Deep Learning project
Computer vision report

Rob Verbeek
Matej Buršík
Zubair Hussain

Table of Contents

Goal

The goal of this project is to develop a basic self-driving car for the game *TrackMania* using computer vision techniques. Using a supervised learning approach, the objective is to create an agent capable of interpreting the in-game environment and making driving decisions based on visual input alone. The focus is on building a straightforward model that can handle basic track navigation, such as following road boundaries and managing simple turns, with moderate consistency.

To achieve the goal of creating a self-driving car in *TrackMania* with computer vision, the following steps should be undertaken:

1. **Data Collection**: Capture gameplay data, including images from the car's point of view and corresponding driving actions (steering, throttle, and brake inputs). This dataset will be essential for training the model to recognize visual cues and associate them with appropriate driving actions.
2. **Data Preprocessing**: Clean and preprocess the collected data by resizing images, normalizing values, and possibly augmenting the dataset to improve the model's ability to generalize across different lighting conditions or slight variations in track appearance.
3. **Model Selection and Training**: Select a suitable deep learning model that can interpret visual data and predict driving actions. Train the model using the pre-processed dataset, ensuring it learns to associate visual input with basic driving controls.
4. **Testing and Evaluation**: Test the model on various *TrackMania* tracks to assess its performance in terms of stability, accuracy, and consistency. Evaluate the model's ability to navigate basic track layouts and make simple turns, adjusting hyperparameters or retraining as needed to improve performance.
5. **Fine-Tuning and Optimization**: Make final adjustments to the model by fine-tuning based on test results. This might include refining the model architecture or modifying the training data to improve the agent's ability to handle specific track challenges or reduce error in its driving actions.
6. **Integration and Demonstration**: Integrate the trained model into the *TrackMania* environment and run demonstrations to showcase its capabilities, highlighting areas where it performs well and identifying potential areas for future enhancement.

As extras, to detect the car within gameplay images, bounding boxes will be created using the YOLO (You Only Look Once) object detection framework. YOLO enables real-time detection with high speed and accuracy. For this project, images will be annotated to define bounding boxes around the car, capturing its position on the track.

An LSTM (Long Short-Term Memory) network will also be tested to improve the driving model's ability to make sequential decisions based on past frames. LSTMs are well-suited for sequence-based data, allowing the model to consider previous actions when predicting future ones. By training the LSTM on sequences of visual inputs and driving actions, the model should better anticipate turns and obstacles, enhancing the agent's navigation in *TrackMania*.

This document describes our journey towards the result.

Data Collection

This section functions as the data collection chapter of our document, describing the methods we used to gather essential data from the game *TrackMania* for developing a self-driving neural network. The goal of these data collectors is to capture in-game screenshots while recording control inputs, creating a comprehensive dataset that reflects real-time driving conditions and player actions. By correlating visual inputs from the game with corresponding driving commands, the collected frames provide labelled training data, allowing the model to learn to navigate the track autonomously based on visual cues.

Collector 1

Collector 1 was our initial approach for data collection. We used the `pyautogui` library to capture screenshots. Then we converted each PIL (Python Imaging Library / Pillow) image into a numpy array to simplify preprocessing. In this version, we didn't perform extensive preprocessing. We primarily cropped the image borders and colour-corrected from RGB to BGR.

While capturing screenshots, another part of the code recorded player inputs using the `keyboard` library, which tracked the keystrokes for up, down, left, and right. These keystrokes were stored in the variables `steering`, `throttle`, and `brake`. For steering, left was represented in the range from -1 to 0, while right was represented in the range from 0 to 1.

We decided to store all of this information in JSON format, but the files became extremely large, even with just 100 images. To address this, we converted the images to base64 to compress them, which helped but was still not sufficient. We exported the data to both a JSON file and an MP4 file to facilitate easy review.

Unfortunately, this collector was partially lost due to versioning issues caused by miscommunication. Fortunately, this happened after we had already transitioned to newer versions of the collector.

Collector 2

The main changes in this version are the method of data storage and improvements in image preprocessing.

