Mobile Robot Navigation

Group 4
Andrea Liboni - Jeeson Kurian - Moritz Gutschi

Project objective

Goal: Develop and integrate AI models for autonomous robotic perception, control, and classification using ROS and Gazebo.

Tasks:

- State Estimation with LSTM to predict the robot position
- DQN/DDPG Architecture for Robot Control using ROS gazebo
- AI Model for YCB Object Classification

State Estimation
Overview

To achieve this task I used the collected rosbag files to train a LSTM model to improve the prediction made with the IMU data in respect with the ground truth extracted from the ODOM data.

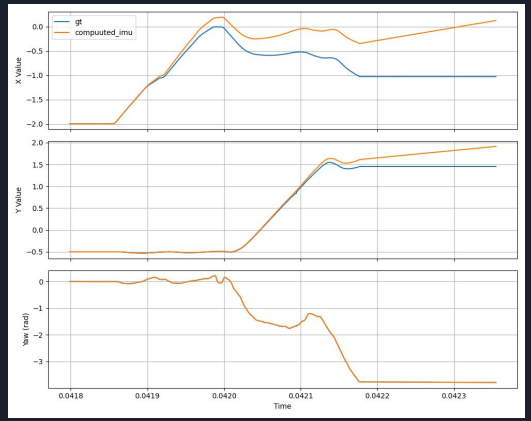
State Estimation
IMU prediction

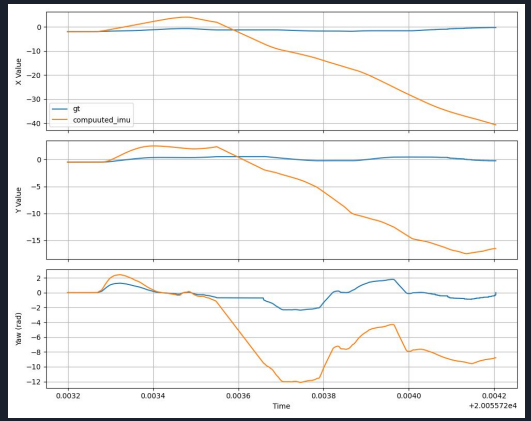
This is the first step to predict the position of the model.

I get the data from the IMU topic and then predict the current position starting from a know position.

This method works well in some situations, when the movement is predictable, but it is very easy that the prediction diverges from the actual position of the robot.

Example of divergence form odom data.





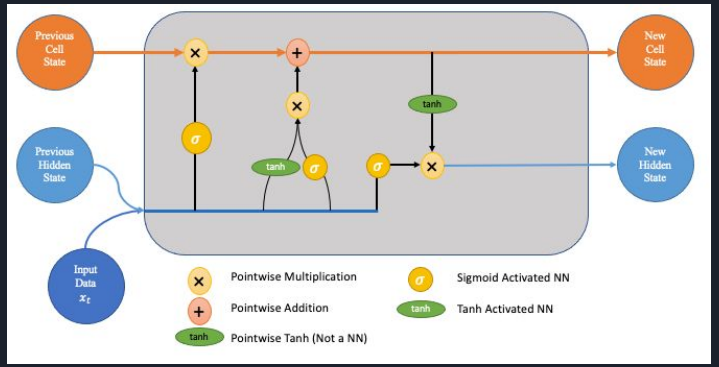
State Estimation
Model Architecture

This model takes as input the data computed from the function that we have seen in the previous slide, and tries to correct it in respect to the ground truth.

It uses 3 LSTM modules, one for each function (X, Y, Yaw) and at the end of each module we have a Linear layer that gives the final value as output.

```
self.lstm_x = nn.ModuleDict({
    'lstm': nn.LSTM(
        input_size=1,
        hidden_size=hidden_size,
        num_layers=num_layers,
        batch_first=True
    ),
    'linear': nn.Linear(hidden_size, 1)
})
```

Module for the X function



LSTM architecture

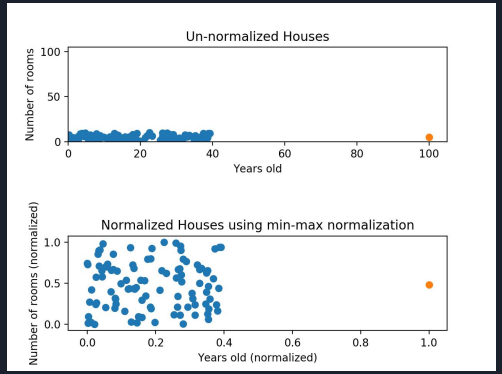
State Estimation
Data Preparation

Before feeding the data directly to the model, I have to prepare them correctly.

Because, since the LSTM works with sequence of data, I have to prepare a sequence of N input data that have a single value as ground truth. In the conclusion the model takes as input in the shape of [batch_size, sequence_length, 3] where the 3 correspond to the 3 value of X,Y and Yaw.

