CODE TEAM INFINITY

MAIN.PY

```
import numpy as np
import torch
import argparse
import os

from TD3 import TD3
from replay_buffer import ReplayBuffer, PrioritizedReplayBuffer

import laserhockey.hockey_env as h_env


def eval_policy(policy, seed, max_episode_timesteps, eval_episodes=20):
    # Set up the evaluation environment
    eval_env = h_env.HockeyEnv(mode=h_env.HockeyEnv.NORMAL)
    eval_env.seed(seed + 100)
    player2 = h_env.BasicOpponent(weak=False)

    # Evaluate
    eval_rewards = []
    eval_results = []
    for _ in range(eval_episodes):
        state, done = eval_env.reset(), False
        state2 = eval_env.obs_agent_two()
        episode_reward = 0
        episode_result = 0

        for _ in range(max_episode_timesteps):
            action = policy.act(np.array(state))
            action2 = player2.act(np.array(state2))

            state, reward, done, info = eval_env.step(np.hstack([action, ac
tion2]))
            state2 = eval_env.obs_agent_two()
            episode_reward += reward + info["reward_touch_puck"] + info["re
ward_puck_direction"]
            episode_result += reward - info["reward_closeness_to_puck"]

            if done:
                break

        eval_rewards.append(episode_reward)
        eval_results.append(episode_result)

    # Print mean and std deviation of episode rewards
    print("---------------------------------------------")
```

```
    print(f"Evaluation over {eval_episodes} episodes: {np.mean(eval_rewards
):.3f} +- {np.std(eval_rewards):.3f}")
    print("---------------------------------------------")

    return eval_rewards, eval_results


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--
policy", default="TD3", help='Policy name (TD3)')
    parser.add_argument("--env", default="Hockey-
v0_NORMAL", help='Gym environment name')
    parser.add_argument("--
trial", default=0, type=int, help='Trial number')
    parser.add_argument("--
seed", default=42, type=int, help='Sets Gym, PyTorch and Numpy seeds')
    parser.add_argument("--
start_timesteps", default=5e4, type=int, help='Time steps initial random po
licy is used')
    parser.add_argument("--
eval_freq", default=5e3, type=int, help='How often (time steps) it will be
evaluated')
    parser.add_argument("--
self_play_freq", default=15e5, type=int, help='Add current agent to list of
 opponents')
    parser.add_argument("--
max_timesteps", default=1e6, type=int, help='Max time steps to run environm
ent')
    parser.add_argument("--
max_episode_timesteps", default=500, type=int, help='Max time steps per epi
sode')
    parser.add_argument("--
max_buffer_size", default=1e6, type=int, help='Size of the replay buffer')
    parser.add_argument("--
expl_noise", default=0.15, type=float, help='Std of Gaussian exploration no
ise')
    parser.add_argument("--
hidden_dim", default=256, type=int, help='Hidden dim of actor and critic ne
ts')
    parser.add_argument("--
batch_size", default=256, type=int, help='Batch size for both actor and cri
tic')
    parser.add_argument("--learning_rate", default=3e-
4, type=float, help='Learning rate')
    parser.add_argument("--
discount", default=0.99, type=float, help='Discount factor')
    parser.add_argument("--
tau", default=0.01, type=float, help='Target network update rate')
```

```
    parser.add_argument("--policy_noise", default=0.1, type=float,
                        help='Noise added to target policy during critic up
date')
    parser.add_argument("--
noise_clip", default=0.5, type=float, help='Range to clip target policy noi
se')
    parser.add_argument("--
policy_freq", default=2, type=int, help='Frequency of delayed policy update
s')
    parser.add_argument("--
prioritized_replay", action="store_true", help='Use prioritized experience
replay')
    parser.add_argument("--
alpha", default=0.6, type=float, help='Amount of prioritization in PER')
    parser.add_argument("--
beta", default=1.0, type=float, help='Amount of importance sampling in PER'
)
    parser.add_argument("--
beta_schedule", default="", help='Annealing schedule for beta in PER')
    parser.add_argument("--
normalize_obs", action="store_true", help='Use observation normalisation')
    parser.add_argument("--
only_win_reward", action="store_true", help='Rewards only wins')
    parser.add_argument("--
early_stopping", action="store_true", help='Use early stopping')
    parser.add_argument("--load_model", default="",
                        help='Model load file name, \"\" does not load')
    args = parser.parse_args()

    file_name = f"{args.policy}_{args.env}_{args.trial}"
    print("---------------------------------------------")
    print(f"Policy: {args.policy}, Env: {args.env}, Seed: {args.seed}")
    print("---------------------------------------------")

    # Create folders for evaluation
    if not os.path.exists("./results"):
        os.makedirs("./results")
    if not os.path.exists("./models"):
        os.makedirs("./models")

    # Create the Environment
    env = h_env.HockeyEnv(mode=h_env.HockeyEnv.NORMAL)
    state_dim = env.observation_space.shape[0]
    print(state_dim)
    print(env.observation_space)
    action_dim = env.action_space.shape[0] // 2  # The policy only controls
 the left player
    print(env.action_space)
```

```
    max_action = float(env.action_space.high[0])

    # Set seeds
    env.seed(args.seed)
    torch.manual_seed(args.seed)
    np.random.seed(args.seed)

    # Initialize policy
    kwargs = {
        "state_dim": state_dim,
        "action_dim": action_dim,
        "hidden_dim": args.hidden_dim,
        "max_action": max_action,
        "lr": args.learning_rate,
        "discount": args.discount,
        "tau": args.tau,
        "policy_noise": args.policy_noise * max_action,
        "noise_clip": args.noise_clip * max_action,
        "policy_freq": args.policy_freq,
        "normalize_obs": args.normalize_obs
    }
    policy = TD3(**kwargs)

    # Create other players
    opponent_policies = [h_env.BasicOpponent(weak=True), h_env.BasicOpponen
t(weak=False)]
    warm_up_player = h_env.BasicOpponent(weak=False)

    # Load previous model if applicable
    if args.load_model != "":
        policy.load(f"./models/{args.load_model}")
        warm_up_player = policy

        # Also add self to list of opponents
        old_self = TD3(**kwargs)
        old_self.load(f"./models/{args.load_model}")
        opponent_policies.append(old_self)

    # Replay buffer
    if args.prioritized_replay:
        replay_buffer = PrioritizedReplayBuffer(
            state_dim,
            action_dim,
            max_size=int(args.max_buffer_size),
            total_t=int(args.max_timesteps),
            alpha=args.alpha,
            beta=args.beta,
            beta_schedule=args.beta_schedule
```

```
        )
    else:
        replay_buffer = ReplayBuffer(state_dim, action_dim, max_size=int(ar
