Sequential Decision Making in CarRacing Game using Proximal Policy Optimization and Dataset Aggregation

Lingzhi Kong
Northeastern University, MA, USA
kong.ling@husky.neu.edu

Shaoshu Xu
Northeastern University, MA, USA
xu.shaos@husky.neu.edu

Abstract

The problem of sequential decision making in autonomous driving has drawn much attention both in industry and academia in recent years due to its importance in the success of next generation of autonomous driving cars. In this project, we implement Proximal Policy Optimization (PPO) and Dataset Aggregation (DAGGER) algorithms and apply them to the OpenAI gym CarRacing-v0 environment. PPO has been shown to be successful in solving continuous control problems in autonomous driving. DAGGER trains the policy iteratively and can give strong performance guarantees both in practice and in theory. We show that DAGGER is able to solve the CarRacing game using expert demonstrations from traditional methods. We also show that PPO is an effective way to address the sequential decision making problem in CarRacing game.

Introduction

With the advent of autonomous driving technology, consumers all around the world are eager to see such innovative technology come to life. Autonomous driving, or self-driving, has been said to be the next ground-breaking innovation in the years to come. It is supposed to have massive societal impact in all kinds of fields [1].

First, transportation accident are one of the major causes of death in the world due to dangerous driving behaviors, such as drunk driving, impaired driving, and speeding. Accidents caused by these illegal behaviors can be reduced significantly by autonomous driving technology on the ground that autonomous cars are rule-following and do not conduct illegal behaviors. Second, autonomous driving also has a huge impact on the economy of the world. Autonomous cars can reduce the cost of crashes, medical bills, vehicle repairs, and traffic fines. Third, autonomous cars are more efficient than traditional human-drive ones because they can be more easily managed by a systematic and efficient method [2]. If all the cars were managed by an autonomous and intelligent system, then this practice would reduce traffic congestion and thus increase efficiency. Plus, parking scarcity would become a historic phenomenon since autonomous cars could drop off passengers and park at any suitable space and return back to pick the passengers up.

This project aims at dealing with decision making problems in autonomous driving by applying deep reinforcement learning and behavioral cloning algorithm in a simulated car racing environment. To be specific, we will implement deep reinforcement learning algorithms (Proximal Policy Optimization) and behavioral cloning algorithm (DAGGER) and use them to teach an agent car to play the CarRacing game.

As a deep reinforcement learning algorithm, Proximal Policy Optimization directly optimizes the cumulative reward without using a value function, and it is able to be trained directly with nonlinear function approximations such as neural networks. Dataset Aggregation is an efficient way to reproduce the expert demonstrated behavior. Expert data will first be obtained by some well-trained expert. Then, the state and actions made by the expert will be recorded at the same time. Our goal for behavioral cloning is to teach an agent to play the game using expert demonstrated data.

Background

Researchers have made several major breakthroughs in recent years. However, issues regarding the decision making for autonomous driving remain open. Related works in this area can be categorized as traditional approach and end-to-end reinforcement learning approach.

In previous works [3][4], autonomous vehicle systems and algorithms have been designed to navigate safely using advanced sensing and control algorithms [5][6][7]. The decision-making process in these systems traditionally integrates many specific independently engineered components, such as perception, state estimation, mapping, planning and control [8][9][12]. However, this can be difficult to scale to an unstructured real-world environment.

Fortunately, in recent years, there have been many breakthroughs in end-to-end learning approaches. Some algorithms have been proposed to train an agent learning a decision-making policy directly from high-dimensional sensory inputs, that is, mapping what we see directly to what we do. We don't need to deal with the complex interdependency between different algorithms and components. These algorithms optimize all parameters of a model jointly with re-

spect to an end goal, thus reducing the effort of tuning each component. Wayve's [10] work has demonstrated that this approach is possible on rural country roads using GPS for course localization and LIDAR to understand local scenes. Here we will show that end-to-end learning approaches, like PPO and DAGGER, could be a feasible approach for the decision making process in autonomous driving.

