Deep Reinforcement Learning for In-Flight Calibration of a Lunar Lander

Robathan Harries rharries@stanford.edu

Ben Alexander balex@stanford.edu

Gordon Downs gwdowns@stanford.edu

Abstract

The OpenAI Gym's lunar lander problem is a classic RL setting for continuous actions in a 2D world, where the goal is to use an upward thruster (controlled by action a_0) and lateral thrusters (controlled by action a_1) to gently land on a landing pad. We consider a more difficult version of this problem, where at the beginning of each episode, we randomly choose an offset $\delta \in \mathbb{R}^2$ to add to the actions outputted by the agent for the duration of the episode. This simulates the scenario where a violent rocket launch knocks the lander's thrusters out of proper calibration, requiring it to calibrate itself mid-flight as it lands.

We train several types of deep models to solve this "lunar lander with offsets" problem. First, we train Deep Deterministic Policy Gradient (DDPG) models that learn a single policy to apply to every environment, and that don't "know" that there is any difference in environment dynamics across episodes. Second, we train a small neural network to explicitly estimate the current episode's offset given the lander's trajectory so far, and use that neural network to correct the actions outputted by a basic DDPG model. This second type of model "knows" that each episode is different, and that the difference lies in the offsets. Third, we train more complex "meta-RL" models, which are motivated by the idea that an agent must learn to adapt to new environment dynamics in each episode. Meta-RL models "know" that each episode is different, but don't know anything about what makes them different.

We compare all three types of models and find that models that explicitly estimate the episode's offset perform the best. While DDPG models are either overly aggressive or overly cautious and meta-RL methods are too unstable, explicit offset estimation models have good performance no matter what the offset is.

1 Problem Statement

We consider a more difficult version of the OpenAI Gym's classic lunar lander problem [1, 2]. In the original lunar lander problem, the rocket is initially shot out with a random force applied to it, and the goal is to train an algorithm that knows how to appropriately fire the left/upward/right thrusters in order to course-correct and safely land the rocket.

It is important to note that by default, OpenAI Gym's lunar lander environment uses discrete actions (fire upward or right or left thruster at full throttle, or do nothing). However, we focus on the continuous action environment, where actions are $a \in \mathbb{R}^2$. In the continuous action environment, the agent controls not only which thruster(s) to fire, but also with how much power to fire them. The first element, $a_0 \in [0, 1]$ controls the upward thruster, and the second element, $a_1 \in [-1, +1]$ controls the left/right thrusters. The details of what action values do are found in Table 1. The agent receives rewards of around 100-140 points for landing in the landing pad. It receives negative reward for moving away from the landing pad, and an additional -100 points if it crashes. If it comes to rest, it receives +100 points. Each leg that comes into contact with the ground receives +10 points. Firing the upward thruster costs -0.3 points per time step, and firing the side thruster costs -0.03 points per time step. A total score of 200 points is considered to be "solved."

We refer to the original problem setting described above as the "vanilla lunar lander problem," as opposed to the "lunar lander problem with offsets" that we describe below.

For the more difficult version that we attempt in this project, we add an additional layer of difficulty to the continuous lunar lander problem by applying an offset $\delta \in \mathbb{R}^2$ to the actions. At the start of every episode, we randomly choose an

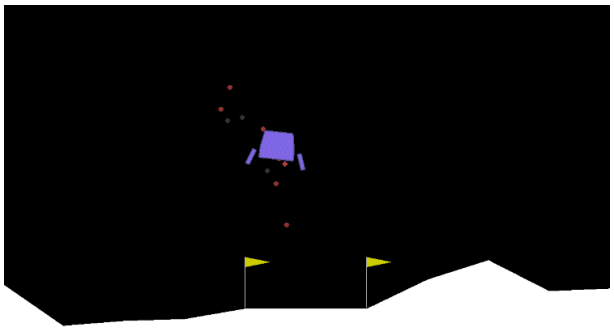

Figure 1: Example visualization of the lunar lander problem. The particles around the lander indicate that the upward and left thrusters are firing.

