Playing Atari with Deep Reinforcement Learning

Abstract

1. First Deep Learning Model
2. Model
   - Q-Leanring을 변형한 Convolutional neural network
   - input : raw pixel
   - output : 미래의 reward를 계산한 value fuction 값
3. 7개의 Atari 2600 게임에 적용

   (구조와 알고리즘의 조정 없음)

Final Goal

create single neural network agent that is able to successfully learn to play as many of games

Apply Deep Learning to Reinforcement Learning

1. 필요성
   - 그 당시 RL의 agent는 high dimensional data를 다루지 못함
   - DL에서 high dimensional data를 다루는 RNN, CNN, RBM 등장

2. Challenges
    1. Difference of Data Type
        - DL : require many hand-labelled training data
        - RL : learn from reward

          → reward : sparse, noisy, delayed

    ⭐Solution : Freeze target Q - network

    2. Dependence between Data
        - DL : to be independence
        - RL : highly correlated between states

    ⭐Solution : Experience Relay

    3. Volatility of Data
        - DL : assume data has fixed underlying distribution
        - RL : data distribution changes by learning new behavior

    ⭐Solution : Clipping reward or normalize network adaptively to sensible range

3. Solutions
    1. Freeze target Q-network

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

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

        - Optimization 과정 : Loss fuction $L_i(\theta_i)$의 이전 parameter θ_{i-1}는 고정
        - RL은 θ에 민감하게 영향을 받아 θ값을 고정해 stable learning을 가능하게 함

    2. Experience Relay
        1. 필요성
            - 연속된 sample의 학습은 비효율적
            - 특정 action이 좋다고 판단하면 계속 반복해 poor local minum에 빠짐
        2. 개념
            - randomly samples previous transition

              → smooth the training distribution over many past behaviors

3. 구현 방법
    - experience replay

      →time마다 $e_t = (s_t, a_t, s_{t+1})$를 date set $D = e_1, e_2, ..., e_n$에 저장해 sampling
    - using Mini-batch

      → performing update 과정 : store last N experience in replay memory, samples uniformly at random

3. Clipping reward or normalize network adaptively to sensible range
    1. 필요성
        - sale of scores가 game에 따라 굉장히 variety함
    2. 방법
        - 1 → all positive reward
        - -1 → all negative reward
        - 0 → leaving reward unchanged
    3. 효과
        - limit the scale of error derivatives
        - make easier to use same learning rate at multiple game
    4. 단점
        - unable to differentiate between rewards of different magnitude

💡 DQL Algorithm

1. Algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N (1)
Initialize action-value function Q with random weights (2)
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$ (3)
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$ (4)
    Execute action a_t in emulator and observe reward r_t and image x_{t+1} (5)
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$ (6)
    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 (7)
  end for
end for

1. 가장 최근의 N개만 저장. 넘치면 오래된 것부터 삭제
2. Q 값을 위한 weight를 random하게 initialize함
3. change gray-scale & crop image
4. ε - Greedy 진행
5. a_t라는 action을 하고 r_t의 reward를 얻고 x_{i+1}의 다음 이미지로 진행
6. a_t , r_t , x_{i+1} : Frame Skipping 위해 필요함

   ⭐Frame Skipping

   모든 연속적인 frame을 보지 않고 현재 frame에서 다음 frame까지의 일부만 선택하고 현재의 action과 image가 진행되었다고 여김
7. y_i의 sample을 가지고 그 action에 대해서 sample에 가까워지도록 gradient step을 진행

2. 장점
   1. data effieciency :
      - experience가 weight update 되는 과정에서 re-use됨
      - mini-batch
   2. 다음 train에서 data sample 부분적으로 결정 가능
      - Freeze target Q-network
3. 한계점
   - memory capacity : 모든 history를 메모리 저장 불가능
   - uniform sampling : 모든 past experience 동일한 weight 가짐

Deep Q Learning

- state : $S_t = x_1, a_1, x_2, ..., a_{t-1}, x_t$

  (x :observation , a : action, s : state (sequence))

- 효율적 :

  optimal action value fuction $Q^*(s, a)$ 이용

$$Q^*(s, a) = max_\pi \mathbb{E}[R_t | s_t = s, a_t = a, \pi]$$

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

  neural network 이용

$$Q(s, a; \theta) \approx Q * (s, a)$$

  Loss fuction $L_i(\theta_i)$ 최소화

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

  Gradient

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

  sequence와 action의 behavior distribution인 $P(s, a)$ 를 뽑아 학습하는 SGD

- Off-Policy 방법

$$a = max_a Q(s, a; \theta) = \pi(s)$$

  학습시 Greed strategy로 policy를 고정