AlphaGo Zero Introduction

Kangkun Mao

mkk@hust.edu.cn

May 31, 2018

Contents

- The key points of AlphaGo Zero
  - Monte Carlo tree search
  - One neural network produces both policies and value
  - Self-play reinforcement learning
- Solving the Rubiks Cube
  - Autodidactic Iteration
  - MCTS Solver
- Discussion
  - Protein folding
  - RNA structure prediction

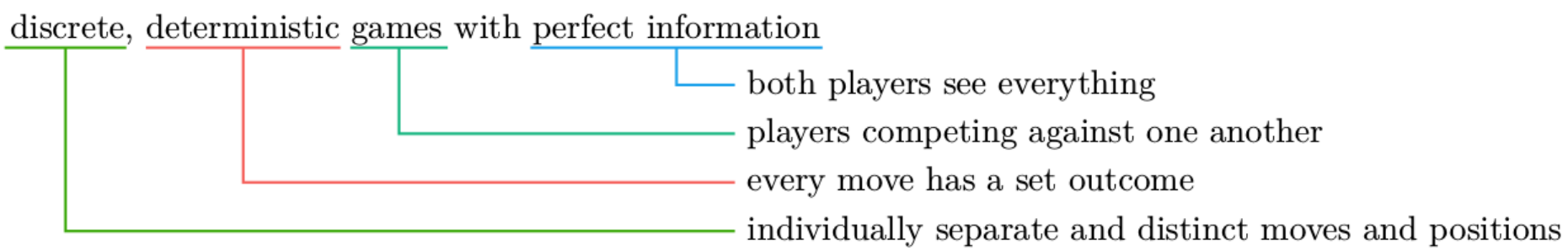
Monte Carlo Tree Search

Combinatorial Game

Satisfies the conditions:

- Zero-Sum
- Fully Information
- Determinism
- Sequential
- Discrete

- Nash Equilibrium
  - Monte Carlo Tree Search
  - Counterfactual Regret Minimization
- Black Box Optimization
  - Monte Carlo Tree Search
  - Evolutionary Algorithms
  - Bayesian Optimization



Monte Carlo Tree Search

Definition

which is a heuristic search algorithm for some kinds of decision processes, typically move planning in combinatorial games. It combines the generality of random simulation with the precision of tree search.

MCTS deals with large trees effectively by sampling many paths down the game tree. This means it repeatedly goes down many, but not all. As it tries more paths, it gains better estimates for which paths are good. Each one of these sample trials is called a Monte Carlo simulation.

The process can be broken down into the four phases

- Selection
- Expansion
- Simulation
- Backpropagation

Basic Algorithm

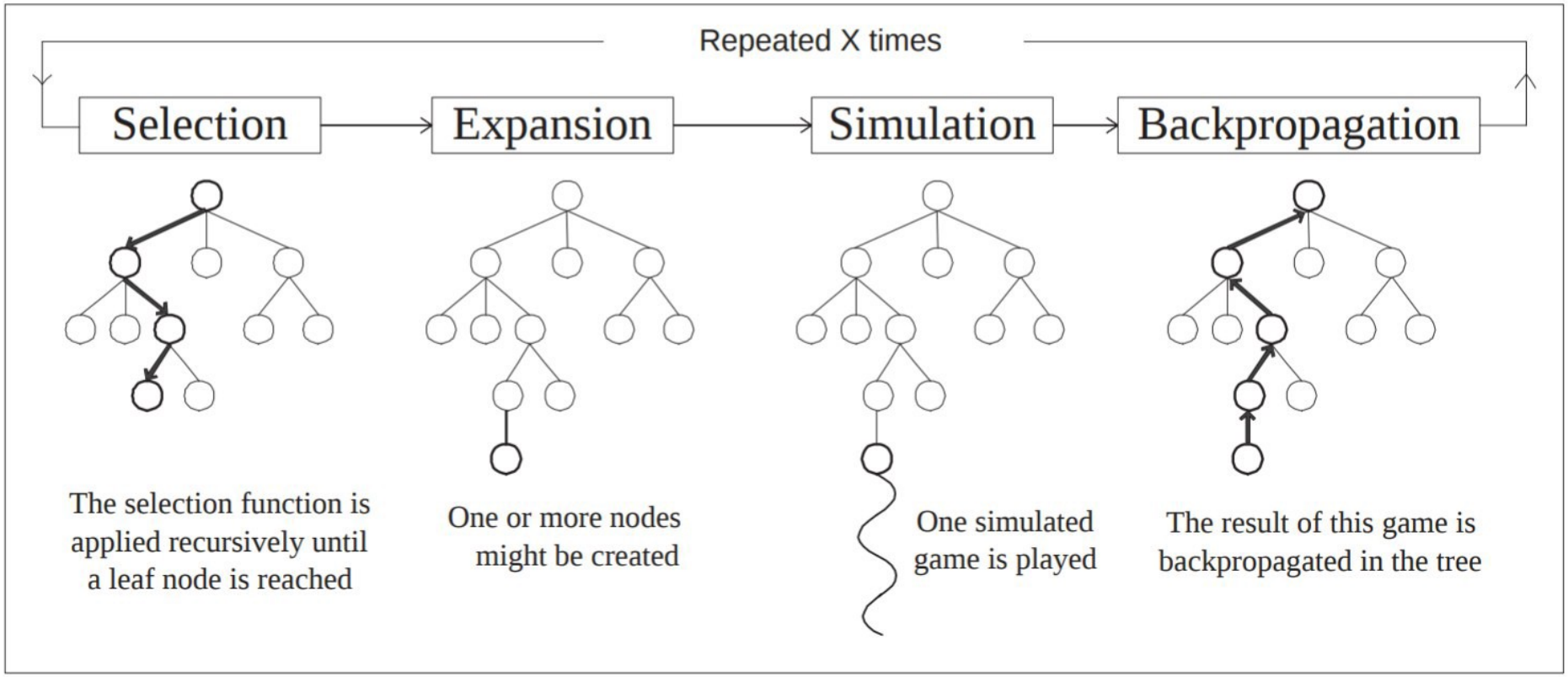
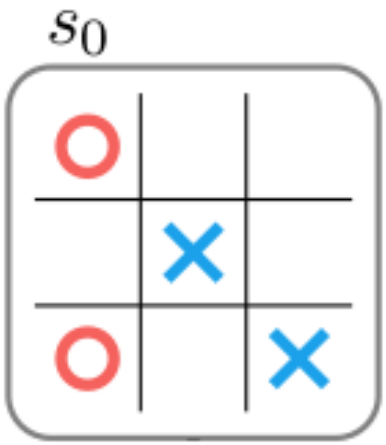


Figure: Monte Carlo tree search procedure, diagram from Chaslot (2006)

MCTS in Detail

Take a game of tic-tac-toe as example

- initial state s_0
- a set of actions $\mathcal{A}$
- terminal state: $\{win, loss, tie\}$
- score: $\{+1, -1, 0\}$



Phase 2: Expansion

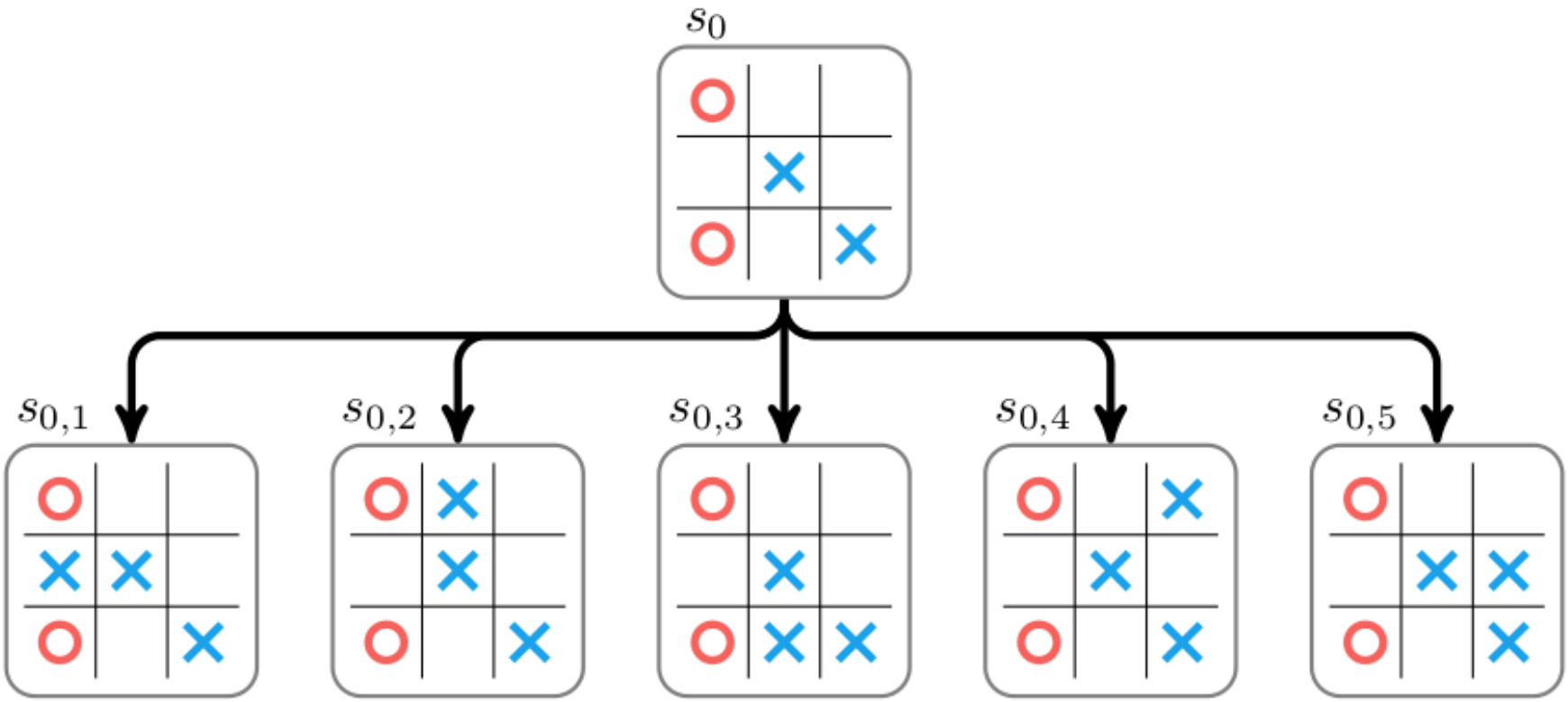


Figure: This node is expanded by trying every action $a \in \mathcal{A}$

Phase 3: Simulation

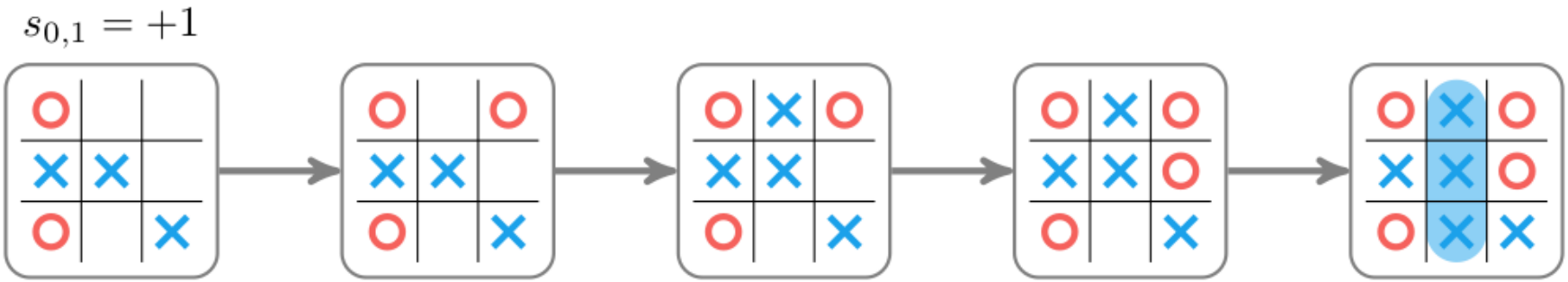


Figure: The child node is rolled out by randomly taking moves from the child state until a win, loss, or tie is reached

