Efficient Learning for Checkers AI in Non-stationary and Stochastic Environments

Shih-Yi Tseng [*1] **Laurence Calancea** [*2]

Abstract

In this paper we construct and test the performance of reinforcement learning and adversarial search AI that play Checkers. We explore the consequences of having the agents learn a non-stationary environment and a stochastic environment. We study how different state representations, learning algorithms and value function approximators affect stable performance in standard games and speed of recovery from changes in game rules. To further allow efficient learning, we develop an algorithm for automatic detection of sudden changes in the environment and dynamically adjust learning parameters. We also test the robustness for both online RL agents and tree search-based Alpha-beta pruning agents, and find that RL agents could perform more robustly in stochastic environment as opposed to non-learning agents that depend on planning using deterministic model of the environments. Our work has demonstrated different components that may influence and improve online RL agents in non-stationary and stochastic environments.

1. Introduction

Checkers, is a board game for two players played on the board shown in Figure 1. With a search space of $5 \cdot 10^{20}$ (Schaeffer et al., 1996), this game is the largest weakly-solved game that remains to be fully solved (Schaeffer et al., 2007). For this reason, AI developments for this game are still taking place. In this project we create learning agents that play this game, compare their performance against Alpha-beta pruning agents and in the process study certain properties of the game and of the agents.

[*]Equal contribution [1]Harvard Medical School, Boston, MA, USA [2]Harvard University, Cambridge, MA, USA. Correspondence to: Shih-Yi Tseng <shihyi_tseng@g.harvard.edu>, Laurence Calancea <calancea_laurentiu@college.harvard.edu>.

Submission for Stat 234.

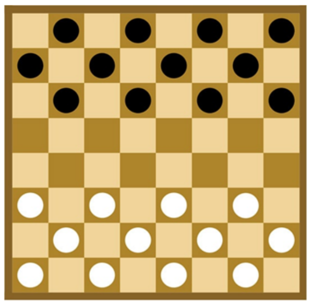

Figure 1. Checkers board with pieces: (Figure from `https://www.regencychess.co.uk/draughts_rules.html`)

1.1. Rules and game variants

The standard rules of checkers are the following:

1. Pieces move only one step diagonally forward. If a piece jump over an opponent's piece it captures it.
2. Pieces are forced to capture any opposing piece they can capture. (**Forced Captures**)
3. If a piece reaches the last row of the board, it becomes a king. Kings can also move backwards.
4. The game ends with a win for the player that eliminated all opponent's pieces.

We explore the ability of RL agents to learn in an environment where the rules change to **LoseToWin**.

1.2. Alpha-beta agents vs. RL agents

Alpha-beta (AB) agents are look-ahead tree search algorithms (Samuel, 1959) that use the alpha-beta pruning optimization of Minimax search. AB agents use custom heuristics to evaluate board positions several steps ahead and make decisions based on that. The RL agents we use are based on Q-learning and temporal difference learning algorithms. As it is infeasible to create a tabular representation for all states and transitions, we reduce the state space using hand-crafted features. To update the values of the game we use linear and non-linear approximations of the states. One notable example is the use neural networks, such as multi-layer perceptron (MLP) for these approximations (Lynch & Griffith, 1997; Caixeta G.S., 2008).

1.3. Problem statement

Efficient learning in non-stationary environments (Sec. 3)

We explore the ability of RL agents to learn in a non-stationary environment by reverting the outcome of the games in the middle of training and comparing performance recovery of RL agents.

Robust learning in a stochastic game environment (Sec. 4)

We explore the ability of RL agents to learn in a stochastic environment. Our goal is checking whether RL agents are more robust than deterministic agents to stochasticity.

