Playing Connect 4 with Artificial Neural Networks

Thomas Brewer

December 18, 2017

1 Summary

In this work, we set ourselves in the context of the popular game Connect-4 (C4) to see if we can train an Artificial Neural Network (ANN) to make decisions without explicitly stating the rules of the game. Specifically, we train a Convolutional Neural Network (CNN) to recognize the winner of a game using over 300,000 examples. We then link the model to an artificial intelligence (AI), which we refer to as `LearningAI`, in a C4 engine. The `LearningAI` uses the model to infer the best decision given the state of the C4 grid and a list of possible moves. To create a database of C4 games and test the performance of our model we use an AI with a hard coded strategy (`SetAI`). We define the `LearningAI`'s *performance* as the percentage of wins when playing against the `SetAI` over 500 games. To make sure that the `LearningAI` is not winning by random chance, we have also implemented the `RandomAI`, which places pieces at random. We define the *baseline performance* as the percentage of games won by the `RandomAI`, also over 500 games. We find that the baseline performance is about 2%, and the model's perfomance around 45%. Granted there is room for improvement, but this does indicate that the `LearningAI` has in fact learned how to play.

We start in Section 2 by introducing the basic structure of our C4 engine and its different AIs. In Section 3 we discuss how we use that engine to generate data for our model, and how that data is saved. We follow up in Section 4 by introducing the details of our model and our assumptions about how it is used to play C4. Finally in Sections 5 and 6 we discuss our model's performance, its caveats, and future models.

2 The Connect-4 Engine

Creating the C4-Engine was the very first step of this project. Without it we would not be able to evaluate the model's performance, generate data, or even play the game. The engine is simply a python class, `C4()`, which moderates the flow of the game. It is designed to be attached to any script to play games quickly, with the possibility of saving them as csv files. To play the game, the engine instantiates two other objects : the `Board()` and the `Player()`. The `Board()` object is responsible for defining the game grid (or board), keep track of available moves, positioning player pieces, and checking whether a player has won or not. The `Player()` class is actually a parent class that has three different children : the `SetAI`, the `LearningAI`, and the `RandomAI`. Details of the `Board()` and `Player()` classes are addressed in sections 2.4 and 2.1 respectively. With these objects, the engine is responsible for looping over the player turns until the game is over. The outline of this process is shown in Figure 1. When a game finishes, the engine also provides some functionality to save the game to a *csv* file. Details about this are provided in the Section 3. For now let us take a closer look at the `Player()` and `Board()` objects.

2.1 Players

Now that we've introduced the basic game flow, we can talk about the different AIs, or Players, and how they operate. As previously mentioned, the three different players are : `SetAI`, `LearningAI` (the whole purpose of this work), and `RandomAI`. Each AI is set up to have a `move` method which ultimately chooses which column the next piece should go in. The `RandomAI` simply returns a random column number with an available move. The other are a little more elaborate and we will discuss their basic flow in the following sections.

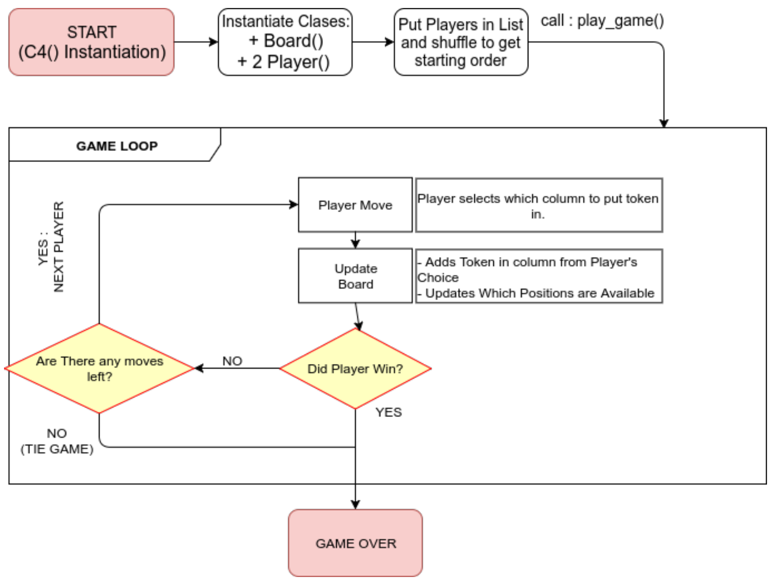
START
(C4() Instantiation)
Instantiate Clases:
+ Board()
+ 2 Player()
Put Players in List and shuffle to get starting order
call : play_game()
GAME LOOP
Player Move
Player selects which column to put token in.
Update Board
- Adds Token in column from Player's Choice
- Updates Which Positions are Available
YES : NEXT PLAYER
Are There any moves left?
NO
Did Player Win?
YES
NO
(TIE GAME)
GAME OVER

