Introduction

To start gaining knowledge of DeepRL, fundamentals in spinning up course on OpenAI's website were referred. Then, as per the order suggested in the tutorial, algorithms **Simple Policy Gradient (PG), Vanilla Policy Gradient (VPG)** and **DQN (Deep Q Network)** were understood and implemented using Pytorch by referring few articles and web links given in references section

Explanation for each algorithm is as follows:

A. Simple Policy Gradient

Actions done:

1. Traced, understood the algorithm and code
2. Noted down the pseudocode
3. Implemented the algorithm in Pytroch for the 'CartPole-v0' environment from openAI gym , confirmed the problem is solved (Batch Average reward > 195 for 10 consecutive episodes) Hypeparameters are same as that used in the original implementation
4. Understood the usage of MPI utils and used them to parallelize the code

Explanation:

In RL, policy is the rule by which the agent decides the action to be taken for a given state input

(Actions are executed by the agent in the environment and it receives rewards and state/observation from the environment)

For the right action, a positive reward is given and negative reward is given for wrong action. The goal of the agent is to maximize the total reward value by taking right actions consistently . Policy (generally implemented by neural network) has to be optimized using the gradients so as right action is predicted by it .
The action value is used by the agent to act in the environment and receives rewards.

The expectation of reward or average reward has to be maximized to solve the problem

$$J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau \sim \pi_\theta} [R(\tau)]$$

Derivative of above expression $\nabla_\theta J(\pi_\theta)$, is the policy gradient .

The end expression of the policy gradient is

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau),$$

which is used for implementation

D = the number of trajectories , T = number of the timesteps in a trajectory or episode

An episode is a sequence of states and actions (in one time step there is one state and one action) . That sequence is ended by an episode terminating condition. Each RL environment has its own terminating conditions. It is similar to 'Game over' condition where the game cannot be continued , rather it has to be restarted from the beginning

$\pi_\theta(a_t|s_t)$ is the prediction of the action given the input state by the probabilistic policy implemented by the neural network

$R(\tau)$ is the total reward for the trajectory

Implementation details:

1. **Multi-layer perceptron**: A neural network with below dimensions is implemented
   one Input layer : 4 Nodes (1 for each observation)
   one hidden layer : 32 Nodes
   One output layer : 2 nodes

   Except the last layer, every layer has Non-Linear Tanh activation . Last layer has nn.Identity which is just forwarding inputs to outputs . These outputs are passed to Categorical Distribution so as to form a stochastic policy

2. **Training :**
   1 . mpi_fork(4) is used to divide main process into 4 processes
   2. setup_pytorch_for_mpi() is called to avoid conflicts due to usage of both Pytorch and OpenMPI
   3. Seed is initialized used to create same random weights for subsequent runs of the program
   4. Batch size is divided into 4 mini batches, one per one process
   5. sync_params(logits_nw) is called to synchronize the neural network parameters across all 4 processes
   6. Action value is sampled from the categorical distribution formed using the logits of the neural network implemented
   7. Loss function implementation:
      a. weights = R(Tau) i.e. total reward for single episode replicated for number of times that is equal to episode length
      b. Log probabilities are used from the Categorical distribution formed
      c. Mean is returned since total value is averaged over total number of timesteps in the batch. Each episode might have different timesteps .Hence ,two one dimensional vectors ae multiplied ,summed and averaged by total timesteps in all trajectories
      Here, sum is divided by DT but in expression it is divided by D (T = num of time steps in trajectory , D = num of trajectories)
      Inferred from discussuion in Link 3 of references that additional division by T in implementation would not effect the performance
      d. loss value is returned with negative sign before , since the gradients computed will have negative sign . This helps Adam optimizer to do gradient ascent instead of gradient descent . Gradient ascent is required for Policy gradient algorithm
   8. Other Hyper params used are: Learning rate = 1e-2,epochs = 50,batchsize = 5000
   9. Until the batch size is less than the minibatch size hyperparameter , acting in the gym

environment, collecting the rewards , summing them per episode is continued

10 . Loss is computed for batch and adam optimizer is used

11. mpi_avg_grads(logits_nw) is used to average the gradients from all the processes in every process . These averaged gradients are used to optimize the neural network in every process

