Project report - AutoPylot

Alexandre Girold
Mickael Bobovitch
Maxime Ellerbach
Maxime Gay

Group: Autonomobile

March 2022

Contents

1 Introduction

1.1 Project presentation

Autonomous vehicles and more specifically self-driving cars have grasp the attention of many people for good or ill. In this spirit, we have decided with the Autonomobile team to create our first ever project, AutoPylot. The name of our team is of course full of meaning in that regard. Autonomobile is a two-word name, the first one a French word for autonomous : "Autonome", the second one a French word for car : "Automobile". These two-word combined literally mean Autonomous car.

What is AutoPylot's goal ? Drive itself on a track and win races. It may, at first glance seem very simple but not everything is at it seems. Yet we will try to make it as easy to understand as possible, without omitting crucial information. To achieve our goal, we need to solve many other problems. Those problems can be separate into two distinct groups.

The first one would be the software part. Indeed, in this project we will need to learn and acquire certain skills, from teamwork to coding in different languages. With those newly acquired skills we will be able to bring machine learning to our car to make it drive itself. This leads use directly to our second part, the more tangible one : hardware. Indeed, as we will progress in our work, we will need to see the results of our work in real life condition. This means implementing our code to a functioning car which will be able to race on a track.

This project will lead by a team of four young developers, Maxime Ellerbach, Mickael Bobovitch, Maxime Gay and Alexandre Girold. In this project work will be divided equally amongst all of us, sometimes we will have to work together to achieve our very tight time frame.

1.2 Team members

1.2.1 Maxime Ellerbach

I am a curious and learning hungry person, always happy to learn and collaborate with new people ! Programming, robotics and tinkering has always attracted me. Writing code and then seeing the results in real life is something that I find amazing ! I had multiple projects in this field : Lego Mindstorms, a robotic arm, more recently an autonomous car and even a simulator in unity to train even without a circuit at home ! Even if I know quite well the domain of autonomous cars, there is always something new to learn. I look forward working with this team full of hard-working people on such a fun project !

1.2.2 Mickael Bobovitch

Roses are red. Violets are blue. Unexpected "Mickael BOBOVITCH" on line 32. Hello I am a French Student with Russian parents. Lived half of my life in Moscow. Passionate in web dev, servers, and business. Started programming at 13 years old. Created many projects. I like to learn everything, from AI, to UI, from Hardware to Software. Actually I am like OCaml, you need to know me well to appreciate me.

1.2.3 Maxime Gay

I am 18 years old, and I am crazy about investment, finance and especially cryptocurrencies and blockchain. I already worked with a team on different Investment projects and during summer Jobs, but this is the first time that I am working on such a project. Furthermore, I am a beginner in computer Science and autonomous car. However, I am impatient to learn new skills with this incredible team.

1.2.4 Alexandre Girold

I am already getting old. I am 19 years of age, yet I am full of resources. I am delighted to be able to learn something new. There are many things which I enjoy from programming to geopolitics. I know this project will push me toward a better me and make great friends along the way.

1.3 State of the art

In this section, we will try to see what was previously made in this sector of industry. It would not be realistic to compare our 1:10 project to real sized cars such as Tesla's, simply because in a racing environment, we don't need to deal with such an amount of safety: pedestrian detection, emergency braking, speed limit detection and other. So we will only see miniature autonomous racing framework that we would likely race against.

The most known is called "DonkeyCar", created by Will Roscoe and Adam Conway in early of 2017. Most of the models trained with DonkeyCar are behavior cloning models, meaning models that tries to replicate the behavior of a driver. This method uses a big amount of images (input) associated to steering angles and throttle (output), it requires the user to drive the car (collect data) prior to training the model: no examples means no training. The lack of training data often leads to the car leaving the track.

One other framework worth looking at is one created by Nvidia called "JetRacer" released in 2019. It uses a different approach from DonkeyCar where the user annotates the images by hand by clicking on where the car should go. The model used is similar to what DonkeyCar uses: a Convolutional Neural Network with one input (the image) and two outputs, one for the steering angle and one for the throttle to apply.

Both of those frameworks are written in python and use packages such as Tensorflow and OpenCV, we will also use them in our project.

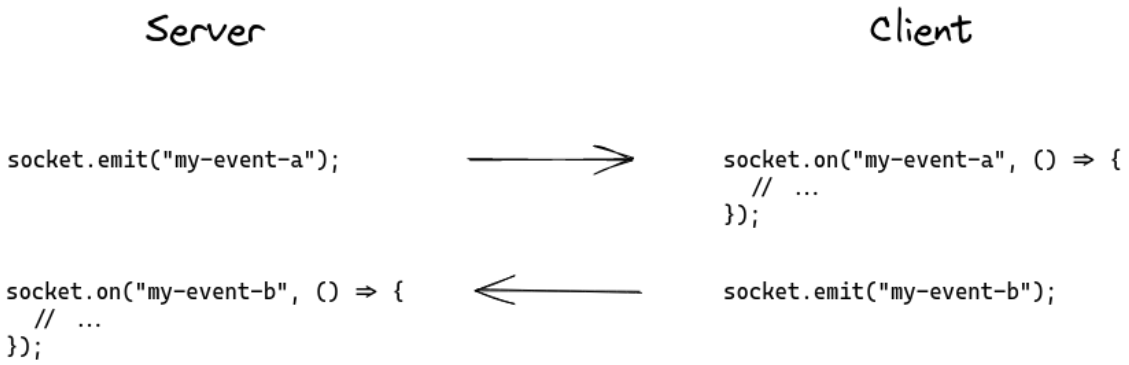
2 Realized tasks

2.1 Telemetry Server

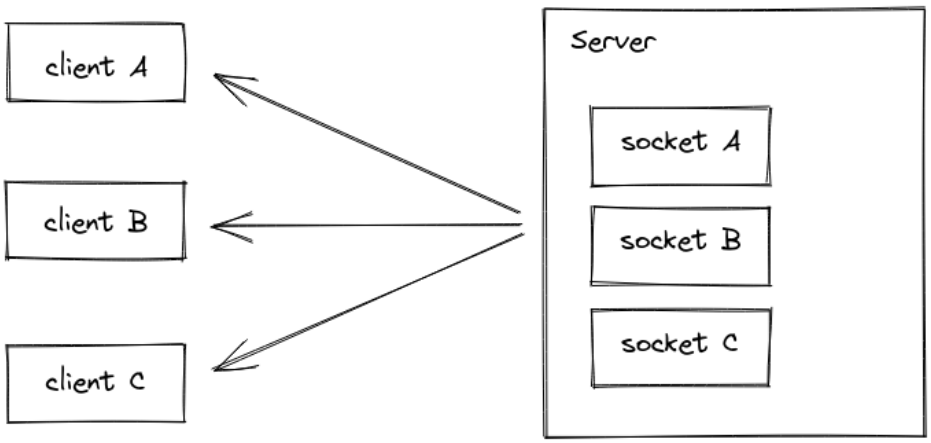
We came to the point of building a telemetry server in a form of a web application. We benefit from great portability, an awesome development process, and a powerful UI. It fills all our requirements. The server makes it possible for us to communicate with the car in real-time. We can detect any problems before they happen. We can control the car without any limitations. We can remotely stop and restart. We can remotely update settings. And we can remotely view logs and car updates.

