Machine Learning - Project Report

Lukas Johannes Ruettgers Isaac Hong Zhang Jie Chenrui Guo

January 15, 2024

1 Introduction

The ability of humans to understand long-range causations between their behaviour and the environment and persuade goals that lie far ahead in a future is a key component to plan to move and manipulate in our world. To teach robots such a behaviour, we often formalise this problem in the field of Reinforcement Learning. Although is field of research achieved much attention in the last decades, it still remains a hard challenge to allow robots to abstract from specific experiences and achieve general navigation and manipulation capabilities. In this project report, we study some popular policy optimization methods that are widely used in practice to teach robots such behaviour. To that end, we first formally introduce the problem formulation of Reinforcement Learning and theoretically derive the core approaches. Afterwards, we motivate the development of actor-critic, policy gradient, and policy optimization methods and explain their functioning. We will then conduct experiments with these algorithms on OpenAI's LunarLander simulation environment to investigate how strongly the performance of these algorithms depends on each of the hyperparameters and conclude with an analysis of our results. The implementations, figures, and evaluation videos are publicly accessible at https://github.com/lukas-ruettgers/lunar-lander.

1.1 Our contributions

- Isaac was in charge of implementing PPO, defining the experiment setup on this agent, conducting the experiments, and reporting the experiment results to Lukas.
- Chenrui implemented a REINFORCE model, also defined the experiment setup on this model, conducted experiments on it, and reported the experiment results along with their evaluation to Lukas.
- The theoretical introduction and derivation of algorithms is attributable to Lukas. Besides the evaluation of the REINFORCE experiments in Section 4.2 that was written by Chenrui, Lukas independently wrote all other parts of the report. The details on the experiment setup and the figures, videos and other data from experiment were provided by Isaac and Chenrui respectively.

2 Reinforcement Learning

2.1 Problem Formulation

Reinforcement Learning addresses the problem of learning successful behaviour in an unknown environment. This environment is usually modeled as a stochastic process, namely a *Markov Decision process* (MDP) (S, A, r, P, γ). Every process-relevant detail of the environment is included in a state vector $s \in S$, where S is the set of all possible states the environment can be in. The agent has a state-independent set of actions $a \in A$ that describe his options to influence or interact with the environment. The current state s of the environment is usually affected by the action a of the agent, else the action would be useless. The *transition model* P describes how this action changes the environment by providing the new distribution of the states $P(s' \mid s, a)$ after the action was executed.

By the Markov assumption, the subsequent state only depends on the prior state and the action and not on states or actions that happened before. Further note that stochastic processes like the MDP regard time as evolving in fixed time steps $t = 0, 1, \dots$, such that the above term can also be written as $P(s_{t+1} \mid s_t, a_t)$. We however refrained from explicitly denoting the time index in the definition of the transition model P as it might suggest that the transition rule is time-dependent. However, we can consider time-independent transition rules without loss of generality since we can model any time-dependent variables in the state variable s itself. After an action a was executed and led to a new state s', the agent receives a reward $r(s, a, s')$ defined by the reward function $r : S \times A \times S \mapsto \mathbb{R}$. This is the only signal from which the agent can infer what behaviour is desirable. The reinforcement learning problem therefore models learning experience as the experience of state transitions (s, a, r, s'), from which the agent shall derive which actions lead in which states to what rewards. Naïvely, we might formulate the objective of the agent as maximizing the expected cumulative reward

$$J(a_0, \cdots) = \mathbb{E}_{s_{t+1} \sim P(s_{t+1} \mid s_t, a_t)} \left[\sum_{t=0}^{T-1} r(s_t, a_t, s_{t+1}) \right]$$

over a given *horizon* T. However, this objective is ill-defined for an infinite horizon $T = \infty$, where any recurring non-zero reward could accumulate to infinity and render the maximization problem useless. To account for this, a *discount factor* $\gamma \in (0, 1)$ is usually used to model the rate with which rewards become more irrelevant over time, such that the objective becomes

$$J(a_0, \cdots) = \mathbb{E}_{s_{t+1} \sim P(s_{t+1} \mid s_t, a_t)} \left[\sum_{t=0}^{T-1} \gamma^t r(s_t, a_t, s_{t+1}) \right].$$

A common choice is $\gamma = 0.99$, which means that after 100 steps, the reward experienced at the current 100th step is approximately 20 times less important than the very first reward. However, this problem formulation is still impractical, since an infinite horizon T will have infinite number of actions as optimization parameters. Instead, we generally describe the behaviour of an agent by a *policy* $\pi(a \mid s)$, which defines a probability distribution over the actions conditioned on each possible state. For a finite number of states and actions, this policy can also be regarded as a lookup table that stores the optimal action distribution in each state. In the case of continuous state domains, this approach renders infeasible and we are left with approximating this function by a set of finite parameters θ. Neural network architectures have proven to serve as accurate non-linear function approximators and are widely used in current Deep RL algorithms to approximate $\pi(a \mid s) \approx \pi_\theta(a \mid s)$. The objective function then finally writes as

$$\begin{aligned} J(\theta) &= \sum_{t=0}^{T-1} \gamma^t r(s_t, a_t, s_{t+1}) \prod_{i=0}^{t} \left(P(s_{i+1} \mid s_i, a_i) \pi_\theta(a_i \mid s_i) \right) \\ &= \mathbb{E}_{s_{t+1} \sim P(s_{t+1} \mid s_t, a_t), a_t \sim \pi_\theta(a_t \mid s_t)} \left[\sum_{t=0}^{T-1} \gamma^t r(s_t, a_t, s_{t+1}) \right]. \end{aligned}$$

In the following, we will write $\mathbb{E}_{s_t, a_t \sim \pi_\theta}[\cdot]$ for notational compactness and to steer the focus on the policy that is to be learnt, while the transition model p is given albeit unknown.

2.2 Algorithms and Approaches

2.2.1 Value-based methods

The popular methods in classical RL are often categorized as *value-based* and *policy-based* methods. Value-based methods consider the RL problem as a problem of learning the true value of a state $V(s)$ with regard to the expected cumulative reward that is obtainable from it. That is, we define

$$V(s) = \mathbb{E}_{s_t, a_t \sim \pi_\theta} \left[\sum_{t=0}^{T-1} \gamma^t r(s_t, a_t, s_{t+1}) \mid s_0 = s \right].$$

Similarly, we can further condition this value on the action chosen in this state and obtain a *state-action value*

$$Q(s,a) = \mathbb{E}_{s' \sim P(s'|s,a)} \left[r(s,a,s') + V(s') \right].$$

If such value functions could be accurately estimated, then an approximately optimal policy could be obtained by deterministically choosing the action that is expected to lead to the states with the highest rewards,

$$a^* = \arg\max_{a \in A} Q(s,a) = \arg\max_{a \in A} \mathbb{E}_{s' \sim P(s'|s,a)} \left[r(s,a,s') + V(s') \right], \quad \pi^*(a \mid s) = \begin{cases} 1, & a = a^* \\ 0, & a \neq a^* \end{cases}.$$

The above equality between $V(s)$ and $Q(s,a)$ with regard to optimality in a is known as the *Bellman equation* and paves thFRe way to solving the problem via Dynamic Programming, where an initial estimate $V^{(0)}(s)$ of each state's value is iteratively updated by empirically perceived rewards,

$$V^{(j+1)}(s) = \max_{a \in A} \mathbb{E}_{s' \sim P(s'|s,a)} \left[r(s,a,s') + V(s') \right].$$

In a similar fashion, $Q(s,a)$ is iteratively updated.

