Imitation Guided Reinforcement Learning For Autonomous Vehicle Navigation

Youssef Farag
Department of Computer Science
University of British Columbia
yous@yous.dev

Abstract

Leveraging expert demonstrations through imitation learning has seen much success in autonomous vehicle navigation tasks. However, imitation learning often results in brittle policies, and expert demonstrations can be costly to collect. Here we explore two methods for integrating imitation learning policies into reinforcement learning pipelines in order to address these shortcomings. These approaches are evaluated on a toy formulation of the driving task. We justify this decision with an experiment based on a more realistic problem formulation, using the urban driving simulator CARLA.

1 Introduction

Autonomous vehicle navigation is a problem domain that continues to garner significant interest, due to its far-reaching ethical and economic implications. Successfully solving this task on a global scale means saving lives and reshaping modern society. Beyond this, however, the problem itself presents another interesting motivation: how can we derive a policy for successful navigation of a complex, spatio-temporally dynamic environment, currently populated by humans? Specifically, an urban driving environment, where each agent's actions are heavily dependent on those around them, and mistakes or missteps can be costly.

The field of machine learning offers a number of frameworks suitable for exploring this task. Unsurprisingly, there is a supervised approach, and an unsupervised (arguably semi-supervised) approach. Supervised learning in this domain takes the form of imitation learning, whereas when labelled data is not available the common approach tends to be reinforcement learning. Each of these frameworks suffers from shortcomings; reinforcement learning has notoriously high variance experiments, low sample efficiency, and requires a carefully chosen reward function. Imitation learning requires data collection and produces brittle policies in regions underrepresented in the data. Combining these methods however could alleviate these issues. The addition of a small amount of expert data into a reinforcement learning pipeline could improve sample efficiency, reduce the variance of each experiment, and lessen the impact of the reward function on learning.

The main motivation for this study is to explore the limits of these two learning frameworks and investigate methods for combining them in this particular application; autonomous navigation in simulation. We present two such hybrid formulations, along with a series of experiments outlining the particular characteristics of each approach.

Naturally, for a task of this scope, we make a number of significant abstractions in order to effectively test our methods. First, we leverage computer simulators instead of training driving agents in the real world. This is obviously more cost-effective, and allows us to experiment with much more flexibility. Our methods make little to no assumptions on the environment however, and so they would be expected to generalize to the real-world task. Additionally, we train with only a single agent

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

in the environment, rather than including any 'traffic'. This greatly reduces both the complexity of the task and the time taken to simulate the environment. With this admittedly much simpler problem formulation, we can more effectively compare a number of different pipelines without having to dedicate a significant amount of time to hyperparameter tuning, model architecture selection, etc.

2 Background

2.1 Imitation Learning

One framework commonly applied in this domain is imitation learning (IL), a form of supervised learning. The key intuition is that leveraging prior knowledge from an expert is likely to be more efficient, if not more successful, at deriving candidate policies than an optimization approach over all possible policies [1]. Given a set of labelled expert demonstrations of a task, the goal is to derive a policy for the task that maximizes behavioural similarity to the expert. In an IL framework, a dataset is collected, consisting of a set of states $s_i \in S$ and the corresponding actions $a_i \in \mathcal{A}$ taken by the expert to be imitated. A family of policies is defined, and its free parameters are optimized with respect to some objective function J. With the advancement of deep learning, common practice is to use neural networks for π, making the parameters θ a set of neural network weights. The objective J is often some measure of distance $\mathcal{D}$ between the current policy's outputs $\hat{a}_i$, and the expert demonstrations a_i, such that improvement on the task corresponds to minimizing the difference. The objective can then be abstractly presented as in [2]:

$$\theta^* = \arg\min_{\theta} J(\theta) = \arg\min_{\theta} \mathcal{D}(\pi_e(a_i|s_i), \pi_\theta(\hat{a}_i|s_i)) \tag{1}$$

where π_e is the expert policy distribution induced by the dataset. Algorithm 1 below describes our IL pipeline, which is effectively the vanilla supervised learning pipeline with mean-squared error loss. Applied to the autonomous navigation domain, IL holds potential given the success of deep learning and relative ease of implementation. In most cases however, the quality of the policies produced by these approaches depends heavily on the quality of data collected, and have they often difficulty generalizing to states not sufficiently represented in the expert demonstrations [3]. However, careful data collection and augmentation techniques in an IL framework have been demonstrated to ameliorate these issues, leading to surprisingly successful agents [4],[5].

