Math 156 - Final Project Report

Max Du, Henry Genus, Jeongwon Park, Temini Segun-Oderinde

July 28, 2020

Contents

1 Introduction

With the recent successes of deep learning and neural networks, the field of Reinforcement Learning (henceforth denoted RL) has been reincarnated. Many of these developments have occurred in a subfield called Deep Reinforcement Learning. Applications such as AlphaGo are proof of this revival, and arguably mark a monumental phase in the development of human knowledge. Despite its recent successes, however, standalone RL still struggles to achieve the performance necessary for mainstream usage. This project attempts to investigate some of the popular methods to mitigate this issue.

The sole goal of an RL agent is to maximise its expected cumulative rewards. This can be viewed as an agent learning through its own experiences, with the reward analogous to a 'signal' that points the agent in which direction to learn. But what happens if the agent's reward function is not appropriate for the task at hand? An agent that is given a single reward upon a successful completion of its full trajectory will fail to learn throughout the atomic steps within the trajectory, as the signals that are supposed to guide the agent are non-existent. Sparse reward functions like this are therefore inappropriate for this simple learner. In circumstances like this, methods such as imitation and inverse reinforcement learning instead attempt to learn a dense reward function, given some ideal trajectories completed by an 'expert'.

In addition, in a general MDP setting, an agent presented with a state faces a dilemma; it must decide whether to gather more data by exploring its environment, or to exploit past data, making the best decision given its limited current information. An agent that chooses to exploit prematurely often finds itself in a local optimum, which is a dilemma that is magnified when an agent is placed in an environment with sparse rewards. As a result, there have been a variety of methods proposed to address this. From this stems our motivation; we ask ourselves: How do we best model and utilize the uncertainties faced by the agent to best optimize the exploration exploitation trade off? What do humans do in their times of uncertainties? Can we utilize those real-life inspired strategies in a MDP setting?

This project investigates a state of the art policy gradient based algorithm called *Proximal Policy Optimization*[5]. We compare its performance on environments with both dense and sparse rewards. In addition, we investigate other popular imitation learning and exploration boosting algorithms called *Generative Adversarial Imitation Learning* [2] and *Variational Information Maximizing Exploration* [3], and compare their performance with the PPO baseline. Finally, we propose a new algorithm, GD-VIME, that attempts to combine these two methods, and we discuss experimental results.

2 Preliminaries

2.1 Markov Decision Process

The main components in RL are the *agent* and the *environment*. The basic framework which the agent uses to learn from continuous interaction with the environment is called a *Markov Decision Process* (henceforth denoted MDP). At each time step, the agent will go through a sequence vectors of observations and actions, and the environment will provide the agent with a scalar reward R. The agent seeks to maximise this reward over the entire trajectory $\{S_0, A_0, R_0, S_1, A_1, R_1...\}$. We denote the set of observations $S \subseteq \mathbb{R}^n$ and the set of actions $A \subseteq \mathbb{R}^m$.

In this project, we assume a finite and fully observed MDP, where each trajectory terminates with a finite set of S, A, R and the set of observations is equal to the set of states. S, R are random variables in a finite MDP, and the dynamics of the environment are modeled by the distribution

$$p(s_{t+1}, r|s, a).$$

The state transition probability s_{t+1} is given by

$$p(s_{t+1}|s, a).$$

Generally in RL literature this referred to as the system dynamics, so we shall refer to it as such. Both the

state and action vectors can be either discrete or continuous, but in this project we deal only with their discrete counterparts.

2.2 Policy

From here on, all discussions will be based solely within the context of deep RL. A policy $\pi(s)$ is a function that maps a state vector to a set of actions. We also assume a stochastic policy where the action is a random variable sampled from the distribution $\pi(.|s_t)$. One can view the policy as the agent's brain, since it is what ultimately makes decisions, and it can be modeled by an artificial neural network. In a discrete action setting, the action is sampled from a categorical distribution, which is attained by funneling the logits through a softmax function at the final layer of the neural network. We then interpret this output as a probability distribution; it then becomes convenient to compute the log likelihood

$$\log(\pi_\theta(a \mid s))$$

of particular actions from this distribution, with θ being the weights of the neural network.

