CONSTRUCTOR
UNIVERSITY

Thesis Presentation by Jan Steinmüller

SAFE-RL DUCKIETOWN

TABLE OF CONTENTS

TABLE OF CONTENTS

PROPOSITION

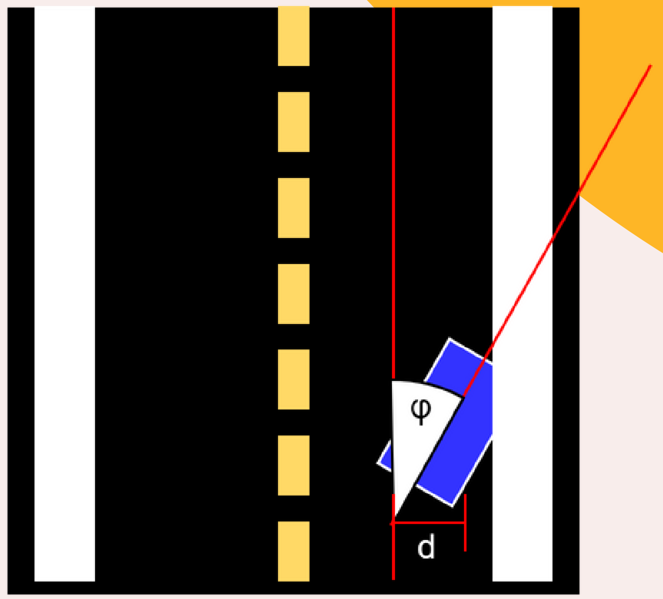
WHY DO WE NEED SAFE REINFORCEMENT LEARNING?

- Reinforcement Learning does not need training data
- Model-free reinforcement learning algorithms
- Agent should not get stuck in local maxima because of the randomness factor
- Can adapt to different environments
- Starts out with completely random actions that could be unsafe
- Will only learn that an action is bad after it is executed

PLAN

HOW CAN THIS BE IMPLEMENTED

- Goal is lane following
- Should stay in its own lane at all times
- Reward can be based on distance and angle to the lane center
- Deep-Q-Networks are the deep reinforcement learning agent
    - Continuous state space
- Input is the lane pose given by the lane_filter node
    - lane_d is the distance to the center of the lane
    - lane_phi is the angle in radians
    - buffered and averaged in this implementation
- Safety layer between action selection and execution



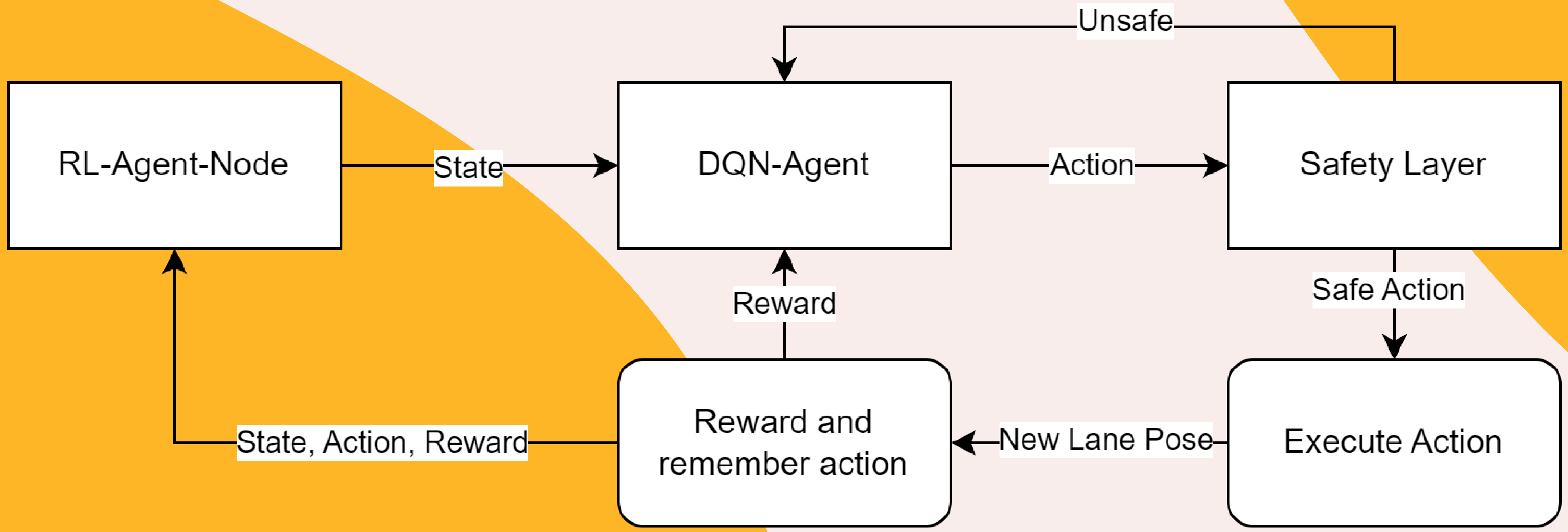
DEEP-Q-NETWORKS

HOW DOES DQN WORK?

