DSCI 565: REINFORCEMENT LEARNING

Ke-Thia Yao

Lecture 22 2025 November 19

Reinforcement Learning

- In reinforcement learning, the machine learning system takes actions sequentially in an environment and receives rewards (or penalties) for its actions in trying to solve a problem
- Key characteristic of reinforcement learning
  - Trial-and-error search: must explore the environment to learn appropriate actions
  - Delay reward: reward is given immediately after each action
- Examples and applications of reinforcement learning
  - Playing the game of Go. Each action is placing a stone on the Go board. The reward is winning the game.
  - Mobile robot finding its way to a charging station. Actions are turning on/off actuators and motors. Penalty for falling downstairs, and reward for reaching the charging station.

2024 Turing Award

- ACM has named Andrew G. Barto and Richard S. Sutton as the recipients of the 2024 ACM A.M. Turing Award for developing the conceptual and algorithmic foundations of reinforcement learning. In a series of papers beginning in the 1980s, Barto and Sutton introduced the main ideas, constructed the mathematical foundations, and developed important algorithms for reinforcement learning—one of the most important approaches for creating intelligent systems.

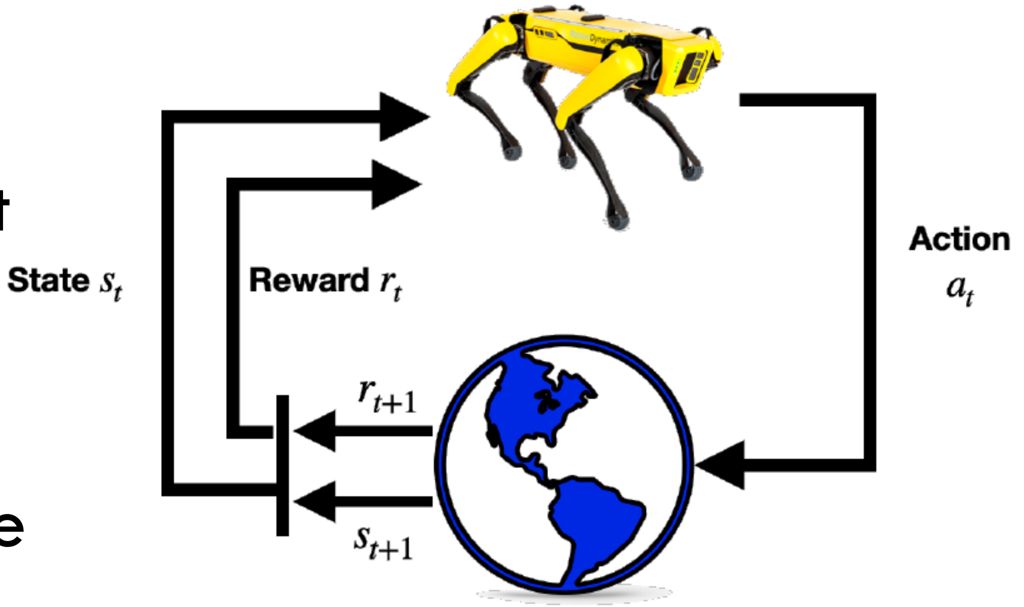
- Barto is Professor Emeritus of Information and Computer Sciences at the University of Massachusetts, Amherst. Sutton is a Professor of Computer Science at the University of Alberta, a Research Scientist at Keen Technologies, and a Fellow at Amii (Alberta Machine Intelligence Institute).

References

- Sutton, Richard S., Andrew G. Barto. *Reinforcement Learning, second edition: An Introduction*. 2nd edition.
- Zhang, Aston, Zachary C. Lipton, Mu Li, and Alexander J. Smola. 2023. *Dive into Deep Learning*. 1st edition. Cambridge University Press.
- Alpaydin, Ethem. 2020. *Introduction to Machine Learning, Fourth Edition*. The MIT Press.
- Hands-On Machine Learning with Scikit-Learn and PyTorch, 3rd Edition, Aurélien Géron

Reinforcement Learning Setup

- An agent that makes decisions
- An environment that agent interacts with
- At any time, the environment is in a certain state, and the agent can directly observe it
- The agent takes an action that changes the environment state
- Based on the sequence of actions taken, the agent sometime receives a reward

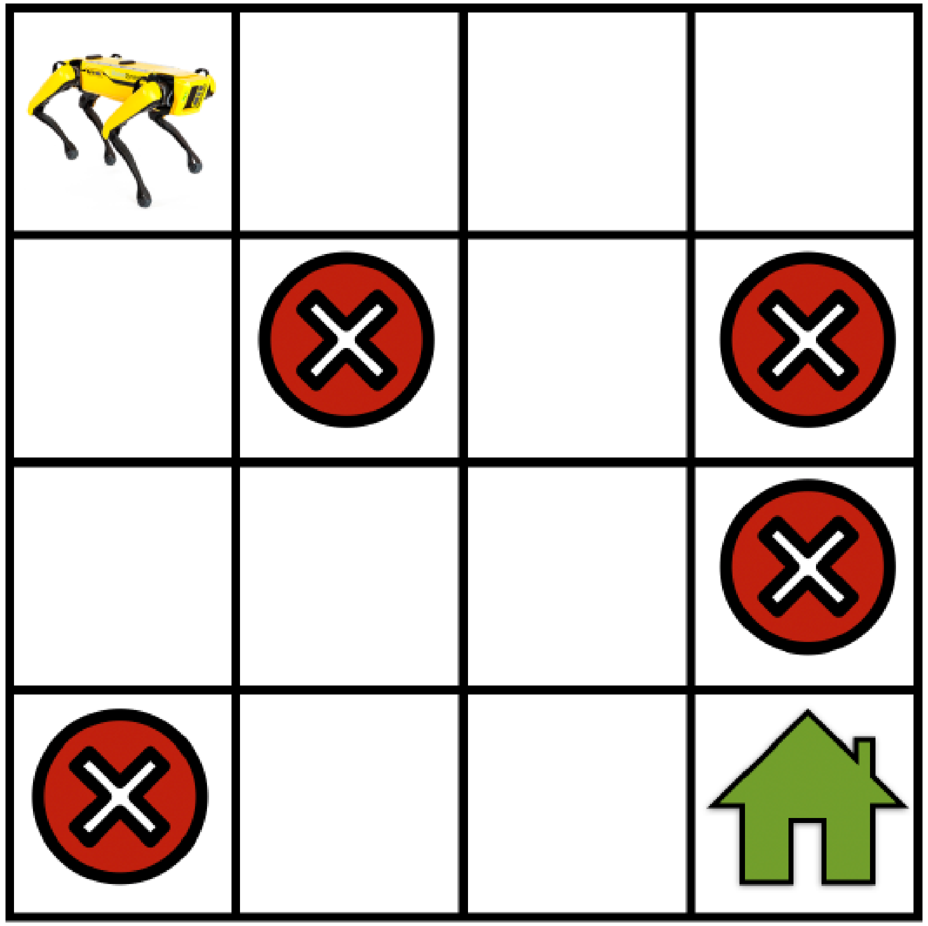


Learning with a Critic

- Different from supervised learning; we do not have a teacher informing the learner which actions are good/bad
- Critic provide feedback about past actions
- The feedback is scarce, and when it comes it comes late
- Need to solve the credit assignment problem: how do we distribute credit for success over the sequence of actions
- General approach is to learn internal value for intermediate states/actions that lead us to goal states with rewards

Markov Decision Process (MDP)

- MDP is a framework for modeling stochastics decision problems
  - Like Markov model but with actions
- MDP is defined by
  - Set of states $\mathcal{S}$, say location of robot
  - Set of actions $\mathcal{A}$ that change state, say *go-right, go-left, go-up, go-down*
  - Action results are probabilistic. Transition function $T: \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0,1]$, i.e., $T(s,a,s') = P(s'|s,a)$
  - Reward function $r: \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$
    - Note: reward can be probabilistic $P(r|s,a)$
- $MDP : (\mathcal{S}, \mathcal{A}, T, r)$



Markov Property

- Markov property for MDP: the environment response at time t+1 only depends on the state at time t, s_t
- It does matter how the environment reached state s_t
- In formulating a problem as an MDP, we should try to incorporate sufficient state information to satisfy the Markov property
- Say, if ability of the robot to turn depends on the its current speed, then we should include the velocity, not just the location

Return and Discount Factor

- Let trajectory be a sequence of state, action and reward
$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, \dots)$$
  - Where $r_t = r(s_t, a_t)$
- The return of a trajectory
$$R(\tau) = r_0 + r_1 + r_2 + \cdots$$
- Goal of reinforcement learning is to the largest return
- Discounted return to favor short trajectory, for $\gamma < 0$
$$R(\tau) = r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots = \sum_t^{\infty} \gamma^t r_t$$
  - And for dealing with infinitely long trajectories

Stochastic policy

- Stochastic policy π defines the agent's behavior
- Stochastic policy is a conditional probability distribution of taking an action $a \in \mathcal{A}$ given in state $s \in \mathcal{S}$:

$$\pi(a|s) \equiv P(a|s)$$

- Deterministic policy is a special case of stochastic policy
  - $\pi(a'|s) = 1$ for $a' \in \mathcal{A}$,and $\sum_{a \neq a'} \pi(a|s) = 0$

