

AI Programming

Lab 08: DQN_dist

소속: 컴퓨터정보공학부

학번: 2019202103

이름: 이은비

< Lab Objective and Whole simulation program flow >

Deep Q Networks

앞선 Lab 07 과제에서 배운 Q-learning 에서 Q-learning 이 다룰 수 있는 데이터의 스케일이 매우 크다면

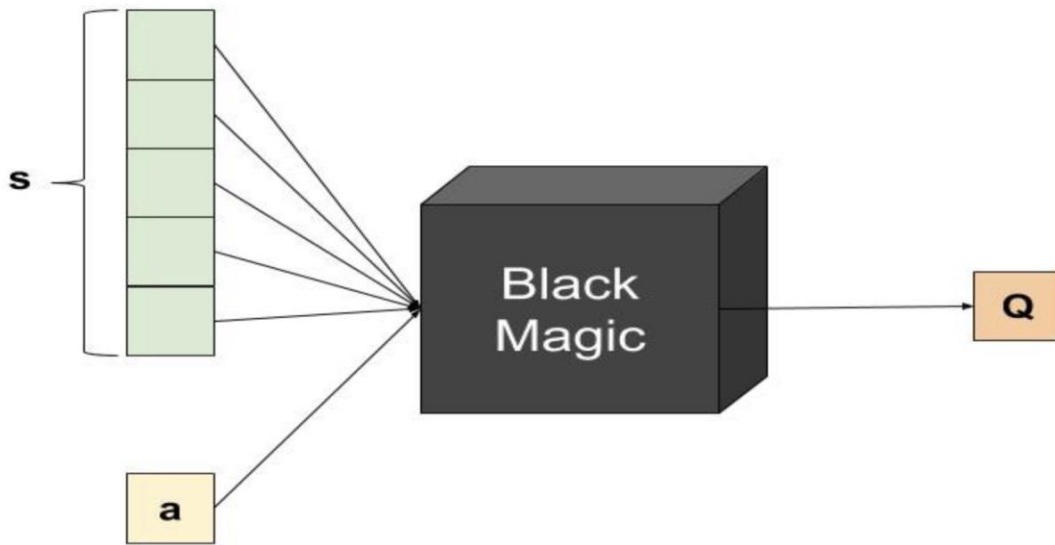
계산 과정이 너무 많아진다는 단점이 생길 것이라는 것을 생각 할 수 있습니다. 따라서 여기서 딥러닝을 과정에 추가하면 어떨까 하는 생각으로 고안된 것이 바로 Deep Q Network 인 DQN 입니다. Q-learning 과 deep learning 을 합친 것을 바로 Deep Q Networks 라고 부릅니다. 아이디어는 Q-table 대신 신경망을 사용해서, 그 신경망 모델이 Q value 를 근사한 값을 낼 수 있도록 학습시키는 것입니다. 그래서 이 모델은 주로 approximator 또는 approximating function 라고 부르기도 합니다. 모델에 대한 식은 $Q(s,a;\theta)$ 으로 표현되며 여기서 θ 는 신경망에서 학습할 가중치를 나타냅니다. 앞서서 Q-learning 에서 사용했던 value estimate 식인 bellman-equation 은 $Q(s,a)=r(s,a)+\gamma \max_a Q(s',a)$ 인데, 이때 Q value 가 수렴해서 참값에 도달했을 때 좌변의 식과 우변의 식이 같아지는 즉, 두 값은 완벽히 같아집니다. 따라서 두 값의 차이를 최소화하는 방향으로 모델을 학습해 나가도록 합니다. 그렇다면, cost function 은 다음과 같은 식으로 간단히 정의됩니다.

$$Cost = \left[Q(s, a; \theta) - \left(r(s, a) + \gamma \max_a Q(s', a; \theta) \right) \right]^2$$

학습 과정에 대해서 보면 강화학습에서는, Training data set 이라는 게 처음에는 따로 없습니다. 지도학습과 비지도학습의 중간 형태의 RL 은 Agent 가 행동을 취함에 따라 데이터셋은 점점 쌓여 나가는 방식을 택하는데 따라서 Agent 는 매 순간 그때까지 학습된 네트워크를 통해 최적이라고 판단된 행동을 취하면서, 에피소드가 끝날 때까지 state, action, reward, 그리고 다음 state 에 대한 데이터를 쌓아 나갑니다.

