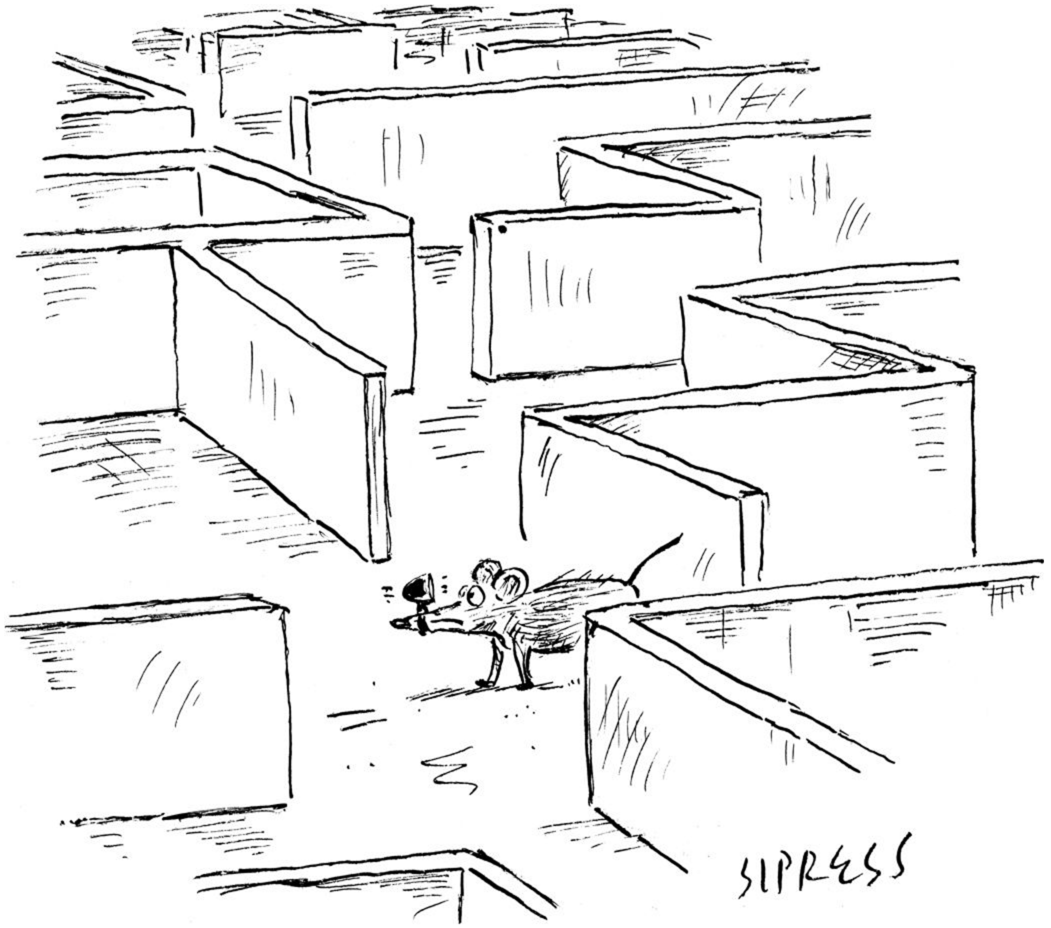


"Recalculating … recalculating …"

CIS 522 Lecture 18

Introduction to Reinforcement Learning – Part 2
3/31/20

Today

- DQN
- A3C
- AlphaGo

...lots of DeepMind

- Most of today ended up being DeepMind algorithms
- DeepMind
    - Research company, focused on deep RL
    - Owned by Alphabet (Google parent company)
- OpenAI
    - Another major non-academic org focused on deep RL
    - Started only in 2015, so will discuss more in next class on recent research

DQN

Review of reinforcement learning terms

"Agent" - Something/someone that is learning

"Environment" - The world where the agent is learning

"State" - What the agent observes

"Action" - What the agent chooses to do

"Policy" - The rule (can be stochastic) by which the agent picks actions

"Reward" - Something the agent gets for certain (state, action) pairs

Review of Q-learning

Want an evaluation Q(state, action) of expected (discounted) reward under policy π:

$$Q_\pi(s,a) = \mathbb{E}_\pi\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right)$$

In particular, want Q*(state, action) for optimum policy:

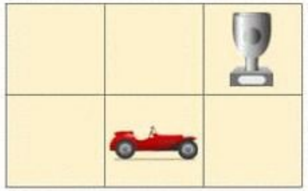
$$Q^*(s,a) = \max_\pi \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t = s,\ a_t = a,\ \pi\right],$$

Q-learning - an initial Q function converges under update rule:

$$Q(s_t, a_a) = Q(s_t, a_a) + \alpha(r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

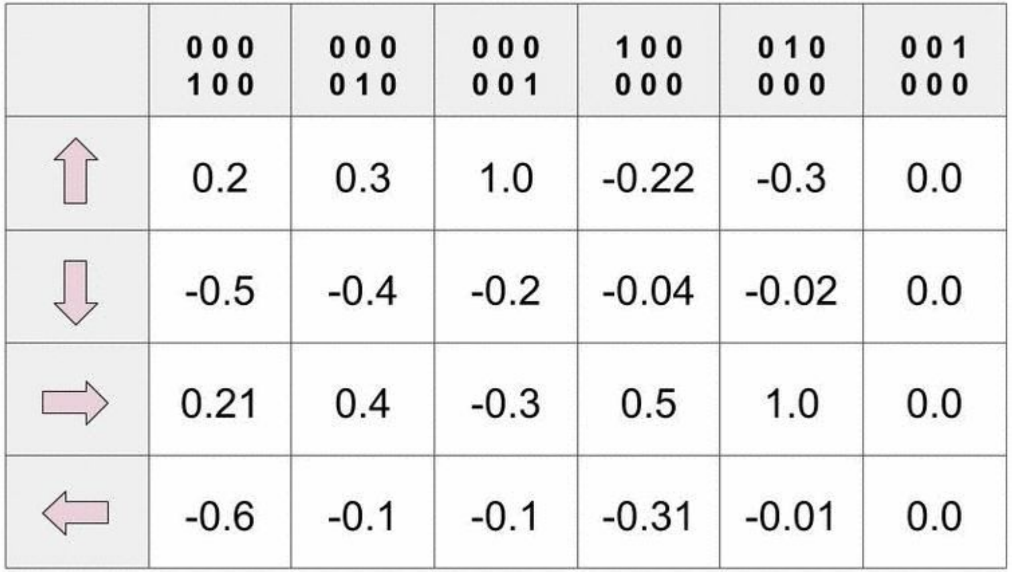
Standard Q-learning

Game Board:



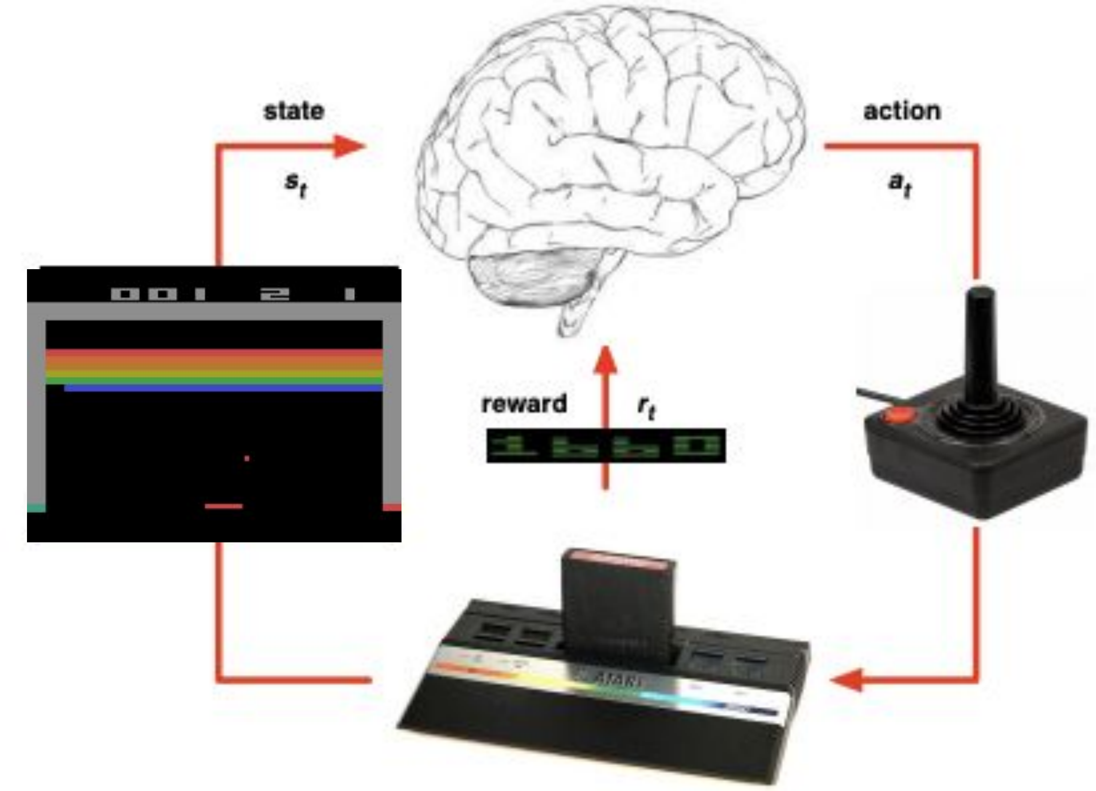
Current state (*s*): $\begin{matrix}0&0&0\\0&1&0\end{matrix}$

Q Table: $\gamma = 0.95$



	$\begin{matrix}0&0&0\\1&0&0\end{matrix}$	$\begin{matrix}0&0&0\\0&1&0\end{matrix}$	$\begin{matrix}0&0&0\\0&0&1\end{matrix}$	$\begin{matrix}1&0&0\\0&0&0\end{matrix}$	$\begin{matrix}0&1&0\\0&0&0\end{matrix}$	$\begin{matrix}0&0&1\\0&0&0\end{matrix}$
⇧	0.2	0.3	1.0	-0.22	-0.3	0.0
⇩	-0.5	-0.4	-0.2	-0.04	-0.02	0.0
⇨	0.21	0.4	-0.3	0.5	1.0	0.0
⇦	-0.6	-0.1	-0.1	-0.31	-0.01	0.0

Could one do this to learn Atari games?



