팡요랩 강의 내용:

actor-critic은 MCPG에서 실제 Reward를 Q에 대한 sample로 썼었는데 그러지 말고, Q 자체도 w로 parameterize하여 학습시키자는 발상에서 출발한다. 보다 보면 기시감이 들 수 있는데, policy iteration과 매우 유사하다고 볼 수 있다.

baseline : 99만과 100만 reward를 그대로 사용하면 차이가 별로 안 느껴진다. ->학습이 비효율적이다. 99만 5000을 각각 빼서 학습을 시켜보자. 수학적으로 맞는 이야기인가? 맞다. s에만 관련한 식인 value function을 빼 주어서 평균 값은 변화 없이, variance만 줄일 수 있다. Q-V를 **advantage함수라고 한다.**

구체적으로 무엇을 이걸로 학습해줘야 하는가? Actor? Critic? Critic을 이걸로 학습시킨다.

여기서 에바참치 발생. V에 대한 파라미터까지 학습시켜야 한다? 오우 쉣.

결론적으로, Q는 안 필요하다. V만으로 advantage를 계산할 수 있다. TD error가 advantage function의 unbiased sample이다. 왜냐? Q(s,a) = r + \gammaV(s’)니까. 마법처럼 TD error만을 신경쓰면 되는 식이 된다.

구현 강의 내용

AutoDiff? loss를 -r\*log\pi로 지정해 주면 된다. 우리가 구해야 하는 것은 r\*log\pi의 미분값.

actor critic code 분석해보기

|  |
| --- |
| import gym  import torch  import torch.nn as nn  import torch.nn.functional as F  import torch.optim as optim  from torch.distributions import Categorical  #Hyperparameters  learning\_rate = 0.0002  gamma         = 0.98  n\_rollout     = 10  class ActorCritic(nn.Module):      def \_\_init\_\_(self):          super(ActorCritic, self).\_\_init\_\_()          self.data = []            self.fc1 = nn.Linear(4,256)          self.fc\_pi = nn.Linear(256,2)          self.fc\_v = nn.Linear(256,1)          self.optimizer = optim.Adam(self.parameters(), lr=learning\_rate)        def pi(self, x, softmax\_dim = 0):    #입력 차원에 따라 dim을 바꾼다          x = F.relu(self.fc1(x))          x = self.fc\_pi(x)          prob = F.softmax(x, dim=softmax\_dim)          return prob        def v(self, x):          x = F.relu(self.fc1(x))          v = self.fc\_v(x)          return v        def put\_data(self, transition):          self.data.append(transition)        def make\_batch(self):          s\_lst, a\_lst, r\_lst, s\_prime\_lst, done\_lst = [], [], [], [], []          for transition in self.data:              s,a,r,s\_prime,done = transition              s\_lst.append(s)              a\_lst.append([a])              r\_lst.append([r/100.0])   #왜 100으로 나눠줬나? normalization?              s\_prime\_lst.append(s\_prime)              done\_mask = 0.0 if done else 1.0              done\_lst.append([done\_mask])  #done\_mask의 존재 이유? 곱해주려고.            s\_batch, a\_batch, r\_batch, s\_prime\_batch, done\_batch = torch.tensor(s\_lst, dtype=torch.float), torch.tensor(a\_lst), \                                                                 torch.tensor(r\_lst, dtype=torch.float), torch.tensor(s\_prime\_lst, dtype=torch.float), \                                                                 torch.tensor(done\_lst, dtype=torch.float)          self.data = []          return s\_batch, a\_batch, r\_batch, s\_prime\_batch, done\_batch        def train\_net(self):          s, a, r, s\_prime, done = self.make\_batch()  #torch.tensor형태로 받아 온다. 동시에 데이터 비움          td\_target = r + gamma \* self.v(s\_prime) \* done  #done=true였다면 TD target = r. 일괄 계산          delta = td\_target - self.v(s)                   #delta 일괄 계산.            pi = self.pi(s, softmax\_dim=1)                  #policy 계산 (마지막에 행마다 softmax해주기)          pi\_a = pi.gather(1,a)   #행 기준으로, index a 부분만 모으기 = pi(s,a)          #detach() : backpropagation할 때 미분에 포함되는 것 방지. .item()효과          #smooth\_l1\_loss : loss function중의 하나로 ㅣ1 loss와 ㅣ2 loss의 중간적인 느낌          loss = -torch.log(pi\_a) \* delta.detach() + F.smooth\_l1\_loss(self.v(s), td\_target.detach())          self.optimizer.zero\_grad()          loss.mean().backward()          self.optimizer.step()    def main():      env = gym.make('CartPole-v1')      model = ActorCritic()      print\_interval = 20      score = 0.0      for n\_epi in range(10000):          done = False          s = env.reset()          while not done:              for t in range(n\_rollout):                  prob = model.pi(torch.from\_numpy(s).float())    #policy 계산                  m = Categorical(prob)                           #sampling                  a = m.sample().item()                           #...                  s\_prime, r, done, info = env.step(a)            #...                  model.put\_data((s,a,r,s\_prime,done))            #데이터 저장                    s = s\_prime                  score += r                    if done:                      break                model.train\_net()                                   #끝, 또는 rollout 다 되면 학습            if n\_epi%print\_interval==0 and n\_epi!=0:              print("# of episode :{}, avg score : {:.1f}".format(n\_epi, score/print\_interval))              score = 0.0      env.close()  if \_\_name\_\_ == '\_\_main\_\_':      main() |
|  |

