AI Project

Self Driving Car

Team 05

Robin Ramaekers

Yunus Kurban

Stan Swinnen

INHOUDSTAFEL

1. INLEIDING

In dit verslag zullen we aantonen hoe we tot het resultaat van de 'Self Driving Car' gekomen zijn. We zullen tonen welke stappen we genomen hebben, welke problemen we tegengekomen zijn en hoe we ze opgelost hebben en uiteindelijk welk resultaat we behaald hebben. Tot slot trekken we ook nog een conclusie over het totale project.

2. VERLOOP PROJECT

Voor we aan het project begonnen, hebben we het opgesplitst in verschillende kleinere onderdelen, namelijk:

- Installatie Unity en Udacity Driving Sim
- Edge Detection met OpenCV op een afbeelding
- Edge Detection met OpenCV in Real Time
- De auto aansturen met OpenCV
- Deep Learning model trainen
- De auto aansturen met het deep learning model
- Extra: F1 Game

Deze zullen we verder in detail bespreken.

2.1 Installatie Unity en Udacity Driving Sim

Als eerste hebben we de gratis game making engine Unity gedownload van hun website. Deze hebben we nodig om alle assets van de simulator te kunnen laden. Vervolgens hebben we uit deze GitHub Repo: https://github.com/udacity/self-driving-car-sim de driving simulator zelf gedownload.

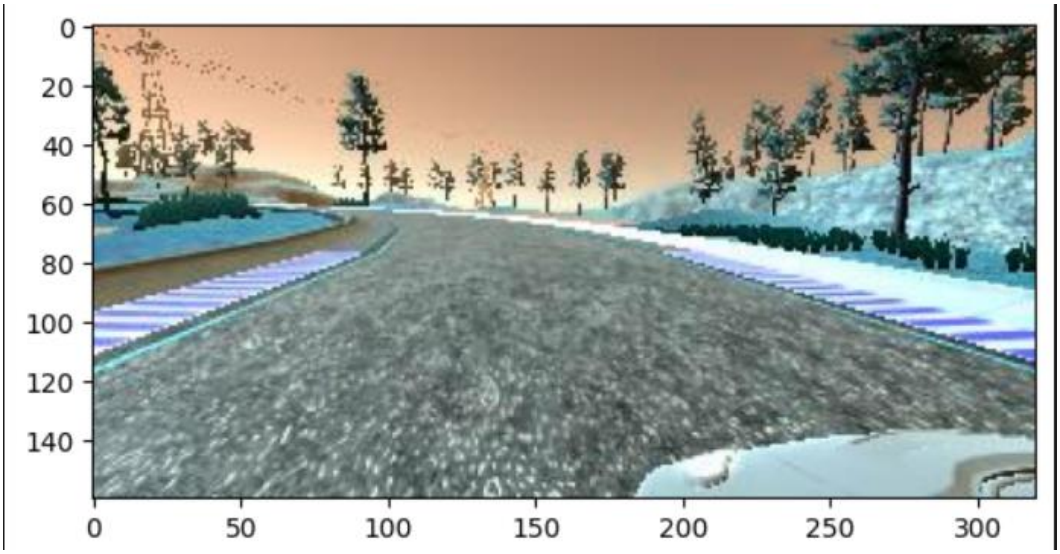
Na de installaties voltooid waren hebben we enkele rondjes gereden in de simulator om de omgeving wat te leren kennen en al wat beelden op te nemen voor we aan de slag gingen met OpenCV.

