#PA2 - DQN and Actor-Critic

##Part 1: DQN

In [ ]:
```
'''
Installing packages for rendering the game on Colab
'''

!pip install gym pyvirtualdisplay > /dev/null 2>&1
!apt-get install -y xvfb python-opengl ffmpeg > /dev/null 2>&1
!apt-get update > /dev/null 2>&1
!apt-get install cmake > /dev/null 2>&1
!pip install --upgrade setuptools 2>&1
!pip install ez_setup > /dev/null 2>&1
!pip install gym[atari] > /dev/null 2>&1
!pip install git+https://github.com/tensorflow/docs > /dev/null 2>&1
!pip install numpy matplotlib tqdm scipy
```

In [ ]:
```
!pip install tensorflow-gpu
```

In [14]:
```
'''
A bunch of imports, you don't have to worry about these
'''

import numpy as np
import random
import torch
import torch.nn as nn
import torch.nn.functional as F
from collections import namedtuple, deque
import torch.optim as optim
import datetime
import gym
from gym.wrappers import Monitor
import glob
import io
import base64
import matplotlib.pyplot as plt
from IPython.display import HTML
from pyvirtualdisplay import Display
import tensorflow as tf
from IPython import display as ipythondisplay
from PIL import Image
import tensorflow_probability as tfp
import tqdm
```

In [ ]:
```
'''
Please refer to the first tutorial for more details on the specifics of environments
We've only added important commands you might find useful for experiments.
'''

'''
List of example environments
(Source - https://gym.openai.com/envs/#classic_control)

'Acrobot-v1'
'CartPole-v1'
'MountainCar-v0'
'''

env = gym.make('CartPole-v1')
env.seed(0)

state_shape = env.observation_space.shape[0]
no_of_actions = env.action_space.n

print(state_shape)
print(no_of_actions)
print(env.action_space.sample())
print("----")

'''
# Understanding State, Action, Reward Dynamics

The agent decides an action to take depending on the state.

The Environment keeps a variable specifically for the current state.
- Everytime an action is passed to the environment, it calculates the new state and updates the curre
- It returns the new current state and reward for the agent to take the next action

'''

state = env.reset()
''' This returns the initial state (when environment is reset) '''

print(state)
print("----")

action = env.action_space.sample()
''' We take a random action now '''

print(action)
print("----")

next_state, reward, done, info = env.step(action)
''' env.step is used to calculate new state and obtain reward based on old state and action taken  ''

print(next_state)
print(reward)
print(done)
print(info)
print("----")
```

DQN

Using NNs as substitutes isn't something new. It has been tried earlier, but the 'human control' paper really popularised using NNs by providing a few stability ideas (Q-Targets, Experience Replay & Truncation). The 'Deep-Q Network' (DQN) Algorithm can be broken down into having the following components.

Q-Network:

The neural network used as a function approximator is defined below

In [5]:
```python
'''
### Q Network & Some 'hyperparameters'

QNetwork1:
Input Layer - 4 nodes (State Shape) \
Hidden Layer 1 - 64 nodes \
Hidden Layer 2 - 64 nodes \
Output Layer - 2 nodes (Action Space) \
Optimizer - zero_grad()

QNetwork2: Feel free to experiment more
'''

import torch
import torch.nn as nn
import torch.nn.functional as F


'''
Bunch of Hyper parameters (Which you might have to tune later **wink wink**)
'''
BUFFER_SIZE = int(1e5)  #replay buffer size
BATCH_SIZE = 64         #minibatch size
GAMMA = 0.99            #discount factor
LR = 0.00025               #learning rate
UPDATE_EVERY = 5      #how often to update the network (When Q target is present)


class QNetwork1(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=128, fc2_units=128):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork1, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

Replay Buffer:

This is a 'deque' that helps us store experiences. Recall why we use such a technique.

In [ ]:
```
import random
import torch
import numpy as np
from collections import deque, namedtuple

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

class ReplayBuffer:
    """Fixed-size buffer to store experience tuples."""

    def __init__(self, action_size, buffer_size, batch_size, seed):
        """Initialize a ReplayBuffer object.

        Params
        ======
            action_size (int): dimension of each action
            buffer_size (int): maximum size of buffer
            batch_size (int): size of each training batch
            seed (int): random seed
        """
        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state", "action", "reward", "next_st
        self.seed = random.seed(seed)

    def add(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, next_state, done)
        self.memory.append(e)

    def sample(self):
        """Randomly sample a batch of experiences from memory."""
        experiences = random.sample(self.memory, k=self.batch_size)

        states = torch.from_numpy(np.vstack([e.state for e in experiences if e is not None])).float()
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not None])).long(
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not None])).float
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if e is not None]
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None]).astype(np.

        return (states, actions, rewards, next_states, dones)

    def __len__(self):
        """Return the current size of internal memory."""
        return len(self.memory)
```

Truncation:

We add a line (optionally) in the code to truncate the gradient in hopes that it would help with the stability of the learning process.

Tutorial Agent Code (ε-Greedy):

In [ ]:
```
class TutorialAgent():

    def __init__(self, state_size, action_size, seed):

