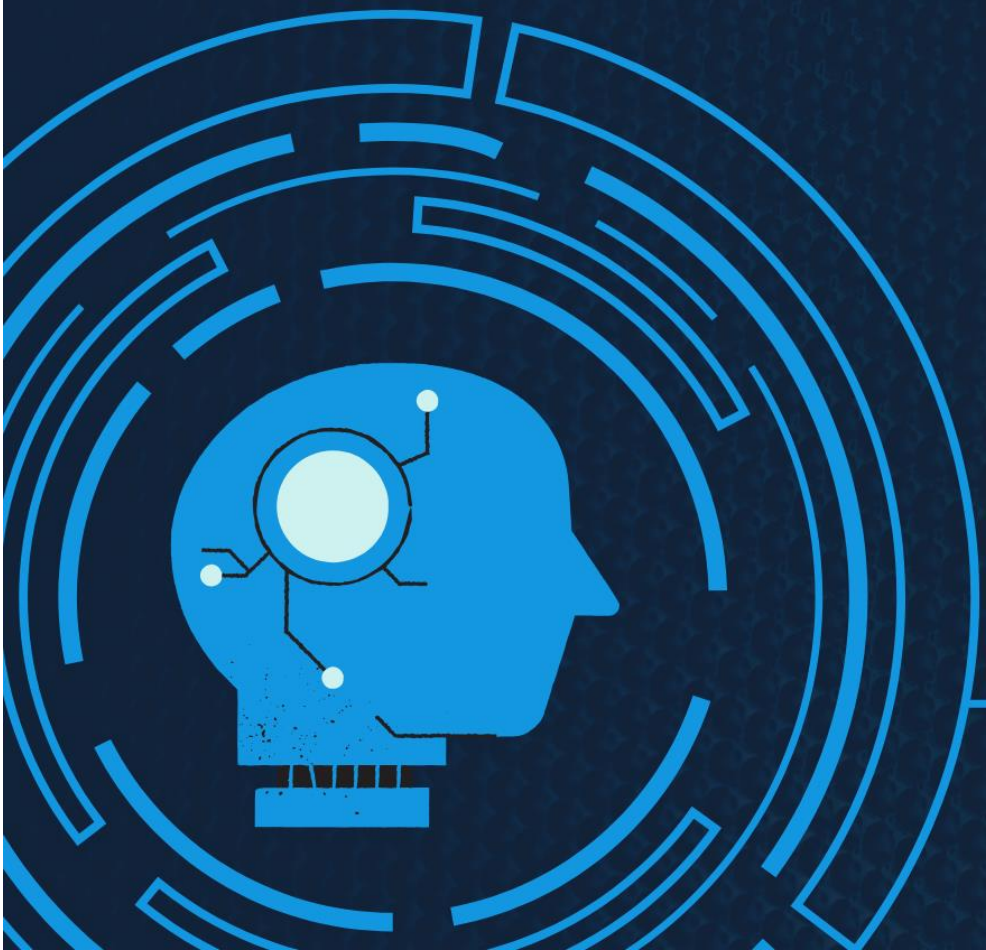
TRACKMANIA REPORT 2024

Presented by
Kobe Schoeters
Thorsten Ooms



1. Project Information

The objective of this project is to develop an AI capable of autonomously driving in the racing game *Trackmania*. The approach involves training the AI on labeled images from the game to predict driving actions. These predictions will correspond to specific key inputs, enabling the AI to navigate the track in real time based on visual data. The end goal is to create an AI that can interpret incoming game images and generate precise driving commands, such as steering, accelerating, and braking, to complete tracks effectively.

Contents

1. Project Information 2
2. Approach 4
2.1 Screen Capturing 4
2.2 Key Logging 4
2.3 Cropped Images 5
2.4 Data Cleaning 5
2.5 Training the AI 6
2.6 Re-Cropping images 7
3. Solution 8
3.1 Take screenshots 8
3.2 Train model 11
3.3 Test model 13
4. Obstacles 17
4.1 Prediction Rate 17
4.2 Racing Line 17
4.3 Flipping & Rotating 17
5. GenAI 18
6. Conclusion 19

2. Approach

2.1 Screen Capturing

We began the project by capturing screenshots while driving in the game. These screenshots formed the foundation of the dataset used to train our model. Initially, we started with full-screen images of the game.

