Group 18: A Reinforcement Learning and Machine Learning Approach to Connect Four

Tudor Tibu
Department of Computer Science
Ryerson University
Toronto, Canada
tudor.tibu@ryerson.ca

Alexander Yaroslavtsev
Department of Computer Science
Ryerson Univerity
Toronto Canada
ayarosla@ryerson.ca

I. Introduction

For our course project, we have decided to focus on the game of Connect Four. Connect Four is a two player strategy board game where each player tries to get four of their colored pieces (also called discs) in a row while trying to prevent the opponent, who would be using a set of differently colored pieces, from doing such themselves. It is played on a 6 tall by 7 across vertical slot board, with each piece inserted into one of the 7 columns and falling down to the lowest available slot within the column. The two players take turns making moves until someone gets four of their pieces in a row and wins, or the game ends in a draw.

The game of Connect Four has been researched quite extensively. The game was first solved back in the year of 1988 by two independent researchers James D. Allen and Victor Allis. Allen provided extensive analysis in his publication "Expert Play in Connect-Four", deconstructing the game into discrete scenarios and how to go about them [1] while Allis presented a knowledge-based solution to the game [2]. Since then, there have been many different approaches to solving the game, including the application of algorithms such as Q-Learning, MiniMax, Alpha-Beta pruning, Monte Carlo Tree Search, and many others

II. Problem Statement

Our original goal was to explore many of these different methods to evaluate their performance and metrics in relation to one another, but this was simplified down to the exploration of a few select methods & evaluation of their own respective strengths and weaknesses (due to the withdrawal of two members from the project). The first explored method consisted of a deep Q-Learning approach, experimenting primarily with different configurations of deep neural networks (DNNs). The second method explored was creating a classifier that could identify if a board state was winning or losing for the player going first. We wanted to explore this approach because of the nature that Connect Four is played, once a piece is played that piece cannot be removed or changed (i.e., pieces are static) unlike a game like chess or go. A game of Connect 4 can be visualized as an adirectional, acyclical tree. Assuming at least one player is playing optimally, once a player makes a move that is winning they will always be winning and vice versa for player making a losing move. Therefore, if we can create a model that can classify winning board positions, and create an agent that searches when playing the game, we can theoretically, create a Connect Four agent that can explore game options with prior knowledge of the game, and therefore speed up learning by reducing the number of episodes needed.

III. Datasets

2 different datasets were obtained online, one from UCI.edu [3] and another one obtained from Kaggle.com [4]

A. *UCI Dataset (Dataset 1)*

This dataset contains every board position in Connect Four eight moves into the game and the result of the game assuming both players are playing optimally. Another condition for the dataset is that neither player has already won the game and the next move is not forced (e.g., neither player 1 or 2 has 3 in a row). This dataset was created and mathematically proven by John Tromp, a Dutch computer scientist [5]. Classes in the dataset are labeled as either {win, loss, draw} and the feature vector contains 42 features, each one representing a position on the board. The feature vector is organized column-wise (x_0 is the first row, first column, x_1 is the second row first column on the board). The feature vector is labeled as {x, o, b} x representing player 1's pieces, o player 2's and b representing unoccupied positions. The data set contains 67,558 entries, of these entries 44,474 (65.8%) are classed as win, 16,636 (24.6%) are classed as loss and 6,450 (9.6%) are classed as draw.

Before any of this data was used for training or testing, feature labels were changed from {x, o, b} to {1, -1, 0}, class labels were changed from {win, loss, draw} to one hot encoding and the dataset was separated into 3 different files one for each class.

B. *Kaggle Dataset (Dataset 2)*

This dataset contains some possible board state after the game as concluded. Classes in the data set are labeled either {1, -1, 0} 1 for a win, -1 for a loss, 0 for a draw and the feature vector contains 42 features, each one representing a position on the board. It is organized row wise (x_0 is the bottom row, first column, x_1 is bottom row, second column). The feature vector is labeled as {1, -1, 0} 1 represents player 1's position, -1 represents player 2's position and 0 represent unoccupied positions. The data set contains 376,620 entries, of these entries 181,256 (48.1%) are classed as 1, 180,868 (48.0%) are classed as -1, 14,498 (3.9%) are classed as 0.