We realized that storing data in a JSON file was inefficient, so we switched to saving each image in a folder, with the corresponding input data stored in a CSV file. For image preprocessing, we downscaled the images since the neural network doesn't require high resolution. This reduces memory usage and decreases computation time during training, as the smaller image size produces fewer nodes per layer in the neural network. We also cropped out more of the image, focusing on the road ahead and removing unnecessary elements like the sky, trees, billboards, and the road behind the car.

Collector 3

The main changes in this version involve adjustments to the variable storing captured inputs and modifications in data preprocessing.

We hypothesized that the neural network might have struggled with the steering variable, as left steering was represented by a range from -1 to 0. We suspected that the negative values might be interpreted as unfavourable by the model, as it consistently steered in only one direction. To address this, we redefined the steering inputs to individual values for left, right, forward, and backward. In data preprocessing, we opted for a fixed resizing of 400x225 pixels, whereas previously we were downscaling the images by 50%. Additionally, we refined the cropping of the images to improve focus on relevant areas.

Data Manipulation

This section serves as the data manipulation chapter of our document, detailing the processes of data cleaning and combining datasets as we prepared the collected data from *Trackmania* for training a self-driving neural network.

Data Cleaning

We experimented with several approaches to clean the data. Initially, we manually reviewed each image and edited the `controls.csv` file directly, but this method proved inefficient. To streamline the process, we developed `cleaner.py`, a script that displayed each image along with the corresponding player inputs on screen. Users could press 's' to save the image or any other key to exclude it from the new `controls.csv`. However, we found that most images were still incorrectly labelled and the process remained slow and labour intensive.

This led us to adopt a combined approach: we manually reviewed the images, but used a script, `control_rewrite.py`, to automatically generate a new `controls.csv` file. This script uses an `inputs` variable to create a DataFrame, which is then saved as a CSV file. The `inputs` variable is a list of lists specifying the range of images to include, the associated player input, and a multiplier to control the number of duplicates for each image range. This multiplier allows us to expand the dataset without needing to capture additional images. This script can be found in every data folder that was cleaned with this method.

```
inputs = [
    [[0, 40], [1, 0, 1, 0], 2],
    [[55, 59], [0, 1, 1, 0], 1]
]
controls_df = pd.DataFrame(columns=['frame', 'left', 'right', 'forward', 'backward'])

for nums,inp,mult in inputs:
    for _ in range(mult):
        for i in range(nums[0], nums[1]+1):
            new_df = pd.DataFrame({
                'frame': [f'frame_{i}.jpg'],
                'left' : inp[0],
                'right' : inp[1],
                'forward' : inp[2],
                'backward' : inp[3]
            })
            controls_df = pd.concat([controls_df, new_df], ignore_index=True)

print(controls_df)
controls_df.to_csv("controls.csv", index=False)
```

Combining Datasets

To maintain organization, we opted to create separate folders based on specific criteria, such as tracks and movement direction. This structure resulted in the images gathered from the game sharing the same names, with each folder containing its own CSV file for the corresponding key inputs. To effectively utilize this data for our model, we needed to compile a combined dataframe that integrated all the data from the various folders. This approach ensured that images with identical names could be included without causing confusion for the model or disrupting the learning process.

```python
# Initialize an empty DataFrame to store all the data
combined_df = pd.DataFrame()

# Loop through each folder and load CSVs and image paths
for folder in sources:
    folder_path = base_data_path/folder

    # Load the CSV file
    csv_file = folder_path/'controls.csv'
    if csv_file.exists():
        df = pd.read_csv(csv_file)

        # Get all image files that exist in the folder
        image_files = [f for f in os.listdir(folder_path) if f.endswith('.jpg')]

        # Dynamically match each row with an actual image filename
        def get_image_filename(filename):
            # Only return the filename if it exists in the folder
            if filename in image_files:
                return folder_path/filename
            else:
                return None

        # Apply the function to create a new column with actual image paths
        df['frame'] = df['frame'].map(get_image_filename)

        # Filter out rows where no corresponding image file was found
        df = df[df['frame'].notnull()]

        # Combine with the main DataFrame
        combined_df = pd.concat([combined_df, df], ignore_index=True)
```

The model

In this chapter, we discuss the process of selecting and optimizing a neural network model to drive our self-driving car in Trackmania. This involves choosing a pretrained architecture that can accurately interpret the game's complex visual scenes, along with applying advanced techniques to enhance the model's accuracy and efficiency. Our approach combines the strengths of ResNet with various optimization methods, including half-precision training, dropout, learning rate adjustments, and adaptive learning schedules. Together, these techniques aim to improve the model's ability to generalize and perform reliably in the *Trackmania* environment.

