ECE239AS Project: Learning to Control Mechanical Systems

Marcus Lucas	Alimzhan Sultangazin	Lucas Fraile	Matteo Marchi
205035456	005035952	505034889	205429335

1 Introduction

Practitioners of machine learning have achieved impressive results in applying reinforcement learning (RL) algorithms to the control of dynamical systems. We would like to investigate the performance of some of these algorithms that draw inspiration established classical control methods. After all, for most real-world systems, we can assume some prior knowledge of the system. Furthermore, classical methods provide the added benefit of performance guarantees, which is particularly important in safety-critical systems.

Our objective is to explore the trade-offs between different learning approaches to controlling mechanical systems. These systems typically admit a range of classical algorithms, however applying these algorithms requires substantial domain-specific knowledge. Furthermore, techniques such as optimal control may require significant on-line computational resources, precluding the solution of control tasks with a high degree of model-complexity. On the other hand, model-free reinforcement learning offers the ability to control a dynamical system without any prior knowledge at all. Some methods, such of those we will investigate, also allow for the computational cost of determining a control policy to be shifted to an off-line training phase. This, in turn, reduces the computational requirements of the controller at run-time. Unfortunately, RL approaches toward controlling physical systems are subject to several well-known disadvantages including high sample-complexity, significant computational requirements, and a lack of any formal performance guarantees.

Our objective is to obtain a qualitative idea of the performance trade-offs inherent in applying these learning-based control strategies to standard control problems, solvable via classical techniques. In particular, we have identified two RL algorithms which bridge the extremes of naive RL, with it's inherent computational complexity, and classical algorithms, which become more difficult to apply as the complexity the dynamical system increases.

In this work, we study the performance of Model-Based Deep RL (MB-DRL) [1] and Generative Adversarial Imitation Learning (GAIL) [2] in mechanical environments from OpenAI Gym [3]. MB-DRL is similar to classical model-predictive control except that the dynamics are approximated with a neural network, while GAIL tries to emulate an expert controller from demonstrations.

For the limited number of environments we tested, we conclude that GAIL outperforms MB-DR. In the case of the model-based algorithm we believe that we can improve its performance by changing the way the optimal action is chosen and by improving the policy with model-free methods.

1.1 OpenAI Gym

We decided to interface the chosen algorithms with two environments from Open AI Gym [3] - Cart-pole environment and LunarLander environment. Interestingly, both of these environments can be solved using classical control methods. These environments are used to train both of the algorithms and, then, to test their performance. We have slightly modified each of the environments before using them. Our version of the cart-pole environment is modified so that action space is continuous rather than discrete. Our version of the LunarLander environment returns an extended version of the state (22 state variables instead of 8).

Preprint. Under review.

2 Model-Based Deep Reinforcement Learning (MB-DRL)

2.1 Description

The method we describe in this section, model-based deep reinforcement learning with model-free tuning, was first described in [1]. According to the authors [1], the main benefit of this method, in comparison to model-free reinforcement learning, is its sample complexity. The main building blocks of this method are the neural network modelling the environment and the model predictive controller (MPC) maximizing the reward. While in the original paper the method was tested on MuJoCo benchmarks, we empirically demonstrate its behavior with classical control problems. Finally, for one of the environments, the model-based controller is used to initialize a model-free controller with a goal of reducing the sample complexity of a model-free algorithm.

2.1.1 Initial model training

The first step of the method is to collect the initial training data-set $\mathcal{D}$, while executing a random policy. This initial data-set will be used to train a neural network $\hat{f}_\theta(s_t, a_t)$ that approximates the dynamics f of the system:

$$s_{t+1} = s_t + f(s_t, a_t) \tag{1}$$

It is collected by initializing the system according to some (arbitrary) initial state distribution $s_0 \sim p(s_0)$ and recording the resulting trajectories $\tau = \{s_0, a_0, s_1, ..., s_{T-2}, a_{T_2}, s_{T-1}\}$. (Remark: the agent will not try to emulate the trajectories that this data-set contains). We pre-process each trajectory by splitting it into tuples (s_t, a_t, s_{t+1}), where s_t and a_t are the observed state and action at time-step t and s_{t+1} is the next observed state. For each tuple, the training set input and corresponding output are (s_t, a_t) and $s_{t+1} - s_t$, respectively. To increase model robustness, we add normally-distributed noise to the training data .

