Playing Atari with Deep Reinforcement Learning

1. Introduction

deep learning의 진보로 CNN, MLP, RNN등의 다양한 neural network architecture가 쏟아져 나왔고, 따라서 위와 같은 유사한 technique가 RL에서도 유용할 것으로 생각되는 건 당연하였다.

하지만 RL은 deep learning의 관점에서 몇 개의 극복해야 할 사항들이 보인다.

1) 대다수의 성공적인 deep learning applications는 hand-labelled된 training data를 필요로 한다.

하지만 RL 알고리즘은 가끔은 sparse하고 지연되기도 하는 scalar reward에 대해서 학습이 이뤄진다.

supervised learning에서는 input과 target이 direct하게 연결되어 있는 반면, RL에서 action에 대해 지연된 보상이 이뤄지는 건 학습을 어렵게 한다.

2) RL에서는 **states가 매우 correlated 되어있다.**

반면, 대다수의 deep learning algorithms은 데이터 샘플이 독립이라고 가정한다.

3) RL에서 **data distribution은 계속 변화한다.**

반면, deep learning은 distribution이 고정되어 있다고 가정한다.

본 논문에서는 다양한 방법을 통해 위의 한계점을 극복하고자 노력한다.

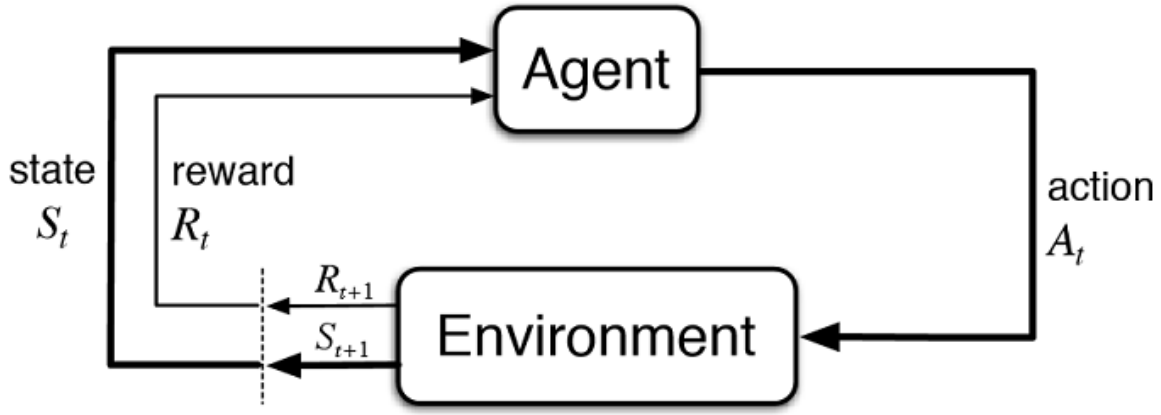
먼저, CNN이 successful control policies를 배울 수 있다고 말한다.

또한 Q-learning을 통해 correlated data와 non-stationary distribution문제를 완화할 수 있으며,

이전의 transitions에서 랜덤하게 sampling하는 replay mechanism을 통해 training distribution을 smooth할 수 있을 것이라 말한다.

2. background

이 논문은 model-free technique 중 하나인 Q-learning algorithm을 neural network를 통해 해결하고 있다.



agent는 environment(epsilon)와 상호작용한다. 여기의 예에서는 Atari emulator가 environment가 된다. 그리고 각 time step마다 agent는 action의 집합 A = {1, 2, ..., K}으로부터 a_t(action)을 결정한다. 이 action은 emulator로 전달이 되고, 그것의 internal state와 game score는 변화된다.

(여기서 게임 점수의 변화라는 reward r_t를 받게 된다.)

기억해야 할 점은 일반적으로 game score는 whole prior sequence of actions and observations에 의존하며, action에 대한 피드백은 많은 time-step이 지난 후에 받을 수도 있다.

또한 agent는 emulator의 internal state는 볼 수 없다. 대신, emulator로부터의 현재의 화면을 나타내는 raw pixel의 vector인 image(x_t)를 볼 수 있다.

이 말은 곧 현재의 situation을 오로지 current screen x_t로부터만 이해할 수 있다는 의미이기도 하다.

따라서 우리는 **sequences of actions and observations s_t** = x1, a1, x2, a2, ..., a_t-1, x_t를 고려하기로 했고, 이 sequence를 기반으로 하여 game 전략을 학습하게 된다. emulator에서의 모든 시퀀스는 유한한 time step 내 종료된다고 가정한다. 이 모델은 엄청나게 크지만, 유한한 MDP가 된다.

agent의 목표는 미래의 보상을 최대화하는 행동을 취하며 emulator와 interact하는 것이다.