Algorithm 1: Vanilla Imitation Learning

Result: Trained IL policy

initialization; states S, corresponding expert actions A, randomly initialized neural network π_{θ_0}, learning rate $\eta \in (0, 1)$, batch size n, number of epochs K;

for $k = 0, 1, ..., K-1$ **do**

  Shuffle data, uniformly draw n samples of (s_i, a_i) tuples;

  for $i = 1, 2, ..., n$ **do**

    $J(\theta_k) \leftarrow \frac{1}{n}\sum_{i=1}^{n}(a_i - \pi_\theta(\hat{a}_i|s_i))^2$; calculate loss on predictions

  end

  $\theta_{k+1} \leftarrow \theta_k - \eta\nabla_\theta J(\theta_k)$; update policy parameters

end

Return: π_{θ_K}

2.2 Reinforcement Learning

Reinforcement learning (RL) explores how agents ought to take actions in an environment in order to maximize the notion of cumulative reward. There are two core components to this framework: the problem formulation, and the optimization objective. This framework does not assume access

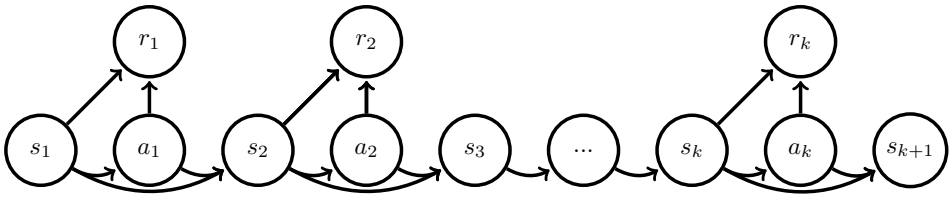

Figure 1: Example of a probabilistic graphical model of an MDP.

to expert data in the form of a labelled dataset like IL, but rather in the form of a reward function. Because of this, it can be argued whether RL is truly unsupervised. This reward function is defined in the scope of a Markov Decision Process, which formalizes the task such that it can be framed as RL.

2.2.1 Problem Formulation: Markov Decision Process

A Markov Decision Process (MDP) is a mathematical formulation of a control process over discrete time steps. It describes an agent acting in some environment, and provides a framework for modeling the decision making of this agent. Outcomes inside this environment can be entirely deterministic, or partly under the agent's control and partly stochastic. This MDP plays the role of the labelled dataset in IL, providing state-action pairs for training along with the reward. In the absence of expert actions, the reward function acts as a performance measure, judging the quality of actions taken based on the task formulation.

Formally, an MDP consists of:

- set of states $s \in S$
- set of actions $a \in A$
- state transition dynamics $\mathcal{P} := p(s_{t+1}|s_t, a_t)$
- initial state distribution $\mathcal{P}_0 := p(s_0)$
- reward function $r =: S \times A \to \mathbb{R}$
- policy $\Pi := \pi : S \to A$

We can then denote this MDP as $\mathcal{M}(S, A, \mathcal{P}, \mathcal{P}_0, r, \Pi)$.

