Utilize reinforcement learning to develop a self-driving car in a simulated environment

Contents

Introduction

For self-driving cars, computer games have to potential to become one of the major driving force in the future development of artificial intelligence (AI). The detailed simulation engines of modern computer games provide cheap and portable development environments to build artificial intelligence. Computer games have already been used to create breakthrough research in the field of AI in the last years. Google's deep mind team has developed AIs, combining deep neural networks and reinforcement learning, which played competitively in more than 50 Atari 2600 games with the same information available to humans (Minh & al, 2015). In 2016, different AI's competed against each other in the first-person shooter DOOM, where these AI's developed strategies similar to humans (VizDOOM, 17).

The advantage of using computer games to train these AIs are: Mistakes don't have severe consequences (besides a negative reward); developing an algorithm is cheaper; training can be easily parallelized; and the work can be reproduced. Thus, the recent studies, using the realistic vehicle dynamics to train neural networks for self-driving cars (Loiacono & Cardamone, 2013; Lau, 2017) pave the avenue for the use of reinforcement learning in the field of autonomous vehicle AI for the next years.

Here, a machine learning platform is developed, which allows training the open source game Torcs (http://torcs.sourceforge.net/, 2017). As a basis, gym-torcs (gym_torcs, 2016) will be used, which allows to use Torcs from python. But since gym-torcs is lacking the possibility to train Torcs in parallel. This disables the possibility to use the state of the art machine learning algorithms, such as asynchronous actor critic or running (Minh & al, 2016) elaborate training and testing regimes.

Background

In reinforcement learning an agent (in this study the race car) interacts with an environment (race track) in discrete time steps t. At each t, the agent gets a state s_t from the state space S, takes an action a_t from the action space A, and gets a reward r_t. The behavior of the agent is defined by the policy π: s_t -> P(a_t), which maps the states to the probability of the various actions. After interacting with the environment according a_t defined by π, the agent receives a new state s_{t+1} and a reward r_t. The agent then performs the next action until it reaches a final stage. This whole procedure defines a Markov decision process, where one can define the discounted reward function from a given state $R_t = \sum_{i=t} \gamma^{(i-t)} r_i(a_i, s_i)$, where γ is the discounting factor. In reinforcement learning (RL), one tries to find a policy agent which maximizes the return from a given state (Lillicrap & al, 2016; Minh & al, 2016).

Recently deep neural nets (DNN) have proven to be very successful in the field of (RL), where DNNs have been used to estimate the action value function (θ represents the weights of the DNN), which is the expected return $Q^{\pi}(s, a, \theta) = E[R_t | s_t, a]$ of selecting an action in a given state following a policy π. Similarly the state value function $V^{\pi}(s) = E[R_t | s_t]$, which describes the value the state when following policy π can be estimated using DNNs.

The optimal policy can then be defined, using various methods, for example Q-learning (Sutton & Barto, 2012). In Q-learning a loss function, $L = E[r + \gamma \max \underline{Q}(s', a', \theta_i) - Q(s, a, \theta_i)]$, where s' is the state following s, is minimized.

In order to tackle continuous action spaces (as also present in Torcs) actor critic methods have been used (Minh & al, 2016). Actor critic methods consist of an actor, adjusting the policy π via gradient descent and a critic, which estimates $Q^{\pi}(s, a)$ required for the actor (Figure 1.) (Sutton & Barto, 2012).

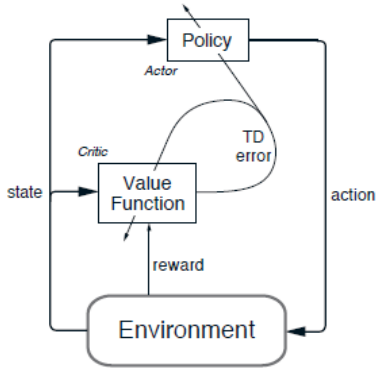

Figure 1. Actor critic methods from ((Sutton & Barto, 2012))

Problem statement

This capstone project will develop a reinforcement learning algorithm using the open source racing game Torcs, which is characterized by a continuous action space. Torcs will simulate the driver (race car driver) and the environment (race track) and provide the necessary sensory data to train this agent. An AI algorithm for this agent will be developed using state of the art reinforcement algorithms, i.e. deep deterministic policy gradient and asynchronous actor critic. The goal is to develop an agent which will drive the race car as a fast as possible around the race track.

