Bootstrapped Ensemble RL in Car Racing

Spring 2022 CS 7649 Group 4 Project

Fei Ding
College of Computing
Georgia Institute of Technology
Atlanta, USA
fding33@gatech.edu

Hanfei Sun
College of Computing
Georgia Institute of Technology
Atlanta, USA
hsun315@gatech.edu

Liqiong Zhao
College of Computing
Georgia Institute of Technology
Atlanta, USA
lzhao336@gatech.edu

I Introduction

Car racing games are simulation games based on real-world auto racing competitions. In these games, cars (agents) compete in a racing course (environment) and try to score high based on some predefined criteria or winning conditions. Like many other types of games, this particular game genre has attracted the attention of many researchers. In particular, numerous methods have been proposed to design car agents which can excel in their respective environments and setups. However, the task of designing capable car agents is quite challenging, as there are many uncertainties in a racing game, like hitting obstacles, heading, wrong way, and driving off road, which all could happen in a short time frame given the high speed of racing cars. All the above situations are possible even with a single agent, so a multi-agent environment is destined to be more complex.

While it may be possible to solve the game using simple control schemes given car dynamics, another way to tackle this challenge in the field of robotics is to use reinforcement learning (RL) as human instincts cannot precisely define what the car should exactly do at each time frame. Instead, reinforcement learning only requires a reward signal to be assigned to the agent after each action, which provides a great starting ground for the agent to learn in an unsupervised fashion. There are multiple ways to define the state given a timestamp in the RL paradigm. Intuitively, we can use raw pixels as the features, for which the convolutions neural networks (CNN) can generalize to more a powerful lower-dimensional state representation. On the other hand, we can also use the car's own sensor data, which includes a vector of the car's stats including its speed and direction as well as the road conditions.

In our work, we will work with the OpenAI Gym environment [11], which is a specifically designed framework and environment for developing and testing reinforcement learning models. What's more, it includes a prefabricated racing simulator, which we could employ readily as a testing environment. In terms of methods, we would like to experiment with multiple setups for the RL agent, which is described in detail in the Methods and Evaluation section.

II Literature Review

Extensive research in many subareas of car racing has been done as the topic is closely tied to autonomous driving and provides many useful insights to this popular and challenging task. Work from the early days grasps and uses prior knowledge of the car dynamics [1], [2] and/or lower dimensional input tensors as from the game-theoretic models [3], [4]. Neither of the assumptions shall be made in our study because we do not assume the dynamics implementation of the OpenAI Gym's racing simulator. We will be either observing raw pixels from the game or use readily-available sensor readings of the car. To the same end, our version of the problem can also be modelled as a partially observable Markov decision process (POMDP) [8]: the vehicle agent must infer its own state solely based on partial observations at each game tick and take some action based on it. Moreover, adding more racing agents [7] will further complicate the situation by asking each agent to predict about the opponents. Instead of directly solving the multi-agent racing game, we are inspired to extend its idea and improve a single agent's policy.

[12] introduces a multi-agent racing environment as an extension to the original OpenAI Gym's simple racing environment, transforming the racing game to a multi-agent RL (MARL) problem. The MARL domain has shown to achieve superhuman results as in competition games like Go [5] and Poker [6]. The authors then come up with the Deep Latent Competition algorithm (DLC), essentially a variant of the policy gradient method, to learn "competitive visual control policies through self-play in imagination" and demonstrate its advantage in learning competitive behaviors in multi-agent environment while retaining the same level of racing skills in the single race case. The agent's "competition sense" comes from its inference of other agents' states and actions based on its own observation of image pixels in the game and impacts made by other agents' actual actions.

Using convolutional neural networks (CNN) [9] to encode raw pixels into lower-dimensional features is a widely

used technique in the machine learning field. Classification techniques as simple as applying CNNs to map from pixels directly to actions in the racing task are claimed to be very powerful [10]. In addition, Zhang and Sun propose data augmentation method to reduce the need for high-quality data from real players [13], but they only achieved good results after combining the features from actual speed and ABS sensor readings. We may not manually collect actual player's data or hand-crafted features but will stick to the idea of autonomous learning by RL algorithms; however, we may consider using sensor readings if they are helpful in our case. In fact, we also doubt the effectiveness of learning from raw pixels as it may be challenging (but still possible) to simultaneously learn good state representations and good policies. To accelerate training our RL agents, we may opt to use sensor readings only as it is possible to later translate the pixel input into sensor readings using CNNs and supervised learning methods, whose idea is similar to training a state encoder.

