1 Introduction

The environment for the task this project solves is one of the environments from the Python library called Gym, made by OpenAI. The environment creates an abstraction that provides easy manipulation with entities used to accomplish a given task. The selected environment is called Lunar Lander with discrete action space.

It is available at https://www.gymlibrary.dev/environments/box2d/lunar_lander/. To find a policy that will solve this task, Proximal policy optimization (PPO) was implemented. This policy was then used in experiments that were trying to improve reward function.

2 Task and Environment Describtion

The given task is to successfully land with this lander in a designated landing pad that is always at zero coordinates. Environment return state as a vector consisting of the lander's position, velocity, angle relative to the horizon, and flags that indicate if the lander's legs are in contact with the ground. All values except the last two are continuous. Lander can each time step perform 4 discrete actions: do nothing, fire left orientation engine, fire main engine, and fire right orientation engine. If the lander touches the ground with anything other than the mentioned legs, it crashes, and the episode ends as unsuccessful. The Environment itself can be considered deterministic. However, in each episode, the lander starts with a different force applied to it and the terrain around the landing pad changes.

Each time a step of an agent state is defined, it can perform an action that will return the next state, reward, a flag indicating a final state, and some other values not relevant to this environment.

3 Proximal Policy Optimization

A reinforcement learning algorithm called Proximal policy optimization (PPO) [1] is used to learn optimal policy. One of the main advantages of PPO is stability. The learning process does not tend to oscillate or vary that much during each algorithm run. PPO is from a family of actor-critic algorithms. In this case, both models are deep neural networks with the present environment state as their input. The difference between the actor and critic is that the actor model is trained to get optimal policy, and the critic is trained to evaluate actions made by the actor. Therefore, the output of the actor network is a probability function describing which actions most likely lead to the goal state, and the output of the critic network is a learned value function that is then used to get the advantage function. In the implemented version of PPO, an update of NN models is performed at the end of every episode. The loss function for the critic network is computed as:

$$L_{\text{critic}} = MSE = \frac{1}{N}\sum_{i=1}^{N}(R_i - V(s_i))^2$$

where N is step count in episode, R_i is the reward of ith step and $V(s_i)$ is critic predicted value of reward given state s_i of ith step. The most significant difference between PPO and other actor-critic algorithms is actor loss function. Firstly, a logarithm of the probability of taking action a in state s for each time step.

$$\log \pi_\theta(a_i|s_i) = \sum_{k=1}^{d}[\log(\pi_\theta(a_k|s_i)) \cdot \delta(a_k, a_i)]$$

where s_i and a_i are states and take action in time step i, and $\delta(a_k, a_i)$ is the Kronecker delta that works as a mask in this equation. The next step is to compute a ratio between new and old probability for a taken action r_i for all steps. This equation is usually transformed into the following expression:

$$r_i = \frac{\pi_\theta(a_i|s_i)}{\pi_{old}(a_i|s_i)} \approx e^{\log \pi_\theta(a_i|s_i) - \log \pi_{old}(a_i|s_i)}$$

The advantage is introduced to determine how to update gradients. In this project, Generalized Advantage Estimation is used.

$$A_t^{\text{GAE}} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

where γ is the discount factor, λ controls the trade-off between bias (advantage shifted) and variance (advantage noisy), r_t is a reward in time step t, and $V(s_t)$ is the value predicted by critic based on state s_t in time step t. PPO uses a clipping function to increase learning stability that cuts values further from sum or difference of 1 and parameter ϵ as presented below.

$$\text{clip}(r_i, 1-\epsilon, 1+\epsilon) = \begin{cases} 1-\epsilon, & \text{if } r_i < 1-\epsilon \\ r_i, & \text{if } 1-\epsilon \leq r_i \leq 1+\epsilon \\ 1+\epsilon, & \text{if } r_i > 1+\epsilon \end{cases}$$

Finally, the actor loss function is defined as:

$$L_{\text{actor}} = -\frac{1}{N} \sum_{i=1}^{N} \min\left(r_i A_i, \text{clipped}_r A_i\right)$$

where r_i is the ratio between new and old probability for a taken action, A_i is the advantage associated with state-action pairs, and clipped_r is the clipped value.