Evaluation Metrics

Reward at time point t:

$$R_t = v_x \cos\theta - v_x \sin\theta$$

Total reward (evaluation metric):

$$R = \sum R_t$$

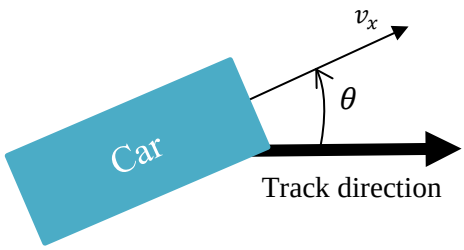

Figure 2. Reward function

As the reward function (Figure 2), we use the accumulated reward of the longitudinal velocity penalized by the transverse velocity, as suggested by (Lau, 2017). The evaluation metric will be the accumulated reward of the different agents on unknown test tracks.

Materials & Methods

Setup

The agents and the Torcs environment run in Docker containers, which can be deployed on Amazon's Web service. Using Docker allows the parallel execution of Torcs. For details please refer to the README.

Interacting with the environment

The data will be generated using a modified version of the open source racing game Torcs (http://torcs.sourceforge.net/, 2017; Loiacono & Cardamone, 2013; gym_torcs, 2016), which simulates the environment. The agent communicates with the environment via a web socket, thereby receiving states (Table 1), rewards (Figure 2) and sending actions (Table 2).

Sensor	Range	Unit	Description
angle	[-π, +π]	rad	Angle between the car and the track axis
track	(0, 200)	meters	Vector of 19 range finder sensors, where each returns the distance between the edge of the track and the car within a range of 200 meters
trackPos	(-∞, +∞)	-	Normalized (w.r.t track width) distance between the car and the track axis. 0 means the car is on the axis, values greater than 1 or -1 means the car is outside the track.
speedX	(-∞, +∞)	km/h	Speed of the car along the longitudinal axis of the car
speedY	(-∞, +∞)	km/h	Speed of the car along the transverse axis of the car
speedZ	(-∞, +∞)	km/h	Speed of the car along the Z-axis of the car
wheelSpinVel	(-∞, +∞)	rad/s	Vector of 4 sensors for the rotation speed of wheels
rpm	(0, +∞)	rpm	Number of rotation per minute of the car engine

Table 1. States of the agent

Actor	Range
steering	(-1.0, 1.0)
acceleration	(-1.0, 1.0)

Table 2. Outputs of the agent to the environment. Please note negative acceleration means braking.

Training & testing data

To train the agent 6 race tracks were selected (Figure 3). The agent is trained on these race tracks, thereby learning an optimal policy to drive around them as fast as possible.

