Curriculum Learning – Group K

This is the homework assignment for Curriculum Learning for group K.

We will show the reward function used to compute the factorized reward. The different rewards used to compute the final one are collision avoidance, progress toward the goal, smooth steering, time to target and goal reaching.

We uploaded the turtlebot3_world.py file, in which the goal sampling curriculum is also implemented.

Reward Function

```
def _compute_reward(self, observations, done):
        # --- 1. State and Distance ---
        x, y, _ = self._get_robot_pose()
        dist = self._distance_to_goal(x, y)

        if self.prev_dist is None:
            self.prev_dist = dist

        # --- 2. Compute factorized components ---

        # A. Progress Reward: based on distance difference (Reward Shaping)
        # If diff > 0 the robot got closer, if < 0 it got farther
        diff = self.prev_dist - dist
        r_progress = self.w_progress * diff

        # B. Safety / Collision Avoidance
        # Compute min valid laser scan
        valid_scan = [o for o in observations if not numpy.isinf(o) and not
numpy.isnan(o)]
        min_scan = min(valid_scan) if len(valid_scan) > 0 else 10.0

        r_collision_avoid = 0.0
        threshold_safe = 0.3 # meters
        if min_scan < threshold_safe:
            # Linear penalty: the closer it is, the more negative the reward
            r_collision_avoid = -self.w_collision * (threshold_safe - min_scan)
        else:
            r_collision_avoid = 0.1 * self.w_collision  # Small reward for being in
a safe zone

        # C. Smooth Steering
        # Penalize if last action was not FORWARDS
        r_smooth = 0.0
```

```
        if self.last_action != "FORWARDS":
            r_smooth = -self.w_smooth

        # D. Terminal Rewards
        r_terminal = 0.0
        if done:
            if self.reached_goal:
                r_terminal = self.terminal_goal
            elif self.steps >= self.max_steps_per_episode:
                r_terminal = self.terminal_timeout
            else:
                # If done is True but not goal or timeout, it's a collision
                r_terminal = self.terminal_crash

        # E. Time Rewards
        r_cumm_time = self.r_time * self.steps

        # --- 4. Final computation and update ---
        reward = r_progress + r_cumm_time  + r_smooth + r_collision_avoid +
r_terminal

        # Update cumulative reward components
        self.cum_r_progress += r_progress
        self.cum_r_time += self.r_time
        self.cum_r_smooth += r_smooth
        self.cum_r_collision_avoid += r_collision_avoid
        self.cum_r_terminal += r_terminal
        # Update state variables for next step
        self.prev_dist = dist
        self.prev_action = self.last_action

        # --- 5. Logging structure ---
        self.last_reward_components = {
            "r_progress": float(r_progress),
            "r_time": float(self.r_time),
            "r_smooth": float(r_smooth),
            "r_collision_avoid": float(r_collision_avoid),
            "r_terminal": float(r_terminal),
            "dist": float(dist),
            "min_scan": float(min_scan),
            "goal_x": float(self.goal_xy[0]),
            "goal_y": float(self.goal_xy[1]),
            "r_total": float(reward),
            "cum_r_progress": self.cum_r_progress,
            "cum_r_time": self.cum_r_time,
            "cum_r_smooth": self.cum_r_smooth,
            "cum_r_collision_avoid": self.cum_r_collision_avoid,
            "cum_r_terminal": self.cum_r_terminal,
            "cumulated_reward": self.cumulated_reward,
        }
```

```
        self.cumulated_reward += reward
        self.steps += 1

        print("Step Reward: {:.2f} | Components: {}".format(
            reward, self.last_reward_components))

        return reward
```

