Evaluating Robustness of Deep Reinforcement Learning for Autonomous Driving

How does entropy maximization affect the training and robustness of final policies under various testing conditions?

Bartu Mehmet Ortal[1]

Supervisor(s): Matthijs Spaan[1]**, Moritz Zanger**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

Abstract

This research paper aims to investigate the effect of entropy while training the agent on the robustness of the agent. This is important because robustness is defined as the agent's adaptability to different environments. A self-driving car should adapt to every environment that it is being used in since a mistake could cost someone's life. Therefore, robustness is of great importance in self-driving cars. An increase in entropy would promote the exploration of different strategies and prevent convergence on local maximum results. In order to test entropy values in training, the Soft-Actor Critic algorithm is used. The algorithm is run on a simulated city environment called Carla. In the end, collected data shows that the agent which is trained with a higher entropy value adapts to environments it did not train on better than the low entropy agent. However, a low entropy agent performs better in the environment it is trained in. Therefore, increasing the entropy increases robustness but it lowers the performance in the training environment itself.

1 Introduction

Autonomous driving is a big point of interest for many car manufacturers and researchers. The ease it provides to humans and allowing quick responses that humans would not be able to perform is a big advantage and reason for it to be researched. Even though it provides such benefits, implementing self-driving car technology without making sure it works in all environments imaginable could result in catastrophic scenarios such as car crashes. In order to prevent this, agents can be trained using Deep Reinforcement Learning methods.

Deep Reinforcement Learning methods are an extension of Reinforcement learning methods and use deep neural networks (neural networks with multiple layers) in order to decide upon an action. Simple Deep Reinforcement Learning approaches tend to perform poorly when the environment changes. This makes them undesirable to train self-driving cars. Therefore the solution should be searched elsewhere. The approach we will take is to augment the typical reinforcement learning objective with a maximum policy entropy term. This means that the method we tackle in this research tries to learn a policy that not only tries to get as much reward as possible but also does it as randomly as possible.

Maximizing entropy means that the agent will try random ways instead of sticking to already learned methods. This allows the agent to discover more ways to complete the given task. Since the agent knows how to complete the given task in multiple ways, introducing a new obstacle to the environment may be solved using knowledge gained using this method. An example can be provided from work done by Benjamin Eysenbach and Sergey Levine [16]. Comparing standard reinforcement learning and maximizing entropy on a robot arm that tries to move an object to a target location shows that when a new wall is introduced, simple reinforcement learning almost always fails, whereas maximum entropy completes the task 95% of the time. Since it can adapt to different environments, it is also said to be a robust method of training the agent.

The research question we will investigate in the paper is "How does entropy maximization affect the training and robustness of final policies under various testing conditions?". In order to show this, the Soft-Actor Critic (SAC) algorithm will be used with various target entropy values. Different entropy values and environments will be simulated in Carla [6]. Carla is a really realistic simulator that will allow me to simulate real-life situations in a controlled environment. Using such technology and utilizing implementation methods provided by work done by previous researchers, entropy maximization will be investigated. This investigation will be done in the following steps:

1. **Performance during agent training**: Different entropy values may have an effect on the performance of training. This can be in the form of faster convergence or convergence on a different value for the agent's rewards. For this step, we will identify differences and try to explain their causes. When we have a good reason for the results, we will comment on their strength and weaknesses.
2. **Performance during evaluation**: When the agents are trained, they will be tested on different environments. This is done in order to answer the research question of this paper. Robustness is the agent's adaptability to different environments therefore performance of the agent in different environments will be investigated.

2 Background Information

2.1 Markov Decision Process

Markov Decision Process is a stochastic decision-making process that uses a mathematical framework to model the decision-making of a dynamic system. It uses tuples $\langle S, A, T, R, \gamma \rangle$ where S is the set of states that the agent can be in. Each state represents a different configuration that the agent can find itself in. A is the set of actions that the agent can take. In an autonomous car example, this can be steering angle, breaking, etc. T is the transition model. It is a function $T(s, a, s')$ that gives the probability that the agent transitions to the state s' from state s by taking the action a. R is the reward model. It is a similar function to the transition function but gives the reward obtained by taking the action rather than probability. Finally, γ is the discount factor. It is a value between 0 and 1 that represents the agent's preference for immediate or future rewards.

