# HW2 - Q Actor Critics - DQN, DDPG, SAC

This assignment builds to a simple Soft Actor Critic (2018) by progressing from predecessor algorithms:
Deep Q Networks (2013) and Deep Deterministic Policy Gradients (2015). They all build on tabular Q learning (~1989). Note, many variations of these algorithms exist. Please use the math contained in this notebook for the coding sections.

# 0. Warm Up Questions [30 pts total; 2 pt each]

ALL ANSWERS IN OTHER SUBMITTED DOCUMENT

Answer each question concisely. One sentence, one formula, one line of code, etc. Use of $\LaTeX$ formatting for math is encouraged.

1. How does the Q function $Q(s_t, a_t)$ relate to sum of discounted rewards $\sum_{t=0}^{T} \gamma^t r_t$?

Type answer here ..

2. How does the Q function $Q(s_t, a_t)$ relate to $Q(s_{t+1}, a_{t+1})$?
3. When Q is accurate are these definitions equivalent?
4. Whats the loss for a neural approximation to the Q network?
5. In the discrete case, how do you select actions given an accurate Q network?
6. In DQN, for an environment with 5 continuous states and 3 discrete action choices: Whats the input and output size of the Q network?
7. In DDPG, for an environment with 5 continuous states and 3 continuous action: Whats the input and output size of the Q network?
8. In DQN, what is a target network and why do we need it?

9. In DQN, what is a replay buffer and why do we need it?
10. Explain this inequality $\mathbb{E}[\max(C_1, C_2)] \geq \max(\mathbb{E}[C_1], \mathbb{E}[C_2])$, assuming $C_1$ and $C_2$ are random variables representing the probability of getting heads when flipping two fair coins. (This is the basis for double Q learning in Double DQN and SAC.)
11. Do off policy algorithms use a replay buffer?
12. What does 'with torch.no_grad():' do and why should you use it when calling target networks but not regular networks?
13. Why do you need a policy network in DDPG but not DQN, and what is the DDPG policy loss.
14. Compare and contrast hard and soft target network updates.
15. In InvertedPendulum-v5 what are the physical meanings of states and actions and are they discrete or continuous?

# Boiler Plate

### (read through atleast once)

## Imports and Set up

Installs gymnasium, imports deep learning libs, sets torch device. **You shouldnt need to change this code.**

This notebook should work with CPU or GPU. To change: **click Runtime (top left of notebook) -> Change runtime type -> select a CPU/GPU -> Save**. I'd recommend debugging on the CPU (to save available GPU time) and doing full runs on GPU (to increase training speed). Regardless, these environments should solve within minutes even on the CPU.

In [48]:
```
!pip install gymnasium[mujoco]
!apt install -y libgl1-mesa-glx libosmesa6 libglfw3 patchelf
import gymnasium as gym

import torch
from torch import nn, zeros
from torch.optim import Adam
from torch.utils.tensorboard import SummaryWriter
```

```python
from collections import deque
import random
import copy

# device = "cuda" if torch.cuda.is_available() else "cpu"
device = "cpu"
print(f"Using device: {device}")

# random seeds for reproducability
torch.manual_seed(0)
torch.cuda.manual_seed_all(0)
random.seed(0)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

```
54149.64s - pydevd: Sending message related to process being replaced timed-out after 5 seconds
```

```
Looking in indexes: https://pypi.org/simple, https://pypi.ngc.nvidia.com
Requirement already satisfied: gymnasium[mujoco] in /home/jblevins32/anaconda3/lib/python3.11/site-packages
(1.0.0)
Requirement already satisfied: numpy>=1.21.0 in /home/jblevins32/anaconda3/lib/python3.11/site-packages (fr
om gymnasium[mujoco]) (1.26.4)
Requirement already satisfied: cloudpickle>=1.2.0 in /home/jblevins32/anaconda3/lib/python3.11/site-package
s (from gymnasium[mujoco]) (3.0.0)
Requirement already satisfied: typing-extensions>=4.3.0 in /home/jblevins32/anaconda3/lib/python3.11/site-p
ackages (from gymnasium[mujoco]) (4.12.2)
Requirement already satisfied: farama-notifications>=0.0.1 in /home/jblevins32/anaconda3/lib/python3.11/sit
e-packages (from gymnasium[mujoco]) (0.0.4)
Requirement already satisfied: mujoco>=2.1.5 in /home/jblevins32/anaconda3/lib/python3.11/site-packages (fr
om gymnasium[mujoco]) (3.2.7)
Requirement already satisfied: imageio>=2.14.1 in /home/jblevins32/anaconda3/lib/python3.11/site-packages
(from gymnasium[mujoco]) (2.37.0)
Requirement already satisfied: pillow>=8.3.2 in /home/jblevins32/anaconda3/lib/python3.11/site-packages (fr
om imageio>=2.14.1->gymnasium[mujoco]) (11.1.0)
Requirement already satisfied: absl-py in /home/jblevins32/anaconda3/lib/python3.11/site-packages (from muj
oco>=2.1.5->gymnasium[mujoco]) (2.1.0)
Requirement already satisfied: etils[epath] in /home/jblevins32/anaconda3/lib/python3.11/site-packages (fro
m mujoco>=2.1.5->gymnasium[mujoco]) (1.12.0)
Requirement already satisfied: glfw in /home/jblevins32/anaconda3/lib/python3.11/site-packages (from mujoco
>=2.1.5->gymnasium[mujoco]) (2.8.0)
Requirement already satisfied: pyopengl in /home/jblevins32/anaconda3/lib/python3.11/site-packages (from mu
joco>=2.1.5->gymnasium[mujoco]) (3.1.9)
Requirement already satisfied: fsspec in /home/jblevins32/anaconda3/lib/python3.11/site-packages (from etil
s[epath]->mujoco>=2.1.5->gymnasium[mujoco]) (2024.12.0)
Requirement already satisfied: importlib_resources in /home/jblevins32/anaconda3/lib/python3.11/site-packag
es (from etils[epath]->mujoco>=2.1.5->gymnasium[mujoco]) (6.5.2)
Requirement already satisfied: zipp in /home/jblevins32/anaconda3/lib/python3.11/site-packages (from etils
[epath]->mujoco>=2.1.5->gymnasium[mujoco]) (3.21.0)
```

```
54158.75s - pydevd: Sending message related to process being replaced timed-out after 5 seconds
E: Could not open lock file /var/lib/dpkg/lock-frontend - open (13: Permission denied)
E: Unable to acquire the dpkg frontend lock (/var/lib/dpkg/lock-frontend), are you root?
Using device: cpu
```

## Replay Buffer

This is boiler plate code that lets your off-policy algorithms store their interactions with the environment. **You shouldn't need to change this code.**

In [49]:
```
class ReplayBuffer:
    def __init__(self):
        self.buffer = deque(maxlen=1_000_000)
        self.batch_size = 32

    def store(self, state, action, reward, next_state, done):
        transitions = list(zip(state, action, reward, next_state, 1 - torch.Tensor(done)))
        self.buffer.extend(transitions)

    def sample(self):
        batch = random.sample(self.buffer, self.batch_size)
        return [torch.stack(e).to(device) for e in zip(*batch)]  # states, actions, rewards, next_states, 
```

## DRL Rollout

Boiler plate code that initiates parallel environments and stores your agents $(s, a, r, s')$ interactions in a replay buffer. Also logs some stats to tensorboard. **You shouldn't need to change this code.**

In [50]:
```
class DRL:
    def __init__(self):
        self.n_envs = 32
        self.n_steps = 128

        self.envs = gym.vector.SyncVectorEnv(
            [lambda: gym.make("InvertedPendulum-v5", reset_noise_scale=0.2) for _ in range(self.n_envs)])

        self.replay_buffer = ReplayBuffer()

    def rollout(self, agent, i):
        """Collect experience and store it in the replay buffer"""

        obs = torch.Tensor(self.envs.reset()[0])
```

```
        total_rewards = torch.zeros(self.n_envs)

        for _ in range(self.n_steps):
            with torch.no_grad():
                actions = agent.get_action(obs.to(device), noisy=True).cpu()
            next_obs, rewards, done, truncated, _ = self.envs.step(actions.numpy())
            next_obs, rewards = torch.Tensor(next_obs), torch.Tensor(rewards)
            # reward scaling by .01 keeps sum of rewards near 1, increases stability
            self.replay_buffer.store(obs, actions, rewards*.01, next_obs, done | truncated)
            obs = next_obs

            total_rewards += rewards

        writer.add_scalar("stats/Rewards", total_rewards.mean().item() / self.n_steps, i)
```

In [51]:

```
# @title Visualization code. Used later.
import os
from gym.wrappers import RecordVideo
from IPython.display import Video, display, clear_output

# Force MuJoCo to use EGL for rendering (important for Colab)
os.environ["MUJOCO_GL"] = "egl"

def visualize(agent):
    """Visualize agent with a custom camera angle."""

    # Create environment in rgb_array mode
    env = gym.make("InvertedPendulum-v5", render_mode="rgb_array", reset_noise_scale=0.2)

    # Apply video recording wrapper
    env = RecordVideo(env, video_folder="./", episode_trigger=lambda x: True)

    obs, _ = env.reset()

    # Access the viewer object through mujoco_py
    viewer = env.unwrapped.mujoco_renderer.viewer  # Access viewer
    viewer.cam.distance = 3.0     # Set camera distance
    viewer.cam.azimuth = 90       # Rotate camera around pendulum
    viewer.cam.elevation = 0   # Tilt the camera up/down
```

```
    for t in range(512):
        with torch.no_grad():
            actions = agent.get_action(torch.Tensor(obs).to(device)[None, :])[:, 0]
        obs, _, done, _, _= env.step(actions.cpu().numpy())
        if done:
            break
    env.close()

    # Display the latest video
    clear_output(wait=True)
    display(Video("./rl-video-episode-0.mp4", embed=True))
```

# Tensorboard

This will launch an interactive tensorboard window within collab. It will display rewards in (close to) real time while your agents are training. You'll likely have to refresh if its not updating (circular arrow to right in the orange bar). **You shouldn't need to change this code.**

In [52]:

```
# Launch TensorBoard
%load_ext tensorboard
%tensorboard --logdir runs
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
Reusing TensorBoard on port 6006 (pid 9628), started 19:25:16 ago. (Use '!kill 9628' to kill it.)
```

TensorBoard TIME SERIES SCALARS INACTIVE

Show data download links

Ignore outliers in chart scalin

Tooltip sorting method: default

Smoothing

0.6

Horizontal Axis

STEP RELATIVE

WALL

Runs

Write a regex to filter runs

DQN

DDPG

SAC

TOGGLE ALL RUNS

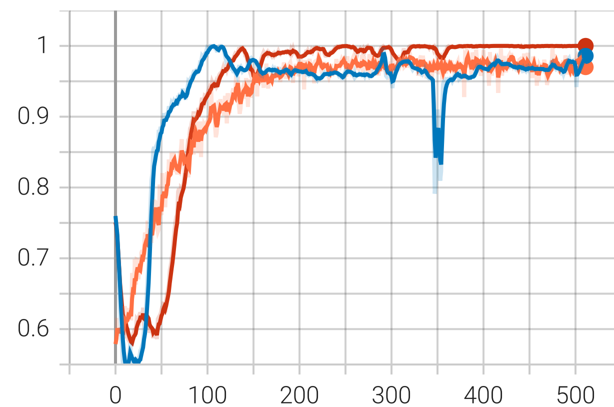
runs

Filter tags (regular expressions supported)

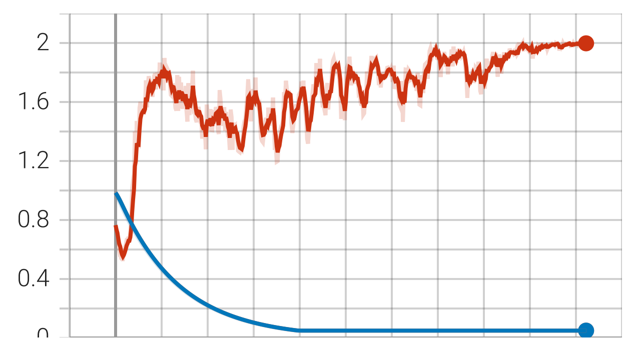
loss 2

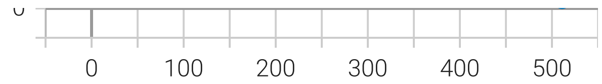
stats 2

stats/Rewards
tag: stats/Rewards




stats/exploration rate
tag: stats/exploration rate






# 1. Deep Q Networks (DQN) [30 pts]

1. Define your Q network and Q target network [5 pts]
2. Define the Q network optimizer [5 pts]
3. Define the Q loss [15 pts]
4. Conceptual questions [5 pts]

---

### Background

DQN is an off-policy reinforcement learning algorithm that extends Q-learning using deep neural networks. It is designed for environments with discrete action spaces and was used to achieve human-level performance in Atari games in a seminal 2013 paper. Its key innovations relative to naive neural fitted Q iteration include replay buffers (which decorrelate samples) and target networks (which give Q learning a stationary target to converge to).

We will use DQN to solve a continuous action space problem by discretizing. We map discete indices $[0, 1]$ to continuous actions $[-3, 3]$ and vis versa. ________________________________

### Temporal Difference Q Loss for DQN:

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=0}^{N}\{Q_\theta(s_t, a_t) - q_{\text{target}}\}^2$$

$$q_{\text{target}} = r_t + \gamma \max_{a_{t+1}} Q_{\theta_{\text{target}}}(s_{t+1}, a_{t+1}) \cdot \text{not\_done}_t$$

or equivalenty, more concisely:

$$\mathcal{L}(\theta) = \mathbb{E}[\{Q_\theta(s,a) - (r_t + \gamma \max_{a'} Q_{\theta_{\text{target}}}(s',a') \cdot \text{not\_done})\}^2]$$

(hint: Don't actually use a for loop. Use torch's batched operations for greater training speed.)

Where:

- $\mathcal{L}$ is the Q net loss; a function of Q network parameters $\theta$
- $N$ is the size of the minibatch

-$Q_\theta$ is the Q network parametrized by $\theta$ -$s_t$ is state at timestep $t$ -$a_t$ is action at timestep $t$ -$r_t$ is reward at timestep $t$ -$\gamma$ is the discount factor on rewards -$Q_{\theta_{\text{target}}}$ is the Q target network parametrized by $\theta_{\text{target}}$ -$s_{t+1}$ or $s'$ is state at timestep $t+1$ (hint: comes from replay buffer) -$a_{t+1}$ or $a'$ is action at timestep $t+1$ (hint: implied from max next Q) -$\text{not\_done}_t$ or $\text{not\_done}$ is the not done flag for timestep $t$ indicating state $s_t$ is not terminal (i.e. Q next should be considered) -$\mathbb{E}$ is the expectation or average over the minibatch

In [53]:
```
class DQN:
    def __init__(self, n_obs, n_actions):
        self.n_actions = n_actions
        self.exploration_rate = 1.
        torch.manual_seed(0)  # for fair comparison