2.2.2 Policy-based methods

The above schematic algorithm requires executing each action in each state, while the final policy is defined at the end. An alternative approach might include policy behaviour directly from the beginning. That is, we do not update the value functions by executing each possible action and regard them as equally likely, but executing actions with a probability according to a current policy. Because the policy and value functions depend on each other, this paves the way to an iterative update of the policy and the value functions. In the *policy evaluation* step, we firstly update our value functions as

$$V_\pi^{(j+1)}(s) = \mathbb{E}_{s' \sim P(s'|s,a), a \sim \pi(s)} \left[r(s,a,s') + V(s') \right]$$

for a fixed number of steps or until a convergence criteria is satisfied. Then, we use the updated value functions for *policy improvement* and update $\pi \leftarrow \arg\max_{a \in A} Q_\pi(s,a)$. The repeated iteration over these two processes is called *policy iteration* and often serves as a template for more sophisticated methods. Note that the policy that is improved — the *target policy* — and the policy that is used for interaction with the environment in the policy evaluation step — the *behaviour policy* — can be different. This might be desirable to foster exploration during training, while our final policy should completely exploit the learned knowledge to make the optimal choices. Our current formulation uses the same greedy policy for both steps and hence constitutes an *on-policy* algorithm. But deviating from the target policy with a small probability $\varepsilon \in (0,1)$ and randomly choosing the next action to explore the environment will allow the behaviour policy to escape from local minima and widen its horizon of reward experiences in view of environments that are too high-dimensional or costly too access exhaustively. This *ε-greedy* behaviour policy is the most widely adapted example of an *off-policy* algorithm, but one could also re-use old policies to enrich the learning progress with prior experience. However, both of the formulations above are intractable for infinite state spaces, which we usually face in real-world problems, which requires heuristic approximations in practice.

2.2.3 Policy Gradient

One alternative viewpoint of the RL problem formulation is to directly regard it as a parametrized optimization problem

$$\theta^* = \arg\max_\theta J(\theta) = \arg\max_\theta \mathbb{E}_{s_t, a_t \sim \pi_\theta} \left[\sum_{t=0}^{T-1} \gamma^t r(s_t, a_t, s_{t+1}) \right].$$

To simplify the following notation, we introduce the term *trajectory* $\tau = (s_0, a_0, \ldots, s_T)$ to describe an entire state-action sequence and abbreviate the probability of a trajectory as

$$P_{\pi_\theta}(\tau) = P(s_0) \prod_{t=0}^{T-1} \pi_\theta(a_t \mid s_t) P(s_{t+1} \mid s_t, a_t),$$

where $P(s_0)$ is the prior probability distribution of initial states, and define the reward of a trajectory as

$$r(\tau) = \sum_{t=0}^{T-1} r(s_t, a_t, s_{t+1}).$$

This way, we streamline the definition of the objective function to $J(\theta) = \mathbb{E}_{\tau \sim P_{\pi_\theta}}[r(\tau)]$. However, note that this is just a notational simplification. The semantic equality only holds if we multiply the probability of the trajectory from s_0 to s_t with the reward *from* t to T and not from 0 to T, where $0 \leq t \leq T-1$. This imperfection does not matter for the following theoretical analysis in which we can regard the rewards as constants that have been collected during execution in the environment. To maximize $J(\theta)$ using gradient ascent, we first have to derive how to compute the gradient; or more specifically, how we compute the gradient of $\pi_\theta(a_t \mid s_t)$, since all other terms are independent of θ. Owing to the chain rule, we have the equality

$$\pi_\theta(a_t \mid s_t) \cdot \nabla \log\left(\pi_\theta(a_t \mid s_t)\right) = \pi_\theta(a_t \mid s_t) \cdot \frac{1}{\pi_\theta(a_t \mid s_t)} \cdot \nabla \pi_\theta(a_t \mid s_t) = \nabla \pi_\theta(a_t \mid s_t), \quad (PG)$$

which allows us to obtain the Policy Gradient Theorem

$$\begin{aligned}
\nabla J(\theta) &= \nabla \mathbb{E}_{\tau \sim P_{\pi_\theta}}[r(\tau)] \\
&\overset{\text{Def.}}{=} \nabla \int_\tau P_{\pi_\theta}(\tau) r(\tau)\, d\tau \\
&\overset{\text{Def.}}{=} \nabla \int_\tau \sum_{t=0}^{T-1} \gamma^t r(s_t, a_t, s_{t+1}) \prod_{i=0}^{t} \left(P(s_{i+1} \mid s_i, a_i)\pi_\theta(a_i \mid s_i)\right)\, d\tau \\
&\overset{\text{sum rule}}{=} \int_\tau \sum_{t=0}^{T-1} \gamma^t r(s_t, a_t, s_{t+1}) \nabla \prod_{i=0}^{t} \left(P(s_{i+1} \mid s_i, a_i)\pi_\theta(a_i \mid s_i)\right)\, d\tau \\
&\overset{\text{prod. rule}}{=} \int_\tau \sum_{t=0}^{T-1} \gamma^t r(s_t, a_t, s_{t+1}) \sum_{j=0}^{t} P(s_{j+1} \mid s_j, a_j) \left(\nabla \pi_\theta(a_j \mid s_j)\right) \prod_{i=0, i\neq j}^{t} \left(P(s_{i+1} \mid s_i, a_i)\pi_\theta(a_i \mid s_i)\right)\, d\tau \\
&\overset{(\text{PG})}{=} \int_\tau \sum_{t=0}^{T-1} \gamma^t r(s_t, a_t, s_{t+1}) \sum_{j=0}^{t} \left(\nabla \log \pi_\theta(a_j \mid s_j)\right) \prod_{i=0}^{t} \left(P(s_{i+1} \mid s_i, a_i)\pi_\theta(a_i \mid s_i)\right)\, d\tau \\
&\overset{\text{Def.}}{=} \mathbb{E}_{\tau \sim P_{\pi_\theta}} \left[\sum_{t=0}^{T-1} \gamma^t r(s_t, a_t, s_{t+1}) \sum_{j=0}^{t} \nabla \log \pi_\theta(a_j \mid s_j) \right] \\
&\overset{\text{rearrange}}{=} \mathbb{E}_{\tau \sim P_{\pi_\theta}} \left[\sum_{j=0}^{T-1} \nabla \log \pi_\theta(a_j \mid s_j) \sum_{t=j}^{T-1} \gamma^t r(s_j, a_j, s_{j+1}) \right] \\
&\overset{T \to \infty}{=} \mathbb{E}_{\tau \sim P_{\pi_\theta}} \left[\sum_{j=0}^{\infty} \nabla \log \pi_\theta(a_j \mid s_j) Q_{\pi_\theta}(s_j, a_j) \right].
\end{aligned}$$

The benefit of this transformation is that we reduced computing the gradient of an expectation value over the rewards to the expectation value of a policy gradient term. Even though the rigorous exact computation remains intractable for infinite or high-dimensional state and action spaces, we can avail to common strategies in Inference Statistics to estimate this expectation value given only few samples of the entire population of possible trajectories. That is, one could for example collect N trajectories at each gradient ascent step and approximate the estimation value of the log terms as the arithmetic

mean of the log terms over these N trajectories, as is done in classical *Monte Carlo* RL. This is the approach that the REINFORCE algorithm takes, which belongs to one of the first algorithms in the category of policy gradient methods. To further alleviate the computation of the log term itself, REINFORCE approximates $\pi(a \mid s)$ as a normal Gaussian probability distribution. While the mean of this distribution is estimated using a neural network $\mu \approx f_\theta(s)$, the covariance matrix is often heuristically fixed as Σ. This way, we have

$$\pi_\theta(a_t \mid s_t) = \frac{1}{\sqrt{(2\pi)^n |\Sigma|}} \exp\left(-\frac{1}{2}(f_\theta(s_t) - a_t)\Sigma^{-1}(f_\theta(s_t) - a_t)\right), \text{ and consequently}$$

$$\log \pi_\theta(a_t \mid s_t) = \log\left(\frac{1}{\sqrt{(2\pi)^n |\Sigma|}}\right) - \frac{1}{2}(f_\theta(s_t) - a_t)\Sigma^{-1}(f_\theta(s_t) - a_t).$$

Because the first term is constant with regard to θ, this finally yields us the gradient

$$\nabla \log \pi_\theta(a_t \mid s_t) = -\Sigma^{-1}(f_\theta(s_t) - a_t)\frac{\partial f_\theta}{\partial \theta}.$$

As all activation functions in today's neural network architectures are differentiable, $\frac{\partial f_\theta}{\partial \theta}$ is simply obtained for each weight θ_i by the back-propagation algorithm. The pseudocode for REINFORCE is depicted below in Figure 1.

