DQN을 이용한 둠 강화학습

B211200 장현석

이영범

양기열

**1. 주제**

일반적인 아타리 게임학습에서 이용하는 DQN을 3D게임 Doom에 적용시켜 보고, 결과를 관찰한다. 환경은 Vizdoom을 이용한다. (http://vizdoom.cs.put.edu.pl/)

**2. 목표**

▶Vizdoom에 제시되어있는 시나리오를 최대한 많이 통과한다.

▶한계가 존재하는 경우, 무엇이 문제인지 생각해 본다.

▶다른방법이 존재하는지 찾아본다.

**3. 역할**

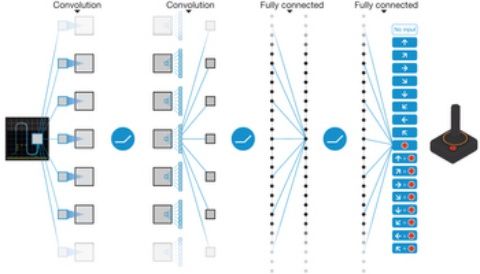
장현석: 자료조사, 코드분석 및 작성, 주석, 네트워크 작성, 테스팅, 보고서 작성

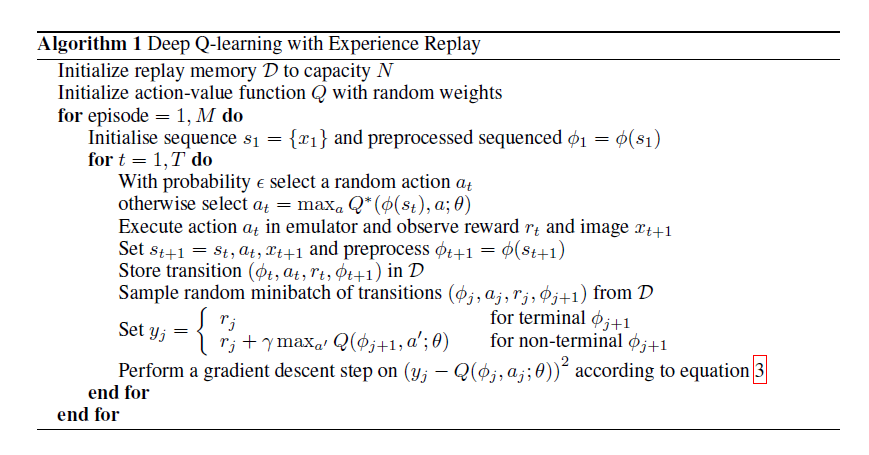
이영범

양기열

**4. 관련 연구 및 알고리즘**

게임학습을 하는 것은 여러 곳에서 하고 있지만, 대표적으로 알려진곳은 구글의 딥마인드 팀입니다. 딥마인드팀이 개발한 Deep Q Learning 기술이 게임 강화학습의 대표적인 연구 예 이며, 여러분야에 두루 쓰이고 있습니다.

.



이미지를 받아서 액션과 다음상태, 보상을 메모리에 넣고, 이러한 과정이 일정이상 반복되었을 때, 랜덤하게 샘플을 뽑아 학습을 진행합니다. 여기서 이미지는 알맞은 형태로 가공되어 받아들여져야 하기 때문에 convolution을 진행합니다. 최종 출력으로 어떤 행동을 할지 결정되며, 주로 one hot을 이용해 어떤 버튼을 눌러야 하는지 결정합니다.

DQN은 우리가 게임을 하는 방식과 비슷하게 동작합니다. 어떤 화면을 보고, 정보를 판단해 이득이 많은쪽으로 버튼을 누르게 하는 방식입니다. 둠과 같은 3D게임도 화면을 보고 정보를 판단해 적절한 행동을 취하는 방법입니다. 기존에 많이 시도되고 있는 아타리 게임뿐만 아니라, 둠과 같은 게임에서도 잘 동작할거 같아서 DQN을 선택해 학습을 진행했습니다.

convolution

프레임

(30\*45)

가능한 행동

128

320

**5. 데이터**

강화학습이기 때문에 데이터 셋을 따로 다운받는 것이 아닌, Vizdoom환경에서 화면프레임을 흑백으로 받아와 가공을해서 입력값으로 넣어줍니다. 화면을 받아서 30\*45의 해상도로 리사이즈를 진행하고, convolution레이어를 통과해서 320의 출력으로 Fully Connected Layer에 전달됩니다. 최종 출력은 Vizdoom에서 취할 수 있는 액션의 index길이를 받아 결정해 줍니다. 이는 시나리오마다 다르기 때문에 변수값으로 전달해 줍니다.

