Policy Gradients - An Alternative

한양대학교 최솔비

Values and policy

- Q-learning : state(value iteration method)나 action + state(Q-learning) 의 value(discounted total reward)를 계산하고, 그 값이 가장 크도록(greedily) 행동한다.

  Bellman equation 사용: 다음 step의 value 를 가지고 현재 step의 value 를 표현할 수 있음

- policy: 모든 state에서 어떻게 행동할지를 결정하는 요소.
- Q-learning에서는 value 자체가 policy

  $\pi(s) = argmax_a Q(s, a)$

  각 state 에서의 policy π 는 가장 큰 Q 를 갖는 action.

Why policy?

- action의 개수가 많은 environment 혹은 continuous action space의 environment 에서는 각각 action의 value를 따지는 것이 어려움.
- neural network 로 q value 를 표현하려고 한다면, highly nonlinear 하게 되어 argument 를 찾는 것이 어려워짐
  -> policy 를 구하는 것이 더 현실적.
- stochasticity : categorical DQN에서처럼 action의 확률로 나타낼 수 있다.

Policy representation

- state 를 neural network에 넣었을 때 action을 리턴하는 것이 아니라 action의 probability distribution 을 리턴하도록 함.
- smooth representation의 장점: 기존의 discrete output에서는 network weight 을 약간 만 바꾸어도 결과가 많이 바뀌어 다른 action을 취하는 경우가 생김.

  output 이 probability distribution 이라면, weight을 약간 바꾸면 output 도 약간만 바뀜.
- Gradient optimization method 에서 네트워크 모델의 파라미터를 조정해 결과를 향상 시키는 것이 쉬워짐.

Policy gradients

- $L = -Q(s, a) \log \pi(a|s)$
- (value of the action taken) * (probability of the action taken)의 합이 최대가 되도록!
- 전체 리워드를 많이 가져오는 action의 확률을 높이고, 적은 리워드를 가져오는 action의 확률을 낮추게 된다.
- 위 공식을 loss 함수로 이용하여 network를 업데이트 한다.
- stochastic gradient descent 방법은 loss 함수를 최소화하는 방식으로 optimize 하기 때문에 minus sign 잊지 말고 붙여야 (value) * (probability) 가 최대가 됨.

The REINFORCE method

- gradient의 scale을 계산하는 다양한 방법이 있다.
- Cross -entropy 방법은 scale을 elite episode 에 대해서는 1로, 나머지 episode에 대해서는 0으로 설정한 것.
- 이 값을 더 자세하게 나눈다면(ex. $Q(s, a)$, expected reward), episode 를 더 다양하게 분류할 수 있다.
- 예를 들면 total reward 가 10 인 에피소드의 transition을 total reward가 1 인 에피소드의 transition 보다 gradient 에 많이 반영한다.
- The second reason to use $Q(s, a)$ instead of just 0 or 1 constants is to increase probabilities of good actions in the beginning of the episode and decrease the actions closer to the end of episode. (?)

The REINFORCE method steps

1. network를 랜덤한 weight으로 초기화한다.
2. N 번의 에피소드를 수행하며 (s, a, r, s') 을 저장한다.
3. 모든 에피소드 k의 step t 에 대해서 다음 스텝들에 대한 discounted total reward를 계산한다.

   $Q_{k,t} = \sum_{i=0} \gamma^i r_i$
4. 모든 transitions 에 대하여 로스 함수를 계산한다.

   $L = -\sum_{k,t} Q_{k,t} \log(\pi(s_{k,t}, a_{k,t}))$
5. 로스 함수를 최소화하도록 network weight을 SGD 업데이트 한다.
6. 수렴할 때까지 2번부터 반복.

Q-learning 과 다른점

- exploration을 따로 할 필요가 없다. (epsilon greedy 안써도 됨.) network가 확률을 리턴하기 때문에 exploration과 exploitation 의 비율을 알아서 맞춰준다.
- replay buffer를 쓰지 않는다. PG method 는 on-policy method에 속하기 때문에 예전 policy로 얻은 데이터를 학습시킬수 없다. 더 빠르게 수렴한다는 장점이 있지만, DQN같은 off-policy method 보다 environment와 더 자주 상호작용해야한다.
- target network가 필요없다. Q-value 가 experience 에서 얻어지기 때문

