강화학습 알아보기(4) - Actor-Critic, A2C, A3C

07 May 2019 • 5 Comments

[reinforcement-learning]

지난 글에서는 Grid World 의 `ball-find-3` 문제를 풀기 위한 DQN 알고리즘의 퍼포먼스를 개선하기 위한 여러 방법들과 Deep SARSA 알고리즘에 대해서 살펴보았습니다. 오늘은 `ball-find-3` 에서 앞선 알고리즘들을 크게 뛰어넘는 성능을 보이는 Actor-Critic 알고리즘에 대해서 알아보고, 에이전트의 액션을 시각화해보겠습니다. 또 이 알고리즘을 더 발전시킨 A3C 에 대해서도 알아보겠습니다.

Actor-Critic 알고리즘, A2C

지난 시간에 살펴봤던 Dueling DQN 은 Q 값을 구하기 전에 네트워크의 결과값을 V 와 A 로 나눈 다음에 다시 합치는 아이디어였습니다. 이와 비슷하지만 다른 Actor-Critic 알고리즘[1]은 Actor 네트워크와 Critic 네트워크라는 두 개의 네트워크를 사용합니다.

Actor는 상태가 주어졌을 때 행동을 결정하고, Critic은 상태의 가치를 평가합니다.[2] Dueling DQN 과 꽤 비슷합니다만, Dueling DQN 은 두 값을 합쳐서 결국 Q 값을 구했고 Actor-Critic 은 마지막에 값을 합치지 않는다는 차이가 있습니다. 구현에 따라서 입력을 받는 네트워크의 전반부(Decoder)는 하나로 합치기도 하지만, 결국 마지막에 합쳐서 Q 값을 구하느냐, 그렇지 않고 분리하느냐에 따라 Dueling DQN 과 Actor-Critic 의 구조는 달라집니다. 물론 이 외에도 DQN 에서는 Replay Buffer 를 쓰고 Actor-Critic 은 쓰지 않는 차이 등이 있습니다.

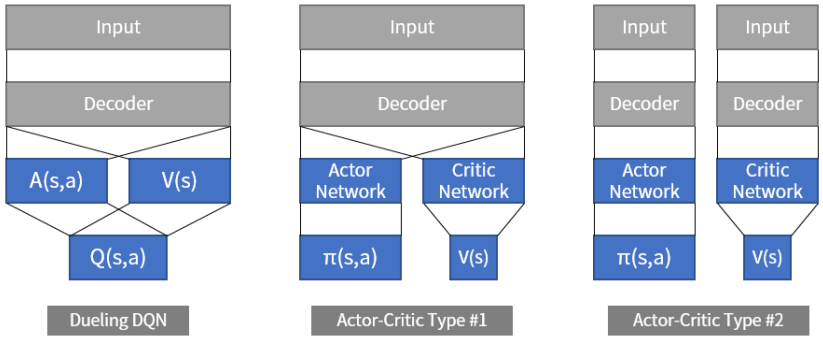


그림 1. 입력을 해석하는 파라미터(Decoder)를 공유(parameter sharing)하느냐 그렇지 않느냐에 따라 actor-critic 알고리즘의 구조는 크게 2가지로 나눌 수 있습니다.

지난 글에서 살펴봤던 DQN 과 Actor-Critic 의 가장 큰 차이점은 Replay Buffer 를 사용하는지 여부입니다.[3] DQN 과 달리 Actor-Critic 은 Replay Buffer 를 사용하지 않고, 매 step 마다 얻어진 상태(s), 행동(a), 보상(r), 다음 상태(s')를 이용해서 모델을 학습시킵니다.

DQN 은 $Q(s,a)$ 값을 얻어내고 Actor-Critic 은 $\pi(s,a)$ 값과 $V(s)$ 값을 구합니다. $V(s)$ 는 지금까지 계속 다뤘던 가치함수이고, $\pi(s,a)$ 는 어떤 상태에서 특정 행동을 취할 확률입니다. 보통 이런 확률은 softmax 를 사용해서 얻어낼 수 있습니다. Softmax 는 일정한 값들을 e 를 밑으로 하는 지수로 계산한 다음 합치고 나눠서 합이 1.0 인 확률로 변환시켜 줍니다.

$$P(z) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad (for \;\; j = 1, \ldots, K)$$

예를 들어 `[2,1,0]` 이라는 값이 있을 경우 softmax 로 변환한다면,

$$sum = \sum_{k=1}^{K} e^{z_k} = e^2 + e^1 + e^0 = 11.1073$$

$$softmax = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} = \left[\frac{e^2}{sum}, \frac{e^1}{sum}, \frac{e^0}{sum} \right] = [0.67, 0.24, 0.09]$$

`[0.67, 0.24, 0.09]` 라는 확률값이 됩니다. Softmax 함수는 이미지 분류(classification) 문제나 텍스트 생성 등 딥러닝의 많은 영역에서 사용됩니다. 강화학습에서도 이렇게 에이전트의 행동 확률을 구하는 데에 쓰일 수 있습니다.