12. Training process is repeated for 50 epochs

Results:

The output data are printed for process with Rank 0 which is as below given by screenshots . It is observed that in epoch 42 and 45 , average reward(for mini batch size 1250) is greater than 195

```
(spinningup) jayaram@jayaram-Inspiron-3521:~$ python MPI_SimplePolicyGradient.py  --env_name 'CartPole-v0' --lr 1e-2
epoch:  0        loss:19.002     return:20.484    episode_length:20.484
epoch:  1        loss:17.322     return:20.323    episode_length:20.323
epoch:  2        loss:21.080     return:24.358    episode_length:24.358
epoch:  3        loss:23.654     return:26.327    episode_length:26.327
epoch:  4        loss:31.863     return:36.257    episode_length:36.257
epoch:  5        loss:31.023     return:35.111    episode_length:35.111
epoch:  6        loss:26.521     return:32.146    episode_length:32.146
epoch:  7        loss:33.719     return:41.355    episode_length:41.355
epoch:  8        loss:42.112     return:53.083    episode_length:53.083
epoch:  9        loss:52.214     return:60.476    episode_length:60.476
epoch: 10        loss:47.880     return:63.190    episode_length:63.190
epoch: 11        loss:39.555     return:55.304    episode_length:55.304
epoch: 12        loss:55.257     return:68.211    episode_length:68.211
epoch: 13        loss:44.842     return:62.810    episode_length:62.810
epoch: 14        loss:50.898     return:72.389    episode_length:72.389
epoch: 15        loss:44.223     return:67.526    episode_length:67.526
epoch: 16        loss:77.464     return:99.231    episode_length:99.231
epoch: 17        loss:55.489     return:74.444    episode_length:74.444
epoch: 18        loss:73.532     return:99.462    episode_length:99.462
epoch: 19        loss:83.018     return:120.545   episode_length:120.545
epoch: 20        loss:97.284     return:157.875   episode_length:157.875
epoch: 21        loss:79.029     return:131.400   episode_length:131.400
epoch: 22        loss:74.023     return:111.333   episode_length:111.333
epoch: 23        loss:90.187     return:144.444   episode_length:144.444
epoch: 24        loss:111.393    return:190.857   episode_length:190.857
epoch: 25        loss:103.424    return:173.875   episode_length:173.875
epoch: 26        loss:104.280    return:166.750   episode_length:166.750
epoch: 27        loss:101.776    return:175.750   episode_length:175.750
epoch: 28        loss:89.620     return:144.222   episode_length:144.222
epoch: 29        loss:101.050    return:169.125   episode_length:169.125
epoch: 30        loss:97.717     return:164.375   episode_length:164.375
epoch: 31        loss:85.607     return:129.300   episode_length:129.300
epoch: 32        loss:94.646     return:146.556   episode_length:146.556
epoch: 33        loss:75.543     return:119.583   episode_length:119.583
epoch: 34        loss:86.387     return:139.900   episode_length:139.900
epoch: 35        loss:95.532     return:163.375   episode_length:163.375
epoch: 36        loss:103.063    return:173.875   episode_length:173.875
epoch: 37        loss:104.917    return:184.714   episode_length:184.714
epoch: 38        loss:96.757     return:161.000   episode_length:161.000
epoch: 39        loss:103.747    return:179.875   episode_length:179.875
epoch: 40        loss:100.149    return:166.375   episode_length:166.375
epoch: 41        loss:106.252    return:192.714   episode_length:192.714
epoch: 42        loss:114.405    return:199.714   episode_length:199.714
epoch: 43        loss:103.789    return:183.714   episode_length:183.714
epoch: 44        loss:109.999    return:198.714   episode_length:198.714
epoch: 45        loss:111.625    return:200.000   episode_length:200.000
epoch: 46        loss:108.139    return:191.714   episode_length:191.714
epoch: 47        loss:108.326    return:190.143   episode_length:190.143
epoch: 48        loss:108.251    return:194.000   episode_length:194.000
/home/jayaram/anaconda3/envs/spinningup/lib/python3.6/site-packages/gym/logger.py:30: UserWarning: WARN: Box bound
g to float32
  warnings.warn(colorize('%s: %s'%('WARN', msg % args), 'yellow'))
/home/jayaram/anaconda3/envs/spinningup/lib/python3.6/site-packages/gym/logger.py:30: UserWarning: WARN: Box bound
g to float32
  warnings.warn(colorize('%s: %s'%('WARN', msg % args), 'yellow'))
/home/jayaram/anaconda3/envs/spinningup/lib/python3.6/site-packages/gym/logger.py:30: UserWarning: WARN: Box bound
g to float32
  warnings.warn(colorize('%s: %s'%('WARN', msg % args), 'yellow'))
  warnings.warn(colorize('%s: %s'%('WARN', msg % args), 'yellow'))
epoch: 49        loss:110.765    return:192.714   episode_length:192.714
/home/jayaram/anaconda3/envs/spinningup/lib/python3.6/site-packages/gym/logger.py:30: UserWarning:
g to float32
  warnings.warn(colorize('%s: %s'%('WARN', msg % args), 'yellow'))
(spinningup) jayaram@jayaram-Inspiron-3521:~$
```

