure 6: Cumulative reward as a function of training iteration for every PPO model

When looking at the overall curves, we notice how most of them follow a similar path: The cumulative reward increases quickly at the beginning of the training (< 500k steps). After the model reaches about 3.75 of cumulative reward, at 500k steps, the cumulative reward stagnates and reaches a constant. Deviations from that constant seem to be due to noise. In particular we notice that our baseline model follows this exact trajectory as seen in plot 7.

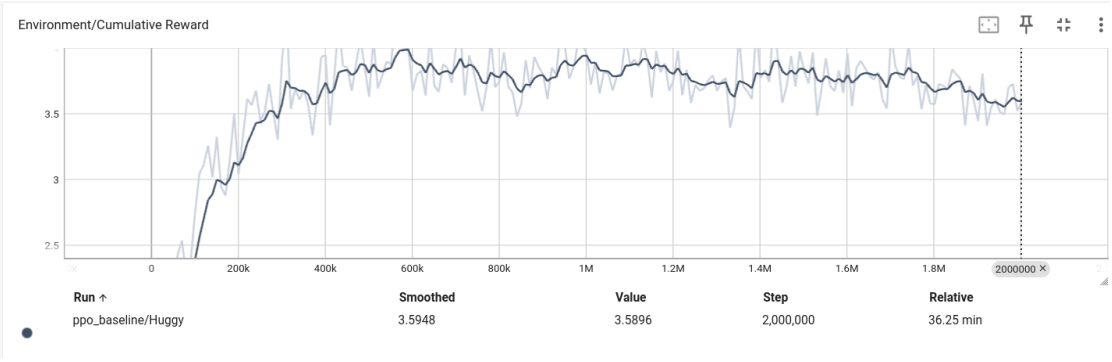

Figure 7: Cumulative reward as a function of training iteration for the PPO baseline model

Due to how we created our training configuration files, it would be natural to just plot every model result against the baseline. That would be a total of 16 plots. Even when grouping models where the same variable has been changed it yields a total of 10 plots. Because we noticed earlier that most training cumulative reward are very similar, we will only plot the outliers here. To see the other plots one can simply select the relevant ones in the hosted tensorboard instance. If the tensorboard instance isn't running anymore, we have also included these plots in the appendix A.2.

Instead we will focus on these few reward functions that did not follow the baseline:

Comparison with Gamma (figure 8): The training is very inefficient for $\gamma = 0.8$, it doesn't converge at all to the 3.75 of cumulative reward like the baseline. It is unclear whether it would ever converge, or if it will converge but very slowly. We can conclude that $\gamma = 0.995$ (value used in the baseline) or higher is needed for the training to be successful.

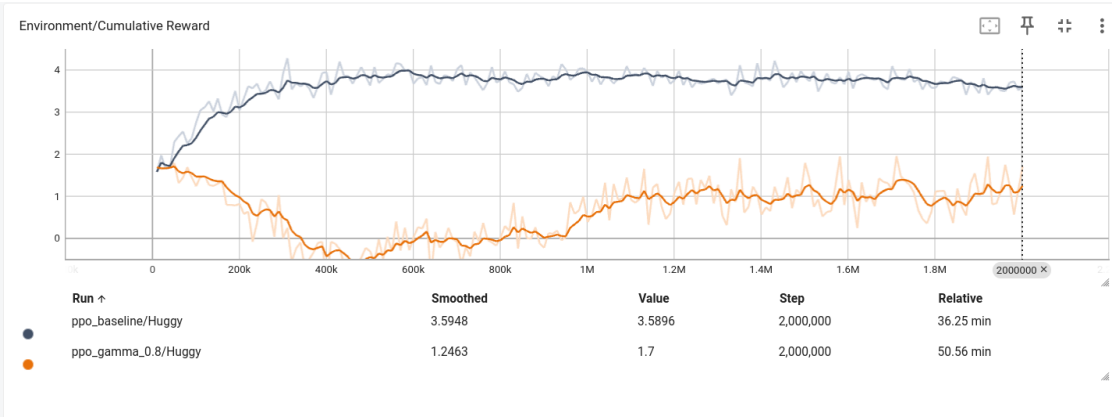

Figure 8: Comparison of the baseline with $\gamma = 0.8$.