에이전트의 행동 확률을 직접적으로 학습하는 방법을 REINFORCE 또는 policy gradient[4] 라고 부릅니다. policy 는 에이전트가 어떤 행동을 취할지에 대한 정책이라는 뜻이고, gradient 는 미분을 통해 policy 값을 업데이트하며 최적의 policy 를 찾아간다는 의미입니다. 그런데 이렇게 에이전트

의 행동 확률을 직접적으로 학습하는 방법은 불안정하기 때문에 가치함수를 같이 써서 안정성을 높이는 것이 Actor-Critic 의 핵심입니다.

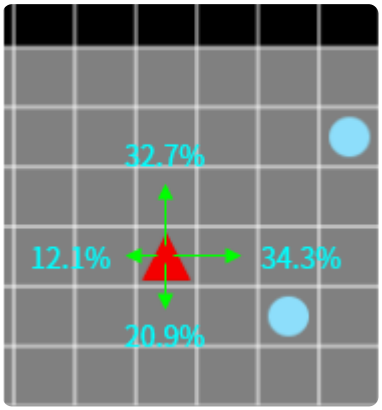


그림 2. policy 값은 어떤 상태(s)에서 각 행동(a)을 할 확률을 직접적으로 나타냅니다.

Actor-Critic 의 Actor 의 기대출력으로 Advantage 를 사용하면 **A**dvantage **A**ctor-**C**ritic, **A2C** 가 됩니다. Advantage 는 예상했던 것($V(s)$)보다 얼마나 더 좋은 값인지를 판단하는 값으로, $Q(s,a)$ 에서 $V(s)$ 를 빼준 값을 많이 사용합니다.

$$A(s,a) = Q(s,a) - V(s)$$

그런데 $Q(s,a)$ 를 구하는 부분은 그림 1의 Actor-Critic 알고리즘 구조에 나와있지 않습니다. 하지만 이 시리즈의 두번째 글에서 다뤘던 내용처럼,

$$\text{실제가치} \simeq \text{현재가치} + \text{미래가치}$$

이고 현재 가치는 보상, 미래 가치는 다음 상태의 가치 함수 $V(s')$ 로 치환한다면 아래와 같은 식을 얻을 수 있습니다.

$$Q(s,a) \simeq R + \gamma V(s')$$

정리하면 Advantage 를 구하는 식은 아래와 같이 바뀝니다.

$$A(s,a) \simeq R + \gamma V(s') - V(s)$$

Critic Network 에서 계산한 $V(s)$ 값이 Actor Network 의 계산에도 영향을 끼치게 됩니다.

Actor-Critic 알고리즘에 영향을 받은 가장 유명한 예로는 딥마인드의 AlphaGo 가 있습니다. 여기에서도 actor 에 해당되는 policy network 와 critic 에 해당되는 value network 를 학습시켜서 주어진 환경(바둑판)에서 최적의 행동(착수)를 찾도록 했습니다.

