Playing Games

John Canny

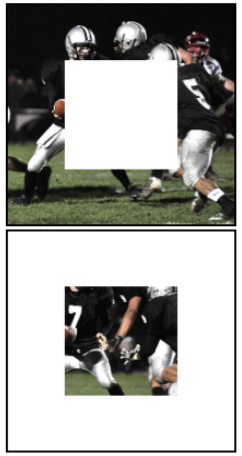
Spring 2020

Lecture 25 of CS182/282A: Designing, Visualizing and Understanding Deep Neural Networks

Last Time: What is Self-Supervised Learning?

- A version of unsupervised learning where data provides the supervision.
- In general, withhold some part of the data and the task a neural network to predict it from the remaining parts.
- Details decide what proxy loss or pretext task the network tries to solve, and depending on the quality of the task, good semantic features can be obtained without actual labels.

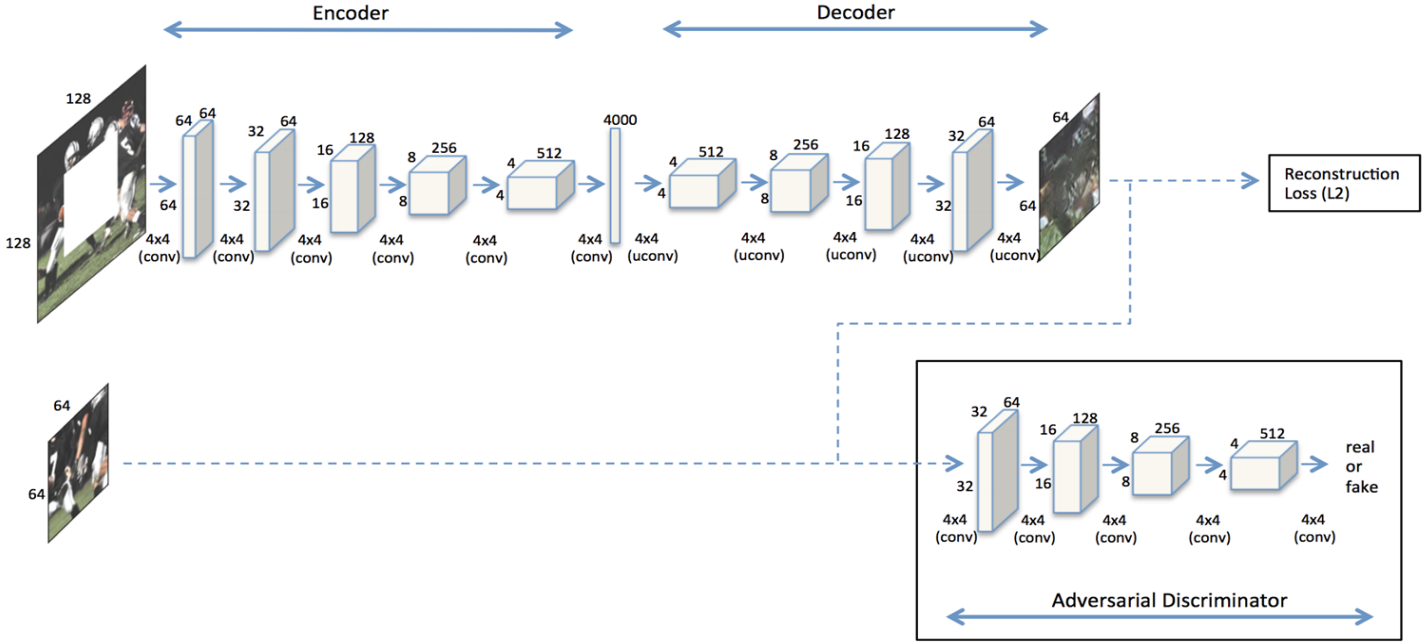
Last Time: Context Encoders



(a) Central region

(b) Random block

(c) Random region

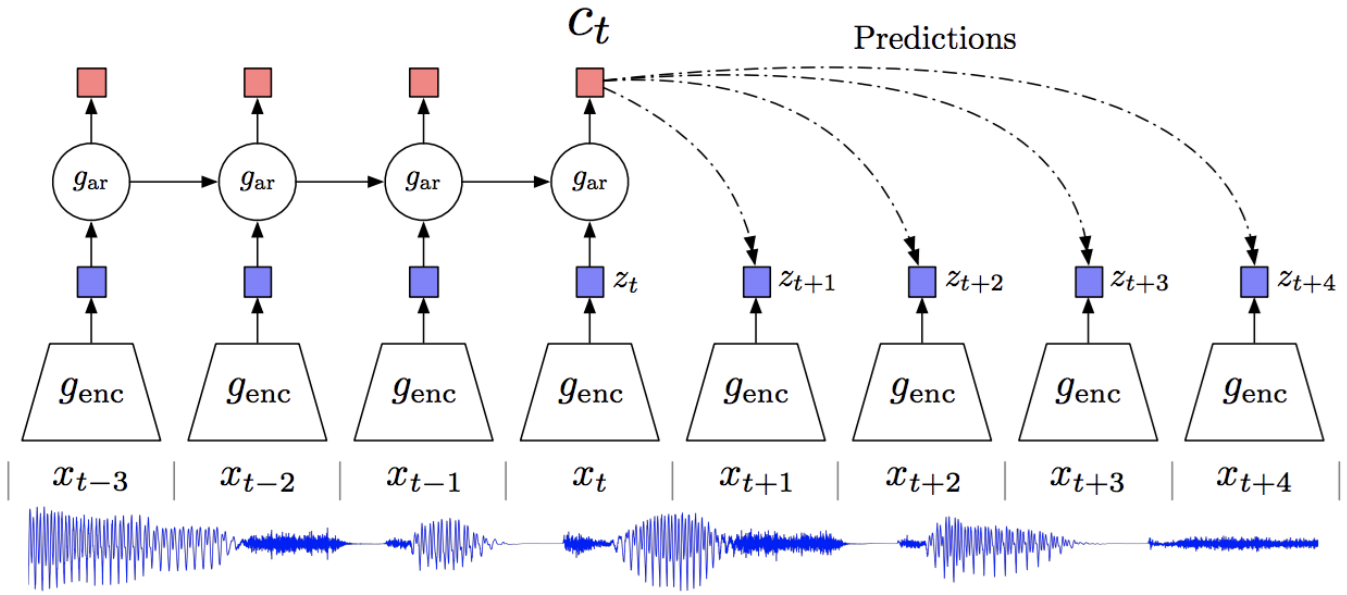


Pathak et al 2016

Last Time: Contrastive Methods

- Simultaneously maximize probability of positive examples while minimizing probability of negative examples
- Positive examples come from real data, negative examples often easy to create by sampling
- Examples: word2vec, CPC (Contrastive Predictive Coding).

Last Time: Contrastive Predictive Coding

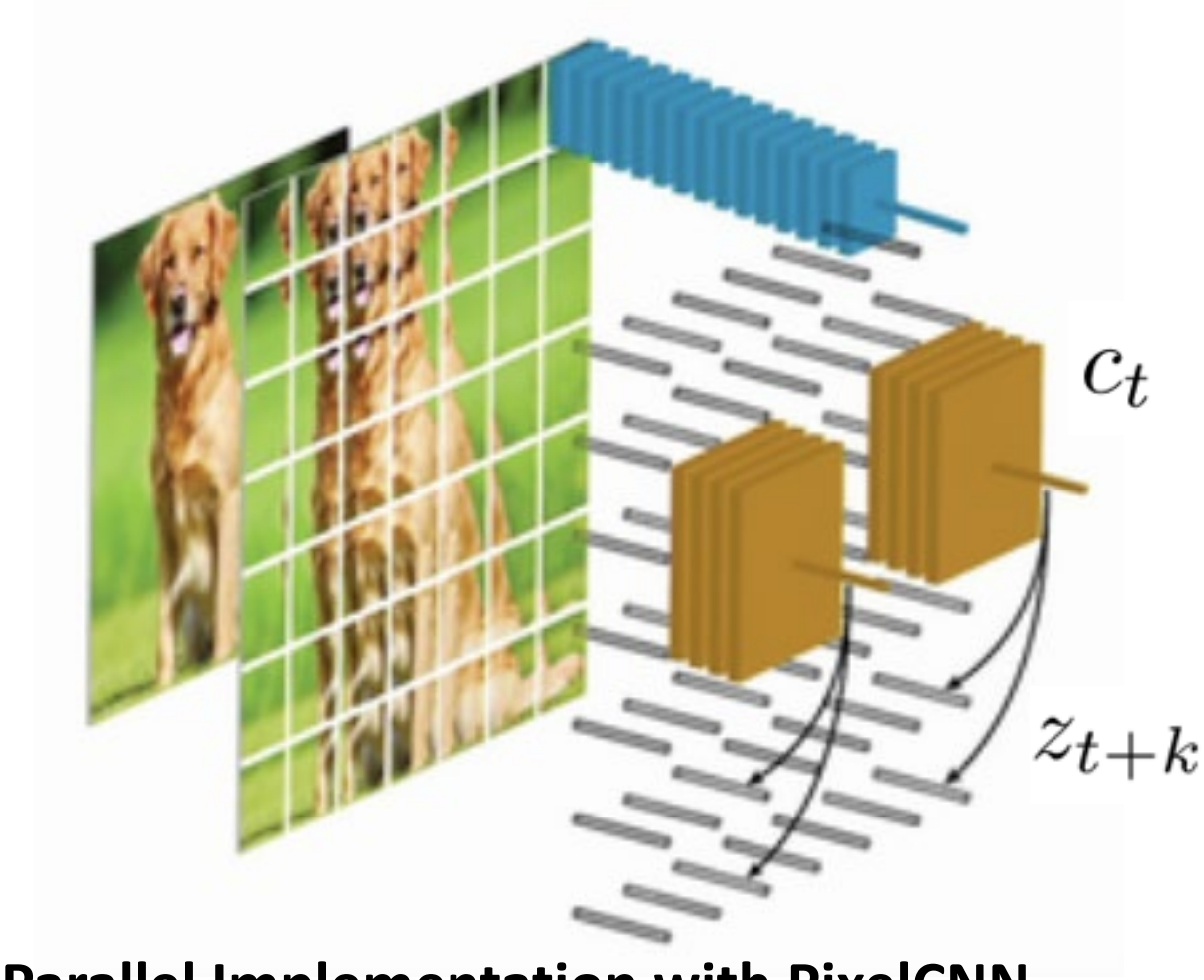


$$f_k(x_{t+k}, c_t) = \exp\left(z_{t+k}^T W_k c_t\right)$$

$$\mathcal{L}_N = -\mathop{\mathbb{E}}_X\left[\log \frac{f_k(x_{t+k}, c_t)}{\sum_{x_j \in X} f_k(x_j, c_t)}\right]$$

Figure from Alex Graves

Last Time: CPCv2 - Large Scale CPC on ImageNet



Parallel Implementation with PixelCNN (masked conv) and 1x1 conv

Figure from Aaron Van den Oord

InfoNCE Loss

$$f_k(x_{t+k}, c_t) = \exp\left(z_{t+k}^T W_k c_t\right)$$

$$\mathcal{L}_N = -\mathop{\mathbb{E}}_{X}\left[\log \frac{f_k(x_{t+k}, c_t)}{\sum_{x_j \in X} f_k(x_j, c_t)}\right]$$

Negatives

1. Other patches within image
2. Patches from other images

Course Logistics

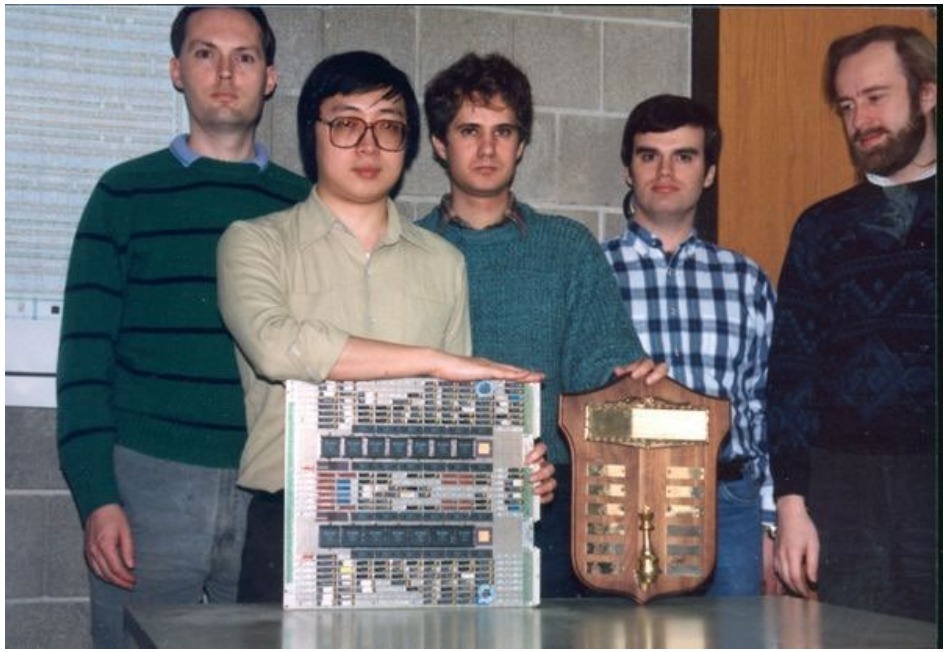
- Quiz 2 is out, due on Thursday.

