Project description

In this work, we want a deep neural network to learn basic car driving actions from a human driver. A huge database containing some hundred thousand kilometers of video data on public roads including all vehicle signals is available for this task. Input to the network shall be the video stream plus vehicle signals such as speed and steering wheel angle. Desired network output is the predicted driver action within the next 1 or 2 seconds: steering left/right/straight, braking, accelerating. For example, we want the network to predict the driver braking at a red traffic light or in front of a stopped vehicle, or to steer around a curve ahead.

Learning of rare events and processing of sequential data are among the sub-tasks and challenges of this project.

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Control Engineering and Information Technology

End-to-End Asynchronous Advantage Actor-Critic Reinforcement Learning for Autonomus Vehicles

Master's project laboratory II

Student	*Ext. Supervisor*	*Int. Supervisor*
András Kalapos	Róbert Moni Continental Automotive Hungary Kft.	Dr. István Harmati

May 20, 2019

Contents

1 Introduction

On December 5 2018, Waymo (formerly the Google self-driving car project) launched its first commercial self-driving taxi service, so far only available to a limited number of test users [1]. Many other companies are pursuing fully automated vehicles and even more are targeting the development of advanced driver-assistance systems. Autonomous vehicles promise several improvements in transportation, one of the most important one being safer and more reliable than human drivers, especially in boring situations like highway commutes. Automated taxis, trucks and vans don't need regular breaks as human drivers do, enabling faster, and potentially cheaper, more profitable rides and deliveries. However, usually, downsides of automated cars are seldom mentioned, apart from the general public fear of automation. The advances of self-driving car technology also increase the use of cars over public transportation which has an obvious negative impact on the environment.

In this project, we study deep learning techniques applied to vehicle control problems, which involve generating control signals such as steering, gas and brake, based on images of a (vehicle-mounted) camera. Even though a car only has two independent degrees of freedom, constructing a machine learning model which is capable of predicting good throttle and steering commands in a wide variety of environments is a very complex task. Today several production cars are capable of performing similar tasks (on varying levels), but still, it is worth researching these technologies on a basic level to gain experience which will help us develop more complex systems in the future. One key difference of our approach to widely used methods is that these have explicit lane detection models first, realised by training on large manually annotated data. End-to-end approaches as the one in this report do not need such large annotations.

Reinforcement learning is one of three branches of data-driven modelling, and learning algorithms along with supervised and unsupervised learning. The core idea behind reinforcement learning is to learn by trial and error, similar to how we, humans learn many of our basic abilities. The key benefit of this approach over supervised learning, it that the agent itself collects the training data via exploration of the environment, eliminating the need for large annotated datasets. In this report, we will investigate such algorithms.

The remainder of this report is organised as follows: Chapter 2 briefly introduces previous works on neural networks applied to automated driving problems and includes a basic overview of modern reinforcement learning methods. A theoretical description of the chosen approach is given in Chapter 3 and Chapter 4 presents the experimental works and results of this project.

Code for this project can be found at `https://github.com/kaland313/A3C-CarRacingGym`

[1] `https://www.theverge.com/2018/12/5/18126103/waymo-one-self-driving-taxi-service-ride-safety-alphabet-cost-app`

2 Related works

2.1 Neural networks for end-to-end driving

One of the earliest works on end-to-end vehicle control, more precisely steering was ALVINN (Autonomous Land Vehicle in a Neural Network) [14], which was published in 1989 and presented a system consisting of a 32x30 pixel camera and a 32x8 pixel laser range finder as an input and a fully connected neural network for predicting steering commands.

The main sensors of self-driving systems today are cameras and lidars which produce image-like outputs. Generally, convolutional neural networks offer one of the most suitable options in machine learning based image processing tasks (image classification, segmentation, object detection), thus many self-driving systems use these for input processing. Many image segmentation networks are fully convolutional (e.g. [1]), but in many other problems, convolutional networks are followed by dense and recurrent modules acting as regressors, classifiers or performing any other higher level task.

As driving is a process happening through time it is logical to build machine learning models which are capable of incorporating inputs from previous states of the surrounding environment and the vehicle. This also opens up the possibility to model higher level driving actions like changing lanes, overtaking a vehicle ahead, turning in a junction, or a roundabout, using acceleration and deceleration lanes, etc. For example considering a lane-changing manoeuvre (which is the simplest of the above-mentioned ones) a lane keeping algorithm would not "understanding" the whole process, and would try to steer the car back to its original lane. Modular automated driving systems might be able to tackle these situations, for example, if the learning part of the algorithm would only do lane detection and localisation, a traditional algorithm can be programmed to do lane changing. Deep learning methods offer are very flexible, scalable and can provide a rich representation of the functions they model making them the most promising candidate to address these complex situations in end-to-end systems. To incorporate temporal clues 1D convolutional (convolution throughout time), recurrent or long short-term memory modules/layers have to be utilised. The downside of using more complex models is the increasing difficulty of their training and the exploding size of their hyperparameter space.

2.2 Reinforcement learning methods for training neural networks, deep reinforcement learning

Recent advances in the field of deep reinforcement learning led to some outstanding results demonstrating the capabilities of this technique. DeepMind's AlphaGo [15] and it's successors, AlphaGo Zero and AlphaZero not only beat the previous state of the art Go algorithm but even World Go Champions Lee Sedol and Ke Jie, in the game which has long been viewed as the most challenging of classic games for artificial intelligence. Even more striking is that the most recent version, AlphaZero beat it's predecessors even though

no human knowledge was used to train it, it only learned through self-play [3]. Many computer games such as the Atari 2600 games can be mastered by deep reinforcement learning algorithms on superhuman levels, e.g. using the DQN algorithm [10] or more recent asynchronous methods such as A3C [11]. In significantly more complex games, like Dota2 agents outperforming the words best human players exist: OpenAI's algorithm [12] defeated the world's top professional payer in a 1v1 game and also recently the world champion team in a 5v5 game [2] (with some limitations to the heroes that the AI can choose). The later result requires the cooperation of 5 agents against a team of humans who mastered to play together.

