Go Learning Buddy

Authors: Nathaniel James, Israel Escobar-Camacho, Hang Shu
Affiliation: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—Go Learning Buddy provides a physical board for Go players to play on, which can be set to indicate the best move for the current position, along with a platform for analysis post-game. These features help players learn and develop strategies for when they play. The physical Go board reads in the game state and sends the data to a computer back-end, which sends back the best move from the engine to be displayed on the board's LEDs. Each move is automatically stored, so that users can analyze their games with the engine post-game.**

***Index Terms*— Convolutional Neural Network, Monte Carlo Tree Search, Policy Network, Reinforcement Learning, Value Network**

1 INTRODUCTION

Go, also known as Weiqi or Baduk is an ancient game that has been played for millennia. As such, it has developed an immense following all around the world, but especially in Eastern Asia. Despite the games elegance and simplicity, due to the boards large size (19 x 19) as well as the ability to capture and replace stones, the amount of possible positions is incomprehensibly large. Players can and do train for years, the best of them playing in domestic and international tournament, followed by millions of fans. Our project is designed with these trainers in mind.

In order to learn from their training matches, beginning and intermediate Go players wish to see which moves they made were strong, and which were blunders. Beyond this, game notation is difficult and time consuming, as there are 361 different intersections on the board to place stones on, and each placement comes with the possibility of captures. Thirdly, in the interest of training, and reaching more advanced board positions, players in a match might want to see the best move(s) in the position.

Our product solves all three of these problems at once. Two players can play against each other on our custom-built Go board and each move will be registered by sensors on each intersection. These moves will be transmitted to our software web application, where a Go engine trained with self-play reinforcement learning will use Monte Carlo Tree Search (MCTS) to calculate the best response. These response will be sent back to the physical board, and the best move will be displayed by LEDs lighting up on the proper row and column. Finally, as each move is made, it will be stored by our web application, so that players can look at their game history, and see move suggestion from the engine at each point in their previously played games.

Of course, Go engines are available to play against and train with online. Players can find opponents online, and analyze their game history on websites. But, there is no current way to play against an opponent over-the-board, while receiving analysis and storing positions for further analysis later. Such technologies exist for game like Chess, but none for Go, and that is what our product provides that no one else can.

2 USE-CASE REQUIREMENTS

The modified Go board is designed with specific use-case requirements that emphasize precision and responsiveness. To fulfill these requirements, it must possess the capability to accurately identify the black and white tiles on the board with 100% certainty. Additionally, it needs to rapidly assess the entire game state, accomplishing this task in under 50 microseconds.

Furthermore, the Arduino unit to which the Go board is connected to is expected to transmit the game state to the COM port within a time frame of less than 120 milliseconds. Similarly, the move suggestions sent from the COM port to the Arduino should be done in under 300 milliseconds.

The self-play reinforcement learning engine is programmed to operate at a skill level similar to that of an amateur 5 dan Go player. It will present the five most promising moves for any given position on our web application. The top-rated move among these will be chosen and relayed as the recommended move to be displayed on the physical Go board. For each of the five optimal moves presented on the web application, a percentage will be provided to estimate the engine's belief in the likelihood of winning the game if that specific move is performed. The calculation required to find the 5 optimal moves should be done in less than 3 seconds.

In addition to these features, our web application must offer a dynamic visualization of live gameplay on the Go board, as well as the move suggestions generated by the Go engine. These real-time updates on the web application should occur in less than 200 milliseconds (not including the time it takes to generate an output from the Go engine) and must maintain a precise representation of the ongoing game. Furthermore, the web application must have the capability to save and retrieve past Go games with 100% accuracy, ensuring the integrity of historical game data.