2.2 OpenCV Edge Detection

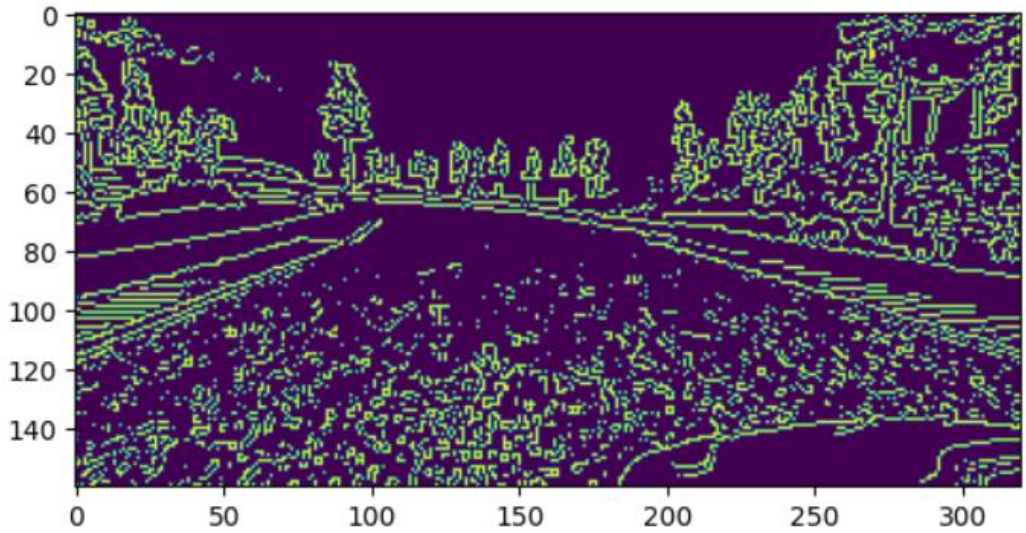
Als tweede stap in het proces hebben we geprobeerd om aan de hand van OpenCV (Computer Vision) edge detection toe te passen op een afbeelding die we uit de simulator gehaald hebben.

2.2.1 Edge detection on image

Eerst hebben we alle nodige packages voor Computer Vision geïnstalleerd. Vervolgens gaan we de import doen van de nodige libraries en packages. We lezen de foto in a.d.h.v. de OpenCV (cv2) en vervolgens plotten we de foto a.d.h.v. pyplot(Matplotlib).

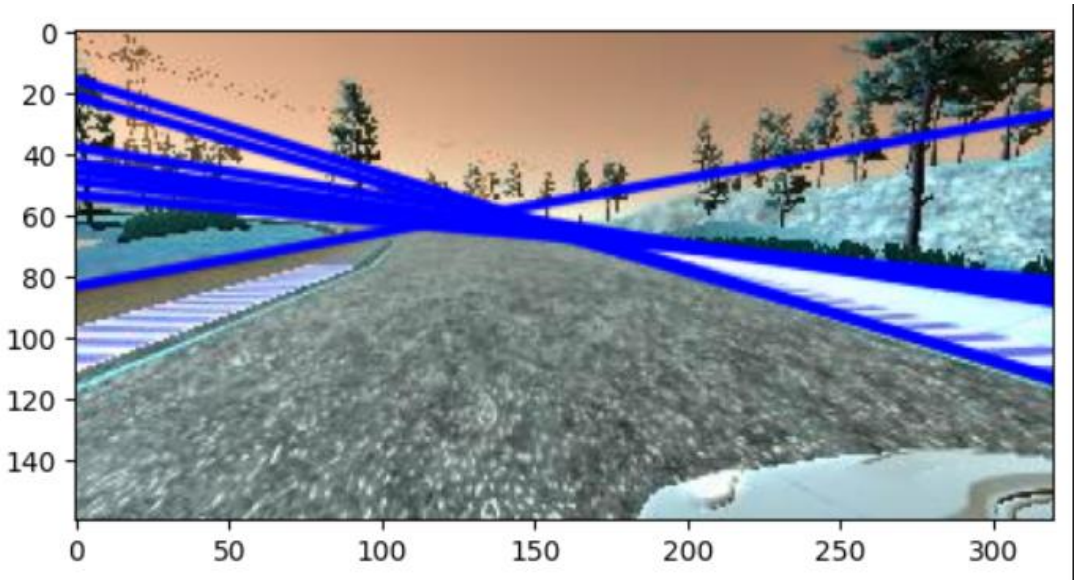


Vervolgens hebben we Canny edge detection toegepast op de foto en hebben we de lijnen berekend. De Canny edge detector is een edge detection operator die een meerstappenalgoritme gebruikt om een breed scala aan randen in beelden te detecteren.



