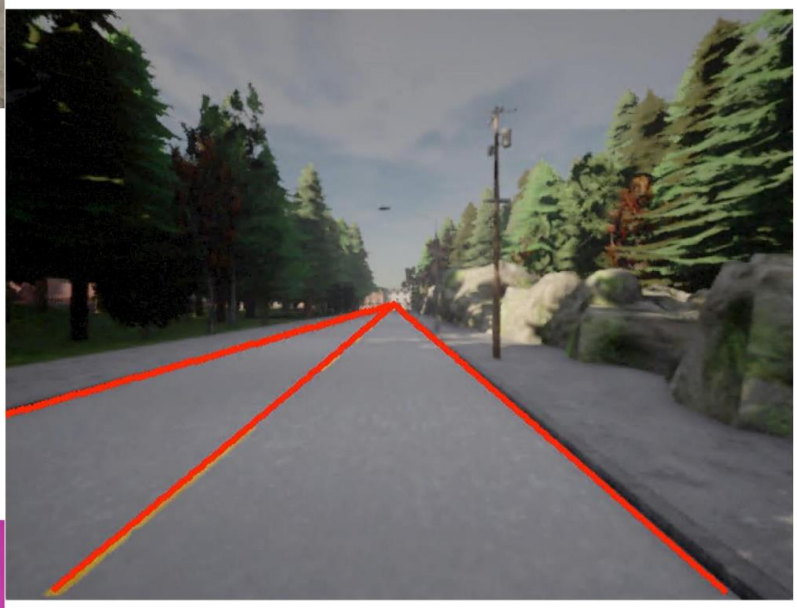
howest / hogeschool

OBJECT DETECTION



LANE DETECTION

SEGMENTATION

TRANSFORMERS FOR COMPUTER VISION TASKS WITHIN SELF-DRIVING CARS.

INTERNAL PROMOTOR: NATHAN SEGERS
EXTERNAL PROMOTOR: SEAN NACHTRAB

RESEARCH QUESTION PERFORMED BY

ALEC HANTSON

IN FULFILLMENT OF BECOMING A BACHELOR IN

MULTIMEDIA & CREATIVE TECHNOLOGIES

HOWEST | 2021-2022

Preface

My name is Alec Hantson, a New Media & Communication Technology student at Howest University of Applied Sciences, West-Flanders in Kortrijk. This undergraduate thesis is the culminating work of my education and will address the following research question: Can the use of transformers grant any advantages to self-driving cars?

In recent years transformers have become very popular in the world of NLP. Meanwhile, some major players have also made progress in implementing transformers in computer vision. In this undergraduate thesis, I will take a deeper dive into the impact of these transformers on self-driving cars.

The conclusion I will be forming in this thesis is based on the research and technical demo I did during the “Research Project” module. I did some theoretical research during this project and built a demo to display how transformer-based models can handle object detection, lane detection, and segmentation. A self-driving simulation environment has been used as the base platform for the research. This way, I could show my work in an as realistic as possible scenario.

Before diving into the research, I would like to thank some people who helped me throughout the thesis. My internal promotor Nathan Segers for his valuable feedback. Jurriaan Biesheuvel data scientist at RIVM, for giving feedback and his advice and reflection. My external promotor Sean Nachtrab, Machine Learning Engineer at OTIV for his feedback, advice, reflection, and general support throughout the entire writing process.

Abstract

This undergraduate thesis will answer the research question: ” Can the use of transformers grant any advantages to self-driving cars?”. The most common computer vision tasks for self-driving cars include Object detection, Lane detection, and Segmentation. The undergraduate thesis focuses on using new techniques for these tasks.

Object detection is used to gain knowledge of the environment. Self-driving cars need to detect traffic lights and their respective color, traffic signs, and the speed they demand, cars, cyclists, and pedestrians. Lane detection is used to find the drivable lanes. Segmentation is often used to distinguish drivable and not drivable areas and to do a simplified form of object detection.

Current computer vision techniques like Yolo for object detection or Hough Transform for lane detection have become established techniques over the years. A lot of effort was put into optimizing these techniques, and that is why they generally seem to work well. We have seen a rise in transformer-based models during the last years, first in the NLP spectrum and now in the computer vision spectrum. These transformer-based techniques are still in the early stages but seem to show much potential.

While doing this research, more established techniques have been compared with new transformer techniques. For object detection and lane detection, this was done by training the models on the same datasets; this gives a fair comparison and gives a lot of insight into the model behavior. For segmentation, the box model was used, which means that no further training or fine-tuning has been performed.

After running both models on the simulator, it became clear that for object detection, the new transformer-based technique of the original DETR performs not only less accurately but also slower than the established YoloV5 model. This is the consequence of the transformer architecture, which is more complex and requires more training data. Luckily DETR has seen a lot of development since the original model, there are now improved versions that have an optimized architecture and require less data. For Lane detection and segmentation, the transformer-based techniques seemed to outperform the traditional techniques comfortably.

Table of contents

List of figures

Table of abbreviations

Abbreviation	Meaning
AP	**A**verage **p**recision
APl	**A**verage **p**recision of **l**arge objects
APm	**A**verage **p**recision **m**edium objects
APs	**A**verage **p**recision of **s**mall objects
DETR	**De**tection **Tr**ansformer
FCN	**F**ully **C**onvolutional **N**etworks
HT	**H**ough **T**ransform
hw	**H**eight*__w__idth
LSTR	**L**ane **S**hape Prediction with **Tr**ansformers
MLP	**M**ulti**l**ayer **P**ercepton
NN	**N**eural **N**etwork
SAT	**S**elf-**A**dversarial **T**raining
SOTA	**S**tate **O**f **T**he **A**rt
ViT	**V**ision **T**ransformers
mAP	**M**ean **A**verage **P**recision

Glossary

Term	Definition
Anaconda	An open-source toolkit that can be used to easily install/manage packages and environments.
CARLA	CARLA is an open-source simulator that is often used for autonomous driving research.
Roboflow	Software to annotate images.
Transformer	A deep learning model with an encoder-decoder architecture and makes use of attention.[1]
(m)AP	(Mean) Score that represents the overlap between the detected bounding box and the ground truth.

1 Introductions

1.1 Context

Self-driving cars have improved a lot in recent times, but there are still some serious flaws when it comes down to consistency and overall safety. Currently, self-driving cars have only reached level 2 autonomy, meaning the car can handle basic tasks like accelerating and steering but human oversight is still required for safety reasons and more complex scenarios. To reach level 5, meaning full autonomy, researchers and industry leaders are still looking for the most optimal approach to handle self-driving tasks. An essential step in reaching level 5 self-driving cars is improving perception systems that influence how the systems observe the world. Making self-driving cars understand their environment is achieved via various tasks including, but not limited to object detection, lane detection, and segmentation. These tasks now get completed by established techniques like Yolo or Fully Convolutional Networks.

A transformer takes a new approach to how deep learning models process data. Transformers are capable of learning from the global context instead of just instance by instance. Each part of the input space can refer to other parts. Learning in a more global approach opens up the possibility to discover patterns and regional knowledge that regular deep learning models are not capable of.

Transformers have been a hot topic in the world of NLP over the last couple of years. There have been many innovations that led to models like BERT or GTP3. Recently big AI players like Facebook AI, Google, Nvidia, etc., have put a lot of effort into using the transformer architecture for computer vision tasks. This turned out to be a big success with some very impressive scores across all kinds of computer vision tasks. Because of these scores, many people have been wondering if it is time to say goodbye to regular CNN models and time to move to transformer-based models fully.

This thesis will research if it is possible to improve computer vision tasks by using new transformer-based techniques instead of the more traditional approaches. Specifically, this thesis will answer the question: ”Can the use of transformers grant any advantages to self-driving cars?”. This thesis mainly focuses on object detection and lane detection, with minor attention to segmentation.

The conclusions made during this thesis are based upon technical research, theoretical research, and interviews with AI and computer vision experts with industry experience.

1.2 Sub questions

What are the most established techniques right now for computer vision tasks in self-driving cars?

To research new techniques within self-driving cars, it is essential to know the established techniques.

As mentioned before, Yolo is a very established technique for self-driving cars. Yolo is popular because it can make predictions very fast, which is vital while in traffic. Yolo is often paired with LIDAR sensors to be able to navigate dense traffic.

It is possible to go with a deep learning approach or a more mathematical approach for lane detection. The technical part of this thesis focuses on the latter one. Hough Transform is used because of its simplicity and speed of detection, it might not be the best option in a production environment, but it works well enough to set a baseline. It might be better to go with a deep learning model like Spatial CNN’s in a more production scope.

Many different models can achieve real-time detection, like FCN or DeepLabV3+. These models were not covered during the technical demo but will be handled in the theoretical research.

For what can transformers be used for self-driving cars?

Transformers are becoming more common in the AI field, for self-driving cars they mostly show potential to take over computer vision tasks. While it is still in the early stages, researchers are trying out transformer-based structures for the actual controlling of the car to create an end-to-end driving model.[2]

What are the advantages of using transformers for self-driving cars?

Transformers have a simple base structure and are very scalable. By using attention, the model can create a better understanding of the entire scene and thus make more accurate predictions. Because of the attention module, it is also possible to gain a better understanding of how these models make their predictions, and what they are *"looking"* at. Lastly, most transformer-based models are pre-trained, meaning you can get an outstanding performance out of the box and finetune them to your use case.

What are the disadvantages of using transformers for self-driving cars?

Nothing is perfect, and neither are transformers. If you have a specific use case that requires you to train a model from scratch, training can be very costly and require a large amount of data. Accessory to this they often require more computational power to have a low inference time.

What are the possibilities for simulating self-driving cars?

When trying to improve self-driving systems, it is crucial to develop and test them in an as realistic as possible environment. Luckily there are multiple options available. During the research, three simulators have been investigated, CARLA, Deepdrive, and AirSim. All of them are open-source and built upon the Unreal Engine.

1.3 Technology

This thesis will use and compare various techniques to create a general understanding of what is to be seen as the best approach. The following techniques have been used during the technical section.

Object detection

Used traditional technology as the baseline: YoloV5[3]

Used transformer technology: Original DETR[4]

Lane detection

Used traditional technology as the baseline: Hough Transform

Used transformer technology: LSTR[5]

Segmentation

Used traditional technology as the baseline: N/A

Used transformer technology: Segformer-b2-finetuned-cityscapes-1024-1024[6]

1.4 Dataset

Object detection dataset

To gain a complete understanding, both the traditional Yolo technique and the new DETR model have been implemented. The two models both received the same dataset as input. For Yolo, this means training, for DETR, only finetuning.

The data has been collected from the CARLA simulator and has been manually annotated on ten different classes:

- Vehicle
- Person
- Traffic_light_green
- Traffic_light_yellow
- Traffic_light_red
- Bike
- Motobike
- Traffic_sign_30
- Traffic_sign_60
- Traffic_sign_90

In total the dataset consists of 1864 images with 3240 annotations.

Lane detection dataset

Creating the lane detection dataset has been done with the help of an open-source repository.[7] The script collected around 100000 images and their corresponding lane points. This data could be used to train the LSTR model from scratch.

Segmentation dataset

For segmentation, there was no dataset used. The goal of segmentation with Segformer was to show the capabilities of a pre-trained transformer model without having to do any training or fine-tuning.

1.5 Working environment

CARLA

The technical research part of this thesis has been developed for and tested in CARLA, which is a simulator for autonomous driving research.

Google collab

Google provides free GPU-powered notebooks that are very useful for quick experimenting or training smaller models. Because there are specific time limits, it is impossible to train extensive models here. During the technical research, google collab was used to train the YoloV5 model.

Pytorch lightning

While fine-tuning the DETR model, PyTorch lightning was used. Pytorch lightning code is more structured than regular PyTorch, and it makes use of a more abstract approach. Lightning has some nice extra features like a built-in progress bar and tensorboard logs. There are some other differences around training on multi-GPU or TPU, but that was not important for this project. The syntax stays the same as regular PyTorch.

Roboflow

Roboflow was used to annotate the object detection dataset manually. Roboflow makes it easy to annotate by using big pre-trained models to help you detect simple objects. To increase the size of the dataset and improve the robustness of the model, you can easily add augmentations to a certain percentage of your images.

2 Research

This chapter of the thesis will discuss all possible techniques for each task. This will give a broad overview of all models and a deep dive into their inner workings. Secondly, a closer look will be taken at simulation environments. Discussing all these topics will also answer the sub-questions more in-depth.

2.1 How does a transformer work?

Transformers were first introduced in the paper “Attention is all you need” by Google Research[1]. It introduces a new approach to sequence modeling. A Transformer is a kind of neural network that uses an encoder-decoder architecture with an attention module. What makes a transformer different from other sequence models is the ability to pass the input in parallel instead of step by step, this is done by having multiple identical encoder and decoder layers. The original transformer architecture mainly focuses on the NLP field, but it is essential to understand it before diving into the computer vision models. Let’s take a closer look into how the original transformers were trained.[8]

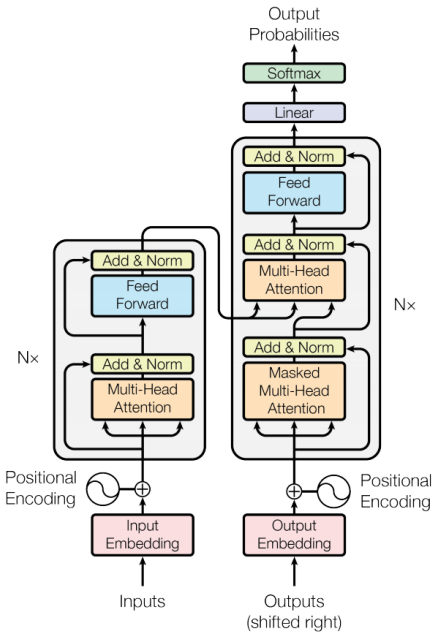


Figure 1 - Transformer architecture[1]

Encoder

Before the input can be passed into the encoder, every word must be mapped to a word embedding. Then the positional encoder gives context to words in a sentence based on how far away they are from each other. After these two steps, you now have a word vector with information about the context.

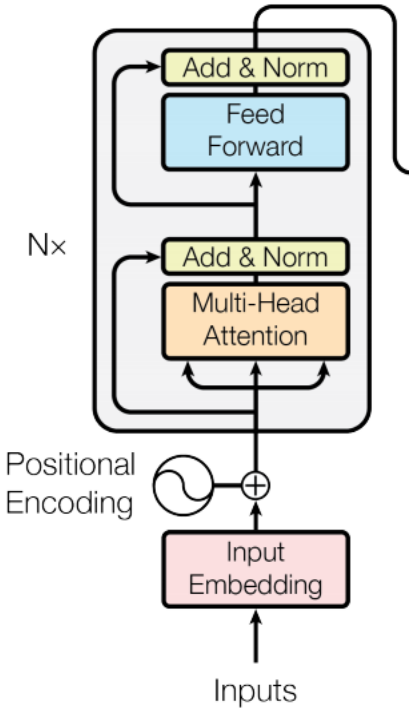


Figure 2 - Encoder architecture[1]

The vectors can now be passed to the encoder. First, it will go through the Attention block. The attention module returns a vector that captures contextual relations between input vectors. The next part of the encoder is a feed-forward NN. This is just a regular NN that is applied to every attention vector. The primary purpose of the NN is to transform the attention vectors into an understandable form for the decoder block.

Decoder

The first steps of the decoder are similar to the ones from the encoder; again, embeddings and positional encoding is used to translate the expected output to vectors that the model will understand. The output vectors go through a first attention module to learn contextual information, just like in the encoder block.

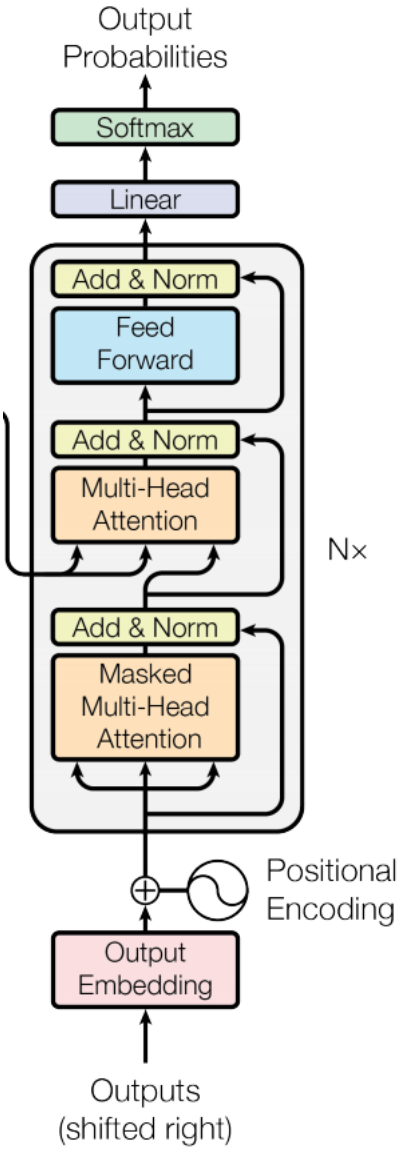


Figure 3 - Decoder architecture[1]

Now we get to a new module that takes the attention vectors that have just been created and, together with the vectors from the encoder block, feeds them into a new attention module. This attention module will try to learn how the input vectors are related to the output vectors. This returns a new attention vector for every input and output vector representing their relationship to every other vector in both the input and output.

At the end of the decoder, we again pass every attention vector into a Feed-Forward NN; this is done to make output understandable for the linear layer. The linear layer is also a Feed-Forward NN that converts the output to the desired dimensions. Lastly, the softmax layer converts everything to a probability distribution, which can be used to get the final output based on the highest probability.

Attention

During the encoder and decoder part, the attention block was already mentioned, but let's take a closer look.

The attention module uses self-attention to learn what part of the input it should focus on. Using self-attention determines how important the current part of the input is relative to the other input parts. For sentences, this could be how important is the first word relative to the other words in the sentence.

If you pay close attention to the architecture graphs in Figures 1,2 or 3, you can see the attention blocks are called Multi-Head attention. This means there has been made use of multiple attention vectors, 8 in total. The concat and linear layers take a weighted average of all these layers and create one weighted vector to continue with. Doing this prevents the attention vector from weighing the relationship with itself much higher than with other vectors.