- DQN is an extension of Q-learning
    - Q-learning has a discrete state table and action list
    - For a given state each action has a Q-Value
    - Q-Values are a combination of the expected reward and the long-term reward
      $Q\_Value(state, action) = reward + discount\_factor * Q\_Value(state' + action')$
- DQN introduces a neural network at the state input
    - Improves the scalability of the input layer
    - Continous input data
    - More resistance against noisy data
    - Replay data from executed actions to train the neural network
    - $reward = 1 - (3 * lane\_d^2) - lane\_\varphi^2$

IMPLEMENTATION

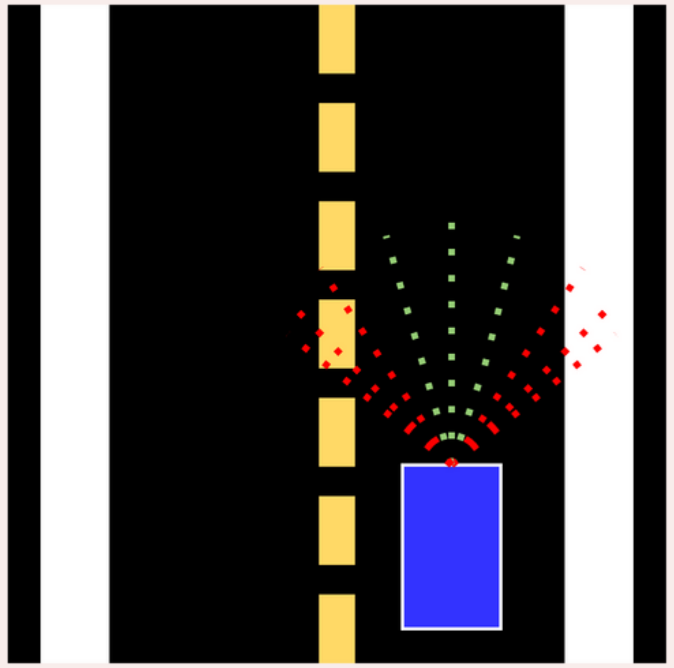
HOW WAS IT IMPLEMENTED?



Safety Layer

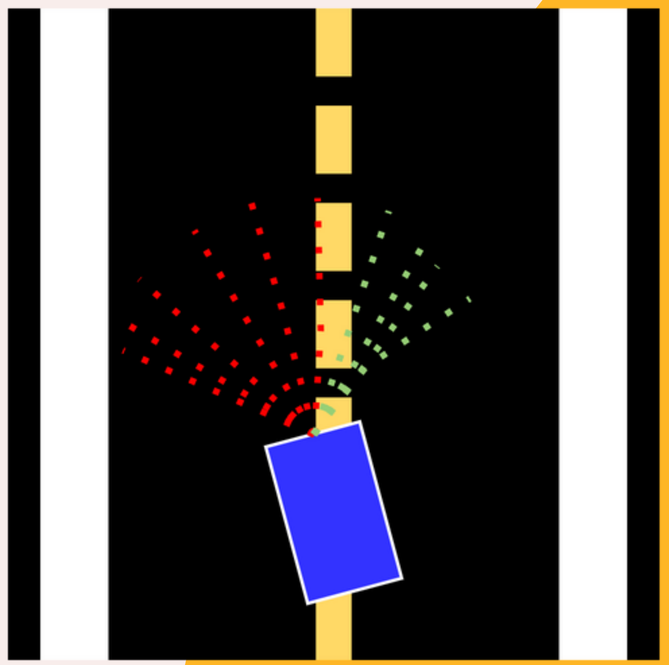
1.