Phase 4: Backpropagation

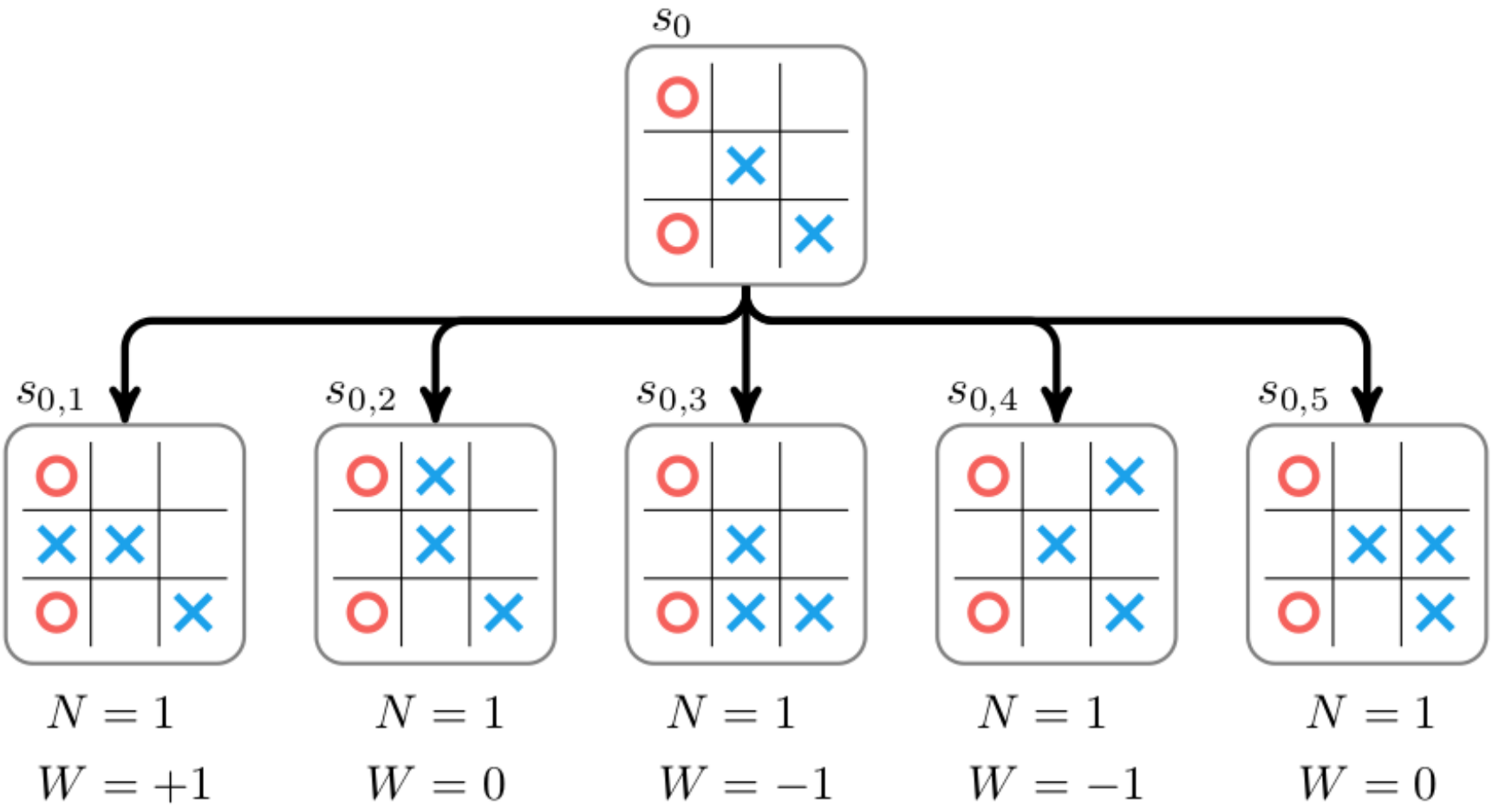


Figure: The expanded tree with approximate values for each child node

Phase 4: Backpropagation

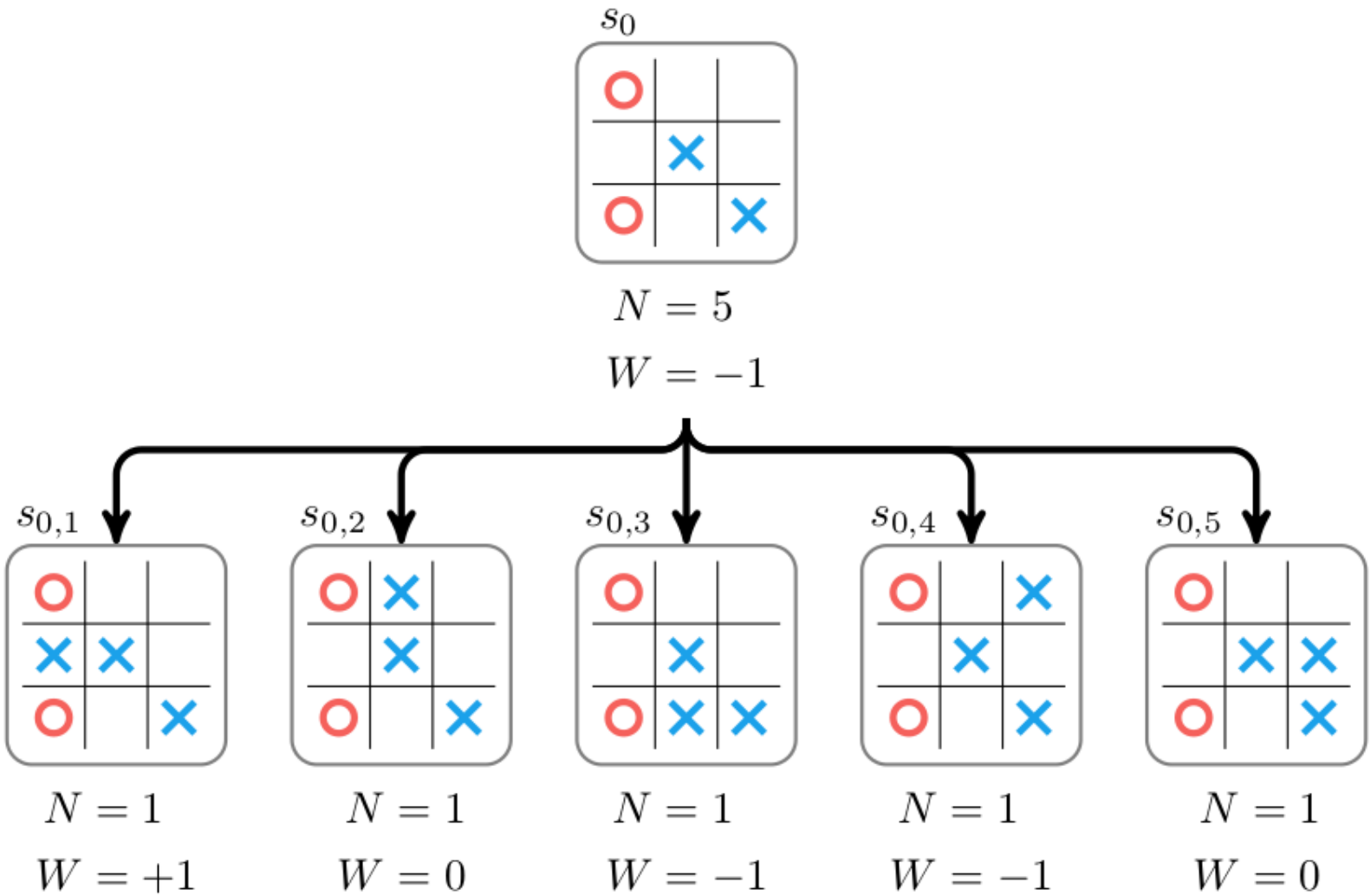


Figure: The information from the child nodes is then propagated back up the tree by increasing the parent's value and visit count

Credit Assignment Problem

Monte Carlo tree search does not expand all leaf nodes, as that would be very expensive. Instead, the selection process chooses nodes that strike a balance between being lucrative – having high estimated values, and being relatively unexplored – having low visit counts.

UCT Score

A leaf node is selected by traversing down the tree from the root node, always choosing the child i with the highest upper confidence tree (UCT) score:

$$U_i = \frac{W_i}{N_i} + c\sqrt{\frac{\ln N_p}{N_i}}$$

where

W_i is the accumulated value of the ith child

N_i is the visit count for ith child

N_p is the number of visit counts for the parent node

The parameter $c \geq 0$ controls the tradeoff between choosing lucrative nodes (low c) and exploring nodes with low visit counts (high c)

