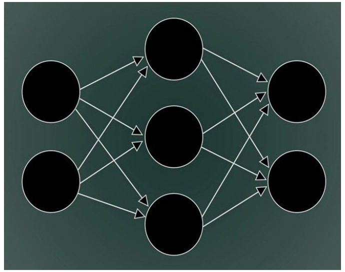
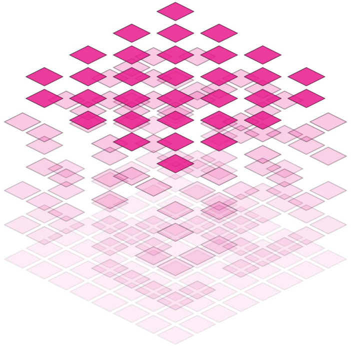
Deep Neural Network Chess Engine Architectures

Mateen Ulhaq

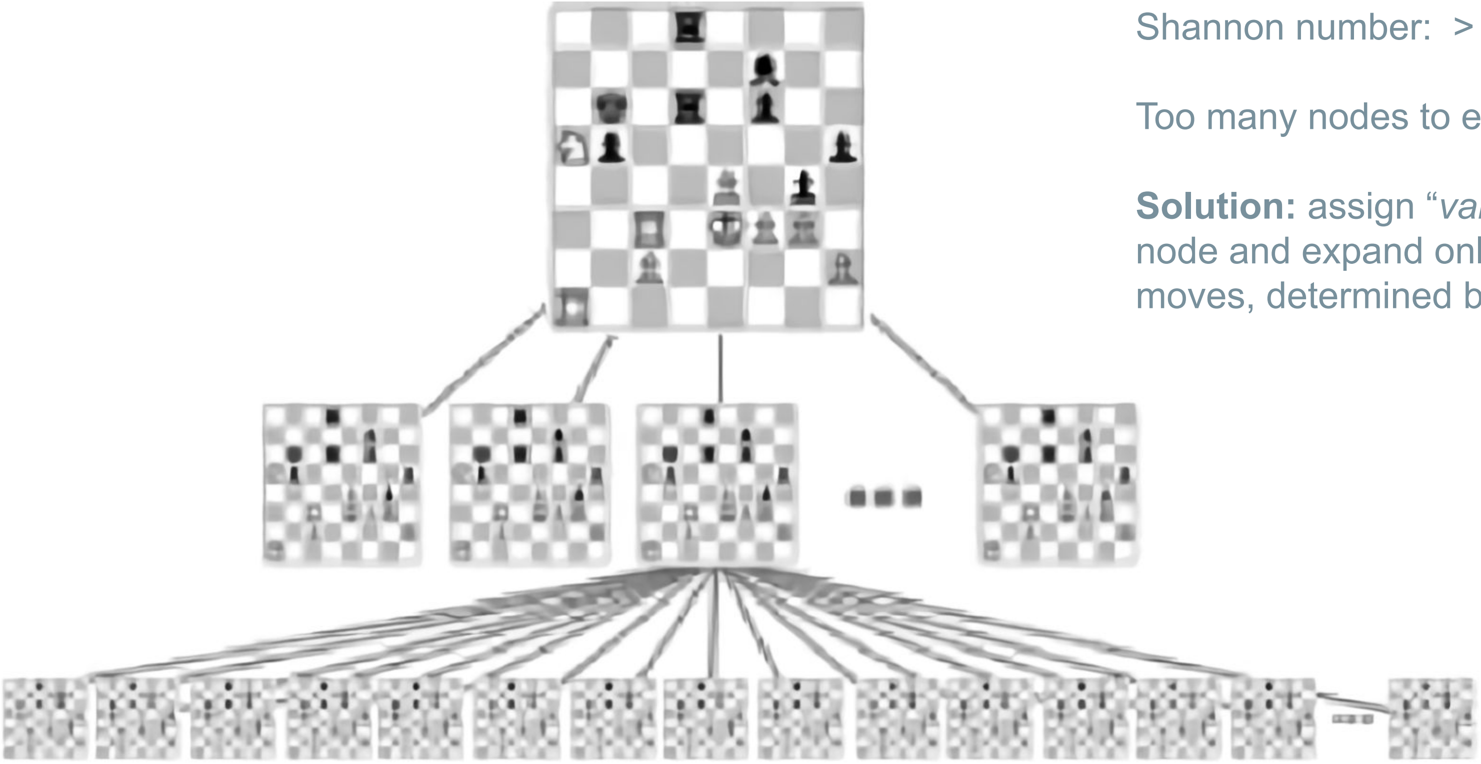




Overview

1. **Background:** Game trees, policy/value, search, board representation
2. **Architecture**
3. **Training**
4. **Further work**

