Scalable ML

10605-10805

AlphaZero

Barnabás Póczos

AlphaGo - AlphaGo Zero – Alpha Zero

1. **Alpha Go:**
   - Versions: Alpha Go Fan, Alpha Go Lee
   - MCTS with rollout
   - Three neural networks
     (supervised learning policy, RL policy, and state value networks)
   - supervised learning from human expert moves

2. **AlphaGo Zero:**
   - MCTS without rollout
   - Uses a single neural network
   - Trained by self-play only without human data

3. **AlphaZero:**
   - MCTS without rollout
   - Uses a single neural network
   - Trained by self-play only without human data
   - Applied to chess and shogi

AlphaGo Zero

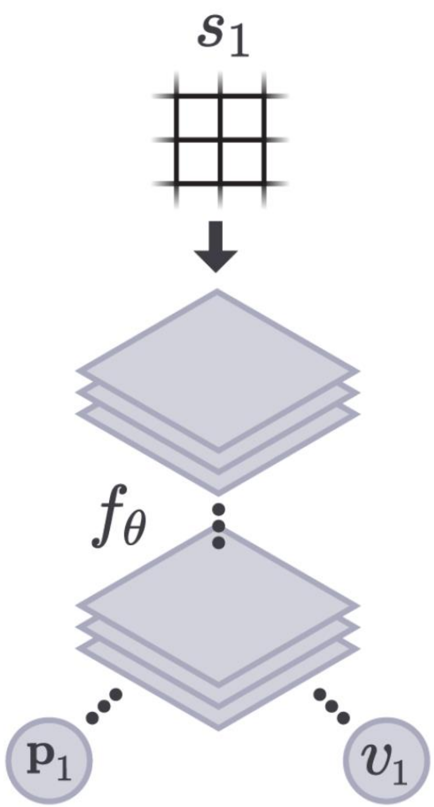
Reading Material

- Silver at al, **Mastering the game of Go without human knowledge**

The Neural Network

It uses only one deep neural network f_θ with parameters θ.

The neural network f_θ takes the board position s as its input.



It outputs both a vector p, representing a probability distribution over moves,

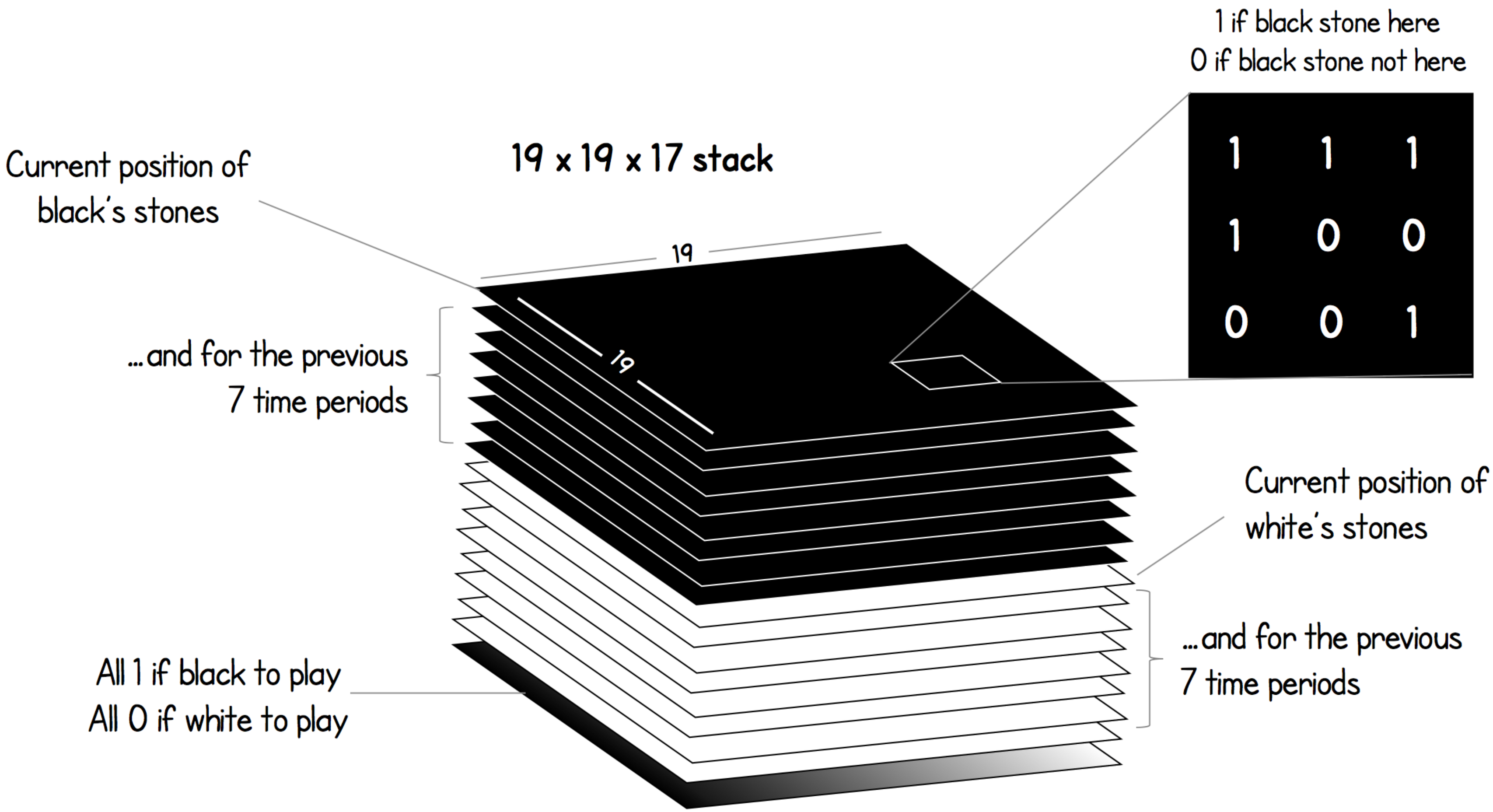
$$p_a = Pr(a|s)$$

and a scalar value v_t, representing the probability of the current player winning in position s_t

