Case Studies

I-Chen Wu

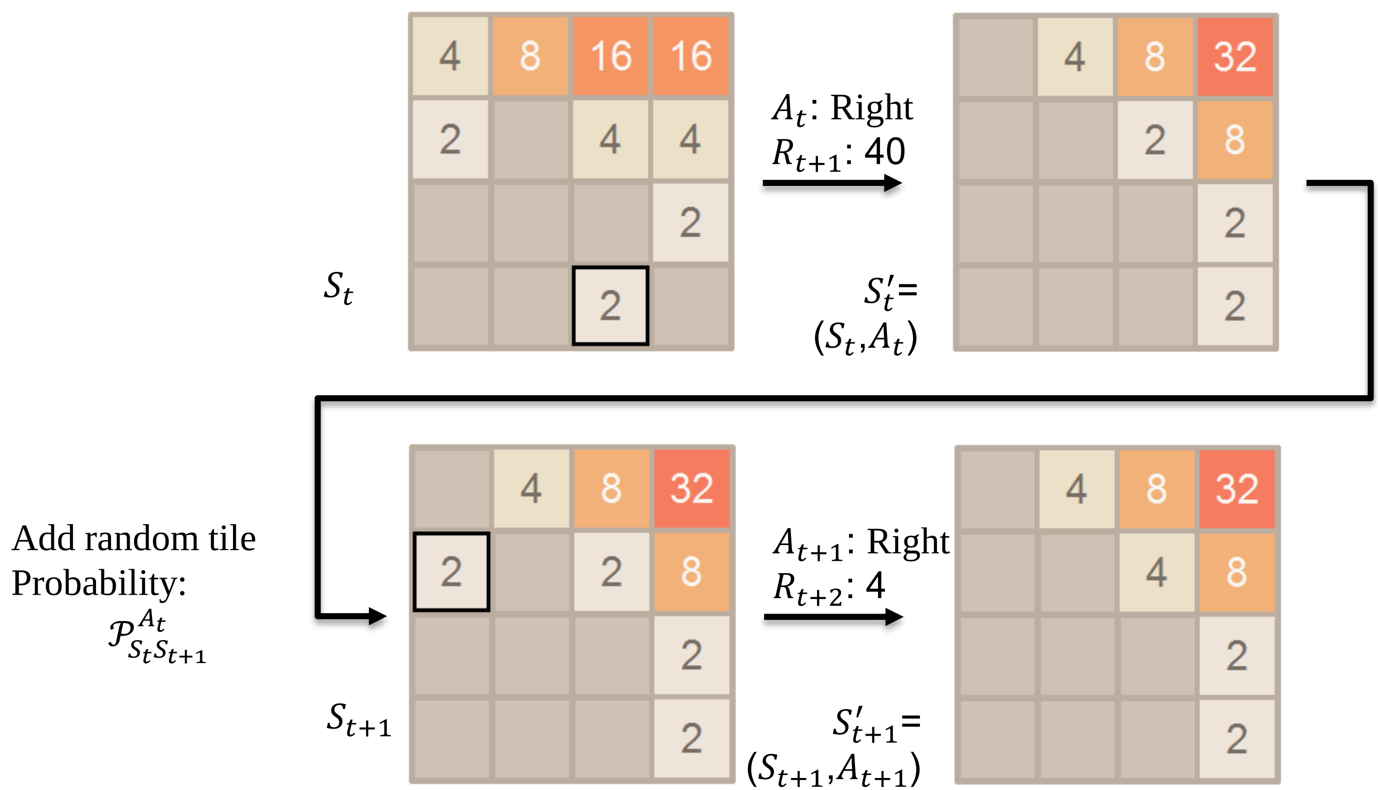
- David Silver, Online Course for Deep Reinforcement Learning.
  - http://www.cs.ucl.ac.uk/staff/D.Silver/web/Teaching.html
- M. Szubert and W. Jaśkowski, “Temporal difference learning of n-tuple networks for the game 2048,” *2014 IEEE Conference on Computational Intelligence and Games (CIG),* Aug. 2014, pp. 1–8.
- Kun-Hao Yeh, et al., Multi-Stage Temporal Difference Learning for 2048-like Games, accepted by IEEE Transactions on Computational Intelligence and AI in Games (SCI), doi: 10.1109/TCIAIG.2016.2593710, 2016.
- Mnih, V. et al. Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015).

