Ex8 Q1: Deep Q-networks

```
!pip install gym==0.15.7 --upgrade

import copy
import math
import os
from collections import namedtuple

import gym
import ipywidgets as widgets
import matplotlib.pyplot as plt
import more_itertools as mitt
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import tqdm
import random

plt.style.use('ggplot')
plt.rcParams['figure.figsize'] = [14, 6]
```

Environments

In this notebook, we will implement DQN and run it on four environments which have a continuous state-space and discrete action-space.

- Cart Pole: Balance a pole on a moving cart (https://gymnasium.farama.org/environments/classic_control/cart_pole/)
- Mountain Car: Gather momentum to climb a hill (https://gymnasium.farama.org/environments/classic_control/mountain_car/)
- Acrobot: Swing a two-link robot and reach the area above a line (https://gymnasium.farama.org/environments/classic_control/acrobot/)
- Lunar Lander: Fly and land a spaceship in the landing spot (https://gymnasium.farama.org/environments/box2d/lunar_lander/)

Note: If you are having trouble loading Lunar Lander due to Box2D/SWIG issues, feel free to comment out the environment for now. It is possible to complete the assignment without using this environment, but it is a particularly fun domain if you can get it working.

```
# Envs for training (no rendering)
envs = {
    'cartpole': gym.make('CartPole-v1'),
    'mountaincar': gym.make('MountainCar-v0'),
    'acrobot': gym.make('Acrobot-v1'),
    #'lunarlander': gym.make('LunarLander-v2'), #not implemented in
```

```
this assignment due to Box error
}
```

These environments are particularly nice because they all include a graphical visualization which we can use to visualize our learned policies. (V. Mnih: "Visualize everything you can think of.") Run the following cell and click the buttons to run the visualization with a random policy.

Note: You may need to restart the kernel after rendering one episode. Also, the random policy often results in very fast termination in Cart Pole and Lunar Lander, so you may not be able to see the visualization.

```
def render(env, policy=None):
    """Graphically render an episode using the given policy

    :param env: Gymnasium environment
    :param policy: Function which maps state to action.  If None, the
random
                   policy is used.
    """

    if policy is None:
        # Random policy
        def policy(state):
            return env.action_space.sample()

    # Basic gym loop
    state = env.reset()
    env.render()
    while True:
        action = policy(state)
        next_state, reward, terminated, truncated = env.step(action)
        env.render()

        if terminated:
            break
        #state = next_state
    env.close()

def button_callback(button):
    for b in buttons:
        b.disabled = True

    env = envs[button.description]
    render(env)
    env.close()

    for b in buttons:
        b.disabled = False

buttons = []
```

```python
for env_id in envs.keys():
    button = widgets.Button(description=env_id)
    button.on_click(button_callback)
    buttons.append(button)

print('Click a button to run a random policy:')
widgets.HBox(buttons)
```

```
Click a button to run a random policy:
```

{"model_id":"7a98642f3c5a41869fc3248b86205052","version_major":2,"version_minor":0}

Part (a): Exponential ε-greedy decay

Instead of keeping the exploration rate (ε) constant, it's typical to gradually decrease it over time following a specific schedule. This approach ensures that actions are predominantly exploratory at the beginning. While DQN utilizes a linear decay schedule for this purpose, in our case, we'll employ an exponential decay schedule:

$$\varepsilon_t = a \exp(b t)$$

Your task involves configuring the parameters (a) and (b) for a scheduler. This scheduler is designed to adjust (\epsilon) (epsilon) from an initial value to a final value over a specified number of timesteps. Once reaching the specified timesteps, (\epsilon) will remain at a minimal constant value to ensure ongoing exploration.

You are required to determine the values of (a) and (b) that will enable this scheduler to transition (\epsilon) as described, given the initial value, the final value, and the number of steps for this transition.

```python
class ExponentialSchedule:
    def __init__(self, value_from, value_to, num_steps):
        """Exponential schedule from `value_from` to `value_to` in
`num_steps` steps.

        $value(t) = a \exp (b t)$

        :param value_from: Initial value
        :param value_to: Final value
        :param num_steps: Number of steps for the exponential schedule
        """
        self.value_from = value_from
        self.value_to = value_to
        self.num_steps = num_steps

        # YOUR CODE HERE: Determine the `a` and `b` parameters such
that the schedule is correct
        self.a = self.value_from
        self.b =
```

```
np.log(self.value_to/self.value_from)/(self.num_steps-1)

    def value(self, step) -> float:
        """Return exponentially interpolated value between
`value_from` and `value_to`interpolated value between.

        Returns {
            `value_from`, if step == 0 or less
            `value_to`, if step == num_steps - 1 or more
            the exponential interpolation between `value_from` and
`value_to`, if 0 <= steps < num_steps
        }

        :param step: The step at which to compute the interpolation
        :rtype: Float. The interpolated value
        """

        # YOUR CODE HERE: Implement the schedule rule as described in
the docstring,
        # using attributes `self.a` and `self.b`.
        if step <=0:
            value = self.value_from
        elif step >= self.num_steps-1:
            value = self.value_to
        else:
            value = self.a * np.exp(self.b * step)

        return value


# DO NOT EDIT: Test code

def _test_schedule(schedule, step, value, ndigits=5):
    """Tests that the schedule returns the correct value."""
    v = schedule.value(step)
    if not round(v, ndigits) == round(value, ndigits):
        raise Exception(
            f'For step {step}, the scheduler returned {v} instead of
{value}'
        )

_schedule = ExponentialSchedule(0.1, 0.2, 3)
_test_schedule(_schedule, -1, 0.1)
_test_schedule(_schedule, 0, 0.1)
_test_schedule(_schedule, 1, 0.141421356237309515)
_test_schedule(_schedule, 2, 0.2)
_test_schedule(_schedule, 3, 0.2)
del _schedule

_schedule = ExponentialSchedule(0.5, 0.1, 5)
```

```
_test_schedule(_schedule, -1, 0.5)
_test_schedule(_schedule, 0, 0.5)
_test_schedule(_schedule, 1, 0.33437015248821106)
_test_schedule(_schedule, 2, 0.22360679774997905)
_test_schedule(_schedule, 3, 0.14953487812212207)
_test_schedule(_schedule, 4, 0.1)
_test_schedule(_schedule, 5, 0.1)
del _schedule
```

```
<>:3: DeprecationWarning: invalid escape sequence '\e'
```

