Autonomous Driving
with DL and RL

Abhishek Naik Mohan Bhambhani
IIT Madras

Abstract

Deep Learning in combination with Reinforcement Learning is today being considered as a strong contender to achieve General Artificial Intelligence. Motivated by the recent success of Google DeepMind in performing a wide variety of tasks using Deep Reinforcement Learning, we propose an end-to-end 2-stage framework for autonomous driving (in the presence of traffic) in The Open Racing Car Simulator (TORCS). First, we train a convolutional neural network to predict the features that appropriately map the environment. With these features, we then train a reinforcement-learning agent using the Deep Deterministic Policy Gradient Algorithm (DDPG), which is one of the state-of-the-art techniques in continuous (real-valued) and high dimensional state and action spaces, to take the driving decisions based on a carefully-designed reward/penalty system.

1. Introduction

Driving a car broadly has the following 2 subproblems:

1. Sensing the environment
   The car/driver needs to first create a reliable picture of the immediate relevant surroundings of the car - the lane markings, nearby cars, distance to them, jaywalking pedestrians, traffic signals, detours.

2. Making driving decisions
   Once it has all the required information, it has to fuse all this data from multiple sensors together to make driving decisions about the best path to take, the amount of braking/acceleration, the steering angle, all in real-time.

One of the main components of the state-of-the-art autonomous driving techniques today involve extracting information about the surroundings from the 3D-point cloud obtained from Velodyne LiDARs [3][9]. The problems with this approach is that only a small portion of the detected objects are indeed relevant to the driving decision to be made, and hence a lot of this information is redundant. While leading to impressive solutions, this approach uses manually labeled maps and GPS localization, suffers from the fact that up-to-date sub-meter accurate maps will most likely be impossible to acquire in practice. Moreover, GPS signals are not always reliable - in presence of buildings, walls, their localization can becomes imprecise to the order of meters, which is unacceptable for safety-critical applications like Autonomous Vehicles.

We use the approach in [4] to first identify and predict a host of features that are believed to be the most useful in making regular driving decisions. Following the collection of labelled data for the same by tapping into the game engine of the popular open source racing game TORCS in variety of driving terrains and traffic conditions, we train a couple of CNN architectures to predict these features, given the drivers visual perspective. This also includes the rear-view mirror perspective as the input. With these features, we then train a reinforcement-learning agent using the Deep Deterministic Policy Gradient Algorithm (DDPG)[10], which is one of the state-of-the-art techniques in continuous (real-valued) and high dimensional state and action spaces, to take the driving decisions based on a carefully-designed reward/penalty system.

2. Related Work

With the rapid advancements in their fields, the combination of modern Reinforcement Learning and Deep Learning approaches holds the promise of making significant progress on challenging applications requiring both rich perception and policy-selection. Autonomous Driving is a great example of one such challenging application, but to the best of our knowledge, there hasn't been much work done in explicitly using a combination of Deep Learning and Reinforcement Learning techniques in building self-driving cars. In the rest of the section, we shall briefly go through some of the notable works in both of the paradigms.

2.1. Deep Learning methods

There are currently 3 major paradigms[4] for approaching these problems:

2.1.1 Mediated Perception

This involves a (very expensive) LiDAR, in combination with a RADAR and normal color cameras, which map the entire surrounding 3D surface [1]- detecting and creating bounding boxes around nearby cars, pedestrians, and then, among other things, estimating the distances to them. After solving these subproblems, which are still an open area of research in CV, decisions about driving are taken[4]. Some driver behaviour is also pre-programmed [1].

2.1.2 Behavioural Reflex

This is an end-to-end deep learning approach [2], which uses feeds from front-facing cameras, and labelled metrics like steering angle and velocity to train their networks, doing away with a lot of the redundant mapping in the approach above. This results in a decent performance in highway-like driving conditions, but steering angle and velocity are not enough in even slightly complex driving scenarios, wherein different decision need to be made for the seemingly very similar input.

2.1.3 Direct Perception

This is somewhere between the above two. It uses the front-camera feed to find metrics like distance of the agent from the lane-markers, and to the cars in the current and adjacent lanes from a CNN, and then uses heuristics like - if car approaching car in front, and if left lane exists and is free, switch to that lane, else slow down [4]. Intuitively, this seems to be the closest to how we humans drive.

