Autonomous Driving using Deep Deterministic Policy Gradients

Ankur Roy Chowdhury [*1] **Karthikeya S. Parunandi** [*2] **Santosh Shyamala Ramasubramanian** [*3]

Abstract

We investigate the application of deep reinforcement learning to the task of autonomous car driving. Autonomous driving is an important area of research today and reinforcement learning algorithms applied to solve this task have been quite rare. However, reinforcement learning coupled with deep learning can act as a potent combination to learn robust policies given the right parameters. In this work, we attempt to make an autonomous driving agent learn to follow the lane in a 'realistic' urban traffic simulation environment. We see that within a short span of time our agent learns a fairly 'sane' policy even with a sparse reward signal and highly stochastic environment. We demonstrate the use of a fairly generic reinforcement learning algorithm to our task. We discuss our methods and various 'engineering hacks' needed to make the training feasible.

1. Introduction

Autonomous Driving is a topic that is gaining a lot of traction from both the research communities and industries. This requires sensing the environment around the car and moving the car with minimal human intervention. The industrial community aims at a robust solution with the help of 3D geometric maps from sensors like LIDAR, Kinect and geographic location from the GPS sensor etc. This approach requires immense processing power and relies on external mapping infrastructure rather than understanding the local scene.

The proposed solution aims at modelling the autonomous driving problem using the local scene captured with a monocular camera as a Markov Decision Process (MDP) where an agent is trained to take actions transitioning from one state to another. The objective of Markov Decision Problems is to maximize the expected cumulative reward obtained based on a momentary reward function. The reward function gives positive reward for every correct move and imposes a penalty for undesired consequences such as in a collision or straying into the other-lane/off-road. The trained agent eventually learns to take the best action for any given state, thereby accumulating positive rewards.

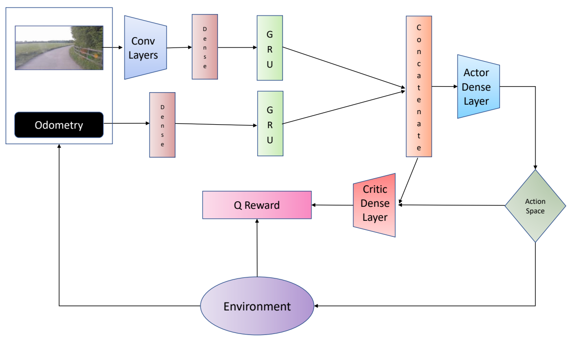

Figure 1. The figures illustrates the entire actor-critic architecture

The task of autonomous driving saw a wide variety of approaches in the past using a multitude of sensors for localization, planning, control, perception techniques such as semantic segmentation, mapping the environment etc. It is evident that breaking this problem into a number of constituent problems lead to an increase in the complexity. Comparatively, reinforcement learning (RL) is a very simplistic tool. Even in the RL paradigm, methods such as imitation learning where actions are learned through expert demonstrations and model-based RL which learn the dynamical model of the system have been tested in the past. However, imitation learning is difficult to scale up to large scale scenarios like the task at hand and model-based RL are data-inefficient and need a huge amount of learning data. Hence, we resort to a model-free RL in our work and employ customized reward function to result in an expected behaviour.

In this paper, we:

[*]Equal contribution [1]Department of Computer Science, Texas A&M University, Texas, USA [2]Department of Aerospace Engineering, Texas A&M University, Texas, USA [3]Department of Electrical Engineering, Texas A&M University, Texas, USA. Correspondence to: Ankur Roy Chowdhury <ankurrc@tamu.edu>.

1. pose the problem of autonomous driving to Markov Decision Process, making it solvable with deep Reinforcement Learning (RL)
2. propose an approach using only RGB images from the local scene for state representation
3. formulate a Deep Deterministic Policy Gradient (DDPG) agent and induce selective attributes by employing some canonical techniques
4. implement the proposed architecture in a simulation environment that emulates a realistic urban traffic scenario.

