Embedded Systems - Project

Final Report

Date: December 15, 2024

Instructor: Muhammad Hassan Jamil

Group Members:

Mashrafi Monon (mmm9886)
Ameen Faraaz (av2851)

1. Introduction

The primary objective of this project is to design and construct a robot that can navigate a modular field of tiles by accurately following a black line. The robot is expected to overcome obstacles and adapt to environmental variability within the field. This dynamic field comprises various tiles that introduce distinct challenges, such as ramps, obstacles, and intersections. The robot must operate independently, with no prior knowledge of the field layout.

2. Project Overview

The robot is engineered to autonomously follow a black line on a white surface, continuously adapting to obstacles and variations in the course. The key specifications include:

- Ability to detect and follow a black line with high accuracy.
- Real-time obstacle detection and avoidance capabilities.
- Efficient navigation through dynamic field layouts.
- Integration of sensors and cameras for enhanced environmental awareness.

3. Methodology

3.1 JetBot Configuration

The motors were first attached to the metal box, followed by the antennas, which were passed through a round hole in the box and secured. The camera holder was then mounted to the box, along with the camera and Acrylic board. Standoffs were installed on both the metal box and the JetBot expansion board to prepare for mounting the Jetson Nano Developer Kit. Once the expansion board was in place, the 18650 batteries were installed with attention to their correct orientation, and the motors were connected and fixed on both sides.

Afterward, the wheels were attached to the metal bottom board, which was then secured to the metal box. The Jetson Nano board was installed on the expansion board after briefly removing it to install the Wireless-AC 8265. The cooling fan was connected to the Jetson Nano, and the glue antenna was installed, followed by the connection of the 6-pin cable according to the color scheme shown below in figure 1 [1].

Finally, the power adapter was connected to the expansion board, and the JetBot was powered on.

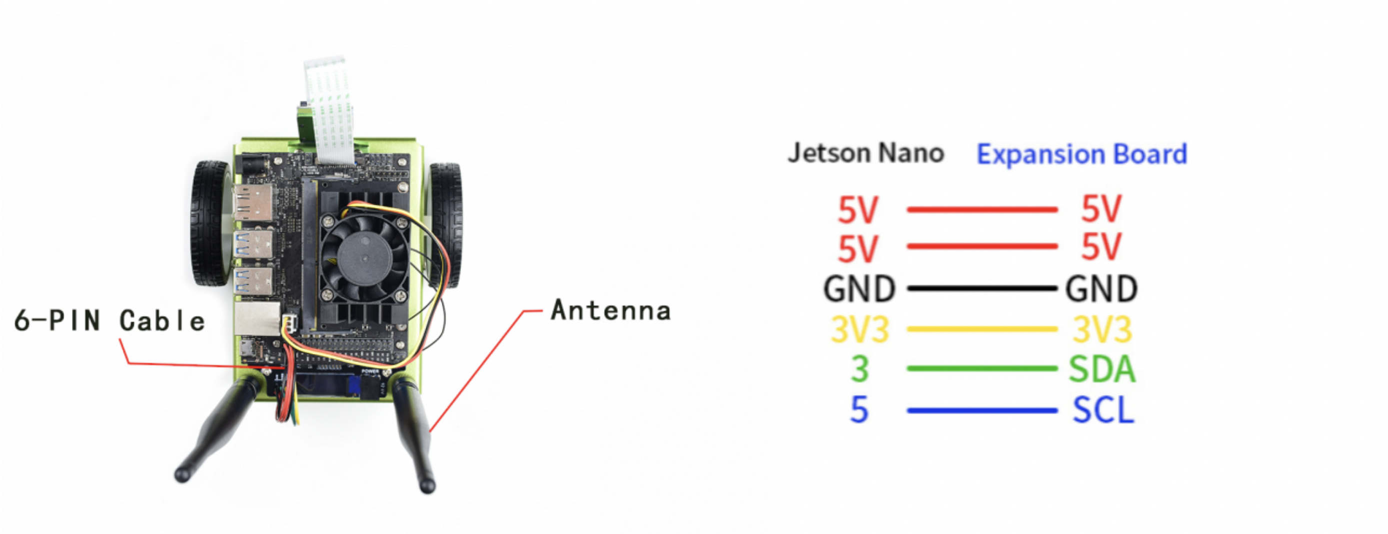

Figure 1: Pin connections between the Jetson Nano and Expansion Board

3.2 Data Collection

The data collection process involved placing the JetBot in various positions along the path, ensuring significant variation in the data. The live camera feed from the JetBot was displayed, and the point on the image to represent the target direction for the robot was selected. The X and Y coordinates of the point, along with the corresponding camera image, were saved as labels for training.

Key steps for this phase included:

- Placing the JetBot in different positions (offset from center, different angles, etc.).
- Displaying the live camera feed and annotating the images with a point to indicate the target direction.
- Saving the annotated images along with their corresponding X and Y values in the dataset.

The dataset was stored in a folder with filenames encoding the X and Y coordinates, such as `xy_<x>_<y>_<uuid>.jpg`. This ensured that the labeling process was efficient and the data was organized systematically.