B.<u>Vanilla Policy Gradient</u>

Actions done:

1. Traced and understood the algorithm and code
2. Noted down the pseudocode
3. Implemented the algorithm in Pytroch for the 'CartPole-v0' environment from openAI gym , Hypeparameters are same as that used in the original implementation
4. Parallelization is not done unlike in original code

Explanation:

Article in Link 4 in References section is used for understanding the algorithm. In this algorithm the gradient for policy is computed in a similar manner as in Policy Gradient algorithm except for the fact that instead of weights or total reward for trajectory: R(tau) , Advantage value for one timestep : At is used

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

Gradient ascent algorithm is used to update the policy (similar as in Policy Gradient)

A value function is implemented by another neural network to predict action value. Gradient descent is used to optimize the value function

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2,$$

$V_\phi(s_t)$ = action values predicted by the action value function

$\hat{R}_t$ = Cumulative sum of discounted rewards calculated at the end of episode or at the end of trajectory cut off . These values are stored in reverse order in the buffer

Following is the Generalized Advantage Estimation used to estimate the advantage values in the loss function mentioned above

Deltas are calculated as below . The cumulative sum of discounted (discount = gamma * lambda) deltas gives the Advantages which are stored in reverse order in the buffer

```
''' Calulation of deltas
 deltaVt = rt +gamma*V(st+1) -V(st)
 deltaVt+1 = rt+1 +gamma*V(st+2) -V(st+1)
 .
 .
```

'''

In the above equations rt = reward in each timestep   V(St), V(St+1) are the values predicted by the Value function

Implementation details:

Multi-layer perceptron for Policy –Actor :   A neural network with below dimensions is implemented

one Input layer : 4 Nodes (1 for each observation)
one hidden layer : 64 Nodes
one hidden layer : 64 Nodes
one output layer : 2 nodes

Except the last layer, every layer has Non-Linear Tanh activation . Last layer has nn.Identity which is just forwarding inputs to outputs .

Multi-layer perceptron for Value function – Critic :   A neural network with below dimensions is implemented

one Input layer : 4 Nodes (1 for each observation)
one hidden layer : 64 Nodes
one hidden layer : 64 Nodes
one output layer : 1 node

Except the last layer, every layer has Non-Linear Tanh activation . Last layer has nn.Identity which is just forwarding inputs to outputs .

Discount cum sum calculation(used for the advantage and return calculation):

A function is implemented which takes input:
vector x, [x0, x1, x2] as input
and returns output:
[x0 + discount * x1 + discount^2 * x2,
x1 + discount * x2,
x2]

VPG Buffer:

It sores state or observations, actions, advantages ,rewards , return, action values and log probabilities for action.