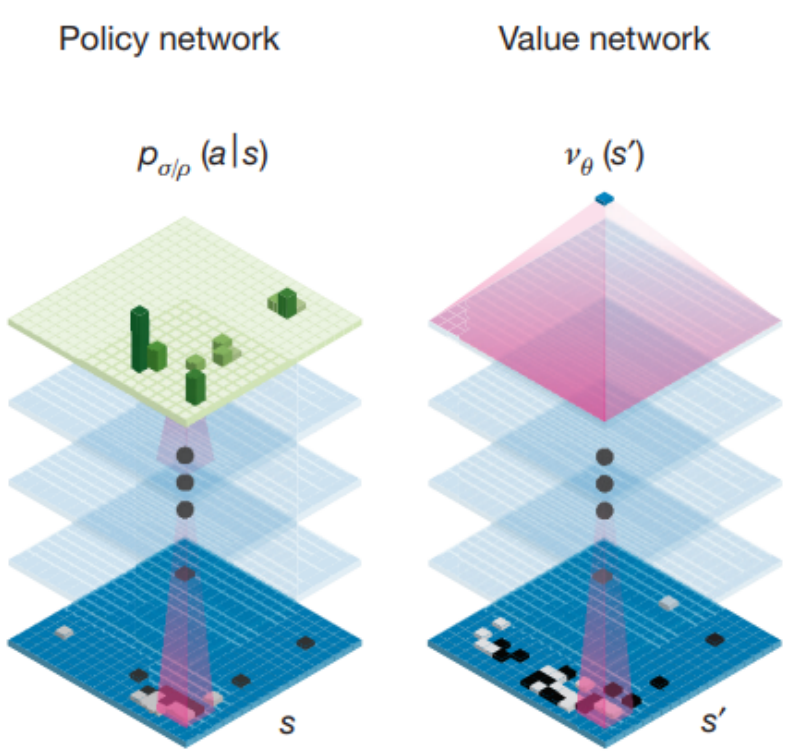
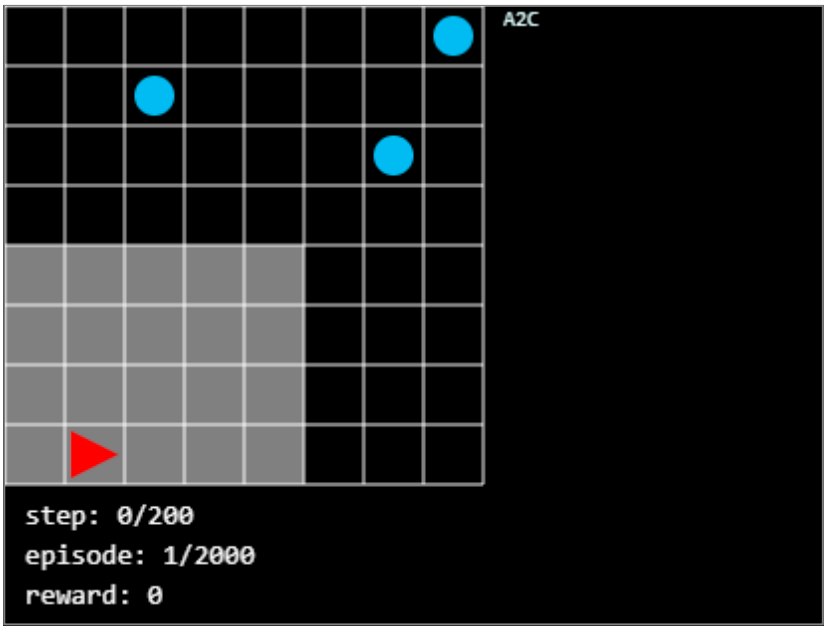


그림 3. 딥마인드의 AlphaGo 는 Policy Network 와 Value Network 라는 두 네트워크로 에이전트를 학습시켰습니다.

당장 알파고를 만들어볼 수는 없지만, 지금까지 살펴본 Grid World 에서 Actor-Critic 알고리즘으로 동작이 크게 개선된 에이전트를 훈련시켜볼 수는 있습니다.



약 100회 이상의 episode 부터 에이전트가 ball을 잘 찾기 시작합니다. 시야에 ball이 없더라도 에이전트는 탐색을 통해 ball에 가까워지고, 시야에 여러 개의 ball이 있을 경우에는 거의 놓치지 않고 모든 ball에 도달합니다.

이런 에이전트는 어떻게 움직이는 것일까요? 유니티(Unity)에서 강화학습 에이전트인 ML-Agent 를 연구하는 아서 줄리아니(Arthur Juliani)의 블로그 글에서 아이디어를 참고하여, A2C 에이전트가 환경을 어떻게 판단하는지를 알아보겠습니다.

A2C의 액션 시각화

`ball-find-3` 에서 A2C 알고리즘으로 38,000 episode 동안 학습시킨 네트워크를 불러와서 에이전트를 실행해보겠습니다. Load Model(Actor-Critic) 버튼을 누르면 웹에 저장되어 있는 미리 학습된 모델과 가중치를 불러올 수 있고, Run(Actor-Critic) 버튼을 눌러서 실행할 수 있습니다.

에이전트 위에 초록색으로 A2C 네트워크의 Actor 가 계산한 각 행동의 확률이 표시되고, 자세한 확률 정보는 최하단에 시안(cyan) 색으로 표시됩니다. Run 버튼은 실행할 때 PAUSE 버튼으로 바뀌기 때문에, 실행 도중에 멈추고 현재 에이전트가 환경을 어떻게 해석하는지 알아볼 수 있습니다.

에이전트가 ball 에 가까울 때와 멀 때, 2개 이상의 ball 이 시야에 한번에 들어올 때, 또는 시야에 아무 것도 없을 때 각 행동 확률이 어떻게 변하는지 확인해보며 에이전트의 학습이 잘 되었는지 확인해볼 수 있습니다.

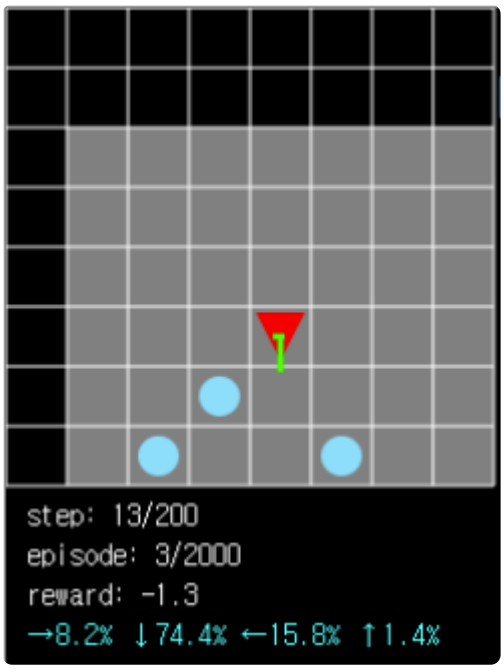


그림 4. 다수의 ball 이 시야에 들어와 있을 때 에이전트는 높은 확률로 최적의 행동을 예측해냅니다(아래쪽, 74.4%).