**6. 소스코드**

Vizdoom의 튜토리얼과 텐서플로우 예제파일로부터 가져온 것이 많으며, 주로 네트워크부분과 하이퍼 파라메터, 드랍아웃레이트를 커스터마이즈 했습니다. 모든파일이 다음 소스코드와 같진않지만, 큰틀에서 벗어나지 않기에 핵심이되는 하나만 기록합니다.

추가로 설치한 라이브러리는 vizdoom에서 요구하는 scikit-image, scipy, tqdm입니다.

# 비즈둠의 환경을 설정합니다. config파일을 넣으면 액션과 환경을 자동으로 설정해줍니다.

def initialize\_vizdoom(config\_file\_path):

print("Initializing doom...")

game = DoomGame()

game.load\_config(config\_file\_path)

game.set\_window\_visible(True)

game.set\_mode(Mode.PLAYER)

game.set\_screen\_format(ScreenFormat.GRAY8)

game.set\_screen\_resolution(ScreenResolution.RES\_640X480)

game.init()

print("Doom initialized.")

return game

# 원하는 해상도값을 받아 이미지 사이즈를 조정하는 함수입니다.

def preprocess(img):

img = skimage.transform.resize(img,resolution,mode='constant')

img = img.astype(np.float32)

return img

#메모리 저장용 함수, 스테이트와 액션, 다음스테이트, 보상, 엔딩여부를 저장합니다.

class ReplayMemory:

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

channels = 1

state\_shape = (capacity, resolution[0], resolution[1], channels)

self.s1 = np.zeros(state\_shape, dtype=np.float32)

self.s2 = np.zeros(state\_shape, dtype=np.float32)

self.a = np.zeros(capacity, dtype=np.int32)

self.r = np.zeros(capacity, dtype=np.float32)

self.isterminal = np.zeros(capacity, dtype=np.float32)

self.capacity = capacity

self.size = 0

self.pos = 0

def add\_transition(self, s1, action, s2, isterminal, reward):

self.s1[self.pos, :, :, 0] = s1

self.a[self.pos] = action

if not isterminal:

self.s2[self.pos, :, :, 0] = s2

self.isterminal[self.pos] = isterminal

self.r[self.pos] = reward

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

self.size = min(self.size + 1, self.capacity)

def get\_sample(self, sample\_size):

i = sample(range(0, self.size), sample\_size)

return self.s1[i], self.a[i], self.s2[i], self.isterminal[i], self.r[i]

#네트워크 만들기

class DQN:

def \_\_init\_\_(self,session, available\_actions\_count):

self.imageIn = tf.placeholder(tf.float32, [None] + list(resolution) + [1], name="State")

self.target\_q\_ = tf.placeholder(tf.float32, [None, available\_actions\_count], name="TargetQ")

self.keep\_prob=tf.placeholder(tf.float32)

# 2개의convolutional 레이어 추가

self.conv1 = tf.contrib.layers.convolution2d(self.imageIn, num\_outputs=8, kernel\_size=[6, 6], stride=[3, 3],

activation\_fn=tf.nn.relu,

weights\_initializer=tf.contrib.layers.xavier\_initializer\_conv2d(),

biases\_initializer=tf.constant\_initializer(0.1))

self.conv2 = tf.contrib.layers.convolution2d(self.conv1, num\_outputs=8, kernel\_size=[3, 3], stride=[2, 2],

activation\_fn=tf.nn.relu,

weights\_initializer=tf.contrib.layers.xavier\_initializer\_conv2d(),

biases\_initializer=tf.constant\_initializer(0.1))

self.conv2\_flat = tf.contrib.layers.flatten(self.conv2)