Hierna hebben we de lijnen berekend aan de hand van de Houghlines function. Dit is een functie die toelaat om rechte lijnen te berekenen aan de hand van randen die gedetecteerd zijn bij de edge detection. Deze functie geeft dan vectoren weer van deze lijnen die we vervolgens op onze afbeelding hebben getekend. Bij het zoeken van deze lijnen merkten we dat er soms veel te veel lijnen stonden en soms geen. Na wat onderzoek kwamen we tot de conclusie dat we met de parameters moesten schuiven tot we een goed resultaat behaalden. In ons geval gaf parameter 120 het beste resultaat op de meeste afbeeldingen.

```
#lines= cv2.HoughLines(dst, 1, math.pi/180.0, 100, np.array([]), 0,0) #100 geeft teveel lijnen, 250 geeft er geen
lines= cv2.HoughLines(dst, 1, math.pi/180.0, 120, np.array([]), 0,0)
```



Vervolgens hebben we het snijpunt van deze lijnen berekend. De houghlines functie geeft de vectoren in polaire coördinaten weer. Dit betekent dat we in tegensteling tot bij cartesische coördinaten een punt vastleggen door een hoek(theta) en een straal(rho) te definiëren. Om deze coördinaten om te vormen tot cartesische coördinaten maken we gebruik van volgende formules:

$$x = rcos\theta$$

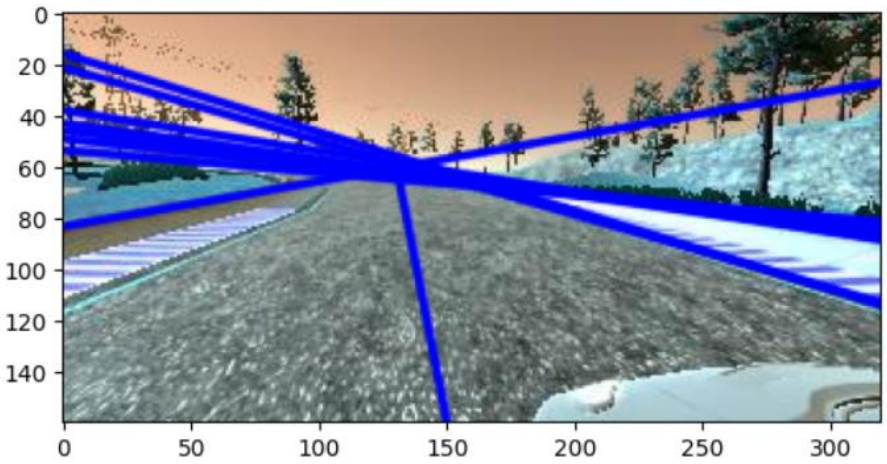
$$y = rsin\theta$$

Dit passen we toe voor 2 punten en dus 2 lijnen. Hier passen we vervolgens een functie op toe die toelaat het snijpunt van die lijnen te bereken. Zo krijgen we de x- en y-waarde van het snijpunt.

```python
def intersection(line1, line2):
    rho1, theta1 = line1[0]
    rho2, theta2 = line2[0]
    A = np.array([
        [np.cos(theta1), np.sin(theta1)],
        [np.cos(theta2), np.sin(theta2)]
    ])
    b = np.array([[rho1], [rho2]])
    x0, y0 = np.linalg.solve(A, b)
    x0, y0 = int(np.round(x0)), int(np.round(y0))
    return [[x0, y0]]

inersection_val = intersection(lines[0],lines[1])
inter1 = inersection_val[0][0]
inter2 = inersection_val[0][1]
```

Vervolgens hebben we een lijn vanuit het midden onderaan het scherm tot in het snijpunt getekend op de image.



