Deep Q Network

RL with Function Approximation

- Linear value function approximators assume value function is a weighted combination of a set of features
- **Linear VFA** often work well given the right set of features
- But can require careful feature engineering

- An alternative is to use a much richer **non-linear** function approximation
- Deep neural networks(DNN)
  - Universal function approximator
  - Can potentially need exponentially less nodes/parameters (compared to a shallow net) to represent the same function
  - Can learn the parameters using gradient descent

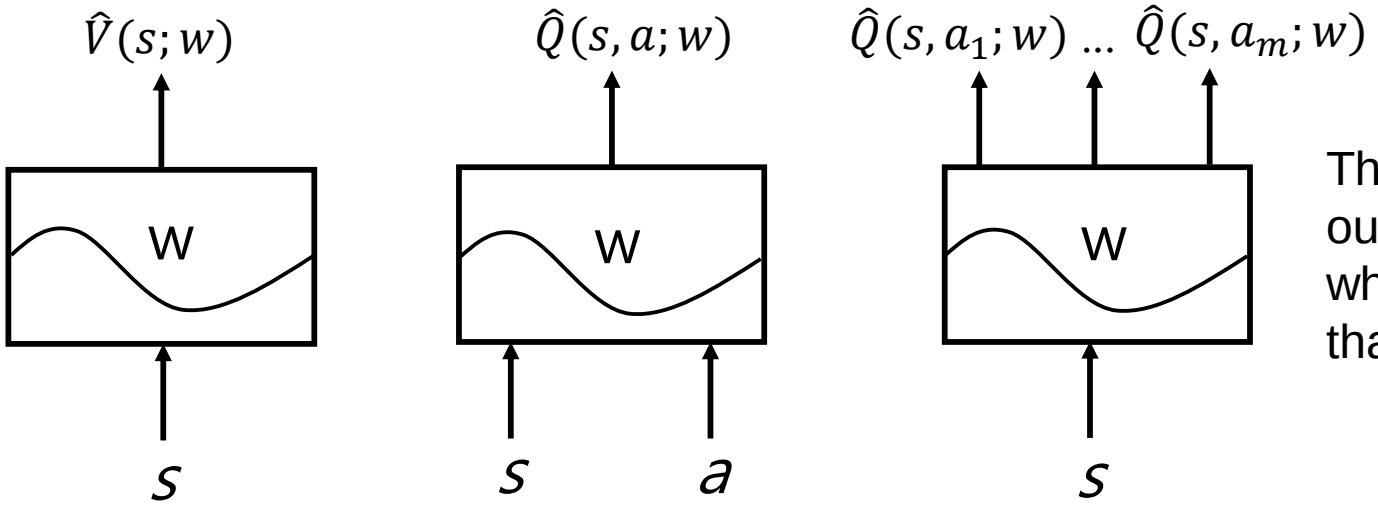
Deep Reinforcement Learning

- Combine Reinforcement Learning with Supervised Learning(e.g., DNN)
- Deep reinforcement learning = reinforcement learning + supervised learning
- Supervised Learning approximates values, policies, rewards, etc.
  - Supervised Learning becomes auxiliary in Reinforcement Learning:
- Represent (state/action) value as deep neural network
  - Value network (e.g.: Deep Q Network)
- Represent policy as deep neural network
  - Policy Network
- Represent both policy and (state/action) value as deep neural network
  - Value network & policy network
  - Actor-Critic Learning
- Represent reward function as deep neural network
  - Inverse Reinforcement Learning

Deep Q-Networks (DQNs)

- Represent state-action value function by Q-network with weights w

$$\hat{Q}(s, a, w) \approx Q(s, a)$$



This neural network has a separate output unit for each possible action, which gives the Q-value estimate for that action given the input state

Q-network is Deep NN => Deep Q-network (DQN)

Deep Q-Networks (DQN)

- Deep Q-Networks (DQN) is Q-learning with a DNN function approximator
- Use a function approximator $\hat{Q}(s, a, w)$ to approximate Q value in Q-learning

$$\hat{Q}(s, a, w) \approx Q(s, a)$$

- Recap: Q-learning

$$Q(S, A) \leftarrow Q(S, A) + \alpha(r + \gamma \max_{a'} Q(S', a') - Q(S, A))$$

