1. Introduction

As the action space of cartMountain-v0 is discrete, I used DQN to solve this problem. DQN is a reinforcement learning algorithm, which applies a neural network to q learning. Before DQN, researchers used q-table to represent q-function, which cost a lot of memory for a large environment. Meanwhile, neural networks show good performance as a function approximator, so it is natural to use neural networks to represent q-function instead of q-table. However it wasn't as easy as it looked. There were two problems.

1. High correlation between samples

NN is effective when distribution of train data is i.i.d. However, samples in the same trajectory have high correlation. To solve this problem, DQN stores experiments in replay-memory. And randomly samples the data to update the neural network.

1. Unstable target value

For each sample (s,a,s’,r), neural-network Q is updated in a way to minimize the difference between Q(s,a) and . is a target value for Q(s,a). But updating the neural network to make Q(s,a) closer to target value also changes Q(s’, a’). This unstable target value makes the neural network hard to converge to actual q-function. DQN uses a target network to solve this problem. Target network is a copy of a prediction network, and gives the value of q-function for the bellman equation. Target network is updated by copying parameters from the prediction network for every C iteration. This gives more stable convergence because it keeps the target value fixed.

DQN introduced two techniques to apply neural networks to q-learning. DQN showed state-of-art performance when it first came out. Just like many other games, we were also able to solve this game by DQN.

2. Code

1. requirement of the code

-gym

-pytorch

-matplotlib

-numpy

import gym

import math

import random

import numpy as np

import matplotlib.pyplot as plt

from collections import namedtuple

%matplotlib inline

import torch

import torch.nn as nn

import torch.optim as optim

import torch.nn.functional as F

import torchvision.transforms as T

import time

1. Class Q

Q is a class which builds neural-network with pytorch.

This class gets shapes as arguments and builds nn according to this shape. Relu is used as an activation function and no activation function is used in the last layer as output of the neural network should have the same domain with the q-function. Forward function gets state as input and outputs value of q-function of every action.

class Q(nn.Module):

def \_\_init\_\_(self, shape):

super(Q, self).\_\_init\_\_()

self.layers = nn.Sequential()

for i in range(len(shape)-1):

self.layers.add\_module(str(i)+"th layer",nn.Linear(shape[i],shape[i+1]))

if(i != len(shape)-2):

self.layers.add\_module(str(i)+"th nonlinear",nn.ReLU())

def forward(self, input):

return self.layers(input)

1. ReplayMemory

ReplayMemory class is an implementation of replay-memory in dqn algorithm. ReplayMemory saves experiences like circle-queue.

It memorizes the front, the first position to insert the new experience.

When a new experience is inserted, replay memory overwrites the data in front and changes the position to (position+1)%capacity.

By this way, replay memory can remove the oldest experience and guarantee there are maximum capacity experiences in memory.

sample method returns random samples with size of batch\_size.

sample method reorganizes the variables by name and converts it to torch tensor. And return samples by following order, state, action, reward, next\_state, done.

Sample = namedtuple('Sample','state, action, reward, next\_state, done')

class ReplayMemory:

def \_\_init\_\_(self, capacity):

self.capacity = capacity

self.memory = []

self.position = 0

def push(self, sample):

if len(self.memory) < self.capacity:

self.memory.append(None)

self.memory[self.position] = sample

self.position = (self.position+1)%self.capacity

def sample(self, batch\_size):

samples = random.sample(self.memory, batch\_size)

states = torch.from\_numpy(np.vstack([sample.state for sample in samples if sample is not None])).float().to(device)

actions = torch.from\_numpy(np.vstack([sample.action for sample in samples if sample is not None])).long().to(device)

rewards = torch.from\_numpy(np.vstack([sample.reward for sample in samples if sample is not None])).float().to(device)

next\_states = torch.from\_numpy(np.vstack([sample.next\_state for sample in samples if sample is not None])).float().to(device)

dones = torch.from\_numpy(np.vstack([sample.done for sample in samples if sample is not None]).astype(np.uint8)).float().to(device)

return (states, actions, rewards, next\_states, dones)

def \_\_len\_\_(self):

return len(self.memory)

1. DQN

DQN class is an implementation of the dqn algorithm.

parameters of the DQN agent can be set by \_\_init\_\_ method.

To train a DQN agent, you can call the train method with the maximum episode and maximum steps of each episode.

class DQN():

"""

\_\_init\_\_

-----------------------

This is the part where parameters of dqn agent is defined

Agent gets the name of the environment by argument and loads the environment from the gym.