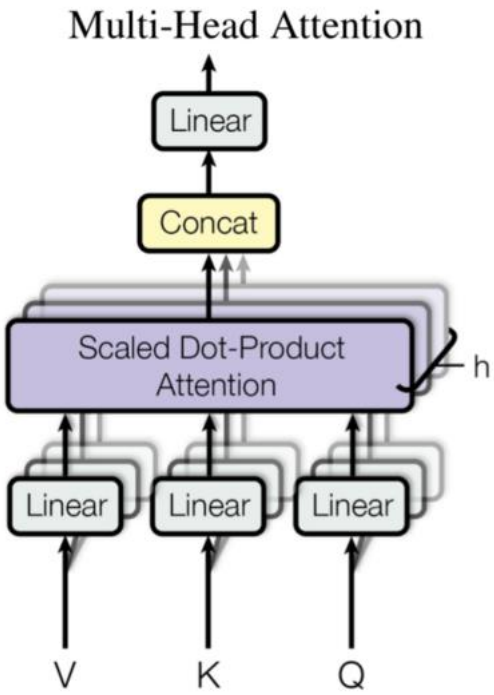


Figure 4 - attention module architecture[1]

In the decoder block, the first attention module is called a masked multi-head attention block. The reason it is called a masked block is that it would not be a good idea to give the decoder all the correct outputs from the start. Otherwise, there would be no learning process. The vector gets masked with zeros, allowing the model to learn.

Add & Norm

After each layer, there is an “Add & Norm” module, used as normalization. Transformer architecture uses layer normalization to speed up the process and help stabilize the learning process.

Conclusion

Now that we have a better understanding of how the original transformer architecture works, we can go ahead and see how this can be used in computer vision tasks.

2.2 Object detection

During this section, an explanation will be given of the working on YOLOv4[9] since this is a very commonly used and well-established model. This will also further answer the sub-questions. After that, the transformer-based approaches will be discussed, more specifically: DETR[4], Deformable DETR[10], and DINO[11].

Self-driving cars use object detection to detect things like but are not limited to: traffic lights, pedestrians, and other vehicles. Object detection for self-driving cars needs to be not only accurate but also fast to be able to operate successfully in traffic. This thesis will evaluate these models by mAP for the accuracy of the detection and the detection speed.

2.2.1 YOLOv4

The reason we will be looking at YOLOv4 instead of YOLOv5 is because of all the controversy around YOLOv5 and the lack of paper. YOLOv4 is the fourth iteration of the original YOLO[12] model, it was released in May 2020 by Alexey Bochoknovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao.

YOLO models have become well known for their excellent performance when used for real-time object detection. This fourth iteration improves upon YOLOv3 by an increase of 10% higher AP and a 12% higher FPS. While creating YOLOv4 there has been a lot of thought put into efficiency, it is possible to train a YOLOv4 model on just 1 regular GPU with 8 or 16GB of RAM.

Architecture

YOLOv4 is built upon the CSPDarknet53 backbone. CSPDarknet53 is a CNN pre-trained on the ImageNet dataset. According to their paper, they also experimented with CSPResNext50 and EfficientNet-B3 but achieved the most satisfying results with the CSPDarknet53 backbone. The backbone is used as a feature extractor.

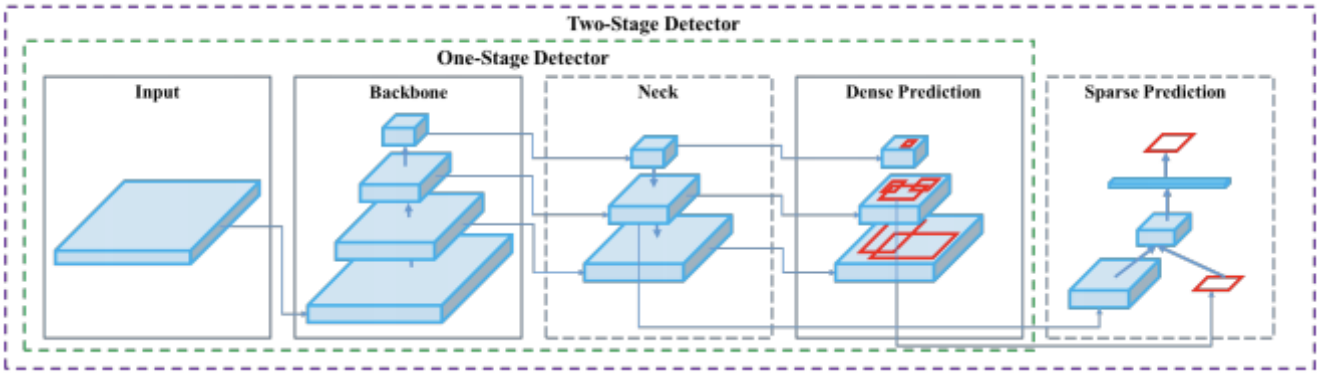


Figure 5 - YOLOv4 architecture[9]

Once the features get extracted by the backbone, they are passed to the Neck stage where they get prepared for the detection step. The neck works as a feature aggregator that collects feature maps from the backbone. YOLOv4 uses PANet as a feature aggregator, but with little to no mention of why. Besides the feature aggregator there is also been made use of spatial pyramid pooling, this helps to increase receptive fields and gathers out only the most important features from the backbone by working on a deeper stage of the network hierarchy.

The detection step a.k.a. the YOLO head remained the same from the previous iteration of YOLOv3. In the head, the first step is to detect in which area an object can be found. At this point, the model is not aware of what object can be found in the area. Second, a backbone model like Faster R-CNN or R-FCN is used to predict the object in the proposed area.

Bag of Freebies

YOLOv4 makes use of multiple data augmentation techniques to improve detection and classification. Some of these techniques are labeled as a bag of freebies because they do not add any inference time in production. The following techniques have been used:

- Cropping
- Rotation
- Flip
- Hue
- Saturation
- Exposure
- Aspect

All of these techniques have been around for a while and already proved their usefulness. YOLOv4 also introduces two new data augmentation techniques. Mosaic tiles merges four images into one. This helps the model to detect smaller objects and to focus on the objects themself rather than the surroundings. The second one is Self-Adversarial Training, the purpose of this is to find areas in the images where the model relies upon during training, these areas are edited, forcing the model to learn different features. It is possible to change the augmentations when training on a custom dataset.

The YOLOv4 researchers used Complete Intersection over Union loss. This works well because it takes into account not only the distance between the center points of the prediction and the ground truth but also the overlap between them. [12]

Bag of Specials

Bag of Specials contains different strategies that add a small amount of inference time but improves the model significantly. [13]

One of these strategies concerns the activation function. YOLOv4 does not make use of a traditional activation function because they do not push feature creations to their optimal point. Instead, it uses Mish, which is an activation function designed to push signals to the left and right. To separate bounding boxes, the model uses Non-Maximum Suppression. This has been done by using Distance IoU. When multiple bounding boxes are predicted over one object they will be merged into one.

As mentioned before training efficiency was taken into consideration when creating YOLOv4. That is why they made use of Cross mini-batch Normalization so that training on 1 GPU stays possible.

2.2.2 DETR

Detection Transformer was created in 2020 by Facebook AI.[4] It introduced a new way to do end-to-end object detection by using transformers. DETR still uses a CNN but only for the feature extraction part, deciding where the bounding boxes need to be placed is done by a transformer architecture. As already mentioned in chapter 2.1 with the transformer architecture it is possible to predict all the bounding boxes in parallel and make use of bipartite matching. Let's take a closer look and find out why this is powerful.

Architecture

The DETR architecture consists of multiple building blocks. To fully understand how DETR works and where its power lies, we need to take a look at it one block at a time.

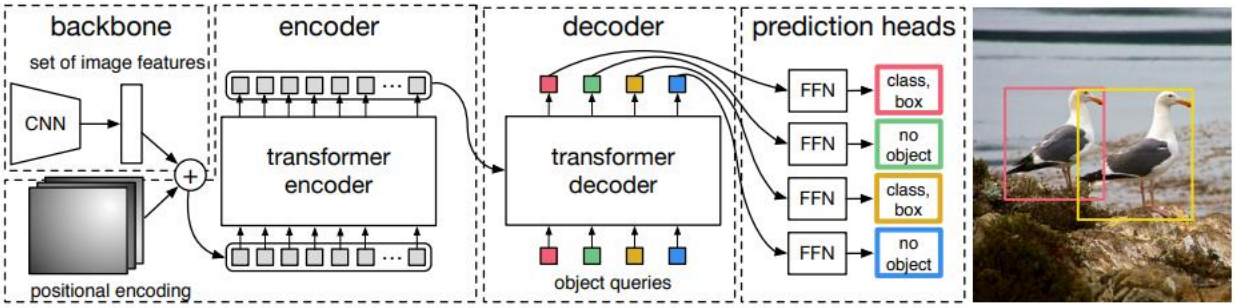


Figure 6 - DETR architecture[4]

Backbone

The first step is to feed the input image into a CNN to do feature extraction. CNN's have been proven to handle this very well. This returns image features in an image structure but with more than 3 channels (RGB).

Encoder

Before the image features can be passed into the transformer, they need to be flattened into size hw x hw, so one value for every pixel in the image. This way we can feed the image into the encoder at once. Now the encoder will use its attention module to learn contextual relationships between pixels based on the image features. This makes sense because now the model has a larger understanding of what the image looks like, which can help to predict bounding boxes more accurately.

Decoder

The decoder input consists of N number of object queries. N is a fixed number of bounding boxes that need to be detected. This will be discussed further later on.

At the start, an object query is a vector that has been randomly initialized. Once training starts, they get updated. An object query learns where to look in the image features to find valuable information.

Prediction heads

The prediction heads consist of tuples that contain information about the bounding box *(class, position)*. The position is a regular x,y coordinate. A class can be one of the predefined classes that the model needs to detect or it could be the Ø class, this class is added before training but does not need to be annotated when creating the dataset.

Bipartite matching

DETR always predicts the same amount of bounding boxes, this can be configured during training but in general, you will always have a fixed output of N bounding boxes. When training on a dataset the annotations get padded up with Ø class bounding boxes to match the fixed N number.

Because of the Ø class, you cannot just compare classes one on one because the order in which they are predicted should not matter. This is solved by using Bipartite matching loss, it helps to compare classes and merge overlapping bounding boxes. It works by calculating how much a prediction and a ground truth have in common. For example, the same class and position result in an optimal low loss, totally different class and position results in a high loss. This loss function is applied to every possible combination between the predicted bounding boxes and the ground truth.

The goal is to assign every prediction to a ground truth that results in a minimum total loss. The matching can only be one on one, if a prediction is assigned to a ground truth it is not possible to

assign another one to that same ground truth. During training, the model gets extra penalized if it outputs too many bounding boxes with a class.

Besides Bipartite matching loss, IoU loss is used to add increase, or decrease total loss based on how much bounding boxes overlap. This is similar to what is used in YOLOv4.

Augmentation

Earlier in YOLOv4, different types of augmentations have already been discussed and how they make a big impact. For DETR this is no different. DETR uses multiple different kinds of augmentations to significantly improve performance. One of the augmentation steps is random resizing, this resizes the image to a dimension somewhere between 480x1333 and 800x1333. Besides random resizing, there is also random cropping. Randomly cropping the image helps the model to learn better global relationships. According to the Meta AI research team, this augmentation alone improved performance by 1 AP.

2.2.3 Deformable DETR

Deformable DETR is an improved version of the original DETR model. The paper was released in March 2021. It can achieve better performance than the original DETR and uses less computational power.[10] This is achieved by using sparse spatial sampling and a deformable attention module. The working of DETR has already been explained, let's see what Deformable DETR does differently.

Deformable attention

One of the main issues with DETR is its slow convergence speed and need for heavy computational power. The reason DETR is so slow is that (as mentioned above) all feature maps from the backbone get flattened. This gives a very long sequence (hw x hw), making it hard for the attention modules to concentrate on features that belong together.

Instead, deformable attention is used, it flattens the image into hw x K. The K values are locations somewhere in the image. The location of the K values is learned over time. So instead of trying to focus on everything Deformable DETR learns where to focus on.

Encoder

In DETR only the last layer from the CNN backbone gets used by the transformer to detect objects. It is possible and already done by other models to use multiple layers from the CNN output. These are called Multi-scale features. The top layers have lower resolutions, the lower you go down the layers the higher the resolution. This makes it possible to extract different features on different resolutions. The deformable attention explained above will be applied to multiple layers.

Two-stage Deformable DETR

Two-stage deformable DETR is inspired by traditional two-stage detectors. In this case, it will be implemented to improve the way object queries are used and make the model less computational demanding. In DETR the object queries are not relevant to the decoder block. This is changed by feeding the proposed regions (K) into the decoder as object queries to further refine them.

In the first stage, an encoder-only structure is used to assign an object query to each pixel and directly output bounding boxes. Doing this increases the recall. The top-scoring bounding boxes from stage one are used in the second stage as proposed regions. These regions are used in the decoder as initial boxes. Meaning the positional embeddings of the object queries are replaced by the positional embeddings of the proposed regions.

2.2.4 DINO

Like Deformable DETR, DINO is also an improved version of the original DETR model. It is meant to be a more efficient and better-performing model than both DETR and Deformable DETR. DINO uses techniques used by other DETR-like variants and combines them with new proposed techniques to create an optimal DETR model.

Architecture

The architecture of DINO is very similar to the original DETR architecture. It uses features from a backbone together with positional encoding and uses the encoder and decoder. DINO also introduces some new parts like query selection/matching and GT+Noise.

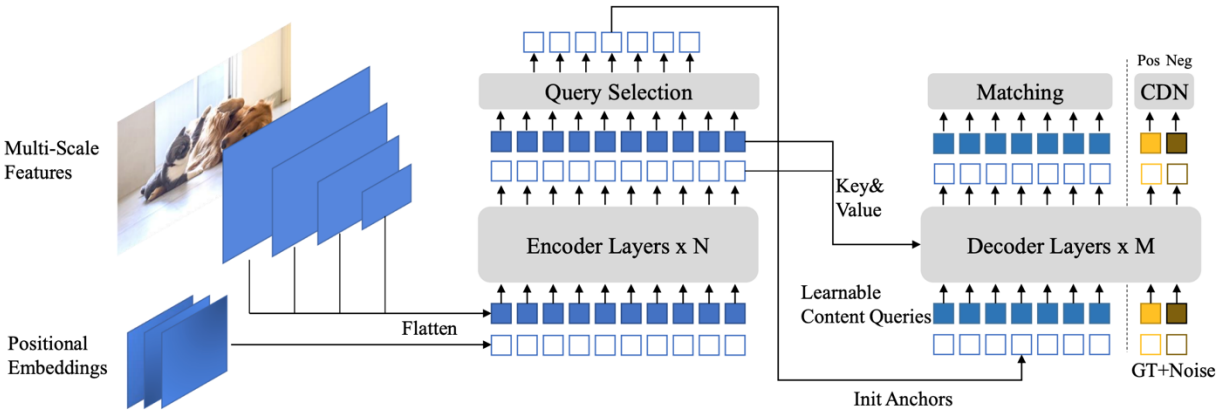


Figure 7 - DINO architecture[10]

Backbone

DINO uses a backbone like ResNet to extract features from the image. As explained above Deformable DETR uses Multi-scale features. DINO does the same thing. The multi-scale features together with the positional embedding get sent to the encoder block.

Mixed Query Selection

The query selection process is heavily based on the two-stage Deformable decoder. Both only use the top proposed regions (K) to feed into the decoder. The difference is that DINO only feeds the positional queries into the decoder while Deformable DETR also feeds in the content of the object queries. DINO does not do this because feeding in the content could be ambiguous and misleading to the encoder. The content queries stay learnable as proposed in the original DETR architecture. The model will now make more use of the positional queries to gather more content features from the encoder.

Look forward twice

Look forward twice is a new way of doing bounding box predictions. DINO does this by updating its layer parameters with the losses of the current layer and the next layer. The precision of a bounding box is thus determined by the quality of the initial bounding box and the offset of the prediction.

2.2.5 Comparison

	AP	AP50	AP75	APs	APm	APl	Epochs	FPS
YOLOv4 – darknet weights	47.1%	71%	51%	27.8%	52.5%	63.6%	300	62
DETR-DC5	43.3%	63.1%	45.9%	22.5%	47.3%	61.1%	500	12
two-stage Deformable DETR	46.2%	65.2%	50%	28.8%	49.2%	61.7%	50	19
DINO	47.9%	65.3%	53.2%	31.2%	50.9%	61.9%	12	24

[10][14] [9][11]

When looking at the table above we can conclude some takeaway points. YOLOv4 seems to outperform DETR and deformable DETR, both on AP scores and FPS. This is not surprising, as the YOLO model has seen a lot of iterations and optimizations. It had more time to improve and only focuses on object detection while the DETR and deformable DETR models also can be used for segmentation without the need for retraining. This is a big advantage in a production environment.

DINO cannot be used for segmentation but does perform very close to the level of YOLOv4 while only using 24 training epochs. This shows that DINO has even more potential if trained on more epochs. DINO currently holds the crown for SOTA models on COCO minival[15]. With a speed of 24 FPS, it theoretically runs fast enough to be used in real-time self-driving cars.

Conclusion

Now that all models have been explored and compared it is possible to conclude that for object detection it is in theory possible to switch to transformer-based models.

YOLO for now is the faster option and should be preferred if a very high FPS is required. While YOLO performance is very good and will continue to do so for quite some time, it is not possible to ignore the possibilities transformer-based object detection models bring to the table. When looking at the top 5 performing models on the COCO minival dataset, 4 of them are transformer based.[15]

They are only in their early stages but can already match and even exceed the accuracy of YOLO models and are fast enough to be used for real-time tasks. The potential for transformer-based object detection models seems to be bigger than traditional CNN models. The graph in figure 8 shows the performance value of the best-performing YOLO or DETR model on the COCO minival dataset over time.