Tests

Run 1

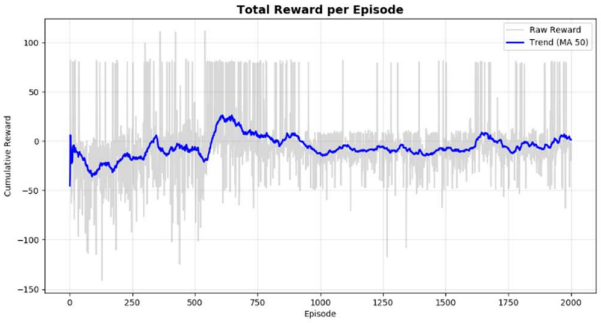
Deep learning model

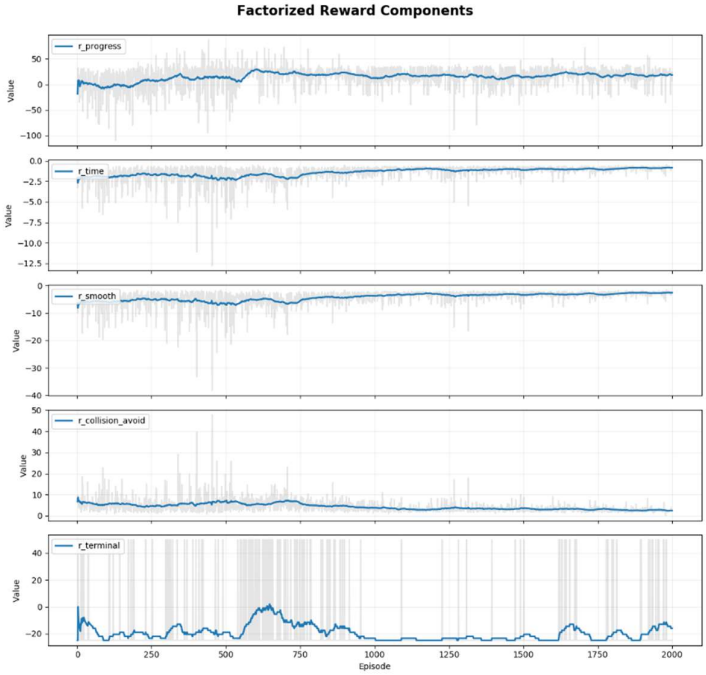
```
def __init__(self, inputs, outputs):
    super(DQN, self).__init__()
    self.fc1 = nn.Linear(inputs, 64)
    self.fc2 = nn.Linear(64, 128)
    self.fc3 = nn.Linear(128, 64)
    self.head = nn.Linear(64, outputs)
```

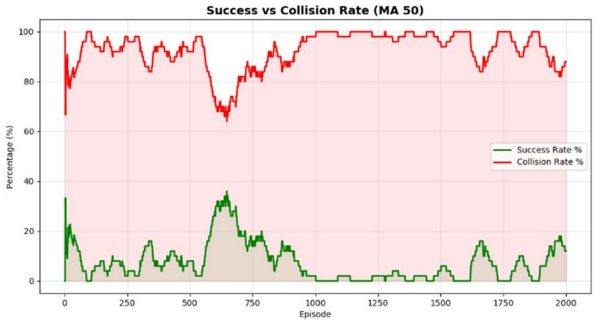
Rewards weights and values

Reward	Value
w_progress	20.0
w_collision	2.0
w_smooth	0.15
terminal_goal	100.0
terminal_crash	-25.0
terminal_timeout	-10.0
r_time	-0.05

