ChessAIThon: Integrating Chess, Coding & AI

A comprehensive educational initiative combining chess-based learning with coding principles and AI concepts to develop critical thinking, problem-solving, and creativity in VET curricula.

Project Objectives

Curriculum Integration

Provide tools to integrate chess-based learning for developing critical thinking, problem-solving, and creativity, including teaching coding principles, AI concepts, and version control through chess scenarios.

Interactive Learning Platform

Develop an online platform with a chessboard interface using Chess.js library for displaying legal moves, validating user moves, and storing valid moves in a database.

Data Management & AI Training

Implement robust data handling capabilities and provide methodologies for students to train AI models, culminating in a chess and AI competition.

Implementation Plan

1

Phase 1: Methodology Development (Months 1-6)

Develop curriculum for coding through chess, transversal skills, data structures, machine learning concepts, and version control. Create lesson plans and prepare teacher training.

2

Phase 2: Platform Development (Months 4-12)

Create database, develop chessboard interface, implement move validation, and integrate AI training modules and version control.

3

Phase 3: Student Engagement (Months 9-18)

Guide students in using the platform, training AI tools, and preparing for competition.

4

Phase 4: Competition & Dissemination (Months 15-24)

Host transnational chess and AI competition, document results, and share resources.

Curriculum Development (Phase 1)

Chapter 1: Coding Fundamentals through Chess

Teaching logic, functions, and implementation using chess problem-solving strategies

Chapter 2: Transversal Skills Development

Fostering cognitive skills, creativity, lateral thinking, problem-solving, and planning

Chapter 3: Chess Data Structures

Explaining representation in PGN, JSON, CSV, FEN, UCI, and SAN formats

Chapter 4: Machine Learning for Chess

Covering theoretical concepts and procedures relevant to chess-based AI

Chapter 5: Version Control

Outlining use of Git and GitHub for storing and sharing chess datasets

Lesson Plan Creation

Developing step-by-step guidance for chess problem-based scenarios

Platform Development (Phase 2)

Database Design

Store chess scenarios (FEN format) and moves (SAN, UCI, or resulting FEN)

Frontend Development

Create intuitive chessboard interface using Chess.js library

Move Validation

Integrate logic to check move legality and store valid moves

AI Integration

Prepare for AI training modules and version control

AI Architecture

The project's AI approach is inspired by AlphaZero but adapted for educational contexts with limited hardware resources.

ChessMarro vs AlphaZero: Key Differences

- AlphaZero learns by playing against itself repeatedly
- ChessMarro trains using human (student) games as examples
- Both use Convolutional Neural Networks (CNNs) for move prediction
- ChessMarro optimized for educational hardware constraints

While traditional algorithms like Stockfish are powerful through brute-force calculation, AI offers different advantages in learning, adaptation, and discovery in complex environments.

AI Implementation: Datasets

The team sourced extensive datasets from Lichess via Kaggle, including 30,000 pre-mate games and many complete games that required transformation into the project's format.

Data Sources

Kaggle datasets from Lichess: chess-games-in-fen-and-best-move

Storage Format

JSON (compressing each of 77 board positions into int64) or Parquet (better native compression)

Conversion Process

Transformation algorithms: convert-to-chessintion-format

AI Implementation: Training & Deployment

Training

- CNN model trained on Kaggle using Parquet files with generic games
- Mate-specific version achieved 90% precision
- Fine-tuned CNN balances precision and performance

Training code: cnn-pytorch-chess-generic

Deployment

- Model deployed on Kaggle with MCTS
- Interactive demo available
- Model shared via Hugging Face
- CPU deployment for accessibility

Demo: play-with-chessmarro

Development Workflow

Model Improvement

Use Kaggle or Colab to improve the CNN model

Version Control

Upload new versions to GitHub and Hugging Face

Deployment

Deploy on local GPU for competition and web app via CI/CD

Testing

Test model via Colab, Kaggle, and Hugging Face deployment

The centralized infrastructure includes GitHub for code and documentation, Kaggle for data transformation and training, and Hugging Face for model sharing and deployment.

