Offline Reinforcement Learning: Training a Lean and Model-Free Chess Agent

Raphael Chew, Shaun Fendi Gan, Peijun Xu
Chess-RL GitHub Repository

***Abstract*—Lean Convolutional Neural Networks were trained using Reinforcement Learning (RL) to mimic a Monte Carlo Tree Search algorithm at playing chess. In 120 training games, this agent achieved a $10.2\% \pm 3.8\%$ win rate and $< 1\%$ loss rate against an opponent making random moves. Similarly, semi-supervised methods achieved a $15.3\% \pm 6.4\%$ win rate with $< 1\%$ loss rate against the same opponent from 120 training games. Reward shaping and behavior cloning were also tested but did not produce effective chess agents.**

I. Introduction

In March 2016, DeepMind's AlphaGo beat World Champion Lee Sedol at Go, 4 games to 1. Seen as a landmark victory in the capabilities of AI, this spurred many conversations about its algorithm and the potential of reinforcement learning. The AlphaGo agent was based on a neural network that was trained to mimic a Monte Carlo Tree Search (MCTS) algorithm and learn from human play.

Later in October 2017, DeepMind unveiled AlphaGo Zero, which did not rely on any data from human games and was trained solely through self-play by reinforcement learning. AlphaGo Zero far surpassed AlphaGo by winning its predecessor 100 games to 0. As a new challenge, Google later developed AlphaZero, which generalized the algorithm to play chess and Shogi in addition to Go. In December 2017, AlphaZero defeated AlphaGo Zero 60 games to 40 and also outplayed the chess world champion program Stockfish with 28 wins, 0 losses, and 72 draws.

AlphaZero was trained through self-play, where it would play against old versions of itself to avoid use of human training data. Using this framework allowed it to continuously improve by playing against slightly worse versions of itself, leading to a reduced learning curve. In order to achieve this independence in reinforcement learning, AlphaZero incurred large hardware and monetary costs. AlphaZero's neural network was trained on 64 TPUs, along with an additional 5,000 TPUs to generate the games to learn from [1]. After completing training, the algorithm also required 4 TPUs on a single machine to play its games. Monetary costs of AlphaZero are not publicly available, however it was reported that its predecessor AlphaGo Zero cost DeepMind \$25 million to train and perform [2]. Our research experiments with lighter versions of AlphaZero's offline reinforcement learning algorithm for chess. We reconstruct a leaner MCTS value and policy network algorithm from scratch, to investigate the possibility of training a less capable chess agent, but within the computational limitations of the average machine learning engineer. Specifically, we investigate the viability of using lean convolutional neural network (CNN) architectures for mimicking the values and policies discovered by the MCTS during self-play.

II. Overview of Methods

Our investigation was conducted in three stages of increasing complexity. Firstly, we test our hypothesis that lean CNN architectures can model chess game-play, by implementing Double Deep Q-Networks (DDQN) in a Capture Chess environment. Capture Chess aims to capture opponent pieces, but does not have the goal of achieving checkmates. By training a DDQN agent to out-capture existing agents, we build confidence in using CNNs as the deep learning base for more complex algorithms.

Secondly, we design from-scratch an offline reinforcement learning framework for chess, inspired by AlphaZero's MCTS algorithm. We experiment with CNN structures, reward shaping, loss functions, and training sequences to understand how architectures affect agent performance.

Lastly, we introduce semi-supervised learning support for our offline RL chess agent, using demonstration learning and behavior cloning to speed up training time and make the process more tractable.

A. Mathematical Representation of Chess

The board states and proposed actions require a mathematical representation in order to encode the information into the neural network. In our work, the state is the board and positions of all pieces, and the action is the move of a piece to another square.

The state is encoded as an $8 \times 8 \times 8$ tensor, where the first six 8×8 matrices convey information for each of the six pieces. The remaining two 8×8 matrices encode information of the move number and if a draw can be claimed. On the other hand, action is encoded as a 64×64 matrix of softmaxed probabilities, where each element in the matrix is the strength of a specific move. The element's first index refers to the coordinates of the move-from square and the element's second index refers to the coordinates of the move-to square.

III. Exploring Double Deep Q-Networks for Capture Chess

We begin our study with a simplified version of chess, Capture Chess, to test if our CNN architecture can effectively learn Q-values in a Double Deep Q-Network (DDQN) setup. The goal in this chess version is to capture all of the opponent's pieces, while maintaining the greatest possible material advantage at the end of the game. We build upon Arjan Groen's Q-Learning algorithm and train agents using the DDQN algorithm elaborated in the next section [3].

A. Double Deep Q-Network Algorithm

DDQN is an off-policy reinforcement learning method that learns Q-values to represent the strength of a policy (move). DDQN utilizes two networks: a primary network and a target network, where the primary network is trained to learn Q-values from a memory buffer and the target network is a fixed version of the primary network used in the Bellman equation. The target network encourages stability, preventing a moving target in training. DDQN is considered off-policy as it uses previous experiences from the memory buffer in order to estimate the next best action.