Action	Condition	Effect
$a_0 \in [0, 1]$	$a_0 = 0$	Upward thruster: off
	$a_0 > 0$	Upward thruster: on with power proportional to a_0
$a_1 \in [-1, +1]$	$a_1 < 0$	Left thruster: on with power proportional to $\lvert a_1 \rvert$
	$a_1 = 0$	Left/right thrusters off
	$a_1 > 0$	Right thruster: on with power proportional to a_1

Table 1: The effect of actions in the continuous lunar lander environment.

offset for each action, and for the duration of the episode, we add that offset to the actions that the agent outputs. As a result, to maximize the reward it receives, the agent will need to quickly identify and account for the offset applied to the thruster strengths for that episode. We sample offsets from the interval between $\pm$ half of each action space, i.e., $\delta_0 \in [-0.5, 0.5]$ and $\delta_1 \in [-1, 1]$. A notable implementation detail is that if $a_0 + \delta < 0$, it has the same effect as when $a_0 = 0$, and if $a_0 + \delta_0 > 1$, it has the same effect as when $a_0 = 1$. Similar conditions hold for a_1. In other words, actions plus offsets get saturated/clipped at the boundaries of the original action spaces, so offsets can't make the lander perform an action that is impossible in the vanilla lunar lander setting.

The vanilla lunar lander problem without offsets can be solved optimally by a policy that only takes in the current state as input, while an optimal policy for the lunar lander problem with offsets requires reasoning across the current episode's trajectory to understand the current episode's dynamics. This added requirement to reason across time is why the problem with offsets is more difficult than the problem without offsets.

We are motivated by the real-life scenario of a violent rocket launch knocking a lander instrument out of proper calibration, resulting in a lander that must calibrate itself mid-flight as it lands. For example, one episode could simulate the scenario where the upward thruster has been weakened (represented in our simulator as $\delta_0 < 0$), and the rocket will have to adjust for this weakness on the fly. Another episode could simulate the scenario where the lander has a left/right alignment issue and so it drifts to the left (represented in our simulator as $\delta_1 < 0$). Every episode has some combination of offsets for the two actions.

2 Related Work

2.1 Deep Reinforcement Learning Methods

Researchers have developed a variety of methods for optimizing performance on reinforcement learning tasks. One popular approach is the Deep Q Network (DQN) [3] algorithm, which was originally developed for playing Atari games. However, although the DQN performs well, it can only handle *discrete* action spaces. For the continuous Lunar Lander problem that we attempt here, the action space is *continuous*. Thus, we instead turn to deep policy gradient methods, which directly optimize the parameters of a policy in order to maximize performance, and can often handle continuous action spaces quite well. A popular version is Deep Deterministic Policy Gradient (DDPG) [4], which is a model-free actor-critic algorithm that is known for its good performance.

2.2 Explicit Offset Estimation Methods

Another class of methods considers the case where we *explicitly estimate* the offset in the current environment, and then directly correct for it. For these methods, we must already have a good policy for the vanilla lunar lander problem, i.e., the lunar lander problem without any offsets. Then, we train a second model to explicitly estimate the current episode's offset δ based on the trajectory of `(state, action, next state)` tuples that we have observed in the current episode so far. Then, we can use this estimated offset to correct the policy outputted by the vanilla lunar lander policy, by simply subtracting the estimated offset δ from the predicted action.

This approach is in many ways similar to a Model Identification Adaptive Control (MIAC) method, a class of methods originally proposed for controlling quadcopters in a similar problem setting to the lunar lander problem with offsets [5].

2.3 Meta-Reinforcement Learning

Reinforcement learning methods learn how to maximize an agent's performance in a specific environment. However, when presented with a slightly different environment, they can often perform quite poorly. In contrast, meta-reinforcement learning methods are designed to be more robust and quickly adapt to small changes in the environment. Specifically, instead of training on a single environment, we train on a *distribution* of similar environments, and use special algorithms that are designed to quickly adapt to the particular environment the agent is currently in. One common method, known as RL^2 [6], makes use of recurrent neural networks (RNNs). By recurrently passing the state/action/reward into the RNN at each time step, the model builds up a better understanding of the current environment, and is able to make better decisions moving forward. Another method is called Probabilistic Embeddings for Actor-Critic RL (PEARL) [7]. PEARL trains a probabilistic encoder to convert a trajectory into a task representation vector, as well as a task-conditioned actor policy.

3 Approach

We consider a variety of approaches that give the agent different amounts of information about the calibration problem.

3.1 Basic Reinforcement Learning: DDPG

We begin by applying standard RL methods to the lunar lander problem. These methods treat the problem as if it were just a regular RL problem with a single environment. Since we are attempting the lunar lander version with continuous actions, we use the Deep Deterministic Policy Gradient (DDPG) algorithm [4]. This is a policy gradient method that uses an actor-critic formulation. The actor is the policy network $\pi(s)$, and the critic is the Q function $Q(s, a)$.