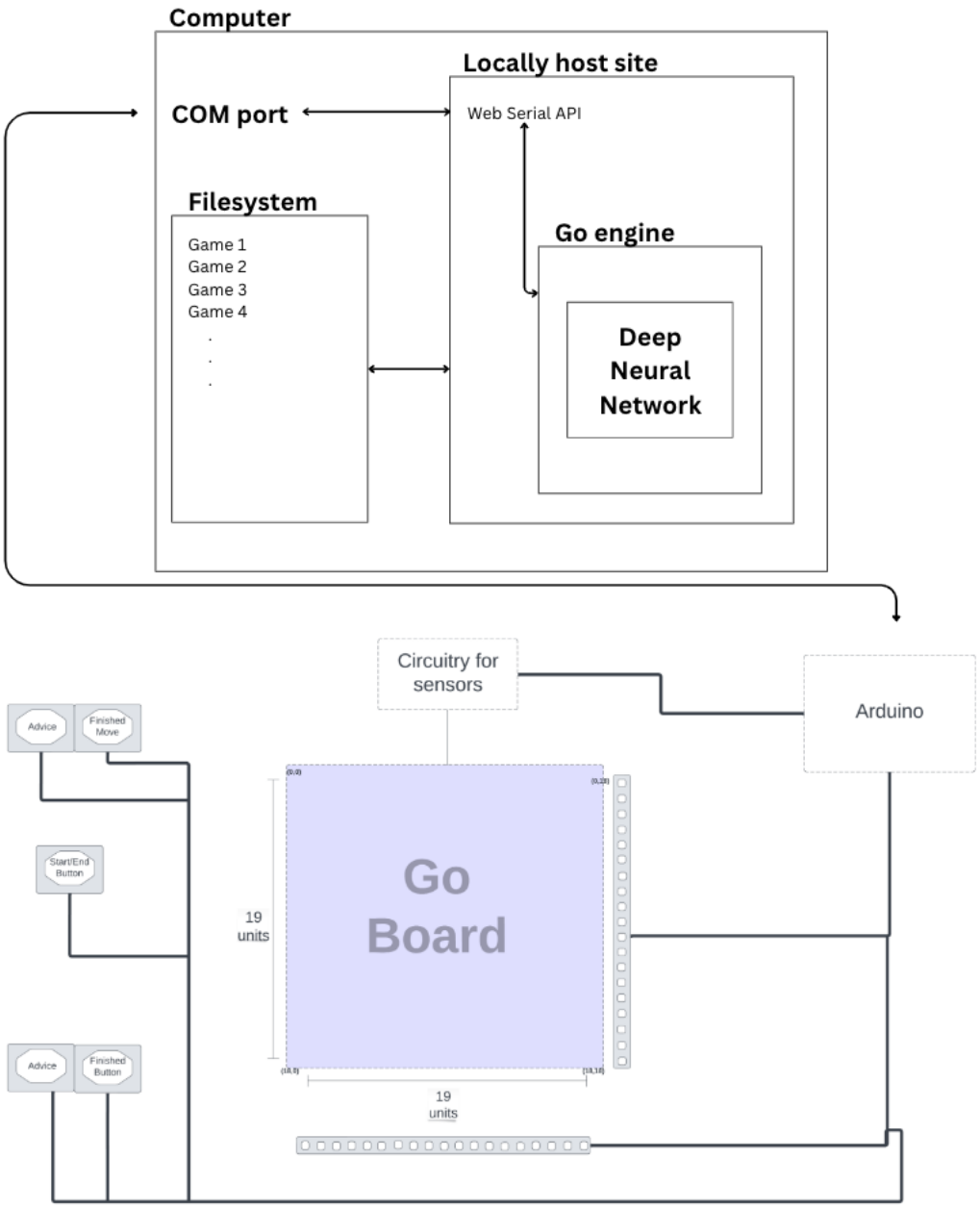


Figure 1: Overall system architecture

3 ARCHITECTURE AND/OR PRINCIPLE OF OPERATION

As can be seen in Fig. 1, users will play Go on our physical board, which reads in the game state through light sensors covering each of its row-column intersection. This game state is captured by our Arduino microcontroller, and is then sent to the computer's COM port when a user has clicked "Finished Move" button on the board. The web application on the computer then reads in the game state via web serial api and feeds the game state into the Go engine. The Go engine will output the 5 best moves move the position, in order of strength, and the best move will be sent back to the Arduino via the same COM port. If the users clicks the "Advice" button, then 2 LEDs corresponding to row and column of said move, will light up on the Go board, showing the user which move the engine recommends. When a Go game is finished, the user can click the end button, which will be conveyed to the web application. When the game is finished, users can download the game into their computer's file system and load it in later into the web application to view the history.

4 DESIGN REQUIREMENTS

4.1 Hardware

For many of the Hardware design requirements, we largely focus on performance and latency of our implementation to fit the needs of competitive players. This means being 100% accurate for our game state analysis at 50 Microseconds. Because our routine for light sensor configuration is very similar, we are requiring that our configuration stage also hold at the same speed before a game starts.

In terms of our communication to our COM port, we are requiring that we send information, such as the game stage, as a rate of 120 Microseconds as fast as our serial communication constrains us. We require for communication for advice to be in 300 Microseconds as this would require sending a request through the COM port while receiving a advice for players usage.

4.2 Software

Since users will directly interact with the web application, the design requirements are almost 1:1 with the use case requirements. For example, to reach the less than 200 milliseconds refresh rate set in the use case requirements, the rendering of each component in a specific page, along with all the computations necessary to generate each component, must sum to less than 200 milliseconds. To be able to save and retrieve past Go games with 100% accuracy, the data structure holding the game states but must be updated with 100% accuracy as the live gameplay is read in from the physical Go board.

4.3 RL Engine

Quantitative design requirements are especially important for the RL Engine component, as it has quite a few moving parts, and contributes by far the most latency that the user will experience. In our use-case requirements we have stated a 3s limit between move input, and recommendation, as this is short enough to not hinder gameplay, but long enough to allow for a proper MCTS simulation. As such, our MCTS must take no longer than 2.5 seconds, as if it does, even if all other component meet requirements, the total latency will be above 3 seconds.

The engine strength requirements are well documented in section II, but to achieve that level we require a MCTS depth approaching 1600, as this is what top engines use today, when time and hardware limited.

5 DESIGN TRADE STUDIES

5.1 Hardware

As users make a move at any moment, we need to keep record of any new piece that is added on the board. This means that we have 361 sensors to gather data on and send to the COM port. Our micro controller can only support 16 analog input ports though. This would mean that in order for us to read each port, we would have to poll 16 sensors for a given 23 cycles. This polling would mean that we would select different sections, requiring binary conversion of the cycle for multiplexer selection inputing.

$$\text{binary\_solution} \leftarrow \text{cycle\_selection} \tag{1}$$

$$\text{mux\_segment\_i(.sel(Binary\_solution))} \tag{2}$$

Depending on embedded software coding for the microcontroller, the polling during a single cycle can be as fast as 10^-6 s (as this is the clock frequency of our Arduino) but the slowest expected duration would be $50 * 10^-6$ s. This analysis would mean that, our testing for our data retrieval would be as follows

$$n_c * t_e + t_{delay} = t_{total} \tag{3}$$

Where n_c is the number of cycles, t_e is time of execution, t_{delay} is the button time delay, and t_{total} is the total data retrieval time.

5.2 RL Engine

As mentioned in section IV, the entire MCTS process must take under 2.5 in order for our analysis to meet timing requirements. Through experimental observation, we have determined the time per simulation t_s corresponds to the number of simulated moves m roughly as follows.

$$t_s = 6.9 * 10^{-6} + .0003m \tag{4}$$

This means that even assuming a larger depth than required of 2000 (the requirement is 1600), a full simulation

would take just over .6 seconds giving lots of space for other processes running long, extra caclulations in particularly complex board states (i.e. lots of captures) etc.