- Shorthand notation: sometimes policy is rewritten as $\pi(s)$

Policy Represented as a Table

	→	←	↑	↓
(0,0)	0.5	0	0	0.5
(0,1)				

Policy Value

- Let value function $V^{\pi}(s_o)$ be the expected return of policy π starting in state s_0 over all possible trajectories

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, \dots)$$

- There are two sources randomness in generating trajectories
  - Policy randomly selects an action $a_t \sim \pi(a_t|s_t)$, or $a_t \sim \pi(s_t)$
  - Transition function randomly selects a state $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$
- Define value function recursively

$$V^{\pi}(s) = E_{a\sim\pi(s)}\left[r(s,a) + \gamma\left[E_{s'\sim P(s'|s,a)} V^{\pi}(s')\right]\right]$$

- Or

$$V^{\pi}(s) = \sum_{a\in\mathcal{A}} \pi(a|s)\left[r(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)V^{\pi}(s'))\right]$$

Action-Value Function Q

- Sometimes it is useful to think about the value of an action a at a particular state s
- Let the action-value function $Q^{\pi}(s,a)$ be the expected return of taking an action a at a given state s

$$Q^{\pi}(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \sum_{a' \in \mathcal{A}} \pi(a'|s') Q^{\pi}(s',a')$$

Optimal Stochastic Policy

- Find the optimal policy π^* that achieves the maximal expected return

$$\pi^* = \text{argmax}_\pi V^\pi(s_0)$$

- Let $V^* \equiv V^{\pi^*}$ be the corresponding optimal value function
- For optimal deterministic policy, select the best action for each state

$$\pi^*(s) = \underset{a \in \mathcal{A}}{\text{argmax}} \left[E[r(s,a)] + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^*(s') \right]$$

- In our case of completely observable states and stationary environment, there always exists an optimal deterministic policy
- How to compute V^*?

Value Iteration

- Optimal value function, aka Bellman's Equation

$$\mathrm{V}^*(s) = \max_{a \in \mathcal{A}} \left\{ E[r(s,a)] + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^*(s')] \right\}$$

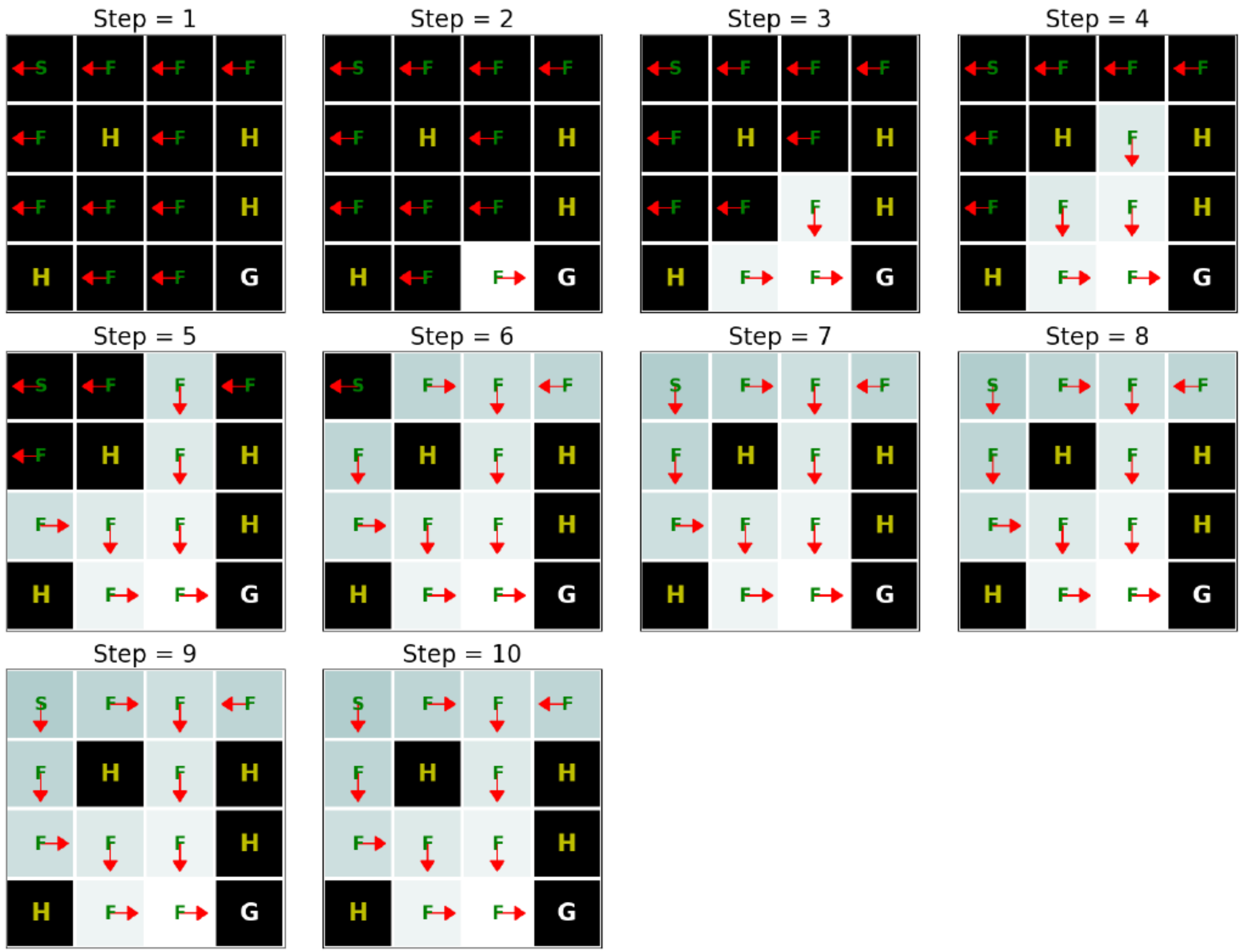
- Value iteration approach finds the optimal value function V^* using principles of dynamic programming,
- Initial arbitrary values for $V_0(s)$ for all $s \in \mathcal{S}$
- Then iterate

$$V_{k+1}(s) = \max_{a \in \mathcal{A}} \left\{ E[r(s,a)] + \sum_{s' \in \mathcal{S}} P(s'|s,a) V_k(s') \right\}$$

- As it turns out as $k \to \infty$

$$V^*(s) = \lim_{k \to \infty} V_k(s) \ \forall s \in \mathcal{S}$$

Value Iteration Example



Model-based Learning

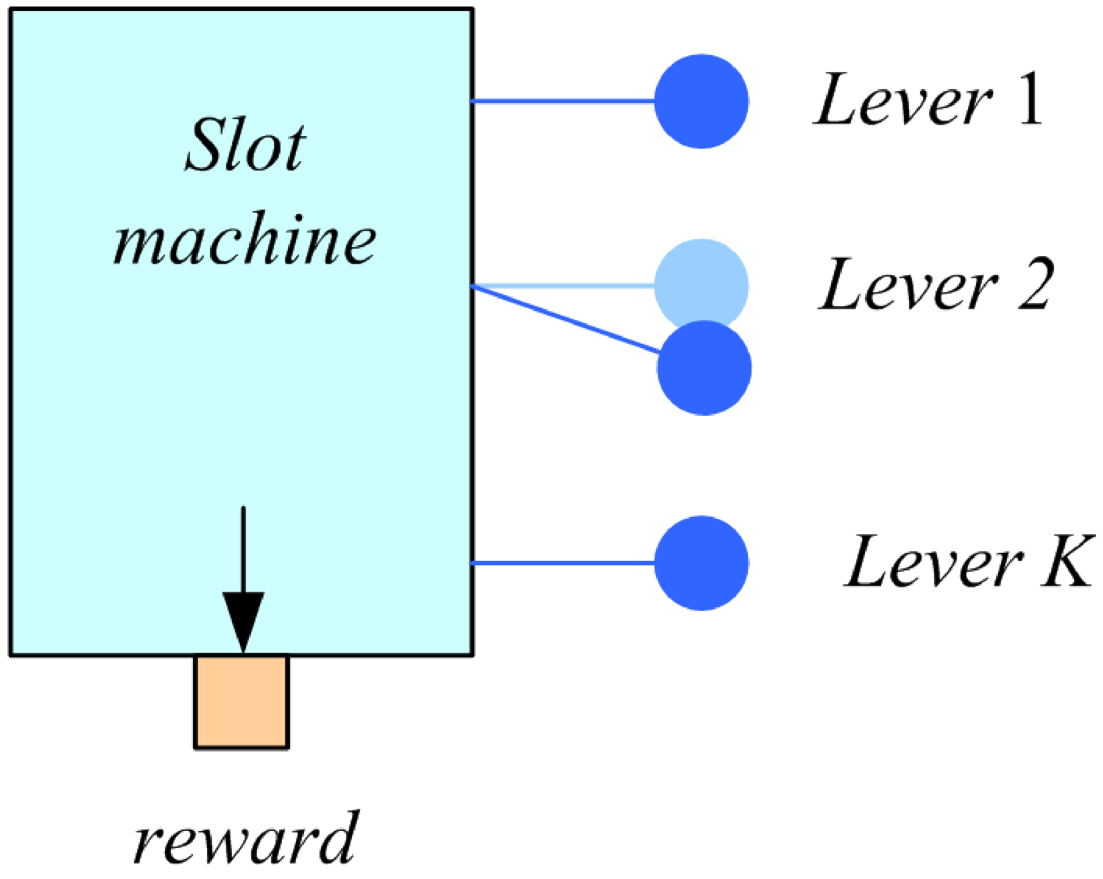
- In model-based learning, we completely know the environment
  - Agent knows the set of states S and the set of actions A
  - Agent knows the transition function T and the reward function r
- Exploration is not needed
- Value iteration is an example of model-based learning

