Exploring Neural Networks and Machine Learning with the Game of Connect Four

Ian Priddle

Carleton University

Supervised by Dr. Yuly Billig

Abstract

In order to explore strategies in machine learning for potential use in the exploration of mathematics, architecture was constructed to allow several implementations of neural networks to learn the game of Connect 4 using a self-play-inspired process [1]. The Python libraries Numpy, TensorFlow, and Matplotlib were utilized throughout, and the Connect 4 game boards were stored in a fashion inspired by the Bitboard format [4, 2, 7, 6]. A Monte Carlo approach was used to generate the suggested moves for the network, used as training data for the network. A number of implementations were used in training, including a shallow network and a two-hidden-layer network making use of transfer learning and the results obtained through the shallow network [9]. The resulting network was able to learn some basic behaviours in the game of Connect 4, including prioritizing wins in the near future and avoiding placing pieces in columns which have already been filled, but it failed to fully prioritize other important behaviours like blocking its opponent from winning or recognizing when its attempt to win has been blocked.

1 Purpose and goals

This project was begun with the intent of exploring neural networks and the training thereof, with the hope to apply this knowledge to future problems with greater scope and ambition. An ultimate goal was to make use of machine learning to allow a computer program to independently explore and discover mathematical concepts such as axioms and theorems with only minimal interaction from a person. This project would require understanding of machine learning processes, as well as the creation of tools that would allow the program to evaluate its own discoveries without requiring a person to assist it. In order to prepare for this, the game of Connect Four was selected for the initial project.

The intent of the Connect Four Project, as it shall henceforth be known, was to explore the training of a neural network on a somewhat complex problem, beginning with a shallow architecture and gradually expanding into a deeper architecture with dynamic elements. To ensure that the program learns independently, the rules of the game were not to be shared with it, nor did it have access to any games of Connect Four played by one or more humans. In other words, the program was to play the game entirely against itself, and learn the rules based on the outcome of the game. This technique, known as self-play, took some inspiration from the *AlphaGo Zero* project, used to train a network to play the game of Go, among other well-known games [1]. Evaluation of its moves would be based entirely on its own estimation of future game outcomes. This mirrors the prospective neural network built to understand mathematics, as both would be fully responsible for learning the "rules" of their problem independently, and evaluating their success based on their own behaviours and knowledge, rather than by a person or by external, unchanging criteria.

In brief, the rules of Connect Four are as follows. Two players compete against one another to be the first to form a line of pieces of their respective colour, typically red or blue. These lines can be formed horizontally, vertically, or diagonally on the board, which is a grid of seven columns and six rows. Additionally, the board is arranged vertically, so pieces may only be placed in the lowest unoccupied space within a given column. If the board is completely filled and no player has succeeded in creating a line of four of their pieces, the game ends and is considered a draw.

In a game of Connect Four played by humans, it is considered obvious that one cannot attempt to place a piece in a column that has already been

filled, since no space exists for such a piece and in a real world situation the piece would fall off the board or into another column. To a program, however, such visual intuitions must be learned. For this reason, rather than forbid the placing of a piece into a full column, we allow such a mistake and have it result in a loss. This means that the network need only worry about selecting a column without restriction, while looking to maximize its wins and minimize its losses. It always has the ability to select whichever column it would like, based on its weights and biases, but it will gradually learn not to select a column that it knows to be full, since doing so will result in a loss. It is in this way that the network learns the rules of the game without being told them explicitly.

To implement the network and other infrastructure for the Connect Four Project, the programming language Python was used, alongside the TensorFlow library, developed by the Google Brain team for public use in deep neural networks, among other tasks [2, 3]. In addition to TensorFlow, the Numpy library was used regularly for its applications in linear algebra and uses in storing iterable objects containing lists of data [4]. Finally, the Nvidia CUDA platform was used to enable TensorFlow to make use of graphics processing to accelerate computations [5].