Ultimately, Markov Decision Processes try to find the best possible mapping from states to actions such that the agent knows which action at state s will give the greatest reward at the end. In the case of reinforcement learning, Q-learning is commonly used in order to find such mapping in Markov Decision Processes.

2.2 Q-learning

Q-learning is a simple way for agents to learn how to act optimally in controlled Markovian domains [18]. In order to do

$$Q(s,a) = (1-\alpha) \cdot Q(s,a) + \alpha \cdot (r + \gamma \cdot max_{a'} Q(s', a'))$$

Figure 1: Bellman equation to update Q-values [18]

this, it keeps track of a table called Q-table. This table stores Q values (expected reward) for all state-action pairs. Each step of Q-learning updates the Q-table based on the Bellman equation (Figure 1). This equation states that the optimal Q-value for the given state-action pair is the immediate reward (r) plus the expected maximum reward from future ($max(Q(s', a'))$) discounted by the discount factor (γ). α is the learning rate which is the weight given to the new information. Q-learning converges to the optimum action-values with probability 1 so long as all actions are repeatedly sampled in all states and the action-values are represented discretely [18].

2.3 Q-networks

Q-network is a specific type of artificial neural network that is used to approximate the Q-function. Instead of using a table to represent Q-values, it uses a neural network in order to estimate the Q-values. However, a single Q-network may suffer from an overestimation of action values. Therefore two separate Q-networks can be used. In this architecture, action selection and action evaluation are done by different networks which in theory reduces overestimation. Training is often unstable due to fast-changing target Q-values, and target networks are employed to regularize the Q-value estimation and stabilize training by using an additional set of lagging parameters [15]. Target networks are replicas of their respective networks. Their parameters are updated slowly through soft updates in order to align with the parameters of their respective networks. The goal of this approach is to mitigate the issues of overestimation bias. It also accelerates learning stability.

2.4 Experience Replay

In traditional reinforcement learning, the training agent uses the experience to learn and immediately moves on to the next experience, discarding previous experiences. In experience replay, the agent has a buffer that stores the history of experiences. While training, rather than using previous experience, a random sample from the buffer can be used. Experience replay not only provides uncorrelated data to train a neural network, but also significantly improves the data efficiency [17].

2.5 Soft Actor-Critic

The algorithm used to train the agent in this paper is the Soft Actor-Critic algorithm. Soft Actor-Critic is an algorithm that optimizes a stochastic policy in an off-policy way, forming a bridge between stochastic policy optimization and DDPG-style approaches [1]. The main reason we chose to use the SAC algorithm is because it allows entropy regularization. This means that the policy is trained to maximize a trade-off between expected return and entropy [1]. In addition to this, our algorithm incorporates the usage of two Soft Q-network to reduce the overestimation bias issue in Q-network-based methods [11]. The method alternates between collecting experience from the environment with the current policy and updating the function approximators using the stochastic gradients from batches sampled from a replay pool. Using off-policy data from a replay pool is feasible because both value estimators and the policy can be trained entirely on off-policy data [10].

$$Q(s,a) = r + \gamma \cdot (V(s') - \alpha \cdot \log(\pi(a|s)))$$

Figure 2: Soft Actor-Critic Bellman update equation [8]

$$y = r\left(s,a\right) + \gamma \pi\left(a|s'\right)^T \left(\min_{\theta_{1,2}} Q_{\theta_i}\left(s'\right) - \alpha \log \pi\left(\cdot|s'\right)\right)$$

Figure 3: CleanRL Bellman update target function [11]

In the Soft Actor-Critic algorithm, Bellman update equation is modified in order to give importance to entropy. This can be seen in figure 2. Other than the variables we have explained before, $V(s')$ represents the estimated expected cumulative reward for being in state s' and following the current policy. $\pi(a|s)$ is the policy's probability of taking action a in state s. In addition to figure 1, this equation subtracts the entropy regularization term in order to calculate the Q-value. This encourages exploration by giving less value to low-entropy actions.

Instead of coding the Soft Actor-Critic(SAC) algorithm from the ground up, the CleanRL [11] implementation is used. This implementation supports discrete action spaces and exploits the discrete action space by using the full action distribution to calculate the Soft Q-targets instead of relying on a Monte Carlo approximation from a single Q-value. In order to do this, it modifies the Bellman update equation as shown in figure 3.