5.3 Software

Rendering an NxN board has a time complexity of $O(N^2)$ as the rendering is done by reading in a flatten version of the NxN board and converting the information into NxN tiles to be rendered. This means that each time a live update is sent from the Go board to the web application, an $O(N^2)$ operation is performed.

When loading in a saved game, users can pick which move to display on the web application. Because the board is saved in the following state: array of length 361 with elements being tuples describing ("W" or "B" or "E", -1 or the move number) where "W", "B", and "E" correspond to white, black, and empty, when the user selects a specific move to display, we need to generate the game state at that specific move. This will be an $O(N^2)$ operation as we need to scan through each element, collecting only the moves <= the specified move.

Since we are playing Go, N is fixed to 19, so all of these operations are technically constant, however, these constant multipliers do matter in the real world as we want latency to be minimal.

6 SYSTEM IMPLEMENTATION

Our system implementation can be split into three parts: hardware, software, and RL engine. How these section interact is shown in Fig. 2. Accordingly, for each subsection and sub-subsection, the specific system implementation is given below.

6.1 Hardware

6.1.1 Board Development and Physical Alterations

For our physical board development, we would need to use an entirely custom hollow board to support our electronics. This board would be the same length and width of a conventional GO board (454.5 mm by 424.2 mm), with the addition more of height so our Arduino can go in (greater than the standard 151.5 mm).

For this to be possible, we will take advantage of laser cutters to cut out wood panels and make holes for our light sensors to detect where pieces are placed. We made a DXF design file to instruct the laser cutter follow such requests (Ref. 12.1.1). At this same time, we developed finger joints for these crates for easy assembly that wouldn't prolong our implementation process.

6.1.2 Circuitry Design and Development

When working on circuitry design, we applied light sensors on every hole a piece could be placed with a vector board connected to these sensors. As mentioned before in 5.1, we would need to have multiple iterations of sensor polling as there are a limited number of ports. To compensate for this, we would use 16:1 multiplexeres that will connect to 16 light sensors as shown in Figure 2.

Now that all the light sensors have been designated assigned to a 16:1 multiplexer out of the 22 we will be using. We will need to apply another series of multiplexeres as, again, we only have 16-analog ports to receive data through. This is our designation for the multiplexeres:

- Arduino analog 0-11 port
  - 16:1 multiplexer components 0-11
    - light sensors 0-191
- Arduino analog 12-15 port
  - 2:1 multiplexer components 0-5
    - 16:1 multiplexer components 12-22
      - light sensors 192-360

We will use the Arduino's digital pins 0-3 for the 16:1 multiplexeres selection port. In addition, we will use digit pins 4 for our 2:1 multiplexeres selection port which result we would like to switch from: either 16:1 multiplexer components 12-15 results or components 17-22 results.

In addition to the data retrieval that we have designated and ported, we will have buttons (specified as hexagon blocks in our figure 2) for users to start and end games, inform when a turn has been finished, as well as allow players to be shown advice mid game. These button/swithes will be connected to digital pins D33-37.

We have also added 38 LEDs to signify the x and y coordinate of a preferred move. These LEDs will be connected in series to each digital Arduino pin D5-23 for Y coordinate and D24-D32 for X coordinates. Each LED will be connected in series with a resistor to regulate current.

6.1.3 Data Retrieval and User Interfacing

For our arduino in sumation, we will have port designations of these sorts:

- analog 0-15 data retrieval sensors (inputs)
- digital 0-3 data control 16:1 mux selections (output)
- digital 4 data control 2:1 mux selections (output)
- digital 5-23 Y cooridnate LEDs (output)
- digital 24-32 X coordinate LEDs (output)
- digital 33-37 Button status (inputs)

For our data retrieval, we will be focusing on grabbing one section of data in 16 cycles. Our software will record 16 data values at a given cycle in a array. All our multiplexer selections will be initialized at zero and our selection ports for our 16:1 will increment from 0-15 in binary. Once we have recorded data from 16 different addresses (totalling to 255 datas retrieved), our software will then have digital

