A Hierarchical Architecture for Sequential Decision-Making in Autonomous Driving using Deep Reinforcement Learning

Majid Moghadam [1] **Gabriel Hugh Elkaim** [1]

Abstract

Tactical decision making is a critical feature for advanced driving systems, that incorporates several challenges such as complexity of the uncertain environment and reliability of the autonomous system. In this work, we develop a multi-modal architecture that includes the environmental modeling of ego surrounding and train a deep reinforcement learning (DRL) agent that yields consistent performance in stochastic highway driving scenarios. To this end, we feed the occupancy grid of the ego surrounding into the DRL agent and obtain the high-level sequential commands (i.e. lane change) to send them to lower-level controllers. We will show that dividing the autonomous driving problem into a multi-layer control architecture enables us to leverage the AI power to solve each layer separately and achieve an admissible reliability score. Comparing with end-to-end approaches, this architecture enables us to end up with a more reliable system which can be implemented in actual self-driving cars.

1. Introduction

Advanced Driving Assistance Systems (ADAS) are developed to increase traffic safety by reducing the impact of human errors. The evolution of various levels of driving autonomy has seen a significant speed up in last years aiming to enhance comfort, safety and driving experience. For a long time, with a limited amount of technological resources, automotive stakeholders were focusing on steady-state maneuvers to achieve some sort of autonomy. However, in recent years the major focus of research in the field of autonomous driving is being directed to the transition maneuvers, most importantly tactical lane changing, required for both fully and partially autonomous driving systems (Alizadeh et al., 2019).

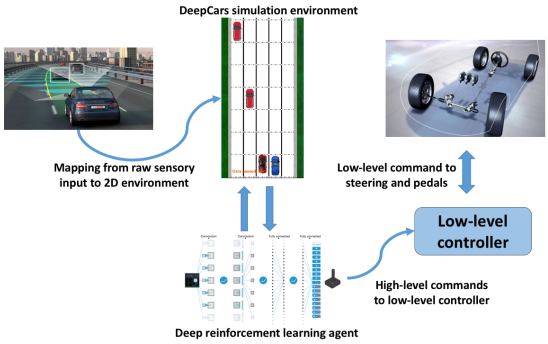


Figure 1. A sketch of our hierarchical approach for the autonomous driving problem

Before AI, control and orientation of ground vehicles were tackled using feedback control techniques (Falcone et al., 2007; Broggi et al., 1999; Moghadam & Caliskan, 2015) that attempt to stabilize the vehicle using the information collected from sensory measurements. The controller used to generate input commands to the actuators like steering angle, accelerator, and break to perform the driving tasks. Because of the extraordinary success of the neural networks (NN) in the classification (Krizhevsky et al., 2012) and regression (Specht, 1991) problems, the researchers have decided to apply learning-based approaches to the control problems. Leveraging Deep Reinforcement Learning (DRL) agents to play Atari games (Mnih et al., 2015) was one of the earliest methods that gained considerable attention in the field. After the success of DRL on video game plays, researchers applied DRL on real-world problems like control and orientation of autonomous systems which have shown a great potential to become a reliable alternative for the classical control approaches. There has been various deep learning (DL) techniques that applied to the problem of autonomous vehicles. One approach is to use an expert knowledge of the problem and train a neural network to learn the expert policies in an end-to-end manner. This method is known as imitation learning and has been applied to both self-driving cars (Bojarski et al., 2016) and unmanned aerial vehicles (Bicer et al., 2019). Although the results are promising for

[1]Computer Science and Engineering Department, University of California, Santa Cruz, USA. Correspondence to: Majid Moghadam <mamoghad@ucsc.edu>.

Real-world Sequential Decision Making workshop at ICML 2019, Long Beach, California.

training an intelligent agent that shows off comparable performance as human, such agents may never outperform the human expert as the training dataset is being annotated using solely expert demonstrations. Similar end-to-end architecture is also used to train DRL agents to drive autonomous cars in simulation environments, like TORCS (Koutník et al., 2013) and World Rally Championship (Perot et al., 2017). Here the term end-to-end indicates that the raw pixel information of on-board camera on the vehicles are being used as the state representation and continuous steering angle and acceleration is being calculated via DRL algorithm. End-to-end DRL approaches have the ability to outperform the human expert in various driving tasks. This can be verifies by the results published in the mentioned articles. However, the safety and reliability of such algorithms become critical problems when applied to real-world problems. In addition, the complexity of the environment in real-world applications may impeach the performance of these algorithms. In other words, combining different layers of the ADAS systems which took long to reach the current performance in an end-to-end architecture which maps the observations directly to the actions, does not seem to guarantee the similar performance in actual systems.

