# Reinforcement Learning Course: WiSe 2020/21

Marin Bukov

January 10, 2021

## 1 Actor-Critic (AC) Methods

The goal of this Notebook is to become familiar with actor-critic (AC) methods. We will do this by coding up the AC algorithm to solve the Cart Pole problem in RL.

Recall from class that AC methods represent an extension of Policy Gradient methods designed to lower the variance of the policy gradient estimate. Moreover, they provide a natural way to apply policy gradient learning on-line, i.e. perform policy updates before the episode has come to an end.

### 1.1 Basic Theory

We have seen that the policy gradient update remains invariant if a baseline $b$ is subtracted from the reward:

$$\nabla_\theta J(\theta) = \sum_{\{\tau_j\}} \sum_{t=1}^{T} \pi_\theta(a_t^j|s_t^j) \nabla_\theta \log \pi_\theta(a_t^j|s_t^j) \left( \sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'}^j, a_{t'}^j) - b \right)$$

where $\pi_\theta$ is the policy, parametrized by the unknown parameters $\theta$, $r(s,a)$ is the reward function, and $\sum_{\{\tau_j\}}$ is the sum over all trajectories. In particular, this invariance also holds true when the baseline is state-dependent, i.e. $b = b(s)$.

The idea behind actor-critic methods is to introduce a second estimator, parametrized by $\varphi$, which estimates the expected return in state $s$ following the policy $\pi_\theta$. The expected return is known as the value function $V_\varphi(s)$. Note that the parameters $\varphi$ are, in general, independent from the parameters of the policy $\theta$ (although some parameters can be shared, if it is believed that $\pi_\theta$ and $V_\varphi$ are to depend on shared common features present in the states $s$).

Using a single-sample estimate for the expected return under the transition probability, we showed in class that the policy gradient can be re-written with the help of the approximate advantage function

$$A^\pi(s_t, a_t) \approx r(s_t, a_t) + \gamma V^\pi(s_{t+1}) - V^\pi(s_t).$$

where $\gamma$ is the discount factor.

The actor-critic updates thus take the form:

$$\varphi \leftarrow \varphi - \beta \, \text{argmin}_\varphi \frac{1}{2} \sum_{j=1}^{N} \sum_{t=1}^{T} ||V_\varphi^\pi(s_t^j) - y_t^j||^2 \theta \leftarrow \theta + \alpha \sum_{\{\tau_j\}} \sum_{t=1}^{T} \pi_\theta(a_t^j|s_t^j) \nabla_\theta \log \pi_\theta(a_t^j|s_t^j) A^\pi(s_t^j, a_t^j)$$

with the step sizes $\alpha, \beta \in [0,1]$. We discussed two possible estimates for $y_t^j$:

1. MC estimate: $y_t^j = \sum_{t'=t}^{T} r(s_t^j, a_t^j)$

2. Bootstrap/Temporal Difference (TD) estimate: $y_t^j = r(s_t^j, a_t^j) + \gamma V_\varphi^\pi(s_{t+1}^j)$

## 1.2 Actor-Critic Algorithms

In class, we derived two AC algorithms, which we now recap.

### 1.2.1 Offline Actor-Critic Algorithm

The offline AC algorithm, also known as Policy Gradient with Value Function Estimation, can be defined using either of the MC and the Bootstrap/TD estimates. The pseudocode reads as

1. Sample $\{s^j, a^j\}$ from $\pi_\theta$ **(go until the end of episode for each trajectory)** (–> offline).
2. Fit the value function $V_\varphi^\pi(s)$ to the sampled data using the mean-square loss $\mathcal{L}_{\text{critic}}(\varphi)$ and *either of the MC or Bootstrap/TD estimates*:

$$\mathcal{L}_{\text{critic}}(\varphi) = \frac{1}{2}\sum_{j=1}^{N}\sum_{t=1}^{T} ||V_\varphi^\pi(s_t^j) - y_t^j||^2.$$

3. Evaluate the advantage function on the sample:

$$A^\pi(s_t^j, a_t^j) \approx r(s_t^j, a_t^j) + \gamma V^\pi(s_{t+1}^j) - V^\pi(s_t^j).$$