2 Game board generation

The first step in the Connect Four project was the creation of the game infrastructure itself. In order to store the board components in a simplified manner, three objects were used, being lists in this case, though other iterable objects could easily be substituted in place of these. Inspired by the Bitboard method of storing Connect Four boards, two of these lists stored the positions of the first and second player respectively, with the digit 1 in places where a piece belonging to that player was located and 0 elsewhere [6]. While these lists would theoretically suffice to describe the entire board, a third list was added for convenience, which denoted with the digit 1 the locations where new pieces would fall, with the digit 0 occupying the remaining positions; in other words, it stores the bottommost empty position in each column, assuming one exists. These three data structures interact with the methods used to modify the board and to verify its states, which can ultimately be called by the network or by a user.

One key step of implementing a game in a way that can be used by a

network is creating the game boards that it is to evaluate. Initially, this was done purely at random: a random number from 0 to 42 was generated, and then a game was played at random until this number of pieces had been placed. The game would end early if the next move would result in a win or a loss, to ensure that the network would only have to interact with games that still required interaction in order to be completed.

One issue with the above implementation of generating boards is the random component. Truly random game positions have a high probability of being unrepresentative of positions reached during true games, meaning that the network would be likely to spend much of its time training on the sort of board it would never see during a game against a person or against another computer program. In order to solve this, the board generation was altered to make use of the network itself to make its moves. Early iterations would make use of a relatively untrained network, and would thus be fairly random, but as time progresses and new boards are generated after the network has been trained, the boards would more and more resemble the results of actual games and would reflect the network's training and experience with the game. There is, however, a potential problem with this approach: if the initial model is heavily biased toward a certain style of play (for example, playing pieces in only the left half of the board), boards generated would reflect this, and the network would train based on these boards, potentially reinforcing this playstyle. The result would be a model oblivious to an important subset of board positions, simply because it has never thought to try creating them. The solution chosen to address this problem is to have the boards typically be generated based on the model's decisions, but occasionally make a choice at complete random to ensure some variety.

An additional bias was added to the board generation methods: the bias to prefer games that were a single move away from their end, either allowing the model to complete a row of four pieces in one move, resulting in a win, or to place a piece in an illegal position, resulting in a loss. Particularly in shallower implementations of the network, this would ensure that the network would have the opportunity to learn two of its most basic behaviours: avoiding playing pieces in full columns (so-called "prohibited moves") and ensuring to play a move resulting in a win whenever the option is available. To do this, some of the boards generated for training were played up until the point where the game ended, and then rewound by one move in order to produce a board through which the model could learn the above behaviours.

3 Training data and desired probabilities

Perhaps the most important component of the program when it comes to training the network is the method known as *desired probabilities*, as well as its several alternate versions created over the course of the project. The goal of this method is rather straightforward: to take in some board position for a game of Connect 4, and to judge which of the seven available moves should ideally be made next. This is judged using a Monte Carlo approach, where games are played at random and the results are tracked. Similarly to the generation of the boards, the probabilities involved in the random decisions are based on the suggestions of the network itself, meaning that as the network trains, the games it uses to judge outcomes will become more and more representative of real games, and ideally will involve increasingly skillful decisions, which may eventually outmatch the skill of a human player.

One basic component of the desired probabilities code is determining the "value" or "score" of a given position. We define this value based on the odds of winning a random game (based on the probabilities of the model) starting at this particular position. To do this, we start by playing some number of random games. For the majority of this project, we selected 10 to be the number of games to play, though higher numbers should theoretically lead to a greater representativeness of the true value of a position. We have the program play these games, using a method which takes the softmax output of the network and selects a number from 1 to the length of the output (in this case, seven), using each of the entries as the probability of selection. When a game ends, we check if the result was a win or a loss for the most recent player using a method built into the game board object, and check which player played that move. Using this information, we can decide if this was a win or a loss for the starting player for the original position in question. We track the wins and losses for this player, simply counting the number of each. We use these counts to determine the final score for the position, by subtracting the number of losses from the number of wins, and then dividing by the number of trials. This would mean that if every game from a position resulted in a win, the score would be 1, and if every game resulted in a loss, the score would be -1. Games with mixed wins and losses would fall somewhere in between, but in general negative numbers represent unfavourable positions where a loss is more likely and positive numbers represent favourable positions where a win is more likely. Notably, we did not involve draws in the math, since they were not considered to necessarily be something the network should avoid

or aim to achieve. In theory, one could create a more aggressive network by training it to avoid draws, or a more cautious network by training it to aim for draws as well as wins, but these changes would likely make a very small difference since draws in Connect 4 are fairly rare. The following equations represent these different value functions, where x_1, x_2, and x_3 represent wins, draws, and losses respectively. V_1 is the function we used, V_2 is the theoretical "aggressive" function, and V_3 is the theoretical "cautious" function.