Now let's see how we implemented this. As seen previously, we have already set up a logger which records the information continuously. Firstly, we extended the functionalities of the logger while maintaining a modular philosophy. The logger gained the ability to send logs and images through a Wi-Fi network. Next, we were confronted to a different challenge: How to send a decent amount of information to multiple clients of different types (web-browsers and raspberry pi hardware) in real-time? We choose to use "Socket.io". This an excellent open source cross-platform library. "Socket.io" enable us to create bidirectional and low-latency communication.

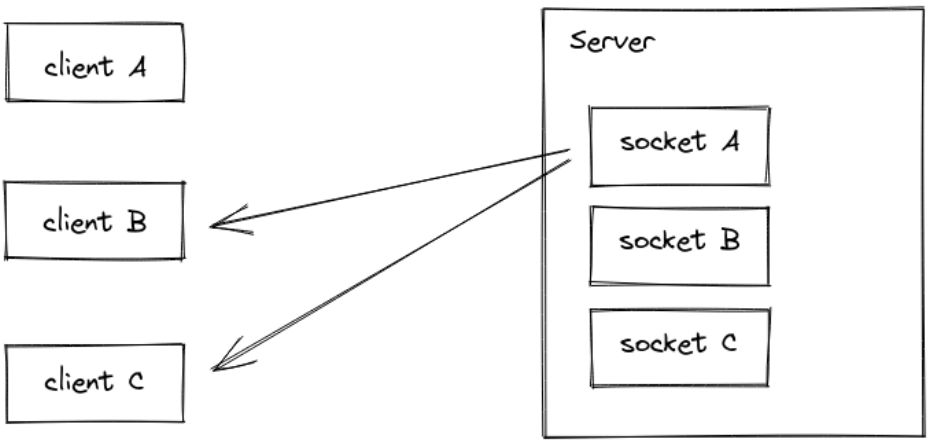
Here is a simple exemple of a communication between a client and a server in javascript. This simple API, simplifies our development process and make data transmition more compliant.



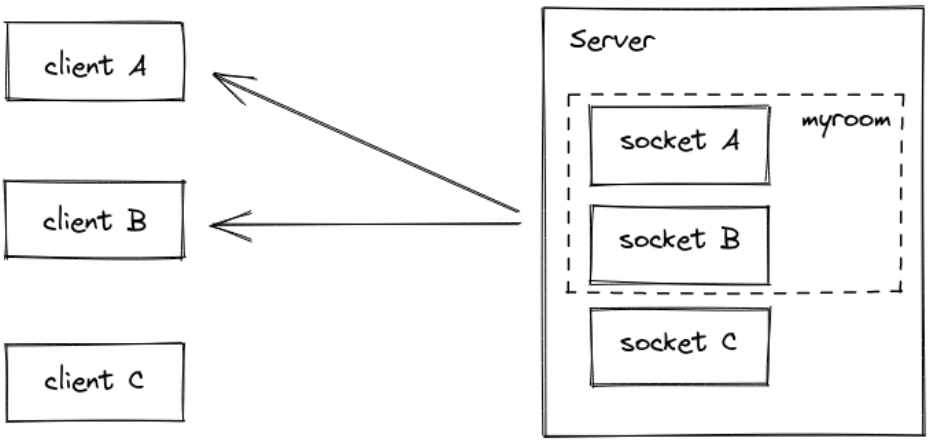
But "Scoket.io" does not stop here. This library is much more powerful. We have the possibility to broadcast a message from the server to all clients :



We can even broadcast from a client to other clients:



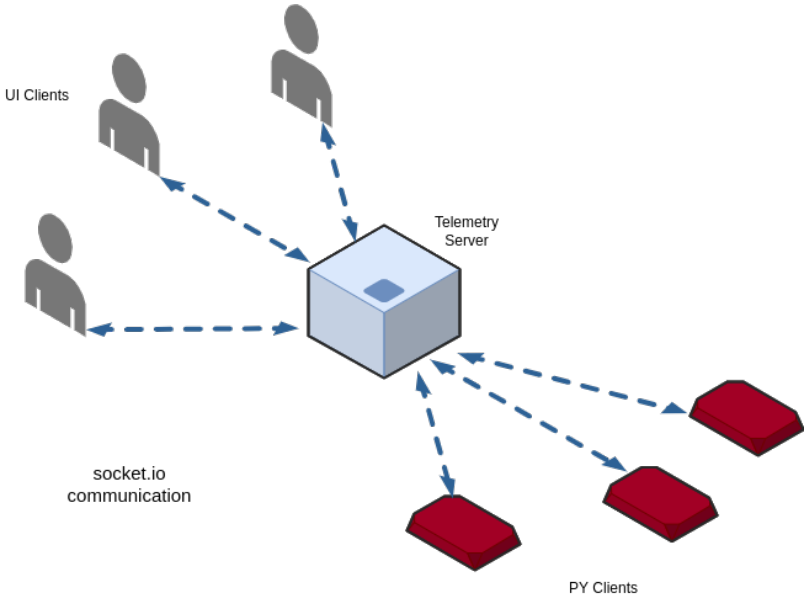
We can even go further by using the concepts of rooms. This means that each client can join a sort of group chart where the server can easily target this group chat:



From now, ui-clients—web-clients—people mean a "socket.io" client instance which runs in a web-browser. A car—py-clients is a "socket.io" client instance wich runs on a raspberry pi4.

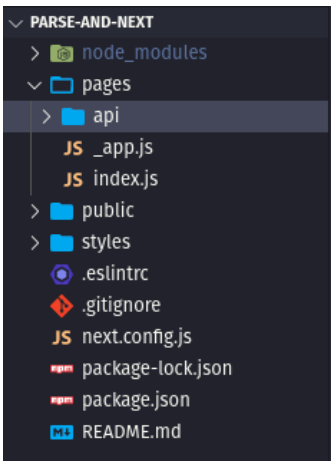
As our main goal is to win races, which implies multiple cars on the track, we have to thinks about a way to manage multiples cars at once. At the same time we wanted every clients of the network would be able to communicate with each other (web-clients with python-clients only). Our design choice was to implement a TV-like mechanism : a web-client can only stream one py-client. And a py-client can send content to an unlimited amount of web-clients, meanwhile a web-clients has the possibility to switch cars at any time.

