Alpha Go Mini

Jason Katz, 07/2025

Project Links

- **Website:** https://alpha-go-mini.vercel.app/
- **Algo Repository:** https://github.com/katzjason/go-algo
- **Backend Repository:** https://github.com/katzjason/go-backend
- **Frontend Repository:** https://github.com/katzjason/go-frontend

Project Overview

Large language models like GPT-4 and Gemini 2.5 Pro are unlocking transformative scientific and societal advancements—from accelerating drug discovery through simulated clinical trials to scaling education with hyper-personalized AI tutors; it's no surprise that OpenAI is projected to spend over $500 million per 6-month training run on its upcoming GPT-5 model.

Alpha Go Mini was born out of my curiosity to see how far one can go with a **minimal**, **cost-efficient AI pipeline** using industry standard tools like PyTorch for deep learning and AWS EC2 for GPU compute. I chose Go, specifically on a 9x9 board, as the problem domain for several reasons:

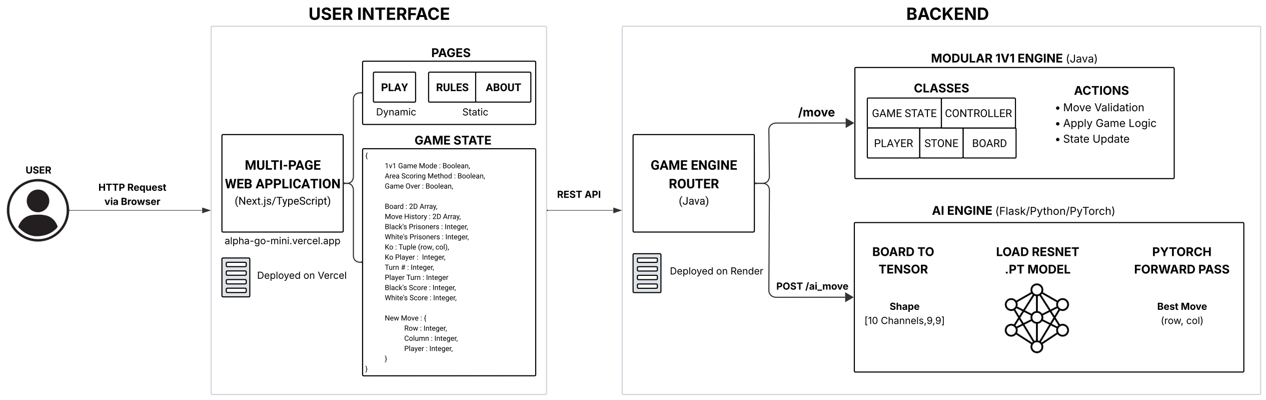
1. **High Usability:** The interactive nature of board games allows users to assess the performance of the model directly by playing against it—an *intuitive* and *visual* approach unlike traditional evaluation benchmarks.
2. **Tractable Complexity:** While 9x9 Go has significantly fewer legal positions (~10^{17}) than 19x19 Go (~10^{172}) or chess (~10^{50}), it still presents a challenging learning problem.
3. **Game Unfamiliarity Favors the Model:** While most people in the United States are familiar with chess, fewer are familiar with Go, **boosting my model's chance of winning!**

Below I outline my approach for developing a **supervised machine learning model** that outputs patterned moves while being **constrained to <$15 in cloud compute costs**.

Tech Stack

- **AI Model:** PyTorch, Python, Flask, Weights & Biases, AWS EC2, Render
- **Game Engine/Backend:** Java, Spring Boot, Maven, Render
- **Frontend:** Next.js, TypeScript, Tailwind, Vercel

System Architecture Diagram



```
{
        1v1 Game Mode : Boolean,
        Area Scoring Method : Boolean,
        Game Over : Boolean,

        Board : 2D Array,
        Move History : 2D Array,
        Black's Prisoners : Integer,
        White's Prisoners : Integer,
        Ko : Tuple (row, col),
        Ko Player :  Integer,
        Turn # : Integer,
        Player Turn : Integer
        Black's Score : Integer,
        White's Score : Integer,

        New Move : {
                Row : Integer,
                Column : Integer,
                Player : Integer,
        }
}
```

Alpha Go Mini consists of modular, microservice-based architecture with a decoupled user interface and backend. These services communicate exclusively via HTTP APIs—offering clean separation of concerns and enabling independent development and deployment. Data collection begins when the user visits the Vercel-hosted site and starts the game. In doing so, the chosen game mode (1v1 local or AI opponent) and scoring method (area or territory) are added to the collection that represents game state. When the user makes a move, a JSON payload representing the game state is generated and is sent to the backend's exposed '/move' or '/ai_move' endpoints via REST API.