The DDQN network training is described in Algorithm 1, with the following Bellman equation form for the Q-value update:

$$Q'(s_t, a) = r(s_t, a) + \gamma \max_a Q_{tgt}(s_{t+1}, a) \tag{1}$$

where Q' is the Q-value, s is the state, a is the action, r is the reward, γ is the discount factor, t refers to a timestep, and Q_{tgt} is the Q-value of the target network.

For capture chess, $\gamma = 0.1$, learning rate = 0.07 and a memory buffer of 1000 were used. We track agent convergence during training by computing the change in the smooth rolling mean with a window of 100 moves.

Algorithm 1: DDQN Training Engine for Chess

```
   Input: Agent, Board
   Output: Simulation_Results
1  for $i = 1, \cdots, N_{\text{sim}}$ do
2      $\epsilon = \frac{1}{1+\frac{i}{150}}$
3      while Gameover == False do
4          Explore ← (x ∼ U[0,1] < $\epsilon$)
5          if Explore == True then
6              Move ← RandomMove
7          else
8              $Q_t$ ← Agent.predict(Board)
9              $a_t \leftarrow \arg\max Q_t$
10         Reward, Board$_{t+1}$ ← STEP(Board$_t$, $a_t$)
11         Memory_Buffer.insert([Board$_t$, $a_t$, Board$_{t+1}$,
            Reward])
12         Agent, errors ← NETWORKUPDATE(Agent)
13 return Agent
```

During the network update, we compute temporal difference errors using the difference between the discounted future rewards and the current agent's Q-value prediction. This error is then directly used as the sampling probabilities for their corresponding state-action pair in the memory buffer. This allows for more frequent sampling of erroneous state-action predictions, so that the agent will gradually correct its poor judgements of the board.

Algorithm 2: NETWORKUPDATE

```
  Input: FixedAgent, Agent, Memory_Buffer
  Output: UpdatedAgent
1 MiniBatch ← Sample(Memory_Buffer)
2 Board, Move, NextBoard, Reward ← MiniBatch
3 Potential_Rewards ← max$_a$
    (FixedAgent.predict(Board$_{t+1}$)) * $\gamma$
4 $Q'$ ← Reward + Potential_Rewards
5 UpdatedAgent ← Agent.fit(x = Board$_t$, y = $Q'$)
6 return UpdatedAgent
```

B. Network Structure & Opponent Selection

A convolutional neural network structure as seen in Fig. 1 was used. This comprised of two convolutional layers with one filter and a kernel size of (1,1) to capture information from the 3D board state. This produced 18 trainable parameters, 2×8 for the board and two for the bias of the convolutional layers. These were then flattened and dotted to produce the (64×64) Q-values. Sigmoid activations were used.

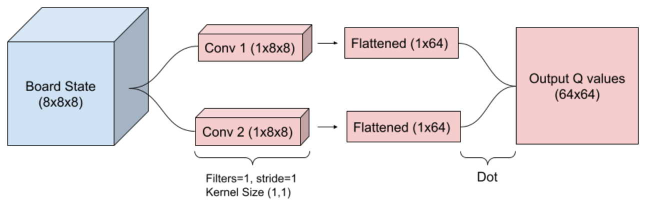

Fig. 1. Convolutional neural network structure for capture chess. Two convolutional layers with a kernel size (1,1) and a filter size of one were dotted together to produce the Q-values. This produced 18 trainable parameters that were sufficient for a competent capture chess engine.

Within chess problems, neural networks typically follow structures found in computer vision tasks. For instance, AlphaZero's structure comprises 40 ResNets along with convolutions to evaluate the board state and make decisions [1].

There are many considerations when selecting the agent's adversary. An appropriate level of difficulty is necessary to ensure a moderate learning curve. If opponents are weak, the agent would not improve with any significance. Conversely, if opponents are too strong, the agent might never see the reward and have the chance to update its network meaningfully.

In our implementation, we train DDQN against two opponents: a naive engine that makes random legal moves and an agent that was trained against that naive engine. From experimentation, the agent's target network was updated every $n = 10$ games to produce bite-sized challenges that lead to favorable results.

C. Results for Capture Chess

Vanilla DDQN was trained on 500 games against a random bot, whilst Self-Play DDQN was trained on 1000 games. Vanilla DDQN only took 300 games to show competence in capturing pieces and converge on optimal behavior. After training, both these networks were tested against an opponent that makes random moves and then against each other for a total of 100 games. Their averaged material advantage at a given turn count can be seen in Fig. 2.