Game tree



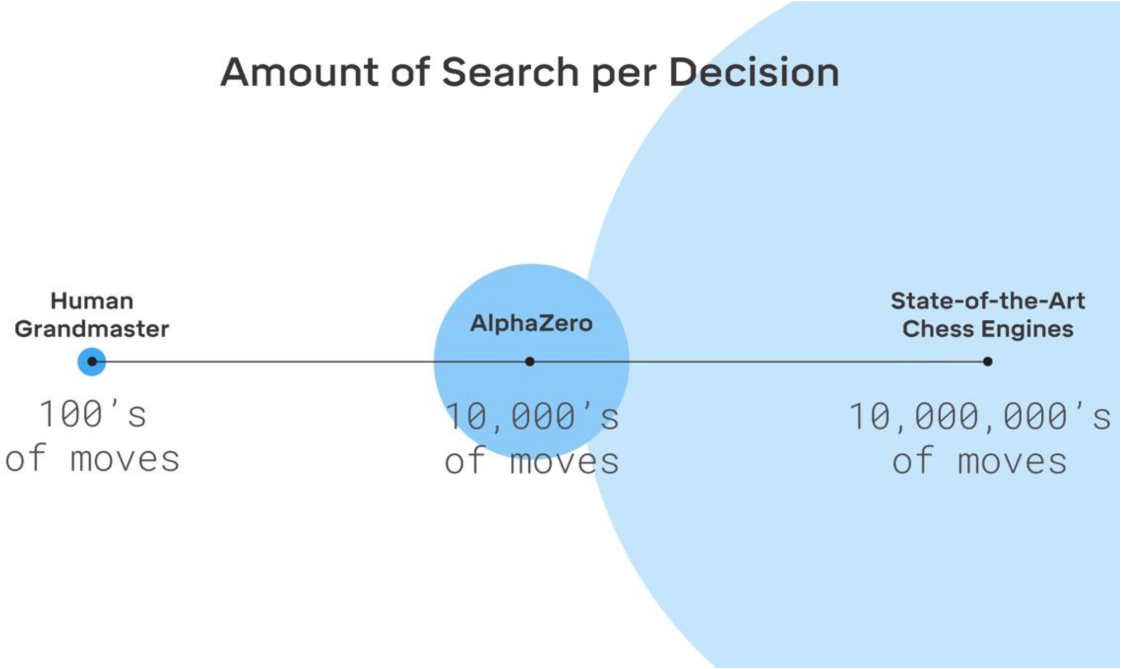
Each node has many valid moves.

Shannon number: > 10^{120} games!

Too many nodes to expand.

Solution: assign "*value*" to each node and expand only the most likely moves, determined by a "*policy*".

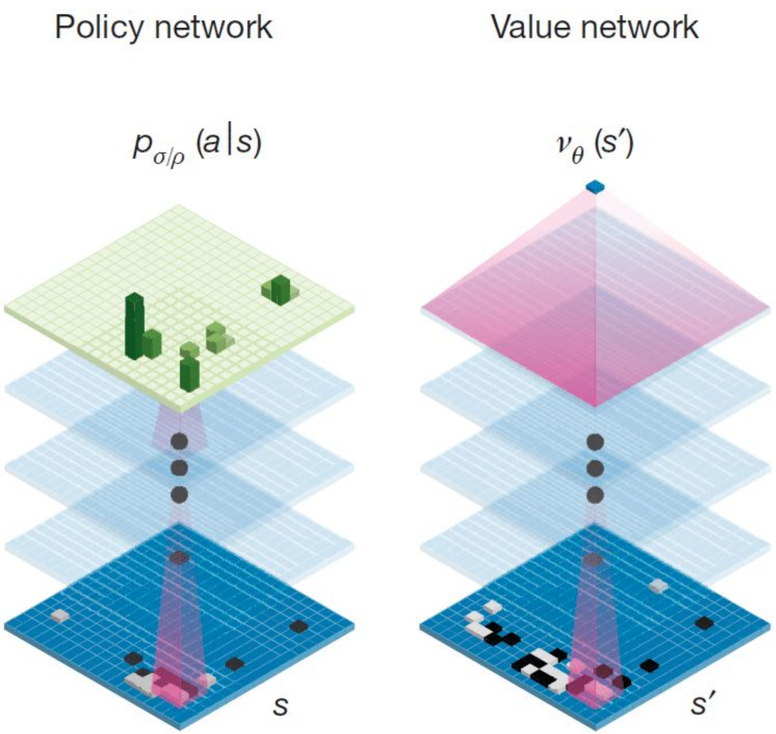
“Comparison” of tree search across chess playing entities