Temporal Difference (TD) Learning

- Learning using model-free learning paradigm
  - Agents know the states and actions
  - But do not know transition function and reward function
- Temporal difference learning is like value iteration, except that it needs to explore to learn the environment, i.e., transition and reward
- Update using temporal difference between our current estimate value and the discounted value of next state and the reward received

Single State Case: K-armed Bandit

- Learning problem
  - Have k different actions, $|\mathcal{A}| = k$
  - Each action receives a reward chose from unknown fixed probability distributions
  - Actions do not change the state, $|\mathcal{S}| = 1$
- Need to estimate action values
  - Explore to improve estimates
  - Exploit estimates to maximize return

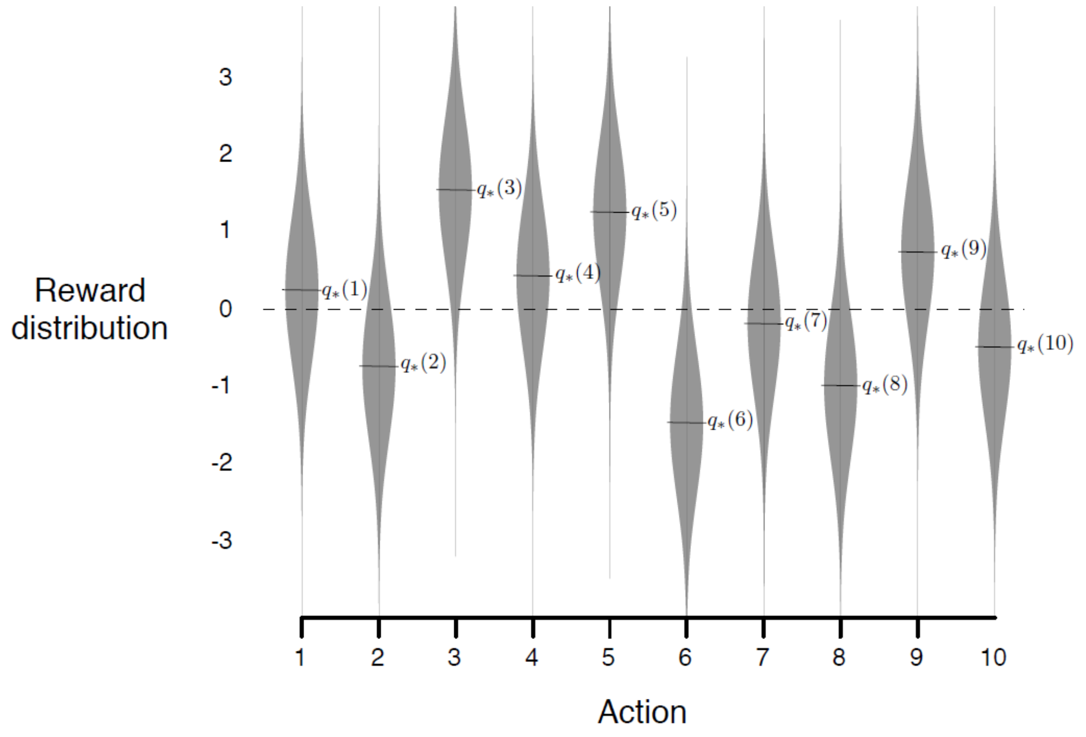


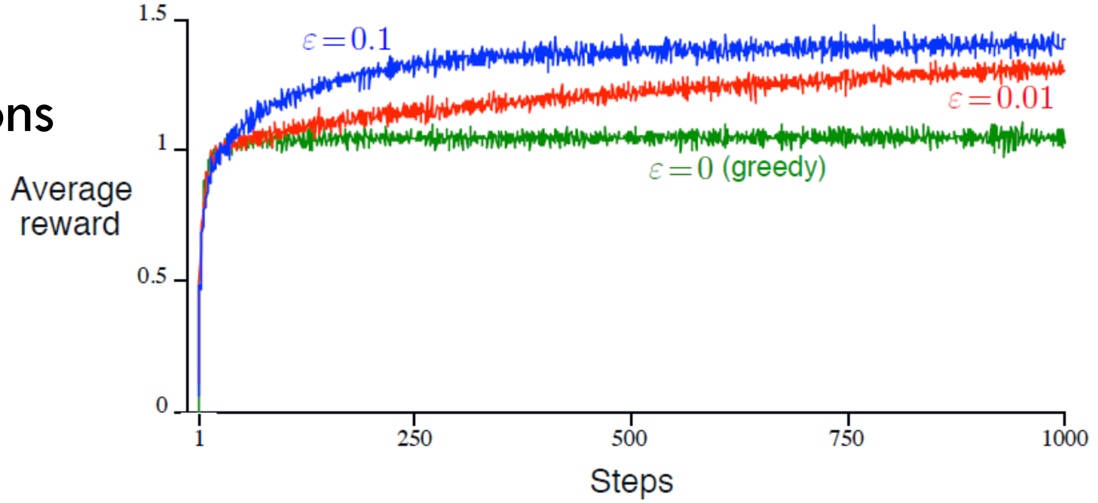
ϵ-greedy Method

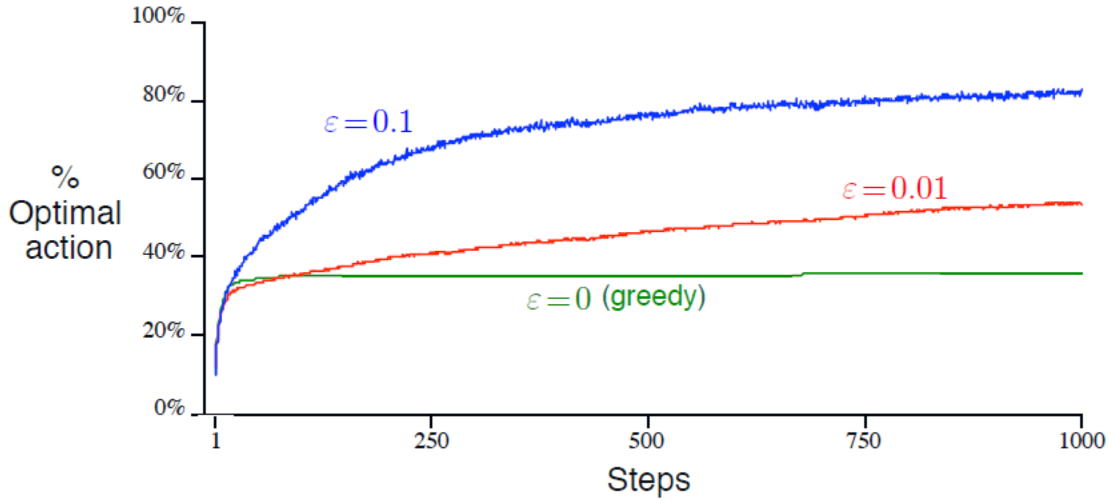
- $\epsilon \in [0,1]$ is a hyperparameter
  - Larger ϵ implies more exploration
- Let $Q_t(a)$ be the estimated reward of taking action a at time t
  - At time $t = 0$, initialize $Q_0(a) = \text{constant}$
- Let $r_{t+1}(a)$ be the actual reward at time $t + 1$ after taking action a
- At each step
  - With probability $1 - \epsilon$, select the action with highest action value $Q_t(a)$
  - Otherwise, randomly select an action
  - Update action value $Q_{t+1}(a) = Q_t(a) + \eta[r_{t+1}(a) - Q_t(a)]$

10-armed Bandit Testbed

- Average results over 2000 simulation runs
- For each run generate random reward distributions







Exploration and Exploitation Strategies

- ϵ-greedy: with probability ϵ choose one action randomly, with probability $1-\epsilon$ choose the best action
- Softmax to probabilistically choose actions with higher rewards:

$$P(a|s) = \frac{\exp Q(s,a)}{\sum_{b\in\mathcal{A}} \exp Q(s,b)}$$

- Softmax with temperature T

$$P(a|s) = \frac{\exp\left[\frac{Q(s,a)}{T}\right]}{\sum_{b\in\mathcal{A}} \exp\left[\frac{Q(s,b)}{T}\right]}$$

- Upper-confidence-Bound (UCB), add term to favor under explored actions

$$\operatorname*{argmax}_{a}\left[Q(s,a) + c\sqrt{\ln\frac{t}{N(a)}}\right]$$

Deterministic Case: Deterministic Rewards and Actions

- Bellman's equation for action-value function

$$Q^*(s,a) = E[r] + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q^*(s',a')$$