Algorithm 1 REINFORCE

Initialize: Neural network π_θ with parameters θ, learning rate α, covariance matrix Σ, batch size N.

while not converged **do**
  Collect N trajectories $(s_0, a_0, r_0, \ldots, s_T)$ by executing π_θ and store them in a buffer B.
  for all trajectories τ in B **do**
    for all transitions $(s_t, a_t, r_t) \in \tau$ **do**
      Forward s_t through π_θ to obtain the distribution $\pi_\theta(a_t \mid s_t)$.
      Track the gradient history to obtain $\nabla \log \pi_\theta(a_t \mid s_t) = -\Sigma^{-1}(f_\theta(s_t) - a_t)\frac{\partial f_\theta}{\partial \theta}$.
      Run the BP algorithm to update $\theta \leftarrow \theta + \alpha \cdot \nabla \log\left(\pi_\theta(a_t \mid s_t)\right) r_t$.
    end for
  end for
end while

Figure 1: REINFORCE pseudo code algorithm for a Gaussian distribution policy network.

As for normal Monte-Carlo RL methods, the estimation on long-horizon trajectories exhibit a huge variance, because the domain of possible state-action combinations grows exponentially in the horizon T. On the other side, the *Temporal Difference* update

$$V^{(j+1)}(s) = V^{(j)}(s) + \alpha\left(r(s, a, s') + \gamma V^{(j)}(s') - V^{(j)}(s)\right),$$

which considers only one transition with empirical reward in the update of the value function, has a low variance. But it has a larger bias, which comes from the larger weight estimate of the value $V(s_{t+1})$ of the subsequent state, which was only discounted once by the discount rate γ. To mitigate the variance while preserving a low bias, an unbiased baseline can be subtracted from the rewards. The value function $V_{\pi_\theta}(s)$ turns out to be such a good baseline, resulting in the adjusted term

$$J(\theta) = \mathbb{E}_{\tau \sim P_{\pi_\theta}}\left[\sum_{j=0}^{\infty} \nabla \log \pi_\theta(a_j \mid s_j)\left(Q_{\pi_\theta}(s_j, a_j) - V_{\pi_\theta}(s)\right)\right].$$

This term is also known as the *advantage function* $A_{\pi_\theta}(s_t, a_t) := Q_{\pi_\theta}(s_t, a_t) - V_{\pi_\theta}(s)$, which measures the advantage, when taking action a_t over alternative actions in s_t and continuing to act by the

policy π_θ. If the rewards perceived in this trajectory are high, then a_t truly exhibits an advantage over the average action, because $V(s_t)$ is defined as the *expectation value* of cumulative rewards. To approximate this advantage function with a good trade-off between Temporal Difference with low variance and Monte Carlo with low bias, one could theoretically choose any $1 \leq n \leq T$, accumulate the empirical rewards until step n and cap off with the prior estimate of the value function at the n-th state,

$$V^{(j+1)}(s_t) = A_n(s_t, a_t) := \sum_{i=0}^{T-1} \left(\gamma^i r(s_{t+i}, a_{t+i}, s_{t+i+1})\right) + \gamma^n V^{(j)}(s_{t+n}) - V^{(j)}(s_t).$$

This term $A_n(s_t, a_t)$ is often referred to as the n-step approximation of the *advantage function*. The *Generalized Advantage Estimation* combines all these terms into one and steers their importance decay with one hyperparameter $\lambda \in (0, 1)$, culminating in

$$A_{GAE}(s_t, a_t) = \frac{1}{1-\lambda} \sum_{n=1}^{T} \lambda^n A_n(s_t, a_t) = \frac{1}{1-\lambda} \sum_{i=0}^{T} (\lambda\gamma)^i \left(r(s_{t+i}, a_{t+i}, s_{t+i+1} + \gamma V(s_{t+i+1}) - V(s_{t+i})\right).$$

2.2.4 Proximal Policy Optimization

There are also numerous other optimization techniques to improve the objective function $J(\theta)$. To unravel this bouquet of opportunities, consider an alternative parametrization θ'. The *Performance Difference Lemma*, whose proof follows a similar spirit as the one for the Policy Gradient Theorem, states that

$$J(\theta') - J(\theta) = \mathbb{E}_{\tau \sim P_{\pi_{\theta'}}} \left[\sum_{t=0}^{\infty} A_{\pi_\theta}(s_t, a_t)\right].$$

Intuitively, this means that θ' is desirable over θ if the new policy $\pi_{\theta'}$ executes those actions with a higher probability that are advantageous even if we follow the old policy π_θ in the subsequent states. Unfortunately, the evaluation of this term requires an approximation of the distribution of states and actions under the very θ' that we are trying to optimize. Naïvely, this would require to empirically estimate the distribution under $P_{\pi_{\theta'}}$ with a large number of trajectories for each θ' we want to consider as a new parameter candidate, which imposes a severe computational limitation on the number of θ's that one can explore in each update step. Instead we can constrain θ' to remain close to θ with regard to P_{π_θ}. Then approximating

$$\mathbb{E}_{\tau \sim P_{\pi_{\theta'}}} \left[\sum_{t=0}^{\infty} A_{\pi_\theta}(s_t, a_t)\right] \approx \mathbb{E}_{\tau \sim P_{\pi_\theta}} \left[\sum_{t=0}^{\infty} \frac{\pi_{\theta'}(a_t \mid s_t)}{\pi_\theta(a_t \mid s_t)} A_{\pi_\theta}(s_t, a_t)\right]$$

will not exhibit a large distributional shift in the likelihood of states. *Trust Region Policy Optimization* formulates this as a constrained optimization problem and maximizes

$$J(\theta') = \mathbb{E}_{\tau \sim P_{\pi_\theta}} \left[\sum_{t=0}^{\infty} \frac{\pi_{\theta'}(a_t \mid s_t)}{\pi_\theta(a_t \mid s_t)} A_{\pi_\theta}(s_t, a_t) - \delta \,\mathrm{KL}\left(\pi_\theta(a_t \mid s_t) || \pi_{\theta'}(a_t \mid s_t)\right)\right],$$

which can be solved by a conjugate gradient algorithm that however requires second-order derivatives to approximate the penalty KL divergence term. Instead, *Proximal Policy Optimization* proposes a first-order approximation

$$J(\theta') = \mathbb{E}_{\tau \sim P_{\pi_\theta}} \left[\sum_{t=0}^{\infty} \min\left(1 + \varepsilon, \frac{\pi_{\theta'}(a_t \mid s_t)}{\pi_\theta(a_t \mid s_t)}\right) A_{\pi_\theta}(s_t, a_t)\right]$$

that clips overly advantageous deviations from the current policy π_θ and therefore evokes a more stable, stepwise optimization procedure. Since the estimation of the advantage function requires a state-value function V, a separate neural network is trained as *critic*. Together with the critic loss and a term that encourages entropy on the action distribution of $\pi_{\theta'}(a_t \mid s_t)$, the above $J(\theta')$ constitute

the loss function $J(\theta', \psi')$. The original paper trains multiple agents in parallel (2), but since it is no specific feature of PPO, we omitted it in the pseudocode in Figure 2.

Algorithm 2 Proximal Policy Optimization

Initialize: Actor network π_θ, value network V_ψ, clip offset ε, scaling coefficients c_1, c_2 for entropy and value loss, batch size N, training epochs K, minibatch size M.

while not converged **do**
  Collect T transitions (s_t, a_t, r_t, s_{t+1}) by executing π_θ and store them in a buffer B.
  Compute advantage estimators $\hat{A}^{(i)}_{\pi_\theta}$ for each transition $1 \leq i \leq T$.
  for epoch $1 \leq j \leq K$ **do**
    for all transitions (s_t, a_t, r_t, s_{t+1}) in minibatch of size M **do**
      Optimize θ' and ψ' with regard to $J(\theta', \psi')$.
    end for
    Update $\theta \leftarrow \theta', \psi \leftarrow \psi'$
  end for
end while

Figure 2: PPO pseudo code algorithm.

3 Experiment Setup

3.1 Experiment Objective

In this experiment, we want to investigate the performance of both PPO and REINFORCE on the `LunarLander-v2` environment provided by OpenAI's `gymnasium` package. We seek to understand how sensitive the model's performance is to some hyperparameters, which are concretized in the following. For PPO, we alternate over

- three different reward configurations, which mainly deviate in sparsity,
- the learning rates $\alpha = 0.01, 0.001, 0.0003$ to validate whether faster rates than the low default learning rate of 0.0003 still achieve stable training progress,
- the discount factors $\gamma = 0.5, 0.7, 0.999$ to compare far more shortsighted with the default, quite longsighted agent with $\gamma = 0.999$, and
- the clip offsets $\varepsilon = 0.1, 0.2, 0.3$ as suggested by the original paper (2) to impose different restrictions on the policy's freedom to profit from its advantages.

