Project Laboratory report

Department of Telecommunications and Media Informatics

Author: **Fábián Füleki**
ffabi1997@gmail.com
Neptun code: **AEP0TG**
Specialization: **Infocommunications - HIT**
Consultant: **Róbert Moni**
robertmoni@tmit.bme.hu
Consultant: **Dr. Bálint Gyires-Tóth**
toth.b@tmit.bme.hu

Subject:

"AI Driver" - End-to-End Driver Activity Prediction

Task description:

In this work, I aim to create a self-driving system based on deep reinforcement learning. The input to the network shall be the video stream from a camera behind the windshield of the vehicle. The desired network output is the predicted steering wheel angle, braking force and acceleration for the next milliseconds. My final goal is to have an autonomous vehicle driving itself in a simulated environment. The further goal is to deploy the created agent on a real-world track or even participate in one of the related competitions, such as Duckietown.

2018/2019 Spring

1 Starting point of the project laboratory, previous works on the topic

1.1 Introduction

Self-driving is one of the most challenging problems of the automotive industry at the moment. Most of the car manufacturers and other technology companies are working on different types of advanced driver-assistance systems (ADAS). Every ADAS is aiming to lower the number of actions made by the driver, thus making the trip safer and less tedious. One key technology behind these systems is the neural network. Neural networks are widely used for image classification, stock price prediction, or even for natural language processing. The field of self-driving contains a lot of subfields, such as lane detection, traffic sign recognition, collision detection, and last but not least pedestrian movement prediction. These tasks can be solved separately with a few dedicated subsystems with or without utilizing a neural network. In this work, my goal was to create a deep neural network for solving the task of lane following.

1.2 Theoretical summaries

Before I introduce the different type of approaches that I have been experimenting with this semester, I would like to provide an overview of the topic of self-driving and deep learning as well.

1.2.1 Autonomous vehicles

Automating road transportation — even partly — has a lot of advantages. The first factor is safety: Even the simplest collision warning system can reduce the number of accidents on the road.

According to the SAE standard[9], six levels of automation can be differentiated by who is in control of the vehicle and who is monitoring the environment:

0. No automation: Full-time monitoring and execution are required by the human driver. A vehicle is on level 0 even if it is equipped with warning or intervention systems.

1. Driver assistance: The human driver is still completely in continuous control of the vehicle, but may be assisted by different systems for different tasks. In most cases, adaptive cruise control, lane keeping system and parking assistance are implemented.

2. Partial automation: In some cases, the onboard system can take control of the vehicle, but the human driver always has to monitor the environment and must take over if needed. Keeping the hands on the steering wheel and the eyes on the road are required. Dynamic driving is completely performed by the human driver.

3. Conditional automation: The autonomous system is in full control of the vehicle and can determine in which cases human intervention is needed. The driver has to be prepared to take over, mostly for the dynamic driving tasks, such as driving in a roundabout.

4. High automation: No human driver attention is ever needed for safety. The onboard system can manage to control the vehicle in most of the driving situations. If the human driver does not take control when needed, the vehicle can safely abort the trip.

5. Full automation: The vehicle can handle all of the driving situations as good as a human driver would.

There are different types of techniques to navigate a car. One of the simplest one is called lane-following, which utilizes infrared sensors placed above the ground. This is a perfect solution for guiding robots in a well-defined environment, i.e. a factory, although it cannot be used in a real-world environment on its own.

The most intuitive way of driving a car on public roads is using vision: human drivers are making decisions based only on their sight. The computer-based implementation of this approach is a forward facing camera used as a sensor. The video feed is processed by the onboard computer using traditional image processing techniques or neural networks. In this semester my goal was to use a deep neural network for the driving action prediction. As it turned out, there is an interesting, state-of-the-art solution for the steering prediction based on reinforcement learning called World Models[3]. It has been published less than a year ago, and I aimed to explore its capabilities.

Other types of sensors can be used for monitoring the environment. I only highlight their strengths and weaknesses, because I don't plan to use any of them later in this work.

Radar can be used to determine the distance of an object from the vehicle. It can be used in any weather condition, at high speed, and has a long range as well. The drawback is, that radar cannot determine the shape of an object precisely. A creative way of utilizing radar is annotation for camera images.