- Course survey is open now, please fill it out.

A Brief History of Game-Playing: Chess

Deep Thought (1989): CMU project. Custom hardware, brute force alpha-beta search, around 1 billion positions per move.

Achieved grandmaster status.



Deep Blue (1996): IBM hired Deep Thought team from CMU. Designed custom chips for alpha-beta search. Around 50 billion evaluations per move.

Defeated world champion Gary Kasparov in 1997.

A Brief History of Game-Playing: Chess

Deep Fritz (2006): A program for standard PC hardware, defeats world champion Vladimir Kramnik.

Stockfish (2013-2018): Open-source program, wins most computer chess championships. Still based on alpha-beta search, but with many enhancements.

In Dec 2017, Stockfish was defeated by AlphaZero, an MCTS engine 25/3/72 (W/L/D).

Jan 2019: An open-source version of AlphaZero called Leela Chess Zero defeats Stockfish.

A Brief History of Game-Playing: Backgammon

TD-Gammon (1991-) was developed using TD-learning and neural networks for board evaluation.

By using reinforcement learning (Temporal Difference Learning), TD-Gammon was able to improve by self-play.

It used only a 2-ply search on top of its neural board evaluation.

Nevertheless by 1993 (after 1.5 million training games) it was competitive with the best human players.

Subsequent Backgammon programs dominate human players.

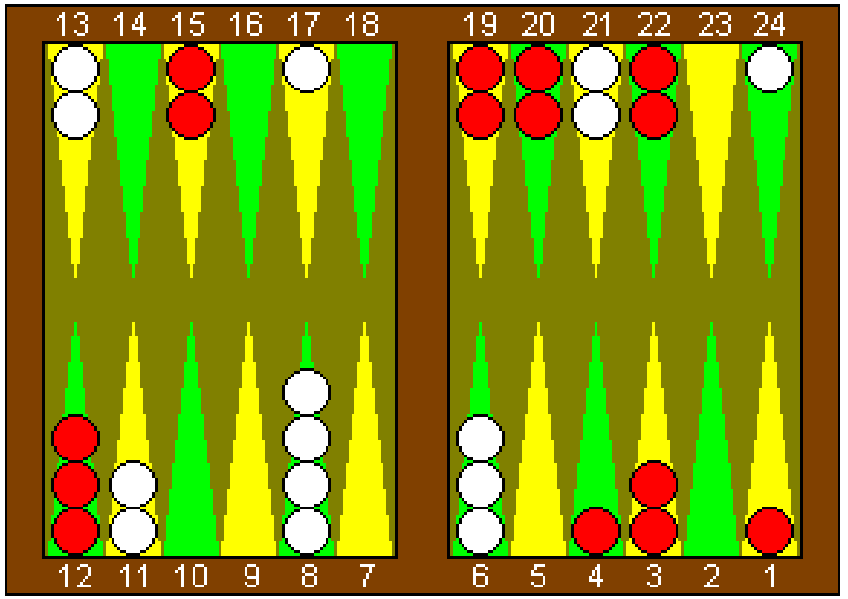


Figure 3. A complex situation where TD-Gammon's positional judgment is apparently superior to traditional expert thinking. White is to play 4-4. The obvious human play is 8-4*, 8-4, 11-7, 11-7. (The asterisk denotes that an opponent checker has been hit.) However, TD-Gammon's choice is the surprising 8-4*, 8-4, 21-17, 21-17! TD-Gammon's analysis of the two plays is given in Table 3.

A Brief History of Game-Playing: Go

Pre-2014: No Go program achieves higher than Amateur Dan (1-7d).

In 2014, a program called “Crazy Stone” (MCTS) plays a 6d professional player and loses, but wins a match.

In 2015, AlphaGo from DeepMind (MCTS) defeats a professional Go player (Fan Hui, 2-dan professional) on a 19x19 board without handicap.

It used both human and machine training.

A Brief History of Game-Playing: Go

In 2016, Deep Mind's AlphaGo (MCTS) defeats former world champion Lee Sedol.

2017: AlphaGo Master defeats current world champion Ke Jie (also beat a 5-person team).

Commentators noted that Ke appeared to borrow moves from AlphaGo 2016.

But Ke noted that "AlphaGo is improving too fast" and is "a different player from last year."

A Brief History of Game-Playing: Multi-Game

In 2017, AlphaGo Zero was introduced which used no human training data.

Using MCTS and RL only, it surpassed the performance of AlphaGo in 3 days, AlphaGo Master in 21 days, and all previous versions in 40 days.

A more general program called AlphaZero was introduced in 2018, which played Chess, Shogi and Go. It is the most powerful player of each, except for AlphaGo Zero.

Outline

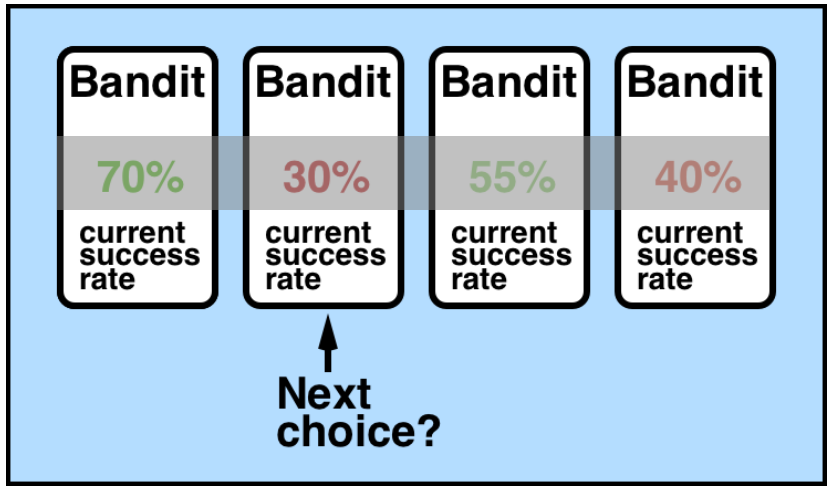
- Bandits and the explore-exploit tradeoff
- Regret and Upper Confidence Bounds
- Monte-Carlo Tree Search
- AlphaGo Zero
- AlphaZero

Bandits

A ***multi-armed bandit*** is a random process with unknown but fixed success probabilities.

It's the simplest system to expose the exploration/exploitation trade-off.

Suppose a set of poker machines have these payout probabilities – which are ***unknown*** to the player:

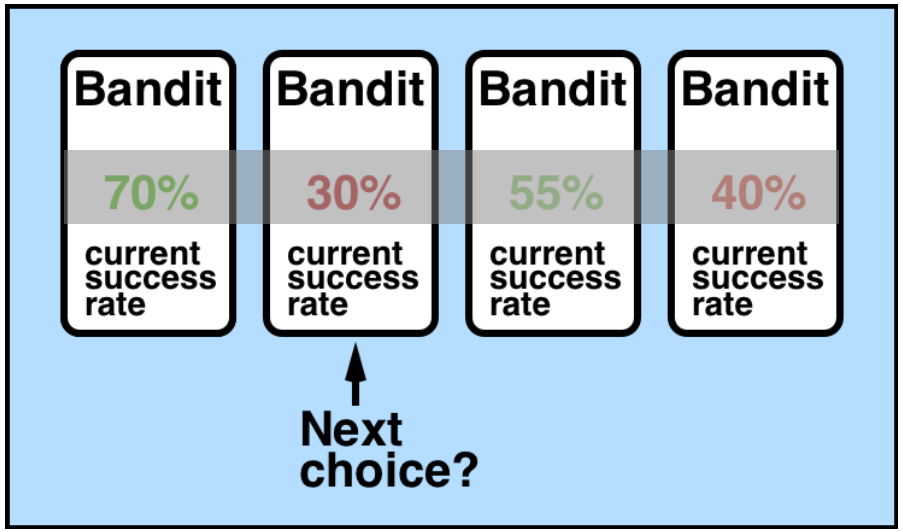


The player gets to chose which machine to play next, gets a reward (or nothing) and continues to the next machine.

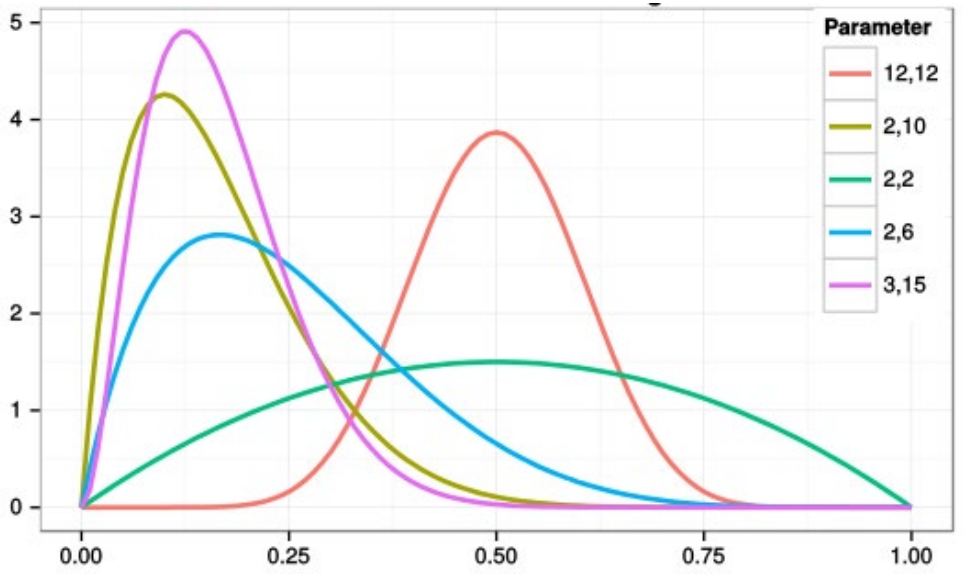
Bandits

The player can estimate the success probabilities for each machine based on their payout history, and should choose the best machine each time.

But what is “best” here?



Assuming the machines are stationary (fixed probabilities), the outcome is a ***Bernoulli*** distribution, and the posterior on the success probability is a ***β-distribution***.