Deep reinforcement learning based chess engines search fewer nodes since calculating “value” and “policy” using a neural network is expensive.

However, the “value” function is often more accurate than classical chess engines.

Policy and value



Common input: game board

(Technically, the input is a 112x8x8 tensor, which we will see soon.)

Policy output: probability vector of most likely moves

Value output: a scalar value, where:

- 1 = win
- 0 = draw
- -1 = loss

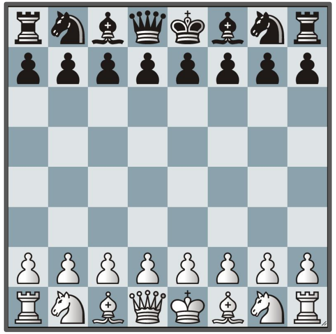
PUCT Search

$$\text{UCB}_i = \frac{w_i}{s_i} + c\sqrt{\frac{\ln s_p}{s_i}}$$

- w_i : this node's number of simulations that resulted in a win
- s_i : this node's total number of simulations
- s_p : parent node's total number of simulations
- c : exploration parameter

- [Polynomial] Upper Confidence Tree Search
- A “better” variant of MCTS (Monte Carlo Tree Search) for chess
- Given infinite time and memory, converges to “ideal” minimax
- **Value** (win/draw/loss) is predicted at each node
- **Policy** determines which nodes to expand next
- Chooses nodes with highest upper confidence bound (UCB)
    - A parent node’s confidence level falls as more descendants are explored
    - Optimistic bound (like admissible heuristics in optimal informed search)

Board representation



A board is represented by a **12x8x8** tensor.

BLACK PAWNS

```
0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

BLACK KNIGHTS

```
0 1 0 0 0 0 1 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

BLACK BISHOPS

```
0 0 1 0 0 1 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

BLACK ROOKS

```
1 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

BLACK QUEEN

```
0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

BLACK KING

```
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

WHITE PAWNS

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0
```

WHITE KNIGHTS

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 1 0 0 0 0 1 0
```

WHITE BISHOPS

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 1 0 0 1 0 0
```

WHITE ROOKS

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1
```

WHITE QUEEN

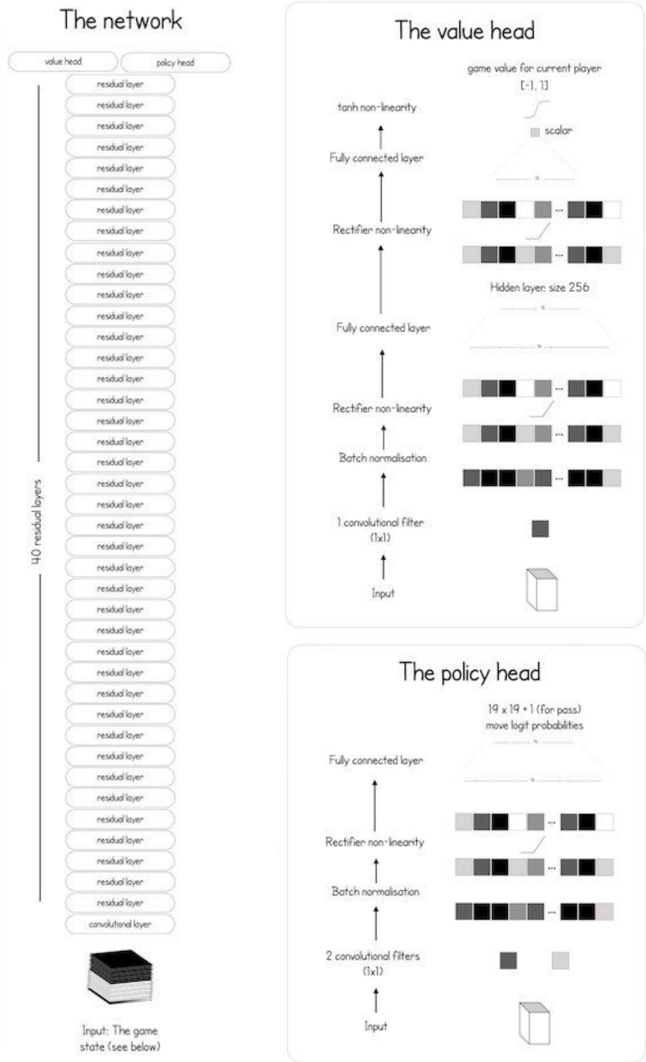
```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
```