- Deep Q-networks (DQN) transfer this to a function approximation solution:

$$w = w + \alpha \left(r + \gamma \max_{a'} \hat{Q}(s', a', w) - \hat{Q}(s, a, w) \right) \frac{\partial \hat{Q}(s, a, w)}{\partial w}$$

DQN

- Define loss function by mean-squared error(MSE) in Q-values

$$L(w) = \frac{1}{2} E_\pi \left(r + \gamma \max_{a'} \hat{Q}(s', a', w) - \hat{Q}(s, a, w) \right)^2$$

- Leading to the following Q-learning gradient (stochastic)

$$\frac{\partial L(w)}{\partial w} = \left(\underbrace{r + \gamma \max_{a'} \hat{Q}(s', a', w)}_{\text{target}} - \underbrace{\hat{Q}(s, a, w)}_{\text{Q network output}} \right) \frac{\partial \hat{Q}(s, a, w)}{\partial w}$$

Target value is implemented as a separate network (target network)

- Minimize loss function by stochastic gradient descent using $\frac{\partial L(w)}{\partial w}$

$$w = w - \alpha \frac{\partial L(w)}{\partial w}$$

Stability Issues with Deep RL

- Native Q-learning oscillates or diverges with neural nets

1. Data are not independent
   - Successive samples are correlated
   - Non-iid(**i**ndependent & **i**dentically **d**istributed)

2. Policy changes rapidly (non-stationary) with slight changes to Q-values
   - Policy may oscillate
   - Distribution of data can swing from one extreme to another

3. Scale of rewards and Q-values is unknown
   - Native Q-learning gradients can be large
   - Unstable when backpropagated

Stability Issues with Deep RL

- Solutions of stability issues in DQN:

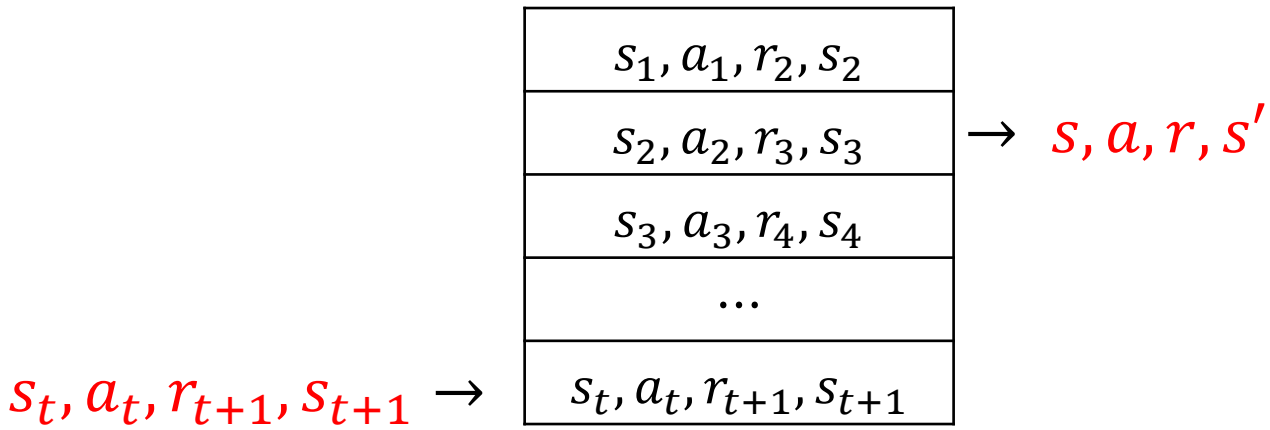
1. Use experience replay
   - Break correlations in data, bring us back to iid setting
   - Learn from all past policies

2. Freeze target Q-network
   - Avoid oscillations
   - Break correlations between Q-network and target

3. Clip rewards or normalize network adaptively to sensible range
   - Robust gradients

Experience Replay

- Store the agent's experiences at each time step $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ in a dataset $D = e_1, e_2, \ldots, e_n$ pooled over many episodes into a replay memory



- You may store the last N experience tuples in the replay memory and sample uniformly from D when performing updates

Experience Replay

- Optimize MSE between Q-network and Q-learning targets, e.g.