Parameters of DQN, like gamma, frequency of updating target network, capacity of replay memory, shape of networks, learning\_rate, should be given as arguments of \_\_init\_\_.

To guarantee that input size and output size of neural network have same size with size of state and number of action,

size of state is inserted in front of net\_shape array and number of action is appended at the end of net\_shape array.

"""

def \_\_init\_\_(self, game\_name, batch\_size, gamma, target\_update\_freq, memory\_capacity, net\_shape, learning\_rate):

self.env = gym.make(game\_name)

self.gamma = gamma

self.memory = ReplayMemory(memory\_capacity)

device = torch.device('cuda:0' if torch.cuda.is\_available() else 'cpu')

observation = self.env.reset()

net\_shape.insert(0,len(observation))

net\_shape.append(self.env.action\_space.n)

self.policy\_Q = Q(net\_shape).to(device)

self.target\_Q = Q(net\_shape).to(device)

self.target\_Q.eval()

self.optimizer = optim.Adam(self.policy\_Q.parameters(), lr=learning\_rate)

self.target\_update = target\_update\_freq

self.batch\_size = batch\_size

#parameters for epsilon-greedy policy

self.eps\_start = 1

self.eps\_end = 0.005

self.eps\_decay = 0.9995

self.epsilon = self.eps\_start

"""

epsilon\_greedy\_policy

------------------------

Epsilon-greedy policy is used during training

To give enough exploration in the beginning and stable greedy policy at the end, the value of epsilon decays from eps\_start to eps\_end, by rate of eps\_decay.

Each parameters are initialized in \_\_init\_\_.

For every action, epsilon is updated to epsilon\_end + (epsilon-epsilon\_end)\*eps\_decay

Agent selects random action by probability of epsilon

"""

def epsilon\_greedy\_policy(self, state):

r = random.random()

state = torch.from\_numpy(state).float().unsqueeze(0).to(device)

self.epsilon = self.eps\_end + (self.epsilon-self.eps\_end) \* self.eps\_decay

if r > self.epsilon:

with torch.no\_grad():

return self.policy\_Q(state).max(1)[1].item()

else:

return random.randrange(self.env.action\_space.n)

"""

update\_policy\_network

------------------------

policy\_network is updated when there are more samples than batch\_size in replay memory

It loads batch\_size number of sample from replay memory.

Than calculate the target value of samples, with bellman equation.

The Q value used in bellman equation si obtained from target network

Than updated policy network in a way to minimize the mse error between predicted q value and target q value.

"""

def update\_policy\_network(self):

if len(self.memory) < self.batch\_size:

return

states, actions, rewards, next\_states, dones = self.memory.sample(self.batch\_size)

max\_action\_values = self.target\_Q(next\_states).detach().max(1)[0].unsqueeze(1)

q\_target = rewards + (self.gamma\*max\_action\_values\*(1-dones)).to(device)

q\_expected = self.policy\_Q(states).gather(1,actions).to(device)

loss = F.mse\_loss(q\_expected, q\_target)

self.optimizer.zero\_grad()

loss.backward()

self.optimizer.step()

"""

update\_target\_network

------------------------

target\_network is updated by copying the value of parameters in policy network.

"""

def update\_target\_network(self):

for policy\_parameters, target\_parameters in zip(self.policy\_Q.parameters(), self.target\_Q.parameters()):

target\_parameters.data.copy\_(policy\_parameters.data)

"""

train

------------------------

Train method trains agent max\_episodes number of episode, which is given as argument.

And terminate each episode after max\_steps of action is made.

At the beginning of each episode, environment is initialized with reset() method.

For each loop, action is selected based on epsilon\_greedy\_policy.

Then agent take action to environment by step method, which return updated state, reward, and whether environment is finished.

Experience is composed of (state, action, reward, next\_state, done). Experience is saved as namedtuple Sample.

Each experience is saved to memory of agent.

Then agent update policy network, and for each target\_update\_freq episode, target network is updated to poilcy network.

State is updated to next\_state and if environment is doned, agent saves total\_reward and whether this episode is succeed.

Agent repeat this till number of step taken is equals to max\_steps.

To check agent is training properly, train method prints average rewards and number of succeed episode for every 100 episode.

Train method return history of training.