4. Compute the policy gradient on the sample:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{j=1}^{N}\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t^j|s_t^j) A^\pi(s_t^j, a_t^j).$$

5. Update the policy:

$$\theta \leftarrow \theta + \alpha \, \nabla_\theta J(\theta).$$

### 1.2.2 Online Actor-Critic Algorithm

The online AC algorithm can be defined **only using the Bootstrap/TD estimate**. The pseudocode reads as

1. Take action $a \sim \pi_\theta(a|s)$ following policy $\pi_\theta$, and obtain the transition $(s, a, r, s')$.
2. Update $V_\varphi^\pi(s)$ using the Bootstrap/TD target $y(s) = r(s,a) + \gamma V_\varphi^\pi(s')$, and the cost function $\mathcal{L}_{\text{critic}}(\varphi)$:

$$\mathcal{L}_{\text{critic}}(\varphi) = \frac{1}{2}||V_\varphi^\pi(s) - y(s)||^2.$$

3. Compute the advantage function for the transition:

$$A^\pi(s, a) \approx r(s, a) + \gamma V^\pi(s') - V^\pi(s).$$

4. Compute the policy gradient for the transition **(no sums over trajctories and time-steps)** (–> online):

$$\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(a|s) A^\pi(s, a).$$

5. Update the policy:

$$\theta \leftarrow \theta + \alpha \, \nabla_\theta J(\theta).$$

### 1.3 Cart Pole Environment

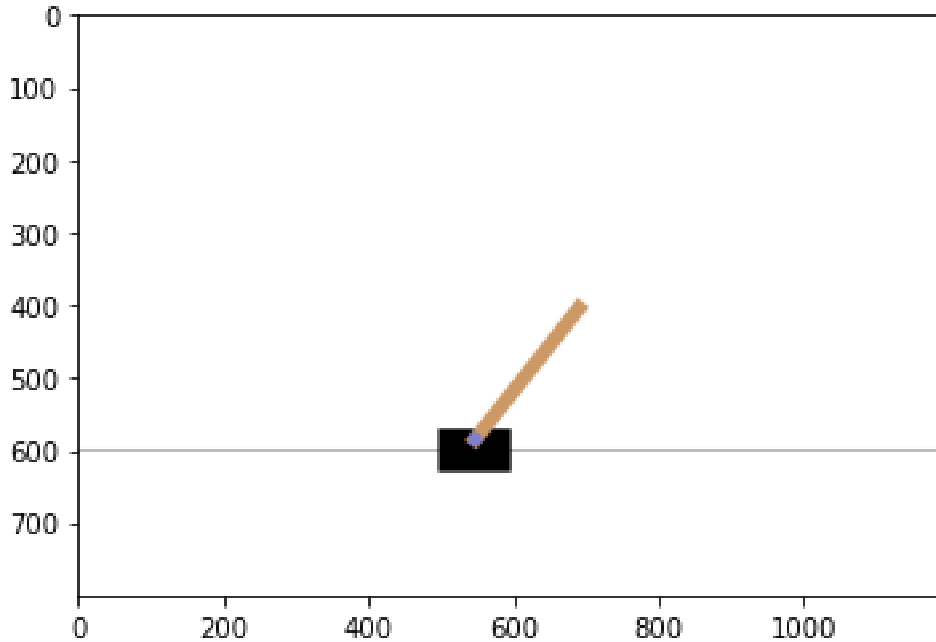
We will apply AC methods on the Cartpole problem, which defines a discounted, non-episodic task.

*A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.*

Let us instantiate and visualize the Cart Pole environment.

[1]:
```
import numpy as np
import gym

# fix numpy rng seed
seed = 42
np.random.seed(seed)

# instantiate environment
env = gym.make("CartPole-v1")

# set environment seed
env.seed(seed)
env.action_space.np_random.seed(seed)

# RL problem parameters
gamma = 0.99  # Discount factor for past rewards
max_steps_per_episode = 10000 # task is non-episodic
return_solved = 300 # return cutoff to consider the task solved
```

