CartPole 예제도 마찬가지로 Qnet 적용해보자.

random\_uniform 쓰던거, 이렇게 바꿔줬음 – xavior initializing

        initializer = tf.initializers.GlorotUniform()

        self.W = tf.Variable(initializer(

            shape = (input\_size, output\_size)))#weight initializing?

my\_model에서 float 64를 float32로 캐스팅하려 한다는 warning이 뜬다.

학습 시도를 해보는데 딱히 학습이 잘되지는 않는다. 강의에서도 평균 몇십 step정도 reward를 받던데, 성능 자체는 정상적이다. 문제는, 속도가 너무 안나온다는 것이다! 강의에서 학습되는 속도보다 한 20배는 느린 것 같다… 해결법이 없을까? 코드를 이곳 저곳에서 누더기처럼 기워 댔기에 어쩔 수 없는 것일까, 아니면 tensorflow 2.0의 속도가 어쩔 수 없이 이정도 느린걸까.

----속도 문제 관련 구글링 하다가 찾게된 잡설 -------

<https://data-newbie.tistory.com/425> 포스트에 따르면, 요즘 대세는 Pytorch라고 한다. 새로운 논문들이 전부 pytorch 기반으로 나온다고 한다. tf진영 창시자인 Deepmind조차 pytorch로 넘어가고 있는 추세라고. 이게 사실이면…

Tf의 장점은 기업에서 쓰고있고 배포에서 장점이 있다는 것인데, 문제는 TF2는 그런 장점이 하나도 없다는 것이다. 게다가 3월기준 TF2는 성능 메모리 버그 문제들로 고통받고있다는데 어…

timeit을 이용해 걸리는 시간을 측정해보았다. 문제가 되는 부분은 생각보다 많은 시간이 걸릴 것이다. 반복되는 구간마다 time stamp를 numpy.array에 더해준 뒤 차이를 구했다. 이 값들을 각각 .sum()해주면 각 구간이 전체적으로 얼마나 시간을 잡아먹는가 확인할 수 있을 것이다. 15 episode동안 구해본 결과…

time12 : 14.157866400000001 (사이클 시작부터 action을 취해 값을 받아 온 직후까지)

time23 : 14.286717899999978 (값을 받아 온 직후부터 optimize할때까지)

whole time: 29.4581684

충격적이게도, 비슷한 시간을 가진다. 균일하게 문제가 있다는 뜻일까? 구간을 조정해서 다시 측정해보았다.

time12 : 16.058064400000035 (action 받아오기 직전까지)

time23 : 16.43890829999998 (action 받고, optimize하는 시간 포함)

whole time: 33.4426092

더 충격적이게도, action을 받아오는 행위 자체는 별다른 시간이 필요하지 않은 것 같다. 얘가 주된 범인일 줄 알았는데?

한칸 더 올려보자… 그랬더니

time12 : 0.004294699999983997(시작~Qs 구하기 위해 modelQpred.predict 하기 전까지)

time23 : 24.76996160000003(predict 이후)

whole time: 25.8092579

prediction이 문제였다!!!

prediction을 tf.matmul(x,modelQpred.W)로 바꿔주었더니, 예전에 만난 에러 (expected to be a double tensor but is a float tensor)가 다시 나왔다. 맞다, prediction 쓴 이유가 이거였지.

modelQpred(x)를 사용했더니

'tensorflow.python.framework.ops.EagerTensor' object does not support item assignment 에러가. Qs가 eager tensor여서 값을 변경하지 못한다는 의미인 것 같은데…?

<https://github.com/tensorflow/tensorflow/issues/40261> 포스트의 tomerk가 남긴 댓글에 따르면 model.predict는 루프 안에서 쓰면 너무 오버헤드가 큰 top-level api라며, 대체품으로 아래 문장을 사용할 것을 추천하고 있다.

model(x), training=False)

training=False 를 넣어보았지만 여전히 같은 오류

오류를 구글링 해보니 tensor가 not assignable하다는 사실에 많은 사람들이 분개해하며 pytorch를 찬양하고 있었다. 어찌되었건, tf variable로 변경 해준 뒤 값을 바꿔줘야 할 것 같다.

variable로 변경해도 assign이 불가하다! Variable.assign을 써야한다는 것 같은데, 차라리 tf.make\_ndarray 를 사용해보자.