Before any of this data was used for training or testing, feature labels were changed to column wise orientation, class labels were changed to one hot encoding and the dataset was separated into 3 different files, one for each class.

IV. Libraries and tools

We primarily used TensorFlow and Keras for our model implementations. The sequential model was used as the

baseline for the Q-Learning approach, as well as the model based approaches.

For initial experimentation, the logistic regression classifier provided in the course assignments was also used. As well as sklearn's implementation of the logistic regression, SVM and GBM classifiers.

V. Explored Methods

A. *Q-Learning*

Q-Learning is a model-free reinforcement learning (RL) algorithm. It is considered an off-policy technique as it can learn an optimal policy largely from exploration (i.e. random moves), independently from the policy being learned and the agent's actions. At its core, Q-Learning can be represented as a state to action-value table, with one cell for every state and action pair. As such, it is a value-based algorithm, as it attempts to learn the net reward values for subsequent action states from any given state, taking the action that maximizes the net reward. Using the Bellman equation value function to propagate rewards, Q-Learning can learn an optimal playing strategy with no prior knowledge of the environment.

In the context of the Connect Four game, there are no more than 7 possible actions at any given state (one for each column slot), but the total number of states for a vanilla 6x7 game board is gargantuan, exceeding 4 trillion legal states [2]. As it is infeasible to store and manage such a large table, the workaround approach is Deep Q-Learning (DQL), where a deep neural network (DNN) is used to estimate the action reward values for any given state.

Two main deep neural network (DNN) models were explored for this problem. The first was a simple fully-connected DNN with varying number of layers and neurons, and the second was a convolutional neural network (CNN) to leverage the spatial relationships of the environment. The initial input for both models was 42 features (one for each slot in the game board, depicting its value), and with a linear activation output of 7 values (one for each column, i.e. the estimated reward for that column action). The ReLU activation function was used for all intermediate layers, Adam was chosen as the optimizer, and mean squared error (MSE) was used as the loss function. The rest of the setup of both model types varied for each testing instance.

Reinforcement learning has a set of fixed variables called hyperparameters. The first of which is epsilon, which signifies exploration rate. For each agent & model, the epsilon value was evaluated as:

$$\epsilon_x = \max(0.995^x, 0.05) \tag{1}$$

Where x is the current episode number. This decay works well to help the agent explore the game board early on, but primarily focus on exploitation once it has good knowledge of the environment. This epsilon function produces the following curve.

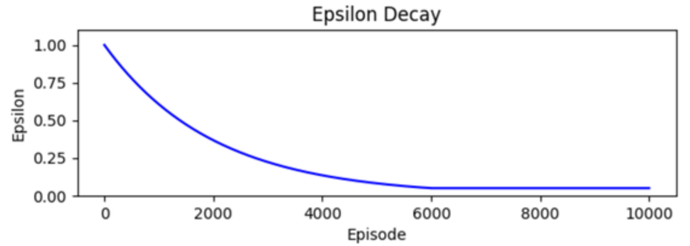

Fig 1. Epsilon decay curve used for all DQL models

Gamma, which is the discount factor used in the Bellman equation, was set between 0.9 to 0.95. The reasoning behind such a high discount factor is its desired effect of reward propagation throughout the board, making prospective rewards in future states easier to notice ahead of time. Low gamma values result in reward short-sightedness, which is not preferable.

Another common technique for DQL models is experience replay. The idea behind this technique is the storing of state transitions and their generated rewards in a memory/feedback buffer as training samples, and then selecting a random batch from this buffer during each training iteration. In contrast to sequential sampling, this approach breaks any high correlation between samples, which tends to result in more efficient learning [6]. This standard technique was used to train the models, with varying sizes of the replay buffer.

To evaluate each model, a fixed set of metrics was used. This included the average loss, win/draw/loss rate, error/invalid move rate (counted each time the agent tried to choose an inaccessible column), average move count, and average episode reward. These metrics, averaged over finite episode intervals, proved sufficient enough to deduce strengths and weaknesses of each configuration, and adequately allowed for comparison between the various tested models and setups.

B. *Model Based Approach*

Since the DQL approach requires a model to estimate action reward values for any given state, we believe that creating a model from existing data that can classify whether a board that is winning or losing would reducing the amount of training required for a DQL network to learn.