2.2 Key Logging

Once we had the image capture process in place, we added code to log key inputs and label each screenshot with the corresponding key input. This was done using a CSV file, where one column recorded the image name and the other column recorded the associated key input.

```
Screenshot,Key Pressed
screenshot_0.jpg,Key.up
screenshot_1.jpg,Key.up
screenshot_2.jpg,Key.up
screenshot_3.jpg,Key.up
screenshot_4.jpg,Key.up
screenshot_5.jpg,Key.up
```

2.3 Cropped Images

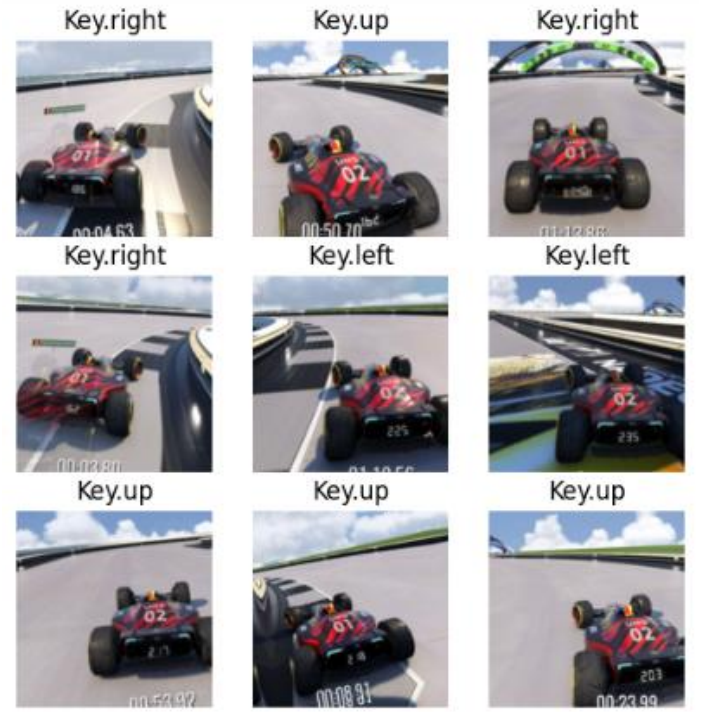
After analyzing the initial screenshots, we realized that they contained a lot of unnecessary information. To address this, we decided to crop the images to focus only on the car and the main part of the track. This ensured that the model trained on the most relevant parts of the scene.

2.4 Data Cleaning

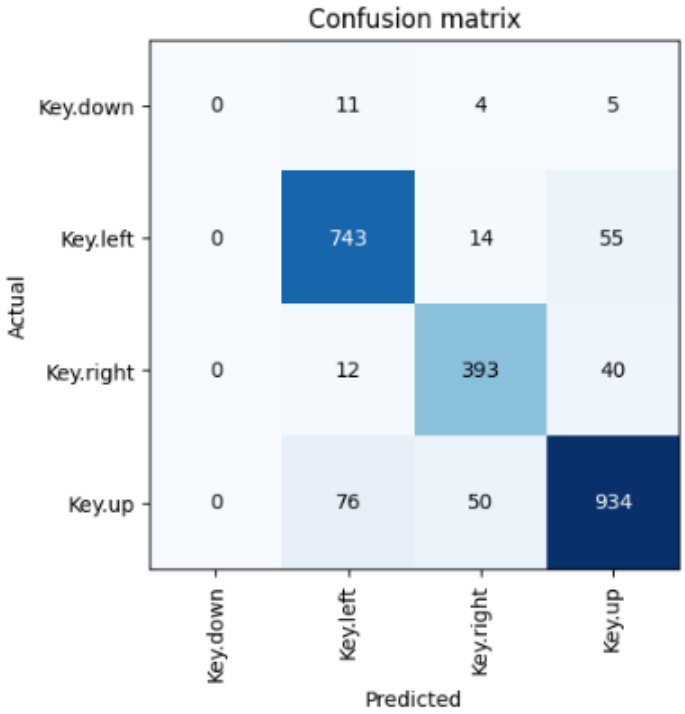
The next step was to clean our dataset by filtering out images where the car was driving toward a wall. Including these images in training could inadvertently teach the AI to steer into obstacles. The image below shows an example of one of the filtered-out images.

2.5 Training the AI

The next thing we did was loading in our data and training our model.

