Duckietown: Right Side Lane Segmentation Network

Duckietown: Jobb oldali vezetősáv Szegmentáló Hálózat

Máté Nyikovics

Márton Tim

Consultant: **Róbert Moni**

Anikó Renáta Szabados

Abstract

Background: Autonomous systems have gained ground since we have been able to use stronger and faster computers on large databases. In the industry, we are experiencing large-scale development of self-driving cars. As such development is very costly and time consuming, it was a given to create a low budget project to help with this. *Goals*: The purpose of this paper was to create a network capable of semantic segmentation of the right lane within the Duckietown project. *Methods*: We generated our data ourselves and used a modified simulator in which Duckiebot was able to capture both the original and the augmented images. For the mesh, we chose an encoder-decoder-based system that met our goal of working with semantic segmentation. Finally, the model had 15 million teachable parameters running for up to 50 epochs, using early stopping to avoid overfitting. *Results*: Our model was able to predict the right lane of the road with an error rate of only 2%.

Absztrakt

Háttér: Az autonóm rendszerek egyre nagyobb teret nyertek mióta lehetőségünk lett erősebb és gyorsabb számítógépeket használni nagy méretű adatbázisokon. Az ipart tekintve az önvezető autók nagy léptékű fejlesztését tapasztaljuk. Mivel egy ilyen fejlesztés nagyon költséges és időigényes, adott volt, hogy létrejöjjön egy kis költségvetésű projekt ennek segítésére. *Célok*: Jelen dolgozat célja az volt, hogy a Duckietown projekt keretein belül, készítsünk egy olyan hálózatot, amely képes szemantikus szegmentációra a jobb oldali sávot illetően. *Módszerek*: Adatainkat magunk generáltuk és az adatkészítés során egy általunk módosított szimulátort használtunk fel, amiben a Duckiebot egyszerre volt képes begyűjteni az eredeti és az

általunk augmentált képeket. A háló esetében egy kódoló-dekódoló alapú rendszert választottunk, ami megfelelt annak a célunknak, hogy szemantikus szegmentációval dolgozzunk. A modellben végül 15 millió tanítható paraméter állt rendelkezésünkre, melyet 50 epoch-ig futtattunk, early stoppingot alkalmazva az overfitting elkerülésének érdekében. *Eredmények*: A modellünk képes volt megjósolni az út jobb sávját, mindössze 2%-os hibaaránnyal.

INTRODUCTION

Duckietown is an autonomous driving test platform that aims to help the teaching and development of self-driving tools and skills. It was first developed by MIT in 2016 and since then its community is steadily growing, both in number of supporting universities and in students choosing to build in this framework. Our project was prepared partly with regard to the aim of building a fully autonomous Duckiebot (the AI assisted small robot car) but in the end we settled on detecting the right lane in a simulated environment. The results of this project will hopefully allow other researchers from our or other universities to obtain better performing self-driving vehicles.

The goal of self-driving cars is to become a very efficient and most comprehensive application of autonomy. However, self-driving cars still face many challenges. Challenges include algorithm codes, parallel interactions between detection and control, and the optimal distribution of finite computing resources for concurrent processes. The financial burden of such a project is heavy too, when it comes to building full-size vehicles, and creating a fleet of them, and it also raises many safety issues. On the other hand, infrastructure development is time consuming and requires significant sums for tasks not directly related to the project. To solve these problems, the Duckietown Project, an open source platform for autonomy research was created. This includes autonomous vehicles called Duckiebots, which use only the Raspberry Pi 2 for all calculations, and a monocular camera for detection, but the Duckiebots, despite their simplicity, are capable of complex behavior [1].

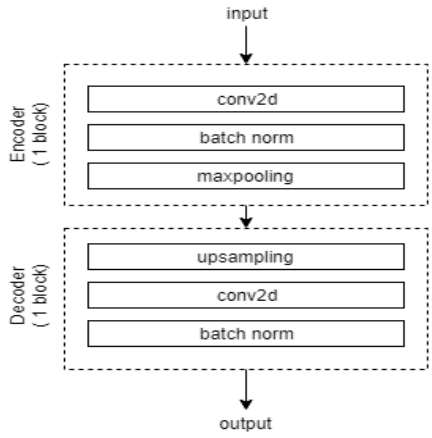
In the case of autonomous cars, the structure of lanes plays a key role in safe driving, especially in an urban environment. In order to the car to move and steer safely, it is essential that you are able to determine the current lane positions, otherwise the vehicle cannot drive safely. Currently used autonome car systems usually use an accurate and detailed environmental map, but they are not completely reliable in many cases. This may be the case when roads are being reconstructed and rebuilt, which is a common occurrence, so capacity of frequent renewal of maps is also key to achieving self-driving. Another problem is that the map-based approach requires very precise location of the car, which is a cumulative difficulty in the city, due to the many narrow streets. Meyer, Salscheider, Orzechowski and Stiller draw attention to this in their 2018 study and according to their theory, it would be much more efficient to have self-driving cars equipped with sensors instead of maps, which would provide accurate and efficient traffic guidance without maps. Based on this hypothesis, they created a deep neural network that was able to successfully and efficiently detect the location of the bands [2].

