Project Report – AVG Project Lab, Autonomous vehicle

Dániel Turóczy – BLRP4W

Summary

The goal behind the project is to train a network which can control an autonomous vehicle. The vehicle is constructed by the team, with multiple sensors to help navigation. The network itself would control the power of the rotors, or the movement of the car would be defined through simple actions, such as right, left, forward or reverse. The network will be trained with reinforcement learning. The initial training will be done through simulation, to achieve a network good enough to perform the learning on the device.

Introduction

In the first few weeks, I have made some research in autonomous driving and technics to control the device. I have found only a few solutions, but the main concept was that the networks mostly rely on convolutional networks. Images seem to have the best information for training these models, although, I have not found information about the size of the images. After analysing this solution what I have concluded was that the definition of the image should be high enough so that the edges of the road or the middle line are visible, this helps most stabilize the input. A picture with too high definition would result in a lot of unnecessary noise for the network and the parameter space would be too much for the microcontroller. The second most prominent information that I have found was recurrent networks that use information from the vehicle, such as speed or gyroscope data. These seemed to help stabilize the long-term effectiveness of the network.

Network and training

We have discussed that the best option for now would be reinforcement learning. It can be easily customized for different scenarios and we would not have to generate data for it. The issues although with reinforcement learning is that it is hard to optimize and hard to test. The necessary components and classes were constructed so that they are usable in both simulation and real-life test.

Reinforcement library

The main purpose of the library is to manage the training for a neural network. Reinforcement learning uses a defined structure and I have decided to implement this for our case. The main part of the library is the **ReinforcementModel**. This model contains a network that is trained through the reinforcement approaches that are implemented in it. The network contained in it is independent from the **ReinforcementModel**. This allows us to change the architecture as necessary and we can work more on the architecture and define the training process in a single place.

To improve accuracy and performance of the network, we needed to increase the exploration of the agent, it is done through a random number generated and a decaying epsilon limit. This allows the vehicle to make more random actions at the beginning of the training, so that later it will have memory of these actions as well, saved in the **ReplayMemory**. As the training progresses the number of random decisions is also decreased.

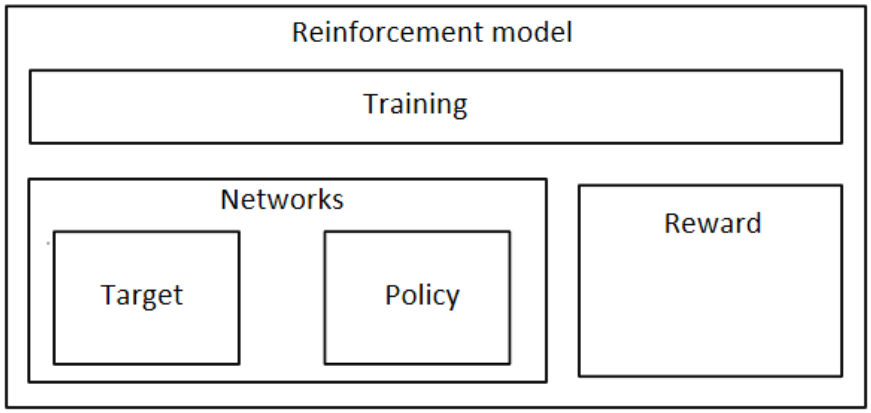
The model contains two networks of the same architecture. A policy network and a target network. At every iteration the policy network is trained, and after a set number of epochs the weights are copied into the target network. Decisions are always made by the target network if the decision is not random. This is done so that the decision making is separate from the training, as training only on the tatget model would slow down the convergence and the stability of the network.

The second solution to increase the stability of the network is batch training on previous information. Training only from second to second could be very inefficient. By saving all inputs, decisions and rewards, the network can be trained with supervised methods. For instance, after every 100^{th} epoch the network is trained on all the previous information with batch learning. This makes the network better converge to the optimum. The pervious states and information are stored in the **ReplayMemory**. The exact element of the **ReplayMemory** are: time, previous image, previous features, decision, next image, next features, reward.

Reward function is separated into a class as well. This is done so they become more interchangeable, and more versions can be defined. We have tested many different reward functions. Currently, only the two best functions are implemented in the class. A base reward and an inline reward.

The base reward looks at the distance between the vehicle and the path, and the further the vehicle is, it loses more and more rewards. The inline reward only adds points to the vehicle if it is close to the line. Otherwise, it is not getting any reward.

Another key point of the **ReinforcementModel** is that it needs both states to calculate rewards, expect when used with the microcontroller, since in that case the microcontroller is calculating the reward. For this reason, the predict method can be called with the current state t and it returns the action, and the optimize needs to be called with the state resulting from the action. There is option for supporting both current and next state to the function and reward also. In this function it calculates the reward, if it was not defined on the input, and adds them to the **ReplayMemory**. Since the ESP already calculates this information there is a function which only preforms the training based on the data inside the **ReplayMemory**, instead of adding any new data.