Cases

- 2048
  - Temporal Difference (TD) Learning
  - N-tuple networks
- Atari games
  - Temporal Difference (TD) Learning
  - Deep Q-networks (DQN), a kind of Deep NN
- Go Programs (with Monte-Carlo Tree Search) – to be added
  - Monte-Carlo (MC) Learning
  - Multi-Armed Bandits
  - Planning
- AlphaGo (with Reinforcement Learning) – to be added.
  - Monte-Carlo (MC) Learning
  - Policy Gradient
- Pole Balancing – to be added.
  - Policy Gradient
  - Actor-Critic

Case Study: 2048

- [Szubert et al., 2014; Yeh et al., 2016]



2048 RL Agent

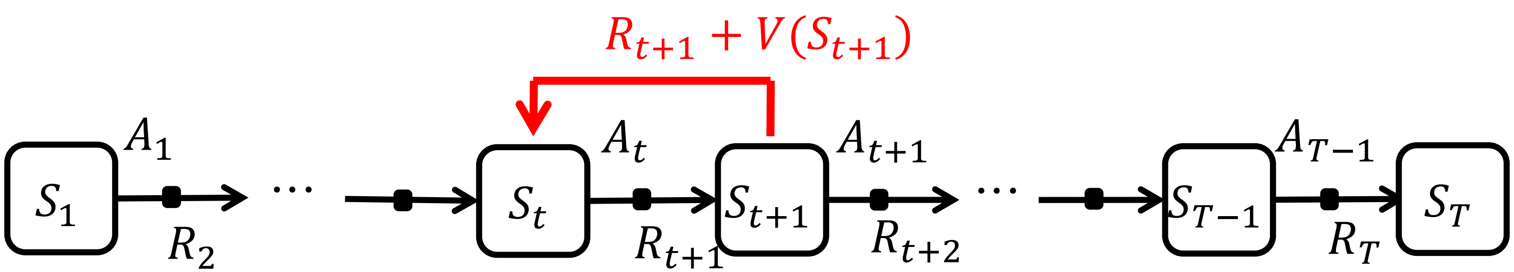
- Value function:
  - The expected score/return G_t from a board S
  - But, #states is huge
    - About 17^{16} ($=10^{20}$).
      - Empty, 2 ($=2^1$), 4 ($=2^2$), 8 ($=2^3$), …, 65536 ($=2^{16}$).
  - Need to use value function approximator.
- Policy:
  - Simply choose the action (move) with the maximal value based on the approximator.
- Model: agent's representation of the environment
  - After a move, randomly generate a tile:
    - 2-tile: with probability of 9/10
    - 4-tile: with probability of 1/10
  - Reward: simply follow the rule of 2048.

TD Learning in 2048

- State value function: (Normally $\gamma = 1$)
  - Update value $V(S_t)$ toward TD target $R_{t+1} + \gamma V(S_{t+1})$

    $V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$
- Making decision (based on value).

  $\pi(s) = argmax_a(R_{t+1} + \mathbb{E}[V(S_{t+1}) \mid S_t = s, A_t = a])$
  - Problem: Less efficient upon making decision.