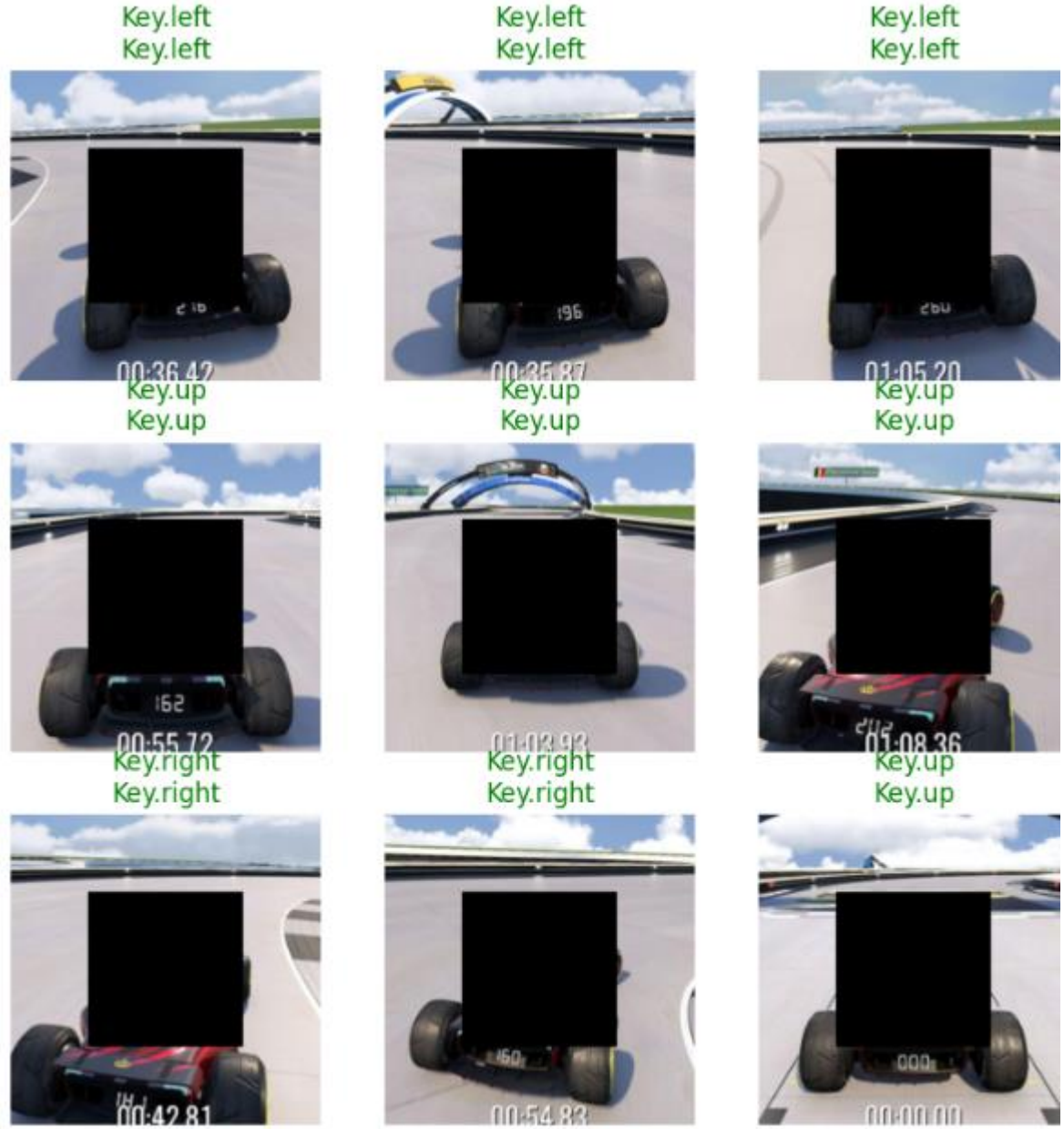


To evaluate our model's performance, we created a confusion matrix that displays the predictions on the validation dataset.



2.6 Re-Cropping images

After the initial tests, we thought that the model was placing too much emphasis on the position of the car's wheels. To address this, we decided to obscure the car by placing a black box over it. This adjustment allowed us to observe whether it would improve the model's predictions by reducing its focus on the car itself and increasing its focus on the track ahead.



Eventually we went back to the full images because the original model was slightly better.