The backend is divided into two core microservices that are each deployed on Render:

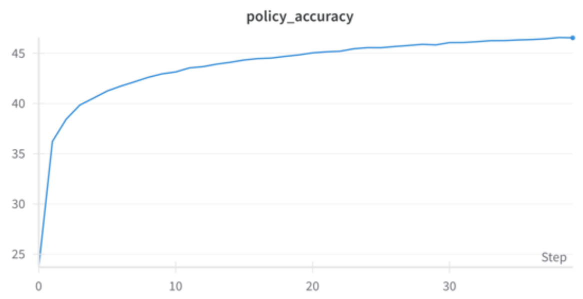
1. **Local 1v1 Engine (Java/Spring Boot):** Using endpoint '/move', this service handles 1v1 games between two players on the user's local machine. The source's JSON payload is parsed and reconstructed as a game state within the Java game engine. Classes like the *Board*, *Stone*, and *Player* support core logic of the game, *Game State* joins it together, and *Controller* communicates back to the user interface. If the user's move is valid, the game state is updated and relayed back to the user interface for visualization on the game board. If not valid, the request fails silently and the game state is not updated, meaning that the user will have an opportunity to select a different move.
2. **AI Engine (Flask/Python/PyTorch):** Using endpoint '/ai_move', this service handles games versus the trained AI model. First, the game state is converted into a tensor of shape [10, 9, 9], where 10 channels of size [9,9] represent the state of the board. The model, implemented in PyTorch, is served using Flask and hosted via Render and predicts moves with low latency. It loads pre-trained .pt weights, generates a probability distribution over all legal moves based on the current board state, and returns the move with the highest calculated probability of winning. This move is reflected in an updated game state and is relayed back to the user interface.

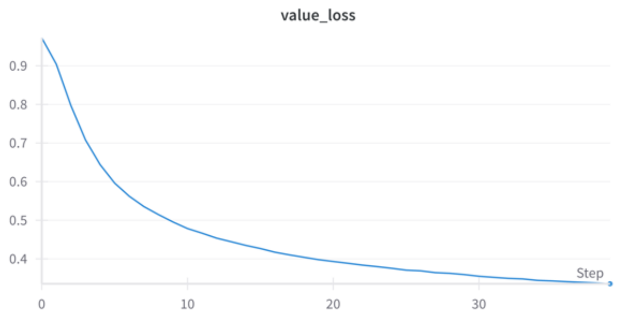
This pattern of data flow between the user interface and the backend continues until a game-over condition is reached, at which point the services no longer process new requests.

Model Design and Performance

Building the model consisted of four main steps: preprocessing the training data into tensors, creating the model architecture, running the training pipeline, and evaluating performance.

1. **Data Preprocessing:** Training data was obtained from this online-go-games repository, where games were stored in SGF format allowing for straightforward parsing using the Python sgfmill library. The downloaded dataset initially had 80k records, but this was filtered down to 8k based on two criteria: 1) games were played on a 9x9 board, and 2) players had a dan ranking above 10k, meaning they were at least intermediate amateurs. This filtered dataset was then preprocessed in batches and transformed into standardized tensors for model training. The batch preprocessing technique allowed me to run this overnight on my local CPU without exceeding working memory constraints, preventing unnecessary spending on EC2 and allowing for scalable reuse.
2. **ResNet Architecture:** The model was developed using a ResNet-based, dual-head neural network and was inspired by the approach taken by Deepmind's AlphaGo in 2015. The network takes a 10-channel tensor representing the board's state as input—supplying enough information for pattern-recognition without incurring elevated training costs. The model then utilizes an initial convolutional layer (nn.Conv2d) followed by two residual blocks; each residual block applies two stacks consisting of a convolutional layer (for pattern recognition), batch normalization (for standardization), and ReLU activation (adding non-linearity), with skip connections to mitigate vanishing gradients. Then, the architecture branches into two heads. The **policy head** outputs a probability distribution over the 81 board positions by using another convolutional layer followed by two layers of linear transformation. The **value head** outputs the probability of winning from the current state by also using a convolutional layer followed by two layers of linear transformation. After adding non-linearity, the tanh function is applied to output a value within [-1, 1], with -1 representing a 100% chance of losing and 1 representing a 100% chance of winning.
3. **Training Pipeline:** The supervised model was trained over 40 epochs on AWS EC2. Each training loop iterated over all 8k game samples, each of which included a board state, labeled move (policy head), and game outcome (value head). A cross-entropy loss function was used to help the policy head learn; board positions were treated as categorical targets, and the loss grew exponentially the further the prediction was from the correct move. Similarly, a mean-squared error loss function was used on the value head since the scalar win probability could be compared to the labeled game result using linear regression.
4. **Performance:** Training performance was logged in real-time to Weights & Biases, enabling me to visually monitor policy accuracy and value loss which was especially helpful during early training runs and for identifying performance plateaus. Ultimately, my model had 46.5% policy accuracy and 0.33 value loss on the 80% training set, and a 41.1% policy accuracy on the 20% testing set.


policy_accuracy
Step


value_loss
Step