Q-Learning in 2048

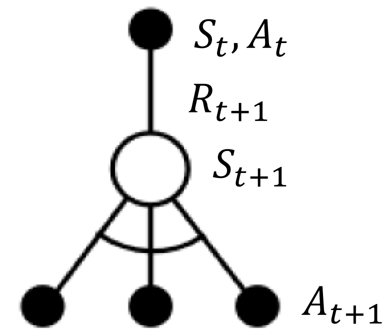
- Q-value function: (Normally $\gamma = 1$)
  - Update value $Q(S_t, A_t)$ toward TD target $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$

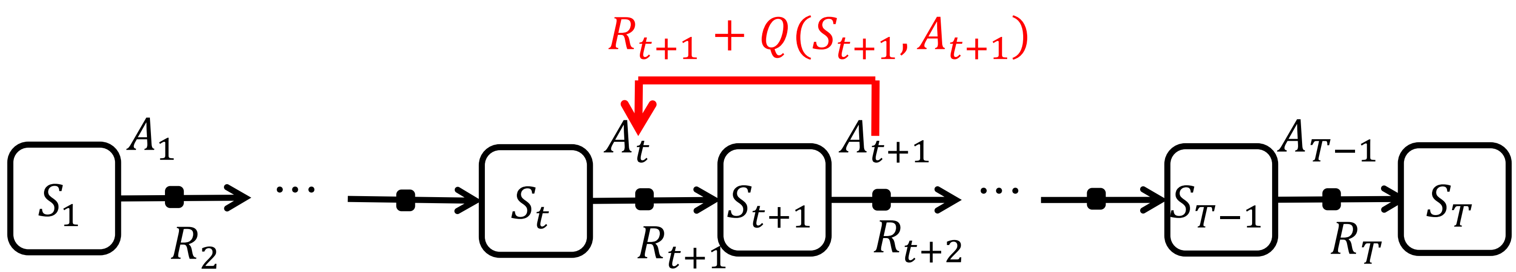
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right)$$

- Making decision (based on value).

$$\pi(s) = argmax_a\left(Q(S_t, A_t)\right)$$

  - more efficient.
  - A minor problem: Four times more memory





Afterstates in 2048

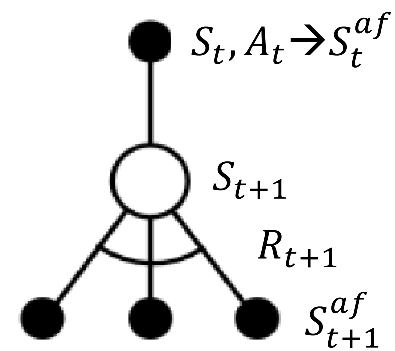
- Afterstate S_t^{af} is a state after action A_t at S_t.
  - Why not use S_t^{af} instead of (S_t, A_t)?
  - Note: in 2048, the reward R_{t+1} is not included in S_t^{af}.
- Afterstate value function: (Normally $\gamma = 1$)
  - Update value $V^{af}(S_t^{af})$ toward TD target $\gamma \max_a (R_{t+1} + V^{af}(S_{t+1}^{af}))$

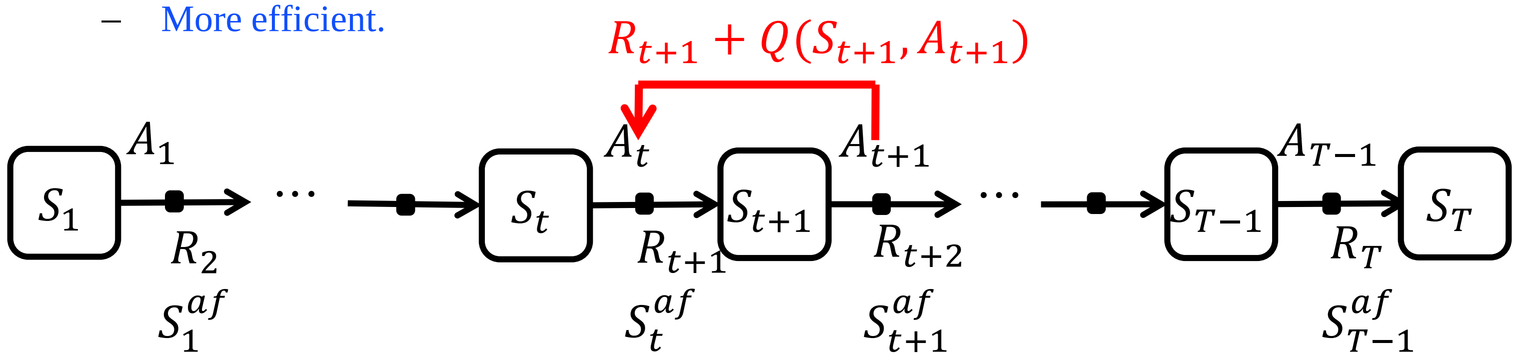
$$V^{af}(S_t^{af}) \leftarrow V^{af}(S_t^{af}) + \alpha(\gamma \max_a (R_{t+1} + V^{af}(S_{t+1}^{af})) - V^{af}(S_t^{af}))$$