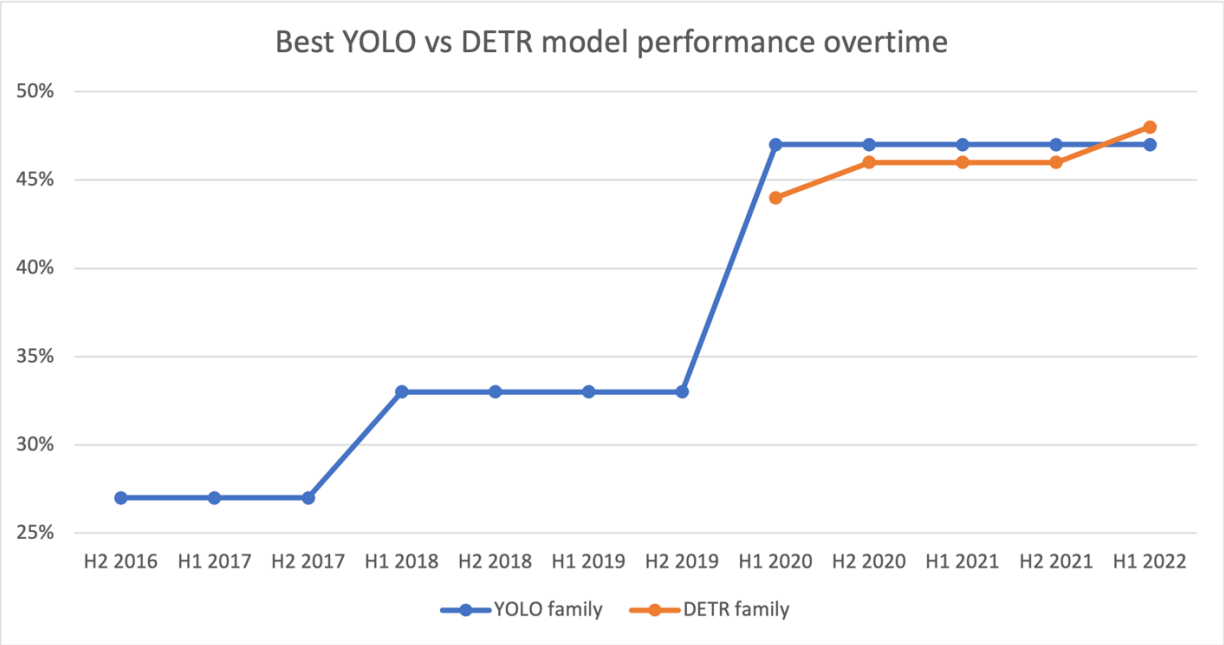


Figure 8 - YOLO vs DETR[4], [9]–[12], [15]

2.3 Lane detection

Lane detection is one of the most important computer vision tasks for self-driving cars. It helps the car understand in what area it should be driving and what route it should be following. The main problems that lane detection faces are that not all roads are the same and neither are the lines. Detecting lines can be impacted by various weather conditions such as rain, snow, or shadows created by the sun.

Lane detection can be done in a couple of different ways. In this section, I will investigate 3 different approaches. First off is Hough transform, this method does not make use of deep learning but a more mathematical approach. PolyLaneNet[16] as a pure CNN approach and LSTR[5] as a new transformer-based technique. Before comparing these techniques, it is important to understand how they work.

2.3.1 Lane detection with hough transform

This approach is different from the other 2 techniques in the sense that it does not use a deep learning model. Instead, it uses mathematical functions to determine where the edges of the road can be found. Connecting these edges results in a line representing the side of the road. It is a very basic technique but works well as a proof of concept.[17], [18]

Canny edge detection

Road lines on an image can be found by looking for sharp edges within the image. This can be done in three steps.

First, the image needs to be grayscaled, this is done to reduce the input to one channel. After grayscaling, a kernel is used to apply gaussian blur. Gaussian blur is based on a Gaussian function that sets the value of one pixel to the weighted average of its neighbor pixels. Doing this reduces the noise within an image.

Once the preparations are done the edges can be detected. This is done by calculating the change in brightness between neighboring pixels. If the change is bigger than a certain predefined threshold value, it is considered an edge point. Visualized this would look like a picture with only black pixels and white pixels representing edges. See figure 9 for reference.

Figure 9 - Edge detection

Region Of Interest

Before applying the Hough transform function it is a good idea to determine the image section of interest. After all, there is no point in trying to detect lanes in the sky. This can be done by creating a white triangle mask. The mask can be used to do an “and operation” on the image. If a certain pixel in the image is white and the pixel in the mask is white then it stays white, otherwise, it turns black. Doing this makes sure only the edges within the area of interest remain. (Figure 10)

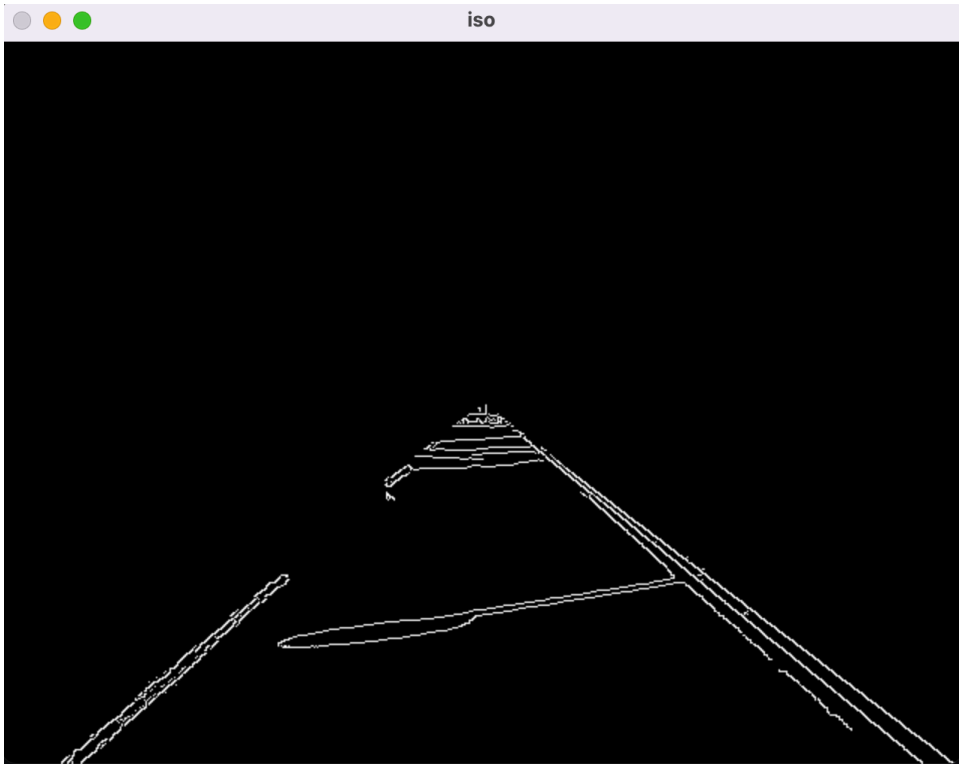


Figure 10 - Region of interest

Hough transform

The last step is to use the Hough transform function to detect lines based on the edge points. Hough transform will find points in an image that result in a line. An image can be seen as a plane with X and Y axes. The basic equation of a line is $y = mx + c$. Where m is the slope and c is equal to the intercept. On this plane, there are two points situated.

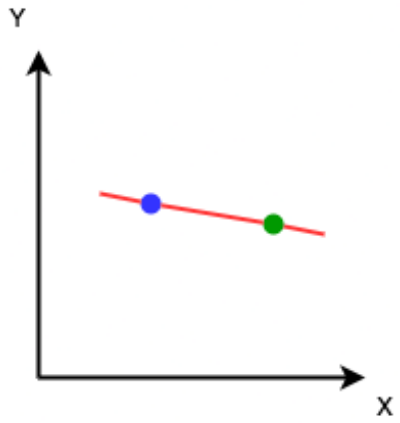


Figure 11 - Hough transform step 1

This plane can be converted into an MC plane or also called the feature space, resulting in a lot of different possibilities for one point on the original plane. Here we can find the intersecting point (m,c). This intersecting point represents the line in the original plane.

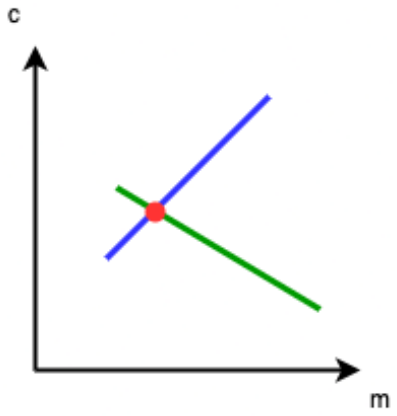


Figure 12 - Hough transform step 2

All edge points that are situated on one line in the original plane represent the same point in the feature space.

This technique will not work well on straight lines because the slope is infinite. To solve this the equation is changed to the polar coordinate system. $\rho = X\cos\theta + Y\sin\theta$. Where ρ = perpendicular distance from the origin, and θ = angle of inclination.

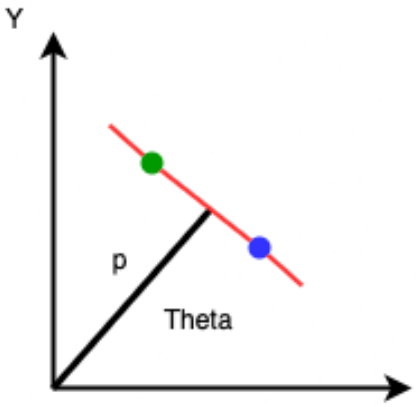


Figure 13 - Hough transform step 3

Now the Hough transform algorithm can be run on the edge image. It will check every pixel to see if it is an edge point. In case it is it will check all pixels on the same line in the feature space to see if it can detect any other edge points. This will create sinusoidal curves, the points where most curves intersect will be used to draw the final lanes.

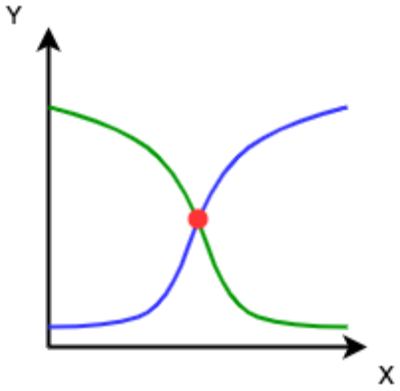


Figure 14 - Hough transform step 4

If you run the same function on the region of interest, you will get the final result as shown in figure 15.

Figure 15 - final Hough transform view

Because of its low computational cost, it is very easy to run this algorithm every few frames. The downside is that it lacks accuracy and has a hard time handling weather conditions like shadows on the road.

2.3.2 PolyLaneNet

PolyLaneNet is a deep learning approach for lane detection. It was introduced in 2020 and makes use of deep polynomial regression. It is a model designed to work accurately and in real-time with a very simple architecture. The advantage of the deep learning approach is that it tends to be more robust than the Hough transform approach. [16]

How it works

PolyLaneNet uses an end-to-end architecture. It takes the image as input and outputs the coordinates of each lane together with their confidence score. This confidence score needs to be higher than a certain predefined threshold for the lane to be considered usable.

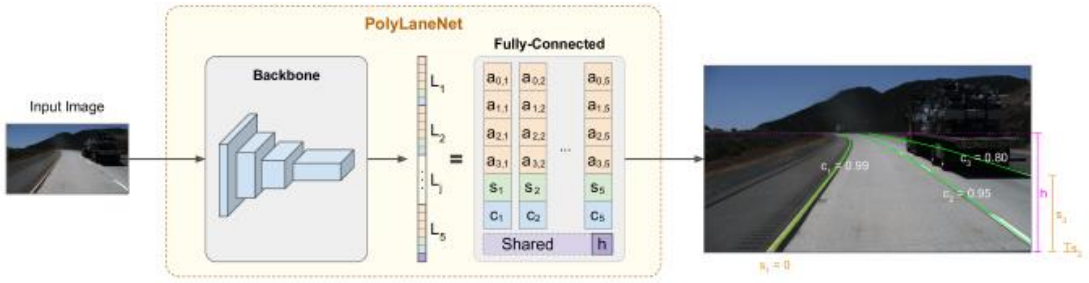


Figure 16 - PolyLaneNet architecture[16]

As seen in other models, the backbone is used to do feature extraction. EfficientNet-b0 was used as the backbone. The returned features are passed into a fully connected layer that has the size of the Mmax + 1. Where Mmax is the maximum number of lanes in the image(according to the annotations). The fully connected layers output polynomial values that represent the position of the lanes and the vertical offset. These can be used to draw the road lanes.

2.3.3 LSTR

Like PolyLaneNet, LSTR is also a deep learning approach but with a transformer-based architecture. The model was released in 2020 and is inspired by the work of DETR. LSTR leverages the self-attention module to learn a broader context of the image, while still being lightweight and fast.[5]

Architecture

When looking at the architecture in figure 17 it immediately becomes clear that LSTR was inspired by the working of DETR.

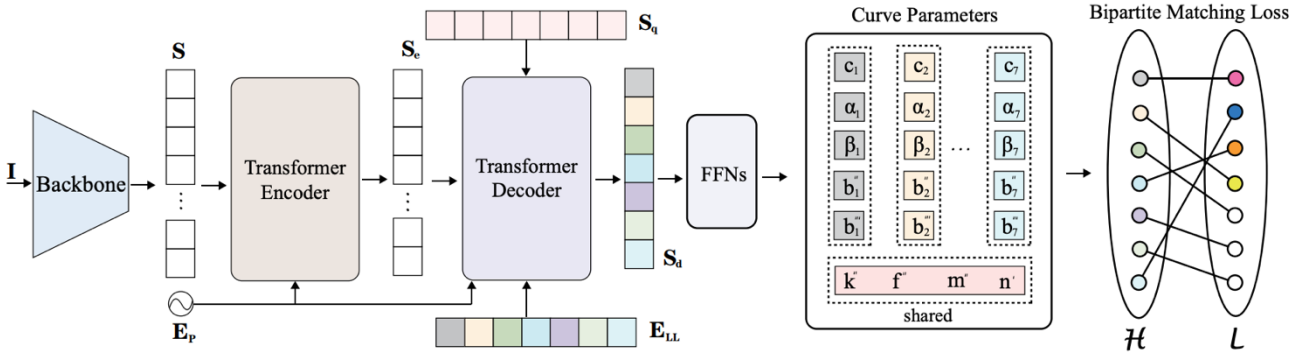


Figure 17 - LSTR architecture[16]

These features are passed into the transformer. At the end of the architecture bipartite matching is used to determine the loss.

Backbone

A backbone is used to do feature extraction and get positional embeddings. LSTR uses ResNet18 as the backbone. The extracted features are not multi-scale, meaning these features only come from the low-resolution layer. Before being fed into the encoder the features are flattened into a spatial sequence with size height*width*channel.

Encoder

The encoder consists of two layers. The structure of the encoder block is the same as seen in the original Attention is all you need paper.[1] The encoder extracts the spatial representations. This is done by using the attention module. The attention module captures the global context of the image as well as slender structures representing the lanes. After the attention module, a normalization layer and feed-forward NN are used before feeding it into the second encoder layer. The process is repeated in the second layer before continuing to the decoder.

Decoder

The decoder also contains two layers, but unlike the encoder, the layers are not equal. This is done to train separate attention modules. Each decoder layer inserts the other layer's attention module and uses the output from the encoder as their input. Doing this enables performing attention to the features that contain spatial information associated with the most related feature elements. Besides the input from the encoder the decoder also receives a learnable lane embedding as input, this lane embedding is empty at first. The lane embedding is used to learn global lane information, similar to the object queries seen in DETR. Next, the lane embedding is passed on to the Feed Forward NN block.

FNN's

In the first step, a set of lane curves get predicted and classified as background or lane with the use of a softmax layer. The background class is equal to the Ø class as seen in DETR.

At the same time, the input sequence gets converted to N*4 size. N represents the max number of lanes. Dimension 4 represents four groups of lane parameters. These parameters get averaged into one dimension resulting in the four shared parameters, as seen in the architecture in figure 17.

In the end, the predicted lane curves (with their corresponding classes) and shared parameters get outputted. As mentioned above bipartite matching is used as a loss function.

2.3.4 Comparison

The comparison table will not consider the Hough transform method. This method works relatively well for experimenting and proof of concept applications but does not compare to the level of SOTA models and thus has not been considered in benchmarks.

Results are based on the TuSimple test dataset

	Accuracy	False positives	False negatives	FPS
PolyLaneNet	93.36%	9.42%	9.33%	115
LSTR	96.18%	2.91%	3.38%	420

[5]

PolyLaneNet can perform at a very high and accurate level while still being fast enough to work in real-time applications. Looking at the PolyLaneNet scores makes the results of LSTR even more astonishing. LSTR is accurate and very fast, it outscores PolyLaneNet on every metric. The FPS score is very noticeable since as we discussed before; the attention module tends to slow down the transformer-based models.

Of course, these two models are not the only SOTA models around for lane detection. In the last couple of years, a lot of advancements have been made in this field, with a lot of them making use of spatial or positional embeddings.

2.3.5 Conclusion

The results and advancements of LSTR are very impressive and will outperform the majority of models like PolyLaneNet. When looking at the benchmark scores on the TuSimple dataset, there is not the same bulk of transformer-based models as seen in object detection, in the top 10, there are three models that use a transformer-based architecture.[19]

At this moment it does not look like transformer-based models are ready to take over lane detection. However, the rise of transformers should not be ignored, techniques like attention and positional embedding are very useful to find lane markings and could be featured in a lot of new models.

2.4 Semantic Segmentation

Self-driving cars use segmentation to give the driving system information about its surroundings. It helps in understanding the difference between drivable areas and objects. This section will take a deeper dive into two models. Fully Convolutional Networks[20] and Segformer[6]. Additionally, an explanation of how DETR[4] can be used for segmentation is provided.

2.4.1 Fully Convolutional Networks

Fully Convolutional Networks or FCN in short, were introduced in 2014. Despite their age, the FCN architecture and performance are still relevant today and have been improved over time. The goal of FCN is to show that with a simple convolutional architecture it is possible to reach a high score with a fast inference time. FCN works end-to-end and makes use of pixel-wise predictions.[20]

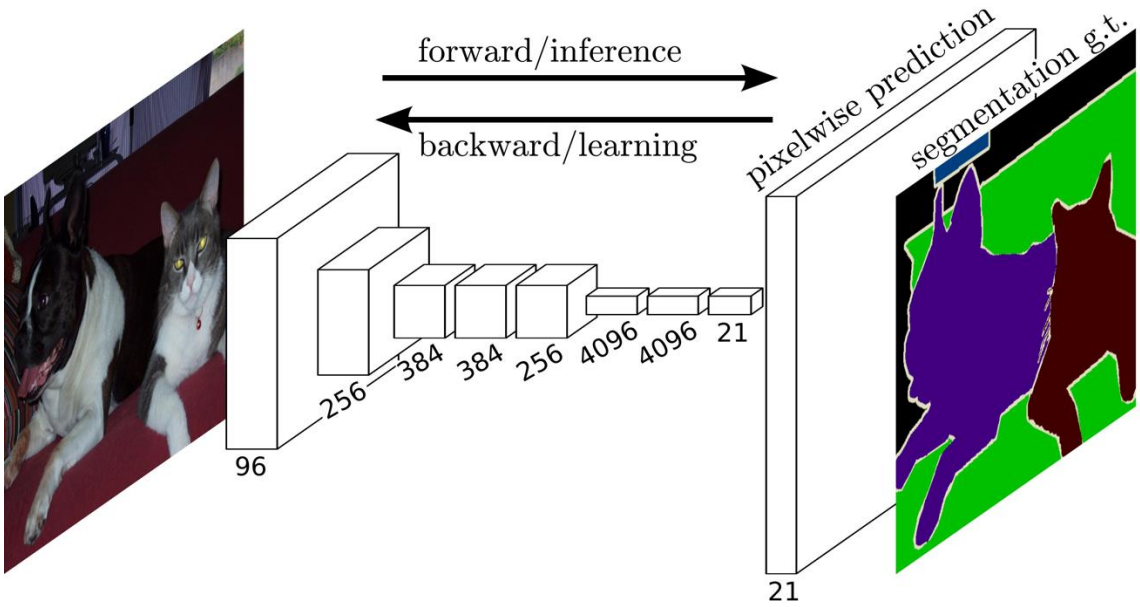


Figure 18 - FCN architecture[20]

Upsampling

Theoretically, it is possible to use a normal CNN model for segmentation tasks. The limitation is that the output image will be downsized drastically compared to the input. This happens because of the use of convolutions and pooling, which are critical for the working of CNN models. FCN tackles this issue by adding an upsampling layer, this gives back pixel-wise predictions.

