COMPUTER VISION CHALLENGE

Deep Learning PE Assignment 1

Emalisa Antonioli, Jeffrey Stynen, Emma Schoofs

3 November 2024

Contents

1 Introduction **3**

2 Data Collection **3**

3 Data Augmentation **3**

4 Training ResNet **5**

4.1 Classification Model 5

4.2 Regression Model 5

5 Results ResNet **6**

6 Driving the Car **7**

7 Object Detection **8**

8 Conclusion **9**

A Generative AI **10**

A.1 Emalisa Antonioli 10

A.2 Emma Schoofs 13

A.3 Jeffrey Synen 13

1 Introduction

In this document, we present a comprehensive guide to building a self-driving car model for *Forza Horizon 4*. This project explores our approach to addressing the technical and conceptual challenges of developing an autonomous driving system within a virtual gaming environment. We outline our end-to-end process, from data collection to implementing a functional model capable of navigating roads in the game.

Beyond the core self-driving functionality, we also developed an object detection model using YOLO (You Only Look Once) to identify in-game elements like other vehicles and obstacles. Due to hardware limitations, running both models simultaneously was beyond the scope of this project, so the object detection model remains a separate component.

By sharing our thought process, key findings, and methods for tackling encountered issues, we hope this document provides valuable insights and practical guidance for others interested in similar projects.

2 Data Collection

To collect our data, we used a custom script. This script identifies the region of the screen where the game is displayed and captures a screenshot of that area every 0.5 seconds. We immediately crop the top 30% of each image, removing redundant elements like the sky and trees. This early augmentation reduces the potential for confusion in the model by focusing on relevant information only.

After capturing each game state (image), we also recorded the corresponding action (controller input). Initially, we attempted to log these inputs in a separate .csv file, with columns for the timestamp, left joystick X value (LX, steering angle), right trigger (RT, gas input), and left trigger (LT, brake value). To link each image (state) with its respective action, we included a timestamp in each filename. However, this approach led to significant complexity and confusion, as combining the separate data sources for model training proved cumbersome, especially for manual data cleaning.

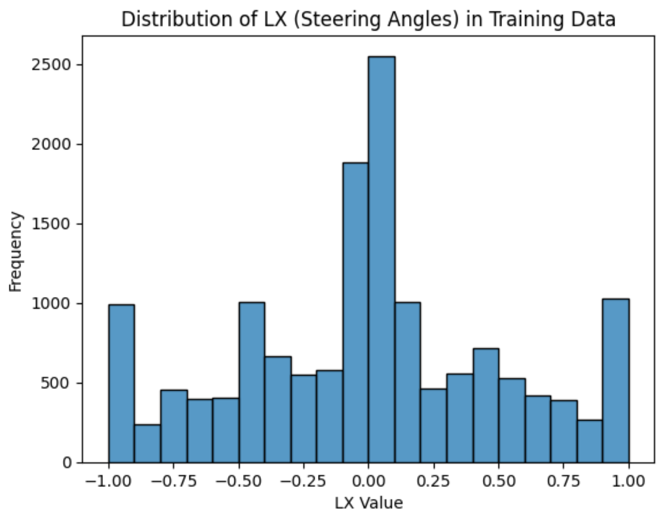
To simplify, we adopted a new approach: embedding the action values directly within each state's filename. This ensured that each state-action pair remained linked, streamlining data cleaning and making it easier to retrieve actions (labels) for the model with a straightforward regex expression.

```
r'(?P<timestamp>\d{8}\d{6})(?P<LX>[-0-9.]+)(?P<LT>[-0-9.]+)(?P<RT>[-0-9.]+).jpg$'
```

3 Data Augmentation

To improve model performance, we applied several augmentation techniques in addition to cropping the top 30% of the image. These techniques include rotating the image by 10 degrees, zooming by 10%, warping by 20%, adjusting brightness by 20%, and disabling horizontal flipping. However, applying only these augmentations during training resulted in poor generalization on the road surface, causing the car to veer off the road. Initially, we assumed this was due to inadequate generalization. However, we discovered the actual cause: an imbalanced dataset in terms of steering angles.