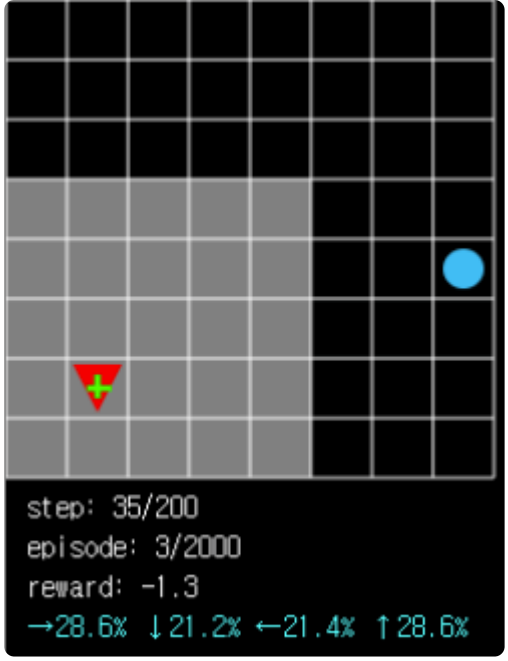


그림 5. 시야에 아무것도 없을 때는 각 방향으로 움직일 확률이 비교적 균등합니다. 하지만 시야에 막힌 부분이 적은 쪽인 오른쪽과 위쪽으로 움직일 확률이 더 높은 것을 알 수 있습니다(오른쪽, 28.6% > 왼쪽, 21.4%).

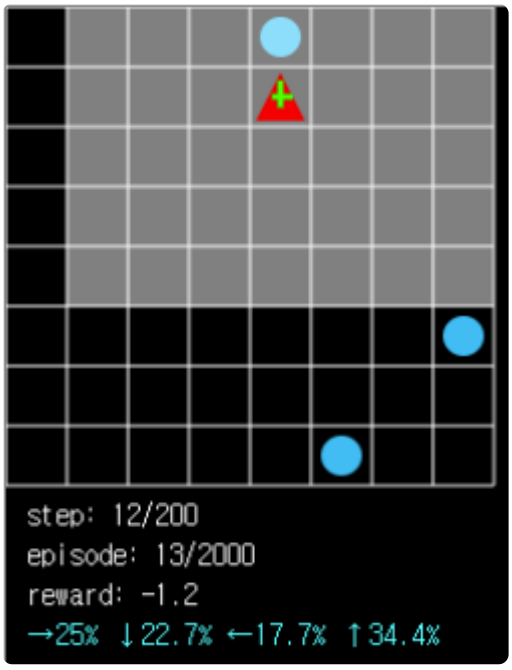


그림 6. ball 이 바로 한 칸 앞에 있는데도 에이전트가 ball 을 향해 움직일 확률이 34.4% 밖에 되지 않습니다. 아직 개선할 여지가 남아있습니다.

그림 6에서 볼 수 있듯이 A2C 알고리즘으로 학습시킨 에이전트에도 아직 부족한 점이 있습니다. 특히 DQN 처럼 replay buffer 를 활용하지 않고 탐색 데이터를 즉시 학습에 이용하기 때문에 잘못된 길로 학습했을 경우 결과값이 안좋게 나올 수 있습니다. 이런 단점을 개선하기 위한 알고리즘인 A3C 가 2016년에 딥마인드에서 발표되었습니다.

A3C

A3C 는 **A**synchronous **A**dvantage **A**ctor-**C**ritic 의 약자입니다. Asynchronous 란 '비동기적'이라는 뜻으로, 한 개가 아닌 **여러 개의 에이전트** 를 실행하며 주기적, 비동기적으로 공유 네트워크를 업데이트한다는 의미입니다. 다른 에이전트의 실행과는 독립적으로 자기가 공유 네트워크를 업데이트하고 싶을 때 업데이트하기 때문에 비동기적이라는 단어가 이름에 붙었습니다.