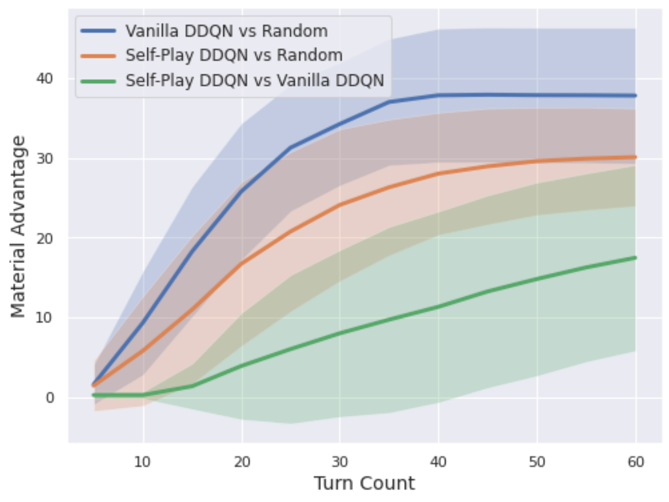

Fig. 2. Simulation Results for DDQN networks in Capture Chess. Self-Play DDQN agent records lower material advantage over a random agent than Vanilla DDQN. However, Self-Play DDQN agent consistently out-captures the Vanilla DDQN agent when competing against each other. This is indicative of its learnt defensive playing style that allows it to obtain a material advantage against the aggressive Vanilla DDQN.

Vanilla DDQN is the strongest bot against random agents, obtaining the largest material advantage of 38 ± 9.5 piece points in 40 turns over the random opponent. It also achieves this material advantage the quickest. Self-Play DDQN reaches a material advantage of 30 ± 6.3 within 60 turns. From analysing the games played by the agents, Vanilla DDQN is seen to overfit against the random bot where overly aggressive capture plays are common. On the other hand, since Self-Play DDQN learns to play against an opponent that is trying to achieve the same goal as itself, it develops a defensive playing style, explaining the slower achievement of material advantage.

Upon training a Self-Play DDQN, we find that this robustifies the agent to a wider variety of opponents that may act differently. As seen in Fig. 2, the Self-Play DDQN agent records lower material advantage over a random agent as Vanilla DDQN. However, the Self-Play DDQN agent consistently out-captures the Vanilla DDQN agent in simulations. We record an average material advantage of 16.0 ± 13.1 for Self-Play DDQN's matchup against Vanilla DDQN, over 100 games, at move count 60. This is indicative of its learnt defensive playing style that allows it to obtain a material advantage against the aggressive Vanilla DDQN. This suggests there is great advantage in periodically updating the agent's opponent and exposing our agent to different approaches.

In this training instance, we observe that the opponent agent should only be updated if training requires more than 1000 games, or else the Self-Play DDQN agent would not effectively learn how to outplay the existing opponent.

Therefore, we report that 18 trainable parameters in a 2-convolutional layer neural network sufficiently learns an adequate capture chess strategy. While more complex CNN structures could be used, this would come at the expense of training time required. The current framework only requires only 9.81 ± 0.3s per training game.

Different rewards shapes were also explored when progressing towards the MCTS algorithm for Real Chess, but it did not produce fruitful results. These experiments can be found in the appendix.

IV. Monte Carlo Tree Search with DDQN for Real Chess

Having explored Capture Chess, we build upon our DDQN framework by incorporating Monte Carlo Tree Search (MCTS) to play Real Chess. As with our original goal of this project, we aim to teach a DDQN agent how to checkmate an opponent, without using any game databases or supervised methods. The agent will have to explore moves and self-learn which mathematical board states edge it closer to the eventual checkmate reward. Notably, checkmate is an extremely sparse reward which is only encountered at the end of the game. In order to accelerate the training process and ensure that the agent encounters enough checkmates to learn good moves from bad ones, we implement a modified version of the MCTS algorithm to effectively search through board states.

A. *MCTS Algorithm*

In order to find the next best move in chess, the standard approach is performing a tree search of all possible outcomes. This quickly becomes computationally intractable, where the game tree complexity for chess is $\sim 10^{120}$ (Shannon Number) [4]. A Monte Carlo Tree Search tackles this issue by searching the most promising and relevant branches, and can be summarized in four steps

1) Selection - Select most promising piece to move
2) Expansion - Obtain all possible next moves
3) Simulation - Estimate the opponents move
4) Backpropagation - Update move values from outcome

where these four steps are repeated for a fixed number of simulations N_{sim}. The output is a developed tree where the most promising trajectories deemed by our agent is explored.

The selection phase is done using upper confidence bounds (UCB)

$$\text{UCB}(\mathcal{N}) = -V_{\text{child}} + P_{\text{child}} \frac{\sqrt{N_{\text{parent}}}}{N_{\text{child}}} \tag{2}$$

where $\mathcal{N}$ is a node in the MCTS, V_{child} is the value of the child, P_{child} is the prior score of the child that is assigned from the Q-values, and N_{parent}, N_{child} are the visit counts of MCTS on the parent and child nodes respectively.