Proximal Policy Optimization (PPO)

Proposed by John Schulman etc. in 2017 [13], Proximal Policy Optimization becomes a new family of Policy Gradient methods. It directly optimizes the cumulative reward without using a value function, and it is able to be trained directly with nonlinear function approximations such as neural networks. This is useful when the action space is continuous or when the policy is stochastic due to the fact that value-based approaches are too expensive computationally in the continuous space. PPO optimizes an objective function using stochastic gradient ascent by alternating between sampling data through interaction with the environment. While standard policy gradient methods perform one gradient update per data sample, PPO uses multiple epochs of mini-batch updates. Before we step into the details of Proximal Policy Optimization method, let's first view some basics about policy gradient method.

The policy gradient methods aim at modeling and optimizing the policy directly. The policy is usually modeled with a parameterized function $\pi_\theta(a|s)$ with respect to θ. REINFORCE policy gradient relies on an estimated return by Monte Carlo methods using episode samples to update the policy parameter θ. The advantage function $A(s,a) = Q(s,a) - V(s)$ was introduced to reduce the variance of gradient estimation while keeping the bias unchanged [15]. Then, Actor-Critic method [16] was proposed by combining value function and policy model.

- Critic: updates the value function parameters w and, depending on the algorithm, it could be action-value $Q(s,a)$ or state-value $V(s)$.
- Actor: updates the policy parameters θ for $\pi_\theta(a|s)$, to the direction suggested by the critic.

Both REINFORCE policy gradient and the vanilla version of Actor-Critic method are on-policy. Off-policy methods have several additional advantages, such as not requiring full trajectories and being able to reuse any past episodes for much better sample efficiency. Here the importance sampling weight $\frac{\pi(a|s)}{\beta(a|s)}$ was used to collect some exploratory data from one policy q and evaluate on different policy p [14]. However, these methods are still unstable and unreliable because the policy would be changed too much at one step parameter updating. To address this issue, Proximal Policy Optimization method was proposed which provides a better performance due to the reason that it introduce the clip function to limit the distance of the two policies in importance sampling term.

Dataset Aggregation

Dataset Aggregation can be seen as a kind of behavioral cloning. The key idea of behavioral cloning is that we are able to train our model using the expert demonstrations, which provide the perfect sequence of decisions. In self-driving applications with imitation learning setting, we assume that we have access to an expert that already knows how to drive. We want to imitate the expert policy given the demonstration data.

For simplest version of behavioral cloning, the learning problem can be formulated as a supervised learning problem in which decision-making policy can be obtained [13]. Behavioral cloning uses neural networks to learn the mapping function between input states and target actions. In practice, we should first generate the expert data using expert demonstrations. Second, we use the expert data to learn how to choose actions or make decisions when facing new states. However, this method can be challenging when demonstration data cannot cover all the states. A small error in an unvisited state may lead to further mistakes, making the car deviate from expert demonstrations. To address this issue, Dagger, an iterative learning method, has been proposed. The basic idea of Dagger is to learn a policy iteratively, that is, it requests additional demonstrations to expand the dataset and re-trains the policy using the new dataset until we get the desired policy.

Environment & Data

The project will be implemented using OpenAI CarRacing-v0 gym environment, a top-down car racing environment. The training data is obtained by interacting with the simulator and the training process is completed using OpenAI gym environment interface.

The details of our environment is described below [14]:

- State: 96*96*3 pixel RGB color screen image;
- Action: the action space is the set of triples (steering, gas, braking) $\in [-1,1],[0,1],[0,1]$, where the steering coefficient ranges from hard left to hard right, the acceleration ranges from none to full steam ahead, and the deceleration ranges from none to slamming on the brakes. In the human version of the game (controlled with arrow keys) the actions are discretized, with the left arrow indicating steering = -1, down indicating = +1, etc.
- Reward: -0.1 every frame and +1000/N for every track tile visited, where N is the total number of tiles in track. For the Proximal Policy Optimization algorithm, a -0.5 penalty will also be applied to every frame when the car runs on the grassland area.

Note that each episode finishes when all tiles have been visited, and some specific changes of the environment will be mentioned in part 4.

Algorithm

Proximal Policy Optimization (PPO)

There are two primary variants of PPO: PPO-Penalty and PPO-Clip. PPO-Penalty approximately solves a KL-constrained update. There is no KL-divergence term in the objective function of PPO-Clip and no constraint at all. Instead, it relies on specialized clipping in the objective func-

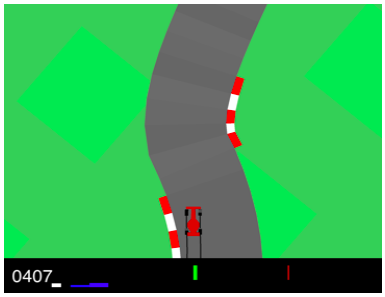

Figure 1: OpenAI CarRacing-v0 Gym environment

tion to remove tendency for the new policy to get far from the old policy. In this project, we'll focus only on PPO-Clip.