Figure 1: Basic flow for how the Connect-4 Engine runs through a game. The engine is designed to be attached to outside scripts, but does provide functionality to save games.

2.2 SetAI

The SetAI makes decisions based on a hard coded strategy which we illustrate in Figure 2. As we can see this not the most elaborate strategy, as it only looks one turn ahead looking for *winning or losing plays*. By winning plays we mean where it already has three in a row, and can place its piece in empty position. Conversely, a losing play is when the opponent has three in a row, but the player can block it. The details about how it finds these plays is discussed in Section 2.4. The AI starts by looking for winning plays, if it finds none it moves on to losing plays, and if there is still nothing it simply plays at random. The strategy is not very elaborate, but the SetAI is only a tool used for generating data and testing the performance of the LearningAI.

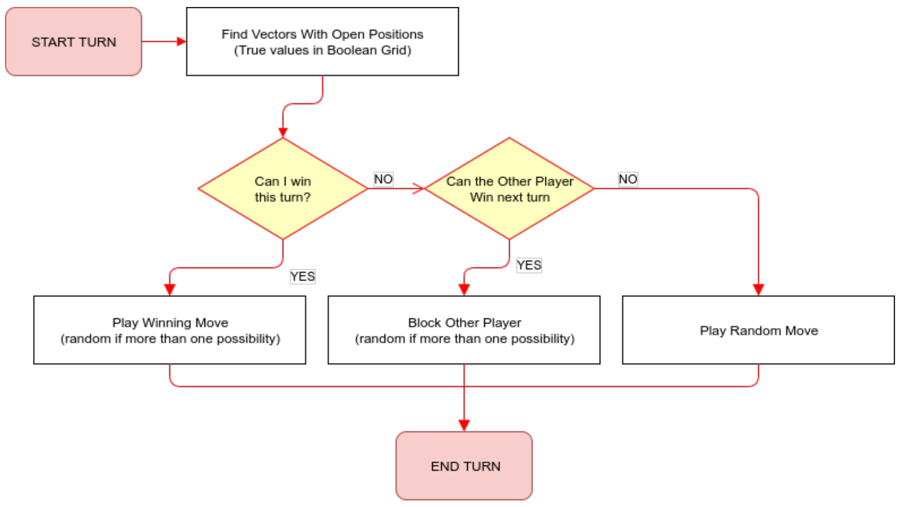
START TURN
Find Vectors With Open Positions
(True values in Boolean Grid)
Can I win this turn?
NO
Can the Other Player Win next turn
NO
YES
YES
Play Winning Move
(random if more than one possibility)
Block Other Player
(random if more than one possibility)
Play Random Move
END TURN

Figure 2: Decision process of the AI with a hard coded strategy: SetAI.

2.3 LearningAI

The LearningAI, which was the focus of this work, used a pre-trained Convolutional Neural Network to estimate the best move at hand. In this section, we focus on how the AI used the model to make decisions. Details about the model will be presented in Section 4.

