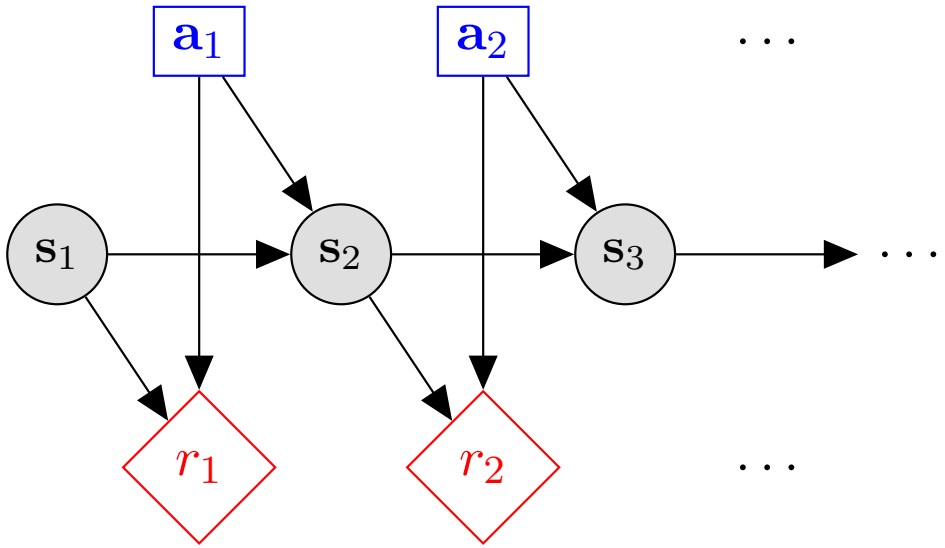
Machine Learning (CS 181): 22. (Deep) Reinforcement Learning

David C. Parkes and Sasha Rush

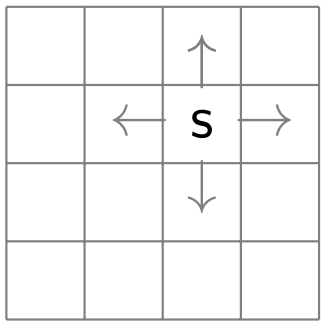
Spring 2017

Markov Decision Process



S	States
A	Actions
$r : S \times A \mapsto \mathbb{R}$	Reward Function
$p(s' \mid s, a)$	Transition Model

Running Illustration: MDP on Gridworld



S	Location of the grid (x_1, x_2)
A	Local movements $\leftarrow, \rightarrow, \uparrow, \downarrow$
$r : S \times A \mapsto \mathbb{R}$	Reward function, e.g. make it to goal
$p(s' \mid s, a)$	Transition model, e.g deterministic or slippages

Contents

Review: Bellman equations

The planning problem for an MDP is:

$$\pi^* \in \arg\max_{\pi} V^{\pi}(s).$$

(exists a solution that is optimal for every state s).

Definition (Bellman equations)

For an optimal policy π^*, we have

$$V^*(s) = \max_{a \in A} \left[r(s,a) + \gamma \sum_{s' \in S} p(s' \mid s, a) V^*(s') \right], \quad \forall s \tag{1}$$

Alternate Form: Bellman equations

Alternate form of the Bellman operator using the Q-Function using:

$$\pi^* \in \arg\max_{\pi} Q^{\pi}(s, a).$$

Definition (Bellman equations)

For an optimal policy π^*, we have

$$Q^*(s, a) = r(s, a) + \gamma \sum_{s' \in S} p(s' \mid s, a) \max_{a' \in A} \left[Q^*(s', a') \right], \quad \forall s, a \tag{2}$$

Model-Free Estimation Strategy

Observe:

- $Q^*(s, a)$ is just a function from $S \times A \mapsto \mathbb{R}$
- If we had Q^* then $\pi^*(s) = \arg\max_a Q^*(s, a)$

Strategy:

- Learn the value of a Q-function to learn Q^*
- Use a parameter table, $\mathbf{w} \in \mathbb{R}^{|S||A|}$:

$$Q(s, a; \mathbf{w}) = w_{s,a}$$

Model-Free Estimation Strategy

Observe:

- $Q^*(s, a)$ is just a function from $S \times A \mapsto \mathbb{R}$
- If we had Q^* then $\pi^*(s) = \arg\max_a Q^*(s, a)$

Strategy:

- Learn the value of a Q-function to learn Q^*
- Use a parameter table, $\mathbf{w} \in \mathbb{R}^{|S||A|}$:

$$Q(s, a; \mathbf{w}) = w_{s,a}$$

Learning Objective

But how do we learn the Q-function without supervision.

- Use **Bellman** equations.
- We know that upon convergence, we should have π.

$$Q(s, a; \mathbf{w}) = r(s, a) + \gamma \sum_{s'} p(s' \mid s, a) \max_{a'} Q(s', a'; \mathbf{w})$$

- Enforce this directly as a least squares **loss function**,

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2} \mathbb{E}_{s,a} (Q(s, a; \mathbf{w}) - [r(s, a) + \gamma \sum_{s'} p(s' \mid s, a) \max_{a'} Q(s', a'; \mathbf{w})])^2$$

- "Find a Q for which the Bellman equation holds."

Learning Objective

But how do we learn the Q-function without supervision.

- Use **Bellman** equations.
- We know that upon convergence, we should have π.

$$Q(s, a; \mathbf{w}) = r(s, a) + \gamma \sum_{s'} p(s' \mid s, a) \max_{a'} Q(s', a'; \mathbf{w})$$

- Enforce this directly as a least squares **loss function**,

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\mathbb{E}_{s,a}(Q(s, a; \mathbf{w}) - [r(s, a) + \gamma \sum_{s'} p(s' \mid s, a) \max_{a'} Q(s', a'; \mathbf{w})])^2$$

- "Find a Q for which the Bellman equation holds."

Learning Objective

But how do we learn the Q-function without supervision.

- Use **Bellman** equations.
- We know that upon convergence, we should have π.

$$Q(s, a; \mathbf{w}) = r(s, a) + \gamma \sum_{s'} p(s' \mid s, a) \max_{a'} Q(s', a'; \mathbf{w})$$

- Enforce this directly as a least squares **loss function**,

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\mathbb{E}_{s,a}(Q(s, a; \mathbf{w}) - [r(s, a) + \gamma \sum_{s'} p(s' \mid s, a) \max_{a'} Q(s', a'; \mathbf{w})])^2$$

- "Find a Q for which the Bellman equation holds."

Sketch of SGD Approach

Loss at a sampled state-action pair s, a

$$\mathcal{L}^{(s,a)}(\mathbf{w}) = \frac{1}{2}(Q(s,a;\mathbf{w}) - [r(s,a) + \gamma \sum_{s'} p(s' \mid s,a) \max_{a'} Q(s',a';\mathbf{w})])^2$$

Take the gradient of the sampled loss (assuming $s' \neq s$),

$$\frac{\partial \mathcal{L}^{(s,a)}(\mathbf{w})}{\partial w_{s,a}} = Q(s,a;\mathbf{w}) - [r(s,a) + \gamma \sum_{s'} p(s' \mid s,a) \max_{a'} Q(s',a';\mathbf{w})]$$

Approximate this by further sampling the next state to approximate the expectation $s' \sim p(s' \mid s,a)$,

$$\frac{\partial \mathcal{L}^{(s,a)}(\mathbf{w})}{\partial w_{s,a}} \approx Q(s,a;\mathbf{w}) - [r(s,a) + \gamma \max_{a'} Q(s',a';\mathbf{w})]$$

Sketch of SGD Approach

Loss at a sampled state-action pair s, a

$$\mathcal{L}^{(s,a)}(\mathbf{w}) = \frac{1}{2}(Q(s,a;\mathbf{w}) - [r(s,a) + \gamma \sum_{s'} p(s' \mid s,a) \max_{a'} Q(s',a';\mathbf{w})])^2$$

Take the gradient of the sampled loss (assuming $s' \neq s$),

$$\frac{\partial \mathcal{L}^{(s,a)}(\mathbf{w})}{\partial w_{s,a}} = Q(s,a;\mathbf{w}) - [r(s,a) + \gamma \sum_{s'} p(s' \mid s,a) \max_{a'} Q(s',a';\mathbf{w})]$$

Approximate this by further sampling the next state to approximate the expectation $s' \sim p(s' \mid s,a)$,

$$\frac{\partial \mathcal{L}^{(s,a)}(\mathbf{w})}{\partial w_{s,a}} \approx Q(s,a;\mathbf{w}) - [r(s,a) + \gamma \max_{a'} Q(s',a';\mathbf{w})]$$

Sketch of SGD Approach

Loss at a sampled state-action pair s, a

$$\mathcal{L}^{(s,a)}(\mathbf{w}) = \frac{1}{2}(Q(s,a;\mathbf{w}) - [r(s,a) + \gamma \sum_{s'} p(s' \mid s,a) \max_{a'} Q(s',a';\mathbf{w})])^2$$

Take the gradient of the sampled loss (assuming $s' \neq s$),

$$\frac{\partial \mathcal{L}^{(s,a)}(\mathbf{w})}{\partial w_{s,a}} = Q(s,a;\mathbf{w}) - [r(s,a) + \gamma \sum_{s'} p(s' \mid s,a) \max_{a'} Q(s',a';\mathbf{w})]$$

Approximate this by further sampling the next state to approximate the expectation $s' \sim p(s' \mid s,a)$,

$$\frac{\partial \mathcal{L}^{(s,a)}(\mathbf{w})}{\partial w_{s,a}} \approx Q(s,a;\mathbf{w}) - [r(s,a) + \gamma \max_{a'} Q(s',a';\mathbf{w})]$$

Contents

Temporal Difference Meta-Algorithm

General update will be to use update (for $s' \sim p(s' \mid s, a)$)

$$\frac{\partial \mathcal{L}^{(s,a)}(\mathbf{w})}{\partial w_{s,a}} \approx Q(s, a; \mathbf{w}) - [r(s, a) + \gamma \max_{a'} Q(s', a'; \mathbf{w})]$$

Issues:

- Need to decide how to obtain s, a
- Want to act in a real-world (possibly non-reversible).