Part (b): Replay memory

Now we will implement the replay memory (also called the replay buffer), the data-structure where we store previous experiences so that we can re-sample and train on them.

```
# Batch namedtuple, i.e. a class which contains the given attributes
Batch = namedtuple(
    'Batch', ('states', 'actions', 'rewards', 'next_states', 'dones')
)


class ReplayMemory:
    def __init__(self, max_size, state_size):
        """Replay memory implemented as a circular buffer.

        Experiences will be removed in a FIFO manner after reaching
maximum
        buffer size.

        Args:
            - max_size: Maximum size of the buffer
            - state_size: Size of the state-space features for the
environment
        """
        self.max_size = max_size
        self.state_size = state_size

        # Preallocating all the required memory, for speed concerns
        self.states = torch.empty((max_size, state_size))
        self.actions = torch.empty((max_size, 1), dtype=torch.long)
        self.rewards = torch.empty((max_size, 1))
        self.next_states = torch.empty((max_size, state_size))
        self.dones = torch.empty((max_size, 1), dtype=torch.bool)

        # Pointer to the current location in the circular buffer
        self.idx = 0
        # Indicates number of transitions currently stored in the
buffer
```

```
        self.size = 0

    def add(self, state, action, reward, next_state, done):
        """Add a transition to the buffer.

        :param state: 1-D np.ndarray of state-features
        :param action: Integer action
        :param reward: Float reward
        :param next_state: 1-D np.ndarray of state-features
        :param done: Boolean value indicating the end of an episode
        """

        # YOUR CODE HERE: Store the input values into the appropriate
        # attributes, using the current buffer position `self.idx`
        self.states[self.idx] = torch.Tensor(state)
        self.actions[self.idx] = torch.Tensor([action])
        self.rewards[self.idx] = torch.Tensor([reward])
        self.next_states[self.idx] = torch.Tensor(next_state)
        self.dones[self.idx] = torch.Tensor([done])



        # DO NOT EDIT
        # Circulate the pointer to the next position
        self.idx = (self.idx + 1) % self.max_size
        # Update the current buffer size
        self.size = min(self.size + 1, self.max_size)

    def sample(self, batch_size) -> Batch:
        """Sample a batch of experiences.

        If the buffer contains less that `batch_size` transitions,
sample all
        of them.

        :param batch_size: Number of transitions to sample
        :rtype: Batch
        """

        # YOUR CODE HERE: Randomly sample an appropriate number of
        # transitions *without replacement*. If the buffer contains
less than
        # `batch_size` transitions, return all of them. The return
type must
        # be a `Batch`.
        if self.size<batch_size:
            batch=Batch(states=self.states, actions=self.actions,
rewards=self.rewards, next_states=self.next_states,dones=self.dones)
        else:
            sample_indices =
```

```
np.random.choice(self.size,batch_size,replace=False)
            sampled_states = self.states[sample_indices]
            sampled_actions = self.actions[sample_indices]
            sampled_rewards = self.rewards[sample_indices]
            sampled_next_states = self.next_states[sample_indices]
            sampled_dones = self.dones[sample_indices]
            batch = Batch(states=sampled_states,
actions=sampled_actions,
                          rewards=sampled_rewards,
next_states=sampled_next_states,
                          dones=sampled_dones)

        return batch

    def populate(self, env, num_steps):
        """Populate this replay memory with `num_steps` from the
random policy.

        :param env: Gymnasium environment
        :param num_steps: Number of steps to populate the replay
memory
        """

        # YOUR CODE HERE: Run a random policy for `num_steps` time-
steps and
        # populate the replay memory with the resulting transitions.
        # Hint: Use the self.add() method.

        for _ in range(num_steps):
            state = env.reset()

            while True:
                action = env.action_space.sample()
                new_state, reward, done, _ = env.step(action)


self.add(state=state,action=action,next_state=new_state,done=done,rewa
rd=reward)
                state = new_state

                if done:
                  break
```

Part (c): Q-network

In this section, we define the object that DQN learns -- the Q-value neural network.

We use the PyTorch framework to define this neural network. PyTorch is a numeric computation library akin to NumPy, which also features automatic differentiation. This means that the library automatically computes the gradients for many differentiable operations, something we will exploit to train our models without having to manually program the gradients' code. *Caveats: Sometimes we have to pay explicit attention to whether the operations we are using are implemented by the library (most are), and there are a number of operations which do not play well with automatic differentiation (most notably, in-place assignments).*

If you are unfamiliar with PyTorch, this will be a great opportunity to learn the basics. The official tutorials are a good start: https://pytorch.org/tutorials

Do not worry about learning the advanced details; the basics are enough. If you can understand the following MNIST code example and are able to run it yourself to train an MNIST digit classifier, you should know more than enough PyTorch to complete the assignment). https://github.com/pytorch/examples/blob/main/mnist/main.py

This library is a tool, and as many tools you will have to learn how to use it well. Sometimes not using it well means that your program will crash. Sometimes it means that your program will not crash but will not be computing the correct outputs. And sometimes it means that it will compute the correct things, but is less efficient than it could otherwise be. This library is very popular these days, and online resources abound, so take your time to learn the basics. If you are having problems, first try to debug it yourself, and also look up the errors you get online. You can also use Piazza and office hours to ask for help with problems.