DDPG is specifically tailored towards continuous action spaces. Normally, in Q learning, to select the optimal action, we choose $a^*(s) = \text{argmax}_a \ Q^*(s, a)$. However, with continuous actions, taking this argmax becomes intractable. So, in DDPG, we instead have the policy network $\pi(s)$ which is trained to maximize the Q function; this maximization is possible because $Q(s, a)$ is differentiable with respect to a. Thus, we can use the approximation $\max_a Q(s, a) \approx Q(s, \pi(s))$.

3.1.1 DDPG trained *without* offsets

As a first step, we use DDPG to solve the vanilla lunar lander problem. At test time, however, we evaluate this trained model on our more complex Lunar Lander problem *with* offsets. We expect this model to perform quite poorly when evaluated in the more complex environment, since it was trained in a much simpler environment where the rocket was assumed to be perfectly calibrated. Therefore, this serves as a good baseline that we hope our more advanced methods will improve upon.

3.1.2 DDPG trained *with* offsets

Next, we retrain the exact same DDPG model, but now *with* offsets during training. Note that we do not do anything to explicitly correct for these offsets; we just enable the random offsets during training. You can essentially think of this as a "data augmentation" technique, similar to how an image classification algorithm might use "random flipping" to make the model more generalizable. The idea is that by simply enabling offsets during training, the model will learn by itself to become more robust to the random offsets.

This model serves as a second baseline. Our initial expectation was that it would outperform the first baseline described in the previous section (since it at least is aware of the offsets during training), but still perform worse than the models

described in the next few sections (which will have explicit architectural improvements designed to handle these offsets better).

3.2 Explicit Offset Estimation Methods

As motivated in Section 2.2 above, we use a method for explicitly estimating the offset of the current episode so that we can correct for it. For this method, we use two separate stages, with a different model for each stage. The first stage estimates the current environment's offset, and the second stage uses a policy to obtain a desired action. The models in the two stages are trained separately.

In the first stage, we use a four-layer MLP to predict the offset $\delta \in \mathbb{R}^2$ given the trajectory that we have observed in the current episode so far. We feed each `(state, action, next state)` tuple from the current trajectory into the MLP separately, and then take the mean (across tuples) of the MLP's outputs to obtain an estimate of the offset for the current episode. The MLP was trained for 150 epochs with the Adam optimizer, at a learning rate of `1e-3`. The model has hidden sizes of 256, 128, and 64, in that order, and a ReLU activation function for every hidden layer.

In the second stage, we use the DDPG agent trained without offsets (see Section 3.1.1) to obtain the action we would want to take if there were no offset in the current episode. Of course, since there *is* an offset in the current episode, we need to combine this with the result of the first stage in order to have a useful policy.

We combine the results of the two stages by creating a policy that outputs the action $a_{\text{no\_offset\_agent}} - \delta_{\text{estimated}}$, i.e., we subtract our estimate of the offset in an attempt to cancel out the true offset.

3.3 Meta-RL

Meta-reinforcement learning methods (as introduced in Section 2.3) are, in some ways, the most generalizable approach for solving this type of problem. For the explicit offset estimation approach described above, we used our knowledge about the environment to specify that the possible offset will always be an additive shift. Meta-RL methods, in contrast, could theoretically adapt to *any* changes in the task environment, even when the exact form of these changes is unknown. So, even though the explicit offset estimation approach is a great solution for our modified lunar lander problem, we were interested in exploring meta-RL methods just to see how far we can push our performance, using as little information about the environment as possible. We implemented two meta-RL methods: RL^2 and PEARL.

Unfortunately, meta-RL methods are also notoriously difficult to train, as we will show later. So, we viewed this as more of a stretch goal that we wanted to explore out of curiosity.

3.3.1 RL^2

The RL^2 algorithm [6] is a well-known meta-RL algorithm that makes use of recurrent neural networks (RNNs). In most regular RL problems, we typically just pass in the state/action/reward tuple from the current time step and choose the next action based on that, since we usually assume the Markov property within an MDP. With RL^2, however, we are now using an RNN, so as we pass in the state/action/reward tuple at each time step, the RNN's internal state gets updated to adapt to the current environment based on the environment dynamics it has observed so far. Concretely, for our Lunar Lander problem, the RNN should learn to implicitly recognize the random offset in the current environment and adapt its internal state accordingly, such that it makes better decisions moving forward.