General approach:

- Start at state $s \in S$, select action $a \in A$, collect r and s' from world, consider new action a', update with gradients, repeat at s'.
- How we pick actions a and a' will give us different algorithms.

Temporal Difference Meta-Algorithm

General update will be to use update (for $s' \sim p(s' \mid s, a)$)

$$\frac{\partial \mathcal{L}^{(s,a)}(\mathbf{w})}{\partial w_{s,a}} \approx Q(s, a; \mathbf{w}) - [r(s, a) + \gamma \max_{a'} Q(s', a'; \mathbf{w})]$$

Issues:

- Need to decide how to obtain s, a
- Want to act in a real-world (possibly non-reversible).

General approach:

- Start at state $s \in S$, select action $a \in A$, collect r and s' from world, consider new action a', update with gradients, repeat at s'.
- How we pick actions a and a' will give us different algorithms.

TD Algorithmic Framework

procedure TD-RL(s)

Select action a based on $Q(s, a)$

Collect $r \leftarrow r(s, a)$ from environment

Sample $s' \sim p(s'|s, a)$ from environment

Consider possible future actions a'*

Update estimate of $Q(s, a)$ function through $w_{s,a}$

return s'

One step of RL on Gridworld

procedure TD-RL(s)

Select action a based on $Q(s, a)$

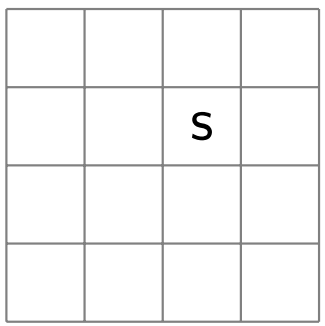
Collect $r \leftarrow r(s, a)$ from environment

Sample $s' \sim p(s'|s, a)$ from environment

Consider possible future actions a'

Update estimate of $Q(s, a)$ function through $w_{s,a}$

return s'



One step of RL on Gridworld

procedure TD-RL(s)

Select action a based on $Q(s, a)$

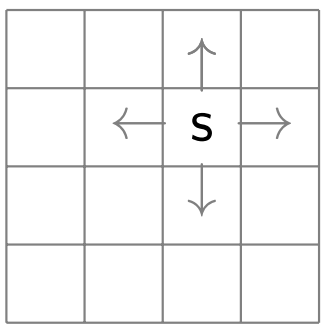
Collect $r \leftarrow r(s, a)$ from environment

Sample $s' \sim p(s'|s, a)$ from environment

Consider possible future actions a'

Update estimate of $Q(s, a)$ function through $w_{s,a}$

return s'



One step of RL on Gridworld

procedure TD-RL(s)

Select action a based on $Q(s, a)$

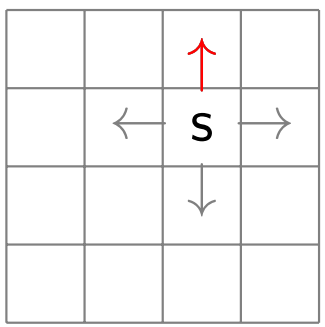
Collect $r \leftarrow r(s, a)$ from environment

Sample $s' \sim p(s'|s, a)$ from environment

Consider possible future actions a'

Update estimate of $Q(s, a)$ function through $w_{s,a}$

return s'



One step of RL on Gridworld

procedure TD-RL(s)

Select action a based on $Q(s, a)$

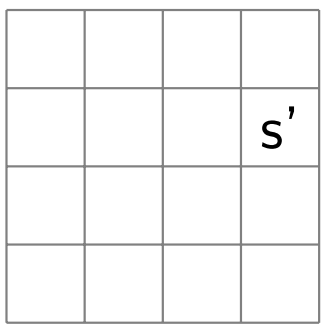
Collect $r \leftarrow r(s, a)$ from environment

Sample $s' \sim p(s'|s, a)$ from environment

Consider possible future actions a'

Update estimate of $Q(s, a)$ function through $w_{s,a}$

return s'



One step of RL on Gridworld

procedure TD-RL(s)

Select action a based on $Q(s, a)$

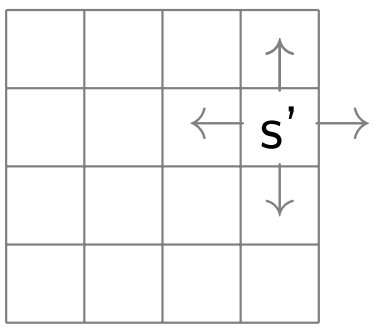
Collect $r \leftarrow r(s, a)$ from environment

Sample $s' \sim p(s'|s, a)$ from environment

Consider possible future actions a'

Update estimate of $Q(s, a)$ function through $w_{s,a}$

return s'



TD Choices

procedure TD-RL(s)

Select action a based on $Q(s, a)$ (1)

Collect $r \leftarrow r(s, a)$ from environment

Sample $s' \sim p(s'|s, a)$ from environment

Consider possible future actions a' (2)

Update estimate of $Q(s, a)$ function through $w_{s,a}$

return s'

1. How should we select with which action a to take to move to the next step?
2. How should we select which action a' for update?

(1) Agent Policies

Need to pick a policy for taking action a.

If we are **exploitative**, it is clear which action a to take. Just take highest expected reward.

$$\pi(s) \leftarrow \arg\max_a Q(s, a; \mathbf{w})$$

(Why might this be bad?)

For **exploration**, an alternative is an epsilon greedy approach.

$$\pi(s) \leftarrow \begin{cases} \arg\max_a Q(s, a; \mathbf{w}) & \text{with prob } 1 - \epsilon \\ \text{random } a \in A & \text{with prob } \epsilon \end{cases}$$

Many other approaches for exploration.

(1) Agent Policies

Need to pick a policy for taking action a.

If we are **exploitative**, it is clear which action a to take. Just take highest expected reward.

$$\pi(s) \leftarrow \arg\max_a Q(s, a; \mathbf{w})$$

(Why might this be bad?)

For **exploration**, an alternative is an epsilon greedy approach.

$$\pi(s) \leftarrow \begin{cases} \arg\max_a Q(s, a; \mathbf{w}) & \text{with prob } 1 - \epsilon \\ \text{random } a \in A & \text{with prob } \epsilon \end{cases}$$

Many other approaches for exploration.

(2) Update Policies

Bellman equation assumes we are acting greedily, however if we are exploring, how do we select the a' to update towards?

Two different approaches:

1. On-Policy: Compute expected reward using a' from same policy that we are acting with.
2. Off-Policy: Update towards with a different policy than we are acting with.

SARSA

- SARSA is an **on-policy update** rule. Instead of max a', use next decision from policy π, e.g. epsilon greedy.
- SGD update at each time step becomes

$$w_{s,a} \leftarrow Q(s, a; \mathbf{w}) - \eta(Q(s, a; \mathbf{w}) - [r + \gamma Q(s', \pi(s'); \mathbf{w})])$$

- Can be seen as taking an avg.

$$w_{s,a} \leftarrow (1 - \eta)Q(s, a; \mathbf{w}) + \eta(r + \gamma Q(s', \pi(s'); \mathbf{w}))$$

SARSA

- SARSA is an **on-policy update** rule. Instead of max a', use next decision from policy π, e.g. epsilon greedy.
- SGD update at each time step becomes

$$w_{s,a} \leftarrow Q(s, a; \mathbf{w}) - \eta(Q(s, a; \mathbf{w}) - [r + \gamma Q(s', \pi(s'); \mathbf{w})])$$

- Can be seen as taking an avg.

$$w_{s,a} \leftarrow (1 - \eta)Q(s, a; \mathbf{w}) + \eta(r + \gamma Q(s', \pi(s'); \mathbf{w}))$$

Q-Learning

- Q-Learning is an **off-policy update** rule, where a' is always max action and a is agent policy.
- SGD update at each time step becomes

$$w_{s,a} \leftarrow Q(s, a; \mathbf{w}) - \eta(Q(s, a; \mathbf{w}) - [r + \gamma \max_{a'} Q(s', a'; \mathbf{w})])$$

- Note, can still use epsilon greedy for a, but always update with $\max$, i.e. pure greedy.

SARSA versus Q-Learning

- **SARSA**: Better convergence and stability, however updates towards "wrong" policy.
- In practice, let $\epsilon \to 0$, update becomes original gradient.
- **Q-Learning**: Gradient always uses $\max$ even if s, a comes from epsilon greedy policy.
- Worse convergence, but widely used in practice.

Contents

Issues with Q-Learning

Q-Learning has so far assumed a very simple **parameterization** where our function estimator $Q(s, a; \mathbf{w})$ had a single parameter for each $S \times A$ i.e. $w_{s,a}$.

This is fine for grid-world but runs into issues on real-data:

- There are likely many states and actions that we will never see, how do we learn these parameters?
- Even if we do see everything, there will be way too many parameters in the model itself.

Luckily, we have spent all semester studying **function approximation**.

Issues with Q-Learning

Q-Learning has so far assumed a very simple **parameterization** where our function estimator $Q(s, a; \mathbf{w})$ had a single parameter for each $S \times A$ i.e. $w_{s,a}$.

This is fine for grid-world but runs into issues on real-data:

- There are likely many states and actions that we will never see, how do we learn these parameters?
- Even if we do see everything, there will be way too many parameters in the model itself.

Luckily, we have spent all semester studying **function approximation**.

Issues with Q-Learning

Q-Learning has so far assumed a very simple **parameterization** where our function estimator $Q(s, a; \mathbf{w})$ had a single parameter for each $S \times A$ i.e. $w_{s,a}$.

This is fine for grid-world but runs into issues on real-data:

- There are likely many states and actions that we will never see, how do we learn these parameters?
- Even if we do see everything, there will be way too many parameters in the model itself.

Luckily, we have spent all semester studying **function approximation**.

Linear Basis for Q-Learning

Instead of an independent parameterization, we can learn a linear model over a **feature basis** $\boldsymbol{\phi} : (S \times A) \to \mathbb{R}^d$.

Then define our **weights** to be $\mathbb{R}^d$. The Q function becomes

$$Q(s, a; \mathbf{w}) = \mathbf{w}^\top \boldsymbol{\phi}(s, a)$$

- Utilizing basis functions allows us to **generalize** to even unseen states and to share information between similar states.
- If we can learn states are similar, then we can better estimate their expected future reward, and inherently learn a better π.