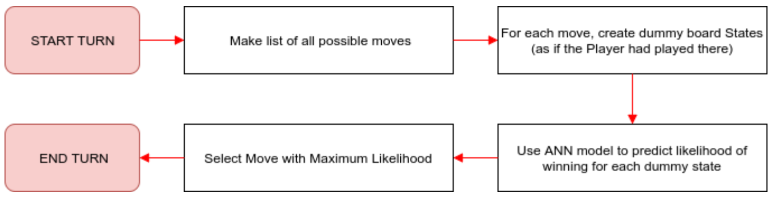
START TURN
Make list of all possible moves
For each move, create dummy board States (as if the Player had played there)
Use ANN model to predict likelihood of winning for each dummy state
Select Move with Maximum Likelihood
END TURN

Figure 3: Decision process of AI using a trained neural network : `LearningAI`

The basic flow of how the `LearningAI` works is illustrated in Figure 3. Effectively, we are using the CNN to predict the likelihood of winning for each available move and selecting the best one. However, it is important to note that the predictions are not actual probabilities or likelihoods. Rather, the model acts more like a regression model predicting values between 0 and 1. Zero meaning : *you have lost*, and one being : *you have won*. The reason for this is because we have only trained the model to classify winning and losing games. Details of the training are provided in Section 4.

The `LearningAI` starts its turn by creating 7 temporary game grids each representing what the grid would look like if it chose to play in a given position (there could be less temporary grids depending on the number of available moves). These grids are fed into the CNN which classifies them as either winning or losing. The difference with actual classification is that we are not converting the output to binary values, instead we are interpreting the numbers (between 0 and 1) as likelihoods of winning. From this the AI chooses the column with the value closest to 1.

2.4 The `Board()` Object

The `Board` object, as the name states, keeps track of where player pieces are on the board (or grid), which moves are available, and can run a check on whether or not a player has won. The C4 grid is represented by a 6×7 array where each element can take one of the following values :

- 0 for empty spaces
- 1 for Player 1 pieces
- -1 for Player 2 pieces

This array is refered to as the *game grid*. In addition to the game grid, the `Board()` object also defines two grids of the same shape : the *boolean grid* and the *column grid*. The boolean grid is used to keep track of which moves are available by assigning `True` to elements where a player can place a piece, and `False` everywhere else. It is also used by the `SetAI` when making decisions about its next move. The column grid, is simply an array where each element holds its corresponding column index. Albeit it is not necessary, it was implemented as a reference table to easily look up column numbers when dealing with diagonals. This should make more sense shortly.

The last important feature of the `Board()` object is what we refer to as the *vectors*. We define three types of vectors: *grid vectors*, *boolean vectors*, and *column vectors*. Each of these vectors are 4 element arrays that present a view of their associated grid (see Figure 4), meaning the contained values always match the ones we see in the grids.

Grid vectors are mainly used to see if a player has won. Because each player is marked on the grid by either 1 or -1, we simply have to sum the elements in a grid vector to check for a winner. For example:

$$\texttt{Sum}\left([1,1,1,1]\right) = 4 \qquad \text{vs} \qquad \texttt{Sum}\left([1,0,1,1]\right) = 3$$

In this example, the first vector only contains 1's which is the only possible way to sum to 4 (since we only have 1, 0, or -1), hence we know that Player 1 has won the game. This method is ultimately more efficient that searching for the four follwing elements in a given direction and checking their values. By searching for grid vectors that sum to 3 (or -3), the `SetAI` can also identify winning or losing vectors.

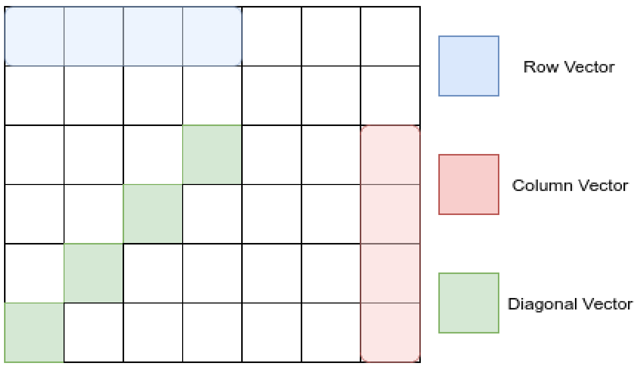

Figure 4: Representation of the three ways a player can win, refered to as *vectors*. For a 6×7 grid there are : 24 *Row vectors*, 18 *Column Vectors*, and 24 *Diagonal Vectors*. Each vector is a view of those 4 elements on the board, and are kept in a list to easily check whether or not a player has won. They are also used in the decision process of the `SetAI`.

