Playing Atari with Deep Reinforcement Learning

2020년 3월 23일 월요일 오전 1:22

1. Introduction

- DL을 RL에 적용했을 때 발생하는 문제
  1) (DL) required large amounts of hand-labelled training data -> direct association btw inputs & targets
     (RL) sparse, noisy and delayed 한 scalar reward signal로 학습 -> **delay**, thounsands of timesteps long
  2) (DL) independent data samples
     (RL) **highly correlated states**
  3) (DL) fixed underlying distribution
     (RL) **data distribution changes** as algorithm learns new behaviors
- CNN이 복잡한 RL 환경에서 성공적으로 control policy를 학습할 수 있음을 증명
- 변형된 Q-Learning을 통해 학습하는 CNN, stochastic gradient descent와 Experience replay memory 사용
- 하나의 NN을 사용하고 게임에 대한 정보 제공 x
- 오로지 video input, reward, terminal signals, set of possible actions로만 학습 진행
- 동일한 모델 아키텍쳐와 하이퍼파라미터로 7개 게임 학습 진행

2. Background

- agent가 각 time-step마다 action을 선택 -> environment(Atari emulator)에 전달되고 내부 state와 game score 바뀜-> agent는 내부 state를 알 수 없고 이미지벡터와 reward(change in game score)만 받음 -> 전체적 상황을 이해하기 어려움
- 따라서, sequences of actions and observations를 고려하며 train 진행

  sequence $s_t = x_1, a_1, x_2, a_2, \ldots, a_{t-1}, x_t$
- **Markov Decision Process(MDP)**에 RL을 적용 (by using s_t as state representation at time t)
- goal of the agent : select actions that **maximises future rewards** and emulator에 전달
- rewards are discounted per time-step -> **discount factor** γ정의
  - 이때의 R_t $R_t = \sum_{t'=t}^{T} r^{t'-t} r_{t'}$ T(game 종료 time)
- **optimal action-value function** : 특정 state에서 취한 action에 따라 얻을 수 있는 expected return의 최댓값 반환

  $Q^*(s,a) = max_\pi E[R_t | s_t = s, a_t = \alpha, \pi]$
  - 파이π 는 정책함수(policy function)
  - 최적의 Q function은 **Bellman equation**을 따름

    $$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma max_{a'} Q^*(s',a') \mid s,a \right]$$
  - iterative update를 통해 action-value function을 추정

    $$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma Q_i(s',a') \mid s,a \right]$$
  - Qi→Q* as i→∞ (value iteration converges to optimal
  - 그런데, 비현실적임(각 sequence마다 독립적으로 측정되기 때문) -> neural network **function approximator** as Q-network

    $Q(s,a;\theta) \simeq Q^*(s,a)$. 세타는 weight
  - each iteration마다 바뀌는 손실함수를 최소화하면서 학습 진행(세타가 수렴할 때까지 반복)

    $$L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot)}\left[\left(y_i - Q(s,a;\theta_i) \right)^2 \right], \text{where, } y_i = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) \mid s,a \right]$$

    - yi는 타겟 value

- θi−1은 Li(θi)를 optimize할때 정해짐(학습 전 fixed된 supervised learning과의 차이점)
- Gradient

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

- **Q Learning** 알고리즘
  - **model-free** : directly using samples from emulator
  - **off-policy** : ε greedy strategy(ε의 확률로 랜덤, 1-ε의 확률로 a=maxQ(s,a,;θ)의 action 선택)

3. Related Work

- **TD-gammon**
  - RL + Model-free + MLP(1 hidden layer)
  - chess, Go 등에서의 적용 실패
- model free RL(ex. Q learning) + non-linear func approximator/ offf-policy learning => Q network 발산
  - 수렴을 위해 linear func approximator에 focus on
- (최근) Revival of DL + RL
  - DNN : estimate environment
  - restricted Boltzmann : estimate value function or policy
  - gradient temporal-difference methods : Q learning의 발산문제 다룸(아래 2가지 상황에서 수렴함을 증명)
    1. evaluate fixed policy with non-linear approxi
    2. learn control policy with linear approxi (restricted variant of Q-learning)
  - 그러나 아직 non-linear control까지 확장되지 않음
- **NFQ(neural fitted Q-learning)** : 이 논문과 가장 유사한 작업
  1. RPROP 알고리즘으로 Equation2의 loss function을 최적화하여 Q-network 파라미터 업뎃, batch gradient descent

     [이 논문] stochastic gradient descent : iteration에 필요한 연산 줄임, large dataset
  2. deep autoencoder 사용하여 low dimensional representation 학습, real world control task에 성공적으로 적용