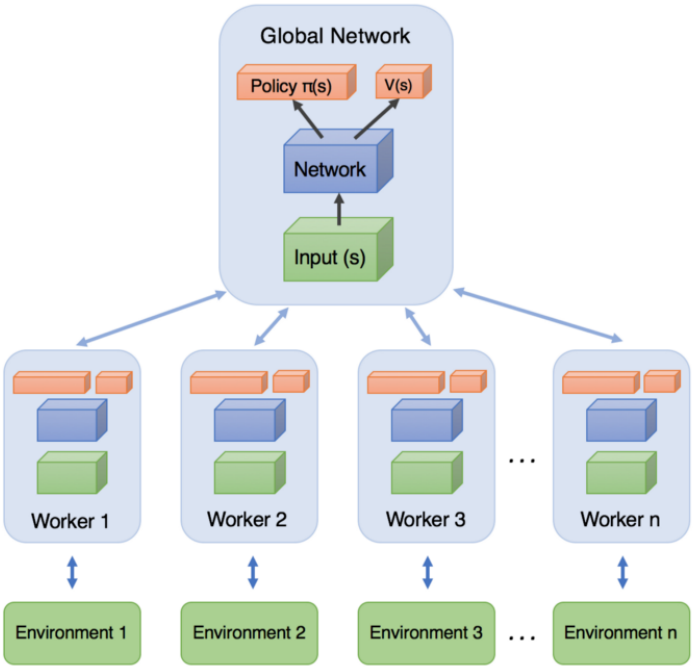
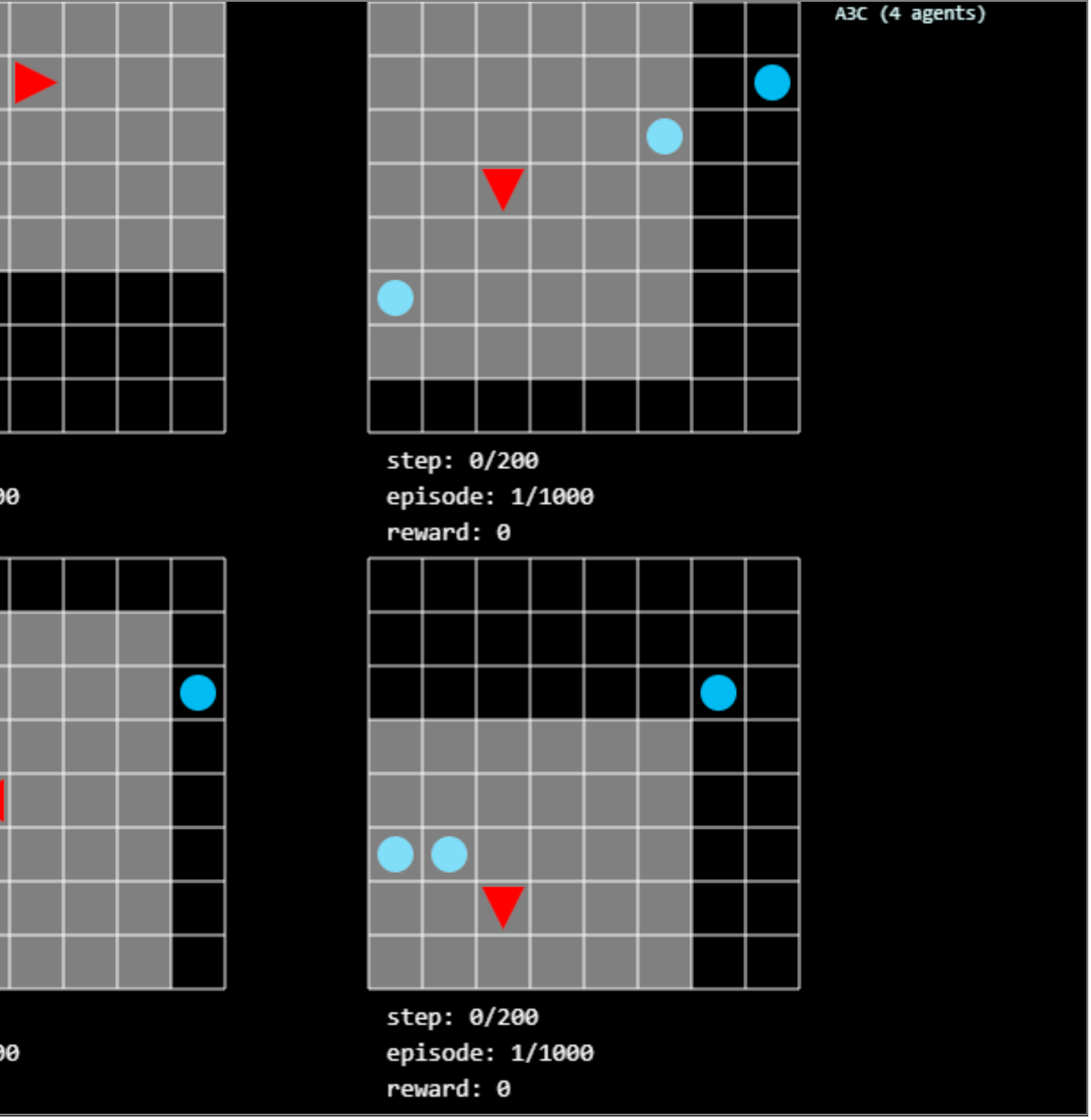