gs.max_buffer_size))

    # Create evaluation data structure
    evaluations = {
        'train_rewards': [],
        'eval_rewards': [],
        'eval_results': [],
        'final_eval_rewards': [],
        'final_eval_results': []
    }

    # Initialize the environment
    state, done = env.reset(), False
    state2 = env.obs_agent_two()
    player2 = np.random.choice(opponent_policies)
    episode_reward = 0
    episode_timesteps = 0
    episode_num = 0
    best_policy = {
        't': 0,
        'average_return': 0
    }

    for t in range(int(args.start_timesteps + args.max_timesteps)):

        episode_timesteps += 1

        # Select action
        if t < args.start_timesteps:
            action = warm_up_player.act(np.array(state))
            action2 = player2.act(np.array(state2))
        else:
            action = (
                policy.act(np.array(state))
                + np.random.normal(0, max_action * args.expl_noise, size=ac
tion_dim)
            ).clip(-max_action, max_action)
            action2 = player2.act(np.array(state2))

        # Perform action (composed of the policy's and the opponent's actio
n) and collect reward
        next_state, reward, done, info = env.step(np.hstack([action, action
2]))
        reward += info["reward_touch_puck"] + info["reward_puck_direction"]
        if args.only_win_reward:
```

```
            reward = 10.0 if info["winner"] == 1 else 0.0
        done_bool = float(done)

        # Store data in replay buffer
        replay_buffer.add(state, action, next_state, reward, done_bool)

        # Update the state and return
        state = next_state
        state2 = env.obs_agent_two()
        episode_reward += reward

        # Train agent after collecting sufficient data
        if t >= args.start_timesteps:
            policy.train(replay_buffer, args.prioritized_replay, args.batch
_size)

        if done or episode_timesteps >= args.max_episode_timesteps:
            # Print episode info
            print(
                f"Total T: {t + 1}, " +
                f"Episode Num: {episode_num + 1}, " +
                f"Episode T: {episode_timesteps}, " +
                f"Reward: {episode_reward:.3f}"
            )

            if t >= args.start_timesteps:
                evaluations['train_rewards'].append(episode_reward)
                episode_num += 1

            # Reset environment
            state, done = env.reset(), False
            state2 = env.obs_agent_two()
            player2 = np.random.choice(opponent_policies)
            episode_reward = 0
            episode_timesteps = 0

        # Evaluate the policy
        if (t + 1) % args.eval_freq == 0 and (t + 1) >= args.start_timestep
s:
            eval_rewards, eval_results = eval_policy(
                policy,
                args.seed,
                args.max_episode_timesteps
            )
            evaluations['eval_rewards'].append(eval_rewards)
            evaluations['eval_results'].append(eval_results)
            np.save(f"./results/{file_name}", evaluations)
```

```python
            # Early stopping
            if not args.early_stopping:
                policy.save(f"./models/{file_name}")
            elif np.mean(eval_rewards) > best_policy['average_return']:
                best_policy['t'] = t
                best_policy['average_return'] = np.mean(eval_rewards)
                policy.save(f"./models/{file_name}")

        # Add opponent
        if (t + 1) % args.self_play_freq == 0:
            opponent_policy = TD3(**kwargs)
            opponent_policy.load(f"./models/{file_name}")
            opponent_policies.append(opponent_policy)

    # Final evaluation
    if args.early_stopping:
        print("------------------------------------------------------")
        print(f"Average return of best policy: {best_policy['average_return
']:.3f} (at t = {best_policy['t']})")
        print("------------------------------------------------------")
    final_policy = TD3(**kwargs)
    final_policy.load(f"./models/{file_name}")
    eval_rewards, eval_results = eval_policy(
        final_policy,
        args.seed,
        args.max_episode_timesteps,
        eval_episodes=100
    )
    evaluations['final_eval_rewards'] = eval_rewards
    evaluations['final_eval_results'] = eval_results
    np.save(f"./results/{file_name}", evaluations)

    # Report moments for observation normalization
    # np.save("./results/observation_moments.npy", replay_buffer.observatio
n_moments())


if __name__ == "__main__":
    main()
```

SUM TREE CODE:

```python
import numpy as np
```

```
class SumTree:
    # A binary tree data structure where the parent’s value is the sum
of its children
    def __init__(self, max_size):
        self.max_size = max_size
        self.max_p = 1.

        self.tree = np.zeros(2 * max_size - 1)

    # Recursively propagate the update to the root node
    def _propagate(self, idx, delta_p):
        parent = (idx - 1) // 2

        self.tree[parent] += delta_p

        if parent != 0:
            self._propagate(parent, delta_p)

    # Find sample on leaf node
    def _retrieve(self, idx, sample_p):
        left = 2 * idx + 1
        right = left + 1

        if left >= self.tree.size:
            return idx

        if sample_p <= self.tree[left]:
            return self._retrieve(left, sample_p)
        else:
            return self._retrieve(right, sample_p - self.tree[left])

    # Return total priority
    def total_p(self):
        return self.tree[0]

    # Store priority
    def add(self, p, ptr):
        idx = ptr + self.max_size - 1
        self.update(idx, p)

    # Update priority
    def update(self, idx, p):
        delta_p = p - self.tree[idx]

        self.tree[idx] = p
        if p > self.max_p:
            print(f' -
> Current maximal priority in the replay buffer: {p:.3f}')
```

```
        self.max_p = max(p, self.max_p)
        self._propagate(idx, delta_p)

    # Get priority and data index
    def get(self, sample_p):
        idx = self._retrieve(0, sample_p)
        data_idx = idx - self.max_size + 1

        return idx, self.tree[idx], data_idx


import numpy as np
import laserhockey.hockey_env as h_env
from TD3 import TD3
```

Code cell <undefined>

```
#%% [code]
# Create the environment
env = h_env.HockeyEnv(mode=h_env.HockeyEnv.NORMAL)

# Basic opponents
weak_basic_opponent = h_env.BasicOpponent(weak=True).act
strong_basic_opponent = h_env.BasicOpponent(weak=False).act

# Rollout function
def rollout(p1, p2, num_games=10, render=False):
    counter = np.zeros(3)

    for _ in range(num_games):
        state_l, done = env.reset(), False
        state_r = env.obs_agent_two()
        while not done:
            if render:
                env.render()
            action_l = p1(state_l)
            action_r = p2(state_r)