        # todo: student code here
        # Define Q-network, hint: dont forget .to(device), use atleast 1 nonlinear activation. Outputs valu
        self.q_net = nn.Sequential(
            nn.Linear(n_obs, 64),
            nn.ReLU(),
            nn.Linear(64, 128),
            nn.ReLU(),
            nn.Linear(128, n_actions)
        ).to(device)

        # Define Q-target network, hint: deepcopy
        self.q_target_net = copy.deepcopy(self.q_net).to(device)

        # Define optimizer, hint: Adam, learning rate 3e-4 is a good place to start but feel free to try o
        self.optimizer = Adam(params=self.q_net.parameters(), lr=3e-4)
        # end student code
```

```
    def get_action(self, states, noisy=False):
        if noisy and torch.rand(1).item() < self.exploration_rate:
            # Random action with probability self.exploration_rate
            actions = torch.randint(0, self.n_actions, (states.shape[0],1))
        else:
            # Greedy action selection
            with torch.no_grad(): actions = self.q_net(states).argmax(dim=-1, keepdim=True)
        return (actions.float() - 0.5) * 6  # maps discrete [0, 1] to continuous [-3., 3.]


    def get_q_loss(self, states, actions, rewards, next_states, not_dones, gamma=.99):
        actions = (actions/6 + .5).long() # maps continous [-3., 3.] to discrete [0, 1]

        # todo: student code here
        with torch.no_grad():
            # hint: compute Q for all next states using q target network
            # States: batch_size x n_obs
            all_next_Q = self.q_target_net(next_states)

        # hint: take the max next q over actions
        max_next_Q, _ = torch.max(all_next_Q, dim=-1)

        # hint: calculate q_target equation
        q_target = rewards + gamma*max_next_Q*not_dones

        # hint: compute the q values of all actions in state using q network
        all_Q = self.q_net(states)

        # hint: gather the q values of the actions that were actually taken
        Q = all_Q.gather(1, actions).squeeze(-1)

        # hint: compute Mean Squared Error loss between Q and q_target
        criterion = nn.MSELoss()
        loss = criterion(Q,q_target)
        # end student code

        return loss


    def update(self, replay_buffer, i):
        for _ in range(64):
```

```
            loss = self.get_q_loss(*replay_buffer.sample())
            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()
        writer.add_scalar("loss/q loss", loss.item(), i)

        # Periodic hard update Q-target network to Q-network
        if i % 16 == 0:
            self.q_target_net.load_state_dict(self.q_net.state_dict())

        # decay and log exploration rate
        self.exploration_rate = max(self.exploration_rate * 0.985, 0.05)
        writer.add_scalar("stats/exploration rate", self.exploration_rate, i)
```

In [54]:

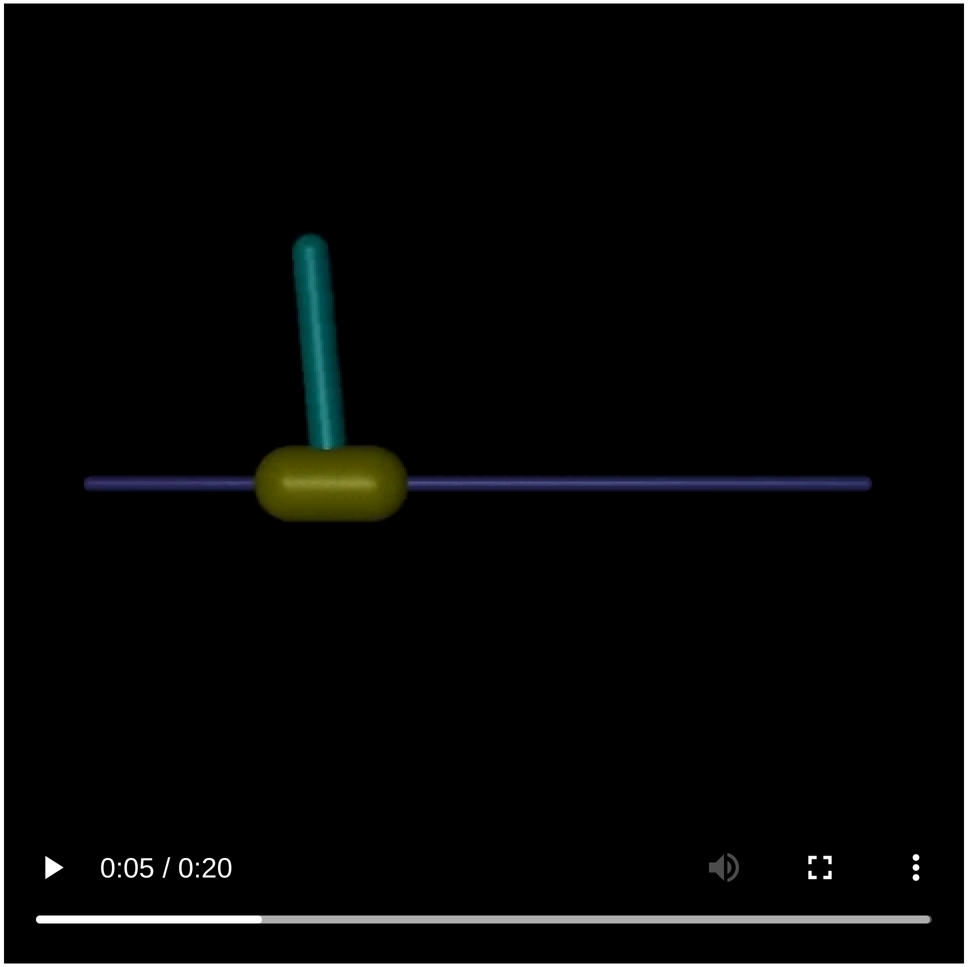
```
# @title DQN Unit Tests (must run DQN Agent cell above first)
def DQN_q_net():
    a = DQN(16, 7)
    assert a.q_net is not None, "q_net not initialized"
    assert a.q_target_net is not None, "q_target_net is not initialized"
    r = torch.randn(8, 16).to(device)
    assert a.q_net(r).shape == (8, 7) and \
    isinstance(list(a.q_net.children())[-1], nn.Linear), \
    f"Network not initialized correctly"
    assert a.q_target_net(r).shape == (8, 7) and \
    isinstance(list(a.q_target_net.children())[-1], nn.Linear), \
    f"Networks not initialized correctly"
    print("Test passed: DQN Q nets appears correct!")
DQN_q_net()

def test_DQN_optimizer():
    a = DQN(16, 7)
    assert a.optimizer is not None, "Optimizer is not initialized"
    assert set(p for p in a.q_net.parameters())  == \
    set(p for group in a.optimizer.param_groups for p in group['params']),\
    "Optimizer does not match q_net parameters"
    print("Test passed: DQN Optimizer appears correct!")
test_DQN_optimizer()

def DQN_loss():
    torch.manual_seed(0)
    # these dont match an actual rollout..
    # print debug values during training loop rather than unit tests
```

```
    batch_size, n_obs, n_actions = 5, 4, 1
    s = torch.rand((batch_size, n_obs))
    a = (torch.randint(0, 2, (batch_size, n_actions)).float() - 0.5) * 6
    r = torch.rand((batch_size,))
    s_ = torch.rand((batch_size, n_obs))
    not_dones = torch.randint(0, 2, (batch_size,)).float()

    dqn = DQN(4, 2)
    torch.manual_seed(0)
    dqn.q_net = nn.Linear(4, 2) # you should not use this architecture..
    dqn.q_target_net = nn.Linear(4, 2)
    loss = dqn.get_q_loss(s, a, r, s_, not_dones)
    assert abs(loss.item() - (0.1567)) < 1e-4, \
    "DQN loss does not match expected value."
    print("Test passed: DQN loss appears correct!")