# Fully Connected 레이어입니다

self.FC1 = tf.contrib.layers.fully\_connected(self.conv2\_flat, num\_outputs=128, activation\_fn=tf.nn.relu,

weights\_initializer=tf.contrib.layers.xavier\_initializer(),

biases\_initializer=tf.constant\_initializer(0.1))

self.FC1=tf.nn.dropout(self.FC1,keep\_prob=self.keep\_prob)

self.FC2 = tf.contrib.layers.fully\_connected(self.conv2\_flat, num\_outputs=128, activation\_fn=tf.nn.relu,

weights\_initializer=tf.contrib.layers.xavier\_initializer(),

biases\_initializer=tf.constant\_initializer(0.1))

self.FC2=tf.nn.dropout(self.FC2,keep\_prob=self.keep\_prob)

#액션의 갯수만큼 q아웃풋이 나옵니다.

self.q = tf.contrib.layers.fully\_connected(self.FC1, num\_outputs=available\_actions\_count, activation\_fn=None,

weights\_initializer=tf.contrib.layers.xavier\_initializer(),

biases\_initializer=tf.constant\_initializer(0.1))

#q아웃풋에서 액션을 결정

self.best\_a = tf.argmax(self.q, 1)

#loss를 계산합니다

self.loss = tf.contrib.losses.mean\_squared\_error(self.q, self.target\_q\_)

self.loss\_summ = tf.summary.scalar('loss',self.loss)

#Optimier과정

self.optimizer = tf.train.RMSPropOptimizer(learning\_rate)

# loss를 줄입니다

self.train\_step = self.optimizer.minimize(self.loss)

def function\_learn(self,s1, target\_q):

self.feed\_dict = {self.imageIn: s1, self.target\_q\_: target\_q, self.keep\_prob: 0.9}

s,l, \_ = session.run([summary,self.loss, self.train\_step], feed\_dict=self.feed\_dict)

writer.add\_summary(s, global\_step=epoch)

return l

def function\_get\_q\_values(self,state):

return session.run(self.q, feed\_dict={self.imageIn: state, self.keep\_prob: 1})

def function\_get\_best\_action(self,state):

return session.run(self.best\_a, feed\_dict={self.imageIn: state, self.keep\_prob: 1})

def function\_simple\_get\_best\_action(self,state):

return self.function\_get\_best\_action(state.reshape([1, resolution[0], resolution[1], 1]))[0]

#학습함수, Q값, 행동반환

#메모리에서 자료를 가져와 학습하는 함수입니다. 밑에 perform\_learning\_step에서 쓰입니다.

def learn\_from\_memory():

# 배치사이즈만큼 차면 러닝을 시작합니다.

if memory.size > batch\_size:

s1, a, s2, isterminal, r = memory.get\_sample(batch\_size)

q2 = np.max(mainDQN.function\_get\_q\_values(s2), axis=1)

target\_q = mainDQN.function\_get\_q\_values(s1)

# Q값을 계산하고 업데이트합니다

target\_q[np.arange(target\_q.shape[0]), a] = r + discount\_factor \* (1 - isterminal) \* q2

mainDQN.function\_learn(s1, target\_q)

#학습을 수행하는 함수입니다.

def perform\_learning\_step(epoch):

#입실론값을 설정합니다

def exploration\_rate(epoch):

start\_eps = 1.0

end\_eps = 0.1

const\_eps\_epochs = 0.1 \* epochs # 10% of learning time

eps\_decay\_epochs = 0.6 \* epochs # 60% of learning time

if epoch < const\_eps\_epochs:

return start\_eps

elif epoch < eps\_decay\_epochs:

return start\_eps - (epoch - const\_eps\_epochs) / \

(eps\_decay\_epochs - const\_eps\_epochs) \* (start\_eps - end\_eps)

else:

return end\_eps

s1 = preprocess(game.get\_state().screen\_buffer)