        ''' Agent Environment Interaction '''
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)

        ''' Q-Network '''
        self.qnetwork_local = QNetwork1(state_size, action_size, seed).to(device)
        self.qnetwork_target = QNetwork1(state_size, action_size, seed).to(device)
        self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=LR)

        ''' Replay memory '''
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, seed)

        ''' Initialize time step (for updating every UPDATE_EVERY steps)           -Needed for Q Targ
        self.t_step = 0

    def step(self, state, action, reward, next_state, done):

        ''' Save experience in replay memory '''
        self.memory.add(state, action, reward, next_state, done)

        ''' If enough samples are available in memory, get random subset and learn '''
        if len(self.memory) >= BATCH_SIZE:
            experiences = self.memory.sample()
            self.learn(experiences, GAMMA)

        """ +Q TARGETS PRESENT """
        ''' Updating the Network every 'UPDATE_EVERY' steps taken '''
        self.t_step = (self.t_step + 1) % UPDATE_EVERY
        if self.t_step == 0:

            self.qnetwork_target.load_state_dict(self.qnetwork_local.state_dict())

    def act(self, state, eps=0.):

        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        self.qnetwork_local.eval()
        with torch.no_grad():
            action_values = self.qnetwork_local(state)
        self.qnetwork_local.train()

        ''' Epsilon-greedy action selection (Already Present) '''
        if random.random() > eps:
            return np.argmax(action_values.cpu().data.numpy())
        else:
            return random.choice(np.arange(self.action_size))

    def learn(self, experiences, gamma):
        """ +E EXPERIENCE REPLAY PRESENT """
        states, actions, rewards, next_states, dones = experiences

        ''' Get max predicted Q values (for next states) from target model'''
        Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)

        ''' Compute Q targets for current states '''
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

        ''' Get expected Q values from local model '''
        Q_expected = self.qnetwork_local(states).gather(1, actions)

        ''' Compute loss '''
        loss = F.mse_loss(Q_expected, Q_targets)

        ''' Minimize the loss '''
        self.optimizer.zero_grad()
        loss.backward()

        ''' Gradiant Clipping '''
        """ +T TRUNCATION PRESENT """
        for param in self.qnetwork_local.parameters():
            param.grad.data.clamp_(-1, 1)
```

```
        self.optimizer.step()
```

###DQN algorithm code(epsilon-greedy):

In [ ]:
```
''' Defining DQN Algorithm '''

state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n
n_episodes=10000


def dqn(n_episodes=100, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.9975):

    scores = []
    ''' list containing scores from each episode '''

    scores_window_printing = deque(maxlen=10)
    ''' For printing in the graph '''

    scores_window= deque(maxlen=100)
    ''' last 100 scores for checking if the avg is more than 195 '''

    eps = eps_start
    ''' initialize epsilon '''

    episode_rewards_e = np.zeros(n_episodes)
    steps_to_completion_e = np.zeros(n_episodes)
    for i_episode in tqdm(range(1, n_episodes)):
        state = env.reset()
        score = 0
        for t in range(max_t):
            action = agent.act(state, eps)
            next_state, reward, done, _ = env.step(action)
            agent.step(state, action, reward, next_state, done)
            state = next_state
            score += reward
            if done:
                break

        steps_to_completion_e[i_episode] = t
        episode_rewards_e[i_episode] = score

        scores_window.append(score)
        scores_window_printing.append(score)
        ''' save most recent score '''

        eps = max(eps_end, eps_decay*eps)
        ''' decrease epsilon '''

        #print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)), end="
        if i_episode % 10 == 0:
            scores.append(np.mean(scores_window_printing))
        '''if i_episode % 100 == 0:
           print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)))'''
        if np.mean(scores_window)>=195:
           print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i_episode-100
           break
    return [np.array(scores),i_episode-100],scores,steps_to_completion_e,episode_rewards_e, np.mean(s
```

###Running 10 independent Experiments

In [ ]:
```
from operator import add
scores_avgs, add_rewards, add_steps = [], [], []
num_exp = 10
for i in range(num_exp):
    begin_time = datetime.datetime.now()
    agent = TutorialAgent(state_size=state_shape, action_size=action_shape, seed=0)

    print("====================Experiment: " + str(i + 1) + "====================")
    score_arr, scores, steps_to_completion_e, episode_rewards_e, scores_window = dqn()
    time_taken = datetime.datetime.now() - begin_time
    scores_avgs.append(scores_window)
    if (i==0):
      add_rewards=episode_rewards_e
      add_steps = steps_to_completion_e
    else:
      add_rewards=list(map(add,add_rewards,episode_rewards_e))
      add_steps=list(map(add,add_steps,steps_to_completion_e))
    print(time_taken)
    avg_rewards = list(map(lambda x: x/num_exp, add_rewards))
    avg_steps = list(map(lambda y:y/num_exp, add_steps))

print("Average score over " + str(num_exp) + " experiments: ", np.mean(scores_avgs))
```

In [ ]:
```
plt.plot(np.arange(len(avg_rewards)),avg_rewards)
plt.ylabel("avg rewards")
plt.xlabel("Episodes")
plt.show()

plt.plot(np.arange(len(avg_steps)),avg_steps)
plt.ylabel("avg steps")
plt.xlabel("Episodes")
plt.show()
```

Tutorial Agent Code (Softmax):

In [ ]:
```
from scipy.special import softmax


class TutorialAgent():

    def __init__(self, state_size, action_size, seed):

