Autonomous Vehicle Control in TORCS using Deep Deterministic Policy Gradient

Arne Gevaert ARNE.GEVAERT@UGENT.BE
Ghent Univeristy

Rien Maertens RIEN.MAERTENS@UGENT.BE
Ghent University

Abstract

We apply the DDPG reinforcement learning algorithm to produce an agent that drives a race car in TORCS. We propose methods improving the learning speed and performance of the agent in our specific use-case. Training on a set of different tracks prevents overfitting, improving performance on unknown tracks. Gradually increasing the episode length improves learning speed and performance because the agent spends less time in a stuck state. Reducing the state dimensionality shortens the actual learning speed while not having a significant performance decrease. Attempts to find a reward function which enables the agent follow the optimal *race line* of the track were unsuccesful.

1. Introduction

The objective of this research project is to apply a reinforcement learning approach to train an agent to drive autonomously in TORCS (The Open Racing Car Simulator) (Wymann et al., 2014).

TORCS is a modern simulation platform frequently used for research in autonomous driving (Ganesh et al., 2016), (Lillicrap et al., 2015), (El Sallab et al., 2016). Advantages of training an agent in a simulation include a highly reduced cost and risk, as well as much faster training. The disadvantage of this is that simulations are only an approximation of reality, making the resulting agent far from usable in any real-world application. Since the focus of this project is mainly reinforcement learning in general, rather than the development of autonomous vehicles, this disadvantage is of little importance.

Classical Q-learning aims to construct a function $Q : S \times A \rightarrow \mathbb{R}$ that predicts the expected total reward for state-action pairs, which is then used to construct an optimal control policy (Sutton & Barto, 1998). The update rule for a state-action pair (s, a) is defined as follows:

$$Q'(s,a) := Q(s,a) + \alpha(R + \gamma \max_{a' \in A} Q(s', a') - Q(s,a))$$

Where A is the action space, R is the immediate reward and α and γ are tunable hyperparameters. By originally performing random actions, the agent gains experience and the Q-function becomes an increasingly accurate estimation of the outcome of actions in different states. When the Q-function is sufficiently accurate, the action that maximizes its value for a certain state can be selected as the optimal action in that state, resulting in an optimal policy.

As the dimensionality of the state space increases, classical Q-learning becomes less computationally feasible. In order to train an agent to efficiently play games with high-dimensional input such as TORCS, deep architectures such as Google DeepMind's DQN (Mnih et al., 2015) (Deep Q-network) are necessary. Characteristic to these deep architectures is their excellence in deriving efficient representations from high-dimensional data, making them particularly fit for our use case.

Although DQN solves problems with high-dimensional observation spaces, it is only capable of handling discrete and low-dimensional action spaces. Our task of interest has a continuous action space, which poses a problem. A naive approach to applying methods such as DQN to continuous action spaces is by discretizing the action space. However, this approach has proven to perform poorly (Ganesh et al., 2016), mainly due to the curse of dimensionality: the number of actions increases exponentially with the number

Paper in fulfillment of the requirements for the course Artificial Intelligence at the University of Ghent.

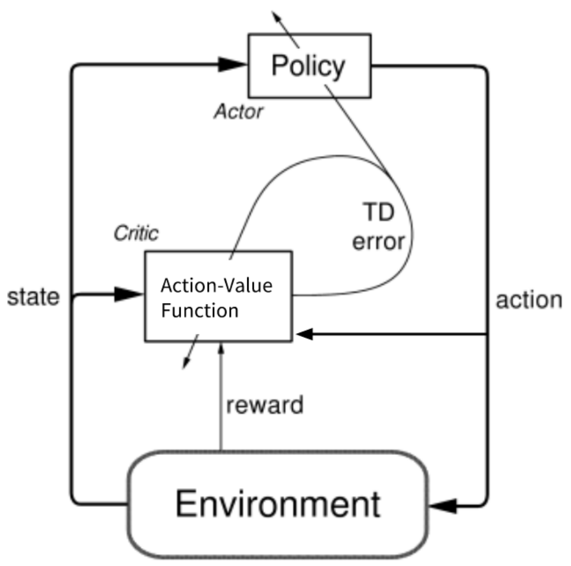


Figure 1. The actor-critic architecture, based on Fig. 6.15 from (Emami, 2016). Note that we modified the figure to represent the fact that the critic is an action-value function, mapping action, reward and state to its output.

of degrees of freedom. This means to make the training computationally feasible, a very coarse discretization must be made. But a coarse discretization results in rough control, which is undesirable in tasks like autonomous driving.