Next Steps

Centralize Resources

Consolidate AI model, libraries, documentation, and source code in official GitHub repository

Finalize Training Pipeline

Complete dataset transformation tools and CNN training workflows

Prepare for Competition

Set up local GPU deployment for the transnational chess and AI competition

Developing a Chess AI with Convolutional Neural Networks

A deep learning approach to predicting optimal chess moves using modern AI techniques

Why Chess is Perfect for AI Learning

Well-defined Rules

Chess has clear rules and a finite state space, making it easier to model computationally

Historical AI Benchmark

From Deep Blue to AlphaZero, chess has rich literature and frameworks for AI development

Scalable Complexity

Simple moves to deep strategic planning provide a natural learning curve

Data Availability

Vast databases of chess games make training AI models more accessible

Our goal: Develop a CNN that can play with either colour, at any point in the game, predicting legal moves with special attention to openings and endgames.

What is a CNN?

A **Convolutional Neural Network (CNN)** is a specialised computer program that helps machines recognise patterns in visual data. It works similarly to how humans process images:

- Scans images in small parts to identify important details
- Recognises shapes, edges, and patterns
- Combines these features to understand the complete image

CNNs are widely used in facial recognition, self-driving cars, and medical imaging—helping computers "see" like humans do.

Why Use CNN for Chess Move Prediction?

Chessboard as a Spatial Grid

An 8×8 grid similar to image pixels, perfect for CNN's spatial pattern recognition

Pattern Recognition

CNNs excel at identifying chess patterns like forks, pins, and checkmate threats

Local Feature Extraction

Detects piece clusters, threats, and control over squares to predict optimal moves

Efficient Processing

Processes the board like an image, reducing complexity and focusing on important regions

Top chess engines like AlphaZero use CNNs to evaluate positions and choose moves like grandmasters.

Our CNN Approach

We're creating a **Convolutional Neural Network** to predict chess moves, similar to AlphaZero and Leela Chess Zero, but with key differences:

- Instead of self-play learning, we'll train using human-played games
- Need to represent the chessboard and moves in AI-friendly format
- Must create a diverse dataset of chess positions
- Ensure suggested moves are both valid and strong

This approach presents unique challenges but allows us to leverage human expertise in training our model.

Representing the Chessboard for AI

Single Matrix Problem

Using one matrix with different numbers for pieces (e.g., 6=king, 1=pawn) causes the CNN to misinterpret piece importance

Better Solution

Use separate binary matrices (0s and 1s) for each piece type—one for white pawns, one for black knights, etc.

AlphaZero Approach

This multi-matrix approach treats all pieces equally and improves move predictions by focusing on positions

Preparing Moves Data

AlphaZero uses a 119-channel matrix to track all possible moves. We'll use a similar approach:

Representing the Best Move (Target)

- Use matrix representation instead of simple notation
- Helps AI learn patterns of strong moves

Representing All Legal Moves

- Show all possible legal moves for every piece
- AI knows both piece positions and where they can move

Data Format: 77-Layer Matrix

1

Layers 1-12: Piece Positions

Separate binary matrices for each piece type and colour (white pawns, black knights, etc.)

2

Layer 13: Turn Indicator

8×8 matrix of all 1s (black's turn) or all 0s (white's turn)

3

Layers 14-69: Queen Moves

56 matrices representing possible queen moves in 8 directions (7 squares in each direction)

4

Layers 70-77: Knight Moves

8 matrices representing the knight's unique L-shaped movement patterns

This format is stored efficiently using int64 compression, reducing dataset size from potential 10GB to about 1.5GB.