Phase 1: Selection

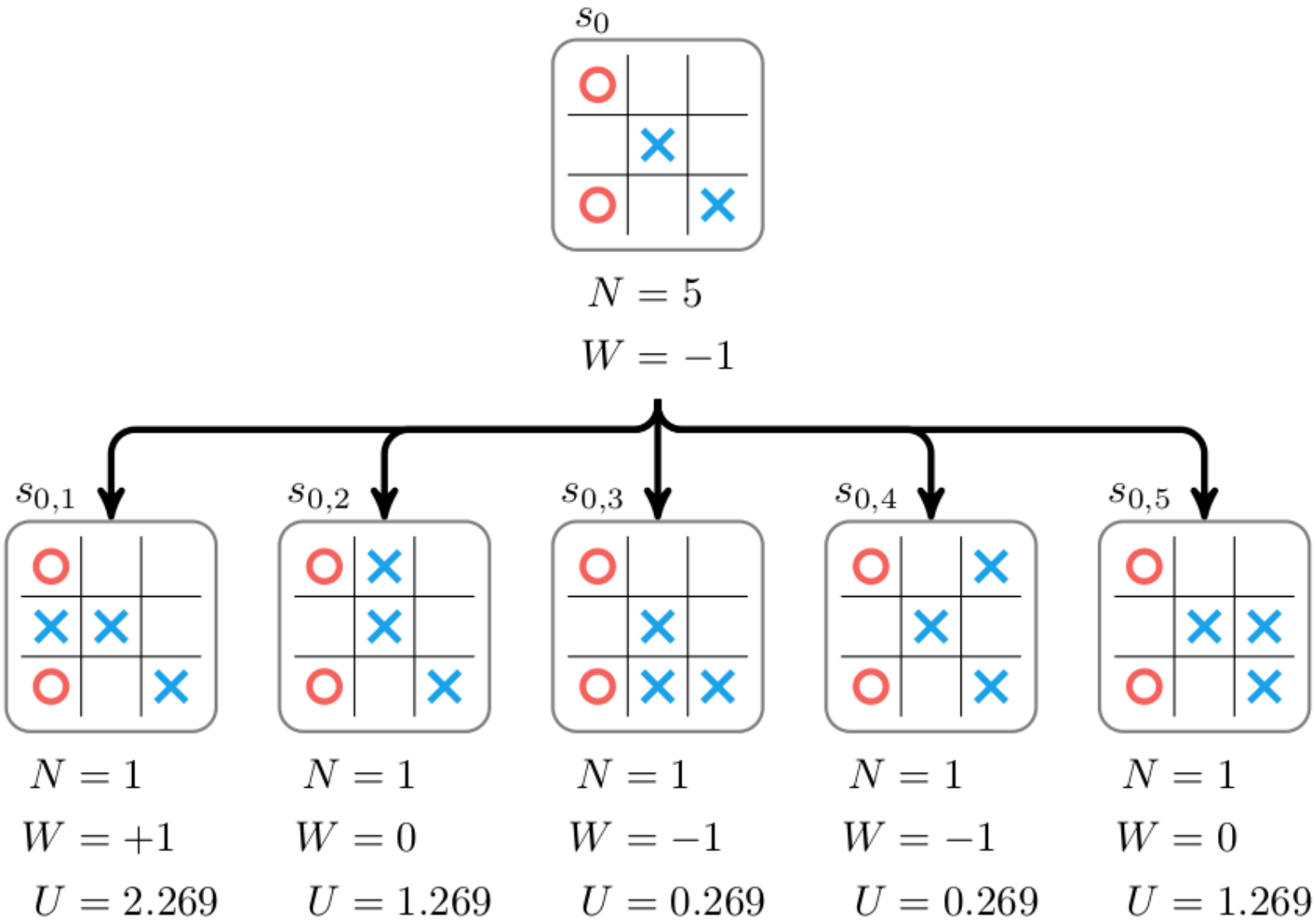


Figure: The UCT scores for the tic-tac-toe tree with $c = 1$

Phase 2 to 4

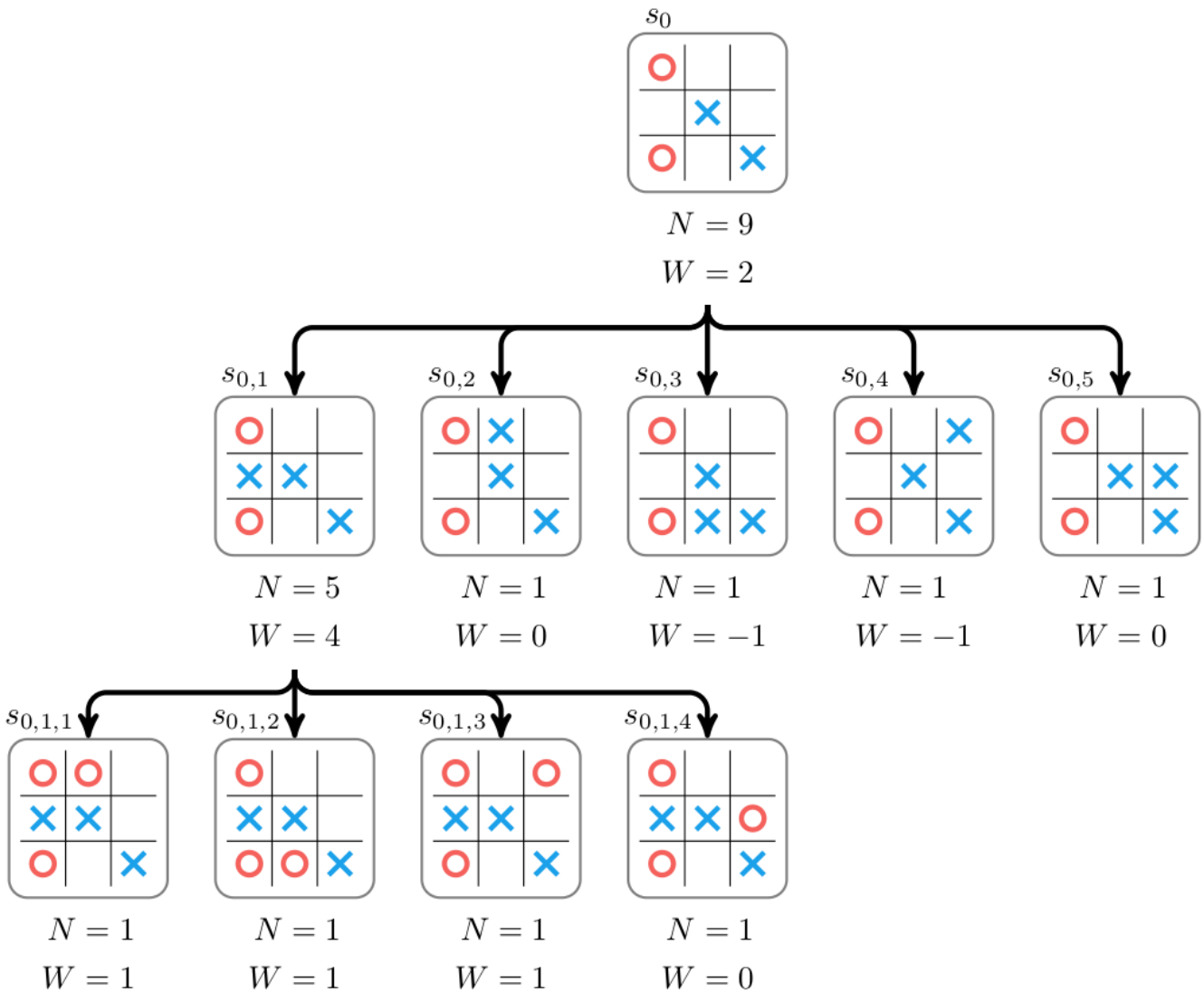


Figure: That node is expanded and the values are propagated back up

Iterations

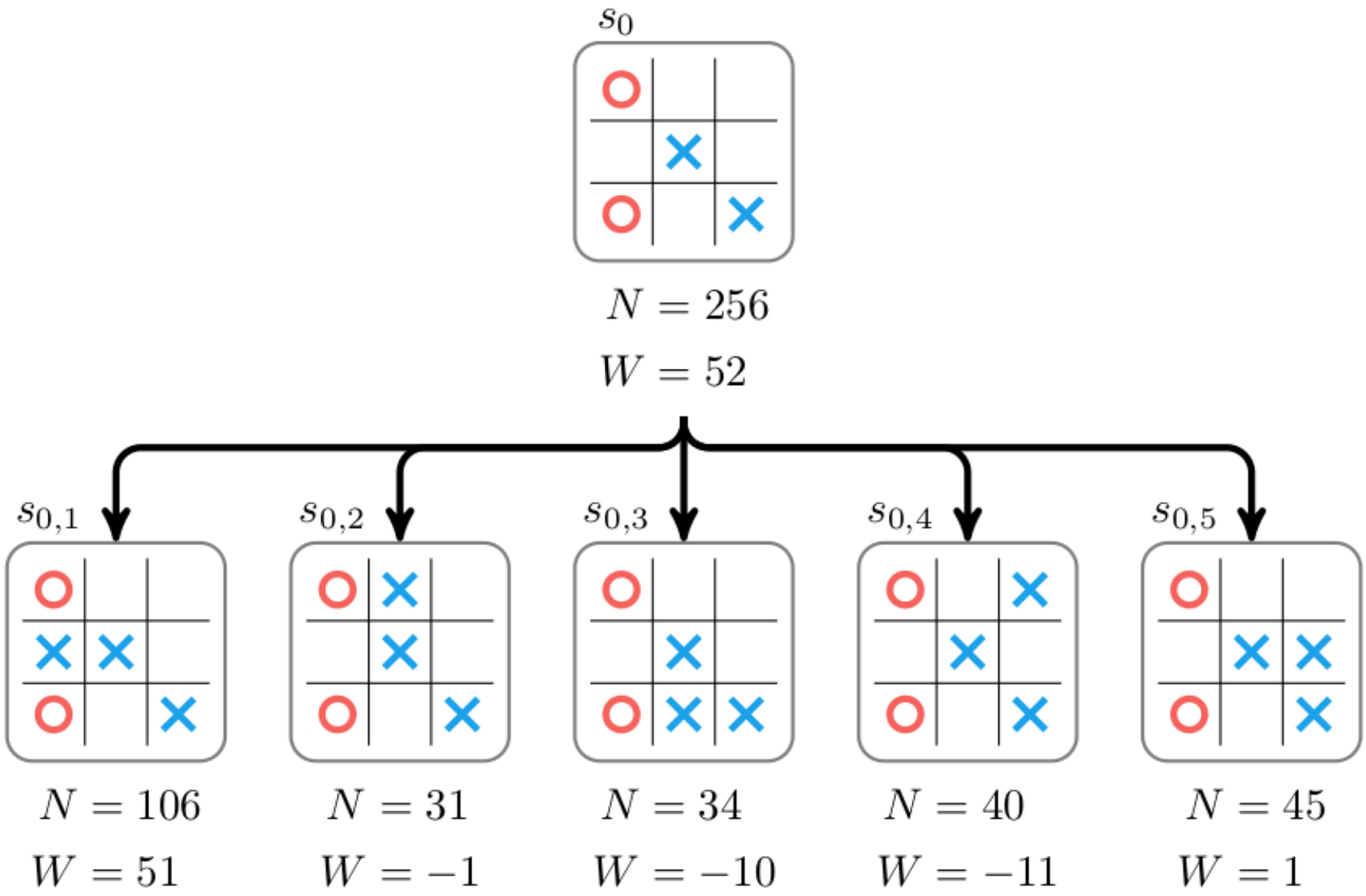


Figure: The tree is expanded and explore the possible moves, identifying the best move to take

Efficiency Through Expert Policies

Games like chess and Go have very large branching factors. there are an estimated 10^{46} board states in chess, and Go played on a traditional 19x19 board has around 10^{170} (Tic-tac-toe only has 5478 states).

Move evaluation with vanilla Monte Carlo tree search just isn't efficient enough.

Suppose we have an expert policy π that, for a given state s, tells us how likely an expert-level player is to make each possible action.

Expert Policies

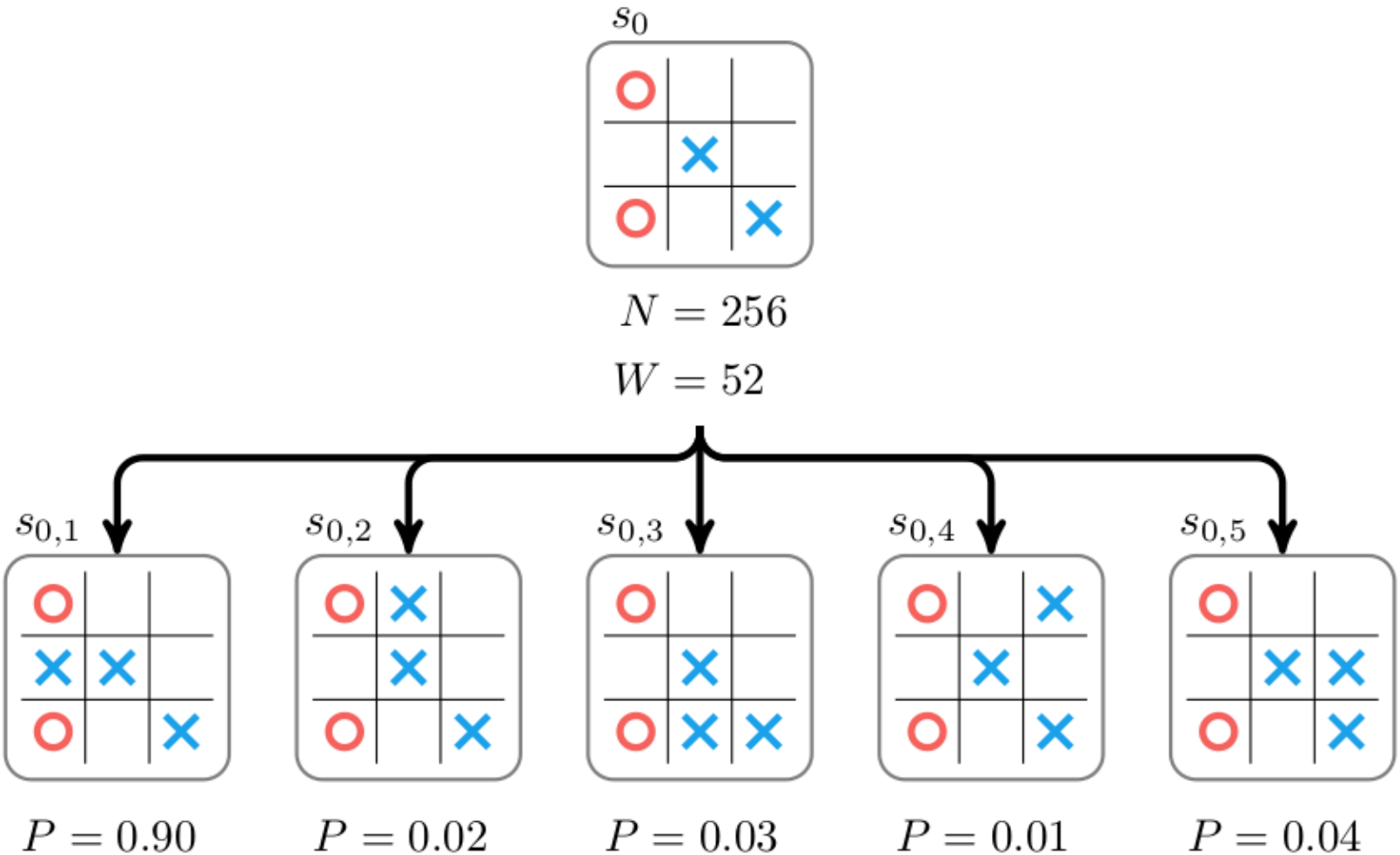


Figure: $P_i = \pi(a_i|s_0)$ is the probability of choosing the ith action a_i given the root state s_0

Expert Policies

A modified form of Monte Carlo tree search will store the probability of each node according to the policy, and this probability is used to adjust the node's score during selection.

The probabilistic upper confidence tree score used by DeepMind is:

$$U_i = \frac{W_i}{N_i} + cP_i\sqrt{\frac{\ln N_p}{1+N_i}}$$

Now, node exploration is guided by the expert policy, biasing exploration towards moves the expert policy considers likely.

Efficiency Through Value Approximation

A second form of efficiency can be achieved by avoiding expensive and potentially inaccurate random rollouts.

- Use the expert policy from the previous section to guide the random rollout
- Avoid rollouts altogether, directly approximate the value of a state with a value approximator function $\hat{W}(x)$

$\hat{W}(x)$ takes a state and directly computes a value in $[-1, 1]$, without conducting rollouts.

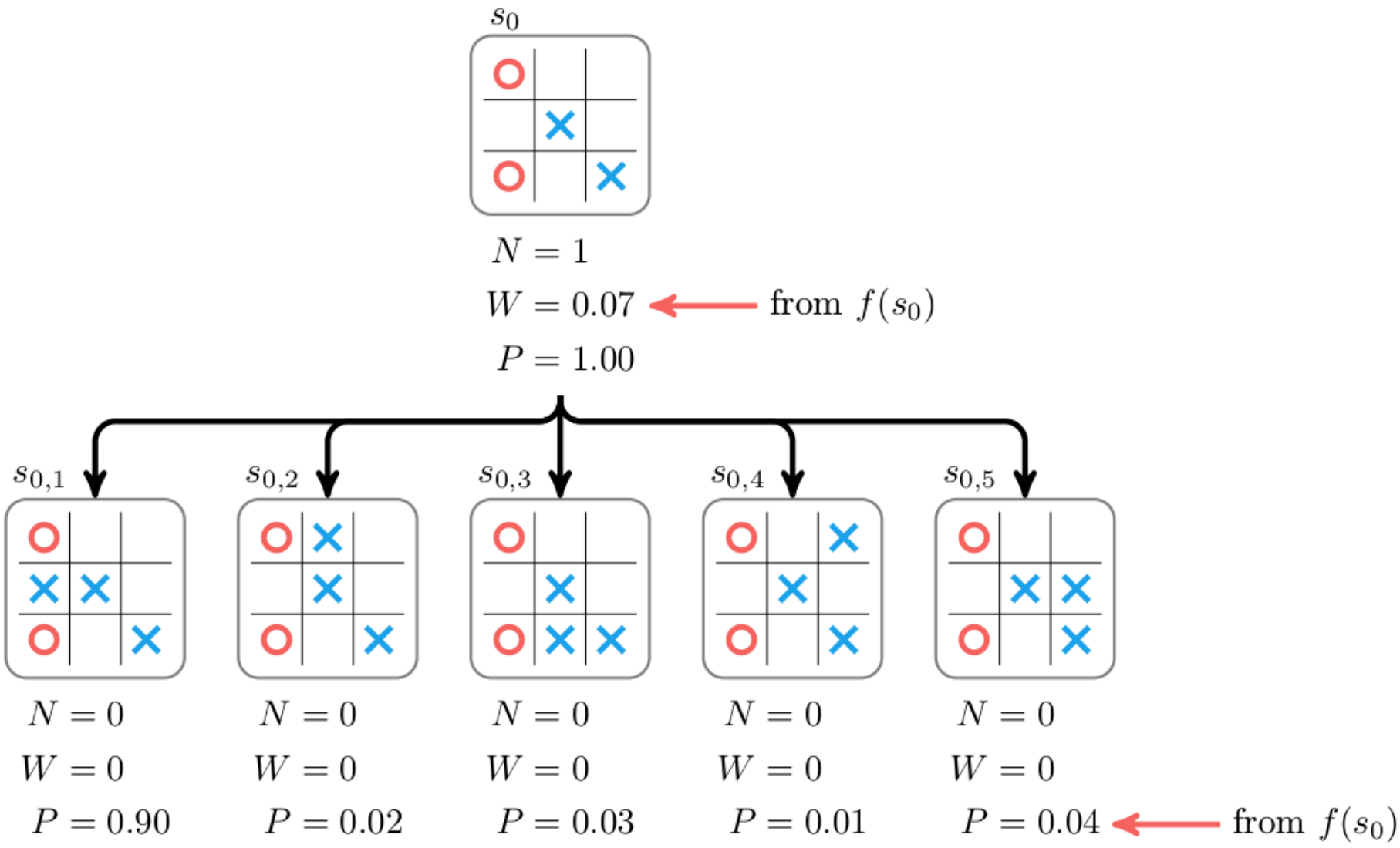
The Alpha Zero Neural Net

Alpha Zero uses one neural network f that takes in the game state and produces both the probabilities over the next move and the approximate state value:

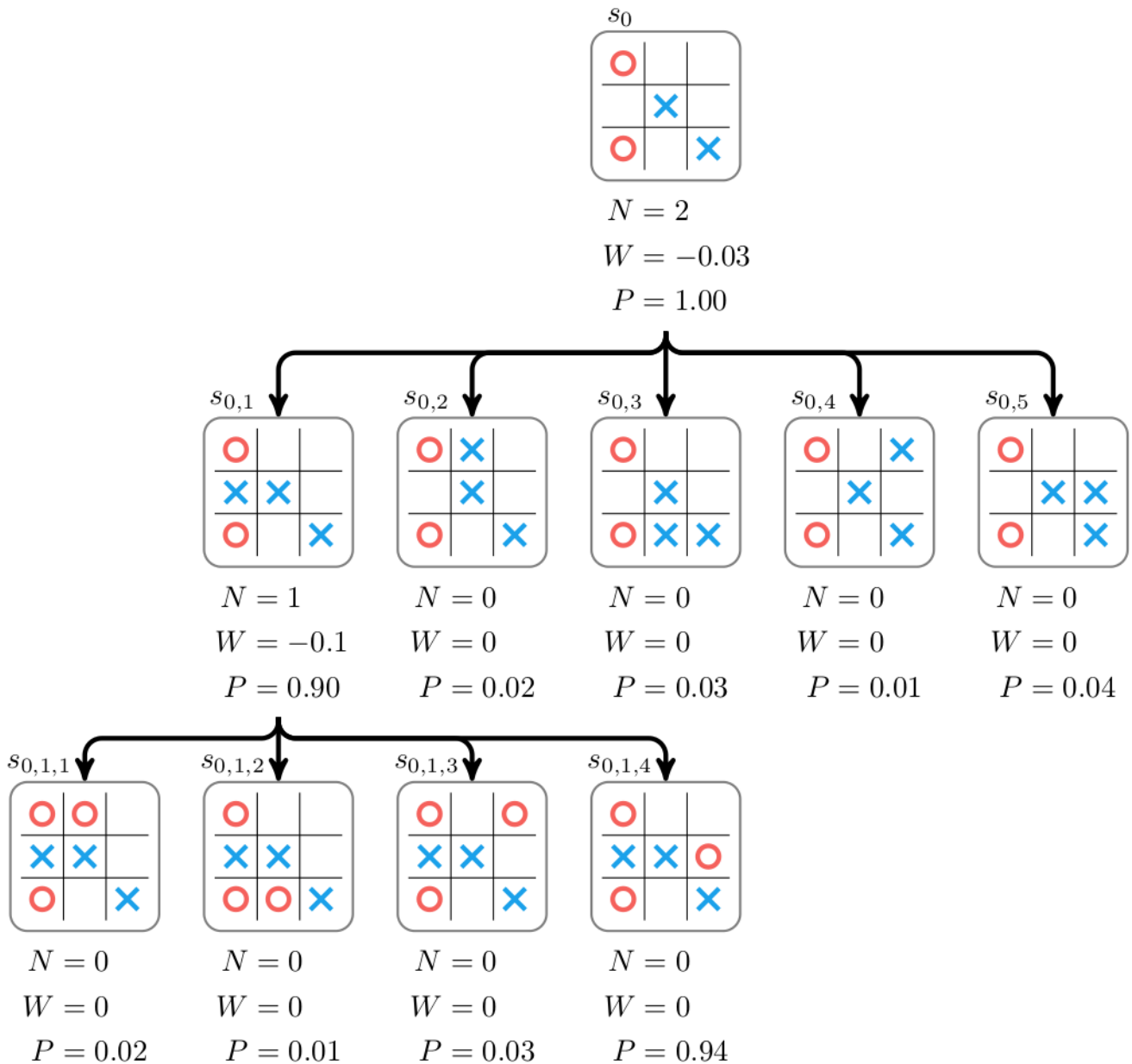
$$f(s) \rightarrow [\vec{P}, W]$$

Leaves in the search tree are expanded by evaluating them with the neural network. Each child is initialized with $N = 0$, $W = 0$ and with P corresponding to the prediction from the network. The value of the expanded node is set to the predicted value and this value is then backed up the tree.

The Alpha Zero Neural Net



The Alpha Zero Neural Net



The Alpha Zero Neural Net

The core idea of the Alpha Zero algorithm is that the predictions of the neural network can be improved, and the play generated by Monte Carlo tree search can be used to provide the training data.

The policy portion of the neural network is improved by training the predicted probabilities P for s_0 to match the improved probability π obtained from running Monte Carlo tree on s_0:

$$\pi_i = N_i^{1/\tau}$$

The value portion of the neural network is improved by training the predicted value to match the eventual win/loss/tie result of the game, Z.

The Loss Function

The Alpha Zero neural network's loss function is:

$$(w - z)^2 + \boldsymbol{\pi}^\intercal \ln \mathbf{p} + \lambda ||\boldsymbol{\theta}||_2^2$$

where

$(w - z)^2$ is the value loss

$\boldsymbol{\pi}^\intercal \ln \mathbf{p}$ is the policy loss

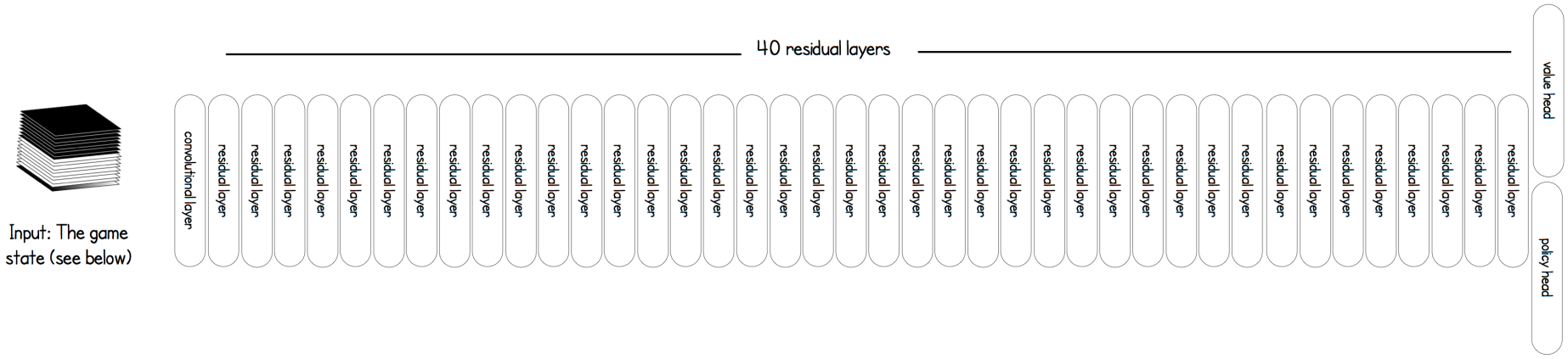
$\lambda ||\boldsymbol{\theta}||_2^2$ is an extra regularization term with parameter $\lambda \geq 0$ and θ represents the parameters in the neural network.

AlphaGo Zero

The Deep Neural Network Architecture

The network learns 'tabula rasa'(from a blank slate)

At no point is the network trained using human knowledge or expert moves



What Is A Game State

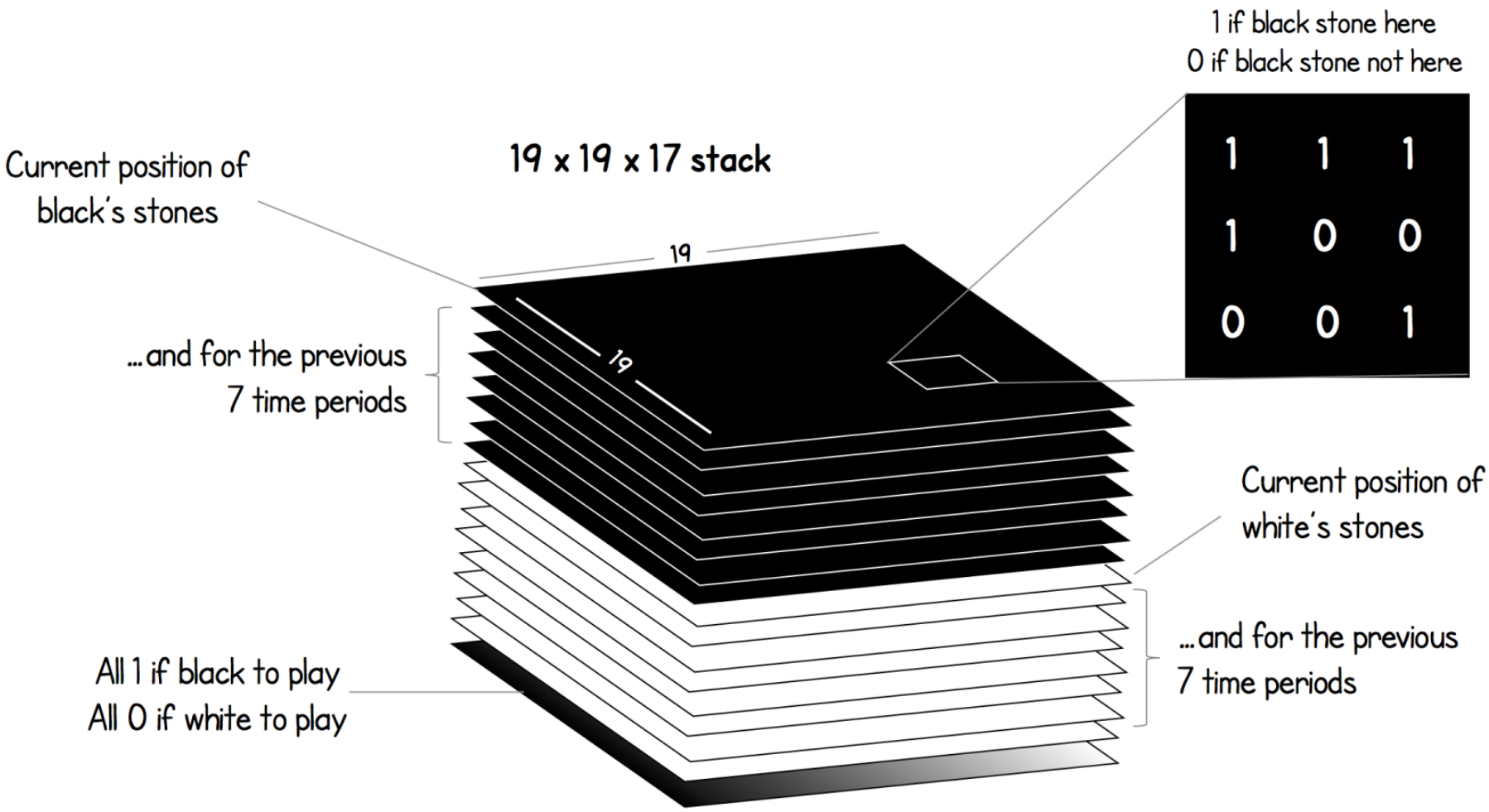
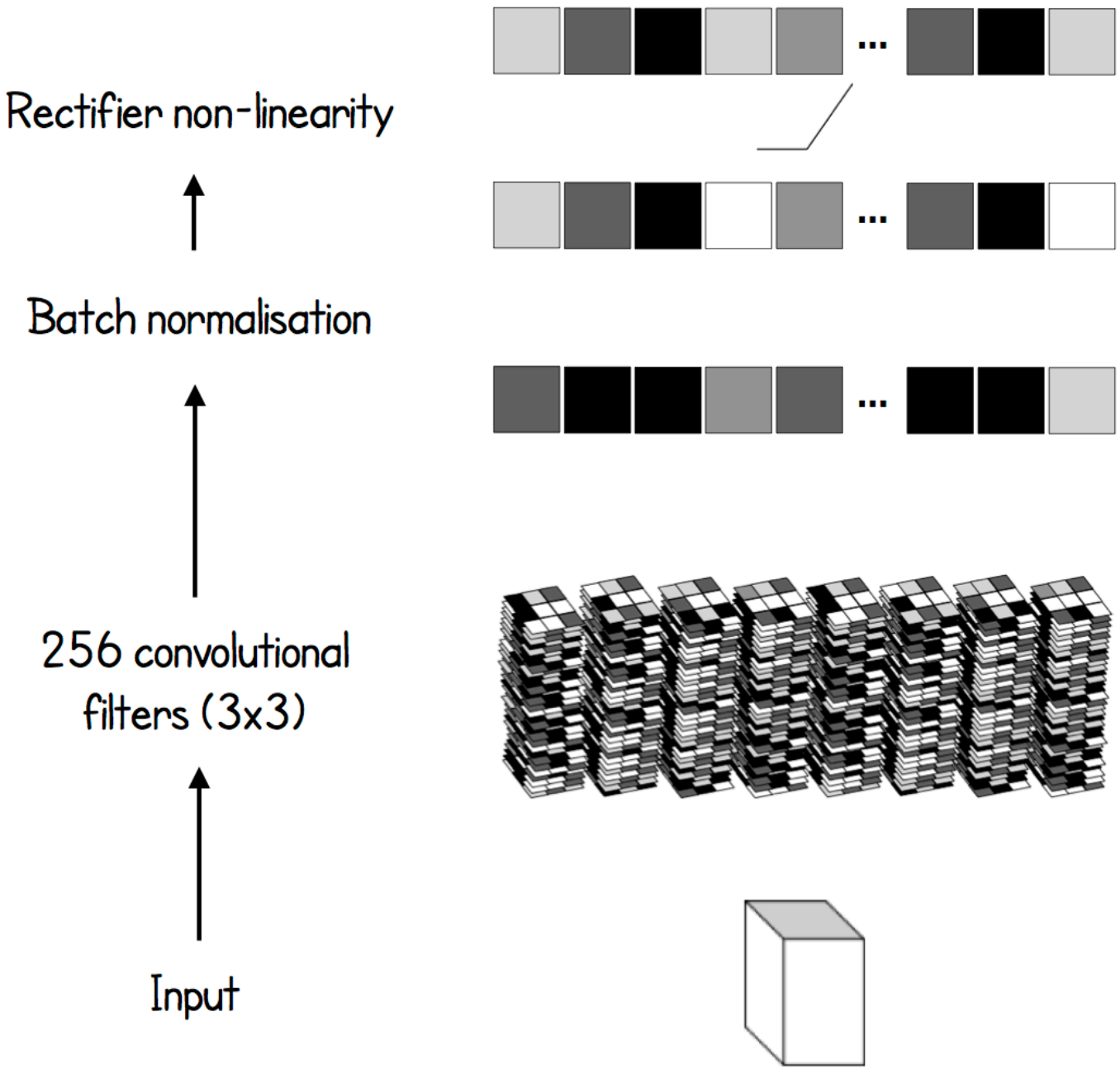
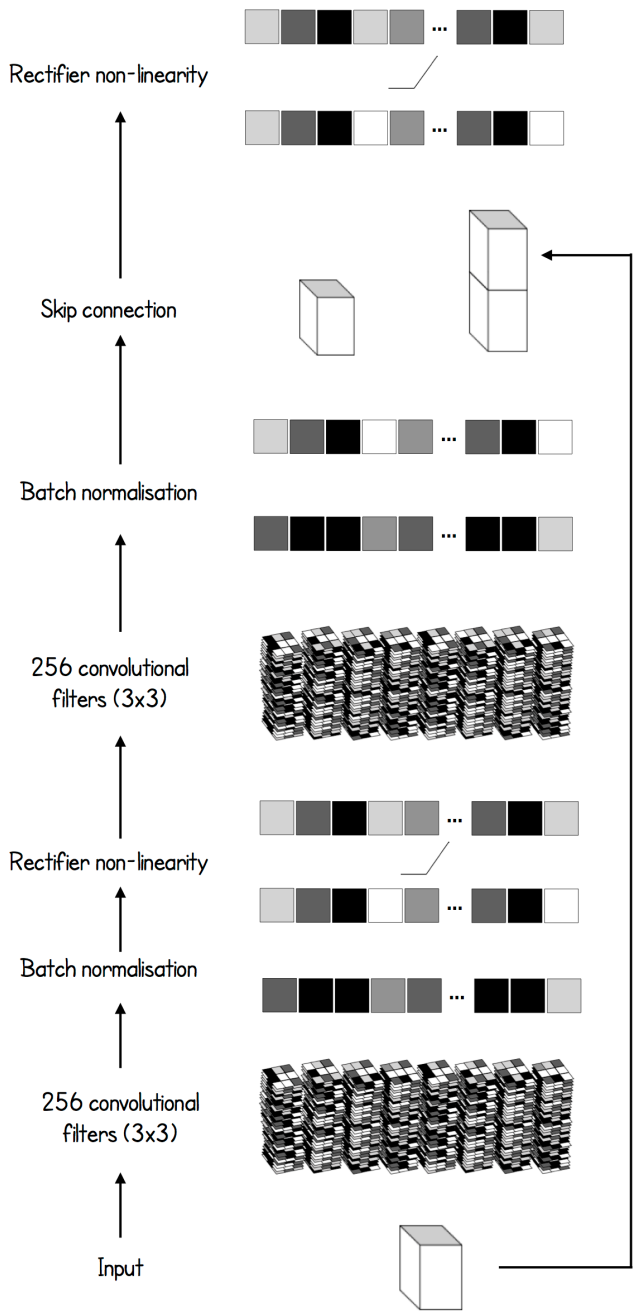