Model selection

Since we are using a pretrained model, we have access to various options to experiment with. Initially, we started with ResNet-34 but later switched to ResNet-50 to capture more detail in our images, hoping this would lead to improved results. ResNet was a good choice because of its proven effectiveness in image recognition tasks, especially in handling complex visual features. Its residual connections allow it to achieve high accuracy without suffering from the vanishing gradient problem, making it well-suited for learning from the diverse and detailed images in *Trackmania*.

However, even with ResNet's advantages, challenges remain because the *Trackmania* car is not a typical or familiar object within standard pretrained models. Unlike ordinary objects, the car's rapid motion and unusual perspectives make it more challenging to track consistently. This means that while ResNet offers a robust starting point, additional fine-tuning or customization is still needed for optimal performance.

FP16 – half precision training

To address slow training times, we implemented FP16, which uses 16-bit floating-point precision instead of the usual 32-bit (FP32). Although FP16 can reduce some detail, *Trackmania's* relatively simple graphics make this trade-off minimal, allowing us to cut training time significantly without impacting model performance.

By using half-precision, FP16 reduces memory usage and increases computational speed, allowing for larger batch sizes and improved model efficiency. This is particularly beneficial when working with limited GPU memory. Fastai makes FP16 integration simple with its .to_fp16() method on the learner object.

Our experience showed that FP16 led to a significant reduction in model training time without sacrificing accuracy.

Learning rate finder

The learning rate finder is a technique to identify the optimal learning rate for training by gradually increasing it and observing the model's loss. It provides an efficient way to choose the best learning rate, which can significantly improve convergence and prevent unstable training. This is especially helpful for new datasets like game images, where an appropriate learning rate might be less intuitive. Fastai's `.lr_find()` function lets users plot loss against varying learning rates, providing an easy way to determine the ideal learning rate.

One fit cycle

This learning rate policy adjusts the learning rate throughout the training cycle, starting low, increasing to a peak, and then decaying. This helps models converge faster and often results in better accuracy, as the one-cycle schedule helps avoid local minima and smooths out the training process. It's particularly useful in projects with limited training time, as it optimizes learning rate adjustments dynamically.

Discriminative Learning

Discriminative learning involves using different learning rates for different layers of a model, lower for earlier layers (which often contain more general features) and higher for later layers (focused on task-specific features). This approach helps fine-tune the model more effectively, allowing earlier layers to retain general learned features while optimizing the layers specific to the self-driving task.

To use this with fast-ai we make use of the `lr_max=slice()` parameter which lets us specify a range of learning rates that should be used. We start with a super low learning rate for the first layers and increase the learning rate for the later layers.

With discriminative learning we also used the learning rate finder to further adjust the optimal learning rate between the learning cycles.

In our experience using discriminative learning completely confused the model and massively increased the losses of both training and validation from the first epoch onwards. During the testing of the model it also showed that the car has no sense of direction anymore and is essentially randomly choosing what inputs to press.