Comparison with different learning rates (figure 9): We observe that the training is comparable to baseline for a learning rate = 0.001. It is however much slower for a learning rate of 0.00001. At the end of the training, the cumulative reward function seems to be still increasing for the smaller learning rate. Our hypothesis is that the cumulative reward would eventually converge to the same plateau the baseline converges to. This is not surprising as the learning rate directly affects how fast our model can evolve from an iteration to the next.

Comparison with different buffer sizes (figure 10): No real noticeable difference with the baseline when the buffer size is larger than in the baseline (buffer size = 409600). However, when the buffer size is much smaller (buffer size = 2048), the training seems to be much less efficient. It only reaches a total cumulative reward of 3 at the end of training. It does seem that with more iteration steps, the training would eventually

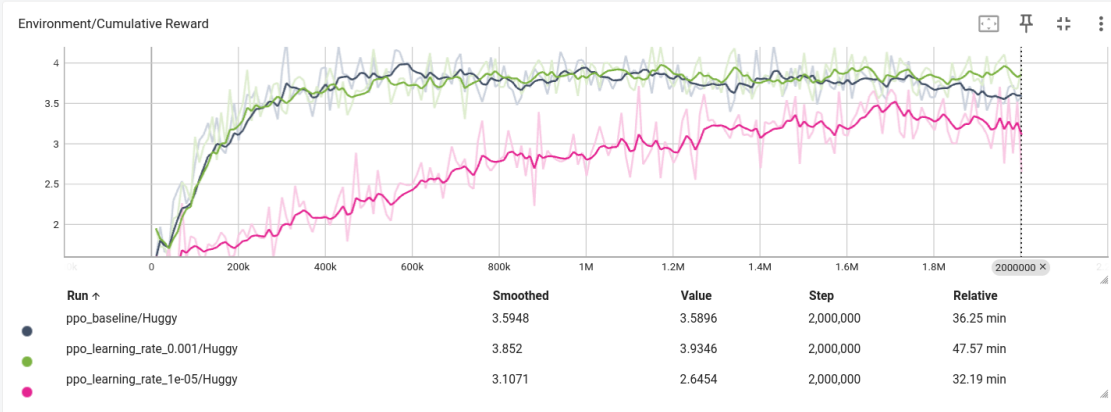

Figure 9: Comparison with learning rate = 0.001 and learning rate = 0.00001

converge to the same plateau as the baseline.