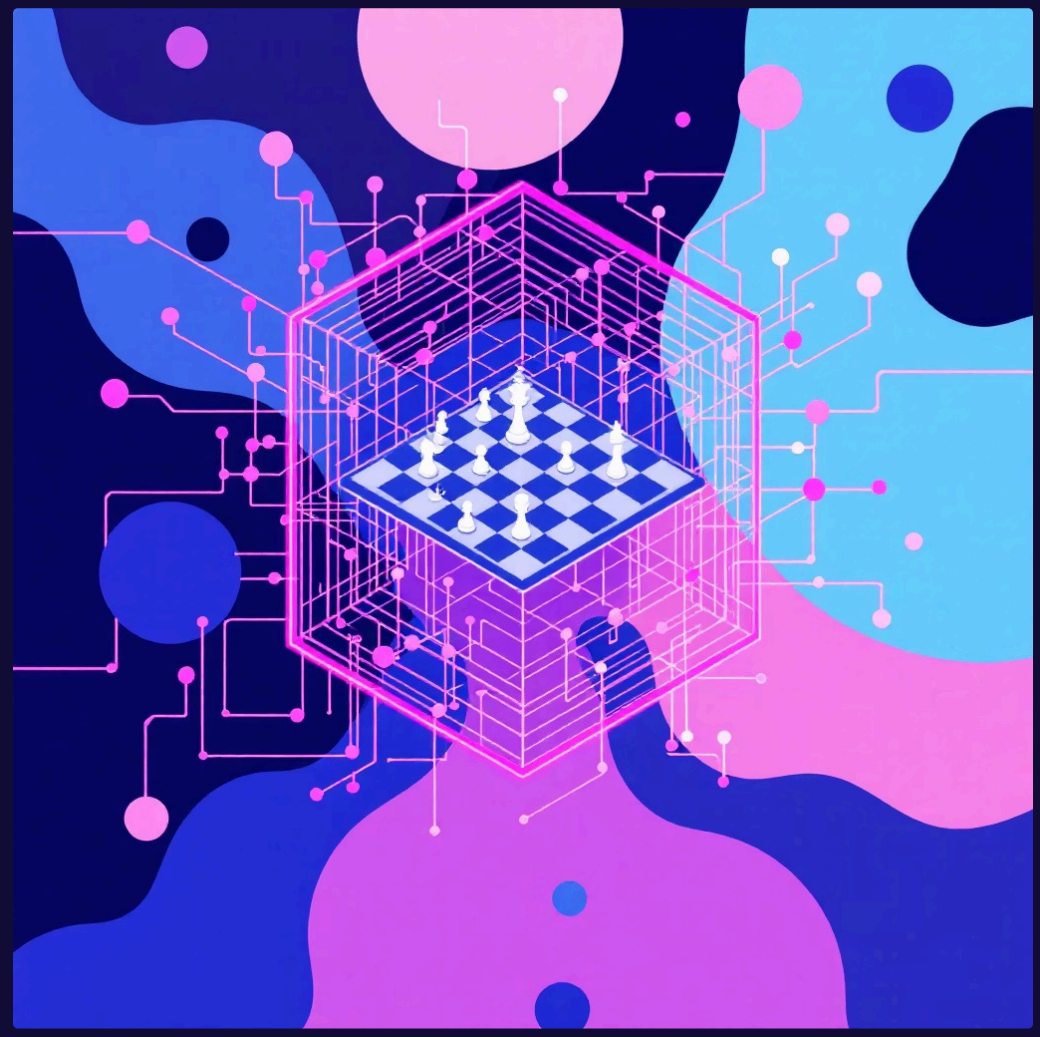
Why This Structure Works

Benefits for CNN Processing

- Hierarchical Features: Learn distinctive patterns for each piece type
- Spatial Information: Preserves relative positions crucial for chess
- Discriminative Learning: Recognizes unique patterns for each piece
- Flexibility: Easily adapts to changing board configurations
- Interpretability: Each matrix corresponds to a specific aspect

Move Representation Benefits

- Alignment with Board: Natural 8×8 grid structure
- Complete Information: Captures all possible moves
- Spatial Relationships: Maintains positional context
- CNN Compatibility: Ideal for convolutional processing