The loss function is the same that is used for normal image classification, but it is performed on each pixel separately. The loss of all pixels gets added up.

Shaping the output

At this point, the output of the model is very coarse. The model knows in what area the objects are but because the outputs just got upsampled there are no clear boundaries. The bigger the upsampling layer the rougher the output becomes.

This issue gets solved by not directly upsampling by 32x, but rather slowly building up the upsampling allowing for more refined results.

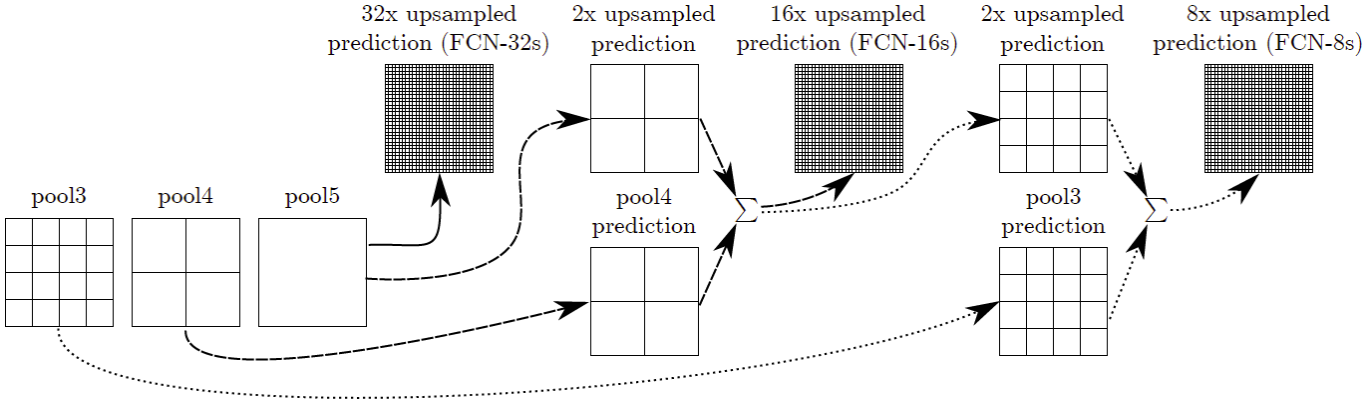


Figure 19 - combining outputs[20]

In figure 19 you can see the upsampling architecture for three different upsampling sizes. 32x, 16x and 8x. Figure 20 represents what the output of an image looks like when upsampled by these different sizes.

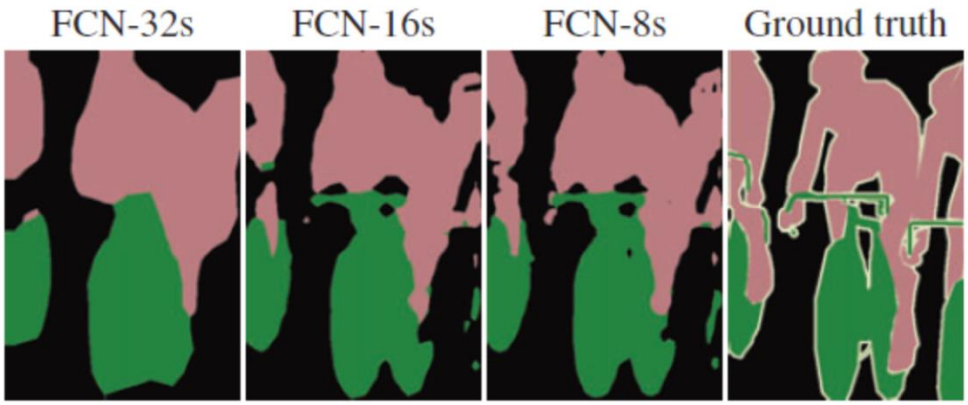


Figure 20 - Different upsampling sizes[20]

It is very clear that combining the outputs to an 8x upsampling layer, gives a more accurate result compared to a quick upsample by 32x.

2.4.2 Segformer

Segformer was introduced in 2021 by NVIDIA. It makes use of a transformer-based architecture, but without the need for positional encoding or complex decoders. One of the main advantages of Segformer is its ability to easily scale. This makes it possible to create a series of models, balancing speed and accuracy. Where B0 is the least accurate but most efficient model and B5 is the most accurate but least efficient. Having this series of models assures there is a Segformer model for every use case.[6]

Architecture

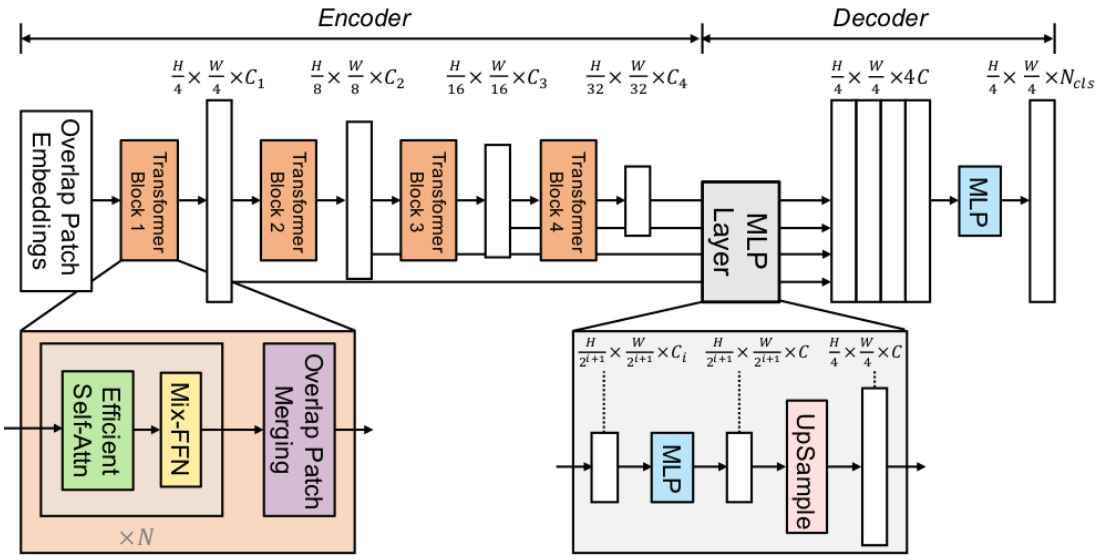


Figure 21 - Segformer architecture[6]

Segformer uses a transformer-based architecture with an encoder and decoder. However, these are not identical to the original transformer architecture. More particularly the decoder has a very different structure and uses an MLP layer to aggregate information from different layers.

Encoder

Inspired by the working of ViT[21], the image gets divided into patches. ViT uses a size of 16x16, segformer uses 4x4 since this happened to give better results for dense predictions.

As can be seen in figure 21, the segformer makes use of an Efficient Self-Attention block. The working of this block is similar to regular Multi-head attention but makes use of sequence reduction to work quicker and more efficiently.

To gain positional information, ViT uses positional encoding. The downside of this is that the positional encoding has a fixed length. Segformer tackles this issue with the use of the Mix-FFN block. In each FFN a 3x3 convolutional layer is added together with an MLP layer. Using this convolutional layer proves to be sufficient in providing positional information and works faster than positional encoding. Segformer calls this data-driven positional encoding. At the end of the encoder, overlap patch merging is used to reduce the size of the feature map.

Decoder

The decoder has a very simple and lightweight architecture that only consists of MLP layers. First, the MLP layer will unify the channel dimension of all input features from the encoder. Before concatenating the features together they are first upsampled to ¼ of their size. Next, a new MLP layer is used to fuse the concatenated layers. The fourth and final step is to use another MLP layer to predict the segmentation mask on the fused features. The number of channels in the final resolution is equal to the number of categories that can be predicted.

2.4.3 DETR

One of the advantages of DETR is the ability to be used for segmentation tasks without the need for retraining. This can be done by applying a mask to the decoder.

Mask working

After the decoder block for regular object detection, an additional mask head block is added. This block creates parallel masks for each detected object. To create the masks the output of the decoder block gets used together with the computed multi-head attention scores of each object embedding. This creates M (number of multi-head attention layers) number of attention heatmaps for each object. The final prediction can be achieved by using an argmax function on each pixel to assign the category with the highest value. This results in a final mask without overlaps or the need for heuristic functions.

2.4.4 Comparison

Since DETR is designed for panoptic segmentation and has not been tested on the cityscape dataset, it will not be taken into account in the comparison table.

	ADE20K		**Cityscapes**	
	mIoU	FPS	mIoU	FPS
Fully Convolutional Networks	19.7%	64.4	61.5%	14.2
Segformer B0	37.4%	50.5	71.9%	47.6
Segformer B5	51.8%	9.8	84.0%	2.5

[6]

Segformer B0 has four options regarding the image size. Images can be scaled to 1024, 768, 640, or 512 pixels on the short side. This gives another balance between accuracy and speed. In the table above the fastest, and thus least accurate version has been used. Either way, Segformer B0 still outperforms FCN on both ADE20K and cityscapes. It does perform slower on ADE20K but has almost double the accuracy.

Cityscapes is a very relevant dataset for this thesis since it has been trained and evaluated on an urban city environment, as often seen in self-driving scenarios. Segformer scores very well on this dataset, both on accuracy and speed. The least accurate model performs (B0) scores similar to other SOTA models, while the most advanced model (B5) is in the top 5 of best performing models on the cityscapes validation dataset and number 22 on the ADE20K validation dataset. Unfortunately, Segformer B5 is not fast enough to be used for real-time applications like self-driving cars.

2.4.5 Conclusion

When it comes down to segmentation, this is probably the computer vision task where transformers hit their biggest home run. Not only does Segformer comfortably beat Fully Convolutional Networks, but we also see a strong trend when diving into the benchmarks for the cityscapes and ADE dataset.

For cityscapes, the top 5 consists of 4 transformer-based techniques, and even more can be found further down the line. When looking at the ADE benchmarks an even stronger trend can be found, only one model in the top 10 does not make use of a transformer-inspired architecture.[22], [23]

When taking all these results into account it becomes clear that transformers are ready, and already are taking over segmentation tasks.

2.5 Simulator

As mentioned in the introduction chapter, the use of a good driving simulator can be crucial to the success of self-driving car applications. This section will take a closer look at CARLA, Deepdrive, and AirSim. At this point in the thesis, it has already become clear that CARLA has been used. Either way, it is important to understand the possible options and why CARLA has been chosen.

2.5.1 CARLA

CARLA is an open-source and highly customizable simulator created to develop self-driving car applications in urban areas.

To communicate with the simulator CARLA uses a server-client system. The server is the CARLA simulator that runs and collects data, and the client can use the CARLA API to influence the environment using python scripts. Figure 22 shows that data gets read in real-time, predictions get made, and sent back to the CARLA server to show them on screen.[24]

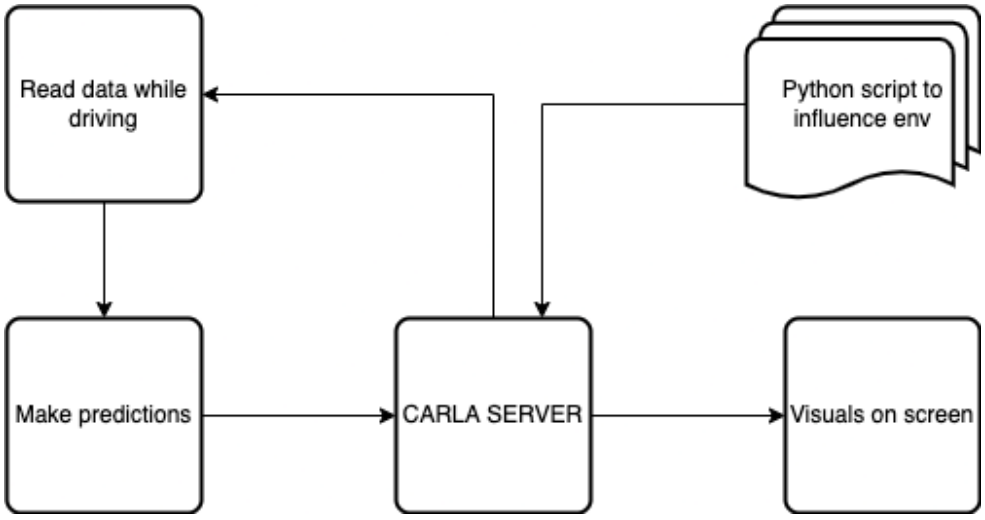


Figure 22 - Working of CARLA

2.5.2 Deepdrive

Deepdrive and CARLA share a lot of common features. Both of them are open-source simulators that use the unreal engine and use a PythonAPI to communicate with the simulator (see figure 22). However, unlike CARLA, Deepdrive does not require 20GB of disk space, but only 10GB.[25]

2.5.3 Airsim

Airsim is a simulator developed by Microsoft. It is a bit different compared to the previously mentioned simulators. Airsim is not solely focused on self-driving cars but also on drones for flying simulations. Like CARLA and Deepdrive it is built on the Unreal engine. One of the strong points of Airsim is that it does not require a GPU to work.[26]

2.5.4 Comparison

It becomes clear that all simulators are very similar both in usage and inner working. Let's compare to see who comes out on top.

	CARLA	Deepdrive	Airsim
Engine	Unreal engine	Unreal engine	Unreal engine
Supported OS	Windows, Linux	Windows, Linux	Windows, Linux, macOS
Requires GPU	Yes	Yes	No
Quick install possible	Yes	Yes	No, requires building
Number of sensors	6	8	6
Quality of documentation	High	Low, Only Github page	High

[24]–[26]

Following this table, it becomes clear why CARLA is the preferred option. It is very easy to install and has high-quality documentation to guide you through the inner workings. If having a GPU is an issue, then Airsim is a very nice and high-quality alternative.

3 Technical research

During the first chapter of this thesis the introduction has been given, the research question has been stated and the goal has been set. In the second chapter theoretical research has been done and different models were compared for each computer vision task. During this chapter, a deeper dive will be taken into technical research that has been performed during the research project earlier this semester.

3.1 General working

Before taking a deeper dive into each computer vision task separately, it is good to understand how the technical demo was built and how it exactly works as a whole.

The backbone of the technical demo is the CARLA simulator. This has been used to simulate the self-driving car and gather data from the RGB sensor on the car. Since the technologies had been implemented for demo purposes, the driving console of the car was operated by CARLA and has not been impacted by the computer vision tasks. To gain a better understanding of how the communication with CARLA works it is recommended to read section 2.5.

When starting up the demo, the desired computer vision models could be passed on as parameters. Object detection and lane detection models could be combined. Once the parameters were passed on, the simulator runs. Predictions are made on each frame. Therefore, the speed of inference was so important.

Figure 23 - LSTR and DETR

Figure 23 shows an example of YOLOv5 and LSTR operating at the same time. This is a realistic scenario since you'll need both to operate self-driving.

3.2 Object detection

For object detection, there has been made use of YOLOv5 and DETR. Because of the previously mentioned reasons in chapter 2, YOLOv5 has not been explained during the previous chapter but works in a very similar way to YOLOv4. It was introduced closely after YOLOv4 but does not have a public paper available. However, in a blog released by the developers, they do recommend YOLOv5 in a production environment because of its improved speed and different model sizes.[27]

3.2.1 The dataset

Before a model could be trained there was a need for data. Unfortunately, there are not a lot of CARLA datasets available, the available datasets were often too general. For example, a single class “traffic light” for all three colors combined. This meant for once it was time to reinvent the wheel. Luckily it was very easy to gather data in CARLA. This could easily be done by simulating a car and saving every frame to disk.

Once the data had been gathered, it needed to be labeled. As mentioned earlier this has been done with roboflow, labeling ten different classes. The dataset turned out to have a size of +- 1800 images and 3240 objects. The used augmentations and processing depended on the used model.

Roboflow also gave interesting insights into the data, these insights helped to gain a better understanding of the data. For example, the annotation heatmap. This shows early on before training even started what the interesting areas are in the images. In theory, these heatmaps should look very similar to the object queries that DETR uses internally.

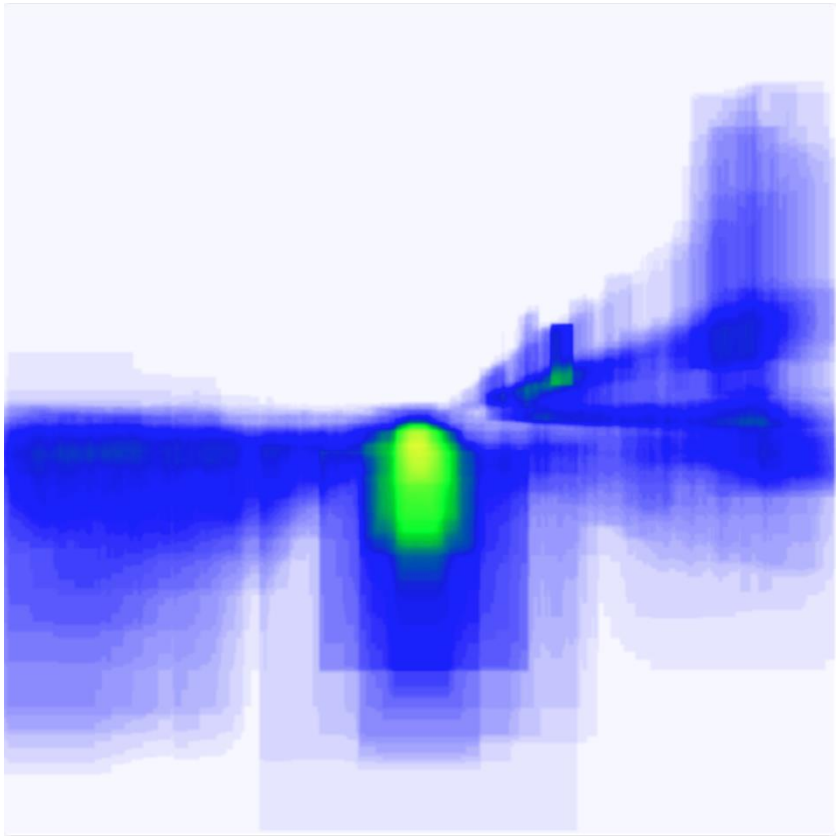


Figure 24 - Annotations heatmap

3.2.2 YOLOv5

Using the gathered data, a YOLOv5 model could be trained. More specifically the small version has been used, the small model is less accurate but has a very fast inference time, around 6.4ms. Training a

Yolo model was very straightforward and did not require a lot of work. The only important step is to change the number of classes in the YAML configuration file. Default this is set to 80, in this use case it was changed to 10. The YOLOv5 model has been trained on 100 epochs.