Cartpole example

```
GAMMA = 0.99
LEARNING_RATE = 0.01
EPISODES_TO_TRAIN = 4 #4개씩 train 시킴
```

```
class PGN(nn.Module):
    def __init__(self, input_size, n_actions):
        super(PGN, self).__init__()

        self.net = nn.Sequential(
            nn.Linear(input_size, 128),
            nn.ReLU(),
            nn.Linear(128, n_actions)
        )

    def forward(self, x):
        return self.net(x)
```

- network가 리턴하는 logits(raw score) 값을 probability로 만들어 주기 위한 softmax는 뒤에서 적용함.
- `log_softmax` 는 softmax 를 한 후 log를 씌워 반환. numerically stable.

```
def calc_qvals(rewards):
    res = []
    sum_r = 0.0
    for r in reversed(rewards):
        sum_r *= GAMMA
        sum_r += r
        res.append(sum_r)
    return list(reversed(res))
```

- 전체 episode 의 reward를 받아 뒤에서 부터 거꾸로 discounted reward 를 계산한다.
- $r_{t-1} + \gamma r_t$

main 함수

```
total_rewards = []
step_idx = 0
done_episodes = 0

batch_episodes = 0
batch_states, batch_actions, batch_qvals = [], [], []
cur_rewards = []
# 현재 episode의 local rewards
# 에피소드가 끝나면 discount reward 계산하여(calc_qvals)
# batch_qvals 에 추가한다.
```

```
for step_idx, exp in enumerate(exp_source):
    batch_states.append(exp.state)
    batch_actions.append(int(exp.action))
    cur_rewards.append(exp.reward)

    if exp.last_state is None:
        batch_qvals.extend(calc_qvals(cur_rewards))
        cur_rewards.clear()
        batch_episodes += 1
    ...
    if batch_episodes < EPISODES_TO_TRAIN:
        continue
```

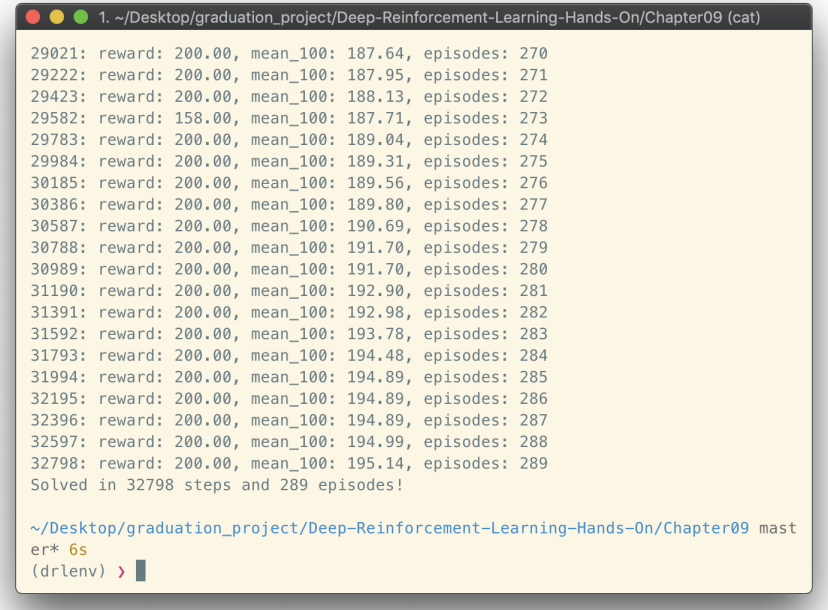
- episode 가 끝나면, reward 를 Q-value 로 바꾼다.

```
optimizer.zero_grad()
states_v = torch.FloatTensor(batch_states)
batch_actions_t = torch.LongTensor(batch_actions)
batch_qvals_v = torch.FloatTensor(batch_qvals)
#  파이토치  텐서로  변환
logits_v = net(states_v)
log_prob_v = F.log_softmax(logits_v, dim=1)
log_prob_actions_v = batch_qvals_v *
                log_prob_v[range(len(batch_states)),
                           batch_actions_t]
loss_v = -log_prob_actions_v.mean()

loss_v.backward()
optimizer.step()
```

- 네트워크에서 logits 형태로 아웃풋을 받아 `log_softmax` 함수로 log probability 형태로 바꾼다.
  (logarithm + softmax)
- loss value 를 계산할때 잊지말고 마이너스 붙이자.(기대 reward를 최대로 만들기 위해서)