Initial architecture

The initial architecture was taken from the following paper. Their architecture seemed to be the best for our case, and we could test the credibility of the device with the network as the paper gives proof that the network was capable to learn the problem. The network can be broken down into three separate parts. The first and the second part are parallel in the architecture. One of them is a convolutional network and the other one is a recurrent network. The third part is a fully connected layer that combines these parts together. https://web.stanford.edu/~anayebi/projects/CS_239_Final_Project_Writeup.pdf

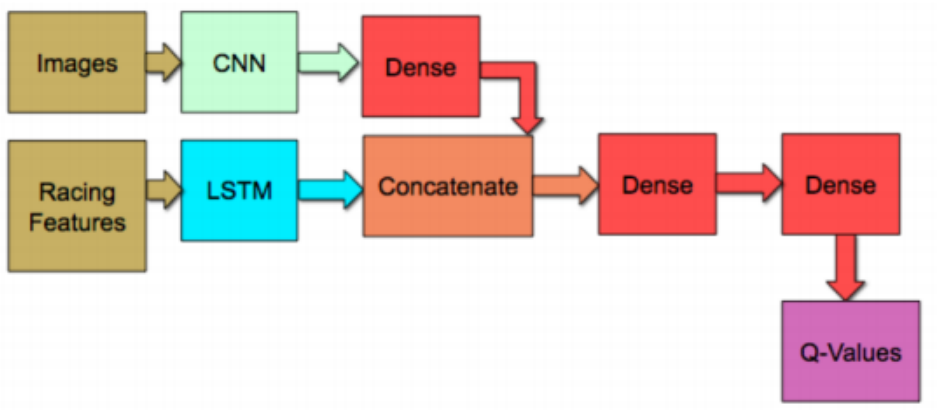


Figure 4: Architecture of the hybrid CNN-RNN agent.

We have tested the architecture, but the training was way too slow for us, and the sizes of the parameters were higher than what we could fit onto the microcontroller. For this reason, we have implemented 5 other architectures next to this one. Each one of them were tested and optimized and their results are shown in the next table. On the input we differentiate two types of information. One is image and the second is features. Image contains the input image from the camera, while the features vector contains any additional information about the state of the vehicle such as speed and sensor data.

Name	Training Time	Stability	Image	Features	Task completed
SCNN	Medium	Medium	Yes	No	Yes(with issues)
LCNN	Slow	High	Yes	No	Yes
CNNwRNN	Very Slow	Medium	Yes	Yes	No
CNNwDense	Slow	Medium	Yes	Yes	Yes
Dense	Fast	Low	No	Yes	No
FlattDense	Fast	Medium	Yes	No	Yes

- SCNN: This is a small Convolutional network. It was designed to reduce parameter space and reach convergence faster. It consists of 3 convolutional layers and two dense layers on the output.
- LCNN: This is a large Convolutional Network, very similar to the SCNN. It contains 5 convolutional layers.
- CNNwRNN: This is a Convolutional Network with a parallel Recurrent Network, same as the initial architecture shown before. It uses the LCNN on the CNN part and a Transformer network on the recurrent part. It was very slow to train with imbalanced results in our case.

- CNNwDense: Same architecture as the CNNwRNN but instead of a Recurrent Network it uses a dense network, so it can be optimized faster, it consists of the LCNN and a parallel dense network.
- Dense: This is a feed forward network; this one does not use the image, only inputs from the feature vector. This consists of 3 Linear layers.
- FlatDense: This network uses the image flattened on the input; it consists of 4 Linear layers.

Additional information, sizes of the layers, and depiction can be found in the source code.

They were all trained between 5000-7000 epochs to avoid overfitting. Stability in this case describes the stableness of the network in the sense, whether it would complete the path in the simulation multiple times, or overoptimize and fall out from the optimum. Some of these networks, especially the CNN-ones, would learn the solution, then overoptimize and fail the simulation multiple times. To avoid this, learning rate for these networks should be lowered to about 0.0001, or adaptive learning rate if RMSPROP is not used.

All of these networks are documented and detailed in the code as well. Some of them worked better than others. The limitations to these networks were the resolution of the images. 32*32 is very low, and hard to utilize. When increased to 128*128 in the simulation, most of the networks did better than the results recorded here.

Due to hardware limitation only SCNN and FlatDense can be used on the microcontroller, the other networks were too high to export. This limits the range of networks available, but the results from the other networks were necessary to optimize other training parameters.

Hyperparameters of Reinforcement learning

During the test, in the simulation environment we have optimized some of the parameters, and in this section, I wish to talk about the parameters and their respected uses. I will explain why we have come to this decision and what type of errors these solutions solved.

One of these parameters was the random decision making. We were using a decaying epsilon strategy. What this means is that the agent will make random decisions, more at the beginning of the training and less at the end. This is in order to encourage exploration over learning one solution and exploiting that solution over time. The general thumb rule we have

uncovered was that the random number of actions should be between 5 to 10 percent of the replay memory. If there is more, then the algorithm will learn to predict unstably and if there are less, the algorithm will generalize to one solution, for instance, turning left all the time.