            state_l, _, done, info = env.step(np.hstack([action_l, acti
on_r]))
            state_r = env.obs_agent_two()
        counter[info["winner"] + 1] += 1
    env.close()

    wins = counter[2]
    defeats = counter[0]
    draws = counter[1]

    return wins, defeats, draws
```

```
Code cell <undefined>
#%% [code]
# TD3 agents
def get_TD3_policy(name):
    policy = TD3(state_dim=18, action_dim=4, hidden_dim=256, max_action
=1.0, normalize_obs=True)
    policy.load(f"./models/{name}")
    return policy.act

TD3_policy = get_TD3_policy('TD3')
SP_TD3_policy = get_TD3_policy('SP-TD3')
aSP_TD3_policy = get_TD3_policy('aSP-TD3')

Text cell <undefined>
#%% [markdown]
### Select the players:

Code cell <undefined>
#%% [code]
p1 = SP_TD3_policy
p2 = strong_basic_opponent

Text cell <undefined>
#%% [markdown]
### Observe some games:

Code cell <undefined>
#%% [code]
import time
time.sleep(5)
rollout(p1, p2, render=True)
Execution output
0KB
  text/plain
    (10.0, 0.0, 0.0)

Text cell <undefined>
#%% [markdown]
### Print the win-rate:

Code cell <undefined>
#%% [code]
num_games = 100
wins, defeats, draws = rollout(p1, p2, num_games)
print(
    f'# Games:   {num_games:8d}\n' +
    f'-------------------\n' +
    f'# Wins:    {int(wins):8d}\n' +
```

```
    f'# Defeats: {int(defeats):8d}\n' +
    f'# Ties:    {int(draws):8d}\n' +
    f'-------------------\n' +
    f'=> ({wins/num_games:.2f}/{defeats/num_games:.2f}/{draws/num_games
:.2f})'
)
Execution output
0KB
  Stream
    # Games:        100
    -------------------
    # Wins:          98
    # Defeats:        1
    # Ties:           1
    -------------------
    => (0.98/0.01/0.01)

Code cell <undefined>
#%% [code]
```

REPLAY BUFFER:

```
import numpy as np
import torch

from sum_tree import SumTree


class ReplayBuffer(object):
    def __init__(self, state_dim, action_dim, max_size=int(1e6)):
        self.max_size = max_size
        self.ptr = 0
        self.size = 0

        self.state = np.zeros((max_size, state_dim))
        self.action = np.zeros((max_size, action_dim))
        self.next_state = np.zeros((max_size, state_dim))
        self.reward = np.zeros((max_size, 1))
        self.not_done = np.zeros((max_size, 1))

        self.device = torch.device("cuda" if torch.cuda.is_available()
else "cpu")

    def add(self, state, action, next_state, reward, done):
        self.state[self.ptr] = state
        self.action[self.ptr] = action
        self.next_state[self.ptr] = next_state
```

```python
        self.reward[self.ptr] = reward
        self.not_done[self.ptr] = 1. - done

        self.ptr = (self.ptr + 1) % self.max_size
        self.size = min(self.size + 1, self.max_size)

    def sample(self, batch_size):
        ind = np.random.randint(0, self.size, size=batch_size)

        return (
            torch.FloatTensor(self.state[ind]).to(self.device),
            torch.FloatTensor(self.action[ind]).to(self.device),
            torch.FloatTensor(self.next_state[ind]).to(self.device),
            torch.FloatTensor(self.reward[ind]).to(self.device),
            torch.FloatTensor(self.not_done[ind]).to(self.device)
        )

    def observation_moments(self):
        obs_mean = np.mean(self.state, axis=0)
        obs_std = np.std(self.state, axis=0)
        return obs_mean, obs_std


class PrioritizedReplayBuffer(ReplayBuffer):
    def __init__(
            self,
            state_dim,
            action_dim,
            max_size=int(1e6),
            total_t=int(1e6),
            alpha=0.6,
            beta=0.4,
            beta_schedule="annealing",
            eps=0.01
    ):
        super().__init__(state_dim, action_dim, max_size)

        self.alpha = alpha
        self.beta = beta
        self.delta_beta = (1.0 - beta) / total_t if beta_schedule == "a
nnealing" else 0.0
        self.eps = eps

        self.tree = SumTree(max_size)

    def _get_priority(self, delta):
        # Proportional priority
        return (np.abs(delta) + self.eps) ** self.alpha
```

```
    def add(self, state, action, next_state, reward, done):
        self.tree.add(self.tree.max_p, self.ptr)
        super().add(state, action, next_state, reward, done)

    def sample(self, batch_size):
        data_indices = []
        indices = []
        priorities = []

        # Approximate cumulative density with segments of equal probabi
lity
        segment = self.tree.total_p() / batch_size

        # Annealing the amount of importance-
sampling over time by increasing beta
        self.beta = np.min([1., self.beta + self.delta_beta])

        for i in range(batch_size):
            # Sample exactly one transition from each segment
            lower_bound = segment * i
            upper_bound = segment * (i + 1)
            sample_p = np.random.uniform(lower_bound, upper_bound)

            (idx, p, data_idx) = self.tree.get(sample_p)
            data_indices.append(data_idx)
            indices.append(idx)
            priorities.append(p)

        # Correct sampling bias by using importance-sampling weights
        sampling_probabilities = np.asarray(priorities) / self.tree.tot
al_p()
        importance_weights = np.power(self.size * sampling_probabilitie
s, -self.beta)

        # Normalize weights for stability reasons
        importance_weights /= importance_weights.max()
        importance_weights = torch.FloatTensor(importance_weights).resh
ape(-1, 1).to(self.device)

        batch = (
            torch.FloatTensor(self.state[data_indices]).to(self.device)
,
            torch.FloatTensor(self.action[data_indices]).to(self.device
),
            torch.FloatTensor(self.next_state[data_indices]).to(self.de
vice),
            torch.FloatTensor(self.reward[data_indices]).to(self.device
),
```

```
            torch.FloatTensor(self.not_done[data_indices]).to(self.devi
ce)
        )

        return batch, indices, importance_weights

    def update(self, idx, delta):
        p = self._get_priority(delta)
        self.tree.update(idx, p)
```

TD3:

```
import copy
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

from utils import normalize

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim, max_action):
        super(Actor, self).__init__()

        self.l1 = nn.Linear(state_dim, hidden_dim)
        self.l2 = nn.Linear(hidden_dim, hidden_dim)
        self.l3 = nn.Linear(hidden_dim, action_dim)

        self.max_action = max_action

    def forward(self, state):
        a = F.relu(self.l1(state))
        a = F.relu(self.l2(a))
        return self.max_action * torch.tanh(self.l3(a))



class Critic(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim):
        super(Critic, self).__init__()

        # Q1 architecture
        self.l1 = nn.Linear(state_dim + action_dim, hidden_dim)
```

```python
        self.l2 = nn.Linear(hidden_dim, hidden_dim)
        self.l3 = nn.Linear(hidden_dim, 1)

        # Q2 architecture