When the trajectory completes (episode ends) or steps complete or epoch ends(trajectory cuts off) , the advantages and return values are calcutated and stored for the length of the episode and saved

Reason to use is that the lengths of trajectories are not constant and it makes easy to look back and calculate

Training :

1. Hyper parameters are as follows

steps per epoch = 4000, epochs = 50,
gamma = 0.99,learning rate of policy = 3e-4, learning rate of action value function = 1e-3,
number of iterations to step for value function training = 80, lambda = 0.97,
maximum length of episode = 1000

2. The policy network is used to predict the action value
3. Agent acts in environment with the obtained action
4. Values are stored in the Buffer for every timestep
5. When the terminating condition is reached, the values are looked back from the starting point of trajectory till end . Deltas , advantages and return (discounted cumulative sum) is calculated

Terminating condition is
Episode ended or episode length == maximum episode length or last step of the epoch

Trajectory will be cut off if the Epsiode did not end , timeout(episode length == maximum episode length) did not happened but it is last iteration of the epoch

6. Loss function is calculated of both policy network and action value function and the networks are optimized for one step . For Value function , optimization is done for 'number of iterations to step for value function training' . Adam optimizer is used for both of them
7. The whole process is repeated for number of epochs

Results: It can be observed in the appendix below that Average return of few epochs is greater than the average return of the final episode in the original code . Also , maximum episode return is 200 in epochs 30,34,49

C. <u>Deep Q Network</u>

<u>Actions done:</u>

1. Traced and understood the algorithm and most of the code (tensorflow) given in links 5,6,7
2. Implemented the algorithm in Pytroch for the 'CartPole-v0' environment from openAI gym , and used the same hyperparameters as in links 5,6,7
3. The episode return is not more than 70 or 80 even after all episodes are completed .
4. Then used the hyper- parameters and implementation tricks that are present in the links 8,9

Explanation :

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Above algorithm is from link 5

1. In Deep Q training, Replay memory buffer is used to store the state, action reward, next state. Its size is a hyper parameter

2. Two neural networks of same architecture, prediction network and target network are used to implement two separate state action value functions. The difference between them is that one takes current state as input while another takes next state as input

3. The action value is predicted by the state action value function. Action which has the maximum q value in the prediction network is selected for the action or otherwise a random action value from the action space is returned based on the value of the epsilon . This is called epsilon greedy method of choosing action

4. If there are sufficient size of values(batch size) stored in the replay memory buffer , then they can be used to compute loss function . The loss function is the mean squared error value which takes the predicted network output and target network output

5. Only Prediction network is optimized and the parameters are deep copied to Target network for every few number of steps which is a hyperparameter

Implementaion details:

Multi-layer perceptron for Prediction network : A neural network with below dimensions is implemented

one Input layer : 4 Nodes (1 for each observation)
one hidden layer : 64 Nodes
one output layer : 2 nodes

Except the last layer, every layer has Non-Linear Tanh activation . Last layer has nn.Identity which is just forwarding inputs to outputs .

Multi-layer perceptron for Target network **:** Same architecture as for the prediction network

Replay memory: Named tuples for (state, action reward, next state,done flag) are created for every time step and stored in the deque . if the size is >=256 old tuple is popped out and new one is appended

Batch size of 16 is randomly sampled from this replay memory for calculating the loss function

Training

1. Hyperparameters

gamma for the bellman update = 0.95

replaymemorycapacity = 256
batch_size for sampling from replay memory = 16

hidden_sizes = [64]
learningrate = 1e-3

epsilon = 1.0
target_nw_update_frequency = 5
n_episodes = 5000
seed = 1423

2. Adam optimizer is used
3. Action value from prediction network is used or a random value from action space (epsilon greedy method is used)
4. Agent acts the environment and output values are stored in the replay memory
5. Once the batch of values is available , loss is computed ,
6. In the compute loss function , initially target network is updated based on the target_nw_update_frequency parameter . Q(s,a) value from the prediction network and target action networks are retrieved . The yj value is calculated as

yj = sampledrewards+ (1-sampleddoneflags)*gamma*max targetnet q value calculated using sampled_nextstate

and mean square loss is calculated using yj and predicted q value

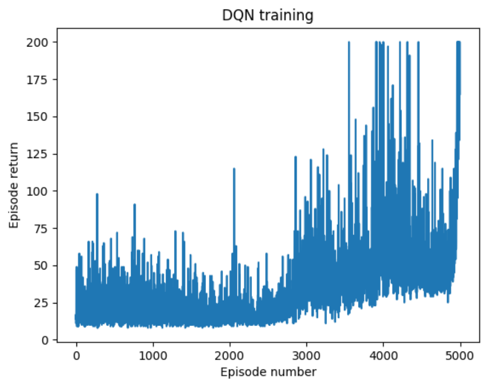
7. Using this loss, prediction network is optimized