- For deterministic rewards and actions, it simplifies to

$$Q^*(s,a) = r + \gamma \max_{a'} Q^*(s',a')$$

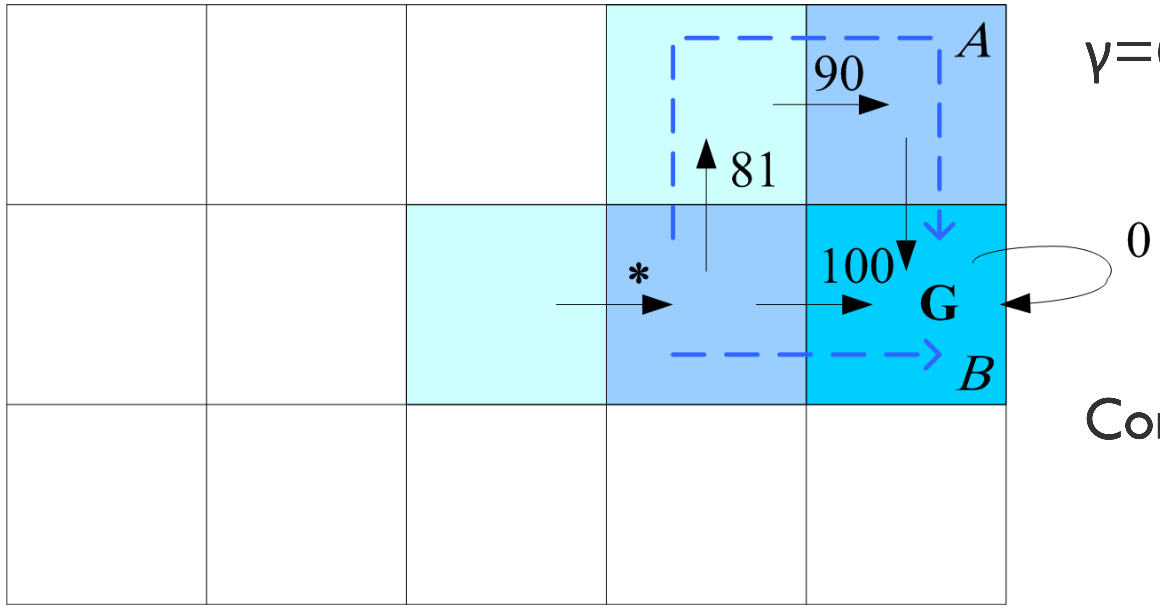
- Use above as update rule,

$$Q(s,a) \leftarrow r + \gamma \max_{a'} Q(s',a')$$

- Create a table $Q(s,a)$ with all entries set to zero
- Then explore to update the table
  - The "max" means explore all possible actions a'

Deterministic Case

- Starting at zero, entries of Q increase and never decreases



γ=0.9

Consider the value of action marked by ‘*’:

If path A is seen first, Q(*)=0.9*max(0,81)=73

Then B is seen, Q(*)=0.9*max(100,81)=90

Or,

If path B is seen first, Q(*)=0.9*max(100,0)=90

Then A is seen, Q(*)=0.9*max(100,81)=90

Q values increase but never decrease

Q-Learning for Nondeterministic Rewards and Actions

- Nondeterminism may be due to the randomness, or to an opponent
- In the deterministic case we directly update Q after each action

$$Q(s_t, a_t) \leftarrow r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$$

- Q-learning algorithm (Watkins, 1989) uses

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta \left(\boxed{\boxed{r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})} - Q(s_t, a_t)} \right)$$

- We keep running averages (expected values), instead of assignment
- Temporal difference between our current estimate value and the discounted value of next state and the reward received

Advantages of TD Prediction Methods

- Does not require model of the environment, i.e., transition and reward
- On-line and fully incremental
  - Update action-value function after each time step.
- TD methods are sound
  - Given fixed policy π, the value function is guaranteed to converge

SARSA

- Q-Learning algorithm is type of off-policy learning, since it does not refer to a specific policy in its update rule

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta\left(r_t + \gamma \boxed{\max_{a_{t+1}} Q(s_{t+1}, a_{t+1})} - Q(s_t, a_t)\right)$$

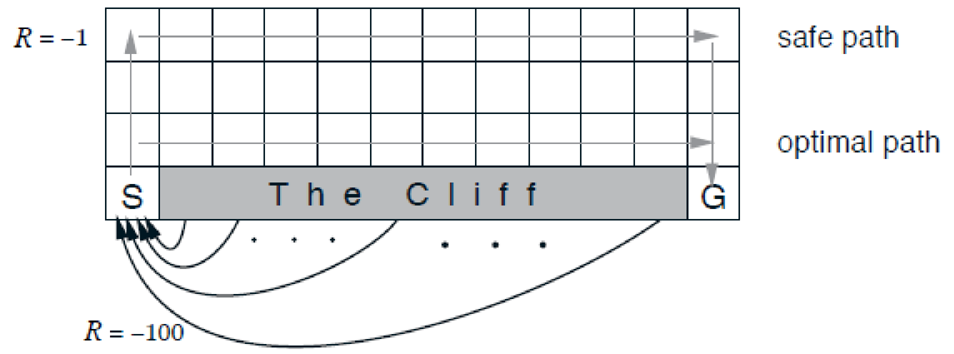
- SARSA is on-policy version

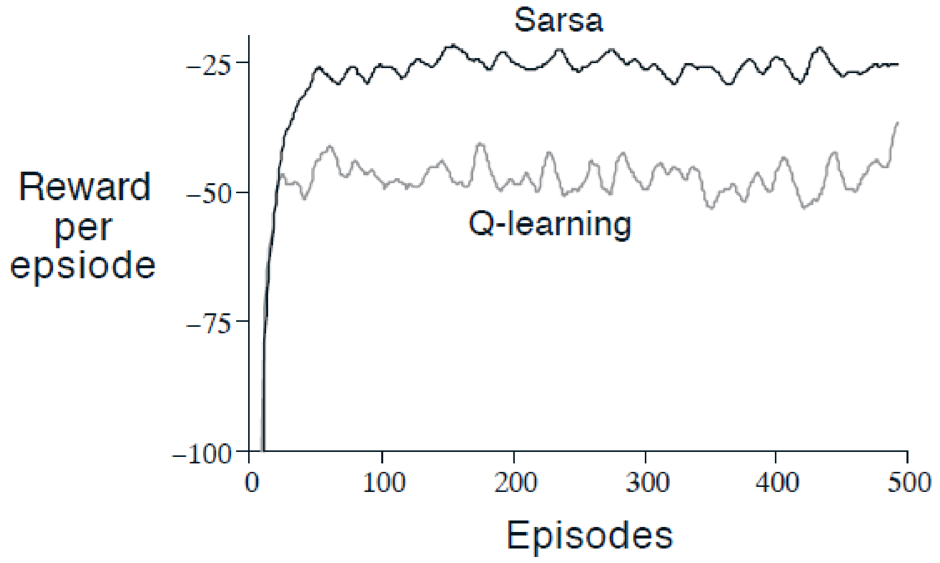
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta(r_t + \gamma \boxed{Q(s_{t+1}, a_{t+1})} - Q(s_t, a_t))$$

  - Instead “max” of all actions, select action based on policy $a_{t+1} \sim \pi(s_t)$, say using ϵ-greedy
- Q-Learning tends to explore more broadly
- While SARSA tends to explore actions more consistent with policy

Cliff Walking Comparison

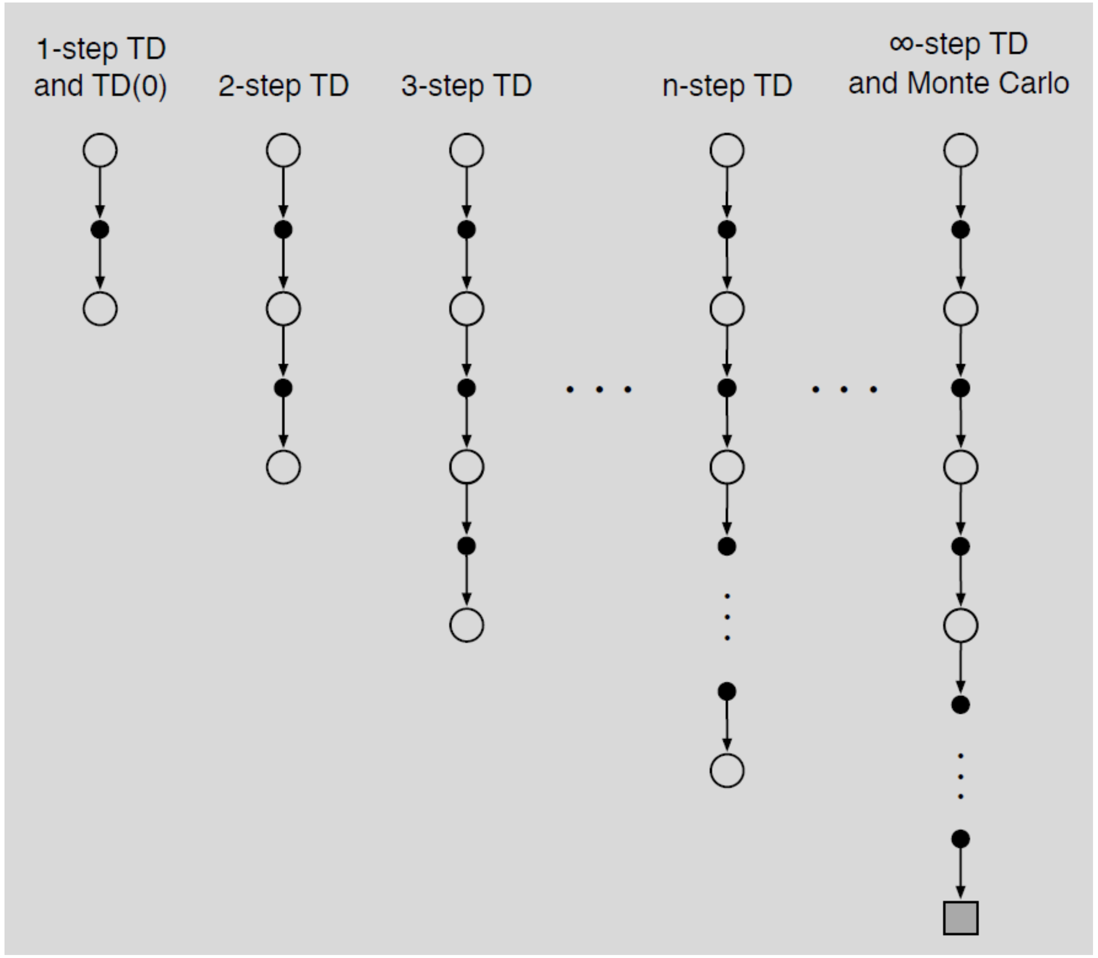
- Cliff walking
  - Agent start at S and most end at G
  - If the agent walks “off” the cliff, then restart at S
- SARSA finds the safe path
- Q-learning finds the optimal path
- On-line performance is worse for Q-learning, since it tends fall off the cliff