In the next cell, we inherit from the base class `torch.nn.Module` to implement our Q-network, which takes state-vectors and returns the respective action-values. Recall that the Q-network outputs the Q-values of all actions in the given input state.

```
class DQN(nn.Module):
    def __init__(self, state_dim, action_dim, *, num_layers=3,
hidden_dim=256):
        """Deep Q-Network PyTorch model.

        Args:
            - state_dim: Dimensionality of states
            - action_dim: Dimensionality of actions
            - num_layers: Number of total linear layers
            - hidden_dim: Number of neurons in the hidden layers
        """

        super().__init__()
        self.state_dim = state_dim
        self.action_dim = action_dim
        self.num_layers = num_layers
        self.hidden_dim = hidden_dim

        # YOUR CODE HERE: Define the layers of your model such that
        # * there are `num_layers` nn.Linear modules / layers
        # * all activations except the last should be ReLU activations
        #   (this can be achieved either using a nn.ReLU() object or
the nn.functional.relu() method)
        # * the last activation can either be missing, or you can use
nn.Identity()
```

```
        # Hint: A regular Python list of layers is tempting, but 
PyTorch does not register
        # these parameters in its computation graph. See nn.ModuleList 
or nn.Sequential

        self.fc1 = nn.Linear(self.state_dim,self.hidden_dim)
        self.fc2 = nn.Linear(self.hidden_dim,self.hidden_dim)
        self.fc3 = nn.Linear(self.hidden_dim,self.action_dim)


    def forward(self, states) -> torch.Tensor:
        """Q function mapping from states to action-values.

        :param states: (*, S) torch.Tensor where * is any number of 
additional
                dimensions, and S is the dimensionality of state-space
        :rtype: (*, A) torch.Tensor where * is the same number of 
additional
                dimensions as the `states`, and A is the 
dimensionality of the
                action-space. This represents the Q values Q(s, .)
        """
        # YOUR CODE HERE: Use the defined layers and activations to 
compute
        # the action-values tensor associated with the input states.
        # Hint: Do not worry about the * arguments above (previous 
dims in tensor).
        # PyTorch functions typically handle those properly.
        x = F.relu(self.fc1(states))
        x = F.relu(self.fc2(x))
        actions = self.fc3(x)

        return actions

    # DO NOT EDIT: Utility methods for cloning and storing models.

    @classmethod
    def custom_load(cls, data):
        model = cls(*data['args'], **data['kwargs'])
        model.load_state_dict(data['state_dict'])
        return model

    def custom_dump(self):
        return {
            'args': (self.state_dim, self.action_dim),
            'kwargs': {
                'num_layers': self.num_layers,
                'hidden_dim': self.hidden_dim,
            },
            'state_dict': self.state_dict(),
```

```
        }


# DO NOT EDIT: Test code

def _test_dqn_forward(dqn_model, input_shape, output_shape):
    """Tests that the dqn returns the correctly shaped tensors."""
    inputs = torch.torch.randn((input_shape))
    outputs = dqn_model(inputs)

    if not isinstance(outputs, torch.FloatTensor):
        raise Exception(
            f'DQN.forward returned type {type(outputs)} instead of
torch.Tensor'
        )

    if outputs.shape != output_shape:
        raise Exception(
            f'DQN.forward returned tensor with shape {outputs.shape}
instead of {output_shape}'
        )

    if not outputs.requires_grad:
        raise Exception(
            f'DQN.forward returned tensor which does not require a
gradient (but it should)'
        )

dqn_model = DQN(10, 4)
_test_dqn_forward(dqn_model, (64, 10), (64, 4))
_test_dqn_forward(dqn_model, (2, 3, 10), (2, 3, 4))
del dqn_model

dqn_model = DQN(64, 16)
_test_dqn_forward(dqn_model, (64, 64), (64, 16))
_test_dqn_forward(dqn_model, (2, 3, 64), (2, 3, 16))
del dqn_model

# Testing custom dump / load
dqn1 = DQN(10, 4, num_layers=10, hidden_dim=20)
dqn2 = DQN.custom_load(dqn1.custom_dump())
assert dqn2.state_dim == 10
assert dqn2.action_dim == 4
assert dqn2.num_layers == 10
assert dqn2.hidden_dim == 20
```

Part (d): Single-batch update

Recall that the Q-network in DQN is trained periodically using batches of experiences sampled from the replay memory. The following function computes the loss on this batch (one-step TD errors, using the Q-network and the target network) and uses the optimizer to perform one step of gradient descent using the gradient of this loss with respect to the Q-network parameters (automatically, thanks to PyTorch!).

```
def train_dqn_batch(optimizer, batch, dqn_model, dqn_target, gamma) ->
float:
    """Perform a single batch-update step on the given DQN model.

    :param optimizer: nn.optim.Optimizer instance
    :param batch: Batch of experiences (class defined earlier)
    :param dqn_model: The DQN model to be trained
    :param dqn_target: The target DQN model, ~NOT~ to be trained
    :param gamma: The discount factor
    :rtype: Float. The scalar loss associated with this batch
    """
    # YOUR CODE HERE: Compute the values and target_values tensors
using the
    # given models and the batch of data.
    # Recall that 'Batch' is a named tuple consisting of
    # ('states', 'actions', 'rewards', 'next_states', 'dones')
    # Hint: Remember that we should not pass gradients through the
target network

    values = dqn_model(batch.states).gather(1, batch.actions)
    max_value = torch.max(dqn_target(batch.next_states), dim=1)
[0].detach()

    for i in range(len(batch.dones)):
        if batch.dones[i]:
            max_value[i] = 0

    max_value = torch.unsqueeze(max_value, 1)
    target_values = batch.rewards + gamma * max_value

    # DO NOT EDIT

    assert (
        values.shape == target_values.shape
    ), 'Shapes of values tensor and target_values tensor do not
match.'

    # Testing that the values tensor requires a gradient,
    # and the target_values tensor does not
    assert values.requires_grad, 'values tensor requires gradients'
    assert (
        not target_values.requires_grad
    ), 'target_values tensor should not require gradients'

    # Computing the scalar MSE loss between computed values and the
TD-target
```

```
    # DQN originally used Huber loss, which is less sensitive to
outliers
    loss = F.mse_loss(values, target_values)

    optimizer.zero_grad() # Reset all previous gradients
    loss.backward() # Compute new gradients
    optimizer.step() # Perform one gradient-descent step

    return loss.item()
```