Figure: The stack is the input to the deep neural network

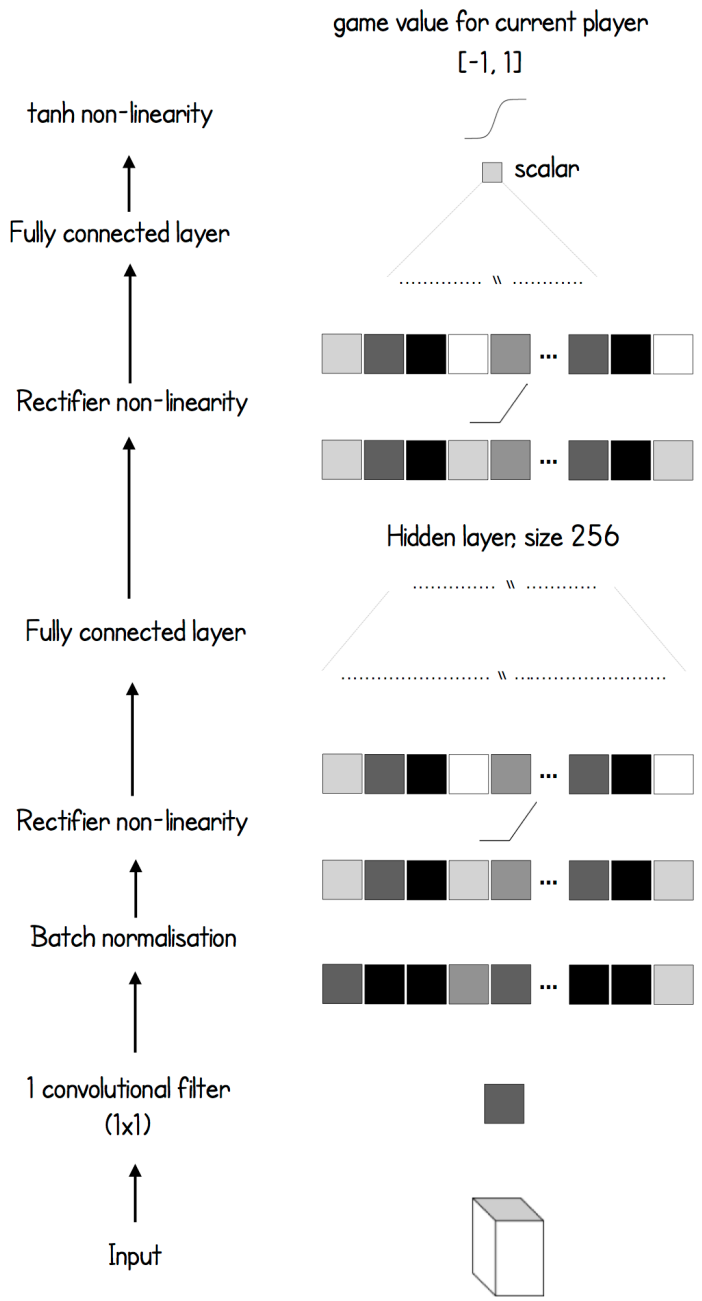
Convolutional Layer



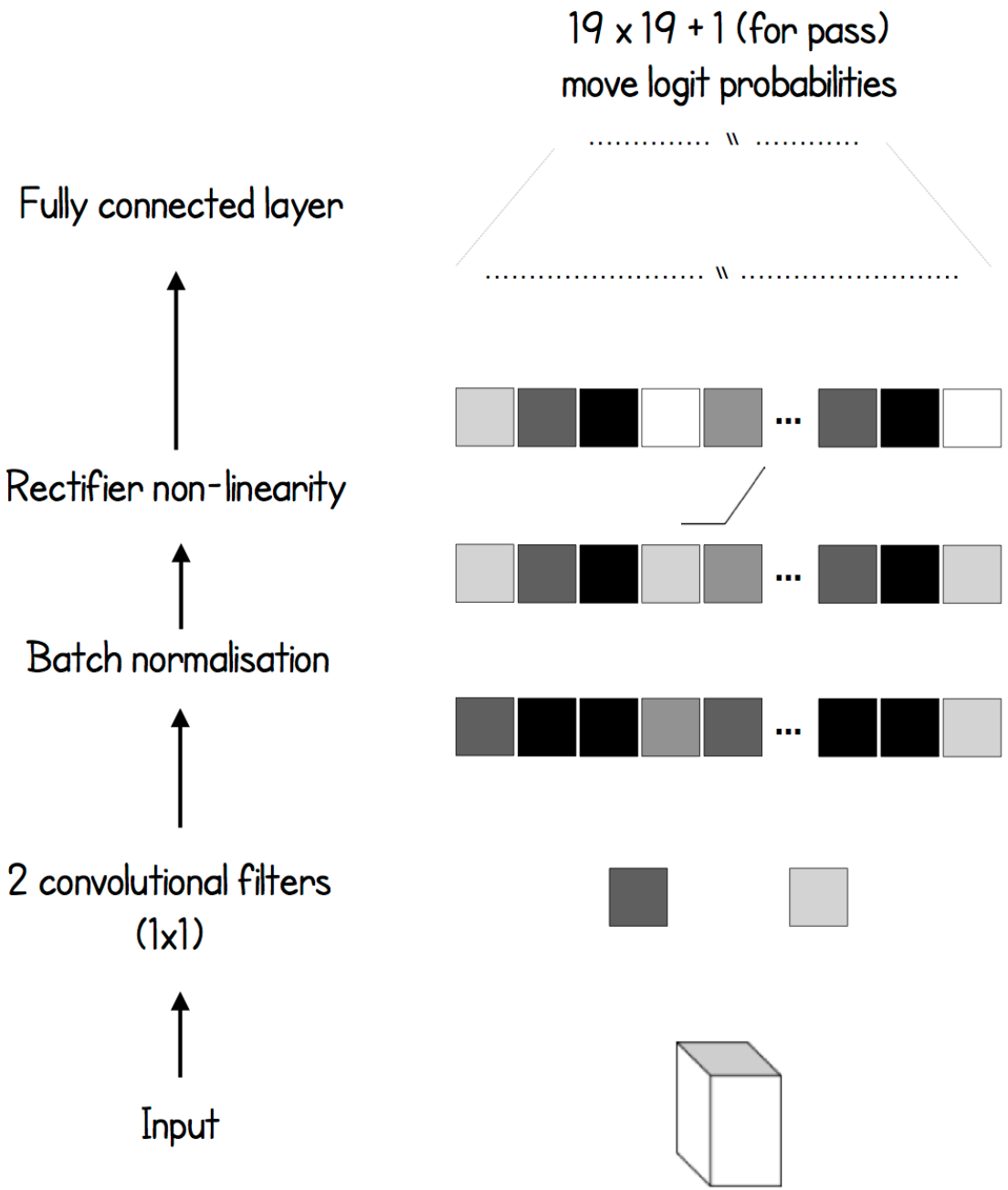
Residual Layer



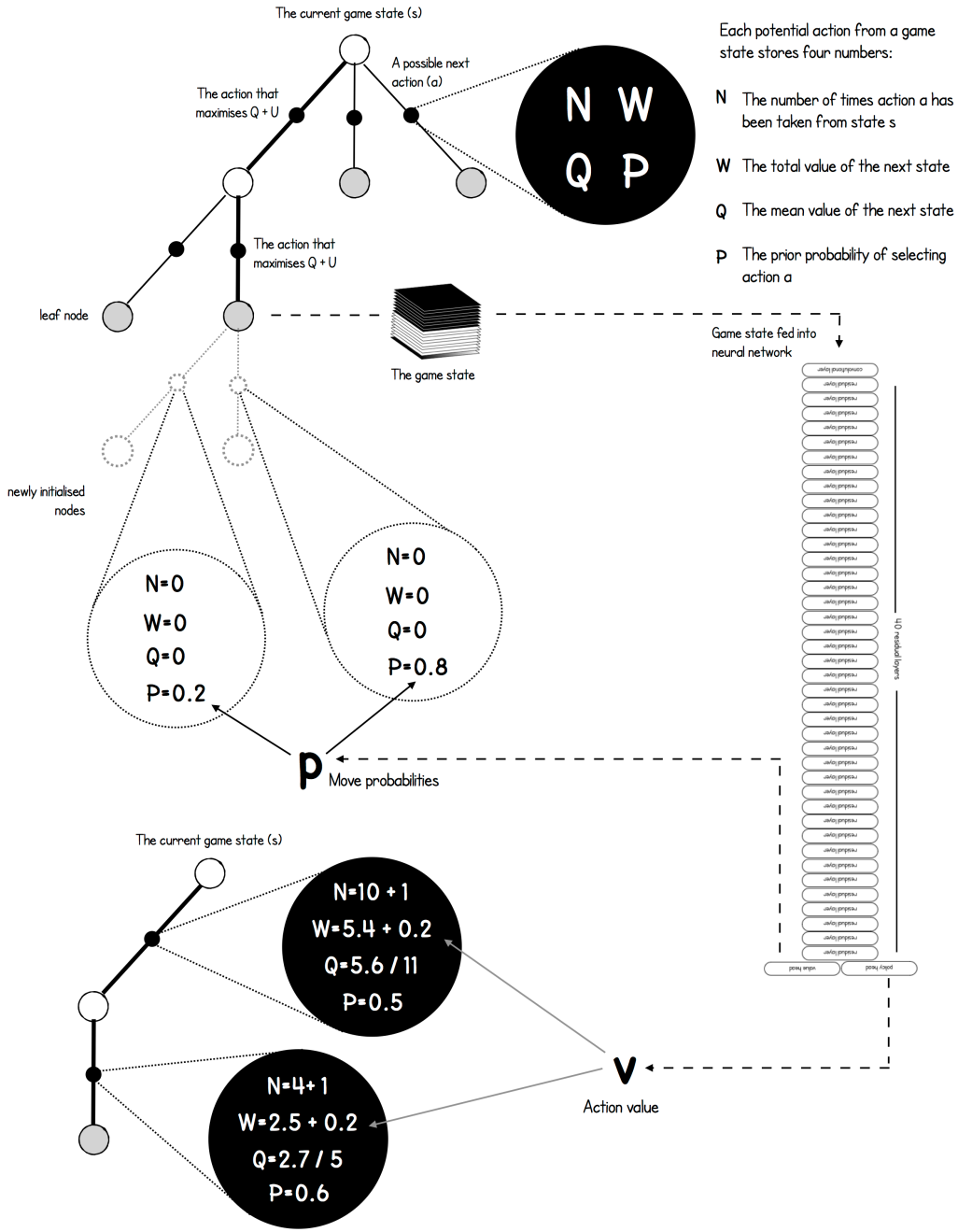
The Value Head



The Policy Head



How AlphaGo Zero Chooses Its Next Move



How AlphaGo Zero Chooses Its Next Move

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)

How AlphaGo Zero Chooses Its Next Move

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

How AlphaGo Zero Chooses Its Next Move

3. Backup previous edges

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

$$N \rightarrow N + 1$$

$$W \rightarrow W + v$$

$$Q = W / N$$

How AlphaGo Zero Chooses Its Next Move

...then select a move

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

Deterministically (for competitive play)

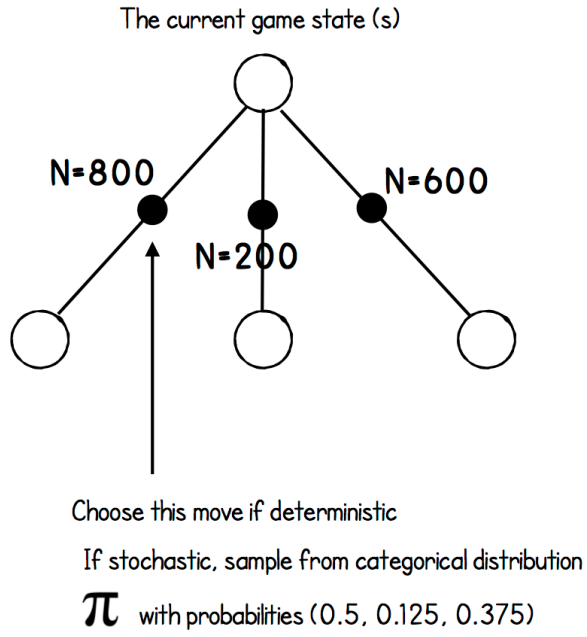
Choose the action from the current state with greatest **N**