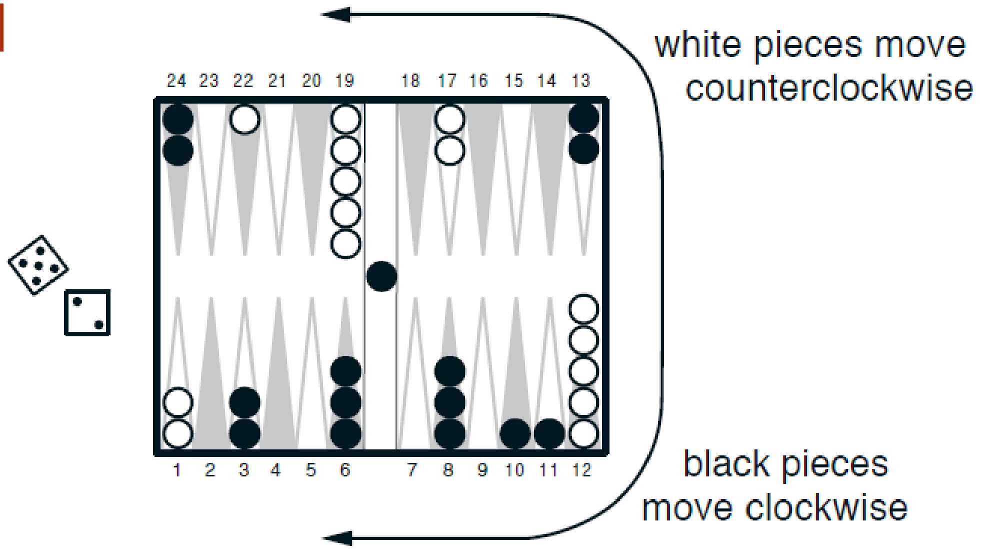
n-Step TD and Monte Carlo

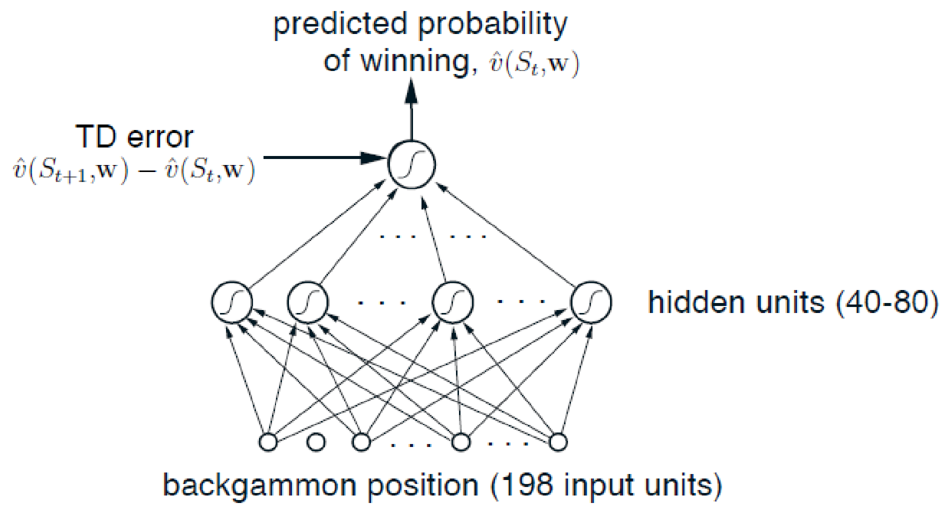
- Q-learning and SARSA algorithms described so far are called one-step TD methods
  - Value-action function $Q(s, a)$ updated after each step
- n-step TD methods updates after n steps
  - Explores deeper into the environment
  - Better updates for some types of problems
- Monte Carlo updates only after reaching the terminal state



TD-Gammon (Tesauro, 1995)

- Able to play at grandmaster level with little backgammon knowledge.
- Used TD(λ) algorithm, which uses an additional memory variable call eligibility trace
  - Eligibility trace stores how often and recent a state was visited
  - Recently/frequently visited states receive more reward
  - $\lambda = 0 \rightarrow \text{one-step}, \lambda = 1 \rightarrow \text{monte carlo}$
- Implemented value function as MLP with one hidden layer
- Trained on 1.5M games of self play


24 23 22 21 20 19 18 17 16 15 14 13
1 2 3 4 5 6 7 8 9 10 11 12
white pieces move counterclockwise
black pieces move clockwise


predicted probability of winning, $\hat{v}(S_t,\mathbf{w})$
TD error $\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w})$
hidden units (40-80)
backgammon position (198 input units)

Learning in Robotics

- See Kober et al 2013

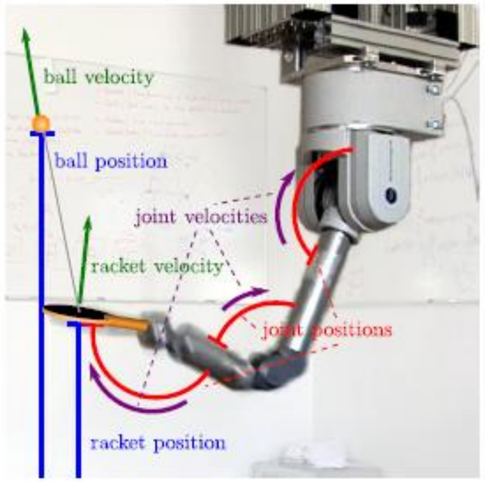
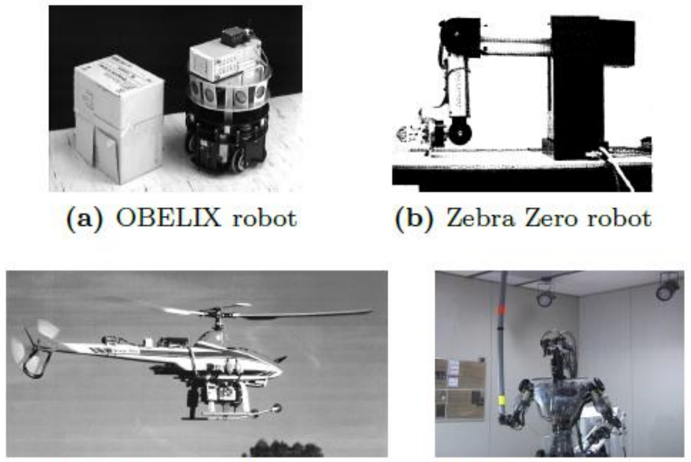


Figure 3: This Figure illustrates the state space used in the modeling of a robot reinforcement learning task of pad-



(a) OBELIX robot **(b)** Zebra Zero robot

(c) Autonomous helicopter **(d)** Sarcos humanoid DB

Figure 1: This figure illustrates a small sample of robots with behaviors that were reinforcement learned. These cover the whole range of aerial vehicles, robotic arms, autonomous vehicles, and humanoid robots. (a) The OBELIX robot is a wheeled mobile robot that learned to push boxes (Mahadevan and Connell, 1992) with a value function-based approach (Picture reprint with permission of Sridhar Mahadevan). (b) A Zebra Zero robot arm learned a peg-in-hole insertion task (Gullapalli et al., 1994) with a model-free policy gradient approach (Picture reprint with permission of Rod Grupen). (c) Carnegie Mellon's autonomous helicopter leveraged a model-based policy search approach to learn a robust flight controller (Bagnell and Schneider, 2001). (d) The Sarcos humanoid DB learned a pole-balancing task (Schaal, 1996) using forward models (Picture reprint with permission of Stefan Schaal).

