Deep Q-Learning, AlphaGo and AlphaZero

Lyle Ungar

With slides from Eric Eaton

Deep RL architecture
AlphaGo
AlphaZero

Remember Q-Learning

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(\underline{R + \gamma Q\big(s', \pi(s')\big) - Q(s,a)}\right)$$

Converges when this is zero

where

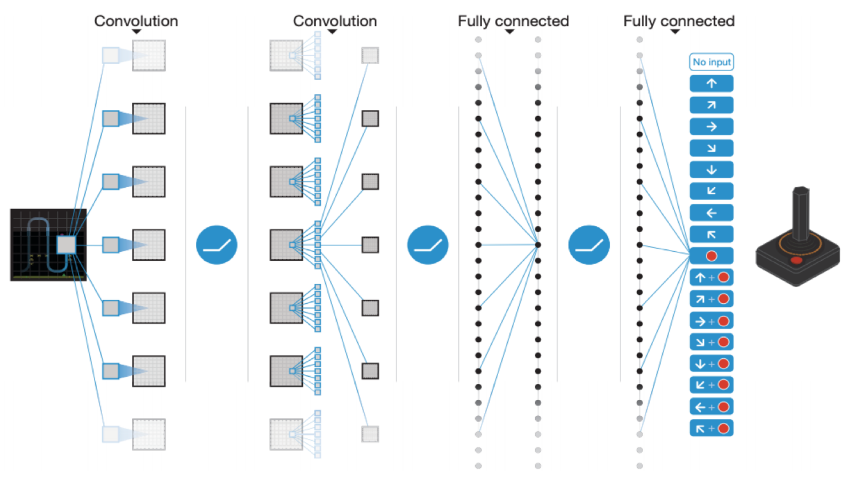
$$Q\big(s', \pi(s')\big) = max_{a'}\, Q(s', a')$$

Deep Q-Learning (DQN)

Input: *s*

Output: *Q(s,a)*

Learning:
gradient descent with the following loss function:



$$\left(\left(R(s,a,s') + \gamma \max_{a'} Q(s',a')\right) - Q(s,a)\right)^2$$

The policy, π*(a)*, is then given by maximizing the predicted Q-value

Separate Q- and Target Networks

Issue: Instability (e.g., rapid changes) in the Q-function can cause it to diverge

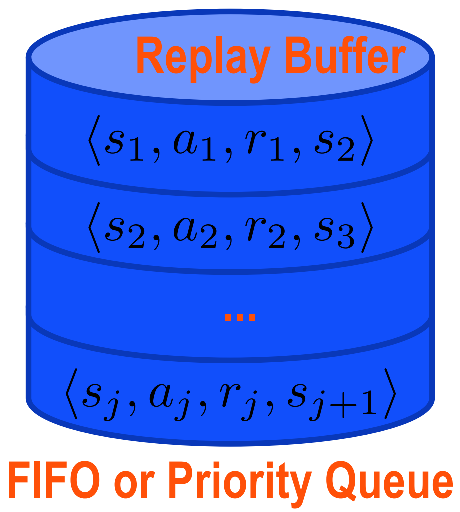
Idea: use two networks to provide stability

- The <u>Q-network</u> is updated regularly
- The <u>target network</u> is an older version of the Q-network, updated occasionally

$$\left(\left(R(s,a,s') + \gamma \max_{a'} \underbrace{Q(s',a')}_{\text{computed via target network}}\right) - \underbrace{Q(s,a)}_{\text{computed via Q-network}}\right)^2$$

Experience Replay

- **Maintain buffer of previous experiences**
- **Perform Q-updates based on a sample from the replay buffer**
- **Advantages:**
  - Breaks correlations between consecutive samples
  - Each experience step may influence multiple gradient updates



Deep Q-Learning (DQN) Algorithm

Initialize replay memory $\mathcal{D}$
Initialize Q-function weights θ
for episode $= 1 \dots M$, do
    Initialize state s_t
    for $t = 1 \dots T$, do

$$a_t \leftarrow \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \max_a Q^*(s_t, a; \theta) & \text{with probability } 1 - \epsilon \end{cases}$$