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



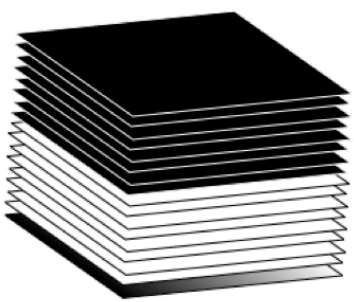
Self Play

Create a 'training set'

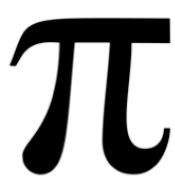
The best current player plays 25,000 games against itself

See MCTS section to understand how AlphaGo Zero selects each move

At each move, the following information is stored



The game state
(see 'What is a Game State section')



The search probabilities
(from the MCTS)

The winner
(+1 if this player won, -1 if this player lost - added once the game has finished)

Retrain Network

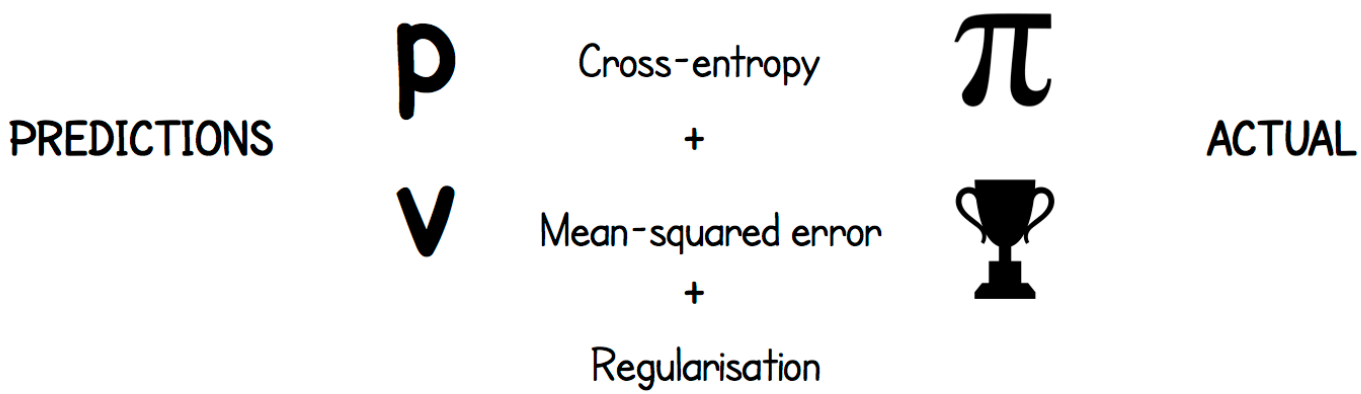
Optimise the network weights

A TRAINING LOOP

Sample a mini-batch of 2048 positions from the last 500,000 games

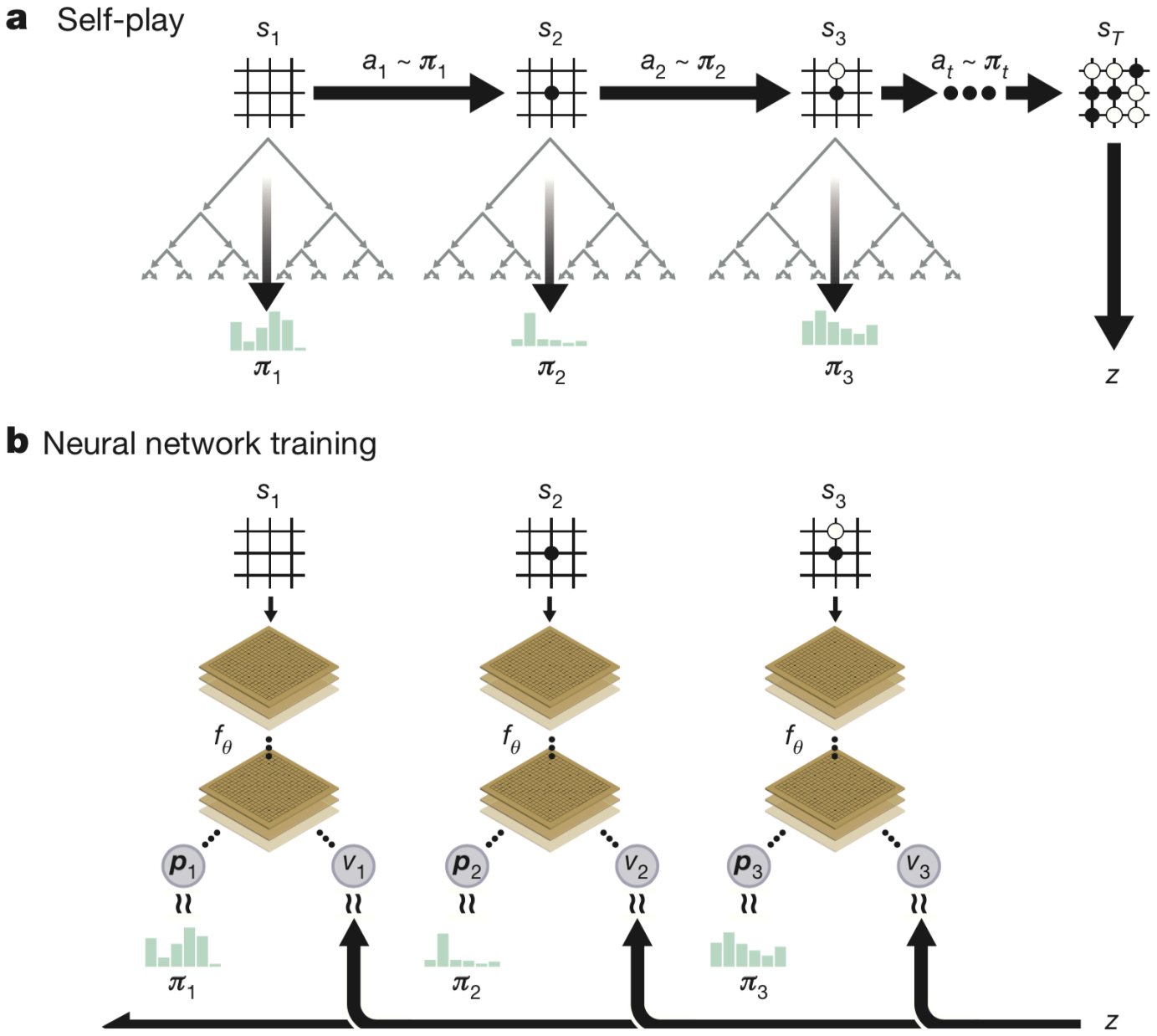
Retrain the current neural network on these positions
– The game states are the input (see 'Deep Neural Network Architecture')

Loss function
Compares predictions from the neural network with the search probabilities and actual winner



After every 1,000 training loops, evaluate the network

Reinforcement Learning



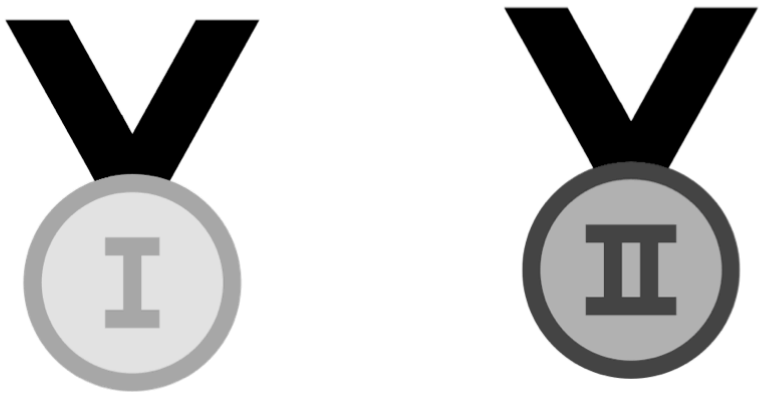
Evaluate Network

Test to see if the new network is stronger

Play 400 games between the latest neural network and the current best neural network

Both players use MCTS to select their moves, with their respective neural networks to evaluate leaf nodes

Latest player must win 55% of games to be declared the new best player



Rubiks Cube Problem

Rubik's Cube

- The 3x3x3 Rubik's cube is a classic 3-Dimensional combination puzzle
- 6 faces, or 3x3x1 planes, which can be rotated 90° in either direction
- The goal state is reached when all stickers on each face of the cube are the same color

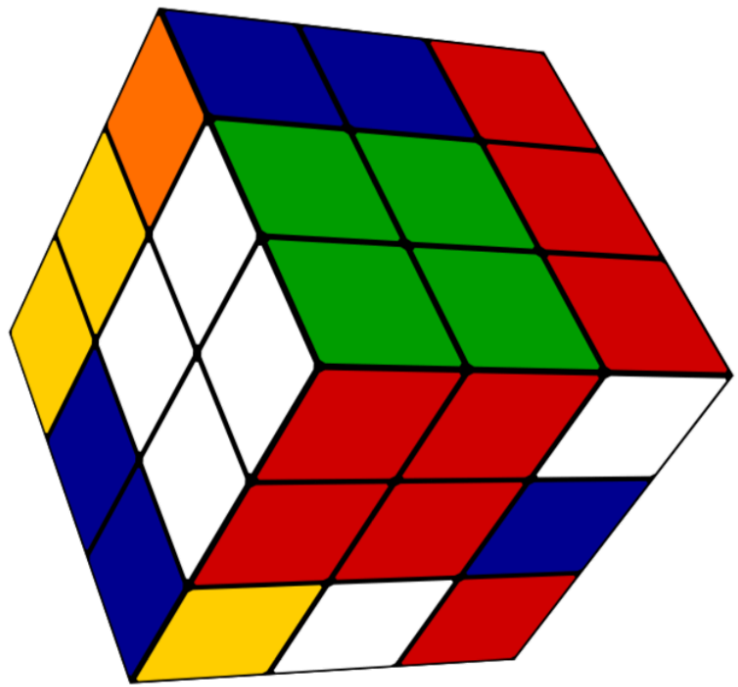


Figure: 3x3x3 Rubik's cube

Problems

- Large state space
  With approximately $4.3x10^{19}$ different possible configurations
- Single one reward state
  Out of all of these configurations, only one state has a reward signal: the goal state

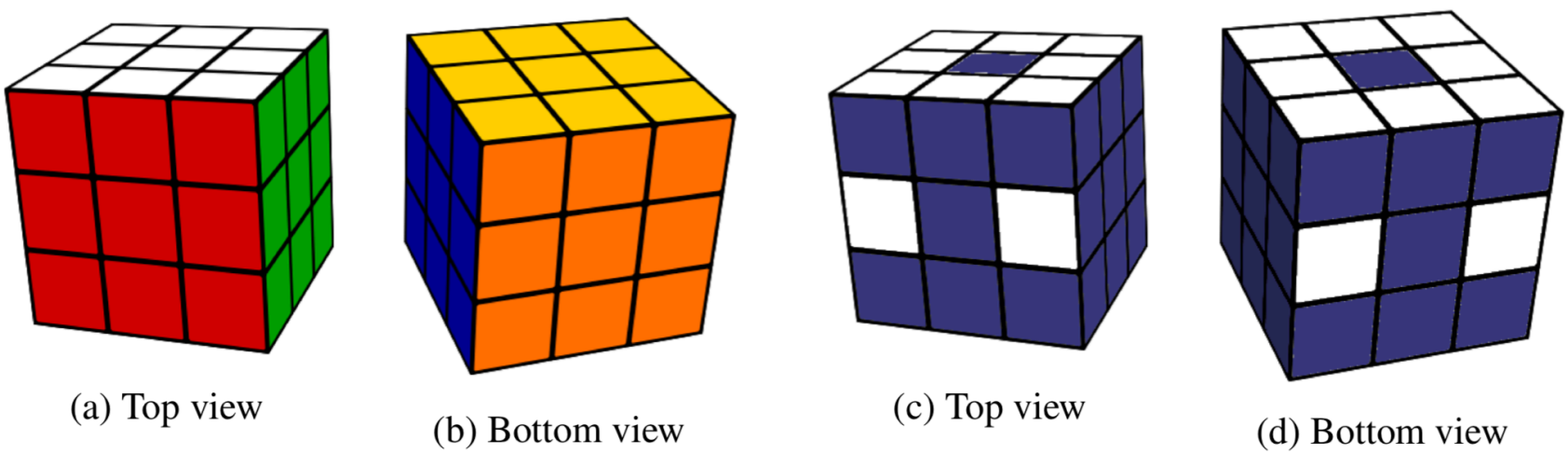
Difficulties

Current DRL(deep reinforcement learning) algorithms struggle in environments with a high number of states and
a small number of reward states.

Starting from random states and applying DRL algorithms, such as asynchronous advantage actor-critic (A3C), could theoretically result in the agent never solving the cube and never receiving a learning signal.

States

- The Rubiks Cube consists of 26 smaller cubes called cubelets
- These are classified by their sticker count
- One-hot encoding for the 54 stickers to represent their location on the cube

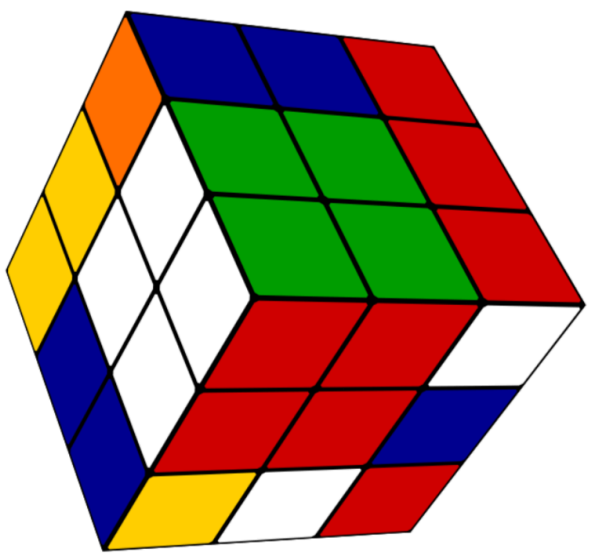


(a) Top view (b) Bottom view (c) Top view (d) Bottom view

Figure: (a) and (b) show the solved cube as it appears in the environment. (c) and (d) show the cube reduced in dimensionality for input into the DNN. Stickers that are used by the DNN are white, whereas ignored stickers are dark.