Changes to the Loss

Weights $\mathbf{w}$ are now shared between each of the different states and actions (s, a).

$$Q(s, a; \mathbf{w}) = \mathbf{w}^\top \boldsymbol{\phi}(s, a)$$

Modify the loss to use the old weights $\mathbf{w}^{old}$ as the "target" next step.

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\mathbb{E}_{s,a}(Q(s, a; \mathbf{w}) - [r(s, a) + \gamma \sum_{s'} p(s' \mid s, a) \max_{a'} Q(s', a'; \mathbf{w}^{old})])^2$$

Stochastic gradient becomes:

$$\frac{\partial \mathcal{L}^{(s,a)}(\mathbf{w})}{\partial \mathbf{w}} \approx (Q(s, a; \mathbf{w}) - [r(s, a) + \gamma \max_{a'} Q(s', a'; \mathbf{w}^{old})]) \frac{\partial Q(s, a; \mathbf{w})}{\partial \mathbf{w}}$$

Research Frontier

- What can be said about the convergence properties of improved function approximators with RL?
- Are there better ways to train these models beyond SGD?
- What if we had even better models for state representations? How could this improve the use of RL techniques in practice?

Contents

Neural Networks in RL

Why stop at linear models for function approximation?

$$Q(s, a; \mathbf{w}) = \mathbf{w}^\top \boldsymbol{\phi}(s, a; \mathbf{W})$$

Here $\boldsymbol{\phi}(s, a; \mathbf{W})$ is an adaptive basis function or neural network.

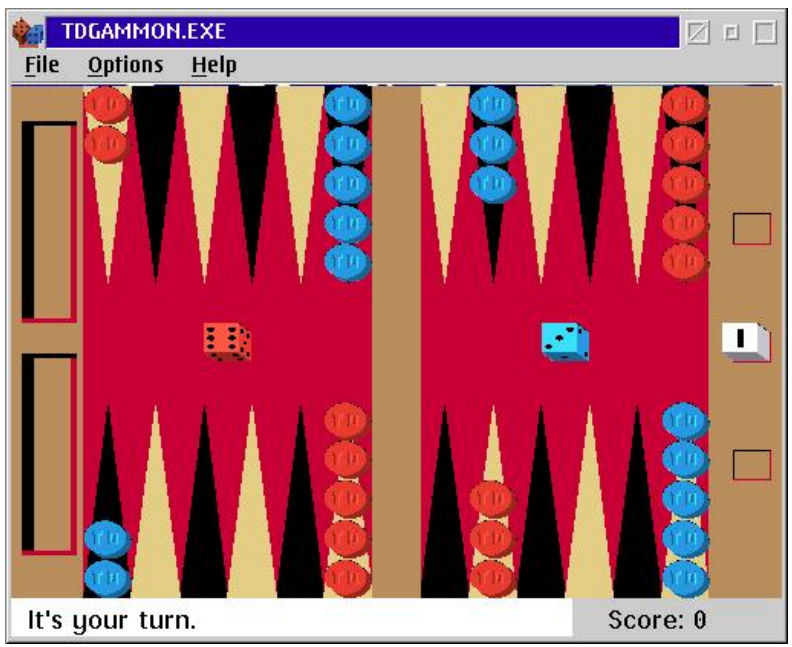
(Note: Pragmatic change. Theory of these models is poorly understood.)

Backgammon

TD-Gammon (1992)

- Computer backgammon system utilizing temporal difference learning and neural network.

$$Q(s, a; \mathbf{w}) = \mathbf{w}^\top \boldsymbol{\phi}(s, a; \mathbf{W})$$



Playing Atari

Atari playing RL bot (DeepMind, 2015)

Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih **Koray Kavukcuoglu** **David Silver** **Alex Graves** **Ioannis Antonoglou**

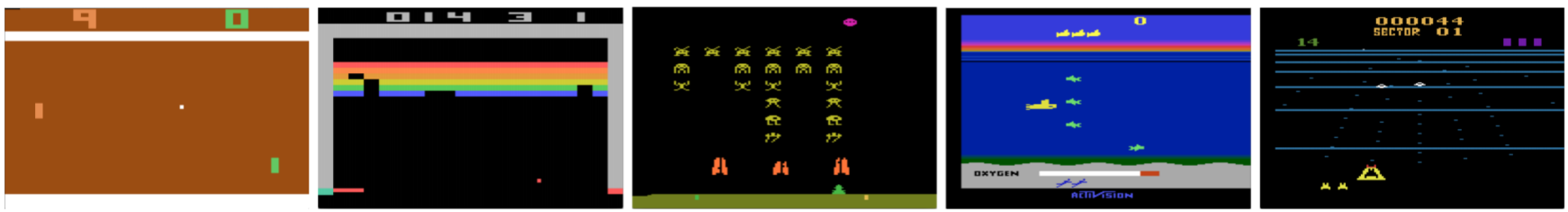
Daan Wierstra **Martin Riedmiller**

DeepMind Technologies

{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com

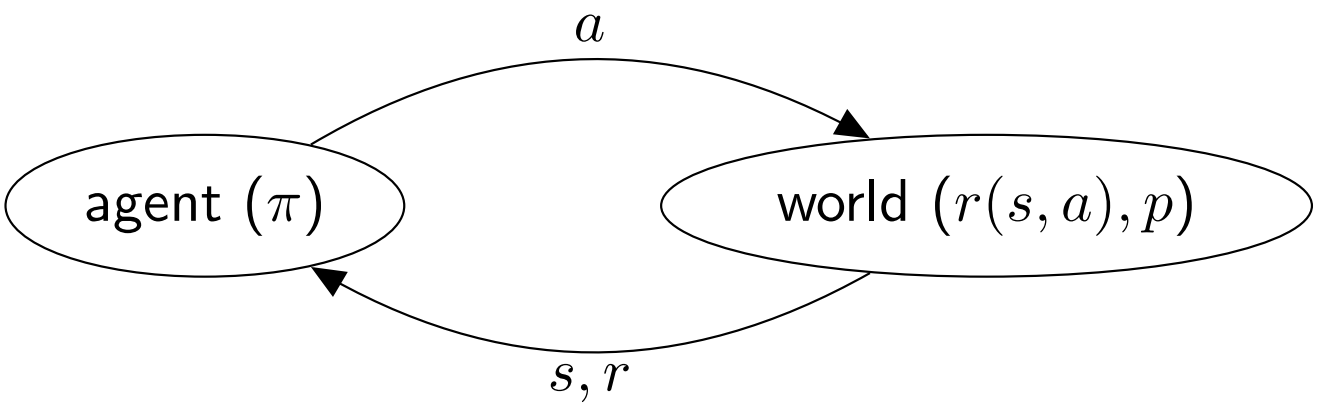
(Relatively easy to read paper, posted on website)

Goal of Paper



- Define a general-purpose model that is able to learn to play multiple arcade games.
- The agent is not told the rules of the game or its environment.

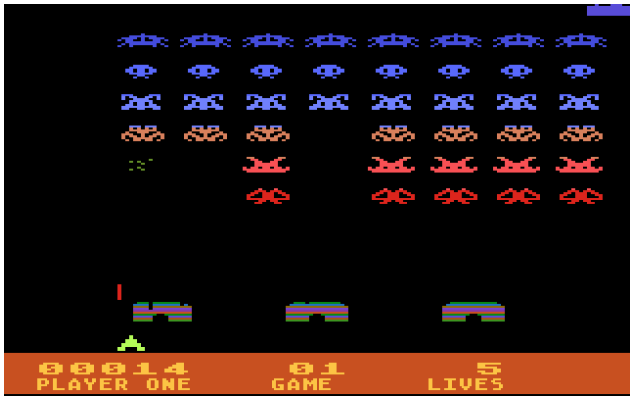
RL in Atari



- States: Image of the game world.
- Actions: Move controller $\leftarrow$, $\rightarrow$, $\uparrow$, $\downarrow$, button
- Reward: Score at the end of the game.
- Transition model: Defined by the rules of the game world itself

MDP: Succinctly

State

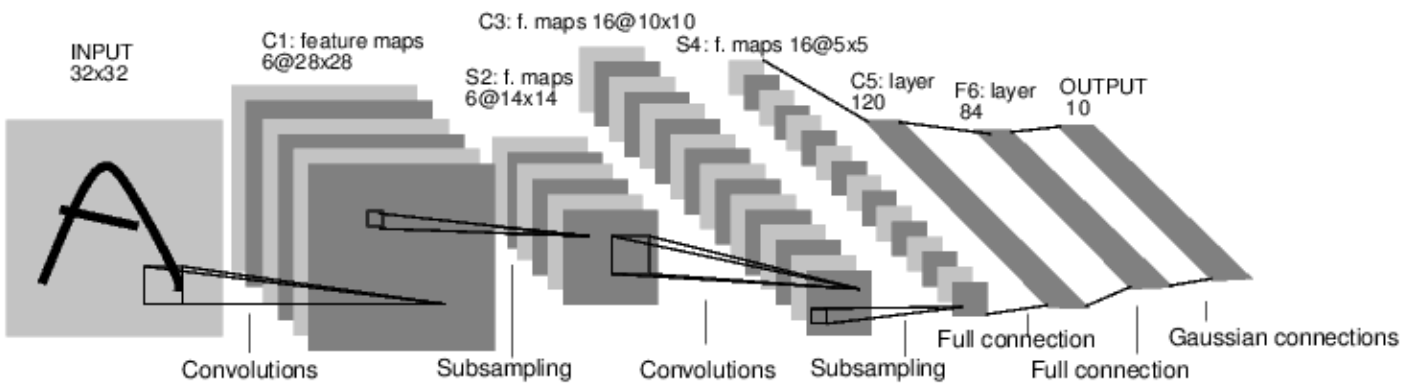


Actions

Approach: Convolutional Neural Networks (Reminder)