$$(p, v) = f_\theta(s)$$

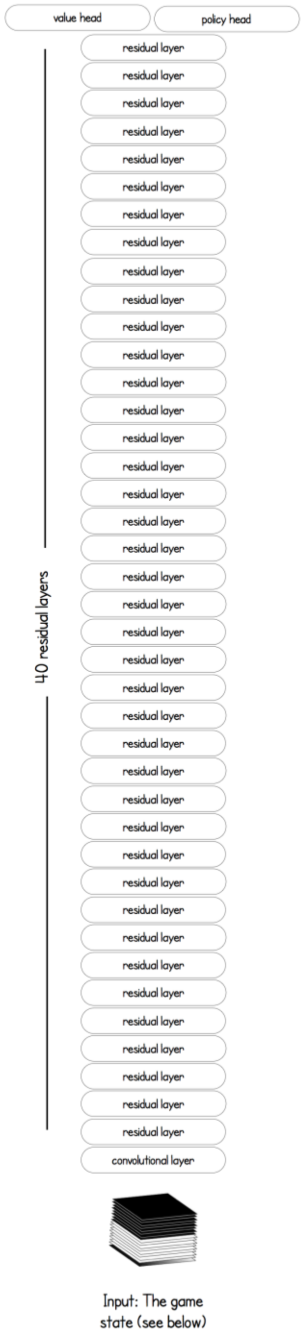
The neural network consists of many residual blocks, convolutional layers, batch normalization, and rectifier non-linearities.

The Game State

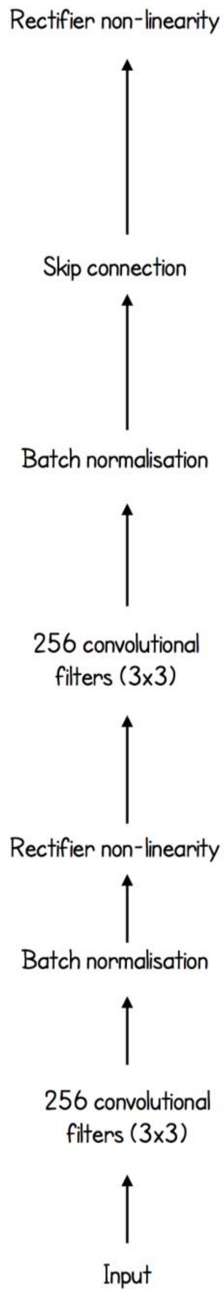


This stack is the input to the deep neural network

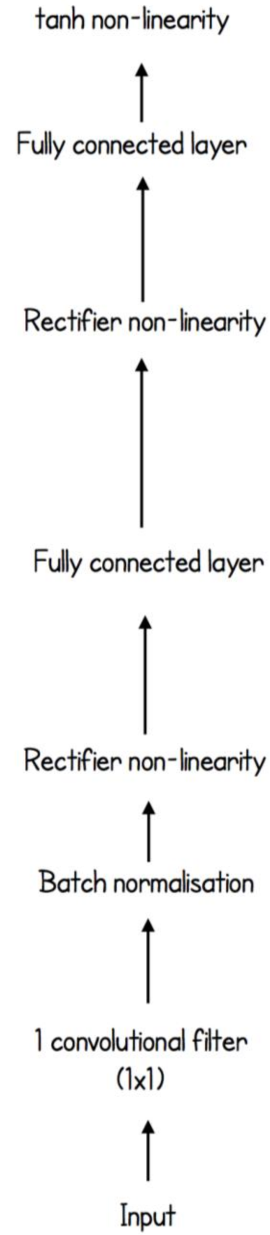
The Network



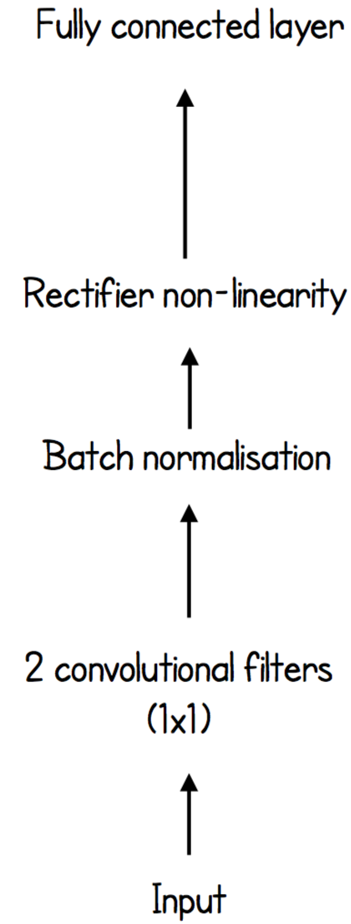
Residual block



The value head



The policy head



AlphaGo Zero

The training pipeline for AlphaGo Zero consists of three stages, executed in parallel:

- **Selfplay RL stage**,
- **Retrain stage**,
- **Evaluate stage**

AlphaGo Zero

The neural network in AlphaGo Zero is **trained from games of self-play** by a reinforcement learning algorithm.

In each position s, an MCTS search is executed, guided by the neural network f_θ.

The MCTS search outputs probabilities π of playing each move.

These search probabilities π usually **select much stronger moves** than the raw move probabilities p of the neural network $f_\theta(s)$.

MCTS may therefore be viewed as a powerful **policy improvement** operator.

AlphaGo Zero

Self-play with search using the improved MCTS-based policy π to select each move, then using the game winner z as a sample of the value may be viewed as a powerful **policy evaluation operator**.

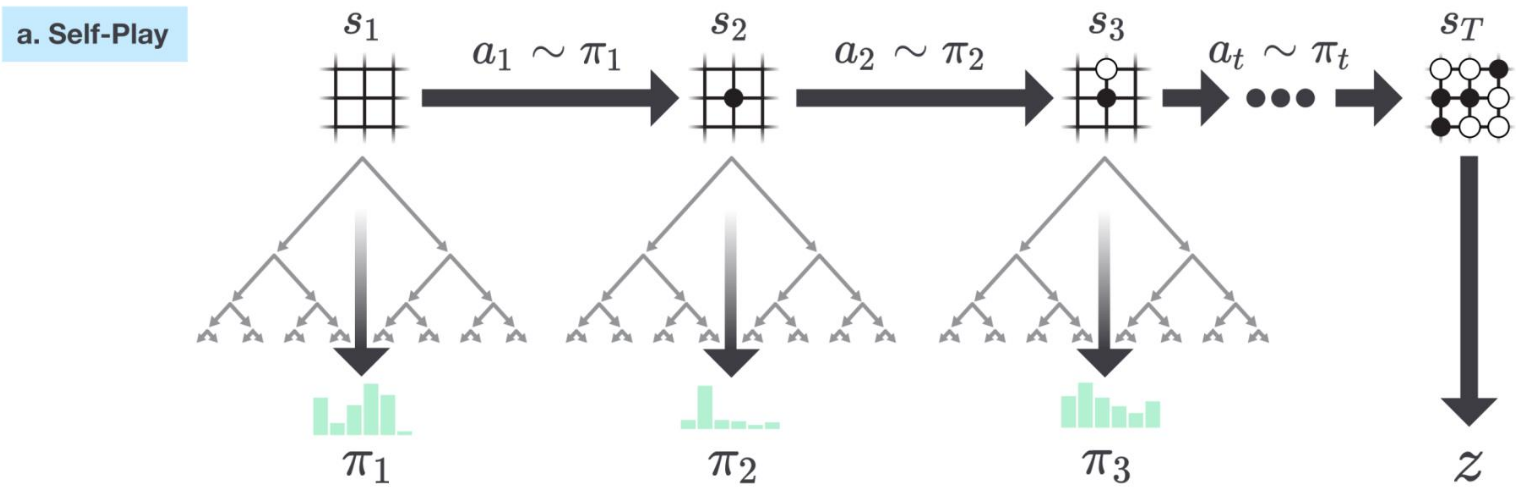
The main idea of the reinforcement learning algorithm:

the neural networks parameters are updated **to make the move probabilities and value** $(p, v) = f_\theta(s)$ **more closely match the improved search probabilities and self-play winner** (π, z).

these new parameters are used in the next iteration of self-play to make the search even stronger.

Self-Play Stage

Self-Play



A simulated game with self-play.

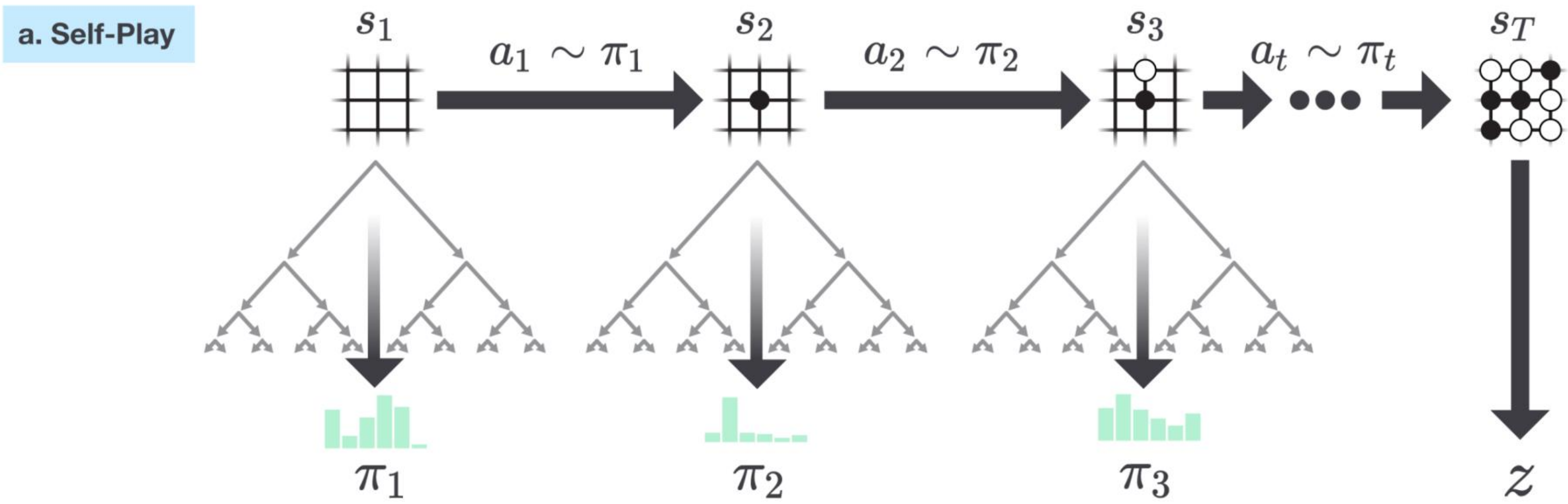
Store $(\{s_t, \pi_t\}_{t=1}^{T}, z)$ for each self-played game.
(game state, MCTS search probabilities, winner)

The game consists of T steps: $(s_1, \ldots, s_T)$

At the end of the simulated game we get reward z

Create a training set: The best current player playes 25,000 games against itself.

Self-Play



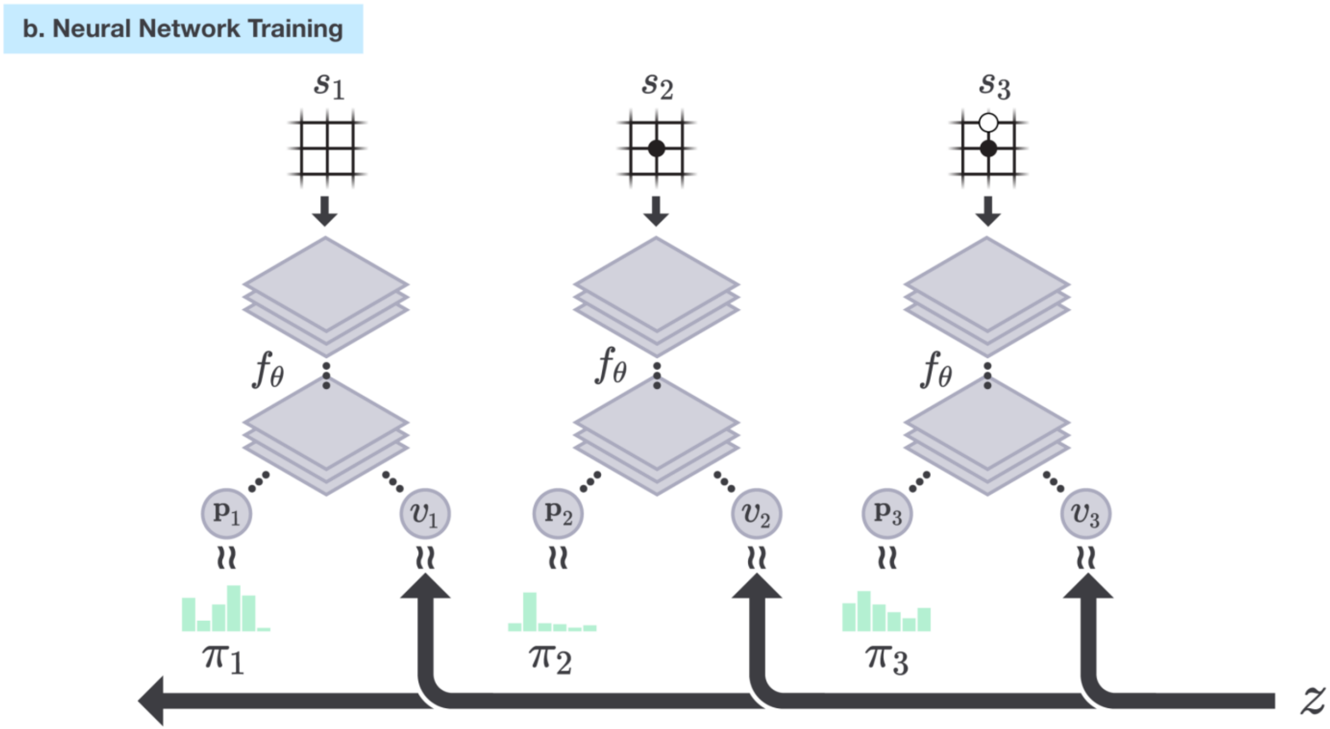
Monte-Carlo tree search (MCTS) α_θ is executed using the latest neural network f_θ. [Details later]

