In [1]:
```
%%javascript
IPython.OutputArea.prototype._should_scroll = function(lines) {
    return false;
}
```

Before submitting, make sure you are adhering to the following rules, which helps us grade your assignment. Assignments that do not adhere to these rules will be penalized.

- Make sure your notebook only contains the exercises requested in the notebook, and the written homework (if any) is delivered in class in printed form, i.e. don't submit your written homework as part of the notebook.
- Make sure you are using Python3. This notebook is already set up to use Python3 (top right corner); Do not change this.
- If a method is provided with a specific signature, do not change the signature in any way, or the default values.
- Don't hard-code your solutions to the specific environments which it is being used on, or the specific hyper-parameters which it is being used on; Be as general as possible, which means also using ALL the arguments of the methods your are implementing.
- Clean up your code before submitting, i.e. remove all print statements that you've used to develop and debug (especially if it's going to clog up the interface by printing thousands of lines). Only output whatever is required by the exercise.
- For technical reasons, plots should be contained in their own cell which should run instantly, separate from cells which perform longer computations. This notebook is already formatted in such a way, please make sure this remains the case.
- Make sure your notebook runs completely, from start to end, without raising any unintended errors. After you've made the last edit, Use the option `Kernel -> Restart & Run All` to rerun the entire notebook. If you end up making ANY edit, re-run everything again. Always assume any edit you make may have broken your code!

Homework 6: Deep Q-Networks in Pytorch

In this assignment you will implement deep q-learning using Pytorch.

In [2]:
```
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
```

```
plt.style.use('ggplot')
plt.rcParams['figure.figsize'] = [12, 4]
```

Environments

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

- CartPole: Balance a pole on a moving cart (https://gym.openai.com/envs/CartPole-v1/).
- Mountain Car: Gather momentum to climb a hill (https://gym.openai.com/envs/MountainCar-v0/).
- AcroBot: A two-link robot needs to swing and reach the area above a line (https://gym.openai.com/envs/Acrobot-v1/).
- LunarLander: A spaceship needs to fly and land in the landing spot. (https://gym.openai.com/envs/LunarLander-v2/).

In [3]:

```
envs = {
    'cartpole': gym.make('CartPole-v1'),
    'mountaincar': gym.make('MountainCar-v0'),
    'acrobot': gym.make('Acrobot-v1'),
    'lunarlander': gym.make('LunarLander-v2'),
}
```

These environments are particularly cool because they all include a graphical visualization which we can use to visualize our learned policies. Run the folling cell and click the buttons to run the visualization with a random policy.

In [4]:

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

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

    if policy is None:

        def policy(state):
            return env.action_space.sample()

    state = env.reset()
    env.render()

    while True:
        action = policy(state)
        state, _, done, _ = env.step(action)
        env.render()

        if done:
            break

    env.close()
```

In [5]:

```
#  Jupyter UI
```

```
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 = []
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:
```

Misc Utilities

Some are provided, some you should implement

Smoothing

In this homework, we'll do some plotting of noisy data, so here is the smoothing function which was also used in the previous homework.

In [6]:

```
def rolling_average(data, *, window_size):
    """Smoothen the 1-d data array using a rollin 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]
```

Q1 (1 pt): Exponential ϵ-Greedy Decay

This time we'll switch from using a linear decay to an exponential decay, defined as

$$\epsilon_t = a \exp(bt)$$

where a and b are the parameters of the schedule.

The interface to the scheduler is the same as in the linear case from the previous homework, i.e. it receives the initial value, the final value, and in how many steps to go from initial to final. Your task is to compute parameters `a` and `b` to make the scheduler work as expected.

In [7]:

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

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

        :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 sche
        # exp(tb) = exp(0b) = 1
        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

        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
        }

        :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 docstri
        # 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)

        #print(value)

        return value


# test code, do not edit


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
```

Q2 (1 pt): Replay Memory

Now we will implement the Replay Memory, the data-structure where we store previous experiences so that we can re-sample and train on them.

In [8]:

```
# 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`.

        sample_indices = min(self.size, batch_size)
        transitions = np.random.choice(self.size, sample_indices, replace=False)

        batch = Batch(self.states[transitions], self.actions[transitions], self.re
                      self.next_states[transitions], self.dones[transitions])

        return batch

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

        :param env:  Openai Gym environment
        :param num_steps:  Number of steps to populate the
        """

        # YOUR CODE HERE:  run a random policy for `num_steps` time-steps and
        # populate the replay memory with the resulting transitions.
        # Hint:  don't repeat code!  Use the self.add() method!
```

```
        state = env.reset()

        i = 0
        while i < num_steps:
            action = env.action_space.sample()
            next_state, reward, done, _ = env.step(action)

            self.add(state, action, reward, next_state, done)

            if done:
                state = env.reset()
            else:
                state = next_state

            i += 1
```

Q3 (2 pts): Pytorch DQN module

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 program the gradients' code. There are a few 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 don't play well with automatic differentiation (most notably, in-place assignments).