Implementation Summary

77

Input Layers

8×8 matrices representing board state and possible moves

4

CNN Layers

Convolutional layers with batch normalization

4096

Output Moves

Possible move combinations the model can predict

1M

Training Positions

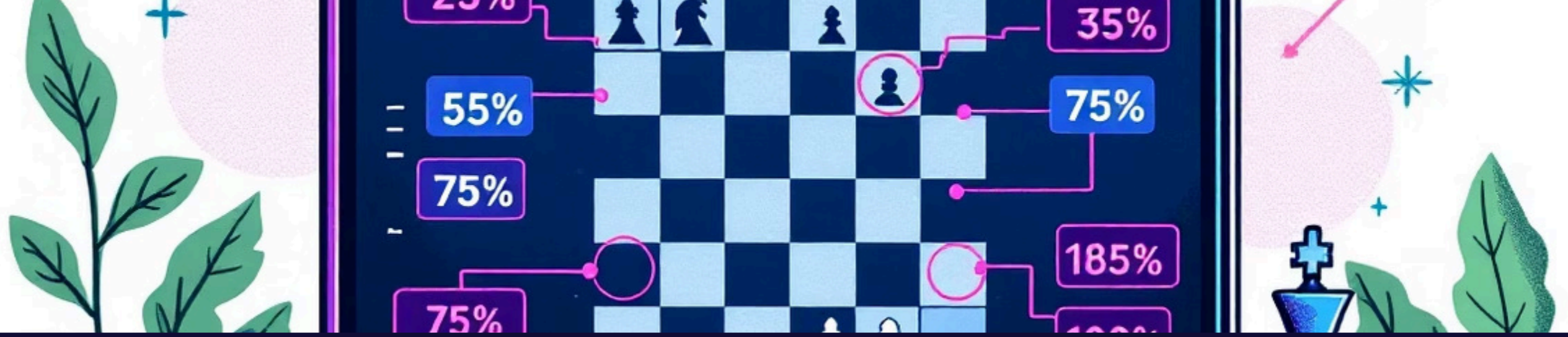
Chess positions available for training (starting with 200k)

Our implementation combines CNN with optional Monte Carlo Tree Search for look-ahead capability, deployable via Streamlit, Gradio, or Hugging Face Spaces with proper model saving techniques.

Training strategy: 80/20 train/test split, 64-128 batch size, 10-30 epochs with early stopping to prevent overfitting.

Monte Carlo Tree Search: Intelligent Game Strategy

A powerful approach for computers to play chess and other strategy games by intelligently exploring possible moves and selecting the best options.



How MCTS Works: The Basics

Making a Move Tree

Create a tree where each branch represents a different possible move and response.

Running Simulations

Play thousands of random games from each position to test outcomes.

Checking Results

Count how often each move leads to wins, losses, or draws.

Choosing the Best Move

Select the move with the highest probability of success based on simulations.

The Challenge of Chess

A chess game tree is **incredibly vast**, with too many possible moves to explore fully:

- ~50 legal moves in a typical position
- After just two moves: 2,500 different positions
- Even powerful computers cannot analyze every possible game path

Instead of attempting an exhaustive search, MCTS focuses on exploring the most promising moves through continuous exploration and learning.

Balancing Exploration vs. Exploitation

Initial Random Selection

At the beginning, MCTS selects moves randomly from all legal options, treating them equally.

Learning from Success

As simulations continue, the search begins to favor moves that have led to successful outcomes more often.

Maintaining Diversity

Occasionally selects less-explored options to ensure no potentially strong move is overlooked.

This balance between focusing on known strong moves (exploitation) and testing new possibilities (exploration) is what makes MCTS powerful.