The approach we apply in our research is DDPG (Deep Deterministic Policy Gradients) (Lillicrap et al., 2015). DDPG is a model-free, off-policy actor-critic algorithm using deep function approximators that is capable of learning policies for problems with high-dimensional, continuous action and state spaces.

In section 2 we will give a short introduction of DDPG. Next, in section 3, we mention the architecture of the network we used. In section 4 we mention which experiments we conducted. The results of these experiments will be discussed in section 5. Finally, in section 6, we form a conclusion together with the issues, shortcomings and possibilities for future work.

2. Deep Deterministic Policy Gradient

DDPG is a policy gradient algorithm that uses a stochastic behaviour policy to improve exploration, while estimating a deterministic target policy. Policy gradient algorithms optimize a policy by evaluating it, calculating its gradients, and then applying gradient descent.

2.1. Properties

DDPG also applies an Actor-Critic approach to represent the policy function separately from the action-value function (see figure 1). Here, the policy function is called the Actor, and the value function is called the Critic. The Actor takes a state of the environment and calculates an action, and the critic estimates the Q-value of this combination of action and state. In DDPG, these Actor and Critic functions are implemented using two neural networks. Both networks generate a temporal-difference error signal at each time step from which the gradients can be calculated.

Apart from these architectural properties, DDPG also implements two major extensions to ensure efficient convergence. First, simply training and evaluating the policy and value functions using many temporally-correlated signals introduces great amounts of variance in the approximation of the Q-function. The solution DDPG provides for this problem is experience replay: experiences, encoded as tuples containing state, action, reward and next state, are stored in a buffer. During training, random samples from this buffer are used to break the temporal correlations between signals.

Experience has shown that directly updating the actor and critic weights using the gradients obtained from the TD error signals causes the algorithm to diverge. DDPG handles this problem by using target networks (Mnih et al., 2015) to generate the targets for the TD error computation. This approach has been proved to increase stability significantly. Target networks are copies of the actual actor and critic networks which are used to calculate the target values for the actor and critic. These target networks are updated by slowly tracking the learned networks.

2.2. Training

First the TD targets y_i are calculated as a weighted sum of the immediate reward and the output of the target actor and critic networks μ' and Q'. These targets are then used to calculate the loss function for the critic network.

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}))$$
$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i))^2$$

Here, a sample of size N has been taken from the replay buffer. The index variable i refers to the i'th sample. Next, the actor network is updated using the policy gradient:

$$\nabla_\theta J = \frac{\delta Q(s,a)}{\delta a}\frac{\delta a}{\delta \theta}$$

Where θ represents the weights of the actor network. Since $a = \mu(s|\theta)$, this formula can be written as follows:

$$\nabla_\theta J = \frac{\delta Q(s,a)}{\delta a} \frac{\delta \mu(s|\theta)}{\delta \theta}$$

Finally, the target network weights are updated:

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

An important part of training is exploration. In DDPG, this is achieved by constructing an exploration policy μ' by adding a noise term to the actor policy:

$$\mu'(s) = \mu(s) + \mathcal{N}$$

Where $\mathcal{N}$ is sampled from a random noise process. The noise process used is an Ornstein-Uhlenbeck process (Uhlenbeck & Ornstein, 1930). This noise process has proven to be fit for temporally correlated exploration, and is thus ideal for exploration with continuous action spaces.

3. Architecture

The architecture for our network is depicted in figure 2. The actor network takes 29 state variables as input. These variables include speed in three dimensions, the angle with respect to the track, engine RPM and 19 sensors measuring the distance to the border of the track in different directions. These inputs are connected to a fully connected normalized layer of size 300 with ReLU activation function. This layer is followed by another fully connected normalized layer of size 400 again with ReLU activation function. Finally, this hidden layer is connected to three outputs: steering, acceleration and brake. The steering output uses a tanh activation function, since values for steering range from -1 to 1. Acceleration and brake both use a sigmoid activation function, resulting in values from 0 to 1.

The critic network has a slightly different architecture than the actor network. Since the critic produces a Q-value for a given state and action, this network takes 32 input variables in total: 29 state variables and 3 action variables. The first hidden layer is a fully connected layer of size 300 with ReLU activation function, and is only connected to the 29 state inputs. The action inputs are directly connected to the second hidden layer, which is a fully connected layer of size 600 with ReLU activation function. This second hidden layer is then connected to a single output, which produces the Q-value of the state-action pair.