ε-greedy

        Execute action a_t, yielding reward r_t and state s_{t+1}
        Store $\langle s_t, a_t, r_t, s_{t+1} \rangle$ in $\mathcal{D}$
        $s_t \leftarrow s_{t+1}$
        Sample random minibatch of transitions $\{\langle s_j, a_j, r_j, s_{j+1} \rangle\}_{j=1}^{N}$ from $\mathcal{D}$

$$y_j \leftarrow \begin{cases} r_j & \text{for terminal state } s_{j+1} \\ r_j + \gamma \max_{a'} Q\left(s_{j+1}, a'; \theta\right) & \text{for non-terminal state } s_{j+1} \end{cases}$$

        Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$
    end for
end for

Based on https://arxiv.org/pdf/1312.5602v1.pdf

DQN on Atari Games

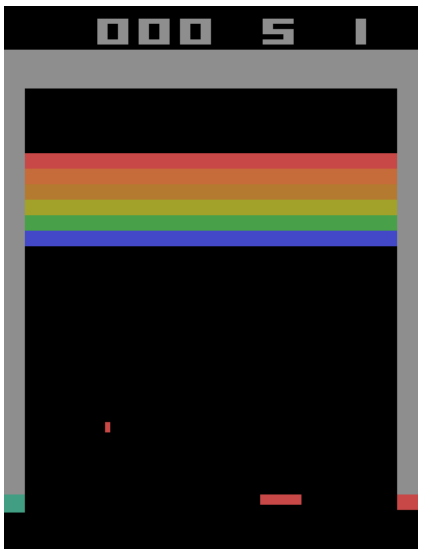
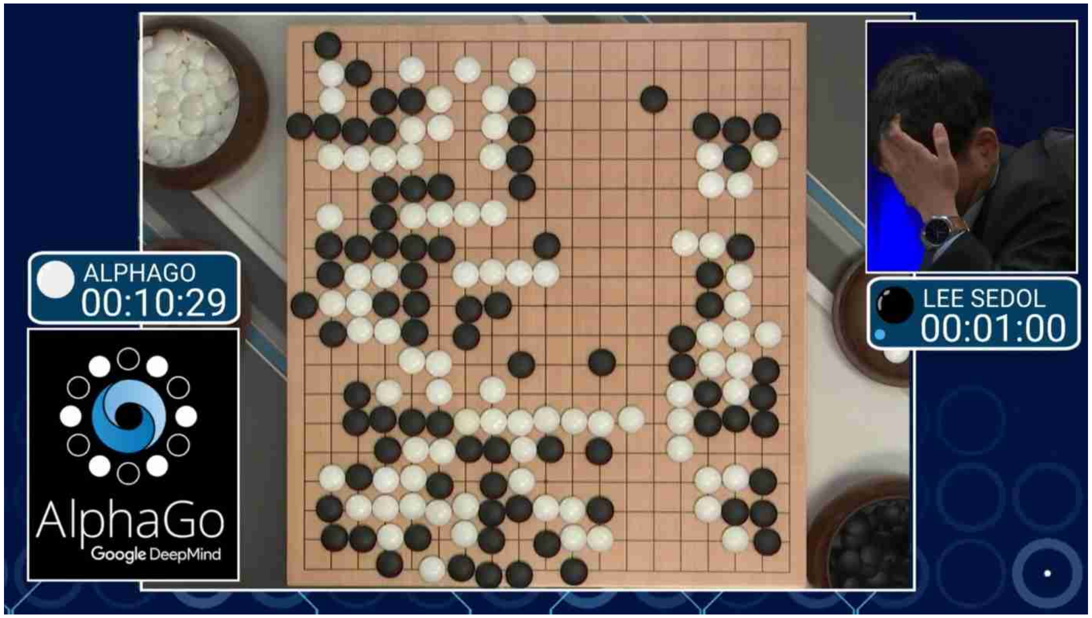