"Socket.io" is just a tool. We developed a high performance server wich is acting as a middleman between clients:

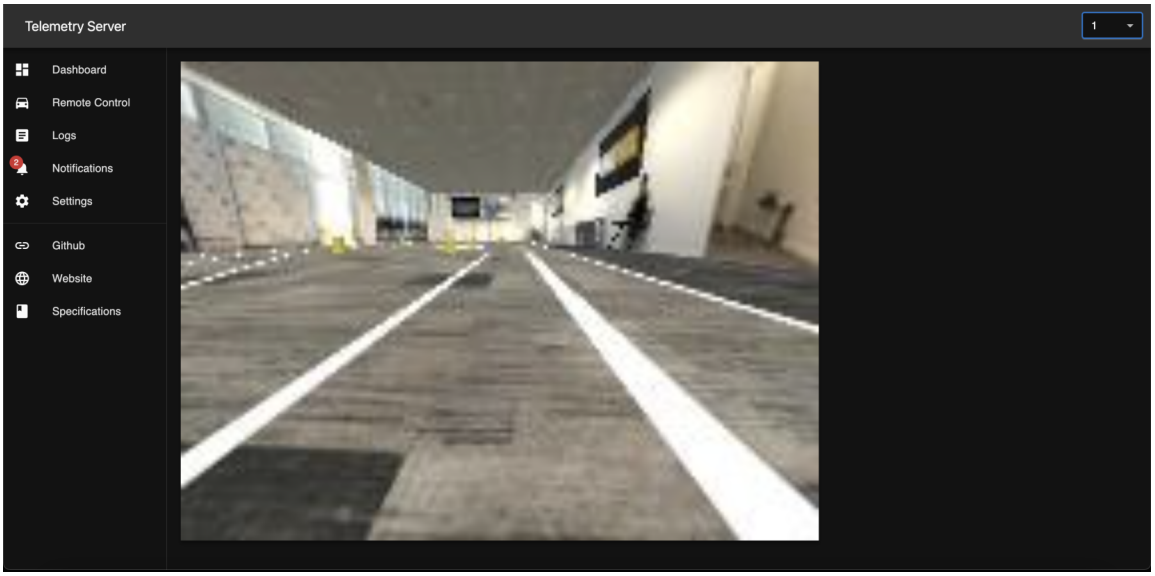


The server uses the JavaScript language (Node JS) as it offers great flexibility, it is very efficient for networking and IO tasks thanks to its language design. Nevertheless, it requires great vigilance because the language is not typed. Any bug could lead to performance issues and memory leaks. But it still remains the best choice for us. We made a great test coverage to extract all bugs.

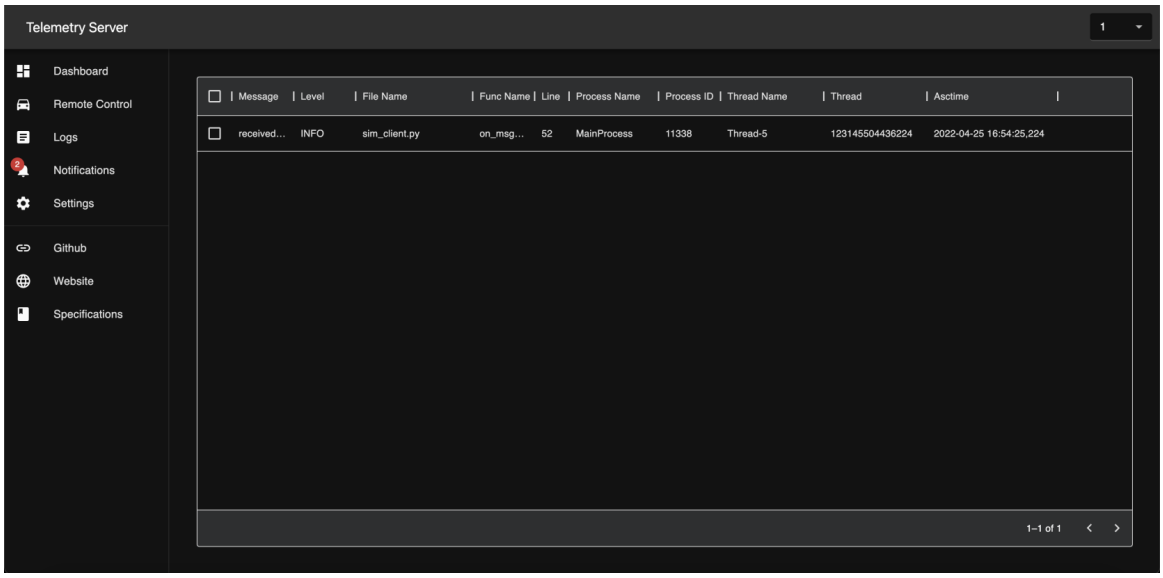
The server is battery included, which means it comes with an exeptional UI inspired by the familiar material design by Google. To build the UI we used the "Next.js" framework which is server side rendered react with an API endpoint. "Next.js" is a top-level industry react framework used in many compagnies. We love "Next.js" for it's simplicity and portability. Here is an exemple of the backend structure which focus on the client side perspective:



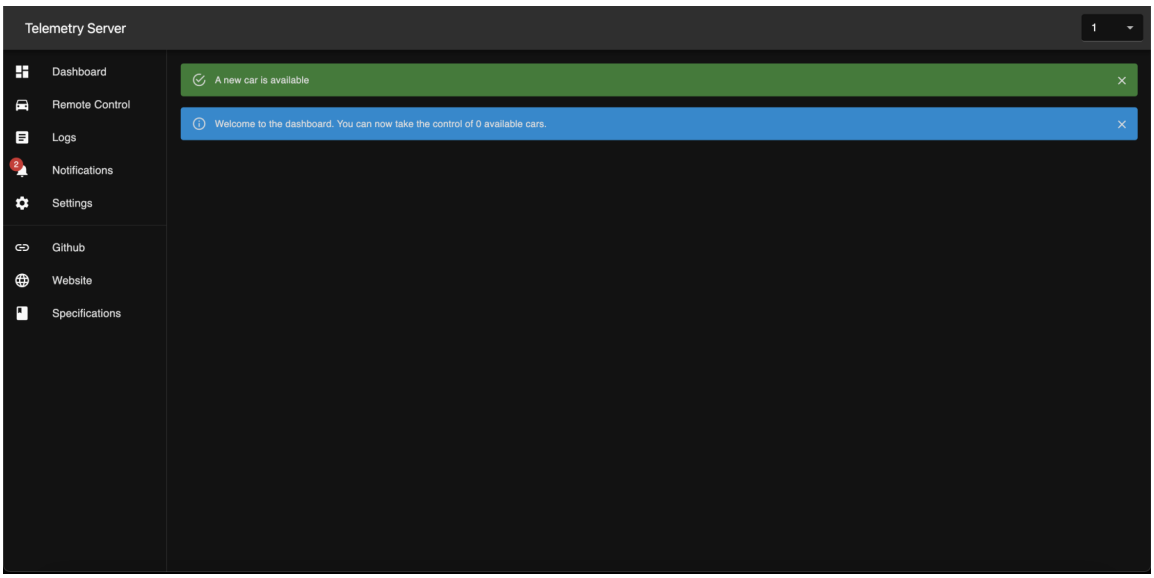
Here is what we have done :



