Go Learning Buddy

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

***Abstract*—The Go Learning Buddy provides a 9x9 physical board for beginning Go players to play on. It can indicate the best move for the current position, and provides a platform for post-game analysis. These features help players learn and develop strategies that they can extend to larger boards after they master our product. The physical Go board reads in the game state and sends the data to a server 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 (19x19) as well as the ability to capture and replace stones, the amount of possible positions is incomprehensibly large. As such, many beginners start on smaller boards (mainly 9x9 and 13x13) to get a grasp of the game fundamentals before moving up to the standard size board. Our product is designed with these beginners in mind.

Our product was originally designed with a 19x19 board, and was aimed towards medium-to-advanced amateur players. However, the construction and wiring of the physical board took far longer than we expected, and we switched to a 9x9 board accordingly. As a consequence, our target audience shifted towards beginners, as basically only beginners use 9x9 boards to play regularly. Additionally, because we had a functioning web app and engine, we built in the capability of being able to play against the engine online without using the physical board.

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 even on a 9x9 board there are 81 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. Finally, players may not have an opponent nearby of appropriate level to play against.

Our product solves all of these problems at once. Two players can play against each other on our custom-built Go board and each move is registered by photoelectric sensors on each intersection. These moves are transmitted to our software web application, where a Go engine trained with self-play reinforcement learning uses Monte Carlo Tree Search (MCTS) to calculate the best responses. These candidate moves are sent back to the web app for display, which sends the best of them (as judged by the engine) to the physical board, where LEDs light up the proper row and column. After each move is made, it is 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. Finally, if the user is on their own, they can play against the engine via the web app, without needing to use the physical board at all. In total, our product supports three modes of usage: human vs. human with hints, human vs. human without hints, and human vs. engine, all with available historical analysis.

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 stones on the board with 100% certainty. This is so that users can be sure historical data is correct when trying to learn from their past games. The engine also requires a completely accurate representation of the board state in order to give viable suggestions. The board must also rapidly assess the entire game state, accomplishing this task in under 50 microseconds so that the delay is imperceptible to our users.

Originally, we aimed for the self-play reinforcement learning engine to be programmed to operate at a skill level similar to that of an amateur 5-dan Go player. However, with the shift to a 9x9 board aimed towards beginners the skill requirement has also shifted. In games such as Go and Chess the most important factor when learning is playing against opponents "better" than yourself. As such, our

9x9 engine must be able to regularly outperform beginning players, quantified as at least a nine-to-one win-to-loss ratio, as this will ensure our users can improve enough from it to progress to larger boards. It will present the five most promising moves for any given position on our web application. This allows users to explore multiple possible lines of play, rather than the singular move the engine deems "best", allowing for deeper positional understanding. 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 top-rated move among these will be chosen and relayed as the recommended move to be displayed on the physical Go board. The calculation required to find the 5 optimal moves should be done in less than 3 seconds to make the game play over our board feel as responsive as it can be, given the computationally-heavy calculation of this step.

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. These combine to make sure that our users are not only learning from the actual games that they played, but that their games are not interrupted by needing to wait for the web app to catch up.

3 ARCHITECTURE AND/OR PRINCIPLE OF OPERATION

As can be seen in Fig. 1, users play Go on our physical board, which reads in the game state through photoelectric sensors (81 in total on a 9x9 board) underneath each of its row-column intersection. Our Arduino microcontroller captures this game state, which is then sent to the Raspberry Pi's serial communication port and then onwards to the web application through a websocket. Once the web application receives a message from the websocket, the web application then decodes the game state from the message and sends the game state to the Go engine, making an HTTP request to the back-end server hosting the Go engine to do so. The Go engine outputs the 5 best moves move the position, in order of strength, which are sent back to the web application as a response. Once the web application receives the response, it sends the recommended game move back to the Raspberry Pi through the same websocket connection, and then the server on the Raspberry Pi places said move on the serial communication port, which is then read in by the Arduino. If the users clicks the "Advice" button, the 2 LEDs corresponding to row and column of said move 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 conveys this 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. This allows users to analyze games they've played in the past, without creating a large storage overhead on the web app. Our overall system design can be seen on the following page in Figure 1, and is shown fully connected in Figure 14.

4 DESIGN REQUIREMENTS

4.1 Hardware

For our hardware design requirements, we largely focus on the performance and latency of our product, in order to fit the needs of our users, beginning Go players. This means accurately reporting the board state in under 50 microseconds (as detailed in 2). Because our routine for light sensor calibration is very similar, we require that our calibration stage also takes under 50 microseconds. In terms of sending communication to the COM port, we require a maximum latency of 120 microseconds. Finally, when sending an advice request or receiving advice, we require a maximum latency of 300 microseconds. These are all designed to make the hardware latency imperceptible to our users.

4.2 Software

Since users directly interact with the web application, the design requirements for this section relate to latency, making sure the site feels responsive to our users. We need the latency of the communication between the server on the RPi and the web application to be less than .5 seconds, so that when a user presses a button on the physical board to update the game state, the web application updates almost immediately, showing the new game state to our users. We also require that the response time for the web application making a recommended move request to our back-end server to be less than 3 seconds. We chose 3 seconds because we need to give sometime for the engine to run its computation, but not too long such that the gameplay feels sluggish. A design requirement that doesn't relate to latency is saving and retrieving past Go games with 100% accuracy, so that when users use our game history visualization feature, it isn't showing them game states that they haven't played.

4.3 RL Engine

Quantitative design requirements are especially important for the RL Engine component, as it contributes most of latency that the user will experience, in addition to directly controlling the quality of suggestions the user receives. In our use-case requirements we have stated a 3s limit between move input, and recommendation, as this is short enough