DQN_loss()
```

```
Test passed: DQN Q nets appears correct!
Test passed: DQN Optimizer appears correct!
Test passed: DQN loss appears correct!
```

In [55]:
```
# tensorboard label can be changed with e.g. f'runs/unique_hyperparam_test'
writer = SummaryWriter(log_dir=f'runs/DQN')

drl = DRL()
dqn = DQN(n_obs=4, n_actions=2)

# takes ~5-10 minutes on google colab gpus
for i in range(512):

    drl.rollout(dqn, i)
    dqn.update(drl.replay_buffer, i)
```

In [56]:
```
visualize(dqn) # run again to see a different rollout
print("DQN")
```




DQN

**DQN Conceptual Question 1 - Target Networks:**

DQN uses target networks to stabilize training. DQN Loss:

$$\mathcal{L}(\theta) = \mathbb{E}[\{Q_\theta(s,a) - (r_t + \gamma \max_{a'} Q_{\theta_{\text{target}}}(s',a') \cdot \text{not\_done})\}^2]$$

But, what if you didn't use a target network:

$$\mathcal{L}(\theta) = \mathbb{E}[\{Q_\theta(s,a) - (r_t + \gamma \max_{a'} Q_\theta(s',a') \cdot \text{not\_done})\}^2]$$

(Optionally, make this actual change in code and observe training results for yourself. Its a one line change. Make sure to revert or comment it before submitting. Any code change is not for credit, but may aid understanding. You can change the name of the tensorboard run for direct visual comparison.)

In 1 or 2 sentences, what could happen to your Q loss over the course of training if you modified the DQN loss equation so that a target network is not used? Why?

**Type answer here...**

**DQN Conceptual Question 2 - Double DQN:**

The loss function for Double DQN improves upon standard DQN by using the Q-network to select the best action and the target Q-network to evaluate it:

$$\mathcal{L}_{\text{double dqn}}(\theta) = \mathbb{E}\{(Q_\theta(s,a) - [r_t + \gamma Q_{\theta_{\text{target}}}(s', \arg\max_{a'} Q_\theta(s',a')) \cdot \text{not\_done}])^2\}$$

(Optionally, implement this in code and observe the training, but comment or revert changes before submitting.)

In 1 or 2 sentences, explain the intuition behind why this might improve performance.

**Type answer here**...

---

# 2. Deep Deterministic Policy Gradients (DDPG) [40 pts]

1. Define your Q network, Q target network, policy network, and policy target network [5 pts]
2. Define the Q network optimizer and policy network optimizer [5 pts]
3. Define the Q loss [15 pts] and policy loss [10 pts]
4. Conceptual questions [5 pts]

---

### Background

DDPG is an off-policy reinforcement learning algorithm that extends DQN to continuous action spaces. It is based off a theortical publication called Deterministic Policy Gradients. It solved many robotics tasks in a seminal 2015 publication. Its key innovations relative to DQN are (1) a policy network which is trained to produce deterministic, continous actions that maximize the Q function, and (2) soft target updates.

We will use it to solve a continuous action space environment natively, without discretization.

---

## DQN vs DDPG

**Q networks**: Q networks in DQN take in states and output the Q value for each action. Q networks in the continuous case take in both the state and action and output a single Q estimate.

**Policies**: The policy in DQN comes from taking the action corresponding to the max Q value over discrete options. The policy in DDPG comes from training a network which takes in states/observations and outputs continuous actions that are trained to maximize Q. Since the policy approximates the max operator, explicit $\max_{a'}$ is dropped from the Temporal Difference Q loss.

---

## Temporal Difference Q Loss for DDPG:

$$\mathcal{L}(\theta) = \mathbb{E}[\{Q_\theta(s,a) - (r_t + \gamma Q_{\theta_{\text{target}}}(s', a') \cdot \text{not\_done})\}^2]$$

Where:

- $\mathcal{L}$ is the Q net loss; a function of Q network parameters $\theta$
- $E$ is the expectation or average over the minibatch

-$Q_\theta$ is the Q network parametrized by $\theta$ -$s_t$ is state at timestep $t$ -$a_t$ is action at timestep $t$ -$r_t$ is reward at timestep $t$ -$\gamma$ is the discount factor on rewards -$Q_{\theta_{\text{target}}}$ is the Q target network parametrized by $\theta_{\text{target}}$ -$s'$ is state at timestep $t+1$ (hint: comes from replay buffer) -$a'$ is action at timestep $t+1$ (hint: comes from policy target network applied to next state. get_target_action())

- $E$ is the expectation or average over the minibatch

---

## Policy Loss for DDPG:

$$\mathcal{L}(\theta_p) = -\mathbb{E}[Q_\theta(s, a)]$$

Where:

- $\mathcal{L}$ is the policy loss; a function of policy parameters $\theta_p$
- $E$ is the expectation or average over the minibatch

-$Q_\theta$ is the Q network parametrized by $\theta$

- $s$ is state
- $a$ is the deterministic action the policy would take in state $s$ (hint: get_action())

In [57]:
```
class DDPG:
    def __init__(self, n_obs, n_actions):
        self.exploration_rate = 1.
        torch.manual_seed(0)

        # todo: student code here
        self.q_net = nn.Sequential(
            nn.Linear(n_obs+n_actions,64),
            nn.ReLU(),
            nn.Linear(64,1)
        ).to(device)

        self.policy = nn.Sequential(
            nn.Linear(n_obs,64),
            nn.ReLU(),
            nn.Linear(64,n_actions)
        ).to(device)

        self.q_target_net = copy.deepcopy(self.q_net).to(device)
        self.policy_target_net = copy.deepcopy(self.policy).to(device)

        self.q_optimizer = Adam(params=self.q_net.parameters(),lr=3e-3)
        self.policy_optimizer = Adam(params=self.policy.parameters(),lr=3e-3)
        # end student code


    def get_action(self, states, noisy=False):
        actions = self.policy(states)
```

```
        if noisy:
          actions += torch.normal(0, self.exploration_rate, size=actions.shape).to(device)
        return actions.clamp(-3, 3)

    def get_target_action(self, next_states):
        actions = self.policy_target_net(next_states)
        return actions.clamp(-3, 3)

    def get_q_loss(self, states, actions, rewards, next_states, not_dones, gamma=.99):

        #todo: student code here
        with torch.no_grad():

            # 1) Get next actions from target policy
            next_actions = self.policy_target_net(next_states)

            # 2) Get target value from next states and next actions
            next_state_action_vec = torch.cat((next_states,next_actions),dim=-1)
            q_next = self.q_target_net(next_state_action_vec)

        # 3) Get value from current states and current actions
        state_action_vec = torch.cat((states,actions),dim=-1)
        q = self.q_net(state_action_vec)