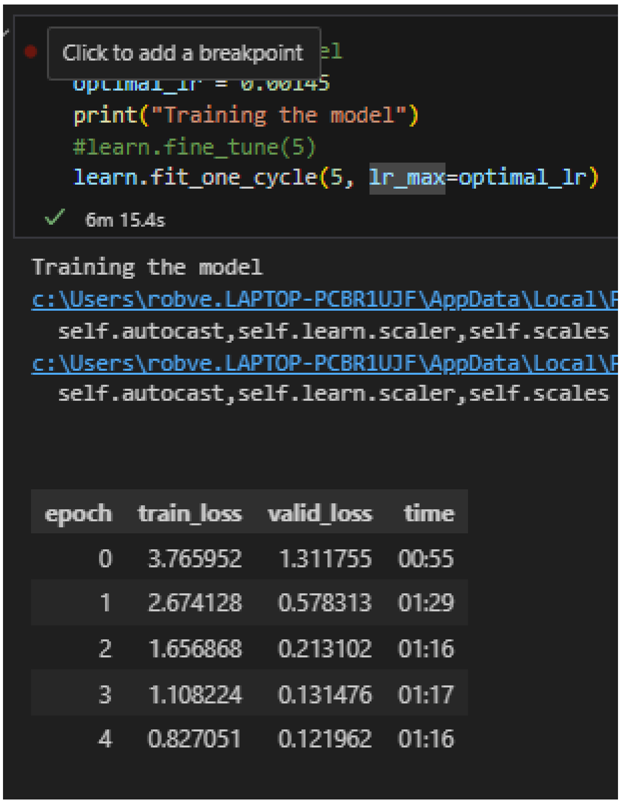

Figure 1: Losses before discriminative learning

```
second_lr = 0.00208
learn.unfreeze()
learn.fit_one_cycle(5, lr_max=slice(0.00001-second_lr))
```

[11] ✓ 12m 29.1s

```
c:\Users\robve.LAPTOP-PCBR1UJF\AppData\Local\Programs\Python\
  self.autocast,self.learn.scaler,self.scales = autocast(dtyp
c:\Users\robve.LAPTOP-PCBR1UJF\AppData\Local\Programs\Python\
  self.autocast,self.learn.scaler,self.scales = autocast(dtyp
```

epoch	train_loss	valid_loss	time
0	54.851154	735.716858	02:27
1	2588.121094	7537.605957	02:29
2	13106.689453	28543.986328	02:29
3	27456.910156	38253.597656	02:30
4	37279.890625	40128.843750	02:32

Figure 21: Losses after discriminative learning

Custom learner

We attempted to develop a custom model by constructing a ResNet body and integrating our own head that employed the Tanh function. This was intended to enhance our model’s ability to predict negative values, as we aimed to forecast values ranging from -1 to 1 for left and right steering. However, this approach was ultimately abandoned because it introduced further confusion for the model, hindering its performance rather than improving it.

It's possible that trying more variations on the custom learner would have been better than the pretrained model, however this took a lot of time that we rather spent on different parts of the project.

```
nfeatures = 512 # resnet34 has 512 features before the head
features_out = 3 # throttle,steering,brake
body = create_body(resnet34(), pretrained=True)

head = create_head(nfeatures, features_out)

model = nn.Sequential(body,head, nn.Tanh())
```

Dropout

To explore ways of improving model generalization, we experimented with a custom model that incorporated dropout as a regularization technique to reduce overfitting. Given that our dataset includes duplicated images to increase its size, dropout seemed promising to prevent the model from simply memorizing specific patterns. By randomly “dropping out” neurons during training, dropout aims to avoid reliance on “neuron superstars,” helping the model focus on learning broader features.

However, the custom model with dropout actually performed worse than the original pretrained model. Given the added complexity and reduced performance, we decided to continue with the pretrained model, which provided better results without requiring additional regularization.

Integration with *Trackmania*

In this chapter, we explore the integration of a self-driving car in *Trackmania* to test and evaluate the performance of our developed model. To achieve this, we use two separate files: one containing all the essential functions and another, `main.py`, which runs the code that enables the car to drive autonomously. This structure keeps the code organized and supports the effective implementation of our driving agent within the game.

Main

The `main.py` file contains the main body of the self-driving application. It runs a continuous `while` loop, allowing you to start and stop the self-driving feature. When the application turns on, it prints "ON" in the terminal. Pressing 'q' starts the self-driving feature, and pressing and holding 'e' stops it. To terminate the application, press and hold 't'. After termination, the application prints "OFF" in the terminal.

```
print("ON")

title = "Trackmania"
run = False

while True:
    if keyboard.is_pressed('q'): # start driving by 'q'
        run = True
        print("Started AutoMania ...")

    while run:
        if keyboard.is_pressed('e'): # end driving by 'e'
            run = False
        screen = take_a_screenshot(title)
        result = decision(screen)
        process_result(result, True)

    if keyboard.is_pressed('t'): # terminate program by 't'
        break
print("OFF")
```

The self-driving feature comprises three primary functions: `take_a_screenshot`, `decision`, and `process_result`. The functions `take_a_screenshot` and `process_result` are found in `functions.py`, while `decision` is located in `main.py`. This organization allows for easier model swapping when testing and comparing training results. First, the `take_a_screenshot` function captures and returns a screenshot. Next, the `decision` function uses this screenshot to generate a driving decision. Finally, `process_result` processes the decision result to control the car in Trackmania.