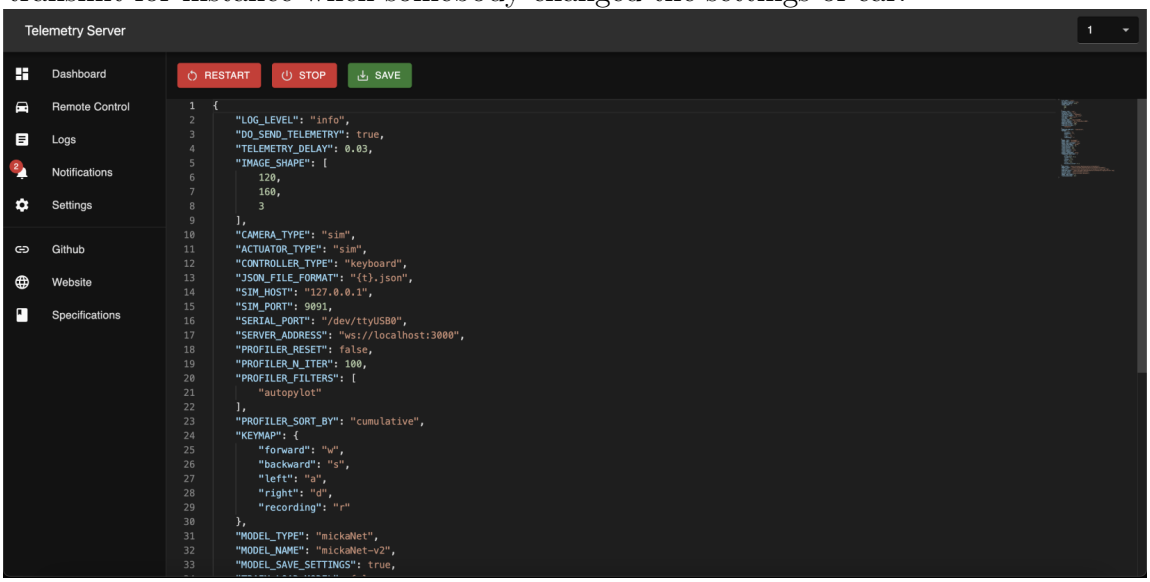
This is the “Remote Control”. This section help us see that the car actualy see in real time. Nothing to say more expted that it’s very cool. We use this in training to we can actualy see better when we drive.



Here are the “Logs”. When someting happen, a new row is inserted. This table support searching. When we have thousands of thousands of logs we can quickly find any message.



Here are the "Notifications". When something happen, We know it. It might be a new Car available for streaming or maybe the server has an important message to transmit for instance when somebody changed the settings of car.



And eventually here are the "Settings". This part is crucial. Here we load the "settings.json" from the car. We can update the car's behaviour, stop or restart with the help of a simple click.

In Conclusion this server is the autopylot's best friend. It's very easy to use and to install, it can also be accessed on a phone. Every time we use the car, we use this telemetry server. This entire server was developed by hand by our team. We hope to make it even better in the future.

2.2 Some theory

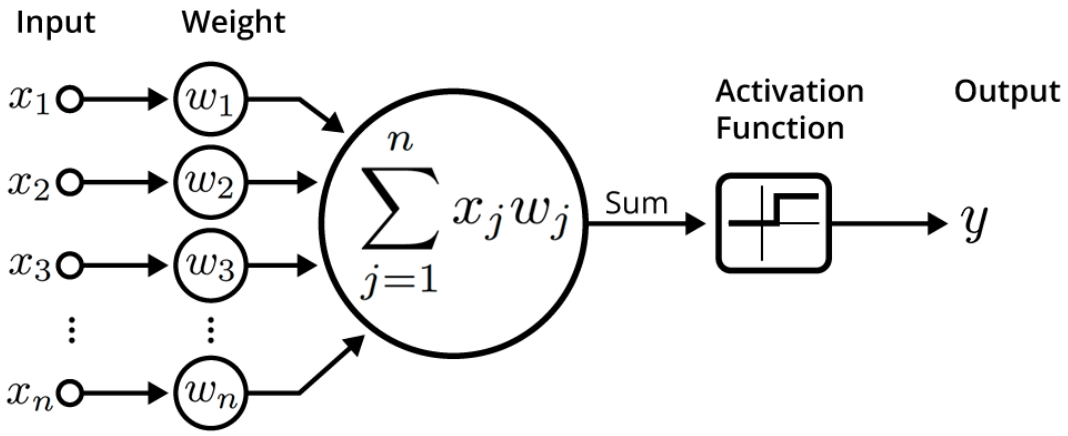
Before going further into the DeepLearning part, let's have a quick reminder of what is a Neural Network and a bit of theory behind all of that.

2.2.1 Neural Network

So, what is a Neural Network ? A Neural Network or "NN" or "NeuralNet" for short is a black box. The role of a Neural Network is to approximate functions. This can be accomplished with a combination of layers that can be also seen as functions with N parameters and P outputs. Each layer can communicate with the next ones. In most architectures the Neural Network can be visualized as a sequential list of layers, but some are more complex featuring: branches, feedforward and other mystical tricks.

This Neural Network by default is only outputting random results, to train it to best approximate our imaginary function, we need one thing: Data ! In our case, we need to predict the next action of our car: steering and throttle from an image: the POV of the car. We can also imagine adding other parameters to our black box like the current speed of the car. During the training process, the prediction (Forward propagation) the model makes along with the expected value are used to correct the weights and inner parameters of every layer (Backward propagation). To have a well-fitted Neural Network, the more the data we have and the best the quality of that data is, the better !

Neural Networks are mainly composed of fully connected layers (or Dense layers), those are composed of a given amount of neurons. They receive one or more input signals, do calculation on them and then communicates its output signals to the next layer. The output signal is calculated from the sum of every input multiplied by its weight along with the addition of a bias. The output is then going through an activation function before outputting to the next layers.



