Designing a Convolutional Neural Network Connect Four Opponent

Neil Wei and Randall Driscoll

I. Abstract

The purpose of this project was to design and train a convolutional neural network model capable of competitively playing Connect Four against human players. To simulate this, the convolutional neural network, henceforth referred to as the CNN, was made to play against a random player opponent, whose moves are selected at random unless it is able to win or block its opponent from winning. Using board configuration data collected from these games, the CNN was designed to learn attributes of a favorable board configuration and play the move that resulted in a board configuration with the highest probability of being classified as a winning configuration. Our objective with respect to the random player opponent was to achieve a 70-80% winrate. In the end we were able to achieve this goal.

II. Introduction

The process of winning games of logic and strategy have been intrinsically tied to algorithmic game theory for centuries, as mathematicians have long sought to solve games. With machine learning, a different avenue of exploring the mathematics behind games opens; rather than determining an algorithm that guarantees success, the focus is placed on algorithms that can learn the mathematical attributes that constitute playing a game well.

Our curiosity and interest in applying machine learning principles to games led us to pursue Connect Four with a machine learning perspective. The natural resemblance of the game board to an array where pattern detection is the key to success made the convolutional neural network model a logical approach to the problem.

Our CNN model takes input as a board configuration in the form of a 6x7 array, and outputs a vector containing the predicted probabilities of a tie, a player 1 victory, and a player 2 victory. In practice, as the CNN model is asked to make moves in Connect Four, it generates board configurations for each possible move it can make, inputs each configuration to the model, and selects the move that outputs the highest probability of the CNN player's success.

III. Related Work

To perfectly play connect 4 using an algorithm, others have used the Monte Carlo tree search (MCTS), which calculates all possible combinations of moves from the current board configuration and returns the move that results in the largest number of wins. Because connect 4 is a solved game, this will theoretically create an unbeatable algorithm, however this is very resource heavy and requires a lot of computational power. There are optimizations such as mimimaxing by assuming the opponent also makes the best moves, alpha beta pruning out the

less possible moves, and creating a hash table of board configurations already visited, however this would still take too much time.[1]

The second paper is an example of a previous attempt of a similar project. They used 4 different ways of setting up the neural network and convolutional layers, and played these against a MCTS algorithm. By training four models it was easy to compare which filters performed better. To reduce computational power, the MCTS algorithm only looked forward to a set amount of moves, which greatly reduced the time taken to train the model.[2]

The third paper goes over training an algorithm for Connect Four using a deep neural network with three CNN filters played against a MCTS algorithm. They also used a Kaggle data set containing board positions and trained the algorithm from that data, which was much faster than manually deriving the data.[3]

The fourth source implemented an idea of playing the model against itself to train itself, and give feedback to the model. This was first notably used in the Alpha zero, which many Connect Four sources reference due to it's similar 2d grid analysis. Using this resulted in a more robust model an an independent opponent algorithm was not needed.[4]

The idea of self playing models can be further expanded upon. The fifth and final source shows how the models can play against previous versions of itself each iteration using agents to play the games with different models, resulting in an even more robust final model, and serving as validation that the model is improving.[5]

IV. Dataset and Features

The dataset used was continually generated throughout the training process as a result of the games that the CNN model played, where each game saved its final and pre-final board configurations as arrays to be used as training data. At the end of each interval, the board configurations collected throughout those games were used to train the CNN, such that the size of its cumulative training dataset increases with each interval. 80% of these configurations were used for training folded into 3 epochs, while 20% were set aside for validation. We then plotted both the CNN and opponent's win rates, this would give us information on how well the model was winning and how well it prevented the opponent from winning.

```
[[[0. 0. 0. 0. 0. 0. 0.]
  [0. 1. 0. 0. 0. 0. 0.]
  [1. 2. 0. 0. 0. 0. 0.]
  [2. 1. 0. 0. 0. 2. 0.]
  [1. 1. 1. 0. 2. 1. 0.]
  [2. 1. 1. 0. 2. 2. 2.]]

 [[0. 0. 0. 0. 0. 0. 0.]
  [0. 1. 0. 0. 0. 0. 0.]
  [1. 2. 0. 0. 0. 0. 0.]
  [2. 1. 0. 0. 0. 2. 0.]
  [1. 1. 1. 0. 2. 1. 0.]
  [2. 1. 1. 2. 2. 2. 2.]]
```

Final and pre-final boards used for training

V. Methods

The primary component and key machine learning element of this project was the convolutional neural network used to predict the best move, implemented as a sequentially layered Keras neural network. The first layer was a 2 dimensional convolutional layer composed of 64 4x4 filters, ReLU activation, and a 6x7 array input shape.