The `decision` function takes a screenshot of the Trackmania game, loads a specified neural network model, and runs a prediction on the screenshot, ultimately returning the prediction result.

Functions

As state before, the functions.py contains functions that are utilized in main.py including `take_a_screenshot`, `process_result`, and `get_window_info`.

The `take_a_screenshot` function accepts a window title as input. It first calls `get_window_info`, which provides the window's position and resolution. Using `pyautogui` along with the window position and resolution, it captures a screenshot. The screenshot is then preprocessed in the same manner used during data collection. This includes converting the PIL (Python Imaging Library / Pillow) image into a numpy array, resizing it to 400x225, cropping it, and color-correcting it from RGB to BGR. The processed screenshot is returned.

The `get_window_info` function also takes a window title as input. It retrieves information about the window using the `getWindowsWithTitle` function from the `pygetwindow` library. This function returns the window's screen position and resolution. If the window is found, a tuple containing the position and resolution is returned otherwise, `None` is returned.

The `process_result` function takes in the decision result and a debug toggle, which controls whether debugging information is printed in the terminal. At the start, it creates two variables: `sens`, which defines the threshold that result values must meet to enable key presses, and `press_t`, which defines the duration for which keys are pressed. The function then compares the decision result values against the `sens` threshold, generating `bool_result` which is a list of boolean values indicating which keys should be pressed. The keys are then pressed for the duration specified by `press_t`, after which they are released. The press and release of the was achieved a library called `pynput`.

Model Evaluation

To gain a clearer insight into the predictions made by our model, we checked the model's loss metrics. This allowed us to assess the model's performance across various training stages, identifying trends and areas for improvement.

In addition to evaluating the losses, we implemented a sanity check to validate the model's predictions on individual frames. This step was crucial in confirming that the model was accurately identifying the expected objects within the images. By comparing the predicted outputs against the ground truth, we were able to ensure the model's reliability and effectiveness in real-world scenarios. Overall, these analyses provided valuable feedback that guided our training process and improved our understanding of the model's capabilities and limitations.