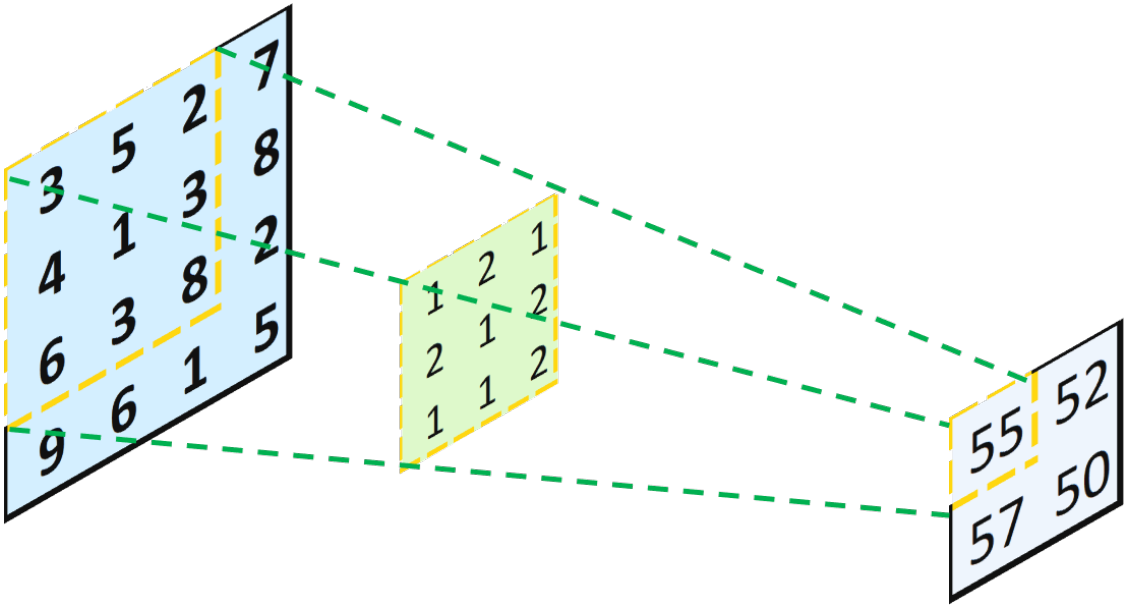
An illustration of an artificial neuron. Source: Becoming Human.

2.2.2 Convolutional Neural Network

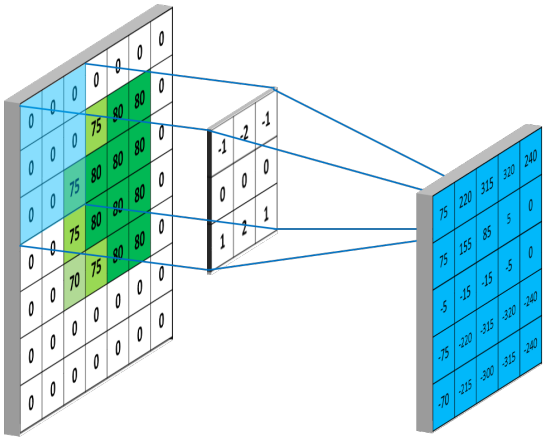
So now, what is a Convolutional Neural Network ? First, a Convolutional Neural Network or "CNN" is a type of Neural Network ! It inherits its name from the kind

of layer it has: Convolution layers.

What is a Convolution ? A convolution is an operation that changes a function into something else. It uses kernels or filters to detect features in a signal. In our case, we use two-dimensional convolution. The signal is the image composed of pixels (usually their values are between 255-0 or 1-0). The main idea behind having convolutions in a Neural Network is to analyze this signal and encode it into a smaller signal.

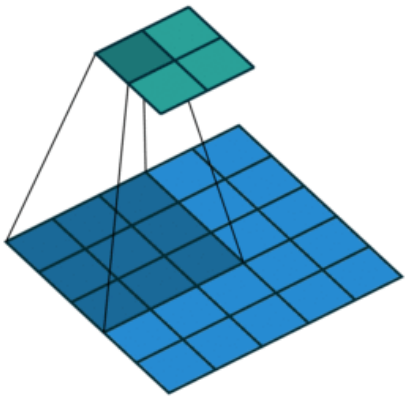


Here you can see the application of a 3x3 convolution kernel on a 4x4 signal, the resulting of the application of this filter is a new signal of size 2x2. The resulting signal is smaller than the input signal, but why is that ? Simply because the kernel here is only applied on the valid areas of the input signal, so the outside border of the signal is lost. To prevent that, we can add zero values around the input filters so that the output size matches the input size.



Now, as said previously, the main idea of having convolutions is to reduce the size of our image, this can be done by introducing strides to our convolution layers. The amount of movement between applications of the kernel of the input signal is referred as the stride. On the above illustrations, we had a stride of 1 meaning at

each step, the kernel moved by 1 pixel. On the illustration below, a 3x3 kernel is applied to a 5x5 signal with strides set to 2 without padding. This results in a 2x2 output signal.

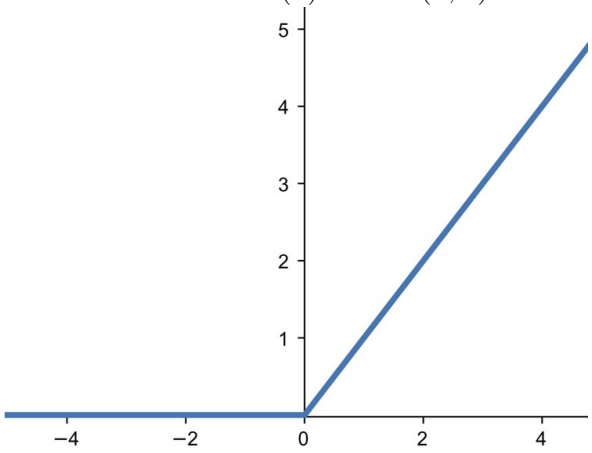


Each filter detects simple features from the previous signal but deepest the Convolutional Neural Network is, the more complex the detected features are. Here is an example where we are trying to detect the face of a dog in an image: The first convolution layer will detect simple features such as edges on the image, those can correspond to the shape of the dog as well as the shape of other stuff in the image. The second will have in input the already detected edges, from those edges it could detect sets of edges looking like features coming from a dog: ears, eyes, fur. The third one will from those features detect even more complex features and so on. After the convolutional layers, we are left with something called the latent space of our image. It is the encoded, simplified form of our image where only the key features are left. From those locally detected features, we can then have a look at the big picture by flattening the 2D signals into a 1D vector to be fed into fully connected layers and then answering the question "Is there a dog in the image?" or "Should we go right or left?".

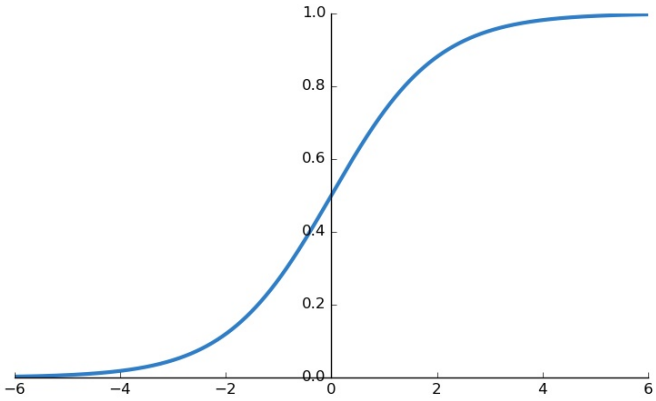
2.2.3 Activation functions

In some cases, we want our output signal to meet some requirements, for example, what if we only want outputs between -1 and 1 ? To answer this need, we apply some activation functions to the output of each layer. Here are some of the most popular activation functions.