To achieve better results in the training is also applied the data normalization techniques MinMax, that compress the data into a range between 0 and 1, to facilitate the learning.

Example of the MinMax normalization



State Estimation
Training

The training is made by taking a sequence of bagfiles, prepare them as seen before, and then feed them into the model.

But before using them I split the sequence into two sets, one that can be used for training an the other one for testing. This spit is made at the 80% of the sequence.

After the computation I check for the loss in respect to the ground truth (the odom data) and save only the result that performed better.

A learning rate scheduler based on steps is also used to avoid divergences during training.

State Estimation
Results

The results are not perfect. This is probably caused by the small amount of data available for the training (8 bag file + 1 for testing) and also by the limited number of epochs used to train the model*.

The best configuration used for the training was with a length of the sequence of 10, a hidden size of 10, and a learning rate of 0.007, with a scheduler that lower it by 30% every 5000 epochs.

In the next slide you can see the actual results compared to the odom data and with the plain imu prediction.

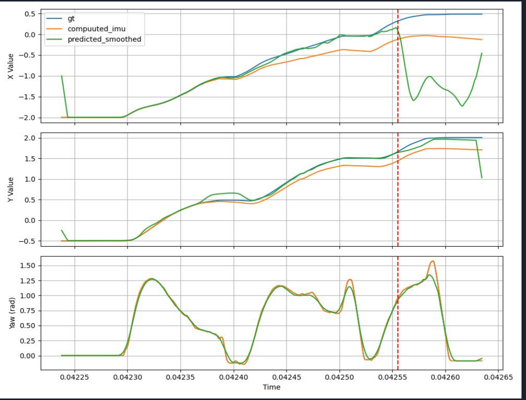
Note also that the function is smoothed to eliminate outliers with the following function

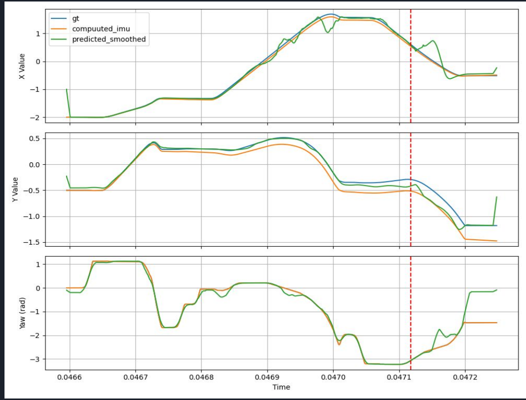
```
def smooth(y, box_pts):
    box = np.ones(box_pts)/box_pts
    y_smooth = np.convolve(y, box, mode='same')
    return y_smooth
```

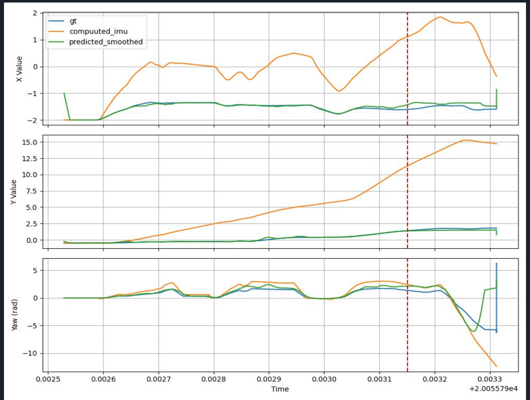
* This is because a limitation were the hardware resources