On the RL aogorithm side, alledgedly the proximity policy optimization (PPO) algorithm and its variant in the car racing environment has reached the SOTA result [15]. PPO alternates between sampling from the game environment by interacting with it and using stochastic gradient descent (SGD) to optimize a surrogate objective function. However, in our preliminary experimental benchmark runs as Figure 1 shows, another algorithm called deep deterministic policy gradient (DDPG) achieves the best practical performance in terms of average episode length/final score before crash. Meanwhile, Guo and Wu [14] also suggests DDPG with action punishment and multiple exploration, which deals with the noise in gradients in the early training phase and promotes exploration in latter training phase, respectively. We believe this is a very promising direction and would like to combine with our definition of states and adapt their technique to our racing game setup.

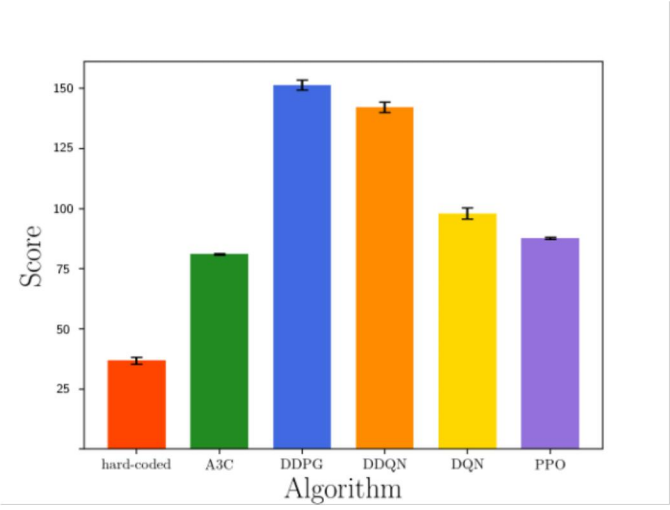

Fig. 1. RL Car Racing Algorithm Benchmarking.

For some backgrounds of DDPG, it is actor-critic style RL algorithm that operates in continuous state space and continuous action space, which are extremely suitable for the car racing environment because its controls are best described continuously. To understand DDPG, one must understand deterministic policy gradient (DPG) first [16]. DPG "maintains a parameterized actor function $\mu(s|\theta^{\mu})$ which specifies the current policy by deterministically mapping states to a specific action" Similar to how we extend from Q-learning to DQN in class, Lillicrap et al. [17] develops DDPG on the top of DPG, using neural networks as function approximators for the actors and critics and exponentially decaying Ornstein–Uhlenbeck (OU) action noises to encourage explorations against exploitations. The full algorithm is described in Algorithm 1.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^{\mu})$ with weights θ^Q and θ^{μ}.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^{\mu}$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^{\mu}) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^{\mu}} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^{\mu}} \mu(s|\theta^{\mu})|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

  end for
end for

A variant of DDPG called MADDPG extends its use to the multi-agent environment. In general, it uses multi-agent actor critic to infer the policies of other agents and performs a policy ensemble [18]. Another way to comprehend this approach is to imagine a group ensemble of DDPG agents whose critics consider the actions of all actors. The idea is also similar to Wu and Li's approach [19] except the latter uses a single-agent environment but aggregate the policies/actions at the end (see Figure 2).

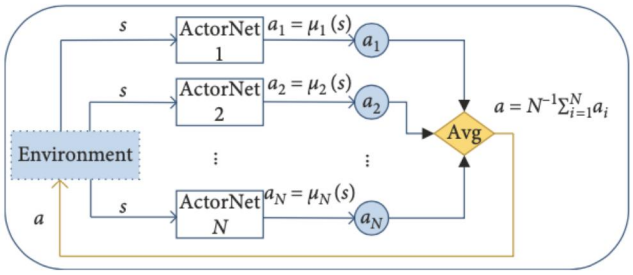

Fig. 2. Aggregation of Subpolicies.

We shall take advantage of both teams' ideas in the car racing environment: we would like to create a policy ensemble in the single-agent environment but let the critics benefit from inspecting all actors' actions.

III Methodology

III-A Baselines

For baseline models, we consider the DDPG algorithm with two types of state representations: pixel input or sensor data input. As stated in the literature review section, DDPG models have shown to excel among several others (see Figure 1).