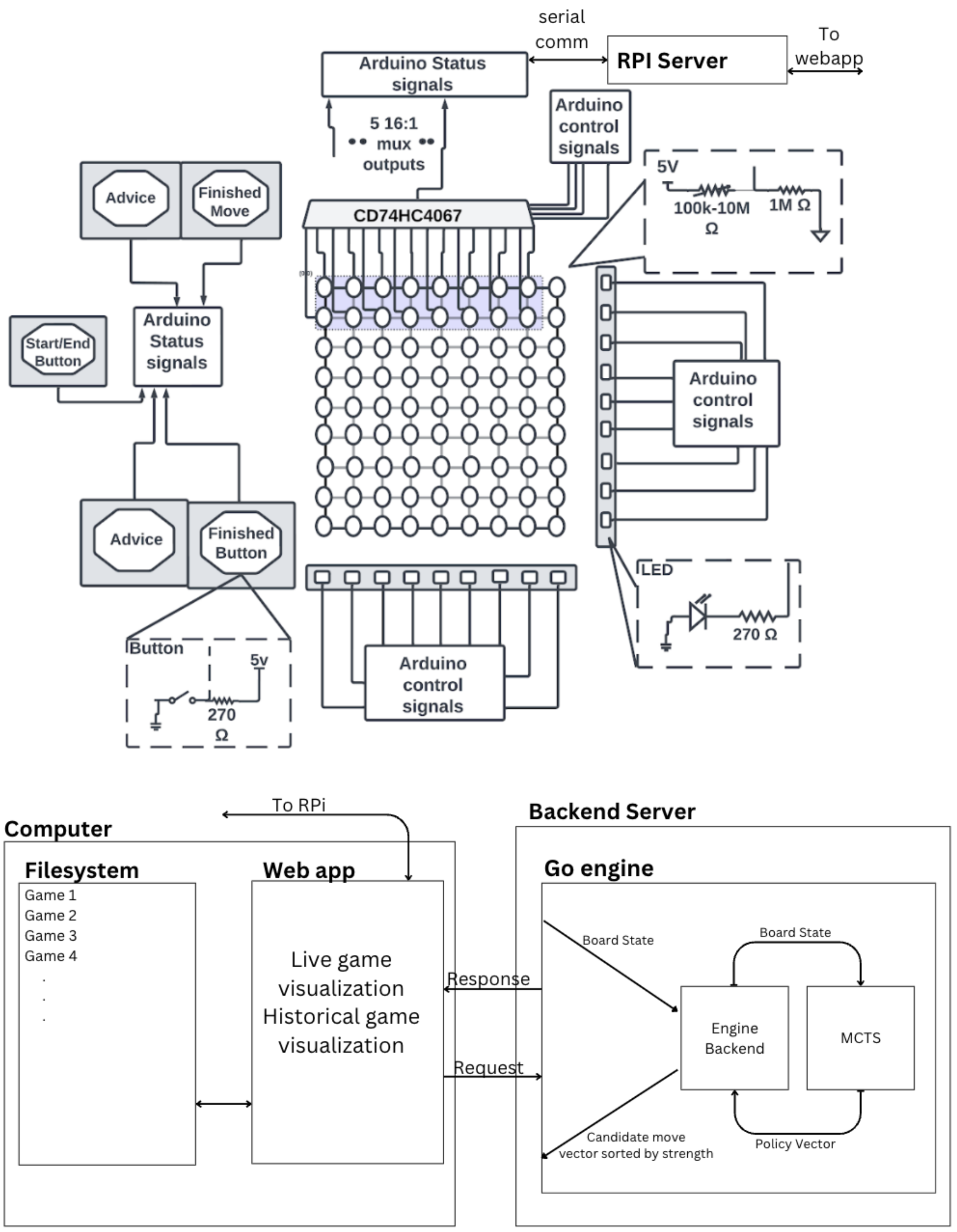


Figure 1: System Architecture

to not hinder gameplay, but long enough to allow for a proper MCTS simulation. As such, our MCTS must take no longer than 2.5s, as if it does, even if all other component meet requirements, the total latency will be above 3 seconds.

5 DESIGN TRADE STUDIES

5.1 Hardware

Whenever users make a move, we need to record how it has changed the board. This means that we have 81 sensors to gather data on and send to the COM port. Our micro controller can only support 16 analog input ports but with our 16:1 multiplexers, we are using only 5 of the ports. This means that in order for us to read each port, we have to poll 5 sensors for a given 16 cycles. This polling means that we need to select different sections, which requires binary conversion of the cycle for multiplexer selection inputting.

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

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

Depending on the embedded software coding for the micro-controller, 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 means 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 (16), t_e is time of execution, t_{delay} is the button time delay, and t_{total} is the total data retrieval time (50 microseconds).

There was the choice of either using light sensors or magnets to detect the board state. With magnets, we have the benefit of not needing to adjust the light sensors to the ambient lighting, and we would not have the issue of a small shadow disrupting our reading of the board state. However, magnets are costly, and when we were making the design for the original 19x19 board, the cost was the main reason why we decided on light sensors.

5.2 RL Engine

As mentioned in section IV, the entire MCTS process must take under 2.5s in order for our analysis to meet timing requirements. However, the simulation parameters (depth and amount of candidate moves considered) can always be adjusted in order to adhere to this. If we let d be the depth of simulation, f be the expansion factor of the MCTS search, x be the time required for a single positional evaluation from the value network (see 6.2.2), and y be the time required for a single evaluation from the policy network (see 6.2.1), through experimental observation, we have determined the time per simulation t_s corresponds roughly as follows.

$$t_s = dy * ((2d + f)/2f) + dfx + .008 \tag{4}$$

This is because for each simulation iteration (i.e. d times) f new board states must be evaluated, on average $f/2$ board states on the same level are explored before moving one layer deeper, and the average overhead is around 8 milliseconds.

Of course, the more computationally dense a neural network is, the more accurate it is likely to be, but also the longer a single evaluation by it will take. As such, we worked hard to try to find an architecture with a balance of computation time and accuracy, as a lower computation times allows for larger expansion and deeper simulation as shown above. The accuracies and latencies of the different value network configurations we tested are shown below. (Those of the policy network are not shown for two reasons: we tied the architecture of the policy network to that of the value network, and as can be seen above, the value network evaluation time has a larger effect on the total latency).

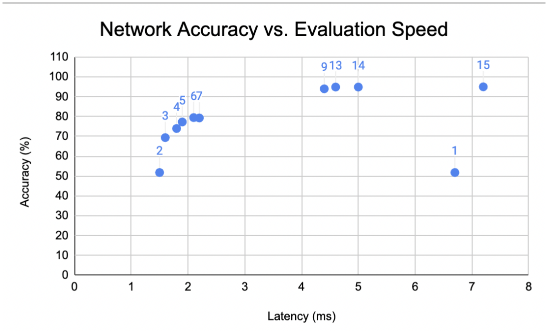


Figure 2: Network Size Trade-Off

It is important to note here that the value network's objective is to output a scalar evaluation of a position (0 for a black win, 1 for a white win), so accuracy in this case is measured by if the network correctly predicts which player won the game a specific board state is from. (16 million board states from expert-level and pro Go matches were used for training, and about 2 million were used for validation.)

Accordingly, we used the architecture from iteration 13, as while it had slightly lower accuracy than 14 and 15, that loss is on the order of .05% accuracy, and the reduced latency allows for deeper and wider simulations.

A second trade-off occurred when our group made the decision to switch from a 19x19 board to a 9x9. We had allocated three weeks for engine training, and by this point, two of them had already passed, meaning the engine was already specialized to work on a 19x19 board. As such, we had to decide whether to continue training the engine on a 19x19 board and just pad the input from a 9x9 when using it, or to stop 19x19 training and focus only on the 9x9. We chose the latter, as the former would have resulted in an almost unusable engine for our physical board and we wanted to prioritize our main, over-the-board, use cases (players would have been able to play against the 19x19 engine on the web app on the proper-sized board). However, this created some issues, as some results of the 19x19 training can still be seen in the engine, as we only had one week to tune them out (more in 7.2).

5.3 Software

An important design trade off that we considered when building the software component of our project was whether to run the engine computation client-side or server-side (server-side in our case, would be a Flask server running on Hang's computer). There were 2 main considerations when deciding on this choice: computation time and network latency.

Regarding network latency, the main benefit of running having the game engine compute client side would be that there would be no need to communicate over the network since the engine code would be embedded into the web application code. This means there would be no added latency to the game engine computation time. Having a separate server hosting the game engine would require our web application to make a request to the server's endpoint, and wait for a response, adding some latency to the game engine computation time due to the communication over the network.

For computation time, a benefit of computing the engine recommendations on a separate back-end server is that the computation time would generally be consistent, and therefore, more testable and easier to adjust on our end. Having this compute run client side would mean that the computation depended on the hardware that the client is running on (phone vs computer, older computer hardware vs newer computer hardware). Another major downside of running engine client side is that we would have to convert the TensorFlow code written in Python into TensorFlow for Javascript (tf.js) which can increase the computation time up to 10-15 times[5].