Figure 25 - YOLOv5 visual results

3.2.3 DETR

Secondly, DETR was trained on the same dataset as YOLOv5. Setting up training for the DETR model is more complex but gives a lot of freedom in customization and optimization. Parameters like object queries, the learning rate of the model and backbone, and weight decay could all be fine-tuned separately.

Training the model was computationally expensive making it difficult to do a lot of experimenting and fine-tuning. In the end, the model has been trained on 150 epochs.

Figure 26 - DETR visual results

3.2.4 Scores

Both models had been trained on the same dataset and thus been evaluated on the same test set as well. During the evaluation mAP 0.5, and mAP 0.5:0.95 were used as a metric. mAP 0.5 means the average precision of the predictions with at least a 50% overlap of the bounding box with the ground truth.

mAP 0.5:0.95 is the average mAP over a series of different overlap thresholds between 50% and 95%.

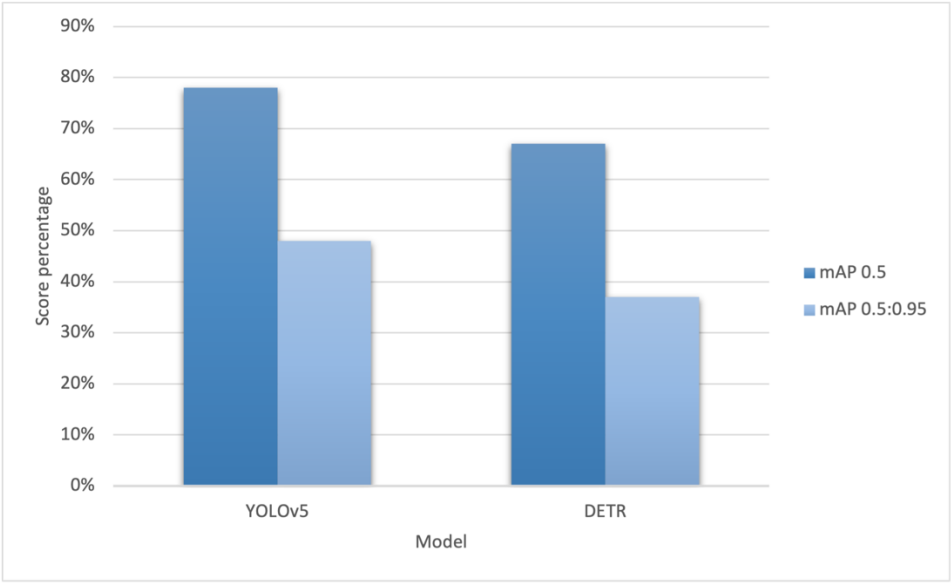


Figure 27 - Object detection scores

The results of YOLOv5 were satisfactory. It detected most objects accurately while drawing straightforward bounding boxes. YOLOv5 achieved a good score of 79% on mAP_0.5 and 49% on mAP_0.5:0.95.

Overall DETR scored decently. It detected most objects but failed to draw confident bounding boxes, this often resulted in messy and unclear results. DETR reached a score of 67% mAP_0.5 and 37% on mAP_0.5:0.95.

Overall this technical part concludes that YOLOv5 worked slightly better than DETR. The Yolo model is more developed and can better handle the small dataset with limited computational resources.

3.3 Lane detection

During the technical research, I made use of Hough transform and LSTR. Hough transform has been used to create a visual-only proof of concept. LSTR has been used to test a new transformer-based technique. Both of these techniques have been explained in chapter 2.

3.3.1 Hough transform

Hough transform was a good first step into lane detection. This was a visual-only approach meaning it only used a couple of detected points in the image to draw lines instead of calculating the path itself. Unfortunately, this meant it was difficult to grasp in numbers how good or bad this model performed. Either way, it was still possible to interpret the visual results.

Understanding the hough transform results was best done by using it in a couple of different scenarios.

Scenario 1 – Straight line with lane markings

This was probably the most common scenario. A classic road in a town or city that has lane markings.

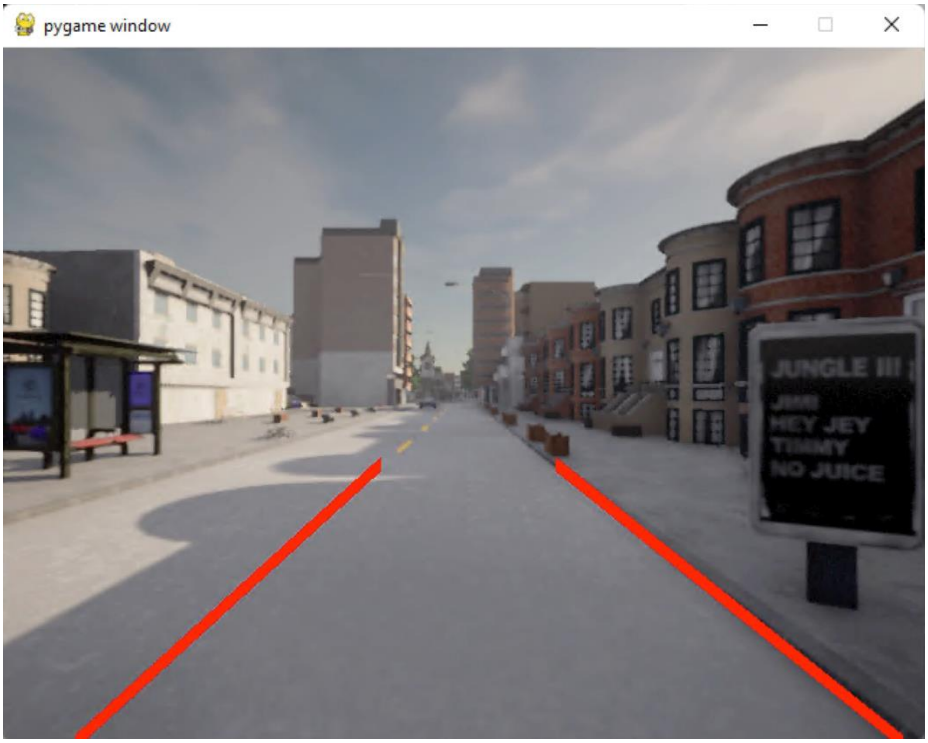


Figure 28 - Scenario 1 HT

In this scenario (figure 28), Hough transform worked pretty well. Not surprisingly because of the sidewalk and stripped line it can detect the lane. Also, the lack of conflicting light conditions or other vehicles helps out the model a lot.

Scenario 2 - stopped at a traffic light

The second scenario was when the car arrived at the traffic light behind another car.

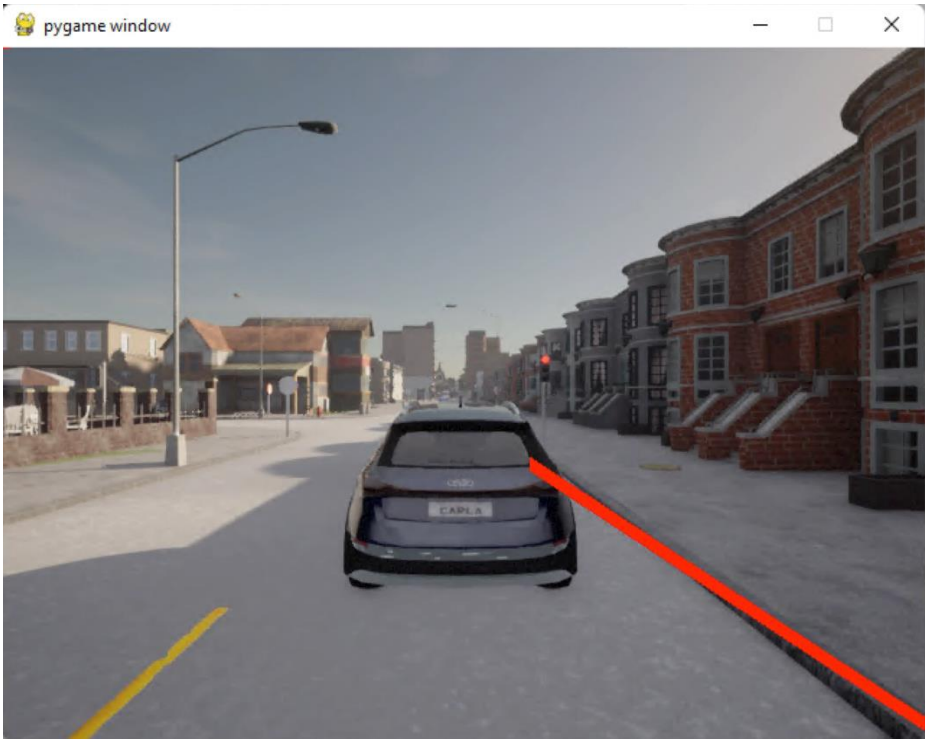


Figure 29 - Scenario 2 HT

During this scenario, the flaws of Hough transform started to show up. In figure 29 can be seen that only a small line marking on the left and a car in front results in an inconsistent lane marking. The sidewalk on the right side made it possible to detect that side of the lane, but it still became clear the model struggles.

Scenario 3 – No lane markings

A realistic scenario within urban areas was the total lack of lane markings.

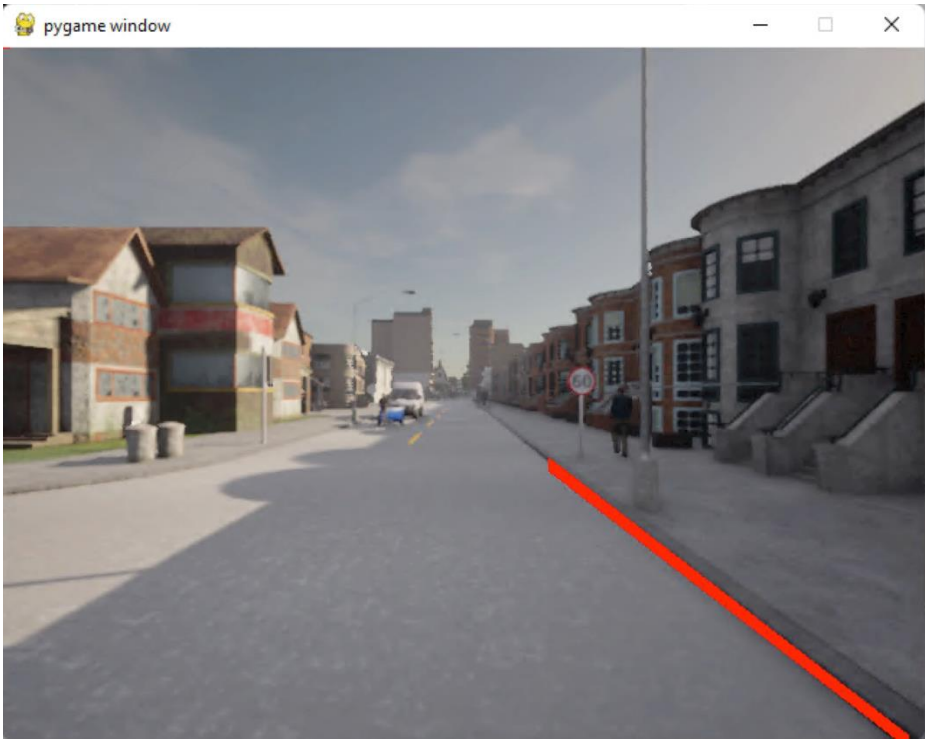


Figure 30 - Scenario 3 HT

As expected this was a big issue for Hough transform. As explained in chapter 2, Hough transform relies on edge detection to find the lanes. Figure 30 shows that without lane markings, the model did not have information to draw the left lane marking.

Scenario 4 – Weather

When driving around town a lot of different weather conditions can be encountered. In this scenario, the road was affected by shadows.

Figure 31 - Scenario 4 HT

This was probably the worst-performing scenario. Figure 31 shows that shadows across the road had a very big impact on the edge detection and lane marking. Even the right lane marking that was pretty accurate in previous scenarios now failed to stick to the sidewalk.

Scenario 5 – Turning

The last scenario explored was the behavior of the model when turning around a corner.

Figure 32 - Scenario 5 HT

In this scenario another downside of the Hough transform model became clear. As mentioned earlier, the Hough transform model does not calculate the lane coordinates but rather tries to detect edge points. As can be seen in figure 32 this resulted in straight lines even when the road did not go in a straight line. For the left lane marking it failed to detect enough edge points to draw a lane marking.

Conclusion

During the scenarios, it became clear that the Hough transform could only be used as proof of concept. It performed decently in perfect conditions with clear lane markings on the road. When it came down to more tricky scenarios or different weather conditions the hough transform model came up short and was only able to detect the right lane marking or misplace both lane markings.

3.3.2 LSTR

Unlike Hough transform, LSTR tried to predict the coordinate points of the lane markings. Because of this, LSTR should have been more robust and been able to detect when turning around corners. During this section, the previous scenarios were used again to see how LSTR handled these.

Scenario 1 – Straight line with lane markings

This was probably the most common scenario. A classic road in a town or city that has lane markings.

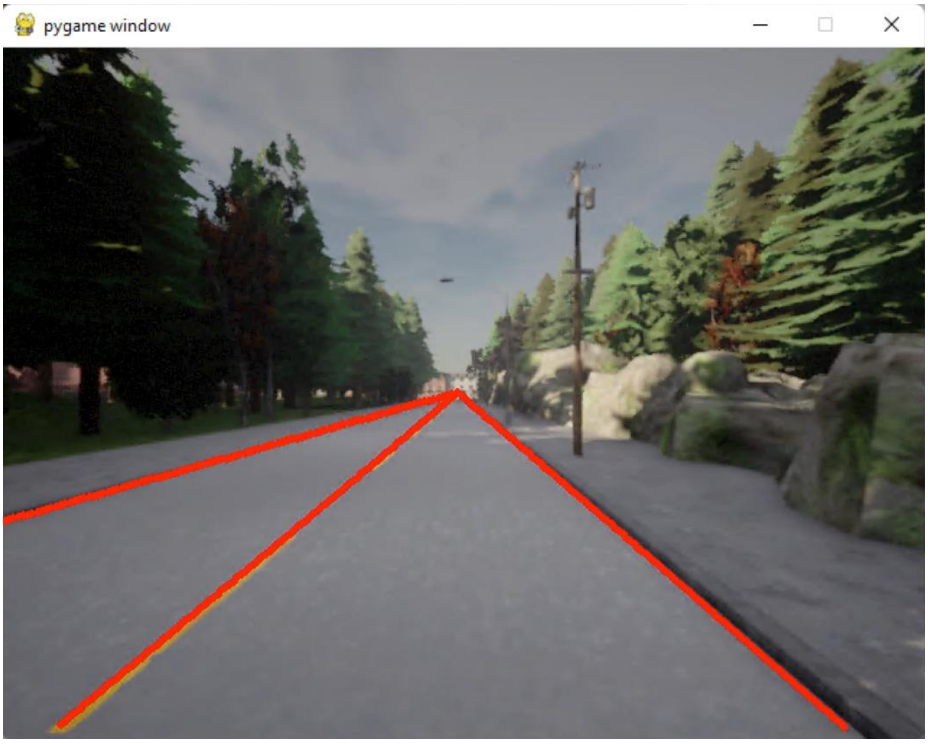


Figure 33 - Scenario 1 LSTR

Earlier it became clear driving in a straight line of a normal road was the best performing scenario for Hough transform. This is no different for LSTR, it performs very solid and was able to mark both lanes until the end of the road.

Scenario 2 – Stopped at a traffic light

The second scenario was when the car arrived at the traffic light behind another car.

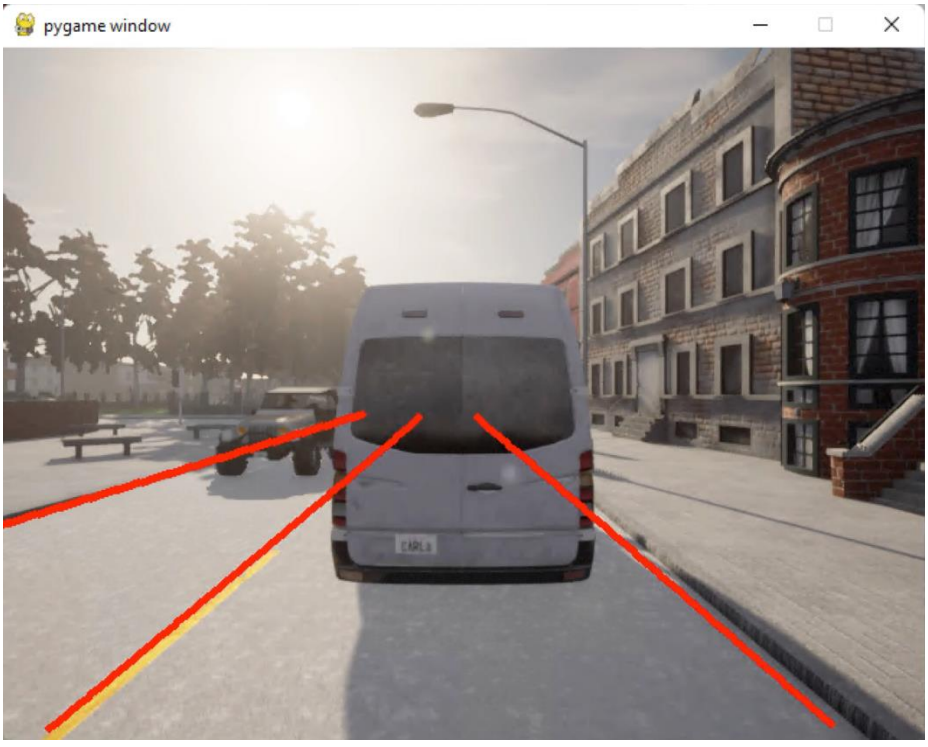


Figure 34 - Scenario 2 LSTR

Unlike Hough transform, LSTR was not impacted by other cars on the road. It maintained its planned markings even when other cars were in front or passed by.

Scenario 3 – No lane markings

A realistic scenario within urban areas was the total lack of lane markings.