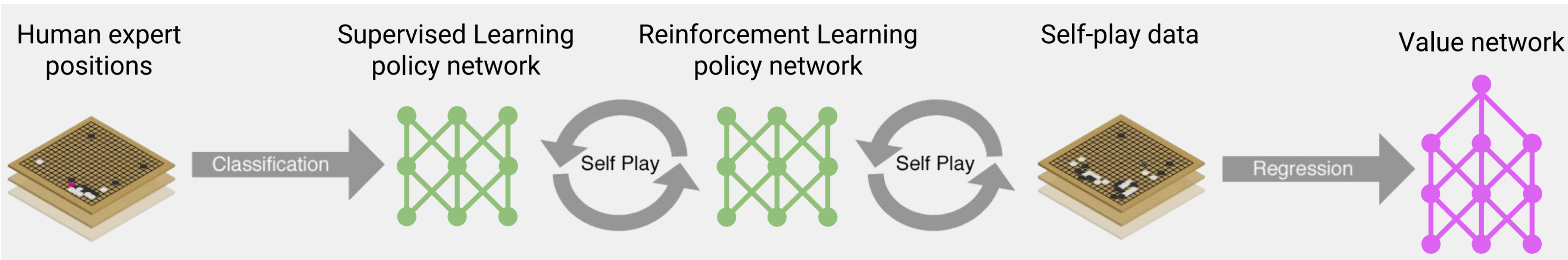
Image Sources:
https://towardsdatascience.com/tutorial-double-deep-q-learning-with-dueling-network-architectures-4c1b3fb7f756
https://deepmind.com/blog/going-beyond-average-reinforcement-learning/
https://jaromiru.com/2016/11/07/lets-make-a-dqn-double-learning-and-prioritized-experience-replay/

AlphaGo



https://medium.com/@jonathan_hui/alphago-how-it-works-technically-26ddcc085319 **2016**

AlphaGo



1. Train a CNN to predict (supervised learning) moves of human experts

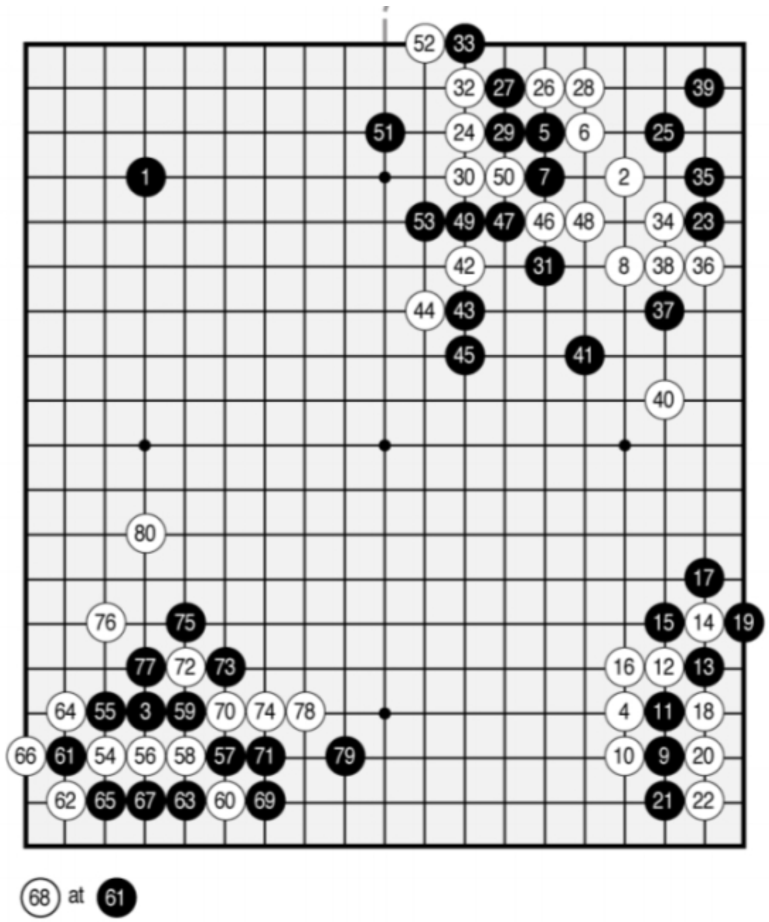
2. Use as starting point for policy gradient (self-play against older self)