Predict the future state with a data driven model when selecting an action



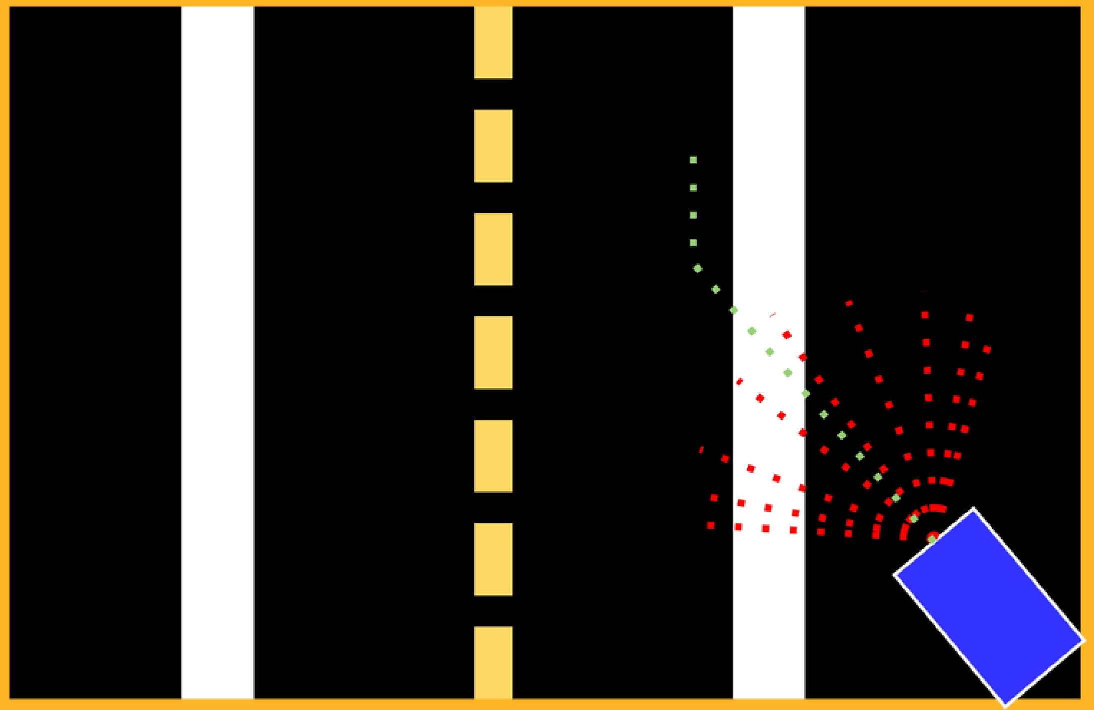
2.

Check if the action is safe and either optimize the action or consider another action if it is unsafe