여전히 안된다. 절망적이다… eager tensor는 tensor\_shape 속성이 없다며 에러를 밷는다.

.numpy()를 사용하면 된다고 한다!

된다!!! 속도가 무지막지하게 빨라졌다!

어찌되었건, 왜 학습이 잘 안되는가?

network 변수가 4개밖에 없다, 너무 shallow함. +

샘플간 correlation, 타겟의 불안정(non-stationary)이 원인.

DQN으로 들어가보자.

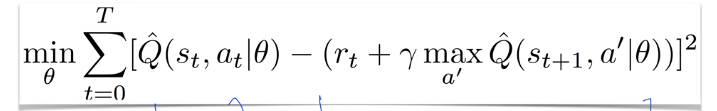
<https://dryjelly.tistory.com/140?category=785786> 참고할만한 tistory post

Nature 표지를 장식한 Deepmind 팀의 DQN 논문.

Correlation between sample의 의미는 무엇인가?

action 1회를 취했을 때 state는 극적으로 변화하지 않는다. 즉, 학습에 필요한 sample(state) 간에유사성이 너무 크다. 샘플이 넓게 퍼져 있으면 학습하기 쉽지만, 비슷한 위치에 몰려 있는 샘플로는 제대로 된 모델이 학습되는 것을 기대하기가 힘들다.

non-stationary targets



같은 네트워크를 사용하기 때문에, Q를 최적화 시키면 target 역시 변경되어버린다. 저번주 정리에 생각했던 문제점.

해결법? 더 깊게, capture and replay, 네트워크 분리.

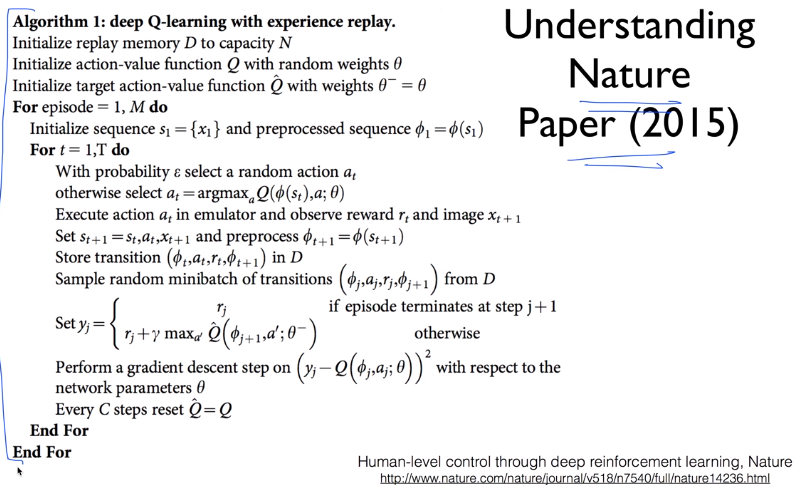
Go deep: convolution network, full connect 등등…

capture and replay : 루프를 도는 도중 동시에 학습시키지 말고, 버퍼에 저장해 놓은 뒤 학습시킨다. 이 때 저장된 transition들 중 랜덤하게 뽑아 minibatch로 만든다.



단 두줄로!

Separate target : 네트워크를 하나 더 만든다. 기본 Qnet 하나와, Target 도출용 network 하나. Target network를 고정시킨 뒤 Qnet을 먼저 학습시키고,일정 C 스텝 마다 이 둘을 동기화 시켜준다.



1 리플레이 저장할 버퍼 D 생성

2 action-value function생성

3 target action-value function 생성

4

5 state 초기화 및 전처리

6

7~8 E greedy 기반 action 선택

8

9 action 취하고 r, 이미지 획득

10 state바꿔주고 전처리

11 저장

12 랜덤 minibatch 뽑은 뒤

13

14 target 을 구하고

15

16 target과 비교해 학습-일단시타만

17

18 시타 바 에 동기하ㅗ

19

20

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

DQN까지 tensorflow2로 스스로 만들어 보기? 차원이 다른 난이도가 되지 않을까 싶다. 그래도, 시도해 볼까?