Why might this not work?

Deep Q-Network

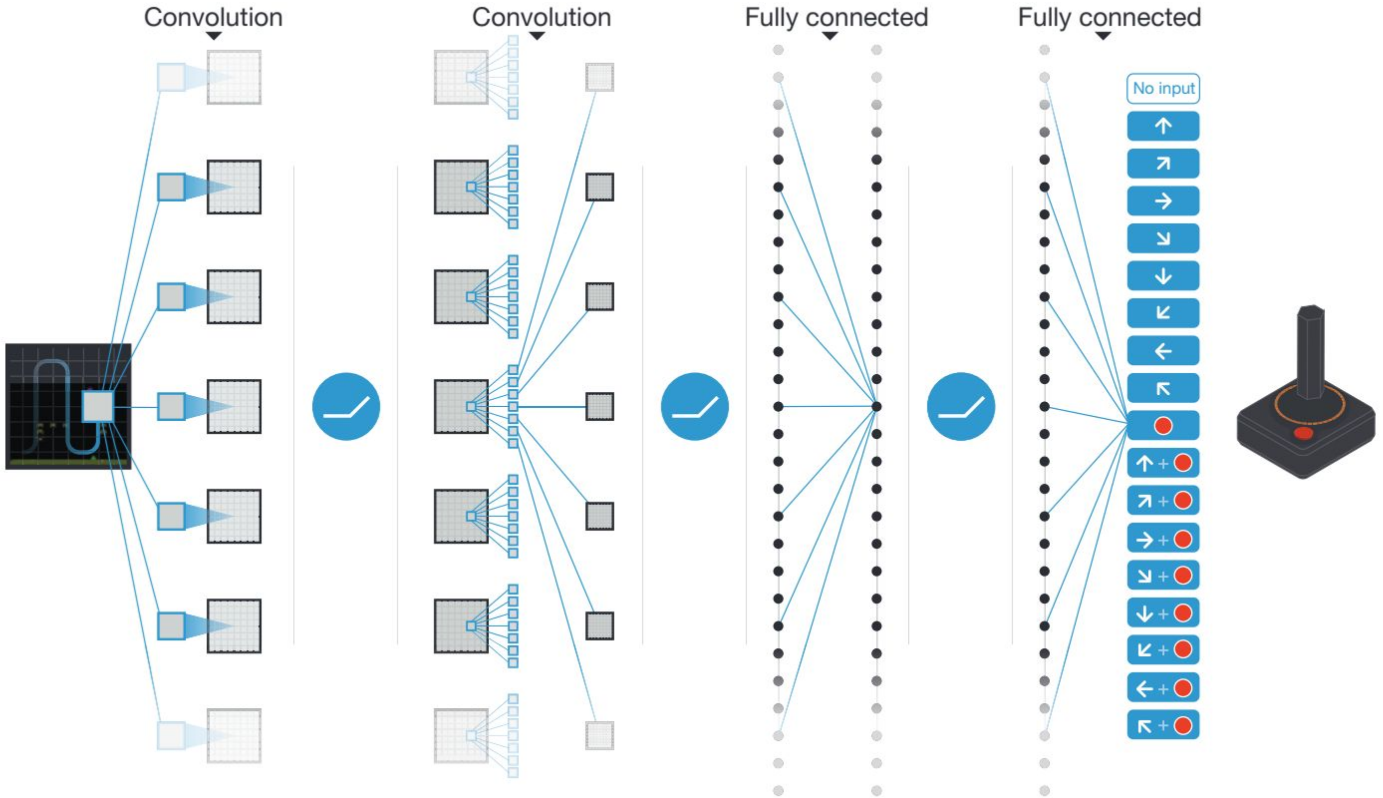
- Problem: Too many possible (state, action) pairs for standard Q-learning
- Instead, we can estimate Q(state, action)
    - Can use many kinds of approximator (simplest: linear)
    - Let's use a neural network

DQN applied to Atari

- End-to-end learning of Q(state, action) from pixels
- States are images (what a player sees)
- Actions are 18 possible joystick/button positions
- Reward is change in score for that step

- (Because high frame rate: only look at every *k* frames and repeat predicted action *k* times, e.g. $k = 4$.)

The original DQN architecture



What is the loss?

- Standard Q-learning:

$$Q(s_t, a_a) = Q(s_t, a_a) + \alpha(r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

- Deep Q-learning - minimize:

$$\left((r_t + \gamma \max_{a_{t+1}} Q_{\text{old}}(s_{t+1}, a_{t+1})) - Q_{\text{new}}(s_t, a_t)\right)^2$$

- Can optimize this using gradient descent!
- Update the parameters for Q_{new} (keeping Q_{old} fixed)

Problem 1. How to pick an action?

- RL algorithms give a way to learn how good different actions are
- But it's a separate question determining which actions to take during learning
- If you always pick what you think is best, then no exploration

Epsilon-greedy approach

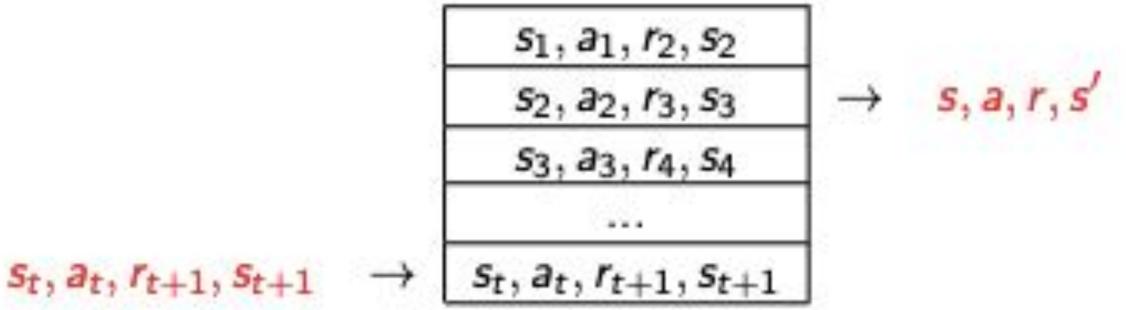
- Epsilon prob of random action
- Otherwise, take the action with highest Q value
- Surprisingly good
- (Though terrible in tasks where some actions are fatal)

Problem 2. Distribution shift

- Data coming in may change over time
- E.g. mistakes made early on
- How to stop forgetting earlier learning?

Experience replay

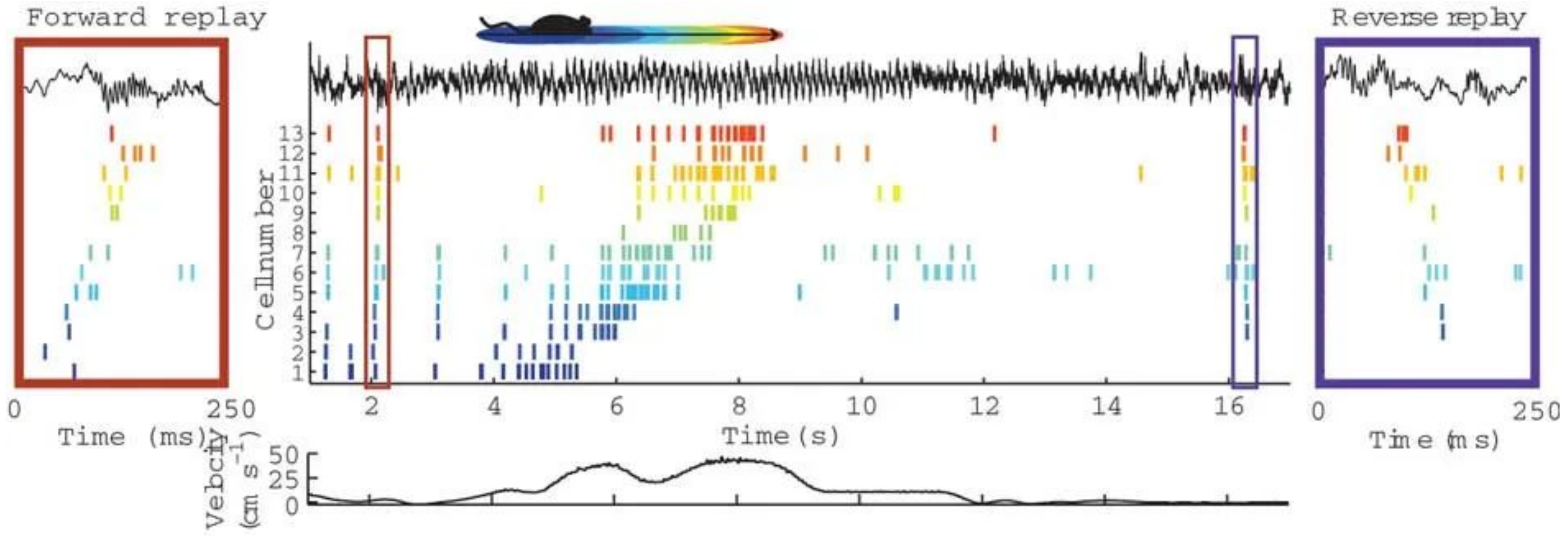
Replay past experiences from a memory buffer, learn as if they were new:



s_1, a_1, r_2, s_2
s_2, a_2, r_3, s_3
s_3, a_3, r_4, s_4
$\dots$
$s_t, a_t, r_{t+1}, s_{t+1}$
$s_t, a_t, r_{t+1}, s_{t+1} \rightarrow$
$\rightarrow s, a, r, s'$

Sample experiences from data-set and apply update

Experience replay



Problem 3. Instability

- What if a single update to Q changes the policy a lot?