$$V_1 \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \frac{x_1 - x_3}{x_1 + x_2 + x_3}$$

$$V_2 \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \frac{x_1 - x_2 - x_3}{x_1 + x_2 + x_3}$$

$$V_3 \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \frac{x_1 + x_2 - x_3}{x_1 + x_2 + x_3}$$

To make the desired probabilities code work, we must do more than evaluate the odds of winning from a particular board position. Since we wish to teach the network which of seven moves is best to make at a given point, we must determine the value of each of the possible moves from that position. Using the algorithm described above, this becomes quite simple: ignoring the initial suggestion of the network, we play a piece in column one and determine the value of the resulting board. We repeat this process for each other column, obtaining the value for each. In the results, the greatest number will correspond to the column that it is best to make a move in, while the smallest number will represent the column where a win is least likely. However, this ignores one possibility: the possibility of the first move resulting in a win or loss. One basic way of resolving this would be to set the score to 1 for an immediate win and -1 for an immediate loss, representing that 100% of the games starting with a move in this column either result in a win or result in a loss. However, to augment the network's ability to learn to avoid prohibited moves and instant wins, we altered this code somewhat, causing the score for a game which ends immediately to be set to number with a greater magnitude, positive or negative depending on a win or a loss. Initially, this greater number was selected to be 3, but it was later switched

to 10 to further prioritize the learning of immediate wins and losses. Notably, for an immediate draw, the score was simply set to 0. It was considered more important for instant wins to have a particularly high score as opposed to instant losses, since, when using the softmax method, small numbers are not changed as much as larger numbers. Finally, once the score was calculated for each of the seven possible initial moves, the results were processed using a softmax algorithm in order to represent them as positive numbers which summed to 1, meaning they corresponded to the probability that a computer program should select their respective column in a game of Connect 4. The softmax function S is shown below

$$S\begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} = \begin{pmatrix} e^{x_1} \\ e^{x_2} \\ \dots \\ e^{x_n} \end{pmatrix} \left(\sum_{i=1}^{n} e^{x_i} \right)^{-1}$$

4 Improvements to desired probabilities

After some testing of the neural network, a slight shortfall was detected. While the network was able to learn to avoid prohibited moves and to play moves that would immediately result in a win, it did not learn to block its opponent's attempts to win. In a scenario where a player's opponent can win on the next turn, and the active player cannot win this turn, the only move that could lead to their victory is preventing their opponent's winning move. Due to the way the desired probabilities code was designed, however, the computer made no particular distinction between the possibility of losing on its opponent's next turn and losing in the distant future, so it made no particular attempts to prevent its opponent from winning. In order to teach it this desired behaviour, a change was made to the desired probabilities code to track wins and losses in a slightly different fashion. Rather than simply counting the number of wins and losses in the prospective games in the board value code, a different number would be added to the total depending on how soon in the future the win or loss occurred. An initial implementation of this was to have the number be inversely proportional to the number of moves made since the initial version of the board, with a proportionality factor of 10. This is to say, the value of the win or loss is equal to 10 divided by the number of moves made since the initial move. One potential issue with this

approach is that moves in the relatively distant future will add a number very near to 0 to the total, meaning they might have no impact on the desired probabilities, so some other possible functions for determining the value of the win or loss could be the above function but rounded up to 1, the above function but with 1 added in all cases, or possibly a negative-slope linear function which flattens at $y = 1$.