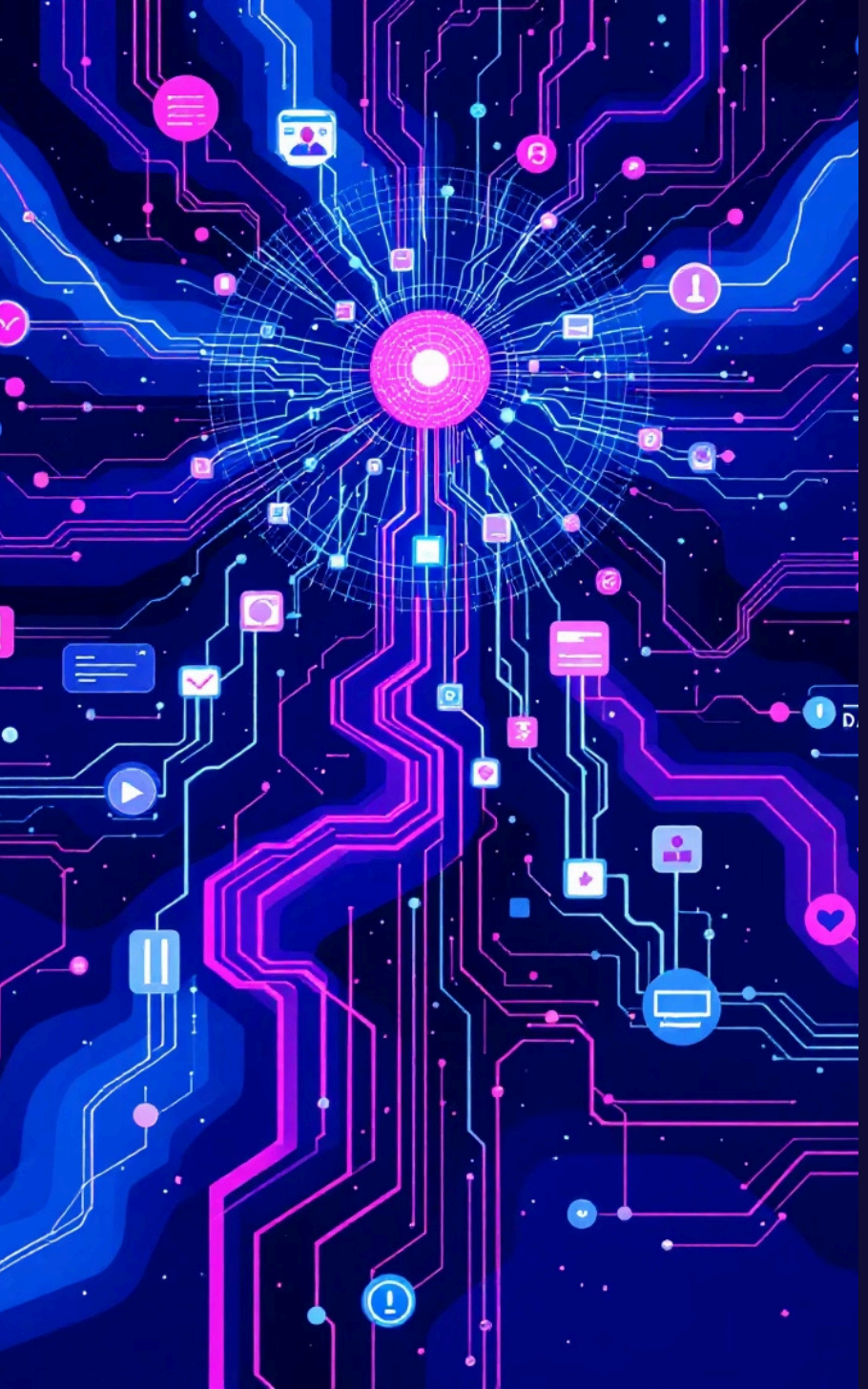
Mathematical Decision Making

To manage the exploration-exploitation balance, MCTS uses mathematical formulas like the Upper Confidence Bound (UCB1) to decide whether to:

- Explore a new, less-tested move
- Reinforce an already successful strategy

This helps MCTS refine its search intelligently, improving decision quality over time without requiring exhaustive calculations.

By repeatedly simulating games and adjusting its choices, MCTS efficiently finds strong moves, even in complex games where brute-force search would be impractical.



Our Enhanced Approach: Neural MCTS

Combining MCTS with Neural Networks

The Problem with Standard MCTS:

- Early moves chosen randomly
- Requires many simulations to reach strong conclusions
- Inefficient exploration of weak moves

Our Solution:

- Neural network guides MCTS toward better moves from the start
- Reduces random choices
- Makes each simulation more meaningful

Smart Move Filtering

Standard Position

~50 legal moves available

AI Filtering

Selects only 3-10 best moves

Focused MCTS

Deeper analysis of quality moves

This approach significantly reduces the number of branches MCTS needs to explore, making simulations faster and more efficient.

Risk: If the AI fails to include the best move in its selection, MCTS will never consider it.

Implementation Details

Our MCTS Implementation

We provide the code to deploy our enhanced MCTS system. You can adjust input variables to optimize performance, but be cautious:

- Some changes may slow down the system
- Others might make it impossible to run on your machine

The implementation balances computational efficiency with strategic depth, creating a powerful chess-playing system.

Deployment Process

Web Service Creation

Gradio creates a web service accessible via browser and as an API for other applications.

