DraughtsZero: Learning Draughts through Self-Play

Ben Hull https://bluehood.github.io/

Abstract

We present *DraughtsZero*, an AlphaZero-style reinforcement learning system for English Draughts that learns through self-play. The method combines three components: a game engine interface for legal move generation and terminal-state handling, a compact dual-head residual neural network that predicts both policy logits over a fixed 64×64 action space and a scalar value in $[-1, 1]$, and a PUCT-based Monte Carlo Tree Search that improves move selection during data generation. Positions are encoded as 8×8 feature planes with configurable channels for piece types, side to move, and optional history. The resultant model validate the feasibility of a lightweight AlphaZero-style draughts model trained on consumer. The project provides a modular baseline for future work on stronger training schedules and larger self-play volumes.

Keywords: Reinforcement Learning, Self-Play, Machine Learning

1 Introduction

Draughts (checkers) provides a compact but interesting benchmark for studying search-guided reinforcement learning. Although the game has a smaller state space than Chess or Go, strong play still requires planning, tactical precision in forced-capture sequences, and robust value estimation under sparse terminal rewards. These properties make draughts a useful setting for evaluating learning systems that combine neural function approximation with Monte Carlo Tree Search (MCTS).

AlphaZero, introduced by Google, is a self-play reineforocement learning system designed to play Chess and Go at the superhuman level. The system learned to play Chess, Go and Shogi (Japenese Chess), starting from random play, over 24 hours of training. This work implements a similar approach with AlphaZero style for English Draughts. The system integrates three core components: (i) a game engine wrapper for legal move generation and terminal detection, (ii) a dual-head residual neural network that predicts both policy logits over a fixed 4096-action space and a scalar value in $[-1, 1]$, and (iii) a MCTS procedure that uses network priors and value estimates to guide exploration. Position encoding is designed for learning efficiency, with configurable feature planes for piece types, optional side-to-move information, and optional temporal history.

2 Implementation

We instantiate a standard policy–value learning framework in which a residual network parameterises both move priors and state values, and Monte Carlo Tree Search (MCTS) uses these predictions to improve action selection during self-play. The implementation is intentionally minimal and modular, with all components exposed through a unified training

configuration. This allows for training a home hardware, rather than requiring significant amounts of compute to train.

2.1 Neural Architecture

Let s_t denote a draughts position at ply t. We encode s_t as a tensor

$$x_t \in \mathbb{R}^{C\times 8\times 8},$$

where C is determined by feature design. The base representation uses four piece planes (white men, white kings, black men, black kings), with optional extensions for side-to-move and temporal history.

The model f_θ is a dual-head residual network [1]:

$$(\ell_t, v_t) = f_\theta(x_t),$$

where $\ell_t \in \mathbb{R}^{4096}$ are unnormalised policy logits and $v_t \in [-1, 1]$ is a scalar value estimate from the current-player perspective. The policy dimensionality follows a fixed mapping

$$a = \_sq \times 64 + to\_sq,$$

yielding $64 \times 64 = 4096$ action indices.

The network begins with a 3×3 convolution and batch normalisation, followed by B residual blocks at width d channels. Two lightweight heads share this trunk: (i) a policy head with 1×1 projection and linear readout to 4096 logits; (ii) a value head with 1×1 projection, an MLP bottleneck, and final tanh activation. In practice, this architecture offers a favourable balance between representational capacity and search-time efficiency.

2.2 Residual Blocks

Each residual block applies two 3×3 convolutions (with batch normalisation and ReLU), adding the block input to the transformed features:

$$h^{(l+1)} = \mathrm{ReLU}\Big(h^{(l)} + \mathcal{F}\Big(h^{(l)}; \theta^{(l)}\Big)\Big).$$

where,

$h^{(l)}$ **(Input/Hidden State):** This represents the activations or features from the current layer l. This is the "information" the network has at this specific stage.

$h^{(l+1)}$ **(Output):** This is the result of the residual block, which becomes the input for the next layer $(l + 1)$.

$\mathcal{F}(h^{(l)}; \theta^{(l)})$ **(Residual Mapping):** This is the "learned" part of the block.

- $\mathcal{F}$ represents the series of operations performed by the network to produce the output logits.

1. A residual network (ResNet) is a type of deep neural network that uses 'skip connections' to allow gradients to flow through layers more easily. This helps with training large deep neural networks

- $\theta^{(l)}$ represents the parameters (weights and biases) of those convolutions that the network learns during training.

(The Shortcut/Skip Connection +): This is the core of the Residual Block. Instead of just passing the input through the convolutions, we add the original input $h^{(l)}$ back to the transformed features. This allows the network to learn an "identity" mapping easily—if $\mathcal{F}$ becomes zero, the information simply passes through unchanged.

ReLU(...) **(Activation Function):** Standing for Rectified Linear Unit, this is the final non-linear "filter" applied after the addition. It ensures that any negative values resulting from the sum are set to zero, helping the network learn complex patterns.

This preserves gradient flow across depth and stabilises optimisation under noisy self-play targets. We use a constant channel width across blocks, allowing straightforward scaling by block count B and width d.

2.3 Search-Guided Self-Play

At each non-terminal position, we run PUCT-style MCTS. For an edge (s, a), the selection score is

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

where,

$U(s,a)$ **(Total Selection Score):** The combined score used to decide which action to take during the simulation. MCTS greedily selects the action $a = argmax_a U(s,a)$ to balance exploitation and exploration.