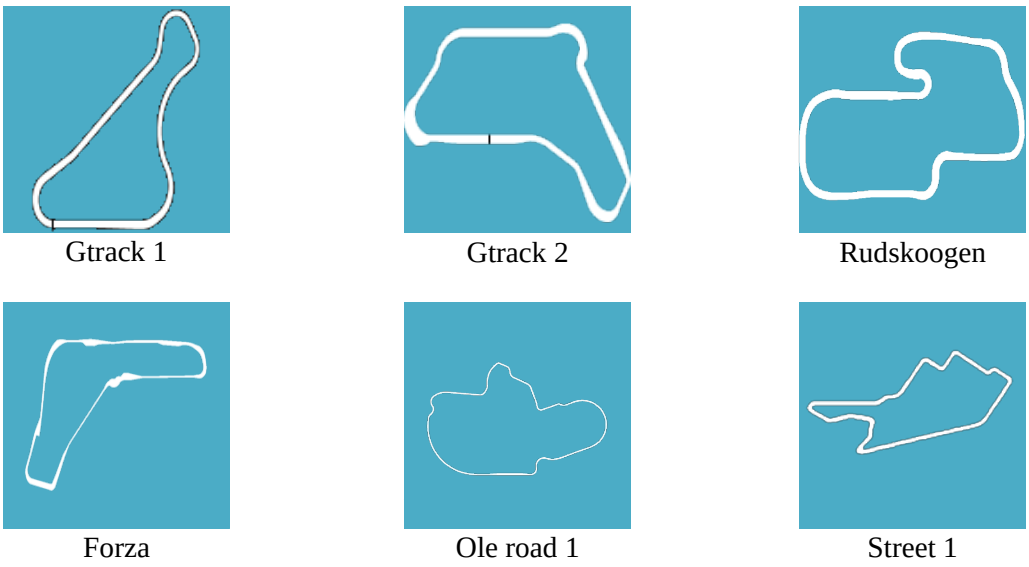

Figure 3. Race tracks used for training

The agent was then tested using the tracks specified in Figure 4.

Figure 4. Race tracks used for testing

Algorithms

Two type of agent were used as the basis: deep deterministic policy gradients (DDPG) and asynchronous actor critic (A3C). The implementation details will be provided in the following section

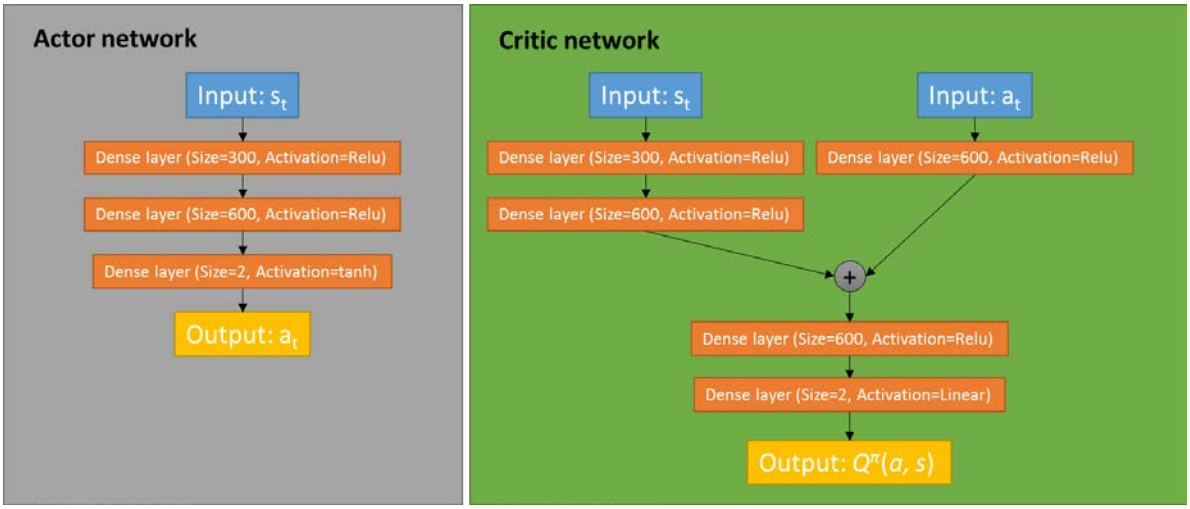

Figure 5. Actor critic network structure

As a first agent, deep deterministic policy gradient algorithm (DDPG) (Lillicrap & al, 2016) was implemented based on (Lau, 2017) (Figure 5). The critic was learned using the Bellman equation from Q-Learning, while the actor was updated using the gradients of the critic with respect to actions (Lillicrap & al, 2016). The networks were trained using minibatch samples from a replat buffer (Minh & al, 2015). To increases the exploration of the algorithm an Ornstein Uhlenbeck process was added to the actions (Lau, 2017; Lillicrap & al, 2016). Further the algorithm contained target actor and critic networks to increase the stability (Minh & al, 2015).