MCTS search provides a policy $\pi_t(s_t)$, which is better than $f_\theta(s_t)$.
$\pi_t(s_t) = \alpha_{\theta_i}(s_t)$

Moves during the self-play are selected according to the search probabilities computed by the MCTS, $a_t \sim \pi_t(s_t)$

Retrain stage

Retrain the Neural Network

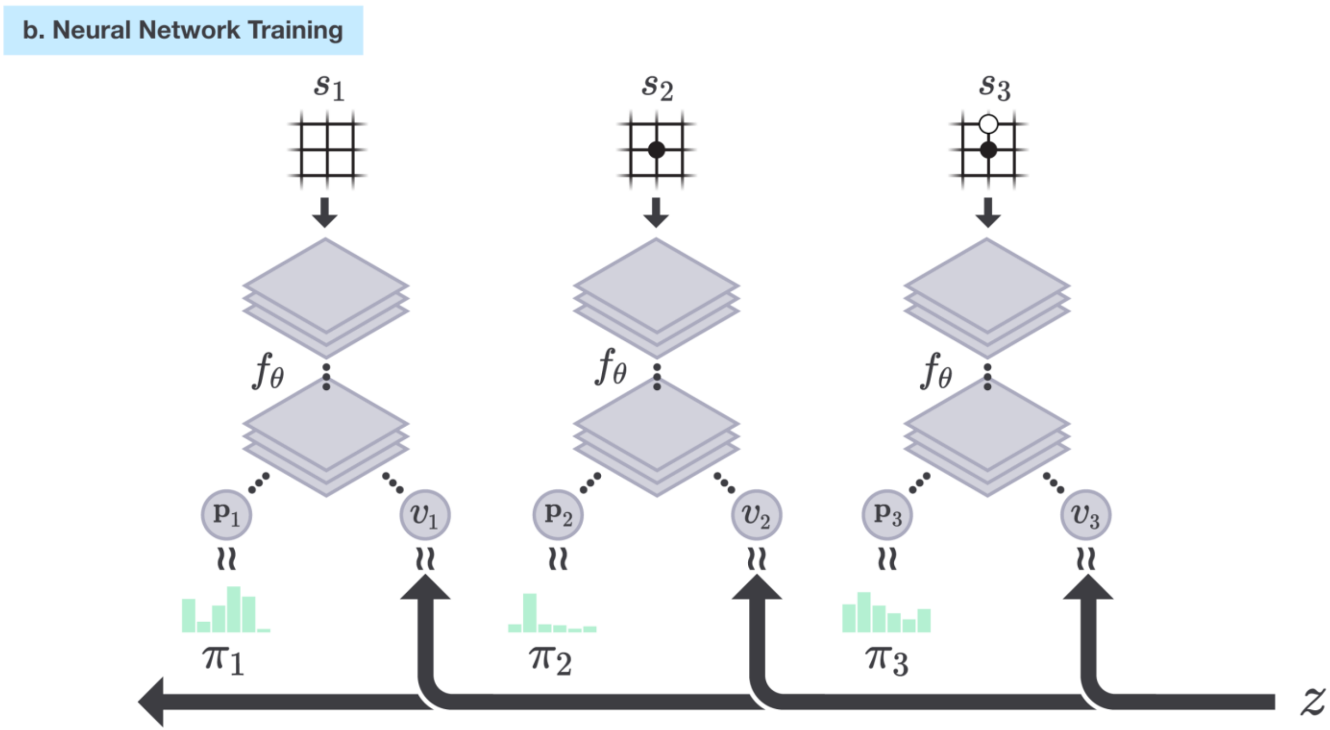


This retraining step runs parallel with the self-play step.

The neural network is optimized from recent self-play data.

In AlphaGo Zero the training data is augmented with rotations and reflections in each position.

Retrain the Neural Network

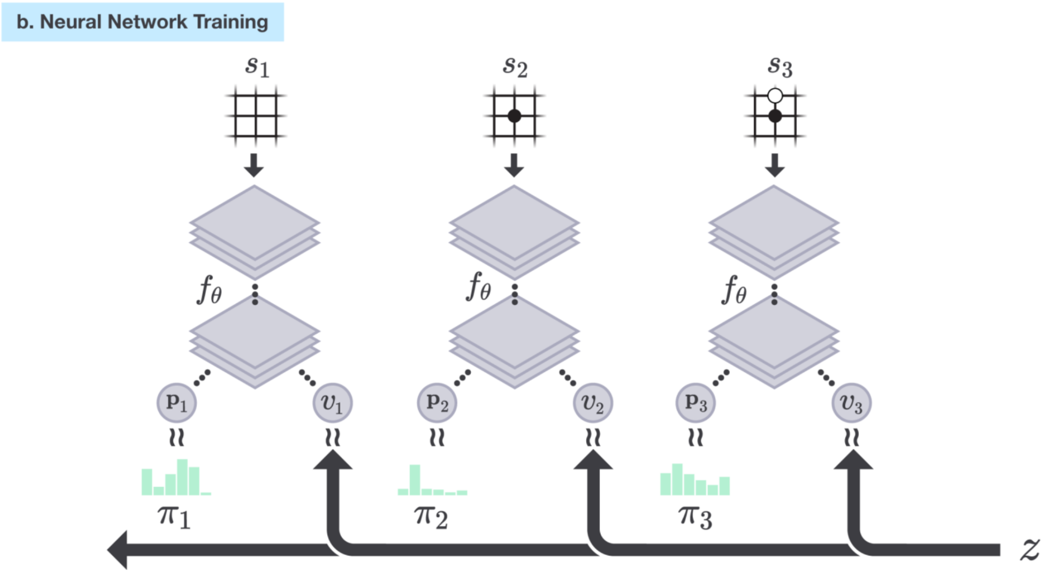


For **training** the neural network, **64 GPU workers and 19 CPU** parameter servers were used.