Due to industrial changes in recent years, which have also affected computing, we have been able to use more powerful computers to study large volumes of data using deep neural networks that go beyond the applicability of previously used algorithms [3]. Because of their size and performance, these networks are very computationally expensive, but they are very needed because they can understand the amount of data that other systems have not been able to process. Deep neural networks have also made tremendous progress in visual understanding. Such is the analysis of the images of modern cameras that are often found in vehicles. These visual inputs are based on many autonomous systems in which semantic segmentation plays an important role. During semantic segmentation, each pixel in an image receives a tag that will belong to a semantic class. It accounts for a large part of semantic segmentation in the development of self-driving cars, where you need to recognize and classify boards, other cars, pedestrians, or lanes [4]. These pixel-based labels were used to make predictions of a category with deep networks help, which they were able to do based on the results [5][6].

In semantic segmentation, it is important to have a good definition of the edges, so that you are able to properly recognize the entities, like other cars and lanes. The encoding-decoding structure is excellent in this edge definition. Typically, encoder-decoder networks include, on the one hand, an encoder module that reduces feature maps and records higher level semantic information, and they also include a decoder module, which in turn recovers spatial information. These models allow for faster computation along the coding path, while decoding gradually returns the boundaries of the objects [7] [8].

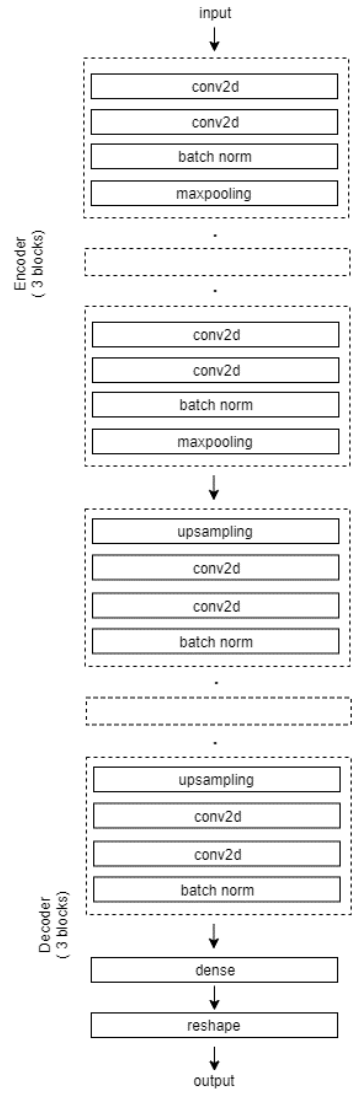
NETWORK ARCHITECTURE

Designing the network architecture was a challenging task, for inspiration we first investigated other popular and well performing models in the semantic segmentation field (SegNet [9], U-Net [10][11], etc.). From these models, we found that all of these share a common architectural feature, a so-called feature extractor, so we started by constructing one of our own. A feature extractor is basically an autoencoder for images, a series of 2D convolution, max pooling, up sampling and batch normalization layers in an allotted order.



1. Figure Basic feature extractor.

The convolution layers in the feature extractor are usually activated by a ReLU activation, we did not deviate from this. The outputs of the feature extractor are the features of the image, however we needed the probability of a pixel being in the right lane, so to fulfill this purpose, we added a dense layer with sigmoid activation to the end of the model. Finally, we concluded the model with a reshape layer meant to remove the last 1 ranked (channel) dimension. The resulting model looks like this:



2. Figure Network model.

DATA ACQUISITION

To be able to correctly build and train a neural network a vast amount of data is required. This data could either be manually created (i.e. annotated by real persons) or generated by a software. In the latter case only simulation data is realizable as computer

generated annotations of non-simulated inputs should also be checked by a human supervisor. As we did not find a publicly available dataset neither designed for right lane segmentation nor for segmentation in Duckietown environment we had to find an alternate source of data.

