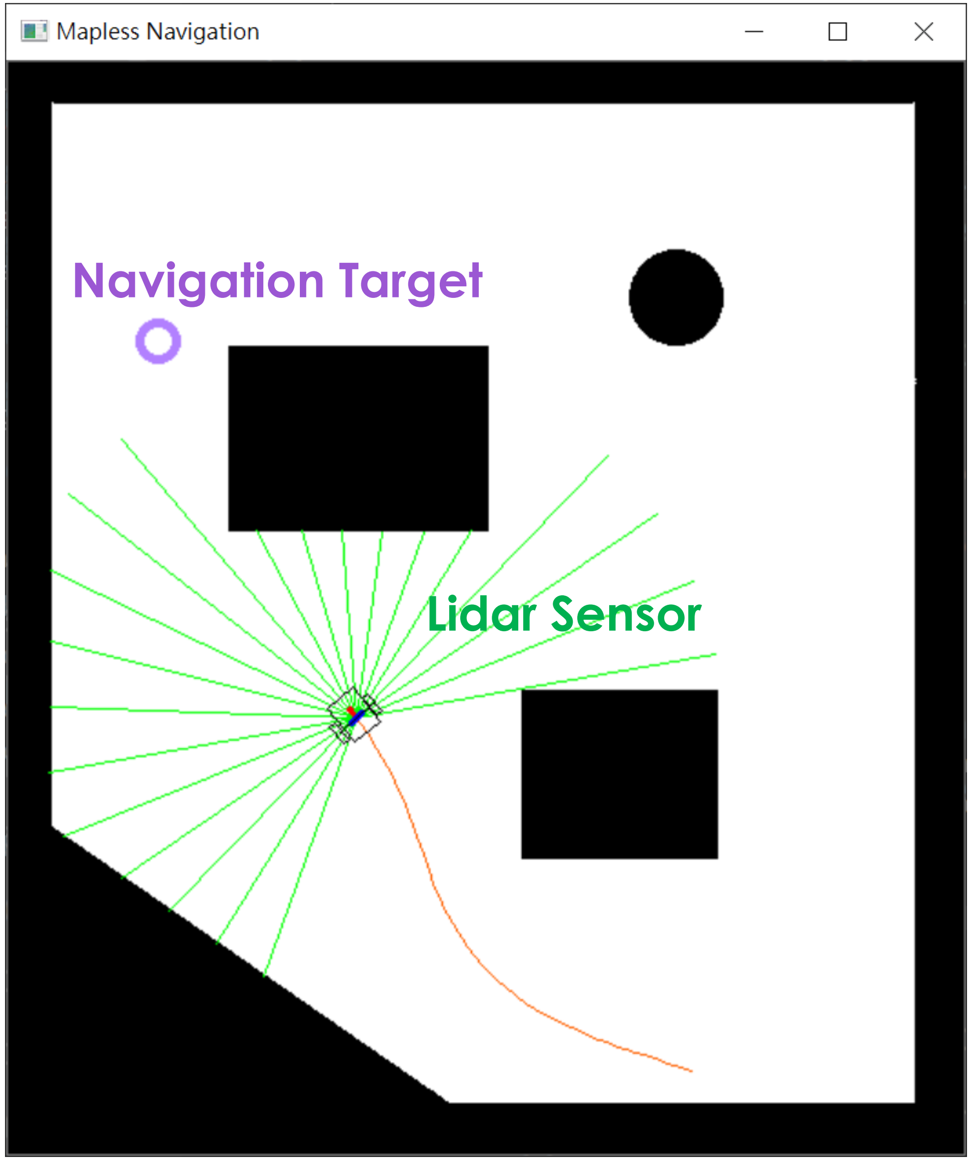
Robotic Navigation and Exploration

Lab6: Model-free RL for Mapless Navigation

Min-Chun Hu anitahu@cs.nthu.edu.tw
CS, NTHU

Mapless Navigation

- Consider the navigation task, we have a two-wheeled mobile car with lidar sensor.
- In traditional robotic methods, we have to build the map, plan the path, and tracking the path.
- As for **reinforcement learning**, we can skip those steps by learning a policy function which directly map the observation to low-level control.



In Project Folder …

- utils.py
- wmr_model.py
- lidar_model.py
- lidar_demo.py

Code for simulation.