Analyzing the original distribution of steering angles, we observed significantly fewer instances of high steering angles (over 75%) in either direction compared to instances of no steering. This imbalance likely limited the model's ability to confidently navigate turns.

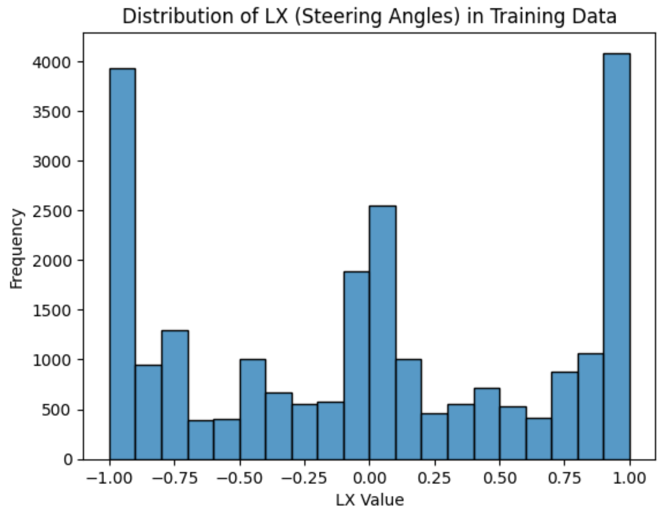


To address this imbalance, we duplicated all images where the steering angle exceeded 75%. However, this required us to adjust the timestamp embedded in each filename, which proved challenging due to the complexity of handling regular expressions. Our solution involved extracting the timestamp from the filename, adding 1000 to it, formatting it correctly, and reconstructing the filename.

The code below illustrates this process:

```
for index, row in filtered_df.iterrows():
    source_file = source_dir / row['filename']

    timestamp, LX, LT, RT = extract_controller_values(row['filename'])
    timestamp = timestamp + 1000
    timestamp = str(timestamp)
    timestamp = f"{timestamp[:8]}_{timestamp[8:]}"
    destination_filename = f'{timestamp}_{round(LX,4)}_{round(LT,4)}_{
round(RT,4)}.jpg'

    destination_file = output_dir / destination_filename

    shutil.copy(source_file, destination_file)
```

By running this duplication process several times, we achieved a more balanced —technically, an intentionally unbalanced distribution with a bias toward extreme steering angles—distribution, giving the model greater confidence in navigating turns. This ultimately enhanced the model's ability to generalize to the road.



4 Training ResNet

4.1 Classification Model

We started by creating a classification model that could decide whether to go right, left, ahead or break based on images it is fed. To do this we divided the data into four folders, each representing one of the labels, as well as split the data into a train set and a validation set. for data augmentation we kept simple by only resizing them and performing aug_transforms(), which applies some random augmentations. We then fine tuned a pretrained ResNet34 model for 10 epochs. See classification_model.ipynb for all the code. This model was able to make predictions, but at this point we decided that we wanted to use joysticks instead, which would require a regression model. So, here we abandon the classification model and switch to the regression model.

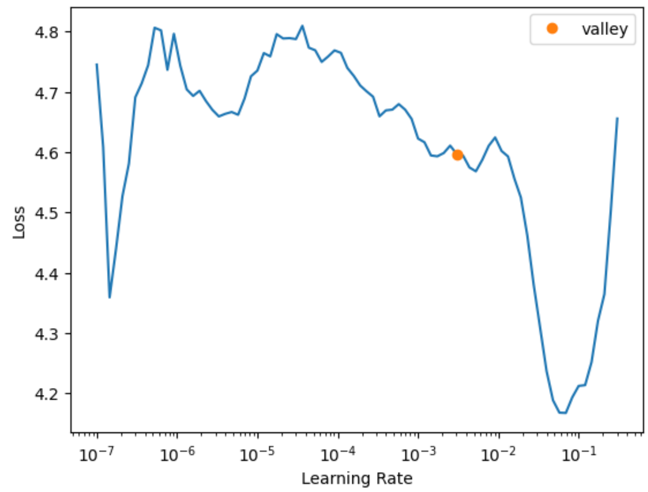
4.2 Regression Model

Our final CNN was based on a pretrained ResNet50. We used mean square error as our loss function.

```
learner = cnn_learner(dls, resnet50, loss_func=MSELossFlat(), metrics=[
mse])
```

We also used 16-bit floating point precision to reduce memory usage.