- Making decision (based on value).

$$\pi(s) = argmax_a \left(V^{af}(S_t^{af})\right)$$

  - For simplicity, we use V, instead of V^{af}, if it can be applied to both.
  - More efficient.





Value Function Approximation

- As mentioned above, #states is huge, so we need to use value function approximation.
  - Use a value function approximator, $\hat{v}(S, w) \approx V(S)$.
  - Simply use deterministic policy: $\pi(S) = argmax_a(\hat{v}(S, w))$
- But, what kind of value function approximator can we use?
  - What features can we choose?
    - Traditionally, # of empty cells, # of continuous cells, big tiles, etc.
  - Linear (like n-tuple network) vs. non-linear (like NN)
- n-tuple network is a powerful network for 2048.
  - Explore a large set of features.
  - Simplify operations by linear value function approximation.

Linear Value Function Approximation

- Represent value function by a linear combination of features

  $$\hat{v}(S;\theta) = x(S)^{\mathrm{T}}\theta = \sum_{j=1}^{n} x_j(S)\theta_j$$

- Gradient of $\hat{v}(S,\theta)$:

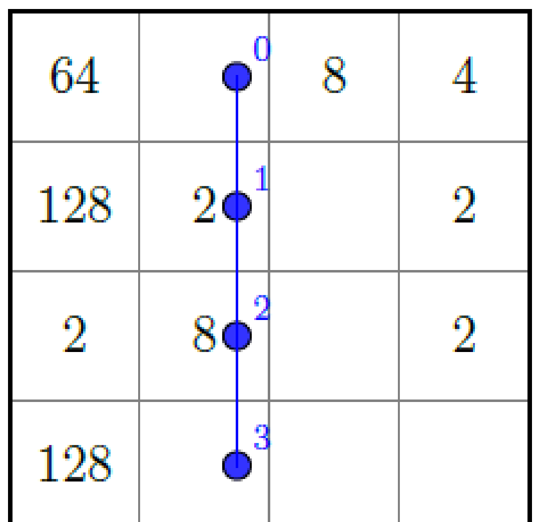
  $$\nabla_\theta \hat{v}(S,\theta) = x(S)$$

Gradient Descent

- Update value $V(S_t)$ towards TD target $y_t = R_{t+1} + V(S_{t+1})$
  $$\Delta V = (R_{t+1} + V(S_{t+1}) - V(S_t)) = (y_t - V(S_t))$$
  $$V(S_t) \leftarrow V(S_t) + \alpha \Delta V$$
  - α: learning rate, or called step size.
  - Note: $\gamma = 1$ here.
- Objective function is to minimize the following loss in parameter θ. (note: $\hat{v}(S, \theta) = x(S)^{\mathrm{T}}\theta$)
  $$\mathcal{L}(w) = \mathbb{E}\left[\left(y_t - \hat{v}(S, \theta)\right)^2\right]$$
  $$\nabla_\theta \mathcal{L}(\theta) = \left(y_t - \hat{v}(S, \theta)\right) \cdot \nabla_\theta \hat{v}(S, \theta) = \Delta V \cdot x(S)$$
- Update features w: step-size * prediction error * feature value
  $$\theta \leftarrow \theta + \alpha \Delta V \cdot x(S)$$