The experiments for PPO were not done in parallel, but sequentially, and the best hyperparameters in each earlier experiment were fixed for the subsequent experiments. Since we already covered the discount factor in the experiments for PPO and did not expect to obtain new enriching results from repeating another comparison of different discount rates, we choose to focus on other hyperparameters in the experiments conducted with REINFORCE. Because the experiments on PPO and REINFORCE are not done in parallel but sequentially, we were able to integrate knowledge from the PPO experiments into the experiment setup of the REINFORCE agent. Specifically, we observed that the environment sometimes does not provide the termination signal although the spacecraft landed on the goal area. The agent then hovers around by activating the side engines until the state is eventually accepted by the environment as goal state. For that reason, we capped the maximum trajectory length on our own and defined a hyperparameter `time_threshold`, which we will refer to as T in the following. After T steps, the run will be regarded as terminated even if the environment does not set the `is_done` variable to True. If the agent did not finish its execution after this number of time steps, it also received a final negative reward of -50 to encourage him to avoid situations that may evoke such bugs in the environment. To assess how such an early termination influenced the environment,

we consider different values in the experiments too. We were concerned to not provide our agent a too strong incentive to quickly end the execution, because erratic crashes would then become more acceptable in comparison to the more careful and long-horizon interplay of engine activations that is required to successfully approach and land in the goal area. Over and above, the learning rate that proved most successful in the PPO optimization failed completely to provide a stable and convergent parameter update in the REINFORCE case. For that reason, we considered the choice of an adequate learning rate again an open question for REINFORCE, which makes sense since the network architectures and limitations are of different nature. As smaller learning rates were suggested from reference implementations, we considered learning rates in the 10^{-4} domain.

For REINFORCE, we hence alternate over

- the learning rates $\alpha = 0.0003, 0.0005, 0.0007$,
- the hidden layer size $H = 64, 128, 256$ of the two hidden layers in the MLP to get a feeling of the required model size to achieve a good performance in the environment, and
- the early termination threshold $T = 20, 30, 40$.

Again, we will conduct the experiments sequentially and continue the subsequent experiments with the hyperparameter that proved best among the investigated values. This eventually requires us to conduct only runs for seven different hyperparameter combinations.

3.2 LunarLander Environment

The lunar environment simulates the problem of landing a spacecraft on a platform. The observation space $(q_x, q_y, \vartheta, v_x, v_y, \omega, \ell, r)$ is quasi-continuous and comprises both the robots coordinates, azimuthal angle, linear and angular velocities and two boolean variables that reflect if the left or right leg contact point touched the land. The actions space is finite and merely continue four actions:

- Action 0: Do nothing,
- Action 1: Fire left orientation engine,
- Action 2: Fire the main engine,
- Action 3: Fire right orientation engine.

3.3 Rewards

We use different reward configurations during training. The first is the default reward configuration, which is fine-tuned and hence provides an accurate, dense signal towards the goal region. Specifically, the reward after an action

- is increased/decreased the closer/further the lander is to the landing pad,
- is increased/decreased the slower/faster the lander is moving,
- is decreased the more the lander is tilted (angle not horizontal),
- is increased by 10 points for each leg that is in contact with the ground,
- is decreased by 0.3 points each frame the main engine is firing,
- is decreased by 0.03 points for each side engine firing,
- is increased by 100 points for landing safely, and
- is decreased by 100 points for crashing.

The reward configuration in the task description of our course project instead awards 100 points for each leg making ground contact and thus incentives safe landing. It also slight penalizes usage of the main engine to promote fuel efficiency. This approach allows for a nuanced modification of the reward system, encouraging behaviors like safe landing and fuel efficiency while still adhering to the game's fundamental objectives. We refer to this reward configuration as the *custom* reward, which — for completeness —

- is increased by 100 points for each leg that is in contact with the ground,
- is decreased by 0.3 points each frame the main engine is firing,
- is increased by 100 points for landing safely, and
- is decreased by 100 points for crashing.

To firstly showcase the relevance of the omitted rewards and secondly study how the higher leg contact reward affects the learning performance, we finally extend the custom reward by all terms that were not included. For the terms that are already included, we leave the values the same and do not adjust them to the values from the default reward configuration. Specifically, this leads to a *combined* reward that deviates only from the default reward in assigning not 10 but 100 points for each leg contact.

3.4 Implementation

3.4.1 PPO

To ensure a correct implementation of PPO, we availed to the PPO implementation of `Stable Baselines 3`. Each model is trained for 1,500,000 iterations, which amounts to 367 policy updates and evaluations. While the model was only trained on one vectorized environment for simplicity, it was evaluated on 10 different environment configurations to obtain a more general evaluation score. Evaluation of the model's mean reward was done after every four policy updates. After training has finished, the model's mean reward was evaluated one final time. The implementation is available in a Jupyter Notebook which was supplied along with the report. The CustomCallback subclass of the BaseCallback class provides the mean rewards in training and evaluation mode for each episode, which were collected and finally plotted together.

3.4.2 REINFORCE

We implemented the REINFORCE algorithm using Tensorflow and chose the policy network as a 3-layer MLP with two hidden layers of default size 256 each. We employed ReLU as activation function in each neuron. Instead of predicting a Gaussian distribution as in the example provided above, which applies to the more general case of continuous action spaces, our environment only included four actions. We hence decided to let the network predict a categorical distribution over the four possible actions by applying a softmax layer at the output layer of size 4. We fixed optimized the network with Adam and a learning rate of $\alpha = 0.0002$. Moreover, we decided to fix discount factor $\gamma = 0.99$, as it proves successful in many environments, and chose a batch size of $N = 5000$ episodes.

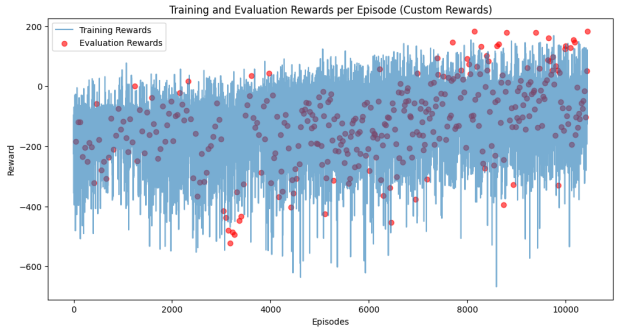
3.4.3 Reproducibility

We tried to force torch and CUDA to use only deterministic algorithms by setting the environment variable `CUBLAS_WORKSPACE_CONFIG` according to the API Reference Guide of cuBLAS (1). However, we weren't able to perform any operations on the GPU if we set this variable. Since some of the models themselves already exhibited variance, we decided to run the experiments on a CPU and not a GPU to clearly distinguish variance that emerges from GPU operations from the variance inherent to the model architecture. In the experiments with Stable Baselines 3 and our implementation using tensorflow, we however still observed variances in the learning curves of agents with the exact same

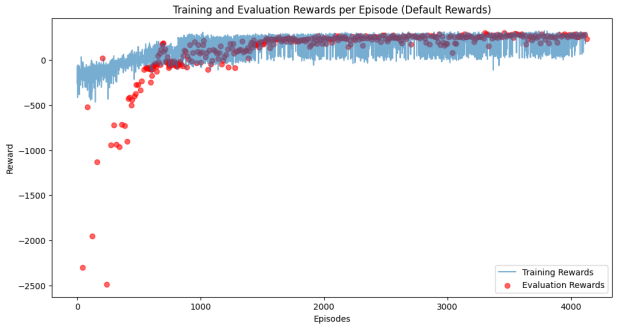
configuration. However, these variances were small and eventually did not perturb the statistical significance of our results.