- Solution: Don't use the same Q network both to learn and to choose the policy
- Target network and learner network
- Target network used to choose the policy
- Periodically, the learner network updates the target network
- But not every update

Algorithm 1: deep Q-learning with experience replay.

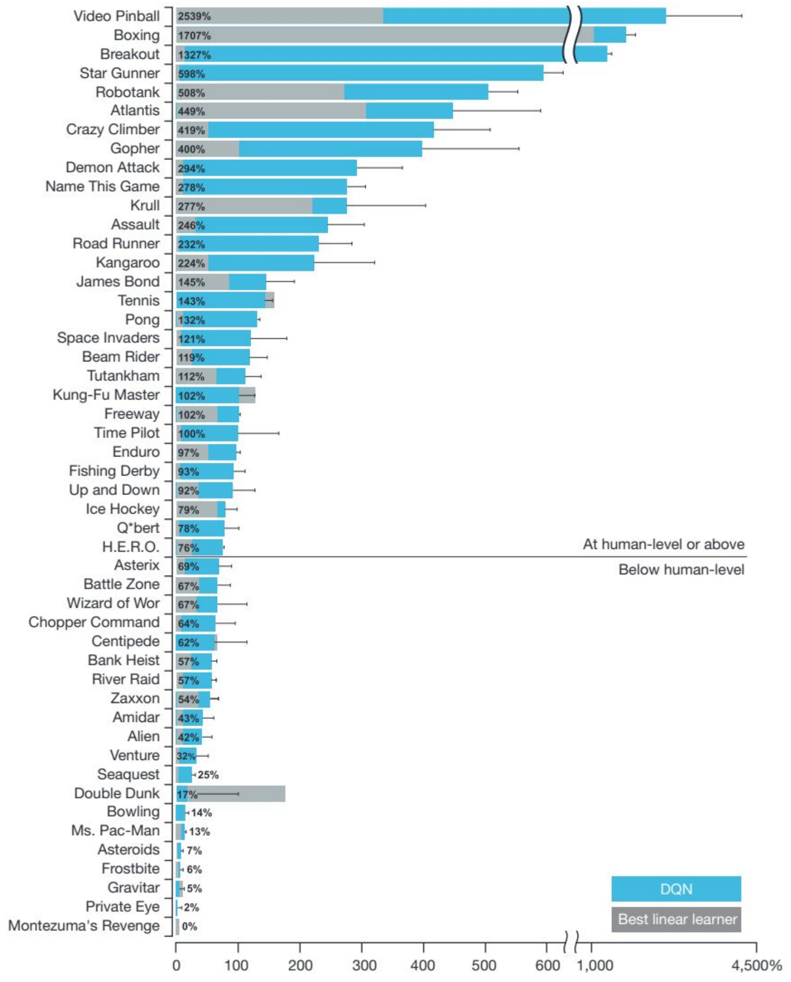
```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T do
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    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 $D$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
    network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  End For
End For
```

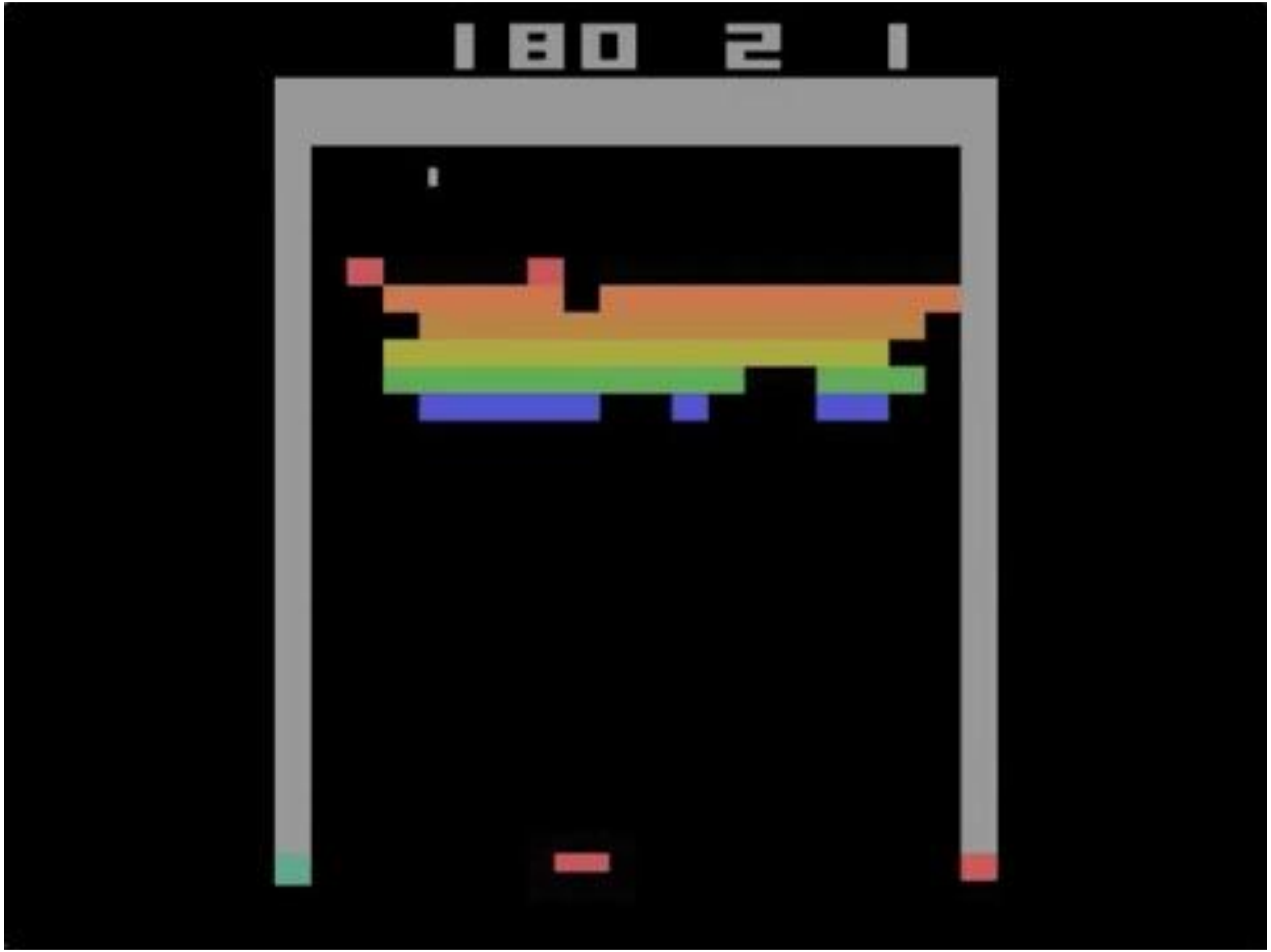
What was that preprocessing step?

- Preprocessing input in an Atari-specific way
- Some objects appear only on even frames, so take max of current image and previous frame's image
- Rescale the luminance channel
- And input is actually not just one image:
  - Stack of ~4 past images to give a little past info

DQN applied to Atari

- At or above human performance on most Atari games
- Fails completely on some, e.g. Montezuma's Revenge (sparse rewards, exploration)




180 2 1

DQN vs standard Q-learning

- DQN: estimate Q values with a neural net
- Traditional Q-learning: start with some random Q and update iteratively
    - Reaches optimum Q* if visit each (state, action) pair some large number of times

Why does DQN work even though not every (state,action) pair is visited even once?

A3C

Review of policy gradient methods

- Policy π(action | state) giving probabilities of actions given a state
- Basic policy gradient - move policy so that reward increases:

$$\text{update} = \sum_{t=1}^{T} r(\tau) \nabla_\theta \log \pi_\theta(a_t | s_t) \qquad \nabla \mathbb{E}_{\pi_\theta}[r(\tau)] = \mathbb{E}_{\pi_\theta}\left[r(\tau) \left(\sum_{t=1}^{T} \nabla \log \pi_\theta(a_t | s_t) \right) \right]$$

- REINFORCE algorithm:

$$\text{update} = \sum_{t=1}^{T} G(t) \nabla_\theta \log \pi_\theta(a_t | s_t) \qquad G_t = r_{t+1} + \gamma r_{t+2} + \ldots$$

- Actor-critic, reduces variance by using value function V(state_t) to estimate G(t)

$$\text{update} = \sum_{t=1}^{T} (G(t) - V(s_t)) \nabla_\theta \log \pi_\theta(a_t | s_t) \approx \sum_{t=1}^{T} (r_{t+1} + \gamma V_\pi(s_{t+1}) - V(s_t)) \nabla_\theta \log \pi_\theta(a_t | s_t)$$

Actor Critic

- Actor-critic, reduces variance by using value function V(state_t) to estimate G(t)

$$\text{update} = \sum_{t=1}^{T} (G(t) - V(s_t)) \nabla_\theta \log \pi_\theta(a_t|s_t) \approx \sum_{t=1}^{T} (r_{t+1} + \gamma V_\pi(s_{t+1}) - V(s_t)) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