LIDAR is another technology for distance measurement, it creates a 3D map of the environment using a laser. LIDAR has higher precision, higher resolution, but shorter range than radar. LIDAR's biggest weakness is that it is relatively expensive and also it has to be cleaned quite often because dirt can pollute the sensor.

Ultrasonic sensors are mostly useful for parking assistance, but they can be used for blind spot monitoring and clearance detection at lane changing as well.

Adapting more than one type of sensors ensures that monitoring the environment can be achieved in all kind of conditions.

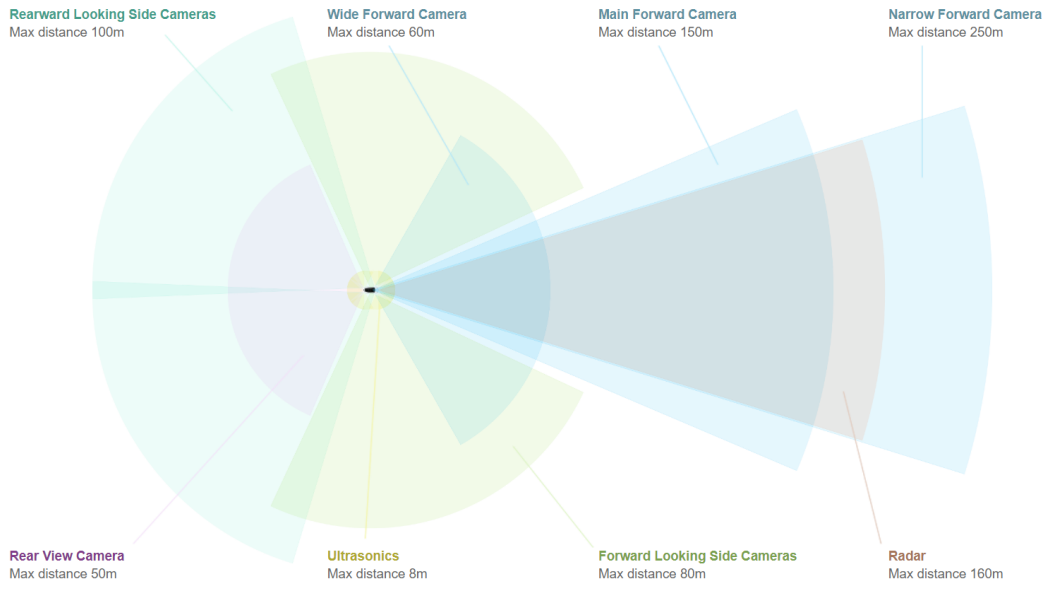

Figure 1: A modern vehicle utilizes different types of technologies at the same time[5]

1.2.2 Deep reinforcement learning

Some tasks, such as determining the species of an animal or identifying a voice sample are so complex problems, that they cannot be typically resolved using traditional techniques only. It would take too much work on the side of the developers and/or it would be exceedingly computational.

In the last couple of decades, neural networks have grown in popularity, and it started being a viable solution for many various kinds of problems. It has proved its capabilities in the fields of computer vision, natural language processing, voice recognition, and healthcare. Deep reinforcement learning is gaining performance as well, as DeepMind has shown with their AlphaGo[2] project.

It seems that maneuvering a car is a complex problem that could be solved by utilizing neural networks. Although deploying and testing self-driving systems in a real-world environment is easier said than done. Safety is the number one metric in the testing phase and at deployment as well. It goes without saying that no one should ever be harmed because of a runtime failure or a mistake made in the design of the system. At the moment, we cannot guarantee that a self-driving system purely based on neural networks will always perform as intended. There are a bunch of situations, where the system can fail and threaten the life of the passengers and/or the pedestrians.

A good workaround of these difficulties is creating a precisely simulated environment and utilizing reinforcement learning. The environment in most cases can be modeled by the Markov decision process, which consists of:

1. A set of possible states of the environment and the agent
2. A set of available actions in a state
3. The probability that an action in a given state will lead to another given state
4. The reward acquired after the transition from one given state to another given state

The reinforcement learning process is the following in brief:

1. The agent takes an action
2. The environment gets changed
3. An observation and a reward are returned to the agent
4. The agent changes its policy in some way

The agent can learn all the necessary features in the environment, and it can be later adapted to the real world.

