HTWG Konstanz

Department of Computer Science

Ubiquitous Computing Laboratory

Lecture: Software project (AIT, SS21)

Title:

[AI]T Racing – Deep Learning for Autonomous Racing:
Description of the work and final results

Submitted by

Name	Student ID	E-Mail address
Jakob Häringer	298095	jakob.haeringer@htwg-konstanz.de
Daniel Košč	301548	da741kos@ htwg-konstanz.de
Marten Kreis	298623	marten.kreis@htwg-konstanz.de
Niklas Kugler	298335	niklas.kugler@htwg-konstanz.de
Finn Lehnert	298642	finn.lehnert@htwg-konstanz.de
Valerio Müller	297197	valerio.mueller@htwg-konstanz.de
Frederik Poschmann	298025	fr391pos@htwg-konstanz.de

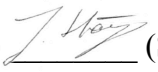
Project time: Start: 15.03.2021
End: 18.06.2021

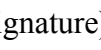
Supervisors: Dipl.-Ing. Lucas Weber

Prof. Dr. Ralf Seepold

Declaration of Authorship

Hereby, I declare that I have composed the presented report independently on my own and without any other resources than the ones indicated. All thoughts taken directly or indirectly from external sources are properly denoted as such.

Konstanz, 17.06.21 (Place, Date) Jakob Häringer (Full name)  (Signature)

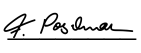
Konstanz, 17.06.21 (Place, Date) Daniel Košč (Full name)  (Signature)

Konstanz, 17.06.21 (Place, Date) Marten Kreis (Full name)  (Signature)

Konstanz, 17.06.21 (Place, Date) Niklas Kugler (Full name)  (Signature)

Konstanz, 18.06.21 (Place, Date) Finn Lehnert (Full name)  (Signature)

Konstanz, 18.06.21 (Place, Date) Valerio Müller (Full name)  (Signature)

Konstanz, 18.06.21 (Place, Date) Frederik Poschmann (Full name)  (Signature)

Abstract

The goal of the challenge was to implement a neural network which is based on a supervised learning approach. The neural network should produce an output to control a simulated car within the OpenAI Gym.
The best score achieved as an average value of 10 different tracks is 895.38. To get an idea of the comparability of the score is that the best possible score would be something around 950 points.
To accomplish our best score, we implemented a DenseNet with 4 dense blocks containing 6,6,7 and 8 layers. This network was optimized with the SGD algorithm, which was initialized with a learning rate of 0.01 and a momentum of 0.9.
The neural network is trained using an unbalanced dataset consisting of 130.000 samples. The images were preprocessed and a gaussian noise was added before passing them to the neural network.
During preprocessing relevant information (speed, abs …) were extracted from the image and added directly to the fully connected layer which is the last layer before using the sigmoid function to get the control output for the car. A dropout function was applied with a percentage of 50% to the fully connected layer to avoid overfitting.
The network was trained with a batch size of 32 for around 100 – 200 epochs. To optimize the performance of car controls, boundaries were applied to the output of the neural network to decide more precisely when to steer, accelerate and brake.

1. Introduction

The goal of every team is to develop a neural network that can drive a simulated car on a racetrack and collects points on its way. The game itself is the Car-Racing-Simulator from the OpenAI-Gym which is implemented in Python. For the challenge it is mandatory to implement a neural network which is trained using the supervised learning approach. The faster the car is, the more points it will gain. The neural networks of every team are tested within *Google Colab* on the same ten racetracks and the average score for every team is calculated. The team with the highest average score in the end wins. The teams have the possibility to submit five different codebases until the end of the challenge phase. Our final goal is to implement a deep learning architecture to achieve an average score of at least 850 points.
Our team is called "[AI]T Racing" and consists of seven team members, that are stated in the beginning of this document. The first part of the project is the time plan to structure our project and get an overview of what has to be done. Our detailed time plan looks like this:

Due	**Task**	**Subtask**
07.05	Training Data	Record Training Data
		Training Data Handling
	Pre-processing	Extract important data from PNG, Resize (No status bar)
	Architecture & Model Structure	Implementation of DenseNet
		Adaption of the structure
21.05	Training of Neural Network	Training Function (1 Day Training Time)
		Data loading PyTorch
28.05	Optimisation	Hyperparameters
	Optimisation	Increase Stability
04.06	INTERNAL DEADLINE	