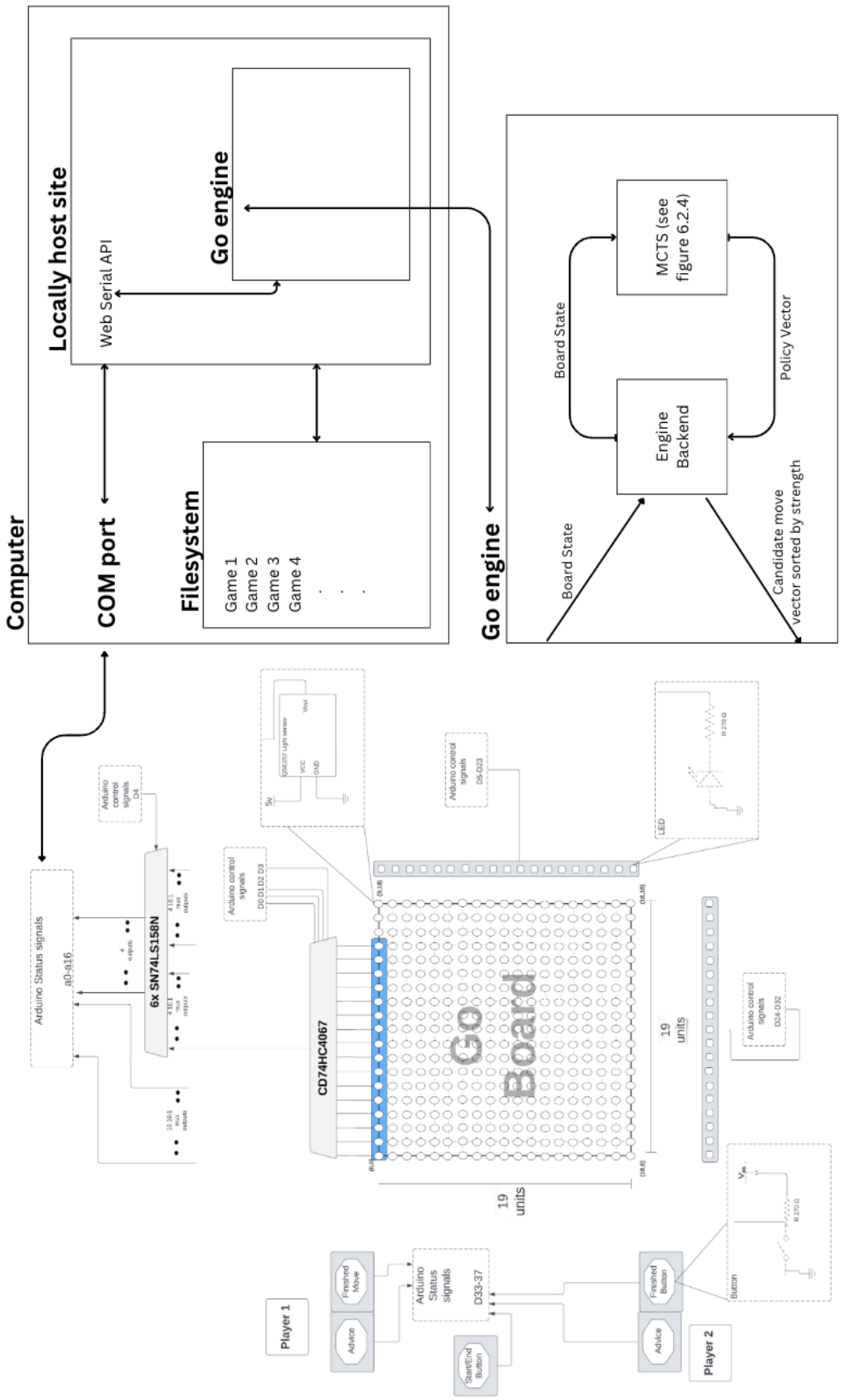


Figure 2: Overall system implementation diagram

pin 4 as high and go through another 16 cycles to retrieve data, but only from analog ports 12-15 instead. The analog pins 12-15 will have a new selection of light sensors selected now, the other ports are unessential results. Once we have completed another 16 cycles, our array will be full of all 361 values and ready to send to COMS port as text values.

For our user interface, we will be using a advice buttons that will trigger the arduino to light up 2 LEDs (one on the x coordinate and one on the Y coordinate). The micro controller will know which ones to turn on by having a pre loaded move to advise the player on.

We will also have buttons of when a turn has been finished for micro controller to trigger a data retrieval action. We will also have our start/end button trigger preconfiguration phases where we find the reference value of our light sensors so we know what values we are to expect if a piece is placed or not on a spot.

6.2 RL Engine

The Go engine built using self-play reinforcement learning will be implemented in two parts: training and usage in analysis. Monte Carlo tree search will be used in both parts, along with the policy and value networks.

6.2.1 Policy Network

The policy network is a convolutional neural network (CNN). The input is a length 361 vector (the flattened board-state), and the output is a length 361 vector of probabilities that sum to 1, meaning that output[0] represents the relative strength of placing a stone on the intersection of the 0th row and column (assuming 0-indexing). The initial weights of this network are determined by a gaussian distribution with mean 0.

6.2.2 Value Network

The value networks is another CNN. The input is the same length 361 vector, and the output is a singular scalar value, representing the players chance of winning from that position. The initial weights for this network will be trained via regression, using a data set of board-states pulled from expert level Go matches, tagged with the outcome. This is because from section V a full game simulation takes about 2 seconds, so it is faster to pull initial training data from a huge database where the quality of match is high (higher quality of matches allows for the network to have a better initial grasp of positional strength) than too simulate enough times to develop a strong value network independently.

6.2.3 MCTS

From any given board state, our engine determines the next move via MCTS. Let us first define a few variables for simplicity of notation: exploitation score (or average strength of position) as Q, exploration bonus (score for adding new information to the tree) as u, board state as s, action (or move) as a, optimal action as a_t, number of states in the MCTS tree as N, the number of states in a given sub-tree of the MCTS tree as n_s, the nth substate of a state s as s_n, the probability said substate is reached as π_n, the policy network serving as a function as p, the value network serving as a function as v, and the balancing constant c. We then are trying to maximize the quantity $Q + u$, yielding

$$a_t = argmax(Q(s,a) + u(s,a)) \tag{5}$$

$$Q(s,a) = (\sum_{i=1}^{n_s} s_n * \pi_n)/n_s \tag{6}$$

$$u(s,a) = P(s,a) * c\sqrt{n_s}/(1 + N(s+a)) \tag{7}$$

Essentially, the exploitation score increases as more positive board states are reached, and the exploration score increases when less explored branches are expanded. The hyper-parameter balancing constant c is chosen depending on the amount of exploration desired (exploration is more encouraged in early game, and less encouraged in the end game).

From these equations, it is also clear that to maximize, only the "strongest" move (from the policy networks point of view) from each leaf node should be evaluated. The following steps are repeated a constant number of times (initially 1600 as this is the industry standard, but if this is too prohibitive on training / execution time it can be reduced):

1. Identify the strongest move from each leaf state in the MCTS tree using the policy network.
2. Using (5), find the state-action pairing with the highest score.
3. Add this move to the MCTS tree, and repeat.