States

- The position of one sticker on a cubelet determines the position of the remaining stickers on that cubelet
- Ignore the redundant center cubelets and only store the 24 possible locations for the edge and corner cubelets



$$\longrightarrow \quad 20 \begin{bmatrix} 0 & 1 & \cdots & 0 \\ 0 & 0 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \cdots & 0 \end{bmatrix}$$

$$24$$

20x24 State Representation

Actions

F, B, L, R, U, and D correspond to turning the front, back, left, right, up, and down faces.

A clockwise rotation is represented with a single letter, whereas a letter followed by an apostrophe represents a counter-clockwise rotation.

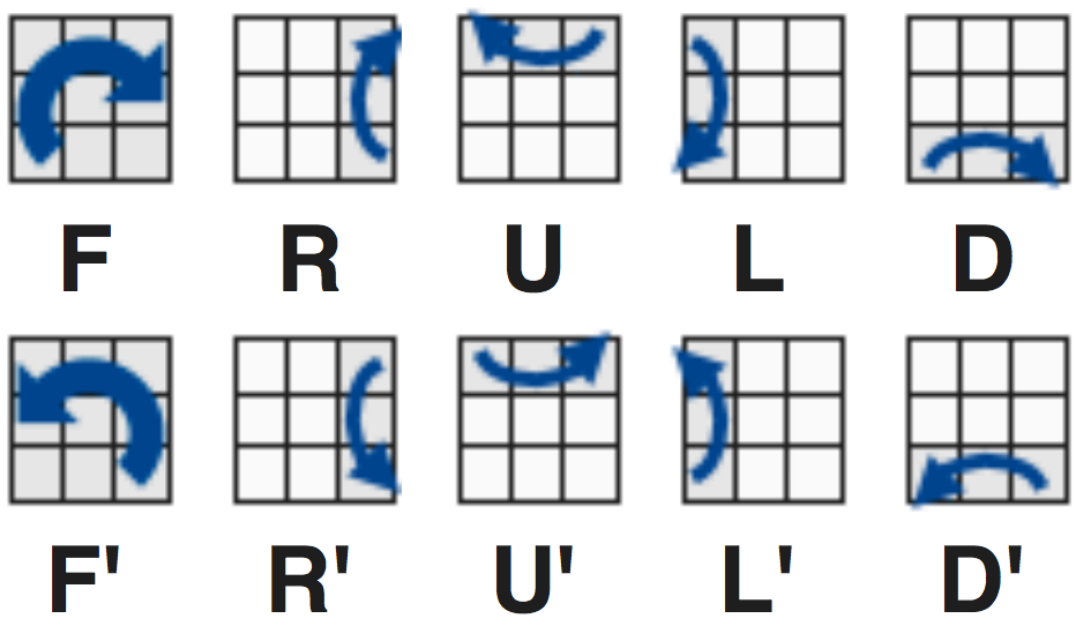


Figure: Moves notation

Model environment

The Rubik's Cube environment consists of

- A set of $4.3x10^{19}$ states $\mathcal{S}$
- One special state, s_{solved}, representing the goal state
- At each timestep, t, the agent
  - observes a state $s_t \in \mathcal{S}$
  - takes an action $a_t \in \mathcal{A}$ with $\mathcal{A} := \{F, F', \cdots, D, D'\}$
  - observes a new state $s_{t+1} = A(s_t, a_t)$
  - receives a scalar reward $R(s_{t+1})$, which is 1 if s_{t+1} is s_{solved} and -1 otherwise

Methods

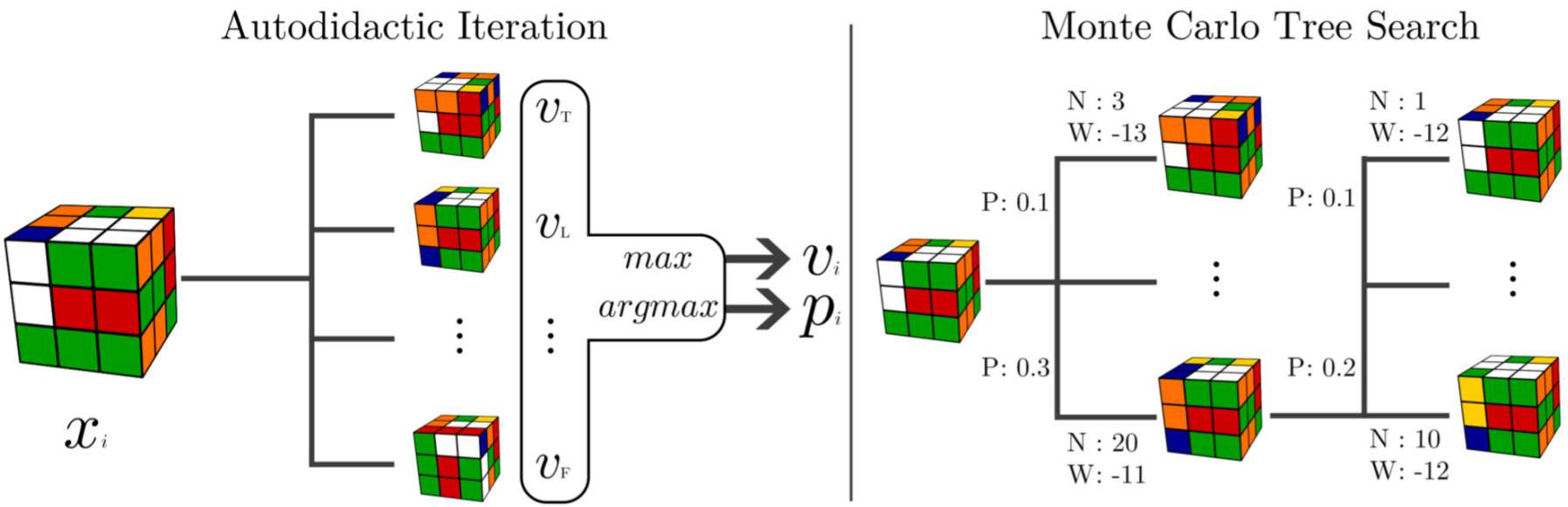


Figure: An illustration of DeepCube. The training and solving process is split up into ADI and MCTS. First, iteratively train a DNN by estimating the true value of the input states using breadth-first search. Then, using the DNN to guide exploration, solve cubes using Monte Carlo Tree Search.

Autodidactic Iteration

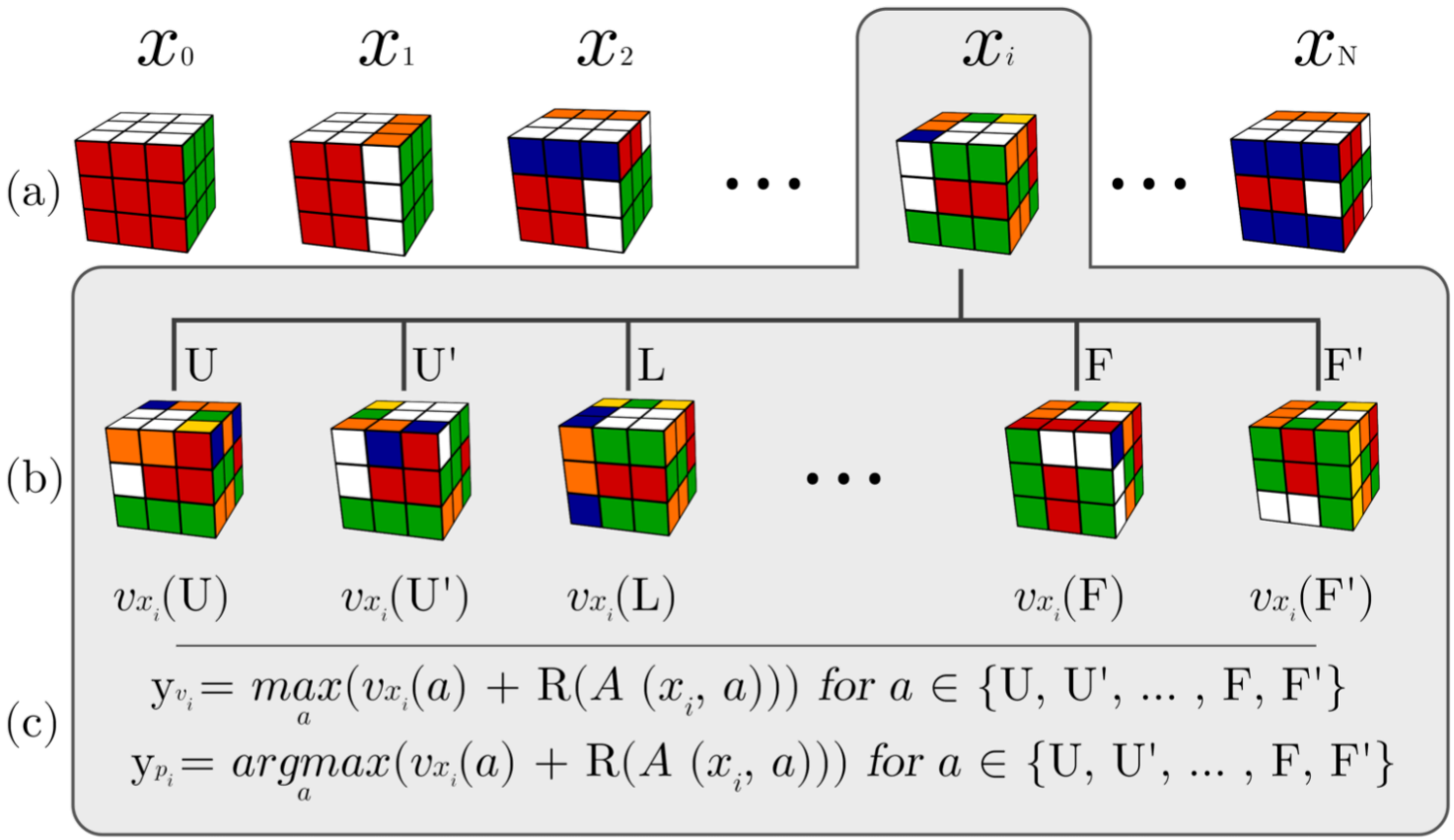


Figure: Visualization of training set generation in ADI. (a) Generate a sequence of training inputs starting from the solved state. (b) For each training input, generate its children and evaluate the value network on all of them. (c) Set the value and policy targets based on the maximal estimated value of the children.

Autodidactic Iteration

Algorithm 1: Autodidactic Iteration

Initialization: θ initialized using Glorot initialization

repeat

  $X \leftarrow$ N scrambled cubes

  for $x_i \in X$ **do**

    for $\alpha \in \mathcal{A}$ **do**

      $(v_{x_i}(\alpha), p_{x_i}(\alpha)) \leftarrow f_\theta(A(x_i, \alpha))$

    $y_{v_i} \leftarrow \max_\alpha(R(A(x_i, \alpha)) + v_{x_i}(\alpha))$

    $y_{p_i} \leftarrow \arg\max_\alpha(R(A(x_i, \alpha)) + v_{x_i}(\alpha))$

    $Y_i \leftarrow (y_{v_i}, y_{p_i})$

  $\theta' \leftarrow train(f_\theta, X, Y)$

  $\theta \leftarrow \theta'$

until *iterations* $= M$

MCTS Solver

Algorithm 2: MCTS Solver

Input: a starting state s_0
Output: shortest path from s_0 to s_{solve}
repeat
  while s_t *is not leaf node* **do**
    $U_{s_t}(\alpha) \leftarrow cP_{s_t}\sqrt{\sum_{\alpha'} N_{s_t}(\alpha')}/(1+N_{s_t}(\alpha))$ /* Select */
    $Q_{s_t}(\alpha) \leftarrow W_{s_t}(\alpha) - L_{s_t}(\alpha)$
    $A_t \leftarrow \arg\max_\alpha U_{s_t}(\alpha) + Q_{s_t}(\alpha)$
    $s_t \leftarrow A(s_t, A_t)$
  $s_\tau \leftarrow s_t$
  foreach *child* $s' \in \{A(s_\tau, \alpha), \forall \alpha \in \mathcal{A}\}$ **do**
    $W_{s'}(\cdot), N_{s'}(\cdot), L_{s'}(\cdot) \leftarrow 0$ /* Expand */
    $P_{s'}(\cdot) \leftarrow p_{s'}$
  $(v_{s_\tau}, p_{s_\tau}) \leftarrow f_\theta(s_\tau)$
  for $t = \tau$ **to** 0 **do**
    $W_{s_t}(A_t) \leftarrow \max(W_{s_t}(A_t), v_{s_\tau})$ /* Backpropagate */
    $N_{s_t}(A_t) \leftarrow N_{s_t}(A_t) + 1$
    $L_{s_t}(A_t) \leftarrow L_{s_t}(A_t) - \nu$
until $s_\tau = s_{solve}$ **or** *iterations* $=$ *limit*
return $Path = \{A_t | 0 \le t \le \tau\} \leftarrow BFS(T)$