[2]:
```
from IPython import display
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

env.reset()
img = plt.imshow(env.render(mode='rgb_array')) # only call this once

for _ in range(80):
    # display settings
    img.set_data(env.render(mode='rgb_array')) # update data
    display.display(plt.gcf())
    display.clear_output(wait=True)
    # choose action
    action = env.action_space.sample()
    # take action
    frame, reward, is_done, _ = env.step(action)
# close pop-up window
env.close()
```




### 1.3.1 State (or Observation) and Action spaces for the Cartpole problem

```
State:
    Type: Box(4)
    Num State                    Min            Max
    0   Cart Position             -4.8            4.8
    1   Cart Velocity             -Inf            Inf
    2   Pole Angle                 -24°           24°
    3   Pole Velocity At Tip      -Inf            Inf

Action:
    Type: Discrete(2)
    Num Action
    0   Push cart to the left
    1   Push cart to the right
```

### 1.3.2 Rewards

As noted above, the reward is +1 at every timestep that the pole remains upright. Since our goal is to find a policy which prevents the pendulum from tipping over, we are presented with a non-episodic task. Therefore, we need an extra condition to define when the task is considered solved.

We render the task solved if the total return within an episode, running-averaged over previous episodes, exceeds a certain cutoff (see variable `return_solved`). The running average is defined by the formula:

```
running_return = 0.05 * episode_return + (1 - 0.05) * running_return
```

Additionally, we also put a large cutoff for the maximal number of steps per episode, see variable max_steps_per_episode above.

## 1.4 Actor-Critic Network

Since the state space is continuous, we can use a deep neural network as a function approximator. We will learn from physical quantities (such as positions, velocities, and angles), not images, and thus we shall focus on an architecture consisting of fully-connected layers.

In order to enable the value function $V_\varphi$ and policy $\pi_\theta$ networks to share common features, we adopt the following architecture, discussed in class:

1. One common base layer with parameters shared by both $V_\varphi$ and $\pi_\theta$, followed by
2. Two independent head leayers, consisting of a $V_\varphi$ head and a $\pi_\theta$ head, which do *not* share parameters.

Thus, whenever the value function is updated, gradients are pushed thru the $V_\varphi$-head and the common layer. Similarly, a policy update changes the $\pi_\theta$-head and the common layer.

The output of the neural network should be a list: the zeroth entry of the list contains the log-probability for the poicy, and the first entry – the value function estimate.

This architecture can be implemented in JAX, by using the stax.serial and stax.parallel modules. stax.serial stacks neural and activation layers on top of each other; stax.parallel puts layers next to each other. To implement splitting the pipeline into parallel heads, we use the stax.FanOut layer (this works similar to stax.Flatten that we used to flatten the output of convolutional layers so it can be fed into a fully-connected layer).

To construct the network, note that the base and heads layers appear in series, becase the base is shared. The base layer should have 128 neurons, followed by a ReLu activation function. The heads layer itself contains the two heads in parallel. While the $V_\varphi$-head has a single number as an output and does not contain any activation functions, the $\pi_\theta$-head has as many outputs as there are actions to take, followed by the LogSoftmax activation; thus, to build the $\pi_\theta$-head, one has to stack in series a Dense layer followed by the LogSoftmax activation.

1. Construct the deep neural network, and test it on a sample dataset.
2. Make sure you understand the output of the network, including the meansing of the shapes/sizes of the ourput.