$$\mathbf{W}^1\mathbf{x}$$

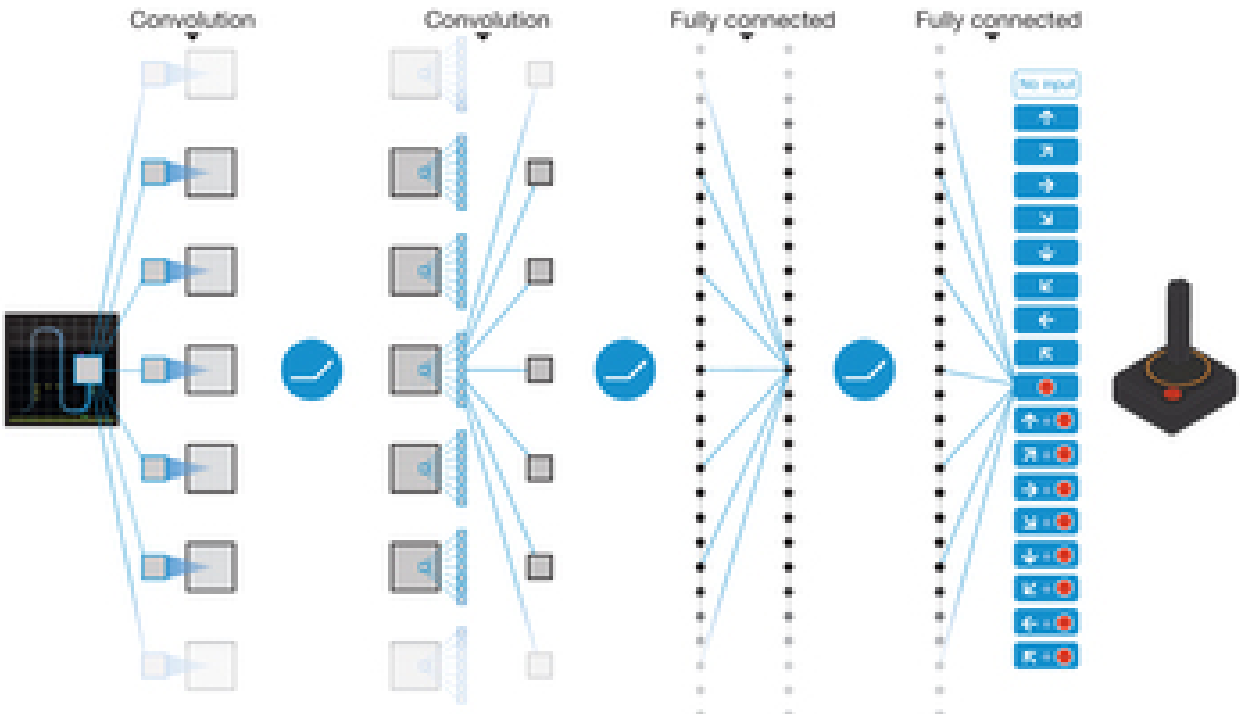
$$\begin{bmatrix} w_1 & w_2 & w_3 & 0 & \dots & & & 0 \\ 0 & w_1 & w_2 & w_3 & 0 & \dots & & 0 \\ 0 & 0 & w_1 & w_2 & w_3 & 0 & \dots & 0 \\ \vdots & & & & & & & \\ & \dots & & 0 & w_1 & w_2 & w_3 & 0 \\ & & \dots & & 0 & w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_4 \\ x_5 \end{bmatrix}$$



Deep Q-Network

Train with Q-Function using a network that takes in s and gives a score for each possible action a.

$$Q(s, a; \{\mathbf{w}_{a'}\}, \mathbf{W}) = \mathbf{w}_a^\top \boldsymbol{\phi}(s; \mathbf{W})$$



Experience Replay

Utilize a slightly different version of the TD-Learning algorithm.

- Keep a buffer $\mathcal{D}$ of past transitions (s, a, r, s') from the world.
- These past transitions must satisfy Bellman equations as well.
- Can enforce this without requiring any actions in the world.

Algorithm

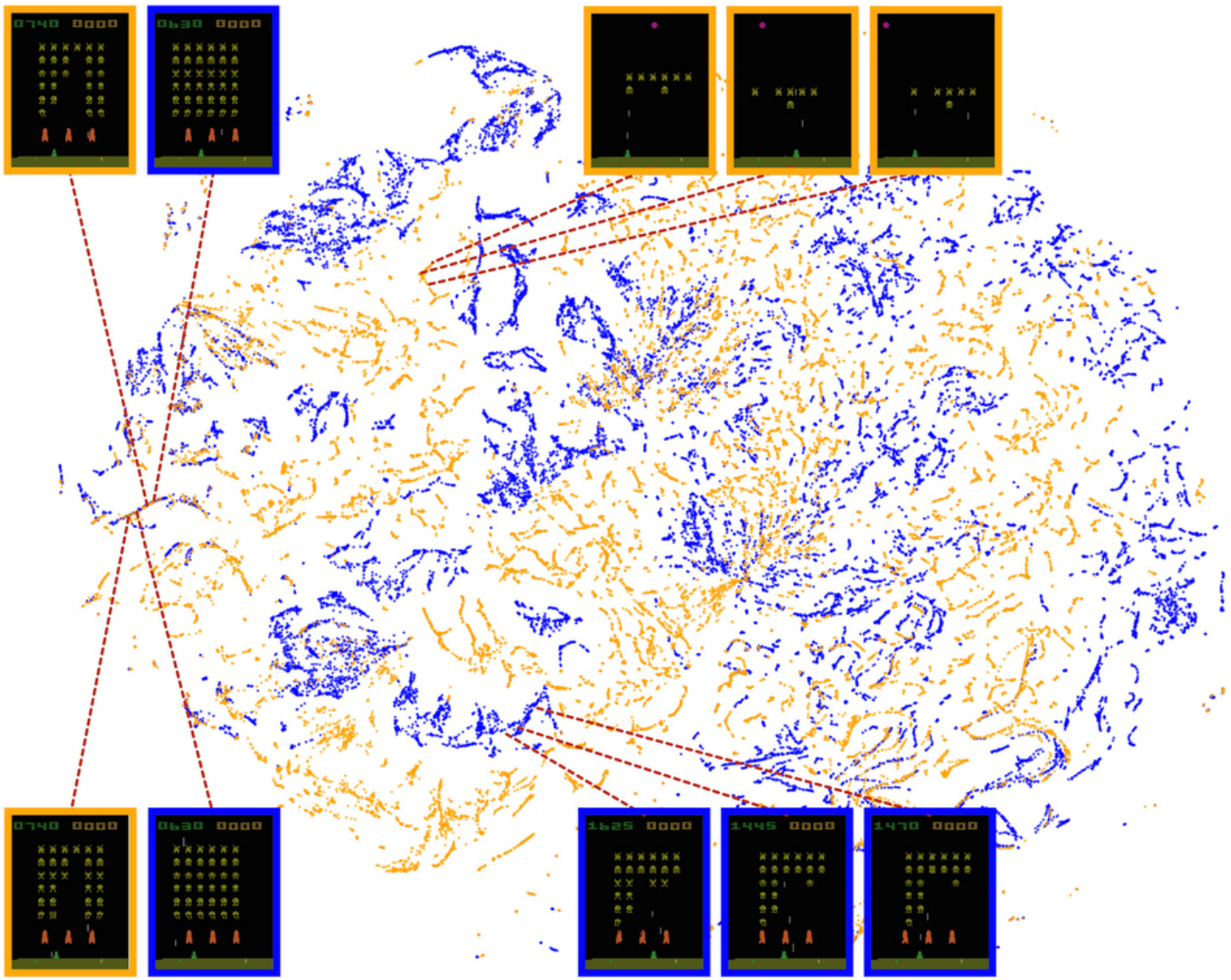
(Algorithm from paper, uses θ instead of $\mathbf{w}, \mathbf{W}$ and ϕ to mean the last 4 frames of the game.)

Algorithm 1 Deep Q-learning with Experience Replay

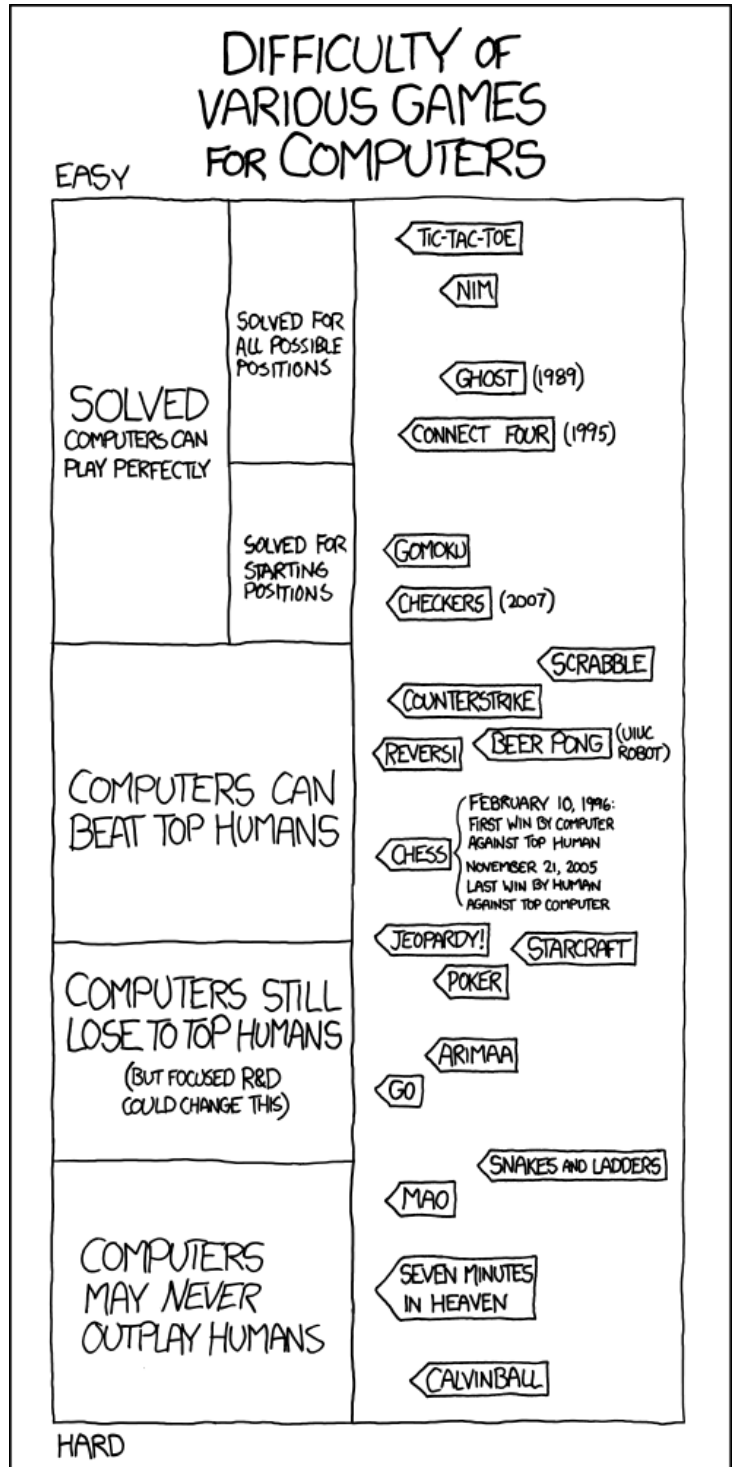
Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise 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