WHITE KING

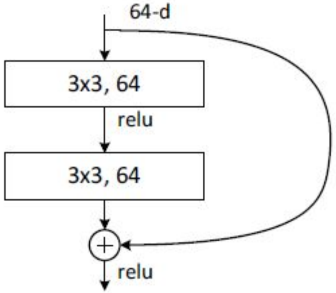
```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
```

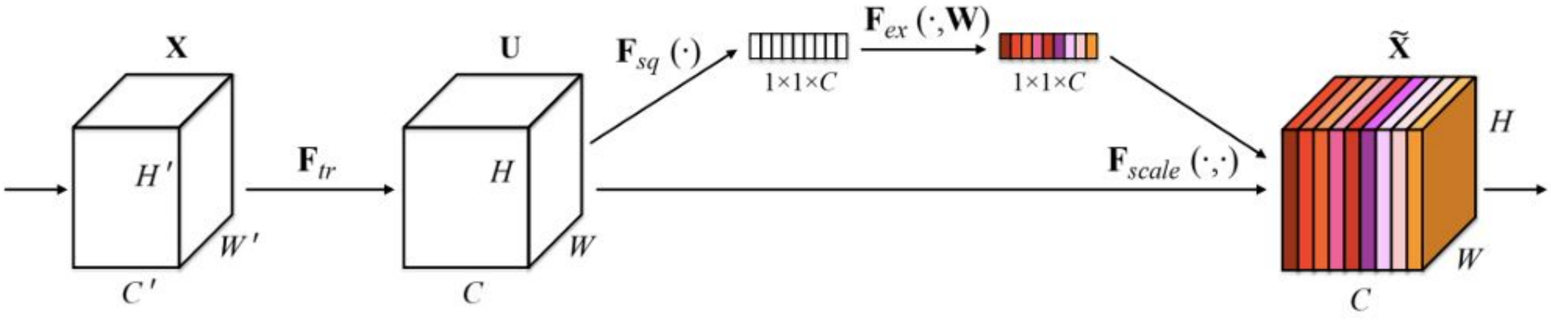
Architecture

- **Input** is game state and history prior (112x8x8)
- **Intermediate** “tower” of residual layers
  - conv layer (Cx8x8 → Cx8x8) with 3x3 padded filters
  - Leela Chess Zero: latest models use Squeeze-Excitation (SE) layers to inject trainable embeddings
- At the **output** are two sub-networks:
  - “**value**” (output_shape = 1) says if node is win/draw/loss
  - “**policy**” (output_shape = 80x8x8) tells us which nodes to expand in PUCT search



