* Introduction.

과거 : 수동으로 설정하는 feature, linear한 value function과 policy 를 이용한 RL이 주로 쓰였다.
그러나 이는 feature representation의 quality에 성능이 많이 의지하곤 했다.

현재의 RL challenge : High dimensional input인 vision과 speech으로부터 agent가 바로바로 control되는 것.

⇒ 최근 딥러닝의 발전으로 CV or speech recognition 의 큰 발전이 일어났고
raw sensory data로부터 high-level feature 를 추출할 수 있게 되었다.
그래서 이를 강화학습에도 적용해보고자 한다.

⇒ 그러나 딥러닝은 강화학습에 적용하기엔 몇가지 어려움이 있었는데,
i) 기존 DL은 다량의 hand-labelled data로 학습하는 반면,
RL은 sparse, noisy, delayed 한 scalar reward로부터 학습한다.
특히 action과 reward 간의 delay가 큰 문제가 될 수 있다.

ii) DL 알고리즘은 data sample들이 서로 independent하다고 가정하는 반면,
RL은 state 간 연관성이 높다.

iii) RL은 새 행동과 reward를 학습함에 따라 data distribution이 변화한다. (DL과 달리)

⇒ CNN이 모두 다 해결가능. Q-learning 알고리즘으로 학습하고, SGD로 가중치 update한다.
ii), iii) 문제 해결을 위해선 replay 세트 도입.

⇒ Testbed로는 Atari 2600을 사용했다.
Atari 2600은 high-dimensional visual input을 넣고, 인간에게도 어려운 다양한 task가 존재한다.
목표는 single NN agent를 만드는 것.
이 agent는 사람과 동일하게 game에 대한 정보나, visual feature를 전혀 모르는 상황에서 시작한다.
오직 video input, reward, terminal signal, set of possible action만 가진다. hyperparameter는 처음부터 끝까지 일정하다.

* Background

$A = \{1, \dots K$
Each time step, agent는 action set 으로부터 action을 고른다.
- Action은 emulator로 전달되어 internal state와 game score (reward) 를 변경한다.
- $\mathcal{E}$ (environment)는 stochastic하다. Internal state는 agent가 볼수 없다. Agent는 오직 screen의 vector of raw pixel만 볼 수 있다.
- MDP 상황을 가정.
  + action과 observation에 순서가 있다고 가정. $S_t = x_1, a_1, x_2, \dots$
  + 유한 번의 time-step 안에 terminal state에 도달한다고 가정
- 기타 Assumption
  - Discounting rate 도입.
  - Optimal action value $Q^*(s,a)$ 도입.
    → state s'에서 모든 잠재적 action들 중에 optimal한 action a'를 알고 있다면, 항상 $Q^*(s', a')$ 를 고른다.

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

- 기존의 value iteration에 따르면, $Q_{i+1}(s,a) = E\left[r + \gamma \max_{a'} Q_i(s',a') | s,a\right]$ 방식으로 $\overset{\text{if}}{i \to \infty}$면 $Q_i \to Q^*$ 라고 알려져 있다.
그러나 action value function이 각 상황에 따라 바뀌니깐 쓰이기 어렵다.
따라서 $Q(s,a;\theta) \approx Q^*(s,a)$ 인 approximator로 추정하는 편을 이용한다.
function approximator로는 weight가 θ인 Neural Network function을 Q-Network로서 사용한다.
Q-Network은 loss function $L_i(\theta_i)$를 최소화하는 방향으로 학습된다.

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

$y_i = E\left[r + \gamma \max_{a'} Q'(s', a'; \theta_{i-1}) | s, a\right]$
$\rho(s,a)$: behavior distribution

- Supervised learning과 달리, target은 network weight에 따라 달라진다.

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

*Related Work

- 원래 가장 성공적인 RL의 예시는 TD-gammon이었다. (value function approximator로 one-hidden layer를 가진 multi-layer perceptron 사용)
  그러나 이후 chess에서는 별로 성공적이지 못하여 주사위의 randomness가 state exploration과 value function을 smooth하게 한 TD-gammon의 특성 때문에 예외적인 성공이 아니냐는 평을 들었다.
- 더욱이, model-free RL (ex. Q-Learning)을 비선형 approximator or Off-policy와 결합하는 것은 Q-network가 수렴하지 않고 발산한다는 게 밝혀졌다.

- 그러나, 최근에는 DL의 발전에 따라 DL과 RL을 combine 하려는 시도가 이어지고 있다.
  위의 diverge하는 문제도 gradient temporal-difference에 의해 다뤄졌다.
  이 방법은 fixed policy와 nonlinear function approximator의 조합 혹은 control policy와 linear function approximator의 조합에서 수렴하게 했다 ✓
  그러나 이런 기존의 연구들로 nonlinear control로는 확장되지 않았다. (우리가 할 것이다.)

- 우리 연구와 가장 비슷한 기존의 연구는 neural fitted Q-learning (NFQ)이다.
  NFQ는
  - RPROP 알고리즘으로 loss function을 최적화 함.
  - 그러나 여기서 사용하는 batch update는 data set의 크기에 따라 계산 비용이 늘어남.
  - 그래서 batch update 대신 Stochastic gradient update 사용.