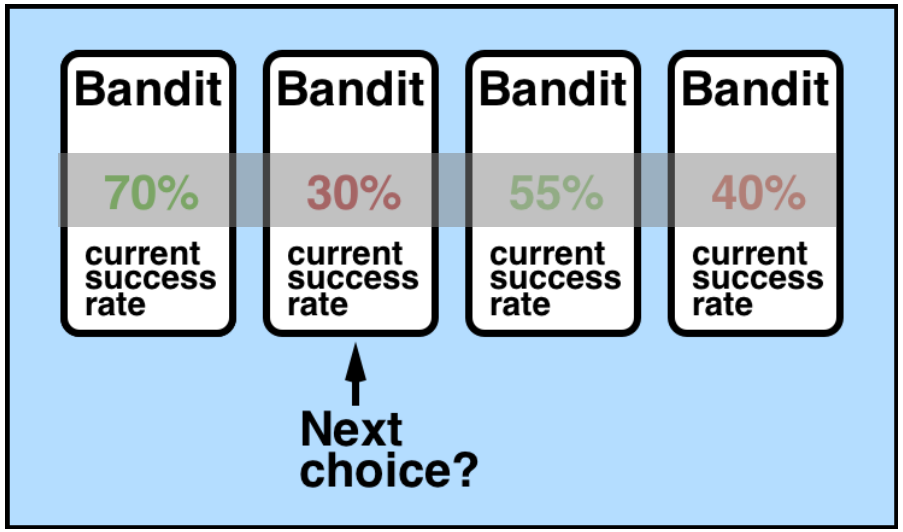
Bandits

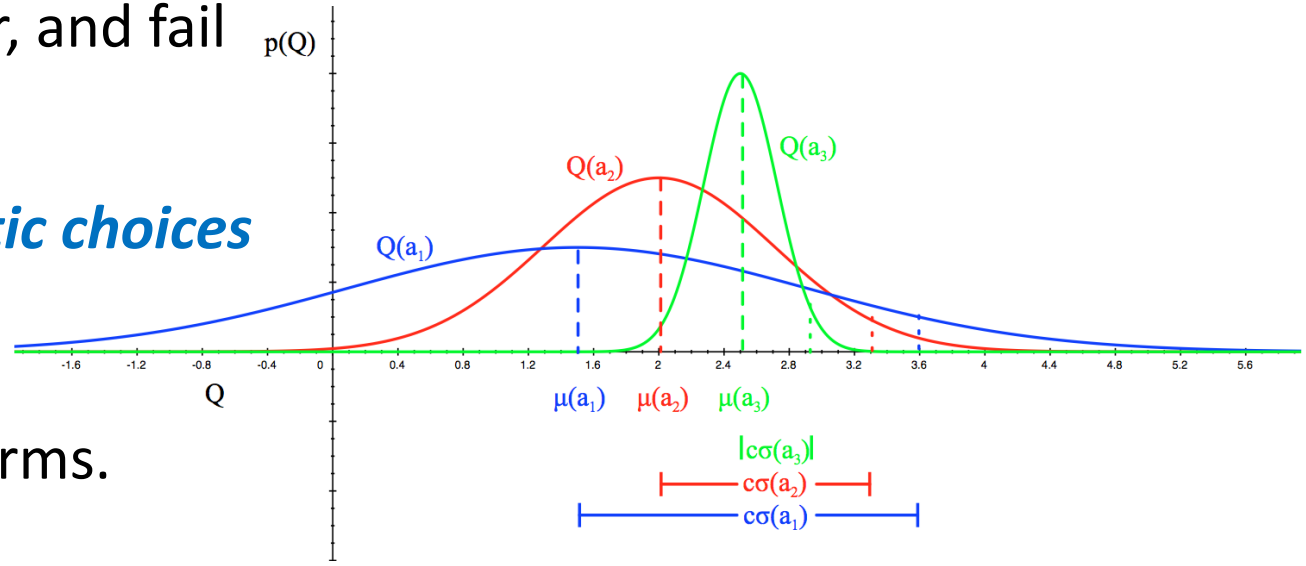
Bandits can be modeled as simplified MDPs (Markov Decision Processes). There are no states, only actions and rewards.

The goal is to maximize expected reward – almost.

Just as in RL, we have an explore-exploit tradeoff. If we use expected value based on trials so far, we would always choose the best arm so far, and fail to find better alternatives.

If we instead make ***reasonable, optimistic choices*** we will either succeed and find a better option, or fail and better quantify the rewards from one of the lesser-known arms.





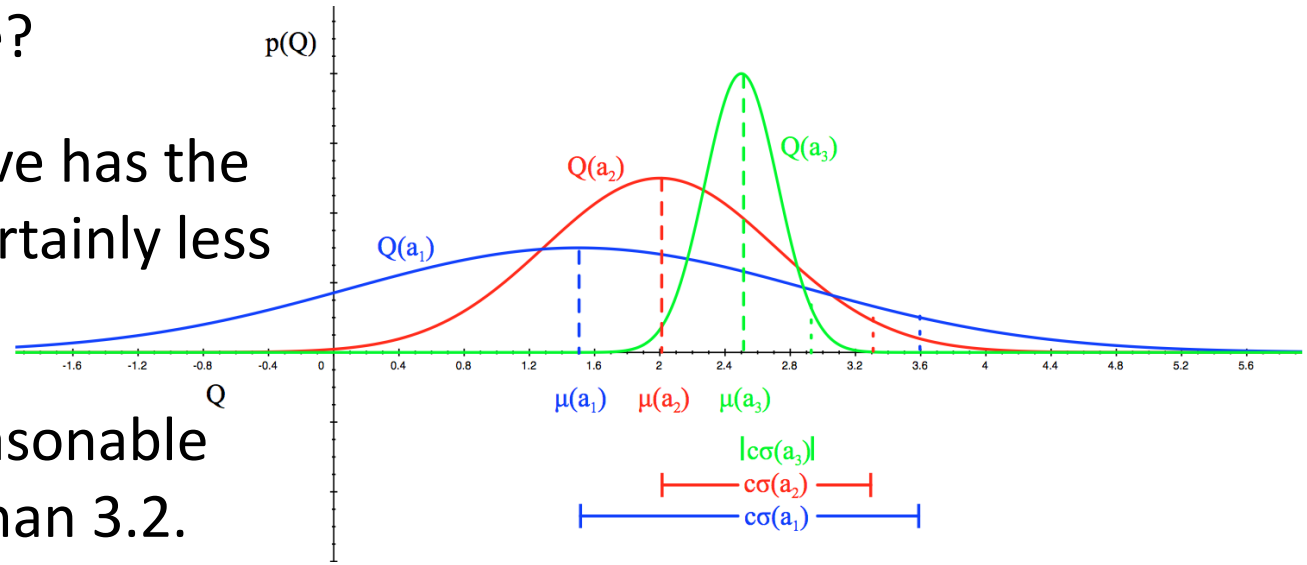
Upper Confidence Bounds (UCBs)

So we ask: where ***might*** the best arm be?

For the curve at the right, the green curve has the highest mean reward, but it is almost certainly less than 3.2.

The red curve has lower mean, but a reasonable chance (at least 5%) of a better return than 3.2.

The blue curve has the lowest mean reward, but the highest reward at a confidence level of 95%.



We have little to lose (only the cost of looking) to try the riskier alternatives. We may fail to find better rewards, but if so we will narrow those curves and ignore them later.

Regret

Where ***might*** the ***best*** arm be?

Assume we play various arms using some policy $\pi(t)$, and let $T_j(n)$ be the number of times we played arm j in the first n steps.

The ***Regret*** of π after n steps is defined as:

$$R_n = \mu_{max} n - \sum_{j=1}^{K} \mu_j E\left[T_j(n)\right] = \sum_{j=1}^{K} \Delta_j E\left[T_j(n)\right]$$

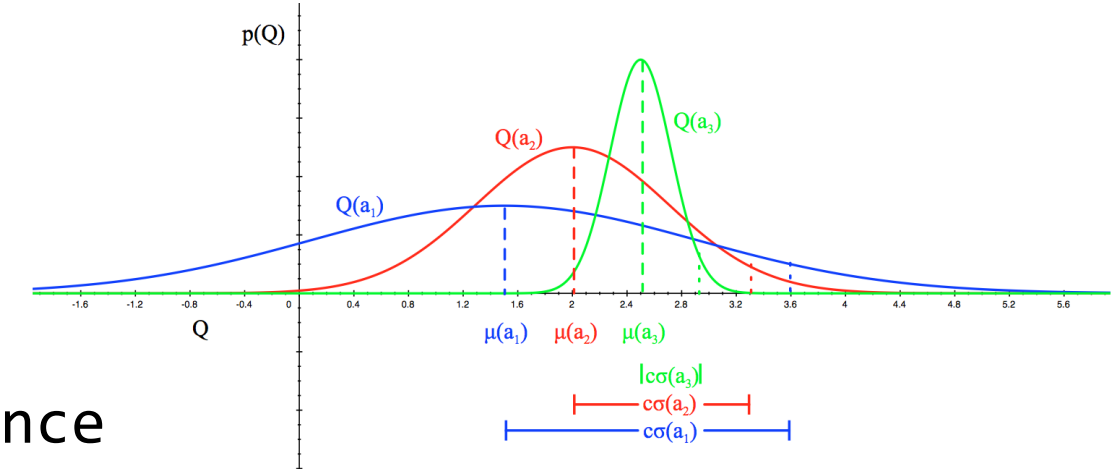
where μ_j is the expected reward of arm j, μ_{max} is the maximum reward $\max_i \mu_i$, and $\Delta_i = \mu_{max} - \mu_i$.

Regret lets us measure the explore/exploit performance of some policy.

An oracle could simply play the best arm ($R = 0$), but any practical policy has to try different arms to find the best one, so $R > 0$.

Upper Confidence Bounds: UCB1

Finite-time Analysis of the Multiarmed Bandit Problem* Auer et al. 2002.



UCB1 is a policy with provably bounded regret:

```
Policy: UCB1
Initialization: Play each machine once
Loop:
    Play machine j that maximizes
```

$$\bar{x}_j + \sqrt{\frac{2 \ln n}{n_j}}$$

Empirical mean

Approx spread

where $\bar{x}_j$ is the reward so far from machine j, n_j is the number of times machine j has been played so far, and n is the total number of plays.

UCB1 Regret

Finite-time Analysis of the Multiarmed Bandit Problem* Auer et al. 2002.

For UCB1 running on K bandits with reward distributions $P_1, \ldots, P_k$, then its expected regret after n steps is at most:

$$\left[8 \sum_{i:\mu_i<\mu_{max}} \left(\frac{\ln n}{\Delta_i}\right)\right] + \left(1 + \frac{\pi^2}{3}\right)\left(\sum_{j=1}^{K} \Delta_j\right)$$

Fixed regret term

Grows logarithmically with n:

Where $\Delta_i = \mu_{max} - \mu_i$, and μ_i is the expected value of P_i.

This value is within a constant factor of optimal (lowest possible) regret growth.