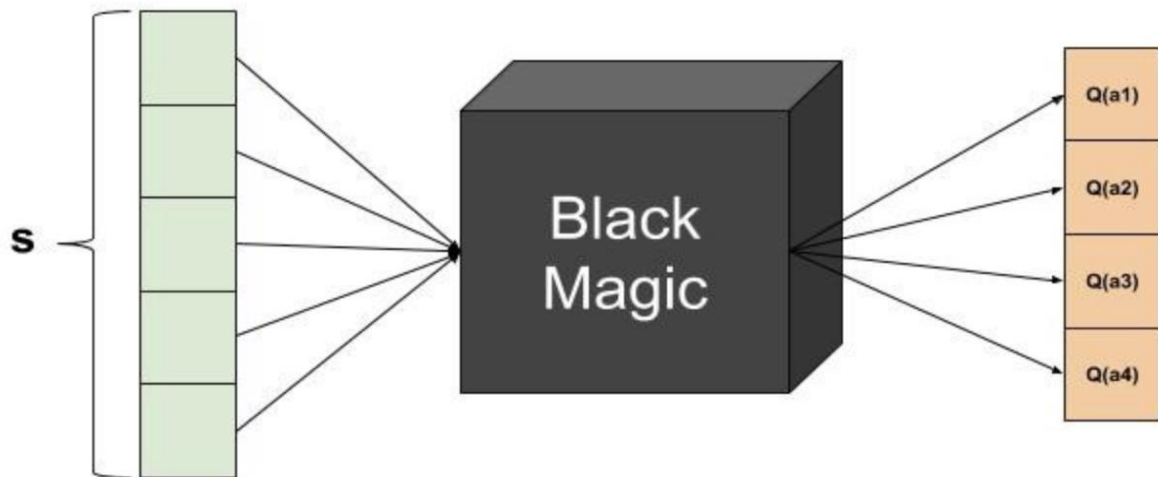
데이터를 쌓으면서 학습하는 과정은 이렇습니다. 학습에 필요한 batch size 를 b 라고 한다면, Agent 는 행동을 b 번 취하면서 b 개의 데이터 set 들이 기록합니다. 이때 저장된 메모리 공간에서 방금 기록된 b 개의 데이터셋으로 학습하는 것이 아니라, 지금까지 쌓인 데이터셋 중 b 개를 랜덤으로 선택해서 신경망을 학습시킨다는 점입니다. 메모리를 저장하는 저장소의 형태는 몇 가지 종류가 있는데, 그 중 가장 많이 쓰이는 것은 cyclic memory buffer 입니다. 이는 Agent 가 학습에 관련 없는 데이터들보다, 조금 더 의미가 있을 최신의 데이터로부터 학습할 수 있도록 도와줍니다.

신경망은 현재 state 에 대한 정보와 현재 취할 action, 두 가지를 입력 받습니다. 그 후에는 모델 내에서 연산을 거쳐 Q-value 를 출력하게 됩니다. 그 이후의 형태는 다음과 같은 그림과 같다고 볼 수 있습니다. 그림에서 s 는 state 를 a 는 action 을 의미합니다.



하지만, 여기 위의 그림을 반영한 식을 생각하면 벨만 방정식에서 우리는 가치를 최대화하기 위해 미래에 선택할 수 있는 행동에 따른 가치 중 최대값을 취해야 하는 부분이 있었습니다.