  - 또한 NFQ는 autoencoder로 기존 pixel을 low-dimensional로 만든 다음 NFQ에 넣었다.
  - 반면 이 연구는 visual input (raw pixel)을 바로 input으로써 사용하고, 처음부터 끝까지 강화학습만 사용한다.

- 뿐만 아니라

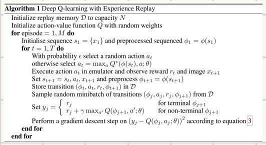
* DQN

- CV와 음성 인식이 발달하면서 raw input을 바로 받아서 처리할 수 있게 되었다.
  DNN을 SGD를 이용한 lightweight update를 이용해서.

⇒ 우리의 목표도 RGB image를 직접 받아서 SGD update를 이용하여 훈련하는, DNN과 RL을 결합한 모델을 만드는 것이다.

- Tesauro의 TD-gammon에 큰 영감을 받았다.
  여기선 value function을 추정하는 parameter들을 on-policy로서 update했다.
  20년 전에도 이 결과가 인간보다 뛰어났다고 하는데, hardware의 진보가 일어난 20년 후인 지금, 현대 DNN과 RL을 결합한다면 상당한 발전을 볼 수 있을 것이다.

- 그러나 기존 연구와는 다르게, 우린 experience replay라는 걸 사용한다.
  $\text{experience}_t = (S_t, a_t, r_t, S_{t+1})$
  $\text{dataset} = e_1, e_2, \cdots e_N$
- 알고리즘 내부 loop에서, D에서 sample을 random하게 꺼내서 Q-learning update or minibatch update를 수행한다.
  이 다음, agent는 ϵ-greedy에 따라 action을 한다.
  또한 input으로 과거 history를 사용하면 길이가 계속 바뀌니까 ϕ function을 이용해 길이를 고정시킨다.
  (계속 늘어나기 때문) (최근 4개의 frame만 쓰는 식)


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 $\epsilon$ 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
```

- 장점 1. 모든 step의 experience가 나중에 잠재적으로 쓰일 수 있다. data efficiency 좋음.

- 장점 2. 연속된 sample로 학습하는 건 sample들 간 연관성 때문에 학습이 어려움.
  그래서 이렇게 experience를 저장해 놓고 random하게 꺼내 쓰는 게 훨씬 나음.

- 장점 3. Off-policy 학습.
  Data distribution이 어느 한 쪽으로 편향되지 않는다.
  On-policy면, 그때 그때 optimal한 data를 집중적으로 학습하기 때문에 local optimum에 빠지거나 diverge할 수 있다.

- 한계점.
  현재 방식에서는 최근 N개의 experience만 저장하고, update 시 random하게 꺼내 쓴다.
  ⇒ 과거의 중요한 경험은 차별화하지 못하고 덮어씀.
  또한 현재의 uniform sampling도 replay memory들에게 똑같은 중요도를 부여하기 때문에 한계가 있다.

* Preprocessing & Model Architecture.

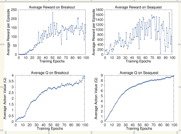
※ Experiments.

- 7개의 서로 다른 Atari game을 대상으로 실험을 진행했다. (사전 정보도 없이)
  그동안 같은 algorithm과 hyperparameter를 사용하여 모델의 강건성을 보여줬다.
  딱 하나, reward 크기만 scaling 했다. 성공 = 1, 실패 = −1, 아니면 0. → error 줄이고, learning rate 안 바꿔도 되고, agent의 performance에도 일관성을 준.

- size = 32의 미니배치를 가진 RMSProp 알고리즘 사용.
  ε는 1에서 시작해 0.1로 선형적 감소 후 0.1 유지.

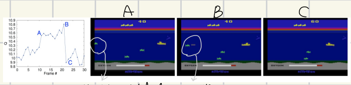
- Frame-Skipping technique.
  매 frame을 보고 action 취하고 하기엔 action을 고르는 게 시간이 많이 걸리기 때문에 비효율적이다.
  따라서 k번째 frame마다 새로운 action을 골라 취해주고, skip된 frame은 기존의 action을 유지.
  Space Invader라는 게임만 k=3 이고 나머진 k=4.

Training & Stability.

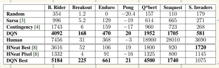


- 지도학습과 달리, 강화학습은 학습 도중에 progress를 평가하기 어렵다.
  기준을 average total reward로 하면, 약간의 weight change에도 policy와 state가 변하기 때문에 아주 noisy하다.
  기준을 action-value function Q로 두면, 특정 state에서 특정 policy를 따랐을 때 reward 얼마나 받을지 예측 가능
  → 훨씬 stable하고 발산도 하지 않았다.
  ⇒ 비록 이론적으로 수렴을 보장할 순 없지만, RL과 SGD를 사용하여 large NN을 안정적으로 학습할 수 있다.

Visualizing the Value Function.



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

- Sarsa와 Contingency는 게임에 대한 사전 지식을 알 수 있음에도 DQN이 성능이 더 낫다.
-