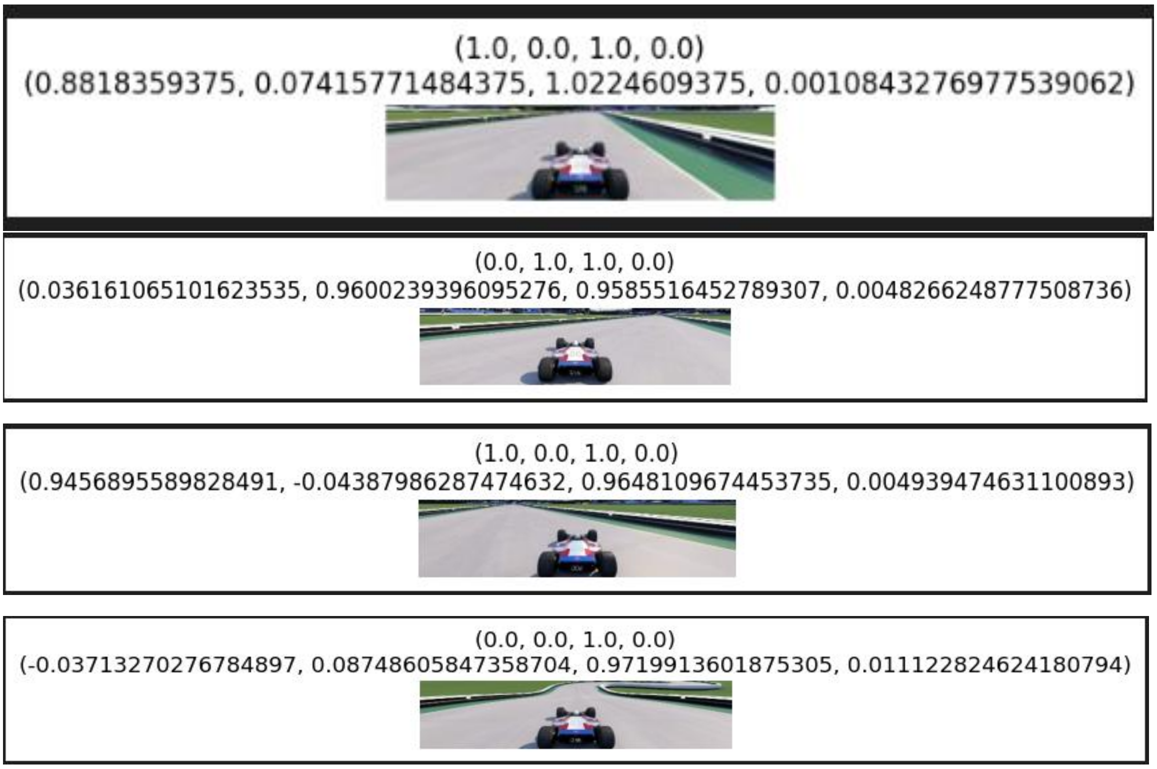

Figure 2 Sanity check inputs and predicted values

epoch	train_loss	valid_loss	time
0	0.866616	0.233225	00:54
1	0.528145	0.058399	00:56
2	0.297720	0.032369	00:56
3	0.181567	0.025021	00:55
4	0.127437	0.023011	00:58

Figure 3 losses per epoch

Yolo

YOLO for Object Detection

To begin our YOLO project, we created a custom dataset and labeled it using Roboflow. During the labeling process, we annotated each image by drawing bounding boxes around objects relevant to our application. Each bounding box is associated with one of six class labels: car, solid white and black lane markers, tree, curve, grass, and obstacles-borders. Along with each class label, the coordinates (x, y, width, height) of each bounding box were recorded.

After completing the annotation, we exported the dataset in YOLO format, organizing it into separate training and validation sets. This prepared dataset was then utilized to train a YOLO model, which we used for object detection in our application.

```
!yolo task=detect mode=train model=yolo11s.pt data={data_path} epochs=50 plots=True
```

We used a configuration file, data.yaml, to specify paths to the training and validation datasets, the number of classes, and their respective names. During training, YOLO learned to detect and draw bounding boxes around objects in the labeled dataset.

After training, we validated the YOLO model to ensure accurate detection on new, unseen images. The model achieved strong precision, recall, and mean Average Precision (mAP) scores, indicating reliable performance. However, certain classes, such as grass, presented occasional challenges due to a lower confidence threshold, leading to occasional misclassifications.

Once training was complete, we tested the YOLO model on new images, which it processed to produce bounding boxes around detected objects. Most objects were detected clearly, though some limitations were noted in specific classes, like grass, where occasional misclassification occurred.

```
[10] !yolo task=detect mode=predict model=runs/detect/train/weights/best.pt source=/content/drive/MyDrive/data/test/images conf=0.25 save=True
```

In this code, each test image is processed by the YOLO model, which outputs images with bounding boxes around detected objects. We inspected these images and confirmed that most objects, including the car, had clear bounding boxes. However, certain classes, like grass, were sometimes misclassified due to a lower confidence threshold.

After training and generating bounding boxes, we used YOLO's predictions to provide additional context to the CNN-LSTM model. The best-performing weights were saved in training iteration 8 on Google Colab. Now, the YOLO model is able to predict bounding boxes on new frames, accurately identifying object classes in the process.

```
# Step 1: Load the best-trained YOLO model
model = YOLO("runs/detect/train/weights/best.pt")

# Step 2: Define classes to detect and initialize yolo_features dictionary
classes = ['Car', 'Solid White and black', 'Tree', 'curve', 'grass', 'obstacles-Borders']

yolo_features = {}

# Step 3: Generate YOLO features by running inference on each frame
for frame_path in labels_df['frame'].unique():
    # Perform inference
    results = model.predict(path / frame_path, save=False)


# Safely extract detected classes for the current frame, converting det[5] to an integer if available
    detected_classes = [classes[int(det[5])] for det in results[0].boxes.data.tolist() if len(det) > 5]



    # Create a binary feature vector for each class
    yolo_features[frame_path] = torch.tensor(
        [1 if cls in detected_classes else 0 for cls in classes], dtype=torch.float32
    )