Residual blocks and Squeeze-Excitation blocks



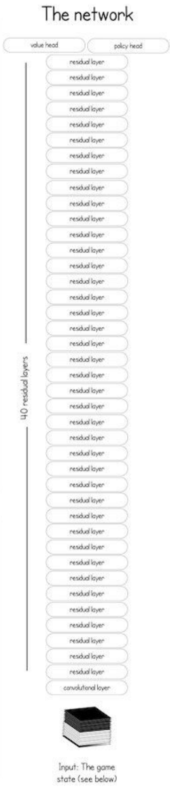


Architectural improvements

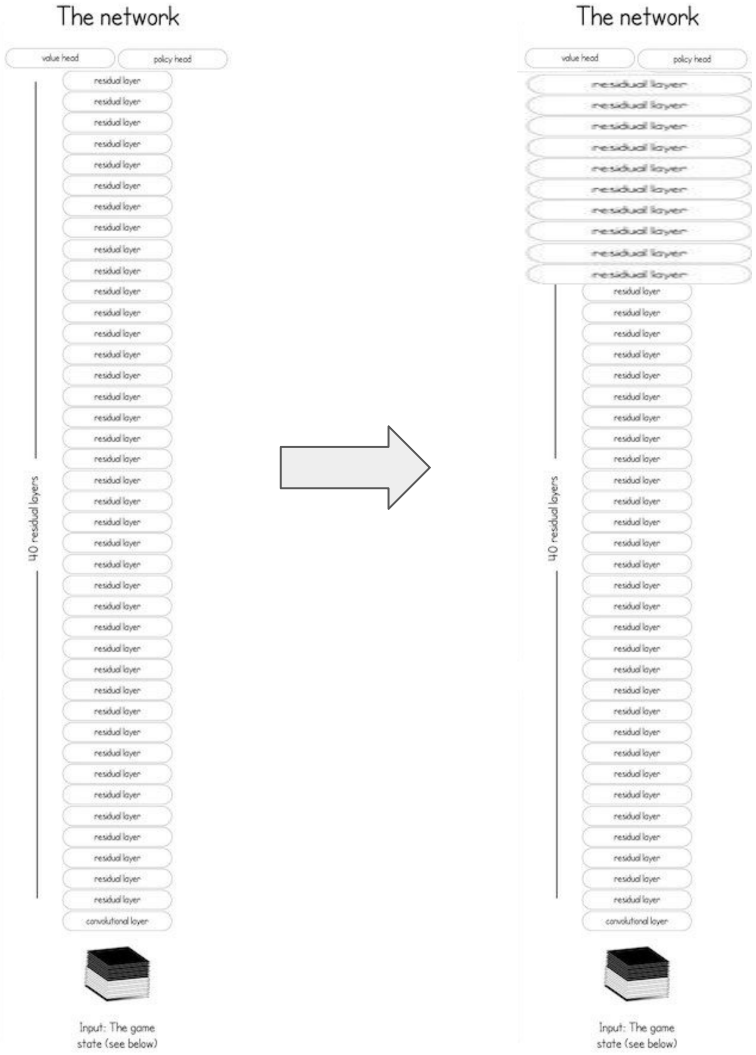
- The current architecture deviates from traditional image classifier CNNs:
  - The number of filters is a constant C for all convolutional layers in the residual tower!
  - No pooling layers so tensors are always Cx8x8.
  - Large ResNets use bottleneck blocks. (decompose 3x3 into 1x1 downsample + 3x3 + 1x1 upsample)
  - CNNs usually increase the number of filters to capture a larger variety of features.

- Architecture improvements could lead to:
  - quicker training
  - faster, more accurate inference (⇒ stronger engine)
  - improved quality of self-play data generation

- **Proposed alternative:** last quarter of residual tower uses 2x features.



Original architecture: residual tower is of constant “width” (number of output channels).



Proposed alternative: last quarter of residual tower uses 2x channels.

Training: Reinforcement learning → Supervised learning

- In order to *train* the network, one needs to perform **supervised** updates.
  Engine self-play is used to generate games, which are used as **training data**.

- **Input:** 112x8x8 tensor
  - `112 = (8 past game states) * (12 piece boards + 1 unused board) + (8 flat planes)`
  - flat planes are 8x8 boards which encode castling rights, rule-50, move counters, bias term

- **Value targets:** scalar with possible values {1, 0, -1} for {win, draw, loss}
  - Each game has a W/D/L result. The goal is to predict this!

- **Policy targets:** 80x8x8 tensor (historically, this used to be a 1858 move vector)
  - Try to predict the moves made in the game!

Training details

- **Dataset** taken from one of LC0's training runs for 100000s of self-play games
- **Loss:** typical multiclass cross entropy loss with softmax
    - `loss = loss_policy_network + λ · loss_value_network`
- **Optimizer:** AdamW (Adam + Weight decay)
    - Typically, SGD momentum is used to get better generalization,
      but AdamW was quicker to train on my hardware.
- Stochastic weight averaging (SWA) to improve generalization.
- BatchNorm layers are inserted within each residual layer.