1.1. Our approach

As discussed above, in most of the studies in the literature the raw sensory inputs like video frames of on-board camera or RGB-D sensors on automobile have been used to train a neural network in order to estimate the required action to control the vehicle (see Fig.2). Most of the studies validated the performance of their methods in a number of unreal simulations or video games. We believe that when it comes to the real-world implementation of these methods, the end-to-end learning techniques may suffer the lack of reliability to control the agent as a result of huge number of uncertainties while interacting in the actual world. To this end, we divide the problem into different control level problems and try to solve each level separately. In other words, the raw sensory inputs are going to be used to percept the environment and map it to a simpler two-dimensional world. Accordingly, the AI agent in decision making layer will generate high-level decision commands like left/right lane change and send them to lower-level control layers which deal with steering angle and gas pedal commands to stabilize the vehicle while following the generated high-level actions. This approach is summarized in Fig.1. Also the ADAS architecture is provided in Fig.2. Note that we use the occupancy grid as the environment model around the ego vehicle. We believe that, this approach is more reliable in real-world implementations than end-to-end techniques available in the literature because we do not perform a single mapping from the highest level observations with considerable number of uncertainties to the lowest level actuator control layer. This approach is much like the multi-loop control techniques in control engineering which has proven to show better performance in stabilizing complex systems (Slotine et al., 1991; Moghadam et al., 2018).

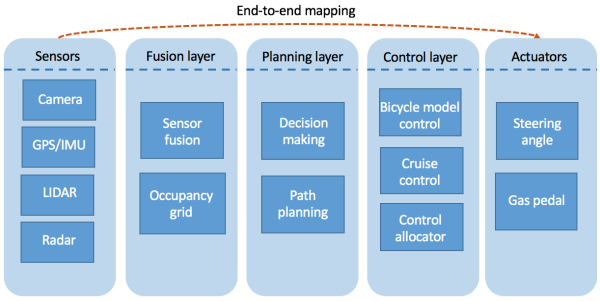


Figure 2. Hierarchical architecture of the general ADAS systems vs. end-to-end approaches

In this study, we address the problem of high-level decision making for an autonomous car using classical reinforcement learning technique known as Q-learning. We implement the $\epsilon-$greedy algorithm to the problem defined in DeepCars simulation environment which is also designed and implemented by the authors. After commenting on the performance of Q-learning algorithm, we implement the deep reinforcement learning method for the same problem. Finally we present the results for both approaches and make a comparison between them.

1.2. Deep Q-Learning with real-time validation

The main idea in the reinforcement learning (RL) and dynamic programming (DP) is to control an agent or a process while interacting with an environment using the observing states and rewards received from the environment (Busoniu et al., 2017). DP and RL are algorithmic methods for solving decision-making problem to achieve the desired goal throughout the interaction with the world. DP methods require the model of the system's behavior, whereas RL is a model-free approach which improves the produced policy while interacting with the environment. RL uses Markov Decision Process (MDP) (Busoniu et al., 2017) to mathematically formalize discrete stochastic environment. As MDP works in discrete time, states and actions in RL are usually discrete that leads to a sequential decision-making problem. Rewards also provide an informative metric of the agent's performance, and the goal is to maximize the accumulated long-term return over the track of the interaction with the environment. Although recent advances in RL (Lillicrap et al., 2015) enables us to apply to continuous problems, here we leverage the discrete MDP to formalize the decision-making problem in autonomous driving and train RL algorithms to achieve a favorable performance.