8. The training process is repeated for the number of episodes (5000)

Note : Before actual training, replaly memory buffer of size 256 is completely filled using random exploration by the agent . Prediction network is not trained as the loss function and optimization are not done for the 256 iterations

Problem found during training: When torch.as_tensor(state,dtype =torch.float32) is used as input the prediction network , the episode values are not increasing . Had to use torch.from_numpy(state).float() which is a 64 bit value to get right episode returns

Results : After around 4000 episodes(1st instance) and before completion of 5000 episodes (2nd instance) , the episode return is constant value 200 for consecutive episode length greater than 10 . Thus it can be considered that cartpole problem is solved



References :

1. https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py
2. https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html#part-3-intro-to-policy-optimization
3. https://github.com/openai/spinningup/issues/59
4. https://spinningup.openai.com/en/latest/algorithms/vpg.html
5. https://lilianweng.github.io/lil-log/2018/02/19/a-long-peek-into-reinforcement-learning.html#deep-q-network
6. https://lilianweng.github.io/lil-log/2018/05/05/implementing-deep-reinforcement-learning-models.html#deep-q-network
7. https://github.com/lilianweng/deep-reinforcement-learning-gym/blob/master/playground/policies/dqn.py
8. https://blog.gofynd.com/building-a-deep-q-network-in-pytorch-fa1086aa5435
9. https://github.com/mahakal001/reinforcement-learning/tree/master/cartpole-dqn

APPENDIX :

Output of VPG end episode in the original implementation

```
Episode 80       EpRet 30.000    EpLen 30
Episode 81       EpRet 42.000    EpLen 42
Episode 82       EpRet 28.000    EpLen 28
Episode 83       EpRet 35.000    EpLen 35
Episode 84       EpRet 36.000    EpLen 36
Episode 85       EpRet 35.000    EpLen 35
Episode 86       EpRet 49.000    EpLen 49
Episode 87       EpRet 48.000    EpLen 48
Episode 88       EpRet 71.000    EpLen 71
Episode 89       EpRet 67.000    EpLen 67
Episode 90       EpRet 36.000    EpLen 36
Episode 91       EpRet 17.000    EpLen 17
Episode 92       EpRet 103.000   EpLen 103
Episode 93       EpRet 59.000    EpLen 59
Episode 94       EpRet 102.000   EpLen 102
Episode 95       EpRet 33.000    EpLen 33
Episode 96       EpRet 18.000    EpLen 18
Episode 97       EpRet 32.000    EpLen 32
Episode 98       EpRet 29.000    EpLen 29
Episode 99       EpRet 16.000    EpLen 16
-------------------------------------
|    AverageEpRet |            37.3 |
|        StdEpRet |            20.2 |
|        MaxEpRet |             109 |
|        MinEpRet |              12 |
|           EpLen |            37.3 |
-------------------------------------
```