- Need to train value function itself, use L2 loss

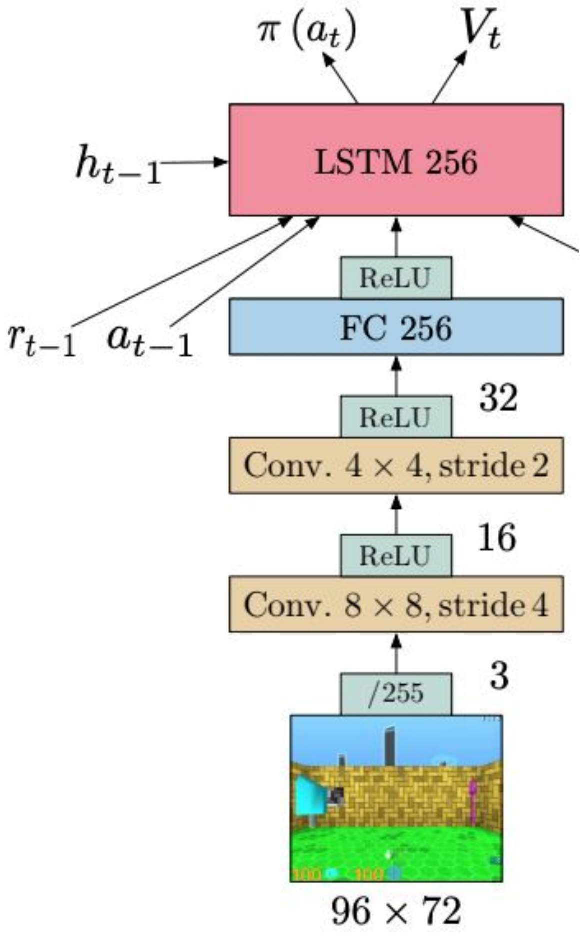
$$(r_{t+1} + \gamma V_\pi(s_{t+1}) - V_\pi(s_t))^2$$

- The "Critic" estimates the value function
- The "Actor" updates the policy distribution in the direction suggested by the Critic

- Deep actor critic methods - use neural networks to parameterize both V and π

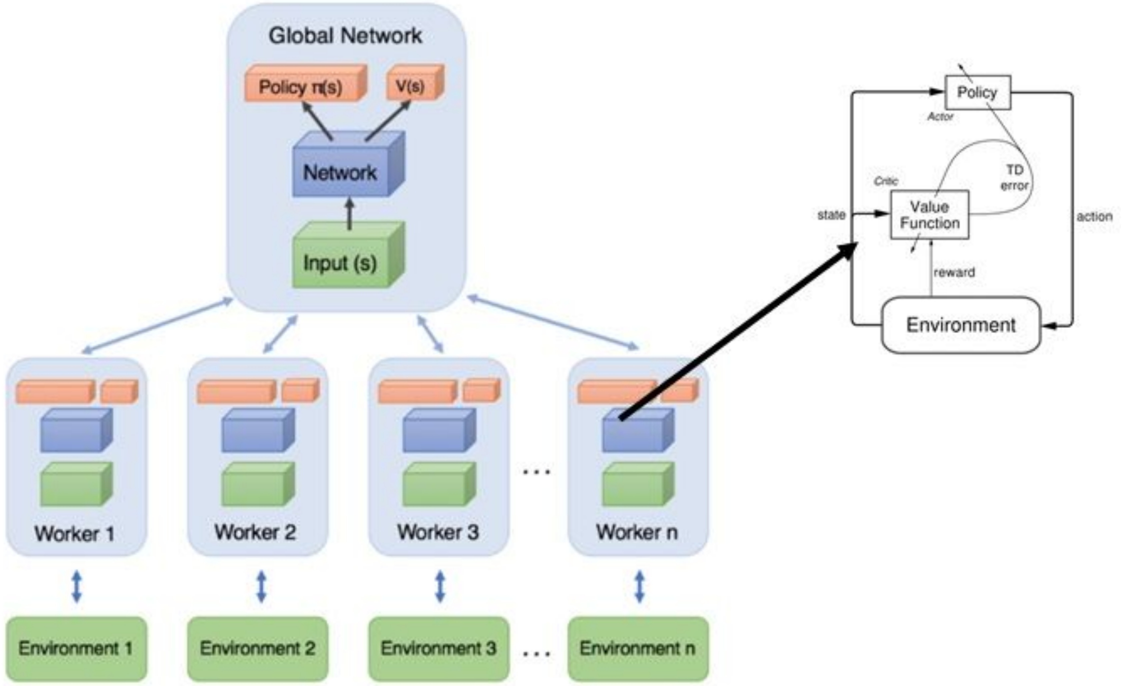
Example architecture

- Input is state (often image)
- ConvNet (sometimes ResNet) processes image
- Fed into LSTM
- Two separate "heads" for:
  - Policy (fully connected, softmax)
  - Value function (fully connected)
- "Body" (rest of architecture) shared



A3C : Asynchronous Advantage Actor Critic

- Central learner accumulates experience from many parallel workers
- Workers send (state, action, policy, value, reward)
- Learner performs a single gradient update
- Update broadcast to the workers



Why asynchronous?

- Parallel training - faster
- Separates gathering experience (in simulation, can be done on CPU, or from real-world interaction) from learning (GPU)
- Different workers means (slightly) different policies, so more exploration

Algorithm S2 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

```
// Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared counter $T = 0$
// Assume thread-specific parameter vectors $\theta'$ and $\theta'_v$
Initialize thread step counter $t \leftarrow 1$
repeat
    Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
    Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
    $t_{start} = t$
    Get state $s_t$
    repeat
        Perform $a_t$ according to policy $\pi(a_t|s_t;\theta')$
        Receive reward $r_t$ and new state $s_{t+1}$
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    until terminal $s_t$ or $t - t_{start} == t_{max}$
    $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
    for $i \in \{t-1, \ldots, t_{start}\}$ do
        $R \leftarrow r_i + \gamma R$
        Accumulate gradients wrt $\theta'$: $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
        Accumulate gradients wrt $\theta'_v$: $d\theta_v \leftarrow d\theta_v + \partial\left(R - V(s_i;\theta'_v)\right)^2 / \partial\theta'_v$
    end for
    Perform asynchronous update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$.
until $T > T_{max}$
```

How to ensure exploration?

- Could do epsilon-greedy approach again
- But now have access to a policy function
- Can make the policy itself do the exploring!
- Add term to the loss function:

$$-\epsilon H(\pi) = -\epsilon \sum_a \pi(a) \log \pi(a)$$

- Pushes towards a higher entropy policy - more exploration
- Less dangerous than just taking random actions

MuJoCo

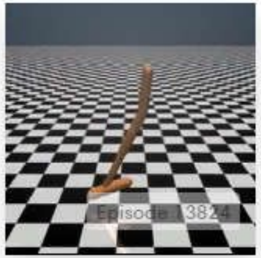
- Physics simulation engine often used to create RL tasks
- E.g. simple balancing and walking
- Used for e.g. OpenAI Gym (open source task library) =>

Episode 49000

HalfCheetah-v0
Make a 2D cheetah robot run.

Episode 47000

Swimmer-v0
Make a 2D robot swim.



Hopper-v0
Make a 2D robot hop.

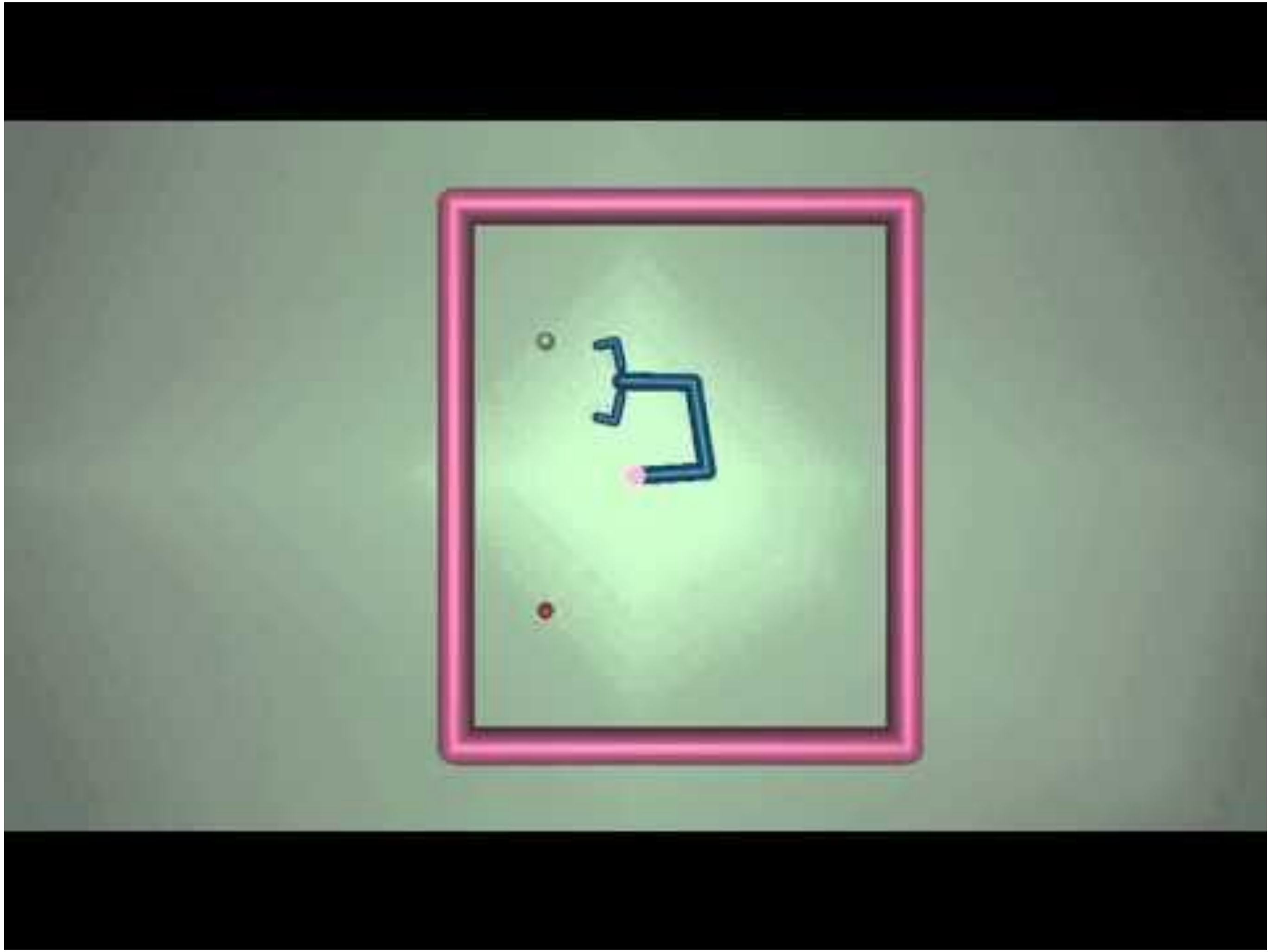
Episode 17576

Walker2d-v0
Make a 2D robot walk.