The batch-size is 32 per worker, for a **total mini-batch size of 32*64=2,048**. Each mini-batch of data is **sampled** uniformly at random from all positions from the **most recent 500,000** games of self-play.

Momentum based SGD was used for training with decreasing learning rate.

Retrain the Neural Network



Loss function:

$$(p, v) = f_\theta(s).$$

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

MSE + Cross entropy + regularization

The neural network parameters θ are updated so as to maximize the similarity of the vector p_t to the search probabilities π_t,

and to minimize the error between the predicted winner v_t and the game winner z.

The new parameters are used in the next iteration of self-play.

Retrain the Neural Network

- Training started from completely random behavior and continued without human intervention for approximately 3 days.
- Over the course of training, 4.9 million games of self-play were generated
- 1,600 simulated steps were used for each MCTS step, which corresponds to approximately 0.4s thinking time per move.
- Parameters were updated from 500,000 mini-batches of 2,048 positions. We used these minibatches to retrain the network
- The neural network contained 20 (or 40?) residual blocks.
- After 1000 training loops, we create a check point and evaluate the network. It may become the new best network.

Evaluate Networks

Evaluator

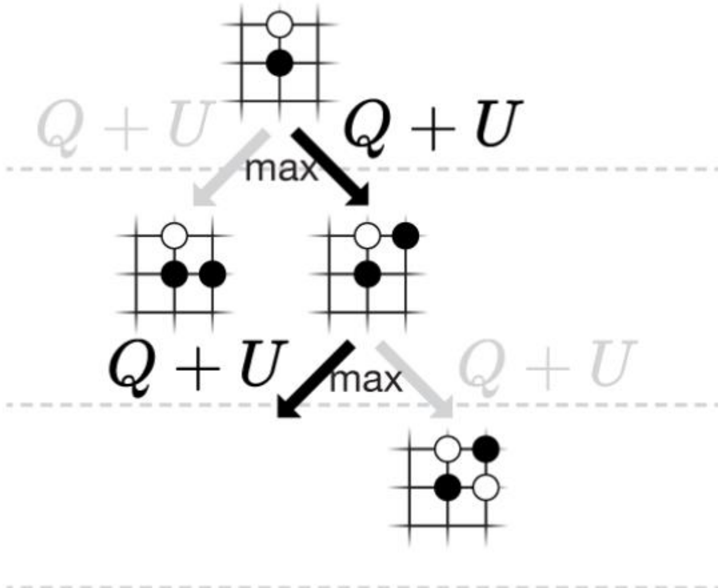
To ensure we always generate the best quality data, we evaluate each new neural network checkpoint against the current best network f_θ before using it for data generation.

Each evaluation consists of 400 games, using an MCTS with 1,600 simulations to select each move

If the new player wins by a margin of $> 55\%$ (to avoid selecting on noise alone) then it becomes the best player, and is subsequently used for self-play generation.

Monte Carlo Tree Sampling (MCTS)

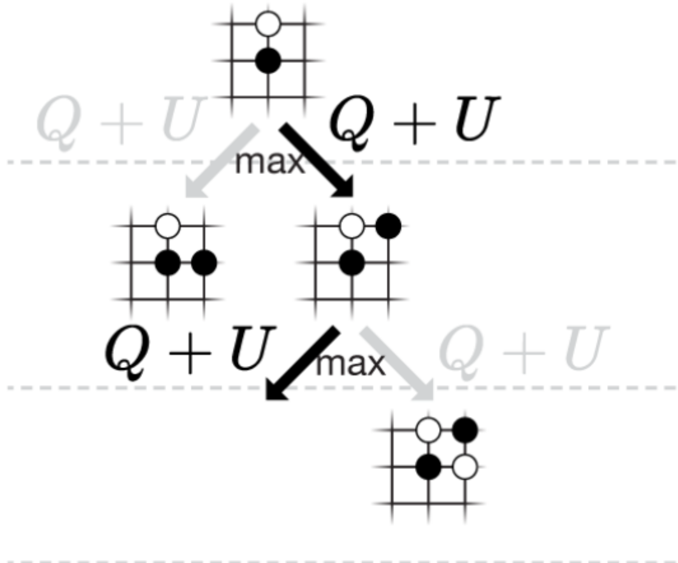
MCTS Select



The Monte-Carlo tree search uses the neural network f_θ to guide its simulations.

Multiple simulations are executed in parallel on separate search threads.

MCTS Select

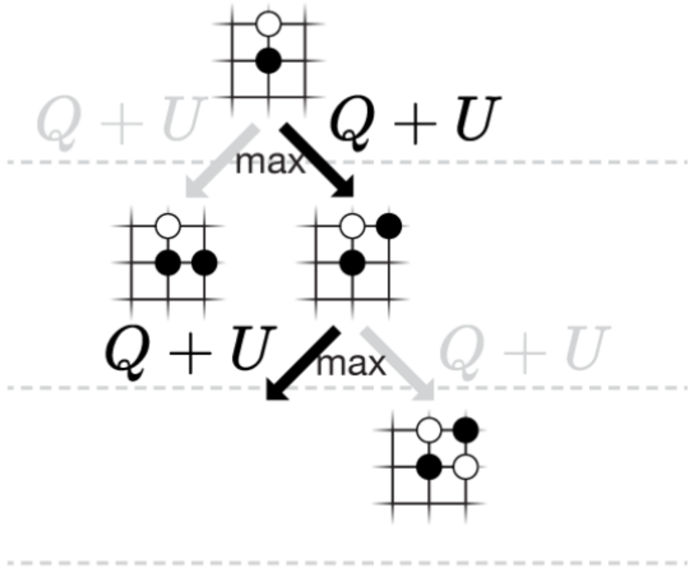


Each node s in the search tree contains edges (s, a) for all legal actions $a \in \mathcal{A}(s)$. Each edge stores a set of statistics,

$$\{N(s,a), W(s,a), Q(s,a), P(s,a)\},$$

where $N(s,a)$ is the visit count, $W(s,a)$ is the total action-value, $Q(s,a)$ is the mean action-value, and $P(s,a)$ is the prior probability of selecting that edge.