     [이 논문] visual input으로 부터 직접적으로 RL 적용 -> action value를 판별하는 특징들 학습

4. Deep Reinforcement Learning

- goal : RL을 DNN과 연결하여 RGB 이미지들에 직접적으로 작동 + stochastic gradient updates로 효율적 학습
- 기존 Q-learning의 문제점 : diverge(발산) b/c of correlations btw samples, non-stationary targets (from 모두를위한딥러닝)
- | 해결책 | 1. Go deep | 2. **experience replay** | 3. seperate target network(이 논문엔 안나온듯) |
  - [Experience replay] 매 time-step마다 상태와 액션, 리워드를 T라는 버퍼에 저장해두고 랜덤으로 샘플링해서 학습
- Deep Q-learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

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

1) buff D를 capacity N으로 초기화
2) action-value function Q를 random weight로 초기화
3) episode를 1~M까지 반복
4) sequence s1을 x1(t=1일때의 이미지)로 초기화, 전처리과정 후 ϕ_1
5) t를 1~T까지 반복
6) e-greedy 알고리즘으로 랜덤 혹은 최대의 보상을 받는 action 선택
7) action 후 reward 와 x_{t+1} observation
8) 현재 state, 현재의 action, 새로운 image x_{t+1} 을 s_{t+1}로 저장, 전처리 과정 후 $\phi(s_{t+1})$
9) 버퍼 D에 ϕ_t, a_t, r_t, ϕ_{t+1} 저장
10) D에서 minibatch만큼 random sampling
11) Set yj
    - ϕ_{j+1}이 목표 지점에 도달하면 r_j
    - 목표지점이 아니라면 $r_j+\gamma \max_{a'} Q(\phi_{j+1}, a'; \theta)$
12) Equation 3을 따라 Loss Function에서 gradient desenct를 수행

★ ○ 핵심은 매 time-step 얻은 결과를 저장해서 랜덤샘플링하여 replay하며 학습한다는 것

4-1) Preprocessing and Model Architecture

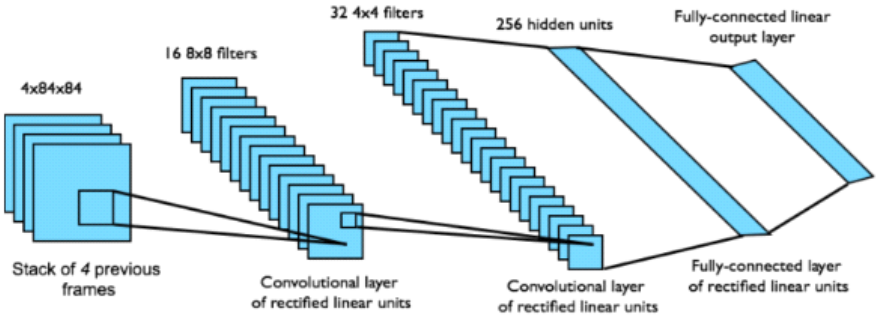
[preprocessing]

- reduce input dimensionality : RGB를 gray-scale로 변환, 210x160픽셀 -> 110x84로 다운샘플링
- final input size는 84x84 (playing area만 캡쳐해서 자름, 정사각형이어야 GPU 연산 가능하기 때문에 필수)
- 전처리 과정 ϕ을 마지막 4개 프레임을 기준으로 처리해서 stack에 넣어둠(4개의 프레임이 한 화면 구성)

[Q-value 구하는 2가지 방법]

1. input = history & action, output = history & 해당 action의 predicted Q-value
   - □ input에 대해 seperate forward pass 진행 -> (단점) action 증가에 따라 연산량 증가
2. input = history, output = 각 action에 대한 predicted Q-value
   - □ NN의 입력이 state라 모든 action에 대한 Q-value를 single forward pass로 구하기 때문에 효율적

[DQN - Deep Q-Networks]



- input : 전처리된 84x84x4 이미지
- 1st hidden layer : 16 channels with 8x8 filters with stride 4 합성곱 연산 -> rectifier nonlinearity 적용
- 2nd hidden layer : 32 channels with 4x4 filters with stride 2 합성곱 연산 -> rectifier nonlinearity 적용

- final hidden layer : fully connected, 256 rectifier units
- output layer : fully connected linear layer with single output

5. Experiments

- 7개의 Atari games에 대해 같은 네트워크구조, 알고리즘, 하이퍼파라미터를 적용하여 진행
- reward structure: 게임마다 점수 scale 차이가 커서 positive reward=1, negative=-1,unchanged=0으로 고정

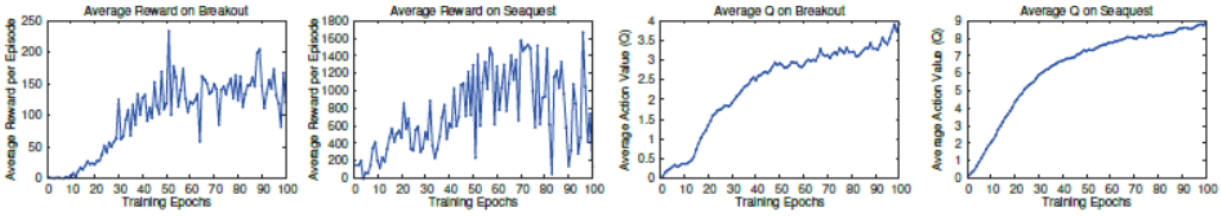
  -> error derivative(오류 도함수)의 sacle 제한하고 게임마다 동일한 learning rate 적용 가능