In our particular formulation, our environment is the CARLA urban driving simulator [6], which provides the transition dynamics $\mathcal{P}$ and defines the form of the states S and actions A. If there is only one agent in the environment, then the state transitions depend only on this agent's state and action, and the reward function becomes deterministic. Given some r, and starting state sampled from $\mathcal{P}_0$, we can define a policy $\pi(a_t|s_t)$ and 'run' our MDP, producing a sequence of tuples $\tau_t := \{s_t, a_t, s_{t+1}, r_t\}$. This set, τ is a trajectory, representing the agent's complete roll-out through the decision process. In RL, performing multiple optimization steps on the same trajectory has been shown to be unsuccessful [7], so we collect many. These trajectories, which can be constantly re-sampled, comprise our training dataset. This means that, unlike IL, the agent is not necessarily exposed to a fix set of training examples, since we can apply the agent's latest policy in the environment and gather new data (this is called on-line learning).

2.2.2 RL Optimization Objective

Having formalized our autonomous navigation task in this way, we can now leverage reinforcement learning to derive a solution. The MDP provides us with a clear directive: find the policy π_θ that maximizes the expected cumulative reward of an agent acting inside this process. Formally:

$$\max_\theta \quad \mathop{\mathbb{E}}_{\tau \sim q_{\pi_\theta}} \left(\sum_{t=1}^{T} r(s_t, a_t) \right) \quad \text{where:} \quad q_{\pi_\theta}(\tau) = p(s_0) \prod_{t=0}^{T-1} p(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t). \tag{2}$$

Note that in the case of infinite-horizon problems ($T = \infty$), reward must be discounted such that convergence is possible. This can be absorbed into the reward function, making it time-dependent ($r'_t = \delta^t r_t$ where $\delta \in (0, 1)$).

Typically in RL, we define first an MDP for the task, then some objective $J(\theta)$ for maximizing reward received from this MDP (given a set of policy parameters θ). The policy π_θ is parameterized by some function or distribution, once again a neural network architecture in this case. Stochastic gradient ascent is then used to update the weights θ of the policy π according to the objective J.

$$\theta^* = \arg\max_\theta \mathop{\mathbb{E}}_{\tau \sim q_{\pi_\theta}} [\sum_{t=1}^{T} r(s_t, a_t)] = \arg\max_\theta J(\theta)$$

Policy gradient methods are one common set of approaches, which maximize reward through repeated estimates of the gradient $\nabla_\theta \mathbb{E}[\sum_{t=1}^{T} r(s_t, a_t)]$. As presented in [8], many different policy gradient objective formulations take the following form:

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau \sim q_{\pi_\theta}} [\Psi_t \nabla_\theta \log \pi_\theta(a_t|s_t)] \tag{3}$$

This breaks down the objective into two terms; the "goodness" measure Ψ_t and the ascent direction $\nabla_\theta \log \pi_\theta$. The log term on the right corresponds to the policy update, the direction in which to move the parameters θ such that the performance of the policy π_θ improves. We then assign this gradient a weight Ψ_t, corresponding to some measure of goodness. This can be simply the total reward of the trajectory $\sum_{t=1}^{T} r(s_t, a_t)$, some measure of improvement over a baseline, or a measure of improvement over time. Here we present a simple on-policy, finite-horizon RL algorithm with the abstract objective J from Equation (3):

Algorithm 2: On-Policy Finite Horizon Policy Gradient Reinforcement Learning

Result: Trained RL policy

initialization; MDP $\mathcal{M}(S, A, r, \mathcal{P}, \mathcal{P}_0, \pi)$, randomly initialized neural network π_{θ_0}, learning rate $\eta \in (0, 1)$, trajectory length T, number of epochs K;

for $k = 0, 1, ..., K-1$ **do**

- $s_0 \sim p_0(s_0)$; — sample starting state
- **for** $i = 0, 1, ..., T-1$ **do**
  - $a_i \leftarrow \pi_{\theta_k}(a_i|s_i)$; — generate predictions from policy
  - $r_i \leftarrow r(s_i, a_i)$; — compute reward for predicted actions
  - $s_{i+1} \leftarrow p(s_{i+1}|s_i, a_i)$; — transition to next state based on action
- **end**
- $\nabla_\theta J(\theta_k) = \mathbb{E}_{\tau \sim q_{\pi_\theta}}[\Psi_t \nabla_\theta \log \pi_\theta(a_t|s_t)]$; — compute goodness score and gradient
- $\theta_{k+1} \leftarrow \theta_k + \eta \nabla_\theta J(\theta_k)$; — update policy parameters