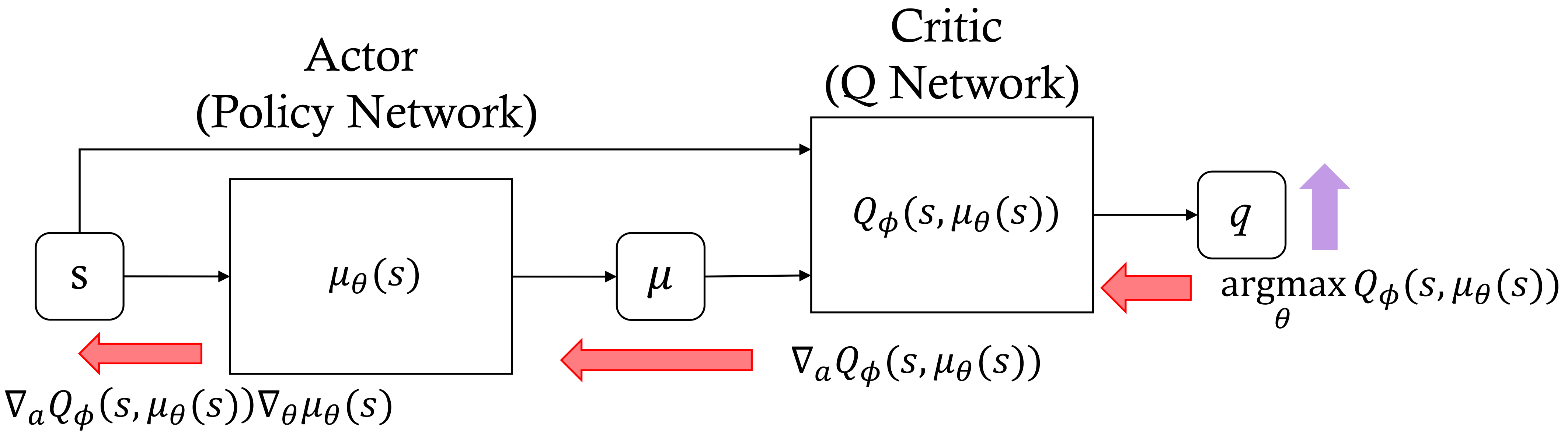
- nav_environment.py: Environment wrapper. (TODO)
- models.py: Neural network model. (TODO)
- ddpg.py: Core of reinforcement learning algorithm. (TODO)
- main_ddpg.py: Main function for training.
- eval_ddpg.py: Evaluate the trained model and generate GIF.

DQN-like Off-policy RL Workflow

main_ddpg.py

```
# Create RL and Env
RL = ddpg.DDPG(...)
env = NavigationEnv()
# Start
for eps in range(max_eps):
    state = env.initialize()
    # Run an episode
    while(True):
        # Sample data
        action = RL.choose_action(state)
        state_next, reward, done = env.step(action)
        end = 0 if done else 1
        # Store memory
        RL.store_transition(state, action, reward, state_next, end)
        env.render()
        # Optimize parameters
        loss_a, loss_c = RL.learn()
        state = state_next.copy()
        if done:
            break
```

Deterministic Policy Gradient (DPG)



Actor (Policy) Loss:

$L(\theta) = \mathbb{E}[-Q_\phi(s, \mu_\theta(s))]$

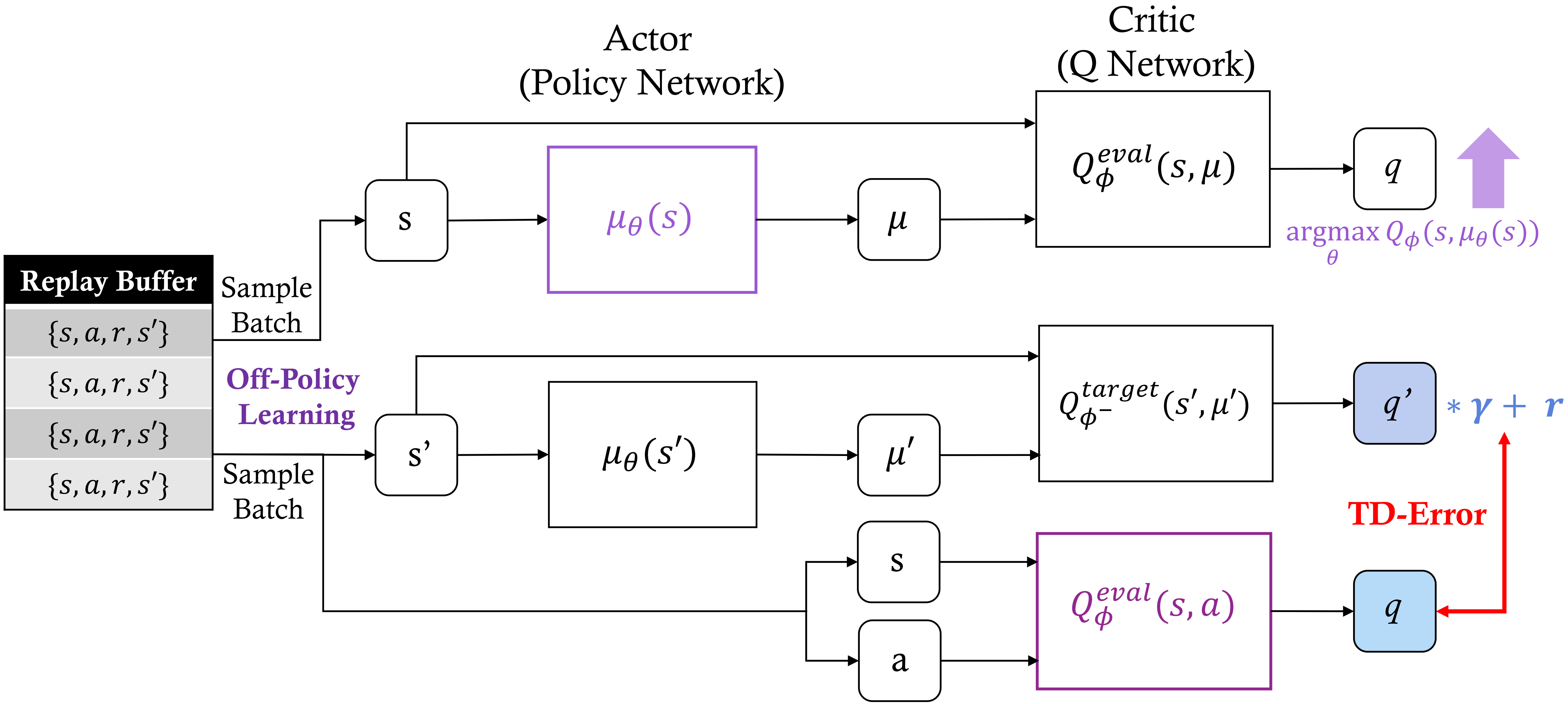
Critic (Q) Loss:

$L(\phi) = \mathbb{E}[\frac{1}{2}\left(r + \gamma Q_\phi(s', \mu(s')) - Q(s,a)\right)^2]$

TD-Error

Off-Policy Learning

Deep Deterministic Policy Gradient (DDPG)



Deep Deterministic Policy Gradient (DDPG)

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

State and Action

- State (23-d):
  - 21d sense distance + 2d relative target coordinate

- Action (2-d)
  - Velocity (-1~1 (wrapper)→ 0~60 (simulation))
  - Angular Velocity (-1~1 (wrapper)→ -45~45 (simulation))

Reward Design (Lab-01)

`nav_environment.py`

- Distance Reward:
  - The reduction of distance from car to navigate target.

```
reward_dist = self.target_dist - curr_target_dist
```

- Orientation Reward:
  - Penalty for the angle between forward direction and target direction.

```
orien = np.rad2deg(np.arctan2(self.target[1] - self.car.y, self.target[0] - self.car.x))
err_orien = (orien - self.car.yaw) % 360
if err_orien > 180:
    err_orien = 360 - err_orien
reward_orien = np.deg2rad(err_orien)
```

- Action Reward:
  - Penalty for small movement.

```
reward_act = 0.05 if action[0]<0.5 else 0
```

Reward Design (Lab-01)

`nav_environment.py`

- Total reward is the weighted sum of the above three rewards.

```
reward = w1*reward_dist - w2*reward_orien - w3*reward_act
```