The boolean vectors and column vectors are used jointly by the `SetAI` when identifying winning or losing vectors. Once such a vector is identified, the column corresponding to the winning move is found by matching the index of the `True` value in the boolean vector and looking up the column number in the column vector. Which is the whole reason why we have defined column vectors in the first place. The boolean vectors are also used to filter out non-playable vectors (only `False` values).

3 Generating Data

ANNs boil down to adjusting weights in a series of seemingly random calculations until the error on the predictions is minimized. Because of this they require a large amount of data to perform well. Thankfully, there is an extremely large number of possible connect 4 games. To get an estimate, we know that each position in the grid can be in one of three states (empty, Player 1, Player 2), and there are 42 positions in a 6×7 grid. Therefore, there are : $3^{42} \sim 10^{20}$ permutations. Of course, not all permutations are valid games, but this gives us an idea of what we are dealing with.

Ideally, we would train the neural network on all possible games so that there could be no surprises. Due to the computational barriers of using so much data, and the complexity algorithmically figuring out all possible games, we have opted to generate data by having two AIs play against each other. At first this meant two instances of the `SetAI`, but every game we have played while testing models has been saved to a data base. Because of the large number of permutations, it is unlikely that two games will be the same, but just in case we have periodically filtered out repeated games.

As mentioned in Section 2, the Connect-4 Engine provides us with some functionality to save a game after it has been played. These games are saved as *csv* with 43 values : 42 values are used for the end result of the grid, and 1 value to indicate the winner. This is illustrated in Figure 3, but effectively we are flattening the game grid and appending the winner's marker (1, -1, or 0 for ties).

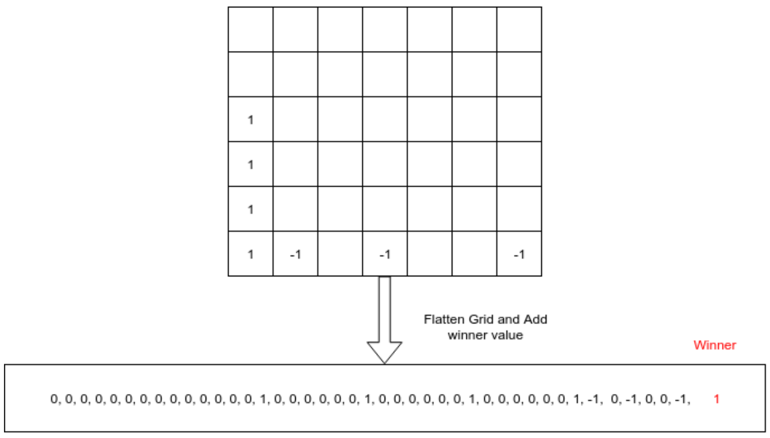

Figure 5: Diagram showing how the end grid of a game is transformed to be saved as a row in a `.csv` file.

With the set up in place, games were played on a loop until we generated about 300,000 games. Far from the number of possibilities, but enough to fit a model relatively quickly and have it perform moderately well.

4 The Model

Now for the fun part : with everything in place we are finally able to train an ANN to play Connect-4. To recap we have :

- A database of over 300,000 unique Connect-4 games
- A C4 engine with different types of players
- An AI with a hard coded strategy, `SetAI`
- An AI that uses a pre-trained CNN to make decisions, `LearningAI`
- An AI to define a baseline performance, `RandomAI`

All that is left to do is to actually make a model and train an ANN. The underlying idea is that by training an ANN to correctly identify the winner of each game in the database we can use that information to identify moves that are mostly likely to lead to victory. For this we have tested a variety of topographies and found that Convolutional Neural Networks work best. Because CNNs are meant to process images, a little preprocessing is required.

