CartPole game –

A well-known control issue in reinforcement learning is the CartPole game. The object of the game is to balance a pole on a cart. The object of the game is to keep the pole upright for as long as you can while the cart moves left and right. When the pole can be balanced for 500 timesteps or more, the game is deemed solved.

Because the CartPole game is easy to implement but difficult to solve, it is a popular testbed for algorithms utilizing reinforcement learning. The fundamentals of reinforcement learning, including the ideas of rewards, states, and actions, are also taught through the game.

The OpenAI Gym is a toolkit for creating and evaluating reinforcement learning algorithms, and it includes the CartPole game. The Gym offers a range of environments for training and assessing reinforcement learning agents, one of which is CartPole..

Here are some of the challenges of solving the CartPole game:

- The pole is shaky and readily topples.
- Because the cart moves continuously, it is challenging to control.
- For the game to be successful, a long-term plan is necessary.

Multiple reinforcement learning algorithms have managed to solve the CartPole game in spite of these difficulties. Deep Q-Network (DQN), a well-liked deep learning algorithm, has demonstrated remarkable efficacy in resolving reinforcement learning issues.

Imports:

```
import gym, os from itertools import count import torch import torch.nn as nn import torch.optim as optim import torch.nn.functional as F from torch.distributions import Categorical import matplotlib.pyplot as plt
```

Environment:

We are using open AI gym environment.

Critic:

Using a three-layered neural network, a critic was created. It adjusts for the number of intermediate nodes by hand.

```
class Critic(nn.Module):     def __init__(self,
state_size, action_size):
        super(Critic, self).__init__()
self.state_size = state_size         self.action_size
= action_size         self.linear1 =
nn.Linear(self.state_size, 128)         self.linear2 =
nn.Linear(128, 256)         self.linear3 =
nn.Linear(256, 1)
```

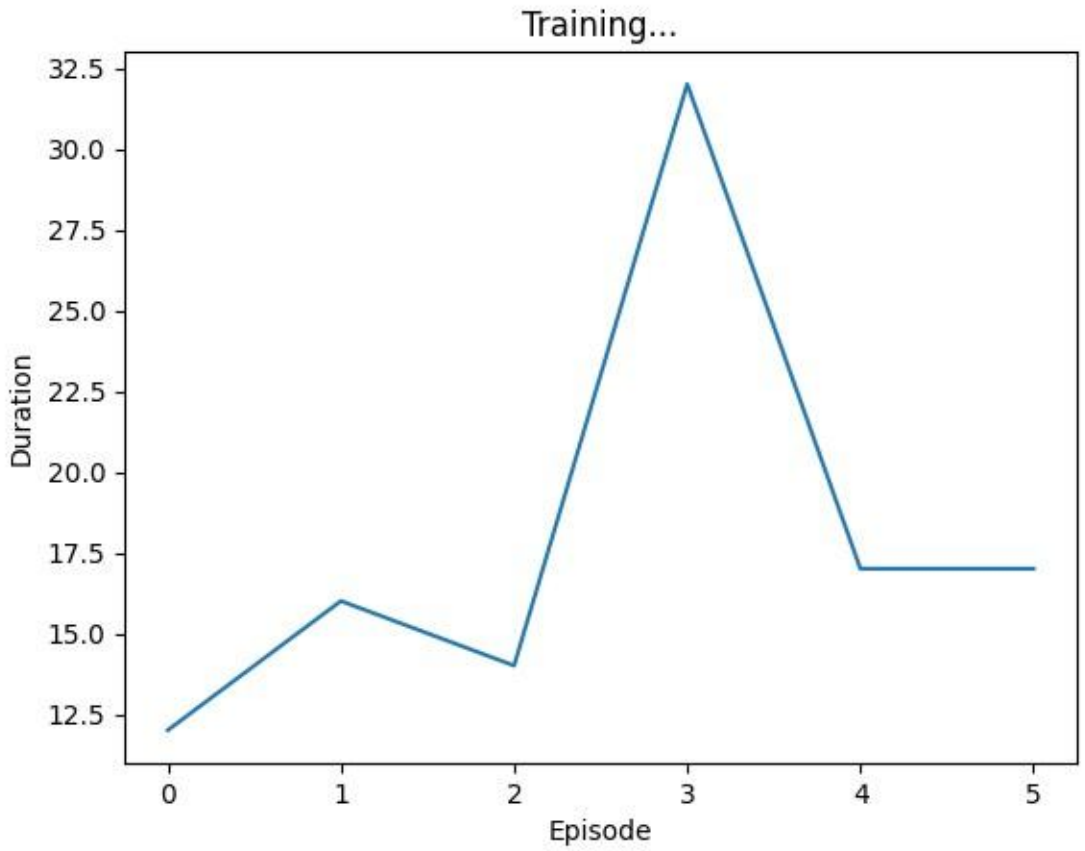
Actor:

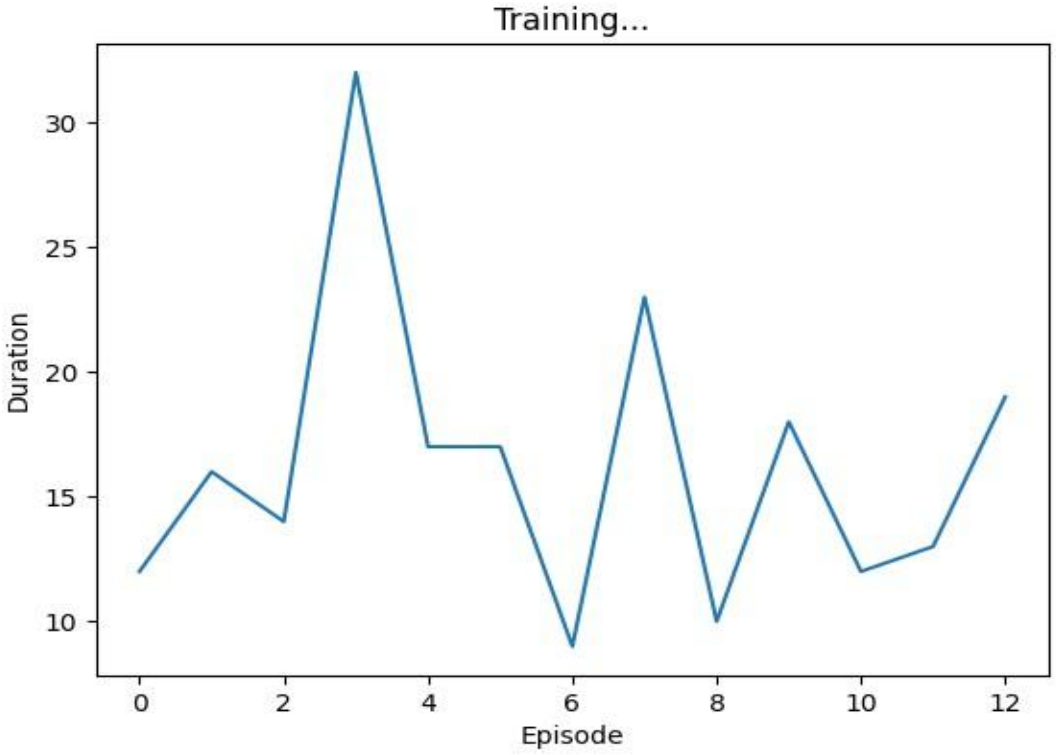
```
class
Actor(nn.Module):
    def __init__(self, state_size, action_size):
        super(Actor, self).__init__()
self.state_size = state_size         self.action_size
= action_size         self.linear1 =
nn.Linear(self.state_size, 128)         self.linear2 =
nn.Linear(128, 256)
        self.linear3 = nn.Linear(256, self.action_size)
     def forward(self,
state):
        output = F.relu(self.linear1(state))
output = F.relu(self.linear2(output))
output = self.linear3(output)
        distribution = Categorical(F.softmax(output, dim=-1))
return distribution
```