end

Return: π_{θ_K}

Commonly, RL methods suffer from poor scalability and sample efficiency, and produce high variance experiments [7]. They are also highly dependent on the specified reward function, such that success of the overall procedure is tightly coupled with the reward surface over which it explores. In selecting the particulars of our own RL pipeline, we searched for methods that addressed some of these shortcomings.

3 Methods: IL assisted RL

3.1 Proximal Policy Optimization

There are two common concerns when defining the objective function in RL. Ψ_t tends to have very high variance, especially early in training when the model has not learned much. As a result, the

gradient term can produce very large (unstable) updates. Proximal Policy Optimization (PPO) [7] is a method that ameliorates both of these issues, leveraging a low-variance goodness function $\hat{A}_t$ presented in [9], alongside a novel policy update function.

Here we introduce the advantage function A_t, a general class of functions used for Ψ_t. Advantage functions aim to estimate the advantage of taking a particular action in the current state. The advantage is calculated based on discounted value function estimates $V(s_t)$, where V is a learned function (neural network) that estimates the cumulative expected reward from being in state s_t. The PPO objective is not limited to this particular formulation of A_t, however in practice the following advantage function has been shown to produce low-variance returns [8]:

$$\hat{A}_t = -V(s_t) + r_t + \gamma r_{t+1} + ... + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T) \tag{4}$$

where $\gamma \in (0, 1)$ is a discount factor and $t \in [0, T]$ specifies the time index in a given length-T trajectory. The other component of the objective that is has not yet been specified is the policy gradient calculation. PPO presented a novel objective function based on the above advantage and a slightly modified version of the Trust Region Policy Optimization (TRPO) [10] objective. The PPO objective is as follows:

$$J_{PPO}(\theta) = \mathop{\mathbb{E}}_{\tau \sim q_{\pi_\theta}} [min(R_t(\theta)\hat{A}_t, clip(R_t(\theta)\hat{A}_t, 1 \pm \epsilon))] \quad \text{where: } R_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{5}$$

Here $\epsilon \in (0, 1)$ is a hyperparameter that bounds the policy update size [1]. This is very similar to the TRPO objective, which took the form $\mathbb{E}[R_t(\theta)\hat{A}_t]$. Maximizing this objective increases the probability of selecting actions, under the current policy, that return high rewards relative to the previous iteration's policy. However, the TRPO method required a constraint on the size of this update. The addition of the $clip$ term in PPO achieves exactly this. It limits size of the update such that positive-advantage steps are not allowed to dominate completely, and negative-advantage steps are mitigated in magnitude.

In practice, this objective is easy to implement and cheap to compute, making it ideal for large state size, continuous action domains like autonomous navigation. Because of this, we use PPO as our baseline RL method, and explore two ways in which this pipeline can be injected with prior expert knowledge in the form of an imitation-learned policy.

3.2 Reward-Reinforced Imitator

The first and most obvious way to incorporate prior knowledge in an RL pipeline is to initialize our procedure with a policy that already has some knowledge of the task. We call this method the Reward-Reinforced Imitator (RRI), since we first train an imitation learning agent, and apply RL to further improve its performance. This requires no algorithmic modification to either approach as they have been presented here; the only constraint is that the policies must be of the same form. In this case, this means that the two neural network architectures from each pipeline must be identical. We take the trained weights θ_{IL} produced by running Algorithm 1 on some expert dataset, and initialize our RL procedure Algorithm 2 such that $\pi_{\theta_0} = \pi_{\theta_{IL}}$.