2. Background

Reinforcement Learning (RL) provides us with a framework to solve Markov Decision Processes (MDPs). Let an MDP be denoted by the tuple $(S_t, A_t, p_{sa}, R_t, \gamma)$, where S_t denotes the current state, A_t is the action, p_{sa} is the transition probability, R_t, the corresponding reward and γ $(\in [0,1])$ being the discount factor, all at time t. The value function for a policy π is then given by

$$V_\pi(s) = E[\Sigma_{t=0}^{\infty}\gamma^t R(s_t, \pi(s_t))|s_0 = s] \tag{1}$$

An equivalent formulation can be derived from the Bellman equation to Q-function following policy π is as follows:

$$Q_\pi(s_0, a_0) = E[R(s_1, s_0, a_0) + \gamma * Q_\pi(s_1, \pi(s_1))]$$

An optimal policy is one that maximizes the expected cumulative rewards

$$\pi^*(s) = \underset{\pi}{argmax} V_\pi(s)$$

or in terms of Q-function,

$$\pi^*(s) = \underset{a}{argmax} Q^*(s, a)$$

Q-learning is an iterative algorithm to solve MDPs based on the aforementioned Bellman equation that gradually converges the Q state-action value to Q^*. In deep reinforcement learning, DDPG is one of the successful approaches that employs actor-critic architecture for Q-learning and policy update to dealing with MDPs in continuous state and action spaces.

(Kendall et al., 2018) is one of the earlier works involving deep reinforcement learning based approach for autonomous car driving. They demonstrated that a DDPG (deep deterministic policy gradient) based method albeit using a single monocular image as input can rapidly learn the task. Further, they showed that a better state representation like using a Variational Autoencoder (VAE) greatly expedites the training process. (Sallab et al., 2017) proposes a framework for an end-to-end autonomous car driving deep RL pipeline using RNNs to deal with POMDP scenarios and implements in TORCS simulator. (Bojarski et al., 2016) empirically shows successful training of convolutional neural networks (CNNs) to obtain steering commands out of a monocular camera image input. The demerit with the above methods is that their environment is simplistic and is not practical for real-world urban traffic scenarios. Also, RNNs suffers from the vanishing gradient problem. Hence, we need networks like Long short-term memory (LSTM) or Gated Recurrent Unit (GRU) or option graphs as mentioned in (Shalev-Shwartz et al., 2016) to extract state from temporally correlated data.

As for dealing with the image data, (Jang et al., 2017) extensively discusses image pre-processing with a convolutional neural network (CNN)for a state representation and how that affects the performance of the deep-rl pipeline. Their proposed techniques includes replacing max-pooling, batch normalization or dropout layers by mean pixels centering on zero, xavier weight initialized with reduced scale and relatively shallow network. Now, we discuss in detail the algorithms that form the core component of our work.

2.1. DDPG

Deep Deterministic Policy Gradient (DDPG) is a policy-gradient based off-policy reinforcement learning algorithm that operates in continuous state and action spaces. It relies on two function approximation networks one each for the actor and the critic. The critic network estimates the $Q(s, a)$ value given the state and the action taken, while the actor network engenders a policy given the current state. Neural networks are employed to represent the networks.

The off-policy characteristic of the algorithm employs a separate behavioural policy by introducing additive noise to the policy output obtained from the actor network. The critic network minimizes loss based on the temporal-difference (TD) error and the actor network uses the deterministic policy gradient theorem to update its policy gradient as shown below:

Critic update by minimizing the loss:

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

Actor policy gradient update:

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

Target networks one each for the actor and the critic are employed for a gradual update of network parameters, thereby reducing the oscillations and a better training stability.

2.2. Exploration

To account for the exploration, the behavioural policy consists of an off-policy term arising from a random process. We obtain discrete samples from Ornstein-Uhlenbeck (OU) process to generate noise as followed in the original DDPG method. The OU process has mean-reverting property and produces temporally correlated noise samples as follows:

$$dx_t = \Theta(\mu - x_t)dt + \sigma dW$$

where Θ indicates how fast the process reverts to mean, μ is the equilibrium or the mean value and σ corresponds to the degree of volatility of the process.