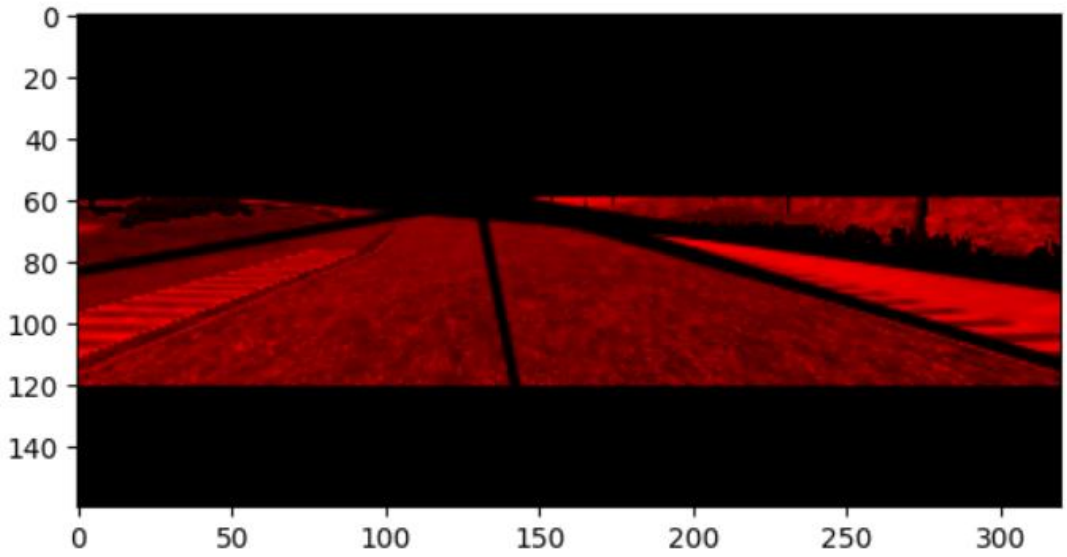
De volgende stap was uit die lijn de hoek te berekenen. Deze hoek hadden we nodig om de steering angle voor de auto te kunnen meegeven. Omdat de driving simulator enkel steering angles van -25 tot 25 accepteert, geven we mee dat elke waarde onder of boven die waardes worden gelimiteerd tot respectievelijk -25 en +25 graden.

```python
degree = np.math.atan2(160 - inter2, 150 - inter1)
degree = np.degrees(degree) - 90

if degree > 25:
    degree = 25
elif degree < -25:
    degree = -25

print(degree)
```

Aangezien we niet het hele scherm nodig hebben, gaan we een ‘Region of Interest’ definiëren die enkel het nuttige deel van de afbeelding toont.



2.2.2 Edge detection in real time

Nadat we edge detection en dus het bereken van de steering angle konden toepassen op een afbeelding, was de volgende stap dit te implementeren op een real time video feed.

Dit hebben we gerealiseerd door een try lus te zetten rond een cv2.imshow commando en hierin geven we de process_image functie in mee.

```
try:
    screen = np.array(ImageGrab.grab(bbox=(0, 40, 1024, 768)))
    new_screen, original_image, m1, m2 = process_img(screen)

    cv2.imshow('window2', cv2.cvtColor(original_image, cv2.COLOR_BGR2RGB))
    if cv2.waitKey(25) & 0xFF == ord('q'):
        cv2.destroyAllWindows()
```

2.2.3 Problemen

Bij dit deel van het project zijn we niet veel problemen tegengekomen. Maar het voornaamste probleem was er een met de Houghlines functie. We kwamen hier maar niet tot een goed resultaat en na een tip van de docent en wat research kwamen we er achter dat we een bepaalde parameter beter moesten optimaliseren. Na wat trial en error kwamen we tot een goed resultaat.

2.3 OpenCV toepassen op de Driving Simulator

De volgende stap was dit te koppelen aan de steering angle en throttle die aan de auto wordt meegegeven. Zo probeerden we de auto te laten rijden aan de hand van OpenCV.

2.3.1 SocketIO

De eerste stap was dit proberen koppelen aan de driving simulator via SocketIO. Hierbij wordt er intern een server opgesteld die communiceert met de racing simulator en zo de besturing doorgeeft vanuit de python code. We geven in de code een vaste snelheid mee en indien die wordt overschreden gaat de auto afremmen. De steering angle wordt meegegeven door onze OpenCV code.

```
#initialize our server
sio = socketio.Server()
#our flask (web) app
app = Flask(__name__)
```

```
#registering event handler for the server
@sio.on('telemetry')
def telemetry(sid, data):
    if data:
        # The current steering angle of the car
        steering_angle = float(data["steering_angle"])
        # The current throttle of the car, how hard to push peddle
        throttle = float(data["throttle"])
        # The current speed of the car
        speed = float(data["speed"])
        # The current image from the center camera of the car
        image = Image.open(BytesIO(base64.b64decode(data["image"])))
        try:
            image = np.asarray(image)       # from PIL image to numpy array
            image = utils.preprocess(image) # apply the preprocessing
            image = np.array([image])       # the model expects 4D array

            # predict the steering angle for the image
            steering_angle = float(model.predict(image, batch_size=1))
            # lower the throttle as the speed increases
            # if the speed is above the current speed limit, we are on a downhill.
            # make sure we slow down first and then go back to the original max speed.
            global speed_limit
            if speed > speed_limit:
                speed_limit = MIN_SPEED  # slow down
                throttle = -1
            else:
                speed_limit = MAX_SPEED
                throttle = 0.5

            print('{} {} {}'.format(steering_angle, throttle, speed))
            send_control(steering_angle, throttle)
        except Exception as e:
            print(e)

        # save frame
```

