Better move-ordering in Combinatorial Games via Learnable Heuristics in a 1D Clobber Solver

Akash Saravanan
Dept. of Computing Science
University of Alberta
asaravan@ualberta.ca

Debraj Ray
Dept. of Computing Science
University of Alberta
debraj1@ualberta.ca

Abstract

Move-ordering functions have been the subject of much study in combinatorial games. Given that most move-ordering algorithms are based on heuristics and expert knowledge, we propose the use of neural networks as learnable move-ordering heuristics due to their extensive use as universal function approximators. Specifically, we demonstrate two different methods of training neural networks to act as move-ordering systems on the one-dimensional combinatorial game of Clobber. We compare these methods against two baselines and demonstrate the effectiveness of our learnable heuristics for this task. Our results show that neural networks show a lot of promise as a move-ordering system.

1 Introduction

Combinatorial games, that is, two-player games which are deterministic and offer perfect information to both players, are considered games of pure strategy. The general objective in such games is to identify an effective strategy capable of beating a perfect opponent. These combinatorial games have been the subject of much study, with several techniques developed to assist in solving these games. A game is considered to be solved if, for any given board position, we can determine the outcome of the game (win, lose or draw) assuming perfect play on both sides. One such combinatorial game is Clobber [Albert et al., 2005].

The rules of Clobber are fairly straightforward. The board consists of a one or two-dimensional checkerboard grid with either black or white stones placed on each grid cell. The starting position of the board is generally a sequence of alternating black and white stones across the grid, that is, each black stone is adjacent to a white stone in each cardinal direction and vice-versa. Each player picks one of the two colors and can only use stones of that color to make a move. During a player's turn, they must pick up a stone and "clobber" an adjacent stone of the opposite color. This results in the clobbered stone being replaced by the stone which was used to clobber it, leaving an empty grid cell in its original position. Play continues in a turn-based manner until no moves are left for the current player. At this point, the last player to make a move is declared the winner. That is, Clobber is a normal-play game. Additionally, Clobber is also a partisan game, meaning the two players have a different set of moves available for any given board position (since each player controls a single color). Although Clobber is usually played on a two-dimensional board, for the purposes of our work we consider a one-dimensional board [Albert et al., 2005, Wojciech et al., 2011, Claessen, 2011, Uiterwijk and Griebel, 2017, Etoeharnowo, 2017].

Move-ordering is an important aspect of combinatorial solvers as it greatly affects the number of nodes to be searched in the future. While there have been several attempts to improve move-ordering, research on leveraging machine learning techniques to improve these algorithms is fairly limited. In this research project, we explore strong move-ordering techniques based on learned insights from past

CMPUT 655 - Algorithms in Combinatorial Game Theory (Winter 2022).

plays. More specifically, we examine two powerful machine learning strategies in deep Convolutional Neural Networks (CNNs) and Reinforcement Learning (RL).

Our contributions are as follows:

- We propose the use of neural networks as a move-ordering heuristic for combinatorial games.
- We demonstrate two distinct approaches towards training these neural networks - using deep CNNs and using deep reinforcement learning.
- We evaluate these move-ordering heuristics on the one-dimensional combinatorial game of Clobber.
- Our results demonstrate the effectiveness of these methods in comparison to our baselines.

2 Background

2.1 Convolutional Neural Networks (CNNs)

CNNs are deep learning models inspired by human visual perception. They are primarily used for pattern recognition in images. A convolutional neural network has a hierarchical structure, with each layer computed using the formula: $x_j = \rho(W_j \cdot x_{j-1})$. Typically $W_j \cdot x_{j-1}$ represents the transformation of an input x_{j-1} via a convolution with weights W_j and ρ is a non-linear rectifier $max(x, 0)$ or sigmoid [Koushik, 2016]. The optimization is performed with gradient descent using backpropagation. CNNs are made up of three distinct types of layers - the convolutional layers, the pooling layers, and the fully connected layers [O'Shea and Nash, 2015]. The convolutional layers comprise a set of small learnable filters that slide over the entire image. These filters intuitively mean that the neurons would trigger if they detect a visual feature such as an edge or a pattern. The pooling layer performs downsampling along the spatial dimensional thus reducing the number of parameters. Moreover, pooling ensures the learned representations are invariant to translations. The use of fully connected layers is the same as with artificial neural networks - to produce class scores for multi-class classification.

2.2 Reinforement Learning (RL)

Reinforcement learning can be thought of as a means for machine learning models to learn through experience. It is distinct from supervised learning which associates inputs with specific labels and it is also distinct from unsupervised learning where there are no labels at all. A reinforcement learning system consists of 5 components - an agent, a state space, an action space, rewards, and an environment. In the context of combinatorial games, the state would encapsulate all information currently available to the player such as the board position and the current player. All the possible moves at that particular state are the actions available, with the super-set of all moves being the action space. The agent receives a state from the environment, based on which it performs an action. The environment peforms this action on the current state to compute the next state as well as a reward associated with this transition. γ, known as the discount factor, is a value associated with the reward that determines how much an agent takes future rewards into consideration. The agent then uses this entire transition (state, action, next state, reward, γ) for its learning process. The collection of states, actions, rewards as well as the state transition probabilities form a Markov Decision Process (MDP), which is the basis on which most reinforcement learning algorithms work.