The central idea of clipped version Proximal Policy Optimization (PPO-Clip) is to avoid having too large policy updates by introducing the clipped surrogate objective function proposed by John Schulman [13]. As a new family of policy gradient methods for reinforcement learning, we optimize this new object function using stochastic gradient ascent through a convolutional neural network (CNN) architecture. We will introduce the details of these two mechanism separately in the following.

1) Clipped Surrogate Objective Function:

First, we denote two policies: $\pi_\theta(a|s)$ is the current policy we want to refined; $\pi_{\theta old}(a|s)$ is the policy we last used to collect samples. With the idea of importance sampling [15], we define the ratio between old and new policies as:

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta old}(a|s)}$$

The ratio measures how different between these two policies.

However, without a limitation on the distance between θ_{old} and θ, there would be instability with extremely large parameter updates and big policy ratios $r(\theta)$. By using clip function, PPO imposes the constraint to force $r(\theta)$ to stay within a small interval around 1 to ensure the new policy to be not too far from the old one. Here ϵ is a hyper-parameter that stand for how far away the new policy is allowed to go from the old one, and the interval is precisely [1-ϵ, 1+ϵ]. The new objective function proposed is: [13]

$$J^{CLIP}(\theta) = E[min(r(\theta)\hat{A}_{\theta old}(s,a), clip(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_{\theta old}(s,a))]$$

If the probability ratio between the new policy and the old policy falls outside the range $(1-\epsilon)$ and $(1+\epsilon)$, the advantage function $\hat{A}_{\theta old}(s,a)$ will be clipped by $clip(r(\theta), 1-\epsilon, 1+\epsilon)$. After that, we take the minimum between the unclipped and clipped objective function, so the final objective is obtained by taking the minimum one between the original value and the clipped version.

Figure 2 shows that the new objective function is a lower bound on the unclipped objective for positive and negative advantage functions. The blue line is the clip function; green line is the unclipped objective; red line shows the lower bound for a positive advantage function $A(s,a)$.

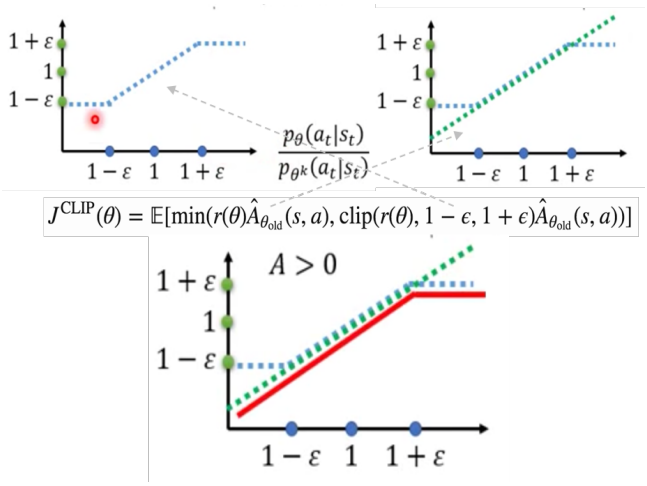

Figure 2: Demonstration of the clipped advantage function

2) Convolutional Neural Network Architecture for Updating Policy:

PPO-clip updates policies by the following eauqtion:

$$\theta_{k+1} = argmax_\theta E_{s,a\ \pi_{\theta_k}}[L(s,a,\theta_k,\theta)]$$

where $L(s,a,\theta_k,\theta)$ is the objective function $J^{CLIP}(\theta)$ shown before.