2.3. Prioritized Experience Replay

In reinforcement learning, typically while training a network, batches of experience tuples are uniform-randomly picked from the memory instead of recent transitions to remove the correlations between consecutive experience samples and thereby enhance the stability of training. The prioritized experience replay (PER) provides an intelligent way of selecting experience samples as opposed to naively picking random samples. PER prioritizes the sample that has high temporal-difference (TD) error. Let p_t be the priority assigned to an experience sample. Then, $p_t = |\delta_t| + e$, where δ_t is the TD error and e is a constant to ensure that p_t is not zero. The probability of a sample i being chosen is then given by

$$P(i) = \frac{p_i^a}{\Sigma_k p_k^a}$$

where a is a hyper-parameter that introduces randomness in the experience selection. Since this inherently produces a bias inclined by using high prioritized samples more likely, importance sampling (IS) weights are used to adjust the updating by reducing the weights of the often-seen samples. The IS weight for a sample i is given by

$$w_i = (\frac{1}{N} \cdot \frac{1}{P(i)})^\beta$$

2.4. Gated Recurrent Units

Gated Recurrent Units (GRUs) are a gating mechanism in recurrent neural networks(RNNs) aimed at solving the 'vanishing gradient problem' in a standard RNN. To deal with the vanishing gradients, GRU uses an update gate and a reset gate. An update gate is used to determine the amount of past information to be passed into the future while the reset gate is used to decide the quantity of data to forget from the model. Also, they require fewer parameters as compared to an LSTM and hence are data-efficient while training.

3. Methodology

In the following sections we

1. discuss the system setup to run the experiments
2. explain the core components of our algorithm in detail

3.1. System Setup

We used a fork of KerasRL (Plappert, 2016) for building our agent. We modified the following components:

1. DDPG Agent: The DDPG agent that comes with KerasRL does not work with Prioritized Experience Replay. We modified the code to take into account PER.
2. PER: KerasRL does not implement PER. We added the requisite code for PER by implementing the datastructures needed, namely: SegmentTree. Also, because we use sequences of observations to form a state, we needed means to concatenate accross observations from actions. Thus, we added support for the same.

On the simulation front, we used Carla (Dosovitskiy et al., 2017).
To take advantage of KerasRL's integration with OpenAI Gym interface, we developed wrapper code on top of Carla's python client that enabled us to control Carla using a Gym interface. Thus, enabling us to use KerasRL code with minimum modifications.
We also developed code that took care of randomising rollouts across different episodes. This was needed in order to make our agent robust and also not make it overfit. We randomised vehicle spawn points and weather conditions.

3.2. Approach

3.2.1. Observation space

We include the following variables in our observation space:

- Odometry: 10 dimensional space.
    1. Location [x, y, z]: the current player position with respect to world-coordinates.
    2. Destination Location [x, y, z]: the position of the destination with respect to world coordinates
    3. Speed: velocity of the vehicle in m/s.
    4. Acceleration [x, y, z]: acceleration of the vehicle.
- Monocular Images: $84 \times 84 \times 3$ dimensional space
    1. We captured RGB images from a camera installed on the hood of the vehicle. The images were captured in 84×84 resolution and were 3 channel.

Figure 2. CARLA environment

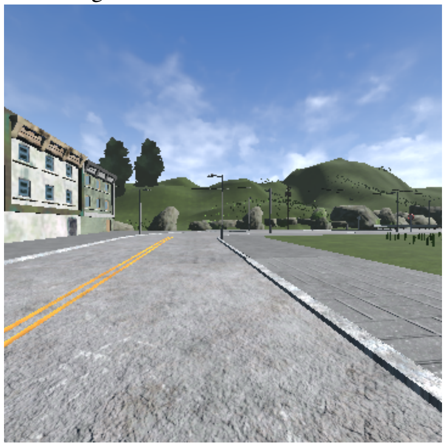

Figure 3. Camera input

Figure 4. CARLA simulator

We concatenated the two observation types as [Odometry, Monocular Images].