Keep in mind that the ideal reward is growing linearly as $n\mu_{max}$, so the actual reward $r(n)$ satisifies $\lim_{n\to\infty} \frac{r(n)}{n\mu_{max}} = 1$

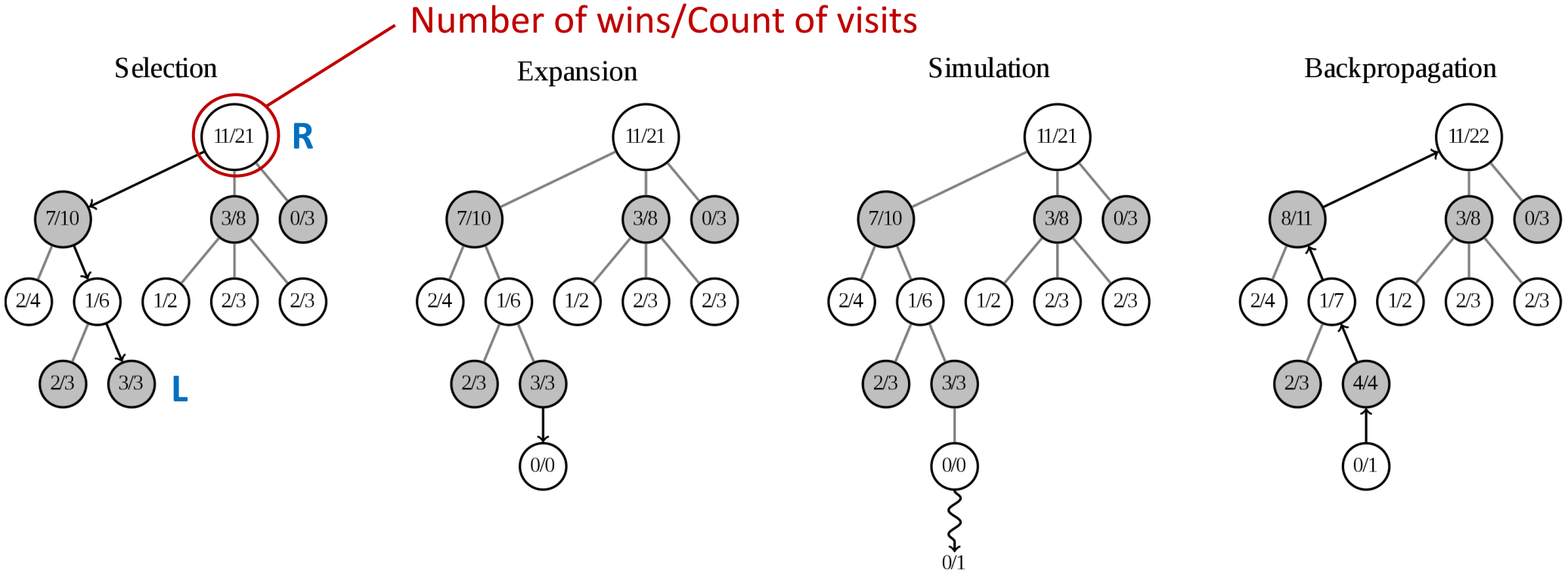
Multiplayer Games

Extend MDPs with player states:

- A set S of states with initial state s_0.
- A set S_T of terminal states.
- A the set of actions.
- $f: A \times S \to S$ the state transition function.
- $R: S \to \mathbb{R}^k$ the utility function ($\mathbb{R}^k$ is for complex/per-player rewards, usually its just $\mathbb{R}$)
- $n \in \mathbb{N}$ the number of players.
- $\rho: S \to \{1, \dots, n\}$ the player about to act in the given state.

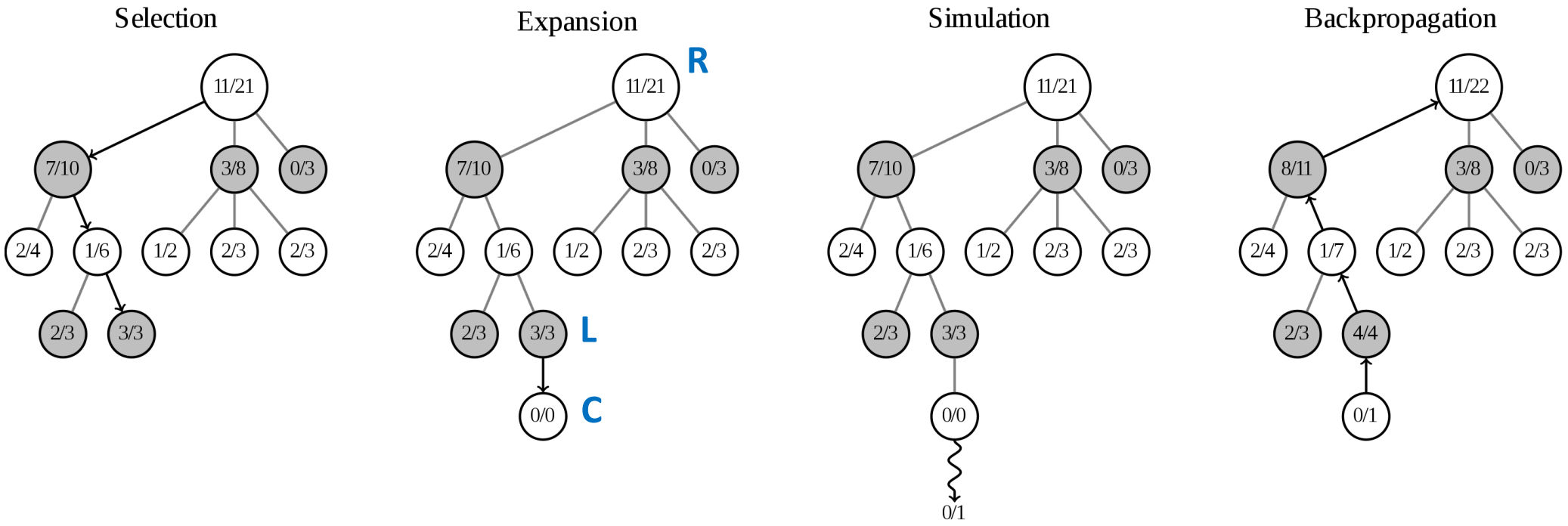
On their turns, each player i from $1, \dots, n$ plays with their own policy $\pi_i: A \times S \to \mathbb{R}$, where $\pi_i(a, s)$ defines a probability distribution over actions.

Monte-Carlo Tree Search



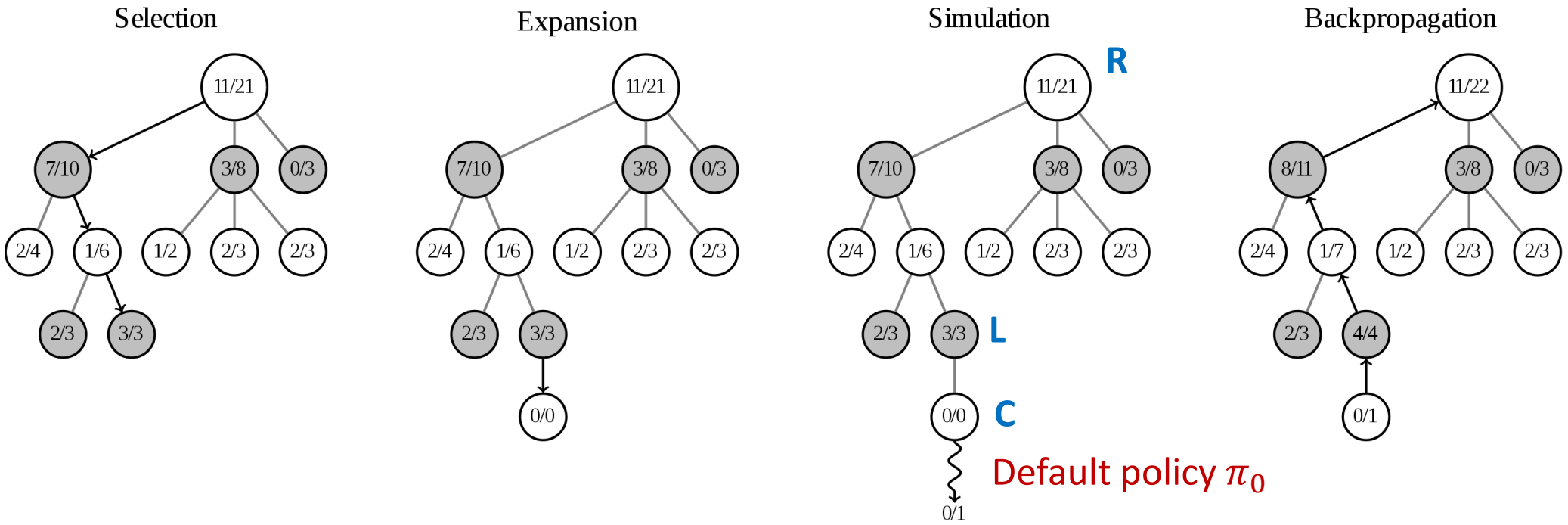
Selection: start from root *R* and select successive child nodes until a leaf node *L* is reached. The root is the current game state and a leaf is any node from which no simulation (playout) has yet been initiated.

Monte-Carlo Tree Search



***Expansion*:** unless the leaf *L* ends the game decisively (e.g. win/loss/draw) for either player, create one (or more) child nodes and choose node *C* from one of them. Child nodes are any valid moves from the game position defined by *L*.

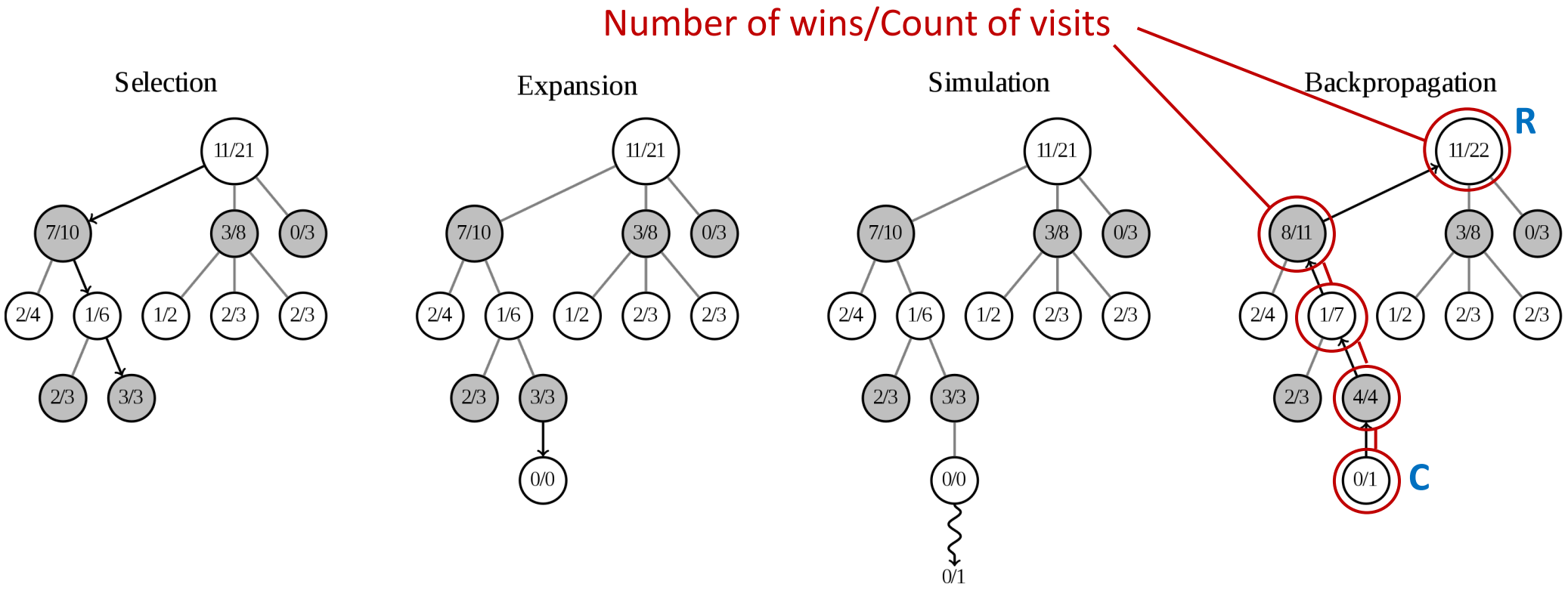
Monte-Carlo Tree Search



Simulation*:** complete one random playout from node *C*. This step is sometimes also called playout or rollout. The playout is typically performed with a ***default policy π_0.

A playout may be as simple as choosing uniform random moves until the game is decided (for example in chess, the game is won, lost, or drawn).