그림 7. 그림으로 나타낸 A3C의 구조. 각 에이전트는 서로 독립된 환경에서 탐색하며 global network 와 학습 결과를 주고 받습니다. 이미지 출처

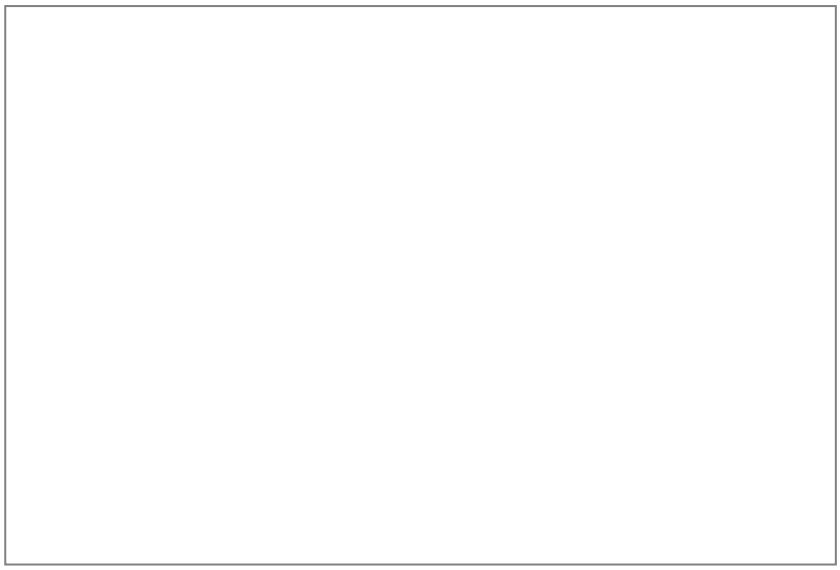
그림 7에서 볼 수 있듯이 A3C 는 A2C 에이전트 여러 개를 독립적으로 실행시키며 global network 와 학습 결과를 주고 받는 구조입니다. 이렇게 여러 개의 에이전트를 실행시켰을 때의 장점은 다양한 환경에서 얻을 수 있는 다양한 데이터로 학습을 시킬 수 있다는 것입니다. 지난 글에서 살펴봤던 DQN 도 replay buffer 를 사용하기 때문에 다양한 경험을 활용할 수 있다는 장점이 있었지만, 오래된 데이터를 사용한다는 단점도 있었습니다. 그에 비해 A3C 는 항상 최신의 데이터를 사용해서 학습하기 때문에 DQN 의 단점을 가지고 있지 않습니다.

`ball-find-3` 문제를 풀기 위한 4개의 에이전트를 사용한 A3C 를 Learn(A3C) 버튼을 눌러서 학습시킬 수 있습니다. 여러 개의 에이전트가 동시에 움직이기 때문에 평소보다 에이전트가 움직이는 속도가 느립니다. 각 에이전트의 진행 episode 총합이 2000 이 되면 학습이 종료됩니다.

이렇게 A3C 네트워크를 학습시킨 뒤 평균 reward 를 그래프로 나타내보면 A3C 는 `ball-find-3` 문제에서 다른 알고리



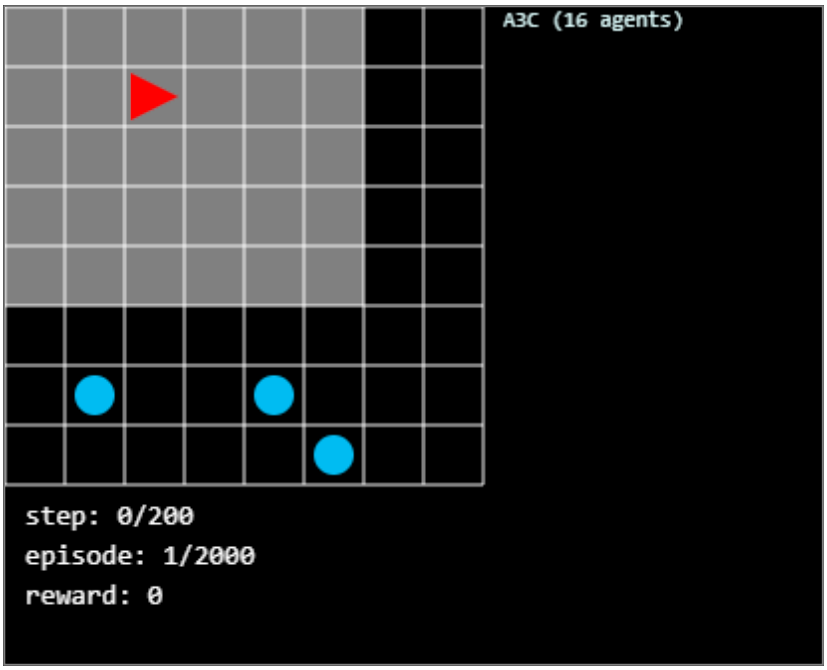
즘에 비해 좋은 퍼포먼스를 냅니다.