It can be seen that the function takes the minimum Q-network value. This is done in order to reduce the overestimation bias. The Bellman update target is modified for discrete action space such that Q only takes states as inputs and it generates Q-values for all actions. The Algorithm uses this and weighs the target by the corresponding action selection probability in order to reduce the variance of the gradient [11]. Ultimately, the objective function is given in figure 4.

Objective function given in Figure 4 has $\pi_\phi(\alpha|s)$ term. Logarithm of this term is the target entropy used at Table 1. By changing this value, we modify the algorithm such that we change the entropy and this changes probability of exploitation vs exploration.

$$\max_{\phi} J_\pi(\phi) = E_{s\sim D}\left[\pi(\alpha|s)^T \left(\min_{i=1,2} Q_{\theta_i}(s) - \alpha \log \pi_\phi(\alpha|s)\right)\right]$$

Figure 4: Objective function used in the CleanRL implementation [11]

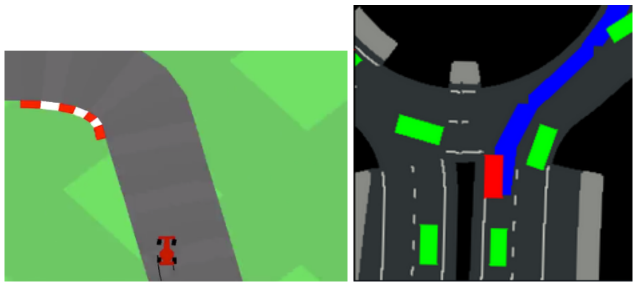
Figure 5: CarRacing(left) and Carla bird-eye view(right) observation space

Given the stochastic nature of the policy in SAC, the actor (or policy) objective is formulated so as to maximize the likelihood of actions that would result in a high Q-value estimate. Additionally, the policy objective encourages the policy to maintain its entropy high enough to help explore, discover, and capture multi-modal optimal policies [11]. Maximum entropy RL augments the reward with an entropy term, such that the optimal policy aims to maximize its entropy at each visited state [8].

Robustness is the adaptability of the trained agent to different environments such as different maps and car populations. The aim of this research is to investigate how entropy affects robustness. Entropy is our likelihood of selecting unexplored random options instead of trying to improve current good ones.

3 Methodology

In order to test how different entropy values affect the robustness, a suitable environment that will run the simulations is required. In order to achieve this, three frameworks are used. First of all, the gym framework by OpenAI [2] is at the center of this research. The gym framework provides a CarRacing environment which is used for early test runs before the Carla simulator [6]. On top of this, a repository called CleanRL [11] is used in order to implement required algorithms and run them on provided environments. After it was verified that the agent learns on the CarRacing environment, we had to make the algorithm compatible with the Carla simulator. This is done by using a framework called gym-carla [3], which forms a communication between Carla and the gym framework. Carla simulations were run after the OpenAI gym simulations. This is because Carla uses a lot more physics logic and is slower than the gym CarRacing environment. Ordering work like this proved to be more efficient than directly running Carla simulations while debugging the algorithm to make sure it learns from the environment. Since the Carla simulator is a computationally (GPU particularly) heavy framework and takes a full day to train, training was done on delft-blue [5] GPU nodes. This way, agents could be trained without interruption and multiple tasks could run at the same time which proved to be more efficient.

In order to tune hyper-parameters, the SAC agent was trained in the CarRacing environment. From these trials, entropy values that will be used to train agents in the Carla environment were selected. Training on the CarRacing environment makes sense because the observation space provided by the CarRacing environment is similar to the one provided by the gym-carla bird-eye view. Both of them essentially require the agent to follow a line (grey road for the CarRacing and blue line for the Carla environment). Carla environment makes the agent's task a bit more complicated by adding other cars to the road. However, due to the similarities, we have chosen to test and debug the training of the agent on the CarRacing environment.

The steps for the procedure are given below to make it easier to understand.

Implement and test entropy maximization using SAC

1. Make sure Soft-Actor Critic learns on visual (pixel) input. Use CarRacing from the gym framework to test.
2. Train SAC.
3. Check trained data and analyze it.
4. Use the gym-carla framework to train the SAC agent on the Carla simulator.
5. Run SAC agent in different environments (maps) on Carla. Check how it performs in different environments.
6. Repeat the last step with different entropy values(0.5, 0.6).

Compare agents

1. Compare how each entropy value performs when tested on a similar environment that they trained on.
2. Compare how each entropy value performs when tested on a different environment/map.