Considering that converting our engine code into Javascript with tf.js may increase our computation time by up to 10-15 times, we decided to have a separate back-end server hosting the game engine. Even though there will be added latency due to the communication happening over the network, this added latency is small compared the performance difference between running the engine code in Python (server side) vs Javascript (client side), considering an overwhelming majority of the engine's latency is caused by TensorFlow operations.

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, needed to use an entirely custom hollow board to support our electronics. As it is now 9x9, this board has a smaller lenght and width than a conventional GO board measuring at 273 mm by 273 mm, with the addition more of height so our Arduino fit inside at 73mm.

For this to be possible, we take advantage of laser cutters to cut out wood panels and make holes for our light sensors to detect where pieces are placed, using a DXF design file shown. At the same time, we developed finger joints for easy board assembly.

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 multiplexers that will connect to 16 light sensors as shown in Figure 1. We are using light sensors since we need a large amount of sensors (81) and light sensors are cheap compared to the magnet alternative we have discussed.

Now that all the light sensors have been designated assigned to a 16:1 multiplexer out of the 5 we will be using. In our initial plans for our 19x19 board, we planned on using additional 2:1 multiplexers to make up for the restriction of ports. But, due to the change to a 9x9 board, such restriction are not applicable as we would have more than enough ports for our implementation. This is our designation for the multiplexers:

- Arduino analog 0-4 port
  - 16:1 multiplexer components 0-4
    - light sensors 0-80
- Arduino analog 5 port
  - light sensors 81

We will use the Arduino's digital pins 0-3 for the 16:1 multiplexers selection port.

In addition to the data retrieval that we have designated and ported, we will have buttons (specified as hexagon blocks in our figure 1) 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/switches will be connected to digital pins D33-37.

We have also added 18 LEDs to signify the x and y coordinate of a preferred move. These LEDs will be connected in series to each digital Arduino pin D14-22 for Y coordinate and D5-13 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 summation, we have port designations as follows:

- analog 0-5 data retrieval sensors (inputs)
- digital 0-3 data control 16:1 mux selections (output)
- digital 14-22 Y coordinate LEDs (output)
- digital 5 -13 X coordinate LEDs (output)
- digital 33-37 Buttons (inputs)

For our data retrieval, we focus on grabbing one section of data in 16 cycles. Our software records 16 data values at a given cycle in a array. All our multiplexer selections are initialized at zero and our selection ports for our 16:1 increment from 0-15 in binary. Once we record data from 16 different addresses (totalling to 80 data points retrieved), our software fetches the 81st value in its own dedicated pin A5. Once we have completed retrieving all our data, our array is full of all 81 values and is ready to send to COMS port as text values.

For our user interface, we use an advice buttons that triggers the arduino to light up 2 LEDs (one row and one column). The micro controller knows which ones to turn on by sending a advice poll to the COMs port and waiting for advice coordinates to be received from the engine.

We also have buttons for when a turn is finished. This triggers a data retrieval action for the micro-controller. We have our start/end button trigger pre-configuration phases where we calibrate our light sensors to the ambient light of the room.

6.2 RL Engine

The Go engine built using self-play reinforcement learning was implemented in two parts: training and usage in analysis. Monte Carlo tree search[1] (MCTS) was used in both cases; for the former it decided what the next move was in the simulation and for the latter it decides the recommendation for a given position. MCTS utilizes two neural networks, known as the value and policy networks, to generate suggested moves for a position (see figure 3).

6.2.1 Policy Network

The policy network is a convolutional neural network (CNN). The input is a 9x9 vector, representing a Go board state, and the output is a length 82 vector (one for each of the 81 intersections in addition to output[81] representing passing the turn) 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 were determined by a Gaussian distribution with mean 0. For each training batch, the ground truth that the policy network is trying to match was the visit count from the simulation the board state was generated in (see 6.2.4).

6.2.2 Value Network

The value network is another CNN. The input is the same 9x9 vector representing the board state, and the output is a singular scalar value, representing the players chance of winning from that position (0 representing a black win, and 1 representing a white win). The initial weights for this network were trained via regression, using a data set of board-states pulled from expert level Go matches, tagged with the outcome of the match. This allowed the value network to obtain a general positional understanding before any of the MCTS simulations detailed in 6.2.4.

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: evaluation 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. (Referred to by these variables in Figure 3 as well.) 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 evaluation score encourages expanding branches of the tree with board states more beneficial for the player who is moving, and the exploration bonus encourages expanding sparser, more rarely visited areas of the tree. 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). Depth of simulation, (d), and expansion factor (f) are set, and then the following steps are repeated d times. (For engine usage these are set to take just under 2.5s per calculation as detailed in 5.2, but for training both d and f are varied to generate more diverse training data.)

1. Start at the root of the built-out tree (the current board state)

2. Until you are at a leaf node, move downwards to the child state with the highest score according to equation 5, marking each node you pass through as visited.

3. Take the top f moves suggested by the policy network in the leaf position, and create child nodes for each one.

4. Evaluate each child node with the value network.

5. Traverse back up to the route, updating the average values Q for each node based off of its new descendants, 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. That is, whichever child of the root has been visited the most in step 2 of the above list is designated as the move of choice by the engine. Each child's visits is divided by the total number of visits to form the target vector used to train he policy network for future iterations.

6.2.4 Training Implementation

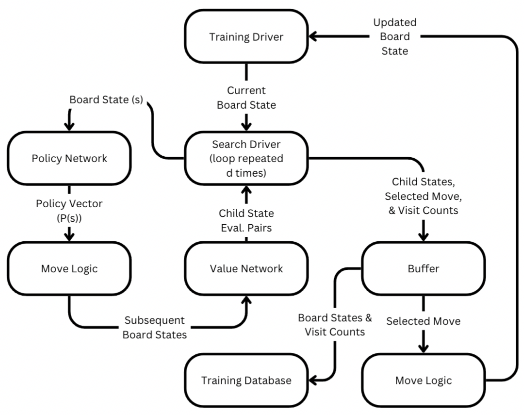


Figure 3: Training Flow

The training driver maintains the current board state which it sends to the search driver after every move. The search driver builds out the game tree through d iterations using the value and policy networks. The strongest available move is processed with move logic and the resulting state is sent back to the training driver. The visit count data is stored with the inputted board state as training data for the policy network, and after the simulated game runs to completion, all constituent 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). Essentially, it becomes an extension of the value network,

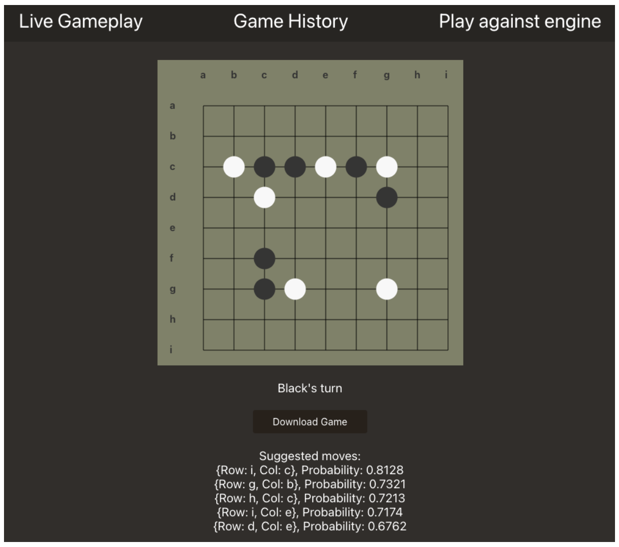
6.2.5 Analysis Usage

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 is built in React and visualizes the live gameplay of the Go board to allow users to analyze their saved Go games. The web application also allows users to play against the game engine.