Episode 74088

Ant-v0
Make a 3D four-legged robot walk.

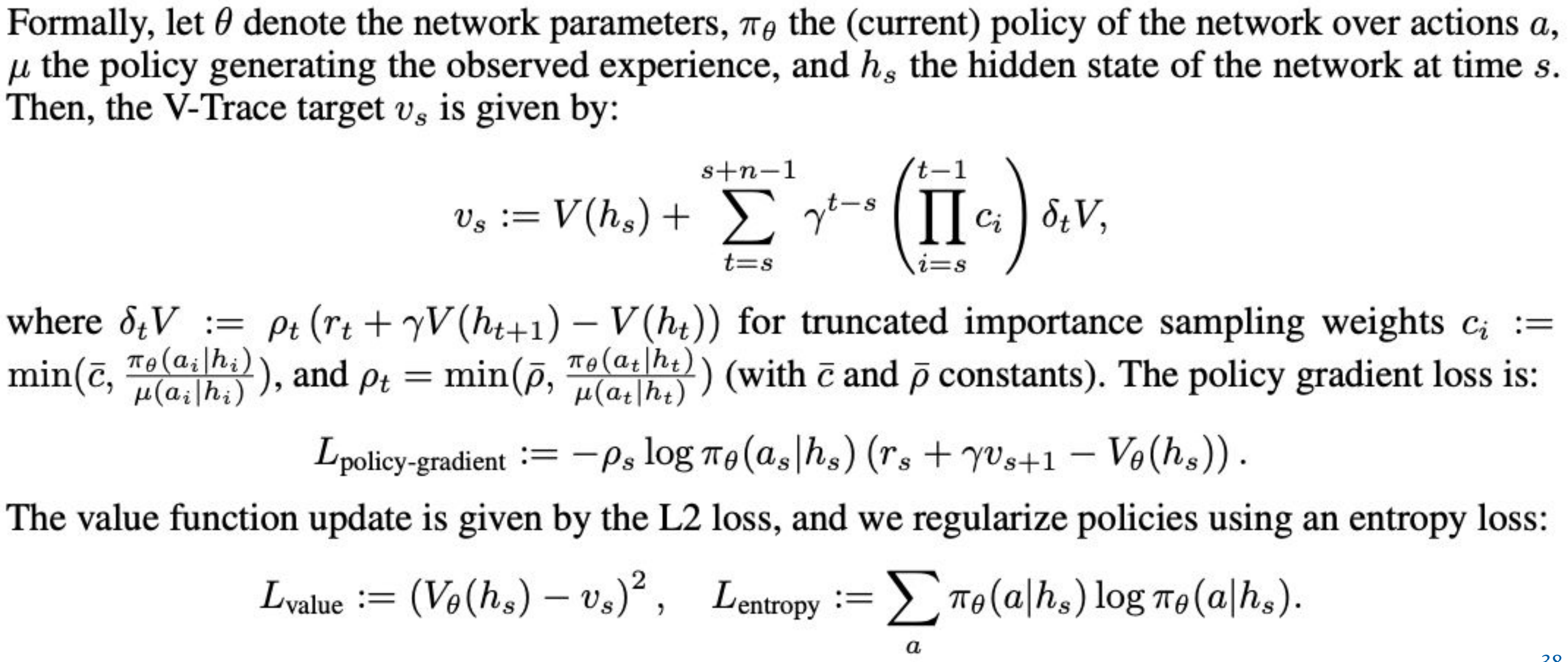
Humanoid-v0
Make a 3D two-legged robot walk.



IMPALA: Importance-Weighted Actor-Learner Architectures

- Designed to fix a problem in A3C where workers' policies can deviate from the policy of the learner
- What if the worker tries actions that the learner doesn't think are probable, and the worker gets a large gradient update?
  - The learner would update its parameters a *lot*
  - But it shouldn't, since it wouldn't take that action
- Idea: Re-weight observed actions according to ratio between worker and learner policies

IMPALA: Importance-Weighted Actor-Learner Architectures



Formally, let θ denote the network parameters, π_θ the (current) policy of the network over actions a, μ the policy generating the observed experience, and h_s the hidden state of the network at time s. Then, the V-Trace target v_s is given by:

$$v_s := V(h_s) + \sum_{t=s}^{s+n-1} \gamma^{t-s} \left(\prod_{i=s}^{t-1} c_i \right) \delta_t V,$$

where $\delta_t V := \rho_t \left(r_t + \gamma V(h_{t+1}) - V(h_t) \right)$ for truncated importance sampling weights $c_i := \min(\bar{c}, \frac{\pi_\theta(a_i|h_i)}{\mu(a_i|h_i)})$, and $\rho_t = \min(\bar{\rho}, \frac{\pi_\theta(a_t|h_t)}{\mu(a_t|h_t)})$ (with $\bar{c}$ and $\bar{\rho}$ constants). The policy gradient loss is:

$$L_{\text{policy-gradient}} := -\rho_s \log \pi_\theta(a_s|h_s) \left(r_s + \gamma v_{s+1} - V_\theta(h_s) \right).$$

The value function update is given by the L2 loss, and we regularize policies using an entropy loss:

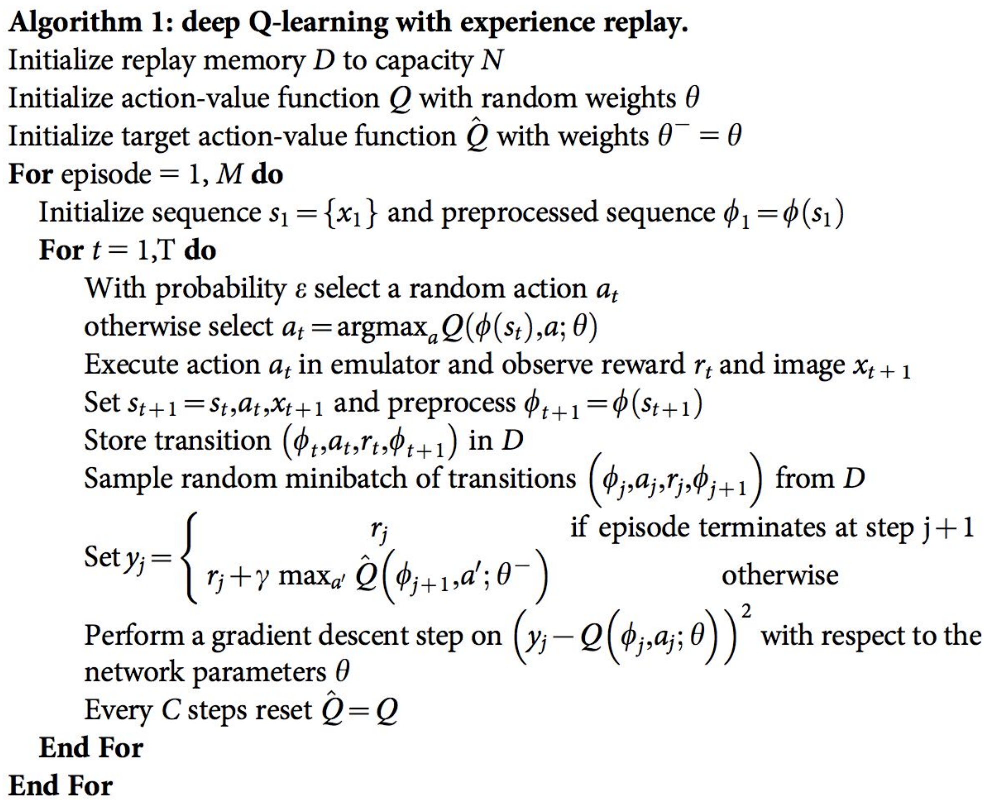
$$L_{\text{value}} := \left(V_\theta(h_s) - v_s \right)^2, \quad L_{\text{entropy}} := \sum_a \pi_\theta(a|h_s) \log \pi_\theta(a|h_s).$$