After these steps are completed the requisite number of times, the local policy (i.e. the next move) is determined by the normalized state visit count over the course of that particular simulation.

6.2.4 Training Implementation

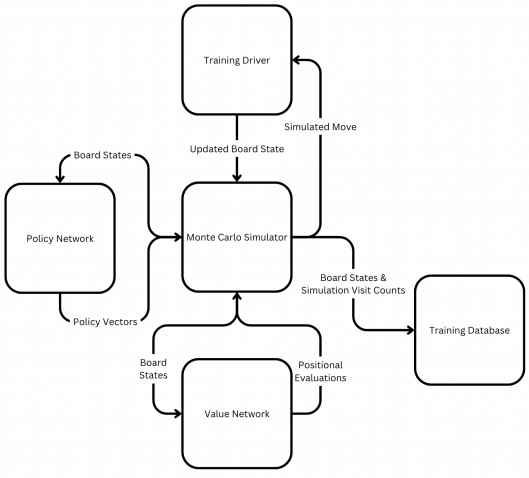


Figure 3: Training flow

The training driver maintains the current board state, and sends prompts to the simulator to continue the game until it reaches completion. The simulator uses the steps described in 6.1.3 to generate a policy vector from visit counts. The strongest available move is sent back to the driver, which updates the board state. The resultant policy vector is stored with the inputted board state as training data for the policy vector, and after the simulated game runs to completion, all board states are tagged with the outcome, and stored as training data for the value network.

Each simulated game generates upwards of 5000 board state pairings (as there are no early resignations common in human vs. human matches), so after each set of 200 training matches, upwards of 1 million training data points have been generated. The policy network is trained to minimize cross-entropic loss between its output and the desired policy vector given a position (characterized by the MCTS visit counts generated from that position), and the value network is trained to minimize mean-squared error between its scalar output, and the result of the game that reached a given position (1 for win, 0.5 for draw, 0 for loss).

6.2.5 Analysis Usage

When engine analysis of a position is desired (whether mid-game or post-game) MCTS is run, using the saved weights of the policy network and value network. Same as training, the normalized visit counts represents the policy vector, but these data points will not be saved for further training. Once prompted by the back-end, the engine will send back the policy vector, from which the back-end will select the strongest move to display (if in-game) or a number of strong moves to consider (if doing post-game analysis on the website).

6.3 Software

The web application will be built from React and will show a visualization of the live gameplay of the Go board and allow users to visualize saved Go games.

6.3.1 The two pages in the web application

Figure 4: Page 1 of web application

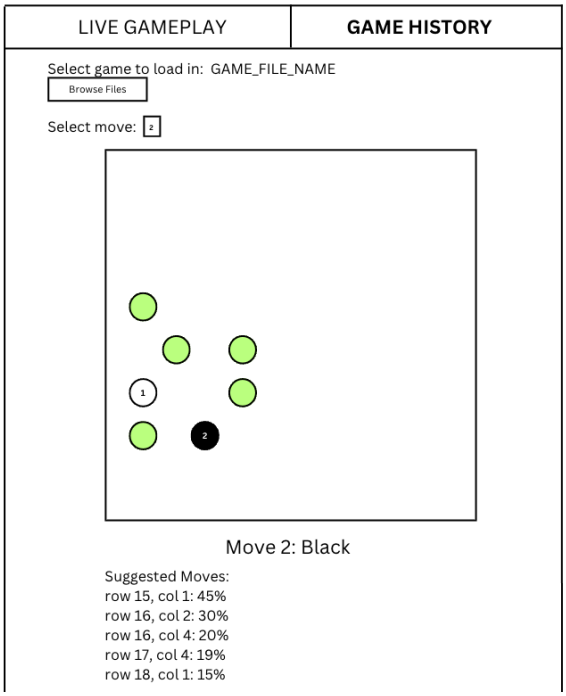


Figure 5: Page 2 of web application

The first page will be the page corresponding to the visualization of the live gameplay over the physical Go board. This page will allow users to see the Go game on the computer, as well as the 5 moves suggested by the Go engine. The page will also allow users to save the game by downloading a game information file to their computer's filesystem.

The second page serves to display saved Go games. The user can pick which Go game to load in as well as which move to view. For each move, the page will also display the suggestions made by the Go engine as well as the engine's

predicted probabilities of winning the game for each of the moves.

6.3.2 Visualization of the Go board

The Go board will be drawn with CSS and can be broken down into 2 parts: background and tiles. The background will simply be a brown square covering the entirety of the board as the background. The tiles have 9 different categories: top left, top, top right, middle left, middle, middle right, bottom left, bottom, and bottom right. The tiles are broken down into such categories because the tiles are divs and in order to draw in the black and white go pieces, the pieces must be centered on each div. Since the pieces fall on the intersection of lines and not in between, dividing the tiles into such categories with lines drawn inside each div and not as the border of each div makes drawing the go pieces easier.

6.3.3 Saving and loading Go games

In the background of the web application, Go game states are handled as a 1d array of length 361. This array only contains the following information: “W”, “B”, “E”, or white, black, and empty corresponding to each index. As the game is running, another 1d array will be updated and this is the array that will eventually be saved. This array will contain tuples of the following format (“W” or ”B” or ”E”, -1 or move number). When the game is over, this array will be converted into a string, written to a file, and then this file can be saved into your computer’s file system.

In the second page of the web application, you can load in this saved file and select which move to show. Once a move is selected, the game state will be reconstructed from the file by reading in all the moves less than or equal to the current move number and fed into the Go engine for the recommended moves.