When this paradigm was first introduced [4], the authors trained a CNN (using the AlexNet architecture) to predict 13 features that they felt were critical for making driving decisions, and then designed hand-crafted if-else based rules like if we are approaching a car in the same lane and the right lane exists and is free, switch to the right lane. They do not use the rear-view mirror perspective, which is a definite point for improvement, along with certain other features that can be used. Also, following hard-coded rules is not how driving is done. Just like us humans do not learn cycling by following a definite set of rules, but by trial-and-error, the driving decisions should be learnt using reinforcement learning.

2.2. Reinforcement Learning methods

There is an international Simulated Car Racing (hereafter referred as SCR) Championship [11] that takes place since 2004, now held at major conferences in the field of Evolutionary Computation and in the field of Computational Intelligence and Games, where competitors are provided a set of features describing the state of the car in real-time for the game of TORCS (The Open Racing Car Simulator) [18]. With these features like speed, angle of car from the tangent to the road, distance to the edge of the road at various angles, etc, competitors design exhaustive, intricate heuristics (like determining the optimum speed on a particular road segment using laws of friction) to drive the cars. Link

2.2.1 Neuroevolution

Koutnik et al. [7] were one of the firsts to approach the task of driving a car around a track autonomously, using vision - i.e. only the driver's perspective of the road. They used neuroevolution to evolve neural network controllers instead of training them via, say, backpropogation. to get rid of noisy, non-stationary gradient information to perform the reinforcement learning task of temporally-delayed credit assignment.

Some approaches have tried discretizing the continuous action space into bins. Instead of using neural networks for function approximation, [15] perform tile coding to discretize the extremely large state space to perform Q-learning. Their reward function had components of distance covered by the agent, 'comfort' of the driver based on the net force on the car, and the distance to the center of the road. With this formulation, they conclude from their experiments that being able to navigate a straightaway and a hairpin turn successfully is not necessarily safe driving.

Yet another approach [12] tries to combine CNN features from the raw pixel input and the RNN features obtained from low-level features like the cars speed, angle from the center of the track, to predict the Q-values. They then compare this hybrid DQN model with a discrete agent, along with baselines of a greedy and a hand-crafted controller, but the best they can achieve is just driving around the empty race track.

2.2.2 Actor-Critic approaches

- Deep Deterministic Policy Gradient (DDPG)
  DeepMind, after their seminal work in achieving human-level performance in ATARI games using Deep Q-Networks (DQNs) [14], adapted the success of Deep Q-learning to the continuous action domain in [10], where they use an actor-critic, model-free algorithm based on the deterministic policy gradient that solves

more than 20 simulated physics task, along with car driving in TORCS. They do this both the the SCR features and the raw-pixel input, and learn reasonable policies that makes an agent complete a circuit around an empty track. Link

- Asynchronous Advantage Actor-Critic (A3C)
  Instead of the idea of experience replay as used in DQN[14] and DDPG[10], they asynchronously execute multiple agents in parallel on a multi-core processor (not GPUs) to reduce non-stationarity and decorrelated updates. The agent takes the RBG frames as input, gets a reward proportional to the velocity along the center of the track. The policies learnt can be viewed here.

An important point to note here is that except A3C, all the approaches use the low-dimensional SCR-features as input, and learn policies for driving on empty tracks. A3C introduces some bots, but to the best of our knowledge, there is no implementation which explicitly learns to drive in traffic-like conditions - driving behind cars in the same lane and overtaking when possible. That too with features (other than the SCR features, as described in the following 'Dataset' section) obtained from the raw driver's perspective of the road.

3. Implementation

3.1. Direct Perception

The input to our convolutional network is a sequence of 210 × 280 RGB image of the first-person driving view in TORCS. The image also contains view of the rear-view mirror of the car of size 35 × 140. An sample view of the frame can be seen in the Figure 1. These images are mapped to 43 features out of which 2 are categorical. The complete list of features as follows:

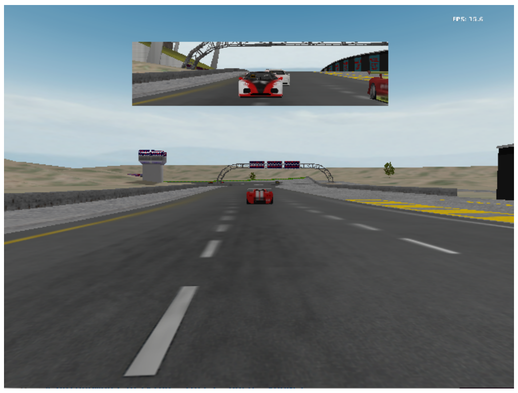
Figure 1. A sample first-person driving view in TORCS

