컴퓨터 네트워크 Project

120210393 김준태

Statement of codes

해당 프로젝트는 DQN을 이용하여 gym의 environment 중 하나인 "MountainCar-v0"의 해답을 찾는다. 첨부된 코드의 Neural Network는 딥러닝 프레임워크 중 하나인 Pytorch를 이용하여 작성하였고, Nvidia cuda를 이용한 gpu환경에서 진행되었다. 해당 소스코드는 dqn_utils.py 와 main.py 총 두개의 파일로 구성되어 있고, main.py 파일을 실행하여 실험이 가능하다.

DQN(Deep Q Network)

Google DeepMind에서 개발한 알고리즘으로, 딥러닝을 이용한 Deep-Q Network를 이용하여 approximation 하는 방법이다. 이전에는 state-action에 대한 Q value를 Q table의 형태로 저장하여 update를 진행했던 반면, DQN은 on policy sample들을 이용해 Q value를 approximate하는 network의 파라미터들을 업데이트 한다.

Agent의 experience를 매 스텝 저장하고, Replay memory를 이용하여 모델의 parameter들을 mini batch로 업데이트하는 안정된 학습 방법을 통해 online Q learning보다 좋은 효과를 얻는다. 즉, sequential한 데이터의 문제점인 현재 값이 다음 state의 값을 결정하는데 큰 영향을 주어 local minimum에 빠질 위험을 줄일 수 있어 on-policy 학습의 문제점을 개선하였다.

Requirements

본 코드에 필요한 라이브러리는 다음과 같으며, requirements.txt 를 이용하여 가상환경 설치 후 적용할 수 있다. main.py 파일을 command line 환경에서 실행하면 실험을 진행할 수 있다.

1. gym==0.20.0
2. matplotlib==3.3.4
3. numpy==1.19.5
4. torch==1.10.0
5. torchvision==0.11.1

GPU

학습 과정에서 NVIDIA CUDA를 이용하여 GPU를 할당한다.

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

Modules

Environment에 DQN을 적용하기 위해 여러 모듈을 적용하였고, 각 부분별 설명을 기술한다. 총 5개의 코드 블록에 대해 설명하며 각 코드의 구체적인 모듈은 여러 구현 방법들을 비교한 후 모듈화하기 쉽고 다른 환경에도 적용하기 용이한 알고리즘들을 선택하여 작성하였다.

#1 ReplayBuffer

논문에서는 이를 Experience Replay 라는 방법을 이용한다고 기술하고 있으며, 코드 상 ReplayBuffer 라는 class를 정의하여 이용한다. 이는 Agent를 training할 때 사용되는 모듈로써 각 step 마다 주어지는 state, action, reward, next state에 대한 정보를 저장하고, 이를 재사용하기 위해 임시로 저장해 두는 큐 형태의 자료구조이다.

이전의 방법들은 매 step마다 한 번씩 weight update가 진행되었던 반면, 해당 모듈을 사용하게 되면 random sampling을 거친 mini batch 형태로 데이터들을 저장하고 불러오기 때문에 sequential한 데이터들의 correlation을 없애는 효과로 현재 step과 독립적인 방법으로 weight update를 가능하게 하여 학습에 도움이 된다.

Sample 함수에서는 이를 vstack을 이용하여 각 정보들을 저장하고 sampling한다.

```
class ReplayBuffer:

    def __init__(self, buffer_size, batch_size, seed):
        self.buffer = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state", "action", "reward", "next_state",
"done"])
        self.seed = random.seed(seed)

    def add(self, transition):
        e = self.experience(*transition)
        self.buffer.append(e)

    def sample(self):
        ex = random.sample(self.buffer, k=self.batch_size)
```

```
            states = torch.from_numpy(np.vstack([e.state for e in ex if e is not None])).float().to(device)
            actions = torch.from_numpy(np.vstack([e.action for e in ex if e is not None])).long().to(device)
            rewards = torch.from_numpy(np.vstack([e.reward for e in ex if e is not None])).float().to(device)
            next_states = torch.from_numpy(np.vstack([e.next_state for e in ex if e is not
None])).float().to(device)
            dones = torch.from_numpy(np.vstack([e.done for e in ex if e is not
None]).astype(np.uint8)).float().to(device)

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

      def __len__(self):
            return len(self.buffer)
```