  -> (단점) reward의 강도에 차이가 없어 성능 문제 발생 가능
- RMSProp Algorithm (최적화) with mini batch size 32
- ∈-greedy Algorithm (behavior policy) : (1~100만번째 프레임) 1 ~> 0.1 동일한 비율로 감소하는 epsilone, (이후) 0.1로 고정
- simple frame skipping technique
  - 모든 프레임이 아닌 k번째 프레임을 보고 action 선택, 마지막 action은 skipped된 frames에 반복 적용
  - 실행시간은 같지만 k배 더 많이 게임 진행 가능
  - k=3으로 처리

5-1) Training and Stability

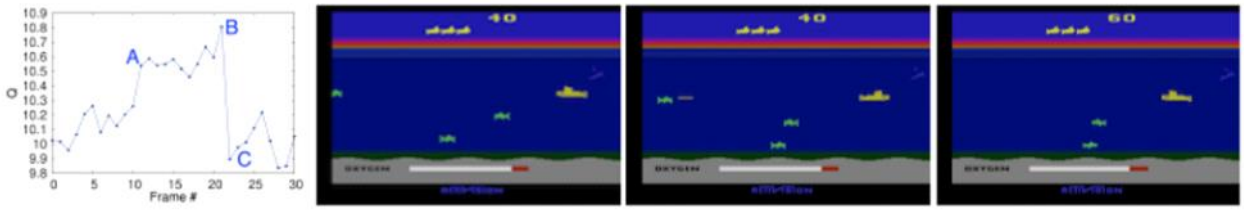
- 지도학습과 달리 RL은 학습 중 agent의 progress를 정확히 측정하는 것이 어려움
- evaluation metric = agent가 episode or game averaged over a number of games에서 얻은 total reward

  -> 학습과정에서 주기적으로 계산해줘야함
- average total reward metric은 noisy한 경향이 있음
  - (policy에 weight를 조금만 변화시켜도 state의 distribution에 큰 변화를 줄 수 있기 때문)



- (왼쪽 두 그래프) Change of average reward during Breakout, Sequest game training -> very noisy
- (오른쪽 두 그래프) average maximum predicted action-value(Q)
  - 왼쪽보다 훨씬 smooth하게 증가함
- Q값이 반드시 수렴한다는 이론적 검증은 없지만 RL과 SGD로 진행한 NN학습이 stable했음

5-2) Visualizing the Value Function



- value function on the game Seaquest

- (point A) 스크린에 적이 등장했을 때 predicted value가 확 증가함
- (point B) 미사일이 적을 맞추려 할 때 정점을 찍음
- (point C) 적이 사라지면 다시 값은 떨어짐

5-3) Main Evaluation

	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	−20.4	157	110	179
Sarsa [3]	996	5.2	129	−19	614	665	271
Contingency [4]	1743	6	159	−17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	−3	18900	28010	3690
HNeat Best [8]	3616	52	106	19	1800	920	**1720**
HNeat Pixel [8]	1332	4	91	−16	1325	800	1145
DQN Best	**5184**	**225**	**661**	**21**	**4500**	**1740**	1075

- 일정한 step까지 0.05의 e값을 갖는 ∈-greedy Algorithm 적용하여 여러 학습방법들에 대해 구한 average total reward
- 가장 좋은결과: HNeat(항상 같은 점수를 얻는 deterministic policy), DQN(e=0.05인 ∈-greedy Algorithm)
- Sarsa : input data를 만들기 위해 수작업으로 설계된 여러 특징들로 linear policies 학습

  -> 가장 성능좋은 feature set 점수
- HNeatBest: obejct의 타입과 location을 output으로 하는 hand-engineered object detector algorithm 사용
- HNeat Pixel : 각 채널에서 object label map을 나타내는 Atari 게임의 special 8 color channel을 사용하여 얻은 결과로 state들의 deterministic sequence 발견을 중요시함
- DQN : 3가지 게임(Breakout, Enduro, Pong)을 제외하고는 사람을 뛰어넘지 못했지만 이전의 방법들보다 높은 performance

6. Conclusion

- RL+ DL의 새로운 방향 제시
- raw pixel들만을 input으로 사용해서 약 2600개가 넘는 control policy 학습
- Deep Q-learning 제시(Stochastic gradient descent + experience memory)
- 모델 구조나 하이퍼파라미터 변화 없이 7개 중 6개의 게임에서 state-of-the-art result!