# 랜덤한 행동을 할지 결정합니다

eps = exploration\_rate(epoch)

if random() <= eps:

a = randint(0, len(actions) - 1)

else:

# 메인DQN에서 행동을 택합니다

a = mainDQN.function\_simple\_get\_best\_action(s1)

reward = game.make\_action(actions[a], frame\_repeat)

isterminal = game.is\_episode\_finished()

s2 = preprocess(game.get\_state().screen\_buffer) if not isterminal else None

# 메모리에 넣어줍니다.

memory.add\_transition(s1, a, s2, isterminal, reward)

#메모리에서 자료를 꺼내 러닝합니다

learn\_from\_memory()

#메인 함수입니다.

if \_\_name\_\_ == '\_\_main\_\_':

# 둠 초기설정

game = initialize\_vizdoom(config\_file\_path)

# 가능한 버튼갯수, 액션을 받습니다.

nOfAction = game.get\_available\_buttons\_size()

actions = [list(a) for a in it.product([0, 1], repeat=nOfAction)]

#저장할 메모리를 만듭니다

memory = ReplayMemory(replay\_memory\_size)

#세션을 엽니다

session = tf.Session()

mainDQN = DQN(session,len(actions))

saver = tf.train.Saver()

summary = tf.summary.merge\_all()

writer = tf.summary.FileWriter('./logs')

writer.add\_graph(session.graph)

#모델 로드

if load\_model:

print("Loading model from: ", model\_savefile)

saver.restore(session, model\_savefile)

else:

init = tf.initialize\_all\_variables()

session.run(init)

print("Starting the training!")

time\_start = time()

if not skip\_learning:

#epoch시작

for epoch in range(epochs):

print("\nEpoch %d\n-------" % (epoch + 1))

train\_episodes\_finished = 0

train\_scores = []

print("Training...")

game.new\_episode()

#epoch당 반복되는 횟수를 받아 반복합니다

for learning\_step in trange(learning\_steps\_per\_epoch):

episode\_buff=[]

perform\_learning\_step(epoch)

if game.is\_episode\_finished():

score = game.get\_total\_reward()

train\_scores.append(score)

game.new\_episode()

train\_episodes\_finished += 1

print("%d training episodes played." % train\_episodes\_finished)

train\_scores = np.array(train\_scores)

print("Results: mean: %.1f±%.1f," % (train\_scores.mean(), train\_scores.std()),

"min: %.1f," % train\_scores.min(), "max: %.1f," % train\_scores.max())

print("\nTesting...")

test\_episode = []

test\_scores = []

for test\_episode in trange(test\_episodes\_per\_epoch):

game.new\_episode()

while not game.is\_episode\_finished():

state = preprocess(game.get\_state().screen\_buffer)

best\_action\_index = mainDQN.function\_simple\_get\_best\_action(state)

game.make\_action(actions[best\_action\_index], frame\_repeat)

r = game.get\_total\_reward()

test\_scores.append(r)

test\_scores = np.array(test\_scores)

print("Results: mean: %.1f±%.1f," % (

test\_scores.mean(), test\_scores.std()), "min: %.1f" % test\_scores.min(),

"max: %.1f" % test\_scores.max())

print("Saving the network weigths to:", model\_savefile)

saver.save(session, model\_savefile)

print("Total elapsed time: %.2f minutes" % ((time() - time\_start) / 60.0))

game.close()

**7. 학습과정**

성공했던 시나리오인 Basic 권총으로 몬스터 맞추기 입니다. Tensor board를 사용하고 싶었지만 오류가 나서 엑셀로 정리했습니다.

Vizdoom에서 제공하는 Score함수를 평균낸 값을 그래프로 만들었습니다. 가로축은 epoch를 나타내며, 세로축은 100번테스팅을 시행했을때 얻은 Score의 평균값입니다. 트레이닝스텝은 2000회입니다.

위 그래프에서 알 수 있듯이 단순 몬스터 맞추기는 성공적으로 학습되었습니다. FC를 1개사용했을때와 FC를 2개사용했을때를보면 2개일때가 성능이 더 좋았고, 여기서 Learning Rate를 0.00025에서 0.0025로 증가시키자 학습속도가 좀 더 빨라졌습니다. 세가지 모두 epoch 5부터 일직선으로 수렴합니다.