CNN Baseline One baseline uses CNN encoded pixel data. For this model, we use a CNN to convert the raw pixels of the game screen (has dimension 96 by 96 by 3) into low-dimension features, as CNNs are the canonical method to extract features from image-type data.

Sensor Net Baseline Another baseline we created uses the car agent's sensor data as input. The intuition behind is that learning from pixels is challenging because the agent must learn a good representation (with convolutional layers) and a good policy (using actor-critic reinforcement learning) at the same time. This may fail frequently as the two learning goals happen at a different pace and phase. Instead, using sensor data is both practical in the real life and effective in terms of learning: they are easy to collect as game state is readily available in the simulator; using them as features results in much fewer parameters in the model which translates to a faster training process and greater chance to converge. Our experimental runs how that the CNN baseline converges with roughly 2000 episodes while the sensor-based data takes only about 400. The differernce is more prominent if we factor in the extra time of computations in the convolutional layers.

For this baseline, we use 7 sensor data. They are

s_1. car_angle[1],
s_2. angle_difference[2],
s_3. is_driving_on_grass,
s_4. is_driving_backward,
s_5. distance_to_left,
s_6. distance_to_right and
s_7. speed.

See an illustration and explanation of these sensor data in Figure 3.

[1]indicates global direction e.g. 0 if car is facing north
[2]indicates the angle between the direction of the road and car's facing; signed

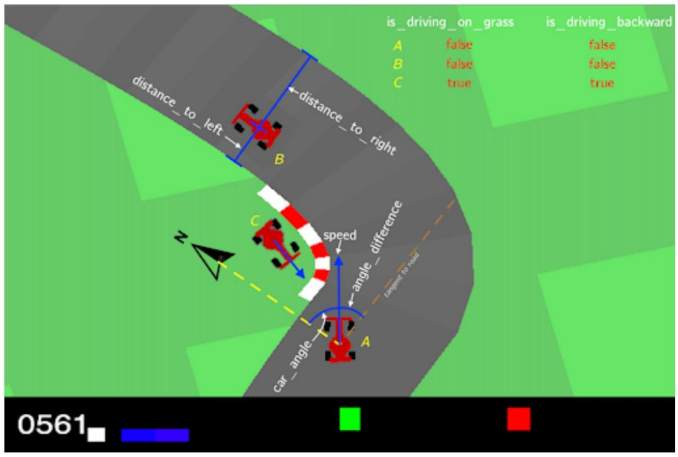

Fig. 3. Illustration of 7 types of sensor data

III-B Bootstrapping

During evaluation of the baseline models, we noticed that sometimes the agent deviates from the paved track, has no idea about how to turn appropriately, or just keeps accelerating. We regard these types of behavior as bad "driving habits", and for some reason the reward function fails to discourage these behaviors from the model. Consequently, the agent stops learning prematurely and is stuck with these sub-optimal policies. To cope with this dilemma, we explored several directions. One of our approach is the following bootstrapping method:

1) Increase the sample weight of bad driving and negative rewards, so the agent can focus on more learning from mistakes instead of repeating already-good behaviors. Specifically, in the replay buffer, repeat the last 50 game ticks before the car crashes to the edge of road for every episode.
2) Impose a "speed limit" by set a threshold on the agent's output action. Our intuition is that a speed greater than some threshold is almost certainly going to cause a crash at a sharp turn. This should also encourage the agent to brake when it's going too fast.

III-C Multi-stage training + bootstrapping

To further encourage good steering behavior and put less emphasis on speed. We divide the training process into 2 stages with different reward functions/"learning goals".

1) In the first phase, using the default default implementation, the reward function mainly rewards fast speed and slightly punishes bad steer. Speed limit is set to 60.
2) In the second phase, the reward function is designed to reward good steer only and punishes extremely bad steers with negative rewards. No reward/punishment is assigned for speed. Speed limit is set to 40.

Mathematically,

$$f_1(s) = 5s_7 - 5|s_1| - 20|s_6 - s_5| - 100s_3$$
$$f_2(s) = 10\left(\frac{\pi}{4} - |s_2|\right) + 10\left(6 - |s_6 - s_5|\right) - 100s_3$$

Note that during testing, we still use the first phase's reward scheme because all evaluations are based on it. The second reward function is created to encourage better steering behaviors, but we do not want we use it without the aid of the first one as otherwise, the agent may never learn to drive forward. That's why we think the multi-stage approach has its merits.