Deep neural networks can be generally viewed as good function approximators, thus they provide a more flexible, richer, more manageable representation allowing reinforcement learning algorithms to perform more effectively compared to training non-neural-net based models. Countless (deep) reinforcement algorithms exist, fig. 1 shows a non-exhaustive but useful classification of modern methods from OpenAI [13][3]. In this report, A3C will be presented in detail with some other methods mentioned later in this section.

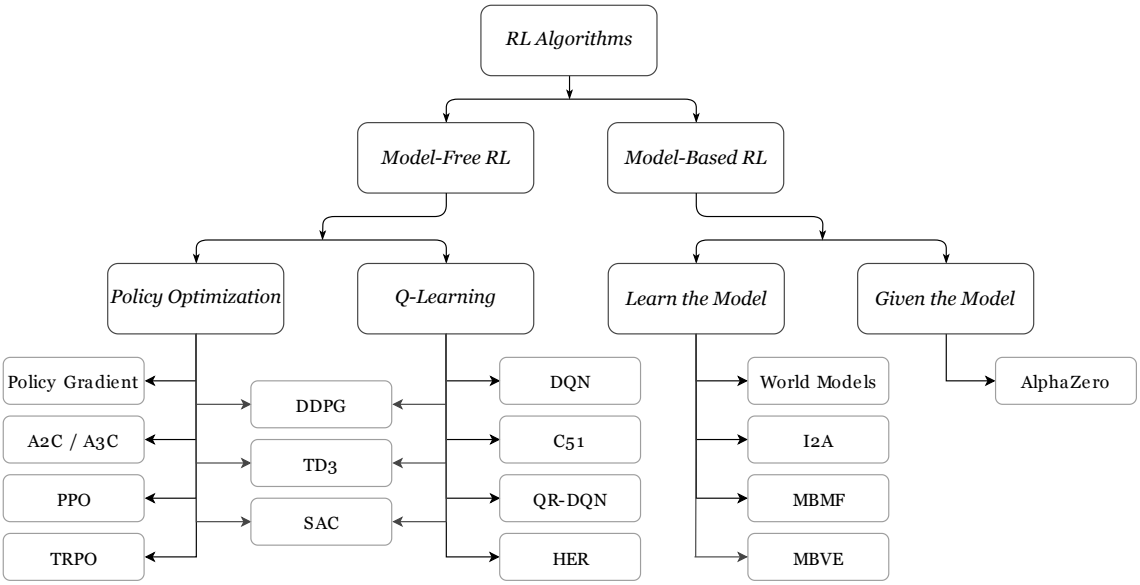


Figure 1: *A non-exhaustive but useful classification of modern reinforcement learning methods from OpenAI [13]*

One of the reasons why deep reinforcement learning methods are being applied to more and more problems is their data efficiency. Supervised learning methods rely on large datasets which usually requires expensive and time-consuming collection, cleaning, annotation of the data. However reinforcement learning methods usually don't require a dataset at all, they learn by interacting with an environment and extract the information they need. The process of learning through interaction is closer to the way we, humans learn, and by assigning the task of exploration to the agent (instead of the humans who create the dataset) it is possible to achieve better an optimal solution to many problems. With supervised

[2] `https://openai.com/blog/how-to-train-your-openai-five/`

[3] Copyright ©2018 OpenAI (http://openai.com)

learning, the performance of the trained models heavily depends on the quality of the data and how well it represents the optimal solution to the problem. On the other hand in most cases training reinforcement learning algorithms is only possible or practical in a simulation, because the amount of trial and error an agent goes through during its training is too time-consuming or expensive in a real-world environment (e.g. one of OpenAI's Dota2 algorithms was trained on 180 years of gameplay [12]). The downside of training in a simulation is the limits to the real world use of the trained agents. To solve this problem, either the simulations need to be sufficiently realistic, or the transfer learning problem of adapting a trained agent to the real world has to be solved efficiently. Another challenge of using reinforcement learning is usually the instability of the training process.

3 Asynchronous advantage actor-critic algorithm

Among multiple asynchronous reinforcement learning methods proposed by Mnih et al. [11] asynchronous advantage actor-critic (A3C) performs the best on many Atari 2600 games both in terms of training speed and the achieved score. All of the proposed methods execute parallel agents in separate environments, each of which accumulates gradients over multiple time steps to be used to update the parameters of a global network (in a mini-batch-like manner). Using multiple agents reduces training time and the diversity of their experience stabilises training. Many earlier deep reinforcement learning methods (e.g. DQN) rely on experience replay to stabilise the training. The use of multiple agents in the asynchronous methods eliminate the need for this which opens up the possibility to apply several reinforcement learning algorithms on deep neural networks. It is also possible to run these algorithms on a single machine with a multi-core CPU which is a huge benefit over many earlier techniques which relied on massively distributed architectures or GPUs. The authors claim that A3C is the most general and successful reinforcement learning agent to date because it performs well both on continuous and discrete action spaces and is capable of training feedforward and recurrent networks as well.

The A3C method's multiple agents/workers run parallel to each other and a process managing the global model. Each agent has a copy of the global model and interacts with its own environment. After t_{max} number of steps or upon reaching a terminal state of the environment they calculate the gradients of the objective function and then send them to the master process which performs the parameter update on the global network and transmits the updated parameters back to the worker. This process is illustrated by fig. 2 and is explained in more detail on the following pages.