1. $angle$: angle between the cars heading and the tangent of the road
   In lane system, when driving in the lane.
2. $toMarking\_LL$[4]: distance to the left lane marking of the left lane
3. $toMarking\_ML$[4]: distance to the left lane marking of the current lane
4. $toMarking\_MR$[4]: distance to the right lane marking of the current lane
5. $toMarking\_RR$[4]: distance to the right lane marking of the right lane
6. $dist\_LL$[4]: distance to the preceding car in the left lane
7. $dist\_MM$[4]: distance to the preceding car in the current lane
8. $dist\_RR$[4]: distance to the preceding car in the right lane.
9. $dist\_LL\_rear$: distance to the succeeding car in the left lane
10. $dist\_MM\_rear$: distance to the succeeding car in the current lane
11. $dist\_RR\_rear$: distance to the succeeding car in the right lane.
    on marking system, when driving on the lane marking:
12. $toMarking\_L$[4]: distance to the left lane marking.
13. $toMarking\_M$[4]: distance to the central lane marking
14. $toMarking\_R$[4]: distance to the right lane marking
15. $dist\_L$[4]: distance to the preceding car in the left lane
16. $dist\_R$[4]: distance to the preceding car in the right lane
17. $dist\_L\_rear$: distance to the succeeding car in the left lane
18. $dist\_R\_rear$: distance to the succeeding car in the right lane
    Other features:
19. $distToVisibleTurn$: Distance to turn ahead.
20. $visibleTurnRadius$: Radius of the turn ahead.
21. $visibleTypeRoad$: Type of the turn, i.e., left or right.
22. $currRadius$: Radius of the road at current segment.
23. $typeRoad$: Type of the track at the current segment, i.e. straight, left turn or right turn.
24. $toMiddle$: Normalised Distance to the center of the track.

25-43. 19 SCR features[11]: 19 samples of the space in front of the car taken at various angles ranging from $-90°$to $90°$w.r.t the car axis. A samples contains the distance between the track edge and the car axis with the corresponding shift.

3.2. Controller

The next step is to use these features and the following features which are assumed to be directly available from the car, in order to make driving decisions. The controller will then output the action to be taken to the game engine at a frequency of 10Hz.

---

1. $speed$[11]: Speed of the car.
2. $speedY$[11]: Speed of the car along the Y axis of the car.
3. $speedZ$[11]: Speed of the car along the Z axis of the car.
4. rpm[11]: Number of rotations per minute of the car engine.
5. $damage$[11]: Current damage of the car.

6-9. $wheelSpinVel$[11]: rotations speeds of the 4 wheels.

---

4. Technical Approach

4.1. Mapping from an image to features

4.1.1 VGGNet

We use VGGnet with 16 layers architecture followed by 4 fully connected layers with dimensions 4096-1024-1024-47. For 2 categorical features with 3 classes each, cross entropy is taken as the loss function, and the Euclidean loss in minimized for the rest.

4.1.2 Modified VGGNet

The previous model has more than 80M parameters. Most of them are in the first fully connected layer. So we propose to use 3-D convolution instead of the first fully connected layer. The kernel size of the 3-D convolution was $4 \times 5 \times 256$. Stride for the 3rd dimension was set as 256. Number of output channels were set as 1024. Each channel has 4 outputs. Number of parameters get reduced from $4 \times 5 \times 1024 \times 4096$ to $4 \times 5 \times 256 \times 1024$, i.e. by 16 times. So after this we could increase the number of neurons in the second fully connected layer to 2048.

4.2. Mapping from the features to action

We feed these 52 continuous features to the Actor network, which outputs a policy for the 3 continuous actions - steering, acceleration, braking. The policy is updated according to the direction of gradient specified by the critic's parameterised value function. The critic's parameters are updated by minimising the loss:

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$$

The actor is then updated using the sampled policy gradient:

$$\nabla_{\theta^\mu} J = \nabla_a Q(s, a, |\theta^Q)|_{a=\mu(s)} \nabla_\theta \mu(s|\theta^\mu)$$

This was proven by Silver et al. in [17] as the deterministic policy gradient theorem.
These are the 2 governing equations of the DDPG algorithm. Readers can refer to [10] for the complete algorithm and architectural details.
The objective is to maximise the discounted return at every point, and the reward function was roughly set proportional to the velocity of the agent along the center of the track. The exact formulation and modifications made for learning to drive in traffic are specified in the 'Experiments' section.