3. Train value network with examples from policy network self-play

4. Use Monte Carlo tree search to explore possible games

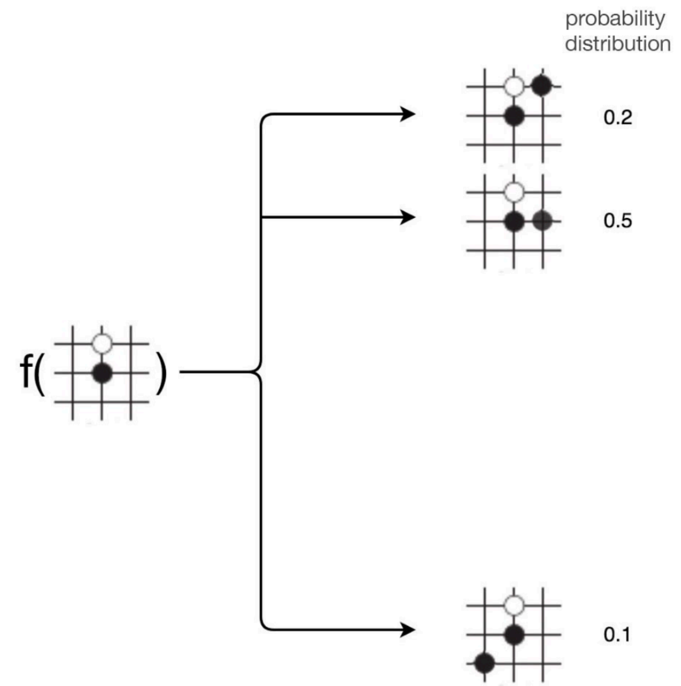
Image from DeepMind's ICML 2016 tutorial on AlphaGo: https://icml.cc/2016/tutorials/AlphaGo-tutorial-slides.pdf

Go – surround territory on 19x19 board



Learn policy

- ***a*** = *f*(***s***)
- ***a*** = where to play (19*19)
- ***s*** = description of board



State ~ 19*19*48

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

Train “SL policy network”

- 13 layers of convolutional filters and rectifiers
  - softmax classifier
- Train using moves for 30 million board positions
  - From previous human go games
- 50 GPUs for 3 weeks

SL = “supervised learning”

Approximate using rollout policy net

- SL policy network (55.7% accuracy) **3 ms**
- Rollout policy network (24.2% accuracy) **2 μs**
  - Also trained on human expert positions

RL policy network

- Initialize with SL policy network
- Train using self-play with older network
- Let z_t=*1* if it wins the game or *-1* if it loses

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

policy gradient RL

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a \mid s)}{\partial \sigma}$$