Performance metrics







Run 2

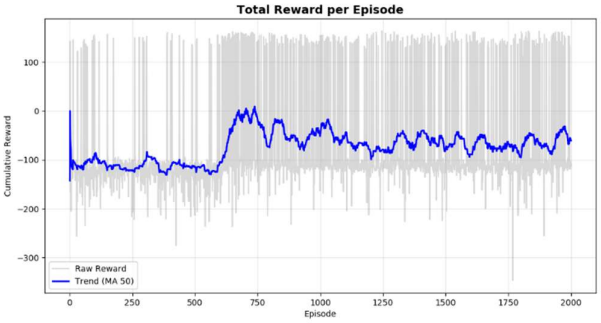
Deep learning model

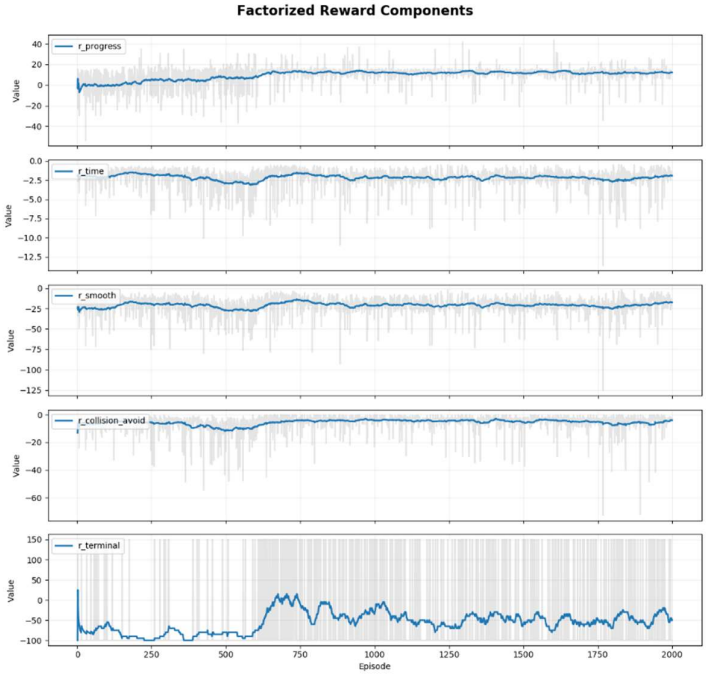
```
def __init__(self, inputs, outputs):
    super(DQN, self).__init__()
    self.fc1 = nn.Linear(inputs, 64)
    self.fc2 = nn.Linear(64, 128)
    self.fc3 = nn.Linear(128, 64)
    self.head = nn.Linear(64, outputs)
```

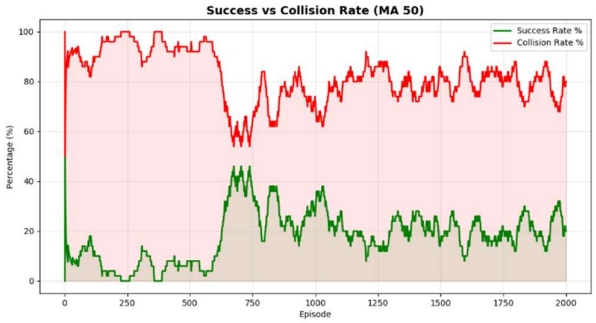
Rewards weights and values

Reward	Value
w_progress	20.0
w_collision	10.0
w_smooth	0.1
terminal_goal	150.0
terminal_crash	-100.0
terminal_timeout	-10.0
r_time	-0.05