There are several different algorithms for reinforcement learning. One such algorithm is the Deep Q-Network (DQN) [Mnih et al., 2013]. Based on tabular Q-learning [Sutton and Barto, 2018] where the agent attempts to learn the values of actions in states, deep Q-learning replaces the table with a deep neural network. To improve training, they also make use of an experience replay buffer, which allows us to disconnect transitions and consider them outside the context of specific games. Double Deep Q-Networks (DDQN) [van Hasselt et al., 2016] introduces several optimizations for the DQN, including the use of a secondary target network with soft target updates to improve learning stability. The ultimate goal of these algorithms is to learn a strong policy, that is, to define what actions to take in each state.

2.3 Combinatorial Game Theory (CGT)

Previous research has demonstrated several techniques in quickly solving Clobber boards of varying size and complexity. Albert et al. [2005] introduced the game of Clobber and proposed several conjectures. One such proposal was that boards of the form BW^n are a first player win for $n \neq 3$. They verified this up to $n = 19$ by computer. Uiterwijk and Griebel [2017] showed that by combining an alpha-beta solver (NegaScout) with a subgame database in 2D Clobber, the number of nodes visited could be significantly reduced. Wojciech et al. [2011] studied four new heuristics for playing 2D Clobber. Lin [2019] primarily focused on space efficiency and attempted to build a strong endgame database of orthogonal groupings for Clobber. For example, symmetry and canonical values were considered to avoid storing redundant information. However, the authors acknowledged that the size of the game's canonical forms quickly exploded when non-trivial canonical form games were added together.

Machine learning has shown promise in solving combinatorial games in the past. A neural network architecture worked well for the game of Go when the board was segmented based on subgames [Enzenberger, 2003]. Moreover, with the advent of deep learning, research has shown that deep CNNs can learn strong evaluation functions even for Go having a state space as large as 10170 [Maddison et al., 2015]. Gao et al. [2018] showed that CNNs could capture the knowledge of the game of Hex quite well and could predict the correct move 55% times without any search. Moreover, when the CNN was used as a learned prior along with MCTS, the result was better than the state-of-the-art MCTS player MoHex 2.0. Claessen [2011] developed a CGT-based 2D Clobber solver and demonstrated how the use of Monte-Carlo Tree Search (MCTS) to handle CGT values of subgames resulted in a significant performance boost. Etoeharnowo [2017] compared different 2D Clobber algorithms including MCTS and neural networks, though they found that the MCTS player worked best in the majority of cases. Järleberg [2011] studied the game of Nim and verified that the Q-Learning algorithm converged to the optimal strategy. There have also been several works such as AlphaGo and AlphaStar that have resulted in expert-level players for games like Go [Silver et al., 2017, Vinyals et al., 2019].

3 Methodology

In this section we describe the core algorithms and methodology we use. We begin with a brief discussion of our baseline solver and move-ordering system, followed by an explanation of the deep CNN approach to move-ordering and finally the reinforcement learning move-ordering scheme.

3.1 Baseline (Assignment 2 Solver)

The baseline Clobber solver is purely based on CGT and transposition table. For example, it removes zero sum games and certain patterns which are second player win. It also uses a subgame database for deciding game outcome based on outcome classes (L, R, N, P) of subgames. The CGT based solver also uses a move ordering based on subgame classes - R subgames played first, L subgames played last and the remaining subgames are played in the order of their size from largest to smallest. This is what we consider as the default move ordering (baseline) for comparison with the machine learning based move ordering. We have observed that the default move ordering results in significantly fewer node expansion than the random move ordering (play any move first). Apart from these CGT techniques, the baseline solver uses other code optimization strategies (especially leveraging the transposition table) which resulted in good speed up. That said, we understand the baseline Clobber solver can be further strengthened. In that case, we believe that our neural network based move ordering strategies would still be useful but may require training on much larger boards. Clearly, this is an interesting question that remains to be answered.

3.2 Deep Convolution Neural Networks

We use a deep CNN model consisting of a sequence of 5 stacks of 1 dimensional convolutional layers followed by a nonlinear ReLU activation and a max pooling layer. At the end of these stacks is a standard softmax for binary classification. This model is trained by minimizing the binary cross-entropy loss via backpropagation of the gradient. We use the Adam optimizer in gradient descent and varying Kernel sizes. We use dropout on each convolutional layer to prevent overfitting.

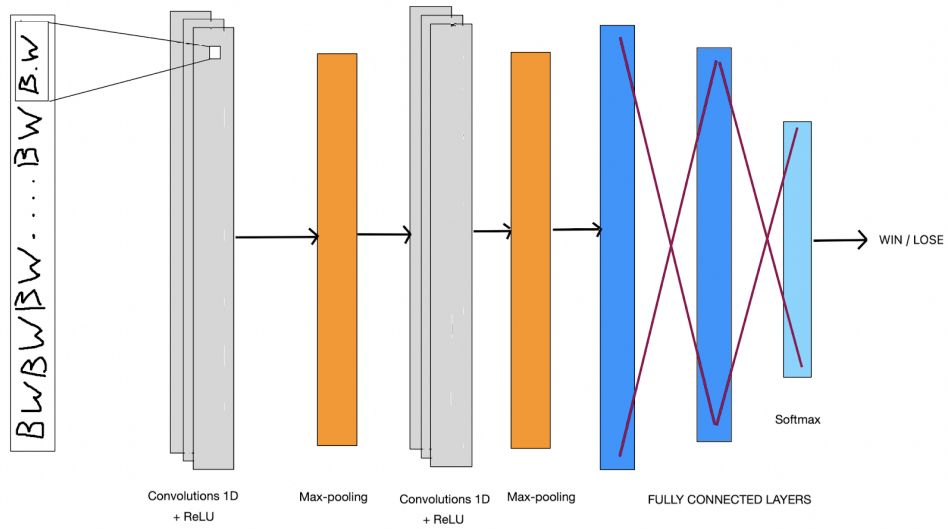

Figure 1: An example of the CNN architecture used in the Clobber solver. The actual model is deeper with more layers. To save space, the exact CNN architecture is not shown.