Brief interlude: DDPG

Continuous action spaces

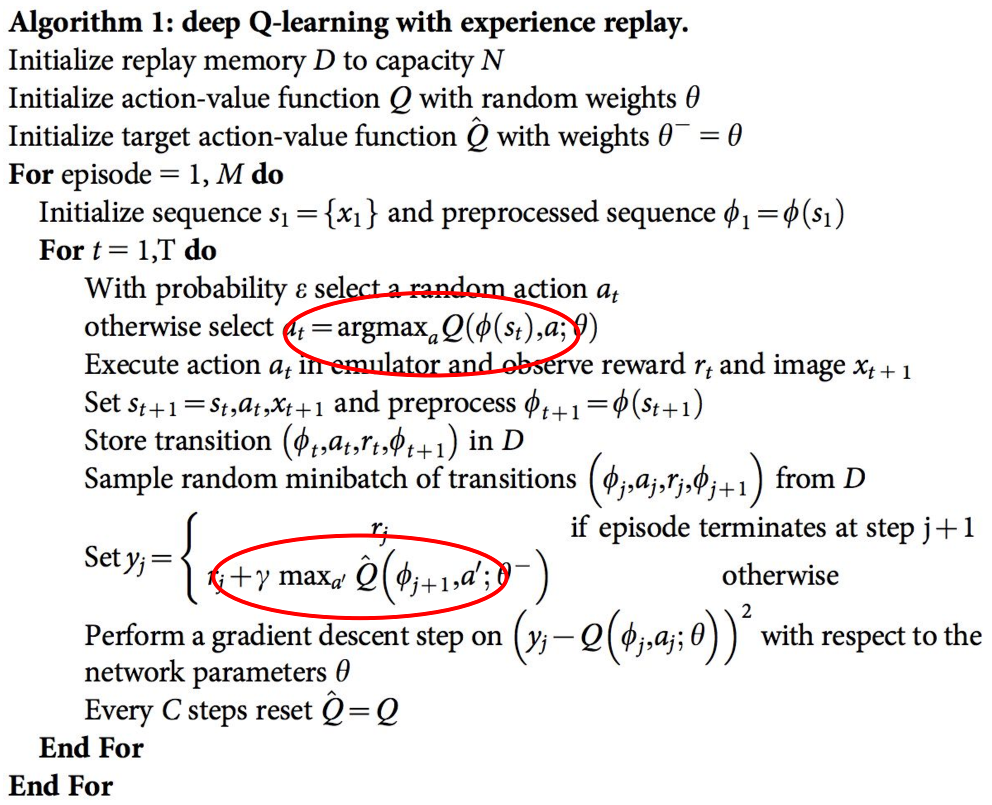
- All the algorithms we've talked about were for *discrete* actions
- We even used softmax vector output
- What about continuous actions, e.g. moving a robot?

Where DQN would break for continuous actions?



```
**Algorithm 1: deep Q-learning with experience replay.**
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
**For** episode $= 1, M$ **do**
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    **For** $t = 1$,T **do**
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
        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 $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the
        network parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    **End For**
**End For**
```

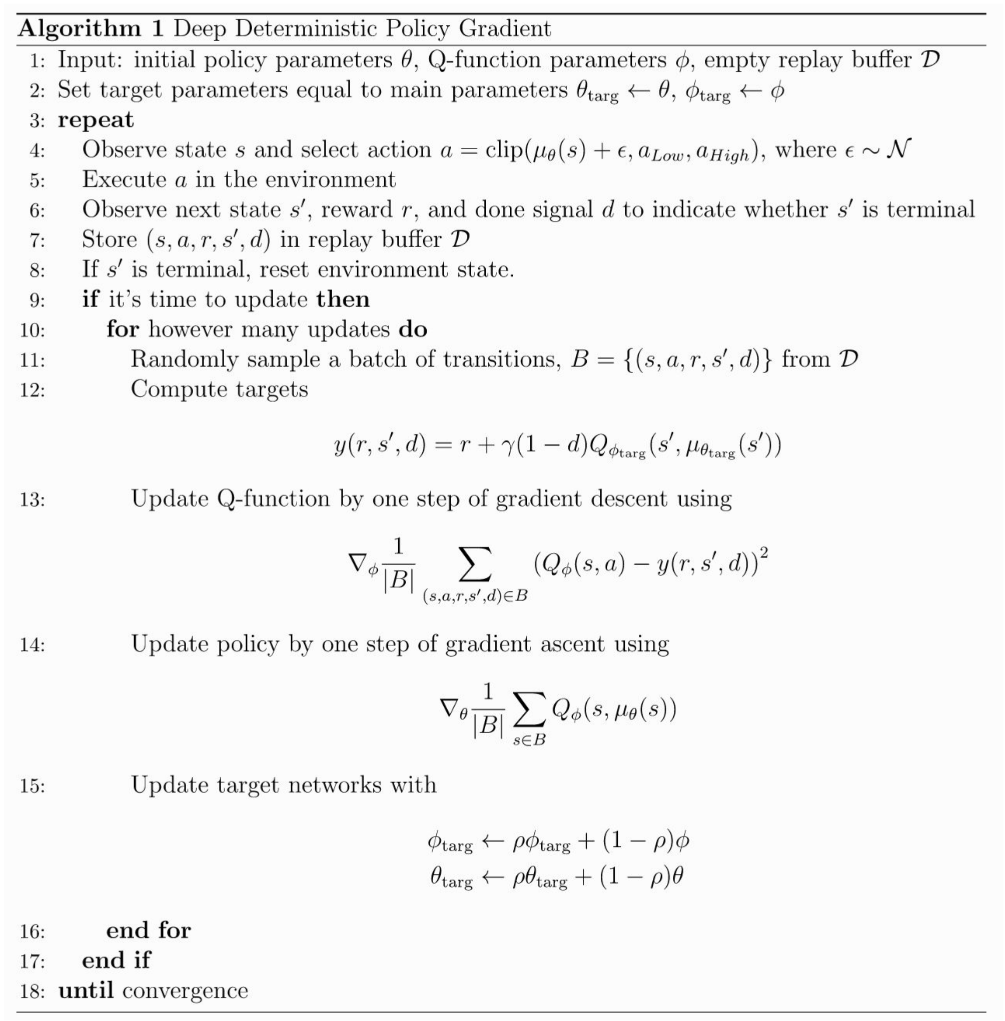
Where DQN would break for continuous actions?



```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode $= 1, M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T do
     With probability $\varepsilon$ select a random action $a_t$
     otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
     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 $D$
     Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
     Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
     Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
     network parameters $\theta$
     Every $C$ steps reset $\hat{Q} = Q$
  End For
End For
```

DDPG: Deep Deterministic Policy Gradient

- Q-network trained mostly as in DQN
- Policy network that approximates the best action
    - Train policy network by gradient ascent with objective given by Q-network



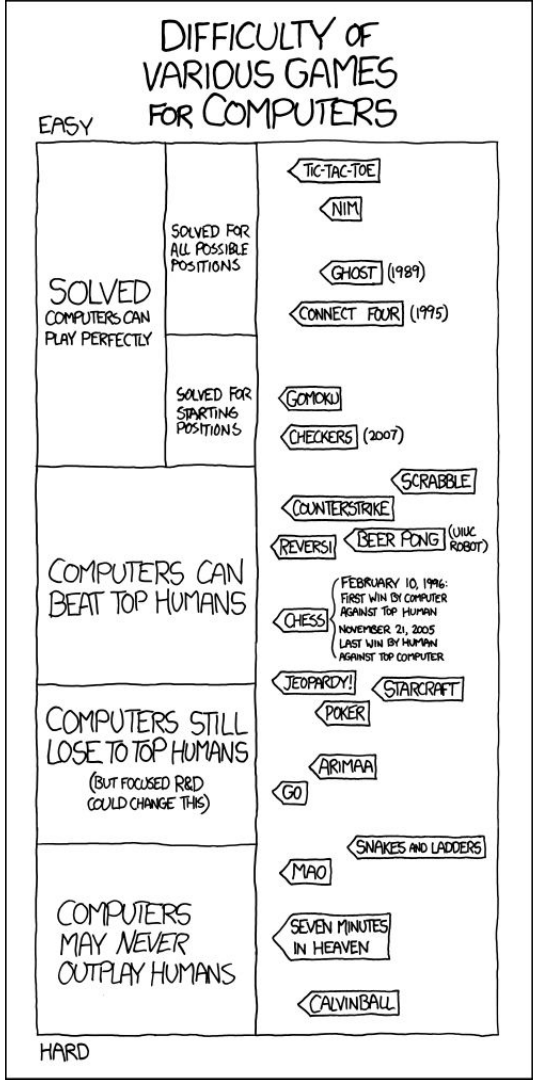
Algorithm 1 Deep Deterministic Policy Gradient

```
1: Input: initial policy parameters $\theta$, Q-function parameters $\phi$, empty replay buffer $\mathcal{D}$
2: Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
3: repeat
4:   Observe state $s$ and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
5:   Execute $a$ in the environment
6:   Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
7:   Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
8:   If $s'$ is terminal, reset environment state.
9:   if it's time to update then
10:    for however many updates do
11:      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12:      Compute targets
             $y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$
13:      Update Q-function by one step of gradient descent using
             $\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$
14:      Update policy by one step of gradient ascent using
             $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$
15:      Update target networks with
             $\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$
             $\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$
16:    end for
17:  end if
18: until convergence
```

AlphaGo

Hardness of games for AI

- Tic-Tac-Toe: evaluate all positions
- Chess: play out possible moves
- Go: ???