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

DPG로 넘어가 보자!

<https://reinforcement-learning-kr.github.io/2018/06/27/2_dpg/>

<https://talkingaboutme.tistory.com/entry/RL-Policy-Gradient-Algorithms>

DPG는 Expected gradient of the action-value function의 형태이다.

Policy variance가 0에 수렴할 경우 DPG는 Stochastic Policy Gradient와 동일해진다.

적절한 exploration을 위해 model-free, off policy actor critic algorithm을 제안한다.

DPG는 SPG보다 성능이 좋은데, 특히 high dimensional action spaces를 가지는 tasks에서의 성능 향상이 크다.

* SPG의 policy gradient는 state와 action spaces 모두에 대해서, DPG의 policy gradient는 state spaces에 대해서만 평균을 취합니다.
* 결과적으로, action spaces의 dimension이 커질수록 data efficiency가 높은 DPG의 학습이 더 잘 이뤄지게 됩니다.
* 무한정 학습을 시키면, SPG도 최적으로 수렴할 것으로 예상되기에 위 성능 비교는 일정 iteration 내로 한정합니다.

>> 결국 빠르게 수렴할 뿐, 최종 성능을 비교했을 땐 별 차이 없다는 뜻.?

기존 기법들에 비해 computation 양이 많지 않습니다.

* Computation 은 action dimensionality 와 policy parameters 수에 비례합니다.

Off-policy Actor-Critic 라는 개념이 등장한다. \pi 가 아닌 distinct behavior policy \beta로부터 샘플링된 trajectories를 이용한 Actor-critic이다. 왜 이게 필요한 걸까?

이걸 사용하면 지난번 경험을 저장해둬서 **experience replay**를 활용할 수 있다. sample efficiency 측면에서 유리하다.

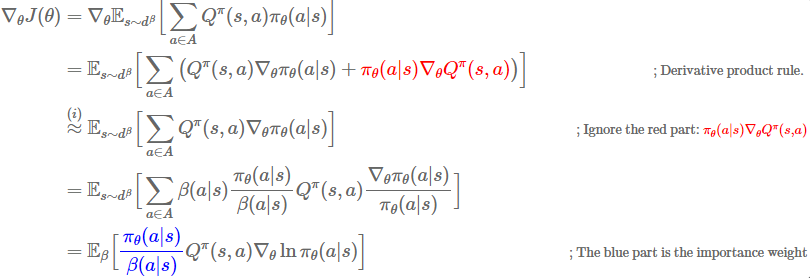
exploration 측면에도 좋다.

기존 performance objective function은 다음과 같았다.

dads가 뭐지? da ds구나. 한참 구글링했네 ㅋㅋㅋㅋㅋ

behavior function을 이용한 POF는 다음과 같다. distribution function은 \beta를 이용하고, Q값은 \pi를 이용함에 주목하라.

gradient를 구하기 위해 양 변을 미분하자.



두번째 줄 식의 빨간 부분은 구하기가 영 까다롭다. 정확한 워딩으로는 “difficult to estimate in incremental off policy setting”이라고 한다. 애초에 Q\pi의 gradient를 구하려는 시도는 지금까지 안해봤던 것 같다. 그래서 과감하게 **삭제하고** , 근사 식을 만든다. 이후, importance sampling ratio를 이용해 식을 맨 아랫줄처럼 바꾼다. 이 식으로도 수렴한다고 한다. 증명은

<https://arxiv.org/pdf/1205.4839.pdf> 이 논문에 되어있다.

논문 해석 :

abstract

In this paper we consider deterministic policy gradient algorithms for reinforcement learning with continuous actions.

Continuous action을 가진 강화학습에 사용하는 알고리즘이다.

The deterministic policy gradient has a particularly appealing form: it is the expected gradient of the action-value function.