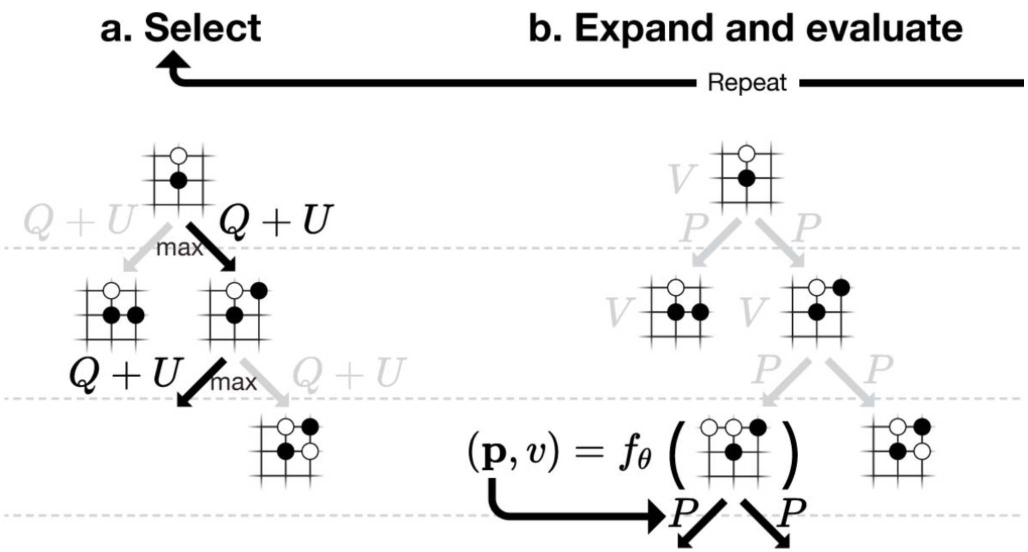
MCTS Select



Each simulation starts from the root state and iteratively selects moves that maximise an upper condence bound $Q(s,a)+U(s,a)$, until a leaf node s is encountered.

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

MCTS Expand and Evaluate

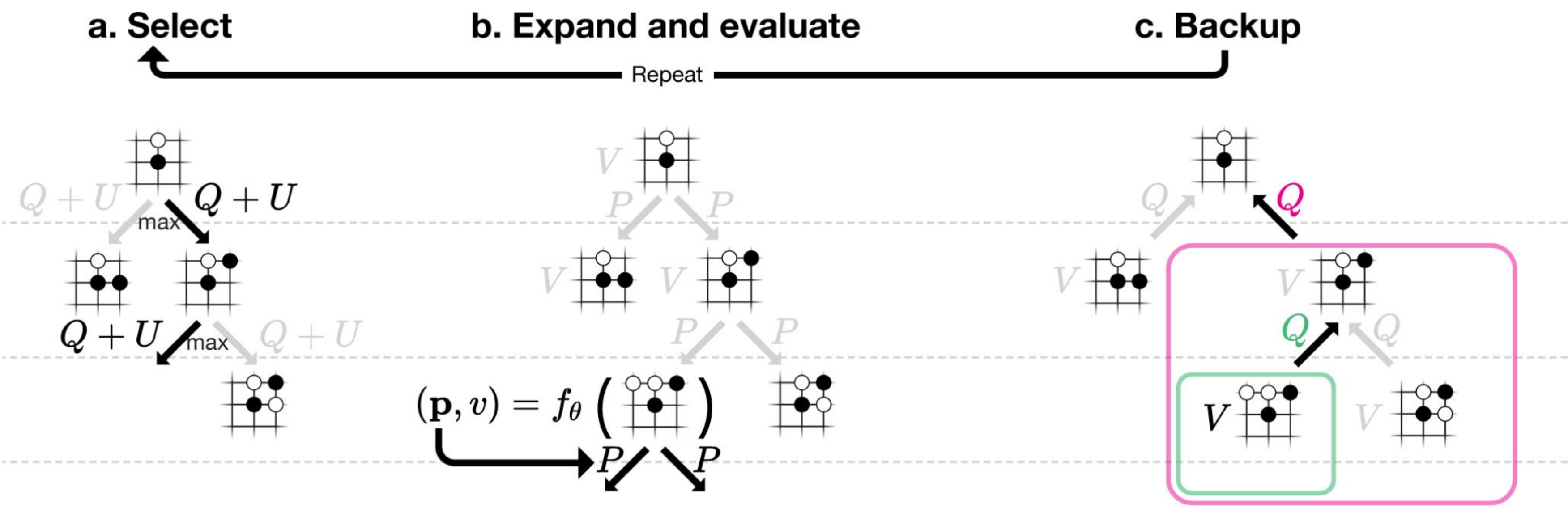


b) The leaf node is expanded and the associated position s_L is evaluated by the neural network $(P(s_L, \cdot), V(s_L)) = f_\theta(s_L)$; the vector of P values are stored in the outgoing edges from s_L.

The new edge (s_L, a) is initialized to

$\{N(s_L, a) = 0, W(s_L, a) = 0, Q(s_L, a) = 0, P(s_L, a) = p_a\}$; the value v is then backed up.