Agents are trained until their rewards converge and their episodic return versus step graphs are compared. The goal is to identify differences between obtained graphs such as converging to better results or converging faster. Ultimately, the rewards of different agents in different environments are compared in order to find a relationship between entropy and robustness.

Using this methodology, it is aimed to show the effect of different entropy values while training on adaptability to different environmental variables such as different maps and car density.

4 Experimental Setup and Results

4.1 Setting up the environment

As we have indicated before, the implementation of the SAC algorithm is taken from the CleanRL [11] repository. This implementation offers out-of-the-box discrete action space agent training. It utilizes Convolution Layers [14] and rescaling of observation space [13]. However, some changes to the CleanRl code have been made. Due to hardware limitations, we have chosen to use Delft Blue [5] supercomputer. Even though the GPU node we used in order to train our agent trained around 50% faster, the SAC algorithm is computationally heavy. Even when we run the algorithm for 24 hours (the maximum amount of time that the GPU node can be reserved for our permission level), the agent would complete

Parameter Name	Value
Seed	1
total timesteps	100000
buffer size	100000
gamma	0.99
tau	1.0
batch size	64
learning starts	10000
policy learning rate	0.0003
q learning rate	0.0003
update frequency	2
target network frequency	4000
alpha	0.2
autotune	True
target entropy scale	[0.5, 0.6]

Table 1: Parameters used to train SAC

100 thousand steps with no clear convergence. This meant that we have to run the algorithm in multiple sessions.

In order to make the algorithm usable for the purposes of this paper, following changes were made:

1. **Checkpointing and saving the model**: The CleanRL implementation of the SAC algorithm did not have checkpoint and model-saving functionality. For the purposes of this paper, the code was modified in order to add these functionalities.
2. **Implementing a way to evaluate the trained agent**: The CleanRL implementation did not provide a way to evaluate the trained agent out of the box. In order to be able to train the saved agent, the evaluation code of the DQN algorithm in the same repository is taken as a basis. It was modified so that the action space generation was done through the trained agent of the SAC algorithm.

4.2 Training the agent

After these steps were complete, our agent could be safely trained remotely on the Delft Blue [5] nodes. Different jobs were submitted at the same time to use our time as efficiently as possible. Hyper-parameter values obtained from running the CarRacing environment given in Table 1 are used for the Carla environment. When we investigate Figure 6, no clear difference can be seen between the agent which trained with an entropy value of 0.5 and 0.6. Looking at Figure 7 and 8, which shows Q-values, we can see that the training performance of the agents is similar. It can be observed that none of the agents trained until convergence. We will explain the reason behind this in section 7, limitations.

4.3 Evaluating the agents

After agents were done with training, they were evaluated in different environments. Each agent ran 10 times on 3 different maps, Town4 being the map they have trained on. This means that agents were evaluated based on how they performed in the environment they trained on as well as how they performed in the environment that they did not train on.

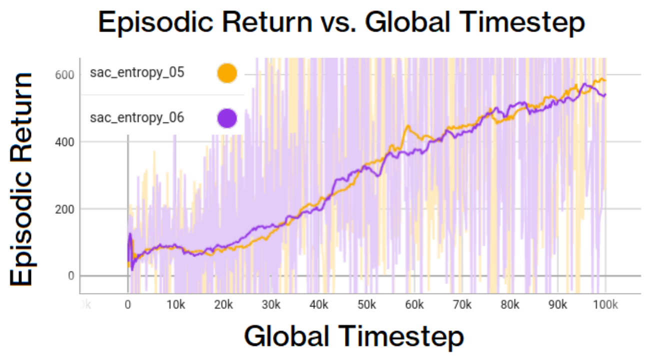

Figure 6: SAC Agent episodic-return vs steps graph. Steps (horizontal), ER (vertical)

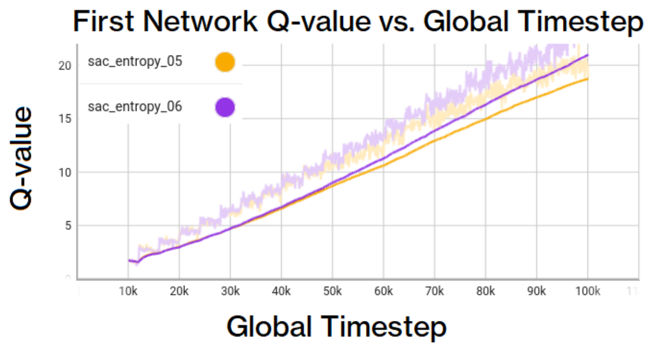

Figure 7: SAC Agent Q-values (first q-network) vs steps graph. Steps (horizontal), Q-values (vertical)

Their performance on the environments that they did not train on will give us their robustness as that is how we defined robustness. Figure 8 shows agents' performances on different maps. Every agent trained on Town4 and did not train on Town3 or Town5.