3. Solution

3.1 Take screenshots

We began by capturing a significant number of screenshots—11,723, to be exact. The initial images include our code used to take these screenshots. Rather than capturing the entire screen, we focused on recording only a specific "box" area.

```python
import time
import mss
import csv
from PIL import Image
from pynput import keyboard
import threading

screenshot_count = 0
key_input_log = []
stop_flag = False
log_lock = threading.Lock()  # Ensure thread safety when modifying key_input_log

# Specify the bounding box (this limits the screenshot size)
monitor = {"top": 190, "left": 460, "width": 1200, "height": 650}  # Modify as per the game window
```

Here's the function we use to capture screenshots. To keep file sizes manageable, each screenshot is saved as a JPEG file.

```python
# Capture the screen region only on key press
def take_screenshot(filename):
    global screenshot_count
    try:
        # Initialize a new MSS object inside the function
        with mss.mss() as sct:
            # Capture the defined region (smaller screen area)
            screenshot = sct.grab(monitor)

            # Convert to an image and save it (JPEG for smaller size)
            img = Image.frombytes("RGB", screenshot.size, screenshot.rgb)
            img.save(filename, format='JPEG', quality=80)  # Compress to save space and speed

        screenshot_count += 1
    except Exception as e:
        print(f"Error while taking screenshot: {e}")
```

This function captures a screenshot each time a key is pressed, resulting in a large number of images. Although this approach is demanding on the computer's performance, it provides us with a substantial amount of data to work with.

```
# Capture key press events
def on_press(key):
    global screenshot_count
    try:
        # Lock the key log list to avoid race conditions
        with log_lock:
            # Use str(key) to handle special keys like Shift, Ctrl, etc.
            key_input_log.append((f'screenshot_{screenshot_count}.jpg', str(key)))

        # Take screenshot after logging key press
        take_screenshot(f'screenshot_{screenshot_count}.jpg')

    except Exception as e:
        print(f"Error while processing key press: {e}")
```

In this part, we record each screenshot’s filename along with the corresponding key input into a CSV file.

```
# Write the log to a CSV file periodically
def write_key_log_to_csv():
    try:
        with open('key_log.csv', 'w', newline='') as file:
            writer = csv.writer(file)
            writer.writerow(['Screenshot', 'Key Pressed'])

            # Lock the key log list to safely read from it
            with log_lock:
                writer.writerows(key_input_log)
    except Exception as e:
        print(f"Error writing to CSV: {e}")
```

This code ensures that all keyboard inputs are captured accurately.

```
# Start listening to key inputs
def start_key_listener():
    listener = keyboard.Listener(on_press=on_press)
    listener.start()
    return listener
```

The code saves the file periodically to reduce the load on your computer and improve performance.

```
# Periodically save key logs
def periodic_save(interval=60):
    while not stop_flag:
        time.sleep(interval)
        write_key_log_to_csv()
        print(f"Log saved to CSV at {time.strftime('%Y-%m-%d %H:%M:%S')}")
```

This is the main loop of the screenshot code. It begins by listening for keyboard inputs and runs a separate thread to handle file-saving, allowing the main loop to continue uninterrupted on the primary thread. When the main loop is interrupted, it stops listening to keyboard inputs and writes the final data to the CSV file.

```
# Main loop to capture screenshots and log keys
def main_loop():
    global stop_flag

    listener = start_key_listener()

    # Start periodic saving of the log in a separate thread
    save_thread = threading.Thread(target=periodic_save, args=(10,))
    save_thread.start()

    try:
        while not stop_flag:
            time.sleep(10)  # Main loop can handle additional tasks if needed
    except KeyboardInterrupt:
        stop_flag = True
    finally:
        # Stop the listener and save the log when the program ends
        listener.stop()
        save_thread.join()
        write_key_log_to_csv()
        print("Key logging and screenshots saved!")
```

```
if __name__ == "__main__":
    main_loop()
```

3.2 Train model

To train our model, we used FastAI. We start by ensuring the code can access the data by storing the file path in a variable.

```
from fastai.vision.all import *
import pandas as pd

# Path to the base folder where images are stored
path = Path('../screenshots')

# Load the CSV file into a pandas DataFrame
df = pd.read_csv(path/'key_log.csv')  # CSV with filenames and labels
```