N-Tuple Network

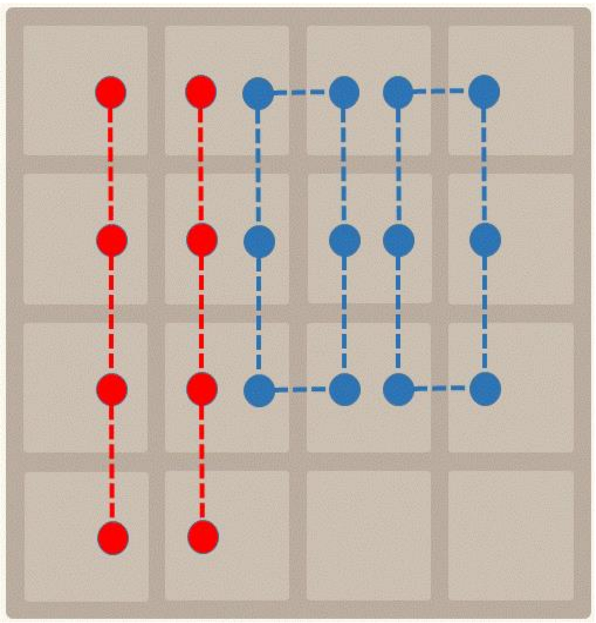
- Example: 4-tuple networks as shown.
  - Each cell has 16 different tiles
  - 16^4 features for this network.
    - But only one is on, others are 0.
    - So, we can use table lookup to find the feature weight.

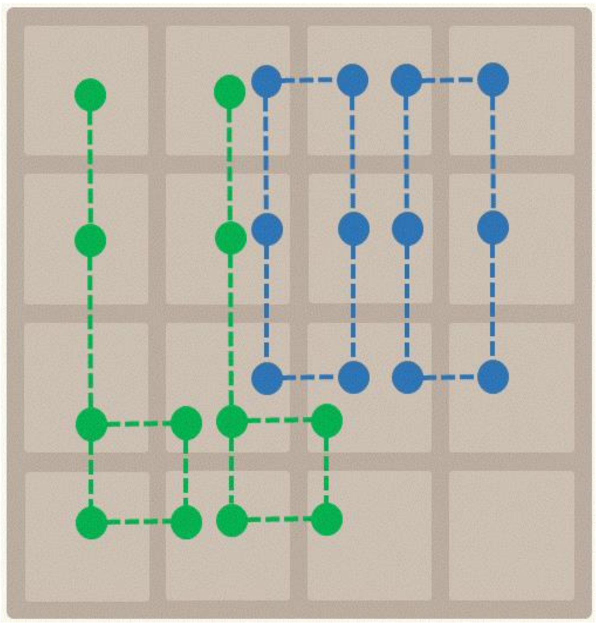


0123	weight
0000	3.04
0001	-3.90
0002	-2.14
⋮	⋮
0010	5.89
⋮	⋮
0130	**-2.01**
⋮	⋮

Other N-Tuple Networks

- Left: [Szubert et al., 2014]; Right: [Yeh et al., 2016]
- Some researchers even used 7-tuple network.





Update Features in N-Tuple Networks

- For n-tuple networks, simply update values with $\alpha\Delta V$ at $LUT_i[index(s_i)]$
- Features:
  - 8 x 16^4 features, $x(S) = [\underline{0, 1, 0, \ldots,}\ \underline{0, 0, 1, \ldots,} \ldots, \underline{1, 0, 0, \ldots}]$
    - All 0s, except for 8 ones.
      - One 1 every 16^4 features.
      - Let their indices be $\boldsymbol{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8}$.
  - Only need to update $\alpha\Delta V$ at the features indexed by these indices.
  - Very efficient and fast.
- For k n-tuple networks,

$$\hat{v}(S,\theta) = x(S)^{\mathrm{T}}\theta = \sum_{j=1}^{n} x_j(S)\theta_j = \sum_{i=1}^{k} LUT_i[index(s_i)]$$

  - LUT_i: the i-th n-tuple network lookup table.
  - $index(s_i)$: The index in the i-th n-tuple network of state S.
- Update features w: step-size * prediction error * feature value
  - $\theta \leftarrow \theta + \alpha\Delta V \cdot x(S)$
  - Only need to update values θ_j with $\alpha\Delta V$ at $LUT_i[index(s_i)]$.