We organised our subtasks in three more summarized goals:
The first big step was due May 7th and focussed on everything regarding the neural network implementation, which includes training data generation, pre-processing and the architecture of the network. From this point on, we could concentrate on the training. The training is goal number 2 and was scheduled for May 21st. Then we planned seven days for optimizing the output of the trained neural network to try to aim for the highest possible score and consistency. The internal deadline for our team was set to June 4th so that we had a time buffer of one week for any issues that occurred on the way.
The following parts of the document describe each task in detail, that we defined in our time plan.

2. Methods and Implementation

2.1. Methods

Training Data Generation
"training" a neural network means, we have to take one possible input and show the network what output it is supposed to calculate from the input. Since our input consists of images from the current car position on the track, we wrote a script, that would save every frame of the games' display with the keyboard inputs while playing it manually. In total we generated 130.000 training samples with corresponding labels.

Data Balancing
Data balancing is an important step in training the network. Depending on the source of the training data, the data might not be ideally distributed across all categories. This might lead to a network which is overtrained in a specific scenario and undertrained in another one.

To counteract this, data should be balanced before it is fed into the network. By reducing large amounts of unwanted and unbalanced data, the network will be trained more precisely. We chose to balance our data to a small extend, in which we only corrected the percentage of each category by a few points.

Game Environment

The frames consists of 96x96 pixels. The input can be acceleration, braking and steering. The points 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 750 frames, your reward is 1000 - 0.1*750 = 925 points []. The game round finishes when all tiles are visited. Some indicators are shown at the bottom of the window. From left to right: true speed, four ABS sensors, steering wheel position, gyroscope.

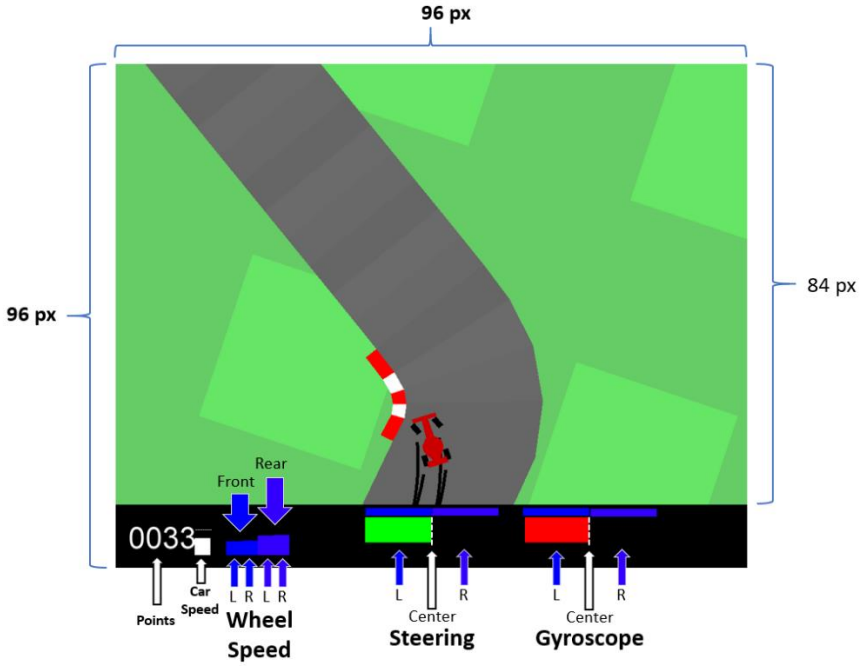


Figure 1: Game frame with state bar

Preprocessing

The purpose of preprocessing the input frames before entering the neural network is to reduce the information within the input frame (e.g. grey stripes on the track). In addition to that it makes training faster and more accurate. Furthermore its easier for the network to detect important features like the border between the grass and the track. Another advantage of preprocessing is the possibility to transfer analog bar information into digital data.

Network

The Network used is this project is an adaption of the DenseNet. First introduced by Guang Huang et al. (2017) a densely connected convolutional neural network describes a deep neural network architecture which contains connected dense layers. The connection between these layers makes it possible to bypass some convolutions, so that information gets preserved. This method drastically increases the accuracy of deep neural networks and allows a more efficient training. Furthermore, it is possible to reduce the overall number of parameters compared to other deep convolutional networks [1].