REINFORCE: cartpole 결과



REINFORCE: cartpole 결과

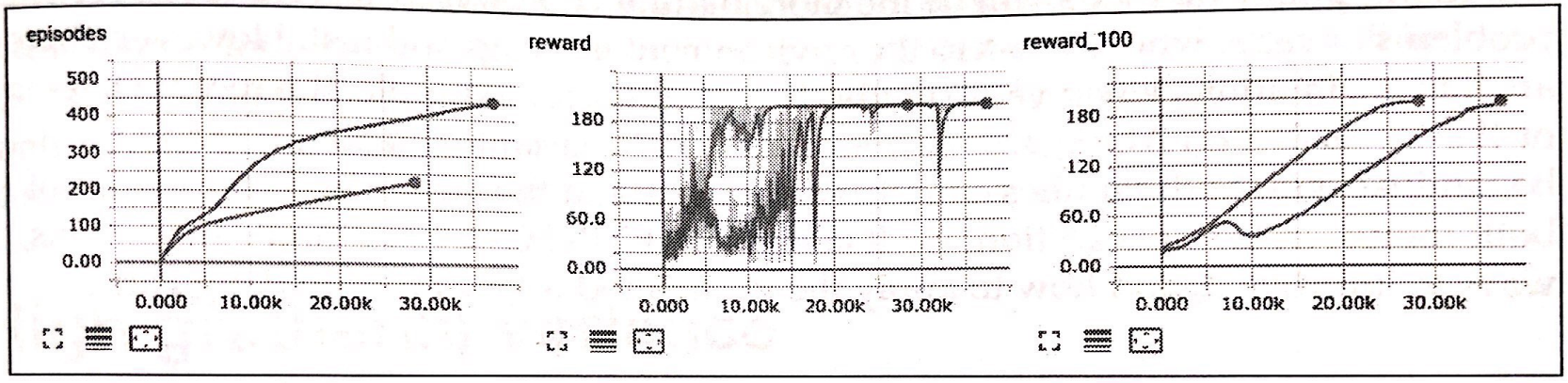
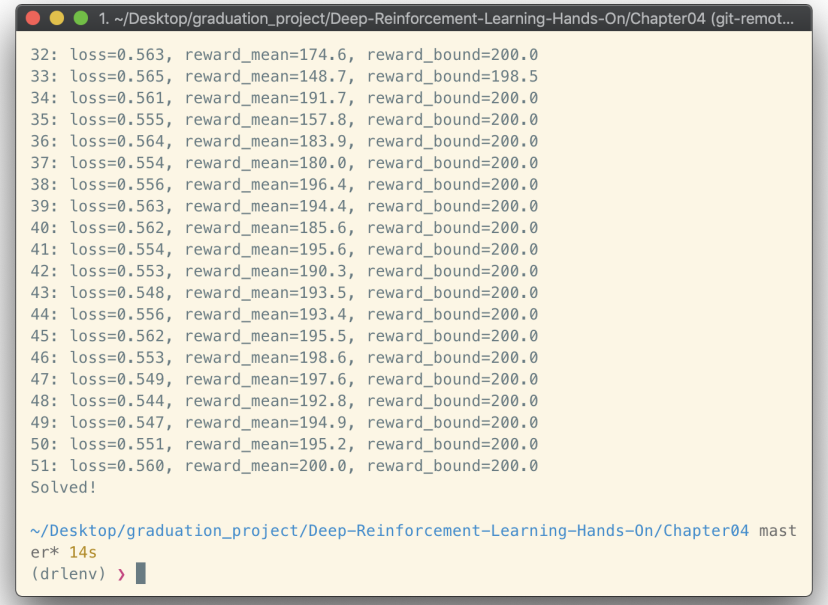


Figure 2: Convergence of DQN (orange) and REINFORCE (blue line)

- REINFORCE가 DQN 보다 적은 training steps(및 episodes)으로 더 빠르게 수렴한다.

같은 문제를 cross-entropy 로 풀었을 때



- cross-entropy 는 (16 episodes) * (51 batches) = 816
- REINFORCE 는 289 episodes.

Policy-based vs value-based methods

- Policy methods
  - agent의 행동을 직접적으로 optimize.
  - on-policy. old data 는 사용 못함.
  - less sample-efficient. environment 와 계속 상호작용하며 fresh sample 을 얻어야함.
  - continuous control 문제, environment 에 접근 하는 것이 쉽고 빠를 때는 policy method.
- Value methods
  - 간접적으로 optimize. value 를 먼저 계산한후, policy 를 제공.
  - old data 로부터도 학습할 수 있음. replay buffer 사용
  - sample-efficient 하지만 계산양이 더 많을 수 있다.