Given that we have the policy, we can then frame the central optimization objective for RL. Recall that we assume the environment dynamics are stochastic and the MDP is finite. We can then write the probability and RL objective of a T step trajectory as

$$P(\tau \mid \pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1} \mid s_t, a_t)\pi(a_t \mid s_t)$$

$$J(\pi) = \int_\tau P(\tau \mid \pi)R(\tau) = \mathbb{E}_{\tau\sim\pi}[R(\tau)]$$

where $R(\tau) = \sum_{t=0}^{\tau}$. Thus the optimal policy is given by

$$\pi^* = \arg\max_\pi J(\pi).$$

Notice that the expected return is sampled with the given policy π, after which the expectation can be computed to obtain the optimal policy that maximises return.

2.3 Value

A value function $V(s)$ maps a state vector s_i to the expected total reward of the given state, given by

$$V(s) = \mathbb{E}[R(\text{trajectory}) \mid s_0 = s]$$

This can be interpreted quite literally — an agent that starts at a state with a high value will be expected to obtain high total returns. $Q(s, a)$ is an another formulation of the value function, wherein the function takes in the state-action pair to give

$$Q(s, a) = \mathbb{E}[R(\text{trajectory}) \mid s_0 = s, a_0 = a]$$

Normally in deep RL the value function is modeled by a neural network.

2.4 Advantage

One can also obtain the advantage function by

$$A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s, a).$$

Conventionally, the advantage measures how beneficial the particular action a taken at s is relative to all the possible actions at s. We will see later that advantage is generally computed from a sequence of state-action pairs, using a discount γ, which drives the optimization toward closer rewards rather than the more distant ones. We find $Q_\pi(s, a)$ from the collected discounted trajectories, and $V_\pi(s, a)$ is an estimator outputted by a neural network. This is generally incorporated in 'Actor Critic' style algorithms, with the actor being $Q_\pi(s, a)$ and the critic being $V_\pi(s, a)$.

2.5 Policy Gradients and Importance Sampling

This report discusses a class of RL algorithm, called Policy Gradients (henceforth denoted PG). Given a policy $\pi_\theta(a|s)$, PG methods aim to find the θ that parametrizes the optimal policy. In general, PG gives an empirical estimate from the collected trajectories of the current policy, and can be written as

$$\hat{g} = \hat{\mathbb{E}}_t \left[\nabla_\theta \log \pi_\theta(a_t \mid s_t) \hat{A}_t \right].$$

Given that the advantage function quantifies how advantageous the a_t is relative to other actions, a gradient ascent on this estimator will yield the θ that parametrizes the policy $\pi_\theta(a_t|s_t)$ that most increases the likelihood of sampling the advantageous action.

Vanilla PG methods, although mathematically appealing, are found to be empirically unstable and sample inefficient. This is because the $\hat{A}_t$ is merely a noisy estimator of the advantage. The *Natural Policy Gradient* and *Trust Region* methods [4] have been proposed to address this issue. The high level idea of these algorithms is to provide a clip to the policy updates when maximising the objective

$$L^{PG} = \hat{\mathbb{E}}_t \left[\log \pi_\theta(a_t \mid s_t) \hat{A}_t \right]$$

Specifically, the TRPO algorithm indirectly maximises the objective by instead optimizing a "surrogate" objective

$$L^{IS}_{\theta_{old}} = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \hat{A}_t \right]$$