The target placement guidelines included:

- Visualizing the path the robot should follow based on the live camera feed.
- Placing the target as far along the imagined path as possible while ensuring the robot could navigate safely without going off-road.
- Adjusting target placement for sharp turns by bringing it closer to the robot to prevent boundary violations.

Following these guidelines ensured the robot could navigate smoothly, with the target point progressing along the desired trajectory.

The collected dataset was used to train a neural network to predict the X and Y coordinates of the target points. The training process utilized a regression model, and

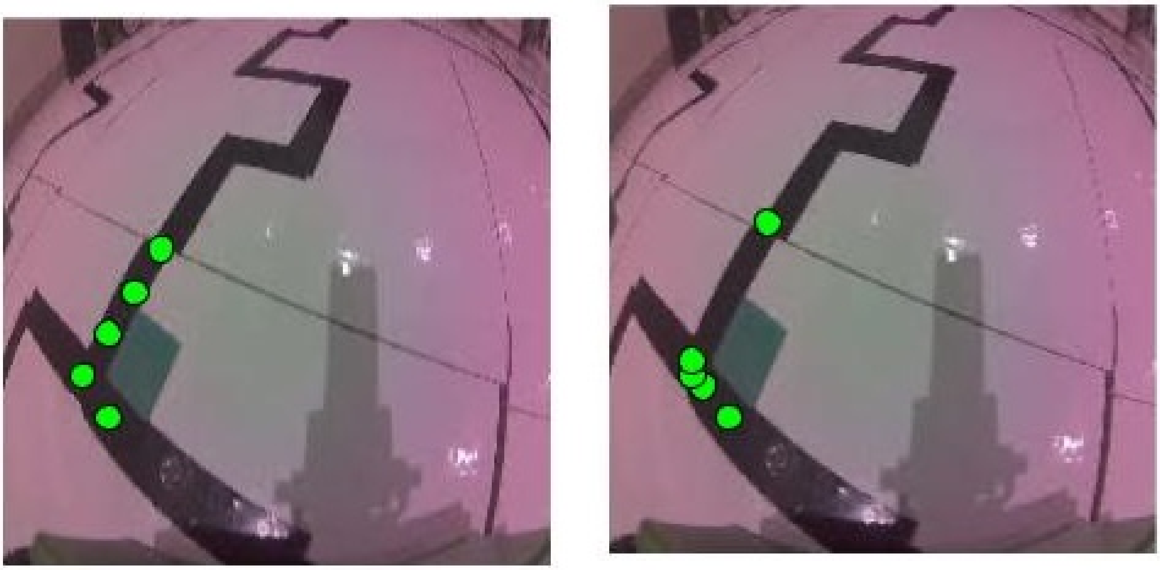

Figure 2: Labeling strategy. Left: Gradual Turn. Right: Sharp Turn. Green boxes represent position of the labels

the trained network was deployed on the JetBot. During deployment, the predicted X and Y values were used to compute an approximate steering direction. Although the values were not calibrated to angles, they were proportional to the required steering adjustments, enabling the JetBot to follow the path effectively.

The implementation involved the use of various Python libraries and tools:

- OpenCV for image visualization and annotation.
- `ipywidgets` and `traitlets` for creating interactive widgets in Jupyter notebooks.
- The `jetbot` library for interfacing with the JetBot's camera and motor.
- A clickable image widget, `jupyter_clickable_image_widget`, to capture annotation points directly from the live camera feed.

For labeling the image data we tested two methods:

- Initially we tried to label the data in a way that it follows the turn smoothly, i.e. starts turning early and gradually turns to the right or left rather than a sharp turn. The labeling is visualized in the Figure
- Although this method was working very well, we start having trouble in the box shaped paths. As the turns are very frequent, smoother turns means the jetbot won't be able to stay in the line most of the time and will keep turning. This meant higher chances of detecting wrong turns when it cannot align itself properly with the line. For this reason we labeled the data so that it makes a sharp turn when it nears the intersection. The way we achieved it is that instead of gradually changing the position to follow the turn, it points to the intersection until it gets pretty close to the intersection at which point the target point shifts towards the turn (right or left depending on specific scenarios). This ensures the jetbot doesn't deviate too much away from the line while turning and solves any issues arising while following box shaped path.

3.3 Deep Learning Model Training

To enable autonomous navigation of the JetBot, two distinct models were employed: the Steering Model and the Classification Model, each serving complementary roles in the road-following task. The ResNet-18 architecture was selected as the base model for both cases. A single fully connected convolutional layer was added at the end of each model. The output for the steering model was two values (x, y) for steering regression, while the output for the classification model was four values representing four classes. The final class was ignored, and the argmax function was applied to determine the desired class.

3.3.1 The Steering Model

The principal function of the steering model was to determine the steering direction based on visual input from the camera. This model was trained to manage various conditions and to direct the JetBot along the track. Its primary focus was on the continuous adjustment of the bot's trajectory. However, the steering model alone was inadequate for recognizing specific objects or stopping conditions.