A DenseNet contains a certain number of dense blocks which again contain dense layers. The total amount of dense blocks and dense layers per block depends on the specific architecture which is used. The most common DenseNet architecture, the DenseNet-121 consists of four dense blocks. The first block includes 6 dense layers, the second block 12, 13 layers are implemented within the third block and the last block contains 16 layers. Each layer is directly connected to all subsequent layers within the same block, which can be seen in the figure below.

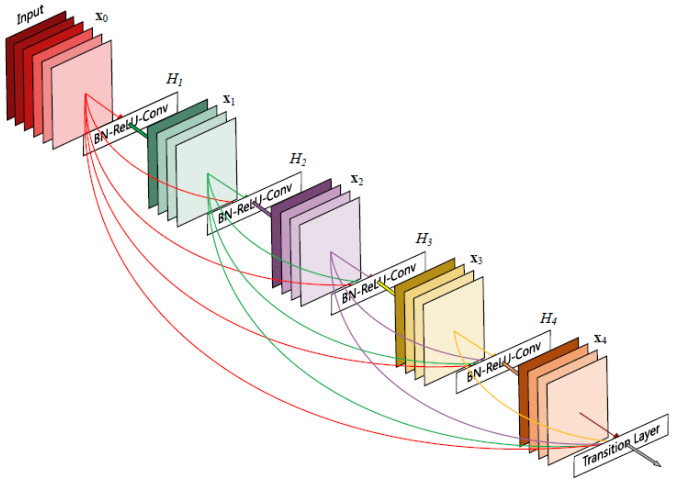


Figure 2: Structure of a Dense Block [1]

In contrast to ResNet, where features are combined through summation, before being passed the to the next layer, the feature maps are concatenated [1]. Each layer has $k_0 + k \cdot (l - 1)$ input feature maps, where k_0 is the number of channels passed to the dense block. The parameter k is referred to as *growth rate*, since it determines the number on added feature maps by each layer. Common values for the growth rate are 12, 24 and 32. Since the number of parameters for each convolutional layer can grow rapidly depending on the growth rate, bottleneck layers are used, which perform a 1x1 convolution before the 3x3 convolution. Moreover, transition layers are placed between the dense blocks which perform a 1x1 convolution to bisect the number of channels. Furthermore, a n average pooling operation is being used to shrink the size of the feature maps.
After the last dense block, a global average pooling layer is placed to compress the feature maps to a dimension of 1. Finally, a fully connected layer with a softmax function is used as the output layer of the neural network.
The decision to use an adapted DenseNet architecture for this project is mainly based on the convincing performance of this architecture compared to other Networks and the low number of required parameters. Many different DenseNet implementations achieve superior benchmark scores (e.g., CiFar-10, ImageNet) compared to architectures like VGG. Furthermore, the necessity for a low parameter count (max. 300.000) given by the challenge guideline can be fulfilled by the DenseNet architecture, while still delivering accurate results.

Optimizer
The main optimizer, which is used to train the Network is the Stochastic Gradient Descent (SGD). SGD takes an iterative approach to determine the minima of a given cost function [2]. The actual gradient is replaced with an estimation calculated by a subset of the data, which has the advantage of a much faster computation speed compared to Gradient Descent. Additionally, *momentum* can be added, which helps to accelerate SGD in the optimal direction and minimizes oscillation. Without momentum, SGD usually has problems to navigate around areas, where the surface curves much more steeply in one direction than in the other [3].
Compared to other state of the art optimizers like Adam, SGD yields the potential of reaching an even smaller loss, whilst the biggest drawback is that it does not converge as fast as some of its competitors. Since the computation time required to train the neural network for the project does not pose a problem, SGD is used for the main training sessions. Adam however is used for an initial evaluation of the Network architecture since it converges faster.

Dropout
To further increase the performance of the neural network we used the dropout function.
Dropout is a function which randomly disables a percentage of the neurons in a specific layer to reduce overfitting [4]. For every batch processed by the neural network the dropout function will choose randomly different neurons which will be disabled for the batch. This will in theory avoid the

network from entering a specific path on a high percentage of the time. The dropout function can be added in several different layers and in multiple layers.