Notice that this is in fact an important sampling method where we seek to estimate the behavior of the new policy π_θ with samples from the old policy $\pi_{\theta_{old}}$. The advantage of this surrogate objective is that we can repeatedly update our policy using without needing to sample or generate more data each time. A naive implementation of this objective, however, can in many occasions cause a destructively large policy update, where π_θ differs significantly from $\pi_{\theta_{old}}$. When the policy is faced with a 'bad' state due to the stochastic nature of the environment dynamics, an excessive update will decrease the algorithm's performance to the extent that is difficult to recover from. Thus a constraint on the update's difference

$$\hat{\mathbb{E}}_t \left[D_{KL} \left[\pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t) \right] \right] \leq \delta$$

is utilised to bound the updates of the policy. Schulman et. al effectively shows [5] that a Euclidean distance instead of the KL divergence of the policy parameters is mathematically equivalent to the objective of the vanilla Policy Gradient.

3 Established Algorithms

3.1 Proximal Policy Optimization

The more recent approach to conservative policy optimizations is a simpler one, which will serve as a baseline of the investigations of this project. PPO [5] is a successor algorithm of TRPO, which instead optimizes the value objective

$$L^{CLIP} = \hat{E}_t \left[\min \left(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) \hat{A}_t \right) \right]$$
$$\text{where } r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

and ϵ is a hyperparameter on the order of $\epsilon = 0.2$. Instead of setting the KL divergence between the new and old policies as a constraint, PPO instead updates the policy with the minimum of the clipped and unclipped objectives. Thus we use the min to set the clipped objective as the lower bound of the unclipped. The policy

update will be clipped by $1+\epsilon$ with a a positive advantage, and $1-\epsilon$ with a negative. The error function optimizes the minimization

$$L^{VF} = \text{MSE}(R(\tau), V(\tau))$$

where

$$\begin{aligned} \hat{A}_t &= \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t} V(s_T) \\ \delta_t &= r_t + \lambda V(s_{t+1}) - V(s_t) \end{aligned} \tag{1}$$

The policy and the value function share the same network parameters and only differentiate themselves by their final layer node; the final layer therefore outputs two separate values, the action distribution obtained by the softmax function and the estimated value logit of a state. The final loss optimized is thus

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t[L_t^{CLIP} - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)] \tag{2}$$

where $S[\pi_\theta]$ is an entropy regularizer which prevents the policy from being too certain about its actions.

The Pseudocode for implementing PPO is as below

Algorithm 1: PPO

Result: policy and value network parameters
for *epochs=1,2,...,T* **do**
  for *actor=1,2,...,N* **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps;
    Compute advantage estimates $\hat{A}_i$ by (1)
  end
  Update θ by optimizing (2) wrt θ, with K epochs and minibatch size $M \leq NT$;
end

Policy optimization methods as presented above are often quite stable and reliable, being the case that we directly optimize for the thing we want — advantageous actions. However, these methods are weak when it comes to maximising exploration, as the only stochasticity is introduced when sampling from the categorical distribution of the network's output. The entropy regularizer $h(\pi)$ serves to mitigate this issue, but it is still generally futile when the agent faces an environment with extremely sparse rewards.

3.2 Variational Inference Maximizing Exploration

Numerous methods have been proposed to resolve the exploration-exploitation dilemma. VIME [3] is a method based on the maximization of information gain about the environment dynamics. Recall that the sole purpose of an agent is to maximise $\mathbb{E}[R]$. At a high level, VIME attempts to add an additional intrinsic reward to $\mathbb{E}[R]$, and this intrinsic reward is the information gain of the state transition probabilities of the environment.

In VIME, in addition to performing a baseline RL algorithm, the environment dynamics

$$p_\theta(s_{t+1}|s_t, a_t)$$

are learned by a Bayesian neural network (henceforth denoted BNN). A BNN is simply a network parametrized by weights θ sampled from probability distributions. This serves to capture the uncertainty of the agent, where θ is analogous to the agent's 'belief' in the dynamics. The BNN is then trained in a supervised fashion with the sampled (s, a) pairs as inputs and s_{t+1} as outputs.