The actual implementation of the desired probabilities code went through a number of iterations. It was initially coded with little regard for efficiency as a simple proof of concept. The aforementioned process for determining the value of a position was implemented as its own method, where a board position was inputted, and following the random gameplay, the board's score was returned. The main desired probabilities method took the initial board as an input, and then determined the score of the board following each of the seven possible initial moves, and applied the softmax function to these results. In order to determine the moves made during the games, an additional method was designed to select the probabilities, and the probabilities themselves were obtained as the suggested moves by the network for the given boardstate, using the Tensorflow model.predict() method. This implementation was useful due to the intuitive nature of its approach. Each game was handled individually, played until its end was reached and the results could be reported back to the desired probabilities method. The intuitive design, however, was vastly outweighed by performance issues caused by some of the design choices. The most glaring flaw was the poor performance of the model.predict() method, which is not intended to be called nearly as frequently as was done in the board value method. An initial solution was to replace the model.predict() method with the simple model call method (simply model()), which significantly improved performance time owing to the fact that this method runs significantly faster on smaller inputs, such as those used in this implementation.

While switching model.predict() to model() did improve performance, the performance of the method was still far too poor to be used in training the model. In order to further improve performance, it was noted that the model() method could operate on a large number of inputs simultaneously at approximately the same speed as a single input. Assuming a single use of model() takes one unit of time, calling model() one hundred times takes one hundred units of time, yet calling model() on all hundred inputs at once would only take approximately one unit of time. While the exact factor by which this increases performance is slightly unclear, testing showed that it could

majorly improve performance for even extremely large input sizes, larger than we expected the desired probabilities method to ever have to handle. In order to implement this, however, it would be necessary to deconstruct the aforementioned structure for desired probabilities to allow the games to run in parallel, since the original method played the games until they were finished, one at a time. The initial board was copied and placed into a list with length equal to 7 times the number of trials for each of the 7 initial moves. These initial moves were made in the respective blocks of the list, and then the games began. Each move, the data from all of the boards was compiled to be used as an input for the model() call, and the output from that method was then split up for each board and used to decide the move for that board. A separate list stored references to each of the games that had finished, and another stored the data on the results of each game to be later used to calculate the board scores and eventually the desired probabilities. Games marked as finished no longer had moves made, and if any of the seven initial moves resulted in a win, loss, or draw, all of the games based on it would immediately be marked as finished. Earlier versions of this method, and of the previous desired probabilities methods, had move limits in place, so the method would only test hypothetical moves up to a certain number of moves before stopping. This had been in place to improve performance, preventing the method from playing almost an entire game of Connect 4 should it be given a nearly empty board. However, as performance improved, the method was altered to simply play all games until they had finished.

In an attempt to further improve performance, an additional version of desired probabilities was implemented, one that would determine the desired probabilities for multiple initial boards simultaneously, theoretically improving performance by completing all model() calls in parallel. This method was referred to as *desired_probabilities_batch*. In general, the implementation was the same as the single-input desired probabilities, with slightly different data structures for handling so many boards of different origins. Some additional work had to be done to track when all of the games had been finished, since it would take dramatically different numbers of moves to finish different games. This method was tested in comparison to the single-input method to determine whether it had improved performance, and the results were somewhat surprising. The runtimes for the two different methods appeared to be nearly identical in most cases. When a sufficiently large number of initial boards was used, it became clear that the batch method did have a slightly

faster runtime, but only by a very small percentage, no more than 5% in the tests we completed. While the reason for the very small improvement is unclear, it may be the case that the inputs to model() were sufficiently large to begin to slow the method down. Both methods remain in the code, and while the batch method is used wherever possible, it seems they are largely interchangeable.

5 Network parameters

With the ability to generate the desired outputs for given game board inputs, it was possible to begin training the network. While the layout of the hidden layers was fairly variable, the input and output layers were simple. The output layer would use a softmax activation function over seven nodes, generating seven probabilities determining the odds that a trained network would play a move in the respective columns of the board. The input layer would accept a numerical summary of the board. Since a standard Connect 4 board has seven columns and six rows, this could be done with an input layer of size 42. However, in order to provide the network more data to work with, an input size of 126 was selected, making 42 nodes for each of the three representations of the board within the board class. Each input could be 1 or 0. The first two sections represented red and blue pieces respectively, while the third represented placeable positions. These would give the network information on its own board state as well as that of the opponent, as well as the points on the board where its pieces would ultimately fall, which should prove particularly useful to the avoidance of prohibited moves.