4 Reward Function

One of the objects of interest is the reward function, which defines how the outcome of actions in an environment is evaluated. In literature, it is almost always presented that the reward function comes from the environment, which can be partially confusing. The implementation of reward function in more complex problems is not straightforwardly defined by a given task or a model that describes transitions between states with given actions. In real-world problems, it is defined by the solver itself. It is also important to note that reward functions that can correctly distinguish between two almost same state-action pairs, which one is better than the other, can significantly improve the learning process. on the other hand, more complex reward functions can easily lead to undesirable behavior, such as policy will oscillate around an optimal solution or, on the contrary, will slow down the whole learning process.

The reward function in the baseline of the mentioned lunar lander evaluates the last action and current state or $R(a_{t-1}, s_t)$. It considers the distance to the landing pad, velocity, and angle if thrusters were fired up if the lander landed successfully or crashed. In the next subsections of this section, a few suggestions will be presented to improve reward function and increase learning speed.

4.1 Complex Reward for Crash Landing

One suggested improvement was defining negative rewards for crash landing more effectively. A baseline solution does not make a difference between a hard crash landing when the lander crashes to the ground with high velocity, wrong approach angle, and far away from the landing pad, and a small dilatation from optimal landing that results in a crash. The suggested negative reward is calculated as:

$$r_{\text{crash}} = -30(1+v) - 30(1+d) - 30(1+\theta)$$

where v is velocity, d is distance to landing pad and θ is angle. This function was later adjusted only to

$$r_{\text{crash}} = -80(1+v)$$

which show even more promising results in experiments (called version 2). Unfortunately, this function upgrade does not solve the problem of finalizing episodes because of the maximum step count per episode. The main reason is that the algorithm optimizes the maximum sum value of the reward function rather than solving the task. In this case, a crash landing comes with a significant negative reward, but hovering above the landing pad until the episode ends gives no reward. The actor decides to get a small negative amount based on firing thrusters rather than risking receiving a huge penalty for a crash landing.

4.2 Considering Time of Landing

The main reason for adjusting the time domain is to end as many episodes as possible before 1000 steps are taken. The time domain can be adjusted using a negative reward for firing thrusters because their usage correlates with the episode's length (especially in late phases).

The first set of experiments was to increase the negative reward for using thrusters by a fixed coefficient. This approach solves the problem only partially, mainly because it almost entirely suppresses learning in the early stage when it is important to learn to stabilize the lander and perform controlled descent rather than primarily optimize for thrust usage.

The second set of experiments tries to increase the negative reward for using thrusters by a coefficient related to episode number. Fuel and crash landing parts of reward computation are adjusted by coefficient. In the beginning, the coefficient is low, which results in possible high negative rewards for a crash landing and tiny rewards for using thrusters. During later phases of training, the coefficient increases the negative reward for thrusters with a positive reward for successful landing (to increase the chance of keeping on the main goal). This approach at the beginning is successful but is very sensitive to overlearning because, during later episodes, the lander starts to lose the ability to control the lander. In extreme cases, a coefficient can be so high that the lander will only learn not to use thrusters.

The third set of experiments tries to increase the negative reward for using thrusters by coefficient c related to a mean count of steps used in l last episodes

$$c = a \ \cdot mean(last\_episode\_steps(l))$$

where a is the coefficient that needs to be determined. Experimentally, it was found that suitable l seems to be 20, which compromises noise canceling and quick reaction to insufficiently long episodes, and $a = 0.00000001$ as it brings the maximal value of $c = 10$.