B. *MCTS and DDQN Engine*

The MCTS algorithm is embedded in the DDQN Engine to help our chess agent overcome the sparse reward challenge. By searching for moves that are worth the most value, the MCTS arm pushes the agent to pick better moves during training, in hopes of encountering a final checkmate. These board states and actions from the MCTS are continually loaded in to

Algorithm 3: Monte Carlo Tree Search

```
Input: Tree Root, Board, ToPlay
Output: Tree Root Node, Expanded with Children
$V, Q \leftarrow$ Agent.predict(Board)
Root $\leftarrow$ TreeNode(player, ToPlay)
Root $\leftarrow$ Root.Expand($Q$, ToPlay)
for $i = 1, \cdots, N_{\text{sim}}$ do
    $\mathcal{N} \leftarrow$ Root
    Search_Path $\leftarrow [\mathcal{N}]$
    repeat
        $a_w, \mathcal{N} \leftarrow$ SELECTBESTCHILD($\mathcal{N}$)
        Search_Path.insert($\mathcal{N}$)
    until $\mathcal{N}$.children == None;
    Select Penultimate Node as Parent
    Parent $\leftarrow$ Search_Path[-2]
    Board $\leftarrow$ STEP(Board, $a_w$)
    if Gameover == False then
        $V, Q \leftarrow$ Agent.predict(Board)
        $\mathcal{N} \leftarrow \mathcal{N}$.Expand($Q$, $-$ Parent.ToPlay)
    BackPropagate($V$, Search_Path, $-$ Parent.ToPlay)
return Root
```

Algorithm 4: SELECTBESTCHILD

```
Input: Tree Node $\mathcal{N}$ of MCTS
Output: Best Action $a^*$& Best Child Node
score$^*$, child$^*$, $a^* \leftarrow -\infty$, None, $-1$
for child $\in \mathcal{N}$.children do
    score, $a \leftarrow$ UCB(child), child.action
    if score $>$ score$^*$ then
        score$^*$, child$^*$, $a^* \leftarrow$ score, child, $a$
return $a^*$, child$^*$
```

the memory buffer helping the agent's network to learn. The inclusion of UCB scores help our agent prioritize exploration over exploitation in the earlier searches.

Fig. 3 illustrates the modified DDQN deep neural network structure, where the key difference is the inclusion of both a policy and a value head. The value head outputs a single value that describes board advantages given a particular board state, where higher values signal stronger positions. The policy head outputs a 64x64 matrix that describes the action probabilities, where higher probabilities signal stronger moves.

With each network update, the agent attempts to bring its value head and policy head outputs closer to the MCTS' value and policy evaluation of the board. For real chess, $\gamma = 0.1$, learning rate = 0.1, a memory buffer of 1000, MSE value loss, and cross-entropy policy loss were used.

C. *Results for Real Chess*

The MCTS and DDQN pair sees promising results in learning how to checkmate a random opponent. Given the computational overhead of 95s±10s per game during training, we limit training to 120 games to verify if the current deep learning networks can capture any signals. We test our agents in 40-move games for faster evaluation. With our current

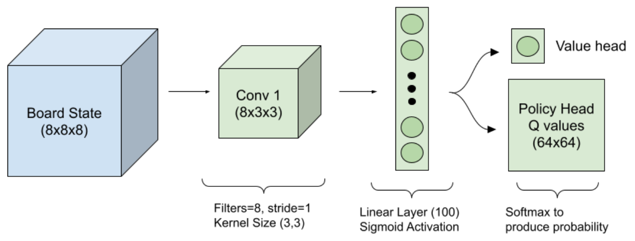
Board State (8x8x8)
Conv 1 (8x3x3)
Filters=8, stride=1 Kernel Size (3,3)
Linear Layer (100) Sigmoid Activation
Value head
Policy Head Q values (64x64)
Softmax to produce probability

Fig. 3. Neural network structure for real chess. The input is passed into a convolutional layer with 8 filters, stride of 1 and a kernel size of (3,3), followed by a fully connected linear layer that outputs both a policy and a value head. This had 400,000 trainable parameters

lean framework, we report a $10.2\% \pm 3.8\%$ win rate over random agents across 150 testing games and a $< 1\%$ loss rate. Remarkably, the agent also establishes a steady material advantage over a random agent despite there being no explicit reward encoded for capturing pieces. The sparse reward of checkmate led the agent to gradually learn that capturing pieces on its way to checkmate is an advantageous strategy. We report a mean 3.4 ± 1.07 piece point advantage over random agents at the 40 move mark.

From experimentation, we observe that lean CNNs that account for both value and policy loss equally help the agent learn chess at a faster rate. The following sub-sections detail the relationships between CNN layers, loss types, network parameters, network structures, and win rates during agent evaluation.