Data generation

A simulation program was shared on GitHub for Duckietown that was called Gym and it was originally designed for helping out transfer learning and reinforcement learning projects. Transfer learning is the process where we train a classifier using simulated inputs and outputs and the challenge is to find a way to transfer the results of this trained network to the real world with constantly changing and noisy inputs.
This Gym was modified in such a way that it collects samples of images seen by the simulated Duckiebot while also acquiring the annotated output for each input image. An annotated output is practically a binary image, with all the right lane pixels set to 255 and all the non-right lane pixels set to 0. An example for an original and an annotated image can be seen in Fig x. These streams of images were saved as videos in case we would like to use it with recurrent classifiers.

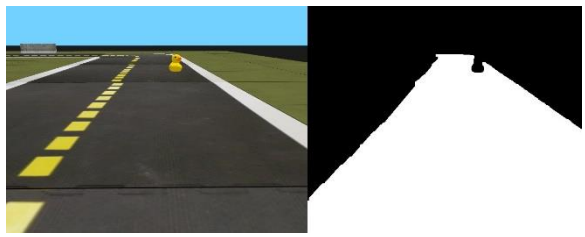


Figure 1. An example pair of original (left) and annotated (right) image.

The videos were then distributed to the usual three datasets that were train, validation and test sets with their ratio kept at about 60-20-20% with set size measured in number of videos. As the length of videos were not the same, this method of distribution resulted in a slightly different number of images allocated to each dataset than what would be otherwise expected from the ratios.

Data preprocessing

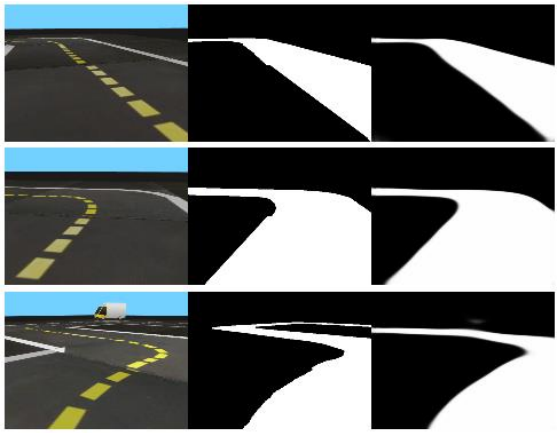
As the originally created database contains image streams, first, they had to be separated to single images in order to train a single-image classifier. This was done in-place in the same environment where training was due to happen. For the generated images we assigned a single, 6-digit long number as the file name. As the size of data was relatively large, a data loader was designed with the aim to load batches of data dynamically, thus keeping the required size of system memory small, and also to select random combinations of those input images, as shuffling greatly reduces the effect of the fact that our pictures were saved in the same order as they appeared in the videos. Then the image pixels were rescaled from 0-255 to 0-1 using linear scaling.

TRAINING

After we got the input and output images/labels for all three categories, we used a generator function to divide them into smaller batches. A training batch consisted of 68 randomly selected images, and we used 20 of these batches for one training iteration (epoch). As for the network hyperparameters, we doubled the convolutional filter count starting form 64 in each encoder block, then halved them back in the decoder blocks, while we kept the convolution kernel sizes a constant 8 by 6, we found this to be a good compromise. With this configuration we had over 15 million parameters to train. For the actual training we used Adam optimizer and binary cross entropy as loss function. We ran the training for 50 epochs while employing early stopping to avoid overfitting and learning rate decay to improve accuracy. During training we saved the best model states based on validation accuracy.

EVALUATION

The model trained quickly, it always stopped after a good 40 epochs. This took around 1h on Google Colaboratory with GPU accelerated session. During this time, it managed to get over 98% of testing accuracy. If one were to interpret the output not as a probability distribution, rather a black and white image, the results would look something like this:

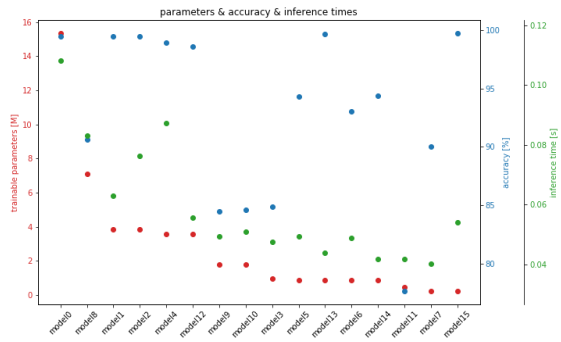


3. Figure Three prediction results on the test dataset. The order of images (from left to right) are input image (camera image), the expected output, and the predicted output.