Dit werkte echter niet. De server startte op maar communiceerde niet met de simulator. De auto kwam dus niet in beweging en er gebeurde niets. Dus besloten we te kijken naar andere opties.

2.3.2 Keyboard inputs

Aangezien SocketIO initieel niet wou werken, hebben we gezocht naar andere oplossingen. We kwamen op het idee om keyboard inputs te simuleren om zo de auto te doen rijden.

```
def straight():
    PressKey(W)
    ReleaseKey(S)
    ReleaseKey(A)
    ReleaseKey(D)

def left():
    PressKey(A)
    ReleaseKey(W)
    ReleaseKey(D)
    ReleaseKey(S)

def right():
    PressKey(D)
    ReleaseKey(A)
    ReleaseKey(W)
    ReleaseKey(S)

def slow_ya_roll():
    PressKey(Z)
    ReleaseKey(A)
    ReleaseKey(comma)
    ReleaseKey(dot)
# def main():
```

Dit werkte en de auto begon te rijden in de train modus van de simulator. Het probleem met deze manier van werken is dat een keyboard input alles of niets is. De throttle wordt dus volledig ingedrukt en als er wordt bijgestuurd, gaat de auto altijd onder de maximum steering angle draaien. Dit maakt dat de auto wel kan rijden, maar dat het niet heel vlot gaat.

2.3.3 Problemen

Zoals eerder vermeld, doken de eerste problemen reeds op bij het opstellen van de SocketIO server. De connectie tussen python en de simulator kwam wel tot verbinding maar de code werd niet doorgegeven. Na lang zoeken en de hulp van een medestudent zijn we tot de conclusie gekomen dat het lag aan de versie van Python, SocketIO en EngineIO. Deze waren niet compatibel waardoor er niets gebeurde. Online(https://github.com/udacity/self-driving-car-sim/issues/131) vonden we dat we engineIO en SocketIO moesten downgraden aan de hand van deze commands:

```
pip install python-engineio==3.13.2
pip install python-socketio==4.6.1
```

Na het downgraden van die packages, kwam de auto uiteindelijk in bewegen en merkten we dat de auto werd aangestuurd. Hij begon te rijden.

2.4 Model training met CNN

2.4.1 Model trainen

Om het model te trainen hebben we eerst een aantal rondjes gereden in de simulator om data te verzamelen. Dit deden we door te recorden binnen de simulator. Deze data werd opgeslagen als screenshots van de linkse, centrale en rechtse camera vanuit het perspectief van de auto. Er werd ook een driving log bijgehouden waar de steering angle(de hoek waaronder de auto stuurt) werd gelinkt aan de verzamelde beelden. Deze beelden werden gebruikt om een CNN te trainen.

Deze data hebben we gesplitst in training en validation data.

```
def load_data(args):
    """
    Load training data and split it into training and validation set
    """
    #reads CSV file into a single dataframe variable
    data_df = pd.read_csv(os.path.join(os.getcwd(), args.data_dir, 'driving_log.csv'), names=['center', 'left', 'right', 'steering', 'throttle', 'reverse', 'speed'])

    #yay dataframes, we can select rows and columns by their names
    #we'll store the camera images as our input data
    X = data_df[['center', 'left', 'right']].values
    #and our steering commands as our output data
    y = data_df['steering'].values

    #now we can split the data into a training (80), testing(20), and validation set
    #thanks scikit learn
    X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=args.test_size, random_state=0)

    return X_train, X_valid, y_train, y_valid
```