We decided to compare how effective both a multi-layer perceptron (MLP) and a convolutional neural network (CNN) at classifying unfinished games. We chose a CNN because of the spatial nature of the Connect Four board and because the feature vector of our data is small, we believed that a MLP could also be very effective and efficient. To evaluate each of these models, we decided that accuracy would be the best metric to compare the model's effectiveness, and since we would be working with multiclass data creating confusion matrices of our data would let us know which classes the models were having trouble identifying.

Since we were working with 2 separate datasets we decided to start with simple classifiers, since they are easier and faster to train. By training and testing with out simple classifiers we can see which combination of data is most effective for comparing MLP and CNN structure. Ideally, we would want data that creates the least confusion in our results.

1) Data Splitting and Simple Classifier Tests. For training our simple classifiers we decided to evaluate the following methods: 1) Training on incomplete games and testing on incomplete games. 2) Training on incomplete games and testing on complete games 3) Training on complete games and testing on incomplete games. We chose logistic regression (logreg), state vector machines (SVM) and gradient boosting machine (GBM) for our simple classifiers. These were chosen because these algorithms are fast and are easy to implement.

As for splitting our data for each method, all training data had 10,000 samples and test data had 5,000 samples. Ratios between classes were always equal regardless of the method we were experimenting, for example, if we were training/testing a model that included the class labels {draw, win, loss}, their respective ratios would be {1/3, 1/3, 1/3} for both testing and training sets. If we were training a model that only included the class labels {win, loss} their respective ratios would be {1/2, 1/2}.

VI. Results and Observations

A. Q-Learning

The initial three tests consisted of a simple fully-connected DNN with around 5 dense layers of 42 neurons each. These tests were conducted to evaluate different learning rates, with the agent playing against a random adversary – one that picks actions at random. The lowest attempted learning rate, 1e-5, gave good results and allowed the agent to learn an optimal strategy fairly quickly. The next learning rate, 1e-4, gave similar results, but the various metrics were less stable and slightly more variant than those from 1e-5, but the agent learned an optimal strategy a bit faster. The last test used the largest learning rate of 1e-3, and the agent managed to learn an optimal strategy the quickest, but with the most instability (as each reward affected the model considerably). This final test also succumbed to the problem of not using a sufficiently large enough replay buffer, causing the model to erroneously adapt to a poorer strategy after 20k episodes and never recover (due to a low epsilon value past 6k episodes). As a result of these tests, 1e-4 was used as the baseline learning rate.

The next phase of testing shifted to a 2D convolutional approach, using the same problem of random adversarial play but with a CNN model. The model was constructed with a single 2D convolutional layer, followed by a few dense layers. For the first model, 2x2 max pooling was also added, but was not found to be helpful. This first model performed considerably better than its DNN counterparts, reaching a stable 90% win-rate average after around 40,000 episodes. The even-shaped 4x4 filter size did not seem to negatively affect the learning ability very much, despite the common practice to avoid odd-sized filters due to distortions [7]. However, to address this notion, another CNN model was tested with a 5x5 filter instead, and this showed an improvement to the previous model, with a 95% win-rate at 40k episodes. For this latter model, the invalid rate was also considerably lower (Figure 2).

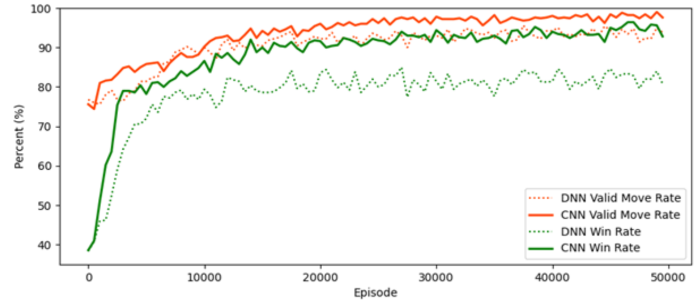

Fig 2. Performance comparison between simple dense DNN model and a CNN model, both using a learning rate of 1e-5, averaged by groupings of 500 episodes.