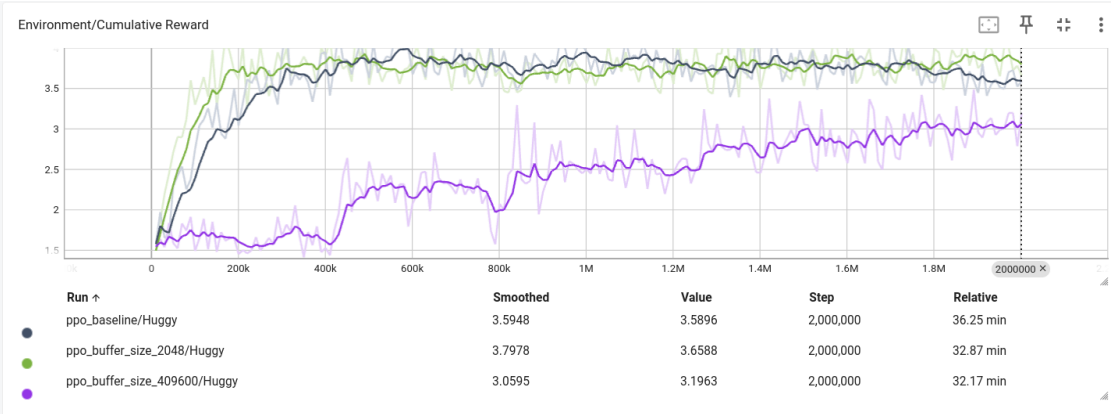

Figure 10: Comparison with Buffer Size = 2048 and buffer size = 409600

Comparison with Hidden units (11): No noticeable difference with baseline when we use 128 units in the hidden layer. For 32 hidden units however, the training seems to be performing well initially and reaches the same plateau as the baseline, but after 1.4M steps, the cumulative reward decreases. This indicates that the training is less robust when the number of hidden units is lower. One way to understand this would be to say that the neural network is so small that small perturbations in the inputs can cause big perturbations in the output.

The other hyperparameters we changed in the configuration don't seem to have much of an effect on the training when compared to the baseline. This could simply be because the values we tested were too close to the baseline.

Now we have discussed how well trained our models are, it is possible to load them in

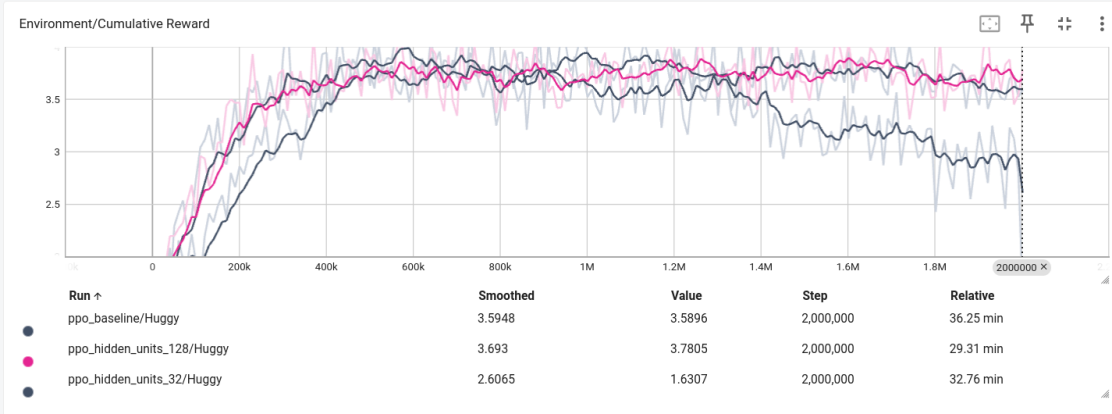

Figure 11: Comparison with Hidden units

a special unity scene and see how well it fetches the stick. This special scene is hosted on Thomas Simonini's HuggingFace page, the creator of the Huggy project.

In order to play with the models we trained:

- Go to this page: https://huggingface.co/spaces/ThomasSimonini/Huggy
- Select our HuggingFace repository: Cereale/trained_huggy_models
- Select the model you want and click 'Play'

When playing with our models, we observe that most of them (those that reached the plateau of 3.75 of cumulative reward), all are pretty good at fetching the stick.

4.2 Result discussion

So we've seen that for all our trained models, most of them seemed to converge to a specific value of the cumulative reward, around 3.75. Why is that, and how can we interpret these results? Our interpretation lies in the fact that our freedom of action is limited to the PPO configuration file. Yet as we have seen previously (in graph 4), the optimiser configuration only impacts **how** the optimiser optimises, not **what** it optimises.

This basically means that most of our trained models have converged toward a single maximum of cumulative reward. Therefore these models should all perform very similarly as we suspect the neural network is performing the exact same function.

It is impossible from this analysis to prove that this maximum our model reached is a local or a global maximum. Though, considering the number of different configurations file, we suspect that this is at the very least the best maximum one can reach using the PPO optimiser.