Part (e): DQN training loop

This is the main training loop for DQN. Please refer to Algorithm 1 in the DQN paper (reproduced in lecture slides).

```
def train_dqn(
    env,
    num_steps,
    *,
    num_saves=5,
    replay_size,
    replay_prepopulate_steps=0,
    batch_size,
    exploration,
    gamma,
):
    """
    DQN algorithm.

    Compared to previous training procedures, we will train for a
given number
    of time-steps rather than a given number of episodes. The number
of
    time-steps will be in the range of millions, which still results
in many
    episodes being executed.

    Args:
        - env: The Gymnasium environment
        - num_steps: Total number of steps to be used for training
        - num_saves: How many models to save to analyze the training
progress
        - replay_size: Maximum size of the ReplayMemory
        - replay_prepopulate_steps: Number of steps with which to
prepopulate
                                    the memory
        - batch_size: Number of experiences in a batch
        - exploration: An ExponentialSchedule
        - gamma: The discount factor
```

```
    Returns: (saved_models, returns)
        - saved_models: Dictionary whose values are trained DQN models
        - returns: Numpy array containing the return of each training
episode
        - lengths: Numpy array containing the length of each training
episode
        - losses: Numpy array containing the loss of each training
batch
    """
    # Check that environment states are compatible with our DQN
representation
    assert (
        isinstance(env.observation_space, gym.spaces.Box)
        and len(env.observation_space.shape) == 1
    )

    # Get the state_size from the environment
    state_size = env.observation_space.shape[0]

    # Initialize the DQN and DQN-target models
    dqn_model = DQN(state_size, env.action_space.n)
    dqn_target = DQN.custom_load(dqn_model.custom_dump())

    # Initialize the optimizer
    optimizer = torch.optim.Adam(dqn_model.parameters())

    # Initialize the replay memory and prepopulate it
    memory = ReplayMemory(replay_size, state_size)
    memory.populate(env, replay_prepopulate_steps)

    # Initialize lists to store returns, lengths, and losses
    rewards = []
    returns = []
    lengths = []
    losses = []

    # Initialize structures to store the models at different stages of
training
    t_saves = np.linspace(0, num_steps, num_saves - 1, endpoint=False)
    saved_models = {}

    i_episode = 0  # Use this to indicate the index of the current
episode
    t_episode = 0  # Use this to indicate the time-step inside current
episode

    state = env.reset()  # Initialize state of first episode

    # Iterate for a total of `num_steps` steps
```

```
    pbar = tqdm.trange(num_steps)
    for t_total in pbar:
        # Use t_total to indicate the time-step from the beginning of
training

        # Save model
        if t_total in t_saves:
            model_name = f'{100 * t_total /
num_steps:04.1f}'.replace('.', '_')
            saved_models[model_name] = copy.deepcopy(dqn_model)

        # YOUR CODE HERE:
        #   * sample an action from the DQN using epsilon-greedy
        #   * use the action to advance the environment by one step
        #   * store the transition into the replay memory

        epsilon = exploration.value(num_steps)
        if random.random()<=epsilon:
            action = env.action_space.sample()


        else:
            action =
int(torch.argmax(dqn_model(torch.FloatTensor(state))))


        new_state, reward, done, _ = env.step(action)
        memory.add(state, action, reward, new_state, done)
        rewards.append(reward)

        # YOUR CODE HERE: Once every 4 steps,
        #   * sample a batch from the replay memory
        #   * perform a batch update (use the train_dqn_batch() method)

        if t_total % 4 == 0:
            batch = memory.sample(batch_size)
            loss = train_dqn_batch(optimizer, batch, dqn_model,
dqn_target, gamma)
            losses.append(loss)


        # YOUR CODE HERE: Once every 10_000 steps,
        #   * update the target network (use the dqn_model.state_dict()
and
        #     dqn_target.load_state_dict() methods)

        if t_total % 10_000 == 0:
            dqn_target.load_state_dict(dqn_model.state_dict())

        if done:
```

```
            # YOUR CODE HERE: Anything you need to do at the end of an
episode,
            # e.g., compute return G, store returns/lengths,
            # reset variables, indices, lists, etc.

            lengths.append(len(rewards))

            G = 0
            for i in rewards:
                G = i + gamma*G

            returns.append(G)
            rewards = []
            t_episode  =  0
            state = env.reset()
            i_episode += 1

            pbar.set_description(
                f'Episode: {i_episode} | Steps: {t_episode + 1} |
Return: {G:5.2f} | Epsilon: {epsilon:4.2f}'
            )

        else:
            # YOUR CODE HERE: Anything you need to do within an
episode
            state = new_state
            t_episode += 1

    saved_models['100_0'] = copy.deepcopy(dqn_model)

    return (
        saved_models,
        np.array(returns),
        np.array(lengths),
        np.array(losses),
    )
```

Part (f): Evaluation of DQN on the 4 environments

In the following section, run DQN on some/all of the 4 environments loaded in the beginning of this notebook (Cart Pole, Mountain Car, Acrobot, Lunar Lander).

Each trial (in each environment) is trained for 1.5 million steps, which takes substantially longer than previous assignments, estimated to be around 1-2 hours on a typical desktop/laptop CPU. Because of this, we only expect you to train for one trial in each environment. Additionally, it is fine to only evaluate a minimum of 2 out of the 4 environments, although we recommend trying all 4 (and/or multiple trials) if you are able to.

Since there is only one trial, we cannot obtain meaningful averages / confidence bands as in past assignments. Instead, we will just apply a moving average to smooth out the data in our graphs (you should plot both the raw data and the moving average).

Obviously, during development and debugging, you should set the number of training steps (and possibly other hyperparameters) to be much lower. However, please remember to restore the original values when you perform the final evaluation. Also, different environments train at different speeds -- be cognizant of this when choosing an environment to develop in (e.g., Mountain car is designed as a hard exploration problem). We recommending starting in Cart Pole because it is an easier problem and the environment runs faster. For reference, we can run this environment at ~650 steps/s = 40K steps/min, and it usually takes 2000-4000 episodes = ~100K steps (with the default exponential schedule) to see some initial signs of learning.

```
def moving_average(data, *, window_size = 50):
    """Smooths 1-D data array using a moving average.

    Args:
        data: 1-D numpy.array
        window_size: Size of the smoothing window

    Returns:
        smooth_data: A 1-d numpy.array with the same size as data
    """
    assert data.ndim == 1
    kernel = np.ones(window_size)
    smooth_data = np.convolve(data, kernel) / np.convolve(
        np.ones_like(data), kernel
    )
    return smooth_data[: -window_size + 1]
```