Information gain is characterized as the quantification of the information gain on z, after taking action a, and observing y. This can often be estimated from

$$D_{KL}(p(z \mid y) \parallel p(z)),$$

the KL divergence between the conditional distribution of z given y and the prior distribution of z. This measures to what extent the observation of the variable y changes the distribution of z. $p(z|y)$ will always

have an entropy value less than or equal to that of $p(z)$, and the difference in entropy quantifies how much is learned about the distribution of z after observing y. A paper by Houthooft et al. [3] instantiates this idea and formulates an additional objective for the RL agent, where θ from $p_\theta(s_{t+1}|s_t, a_t)$ is analogous to z and the set $\{\xi_t, a_t, s_{t+1}\}$ is analogous to y from the equation above. The final reward function for the agent then can be written as

$$r'(s_t, a_t, s_{t+1}) = r(s_t, a_t) + \eta D_{KL}[p(\theta \mid \xi_t, a_t, s_{t+1}) \parallel p(\theta \mid \xi_t)], \tag{3}$$

where ξ_t defines the entire history of trajectories, hitherto captured by the bayesian NN, and η is a hyperparameter. The second term is now the intrinsic reward which quantifies the agent's 'curiosity' of its environment, and η can be interpreted as its urge to explore.

Assuming a fully Bayesian setting, the posterior distribution used to compute the KL divergence is expressed as the following:

$$p(\theta \mid \xi_t, a_t, s_{t+1}) = \frac{p(\theta \mid \xi_t)p(s_{t+1} \mid \xi_t, a_t; \theta)}{p(s_{t+1} \mid \xi_t, a_t)}$$

However, with neural networks it is generally intractable to compute the integral that comprises the denominator. Thus the authors of this paper have chosen to apply variational inference and approximate the posterior with an alternative distribution $q(\theta; \phi)$ by minimizing

$$D_{KL}[q(\theta; \phi) \parallel p(\theta|D)]. \tag{4}$$

This is explicitly achieved through gradient ascent on the evidence lower bound,

$$E_{\theta \sim q(\cdot;\phi)}[\log p(D \mid \theta)] - D_{KL}[q(\theta; \phi) \parallel p(\theta)],$$

where ϕ parametrizes the BNN, $D = \xi_t, a_t, s_{t+1}$, and a zero mean Gaussian distribution is assumed for the prior. The authors also suggest a fully factorized Gaussian distribution for the parameters.

The new reward function that captures this approximation is

$$r'(s_t, a_t, s_{t+1}) = r(s_t, a_t) + \eta D_{KL}[q(\theta; \phi_{t+1}) \parallel q(\theta; \phi_t)],$$

which can be obtained through computing, at each timestep, the KL divergence between the old and new parameters of the BNN. Recall that the parameters of the BNN represent the agent's belief in the dynamics of its environment. The updated belief required for computing the KL divergence is obtained by

$$\phi_{t+1} = \underset{\phi_t}{\operatorname{argmin}} \left(D_{KL}[q(\theta; \phi) \parallel q(\theta; \phi_{t-1})] - \mathbb{E}_{\theta \sim q(\cdot;\phi)}[\log p(s_t \mid \xi_t, a_t; \theta)] \right). \tag{5}$$

As a side note, this is implemented as a temporary update of the BNN's parameter to obtain ϕ_{t+1} and compute $D_{KL}[q(\theta; \phi_{t+1}) \parallel q(\theta; \phi_t)]$, after which the model will be reset to its original parameters.

To summarize the procedure by pseudocode,

Algorithm 2: VIME

Result: policy and value network parameters
for $n = 1, 2, ...$ **do**
  for $t = 1, 2, ...$ **do**
    generate the current state and action $\{\tau\}$ from policy π_θ;
    compute a new value of ϕ'_{n+1} by equation (5);
    update r' according to equation (3);
  end
  add attained intrinsic reward values to $R_\tau = \sum r$;
  train model through updating the posterior q by maximising ELBO;
  perform gradient descent on policy wrt cost R_τ;
end

3.3 Generative Adversarial Imitation Learning

Generative Adversarial Networks [1] (henceforth denoted GAN) belong to a class of generative models, where a distribution generated from a generator network is discriminated against the distribution of real data. This is done by a separate discriminator network. In a conventional GAN, a class 1 is assigned to the distribution of real data and a class 0 is assigned to the distribution generated from the generator. The discriminator's role is to assign the probability $D(G(z))$ of the generated data being in either class 1 to the input data $G(z)$, where z is a latent distribution for the generator to sample from. We will see later that this framework is easily transferable to GAIL.