As Connect Four is a game focused on lines of 4 consecutive pieces, the 4x4 filter size was chosen with the intention of extracting features indicative of these lines and their correspondence to winning or losing positions. Each 4x4 filter is shifted across every unique 4x4

position of the board, and, with the ReLU activation function, the convolved feature is derived from:

$$a_n = ReLU(W^{[1]}x_n + b^{[1]})$$

Where a_n represents the convolved feature, x_n represents each unique 4x4 position of the board, and $W^{[1]}$ and $b^{[1]}$ represent the learned parameters that are optimized through training. The primary convolutional operation that is done in this formula is the dot product that is taken from $W^{[1]}x_n$.

We next use a flatten layer to condense the output of the convolutional layer into a 1-dimensional vector for efficient and compatible propagation through the dense layers that follow.

The first two densely connected layers are each composed of 42 neurons and, similarly to the convolutional layer, use the ReLU activation function. We decided to use two of these layers due to their adeptness at retaining weights and information, with the objective being that the information calculated from the convolutional layer will propagate through and reside within the dense layers. Each neuron of each layer is fully connected, meaning that the weights that it calculates and stores are computed as a function of all outputs of all the previous layer's neurons.

All together, the convolutional layer and dense layers work in tandem. The convolutional layer learns optimal filters, uses those filters to extract features, and outputs the convolved results to the first dense layer. The neurons in the first dense layer each derive weighted features as a function of the convolutional layer's output, and the neurons in the second layer each derive meta-features as a function of all outputs from the previous layer.

The final layer is another dense layer composed of 3 neurons and uses the softmax activation function, which outputs categorical probabilities that sum to 1. This layer outputs a vector of 3 probabilities which, in order, correspond to the probability of a tie, a player 2 (the random player) victory, and a player 1 (the CNN model) victory. Moves are made by choosing the move that results in a board configuration that is most likely to be classified as a player 1 victory.

In order to optimize the neural network's weights, an optimization function needed to be selected. For this purpose, the Adam stochastic gradient descent algorithm was selected due to its versatility and strengths with respect to large amounts of data and parameters.

As for the loss that was to be optimized by Adam, the categorical crossentropy loss function, which computes crossentropy loss with respect to true labels and predictions, was selected for our model due to its high compatibility with classification problems. The neural network model that was created for this project largely operates as a classification algorithm, as it classifies board configurations as winning or losing and assigns probabilities to those labels.

VI. Experiments / Results / Discussion

For the CNN model, we decided to use a 4x4 convolution filter because a win state was more likely in a 4x4 grid, which the model could detect. We found the best results running 2

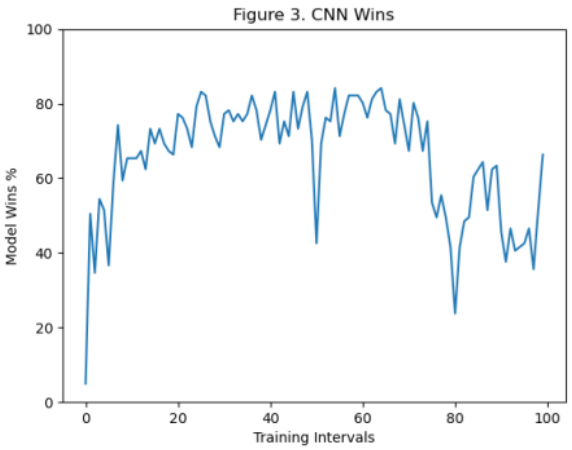
dense layers of 42 nodes with the final layer having 3 nodes with outputs that represented probabilities of player 1 winning, player 2 winning, and a tie game. We used categorical cross entropy loss and measured accuracy. To prevent overfitting from previous tests, we cross validated with 3 folds and an 80:20 training/test validation split. Here are the results:

The first successful time we ran this was with the model training with only the last board configuration of each game:

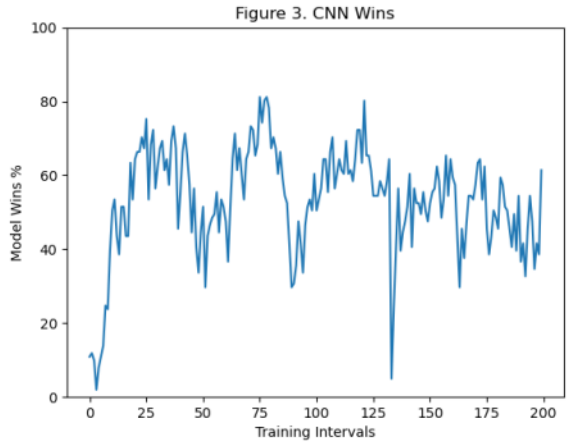
```
_
P1 Wins: 58
P2 Wins: 16
Epoch 1/3
3/3 [==============================] - 0s 22ms/step - loss: 0.4676 - accuracy: 0.7375 - val_loss: 0.3563 - val_accuracy: 0.8095
Epoch 2/3
3/3 [==============================] - 0s 11ms/step - loss: 0.4141 - accuracy: 0.8250 - val_loss: 0.3330 - val_accuracy: 0.8571
Epoch 3/3
3/3 [==============================] - 0s 11ms/step - loss: 0.3735 - accuracy: 0.8750 - val_loss: 0.3342 - val_accuracy: 0.9048
```

This resulted in an average validation accuracy of 85%. Once the model had completed 100 training intervals of 100 games each, P1(CNN model) won 58 games and the opponent won 16, with 26 ties. If we count each tie as half of a win, this returns a 71% win rate. This large amount of ties was consistent; for the last 10 training intervals, there were 21.2 ties per 100 games, on average.

In the second attempt we factored in the second to last board configuration as well as the final board configuration into training the model. This would give us the board configurations of the two final moves instead of just the winning one. The results were then plotted. This resulted in much less ties than before, as no set of 100 games had over 5 ties. The CNN reached above 80% winrate, but one thing we noticed was that our results were not extremely consistent, we found that at around 80 iterations, the performance drops drastically, which may be due to overfitting. Another strange thing we noticed was that our training accuracy tended to be lower than our validation accuracy.



In another attempt, we ran the model through 200 iterations and cross validated through 5 folds to further explore the dip in performance and combat possible overfitting. There seems to have been areas of growth followed by areas of decay, eventually evening out around the 50% win rate. This is suspect of having much more overfitting and would require more modifications to the model. This is also evidenced with our accuracy data during the last training session. While our



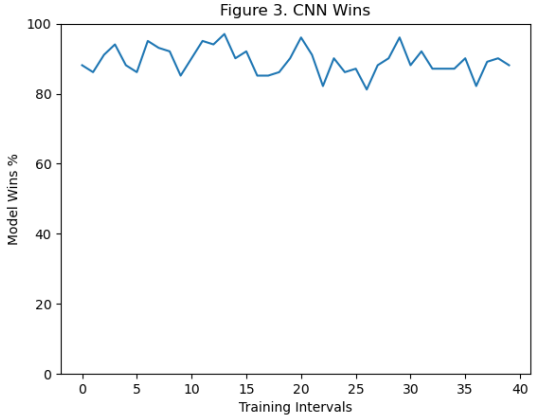
validation and training accuracy made more sense with 200 iterations, the training accuracy was significantly higher than the validation accuracy, considering the model was playing against a random player, further signifying overfitting. A 5-7% difference in accuracies would have been ideal, but our final difference was around 15%.

```
Starting training interval 200/200
100.00%   time: 384 secs
Time for 100 games: 383.98811507225037

P1 Wins: 62
P2 Wins: 31
Epoch 1/5
6/6 [==============================] - 0s 12ms/step - loss: 0.4808 - accuracy: 0.7826 - val_loss: 0.5960 - val_accuracy: 0.7561
Epoch 2/5
6/6 [==============================] - 0s 7ms/step - loss: 0.3945 - accuracy: 0.8199 - val_loss: 0.6078 - val_accuracy: 0.7561
Epoch 3/5
6/6 [==============================] - 0s 7ms/step - loss: 0.3669 - accuracy: 0.8323 - val_loss: 0.6458 - val_accuracy: 0.7073
Epoch 4/5
6/6 [==============================] - 0s 7ms/step - loss: 0.3424 - accuracy: 0.8571 - val_loss: 0.6412 - val_accuracy: 0.7073
Epoch 5/5
6/6 [==============================] - 0s 7ms/step - loss: 0.3129 - accuracy: 0.8634 - val_loss: 0.6305 - val_accuracy: 0.7073
```