3.3 Imitation Bonus RL

Another way to bootstrap an RL agent with expert knowledge is to augment the reward function with a bonus corresponding to how close the current policy π_θ is to some expert policy π_{expert}. Unsurprisingly, we choose a policy produced by an IL pipeline as our π_{expert}. Once again, this requires no algorithmic modification to either pipeline, since the change can be absorbed into the reward function defined for $\mathcal{M}$ in Algorithm 2, replacing the original reward r_t with:

$$r'_t(s_t, a_t) = r(s_t, a_t) + \lambda \log \frac{\pi_{IL}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \tag{6}$$

[1] $clip(x, a \pm b)$ is a function that returns x if $a - b < x < a + b$, else the closer of the two bounds to x

where λ is a hyperparameter controlling the temperature of the bonus. In expectation, the bonus $\log$ term corresponds to the Kuhlback-Leibler divergence between the two policies. We want to minimize $D_{KL}(\pi_\theta||\pi_{IL})$ such that the learned RL policy remains close in distribution to the trained IL policy. Since our procedure maximizes reward, we subtract the negative of the divergence (equivalent to adding the inverse of the $\log$ term), such that the policy can maximize its reward by minimizing its distance from the IL expert policy.

$$D_{KL}(\pi_\theta||\pi_{IL}) = \mathop{\mathbb{E}}_{a_i\sim\pi_\theta}[\log \frac{\pi_\theta(a_t|s_t)}{\pi_{IL}(a_t|s_t)}] = \mathop{\mathbb{E}}_{a_i\sim\pi_\theta}[-\log \frac{\pi_{IL}(a_t|s_t)}{\pi_\theta(a_t|s_t)}] \tag{7}$$

4 Experiments

The urban driving simulator CARLA [6] affords us the opportunity to evaluate our methods on the autonomous navigation task. At the outset of this project, the goal was to train two baseline policies in the CARLA simulator; π_{IL} and π_{PPO}. This would have then allowed us to generate two more policies, π_{RRI} and π_{IBRL}, both leveraging the trained IL policy. We could then compare the three RL policies (baseline, RRI, IBRL) on a variety of task-performance metrics, as well as convergence rates.

However, using images as input generally requires far larger neural network policies to achieve task success. This means more hyperparameter tuning and slower pipelines from utilizing the simulator. Unfortunately, due to a variety of logistical issues [2], this approach quickly became infeasible. Instead, we have developed a toy version of the problem, termed Fake CARLA, which still manages to evaluate one of the methods proposed here. We cannot sufficiently test RRI with this method unfortunately, since this is a relatively easy task for IL, leaving no room for improvement with RL. We also demonstrate the difficulty of applying these methods in Real CARLA even with small policies, showing the poor sample efficiency of vanilla RL.

For both of these experiments, the action space formulation we use follows from the CARLA simulator. The actions are defined as $\mathbf{a}_t \in A = (Str \times Brk \times Thr)$, where:

- $str \in Str = [-1, 1]$: **steering angle** to be applied to the vehicle.
- $brk \in Brk = [0, 1]$: **brake** magnitude to be applied to the vehicle.
- $thr \in Thr = [0, 1]$: **throttle** magnitude to be applied to the vehicle.

The reward is also fixed for both experiments, such that the task goal is learning to drive straight at a moderate speed. This corresponds to a reward function of the form:

$$r(a_t) = \frac{1}{2}[(1 - a_t[0]) + (1 - \sqrt{a_t[1]^2}) - (0.25 - a_t[2])^2] \tag{8}$$

Note that this particular reward function is state-agnostic (for compatibility with Fake CARLA), and does not depend on the current time-step either. The optimal action under this reward would be:

$$a^* = [str = 0, brk = 0, thr = 0.25] \tag{9}$$

4.1 Fake CARLA Methods Comparison

The Fake CARLA consists of training our agents, using the methods described above, with randomly generated states $s_i \in S$. The intuition is that this would still allow us to test if our approaches converge to outputting the maximally-rewarding action, regardless of the input. While this seems trivial, it will be demonstrated in Section 5.1 that even this takes some time. While the real task, training with images from a simulated dashboard camera in CARLA, is much more difficult, the relative performance of our methods on the real task ought to be the same as their performance on the Fake CARLA task. However, since the models are not trained on actual data, they cannot be tested on task performance. Therefore we compare the methods for their average reward received once converged, as well as their rate of convergence.