        self.l4 = nn.Linear(state_dim + action_dim, hidden_dim)
        self.l5 = nn.Linear(hidden_dim, hidden_dim)
        self.l6 = nn.Linear(hidden_dim, 1)

    def forward(self, state, action):
        sa = torch.cat([state, action], 1)

        q1 = F.relu(self.l1(sa))
        q1 = F.relu(self.l2(q1))
        q1 = self.l3(q1)

        q2 = F.relu(self.l4(sa))
        q2 = F.relu(self.l5(q2))
        q2 = self.l6(q2)

        return q1, q2

    def Q1(self, state, action):
        sa = torch.cat([state, action], 1)

        q1 = F.relu(self.l1(sa))
        q1 = F.relu(self.l2(q1))
        q1 = self.l3(q1)

        return q1


class TD3(object):
    def __init__(
            self,
            state_dim,
            action_dim,
            hidden_dim,
            max_action,
            lr=3e-4,
            discount=0.99,
            tau=0.005,
            policy_noise=0.2,
            noise_clip=0.5,
            policy_freq=2,
            normalize_obs=False
    ):

        self.actor = Actor(state_dim, action_dim, hidden_dim, max_actio
n).to(device)
```

```
        print(self.actor)
        self.actor_target = copy.deepcopy(self.actor)
        self.actor_optimizer = torch.optim.Adam(self.actor.parameters()
, lr=lr)

        self.critic = Critic(state_dim, action_dim, hidden_dim).to(devi
ce)
        print(self.critic)
        self.critic_target = copy.deepcopy(self.critic)
        self.critic_optimizer = torch.optim.Adam(self.critic.parameters
(), lr=lr)

        self.max_action = max_action
        self.discount = discount
        self.tau = tau
        self.policy_noise = policy_noise
        self.noise_clip = noise_clip
        self.policy_freq = policy_freq

        # Load observation mean and std
        self.normalize_obs = normalize_obs
        self.obs_mean, self.obs_std = np.load('./results/observation_mo
ments.npy')
        self.obs_mean, self.obs_std = self.obs_mean.astype(np.float32),
 self.obs_std.astype(np.float32)

        self.total_it = 0

    def act(self, state):
        if self.normalize_obs:
            state = normalize(state, self.obs_mean, self.obs_std)

        state = torch.FloatTensor(state.reshape(1, -1)).to(device)
        return self.actor(state).cpu().data.numpy().flatten()

    def train(self, replay_buffer, prioritized_replay, batch_size=100):
        self.total_it += 1

        # Sample from replay buffer
        if prioritized_replay:
            batch, indices, importance_weights = replay_buffer.sample(b
atch_size)
            state, action, next_state, reward, not_done = batch
        else:
            state, action, next_state, reward, not_done = replay_buffer
.sample(batch_size)

        if self.normalize_obs:
```

```
            state = normalize(state, self.obs_mean, self.obs_std)
            next_state = normalize(next_state, self.obs_mean, self.obs_
std)

        with torch.no_grad():
            # Select action according to policy and add clipped noise
            noise = (
                    torch.randn_like(action) * self.policy_noise
            ).clamp(-self.noise_clip, self.noise_clip)

            next_action = (
                    self.actor_target(next_state) + noise
            ).clamp(-self.max_action, self.max_action)

            # Compute the target Q value
            target_q1, target_q2 = self.critic_target(next_state, next_
action)
            target_q = torch.min(target_q1, target_q2)
            target_q = reward + not_done * self.discount * target_q

        # Get current Q estimates
        current_q1, current_q2 = self.critic(state, action)

        if prioritized_replay:
            # Update priority
            delta = (torch.abs(current_q1 - target_q) + torch.abs(curre
nt_q2 - target_q)).cpu().data.numpy().flatten()
            for i in range(batch_size):
                idx = indices[i]
                replay_buffer.update(idx, delta[i])

            # Compute critic loss with importance weights
            critic_loss = F.mse_loss(current_q1, target_q, reduction='n
one') + \
                          F.mse_loss(current_q2, target_q, reduction='n
one')
            critic_loss = (importance_weights * critic_loss).mean()
        else:
            # Compute critic loss
            critic_loss = F.mse_loss(current_q1, target_q) + F.mse_loss
(current_q2, target_q)

        # Optimize the critic
        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        self.critic_optimizer.step()

        # Delayed policy updates
```

```
        if self.total_it % self.policy_freq == 0:

            # Compute actor loss
            actor_loss = -
self.critic.Q1(state, self.actor(state)).mean()

            # Optimize the actor
            self.actor_optimizer.zero_grad()
            actor_loss.backward()
            self.actor_optimizer.step()

            # Update the frozen target models
            for param, target_param in zip(self.critic.parameters(), se
lf.critic_target.parameters()):
                target_param.data.copy_(self.tau * param.data + (1 - se
lf.tau) * target_param.data)

            for param, target_param in zip(self.actor.parameters(), sel
f.actor_target.parameters()):
                target_param.data.copy_(self.tau * param.data + (1 - se
lf.tau) * target_param.data)

    def save(self, filename):
        torch.save(self.critic.state_dict(), filename + "_critic")
        torch.save(self.critic_optimizer.state_dict(), filename + "_cri
tic_optimizer")

        torch.save(self.actor.state_dict(), filename + "_actor")
        torch.save(self.actor_optimizer.state_dict(), filename + "_acto
r_optimizer")

    def load(self, filename):
        self.critic.load_state_dict(torch.load(filename + "_critic", ma
p_location=device))
        self.critic_optimizer.load_state_dict(torch.load(filename + "_c
ritic_optimizer", map_location=device))
        self.critic_target = copy.deepcopy(self.critic)

        self.actor.load_state_dict(torch.load(filename + "_actor", map_
location=device))
        self.actor_optimizer.load_state_dict(torch.load(filename + "_ac
tor_optimizer", map_location=device))
        self.actor_target = copy.deepcopy(self.actor)
```

ENVIRONMENT:

```
import math
import numpy as np
```

```python
import Box2D
# noinspection PyUnresolvedReferences
from Box2D.b2 import (edgeShape, circleShape, fixtureDef, polygonShape,
 revoluteJointDef, contactListener)

import gym
from gym import spaces
from gym.utils import seeding, EzPickle
import colorama
from colorama import Fore, Style

# import pyglet
# from pyglet import gl

FPS = 50
SCALE = 60.0  # affects how fast-
paced the game is, forces should be adjusted as well (Don't touch)

VIEWPORT_W = 600
VIEWPORT_H = 480
W = VIEWPORT_W / SCALE
H = VIEWPORT_H / SCALE
CENTER_X = W / 2
CENTER_Y = H / 2
ZONE = W / 20
MAX_ANGLE = math.pi / 3  # Maximimal angle of racket
MAX_TIME_KEEP_PUCK = 15