        ''' Agent Environment Interaction '''
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)

        ''' Q-Network '''
        self.qnetwork_local = QNetwork1(state_size, action_size, seed).to(device)
        self.qnetwork_target = QNetwork1(state_size, action_size, seed).to(device)
        self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=LR)

        ''' Replay memory '''
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, seed)

        ''' Initialize time step (for updating every UPDATE_EVERY steps)           -Needed for Q Targ
        self.t_step = 0

    def step(self, state, action, reward, next_state, done):

        ''' Save experience in replay memory '''
        self.memory.add(state, action, reward, next_state, done)

        ''' If enough samples are available in memory, get random subset and learn '''
        if len(self.memory) >= BATCH_SIZE:
            experiences = self.memory.sample()
            self.learn(experiences, GAMMA)

        """ +Q TARGETS PRESENT """
        ''' Updating the Network every 'UPDATE_EVERY' steps taken '''
        self.t_step = (self.t_step + 1) % UPDATE_EVERY
        if self.t_step == 0:

            self.qnetwork_target.load_state_dict(self.qnetwork_local.state_dict())

    def act(self, state, temp1):

        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        self.qnetwork_local.eval()
        with torch.no_grad():
            action_values = self.qnetwork_local(state)
        self.qnetwork_local.train()

        ''' Softmax action selection '''
        actionVal = action_values.cpu().data.numpy()[0]
        n = np.exp((actionVal- np.max(actionVal, axis=-1, keepdims=True))/temp1)
        d=np.sum(n, axis=-1, keepdims=True)
        prob = n/d

        return np.random.choice(2, p=prob)


    def learn(self, experiences, gamma):
        """ +E EXPERIENCE REPLAY PRESENT """
        states, actions, rewards, next_states, dones = experiences

        ''' Get max predicted Q values (for next states) from target model'''
        Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)

        ''' Compute Q targets for current states '''
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

        ''' Get expected Q values from local model '''
        Q_expected = self.qnetwork_local(states).gather(1, actions)

        ''' Compute loss '''
        loss = F.mse_loss(Q_expected, Q_targets)

        ''' Minimize the loss '''
        self.optimizer.zero_grad()
        loss.backward()
```

```
        ''' Gradiant Clipping '''
        """ +T TRUNCATION PRESENT """
        for param in self.qnetwork_local.parameters():
            param.grad.data.clamp_(-1, 1)

        self.optimizer.step()
```

###DQN algorithm code:

In [ ]:
```
''' Defining DQN Algorithm '''

state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n
n_episodes=10000



def dqn(n_episodes=10000, max_t=1000, temp_start=5.0, temp_end=0.01, temp_decay=0.995):

    scores = []
    ''' list containing scores from each episode '''

    scores_window_printing = deque(maxlen=10)
    ''' For printing in the graph '''

    scores_window= deque(maxlen=100)
    ''' last 100 scores for checking if the avg is more than 195 '''

    temp = temp_start
    ''' initialize epsilon '''
    episode_rewards = np.zeros(n_episodes)
    steps_to_completion = np.zeros(n_episodes)

    for i_episode in tqdm(range(1, n_episodes+1)):
        state = env.reset()
        score = 0
        for t in range(max_t):
            action = agent.act(state, temp)
            next_state, reward, done, _ = env.step(action)
            agent.step(state, action, reward, next_state, done)
            state = next_state
            score += reward
            if done:
                break
        steps_to_completion[i_episode] = t
        episode_rewards[i_episode] = score


        scores_window.append(score)
        scores_window_printing.append(score)
        ''' save most recent score '''

        temp = max(temp_end, temp_decay*temp)
        ''' decrease epsilon '''

        #print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)), end="
        if i_episode % 10 == 0:
            scores.append(np.mean(scores_window_printing))
        '''if i_episode % 100 == 0:
           print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)))'''
        if np.mean(scores_window)>=195:
           print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i_episode-100
           break
    return [np.array(scores),i_episode-100], scores, episode_rewards, steps_to_completion, np.mean(sc
```

###Running 10 independent experiments

In [ ]:
```
from operator import add
scores_avgs, add_rewards, add_steps = [], [], []
num_exp = 10
for i in range(num_exp):
    begin_time = datetime.datetime.now()
    agent = TutorialAgent(state_size=state_shape, action_size=action_shape, seed=0)