#2. DQN

딥러닝 모델을 정의하기 위한 모듈로, 총 세 개의 layer로 구성되어 있다. 각 fully connected layer의 activation function은 ReLU를 사용하였고, hidden layer의 size는 임의로 설정하였다. (마지막 fc layer는 activation function을 적용하지 않음) Layer의 input과 output 사이즈는 각각 state의 dimension과 action의 dimension이 된다.

```
class DQN(nn.Module):
      def __init__(self, state_n, action_n, seed):
            super(DQN, self).__init__()
            self.seed = random.seed(seed)
            self.layers = nn.Sequential(nn.Linear(state_n, 256), nn.ReLU(),
                                                nn.Linear(256, 256), nn.ReLU(), nn.Linear(256, action_n))

      def forward(self, state):
            return self.layers(state)
```

#3 eps_greedy

Exploration을 위해 epsilon 값을 이용한 알고리즘으로, exploitation + exploration 방식이다. 현재 코드 상에서는 학습 시 'action을 선택할 때' 본 알고리즘을 이용하며, main.py에서는 epsilon 값은 decay를 이용하여 점점 작아지도록 변형을 주어 exploration을 줄이는 방향으로 학습을 진행한다.

```
def eps_greedy(agent, action_n, state, eps):
      if random.random() < eps:
            return random.choice(np.arange(action_n))
      return agent.act(state)
```

#4 Agent

DQN 알고리즘을 직접 수행하는 agent 라는 class를 정의하였다. 해당 부분에서는 딥러닝을 어떻게 적용했고, 각 함수들이 담당하는 부분들은 무엇인지 설명한다.

먼저 Q값을 두개로 정의하는데, Q 네트워크를 EstimateQ 와 TargetQ 두 개로 설정한다. Q 값을 업데이트 할 때 TargetQ는 고정된 값으로 일정 주기마다 EstimateQ의 값을 가져오는 방식을 이용하게 되며, optimizer를 설정할 때에는 EstimateQ의 parameter들만 업데이트를 해주어야 한다.

act 함수는 어떤 state가 입력으로 들어올 때 EstimateQ 값이 가장 높은 action을 선택하는 함수로 본 코드 상에서는 epsilon greedy 알고리즘을 적용하여 사용된다.

step 함수는 위에서 언급한 ReplayBuffer를 이용하여 training에 이용될 mini batch를 설정하기 위한 함수이다. Buffer를 이용하여 에피소드 결과를 저장하고, 설정한 batch size만큼이 모이게 되면 mini batch를 생성하여 training을 진행할 수 있도록 train 함수를 실행시킨다. 또한 학습의 불안정함을 줄이기 위해서 각 episode의 정보들이 학습되는 과정에서 지정된 횟수(1000) 마다 주기를 두고 한 번씩 Target Network의 Q 값을 업데이트 하는 방식을 적용한다.

```
class Agent:
    def __init__(self, state_n, action_n, gamma, seed, buffer_size=2000, batch_size=128,
learning_rate=0.0005):
        self.state_n = state_n
        self.action_n = action_n
        self.batch_size = batch_size
        self.gamma = gamma
        self.EstimateQ = DQN(state_n, action_n, seed)
        self.TargetQ = copy.deepcopy(self.EstimateQ)
        self.buffer = ReplayBuffer(buffer_size, batch_size, seed)
        self.loss = nn.MSELoss()
        self.count = 0
        self.optimizer = torch.optim.Adam(self.EstimateQ.parameters(), lr=learning_rate)
        self.EstimateQ.to(device), self.TargetQ.to(device)

    # state에 대한 argmax action 출력
    def act(self, state):
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        self.EstimateQ.eval()
        with torch.no_grad():
            Q_value = self.EstimateQ(state)
        self.EstimateQ.train()
        return torch.argmax(Q_value).item()

    # 설정한 batch_size 만큼 모이면 random sampling하여 train_batch 생성 후 training 진행
    def step(self, transition):
        self.count = (self.count + 1) % 1000
        self.buffer.add(transition)
        if len(self.buffer) > self.batch_size:
```

```
                train_batch = self.buffer.sample()
                self.train(train_batch)
            if self.count == 0:
                self.TargetQ = copy.deepcopy(self.EstimateQ)

    # training
    def train(self, train_batch):
        states, actions, rewards, n_states, done = train_batch

        Q_value = self.EstimateQ(states).gather(1, actions)
        Q_prime = torch.max(self.TargetQ(n_states), -1)[0].unsqueeze(1)
        Q_target = rewards + (self.gamma * Q_prime.detach() * (1-done))
        loss = self.loss(Q_value, Q_target)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

#5 main

Main 함수에서는 environment를 불러오고 각 state와 action을 pytorch 프레임워크에 맞도록 shape을 수정하고, 변경 가능한 parameter들을 이용하여 테스트를 진행할 수 있도록 구현하였다.

학습을 진행할수록 exploration 횟수를 줄이기 위해 epsilon값을 작아지도록 설정하였고, epsilon greedy를 이용하여 action을 설정하도록 하였다. 또한 agent.step() 을 이용하여 ReplayBuffer를 이용하여 mini batch 단위로 Q 값을 업데이트하도록 학습을 방법을 설정하였다. Episode가 100번씩 학습될 때마다 Score 값이 찍히도록 설정하였고 parameter들을 변경해 가며 학습을 진행하여 결과를 분석하였다.