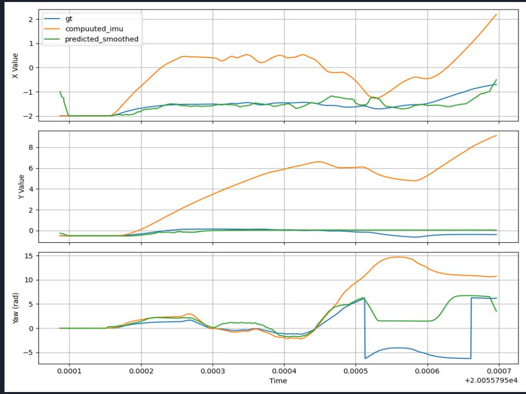
State Estimation

Results









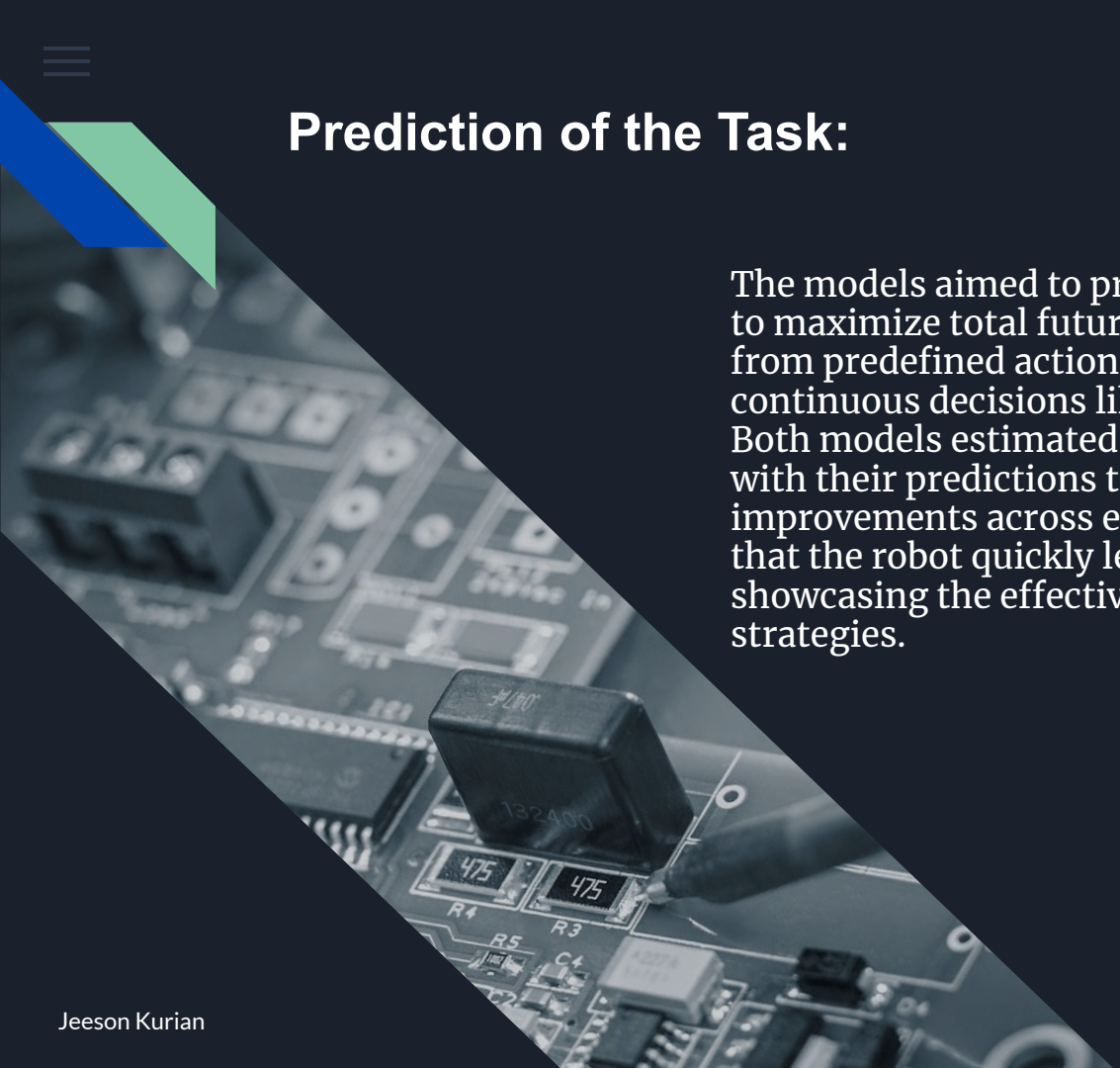
Note:
The red line is the limit of the training set of the rosbag.
The last graph (bottom right) is from a rosbag file that was NOT used as training data.
It is possible to see more results in the zip file.

Task 2 Overview:

This task trained a TurtleBot3 robot to navigate autonomously in a simulated environment using reinforcement learning (RL) techniques, implementing Deep Q-Network (DQN) and Deep Deterministic Policy Gradient (DDPG) within the ROS Noetic and Gazebo framework. The robot learned optimal movements by interacting with its environment, determining actions that would yield the best long-term rewards through hyperparameter tuning and neural network architecture exploration. The goal of achieving stable, high-performance navigation was successfully met by evaluating learning stability and reward progression.