3.3.2 The Classification Model

The classification model complemented the steering model by identifying specific objects or conditions within the environment. It was designed to classify situations such as:

- **Stop conditions:**
  - Red lines indicating a final stop.
  - Obstacles appearing anywhere on the track.
- **Normal driving conditions:** A clear track without obstacles.

This dual-model system allowed the JetBot to adapt effectively to its environment by halting when necessary and maintaining proper steering during normal conditions.

3.3.3 U-turn Detection

Detecting U-turns presented unique challenges that required specialized handling. Initially, U-turns were used as one of the classes in the classification model but it did not perform well. We devised the following labeling strategy to detect U-turns in the steering model:

- The y-coordinate of the predicted point was labeled very close to the JetBot, similar to sharp turns.
- The x-coordinate was labeled towards the center. This ensured the U-turn labeling was distinct from other sharp turns where the x-coordinate tended to be at the edges.

This distinct combination of x and y values enabled the differentiation of U-turns from other scenarios.

Despite these efforts, errors occasionally occurred when other conditions were misidentified as U-turns. To address this, a running average of the absolute steering value was implemented:

- During sharp turns, a single incorrect U-turn detection was ignored, as the running average of the steering value remained high.
- When approaching a U-turn from a straight path, the running average was low, ensuring accurate detection even if the x, y combination briefly matched for a few frames.

This refined approach significantly reduced false detections and improved the JetBot's ability to execute U-turns smoothly, ensuring reliable navigation through complex scenarios.

3.4 Robot Control

The autonomous navigation system for the JetBot involved several key steps to ensure smooth and effective operation, including image pre-processing, camera setup for real-time visualization, and implementing a control mechanism for steering the robot.

3.4.1 Image Pre-processing and Camera Setup

The image captured by the camera was pre-processed to ensure it was in the correct format for the neural network. The image was normalized by subtracting the mean pixel values and dividing by the standard deviation. This normalization step helped the network perform better during inference by matching the format used during the training process. The image was then reshaped to fit the network's input size.

The JetBot used a camera to capture real-time video frames. The feed from the camera was continuously processed frame-by-frame to ensure real-time decision-making. This feed was then passed to the neural network for processing.

3.4.2 Steering and Control Mechanism

The trained steering model was used to predict steering directions based on the pre-processed camera input. The model's output, consisting of x and y coordinates, was flattened and converted to a NumPy array for further processing. Using the predicted coordinates, the angle of deviation (θ) was calculated using the arccosine function and validated through trigonometric computations.

After the neural network performed inference on the image, the angle of deviation was used as the error. A PID controller was implemented to ensure smooth steering control by adjusting the steering commands based on error feedback. Furthermore, the classification model was checked first to confirm normal driving conditions; only then was the steering output utilized.

3.4.3 Optimizing Performance

To optimize performance, a fast mode was implemented. When the running average of the absolute steering value was significantly low and the y-coordinate was far from the JetBot—indicating a clear path ahead—the robot's speed was increased.

3.4.4 Turning Policies

For turning, two policies were used based on the steering value:

- **Low steering values:** Slight adjustments were made to the motor values to steer left or right gradually.
- **Sharp turns:** Signaled by a steering value exceeding a set threshold, opposite motor values were applied. This caused the JetBot to turn in place, ensuring alignment with the path without losing sight of the turn.

Results and Conclusion

The live demo demonstrated the effectiveness of the trained model in controlling the JetBot. The model successfully predicted the steering directions required for the JetBot to navigate the track. The outputs from the model were consistent with the expected behavior, and the JetBot was observed to move smoothly along the path without significant deviations.

The pre-processing function and model inference pipeline were verified to work effectively, enabling the JetBot to navigate the track autonomously.

A video is attached along with this report showing successful execution of the JetBot along the path.