DL backprogagation

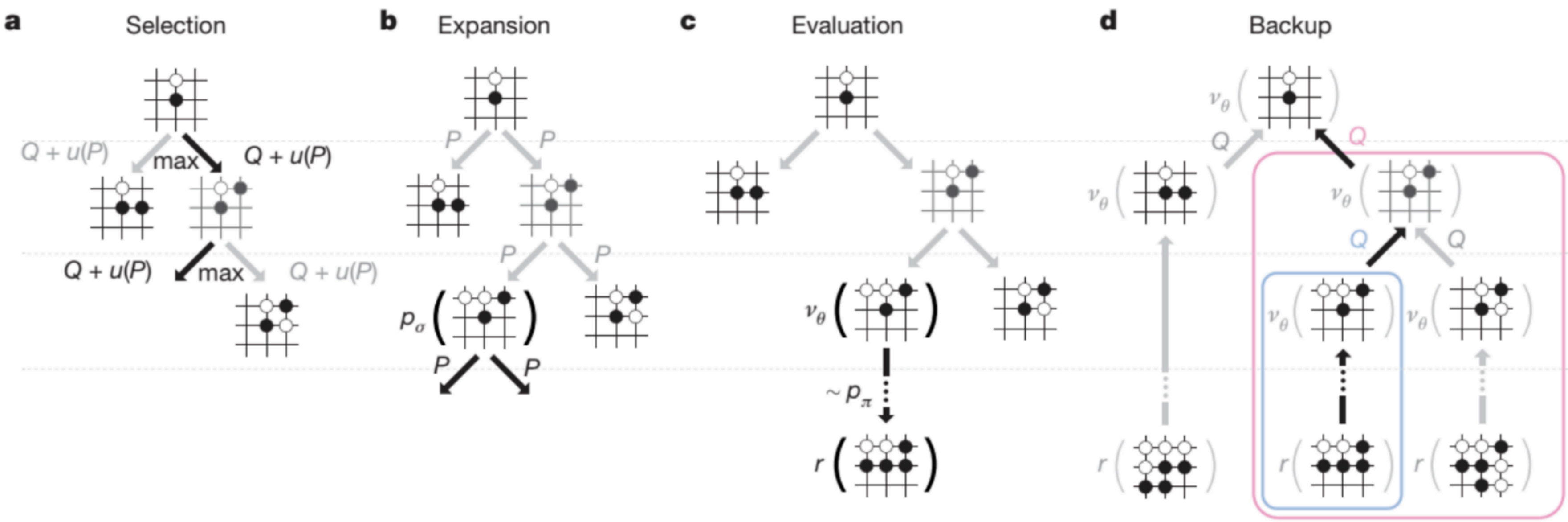
Value network

- **Estimate value of board state under the policy followed by the policy network**
- **Input:** one board from each self-play game
- **Output:** z (win/loss for that game)
- again, 50 GPUs for one week.

Monte Carlo Tree Search

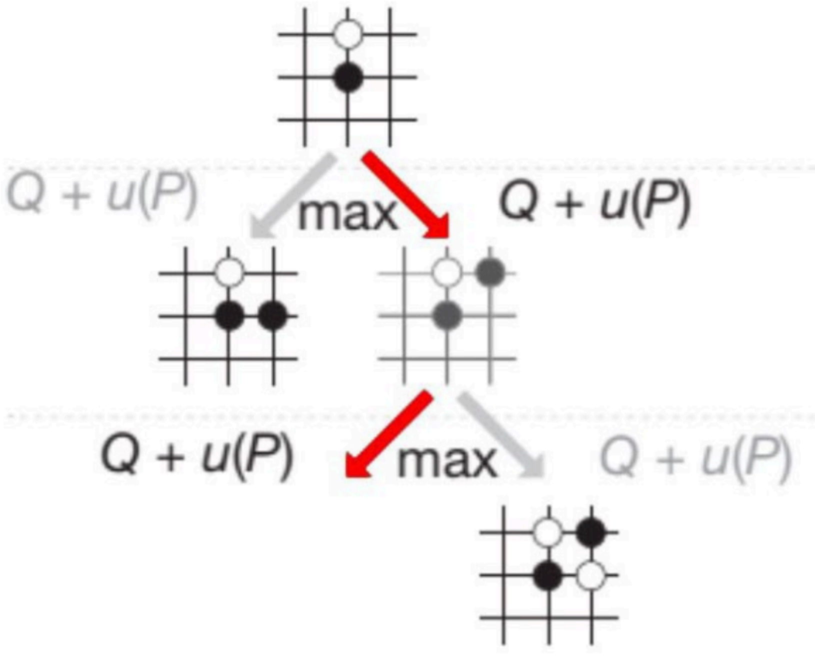
- **Need to trade off exploration and exploitation**
  - But can't afford to do a full search
- **Use Monte Carlo tree search**

Monte Carlo Tree Search



Action selection

$$a_t = \underset{a}{\operatorname{argmax}}\big(\underbrace{Q(s_t, a)}_{\text{exploitation}} + \underbrace{u(s_t, a)}_{\text{exploration}}\big)$$



$$u(s,a) \propto \frac{P(s,a)}{1 + N(s,a)}$$

$$P(s,a) = p_\sigma(a|s)$$ **SL policy net**

$$N(s,a) = \sum_{i=1}^{n} 1(s,a,i)$$ **# times *a* picked**

$$V(s_L) = (1-\lambda)\underbrace{v_\theta(s_L)}_{\text{value net}} + \underbrace{\lambda z_L}_{\text{previous game result}}$$