Asynchronous actor critic (A3C) (Minh & al, 2016) was implemented based on (Juliani, 2016). A3C uses multiple agents to explore the policy space, which periodically synchronize their weights (Figure 6). Here, the advantage $A_t(a_t,s_t) = Q^{\pi}(s, a, \theta) - V^{\pi}(s)$ is used to scale the probability returned by the policy network. The basic network topology was the one from DDPG. Further, generalized advantage estimation was used to improve the stability in continuous action spaces (Schulman & al, 2016).

All agents were implemented in Tensorflow. For the gradient descent, the Adam optimizer was used. The weights were normalized using batch normalization. For details, on the meta parameters, please refer to the implementation.

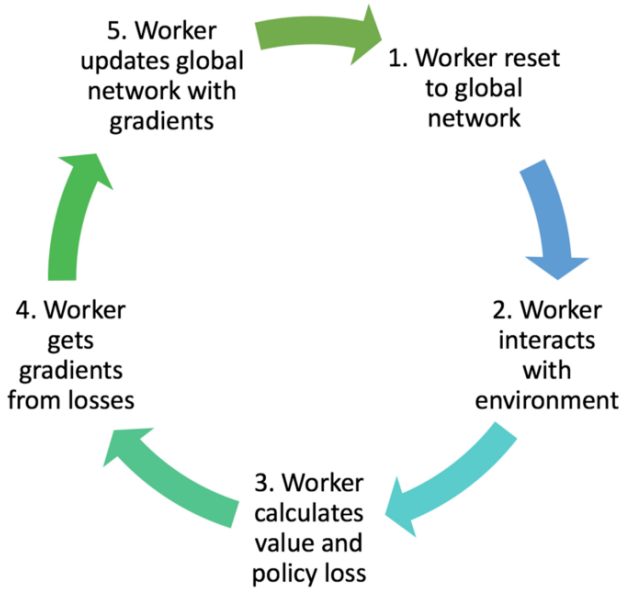

Figure 6. Algorithm used to train the asynchronous actor critic (taken from (Juliani, 2016))