The dynamics model is trained by minimizing the following error:

$$\min_\theta \sum (s_t, a_t, s_{t+1}) \in \mathcal{D} \frac{1}{2} \|(s_{t+1} - st - \hat{f}_\theta(s_t, a_t)\|^2, \tag{2}$$

using stochastic gradient descent.

2.1.2 Model-based control

The learned model is used, together with a random-sampling shooting method MPC, to implement a policy. The MPC is supposed to determine what sequence of actions of length H provides the greatest estimated cumulative reward (using the trained model approximator), that is:

$$A_t^{(H)} = \arg\max_{A_t^{(H)}} \sum_{t'=t}^{t+H-1} r(\hat{s}_{t'}, a_{t'})$$
$$\hat{s}_t = s_t$$
$$\hat{s}_{t'+1} = \hat{s}_{t'} + \hat{f}_\theta(\hat{s}_{t'}, a_{t'}),$$

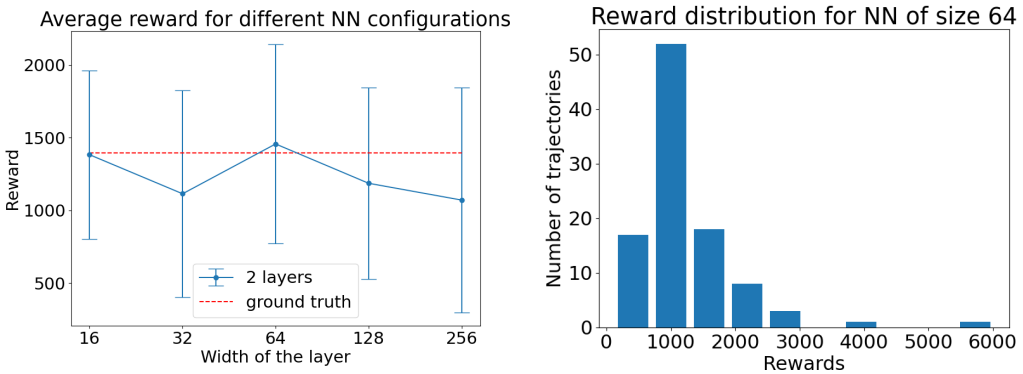
where $A_t^{(H)}$ denotes an H-length sequence of actions. Then, the policy is to execute the first action of the best sequence. In the experiments, this problem is solved by randomly sampling N sequences and picking the one leading to the best estimated cumulative reward.

We can improve the performance of the model-based learning by gathering additional training data using MPC policy and adding it to the original data-set. Similarly to [4], the on-policy data aggregation reduces the mismatch between the distribution of the data and the distribution of the model-based controller's distribution. We collect this additional data by using MPC along with the previously trained dynamics model. Afterwards, the neural network is trained further using this augmented data-set. This process is then iterated a number of times. The full process is described by Algorithm 1, taken from [1].

Finally, to refine the policy, the authors in [1] suggest training a neural network to imitate the MPC policy and to use it as a starting point to jump-start a model-free algorithm.

Algorithm 1 Model-based Reinforcement Learning

gather dataset $\mathcal{D}_{RAND}$ of trajectories using a random policy
Initialize empty dataset $\mathcal{D}_{RL}$ and $\hat{f}_\theta$
for $k \leftarrow 1$ to max_iter **do**
  train $\hat{f}_\theta$ using data from $\mathcal{D}_{RAND}$ and $\mathcal{D}_{RL}$
  for $t \leftarrow 1$ to T **do**
    get agent's current state s_t
    use $\hat{f}_\theta$ to estimate optimal action sequence $A_t^{(H)}$ (see (1))
    execute the first action a_t from optimal $A_t^{(H)}$
    add (s_t, a_t) to $\mathcal{D}_{RL}$



(a) (b)

Figure 1: Cart Pole environment. a) Average rewards when controlled by MPC using a neural network as approximate dynamics model depending on neural network layer width. The red dashed line represents average reward of MPC + actual dynamics. b) Distribution of rewards for the MPC controller with approximate dynamics model using a two-layer neural network with layer width 64.