$\max_a Q(s', a)$ 부분 이 존재하며 이부분을 위한 계산으로 모든 action 을 취해본 결과값을 비교하는 과정 즉, 신경망 연산을 여러 번 돌리는 과정이 필요하다는 비효율적인 부분이 존재합니다. 따라서 구조를 아래와 같이 변경하면 다음과 같습니다.



이렇게 수정하면 우리는 model 의 각 state 들만을 입력해줬을 때 각 행동에 대한 모든 Q-value 를 한번에 얻을 수 있습니다. 그리고 이때 사용하는 수정된 구조에 쓰이는 모델은 논문이나 데이터셋이 큰 경우에는 CNN 구조를 취하며 우리 과제에서는 작은 데이터에 대한 과정이므로 FNN 을 이용하도록 하였습니다. 따라서 이번 과제 Lab08 에서도 이러한 DQN 을 이용하여 cartpole 게임을 실행하도록 해 볼 수 있도록 합니다.

전체적인 흐름을 간단히 코드 블록 단위로 나눠서 전체 진행을 보면 다음과 같이 설명할 수 있습니다. Library Packages 를 준비합니다. GPU 환경을 setting 하고 현재 gpu 가 갖춰져 있는지와 gym 의

version 을 확인합니다 그리고 cartpole 의 예제를 simulaiton 할 수 있는 environment 를 select 합니다. 그리고 training 에 쓰일 함수들인 reset 과 step 의 등등의 함수를 정의합니다. Environment variables 를 check 하고 보여줍니다. 여기서 environment 는 action, observation, state, value 와 같은 것들의 상세한 정보를 보여주도록 합니다. Agent 를 define, initialize 하는과정을 거칩니다. DQNet 을 FNN 으로 layer 를 쌓고 build 한 뒤 policy function 을 define 하고 그 모델을 summary 를 통해 잘 되었는지 확인하고, replay memory 를 define 하고 initialize 하는 과정을 거칩니다. 그리고 Training loop 와 evaluation loop 의 one step 을 define 하고 epsilon function 또한 define 한 뒤 이에 쓰이는 hyperparameters 를 setting 한 뒤 main training loop 를 define 하고 training 시킵니다. 그리고 그에 따른 결과, Training Histories 를 plot 하고 agent 를 evaluate 하고 agent 가 어떻게 진행되는지 video 를 생성 후 보여주는 것으로써 그 결과를 확인합니다.

< Simulation results >

Cartpole 을 DQN 을 이용해서 시뮬레이션 할 수 있도록 환경설정을 합니다.

```
# Check if the code is running in a Colab environment.
if RunningInCOLAB:
    # If in Colab, install necessary packages.
    # Gymnasium is installed with version 0.26.3 along with Box2D dependencies.
    !pip install swig
    !pip install gymnasium==0.26.3
    !pip install gymnasium[box2d]
    !pip install pygame
    !pip install 'moviepy>=1.0.3'
    # Import tqdm for Jupyter Notebooks in Colab.
    from tqdm.notebook import tqdm
else:
    # If not in Colab, import tqdm for other environments.
    from tqdm import tqdm

Collecting swig
  Downloading swig-4.1.1.post1-py2.py3-none-manylinux_2_5_x86_64.manylinux1_x86_64.whl (1.8 MB)
    1.8/1.8 MB 7.2 MB/s eta 0:00:00
Installing collected packages: swig
Successfully installed swig-4.1.1.post1
Collecting gymnasium==0.26.3
  Downloading Gymnasium-0.26.3-py3-none-any.whl (836 kB)
    836.9/836.9 kB 4.3 MB/s eta 0:00:00
Requirement already satisfied: numpy>=1.18.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium==0.26.3) (1.23.5)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium==0.26.3) (2.2.1)
Collecting gymnasium-notices>=0.0.1 (from gymnasium==0.26.3)
  Downloading gymnasium-notices-0.0.1-py3-none-any.whl (2.8 kB)
Installing collected packages: gymnasium-notices, gymnasium
Successfully installed gymnasium-0.26.3 gymnasium-notices-0.0.1
Requirement already satisfied: gymnasium[box2d] in /usr/local/lib/python3.10/dist-packages (0.26.3)
Requirement already satisfied: numpy>=1.18.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium[box2d]) (1.23.5)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium[box2d]) (2.2.1)
Requirement already satisfied: pygame in /usr/local/lib/python3.10/dist-packages (from gymnasium[box2d]) (2.5.1)
```