5 Discussion

In contrast to other off-policy algorithms, SAC approach is very stable, achieving very similar performance across different random seeds [9]. It was shown that the SAC algorithm itself is a robust algorithm. However, it does need a good value for the target entropy value. In this paper, how the entropy value used in the algorithm affects robustness is investigated. Looking at the final evaluation data collected represented in Figure 8, we can see that the entropy value of 0.5 outperforms the entropy value of 0.6 in the Town4 environment, which is the environment that they trained on. This can be explained as the agent doing fewer random actions which makes it easier to adapt to the same environment. Since it sticks to better actions more often, it adapts to the environment better. However, when we look at Town2, we can see that agent with higher entropy outperforms lower entropy. Reasoning behind this can be random actions preparing the agent better for states that the agent is not familiar with. Therefore when agent faces a new environment, the agent which trained with

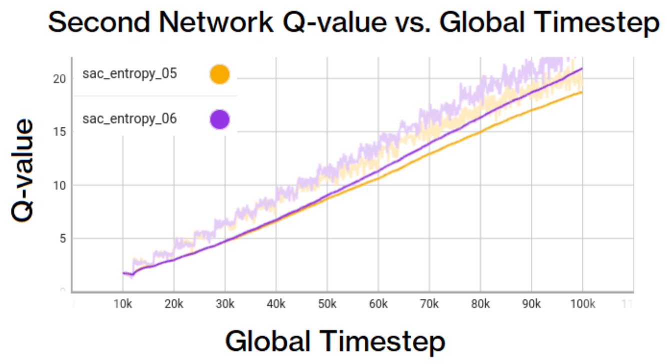

Figure 8: SAC Agent Q-values (second q-network) vs steps graph. Steps (horizontal), Q-values (vertical)

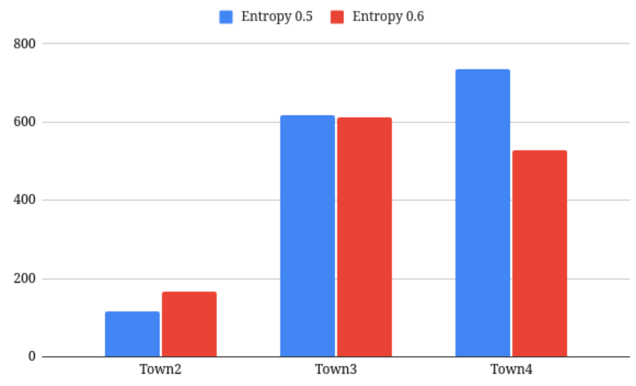

Figure 9: Comparison of evaluation results of different agents on different maps

more entropy is more likely to be familiar with those states. Even though Town2 makes us think this way, Town3 does not directly suggest this conclusion. This suggests that further testing is required (training longer and more samples for evaluation) to reach a more concrete conclusion. Ultimately, our data suggest that entropy makes agents adapt to different environments faster while making it harder to train for a specific environment. This suggests that the data collected through experiments support our initial hypothesis. This shows that the agent trained with a higher entropy value adapts to different environments better due to using its time more on exploration rather than exploitation.

6 Responsible Research

Even though the main goal of this research is to train cars in order to teach them how to drive in traffic, the models used in this paper are very primitive compares to what is required for agents that should be implemented in the real world. This is also the reason why both training and testing processes are in virtual environments. Therefore, the findings of this research should be taken into account but the produced model should not be directly adapted. In addition to this, the conclusion was reached by non-converging agents. Therefore, there is a chance that the outcome will change and suggest a different conclusion when it is possible to train the agents until they converge. Conclusions of this paper should be taken with caution because of this and it is encouraged to try and reproduce what is described in this paper.

