DqN

2016. 9. 12.

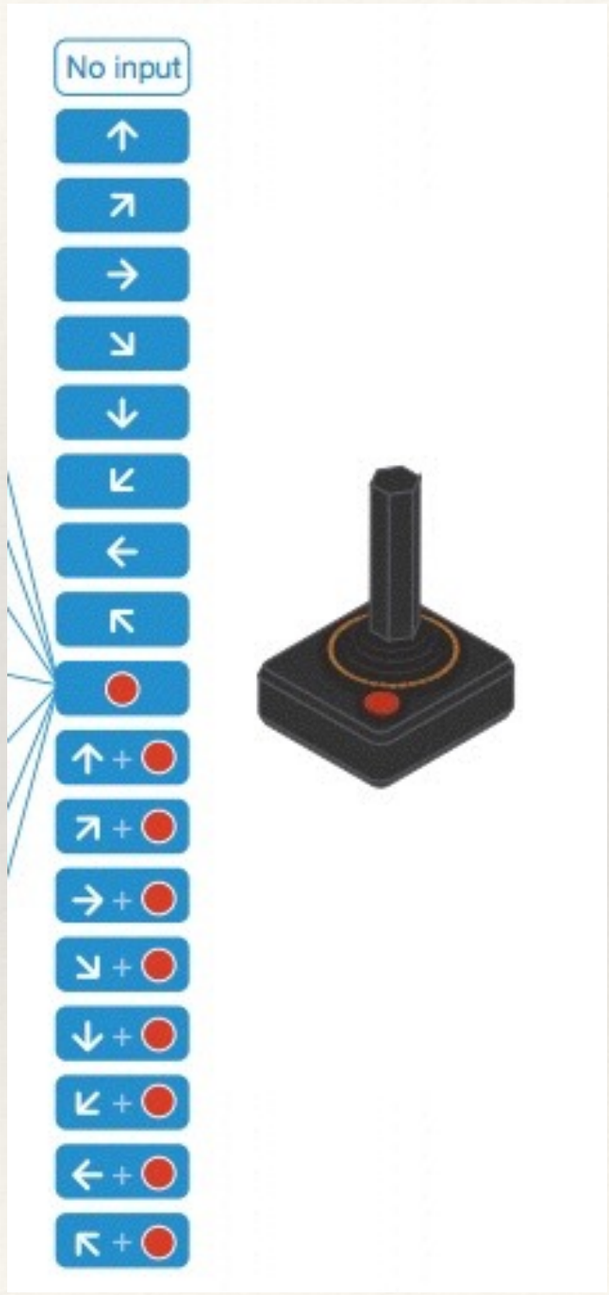
박성준
park@move.is
Move Inc.

Today's Agenda

- Review of DeepMind's Atari DqN formulation
  - Playing Atari with Deep Reinforcement Learning (Mnih et al, 2013 ; http://arxiv.org/pdf/1312.5602v1.pdf)
  - Human-level control through deep reinforcement learning (Mnin et al, 2015 ; http://home.uchicago.edu/~arij/journalclub/papers/2015_Mnih_et_al.pdf)
  - David Silver's own take on DqN from his course on RL (http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching.html)
  - Demystifying Deep Reinforcement Learning by Tambet Matiisen (https://www.nervanasys.com/demystifying-deep-reinforcement-learning/)
  - DQNをKerasとTensorFlowとOpenAI Gymで実装する (https://elix-tech.github.io/ja/2016/06/29/dqn-ja.html ; https://github.com/elix-tech/dqn)
  - devsisters/DQN-tensorflow (https://github.com/devsisters/DQN-tensorflow)
  - Episodic Exploration for Deep Deterministic Policies: An Application to StarCraft Micromanagement Tasks (http://arxiv.org/pdf/1609.02993.pdf)