Q-learning evaluates how good taking an action might be at a particular state through learning the action-value function $Q(s,a)$. In Q-learning a memory table $Q[s,a]$ is built to store the Q-values for all the possible combinations of states and actions. By sampling an action from the current state, the reward R and the new states are found out, by which the next action a that has the maximum $Q(s',a')$ from the memory table is taken. Taking an action in a particular state has a $Q-value$ which is depicted in eq. 1.

$$Q(s,a) = R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \tag{1}$$

Where s' and a' are next state and action respectively. However if the combinations of state and actions are too large or states and actions are continuous, the memory and computation requirement for action-value function Q will be too high. To address this issue, Deep Q-Network (DQN) (Mnih et al., 2015) is utilized that approximates the action-value function $Q(s,a)$.

In this work, as shown in Algorithm 1, (Mnih et al., 2015; Alizadeh et al., 2019), two networks, θ^- and θ are created and trained, one for retrieving Q values and one including all updates in the training. Ultimately we synchronize θ^- and θ to fix the $Q-value$ targets temporarily, so it keeps the target funciotn from changing abruptly. The loss will be calculated as in eq. 2.

$$L_i(\theta_i) = \mathop{\mathbf{E}}_{s,a,s',r\sim D}\left(r+\gamma \underset{a'}{max} Q(s',a';\theta_i^- - Q(s,a;\theta_i))\right)^2 \tag{2}$$

where the transitions (s,a,s',r) are retrieved from the experience replay D. Experience replay is used as a buffer, which we sample a mini-batch of samples from it to train the deep neural network. Since we are randomly sampling from the replay buffer, the data is closer to i.i.d and more independent of each other that in turn makes the training more stable. In addition, $\mathbf{E}$ indicates the expectation over the probability distribution.

By utilizing the experience replay and target network, the input and output of the model turns to be more stable to train and the network behaves more like a supervised learning algorithm.

We also appended the real-time validation phase to traditional DQN in order to record the best trained model during the training. For this purpose, we define two periods, by which validation phase flag is activated and lates network weights are being recorded and exploited during validation. Depending on which period, the agent's performance is evaluated for a number of episodes and the achieved mean reward is compared to the latest maximum value. This enables the agent to record the best trained model by validating on unseen scenarios. Defining two various periods with different number of episodes helps the training to be faster and record more generalized model at the same time.

Algorithm 1 Deep Q-Network (DQN) with experience replay and real-time validation

Input: Initialize:
replay memory D to capacity M
action-value function Q with random weights θ
target action-value function $\hat{Q}$ with wights $\theta^- = \theta$
Output: Q^*
initialize sequence $s_1 = \{x_1\}$
initialize preprocesses sequence $\phi_1 = \phi(s_1)$
for *t = 1, ..., T* **do**
  $a_t = \begin{cases} \text{random action} & \text{, with probability } \epsilon \\ arg\max_a Q(\phi(s_t), a; \theta) & \text{, otherwise} \end{cases}$
  r_t, x_{t+1}: apply(a_t)
  $s_{t+1} = s_t, a_t, x_{t+1}$
  $\phi_{t+1} = \phi(s_{t+1})$
  $D \leftarrow (\phi_t, a_t, r_t, \phi_{t+1})$
  $(\phi_j, a_j, r_j, \phi_{j+1}) \leftarrow random(D)$
  $y_j = r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-)$
  preform a gradient descent step on $\frac{\partial\left(y_j - Q(\phi_j, a_j, ;\theta)\right)}{\partial\theta}$
  if *validation_phase* **then**
    fix network parameters θ
    for *t' = 1, ...,val_episodes* **do**
      take greedy actions: $a = \max Q(\phi(s_{t'}), a; \theta)$
      record episode rewards
    if *mean validation reward is increased* **then**
      $\theta_{backup} \leftarrow \theta$
  every C steps; $\hat{Q} = Q$