If we wanted to pursue the analysis and train an even better performing model, we should look into changing other parts of the pipeline:

- Change how we reward our agent Huggy. Maybe it is possible to get an even better performing model by tweaking or completely changing how we reward Huggy (one could imagine putting more emphasis on fetching speed, or on minimising superflous movements). We unfortunately didn't get to try this approach as described at the end of the methodology section.
- Use different optimiser algorithms. Maybe for some PPO is inherently incapable of reaching some hypothetical global maximum. We have briefly tried to train our model using the Self Actor Critic (SAC) optimiser, but it didn't seem to work very well and the training would hang at the very beginning.

5 Conclusion

In this project, we have implemented a reinforcement learning system which aimed to train Huggy the dog to fetch a stick. In practice, we have used the Unity environment of Huggy the dog, and performed training using the PPO algorithms provided by ml-agents.

We have tried multiple PPO configurations and found that most of them seem to converge toward a single final cumulative reward. We interpreted this as PPO finding a cumulative reward maximum, which provided a successful Huggy model. In our opinion, with the tools given to us, we couldn't train a better performing model.

In order to further this analysis, we considered changing the reward function: Either tweaking the importance of each component, or completely changing the nature of the rewards. However, due to the technical implementation of the Huggy environment, we couldn't in practice modify the reward function. If one was to continue this project, one could try to recompile the Huggy environment using its parent project: Puppo the dog.

On a personal note, we all agreed that this project was interesting. We learned a lot about reinforcement learning and how it can solve complex problems. While we found the technical implementation of Huggy useful and generally hassle free to implement, we found ourselves to be limited in how we could improve the model. We feel it would have been an even better learning experience if we were to implement a reinforcement learning solution from scratch, even if it was a simpler problem to solve.

References

Juliani, A., Berges, V., Vckay, E., Gao, Y., Henry, H., Mattar, M., and Lange, D. (2018). Unity: A general platform for intelligent agents. *CoRR*, abs/1809.02627.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms.

A Appendix

A.1 Reward function code

```
    /// <summary>
    /// Reward moving towards target & Penalize moving away from target.
    /// This reward incentivizes the dog to run as fast as it can
    /// towards the target,
    /// and decentivizes running away from the target.
    /// </summary>
    void RewardFunctionMovingTowards()
    {

float movingTowardsDot = Vector3.Dot(
    jdController.bodyPartsDict[body].rb.velocity, dirToTarget.normalized);
        AddReward(0.01f * movingTowardsDot);
    }


    /// <summary>
    /// Time penalty
    /// The dog gets a pentalty each step so that it tries to finish
    /// as quickly as possible.
    /// </summary>
    void RewardFunctionTimePenalty()
    {
        AddReward(- 0.001f);  //-0.001f chosen by experimentation.
    }

}

        RotateBody(rotateBodyActionValue);

        // Energy Conservation
        // The dog is penalized by how strongly it rotates towards the target.
        // Without this penalty the dog tries to rotate
        // as fast as it can at all times.
        var bodyRotationPenalty = -0.001f * rotateBodyActionValue;
        AddReward(bodyRotationPenalty);
```

```
// Reward for moving towards the target
RewardFunctionMovingTowards();
// Penalty for time
RewardFunctionTimePenalty();
```

A.2 All PPO result comparison

Below are plotted all the comparison of the PPO training between the baseline and the changed parameter:

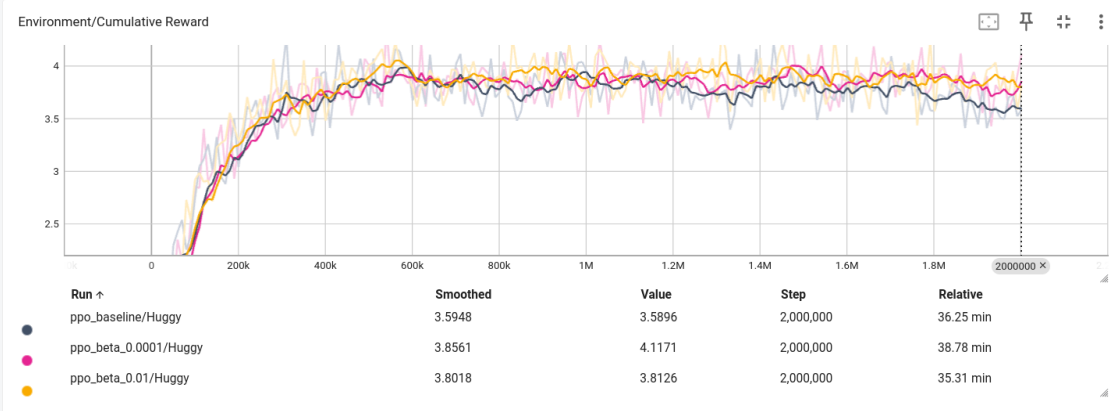

Figure 12: Comparison with Beta: No real noticeable difference with the baseline for $\beta \in (0.0001, 0.01)$

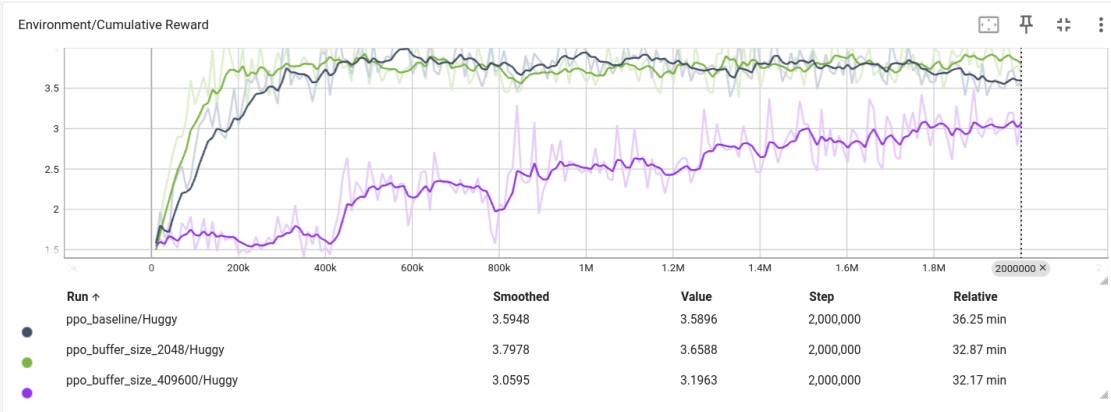

Figure 13: Comparison with Buffer Size: No real noticeable difference with the baseline for buffer size = 409600. The training is much less efficient for buffer size = 2048. It only reaches a total cumulative reward of 3 at the end of training.

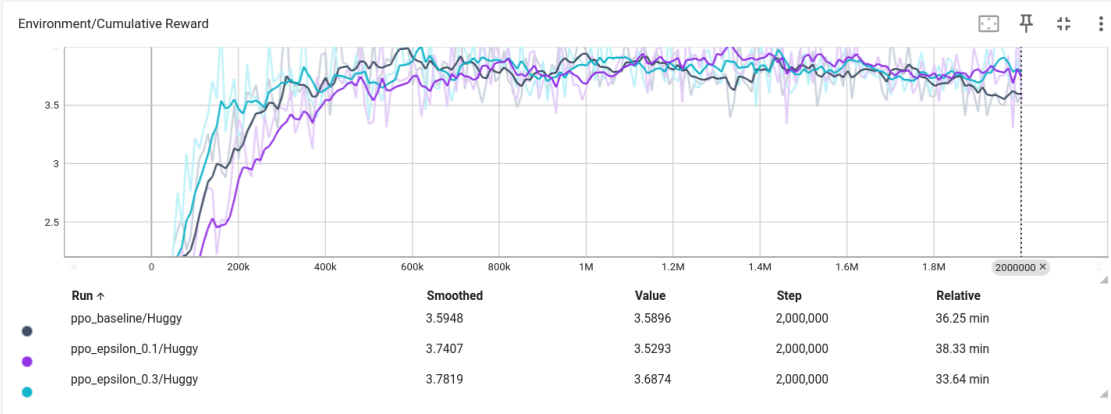

Figure 14: Comparison with Epsilon: No noticeable difference with baseline for fully trained cumulative reward. The training seems slightly faster (i.e. the cumulative reaches its final value faster) for higher values of epsilon.