    print("====================Experiment: " + str(i + 1) + "====================")
    score_arr, scores, steps_to_completion_e, episode_rewards_e, scores_window = dqn()
    time_taken = datetime.datetime.now() - begin_time
    scores_avgs.append(scores_window)
    if (i==0):
      add_rewards=episode_rewards_e
      add_steps = steps_to_completion_e
    else:
      add_rewards=list(map(add,add_rewards,episode_rewards_e))
      add_steps=list(map(add,add_steps,steps_to_completion_e))
    print(time_taken)
    avg_rewards = list(map(lambda x: x/num_exp, add_rewards))
    avg_steps = list(map(lambda y:y/num_exp, add_steps))

print("Average score over " + str(num_exp) + " experiments: ", np.mean(scores_avgs))
```

###Plotting graphs

In [ ]:
```
plt.plot(np.arange(len(avg_rewards)),avg_rewards)
plt.ylabel("avg rewards")
plt.xlabel("Episodes")
plt.show()

plt.plot(np.arange(len(avg_steps)),avg_steps)
plt.ylabel("avg steps")
plt.xlabel("Episodes")
plt.show()
```

###Plotting episodes vs Scores (moving average)

Part 2: One-Step Actor-Critic Algorithm

Actor-Critic methods learn both a policy $\pi(a|s;\theta)$ and a state-value function $v(s;w)$ simultaneously. The policy is referred to as the actor that suggests actions given a state. The estimated value function is referred to as the critic. It evaluates actions taken by the actor based on the given policy. In this exercise, both functions are approximated by feedforward neural networks.

- The policy network is parametrized by θ - it takes a state s as input and outputs the probabilities $\pi(a|s;\theta)\ \forall\ a$
- The value network is parametrized by w - it takes a state s as input and outputs a scalar value associated with the state, i.e., $v(s;w)$
- The single step TD error can be defined as follows:
$$\delta_t = R_{t+1} + \gamma v(s_{t+1};w) - v(s_t;w)$$
- The loss function to be minimized at every step ($L_{tot}^{(t)}$) is a summation of two terms, as follows:
$$L_{tot}^{(t)} = L_{actor}^{(t)} + L_{critic}^{(t)}$$
where,
$$L_{actor}^{(t)} = -\log \pi(a_t|s_t;\theta)\delta_t$$
$$L_{critic}^{(t)} = \delta_t^2$$
- **NOTE: Here, weights of the first two hidden layers are shared by the policy and the value network**
  - First two hidden layer sizes: [1024, 512]
  - Output size of policy network: 2 (Softmax activation)
  - Output size of value network: 1 (Linear activation)

Type *Markdown* and LaTeX: α^2

###**Task 1 Answer** Softmax is better, as it is converging in 1013 episodes.

Initializing Actor-Critic Network

In [6]:
```
class ActorCriticModel(tf.keras.Model):
    """
    Defining policy and value networkss
    """
    def __init__(self, action_size, n_hidden1=256, n_hidden2=256):
        super(ActorCriticModel, self).__init__()

        #Hidden Layer 1
        self.fc1 = tf.keras.layers.Dense(n_hidden1, activation='relu')
        #Hidden Layer 2
        self.fc2 = tf.keras.layers.Dense(n_hidden2, activation='relu')

        #Output Layer for policy
        self.pi_out = tf.keras.layers.Dense(action_size, activation='softmax')
        #Output Layer for state-value
        self.v_out = tf.keras.layers.Dense(1)

    def call(self, state):
        """
        Computes policy distribution and state-value for a given state
        """
        layer1 = self.fc1(state)
        layer2 = self.fc2(layer1)

        pi = self.pi_out(layer2)
        v = self.v_out(layer2)

        return pi, v
```

Agent Class

###**Task 2a:** Write code to compute δ_t inside the Agent.learn() function

In [7]:
```
class Agent:
    """
    Agent class
    """
    def __init__(self, action_size, lr=0.0002, gamma=0.98, seed = 85):
        self.gamma = gamma
        self.ac_model = ActorCriticModel(action_size=action_size)
        self.ac_model.compile(tf.keras.optimizers.Adam(learning_rate=lr))
        np.random.seed(seed)

    def sample_action(self, state):
        """
        Given a state, compute the policy distribution over all actions and sample one action
        """
        pi,_ = self.ac_model(state)

        action_probabilities = tfp.distributions.Categorical(probs=pi)
        sample = action_probabilities.sample()

        return int(sample.numpy()[0])

    def actor_loss(self, action, pi, delta):
        """
        Compute Actor Loss
        """
        return -tf.math.log(pi[0,action]) * delta

    def critic_loss(self,delta):
        """
        Critic loss aims to minimize TD error
        """
        return delta**2

    @tf.function
    def learn(self, state, action, reward, next_state, done):
        """
        For a given transition (s,a,s',r) update the paramters by computing the
        gradient of the total loss
        """
        with tf.GradientTape(persistent=True) as tape:
            pi, V_s = self.ac_model(state)
            _, V_s_next = self.ac_model(next_state)

            #V_s_next = tf.stop_gradient(V_s_next)