5. Dataset creation

The TORCS game source code was patched with Chen et al.'s [4] code and the SCR code [11] to help generate additional features using the TORCS tutorial. The 'berniw' bot was modified to collect all these features in automated runs over different tracks of varying orientations and illuminations.

To explore more of the state space and avoid restricting the data to the space where the optimal policy lies, noise was introduced in the angle output of the bot's action. This was the Type-1 noise. But since this noise may get cancelled out over the time, another Type-2 noise was added where the sampled noise was kept same for next 6 time-steps. Noise was gaussian with 0 mean and variance as 2 in Type-1 and 1.5 in Type-2.

Data was collected on 4 tracks modified by Chen et al. [4], namely chenyi-E track 6, chenyi-CG Speedway, chenyi-CG Track 2 and chenyi-Wheel 1. On each track data for 60 laps was collected, where 20 were without noise, 20 with Type-1 noise and 20 with Type-2 noise. 12 custom slow cars from [4] and 20 existing racing cars by 'tita' and 'illiaw' were added as traffic. In total, $221,114$ frames with their corresponding features were collected, equivalent to almost 6 hours of gameplay data. Chen et al. [4]'s data collection scripts were used to store the data in a 'LevelDB' database, which was later migrated to 'h5py' for usage on AWS.

6. Experiments

6.1. Phase I - CNN

The convolutional neural network was trained on P2 instance of Amazon AWS, which uses 1 NVIDIA K80 GPU. Loss function for the training was L2-error for all except the categorical features. For categorical features cross entropy was used. All the weights were initialised

Table 1. Mean Absolute Error on the 13 baseline features

Method	angle	to_LL	to_ML	to_MR	to_RR	dist_LL	dist_MM	dist_RR	to_L	to_M	to_R	dist_L	dist_R
Baseline[4]	0.033	0.188	0.155	0.159	0.183	5.085	4.738	7.983	0.316	0.308	0.294	8.784	10.740
VGGNet	0.028	0.242	0.241	0.235	0.236	3.349	2.446	3.87	0.245	0.137	0.249	3.06	3.323
Modified	0.024	0.198	0.196	0.195	0.198	3.411	2.935	3.133	0.219	0.114	0.212	2.86	3.39

Table 2. Mean Absolute Error on the other features

Method	dist_LL_r	dist_MM_r	dist_RR_r	dist_L_r	dist_R_r	toMiddle	DVT	VTR	CR
VGGnet	1.962	1.933	2.270	1.947	2.009	0.032	3.66	15.9	16.1
Modified	1.660	1.742	1.989	1.648	1.810	0.027	3.17	13.8	13.5

Table 3. Mean Absolute Error on SCR features

Method	SCR0	SCR5	SCR10	SCR15	SCR20	SCR30	SCR45	SCR60	SCR75	SCR90
VGGnet	4.63	4.70	4.18	3.54	2.87	2.30	1.81	1.49	1.52	1.60
Modified	4.04	4.26	3.80	3.19	2.65	1.90	1.43	1.25	1.19	1.37

Table 4. Mean Absolute Error on SCR features

Method	SCRn5	SCRn10	SCRn15	SCRn20	SCRn30	SCRn45	SCRn60	SCRn75	SCRn90
VGGnet	4.72	3.94	3.55	2.77	2.07	1.53	1.62	1.57	1.63
Modified	4.15	3.66	3.17	2.70	1.73	1.26	1.38	1.27	1.35

using Xavier initialisation. Learning rate was set as $1e-3$. Batch size was set as 64. Initially the learning was very slow as the ranges were different for all the features. Data was preprocessed before training by normalising all the output features to the range of [0,1]. All the convolutional layers and fully connected were followed by batch normalisation layer for regularisation and ReLU layer for non-linearity. Adam optimizer was used with $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1e-08$. Learning rate was decayed by 0.9 after every 500 steps.

8 epochs were run for both the architectures. Training took almost 18 hours for the first architecture. Training in the second architecture was a bit slower as compared to the first as it took almost 20 hours. But, performance on the second was slightly better than the first. The L1-error obtained for all the features is shown in the Tables 1, 2, 3, 4.

It can be seen the our model outperforms the baseline with a significant margin for all except the 4 distances to lane marking in the lane system. Decent performance is obtained for other additional features also. It is also evident that the modified VGGnet architecture performs better than the original VGGnet architecture.

6.2. Phase II - DDPG

Due to the lack of a machine with the patched TORCS simulator and a large enough RAM to store the entire 6GB of the trained VGGNet weights, the following DDPG experiments were performed by interacting directly with the game engine. Some noise was added to the 52 features generated in order to roughly simulate the imperfect performance of the trained CNN.