Cart Pole

Test your implentation on the Cart Pole environment. Training will take much longer than in the previous homeworks, so this time you will not have to find good hyperparameters or train multiple runs. This cell should take about 1-2 hours to run. After training, run the last cell in this notebook to view the policies which were obtained at 0%, 25%, 50%, 75% and 100% of the training.

```
env = envs['cartpole']
gamma = 0.99

# We train for many time-steps; as usual, you can decrease this during
development / debugging,
# but make sure to restore it to 1_500_000 before submitting
num_steps = 1_500_000
num_saves = 5  # Save models at 0%, 25%, 50%, 75% and 100% of training

replay_size = 200_000
replay_prepopulate_steps = 50_000
```

```
batch_size = 64
exploration = ExponentialSchedule(1.0, 0.05, 1_000_000)

# This should take about 1-2 hours on a generic 4-core laptop
dqn_models, returns, lengths, losses = train_dqn(
    env,
    num_steps,
    num_saves=num_saves,
    replay_size=replay_size,
    replay_prepopulate_steps=replay_prepopulate_steps,
    batch_size=batch_size,
    exploration=exploration,
    gamma=gamma,
)

assert len(dqn_models) == num_saves
assert all(isinstance(value, DQN) for value in dqn_models.values())

# Saving computed models to disk, so that we can load and visualize
them later
checkpoint = {key: dqn.custom_dump() for key, dqn in
dqn_models.items()}
torch.save(checkpoint, f'checkpoint_{env.spec.id}.pt')
```

```
Episode: 10972 | Steps: 1 | Return: 99.34 | Epsilon: 0.05: 100%|
██████████| 1500000/1500000 [1:38:16<00:00, 254.39it/s]
```

Plot the returns, lengths, and losses obtained while running DQN on the Cart Pole environment.

Again, plot both the raw data and the moving average in the same plot, i.e., you should have 3 plots total. Think about what the appropriate horizontal axis should be.

```
# YOUR PLOTTING CODE HERE

plt.figure(1)
plt.xlabel('Episodes')
plt.ylabel('Return')
plt.title('CartPole')
plt.plot(returns, color = 'orchid', label = 'Returns')
plt.plot(moving_average(returns, window_size =
int(len(returns)/200)),'r', label = 'Average')
plt.legend()

plt.figure(2)
plt.xlabel('Episodes')
plt.ylabel('Length')
plt.title('CartPole')
plt.plot(lengths, 'lightblue', label = 'Lengths')
```

```
plt.plot(moving_average(lengths, window_size =
int(len(lengths)/200)),'r', label = 'Average')
plt.legend()

plt.figure(3)
plt.xlabel('Episodes')
plt.ylabel('Losses')
plt.title('CartPole')
plt.plot(losses, 'chartreuse', label = 'Losses')
plt.plot(moving_average(losses, window_size =
int(len(losses)/200)),'r', label = 'Average')
plt.legend()

plt.show()
```

Mountain Car

Test your implentation on the Mountain Car environment. Training will take much longer than in the previous homeworks, so this time you will not have to find good hyperparameters or train multiple runs. This cell should take about 1-2 hours to run. After training, run the last cell in this notebook to view the policies which were obtained at 0%, 25%, 50%, 75% and 100% of the training

```
env = envs['mountaincar']
gamma = 0.99

# We train for many time-steps; as usual, you can decrease this during
development / debugging,
# but make sure to restore it to 1_500_000 before submitting
num_steps = 1_500_000
num_saves = 5  # Save models at 0%, 25%, 50%, 75% and 100% of training

replay_size = 200_000
replay_prepopulate_steps = 50_000

batch_size = 64
exploration = ExponentialSchedule(1.0, 0.05, 1_000_000)

# This should take about 1-2 hours on a generic 4-core laptop
dqn_models, returns, lengths, losses = train_dqn(
    env,
    num_steps,
    num_saves=num_saves,
    replay_size=replay_size,
    replay_prepopulate_steps=replay_prepopulate_steps,
    batch_size=batch_size,
    exploration=exploration,
    gamma=gamma,
)
```

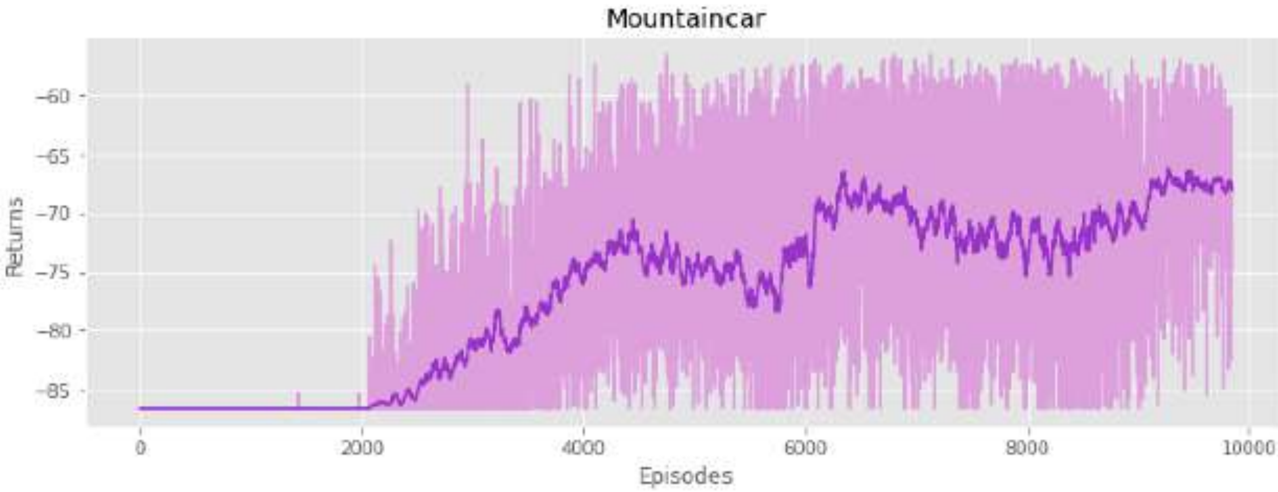
```
assert len(dqn_models) == num_saves
assert all(isinstance(value, DQN) for value in dqn_models.values())

# Saving computed models to disk, so that we can load and visualize
them later
checkpoint = {key: dqn.custom_dump() for key, dqn in
dqn_models.items()}
torch.save(checkpoint, f'checkpoint_{env.spec.id}.pt')
```