There are two types of reinforcement learning algorithms, one which utilizes a model and one without. The goal of the model is to create a representation of the environment first, which can result in a faster learning for the agent.

Training an agent in a virtual environment has a number of advantages and one significant weakness. One benefit is that the simulation can run at any speed, regardless of the laws of physics. The agent makes a decision and the environment can provide the observation almost immediately. The other advantage is that the simulation can be adjusted as needed, i.e. the difficulty of learning rare events can be solved effortlessly. The environment can generate and label training sets as well if required. The disadvantage of this technique is that the real world cannot be precisely and completely recreated, so domain adaptation has to be applied at the end of the training.

1.2.3 Starting point of the project laboratory, previous works on the subject

In the matter of machine learning, the VITMAV45[5] course has showcased exceedingly well the theory behind deep learning for me, although I have never implemented a reinforcement learning agent before.

At the start of the semester, I did not receive any preceding work done on this topic.

2 Research and development performed on the topic during the semester

2.1 The motivation for self-driving and deep reinforcement learning

Before I go into the details of the topics is have been working on during the semester, I would like to provide an overview of my personal motivations, and the reasons behind the choice of making research on self-driving and deep reinforcement learning.

In the last few years, the topic of self-driving cars was really popular among tech enthusiasts, a lot of companies try to make better and better results in this field and I was really curious about the technology that made those achievements possible. During this semester I was constantly updating my knowledge about self-driving related researches, approaches and achievements. For example, a few weeks ago Tesla Inc. released a lot of technical details about their methods of self-driving[6], it is definitely worth a look. On the other hand, I met some other deep learning based researches for driving, e.g. driver attention monitoring.

In the preceding semester, I was studying about deep learning and I found it's concept fascinating enough to choose one of its subtopics (over vehicle-to-everything communication, which technology will be also widely used in the far future in my opinion). The motivation behind the reinforcement learning method was that I have never implemented any agent or any kind of evolutionary algorithm before, although I have been studying about it in the previous semester and I find it a captivating subject as well.

2.2 Initial research on reinforcement learning approaches

In the first month, I spent my time reading several articles and publications about reinforcement learning, and deep learning in general. I was looking for a specific project that I could use as a starting point or as a baseline solution. During this period, my consultant recommended a model-based reinforcement learning method called World Models. The research paper has been published in March 2018, by David Ha and Jürgen Schmidhuber. I found an implementation based on Keras for the publication, that I could use later on. I spent a week on understanding the key elements of the publication.

World Models consists of three neural networks:

1. Variational Autoencoder (VAE)[7]
2. Recurrent Neural Network with Mixture Density Network (MDN-RNN or RNN)
3. Controller (C)

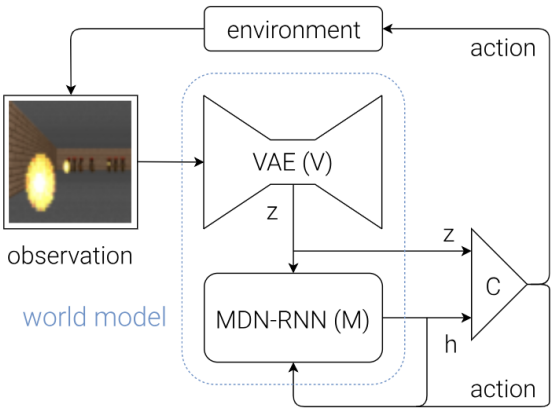

Figure 2: The described structure of the World Models

In short, the world model part consists of the VAE and the MDN-RNN, which both are used to create some kind of description of the environment. The controller is making decisions based on these representations. The VAE and the MDN-RNN are trained supervised and the Controller is trained with reinforcement learning. On the figure above, 'z' is the latent vector of the VAE and 'h' is the hidden state of the RNN.

I found this concept really interesting because self-driving is one of the problems where the system's input is video data, which has to be processed both spatially and temporally. If we take a look at processing a photo, convolutional neural networks are usually utilized, and in the case of temporal data, i.e. speech recognition,

recurrent neural networks are in use. World Models combines these two methodologies by letting the VAE create a lower dimensional representation of the environment (also knows as a latent vector) and using the RNN for taking care of the temporal changes in the latent vector.