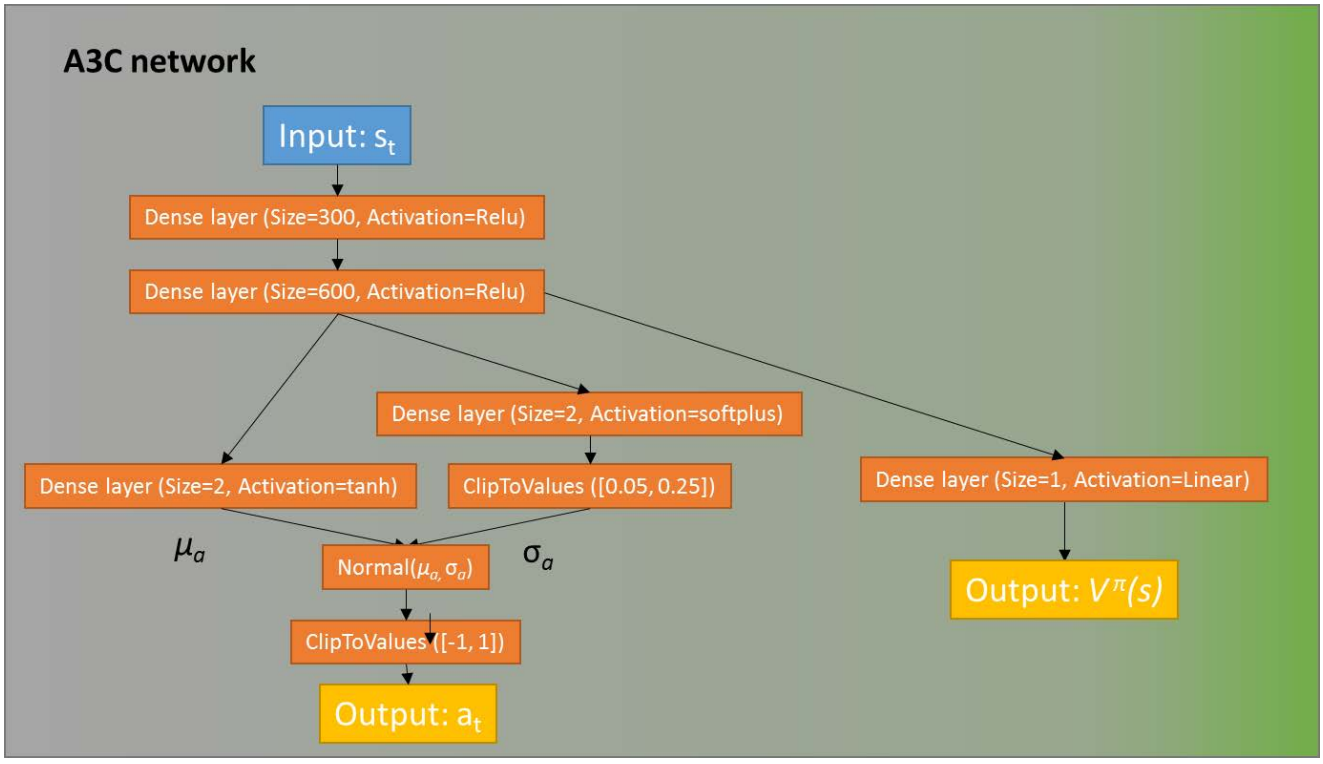

Figure 7. Network structure for the asynchronous actor critic

Benchmark models

As a benchmark model, the DDPG with batch normalization was used. To compare the models the agents were run on each of the test tracks for maximally 1000 actions. The final score was the accumulated score across all test.

Results

Improving the agents

The DDPG was improved by two measures to avoid the agent being stuck in local minima:

1. Early stoppage of the episode, if the agent is stuck. For the early stoppage, the 1000 recent rewards are stored, if the median reward across this 1000 episodes is less than 0.5 end the episode. This feature gets turned out, one
2. Randomly selecting a new track from the training tracks (Figure 3) at the start of each 3rd episode.

Resulting in 3 scenarios for the deep policy gradient algorithm

- DDPG_REF: Benchmark implementation of the deep policy gradient algorithm.
- DDPG_1: DDPG with early stoppage
- DDPG_2: DDPG with random tracks and early stoppage.

The A3C algorithm was improved to allow for continuous actions. To this end, the network returns the mean and standard deviation of a normal distribution, where the new action is then sampled from this distribution. The probability of an action can then be determined using the log probabilites. In order to avoid numerical difficulties, the output of the normal distribution and the standard deviation were clipped (Figure 7).

Training

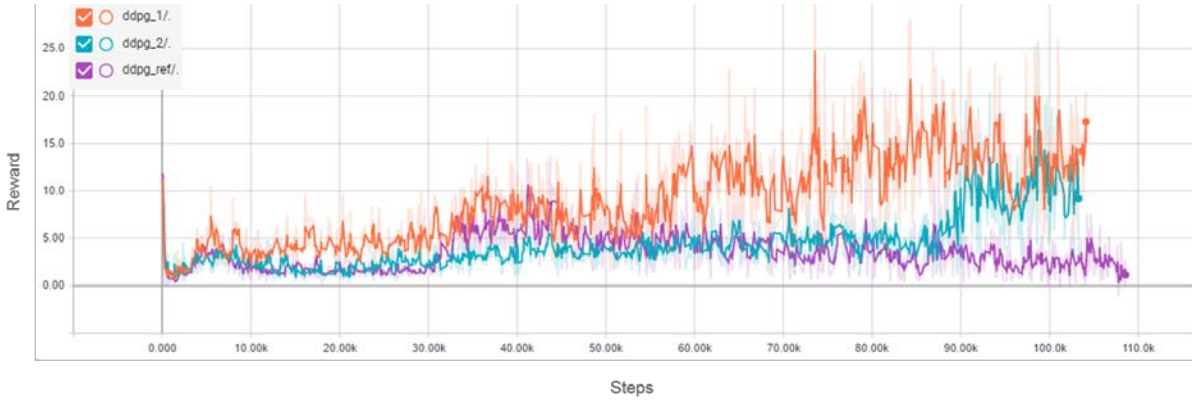

Figure 8. Algorithm used to train the asynchronous actor critic (taken from (Juliani, 2016))