GOAL_SIZE = 75

RACKETPOLY = [(-10, 20), (+5, 20), (+5, -20), (-10, -20), (-18, -
10), (-21, 0), (-18, 10)]
RACKETFACTOR = 1.2

FORCEMULTIPLIER = 6000
SHOOTFORCEMULTIPLIER = 60
TORQUEMULTIPLIER = 400
MAX_PUCK_SPEED = 25


def dist_positions(p1, p2):
  return np.sqrt(np.sum(np.asarray(p1 - p2) ** 2, axis=-1))


class ContactDetector(contactListener):
  def __init__(self, env, verbose=False):
    contactListener.__init__(self)
    self.env = env
    self.verbose = verbose
```

```
  def BeginContact(self, contact):
    if self.env.goal_player_2 == contact.fixtureA.body or self.env.goal
_player_2 == contact.fixtureB.body:
      if self.env.puck == contact.fixtureA.body or self.env.puck == con
tact.fixtureB.body:
        if self.verbose:
          print(f'{Fore.GREEN}Player 1{Style.RESET_ALL}', end="")
        self.env.done = True
        self.env.winner = 1
    if self.env.goal_player_1 == contact.fixtureA.body or self.env.goal
_player_1 == contact.fixtureB.body:
      if self.env.puck == contact.fixtureA.body or self.env.puck == con
tact.fixtureB.body:
        if self.verbose:
          print(f'{Fore.RED}Player 2{Style.RESET_ALL}', end="")
        self.env.done = True
        self.env.winner = -1
    if (contact.fixtureA.body == self.env.player1 or contact.fixtureB.b
ody == self.env.player1) \
        and (contact.fixtureA.body == self.env.puck or contact.fixtureB
.body == self.env.puck):
      if self.env.keep_mode and self.env.puck.linearVelocity[0] < 0.1:
        if self.env.player1_has_puck == 0:
          self.env.player1_has_puck = MAX_TIME_KEEP_PUCK

    if (contact.fixtureA.body == self.env.player2 or contact.fixtureB.b
ody == self.env.player2) \
        and (contact.fixtureA.body == self.env.puck or contact.fixtureB
.body == self.env.puck):
      if self.env.keep_mode and self.env.puck.linearVelocity[0] > -0.1:
        if self.env.player2_has_puck == 0:
          self.env.player2_has_puck = MAX_TIME_KEEP_PUCK

  def EndContact(self, contact):
    pass


class HockeyEnv(gym.Env, EzPickle):
  metadata = {
    'render.modes': ['human', 'rgb_array'],
    'video.frames_per_second': FPS
  }

  continuous = False
  NORMAL = 0
  TRAIN_SHOOTING = 1
  TRAIN_DEFENSE = 2

  def __init__(self, keep_mode=True, mode=NORMAL, verbose=False):
```

```python
""" mode: is the game mode: NORMAL, TRAIN_SHOOTING, TRAIN_DEFENSE,
    keep_mode: whether the puck gets catched by
    it can be changed later using the reset function
    """
EzPickle.__init__(self)
self.seed()
self.viewer = None
self.mode = mode
self.keep_mode = keep_mode
self.player1_has_puck = 0
self.player2_has_puck = 0

self.world = Box2D.b2World([0, 0])
self.player1 = None
self.player2 = None
self.puck = None
self.goal_player_1 = None
self.goal_player_2 = None
self.world_objects = []
self.drawlist = []
self.done = False
self.winner = 0
self.one_starts = True  # player one starts the game (alternating)

self.timeStep = 1.0 / FPS
self.time = 0
self.max_timesteps = None  # see reset

self.closest_to_goal_dist = 1000

# 0  x pos player one
# 1  y pos player one
# 2  angle player one
# 3  x vel player one
# 4  y vel player one
# 5  angular vel player one
# 6  x player two
# 7  y player two
# 8  angle player two
# 9 y vel player two
# 10 y vel player two
# 11 angular vel player two
# 12 x pos puck
# 13 y pos puck
# 14 x vel puck
# 15 y vel puck
# Keep Puck Mode
# 16 time left player has puck
```

```
    # 17 time left other player has puck
    self.observation_space = spaces.Box(-
np.inf, np.inf, shape=(18,), dtype=np.float32)

    # linear force in (x,y)-direction and torque
    self.num_actions = 3 if not self.keep_mode else 4
    self.action_space = spaces.Box(-
1, +1, (self.num_actions * 2,), dtype=np.float32)

    # see discrete_to_continous_action()
    self.discrete_action_space = spaces.Discrete(7)

    self.verbose = verbose

    self.reset(self.one_starts)

  def seed(self, seed=None):
    self.np_random, seed = seeding.np_random(seed)
    self._seed = seed
    return [seed]

  def _destroy(self):
    if self.player1 is None: return
    self.world.contactListener = None
    self.world.DestroyBody(self.player1)
    self.player1 = None
    self.world.DestroyBody(self.player2)
    self.player2 = None
    self.world.DestroyBody(self.puck)
    self.puck = None
    self.world.DestroyBody(self.goal_player_1)
    self.goal_player_1 = None
    self.world.DestroyBody(self.goal_player_2)
    self.goal_player_2 = None
    for obj in self.world_objects:
      self.world.DestroyBody(obj)
    self.world_objects = []
    self.drawlist = []

  def r_uniform(self, mini, maxi):
    return self.np_random.uniform(mini, maxi, 1)[0]

  def _create_player(self, position, color, is_player_two):
    player = self.world.CreateDynamicBody(
      position=position,
      angle=0.0,
      fixtures=fixtureDef(
```

```
        shape=polygonShape(vertices=[(-
x / SCALE * RACKETFACTOR if is_player_two else x / SCALE * RACKETFACTOR
,
                                              y / SCALE * RACKETFACTOR)
                                             for x, y in RACKETPOLY]),
        density=200.0 / RACKETFACTOR,
        friction=1.0,
        categoryBits=0x0010,
        maskBits=0x011,  # collide only with ground
        restitution=0.0)  # 0.99 bouncy
    )
    player.color1 = color
    player.color2 = color
    # player.linearDamping = 0.1
    player.anguarDamping = 1.0

    return player

  def _create_puck(self, position, color):
    puck = self.world.CreateDynamicBody(
      position=position,
      angle=0.0,
      fixtures=fixtureDef(
        shape=circleShape(radius=13 / SCALE, pos=(0, 0)),
        density=7.0,
        friction=0.1,
        categoryBits=0x001,
        maskBits=0x0010,
        restitution=0.95)  # 0.99 bouncy
    )
    puck.color1 = color
    puck.color2 = color
    puck.linearDamping = 0.05

    return puck

  def _create_world(self):
    def _create_wall(position, poly):
      wall = self.world.CreateStaticBody(
        position=position,
        angle=0.0,
        fixtures=fixtureDef(
          shape=polygonShape(vertices=[(x / SCALE, y / SCALE) for x, y
in poly]),
          density=0,
          friction=0.1,
          categoryBits=0x011,
          maskBits=0x0011)
```

```
      )
      wall.color1 = (0, 0, 0)
      wall.color2 = (0, 0, 0)

      return wall

    def _create_decoration():
      objs = []
      objs.append(self.world.CreateStaticBody(
        position=(W / 2, H / 2),
        angle=0.0,
        fixtures=fixtureDef(
          shape=circleShape(radius=100 / SCALE, pos=(0, 0)),
          categoryBits=0x0,
          maskBits=0x0)
      ))
      objs[-1].color1 = (0.8, 0.8, 0.8)
      objs[-1].color2 = (0.8, 0.8, 0.8)

      # left goal
      objs.append(self.world.CreateStaticBody(
        position=(W / 2 - 250 / SCALE, H / 2),
        angle=0.0,
        fixtures=fixtureDef(