Noise
To make the network more stable and to maybe reduce the impact of the different rendering of the image when running in *Google Colab* we added a gaussian noise to the image [5]. Based on the mean and the standard deviation there will be a random number of small blocks added to the image. Noise can be added within the entire network after and before several layers.

2.2. Implementation

Training Data Generation
To generate training data, we modified the original script that runs the game locally to save every displayed frame into a defined directory. Furthermore to make it easier to drive the car in order to generate good training samples we slowed down the game by 30 ms for each frame. The script contains a naming system, so every image has the same name except for an iterating number at the end.
Since there is a "zoom-in" effect when the game starts, the recording function had to be activated later. This was solved by binding the start of the recording to the first key press. This means, everyone who wants to generate training data has to wait for the game to be zoomed in completely before starting to drive.
When the images are saved, their file name gets written into a csv-file as well as the input values for the arrow keys, that are active at this moment. In the end, the directory consists of *n* images and one "labels.csv" file, that contains *n* rows of file names with the associated arrow key inputs.

Data Balancing
Our dataset consists of 9 different actions: accelerating, braking, turning left, turning right, braking left, braking right, accelerating left, accelerating right, coasting.
The training data images are assigned to those actions based on the inputs that were made during the training data generation. Due to most of the tracks beeing counterclockwise, turning and accelerating left had a bigger proportion than turning and accelerating right. Coasting (no inputs at all) was also quite prominent, despite trying to minimize it.

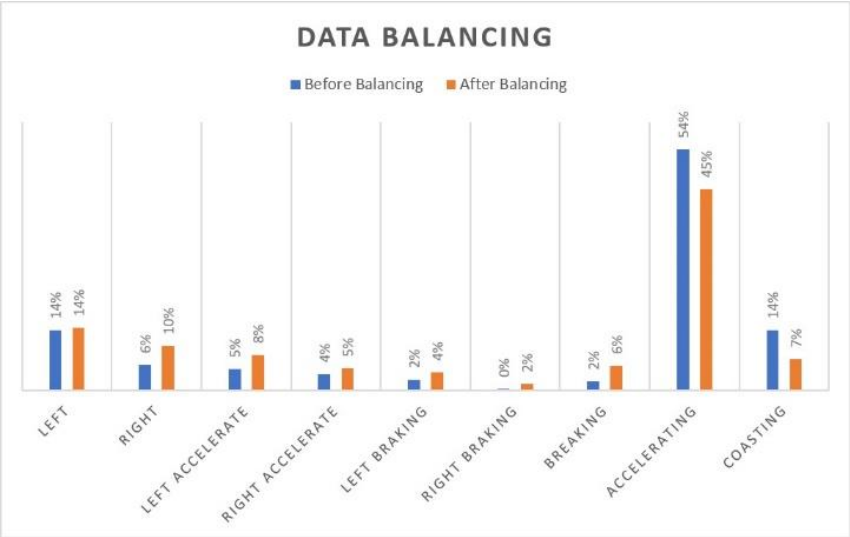


Figure 3: distribution of each category while balancing

In the image "data balancing" you can see how we adapted the percentages of the dataset. We tried to reduce coasting and increase turning and accelerating right. In the end, our neural net worked better with the unbalanced dataset, and we scrapped the idea of balancing. Most likely this was due to a reduction of the overall sample size, as the balancing reduced it from 130.000 images to around 35.000-50.000 images. Due to a lack of time and good results with the unbalanced data set, we decided against producing more samples, to have a sufficiently sized balanced set.

Preprocessing
We cropped the original image from 96x96 to 96x84 pixels and extracted all information stored in the bottom part of the frame. There were horizontal and vertical bars representing the speed, abs sensors for each wheel, gyroscope and accelerometer. They were mapped into a dictionary using 1 pixel of length as an atomic value. Moreover, the frame colours were reduced from three colours with 255 tones to only two colours with only one tone (not counting the car itself because it always occupied the same pixels). Processed frames displayed road with colour (0,0,0) and grass with color (0,255,0). Afterwards, processed values in the dictionary plus frame path were stored in an CSV file for later training purposes. In the following images you can see a example before and after preprocessing.