The asynchronous advantage actor-critic (A3C) algorithm's naming can be explained by the following components:

Asynchronous because the workers send the gradients to the global network asynchronously, i.e. if a worker took t_{max} steps or reached a terminal state it immediately tries to send the gradients to the global network. The parameter update is done in locking mode, so if the

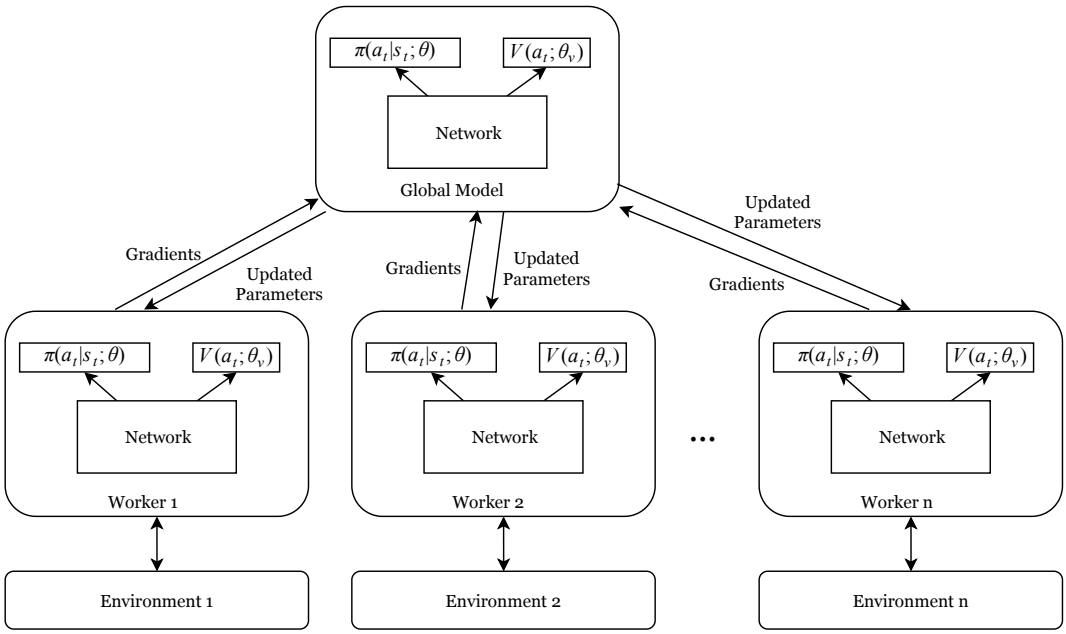


Figure 2: *Parallel workers and the global model of the asynchronous advantage actor-critic algorithm*

global network is interacting with a worker, another worker attempting to send it's gradients to the master has to wait until the previous update is finished. Whenever the global model is updated its new parameters are copied to the worker's model, but only to the one which sent the gradients. In contrast to A3C, the method called advantage actor-critic (A2C) does the parameter updates synchronously. The algorithm waits until all workers reach a stop condition (took t_{max} steps or reached a terminal state) the global model is updated and then the workers continue with the updated parameters.

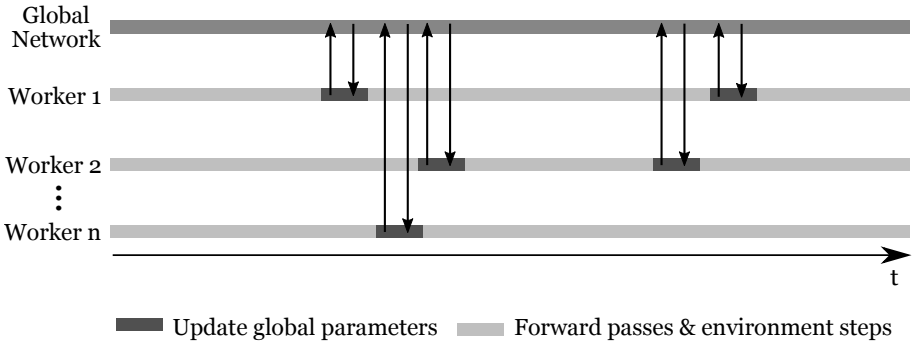


Figure 3: *Illustration of the asynchronous nature of global parameter updates*

Actor-critic because the learned model outputs a policy $\pi(a_t|s_t;\theta)$ and an estimate of the value function $V(s_t;\theta_v)$. The agent is acting according to its policy, thus it can be considered as the actor part, and the value estimate can be viewed as the critic (see fig. 4). Note that even though θ and θ_v appear to be separate parameter sets, in practice some of the parameters are shared between the two models. For discrete action spaces the policy $\pi(a_t|s_t;\theta)$ is usually implemented as a softmax output of an artificial neural network and the $V(s_t;\theta_v)$ as a linear output.

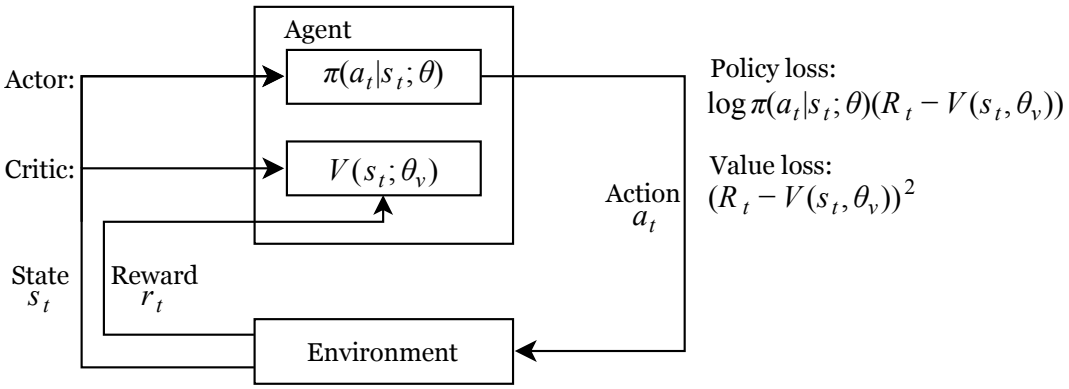


Figure 4: *The actor and the critic components of the agent highlighted in the agent-environment loop*

Advantage because the gradient used to update the policy includes a weighting factor $A^\pi(s_t, a_t) = Q^\pi(a_t, s_t) - V^\pi(s_t)$ which can be viewed as the estimate of the advantage function. The advantage is the difference of the expected reward if we take action a_t in state s_t and then follow policy π compared to if we would follow the π policy immediately. Intuitively, the advantage function is a measure of how good is taking action a_t over following π policy at a given time step.