5 Experiments

A series of experiments has been conducted with suggested reward functions. To decrease noise in overall evaluation and obtain more relevant data for each reward function. Graph 1 shows visible progress during episodes that is common in evaluating different approaches in reinforcement learning. However, this visualization has its cons. on the x-axis, there are episodes, but these episodes are not the same in a count of steps for different implementations. Moreover, some approaches have shorter episodes than others because of their behavior. The second graph 2 is presented with overall steps instead of episodes to put a step count per episode into context.

The second issue comes from evaluation. In graphs 1 and 2 is as evaluation metric used baseline reward function. This means that every experiment with an improved reward function is optimized for a different function than is evaluated. This can benefit approaches with only slight changes in their reward functions. To overcome this problem and see different reward functions from a more fair perspective, Figure 3 is introduced. It represents the ratio of successful landings in the last 100 episodes.

To describe the legend more as it is presented in graphs: improved_crash_landing means that the reward function adopted a different approach to value crash landing that baseline, increase_fuel means that negative fuel reward was increased by a fixed coefficient, increase_fuel_episode refers to increasing the coefficient during training and increase_fuel_relative stands for increased fuel price according to a step count taken in of last 20 episodes.

It seems that the most promising are reward functions with improved crash landing rewards and relative increases based on the length of previous episodes.