4. Experiments

We have conducted a few experiments using our DDPG implementation to find ways to improve the performance and learning speed of the algorithm on TORCS specifically.

4.1. Track randomization

The first version of the algorithm is trained on only one track, which results in heavy overfitting on this track. The reward during training is reasonable and performance on the training track is good, but performance on unseen tracks is extremely poor. We then trained the algorithm on an ensemble of 5 tracks, one of which is chosen at random at the beginning of each episode. This results in a much more robust system, while surprisingly converging at about the same rate. This system performs well on unseen tracks and is able to finish a lap without going out of track on most of them.

4.2. Growing episode length

By default, the maximum length of an episode is 2000 steps. Episodes are terminated only if this limit is reached or if the car has turned more than 90 degrees with respect to the track. The result of this is that at the beginning of training, a lot of time is spent almost stationary, as the agent has crashed into a wall and is unable to gain speed without reversing (which it isn't capable of doing). Later on in training, the episodes are often cut before a lap is completed, which means the agent is never able to see certain parts of the track.

We solved this problem by gradually increasing the maximum episode length and starting at a lower value. This significantly reduces the time spent in stationary situations, which results in faster training. Overall performance is also significantly improved, as the agent is able to complete multiple laps in one episode in later stages of training.

4.3. State space dimensionality reduction

TORCS provides 19 range sensors that each measure the distance of the car to the side of the road in a specific angle. These sensors are spread out from 90 degrees left to 90 degrees right. The information provided by these sensors overlaps significantly, as different sensors are only 10 degrees apart. We reduced the dimensionality of the state space by removing every other sensor from the state, resulting in only 10 track sensors (sensors with even indices are kept, thus including 0 and 18). Next to these track sensors, the engine RPM values (4 floating-point variables) are also removed, since the information they provide seems irrelevant when three-dimensional speed is