This library is a tool, and as many tools you'll 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 won't crash but won't 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 SUPER popular, and online resources abound, so take your time to learn the basics. If you're having problems, first try to debug it yourself, also looking up the errors you get online. You can also use Piazza and the office hours to ask for help with problems.

In the next cell, we inherit from the base class `torch.nn.Module` to implement our DQN module, which takes state-vectors and returns the respective action-values.

In [9]:

```
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.functi
        # * the last activation can either be missing, or you can use nn.Identity(

        self.net = [nn.Linear(state_dim, hidden_dim),
                    nn.ReLU(True)]
        for _ in range(num_layers - 2):
            self.net += [nn.Linear(hidden_dim, hidden_dim),
                         nn.ReLU(True)]
        self.net += [nn.Linear(hidden_dim, action_dim)]
        self.models = nn.Sequential(*self.net)

    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.

        return self.models(states)

    # utility methods for cloning and storing models.  DO NOT EDIT
    @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(),
        }


# test code, do not edit


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 {o
        )

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


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
```

Q4 (1 pt): Single batch-update

In [10]:

```
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.

    state, action, reward, next_state, done = batch
    values = dqn_model(state).gather(1, action)
    with torch.no_grad():
        non_final_mask = torch.tensor(tuple(map(lambda s : not(s), done)), dtype=t
        non_final_next_states = torch.stack(tuple(next_state[i] for i in range(len
        next_state_values = torch.zeros(batch_size, 1)
        next_state_values[non_final_mask] = dqn_target(non_final_next_states).deta
        target_values = (batch.rewards + gamma * next_state_values).detach()

    # DO NOT EDIT FURTHER

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

    # testing that the value tensor requires a gradient,
    # and the target_values tensor does not
```

```
    assert values.requires_grad, 'values tensor should not require gradients'
    assert (
        not target_values.requires_grad
    ), 'target_values tensor should require gradients'

    # computing the scalar MSE loss between computed values and the TD-target
    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()
```

Q5 (2 pts):

In [11]:

```
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 openai Gym 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: a 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)

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

    # initiate 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.notebook.tnrange(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

        eps = exploration.value(t_total)
        num = np.random.random()

        if num < eps:
            action = env.action_space.sample()
        else:
            value = torch.tensor(state)
            Q = dqn_model(value).detach().numpy()
            action = np.random.choice(np.nonzero(Q == np.max(Q))[0])

        next_state, reward, done, _ = env.step(action)
        memory.add(state, action, reward, next_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:
            target_network = dqn_model.state_dict()
            dqn_target.load_state_dict(target_network)
#         if t_total % 10_000 == 0:
#             print("num_step: ", t_total, "/", num_steps)

        if done:
            # YOUR CODE HERE:  anything you need to do at the end of an
            # episode, e.g. compute return G, store stuff, reset variables,
            # indices, lists, etc.

            G = 0
            for i in range(len(rewards)):
                G = rewards[i] + gamma * G

            returns.append(G)
            lengths.append(len(rewards))

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

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

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

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

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

Q6 (1 pt): Evaluation of DQN on the 4 environments

CartPole

Test your implentation on the cartpole environment. Training will take much longer than in the previous homeworks, so this time you won't have to find good hyper-parameters, or to train multiple runs. This cell should take about 60-90 minutes 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.

In [12]:

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

# we train for many time-steps;  as usual, you can decrease this during developmen
# 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 90-120 minutes on a generic 4-core laptop
dqn_models, cartpole_returns, cartpole_lengths, cartpole_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')
```

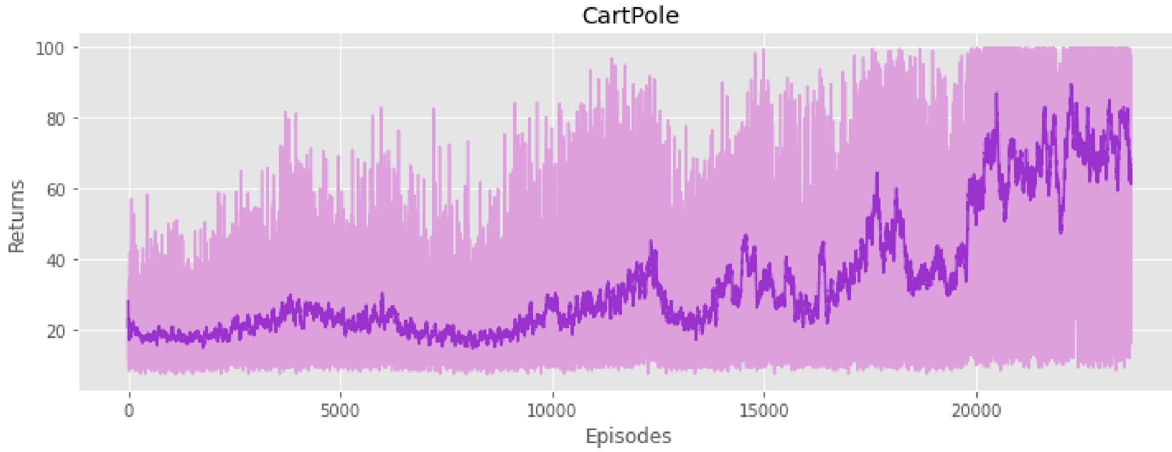
Plot the returns, lengths and losses obtained while running DQN on the cartpole environment.