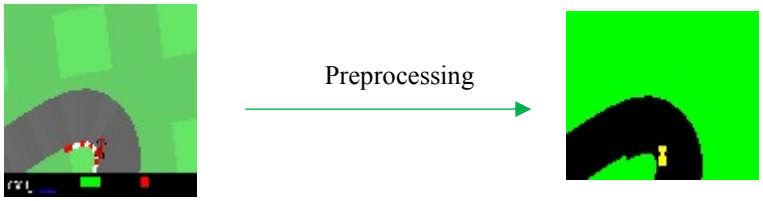


Figure 4: Frame preprocessing (left before and right after)

Network
The network architecture used in this project is based on the DenseNet described in [1]. However, the task to steer a car on a racetrack requires adaptions to the normal network structure used for ImageNet or Cifar-10. Two different models were developed to cope with the tasks.
One main concept of this project is to extract the relevant values for speed, abs, gyroscope, and steering angle, provided only visually by the game environment as bars, during the image preprocessing. By doing this there is no risk of losing relevant information (e.g., speed) during the convolution. The extracted values are then passed to the network and are inserted in the fully connected layer at the end of the Network. Therefore, the existing fully connected layer must be extended with additional weights of the 8 new input neurons and the input tensor previously only containing the output of the convolutional layers must be concatenated with the additional 8 input values. Furthermore, the kernel size of the global average pooling layer must be adapted to the input size of 96*84 pixels to compress every feature map to a single value.
Another impactful limitation of the project is the maximum permissible number of parameters, set to 300.000. Most DenseNets, even though much smaller than other competitive architectures, have a parameter count of around 1-20 million. The number of parameters needed can drastically be shrunken by adapting the growth rate of the network and reducing the number of dense layers in each dense block. The growth rate describes the number of feature maps produced by each layer. This hyperparameter is set to 12. Furthermore, the numbers of dense layers in each dense block for the first implementation is [6, 6, 7, 8]. The second implementation only consist of three dense blocks with a layer count of [8, 8, 8]. The growth rate 12 is a commonly used parameter for smaller DenseNets and the number of layers for each block is roughly the same as suggested by the developers of the DenseNet. The first implementation has a certain similarity to DenseNet-121-BC (used for ImageNet) with the first dense blocks containing less dense layers. The second implementation is based on a Cifar-10 architecture with only three dense blocks and an equal number of layers. This reduces the deepness of the second implementation compared to the first one, which might prove to be an advantage considering that the given input frame of this task is not really that complex and there is no necessity to distinguish between small details. Furthermore, an additional fully connected layer was added at the end with the aim to increase the performance, especially in hindsight that additional input neurons are added at the end which is quite unusual.
The desired output of the neuronal network are four output neurons for steering left, steering right, acceleration and brake. It shall be possible that multiple output neurons are activated at the same time. This can be achieved by adapting the number of output channels of the fully connected layer. Moreover, the default activation function of DenseNets, softmax, must be replaced, since softmax is used for standard classification problems where only one class out of a set of different classes is selected. In this case however, the possibility that multiple classes at the same time can be selected, is given (e.g., Accelerate & Steer Left or Brake & Steer Right). Therefor a sigmoid function is used as activation function, which maps the outputs of the neural network individually between values of 0 to 1.

Optimizer
For a major part of the training sessions the SGD optimizer is used, since it has the potential to achieve even better accuracy than Adam. A great advantage of Adam is to produce good results in a relatively short amount of time, so this optimizer is used if a quick validation of the network structure or parameter changes is needed.
The learning rate used for Adam is 0.001. This learning rate is not further tuned, since the Networks trained with Adam that achieved a good result were retrained again with SGD later, so there is no necessity of getting the best possible result with Adam by investing much time in tuning the learning rate. For SGD two different learning rates were used. The first learning rate, default of the *pytorch* SGD optimizer, of 0.01 achieved the best training results overall. The other learning rate used had a value of 0.1, since some research results showed that in certain cases this causes a fluctuation decrease of the validation accuracy and a slightly lower loss of the training data. The momentum value for the SGD optimizer is set to 0.9 for both configurations. This value is commonly used for SGD and prevents heavy oscillation whilst still delivering accurate results.