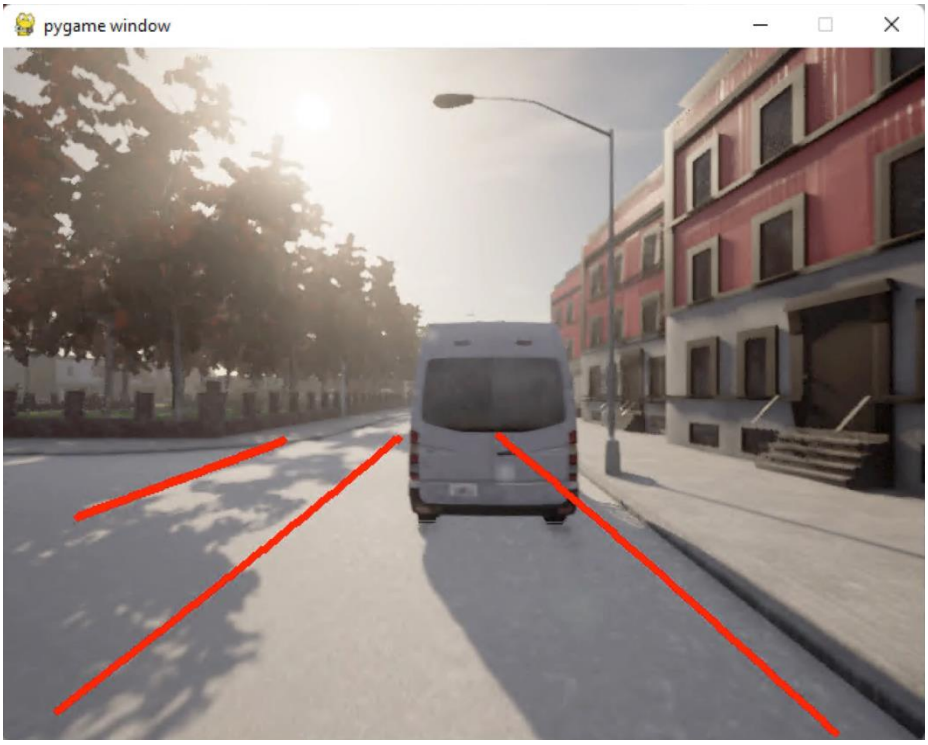


Figure 35 - Scenario 3 LSTR

It was very impressive how LSTR was able to keep a sense of the road without any lane markings on the ground. This showed very well the impact of using a deep learning transformer-based approach compared to a mathematical approach.

Scenario 4 – Weather

When driving around town a lot of different weather conditions can be encountered. In this scenario, the road was affected by shadows.

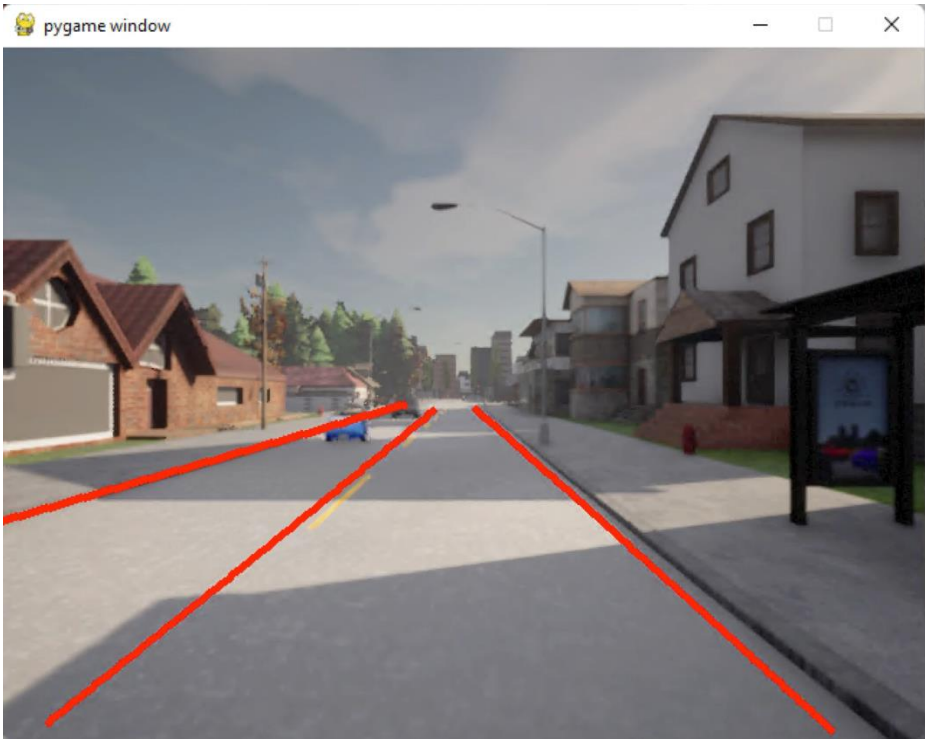


Figure 36 - Scenario 4 LSTR

Again the superiority of LSTR became clear, even when obvious shadows came across the road. The model was still able to mark the lanes accurately.

Scenario 5 – Turning

The last scenario explored was the behavior of the model when turning around a corner.

Figure 37 - Scenario 5 LSTR

Scenario 5 was probably the most difficult since it relied on detecting lane markings that were not in a straight line. Unlike Hough transform, LSTR did a very decent job and was able to predict the rough turning point with curved detection lines. However, predicting where to turn would end was not perfect and could have been improved on.

Conclusion

When facing LSTR with the same scenarios that Hough transform faced, it became clear that LSTR was a more robust and accurate approach. LSTR was also able to detect both driving lanes instead of just the right lane. These results should not have come as a surprise but it was still impressive to see the difference in accuracy, robustness, and flexibility.

3.4 Segmentation

This section differs from the two previous ones. Implementing the segmentation model was less about comparing technologies and more about showing the capabilities of big pre-trained transformer-based models and how they could be very easily implemented for different use cases. While for object detection and lane detection the models have been fine-tuned or trained from scratch, this was not done for segmentation.

3.4.1 SegFormer

The architecture and inner workings of SegFormer have already been explained in the previous chapter. One of the main reasons SegFormer was so interesting to use was because of all the different and easily accessible model options. On the huggingface page of NVIDIA, a total of 15 pre-trained models could be found, finetuned on 2 different datasets, 6 performance levels, and multiple screen resolutions.

SegFormer fitted the self-driving car use case perfectly since it has not only been trained on the ADE dataset but also on cityscapes. Because of this, the implementation was very easy, the full SegFromer class used in the technical demo is only 50 lines of code.

The downside of not having fine-tuned or re-trained the model is the difference in labels. SegFormer uses the labels proposed by cityscapes and thus does not use the same labels that had been used for the object detection models.

Figure 38 - SegFormer B2 visuals

In the visuals, it quickly became clear that the model tends to work pretty well out of the box, showing the strong advantages of using pre-trained transformer-based models.

3.5 Difficulties

While some satisfactory results were achieved, this did not come without difficulties and challenges to conquer.

3.5.1 Training DETR

While using the pre-trained DETR model was very quick and easy, finetuning DETR was less straightforward. At the time the technical research had been performed there was no straightforward documentation provided on how the model could be fine-tuned on a custom dataset. There was however documentation on how to train from scratch, sadly the available dataset was too small.

Luckily there were some Github issues open on this topic that discussed possibilities of fine-tuning. This resulted in the idea of loading in the model, making the classification head trainable, and restarting training. After doing some extra research and with the help of some tutorials it eventually worked out to train only the classification head from the pre-trained model with a different number of labels.

3.5.2 DETR scores

At first, the DETR scores were disappointing, only scoring 8% mAP 0.5, and 4.2% mAP 0.5:0.95. The reason was quite clear, a lack of data. At the time the dataset consisted of 700 images. This turned out to be nowhere near enough to properly fine-tune the classification head. After putting a lot of effort into annotating the new images. The size of the dataset was more than doubled to 1600 images, the score was now around 8 times better. On the COCO dataset, Yolov4 outperformed DETR by 3.8%. In this use case, the difference was still 12% in favor of YoloV4. This showed that DETR had even more potential to close the gap if the dataset had been bigger.

3.5.3 Training LSTR

Since gathering data for lane detection was quite easy, the model has been trained from scratch. Of course, this required quite some computational power and at the time only Linux was supported for training. To solve this all training steps were moved to azure machine learning. To do this the original training script had to be modified to work on Azure instead of the local machine. The setup was tricky but worked out perfectly. In the end, the LSTR model trained for around 20 hours.

4 Reflection

During this section, there will be reflected on the achieved results and research performed during the thesis. The reflection is not only based on personal thoughts but also interviews with experts within the AI field.

Experts:

- Sean Nachtrab. Machine Learning Engineer at OTIV.
- Jurriaan Biesheuvel. Data scientist at RIVM.

4.1 Strengths and weaknesses

4.1.1 Strengths

According to Sean Nachtrab, the strong point of this research is the clear understanding of new cutting-edge technologies that come from different fields before being investigated over multiple computer vision domains. This point was also brought up by Jurriaan Biesheuvel. A second strong point is the use of a self-created dataset, as this shows knowledge about how to look at data.

Object detection

The research around object detection shows the current problems that the first-generation transformer-based models encounter while also going into the potential of the newer generation models. Overall, the results achieved during the technical research were decent and expected.

Lane detection

The lane detection section introduced a new technique that is not widely known or used but does work surprisingly well in all 5 different scenarios compared to the Hough transform technique. While Hough transform certainly did not make a lasting impact, it did show that it can be used when setting up a quick proof of concept application.

Segmentation

Segformer showed the possibilities of using transformer-based models for segmentation. The use of the pre-trained models showed great results out of the box.

4.1.2 Weaknesses

Object detection

Whilst the result for YOLOv5 and DETR were decent, it would have been good to have had even more data. It does look like the full potential of DETR has not been unlocked because of the lack of data. This is of course a fine line since the technical research needed to be accomplished in a small-time window.

Lane detection

The weakness of the technical lane detection section comes down to the baseline model. Hough transform has been used because this is a very straightforward approach that works well as proof of concept. Unfortunately, this also meant that it was not possible to compare the accuracy of the Hough transform and LSTR model with numbers, but only by comparing them in certain scenarios.

Sean Nachtrab agrees with this point and states that Hough transform is a proof of concept and not an established technique used within lane detection.

A weak point brought up by Jurriaan Biesheuvel was that whilst the scenarios for both HT and LSTR are the same, the exact images are not. He stated that this does not show a 100% accurate comparison and some deviation should be taken into account. This is a fair point and should have been considered while doing the technical research.

Segmentation

While the segmentation section is not supposed to be comparing models, it would have painted a better picture if an extra non-transformer-based model was used to compare the FPS scores and check for visual differences.

4.2 Implementation possibilities and their value

The opinions of both interviewees differ quite a bit on this topic. This shows that this technology is still new and has not established a trusted reputation. Either way, it is important to bring forward both opinions and see how they differ from each other.

According to Jurriaan Biesheuvel, there are currently not a lot of implementation possibilities in a production environment. Although he is not convinced of the value it can bring right now, he does believe that this thesis can bring more value to R&D departments to improve these technologies in the coming years

On the other side, Sean Nachtrab brought up that there certainly are possibilities to use a transformer-based model for autonomous driving. OTIV is currently implementing transformer-based models, but also bigger companies like Tesla have adapted their architecture to use transformers-based models for object detection. This has been reported during their latest Tesla AI Day.[28] Mr. Nachtrab also acknowledged that the ability of transformers to reason over time is a big part of why they are becoming so valuable.

However, there are also some cons to the use of transformer-based models. The main pain points that have been brought up in this thesis and have been confirmed by Sean Nachtrab are the need for a lot of data, heavy computational cost, and finding established models.

Another interesting point that has been brought up was the lack of knowledgeable people within the field. This is a valid point since the technology is still relatively new, especially within the computer vision field. Although it is very difficult to bring the levels of knowledge between CNN and Transformers into perspective, below is a table with the 5 most popular AI courses on Coursera. The results show that only 1 out of the 5 courses talk about transformers. Whilst this is not fully foul proof, it shows that transformers are currently not seen as default AI knowledge.

Course	Teacher	Covers transformers?
Machine Learning	Stanford University	Yes
Deep Learning Specialization	DeepLearning.AI	No
IBM Machine Learning Professional Certificate	IBM	No
Machine Learning Specialization	University of Washington	No
Machine Learning: Theory and Hands-on Practice with Python Specialization	University of Colorado Boulder	No

[29][30][31][32][33]

4.3 Alternatives

A possible alternative brought up by Sean Nachtrab and Jurriaan Biesheuvel was to change the used prebuilt/pre-trained models with custom models. These custom models would realistically be mainly inspired by the existing models but could be altered to fit specific use cases better.

More specifically it could be possible to add extra augmentation steps to the pipeline. These augmentation steps could be inspired by all the possibilities seen in YOLO models and should not be kept to object detection alone. Other possibilities include changing the backbone, adding more input modalities, or playing around with different kinds of attention models.

4.4 Social/ Economical/Socio economical value

Both Sean Nachtrab and Jurriaan Biesheuvel support that this research is bringing new knowledge to the autonomous vehicle field and can help the R&D field in striving to achieve new things that have not been possible before. This research can also help IT professionals without transformers knowledge to gain a better understanding of the different possibilities AI has to offer for certain computer vision tasks.

Transformer-based models are helping companies like Tesla, Wayve, and OTIV to build better autonomous vehicles helping in lowering carbon footprint and general costs within the transportation field.

4.5 Suggestions follow up research

In this section, some advice and tips will be given on what steps could be taken to continue this research.

A nice place to start would be to take a deeper dive into the ConvNets vs Transformers discussion. This discussion is too big and way broader than the scope of this thesis. Still, it gives a lot of nice insights and thoughts that are related to this research. Over recent years multiple papers have been released talking about the effectiveness, robustness, and future of ConvNets and Transformers.

Very interesting research has been done by Meta AI when they released the graph in figure 39 in their paper “A ConvNet for the 2020s”.[34] In this paper, they claim that their new pure CNN model ConvNeXt has the potential to perform equal to or better than transformer-based models on all computer vision tasks. These claims have caused quite some dispute, with researchers claiming that the provided chart in figure 39 is incomplete or incorrect.

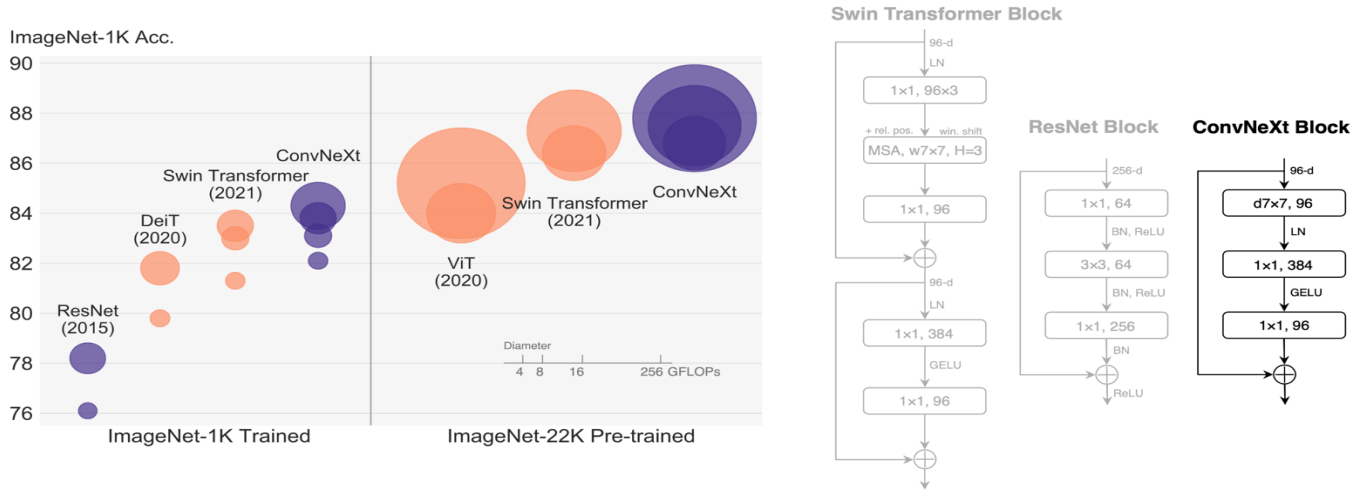


Figure 39 - ConvNets vs Transformers[34]

4.5.1 Object detection

During the research around object detection, a lot of SOTA models have already been discussed. These models scored well in the benchmarks but were all based on the same DETR architecture.

A nice follow-up step might be to take a closer look at object detection models that are inspired by Vision transformers. Vision transformers were introduced a couple of months after DETR and are already showing groundbreaking results in image classification. 3 out of the top 5 models on the ImageNet dataset are based on Vision transformers.[35] It is very interesting to see if these SOTA results can be achieved for object detection tasks in the near future.

4.5.2 Lane detection

A good place to continue this research is to take a closer look at the models at the top of the TuSimple and CuLane benchmarks. Here it becomes clear that while some top models are currently making use of spatial and positional embeddings, there is no clear trend to be seen. Meaning research is still actively ongoing and the best approach still needs to be found.[19] [36]

The most interesting models are probably the ones with an end-to-end architecture, like LSTR they directly output coordinates without the need for extra post-processing. This architecture gives a big advantage for ease of implementation.

4.5.3 Segmentation

During the theoretical research performed in this thesis, it became quite clear that transformer-based models are performing very well on segmentation tasks. While the results are very good it must be said that a lot of transformer-based segmentation models require extra data to perform at the same level as SOTA CNN models.

With Segformer a transformer-based model that does not require the extra data has already been discussed. Either way, it still is very interesting to investigate the working of models like VOLO. They claim to perform at an equal or higher level than CNN-based models without the need for extra training.[37] On the ADE20K and Cityscapes validation dataset, a VOLO-based model is the highest scoring model that did not make use of extra training data.[22], [23]

5 Advice

The advice formulated by this thesis is mainly focused towards the self-driving car industry but could easily be interpreted by the autonomous vehicles industry in general. The research performed in this thesis could help and inspire the industry to start looking away from pure CNN models and move towards newer transformer-based solutions.

As mentioned in the previous chapters, for this thesis a new dataset was created for object detection. The dataset consists of +-1800 hand-annotated images that can easily be used by researchers in the industry to get familiar with the new transformer-based models or try to train other object detection models. The dataset has already been noticed by the industry and was selected to be featured on roboflow universe[38], making it easy for every expert or researcher to get to work.

Since the research has been done on multiple computer vision tasks, the advice will be specified and altered to each task instead of giving only general advice.