            V_s = tf.squeeze(V_s)
            V_s_next = tf.squeeze(V_s_next)


            #### TO DO: Write the equation for delta (TD error)
            ## Write code below
            delta = reward + self.gamma*V_s_next - V_s

            loss_a = self.actor_loss(action, pi, delta)
            loss_c =self.critic_loss(delta)
            loss_total = loss_a + loss_c

        gradient = tape.gradient(loss_total, self.ac_model.trainable_variables)
        self.ac_model.optimizer.apply_gradients(zip(gradient, self.ac_model.trainable_variables))
```

Train the Network

In [10]:
```
env = gym.make('Acrobot-v1')

def one_step_AC():
  #Initializing Agent
  agent = Agent(lr=2e-4, action_size=env.action_space.n)
  #Number of episodes
  episodes = 1800

  s=[]
  tf.compat.v1.reset_default_graph()

  reward_list = np.zeros(episodes+1)
  average_reward_list = []
  step_list=[]
  variance_episodic_reward = []
  begin_time = datetime.datetime.now()

  #for ep in tqdm(range(1, episodes+1)):
  with tqdm.trange(1, episodes+1) as t:
    for ep in t:
      a=0
      state = env.reset().reshape(1,-1)
      done = False
      ep_rew = 0
      while not done:
          a+=1
          action = agent.sample_action(state) ##Sample Action
          next_state, reward, done, info = env.step(action) ##Take action
          next_state = next_state.reshape(1,-1)
          ep_rew += reward  ##Updating episode reward
          agent.learn(state, action, reward, next_state, done) ##Update Parameters
          state = next_state ##Updating State
      reward_list[ep] = ep_rew

      s.append(a)
      #step_list[ep+1]=ep
      '''if ep % 100 == 0:
          avg_rew = np.mean(reward_list[-10:])
          print('Episode ', ep, 'Reward %f' % ep_rew, 'Average Reward %f' % avg_rew)'''

      if ep > 100:
          avg_rew = np.mean(reward_list[-100:])

          t.set_description(f'Episode {ep}')
          t.set_postfix(ep_rew = ep_rew, avg_rew = avg_rew)

      if ep > 100:
          avg_100 =  np.mean(reward_list[-100:])

          average_reward_list.append(avg_100)
          if avg_100 > -81.0:
              print('Stopped at Episode ',ep-100)
              break
  variance=np.var(reward_list)
  variance_episodic_reward.append(variance)
  time_taken = datetime.datetime.now() - begin_time
  print(time_taken)
  return reward_list, variance_episodic_reward,s
```

In [ ]:
```
variance_list = []
from operator import add
scores_avgs, add_rewards, add_steps = [], [], []
num_exp = 10
for i in range(num_exp):
    print("====================Experiment: " + str(i + 1) + "====================")
    reward_list, variance,ep = one_step_AC()

    variance_list.append(variance)
    if (i==0):
      add_rewards=reward_list
      add_steps = ep

    else:
      add_rewards=list(map(add,add_rewards,reward_list))
      add_steps=list(map(add,add_steps,ep))

    avg_rewards = list(map(lambda x: x/num_exp, add_rewards))
    avg_steps = list(map(lambda y:y/num_exp, ep))

#print("Average score over " + str(num_exp) + " experiments: ", np.mean(scores_avgs))
```

Task 2b: Plot total reward curve

In the cell below, write code to plot the total reward averaged over 100 episodes (moving average)

In [ ]:
```
### Plot of total reward vs episode
## Write Code Below

plt.xlabel('Episodes')
plt.ylabel('Total Reward')
plt.plot(np.arange(19), avg_rewards)
plt.show()

plt.xlabel('Episodes')
plt.ylabel('Variance')
plt.plot(np.arange(10), variance_list)
plt.show()

plt.xlabel('Episodes')
plt.ylabel('steps')
plt.plot(np.arange(18),avg_steps)
plt.show()
```

Full step Actor Critic

In [161]:
```
import collections
import gym
import numpy as np
import statistics
import tensorflow as tf
import tqdm

from matplotlib import pyplot as plt
from tensorflow.keras import layers
env = gym.make("CartPole-v1")

# Set seed for experiment reproducibility
seed = 42
env.seed(seed)
tf.random.set_seed(seed)
np.random.seed(seed)

# Small epsilon value for stabilizing division operations
eps = np.finfo(np.float32).eps.item()
```

In [162]:
```
from typing import Any, List, Sequence, Tuple
class ActorCritic(tf.keras.Model):
  """Combined actor-critic network."""

  def __init__(
      self,
      num_actions: int,
      num_hidden_units: int):
    """Initialize."""
    super().__init__()

    self.common = layers.Dense(num_hidden_units, activation="relu")
    self.actor = layers.Dense(num_actions)
    self.critic = layers.Dense(1)

  def call(self, inputs: tf.Tensor) -> Tuple[tf.Tensor, tf.Tensor]:
    x = self.common(inputs)
    return self.actor(x), self.critic(x)
```

In [163]:
```
class ActorCritic(tf.keras.Model):
  """Combined actor-critic network."""