Example Video

Dimensionality Reduction



Contents

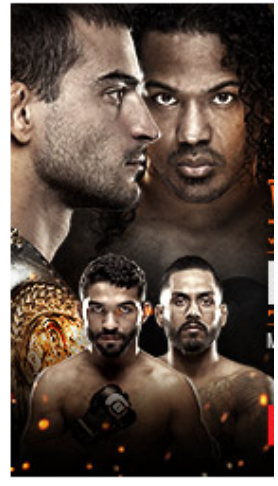

DIFFICULTY OF VARIOUS GAMES FOR COMPUTERS
EASY
SOLVED
COMPUTERS CAN PLAY PERFECTLY
SOLVED FOR ALL POSSIBLE POSITIONS
SOLVED FOR STARTING POSITIONS
COMPUTERS CAN BEAT TOP HUMANS
COMPUTERS STILL LOSE TO TOP HUMANS
(BUT FOCUSED R&D COULD CHANGE THIS)
COMPUTERS MAY NEVER OUTPLAY HUMANS
HARD
TIC-TAC-TOE
NIM
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

https://www.youtube.com/watch?v=Jq5SObMdV3o

ASIA PACIFIC

Google's Computer Program Beats Lee Se-dol in Go Tournament

By CHOE SANG-HUN MARCH 15, 2016



Lee Se-dol with his daughter Lee Hye-lim on his way to the last Go match with Google's AlphaGo artificial intelligence program in Seoul, South Korea. Kim Hong-Ji/Reuters

AlphaGo Overview

1. Learn a model to predict one-step move from experts
2. Refine by self-play reinforcement learning
3. Use as part of game-tree search.

Policy Setup

Given current board state s, distribution over actions a.

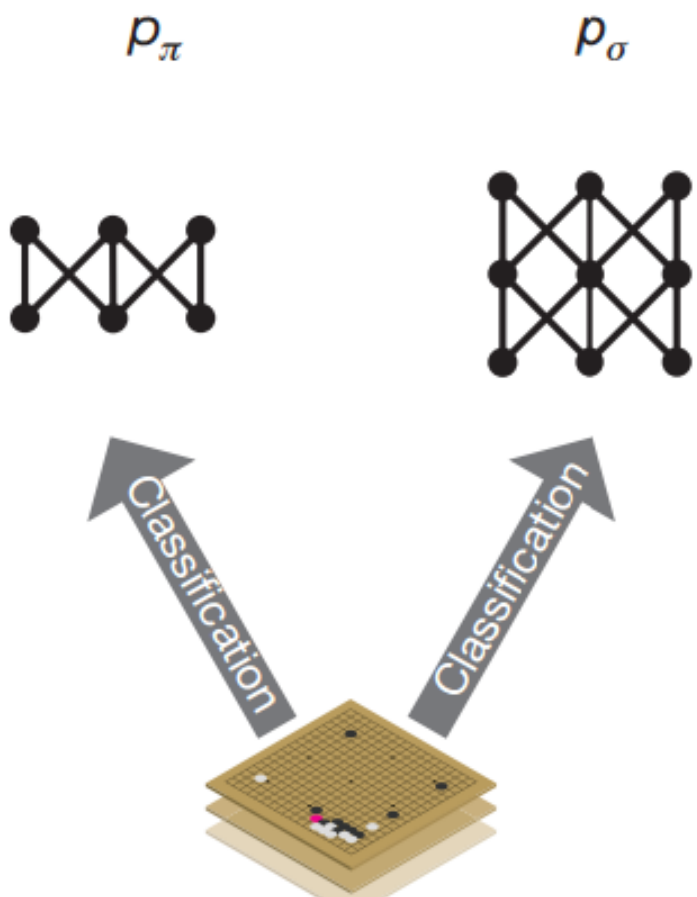
- Learn a policy, $p(a|s)$
- Estimate distribution with softmax.
- Gives a one-step Go player.

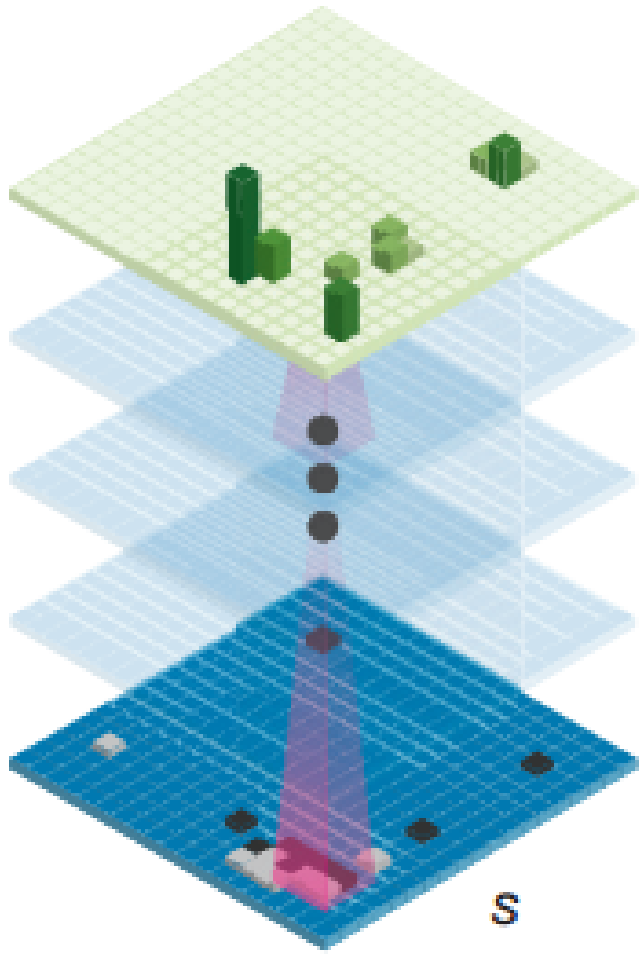
(1) Policy Network

Learned from 29.4 million positions from 160,000 expert games

Two models:

- $p_\pi(a|s)$; multiclass logistic regression (pattern+sparse features)
- $p_\sigma(a|s)$; deep convolutional network




$p_{\sigma/\rho}(a|s)$
s

Deep Convolutional Network

The first hidden layer zero pads the input into a 23x23 image, then convolves k filters of kernel size 5x5 with stride 1 with the input image and applies a rectifier nonlinearity. Each of the subsequent hidden layers 2 to 12 zero pads the respective previous hidden layer into a 21x21 image, then convolves k filters of kernel size 33 with stride 1, again followed by a rectifier nonlinearity.

Deep Convolutional Network

The first hidden layer zero pads the input into a 23x23 image, then convolves k filters of kernel size 5x5 with stride 1 with the input image and applies a rectifier nonlinearity. Each of the subsequent hidden layers 2 to 12 zero pads the respective previous hidden layer into a 21x21 image, then convolves k filters of kernel size 33 with stride 1, again followed by a rectifier nonlinearity.

The step size α was initialized to 0.003 and was halved every 80 million training steps, without momentum terms, and a mini-batch size of $m = 16$. Updates were applied asynchronously on 50 GPUs using DistBelief 61; gradients older than 100 steps were discarded. Training took around 3 weeks for 340 million training steps.

(2) Reinforcement Learning

- Refine one-step player by playing against itself.
- Popular technique for stochastic games (TD-Gammon)
- Reinforcement learning objects to account for single-step bias

Self-Play with Policy Gradient

Start with p_σ and play against itself to learn:

- $p_\rho(a|s)$; deep convolution network (policy gradient)

Process: Training epoch $J+1$

1. Sample opponent from previous version of model $j < J$
2. Play game between players p_{ρ^J} and p_{ρ^j}
3. Update weights using policy gradient on RL objective

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

where $z_t \in \{-1, 1\}$ represents the final outcome of the game.