Another issue that aroused during the training was that the network was learning to go straight to the left or right no matter the circumstances. The issue was caused by that the time interval of updating the target network was too low, 25 to be precise. This meant that after every 25^{th} training the weights of the policy network were copied into the target network. This resulted in a lot of changes in the network, so the training was inconsistent. To improve this the update parameter must be above 50, ideally 75 and 100. This number corresponded to 4 or 5 decisions per second. The higher the decision per second metric, the higher this number should also be for optimized training.

We have also tried an ADAM optimizer, but it was not optimizing well, minor success was achieved with it, but RMPROP performed the best. The learning rate of the optimizer must be quite low for reinforcement learning, around 0.001. This also decreases if used with Convolutional Neural Networks to 0.0001.

Another key parameter that allowed for better performance in the simulation was batch training. This makes the network more stable, and balances out the training data between the random actions and decesions made by the network. The optimal number we found was 128 or 256. Any higher, and we ran into hardware limitations on the training computer with larger networks, and with smaller networks 512 caused no difference in the training.

Each network was very sensitive to the reward function. The inline reward mentioned earlier was generally good for each network. To reach better results we would need to define different rewards for different functions. For instance, a FlatDense and an SCNN with the inline reward perform well, but the FlatDanse can also be trained with the base reward function while the SCNN is not optimal with it.

Additional Architecture ideas

One of our teammates proposed to use the earlier mentioned network with a new convolutional network that would do a feature extraction and add useful inputs for other parts of the network. Unfortunately, we did not find a dataset for this, but if the earlier

network and its changes will not solve the task, we will think about how to create a dataset in Carla.

These feature extractions would be done from the picture and try to estimate some of the key data that we might need, such as distance from the line. Training data can be generated from Carla as well, saving the information of a simulation include both image and distance from the vehicle.

Another idea we had was in later stages, when instead of path following, we want to focus on different types of objectives for the vehicle, we can include direction on the input as well, so instead, we could have an algorithm predicting the direction we need to go and a network choosing the best course of action for the given direction.

Simulation

Training in the simulation worked flawlessly, we were able to gather useful information for optimizing the network. The information received in the simulation was very clean and easy to utilize for a network. When training pretrained weights for the microcontroller one has to avoid overtraining. One transformation we have made to make it optimal was inversing the input of the network inside the simulation, since in the real-life test scenario the line is black and inside the simulation it is white, we believe this will help with the retraining on the vehicle.

Training on the microcontroller

Due to the nature of reinforcement learning, training on the microcontroller would be very hard and not optimal at all, so instead we did training remotely and not on the device. Most of the networks of pytorch can be used on the microcontroller, an extensive list of supported network components can be found on the cubeAI website.

To recreate data and training scenarios from the microcontroller, the **ReplayMemory** was extended with a save function, so it can be saved from the memory to a .csv file. This contains all the information stored in the replay memory with timestamps. Secondly, a function was created with FFMPEG to extract the images from the .csv file and reconstruct a video from it.

To also allow easy replay of scenarios, a new script was added to the network library for reconstructing the images into a video. The script uses FFMPEG for the reconstruction so the program has to be installed and added to path for use.

Real Test

During the real test we have resolved major compatibility issues between the network and the communication. To successfully import network onto the microcontroller, the input sizes for the network must be provided for the ONNX exporter. This can be done in the export.py script.

The vehicle works in the following way: First, it initialises weights from the remote computer, then it starts performing actions until a set number of decesions, requests training and new weights from the remote controller. The server.py must be started on a computer for the ESP to connect. There are sometimes issues with this on the BME net or any other local networks, so we suggest using laptop or mobile hotspots.

During our tests we were unable to load pretrained weights due to network conflicts, meaning the trained weights did not correspond to the network we have uploaded onto the microcontroller. So we have decided to use random weights instead and start the training from ground zero. The agent started the training but we were unable to achieve a sufficient solution.

Issues were that the agent turned too fast and the image changed very rapidly and it was hardly recognisable. The resolution was also a little bit too low in my opinion. This should be increased later. Another issue was not using pretrained weights. These would have helped in stabilizing the agent's action, before it goes out of the track, results in smoother training. Last issue we found was that the camera was not affixed, so the input becomes noisy due to this. The camera should be fixed in place. Here comes a very hard decision, of what the angle of the camera should be. I would propose the angle in a way so that it sees the start of the line, and the unseen length of the line from the front of the vehicle is about 5 centimetres.

Next steps

Key missing element from the reinforcement library are visualization. This was hard to make in the simulation, but from the incoming data from the robot this should be easier now. New

metrics have to be defined to measure the learning of the agent. Lastly, optimization of the hyperparameters has to be separated from the simulation optimum. Some of the optimization numbers are too high for the microcontroller. For instance, waiting of the microcontroller to perform 256 action before starting the batch training is way too much time. Here, I would propose to increase the batch size with the number of actions performed. For the same reasons I would also suggest an adaptive learning rate of the real vehicle as well, to reduce training time. This adaptive learning rate should not fall below the limits I have mentioned in the hyperparameter optimization, but at the start should be higher.