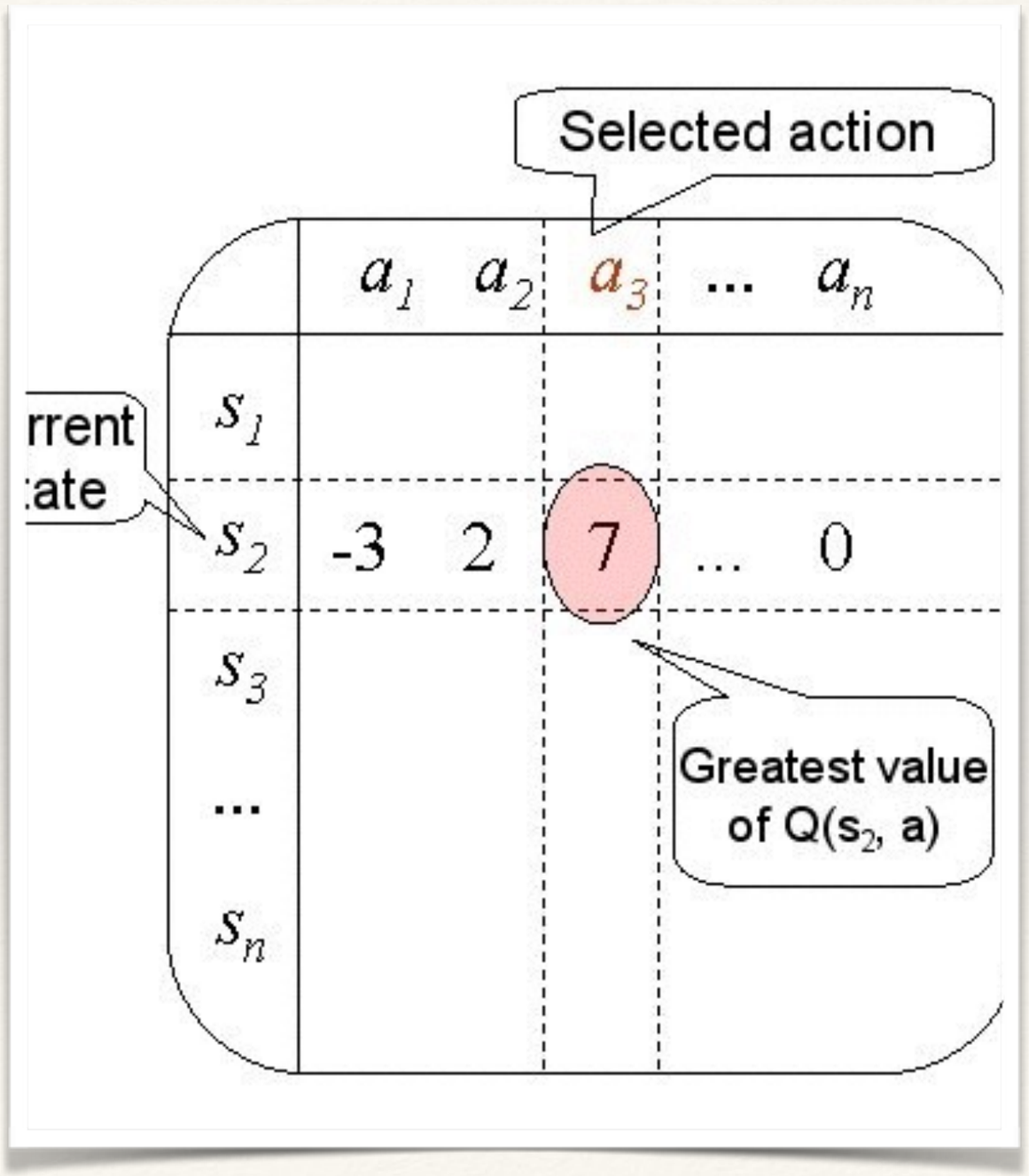
What is the Problem?