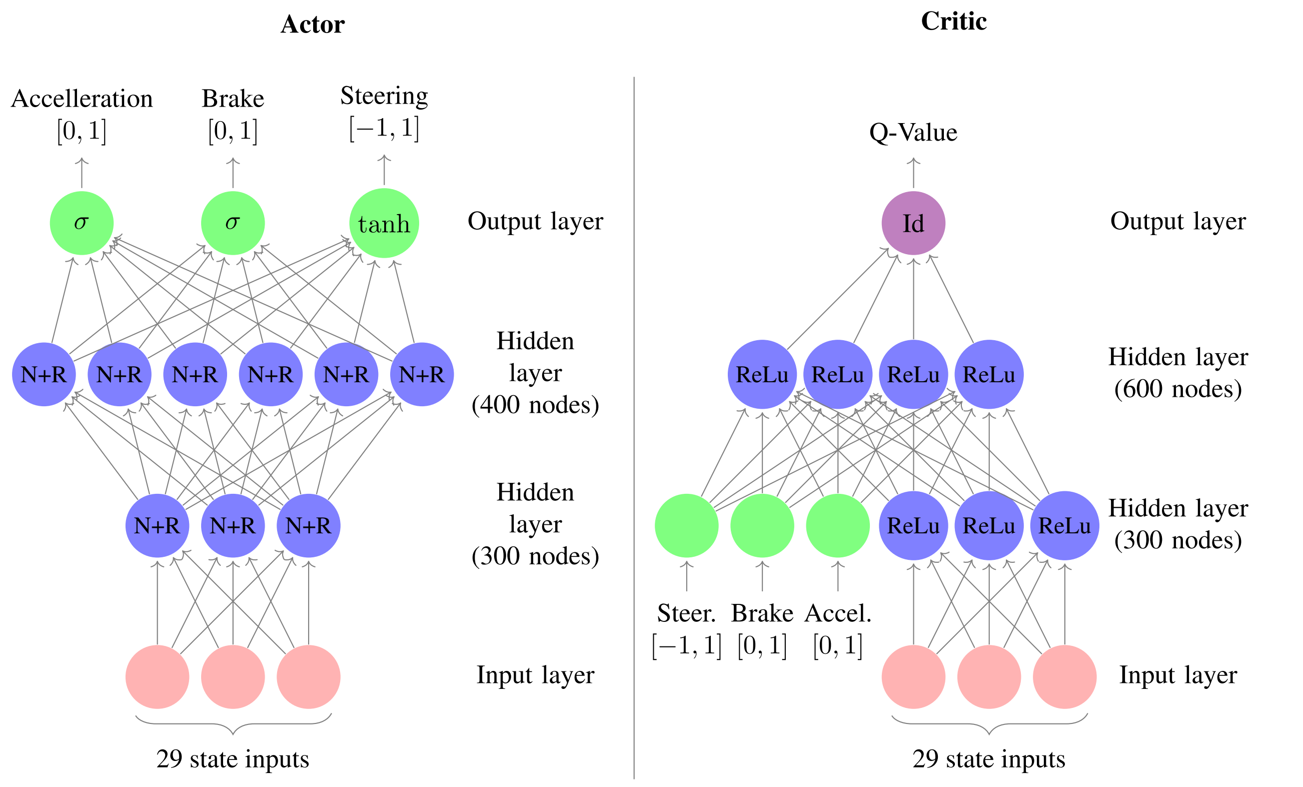


Figure 2. The network architecture. Left is the actor network or the policy, right is the critic network or the Q-value estimator function.

known. The total result is the reduction of the state space dimensionality from 29 to 16.

Our experiments conclude that despite this significant reduction of information, training speed and performance are only slightly influenced. However, the resulting network and its training process are both significantly computationally less intensive, resulting in a more lightweight actor network for evaluation. Training the network is also significantly less intensive, causing the wall-clock time per episode to drop. No exact measurements of wall-clock training time were made however, so we are unsure of the magnitude of the speed increase in training time.

We also performed a similar experiment, keeping the sensors with odd index and removing the sensors with even index. This leaves 9 range sensors, removing one more dimension. Interestingly, this setup learns more quickly, despite having more information removed from its state. A possible explanation for this is that sensors directed perpendicularly to the car axis (sensors 0 and 18) are less valuable than sensors directed in a smaller angle, since the perpendicular sensors measure information about parts of the track that have almost entirely been passed by the car. This setup removes the perpendicular sensors, while keeping more sensors with small angles, making more valuable information available. Also note that the odd-indexed sensors also include the sensor pointing straight forward, enabling the agent to see further than with only the even-indexed sensors.

4.4. Reward function experiments

The reward function used in the first experiments is taken from literature (Ganesh et al., 2016):

$$R = \sum_t |V_{x,t} \cos\theta_t| - |V_{x,t} \sin\theta_t| - |V_{x,t}||x_t - \mu_t|$$

Where θ_t is the angle of the car w.r.t. the track at time t, $V_{x,t}$ is the forward speed of the car at time t, x_t is the relative horizontal position of the car on the track, and μ_t is the middle of the track. This reward function rewards high speeds in the direction of the track (the cosine term), while penalizing high speeds in other directions (the sine term) and distance from the middle of the track (the last term). The resulting behaviour is an agent that tries to follow the exact middle of the track.

Although this behaviour is favourable if the main objective is to stay on track on a possibly unseen track, it is not optimal if the objective is to minimize lap time on a particular track (actual racing, as opposed to simply

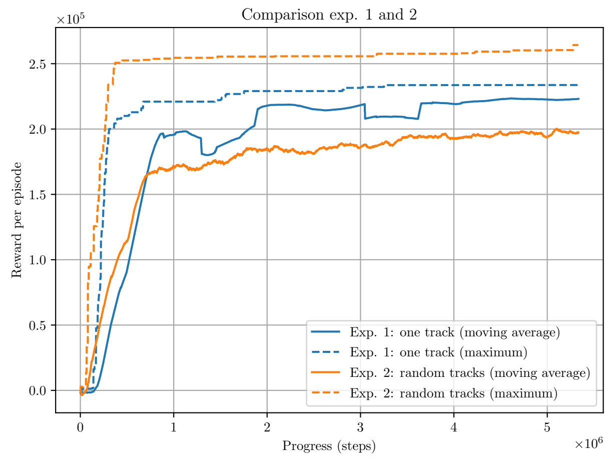


Figure 3. Training on one track vs. on five tracks.