우리는 미래의 보상은 gamma라는 factor에 의해 감가된다고 가정하며 t 시점에서의 future discounted return은 $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$ 가 되며, 여기서 T는 game이 종료되는 시점이다.

우리는 optimal action-value function을 어떠한 strategy를 따라 달성할 수 있는 expected return의 maximum을 의미하며 $Q^*(s,a) = \max_\pi \mathbb{E}\left[R_t | s_t = s, a_t = a, \pi\right]$이라고 정의된다.

여기서 pi는 정책으로, sequences를 actions에 mapping한다.

optimal action-value function은 벨만 방정식을 따르는데, 이것은 sequence s'의 다음 step에서 최적 value $Q^*(s', a')$이 가능한 모든 action a'에 대해서 알려져 있다면, action a'를 고르는 최적의 전략은 $r + \gamma Q^*(s', a')$을 maximizing하는 것이다.

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') \middle| s, a \right]$$

(재귀적으로 반복되는 것 같다)

많은 RL 알고리즘의 basic idea는 **action-value function을 벨만 방정식을 사용하여 추정**하는 것이다.

이러한 value iteration 알고리즘은 optimal action-value function으로 수렴하게 된다. (언젠가 ...)

하지만 실제로 이러한 basic approach는 비실용적이다. 왜냐하면 action-value function은 각 sequence 마다 개별적으로 추정되기 때문이다. 대신에 action-value function을 추정하기 위해 **function approximator를 사용**하는 것이 일반적이다. 이것은 일반적으로 linear function approximator이지만, 때때로 neural network와 같은 non-linear function approximator가 사용되기도 한다. 우리는 neural network approximator with weights theta를 **Q-network**라고 부를 것이다. Q-network는 각 iteration i마다 변하는 sequence of loss function을 minimizing하면서 학습된다.

$$L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot)} \left[(y_i - Q(s, a; \theta_i))^2 \right]$$

$$y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a \right]$$

4. deep reinforcement learning

우리의 목표!

connect a reinforcement learning algorithm to a deep neural network which operates directly on RGB images and efficiently process training data by using stochastic gradient updates.

Tesauro's TD-Gammon architecture는 이러한 접근의 시작점을 제공하였다.

이 architecture는 value function을 추정하는 network의 파라미터를 environment의 on-policy samples of experience로부터 직접 업데이트 한다.

이러한 접근은 20년 전에도 사람을 능가하였으므로, 20년동안 hardware가 향상된 지금 유의미한 진보를 보여줄 것으로 예상하였다.

TD-Gammon과는 달리, 우리는 agent의 experiences를 각 time-step마다 저장하는 experience replay 기술을 활용하였다.

알고리즘의 inner loop에서 stored samples에서 random으로 experience를 뽑아서 learning updates, minibatch update를 적용하였다.

이러한 experience replay를 적용한 후, agent는 epsilon-greedy policy에 따라 action을 고르고, 실행했다. arbitrary 길이의 input을 neural network의 input으로 사용하는 것은 어려우나, 우리의 Q-function은 특정 function에 의해 고정된 길이의 representation위에서 동작할 수 있게 되었다. deep Q-learning이라 불리는 전체 알고리즘은 Algorithm 1에 나와있다.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

이러한 접근은 standard online Q-learning 대비 몇 가지 이점을 갖는다.

1. 각 step의 experience는 많은 weight의 업데이트에 사용된다.

2. 연속된 sample로부터 학습을 진행하는 것은 비효율적이다. 왜냐하면 두 샘플 사이에는 강한 상관관계가 존재하기 때문이다. 샘플을 randomizing하는 것은 이러한 상관관계를 부수고, 따라서 update의 분산을 줄여준다.

3. learning on-policy the current parameters는 parameter가 학습될 다음 sample data를 고른다.

예를 들어, maximizing된 action이 왼쪽으로 이동하는 것이라면, left-hand side에서의 sample로 training samples가 dominated될 것이다. 이처럼 원하지 않았던 feedback loop가 발생할 수 있고, parameter가 local minimum에 빠질 수 있게 되는 것이다.

하지만 experience relay를 사용하여 behavior distribution은 averaged되고, 파라미터의 oscillations or divergence를 막는다.

여기서 주의할 점은 experience replay로 학습할 경우 off-policy를 학습하는 것은 필수적이라는 것이다.

4.1 Preprocessing and model architecture

210*160 pixel images with 128 color palette인 raw Atari frames는 computationally demanding하므로, 우리는 input 차원을 줄이기 위한 전처리를 적용하였다.

raw frames의 RGB representation을 gray-scale로 바꾸었으며, 110*84 image로 down sampling하였다.