Recall that the objective of inverse RL is to obtain a reward function from expert trajectories, and optimize the policy wrt to that reward function. From here on we utilise a cost function

$$c(s,a) = -r(s,a)$$

with expert policy trajectories given by π_E. Let us then consider a maximum causal entropy inverse RL problem [6] where the objective is

$$\underset{c\in C}{\text{maximize}}(\min_{\pi\in\Pi} -H(\pi) + \mathbb{E}_{\pi}[c(s,a)]) - \mathbb{E}_{\pi_{\mathbb{E}}}[c(s,a)]$$

where $H(\pi)$ denotes the causal entropy regularizer of policy π, In the context of inverse RL, where we assume a finite amount of expert data, maximisation of the entropy term minimizes the probability of selecting the worst policy [6]. This objective then finds a cost function that assigns a low cost to the expert policy and a high cost to the generating policy, subject to the entropy term.

Let us extend this inverse RL problem to the context of deep RL, where we can utilise expressive function approximators like neural networks to model the cost function. A powerful approximator will easily overfit its parameters to the given expert data, so the authors introduce a convex cost function regularizer $\psi(c)$. The overall optimization procedure then can be written as

$$\underset{\psi}{\mathbf{IRL}}(\pi_E) = \underset{c\in\mathbb{R}^{\mathcal{S}\times\mathcal{A}}}{\operatorname{argmax}} \left(-\psi(c) + \min_{\pi\in\Pi} \big(-H(\pi) + \mathbb{E}_{\pi}[c(s,a)]\big) - \mathbb{E}_{\pi_{\mathbb{E}}}[c(s,a)] \right)$$

Inverse RL obtains an optimal cost function c for this objective, given by

$$\mathbf{RL}(c) = \underset{\pi\in\Pi}{\operatorname{argmin}} \left(-H(\pi) + \mathbb{E}_{\pi}[c(s,a)]\right).$$

Standard RL then uses the obtained cost function to obtain the optimal policy.

This is a two step process, one that can be inefficient and expensive. The authors of GAIL then propose a new optimization scheme as to combine the two steps,

$$\mathbf{RL} \circ \mathbf{IRL}_{\psi}(\pi_E) = \underset{\pi\in\Pi}{\operatorname{argmin}} \left(-H(\pi) + \psi^*(\rho_\pi - \rho_{\pi_E})\right) \rho_\pi(s,a) = \pi(a \mid s) \sum_{t=0}^{\infty} \gamma^t P\left(s_t = s \mid \pi\right)$$

where ψ^* is the convex conjugate of ψ and ρ_π denotes a new term, *occupancy measure*, representing the distribution of state-action pairs generated by policy $\pi(a|s)$. The new objective seeks to directly obtain an optimal policy that minimizes the occupancy measure between the expert policy and the generating policy.

Now the authors choose the cost regularizer explicitly to match the $\boldsymbol{RL} \circ \boldsymbol{IRL}_{\psi}$ objective to that of GANs.

$$\psi_{\mathrm{GA}}(c) \triangleq \begin{cases} \mathbb{E}_{\pi_E}[g(c(s,a))] & \text{if } c < 0 \\ +\infty & \text{otherwise} \end{cases} \quad \text{where } g(x) = \begin{cases} -x - \log\left(1 - e^x\right) & \text{if } x < 0 \\ +\infty & \text{otherwise} \end{cases}$$

which allows us to define its conjugate function.