Noise
In order to get an idea on how to choose the size of mean and standard deviation within the gaussian noise function we inserted a preprocessed image and applied the noise on the image. For validation of the result, we displayed the output of the image before and after adding the noise. The augmentation will be taking place after the image is transformed to a tensor. To display the image, you need to transform the tensor back to an image.
In many cases where the noise is being used it is used with a quite strong/ big impact on the image. [5] Otherwise, there will be no notable change of the performance of the NN. From looking on the image and how heavily it was influenced by the noise we applied a mean of 0.001 and a standard deviation of 0.002.

Dropout
To reduce complexity on the dropout function we applied the dropout function before the fully connected layer. Our research concluded to add a high percentage of the dropout. We used a percentage of 0.5 percent. In retrospective it might be better to add the dropout before adding the additional neurons. Therefore, in our implementation the additional neurons will be dropped out as well.

Testing of different training settings and networks
After setting up the basic implementation the first training session were to find out about different batch sizes, optimizers and number of epochs. We chose a variety of 4 settings for the ImageNet based DenseNet and 3 settings for the Cifar-10 based DenseNet. The approach was to decide on a specific architecture and to make further optimizations on this architecture. To find out about the best epochs we saved the parameters every 5 epochs which also enabled us to continue training with several epochs. To confirm our ideas about the balancing of the dataset we also trained with the not balanced dataset. For the optimizers we used the following parameters, for Adam: learning rate = 0.001, betas = (0.9, 0.999) and for SGD: learning rate = 0.01, momentum = 0.9.

The settings for the first training session were the following:
Cifar-10 based DenseNet:
- Batch: 32 Epoch: 5-120 (AdamOptimizer)
- Batch: 32 Epoch: 5-120 (SGD Optimizer)
- Batch: 16 Epoch: 5-120 (SGD Optimizer)

ImageNet based DenseNet:
- Batch: 32 Epoch: 5-120 (Adam Optimizer)
- Batch: 32 Epoch 5-120 (SGD Optimizer)
- Batch: 32 Epoch 5-120 (SGD Optimizer) - Not Balanced Dataset
- Batch: 16 Epoch 5-120 (SGD Optimizer)

The final training session were based on the results of the first evaluation of the trainings and was to focus on further optimisations which we expected to get when adding noise and/or dropout. There were done after fixing a preprocessing bug which resulted in wrong values for the gyroscope. Furthermore, a rendering issue on *Google Colab* causes the score value to overlap with the speed bar. Since the score and the speed bar have pixel values of (255,255,255 - white) the preprocessing script extracted incorrect readings. By shifting the analyzed pixel row by one increment to the left, this problem is solved. The speed value is a critical information for the network since it heavily influences the decision, so this bug justifies the bad performance for the first submission.
Every setting for the last training session were done on the ImageNet based network with batch size of 32 and an SGD optimizer with a learning rate of 0.01. The chosen data set is the unbalanced dataset, and the training was done up to 150 to 195 epochs. Every 5 epochs were saved as usual.
The 4 settings were the following:

- No Dropout, No Noise
- Dropout, No Noise
- No Dropout, Noise
- Dropout and Noise

Evaluation of different training settings
After doing the first training session we found out that the validation of the network is not easy to compare. A validation dataset was used to find the epoch where the loss is minimal. The suggested epochs to use were between a range of five to 25 which is surprisingly low. Using these epochs for the agents playing the actual game showed a worse result compared to higher epoch numbers which suggests that a lower loss does not automatically infer a higher accuracy.
To compare the different trained networks and to have a comparable method for choosing which one works best, we implemented boundaries for the output of the neural network. If the output of a neuron within the output layer exceeds the threshold set by the boundary, the corresponding action will be executed. To more abilities to optimize the performance of the network we decided to test with the level of the different commands as well. So for example if you put the level of steering to 0.5 instead of 1.0 it will steer only with an amount of 50%. For a higher number of epochs the boundaries for steering, acceleration and braking will be higher as the network gets higher output results due to the training. For further optimisations we also tried to adapt those boundaries as we found out that in our case the boundaries were quite crucial looking at the performance of the network.

3. Results

This chapter describes the results of our submission and the adaptions we made from our observations during the testing and optimisation process.