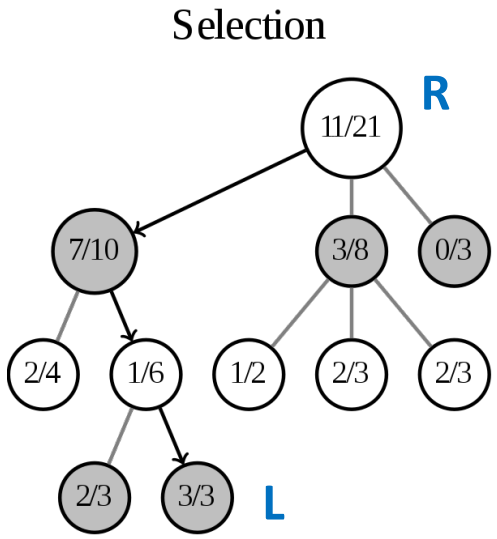
Monte-Carlo Tree Search



***Backpropagation*:** use the result of the playout to update information in the nodes on the path from *C* to *R*. For 1-0 (win-loss games), that means updating the win-loss counts of all the ancestor nodes.

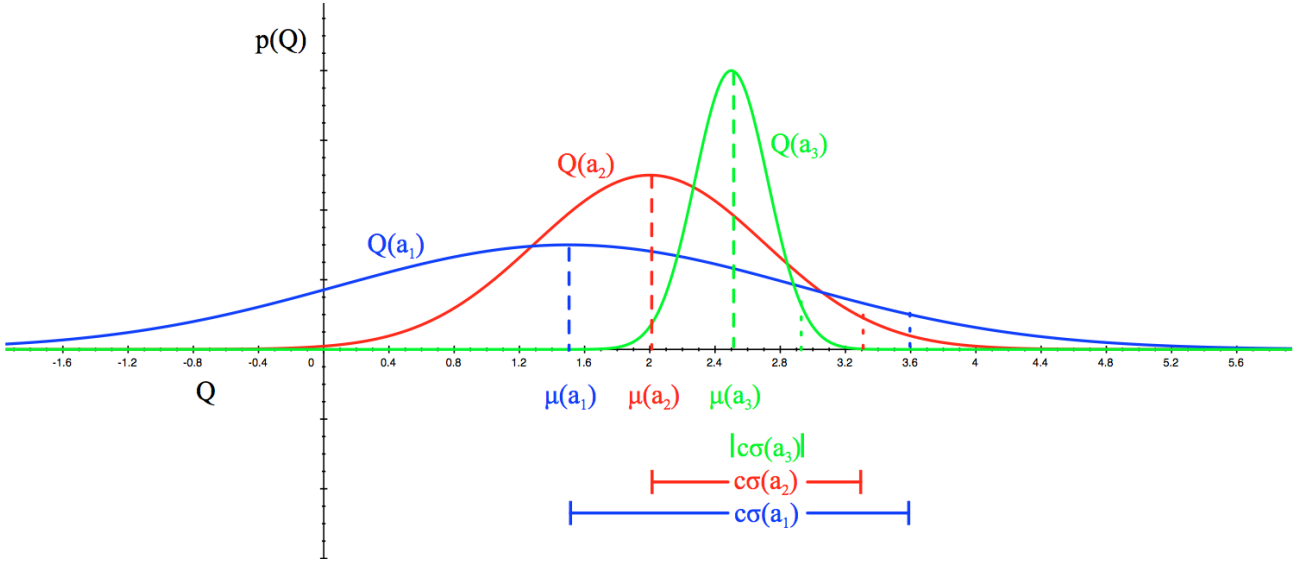
Note that the win count updates alternate between 0 or 1 (e.g. for black-white players) as you go up the tree.

Monte-Carlo Tree Search and Bandits

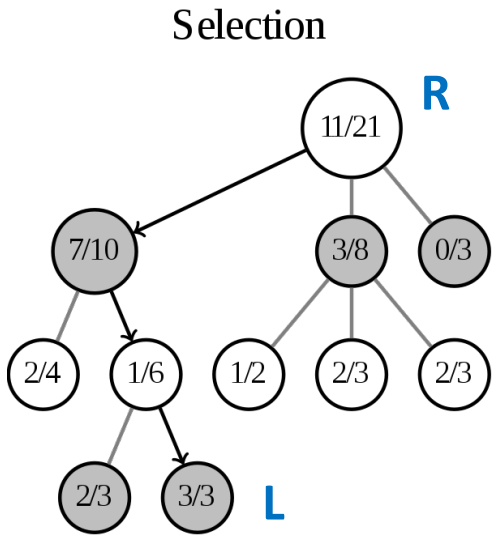


Assume there is some a-priori probability μ of winning from a given node (game state).

Nodes keep counts of games won or lost from previous visits, and so the posterior distribution of μ at a given state is like a bandit posterior. We can use bandit methods to chose nodes to explore.



Monte-Carlo Tree Search with UCB1



Selection: Treat child nodes as though they were bandit options.

Start from root *R* and select successive child nodes using UCB1, choose each child j to maximize:

$$\bar{x}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

where $\bar{x}_j$ is the average node utility so far, n_j is the visit count for the child, and n is the total number of steps. C_p is a constant e.g. $1/\sqrt{2}$.

Note: $n_j = 0$ for unvisited nodes ⇒ utility is ∞ which is greater than any visited node. So this approach fully expands a node before going deeper from its children.

AlphaGo Zero

Note that MCTS so far used neither neural networks nor reinforcement learning. So Deep Mind (of course) added those.

They defined a combined ***Policy-Value Deep Network***:

$$(\mathbf{p}, v) = f_\theta(s)$$

Where $\mathbf{p}$ is a probability distribution over the set of possible actions, and v is the value of the current state.

You can think of a policy-value network as an actor/critic pair implemented with a mostly-shared network with two heads (outputs): the action head and the value head.

Note: "state" here is actually the current state of the game plus a few moves of history. The history is only needed for certain games whose rules depend on recent history (ko in Go, 3-fold repetition in Chess). For simplicity assume it is just the current state.

AlphaGo Zero Reinforcement Learning

A ***policy iteration*** algorithm alternates these steps:

- ***Policy evaluation:*** Improve the current value function using a policy (e.g. using look-ahead with that policy).
- ***Policy improvement:*** Improve the policy using the updated value function.

AlphaGo uses lookahead (with MCTS) to improve both the value function and the policy probabilities at each board position. Then it uses these to update the policy-value network $f_\theta(s)$.

AlphaGo Zero Reinforcement Learning

The number of states visited during game play is large but manageable. So AlphaGo's RL algorithm stores ***tables*** (not deep nets) for each (s, a) edge in the game tree:

$\boldsymbol{N(s,a)}$ the visit count for action a in state s.

$\boldsymbol{Q(s,a)}$ the estimated action-value at the state s.

$\boldsymbol{P(s,a)}$ a prior probability for taking action a in state s.

Updates to these estimators and $f_\theta(s)$ comprise a ***policy iteration*** algorithm.

AlphaGo Zero Reinforcement Learning

The edge tables are updated as follows.

$\boldsymbol{N(s,a)}$: visit count for action a in state s. Increment when MCTS visits the edge (s,a).

$\boldsymbol{Q(s,a)}$: the estimated action-value at the state s. Compute a dynamic average:

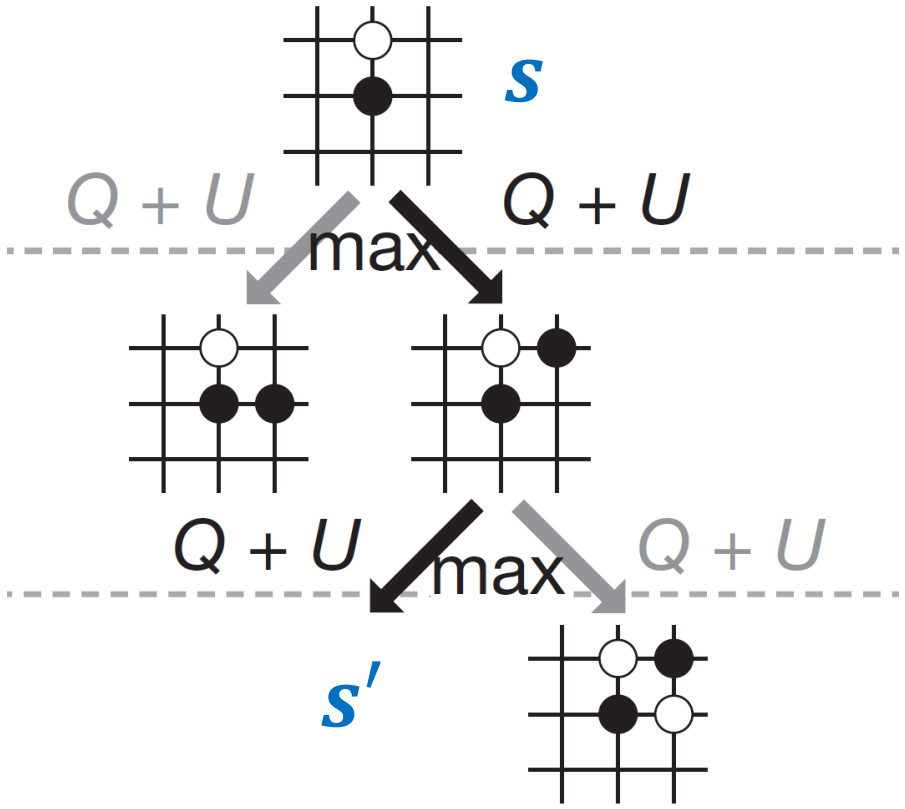
$$Q(s,a) = \frac{1}{N(s,a)} \sum_{s,a,\to s'} V(s')$$

Includes each s' reached from s,a in a previous traversal. There are $N(s,a)$ terms in this sum.

i.e. update $Q(s,a) = \frac{1}{N(s,a)}\left(V(s') + Q^{old}(s,a)(N(s,a)-1)\right)$ for each new leaf state s'.

$\boldsymbol{P(s,a)}$: Prior probability of action a in state s. More complicated. The first time (s,a) is visited, it comes from the policy: $P(s,a)$ is $\mathbf{p}(a)$ where $(\mathbf{p}, v) = f_\theta(s)$

AlphaGo Zero MCTS Selection



AlphaGo does MCTS by choosing the child node that maximizes the upper confidence bound:

$$Q(s,a) + U(s,a)$$

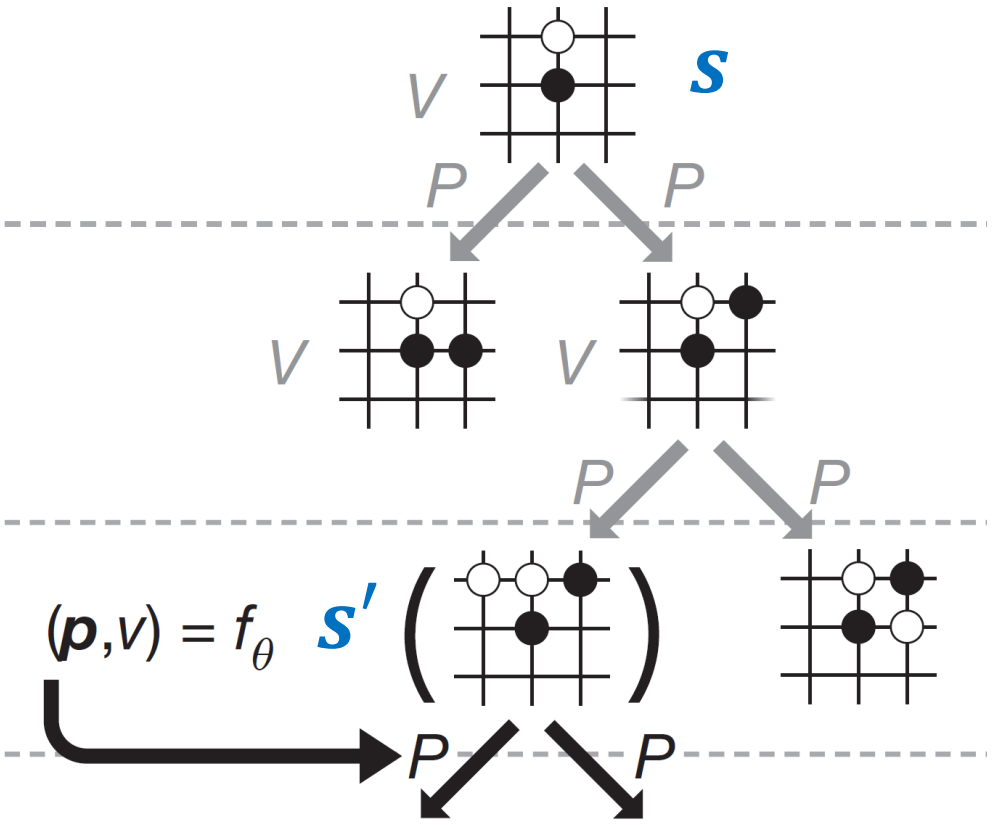
where $U(s,a)$ is given by the PUCT algorithm*:

$$U(s,a) = c_{puct} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)}$$

It recursively traverses the game tree until it reaches a leaf node s'.