GPU 가 setting 하고 현재 GPU 가 갖춰진 환경인지 확인하며 앞서서 gymnasium 의 버전을 0.26.3 으로 설정하였는데 맞게 설정되었는지 gym version 을 확인합니다.

```
# Retrieve a list of physical devices (GPUs) available for TensorFlow.
physical_devices = tf.config.list_physical_devices('GPU')
print(physical_devices)

# Try to set memory GPU.
try:
    # if available
    tf.config.experimental.set_memory_growth(physical_devices[0], True)
except:
    # If no GPU is detected
    print('GPU is not detected or an error occurred during configuration.')
```

```
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

```
# check the gym version
gym.__version__
```

```
'0.26.3'
```

Action, Observation, State, Value 등의 정보를 print 하는 과정을 거칩니다.

```
# print the information
print('Action space ', action_space_type)
print('Action shape ', action_shape)
print('Action dimensions ', action_dims)
print('Action range ', action_range)
if action_space_type==gym.spaces.box.Box:
    print('Max Value of Action ', actn_uppr_bound)
    print('Min Value of Action ', actn_lowr_bound)
else: pass
print('Action batch shape ', action_batch_shape)

print('Observation space ', observation_space_type)
print('Observation shape ', observation_shape)
print('Size of State Space ', num_states)
print('State shape ', state_shape)
print('State batch shape ', state_batch_shape)

print('Vallue shape ', value_shape)
print('Value dimensions ', num_values)
```

```
Action space <class 'gymnasium.spaces.discrete.Discrete'>
Action shape (1,)
Action dimensions 1
Action range 2
Action batch shape (None, 2)
Observation space <class 'gymnasium.spaces.box.Box'>
Observation shape (4,)
Size of State Space 4
State shape (4,)
State batch shape (None, 4)
Vallue shape (1,)
Value dimensions 1
```

Layer 를 fnn 으로 하여서 쌓은 후 그 model 의 layer 의 정보를 summary 로 확인합니다.

```
# Create an instance of the Agent_Net class
agent = Agent_Net()
# Call the summary() method on the policy_q network to display a summary of its architecture.
agent.policy_q.summary()
```

```
Model: "q_net"
-----
Layer (type)                 Output Shape              Param #
-----
input_1 (InputLayer)         [(None, 4)]               0
dense (Dense)                 (None, 32)               160
dense_1 (Dense)              (None, 32)               1056
dense_2 (Dense)              (None, 2)                66
-----
Total params: 1282 (5.01 KB)
Trainable params: 1282 (5.01 KB)
Non-trainable params: 0 (0.00 Byte)
-----
```

DQN의 main loop를 training 하는데 10 초이상 막대가 서있을 경우 break 하는 조건으로 training 한 결과 18%진행되었을 때 episodes 는 42, loss 는 1.42750, val_reward 는 470.80 으로 break 된 것을 알 수 있습니다.