Each convolutional layer extracts increasingly abstract features from the input as we go deeper into the model. In addition, the number of features extracted decreases with depth from a filter size of 64 to a filter size of 8. The convolutional kernel uses a shift of stride length 1 for smooth transitions. The input to the model is a one hot vector encoding of the board where BLACK is represented as [0,1], WHITE as [1,0] and EMPTY as [0,0]. The primary objective of this design is to ensure the CNN can capture the relationship between subgames and discover non-trivial win and loss patterns.

We fix the maximum size of the board to 40. The training uses boards of size 17 with each board position randomly assigned a “B”, “W” or “.” and the remaining positions (23) filled with “.”. The target label of each board is a win (1) or a loss (0) and is determined using our baseline clobber solver. We trained two different CNNs – one to play as Black, the other to play as White. However, we found that training on a small board of size 17 resulted in poor accuracy during tests on larger board (>20), and also on small boards (<=17) when they are prefixed (left-padding) with dots. The solution was to pad the training samples with random number of dots in the beginning (effectively placing the game at different locations of the input vector). This ensured all the neurons are learning the desired patterns (including game translocation) even when the training board is much smaller than the input vector. We observed significant improvement in test accuracy of the model including on larger boards of size 24+.

Further refinement of the model was performed by refitting the model on sparse boards. A sparse board is generated by randomly placing 20 "B" and "W" on the entire input board of size 40. The objective of generating sparse board is to simulate subgame structures common in Clobber. The number 20 is a hyperparameter chosen through cross validation. The initial training process used 800,000 synthetic examples while the fine-tuning process used 600,000 samples.

The process of training the CNN was quite time-consuming and required a lot of parameter tuning and design decisions. Among the things that did not benefit training were game board compression and exhaustive board position generation for the training set. Board compression implies replacing multiple dots in a sequence with a single dot (B...W -> B.W). Since the CNN had already learned this behavior during training, manual compression did not improve accuracy but added computational overhead. Generating all possible board positions for small boards (such as size less than 15) for training was not effective either as it led to overfitting, a situation where training accuracy is high but test accuracy over unseen samples is poor. Instead, sparse boards and sampling techniques from larger boards were quite effective and did not overfit. Another observation was that attempts to use deeper layers were less rewarding as it resulted in more overfitting and higher inference time. We feel that such deep layers could be useful when the training set is much larger and enough computational resource is available for running the training in acceptable time-frame.

Before deployment in our Clobber solver, the trained models (black-CNN and white-CNN) were tested on unseen data. Boards of size 40 were randomly assigned black and white pieces at random positions following a non-standard distribution. We then evaluate the model accuracy of both the black and the white CNN over 10,000 randomly generated boards. A correct prediction is when the CNN can predict win or lose accurately (and the verification is done by playing the game using the baseline solver). Accuracy is calculated by counting the number of correct predictions and dividing by the total number of test samples. After a successful test run, the distribution is changed to reflect a board with more or less pieces black and white pieces (updated density). This is done by decreasing or increasing the percentage of dots. Again the test is executed with 10,000 randomly generated board positions following the modified distribution. This process is iterated several times. Finally we accept the model if it's test accuracy is at least better than the previous best model for various board densities. After this step, the model is deployed in the Clobber solver if it is found satisfactory. We observed that running tests by sampling from different board sizes with padding (prefixing or suffixing with dots) and compression (replacing multiple dots in sequence with a single dot) does not reflect true accuracy when deployed in the Clobber solver (instead sparse boards do). So, even when we may get high accuracy on the test set, yet deploying the models in the Clobber solver resulted in expansion of more nodes compared to the baseline. This is the reason the test protocol was followed meticulously before any deployment in the Clobber solver.

After training, evaluating and arriving at a final model, the next step was to optimize the trained model for fast inference. To achieve this, we used standard Tensorflow optimization techniques. Specifically, we performed weight quantization and converted the models from Keras to tflite model. Compared to the original Keras model with an inference time of about 0.4 seconds, the optimized tflite version had an inference time of about 4 microseconds. We found this to be sufficient for our experiments and did not optimize the model further, though we believe that further improvement is possible through model pruning and clustering.

Finally, we discuss some of the design considerations which helped minimize the computational overhead of the CNN inference step of our Clobber solver. First, we describe the standard usage of our CNN move-ordering scheme. Specifically, after finding the legal moves for the current player, we play each move on the current board and call the CNN (white-CNN if the current player is black and vice versa) to predict if the resultant board position is a losing one for the opponent. Then we undo the move and apply the next move and call the opponent's CNN again. We repeat this for all legal moves and finally sort the moves on the basis of the confidence score - the move with highest confidence of winning (for the current player) is played first. While each CNN call taking only 4 microseconds of inference time seems negligible, the entire CNN inference process including the playing and undoing of moves significantly degraded performance in comparison to the baseline CGT version of Clobber solver.