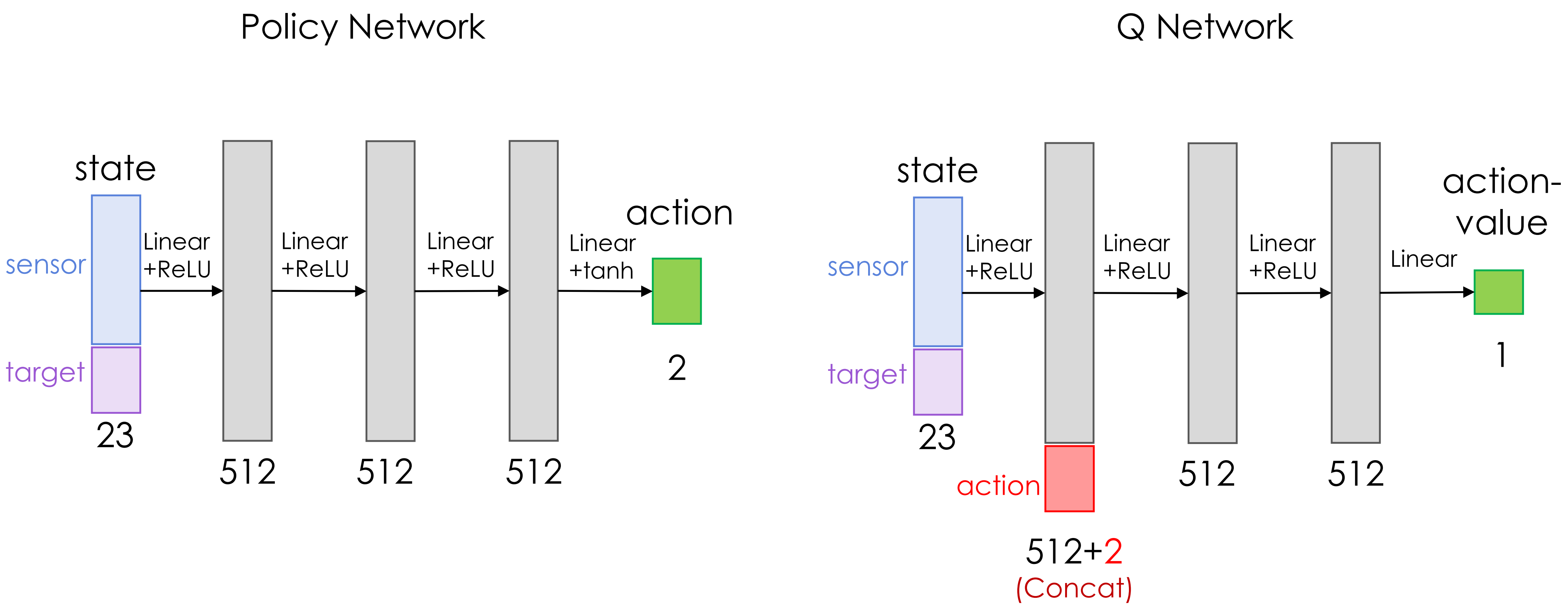
- The reward of terminate state.
  - Collision
  - Reach target

```
# Terminal State
done = False
if collision:
    # reward = ??
    done = True
if curr_target_dist < 20:
    # reward = ??
    done = True
```

Try to adjust the weighting to get better performance !!

Neural Network Model (Lab-02)

`models.py`



DDPG Class Structure

- __init__(…)
- _build_net(anet, cnet)
- save_load_model(…)
- choose_action(s, eval): Select action. (TODO)
- init_memory()
- store_transition(s,a,r,sn,end): Store the sample data
- soft_update(): Update the parameters of target network.
- learn(): Train RL model. (TODO)

Replay Buffer

ddpg.py

```
def init_memory(self):
    self.memory_counter = 0
    self.memory = {"s":[], "a":[], "r":[], "sn":[], "end":[]}

def store_transition(self, s, a, r, sn, end):
    if self.memory_counter <= self.memory_size:
        self.memory["s"].append(s)
        self.memory["a"].append(a)
        self.memory["r"].append(r)
        self.memory["sn"].append(sn)
        self.memory["end"].append(end)
    else:
        index = self.memory_counter % self.memory_size
        self.memory["s"][index] = s
        self.memory["a"][index] = a
        self.memory["r"][index] = r
        self.memory["sn"][index] = sn
        self.memory["end"][index] = end

    self.memory_counter += 1
```