World Models has another key element called Dreaming. Dreaming is a method that utilizes the RNN and the VAE a little different than described before: this time the RNN is generating the latent vector for the next frame based on the previous frames and the taken actions and the VAE is only used for visualization (decoding the latent vector). As we can see, the agent is not communicating with the simulated environment at all, thus comes the method name Dreaming. As this approach considers a properly trained VAE and RNN, I could not yet focus on it in the semester at all.

2.3 Preparations needed before the development

Before I could start any work on the subject, I had to set up my own developer environment. At first, I wanted to run the implementation of World Models 'as is', to see if every aspect of the project was working as intended. After a few small changes made to the source, I was able to run every part of it. This was important for me because of the size of the project: it consisted of roughly 2000 lines of Python code, blank lines not included. Completely recreating such a big project would have required a full focus on implementation. As I did not want to lose the research part of the project, I decided to use as much of the codebase as possible and implement some useful tools or crucial parts only.

One important thing that I had to keep in mind is that the training of a deep neural network is computationally expensive. I tried to train the VAE on my desktop machine which was way less powerful than my development workflow dictated. The other backside was, that I was not able to effectively use my computer during training.

Luckily, the TMIT department owns some multiple times faster computers which I was allowed to use for my research. The only criteria were that I had to run the training inside a Docker container. Containerization can be really useful later on as well if I need to move to another host machine. There is a disadvantage of using a container and that is real-time inferencing. My attempts were not successful for setting up the display for the container, so I had to run the inferencing on my computer. Luckily, inferencing requires way less computational power, and my assumption is that even a Raspberry Pi B would be capable enough for it (which is actually required if I want to run the agent in a real-world environment). I spent a week setting up the container to work well, and after a few weeks, I was able to run the training.

2.4 About the simulated environment

At the start of the project, I decided to focus on one and single simulation provided by OpenAI Gym called CarRacing. This environment provides observation as a bird's eye view of a car on a road on a grass field, as seen on the figure below. I slightly modified the environment: I removed some different color grass for some speed advantage of the data generation. This modification does not influence the final results as Autoencoders are exceedingly well used for noise reduction and those grass tiles would have been eliminated anyway.

One of the most important aspects of an environment is the reward. This environment provides +1 point after every visited tile and -0.1 points after every frame rendered. If the agent drives outside the bounds of the environment, it receives a reward of -100 points. The episode is over if the agent drives through all of the tiles or reaches the bounds. The driving task is considered solved if the agent can score over 900 points on average. In the paper of World Models, the authors claim that their agent was able to achieve this limit for the first time ever. The possible actions in this environment are the following: the agent can accelerate, brake and turn left or right.

2.5 Generating training data from the environment

For the training of the VAE, I needed frames exported from the environment. These frames should reflect as much of the possible situations as possible. For the generation, random rollouts are considered as the only possible approach. I modified this aspect of the project as the following: One episode consists of 230 frames, from which the first 30 frames are not saved because I found that almost every rollout had the same starting. In the first 60 frames, the car accelerates and does not turn. After that, every 5 frames a random action is chosen with the given chances: 40% for accelerating; 30% for turning left; 10% for steering right; 10% for braking; 10% for doing nothing. The acceleration and the steering angle is a randomly generated number in the proper range. It is not obvious why turning left has a way higher chance: the environment provides a track on which the vehicle is going around counterclockwise. Thus, there are more left turns than right, and also in my experience, the first turn is always a left turn. Also, I ensured that the car won't start drifting sideways by setting the gas to 0 at the turnings (I found the grass really slippery). This configuration is specific to this environment and does not affect

the performance of World Models. The actions are also saved because they will be needed for the training of the RNN. The overall size of the generated data became so big, that it caused some trouble for me: loading the training data into system memory was not possible. I decided to save the generated data to several smaller compressed files. The reason for compression is reducing disk usage and speeding up load times.

2.6 Detailed work performed on the Variational Autoencoder

The idea behind utilizing autoencoders for image-related tasks is dimensionality reduction. In the case of CarRacing, the observation is a 64x64x3 image (12 KiB), which was reduced to a 32-dimensional vector (128 B), also known as latent vector.