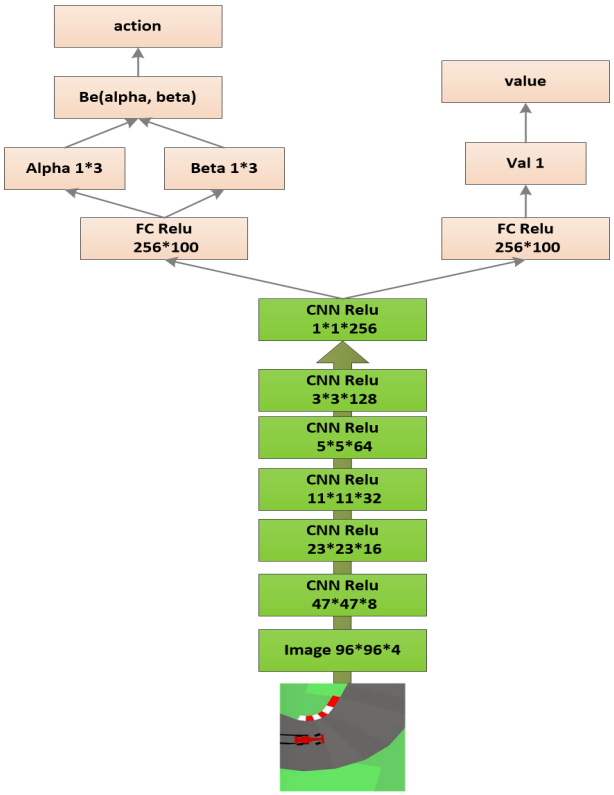

Figure 3: Convolutional Neural Network Architecture for Updating Policy

By constructing a convolutional neural network, we optimize the objective function using stochastic gradient ascent with Adam optimization. Here we implement multiple epochs of mini-batch updates during each iteration, whereas the standard policy gradient methods perform only one gradient update per data sample.

The architecture of our network is shown in figure 3. The input is adjacent 4 frames gray images in the shape of (4, 96, 96), standing for the state of the agent. This two-head network outputs the estimate state value as the critic and the sample action as the actor. One of the head network outputs a set of two parameters, Alpha and Beta, which are used to sample action from the Beta distribution and make sure the sample is between [0, 1]. The other head outputs the state-value $V(s)$, which is used in advantage function.

The pseudocode of PPO with clipped objective function is shown in Figure 4.

Algorithm 5 PPO with Clipped Objective

Input: initial policy parameters θ_0, clipping threshold ϵ
for $k = 0, 1, 2, \ldots$ **do**
  Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
  Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
  Compute policy update

$$\theta_{k+1} = \arg\max_{\theta} \mathcal{L}_{\theta_k}^{CLIP}(\theta)$$

  by taking K steps of minibatch SGD (via Adam), where

$$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \mathop{\mathrm{E}}_{\tau \sim \pi_k}\left[\sum_{t=0}^{T}\left[\min(r_t(\theta)\hat{A}_t^{\pi_k}, \mathrm{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t^{\pi_k})\right]\right]$$

end for

Figure 4: Pseudocode of PPO [17]

DAGGER

Dagger, also called Dataset Aggregation, is a kind of Behavioral cloning algorithm. Behavioral cloning uses neural networks to learn a policy (mapping function) between states (screen images) and actions. Figure 5 shows the overall process of the simplest behavioral cloning. The ingredients of behavioral cloning include demonstrator, simulator, policy neural network, loss function and learning algorithms. [18]

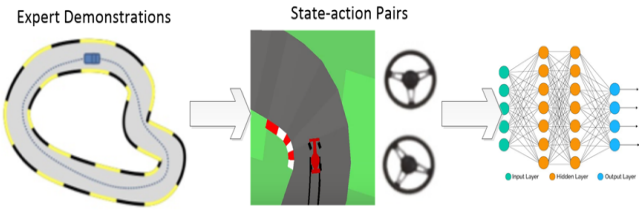

Figure 5: Scheme of Behavioral cloning; left: expert demonstrations; middle: state as input, action as label; right: policy network

We first introduce notation and setup:

- State: s = game screen (image)
- Action: $a = (steer, gas, brake) \in [-1, 1], [0, 1], [0, 1]$
- Policy: π Policy maps states to actions $\pi(s) = a$ or distributions over actions: $\pi(s) = P(a|s)$
- Rollout: sequentially execute $\pi(S_0)$ on initial state Produce trajectory τ(S1, A1, R1, S2, A2, R2 .....)
- $P(t|\pi)$: Distribution of trajectories induced by a policy.

Our learning objective in the simplest behavioral cloning setting is:

$$argmin_\theta E_{(s,a^*)\ P^*} L(a^*, \pi_\theta(S))$$

which can be seen as a supervised learning problem.