          shape=circleShape(radius=GOAL_SIZE / SCALE, pos=(0, 0)),
          categoryBits=0x0,
          maskBits=0x0)
      ))
      orange = (239. / 255, 203. / 255, 138. / 255)
      objs[-1].color1 = orange
      objs[-1].color2 = orange

      poly = [(0, 100), (100, 100), (100, -100), (0, -100)]
      objs.append(self.world.CreateStaticBody(
        position=(W / 2 - 240 / SCALE, H / 2),
        angle=0.0,
        fixtures=fixtureDef(
          shape=polygonShape(vertices=[(x / SCALE, y / SCALE) for x, y
in poly]),
          categoryBits=0x0,
          maskBits=0x0)
      ))
      objs[-1].color1 = (1, 1, 1)
      objs[-1].color2 = (1, 1, 1)

      # right goal
      objs.append(self.world.CreateStaticBody(
        position=(W / 2 + 250 / SCALE, H / 2),
```

```
        angle=0.0,
        fixtures=fixtureDef(
          shape=circleShape(radius=GOAL_SIZE / SCALE, pos=(0, 0)),
          categoryBits=0x0,
          maskBits=0x0)
      ))
      objs[-1].color1 = orange
      objs[-1].color2 = orange

      poly = [(100, 100), (0, 100), (0, -100), (100, -100)]
      objs.append(self.world.CreateStaticBody(
        position=(W / 2 + 140 / SCALE, H / 2),
        angle=0.0,
        fixtures=fixtureDef(
          shape=polygonShape(vertices=[(x / SCALE, y / SCALE) for x, y
in poly]),
          categoryBits=0x0,
          maskBits=0x0)
      ))
      objs[-1].color1 = (1, 1, 1)
      objs[-1].color2 = (1, 1, 1)

      return objs

    self.world_objects = []

    self.world_objects.extend(_create_decoration())

    poly = [(-250, 10), (-250, -10), (250, -10), (250, 10)]
    self.world_objects.append(_create_wall((W / 2, H - .5), poly))
    self.world_objects.append(_create_wall((W / 2, .5), poly))

    poly = [(-
10, (H - 1) / 2 * SCALE - GOAL_SIZE), (10, (H - 1) / 2 * SCALE - GOAL_S
IZE - 7), (10, -5), (-10, -5)]
    self.world_objects.append(_create_wall((W / 2 - 245 / SCALE, H - .5
), [(x, -y) for x, y in poly]))
    self.world_objects.append(_create_wall((W / 2 - 245 / SCALE, .5), p
oly))

    self.world_objects.append(
      _create_wall((W / 2 + 245 / SCALE, H - .5), [(-x, -
y) for x, y in poly]))
    self.world_objects.append(_create_wall((W / 2 + 245 / SCALE, 0.5),
[(-x, y) for x, y in poly]))

    self.drawlist.extend(self.world_objects)
```

```
  def _create_goal(self, position, poly):
    goal = self.world.CreateStaticBody(
      position=position,
      angle=0.0,
      fixtures=[
        fixtureDef(
          shape=polygonShape(vertices=[(x / SCALE, y / SCALE) for x, y
in poly]),
          density=0,
          friction=0.1,
          categoryBits=0x0010,
          maskBits=0x001,
          isSensor=True),
        fixtureDef(
          shape=polygonShape(vertices=[(x / SCALE, y / SCALE) for x, y
in poly]),
          density=0,
          friction=0.1,
          categoryBits=0x010,
          maskBits=0x0010)]
    )
    goal.color1 = (.5, .5, .5)
    goal.color2 = (.5, .5, .5)

    return goal

  def reset(self, one_starting=None, mode=None):
    self._destroy()
    self.world.contactListener_keepref = ContactDetector(self, verbose=
self.verbose)
    self.world.contactListener = self.world.contactListener_keepref
    self.done = False
    self.winner = 0
    self.prev_shaping = None
    self.time = 0
    if mode is not None and mode in [self.NORMAL, self.TRAIN_SHOOTING,
self.TRAIN_DEFENSE]:
      self.mode = mode

    if self.mode == self.NORMAL:
      self.max_timesteps = 250
      if one_starting is not None:
        self.one_starts = one_starting
      else:
        self.one_starts = not self.one_starts
    else:
      self.max_timesteps = 80
    self.closest_to_goal_dist = 1000
```

```
    W = VIEWPORT_W / SCALE
    H = VIEWPORT_H / SCALE

    # Create world
    self._create_world()

    poly = [(-10, GOAL_SIZE), (10, GOAL_SIZE), (10, -GOAL_SIZE), (-
10, -GOAL_SIZE)]
    self.goal_player_1 = self._create_goal((W / 2 - 245 / SCALE - 10 /
SCALE, H / 2), poly)
    self.goal_player_2 = self._create_goal((W / 2 + 245 / SCALE + 10 /
SCALE, H / 2), poly)

    # Create players
    red = (235. / 255., 98. / 255., 53. / 255.)
    self.player1 = self._create_player(
      (W / 5, H / 2),
      red,
      False
    )
    blue = (93. / 255, 158. / 255., 199. / 255.)
    if self.mode != self.NORMAL:
      self.player2 = self._create_player(
        (4 * W / 5 + self.r_uniform(-
W / 3, W / 6), H / 2 + self.r_uniform(-H / 4, H / 4)),
        blue,
        True
      )
    else:
      self.player2 = self._create_player(
        (4 * W / 5, H / 2),
        blue,
        True
      )
    if self.mode == self.NORMAL or self.mode == self.TRAIN_SHOOTING:
      if self.one_starts or self.mode == self.TRAIN_SHOOTING:
        self.puck = self._create_puck((W / 2 - self.r_uniform(H / 8, H
/ 4),
                                       H / 2 + self.r_uniform(-
H / 8, H / 8)), (0, 0, 0))
      else:
        self.puck = self._create_puck((W / 2 + self.r_uniform(H / 8, H
/ 4),
                                       H / 2 + self.r_uniform(-
H / 8, H / 8)), (0, 0, 0))
    elif self.mode == self.TRAIN_DEFENSE:
      self.puck = self._create_puck((W / 2 + self.r_uniform(0, W / 3),
```

```
                                      H / 2 + 0.8 * self.r_uniform(-
H / 2, H / 2)), (0, 0, 0))
      direction = (self.puck.position - (
        0, H / 2 + .6 * self.r_uniform(-
GOAL_SIZE / SCALE, GOAL_SIZE / SCALE)))
      direction = direction / direction.length
      force = -
direction * SHOOTFORCEMULTIPLIER * self.puck.mass / self.timeStep
      self.puck.ApplyForceToCenter(force, True)
    # Todo get the scaling right

    self.drawlist.extend([self.player1, self.player2, self.puck])

    obs = self._get_obs()

    return obs

  def _check_boundaries(self, force, player, is_player_one):
    if (is_player_one and player.position[0] < W / 2 - 210 / SCALE and
force[0] < 0) \
        or (not is_player_one and player.position[0] > W / 2 + 210 / SC
ALE and force[0] > 0) \
        or (is_player_one and player.position[0] > W / 2 and force[0] >
 0) \
        or (not is_player_one and player.position[0] < W / 2 and force[
0] < 0): # Do not leave playing area to the left/right
        vel = player.linearVelocity
        player.linearVelocity[0] = 0
        force[0] = -vel[0]
    if (is_player_one and player.position[1] > H - 1.2 and force[1] > 0
) \
        or (not is_player_one and player.position[1] > H - 1.2 and forc
e[1] > 0) \
        or (is_player_one and player.position[1] < 1.2 and force[1] < 0
) \
        or (not is_player_one and player.position[1] < 1.2 and force[1]
 < 0): # Do not leave playing area to the top/bottom
        vel = player.linearVelocity
        player.linearVelocity[1] = 0
        force[1] = -vel[1]

    return force

  def _apply_translation_action_with_max_speed(self, player, action, ma
x_speed, is_player_one):