```

```
image 1/1 /content/drive/MyDrive/data/training ultimate/frame_406.jpg: 128x640 1 Car, 1 grass, 9.2ms
Speed: 0.9ms preprocess, 9.2ms inference, 1.3ms postprocess per image at shape (1, 3, 128, 640)

image 1/1 /content/drive/MyDrive/data/training ultimate/frame_407.jpg: 128x640 1 Car, 1 grass, 9.4ms
Speed: 0.9ms preprocess, 9.4ms inference, 1.3ms postprocess per image at shape (1, 3, 128, 640)
```

YOLO feature extraction

In this phase, for each frame, the YOLO model performed inference to detect objects. Each frame's detections were converted into a binary vector that indicated which objects were present. For example, a vector like `[1, 0, 1, 0, 1, 0]` could mean that a car, tree, and grass were detected, while solid white and black lanes, curve, and obstacles-borders were not detected. Each frame's feature vector was stored in `yolo_features`, keyed by the frame's filename, to use as input features in the subsequent CNN-LSTM model.

Creating Dataset With Yolo features

We created a custom dataset class, `SequenceDatasetWithYOLO`, which combines image sequences and the extracted YOLO features. For each sequence of frames, both image data and corresponding YOLO features were retrieved. The labels were derived from driving actions (left, right, forward, backward), corresponding to the action in the last frame of each sequence.

LSTM

The CNN_LSTM_MODEL combines CNN-based image feature extraction with LSTM-based sequence processing, using YOLO features as additional input. Starts off by Using CNN Feature Extraction where Frames are passed through the CNN (ResNet34 without fully connected layers) to extract image features. Then the LSTM Layer where the Image features and YOLO object features are concatenated and passed through a bidirectional LSTM, allowing the model to learn temporal dependencies. Finally in the output layer a fully connected layer predicts driving actions for the sequence.

CNN Feature Extraction

The frames are passed through a CNN (using ResNet34 without the fully connected layers) to extract image features. This step ensures that important spatial features from each frame are retained.

Integration of YOLO Features

YOLO features are concatenated with CNN features for each frame. This combined vector captures both spatial (CNN) and object presence (YOLO) data, allowing the model to consider both the visual content and contextual information from detected objects.

LSTM Layer

The image features and YOLO object features are concatenated and passed through a bidirectional LSTM. This allows the model to learn temporal dependencies, which are crucial for understanding the sequence of events across multiple frames, especially for predicting driving actions. The bidirectional LSTM processes the sequence forward and backward, considering both past and future frames, which helps anticipate driving actions based on temporal dynamics. Finally, The final LSTM output at the last time step is used as it captures accumulated information from the sequence, ready for action prediction.

Output Layer

A dropout layer is applied to the final LSTM output for regularization. Then finally, the output layer, The LSTM's final output is passed through a fully connected layer, predicting the driving action for the sequence (e.g., **left**, **right**, **forward**, **backward**). This layer translates the learned features into actionable predictions.

```python
class CNN_LSTM_YOLO(nn.Module):
    def __init__(self, cnn_base, lstm_hidden_dim, n_classes, yolo_feature_dim):
        super(CNN_LSTM_YOLO, self).__init__()
        # Use the provided CNN base (e.g., ResNet without the fully connected layers)
        self.cnn_base = cnn_base
        self.lstm_hidden_dim = lstm_hidden_dim

        # LSTM layer to process YOLO features along with CNN features
        self.lstm = nn.LSTM(input_size=25088 + yolo_feature_dim,  # CNN output size + YOLO feature vector size
                            hidden_size=lstm_hidden_dim,
                            num_layers=2,  # Increased number of layers for LSTM
                            batch_first=True,
                            bidirectional=True)  # Bidirectional LSTM
```

AI Usage

Prompt: Review this text: *documentation text*

Here's a revised version of your text with improved grammar, clarity, and consistency:

The `main.py` file contains the main body of the self-driving application. It runs a continuous `while` loop, allowing you to start and stop the self-driving feature. When the application turns on, it prints "ON" in the terminal. Pressing 'q' starts the self-driving feature, and pressing and holding 'e' stops it. To terminate the application, press and hold 't'. After termination, the application prints "OFF" in the terminal.