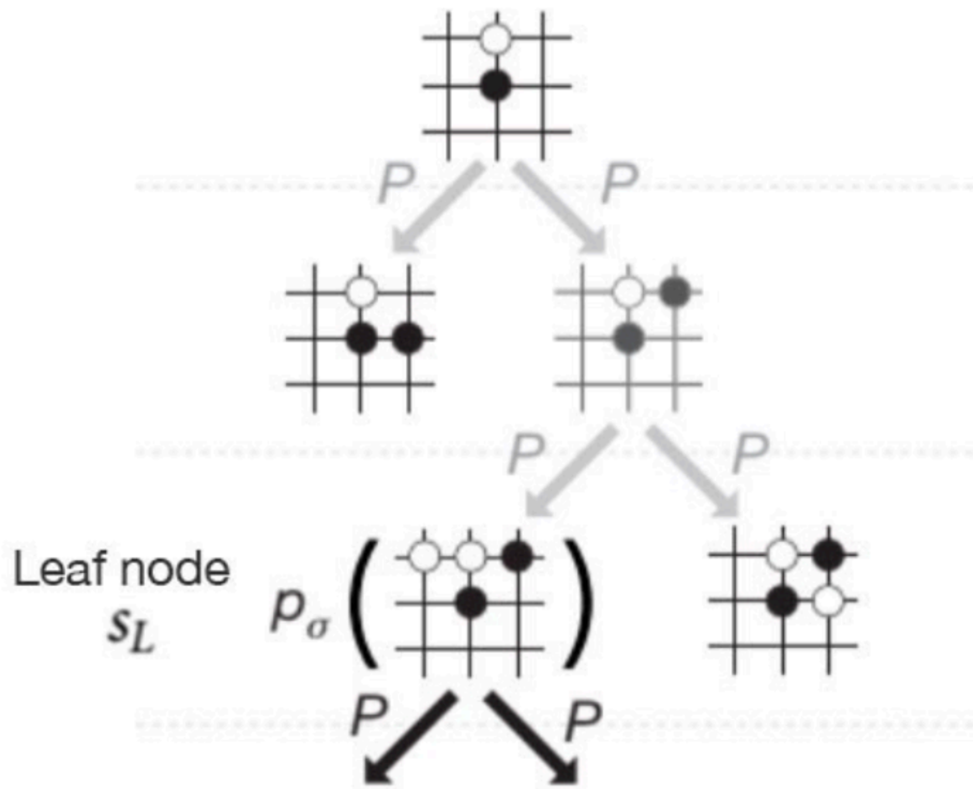
$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{n} 1(s,a,i) V(s_L^i)$$

$p_\sigma(a|s)$ - From the policy network: how good to take action a.
$v_\theta(s_L)$ - From the value network: how good to be in positions s_L.
$N(s,a)$ - How many times have we select action a so far.
z_L - the previous simulated game result.

You don't need to know this!

Expansion

- ***Q*** from RL value net
  - more accurate
  - use for exploitation
- ***P*** from SL policy network
  - more diverse
  - use for exploration (***u***)



You don't need to know this!

Evaluation

- Simulate the rest of the game using **Monte Carlo Rollout** starting from the leaf node
- Sample moves using the rollout policy.
  - Use the fast (but inaccurate) rollout net
    - 1500x faster
- Predicts a win or a loss z_L