[3]:
```
import jax.numpy as jnp # jax's numpy version with GPU support
from jax import random # used to define a RNG key to control the random input in JAX
from jax.experimental import stax # neural network library
from jax.experimental.stax import Dense, Relu, LogSoftmax, FanOut # neural network␣
 ↪layers

# set key for the RNG (see JAX docs)
rng = random.PRNGKey(seed)


# define functions which initialize the parameters and evaluate the model
initialize_params, predict = stax.serial(
                                        # common base layer
                                        stax.serial(
                                            ### fully connected DNN
                                            Dense(128), # 128 hidden neurons
                                            Relu, # ReLu activation
                                            ),
```

```
                                          # actor and critic output heads
                                          FanOut(2), # split architecture pipeline into␣
 ↪two heads using FanOut

                                          stax.parallel(
                                              # actor head
                                              stax.serial(
                                                  Dense(env.action_space.n), # 2 output␣
 ↪neurons (actor)

                                                  LogSoftmax # LogSoftmax; NB: computes␣
 ↪the log-probability
                                              ),
                                              # critic head
                                              Dense(1), # 1 output neuron (critic), no␣
 ↪activation
                                          ),
                                  )


# initialize the model parameters
input_shape = (-1,)+env.observation_space.shape # -1: number of time steps, size of␣
 ↪state vector
output_shape, inital_params = initialize_params(rng, input_shape) # fcc layer 28x28␣
 ↪pixes in each image

print('\noutput shape of the AC network is {} for (actor, critic).\n'.
 ↪format(output_shape))


# test network
states=np.ones((3,)+env.observation_space.shape, dtype=np.float32)

actor_predictions, critic_predictions = predict(inital_params, states)
# check the output shape
print("actor head shape:", actor_predictions.shape) # actor
print("critic head shape:", critic_predictions.shape) # critic

# check conservation of probability for actor
print('\nconservation of probability for actor:', np.sum(jnp.exp(actor_predictions),␣
 ↪axis=1))
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0
and rerun for more info.)


output shape of the AC network is ((-1, 2), (-1, 1)) for (actor, critic).

actor head shape: (3, 2)
critic head shape: (3, 1)

conservation of probability for actor: [1. 1. 1.]
```

### 1.5 (Pseudo-) Loss Function

Let us denote the parameters of the common base layer by $\eta$, the policy head parameters – by $\theta$, and the value function head parameters – by $\varphi$.

To appreciate the variance reduction offered by AC algorithms, we will implement the offline AC method using a *single trajectory* to estimate the network gradients. Because the trajectory length can vary (non-episodic task), we use an average over the timesteps within the trajectory.

For the critic loss $\mathcal{L}_{\text{critic}}(\eta, \varphi)$, we use discounted MC estimates $y_t = \sum_{t'=t} \gamma^{t'-t} r(s_{t'}, s_{t'})$, and a Huber loss to cut off excessively large gradients:

$$\mathcal{L}_{\text{critic}}(\eta, \varphi) = \frac{1}{T} \sum_{t=1}^{T} \text{Huber}\left( V^{\pi}_{\eta,\varphi}(s_t), y_t \right).$$

Further, the policy pseudo-loss function is given by (keeping in mind the negative sign required by gradient ascent)

$$\mathcal{L}_{\text{actor}}(\eta, \theta) = -\frac{1}{T} \sum_{t=1}^{T} \log(\pi_{\eta,\theta}(a_t, s_t)) \left( \sum_{t'=t}^{T} \gamma^{t'-t} r(s_t, a_t) - V^{\pi}_{\eta,\varphi}(s_t) \right).$$

Note that no gradient should be pushed thru the critic, $V^{\pi}_{\eta,\varphi}$, here, and hence $\mathcal{L}_{\text{actor}}$ is not considered a function of $\varphi$.

Finally, we also use an L2 regularizer on all network parameters

$$L^2_{\text{reg}}(\eta, \theta, \varphi) = \lambda \left( \sum_l ||\eta_l||^2 + \sum_m ||\theta_m||^2 + \sum_n ||\varphi_n||^2 \right),$$

with $\lambda = 0.001$ the regularization strength.

For simplicity, we perform steps 2, 4, and 5 of the offline AC algorithm together. This is enabled by JAX, which can push the gradients thru the parameters $(\eta, \theta, \varphi)$ at once, using the total cost function

$$\mathcal{L}_{\text{AC}}(\eta, \theta, \varphi) = \mathcal{L}_{\text{actor}}(\eta, \theta) + \mathcal{L}_{\text{critic}}(\eta, \varphi) + L^2_{\text{reg}}(\eta, \theta, \varphi; \lambda).$$

Let us implement the above instructions:

1. Define the Huber Loss function
2. Define the L2 Regularizer
3. Define the total Actor-Critic loss function for a single trajectory. The function body should contain the calculation of the different loss controbutions.