Jeeson Kurian

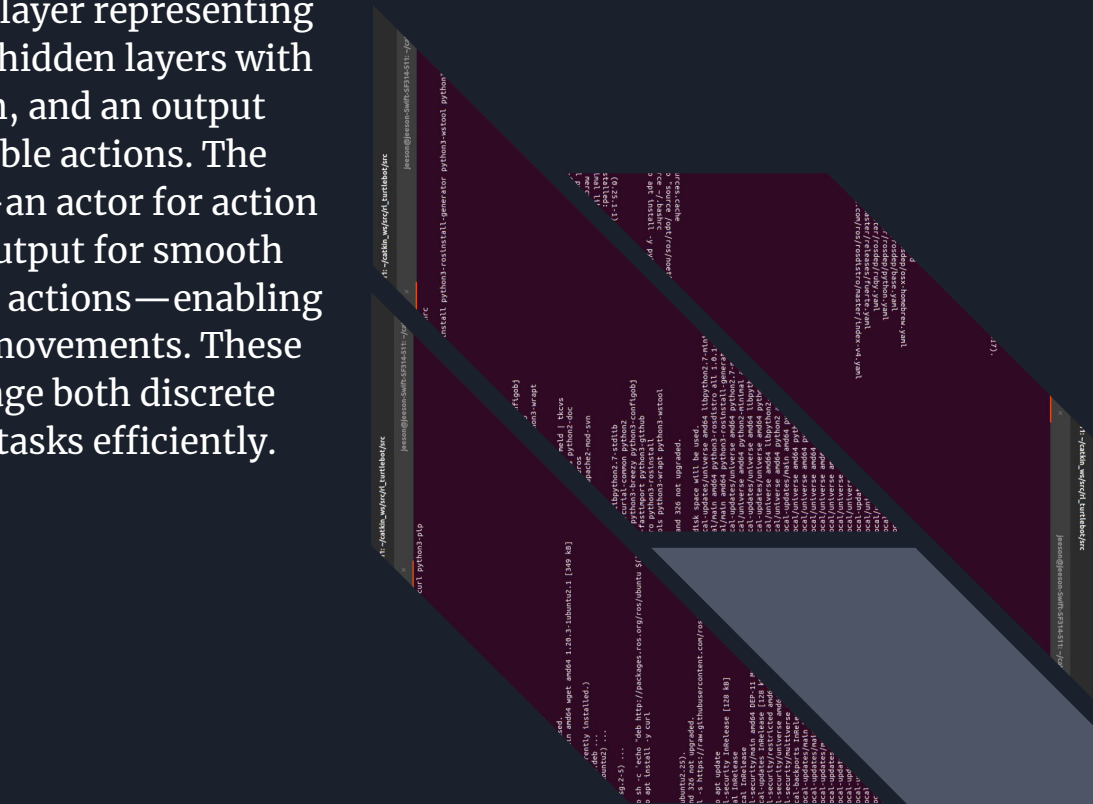
Prediction of the Task:



The models aimed to predict the best action at each step to maximize total future rewards, with DQN selecting from predefined actions and DDPG making flexible, continuous decisions like adjusting speed and direction. Both models estimated the long-term value of actions, with their predictions tested by tracking performance improvements across episodes. The results demonstrated that the robot quickly learned to select smarter actions, showcasing the effectiveness of both prediction strategies.

Model Architecture:

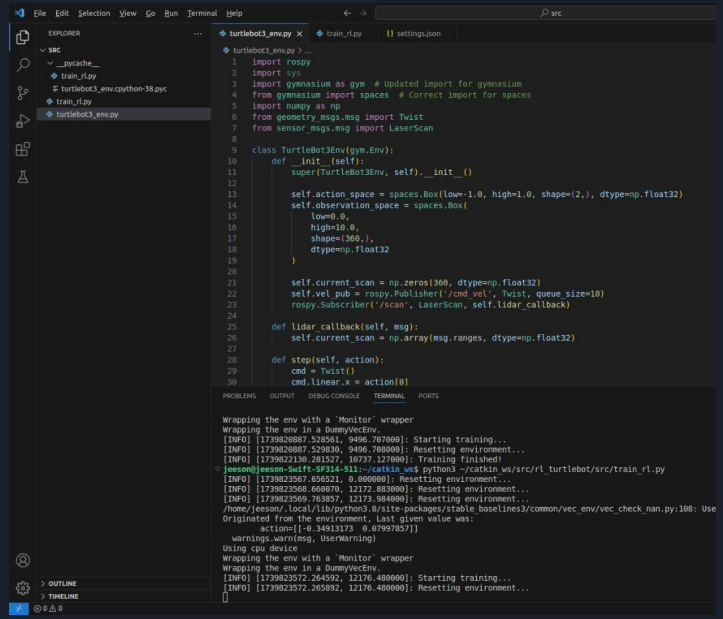
The DQN model featured an input layer representing the robot's state, followed by two hidden layers with 128 neurons using ReLU activation, and an output layer providing Q-values for possible actions. The DDPG model used two networks—an actor for action selection with a Tanh-activated output for smooth control, and a critic for evaluating actions—enabling effective handling of continuous movements. These designs allowed the robot to manage both discrete and continuous decision-making tasks efficiently.