In short, Autoencoder neural networks consist of three part: an encoder, one or more bottleneck layer, and a decoder. The type of the encoder and the decoder varies according to the type of data that the network has to learn. In the case of images, convolutional layers are mostly utilized. The size of the bottleneck layer(s) depends on the size of the images and the required precision, and the type of the bottleneck layer can vary depending on the task.

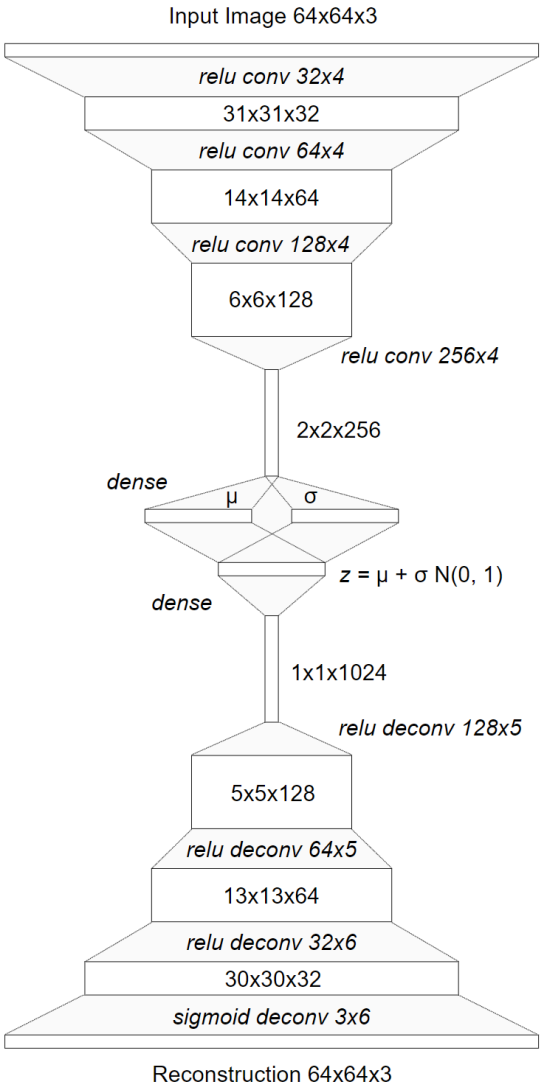

Figure 3: The structure of the utilized VAE

The assumption is that we can train the network by calculating an error between the original image and the reconstructed image. By changing the weight values of the network the error can be minimized, thus the network is capable of describing the image only by its latent vector.

Variational Autoencoder is a type of Autoencoder, where the bottleneck has two fully connected layers. The first layer consists of two parts, one for representing the mean of the latent vector and another one for the variance.

The second layer then takes a sample from this distribution and that will be fed to the decoder. To achieve the desired Gaussian distribution of the latent vector values, Kullback–Leibler divergence (1) (KL-loss in short) is introduced, which can calculate the difference between two distributions. For the proper reconstruction of the images, I used the Mean Absolute Error (2) function (MAE, or reconstruction loss).

$$D_{KL}(P||Q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} dx \tag{1}$$

$$MAE = \sum_{i=1}^{n} \frac{|y_i - x_i|}{n} \tag{2}$$

The problem that presented itself is combining the two losses into one, from which the gradient can be calculated for backpropagation. Solving this problem took a while, because as other implementations were using a weighted sum of the two losses, somehow it did not perform well in my case. I found that using the reconstruction loss only during training the KL-loss starts to decrease as well, after a few epochs. It seemed like the network discovers that using a Gaussian distribution is effective. My first idea was to separate the two losses into two different training: the first one uses the reconstruction loss only and the second one lowers the KL-divergence. It turned out that using the KL-loss only increases the reconstruction loss in the second part, so I needed to keep the MAE as well. I was even thinking of freezing the layers somehow, but I did not implement such training in the end. My final solution is the following: at the start of the training, the KL-loss has a really small weight, which gets increased at the end of each epoch. The change of the two losses can be seen in the figure at the end of this session. In my experiences, the MAE loss can be considered good enough below 0.01. I would like to point out that I had no time for hyperparameter optimization. On the figure below, there is an example for the changes of the losses: at first, the reconstruction loss decreases dramatically, but as the weight of the kl-loss increases, the reconstruction loss grows later on. One important aspect of training is when to stop, and which weights to save. Early stopping and model checkpoint are the two commonly used methods at training, which two I was using as well. In the example, the saved weights are from epoch 9. I was using Tensorboard for the loss visualizations.