"""

def train(self, max\_episodes, max\_steps):

start = time.time()

rewards = [] #array that saves reward agent had achieved for each episode

succ\_episode = [] #array that saves whether each episode had succeed.

print\_interval = 100 #frequency of printing average reward and number of succeed episode.

for episode in range(1,1+max\_episodes):

state = self.env.reset()

total\_reward = 0

for t in range(max\_steps):

action = self.epsilon\_greedy\_policy(state)

next\_state, reward, done, \_ = self.env.step(action)

sample = Sample(state, action, reward, next\_state, done)

self.memory.push(sample)

self.update\_policy\_network()

if t % self.target\_update == 0:

self.update\_target\_network()

state = next\_state

total\_reward += reward

if done:

succ\_episode.append(1 if t != max\_steps-1 else 0)

rewards.append(total\_reward)

break

if episode % print\_interval == 0:

print(f"{episode}th episode : {np.mean(rewards[episode-print\_interval:])} "+f"probability of success : {np.mean(succ\_episode[episode-print\_interval:])}"+" time : "+str(time.time()-start))

return rewards, succ\_episode

1. Train agent & plot the performance

"""

To check the accurate performance of agent, I trained multiple agent and check the average reward.

I plotted average rewards for each episode as blue line, and deviation of rewards as green area.

Therefore we can see average performance and how stable the learning was by this learning curve graph.

"""

agent\_number = 7 #number of agents to be trained

all\_rewards = np.zeros((500, agent\_number))

all\_success = np.zeros((500, agent\_number))

for t in range(agent\_number) :

Agent = DQN('MountainCar-v0', 32, 0.99, 100, 10000, [256, 128], 0.005)

rewards, success = Agent.train(500, 200) #rewards, success saves the log of training

for i in range(len(rewards)) :

all\_success[i, t] = success[i]

all\_rewards[i, t] = rewards[i]

#obtain average rewards and standard deviation of rewards.

mean1 = np.mean(all\_rewards, axis = 1)

std1 = np.std(all\_rewards, axis = 1)

#plotting learning curve

plt.plot(range(len(all\_rewards)), mean1)

plt.fill\_between(range(len(all\_rewards)), mean1-std1, mean1+std1, color = 'g', alpha = 0.5)

plt.xlabel('Epochs')

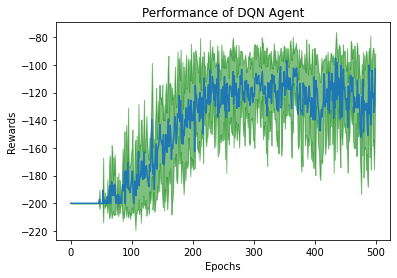
plt.ylabel('Rewards')

plt.title('Performance of DQN Agent')

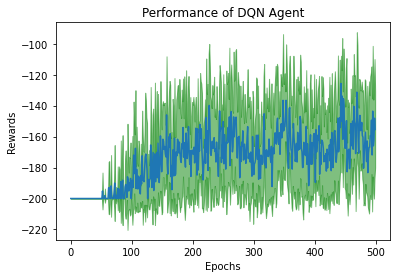
3. Lesson learned from this project

Best performance was obtained when value of each parameters were

(batch size = 32, gamma = 0.99, target\_update\_frequency = 100, memory\_size = 100, net\_shape = [state\_size, 256, 128, action\_size], learing\_rate = 0.005). Learning curve of agents are shown below.

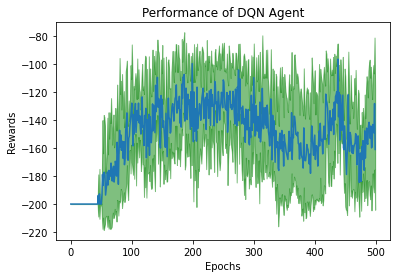


I’ve done ablation tests to check the effects of each technique

1. Ablation of target network

This learning curve is from the agent which has the same parameters, except for target\_update\_freq. target\_update\_freq was set to 1, which means target\_network will have the same parameters with policy network, ignoring the effect of having target network. We can see that the learning curve of DQN agents is very unstable, and has high variance. This gives negative effects to training and we can see the average rewards after 500 epochs is much lower than the ones with best performance. This shows that the target network gives stable learning and better performance.

1. Ablation of replay memory



This learning curve is from the agent with a memory size of 100. Other parameters were set equal to the best one. Small memory increases the correlation between data and lowers the performance of training neural networks. We can see that agent are having much lower performance after 500 epochs. This shows that breaking correlation between data by replay memory is quite important in training.

I’d suffered a lot from low performance of agents, but after setting target\_network\_freq and memory\_size to bigger values, I was able to solve this problem. What I learned from this lesson is that the techniques DQN suggested for using neural networks in NN, plays a very important role in training.