4.1 Preprocessing

Since we have generated our own data the preprocessing is minimal, all we have to do is reshape the data to mimic an image. After loading, we separate the data into features (the state of the grid) and target variable (the winner). We then reshape each row into a (6, 7, 1) array, where (6,7) is to recreate the grid (imagine Figure 3 in reverse). The extra dimension (1) is to imitate a color chanel, but contains no additional information. With this shape, the data is able to fit into a Convolutional layer in a CNN.

The second (and last) step in our preprocessing scheme is to binarize the target variable. In our model, we are not actually predicting who won, but instead using a *one vs all* scheme to identify whether or not Player 1 has won. This step is boils down changing all -1 values in the target column to zeros.

4.2 Topography

Seeking out the right topography for any Neural Network often requires a lot of trial and error. Because this is essentially a binary image classification problem we already know what our input and output layers should be. The input layer is the convolutional layer, and the output layer is a single node with a sigmoid activation function. Output values above 0.5 are classified as 1 (Player 1 wins), and others as 0 (Player 2 wins or a tie). For the hidden layers, we attempted many different topographies, which were mostly successful in predicting the target value. However, being able to correctly classify the data does not mean the model does a good job at winning. Nonetheless we used the accuracy of our model as a first estimate of its value. In the end the model we found played the best was as follows:

1. **Input Layer :** 2D Convolution with
   - a $(4, 4)$ filter
   - strides of $(1, 1)$
   - a *hyperbolic tangent* activation function
   - 42 Outputs
2. **Hidden Layer 1:** MaxPooling
   - pool size = 2
3. **Hidden Layer 2:** Dense Layer
   - 7 Nodes
   - simgoid activation function

4. **Output Layer :** Dense Layer with
    - 1 Neuron
    - a *sigmoid* activation function

The model was trained on a 70-30 train test split over 100 epochs with a batch size of 1000.

The choice of a 4 × 4 filter and *tanh* activation in the input layer was made to get a first evaluation of the Players likelihood of winning. Meaning if the ouput of that layer leans more towards -1 it should mean that Player 2 is in better position for that section of the grid. Note that the `LearningAI` was always assigned to Player 1. To be clear, this does not mean that the `LearningAI` always went first, it only affects his marker on the game grid (1 or -1) , and the target value when fitting the model.

Figure 4.2 shows the accuracy of predicting if Player 1 won as function of the number of epochs for both the training and testing set. When identifying if Player 1 won, this model led to about 99% accuracy. Of course, accuracy alone does not mean anything, if Player 1 won 99/100 games and we predicted that he won 100/100, we would still get 99% accuracy. To put our mind at ease, we refer to the confusion matrix presented in Table 1. This table shows us the predictions from the test set (112992 games). In our case, a `True` value in the confusion matrix means that Player 1 has won. Out of the 58,687 `True` values, our model only missed 143 of them. Conversly, out of the 54305 `False` values, only 114 were misclassified. The over all accuracy is therefore ≈ 0.998. This means that our model has a pretty good idea of what it means to win a Connect-4 game, and all this without explicitly telling it (granted there are many examples). However, as we will see in the next section accuracy on predicting the winner does not directly translate to the `LearningAI` being able to play.

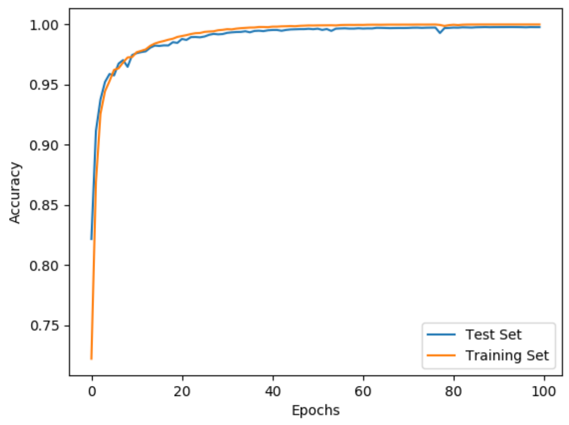

Figure 6: Accuracy of predictions on training and testing sets as a function of the number of epochs when fitting the neural network.