```
loss_sum = 0.0
epis_reward = 0.0
epis_steps = 0
num_episodes += 1
else:
    pass

pbar.set_postfix({'episode': num_episodes, 'loss': step_pi_loss, 'reward': eval_reward, 'steps': epis_steps})

# conditions to stop simulation
if eval_reward > goal_score:
    eval_reward = evaluate_policy(env, agent, val_episodes) # evaluate policy multiple times
if eval_reward > goal_score:
    break
if num_episodes > max_episodes:
    break

print('episodes: {0:5d}, loss: {1:7.5f}, val_reward {2:4.2f}'.format(num_episodes, pi_loss, eval_reward))
print('total steps:', sim_steps+1)
```

18% | 5518/30000 [09:18<33:44, 12.09it/s, episode=42, loss=0.0937, reward=500, steps=0]
episodes: 42, loss:1.42750, val_reward 470.80
total steps: 5519

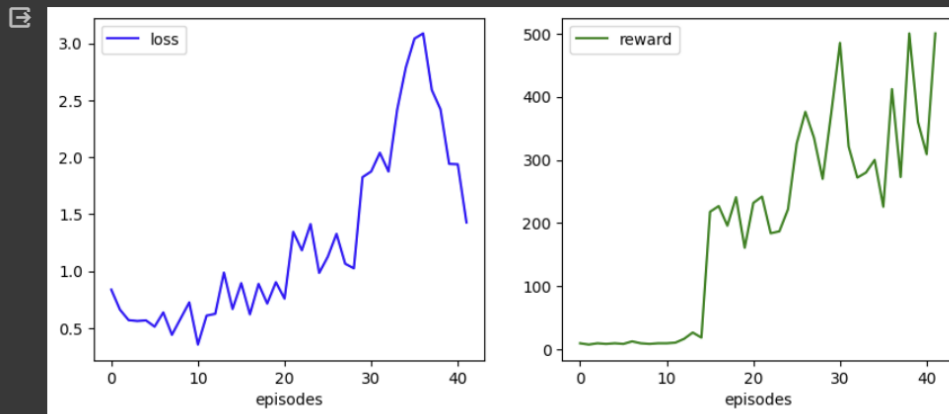
Episodes 에 따른 loss 와 reward 를 확인할 수 있습니다. Reward 가 최대가 되는 것을 궁극적 목표로 하기 때문에 그에 따라 break 되는 지점에서 reward 가 최대인 것을 알 수 있습니다.

```

plt.subplot(1,num_graphs,i+1)
plt.plot(log_history[log_labels[i]], graph_colors[i], label=graph_labels[i])
plt.xlabel('episodes')
plt.legend()
plt.show()
return

log_labels = ['pi_loss', 'vreward']
label_strings = ['loss', 'reward']
label_colors = ['b-', 'g-']
plot_graphs(logs.history, log_labels, label_strings, label_colors)

```



Agent 를 evaluate 한 결과 화면이며 Evaluate Result 는 459.6 입니다.


```

# Evaluate the Agent
evaluate_episodes = 20
sum_episode_rewards = 0.0
pbar = tqdm(range(evaluate_episodes))

for i in pbar:
    sum_episode_rewards += evaluate_policy(env, agent, 1)

print('Evaluation Result:', sum_episode_rewards/evaluate_episodes)

```

100%  20/20 [00:26<00:00, 1.43s/it]
Evaluation Result: 459.6

Cartpole 의 video 를 만듭니다.

```
env.close()

env = create_env()
env = wrappers.RecordVideo(env, video_folder='./gym-results/', name_prefix=res_prefix)

eval_reward = evaluate_policy(env, agent, 1)

print('Sample Total Reward:', eval_reward)

env.close()
```

Moviepy - Building video /content/gym-results/cart-episode-0.mp4.
Moviepy - Writing video /content/gym-results/cart-episode-0.mp4


Moviepy - Done !
Moviepy - video ready /content/gym-results/cart-episode-0.mp4
Sample Total Reward: 500.0

만들어진 video 를 play 할 수 있도록 합니다.