Soft Update of the Target Network

ddpg.py

- Replace the parameter smoothly for training stability.

$$\theta_{target} \leftarrow (1-\tau)\theta_{target} + \tau\theta_{eval}$$

```
def soft_update(self, TAU=0.01):
    # Store sample to replay buffer
    with torch.no_grad():
        for targetParam, evalParam in zip(self.critic_target.parameters(), self.critic.parameters()):
            targetParam.copy_((1 - self.tau)*targetParam.data + self.tau*evalParam.data)
```

Choose Action (Lab-03)

ddpg.py

- Apply an decay epsilon noise for exploration.

```
epsilon_params = [1.0, 0.5, 0.00001], # init var / final var / decay
```

```
def choose_action(self, s, eval=False):
    s_ts = torch.FloatTensor(np.expand_dims(s,0)).to(device)
    action = self.actor(s_ts)
    action = action.cpu().detach().numpy()[0]

    if eval == False: # Use epsilon
        action += np.random.normal(0, self.epsilon, action.shape)
    else: # Use final variance
        action += np.random.normal(0, self.epsilon_params[1], action.shape)

    action = np.clip(action, -1, 1)
    return action
```

Learn (Lab-04)

ddpg.py

- Construct the torch tensor and update to GPU.

```
# Construct torch tensor
s_ts = torch.FloatTensor(np.array(s_batch)).to(device)
a_ts = torch.FloatTensor(np.array(a_batch)).to(device)
r_ts = torch.FloatTensor(np.array(r_batch)).to(device).view(self.batch_size, 1)
sn_ts = torch.FloatTensor(np.array(sn_batch)).to(device)
end_ts = torch.FloatTensor(np.array(end_batch)).to(device).view(self.batch_size, 1)
```

Critic (Q) Loss:

$$L(\phi) = \mathbb{E}[\frac{1}{2}\left(r + \gamma Q_{\phi^-}(s', \mu(s')) - Q_\phi(s, a)\right)^2]$$

Learn (Lab-05)

ddpg.py

- Compute critic loss and optimize

```
# TD-target
with torch.no_grad():
    a_next = self.actor(sn_ts)
    q_next_target = self.critic_target(sn_ts, a_next)
    q_target = r_ts + end_ts * self.gamma * q_next_target

# Critic loss
q_eval = self.critic(s_ts, a_ts)
self.critic_loss = self.criterion(q_eval, q_target)

self.critic_optim.zero_grad()
self.critic_loss.backward()
self.critic_optim.step()
```

Learn (Lab-06)

$$\textit{Actor (Policy) Loss}: \\ L(\theta) = \mathbb{E}[-Q_\phi(s, \mu_\theta(s))]$$

`ddpg.py`

- Compute actor loss and optimize

```
# Actor loss
a_curr = self.actor(s_ts)
q_current = self.critic(s_ts, a_curr)
self.actor_loss = -q_current.mean()

self.actor_optim.zero_grad()
self.actor_loss.backward()
self.actor_optim.step()
```

Learn (Lab-07)

ddpg.py

- Update target network and epsilon noise

```
self.soft_update()
if self.epsilon > self.epsilon_params[1]:
    self.epsilon -= self.epsilon_params[2]
else:
    self.epsilon = self.epsilon_params[1]
```

Run on Google Colab

- You can directly run the `main_ddpg.py` if you have computing resource, otherwise you can run the `main_ddpg.ipynb` on Google Colab.

- Put the whole project folder to your google drive and ensure the project path is correct.