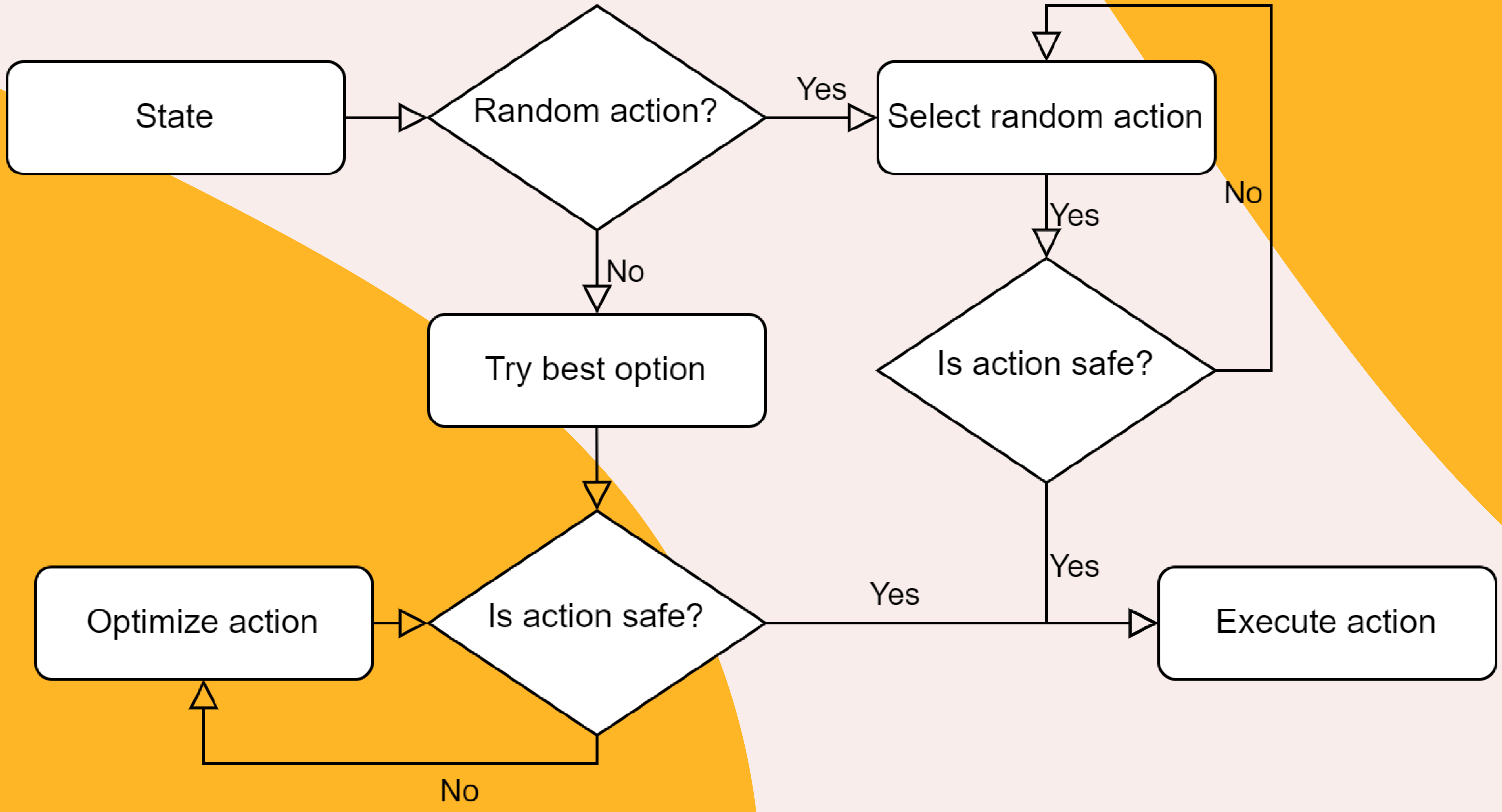
3.

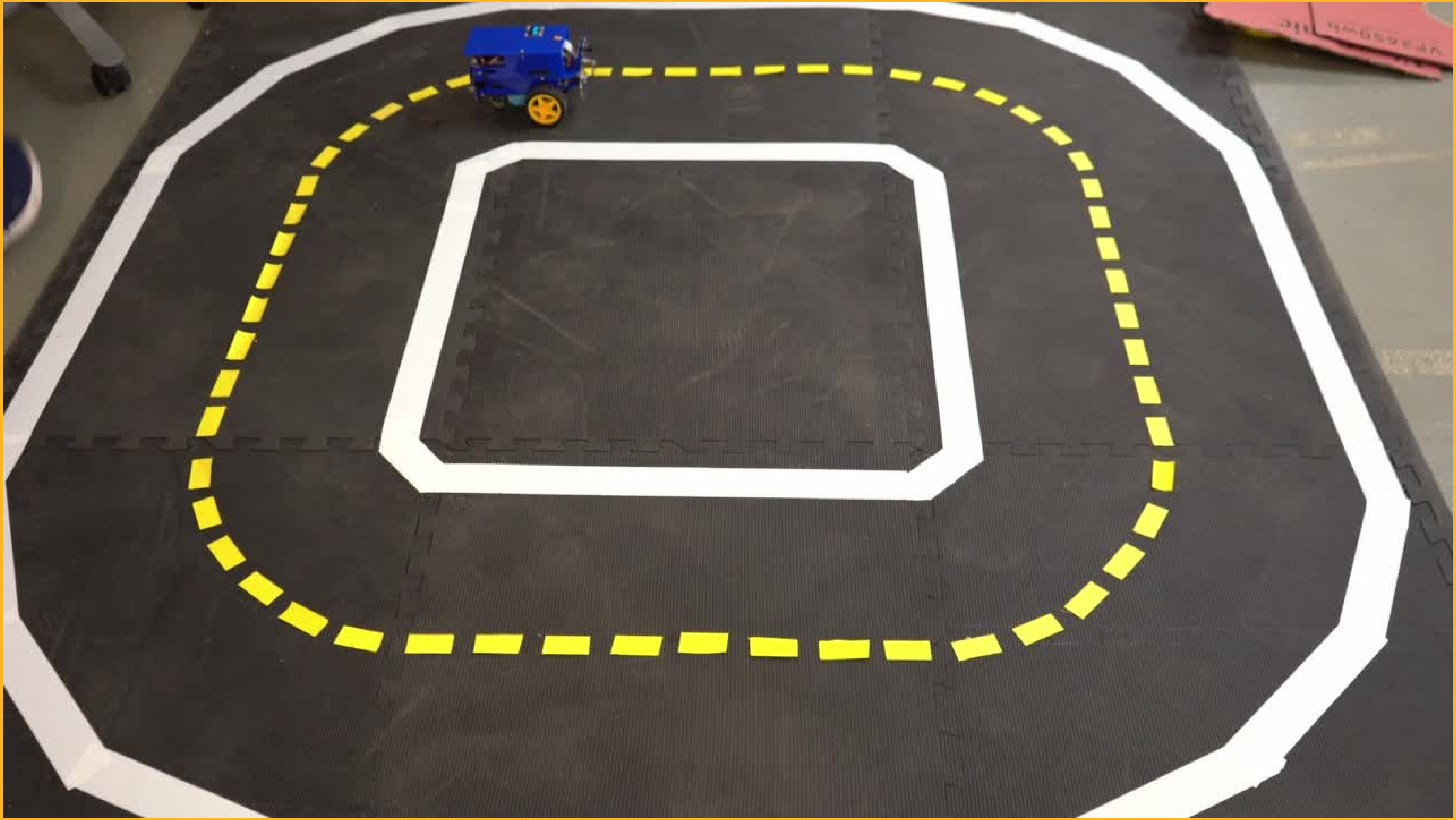
Recover from unsafe states and learning which unsafe action got us there