5.1 General usability

While in general the usability of transformer-based models should not be doubted, the truth is that some tasks are a better fit than others. Although there has been a lot of progress within AI over the last couple of years the fact remains that AI models generally are good at only one specific task at a time. This understandably results in better models for more popular tasks. When looking at object detection and segmentation a lot of production-ready transformer-based models can be found, not only created by researchers but also maintained by big names within AI such as Meta and NVIDIA. Models like DETR and Segformer are well developed and can be used in real-world applications instead of only being run on benchmark datasets.

However, when looking at a less popular computer vision task like lane detection that only gets used in a very niche industry, it becomes clear that the evolution of new models goes a lot slower. By now most established models within lane detection are CNN based. Although transformer-based models like LSTR show very promising results on benchmark datasets like TuSimple and CuLane, it is currently not known or shown that these models are used within production environments. This does not make transformer-based models unusable for lane detection, but it does certainly show that extra caution is required, and a lot of additional development could be needed.

When looking at the general trend in other computer vision tasks, it looks like only a matter of time until transformer-based models will be used in production environments. But because of the specific and competitive niche, it is only to be seen if these models will be made open source.

5.2 Implementation steps

Since this thesis handles three different computer vision tasks, the steps will be explained for every task separately. This section focuses on the implementation of the performed research and thus not on suggestions for follow-up research. The implementation only focuses on the transformer-based models.

5.2.1 Object detection

Figure 40 shows the general steps that should be taken to implement DETR.

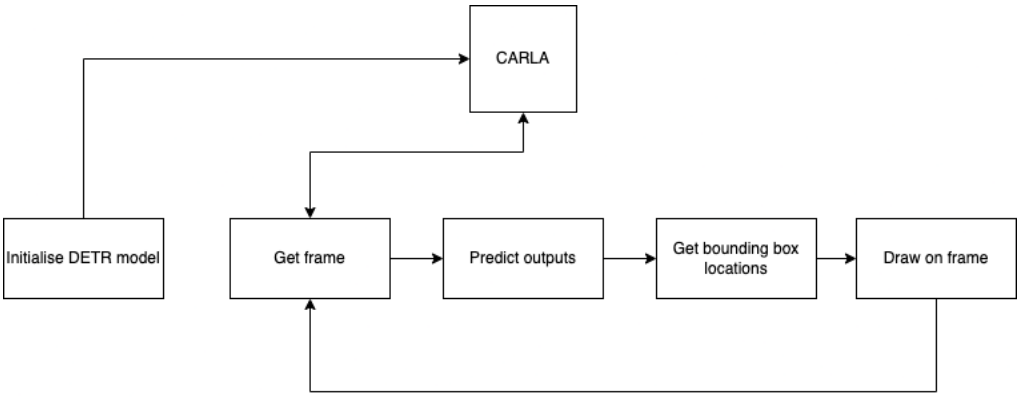


Figure 40 - DETR steps visualized

Step 1 - Training

When using the DETR model for inference, three options are available. It is possible to use the pre-trained version on the COCO dataset provided by Meta AI, use the trained model from this thesis, or use the provided tools to train the model yourself. In this section, we will go for the second approach of using the trained model in this thesis. In section 5.3 and the installation/user guide, a description is provided of how you can gather the necessary data and train the model yourself. In case you have a different model, it is as easy as changing the file name.

Step 2 – Module

To be able to easily use the DETR model it is recommended to build a separate module that can be used from the main code. The DETR module has already been made during the technical research of this thesis. This module consists of a DETR lightning class and code used in the official DETR implementation. This module can be used to predict the location of the bounding boxes and with some small modifications can be adapted to every use case.

Step 3 – Inference

When initializing the DETR module for inference, two parameters should be passed on. The model path and the confidence level. The model path can lead to any saved DETR model. Either the original pre-trained or a custom-trained model. The confidence score determines how quickly a predicted bounding box will be seen as valid.

DETR should be performed on every frame after the necessary image operations. The DETR input needs to be an image in the RGB scale. Predicting an object goes in two steps. First, the prediction of the output values itself before passing these on to get the locations of the bounding boxes in the image. The original input and the location of the bounding boxes can now be used to draw them however you desire.

5.2.2 Lane detection

Figure 41 shows the general steps that should be taken to implement LSTR.

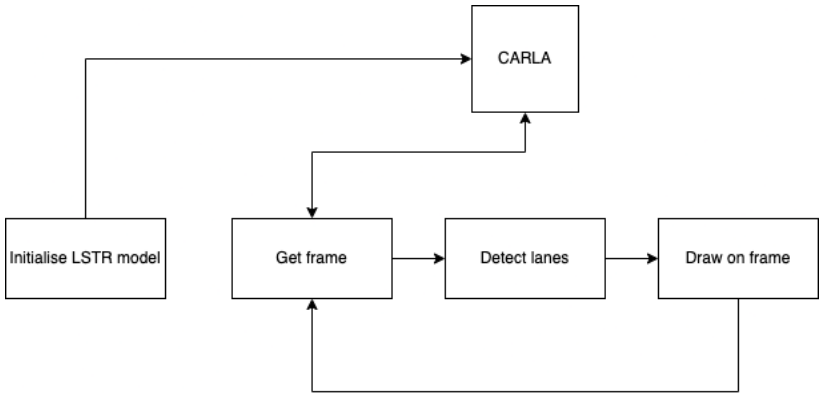


Figure 41 - LSTR steps

Step 1 – Training

A baseline LSTR model trained on the TuSimple dataset is provided on the official GitHub page. Either way by following the instructions in the user guide you can train an LSTR model on the type of environment that you desire. Whilst the DETR model needs to be a checkpoint file the LSTR model requires a pickle file to load the model.

Step 2 – Module

Before inferencing the model, it is recommended to use the LSTR module provided in the technical research. This module is based on code from the original LSTR implementation. The module helps with initializing the right configurations, as well as making lane detection easier.

Step 3 – Inference

Initializing the LSTR module for inference is very simple and does not require any parameters. All. Configurations are handled automatically. Doing the lane detection is very similar to LSTR. First, the image needs to be processed to have the correct RGB format. Then simply run the lane detection function to retrieve the coordinates. These coordinates can then be used to draw the lines.

5.2.3 Segmentation

Figure 42 shows the general steps that should be taken to implement LSTR.

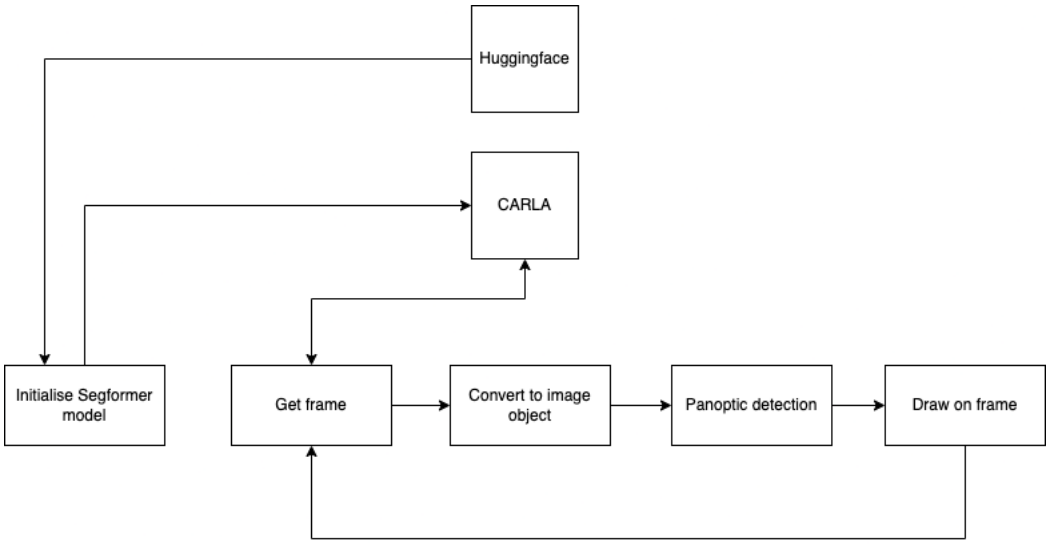


Figure 42 - Segformer steps

Step 1 – Module

The segmentation module created during the technical research has a very straightforward built-up and works with every pre-trained Segformer model that is published on hugging face. Besides initializing the huggingface model there is also a proposed color pallet. This color pallet can be changed according to a specific use case or preference. The colors need to be in an RGB format.

Step 2 – Inference

The inference is very easy and similar to DETR and LSTR. Each frame should be used to do panoptic detection. This function returns a mask that can be placed upon the original image. The only difference compared to the previous models is the input. Instead of inputting an image array, a PIL image object is required. This can easily be achieved by adding an extra conversion step.

5.3 Implementation recommendations

Now that the implementation steps have been discussed, it is a good idea to investigate some extra recommendations when converting this research into actual production-ready models.

5.3.1 Object detection

The preferred model for object detection comes down to the specific use case. If the scope of a project also includes segmentation tasks it is recommended to start using the original DETR model. It is not the best-performing model, but it does have the advantage of being able to do panoptic segmentation tasks without the need for retraining. If segmentation is not needed it is better to opt for a more improved version of DETR like DINO.

While the manually annotated dataset created for this thesis is a good starting point, it is important to collect more data for retraining or fine-tuning if the model needs to be used in production environments. During the technical research, the gathering of data has been done in the CARLA simulator with the use of a python script. The created script is publicly available on the Github of the technical research[39]. Besides data, it is highly recommended to possess a sufficient amount of computational power, making it easy to train or experiment with the transformer-based models.

5.3.2 Lane detection

Even though LSTR showed some promising results, it is probably the model that should be implemented with the most caution. The model does not have the backing of a big player within the AI field, it was created, and is currently maintained by Ph.D. candidate students. Luckily, they are planning new features to improve their model. One of them is mosaic augmentation, this might ring a bell as it was first introduced in YOLO.

Pretrained weights are available based on the TuSimple dataset. When implementing this model for urban environments it is recommended to fine-tune or retrain if enough data is available. Collecting lane points within CARLA can easily be done by an open-source python script.[7] When using CARLA for data collection it is necessary to convert the data from a JSON format to a .txt format before feeding it into the model. To do this a conversion script has been created during the technical research, this script is publicly available on the Github of the technical research[39] for anyone to use.

5.3.3 Segmentation

The implementation of Segformer does not require a lot of extra recommendations. However, it is important to check all the different options available. The thesis already discussed the different models (B0 up until B5), beside this it is also important to take into account the different resolutions that have been used. The same model trained on a different resolution will differ in speed and accuracy.

5.3.4 Quantizing

When looking for external advice and opinions on using a transformer-based model in production, it becomes clear that the slow inference time is one of the main roadblocks. Finding ways to speed up these heavy models is very important for the future of transformer-based models. Big industry players like Huggingface and Stanford are contributing to research around quantizing transformer models.

The goal of quantizing a model is to reduce the number of bits that represent a float number. For example, from 32 bits to 8bits or even 1 bit. Experiments run by Stanford University show that quantizing can significantly decrease the model in size and make inference less computational heavy. However, they also point out that this process can decrease the model performance. How drastically the decrease depends on the exact architecture and model. Currently quantizing can easily be performed by using the ONNX format through Huggingface.[40][41]

6 Conclusion

At the start of this thesis, the research question was raised “Can the use of transformers grant any advantages to self-driving cars?”. This thesis tries to answer this question by taking a deeper dive into 3 computer vision tasks for self-driving cars. These tasks are object detection, lane detection, and segmentation. To give a better understanding of the state of transformer-based models, a comparison is done with more traditional models. Not only from a theoretical point of view but also with the help of technical research. The technical research shows the working and performance of these transformer models in a realistic self-driving car simulator like CARLA.

Object detection is one of the first computer vision tasks where transformer-based models were introduced. During the theoretical research, it already became clear that plenty of new transformer-based models are starting to arise. Models like DETR and DINO have not been around for a long time, but they are already bringing the fight to the more established YOLO family models on benchmark datasets like COCO.

However, during the technical research, the downsides of a transformer-based model like DETR started to show. The lack of computational resources and big datasets resulted in rather slow and mediocre results compared to YOLOv5. It is important to keep in mind that DETR is a first-generation model while YOLOv5 is the fifth generation of the YOLO family. Later generations like deformable DETR and DINO improve on the original DETR implementation and can perform faster and better.

Unlike object detection, the architecture trend in lane detection is less strong. The benchmarks show a lot of different approaches and in general, the industry is still looking to define the most optimal one. LSTR showed some nice results both in benchmark scores as in the technical research, but it remains a very immature solution that does not have the backing of an established organization to maintain and improve it.

Compared with the previous two computer vision tasks, segmentation is probably the task where transformers shine the most. When looking at benchmark datasets like ADE20 and cityscape most state-of-the-art models make use of transformer-based techniques. When put in practice Segformer gave some very satisfactory results out of the box. A model like Segformer makes it possible for everybody to find a variant to fit their use case. Even though transformer-based models make a very good impression on segmentation tasks. It is important to acknowledge their flaws. Most transformer-based models that achieve high scores use extra training data compared to the CNN models. That is why new research is performed to move toward models like VOLO, which can reach the same scores without the need for extra training.

Answering this thesis eventually comes down to the task and use case. For object detection and segmentation, it looks like there is some added value in using transformer-based tasks. Not only will it help in achieving SOTA results, but it will also give a future-proof approach to these tasks. Lane detection however looks less convincing, whilst results are not bad it might be best to give the industry some extra time in figuring out what way to go.

Although this thesis answered a very specific question in only one industry, there is no doubt that the CNN vs Transformers debate will go on for quite a while. Big names within the AI industry are putting a lot of money and effort into finding breakthrough models and architectures. The coming years will bring a more structured answer to what is truly the best approach, this will have a direct impact on the autonomous driving field and should be followed accordingly.

7 Literature list

[1] A. Vaswani *et al.*, “Attention Is All You Need,” *ArXiv170603762 Cs*, Dec. 2017, Accessed: Mar. 26, 2022. [Online]. Available: http://arxiv.org/abs/1706.03762

[2] A. Prakash, K. Chitta, and A. Geiger, “Multi-Modal Fusion Transformer for End-to-End Autonomous Driving,” *ArXiv210409224 Cs*, Apr. 2021, Accessed: Mar. 26, 2022. [Online]. Available: http://arxiv.org/abs/2104.09224

[3] “ultralytics/yolov5: YOLOv5 🚀 in PyTorch > ONNX > CoreML > TFLite.” https://github.com/ultralytics/yolov5 (accessed Mar. 27, 2022).

[4] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, “End-to-End Object Detection with Transformers,” *ArXiv200512872 Cs*, May 2020, Accessed: Nov. 09, 2021. [Online]. Available: http://arxiv.org/abs/2005.12872

[5] R. Liu, Z. Yuan, T. Liu, and Z. Xiong, “End-to-end Lane Shape Prediction with Transformers,” *ArXiv201104233 Cs*, Nov. 2020, Accessed: Jan. 31, 2022. [Online]. Available: http://arxiv.org/abs/2011.04233

[6] E. Xie, W. Wang, Z. Yu, A. Anandkumar, J. M. Alvarez, and P. Luo, “SegFormer: Simple and Efficient Design for Semantic Segmentation with Transformers,” *ArXiv210515203 Cs*, Oct. 2021, Accessed: Jan. 31, 2022. [Online]. Available: http://arxiv.org/abs/2105.15203

[7] M. Heck, *Carla-Lane-Detection-Dataset-Generation*. 2021. Accessed: Mar. 20, 2022. [Online]. Available: https://github.com/Glutamat42/Carla-Lane-Detection-Dataset-Generation

[8] CodeEmporium, *Transformer Neural Networks - EXPLAINED! (Attention is all you need)*, (Jan. 13, 2020). Accessed: Mar. 27, 2022. [Online Video]. Available: https://www.youtube.com/watch?v=TQQlZhbC5ps

[9] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, “YOLOv4: Optimal Speed and Accuracy of Object Detection,” *ArXiv200410934 Cs Eess*, Apr. 2020, Accessed: Mar. 27, 2022. [Online]. Available: http://arxiv.org/abs/2004.10934

[10] X. Zhu, W. Su, L. Lu, B. Li, X. Wang, and J. Dai, “Deformable DETR: Deformable Transformers for End-to-End Object Detection,” *ArXiv201004159 Cs*, Mar. 2021, Accessed: Mar. 27, 2022. [Online]. Available: http://arxiv.org/abs/2010.04159

[11] H. Zhang *et al.*, “DINO: DETR with Improved DeNoising Anchor Boxes for End-to-End Object Detection,” *ArXiv220303605 Cs*, Mar. 2022, Accessed: Mar. 27, 2022. [Online]. Available: http://arxiv.org/abs/2203.03605

[12] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection,” *ArXiv150602640 Cs*, May 2016, Accessed: Mar. 27, 2022. [Online]. Available: http://arxiv.org/abs/1506.02640

[13] J. S. JUN 4 and 2020 10 Min Read, “YOLOv4 - An explanation of how it works,” *Roboflow Blog*, Jun. 04, 2020. https://blog.roboflow.com/a-thorough-breakdown-of-yolov4/ (accessed Mar. 27, 2022).

[14] Tianxiaomo, *Pytorch-YOLOv4*. 2022. Accessed: Mar. 30, 2022. [Online]. Available: https://github.com/Tianxiaomo/pytorch-YOLOv4

[15] “Papers with Code - COCO minival Benchmark (Object Detection).” https://paperswithcode.com/sota/object-detection-on-coco-minival (accessed Apr. 17, 2022).

[16] L. Tabelini, R. Berriel, T. M. Paixão, C. Badue, A. F. De Souza, and T. Oliveira-Santos, “PolyLaneNet: Lane Estimation via Deep Polynomial Regression,” *ArXiv200410924 Cs*, Jul. 2020, Accessed: Apr. 04, 2022. [Online]. Available: http://arxiv.org/abs/2004.10924

[17] “Understanding Hough Transform With A Lane Detection Model,” *Paperspace Blog*, Jul. 23, 2021. https://blog.paperspace.com/understanding-hough-transform-lane-detection/ (accessed Apr. 04, 2022).

[18] N. Ferdinand, “A Deep Dive into Lane Detection with Hough Transform,” *Medium*, May 05, 2020. https://towardsdatascience.com/a-deep-dive-into-lane-detection-with-hough-transform-8f90fdd1322f (accessed Jan. 31, 2022).

[19] “Papers with Code - TuSimple Benchmark (Lane Detection).” https://paperswithcode.com/sota/lane-detection-on-tusimple (accessed Apr. 17, 2022).