Backup

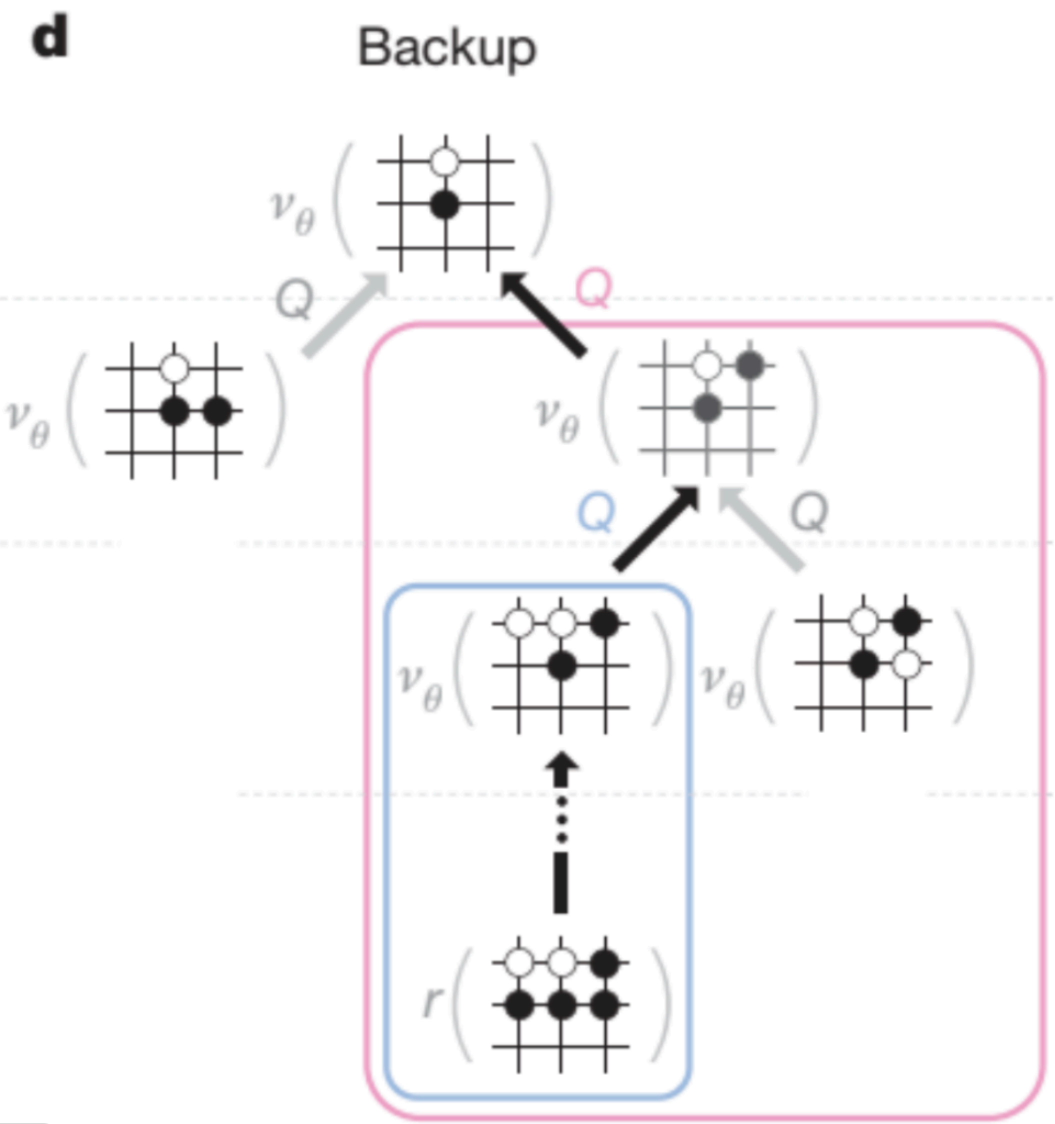
- Update **Q** with result of MC

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

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

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

You don't need to know this!