Performance metrics







Run 3

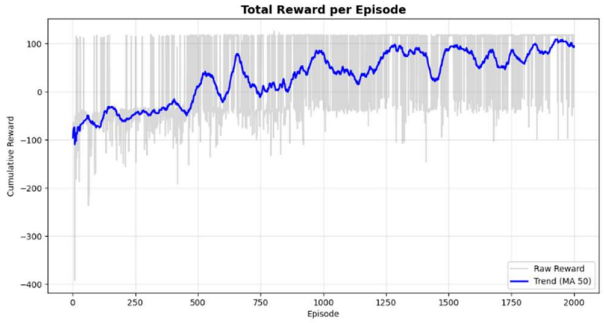
Deep learning model

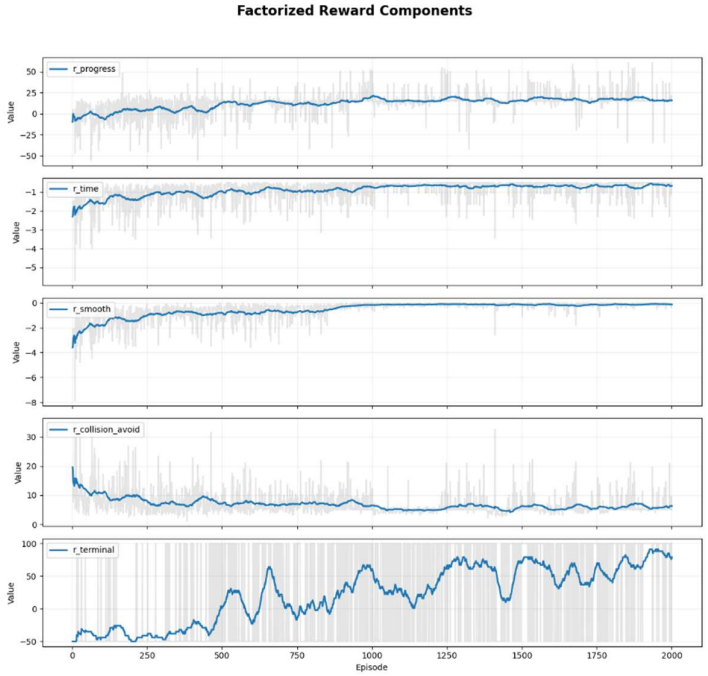
```
def __init__(self, inputs, outputs):
    super(DQN, self).__init__()
    self.fc1 = nn.Linear(inputs, 64)
    self.fc2 = nn.Linear(64, 128)
    self.fc3 = nn.Linear(128, 256)
    self.fc4 = nn.Linear(256, 64)
    self.head = nn.Linear(64, outputs)
```

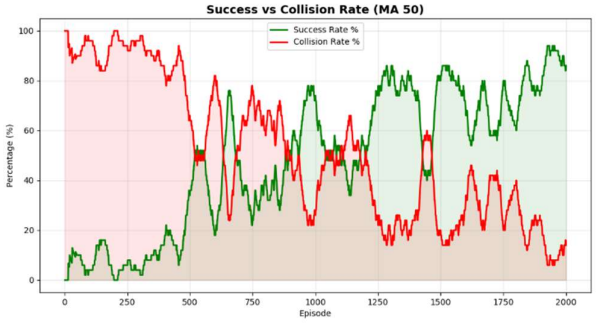
Rewards weights and values

Reward	Value
w_progress	20.0
w_collision	5.0
w_smooth	0.1
terminal_goal	100.0
terminal_crash	-50.0
terminal_timeout	-10.0
r_time	-0.05

Performance metrics







Run 4

Deep learning model

```python
class ResidualBlock(nn.Module):
    def __init__(self, dim: int):
        super().__init__()
        self.ln1 = nn.LayerNorm(dim)
        self.fc1 = nn.Linear(dim, dim)
        self.ln2 = nn.LayerNorm(dim)
        self.fc2 = nn.Linear(dim, dim)

    def forward(self, x):
        h = self.fc1(F.relu(self.ln1(x)))
        h = self.fc2(F.relu(self.ln2(h)))
        return x + h

class DQN(nn.Module):
    def __init__(self, inputs, outputs, dim: int = 128, n_blocks: int = 4):
        super(DQN, self).__init__()
        self.stem = nn.Sequential(
            nn.Linear(inputs, dim),
            nn.ReLU(),
        )
        self.blocks = nn.Sequential(*[ResidualBlock(dim) for _ in range(n_blocks)])
        self.head = nn.Linear(dim, outputs)

    # Called with either one element to determine next action, or a batch
    # during optimization. Returns tensor([[left0exp,right0exp]...]).
    def forward(self, x):
        x = x.to(device)
        x = self.stem(x)
        x = self.blocks(x)
        return self.head(x)
```

Rewards weights and values

Reward	Value
w_progress	20.0
w_collision	5.0
w_smooth	0.1
terminal_goal	100.0
terminal_crash	-50.0
terminal_timeout	-10.0
r_time	-0.05

Performance metrics