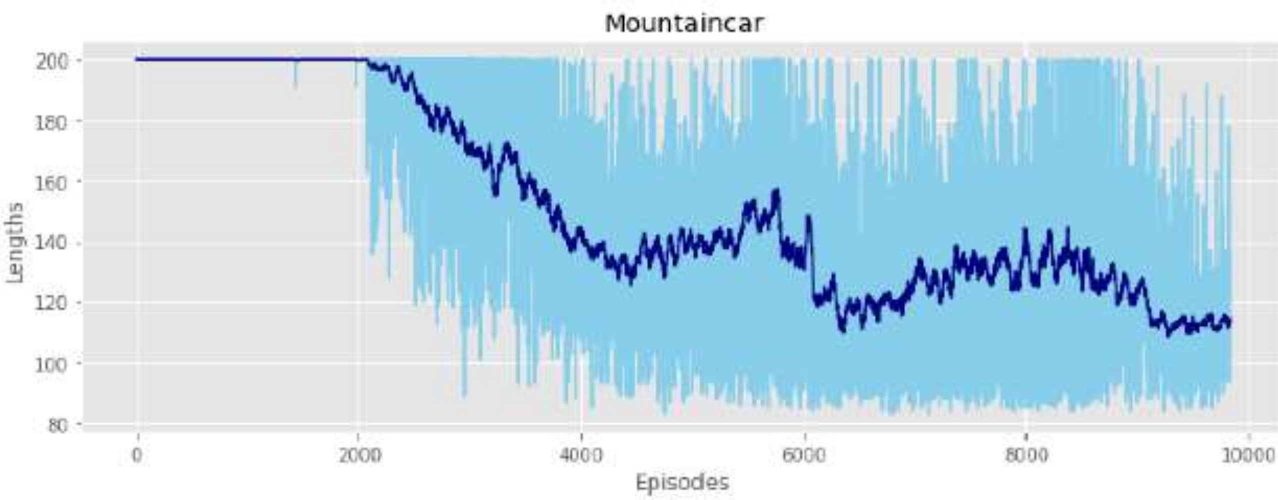
```
Episode: 10513 | Steps: 1 | Return: -67.56 | Epsilon: 0.05: 100%|
██████████| 1500000/1500000 [1:35:27<00:00, 261.92it/s]
```

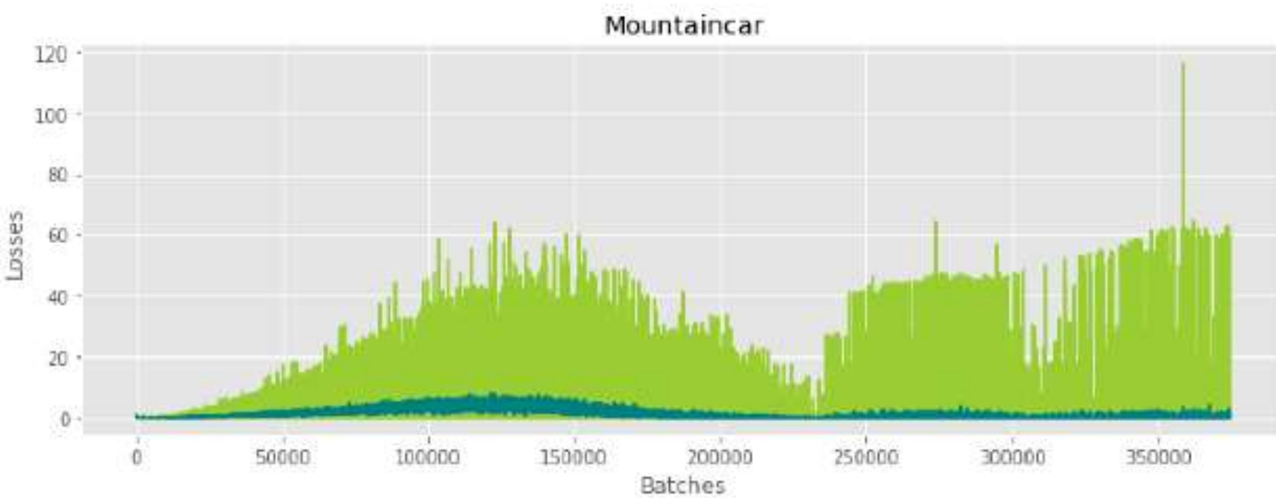
Plot the returns, lengths, and losses obtained while running DQN on the Mountain Car environment.

Again, plot both the raw data and the moving average in the same plot, i.e., you should have 3 plots total. Think about what the appropriate horizontal axis should be.

```
# YOUR PLOTTING CODE HERE

plt.figure(1)
plt.xlabel('Episodes')
plt.ylabel('Return')
plt.title('Mountaincar')
plt.plot(returns, color = 'orchid', label = 'Returns')
plt.plot(moving_average(returns, window_size =
int(len(returns)/200)),'r', label = 'Average')
plt.legend()

plt.figure(2)
plt.xlabel('Episodes')
plt.ylabel('Length')
plt.title('Mountaincar')
plt.plot(lengths, 'lightblue', label = 'Lengths')
plt.plot(moving_average(lengths, window_size =
int(len(lengths)/200)),'r', label = 'Average')
plt.legend()

plt.figure(3)
plt.xlabel('Episodes')
plt.ylabel('Losses')
plt.title('Mountaincar')
plt.plot(losses, 'plt.plot(losses, 'chartreuse', label = 'Losses')
plt.plot(moving_average(losses, window_size =
int(len(losses)/200)),'r', label = 'Average')
plt.legend()

plt.show()
```







Acrobot

Test your implentation on the Acrobot environment. Training will take much longer than in the previous homeworks, so this time you will not have to find good hyperparameters or train multiple runs. This cell should take about 1-2 hours to run. After training, run the last cell in this notebook to view the policies which were obtained at 0%, 25%, 50%, 75% and 100% of the training