III-D Ensemble + Bootstrapping

Lastly, we would like to take advantage of ensemble learning method. In our setup, the environment/state is shared between all agents, and each agent (one actor + one critic) outputs an action vector (with proper noise addition if in training). The actions are aggregated using "mean" operation, which becomes the final action the actual agent takes. Our implementation of the ensemble agent is modified from the MADDPG algorithm, where each critic can also see and evaluate all actors' choices of actions. We think there are at least two main advantages of using ensemble methods in our setting:

1) Less sensitivity to initialization: single-agent DDPG is very sensitive to initial parameters and hyperparameters. Multiple DDPG agents reduce the group variance and are less likely to all have bad initialization.
2) Multiple exploration with good stability: more agent means more directions to explore in the action space. Taking the average of all results in good stability of the ensemble policy.

IV Experiment Setup

Car Racing Environment We conducted experiments in the `MultiCarRacing-v0` environment[3]. This environment is an extension of the original `CarRacing-v0` environment[4] from OpenAI Gym[5]. The environment includes a race track with curves, and a car agent finishes the race when it has traversed all track tiles. The race track shape and the car agent's positions are randomly generated for each episode. During the game, the car agent gains -0.1 reward per frame elapsed, and gains $+\frac{1000}{N}$ reward when it steps on a new track tile it has not stepped on before, where N is the total number of track tiles. The total reward is thus defined as $1000 - 0.1t$, where t is the time it takes for the car to finish the race.

Moreover, to facilitate training as well as preventing erratic behaviors by the RL agent, we immediately stop the episode once the car starts to drive on grass and assign it a -100 reward. While we are not fully utilizing the multi-agent setting of the environment, this extension certainly makes our ensemble method (MADDPG) implementation much easier to work with.

Metrics We then evaluated all of our methods based on the following metrics. Each method is evaluated and tested for 100 episodes and the average score is taken.

1) Average reward: a direct measure of the performance of the car agent. The reward calculation is multi-modal. There are many ways to achieve high reward values: driving faster, driving longer, or making better steer. A higher reward generally means the agent drives better.
2) Average episode length: how many game ticks the car agent can drive before it crashes/dirves to grass. The longer this value is, the less likely the agent makes fatal mistakes.
3) Average speed

Architecture Architectures of the CNN baseline we used are detailed in Table I and II. Architectures of our sensor net baseline are detailed in Table III and IV. All other non-baseline methods we explored follow the same architectures as those of our sensor net baseline, with slight adjustments as needed.

V Results

We show results of all of our methods in table V. Below are detailed discussion of our sensor network based methods.

Sensor Net Baseline For the sensor net baseline, we made the reward vs. epoch/episode plot. See Figure 8 for details. There are clearly two phases of rapid learning: 1. the smaller circle in the bottom left is when the agent learns to drive straight on a straight road, and 2. the larger one is when the agent learns to steer at a turn.

Also, during testing, we record the agent's state and action pairs for all game ticks. Since they are both continuous vectors, we divide the state variable (speed, angle, margin) into bins and for each bin we calculate the average of observed action variables (brake/gas, steer) to make the following Figure 4. These are three different visualizations: speed vs. brake/gas, angle vs. steer and margin[6] vs. steer.

[3]https://github.com/igilitschenski/multi_car_racing
[4]https://gym.openai.com/envs/CarRacing-v0/
[5]https://github.com/openai/gym

[6]The margin is defined as the difference of the distance to the right curb and the distance to the left curb

TABLE I
BASELINE CNN ACTOR

Layer (Type)	Output Shape	Number of Parameters
Input_1 (InputLayer)	[(None,84,96,3)]	0
Conv2d_1 (Conv2D)	(None,20,23,16)	1200
Conv2d_2 (Conv2D)	(None,6,7,32)	4608
Conv2d_3 (Conv2D)	(None,2,2,32)	9216
Flatten (Flatten)	(None,128)	0
Dense_1 (Dense)	(None,64)	8256
Dense_2 (Dense)	(None,2)	130

Total parameters: 23,410

TABLE II
BASELINE CNN CRITIC

Layer (Type)	Output Shape	Number of Parameters	Connected To
Input_3 (InputLayer)	[(None,84,96,6)]	0	-
Conv2d_6 (Conv2D)	(None,20,23,32)	4800	Input_3[0][0]
Conv2d_7 (Conv2D)	(None,6,7,64)	18432	Conv2d_6[0][0]
Conv2d_8 (Conv2D)	(None,2,2,64)	36864	Conv2d_7[0][0]
Flatten_2 (Flatten)	(None,256)	0	Con2d_8[0][0]
Input_4 (InputLayer)	[(None,4)]	0	-
Concatenate (Concatenate)	(None, 260)	0	Flatten_2[0][0],Input_4[0][0]
Dense_4 (Dense)	(None,128)	33408	Concatenate[0][0]
Dense_5 (Dense)	[(None,64)]	8256	Dense_4[0][0]
Dense_6 (Dense)	[(None,32)]	2080	Dense_5[0][0]
Dense_7 (Dense)	[(None,1)]	33	Dense_6[0][0]