- sparse and time-delayed labels
  - credit assignment problem
  - explore-exploit dilemma



Mathematical Formalisation

- r_t : immediate reward
- G_t : return == total accrued rewards from top t on
- $q_\pi(s, a) = E_\pi[G_t : S_t = s, A_t = a]$
- $\pi^* = greedy(q^*)$



Learning Algorithm

Sarsa(λ) Algorithm

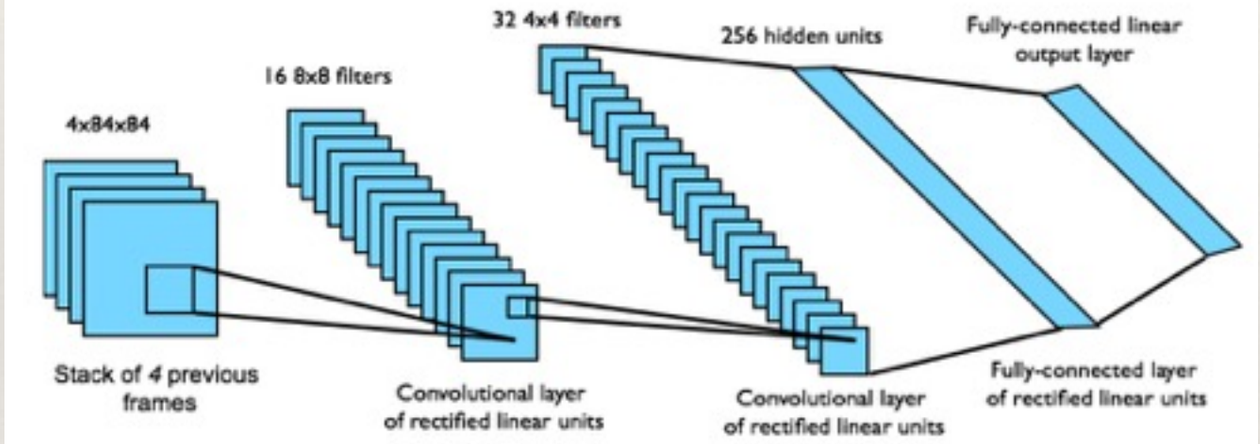
Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Repeat (for each episode):
    $E(s, a) = 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
    Initialize S, A
    Repeat (for each step of episode):
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g., ε-greedy)
        $\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$
        $E(S, A) \leftarrow E(S, A) + 1$
        For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
            $Q(s, a) \leftarrow Q(s, a) + \alpha \delta E(s, a)$
            $E(s, a) \leftarrow \gamma \lambda E(s, a)$
        $S \leftarrow S'$; $A \leftarrow A'$
    until S is terminal

Value Function Approximation

- Number of States : 84×84 pixels with 256 grey levels ==> we would have 256x84x84x4 ≈ 10^{67970} possible game states.
- That is, 10^{67970} rows in our imaginary Q-table – more than the number of atoms in the known universe! One could argue that many pixel combinations (and therefore states) never occur – we could possibly represent it as a sparse table containing only visited states. Even so, most of the states are very rarely visited, etc, etc, etc

DQN in Atari (I)

- End-to-end learning of values $Q(s, a)$ from pixels s
- Input state s is stack of raw pixels from last 4 frames
- Output is $Q(s, a)$ for 18 joystick/button positions
- Reward is change in score for that step



Layer	Input	Filter size	Stride	Num Filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

* No (max) pooling
** 18 outputs to match 18 available actions

DQN in Atari (II)

1. Loss function : $L = \frac{1}{2}[r + \underset{a'}{max}\, Q(s', a') - Q(s, a)]^2$
2. Do a feedforward pass for the current state s to get predicted Q-values for all actions
3. Do a feedforward pass for the next state s' and calculate maximum overall network outputs $maxQ(s', a')$
4. Set Q-value target for action to $r + \gamma\, \underset{a'}{max}(s', a')$(use the max calculated in step3). For all other actions,set the Q-value target to the same as originally returned from step 2, marking the error 0 for those outputs
5. Update the weights using back propagation

David Silver's Take

Experience Replay in Deep Q-Networks (DQN)

DQN uses experience replay and fixed Q-targets

- Take action a_t according to ϵ-greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory $\mathcal{D}$
- Sample random mini-batch of transitions (s, a, r, s') from $\mathcal{D}$
- Compute Q-learning targets w.r.t. old, fixed parameters w^-
- Optimise MSE between Q-network and Q-learning targets

$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i}\left[\left(r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i)\right)^2\right]$$