        # 4) Get target q
        q_target = rewards.unsqueeze(-1) + gamma*q_next*not_dones.unsqueeze(-1)

        # 5) Get q loss
        criterion = nn.MSELoss()
        Q_loss = criterion(q,q_target)
        # end student code

        return Q_loss


    def get_policy_loss(self, states):
        # todo: student code here
        # 1) Get actions
        actions = self.policy(states)
        state_action_vec = torch.cat((states,actions),dim=-1)

        policy_loss = -torch.mean(self.q_net(state_action_vec))
```

```
        # end student code
        return policy_loss


    def update(self, replay_buffer, i):

        for _ in range(64):
            loss = self.get_q_loss(*replay_buffer.sample())
            self.q_optimizer.zero_grad()
            loss.backward()
            self.q_optimizer.step()
        writer.add_scalar("loss/q loss", loss.item(), i)

        for _ in range(4):
            states, _, _, _, _ = replay_buffer.sample()
            loss = self.get_policy_loss(states)
            self.policy_optimizer.zero_grad()
            loss.backward()
            self.policy_optimizer.step()
        writer.add_scalar("loss/ - policy loss", -loss.item(), i)

        tau = 0.1  # Continual soft target update
        for target_param, param in zip(self.q_target_net.parameters(), self.q_net.parameters()):
            target_param.data.copy_(tau * param.data + (1 - tau) * target_param.data)

        for target_param, param in zip(self.policy_target_net.parameters(), self.policy.parameters()):
            target_param.data.copy_(tau * param.data + (1 - tau) * target_param.data)

        self.exploration_rate  = max(self.exploration_rate  * 0.985, 0.05)
        writer.add_scalar("stats/exploration rate", self.exploration_rate , i)
```

In [58]:

```
# @title DDPG Unit Tests (must run DDPG Agent cell above first)
def DDPG_nets():
    a = DDPG(16, 7)
    assert a.q_net is not None, "q_net not initialized"
    assert a.q_target_net is not None, "q_target_net is not initialized"
    assert a.policy is not None, "policy is not initialized"
    assert a.policy_target_net is not None, "policy_target_net is not initialized"
    r = torch.randn(8, 23).to(device)

    # this doesnt check if target is the same architecture as q
```

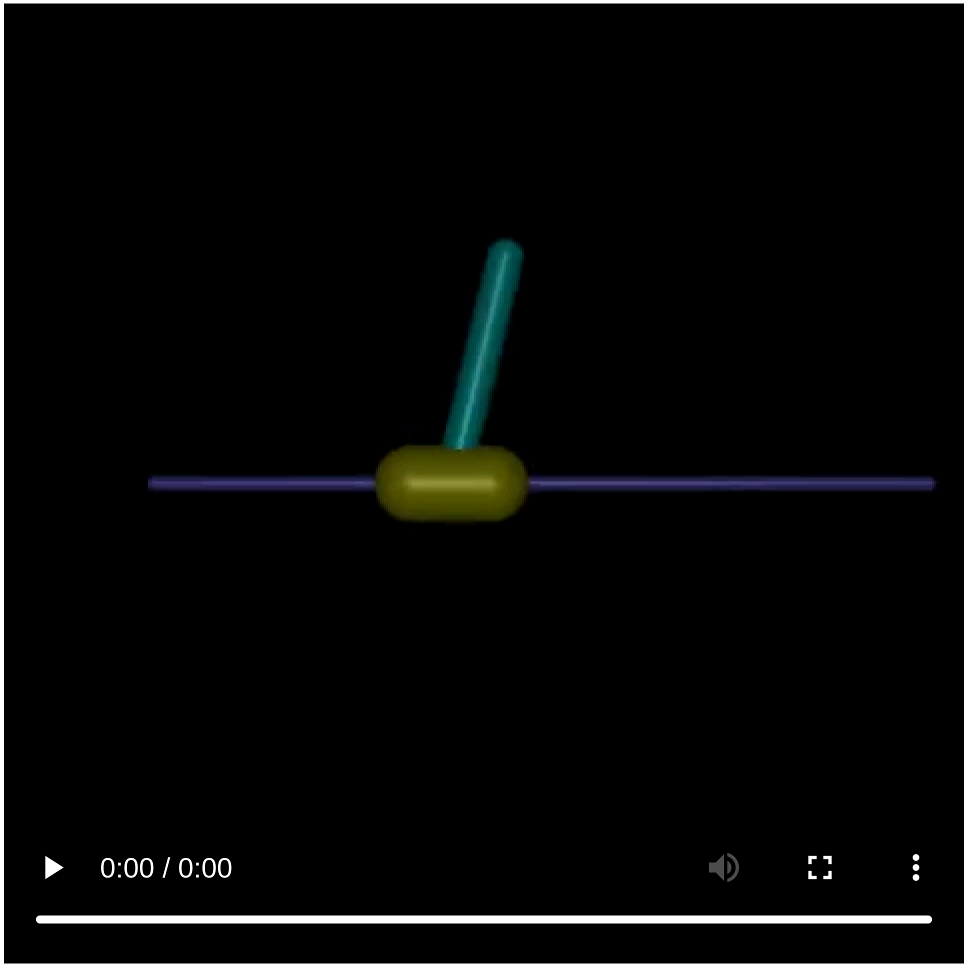
```
    # but it should be
    assert a.q_net(r).shape == (8, 1) and \
    isinstance(list(a.q_net.children())[-1], nn.Linear), \
    f"Network not initialized correctly"
    assert a.q_target_net(r).shape == (8, 1) and \
    isinstance(list(a.q_target_net.children())[-1], nn.Linear), \
    f"Networks not initialized correctly"

    r = torch.randn(8, 16).to(device)
    assert a.policy(r).shape == (8, 7), \
    f"Networks not initialized correctly"
    assert a.policy_target_net(r).shape == (8, 7), \
    f"Networks not initialized correctly"
    print("Test passed: DDPG nets appear correct!")
DDPG_nets()

def test_DDPG_optimizer():
    a = DDPG(16, 7)
    assert a.q_optimizer is not None, "Q Optimizer is not initialized"
    assert a.policy_optimizer is not None, "Policy Optimizer is not initialized"

    assert set(p for p in a.q_net.parameters())  == \
    set(p for group in a.q_optimizer.param_groups for p in group['params']),\
    "Q optimizer does not match q_net parameters"

    assert set(p for p in a.policy.parameters())  == \
    set(p for group in a.policy_optimizer.param_groups for p in group['params']),\
    "Policy optimizer does not match policy parameters"

    print("Test passed: DDPG optimizer appears correct!")
test_DDPG_optimizer()

def DDPG_q_loss():
    torch.manual_seed(0)

    # these dont match an actual rollout..
    # print debug values during training loop rather than unit tests
    batch_size, n_obs, n_actions = 5, 4, 1
    s = torch.rand((batch_size, n_obs))
    a = (torch.rand((batch_size, n_actions)) - 0.5) * 6
    r = torch.rand((batch_size,))
    s_ = torch.rand((batch_size, n_obs))
```

```
    not_dones = torch.randint(0, 2, (batch_size,))

    ddpg = DDPG(4, 1)
    torch.manual_seed(0)
    ddpg.q_net = nn.Linear(5, 1) # you should not use this architecture..
    ddpg.q_target_net = nn.Linear(5, 1)
    ddpg.policy = nn.Linear(4, 1)
    ddpg.policy_target_net = nn.Linear(4, 1)
    loss = ddpg.get_q_loss(s, a, r, s_, not_dones)
    # print(loss)
    assert abs(loss.item() - (0.6036)) < 1e-4, \
    "DDPG q loss does not match expected value."
    print("Test passed: DDPG q loss appears correct!")
DDPG_q_loss()

def DDPG_policy_loss():
    torch.manual_seed(0)

    batch_size, n_obs, n_actions = 5, 4, 1
    s = torch.rand((batch_size, n_obs))

    ddpg = DDPG(4, 1)
    torch.manual_seed(0)
    ddpg.q_net = nn.Linear(5, 1) # you should not use this architecture..
    ddpg.q_target_net = nn.Linear(5, 1)
    ddpg.policy = nn.Linear(4, 1)
    ddpg.policy_target_net = nn.Linear(4, 1)
    loss = ddpg.get_policy_loss(s)
    # print(loss)
    assert abs(loss.item() - (-0.0553)) < 1e-4, \
    "DDPG policy loss does not match expected value."
    print("Test passed: DDPG policy loss appears correct!")
DDPG_policy_loss()
```