Total parameters: 103,873

TABLE III
BASELINE SENSOR NET ACTOR

Layer (Type)	Output Shape	Number of Parameters
Input_1 (InputLayer)	[(None,7)]	0
Dense_1 (Dense)	(None,64)	512
Dense_2 (Dense)	(None,2)	130

Total parameters: 642

TABLE IV
BASELINE SENSOR NET CRITIC

Layer (Type)	Output Shape	Number of Parameters	Connected To
Input_5 (InputLayer)	[(None,7,1)]	0	-
Flatten_1 (Flatten)	(None,7)	0	Input_5[0][0]
Dense_11 (Dense)	(None,64)	512	Flatten_1[0][0]
Dense_12 (Dense)	(None,64)	4160	Dense_11[0][0]
Input_6 (InputLayer)	[(None,2)]	0	-
Concatenate_1 (Concatenate)	(None, 66)	0	Dense_12[0][0],Input_6[0][0]
Dense_13 (Dense)	[(None,64)]	4288	Concatenate_1[0][0]
Dense_14 (Dense)	[(None,32)]	2080	Dense_13[0][0]
Dense_15 (Dense)	[(None,1)]	33	Dense_14[0][0]

Total parameters: 11,073

TABLE V
MODEL PERFORMANCES

Method	Average Reward	Average Episode Length	Average Speed
CNN Baseline	23313	104	25.6
Sensor Net Baseline	40593	201	35
Bootstrapping	42536	220	**36**
Multistage Training + Bootstrapping	50277	306	28*
Ensemble + Bootstrapping	**77162**	**331**	28*