3.2.2. State Space

The observation space alone cannot account for the entire history of our Markov Decision Process. Thus, we use a window length of size 4 in order to account for the state space. We take 4 observations in sequence and concatenate them such that it looks like:

$$[[O_t, O_{t+1}, O_{t+2}, O_{t+3}], [I_t, I_{t+1}, I_{t+2}, I_{t+3}]]$$

Where:

$$O \equiv Odometry$$
$$I \equiv Image$$

3.2.3. Action Space

Since we are working with continuous control, we include the following components in our action space:

- Steering $[-1, 1]$: the steering angle to be applied at a given moment. A positive value indicates steering right, while a negative values indicates going left.
- Throttle $[-1, 1]$: throttle to be applied at a given moment. A positive value indicates accelerating forward, while a negative value indicates accelerating backwards.

3.3. Network Architecture

Our neural network consists of three distinct parts:

3.3.1. Input Head

The input head consists of 2 parts:

- Odometry: This part of the neural network reads inputs:

  $$[b, t_s, [O_t, O_{t+1}, O_{t+2}, O_{t+3}]]$$

  It is responsible for distilling the raw state space into a dense representation to be consumed by the actor and critic networks. We use a *TimeDistributed* layer to process each of the time steps individually by connecting it to a fully-connected layer with 32 nodes. We batch normalise the output by mean centering it and feed it to a recurrent layer. The recurrent layer is a Gated Recurrent Unit (GRU) which distills the output to 16 dimensions. Note, $t_s = 4$.

- Image: This part of the neural network reads inputs:

  $$[b, t_s, [I_t, I_{t+1}, I_{t+2}, I_{t+3}]]$$

  We use a TimeDistributed layer to process each of the time steps individually by connecting it to 3 Convolutional layers with 16, 32, 32 filters and strides 3, 3, 2, respectively. We batch normalise the output by mean centering it and feed it to a recurrent layer. The recurrent layer is a Gated Recurrent Unit (GRU) which distills the output to 256 dimensions. Note, $t_s = 4$.

We take the output of the two branches and concatenate them to form an output of:

$$[b, 272]$$

All the layers used He Uniform normal for initialization. The non-linearity used was ReLU. The recurrent units used a dropout of 0.2 and recurrent dropout of 0.2.

3.3.2. Actor

The ’actor’ is a fully-connected network. It takes input of the form:

$$[b, 272]$$

It has 2 fully-connected layers of 200 nodes each. All the nodes are initialized using random weights from of uniform distribution between -3×10^{-4} to 3×10^{-4}. Hidden layers used *ReLU* as the non-linearity except for the output which uses *TanH*. The output of the network is 2 scalars (steer, throttle) between [-1, 1] and of the form:

$$[b, 2]$$

3.3.3. CRITIC

The 'critic' is a fully-connected network. It takes input of the form:

$$[b, 274]$$

It uses the output from the input head as well as the 'actor'. It has 2 fully-connected layers of 200 nodes each. All the nodes are initialized using random weights from of uniform distribution between -3×10^{-4} to 3×10^{-4}. Hidden layers used ReLU as the non-linearity except for the output which is linear. We also use l2 regularization for the weights of the network with $\lambda = 0.01$. The output of the network is a scalar (Q-value of state and action) and of the form:

$$[b, 1]$$

NB: here b is the batch size and t_s denotes the time steps

Hyperparameters:

- Adam as the optimizer with a learning rate of 1×10^{-3} and 1×10^{-4} for the 'actor' and 'critic' respectively.
- $\tau = 1 \times 10^{-3}$
- $\gamma = 0.99$
- Ornstein-Uhlenbeck Process:
  - $\theta = .15$
  - $\mu = 0.$
  - $\sigma = .2$
  - Linear annealing
- Prioritized Experience Replay:
  - $\alpha = 0.6$
  - $\beta = 0.6$
  - Linear annealing