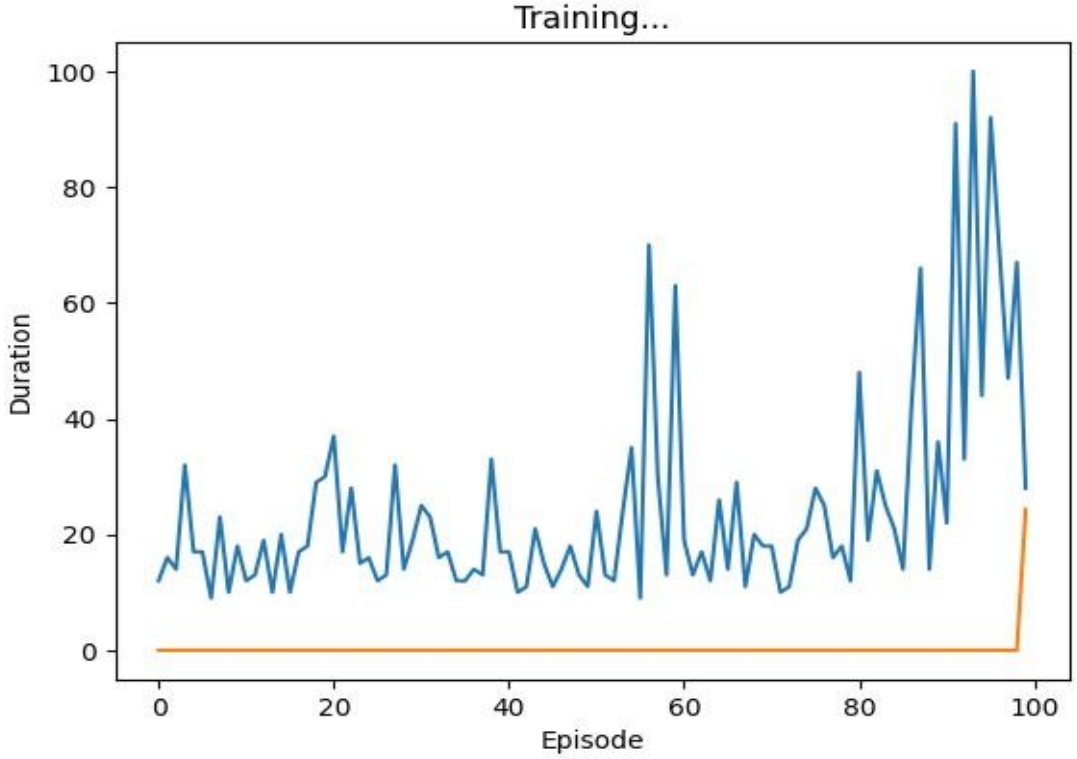
```
actor = Actor(state_size, action_size).to(device) critic
= Critic(state_size, action_size).to(device)
trainIters(actor, critic, n_iters=100)
```

```
# Plot duration curve:
# From
http://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html
episode_durations = [] def plot_durations():
    plt.figure(2)     plt.clf()     durations_t =
torch.FloatTensor(episode_durations)
plt.title('Training...')     plt.xlabel('Episode')
plt.ylabel('Duration')
plt.plot(durations_t.numpy())
    # Take 100 episode averages and plot them too
if len(durations_t) >= 100:
        means = durations_t.unfold(0, 100, 1).mean(1).view(-1)
means = torch.cat((torch.zeros(99), means))
plt.plot(means.numpy())
     plt.pause(0.001)  # pause a bit so that plots are updated
```

Graph plot:




Training...
Duration
Episode
30
25
20
15
10
0
2
4
6
8
10
12


Training...
Duration
Episode
100
80
60
40
20
0
0
20
40
60
80
100