To train the algorithms they were optimized using Tensorflow. When training the DDPG agent, the advantage of using the early stoppage was clearly observable (Figure 8), where both the DDPG_1 and DDPG_2 received better rewards than the baseline agent (DDPG_ref). The DDPG_2 seemed to be not fully started, which might be partially explained by the increased variability introduced by constantly switching the scenarios

The A3C algorithm showed very poor performance. The observed rewards (Figure 9) indicate that the agent was constantly jumping between different policies. The reward course also indicates that the agent is stuck in a local minimum.

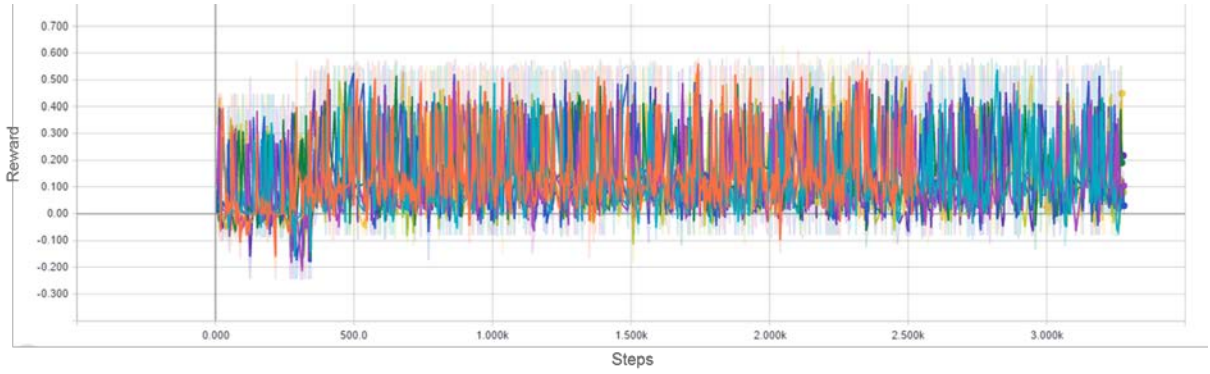

Figure 9. Algorithm used to train the asynchronous actor critic (taken from (Juliani, 2016))

Testing

Track / Scenario	DDPG_REF	DDPG_1	DDPG_2
alpine-2	-16.621978	215.928331	218.888216
e-track-6	-18.431343	296.924800	305.312837
g-track-3	-11.332032	280.188617	280.339358
∑	**-46.38**	**793.04**	**804.54**

Table 3. Accumulated test score of the agent across all race tracks (cf. Figure 4)

As a final test, we measure the agents perform on the previously unknown test tracks. To this end the agent gets 1000 steps which it can perform on the track. If the agent stops early, the test it over. Here, it clearly shows that the early-stoppage criterion increases not only the training put also the test performance of the agent. The DDPG algorithms with early stoppage (1/2) clearly outcompete the baseline agent. Using various race tracks in the training on the other hand did not yield significant improvement (compare DDPG_1 with DDPG_2).

The A3C algorithm performed very poorly, indicating that the current set of meta parameters is not sufficient to handle the environment in a continuous action space. Confirming the notion that the agent was stuck in a local minimum.