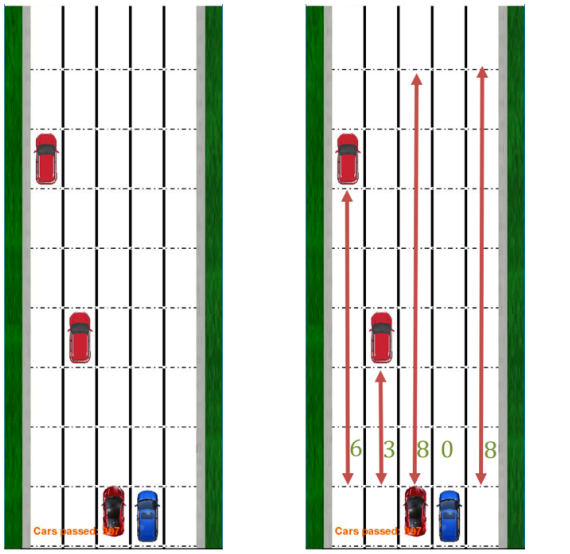
In order to improve the DQN performance, we also implemented Double DQN (DDQN) algorithm (Van Hasselt et al., 2016) which uses two $Q-$networks in order to deal with the overoptimistic value estimations. Target value estimation in Double Q-learning is performed using eq. 3

$$Q(s,a)^{DoubleQ} = R(s,a,s') + \gamma Q(s', argmax Q(s',a,\theta); \theta') \tag{3}$$

where, θ and θ' represent different $Q-$network parameters, one to determine the greedy policy and the other to decide on its value.

2. DeepCars Simulation Environment [2]

As discussed before, we are planning to map a 3-dimensional world into a much simpler 2D simulation environment and form the occupancy grid of the surrounding actors around the ego vehicle. To make this possible, we decided to design our own environment using the pygame (a free and open source python programming language library) to build our gaming environment. The source code is available in our GitHub repository (see footnote). We called the environment DeepCars and a screen-shot from the game screen is shown in Fig.3.a.



a) DeepCars simulation b) State representation in classical RL

Figure 3. A screen-shot from the DeepCars 2D environment and state representation

The game receives high-level control commands as the input vector and gives the game state and reward as the output. In order to be able to formalize the problem in MDP to be used in RL algorithm, the states and actions are discrete. In fact the action space consist of three actions:

$$\mathcal{A} = \{left \quad stay \quad right\} \tag{4}$$

switch to left/right lane or stay in the same lane. The observed Markov state for each RL is explained in corresponding sections.

3. Results

We have implemented a tabular Q-learning, DQN, and Double-DQN (DDQN) algorithms in DeepCars and compared the results. We will see that DDQN will demonstrate better results in terms of the performance-training speed trade-off in this problem.

[2] https://github.com/MajidMoghadam2006/gym-deepcars

3.1. Tabular Reinforcement Learning on DeepCars

As a starting point, we are going to implement the tabular Q-learning algorithm (Fig.4) for the control of the agent in DeepCars environment. The observed Markov state for the classical RL is defined as the following:

$$\mathcal{S} = \begin{bmatrix} \text{ego\_lane\_ID} & x_0 & x_1 & \dots & x_n \end{bmatrix} \tag{5}$$

in which the first element is the ego lane ID and x_i indicates the distance to the closest car in lane i. Note that the lane number and grid numbers start from 0. For instance the state vector for the Fig.3.b is $S = \begin{bmatrix} 2 & 6 & 3 & 8 & 0 & 8 \end{bmatrix}$. Also note that the line-of-sight of the vehicle is 9 (from 0 to 8). This means that the furthest car that the agent can percept can be 9 grids (number of rectangles) away. This constrain in fact simulates the sensor range that are implemented on the vehicle.

ALGORITHM Q-learning with ε-greedy exploration.

Input: discount factor γ,
exploration schedule $\{\varepsilon_k\}_{k=0}^{\infty}$, learning rate schedule $\{\alpha_k\}_{k=0}^{\infty}$

1: initialize Q-function, e.g., $Q_0 \leftarrow 0$
2: measure initial state x_0
3: **for** every time step $k = 0, 1, 2, \dots$ **do**
4: $u_k \leftarrow \begin{cases} u \in \arg\max_{\bar{u}} Q_k(x_k, \bar{u}) & \text{with probability } 1 - \varepsilon_k \text{ (exploit)} \\ \text{a uniformly random action in } U & \text{with probability } \varepsilon_k \text{ (explore)} \end{cases}$
5: apply u_k, measure next state x_{k+1} and reward r_{k+1}
6: $Q_{k+1}(x_k, u_k) \leftarrow Q_k(x_k, u_k) + \alpha_k[r_{k+1} + \gamma \max_{u'} Q_k(x_{k+1}, u') - Q_k(x_k, u_k)]$
7: **end for**