$$L(w) = E_{s,a,r,s' \sim D}\left[\left(r + \gamma \max_{a'} \hat{Q}(s', a', w) - \hat{Q}(s, a, w)\right)^2\right]$$

$$\frac{\partial L(w)}{\partial w} = E_{s,a,r,s' \sim D}\left[\left(r + \gamma \max_{a'} \hat{Q}(s', a', w) - \hat{Q}(s, a, w)\right) \frac{\partial \hat{Q}(s,a,w)}{\partial w}\right] \text{ (batch)}$$

- DQN algorithm with Experience Replay:
- Loop until converge
  - Take action a_t according to ε-greedy policy
  - Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory *D*
  - Sample random mini-batch of transitions (s, a, r, s') from *D*
  - Compute the target value for the sampled s: $r + \gamma \max_{a'} \hat{Q}(s', a', w)$

$$w = w - \alpha\left(r + \gamma \max_{a'} \hat{Q}(s', a', w) - \hat{Q}(s, a, w)\right) \frac{\partial \hat{Q}(s,a,w)}{\partial w} \text{ (stochastic)}$$

Prioritized Experience Replay

- In experience replay, data are selected randomly.
- Some data are more important than others
- prioritized experience replay: each data has different weight
- For a certain data $(s_i, a_i, r_{i+1}, s_{i+1})$, define priority function p_i
- priority function: Importance of data

$$p_i = \left| r_{i+1} + \gamma \max_{a'} \hat{q}(s_{i+1}, a', w) - \hat{q}(s_i, a_i, w) \right| + \epsilon$$

- Suppose $P(i)$ is the prob. of the data being selected

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

  - $P(i)$ is less sensitive to outliers
  - more robust

Target Q-Network

- Target network uses a different set of weights than the weights being updated
- To avoid non-stationary/ocsillations, fix parameters used in Q-learning target
- Let parameters w^- in target network, and w be parameters in prediction network
  - Compute Q-learning targets w.r.t. old, fixed parameters w^-

$$r + \gamma \max_{a'} \hat{Q}(s', a', w^-)$$

  - Optimize MSE between prediction Q-network and target Q-network

$$L(w) = E_{s,a,r,s' \sim D}[\left(r + \gamma \max_{a'} \hat{Q}(s', a', w^-) - \hat{Q}(s, a, w)\right)^2]$$

$$\frac{\partial L(w)}{\partial w} = E_{s,a,r,s' \sim D}[(r + \gamma \max_{a'} \hat{Q}(s', a', w^-) - \hat{Q}(s, a, w)) \frac{\partial \hat{Q}(s, a, w)}{\partial w}]$$

- Periodically update fixed parameters: $w^- \leftarrow w$ or
- Polyak update (κ: Polyak constant)

$$w^- \leftarrow \kappa w + (1 - \kappa) w^-$$

Reward/Value Range

- DQN clips the rewards to [-1,+1]
- This prevents Q-values from becoming too large
- Ensure gradients are well-conditioned

DQN Pseudo Code

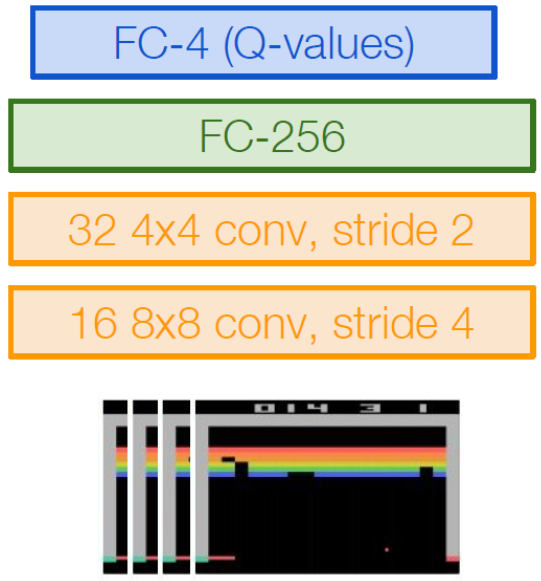
loop (converge)
// sample
perform action a_t in s_t using ϵ-greedy policy
generate r_{t+1} and next state s_{t+1}
store $(s_t, a_t, r_{t+1}, s_{t+1})$ in memory *D*
// train
sample minibatch M data from *D*
    for $(s, a, r, s') \in \mathrm{M}$ **do**
        compute $\hat{q}(s, a, w)$ using prediction network
        compute $y_s = r + \gamma \max_{a'} \hat{q}(s', a', w^-)$ using target network
            if (s' = final state) then $y_s = r$
        $w \leftarrow w - \alpha(y_s - \hat{q}(s, a, w)) \frac{\partial \hat{q}(s,a,w)}{\partial w}$ (stochastic update)
    end for
$t \leftarrow t + 1$
periodically update w^- using w
end loop