Conclusions and further recommendations

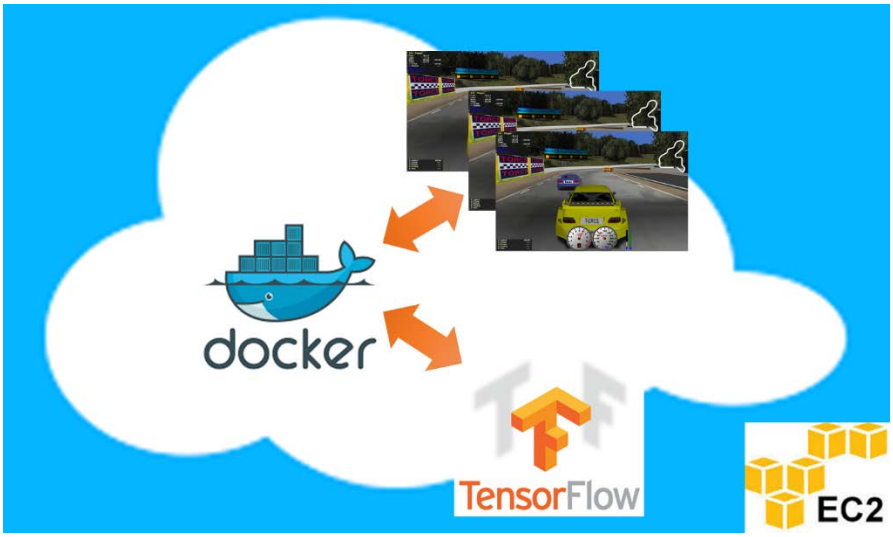

Figure 10. Overview of the cloud based reinforcement learning platform. Docker is hosted on Amazon EC2 and manages on the one hand the agents implemented in Tensorflow and the Torcs environments.

This study developed based on two state of the art reinforcement learning algorithms, DDPG and A3C, agents to tackle the continuous action space of the race game Torcs. Especially from training the A3C algorithm, it became apparent that continuous action space has many local minima and the algorithm has the tendency to get stuck in those. In general, RL agents are still suffering a high amount of meta parameters, thus, scaling these algorithms will be the next logical step, so that these meta parameters can be included in the machine learning pipe line. Tensorflow provides here a great tool to achieve this task.

One possible route would be to parallelize DDPG algorithm. Also, the A3C algorithm can be improved. Here, trust region policy optimization to optimize the neural network could be a promising route, because it shows promising results when a lot of correlated actions are present (Schulman & al, 2016; Schulman & al, 2015).

Next, to the agents, using Docker to host the environment on Amazon's Elastic Compute Cloud was very to be successful. It allowed for easy parallelization, scaling, and very importantly for easy transfer, conservation, and communication of the agents and results (Figure 10). The initiative by OpenAI with Universe also provides a good direction. Further, capitalizing on this scalability, will provide a huge benefit in the development of further agents.

In general, using a game like Torcs, and in the next step a even more realistic game such as GTA V provides a great simulation engine for training, testing and benchmarking AI agents.

References

gym_torcs. (2016). Von https://github.com/ugo-nama-kun/gym_torcs abgerufen

http://torcs.sourceforge.net/. (2017).

Juliani, A. (2016). Von https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-8-asynchronous-actor-critic-agents-a3c-c88f72a5e9f2 abgerufen

Lau, B. (2017). Von https://github.com/yanpanlau/DDPG-Keras-Torcs abgerufen

Lillicrap, & al, e. (2016). Von https://arxiv.org/pdf/1509.02971.pdf abgerufen

Loiacono, D., & Cardamone, L. L. (2013). Von https://arxiv.org/pdf/1304.1672.pdf abgerufen

Minh, & al, e. (2015). *Nature*.

Minh, & al, e. (2016). Von https://arxiv.org/pdf/1602.01783.pdf abgerufen

Schulman, & al, e. (2015). Von https://arxiv.org/abs/1502.05477 abgerufen

Schulman, & al, e. (2016). Von https://arxiv.org/pdf/1506.02438.pdf abgerufen

Sutton, R. S., & Barto, A. G. (2012). *Reinforcement Learning.* Second edition, in progress.

VizDOOM. (24. 04 17). Von http://vizdoom.cs.put.edu.pl/ abgerufen