6.3.4 Interfacing with the microcontroller

To interface the web application with the microcontroller, the serialport javascript library will be used. The web application will connect to a specific device port given the device path and the device’s baud rate. Data can be read in from the microcontroller with serialport.pipe() function, and the engine recommended move can be sent to the microcontroller with the serialport.write() function.

7 TEST & VALIDATION

7.1 Tests for Hardware

One of the main focues for testing is Latency during our data retrieval phases, pre-configuration phase, and communication phase when receiving data and sending it.

For testing latency of data retreval and configuration phase, we will run Arduino made functions (1) to record the execution of our data retrieval actions and apply our previous equation in section 5.

```
millis()                                    (1)
```

For testing communication latency, we will post the execution time of before and after a transaction has been sent and received to ensure that there is no lag in communication that is driven by hardware.

As for accuracy of our hardware, we will put it through a series, 50-100, different states and make sure each game state is obtained and processed with no error.

7.2 Tests for RL Engine

Accuracy testing: Our requirement is an engine at or above the level of amateur 5-Dan, and engines of that level are able to be played against online. By pitting our engine against those manually, if our engine scores at 50% or greater, it has met its accuracy requirement.

Latency testing: This is also eminently manually testable, as the timing constraints are relatively lax at {3} seconds, almost all of which will be taken up by the MCTS. If all other components meet their requirements, the MCTS needs to take {2.5} seconds or lower, and this can be iteratively tested across different positions to make sure it is executing in time.

7.3 Tests for Software

For the web application, unit tests can be used to test the performance of the site as well as testing the correctness of the code running the website. The testing framework that will be used is Jest: a Javascript test runner. Unit tests can be written with the Jest framework for each React component in the web application, and can be written in a way to check the contents of the game files saved to make sure 100% accuracy is maintained. Specific operations, such as rendering a board state, can be benchmarked with Jest to make sure the 200 millisecond latency goal is reached.

8 PROJECT MANAGEMENT

8.1 Schedule

We have organized our schedule according to our roles as we state in Team Member Responsibilities in the following point. We have also include benchmarks, goals on the top of our scheduler for us to see and understand when we will do either intergration or expect certain deadlines. The schedule is shown in Fig. 12.2.

8.2 Team Member Responsibilities

Nathan: Primary responsibility is the training and implementation of the RL engine. Secondary responsibility is linking the engine with the analysis back-end.

Hang: Primary responsibility is the creation of the game history analysis front-end and back-end. Secondary responsibility is linking the analysis back-end to the engine and the physical board.

Israel: Primary responsibility is the creation of the physical Go board, along with the implementation of sensors to determine where stones are played. Secondary responsibility of interfacing the physical board data with the back-end.

8.3 Bill of Materials and Budget

Many of the costs for our project have gone for the sensory of the game state as we require many sensors for this project. Our micro-controller (Fig. 12.3 Arduino) is also of great cost. The schedule is shown in Fig.12.3.

8.4 Risk Mitigation Plans

8.5 Hardware

For our hardware implementation, we have many issues that could occur with our system overall. One of the risks we have and are currently taking is developing a board of our own for this project with the help of external personnel. Such risk to take has affected our time line, but if it continues to affect us further, we plan to shift to having no custom board but a board with holes. This would expose our circuitry and even be a user hazard but the functionality will still be as intended.

In addition, because we are ordering so many components and so many sensors, there is a high chance that some of these sensors can break in our circuitry due to misconnections or too much wiring to be exposed that could cause short circuiting. We have ordered additional sensors to take care of possible issues occurring.

Another issues we have taken into consideration is possible communications problems with the COMs port and their possible usage on other computers. If our Arduino can not communicate correctly with the computer, we will have pre-installed Operating systems on our computers to ensure that the Arduino has no problems what ever the software is used.

8.5.1 RL Engine

With regards to the engine there are three main possible issues. One is if training is taking a prohibitively long time, even with GPU acceleration, and the second is that the MCTS itself (which is involved in the training and will be the main performance bottleneck) is taking too long. In both cases, the search depth of the MCTS can be reduced, which will harm the engines strength, but only marginally. Due to the previously mentioned exploitation weighting (see equation x), the first steps taken in the tree search matter much more than the later steps. The third, more damaging issue would be if the engine doesn't play at a high enough level for player improvement, regardless of cause. In this case, we could use an open-source Go engine as a replacement. While this would remove the signal-processing component to our project, it would be more useful than having a non-working engine.

9 RELATED WORK

One part of our project that is similar to work that has been done before is related to our Go engine. There have been historical Go engines such as Goemate and Zen, but the one that everyone knows today is AlphaGo. AlphaGo was developed in 2015, and it was a pivotal moment for Go engines as it was the first Go engine to defeat a world champion in a 5 game match using novel techniques as it combined deep neural networks with MCTS. AlphaGo does not rely on predefined heuristics, but, instead, it starts from scratch, learning the game solely through self-play. This approach has enabled AlphaGo to achieve superhuman levels of play in these games and has had a profound impact on the field of AI and its applications in various domains.

Relating to our project's hardware component, a similar product comparison can be seen with Square Off Pro's chess board. Square Off Pro's chess board not only facilitates game recording but also enables players to engage in chess matches against artificial intelligence on the very board. Our project, however, aims to provide a similar service, but for Go.