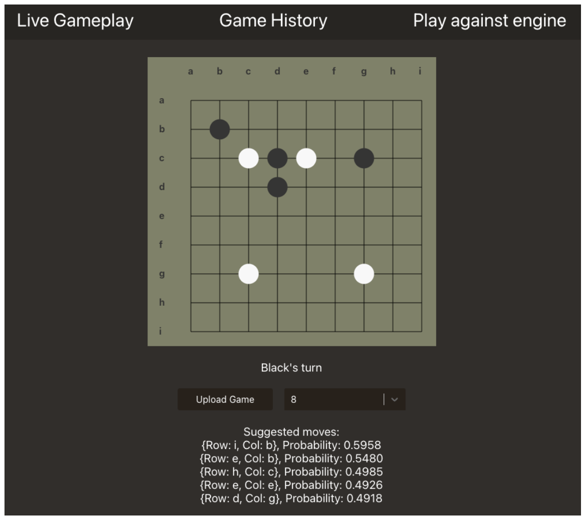
6.3.1 Web pages



Black's turn

Suggested moves:
{Row: i, Col: c}, Probability: 0.8128
{Row: g, Col: b}, Probability: 0.7321
{Row: h, Col: c}, Probability: 0.7213
{Row: i, Col: e}, Probability: 0.7174
{Row: d, Col: e}, Probability: 0.6762

Figure 4: Live gameplay page



Black's turn

Suggested moves:
{Row: i, Col: b}, Probability: 0.5958
{Row: e, Col: b}, Probability: 0.5480
{Row: h, Col: c}, Probability: 0.4985
{Row: e, Col: e}, Probability: 0.4926
{Row: d, Col: g}, Probability: 0.4918

Figure 5: Game history page

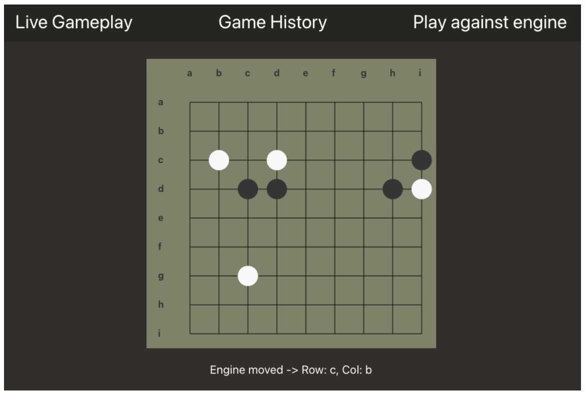


Figure 6: Play against the engine page

The first page (figure 3) is the page corresponding to the visualization of the live gameplay over the physical Go board. This page allows users to see the Go game on the computer, as well as the 5 moves suggested by the Go engine. The page also allows users to save the game by downloading a game information file to their computer's filesystem so that they can later use this file to visualize their saved gameplay.

The second page (figure 4) 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 also displays the suggestions made by the Go engine along with the engine's predicted probabilities of winning the game for each respective move.

The third and final page (figure 5) allows users to play directly against the engine without playing over the physical board. The probabilities shown in figure 3 and figure 4 describe the chance of either black or white winning the game if they make the recommended move.

6.3.2 Visualization of the Go board

The Go board is drawn with CSS and can be broken down into 2 parts: background and tiles. The background is simply 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 81. 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 2d array is updated and this is the array that will eventually be saved. This 2d array contains arrays that correspond to the board state at different moves (array at index 0 corresponds to the board state at move 0). When the game is over, this array is converted into a dictionary where the key is the move number and the value is the board state, written to a file, and then this file can be saved into your computer's file system. We chose to save to the user's file system rather than database because we don't want to require users to make an account to store their saved games as we want saved games to be private to each user.

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 Arduino

To interface the web application with the Arduino, we have a RPi connected to the Arduino via a USB. On the RPi runs a Flask server that on start up, starts a thread that continuously polls the serial communication port for data written by the Arduino whenever a board state is read in. Once a board state is read in, the server on the RPi emits a message to the web application through a websocket. There are 4 types of messages that the server on the RPi can emit to the web application: start game, get advice (recommended moves), update board state, and finish game. For communicating the recommended move back to the Arduino, the web application receives the best move from the engine, emits this data back to the server on the RPi via the same websocket, and then the server places the data on the serial communication port for the Arduino to read in the recommended move.

6.3.5 Interfacing with the game engine

The game engine runs on a Flask server hosted by a laptop. For the web application to communicate with the engine, the web application makes a request to the get_recommended_moves endpoint with the current board state as the payload. The engine receives this board state and runs inference to get the top 5 best moves and returns the best moves in a sorted array as a response.

7 TEST & VALIDATION

7.1 Tests for Hardware

One of the main focus 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 retrieval and configuration phase, we will run Arduino made function, millis(), to record the execution of our data retrieval actions and apply our previous equation in section 5.

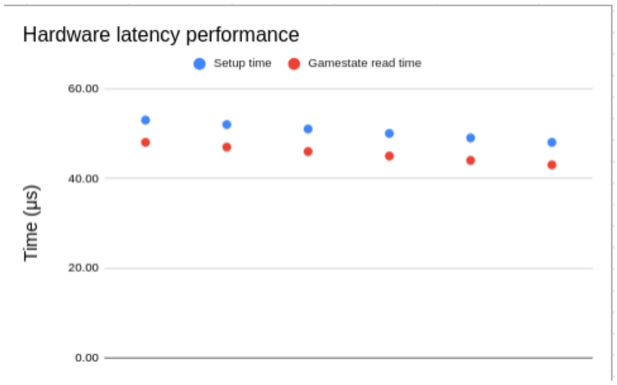


Figure 7: Hardware latency performance

As we can see, the performance of our game retrieval and game pre-configuration are fairly consistent where pre configuration takes slightly longer due to it doing the same process as game retrieval along with assigning all 81 values to be used in the future with a constant sensitivity margin filter applied.

As for accuracy of our hardware, we will put it through a series of 50-100 different games and make sure each game state is obtained and processed with no error. This would mean we have a accuracy of 100% for our implementation as this is expected performance for the average user.

We have initial phase of hardware testing accuracy for our resistor choices which includes a variety of different photo resistors in series with a 1MΩ or a 10MΩ.

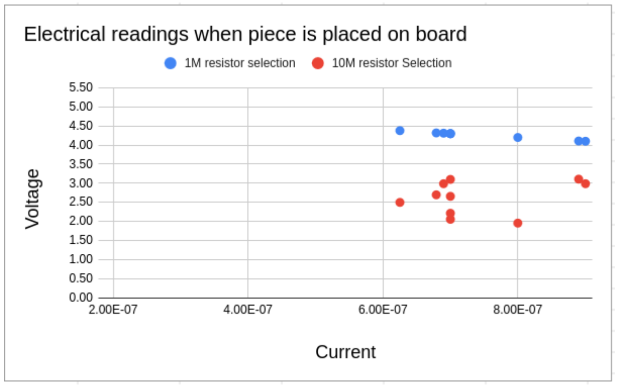


Figure 8: Electrical readings for occupied intersection

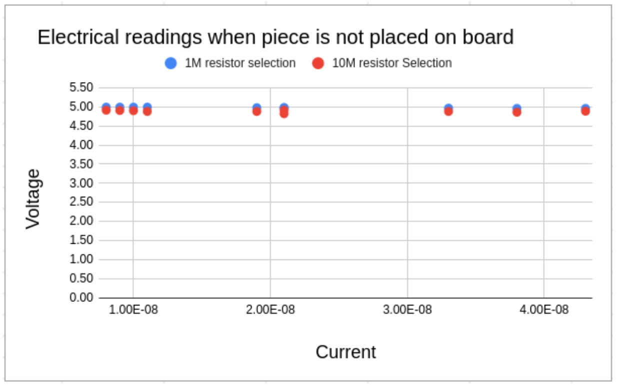


Figure 9: Electrical readings for open intersection

As we can see through these results, we can see a larger difference in voltages for our 10MΩ resistors compared to our 1MΩ resistor configuration. This larger difference when compared to placed and not placed pieces helps differentiate when pieces are placed or not. Therefore, the 10MΩ resistors are the what we used for our implemention to allow us to know when pieces are placed.