최종적인 input representation은 image의 paying area가 담겨있는 84*84 region을 cropping하여 얻었다.

마지막 84*84로 cropping한 것은 2D convolutions form의 GPU를 사용하기 위해서이다. (이것은 square를 input으로 받기 때문에)

neural network를 사용하여 Q를 parameterizing하는 몇 가지 방법이 있다.

Q는 history-action pairs를 Q-value의 scalar estimate로 mapping하기 때문에 history와 action은 neural network의 input으로 사용된다.

이러한 architecture의 단점

- separate forward pass is required to compute the Q-value of each action, resulting in a cost that scales linearly with the number of actions.

이러한 architecture의 장점

- ability to compute Q-values for all possible actions in a given state with only a single forward pass through the network.

구체적으로 어떻게 input이 들어가는 지 알아보면,

neural network의 input은 84*84*4 이미지이다.

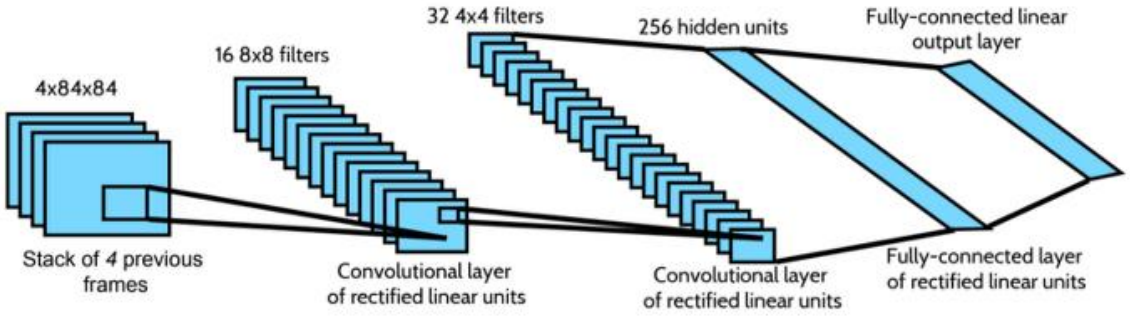
8*8 filters * 16 + stride = 4로 conv + rectifier nonlinearity 적용

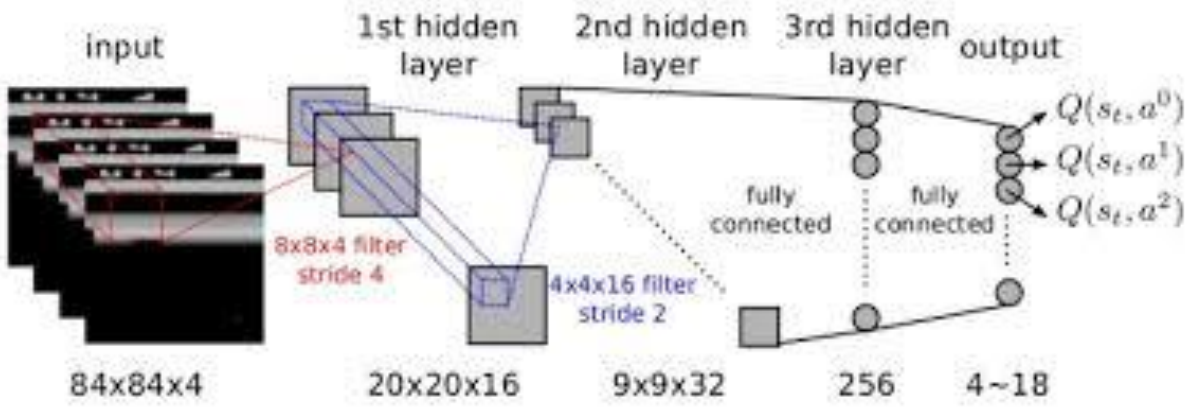
4*4 filters * 32 + stride = 2로 conv + rectifier nonlinearity 적용

fully connected (256 rectifier units)

output는 single output for each action갖는 linear layer

(valid action은 게임에 따라 상이)


4x84x84
Stack of 4 previous frames
16 8x8 filters
Convolutional layer of rectified linear units
32 4x4 filters
Convolutional layer of rectified linear units
256 hidden units
Fully-connected linear output layer
Fully-connected layer of rectified linear units


input
1st hidden layer
2nd hidden layer
3rd hidden layer
output
8x8x4 filter stride 4
4x4x16 filter stride 2
fully connected
fully connected
$Q(s_t, a^0)$
$Q(s_t, a^1)$
$Q(s_t, a^2)$
84x84x4
20x20x16
9x9x32
256
4~18