In order to ensure optimal use of the CNN, we decided to use the CNN to find only one winning move with a high confidence score (0.7). If, during this process we encounter moves with a high chance of losing for the current player, we play those moves last. All other moves are sorted based on the baseline move-ordering technique. As a reminder, the baseline move-ordering orders the move such that all moves in R subgames are prioritized, all moves in L subgames are deprioritized and the remaining moves are ordered based on the sizes of the subgames with larger subgames played first. To further limit the use of CNN, our algorithm invokes the CNN for the current player only if in the previous step, the opponent could not find a winning move with high confidence. Lastly, the CNN is invoked when the number of remaining pieces (Black / White) is more than 24 and the total number of legal moves is greater than 17. This ensures that the CNN move-ordering comes into play at the highest levels of the game tree where it can make a big difference in the number of nodes searched and in turn make up for the expensive inference time. We have used the baseline move-ordering in all situations where the CNN move-ordering is not invoked

3.3 Deep Reinforcement Learning

In this section we describe the details of our approach towards move-ordering via deep reinforcement learning. Since our task of move-ordering can be thought of as learning the value of each move in a state and sorting them by this value, there are two approaches to consider. The first is to learn a value function for each move. However, as seen in the previous section, having to call the model multiple times while playing and undoing moves between each call has a significant impact on performance.

Table 1: The different initial board position templates used by the Clobber reinforcement learning environment. N is randomly selected such that the final board size is less than B.

Board	**Example**	**Probability**
$(BW)^N$	BWBWBW	12.5%
$(WB)^N$	WBWBWB	12.5%
$(BBW)^N$	BBWBBW	12.5%
$(WWB)^N$	WWBWWB	12.5%
$[B/W]^N$	BWWBWB	25%
$[B/W/.]^N$	BW.B.W	25%

Thus we instead look at learning a policy through the DQN algorithm, which offers us Q-Values for each move. More specifically, these Q-Values are defined as the expected reward from taking a particular action in a particular state. Thus by considering them as move-values, we can obtain values for all moves in a single model call. We detail the exact specifications of our work in the rest of this section.

We first frame Clobber as a Markov Decision Process. Specifically, the state is the concatenation of the current board position and the current player and the actions are the list of legal moves in that state. We note that the next state is not the state immediately after the player's move but rather, the state after the opponent's move. This is done in order to take into consideration the two-player nature of the game. In other words, the combination of the board position and current player once a player moves, is the state for the opponent. The state after the opponent moves is the next state for the player. We use a straightforward reward function as shown in Equation 1. Specifically, we give a positive reward of 1 for a first player win, a negative reward of -1 for a second player win and a small negative reward in all other cases to incentivize picking the fastest winning move. This latter reward is to handle cases where there are multiple winning moves by picking the move with the least number of moves needed to win I.E., the move with the greatest reward. We scale the value of this reward based on the size of the board to avoid scenarios where the expected reward of the winning move may end up lower than the expected reward of other moves.

$$R = \begin{Bmatrix} +1, \text{if first player win} \\ -1, \text{if second player win} \\ \frac{-1}{100*B}, \text{Otherwise} \end{Bmatrix} \tag{1}$$

All reinforcement learning agents require an environment to interact with during the training process. Thus, our first task was to develop a reinforcement learning environment for our agent. This environment is fairly straightforward and requires only a single argument to define - the maximum size of the Clobber board, B. This is important as it defines our action space. At the same time, this does create a limitation on the final DQN - we can only utilize it for boards of this size or lower. Generating this action space is trivial in 1D Clobber as there are only two moves available at each position - clobber the position to the left or clobber the position to the right. The two exceptions are the first and last columns of the board which have only one move. Thus, the action space for a board of size B is always $2 * (B - 1)$. One challenge that arose in constructing this environment was that of generating initial boards. Our first attempt was to use random strings "B", "W", or ".". However, we found that this resulted in sub-par performance from the trained models due to the sparsity of the board. Thus we setup a number of templates from which the board was randomly initialized from. We did not stick to any one single template as that might have lead to the model performing well only on certain templates. Table 1 illustrates the different templates, their individual probabilities and an example for each template. At each step, the environment takes as input an action, plays that action, generates the resultant state, calculates the legal moves remaining for the current player and computes the reward before returning the new state, reward and a flag denoting if the game is over. We note that all states are represented as vectors, with 0.0, 1.0 and 2.0 indicating a blank space, a black token and a white token respectively.

We train our DQN agent based on prior work [Mnih et al., 2013, van Hasselt et al., 2016]. Specifically, we make use of the Double DQN. For our policy, we utilize an Epsilon-Greedy policy with an

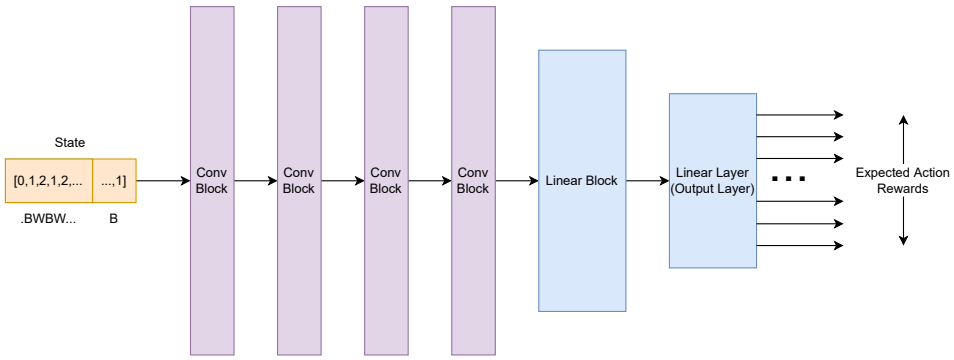

Figure 2: Our final DQN architecture. Each conv block consists of a convolutional layer followed by a ReLU activation function and dropout regularization. The linear block is the same but uses a fully connected layer in-place of a convolutional layer.