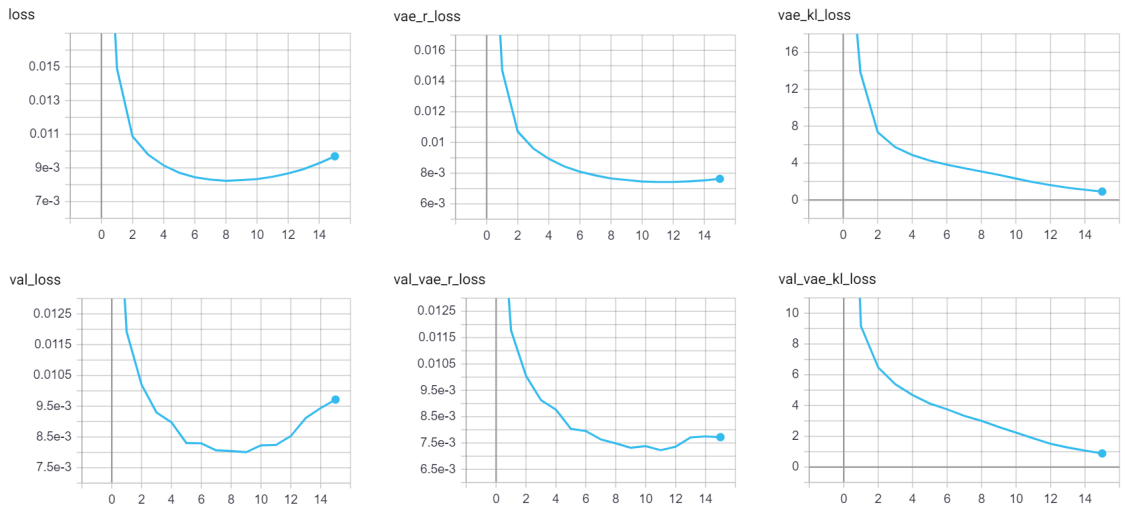

Figure 4: Comparison of the losses: first row is the training loss, second row is the validation loss. From left two right: Sum of the kl and the reconstruction losses, reconstruction loss, kl loss.

The reason for using VAE is because the RNN needs to make a prediction based on the latent vector. With the VAE, we can guarantee, that a little change in the observation cases just a little change in the latent vector. With other words, the latent vector changes smoother. One of the useful tools that I created is demonstrating this behavior. The tool takes an observation, encodes it to a latent vector using the VAE and then decodes it to an image. After that, the values of the latent vector can be changed manually, and the tool decodes the new vector automatically. The new image correlates to the changes in the latent vector.

As the original project assumed way more available system memory than my hardware had, a generator was needed, which would feed the training images for the Autoencoder. As I mentioned before, I saved the images to

several compressed files. I implemented a DataGenerator class for uncompressing one file at a time and loading all of its frames into memory and creating batches of them. Then the training can be accomplished by feeding these batches to the VAE. One backside of this approach is that I cannot shuffle the dataset perfectly. If the batch size is smaller than the number of images in a compressed file, every batch contains images from two files at most, but usually only from one file. My opinion is that making the dataset non-sequential is the most important requirement, and the imperfect shuffling method caused no side-effects in the performance of the VAE.

After the Datagenerator was done, I ran the training of the original network to provide a baseline performance of the VAE. I was making changes to some of the parameters, such as batch size, latent vector dimension, optimizer, learning rate and I tuned the increasing weight of the KL-loss. The last thing I tried to achieve is applying batch normalization on the layers. It did not turn out well, the performance of the trained model was well below expectation. As I did not have more on this part of the project, I moved on to the RNN.

The final result of my VAE seems quite good, an example of the original and the reconstructed observation image can be seen on the figure below.