Output of VPG implementation:

```
(spinningup) jayaram@jayaram-Inspiron-3521:~$ python
SingleThreaded_VanillaPolicyGradient.py --env 'CartPole-v0' --seed 0
/home/jayaram/anaconda3/envs/spinningup/lib/python3.6/site-packages/gym/logger.py:30:
UserWarning: WARN: Box bound precision lowered by casting to float32
  warnings.warn(colorize('%s: %s'%('WARN', msg % args), 'yellow'))
Trajectory cut off by epoch at 18 steps
Epoch:0    BatchLoss_Pi :0.007846    BatchLoss_V :122.357323
AverageBatchReturn:22.222    AverageBatchEpisodeLen:22.222    Max_episode_return: 73.000
Trajectory cut off by epoch at 27 steps
Epoch:1    BatchLoss_Pi :0.003028    BatchLoss_V :56.488045
AverageBatchReturn:21.622    AverageBatchEpisodeLen:21.622    Max_episode_return: 61.000
Trajectory cut off by epoch at 13 steps
Epoch:2    BatchLoss_Pi :0.004312    BatchLoss_V :75.874138
AverageBatchReturn:22.222    AverageBatchEpisodeLen:22.222    Max_episode_return: 77.000
Trajectory cut off by epoch at 23 steps
Epoch:3    BatchLoss_Pi :0.002683    BatchLoss_V :61.030365
AverageBatchReturn:21.277    AverageBatchEpisodeLen:21.277    Max_episode_return: 67.000
Trajectory cut off by epoch at 3 steps
Epoch:4    BatchLoss_Pi :-0.001101    BatchLoss_V :94.463219
AverageBatchReturn:21.739    AverageBatchEpisodeLen:21.739    Max_episode_return: 89.000
Epoch:5    BatchLoss_Pi :-0.001243    BatchLoss_V :67.546295
AverageBatchReturn:22.222    AverageBatchEpisodeLen:22.222    Max_episode_return: 81.000
Trajectory cut off by epoch at 12 steps
Epoch:6    BatchLoss_Pi :-0.001570    BatchLoss_V :146.718460
AverageBatchReturn:25.316    AverageBatchEpisodeLen:25.316    Max_episode_return:
163.000
Trajectory cut off by epoch at 25 steps
Epoch:7    BatchLoss_Pi :-0.004588    BatchLoss_V :69.507866
AverageBatchReturn:24.096    AverageBatchEpisodeLen:24.096    Max_episode_return: 74.000
Trajectory cut off by epoch at 7 steps
Epoch:8    BatchLoss_Pi :-0.008956    BatchLoss_V :92.899895
AverageBatchReturn:24.845    AverageBatchEpisodeLen:24.845    Max_episode_return:
103.000
Trajectory cut off by epoch at 12 steps
Epoch:9    BatchLoss_Pi :-0.011398    BatchLoss_V :71.129662
AverageBatchReturn:23.392    AverageBatchEpisodeLen:23.392    Max_episode_return: 91.000
Trajectory cut off by epoch at 39 steps
Epoch:10    BatchLoss_Pi :-0.014520    BatchLoss_V :80.243759
AverageBatchReturn:25.000    AverageBatchEpisodeLen:25.000    Max_episode_return: 92.000
Trajectory cut off by epoch at 36 steps
```

Epoch:11 BatchLoss_Pi :-0.014474 BatchLoss_V :95.814285
AverageBatchReturn:26.667 AverageBatchEpisodeLen:26.667 Max_episode_return: 131.000
Trajectory cut off by epoch at 3 steps
Epoch:12 BatchLoss_Pi :-0.017533 BatchLoss_V :88.703331
AverageBatchReturn:25.641 AverageBatchEpisodeLen:25.641 Max_episode_return: 104.000
Trajectory cut off by epoch at 8 steps
Epoch:13 BatchLoss_Pi :-0.017020 BatchLoss_V :101.320930
AverageBatchReturn:26.667 AverageBatchEpisodeLen:26.667 Max_episode_return: 105.000
Trajectory cut off by epoch at 57 steps
Epoch:14 BatchLoss_Pi :-0.021664 BatchLoss_V :71.943832
AverageBatchReturn:25.641 AverageBatchEpisodeLen:25.641 Max_episode_return: 97.000
Trajectory cut off by epoch at 9 steps
Epoch:15 BatchLoss_Pi :-0.024604 BatchLoss_V :56.053185
AverageBatchReturn:26.316 AverageBatchEpisodeLen:26.316 Max_episode_return: 78.000
Trajectory cut off by epoch at 79 steps
Epoch:16 BatchLoss_Pi :-0.026022 BatchLoss_V :64.244087
AverageBatchReturn:26.667 AverageBatchEpisodeLen:26.667 Max_episode_return: 79.000
Epoch:17 BatchLoss_Pi :-0.028230 BatchLoss_V :60.220501
AverageBatchReturn:27.397 AverageBatchEpisodeLen:27.397 Max_episode_return: 83.000
Trajectory cut off by epoch at 41 steps
Epoch:18 BatchLoss_Pi :-0.026421 BatchLoss_V :113.842346
AverageBatchReturn:29.630 AverageBatchEpisodeLen:29.630 Max_episode_return: 152.000
Trajectory cut off by epoch at 32 steps
Epoch:19 BatchLoss_Pi :-0.026783 BatchLoss_V :129.379654
AverageBatchReturn:32.000 AverageBatchEpisodeLen:32.000 Max_episode_return: 118.000
Trajectory cut off by epoch at 15 steps
Epoch:20 BatchLoss_Pi :-0.034401 BatchLoss_V :80.090324
AverageBatchReturn:27.397 AverageBatchEpisodeLen:27.397 Max_episode_return: 112.000
Trajectory cut off by epoch at 22 steps
Epoch:21 BatchLoss_Pi :-0.029290 BatchLoss_V :95.608253
AverageBatchReturn:30.303 AverageBatchEpisodeLen:30.303 Max_episode_return: 97.000
Trajectory cut off by epoch at 10 steps
Epoch:22 BatchLoss_Pi :-0.033593 BatchLoss_V :108.552696
AverageBatchReturn:30.303 AverageBatchEpisodeLen:30.303 Max_episode_return: 113.000
Trajectory cut off by epoch at 11 steps
Epoch:23 BatchLoss_Pi :-0.035480 BatchLoss_V :102.868874
AverageBatchReturn:31.496 AverageBatchEpisodeLen:31.496 Max_episode_return: 96.000
Trajectory cut off by epoch at 56 steps