To prevent incorrect predictions, we ensure that the images aren't flipped or rotated during preprocessing, as this could mislead the model. We then set up a `DataBlock` for training, specifying an 80/20 split between training and validation data. After loading the data into the `DataBlock`, we display a preview to verify everything is in order.

```
from fastai.vision.all import *
from fastai.metrics import Precision, Recall
from fastai.vision.augment import *  # Import all augmentations

# Define the DataBlock for loading the dataset
batch_tfms = [
    Normalize.from_stats(*imagenet_stats),  # Normalize using ImageNet stats
    *aug_transforms(do_flip=False, max_lighting=0.2, max_zoom=1.1, min_zoom=0.9, max_warp=0),  # Disable flipping and rotation
    # Add any additional transformations as needed
]
# Define the DataBlock for loading the dataset
block = DataBlock(
    blocks=(ImageBlock, CategoryBlock),  # For image classification
    get_x=ColReader('Screenshot', pref=str(path)+'/'),  # Read file paths from 'Screenshot' column
    get_y=ColReader('Key Pressed'),  # Get labels from 'Key Pressed' column
    splitter=RandomSplitter(valid_pct=0.2, seed=42),  # Split 20% of data for validation
    item_tfms=Resize(224),  # Resize images to 224x224
    batch_tfms=batch_tfms   # Apply custom batch transformations
)


# Load the data
dls = block.dataloaders(df)

# Show a batch of images to verify data loading
dls.show_batch(max_n=9, figsize=(6,6))
```

This is the code for training our model. We used the `resnet50` architecture because it delivered the best results. The initial training phase ran for 5 epochs with a learning rate of 0.1. After this, we "unfroze" the entire model to fine-tune it further, running an additional 3 epochs with a variable learning rate ranging from 0.0001 to 0.00001. Instead of a fixed rate, we used the `fit_one_cycle` approach, which adjusts the learning rate by gradually increasing and then decreasing it within each epoch, optimizing performance.

Finally, we display some graphs to visualize the training process and better understand the model's performance. Once training is complete, we export the model for testing.

```
# Create a learner with a pretrained model (resnet50 in this case)
learn = vision_learner(dls, resnet50, metrics=[accuracy])

# Optional: Use the learning rate finder to select an optimal learning rate
# learn.lr_find()
# Uncomment the following line to visualize the learning rate plot
# learn.recorder.plot_lr_find()

# Fine-tune the model with a specific learning rate
learn.fine_tune(5, base_lr=1e-1)  # You can adjust the number of epochs as needed

# Unfreeze the model and further train it
learn.unfreeze()
learn.fit_one_cycle(3, slice(1e-5, 1e-4))  # Adjust the learning rate range based on results

# Show results including predictions and metrics
learn.show_results()

# Show the confusion matrix and the most confused classes
interp = ClassificationInterpretation.from_learner(learn)
interp.plot_confusion_matrix()
interp.most_confused()

# Export model
learn.export('../models/resnet_fullImagesV2.pkl')
```

3.3 Test model

To test our model, we first load the specific model we want to evaluate and ensure that our screen is focused on TrackMania.

```
import cv2
import numpy as np
from mss import mss
from fastai.vision.all import *
import time
import pydirectinput
import pygetwindow as gw
from pygetwindow import PyGetWindowException
import threading

# Load the model
learn = load_learner('../models/resnet_fullImages.pkl')

# Initialize a shared variable and lock for threading
current_action = None
action_lock = threading.Lock()

# Bring the TrackMania window to the front
def focus_trackmania_window():
    time.sleep(1)  # Short delay before activating
    try:
        window = gw.getWindowsWithTitle('Trackmania')[0]
        window.activate()
    except IndexError:
        print("Could not find TrackMania window.")
    except PyGetWindowException as e:
        print(f"Error focusing window: {e}")
```

In this section, we capture the screen for the model to utilize. We also preprocess the frame and define the possible actions that the model can take.

```
# Function to capture the screen (adjust monitor size accordingly)
def capture_screen():
    monitor = {"top": 390, "left": 460, "width": 1000, "height": 650}  # Modify based on game window position
    sct = mss()
    img = np.array(sct.grab(monitor))
    return img

# Function to preprocess the screen capture for model input
def preprocess_frame(img):
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)  # Convert BGR to RGB
    img = cv2.resize(img, (224, 224))  # Resize to 224x224 if using resnet34
    return PILImage.create(img)  # Convert to a FastAI PILImage

# Define the action mappings
actions = {
    'left': lambda: pydirectinput.keyDown('left'),
    'right': lambda: pydirectinput.keyDown('right'),
    'accelerate': lambda: pydirectinput.keyDown('up'),
    'brake': lambda: pydirectinput.keyDown('down'),
    'stop_left': lambda: pydirectinput.keyUp('left'),
    'stop_right': lambda: pydirectinput.keyUp('right'),
    'stop_accelerate': lambda: pydirectinput.keyUp('up'),
    'stop_brake': lambda: pydirectinput.keyUp('down')
}
```