* D. Auger et al. Continuous Upper Confidence Trees with Polynomial Exploration – Consistency, ECML/PKDD 2013.

AlphaGo Zero MCTS Expansion



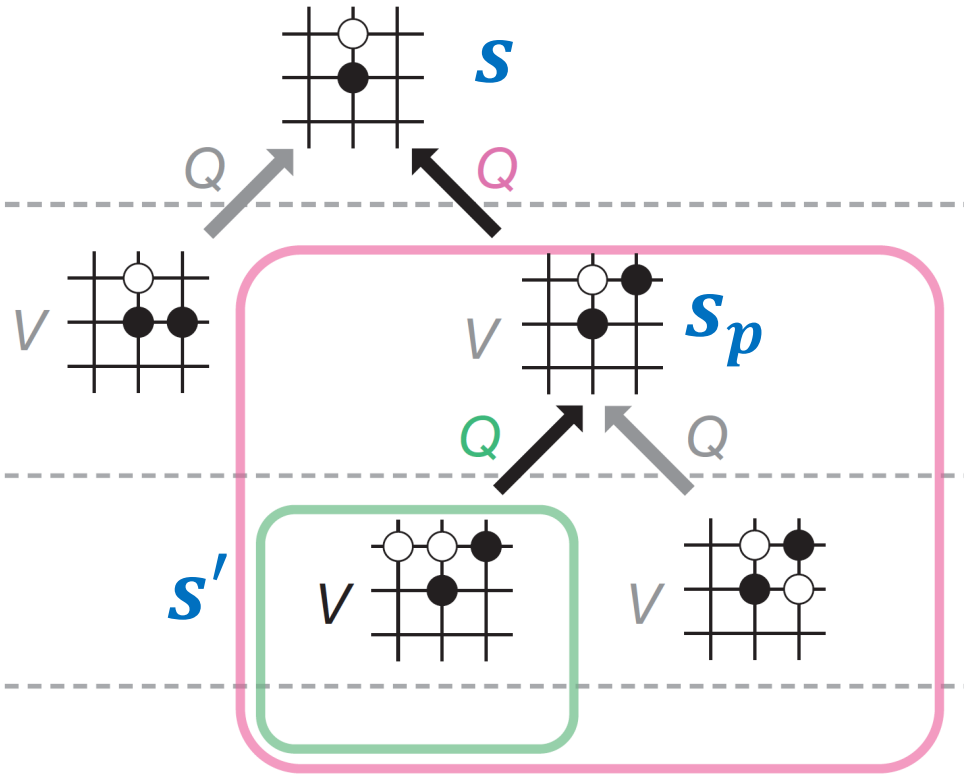
Once at a leaf node s', AlphaGo Zero expands the node just once using $f_\theta(s')$.

That produces a value $V(s')$ and the action probabilities for all the descendants of s'.

Those action probabilities become the new prior probabilities $P(s', a)$ for the actions a that follow s'.

Note: earlier versions of AlphaGo (Fan) used **simulation** (rollouts with a fast policy to the end of the game) as well as f_θ to estimate the leaf node's value. This was dropped in AlphaGo Zero.

AlphaGo Zero MCTS Backpropagation



The value of s' then becomes the initial value of Q for its parent (green edge).

The value of the parent s_p is updated by taking an action-probability-weighted average of the Q-values of its child edges.

This value then becomes the Q-value of the edge of s_p's parent (pink).

This concludes one round of MCTS. The search continues until the computation or time limit is hit.

AlphaGo Zero Training Policy

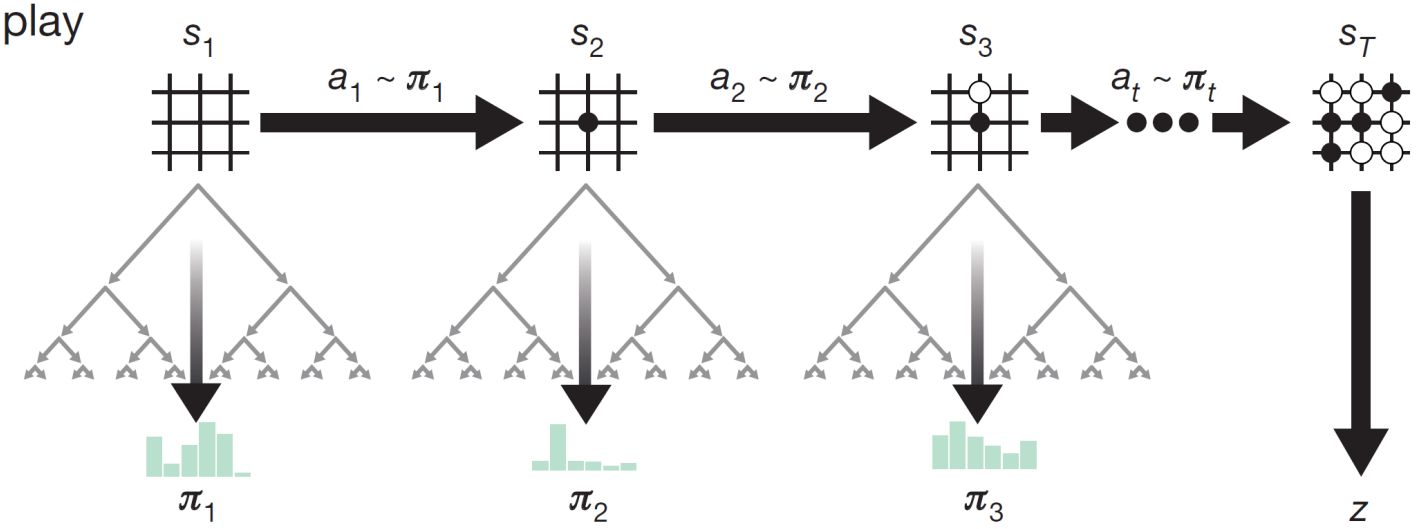
After MCTS, we several options for the policy $\pi(s)$ to be used for training (self-play).

AlphaGo Zero actually uses the counts $N(s,a)$ with (τ is a temperature parameter):

$$\pi(a) \propto N(s,a)^{1/\tau}$$

Thus AlphaGo Zero doesn't play the best move when training to play against itself. Rather it chooses a distribution across moves which allows it to explore.

As temperature τ decreases,
it is progressively more likely
to play only the best moves.



AlphaGo Zero Policy Iteration

AlphaGo Zero plays an entire game against itself with fixed policy parameters θ_{t-1}.

The result of this game is:

- A state sequence $s_1, \ldots, s_T$
- The corresponding policies at each state $\pi_1, \ldots, \pi_T$
- The final game reward $r_T = \{-1, +1\}$

Thus we have a sequence of experience tuples (s_t, π_t, z_t) where $z_t = \pm r_T$ the final reward from that player's perspective. We use this data to minimize a loss:

$$\sum_{t=1}^{T} (z_t - v_t)^2 + \pi_t^T \log p_t + c||\theta||^2$$

where $(p_t, v_t) = f_\theta(s_t)$ are the current model predictions.

AlphaGo Zero Policy Iteration

We use the MCTS data tuples (s_t, π_t, z_t) for a full game to minimize a loss:

$$\sum_{t=1}^{T} (z_t - v_t)^2 + \pi_t^T \log p_t + c||\theta||^2$$

where $(p_t, v_t) = f_\theta(s_t)$ are the current model predictions.

This loss combines L_2 loss on the value predictions, cross-entropy loss on the action probabilities and an L_2 regularization loss on the model parameters θ.

We can then update the model parameters using one or more SGD steps on minibatches of data sampled from the full-game sequence (s_t, π_t, z_t) for $t = 1, \ldots, T$.

AlphaGo Zero Training Summary

- We start with a policy-value deep network $(p, v) = f_\theta(s)$.
- We play a full game with parameter θ fixed:
  - At each state s_t, we run an MCTS search:
    - Use UCB search to find a leaf s', and use $f_\theta(s')$ to expand the leaf.
    - Backpropagate values, counts, probabilities up to s_t.
    - Repeat until time is up.
  - Use the counts to define a state policy $\pi_t(a) \propto N(s_t, a)^{1/\tau}$, and sample the next move.
  - Continue until the game is won or lost, $r_T \in \{-1, +1\}$, allocate reward to players $z_t = \pm r_T$.
- Collect the experience (s_t, π_t, z_t) for $t = 1, \ldots, T$ from this game. Update θ to minimize:

$$\sum_{t=1}^{T} (z_t - v_t)^2 + \pi_t^T \log p_t + c||\theta||^2$$

where $(p_t, v_t) = f_\theta(s_t)$

AlphaGo Zero Neural Network

The input features s_t are processed by a residual tower that consists of a single convolutional block followed by either 19 or 39 residual blocks.

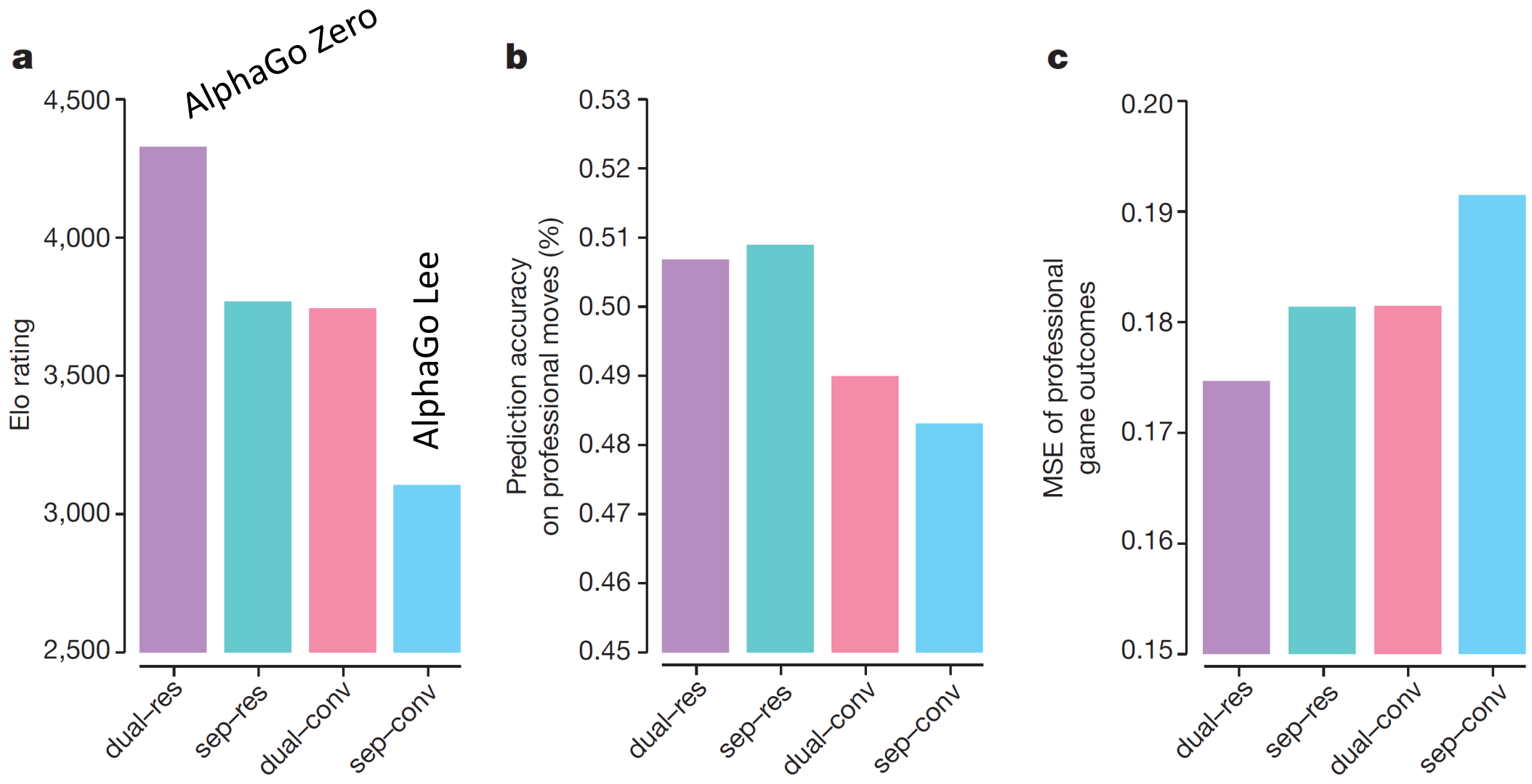
Convolution Block:
(1) A convolution of 256 filters of kernel size 3 × 3, stride 1
(2) Batch normalization
(3) A rectifier nonlinearity