However, there are limitations in this method. Expert only samples limited observations or states, and the policy will make big mistakes when the agent goes to a state that has not been encountered before. Furthermore, a small error e made by the policy network can lead to as many as $T * T * e$ mistakes after T steps [19]. Intuitively, that is because when the agent makes a small mistake, it may go to different states that have not been encountered before, which causes even more errors when making decisions based on the new states [20].

To address this issue, we applied Dagger algorithms to the car racing game environment. Dagger uses expert data to train a policy and uses the policy to further interact with the environment (simulator), expanding the expert dataset, which in turn can be used to train a new policy. We iterate this process until we get the desired policy. Figure 6 shows the details of general Dagger algorithm [20].

Initialize $\mathcal{D} \leftarrow \emptyset$.
Initialize $\hat{\pi}_1$ to any policy in Π.
for $i = 1$ **to** N **do**
  Let $\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_i$.
  Sample T-step trajectories using π_i.
  Get dataset $\mathcal{D}_i = \{(s, \pi^*(s))\}$ of visited states by π_i and actions given by expert.
  Aggregate datasets: $\mathcal{D} \leftarrow \mathcal{D} \bigcup \mathcal{D}_i$.
  Train classifier $\hat{\pi}_{i+1}$ on $\mathcal{D}$.
end for
Return best $\hat{\pi}_i$ on validation.

Figure 6: Pseudocode of DAGGER

To be specific, Dagger proceeds by collecting a dataset under current policy, and aggregates it into all collected dataset. In the next iteration, we train a new policy using the new aggregated dataset. Then, we repeat this process until we get the desired policy.

The loss function we used is cross-entropy loss:

$$L_{cross-entropy} = -\sum_{c=1}^{M} y_c \log(p_c)$$

Experiments and Results

Proximal Policy Optimization (PPO)

(a) Experimental Details

To demonstrate the performance of PPO algorithm, we trained and evaluated it in the CarRacing-v0 gym environment. Our goal was to train the agent by adjusting the value

of steering, gas and braking in order to drive as far as possible on the track. The hyper-parameters for neural network and policy gradient process were: learning rate = 1e-3, 10 epochs for every weight updating, the clip parameter $\epsilon = 0.1$.

The whole training process was implemented on 2016 MacBook Pro with 8 core Intel i7 Processor and 16GB ram, finally terminating at 2120th episode with time spending about 10 hours.

(b) All the results and findings

During the 2000-episode training process, we saved several models and evaluated the agent's policy by observing how well the agent performed in the gym environment. The score that the agent obtained from each episode was also been plotted to analyze the whole training process.

Here are three set of screenshots showing the agents policy at episode 90, 1500 and 2060, respectively.

As shown in figure 7, at the beginning of the training process, episode = 90, the agent had no idea on how to drive on the track. It drove slowly and with random steering (the green bar at the bottom shows the steering angle).

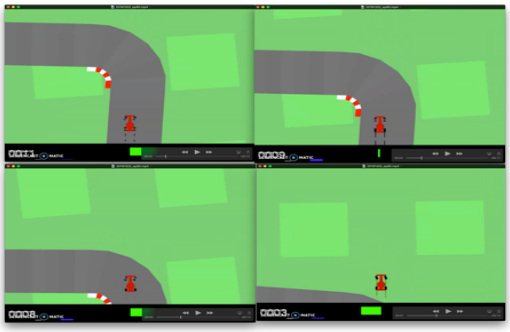

Figure 7: Demo of episode = 90

After training around the 1400 episodes, we could observe that the agent had gained some knowledge about how to take actions. It is able to pass some of the simple turns with a relatively appropriate value of steering, gas and braking. But for the U-turn or S-turn, it sometimes still took the wrong actions and went out of the track. Also, there were times that the car cut straight across the turn from the green area, as shown in figure 8 and 9. More training needs to be done.