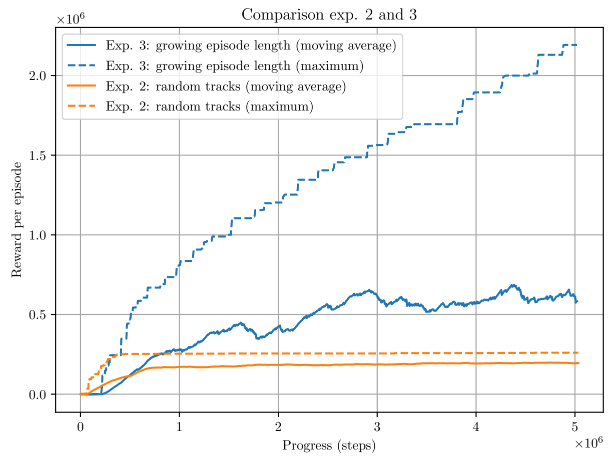


Figure 4. Training with a constant vs. a growing episode length.

driving). In order to minimize lap time, the agent should follow the "racing line": the optimal path of the road, using the entire width of the road to lengthen the radius of a turn. This means the agent should enter a turn from the outside edge, almost touching the apex on the inner edge, and exit the turn back on the outside edge. We tried a few approaches to try to achieve this goal, mainly focused on removing or reducing the track center distance penalty term, while training on only one track:

1. Removing the last penalty term entirely
2. Gradually lowering the last penalty term
3. Halving the last penalty term

None of these approaches have good results. Note that, even though the reward might be high (see further: Results), this doesn't entail that the behaviour is good, since the reward function has been modified. The first and last approach resulted in relatively high rewards, but the resulting behaviour is very poor. The agent quickly loses control of the vehicle. Gradually lowering the last penalty term did not succeed in producing high rewards. This is because the reward function is being changed during training, meaning that actions giving high rewards in early training stages might result in low rewards at later stages.

5. Results

5.1. General experiments

Figure 3 shows the results of the first two experiments. We can see that training on multiple tracks allows the agent to achieve a higher maximum, without reducing the training speed. When running on unseen tracks, the strong reduction in overfitting becomes apparent, as the first agent

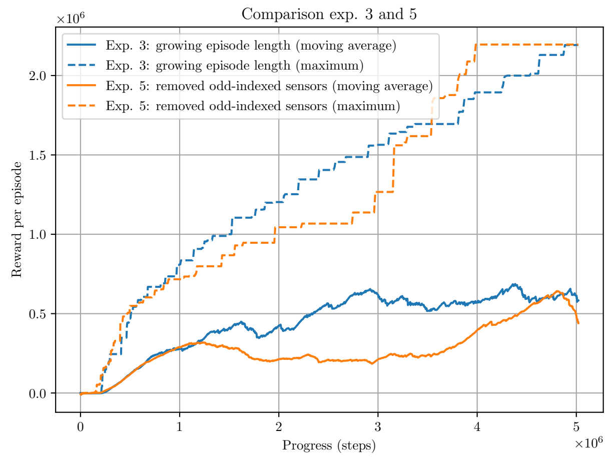


Figure 5. Training with all vs. half of the sensors.

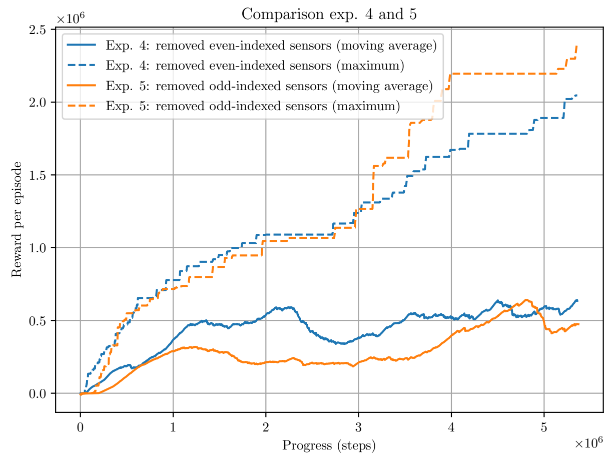
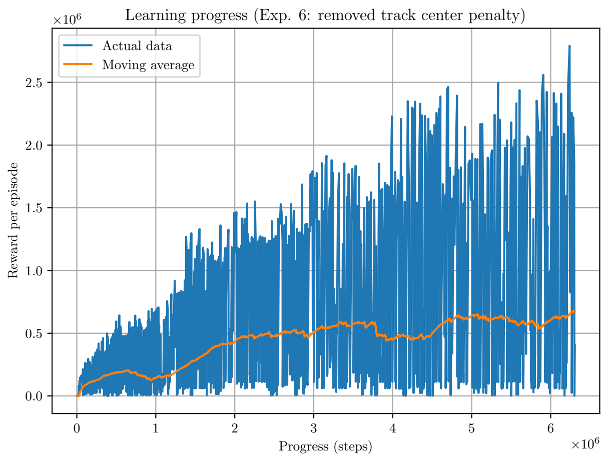
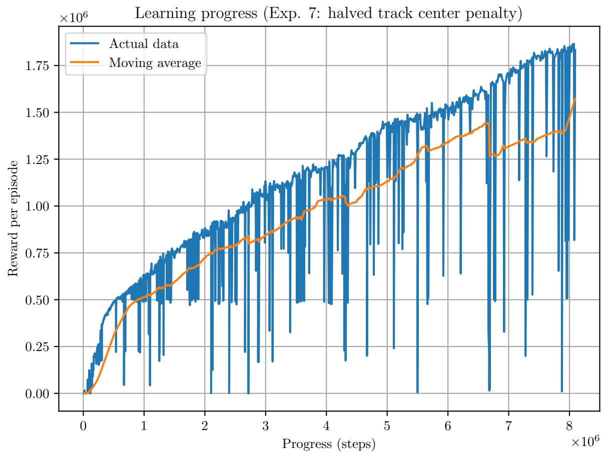