Figure 4. Tabular Q-learning algorithm (Busoniu et al., 2017) where x and u indicate the observed state and input respectively

The main objective is to train the agent to avoid making collisions with other vehicles in the environment. Thus, we define a simple reward function

$$\rho(s, a, s') = \begin{cases} +1 & s' \neq s_T \\ -1 & s' = s_T \end{cases} \tag{6}$$

Where s_T indicates the terminal state that the agent makes a collision. For the hyper-parameter tuning, we performed a simple grid search in the parameter space and found the optimal setting:

- discount factor: $\gamma = 0.9$
- learning rate: $\alpha = 0.1$
- exploration: $\epsilon = 0.2$
- exploitation: $1 - \epsilon = 0.8$

We have trained the agent for 50000 steps. The resulting Q-table were recorded in order to be used in the evaluation phase. We evaluated the agent performance for 100'000 steps and recorded the results. In order to comment on

the performance of the agent we define the accuracy as the consistent comparison metric between algorithms:

$$\text{Accuracy} = \frac{\text{\# of passed cars}}{\text{\# of all cars (passed + collided)}} \times 100 \quad (7)$$

The results were promising though imperfect. In fact for the training and evaluation phases, the results were as follows:

- Training: Accuracy = 98.28%
- Evaluation: Accuracy = 99.14%

This shows that after 50000 frames the agent has learned to control the vehicle by avoiding other vehicles. However, this is not the ideal performance, because there are a number of collisions in the test set which are not appealing. In fact our aim is to design an agent that can avoid collisions for all cases. If we take a look at the recorded Q-table after the training is finished in Fig.5 we can see that most of the rows are still zeros which indicates a sparse Q-table that is expected in tabular Q-learning. This situation arises because not all of the state-action combinations were experienced in the training phase. And, as the agent is not using any function approximator, the similar situations cannot be approximated and collisions are unavoidable. We expect to overcome this problem by approximating Q-values using a deep neural network architecture which indicates the DQN algorithm. Note that the evaluation performance is better than training because in evaluation phase we use the best trained model. In addition, the actions are greedy in evaluation phase in contrast with $\epsilon-$greedy strategy in training phase.

Row #	State vector	Left	Stay	Right
51242	[1, 6, 2, 6]	0.000000	0.000000	0.000000
51243	[1, 6, 2, 7]	0.000000	0.000000	0.000000
51244	[1, 6, 2, 8]	0.000000	0.000000	0.000000
51245	[1, 6, 3, 0]	7.843658	8.111676	9.612881
51246	[1, 6, 3, 1]	0.000000	0.000000	0.000000
51247	[1, 6, 3, 2]	0.000000	0.000000	0.000000
51248	[1, 6, 3, 3]	0.000000	0.000000	0.000000

Figure 5. A crop of sparse Q-table in tabular RL (# of lanes: 3). Red rectangle indicates the greedy optimal action where $s = \begin{bmatrix}1 & 6 & 3 & 0\end{bmatrix}$

3.2. Deep Reinforcement Learning on DeepCars[3]

As discussed in previous section, the tabular Q-learning approach lacks the generalization property. In addition, the course of dimensionality is another problem while using Q-learning techniques. In our case, the number of Markov states ascend exponentially as the number of lanes increases. In addition, we are planning to feed the occupancy grid of the environment into the agent which makes the use of classical RL impossible for this case. This leads us to use the DQN and DDQN algorithms. The source code for DQN and DDQN algorithms are provided in our GitHub repository (see footnote). Here, we modified the original DQN algorithm and added the real-time validation phase. This helps us to perform the evaluation of the most recent model during the training and select the best trained model by far. This is possible by evaluating the latest model on a number of game episodes, say 100, and recording the agent's performance. Finally, we record the model that had the best performance at the end of training.

Note that in DQN the observed MDP state is the occupancy grid while in tabular Q-learning, in order to reduce the state space dimension, we defined the game state as in eq. 5. The occupancy grid incorporates the actors' position in a simple matrix with the same size as the environment grid (Fig.3.a). Empty grids are filled with zeros, and ones indicate the occupied grid with an actor. The flattened version of this matrix forms the state vector. Finally, we fuse the ego lane number as a binary code of the lane ID to create the MDP state. Including ego lane ID as a binary code, enables us to have consistent binary values in the state vector, though it increases the observation space dimension.