```
if __name__ == "__main__":
    env = gym.make("MountainCar-v0")
    env.seed(1)

    state_n = env.observation_space.shape[0]
    action_n = env.action_space.n

    #Parameters
    batch_size = 128
    num_episodes = 1000
    gamma = 0.998
    learning_rate = 0.001
    buffer_size = 1000

    eps = 1
    eps_max = 0.5
    eps_min = 0.1
    eps_decay = 200
```

```
    max_steps = 200

    max_avg_reward = float('-inf')
    total_reward = 0
    rewards = []
    rewards_window = deque(maxlen=100)

    agent = Agent(state_n, action_n, seed=1, learning_rate=learning_rate, gamma=gamma,
buffer_size=buffer_size)


    for i in range(1, num_episodes+1):
        #eps = eps_max - (eps_max - eps_min) * i / max_steps
        eps = eps_min + (eps_max - eps_min) * math.exp(-1. * i / eps_decay)
        state = np.array(env.reset())
        done = False

        total_reward = 0

        while not done:
            action = eps_greedy(agent, action_n, state, eps)

            next_state, reward, done, _ = env.step(action)
            total_reward += reward
            reward += 15*(abs(next_state[1]))
            next_state = np.array(next_state)
            agent.step((state, action, reward, next_state, done))
            state = next_state

        rewards.append(total_reward)
        rewards_window.append(total_reward)

        if i >= 100:
            avg_reward = np.mean(rewards_window)
            if avg_reward > max_avg_reward:
                max_avg_reward = avg_reward
                agent.save()
            if i % 100 == 0:
                print("Episode {}/{} | Max Average Score: {:.2f} | eps: {:.2f}".format(i, num_episodes,
max_avg_reward, eps))

    fig = plt.figure()
    ax = fig.add_subplot(111)
    plt.plot(np.arange(len(rewards)), rewards)
    plt.xlabel('Episode')
    plt.ylabel('Reward')
    plt.show()