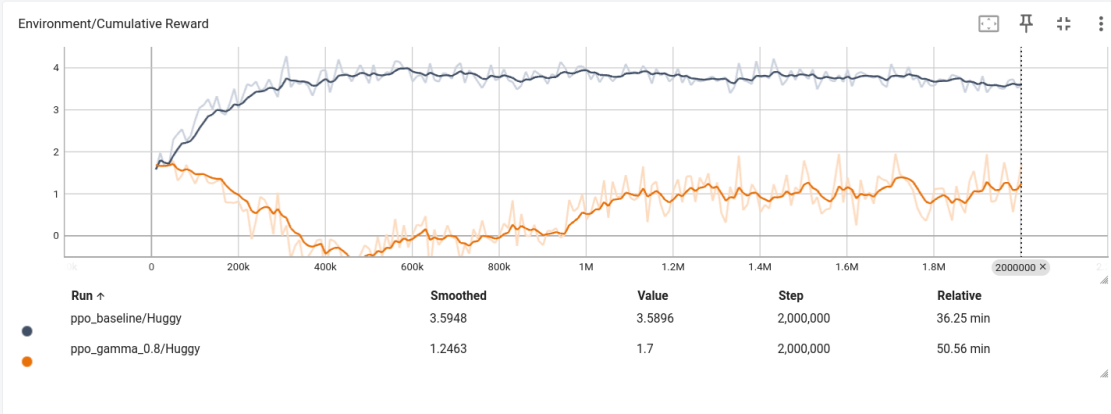

Figure 15: Comparison with Gamma: The training is very inefficient for $\gamma = 0.8$.

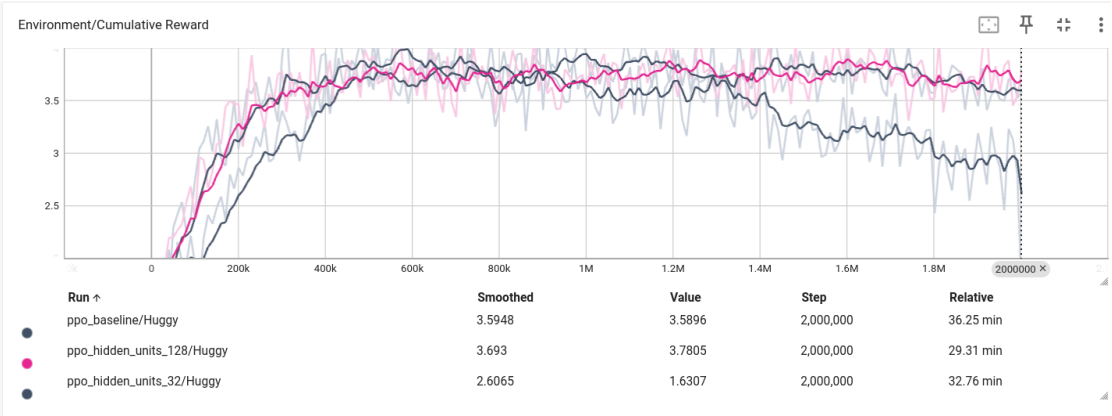

Figure 16: Comparison with Hidden units: No comparable difference with baseline for 128 units in the hidden layer. For 32 hidden units, the training seems to be performing well initially and reaches the same plateaus as the baseline, but after 1.4M steps, the cumulative reward decreases. Maybe the training is less stable with less hidden units.