Epoch:24 BatchLoss_Pi :-0.033929 BatchLoss_V :103.283920
AverageBatchReturn:34.783 AverageBatchEpisodeLen:34.783 Max_episode_return: 92.000
Trajectory cut off by epoch at 13 steps
Epoch:25 BatchLoss_Pi :-0.039758 BatchLoss_V :86.334396
AverageBatchReturn:32.258 AverageBatchEpisodeLen:32.258 Max_episode_return: 89.000
Trajectory cut off by epoch at 35 steps
Epoch:26 BatchLoss_Pi :-0.039852 BatchLoss_V :119.893684
AverageBatchReturn:31.496 AverageBatchEpisodeLen:31.496 Max_episode_return:
118.000
Trajectory cut off by epoch at 26 steps
Epoch:27 BatchLoss_Pi :-0.033178 BatchLoss_V :206.331284
AverageBatchReturn:37.037 AverageBatchEpisodeLen:37.037 Max_episode_return:
178.000
Trajectory cut off by epoch at 21 steps
Epoch:28 BatchLoss_Pi :-0.040517 BatchLoss_V :92.742645
AverageBatchReturn:33.333 AverageBatchEpisodeLen:33.333 Max_episode_return:
107.000
Trajectory cut off by epoch at 8 steps
Epoch:29 BatchLoss_Pi :-0.039011 BatchLoss_V :111.395767
AverageBatchReturn:32.258 AverageBatchEpisodeLen:32.258 Max_episode_return:
124.000
Trajectory cut off by epoch at 30 steps
Epoch:30 BatchLoss_Pi :-0.040287 BatchLoss_V :157.422379
AverageBatchReturn:33.898 AverageBatchEpisodeLen:33.898 Max_episode_return:
200.000
Trajectory cut off by epoch at 21 steps
Epoch:31 BatchLoss_Pi :-0.042987 BatchLoss_V :83.450996
AverageBatchReturn:33.333 AverageBatchEpisodeLen:33.333 Max_episode_return: 91.000
Trajectory cut off by epoch at 27 steps
Epoch:32 BatchLoss_Pi :-0.044457 BatchLoss_V :109.882782
AverageBatchReturn:36.036 AverageBatchEpisodeLen:36.036 Max_episode_return:
106.000
Trajectory cut off by epoch at 7 steps
Epoch:33 BatchLoss_Pi :-0.038965 BatchLoss_V :208.650955
AverageBatchReturn:40.816 AverageBatchEpisodeLen:40.816 Max_episode_return:
182.000
Trajectory cut off by epoch at 45 steps
Epoch:34 BatchLoss_Pi :-0.042733 BatchLoss_V :157.462341
AverageBatchReturn:35.088 AverageBatchEpisodeLen:35.088 Max_episode_return:
200.000
Trajectory cut off by epoch at 30 steps
Epoch:35 BatchLoss_Pi :-0.046928 BatchLoss_V :120.962708
AverageBatchReturn:36.364 AverageBatchEpisodeLen:36.364 Max_episode_return:
135.000
Trajectory cut off by epoch at 29 steps