10 SUMMARY

The goal of our project, Go Learning Buddy, is to provide a platform for users to learn Go over the board. The way our project accomplishes this task is to provide users an easy way of recording and viewing their previous Go games and to provide suggested moves from our Go engine. Both of these features serve to help users learn and devise their own strategies to improve upon their gameplay. These two features are provided via an integrated system of a physical Go board whose state is read in by light sensors and an Arduino microcontroller, and a software system on a computer which takes in the gameplay data and visualizes and saves gameplay. The software system also will run a Go engine in the background which takes in game states and outputs what it thinks is the top 5 best moves. The top rated move can then be sent back to the Arduino microcontroller which will light up 2 LEDs on the board, displaying to the users the recommended move.

11 Glossary of Acronyms

- CNN - Convolutional Neural Network
- MCTS - Monte Carlo Tree Search
- RL - Reinforcement Learning
- LED- Light Emiting Diode

12 References

1. Silver, Hubert, et. al., *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*, 5 Dec. 2017

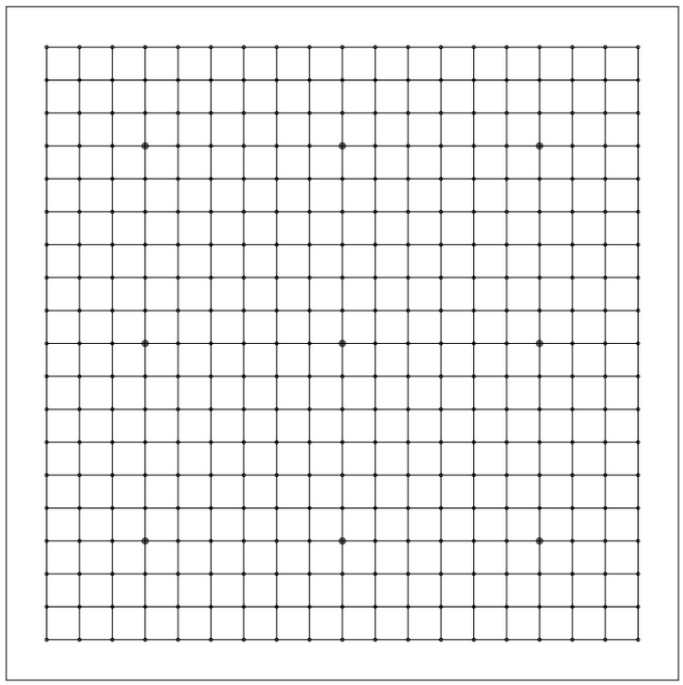
2. Schrittwieser, Antonoglou, et. al., *Mastering Atari, Go, Chess and Shogi by Planning with aLearned Model*, 21 Feb. 2020

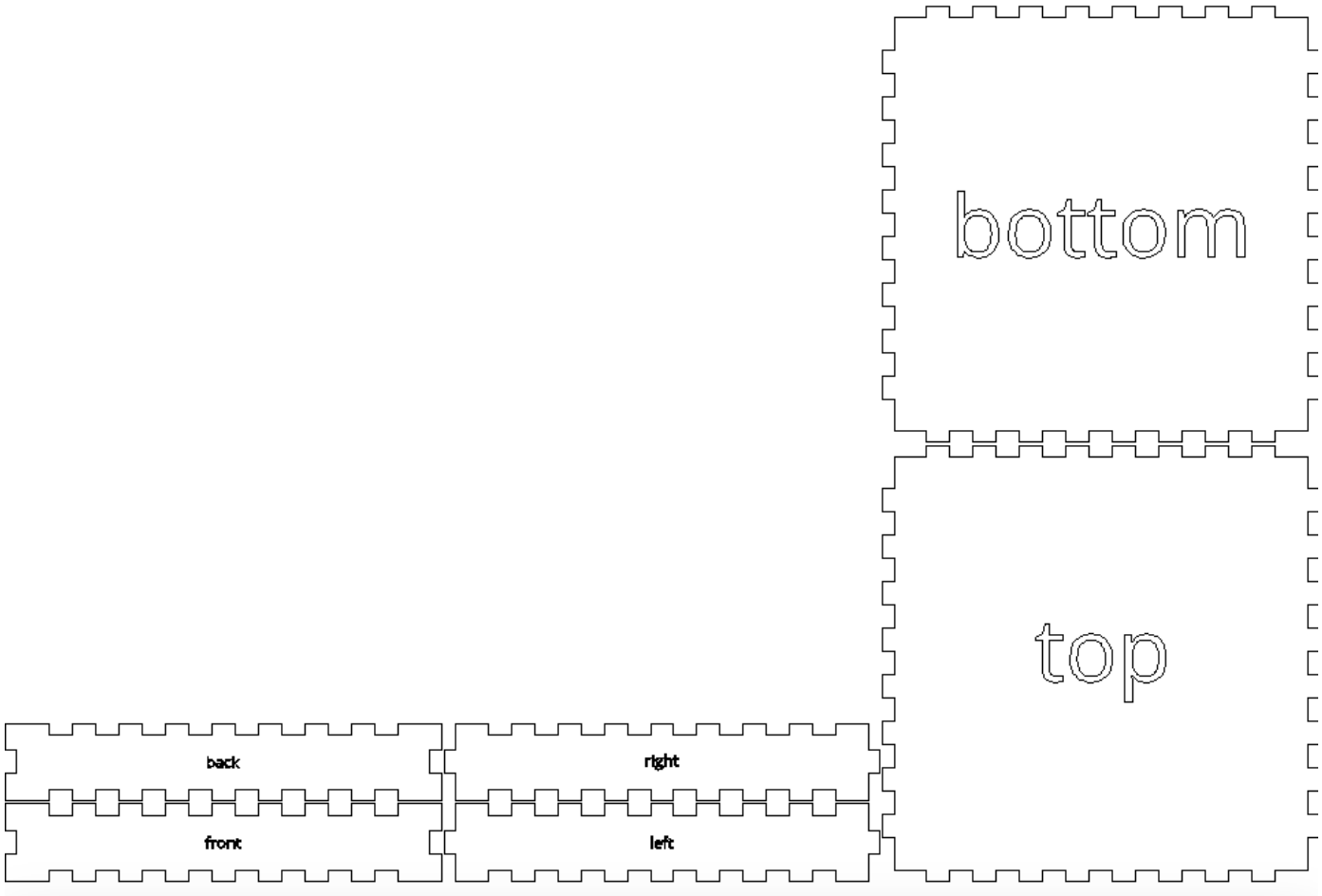
3. Hui, Jonathan, *Monte Carlo Tree Search (MCTS) in AlphaGo Zero*, 20 May 2018