REINFORCE issues

- Full episodes are required
- High gradients variance
- Exploration
- Correlation between samples

REINFORCE issues

Full episodes are required

- 에피소드 길이가 길어지면 적용하기 어려움. 한번 학습시키기 위해 environment 와 여러번 communicate 해야하는 것이 비효율적.
- DQN 에서는 정확한 discounted reward 가 아니라 Bellman equation으로 approximate 한 값을 이용했다.

  $$Q(s, a) = r_a + \gamma V(s')$$

- 이를 해결하기 위해

  i. V(s)를 추정하여 이를 가지고 Q 값을 구한다. (actor-critic)

  ii. Bellman equation unrolling 해서 N step 앞까지만 본다.

High gradients variance

- Policy gradient 공식에서 scale(Q(s,a))의 분산 문제.
- scale의 범주가 environment 에 따라 다르다.
- 예를 들어 cartpole 에서 어떤 에피소드의 리워드는 5, 어떤 것은 100 일 것이다. PG method 에 따르면 뒤의 것이 20배 더 가중치를 있게 학습 된다. 분산이 너무 크다면 일부 에피소드가 학습을 독식할 수 있다.
- 학습 과정이 unstable 해지는 것을 막기 위해 Q에서 baseline 이라는 값을 빼주도록 한다.

Exploration

- probability를 사용하더라도 local optimum에 빠질수 있다.
- DQN 에서는 이를 해결하기 위해 epsilon-greedy를 사용했는데, PG 에서는 entropy를 이와 유사하게 사용한다.
- 엔트로피는 불확실성을 나타냄: agent가 어떤 action을 취할 것인지 모름.
- $H(\pi) = -\sum \pi(a|s) \log \pi(a|s)$
- 엔트로피값은 항상 0보다 크고, uniform distribution 일 때 single maximum
- 엔트로피 값을 loss 함수로부터 뺀다-> 너무 확실하게(certain) 행동하는 것을 막는다.

Correlation between samples

- on-policy 이기 때문에 replay buffer 를 사용할 수 없음 (old data 사용 불가)
- parallel environments 를 사용하여 해결. 여러개의 environment 를 동시에 사용하여 이들의 transition 을 학습에 이용한다.

Cartpole again.

```
GAMMA = 0.99
LEARNING_RATE = 0.001
ENTROPY_BETA = 0.01
# 엔트로피 값을 얼마나 적용할 건지
BATCH_SIZE = 8

REWARD_STEPS = 10
# Bellman equation 몇 스텝 unroll 할건지
```

- 네트워크는 동일

```
exp_source = ptan.experience.ExperienceSourceFirstLast(
env, agent, gamma=GAMMA, steps_count=REWARD_STEPS)
```

- Bellman equation unroll

```
for step_idx, exp in enumerate(exp_source):
  reward_sum += exp.reward
  baseline = reward_sum / (step_idx + 1)
  #policy scale 에 대한 baseline 계산
  writer.add_scalar("baseline", baseline, step_idx)
  batch_states.append(exp.state)
  batch_actions.append(int(exp.action))
  batch_scales.append(exp.reward - baseline)
```

- entropy bonus 부분 추가

```
states_v = torch.FloatTensor(batch_states)
batch_actions_t = torch.LongTensor(batch_actions)
batch_scale_v = torch.FloatTensor(batch_scales)

optimizer.zero_grad()
logits_v = net(states_v)
log_prob_v = F.log_softmax(logits_v, dim=1)
log_prob_actions_v = batch_scale_v * log_prob_v[
                range(BATCH_SIZE), batch_actions_t]
loss_policy_v = -log_prob_actions_v.mean()
# 여기까지는 똑같음
prob_v = F.softmax(logits_v, dim=1)
entropy_v = -(prob_v * log_prob_v).sum(dim=1).mean()
entropy_loss_v = -ENTROPY_BETA * entropy_v
loss_v = loss_policy_v + entropy_loss_v
# 엔트로피를 빼준다.
loss_v.backward()
optimizer.step()
```

```
# calc KL-div
new_logits_v = net(states_v)
new_prob_v = F.softmax(new_logits_v, dim=1)
kl_div_v = -((new_prob_v / prob_v).log() * prob_v).
                                          sum(dim=1).mean()
writer.add_scalar("kl", kl_div_v.item(), step_idx)
```