지금 내가 쓰는 코드는 여러 부분에서 코드를 기워왔기 때문에 엉망진창인 점이 있다. 따라서 이번엔 새로 코드를 가져와서 어떤식으로 구현했는지 비교도 해 보고 배울점은 배우고 해보자.

<https://towardsdatascience.com/deep-reinforcement-learning-build-a-deep-q-network-dqn-to-play-cartpole-with-tensorflow-2-and-gym-8e105744b998> 참고 코드는 여기에서 가져온다.

|  |
| --- |
| import numpy as np  import tensorflow as tf  import gym  import os  import datetime  from gym import wrappers  class MyModel(tf.keras.Model):      def \_\_init\_\_(self, num\_states, hidden\_units, num\_actions):          super(MyModel, self).\_\_init\_\_()          self.input\_layer = tf.keras.layers.InputLayer(input\_shape=(num\_states,))          self.hidden\_layers = []          for i in hidden\_units:              self.hidden\_layers.append(tf.keras.layers.Dense(                  i, activation='tanh', kernel\_initializer='RandomNormal'))          self.output\_layer = tf.keras.layers.Dense(              num\_actions, activation='linear', kernel\_initializer='RandomNormal')      @tf.function      def call(self, inputs):          z = self.input\_layer(inputs)          for layer in self.hidden\_layers:              z = layer(z)          output = self.output\_layer(z)          return output |

tf.function을 사용한 것이 눈에 띈다. <https://www.tensorflow.org/guide/effective_tf2>의 설명에 따르면 기본 eager execution 으로 바뀐 TF2에서 그래프모드의 이점(좋은 성능!) 또한 챙기기 위해, @tf.function 데코레이터를 지원한다. 이걸로 감싸진 함수는 for/while문이 자동으로 tf.while\_loop으로 번역되는 등 파이썬 코드가 텐서플로우 코드로 변화되며, 이 함수를 하나의 그래프로 실행하기 위해 JIT 컴파일하기에 더 좋은 성능을 기대할 수 있다고 한다. 모든 함수를 감쌀 필요는 없고, 모델 훈련의 한 단계나 정방향 연산과 같은 고수준 연산에만 적용하라고 권장하고 있다.

tf.keras.Model을 상속받아 모델을 만들고, layer 클래스를 사용하는 것이 보인다. 왜 input\_shape 의 입력값은 (num\_states,) 이런 불완전한 튜플모양인가? batch axis를 포함하지 않는다는 의미를 나타내기 위함이 아닐까 추측해본다. 어디까지나 추측이다.

dense layer를 사용하는데, use\_bias=False로 지정해주면 matmul과 같은 효과를 나타낼 것이다.

kernel\_initializer도 일부러 random normal 쓸 필요 없이 기본 glorot uniform을 사용해보자.

hidden\_units에는 아무래도 튜플이나 리스트 같은 것을 받아서, hidden layer모양을 만들어 줄 것이다. (3,4,5)를 받으면 hidden layer가 각 layer당 3,4,5개 node를 가지고 생성되겠지.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | class DQN: | |  | def \_\_init\_\_(self, num\_states, num\_actions, hidden\_units, gamma, max\_experiences, min\_experiences, batch\_size, lr): | |  | self.num\_actions = num\_actions | |  | self.batch\_size = batch\_size | |  | self.optimizer = tf.optimizers.Adam(lr) | |  | self.gamma = gamma | |  | self.model = MyModel(num\_states, hidden\_units, num\_actions) | |  | self.experience = {'s': [], 'a': [], 'r': [], 's2': [], 'done': []} | |  | self.max\_experiences = max\_experiences | |  | self.min\_experiences = min\_experiences | |

DQN class를 만들어준다. 전반적인 attribute들. optimizer는 adam이 아닌 SGD로 바꿔줘야겠다.

experience는 아마 replay buffer역할인가?

**def** predict(self, inputs):

**return** self.model(np.atleast\_2d(inputs.astype('float32')))

전에 성능저하의 원인이었던 predict 메소드를 재정의해준다. 근데 이렇게 정의하면 또 eager tensor가 나오는 거 아니냐? atleast\_2d는 model이 default로 batch를 input으로 받아들이기 때문에, single state를 넣어도 at least 2 dimension을 확정해주기 위해 사용한다.