3.4. Reward Function

The reward function used is as follows:

$$\begin{aligned} r = & v - i_{otherlane} \times 15 - i_{otherlane} \times 15 \\ & - c \times 100 - |s| \times 10 \\ & + (d_{last} - d_{current}) \times 1000 \end{aligned}$$

Here,

- v: Velocity of the vehicle. Reward is limited to 30 for speed exceeding 30 m/s. We reward the agent for maintaining speed, at the same time we discourage overspeeding.
- $i_{otherlane}$: Percentage of the car on the wrong lane. We punish the agent for going on to the other lane.
- $i_{offroad}$: Percentage of the car outside the road. We punish the agent for going offroad.
- c: Whether the vehicle collided with pedestrians, vehicles or property. We highly punish the agent for collisions.
- s: Steering. We mildy punish the agent for steering too much.
- d_{last}: Last recorded distance between the agent location and the destination.
- $d_{currernt}$: Current distance between the agent and the destination.
  We check to see the difference between the last distance to destination and the current distance to destination. If the difference is positive, that means the agent is heading the right way and we reward it. However, on the other hand, if the difference is negative, the agent is heading farther away from the destination and we punish the agent heavily.

4. Results

The architecture whose description is aforementioned is implemented and trained for 400 episodes. It has been observed that the policy does not converge to the desired command output. At the 70th episode, we observed the best performance. The agent followed the lane perfectly for around 180 meters and struggled to take a decision at a sharp turn. The same can be seen in $video - 1$. It shows the execution in a different environment and starting location. It is noted that it exhibited an identical behaviour as long as it is on a straight road and failed at the corner. The total reward vs episodes plot on fig.1 shows that the total reward increases at first and gradually declines towards a zero reward. After a thorough introspection, we interpret the results as follows:

1) CARLA is a photo-realistic environment. The environment consists of every possible traffic element emulating a real-world urban scenario including the traffic signals, poles, footpath, various marking on lanes. Hence, there is a greater scope for the car to be off-road and off-lane.
2) The road-markings are ambiguous to the car in a situation when it doesn't know whether to take right or left. Apparently, this complexity is not considered in the problem.

Hence, in a realistic environment such as ours, we need a much more sophisticated reward function. We believe it is possible only using approaches such as transfer learning to learn the reward instead of manual design and tuning.

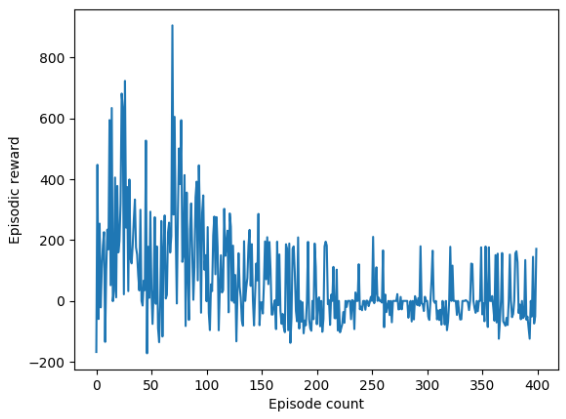

Figure 5. Total rewards vs no. of episodes

Modifications:
Apart from the general framework, the following 'engineering hacks' were employed to get the agent to train better:

1. Action Repetition and Frame Skipping: If using different actions for every step, we might end up with consecutive frames that are too similar to each other. As a result, the observations may be too similar to be of any use to be stacked together. We use action repeats where a particular actions is repeated 4 times and we take the observation of every 4^{th} frame (frame-skipping). This way of replay buffer can stack observations that have much more information.

2. Frame Stacking: Rather than using a single observation as state, we stacked a sequence of 4 observations to be the state.

3. Reward Clipping: We clip the rewards obtained so that the delta change used to update the parameters of the neural network are not too large.