2.2 Experiments

In this section, we report the experiments we conducted on our implementation of the algorithm described above. Like in the alternative method we implemented, we used two OpenAI-gym environments, the CartPole and the continuous version of the LunarLander, to range from a simple mechanical system to a more complicated environment that includes discontinuities in the dynamics and reward.

2.2.1 CartPole

For experiments on this environment, we modified the reward function to give a reward proportional to the deviation of the angle of the pole from the vertical. Although we attempted to penalize the deviation of the cart's horizontal position from the center, we have empirically found that better results were obtained by not penalizing it at all. This may be a consequence of the limited horizon of the used MPC policy. Unfortunately, increasing the horizon and the number of sampled action sequences rapidly increases the computational complexity of the MPC, limiting us to small numbers for both.

To approximate the environment dynamics, we have used two-layer fully connected networks of various layer size. From Figure 1a, we can see that the performance of MPC with most of the approximate dynamics models is comparable to that of MPC with the actual dynamics model, which we took as the baseline. Since the cumulative reward has a high variance over the individual experiments, some neural network configurations have better empirical average than the baseline empirical average. However, this still allows us to establish that the neural network approximation for the dynamics is sufficient for MPC policy on CartPole example.

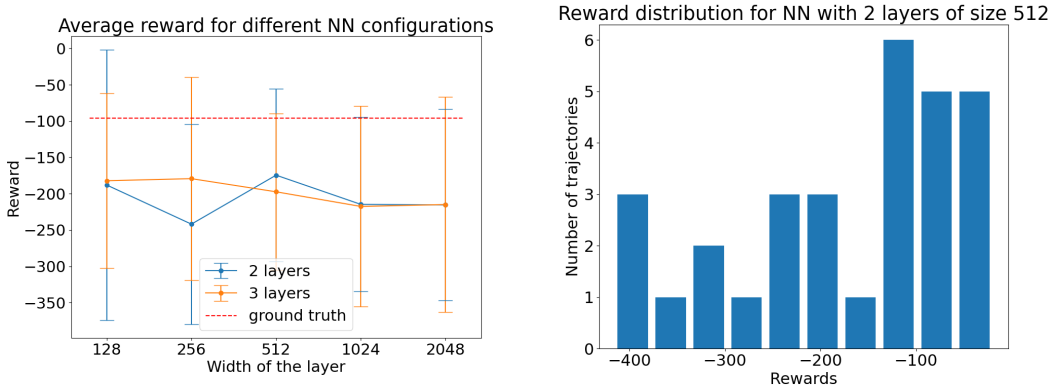


Figure 2: Lunar Lander environment. a) Average rewards when controlled by MPC using a neural network as approximate dynamics model depending on neural network layer width. The red dashed line represents average reward of MPC + actual dynamics. b) Distribution of rewards for the MPC controller with approximate dynamics model using a two-layer neural network with layer width 512.

Figure 1b shows the empirical distribution of cumulative rewards for the MPC with a two-layer neural network with layer width 64. We use this configuration as an example because it has empirically shown the best performance (see Figure 1a). Distributions for the other configurations have a similar profile.

Although the system's horizontal position will eventually drift outside the acceptable boundaries, as seen in Figure 1a this occurs only after thousands of time steps, well beyond the threshold where the task is considered solved.

2.2.2 LunarLander

We tested the model-based algorithm on a modified version of OpenAI-gym's Lunar Lander environment. The modifications were minor and mostly made for technical reasons. We expect that the obtained policies would be able to perform comparably on an unmodified environment. For the first change We increased the observation space to include 2 binary variables that are used in the environment to indicate a successful or unsuccessful episode (implying a +100 or -100 reward), primarily needed to be able to run the MPC controller and enable it to correctly predict the end of episode reward (12 more variables are included in the state, but these are only needed to run the MPC with the true model for comparisons). We also changed the successful episode termination condition so that when the spaceship's legs are touching the ground and the velocity is below a small threshold, the episode terminates successfully. The original condition requires the spaceship to come to a full stop. The change was made because the MPC algorithm searches among random sequences of actions, so even when the spaceship landed correctly, the thrusters would (with overwhelming probability) never be exactly zero for a sufficiently long stretch of time and the episode would never terminate.