With the established data from the previous tests, the last phase of testing attempted to teach an agent how to play against a smarter adversary. This adversary would place a winning move whenever possible, and avoid/block any available winning moves for the agent. This forced the agent to play more defensively, and search for states with several winning move options. The most effective model was a CNN using a single 5x5 filter 2D convolutional layer with 256 outputs, 1e-4 learning rate, 10% spatial dropout, and 3 additional dense layers. Using this model, after 50k episodes the agent was able to cleverly win around 35% of the time, with a relatively low invalid rate of 15% (see Figure 3). The other two training models, one being another CNN with a lower learning rate, and the other a regular DNN, the results were not as impressive, with less than 25% win-rate at 50k episodes.

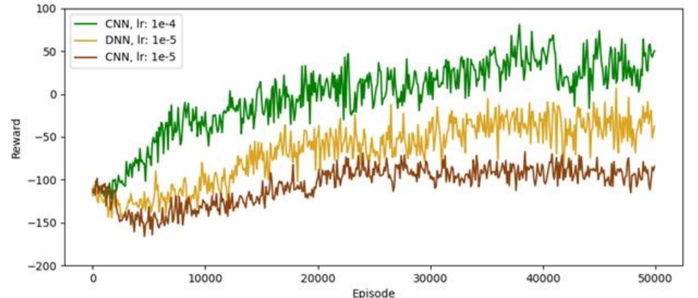

Fig 3. Reward comparison between three similar models, averaged by groupings of 100 episodes. The model with the smaller learning rate gave the best results among these training runs.

CNN models were found to be an effective method for this deep Q-Learning approach. However, this approach would likely require a very high number of training episodes in order to "master" the game board through DQL, thus is likely not the best for solving the game in an efficient manner. For simple tasks however, like playing against a random adversary, this approach is fairly effective, as the complexity of blocking opponent wins and relying on multiple win strategies is practically eliminated.

To continue this approach going forward, the training can be run for a 50k+ more episodes. If no substantial progress is

made, increasing the learning rate further may be beneficial, as long as the replay buffer is made sufficiently large such that significant rewards are not excessively sampled (as this could give the model an erroneous bias). A better reward function should also theoretically help this approach for harder adversary play, as our implementation was limited to end-game rewards, making it more difficult for the agent to learn on such relatively rare stimulations, given that the smarter adversary makes it very difficult for wins to be discovered. Using a complementary model that can estimate win chance for reward distribution may be a promising addition. For efficiency, this approach would also need to be migrated to run on the GPU, as the standard implementation of Q-Learning is largely sequential.

B. *Simple Classifiers*

1) Training Simple Classifiers with Incomplete Games and Testing on Incomplete Games. First training tests that we did was training non neural network classifiers. We chose this approach to see how the classifiers behaved working with this dataset and to see if we needed more data. From this test max accuracy was below 70% (Figure 5) which we found unacceptable. Creating a confusion matrix (Figure 4) of the results showed that the simple classifiers had a difficult time classifying draws correctly, with the highest accuracy being 69% and the lowest being 33%.