```
# play the game
from IPython.display import HTML
from base64 import b64encode

def show_video(video_path, video_width = 320):
    video_file = open(video_path, "r+b").read()
    video_url = f"data:video/mp4;base64,{b64encode(video_file).decode()}"
    return HTML(f'<video width={video_width} controls><source src="{video_url}"></video>')

show_video('./gym-results/' + res_prefix + '-episode-0.mp4')
```



Cartpole.mp4

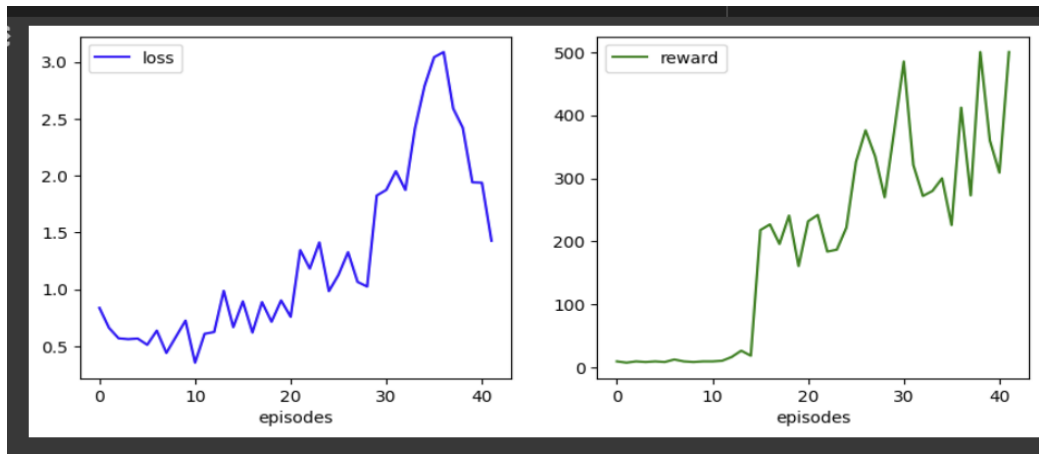
< Discussion >

plot_graphs 함수를 통해 훈련 중에 수집된 로그(history)를 받아서 지정된 두 개의 그래프를 에피소드 수에 따라 플롯합니다.

왼쪽 그래프는 훈련 중 손실의 변화를 시각화 합니다. 이 값이 각 에피소드에서 어떻게 변하는지 보여줍니다.

오른쪽 그래프는 검증 또는 평가 과정에서 얻은 보상(reward) 값을 나타냅니다. 이 그래프는 에피소드당 보상의 변화를 보여줍니다.

앞서서 simulation result 에서 언급한 것과 같이 agent 의 목표는 future reward 가 maximum 한 것을 취하는 방식으로 action 을 선택하여 emulator 과 interaction 합니다. 즉, future reward 를 극대화하는 action 을 선택하는 agent 이므로 최대 reward 이면서 주어진 조건을 만족했을 때(10 초) break 되는 것을 알 수 있습니다.(이때 future reward 는 time-step 마다 factor γ 만큼 discounted 됩니다)



10 초동안의 막대기가 온전히 서있는 것을 확인할 수 있습니다. 위에서 simulation result 로 언급하였을 때의 결과 주어진 조건인 10 초동안 막대기가 서있었을 때 training 을 중단하였고, episodes 는 42, loss 는 1.42750, val_reward 는 470.80 으로 break 된 것을 알 수 있습니다.



Cartpole.mp4

q-learning 과 다르게 DQN 은 더 큰 데이터에 대해서 처리가 가능하며 그 외에도 장점을 보면 experience 의 각 step 은 잠재적으로 많은 weight update 에 사용되어 data efficiency 를 높인다는 점과,

sample 를 randomize 하므로써 sample 간의 correlation 가 깨져(연속적인 sample 에서 직접 학습하는 것은 sample 간의 correlation 으로 인해 비효율적이므로) update 의 variance 가 줄어든다는 것과 off-policy 의 사용으로 안정적인 학습진행이 가능하다는 점을 생각해볼 수 있었습니다.