Value Network

- Policy network trains only move at state.
- Useful also to know the value of a state.

$$v(s) = E_{p_\rho}[z_t|s]$$

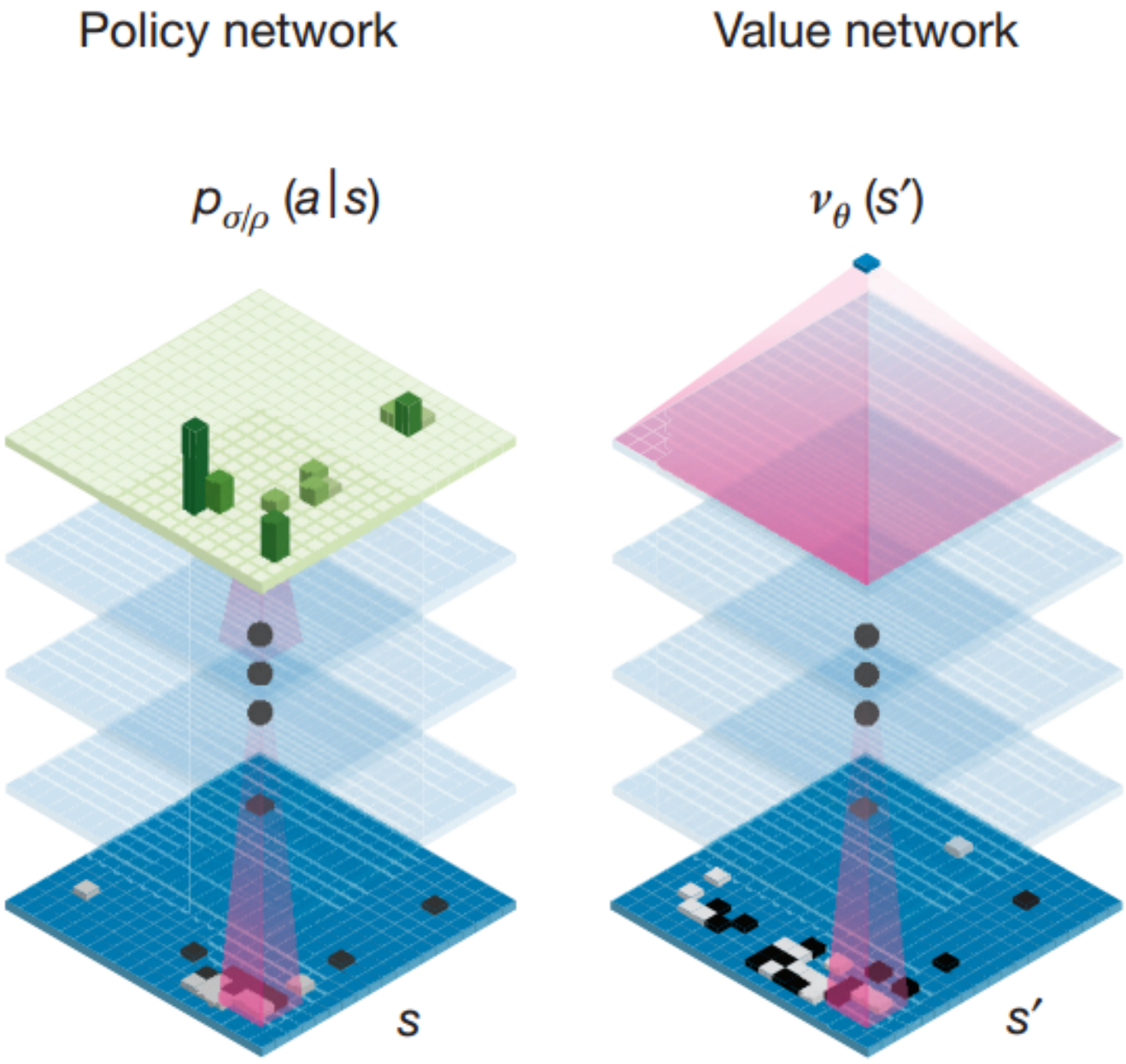
Generally done using game-specific heuristics.

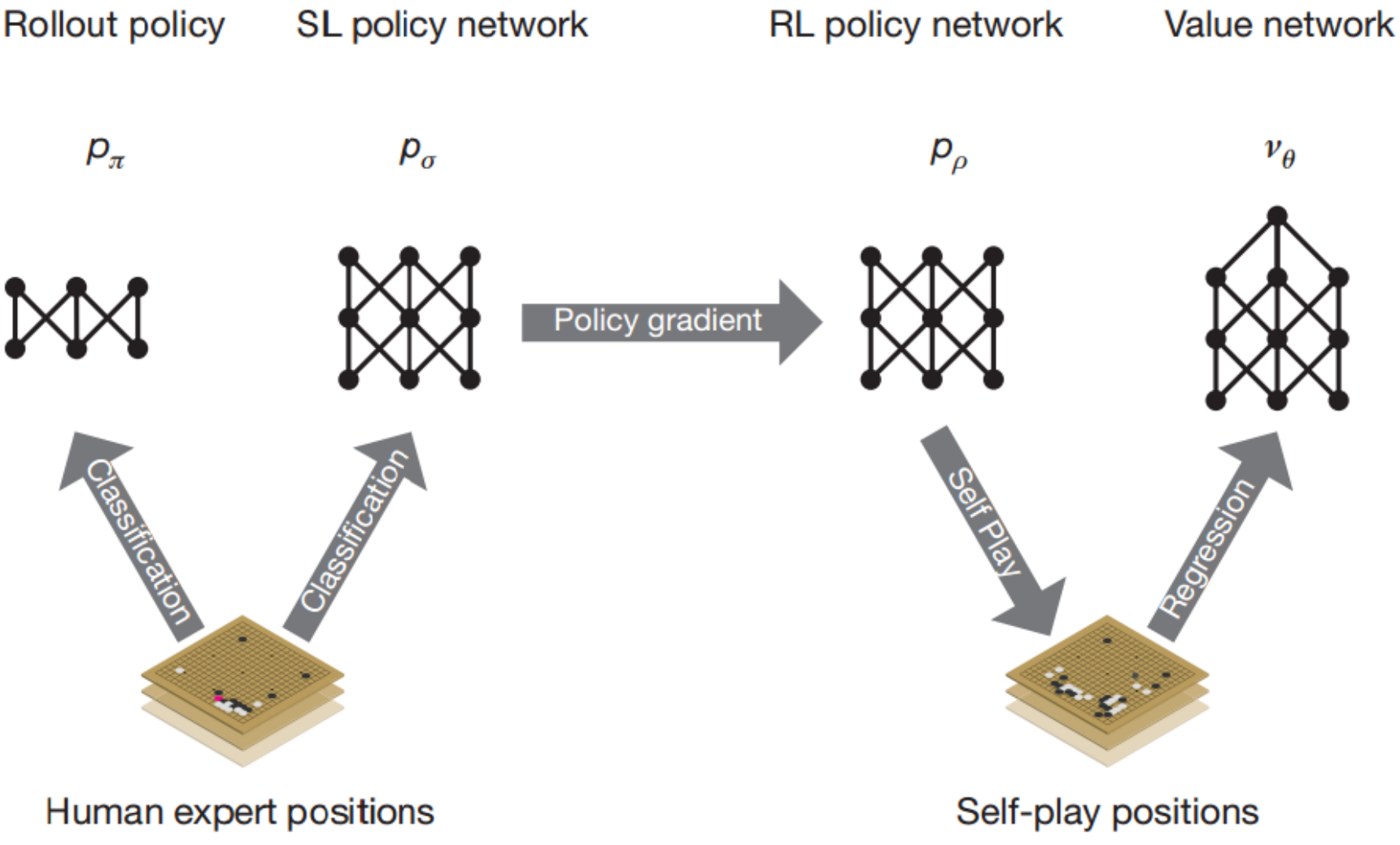
Value Network

Apply similar architecture for computing state value,

- v_θ; deep CNN regression
- Trained on self-play data set.
- Minimize MSE with final self-play result.

When trained on the KGS data set in this way, the value network memorized the game outcomes rather than generalizing to new positions, achieving a minimum MSE of 0.37 on the test set, compared to 0.19 on the training set. To mitigate this problem, we generated a new self-play data set consisting of 30 million distinct positions, each sampled from a separate game. Each game was played between the RL policy network and itself until the game terminated.


Policy network
Value network
$p_{\sigma/\rho}(a|s)$
$v_\theta(s')$
s
s'


Rollout policy
SL policy network
RL policy network
Value network
p_π
p_σ
p_ρ
v_θ
Policy gradient
Classification
Classification
Self Play
Regression
Human expert positions
Self-play positions

(3) Game Search

Utilize the learned models with an advanced game-search algorithm

- Similar to standard game tree algorithms (CS182)
- Monte Carlo Tree Search (MCTS)
  - Select
  - Expand
  - Eval
  - Update/Backup
- Progressively expands the search space based on models

Select and Expansion

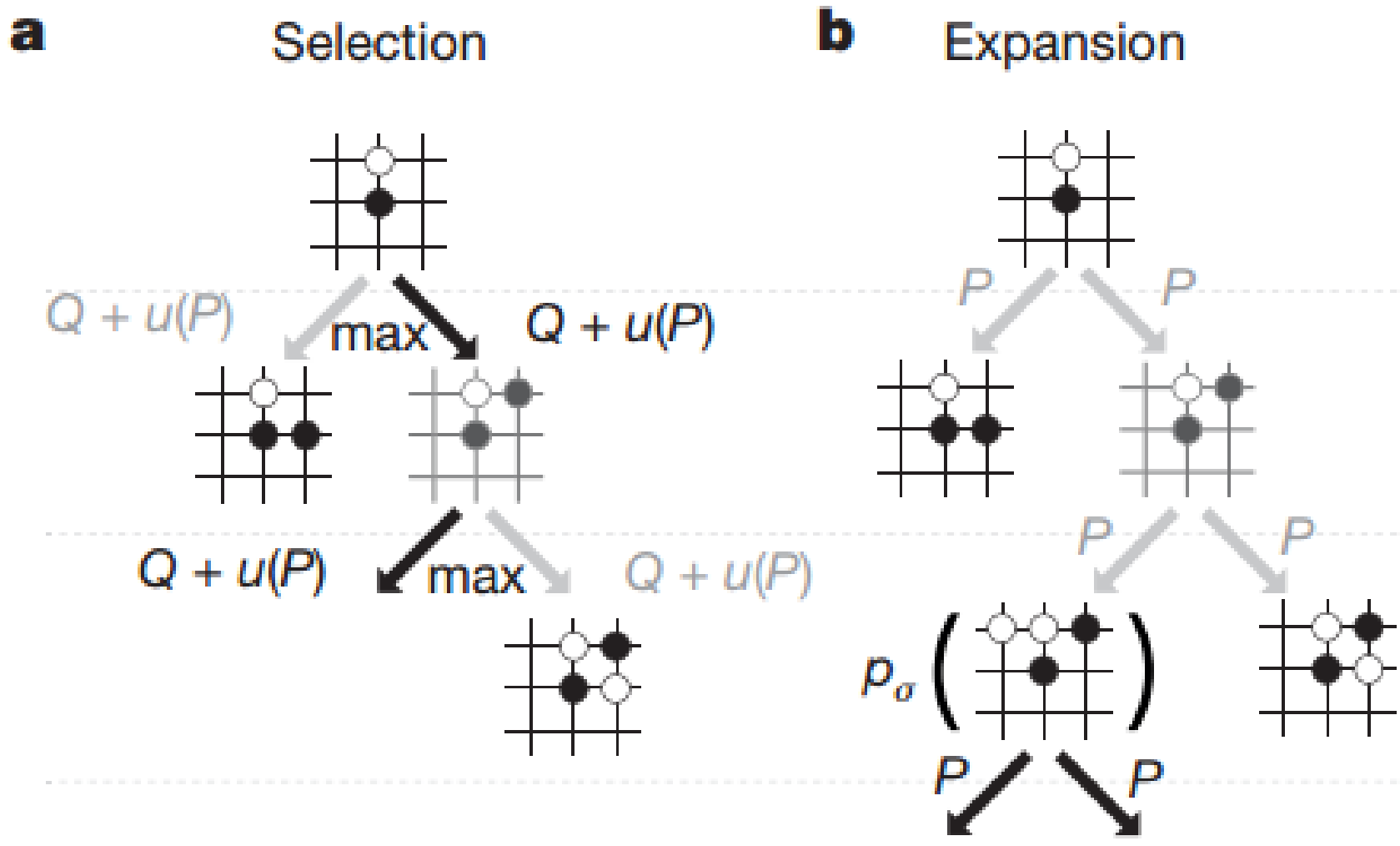
- $Q(s, a)$; current expected value of taking action a at s
- $u(s, a)$; prior for taking a at s defined by p_σ

Selection step at state s,

$$\arg\max_a Q(s, a) + u(s, a)$$

Based on selection, either move to seen node or **expand**.