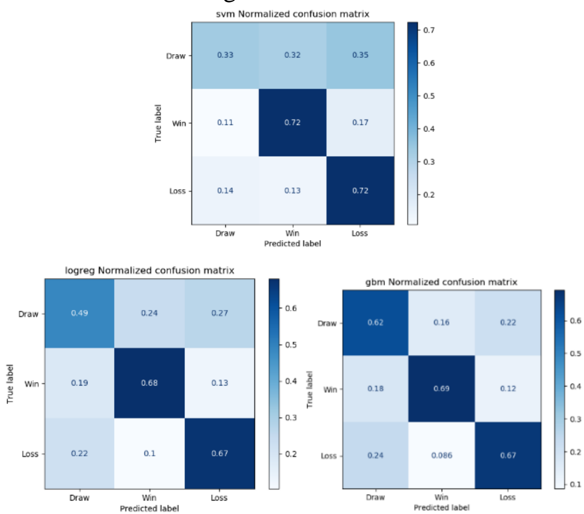

Fig 4. SVM, LogReg and GBM confusion matrix for trained and Tested on Dataset 1

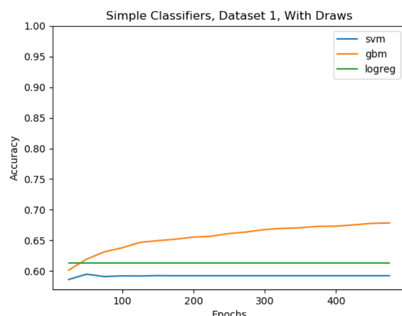

Fig 5. Simple Classifier Accuracy, Trained and Tested on Dataset 1, with Draws

Since training our data with draws created a lot of confusion, we wanted to see how the algorithms performed on classifying wins or losses. We expected that overall accuracy would increase by 10-20% (the average of falsely predicting a draw for a win or loss). While our prediction was semi true, and that overall accuracy did increase (Figure 6), we noticed that number of false positives for win and loss labels also increased, for some cases it was insignificant margins (<5%) in other cases it was as high as 10% (Figure 7). As a result draw cases were excluded for future models.

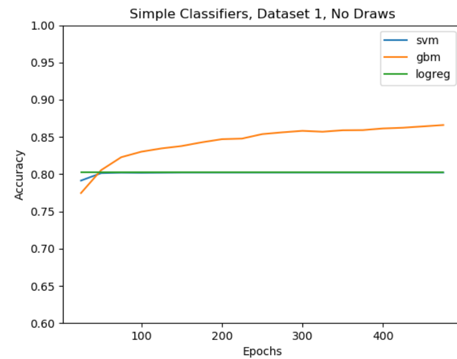

Fig 6. Simple Classifier Accuracy, Trained and Test on Dataset 1, no draws

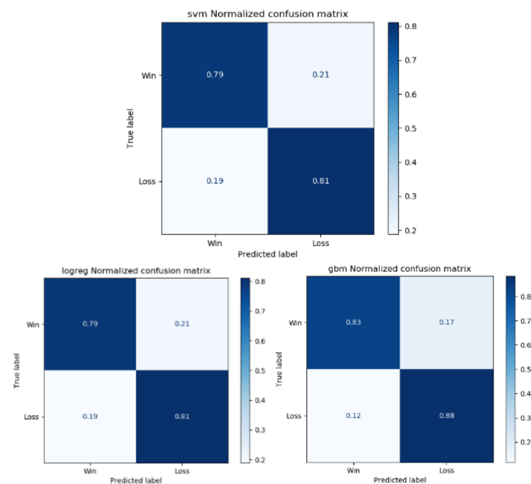

Fig 7. SVM, LogReg and GBM confusion matrix for Trained and Tested on Dataset 1, no draws

2) Training Simple Classifiers with Complete Games including Testing on Complete Games. Since Dataset 1 only contained 8-ply moves, we believe that our algorithms would have a difficult time determining if a board state is winning or losing if it contained a configuration that has more that eight pieces on the board. Since Dataset 2 contained positions that were more than 8-ply we created this test to see how it would perform. We noticed our algorithms performed extremely poorly in this scenario, performing below 50% accuracy (Figure 8).

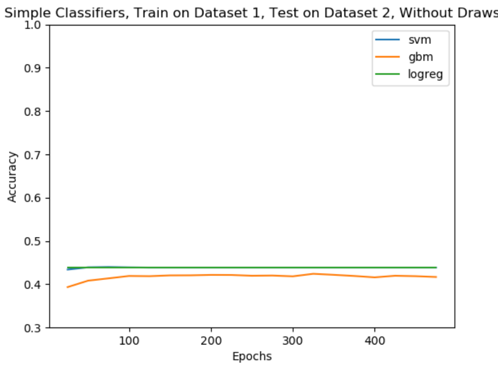

Fig 8. Simple Classifiers Accuracy Trained on Incompleted games, Tested on Complete Games

Given the results of our last test we believed that training the models on complete games would improve the overall accuracy since the data is more diverse data points to learn from. Training and testing on complete game resulted in very high accuracies, higher than previous examples (Figure 9). Therefore, when calculating which neural network setups would be best to train Connect Four boards on, we will be training our examples on completed games and tested them on complete games.

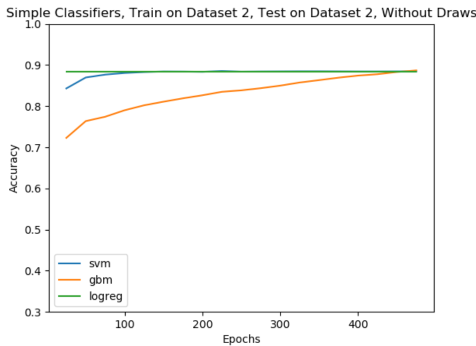

Fig 9. Simple Classifier Accuracy, Trained and Test on Dataset 2

C. Creating Neural Networks

For out network structures, we decided the best approach was to use cross entropy loss and an Adam optimizer. We compared different network structures by training them with completed games and testing them on completed games.