DIFFICULTY OF VARIOUS GAMES FOR COMPUTERS
EASY
HARD
SOLVED COMPUTERS CAN PLAY PERFECTLY
SOLVED FOR ALL POSSIBLE POSITIONS
SOLVED FOR STARTING POSITIONS
COMPUTERS CAN BEAT TOP HUMANS
COMPUTERS STILL LOSE TO TOP HUMANS (BUT FOCUSED R&D COULD CHANGE THIS)
COMPUTERS MAY NEVER OUTPLAY HUMANS
TIC-TAC-TOE
NIM
GHOST (1989)
CONNECT FOUR (1995)
GOMOKU
CHECKERS (2007)
SCRABBLE
COUNTERSTRIKE
REVERSI
BEER PONG (UIUC ROBOT)
CHESS
FEBRUARY 10, 1996: FIRST WIN BY COMPUTER AGAINST TOP HUMAN
NOVEMBER 21, 2005 LAST WIN BY HUMAN AGAINST TOP COMPUTER
JEOPARDY!
STARCRAFT
POKER
ARIMAA
GO
SNAKES AND LADDERS
MAO
SEVEN MINUTES IN HEAVEN
CALVINBALL

xkcd from 2012

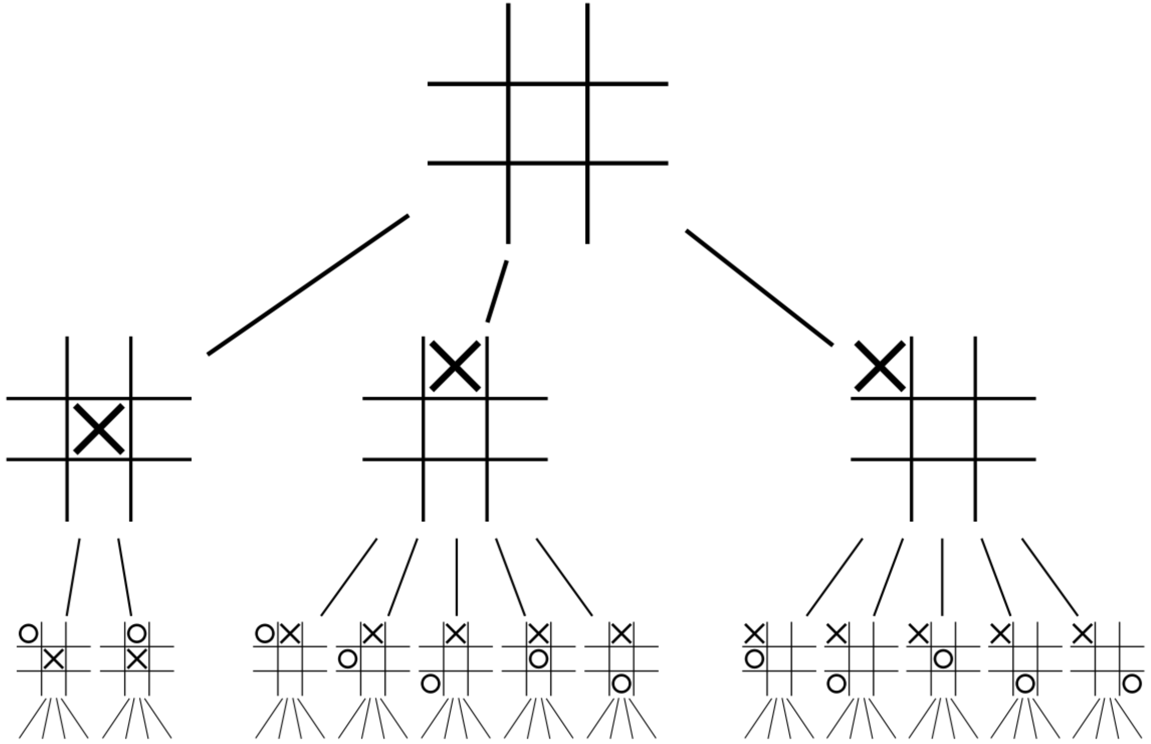
Why is Go hard?

- Chess: ~35 possible moves, ~80 moves per game
- Go: ~250 possible moves, ~150 moves per game
- Too many for playing out possible games (~10^{172} possible positions)
- High intuition, widely seen as an art form

Why not just explore the game tree?

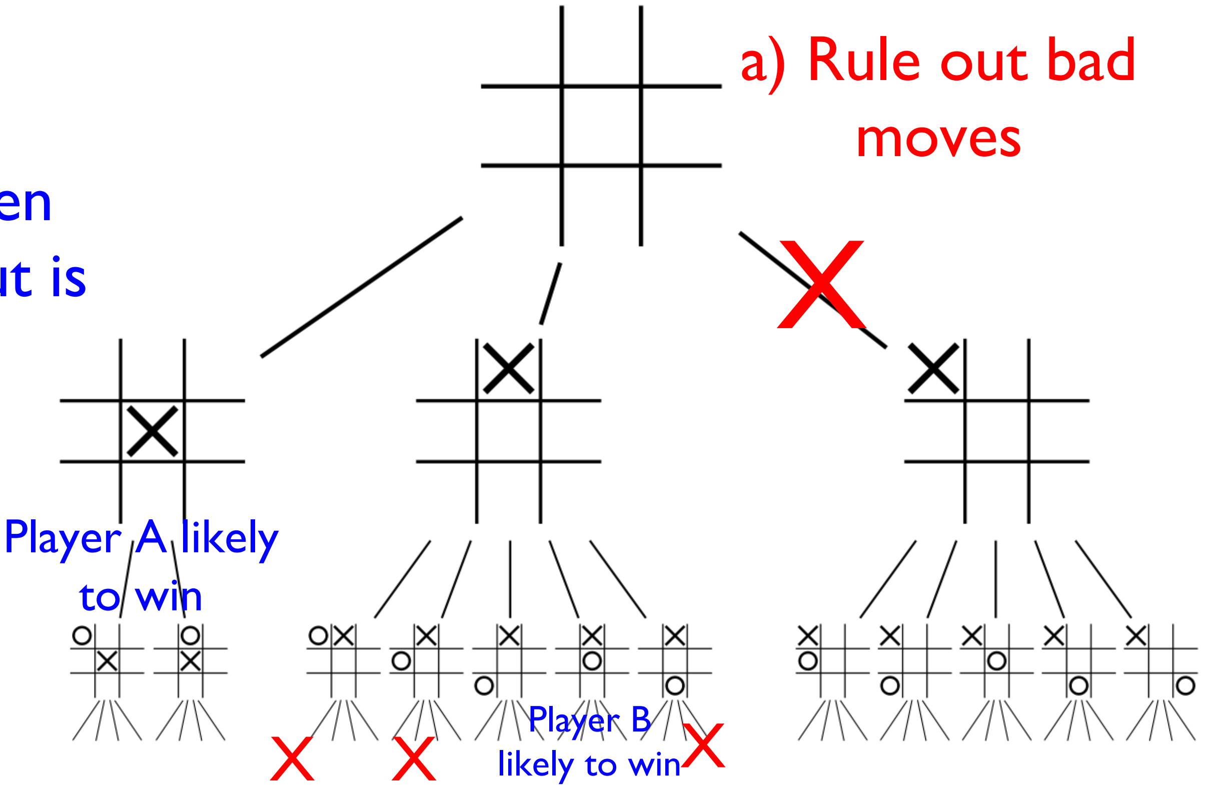
- Many moves ⇔ each node has many branches

- Long games ⇔ have to play out many moves to determine a winner



Trimming the game tree

b) Evaluate when no more play-out is necessary

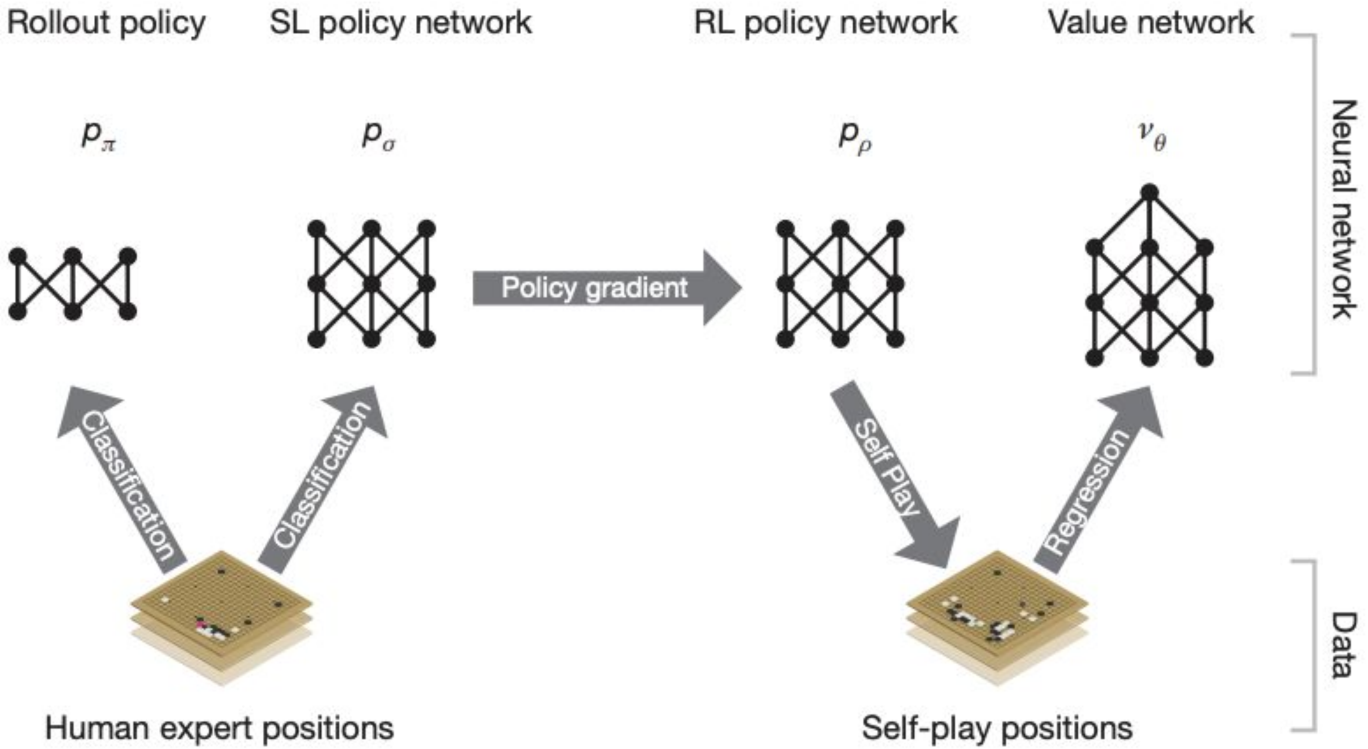


AlphaGo

nature
THE INTERNATIONAL WEEKLY JOURNAL OF SCIENCE
At last — a computer program that can beat a champion Go player PAGE 484
ALL SYSTEMS GO
CONSERVATION
SONGBIRDS À LA CARTE
Illegal harvest of millions of Mediterranean birds
PAGE 452
RESEARCH ETHICS
SAFEGUARD TRANSPARENCY
Don't let openness backfire on individuals
PAGE 459
POPULAR SCIENCE
WHEN GENES GOT 'SELFISH'
Dawkins's calling card 40 years on
PAGE 462
NATUREASIA.COM
28 January 2016
Vol. 529, No. 7587