Game Search


a
Selection
Q + u(P)
max
Q + u(P)
Q + u(P)
max
Q + u(P)
b
Expansion
P
P
P
P
P
P
p_σ

State Evaluation

Reached "leaf" state s_L, want to **evaluate**

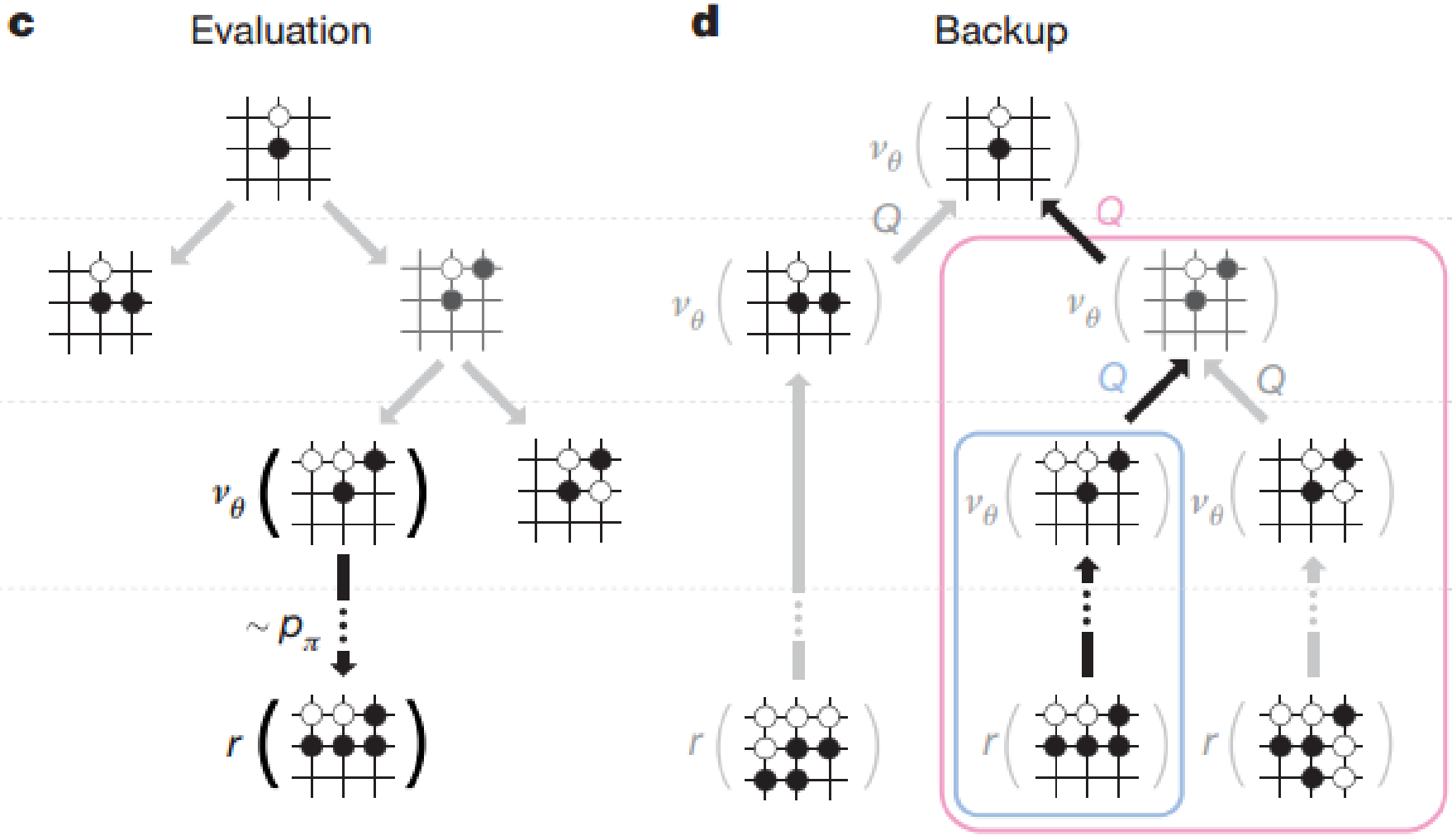
- Compute value as $V(s_L)$,

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

  where R is a rollout. Monte carlo simulation using p_π

- Convex combination of value network and simulation under simple model

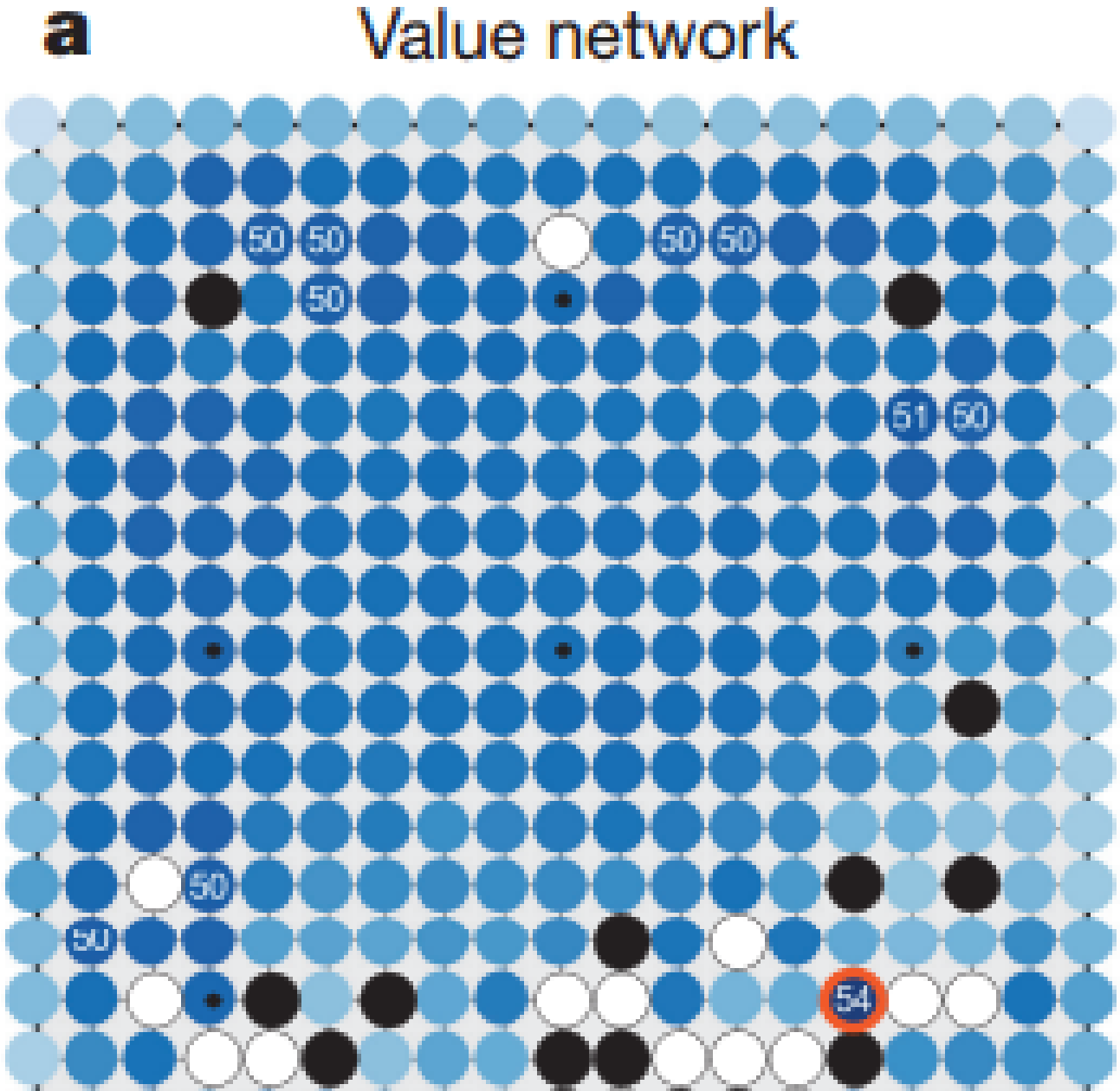
Why not p_σ? Where did p_ρ go?