7.2 Tests for RL Engine

Accuracy testing: Originally, we were going to test our engine's accuracy by playing it manually against engines of known strength available online. As our target strength was that of amateur 5-dan, we had planned on playing our engine against an engine with that strength a representative amount of times ($\geq$20), and if our engine scored at least as many wins as losses, it could then me said with reasonable confidence that our engine was at at least that level. However, switching from a 19x19 board to a 9x9 board threw a wrench in these plans.

There are some engines available online to play against on 9x9 boards, however, they do not have a known strength. In fact, the standard ranking system of dan is for specifically a 19x19 board, so we now cannot apply it to our product. This is why we switched to the metric detailed in 2, which involved playing at least 20 games, and maintaining a win-to-loss ratio of at least nine-to-one. The reasoning behind this is that in order to improve, players need to play against and get suggestions from players (or in our case engines) better than them. As our main audience shifted to beginners, our main focus was to ensure our engine was better than them by enough for them to be able to learn from it. A nine-to-one ratio makes this difference in level clear.

Fortunately, all three of us were beginners, so we could test against the engine, but we also found roommates and friends to test our engine against as well. The smaller 9x9 board lends itself to draws, but of the 35 games played to a result (each of the seven testers played 5 games to a result) the engine won 32, resulting in a win-loss ratio of around 10.7-to-1. It is also interesting to note that all 3 of the wins came in the fourth and fifth games the testers were playing, suggesting that they were able to improve by playing against our engine even a small amount of times.

As additional metrics, we can look back to section 5.2 and Figure 2 to see that our value network accuracy clocked at just under 95%. This means that given any board position, our evaluation engine can predict the winner from that point correctly, 95% of the time. This is particularly impressive, as many of the states used to test this are from the beginnings of games, from before any meaningful mistake has been made. Considering even the strongest engine would only perform slightly better than 50% in these circumstances, an overall mark of 95% is quite strong. Additionally, our policy network scored an accuracy of 84% with a mean-squared error loss of $1.927 * 10^{-4}$, meaning that given a position, the strongest suggestion the policy network gives matches the target vector (as detailed in 6.2.1).

In all cases our expansion factor was at least 10, meaning the probability the strongest move was in the top 10 suggestions of the policy network is far greater than that recorded number of 84%.

This all notwithstanding, there were some issues identified during the player-vs-engine portion of the testing. The first was the engine's seeming aversion to captures. While the engine has a concept of captures in a general sense (i.e. it knows that board positions with more of their stones than their opponents are generally favorable) there is no explicit bonus programmed in for captures. This resulted in the engine sometimes not making captures that would have seemed obvious to a human player. Generally, once stones have the ability to be captured, this option does not go away, so it is possible the engine just ”sees” a more important move at the time and is ”planning” to make the capture later, but that is difficult to evaluate on its own. Secondarily, because the first two weeks of engine training was on a 19x19 board, it seems to ”hallucinate” occasionally, suggesting moves near the right and bottom edges of the board that do not make a huge amount of sense. This is likely because on a 19x19 board, moving towards those specific edges would expand a player's territory, and if we had had more than one week to tune for a 9x9 board, this might have also removed this issue, but as it was, it definitely cut into our engine's strength.

Finally, while our engine performed well against users who were complete beginners, it did struggle against two more intermediate players, suggesting it has quite bit of room for improvement. One of these players, who we met at the TechSpark demo, helped us identify the hallucination issue.

Latency testing: As shown in 5.2, we controlled the parameters of our MCTS to ensure our engine computation time remained underneath the 2.5s barrier. That being said, there is a difference between theory and practice. As is shown below, at all times our response time was below the 2.5s threshold, decreasing as the game goes on due to there being fewer possible moves that needed to be considered.

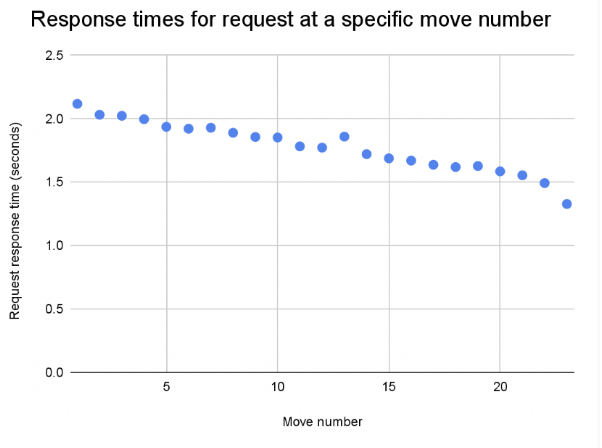


Figure 10: Average Engine Calculation Time

7.3 Tests for Software

For testing the latency for the communication between the server on the RPi and the web application, we set up a simple back and forth between the two by having the web application send the board state to the server on the RPi and the server immediately sending that same board state back. We recorded the timestamp of when the web application first emitted the message, and the timestamp of when the web application received a message through the websocket and subtract the two timestamps. We've collected 34 datapoints and the results are shown below:

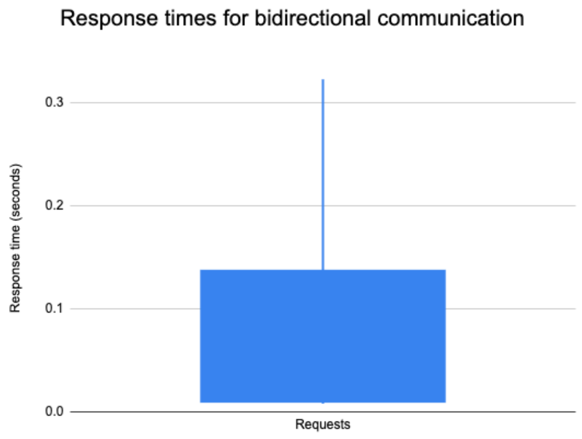


Figure 11: Response times for bidirectional communication graph

As we can see from figure 6, all of the recorded times fall below the .5 second mark, with the vast majority of them (75%) them falling below the .15 second mark, meeting our goal stated in the design requirements.