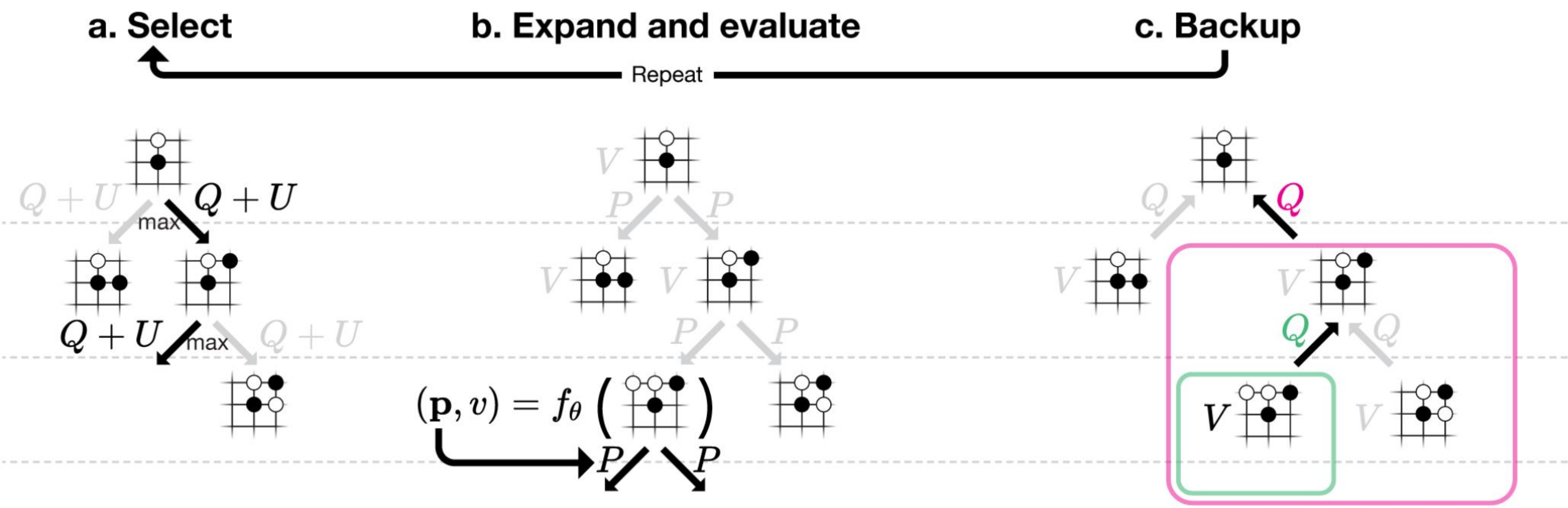
MCTS Backup



c) Action-values Q are updated to track the mean of all evaluations V in the subtree below that action.

Each edge (s, a) traversed in the simulation is updated to increment its visit count $N(s, a)$, and to update its action-value $Q(s, a)$ to the mean evaluation over these simulations.

MCTS Backup



The edge statistics are updated in a backward pass through each step $t \leq L$.

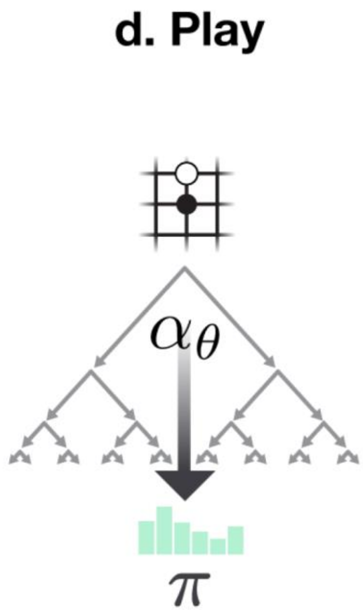
The visit counts are incremented, $N(s_t, a_t) = N(s_t, a_t) + 1$,

and the action-value is updated to the mean value:

$$W(s_t, a_t) = W(s_t, a_t) + v, \qquad \text{where } (p, v) = f_\theta(s_L)$$

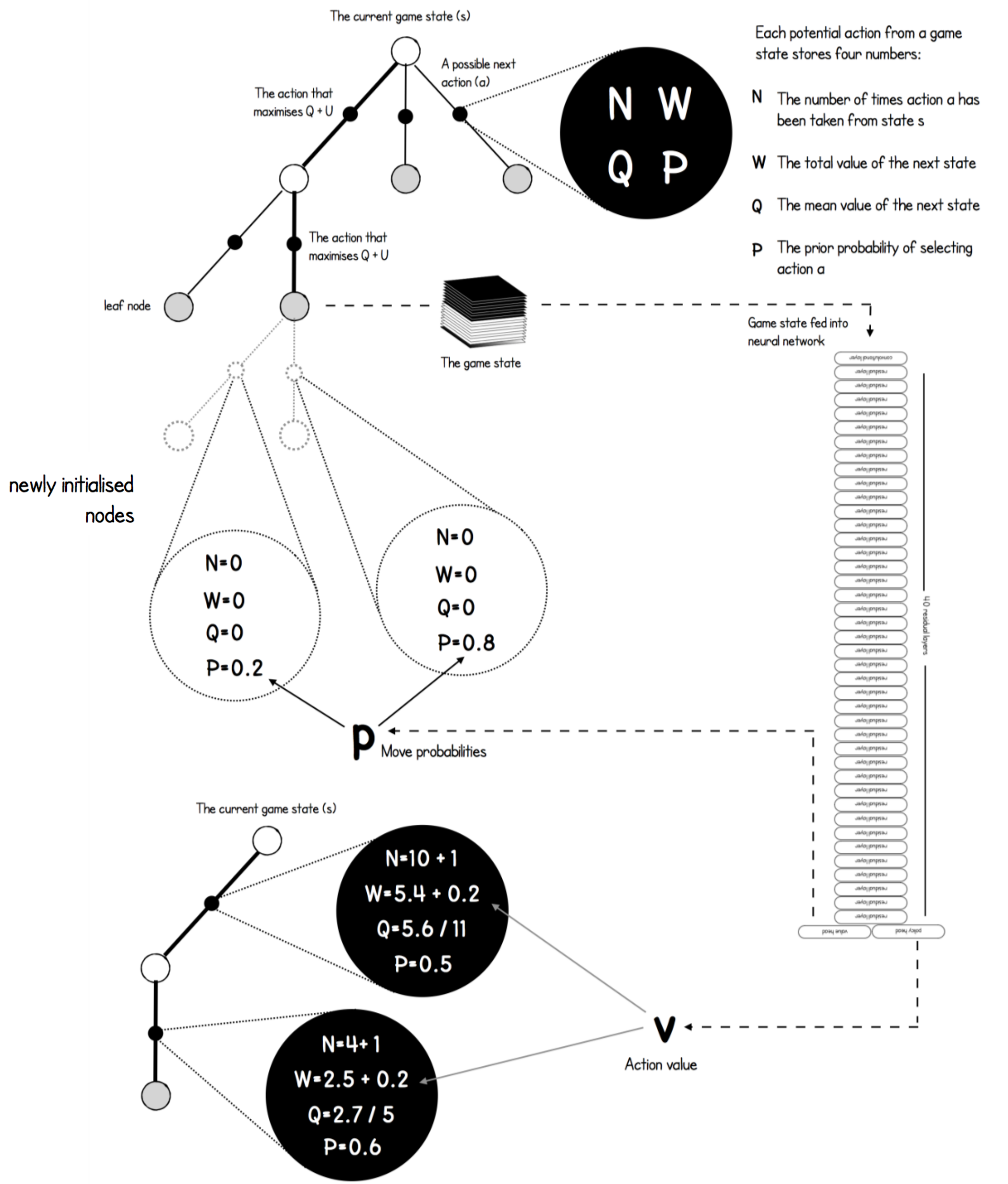
$$Q(s_t, a_t) = \frac{W(s_t, a_t)}{N(s_t, a_t)}$$

MCTS Play



d) Once the search is complete, search probabilities π are returned, proportional to $N(s, a)^{1/\tau}$, where $N(s, a)$ is the visit count of each move a from the root state s and τ is a parameter controlling temperature.