c
Evaluation
v_θ
$\sim p_\pi$
r
d
Backup
v_θ
Q
Q
v_θ
v_θ
Q
Q
v_θ
v_θ
r
r
r

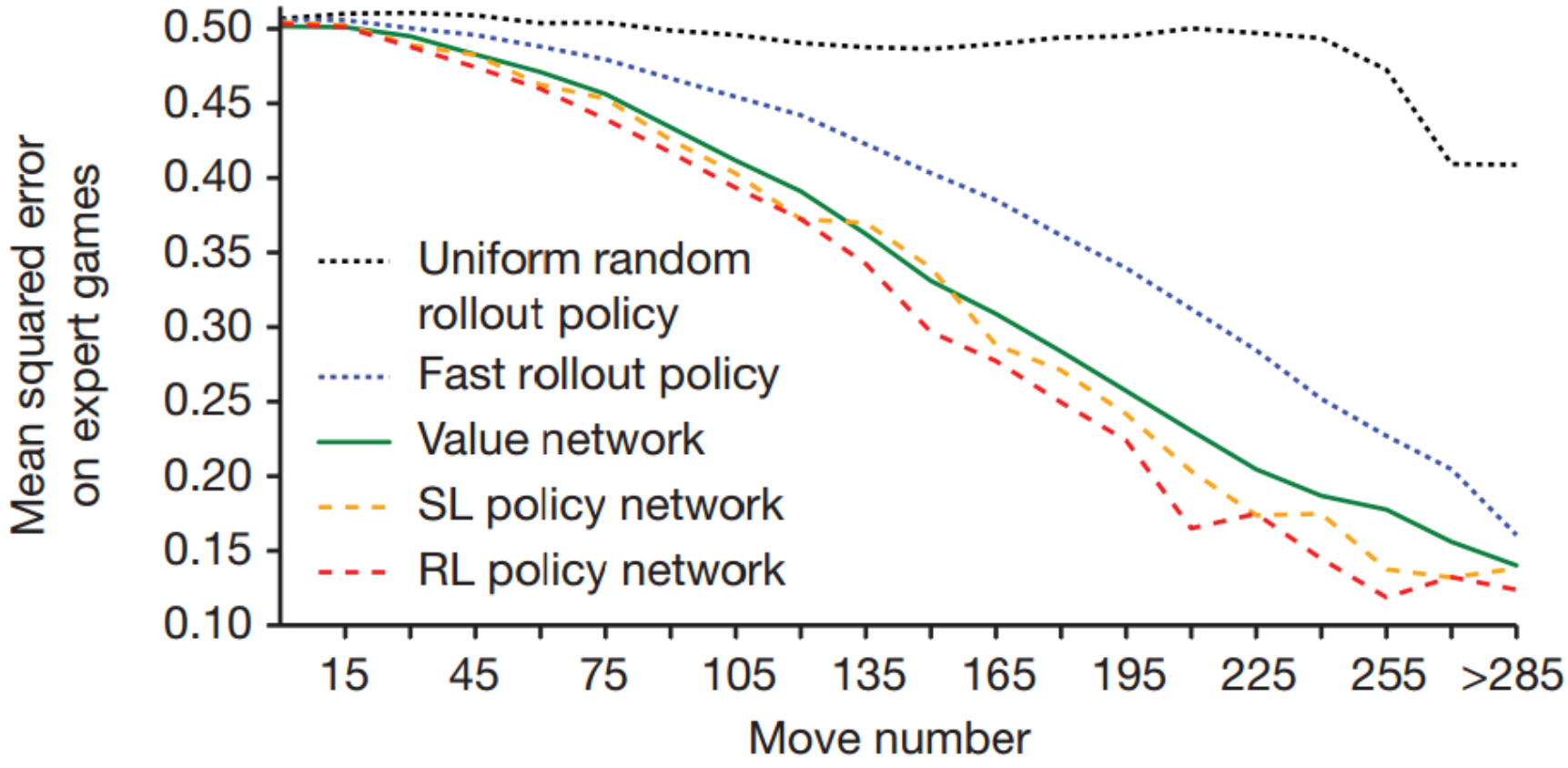
Move Selection

- After leaf evaluation all previous Q values are **updated** based on $V(s_L)$
- Process is run many times.
- Actual play is based on most commonly taken action.

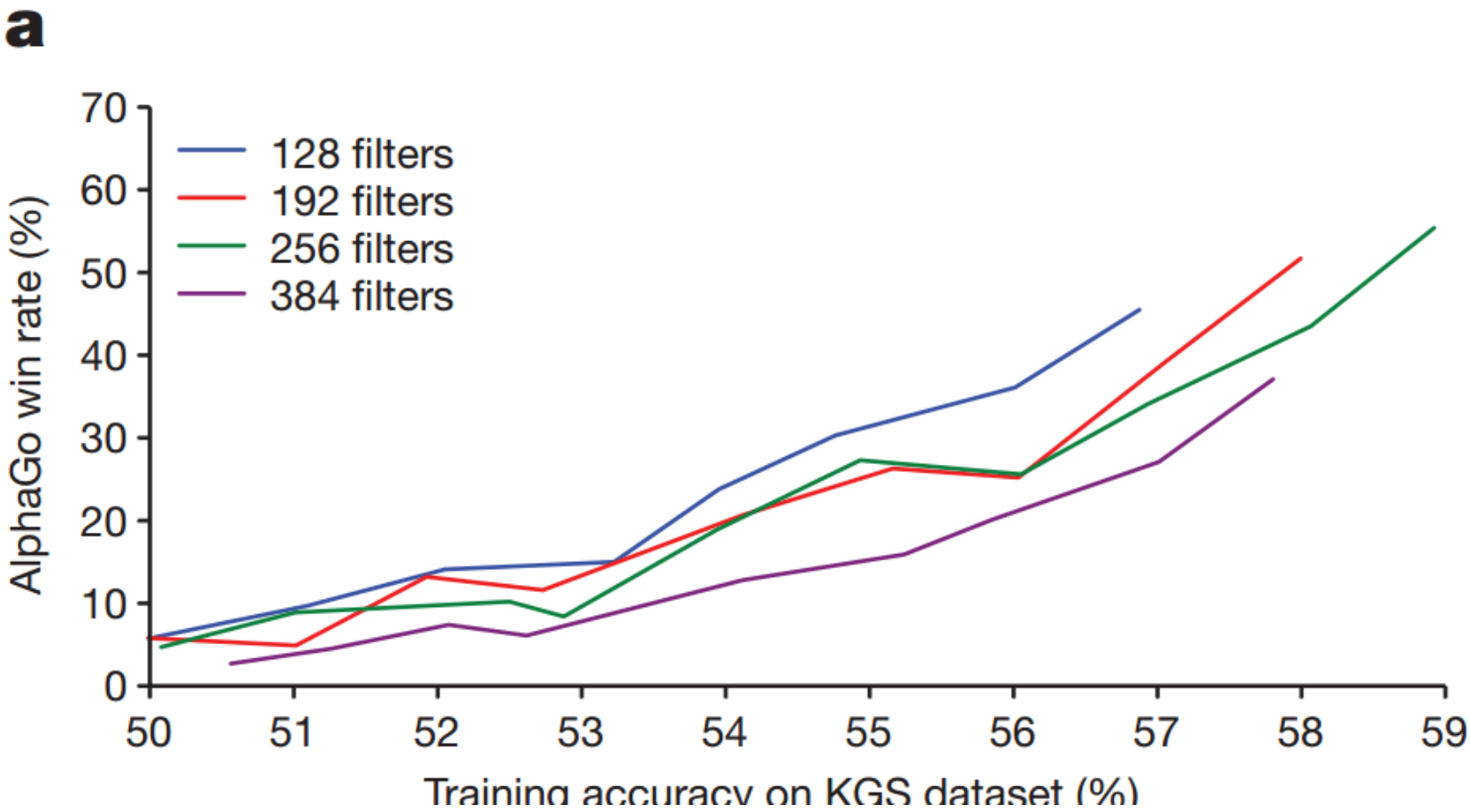
Results



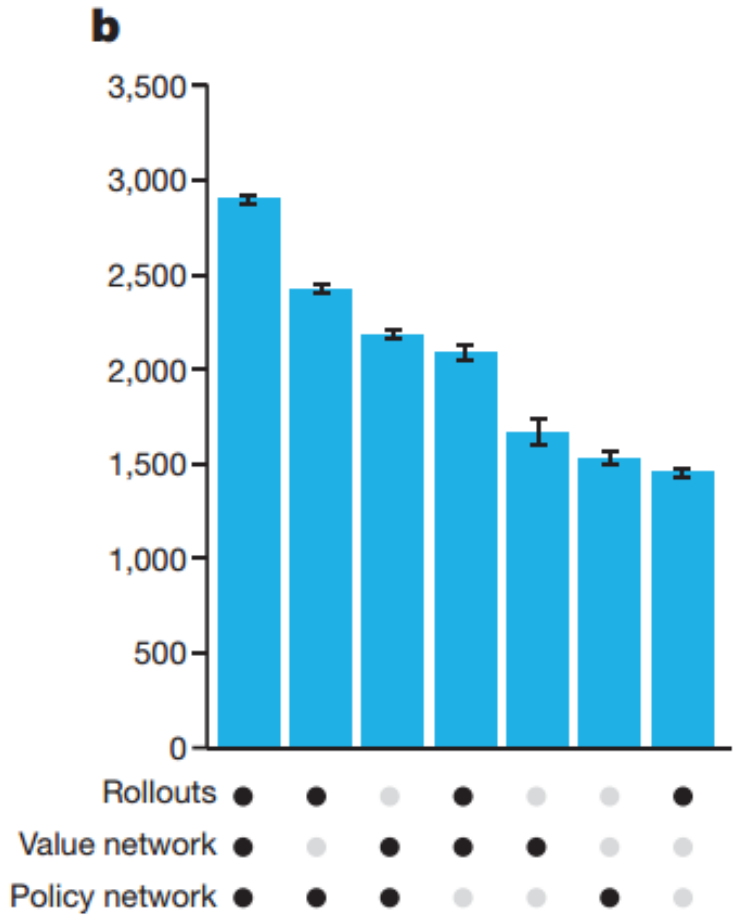
Results


Mean squared error on expert games
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
Uniform random rollout policy
Fast rollout policy
Value network
SL policy network
RL policy network
15
45
75
105
135
165
195
225
255
>285
Move number

Results


a
AlphaGo win rate (%)
70
60
50
40
30
20
10
0
128 filters
192 filters
256 filters
384 filters
50
51
52
53
54
55
56
57
58
59
Training accuracy on KGS dataset (%)

Results


b
3,500
3,000
2,500
2,000
1,500
1,000
500
0
Rollouts
Value network
Policy network