Because of how we generate the target data for training, we chose to perform training in a particular and possibly unusual way. In addition to performing multiple epochs of training, we iterate several times over what we called the *outer loop*, within which lay our iteration over the epochs of training. Each instance of the outer loop, we would generate a new set of sample boards with which to train, as well as extracting the training data (126-digit binary strings representing the board state) and the target data (the desired probabilities for each board). Using this data, we would train the model over a large number of epochs, and then we would move to the next instance of the outer loop, until we had finished all of these. The reason for this approach is the fact that the desired probabilities are generated using the model itself, so as the model gets better and better the training data

should become increasingly useful to the training. All of these numerical parameters, meaning the outer loop repetitions, epochs, set and batch size, as well as the games tested for desired probabilities, were designed to be variable for testing, but in general, for training sessions, a set size of several thousand was selected, with each batch about one tenth this size. About 100 epochs was the usual number used in order to prevent overfitting to particular sets of samples, and the outer loop number tended to be about 10. The number of trials in desired probability calculation remained at about 10. The Adaptive Moment Estimator, or *Adam*, algorithm was used as the optimizer for the network [8].

6 Performance evaluation

In order to evaluate the performance of the network as it trained, a method was created to compute the average euclidean distance between the desired probabilities for a set of boards and the actual output of the network itself, a distance which would range from 0 to $\sqrt{2}$ given that the vectors involved have exclusively components ranging from 0 to 1. The euclidean distances for each individual board was computed within the function and made visible with the right parameters, and these were used to compute the average distance, corresponding roughly with the performance of the network, with a distance near 0 representing good performance. The library Matplotlib was utilized in order to represent the model's performance as it trained [7].

The results of the evaluating method would theoretically be much less useful if a different set of boards were evaluated each time, since the number would correspond largely to the network's fit to that particular set and random chance would play a much larger role in the evaluation. To rectify this problem, an evaluation set was added to the code. This was a set of ten boards (though the number could easily be altered) which would be generated at the beginning and never altered and which would be evaluated after each iteration of the outer loop. Assuming this random set is reasonably representative of the set of all Connect 4 games, improvement in the network's predictions for this set would be indicative of the network's overall performance.

A final tool for testing the network was a simple console-based game module, where a human user could play a game of Connect 4 against a trained network. Each boardstate was fed as input to the network, and a

column was selected based on the output of the network. While this tool is very simple and not useful in training itself, it was very effective in observing the actions of the network in particular scenarios and testing its overall skill relative to a human player.

7 Network implementations

The number of hidden layers could vary wildly depending on implementation, but in our initial version we made use of a shallow network with one hidden layer of a thousand nodes. This layer made use of a sigmoid activation function with randomly initialized weights and biases, initialized automatically by TensorFlow. With this number of neurons, it was not suspected that the network would be able to learn to play Connect 4 in any advanced fashion, but it was hoped that it would learn prohibited moves and instant wins. Testing with the console game revealed that this seemed to be the case: no instances were observed where the network decided to play a piece illegally, and the model seemed to identify instant wins the vast majority of the time. It seems likely that the network learned prohibited moves more effectively thanks to the data from the third of the input sections, since a particular grouping of zeros would correspond to the lack of an available space in a particular column. The actual output probabilities of the model in these two scenarios tended to be rather decisive, with high 90s in winning columns and near-0 chances in prohibited columns.

In order to obtain better results from the network, the architecture was later expanded to have two hidden layers. Rather than restart training from scratch with a brand new untrained model, the technique known as transfer learning was utilized in the model [9]. In brief, transfer learning involves using an older, shallower network as one of the layers of a deeper network. In theory, the shallower network will serve as a *feature detector* and will interpret some elements of the input and somehow summarize them, and the deeper network will use the data given by this network to assemble a more complete picture and hopefully obtain the correct output. For example, a shallow network may be able to isolate eyes in an image, whereas the deeper network can use this data to recognize and count the number of faces in an image using these features from the shallow network.