Epoch:36 BatchLoss_Pi :-0.039468 BatchLoss_V :156.014999
AverageBatchReturn:43.478 AverageBatchEpisodeLen:43.478 Max_episode_return: 121.000
Trajectory cut off by epoch at 114 steps
Epoch:37 BatchLoss_Pi :-0.040862 BatchLoss_V :155.483582
AverageBatchReturn:40.816 AverageBatchEpisodeLen:40.816 Max_episode_return: 136.000
Trajectory cut off by epoch at 24 steps
Epoch:38 BatchLoss_Pi :-0.035497 BatchLoss_V :155.799088
AverageBatchReturn:43.011 AverageBatchEpisodeLen:43.011 Max_episode_return: 127.000
Trajectory cut off by epoch at 46 steps
Epoch:39 BatchLoss_Pi :-0.039703 BatchLoss_V :155.850281
AverageBatchReturn:41.237 AverageBatchEpisodeLen:41.237 Max_episode_return: 146.000
Trajectory cut off by epoch at 33 steps
Epoch:40 BatchLoss_Pi :-0.043442 BatchLoss_V :127.975906
AverageBatchReturn:40.816 AverageBatchEpisodeLen:40.816 Max_episode_return: 135.000
Trajectory cut off by epoch at 12 steps
Epoch:41 BatchLoss_Pi :-0.044263 BatchLoss_V :119.810371
AverageBatchReturn:43.011 AverageBatchEpisodeLen:43.011 Max_episode_return: 105.000
Trajectory cut off by epoch at 12 steps
Epoch:42 BatchLoss_Pi :-0.046383 BatchLoss_V :105.436058
AverageBatchReturn:41.237 AverageBatchEpisodeLen:41.237 Max_episode_return: 108.000
Trajectory cut off by epoch at 22 steps
Epoch:43 BatchLoss_Pi :-0.038583 BatchLoss_V :188.466507
AverageBatchReturn:43.956 AverageBatchEpisodeLen:43.956 Max_episode_return: 183.000
Trajectory cut off by epoch at 23 steps
Epoch:44 BatchLoss_Pi :-0.049172 BatchLoss_V :148.792358
AverageBatchReturn:43.956 AverageBatchEpisodeLen:43.956 Max_episode_return: 133.000
Trajectory cut off by epoch at 67 steps
Epoch:45 BatchLoss_Pi :-0.050614 BatchLoss_V :158.116806
AverageBatchReturn:44.444 AverageBatchEpisodeLen:44.444 Max_episode_return: 153.000
Trajectory cut off by epoch at 25 steps
Epoch:46 BatchLoss_Pi :-0.056723 BatchLoss_V :111.397614
AverageBatchReturn:41.667 AverageBatchEpisodeLen:41.667 Max_episode_return: 101.000
Trajectory cut off by epoch at 35 steps

```
Epoch:47	BatchLoss_Pi :-0.048568	BatchLoss_V :96.069359
AverageBatchReturn:44.444	AverageBatchEpisodeLen:44.444	Max_episode_return:
104.000
Trajectory cut off by epoch at 14 steps
Epoch:48	BatchLoss_Pi :-0.037875	BatchLoss_V :133.030396
AverageBatchReturn:46.512	AverageBatchEpisodeLen:46.512	Max_episode_return:
132.000
Trajectory cut off by epoch at 51 steps
Epoch:49	BatchLoss_Pi :-0.041197	BatchLoss_V :198.017532
AverageBatchReturn:47.059	AverageBatchEpisodeLen:47.059	Max_episode_return:
200.000
(spinningup) jayaram@jayaram-Inspiron-3521:~$
```