The results of the implementation are very adequate. The adapted DenseNet containing dropout as well as a gaussian noise function trained with an unbalanced training data set showed the best results. This network was optimized with the SGD algorithm, which was initialized with a learning rate of 0.01 and a momentum of 0.9. The best score achieved as an average value of 10 different tracks is 895.38. The network can keep the car on the track most of the time and collects all tiles within one lap for most of the circuits. The average speed is quite high, and the car accelerates on the straights perfectly. The overall consistency of the car on different tracks is also acceptable, however sometimes if a track layout is encountered that differs strongly from the training data, the network is mostly not able to keep the car on the track.

After looking at the first results of the training settings we found out that the network is performing best on the ImageNet based network. We got some good results at scores of an average of about 800 locally for the SGD optimizer with learning rate 0.01 for the balanced and the unbalanced data set. Both were trained with batch size of 32. For the Cifar-10 based network we couldn’t get good results, so we decided to focus on the ImageNet based network.

For the unbalanced data set we got slightly better results so we focussed on this data set for all submissions. The first and second submission were made with the epoch 110 of the first training session. If we chose a lower amount of epochs there were some indications that the network is not trained enough. For example for choosing a lower amount of epochs we found out that when the car drives straight, it will not steer so eventually it will drive on the grass. With higher epochs this issue was resolved.
While testing we tried out a bunch of different adaptions of the boundaries for the output of the neural network. There was a crucial change in performance when adapting the boundaries. To find out about those boundaries we looked at the car driving and printed the output of the network for situations where the car didn't perform well. Due to that information's we could adapt the boundaries for our purpose.
For the first submission we started with a high percentage of 0.91 for acceleration since the dataset contains a lot of frames with a straight track where we accelerate only. For steering we found out that network isn't performing well in specific, rare turns so we put the boundary for steering right and left to 0.6. For a higher boundary the car wouldn't steer in those turns. We decreased the level of steering to only 0.5 because otherwise the car was flipping to often. For braking we chose a boundary of 0.4 because it would overshoot corners when driving too fast (braking not enough) and a level of 0.8 to avoid blocking the tires and loosing control over steering.
We also used an adaption which will accelerate when the car is driving below a speed of 2 and the gyroscope is 0. This is used to avoid the car from accelerating and turning at the same time which resulted in drifting a lot. This is included in all submissions.

The biggest issue in the first submission was that on some tracks the car would get off the track because it overshoots some corners because of entering the corner at a very high speed.
For the second submission we did some minor adaptions on the steering to put the level of steering to 0.7 so the car will steer a bit harder. We also integrated to set braking on 0 if the car is accelerating to avoid accelerating and braking at the same time. To face the issue of getting off the track we set a speed limit to 3 which worked a lot better and got the car finishing the lap almost every time. From this submission on we added to not steer for the first 30 frames to avoid drifting off in the start position.

For the third submission we added dropout to the training and we figured out a good setting for the epoch 175. After adding dropout and testing on this epochs we decided to get rid of the speed limit because the car will not have a chance to get higher scores if it can't go faster. The steering already worked a lot better for the training with dropout. We decreased the steering boundary to 0.35 and increased the level to 1.0. This affected the car to steer much more but the car didn't drift off anymore. For solving the speed issue we figured out that we have to adapt the boundaries for several speed limits. Therefore we implemented adaptive acceleration and adaptive braking. The basic idea was to accelerate more and brake less if the car is slow and if the car drives fast accelerate less and brake more. For a speed of up to 2 the boundaries were 0.6 for acceleration and 0.6 for braking. Between speed 3 and 4 the boundaries were 0.85 for acceleration and 0.4 for braking. If the speed is greater equal 5 the boundaries were 0.95 for acceleration and 0.15 for braking. All levels of the commands for the driver were set back to the maximum at 1.0 except for the braking which was still at 0.8. Dropout has to be removed from the model definition for evaluation and is only implemented in the training.

However, after adding noise and dropout to the training we could improve our best results which lead to the fourth submission. For this submission we used epoch 35. There was still an small issue within the third submission that the car will cut corners if its too slow. To fix this issue we added adaptive steering which reduces steering if the car is slow. Adaptive steering was used with the boundaries of 0.65 for speed up to 2, 0.7 for speed equals 3 and for a speed above 3 a boundary of 0.32. The adaptive braking was also changed to 0.9 for low speed up to 2 and for high speed greater equal 5 the boundary was only 0.1.