Ons model is gebaseerd op het bestaande Nvidia ‘Self driving car’ model. Dit model hebben we verder getraind met onze eigen data om zo tot een model te komen dat werkte voor onze toepassing. We hebben gekozen om ons op dit model te baseren omdat dit ons reeds een goede basis gaf en het zo minder lang duurde om te trainen.

```python
def build_model(args):
    """
    NVIDIA model used
    Image normalization to avoid saturation and make gradients work better.
    Convolution: 5x5, filter: 24, strides: 2x2, activation: ELU
    Convolution: 5x5, filter: 36, strides: 2x2, activation: ELU
    Convolution: 5x5, filter: 48, strides: 2x2, activation: ELU
    Convolution: 3x3, filter: 64, strides: 1x1, activation: ELU
    Convolution: 3x3, filter: 64, strides: 1x1, activation: ELU
    Drop out (0.5)
    Fully connected: neurons: 100, activation: ELU
    Fully connected: neurons: 50, activation: ELU
    Fully connected: neurons: 10, activation: ELU
    Fully connected: neurons: 1 (output)

    # the convolution layers are meant to handle feature engineering
    the fully connected layer for predicting the steering angle.
    dropout avoids overfitting
    ELU(Exponential linear unit) function takes care of the Vanishing gradient problem.
    """
    model = Sequential()
    model.add(Lambda(lambda x: x/127.5-1.0, input_shape=INPUT_SHAPE))
    model.add(Conv2D(24, 5, 5, activation='elu', subsample=(2, 2)))
    model.add(Conv2D(36, 5, 5, activation='elu', subsample=(2, 2)))
    model.add(Conv2D(48, 5, 5, activation='elu', subsample=(2, 2)))
    model.add(Conv2D(64, 3, 3, activation='elu'))
    model.add(Conv2D(64, 3, 3, activation='elu'))
    model.add(Dropout(args.keep_prob))
    model.add(Flatten())
    model.add(Dense(100, activation='elu'))
    model.add(Dense(50, activation='elu'))
    model.add(Dense(10, activation='elu'))
    model.add(Dense(1))
    model.summary()

    return model
```

```python
def train_model(model, args, X_train, X_valid, y_train, y_valid):
    """
    Train the model
    """
    #Saves the model after every epoch.
    #quantity to monitor, verbosity i.e logging mode (0 or 1),
    #if save_best_only is true the latest best model according to the quantity monitored will not be overwritten.
    #mode: one of {auto, min, max}. If save_best_only=True, the decision to overwrite the current save file is
    # made based on either the maximization or the minimization of the monitored quantity. For val_acc,
    #this should be max, for val_loss this should be min, etc. In auto mode, the direction is automatically
    # inferred from the name of the monitored quantity.
    checkpoint = ModelCheckpoint('model-{epoch:03d}.h5',
                                 monitor='val_loss',
                                 verbose=0,
                                 save_best_only=args.save_best_only,
                                 mode='auto')

    #calculate the difference between expected steering angle and actual steering angle
    #square the difference
    #add up all those differences for as many data points as we have
    #divide by the number of them
    #that value is our mean squared error! this is what we want to minimize via
    #gradient descent
    model.compile(loss='mean_squared_error', optimizer=Adam(lr=args.learning_rate))

    #Fits the model on data generated batch-by-batch by a Python generator.

    #The generator is run in parallel to the model, for efficiency.
    #For instance, this allows you to do real-time data augmentation on images on CPU in
    #parallel to training your model on GPU.
    #so we reshape our data into their appropriate batches and train our model simulatenously
    model.fit_generator(batch_generator(args.data_dir, X_train, y_train, args.batch_size, True),
                        args.samples_per_epoch,
                        args.nb_epoch,
                        max_q_size=1,
                        validation_data=batch_generator(args.data_dir, X_valid, y_valid, args.batch_size, False),
                        nb_val_samples=len(X_valid),
                        callbacks=[checkpoint],
                        verbose=1)
```

2.4.2 Problemen

In dit proces hebben we ook behoorlijk veel problemen gehad. Als eerste werden de headers van de csv(driving log) niet gevonden. De data van de images was incorrect en een degelijk model met een goede validation loss krijgen bleek niet zo eenvoudig. Om deze problemen op te lossen hebben we de volgende stappen ondernomen:

- Headers toevoegen in de driving log (center, left, right, steering, throttle, reverse, speed)
- Het pad van de images in de driving log aanpassen naar de huidige directory.
- Het model trainen op 20 epochs in de plaats van 10 epochs

Na verschillende pogingen en verschillende modellen getraind te hebben zijn we tot een model gekomen dat in onze ogen moest volstaan om de auto aan te sturen.