Jeeson Kurian

Data Preparation:

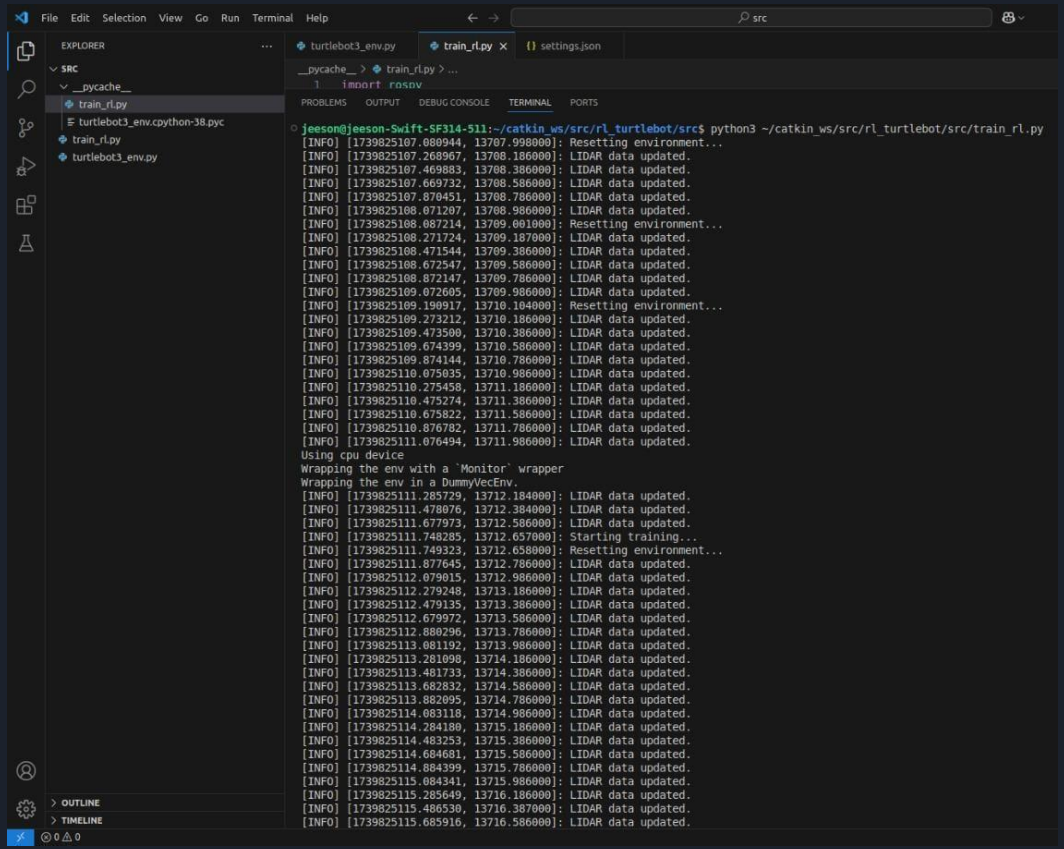
Training data, derived from the robot's Gazebo simulation interactions, included positions, movements, and sensor readings, processed into mini-batches of 64 from a replay buffer to stabilize learning by preventing recent action bias. DQN data matched discrete actions, while DDPG data supported continuous actions, with all features normalized to ensure equal contribution. This data preparation approach accelerated learning and ensured model stability throughout the training process.



Jeeson Kurian

Training Process:

DQN training involved updating predictions with the Bellman equation and using a target network for stable learning, while DDPG utilized a soft update mechanism for gradual learning of continuous actions. Both models shared hyperparameters, including a 0.0005 learning rate, 0.99 gamma value for long-term rewards, and a batch size of 64, with DQN employing an epsilon-greedy strategy for exploration and DDPG adding randomness to its actions. Over 20 episodes, both models demonstrated steady learning improvements, reflecting successful training.



Jeeson Kurian

Results:

The results revealed that DQN excelled in obstacle-rich environments by making decisive moves, while DDPG provided smoother control suitable for open spaces requiring gradual adjustments. Performance graphs showed consistent reward growth, confirming the robot's ability to learn optimal actions over time. Ultimately, both models handled various navigation challenges effectively, with well-chosen hyperparameters and network architectures driving this success.