For approximating LunarLander environment, we have used both two-layer and three-layer fully connected networks of various layer size. From Figure 1a, we can see that the performance of MPC with the approximate dynamics models is mostly worse to that of MPC with the actual dynamics model, which we took as the baseline. This can be explained by an increased state-space and action-space complexity of the LunarLander environment. Therefore, using the neural network approximation in this case leads to a loss in performance quality. Moreover, we can see that neither MPC with neural network approximation nor MPC using actual dynamics achieves positive reward, meaning that they were unable to successfully land the vehicle. We suspect that this is due to a myopic nature of the small-horizon limited-sequence random sampling method we employ.

Figure 1b shows the empirical distribution of cumulative rewards for the MPC with a two-layer neural network with layer width 512. We use this configuration as an example because it has empirically

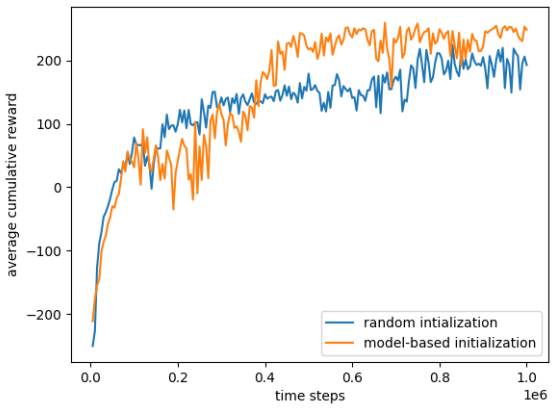


Figure 3: Average reward of a randomly initialized policy and of a model-based initialized policy in LunarLander environment over TRPO iterations.

shown the best performance (see Figure 1a). We have to note that most trials have had a reward around -100, which is a result comparable to the baseline.

Since the performance of model-based reinforcement learning in LunarLander environment was unsatisfactory, we attempted to improve the policy by model-free fine tuning, as suggested in [1]. First, we train a neural network to imitate our MPC policy and use it as a starting point in a trust region policy optimization (TRPO) algorithm [5]. We use the a third-party implementation of TRPO algorithm from [6]. In order to see if this resulted in improvement in sample complexity, we also use TRPO to train a randomly initialized policy. In our TRPO experiments we used a policy network with 2 layers with 64 neurons each, producing as output the mean and standard deviation of a normal distribution that is used to sample the actual action. Empirically, the best performing batch size (how many time steps are simulated before updating the network) was 5000. Both approaches have resulted in the agent receiving a positive cumulative reward, indicating satisfactory performance.

The results (shown in Fig.3) are certainly curious. While the model-based initialized policy appears to be worse than the randomly initialized policy at the beginning (indicating a worse sample complexity), the relationship switches around 0.4M time steps of simulation, where the model-based initialized policy rapidly increases in performance. This may indicate that the MPC policy was able to perform well on a subset of the state space, but this "prior knowledge" could not be exploited until the rest of the policy learned to reliably drive the system in this subspace.

3 Generative Adversarial Imitation Learning (GAIL)

3.1 Overview

The GAIL algorithm attempts to solve an imitation learning problem more efficiently than traditional methods such as behavioral cloning and inverse reinforcement learning (IRL). While other solutions to the IRL problem involve learning a candidate cost function, which is then used to determine an optimal policy, GAIL bypasses this intermediate step, learning the policy directly from expert demonstrations. It's derivation, however, follows similar reasoning to that of other IRL solutions and it is closely related to Generative Adversarial Networks (GANs).

3.2 Motivation

We are particularly interested in GAIL, and imitation learning more broadly, because it easily lends itself to comparison against classical control algorithms. For both the cart-pole and lunar lander systems, it is relatively easy to derive state-feedback controllers which stabilize either around a point or trajectory. For example, the dynamics of the cart-pole system (neglecting friction) can be described as follows:

$$\ddot{x} = \frac{F + ml\dot{\theta}\sin\theta) - gm\sin\theta\cos\theta}{M + m\sin^2\theta}$$
$$\ddot{\theta} = -\frac{F\cos\theta + ml\dot{\theta}^2\sin\theta\cos\theta - (M+m)g\sin\theta}{l(M + m\sin^2\theta)}$$

where M and m are the masses of the cart and pole respectively, l is one-half the length of the pole, and F is the external force applied laterally to the cart.