$Q(s,a)$ **(Exploitation Term):** The mean action value. It represents the average value (expected reward) of all states visited in the subtree below edge (s,a). High Q-values encourage the search to revisit moves that have performed well in previous simulations.

c_{puct} **(Exploration Constant):** A hyperparameter that controls the influence of the prior probability and visit counts. It determines the weight given to exploration relative to the established value Q.

$P(s,a)$ **(Prior Probability):** The initial 'hunch' provided by the policy head of the neural network. This biases the search toward moves the network thinks are promising before they have been extensively explored.

$\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$ **(Exploration Bonus):** This fraction provides the exploration drive:

- $\sum_b N(s,b)$ is the total number of visits to the parent state s. As the search progresses, the numerator grows, increasing the urge to explore.
- $1 + N(s,a)$ is the number of times this specific action has been visited. As we visit a move more often, this denominator grows, shrinking the exploration bonus and forcing the search to eventually rely more on the Q-value.

Dirichlet Noise: During root expansion, noise $\eta \sim Dir(\alpha)$ is added to the priors $P(s, a)$. This ensures the search explores a wider variety of moves than the network might initially suggest, preventing the agent from becoming "stuck" on a single line of play.

Leaf expansion evaluates the network once, masks illegal moves, and normalises over legal actions. At the root, Dirichlet perturbation is optionally applied,

$$\tilde{P} = (1 - \varepsilon)P + \varepsilon\eta, \qquad \eta \sim \mathrm{Dir}(\alpha),$$

to encourage exploration in early game trajectories.

After K simulations, the improved search policy is formed from normalised root visit counts:

$$\pi_t(a) = \frac{N(s_t, a)}{\sum_b N(s_t, b)}.$$

Action selection is temperature-controlled for early plies and becomes greedy later in the game. Self-play stores triples (x_t, π_t, z_t), where $z_t \in \{-1, 0, 1\}$ is the terminal outcome mapped to the perspective at time t.

2.4 Training Loop

Training proceeds in iterations, each consisting of (1) self-play data generation, (2) replay-buffer updates, and (3) minibatch optimisation. The objective combines value regression and policy imitation:

$$\mathcal{L}(\theta) = (z - v)^2 - \pi^\top \log p + \lambda\theta_2^2,$$

where $p = \mathrm{softmax}(\ell)$, v is the predicted value, and λ is weight decay. We optimise with Adam and optional gradient clipping.

To control policy drift, we maintain a *candidate* and *reference* (champion) model. At evaluation intervals, the candidate is pitted against the reference in alternating colors. Promotion occurs only if the candidate exceeds a target win-rate threshold; the promoted champion is checkpointed for downstream inference (including interactive play).

2.5 Our Implementation

The training loop instantiates a `Coach` with a `TrainConfig`. Concretely, the model is

$$f_\theta : \mathbb{R}^{C\times 8\times 8} \to \left(\mathbb{R}^{4096},\, [-1, 1]\right),$$

where C is the configured input-channel count and 4096 corresponds to the fixed action space (64×64). In the default CLI path, `run.py` sets:

$$B = 4, \qquad d = 64, \qquad h_v = 32,$$

where B is the number of residual blocks, d is trunk channel width, and h_v is the hidden size of the value head MLP.

The *actual* model trained by default in the loop is a compact dual-head residual network with:

- input tensor shape $4 \times 8 \times 8$,

- residual trunk depth $B = 4$ and width $d = 64$,
- policy head producing 4096 logits,
- value head producing a scalar in $[-1, 1]$ via tanh.

During each iteration, self-play data are generated using this candidate model, and gradient updates are applied to the same parameterization f_θ; evaluation/promotion compares this candidate against a reference copy maintained by the trainer.

3 Results

Due to hardware constraints, we could not train the model for a significant number of games. We trained our model on approximately 100 games over a period of 8 hours on an M2 Apple chip.

The resultant model played well in initial human to agent games, particularly early on in the game. The model tended to play poorly towards the end of the game in positions that were not seen during training. In particular, the model had difficulty converting winning positions at the end of the game.

Ideally, we would train the model on dedictated hardware with a GPU or TPU, however, such resources were not avaliable to the author at the time of writing.

4 Conclusion

This paper presented *DraughtsZero*, an AlphaZero-style system for English Draughts that combines a dual-head residual network with PUCT-based MCTS and iterative self-play training. The implementation demonstrates that the core policy–value + search paradigm can be reproduced in a compact, modular form and trained on consumer hardware, while still producing a model capable of competitive early- and mid-game play against a human opponent.

The main limitation of this study is training scale: with only ~ 100 self-play games generated over 8 hours, the learned policy and value function remain undertrained, especially in the endgame. This is reflected in unstable conversion of winning positions and weaker play in board states that are rarely seen.

Future work could focus on increasing self-play volume and search budget, improving evaluation and promotion protocols, and incorporating stronger endgame supervision (e.g., tablebase-style targets or targeted curriculum self-play). Additional experiments on larger hardware and different configurations of network depth, channel width, and input features would help quantify which design choices most improve sample efficiency and playing strength. Overall, these results suggest that Draughts remains a useful, accessible benchmark for studying search-guided reinforcement learning under compute constraints.