Afterstate Evaluation Function

function $\textsc{Evaluate}(s, a)$

  $s', r \leftarrow \textsc{Compute Afterstate}(s, a)$

  return $r + V(s')$

function $\textsc{Learn Evaluation}(s, a, r, s', s'')$

  $a_{next} \leftarrow \arg\max_{a' \in A(s'')} \textsc{Evaluate}(s'', a')$

  $s'_{next}, r_{next} \leftarrow \textsc{Compute Afterstate}(s'', a_{next})$

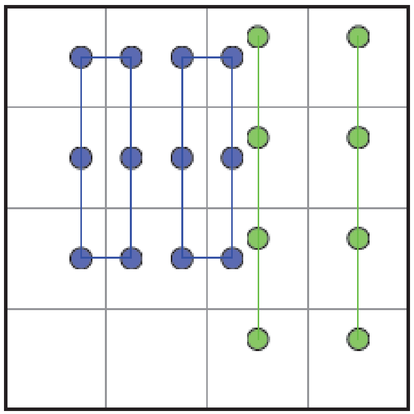
  $V(s') \leftarrow V(s') + \alpha(r_{next} + V(s'_{next}) - V(s'))$

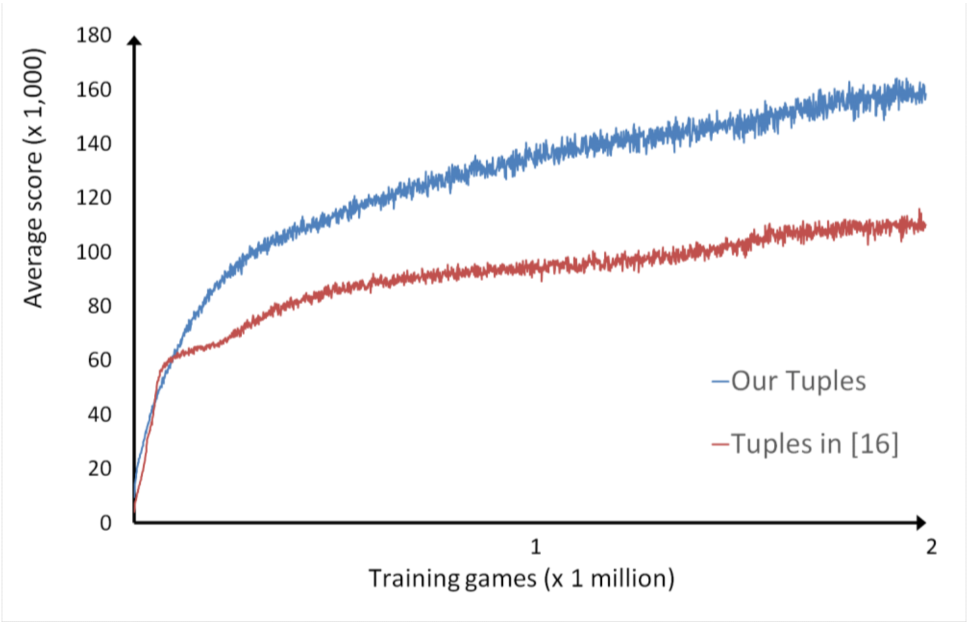
```
function PLAY GAME
    score ← 0
    s ← INITIALIZE GAME STATE
    while ¬IS TERMINAL STATE(s) do
        a ← arg max_{a'∈A(s)} EVALUATE(s, a')
        r, s', s'' ← MAKE MOVE(s, a)
        if LEARNING ENABLED then
            LEARN EVALUATION(s, a, r, s', s'')
        score ← score + r
        s ← s''
    return score

function MAKE MOVE(s, a)
    s', r ← COMPUTE AFTERSTATE(s, a)
    s'' ← ADD RANDOM TILE(s')
    return (r, s', s'')
```