1) CNN Structure. 9 different CNN setups were considered (Figure 10), ranging from the amount and size of fully connected layers amount and size of convolution filters, and we found that the best performing setup was the following. The first layer is a reshape layer, it reshapes the 1x42 input into a 6x7 for the convolution. Second layer is a 2D convolution with 256 filters and a filter size of 5x5. Third layer was a flatten layer. The fourth and fifth layer were 2 fully connected layers with over 200 outputs. Sixth layer was a fully connected layer with ReLU activation and an output size of 42. Seventh layer was a dropout layer with a rate of 0.1. The output layer was fully connected layer with softmax activation and an output of 3 (1 for each class).

Notice that, networks 1-5 has a convolution filter of 3x3, network 6 had a 4x4 filter and networks 6-9 had a convolution filter of size 5x5. The most significant improvement was increasing the filter size to 5x5, this is most likely since the 5x5 filter is large enough to filter board sequences that are winning, (i.e. 4 in a row opportunities which the 3x3 filter is not larger enough to filter). Max pooling (Network 5) and using multiple convolution layers (Networks 4 & 5) had no effects.

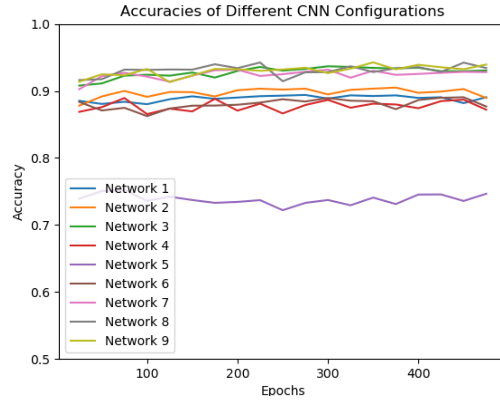

Fig 10. Accuracy Comparison of 9 different CNN Structures

2) MLP Structure. 7 Different MLP setups were considered (Figure 11), with ranging from wide networks with few layers but more parameters (Networks 1-4) to more deep networks with fewer parameters (Networks 5-7). We found the best performing setup was a wide network structure, based on average accuracy per epoch. It had 3 large dense layers (>200 outputs), 1 small dense layer (<50 outputs), 1 dropout layer with a rate of 0.1 and the output layer with 3 outputs. All dense layers used ReLU activation and the output layer used softmax activation.

The accuracy spread between the setups with the highest average accuracy vs. lowest average accuracy was much closer than the spread for the different CNN setups.

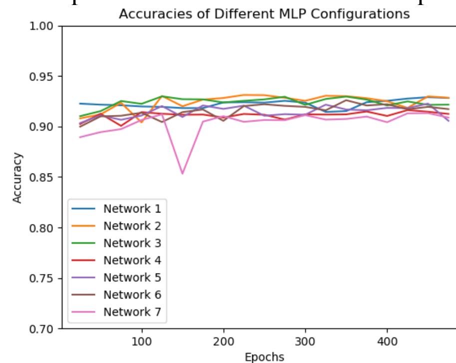

Fig 11. Accuracy Comparison of 7 different MLP Structures

D. Comparing NN classifiers with Simple Classifiers

1) Training with Complete Games, Testing on Complete Games The neural networks performed better than the simple classifiers (Figure 12) for this test. We found that the neural networks were better at determining if there was a draw in a finished game than the simple classifiers

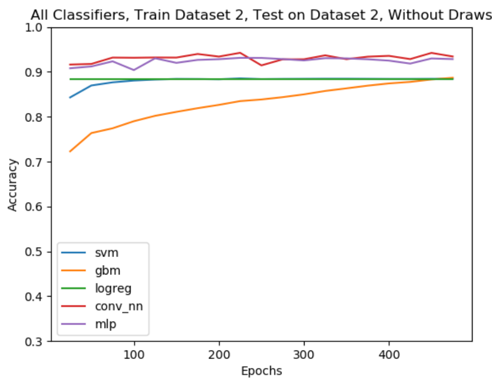

Fig 12. Comparing NNs vs Simple Classifiers, Training and Testing on Completed Games Data

2) Training with Complete Games, Testing on Incomplete Games. We found that the neural networks performed on par or worse than the simple classifiers when we trained it in on complete games data and testing it on incomplete game data (Figure 13).