특이한 형태? 매력적인 형태?가 있는데, action value function의 gradient 기대값이다.

This simple form means that the deterministic policy gradient can be estimated much more efficiently than the usual stochastic policy gradient.

이게 뭘 의미하냐면 SGD보다 훨씬 효과적이라는 뜻이다. (왜죠…?)

To ensure adequate exploration, we introduce an off-policy actor-critic algorithm that learns a deterministic target policy from an exploratory behaviour policy.

충분한 exploration을 보장하기 위해, off policy actor-critic algorithm을 소개한다. 이것은 deterministic target policy를 탐색적 behavior policy로부터 학습한다.

We demonstrate that deterministic policy gradient algorithms can significantly outperform their stochastic counterparts in high-dimensional action spaces.

DPG 알고리즘이 SGD알고리즘보다 high-dimensional action space에서 눈에 띄게 좋은 성능을 보임을 입증한다./시범으로 보인다?

Policy gradient algorithms are widely used in reinforcement learning problems with continuous action spaces. The basic idea is to represent the policy by a parametric probability distribution that stochastically selects action a in state s according to parameter vector .

PG는 continuous action space 문제에 널리 사용되어왔다.

Policy gradient algorithms typically proceed by sampling this stochastic policy and adjusting the policy parameters in the direction of greater cumulative reward.

PG algorithm은 이 stochastic한 policy를 sampling하고 누적 reward가 높아지는 방향으로 policy parameter를 보완해가는 식으로 일반적으로 진행된다.

In this paper we instead consider deterministic policies a = \mu\_\theta(s). It is natural to wonder whether the same approach can be followed as for stochastic policies: adjusting the policy parameters in the direction of the policy gradient. It was previously believed that the deterministic policy gradient did not exist, or could only be obtained when using a model (Peters, 2010).

동일한 방법이 DPG에도 적용될 수 있을지 궁금한 것이 당연하다. DPG는 존재하지 않거나 model을 사용할 때나 가질 수 있는 것으로 여겨져 왔다.

However, we show that the deterministic policy gradient does indeed exist, and furthermore it has a simple model-free form that simply follows the gradient of the action-value function.

하지만 있다는 것을 우린 보인다. 게다가 action-value function의 gradient를 따르는 간단한 model form도 가지고 있다.

In addition, we show that the deterministic policy gradient is the limiting case, as policy variance tends to zero, of the stochastic policy gradient.

그리고 우리는 이 DPG가 SPG에서 policy variance가 0인 특수 케이스임을 보인다.

From a practical viewpoint, there is a crucial difference between the stochastic and deterministic policy gradients. In the stochastic case, the policy gradient integrates over both state and action spaces, whereas in the deterministic case it only integrates over the state space.

실용적 관점에서는 중요한 차이가 있는데, SPG에서는 policy gradient가 state 와 action space 양쪽에서 모아지는 반면, DPG는 state space에서만 모은다.

-이게 무슨 이야기인가? SPG도 baseline을 사용하면 state space만 평균내 학습하면 되는 것 아니었나? … pi(s,a)를 사용하니까?

As a result, computing the stochastic policy gradient may require more samples, especially if the action space has many dimensions.

결과적으로 SPG는 특히 action space가 dimension이 높을 때 더 많은 샘플이 필요할 수 있다.

In order to explore the full state and action space, a stochastic policy is often necessary. To ensure that our deterministic policy gradient algorithms continue to explore satisfactorily, we introduce an off-policy learning algorithm.

우리의 DPG가 만족스러운 수준의 explore를 하기 위해 off policy learning algorithm을 제안한다.

The basic idea is to choose actions according to a stochastic behaviour policy (to ensure adequate exploration), but to learn about a deterministic target policy (exploiting the efficiency of the deterministic policy gradient).

action은 stochastic behavior policy로 선택하지만 학습하는 것은 deterministic target policy이다.

We use the deterministic policy gradient to derive an off-policy actorcritic algorithm that estimates the action-value function using a differentiable function approximator, and then updates the policy parameters in the direction of the approximate action-value gradient. We also introduce a notion of compatible function approximation for deterministic policy gradients, to ensure that the approximation does not bias the policy gradient.

action value function을 미분가능한 function approximator로 estimate하고, policy parameter를 approximate action value gradient 방향으로 업데이트 하기 위해 DPG를 사용한다. compatible function approximation을 사용해서 approximation이 policy gradient를 bias하지 않음을 보장한다.

-첫번째 문장은 SGD에서도 하는 일 아닌가? 그리고 아까는 state space만 다룰 것처럼 굴더니, action value function을 또 estimate하려고 한다네. 모순 아닌가?