For our implementation, we used a Gated Recurrent Unit (GRU) [8] with hidden size 256, and a small multi-layer perceptron to preprocess the input state/action/reward tuple before passing it into the GRU. We use the Adam optimizer with a learning rate of `2e-4`. We use Proximal Policy Optimization (PPO) [9] as the optimization algorithm.

3.3.2 PEARL

PEARL [7] is a context-based meta-RL algorithm which uses a probabilistic task encoder to convert on-policy task experience into a latent task vector, and a latent-conditioned actor-critic network trained on off-policy data. The task encoder uses the first 8 steps of an episode to produce a Gaussian distribution of possible latent task vectors. These latent task vectors are trained to represent different action offsets, and the width of the distribution reflects the uncertainty of the task encoder's prediction. Samples from this distribution are used as additional inputs to actor and critic networks, and the task encoder is trained end-to-end alongside the critic network, which encourages task representations to usefully improve the critic network's accuracy.

Specifically, the algorithm maintains separate on-policy and off-policy replay buffers for each task. In each training step, for some batch of tasks, the task encoder samples a latent task vector for each batch task using on-policy data, and

the latent-conditioned actor-critic network performs an update using off-policy data for each batch task (conditioned on the corresponding sampled latent vectors). By using off-policy data, PEARL is far more sample-efficient than recurrent and gradient-based meta-RL algorithms [7].

3.4 Summary of All Methods

In Table 2 below, we summarize all five methods we use in this project.

Method Name	**Summary**
DDPG trained without offsets	Standard continuous-action RL method, trained in the vanilla lunar lander setting (no offsets)
DDPG trained with offsets	Standard continuous-action RL method, trained in the lunar lander setting with offsets
Explicit offset estimation	Method that corrects actions from "DDPG trained without offsets" agent with an MLP that estimates the current episode's offset
RL^2	Meta-RL method that uses an RNN to quickly adapt to the current environment based on the environment dynamics it observes
PEARL	Meta-RL method that encodes tasks/environments, then uses a task/environment-conditioned actor policy

Table 2: Summary of the methods used in this project.

4 Results and Analysis

Figure 2 shows the mean episode rewards for each of our methods, looking at one action dimension at a time.

We see that the DDPG model trained *without offsets* solves the lander problem when the action offsets are small; near 0 on the x-axes, the average episode reward is around 250, which is well above the 200-reward "solved" threshold. However, it fails to solve the lunar lander problem when the action offsets are significant, i.e. when $|\delta_0| > 0.25$ or $|\delta_1| > 0.5$. This makes sense, because small offsets don't change the dynamics that much, but the larger offsets cause a significant difference in dynamics that the model is unprepared for. For example, when the lateral offset δ_1 is very large, with absolute value close to 1, the rewards are below -200, which indicates that the lander is immediately veering off to one side and crashing.

Meanwhile, the DDPG model trained *with offsets* fails to solve the lunar lander problem at all, with or without offsets. It is worst when the offsets are large, but is still below the 200-reward "solved" threshold everywhere. Figure 4 in the Appendix (which was cut from the main paper for length) illuminates why: this model is so cautious that unless it immediately crashes due to extreme offsets, the agent essentially hovers until the maximum episode length is reached.