[4]:
```
### define loss and accuracy functions

from jax import grad, lax
from jax.tree_util import tree_flatten # jax params are stored as nested tuples; use␣
↪this to manipulate tuples



def huber_loss(x, delta: float = 1.0):
    """
    # 0.5 * x^2                  if |x| <= delta
    # 0.5 * d^2 + d * (|x| - d)  if |x| > delta
```

```
    """
    abs_x = jnp.abs(x)
    quadratic = jnp.minimum(abs_x, delta)
    # Same as max(abs_x - delta, 0) but avoids potentially doubling the gradient.
    linear = abs_x - quadratic
    return 0.5 * quadratic ** 2 + delta * linear

def l2_regularizer(params, lmbda):
    """
    Define l2 regularizer: $\lambda \ sum_j ||theta_j||^2 $ for every parameter in the␣
↪model $\theta_j$

    """
    return lmbda*jnp.sum(jnp.array([jnp.sum(jnp.abs(theta)**2) for theta in␣
↪tree_flatten(params)[0] ]))


def AC_loss(params, trajectory):
    """
    Define the Actor-Critic loss function.

    params: object(jax pytree):
        parameters of the deep policy network.
    trajectory: tuple (states, actions, returns) containing the RL states, actions and␣
↪returns (not the rewards!):
        states: np.array of size (trajectory length, env.observation_space.shape)
        actions: np.array of size (trajectory length, env.action_space.n)
        returns: np.array of size (trajectory length)

    """
    # extract data from the batch
    states, actions, returns = trajectory
    # compute policy predictions
    actor_preds, critic_preds = predict(params, states)
    critic_preds = critic_preds.squeeze() # remove extra array dimensions
    # select those values of the policy along the action trajectory
    actor_preds_select = jnp.take_along_axis(actor_preds, jnp.expand_dims(actions,␣
↪axis=1), axis=1).squeeze()
    # actor pseudoloss: negative pseudo loss function (want to MAXimize reward with␣
↪gradient DEscent)
    loss_actor = -jnp.mean(actor_preds_select * (returns - lax.
↪stop_gradient(critic_preds) ) )
    # critic loss: use Huber loss
    loss_critic = jnp.mean(huber_loss(critic_preds - returns))
    #
    return loss_actor + loss_critic + l2_regularizer(params, 0.001)
```

### 1.5.1 Define generalized gradient descent optimizer

Define the optimizer and the `update` function which computes the gradient of the pseudo-loss function and performs the update.

We use the Adam optimizer here with `step_size = 0.01` and the rest of the parameters have default values. Since both the actor and the critic are encoded using the same network, we can use a single step size.

[5]:
```
### define generalized gradient descent optimizer and a function to update model␣
↪parameters

from jax.experimental import optimizers # gradient descent optimizers
from jax import jit

step_size = 0.01 # step size or learning rate

# compute optimizer functions
opt_init, opt_update, get_params = optimizers.adam(step_size)


# define function which updates the parameters using the change computed by the␣
↪optimizer
@jit # Just In Time compilation speeds up the code; requires to use jnp everywhere;␣
↪remove when debugging
def update(i, opt_state, trajectory):
    """
    i: int,
        counter to count how many update steps we have performed
    opt_state: object,
        the state of the optimizer
    trajectory: np.array
        batch containing the data used to update the model

    Returns:
    opt_state: object,
        the new state of the optimizer

    """
    # get current parameters of the model
    current_params = get_params(opt_state)
    # compute gradients
    grad_params = grad(AC_loss)(current_params, trajectory)
    # use the optimizer to perform the update using opt_update
    return opt_update(i, grad_params, opt_state)
```

### 1.5.2 Offline Actor-Critic Algorithm

Finally, write down the offline AC algorithm.

Recall that we want to use **single**-trajectory estimates of the neural network gradients.

Moreover, keep in mind that trajectories do not have a fixed length here, so consider using lists instead of arrays.