For testing the latency between the web application and the back-end server hosting the game engine, we recorded the timestamp right before the web application made a request to the server, and recorded the timestamp as soon as the web application received a response from the server and subtract the two timestamps. We've collected 30 datapoints and the results are shown below:

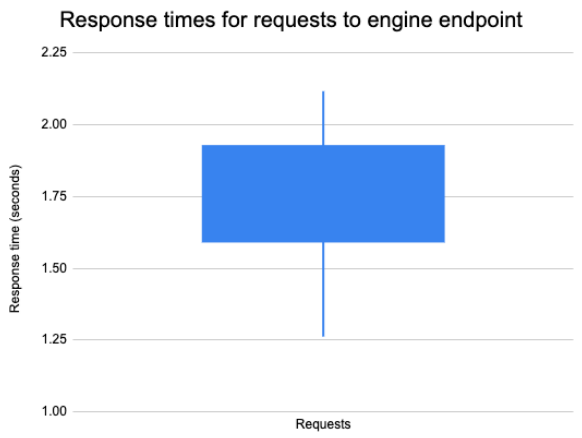


Figure 12: Response times for requests to engine endpoint graph

We see from figure 7 that all of the recorded times fall below the 3 second mark which meets our stated goal in the design requirements.

With both of the latency tests reaching our stated goals, the web application (and integration of the physical board, web application, and engine) feels responsive and doesn't distract user from playing the game.

For testing the accuracy of saving and displaying the Go board game states, jest unit testing was written in the web application code. These unit tests ensure that the files written match the specification given the state data in the web application. These unit tests also ensure that the rendering of the historical gameplay match specification.

8 PROJECT MANAGEMENT

8.1 Schedule

We organized our schedule by team member, the full schedule can be found in Fig. 14. Said schedule is quite different from our original, mainly because of the switch from a 19x19 board to a 9x9. As you can see, all three team members had added sections on converting their former work to fit a 9x9 board, or to rebuilding the physical board in Israel's case. Additionally, the switch from standard photo-electric sensors to photo resistors also caused a change you see reflected in Israel's schedule.

8.2 Team Member Responsibilities

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

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

Israel: Primary responsibility was 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 is also of great cost. Also, because of the many setbacks of our board development, there were purchases such as the acrylic boards that were not needed in the final product sadly. Such purchases being removed would've made our project way more cost efficient if planning went more according to plan.

The full list of our purchases and parts are found in Fig. 15.

8.4 Risk Management

8.4.1 Hardware

For our hardware implementation, we had many issues that could have occured with our system.

One of the risks we had was developing a board of our own for this project without the help of external personnel. Our planned mitigation was to shift to having no custom board but a board with holes. Even though this would have exposed our circuitry it would have provided the necessary functionality. Fortunately we were able to design and rebuild a 9x9 board ourselves.

In addition, because we shifted from PCB/Vector board design to wiring our circuitry, there was a high chance that some of these sensors would break due to miswiring or touching wires. To make sure we covered this, we did compression tests (fitting the wiring into the board), and marked all the sensors that stopped working. We then insulated all the wires related to said sensors, and retested until we caught all issues.

Another risk we considered were possible communications problems with the COMs port and other computers. If our Arduino did not communicate correctly with the computer, we would have pre-installed Operating systems on our computers to ensure that the Arduino has no problems what ever the software is used.

8.4.2 RL Engine

We foresaw three possible risks with the engine. The first two related to training time, the former being a situation where the training of the networks took a prohibitive amount of time, and the latter being a situation where the MCTS simulation itself was taking too long generating training data. After porting every required file to afs storage, network training did not end up being a bottleneck. In fact, we were more limited by the time it took the network to evaluate positions than we were by the time it took to train. However, the MCTS simulation did take quite a bit longer than expected. We managed it in two ways, the first being our planned solution of down-scaling the simulation depth and expansion factor. This harmed the engines strength, but only marginally. Due to the previously mentioned exploitation weighting (see equation 5 6), the first steps taken in the tree search matter much more than the later steps. Reducing the depth does not affect the engine's ability to select this first move accurately when building the game tree.

The third, more damaging issue would have been if the engine didn't play at a high enough level for player improvement, regardless of cause. In this case, we planned to use an open-source Go engine as a replacement. Fortunately, due to our shift towards targeting beginners, this was not required, though it might have been had we remained with a 19x19 board targeting higher level players.

8.4.3 Software

The main risk that can occur for the software component is any last minute overhauls of the designs, which ended up happening when we changed our design from a 19x19 board to a 9x9 board. To mitigate this risk, the software component is designed to be as modular as possible, making the switch from the 19x19 to 9x9 almost seamless since the rendering of the Go board is parameterized.

9 Ethical Issues

Our worst-case scenario is for a user to accidentally misuse or break our product, exposing dangerous materials (lead & electronics). These dangerous materials can harm the user, either through lead contamination or mild shock. As such, the worst way someone can use our product is to either treat it roughly or not dispose of it properly. Slamming the hinges to the board or hitting the board itself can cause the hinges to break, exposing the underside where the electronics are located, and lead is used to solder. Touching the exposed electronics can cause shocks, and if pieces of lead break off, or a user touches it and then their mouth, small amounts of lead can be consumed. Additionally, if our product is just thrown in the general trash and not disposed of properly, the lead can leak into the environment.

A scenario that can lead to this worst case involves a user treating the product in a very rough or careless manner. A user may store the product in an unsafe manner; for example, they may place the product on the edge of a table and accidentally push the board off, causing the hinges to break, exposing the underside where all the lead and electronics are. In this case, this is disjoint from our intended use as it does not connect to the usage of our tools itself but how to properly handle them. As such, we must make sure to give as explicit as possible directions for how our product is to be handled, used, stored, and disposed of.

The users can get harmed if they come into contact with exposed electronics or lead. The environment can get harmed through improper disposal of the product.

Children are particularly vulnerable to this worst-case scenario. They have worse emotion control than adults and are more likely to throw the product off a table in anger or treat it roughly in some other way. They are also worse at following directions, or at least understand the value of doing so less, so they are more likely to disregard storage or disposal instructions.

The main ethical concept being violated here is accountability. This worst-case scenario revolves around the misuse of our product in some way. Of course, we cannot fully control how our product is treated, but we must do our utmost to prevent misuse and mitigate the circumstances leading to the physical damage of the board. For example, we can't make our product indestructible, but we can reinforce the hinges to reduce the chance of them breaking. We can't guarantee our users will store or dispose of the product correctly, but we can include detailed instructions, warnings for what could go wrong, and possibly mandate that a parent be there for the product's purchase to ensure that at least one responsible adult knows the risks (in the hopes that they can guide the user if it is not themselves to safety).

10 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 similar to what our engine uses (MCTS with CNNs). 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.

11 SUMMARY

The goal of our project, Go Learning Buddy, was to provide a platform for beginners to learn Go over the board. Our project accomplished this task by providing users an easy way of recording and viewing their previous Go games and suggesting moves via our Go engine. Both of these features serve to help users learn and devise their own strategies to improve 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 micro-controller, and a software system on a computer which takes in the gameplay data and visualizes and saves gameplay. The software system also runs 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 is then sent back to the Arduino micro-controller which lights up 2 LEDs on the board, displaying to the users the recommended move.

With regards to our requirements, we met all of them (aside from the original engine strength requirement that shifted). The board records the game state with 100% accuracy, all of our latencies are below their given thresholds, and the engine has a greater than 9-to-1 win-to-loss ratio against beginner opponents.

11.1 Future work

One potential item we can add to our project is to convert the 9x9 board into the bigger 19x19 board, but with a custom PCB instead of hand-wiring the components together. The PCB can streamline the building process as many of the build issues we ran into for this project was with wires touching when they should not have. This would make the game more applicable to a professional audience, as they may want to save their games automatically.

On the software side, a database can be added such that players have the option of making an account and saving their games to the database, or of simply using the current feature which is downloading the game onto their computer's file system.