Assessing performance

- No “accuracy” metric since the theoretical “best move” is usually unknown. This would not accurately reflect playing strength, either.

  (For example, an engine that plays 1 blunder for every 10 perfect moves is quite mediocre… chess is a brutal game!)

- Running game trials against other engines and architectures is useful, but only after the model is fully trained. Also requires writing custom high performance C++ and CUDA code.
- Hardware: engines may perform wildly differently across CPUs/GPUs.
- Various goals exist other than maximum playing strength:
  - better quality self-play data generation
  - quicker training

Further work

- Assess model performance by either writing slower engine in Python, or integrating the alternative network architecture(s) into LC0’s C++/CUDA code.
- Trying different model architectures.
- Other ideas: improving “shallow tactics” performance by fine tuning model on data containing a larger proportion of shallow tactics.

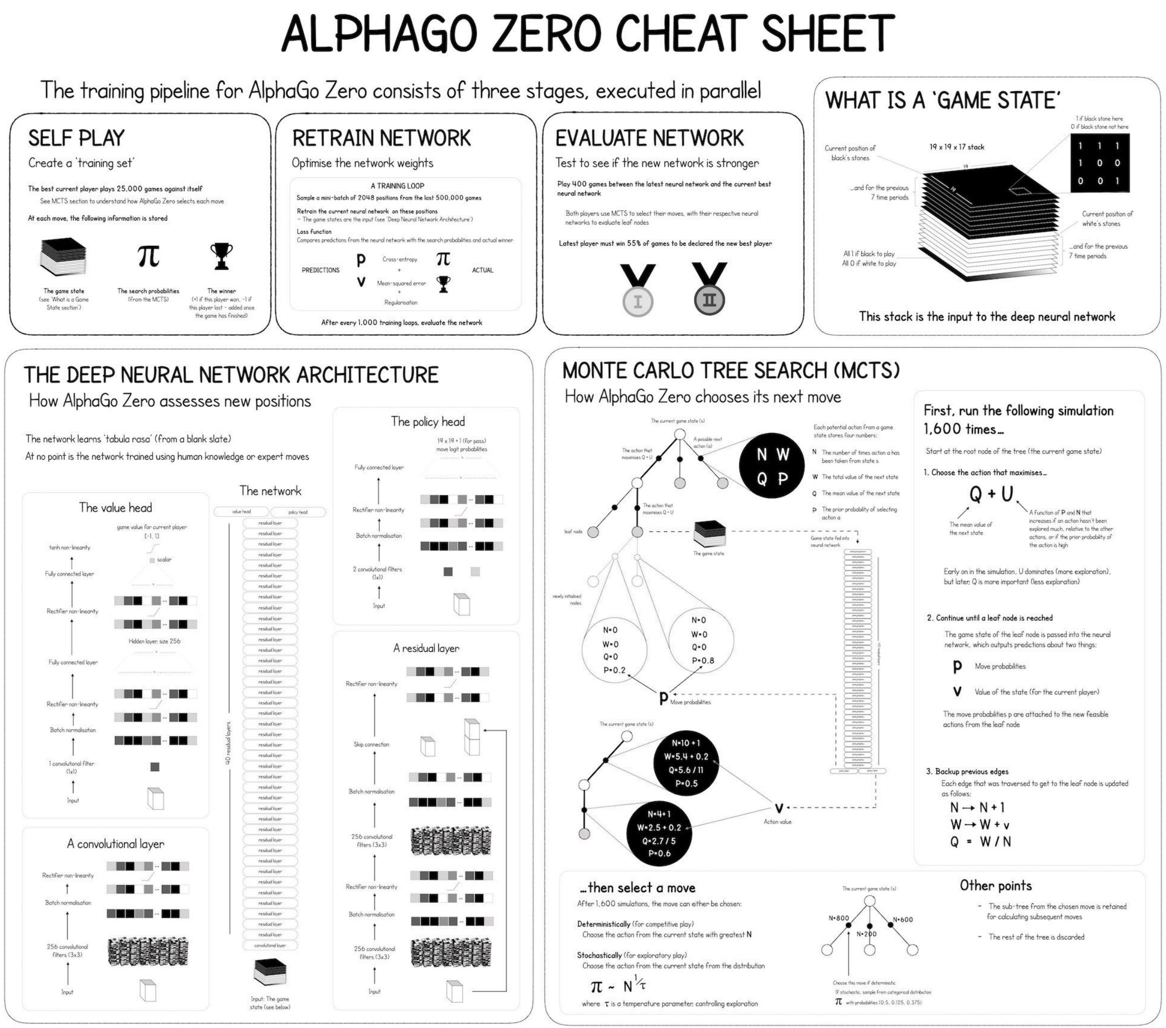
Thank you

Bellman equations for Markov Decision Process (MDP)