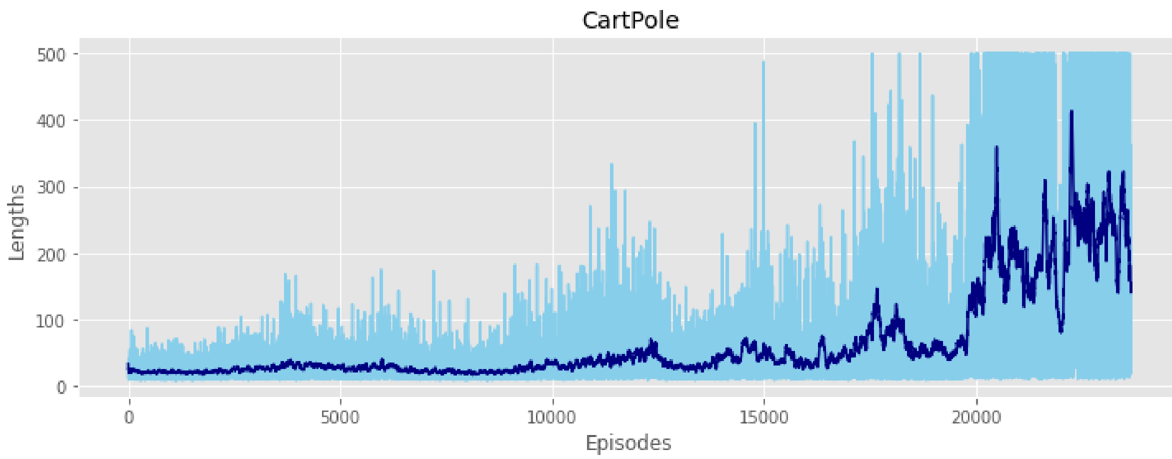
Again, plot the raw data and the smoothened data **inside the same plot**, i.e. you should have 3 plots total.

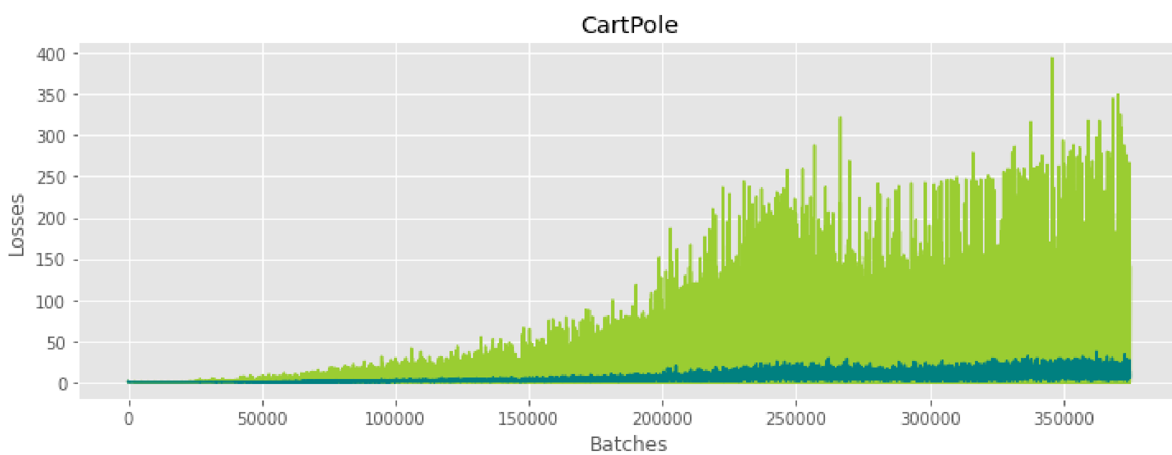
In [25]:

```
### YOUR PLOTTING CODE HERE

plt.figure(0)
plt.plot(cartpole_returns, color='plum')
plt.plot(rolling_average(cartpole_returns, window_size=50), color='darkorchid')
plt.xlabel('Episodes')
plt.ylabel('Returns')
plt.title('CartPole')

plt.figure(1)
plt.plot(cartpole_lengths, color='skyblue')
plt.plot(rolling_average(cartpole_lengths, window_size=50), color='navy')
plt.xlabel('Episodes')
plt.ylabel('Lengths')
plt.title('CartPole')
```

```
plt.figure(2)
plt.plot(cartpole_losses, color='yellowgreen')
plt.plot(rolling_average(cartpole_losses, window_size=50), color='teal')
plt.xlabel('Batches')
plt.ylabel('Losses')
plt.title('CartPole')
plt.show()
```







MountainCar

Test your implentation on the mountaincar environment. Training will take much longer than in the previous homeworks, so this time you won't have to find good hyper-parameters, or to train multiple runs. This cell should take about 60-90 minutes 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.