결과모델은 성공한 것을 기준으로 저장했으며, 파이썬파일을 재생하면 실행됩니다.

다음으로 동작한다고 할 수 있는 시나리오는 로켓으로 몬스터 맞추기 입니다.

이전에는 권총이기 때문에 버튼을 누르면 즉발로 맞췄지만, 로켓은 투사체기 때문에 날아가는 시간이 있습니다. 때문에 움직이면서 쏴야 할 경우 자신이 이동하는 방향과, 투사체의 발사 속도를 계산해야 합니다.

FC를 1개쓰는경우와 2개쓰는경우, 그리고 Learning Rate에 변화를 줘봤고, drop rate도 0.9로 설정해 봤습니다. 결과는 셋다 큰 차이가 없었습니다. 어느쪽이 빠르다고 하기에는 편차가 크고, 그래프가 일관성있다고 하기 어렵습니다. 다만 중반 이후로는 세가지 모델 전부 비슷한 추세를 보이고 있습니다.

플레이 영상을 확인해본 결과, 빗맞출때도 있지만 대부분 3정도의 격발로 몬스터를 맞춥니다. 썩 만족할만한 결과는 아니지만, 그래도 학습을 했다고는 할 수 있는 수준의 결과였습니다.

다음은 실패한 예 입니다. 위에 두 시나리오를 빼고는 학습에 실패했습니다. 그 중 대표적인 예 하나로 HP키트 모으는 시나리오 입니다.

이 시나리오는 지속적으로 데미지를 주는 땅에서 HP키트를 모아서 살아남는 시나리오 입니다. 입력값으로 이전과는 다르게 화면회전 좌,우와 앞으로 가는입력이 있습니다.

FC2개에서 Learning Rate를 0.00000002로했을경우 AI가 전혀 움직이지를 않았습니다. 레이트가 너무 작아 그런것이라 판단하고 10배인 0.0000002로 했지만 결과는 비슷했습니다. FC를 1개로 줄인 후 0.0000002로 해본 결과 중간에는 학습이 일어나는가 싶더니 곧 예전과 같은 상태로 돌아갔습니다. 다음으로 레이트를 0.00000017로 수정했더니 약간은 점수가 올라간 모습입니다.

드롭레이트를 0.7에서 0.9까지 변화도 시켜봤지만, 큰 변화가 없어서 엑셀그래프에는 따로 명시 하지 않았습니다.

플레이 영상을 확인해 본 결과, 러닝레이트를 조정해도 아에 움직이지 않거나, 한쪽으로만 빙빙 돌거나, 아니면 방향전환 없이 앞으로만 가는 모습을 볼 수 있었습니다.

8. 실험 및 결과

실험 결과는 권총으로 몬스터맞추기와 로켓으로 몬스터맞추기만 학습했다는 수준의 결과물이 나오고 나머지는 실패했습니다. 실패한 시나리오의 공통점은 회전으로 인한 화면의 변환과 어떤 물체의 인접성을 판단하는 경우였습니다.

성공했던 두 시나리오는 회전이 없기 때문에 화면이 실제로는 거의 고정되어있다고 봐도 됩니다. 좌, 우 이동을 하긴하지만 그렇게 큰 변화가 없기에 몬스터를 맞추는 학습은 어렵지 않게 진행되었습니다.

반면 나머지 시나리오들은 몬스터가 다가오거나, 투사체가 날아오는등 한가지 프레임을 줬을때는 어떤행동을 해야하는지 판단하기 어려운 경우가 많았습니다. 첨부한 영상을 보면 HP키트를 먹는 학습은 벽을향하여 돌진하는행동밖에 안합니다. 다르게 러닝레이트를 조정해 봤지만, 가만히 있거나 오른쪽이나 왼쪽으로 빙빙도는 행동을 할 뿐이였습니다.

결론은 DQN은 3D환경에서의 학습에는 적합하지 않다는 것입니다.