$$\psi^*_{\mathrm{GA}}\left(\rho_\pi - \rho_{\pi_E}\right) = \sup_{D\in(0,1)^{\mathcal{S}\times\mathcal{A}}} \mathbb{E}_{\pi}[\log(D(s,a))] + \mathbb{E}_{\pi_E}[\log(1 - D(s,a))].$$

This is the *Jensen Shannon divergence*, a symmetric distance measure of distributions of the two occupancy measures. This objective finds a D that assigns a high cost to the generator's trajectory and a low cost to the expert trajectory. This ultimately leads to a training procedure very similar to GANs, but where a neural network discriminates the trajectories generated by our policy neural network from the expert data. The two networks then find a saddle point of the following optimization objective,

$$\underset{\pi \in \Pi}{\text{minimize}}\, \psi^*_{\text{GA}} \left(\rho_\pi - \rho_{\pi_E}\right) - \lambda H(\pi) = \mathcal{D}_{JS}(\rho_\pi, \rho_{\pi_E}) - \lambda H(\pi) \tag{6}$$

, as the discriminator and the generator takes consecutive steps of gradient ascent and descent wrt to this objective, in respective order. In implementation, the expert and the policy's data are labeled with 1 and 0 respectively, and the discriminator optimizes the binary cross entropy loss for the two classes.

To summarize the procedure by pseudocode,

Algorithm 3: GAIL

Result: policy and value network parameters
for *epochs = 1, 2, ...* **do**
  sample trajectories $\{\tau\}$ from policy π_θ and obtain cost $D(s, a)$ from the discriminator for each timestep;
  train discriminator with BCE loss;
  take a gradient descent step on policy that minimizes cost;
end

4 GD VIME

We now introduce a combination of the two methods presented above, and coin the term 'GD VIME' as a name for our algorithm. Much like VIME, GD VIME incorporates a discriminator network when processing rewards. Recall that in VIME, the additional intrinsic reward captured by the KL divergence of the old and updated model dynamics parameters is added to gross reward. This serves to increase the agent's urge to explore. This may naturally lead one to formulate the question: can we somehow discriminate against this exploration?

Let us reformulate our motivation in intuitive terms. Imagine being a toddler trying to master the field of mathematics. You have multiple lives to waste and earn a single reward of 1 when you earn a phD in mathematics, or perish in the process if unsuccessful in some arbitrary number of years. At first you will have no clue of which books to read nor even the vaguest notion of mathematics. A natural way of resolving this is to try out many things only to find yourself stumbling upon books and courses unrelated to mathematics in the most abstract sense. Then you will slowly but surely learn from distant memories of your previous lives to choose the resources which maximise your probability of earning a phD. One might ask, wouldn't you be more successful if someone, like a math professor, who truly knows what gives rise to a good mathematician was able to judge the ill fitted nature of your exploration?

This is an important issue with VIME. An agent naively maximising exploration, while able to surpass the local minima, will still not be very successful. Thus we introduce a discriminator network that, rather than being trained with (s, a) pairs, is trained with transitions (s, a, s_{t+1}) of the policy and expert's trajectories. This is done to permit the discriminator to learn both the transitions and the occupancy measures of the two trajectories, which allows it to discriminate the "exploration" of the agent.

Similar to GAIL, we train a discriminator to output a probability $D(s_t, a_t, s_{t+1})$ that the input transition belongs to class 1, the expert data. We then obtain the following reward function for the policy to maximise:

$$\lambda^k \exp(D(s_t, a_t, s_{t+1}) - 1)R_\tau + \eta D_{\text{KL}}[q(\theta; \varphi_{t+1}) \parallel q(\theta; \varphi_t)], \quad D \subseteq [0, 1]. \tag{7}$$

Here R_τ is raw reward obtained from trajectory τ, k is iteration index, and $0.99 < \lambda < 1$ is a hyperparameter. As $k \to \infty$, the agent will increasingly put more emphasis on exploration. This gels with the intuition that it would exhaust its expert knowledge and should theoretically minimally be able to match the performance of π_E.