In [14]:

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

# we train for many time-steps;  as usual, you can decrease this during developmen
# 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 90-120 minutes on a generic 4-core laptop
dqn_models, mountaincar_returns, mountaincar_lengths, mountaincar_losses = train_d
    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')
```

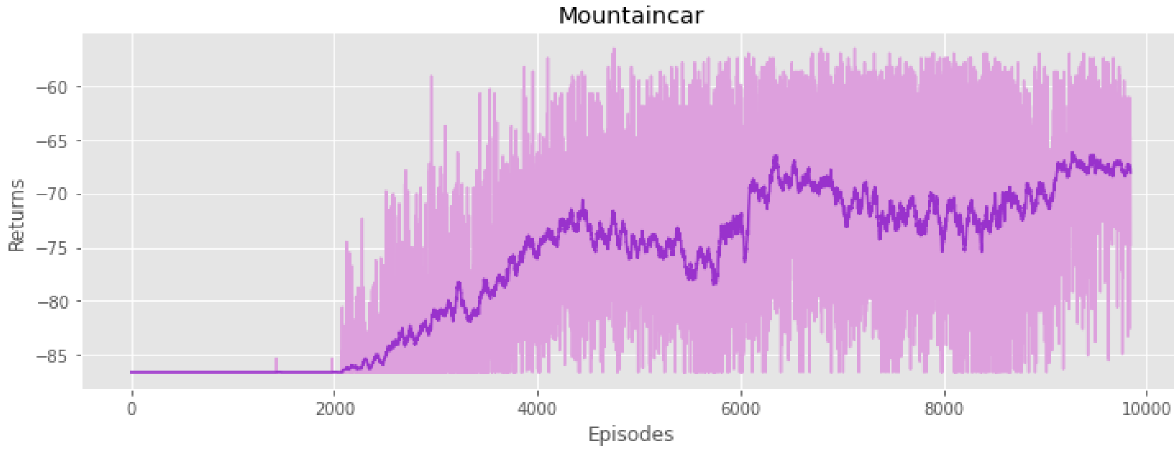
Plot the returns, lengths and losses obtained while running DQN on the mountaincar environment.

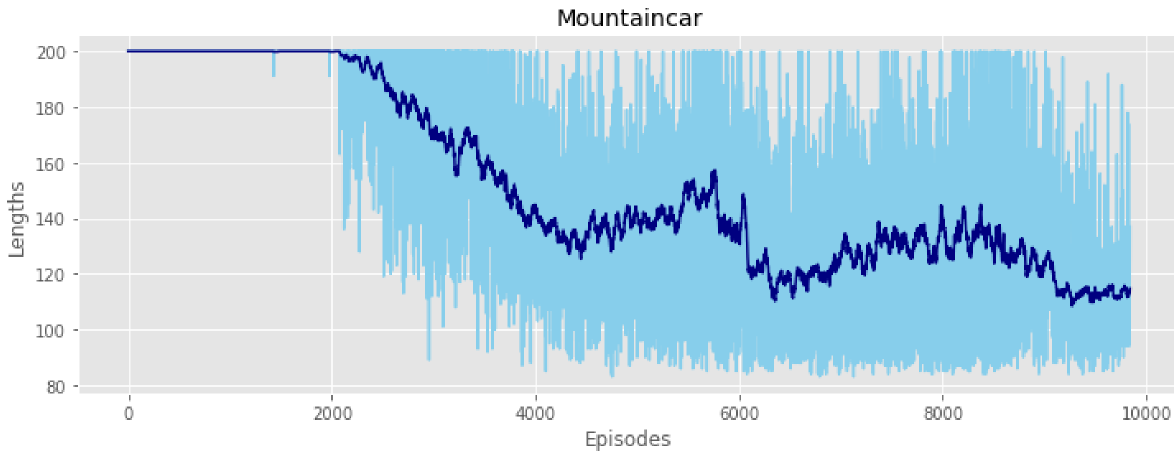
Again, plot the raw data and the smoothened data **inside the same plot**, i.e. you should have 3 plots total.

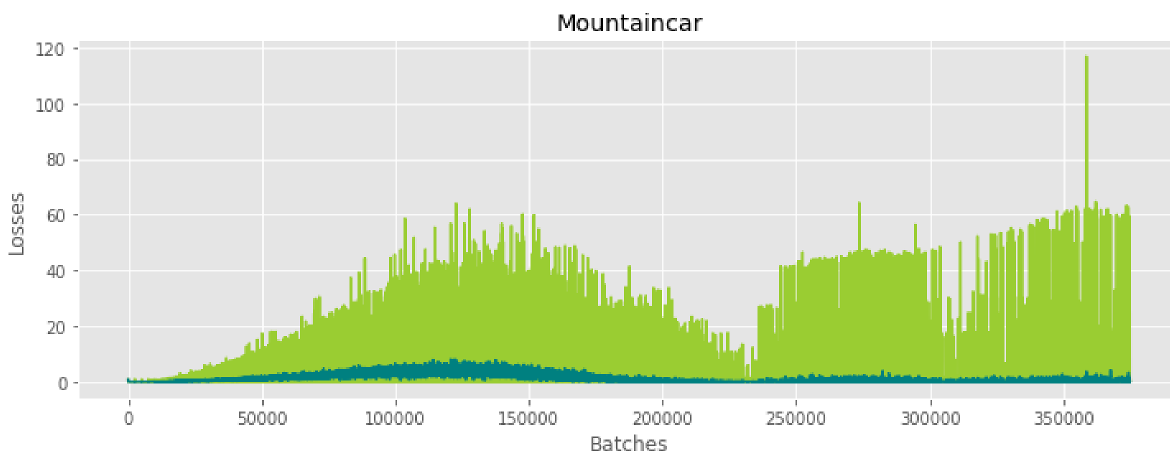
In [23]:

```
### YOUR PLOTTING CODE HERE

plt.figure(0)
plt.plot(mountaincar_returns, color='plum')
plt.plot(rolling_average(mountaincar_returns, window_size=50), color='darkorchid')
plt.xlabel('Episodes')
plt.ylabel('Returns')
plt.title('Mountaincar')

plt.figure(1)
plt.plot(mountaincar_lengths, color='skyblue')
plt.plot(rolling_average(mountaincar_lengths, window_size=50), color='navy')
plt.xlabel('Episodes')
plt.ylabel('Lengths')
plt.title('Mountaincar')
```

```
plt.figure(2)
plt.plot(mountaincar_losses, color='yellowgreen')
plt.plot(rolling_average(mountaincar_losses, window_size=50), color='teal')
plt.xlabel('Batches')
plt.ylabel('Losses')
plt.title('Mountaincar')
plt.show()
```







AcroBot

Test your implentation on the acrobot environment. Training will take much longer than in the previous homeworks, so this time you won't have to find good hyper-parameters, or to train multiple runs. This cell should take about 60-90 minutes 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.

In [16]:

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

# we train for many time-steps;  as usual, you can decrease this during developmen
# 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 90-120 minutes on a generic 4-core laptop
dqn_models, acrobot_returns, acrobot_lengths, acrobot_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')
```

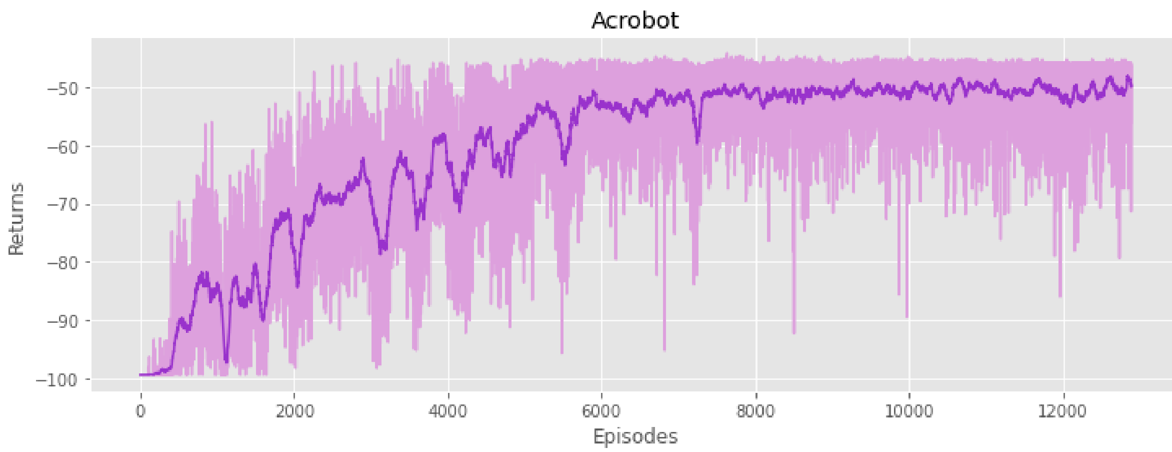
Plot the returns, lengths and losses obtained while running DQN on the acrobot environment.