4 Experiment Results

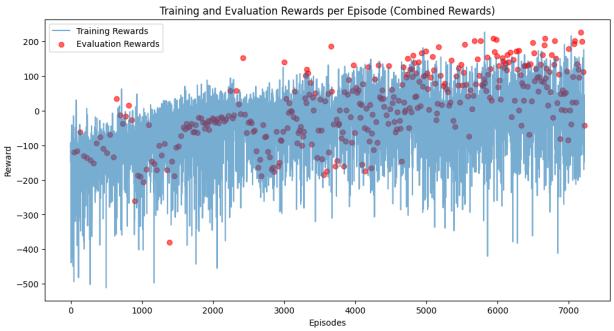
4.1 PPO



(a) Project-custom rewards



(b) Default rewards

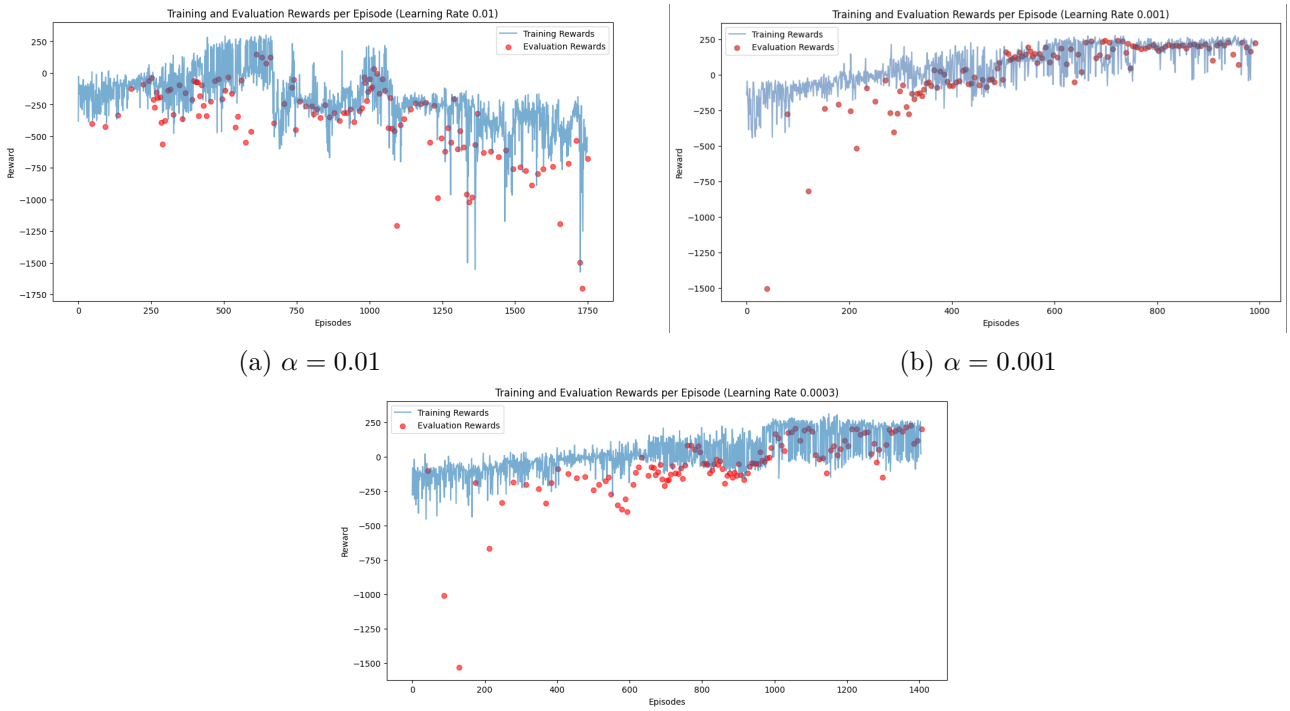


(c) Combined rewards

Figure 3: Training and evaluation rewards of PPO for different reward configurations, namely the custom reward configuration proposed in the course project description, the default reward configuration of the environment, and a combination of both.

In the custom reward configuration, the agent will only receive any positive signal at the very end of the landing process, that is when a leg touches the ground and the overall landing process was successful. Before, it will only receive negative rewards for activating the main engine. Moreover, there is neither punishment nor reward for using the side engines. The agent actions will therefore obtain two degrees of freedoms on these two actions and result in uncontrolled, randomized behaviour with overwhelming probability. This exacerbates the ability to learn stable successful long-horizon behaviour, as the intermediate steps taken do not directly receive any feedback. Conversely, the default reward configuration provides a dense training signal that rewards the agent for any improvement in the distance between the landing unit and the spacecraft, which quickly steers the agent into the region of interest where he then perceives rich reward experience. As the comparison in Figure 3a demonstrates, the custom rewards exhibit a huge variance, because the agent lacks reward signals that teach him a fine-granular usage of his engines and will not acquire a stable behaviour. Conversely, the agent that perceives the default rewards converges faster and more stable to successful behaviour (Fig. 3b), although its variance is only slightly smaller than the variance of the agent with the custom rewards. The quite negative rewards at the first evaluations scale the figure to a far larger range and creates the wrong illusion that the variance decreased drastically. For the combined model where only the positive reward for leg contact was enlarged from 10 to 100, the variance remains similar to the agent with the default rewards (Fig. 3c). However, it fails to live up to the average final reward of around 150 of the

default reward agent, since it oscillates rather around 110 during the final evaluations. For each of the three models, we have provided evaluation videos in the `ppo_eval_videos` folder. While the model with the default rewards succeeds in all final evaluation runs, the model with combined rewards only succeeds in half of them and the model with the custom rewards in almost none. The evaluation video of the model with custom rewards illustrates the uncontrolled usage of the side engines stressed in the experiment setup section. Although the model even has a quite fortunate initial force applied that pushes it directly to the landing area without requiring usage of the side engines, the model starts to activate the side engines in a seemingly uncontrolled way and drafts to the left. What's more, the model with the combined rewards seems to continue using the side engines although it has already landed in the goal area, as our example evaluation video illustrates. This behaviour is particularly odd since the model even receives a small punishment for using the side engines. We believe it might be due to a bug in the environment which does not regard the agent's position as goal state and refrains from terminating the run, such that our model tries to trigger the termination by hovering over the landing area. Nevertheless, this behaviour did not occur for the model with the default rewards, which only changed in the smaller reward scale for leg contact with the ground. This proves how brittle the reward function can become from small deviations from a fine-tuned superposition and underscores the importance of providing a dense reward signal to stabilise the optimization procedure. For the following, we hence continued with the default reward configuration.



(a) $\alpha = 0.01$

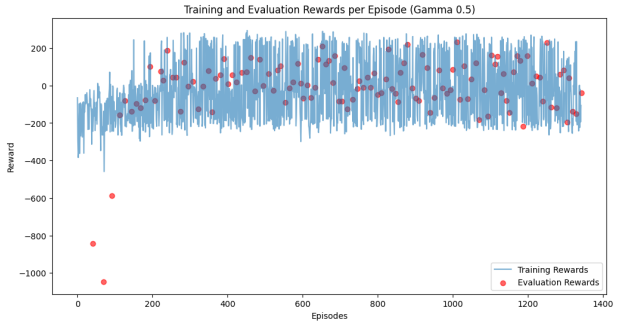
(b) $\alpha = 0.001$

(c) $\alpha = 0.0003$

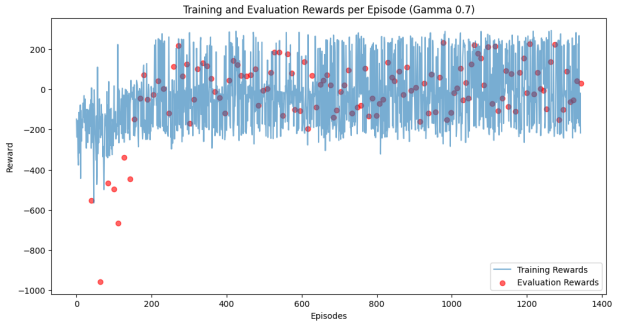
Figure 4: Training and evaluation rewards of PPO for different learning rates $\alpha = 0.01, 0.001, 0.0003$.