We ended up training our model with multiple rounds of fit_one_cycle of either 5 or 10 epochs. We chose this because it applies the concept of simulated annealing which will help us prevent getting stuck on local minima. For each of these cycles we set a max learning rate. We chose this learning rate by performing a lr_find, which gives a result such as the image below. We then use the suggested learning rate as the max learning rate. In the example below that was 3e-3.



Each learning cycle we looked at the train and validation loss, as well as the mean square error. Once we seemed to have reached a plateau, we unfroze the pretrained layers and ended off with some other rounds of fit_one_cycle. We did this until we felt the train and validation losses were converged. See model.ipynb for all of the code.

We got this result via trial and error. Something we experimented with was batch size: we now use a batch size of 64. Using smaller batch sizes gave us poorer results, using larger ones became too computationally taxing.

We also tried training more or less epochs in between changing the maximum learning rate.

It is important to note that ResNet automatically uses batch normalization to prevent covariate shift between the different layers. It also uses ReLu activation functions which helps prevent vanishing gradients.

5 Results ResNet

After training the model up to the point of train and validation loss convergence, we could let the model play the game. See Driving the Car for more details on how the model was able to do this. Here, let's talk about the model's driving behavior. The model is able to keep the car on the road, it is clear that it is able to make micro decisions that help him stay on the road while going straight ahead or taking turns. It is able to do this at both low and high speeds. What it cannot do is avoid objects, such as other cars on the road. Whenever it crashes into another car it is not able to correct itself and usually stops driving until started again. We suspect that this is because we had made the decision to remove data of crashes, the idea being that we do not want it to learn the behavior that results in crashes. We now realize that this means that there is also little data available in regard to recovering from a crash. In a future iteration of this project we would start by collecting more data to help the model recover from crashes, as well as combine its driving behavior with object detection to prevent crashes.

6 Driving the Car

To test our model, we needed to enable it to control the car in-game. First, we ensured the model could "see" the game by feeding it unlabeled images, allowing it to receive real-time visual input. Fortunately, we could reuse the screen capture class created earlier, applying the same pre-processing step of cropping the top 30% of each image to eliminate irrelevant information. The model then outputs controller values corresponding to each image, but directly using these values introduces some issues.

Initially, the controller values in our labels ranged from -1 to 1, while the virtual controller library requires a scale from 0 to 255. We also rounded these values to three decimal places for simplicity. This conversion can be achieved with the following code:

```
steering_angle = round(int((lx + 1) * 127), 3)
brake_value = round(int(((lt + 1) / 2) * 255), 3)
gas_value = round(int(((rt + 1) / 2) * 255), 3)
```

With the values properly scaled, we could pass them to the virtual controller to let the model drive. However, this initially caused the car to stall frequently. The model would predict non-zero values for both throttle and brake, resulting in simultaneous throttle and braking and causing the engine to rev in place. To address this, we implemented a priority system that checks the throttle and brake values and passes only the highest of the two to the controller, setting the other to zero.

The following code demonstrates this priority-based control:

```
priority = max(brake_value, gas_value)

if priority == brake_value:
     self.gamepad.left_trigger(brake_value)
elif priority == gas_value:
    gas_value = int(gas_value)
    self.gamepad.right_trigger(gas_value)

self.gamepad.left_joystick(steering_angle, 128)

self.gamepad.update()
```

Notice that we specify a second parameter (128) for the left joystick, which represents the Y-axis. This Y-axis needs to remain centered at all times to ensure the model focuses on the road.

While testing this code, we encountered two primary issues. First, the virtual controller occasionally failed to be destroyed after a session, causing the car to continue moving indefinitely in its last direction, making it difficult to stop. We resolved this by creating a function using the built-in reset method, which returns all controller inputs to their neutral state:

```
def reset(self):
    self.gamepad.reset()
    self.gamepad.update()
```

The second issue was an intermittent problem where the virtual controller was only created about half of the time, with no clear fix despite multiple attempts. This means the code only functions reliably on every other run, which remains a frustrating limitation.