```
Test passed: DDPG nets appear correct!
Test passed: DDPG optimizer appears correct!
Test passed: DDPG q loss appears correct!
Test passed: DDPG policy loss appears correct!
Test passed: DDPG q loss appears correct!
Test passed: DDPG policy loss appears correct!
```

In [64]:
```
# DDPG training loop

# tensorboard label can be changed with e.g. f'runs/unique_hyperparam_test'
writer = SummaryWriter(log_dir=f'runs/DDPG')

drl = DRL()
ddpg = DDPG(n_obs=4, n_actions=1)

# takes ~5-10 minutes on colab gpus
for i in range(512):

    drl.rollout(ddpg, i)
    ddpg.update(drl.replay_buffer, i)
```

In [60]:
```
visualize(ddpg)
```




**DDPG Conceptual Question 1 - Optimizers:**

In HW1 we used a single combined optimizer for both value and policy nets. For DDPG, we need separate optimizers for Q and Policy nets. Why is that? (hint: policy loss)

**Type answer here...**

**DDDP Conceptual Question 1 - Replay Buffers:**

Every policy rollout (defined in boiler plate code) uses 32 parallel environments simulated for 128 timesteps.

```
class DRL:
    def __init__(self):
        self.n_envs = 32
        self.n_steps = 128
```

Additionally, our replay buffer (defined in boiler plate code) is large enough to hold 1,000,000 transitions.

```
class ReplayBuffer:
    def __init__(self):
        self.buffer = deque(maxlen=1_000_000)
        self.batch_size = 32
```

In 1 or 2 sentences, what might happen to our training speed and stability if we collected less data per rollout and used a smaller replay buffer? Why? Lets say 1 environment, 32 steps, size 32 replay buffer. (Optionally, make these changes in code and observe training results yourselves. Note, if you test lower than 32 transitions you need to reduce ReplayBuffer.batch_size aswell. Comment or revert changes before submitting.)

**Type answer here...**

---

# 3. Optional Extra Credit: Soft Actor-Critic (SAC) [10 pts]

1. Implement a stochastic policy [4 pts]
2. Implement double Q Learning [3 pts]
3. Implement entropy regularization [3 pts]

---

## Background

SAC is a reinforcement learning algorithm that improves DDPG with better stability and exploration. It was introduced in a seminal publication in 2017, and is often considered the go-to model-free off-policy method. Its key innovations relative to DDPG are a stochastic policy, double Q learning, and entropy regularization.

We will use SAC to solve a continuous action space environment more robustly than DDPG.

## DDPG vs SAC

**Determinism vs Stochasticity**: DDPG trains a deterministic policy network, which maps states to single continuous actions. Exploration noise has to be injected externally. SAC, on the other hand, learns a stochastic policy represented by a probability distribution over actions. It inherently explores. Additionally, DDPG can overfit to quirks of the q function, which can cause instability, premature convergence, or collapse. In contrast, SAC has stochasticity built into the loss equations, which results in an averaging effect that makes SAC networks less brittle and more stable.

**Double Q**: DDPG uses one Q network. Building on the insight from Double DQN ($\mathbb{E}[\max(C_1, C_2)] \geq \max(\mathbb{E}[C_1], \mathbb{E}[C_2])$), SAC learns two separate Q networks, and uses them to mitigate over estimation bias.

**Exploration and Entropy**: However exploration noise is injected in DDPG, it is state independent. SAC has adaptive state dependent exploration. As a function of state, its policy outputs the mean and log standard deviation of a guassian policy. The backprop process naturally produces broad guassians when q values are uncertain, and narrow guassians as q values converge. Furthermore, SAC uses entropy regularization to further encourage broad guassians which discourages premature suboptimal convergence.

---

#### Milestone 1 - Stochastic Policy

Re-implement DDPG with a stochastic policy, single q, no entropy.

1. Copy-Paste your DDPG code (networks, optimizers, loss functions)

- update policy net output to be twice as large as before

2. Finish implementing get_action() and get_target_action()

- construct a torch.Normal distribution using state dependent mean and std_dev
- *rsample* actions (has to be rsample not sample for differentiability)
- clamp actions to the valid range $[-3, 3]$

3. Update DDPG losses for the stochastic policy. These are the same equations as DDPG, but the actions are now stochastically sampled.

- Q Loss

$$\mathcal{L}(\theta) = \mathbb{E}[Q_\theta(s,a) - (r_t + \gamma(Q_{\theta_{\text{target}}}(s',a'))]^2$$

where $a'$ comes from calling get_target_action() on $s'$ with noisy=True.

- Policy Loss

$$\mathcal{L}(\theta_p) = -\mathbb{E}[Q_\theta(s,a)]$$

where $a$ comes from calling get_action() on $s$ with noisy=True.

If you can pass the M1 unit test below, and can run a succesful training at this point, you've earned 4 points!

**Milestone 2 - Double Q**

Upgrade to Double Q learning for reduced overestimation bias.

1. Update Q Networks

- Replace Q and Q_target with Q1, Q2, Q1_target, Q2_target.
- Update q_optimizer to hold parameters for Q1 and Q2

2. Update Loss functions

- Q loss : Evaluate both q target nets on $s'$, use the minimum in constructing $q_{\text{target}}$. Regress both networks to $q_{\text{target}}$, by adding their MSE losses.

$$q_{\text{target}} = r_t + \gamma \min_{i=1,2} Q_{\theta_{\text{target},i}}(s',a') \cdot \text{not\_done}$$
$$\mathcal{L}(\theta) = \mathbb{E}[\{Q_{\theta_1}(s,a) - q_{\text{target}}\}^2] + \mathbb{E}[\{Q_{\theta_2}(s,a) - q_{\text{target}}\}^2]$$

- Policy loss : Same as before but use Q1

$$\mathcal{L}(\theta_p) = -\mathbb{E}[Q_{\theta_1}(s,a)]$$

3. Modify the soft target updates in the update function to work for both Q1 and Q2

If you can pass the M2 unit test below, and can run a succesful training at this point, you've earned 3 more points! (7 total)

**Milestone 3 - Entropy Regularization**

Upgrade to Entropy Regularization for better exploration.

1. Update Q Loss : add an entropy term to $q_{\text{target}}$

$$q_{\text{target}} = r_t + \gamma\{\min_{i=1,2} Q_{\theta_{\text{target},i}}(s', a') + \alpha H(\pi(s'))\} \cdot \text{not\_done}$$

where $\alpha$ is a scaling *temperature* and $H$ is entropy of policy $\pi$ at state $s'$ (hint: get_entropy function)

2. Update Policy Loss :

$$\mathcal{L}(\theta_p) = -\mathbb{E}[Q_{\theta_1}(s, a) + \alpha H(\pi(s))]$$

If you can pass the M3 unit test below, and can run a succesful training at this point, you've earned 3 more points! (10 total)

In [61]:
```
# THIS IS FOR M3

from torch.distributions import Normal

class SAC:
    def __init__(self, n_obs, n_actions):
        torch.manual_seed(0)
        self.alpha = .002