*: decreased due to lower speed cap

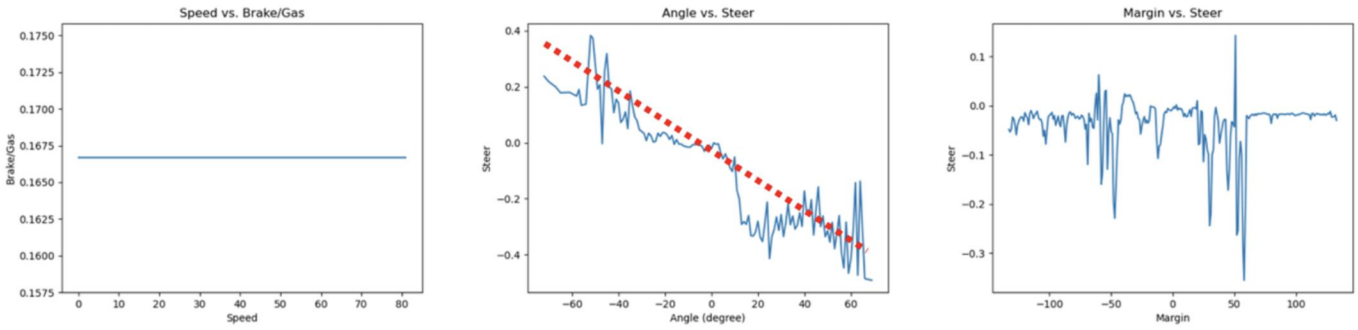

Fig. 4. Sensor net visualizations

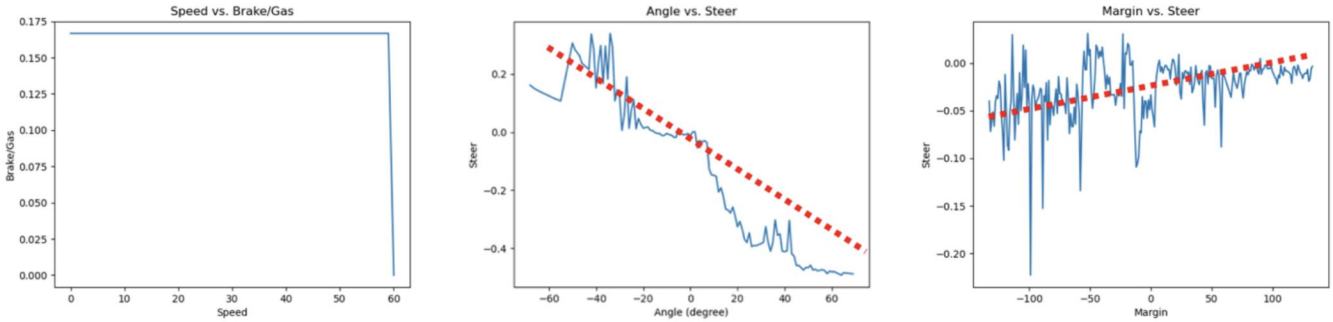

Fig. 5. Bootstrapping visualizations

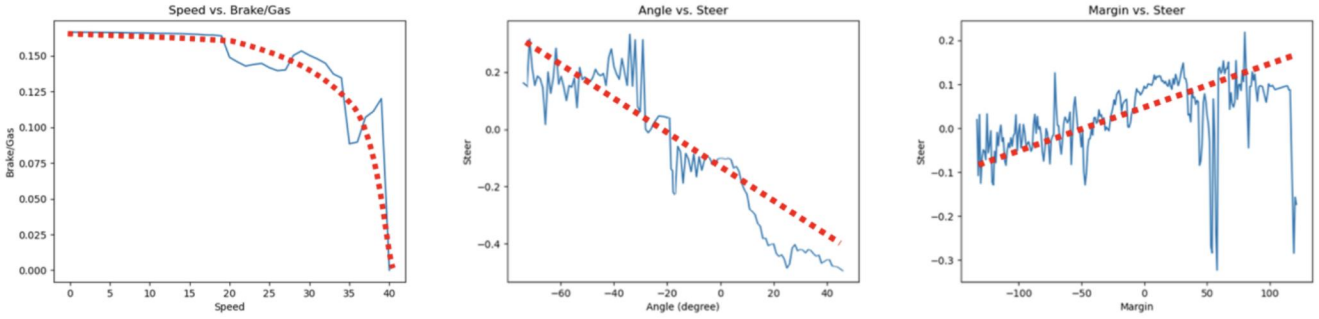

Fig. 6. Multistage training + Bootstrapping visualizations

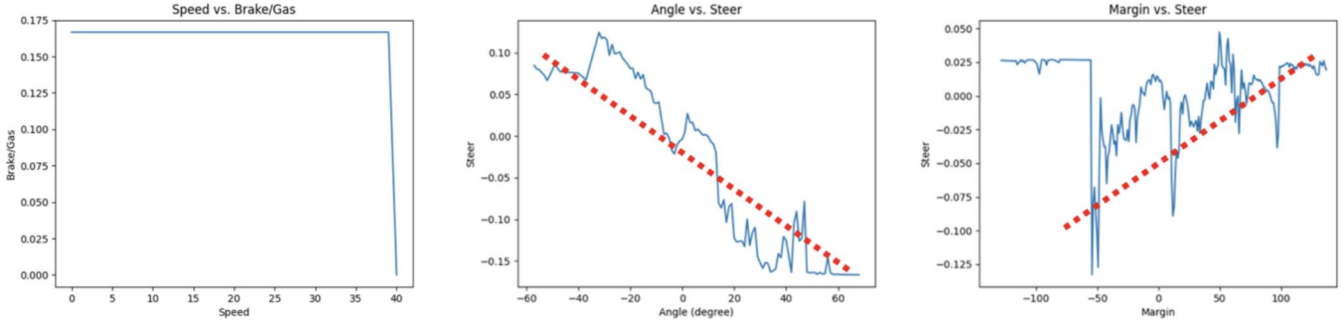

Fig. 7. Ensemble + Bootstrapping visualizations

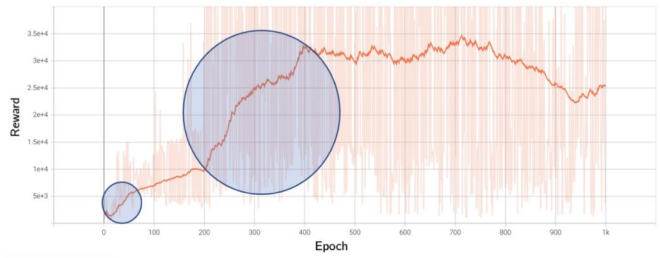

Fig. 8. Sensor net reward vs. epoch plot

We observe that the agent seems not to have learned to use the brake properly. It applies gas as much as it can. This is expected because the agent is rewarded for speed and penalized for stopping or slowing down. We also observe a strong negative linear relationship between the angle and the steer: the agent learns that the steer is strongly associated with the difference of the car's angle to the direction of the road. In other words, it can correct itself when it's heading a wrong direction. We presume this is how a sensor-based driving agent makes a turn. Lastly, the agent seems to not take advantage of knowing how close it is to the edge/curb of the road (margin).