4. Fragkiadaki, Katerina, *Deep Reinforcement Learning and Control: AlphaGo, AlphaGoZero, MuZero*, Spring 2022

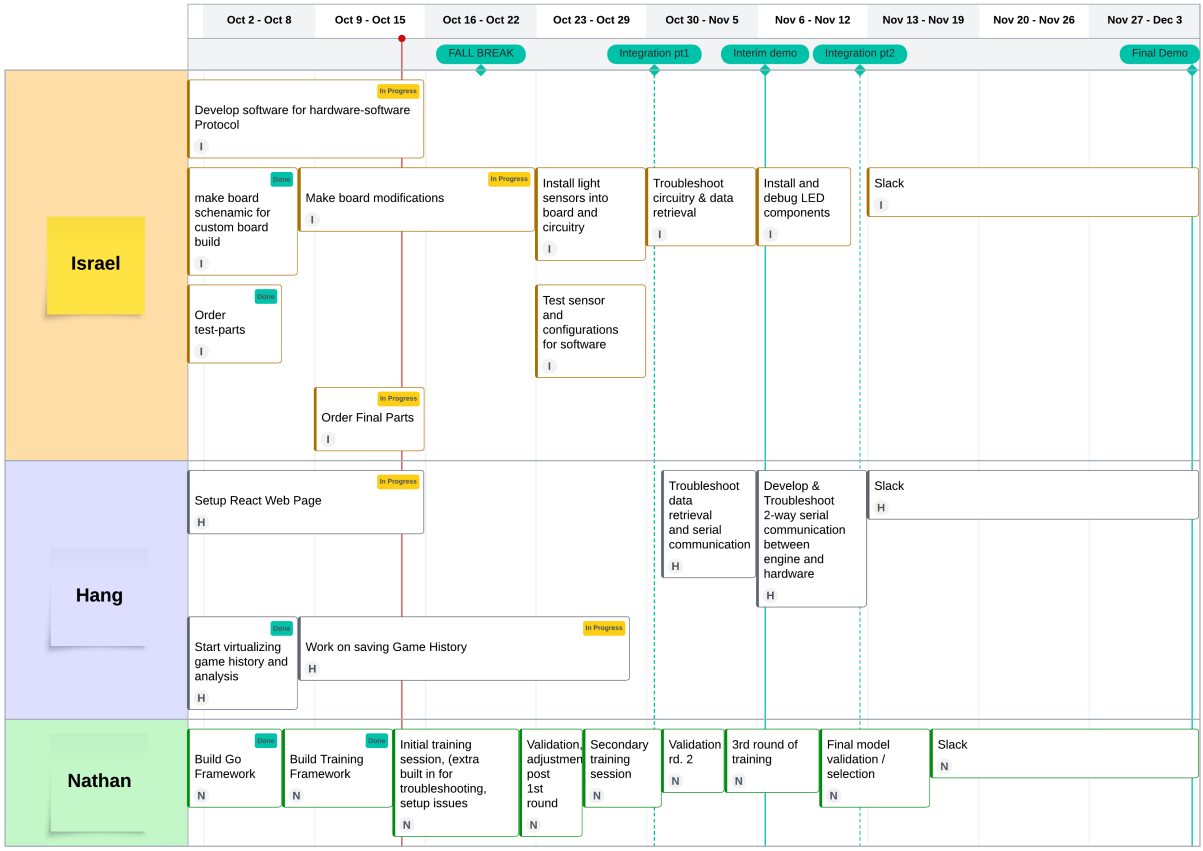
12.1 Architecture and system description figures (Ref 3)

12.1.1 Physical board implementation





12.2 Milestone and Schedule chart (Ref 8.1)



12.3 Budget and Parts list (Ref 8.3)

.

Parts	Model No.	Manafacturer	Amount	Cost	Description
16:1 analog 2pack	CD74HC4067	Ximimark	2	$ 11.98	16 to 1 muliplexer for digital and analog inputs/outputs.
16:1 analog 5pack	CD74HC4067	Ximimark	4	$ 27.96	
Light Sensor: QSE257	QSE257	onsemi	361	$ 327.46	Analog light sensor
ARDUINO MEGA 2560 REV3	A000067	Arduino	1	$ 48.99	Microcontroller purpose
8x 2:1 analog/digital	SN74LS158N	Texas Instruments	2	$ 1.80	A 2 to 1 muliplexer for digital and analog inputs/outputs. Each chip has 8 mulitplexers.
10-piece vector boards 70mmx90mm	43237-2	GENNEL	4	$ 39.80	PCB Board purposed for wiring and circuitry designing
Button	D6R90 F1 LFS	C&K	5	$ 5.60	Push toggle button to user-board interface
LED	754-1732-ND	KingBright	38	$ -	(recieve from ECE labs) LED indicators
resistors (100k-1k)	BJ-1OHM-1M OHM	BOJACK	38	$ -	(recieve from ECE labs) Resitors for regulating current through LEDs
Wood 100cm x 100cm	-	TechSpark	1		
Total cost				$ 463.59	