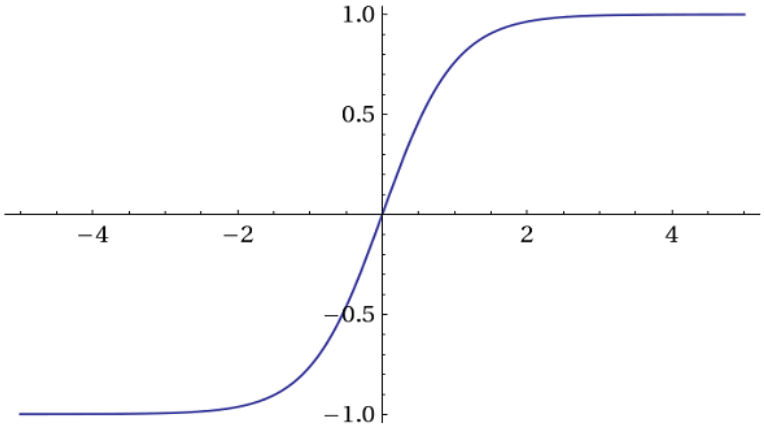
- Rectified Linear unit or "Relu" is the most widely used activation, it cuts off the negative values. It is defined as $Relu(z) = \max(0, z)$.



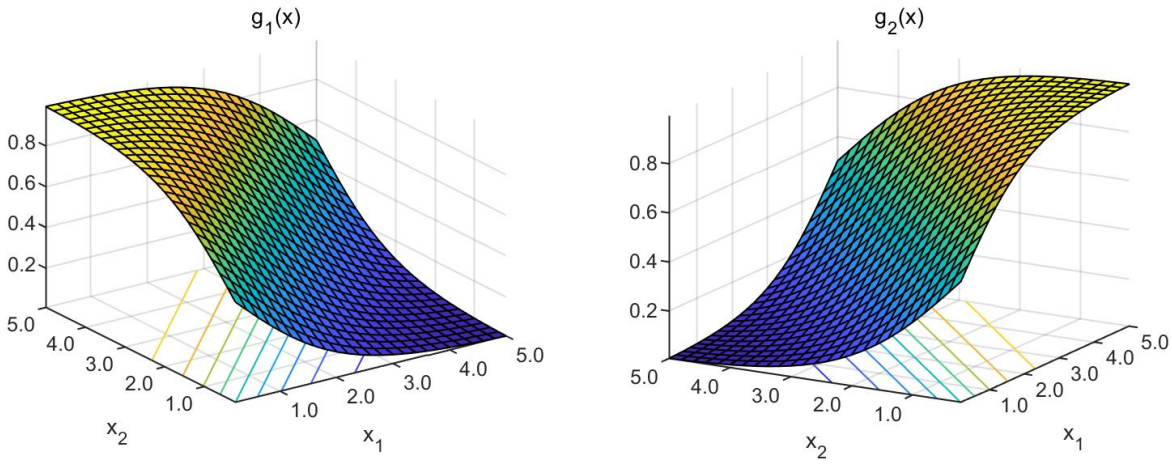
- Sigmoid is another widely used function, it maps every values between 0 and 1. It is defined as $Sigmoid(z) = \frac{1}{1+e^{-z}}$



- Tanh, similarly to Sigmoid maps every values between -1 and 1. It is defined as $Tanh(z) = \frac{1-e^{-2z}}{1+e^{-2z}}$



- Softmax is mainly used for single label classification problems. It is defined as $Softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad for\ i = 1, 2, \ldots, K$
  Here is the plot corresponding to the softmax activation for 2 outputs.



2.3 Model architectures

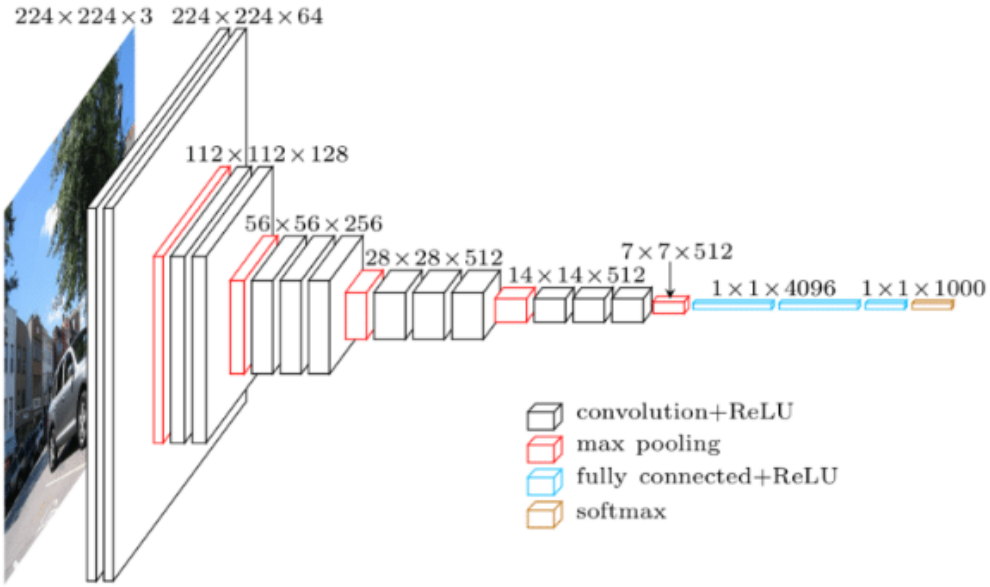
So to summarize a bit what was said, a Convolutional Neural Network is a black box that have at least an image as an input and tries to predict something out of this image. As a common trend, the more convolution layers we have, the better our understanding of the image will be. But we have to be careful of how much we should put because we have to keep in mind the performance aspect of our model. In this section, you will see how every member of the team designed their model. The detailed architectures will be at the end of this section.

2.3.1 Maxime Gay

In order to create my model, I was inspired by the VGG-16 model.

This model was proposed by Karen Simonyan and Andrew Zisserman of the Visual Geometry Group Lab of Oxford University in 2014. It won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) the same year. The model achieves 92.7% accuracy in ImageNet. By the way, ImageNet is a gigantic database of more than 14 million of images. At this time, it was a sharp evolution regarding the other models because VGG-16 uses kernels of a smaller size.

Architecture of VGG-16 model :



The problem with this model is that it is slow, and it is even slower in our case because it must work with our car which does not have a lot of resources. Therefore, I decided to change the VGG-16 Model a bit.

I started by removing some convolutional layers.

My first convolutional layer has a kernel of 5x5 and not 3x3 to have a filter with a bigger size at the beginning. Furthermore, I start with only four kernels on this first layer to reduce the cost of energy. Then with the other layers, I increased progressively the number of kernels and I reduced the kernel size to 3x3. Moreover, I apply a stride of 2 for the first four layers and a stride of 1 for the last two. In the end, I use two Dense layers with an activation function "relu" like the other layers. Finally, I tested many different optimizers like SGC, Adadelta, Nadam and Ftrl but I decided to use the Adam Optimizer because it gave me the best result. The Adam optimizer involves a combination of two gradient descent methodologies.