Fig 13. Comparing NNs vs Simple Classifiers, Training on Completed Games Data and Testing on Incomplete Game Data

We speculate that the cause for this poor result is either due to overfitting, not enough diversity in the training data (i.e. since the training data used finished game states with mostly filled boards, it improperly knows how to predict emptier board states) or our complete game data set did not contain enough examples of perfect play.

3) Training on Mixed Data, Testing on Incomplete Games. For this test, we took approximately 2,000 samples of incomplete game data and added it to our complete data set (removing 2,000 samples that were originally there). We found that our neural networks performed a lot better (Figure 14) than on the previous example. This supports our previous hypothesis that the number of blank board states and our data containing non perfect play since our models were able to adjust and perform even better than the simple classifiers.

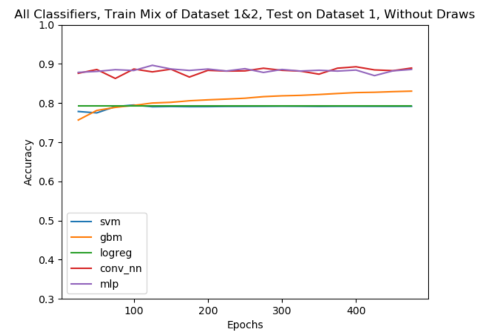

Fig 14. Comparing NNs vs Simple Classifiers, Training on Completed Games Data and Testing on Incomplete Game Data

E. Next Steps for the ML approach to this problem

We consider this part of the approach unsatisfactory, since our goal was to create a classifier that can test to see if a board state is winning or losing by only training it with finished games. However, we see there were some minor successes, training the data on the mixed data set gave valuable insight on improving this approach to the problem. Should this project continue, an ideal next step would be to identify the error that caused poor results. Going off earlier speculations, possible solutions include 1) Using deeper networks not wider ones, 2) Diversifying our data used 3) Adjusting our model to better identify key configuration and reduce the affect blank spaces have on the prediction. Using a deeper network with fewer parameters in some cases was shown to reduce the amount of overfitting and improve generalizability [8]. By diversifying the amount of data we are training on, our model can learn from different examples. Finding a dataset that includes more than just 8-ply moves of perfect play position, we would be able to evaluate how our models compare in board positions with more pieces.

REFERNECES

Code available at https://github.com/tudortibu/connect4

[1] James D. Allen, *The Complete Book of Connect 4*. Puzzlewright, 1990.

[2] V. Allis, "A Knowledge-Based Approach of Connect-Four: The Game Is Solved: White Wins," *ICG*, vol. 11, no. 4, pp. 165–165, Dec. 1988, doi: 10.3233/ICG-1988-11410.

[3] "UCI Machine Learning Repository: Connect-4 Data Set." http://archive.ics.uci.edu/ml/datasets/connect-4 (accessed Dec. 07, 2020).

[4] "Connect-4 Game Dataset." https://kaggle.com/tbrewer/connect-4 (accessed Dec. 15, 2020).

[5] "John's Connect Four Playground." https://tromp.github.io/c4/c4.html (accessed Dec. 15, 2020).

[6] "Connect Four - Deep Reinforcement Learning," *Stefan Voigt*, Apr. 01, 2020. /post/connectx/ (accessed Dec. 15, 2020).

[7] S. Sahoo, "Deciding optimal filter size for CNNs," *Medium*, Nov. 29, 2018. https://towardsdatascience.com/deciding-optimal-filter-size-for-cnns-d6f7b56f9363 (accessed Dec. 15, 2020).

[8] R. Eldan and O. Shamir, "The Power of Depth for Feedforward Neural Networks," *arXiv:1512.03965 [cs, stat]*, May 2016, Accessed: Dec. 14, 2020. [Online]. Available: http://arxiv.org/abs/1512.03965.