[2] Computing resources dedicated for this project were damaged, the mechanics of our resource allocation were changed mid-semester, and of course the global COVID-19 pandemic all restricted our timeline

The MDP for the Fake CARLA experiments is defined as follows:

- set of states $s \in S := [0,1]^{200\times88\times33}$
- set of actions $a \in A := (Str \times Thr \times Brk)$
- state transition dynamics $\mathcal{C} := p(s_{t+1}|s_t, a_t)$ (randomly generated states s_{t+1})
- initial state distribution $\mathcal{C}_0 := p(s_0)$ (randomly generated states s_0)
- reward function $r =: S \to [-\frac{9}{16}, 1]$
- policy $\pi : S \to A$ (convolutional neural network architecture presented in [4])

4.2 Real CARLA RL Agent

For the Real CARLA task, we found that the convolutional neural network (CNN) architecture used in [4] and our Fake CARLA experiments resulted in a training procedure that was far too slow. We assume this is because the state-space induced by training on 3-channel images is massive ($3 \times 88 \times 200 = 52,802$), requiring many more samples over slower update steps to arrive at reasonable solutions. To prove that this result is not caused by a flaw in our methods, we tested our baseline PPO method on a much smaller multi-layer perceptron (MLP) policy, a neural network which uses only feedforward and activation layers. The resulting architecture had a fraction of the number of parameters, which we hoped would allow for a policy to be learned much more quickly. Unfortunately, this meant that our policy could no longer take images as input, meaning it would be quite limited in its navigational abilities.

This is why we defined the reward function as above, such that the agent could still learn some driving behaviour without needing visual input. Instead of images from a dashboard camera, we provided this agent with simply 3 values: velocity in the x and y directions, and rotation about z, or yaw, corresponding to the vehicle's heading.

The MDP for the Real CARLA experiment is defined as follows:

- set of states $s \in S := \mathbb{R}^3$
- set of actions $a \in A := (Str \times Thr \times Brk)$
- state transition dynamics $\mathcal{C} := p(s_{t+1}|s_t, a_t)$ (from CARLA)
- initial state distribution $\mathcal{C}_0 := p(s_0)$ (uniform over CARLA spawn-points)
- reward function $r =: S \to [0, 1]$
- policy $\pi : S \to A$ (4-layer MLP neural network architecture)

In order to train an IL agent, we need access to expert demonstrations. For the Fake CARLA task, we approximate these with samples from a multi-variate normal distribution representing the expert policy. This distribution outputs the optimal action for our reward function with a small degree of noise (mean action a^*, diagonal co-variance of 0.05).

5 Results

5.1 Fake CARLA

The average reward per simulator time-step for the baseline RL method and IBRL are shown in Figure 5.1. For both of these experiments we used a batch size of 1000, held the training hyperparameters fixed (learning rate, optimizer, etc.), and ran our procedure for 150 epochs ($K = 150$). The mean predicted action at completion of the two training procedures is shown here:

	str	brk	thr
Base	0.07	0.31	0.52
IBRL	0.003	0.24	0.37

[3] We use these dimensions for our images, taken from RGB camera sensors inside CARLA, following [6]

This experiment confirms our belief that injecting prior expert knowledge into an RL pipeline improves performance. The IBRL method is more sample efficient and produces better policies than the baseline RL method, as shown by Figure 5.1.