2.3.2 Mickael Bobovitch

I Tried different implementation. One that worked pretty well for me is a model inspired by Nvidia. I also tried a model based on the AlexNet Architecture. The common point between architecture is their use on 2D convolution. For instance at the top of the stack we have 5 layers of 2D convolution, followed by a Flatten Layer and finally by few Dense Layers. Some architetures used MaxPooling to translate images or change the size of the searched object. This technique is great for searching object that can be anywhere on a picture. But form lane recognition this might be not necessary. I am still experimenting a lot. For the next time i will have a more robust architecture. The only chalenge for me is to find the best values that can produce a great model.

2.3.3 Alexandre Girold

For my part, I decided on making a simple model, capable of steering properly. In order to make this model I spend a lot of time doing some research and I realized as did my comrades that as of today there is one clear winner in the architecture of a prediction model based on an image. It is composed of spatial convolution over images (known as Conv2D) followed by a pooling layer, which is important to avoid overfitting, repeating this a given number of times then followed by a fully connected layer. For these two types of layers, we want to use backpropagation to reduce loss. Finally, the output layer uses functions to give out a probability (I.e: Sigmoid or SoftMax).

In my model, I tried to reduce the number of parameters to try and make it as light as possible. I also tried to incorporate the capacity for the car to adapt its speed, but as of today, I haven't been successful. I am sure i will find a solution soon.

2.3.4 Maxime Ellerbach

When I started to make my model, I thought about: "What could make my model different from the others?" When looking at what my teammates did, it is obvious that the main difference in the model is not in the architecture itself of the model but rather in the data that is fed to it. One other observation is that the less parameters we have in the model, the less sensible it is to overfitting, so the main idea was to have a light model that would also have a rather small latent space.

I did see that the combination of [Layer → Relu Activation → BatchNormalization] worked better with unseen testing images, so I did use this combination for every convolution and Dense layers.

Layer (type)	Output Shape	Param #
image (InputLayer)	[(None, 120, 160, 3)]	0
cropping2d (Cropping2D)	(None, 80, 160, 3)	0
batch_normalization (BatchNo	(None, 80, 160, 3)	12
conv2d (Conv2D)	(None, 38, 78, 4)	300
conv2d_1 (Conv2D)	(None, 17, 37, 8)	800
conv2d_2 (Conv2D)	(None, 8, 18, 16)	1152
conv2d_3 (Conv2D)	(None, 6, 16, 32)	4608
conv2d_4 (Conv2D)	(None, 4, 14, 48)	13824
conv2d_5 (Conv2D)	(None, 2, 12, 64)	27648
flatten (Flatten)	(None, 1536)	0
dropout (Dropout)	(None, 1536)	0
dense (Dense)	(None, 100)	153600
dense_1 (Dense)	(None, 50)	5000
steering (Dense)	(None, 1)	50

Total params: 206,994
Trainable params: 206,988
Non-trainable params: 6

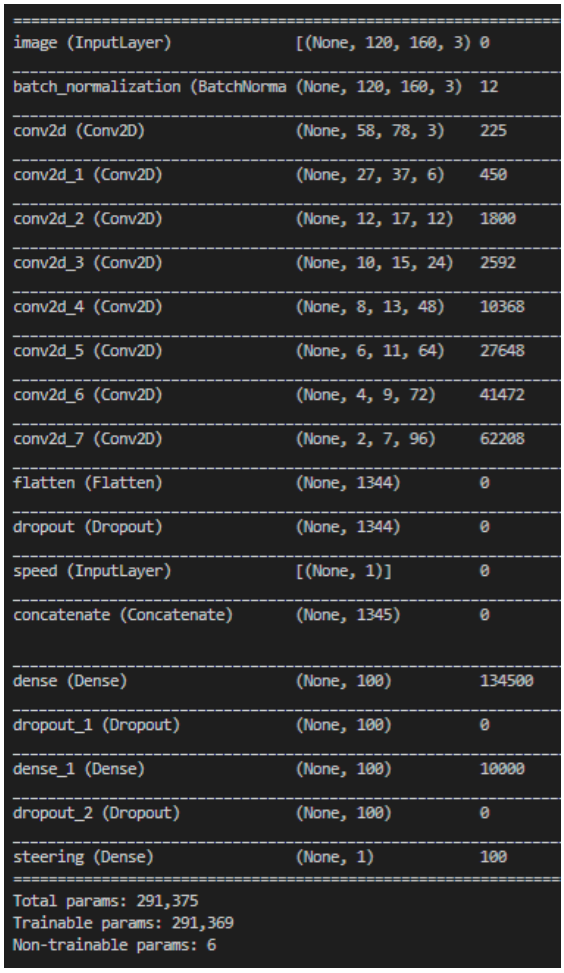
(a) Model MaximeG

Layer (type)	Output Shape	Param #
image (InputLayer)	[(None, 120, 160, 3)	0
cropping2d (Cropping2D)	(None, 80, 160, 3)	0
batch_normalization (BatchNorma	(None, 80, 160, 3)	12
conv2d (Conv2D)	(None, 38, 78, 12)	900
activation (Activation)	(None, 38, 78, 12)	0
conv2d_1 (Conv2D)	(None, 17, 37, 24)	7200
activation_1 (Activation)	(None, 17, 37, 24)	0
conv2d_2 (Conv2D)	(None, 7, 17, 32)	19200
activation_2 (Activation)	(None, 7, 17, 32)	0
conv2d_3 (Conv2D)	(None, 3, 8, 48)	13824
activation_3 (Activation)	(None, 3, 8, 48)	0
conv2d_4 (Conv2D)	(None, 1, 6, 64)	27648
activation_4 (Activation)	(None, 1, 6, 64)	0
flatten (Flatten)	(None, 384)	0
dropout (Dropout)	(None, 384)	0
dense (Dense)	(None, 200)	76800
activation_5 (Activation)	(None, 200)	0
dense_1 (Dense)	(None, 100)	20000
activation_6 (Activation)	(None, 100)	0
dense_2 (Dense)	(None, 100)	10000
activation_7 (Activation)	(None, 100)	0
dropout_1 (Dropout)	(None, 100)	0
steering (Dense)	(None, 1)	100
zone (Dense)	(None, 3)	300

Total params: 175,984
Trainable params: 175,978
Non-trainable params: 6

(b) Model MaximeE

Figure 1: Model Architectures 1/2



Layer (type)	Output Shape	Param #
image (InputLayer)	[(None, 120, 160, 3)	0
batch_normalization (BatchNorma	(None, 120, 160, 3)	12
conv2d (Conv2D)	(None, 58, 78, 3)	225
conv2d_1 (Conv2D)	(None, 27, 37, 6)	450
conv2d_2 (Conv2D)	(None, 12, 17, 12)	1800
conv2d_3 (Conv2D)	(None, 10, 15, 24)	2592
conv2d_4 (Conv2D)	(None, 8, 13, 48)	10368
conv2d_5 (Conv2D)	(None, 6, 11, 64)	27648
conv2d_6 (Conv2D)	(None, 4, 9, 72)	41472
conv2d_7 (Conv2D)	(None, 2, 7, 96)	62208
flatten (Flatten)	(None, 1344)	0
dropout (Dropout)	(None, 1344)	0
speed (InputLayer)	[(None, 1)]	0
concatenate (Concatenate)	(None, 1345)	0
dense (Dense)	(None, 100)	134500
dropout_1 (Dropout)	(None, 100)	0
dense_1 (Dense)	(None, 100)	10000
dropout_2 (Dropout)	(None, 100)	0
steering (Dense)	(None, 1)	100