        # todo: student code here
        self.q1_net = nn.Sequential(
            nn.Linear(n_obs+n_actions,64),
            nn.ReLU(),
            nn.Linear(64,1)
        ).to(device)

        self.q2_net = nn.Sequential(
            nn.Linear(n_obs+n_actions,64),
            nn.ReLU(),
            nn.Linear(64,1)
        ).to(device)
```

```
        self.policy = nn.Sequential(
            nn.Linear(n_obs,64),
            nn.ReLU(),
            nn.Linear(64,n_actions*2)
        ).to(device)

        self.q1_target_net = copy.deepcopy(self.q1_net).to(device)
        self.q2_target_net = copy.deepcopy(self.q1_net).to(device)
        self.policy_target_net = copy.deepcopy(self.policy).to(device)

        params_combined = list(self.q1_net.parameters()) + list(self.q2_net.parameters())
        self.q_optimizer = Adam(params=params_combined,lr=3e-3)
        self.policy_optimizer = Adam(params=self.policy.parameters(),lr=3e-3)
        # end student code

    def get_entropy(self, states):
        mean, log_std_dev = self.policy(states).chunk(2, dim=-1)
        std_dev = log_std_dev.exp().clamp(.2, 2)
        H = Normal(mean, std_dev).entropy()
        return H

    def get_action(self, states, noisy=False):
        mean, log_std_dev = self.policy(states).chunk(2, dim=-1)
        if noisy == False:
            return mean
        else:
            std_dev = log_std_dev.exp().clamp(.2, 2)

            #todo: student code
            # Get the action
            dist = Normal(mean, std_dev)
            action = dist.rsample().clamp(-3,3)
            return action

    def get_target_action(self, states, noisy=False):
        mean, log_std_dev = self.policy_target_net(states).chunk(2, dim=-1)
        if noisy == False:
            return mean
        else:
            std_dev = log_std_dev.exp().clamp(.2, 2)

            #todo: student code
```

```
            # Get the action
            dist = Normal(mean, std_dev)
            action = dist.rsample().clamp(-3,3)
            return action

    def get_q_loss(self, states, actions, rewards, next_states, not_dones, gamma=.99):

        #todo: student code here
        with torch.no_grad():

            # 1) Get next actions from target policy
            next_actions = self.get_target_action(next_states, noisy=True)

            # 2) Get target value from next states and next actions
            next_state_action_vec = torch.cat((next_states,next_actions),dim=-1)
            q1_next = self.q1_target_net(next_state_action_vec)
            q2_next = self.q2_target_net(next_state_action_vec)

        # 3) Get value from current states and current actions
        state_action_vec = torch.cat((states,actions),dim=-1)
        q1 = self.q1_net(state_action_vec)
        q2 = self.q2_net(state_action_vec)

        # 4) Get target q, starting by getting  entropy H
        mean, log_std_dev = self.policy(next_states).chunk(2, dim=-1)
        std_dev = log_std_dev.exp().clamp(.2, 2)
        dist = Normal(mean, std_dev)
        H = dist.entropy()

        q_next = torch.min(q1_next,q2_next)
        q_target = rewards.unsqueeze(-1) + (gamma*q_next + self.alpha*H)*not_dones.unsqueeze(-1)

        # 5) Get q loss
        criterion = nn.MSELoss()
        Q_loss = criterion(q1,q_target) + criterion(q2,q_target)
        # end student code

        return Q_loss

    def get_policy_loss(self, states):
        # todo: student code here
        # 1) Get actions
```

```
        actions = self.get_action(states, noisy=True)
        state_action_vec = torch.cat((states,actions),dim=-1)

        mean, log_std_dev = self.policy(states).chunk(2, dim=-1)
        std_dev = log_std_dev.exp().clamp(.2, 2)
        dist = Normal(mean, std_dev)
        H = dist.entropy()

        policy_loss = -torch.mean(self.q1_net(state_action_vec) + self.alpha*H)

        # end student code
        return policy_loss

    def update(self, replay_buffer, i):

        for _ in range(64):
            loss = self.get_q_loss(*replay_buffer.sample())
            self.q_optimizer.zero_grad()
            loss.backward()
            self.q_optimizer.step()
        writer.add_scalar("loss/q loss", loss.item(), i)

        for _ in range(4):
            states, _, _, _, _ = replay_buffer.sample()
            loss = self.get_policy_loss(states)
            self.policy_optimizer.zero_grad()
            loss.backward()
            self.policy_optimizer.step()
        writer.add_scalar("loss/ - policy loss", -loss.item(), i)

        # exploration rate logging
        with torch.no_grad():
            _, log_std_dev = self.policy(states).chunk(2, dim=-1)
        std_dev = log_std_dev.exp().clamp(.2, 2)
        writer.add_scalar("stats/exploration rate", std_dev.mean().item(), i)

        tau = 0.1  # Soft update factor  # student code here for M2
        for target_param, param in zip(self.q1_target_net.parameters(), self.q1_net.parameters()):
            target_param.data.copy_(tau * param.data + (1 - tau) * target_param.data)

        for target_param, param in zip(self.q2_target_net.parameters(), self.q2_net.parameters()):
            target_param.data.copy_(tau * param.data + (1 - tau) * target_param.data)
```

```
        for target_param, param in zip(self.policy_target_net.parameters(), self.policy.parameters()):
            target_param.data.copy_(tau * param.data + (1 - tau) * target_param.data)
```

Here's some SAC unit tests. Its not possible to pass them all with the same code. You can either make separate classes to pass each one, or simply edit the one SAC class repeatedly to get the highest milestone. You get cumulative credit for the highest milestone you acheive. Work on M1, then once you pass, work on M2, then M3.

In [62]:
```
# @title SAC Milestone 1 loss unit tests
def SAC_M1_losses():
    torch.manual_seed(0)

    # these dont match an actual rollout..
    # print debug values during training loop rather than unit tests
    batch_size, n_obs, n_actions = 5, 4, 1
    s = torch.rand((batch_size, n_obs))
    a = (torch.rand((batch_size, n_actions)) - 0.5) * 6
    r = torch.rand((batch_size,))
    s_ = torch.rand((batch_size, n_obs))
    not_dones = torch.randint(0, 2, (batch_size,))

    sac = SAC(4, 1)
    torch.manual_seed(0)
    sac.q_net = nn.Linear(5, 1) # you should not use this architecture..
    sac.q_target_net = nn.Linear(5, 1)
    sac.policy = nn.Linear(4, 2)
    sac.policy_target_net = nn.Linear(4, 2)
    q_loss = sac.get_q_loss(s, a, r, s_, not_dones)
    # print(q_loss)
    assert abs(q_loss.item() - (0.7857)) < 1e-4, \
    "SAC M1 q loss does not match expected value."
    print("Test passed: SAC M1 q loss appears correct!")

    p_loss = sac.get_policy_loss(s)
    # print(p_loss)
    assert abs(p_loss.item() - (0.2232)) < 1e-4, \
    "SAC M1 policy loss does not match expected value."
    print("Test passed: SAC M1 policy loss appears correct!")