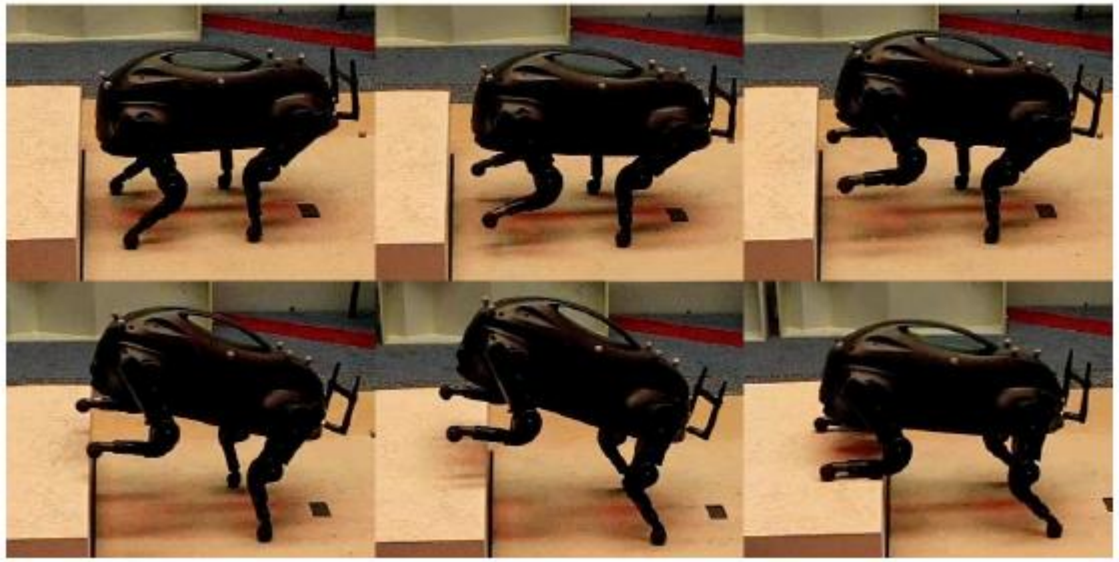
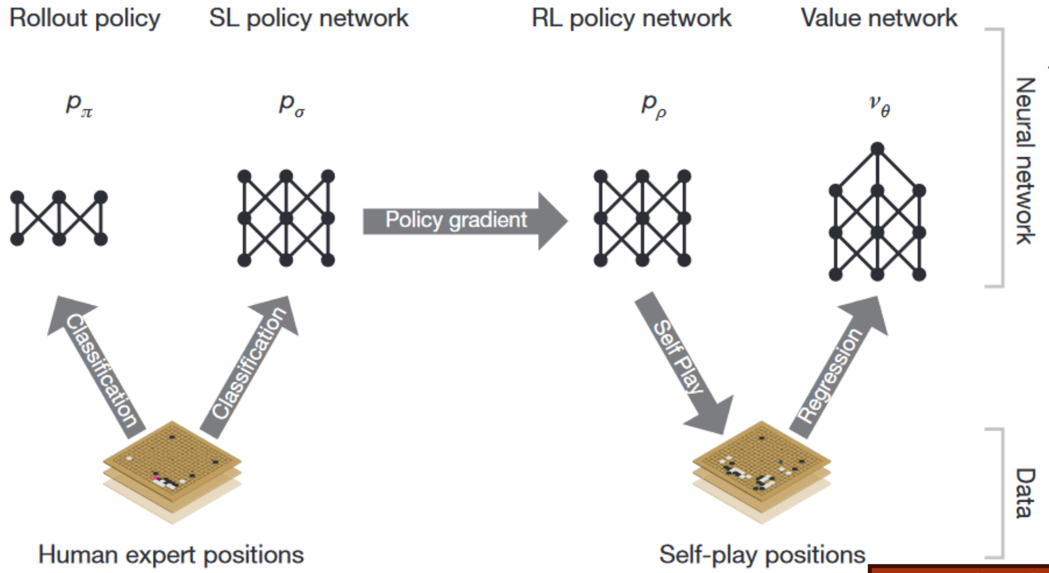


Figure 5: Boston Dynamics LittleDog jumping (Kolter and Ng, 2009a) (Picture reprint with permission of Zico Kolter).

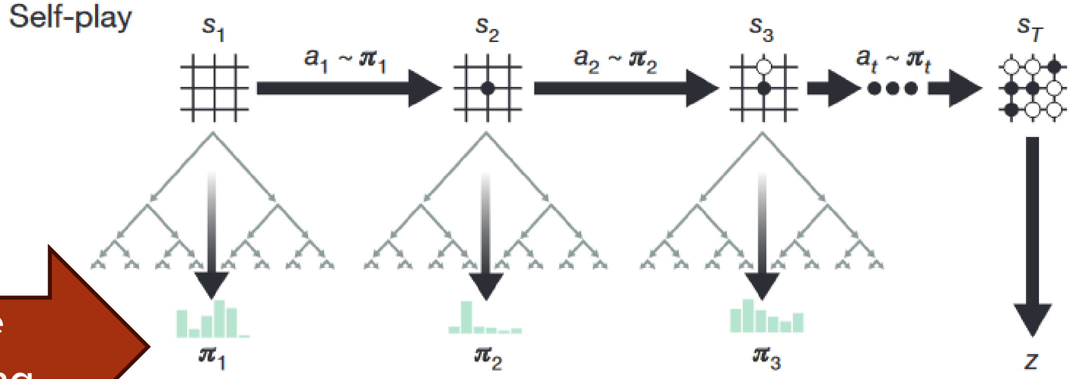
Reinforcement Learning: AlphaGo, AlphaGo Zero

- 2015: AlphaGo won Fan Hui, European champion
- 2016: AlphaGo won Lee Sedol, one of the strongest player
- 2017: AlphaGo won Ke Jie, the top-ranked player
- 2017: AlphaGo Zero beats AlphaGo 100 to 0 games



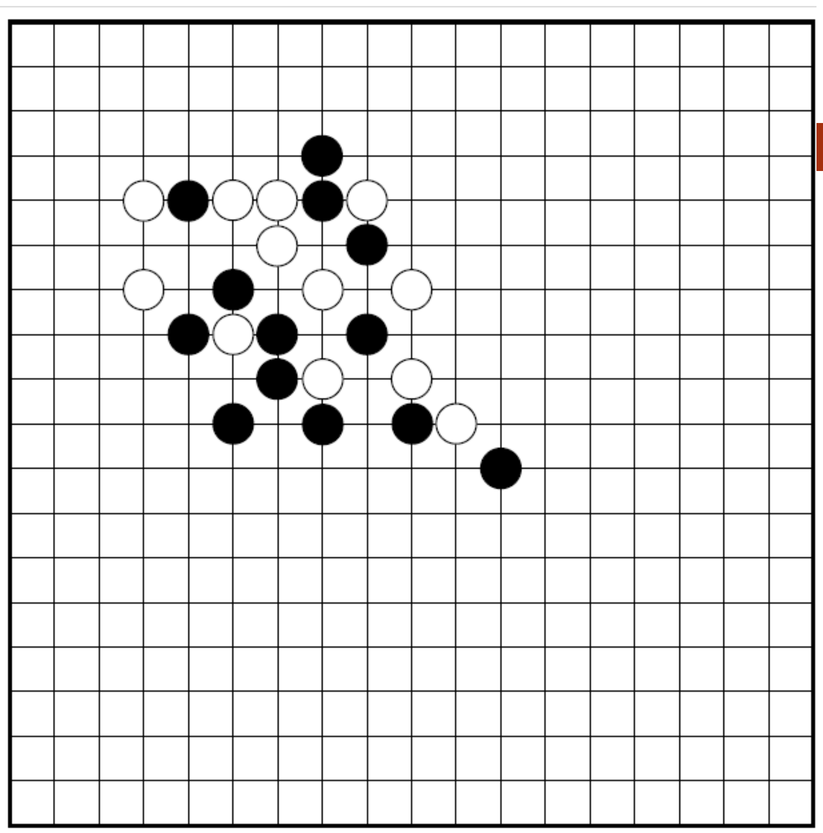
AlphaGo: Initiate with Supervised Learning

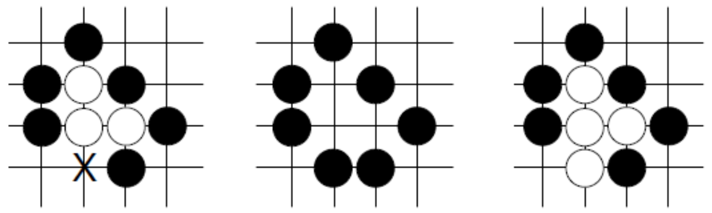
AlphaGo Zero: Pure Reinforcement Learning