exponentially decaying value of epsilon. Additionally, we utilize a number of optimizations from previous literature to assist in this training process. These are soft-target updates, experience replay and action masking. Soft-target updates refer to a particular way of updating the target network in DDQN. That is, instead of simply replacing the target network with the weights of the policy network, we instead make a soft update (shown in Equation 2).

$$\theta_{\text{target}} = \tau.\theta_{\text{policy}} + (1 - \tau).\theta_{\text{target}} \tag{2}$$

Where τ is a hyperparameter indicating the percentage amount the target network must move towards the policy network, with a value of 1 equivalent to replacing the target network by the policy network at each step. Experience replay essentially refers to a memory buffer that holds the E most recent transitions that the model has experienced. While training, instead of simply using the latest transition to make an update to the model, we instead sample from this experience replay buffer. This has multiple advantages. Most importantly, it allows for better convergence of the deep neural network as it decouples each transition from the other transitions due to the random sampling. It also allows for learning from the same transition multiple times, allowing the model to learn more efficiently. In order to prevent learning from very old transitions where the agent might not have had a good policy, we keep only the latest E samples in the buffer. Finally, we use a simple action-masking scheme to ignore otherwise illegal actions. Specifically, during the training process, before we make a prediction, we add a large negative value to all invalid actions before taking the greedy action. If we take a random action due to the epsilon-greedy policy, we simply sample from the pool of legal actions.

Our final DQN model consists of 4 convolutional blocks, each of which consist of a convolutional layer, a ReLU activation function and dropout regularization. These layers start with a kernel size of 16 and have 64 such kernels. These values are halved for each successive layer. We use a dropout probability of 0.3 in all cases. The first linear block consists of a fully connected layer followed by a ReLU activation function and dropout with probability 0.3. This layer has 128 neurons and links to our final output layer which reduces the number of neurons to the size of the action space. A pictorial representation of our architecture can be seen in Figure 3.3.

Our training regimen is fairly straightforward and is depicted in Figure 3.3. A single episode consists of the environment being reset to a new board and the game being played out while following our policy. After each move is played, we record the transition (state, action, next state, reward) and store it in our replay buffer. We also sample from the buffer and train the neural network on this sampled batch. This process is repeated over a large number of episodes (1,000,000) with the Adam optimizer and the standard DQN weight update for training the neural network. For the opponent, we use either a random agent or via self-play. The random agent simply makes a random action on each turn. The self-play agent while a little more complex in the implementation, simply involves the same agent playing on both sizes. However, we still ensure that each transition added to the replay buffer is valid (the next state is always the state after the opponent's move). We compare the differences between these two approaches in the following section. To evaluate the trained model, we periodically test its performance against a random agent by simulating a significant number of games. We repeat

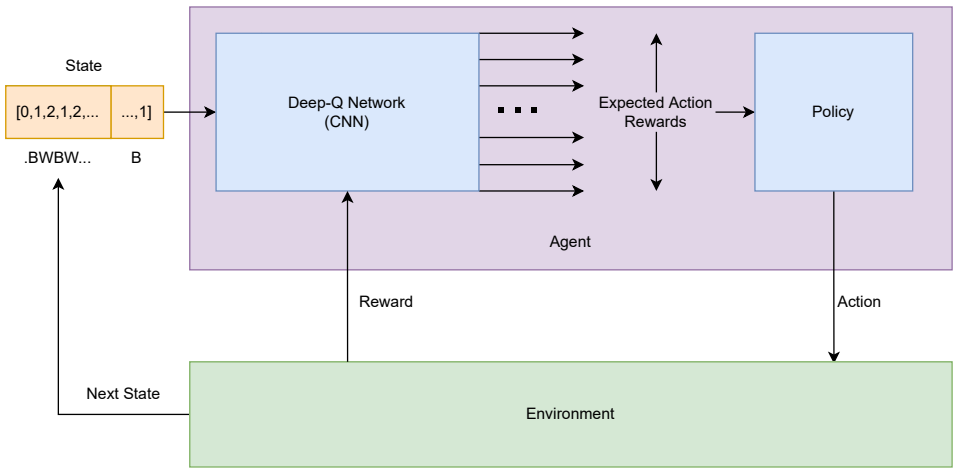

Figure 3: The reinforcement learning model in action.

Table 2: The model's inference accuracy for different board sizes are listed below. Each accuracy score is calculated over 10,000 randomly generated boards.

Board Size	**CNN-Black (% accuracy)**	**CNN-White (% accuracy)**
20	85.6	88.6
25	81	83.4
30	80.78	79.11
35	79.66	72.78

this process more elaborately and run it 5 times at the end of the training process and consider the win-rate. The agent uses a purely greedy policy for this evaluation step.

