Self-flying Airplane using Proximal Policy Optimization

Daniel Hu[1], Drinas Kastrati[2], Emil Bråkenhielm[3]

Abstract
This report covers the theory and implementation of training an AI-agent to navigate through an environment, reaching checkpoints without crashing along the way. The agent takes the shape of an old fighter jet and everything is visualized in a 3D environment using Unity. The agent is trained using reinforcement learning methods, more specifically using the algorithm called Proximal Policy Optimization which allows for a more stable and consistent training. The results show that Proximal Policy Optimization can successfully train an agent to follow a set of checkpoints in a certain order and that more complex actions and environments are possible to achieve.
Source code: https://gitlab.liu.se/emibr678/tnm095-ai-plane.git

Video: https://www.youtube.com/watch?v=CplOO-hsRcI

Authors
[1] *Computer Science Student at Linköping University, danhu028@student.liu.se*
[2] *Computer Science Student at Linköping University, drika827@student.liu.se*
[3] *Computer Science Student at Linköping University, emibr678@student.liu.se*

Keywords: Reinforcement Learning — Proximal Policy Optimization

Contents

1. Introduction

The aim of this project is to explore how an agent can be implemented and trained using reinforcement learning methods to fly an airplane. Following a set path in the air made out of checkpoints whilst maintaining a stable altitude and not crashing. The airplane is a heavily simplified model based on any normal commercial airplane or an old fighter jet, and the visualization is represented in a 3D environment using Unity.

The reinforcement learning method used in this project to train the AI-agent is called *Proximal Policy Optimization* (PPO), which was first introduced by an *OpenAI* research team in 2017 [1]. PPO is a popular and efficient reinforcement learning approach in training AI-agents to solve tasks in game-like environments and control problems in the field of robotics. PPO is an on-policy algorithm that can be used in environments that uses both discrete and continuous action spaces. It also supports parallelization to accelerate agent training.

This report will cover the theory behind reinforcement learning using PPO, as well as the details and specifics of the implementation of the project. The results will be presented with training data and statistics. Finally, a conclusion is drawn and discussed with future work in consideration.

2. Theory

In this section, all theoretical foundations of the project will be presented, including a short introduction to the reinforcement learning method as well as the theory behind the *proximal policy optimization* (PPO) algorithm.

2.1 What is Reinforcement Learning

The idea behind reinforcement learning is to train an agent to solve tasks through guidance in form of positive and negative rewards without specifying the details of *how* the agent should solve the problem. The most common form of reinforcement learning applied in the field of artificial intelligence, involves an agent that is connected to an environment via *observations* and *actions*.

Observing is the agent's way of perceiving the world and environment. This can range from partial to full knowledge of the world. Partial knowledge is achieved from specific sensors that are available to the agent whilst full knowledge is provided directly to the agent by the user or creator of the agent. These observations are used to represent the current state of the world around the agent.

Actions, on the other hand are the ways for the agent to interact with the world as perceived through observations. Each action input made by the agent affect the current state of the world and results in a new state as an output. Depending on the output, the agent would receive either a positive or negative reward (reinforced learning), which can be saved in the agents memory for further *exploitation*. In the long run, the agents behavior should converge into selecting actions that tend to result in higher positive rewards.

An important part of reinforcement learning is the balance of ratio between *exploitation* and *exploration*. Exploration is based on random actions for the agent to learn what actions result in positive or negative rewards and saves that knowledge in the agents memory, while exploitation utilizes the agents memory of previous experiences in choosing future actions. For the agent to discover what behaviors yield the highest reward, the world has to be explored through random actions. However, since the whole point in training an agent is for the agent to take the best actions to solve a task, the exploration ratio should be decreased in the long run.

Finding the perfect ratio between exploration and exploitation is not an easy task and testing is most likely required in all cases in order to find the best configuration for a certain project. For example, if the exploration phase is removed too early from training, the agent might not converge into selecting the best possible actions. However, instead converge into picking sub-optimal actions since more optimal actions have not been discovered.

2.2 Proximal Policy Optimization

Proximal Policy Optimization or PPO for short was first proposed by an *OpenAI* research team consisting of Schulman et al.[1]. PPO is a novel reinforcement learning algorithm based on *policy gradient methods*. The advantages of PPO compared to standard policy gradient methods is that the performance is better when it comes to computational complexity and is significantly easier to implement.

The main point that PPO seeks to improve is what other reinforcement learning methods were lacking in at the time, such as poor data efficiency and robustness, computational complexity and noise, and parameter sharing management.

PPO does this by trying to make the largest possible improvements on any given decision making policy only using currently available data without cause any performance crashes. In practice, this method can be roughly explained in four simple steps:

1. An agent collects a small batch of data (experiences) by interacting with the world through actions.
2. The collected batch is used to update the decision making policy.
3. Each time the decision making policy is updated, the collected batch of data is discarded.
4. Repeat from step 1 using the newly updated decision making policy.