From the predicted samples one can see that the model was indeed able to make good predictions, however it had trouble with complicated road patterns in the distance. This fact accounts for the approximately 2% error rate. Our guess it that the kernel size of the model is too big to be able to accurately learn these fine details, but we didn't want to reduce it, because it would have increased the parameters of our already substantial model. To combat this effect, we recommend to simply reduce the FOV (Field of View) of the input image, to get a larger viewing angle.

OPTIMIZATION

When we were done with the initial model and result evaluation, we applied optimizations to our model, keeping the inference time in mind as the main parameter to optimize for. Our initial presumption was that inference times are tightly correlated with the trainable parameter count of the model, so we tried to minimize these. We approached this task from two different angles. Once we tweaked the hyperparameters of the model, whilst the second time we searched for more optimal network architectures on the web. As for the hyperparameter tweaking part, we considered the input shape, the convolution filter count and the kernel size. Apart from the original configuration, we halved each parameter and trained the network with every possible variation of these. This resulted in 2^3=8 variations (including the original), but then we went a little further and modified the network architecture in small ways (like removing every 2nd convolution layer, or skipping the middle convolution block), because we wanted to know the effect of these changes as well. In total we have got 16 different models this way. We trained them using the same technique described under training section and measured their inference times by averaging the time of 100 predictions. The results of these measurements are shown in the scatterplot below.



4. Figure Results of hyperparameter optimization.

From the figure above, we concluded that our original assumption was indeed correct, the less the parameter count of the model the less time it takes for it to predict. Furthermore, we also saw that by choosing the model parameters carefully one can get serious performance gains without losing much of the accuracy. With this statement we are referring mainly to model nr. 15, which accomplishes the feats of the original model with only having $1/10^{th}$ nr. of parameters. This could beg the question why shouldn't one just keep decreasing the value of hyperparameters, when there is no noticeable accuracy loss? The reason is that by using ever smaller images, one truly doesn't necessarily lose prediction accuracy, but most certainly does lose image quality/details, and the effect detailed by 3. Figure gets more pronounced. It's a compromise one has to make, we deemed it not worth the degradation of quality, that's why we stopped with this part of the optimization where we did.

CONCLUSION

We feel that we achieved the goals we set out to do, designed, trained and optimized a network which could reliably segment the right side of a lane from the other parts of the road.

As a further goal we would much like to see how our model handles a real word environment, with all its additional challenges.

ACKNOWLEDGMENT

We would like to thank Róbert Moni the guidance and constant help he provided us during the development process.

¢ REFERENCES

1. Paull, Liam, et al. "Duckietown: an open, inexpensive and flexible platform for autonomy education and research." 2017 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2017.
2 Meyer, Annika, et al. "Deep semantic lane segmentation for mapless driving." 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2018.
3. Paszke, Adam, et al. "Enet: A deep neural network architecture for real-time semantic segmentation." arXiv preprint arXiv:1606.02147 (2016).
4 Treml, Michael, et al. "Speeding up semantic segmentation for autonomous driving." MLITS, NIPS Workshop. Vol. 2. 2016.
5. Farabet, Clement, et al. "Learning hierarchical features for scene labeling." IEEE transactions on pattern analysis and machine intelligence 35.8 (2012): 1915-1929.
6. Chen, Liang-Chieh, et al. "Semantic image segmentation with deep convolutional nets and fully connected crfs." arXiv preprint arXiv:1412.7062 (2014).
7. Chen, Liang-Chieh, et al. "Encoder-decoder with atrous separable convolution for semantic image segmentation." Proceedings of the European conference on computer vision (ECCV). 2018.
8. Badrinarayanan, Vijay, Alex Kendall, and Roberto Cipolla. "Segnet: A deep convolutional encoder-decoder architecture for image segmentation." IEEE transactions on pattern analysis and machine intelligence 39.12 (2017): 2481-2495.
9. Badrinarayanan, Vijay, Ankur Handa, and Roberto Cipolla. "Segnet: A deep convolutional encoder-decoder architecture for robust semantic pixel-wise labelling." arXiv preprint arXiv:1505.07293 (2015).
10. Zhang, Z., Liu, Q., & Wang, Y. (2018). Road extraction by deep residual u-net. IEEE Geoscience and Remote Sensing Letters, 15(5), 749-753.
11. Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation." International Conference on Medical image computing and computer-assisted intervention. Springer, Cham, 2015.