4. Discussion

The good performance of the network can be explained by considering that several different methods are used which prevent important information from getting lost during convolution. Firstly, the used DenseNet itself is equipped with connections between the different layers, which allows information to be preserved since some feature will not undergo convolution. Secondly, the core concept of this project, which is to add 8 new input neurons to the fully connected layer in order to pass the values displayed in the status bar to the network, ensures that this information will be used to calculate the output of the neural network. Furthermore, it is possible to change the boundaries, which determine whether an output should be considered dynamically during pre-processing. Implementing dynamic boundaries for steering as well as acceleration and brake resulted in a strong accuracy increase. Especially one of the main problems during earlier submission, the corner cutting issue, was completely resolved by applying this method. The corners were mostly cut due to a slow speed of the car, caused by incorrect steering inputs. This scenario is not part of the training data, which most likely caused the incorrect assessment by the network.

Two problems can still be identified after applying dynamic boundaries. The first issue is that sometimes the car drives with two tires on the track and two tires on the grass. The second problem is that for certain course layouts with unique corners, the network does not adapt the speed correctly and sometimes overshoots it. Both issues can most likely be resolved by providing more training data which contain more circuit layouts the network can learn. A bigger training data base also means that data balancing can be done by decreasing the number of frames where the car is only accelerating. This avoids overfitting and helps the agent to make the right assessment in rare cases, like steering on a straight for a brief amount of time to align the yaw angle with the street.

Another point which has an impact on the performance is the difference between rendered outputs on a local machine and *Google Colab*. The training data samples used in this project were all recorded on a local machine. Hence, they differ from the actual rendered image used during submissions. This causes inconsistencies in the score when an identical agent is used on the same track. Furthermore, the agent is optimized for locally generated images, so likely has a worse performance on *Google Colab*. This issue can be resolved by recording all training samples on the remote server instead of a local machine.

5. Outlook

This chapter describes some methods that can be applied to further increase the performance of the network.
The correct selection of hyperparameters like learning rate and momentum for the SGD optimizer has a strong impact on the accuracy of the network. If the learning rate is set to an incorrect value, it might happen that the minima cannot be found by the optimizer. To improve this, a decaying learning rate can be used, which typically has a big value at the start of the training and gets smaller while training. This improves the quality of the optimizer immensely.
Furthermore, since the application of the gaussian noise function at the beginning and dropout within the fully connected layer showed good results, it could be worthwhile to use different mean and standard deviation values for the noise function and apply them to different layers of the network.
One of the biggest issues is still the fact that with our implementation it is hard to tell when the network is overfitting and when its trained well enough. So a big increasement would be to find out about a method which enables us to find the sweet spot of the network. The sweet spot could be compared a lot better with different training settings and to optimize the boundary adaption of the output.
Another big improvement would be to find access to more accurate speed values, due to the small size of the saved images there are only 8 pixels available to display the speed bar, which results in only 9 different speed states. While driving the game locally the speed bar has a much higher resolution, thus delivering more accurate speed values. The neural network should perform better if the speed value is more accurate.

6. Literature

[1] G. Huang et al.: "Densely Connected Convolutional Networks," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017*

[2] Kiefer, J., and J. Wolfowitz. “Stochastic Estimation of the Maximum of a Regression Function.” *The Annals of Mathematical Statistics*, vol. 23, no. 3, 1952

[3] Sutton, R. S: „Two problems with backpropagation and other steepest-descent learning procedures for networks.” *Proc. 8th Annual Conf. Cognitive Science Society*, 1986

[4] Srivastava, Hinton, Krizhevsky, Sutskever and Salakhutdinov: “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, *Journal of Machine Learning Research 15*, 2014

[5] Brownlee, Jason (2018): Train Neural Networks With Noise to Reduce Overfitting. In: *Machine Learning Mastery*, 11.12.2018. Online verfügbar unter https://machinelearningmastery.com/train-neural-networks-with-noise-to-reduce-overfitting/, zuletzt geprüft am 17.06.2021.