Each batch of data collected is used to update the current decision making policy once and is thereafter discarded after each update of the decision making policy.

2.2.1 Clipped Surrogate Objective

Clipped Surrogate Objective is the method used to avoid having a policy diverging too far from the old policy. The cause behind a policy diverging too far is because the update made to the policy in a single update is too large making the difference between the new and old too dissimilar. The clipping method is used to avoid this by yielding less variance to ensure that a more stable training is achieved at a minor cost of slightly higher bias.

The vanilla policy gradient methods work by optimizing the following loss shown in equation 1:

$$L^{PG}(\theta) = \hat{E}_t[log\pi_\theta(a_t|s_t)\hat{A}_t] \tag{1}$$

where $\hat{A}$ is the advantage function. By performing gradient ascent, we will take actions that the would more likely lead to higher rewards. Using the idea of importance sampling, we can replace the objective function with:

$$\widehat{r_\theta} = \underset{\theta}{maximize} \quad \hat{E}_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}\hat{A}_t] \tag{2}$$

The ratio in equation 2 will be greater than 1 if the action a_t is more probable than it was in the old policy π_{old}. Otherwise, it will be between 0 and 1. In *trust region policy optimization* (TRPO), the objective function is replaced by a surrogate objective function as shown in equation 3 [2].

$$\underset{\theta}{maximize} \quad \hat{E}_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}\hat{A}_t - \beta KL[\pi_{\theta old}(\cdot|s_t), \pi_\theta(\cdot|s_t)]] \tag{3}$$

The KL term above is to bound the improvement step to a trust region for some fixed penalty coefficient β.

But using a KL divergence term, it becomes more complicated to implement and harder to understand. So instead, Schulman et al. decided to bound the policy update with the following clipped surrogate objective function as shown in equation 4 [1].

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t] \tag{4}$$

Here, $\widehat{r_\theta}$ is defined in equation [2] and ε is a hyperparameter (say $\varepsilon = 0.2$, which was used as an example by Schulman et al. [1]). The $clip$ function prevents the ratio from going outside the boundary of the interval $(1-\varepsilon, 1+\varepsilon)$. Taking the minimum of the clipped and unclipped objective gives a lower bound on the final objective function (i.e. a pessimistic bound).

2.2.2 PPO Algorithm

PPO uses an Actor-Critic style of approach when training an agent. This style of approach involves an *actor* model and a *critic* model. The actor model learns to predict what actions to take in the current observable state and the critic model learns to evaluate the outcome of the actions (i.e. positive or negative rewards) and returns a feedback back to the actor model.

The algorithm proposed by Schulman et al. [1], uses N parallel actors for each iteration and collects data for each timestep T as shown in algorithm 1. The surrogate loss is then constructed on the NT timesteps of data and then optimized with regards to the objective θ using minibatch *stochastic gradient decent* (SDG) for K epochs.

As the optimization epoch goes on policy π will diverge more and more from π_{old} until the objective starts to be clipped and the gradient dies. Then $\pi_{old} \leftarrow \pi$ and a new iteration starts.

Algorithm 1: PPO, Actor-Critic Style

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

3. Method

In this section, details about the implementation of the project will be presented. Firstly, the environment where the training will occur will be covered, followed by the explanation of how the agent was built, and lastly the reward and penalty system will be explained.

The project was implemented in the game engine *Unity* and by using the *ML-Agents* library we implemented the proximal policy optimization. The ML-Agents library includes a ready-to-use template which contains all the necessary parts of the PPO algorithm, as well as a configuration file for modifying the training parameters.

3.1 The Environment

The main objective of this project was to teach an agent to self-fly a plane model through a rough and ready path made of checkpoints in the air while avoiding obstacles such as hills and mountains in this project's case. The world was designed with a few key points in mind:

1. World boundary
2. Checkpoint placement and spacing
3. Starting area
4. Goal area
5. Obstacles placement (hills, walls, mountains, etc.)

Looking at figure 1, we have an image of the world. The starting point of the agent is just below the white object (checkpoint) farthest down the image. Looking at the image, we can see that there are some smaller hills placed around the first two checkpoints. These hills were placed there to encourage the agent to ascend to a higher altitude rather than flying close to the ground where there is a higher risk of crashing into the ground. Farther up the world, there is a large mountain with a checkpoint on either side of the mountain. This mountain was placed there to teach the agent to maneuver the most optimal

path from one checkpoint to another with an obstacle in the way and to teaching the agent to avoid larger obstacles in general.