- Using variant of stochastic gradient descent

Stochastic Gradient Descent with Experience Replay

Given experience consisting of $\langle state, value \rangle$ pairs

$$\mathcal{D} = \{\langle s_1, v_1^\pi \rangle, \langle s_2, v_2^\pi \rangle, ..., \langle s_T, v_T^\pi \rangle\}$$

Repeat:

1. Sample state, value from experience

$$\langle s, v^\pi \rangle \sim \mathcal{D}$$

2. Apply stochastic gradient descent update

$$\Delta \mathbf{w} = \alpha(v^\pi - \hat{v}(s, \mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w})$$

Converges to least squares solution

$$\mathbf{w}^\pi = \underset{\mathbf{w}}{\text{argmin}}\ LS(\mathbf{w})$$

Deep Q-Learning Algorithm

- Initialise replay memory D
- initialise action-value function Q with random weights
- observe initial state s
- **Repeat**
    - select an action a
        - with probability e select a random action
        - otherwise select a = argmaxQ(s, a)
    - carry out action a
    - observe reward r and new state s’
    - sample random transitions <ss, aa, rr, ss’> from replay memory D
    - calculate target for each mini batch transition
        - if ss’ is terminal state, then tt = rr
        - otherwise tt = rr + gamma maxQ(ss’, aa’)
    - train the Q network using $(tt - Q(ss, aa))^2$
- **until** terminated

Model Specifics

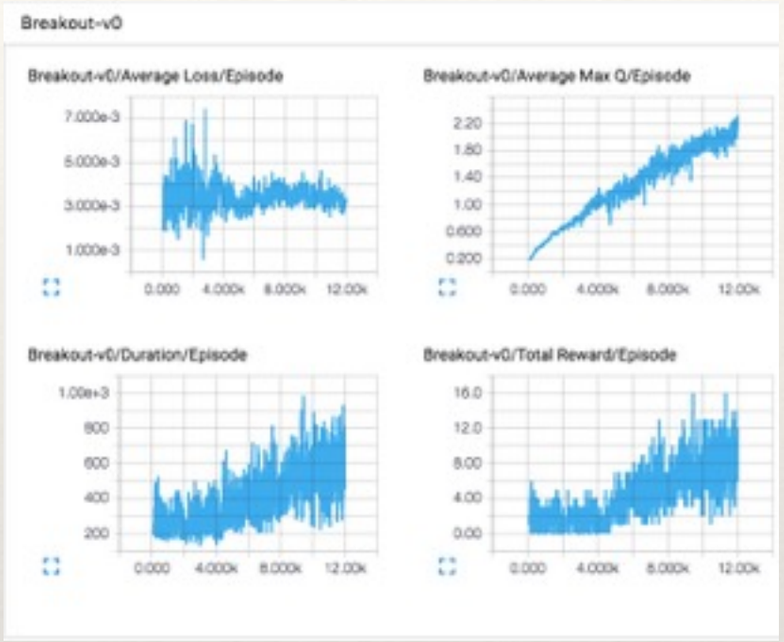
- The raw frames preprocessed by first converting their RGB representation to grey-scale and down-sampling it to a 110×84 image. The final input representation obtained by cropping an 84 × 84 region of the image that roughly captures the playing area. The final cropping stage is only required because we use the GPU implementation of 2D convolutions from [11], which expects square inputs. For the experiments in this paper, the function ϕ from algorithm 1 applies this preprocessing to the last 4 frames of a history and stacks them to produce the input to the Q-function.
- Used an architecture in which there is a separate output unit for each possible action, and only the state representation is an input to the neural network.
- Since the scale of scores varies greatly from game to game, we fixed all positive rewards to be 1 and all negative rewards to be −1, leaving 0 rewards unchanged. Clipping the rewards in this manner limits the scale of the error derivatives and makes it easier to use the same learning rate across multiple games. At the same time, it could affect the performance of our agent since it cannot differentiate between rewards of different magnitude.
- Used the RMSProp algorithm with minibatches of size 32. The behavior policy during training was -greedy with annealed linearly from 1 to 0.1 over the first million frames, and fixed at 0.1 thereafter.
- Agent sees and selects actions on every k_{th} frame (k = 4, usually) instead of every frame, and its last action is repeated on skipped frames.
- Trained for a total of 10 / 50 million frames (that is, around 38 days of game experience in total) and used a replay memory of 1 million most recent frames.
- We also found it helpful to clip the error term from the update $e = y_i - Q$ to be between -1 and 1. (carpediem20의 코드는 이 부분이 구현이 안 되어 있음: 이 개발자 역시 함수를 불 연속적으로 만들었음. 역시 좀 이상함…)
  - $\nabla L = \mathbb{E}[e \nabla Q(s,a;\theta)]$ —> $\nabla L = \mathbb{E}[\pm 1 \nabla Q(s,a;\theta)]$ (outside of [-1, 1])
    - $L = 1/2\mathbb{E}[e^2]$ ($[-1,1]$)
    - $L = \mathbb{E}[|e|]$(outside of $[-1,1]$)
- 1 epoch = 50,000 minibatch weight updates = ~30 minutes of training time. 1 episode <= ~ 5 min