DQNs in Atari

Q-network Architecture

$Q(s, a; \theta)$:
neural network
with weights θ

A single feedforward pass to compute Q-values for all actions from the current state => efficient!



Last FC layer has 4-d output (if 4 actions), corresponding to $Q(s_t, a_1)$, $Q(s_t, a_2)$, $Q(s_t, a_3)$, $Q(s_t, a_4)$

Number of actions between 4-18 depending on Atari game

Current state s_t: 84x84x4 stack of last 4 frames
(after RGB->grayscale conversion, downsampling, and cropping)

Source: cs231n Reinforcement learning, Stanford

Double DQN

- Recap: Double Q learning

$$q_{\mathrm{A}}(s,a) \leftarrow q_{\mathrm{A}}(s,a) + \alpha(r + \gamma q_B(s', a_*) - q_{\mathrm{A}}(s,a))$$

- Double Q learning keeps two q value sets: q_{A} and q_B

- Extend Double Q learning to DQN
- Double DQN keeps two DQNs: $\hat{q}(s,a,w_A)$ and $\hat{q}(s,a,w_B)$
  - $\hat{q}(s,a,w_A)$ and $\hat{q}(s,a,w_B)$ should be two independent networks

- Basic DQN update formula

$$w = w + \alpha\left(r + \gamma \max_{a'} \hat{q}(s',a',w) - \hat{q}(s,a,w)\right)\frac{\partial \hat{q}(s,a,w)}{\partial w}$$

- Double DQN update formula

$$w_A = w_A + \alpha\left(r + \gamma \hat{q}(s', a_*, w_B) - \hat{q}(s,a,w_A)\right)\frac{\partial \hat{q}(s,a,w_A)}{\partial w_A}$$

$$\text{where } a_* = \operatorname*{argmax}_a \hat{q}(s',a,w_A)$$

Double DQN with Target Network

- When $\hat{q}(s,a,w_A)$ and $\hat{q}(s,a,w_B)$ are trained with target network, we may need 4 separate networks
  1) $\hat{q}(s,a,w_A)$ and its target network $\hat{q}(s,a,w_A^-)$
  2) $\hat{q}(s,a,w_B)$ and its target network $\hat{q}(s,a,w_B^-)$

- Compromise:
  - Current Q-network w is used to select actions: ($\hat{q}(s,a,w_A)$ -> $\hat{q}(s,a,w)$)
  - Target Q-network w^- is used to evaluate actions: ($\hat{q}(s,a,w_B)$ -> $\hat{q}(s,a,w^-)$)

$$r + \gamma \hat{q}\left(s', \underset{a'}{\operatorname{argmax}}\, \hat{q}(s',a',w), w^-\right) - \hat{q}(s,a,w)$$

- $\hat{q}(s,a,w)$ and $\hat{q}(s,a,w^-)$ are not completely independent, but works pretty well

Double DQN with Target Network

Double DQN w Target Network

Initialize parameter of DQN w & target network w^-
Repeat the following with different s_t
loop (until converge)
Perform action a_t in state s_t using ϵ-greedy
Generate r_{t+1} and s_{t+1}
Store $(s_t, a_t, r_{t+1}, s_{t+1})$ in memory D
Mini-batch sample B from D
for $(s, a, r, s') \in$ B do

$$y = r + \gamma \hat{q}\left(s', \max_{a'} \hat{q}(s', a', w), w^-\right)$$

$$w = w + \alpha\left(y - \hat{q}(s, a, w)\right)\frac{\partial \hat{q}(s,a,w)}{\partial w}$$ (stochastic)