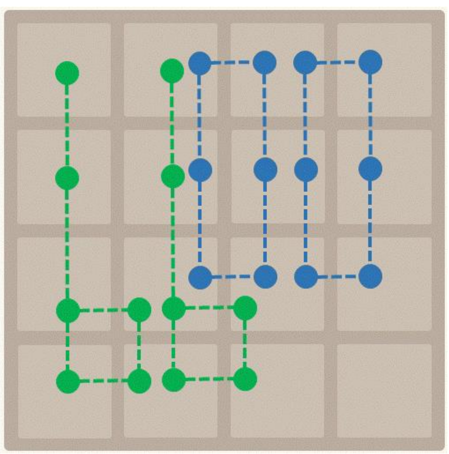
The N-Tuple Networks Used

- Use the following [Szubert and Jaskowaski 2014]





- Ours:



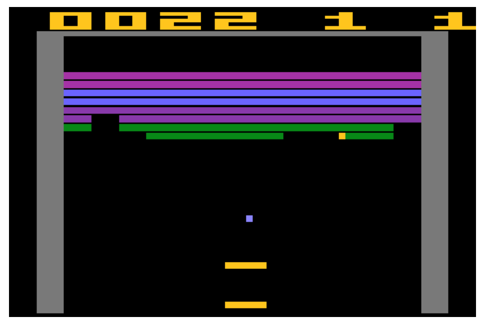
Performance Results (without search)

2048 rate	**100%**
4096 rate	100%
8192 rate	99.20%
16384 rate	83.30%
32768 rate	8.10%
Maximum score	607488
Average score	331820

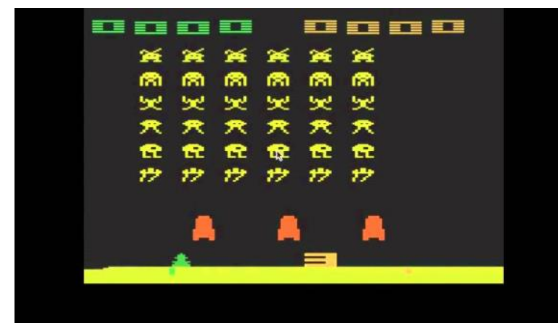
Case Study: Atari 2600 Games

- Learn to play Atari games **from video only** (without knowing the game a priori)