$$V^*(s) = \max_a Q^*(s,a)$$

$$Q^*(s,a) = \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V^*(s')\right]$$

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V^*(s')\right]$$


ALPHAGO ZERO CHEAT SHEET
The training pipeline for AlphaGo Zero consists of three stages, executed in parallel
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
Loss function
PREDICTIONS
ACTUAL
Cross-entropy
Mean-squared error
Regularisation
After every 1,000 training loops, evaluate the network
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
WHAT IS A 'GAME STATE'
19 x 19 x 17 stack
Current position of black's stones
...and for the previous 7 time periods
Current position of white's stones
...and for the previous 7 time periods
All 1 if black to play
All 0 if white to play
1 if black stone here
0 if black stone not here
This stack is the input to the deep neural network
THE DEEP NEURAL NETWORK ARCHITECTURE
How AlphaGo Zero assesses new positions
The network learns 'tabula rasa' (from a blank slate)
At no point is the network trained using human knowledge or expert moves
The value head
The network
The policy head
A convolutional layer
A residual layer
40 residual layers
Input: The game state (see below)
MONTE CARLO TREE SEARCH (MCTS)
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
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 → N + 1
W → W + v
Q = W / N
N W
Q P
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
...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
π ~ N^(1/τ)
where τ is a temperature parameter, controlling exploration
Other points
- The sub-tree from the chosen move is retained for calculating subsequent moves
- The rest of the tree is discarded

PUCT algorithm

Input: a root node r, a transition function, an action sampler, a time budget, a depth d_{max}, parameters α and e for each layer

Output: an action a

while time budget not exhausted **do**

  while current node is not final **do**

    if current node is a decision node z **then**

      if $\lfloor n(z)^\alpha \rfloor > \lfloor (n(z)-1)^\alpha \rfloor$ **then**

        we call the action sampler and add a child $w = (z, a)$ to z

      else

        we choose as an action among the already visited children (z, a) of z, the one that maximizes its score, defined by:

$$\hat{V}(z,a) + \sqrt{\frac{n(z)^{e(d)}}{n(z,a)}}. \tag{4}$$

      end if

    else

      if $\lfloor n(w)^\alpha \rfloor = \lfloor (n(w)-1)^\alpha \rfloor$ **then**

        we select the child of z that was least visited during the simulation

      else

        we construct a new child (i.e. we call the transition function with argument w)

      end if

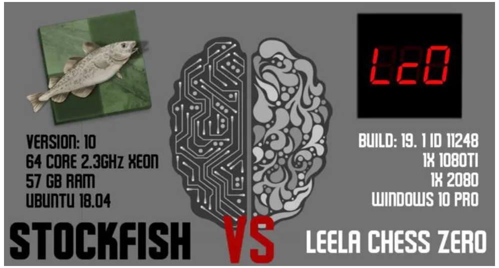
    end if

  end while

  we reached a final node z with reward $r(z)$; we back propagate all the information in the constructed nodes, and we go back to the root node r.

end while

Return the most simulated child of r.


LcO
VERSION: 10
64 CORE 2.3GHz XEON
57 GB RAM
UBUNTU 18.04
BUILD: 19. 1 ID 11248
1X 1080TI
1X 2080
WINDOWS 10 PRO
STOCKFISH
VS
LEELA CHESS ZERO


LcO

Other improvements

- Mixing known shallow tactics in training data to improve policy network on “shallow tactics”
- Model distillation

Benefits

- Smaller, faster model
- Save energy, run on smaller devices
- Can get better tradeoff for number of nodes expanded vs accuracy of e.g. value function

Why not just train on smaller model?

- Smaller may have lower accuracy or is harder to train

Experiments/Results

Training loss curves? These don't really show much info though… other than "network appears to learn somewhat".

# of epochs = 30.