Figure 6. Training with even-indexed vs. odd-indexed sensors.





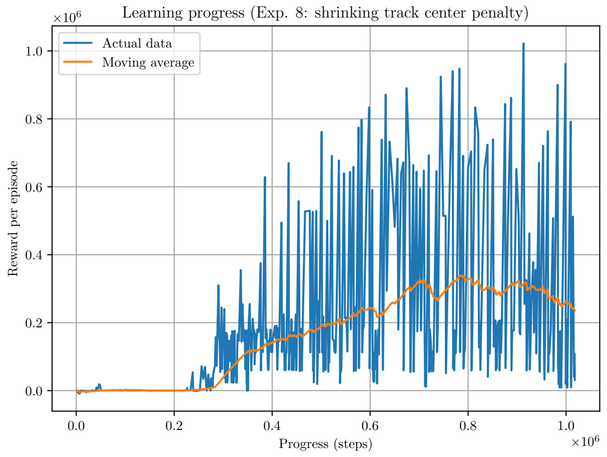


Figure 7. Reward function experiments. From top to bottom: Removed track center distance penalty, halved track center distance penalty, gradually reduced track center distance penalty. Note that removing or halving the track center penalty results in big rewards, but the resulting behaviour is poor. Gradually reducing track center penalty results in poor rewards.

goes outside of any unseen track almost immediately, whereas the agent trained on multiple tracks is able to traverse the bigger part of most tracks.

The third experiment (figure 4) shows that gradually increasing the maximum episode length significantly improves performance, and increases learning speed. This is because a bigger portion of the track is seen during training (in fact, the entire track is seen eventually), while less time is spent driving into walls in earlier stages of training. The training was cut off at 5 million steps, but we can see that the agent has not yet converged. Training for a longer time on more powerful hardware would most likely result in even better performance.

The fourth and fifth experiments illustrate the effect of removing some of the state variables. We can see on figure 5 that this architecture converges a slightly more slowly, but training is significantly less computationally intensive, causing training clock-time to remain the same or even reduce by a small factor (this hasn't been measured extensively enough to draw conclusions). The resulting network is also less computationally intensive to run in a feed-forward manner, which can have advantages in some situations.

Figure 6 compares the effect of keeping the even-index range sensors vs. the odd-indexed range sensors (the fourth and fifth expirement). Keeping the odd-indexed range sensors results in a better performance, despite more information being removed from state.

5.2. Reward function experiments

Figure 7 illustrates the results of different perturbations of the reward function. Despite having greater reward values, removing or halving the track center distance penalty term results in very poor behaviour on both seen and unseen tracks. Gradually decreasing the reward function as training progresses results in equally poor behaviour and lower rewards. This is because the DDPG architecture does not tolerate a reward function evolving over time, as actions that give a high reward in early stages of training give a low reward later on, and vice versa.

We believe that further experimentation on the reward function should be able to produce a better function that gives the agent less restrictions, enabling it to leave the center of the road when necessary. This could result in an agent being able to follow the race line (hugging the corners).