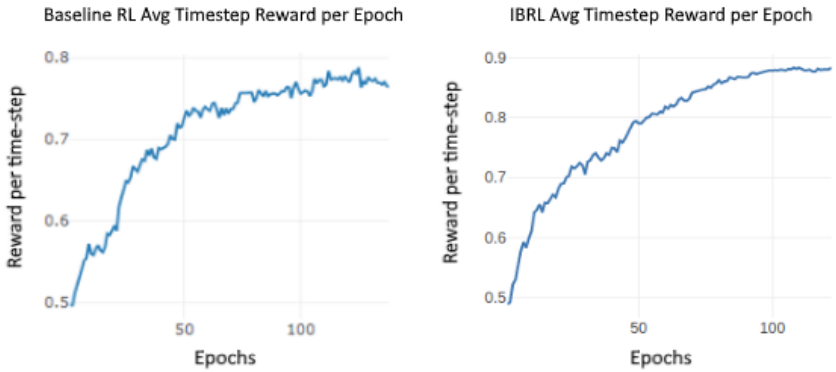

Figure 2: The two learning curves for the baseline (left) and IBRL (right) methods.

5.2 Real CARLA

The reward per simulator time-step, averaged over epochs, is shown here in Figure 3. The batch size used was 1000, so this policy saw 100,000 states sampled from the simulator throughout our training procedure. Due to time constraints, we limited our Real CARLA runs to 100 epochs, of which this was the best one. Additionally, it only managed to converge to a mean predicted action of $a = [0.28, \pm 0.003, 0.42]$, achieving an average reward of approximately 0.84 per frame.

Even though we used PPO, designed a comparatively tiny policy network, and chose a reward function that was completely agnostic of input, this baseline RL procedure took over 36 hours to run on a relatively powerful machine, and only achieved under 90% of the maximum possible reward. While we expected the policy to very quickly converge to a deterministic function outputting a^* for all states, in practice this was not quite so straightforward. There was still a decent amount of hyperparameter tuning to be done, and the resulting procedure required a surprising number of samples.

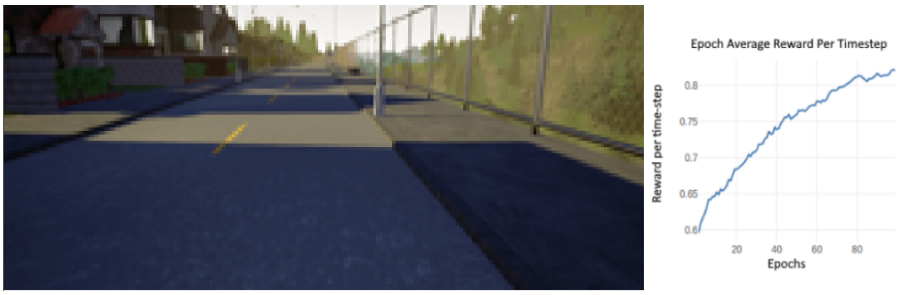

Figure 3: On the left: an instance of a frame from a CARLA RGB sensor, representing s_t in the Real CARLA task. On the right: the reward averaged over time-steps t for a single update step (epoch) in our procedure.

6 Discussion

6.1 Fake Carla

The results on the Fake CARLA task are unsurprising: adding a bonus to the reward term corresponding to behavioural similarity to an expert leads to better task performance. While this IBRL procedure does require a separate, pre-trained IL model , this can be seen as amortization, such that the cost of training an IL agent to guide the RL agent remains less than the cost of training a baseline RL agent to conversion. In our particular case, there was no data collection necessary, since we trained on randomly generated data. However, the increase in signal available to the RL agent during training is likely to be quite valuable, and depending on the task it may be quite beneficial to incur this overhead cost.

One limitation of our IBRL method was the difficulty of selecting a value for λ. The training procedure is quite sensitive to this parameter, as it shapes the reward surface to an extent, meaning some level of tuning is often necessary. In practice, we found most of our runs to be relatively consistent across similar values of λ (we use 0.05 for the procedure shown in 5.1). However, it's quite likely that on a more complicated task, λ plays a larger role in agent success.