**def** train(self, TargetNet):

**if** len(self.experience['s']) < self.min\_experiences:

**return** 0

ids = np.random.randint(low=0, high=len(self.experience['s']), size=self.batch\_size)

states = np.asarray([self.experience['s'][i] **for** i **in** ids])

actions = np.asarray([self.experience['a'][i] **for** i **in** ids])

rewards = np.asarray([self.experience['r'][i] **for** i **in** ids])

states\_next = np.asarray([self.experience['s2'][i] **for** i **in** ids])

dones = np.asarray([self.experience['done'][i] **for** i **in** ids])

value\_next = np.max(TargetNet.predict(states\_next), axis=1)

actual\_values = np.where(dones, rewards, rewards+self.gamma\*value\_next)

**with** tf.GradientTape() **as** tape:

selected\_action\_values = tf.math.reduce\_sum(

self.predict(states) \* tf.one\_hot(actions, self.num\_actions), axis=1)

loss = tf.math.reduce\_mean(tf.square(actual\_values - selected\_action\_values))

variables = self.model.trainable\_variables

gradients = tape.gradient(loss, variables)

self.optimizer.apply\_gradients(zip(gradients, variables))

**return** loss

train할 때 minibatch를 만든다. replay buffer에 최소량이 차 있지 않으면 train을 하지 않으며, batch size만큼 랜덤한 index를 뽑아 state, action, reward, state\_next, done 배열을 만들어 준다. np.where이 생소한데, <https://pinkwink.kr/1236> 에 의하면 조건에 맞는 index를 꺼내는게 주요 역할이며, 또는 조건에 맞을 시 x, 아닐 시 y로 변화시키도록 해주는 기능도 수행한다. for문보다 빠르다고 한다. 따라서 위의 where문은 해석하면 array를 출력하는 데 done =1 위치의 index에는 동일 index의 reward를 출력하고 아닐 시 reward + gamma\*value\_next를 넣으라는 의미이다.

GradientTape가 나왔다. 그때 그 블랙박스. selected\_action\_values를 구한다. 약간 복잡해 보이는데, 구체적으로 각각의 모양이 어떻게 나오길래 저 식으로 selected\_action\_values가 나오는 것일까. 간단하다. action을 one\_hot해 마스크로 만들어 곱해주면 실제 선택으로 인해 얻은 reward만 나오게 된다. 이걸 axis=1로 reduce sum한다는 것은 무슨 의미인가? 0은 열단위로, 1은 행단위로 더한다는 의미이다. 2\*3 배열 : 2행 3열의 의미. 0 은 1행으로 만들고, 1은 1열로 만든다는 의미?

|  |  |  |
| --- | --- | --- |
| [[1 | 2 | 3] |
| [4 | 5 | 6]] |

모양 문제는 언제나 골치가 아프다……

어찌되었건 loss를 구하고 variable의 loss에 대한 gradient를 구하고, 이걸 optimizer에 적용해준다. zip(gradients, variables)는 무엇? itertool로서 동일 개수 자료형을 튜플로 묶어 반환하는 iterator를 만든다. 왜 이걸 쓰는가? 그냥 이런 형태를 받는 게 apply\_gradient 메서드인가보다.

**def** get\_action(self, states, epsilon):

**if** np.random.random() < epsilon:

**return** np.random.choice(self.num\_actions)

**else**:

**return** np.argmax(self.predict(np.atleast\_2d(states))[0])

E-greedy 방식으로 action 선택하는 부분이다. predict에 최소 2차원 넣어주는 것 잊지 말자.

**def** add\_experience(self, exp):

**if** len(self.experience['s']) >= self.max\_experiences:

**for** key **in** self.experience.keys():

self.experience[key].pop(0)

**for** key, value **in** exp.items():

self.experience[key].append(value)

**def** copy\_weights(self, TrainNet):

variables1 = self.model.trainable\_variables

variables2 = TrainNet.model.trainable\_variables

**for** v1, v2 **in** zip(variables1, variables2):

v1.assign(v2.numpy())

experience dictionary, 즉 replay buffer에 추가값 넣어주는 방법 (용량이 꽉 찼으면 pop하고 넣어줌)

copy\_weights. variable 복사.variable 자체는 assign 인자로 쓸 수 없으므로 numpy()를 사용.