1) Experiments with CNN Layers: In an environment with reduced computational resources, extremely deep neural networks are challenged by the need for extensive amounts of training iterations to adequately update its weights. Although deeper CNNs would capture more nuanced information of a chess board's states, they prove to be infeasible with limitations of training games. We experiment with three neural network depths that vary by their number of CNN layers and report that the shallowest neural network surpasses deeper variants within a 120 game training phase. Notably, deeper networks have the potential for developing more complex chess strategies as shown in AlphaZero's 40 ResNet implementation. However, a 1-layer CNN captures significant signal and presents itself as the structure of choice in this instance.

2) Experiments with Loss Types: The DDQN chess agent uses the policy values to derive the best move for a given board state. We explore the use of three different loss types when training the agent to observe that cross-entropy loss outperforms mean squared error and mean absolute error loss. Given that the policy predictions hover between 1×10^{-2} and 1×10^{-5}, the loss function should be sensitive to minor changes. On this note, MSE shrinks the loss unnecessarily. On the other hand, cross-entropy loss brings all losses to the same magnitude by taking the logarithm of each loss. This penalizes differences in policy predictions more than MAE, yielding eventual policy predictions that follow the MCTS' policy values closer.

3) Exploring Learning Rates: The many strategies of chess suggest there are many local minima that can be found when

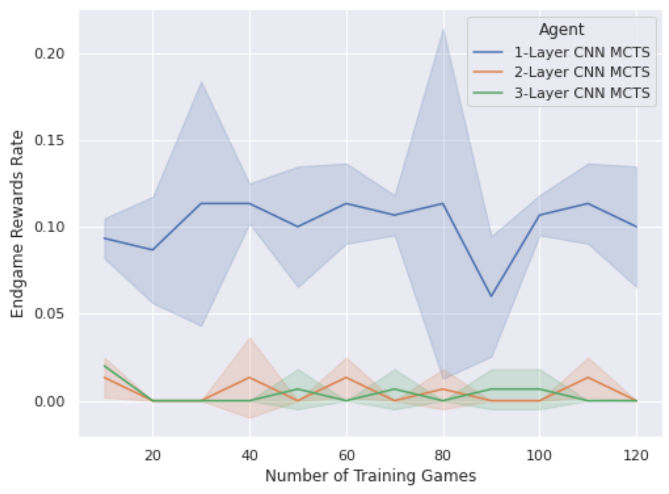

Fig. 4. Effect of the number of CNN layers on the endgame rewards (checkmate) rate. 1-layer CNN MCTS performed best over deeper CNNs. This suggests that more complex networks require significant training time before they can update their weights adequately.

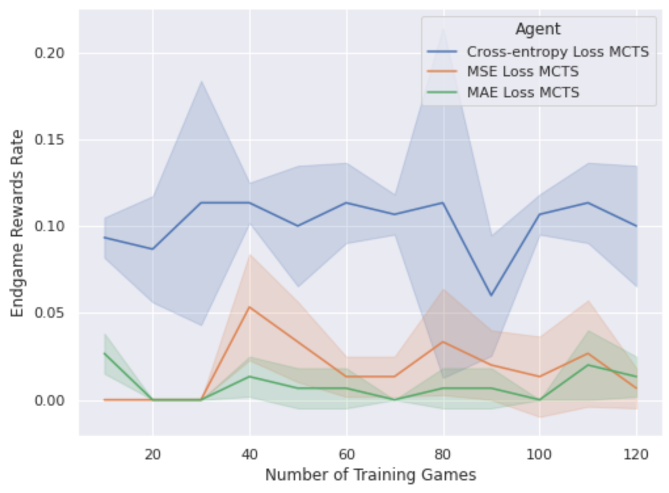

Fig. 5. Effect of policy loss types on endgame rewards (checkmate) rate. Cross-entropy loss performed best in learning Q-values and learned the small probabilities better, which can be neglected when using MSE or MAE. Tests were performed on the 1-Layer CNN MCTS

playing a game, which can also be seen in the mathematical representations. During training, it is common to see the training loss fluctuating without decreasing over time, implying that the network is unable to discover the global minimum.

When lowering the learning rate during training, the training loss was seen hover at the same loss indefinitely. These observations point to the need for dynamically changing the learning rate as training progresses. In our research, we train our DDQN agent using stochastic gradient descent optimizers, but have also experimented with adaptive learning rate optimization algorithms like Adam optimizers. While the latter optimizer was hypothesized to outperform SGD optimizers, it did not yield advantageous results. Hence, SGD optimizers were used throughout training.

4) Exploring Value and Policy Network Separation: Within the MCTS algorithm, the DDQN's value prediction is used in prescribing values to each move for the tree search, picking the eventual path to explore. After tree search, MCTS' chosen policy is then used to teach DDQN's policy head about the best moves, with the goal of mimicking MCTS. Due to the value and policy head having separate functions, we hypothesize that it may be beneficial to have separate networks for each head. This would allow the networks to learn more specific information.