Bootstrapping The reward vs. epoch plot of bootstrapping is similar to that of the sensor net baseline. However, we notice some changes in the other visualizations (see Figure 5).
The speed vs. brake plot clearly shows that the gas is cut sharply at 60, but the agent still does not learn how to brake properly. The angle and steer relationship is preserved from the last baseline. Lastly, we see a weak positive linear relationship between the margin and steer. The agent somewhat learns to steer left when it is too close to the right curb, but not the other way around.

Multistage training + Bootstrapping Here is the reward vs. epoch plot for the second stage training of this method (see Figure 9).

The circle marks the fast learning stage recurs in the second stage training, where the agent's steering policy rapidly improves. Note the reward here (training) is assigned by the new reward function in the second stage.
The visualizations of this method (Figure 6) shows that the agent learns to reduce gas when the speed is high due to the speed limit, which means enforcing the speed cap is beneficial and rewarding. The angle and steer relationship is preserved from the last baseline, and the margin and steer relationship becomes more prominent. We presume this is where the agent's steering policy mainly improves by the introduction of the multistage setup.

Ensemble + Bootstrapping The reward vs. epoch plot for this method is shown in Figure 10.

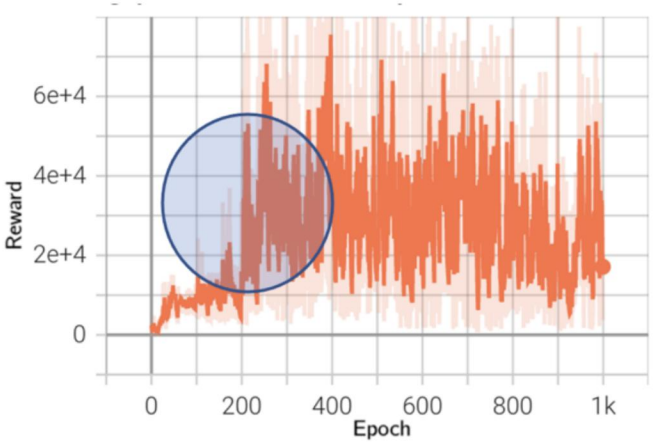

Fig. 9. Multistage + Bootstrapping second stage reward vs. epoch plot

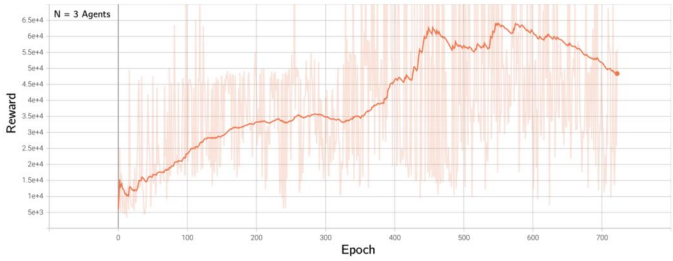

Fig. 10. Ensemble + Bootstrapping reward vs. epoch plot

The ensemble model takes more epochs to learn for each milestone (drive straight and make turns). But overall, the training plot looks much more stable and smooth compared those non-ensemble methods. The agent can garner a higher reward after training for more than approximately 450 epochs.

In Figure 7, We observe that the ensemble method does not learn to brake, unlike the agent in multistage training. The angle-and-steer relationship is preserved from the last method. The margin-and-steer relationship becomes very prominent compared to all other methods.

VI Conclusion

We see a progression of improvements been made along the way in terms of rewards and episode lengths. From the the sensor net baseline, which achieves an average reward of 40k and an average episode length of 201, we see a small increase by adopting the bootstrapping method, and a much larger increase by introducing the multistage training approach. This shows that we can improve the the performance of a RL agent by identifying the issues the agent is struggling with (in our case, learning to drive and learning to take turns), and breaking down the learning process into multiple steps to let the the agent focuses on learning each one individually.