end for

$$t \leftarrow t + 1$$

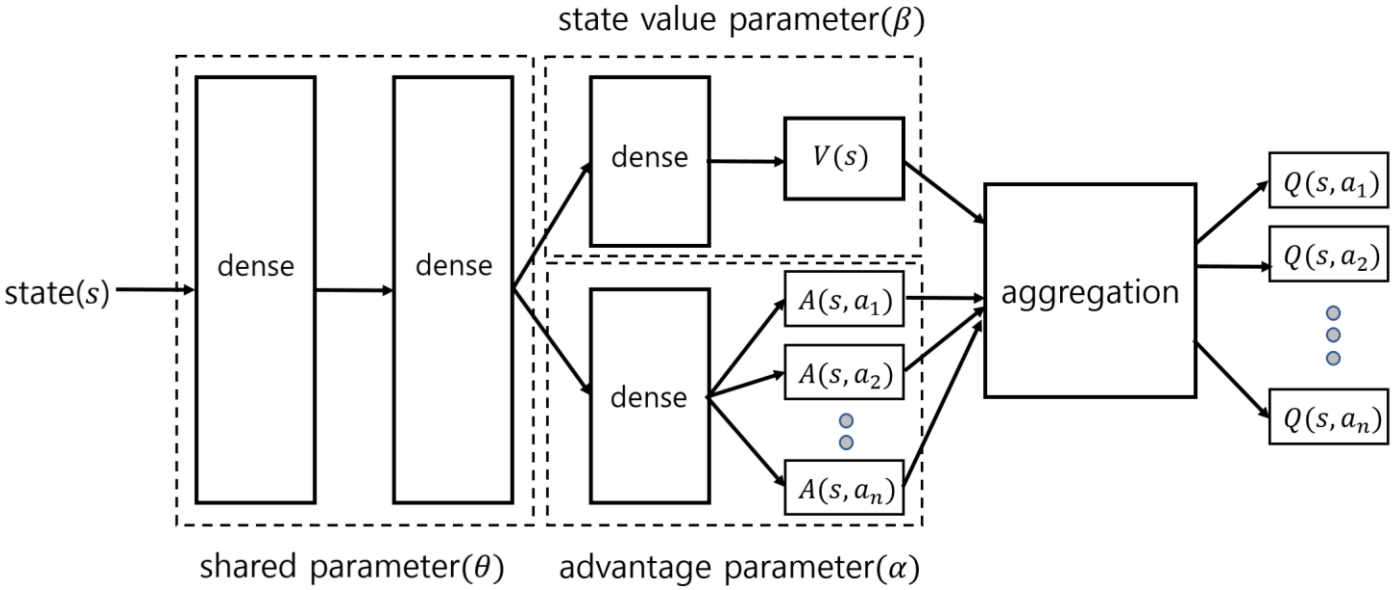
Update w^- from w periodically
end loop

Duel DQN

- Q value represents the value of choosing a specific action at a given state
- V value represents the value of the given state regardless of the action taken.
- A(advantage) value shows how advantageous an action is relative to the others at the state.

$$A(s,a) = Q(s,a) - V(s)$$

- In many cases, it is unnecessary to know the value of each action at every timestep
- By explicitly separating two estimators (V and Q), the dueling architecture can learn which states are (or are not) valuable, without having to learn the effect of each action for each state



Duel DQN

- Now, how do we combine/aggregate the two values?

$$Q(s,a) = V(s) + A(s,a)$$

- How do we compute V value and A value from Q value.
- The naive sum of the two is “unidentifiable,” in that given the Q value, we cannot recover the V and A uniquely.

- Two prior information
  - Since $V_\pi(s) = E_{a\sim\pi}[Q(s,a)]$ & Q=V+A, $E_{a\sim\pi}[A(s,a)] = 0$

  - In deterministic policy, we choose greedy action.

$$a_* = \underset{a'}{\operatorname{argmax}}\, Q(s,a')$$

therefore,

$$Q(s,a_*) = V(s)$$

and

$$A(s,a_*) = A\left(s, \underset{a'}{\operatorname{argmax}}\, Q(s,a')\right) = 0$$

Duel DQN

- Therefore, the last module of the neural network implements forward mapping shown below

$$A(s,a;\theta,\alpha) - \max_{a'} A(s,a';\theta,\alpha)$$

- To solve this, force the Q value for the maximizing action to equal V, solving the identifiability issue.
- Force the highest Q-value to be equal to the value V, thus making the highest value in the advantage function be zero and all other values negative