[6]:
```
### Train model

import time
```

```
# preallocate aux variables
running_return = 0.0
episode = 0


print("\nStart training...\n")

# set the initial model parameters in the optimizer
opt_state = opt_init(inital_params)


while True:  # run until "solved", see break condition below

    # record time
    start_time = time.time()

    # reset environment
    state = env.reset()
    episode_return = 0.0

    # get current parameters
    current_params = get_params(opt_state)

    # preallocate empty lists for the states, actions and rewards within a trajectory
    states,actions,rewards = [],[],[]


    # loop over timesteps of episode to generate a trajectory
    for time_step in range(max_steps_per_episode):

        # record state
        states.append(state)

        # call network to compute \log\pi(:|s)
        log_pi_s, _ = predict(current_params,state)

        # select action according to actor probability distribution
        action = np.random.choice(env.action_space.n, p=np.exp(log_pi_s) )

        # record selected action
        actions.append(action)

        # take action observe next state and receive reward
        state, reward, done, _ = env.step(action)

        # record reward
        rewards.append(reward)

        # update current episode return
        episode_return += reward

        # break if episode has come to an end (i.e. the pendulum has fallen below 15°)
        if done:
```

```
            break

    # compute discounted returns from the bare rewards
    returns = np.array(rewards)
    returns = returns[::-1] * (gamma*np.ones_like(returns) )**np.arange(returns.
↪shape[0])
    returns = jnp.cumsum(returns)[::-1]

    # define trajectory data
    trajectory = (np.array(states), np.array(actions), returns)

    # update model
    opt_state = update(episode, opt_state, trajectory)


    ### record time needed for a single epoch
    episode_time = time.time() - start_time

    # compute running return to check condition for solving the task
    running_return = 0.05 * episode_return + (1 - 0.05) * running_return

    # print stats
    episode += 1
    if episode % 10 == 0:
        template = "episode {}: averaged running return: {:.2f}; took {:0.2f} secs."
        print(template.format(episode, running_return, episode_time))

    ### check if task is considered solved
    if running_return > return_solved:  # condition to consider task solved
        print("\nSolved at episode {} with average running return {}!".format(episode,␣
↪running_return))
        break
```

```
Start training...

episode 10: averaged running return: 16.37; took 0.08 secs.
episode 20: averaged running return: 27.26; took 0.80 secs.
episode 30: averaged running return: 39.75; took 0.70 secs.
episode 40: averaged running return: 48.66; took 0.91 secs.
episode 50: averaged running return: 55.18; took 0.12 secs.
episode 60: averaged running return: 44.94; took 0.66 secs.
episode 70: averaged running return: 33.68; took 0.05 secs.
episode 80: averaged running return: 29.15; took 0.06 secs.
episode 90: averaged running return: 29.65; took 0.07 secs.
episode 100: averaged running return: 30.32; took 0.05 secs.
episode 110: averaged running return: 30.61; took 0.07 secs.
episode 120: averaged running return: 39.97; took 1.04 secs.
episode 130: averaged running return: 41.64; took 0.12 secs.
episode 140: averaged running return: 63.08; took 1.11 secs.
episode 150: averaged running return: 61.51; took 0.93 secs.
episode 160: averaged running return: 69.63; took 0.95 secs.
episode 170: averaged running return: 60.88; took 0.78 secs.
```

```
episode 180: averaged running return: 70.87; took 1.19 secs.
episode 190: averaged running return: 87.42; took 0.34 secs.
episode 200: averaged running return: 111.04; took 1.62 secs.
episode 210: averaged running return: 112.74; took 0.88 secs.
episode 220: averaged running return: 87.47; took 0.07 secs.
episode 230: averaged running return: 71.55; took 0.84 secs.
episode 240: averaged running return: 81.89; took 0.90 secs.
episode 250: averaged running return: 140.31; took 1.34 secs.
episode 260: averaged running return: 140.98; took 0.32 secs.
episode 270: averaged running return: 131.98; took 0.17 secs.
episode 280: averaged running return: 116.49; took 0.20 secs.
episode 290: averaged running return: 157.31; took 1.76 secs.
episode 300: averaged running return: 155.98; took 0.25 secs.
episode 310: averaged running return: 235.59; took 1.10 secs.

Solved at episode 316 with average running return 305.6379016611696!
```

### 1.6 Questions

1. Plot the training curve: running return vs episode number.
2. Check the learned policy: does it make sense physicaly?
3. Modify the network architecture to use two completely independent networks for the policy and the value function. Note that this allows us to use two optimizers, i.e. two independent learning rates. Compare the performance.
4. Modify the code to implement the online AC algorithm.
5. Try solving the Cart Pole problem using the bare images for states instead of the physical quantities.