7 Object Detection

As an extra, we added object detection to our project. Here we used the YOLOv11 model. To make it work in a game we were inspired by the script of Moises de Paulo Dias[1]. With this model, we put in real-time bounding boxes around the objects to be detected along with a label and the confidence of the model. To accomplish this we used 3 major steps.

First, we loaded the model and set the confidence threshold at 0.5 as well as so for the intersection over union(IoU). The second step is to retrieve the screen coordinates of the game window. This way the model knows where to perform object detection. The last step is to detect and annotate the objects. After obtaining the coordinates, the function converts this frame to a format suitable for OpenCV processing. Because OpenCV uses the BRG color order, the frames look a little weird. The model then performs object detection based on the converted frame. We modified the model so that it will only detect cars, buses and trucks of which it is at least 50% sure. We also set the Non-Max Surpression to 0.7.

```
    detections = sv.Detections.from_ultralytics(results[0]).with_nms(
    threshold=0.7)
    selected_classes = [2, 5, 7] #detect only cars, busses and trucks
    detections = detections[np.isin(detections.class_id, selected_classes)]
    detections = detections[detections.confidence > 0.5]
```

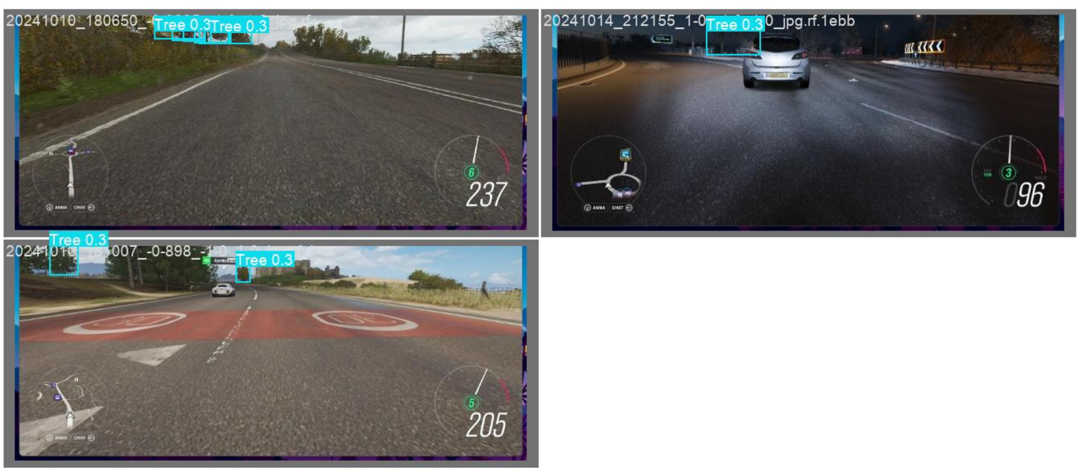
To realize all this continuously we used a loop so that we get continuous real-time Object Detections.

```
game_title = "Forza Horizon 4"
while True:
    screen_capture = ScreenCapture()

    if screen_capture.game_area:
        screen = screen_capture.capture_frame()
        detected_frame = detect_objects(screen)

        # Display the detection
        cv2.imshow("Game Object Detection", detected_frame)


    # Exit on 'q' key
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cv2.destroyAllWindows()
```

This is the result:

[1]Moises de Paulo Dias, author = moises-dias, title = Repository: yolo-opencv-detector, url = https://github.com/moises-dias/yolo-opencv-detector, note = Consulted on October 20, 2024

Furthermore, we also tried to create our own object detection model that could detect houses and trees. This model was trained on 14 in-game images that I had available. Nevertheless, the model is reasonably good at recognizing trees. To train the model, we used Roboflow to apply labels to the images. Then we had the YOLOv11 model trained on these images. This is the result:



Unfortunately, we were not able to implement this model in our project.