Prompt included in the image:

Exception has occurred: PicklingError
Can't pickle <function <lambda> at 0x0000027E91CB4AE0>: attribute lookup <lambda> on __main__ failed
File "C:\Users\matej\Desktop\Programing\AutoMania\Supervised\model.py", line 31, in <module>
learn.fine_tune(5)
_pickle.PicklingError: Can't pickle <function <lambda> at 0x0000027E91CB4AE0>: attribute lookup <lambda> on __main__ failed

what

The `PicklingError` you're encountering is caused by the fact that the Python `pickle` module cannot serialize (or "pickle") a lambda function. When `fastai` is saving or processing objects, it tries to serialize them, and if it encounters a lambda function, it will raise this error because lambda functions are not picklable.

Why This Error Occurs:

1. Lambda functions are anonymous and do not have a proper name that can be looked up in the module's namespace. Python's `pickle` module requires functions to be fully named in order to be serialized, but lambda functions do not meet this requirement.
2. The `learn.fine_tune()` method likely tries to save or serialize your model, and if a lambda is used anywhere in your code (e.g., in `dataloaders`, transformations, or custom callbacks), it will fail to pickle the object.

Prompt: How do I save a fastai model

To save the trained model in `fastai`, you can use the `save` method provided by the `Learner` class. This method allows you to save the model's state and parameters so you can load it later for inference or further training.

Here's how you can save and load the model:

1. Saving the Model

After training your model, you can save it using `learn.save()`. This will save the model weights (parameters) and optimizer state to a file.

```python
# Save the model after training
learn.save('trackmania_model')
```

This will create a file named `trackmania_model.pth` inside the `models` folder in your current working directory. FastAI automatically handles the saving of the model as a `.pth` file, which contains the model's weights and optimizer state.

If you want to specify a custom path to save the model, you can provide the path argument:

```python
learn.path = Path('path/to/save/directory')
learn.save('trackmania_model')
```

2. Loading the Model

When you want to use the model later (for inference or continued training), you can load it using `learn.load()`.

```python
# Load the saved model
learn.load('trackmania_model')
```

If you saved the model to a custom directory, make sure the `path` is set accordingly before loading:

```python
learn.path = Path('path/to/save/directory')
learn.load('trackmania_model')
```

3. Saving the Entire Learner

If you want to save the entire `Learner` object (not just the model weights, but also the data, metrics, loss function, etc.), you can use the `export` method. This saves the entire model pipeline, making it easier to reuse the model without having to recreate the `Learner`.

```python
# Export the entire learner (including preprocessing steps)
learn.export('trackmania_learner.pkl')
```

This will create a file called `trackmania_learner.pkl`. It includes the model, the architecture, and all of the preprocessing steps.

Prompt: Combine csv's and images from different folders into a single data frame:

Here's some code that does this:

python   Code kopiëren

```python
import os
import pandas as pd

# Base path to the data folder
base_data_path = 'Supervised/data/track2/'

# List all the folders in the base directory
folders = [f for f in os.listdir(base_data_path) if os.path.isdir(os.path.join(base_data_p

# Initialize an empty DataFrame to store all the data
combined_df = pd.DataFrame()

# Loop through each folder
for folder in folders:
    folder_path = os.path.join(base_data_path, folder)

    # Load the CSV file
    csv_file = os.path.join(folder_path, 'controls.csv')
    if os.path.exists(csv_file):
        df = pd.read_csv(csv_file)

        # Add a new column for the image path
        df['image_path'] = df.index.map(lambda i: os.path.join(folder_path, f'frame_{i}.jp

        # Combine with the main DataFrame
        combined_df = pd.concat([combined_df, df], ignore_index=True)

# Check the combined DataFrame
print(combined_df.head())
```