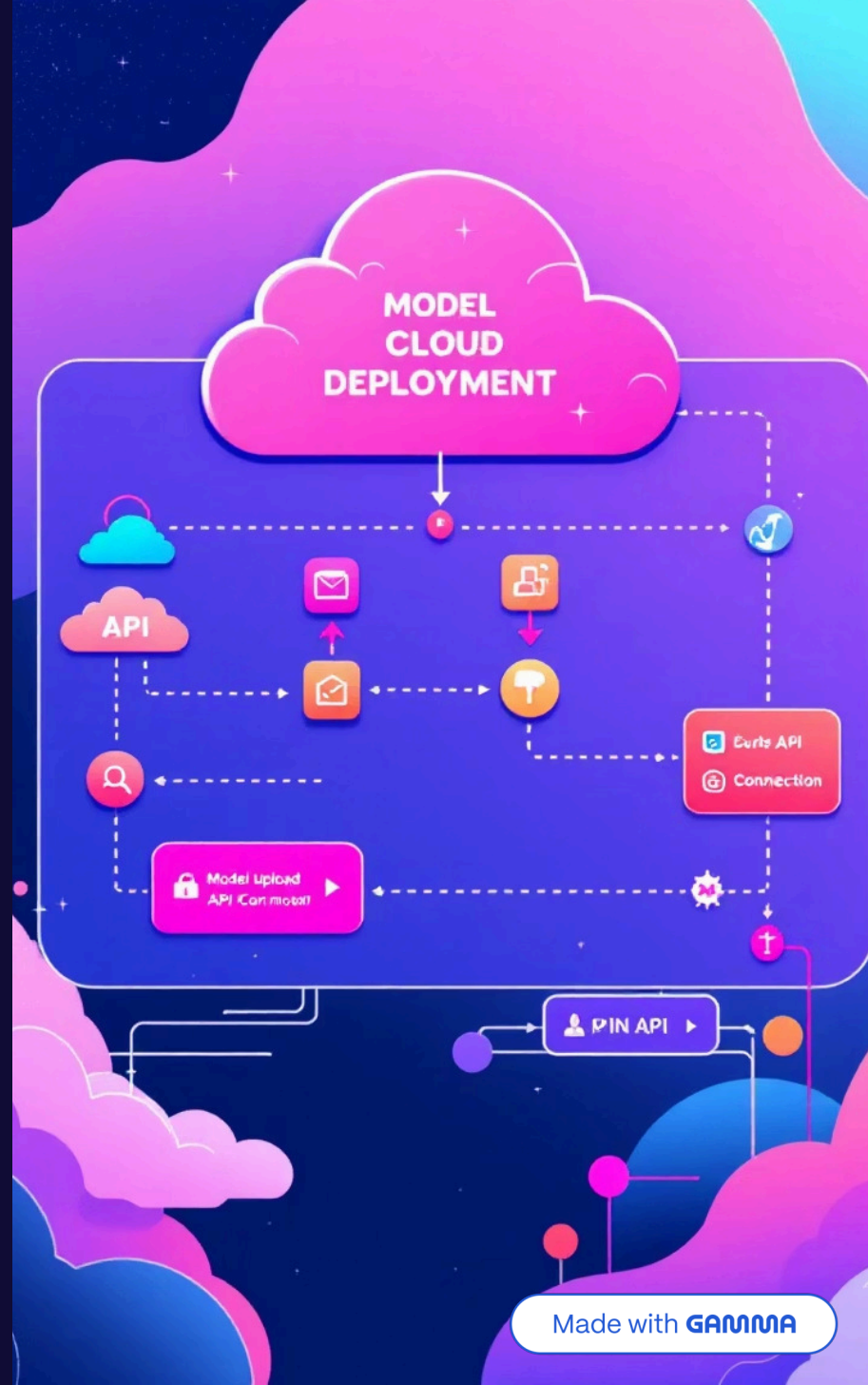
Model Uploading

We use Huggingface to upload the trained model, which will be downloaded by a Google Colab Notebook.

API Integration

The deployed model serves as an API for your AI player in competition.

We provide the deployment code and don't recommend modifying it. Copy the URL and paste it into our web platform to test it.



Competition Setup

How the Competition Works:

- Choose your AI player and opponent
- Players compete with limited time per move
- Multiple games determine the best AI

Your trained AI will be downloaded from Huggingface to compete, showcasing the effectiveness of your implementation of Neural MCTS.

This competition format tests both the strategic depth of your AI and its ability to make decisions efficiently under time constraints.