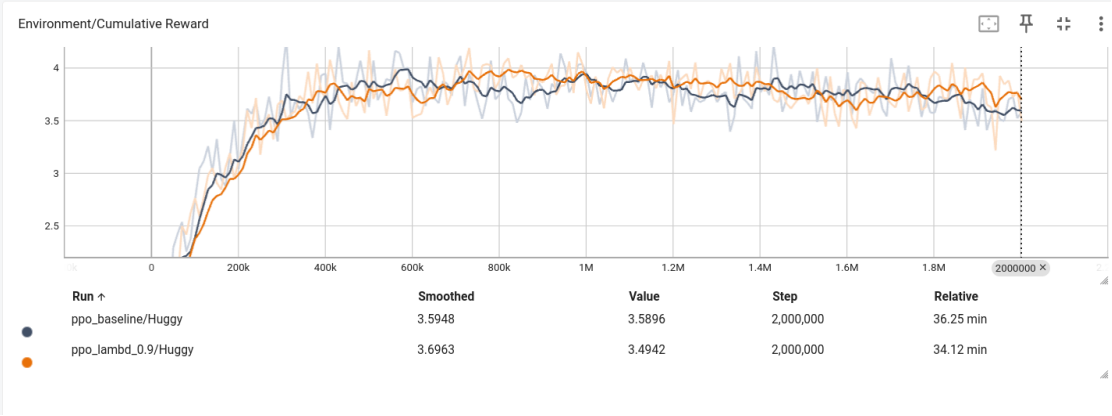

Figure 17: Comparison with lambda: Training comparable to baseline for $\lambda = 0.9$.

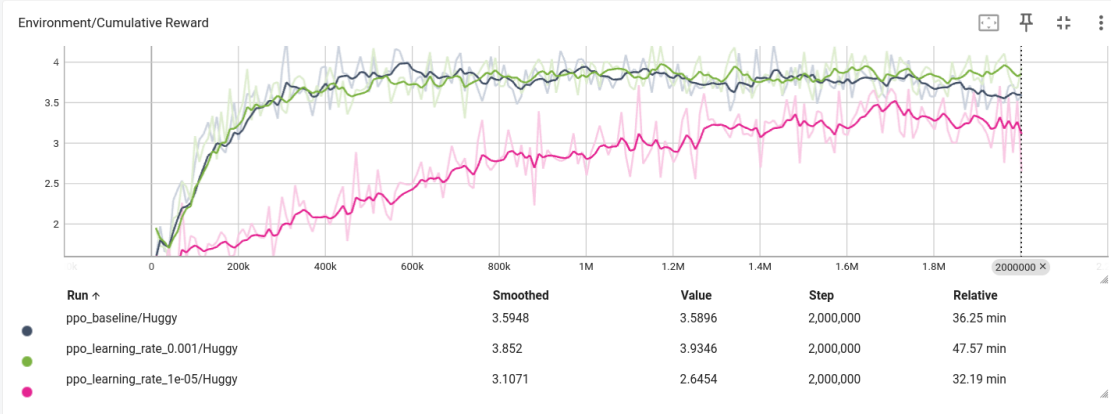

Figure 18: Comparison with learning rate: Training comparable to baseline fora learning rate of 0.001. It is however slower for a learning rate of 0.00001 which is not surprising.