Finally, before integration with the Clobber solver, we had to optimize the model for inference. To that end, we developed a stripped version of the agent which only had the functions and properties required for inference and we converted this into the ONNX format, an open format in which machine learning models can be stored and loaded for cross-compatibility and fast inference. While we acknowledge that other steps could be taken to optimize the model's inference time further, we did not proceed further due to time constraints. Our final models had an inference time of 96 microseconds per call. In addition, this model only took up 178 kilobytes in total, making it compact enough to fit inside the 1 MB requirement used for previous assignments. In order to make the best use of time, we enable this RL move-ordering system if and only if there are at least 20 tokens on the board. While this value is not exact, we assume that it is the value at which the trade-off between the overhead of our RL move-ordering and the speed of a brute-force negamax solver works in our favor. We note that the actual value may be lower than 20, though we have set it as an upper bound. We also reiterate this move-ordering scheme cannot be used in boards of size greater than B, though we do not foresee such boards needing much study.

Our final model uses the following hyperparameters: Soft-target update parameter $\tau = 0.001$, experience replay buffer size $E = 10000$, discount factor $\gamma = 0.99$, batch size of 64, learning rate $\alpha = 0.0001$, and a maximum board size of 40.

4 Experimental Results & Discussion

In this section we describe the different experiments we performed in the case of both proposed move-ordering approaches.

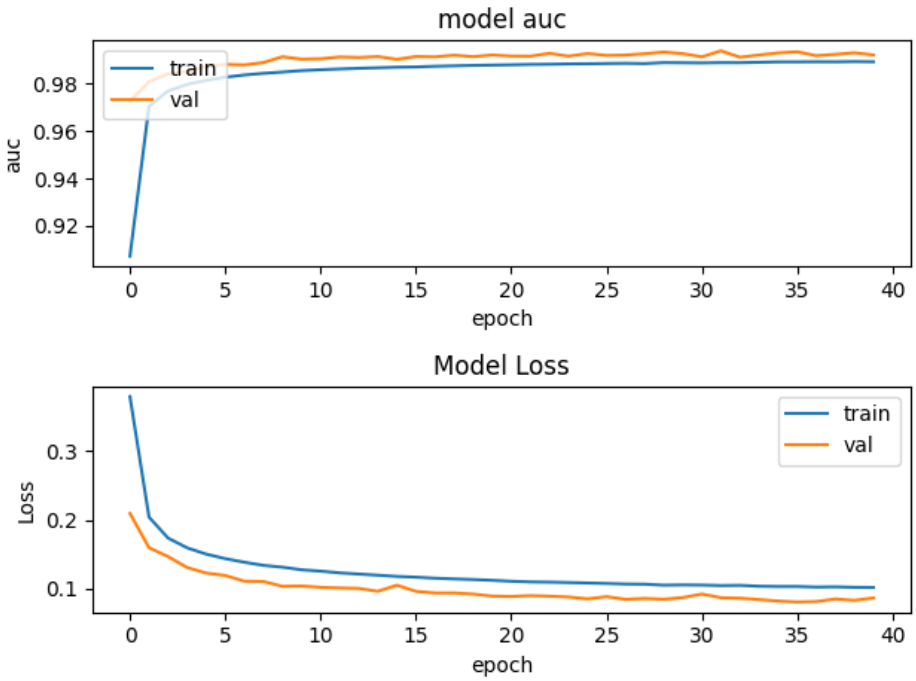

Figure 4: The plot shows the training and validation loss and the ROC-AUC score of the black-CNN.

Table 3: A comparison of the execution results of the Clobber solver using the CNN move-ordering versus the baseline move-ordering.

Board	**Size**	**First Player**	**CNN-ordering**	**Baseline-ordering**
$(WB)^{15}$	30	B	B 19-20 14.68 304944	B 9-8 33.15 698817
$(WB)^{15}W$	31	W	B None 82.12 1604209	B None 161.75 3347088
$(WB)^{15}W$	31	B	B 23-22 10.50 218937	B 5-4 73.95 1532934
$(BW)^{16}$	32	W	W 29-30 46.45 916989	W 21-22 84.38 1721429
$(WB)^{16}$	32	W	W 12-13 48.34 954372	W 12-13 117.33 2388654
$(BW)^{17}$	34	W	W 31-32 197.94 3715242	W 21-22 507.67 9820176
$(BW)^{17}$	34	B	B 22-23 229.40 4390859	B 18-17 329.96 6435019
$(WB)^{17}$	34	B	B 23-24 180.76 3336638	B 21-22 514.61 9820176
$(BW)^{18}$	36	B	B 22-23 808.145 10291762	B 18-17 1199.84 22390724

4.1 Deep Convolution Neural Networks

Figure 4.1 and Figure 4.1 depicts the training loss, validation loss and the ROC-AUC score of the black-CNN and white-CNN.

Table 2 compares the accuracy score for the final model on boards of different sizes. We see that the accuracy reduces as the size of the board increases due to the explosion in the size of the state space.

Table 3 compares the execution results of our CNN move-ordering against the baseline move-ordering discussed previously. The results show that the CNN is able to cut down on the number of nodes visited significantly, sometimes by about 50%. We can also see that the time taken to solve these boards has reduced by a significant margin. However, we also found that there is still scope for improvement. Importantly, the CNN does poorly for $(BBW)^n$ patterns and often expands more

