KAN-Chess: Direct Move prediction using Kolmogorov-Arnold Networks

Abstract

This research proposes the development of a chess engine inspired by AlphaZero's reinforcement learning architecture but utilizing Kolmogorov-Arnold Networks (KANs) instead of traditional Multi-Layer Perceptron (MLPs). The primary objective is to achieve competitive chess-playing performance while significantly reducing computational requirements for training. KANs, which learn activation functions on edges rather than fixed activation functions at nodes, offer potential advantages in parameter efficiency and interpretability. This project aims to demonstrate that KANs can capture the complex positional and tactical patterns in chess with fewer parameters and training iterations than conventional deep learning architectures.

1. Background and Motivation

1.1 AlphaZero and Modern Chess Engines

AlphaZero, developed by DeepMind in 2017, revolutionized game-playing AI by combining deep neural networks with Monte Carlo Tree Search (MCTS) and reinforcement learning through self-play. Unlike its predecessors that relied on hand-crafted evaluation functions and opening books, AlphaZero learned chess entirely from scratch, achieving superhuman performance after just 4 hours of training on specialized hardware (5,000 TPUs).

However, the computational requirements of AlphaZero remain prohibitive for most researchers:

- Training required approximately 44 million self-play games
- Utilized 5,000 TPUs for training
- The neural network architecture employed deep residual networks with millions of parameters
- Estimated training cost exceeded $35 million in compute resources

1.2 Kolmogorov-Arnold Networks

Kolmogorov-Arnold Networks (Liu et al., 2024) represent a paradigm shift in neural network architecture, inspired by the Kolmogorov-Arnold representation theorem. Unlike traditional

MLPs that apply fixed activation functions to weighted sums at nodes, KANs place learnable activation functions on the edges (weights) themselves. Key theoretical advantages of KANs include:

- **Parameter Efficiency**: KANs can achieve similar or better accuracy with 10-100× fewer parameters than MLPs
- **Interpretability**: Learnable univariate functions on edges can be visualized and understood
- **Adaptive Expressiveness**: Networks can learn problem-specific activation functions rather than relying on generic ones (ReLU, tanh)
- **Reduced Depth Requirements**: KANs often require fewer layers to approximate complex functions

Recent empirical results show KANs outperforming MLPs on various mathematical and scientific tasks, but their application to complex strategic domains like chess remains unexplored.

1.3 Research Gap

While KANs have demonstrated promise in function approximation and scientific computing tasks, no existing work has applied them to strategic game playing or compared their efficiency against AlphaZero-style architectures. This research addresses three critical questions:

1. Can KANs effectively learn chess position evaluation and move policy with reduced parameters?
2. What computational savings can be achieved without sacrificing playing strength?

2. Significance

1. **First KAN-based game-playing agent**: No prior work has applied KANs to adversarial strategic games
2. **Efficiency-focused architecture design**: Explicitly targeting computational reduction rather than maximum performance
3. **Hybrid approach**: Combining KAN innovations with proven MCTS reinforcement learning frameworks

3. Proposed Solution

3.1 Architecture Overview

The proposed KAN-Chess engine consists of three integrated components:

KAN-Based Policy and Value Network:

- Input: 8×8×119 tensor representing board state (piece positions, castling rights, en passant, move history)
- Architecture: KAN layers replacing traditional convolutional and fully connected layers

Monte Carlo Tree Search (MCTS)

Self-Play Training Pipeline

3.2 KAN Network Design

Input Processing:

- Board representation: 8×8 grid with 119 channels (piece types, colors, auxiliary features)
- Initial KAN layer: 8×8×119 → 8×8×256, utilizing spatial locality

Residual KAN Blocks (6-12 blocks):

- Each block contains:
    - KAN layer with learnable B-spline activation functions (degree 3)
    - Skip connections for gradient flow
    - Group normalization (replacing batch normalization for stability)
- Grid size: 5-10 intervals per spline (tunable hyperparameter)

3.3 Training Methodology

Phase 1: Random Play Initialization (Days 1-3)

- Generate 10,000 games from random move selection
- Provides diverse initial training data
- Prevents early overfitting to specific openings

4. Dataset

4.1 Self-Play Generated Data

Primary Dataset: Self-play games generated during training

- **Source**: Internal game generation through MCTS self-play
- **Size**: Target 200,000-500,000 games (20-50 million positions)
- **Features per position**:
  - Board state: 8×8×119 tensor
  - MCTS policy: 4,672-dimensional probability vector
  - Game outcome: {-1, 0, 1} (loss, draw, win)
  - Metadata: move number, castling rights, repetition count

5. Evaluation Methods

Value Head Accuracy:

- Mean Squared Error (MSE) between predicted values and game outcomes
- Correlation coefficient on held-out test positions
- Comparison with Stockfish evaluations (centipawn error)

Computational Cost:

- Total training time (GPU-hours)
- Number of self-play games required to reach milestones (1500 Elo, 1800 Elo, 2000 Elo)
- FLOPS per forward pass
- Memory footprint (training and inference)

6. Planned Tools and Technologies

6.1 Core Frameworks

Python 3.9+: Primary programming language

- **PyTorch 2.0+**: Deep learning framework for KAN implementation and training
- **python-chess**: Chess move generation, validation, and board representation
- **NumPy**: Numerical operations and array manipulation
- **SciPy**: Statistical analysis and scientific computing

pykan: Official Kolmogorov-Arnold Network library

Stockfish Python API: Baseline evaluation and testing **chess-mcts**: Custom Monte Carlo Tree Search implementation

Numpy & Pandas: Data analysis and results aggregation

Matplotlib: Training curves, activation function plots, and statistical charts **Seaborn**: Advanced statistical visualizations

TensorBoard: Real-time training monitoring, histograms, and embeddings.

Plotly: Interactive 3D visualizations of activation functions.

chess.svg: Board position rendering for debugging and analysis