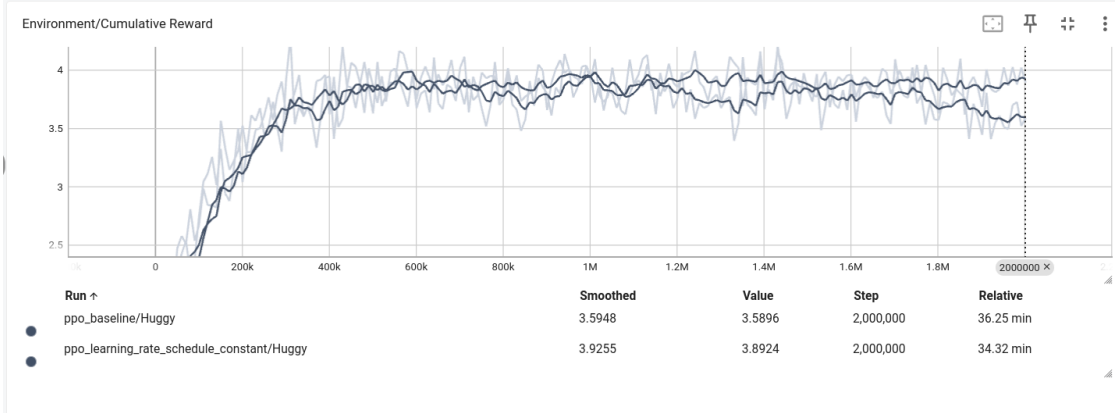

Figure 19: Comparison with learning rate schedule: the training seems to be the same whether the learning rate is constant or evolves linearly.

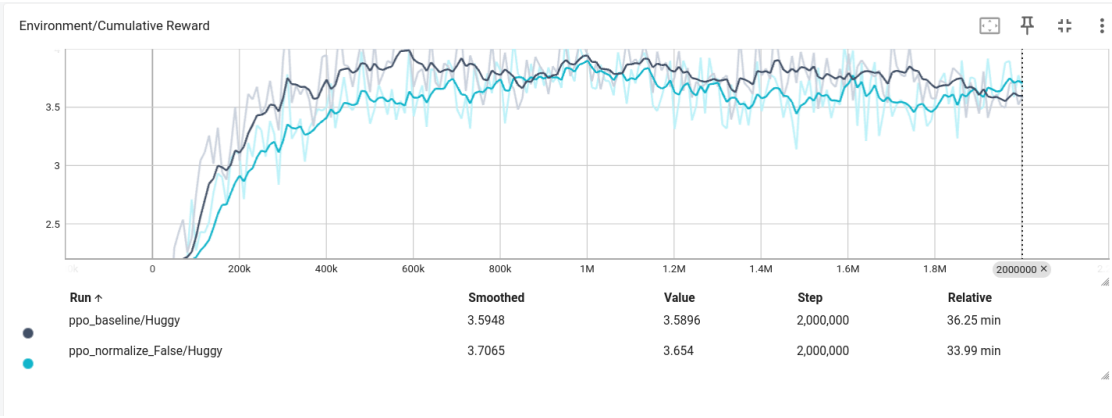

Figure 20: Comparison with Normalize: The training is comparable whether or not the normalize parameter is on. It seems to be slightly faster to train with the normalization

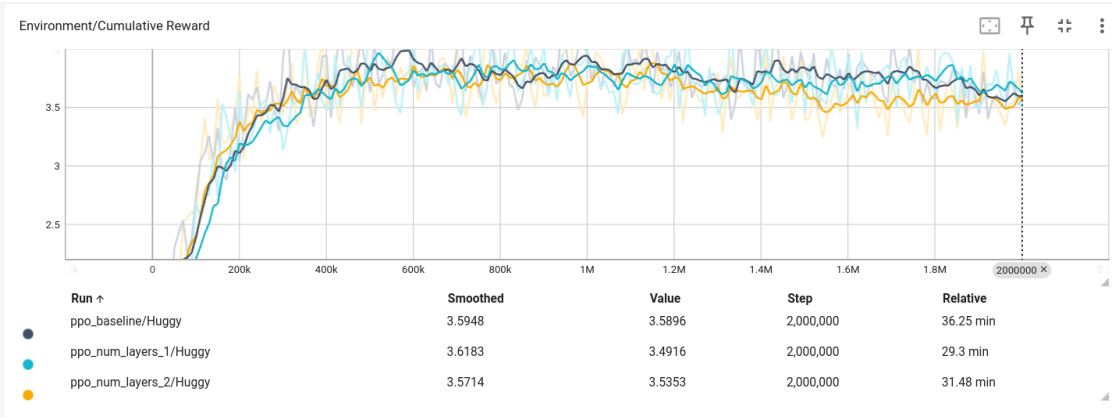

Figure 21: Comparison with Number of layers: No difference between the different numbers of layers in the cumulative reward.