After experimentation, we found that network separation led to poorer network learning overall and the corresponding agent did not register significant wins over random agents. This suggests that the information learnt in both outputs are in fact closely tied to one another and should not be separated. The value head conveys information on the advantage of a board state and the policy head conveys information on how it could achieve this advantage. Intuitively, the parallel information should rightly stem from the same set of network weights and only diverge at the final output layer as illustrated in Fig. 3.

5) Picking Opponents: The DDQN is best trained against a random agent in this low-training environment. Following our experimentation in capture chess, the opponent of our agent was replaced with a more competent agent after it was able to checkmate above the 10% level. Doing this however resulted in the agent 'forgetting' how to win against all agents it trained against. When implementing a self-play mechanism similar to Capture Chess, we observe sharp overfitting towards playing against an older version of itself. This could indicate that the framework easily overfits to the opponent it plays against. Additionally, this shows that far greater training periods are required for learning against a competitive opponent.

V. Demonstration Learning with Semi-Supervised Methods

After exploring MCTS in a regular DDQN, we still sought to create a more competent chess engine. With this aim, semi-supervised methods were investigated to create more reliable and capable agents. This would shift our model away from pure RL and towards the middle of the spectrum with supervised models.

Two kinds of semi-supervised models were tested. The first approach looked at warm starting MCTS with a competent value network. DDQN-MCTS contained a circular process of using its value network to improve the tree search, to find the best moves to improve the value and policy network. Having poor values would lead to poorly selected moves, resulting in an inability to learn and therefore reinforcing poorer values. Introducing this warm start would provide expert feedback to DDQN as to what the good moves are. Secondly, a supervised learning on expert games was carried out to explore the potential of utilizing behavior cloning and DAgger algorithms.

A. Demonstration Learning

The value head of our DDQN was replaced with an expert agent. Two expert agents were used:

- StockFish - An expert open source chess engine, based on hard-coded rules and years of community contributions.
- Oracle - From scratch self-trained neural network valuation engine based on historical chess games of all levels.

The training dataset for Oracle was produced by labeling game states with the reciprocal value of the number of moves away from checkmate.

Stockfish's evaluation engine provided values in centipawns or moves to checkmate depending on the board state. This evaluation was converted into a normalized value using

$$V(r) = \begin{cases} M \cdot r(\text{centipawn}), & \text{if } r \in \text{centipawn} \\ e^{-n_{\text{mate}}/\gamma} & \text{if } r \in \text{mate} \end{cases}$$

where $M = 1 \times 10^{-4}$, n_{mate} is the number of moves to checkmate, r is the value of the board and the decay rate $\gamma = 50$ was used to ensure mate in two or three also obtain sufficiently important rewards.

B. Behavior Cloning

Additionally, behavior cloning was used to assess our DDQN's capacity to learn and mimic expert chess engine bots. Stockfish was used to generate expert 100,000 expert trajectories. Two neural networks were used to clone Stockfish's decision process; the first can be seen in Fig. 3, and the second network contained 6 convolutional layers and 4 linear layers amounting to 3 million trainable parameters.

We seek to train a neural network that could learn the best move to make from the possible action space of 4096 moves.

C. Results & Discussions

The impact of different semi-supervised methods can be seen in Fig. 6. Stockfish MCTS achieved the highest mean win rate of $15.3\% \pm 6.4\%$ at 20 games with a mean advantage of 2.98 ± 0.72, with a $< 1\%$ loss rate. Oracle appeared to follow a similar behavior but with lower rewards of $4.76\% \pm 4.16\%$. This pattern was thought to be a local minima that performed well, before the learning rate pushes it find the global minima.

Testing these semi-supervised methods showed that demonstrations can help the neural network develop a sense of the right moves, but it depended heavily on the learning rates which could quickly push the networks into local minima. Since demonstration learning uses a fixed reference value network, the lack of adaptability might also cause more local minima. Regular MCTS with no demonstration showed more stable results above zero despite no performance increase. These results are seen in Fig. 6.

Additionally, we pilot-tested behavior cloning. Both simple and complex neural network structures were unable to demonstrate effective learning of Q-values taken from Stockfish's expert trajectories. However, they were able to learn to mimic Stockfish's evaluation. The value network's parameters were also used as a warm start for the policy network, but did not improve the loss. This steep network learning curve prevented additional research into more comprehensive agents such as DAgger.