Residual Block:
(1) A convolution of 256 filters of kernel size 3 × 3, stride 1
(2) Batch normalization
(3) A rectifier nonlinearity
(4) A convolution of 256 filters of kernel size 3 × 3, stride 1
(5) Batch normalization
(6) A skip connection that adds the input to the block
(7) A rectifier nonlinearity

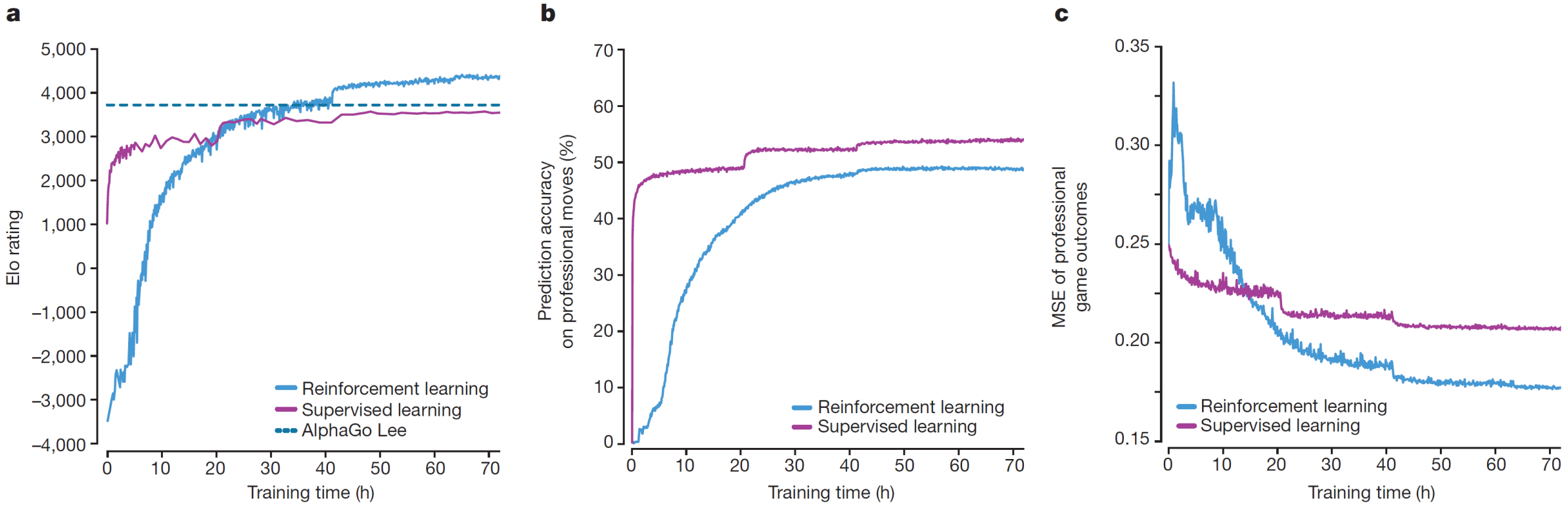
Policy Head:
(1) A convolution of 2 filters of kernel size 1 × 1, stride 1
(2) Batch normalization
(3) A rectifier nonlinearity
(4) A fully connected linear layer that outputs a vector of size $19_2 + 1 = 362$

Value Head:
(1) A convolution of 1 filter of kernel size 1 × 1, stride 1
(2) Batch normalization
(3) A rectifier nonlinearity
(4) A fully connected linear layer to hidden layer of size 256
(5) A rectifier nonlinearity
(6) A fully connected linear layer to a scalar
(7) A tanh nonlinearity outputting a scalar in range [− 1, 1]

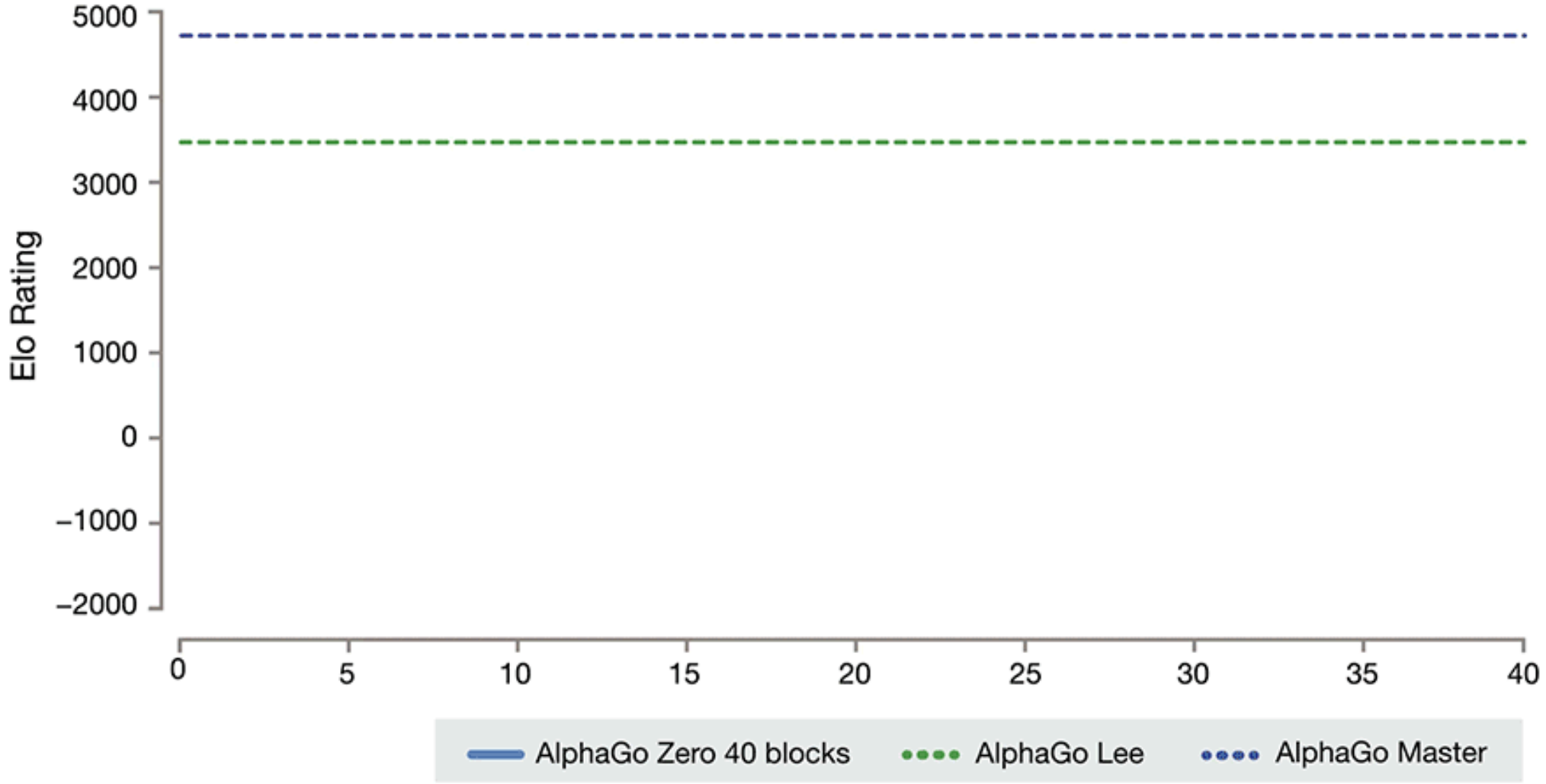
AlphaGo Zero Neural Network



AlphaGo Zero and Human Training Data



AlphaGo Zero Performance

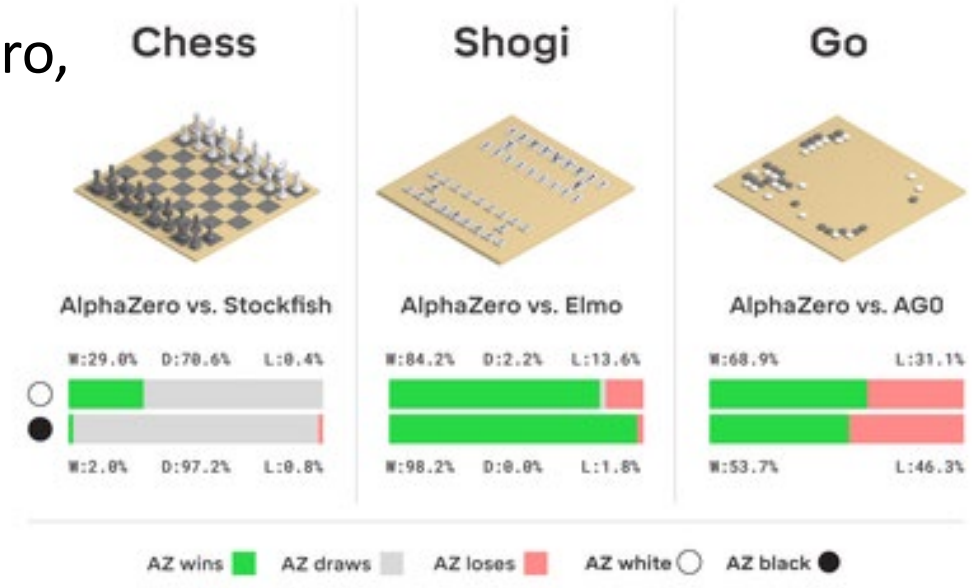


AlphaZero

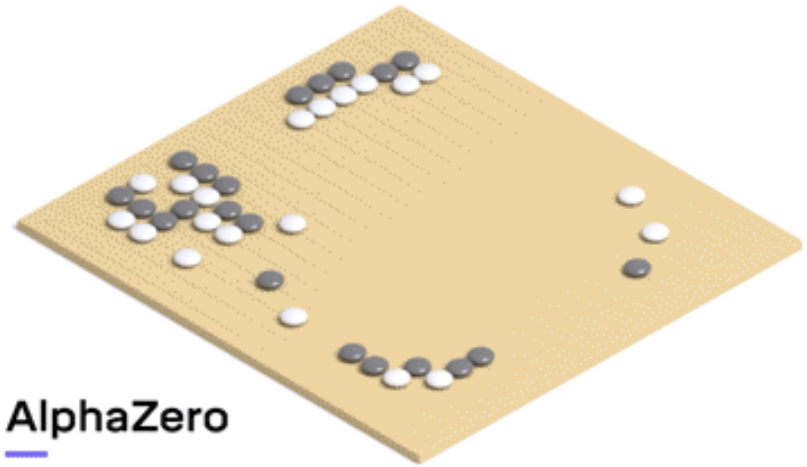
AlphaZero is nearly identical in design to AlphaGo Zero, and plays Chess, Go, and Shogi.