The default learning rate of $\alpha = 0.0003$ slowly approaches a sound mean reward (Fig. 4c), but interestingly exhibits more variance during training and evaluation than the agent with the slightly larger learning rate of $\alpha = 0.001$ (Fig. 4b). Originally, we expected the opposite behaviour, since a smaller learning rate should slow down the overall parameter change and hence the change in behaviour. What we did not consider in the beginning is that the behaviour itself can exhibit a large variance in performance due to imperfect actions. We hypothesize that the slightly larger learning rate allows the optimization process to easier escape from local minima and converge faster to stable and successful performance. Especially at the end, both the training and evaluation mean reward achieve an unparalleled stability in a high value range of more than 150 (Fig. 4b). On the contrary,

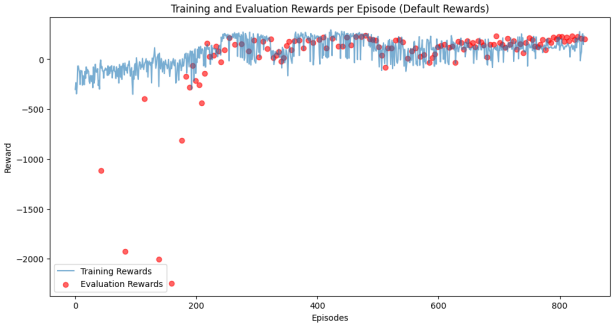
the slower parameter changes for $\alpha = 0.0003$ might prolong the required training time for culminating in successful behaviour. This reminds us of the importance of choosing the learning rate not too small as it might be less robust to a function landscape with many local minima. On the other hand, Fig. 4a illustrates that convergence can not be guaranteed for higher learning rates and in our case eventually leads to divergence. The agent fails to acquire any improvement in the policy and never settles down in a good solution region. We therefore continue our experiments with $\alpha = 0.001$.



(a) $\gamma = 0.5$



(b) $\gamma = 0.7$



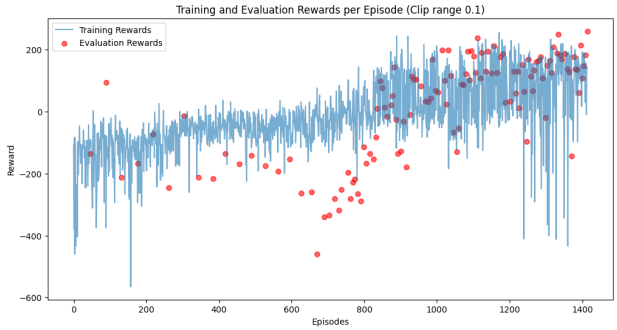
(c) $\gamma = 0.999$

Figure 5: Training and evaluation rewards of PPO for different discount rates $\gamma = 0.5, 0.7, 0.999$.

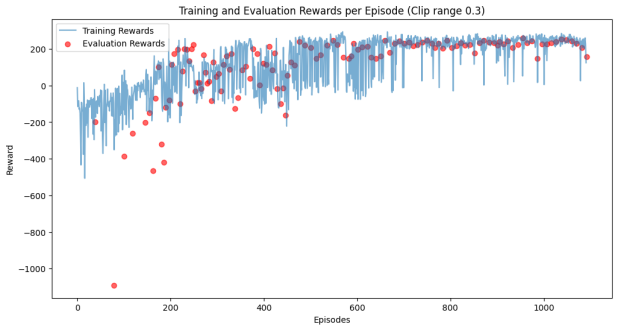
To end up stably on the ground, the spacecraft has to go through more than multiple dozens of actions and understand long-term physical relations between the change in position and velocity due to activations of the main and side engines. The default high discount rate of $\gamma = 0.999$ implies a quite farsighted agent, which still weighs rewards perceived after 100 steps with a weight of 0.9 and will hence allow the terminal reward signals of successful landing or leg contact to effect actions made far earlier. In general, a sufficiently high discount rate is necessary to allow the agent to understand long-range causations. The agent with this high learning rate has the same configuration as the agent from the earlier Figure 4b, but obtains even more stable results due to randomness we could not fix during the training process. On the other hand, an agent with $\gamma = 0.7$ will result in a cumulative discount of around 3% after only 10 steps, while the agent with $\gamma = 0.5$ even reaches 0.1%. As mentioned before, this prohibits the agent from comprehending the long-term impacts of engine activations on the dynamics of the space craft and hinders him from obtaining stable behaviour as Figures 5a and 5b depict. The discount factor depends both on the reward scale and the usual trajectory horizon and clearly profits from a high value of $\gamma = 0.999$, while $\gamma = 0.99$ might be a better choice for other environments. Here, we stick with $\gamma = 0.999$.

Finally, we compared the performance of PPO on its salient clip value ε. $\varepsilon = 0.2$ was the originally proposed default choice of the authors and proved optimal among our three candidates with regard to variance and convergence rate (Fig. 6c). While the agent still culminated in successful behaviour for $\varepsilon = 0.3$, the mean rewards during evaluation clearly depict that the policy parameters oscillate more strongly, which we hypothesize to be attributable to the approximation error of the state distribution $P_{\pi_{\theta'}}$ because of the larger allowed distributional shift (Fig. 6b). On the contrary, the variance and

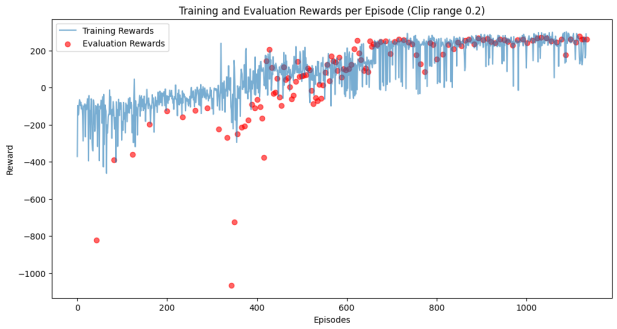
performance of the policy aggravates for a smaller $\varepsilon = 0.1$ too (Fig. 6a). We believe that the reason for this might be similar to the results observed for the low learning rate in Figure 4c. The constraint might be so strong that it hinders the agent from escaping from local minima and reaching unseen and more desirable regions in the optimization landscape.



(a) $\varepsilon = 0.1$



(b) $\varepsilon = 0.3$



(c) $\varepsilon = 0.2$

Figure 6: Training and evaluation rewards of PPO for different clip rates $\varepsilon = 0.1, 0.2, 0.3$

4.2 REINFORCE

Eventually, our experiment comprised the following seven hyperparameter combinations:

1. run: learning rate $\alpha = 0.0003$, hidden layer size $H = 256$, time threshold $T = 40$,
2. run: learning rate $\alpha = 0.0005$, hidden layer size $H = 256$, time threshold $T = 40$,
3. run: learning rate $\alpha = 0.0007$, hidden layer size $H = 256$, time threshold $T = 40$,
4. run: learning rate $\alpha = 0.0005$, hidden layer size $H = 128$, time threshold $T = 40$,
5. run: learning rate $\alpha = 0.0005$, hidden layer size $H = 64$, time threshold $T = 40$,
6. run: learning rate $\alpha = 0.0005$, hidden layer size $H = 256$, time threshold $T = 30$,
7. run: learning rate $\alpha = 0.0005$, hidden layer size $H = 256$, time threshold $T = 20$.

That is, we first studied the effects of the learning rate in the runs 1, 2 and 3, then fixed $\alpha = 0.0005$ and continued with the hidden layer size H across runs 2, 4, 5. Then, we fixed $H = 256$ and assessed the time threshold T across runs 2, 6, 7.