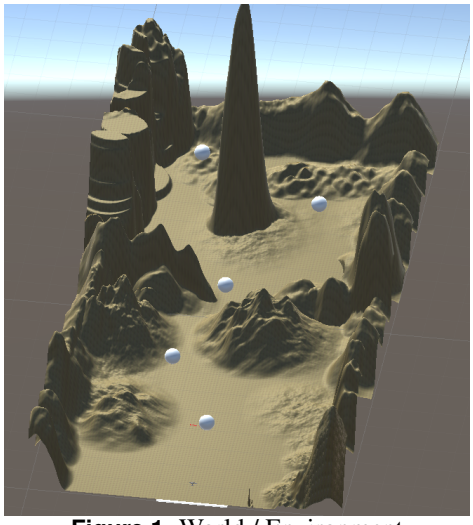

Figure 1. World / Environment

In figure 2, we have an illustration of one of the many checkpoints placed around the world. The design of the checkpoints are made to be simple and easy to adjust the size of. Having a larger checkpoint with more surface area means that the checkpoints will be easier to reach while when they are smaller makes them harder to reach. This teaches the agent to fly more precisely when the checkpoints are made to be smaller. The agent constantly checks the distance between the desired target and current position, once the distance has reached a normalized value of 0 it counts towards the agent having reached the checkpoint.

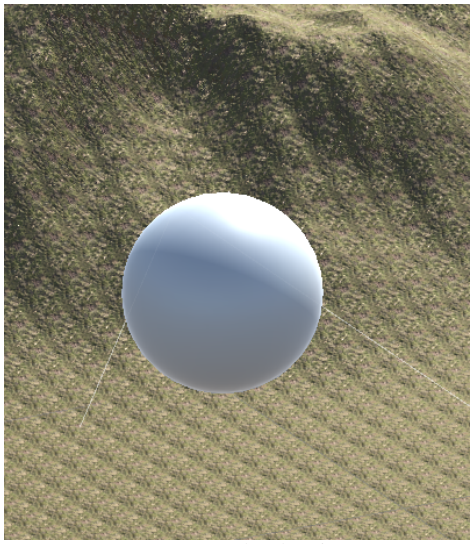

Figure 2. Checkpoints

3.2 The Agent

The model used for the agent is an old fighter jet taken directly from the free assets in the Unity asset store and can be seen in figure 3. Note that the agent is the plane itself and not a pilot that is sitting in the plane.

Figure 3. Plane Model

3.2.1 Actions

In order for the agent to be able to explore and interact with the world, a set of actions are necessary. The plane model is represented using a *rigid body* and moves by having forces applied to it. In this projects case, the force pushing the model forward is set to a constant and the only available actions the agent has are the horizontal and vertical movement.

The idea of having a constant force propelling the agent forward is to simplify the agent's actions so that the agent doesn't have control of how fast it is moving. So, rather than flying fast towards a checkpoint, the agent has to take the most optimal path.

To reduce the complexity of the the learning, the agent is only able to move vertically, but it rotates horizontally.

3.2.2 Observations and Sensors

For the agent to be able to perceive the world and make decisions based on the current state of the world, some form of observation is needed.

To do so, the agent is equipped with a set of perception sensors which identifies obstacles and distances to them. This is implemented using ray casting techniques.

An image of the agents sensors can be seen in figure 4 The red rays indicate that it perceives an obstacle at a certain distance from the agent, while the white rays indicate that no obstacles are close to the agent in that given direction.

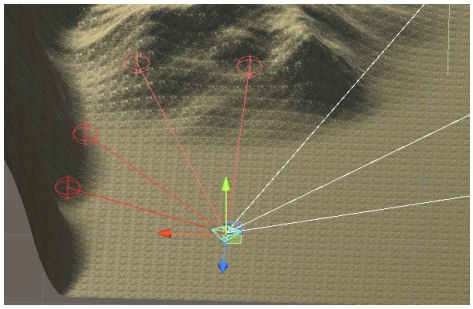

Figure 4. Sensors

3.3 The Reward System

For the agent to know which conducted actions are good and which are bad a reward system is implemented. Based on the actions done by the agent it will either receive a reward if a desired action is performed or a penalty if an undesired action, such as crashing, leaving the environment is performed. The reason behind a reward system is to make the agent learn a specific behavior, which in our case is to reach the end goal.

3.3.1 Step Penalty

A step penalty was introduced to put some time constraints on the agent. The step time-out is set to 500, and for each step a small penalty of (-1/max-steps) is given to the agent. If an agent reaches a set checkpoint, the step time-out will refresh to 500 steps, otherwise the episode will end.

3.3.2 Crash Penalty

As crashing is an undesired action, a penalty is given to the agent if this occurs. The penalty is set to -0.5. This penalty is given to the agent to sway it from conducting similar actions as it minimizes the agents reward.