It's also worth noting that the additional logic that allows the random player to detect rows of three so that it can win and, more significantly, block its opponent from winning, often make it a difficult adversary for our CNN model. This ability means that, commonly, the only way for the CNN model to win is to set up two rows of three with the same move, so that a row of three is still available to secure a win after the random player blocks one of them. With this in mind, our model's 80% peak win rate is much more satisfactory. Facing a random player without this logic, the CNN model that was trained in our second attempt mentioned above won the vast majority of games, as seen in the figure on the right.

Figure 3. CNN Wins



VII. Conclusion / Future Work

We approached this project with the goal of training a CNN model to win 70-80% of Connect Four games when playing a human equivalent player. We were able to accomplish this through training the network with boardstates from the games it played with a random player that checks for wins. We ran into issues with overfitting, ties, and consistency, but through manipulation of the training data and model parameters were able to achieve our goal. With a model that trained 100 iterations of 100 games, learning the last two board states of each game, which were put into the model consisting of a 4x4 convolution layer and 2 hidden dense layers of 42 nodes each, we were able to reach an above 80% peak win rate.

If we had more time and computing power, we could continue to go on to explore training by playing the model against current and previous versions of itself, as well as a few predefined opponents including random players and MCTS players that look ahead 300, 500, and 1000 moves. Ideally we would then be able to compare using the MCTS players. We would continue to experiment with different convolutional layers and sizes, and we would initiate training with predefined board state data from Kaggle.

A. Contributions

The work was evenly split among both group members. Neil worked on background research pertaining to previous uses of Connect 4 with machine learning, creating an opponent, developing an effective game environment, researching effective neural networks, Convolutional filters, and keras implementations of models to code, refining the model, and data manipulation to train it correctly.

In addition to also doing research on convolutional neural networks and implementation, Randall also wrote the first iteration of the model code, created the agent who uses the model to play the game, created the main file that ran the simulation, ran most of the simulations, and made further optimizations to the model to increase performance and combat overfitting.

Despite both members having areas they were focused on, most sections of the project were worked on together. Both members helped optimize the model, the implementation of the model, the game environment, and the main script. There were weeks when one person in the pair was very busy, and the other would help by contributing more work to stay on schedule. Notably during midterm 2 week, both members were busy and little progress was made then, but with collaborative coordination of frequent meetings and a well managed time schedule with effective short term and long term goals, the group never had to cram in work and progress continued smoothly.

B. References / Bibliography

Related Works

[3] Kim, Luke, et al. “Applying Machine Learning to Connect Four.” *Stanford.edu*, cs229.stanford.edu/proj2019aut/data/assignment_308832_raw/26646701.pdf.

[2] “Learning to Play Connect 4 with Deep Reinforcement Learning.” *Codebox Software*, codebox.net/pages/connect4.

[4] Surag. “Simple Alpha Zero.” *Simple Alpha Zero Tutorial*, 29 Dec. 2019, web.stanford.edu/~surag/posts/alphazero.html.

[5] Wang, Hui, et al. “Warm-Start AlphaZero Self-Play Search Enhancements.” *Leiden Institute of Advanced Computer Science*, 26 Apr. 2020, arxiv.org/pdf/2004.12357.pdf.

[1] Wisney, Gilad. “Deep Reinforcement Learning and Monte Carlo Tree Search With Connect 4.” *Medium*, Towards Data Science, 28 Apr. 2019, towardsdatascience.com/deep-reinforcement-learning-and-monte-carlo-tree-search-with-connect-4-ba22a4713e7a.

Libraries Used

Abadi, Martin, et al. “TensorFlow: A System for Large-Scale Machine Learning”, Google, 2 Nov. 2016, www.tensorflow.org/api_docs/python/tf.

Chollet, François, et al. “Keras”, 2015, https://keras.io.

Harris, C.R., Millman, K.J., van der Walt, S.J. et al. “Array programming with NumPy”. Nature 585, 357–362 (2020). DOI: 0.1038/s41586-020-2649-2.

J. D. Hunter, "Matplotlib: A 2D Graphics Environment", Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, 2007.

Virtanen, Pauli et al. (2020) “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. Nature Methods, 17(3), 261-272.