4. Gradient Clipping: We clip the gradients for the same reason as mentioned above. Moreover, we use Huber Loss to achiever gradient clipping, which is given as:

$$f(x) = \frac{1}{2}x^2 \text{ if } |x| \leq \delta,\ \delta(|x| - \frac{1}{2}\delta) \text{ otherwise.}$$

5. Storing images in *int8* format: This enables us to save a lot of space instead of saving them in *float32*. We convert the images to *float32* (needed for consumption by neural networks), only when we sample from the replay buffer.

5. Conclusion

We verified that a generic algorithm like DDPG can be applied to the continuous control task of driving an autonomous vehicle. Even though the policy performed fairly well, the optimal policy could not be obtained. This may be attributed to the fact that the agent needed more episodes to learn. Also, we learned that using deep reinforcement learning requires a lot of compute power, and a viable alternative is to look into asynchronous algorithms like A3C (Mnih et al., 2016).

For future work, we consider using a form of curriculam learning (Bengio et al., 2009). Our intention is to first make the agent learn simpler tasks like following the lane along two fixed locations. Once the agent learns to do so, we next randomise the positions so that the agent can generalise better. Another improvement would be to add a PID controller for smoothing out the output from the neural network. Yet another avenue would be to explore meta-learning algorithms rather than learning tabula-rasa. Ultimately, we would like to consider running the algorithm on an actual vehicle.

Acknowledgements

We sincerely acknowledge the support and encouragement from our instructor Dr. Dileep Kalathil.

References

Bengio, Y., Louradour, J., Collobert, R., and Weston, J. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pp. 41–48. ACM, 2009.

Bojarski, M., Testa, D. D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., and Zieba, K. *End to End Learning for Self-Driving Cars*. *CoRR*, abs/1604.07316, 2016. URL http://arxiv.org/abs/1604.07316.

Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., and Koltun, V. *CARLA: An Open Urban Driving Simulator*. pp. 1–16, 2017.

Jang, S. W., Min, J., and Lee, C. *Reinforcement Car Racing with A3C*. 2017.

Kendall, A., Hawke, J., Janz, D., Mazur, P., Reda, D., Allen, J.-M., Lam, V.-D., Bewley, A., and Shah, A. *Learning to Drive in a Day*. *CoRR*, abs/1807.00412, 2018.

Legrand, M. *Deep Reinforcement Learning for Autonomous Vehicle Control among Human Drivers*. PhD thesis, Department of Computer Science, Universit libre de Bruxelles, 2016-2017.

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*. *CoRR*, abs/1509.02971, 2015. URL http://arxiv.org/abs/1509.02971.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937, 2016.

Plappert, M. *keras-rl*. https://github.com/keras-rl/keras-rl, 2016.

Pomerleau, D. A. *Advances in Neural Information Processing Systems 1*. chapter ALVINN: An Autonomous Land Vehicle in a Neural Network, pp. 305–313. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1989. ISBN 1-558-60015-9. URL http://dl.acm.org/citation.cfm?id=89851.89891.

Sallab, A., Abdou, M., Perot, E., and Yogamani, S. *Deep Reinforcement Learning framework for Autonomous Driving*. *Electronic Imaging*, 2017:70–76, jan 2017.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. *Prioritized Experience Replay*. *CoRR*, abs/1511.05952, 2015. URL http://arxiv.org/abs/1511.05952.

Shalev-Shwartz, S., Shammah, S., and Shashua, A. *Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving*. *CoRR*, abs/1610.03295, 2016. URL http://arxiv.org/abs/1610.03295.

Williams, G., Drews, P., Goldfain, B., Rehg, J. M., and Theodorou, E. A. Information theoretic model predictive control: Theory and applications to autonomous driving. *CoRR*, abs/1707.02342, 2017. URL http://arxiv.org/abs/1707.02342.

A. Codebase

https://github.com/nautilusPrime/autodrive_ddpg