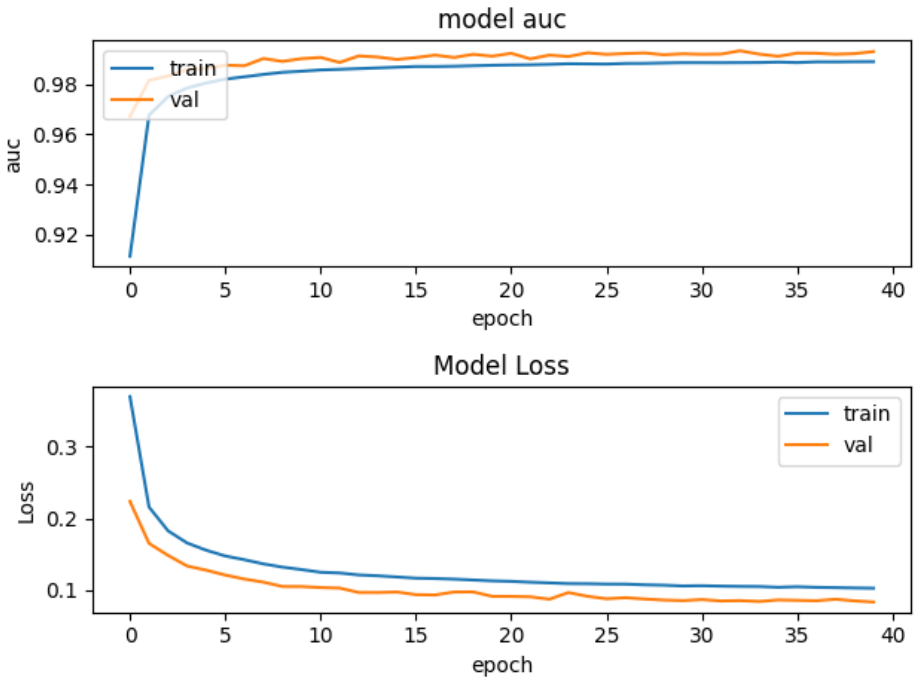

Figure 5: The plot shows the training and validation loss and the ROC-AUC score of the white-CNN.

nodes than the baseline. This issue might be fixable by re-training the models with samples of the $(BBW)^n$ pattern. However, due to a lack of time we leave this for future work.

Another interesting observation is that the CNN move-ordering almost always finds different winning moves compared to the baseline ordering. This raises the question of correctness. We have verified independently using a negamax solver that even though the winning moves are different, they are indeed winning moves.

4.2 Deep Reinforcement Learning

Although the bulk of our efforts were in setting up the training algorithm and the clobber environment, we performed a number of experiments to analyze our reinforcement learning models. We first examine the relative performance of models trained on different board sizes with the expectation of lower performance on larger boards. Second, we compare the difference between the model trained against a random agent and one trained via self-play. Finally, we examine the results of our reinforcement learning move-ordering scheme in our integrated clobber solver. Table 4 depicts the win-rate of our trained agent against a random player on 10,000 random boards. We repeat this experiment 5 times and take the average win-rate to account for randomness.

We see that the model does quite well on the smaller boards, boasting an almost 63% win-rate on board size 15. However, as the size of the board increases, the win-rate reduces. This is because all of these models were trained on the same number of episodes, despite the vast difference in the number of possible games. To illustrate, boards of size 15 have $3^{15} \approx 14$ million different possibilities. However, boards of size 40 have $3^{40} \approx 10^{19}$ possibilities. This is a staggering difference and is the reason we attribute for this discrepancy in win-rates.

We also consider the difference, or lack thereof, between the self-play model and the random play model trained on board size 40. We see that there is little difference between the actual win-rates of these models. In addition, we found that the self-play model was significantly worse in terms of performance in the final solver. We suspect an issue with self-play that has occurred in previous work [Silver et al., 2017] - the agent gets stuck in a cycle of strategies without evolving further. A relatively

Table 4: Comparison of agent win-rates against a random agent. Models differ only on the maximum size of the board with all other parameters being different.

Board Size	**Model win-rate**
Board Size 15	62.998
Board Size 20	60.513
Board Size 25	51.144
Board Size 30	51.119
Board Size 40	52.200
Board Size 40 (Selfplay)	52.152

Table 5: A comparison of the different move-ordering algorithms. Bold indicates best approach, italics indicate a timeout.

Format	Player	**Size**	Results	**Random**	**Default**	**CNN**	**RL**
$(BW)^{10}$	W	20	**Nodes**	4130	1722	1722	**1162**
			Time	0.44683	0.17855	0.20323	**0.15159**
$(BW)^{13}$	W	26	**Nodes**	204774	78215	32887	**25444**
			Time	25.2852	9.46971	4.41373	**4.38826**
$(BBW)^{9}$	B	27	**Nodes**	18083	7844	7844	**5709**
			Time	2.30008	1.00434	1.19046	**1.00204**
$(WB)^{14}W$	B	29	**Nodes**	*5889300*	1270778	526587	**439652**
			Time	*980.000*	205.293	122.388	**115.612**
$(BW)^{15}$	W	30	**Nodes**	*7090539*	698817	560718	**497985**
			Time	*980.000*	99.7126	**99.2482**	114.293
$(BW)^{15}B$	W	31	**Nodes**	*5963465*	1532934	537206	**300876**
			Time	*980.000*	281.934	100.456	**86.1686**

simple solution to this is to use a technique known as league training [Vinyals et al., 2019]. However, we do not implement this due to a lack of time.