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


The current game state (s)
A possible next action (a)
The action that maximises Q + U
N W
Q P
Each potential action from a game state stores four numbers:
N The number of times action a has been taken from state s
W The total value of the next state
Q The mean value of the next state
P The prior probability of selecting action a
The action that maximises Q + U
leaf node
The game state
Game state fed into neural network
newly initialised nodes
N=0
W=0
Q=0
P=0.2
N=0
W=0
Q=0
P=0.8
P Move probabilities
The current game state (s)
N=10 + 1
W=5.4 + 0.2
Q=5.6 / 11
P=0.5
N=4+ 1
W=2.5 + 0.2
Q=2.7 / 5
P=0.6
V
Action value

First, run the following simulation 1,600 times...

Start at the root node of the tree (the current game state)

1. Choose the action that maximises...

$$Q + U$$

The mean value of the next state

A function of **P** and **N** that increases if an action hasn't been explored much, relative to the other actions, or if the prior probability of the action is high

Early on in the simulation, U dominates (more exploration), but later, Q is more important (less exploration)

2. Continue until a leaf node is reached

The game state of the leaf node is passed into the neural network, which outputs predictions about two things:

p Move probabilities

v Value of the state (for the current player)

The move probabilities p are attached to the new feasible actions from the leaf node

3. Backup previous edges

Each edge that was traversed to get to the leaf node is updated as follows:

$$N \rightarrow N + 1$$
$$W \rightarrow W + v$$
$$Q = W / N$$

...then select a move

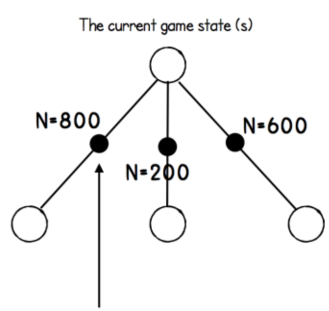
After 1,600 simulations, the move can either be chosen:

Deterministically (for competitive play)
Choose the action from the current state with greatest **N**

Stochastically (for exploratory play)
Choose the action from the current state from the distribution

$$\pi \sim N^{1/\tau}$$

where τ is a temperature parameter, controlling exploration



Choose this move if deterministic

If stochastic, sample from categorical distribution π with probabilities (0.5, 0.125, 0.375)

Other points

- The sub-tree from the chosen move is retained for calculating subsequent moves
- The rest of the tree is discarded

Results

Results

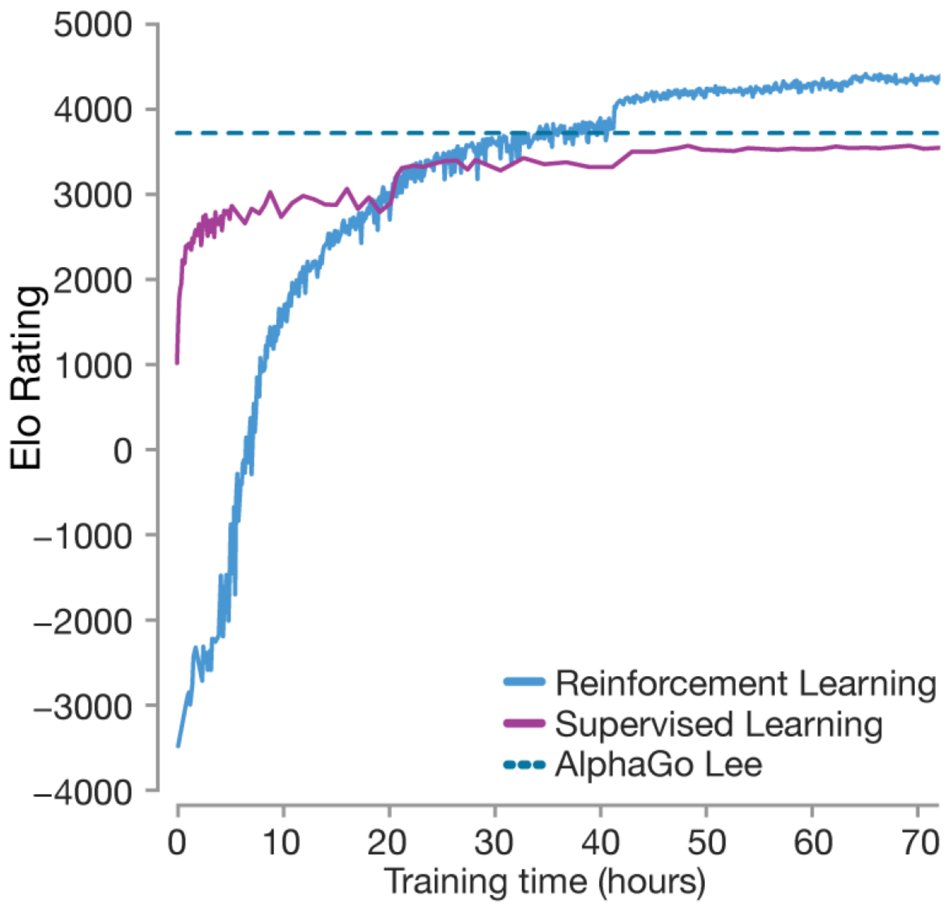
Surprisingly, AlphaGo Zero outperformed AlphaGo Lee after just 36 hours;

For comparison, AlphaGo Lee was trained over several months.

AlphaGo Zero used a single machine with 4 Tensor Processing Units (TPUs) 29, while AlphaGo Lee was distributed over many machines and used 48 TPUs.

AlphaGo Zero defeated AlphaGo Lee by 100 games to 0

Results



Elo ratings were computed from evaluation games between different players, using 0.4 seconds of thinking time per move.
For comparison, a similar player trained by supervised learning from human data, using the KGS data-set, is also shown

Thanks for your attention! ☺