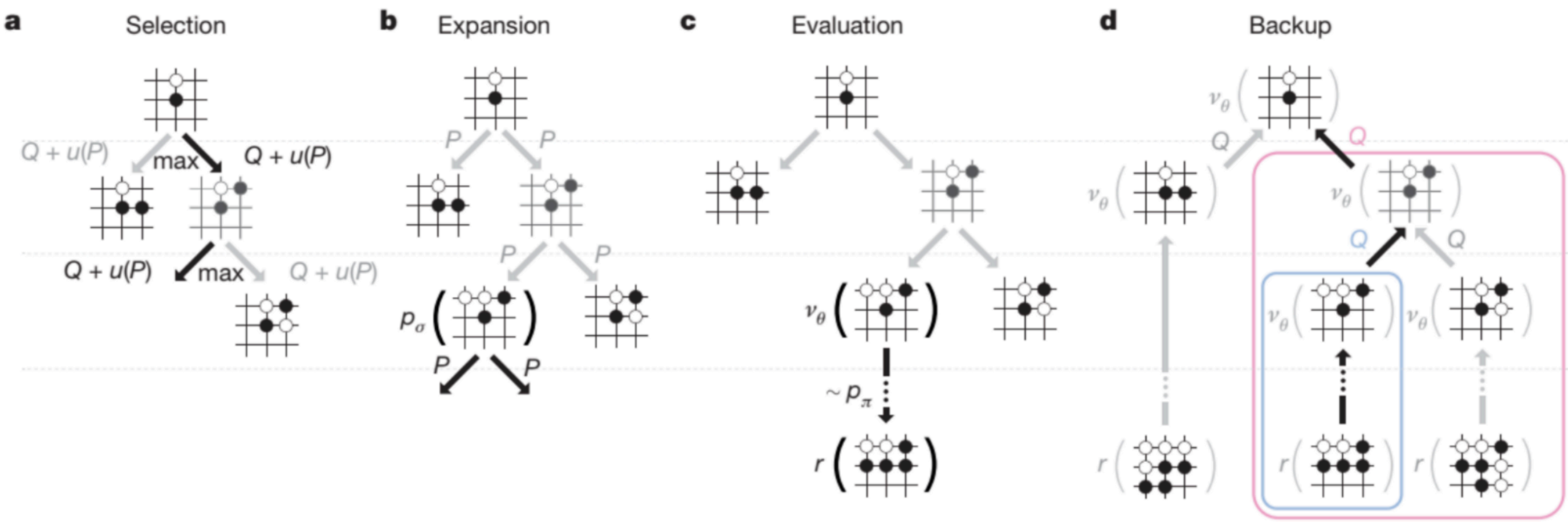


Picking the next move

- **Could use Q(s,a) -- but don't**
- **Use the move that was most often picked for the current board position**
  - Leads to decreasing exploration over time

You don't need to know this!

Monte Carlo Tree Search



Pick action

play it out
multiple times

see the game outcome

use to update Q(s,a)

AlphaGo

Training Requirements:

- CNN network: 30M human expert moves, 50 GPUs for 3 weeks
- Policy network: 10K minibatches of 128 games, 50 GPUs for 1 day
- Value network: 50M minibatches of 32 positions, 50 GPUs for 1 week (30M distinct positions from separate self-play games)

Computational Requirements:

- Stand-alone version: 40 search threads, 48 CPUs, 8 GPUs
- Distributed version: 40 search threads, 1,202 CPUs, 176 GPUs

Image from https://www.theverge.com/circuitbreaker/2016/5/19/11716818/google-alphago-hardware-asic-chip-tensor-processor-unit-machine-learning

AlphaGo take-aways

- **Bootstrap**
  - Initialize with policy learned from human play
- **Self-play**
- **Speed matters**
  - Rollout network (fast, less accurate game play)
  - Monte Carlo search
- **It still needs fast computers**
  - > 100 GPU weeks

AlphaZero

- **Single network**
  - instead of separate policy and value nets
  - Self-play with a single, continually updated neural net
  - No annotated features - just the raw board position
- **Uses Monte Carlo Tree Search**
- **Beat AlphaGo (100-0) after just 72 hours of training**
  - On 5,000 TPUs

https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go

2017

Monte Carlo Tree Search (MCTS)

- **In each state s_{root}, select a move, $a_t \sim \pi_t$ either proportionally** (exploration) **or greedily** (exploitation)
- **pick a move a_t with**
  - low visit count (not previously frequently explored)
  - high move probability (under the policy
  - high value (averaged over the leaf states of MC plays that selected *a* from *s*) according to the current neural net
- **The MCTS returns an estimate z of $v(s_{root})$ and a probability distribution over moves, $\pi = p(a|s_{root})$**

AlphaZero loss function

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

- Minimizes the error between the predicted outcome (value function) *v(s)* and the actual game outcome *z*
- Maximizes the similarity of the policy vector **p**(s) to the MCTS probabilities π*(s)*.
- L2 regularize the weights θ

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

RL Summary

- Why is DeepMind losing $500 million/year?

StarCraft

- StarCraft-playing AI model consists of 18 agents, each trained with 16 Google v3 TPUs for 14 days.
- Thus, at current prices ($8.00 / TPU hour), the company spent $774,000 on this model