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \max_{a'} A(s,a';\theta,\alpha)\right)$$

- Over time, $A(s,a;\theta,\alpha) - \max_{a'} A(s,a';\theta,\alpha)$ becomes zero ($A(s,a_*) = 0$)

- A small change to above. Instead of calculating the max, we replace it with the mean

$$A(s,a;\theta,\alpha) - \frac{1}{|A|}\sum_{a'} A(s,a';\theta,\alpha) \qquad (E_{a\sim\pi}[A(s,a)] = 0)$$

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|A|}\sum_{a'} A(s,a';\theta,\alpha)\right)$$

- We then choose the optimal action a_* (or Q) based on

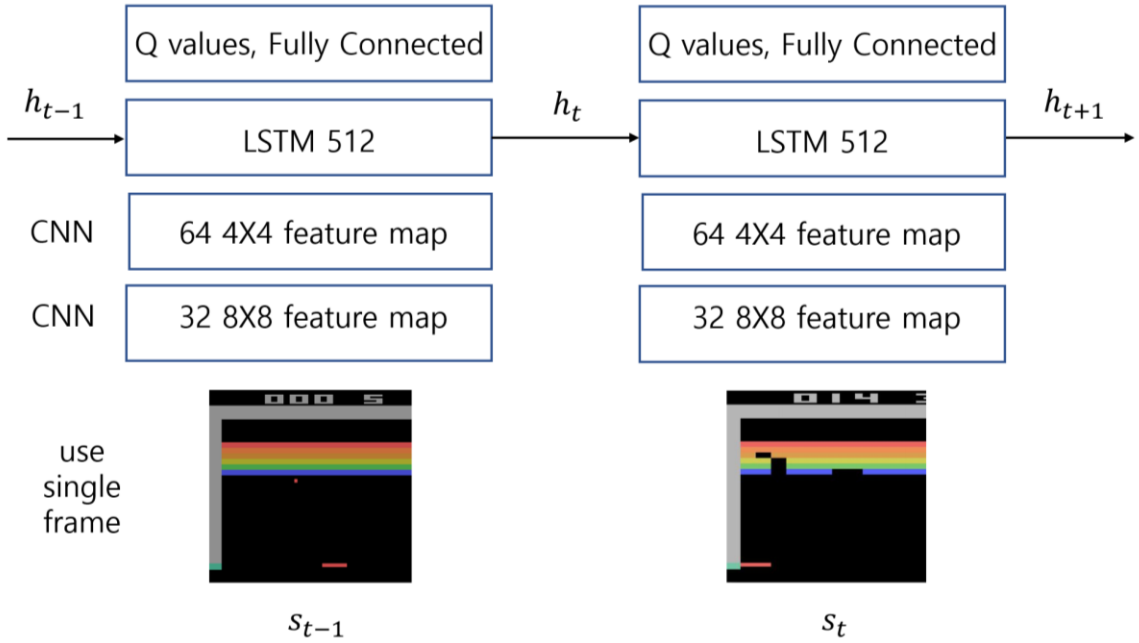
$$a_* = \max_{a} Q(s,a;\theta,\alpha,\beta)$$

Deep Recurrent Q Network(DRQN)

- Many real-world tasks are mostly Partially-Observable Markov Decision Process (POMDP)

- In Atari game, with the current screen, it can only observe location of padles and the ball not velocity of the ball (POMDP model)
  - Instead, uses 4 previous screens (now MDP model)
  - DQN has a limitation that can only remember last four screens in the past.
  - Games that require more than four frames are non-Markovian because it depends on more than just DQN's current input.

- The DQN architecture is modified: The first fully-connected layer is replaced with a recurrent LSTM layer of the same size.



Deep Recurrent Q Network(DRQN)

- DQRN is trained using BTTP(backpropagation through time)

- Experience Replay:
- Entire episodes must be stored
- Two types of updates: sequential and random
  1) sequential update: entire episode is used to update
     - learn faster because they can carry forward the LSTM hidden state
     - but violate DQN's independent sample assumption.
  2) random update: each step of episode is used to update
     - cannot learn as quickly as sequential update
     - Don't violate the independence assumption.
- Both strategies are shown to perform well.