초기 학습에서 A3C 의 reward 는 거의 일정하게 증가하고, 학습을 지속할수록 A2C 와 차이가 나는 퍼포먼스를 보입니다. 오리지널 DQN 이나 softupdate 버전의 DQN 과 비교하면 장족의 발전이라고 할 수 있습니다.

A3C의 액션 시각화

그럼 이제 A2C 에 비해 더 똑똑해진 A3C 에이전트의 액션도 시각화해서 A2C에 비해 얼마나 잘 움직이는지 알아보도록 하겠습니다. 최고의 에이전트를 학습시키기 위해 로컬 환경에서 16개의 에이전트를 사용한 A3C 를 50,000 episode 동안 학습시켰습니다.



위의 A2C 예제와 마찬가지로 Load Model(A3C) 버튼을 누르면 웹에 저장되어 있는 미리 학습된 모델과 가중치를 불러올 수 있고, Run(A3C) 버튼을 눌러서 실행할 수 있습니다.

평균 reward 는 `-1.0~-0.0` 정도가 나옵니다. 에이전트가 목표를 향해 좀 더 정확하게 움직이는 것을 확인할 수 있습니다.

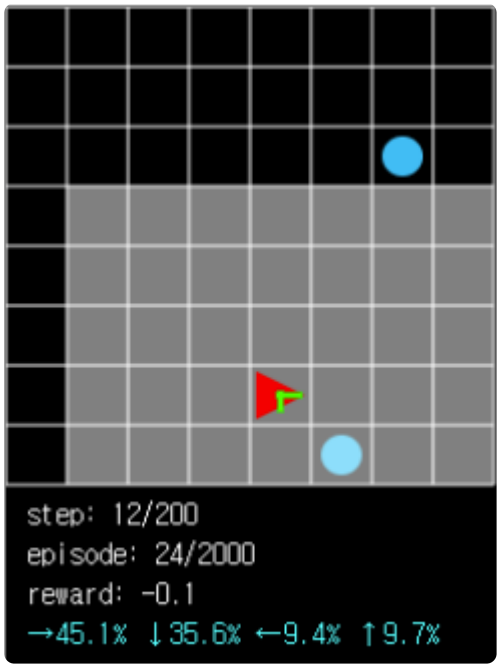


그림 8. ball 이 가까운 위치에 있을 때 ball 과 멀어지는 방향으로 이동할 확률은 한 자리 수로 줄어들었습니다.

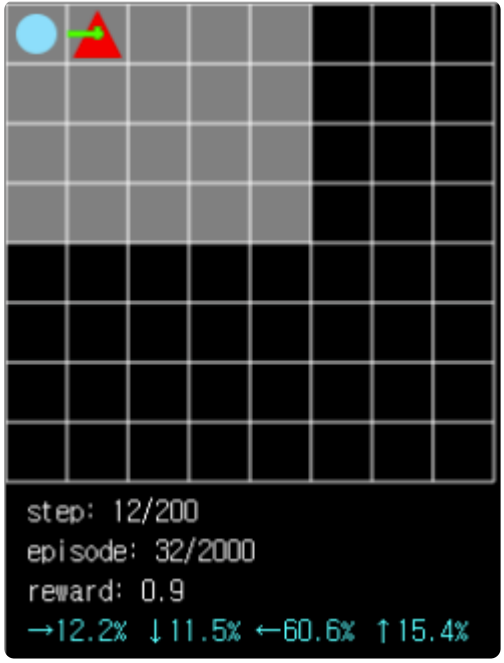


그림 9. 그림 6과 비슷한 상황에서 ball 로 이동할 확률은 60% 로 2배 가까이 증가했습니다.

이것보다 좀 더 퍼포먼스를 개선하고 싶다면 에이전트에 메모리를 추가하면 될 것입니다. 즉 과거에 봤던 상태를 LSTM 등으로 기억해서 현재의 행동을 정하는 데에 활용하는 것입니다. 다만 이런 방식의 예제는 웹에서 돌리기에는 너무 무거운 것 같아서 이번에는 구현하지 않겠습니다.