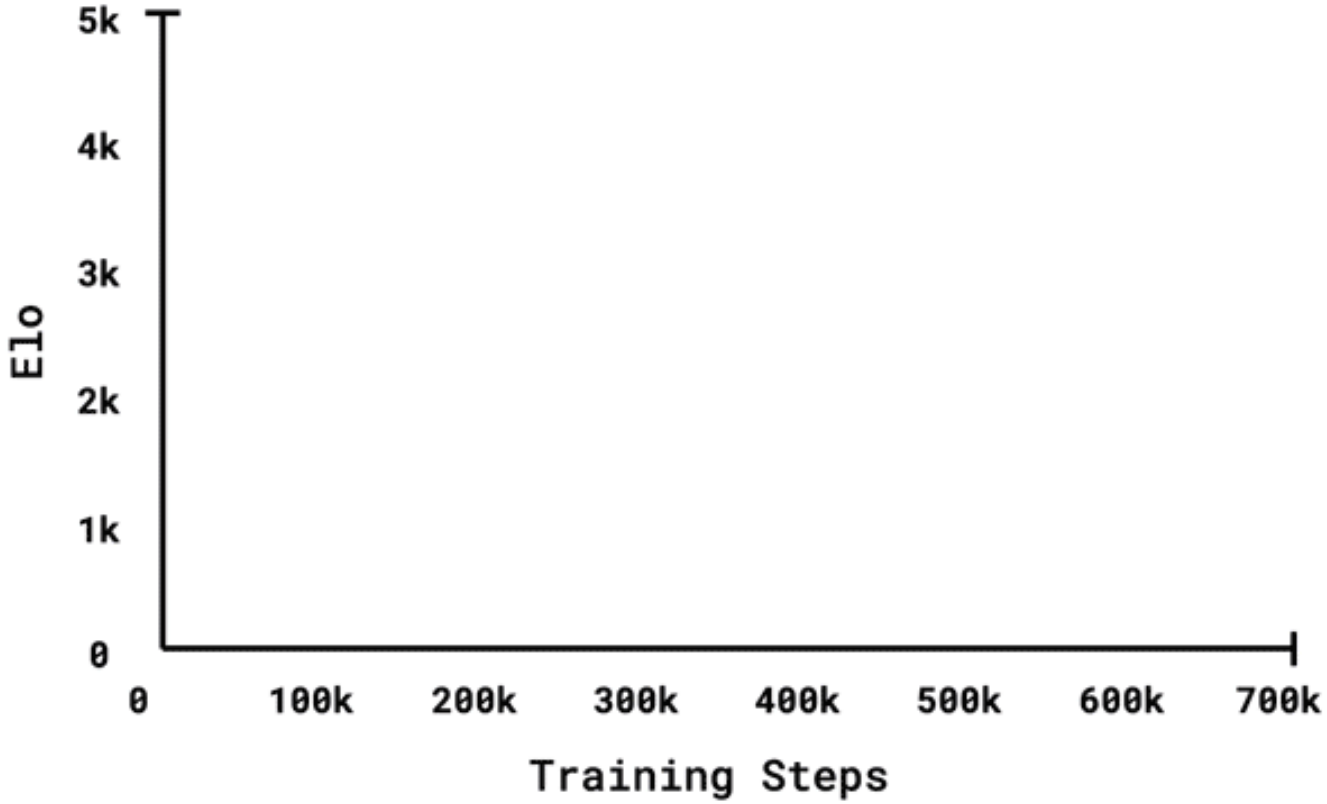
It sheds some Go-specific optimizations:

- Binary game outcomes replaced with real values.
- Symmetries are not supported.
- AlphaGo Zero trains with the best-to-date policy, AlphaZero simply updates the same policy.
- Compared to state-of-the-art Chess/Shogi programs, AlphaZero only searches about 1/1000th as many board positions (80,000 per second).



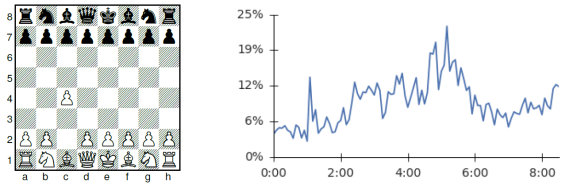
AlphaZero Training





AlphaZero Chess Opening Analysis

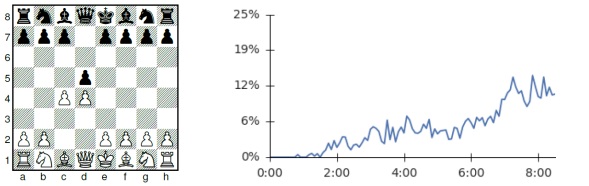
A10: English Opening



w 20/30/0, b 8/40/2

1...e5 g3 d5 cxd5 ♘f6 ♗g2 ♘xd5 ♘f3

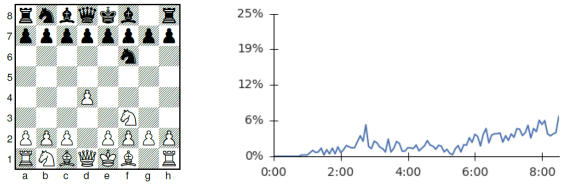
D06: Queens Gambit



w 16/34/0, b 1/47/2

2...c6 ♘c3 ♘f6 ♘f3 a6 g3 c4 a4

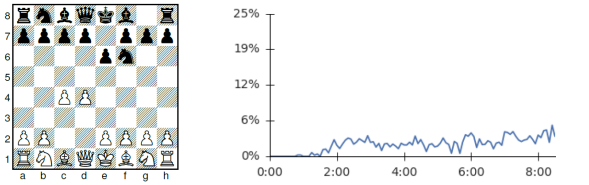
A46: Queens Pawn Game



w 24/26/0, b 3/47/0

2...d5 c4 e6 ♘c3 ♗e7 ♗f4 O-O e3

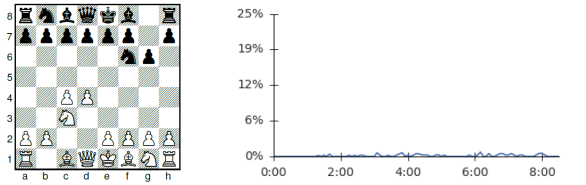
E00: Queens Pawn Game



w 17/33/0, b 5/44/1

3.♘f3 d5 ♘c3 ♗b4 ♗g5 h6 ♕a4 ♘c6

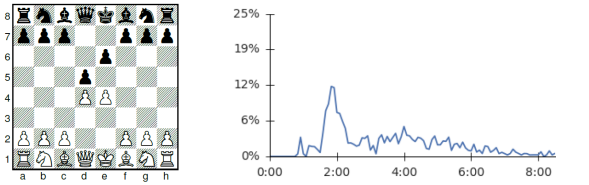
E61: Kings Indian Defence



w 16/34/0, b 0/48/2

3...d5 cxd5 ♘xd5 e4 ♘xc3 bxc3 ♗g7 ♗e3

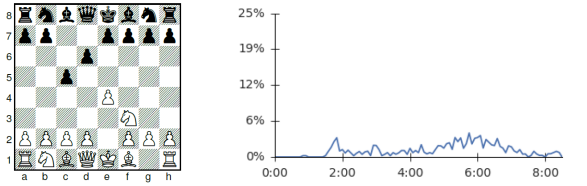
C00: French Defence



w 39/11/0, b 4/46/0

3.♘c3 ♘f6 e5 ♘d7 f4 c5 ♘f3 ♗e7

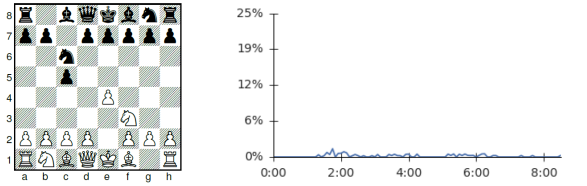
B50: Sicilian Defence



w 17/32/1, b 4/43/3

3.d4 cxd4 ♘xd4 ♘f6 ♘c3 a6 f3 e5

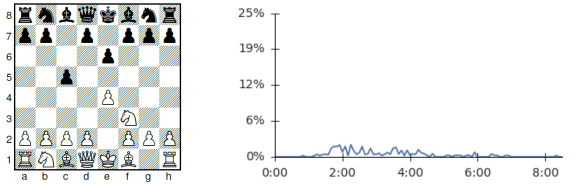
B30: Sicilian Defence



w 11/39/0, b 3/46/1

3.♗b5 e6 O-O ♘e7 ♖e1 a6 ♗f1 d5

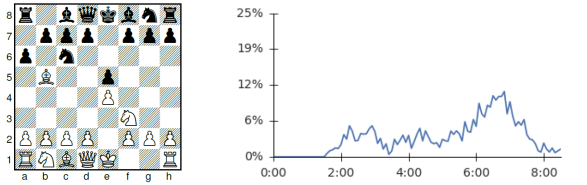
B40: Sicilian Defence



w 17/31/2, b 3/40/7

3.d4 cxd4 ♘xd4 ♘c6 ♘c3 ♕c7 ♗e3 a6

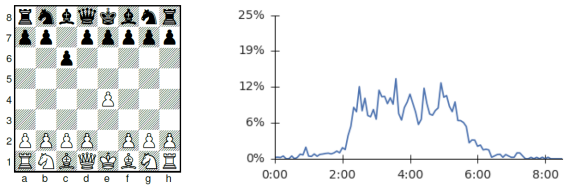
C60: Ruy Lopez (Spanish Opening)



w 27/22/1, b 6/44/0

4.♗a4 ♗e7 O-O ♘f6 ♖e1 b5 ♗b3 O-O

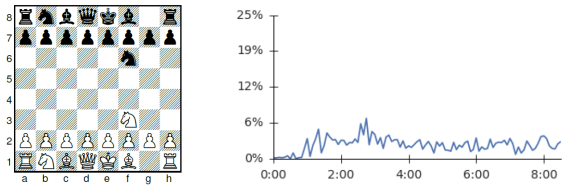
B10: Caro-Kann Defence



w 25/25/0, b 4/45/1

2.d4 d5 e5 ♗f5 ♘f3 e6 ♗e2 a6

A05: Reti Opening



w 13/36/1, b 7/43/0

2.c4 e6 d4 d5 ♘c3 ♗e7 ♗f4 O-O

Win/Draw/Loss rates against Stockfish, percent time that opening was played, and principal variations.

AlphaZero learns all the classical Chess openings through self-play only.

Take-aways

- Bandits are simple MDPs modeling explore-exploit tradeoffs.
- UCB methods encourage exploration, but provide near-optimal bounds on regret.
- Multiplayer games extend MDPs with player state.
- Monte-Carlo Tree Search applies UCB methods to tree search, allowing effective exploration/exploitation when playing games.
- AlphaGo Zero is a relatively simple UCB Monte-Carlo search algorithm with very good performance. It uses reinforcement only, no human data.
- AlphaGo Zero collects experience from an MCTS-lookahead neural policy and uses this experience to improve the policy using policy iteration.
- AlphaZero uses a virtually identical algorithm, but generalizes the game targets to Chess, Shogi and Go.