Abstraction Model

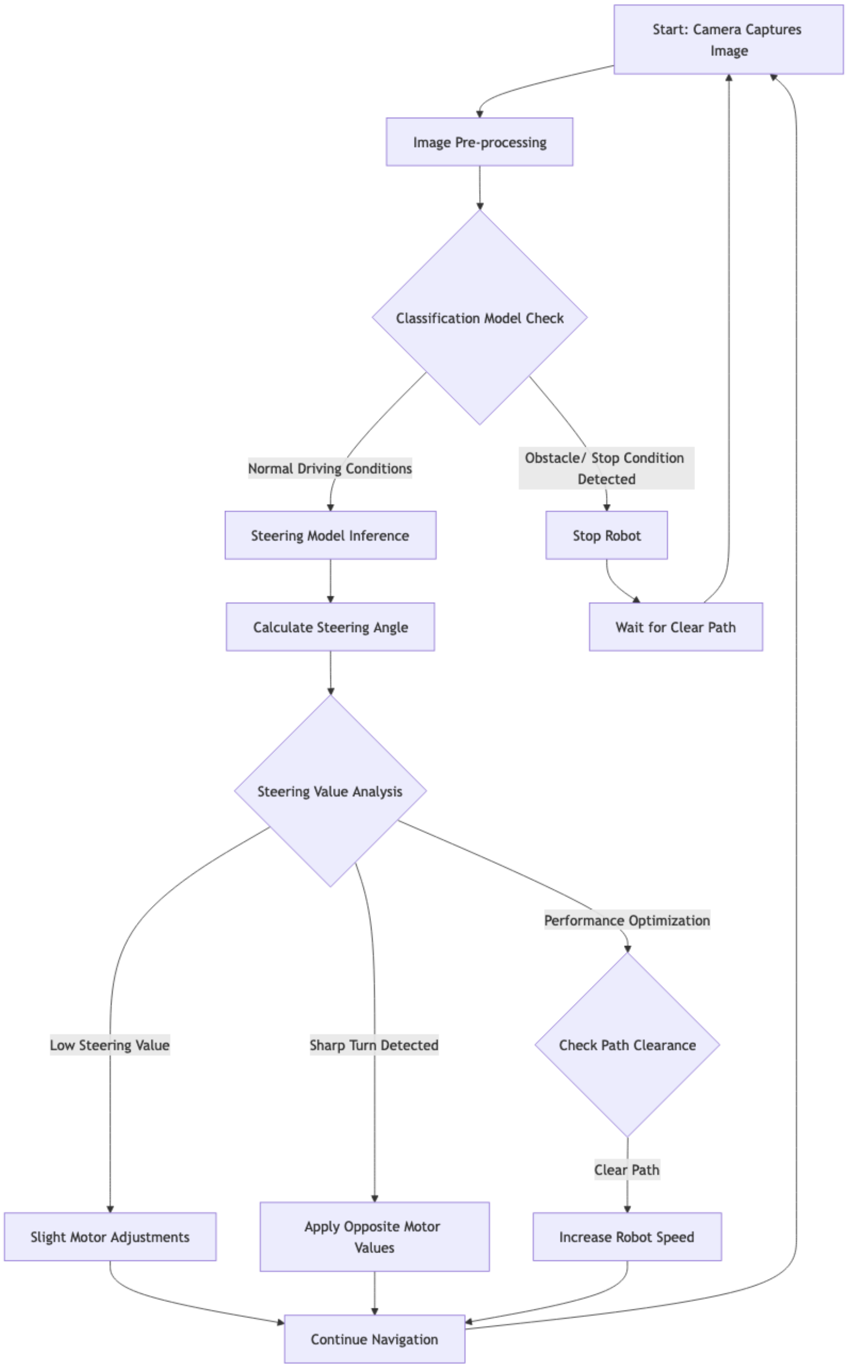

Figure 3: Abstraction Model

1 Appendix with Codes Discussed

A Code for Road Following Data Collection

Below is the Python code used for data collection. OpenCV is used to visualize and save image with labels. Libraries such as uuid, datetime are used for image naming.

```python



# Directory where dataset images will be stored
DATASET_DIR = 'dataset_final'

# Try to create the dataset directory if it doesn't already exist
# This prevents errors when re-running the script if the directory already exists
try:
    os.makedirs(DATASET_DIR)
except FileExistsError:
    print('Directories not created because they already exist')

# Initialize the camera object (assumes a pre-configured Camera class is available)
camera = Camera()

# Create an interactive image preview widget
# `camera_widget` will display the live feed from the camera
camera_widget = ClickableImageWidget(width=camera.width, height=camera.height)

# Widget to display snapshots (saved images from the camera)
snapshot_widget = ipywidgets.Image(width=camera.width, height=camera.height)

# Link the camera feed (BGR image) to the `camera_widget` for real-time display
# Convert BGR to JPEG format for display
traitlets.dlink((camera, 'value'), (camera_widget, 'value'), transform=bgr8_to_jpeg)

# Widget to track and display the number of saved images
count_widget = ipywidgets.IntText(description='count')

# Initialize the `count_widget` value by counting the existing image files in the dataset directory
count_widget.value = len(glob.glob(os.path.join(DATASET_DIR, '*.jpg')))

# Function to handle clicks on the camera widget and save snapshots
def save_snapshot(_, content, msg):
    # Check if the event is a 'click' (to avoid processing other types of events)
    if content['event'] == 'click':
        data = content['eventData']
        x = data['offsetX']  # X-coordinate of the click within the image
        y = data['offsetY']  # Y-coordinate of the click within the image
```

```python

        # Save the snapshot to the dataset directory with a unique filename
        uuid = 'xy_%03d_%03d_%s' % (x, y, uuid1())  # Generate a unique identifier
        image_path = os.path.join(DATASET_DIR, uuid + '.jpg')  # Path to save the image
        with open(image_path, 'wb') as f:
            f.write(camera_widget.value)  # Save the current image displayed in the widget

        # Update the snapshot preview widget to show the saved image with a marker
        snapshot = camera.value.copy()  # Copy the current camera frame
        snapshot = cv2.circle(snapshot, (x, y), 8, (0, 255, 0), 3)  # Mark the click location with a green circle
        snapshot_widget.value = bgr8_to_jpeg(snapshot)  # Update the snapshot widget with the modified image

        # Update the count of saved images
        count_widget.value = len(glob.glob(os.path.join(DATASET_DIR, '*.jpg')))

# Register the `save_snapshot` function to handle click events on the `camera_widget`
camera_widget.on_msg(save_snapshot)

# Create a layout for the data collection interface
# This interface includes the camera feed, snapshot preview, and image count display
data_collection_widget = ipywidgets.VBox([
    ipywidgets.HBox([camera_widget, snapshot_widget]),  # Side-by-side layout of camera and snapshot widgets
    count_widget  # Display the image count widget below the previews
])

# Display the data collection widget in the Jupyter notebook
display(data_collection_widget)
```