[20] J. Long, E. Shelhamer, and T. Darrell, “Fully Convolutional Networks for Semantic Segmentation,” *ArXiv14114038 Cs*, Mar. 2015, Accessed: Apr. 10, 2022. [Online]. Available: http://arxiv.org/abs/1411.4038
[21] A. Dosovitskiy *et al.*, “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale,” *ArXiv201011929 Cs*, Jun. 2021, Accessed: Nov. 09, 2021. [Online]. Available: http://arxiv.org/abs/2010.11929
[22] “Papers with Code - Cityscapes val Benchmark (Semantic Segmentation).” https://paperswithcode.com/sota/semantic-segmentation-on-cityscapes-val (accessed Apr. 17, 2022).
[23] “Papers with Code - ADE20K val Benchmark (Semantic Segmentation).” https://paperswithcode.com/sota/semantic-segmentation-on-ade20k-val (accessed Apr. 17, 2022).
[24] C. Team, “CARLA,” *CARLA Simulator*. http://carla.org// (accessed Apr. 16, 2022).
[25] “Deepdrive.” https://deepdrive.io/ (accessed Apr. 16, 2022).
[26] “Home - AirSim.” https://microsoft.github.io/AirSim/ (accessed Apr. 16, 2022).
[27] J. Nelson, J. S. JUN 12, and 2020 16 Min Read, “Responding to the Controversy about YOLOv5,” *Roboflow Blog*, Jun. 12, 2020. https://blog.roboflow.com/yolov4-versus-yolov5/ (accessed Apr. 13, 2022).
[28] “Tesla AI Day - YouTube.” https://www.youtube.com/watch?v=j0z4FweCy4M&t=3677s (accessed May 02, 2022).
[29] “Machine Learning,” *Coursera*. https://www.coursera.org/learn/machine-learning (accessed May 15, 2022).
[30] “Deep Learning,” *Coursera*. https://www.coursera.org/specializations/deep-learning (accessed May 15, 2022).
[31] “IBM Machine Learning,” *Coursera*. https://www.coursera.org/professional-certificates/ibm-machine-learning (accessed May 15, 2022).
[32] “Machine Learning,” *Coursera*. https://www.coursera.org/specializations/machine-learning (accessed May 15, 2022).
[33] “Machine Learning: Theory and Hands-on Practice with Python | Coursera.” https://www.coursera.org/specializations/machine-learnin-theory-and-hands-on-practice-with-pythong-cu#instructors (accessed May 15, 2022).
[34] *A ConvNet for the 2020s*. Meta Research, 2022. Accessed: Apr. 19, 2022. [Online]. Available: https://github.com/facebookresearch/ConvNeXt
[35] “Papers with Code - ImageNet Benchmark (Image Classification).” https://paperswithcode.com/sota/image-classification-on-imagenet (accessed Apr. 19, 2022).
[36] “Papers with Code - CULane Benchmark (Lane Detection).” https://paperswithcode.com/sota/lane-detection-on-culane (accessed Apr. 20, 2022).
[37] L. Yuan, Q. Hou, Z. Jiang, J. Feng, and S. Yan, “VOLO: Vision Outlooker for Visual Recognition,” *ArXiv210613112 Cs*, Jun. 2021, Accessed: Apr. 20, 2022. [Online]. Available: http://arxiv.org/abs/2106.13112
[38] “CARLA Dataset > Overview,” *Roboflow*. https://universe.roboflow.com/alec-hantson-student-howest-be/carla-izloa (accessed May 27, 2022).
[39] “Research-Project-CARLA/collect_data.py at main · HantsonAlec/Research-Project-CARLA,” *GitHub*. https://github.com/HantsonAlec/Research-Project-CARLA (accessed May 26, 2022).
[40] “Exporting transformers models.” https://huggingface.co/transformers/v4.6.0/serialization.html (accessed May 26, 2022).
[41] “15742249.pdf.” Accessed: May 26, 2022. [Online]. Available: https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1194/reports/custom/15742249.pdf

8 Attachments

8.1 Additional images

8.1.1 Yolov5 with LSTR

8.1.2 Yolov5 with HT

8.1.3 DETR with LSTR

8.2 Report ML6

8.2.1 Introduction

On Wednesday 19/01/2022 Matthias Feys from ML6 gave an interesting talk about explainability within AI. Explainability is a very important topic because it allows the machine learning engineer to explain to clients or users why the AI models make certain choices. Being able to explain these choices helps the user or client to trust the model.

8.2.2 When AI needs to be explained

To start the session the speaker presented three scenarios where it is important to explain the model's choices. Explaining these scenarios helped to not only understand what AI explainability is but also why and when it is needed.

AI is weaker than a human

When an AI model is weaker than a human for a specific task, explainability can help to improve the AI model. If the ML engineers know why the models do not perform optimally it becomes easier to improve the model. This was very nice explained by the speaker with the help of a real-life example.

To help the clients understand the (poor) decisions a couple of actions can be taken. The easiest thing that can be done is to visualize what the model predicts. The second action is to implement active learning. Doing this enables the user to help the model improve, as time goes on the user will need to do less and less "teaching". The last step is to implement a failsafe that can put the control back in the hands of the user.

AI is equal to a human

If the AI model performs on the same level as a human, it is important to convince the user that the AI model can help them and is not trying to replace them. This can be done by not only making predictions but also giving arguments as to why certain predictions have been made. The example used during the presentation was focused on cold calling. When predicting a certain prospect to call it is also important to supply reasons as to why that prospect is a good fit. These reasons can then be used by the caller to convince the prospect. Without the extra arguments, the caller would be more prone to just following his or her gut feeling. In this case, the explainability added extra value and trust to the AI model.

AI is better than a human

A good-performing AI model is always a good thing, but either way, it is still important to explain the model's predictions. If the user knows why certain predictions are made, it gives them the power and knowledge to improve their internal systems or work. The presentation provides an example where time series prediction for wind turbines has been performed. This is a very good example of where feature importance is very important and can add a lot of value.

8.2.3 Design patterns

To help us out in the future, ML6 has gathered all their knowledge in AI Explainability and has come up with five design patterns.

Problem framing

This design pattern should be used before the project even starts. Before jumping into the code, it is important to know what you are trying to solve and why. Having these conversations with the client can help better understand the scope. Depending on the use case it is possible to create more value for the user by just slightly changing the scope of the model. For example, instead of predicting if a car has damage or not, object detection can be used to visually show what kind of damage is detected on a car.

Interpretable models

When using AI models, a part of your model choice should be influenced by how explainable a model is on its own. It is easier to gain an explanation from certain models compared to others. Traditionally pure machine learning models like Random Forrest trees or Support vector machines are better for explainability. However recently there has been a lot of research into how high-performing deep learning models make their predictions. One approach is to do model decomposition, this means splitting up a big AI task into multiple sub-tasks. Doing this can help to gain a broader understanding of how the big task is solved. A second approach is to use AI to learn rules instead of outcomes. If the model predicts the rules of the outcomes, it can help create a better understanding of the outcomes themself.

Feature attribution

What kind of conclusions can be made from just looking at the most important features of a model? This question can be answered in multiple ways. As mentioned before the more traditional models allow you to instantly retrieve the feature importance. For deep learning models, this can be done by using techniques like SHAP or LIME. These techniques can give insights into what part of a text or images have the biggest impact on the predictions.

Transparency & transferability

When making assumptions based on data analysis the training data must be representative of the test data and real-life data. This can be done by using data visualization tools.

Understanding the data before tackling a project is very important. For example, in some industries, data and behavior changed a lot during the pandemic. Data from before the pandemic can suddenly be less relevant to new projects. Besides this, it is also important to check if the data is realistic. This can be done by implementing some simple rules. For example, the price of a house cannot be 2 euros.

Intuitive visualizations

It is important to find the optimal approach to visualize model decisions. Doing more than just providing a text or number answer can drastically improve how the user is feeling about the model's decisions. For example, highlighting part of an image that is most important or returning multiple results that are all very similar.

8.2.4 Conclusion

During this session, I learned that explainability should be an important part of creating AI models. A user or client might trust the AI model more if he/she can also understand why the model makes certain choices. There are many ways to implement explainability and it is definitely worth it to find the best fit for each individual project.

8.3 Installation guide

8.3.1 vereisten

CARLA

Volgens de officiele CARLA documentatie zijn dit de vereisten:

- Windows of Linux OS.
- GPU met minimum 6GB geheugen, 8GB is aangeraden.
- 20GB vrije ruimte.
- Python3 (python 2.7 is ook mogelijk voor linux).
- 2 vrije TCP poorten, Standaard is dit 2000 & 2001

AI Modellen

Om de AI modellen te gebruiken is het **sterk aan te raden** om gebruik te maken van anaconda. Deze installatie handleiding is gebaseerd op gebruik van anaconda maar kan met een paar installatie aanpassingen ook zonder worden gebruikt.

Verder vereisten:

- Pyhton 3(.7)
- Windows of Linux OS.
- Indien geen Linux OS is het aan te raden toegang te hebben tot een cloud GPU zoals Azure.

8.3.2 Downloads

Download carla

1. Ga naar de downloads pagina op de officiele CARLA github.
2. Kies een CARLA release, in dit geval CARLA 0.9.13.
3. Download de windows of linux versie.
    - Opmerking. Dit project gebruikt de windows versie van CARLA, verder installatie zal zich hier op baseren.
4. Extract de zip file. In deze folder kan je de simulator vinden.
5. De client library zal mee worden geinstalleerd bij installatie van de requirements/enviroment.

Download environment

1. Clone de Research-Project-CARLA repositry naar de PythonAPI folder
    - De PythonAPI folder kan je vinden in de zelfde folder waar de simulator zich bevindt.
2. Maak een anaconda envoirement a.d.h.v. de envoirement.yml file uit de repo.
    - `conda create --file environment.yml`

Download externe githubs

Mijn repositories maakt ook gebruik van enkele externe repositories. Daarom is het ook aan te raden deze te clonen.

1. In de hoofdfolder(RESEARCH-PROJECT-CARLA)
    - Carla-Lane-Detection-Dataset-Generation
        - Voor het genereren van een dataset voor lane detection.

- LSTR
  - Voor het gebruik van hulp classes bij het maken van predictions & training.

2. In de detr train folder(RESEARCH-PROJECT-CARLA/train/detr-pytorch)
   - DETR
     - Voor evaluation bij DETR training.
3. In de yolo train folder(RESEARCH-PROJECT-CARLA/train/yolo)
   - YOLO
     - Voor trainen van YOLO model.

8.3.3 Dataset

dataset download/upload

Om zelf de modellen te hertrainen is het belangrijk de nodige data te hebben. De data waarmee ik de modellen getrained heb is te vinden in een zip file op github of op kaggle. Het is sterk aangeraden deze data te uploaden naar roboflow. Roboflow maakt het makkelijk nieuwe data toe te voegen, augmentatie toe te passen en data te downloaden voor gebruik.

8.3.4 EXTRA

Path update

Na het doornemen van bovenstaande stappen rest er nog 2 zaken.

1. Verplaats de trainings file voor azure van RESEARCH-PROJECT-CARLA/train/lstr-azure/train_azure.py naar RESEARCH-PROJECT-CARLA/LSTR/
2. Verplaats het LSTR model van RESEARCH-PROJECT-CARLA/models/LSTR_500000.pkl naar RESEARCH-PROJECT-CARLA/LSTR/cache/nnet/LSTR/

8.4 User guide

8.4.1 carla

Start simulator

CARLA gebruikt een server-client systeem. De server is de CARLA simulator die draait en data verzamelt. De client kan de CARLA API gebruiken om zo de envoirement te beinvloeden a.d.h.v. python scripts. Het is dus belangrijk dat de CARLA simulator altijd aan het draaien is wanneer je de simulatie wilt starten.

Start simulator

Het starten van de CARLA simulator is zeer eenvoudig:

1. Ga naar de CARLA folder die je hebt gedownload volgens de installatiehandleiding.
2. Start CARLAUE4.exe

aanpassen map

De carla simulator bezit standaard heel wat verschillende mappen. Om de map te veranderen moet de simulator al gestart zijn(zie stap 1.1). De map kan je op volgende manier aanpassen:

1. Open het bestand “CARLA_0.9.13/PythonAPI/Research-Project-CARLA/layers.py”
2. Op lijn 17 kan je de map aanpassen. Alle mogelijkheden zijn te vinden in de CARLA documentatie.

    a. Deze staat momenteel standaard op Town02_Opt
3. Run layers.py en je zal zien dat de map verandert.

Verkeer simuleren

Om de simulatie zo realistisch mogelijk te maken is het mogelijk om verkeer te simuleren. Dit kan a.d.h.v. een script dat carla zelf voorziet.

1. Open een terminal en ga naar "CARLA_0.9.13/PythonAPI/examples"
2. Run generate_traffic.py en het verkeer zal starten.
    a. Aantal voertuigen en voetgangers kunnen eventueel worden aangepast met parameters -n voor "number of vechicles" en -w voor "number of walkers"

8.4.2 data verzamelen

De data die ik gebruikt heb is openbaar verkrijgbaar(zie installatiehandleiding). Indien je verkiest om zelf data te gaan verzamelen kan dat op volgende manier.

Object detectie data

1. Open een terminal en ga naar "CARLA_0.9.13/PythonAPI/Research-Project-CARLA"
2. Run collect_data.py
3. De image data zal nu te vinden zijn in de output folder. ("CARLA_0.9.13/PythonAPI/Research-Project-CARLA/output")
4. De data annotatie kan op verschillende manieren gedaan worden. Zelf heb ik gebruik gemaakt van roboflow.
    a. Dit zijn de labels die ik heb gebruikt: vehicle, person, traffic_light_red, traffic_light_orange, traffic_light_green, traffic_sign_30, traffic_sign_60, traffic_sign_90, bike, motobike
    b. De labels die je gebruikt bij het annoteren kan je zelf kiezen, alle code is dynamisch en zal dus werken met andere labels.

lane detectie data

1. In "CARLA_0.9.13/PythonAPI/Research-Project-CARLA/ Carla-Lane-Detection-Dataset-Generation/src/config.py" kan je instellen hoeveel data je wilt verzamelen per run en voor welke map.
2. Open een terminal en ga naar "CARLA_0.9.13/PythonAPI/Research-Project-CARLA/ Carla-Lane-Detection-Dataset-Generation/src"
3. Run fast_lane_detection.py
    a. Dit maakt numpy files aan in de "data/raws" folder en een JSON file met de coordinaten in de "data/dataset" folder
4. Run dataset_generator.py
    a. Dit maakt de afbeeldingen aan op basis van de numpy files. De afbeeldingen zijn te vinden in de "data/debug" folder.
5. Nu moeten we de coordinaten uit de JSON file omzetten naar een .txt file.
    a. Ga terug naar "CARLA_0.9.13/PythonAPI/Research-Project-CARLA"
    b. Maak een 'labels' folder aan.
    c. Run convert_json_to_txt.py

i. Pas eventueel het pad aan op basis van welke map je gebruikt in de simulator.(standaard Town01_Opt)

8.4.3 trainen

Na het verzamelen van de data kunnen we nu zelf modellen gaan trainen.

Yolov5

1. Ga naar "CARLA_0.9.13/PythonAPI/Research-Project-CARLA/train/yolo/yolov5/models/yolov5s.yaml"
2. Open de file en pas op lijn 4 het aantal classes aan naar het aantal classes dat je zelf hebt.(indien je mijn classes hebt gebruikt is dit nummer 10.)
3. Plaats de folder met data in yolov5 formaat in de yolov5 folder.(zie installatiehandleiding om de data te downloaden)
4. Ga in terminal naar "CARLA_0.9.13/PythonAPI/Research-Project-CARLA/train/yolo/yolov5" een run `python train.py --img 416 --batch 16 --epochs 100 --data ./FOLDER_MET_DATA/data.yaml --cfg ./models/yolov5s.yaml --weights '' --name yolov5s_results  --cache`
5. Het is aan te raden om een wandb account aan te maken om zo makkelijk logs te kunnen ontvangen, wandb support zit geimplementeerd in het yolo trainingscript.
   a) Alternatief kan gebruik worden gemaakt van tensorboard
   Het model is nu terug te vinden in de logs folder:
   "runs/train/yolov5s_resultsX/weights/best.pt"
   b) Indien je dit model wilt gebruiken verplaats te file dan naar "CARLA_0.9.13/PythonAPI/Research-Project-CARLA/models"

detr

1. Ga naar "CARLA_0.9.13/PythonAPI/Research-Project-CARLA/train/detr-pytorch"
2. Plaats de data in de "content" folder handmatig off via roboflow in de notebook.
3. Run train_detr.ipynb
4. Alle verdere info is te zien in de notebook.

lstr

Voor het trainen van het LSTR model heb ik gebruik gemaakt van Azure.

1. Ga naar "CARLA_0.9.13/PythonAPI/Research-Project-CARLA/train/lstr_azure"
2. Maak een .env file aan op basis van env_example.txt en vul uw azure credentials in.
3. Run train.py
4. Na het trainen is het model te vinden op azure in de outputs folder.
5. Download het model en plaats het in "CARLA_0.9.13/PythonAPI/Research-Project-CARLA/models"

8.4.4 Simulatie

modellen

Indien u zelf modellen heeft getrained moet u eerst in de drive_pygame.py file op lijn 224 & 225 de paden aanpassen naar de correcte modellen in de "models" folder.

Start simulatie

1. Open de terminal en ga naar CARLA_0.9.13/PythonAPI/Research-Project-CARLA"
2. Run drive_pygame.py met de gewenste modellen als parameter.
    a. Parameter --lane. Kies het model voor lane detection.
        i. Ht => Houghtransform model
        ii. Lstr=> LSTR model
    b. Parameter –object. Kies het object detection model.
        i. Yolo=> You Only Look Once model.
        ii. Detr=> DETR model.
    c. Parameter –segmentation. Gebruik van segmentation model.
        i. Segform=> Gebruik het segform model.
        ii. BELANGRIJK: Deze parameter is niet combineerbaar met lane en object parameter.
3. VB: `python drive_pygame.py -l lstr -o yolo.`
    a. Start de simulatie met het lstr model voor lane detection en het yolo model voor object detection.

8.4.5 Mogelijke problemen

Script runt niet door timeout error

Twee vaak voorkomende redennen voor deze error zijn:

1. De CARLA simulator is niet gestart
    a. De scripts en simulatie kan alleen werken als de CARLA simulator is opgestart.
2. CARLA simulator draait maar op de verkeerde poort.
    a. Zorg zeker dat poort 2000 vrij is voor CARLA. Indien je poort 2000 niet vrij kan maken dan moet je in de code de poort aanpassen naar de door u gekozen poort.
    b. Het kan zijn dat door een probleem dat CARLA op poort 2000 niet meer reageert, CARLA volledig afsluiten via task manager en herstarten kan dit probleem verhelpen.

Script runt niet door import error

Controleer zeker dat uw scripts runt vanuit de anaconda envoirement.

Yolo training geeft dataset not found error.

1. Ga naar de folder waar uw data zich bevindt.
2. Open de data.yaml file.
3. Controleer als het pad van de train/test/val folders kloppen.