8 Conclusion

In conclusion, this project offered an in-depth learning experience in designing and refining a self-driving car model within a game. From data collection and augmentation to model training and testing, each phase of the project provided valuable insights into the challenges and nuances of applying deep learning to autonomous driving. We experimented with different approaches to data handling, such as embedding controller inputs directly in filenames to streamline the data-labeling process.

Through experimentation, we enhanced our understanding of model limitations and how to address them. For example, balancing the dataset by duplicating images with high steering angles helped the model handle turns, an insight that deepened our understanding of data imbalance's impact on model generalization. Additionally, transitioning from a classification to a regression model allowed us to grasp the advantages of continuous control in steering prediction, aligning the model's outputs with real-time joystick movements for smoother in-game performance.

Implementing YOLO for object detection further expanded our knowledge, as we applied this model to recognize in-game vehicles.

This project not only advanced our technical skills but also reinforced critical problem-solving techniques.

A Generative AI

A.1 Emalisa Antonioli

I used Chat GPT as my generative AI tool. I mostly asked it for further explanation about the things taught in class or about the code that I found online or about code that my teammates wrote if they were not available for explanation. I also liked to use Chat GPT to generate code that I could use as a starting point if I could not easily find some online. The below question and answer combo is an example of this.

```
I am creating a DL to drive on Forza 4. I am in the data collection stage. I
    am playing with a ps4 controller on a pc via steam, how can i best collect
     the input from the controller?
```

```
To collect input from your PS4 controller for training a deep learning (DL)
    model to drive in Forza Horizon 4, you'll want to capture the control
    signals (e.g., steering, throttle, brake, etc.) while you play the game.
     Heres a guide to help you collect the input data from the PS4
    controller:
1. Use a Controller Input Monitoring Tool

You'll need a tool that can capture input from your controller and log it for
     later use. The controller typically sends data like:

    Left joystick (for steering)
    Right joystick (camera, or acceleration/braking depending on
    configuration)
    Triggers (usually acceleration and braking)
    Buttons (e.g., gear shifting or handbrake)

Here are some ways to do that:
Option 1: Use Steams API for Controller Input

Since you are playing via Steam, the platform natively supports PS4
    controllers. You can use Steams Controller Input API to capture data.
    The Steam API will allow you to collect real-time input data from the
    controller as you play.

You can write a script in Python or another language that interfaces with
     Steams API to capture the inputs from the controller. You'll need to:

    Install the Steamworks SDK (which includes the controller input API).
    Write a script that logs the controller inputs in real-time while playing
    .
```

```
Note: This might require some additional configuration or code to run in
    parallel with Forza Horizon 4 while it records the inputs.
Option 2: Use an External Input Logging Tool

There are several tools that log controller inputs outside of Steam. These
    tools capture the input data in real-time and log it into files that you
    can use for training your DL model. Some tools that might help are:

    DS4Windows: This utility allows you to interface with the PS4 controller
    on Windows, and can provide detailed input data (axis and button presses).
     You may need to modify or extend it to log inputs in a format suitable
    for your dataset.
        Link DS4Windows to an external logger that records the inputs.

    JoyToKey: Another option to map controller inputs to keyboard/mouse
    inputs and log the actions.

    Game Controller Tester: This is a lightweight option that lets you view
    real-time controller inputs. However, you would need to script something
    to log the inputs.

2. Write a Data Logging Script

 You ll  need a script to capture and log the input data in real-time. For
    instance, if you are using Python, you can use libraries like pygame or
    inputs to capture joystick and button data from your PS4 controller.

 Heres  a basic outline in Python to capture input data:
Example: Python Script Using inputs Library

python

import inputs
import csv
import time

def log_controller_data():
    # Open a CSV file to log inputs
    with open('controller_log.csv', mode='w') as file:
        writer = csv.writer(file)
        writer.writerow(["Time", "Event Type", "Event Code", "Value"])

        print("Starting data collection... Press Ctrl+C to stop.")

        try:
            while True:
                events = inputs.get_gamepad()
                for event in events:
                    timestamp = time.time()
                    writer.writerow([timestamp, event.ev_type, event.code,
    event.state])
                    print(f"Logged: {timestamp} {event.ev_type} {event.code}
    {event.state}")

        except KeyboardInterrupt:
            print("Data collection stopped.")
```