Total params: 291,375
Trainable params: 291,369
Non-trainable params: 6

(a) Model MickaelB

Layer (type)	Output Shape	Param #
image (InputLayer)	[(None, 120, 160, 3)	0
batch_normalization (BatchNorma	(None, 120, 160, 3)	12
conv1 (Conv2D)	(None, 58, 78, 8)	600
conv2 (Conv2D)	(None, 54, 74, 16)	3200
conv3 (Conv2D)	(None, 26, 36, 24)	3456
conv4 (Conv2D)	(None, 12, 17, 32)	6912
conv5 (Conv2D)	(None, 5, 8, 48)	13824
conv7 (Conv2D)	(None, 2, 3, 96)	41472
flatten (Flatten)	(None, 576)	0
dropout (Dropout)	(None, 576)	0
fc1 (Dense)	(None, 200)	115200
fc2 (Dense)	(None, 100)	20000
speed (InputLayer)	[(None, 1)]	0
concatenate (Concatenate)	(None, 101)	0
dense (Dense)	(None, 100)	10200
dropout_1 (Dropout)	(None, 100)	0
dense_1 (Dense)	(None, 30)	3030
steering (Dense)	(None, 1)	101
throttle (Dense)	(None, 1)	31

Total params: 218,038
Trainable params: 218,032
Non-trainable params: 6

(b) Model AlexandreG

Figure 2: Model Architectures 2/2

2.4 Load data during training

Thanks to the datagenerator file, we can load data to train the model. The class DataGenerator inherits from "Sequence", a tensorflow.keras utils. We give in inputs a list or a list of lists of json paths to train on. To represent those data, we use X to represent the input data and Y the expected outputs. Both are lists of numpy arrays containing the data. Then it returns a tuple containing X and Y.

How does it work?

Firstly, we created X and Y which are lists of numpy array. Then we had to pick randomly some paths according to the batch size in order to have more realistic data to train on. Secondly, thanks to the previous function load_image_data, we can load the data of the image using a json path

Thirdly, we just have to add the result that we are looking at to the X or to the Y.

2.5 Training Process

After loading our data, we need to be able to train on those loaded data. The training process is where everything takes place. Indeed, training the data requires

the use of all the previous functions. One of these functions is the settings.py. You can see how the function works with the small example below.

```
# Training settings
self.TRAIN_LOAD_MODEL = False
self.TRAIN_BATCH_SIZE = 32
self.TRAIN_EPOCHS = 10
self.TRAIN_SPLITS = 0.9
self.TRAIN_SHUFFLE = True
self.TRAIN_VERBOSE = 1
self.TRAIN_AUGM_FREQ = 0.3
```

This function is the backbone of the training process. It allows us to have a very modular project where everything can be modified from this file only. This allows us to modify our project without deleting a lot of code, making it very modular which was one of our goals from the beginning. For example, we have recently added the telemetry server's restart function, once it was created, we only had to add it to the settings.py and all the other functions would be able to use it. This is also useful to change values in real-time. For example, if we want to train our model on more epochs or that our computer can handle bigger batch sizes then we only must change one value and that is it. This goes for all the functions in our project.

The second import function is of course the train.py function. At first, this function creates all our useful variables (imported from the settings). Then, if the given model doesn't exist, we create it, for now, this model hasn't yet been trained. Then we need to feed the make sure all the variables are correct so that the model can be trained. To train a model we need data, in our case images. For this we need to make sure those data exist, and that the path to them is correct. We also want to know all the inputs and outputs. This is important as we might want to train not only the steering but also the throttle and speed. Finally, we use one important training library. The .fit library from keras. With fit, we are first feeding the training data(X) and training labels(Y) in our case these X and Y come from DataGenerator. We then use Keras to allow our model to train for a given number of epochs and on a given amount of batch sizes.

2.6 Data Augmentation

During the learning process we have the possibility to use data augmentation. It's a technique used to diversify training data and to apply different kind of situations with too much lighs or not enough. The AI has to consider the road without light effects. Data augmentation is used too add noise on training data. It's very useful because it forces neurons to find betters and more complex features in images. The more features are complex, the less we can experiment overfitting. Which means the AI is more capable to drive in a unknown environment and use complex features to drive better.

3 Planning

3.0.1 What is next ?

The objectives we set ourselves for this presentation were acheived, for the next intermediate presentation we plan to finish what we are currently working on meaning The telemetry server and the logging. Moreover, we also plan to have a working prototype of the whole car including the AI part with the development of a basic convolutional neural network in a first time. This means we will have to create a model, then have a script to train it using collected data and finally a script to drive our car using this trained model.

3.0.2 Races

Tasks	Race 1	Race 2	Race 3	Race 4	Race 5	Race 6
Code controlled motors and servo	75%	100%				
Drive the car with a controller	25%	100%				
Data collection		50%	100%			
Telemetry server		25%	100%			
Logging		25%	100%			
Data processing and augmentation			50%	75%	100%	
Basic Convolutional neural network			25%	50%	100%	
Advanced models and optional objectives						50%

3.1 Presentations

Tasks	1st presentation	2nd Presentation	Final presentation
Code controlled motors and servo	100%		
Drive the car with a controller	100%		
Data collection	75%	100%	
Telemetry server	25%	100%	
Logging	25%	100%	
Presentation website	100%	Update	Update
Data processing and augmentation		75%	100%
Basic Convolutional neural network		75%	100%
Advanced models and optional objectives			50%

4 Task allocation

Tasks	Mickael B.	Maxime G.	Alexandre G.	Maxime E.
Low level car control				x
Driving with a controller		x	x	x
Dataset handling		x	x	
Data processing	x	x	x	x
Data visualization				x
Telemetry server	x			x
Logging	x			x
Presentation website	x			
Convolutional neural network	x	x	x	x
Main control loop	x			

5 Conclusion

To sum up, the Autonomobile team has made huge strides towards its final goal. We worked on generating useable data putting them into output and inputs. We also modified the images making them look different from one another to make the training and the future model not overfit. All of us made our own models and we held a little competition to determine the best, this created enthusiasm and teamwork. Finally, we improved our telemetry server making it smoother than ever and adding more and more functionalities, from simply being able to modify settings to using the server on our phones thanks to a QR code. We are more invested than ever and plan to continue this project beyond the S2 as we have planned to make our own tracks for future competitions at Epita.