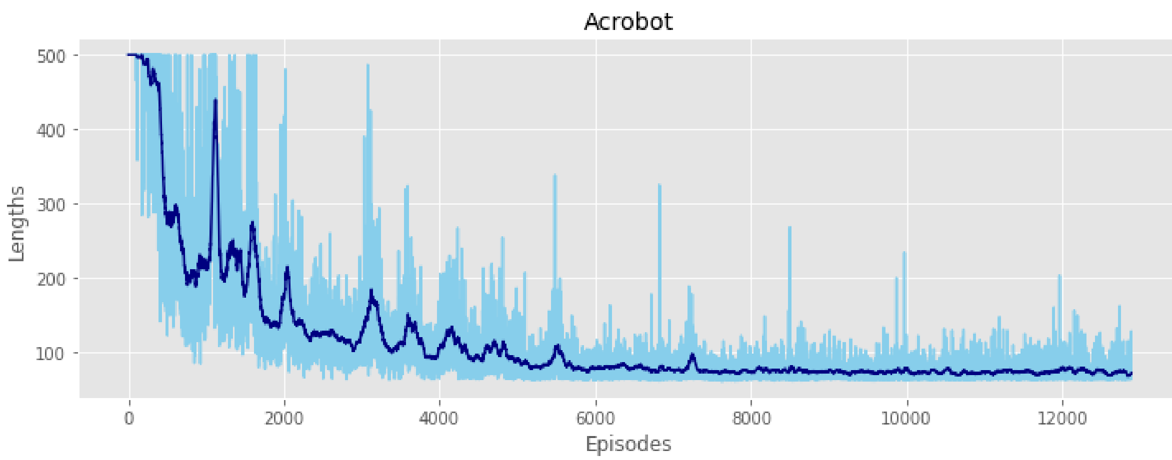
Again, plot the raw data and the smoothened data **inside the same plot**, i.e. you should have 3 plots total.

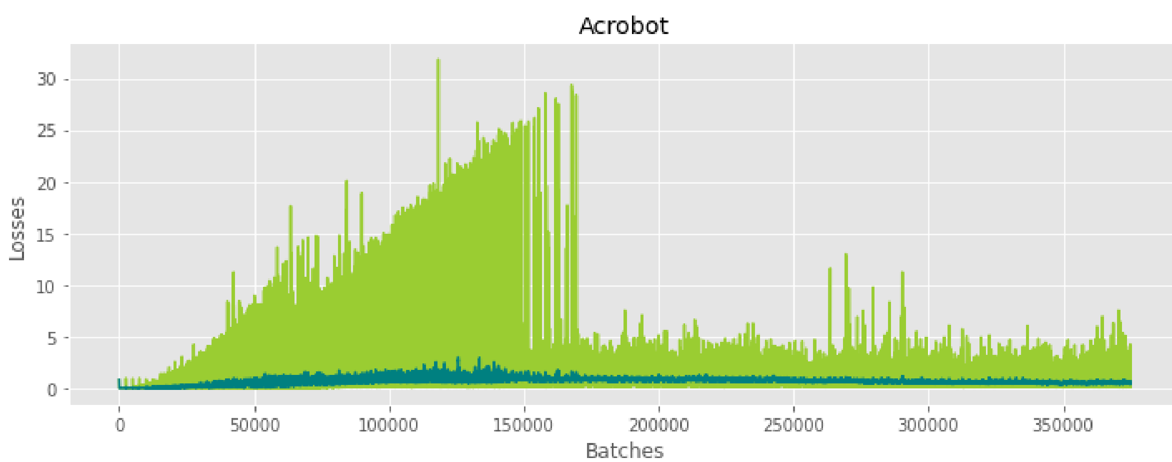
In [22]:

```
### YOUR PLOTTING CODE HERE

plt.figure(0)
plt.plot(acrobot_returns, color='plum')
plt.plot(rolling_average(acrobot_returns, window_size=50), color='darkorchid')
plt.xlabel('Episodes')
plt.ylabel('Returns')
plt.title('Acrobot')

plt.figure(1)
plt.plot(acrobot_lengths, color='skyblue')
plt.plot(rolling_average(acrobot_lengths, window_size=50), color='navy')
plt.xlabel('Episodes')
plt.ylabel('Lengths')
plt.title('Acrobot')
```

```
plt.figure(2)
plt.plot(acrobot_losses, color='yellowgreen')
plt.plot(rolling_average(acrobot_losses, window_size=50), color='teal')
plt.xlabel('Batches')
plt.ylabel('Losses')
plt.title('Acrobot')
plt.show()
```







LunarLander

Test your implentation on the lunarlander environment. Training will take much longer than in the previous homeworks, so this time you won't have to find good hyper-parameters, or to train multiple runs. This cell should take about 60-90 minutes 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.