In addition, in order to have a smooth graph for reward values we keep track of the mean accumulated rewards every 100 episodes as the performance metric. We implemented our algorithm in a PC equipped with CPU: Intel Core i9 2.60GHz and GPU: GeForce GTX 1080 Ti. We trained the agent for 500'000 steps for three different architectures for the neural network; shallow, medium, and deep. Each differs in the number of layers of the neural network. We have used $\{32\}$ for shallow, $\{32, 64, 32\}$ for medium, and $\{64, 128, 128, 64\}$ for deep network architectures as the number of layers. The rest of the hyper-parameters were equal for all cases. The performance comparison of these networks are summarized in Fig.6. Obviously, the shallow

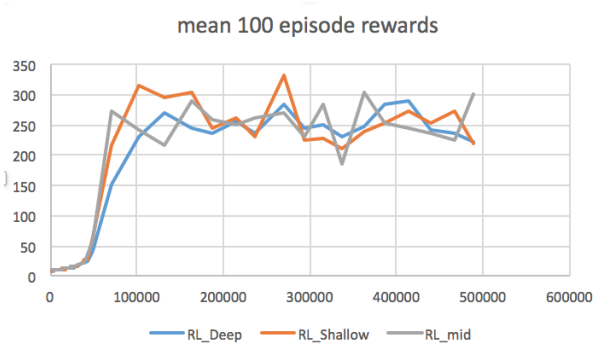


Figure 6. DQN performance for three network architectures

[3] https://github.com/MajidMoghadam2006/deepcars-reinforcement-learning

network has shown a better performance comparing with two others. But the best comparison would be made in th evaluation phase where we save the trained models and evaluate them in consistent sets of evaluation episodes and keep track of the number of collisions. As shallow network promised a better performance, we use the same architecture to compare DQN and DDQN. However, we adjust the number of layers in network at the same time to search for the best option. The performance of these configurations in training phase are represented in Fig.7. Obviously, the deep version of DDQN achieved a higher

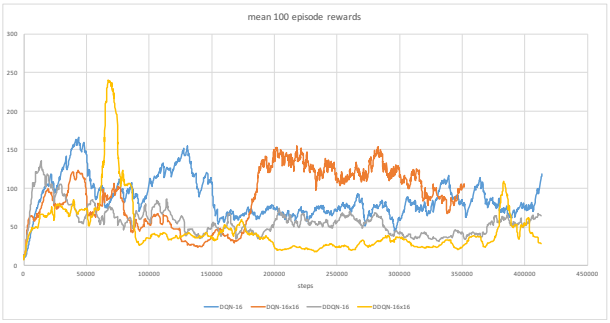


Figure 7. DQN vs. DDQN with different network architectures. Here 16x16 indicates two dense layers with 16 neurons at each.

reward peak value comparing with others, while shallow DDQN converged much faster than its rivals. DQN has shown a mediocre performance between these agents. Following the real-time validation policy in algorithm 1, we recorded the best model in all training phases and evaluated their performance for 100'000 steps where all of the agents achieved 100% accuracy (eq. 7). This means that all agents are thoroughly trained and succeed to perform the evaluation phase without any collisions. This accomplishment ascertains the generalization performance of the DQN comparing with tabular Q-learning in previous section. Among deep approaches, DDQN-16 has shown the best performance-speed trade-off. Intuitively, the advanced RL algorithm (DDQN) is much faster than its predecessor (DQN) which was expected. Also, a shallow network with one hidden layer and 16 neurons seems a reasonable choice for this setting where the size of the occupancy grid as the input is $8 \times 5 = 40$. A rule of thumb to choose the number of neurons in the hidden layer is to select a number between the input and output vector sizes. These facts may justify the choice of DDQN-16 agent as the best performance-speed RL algorithm for the DeepCars simulation environment.

Achieving 100% evaluation accuracy for the unseen scenario in training phase strengthens the assertion that it's possible to decompose the autonomous driving problem into a hierarchical architecture and leverage the AI power to solve each layer separately with an admissible reliability score.