We apply our deterministic actor-critic algorithms to several benchmark problems: a high-dimensional bandit; several standard benchmark reinforcement learning tasks with low dimensional action spaces; and a high-dimensional task for controlling an octopus arm. Our results demonstrate a significant performance advantage to using deterministic policy gradients over stochastic policy gradients, particularly in high dimensional tasks.

여러 benchmark problem에 적용해봤고 SGD보다 눈에 띄는 성능향상이 있었는데 특히 high dimensional task에서 잘 동작하는 것을 확인함.

Furthermore, our algorithms require no more computation than prior methods: the computational cost of each update is linear in the action dimensionality and the number of policy parameters.

또 computation 비용도 줄었는데, action dimensionality와 policy parameter개수에 선형적으로 증가한다.

Finally, there are many applications (for example in robotics) where a differentiable control policy is provided, but where there is no functionality to inject noise into the controller. In these cases, the stochastic policy gradient is inapplicable, whereas our methods may still be useful.

미분가능한 control policy는 주어졌지만 controller에 noise를 주입할 수는 없는 경우가 있다. SGD는 적용 불가능 하지만 DPG는 여전히 유용할 수 있다.

… compatible function approximation이 정확히 뭘 말하는 건지, 웬만하면 성립하는 것인지, 뭔지 잘 모르겠음

DPG가 무엇인지 느낌적으로만 짐작이 가다가 또 오리무중.

objective function을

DPG theorem은 다음이 성립함을 보이는 것이다.



……………………

외계어, 외계어. PG를 그렇게 오래 봤는데, DPG 논문 이해에 도움되는 것 같지가 않다. 부분부분 띄엄띄엄 눈에 들어오는 것만 있을 뿐이다. importance sampling ratio 라던지, off policy actor critic이라던지… 그마저도 정확하게 이해하는 것도 아니다. 강의를 들어보아야 할까?

DDPG : 중간보고서에 쓴 내용 그대로 복사

DDPG 이전 가장 주목받았던 학습법은 DQN으로 사람 수준 또는 그 이상의 Atari game 플레이 실력을 학습시킨 예시가 유명했다. pixel단위의 input을 받아 Q function 값을 도출해 낼 만큼 high dimensional observation space에서 뛰어난 모습을 보였지만 문제는 분절되고 낮은 dimension의 action space에 한정된 기능이었다. 연구자들은 높은 차원의 action space에 유리한 DPG에 DQN을 접목하여 continuous action space 문제까지 다룰 수 있는 algorithm인 DDPG를 제안하였다.

DPG에서 성공적이었던 actor critic과 DQN에서 성공적이었던 Replay buffer, target Q network가 DDPG에 적용되었다. Target Q network를 업데이트할 때 DQN에서는 일정 step마다 한번에 복사하는 방식으로 갱신했지만, DDPG에서는 soft target update를 적용하여 스텝마다 느린 속도로 original network를 따라잡도록 한다. 또한 scale이 다른 feature를 다룰 때 생기는 일반화의 어려움을 해소하기 위해 batch normalization을 적용하며, exploration을 위한 noise를 actor의 행동에 더해 critic에 넣는다.

대략적인 알고리즘은 다음과 같다. critic network 와 actor 를 초기화하고 각각의 target network , 를 초기화한다. 루프의 시작과 함께 action을 deterministic한 actor에서 받아오고, exploration을 위한 noise를 더한다. action을 수행하고 reward와 새로운 state를 받아온 뒤, 이를 replay buffer에 저장한다. Buffer에 일정 수의 transition이 차게 되면 random minibatch를 뽑은 뒤, 이들로 구한 Loss의 평균을 최소화 하는 방향으로 critic을 Q learning처럼 업데이트 한다. 이후 DPG에서 actor를 업데이트 하듯 gradient를 구해 actor policy를 갱신한다. 이후 target network를 soft target update 해준 뒤 루프를 반복한다.

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

최근 결과로는 actor가 아닌 state에 noise를 주는 것이 성능이 더 좋다고 한다.

<https://openai.com/blog/better-exploration-with-parameter-noise/>

TORCS 환경 : gym-torcs. 리눅스 환경에서만 쓸 수 있나?

linux를 설치해야 한다면, github 폴더는…? 옮겨야 하나?

-----------

하루 꼬박을 투자해 virtualbox로 우분투 설치하고, 설정하고, 거기에 gym\_torcs설치했다.

문제는 그래픽카드를 virtualbox에 연동 못한다는 것.

역시나, 렉이 뚝뚝 끊기는 모습이다.

-----------

<https://jayhey.github.io/deep%20learning/2019/01/08/WorldModels_1/>

world모델에 대해 알기 쉽게 해설 해놓은 글