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

This script captures:

    Timestamp
    Event type (e.g., button, axis)
    Event code (e.g., which button or axis)
    Event value (e.g., the amount of pressure or axis movement)

You can then synchronize these inputs with your game state or screen captures
     to create a labeled dataset.
3. Synchronize Game Data with Controller Input

To train your DL model effectively,  y o u ll  need to pair the controller
    inputs with the corresponding game state (like the  c a r s  position, speed
    , and road conditions). You can:

    Use screen recording to capture the visual game state.
    Capture additional telemetry data from the game (if available).

Tools like OBS Studio can record your screen, and there are telemetry tools
    available for Forza Horizon 4 that can capture game state data, such as
    speed, RPM, gear, etc. Ideally, you want to log the telemetry data and
    controller inputs in sync to feed into your model.
4. Log Format

Your input log file might look like this:

scss

Timestamp, Event Type, Event Code, Value
1634325505.12345, Absolute, ABS_X, 1256   # Left Stick X (steering)
1634325505.12355, Absolute, ABS_Y, 3020   # Right Stick Y (throttle)
1634325505.12365, Key, BTN_SOUTH, 1       # Button Press (e.g., handbrake)

    Event types: "Absolute" (joystick) or "Key" (button).
    Event codes: Specific axis or button (e.g., ABS_X for left joystick X-
    axis).
    Value: The raw value from the controller (e.g., 0-65535 for analog sticks
    ).

5. Visualize or Analyze Collected Data

Once you have your controller input data:

    Preprocess it to filter noise and synchronize it with game footage.
    Visualize the inputs (e.g., throttle vs steering) to better understand
    your driving behavior.
    Combine the data with the game state for training your DL model.

Summary

To capture inputs from a PS4 controller while playing Forza Horizon 4:

    Use an input logging tool like DS4Windows or a custom Python script.
```

```
    Synchronize controller inputs with game telemetry and screen recordings.
    Store data in a structured format (e.g., CSV) for training your DL model.

Let me know if you need more specific details for any of the steps!
```

A.2 Emma Schoofs

I asked Chat GPT to transform Moises de Paulo Dias' code so that it uses YOLOv11.

```python
import torch

class ImageProcessor:
    def __init__(self, window_size, model_path):
        # Load YOLOv11 model from a given path
        self.model = torch.hub.load('ultralytics/yolov11', 'custom', path=model_path)  # Y
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self.model.to(self.device).eval()  # Put the model in evaluation mode

    def process_image(self, img):
        # Preprocess the image for YOLOv11
        img = cv.cvtColor(img, cv.COLOR_BGR2RGB)
        img_tensor = torch.from_numpy(img).float().permute(2, 0, 1) / 255.0
        img_tensor = img_tensor.unsqueeze(0).to(self.device)

        # Perform inference
        with torch.no_grad():
            detections = self.model(img_tensor)

        # Extract bounding boxes, confidence, and class predictions
        results = detections.xyxy[0].cpu().numpy()
        coordinates = []
        for x1, y1, x2, y2, conf, cls in results:
            coordinates.append({
                'x': int(x1),
                'y': int(y1),
                'w': int(x2 - x1),
                'h': int(y2 - y1),
                'class': int(cls),
                'confidence': float(conf),
                'class_name': self.model.names[int(cls)]
            })
        return coordinates
```

```python
window_name = "YOUR_GAME_WINDOW_NAME_HERE"
model_path = "yolov11_custom.pt"  # Replace with your YOLOv11 model path

wincap = WindowCapture(window_name)
improc = ImageProcessor(wincap.get_window_size(), model_path)

while True:
    ss = wincap.get_screenshot()

    if cv.waitKey(1) == ord('q'):
        cv.destroyAllWindows()
        break

    coordinates = improc.process_image(ss)

    for coordinate in coordinates:
        print(coordinate)
    print()
```

A.3 Jeffrey Synen

I primarily used generative AI to help solve errors. I usually did this by sending Chat GPT my error message and any relevant code, after which I would talk back and forth until my error was solved.