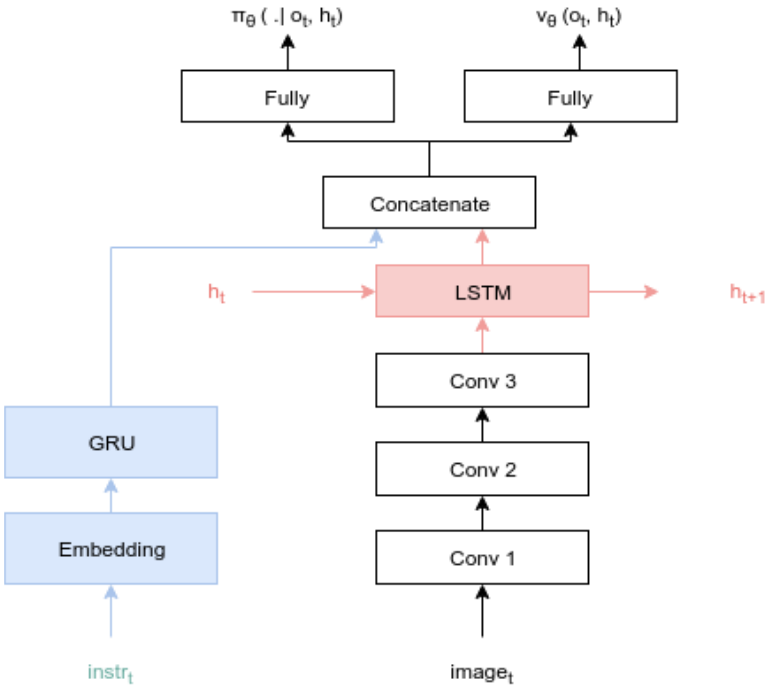


그림 10. 본문의 Grid World 와 비슷한 문제를 푸는 에이전트의 레이어 구조입니다. LSTM 을 쓰면 에이전트에 메모리를 추가할 수 있습니다. 출처 링크

A3C 는 하나의 에이전트 대신 여러 개의 에이전트를 쓴다는 간단한 아이디어지만 퍼포먼스에서는 베이스라인이 될 정도로 좋은 성과를 가져왔습니다. A3C 와 비슷하거나 더 좋은 결과를 내는 최신의 알고리즘으로는 DDPG, TRPO, PPO, TD3, SAC 등이 있습니다. 자세한 내용은 OpenAI 의 강화학

습 학습 & 정리 사이트인 Spinning Up 에 정리되어 있습니다.

지난 글과 달리 이번에는 수식보다는 시각화 위주라 보기에 좀 더 편하셨기를 바랍니다. `ball-find-3` 라는 하나의 문제를 오랫동안 풀고 있으니 다른 문제도 풀고 싶어서 이것저것 시도를 많이 하고 있는데, 다음 시간에는 더 재미있는 문제를 들고 오고 싶습니다. 그럼 이만 이번 글을 마무리하겠습니다. 긴 글을 끝까지 읽어주셔서 감사합니다.

1. 최초의 Actor-Critic 알고리즘은 강화학습의 대가인 A.Barto 와 R.Sutton 이 1983년에 함께 발표한 논문인 Neuron-like elements that can solve difficult learning control problems에서 cart pole 문제를 푸는 데에 사용되었습니다. ↩

2. 딥러닝에서 2개의 네트워크를 사용하는 방법은 이외에 대표적으로 GAN(Generative Adversarial Networks)이 있습니다. 그밖에 2개의 네트워크는 아니지만 2개의 입력을 조합해 새로운 결과물을 만드는 Style Transfer도 있습니다. ↩

3. Actor-Critic 에서 Replay Buffer 를 사용하는 ACER(**A**ctor **C**ritic with **E**xperience **R**eplay)라는 알고리즘도 있습니다. 역시 딥마인드에서 발표했습니다. ↩

4. 원본 논문은 여기에 있습니다. ↩

Related Posts

Shadertoy 'Bleepy blocks' 분석 03 May 2021

『시작하세요! 텐서플로 2.0 프로그래밍』 재고 추적 자동화 01 Mar 2020

『시작하세요! 텐서플로 2.0 프로그래밍』 집필 후기 18 Jan 2020

ALSO ON GREENTEC'S BLOG

2 years ago • 1 comment

Rain drops · greentec's blog

2 years ago • 1 comment

Solving problems with a door and a ...

What do you think?

9 Responses

Upvote Funny Love

Surprised Angry Sad

Comments **Community** **Privacy Policy**

Recommend 3 Tweet Share

Sort by Best

Join the discussion...

김창훈 • 10 months ago

감사합니다.

Reply • Share ›

Minsuk Sung • 10 months ago

감사합니다 Neuron-like elements that can solve difficult learning control problems 논문이 actor-critic을 적용한 최초의 논문이라고 말씀하셨는데, 혹시 NIPS에서 나온 Actor-Critic Algorithms이란 논문은 그럼 최초라고 볼 수 없는거죠?

Reply • Share ›

Hwanhee Kim Mod ➚ Minsuk Sung

• 10 months ago

안녕하세요. NIPS에서 나온 같은 이름의 논문이 여러 개가 검색되어서 어떤 논문을 말씀하시는 것인지는 모르겠지만, 제가 인용한 논문은 1982년에 발표되었고 NIPS는 1987년에 처음