B Road Follower - Train Model

The model is trained using the images collected by the JetBot and includes the setup of the neural network architecture, data preprocessing, and training the model with a dataset.

B.1 Create Dataset Instance

```python
# For steering regression
def get_x(path, width):
    """Extract the normalized x-coordinate value from the image filename.
    The filename is assumed to have coordinates encoded in its structure, with `_` as the delimiter.
```

```python
    The x-coordinate is scaled to the range [-1, 1] relative to the image width.
    Args:
        path (str): The file path of the image.
        width (int): The width of the image.
    Returns:
        float: The normalized x-coordinate.
    """
    return (float(int(path.split("_")[1])) - width / 2) / (width / 2)

def get_y(path, height):
    """Extract the normalized y-coordinate value from the image filename.
    Similar to `get_x`, the y-coordinate is scaled to the range [-1, 1] relative to the image height.
    Args:
        path (str): The file path of the image.
        height (int): The height of the image.
    Returns:
        float: The normalized y-coordinate.
    """
    return (float(int(path.split("_")[2])) - height / 2) / (height / 2)

# For classification
def get_class(path):
    """Determine the class label of an image based on its directory name.
    The method checks the second element of the file path (split by `/`) to assign a one-hot encoded class label.
    Args:
        path (str): The file path of the image.
    Returns:
        list: A one-hot encoded class label (4 possible classes).
    """
    if path.split("/")[1] == 'dataset_xy2':
        return [1, 0, 0, 0]  # Class for 'dataset_xy2'
    elif path.split("/")[1] == 'object':
        return [0, 1, 0, 0]  # Class for 'object'
    elif path.split("/")[1] == 'stop':
        return [0, 0, 1, 0]  # Class for 'stop'
    elif path.split("/")[1] == 'uturn':
        return [0, 0, 0, 1]  # Class for 'uturn'

# Custom PyTorch Dataset for loading and preprocessing images
class XYDataset(torch.utils.data.Dataset):

    def __init__(self, directory, random_hflips=False):
        """Initialize the dataset with image directory and preprocessing options.
        Args:
            directory (str): Path to the directory containing the dataset.
            random_hflips (bool): Whether to randomly flip images horizontally.
        """
        self.directory = directory
        self.random_hflips = random_hflips
```

```
        # Collect all image file paths with the .jpg extension in the directory
        self.image_paths = glob.glob(os.path.join(self.directory, '*.jpg'))
        # Define a color jitter transformation for data augmentation
        self.color_jitter = transforms.ColorJitter(0.3, 0.3, 0.3, 0.3)

    def __len__(self):
        """Return the total number of images in the dataset."""
        return len(self.image_paths)

    def __getitem__(self, idx):
        """Retrieve and preprocess a single image and its label.
        Args:
            idx (int): Index of the image to retrieve.
        Returns:
            tuple: Processed image tensor and corresponding label tensor ([x, y] coordinates or class).
        """
        image_path = self.image_paths[idx]
        # Open the image using PIL
        image = PIL.Image.open(image_path)
        # Extract image dimensions
        width, height = image.size
        # Get normalized x and y values based on the filename
        x = float(get_x(os.path.basename(image_path), width))
        y = float(get_y(os.path.basename(image_path), height))
        # cl = get_class(image_path)  # Uncomment for classification tasks

        # Apply random horizontal flip for data augmentation
        if float(np.random.rand(1)) > 0.5:
            image = transforms.functional.hflip(image)
            x = -x  # Flip the x-coordinate as well

        # Apply color jitter for brightness, contrast, saturation, and hue changes
        image = self.color_jitter(image)
        # Resize the image to 224x224 pixels
        image = transforms.functional.resize(image, (224, 224))
        # Convert the image to a PyTorch tensor
        image = transforms.functional.to_tensor(image)
        # Reverse the channel order (e.g., BGR to RGB) and copy the array to avoid shared memory issues
        image = image.numpy()[::-1].copy()
        image = torch.from_numpy(image)
        # Normalize the image using ImageNet's mean and standard deviation
        image = transforms.functional.normalize(image, [0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

        # Return the processed image and label
        return image, torch.tensor([x, y]).float()
        # return image, torch.tensor(cl).long()  # Uncomment for classification tasks

# Instantiate the dataset
dataset = XYDataset('dataset_final', random_hflips=False)
```

```

# Define the Neural Network for road following
class RoadFollowerNN(nn.Module):
    def __init__(self):
        """Initialize the road-following neural network.
        The network uses two convolutional layers followed by two fully connected layers.
        """
        super(RoadFollowerNN, self).__init__()
        # First convolutional layer: 3 input channels (RGB), 16 output channels
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
        # Second convolutional layer: 16 input channels, 32 output channels
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        # First fully connected layer: input size matches flattened feature map size
        self.fc1 = nn.Linear(32 * 64 * 64, 512)  # Assumes input image size is 64x64
        # Second fully connected layer: output size is 2 (steering angle and speed)
        self.fc2 = nn.Linear(512, 2)

    def forward(self, x):
        """Forward pass through the network.
        Args:
            x (Tensor): Input image tensor.
        Returns:
            Tensor: Predicted steering angle and speed.
        """
        # Apply ReLU activation after the first convolution
        x = torch.relu(self.conv1(x))
        # Apply ReLU activation after the second convolution
        x = torch.relu(self.conv2(x))
        # Flatten the tensor before passing it to the fully connected layers
        x = x.view(x.size(0), -1)
        # Apply ReLU activation after the first fully connected layer
        x = torch.relu(self.fc1(x))
        # Final output layer for steering angle and speed
        x = self.fc2(x)
        return x
```