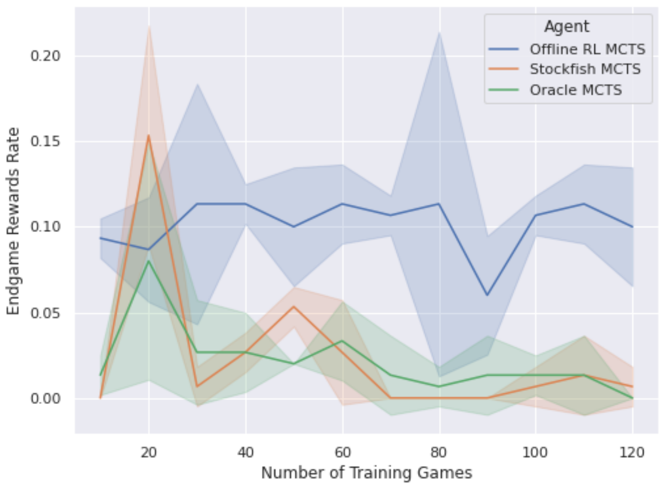

Fig. 6. Different Semi-Supervised methods were experimented with. Stockfish MCTS achieved the highest mean endgame rewards rate at 20 games before decreasing further. Oracle appeared to follow a similar behavior but with lower rewards. This implies that the expert recommendations initialized the agent with a good local minima, before the learning rate began pushing it to find the global minima.

VI. Further Work

A. Computational Complexity

Computational complexity posed the most prominent challenge in our work. Each move search in the MCTS runs for 100 simulations and drastically increases the training time for each game. Even with an enforced 75-move maximum, training games can take up to 95s±10s per game. Although our research aims to investigate lean architectures and not build a competitive chess agent, we are interested in our agent's overall game performance when trained for similar time magnitudes to AlphaZero. Future iterations of this work should connect this engine to multiple processors for distributed training.

VII. Conclusion

Our research demonstrates how lean neural network architectures have the potential to support a Double Deep Q-Network in learning viable chess strategies from a Monte Carlo Tree Search algorithm. Within just 120 training games, the DDQN-MCTS agent picks up both capture and checkmate strategies to a reasonable degree, yielding a $10.2\% \pm 3.8\%$ win rate and mean 3.4 ± 1.1 piece point advantage over random agents. Additionally, utilizing demonstration learning observed a $15.3\% \pm 6.4\%$ endgame rewards rate and a material advantage of 2.98 ± 0.72.

We believe that the next best steps to improve the lean CNN architecture is to implement sparse CNNs structures to learn action policies effectively [5]. This issue lies in a key research area of modern-day machine learning but holds much promise for these leaner networks, that can be augmented with behavior cloning and DAgger. Finally, we acknowledge DeepMind's monumental achievement of the AlphaZero framework, which has not only expanded our view of RL's profound potential, but has also given us a new perspective on human cognition.

References

[1] Silver, David; Hubert, Thomas; Schrittwieser, Julian; Antonoglou, Ioannis; Lai, Matthew; Guez, Arthur; Lanctot, Marc; Sifre, Laurent; Kumaran, Dharshan; Graepel, Thore; Lillicrap, Timothy; Simonyan, Karen; Hassabis, Demis (December 5, 2017). "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm"
[2] Gibney, Elizabeth. "Self-taught AI is best yet at strategy game Go". Nature News. October 2017. Retrieved 10 May 2020.
[3] Groen, Arjan. "A collection of Reinforcement Learning Chess Algorithms". Accessed https://github.com/arjangroen/RLC
[4] Claude Shannon (1950). "Programming a Computer for Playing Chess" (PDF). Philosophical Magazine. 41 (314).
[5] Baoyuan Liu et al. (2015). "Sparse Convolutional Neural Networks". Computer Vision Foundation.

Appendix

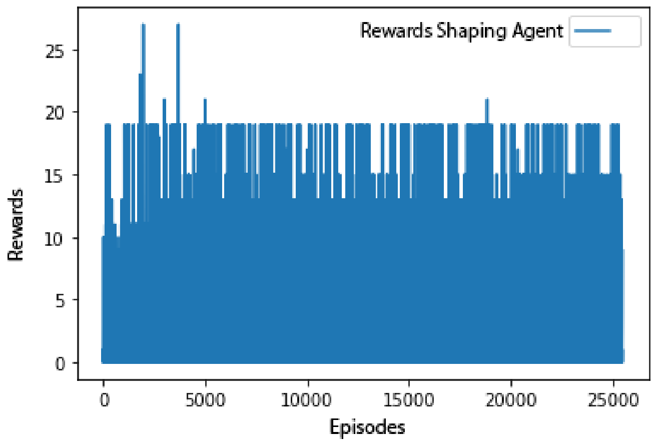

Fig. 7. Rewards curve for agent trained using rewards shaping function of $R = \text{material value} + 10 \cdot \text{check} + 20 \cdot \text{checkmate}$. Here it can be seen that checkmate was achieved many times within the training. This gave alot of promise for rewards shaping, however in testing the agent was unable to display significant competence, displaying 0.1% win rate.