As a side note, we notice in Figure 4.2, that the accuracies of the training and testing set never diverge. This would be unusual in a more chaotic set of data, but do to the consistency of values in the context of Connect-4, this is not so suprising. Possible values for all features are either 1, 0, or -1, so there are no outliers, no null values, and no randomness to the data.

$N_{\text{games}} = 112,992$	Predicted True	Predicted False
Actual True	58659	143
Actual False	114	54076

Table 1: Confusion matrix of CNN predictions on test set. `True` values are Player 1 wins, `False` values are Player 2 wins or tie games.

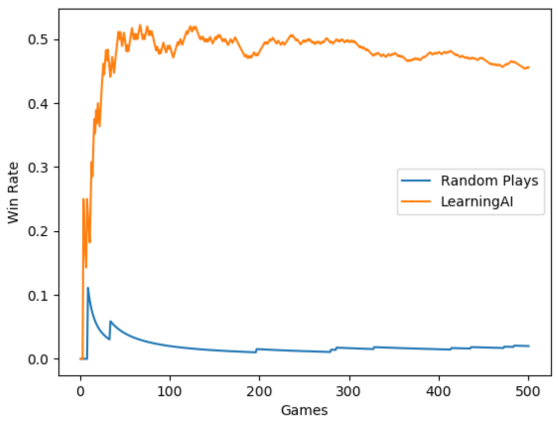

Figure 7: Performance test of `LearningAI` (orange) over 500 games against `SetAI`. Graph displays the win rate (Number of Wins / Number of Games) as a function of the number of games. The blue line represents the win rate for a player that only selects moves randomly. This is for assurance that the `LearningAI` is not winning by pure chance.

5 Performance

With a working model we can now have the `LearningAI` try its luck against the `SetAI`. To evaluate its performance we have the two AIs play 500 games against each other and measure the `LearningAI`'s win rate, W_r, defined as :

$$W_r = \frac{N_{\text{wins}}}{N_{\text{games}}}$$

where N_{wins} is the number of games won against `SetAI`. Figure 5 shows us this win rate as a function of the number of games. To make sure the model is not winning by random chance, Figure 5 also shows the win rate of the `RandomAI` when playing against the `SetAI`. For both these trials, we see that W_r starts to converge towards certain values. The `LearningAI` converges somewhere around 0.45, meaning it is winning 45% of the games. By comparison, placing pieces at random wins against the `SetAI` only about 2% of the time. From this comparison, we see that `LearningAI` has in fact learned how to play the game. Granted there is room for improvement, but this is good step forward.

6 Final Thoughts

In the context of Connect-4, we have seen that it is possible to create an AI using convolutional neural networks that picks up on patterns by example. The AI was never explicitly told how to play the game, and was able to make decisions to win about 45% of the time when playing against an AI with a hard coded strategy. With this in mind, we must think about how to make it win 100% of the time. Tweaking the model's parameters only seemed to help to some extent. The main flaw in this set-up is that the model was only trained to classify wins and losses in a database of complete games. At no point was it weighing the effects of a particular move. Our assumption that this type of training could directly translate to making in-game decisions was limited. Also, this work was mostly focused on whether or not an ANN could pick up on the rules and learn. Had we focused solely on making an AI that could win all the time, a better approach would have been a Monte Carlo model that directly measures the probability of winning with a certain move using the game database. That being said, future work will involve implementing two new types of players. One player will be the Monte Carlo approach to the problem, while the other will still revolve around ANN's. The idea behind the new ANN will be based on reinforcement learning. The main differences with this new idea are that instead of an ouput layer with only 1 node, it will have 7 nodes representing the player's choices. With each action, or completed game, the CNN will adjust its weights to attempt a better outcome the next time around. This type of model should be able to start from scratch (without the database), and progressively learn everything on its own. That being said, this implementation is a good start!