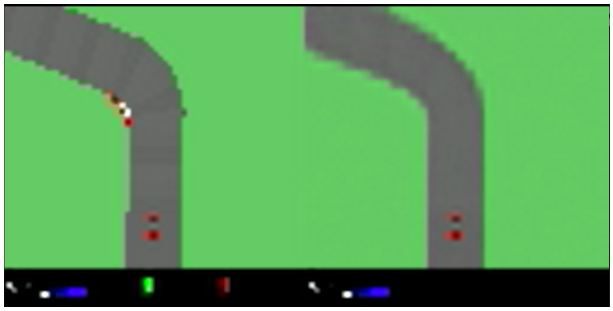

Figure 5: The VAE is capable of reconstructing the original observation (left image) at a lower precision

2.7 Detailed work performed on the Recurrent Neural Network

After everything seemed good with the VAE, I could move my focus on the RNN. The goal of this component is to provide a possible future observation for the controller. The reason behind this method is that the driving task can be barely solved based only on one frame, changes over time have to be considered as well. The original paper described a Gaussian Mixture Model (GMM) to be used in hand with the RNN. The distribution of a latent vector value may not be perfectly describable by one Gaussian, that's why using GMM might be useful.

The other important advantage of the GMM will be useful later on for the Dreaming method, where the uncertainty of the prediction should be increased. The main reason for this is to cover as much of all the possible scenarios as possible while dreaming.

As I had no preceding knowledge about GMM, I made some research on it. The theory is the following: if a distribution cannot be described with one Gaussian distribution precisely enough, the weighted sum of multiple Gaussians may be able to produce a better approximation.

The following example utilizes three Gaussians for the description of the original distribution. Notice how two parameters of a normal distribution grow to nine for the three Gaussians.

$$P(x) = \sum_{k=1}^{K} \pi_k \cdot N_{\mu_k,\sigma_k}(x) \tag{3}$$

At the moment, I am not completely sure how the number of Gaussian Mixtures affects the performance of the RNN, I have to do research on that later. One of its benefits will be the adjustable indetermination later on with the dreaming part of the World Models.

Luckily I was more familiar with the theory behind the recurrent neural networks, as usually, LSTM[4] is utilized.

After training the Variational Autoencoder, I had to generate the dataset for the RNN. The RNN's input is a sequence of the encoded frames by the VAE and the actions taken at that frame. The RNN's output is the sequence of the predicted encoded frames. I saved both the input and the output to separate files because I could easily utilize my previously implemented Datagenerator class. I did not generate other sequences of frames, I reused the previously saved frames that I trained the VAE on.

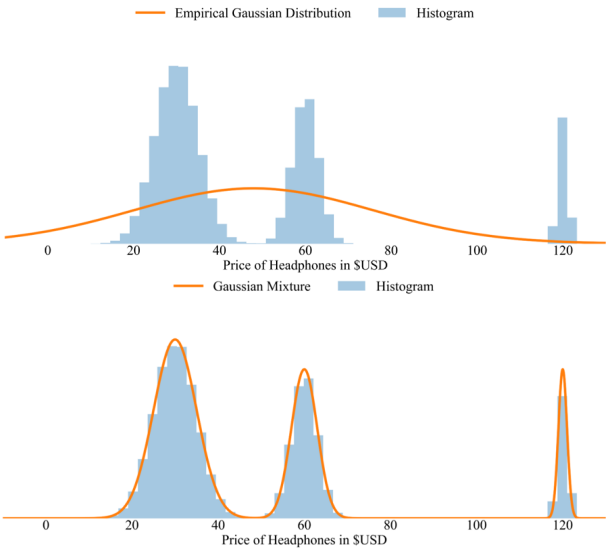

Figure 6: An example for utilizing Gaussian mixtures: the original distribution can be more precisely described by multiple distributions[1]

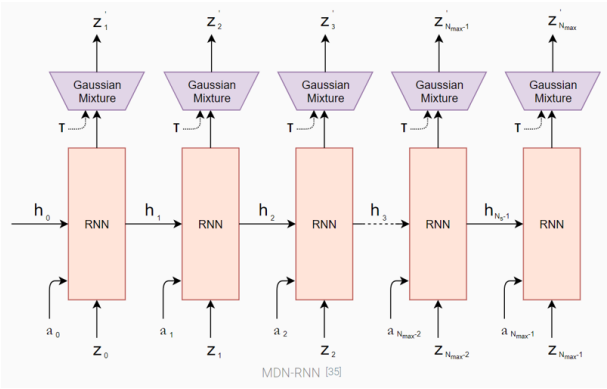

Figure 7: The structure of the MDN-RNN

2.8 Controller

As I was running out of time for the end of the semester, and I could not even finish the work with the RNN completely, I could not make much research on the Controller. This could have been expected if we take a look at the complexity of the World Models, but it is definitely a misfortune for the project laboratory course.