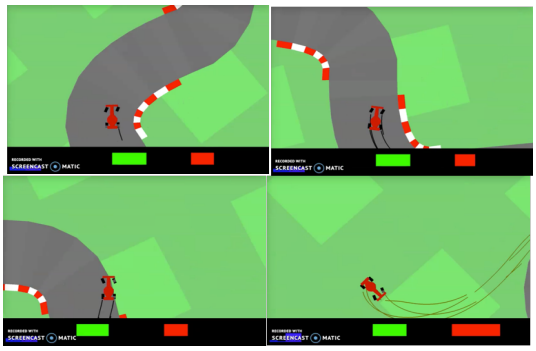

Figure 8: Demo 1 of episode = 1400

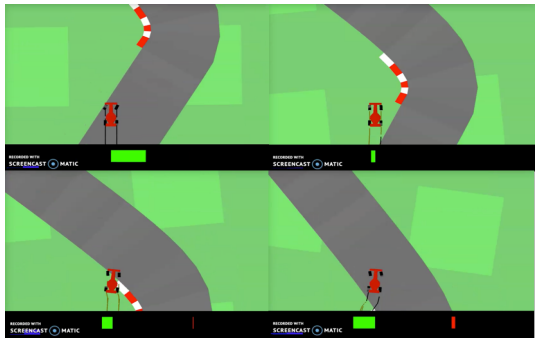

Figure 9: Demo 2 of episode = 1400

After training 2000 episodes, the agent drove faster and more fluently as before. It is exciting to see that the agent already learnt to pass most of the turns with a relatively high precision and efficiency, including Right-angle turn, U-turns and Combined-turns. As shown in figure 10.

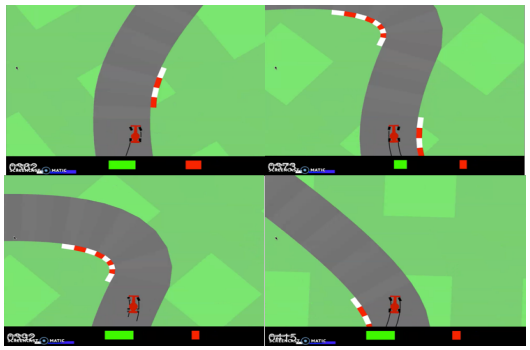

Figure 10: Demo of episode = 2000

However, mistakes are sometimes inevitable. As shown in figure 11, we can see that the agent turned too fast with a serious skidding and almost rushed out of the track, even though it got back on the track by adjusting the steering and braking.

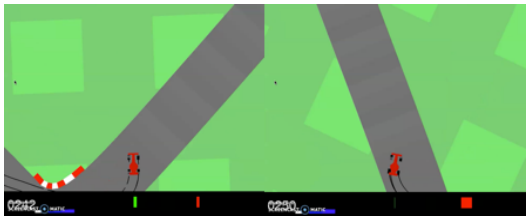

Figure 11: Example of turns too fast with a serious skidding, episode = 1400

In the Figure 12, we can observe that the car skids a whole circle in the U-turn, because it enters the U-turn in a very high speed without enough braking. That might be because the policy we learnt was stochastic and the agent might choose inappropriate actions. A more stable driving policy should be expected, like precise coordinate between each actions.

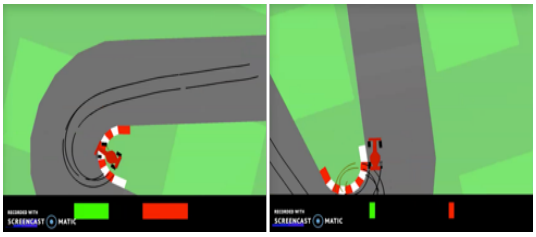

Figure 12: Example of turns too fast with a serious skidding, episode = 1400

Figure 13 shows the score for each episode and moving average score of the whole training processing, which is defined as follows:

- Score for each episode: $score = score + reward$
- Moving average score:
  $runningscore = runningscore * 0.99 + score * 0.01$

At the beginning 800 episodes, there was no obvious improvement on the policy. The policy got significant improvements between around 800th to 1800th episodes. We can also notice that some extremely small scores were obtained at several episodes around 1400th episode, which also has been confirmed by the result of Figure 13. After 2000 training episodes, the scores converged and there was no further change of the policy.