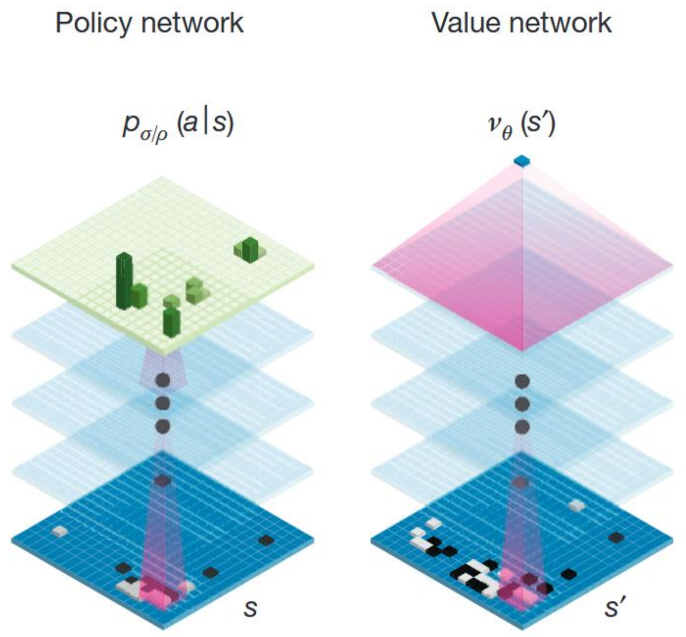
Game of Go

- Played on a 19x19 grid board
- One player plays with black stones; the other plays with white stones
- Players alternate turns by placing black and white stones at grid line intersections
- Goal of the game is to capture opponent stones by surrounding them





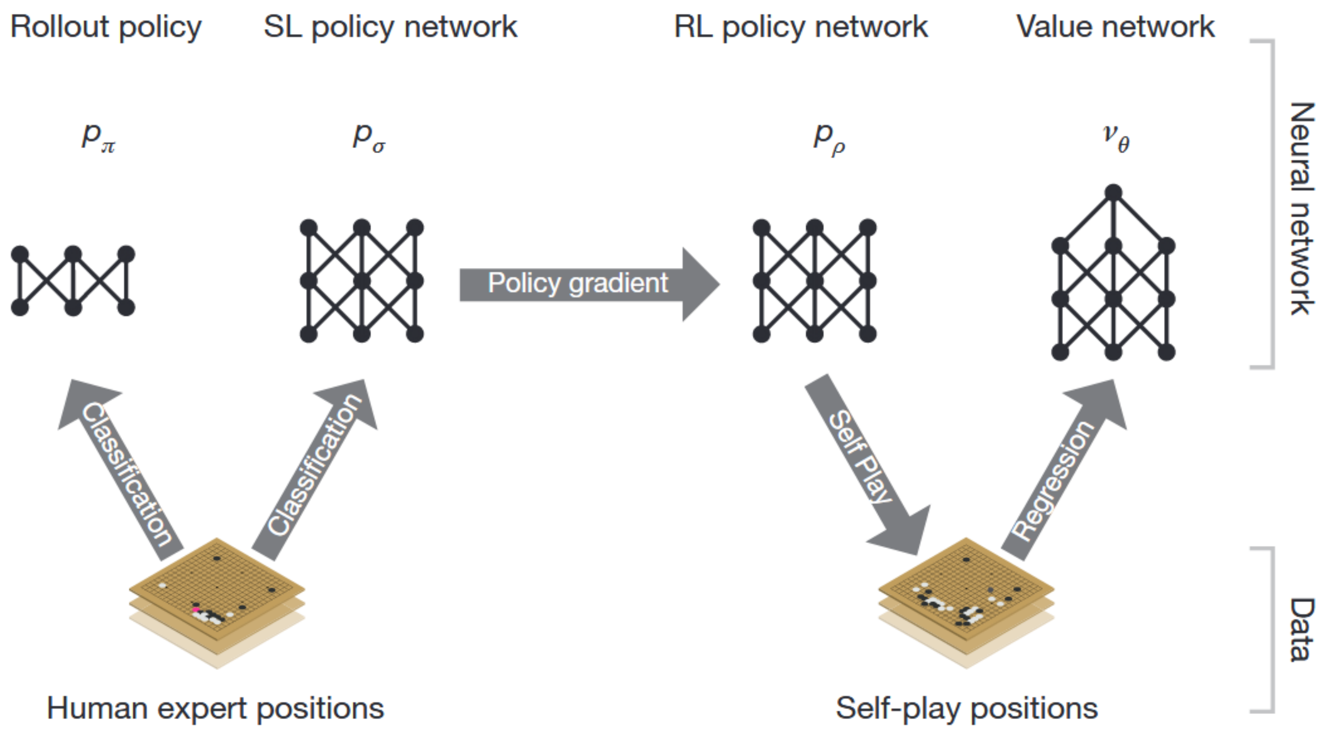
AlphaGo (Silver, et al 2017)



- Go is a perfect information game
- In theory can compute optimal value function $v^*(s)$
- But infeasible because search tree size is $\sim 250^{150}$
  - Go: branch factor~=250, depth~=150
  - Chess: branch factor~=35, depth~=80
- AlphaGo approach
  - Train CNN to approximate $v(s) \approx v^*(s)$ to alleviate depth problem
  - Train CNN to generate to policy function $\pi(\mathrm{a}|\mathrm{s})$ to alleviate breath problem
  - Use Monte Carlo tree search (MCTS) to search long paths (rollouts) into the search tree

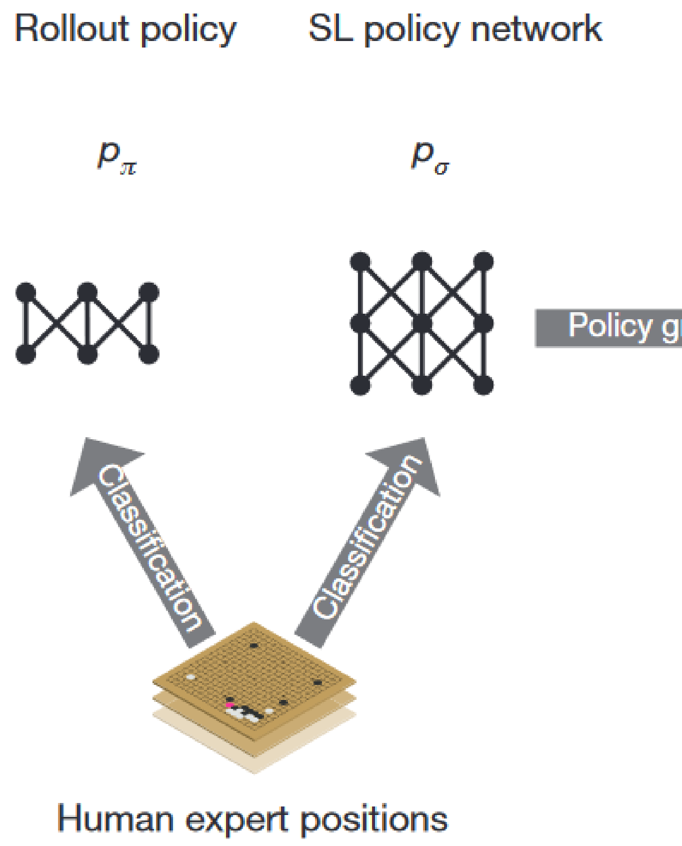
AlphaGo Training Overview

- Two datasets are used for training
  - Go games from human expert play
  - Go games from self-play
- First use supervised learning to train on human expert play data to find good “initial” values for the CNN weights
- Second use reinforcement learning to “tune” the values using self-play
- To avoid overfitting during the second stage, play against multiple version of itself



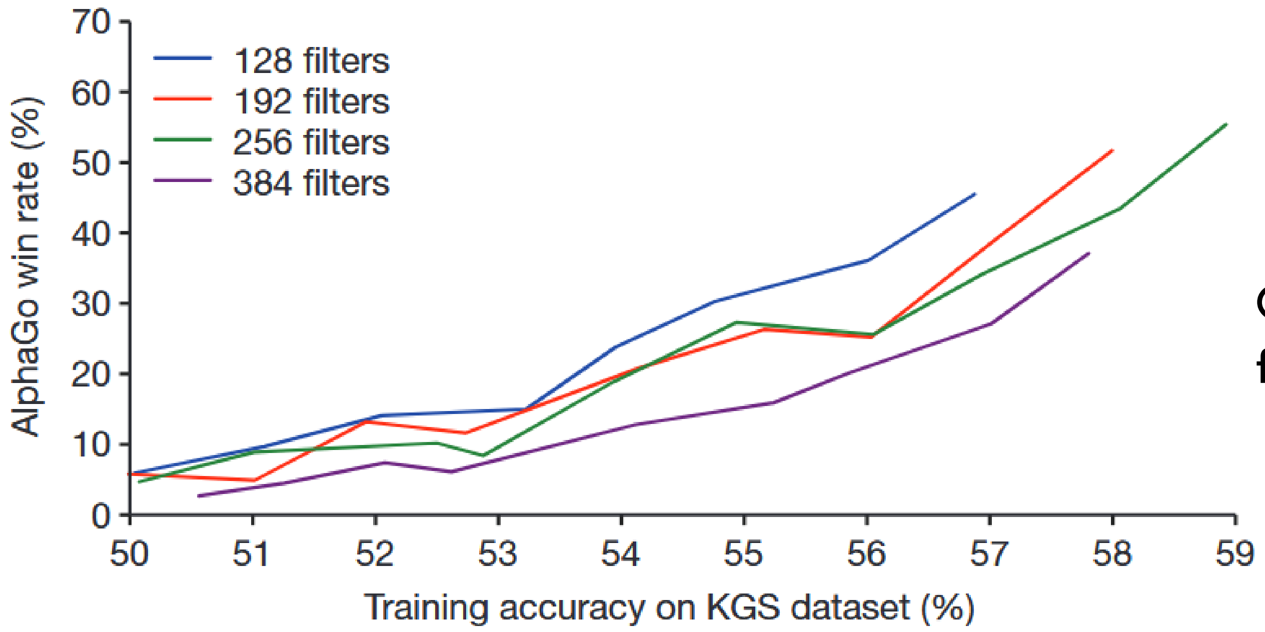
Supervised Learning of Policy Networks

- Train by random sampling state-action pairs (s, a)
- Sample board position and corresponding expert play
  - Dataset contains 30 million positions
- Train 2 policy networks p_σ and p_π
- Policy p_σ is 13-layer CNN with ReLU with
  - Input is raw board position + engineered features
  - Minimize cross entropy loss $p_\rho(a|s)$
  - 57.0% accuracy. With just raw board position 55.7% accuracy
  - 3 ms to select action
- Policy p_π is linear softmax over engineered features
  - 2 μs to select action with 24.2% accuracy
  - Used for fast MCTS rollouts.
  - Can rollout to the end of game before p_σ can select just one action