B.2 Train Regression

We train for 50 epochs and save best model if the loss is reduced.

```

# Number of training epochs
NUM_EPOCHS = 20

# Path to save the best model during training
# Uncomment the relevant model path depending on the model type
# BEST_MODEL_PATH = 'best_steering_model_xy.pth'
# BEST_MODEL_PATH = 'best_turning_model_xy.pth'
```

```python
# BEST_MODEL_PATH = 'best_combined_model_xy.pth'
# BEST_MODEL_PATH = 'best_classify_model_xy.pth'
BEST_MODEL_PATH = 'best_combined_final_model_xy.pth'

# Initialize the best loss value to a very high number (to track
    improvement)
best_loss = 1e9

# Define the optimizer for the model parameters
optimizer = optim.Adam(model.parameters())

# Training loop
for epoch in range(NUM_EPOCHS):
    model.train()  # Set model to training mode
    train_loss = 0.0  # Track cumulative training loss

    # Iterate over the training data loader
    for images, labels in iter(train_loader):
        images = images.to(device)  # Move images to the device (e.g.,
            GPU)
        labels = labels.to(device)  # Move labels to the device
        optimizer.zero_grad()  # Clear gradients from the previous step
        outputs = model(images)  # Perform a forward pass to get
            predictions

        # Uncomment for classification tasks where labels need to be
            one-hot encoded
        # labels = labels.argmax(dim=1)

        # Calculate mean squared error loss
        loss = F.mse_loss(outputs, labels)
        train_loss += float(loss)  # Accumulate loss
        loss.backward()  # Backpropagate gradients
        optimizer.step()  # Update model parameters

    # Average training loss over all batches
    train_loss /= len(train_loader)

    # Switch to evaluation mode for validation
    all_output = []  # Collect all model predictions
    all_label = []  # Collect all ground truth labels
    model.eval()  # Set model to evaluation mode
    test_loss = 0.0  # Track cumulative test loss

    # Iterate over the test data loader
    for images, labels in iter(test_loader):
        images = images.to(device)  # Move images to the device
        labels = labels.to(device)  # Move labels to the device
        outputs = model(images)  # Perform a forward pass

        # Uncomment for classification tasks where labels need to be
            one-hot encoded
        # labels = labels.argmax(dim=1)

        all_output += [outputs]  # Collect predictions
        all_label += [labels]  # Collect true labels

        # Calculate mean squared error loss for the test data
```

```python
        loss = F.mse_loss(outputs, labels)
        test_loss += float(loss)  # Accumulate loss

    # Average test loss over all batches
    test_loss /= len(test_loader)

    # Print the training and test loss for this epoch
    print('%f, %f' % (train_loss, test_loss))

    # Save the model if the test loss improves
    if test_loss < best_loss:
        torch.save(model.state_dict(), BEST_MODEL_PATH)  # Save model parameters
        best_loss = test_loss  # Update the best loss value

# Custom Dataset class to load images and labels
class RoadFollowerDataset(Dataset):
    def __init__(self, image_folder, label_file):
        self.image_folder = image_folder  # Path to the folder containing images
        self.label_file = label_file  # Path to the label file
        self.image_paths = [os.path.join(image_folder, fname) for fname in os.listdir(image_folder)]
        self.labels = np.loadtxt(label_file)  # Load labels from the file

    def __len__(self):
        return len(self.image_paths)  # Return the total number of samples

    def __getitem__(self, idx):
        # Read the image at the given index
        image = cv2.imread(self.image_paths[idx])
        image = cv2.resize(image, (64, 64))  # Resize image to 64x64 pixels
        image = np.transpose(image, (2, 0, 1))  # Convert to (C, H, W) format
        image = torch.tensor(image, dtype=torch.float32) / 255.0  # Normalize to [0, 1]

        # Get the corresponding label
        label = torch.tensor(self.labels[idx], dtype=torch.float32)  # Convert label to tensor
        return image, label  # Return image and label as a tuple

# Load dataset
train_dataset = RoadFollowerDataset(image_folder='images', label_file='labels.txt')
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)  # DataLoader for training data
```

C Road Following - Live Demo

Here we used the model we had trained to move the JetBot smoothly on the track.