Github user yanpanlau's implementation of DDPG [8] on gym_torcs game engine is used as the starting point. The following things are done over and above the theoretical update equations, to ensure stability and fast convergence:

1. Experience Replay Buffer : As suggested by Minh et al. in [13], building a dataset of transitions from the agents own experience, and using a mini-batch of these while training instead of the most recent transition is instrumental in reducing the correlations between the updates.
2. Separate Target Networks : Again, as suggested by Minh et al. in [14], maintaining separate networks for the target values is necessary to handle the problem on non-stationary targets.
3. Episode restarts : The current training episode would be restarted if the agent goes out of the track, is making no progress, or has turned around. If we allow such episodes to run till $maxEpisodeLength$, then the Replay Buffer will get filled with useless transitions, hence solving down the training massively.
4. Exploration : Popular exploration techniques like ϵ-greedy cannot be used here with the 3 continuous actions, because it will result in useless combinations like when the value of brakes is more than that of acceleration, resulting in no movement. Instead, as suggested in [10], a Ornstein-Uhlenbeck process is used to generate temporally correlated exploration with inertia.
5. Other tweaks included switching off the video rendering of the game for faster training, and checking it only once in 10 episodes. And since TORCS is a racing simulator wherein cars start from a starting grid, to en-

sure that the agent actually gets opportunities to overtake, the other cars were given head starts of 500-700 steps in order to sufficiently spread out on the track before the agent starts driving.

As with humans, the agent learnt driving in 2 stages. At first, the DDPG agent is trained on empty roads, without traffic. This is done with a reward function as:

$$R_t = v_t \cos\theta - v_t \sin\theta - v_t|\text{trackPos}|$$

where R_t is the reward at time step t, v_t is the velocity of the car along its axis, θ is the absolute angle of the car's axis with the tangent to the road, and 'trackPos' is the normalized distance between the car and the track's axis - it is 0 when the car is on the axis, -1 and $+1$ when the car is on right and left edge of the track respectively.
Intuitively, this reward function encourages the agent to drive fast along the middle of the road.

6.2.1 Pre-training

With the above reward function, the agent first learns to drive on empty roads with no traffic. It is given a penalty of -200 for going out of the track. After about 200 episodes, the agent learns a decent policy of driving along the center of the road with a considerable speed.

6.2.2 Traffic training

Now that the agent has basic driving skills, some traffic is introduced in the form of bots driving in specified lanes with pre-determined velocities. 15 bots are introduced sequentially as learning progresses. In order to teach the agent to learn the driving behaviour of maintaining the lane speed and overtaking in case the opportunity arises, a couple of important things are done:

1. Stochastic braking is introduced - with a probability of 0.1, the agent is made to brake lightly for exploring braking. Till now, the agent just lets go of the accelerator to reduce it's speed, because explicit braking decreases the reward it gets. With stochastic braking, the agent learns to use the brakes.

2. Encouraging overtaking - the reward function is tweaked to reduce the penalty of driving away from the center of the road by half ($v * |\text{trackPos}|/2$). This encourages the behaviour that driving away from the center of the road is not too bad, hence introducing the concept of overtaking. Reducing the penalty even further increases the risk of skidding off the road in case of sharp turns.

3. Enforcing safe-distances - a high penalty of -200 per time-step is given in case the agent comes too close to another car in any direction. This encourages the agent to maintain a 'safe-distance' with other cars in the traffic at all times.

After about 200-250 more episodes, the agent learns a decent policy of braking behind cars in the same lane and overtaking when the opportunity arises. This behaviour is further fine-tuned and generalised by letting the agent run over multiple training tracks and with different start-settings. The resulting policies can be visualized in the accompanying videos.

Implementation details : Since installing [4]'s patched TORCS game and further adding the SCR patch over it was proving to be too cumbersome on the AWS machines, all the DDPG training was performed on our laptops (sans an NVIDIA GPU) on a modest Intel Core i5-5200U CPU @ 2.20GHz 4, with 8GB RAM. With a maxEpisodeLength of 3500, the entire DDPG training lasted about 6 hours. Using TensorFlow, we trained multi-layer fully connected networks with 300 and 400(600) respective units in each hidden layer of the actor(critic) network. A replay buffer size of 100000 transitions was chosen, with a discount factor of $\gamma = 0.99$. The Adam optimizer was applied, with a batch size of 32 and learning rates of 0.0001 and 0.001 for the actor and critic respectively. The choices of the hyperparameters were adapted from [8]. The target networks were trained with gradually updated parameters.