Training Accuracy vs Win Rates

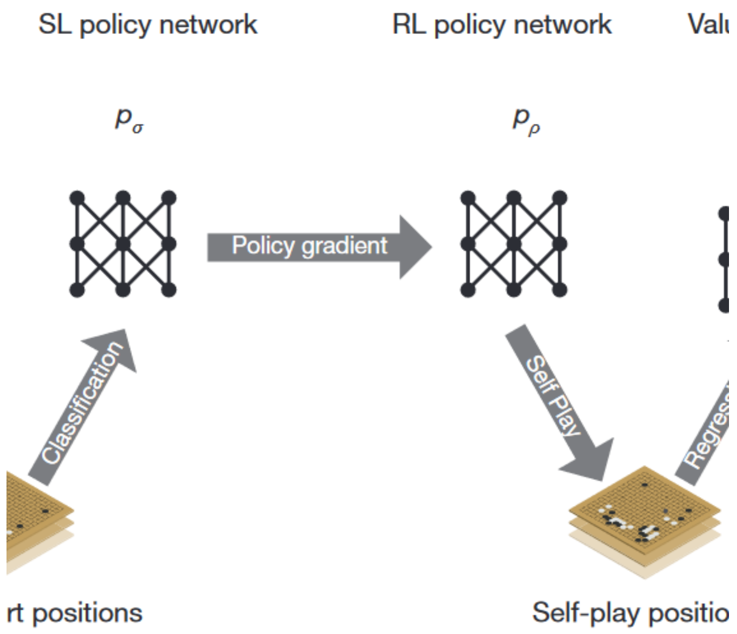
- Small improvements in training accuracy led to large improvements in win rates
  - If training accuracy ~50%, then win rate ~5%
  - If training accuracy ~58%, then win rate ~45%
- Using policy p_σ with no search able to win 85% games against best open-source Go program Pachi



Given same training accuracy, fewer filters are better !?!

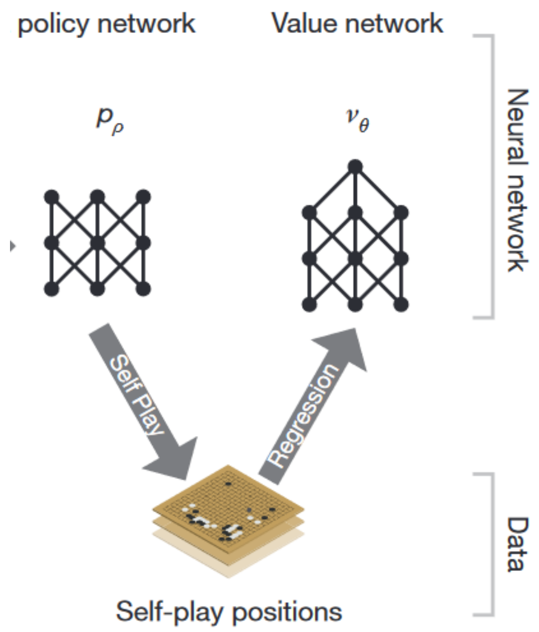
Reinforcement learning of policy networks

- Train policy network p_ρ using reinforcement learning
- Policy network p_ρ is a CNN
  - Same architecture as the supervised learning network p_σ
  - Initialed with the weights of p_σ
- Policy network p_ρ plays against randomly selected previous versions of itself
- Reward function
  - $r(s_t) = 0 \text{ for } t < T$, zero for all in all intermediate states
  - $r(s_T) = \pm 1$ for terminal state
- Games are played until termination then weights are updated
  - Update network using all steps of the play
  - Minimize $\log p_\rho(a_t|s_t) r(s_T)$



Reinforcement learning of value networks

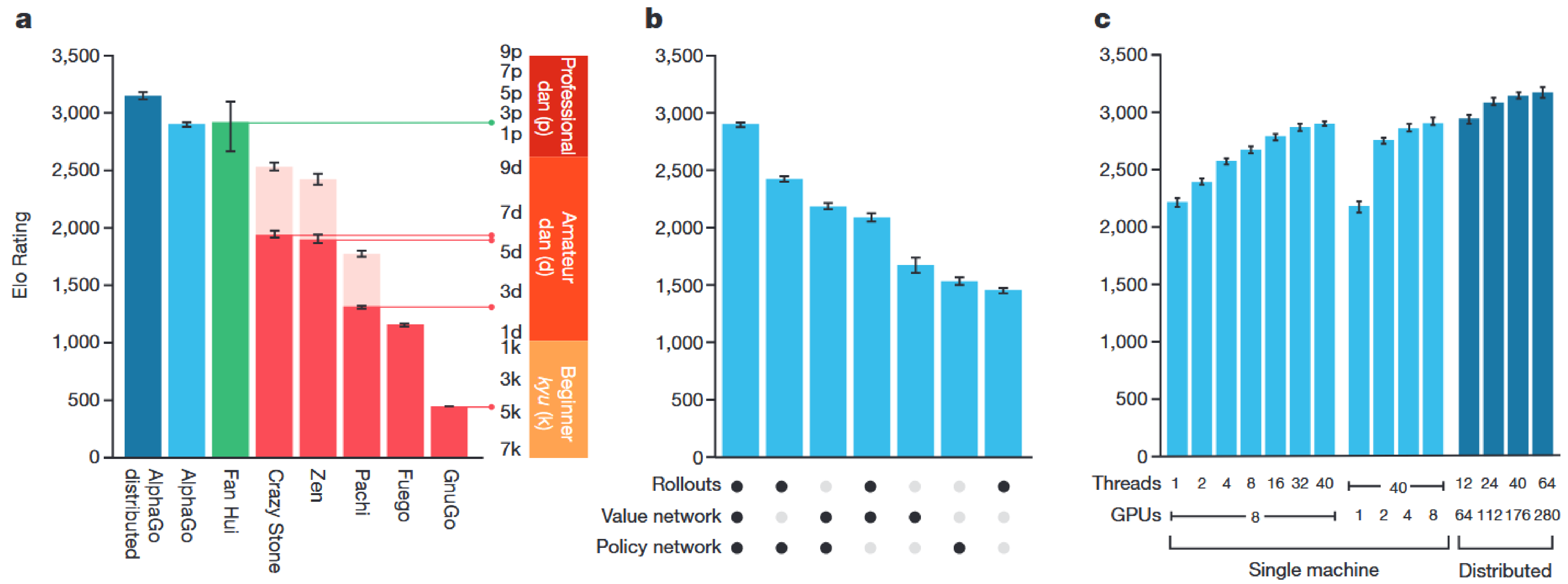
- Given a policy p, train value function $v^p(s_t)$ to predict the outcome $r(s_T)$ of board position s_t
- Training dataset consists of self-plays starting at position s_t
  - $s_t, a_t, a_{t+1}, \dots, a_T$
- Minimize mean squared error between $r(s_T)$ and $v^p(s_t)$
- Naively training using all states of games to train leads to overfitting
  - Difference between s_t and s_{t+1} is just one stone
  - Test MSE is 0.36, and train MSE is 0.19
- Instead sample 30 million positions from separate games
  - Test MSE is 0.234 and train MSE is 0.226



Searching with policy and value networks

- Search procedures combines policy and value networks MCTS
- At each step expand the search tree by selecting the action
$$a_t = \underset{a}{\operatorname{argmax}}(Q(s_t, a) + u(s_t, a))$$
  - Q is the action-value function
  - u is action-value of supervised learning version of action-value function divided by 1 + # of visits, i.e., decreasing importance $\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \dots$
- Value of an expand leaf node s_L is a weight average of RL value function $v_\theta(s_L)$ and the final board position of MCTS rollout $z_L = \pm 1$
$$V(s_L) = (1 - \lambda) v_\theta(s_L) + \lambda z_L$$
- See paper for more details

Evaluation



a) Evaluation against other players ; b) ablation study; c) Compute resource