What I could do, however, is running the training as it was already implemented (using the trained VAE and RNN), while I could focus on other aspects of the project, i.e., this documentation. On the other hand, the provided machine had a weak CPU in terms of performance for the reinforcement training task. Originally the authors were using a 64-core CPU (or cluster) and I had access to a 4-core CPU only. I could train some agents in a one-week period and I had the following conclusions: first, using bigger population results in faster learning, but makes the variance of the final rewards bigger as well. Second, the trained agents were performing quite good, but they were cutting the corners most of the time. I found out that the reason behind this was the length of the episodes at learning. Each episode ended after 1000 frames and the frames needed for the agent to finish one lap was also about 1000. This resulted in the agent to rush the race to collect as much reward as possible. At a higher speed, it's harder to take the corners, so the agent was cutting them. As the episode ended just after the first lap, the agent did not receive punishment for the missed corners. After I increased the maximum episode length to 2000, the agent was taking the corners as intended but had a little slower speed.

2.9 Future works

As I was running out of time for the end of the semester, I have a lot of things to try out in the future. For the VAE, I have to run hyperparameter optimization, because I think it's possible that a smaller network with a bigger latent

vector may be able to converge faster and reach higher precision. I would like to take a second look at the batch normalization as well because it should work way better.

On the topic of the RNN, I should run hyperparameter optimization as well.

In the long run, maybe I would change this type of configuration of the model for further research, there are so many neural network architectures to play with.

I did not have enough time to explore the controller and reinforcement learning as much as I wanted to and should have to. This part of the project is the most poorly discovered.

Luckily, it seems like I am going to continue work on the same topic at Continental in the summer and we will prepare for a Duckietown[8] competition in December.

2.10 Duckietown

Duckietown started as a course on MIT in 2016 and starting from December 2018, competitions are organized on conferences, such as NIPS or ICRA.

During the semester some of my student colleagues were exploring the Duckietown platform further than me, but I did get updates on their projects and I started to participate in the exploration of Duckietown's capabilities. We assembled a few robots and started to make the first demos on them.

2.11 Summary

The introduced World Models concept caught my attention with its complex architecture. The idea to use different types of neural networks for the different parts of the task is fascinating and I think that this approach could be well used in other tasks as well. I think that I was exploring the publication on a level that required way more time and energy than I had in the semester. Despite that, I am totally satisfied with the overall process, because I learned a lot on the topic of researching neural networks.

3 Bibliography of the researched papers and other studied documents

[1] Oliver Borchers. A hitchhiker's guide to mixture density networks. *Towards Data Science*, 2019. https://towardsdatascience.com/a-hitchhikers-guide-to-mixture-density-networks-76b435826cca.

[2] Steven Borowiec. Alphago seals 4-1 victory over go grandmaster lee sedol. *The Guardian*, 15, 2016.

[3] David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. In *Advances in Neural Information Processing Systems 31*, pages 2451–2463. Curran Associates, Inc., 2018. https://worldmodels.github.io.

[4] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[5] Tesla Inc. Autopilot, 2016. https://www.tesla.com/en_EU/autopilot.

[6] Tesla Inc. Tesla autonomy day, 2019. https://youtu.be/Ucp0TTmvqOE?t=4151.

[7] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013. https://arxiv.org/abs/1312.6114v10.

[8] Liam Paull, Jacopo Tani, Heejin Ahn, Javier Alonso-Mora, Luca Carlone, Michal Cap, Yu Fan Chen, Changhyun Choi, Jeff Dusek, Yajun Fang, et al. Duckietown: an open, inexpensive and flexible platform for autonomy education and research. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1497–1504. IEEE, 2017. http://duckietown.org.

[9] SAE. Taxonomy and definitions for terms related to on-road motor vehicle automated driving systems, 2014. https://www.sae.org/standards/content/j3016_201401.