4. Conclusion

In this study, we addressed the problem of autonomous driving in the highway scenario. We considered the hierarchical architecture of the ADAS systems and focused on the decision making layer, where the sequential high-level decision command are being made using a deep reinforcement learning algorithm. The agent receives the occupancy grid of the environment as the state representation and produces lane change commands in order to avoid making collisions with other vehicles. The environment is stochastic in terms of the traffic density and lane position of the other actors which challenges the agent's performance and at the same time, imitates the fusion layer output that may occur in actual test scenarios. We have shown that it's possible to leverage the provided multi-layer architecture to generate high-level commands using DRL with an acceptable reliability score.

Future Work

We are planning to implement the multi-layer architecture in Fig.2 and tackle each layer separately using the same approach that we used in this study. We will exploit DeepCars as the 2D environmental model of the ego surrounding. We summarized this scheme in Fig.1. We are also planning to use Unreal Engine powered simulations like CARLA (Dosovitskiy et al., 2017) or Microsoft AirSim (Shah et al., 2018) to generate the occupancy grid of the ego surrounding from the ground truth information that we receive from the simulator. This way, we may bypass the fusion layer in ADAS (Fig.2) and study the presented hierarchical approach (Fig.1) performance in more realistic and complex situations. In addition, we will compare the original DQN with its most recent extensions like dueling double DQN (Wang et al., 2015), prioritized experience replay (Schaul et al., 2015), and rainbow agent (Hessel et al., 2018) which is a combination of most recent improvements of the RL in recent years. We will also perform a hyper-parameter optimization to search for the best parameter setting in the parameter space.

References

Alizadeh, A., Moghadam, M., Bicer, Y., Ure, N. K., Yavas, U., and Kurtulus, C. Tactical lane changing with deep reinforcement learning in dynamic and uncertain traffic scenarios. In *22nd Intelligent Transportation Systems Conference (ITSC2019-submitted)*, 2019.

Bicer, Y., Moghadam, M., Sahin, C., Eroglu, B., and Üre, N. K. Vision-based uav guidance for autonomous landing with deep neural networks. In *AIAA Scitech 2019 Forum*, pp. 0140, 2019.

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

Broggi, A., Bertozzi, M., Fascioli, A., Bianco, C. G. L., and Piazzi, A. The argo autonomous vehicles vision and control systems. *International Journal of Intelligent Control and Systems*, 3(4):409–441, 1999.

Busoniu, L., Babuska, R., De Schutter, B., and Ernst, D. *Reinforcement learning and dynamic programming using function approximators*. CRC press, 2017.

Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., and Koltun, V. Carla: An open urban driving simulator. *arXiv preprint arXiv:1711.03938*, 2017.

Falcone, P., Borrelli, F., Asgari, J., Tseng, H. E., and Hrovat, D. Predictive active steering control for autonomous vehicle systems. *IEEE Transactions on control systems technology*, 15(3):566–580, 2007.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

Koutník, J., Cuccu, G., Schmidhuber, J., and Gomez, F. Evolving large-scale neural networks for vision-based torcs. 2013.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529, 2015.

Moghadam, M. and Caliskan, F. Actuator and sensor fault detection and diagnosis of quadrotor based on two-stage kalman filter. In *2015 5th Australian Control Conference (AUCC)*, pp. 182–187. IEEE, 2015.

Moghadam, M., Ure, N. K., and Inalhan, G. Autonomous execution of aircraft supermaneuvers with switching nonlinear backstepping control. In *2018 AIAA Guidance, Navigation, and Control Conference*, pp. 1594, 2018.

Perot, E., Jaritz, M., Toromanoff, M., and De Charette, R. End-to-end driving in a realistic racing game with deep reinforcement learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 3–4, 2017.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

Shah, S., Dey, D., Lovett, C., and Kapoor, A. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and service robotics*, pp. 621–635. Springer, 2018.

Slotine, J.-J. E., Li, W., et al. *Applied nonlinear control*, volume 199. Prentice hall Englewood Cliffs, NJ, 1991.

Specht, D. F. A general regression neural network. *IEEE transactions on neural networks*, 2(6):568–576, 1991.

Van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., and De Freitas, N. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.