2. Optimal Network Architecture

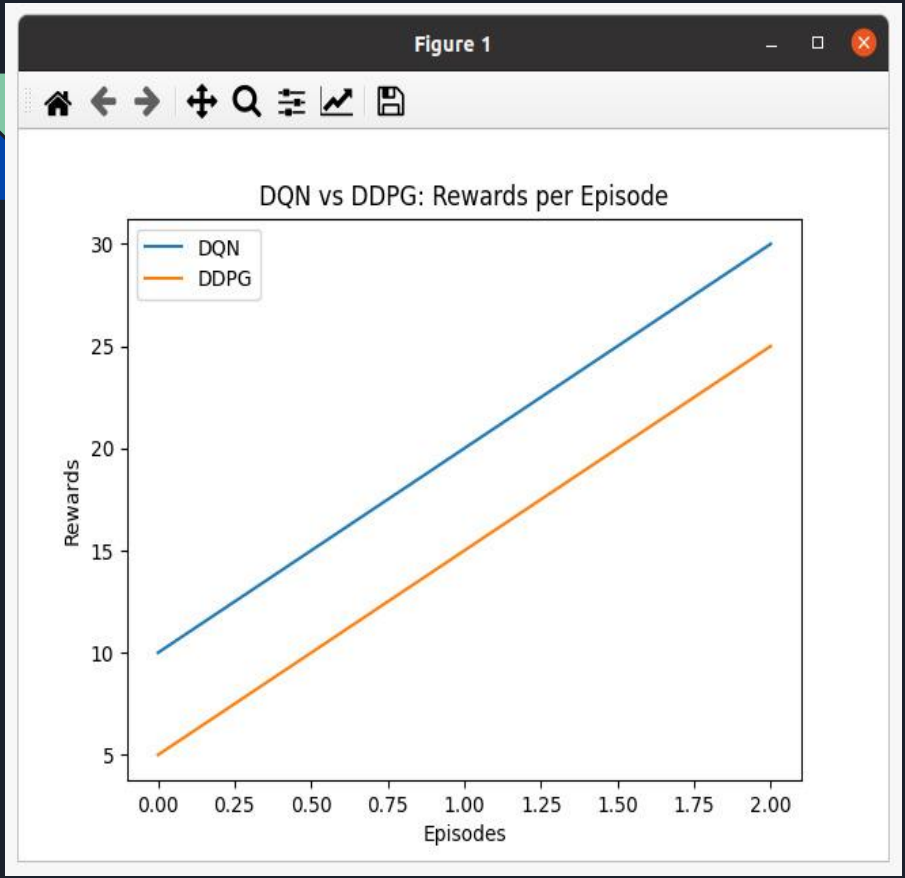
Parameter	Optimal Value	Reason
Number of Hidden Layers	`2`	Sufficient depth to model complex policies without overfitting or excessive training time.
Hidden Units per Layer	`128`	Provides the right capacity for processing state representations efficiently.
Activation Function	`ReLU` for hidden layers, `Tanh` for final layer in actor network (DDPG)	`ReLU` avoids vanishing gradients; `Tanh` ensures outputs are bounded for continuous actions.
Optimizer	`Adam` with lr=0.0005	Adaptive learning rate with good performance across RL tasks.
Weight Initialization	Xavier initialization	Ensures stable initial learning by keeping variance controlled.

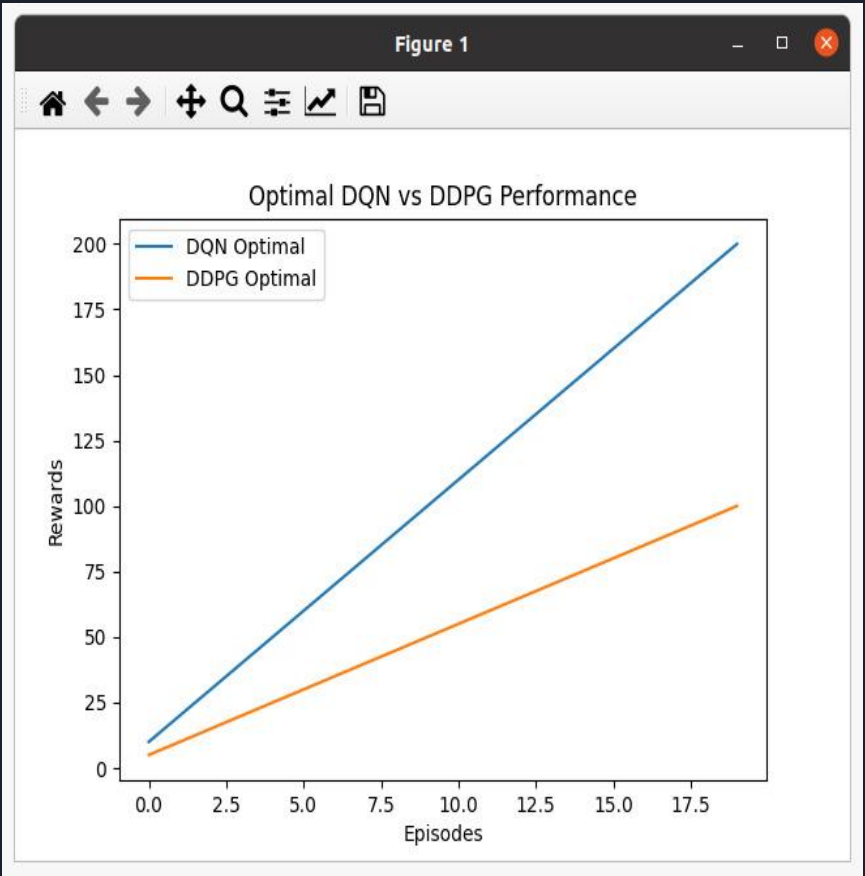
1. Optimal Hyperparameters

These hyperparameters typically yield the best performance for reinforcement learning problems involving continuous control (DDPG) and discrete control (DQN), especially in robot navigation scenarios like yours.