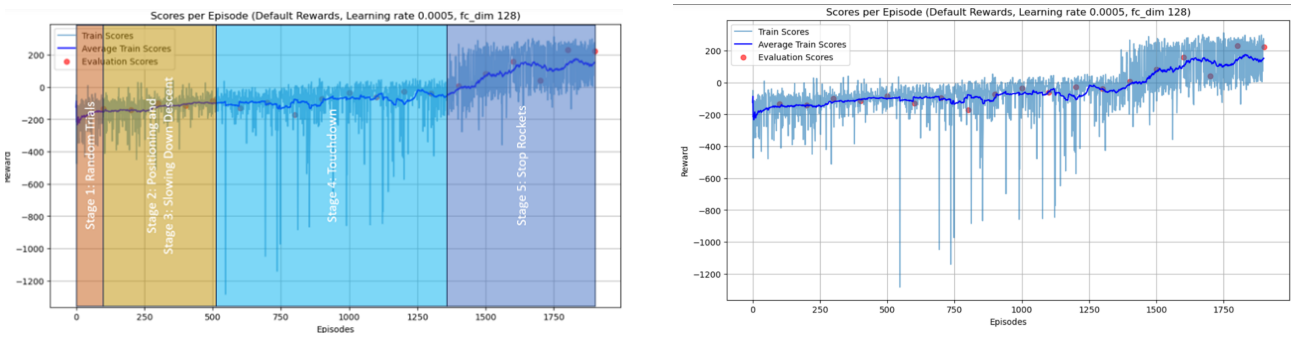
Initially, we used a standardised design of 3000 training episodes in the training process. In addition, to monitor the performance of the agent, we performed an evaluation over 10 episodes after every 100 training episodes. For each training episode, a few key metrics are recorded: training episode score, average training score over the last 50 training episodes, and a highscore that is the highest recorded score in the run until that point. During each evaluation period, the scores of each episode

and the overall average score are recorded. At the end of the run, The training scores and average evaluation scores are plotted against the training episodes. The average training scores curve gives a smooth interpretation of the training process. In addition, the videos of the agents performance over five episodes are recorded and available in `. data_policygradient`. Here, success is measured by the whether the spacecraft landed and came to a stop in between the flags.

By the third run, it is noticed that fixing 3000 iterations is not a good idea as the performance of some agents start to degrade with more episodes. An early stopping mechanism is introduced where the performance of the agent is artificially evaluated after every 100 training episodes and evaluation stint. The run will not continue if the average training or evaluation scores are observed to be beyond 200 and have peaked. Not only does this improve the time and space complexity of running the training algorithm, we are also able to evaluate the performance of the agent when it is at its best.

4.3 Training progress

Over the course of the seven runs, we have observed a few clear stages throughout the agent's learning process, which are depicted in Figure 7



(a) Overlay with stage description.

(b) Original learning curve.

Figure 7: Considering the evaluation videos along the learning process, each model seems to traverse multiple stages of competence throughout the training process.

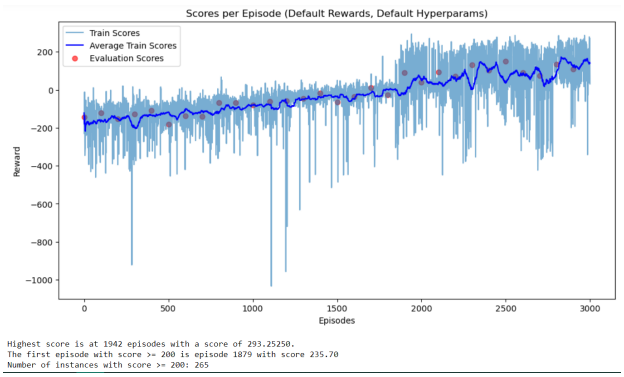
The first stage is random trials, where the agent tries random actions at each step. This stage is marked by a large variance in episode scores. Then, it enters the second stage where it focuses on its positioning to maximise rewards by moving towards the goal; and the third stage, which involves slowing down during the descent. These two stages are represented by a gradual increase in scores, and additionally for the third stage, the time taken per episode is also observed to increase slightly. The fourth stage is the touch down. Here, since the agent does not know when to stop firing its rockets, this is indicated by a huge decrease in score (from the waste of fuel), and an increase in time taken per episode (longer episodes, and to process the larger state, action and rewards arrays). Finally, in the final stage, the model will learn through random action exploration that stopping the rockets when it has landed significantly improves the score. This stage is indicated by a huge increase in individual and average scores, and a huge decrease in time taken per episode. With this framework in mind, we are able to determine the stage each run is in at each episode, and therefore compare the learning process across the different runs.

However, it is wise to remain cognisant that the boundaries of these stages may be blurred. Furthermore, the learning process can be highly stochastic even with the same hyperparameter configuration. The advance to the next stages in the framework, while possibly affected sby hyperparameter configurations, is nevertheless highly unexpected and dependent on chance. Thus, the scores and success rate of each individual run only provide a rough gauge of the performance we can expect.

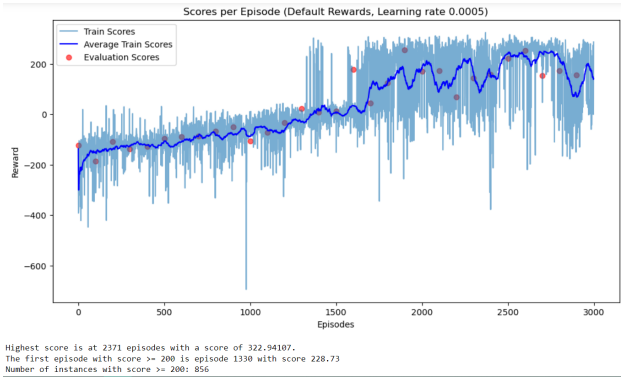
4.4 Hyperparameters

4.4.1 Learning rate

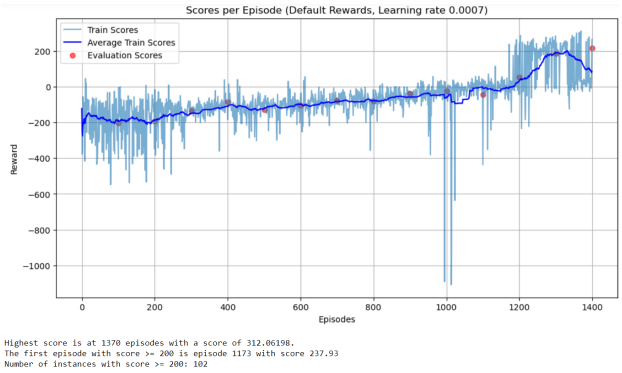
Comparing the results across runs 1, 2 and 3, we observe that for run 1 above, a learning rate of 0.0003 is too low and as a result the model failed to converge within the upper bound of 3000 episodes. Here, convergence can be defined as having a definitive peak in scores, or an average training or evaluation score of at least 200. The slow jagged increase in average training scores later in the run indicates that the craft only managed to stop in a portion of the episodes, and that the agent is transitioning between stages 4 and 5 of the learning framework.



(a) $\alpha = 0.0003$



(b) $\alpha = 0.0005$



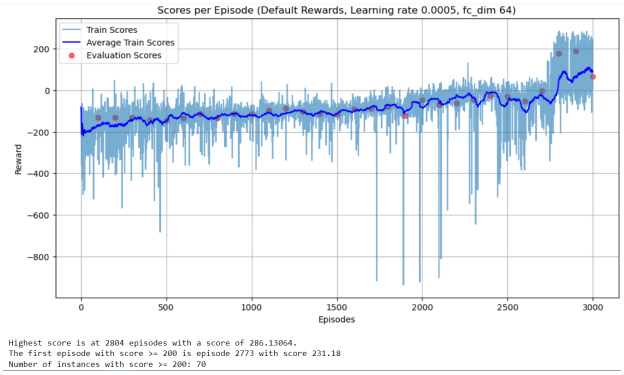
(c) $\alpha = 0.0007$

Figure 8: Training and evaluation rewards of REINFORCE for different learning rates $\alpha = 0.0003, 0.0005, 0.0007$.

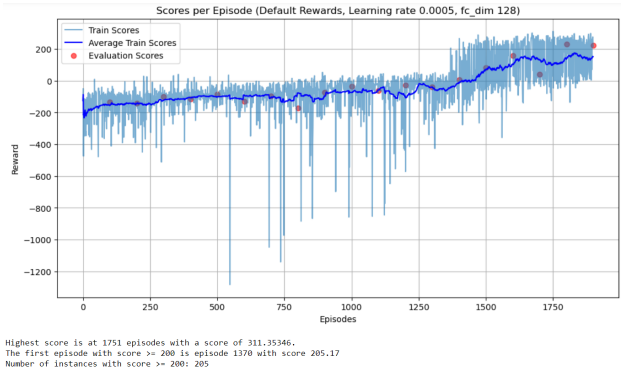
Both runs 2 and 3 above managed to converge beyond the score of 200. For run 2, this occurred at the 2000th episode, evidenced by the high average evaluation score of 254.71. Run 3 converged earlier at the 1300th to 1400th episode mark, with evaluation scores of 187.41 and 216.36 respectively. This shows that the model with a lower learning rate of 0.0005 converges later and has slightly better performance than a higher learning rate of 0.0007, which is in line with our expectations. Comparing the performance of the ship from the video clips, we can see that the ships from run 1 exhibit more risky behaviour of landing quicker, and they also have a tendency to fire the rocket after touch down. The ships from runs 2 and 3 on the other hand all landed steadily, and stopped firing their rockets after touchdown. With success defined as stopping the ship in between the flags, the success rates of runs 1, 2 and 3 over the five episodes in the videos are 2, 1 and 3 respectively. However, the success rate of run 2 is underreported due to the performance of the model having diminished after convergence. This can be due to the learning of wrong behaviour or unlearning, which reduces performance.