The agents operate in the forward view, meaning that at any given step t the return R_t is estimated as the cumulative n-step return:

$$R_t = r_t + \gamma r_{t+1} + ... + \gamma^{n-1} r_{t+n-1} + \gamma^n V^\pi(s_{t+n}) = \sum_{i=0}^{n-1} \gamma^i r_{t+i} + \gamma^n V^\pi(s_{t+n})$$

where r_i is the reward received after each step, $\gamma \in (0, 1)$ is a discount factor used to weigh immediate rewards more and $V^\pi(s_{t+n})$ is the value estimate at step $t + n$.

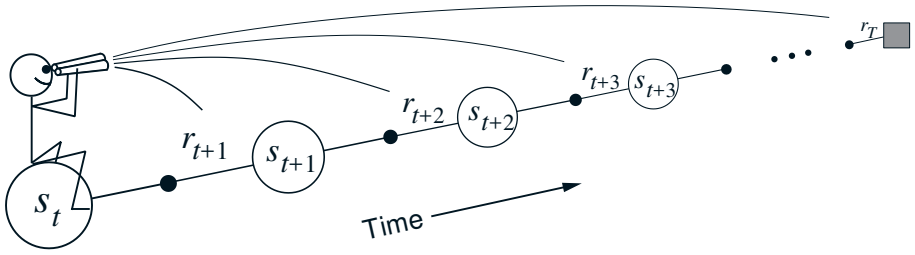


Figure 5: *An intuitive representation of forward view by Sutton and Barto [16]. The returns and model updates are calculated by looking at future rewards and states.*

In practice $Q^\pi(a_t, s_t)$ is estimated by R_t and the estimate of the $A(s_t, a_t, \theta, \theta_v)$ advantage is calculated by the following formula:

$$A^\pi(s_t, a_t) = Q^\pi(a_t, s_t) - V^\pi(s_t) = R_t - V^\pi(s_t) = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}, \theta_v) - V(s_t, \theta_v)$$

where $k \leq t_{max}$ is the number of steps the agent took to collect experience (observations,

rewards and actions leading to them).

Objective function

Each worker follows policy $\pi(a_t|s_t;\theta)$ for t_{max} steps or until terminal state is reached in the environment. Afterwards the policy and the value functions are updated by the master thread. The loss function for updating the weights is formulated as the sum of the value loss and the policy loss. The value loss is usually calculated as the mean squared error of the value function and the R_t reward. The gradient for the policy objective function can be expressed as

$$\nabla_{\theta'} \log \pi(a_t|s_t;\theta') \cdot (R_t - V(s_t, \theta'_v)) + \beta \nabla_{\theta'} H(\pi(a_t|s_t;\theta'))$$

where θ' and θ'_v are the policy and value function parameters of the particular agent, $H(\pi(a_t|s_t;\theta'))$ is the entropy of the policy and β is a hyperparameter, the weight of the entropy regularisation term. Note that the weighing term $R_t - V(s_t, \theta'_v)$ is the estimate of the advantage function.

The entropy of the policy $H(\pi(a_t|s_t;\theta'))$ is added to the policy loss as a regularisation term to encourage exploration and prevent convergence to suboptimal deterministic policies. For discrete policies it is calculated as

$$H(\pi(a_t|s_t;\theta')) = \sum_{i}^{|\mathcal{A}|} -\pi(a_i|s_t;\theta') \cdot \log \pi(a_i|s_t;\theta')$$

where $|\mathcal{A}|$ represents the number of actions (for each a_i action $\pi(a_i|s_t)$ gives the probability of taking that action in state s_t, i.e. $\pi(a_i|s_t)$ can be thought of as a vector of softmax probabilities with length $|\mathcal{A}|$).

The pseudocode of the algorithm according to Mnih et al. [11] is presented as Algorithm 1.

Algorithm 1 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ *and* θ_v *and global shared counter* $T = 0$
// Assume thread-specific parameter vectors θ' *and* θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial \left(R - V(s_i;\theta'_v)\right)^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

4 Implementation and experiments

As part of the track to develop end-to-end driving models, we initially we aim to simplify the problem as much as possible and gradually work towards more advanced solutions. The stability of training deep reinforcement agents is generally challenging and the complexity of the environment determines the complexity of a well-suited agent. With complexity comes more resource-demanding training and more hyperparameters to be optimised. Owing to these reasons we will use the reasonably simple CarRacing-v0 Gym as our environment and the reasonably simple network architecture also used on several Atari games by Mnih et al. [11]. As considering an environment or a network reasonably simple is quite a relative and subjective classification, a more objective description would be that the complexity of CarRacing-v0 is comparable to that of the Atari 2600 games (both in terms of observation space and action space).