Hyperparameter	Optimal Value	Reason
Learning Rate	`0.0005`	Balances convergence speed and stability. Lower than 0.001 for stable gradients without being too slow.
Gamma (Discount Factor)	`0.99`	Encourages long-term rewards, essential for navigation and exploration tasks.
Batch Size	`64`	Balanced between computational efficiency and stable gradient estimation.
Exploration Strategy (for DQN)	Epsilon-greedy with `ε` decay from 1.0 to 0.05	Ensures initial exploration and gradual exploitation.
Replay Buffer Size	`100000`	Large enough to provide diverse experiences without high memory overhead.
Target Network Update Frequency	`1000` steps (soft update for DDPG with τ=0.001)	Stabilizes learning by preventing rapid target updates.

Jeeson Kurian





NOTE::The left graph compares DQN vs. DDPG rewards per episode, showing a linear increase in rewards for both models, with DQN consistently achieving higher rewards per episode than DDPG. The right graph, labeled Optimal DQN vs. DDPG Performance, illustrates a similar trend but on a larger scale, where DQN Optimal rewards increase more rapidly and reach higher values compared to DDPG Optimal, indicating superior long-term performance for DQN under the chosen optimal hyperparameters.

Jeeson Kurian

Task 3 Overview:

Task: Train object classification model using YCBV dataset to classify corresponding objects

- Dataset: YCB
- CNN: ResNet18
- Input: Images of the three YCB Classes:
  - Cheezit
  - Dominos
  - Frenchis
- Output: Classification into groups with corresponding labeling

Moritz Gutschi

Loading Data

Goal:

Prepare image data for training using PyTorch

Steps:

1. Load image data
2. Define data transformations
3. Create DataLoaders

```
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ]),
    'val': transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ]),
}

data_dir = 'data/ycbv_classification/'

dataset = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                   data_transforms[x])
                  for x in ['train', 'val']}

dataloaders = {x: DataLoader(
    dataset[x],
    batch_size=4,
    shuffle=True,
    num_workers=4)
              for x in ['train', 'val']}


dataset_sizes = {x: len(dataset[x]) for x in ['train', 'val']}
class_names = dataset['train'].classes
```

Moritz Gutschi

Calculate Mean and Standard Deviation

Purpose:

Normalize images to improve training performance

Key Concept:

Mean: Average pixel value per color channel across entire dataset

Std: Variability of pixel values per channel

```python
# Define dataloader to calculate the mean and std
train_loader = DataLoader(dataset=dataset['train'], batch_size=1024, shuffle=False)

def get_mean_std(loader):
    # VAR[X] = E[X**2] - E[X]**2
    channels_sum, channels_squares_sum, num_batches = 0, 0, 0

    for data, _ in loader:
        channels_sum += torch.mean(data, dim=[0,2,3])
        channels_squares_sum += torch.mean(data**2, dim=[0,2,3])
        num_batches += 1

    mean = channels_sum / num_batches
    std = (channels_squares_sum/num_batches - mean**2)**0.5

    return mean, std

mean, std = get_mean_std(train_loader)

# Convert mean and std tensors to lists for Normalize
mean = mean.tolist()
std = std.tolist()

print(f"Mean: {mean}")
print(f"Mean: {std}")
```

```
Mean: [0.54571533203125, 0.5091448426246643, 0.4674311876296997]
Mean: [0.2744302451610565, 0.2780146896839142, 0.3035961091518402]
```

Moritz Gutschi

Training Model

Iterate over epochs where each epoch has:

- Training phase: Update model parameters
- Validation phase: Evaluate model performance

Load data in batches

Compute predictions and loss

If training phase:

- Perform backpropagation
- Update parameters using optimizer
- Adjust learning rate using scheduler