Pseudocode is similar to those from GAIL and VIME:

Algorithm 4: GD VIME

```
for epochs = 1, 2, ... do
    for t = 1, 2, ... do
        generate trajectory {τ} from policy π_θ;
        compute information gain of agent's state transitions belief;
    train discriminator with BCE loss from expert and PPO transitions;
    obtain D(s_t, a_t, s_{t+1}), process rewards as (7);
    train BNN;
    take gradient descent step on policy that maximises (7);
```

5 Discussion

We demonstrate the algorithms' performance on two classical control tasks. In the environment CartPole, an agent receives a reward of 1 every timestep it successfully balances a cartpole. The episode ends when the agent fails to meet the balance criteria for some number of timesteps. The maximum possible reward is 200. In the environment MountainCar, an agent receives a reward of -1 for every timestep it fails to reach the top of a hill. The episode ends after 200 timesteps, or when the agent reaches the top. Notice that this reward function doesn't provide any feedback for agents that fail the task consecutively. The plots in the appendix show results of the best performing combination of hyperparameters from the ones we were able to experiment with, and the varying number of iterations were compensated with higher or lower batch sizes.

Notice that our version of PPO is able to solve the task given in CartPole, but is unable to learn anything in MountainCar. Recall that the sole stochasticity inherent to vanilla PPO is $a \sim \pi(s)$, and the entropy regularizer fails to aid in extreme sparse reward settings. As an on-policy algorithm, PPO doesn't utilise past experiences after gradient updates. Thus reaching the top of a mountain from random actions is highly unlikely, and even if it does, it is also highly unlikely that a single gradient ascent step on that advantage will be enough to make it visit again.

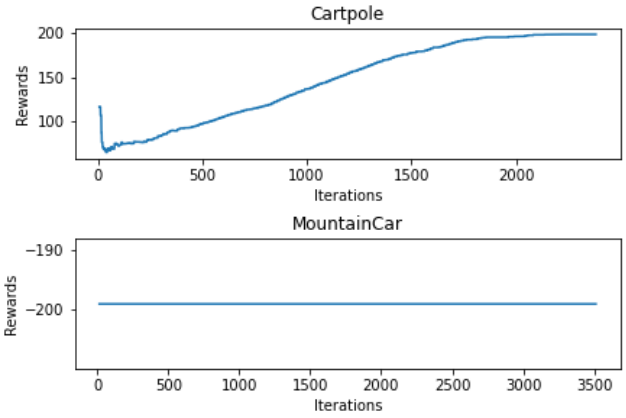
VIME is able to train a policy that at least sends the agent to the top of the hill, subject to the given timesteps limitations. Although it is a theoretical principled algorithm, as is GAIL, in practice it can be difficult to implement. The program exceeds 1000 lines of code, and the sheer number of hyperparameters one needs to optimize decreases the robustness of the algorithm. This is in addition to the limitations discussed earlier in this paper. Also, VIME learns the dynamics of the model but does not utilise this learned model asides from computing information gain.

GAIL is our most successful algorithm implemented. We use only 10 expert trajectories to train the discriminator network, each with an expert MountainCar trajectory of -120 rewards. Notice that GAIL is even able to slightly surpass the expert's performance, and in a sample efficient manner. As the policy progressively generated expert-like trajectories, the discriminator was unable to perform classification better than random. We also emphasize the theoretical soundness of this algorithm; the authors, by utilising the conjugate properties of a cost regularizer, successfully derive Jensen-Shannon divergence minimization objective similar to that of GANs from the inverse RL problem.