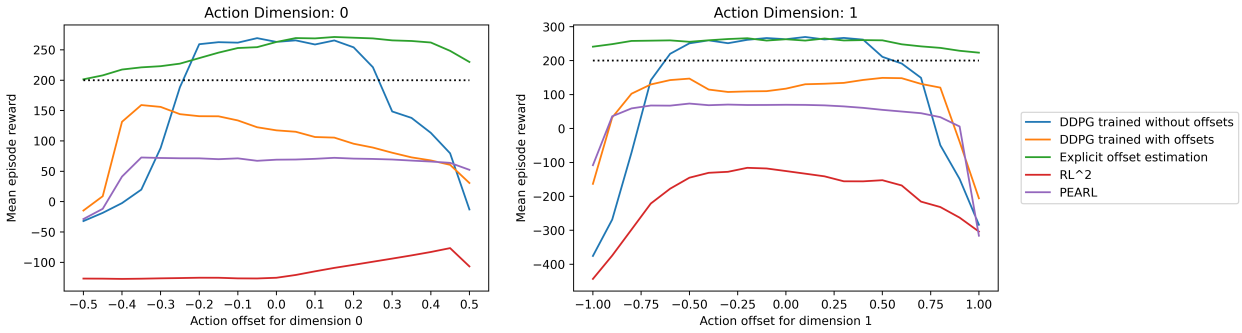

Figure 2: Mean episode rewards for each method, while varying one offset dimension at a time. Each point is averaged across 100 episodes. For each plot, we hold the other offset dimension constant at 0 (i.e., for the left plot, we set $\delta_1 = 0$; for the right plot, we set $\delta_0 = 0$). The dotted lines are at 200 reward, which is considered "solved." We see that the best model, the explicit offset estimation algorithm (green), successfully solves the problem at all offset values.

This agent learned to be so cautious because it can't adapt to the episode's action offset, and instead can only at best avoid crashing by staying level and high in the air.

On the other hand, the explicit offset estimation model performs extremely well; for all offsets, it is above the 200-reward "solved" threshold. It makes sense that this model performs better than the previous two, because it is the only one of these three models that can adapt to each new episode's action offset. The offset estimation MLP results can be seen in Figure 5 in the Appendix. This algorithm successfully solved the problem.

Finally, we have the meta-RL results from RL^2 and PEARL. As you can see, neither of them performed well. This is because we found both models to be unstable during training, and thus we were unable to reach a satisfactory solution. In the Appendix, Figure 6 shows the training curves for each model; you can observe that the performance spikes up and down quite a bit, and never converges to a good solution. However, we still believe these methods could eventually work, given enough training time and hyperparameter tuning; they are theoretically solid but just a bit unstable to train.

Figure 3 visualizes the mean episode rewards of the deep RL and explicit offset estimation methods in a different way, by changing both δ_0 and δ_1 at the same time, instead of only varying one at a time. Unsurprisingly, we see the same pattern as when we change just one at a time: the DDPG agent trained without offsets only performs well near $\delta = [0, 0]$; the DDPG agent trained with offsets performs at a uniformly poor level, but a bit worse at extreme offsets; and the explicit offset estimation agent performs well for all offsets, but best when δ_0 is not too negative.

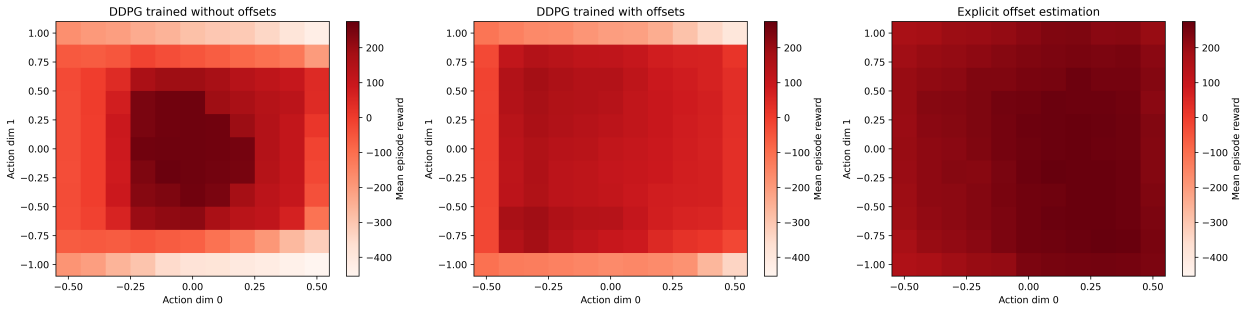

Figure 3: Heat maps showing average reward (darker red = better) for three of our methods, evaluated across many combinations of offset values (δ_1 on y-axis, δ_0 on x-axis). DDPG trained without offsets (left) only performs well with offsets around [0,0], since that is the setting it was trained on. DDPG trained with offsets (center) performs relatively poorly everywhere. The explicit offset estimation (right) performs the best, by far.

5 Conclusion

In this project, we solved the lunar lander with offsets problem. This is a modification of the basic lunar lander problem, where we randomly sample a (2D) action offset at the beginning of each new episode, and then add this offset to the lander's (continuous) actions throughout that episode. A basic deep RL agent trained in the vanilla (no offsets) setting solved the problem only for small offsets; a basic deep RL agent trained in the setting with offsets was too cautious and didn't solve the problem at all; and meta-RL methods which were designed to adapt to new episode/environment dynamics were too unstable during training. However, one model successfully solved the lunar lander problem across a wide range of offsets: our model that explicitly estimates the offset of the current episode and uses the estimate to correct the actions of a basic deep RL agent trained in the vanilla (no offsets) setting.