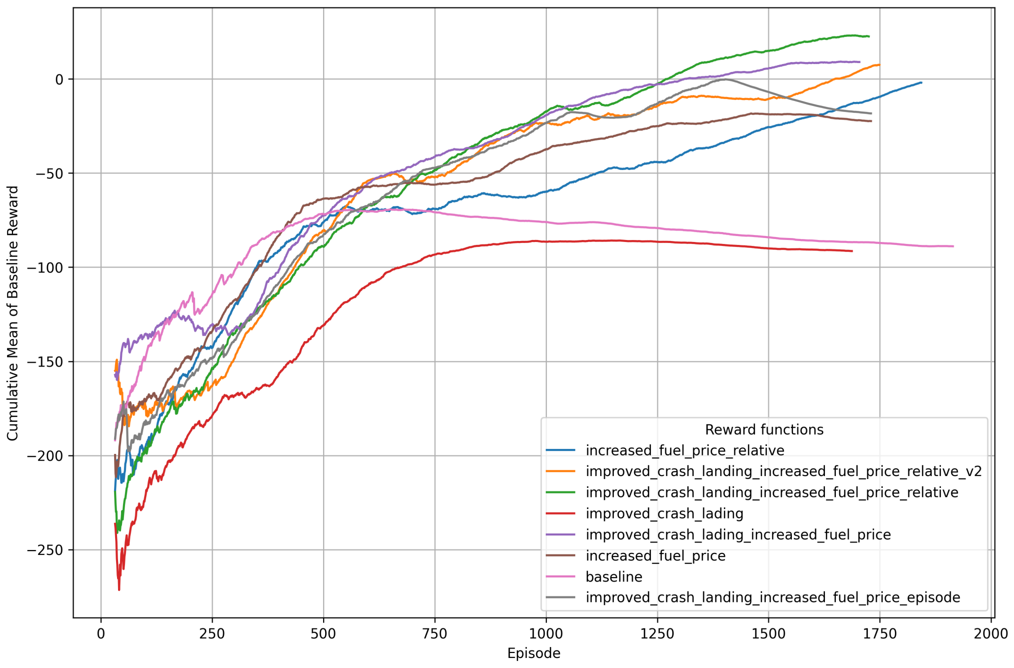

Figure 1: Performance of different reward functions in relation to episodes

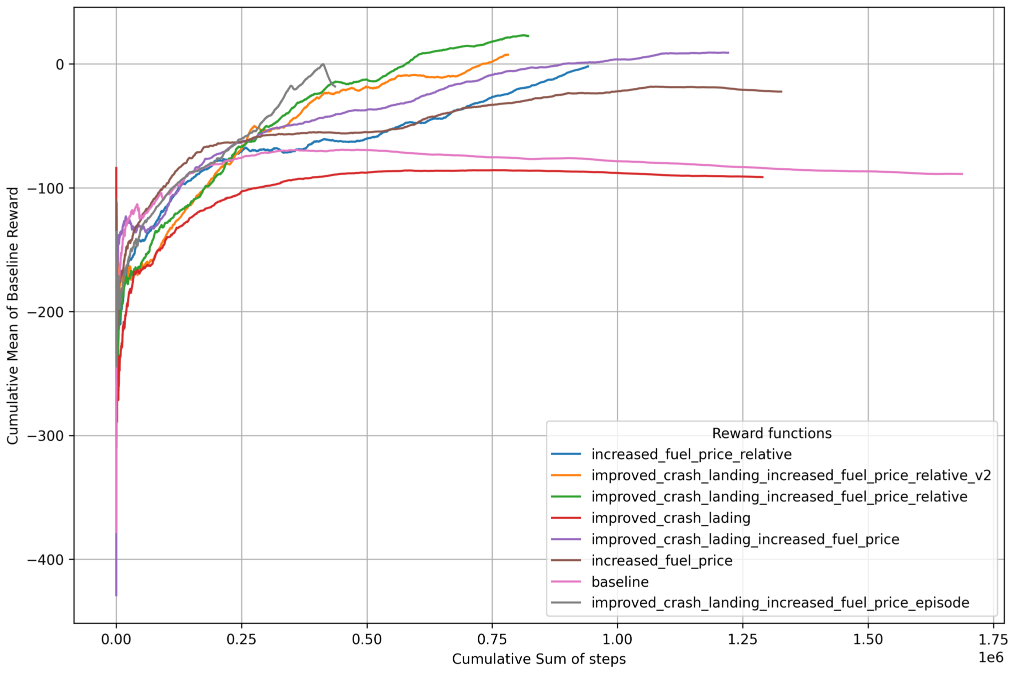

Figure 2: Performance of different reward functions in relation to step count

6 How to run

There are several important notes about managing the implementation of virtual machines. Firstly, there seems to be a problem with installing and using TensorFlow, which says ”core dumped” error. The PyTorch library implementation was added to make it runable in every case. To run the script:

```
$ sudo bash install.sh
```