Here, we map the model's predictions to the corresponding actions that should be taken.

```
# Function to map the model's output to an action
def map_prediction_to_action(pred_class):
    if pred_class == 'Key.left':
        return 'left'
    elif pred_class == 'Key.right':
        return 'right'
    elif pred_class == 'Key.up':
        return 'accelerate'
    elif pred_class == 'Key.down':
        return 'brake'
    else:
        return 'stop_accelerate'  # Default to no acceleration
```

This function handles the actual prediction process. It takes the captured screen input and feeds it into the model, which then predicts an action. We subsequently map this prediction to the corresponding action that the computer needs to perform.

```
# Prediction thread: Continuously captures screen, preprocesses, and updates the current action
def predict_action():
    global current_action
    while True:
        screen = capture_screen()
        processed_img = preprocess_frame(screen)

        # Get the model prediction
        pred_class, _, _ = learn.predict(processed_img)

        # Map the prediction to an action
        action = map_prediction_to_action(pred_class)

        # Update the current action in a thread-safe way
        with action_lock:
            current_action = action
```

This function executes the predicted action. The short delay ensures that the key is held down for 0.02 seconds.

```
# Action execution thread: Executes the action based on the latest prediction
def execute_action():
    while True:
        # Access the predicted action
        with action_lock:
            action = current_action

        # Execute the current action
        if action:
            actions[action]()

            # Release the key after a short delay for smooth transition
            time.sleep(0.02)
            pydirectinput.keyUp(action)
```

This is the main loop that allows the model to drive. We utilize separate threads for predicting and executing actions, which significantly speeds up the loop and results in smoother driving performance.

```
# Main function to start autonomous driving
def autonomous_drive(learn):
    focus_trackmania_window()

    # Start the threads for prediction and action execution
    predict_thread = threading.Thread(target=predict_action, daemon=True)
    action_thread = threading.Thread(target=execute_action, daemon=True)

    predict_thread.start()
    action_thread.start()

    # Keep the main program running
    while True:
        time.sleep(1)

# Start the autonomous driving
autonomous_drive(learn)
```

4. Obstacles

4.1 Prediction Rate

One of the main challenges we faced was that our model's predictions were too slow to keep up with the speed of the car. We implemented several optimizations to improve the prediction rate. First, we removed all print statements from the code to reduce processing time. We also minimized the use of time.sleep() to avoid unnecessary pauses in execution. Additionally, we tried to make the car drive slower to allow the model more time for each prediction. Finally, we introduced multithreading, enabling the model to handle predictions and actions simultaneously. These adjustments helped us achieve faster, more responsive predictions.

4.2 Racing Line

When capturing screenshots for our dataset, we initially followed a racing line. During testing, we noticed that the AI also attempted to follow this line, which increased the risk of driving into the wall. To address this, we added more images of the car driving in the center of the track, where the risk of hitting the wall is lower. This adjustment helped the model learn to navigate more safely.

4.3 Flipping & Rotating

While training our model, we initially used Fastai's standard data augmentation tools. Later, we realized that flipping and rotation were included in these augmentations. This caused an issue, as flipping or rotating the images made the labels incorrect. To address this, we created a custom batch transformation that excluded both flipping and rotation functions.

5. GenAI

We used generative AI for multiple purposes in our project. First, we used it to review this document to ensure there are no grammatical mistakes. We also leveraged generative AI to generate basic code, which we then adapted to fit our specific needs. Additionally, if any errors arose in the code, we used AI to help review and potentially resolve them. Lastly, we used generative AI to identify possible improvements for our project and explore ways to enhance its overall quality.

6. Conclusion

Overall, the project was both educational and enjoyable. We believe it is a valuable initiative worth continuing in the future. Initially, we underestimated the complexity of developing the AI, as we thought it would be easier than it turned out to be.

What would we do differently in the future?

First, we would implement multiple key inputs to improve the AI's learning potential based on the images. Additionally, we would conduct more thorough research into the available resources and tools before starting the project, ensuring a smoother and more effective launch.