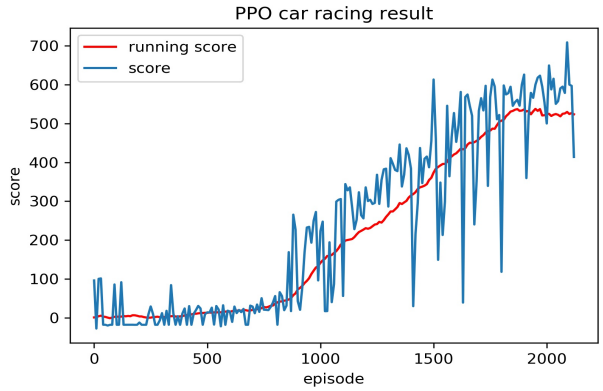

Figure 13: Plots of training score and moving average score

DAGGER

The experiment was carried out under the Ubuntu 16.04 operating system and Thinkpad T480s PC with 16 GB RAM and Nvidia GPU. The whole experiment was implemented with Python. The libraries we used include Pytorch, numpy, openAI gym.

(a) Experimental details

We first generated the expert data using traditional methods. In particular, once an image of the game scene is captured, the track and car from the green land of the image were extracted, and then the car positions and angel with respect to the track were calculated. Next, the traditional PID control method was used to calculate the actions that needed to be taken under that game scene. We stored the actions as the file name of the image so that we can get the actions as a label during the training process.

For the training process, we built a neural network. The structure of the neural network is summarized in Figure 14. We built the convolutional neural network (CNN) connected with fully connected layers.

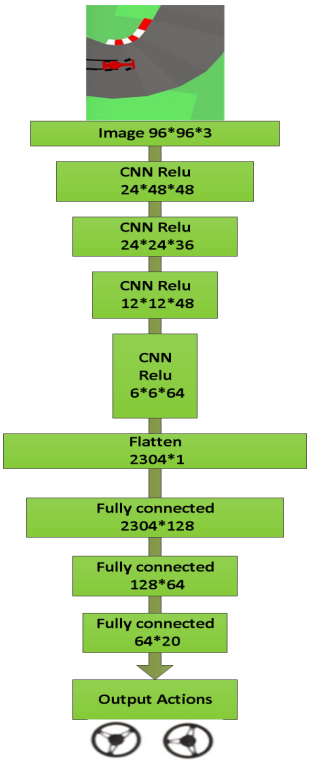

Figure 14: Neural network structure of DAGGER

Now that we have built the neural network, we used the expert data to learn a policy. In particular, we take images as input and actions as label. Then we could formulate it as a supervised learning problem and train an initial policy to mimic demonstrations. The initial policy is trained to regress to the desired actions by minimizing the cross-entropy loss. The parameters of the training process are:

- Learning rate: 0.001
- Batch size: 256
- Workers: 8
- Epochs: 50

The network was trained using the optimizer Adam, which is similar to Stochastic Gradient Descent (SGD). Since the majority of the game scenes were straight track, the expert demonstration data were mostly driving straight forward, leading to a significant label imbalance. Observing this situation, we took the imbalance into account during training. In particular, we calculated the frequency of occurrence of each class label. Then, we used a weighted

loss that weights errors in each class label according to the inverse frequency of their occurrence. In Pytorch, cross entropy function enabled us to calculate the weighted loss.

Now that we had the initial policy, we expanded the dataset by means of interacting with the environment using the initial policy. Then we trained a new policy using the new aggregated dataset. In each iteration, the dataset would get bigger than previous iteration, and the weight of each iteration was saved so that we could load the weight interacting with the simulator and see the performance of the policy trained in each iteration.

(b) All the results and findings

We trained a total of 6 iterations. At 6th iteration, the agent performed reasonably well. The following plots (Figure 15) show the returns of each iteration.

Figure 15: DAGGER CarRacing returns for iteration 0,1,3

Figure 16 shows the average return of each iteration. The average return increases as the training iteration goes on, indicating that the performance of the policy becomes better. For details about the agent interacting with the simulator under the policy in different iteration, please see the attached videos.

Figure 16: Mean returns vs. Iterations

Conclusion & Future Work

PPO

For Proximal Policy Optimization, we made some modification on the original CarRacing Gym environment, by combining adjacent 4 frames as a single input state and repeating every action for 8 frames.

In training part, we built a two-head neural network, one outputs the action sampling from Beta distribution and the other outputs the state value directly for computing the advantage function. A loss function that combined the policy objective and the value function error term was also been used to update the weights. The network was trained using stochastic gradient ascent with Adam optimization. The hyper-parameters were fine tuned during the training process.