A potential flaw with using transfer learning was the chance that our shallow network implementation was effectively ignoring certain features in

the inputs, meaning all future layers of the network would be blind to these features. For this reason, additional nodes were added on the first hidden layer to transfer the input data through to future layers. Ideally, these would have made use of a pass-through activation function ($f(x) = x$), but in TensorFlow activation functions must be consistent across a layer, making this impossible. A workaround was to continue using a sigmoid activation function with particular weights and biases. The bias for these nodes was set to -50, and the weight for the respective input component was set to 100. Since input values can only be 1 or 0, an input of 1 would result in an output of $\sigma(50) \approx 1$ and an input of 0 would result in an output of $\sigma(-50) \approx 0$. The weights from all of the other input components were set to 0 for each of these pass-through nodes.

In order to maintain the performance gained when training the shallow network, the deep network had to be set up with a particular initial configuration. The second hidden layer was designed to keep the same output as the shallow network's first hidden layer. In order to increase the number of degrees of freedom, the second hidden layer was set up with twice the number of nodes as the first, in this case 2000 (since the pass-through nodes were not counted). In order to maintain the output of the shallow network, a particular arrangement of weights, biases, and functions was used. Each of the non-pass-through nodes on the first hidden layer was connected (with a non-zero weight) to exactly two nodes on the second hidden layer, and each of the nodes on the second hidden layer was connected to precisely one of these nodes. For a given node on the first hidden layer, the first of the nodes that it connected to would have the same output weight as it had in the shallow network, and the other would have an output weight with the same magnitude and opposite sign. The weights to these two nodes on the second hidden layer were 1 and -1 respectively, and each of the biases were 0. The second hidden layer made use of the ReLU activation function, where the output is equal to the input if the input is positive, and is equal to 0 otherwise. Now, consider x to be the output of some node on the first hidden layer. If $x = 0$, the input to each of the connected nodes will be 0, meaning the output from each will be 0, meaning the node will contribute a value of 0 to the output layer. If $x > 0$, then the output of the first connected node will be x, but the second connected node will output 0, since $-x < 0$ and the ReLU function will output 0 as a result. Similarly, if $x < 0$, the second node will output $-x$ and the first 0, which combines with the negated output weight to produce an identical overall output. The result is that the

output of the first and second hidden layer combined will be the same output as the first layer in the shallow network. The first hidden layer was frozen, making it unable to change during training, meaning it will maintain all of the data it obtained during its previous training. The output weights of the pass-through nodes are initially set to 0, though they can change over the course of training. The result is a deeper network with twice as many degrees of freedom that maintains the initial behaviour of the original network and should only be able to improve upon it.

8 Analysis, conclusion, and next steps

Following the implementation of the deeper network using transfer learning, the network appears to demonstrate slightly more complex behaviours than when using the shallow network. The network continues to avoid prohibited moves seemingly all of the time, and capitalizes on instant wins the majority of the time. It occasionally manages to block its opponent's attempts to win, but this behaviour is much less consistent. It does seem, however, that the blocking behaviour is a little more than chance, with the network displaying a fair amount of preference for these moves in many situations. Often, however, this is disrupted fairly often by a different behaviour of the network, wherein the network seems to highly prioritize creating a line of pieces even if this will cause it to lose the game. This behaviour is somewhat difficult to explain, and may be caused by several factors. Firstly, the desired probabilities do attribute value to winning several moves in the future, meaning the network should learn to continue creating a column if there is no cost to doing so. However, the desired probabilities should prioritize avoiding a loss in the near future over a slightly more distant victory, so this does not seem to fully explain this behaviour. Another possible explanation is that a board with a partially completed row is rather similar, structurally, to a board with an immediate win. Though one fewer piece belonging to the network is on the board, it may be that all of the existing weights and balances work to nonetheless place in the next column, comparing it with an immediate win and ignoring the slightly-less-valuable blocking move. This explanation seems more likely, given that the network will occasionally continue forming a line even when its opponent has blocked it from winning. Notable also to this behaviour is the fact that this occurs most often within a single column. The reason for this is unclear, and does not seem to be linked with either of

the above explanations.