On the engine side, if we continued with a 9x9 we would want to more fully train on a 9x9, removing the hallucination issue detailed in 7.2. However, if it were shifted to a 19x19 as suggested just above, we would want to revert to the original 19x19 training, and continue from there, as we never had a fully functional 19x19 policy network. Added training serves only to increase the quality of the engine, which would be vital as with a 19x19 board we would not only be catering to beginners, but also to stronger players.

11.2 Lessons Learned

When developing the board and hardware implementation, there were many setbacks and design changes to the board development. Many parts came in late, and the build processes for each component took much longer than expected. As such, we learned that it was necessary to plan for the build processes of each hardware component to take longer than we expect, and to have contingencies in case we are not able to finish certain components in time.

This lesson leads directly into the next lesson we have learned: to make our designs as modular as possible to account for any possible contingency plan we may take instead of the original plan. Because the software and engine were designed to be extremely modular, we were able to seamlessly transition the board from a 19x19 grid into a 9x9 grid, even with the extremely tight deadline we were on.

12 Glossary of Acronyms

- CNN - Convolutional Neural Network
- LED - Light Emitting Diode
- MCTS - Monte Carlo Tree Search
- PCB - Printable Circuit Board
- RL - Reinforcement Learning
- RPi – Raspberry Pi

13 References

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

2. "Get Started with TensorFlow.Js." TensorFlow, www.tensorflow.org/js/tutorials. Accessed 15 Dec. 2023.