SAC_M1_losses()
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[62], line 32
     28     assert abs(p_loss.item() - (0.2232)) < 1e-4, \
     29     "SAC M1 policy loss does not match expected value."
     30     print("Test passed: SAC M1 policy loss appears correct!")
---> 32 SAC_M1_losses()

Cell In[62], line 22, in SAC_M1_losses()
     20 q_loss = sac.get_q_loss(s, a, r, s_, not_dones)
     21 # print(q_loss)
---> 22 assert abs(q_loss.item() - (0.7857)) < 1e-4, \
     23 "SAC M1 q loss does not match expected value."
     24 print("Test passed: SAC M1 q loss appears correct!")
     26 p_loss = sac.get_policy_loss(s)

AssertionError: SAC M1 q loss does not match expected value.
```

In [ ]:

```python
# @title SAC Milestone 2 loss unit tests
def SAC_M2_losses():
    torch.manual_seed(0)

    # these dont match an actual rollout..
    # print debug values during training loop rather than unit tests
    batch_size, n_obs, n_actions = 5, 4, 1
    s = torch.rand((batch_size, n_obs))
    a = (torch.rand((batch_size, n_actions)) - 0.5) * 6
    r = torch.rand((batch_size,))
    s_ = torch.rand((batch_size, n_obs))
    not_dones = torch.randint(0, 2, (batch_size,))

    sac = SAC(4, 1)
    torch.manual_seed(0)
    sac.q1_net = nn.Linear(5, 1) # you should not use this architecture..
    sac.q1_target_net = nn.Linear(5, 1)
    sac.q2_net = nn.Linear(5, 1)
    sac.q2_target_net = nn.Linear(5, 1)
    sac.policy = nn.Linear(4, 2)
    sac.policy_target_net = nn.Linear(4, 2)
    q_loss = sac.get_q_loss(s, a, r, s_, not_dones)
    # print(q_loss)
    assert abs(q_loss.item() - (1.0490)) < 1e-4, \
```

```
    "SAC M2 q loss does not match expected value."
    print("Test passed: SAC M2 q loss appears correct!")

    p_loss = sac.get_policy_loss(s)
    # print(p_loss)
    assert abs(p_loss.item() - (-0.1319)) < 1e-4, \
    "SAC M2 policy loss does not match expected value."
    print("Test passed: SAC M2 policy loss appears correct!")

SAC_M2_losses()
```

```
Test passed: SAC M2 q loss appears correct!
Test passed: SAC M2 policy loss appears correct!
```

In [ ]:

```
# @title SAC Milestone 3 loss unit tests
def SAC_M3_losses():
    torch.manual_seed(0)

    # these dont match an actual rollout..
    # print debug values during training loop rather than unit tests
    batch_size, n_obs, n_actions = 5, 4, 1
    s = torch.rand((batch_size, n_obs))
    a = (torch.rand((batch_size, n_actions)) - 0.5) * 6
    r = torch.rand((batch_size,))
    s_ = torch.rand((batch_size, n_obs))
    not_dones = torch.randint(0, 2, (batch_size,))

    sac = SAC(4, 1)
    torch.manual_seed(0)
    sac.q1_net = nn.Linear(5, 1) # you should not use this architecture..
    sac.q1_target_net = nn.Linear(5, 1)
    sac.q2_net = nn.Linear(5, 1)
    sac.q2_target_net = nn.Linear(5, 1)
    sac.policy = nn.Linear(4, 2)
    sac.policy_target_net = nn.Linear(4, 2)
    q_loss = sac.get_q_loss(s, a, r, s_, not_dones)
    # print(q_loss)
    assert abs(q_loss.item() - (1.0530)) < 1e-4, \
    "SAC M3 q loss does not match expected value."
    print("Test passed: SAC M3 q loss appears correct!")

    p_loss = sac.get_policy_loss(s)
```

```
    # print(p_loss)
    assert abs(p_loss.item() - (-0.1341)) < 1e-4, \
    "SAC M3 policy loss does not match expected value."
    print("Test passed: SAC M3 policy loss appears correct!")

SAC_M3_losses()
```

```
Test passed: SAC M3 q loss appears correct!
Test passed: SAC M3 policy loss appears correct!
```

In [63]:

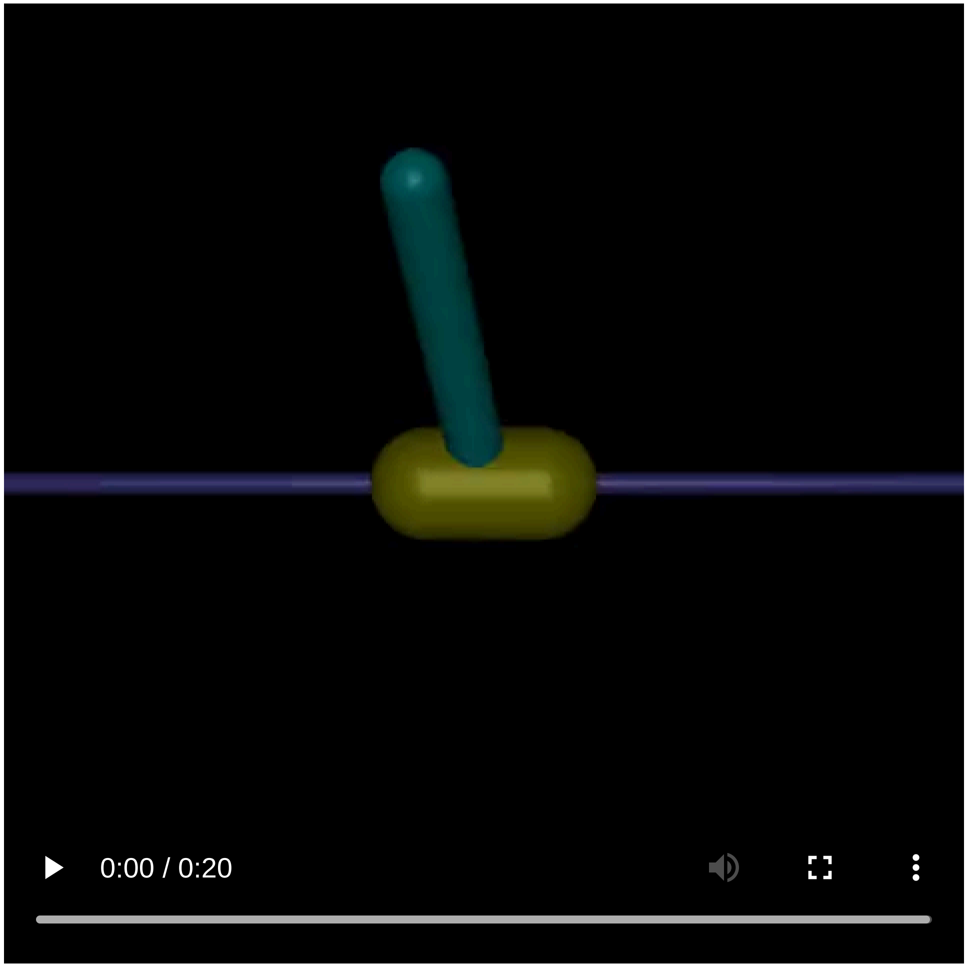
```
# run this for whatever highest milestone you reach
writer = SummaryWriter(log_dir=f'runs/SAC')

drl = DRL()
sac = SAC(n_obs=4, n_actions=1)

# takes ~5-10 minutes on colab gpus
for i in range(512):

    drl.rollout(sac, i)
    sac.update(drl.replay_buffer, i)
```

```
MoviePy - Building video /home/jblevins32/DRL2/rl-video-episode-0.mp4.
MoviePy - Writing video /home/jblevins32/DRL2/rl-video-episode-0.mp4
```

```
MoviePy - Done !
MoviePy - video ready /home/jblevins32/DRL2/rl-video-episode-0.mp4
```

In [ ]:

```
visualize(sac)
```


0:00 / 0:20