Future works could consider attempting an end-to-end version of the explicit offset estimation model where both stages—the offset estimation MLP and the agent—are trained together rather than separately. In addition, a broader hyperparameter sweep might be able to find hyperparameters that allow RL^2 or PEARL to successfully solve the lunar lander problem with offsets. Finally, a setting with linear/affine (not just additive) offsets, or possibly even offsets that change over the course of an episode, could be interesting to explore.

6 Team Contributions

Rob set up the environment and trained the standard deep RL methods and PEARL meta-RL method.

Ben trained the RL^2 meta-RL method.

Gordon trained the offset estimation MLP and wrote the evaluation code.

All authors contributed to writing the paper.

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[2] Lunar lander. https://www.gymlibrary.dev/environments/box2d/lunar_lander/.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[4] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[5] Matthias Schreier. Modeling and adaptive control of a quadrotor. In *2012 IEEE International Conference on Mechatronics and Automation*, pages 383–390, 2012.

[6] Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl^2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

[7] Kate Rakelly, Aurick Zhou, Chelsea Finn, Sergey Levine, and Deirdre Quillen. Efficient off-policy meta-reinforcement learning via probabilistic context variables. In *International conference on machine learning*, pages 5331–5340. PMLR, 2019.

[8] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

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

7 Appendix

7.1 Appendix A: Average Episode Lengths

In Figure 4 below, we plot the average episode length for each method.

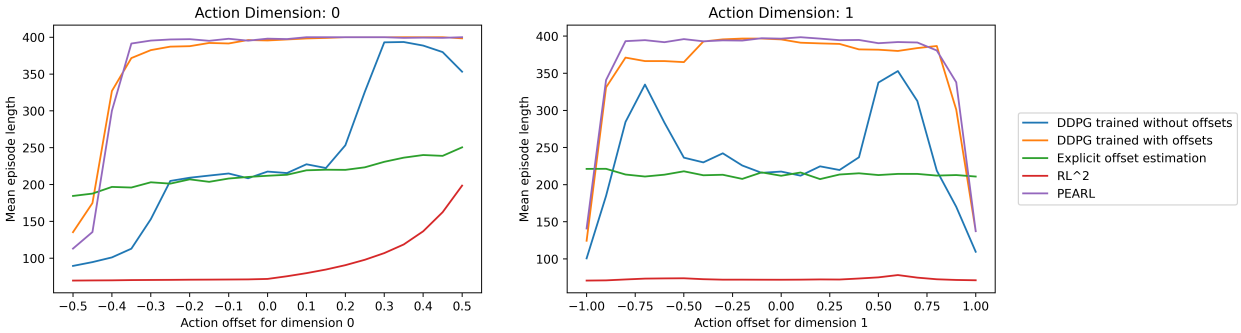

Figure 4: Average episode length (number of steps per episode) for each model.

We can observe some interesting trends. The DDPG trained with offsets model has very long episode lengths on average (around 400 steps per episode). By watching some videos of this model attempting the lunar lander task, we found that

this is because this model is extremely cautious and basically learns to hover in the air most of the time. This makes sense, because we are applying offsets during training but not explicitly doing anything to help the model do better in this environment; therefore, it learns to be very "safe" and hover in place, to avoid crashing and receiving the associated -100 point penalty. PEARL exhibits the same behavior; it seems to have learned to just hover most of the time.

On the flip side, the RL^2 model has very short episode lengths on average. This is because it basically learns to not do much at all, and mostly just falls towards the ground, which happens quite quickly. This is likely just because the model didn't train very well.

The other methods lie somewhere in the middle in terms of average episode length.

7.2 Appendix B: Offset Estimation MLP Performance

As described in Section 3.2, for the explicit offset estimation model (our best model), we trained an MLP to identify the offset in the current environment based on the `(state, action, next state)` tuple from the previous time step. We then used this estimate to manually correct the outputs of our vanilla DDPG model. The evaluation results of the MLP itself are shown in the heat maps below, where we plot the actual vs. predicted offset. You can see that the MLP is pretty good overall (although not perfect); the predictions mostly follow the diagonal $y = x$ line, as they should. Of course, during inference we average these estimates over multiple time steps from the current episode, which provides a much more accurate estimate. The test RMSE of this model was 0.1513.