Another minor issue encountered during training attempts was the issue of overfitting to the evaluation set. In some versions of the code, the evaluation set was fed to the network alongside the training set. In these instances, the graph of euclidean distance over the course of training had a much steeper downward slope. This steeper downward slope is likely because the network was using the evaluation set to train and therefore getting better at correctly generating probabilities for the evaluation set. It was unclear, however, whether this was due to overfitting to the evaluation set, especially since the training set was larger by one to two orders of magnitude in most tests. Whether or not overfitting was occurring, however, it is likely best not to include the evaluation set in the training set, since at the very least the euclidean distances calculated become unrepresentative of the model's general behaviour.

A possible alteration to the code could be the changing of the *pick_probability* method from a random decision based on the probabilities to a method which simply selects the largest number in a set. This may be useful in the cases where the network displays only a slight preference toward the correct course of action, but may also cause many problems if the initialization of the network consistently leads it to prefer certain suboptimal courses of action. This selection of the highest probability may be the most useful in games of a player against the network to ensure that the network avoids random mistakes. In general, the utility of this change remains to be tested.

Next steps for the Connect 4 Project include continued testing with the current two-hidden-layer architecture and further analysis of the behaviours explained above. It seems that the blocking moves discussed above are one of the next behaviours the network should be able to learn, and experimentation may be required to determine why they are prioritized less strongly than expected. One way of doing this may involve altering the mathematics of the desired probabilities code so that the value of a win or loss decays differently depending on how far into the future it occurs. Another valuable addition to the current architecture may be fine-tuning, which is a technique often used alongside transfer learning involving the unfreezing of frozen nodes and the training of these nodes with a very small learning rate, to allow the network to optimize itself slightly following a longer, more constructive training session [9].

Additionally, it may be desirable to alter the network architecture such that it is much more dynamic, allowing it to add new nodes or even entire

layers when it is determined that the current available nodes do not offer sufficient degrees of freedom. Unfortunately, TensorFlow is not constructed in such a way that allows dynamic architecture, but Keras, the library upon which TensorFlow is built, does allow for such things. Another possibility for dynamic architectures is the library PyTorch, which seems fairly promising after some brief exploration of its capabilities [10]. It would likely also be beneficial to slightly rewrite the program's code to be slightly more modular, separating the model code and gameboard code to the greatest possible extent.

Overall, the current architecture is showing some promise in learning the desired basic behaviours in the game of Connect Four. The current layout needs some work to improve the capabilities of the network, but it seems as though it would be more valuable to add onto the current architecture rather than restart from scratch, whether it be additional layers or dynamic elements which would allow the network to modify itself to overcome challenges. Some of the network's current behaviours are difficult to explain, and a slight modification to how target data is constructed by desired probabilities code may be necessary. Above all else, more experimentation would be greatly beneficial to more thoroughly understand the current state of the network.

References

[1] Silver, D., Hubert, T., Schrittwieser J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., Hassabis, D. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.*
`https://arxiv.org/pdf/1712.01815.pdf`.

[2] Google Brain Team. *Introduction to TensorFlow.*
`https://www.tensorflow.org/learn`.

[3] Google Brain Team. *All symbols in TensorFlow 2.*
`https://www.tensorflow.org/api_docs/python/tf/all_symbols`.

[4] NumPy. *NumPy.*
`https://numpy.org/`.

[5] NVIDIA Developer. *CUDA Toolkit.*
https://developer.nvidia.com/cuda-toolkit.

[6] Herzberg, D. *Bitboards and Connect 4.*
https://github.com/denkspuren/BitboardC4/blob/master/BitboardDesign.md.

[7] Droettboom et al. *Matplotlib.*
https://matplotlib.org/

[8] Doshi, S. *Various Optimization Algorithms For Training Neural Network.*
https://towardsdatascience.com/
optimizers-for-training-neural-network-59450d71caf6.

[9] TensorFlow team. *Transfer learning and fine-tuning.*
https://www.tensorflow.org/guide/keras/transfer_learning.

[10] PyTorch. *PyTorch documentation.*
https://pytorch.org/docs/stable/index.html.

[11] Michael Nielsen. *Neural networks and deep learning.*
http://neuralnetworksanddeeplearning.com/index.html.