- Atari 2600

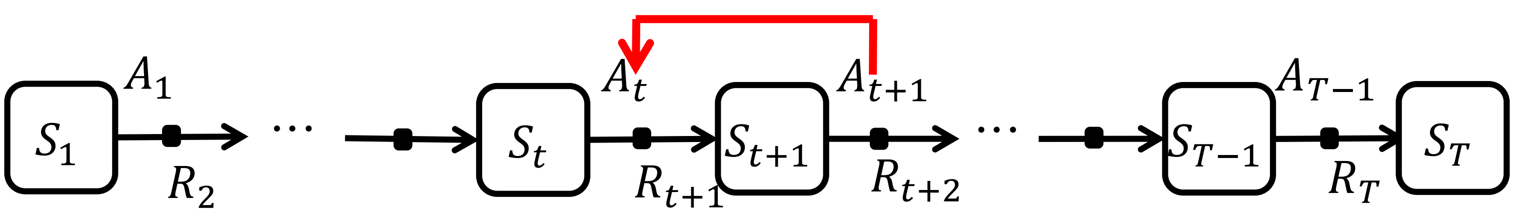


- Breakout



- Space Invaders

$R_{t+1} + Q(S_{t+1}, A_{t+1})$



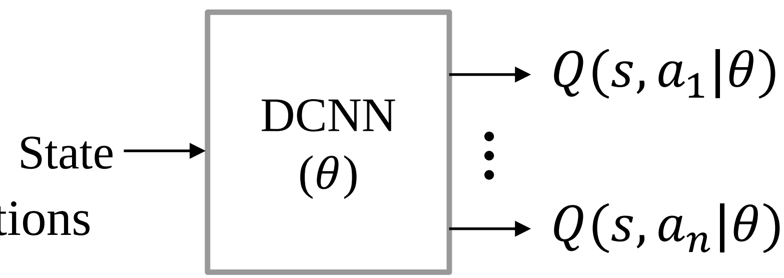
Deep Q-Networks (DQN)

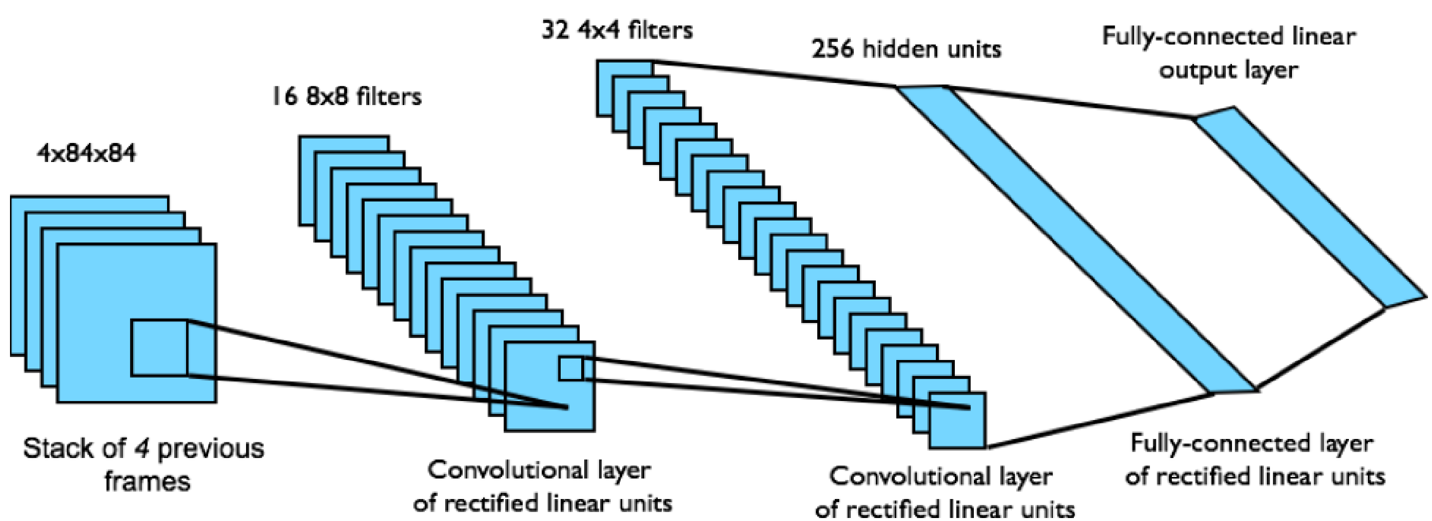
DQN uses experience replay and fixed Q-targets

- Take action 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 θ^-
- Optimize MSE between Q-network and Q-learning targets
  - Minimize a sequence of loss functions $\mathcal{L}(\theta_i)$ that changes at each iteration i.
  - $\mathcal{L}_i(\theta_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right)^2\right]$
- Using variant of stochastic gradient descent
  - Differentiating the loss function with respect to the weights we arrive at the following gradient
  - $\nabla_{\theta_i}\mathcal{L}_i(\theta_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right) \cdot \nabla_{\theta_i} Q(s, a; \theta_i)\right]$

DQN in Atari

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





Performance of Deep Q-Learning

- Left (stronger than human)