4.1 Environment

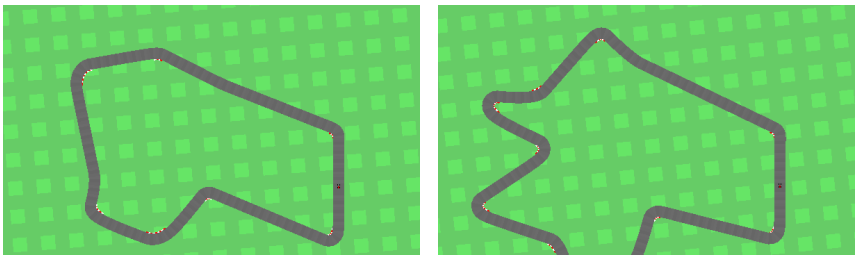
As earlier mentioned the training of reinforcement learning agents is practical only in simulated environments due to the cost, time requirement, and added complexity of interacting with the real world. Simulated environments provide a more controlled, more flexible and quicker (both in terms of set up and training time) option for training. Reproducing such environment is also much easier enabling easy comparison of different approaches and results. As the simulation has full knowledge of the environment (in order to run itself) the

calculation of the reward is simpler too.

Throughout this semester project, the OpenAI's CarRacing-v0 Gym [2] was used to evaluate the A3C algorithm. The main reason for choosing this environment is its similarity to the general problem we are aiming for, image input based control of vehicles and the availability of similar reports (Khan and Elibol [8], Jang et al. [7]) to this one about the same environment, algorithm and model, which can be used to compare our results to.

The OpenAI Gym environments were created to fill a role in the field of reinforcement learning research similar to the benchmark datasets such as MNIST, CIFAR10, etc. Different approaches, algorithms can be trained and evaluated in these environments and then compared based on their achieved score. Every gym has a clear version labelling to make the comparison of results unambiguous. Apart from the benchmark role, the gyms implement a universal interface for reinforcement learning agents, which since then was adapted for many more environments. This allows for easy adaptation of an implemented algorithm for many environments.

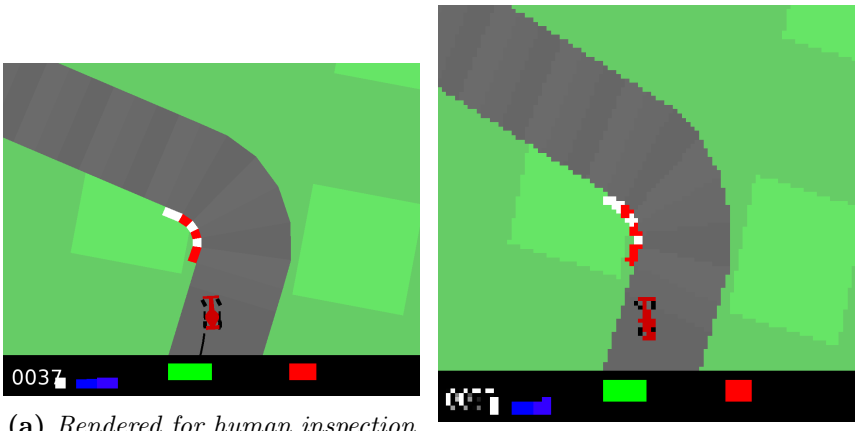
The CarRacing-v0 environment is a simple top view car racing game with only a single race car appearing on the track, the one which is to be controlled. The tracks are randomly generated, but the car always has to complete it in a clockwise direction, two example tracks are shown on fig. 6. The environment shows two different views to a human observer and to the agent controlling it as shown of fig. 7. The human view is much higher resolution and contains more details like skid marks and some indicators on the bottom. The state of the game which is to be used as the input of our agent has a resolution of 96x96 pixels and is shown on fig. 7b. Input to the environment are the steering, gas and break signals of the car encoded as continuous values in ranges $[-1, 1]$, $[0, 1]$, $[0, 1]$ respectively. Discrete control of the car is also possible by only using a set of certain input combinations, as will be presented later. *Reward is -0.1 every frame and +1000/N for every track tile visited, where N is the total number of tiles in track. For example, if you have finished in 732 frames, your reward is 1000 - 0.1*732 = 926.8 points. Episode finishes when all tiles are visited. Some indicators shown at the bottom of the window and the state RGB buffer. From left to right: true speed, four ABS sensors, steering wheel position, gyroscope.*[4]



(a) *Track with lighter turns* **(b)** *Track with sharp U-turns*

Figure 6: *Randomly generated tracks of the CarRacing-v0 gym*

[4]Quoted from: https://gym.openai.com/envs/CarRacing-v0/



(a) *Rendered for human inspection*

(b) *Rendered as the state: "What the agent sees"*

Figure 7: *Appearance of the CarRacing-v0 gym*

4.2 Network architecture and training

Similarly to Jang et al. [7] and Khan and Elibol [8] we used the same fully connected convolutional network architecture and input reprocessing as Mnih et al. [11]. As most convolutional networks with single or few numeric outputs the one used by [11] has a similar architecture to *LeNet* [9]: it consists of a feature detector, which uses convolutional layers followed by fully connected layers.

The input to the network is formed by converting states to grayscale and stacking 4 consecutive ones along the 3rd ("depth axis") to encode some information about the speed and acceleration of the vehicle in the input. The networks consist of two convolutional layers with followed by a dense layer and two outputs for the policy and the value function. The first convolutional layer has 16 filters of size 8×8 with stride 4 followed by another one with 32 filters of size 4×4 with stride 2, both used ReLU as their activation function. The output of the second convolutional layer is flattened and fed into a dense layer with 256 units. Both output layers are connected to this dense layer, the policy output is formed by a dense layer with as many units as the number of discrete actions with softmax activation function and the value estimate is a single neuron with linear activation. The network architecture is presented on fig. 8.