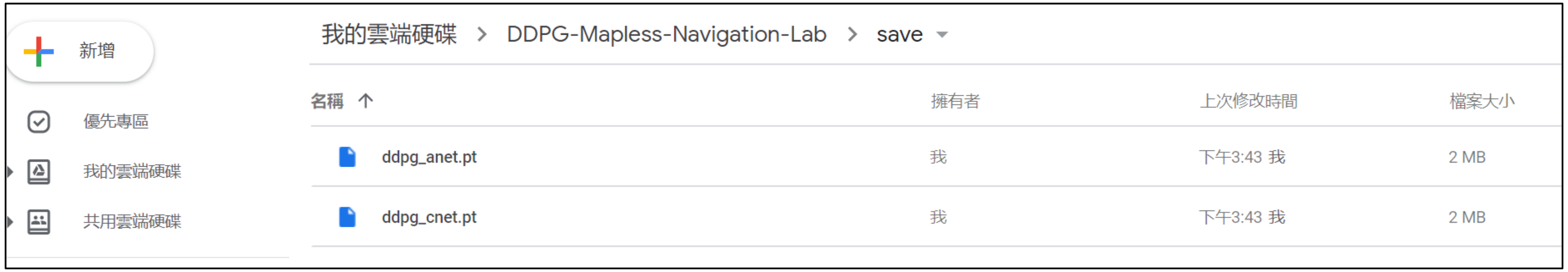
```
import sys
project_root = '/content/drive/My Drive/DDPG-Mapless-Navigation-Lab/'
sys.path.append(project_root)
```

- Make sure the render is **False** because Google Colab cannot handle the GUI in openCV.

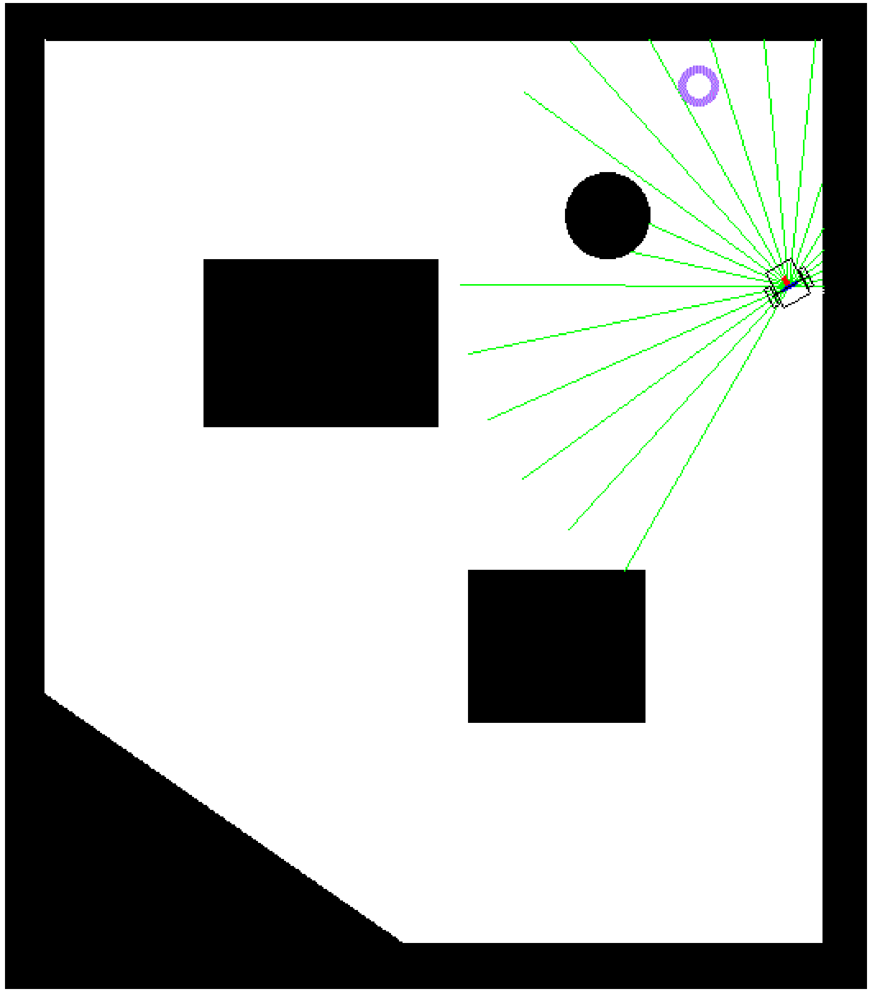
```
is_train = True
render = False
load_model = False
```

Run on Google Colab

- The parameters will store in "**save/**" and the GIF will store in "**out/**" during training.



Result Demo



Q&A