6. Conclusion

A significant improvement in the performance of DDPG for simulated autonomous driving in TORCS has been made using very simple extensions to the existing algorithm. Training on multiple tracks and a growing episode length have a big impact on training efficiency and result in a significant reduction in overfitting. We have also been able to reduce the state dimensionality without reducing the performance significantly, resulting in a computationally more efficient network. However, further work on the subject is definitely still possible.

First of all, in many cases it is clear that the maximum performance has not yet been reached. Longer training on more potent hardware might increase performance significantly. Also, many parameters in the performed experiments are chosen rather arbitrarily. For example, the base episode length in experiment 3 was set to 1500 steps, with an increase of 10 per episode. It is very likely that another setup for these parameters could give even better results. Experiments 4 and 5 suggest that the optimal selection of state variables could also still be undiscovered. We are also still convinced that the optimal reward function has not yet been found for the use case, and a better reward function that allows the agent to leave the center of the track when needed should be feasible to find.

Apart from tweaking in the existing experiments and architecture, some fundamental changes to the architecture might also be worth investigating. The strong temporal correlation in our use case suggests the use of Recurrent Neural Networks (Lipton, 2015) or Long Short-Term Memory (Hochreiter & Schmidhuber, 1997) as an alternative architecture. Another recent development in deep learning called Self-Normalizing Neural Networks (Klambauer et al., 2017) might be an interesting direction of research. Finally, recent research at OpenAI suggests the use of parameter noise (Plappert et al., 2017) to make exploration in reinforcement learning more efficient.

We conclude that, although significant results on the domain of autonomous driving and reinforcement learning have already been reached, there is still a lot of research to be done and the best performing system for the problem remains to be found.

References

El Sallab, A., Abdou, M., Perot, E., and Yogamani, S. End-to-End Deep Reinforcement Learning for Lane Keeping Assist. *ArXiv e-prints*, December 2016.

Emami, P. Deep deterministic policy gradients in tensorflow, August 2016. URL `http://pemami4911.github.io/blog/2016/08/21/ddpg-rl.html`.

Ganesh, A., Charalel, J., Das Sarma, M., and Xu, N. Deep reinforcement learning for simulated autonomous driving, 2016.

Hochreiter, Sepp and Schmidhuber, Jrgen. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735. URL `https://doi.org/10.1162/neco.1997.9.8.1735`.

Klambauer, Günter, Unterthiner, Thomas, Mayr, Andreas, and Hochreiter, Sepp. Self-normalizing neural networks. *CoRR*, abs/1706.02515, 2017. URL `http://arxiv.org/abs/1706.02515`.

Lillicrap, Timothy P., Hunt, Jonathan J., Pritzel, Alexander, Heess, Nicolas, Erez, Tom, Tassa, Yuval, Silver, David, and Wierstra, Daan. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015. URL `http://arxiv.org/abs/1509.02971`.

Lipton, Zachary Chase. A critical review of recurrent neural networks for sequence learning. *CoRR*, abs/1506.00019, 2015. URL `http://arxiv.org/abs/1506.00019`.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A., Veness, Joel, Bellemare, Marc G., Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K., Ostrovski, Georg, Petersen, Stig, Beattie, Charles, Sadik, Amir, Antonoglou, Ioannis, King, Helen, Kumaran, Dharshan, Wierstra, Daan, Legg, Shane, and Hassabis, Demis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015. ISSN 00280836. URL `http://dx.doi.org/10.1038/nature14236`.

Plappert, Matthias, Houthooft, Rein, Dhariwal, Prafulla, Sidor, Szymon, Chen, Richard Y., Chen, Xi, Asfour, Tamim, Abbeel, Pieter, and Andrychowicz, Marcin. Parameter space noise for exploration. *CoRR*, abs/1706.01905, 2017. URL `http://arxiv.org/abs/1706.01905`.

Sutton, Richard S and Barto, Andrew G. *Reinforcement learning: An introduction*, volume 1. MIT Press Cambridge, 1998.

Uhlenbeck, G. E. and Ornstein, L. S. On the theory of the brownian motion. *Phys. Rev.*, 36:823–841, Sep 1930. doi: 10.1103/PhysRev.36.823. URL `https://link.aps.org/doi/10.1103/PhysRev.36.823`.

Wymann, Bernhard, Espi, Eric, Guionneau, Christophe, Dimitrakakis, Christos, Coulom, Rmi, and Sumner, Andrew. TORCS, The Open Racing Car Simulator, 2014. URL `http://www.torcs.org`.