As earlier mentioned our agent uses discrete actions to control the car, similar to controlling it using the arrow keys of a keyboard. The allowed input combinations are shown in table 1. As the policy, the network outputs the $\pi(a_t|s_t)$ softmax probability for taking a particular discrete action given the input (the last 4 states received from the environment). During training and evaluation, the action with the highest probability is selected, the corresponding continuous action combination is multiplied with the probability and the resulting action is used as the input to the environment for getting the next state. Multiplying the continuous values with the probability results in less aggressive actions, helping to stabilise the training according to [7]. The complete agent-environment loop including all major operations on the data is illustrated by fig. 9

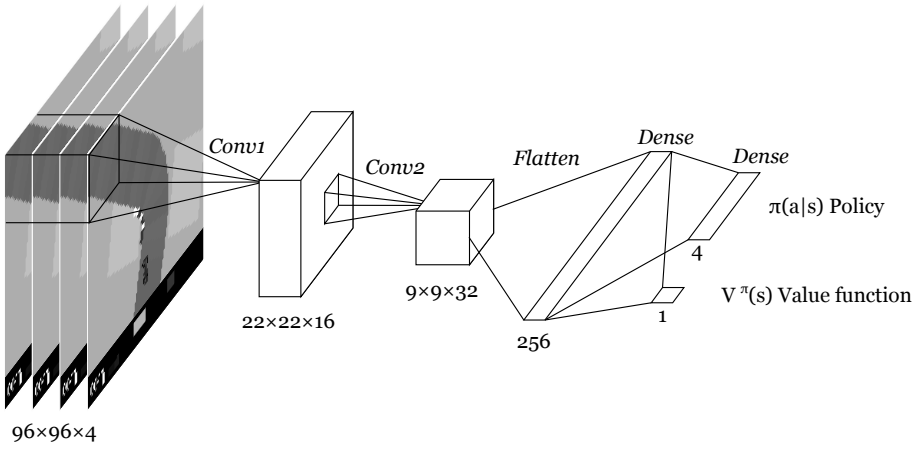


Figure 8: *Network architecture*

Table 1: *Discrete actions with corresponding continuous input combinations. Note: The brake value is* 0.8 *instead of* 1.0*, because the later one would mean complete blocking of the wheels, leading to the instability of the vehicle.*

Action	Steering	Gas	Brake
Accelerate	0	**1**	0
Brake	0	0	**0.8**
Steer left	**-1**	0	0
Steer right	**1**	0	0

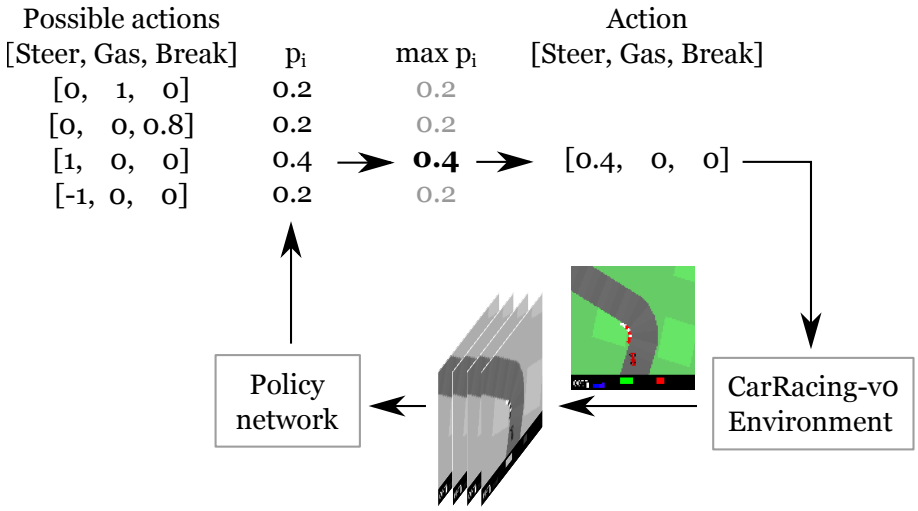


Figure 9: *Agent-environment loop*

The model was built using Tensorflow, mostly via the `tf.keras` API and using eager execution. The A3C algorithm was implemented in python mostly based on the code of [17] and [8].

We used the RMSProp optimiser with an initial learning rate of $10^{-}3$, which was linearly decayed to 0 over the training process. To allow further improvement of the model the training was restarted by loading the weights of the trained model and changing the initial staring learning rate to $10^{-}4$. The β entropy regularisation coefficient was 0.01 throughout

the full training and value loss was weighted with a 0.5 coefficient, resulting in a total training loss: $L = L_{policy} + 0.5L_{value} - 0.01Entropy$.

To reduce the chance of exploding gradients we applied gradient clipping, before updating the network parameters the gradient norm was clipped to be less than 40. We also experimented with reward clipping, each step the reward received from the environment was clipped to the range $[-1, 1]$. However, this mostly affects the reward if the vehicle leaves the green area of the environment, in which case it receives a negative reward of -100. Otherwise, the reward is -0.1 at each step or a positive number in the range around 1 to 10. Our experimental results show no benefit of using reward clipping.

4.3 Results and evaluation

The training was carried out on an Amazon Web Services EC2 server running a 16 threaded Skylake-SP Intel Xeon processor with a sustained all core Turbo CPU clock speed of up to 3.4GHz. With this machine up to 15 (maybe 16) workers and one master process could run efficiently (we didn't test whether the 15 or 16 worker setup was more efficient, simply used only 15 workers all times).

The model was trained for 15 000 episodes in total, resulting in an average score of 625, whit standard deviation of 214, calculated over 100 test episodes. This result along with some other results found in the literature is displayed on fig. 10. The best performing method, World Models [4] uses significantly more complex model both in terms of parameter count and architecture. It uses a variations autoencoder and a recurrent network to not only learn the control task of the environment but the full dynamic model of it.