In [18]:

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

# we train for many time-steps;  as usual, you can decrease this during developmen
# 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 90-120 minutes on a generic 4-core laptop
dqn_models, lunar_returns, lunar_lengths, lunar_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')
```

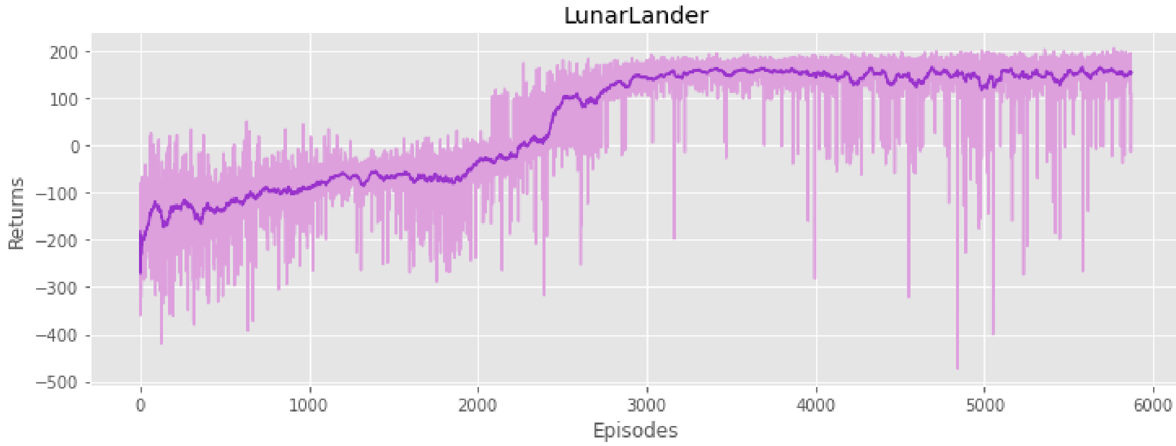
Plot the returns, lengths and losses obtained while running DQN on the lunarlander environment.

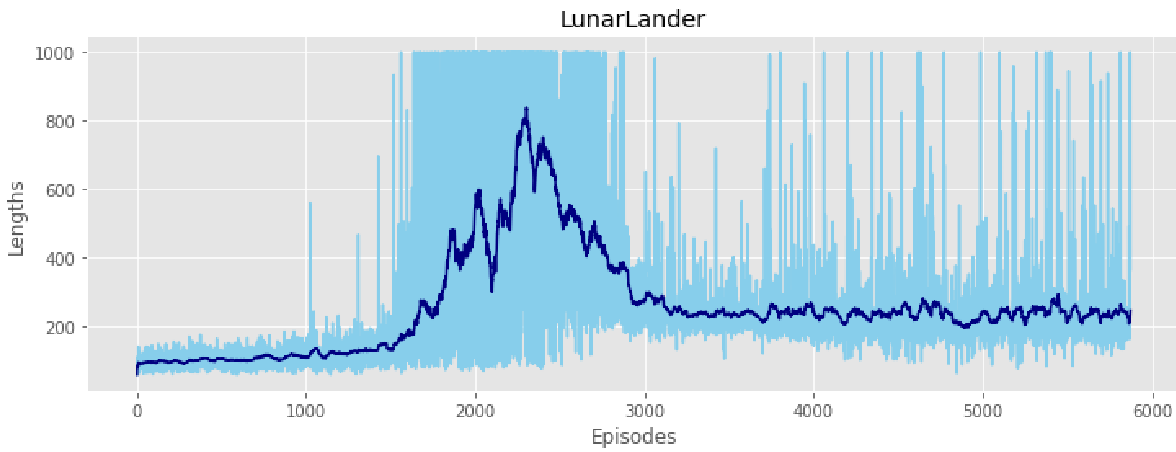
Again, plot the raw data and the smoothened data **inside the same plot**, i.e. you should have 3 plots total.

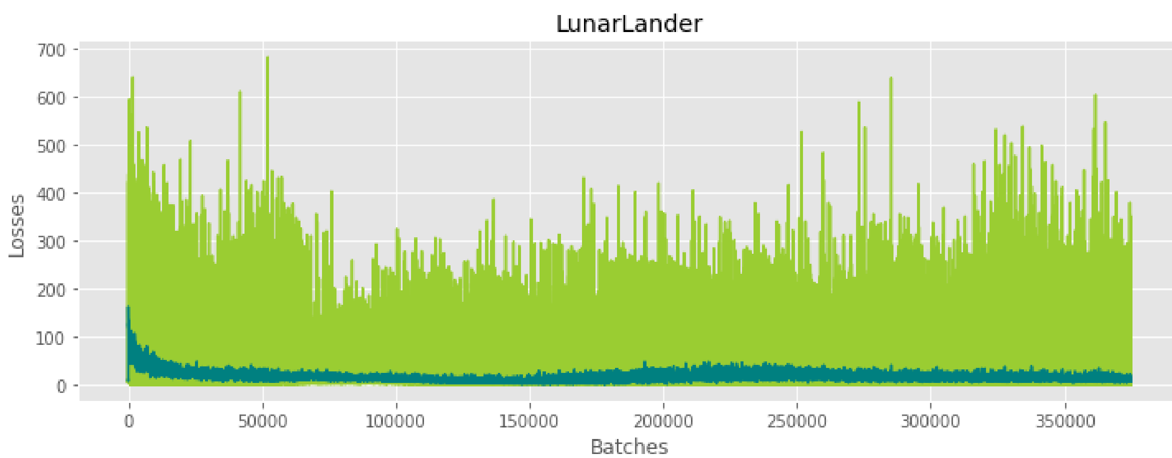
In [21]:

```
### YOUR PLOTTING CODE HERE

plt.figure(0)
plt.plot(lunar_returns, color='plum')
plt.plot(rolling_average(lunar_returns, window_size=50), color='darkorchid')
plt.xlabel('Episodes')
plt.ylabel('Returns')
plt.title('LunarLander')

plt.figure(1)
plt.plot(lunar_lengths, color='skyblue')
plt.plot(rolling_average(lunar_lengths, window_size=50), color='navy')
plt.xlabel('Episodes')
plt.ylabel('Lengths')
plt.title('LunarLander')
```

```
plt.figure(2)
plt.plot(lunar_losses, color='yellowgreen')
plt.plot(rolling_average(lunar_losses, window_size=50), color='teal')
plt.xlabel('Batches')
plt.ylabel('Losses')
plt.title('LunarLander')
plt.show()
```







Visualization of the trained policies!

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

In [20]:

```
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.

                    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)
```

```
cartpole:

mountaincar:

acrobot:

lunarlander:
```

Q7 (2 pts): Analysis

For each environment, 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.

Say something for each phase (i.e. this exercise is worth 1 point for every phase of every environment). 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.

CartPole

- 0%) Whichever direction the cart is moving, the pole on the cart suddenly falls and then terminates.
- 25%) The pole on the cart tends to fall and leads the cart to move in the same direction. When the cart moves outside the window, it terminates. It learns and starts to fall slowly.

- 50%) The cart and pole try to find a stabilized way to move slower and smoother. In a small chance, the cart and pole could stay without falling out of the window. This brings successful experience for the agent to learn.
- 75%) The cart catches the pole and tries to stabilize faster than 50% progress. With enough learning, the agent is able to prevent the pole from falling and succeed most of the time.
- 100%) The cart catches and pole and stabilize in a fixed position with the same time as 75% but it could keep still to a stable state in the environment with a shorter time. It achieves optimality with this num_steps. It achieves optimality at 75% learning process.

MountainCar

- 0%) The agent always tends to move in the opposite direction and go back to the leaving point.
- 25%) The agent is also moving in an opposite direction but with a smaller height and distance.
- 50%) The agent learns to climb up the mountains by this time. It is climbing higher and sliding between mountains to get a higher velocity and speed to and get the flag. It always finishes with 2 complete rushes climbing between mountains.
- 75%) The agent slows down the speed and no more need to climb higher in the opposite direction to rush up to the mountains. The time it took is shorter than the 50% progress one and could success with 1 rushes.
- 100%) This process is pretty much similar to the 75% one. The car successfully climbs up the mountains at a steady and same rate each time. The optimal policy is at the 75% learning process and converged at that time.

Acrobot

- 0%) The agent is randomly slightly moving without learning.
- 25%) The agent is still randomly moving, but the moving angle is slightly larger than the 0% learning process.
- 50%) The agent swings higher and faster with random actions. In a little chance, the agent could reach the goal by swinging multiple times.
- 75%) The agent swings several times even faster and reach the goal in a shorter time.
- 100%) The agent swings to reach the goal in a slightly shorter time. The running time and result are close to the 75% training process, so it might be converged at 75%. It is not the optimal solution.

LunarLander

- 0%) The random action selection cannot control the lander and lead the agent flying randomly to hit the ground or outside the space.
- 25%) The agent learns not to hit the ground with and controls the agent to move slightly in the sky without falling.
- 50%) The agent learns to land and sometimes it may successfully land between the flags but sometimes fly outside. It takes time to move and land the agent to the goal.
- 75%) The agent learns from the previous success and land near the goal quickly and learns how to land smoothly without crashing. It finds the flags quickly but takes time to land.

- 100%) The agent learns to move the lander to near the goal in a shorter time and quickly and find a shorter path that helps to a more efficient way to land near to the goal by taking action earlier than the 75% case. It achieves optimality.

In [ ]:

```

```