7 Conclusions and Future Work

In this paper, we wanted to investigate how entropy maximization affects the training and robustness of final policies under various testing conditions. In order to show that, two different agents were trained with different entropy values. Their training data was investigated and it was seen that they do not show any significant difference during learning. However, when they were evaluated in different environments (different maps), agents performed differently. The agent with higher entropy was able to outperform in the environment Town2 which they did not train on and nearly the same result in the environment Town3. As for the environment they trained on, the lower entropy agents achieved better results during evaluation. This suggests that when the entropy is increased, agents adapt better to the environment that they are training in, however, agents with higher entropy will achieve better results in environments that they are not familiar with. In conclusion, increasing the entropy increases the robustness of the agent.

For future improvements, the experimentation step can be improved as discussed in parts 5 and 6. In order to confirm the findings of this paper, agents can be trained more steps (if converges, until convergence) and more samples for the evaluation step can be used. Other possible improvements include changing different settings such as the number of cars and pedestrians in CARLA. Lastly, more agents can be trained and compared with each other in order to make the findings more significant. Due to limitations of time and hardware, only two agents could be trained. However, more agents with a wider range of entropy values would make the research more significant in terms of testing the hypothesis and findings of this paper.

References

[1] Joshua Achiam. Spinning Up in Deep Reinforcement Learning. 2018.

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

[3] Jianyu Chen. gym-carla. https://github.com/cjy1992/gym-carla, 2020.

[4] Petros Christodoulou. Soft actor-critic for discrete action settings, 2019.

[5] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 1). https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase1, 2022.

[6] Alexey Dosovitskiy, Germán Ros, Felipe Codevilla, Antonio M. López, and Vladlen Koltun. CARLA: an open urban driving simulator.

[7] Tuomas Haarnoja, Sehoon Ha, Aurick Zhou, Jie Tan, George Tucker, and Sergey Levine. Learning to walk via deep reinforcement learning, 2019.

[8] Tuomas Haarnoja, Haoran Tang, Pieter Abbeel, and Sergey Levine. Reinforcement learning with deep energy-based policies, 2017.

[9] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

[10] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft actor-critic algorithms and applications, 2019.

[11] Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G.M. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022.

[12] Beakcheol Jang, Myeonghwi Kim, Gaspard Harerimana, and Jong Wook Kim. Q-learning algorithms: A comprehensive classification and applications. *IEEE Access*, 7:133653–133667, 2019.

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[14] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks, 2015.

[15] Alexandre Piché, Valentin Thomas, Joseph Marino, Gian Maria Marconi, Christopher Pal, and Mohammad Emtiyaz Khan. Beyond target networks: Improving deep q-learning with functional regularization, 2022.

[16] Wenjie Shi, Shiji Song, and Cheng Wu. Soft policy gradient method for maximum entropy deep reinforcement learning, 2019.

[17] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay, 2017.

[18] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

A CARLA Parameters

’number_of_vehicles’: 100,

’number_of_walkers’: 0,

’display_size’: 256, # screen size of bird-eye render

’max_past_step’: 1, # the number of past steps to draw

’dt’: 0.1, # time interval between two frames

’discrete’: False # whether to use discrete control space

’discrete_acc’: [-3.0, 0.0, 3.0], # discrete value of accelerations

’discrete_steer’: [-0.2, 0.0, 0.2], # discrete value of steering angles

’continuous_accel_range’: [-3.0, 3.0], # continuous acceleration range

’continuous_steer_range’: [-0.3, 0.3], # continuous steering angle range

’ego_vehicle_filter’: ’vehicle.lincoln*’, # filter for defining ego vehicle

’port’: 2000, # connection port

’town’: ’Town04’, # which town to simulate

’task_mode’: ’random’, # mode of the task, [random, roundabout (only for Town03)]

’max_time_episode’: 1000, # maximum timesteps per episode

’max_waypt’: 12, # maximum number of waypoints

’obs_range’: 32, # observation range (meter)

’lidar_bin’: 0.125, # bin size of lidar sensor (meter)

’d_behind’: 12, # distance behind the ego vehicle (meter)

’out_lane_thres’: 2.0, # threshold for out of lane

’desired_speed’: 8, # desired speed (m/s)

’max_ego_spawn_times’: 200, # maximum times to spawn ego vehicle

’display_route’: True, # whether to render the desired route

’pixor_size’: 64, # size of the pixor labels

’pixor’: False, # whether to output PIXOR observation