  def __init__(
      self,
      num_actions: int,
      num_hidden_units: int):
    """Initialize."""
    super().__init__()

    self.common = layers.Dense(num_hidden_units, activation="relu")
    self.actor = layers.Dense(num_actions)
    self.critic = layers.Dense(1)

  def call(self, inputs: tf.Tensor) -> Tuple[tf.Tensor, tf.Tensor]:
    x = self.common(inputs)
    return self.actor(x), self.critic(x)
```

In [164]:
```
num_actions = env.action_space.n  # 2
num_hidden_units = 128

model = ActorCritic(num_actions, num_hidden_units)
```

In [165]:
```
# Wrap OpenAI Gym's `env.step` call as an operation in a TensorFlow function.
# This would allow it to be included in a callable TensorFlow graph.

def env_step(action: np.ndarray) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
  """Returns state, reward and done flag given an action."""

  state, reward, done, _ = env.step(action)
  return (state.astype(np.float32),
          np.array(reward, np.int32),
          np.array(done, np.int32))


def tf_env_step(action: tf.Tensor) -> List[tf.Tensor]:
  return tf.numpy_function(env_step, [action],
                           [tf.float32, tf.int32, tf.int32])
```

In [166]:
```
def run_episode(
    initial_state: tf.Tensor,
    model: tf.keras.Model,
    max_steps: int) -> Tuple[tf.Tensor, tf.Tensor, tf.Tensor]:
  """Runs a single episode to collect training data."""

  action_probs = tf.TensorArray(dtype=tf.float32, size=0, dynamic_size=True)
  values = tf.TensorArray(dtype=tf.float32, size=0, dynamic_size=True)
  rewards = tf.TensorArray(dtype=tf.int32, size=0, dynamic_size=True)
  #steps1 = tf.TensorArray(dtype=tf.int32, size=0, dynamic_size=True)

  initial_state_shape = initial_state.shape
  state = initial_state

  for t in tf.range(max_steps):
    # Convert state into a batched tensor (batch size = 1)
    state = tf.expand_dims(state, 0)

    # Run the model and to get action probabilities and critic value
    action_logits_t, value = model(state)

    # Sample next action from the action probability distribution
    action = tf.random.categorical(action_logits_t, 1)[0, 0]
    action_probs_t = tf.nn.softmax(action_logits_t)

    # Store critic values
    values = values.write(t, tf.squeeze(value))

    # Store log probability of the action chosen
    action_probs = action_probs.write(t, action_probs_t[0, action])

    # Apply action to the environment to get next state and reward
    state, reward, done = tf_env_step(action)
    state.set_shape(initial_state_shape)

    # Store reward
    rewards = rewards.write(t, reward)

    if tf.cast(done, tf.bool):
      break

  action_probs = action_probs.stack()
  values = values.stack()
  rewards = rewards.stack()

  return action_probs, values, rewards
```

In [167]:
```
def get_expected_return(
    rewards: tf.Tensor,
    gamma: float,
    standardize: bool = True) -> tf.Tensor:
  """Compute expected returns per timestep."""

  n = tf.shape(rewards)[0]
  returns = tf.TensorArray(dtype=tf.float32, size=n)

  # Start from the end of `rewards` and accumulate reward sums
  # into the `returns` array
  rewards = tf.cast(rewards[::-1], dtype=tf.float32)
  discounted_sum = tf.constant(0.0)
  discounted_sum_shape = discounted_sum.shape
  for i in tf.range(n):
    reward = rewards[i]
    discounted_sum = reward + gamma * discounted_sum
    discounted_sum.set_shape(discounted_sum_shape)
    returns = returns.write(i, discounted_sum)
  returns = returns.stack()[::-1]

  if standardize:
    returns = ((returns - tf.math.reduce_mean(returns)) /
               (tf.math.reduce_std(returns) + eps))

  return returns
```

In [168]:
```
huber_loss = tf.keras.losses.Huber(reduction=tf.keras.losses.Reduction.SUM)

def compute_loss(
    action_probs: tf.Tensor,
    values: tf.Tensor,
    returns: tf.Tensor) -> tf.Tensor:
  """Computes the combined actor-critic loss."""

  advantage = returns - values

  action_log_probs = tf.math.log(action_probs)
  actor_loss = -tf.math.reduce_sum(action_log_probs * advantage)

  critic_loss = huber_loss(values, returns)

  return actor_loss + critic_loss
```

In [169]:
```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)


@tf.function
def train_step(
    initial_state: tf.Tensor,
    model: tf.keras.Model,
    optimizer: tf.keras.optimizers.Optimizer,
    gamma: float,
    max_steps_per_episode: int) -> tf.Tensor:
  """Runs a model training step."""

  with tf.GradientTape() as tape:

    # Run the model for one episode to collect training data
    action_probs, values, rewards = run_episode(
        initial_state, model, max_steps_per_episode)