**def** play\_game(env, TrainNet, TargetNet, epsilon, copy\_step):

rewards = 0

iter = 0

done = **False**

observations = env.reset()

losses = list()

**while** **not** done:

action = TrainNet.get\_action(observations, epsilon)

prev\_observations = observations

observations, reward, done, \_ = env.step(action)

rewards += reward

**if** done:

reward = -200

env.reset()

exp = {'s': prev\_observations, 'a': action, 'r': reward, 's2': observations, 'done': done}

TrainNet.add\_experience(exp)

loss = TrainNet.train(TargetNet)

**if** isinstance(loss, int):

losses.append(loss)

**else**:

losses.append(loss.numpy())

iter += 1

**if** iter % copy\_step == 0:

TargetNet.copy\_weights(TrainNet)

**return** rewards, mean(losses)

실제 게임을 플레이하며 트레이닝 하는 메소드. loss가 int인지 아닌지는 왜 묻는것인가?

이부분은 도저히 이해가 안된다. 왜 이런 코드를 넣었지? loss는 애초에 int일 리가 없을텐데? 아, 하나 있다. experience가 일정량 안 차서 학습되지 않고 0을 반환하는 경우. 이때는 그냥 loss = 0을 넣어주고, 아닌 경우에는 tensor로 나오니까 numpy()를 해줘서 넣는다는 뜻 인듯?

**def** make\_video(env, TrainNet):

env = wrappers.Monitor(env, os.path.join(os.getcwd(), "videos"), force=**True**)

rewards = 0

steps = 0

done = **False**

observation = env.reset()

**while** **not** done:

action = TrainNet.get\_action(observation, 0)

observation, reward, done, \_ = env.step(action)

steps += 1

rewards += reward

print("Testing steps: **{}** rewards **{}**: ".format(steps, rewards))

테스트를 비디오로 저장.

**def** main():

env = gym.make('CartPole-v0')

gamma = 0.99

copy\_step = 25

num\_states = len(env.observation\_space.sample())

num\_actions = env.action\_space.n

hidden\_units = [200, 200]

max\_experiences = 10000

min\_experiences = 100

batch\_size = 32

lr = 1e-2

current\_time = datetime.datetime.now().strftime("%Y%m**%d**-%H%M%S")

log\_dir = 'logs/dqn/' + current\_time

summary\_writer = tf.summary.create\_file\_writer(log\_dir)

TrainNet = DQN(num\_states, num\_actions, hidden\_units, gamma, max\_experiences, min\_experiences, batch\_size, lr)

TargetNet = DQN(num\_states, num\_actions, hidden\_units, gamma, max\_experiences, min\_experiences, batch\_size, lr)

N = 50000

total\_rewards = np.empty(N)

epsilon = 0.99

decay = 0.9999

min\_epsilon = 0.1

**for** n **in** range(N):

epsilon = max(min\_epsilon, epsilon \* decay)

total\_reward, losses = play\_game(env, TrainNet, TargetNet, epsilon, copy\_step)

total\_rewards[n] = total\_reward

avg\_rewards = total\_rewards[max(0, n - 100):(n + 1)].mean()

**with** summary\_writer.as\_default():

tf.summary.scalar('episode reward', total\_reward, step=n)

tf.summary.scalar('running avg reward(100)', avg\_rewards, step=n)

tf.summary.scalar('average loss)', losses, step=n)

**if** n % 100 == 0:

print("episode:", n, "episode reward:", total\_reward, "eps:", epsilon, "avg reward (last 100):", avg\_rewards,

"episode loss: ", losses)

print("avg reward for last 100 episodes:", avg\_rewards)

make\_video(env, TrainNet)

env.close()

rewards += reward

print("Testing steps: **{}** rewards **{}**: ".format(steps, rewards))

summary writer를 만들고…이때 log directory는 ssd가 있는 c:드라이브로 옮겨줘야지.

문제점이 보인다, 최대 실행수가 200이라는 점?

그래서 학습을 돌려보는 중인데, 아무런 반응이 없다. 왜죠? hidden layer가 너무 많아서?

아니, 다시 보니까 epsilon이 0.99에서 시작하네. 실화냐? 대체 왜…?

그래서 학습이 잘 안되나?

내가 뭔가를 잘못 바꿨나 싶어서 오리지널 코드를 써봤다. 그래도 학습이 안된다. 속은 것이다…