minibatch size	32	Number of training cases over which each stochastic gradient descent (SGD) update is computed.
replay memory size	1000000	SGD updates are sampled from this number of most recent frames.
agent history length	4	The number of most recent frames experienced by the agent that are given as input to the Q network.
target network update frequency	10000	The frequency (measured in the number of parameter updates) with which the target network is updated (this corresponds to the parameter *C* from Algorithm 1).
discount factor	0.99	Discount factor gamma used in the Q-learning update.
action repeat	4	Repeat each action selected by the agent this many times. Using a value of 4 results in the agent seeing only every 4th input frame.
update frequency	4	The number of actions selected by the agent between successive SGD updates. Using a value of 4 results in the agent selecting 4 actions between each pair of successive updates.
learning rate	0.00025	The learning rate used by RMSProp.
gradient momentum	0.95	Gradient momentum used by RMSProp.
squared gradient momentum	0.95	Squared gradient (denominator) momentum used by RMSProp.
min squared gradient	0.01	Constant added to the squared gradient in the denominator of the RMSProp update.
initial exploration	1	Initial value of ε in ε-greedy exploration.
final exploration	0.1	Final value of ε in ε-greedy exploration.
final exploration frame	1000000	The number of frames over which the initial value of ε is linearly annealed to its final value.
replay start size	50000	A uniform random policy is run for this number of frames before learning starts and the resulting experience is used to populate the replay memory.
no-op max	30	Maximum number of "do nothing" actions to be performed by the agent at the start of an episode.

The values of all the hyperparameters were selected by performing an informal search on the games Pong, Breakout, Seaquest, Space Invaders and Beam Rider. We did not perform a systematic grid search owing to the high computational cost, although it is conceivable that even better results could be obtained by systematically tuning the hyperparameter values.