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

4. "MakerCase" Makercase, https://en.makercase.com//. Accessed 9 Nov. 2023

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

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

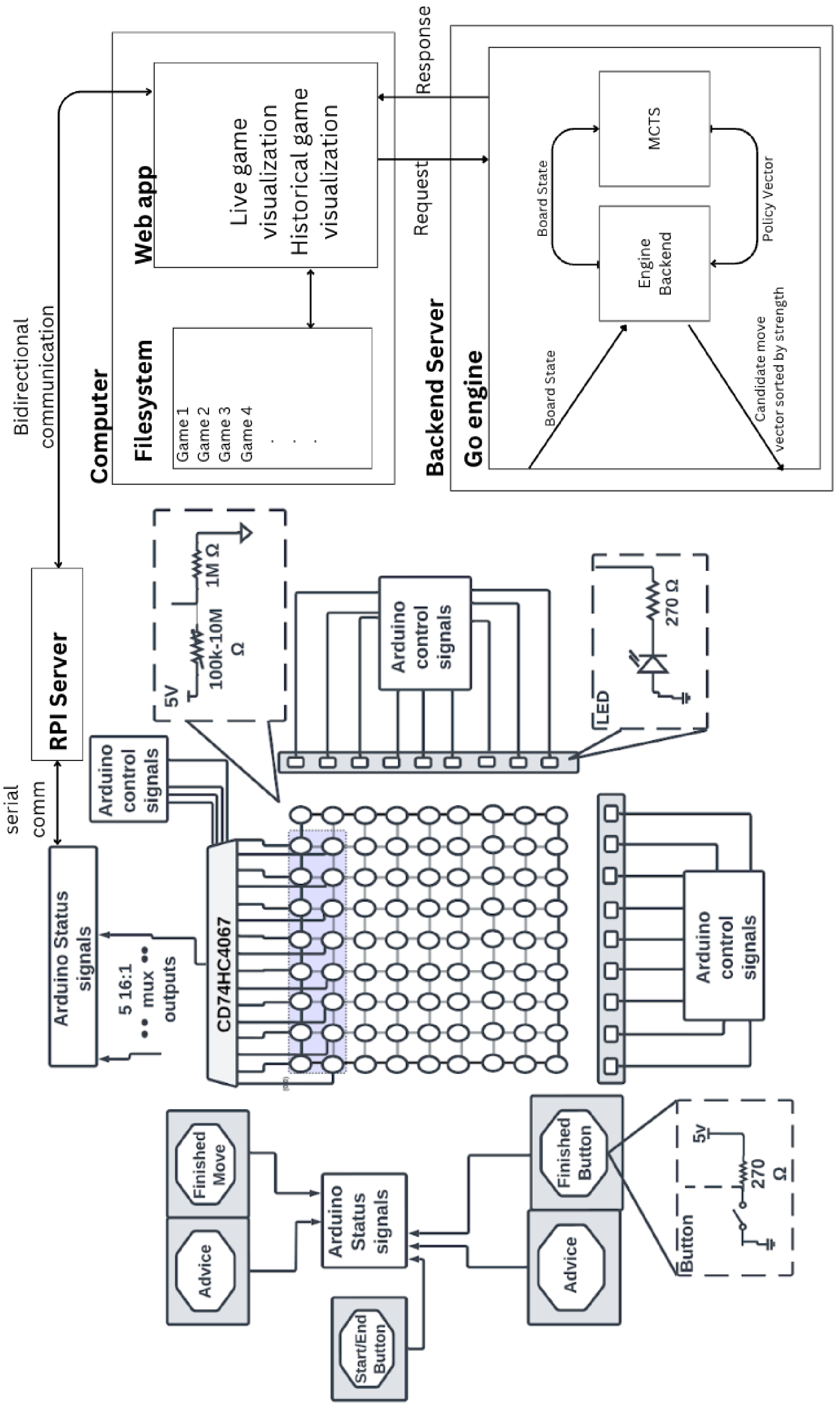


Figure 13: Fully connected system architecture diagram

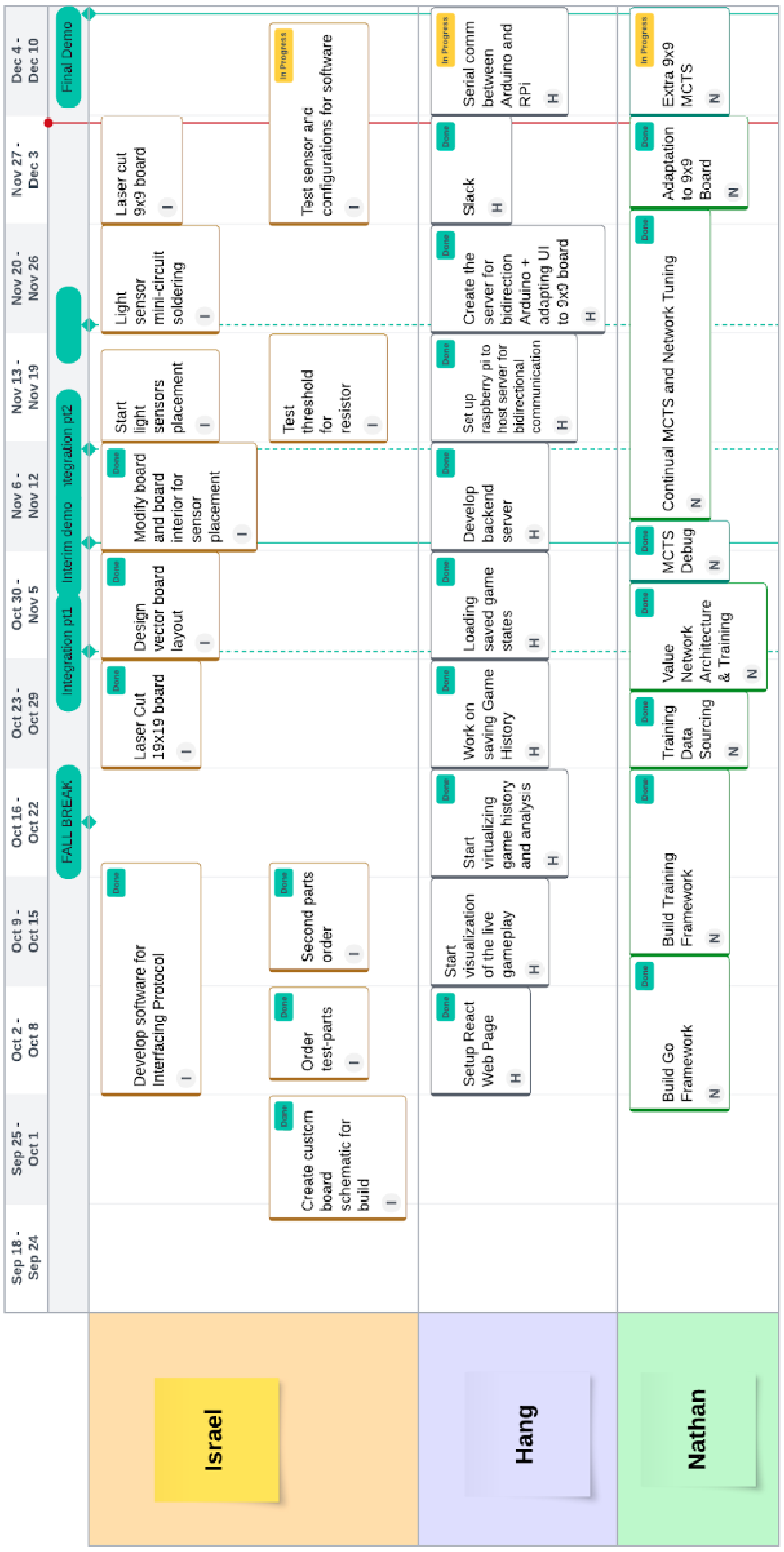
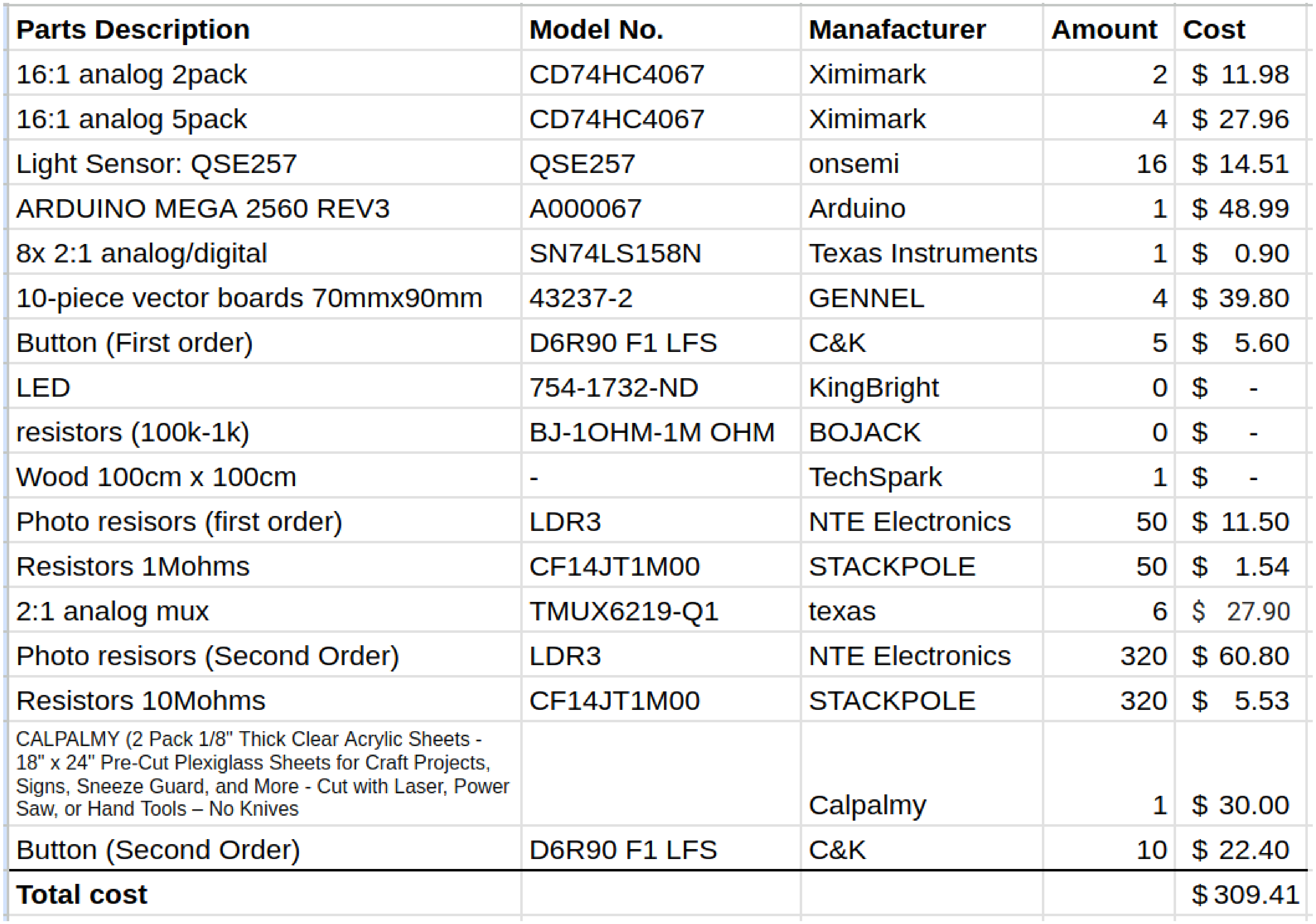


Figure 14: Gantt Chart



Parts Description	Model No.	Manafacturer	Amount	Cost
16:1 analog 2pack	CD74HC4067	Ximimark	2	$ 11.98
16:1 analog 5pack	CD74HC4067	Ximimark	4	$ 27.96
Light Sensor: QSE257	QSE257	onsemi	16	$ 14.51
ARDUINO MEGA 2560 REV3	A000067	Arduino	1	$ 48.99
8x 2:1 analog/digital	SN74LS158N	Texas Instruments	1	$ 0.90
10-piece vector boards 70mmx90mm	43237-2	GENNEL	4	$ 39.80
Button (First order)	D6R90 F1 LFS	C&K	5	$ 5.60
LED	754-1732-ND	KingBright	0	$ -
resistors (100k-1k)	BJ-1OHM-1M OHM	BOJACK	0	$ -
Wood 100cm x 100cm	-	TechSpark	1	$ -
Photo resisors (first order)	LDR3	NTE Electronics	50	$ 11.50
Resistors 1Mohms	CF14JT1M00	STACKPOLE	50	$ 1.54
2:1 analog mux	TMUX6219-Q1	texas	6	$ 27.90
Photo resisors (Second Order)	LDR3	NTE Electronics	320	$ 60.80
Resistors 10Mohms	CF14JT1M00	STACKPOLE	320	$ 5.53
CALPALMY (2 Pack 1/8" Thick Clear Acrylic Sheets - 18" x 24" Pre-Cut Plexiglass Sheets for Craft Projects, Signs, Sneeze Guard, and More - Cut with Laser, Power Saw, or Hand Tools – No Knives		Calpalmy	1	$ 30.00
Button (Second Order)	D6R90 F1 LFS	C&K	10	$ 22.40
Total cost				$ 309.41

Figure 15: Bills