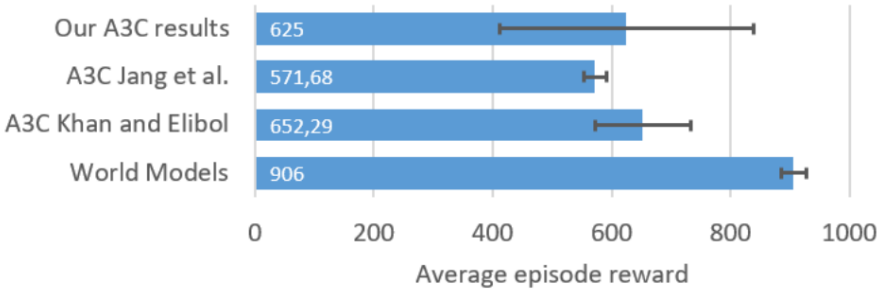
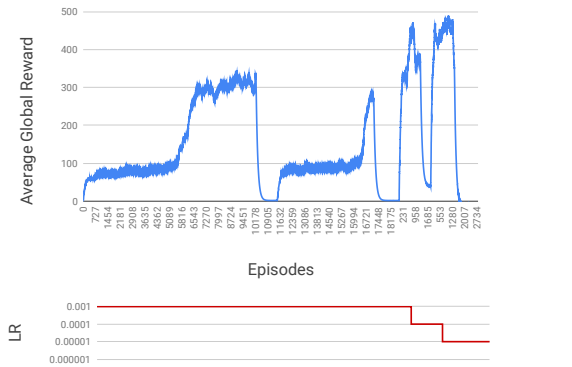


Figure 10: *CarRacing-v0 scores achieved using various methods: World models by Ha and Schmidhuber [4] and two other A3C implementations by Jang et al. [7] and Khan and Elibol [8]*

The moving average episode reward during the first 10000 episodes of the training and the learning rate is shown on fig. 11a. By manually restarting the training and adjusting the learning rate in a step-wise manner slightly higher average reward was achieved in the first 10 thousand episodes as shown on fig. 11b however the main reason for manually restarting and adjusting the training was trying to cope with the instability, which causes the reward collapses. This problem is addressed in more detail in section 4.4.

(a) *Training with learning rate annealing*



(b) *Training with manual restarting and learning rate adjusting*

Figure 11: *Global moving average reward during two different trainings. Note that at any point where the learning rate abruptly changes, the training was restarted manually.*

After experimenting with several modifications to the training the model which was the result of the most stable training was further trained for 5000 episodes. During these extra episodes, the entropy regularisation coefficient β was reduced to 10^{-4} and L2 weight regularisation was applied to the network. The complete training history for this training is shown on fig. 12. As the training was the most stable without early termination the 15 thousand episodes took a total of 10.6 hours and 14 million steps.

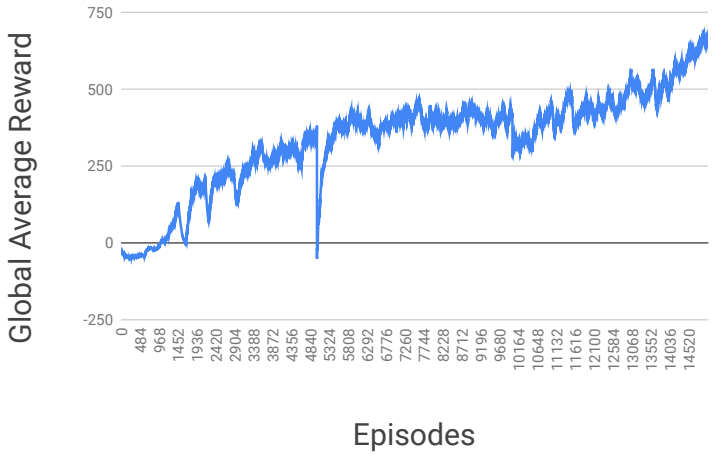


Figure 12: *Global average episode reward over the course of the training of our best performing model*

4.4 Challenges

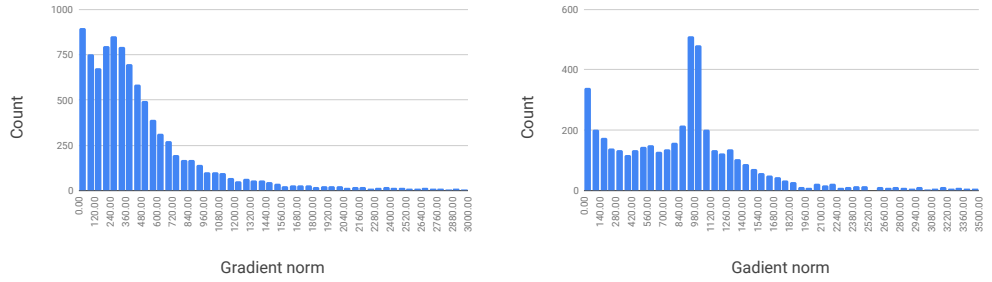
One challenge in implementing A3C was parallelising the workers and the global model because the CarRacing-v0 gym wasn't designed for multi-threaded use so we had to use multiple processes with `mpi4py`, a python library for the Message Passing Interface.

To speed up the training we utilised early termination, i.e. if the reward started to decrease

and fell below the maximum by 5 the episode was terminated. This allowed a 4-5 times speedup of training (with early termination 5000 episodes took 45 minutes compared to 3.5 hours without it). However this seems to lead to the instability of the training, but that could also be the result of higher learning rates because of the faster training process and the linear learning rate annealing.

Training instability, reward collapse

Probably the biggest challenge of the training was the occasional collapse of the reward (e.g. as can be seen on fig. 11b), which we believe is an instability in the training process due to exploding gradients. You can see the histogram of the gradient norms on fig. 13 acquired during the trainings shown on fig. 11. Clearly, the more unstable training on fig. 11b, has larger gradients compared to the more stable training on fig. 11a. Note that the valley in the reward on fig. 11a is due to the restart of the training not to instability.