    env.close()
```

RESULTS

초기에는 총 1500회의 episode를 학습하였지만, 1000번 이후로는 증가하는 추세가 줄어들어 세번째 실험부터는 1000회로 제한하여 실험을 진행하였다.

먼저 초기 parameter 값은 다양한 방법으로 구현된 코드들이 적용하는 각자의 실험적 결과에 따른 최적값을 적용하였고 다음과 같다.

Batch size = 128 | Gamma = 0.98 | Learning rate = 0.0001 | buffer size = 2000
Epsilon = 1 | Maximum epsilon = 0.5 | Minimum epsilon = 0.1 | Epsilon decay = 200

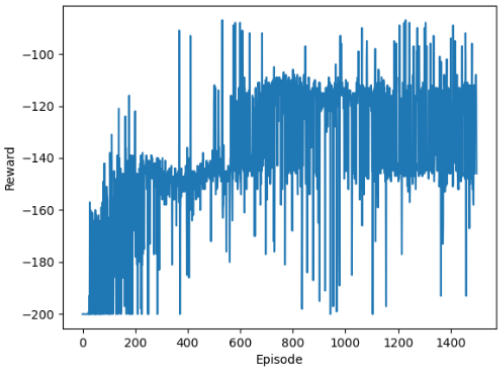
실험은 총 5회 진행하였고, 변형할 parameter 이외 나머지 값은 고정하여 실험을 진행하였다.

Learning rate를 0.001로 설정하였을 경우 더 좋은 결과를 보여주었고, batch size는 줄어들게 되면 #1에서 언급한 mini batch를 이용하여 sequential한 data의 correlation을 줄이는 효과도 줄어들게 되어 성능이 줄어든 것으로 보인다. Exploration의 정도를 조절하기 위해 epsilon의 최대값과 최소값을 조절한 결과 적용중인 epsilon 설정 함수를 사용하였을 때 첫 100번의 iteration부터 0.35 정도의 값을 평균적으로 사용하게 되어 eps_max 설정은 의미가 없었고, eps_min의 경우에도 epsilon 값은 최소값에 가까워지게 설정이 되나 성능에는 큰 영향을 미치지 않음을 확인하였다.

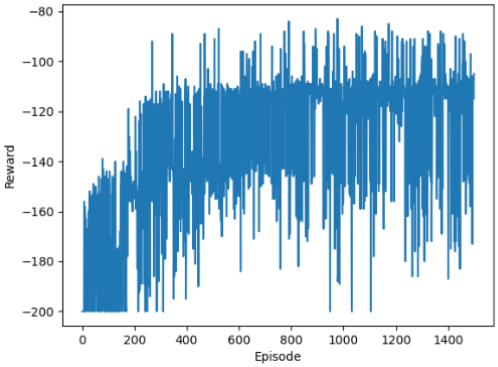
Next state의 감가율에 해당하는 gamma의 경우 0.999와 같은 큰 값을 사용하는 것이 성능에 더 좋은 영향을 미치는 것을 확인하였다.

Buffer size의 경우 학습이 지속될수록 성능에 대한 개선 폭이 크진 않지만, batch size의 효과와 유사하게 학습을 안정화하는 역할을 하는 것으로 확인된다.

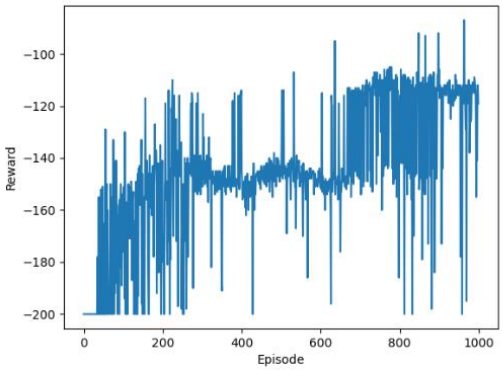
추가적으로 더 많은 실험들을 진행하여 여러 parameter들의 조합을 찾아내야 하지만, 본 프로젝트에서는 분석을 더 진행하지 않고 gym environment가 아닌 open project에 해당 내용을 적용하여 연구를 진행할 계획이다.



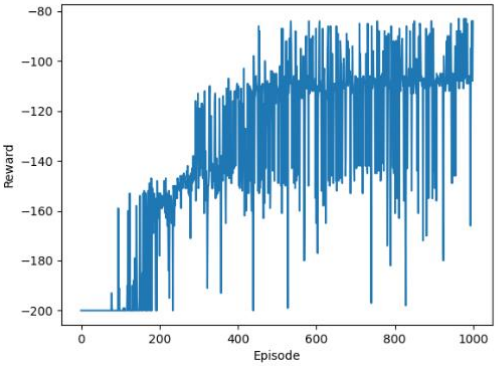
default



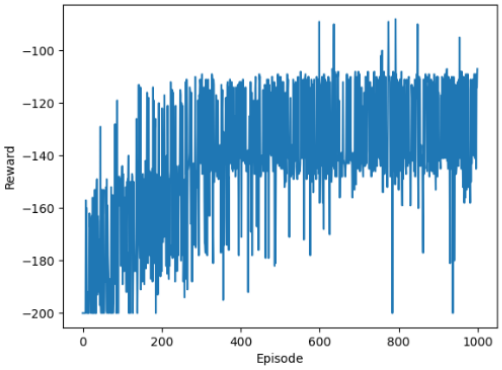
learning_rate = 0.001



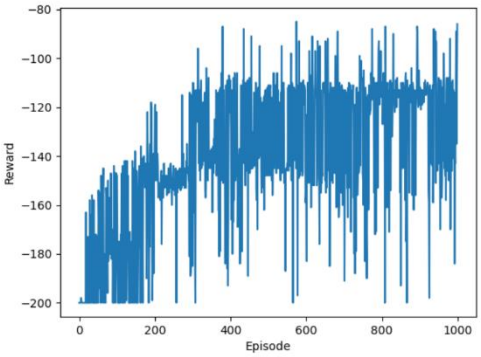
batch_size = 32



eps_max = 0.9, epx_min = 0.01



gamma = 0.9



buffer_size = 1000

Reference: https://github.com/LamaLenny/MountainCar-v0