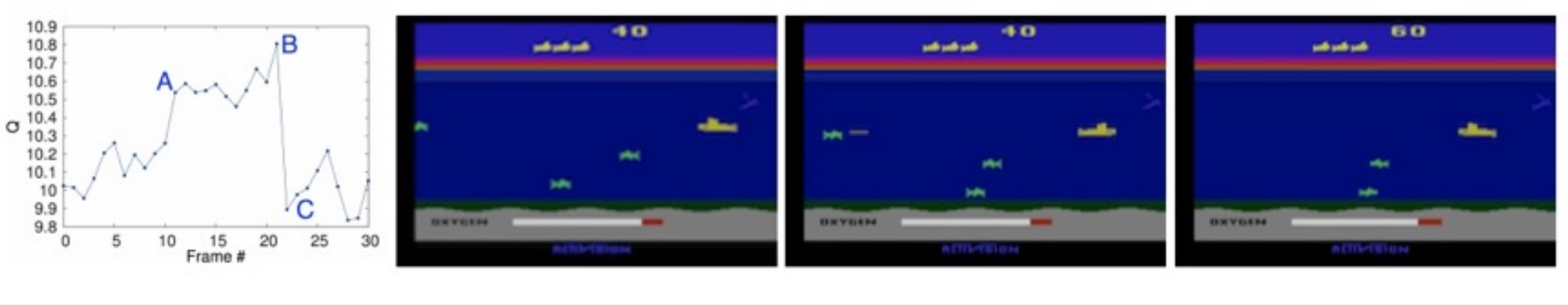
Monitoring During Training

- The average total reward metric tends to be very noisy because small changes to the weights of a policy can lead to large changes in the distribution of states the policy visits
- Collect a fixed set of states by running a random policy before training starts and track the average of the maximum2 predicted Q for these states
- The trained agents were evaluated by playing each game 30 times for up to 5 min each time with different initial random conditions (‘noop’; see Extended Data Table 1) and an e-greedy policy with e = 0.05. This procedure is adopted to minimize the possibility of overfitting during evaluation




```
EPISODE:  13344 / TIMESTEP:  7503639 / DURATION:   666 / EPSILON: 0.10000 / TOTAL_REWARD:   9 / AVG_MAX_Q: 2.6048 / AVG_LOSS: 0.00293 / MODE: exploit
EPISODE:  13345 / TIMESTEP:  7504288 / DURATION:   649 / EPSILON: 0.10000 / TOTAL_REWARD:   9 / AVG_MAX_Q: 2.6511 / AVG_LOSS: 0.00270 / MODE: exploit
EPISODE:  13346 / TIMESTEP:  7504558 / DURATION:   270 / EPSILON: 0.10000 / TOTAL_REWARD:   2 / AVG_MAX_Q: 2.4739 / AVG_LOSS: 0.00203 / MODE: exploit
EPISODE:  13347 / TIMESTEP:  7505251 / DURATION:   693 / EPSILON: 0.10000 / TOTAL_REWARD:   9 / AVG_MAX_Q: 2.6304 / AVG_LOSS: 0.00256 / MODE: exploit
EPISODE:  13348 / TIMESTEP:  7505850 / DURATION:   599 / EPSILON: 0.10000 / TOTAL_REWARD:   8 / AVG_MAX_Q: 2.6045 / AVG_LOSS: 0.00304 / MODE: exploit
EPISODE:  13349 / TIMESTEP:  7506938 / DURATION:  1088 / EPSILON: 0.10000 / TOTAL_REWARD:  19 / AVG_MAX_Q: 2.6468 / AVG_LOSS: 0.00325 / MODE: exploit
EPISODE:  13350 / TIMESTEP:  7507534 / DURATION:   596 / EPSILON: 0.10000 / TOTAL_REWARD:   8 / AVG_MAX_Q: 2.6194 / AVG_LOSS: 0.00247 / MODE: exploit
EPISODE:  13351 / TIMESTEP:  7508096 / DURATION:   562 / EPSILON: 0.10000 / TOTAL_REWARD:   8 / AVG_MAX_Q: 2.6402 / AVG_LOSS: 0.00304 / MODE: exploit
EPISODE:  13352 / TIMESTEP:  7508667 / DURATION:   571 / EPSILON: 0.10000 / TOTAL_REWARD:   8 / AVG_MAX_Q: 2.6044 / AVG_LOSS: 0.00313 / MODE: exploit
EPISODE:  13353 / TIMESTEP:  7509111 / DURATION:   444 / EPSILON: 0.10000 / TOTAL_REWARD:   6 / AVG_MAX_Q: 2.5894 / AVG_LOSS: 0.00239 / MODE: exploit
```

Afterthoughts



- 5min = 300s = 18, 000 frames = 4,500 time steps
- gamma = 0.99 —> Q reset @ < 100 time steps
- Q 'foresees' 5/50 min = 6 seconds….
- Games demanding more temporally extended planning strategies still constitute a major challenge for all existing agents including DQN (for example,Montezuma's Revenge)
- In practice, our algorithm only stores the last N experience tuples in the replay memory, and samples uniformly at random from D when performing updates. This approach is in some respects limited because the memory buffer does not differentiate important transitions and always overwrites with recent transitions owing to the finite memory size N. Similarly, the uniform sampling gives equal importance to all transitions in the replay memory. A more sophisticated sampling strategy might emphasize transitions from which we can learn the most, similar to prioritized sweeping.
- Depending on the problem, variable to initialisation

Figure 1: Example of the training uncertainty on 5 different initialization for DQN (left) and zero-order (right) on the m5v5 scenario.