2.5 De simulator aansturen met het model

Nadat het model getraind was hebben we het model gedeployed op de Udacity driving simulator. Dit hebben we gedaan via python code die een constante snelheid doorgeeft en de steering angle in real time berekent op basis van wat op het scherm te zien is. Deze informatie wordt weer via SocketIO doorgestuurd naar de simulator.

Ook hier kwamen enkele problemen naar boven, maar deze waren voornamelijk te wijten aan het model dat niet voldoende getraind bleek te zijn. Dit hebben we opgelost door het model verder te trainen met meer epochs, maar ook door meer data te verzamelen en het rondje een aantal keer in de beide richtingen te rijden.

2.6 Extra: F1 Game

Als extra wilden wij een integratie doen naar de F1 2019 game. Hiervoor hebben wij de code die wij met Udacity driving simulator hebben gebruikt, hergebruikt. Dit is redelijk vlot verlopen maar er waren wat problemen met het controleren van de snelheid van de wagen. Hij ging meteen naar zijn maximumsnelheid en vloog dus snel uit de bocht. Om dit op te lossen zouden we gebruik moeten gemaakt hebben van een virtuele controller maar door tijdsgebrek hebben we dit geschrapt. We wilden ook een deep learning model maken om de auto zelfrijdend te maken. Hiervoor hebben we data kunnen verzamelen maar we hebben geen juiste tool gevonden om screenshots te linken aan de driving log.

3. RESULTAAT

3.1 Self driving car met OpenCV

3.1.1 Keyboard inputs

Als eerst hebben we de 'Udacity driving sim' aangestuurd via keyboard inputs. We gebruiken hiervoor onze real time edge detection en een library die functies heeft om de auto via toetsen aan te sturen. Op basis van de hoek die onze lijn heeft sturen we de auto naar links of rechts. De auto rijdt niet heel vlot aangezien de keyboard inputs binair zijn. Oftewel wordt er niets doorgegeven, of alles. Daardoor gaat hij vaak te snel of stuurt hij te hard waardoor hij tegen de rand rijdt.

Link naar demo: https://www.youtube.com/watch?v=j8xh6MdLywI

3.1.2 SocketIO

De integratie naar socketio was niet veel verschillend, we gebruiken nog steeds de in real time edge detection. De steering angle van de auto is hier gebaseerd op de berekende hoek. In socketio is het ons gelukt om de precieze hoek door te geven in de plaats van alleen rechts en links.

3.2 Self Driving car met DL model

Uiteindelijk zijn we tot een resultaat gekomen waarbij de auto het eerste parcours probleemloos op zichzelf kan afleggen. Hij rijdt tegen een constante snelheid van 25mph en gebruikt het model dat we zelf getraind hebben om de juiste steering angle te berekenen en door te geven. Hierdoor kan hij het parcours afleggen zonder uit de bocht te vliegen of de controle te verliezen.

Link naar demo parcours 1: https://youtu.be/3tkrmjMD91I

Bij het tweede parcours rijdt hij tot op zeker hoogte ook probleemloos, maar rijdt hij na verloop van tijd toch tegen de muur. Dit komt doordat ons model getraind is op het eerste parcours en niet specifiek op het tweede. Desalniettemin is het model toch ook deels bruikbaar voor dit parcours.

Link naar parcours 2: https://youtu.be/Y6-63QuHWaE

4. CONCLUSIE

In dit project hebben we geleerd hoe we aan de hand van de zeer uitgebreide OpenCV library een zelfrijdende auto kunnen maken. Dit gebeurde aan de hand van edge detection en hough lines. We zijn ook gevorderd in onze kennis van socketio en hoe we dit kunnen gebruiken om de udacity driving sim aan te sturen. Op vlak van deep learning was dit weer een grote herinnering dat correcte en voldoende data zeer belangrijk is in het trainen van een AI-model en zelfs dan zul je nog moeten spelen met parameters zoals epochs en batch size. Over het algemeen was dit een zeer interessante en leuke opdracht, we zijn ook veel obstakels overkomen maar op het einde zijn we er toch geraakt.