The most straight-forward way to control a nonlinear system is to linearize its dynamics about one of its equilibrium points and design a controller for the approximate model. In this case, $(x, \dot{x}, \theta, \dot{\theta}) = (0, 0, 0, 0)$, is an equilibrium point of the cart-pole system representing an upright pendulum. Its corresponding linearization is:

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dot{\theta} \\ \ddot{x} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{gm}{M} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{g(M+m)}{Mml^2} & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{M} \\ 0 \\ \frac{1}{Ml} \end{bmatrix} F$$
$$\dot{\vec{x}} = A\vec{x} + BF$$

For any stabilizable linear system of the form $\dot{\vec{x}} = A\vec{x} + Bu$, we can design an infinite-time linear quadratic regulator (LQR) by solving the Algebraic Ricatti Equation:

$$A^T P + PA - PBR^{-1}B^T P + Q = 0$$

for P. The matrix P defines a linear state feedback control law $K = R^{-1}B^T P$ which minimizes the following cost function $V = \sum_{t=1}^{T} x_t^T Q x_t + u_t^T R u_t$, where Q weights the quadratic cost on the states and R the weight on the input. By changing the ratio between the elements of Q and R one can prioritize energy expenditure over settling time or vice versa.

We can stabilize the linearized cart-pole system by designing an LQR controller $F = K\vec{x}$ for some positive semi-definite weight matrices Q and R. A well known result from control theory tells us that as long and the state of our system does not deviate too much from its equilibrium point, this linear controller will also stabilize the true system. This makes an LQR controller an excellent expert for trajectory generation as the domain of the cart-pole's state in the Open AI Gym environment lies inside the domain in which the linearization is valid. The expert trajectories we generate for later training include, for every step, a tuple composed of the state of the system together with the resulted input F computed by the controller.

3.3 GAIL Description

Generative Adversarial Imitation Learning (GAIL) [2], is a model-free algorithm which is able to learn a control policy from expert examples. The algorithm is in fact closely related to generative adversarial networks (GANs), and leverages their power in order to successfully learn controllers for otherwise prohibitively complex environments. The intuition behind GAIL, which makes the connection with GANs clear, is at follows: a neural network we will call Actor is trained to forge controllers based on the expert trajectories, receiving the state of the system and returning an input F; a second neural network we will call Discriminator is trained to differentiate between forged trajectories created by the Actor and expert trajectories created through classical control methods; by pitting the Actor and Discriminator against each other, both improve in their own tasks. If the Actor improves enough that there is no significant differences between the inputs it generates and those generated by the LQR controller, it is safe to expect the Actor to succeed in stabilizing the system.

More precisely, the Actor attempts to learn a policy π whose occupancy measure ρ_π approximates the occupancy measure ρ_{π_E} of the expert's policy π_E, and the Discriminator's job is to distinguish between the two policies' occupancy measures. Here, occupancy measure can be thought of as the distribution of state-action pair encountered while navigating the environment with a certain policy.

In practice, both the parameterized policy π_θ with weights θ and the discriminative classifier D_w with weights w are implemented by neural networks. The goal is to find the saddle point of the following expression:

$$\mathbb{E}_\pi[\log(D(s,a))] + \mathbb{E}_{\pi_E}[\log(1 - D(s,a))] - \lambda H(\pi), \tag{3}$$

where $s \in \mathcal{S}$ and $a \in \mathcal{A}$ are the state and action, respectively, and $H(\pi)$ is the γ-discounted causal entropy of the policy π. To achieve this, the algorithm alternates between an Adam gradient step on w to increase (3) with respect to the Discriminator D and proximal policy optimization (PPO) [7] on θ to decrease (3) with respect to π.

3.4 Experiments

As in the previous section, we implemented, trained and tested this learning algorithm on two Open AI-gym environments, continuous input versions of the cart-pole and of the Lunar Lander. For the cart-pole environment we obtained expert trajectories from an LQR controller as detailed in section 3.2. For the second environment, we trained a policy using an implementation of the PPO algorithm taken from the RL Baselines3 Zoo [8]. This was done to save time, however it is entirely possible to derive a classical control policy for the Lunar Lander model as well.