Table 5 compares the outputs of our reinforcement learning move-ordering system against the CNN approach discussed previously, our baseline move-ordering and a random (no explicit move-ordering) scheme. All tests were run on the same system with the same parameters, with the only difference being the move-ordering scheme. A standard timeout of 1000 seconds was used for all cases. We see that our reinforcement learning move-ordering scheme outperforms the others by a significant margin, especially in larger boards. We especially consider the final row where using no move-ordering times out and the default move-ordering takes over 1.5 million nodes. We see that the reinforcement learning method takes just over 300,000 nodes to solve the board. We do note that the reinforcement learning method is a little slow, as seen in the board of size 30 where the reinforcement learning method takes longer than the CNN and default despite having searched significantly less nodes. We attribute this to discrepancy to the overhead caused by model calls and note that further optimization is possible (the CNN takes 4 microseconds for a call versus versus 96 microseconds for the RL model). However, despite this, we believe that the results prove the efficacy of the reinforcement learning approach.

5 Conclusion

Although our proposed move-ordering schemes work well, there is still room for improvement. We highlight a few key areas and also offer suggestions for future work to improve on. Both approaches are significantly better than the baseline in many cases. However, this is not always true. The CNN approach does not work well for boards of the BBW format while the RL approach does not work

very well on boards greater than size 30. The solution to both of these issues is to alter the training process. Specifically, incorporating boards of different formats while training the CNN approach and simply training for longer on the RL side. In addition, changes could be made to the model architecture as both our models are relatively simple in architecture. In the case of the RL model, the DQN we used, though effective, is not the state of the art. Alternative training approaches do exist. Another alternative is to use a league training system for a self-play agent or use MCTS in conjunction with either model. Finally, while our baseline solver does use combinatorial game theoretic techniques, it can be improved. Upgrading this solver to use the current state-of-the-art methods and using a neural network based move-ordering heuristic could yield a very strong solver. We believe that our success in 1-dimensional clobber could mean that these methods would work in similar or more complex games such as impartial clobber of 2-dimensional clobber.

To summarize, in this project we examined the effectiveness of two novel neural network based move-ordering for the 1-dimensional version of the combinatorial game of clobber. We empirically show that using neural networks for move-ordering, trained either via a supervised deep learning algorithm or a reinforcement learning algorithm, are effective move-ordering heuristics. We have demonstrated their usefulness on 1-dimensional Clobber, particularly in larger boards. We hope that our work acts as a starting point for future work in this area.

References

Michael Albert, J.P. Grossman, Richard Nowakowski, and David Wolfe. An introduction to clobber. *Integers*, 5, 08 2005.

Jeroen Claessen. Combinatorial game theory in clobber. *Master's thesis, Maastricht University*, 2011.

Markus Enzenberger. Evaluation in go by a neural network using soft segmentation. In H. Jaap van den Herik, Hiroyuki Iida, and Ernst A. Heinz, editors, *Advances in Computer Games, Many Games, Many Challenges, 10th International Conference, ACG 2003, Graz, Austria, November 24-27, 2003, Revised Papers*, volume 263 of *IFIP*, pages 97–108. Kluwer, 2003.

Teddy Etoeharnowo. Neural networks for clobber. *Bachelor's thesis, Leiden Institute of Advanced Computer Science*, 2017.

Chao Gao, Ryan Hayward, and Martin Müller. Move prediction using deep convolutional neural networks in hex. *IEEE Transactions on Games*, 10(4):336–343, 2018. doi: 10.1109/TG.2017.2785042.

Erik Järleberg. Reinforcement learning on the combinatorial game of nim. *Bachelor's thesis, KTH Royal Institute of Technology*, 2011.

Jayanth Koushik. Understanding convolutional neural networks, 2016. URL `https://arxiv.org/abs/1605.09081`.

Andrew Lin. Using combinatorial solutions and atomic weights to play competitive ai clobber. In *2019 International Conference on Technologies and Applications of Artificial Intelligence, TAAI 2019, Kaohsiung, Taiwan, November 21-23, 2019*, pages 1–7. IEEE, 2019. ISBN 978-1-7281-4666-9. doi: 10.1109/TAAI48200.2019.8959940. URL `https://doi.org/10.1109/TAAI48200.2019.8959940`.

Chris J. Maddison, Aja Huang, Ilya Sutskever, and David Silver. Move evaluation in go using deep convolutional neural networks. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL `http://arxiv.org/abs/1412.6564`.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*. 2013.

Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *CoRR*, abs/1511.08458, 2015. URL `http://arxiv.org/abs/1511.08458`.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017. URL `http://dx.doi.org/10.1038/nature24270`.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018. ISBN 0262039249.

JWHM Uiterwijk and Janis Griebel. Combining combinatorial game theory with an alpha-beta solver for clobber: Theory and experiments. In *BNAIC 2016: Artificial Intelligence*, Communications in Computer and Information Science, page 78/92, United States, 2017. Springer.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 30(1), Mar. 2016. URL `https://ojs.aaai.org/index.php/AAAI/article/view/10295`.

Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, L. Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Caglar Gulcehre, Ziyun Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, pages 1–5, 2019.

Wieczorek Wojciech, Rafał Skinderowicz, Jan Kozak, and Przemysław Juszczuk. New trends in clobber programming. *ICGA journal*, Vol. 34:150–158, 09 2011. doi: 10.3233/ICG-2011-34304.

Table 6: Authorship statement on the entire project.

Task	**Person**
Literature Survey	Akash, Debraj
Problem Formalization	Akash, Debraj
Project Presentation	Akash, Debraj
Project Proposal	Akash, Debraj
CNN Approach	Debraj
RL Approach	Akash
Early Milestone Report	Akash, Debraj
CNN Integration	Debraj
RL Integration	Akash
CNN Evaluation	Debraj
RL Evaluation	Akash
Final Presentation	Akash, Debraj
Code Finalization	Akash, Debraj
Final Report	Akash, Debraj