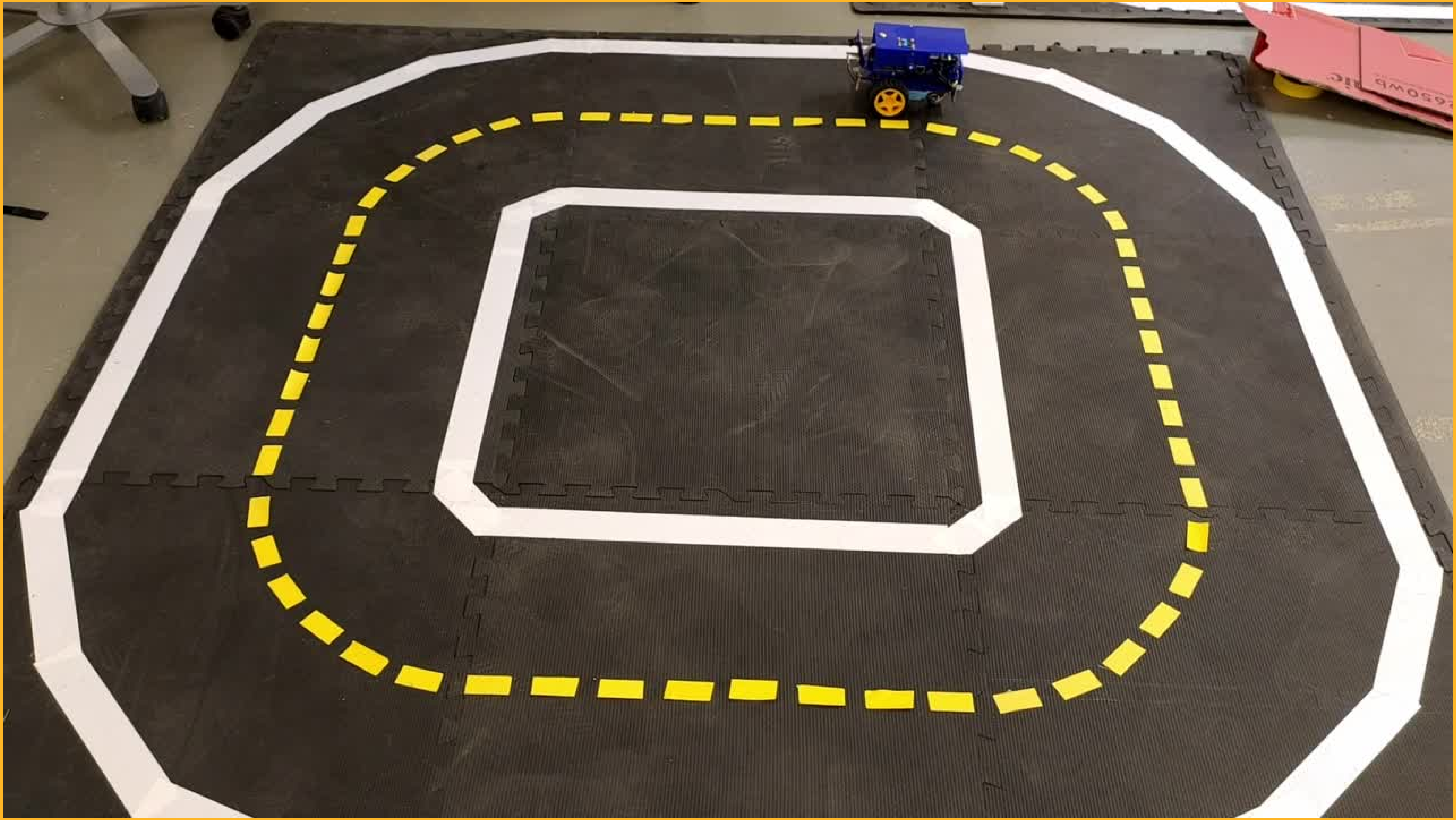
ACT FUNCTION

HOW ARE ACTIONS SELECTED





Baseline with no safety layer



DQN with safety layer

RESULTS

WHAT DO WE LEARN FORM THE VIDEOS?

- There is a clear safety performance difference
- The safety layer is not perfect in the real world
- The robot mostly manages to recover by itself
  - When the lane gets out of the robot's view, it cannot recover
- The pose estimation by the linear regression model is not perfect
- The robot cannot execute too small velocities
- Too fast velocities make the robot overshoot easily
- The robot has trouble in corners
  - Most likely to the yellow lines being out of view
- Safety layer does not seem to impact execution time too much

IMPROVEMENTS

WHAT COULD BE ADJUSTED OR IMPROVED?

- Adjust the safe lane_d distance
- Adjust angle and distance multiplier in reward
- Modify input data to the agent
- Try out different models for the pose estimation

Any Questions?

ACT FUNCTION

Algorithm 1 act

```
function ACT(self, state)
    if np.random.rand() ≤ self.epsilon then
        safe ← False
        action_list ← self.actions.copy()
        while not safe do
            action ← random.randrange(len(action_list))
            predicted_state ← self.predict_state_lr(state, self.actions[action])
            safe ← self.safety_layer.check_safety(predicted_state)
            action_list ← action_list.remove(action_list[action])
            if safe then
                Output "Random action"
                return action
            if action_list = [] or action_list = None then
                return self.get_back_to_safety(state)
    q_values ← self.model.predict(state, verbose = 0)
    action_list ← self.actions.copy()
    // sorting actions by q_values from high to low
    action_list ← [x for _, x in sorted(zip(q_values[0], action_list), reverse =
True)]
    action_list ← action_list[:5]
    for action in action_list do
        predicted_state ← self.predict_state_lr(state, action)
        safe ← self.safety_layer.check_safety(predicted_state)
        if safe then
            Output "Learned Action"
            return self.actions.index(action)
        else
            self.iter ← 0
            action ← self.optimize(state, action)
            if action is not None then
                Output "Optimized Action"
                return action
    // If no safe action was found, we try to recover to a safe state
    Output "Using inverse model to get back to safety"
    return self.get_back_to_safety(state)
```

OPTIMIZE

Algorithm 1 Optimize Action

```
function OPTIMIZE(self, state, action)
    if self.iter > 3 then
        return None
    new_v ← action[0]/2
    new_omega ← action[1]/2
    new_action ← [new_v, new_omega]
    new_state ← self.predict_state_lr(state, new_action)
    Output "Predicted state: ", new_state, " with action: ", new_action,
""
    // checking if new state is safe
    safe ← self.safety_layer.check_safety(new_state)
    if safe then
        return new_action
    else
        Output "New action  is unsafe, trying again".format(new_action)
        self.iter ← self.iter + 1
        return self.optimize(state, new_action)
```

RECOVER

Algorithm 1 Get Back To Safety

```
function GET_BACK_TO_SAFETY(state)
    dt ← self.sleep_time
    // Getting back to a safe state
    // If we are angled away from the lane
    if (state[1] > 0 and state[0] > 0) or (state[1] < 0 and state[0] < 0) then
        // Turning towards lane
        theta ← -1.5 * state[1] / dt
        v ← 0
    else
        theta ← state[1] / dt
        v ← abs(state[0]) * np.sin(state[1]) * dt
    // Getting new action
    new_action ← [v, theta]
    OUTPUT "Using inverse model to get back to safety"
    return new_action
```