3.4.1 Cart-pole

For experiments on this environment, we modified the action space to be the segment of the reals between -10 and 10. We chose $Q = 10I$ where I is the rank 4 identity matrix and $R = 2$, to reflect we care more about stabilization than control effort. This results in the expert controller $u = K\vec{X}$ where $K = [2.2361 \quad 4.6286 \quad 46.7714 \quad 15.3921]$. As this expert is deterministic, we would query it at every state encountered during the training.

The neural networks for both the Actor and Discriminator were given similar architectures, both fully connected feed-forward networks with a single $200x200$ hidden layer, the input and hidden layers with ReLu and the output layers with hyperbolic tangent activation. The Actor's output layer is scaled by the maximum value actions can take.

We trained the GAIL algorithm for 400 epochs of 20 episodes each, we considered the training to succeed if all twenty episodes in an epoch lasted 1500 steps 30 times. An episode would finish whenever one of the state's left the domain of the environment or when 1500 steps were reached without doing so. This threshold was selected heuristically as experiments seemed to indicate that to succeed the actor was forced to learn how to completely stabilize the system.

An example of the training curve is shown below:

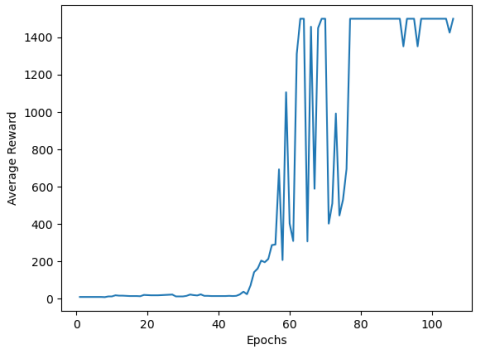


Figure 4: Average rewards given by cart-pole environment for 20 episodes per epoch.

We can see in the plots below how when tested this Actor succeeds in stabilizing the system for an infinite number of steps as it reaches and equilibrium for both states and input.

3.4.2 Lunar Lander

We used a very similar setup for the Lunar Lander, where both the Actor and Discriminator were represented by feed-forward networks consisting of two 200-node hidden layers interleaved with

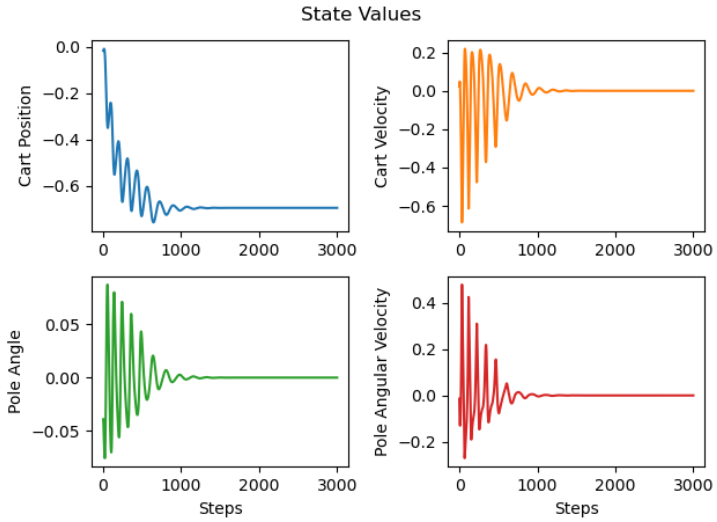


Figure 5: State evolution for random initial conditions when testing the trained Actor from Gail.

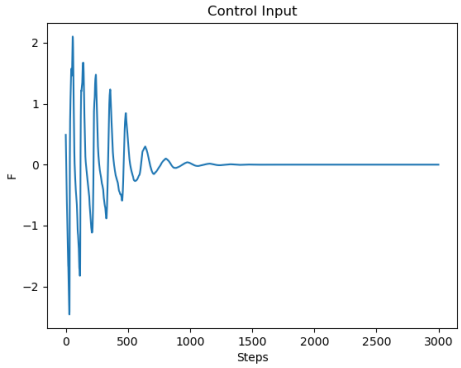


Figure 6: Action selected by the trained Actor corresponding to the states in Figure 5

ReLU non-linearities. The Actor network maps an 8-dimensional state space to a 4-dimensional action space, while the discriminator maps the action space to a discrete set of cardinality 2. Expert traces were taken from another network, trained beforehand using the Proximal Policy Optimization (PPO) method. The choice to use a learned controller, rather than a classically designed optimal controller, was made in order to save time.