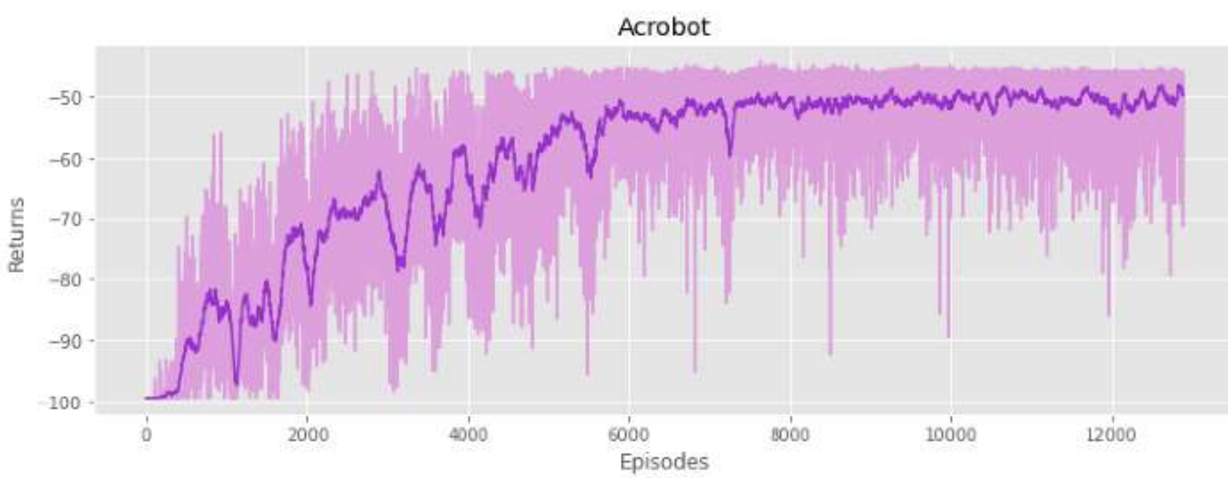
```python
env = envs['acrobot']
gamma = 0.99

# We train for many time-steps; as usual, you can decrease this during development / debugging,
# but make sure to restore it to 1_500_000 before submitting
num_steps = 1_500_000
num_saves = 5  # save models at 0%, 25%, 50%, 75% and 100% of training

replay_size = 200_000
replay_prepopulate_steps = 50_000

batch_size = 64
exploration = ExponentialSchedule(1.0, 0.05, 1_000_000)

# This should take about 1-2 hours on a generic 4-core laptop
dqn_models, returns, lengths, losses = train_dqn(
    env,
    num_steps,
    num_saves=num_saves,
    replay_size=replay_size,
    replay_prepopulate_steps=replay_prepopulate_steps,
    batch_size=batch_size,
    exploration=exploration,
    gamma=gamma,
)

assert len(dqn_models) == num_saves
assert all(isinstance(value, DQN) for value in dqn_models.values())

# Saving computed models to disk, so that we can load and visualize them later
checkpoint = {key: dqn.custom_dump() for key, dqn in dqn_models.items()}
torch.save(checkpoint, f'checkpoint_{env.spec.id}.pt')
```

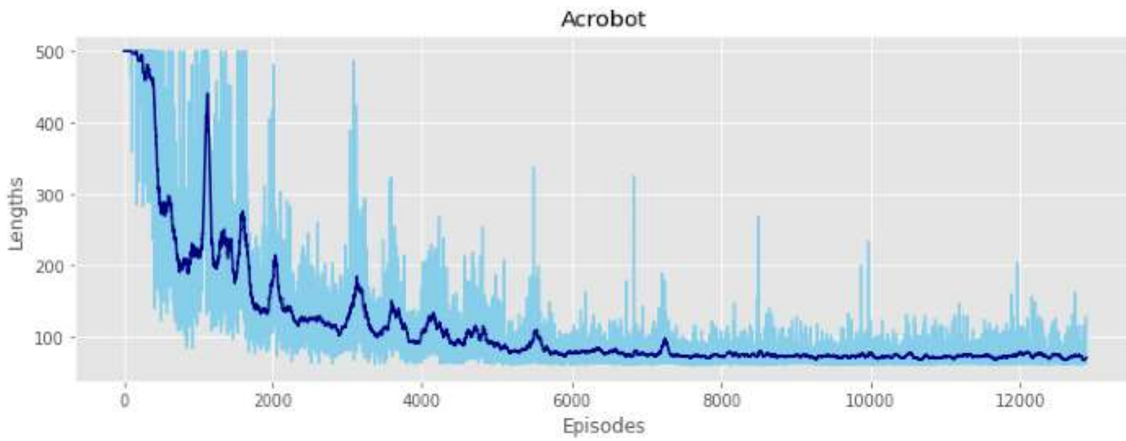
```
Episode: 12985 | Steps: 1 | Return: -49.02 | Epsilon: 0.05: 100%|
██████████| 1500000/1500000 [1:53:06<00:00, 221.02it/s]
```

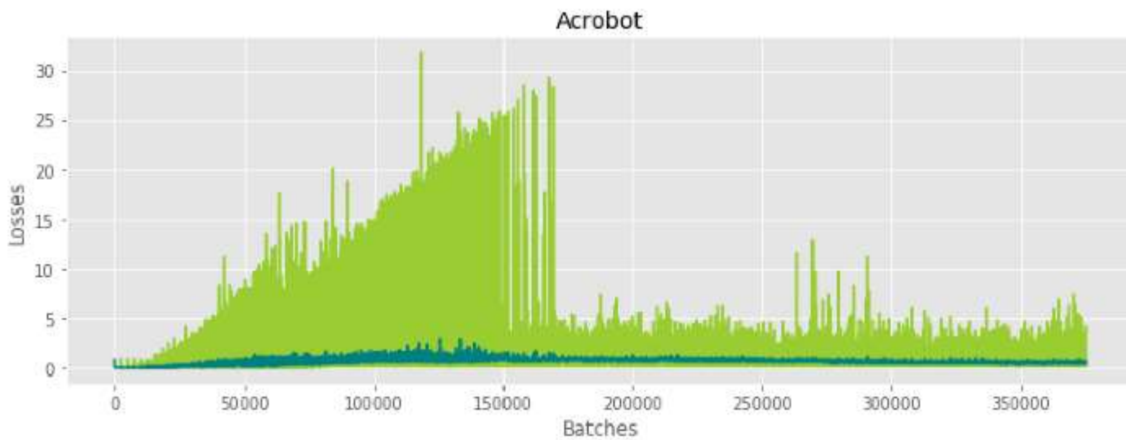
Plot the returns, lengths, and losses obtained while running DQN on the Acrobot environment.