Nueral network architecture

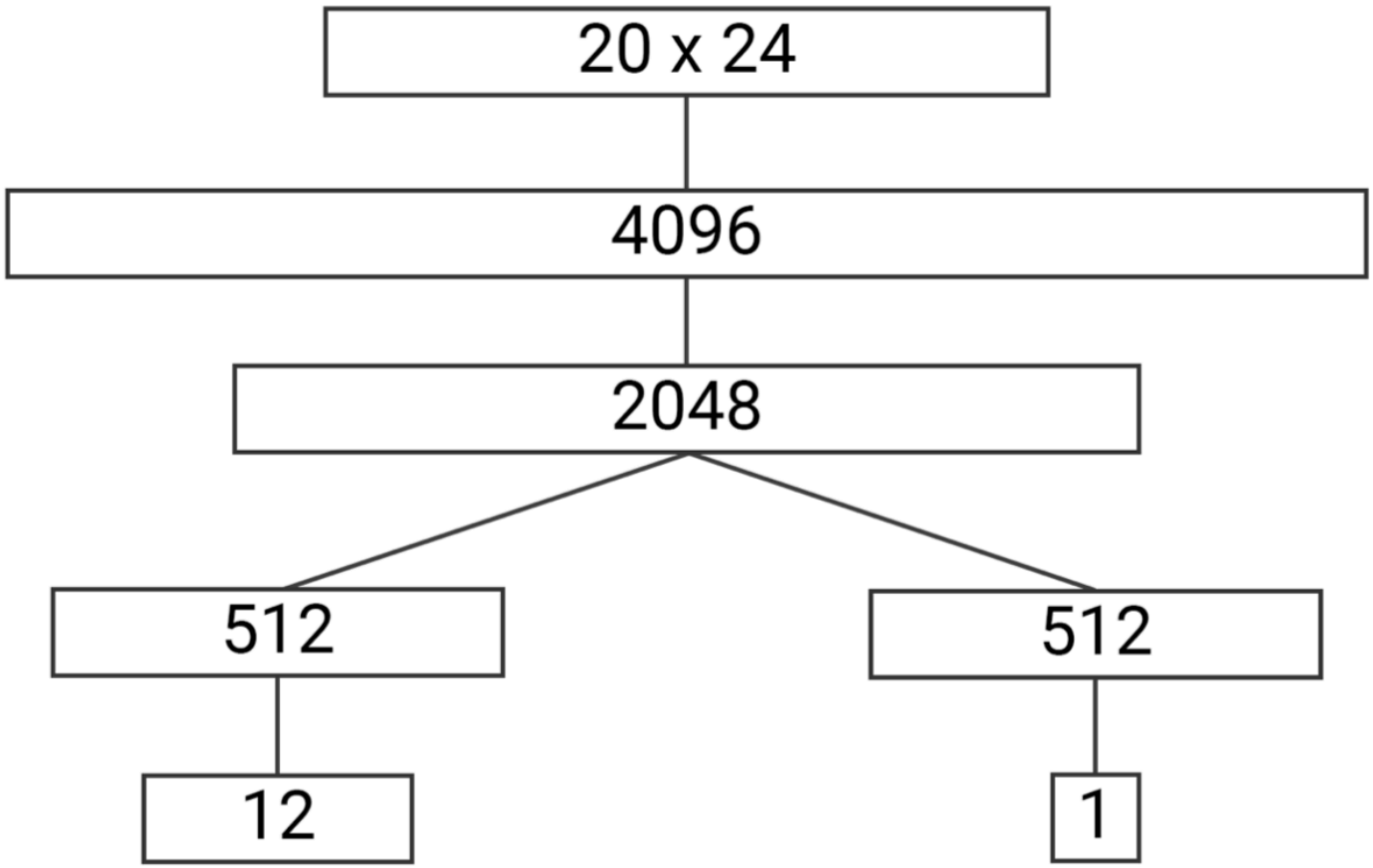


Figure: Architecture for f_θ. Each layer is fully connected. Using elu activation on all layers except for the outputs. A combined value and policy network results in more efficient training compared to separate networks.

Discussion

Discussion

“

Many combinatorial optimization problems can be thought of as sequential decision making problems, in which case we can use reinforcement learning.

Besides further work with the Rubiks Cube, we are working on extending this method to find approximate solutions to other combinatorial optimization problems such as prediction of protein tertiary structure.

”

Protein folding

“

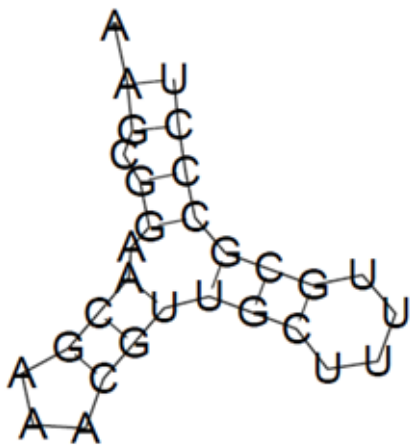
For example, in protein folding, we can think of sequentially placing each amino acid in a 3D lattice at each timestep. If we have a model of the environment, ADI can be used to train a value function which looks at a partially completed state and predicts the future reward when finished. This value function can then be combined with MCTS to find approximately optimal conformations.

”

Some thoughts on RNA structure predition

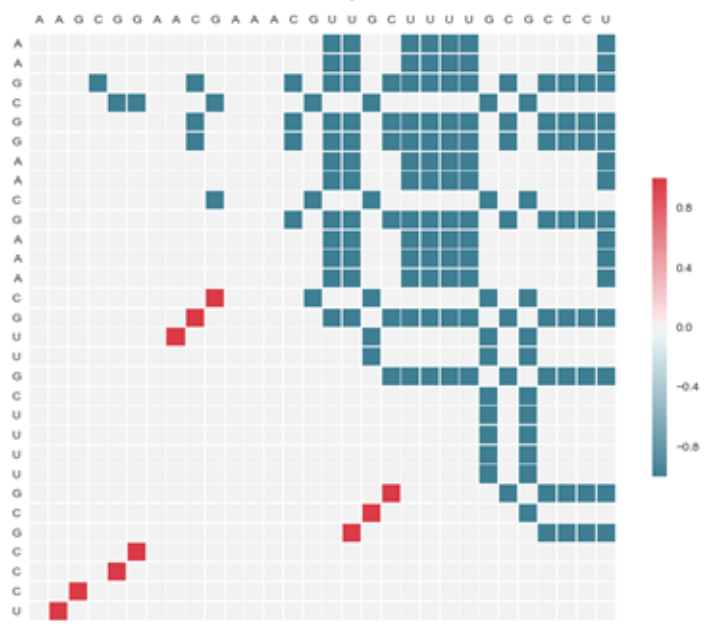
Initial State S_0 -A-A-G-C-G-G-A-A-C-G-A-A-A-C-G-U-U-G-C-U-U-U-U-G-C-G-C-C-C-U-

Target State S_n

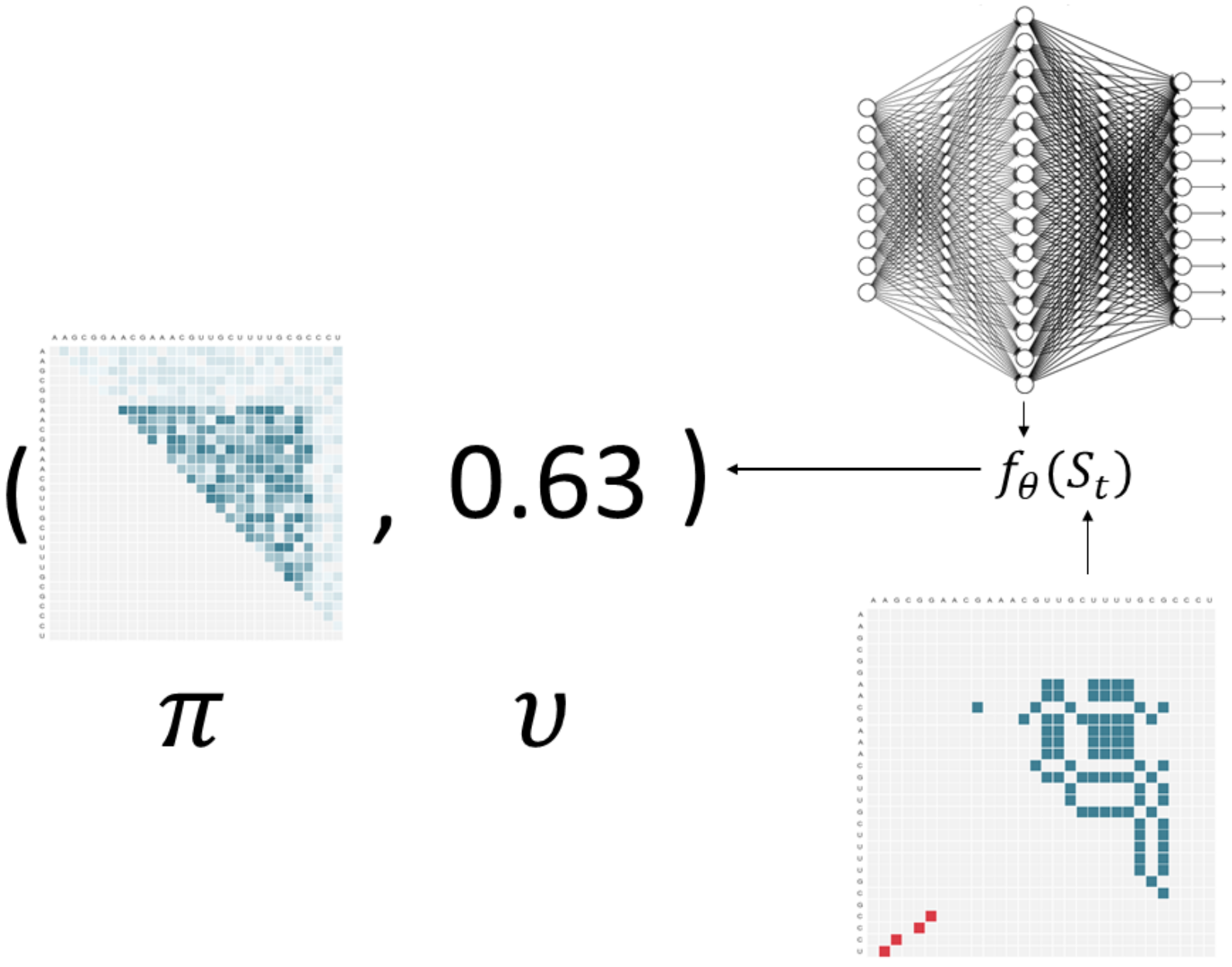


Action Space

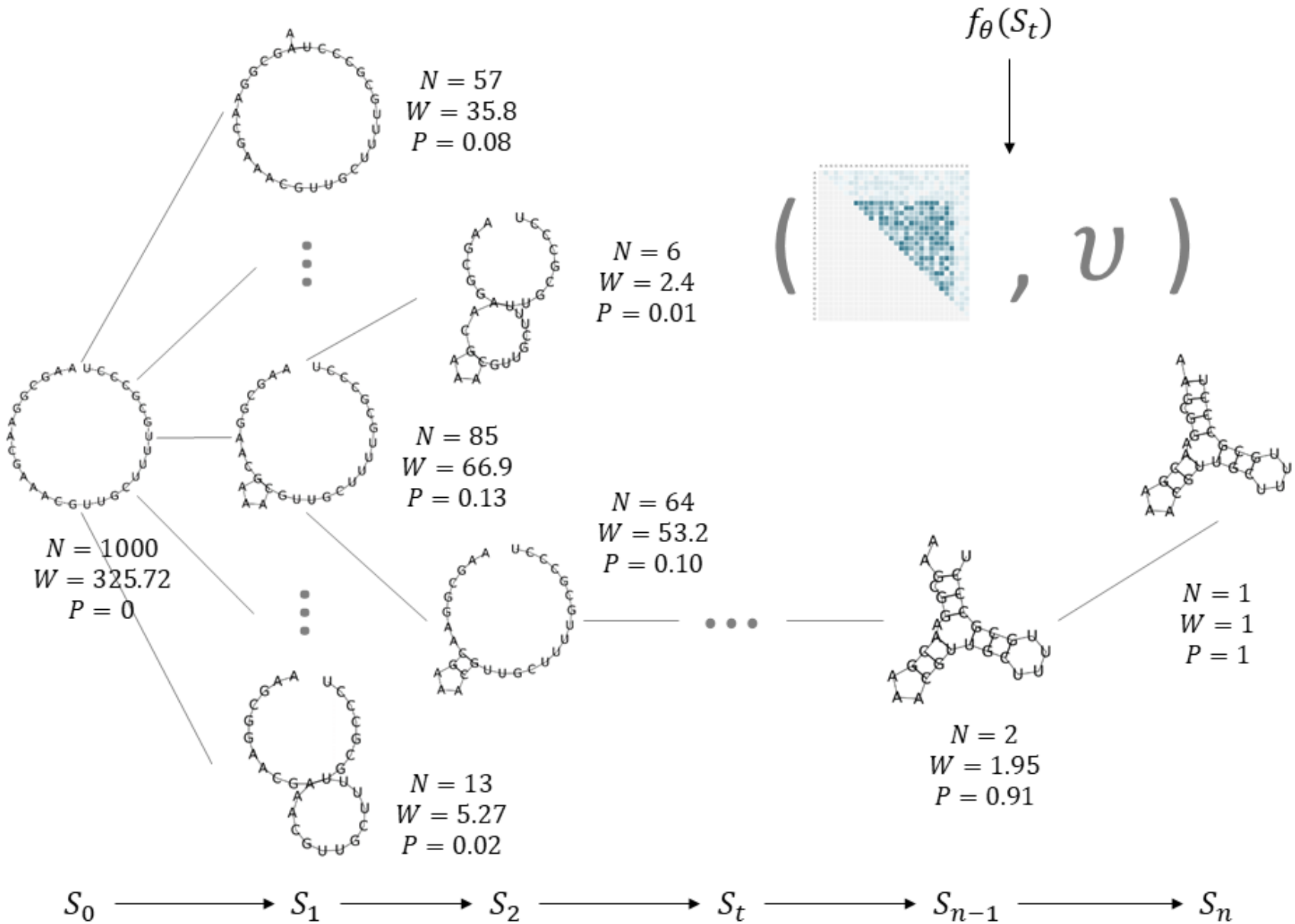
Native Pairs



State evaluation function



MCTS to find optimal conformation



Deep insight into AlphaGo Zero

Previous methods also using self-play and MCTS but not well, maybe the key point is CNN.

End

References

- AlphaGo Zero - How and Why it Works
  http://tim.hibal.org/blog/alpha-zero-how-and-why-it-works/
- Alpha Go Zero Cheat Sheet
  https://applied-data.science/static/main/res/alpha_go_zero_cheat_sheet.png
- Mastering the game of Go with deep neural networks and tree search
  https://deepmind.com/research/publications/mastering-game-go-deep-neural-networks-tree-search/
- Mastering the game of Go without Human Knowledge
  https://deepmind.com/research/publications/mastering-game-go-without-human-knowledge/
- Solving the Rubiks Cube Without Human Knowledge
  https://arxiv.org/pdf/1805.07470v1.pdf