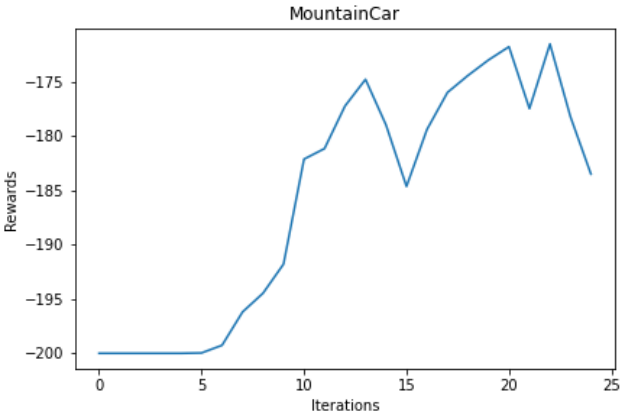
At its peak, our algorithm seems to perform marginally better than vanilla VIME, but doesn't quite reach the performance of GAIL. The performance also drop sharply after some number of iterations, and we find this to be true across all our experiments. We deduce the issue is with increasing value of K, and delete the λ term to find similar results.

It was quite challenging, within the constraints of the due date, to correctly diagnose the issues of our algorithm. One might argue that our techniques are crude attempts at combining the two. Given more time, we would work on gaining a deeper theoretical understanding of GAIL and VIME, which will lead to a more principled insight into combining the two algorithms. The concept stands, however, and we believe we have demonstrated non-trivial performance changes which justify future exploration.

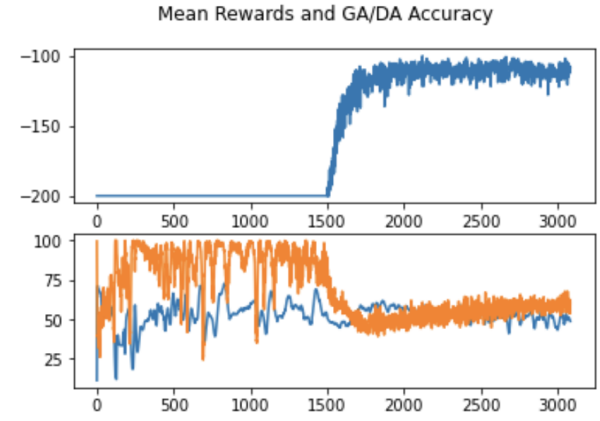
A Graphs



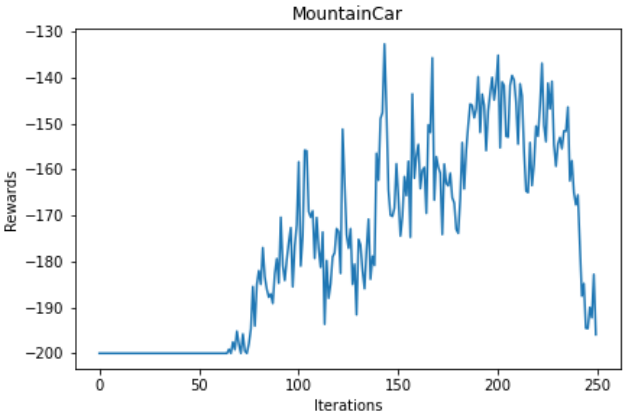
(a) Testing with vanilla PPO algorithm.



(b) Testing with VIME algorithm.



(c) Testing with GAIL algorithm.



(d) Testing with GDVIME algorithm.

Figure 1: Performance of various algorithms on CartPole and MountainCar

Code for this project: https://github.com/jpark0315/pytorch-PPO-VIME-GAIL

References

[1] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[2] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning, 2016.

[3] Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. VIME: Variational Information Maximizing Exploration. *arXiv e-prints*, page arXiv:1605.09674, May 2016.

[4] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2015.

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

[6] Brian D. Ziebart, Andrew Maas, J. Andrew Bagnell, and Anind K. Dey. Maximum entropy inverse reinforcement learning. In *Proc. AAAI*, pages 1433–1438, 2008.

Repo

Code for this project: https://github.com/jpark0315/pytorch-PPO-VIME-GAIL.
Code snippets are quite long, please excuse us for not attaching them to this pdf directly.