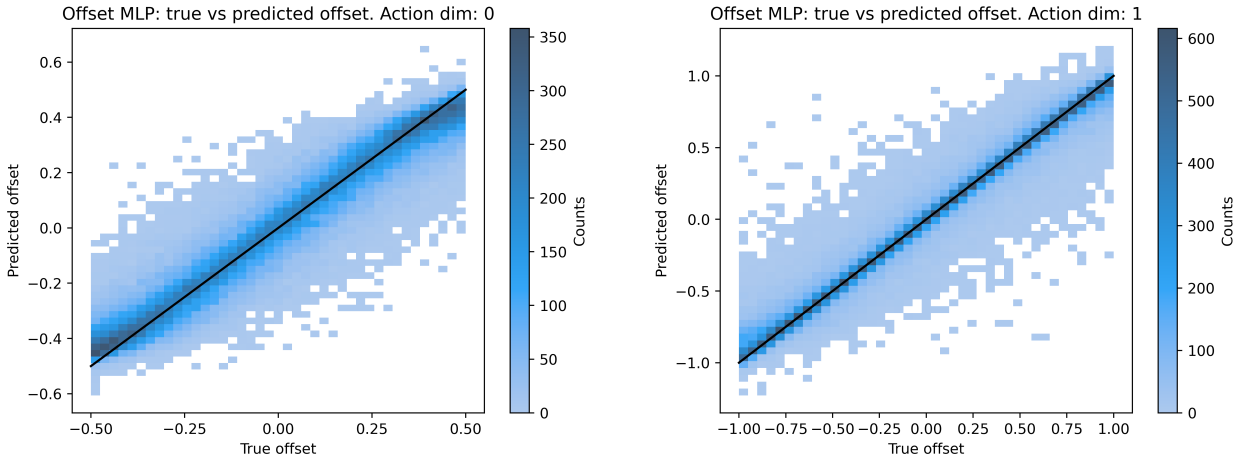

Figure 5: Offset MLP test performance for predicting action a_0 offset (left) and action a_1 offset (right). The points mostly hug the $y = x$ line (which would represent perfect performance), suggesting that the performance is quite strong for both action dimensions, especially action 1.

7.3 Appendix C: Meta-RL Training Curves

Below, we display the training curves for our meta-RL algorithms: RL^2 and PEARL.

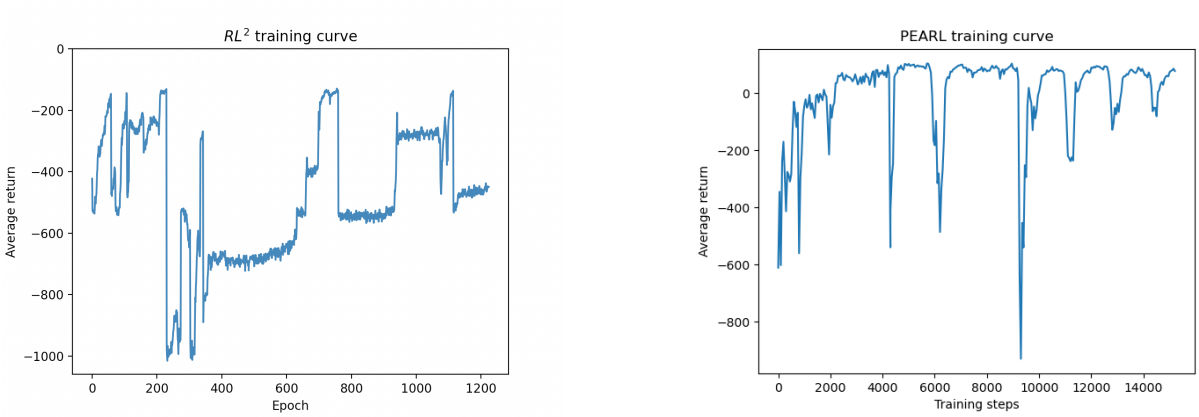

Figure 6: Training curves for our meta-RL methods: RL^2 (left) and PEARL (right).

It is clear that training was quite unstable for both algorithms. There are many points at which the performance increased quite a bit, only to drop down dramatically. We were never able to achieve satisfactory solutions using these methods, but we suspect that if we had more computing resources and could do a more extensive hyperparameter search, they would eventually find good solutions.