4.4.2 Hidden layer size

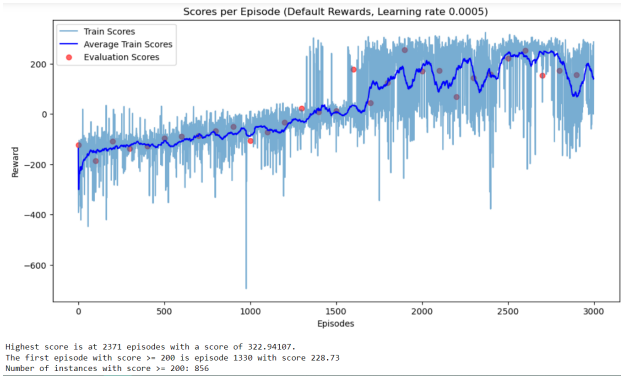
Comparing runs 2, 4 and 5 ($H = 256, 128, 64$ respectively), we can see that the model in run 4 exhibited convergence first at 1800 to 1900 episodes with average evaluation scores of 228.89 and 222.95 respectively (Fig. 9b), followed by the model in run 2 at 2000 episodes with an average evaluation score of 254.71 (Fig. 9c), while lastly run 5 only converged after 2900 episodes with an average evaluation score of 189.20 (Fig. 9a). The latter is plausible as the lower hidden layer size $H = 64$ shrinks the capacity of the neural network to capture key trends in the state, action and rewards dynamics, leading to poorer parameter estimation. However, because policy gradient approach is highly stochastic and unexpected, how early or late a model converges (and the best scores reached at convergence) could also be due to pure chance.



(a) $H = 64$



(b) $H = 128$



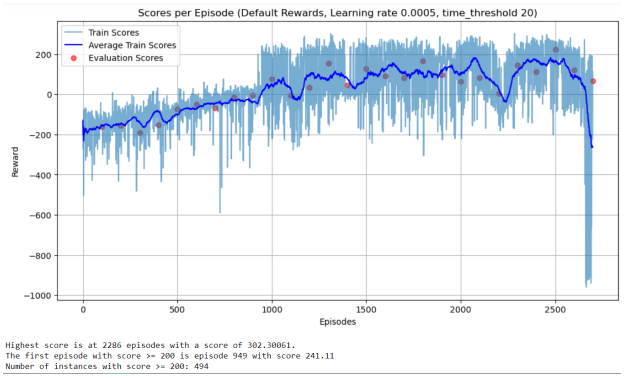
(c) $H = 256$

Figure 9: Training and evaluation rewards of REINFORCE for different hidden layer sizes $H = 64, 128, 256$ of our policy network.

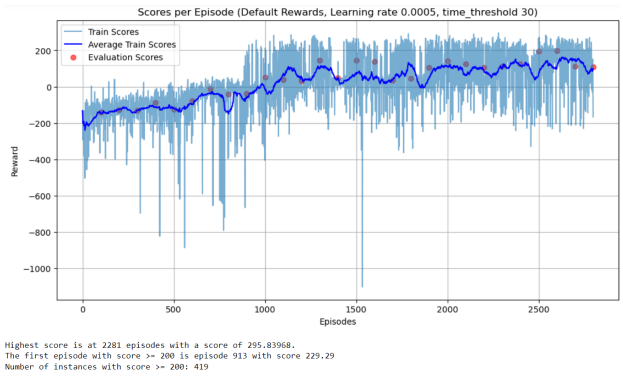
With the exception of the number of episodes required to converge, the general shape of the scores curves across the three runs are similar. As for the video playback, the ships in all three runs managed to stop firing the rocket after landing, exhibiting advanced stage 5 behaviour and demonstrated a thoroughly learned behaviour. The ships also descended steadily, with very few crashes on impact. The success rates of runs 2, 4 and 5 over five episodes are 1 (past convergence), 4 and 3.

4.4.3 Time threshold

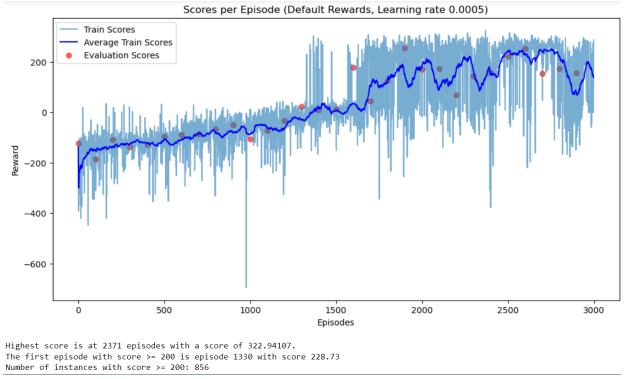
The time threshold is a hyperparameter dedicated to this LunarLander environment. It defines the maximum allowable time for each episode in seconds, and effectively discourages the ship from prolonging an episode by firing its side engines in an idle position, as was observed to occur frequently in the original evaluations. When the time threshold is met, the done variable is forcefully set to True and a reward of -50 is given as a punishment.



(a) $T = 20$



(b) $T = 30$



(c) $T = 40$

Figure 10: Training and evaluation rewards of REINFORCE for different time thresholds $T = 20, 30, 40$.

Comparing runs 2, 6 and 7 (time threshold $T = 40, 30, 20$ respectively), we observe convergence first for run 2 at 2000 episodes, with an average evaluation score of 254.71 (Fig. 10c), followed by run 6 and 7 at 2500 episodes, with average evaluation scores of 194.40 and 223.32 respectively (Figures 10b and 10a). This indicates that a larger time threshold can accelerate model convergence, which is possible considering that a longer episode provides more data for the neural network model to train on, and leaves more room for exploration. In addition, the video evidence underscores that the ships from run 2 learned the best, as most of them stopped firing their rockets after touchdown, while the ships from runs 6 and 7 all tend to keep firing their side rockets and move themselves to the left. The success rate is the highest for run 6 with three out of five ships successfully stopping within the flags – despite unnecessarily firing their side rockets –, followed by runs 2 and 7 with one out of five each.

4.5 Comparison between PPO and REINFORCE

A notable difference between the models lies within the learning rate and their variance they exhibit during training. To that end, compare the PPO learning curve in Figure 4b with the learning curve of the best REINFORCE model in 8b. While the reward of the PPO model exhibits a graceful variance that does seldom exceed 200 and even converges to a quite steady reward with extremely low the variance, even the best REINFORCE model retains a high variance of more than 200 until the end of the learning process, even though its learning rate is only have of that of the PPO model.

This naturally reflects the high variance in the Monte Carlo estimations of the policy gradient that REINFORCE uses to update its parameters. Even a lower learning rate cannot account for this strong fluctuation in the parameter update. On the other hand, the PPO model avails to the Generalized Advantage Estimation (GAE) introduced in Section 2.2.3 and thereby achieves a good trade-off between the bias in TD errors and the variance in Monte Carlo updates. We therefore

hypothesize that the addition of momentum or the application of Generalized Advantage Estimation could further stabilize this process.

5 Conclusion

The LunarLander environment is just an oversimplified representative for the challenging real-world problems we seek to tackle with RL agents. Even though PPO is currently one of the most widely used algorithms in practice, the experiments show that its performance is still sensitive to the specific hyperparameter choice and exhibits different preferences across environments. The quest for stable learning algorithms that dynamically adjust themselves to their learning progress in several environments still leads over unpaved paths that need to be explored in the upcoming years of research.

References

[1] NVIDIA Corporation & affiliates. cuBLAS API Reference Guide. https://docs.nvidia.com/cuda/cublas/index.html#cublasApi_reproducibility, 2023. Accessed: 2023-12-20.

[2] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.