    velocity = np.asarray(player.linearVelocity)
    speed = np.sqrt(np.sum((velocity) ** 2))
    if is_player_one:
```

```
        force = action * FORCEMULTIPLIER
      else:
        force = -action * FORCEMULTIPLIER

      if (is_player_one and player.position[0] > CENTER_X - ZONE) \
          or (not is_player_one and player.position[0] < CENTER_X + ZONE)
:  # bounce at the center line
        force[0] = 0
        if is_player_one:
          if player.linearVelocity[0] > 0:
            force[0] = -
2 * player.linearVelocity[0] * player.mass / self.timeStep
          force[0] += -
1 * (player.position[0] - CENTER_X) * player.linearVelocity[
            0] * player.mass / self.timeStep
        else:
          if player.linearVelocity[0] < 0:
            force[0] = -
2 * player.linearVelocity[0] * player.mass / self.timeStep
          force[0] += 1 * (player.position[0] - CENTER_X) * player.linear
Velocity[0] * player.mass / self.timeStep

        player.linearDamping = 20.0

        player.ApplyForceToCenter(self._check_boundaries(force, player, i
s_player_one).tolist(), True)
        return

      if speed < max_speed:
        player.linearDamping = 5.0
        player.ApplyForceToCenter(self._check_boundaries(force.tolist(),
player, is_player_one), True)
      else:
        player.linearDamping = 20.0
        deltaVelocity = self.timeStep * force / player.mass
        if np.sqrt(np.sum((velocity + deltaVelocity) ** 2)) < speed:
          player.ApplyForceToCenter(self._check_boundaries(force.tolist()
, player, is_player_one), True)
        else:
          pass

    def _apply_rotation_action_with_max_speed(self, player, action):
      angle = np.asarray(player.angle)
      torque = action * TORQUEMULTIPLIER
      if (abs(angle) > MAX_ANGLE):  # limit rotation
        torque = 0
        if player.angle * player.angularVelocity > 0:
```

```
        torque = -
0.1 * player.angularVelocity * player.mass / self.timeStep
      torque += -0.1 * (player.angle) * player.mass / self.timeStep
      player.angularDamping = 10.0
    else:
      player.angularDamping = 2.0
    player.ApplyTorque(torque, True)

  def _get_obs(self):
    obs = np.hstack([
                      self.player1.position - [CENTER_X, CENTER_Y],
                      [self.player1.angle],
                      self.player1.linearVelocity,
                      [self.player1.angularVelocity],
                      self.player2.position - [CENTER_X, CENTER_Y],
                      [self.player2.angle],
                      self.player2.linearVelocity,
                      [self.player2.angularVelocity],
                      self.puck.position - [CENTER_X, CENTER_Y],
                      self.puck.linearVelocity]
                    + ([] if not self.keep_mode else [self.player1_has_
puck, self.player2_has_puck]))
    return obs

  def obs_agent_two(self):
    """ returns the observations for agent two (symmetric mirrored vers
ion of agent one)
        """
    obs = np.hstack([
                      -(self.player2.position - [CENTER_X, CENTER_Y]),
                      [self.player2.angle],  # the angle is already rot
ationally symmetric
                      -self.player2.linearVelocity,
                      [self.player2.angularVelocity],
                      -(self.player1.position - [CENTER_X, CENTER_Y]),
                      [self.player1.angle],  # the angle is already rot
ationally symmetric
                      -self.player1.linearVelocity,
                      [self.player1.angularVelocity],
                      -(self.puck.position - [CENTER_X, CENTER_Y]),
                      -self.puck.linearVelocity
                    ] + ([] if not self.keep_mode else [self.player2_ha
s_puck, self.player1_has_puck]))

    return obs

  def _compute_reward(self):
    r = 0
```

```
    if self.done:
      if self.winner == 0:  # tie
        r += 0
      elif self.winner == 1:  # you won
        r += 10
      else:  # opponent won
        r -= 10
    return r

  def _get_info(self):
    # different proxy rewards:
    # Proxy reward/penalty for not being close to puck in the own half
when puck is flying towards goal (not to opponent)
    reward_closeness_to_puck = 0
    if self.puck.position[0] < CENTER_X and self.puck.linearVelocity[0]
 <= 0:
      dist_to_puck = dist_positions(self.player1.position, self.puck.po
sition)
      max_dist = 250. / SCALE
      max_reward = -30.  # max (negative) reward through this proxy
      factor = max_reward / (max_dist * self.max_timesteps / 2)
      reward_closeness_to_puck += dist_to_puck * factor  # Proxy reward
 for being close to puck in the own half
    # Proxy reward: touch puck
    reward_touch_puck = 0.
    if self.player1_has_puck == MAX_TIME_KEEP_PUCK:
      reward_touch_puck = 1.

    # puck is flying in the right direction
    max_reward = 1.
    factor = max_reward / (self.max_timesteps * MAX_PUCK_SPEED)
    reward_puck_direction = self.puck.linearVelocity[0] * factor  # Puc
k flies right is good and left not

    return {"winner": self.winner,
            "reward_closeness_to_puck": reward_closeness_to_puck,
            "reward_touch_puck": reward_touch_puck,
            "reward_puck_direction": reward_puck_direction,
            }

  def set_state(self, state):
    """ function to revert the state of the environment to a previous s
tate (observation)"""
    self.player1.position = (state[[0, 1]] + [CENTER_X, CENTER_Y]).toli
st()
    self.player1.angle = math.atan2(state[2], state[3])
    self.player1.linearVelocity = [state[4], state[5]]
```

```
    self.player1.angularVelocity = state[6]
    self.player2.position = (state[[7, 8]] + [CENTER_X, CENTER_Y]).toli
st()
    self.player2.angle = math.atan2(state[9], state[10])
    self.player2.linearVelocity = [state[11], state[12]]
    self.player2.angularVelocity = state[13]
    self.puck.position = (state[[14, 15]] + [CENTER_X, CENTER_Y]).tolis
t()
    self.puck.linearVelocity = [state[16], state[17]]

  def _limit_puck_speed(self):
    puck_speed = np.sqrt(self.puck.linearVelocity[0] ** 2 + self.puck.l
inearVelocity[1] ** 2)
    if puck_speed > MAX_PUCK_SPEED:
      self.puck.linearDamping = 10.0
    else:
      self.puck.linearDamping = 0.05
    self.puck.angularSpeed = 0

  def _keep_puck(self, player):
    self.puck.position = player.position
    self.puck.linearVelocity = player.linearVelocity

  def _shoot(self, player):
    # self.puck.position = player.position
    self.puck.ApplyForceToCenter(
      Box2D.b2Vec2(math.cos(player.angle),
                   math.sin(player.angle)) *
      self.puck.mass / self.timeStep * SHOOTFORCEMULTIPLIER, True)

  def discrete_to_continous_action(self, discrete_action):
    ''' converts discrete actions into continuous ones (for each player
)
        The actions allow only one operation each timestep, e.g. X or Y
 or angle change.