**문제 인식, 개선법**

3D환경에서는 DQN이 잘 동작하지 않았습니다. 문제는 환경의 차이인 것 같습니다. 아타리 환경 게임들은 주로 3인칭시점의 Fully Observable환경입니다. 따라서 화면에 모든 요소가 한프레임에 나타나기 때문에 잘 동작합니다.

하지만 둠의 경우는 3D게임이기 때문에 Partially Observable환경이 됩니다. 이동을 어떻게 하느냐에 따라 인식할 수 있는환경이 변하기 때문에 한프레임 가지고 행동을 판단하기엔 너무 비효율적입니다. 둠에서의 경우는 어떤 물체가 앞에왔을 때 쏴야한다는 것을 판단할 순 있지만, 다가오고 있는중인지 판단하기가 힘든 것이 됩니다.

이에대한 개선법을 생각해보면 RNN을 이용하는 것이 있습니다. 한 프레임으로 상황을 판단하기가 부적절하다면, 연속된 여러 프레임으로 행동을 판단하게 하면 됩니다. Sequential한 데이터를 판단하는데 유리한 RNN의 특성을 DQN과 결합하면 둠에서도 잘 동작할 것 같습니다. 실제로 8개의 연속된 프레임을 저장하여 convolution진행후 RNN셀로 보내는 구조를 생각해 보았습니다.

프레임

프레임

프레임

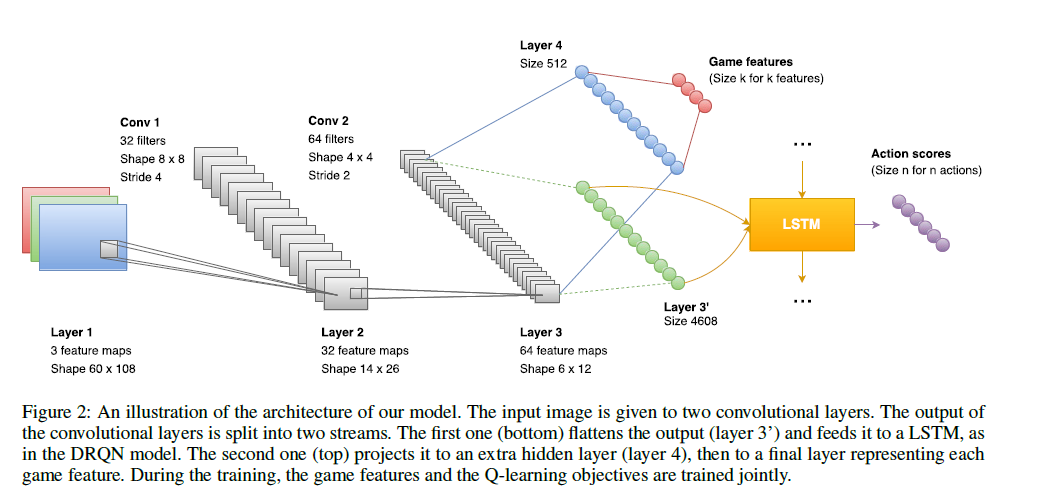
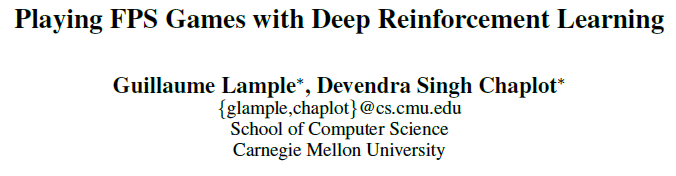
프레임

RNN셀

FC

위와 같은 방법으로 DQN에 RNN을 추가한 네트워크를 만들려고 했으나, 데이터셋을 넣는 방식에 문제가 생겨 구현에는 실패했습니다.

실제 사례를 찾아보면 이러한 방식을 Deep Recurrent Q Learning이라고 합니다. 카네기 멜론대학에서 이러한 구조로 둠 AI를 만든적이 있고, 꽤 잘 동작합니다.



카네기 멜론에서 만든 DRQN 구조