C.1 Load Trained Model

The following code was executed to initialize the PyTorch model

```
# Initialize the steering model using ResNet-18 architecture
# ResNet-18 is a convolutional neural network (CNN) commonly used for image-related tasks
model_steer = torchvision.models.resnet18(pretrained=False)  # Load ResNet-18 without pretrained weights
model_steer.fc = torch.nn.Linear(512, 2)  # Replace the fully connected layer to output 2 values (x, y)

# Initialize the classification model using ResNet-18 architecture
# This model classifies inputs into 4 different classes
model_cls = torchvision.models.resnet18(pretrained=False)  # Load ResNet-18 without pretrained weights
model_cls.fc = torch.nn.Sequential(  # Replace the fully connected layer with a sequential block
    torch.nn.Linear(512, 4)  # Fully connected layer outputs 4 values (one for each class)
)
```

C.2 Robot Control with PID

Now we define the PID control class for the robot, and adjust motor speeds based on steering and speed values.

Listing 1: PID Control Class

```
class JetbotPIDControl:
    def __init__(self):
        # Initialize the robot instance
        self.robot = Robot()

        # PID coefficients for controlling the steering
        self.kp = 0.15  # Proportional gain (how strongly to react to the current error)
        self.ki = 0.005  # Integral gain (accounts for past errors)
        self.kd = 0.005  # Derivative gain (reacts to the rate of error change)
#          self.y_coeff = 0.2  # Optional adjustment coefficient for the y-direction
        self.steering_threshold = 0.05  # Steering adjustment threshold to avoid over-correcting

        # Speed settings
        self.base_speed = 0.08  # Base speed for the robot
        self.min_speed = 0.08  # Minimum speed allowed
        self.max_speed = 0.09  # Maximum speed allowed

        # Error tracking variables for PID control
        self.last_error = 0  # Last error value for calculating the derivative term
        self.integral = 0  # Accumulated error for the integral term
        self.last_time = time.time()  # Timestamp of the last PID update

#      def jitter(self, l, r):
```

```
#         """(Optional) A jitter movement to quickly adjust motors."""
#         self.robot.stop()
#         time.sleep(400/1000)  # Pause for stabilization
#         self.robot.left_motor.value = 0.2 * l  # Set left motor speed
#         self.robot.right_motor.value = 0.2 * r  # Set right motor speed
#         time.sleep(40/1000)  # Brief movement
#         self.robot.stop()
#         time.sleep(400/1000)  # Pause again for stabilization

#     def right(self):
#         """(Optional) Specific right-turn adjustment."""
#         self.jitter(4, 4)  # Perform jitter
#         self.robot.stop()
#         time.sleep(400/1000)
#         self.robot.left_motor.value = 0.2  # Left motor forward
#         self.robot.right_motor.value = -0.2  # Right motor reverse
#         time.sleep(300/1000)  # Move for a short duration
#         self.robot.stop()
#         time.sleep(400/1000)

    def left(self):
        """Perform a left turn by reversing the left motor and moving the right motor forward."""
        self.robot.stop()
        time.sleep(400/1000)  # Pause before starting
        self.robot.left_motor.value = -0.2  # Reverse left motor
        self.robot.right_motor.value = 0.2  # Forward right motor
        time.sleep(500/1000)  # Turn duration
        self.robot.stop()
        time.sleep(400/1000)  # Pause after turn

    def update_motors(self, x, y, cl, fast_mode):
        """
        Update motor speeds based on the target point (x, y).
        x: [-1, 1] Horizontal position, where -1 is far left, and 1 is far right.
        y: [-1, 1] Vertical position, where -1 is far back, and 1 is far forward.
        cl: Classifier result for specific conditions (e.g., stop, left).
        fast_mode: Boolean to enable faster speed mode.
        """
        if (cl == 1) or (cl == 2):  # Stop the robot for specific classifier results
            self.temp_stop()
            return None, None

        elif cl == 3:  # Perform a double left turn for this classifier condition
            self.left()
            self.left()
            return None, None
        else:
            current_time = time.time()
            dt = current_time - self.last_time  # Time difference since last update