Again, plot both the raw data and the moving average in the same plot, i.e., you should have 3 plots total. Think about what the appropriate horizontal axis should be.

```python
# YOUR PLOTTING CODE HERE

plt.figure(1)
plt.xlabel('Episodes')
plt.ylabel('Return')
plt.title('Acrobot')
plt.plot(returns, color = 'orchid', label = 'Returns')
plt.plot(moving_average(returns, window_size =
int(len(returns)/200)),'r', label = 'Average')
plt.legend()

plt.figure(2)
plt.xlabel('Episodes')
plt.ylabel('Length')
plt.title('Acrobot')
plt.plot(lengths, 'lightblue', label = 'Lengths')
plt.plot(moving_average(lengths, window_size =
int(len(lengths)/200)),'r', label = 'Average')
plt.legend()

plt.figure(3)
plt.xlabel('Episodes')
plt.ylabel('Losses')
plt.title('Acrobot')
plt.plot(losses, 'chartreuse', label = 'Losses')
plt.plot(moving_average(losses, window_size =
int(len(losses)/200)),'r', label = 'Average')
plt.legend()

plt.show()
```







Lunar Lander

Test your implentation on the Lunar Lander environment. Training will take much longer than in the previous homeworks, so this time you will not have to find good hyperparameters or train multiple runs. This cell should take about 1-2 hours to run. After training, run the last cell in this notebook to view the policies which were obtained at 0%, 25%, 50%, 75% and 100% of the training

```
env = envs['lunarlander']
gamma = 0.99

# We train for many time-steps; as usual, you can decrease this during
development / debugging,
# but make sure to restore it to 1_500_000 before submitting
num_steps = 1_500_000
num_saves = 5  # save models at 0%, 25%, 50%, 75% and 100% of training

replay_size = 200_000
replay_prepopulate_steps = 50_000
```

```
batch_size = 64
exploration = ExponentialSchedule(1.0, 0.05, 1_000_000)

# This should take about 1-2 hours on a generic 4-core laptop
dqn_models, returns, lengths, losses = train_dqn(
    env,
    num_steps,
    num_saves=num_saves,
    replay_size=replay_size,
    replay_prepopulate_steps=replay_prepopulate_steps,
    batch_size=batch_size,
    exploration=exploration,
    gamma=gamma,
)

assert len(dqn_models) == num_saves
assert all(isinstance(value, DQN) for value in dqn_models.values())

# Saving computed models to disk, so that we can load and visualize
them later
checkpoint = {key: dqn.custom_dump() for key, dqn in
dqn_models.items()}
torch.save(checkpoint, f'checkpoint_{env.spec.id}.pt')
```

Plot the returns, lengths, and losses obtained while running DQN on the Lunar Lander environment.

Again, plot both the raw data and the moving average in the same plot, i.e., you should have 3 plots total. Think about what the appropriate horizontal axis should be.

```
# YOUR PLOTTING CODE HERE # this has not been implemented here, but
can be done after rectifying the box error
```

Visualization of the trained policies

Run the cell below and push the buttons to view the progress of the policy trained using DQN.

```
buttons_all = []
for key_env, env in envs.items():
    try:
        checkpoint = torch.load(f'checkpoint_{env.spec.id}.pt')
    except FileNotFoundError:
        pass
    else:
        buttons = []
        for key, value in checkpoint.items():
            dqn = DQN.custom_load(value)
```

```
            def make_callback(env, dqn):
                def button_callback(button):
                    for b in buttons_all:
                        b.disabled = True

                    render(env, lambda state: dqn(torch.tensor(state,
dtype=torch.float)).argmax().item())

                    for b in buttons_all:
                        b.disabled = False

                return button_callback

            button = widgets.Button(description=f'{key.replace("_",
".")}%')
            button.on_click(make_callback(env, dqn))
            buttons.append(button)

        print(f'{key_env}:')
        display(widgets.HBox(buttons))
        buttons_all.extend(buttons)
```

Analysis

For each environment that you trained in, describe the progress of the training in terms of the behavior of the agent at each of the 5 phases of training (i.e. 0%, 25%, 50%, 75%, 100%). Make sure you view each phase a few times so that you can see all sorts of variations.

Describe something for each phase. Start by describing the behavior at phase 0%, then, for each next phase, describe how it differs from the previous one, how it improves and/or how it becomes worse. At the final phase (100%), also describe the observed behavior in absolute terms, and whether it has achieved optimality.

Note: You may need to restart the kernel after rendering some episodes. Do not manually close the Pygame window. Even if you restart the kernel, you do not need to re-train on the environments; the relevant Q-network parameters should be stored in the corresponding PyTorch checkpoint .pt file.

Cart Pole

- 0%) The window flashes by and can only see the agent move randomly without any learning process. .
- 25%) The agent starts moving to the left and right trying to keep the pole from falling down, but soon fails. .
- 50%) This agent is still learning how to keep the pole from falling down. It moves a little faster, but not much of a boost. .
- 75%) The agent moved too much at the beginning, but it soon learned how to maintain balance and stayed at the edge of the window for a while without falling down .

- 100%) Certainly an improvement over the last phase. Compared to the previous phase, the agent mastered its balance more quickly after the initial random movement and remained more stable, not swaying from side to side like the last phase. .

Mountain Car

- 0%) The car just stays in the vicinity of the starting point. It is not able to make much progress towards the goal. It just oscillates. Basically, the agent just swayed slowly and randomly at the bottom of the mountain. .
- 25%) The agent moved a longer distance, but it looks like it's still learning how to reach the top of the mountain. .
- 50%) This time the agent moved faster and further and was able to reach the top of the mountain in two or three moves back and forth. .
- 75%) The agent is now able to learn to summit in fewer round trips than in the last phase. .
- 100%) The number of round trips before the agent summit did not change significantly than the last phase, but the speed is faster. This would seem to be optimal .

Acrobot

- 0%) The agent just shakes slightly at random. .
- 25%) The agent shakes a little more, but not enough to reach the goal level. .
- 50%) The agent is increasing its speed and amplitude of shaking and can reach the goal height after some time. .
- 75%) The agent can reach the goal height faster with a larger swing. .
- 100%) In this last phase, the agent's learning speed and the speed of reaching the goal height are the fastest and most stable. .

Lunar Lander (not performed here)

- 0%) YOUR ANSWER HERE.
- 25%) YOUR ANSWER HERE.
- 50%) YOUR ANSWER HERE.
- 75%) YOUR ANSWER HERE.
- 100%) YOUR ANSWER HERE.

[Extra credit.] Part (g): DQN extensions

We briefly gave an overview of several extensions to DQN, described in the "Rainbow" paper: Rainbow: Combining improvements in deep reinforcement learning Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, David Silver AAAI Conference on Artificial Intelligence, 2018 https://aaai.org/papers/11796-rainbow-combining-improvements-in-deep-reinforcement-learning/

Read about these extensions in the Rainbow paper and their corresponding source paper(s). Implement one or more of these extensions from scratch and evaluate them on at least 2 of the environments above. Compare against the original DQN and discuss your findings.