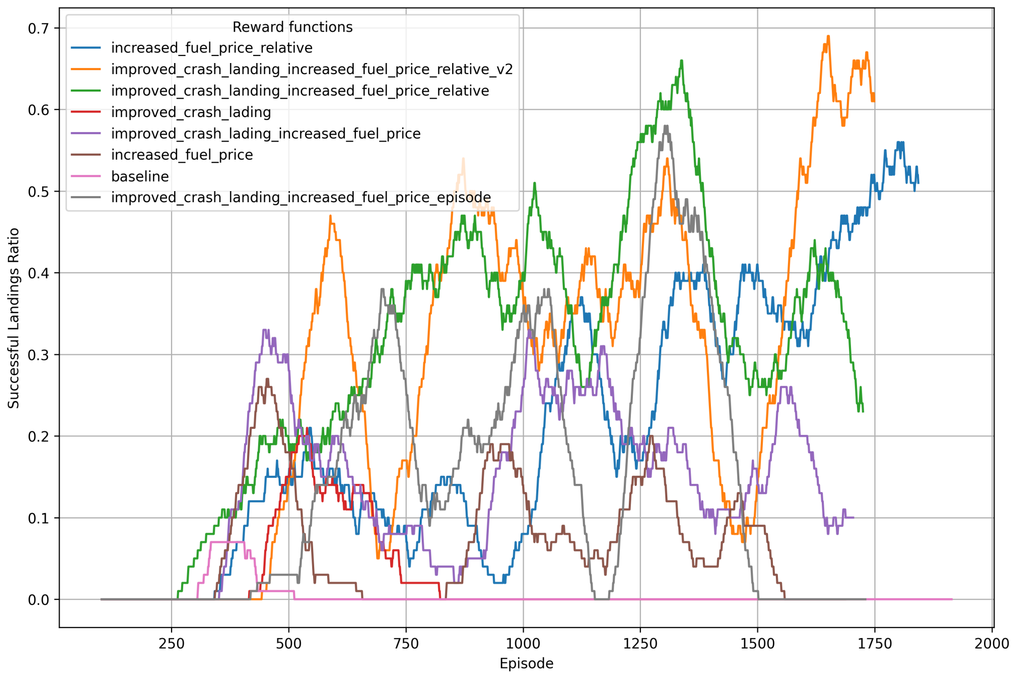

Figure 3: Ration of successful landings in latest 100 episodes

It is important to note that the script will try to install the PyTorch library that, for some reason, needed at least 8GB of memory on a virtual machine when it was tested. Otherwise, there was an error because of memory size. The script will install all necessary dependencies and create an environment. To activate the environment, use the following command:

```
$ source lunar_lander\bin\activate
```

To begin the learning process, run:

```
$ sudo python3 lander_torch.py --reward_model improved_crash_lading
```

This script will execute with default arguments and reward function improved_crash_lading. Other options are:

- `baseline`
- `improved_crash_lading`
- `improved_crash_lading_increased_fuel_price`
- `increased_fuel_price`
- `increased_fuel_price_relative`
- `improved_crash_landing_increased_fuel_price_episode`
- `improved_crash_landing_increased_fuel_price_relative`
- `improved_crash_landing_increased_fuel_price_relative_v2`

To customize the arguments, consult the help hint using:

```
$ sudo python3 lander_torch.py -h
```

The help menu also mentions the default values of arguments. Besides paths to output files, parameters include all hyperparameters used in PPO. The script can be stopped prematurely using `CTRL+C`. All results will be saved before exiting, ensuring no data or models are lost. To visualize the learned model, use the following script:

```
$ sudo python3 show_torch.py
```

This script will use the model specified in the arguments and create a short video of the current landing attempt. Part of the submitted repository is also a pre-trained model of the actor that can be run using the following command.

```
$ sudo python3 show_torch.py --actor_model actor_model_trained.pth
```

To generate all graphs used in the documentation, run the following script.

```
$ sudo python3 graph.py
```

Note that graphs will be very different from the results presented here because of the reduced dataset. To achieve the same results, download the full dataset from here. In case of any problems, do not forget to see help that every script has. To simply test functionality, run the following script.

```
$ sudo bash quick_test.sh
```

The quick test will create a short training session to create a policy that will be then visualized using video. Note that training is too short to learn how to land. For better results, the number of training episodes has to be increased to at least 2000.

7 Conclusion

This project was focused on the Lunar Lander Problem with a limit of 1000 steps per episode. The first part was implementing a RL algorithm called Proximal Policy Optimization with Generalized Advantage Estimation. Several different reward functions were implemented and evaluated to increase the overall speed of learning and overall results given by not fully trained suboptimal policies due to constrained time to learn. Because of high computational demand, experiments were taken on machines provided by Metacentrum. Experiments proved that properly adjusting reward function can lead to better results. Especially reward functions that considered the length of previous episodes proved to break the barrier of convergence that the baseline indicated.

It seems that the most promising are reward functions with improved crash landing rewards and relative increases based on the length of previous episodes.

Future work should focus on training for a more significant number of episodes per one training and more independent runs that will explore and possibly find a convergence line for presented reward functions and cancel noise still visible in visualized results. Different studies can focus on switching reward functions during one training session to converge faster to optimal policies or to overcome any possible barriers given by other problems.

Part of the submitted repository is also a pre-trained model learned using the most suitable reward function found. The model was trained in 3200 episodes using the most promising reward function. Despite not fully converging to the optimal policy, its results are more than satisfactory. Video of a successful landing can be seen in the video repository or, with a tiny bit of luck, generated.

8 Acknowledgement

Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

References

[1] Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A. and Klimov, O. Proximal Policy Optimization Algorithms. *CoRR*, 2017, abs/1707.06347. Available at: `http://arxiv.org/abs/1707.06347`.