(a) *Result of the training seen on fig. 11a* **(b)** *Result of the training seen on fig. 11b*

Figure 13: *Gradient norm histograms*

We tried multiple options to tackle the problem:

1. Automatically reloading the best model if the reward collapsed also provides an acceptable solution, but it doesn't truly solve the problem of instability. We applied this to all the options below as well, but it only made a difference if the reward collapsed.

2. By annealing the learning rate and selecting smaller initial learning rate the problem is less severe, but not completely eliminated, this was applied to the method below as well. Compare fig. 14a or fig. 11a to fig. 11b to see the effect of applying annealing.

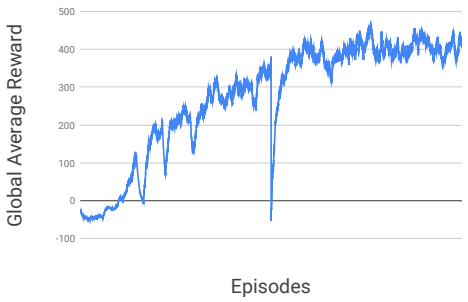
3. By not using early termination we didn't encounter a single event of reward collapse, but the training times were 4-5 times longer. Note that the valley in the reward on fig. 14b is due to the restart of the training not to instability.

4. L2 regularisation (fig. 14c) seems to reduce the chance of reward collapse, as it is supposed to cope with exploding gradients, however, in one occasion the reward still collapsed (not shown on the reward plots). It also improved the achieved score.

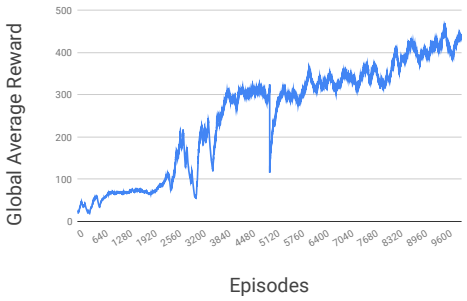
5. Batch normalisation (Ioffe and Szegedy [6]) with early termination didn't decrease the number of reward collapse events during training, in fact, it made it worse, moreover, it increased the training time.

Out of these, we conclude that the combination not using early termination (or maybe using it less aggressively), annealing the learning rate and applying L2 regularisation to the weights of the network stabilises the training and helps to avoid reward collapse.

(a) *Learning rate annealing with early termination*



(b) *Learning rate annealing without early termination*



(c) *L2 Regularisation with learining rate annealing*

Figure 14: *Attempts to stabilize the training and avoid reward collapse*

5 Future works

In the future, we plan to continue working on asynchronous advantage actor-critic reinforcement learning agents in the CarRacing-v0 environment and others as well, especially the Duckietown gym, and later on in the physical Duckietown system. With the network presented in this report we could work on stabilising the training and on hyperparameter optimisation, experiment with different values of the entropy regularisation coefficient β or change the amount of gradient clipping. However, the current network's architecture isn't complex enough to model the dynamics required to achieve as high scores as World models[4]. Thus we would like to train models with recurrent, LSTM layers or variational auto encoders as used in the PlaNet networks [5].

Our long term goal is to train reinforcement learning agents int the Duckietown Gym and apply them to the real physical Duckietown environment.

6 Conclusions

In the first section of this report a short introduction to neural network based automated driving and deep reinforcement learning was given. A detailed theoretical description of the asynchronous advantage actor-critic (A3C) reinforcement learning algorithm was presented.

A3C was implemented in Tensorflow and successfully used to train a convolutional network with parallel workers. Training and evaluation were carried out in the CarRacing-v0 gym environment. By using a feedforward convolutional network trained with A3C no published results surpass the best scores achieved in this environment by Ha and Schmidhuber [4]. However our best model achieves similar scores to other A3C implementations using this environment, thus we can conclude that the network was trained successfully.

References

[1] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *CoRR*, abs/1511.00561, 2015. URL `http://arxiv.org/abs/1511.00561`.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[3] DeepMind. Alphago. URL `https://deepmind.com/research/alphago/`.

[4] David Ha and Jürgen Schmidhuber. World models. *CoRR*, abs/1803.10122, 2018. URL `http://arxiv.org/abs/1803.10122`.

[5] Danijar Hafner. Introducing planet: A deep planning network for reinforcement learning, 2019. URL `https://ai.googleblog.com/2019/02/introducing-planet-deep-planning.html`.

[6] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL `http://arxiv.org/abs/1502.03167`.

[7] Se Won Jang, Jesik Min, and Jae Hyun Kim. Reinforcement car racing with a3c. 2017. URL `https://sites.google.com/view/jesikmin/course-projects/reinforcement-car-racing-with-a3c`.

[8] M. A. Farhan Khan and Oguz H. Elibol. Car racing using reinforcement learning. 2016. URL `https://github.com/oguzelibol/CarRacingA3C`.

[9] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998. ISSN 0018-9219. doi: 10.1109/5.726791.

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

[11] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016. URL http://arxiv.org/abs/1602.01783.

[12] OpenAI. Openai five, . URL https://openai.com/five/.

[13] OpenAI. Spinning up in deep rl, . URL https://spinningup.openai.com/en/latest/user/introduction.html.

[14] Dean A. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 1*, pages 305–313. Morgan-Kaufmann, 1989. URL http://papers.nips.cc/paper/95-alvinn-an-autonomous-land-vehicle-in-a-neural-network.pdf.

[15] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016. URL http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

[16] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998. ISBN 0262193981. URL http://www.worldcat.org/oclc/37293240.

[17] Raymond Yuan. Deep reinforcement learning: Playing cartpole through asynchronous advantage actor critic (a3c) with tf.keras and eager execution, 2018. URL https://medium.com/tensorflow/deep-reinforcement-learning-playing-cartpole-through-asynchronous-advantage-actor-critic-a3c-7eab2eea5296.