    # Calculate expected returns
    returns = get_expected_return(rewards, gamma)

    # Convert training data to appropriate TF tensor shapes
    action_probs, values, returns = [
        tf.expand_dims(x, 1) for x in [action_probs, values, returns]]

    # Calculating loss values to update our network
    loss = compute_loss(action_probs, values, returns)

  # Compute the gradients from the loss
  grads = tape.gradient(loss, model.trainable_variables)

  # Apply the gradients to the model's parameters
  optimizer.apply_gradients(zip(grads, model.trainable_variables))

  episode_reward = tf.math.reduce_sum(rewards)

  return episode_reward
```

In [ ]:
```
experiments=10
variance_episodic_reward=[]
total_experiment_running_reward=[]
total_steps=[]
for i in range(experiments):
  print("=======================================================Experiment")
  min_episodes_criterion = 100
  max_episodes = 2000
  max_steps_per_episode = 1000
  reward_threshold = 195
  running_reward = 0
  total_episodic_reward=[]
  total_running_reward=np.zeros(max_episodes)
  steps=np.zeros(max_episodes)
  a=0

# Discount factor for future rewards
  gamma = 0.925

# Keep last episodes reward
  episodes_reward: collections.deque = collections.deque(maxlen=min_episodes_criterion)

  with tqdm.trange(max_episodes) as t:
    for j in t:
      a+=1
      initial_state = tf.constant(env.reset(), dtype=tf.float32)
      episode_reward = int(train_step(
         initial_state, model, optimizer, gamma, max_steps_per_episode))

      episodes_reward.append(episode_reward)
      running_reward = statistics.mean(episodes_reward)
      total_episodic_reward.append(episode_reward)
      total_running_reward[j] = running_reward
      #total_experiment_running_reward.append(np.array(total_running_reward))



      t.set_description(f'Episode {j}')
      t.set_postfix(
        episode_reward=episode_reward, running_reward=running_reward)

      steps[j] = a

    # Show average episode reward every 10 episodes
      if i % 10 == 0:
        pass # print(f'Episode {i}: average reward: {avg_reward}')

      if running_reward > reward_threshold and j >= min_episodes_criterion:
         break

  total_experiment_running_reward.append(total_running_reward)
  total_steps.append(steps)

  variance=np.var(total_episodic_reward)
  variance_episodic_reward.append(variance)
  print(f'\n Variance of the {i}th run is{variance}')
  print(f'\nSolved at episode {j}: average reward: {running_reward:.2f}!')
print(variance_episodic_reward)
```

In [171]:
```
p = np.array(total_experiment_running_reward)
s = np.array(total_steps)
variance = np.var(p, axis=0)
rewards = np.mean(p, axis = 0)
steps = np.mean(s, axis = 0)
```

In [ ]:
```
plt.xlabel('Experiments')
plt.ylabel('Variance')
plt.plot(variance)
plt.show()

plt.xlabel('Episodes')
plt.ylabel('Reward')
plt.plot(rewards)
plt.show()

plt.xlabel('Episodes')
plt.ylabel('Steps')
plt.plot(steps)
plt.show()
```

n-step Actor Critic

In [ ]:
```
import collections
import gym
import numpy as np
import statistics
import tensorflow as tf
import tqdm

from matplotlib import pyplot as plt
from tensorflow.keras import layers
env = gym.make("CartPole-v1")

# Set seed for experiment reproducibility
seed = 42
env.seed(seed)
tf.random.set_seed(seed)
np.random.seed(seed)

# Small epsilon value for stabilizing division operations
eps = np.finfo(np.float32).eps.item()
```

In [ ]:
```
from typing import Any, List, Sequence, Tuple
class ActorCritic(tf.keras.Model):
  """Combined actor-critic network."""

  def __init__(
      self,
      num_actions: int,
      num_hidden_units: int):
    """Initialize."""
    super().__init__()

    self.common = layers.Dense(num_hidden_units, activation="relu")
    self.actor = layers.Dense(num_actions)
    self.critic = layers.Dense(1)

  def call(self, inputs: tf.Tensor) -> Tuple[tf.Tensor, tf.Tensor]:
    x = self.common(inputs)
    return self.actor(x), self.critic(x)
```

In [ ]:
```
class ActorCritic(tf.keras.Model):
  """Combined actor-critic network."""

  def __init__(
      self,
      num_actions: int,
      num_hidden_units: int):
    """Initialize."""
    super().__init__()

    self.common = layers.Dense(num_hidden_units, activation="relu")
    self.actor = layers.Dense(num_actions)
    self.critic = layers.Dense(1)

  def call(self, inputs: tf.Tensor) -> Tuple[tf.Tensor, tf.Tensor]:
    x = self.common(inputs)
    return self.actor(x), self.critic(x)
```

In [ ]:
```
num_actions = env.action_space.n  # 2
num_hidden_units = 512

model = ActorCritic(num_actions, num_hidden_units)
```

In [ ]:
```
# Wrap OpenAI Gym's `env.step` call as an operation in a TensorFlow function.
# This would allow it to be included in a callable TensorFlow graph.