We were unable to experiment with our RRI methods, given that there was little room for improvements to be made to the generated IL agent. However, we envision one potential limitation with this method being similar to issues with the λ parameter. Namely, how do we control how far away the RL procedure can move away from the IL policy? The use of PPO alleviates this to some extent, but PPO only limits the distance covered by a single update step, rather than the entire pipeline. For this reason, we would recommend experimenting freezing parts of the neural network policy being learned, such that only a subset of the weights are changed by the RL procedure. This would guarantee the agent never 'forgets' the entire IL policy, always maintaining some of that prior knowledge in its own current parameters.

6.2 Real CARLA

Given that the original task formulation, with visual input and a CNN architecture, has a state space that is roughly $17,000$ times larger than this MLP formulation, this experiment demonstrates the vast timescales and computational resources that would be required to implement the Real CARLA task with a more realistic notion of state. We were somewhat validated by the fact that even this drastically simplified problem formulation was quite difficult to train. This procedure took 36 hours to learn a single action over all states, leading us to conclude it would take weeks, if not months, to learn a policy on the full task with our current pipeline and hardware.

This formulation of the autonomous navigation task is clearly a far cry from self-driving cars. This agent received no inputs that would have allowed for obstacle avoidance or even navigation, and was only capable of driving on perfectly straight roads. In spite of this, there are valuable insights to be gained from this experiment. It outlines the difficulty of relying on purely visual input, and highlights the ease with which agents tend to train with more explicitly relevant inputs. This experiment could be extended, teaching the driving agent one simple driving behaviour at a time while gradually giving it more complicated inputs and state-dependent rewards. For example, we could reward our driving agent based on its distance from the centre-lane line, allowing it to learn to stay in-lane without ever seeing the lane. The problem with such approaches is that the real-world driving agent would also need access to this exact same set of inputs, which may not be as feasible outside a simulator.

7 Conclusion

We present a structured approach for including prior expert knowledge into reinforcement learning pipelines. In seeking to address issues of sample efficiency and convergence rates in RL, we demonstrate the ease with which methods can be modified slightly to produce better results. We demonstrated how PPO, an already lightweight and easy to implement objective, can be further improved through reward shaping (IBRL) and simple pretraining (RRI). These changes required no algorithmic modifications, and can effectively be used in almost any RL formulation. Applied to autonomous navigation, a domain where human experts are plentiful, we believe these methods would address some of the biggest challenges in this task.

8 Acknowledgements

Many thanks to Frank Wood and Adam Ścibor for their guidance and supervision in this project. Also a sincere thank you to my fellow PLAI collaborators Onur Tuna, Wilder Lavington, and Daniele Reda for their hours of hard work, countless patient conversations, and continued support and encouragement.

References

[1] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*, 50(2):1–35, 2017.

[2] Takayuki Osa, Joni Pajarinen, Gerhard Neumann, J Andrew Bagnell, Pieter Abbeel, Jan Peters, et al. An algorithmic perspective on imitation learning. *Foundations and Trends® in Robotics*, 7(1-2):1–179, 2018.

[3] Felipe Codevilla, Eder Santana, Antonio M López, and Adrien Gaidon. Exploring the limitations of behavior cloning for autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 9329–9338, 2019.

[4] Felipe Codevilla, Matthias Miiller, Antonio López, Vladlen Koltun, and Alexey Dosovitskiy. End-to-end driving via conditional imitation learning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–9. IEEE, 2018.

[5] Lei Tai Peng Yun, Yuying Chen, Congcong Liu, and Haoyang Ye Ming Liu. Visual-based autonomous driving deployment from a stochastic and uncertainty-aware perspective.

[6] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. *arXiv preprint arXiv:1711.03938*, 2017.

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

[8] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[9] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[10] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897, 2015.