```

```
            # Prevent division by zero or very small dt
            if dt < 0.001:
                dt = 0.001

            # Calculate steering angle using the x and y values
            theta = np.arccos(x / np.sqrt(x * x + (-y + 1) * (-y + 1)))  # Angle between target and center
            error = -(theta - (pi / 2))  # Error relative to a straight path

            # PID control terms
            p_term = self.kp * error  # Proportional term
            self.integral += error * dt  # Accumulate error for the integral term
            i_term = self.ki * self.integral  # Integral term
            derivative = (error - self.last_error) / dt  # Change in error for derivative term
            d_term = self.kd * derivative  # Derivative term

            # Calculate total steering value (sum of PID terms)
            steering = p_term + i_term + d_term

            # Clamp steering to valid range [-1, 1]
            steering = max(min(steering, 1.0), -1.0)

            # Calculate speed based on y value
            speed = self.min_speed + (self.max_speed - self.min_speed) * (-y + 1) / 2  # Map y to speed range

            # Determine motor speeds based on steering
            if abs(steering) < self.steering_threshold:  # Minor steering adjustments
                left_speed = speed + steering
                right_speed = speed - steering
            else:  # Larger steering adjustments
                left_speed = speed * 1.5 * steering / abs(steering)
                right_speed = -speed * 1.5 * steering / abs(steering)

            # Normalize speeds if exceeding [-1, 1]
            max_speed = max(abs(left_speed), abs(right_speed))
            if max_speed > 1.0:
                left_speed /= max_speed
                right_speed /= max_speed

            # Update motor speeds
            if fast_mode:  # Fast mode applies an additional multiplier
                self.robot.left_motor.value = float(left_speed * 1.8)
                self.robot.right_motor.value = float(right_speed * 1.8)
            else:  # Normal mode
                self.robot.left_motor.value = float(left_speed)
                self.robot.right_motor.value = float(right_speed)

            # Update error and time tracking for next iteration
            self.last_error = error
            self.last_time = current_time
            return steering, speed

    def stop(self):
```

```
        """Stop the robot and reset PID variables."""
        self.robot.stop()
        self.last_error = 0
        self.integral = 0

    def temp_stop(self):
        """Temporary stop for the robot."""
        self.robot.stop()
```

```
# Initialize the Jetbot PID controller
controller = JetbotPIDControl()

# Import deque for maintaining a fixed-length buffer
from collections import deque

# Define buffer length for calculating average steering
n = 20

# Initialize the buffer with default values (0.1 repeated n times)
buffer = deque([0.1] * n, maxlen=n)

while True:
    fast_mode = False  # Flag to toggle fast mode for the controller

    # Preprocess the camera input and obtain steering coordinates (x, y)
    xy = model_steer(preprocess(camera.value)).detach().float().cpu().numpy().flatten()
    x = xy[0]  # Steering direction (x-axis)
    y = xy[1]  # Steering speed (y-axis)

    # Preprocess the camera input for classification (e.g., object detection)
    cl = model_cls(preprocess(camera.value)).detach().float().cpu().numpy().flatten()
    cl = cl.argmax()  # Get the class with the highest probability

    # Reset cl to 0 if it equals 3 (specific behavior adjustment)
    if cl == 3:
        cl = 0

    # Check conditions to trigger a double left turn
    if abs(x) < 0.15 and y > 0.35 and abs(sum(buffer)) / n < 0.015:
        controller.left()  # Execute a left turn
        controller.left()  # Repeat the left turn
        continue  # Skip the remaining logic and restart the loop

    # --- Uncommented block for jitter motion (can be removed if not needed) ---
    # cum_cl = [cl]
    # if cl != 0:
    #     controller.jitter(1, 1)
    #     cum_cl += [cl]
    #     controller.jitter(-1, 1)
    #     cum_cl += [cl]
    #     controller.jitter(1, -1)
    #     cum_cl += [cl]
    #     controller.jitter(-1, -1)
```

```
    #     cum_cl += [cl]
    # cl = mode(cum_cl)
    # controller.temp_stop()
    # --------------------------------------------------------------------------------

    # Enable fast mode if buffer's average is below the threshold and y indicates reverse motion
    if abs(sum(buffer)) / n < 0.0035 and y < -0.4:
        fast_mode = True
    else:
        fast_mode = False  # Otherwise, stay in normal mode

    # Update motor controls based on x, y, cl (class), and fast_mode
    steering, speed = controller.update_motors(x, y, cl, fast_mode)

    # Clear previous console output (useful in Jupyter or real-time display)
    clear_output(wait=True)

    # If valid steering and speed values are returned, display them
    if steering is not None and speed is not None:
        buffer.append(abs(steering))  # Update buffer with the absolute steering value
        display(f'''steering: {steering:.2f}
                    Speed: {speed:.2f}
                    x: {x:.2f}
                    y: {y:.2f}
                    buffer: {sum(buffer)/n:.4f}
                    cl: {cl}
                    ''')  # Display relevant metrics

    # Small delay to prevent CPU overload during the loop
    time.sleep(1 / 1000)
```

References

[1] Mingxing Tan, Quoc V. Le. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*. Available: https://arxiv.org/pdf/1905.11946.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. *Deep Residual Learning for Image Recognition*. Available: https://arxiv.org/abs/1512.03385.

[3] Waveshare. *JetBot AI Kit Assemble Manual*. Available: https://www.waveshare.com/wiki/JetBot_AI_Kit_Assemble_Manual.

[4] Viso AI, "ResNet (Residual Neural Network) – Comprehensive Guide," available at: https://viso.ai/deep-learning/resnet-residual-neural-network/, Accessed: December 15, 2024.