To the best of our knowledge, this is the first work which trains an agent to drive in traffic-like conditions. Hence there is no explicit evaluation metric to benchmark it on. Using traditional driving metrics like maximum speed achieved, average reward received per lap, etc, do not make sense here because the objectives are no longer the same - one strives to cover the maximum distance in the shortest amount of time, while the other attempts to manoeuvre its way safely through an traffic environment.

7. Conclusions and Future Work

Our contribution in this project is three-fold:

1. Firstly, we've created a rich new dataset of a simulated driving environment with traffic, consisting of around 6 hours of driving annotated with 52 features that we believe are important for making driving decisions.

2. Secondly, we have successfully trained a Convolutional Neural Network that outperforms the baseline set by Chen et al. in [4] (they used just 13 of our 52 features).

3. Thirdly, we have successfully trained a DDPG agent to drive in traffic-like conditions, where it learns to overtake and maintain the lane speed.

Some areas that we've identified for immediate improvement:

1. Run an end-to-end 2-stage model, which takes in the raw pixels as input, does a forward-pass through the CNN to predict discriminative features, and then a forward pass through the DDPG network to make appropriate driving decisions. Even with our 'compressed' VGGNet implementation, this will require a powerful machine with a large amount of RAM to ensure a low latency between all these operations for a decent real-time play.
2. The policies learnt are seen to be a little wobbly in terms of the rapid tiny changes the agent makes to maximize the reward obtained. We propose regularizing the steering action with a mixture formulation:

$$a = \beta a_{new} + (1 - \beta) a_{old}$$

   where a_{new} and a_{old} are respectively the current and previous actions predicted by the DDPG network, and a is the action actually taken. Currently, we have $\beta = 1$. It would be interesting to see how this affects the training and the subsequent performance.
3. One of the most common problems drivers today face is the presence of blind-spots in cars. Cars in adjacent lanes behind you become 'invisible' in certain locations. In order to ensure some context, we propose using LSTM representations of a window of input frames to encode the state space as well, in addition to the Actor-network input.
4. Recently, planning by imitation techniques have been showing some promise [6][16]. Along these lines, we think that providing the agent with expert trajectories in the beginning, or even using interleaving as in [6] will help the training converge even faster. Some interesting preliminary work has been done in [5].

References

[1] The tech that powers google's self-driving cars. *https://www.national.co.uk/tech-powers-google-car/*. 2

[2] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016. 2

[3] J. Byun, B.-S. Seo, and J. Lee. Toward accurate road detection in challenging environments using 3d point clouds. *ETRI Journal*, 37(3):606–616, 2015. 1

[4] C. Chen, A. Seff, A. Kornhauser, and J. Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015. 1, 2, 3, 4, 5, 6

[5] J. Fischer, N. Falsted, M. Vielwerth, J. Togelius, and S. Risi. Monte-carlo tree search for simulated car racing. 7

[6] X. Guo, S. Singh, H. Lee, R. L. Lewis, and X. Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in Neural Information Processing Systems 27*, pages 3338–3346. Curran Associates, Inc., 2014. 7

[7] J. Koutník, G. Cuccu, J. Schmidhuber, and F. Gomez. Evolving large-scale neural networks for vision-based reinforcement learning. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1061–1068. ACM, 2013. 2

[8] B. Lau. Using keras and deep deterministic policy gradient to play torcs. *https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html*. 5, 6

[9] M. Li and Q. Li. Real-time road detection in 3d point clouds using four directions scan line gradient criterion. *Future*, 5, 2009. 1

[10] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015. 1, 2, 3, 4, 5

[11] D. Loiacono, L. Cardamone, and P. L. Lanzi. Simulated car racing championship: Competition software manual. *CoRR*, abs/1304.1672, 2013. 2, 3, 4

[12] A. N. Matt Vitelli. Carma: A deep reinforcement learning approach to autonomous driving. *Project*. 2

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013. 5

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. 2, 3, 5

[15] H. Y. Perez Angelica, Paul Quigley. Driving in torcs with a reinforcement learning agent. *Project*. 2

[16] S. Ross, G. J. Gordon, and D. Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *AISTATS*, volume 1, page 6, 2011. 7

[17] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 387–395, 2014. 4

[18] B. Wymann, C. Dimitrakakisy, A. Sumnery, and C. Guionneauz. Torcs: The open racing car simulator, 2015. 2