The preliminary results shown above are satisfactory but a better result was expected in the future. First, we consider extracting the gyroscope and ABS sensors value from CarRacing Gym environment to detect whether the car is skidding or not, and then try to solve the skidding problem by adding a punishment to the reward. Second, we will consider using TanhNormal distribution instead of Beta distribution to sample the action. Third, we notice that the score didn't increase any more after training 2000 episodes. We consider to use DDPG to obtain a more stable and smooth policy.

DAGGER

For imitation learning, we used DAGGER algorithm. The expert data was obtained by traditional method in self-driving. We recorded states (96*96*3 pixel images) and actions (a sequence of actions leading by the expert run). The neural network used in DAGGER was built through Pytorch library. The network took state images as input and actions as label, and trained an initial policy. Finally, we applied Dagger to CarRacing game environment and carried out simulation experiments to evaluate the performance of the policy trained in each iteration. The result shows that Dagger is able to clone expert demonstration and learn a new policy.

In future work, we will consider modifying the network structure speeding up the training process. We will also consider applying Data as Demonstrator algorithm to car racing game, which extends Dagger to address the problem of multi-step prediction in imitation learning.

References

[1] Rosenzweig, Juan, and Michael Bartl. "A review and analysis of literature on autonomous driving." E-Journal Making-of Innovation (2015).

[2] Bimbraw, Keshav. "Autonomous cars: Past, present and future a review of the developments in the last century, the present scenario and the expected future of autonomous vehicle technology." 2015 12th International Conference on Informatics in Control, Automation and Robotics (ICINCO). Vol. 1. IEEE, 2015.

[3] T. Kanade, C. Thorpe, W. Whittaker, "Autonomous land vehicle project at CMU", Proceedings of the 1986

ACM fourteenth annual conference on Computer science, pp. 71-80, 1986.

[4] R. S. Wallace, A. Stentz, C. E. Thorpe, H. P. Moravec, W. Whittaker, T. Kanade, "First results in robot road-following", IJCAI. Citeseer, pp. 1089-1095, 1985.

[5] M. Montemerlo, J. Becker, S. Bhat, H. Dahlkamp, D. Dolgov, S. Et-tinger, D. Haehnel, T. Hilden, G. Hoffmann, B. Huhnke et al., "Junior: The stanford entry in the urban challenge", Journal of field Robotics, vol. 25, no. 9, pp. 569-597, 2008.

[6] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt et al., "Towards fully autonomous driving: Systems and algorithms", Intelligent Vehicles Symposium (IV) 2011 IEEE, pp. 163-168, 2011.

[7] U. Franke, D. Gavrila, S. Gorzig, F. Lindner, F. Puetzold, C. Wohler, "Autonomous driving goes downtown", IEEE Intelligent Systems and Their Applications, vol. 13, no. 6, pp. 40-48, 1998.

[8] S. Thrun, W. Burgard, D. Fox, Probabilistic robotics, MIT press, 2005.

[9] Fridman, Lex, et al. "MIT advanced vehicle technology study: Large-scale naturalistic driving study of driver behavior and interaction with automation." IEEE Access 7 (2019): 102021-102038.

[10] https://wayve.ai/blog/learning-to-drive-in-a-day-with-reinforcement-learning

[11] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529.

[12] Kendall, Alex, et al. "Learning to drive in a day." 2019 International Conference on Robotics and Automation (ICRA). IEEE, 2019.

[13] Schulman, John, et al. "Proximal policy optimization algorithms." arXiv preprint arXiv:1707.06347 (2017).

[14] https://gym.openai.com/envs/CarRacing-v0/

[15] Sutton, Richard S., et al. "Policy gradient methods for reinforcement learning with function approximation." Advances in neural information processing systems. 2000.

[16] Konda, V., Tsitsiklis, J. (2003). Actor-critic algorithms. SIAM Journal on Control and Optimization, 42(4), 1143–1166.

[17]http://rail.eecs.berkeley.edu/deeprlcourse-fa17/f17docs/lecture13advancedpg.pdf

[18] Imitation learning 2018 ICML Tutorial, Yisong Yue, et, al

[19] Ross, Stéphane, and Drew Bagnell. "Efficient reductions for imitation learning." Proceedings of the thirteenth international conference on artificial intelligence and statistics. 2010.

[20] Ross, Stéphane, Geoffrey Gordon, and Drew Bagnell. "A reduction of imitation learning and structured prediction to no-regret online learning." Proceedings of the fourteenth international conference on artificial intelligence and statistics. 2011.