3.3.3 Checkpoint Reward

Reaching a checkpoint indicates that the agent is on the right path and should therefore be rewarded. A reward of +1.0 is given to the agent when a checkpoint is reached.

3.3.4 Goal Reward

When the last checkpoint or end goal is reached the agent gets the highest reward as it has completed the course. This is to encourage the agent to continue to reach the end goal to accumulate as much reward as possible.

4. Result

The agent was trained for 6 million steps and learned to follow the pre-defined sequence of checkpoints relatively well (see full video[1]). Figure 5 together with figure 6 illustrates an increasing episode length in relation to an increasing cumulative reward.

Looking at figure 7 the value loss is steadily increasing the first 1.5 million steps when the increase rates are a bit lower until its peak of 0.096 at around 3.5 million steps. After the peak, it is slowly decreasing to 0.085.

[1] https://youtu.be/CplOO-hsRcI

Figure 5. Episode length on the y-axis and training steps on the x-axis.

Figure 6. Cumulative Reward on the y-axis and training steps on the x-axis.

Figure 7. Value loss on the y-axis and training steps on the x-axis.

5. Discussion

Looking at the results we can see that the agent seemed to learn steadily. Figure 5 shows an increased episode length during the time which is most likely related to the agent being able to stay alive longer and complete more checkpoints. As the cumulative reward is also increasing at a similar pace (see figure 6) strengthen that statement.

However, by looking at both figure 5 and 6, the model does not seem to be done training as neither of the graphs has converged yet. With a reward of 1.0 for each of the 5 checkpoints, the cumulative reward could for sure be higher than 1.8 even though the behavior of the agent seemed decent.

The value loss shown in figure 7, on the other hand, seems to be decreasing after about 3.5 million steps. The steps before

3.5 million with the increasing value loss show that the agent is learning, and a decrease usually indicates that the learning is stabilizing. To be completely sure we would have needed to train the agent further but were not able to due to time constraints.

In this project, we used a reward system that forced the agent to complete the course quickly using a small penalty for each step and a step time-out. It would be interesting to try a more positive approach. It can be hard to predict the behavior of such an approach, but if you would look at both human and animal psychology some suggest that the most effective approach is to use positive reinforcement [3]. Perhaps the outcome would be the same, but the learning time might differ. If it would result in faster learning, it might be more useful than saving a few seconds on completing the course. So far this is only speculation and would need an actual experiment to compare the results.

5.1 Problems

The project ended up in a lot of troubleshooting and debugging before the agent learned as intended. The reason was that since we initially made both the observation and reward system too complex before finding a more stable implementation as presented in the method. In hindsight, we should have started to make sure that the agent was able to learn something, before training overnight. As we went straight into the full implementation, a lot of problems we thought were in the reward system were in fact very basic errors in code and unity settings. Eventually, we changed the implementation to a simple horizontal 2D controller before moving on to both vertical and horizontal movements which made it easier to identify bugs.

Another thing we learned was to be patient. Initially, we tried to compensate for the slow learning rate by introducing additional rewards and penalties to help the agent move towards the checkpoints. As the ratio between penalty and reward was not balanced enough, it only caused undesirable behaviors. In the end, after discovering a few bugs in the code, the agent went through an exploration phase and quite quickly stabilized.

6. Conclusion

In conclusion, the agent was able to follow the sequence of checkpoints most of the time. Though it would still have been interesting to see the success rate when the cumulative reward was starting to converge after a bit more training. It would give us some more interesting observations of training. Overall our idea to use PPO for a self-flying airplane proved to work, and in the end even with quite a simple reward and observation system.

6.1 Future Work

Initially, we aimed to implement a few more things but were not able to due to the time spent debugging. As the training observations were mainly based on raycasts and the distance to the target, a fully trained agent would theoretically be able to find any randomly generated checkpoint. It would be interesting to see if this was possible, instead of only using a pre-defined sequence of checkpoints.

We did implement controllers that were a bit more realistic, introducing barrel rolls and gravity and turn-speed force. The extra controller features do, however, increase the complexity of the training significantly and we were not able to get stable training. This would make the agent fly more like an airplane, instead of it being limited to only move either horizontally or vertically.

Lastly, it would be interesting to see if the agent could be trained using another type of reward system. As mentioned in the discussion, we used a reward system that were mostly based on penalty to decrease certain behaviors. Perhaps another more positive behavior could have been trained using more rewards than penalties.

References

[1] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *Computing Research Repository*, abs/1707.06347, 2017.

[2] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *Computing Research Repository*, abs/1502.05477, 2015.

[3] SA McLeod. Bf skinner: Operant conditioning. *Retrieved September*, 9(2009):115–144, 2007.