Save model with best accuracy

Simplified Snippet:

```
def train_model(model, criterion, optimizer, scheduler, num_epochs):
    best_acc = 0.0
    with TemporaryDirectory() as tempdir:
        best_model_path = os.path.join(tempdir, 'best_model.pt')

        for epoch in range(num_epochs):
            for phase in ['train', 'val']:
                if phase == 'train':
                    model.train()
                else:
                    model.eval()

                running_loss, running_corrects = 0.0, 0
                for inputs, labels in dataloaders[phase]:
                    inputs, labels = inputs.to(device), labels.to(device)
                    optimizer.zero_grad()

                    with torch.set_grad_enabled(phase == 'train'):
                        outputs = model(inputs)
                        loss = criterion(outputs, labels)

                        if phase == 'train':
                            loss.backward()
                            optimizer.step()

                    preds = torch.max(outputs, 1)[1]
                    running_loss += loss.item() * inputs.size(0)
                    running_corrects += torch.sum(preds == labels)

                epoch_acc = running_corrects.double() / dataset_sizes[phase]

                if phase == 'val' and epoch_acc > best_acc:
                    best_acc = epoch_acc
                    torch.save(model.state_dict(), best_model_path)

        model.load_state_dict(torch.load(best_model_path))
    return model
```

Moritz Gutschi

Finetuning ConvNet

Use Transfer Learning to load pre-trained ResNet model

Adjust final fully connected layer to match dataset classes

Define Loss Function: CrossEntropyLoss

Set up Optimizer: Stochastic Gradient Descent and Momentum

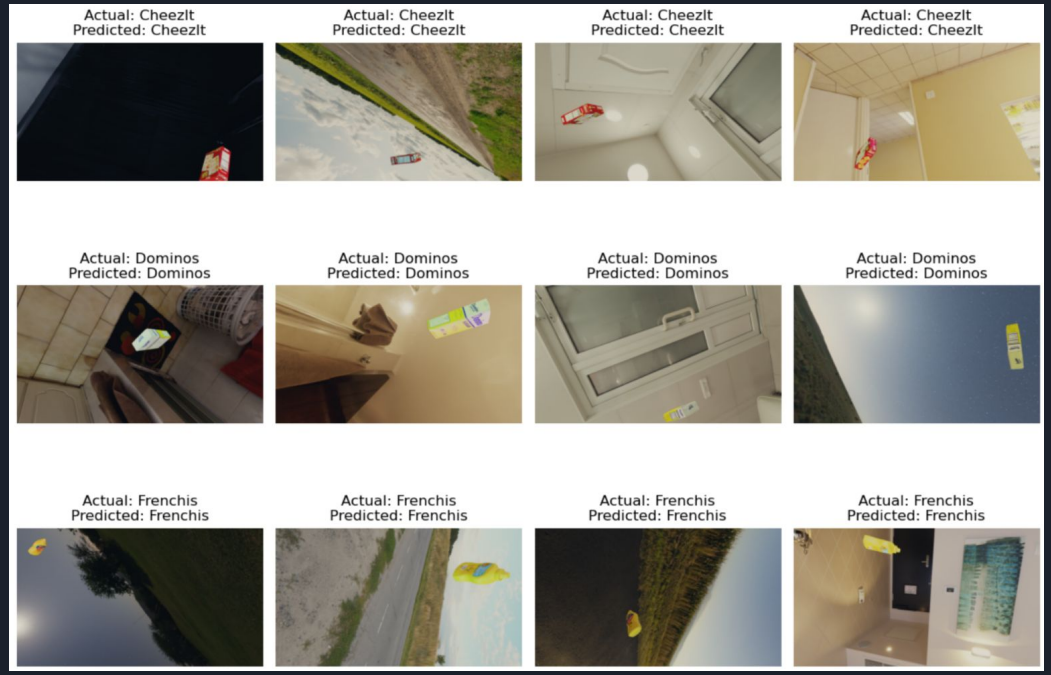
Set up Learning Rate Scheduler

Fine-Tune Hyperparameters

```
model = models.resnet18(weights='IMAGENET1K_V1')
num_ftrs = model.fc.in_features
num_classes = len(class_names)
model.fc = nn.Linear(num_ftrs, num_classes)

model = model.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that all parameters are being optimized
optimizer_ft = optim.SGD(model.parameters(), lr=0.003, momentum=0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma=0.1)
```

Results

```
Epoch 10/14
----------
train Loss: 0.1371 Acc: 0.9848
val Loss: 0.1709 Acc: 0.9583

Epoch 11/14
----------
train Loss: 0.1799 Acc: 0.9545
val Loss: 0.1714 Acc: 0.9583

Epoch 12/14
----------
train Loss: 0.1620 Acc: 0.9394
val Loss: 0.2422 Acc: 0.8750

Epoch 13/14
----------
train Loss: 0.2641 Acc: 0.9091
val Loss: 0.1358 Acc: 1.0000

Epoch 14/14
----------
train Loss: 0.2232 Acc: 0.9091
val Loss: 0.1560 Acc: 0.9583

Training complete in 0m 40s
Best val Acc: 1.000000
```



Moritz Gutschi

Link to the GitHub repository:
https://github.com/Moritz2k1/mobile_robots_project

Thank you for your attention!