def env_step(action: np.ndarray) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
  """Returns state, reward and done flag given an action."""

  state, reward, done, _ = env.step(action)
  return (state.astype(np.float32),
          np.array(reward, np.int32),
          np.array(done, np.int32))


def tf_env_step(action: tf.Tensor) -> List[tf.Tensor]:
  return tf.numpy_function(env_step, [action],
                           [tf.float32, tf.int32, tf.int32])
```

In [ ]:
```
def get_expected_return(
    rewards: tf.Tensor,
    gamma: float,
    standardize: bool = False) -> tf.Tensor:
  """Compute expected returns per timestep."""

  n = tf.shape(rewards)[0]

  returns = tf.TensorArray(dtype=tf.float32, size=n)

  # Start from the end of `rewards` and accumulate reward sums
  # into the `returns` array
  rewards = tf.cast(rewards[::-1], dtype=tf.float32)
  discounted_sum = tf.constant(0.0)
  discounted_sum_shape = discounted_sum.shape
  a1=8
  #for i in tf.range(a1):
  for i in range(0, a1):
    reward = rewards[i]
    discounted_sum = reward + gamma * discounted_sum
    discounted_sum.set_shape(discounted_sum_shape)
    returns = returns.write(i, discounted_sum)
  returns = returns.stack()[::-1]

  if standardize:
    returns = ((returns - tf.math.reduce_mean(returns)) /
               (tf.math.reduce_std(returns) + eps))

  return returns
```

In [ ]:
```
huber_loss = tf.keras.losses.Huber(reduction=tf.keras.losses.Reduction.SUM)

def compute_loss(
    action_probs: tf.Tensor,
    values: tf.Tensor,
    returns: tf.Tensor) -> tf.Tensor:
  """Computes the combined actor-critic loss."""

  advantage = returns - values

  action_log_probs = tf.math.log(action_probs)
  actor_loss = -tf.math.reduce_sum(action_log_probs * advantage)

  critic_loss = huber_loss(values, returns)

  return actor_loss + critic_loss
```

In [ ]:
```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.0005)


@tf.function
def train_step(
    initial_state: tf.Tensor,
    model: tf.keras.Model,
    optimizer: tf.keras.optimizers.Optimizer,
    gamma: float,
    max_steps_per_episode: int) -> tf.Tensor:
  """Runs a model training step."""

  with tf.GradientTape() as tape:

    # Run the model for one episode to collect training data
    action_probs, values, rewards = run_episode(
        initial_state, model, max_steps_per_episode)

    # Calculate expected returns
    returns = get_expected_return(rewards, gamma)

    # Convert training data to appropriate TF tensor shapes
    action_probs, values, returns = [
        tf.expand_dims(x, 1) for x in [action_probs, values, returns]]

    # Calculating loss values to update our network
    loss = compute_loss(action_probs, values, returns)

  # Compute the gradients from the loss
  grads = tape.gradient(loss, model.trainable_variables)

  # Apply the gradients to the model's parameters
  optimizer.apply_gradients(zip(grads, model.trainable_variables))

  episode_reward = tf.math.reduce_sum(rewards)

  return episode_reward
```

In [ ]:
```
%%time

min_episodes_criterion = 100
max_episodes = 10000
max_steps_per_episode = 10000

# Cartpole-v1 is considered solved if average reward is >= 195 over 100
# consecutive trials
reward_threshold = 195
running_reward = 0

# Discount factor for future rewards
gamma = 0.995

# Keep last episodes reward
episodes_reward: collections.deque = collections.deque(maxlen=min_episodes_criterion)

with tqdm.trange(max_episodes) as t:
  for i in t:
    initial_state = tf.constant(env.reset(), dtype=tf.float32)
    episode_reward = int(train_step(
        initial_state, model, optimizer, gamma, max_steps_per_episode))

    episodes_reward.append(episode_reward)
    running_reward = statistics.mean(episodes_reward)

    t.set_description(f'Episode {i}')
    t.set_postfix(
        episode_reward=episode_reward, running_reward=running_reward)

    # Show average episode reward every 10 episodes
    if i % 10 == 0:
      pass # print(f'Episode {i}: average reward: {avg_reward}')

    if running_reward > reward_threshold and i >= min_episodes_criterion:
        break

print(f'\nSolved at episode {i}: average reward: {running_reward:.2f}!')
```

In [ ]:
```
p = np.array(total_experiment_running_reward)
s = np.array(total_steps)
variance = np.var(p, axis=0)
rewards = np.mean(p, axis = 0)
steps = np.mean(s, axis = 0)
```

In [ ]:
```
plt.xlabel('Experiments')
plt.ylabel('Variance')
plt.plot(variance)
plt.show()

plt.xlabel('Episodes')
plt.ylabel('Reward')
plt.plot(rewards)
plt.show()

plt.xlabel('Episodes')
plt.ylabel('Steps')
plt.plot(steps)
plt.show()
```