ALPHAGO 01:59:50
LEE SEDOL 01:59:44
AlphaGo
Lee Sedol

Training Algorithm



Training algorithm

1. Acquire large amounts of data from human gameplay
2. Train a supervised learning (SL) model to predict human moves based on the position of the board
3. Use final parameters from SL model to initialize an RL model
4. Have the RL model train and improve via policy gradient by playing many simulated games against itself
5. Use the data set of self-play games to train a value network that predicts who will win a game based on the current board position

Training details for RL models

Policy network

- Reward only at terminal state: +1 if wins, -1 if loses
- In self-play, plays against past version of self (from random time)
- Trained by policy gradient:

$$\Delta\rho \propto \frac{\partial \log p_\rho(a_t|s_t)}{\partial \rho} z_t$$

Value network

- *Completely separate* from policy network - and trained after it
- Unlike in actor-critic framework we've seen
- Trained by regression to minimize L2 loss between value and outcome
    - Looks a bit like supervised learning :)

$$\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial \theta}(z - v_\theta(s))$$

Gameplay algorithm

- When playing, uses Monte Carlo tree search to expand possible paths of play for many moves and determine which one will be best

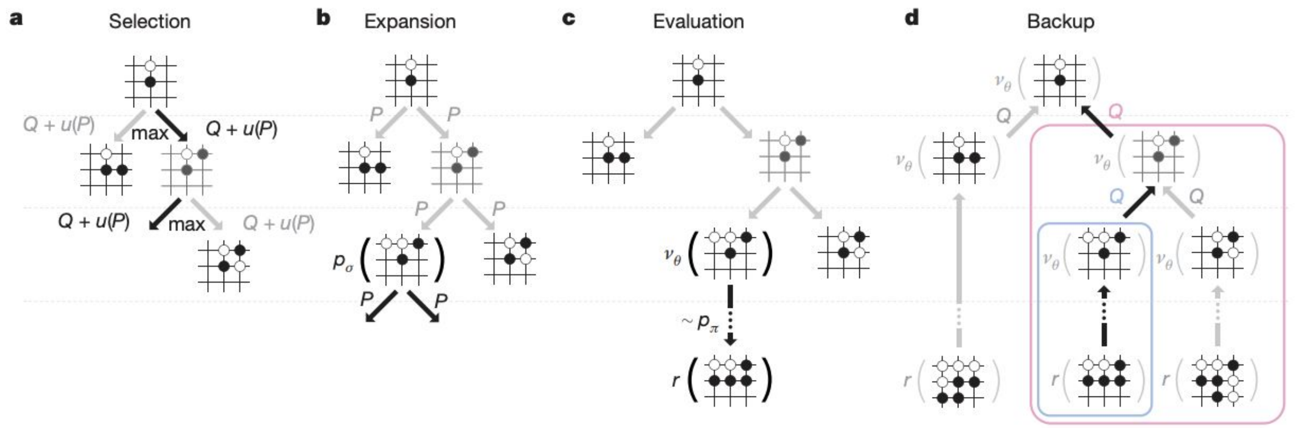


Figure 3 | Monte Carlo tree search in AlphaGo. a, Each simulation traverses the tree by selecting the edge with maximum action value Q, plus a bonus $u(P)$ that depends on a stored prior probability P for that edge. **b**, The leaf node may be expanded; the new node is processed once by the policy network p_σ and the output probabilities are stored as prior probabilities P for each action. **c**, At the end of a simulation, the leaf node is evaluated in two ways: using the value network v_θ; and by running a rollout to the end of the game with the fast rollout policy p_π, then computing the winner with function r. **d**, Action values Q are updated to track the mean value of all evaluations $r(\cdot)$ and $v_\theta(\cdot)$ in the subtree below that action.

Gameplay algorithm

- In exploring possible games, actions are taken according to

$$a_t = \underset{a}{\operatorname{argmax}}(Q(s_t, a) + u(s_t, a))$$

- Two components - one is exploration:

$$u(s,a) \propto \frac{P(s,a)}{1+N(s,a)} \qquad N(s,a) = \sum_{i=1}^{n} 1(s,a,i)$$

- Other is estimated Q-value:

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{n} 1(s,a,i) V(s_L^i)$$

- The V-value is actually a combination of value network and fast rollout policy:

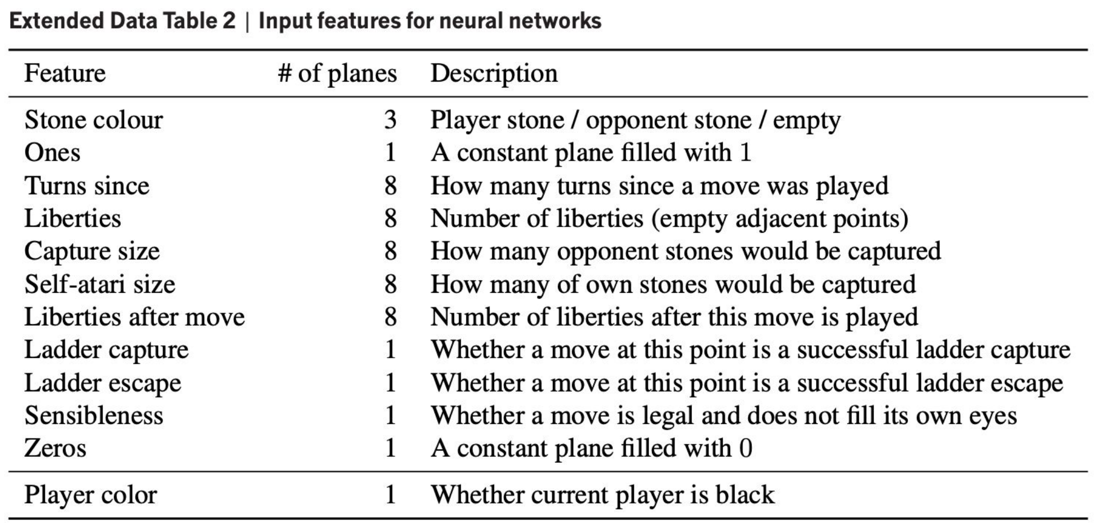
$$V(s_L) = (1-\lambda) v_\theta(s_L) + \lambda z_L$$

Network architecture

- ConvNet that takes board as input image
- 13 conv layers, then a fully connected layer
- Softmax for policy network
- Scalar output for value network

Details

- Symmetries of board - 8 rotations/reflections
- Take average of output of network over all of these
- Additional features as input:



Extended Data Table 2 | Input features for neural networks

Feature	# of planes	Description
Stone colour	3	Player stone / opponent stone / empty
Ones	1	A constant plane filled with 1
Turns since	8	How many turns since a move was played
Liberties	8	Number of liberties (empty adjacent points)
Capture size	8	How many opponent stones would be captured
Self-atari size	8	How many of own stones would be captured
Liberties after move	8	Number of liberties after this move is played
Ladder capture	1	Whether a move at this point is a successful ladder capture
Ladder escape	1	Whether a move at this point is a successful ladder escape
Sensibleness	1	Whether a move is legal and does not fill its own eyes
Zeros	1	A constant plane filled with 0
Player color	1	Whether current player is black

Resources

SL model:

- 30M human moves
- 50 GPUs for 3 weeks

RL model:

- Policy network: 50 GPUs for 1 day, 1M games of self-play
- Value network: 50 GPUs for 1 week, 30M games of self-play
    - Just one position from each game used (otherwise ended up overfitting)

Match Versus Lee Sedol

- March 2016
- Lee Sedol ranked 2nd in world
- AlphaGo won 4 games to 1
- Move 37

- Huge event for much of the world
- And in perception of deep RL

AlphaZero

- General-purpose - plays Go, chess, shogi, etc.
- Trained using only self-play
- Simpler algorithm
- Beats AlphaGo
- Beats Chess algorithms like Stockfish

AlphaZero

- No SL network, just RL
- Single shared network outputs both policy and value (different heads)
- Trained just on self-play
- Monte Carlo tree search rolled into the *training*, not the game play:
    - Policy trained to replicate probabilities from tree search
    - Value trained to replicate ultimate game outcome

$$l = (z - v)^2 - \boldsymbol{\pi}^\top \log \mathbf{p} + c||\theta||^2,$$

- No special features or symmetries, just the board

Summary

- DQN
  - Approximate Q-values with deep net
  - Trained to satisfy Bellman equations
- A3C
  - Policy and value functions with shared deep net
  - Actor-critic training of both simultaneously
- AlphaGo
  - Policy network trained by policy gradient
  - Value network trained on game outcomes
  - Monte Carlo tree search
  - AlphaZero rolls all this into one network

Model-free and model-based methods

- DQN, A3C, etc. are all examples of model-free RL
  - No knowledge/learning of explicitly how the task works
- Deep learning often works this way
- But can be limiting when we do know stuff about the world
- AlphaGo is a model-based method that incorporates knowledge of how the task works
  - Simulates possible games
  - Uses deep learning within that framework

What could have been better about this lecture?