Algorithm 5: STEP

```
Input: action
Output: Reward, Board$_{t+1}$
1 Board$_{t+1}$ ← Push(action)
2 Reward ← isCheckmate(Board$_{t+1}$)
3 return Reward, Board$_{t+1}$
```

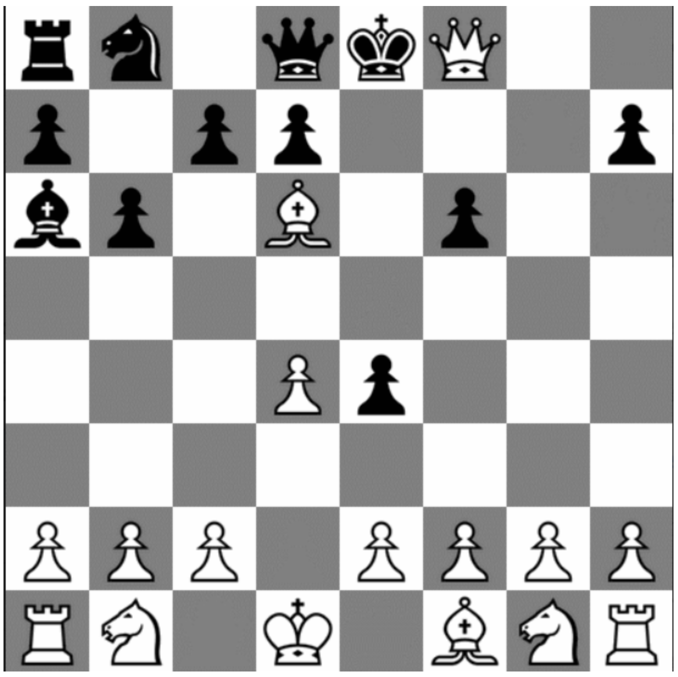

Fig. 8. Offline RL MCTS against a random agent, example game with checkmate. It can be seen that a scholar's mate was achieved. In this game, the agent took multiple moves to edge the queen closer to the king, making it an inefficient player. Checkmate achieved in 10 moves. PGN: [1. d4 Na6 2. Bh6 Nb8 3. Bxg7 f6 4. Bxf8 Nh6 5. Qc1 e5 6. Qxh6 e4 7. Bd6 b6 8. Kd1 Ba6 9. Qg7 Rf8 10. Qxf8]

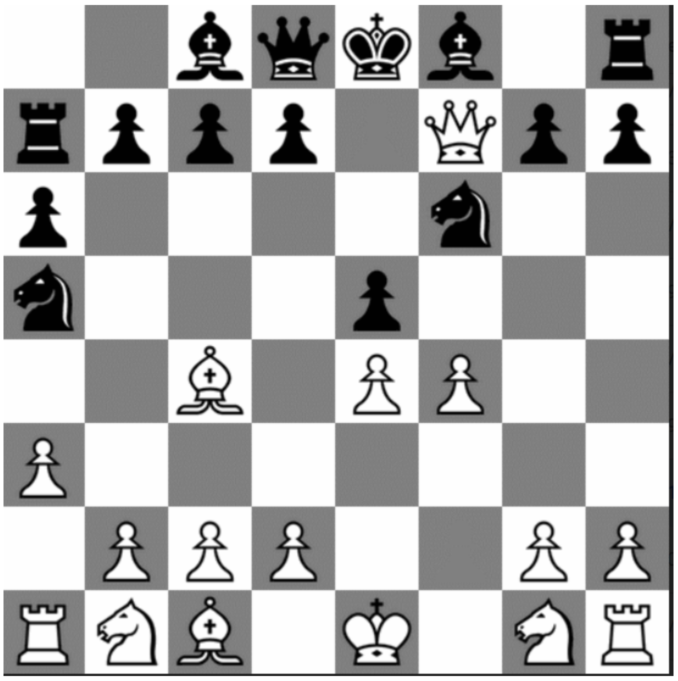

Fig. 9. Stockfish MCTS against a random agent. Here, it is seen that our agent is much more efficient, achieving checkmate in seven moves. Scholar's mate can also be seen again, this could be because it is the most efficient way of reaching the reward against a random agent. PGN: [1. f4 Nf6 2. a3 Nc6 3. e4 a6 4. Be2 Na5 5. Bc4 e5 6. Qh5 Ra7 7. Qxf7]

Demarcation of Work

Algorithm Development

- Capture Chess - Raphael, Shaun
- MCTS - Raphael, Shaun
- Semi-Supervised MCTS - Raphael, Shaun

Writeups

- Project Presentation - Raphael, Shaun
- Project Final Report - Raphael, Shaun
- Project Midterm - Raphael, Shaun, Peijun
- Project Proposal - Raphael, Shaun, Peijun

Exploration

- Exploring CNN Architectures - Raphael, Shaun, Peijun