There is another huge boost to the average reward (and a smaller boost to the average episode length) after utilizing the ensemble learning strategy. Ensemble methods have proven to be able to provide more robust and better overall results in many domains [20], and we show it is also quite powerful in our car racing setting, with a huge performance increase compared to non-ensemble methods. However, the ensemble method does not learn to brake, which is a drawback.

In terms of speed, we set a lower speed limit for multistage learning and ensemble methods, and as a result, the average speed decreases for those two. However, we still see major reward increases for these two methods, which shows that a higher speed does not necessarily leads the agent to a higher reward in the end. This aligns with the common sense in the domain of car racing, that it is better to steer smartly and turn properly, than to recklessly speeding up.

VII Future Works

We think there are several directions for potential future work:

1) Training is still unstable: ensemble model mitigates but not eliminates this problem. This might be an inherent drawback of DDPG. How about using other backbone RL algorithms?
2) We showed the effectiveness of ensemble methods. However, our current ensemble method is a simple average of all sub-models. Can we do better by assigning weights for sub-models? Then, how can we calculate these weights? Can we learn these weights by using some neural network?
3) We bypassed representation learning by avoiding CNNs. If we had enough computing resource (much more powerful GPUs), will our add-ons (bootstrapping, multistage training, ensemble) still be as effective?

References

[1] E.Alcala, V.Puig, J.Quevedo, and U.Rosolia. Autonomous racing using Linear Parameter Varying-Model Predictive Control (LPV-MPC).

[2] A. Liniger and J. Lygeros. Real-Time Control for Autonomous Racing Based on Viability Theory. IEEE Transactions on Control Systems Technology, 27(2):464–478, 2019. 1, 2.

[3] R. Spica, E. Cristofalo, Z. Wang, E. Montijano, and M. Schwager. A Real-Time Game Theoretic Planner for Autonomous Two-Player Drone Racing.

[4] Z. Wang, R. Spica, and M. Schwager. Game Theoretic Motion Planning for Multi-robot Racing. In International Symposium on Distributed Autonomous Robotic Systems (DARS), 2019. 2.

[5] D.Silver, J.Schrittwieser, K.Simonyan, I.Antonoglou, A.Huang, A.Guez, T.Hubert, L.Baker, M.Lai, A. Bolton, et al. Mastering the game of go without human knowledge. nature, 550(7676):354–359, 2017. 2.

[6] N. Brown and T. Sandholm. Superhuman ai for multiplayer poker. Science, 365(6456):885–890, 2019. 2.

[7] M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In International Conference on Machine Learning (ICML), 1994. 3.

[8] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. Artificial intelligence, 101(1-2):99–134, 1998. 3

[9] Pomerleau, D.A., 1989. Alvinn: An autonomous land vehicle in a neural network. In Advances in neural information processing systems (pp. 305-313).

[10] Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L.D., Monfort, M., Muller, U., Zhang, J. and Zhang, X., 2016. End to end learning for self-driving cars. arXiv preprint arXiv:1604.07316.

[11] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. and Zaremba, W., 2016. Openai gym. arXiv preprint arXiv:1606.01540.

[12] Schwarting, W., Seyde, T., Gilitschenski, I., Liebenwein, L., Sander, R., Karaman, S., & Rus, D. (2021). Deep latent competition: Learning to race using visual control policies in latent space. arXiv preprint arXiv:2102.09812.

[13] Zhang, Y. (2020). Deep reinforcement learning with mixed convolutional network. arXiv preprint arXiv:2010.00717.

[14] Guo, F., & Wu, Z. (2018, June). A Deep Reinforcement Learning Approach for Autonomous Car Racing. In International Conference on E-Learning and Games (pp. 203-210). Springer, Cham.

[15] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, Proximal Policy Optimization Algorithms. arXiv, 2017.

[16] Silver, David, Lever, Guy, Heess, Nicolas, Degris, Thomas, Wierstra, Daan, and Riedmiller, Martin. Deterministic policy gradient algorithms. In ICML, 2014.

[17] T. P. Lillicrap, Continuous control with deep reinforcement learning. arXiv, 2015.

[18] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, I. Mordatch, Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. arXiv, 2017.

[19] J. Wu and H. Li, Deep Ensemble Reinforcement Learning with Multiple Deep Deterministic Policy Gradient Algorithm, Mathematical Problems in Engineering, vol. 2020, pp. 12, 2020

[20] Wiering, M.A. and Van Hasselt, H., 2008. Ensemble algorithms in reinforcement learning. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 38(4), pp.930-936.