Here are examples of the training curves obtained for the LunarLander:

The performance of the GAIL-derived policy was comparable to that of the expert, however it trained in a fraction of the time. It stands to reason that were we to implement a classical feedback controller for the lander, we would see similar results.

4 Discussion

4.1 Model-based Reinforcement Learning

While model-based reinforcement learning has shown satisfactory results in cart-pole example, its performance in LunarLander was inadequate. We suppose that this is due to a larger state space and action space in the latter example. An argument can be made against this conjecture by considering previous work employing this method. In [1], the authors have shown that model-based reinforcement learning works for MuJoCo examples, which also have relatively large state space and action space.

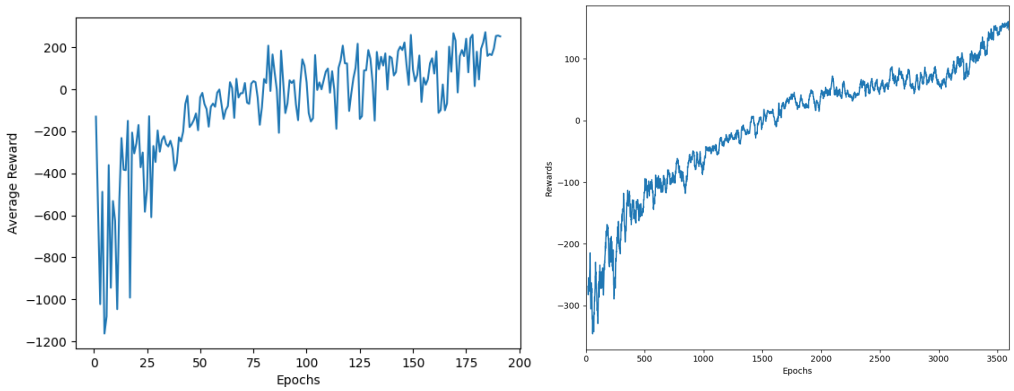


Figure 7: (Left) Average rewards given by GAIL during training on LunarLander with 20 episodes per epoch. (Right) Rewards during PPO training of initial LunarLander expert policy.

The experiments have shown that performance of a model predictive controller does not change significantly depending on whether a neural network or actual dynamics are used. This indicates that the main problem of the policy lies in the choice of random sampling method to execute MPC. In large action space, it is well possible that all of the randomly selected action sequences are highly sub-optimal, which leads to a choice of sub-optimal action and, thus, to bad performance.

4.2 Generative Adversarial Imitation Learning

We were able to successfully train a GAIL network for both the cart-pole and LunarLander systems. Our biggest problem was that we were not able to train control policies for more complicated systems such as the MuJoCo examples. Obviously using knowledge of a system, such as its model or state-representation, is useful for the purposes of control. We confirmed that, at the very least, such knowledge can significantly speed up the training time for a learned controller.

The next step for our research must be to analytically characterize the relationship between our learned controllers and the dynamical systems they act on. This is because a main focus of control theory is safety, and without a precise characterization of how a learned controller will affect the system it acts on, it is unreasonable to apply any of our learning techniques to real-world systems.

References

[1] Anusha Nagabandi, Gregory Kahn, Ronald S. Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. *CoRR*, abs/1708.02596, 2017.

[2] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 4565–4573. Curran Associates, Inc., 2016.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[4] Stephane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. *arXiv e-prints*, page arXiv:1011.0686, November 2010.

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

[6] Ilya Kostrikov. Pytorch implementation of trust region policy optimization. `https://github.com/ikostrikov/pytorch-trpo`, 2018.

[7] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[8] Antonin Raffin. Rl baselines3 zoo. https://github.com/DLR-RM/rl-baselines3-zoo, 2020.