        This is surely limiting. Other discrete actions are possible
        Action 0: do nothing
        Action 1: -1 in x
        Action 2: 1 in x
        Action 3: -1 in y
        Action 4: 1 in y
        Action 5: -1 in angle
        Action 6: 1 in angle
        Action 7: shoot (if keep_mode is on)
        '''
    action_cont = [(discrete_action == 1) * -
1 + (discrete_action == 2) * 1,  # player x
```

```
                   (discrete_action == 3) * -
1 + (discrete_action == 4) * 1,  # player y
                   (discrete_action == 5) * -
1 + (discrete_action == 6) * 1]  # player angle
    if self.keep_mode:
      action_cont.append(discrete_action == 7)

    return action_cont

  def step(self, action):
    action = np.clip(action, -1, +1).astype(np.float32)

    self._apply_translation_action_with_max_speed(self.player1, action[
:2], 10, True)
    self._apply_rotation_action_with_max_speed(self.player1, action[2])
    player2_idx = 3 if not self.keep_mode else 4
    self._apply_translation_action_with_max_speed(self.player2, action[
player2_idx:player2_idx + 2], 10, False)
    self._apply_rotation_action_with_max_speed(self.player2, action[pla
yer2_idx + 2])

    self._limit_puck_speed()
    if self.keep_mode:
      if self.player1_has_puck > 1:
        self._keep_puck(self.player1)
        self.player1_has_puck -= 1
        if self.player1_has_puck == 1 or action[3] > 0.5:  # shooting
          self._shoot(self.player1)
          self.player1_has_puck = 0
      if self.player2_has_puck > 1:
        self._keep_puck(self.player2)
        self.player2_has_puck -= 1
        if self.player2_has_puck == 1 or action[player2_idx + 3] > 0.5:
  # shooting
          self._shoot(self.player2)
          self.player2_has_puck = 0

    self.world.Step(self.timeStep, 6 * 30, 2 * 30)

    obs = self._get_obs()
    if self.time >= self.max_timesteps:
      self.done = True

    reward = self._compute_reward()
    info = self._get_info()

    self.closest_to_goal_dist = min(self.closest_to_goal_dist,
```

```
                                    dist_positions(self.puck.position,
(W, H / 2)))
    self.time += 1
    return obs, reward + info["reward_closeness_to_puck"], self.done, i
nfo

  def render(self, mode='human'):
    from gym.envs.classic_control import rendering
    if self.viewer is None:
      self.viewer = rendering.Viewer(VIEWPORT_W, VIEWPORT_H)
      self.viewer.set_bounds(0, VIEWPORT_W / SCALE, 0, VIEWPORT_H / SCA
LE)
      # self.score_label = pyglet.text.Label('0000', font_size=50,
      #                                      x=VIEWPORT_W/2, y=VIEWPORT
_H/2, anchor_x='center', anchor_y='center',
      #                                      color=(0, 0, 0, 255))

    # arr = None
    # win = self.viewer.window
    # win.clear()
    # gl.glViewport(0, 0, VIEWPORT_W, VIEWPORT_H)
    for obj in self.drawlist:
      for f in obj.fixtures:
        trans = f.body.transform
        if type(f.shape) is circleShape:
          t = rendering.Transform(translation=trans * f.shape.pos)
          self.viewer.draw_circle(f.shape.radius, 20, color=obj.color1)
.add_attr(t)
          self.viewer.draw_circle(f.shape.radius, 20, color=obj.color2,
 filled=False, linewidth=2).add_attr(t)
        else:
          path = [trans * v for v in f.shape.vertices]
          self.viewer.draw_polygon(path, color=obj.color1)
          path.append(path[0])
          self.viewer.draw_polyline(path, color=obj.color2, linewidth=2
)

    # self.score_label.draw()

    return self.viewer.render(return_rgb_array=mode == 'rgb_array')

  def close(self):
    if self.viewer is not None:
      self.viewer.close()
      self.viewer = None


class BasicOpponent():
  def __init__(self, weak=True, keep_mode=True):
```

```
    self.weak = weak
    self.keep_mode = keep_mode
    self.phase = np.random.uniform(0, np.pi)

  def act(self, obs, verbose=False):
    alpha = obs[2]
    p1 = np.asarray([obs[0], obs[1], alpha])
    v1 = np.asarray(obs[3:6])
    puck = np.asarray(obs[12:14])
    puckv = np.asarray(obs[14:16])
    # print(p1,v1,puck,puckv)
    target_pos = p1[0:2]
    target_angle = p1[2]
    self.phase += np.random.uniform(0, 0.2)

    time_to_break = 0.1
    if self.weak:
      kp = 0.5
    else:
      kp = 10
    kd = 0.5

    # if ball flies towards our goal or very slowly away: try to catch
it
    if puckv[0] < 30.0 / SCALE:
      dist = np.sqrt(np.sum((p1[0:2] - puck) ** 2))
      # Am I behind the ball?
      if p1[0] < puck[0] and abs(p1[1] - puck[1]) < 30.0 / SCALE:
        # Go and kick
        target_pos = [puck[0] + 0.2, puck[1] + puckv[1] * dist * 0.1]
      else:
        # get behind the ball first
        target_pos = [-210 / SCALE, puck[1]]
    else:  # go in front of the goal
      target_pos = [-210 / SCALE, 0]
    target_angle = MAX_ANGLE * np.sin(self.phase)
    shoot = 0.0
    if self.keep_mode and obs[16] > 0 and obs[16] < 7:
      shoot = 1.0

    target = np.asarray([target_pos[0], target_pos[1], target_angle])
    # use PD control to get to target
    error = target - p1
    need_break = abs((error / (v1 + 0.01))) < [time_to_break, time_to_b
reak, time_to_break * 10]
    if verbose:
      print(error, abs(error / (v1 + 0.01)), need_break)
```

```
    action = np.clip(error * [kp, kp / 5, kp / 2] - v1 * need_break * [
kd, kd, kd], -1, 1)
    if self.keep_mode:
      return np.hstack([action, [shoot]])
    else:
      return action


class HumanOpponent():
  def __init__(self, env, player=1):
    self.env = env
    self.player = player
    self.a = 0

    if env.viewer is None:
      env.render()

    self.env.viewer.window.on_key_press = self.key_press
    self.env.viewer.window.on_key_release = self.key_release

    self.key_action_mapping = {
      65361: 1 if self.player == 1 else 2,  # Left arrow key
      65362: 4 if self.player == 1 else 3,  # Up arrow key
      65363: 2 if self.player == 1 else 1,  # Right arrow key
      65364: 3 if self.player == 1 else 4,  # Down arrow key
      119: 5,  # w
      115: 6,  # s
      32: 7,  # space
    }

    print('Human Controls:')
    print(' left:\t\t\tleft arrow key left')
    print(' right:\t\t\tarrow key right')
    print(' up:\t\t\tarrow key up')
    print(' down:\t\t\tarrow key down')
    print(' tilt clockwise:\tw')
    print(' tilt anti-clockwise:\ts')
    print(' shoot :\tspace')

  def key_press(self, key, mod):
    if key in self.key_action_mapping:
      self.a = self.key_action_mapping[key]

  def key_release(self, key, mod):
    if key in self.key_action_mapping:
      a = self.key_action_mapping[key]
      if self.a == a:
        self.a = 0
```

```python
  def act(self, obs):
    # print(self.a)
    return self.env.discrete_to_continous_action(self.a)


class HockeyEnv_BasicOpponent(HockeyEnv):

  def __init__(self, mode=HockeyEnv.NORMAL, weak_opponent=False):
    super().__init__(mode=mode, keep_mode=True)
    self.opponent = BasicOpponent(weak=weak_opponent)
    # linear force in (x,y)-direction, torque, and shooting
    self.action_space = spaces.Box(-1, +1, (4,), dtype=np.float32)

  def step(self, action):
    ob2 = self.obs_agent_two()
    a2 = self.opponent.act(ob2)
    action2 = np.hstack([action, a2])
    return super().step(action2)


from gym.envs.registration import register

try:
  register(
    id='Hockey-v0',
    entry_point='laserhockey.hockey_env:HockeyEnv',
    kwargs={'mode': 0}
  )
  register(
    id='Hockey-One-v0',
    entry_point='laserhockey.hockey_env:HockeyEnv_BasicOpponent',
    kwargs={'mode': 0, 'weak_opponent': False}
  )
except Exception as e:
  print(e)
```