2. Baseline RL agents for playing checkers

2.1. Online learning agent SARSA

The SARSA agent learns action values relative to the policy it follows, which makes it an on-policy learning algorithm (Sutton & Barto, 2018). For the control problem we use generalized policy iteration. The SARSA control algorithm adopted in this project using differentiable function approximator (semi-gradient SARSA) is presented below:

Algorithm 1 Semi-gradient SARSA

```
input: a differentiable action-value function parametrization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathcal{R}^d \to \mathcal{R}$
parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
initialize: action-value function weights $\mathbf{w} \in \mathcal{R}^d$ arbitrarily except $\hat{q}(terminal, \cdot) = 0$.
for each episode do
    initialize: $S$
    choose: $A$ from $S$ using policy derived from $\hat{q}$ (e.g. $\epsilon$-greedy)
    while episode is not over do
        take action: $A$, observe $R$, $S'$
        choose: $A'$ from $S'$ using policy derived from $\hat{q}$ (e.g. $\epsilon$-greedy)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[R + \gamma \, \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\right]$
                $\nabla \hat{q}(S, A, \mathbf{w})$
        $A \leftarrow A'$; $S \leftarrow S'$;
    end while
end for
```

Algorithm extracted from (Sutton & Barto, 2018)

State representation

The board position is encoded by an 8×4 matrix. The board position occupied by pawns and kings of the player are denoted by 1 and 2, whereas the those occupied by the opponent's pawns and kings are denoted by -1 and -2. Empty positions are denoted by 0.

To simplify the board position to construct the Q functions, we obtained two sets of features from the raw board position before (i.e. current state s) and after a move a.

1. Simple features: basic summary statistics of the pre- and post-move board position, including the difference in number of pawns and kings for the agent and the opponents, as well as the number of possible attacks post-move.

2. Augmented features: 33 hand-crafted features about specific local/global board positions from literature (Samuel, 1959; Lynch & Griffith, 1997). Some examples are given:

    (a) *PieceThreat* - The number of pieces under threat from the opponent.

    (b) *TotalMobility* - Credited with 1 for each square open to the opponent.

    (c) *Exposure* - Credited with 1 for each piece that is flanked on each side of either diagonal by empty squares.

The Q value is obtained using linear and non-linear function approximators (MLPs) applied on the features. In some MLP-based models, the raw board position is vectorized and supplied to the agent as part of the state-action features. More details are provided in (Sec. 3.1).

Reward

When training the RL agents, each game is considered as one episode. At the end of each episode, the agent is given $+500$ for winning, -500 for losing the game, 0 for draws (maximum number of moves reached, which was 500 in our case). To facilitate learning, between each state transition within a game, we also supply the agent with small intermediate rewards: $0.1\cdot$ decrease in opponent's pawns $+0.2\cdot$ decrease in opponent's kings $-0.1\cdot$ decrease in agent's pawns $-0.2\cdot$decrease in agent's kings. To encourage the agent to end each game sooner, we also give a living reward -0.1 for every unrewarded move. In the case of **LoseToWin** game, we revert the sign of the intermediate rewards as well.

2.2. Opponents

1. **Random agents** operate by taking a random action out of the available actions with an equal probability.

2. **Alpha-beta** pruning agents (Alg. 2). In the remainder of the paper we index AB agents of various $depths$ as AB($depth$). The AB agents we use employ the following heuristic evaluation function for terminal states or nodes of $depth = 0$:
   (a) $+500$ for a win,
   (b) -500 for a loss,
   (c) 2·difference in kings $+1$·difference in pawns non-terminal states otherwise.

   We train the RL agents to play against the random agent, AB(1), AB(2), and AB(3) by randomly selecting one of the four opponents at the start of each game, unless specified otherwise. To create a set of diverse ”initial states”, we also let both the RL agent and the opponent to randomly select actions for n number of move, n as an integer randomly chosen between 0 to 3. Thus the RL agents can experience a variety of starting positions during learning.

   By examining the algorithm, we also note a crucial difference between tabular methods and RL methods. Using an AB agent of $depth > 1$ explicitly gives access to the state transition function, which can only be approximated by the RL agents. More concretely an AB(3) agent's decision is determined by the heuristic value of the states it can reach in 3 moves, which are obtained directly using the knowledge of deterministic transitions of the environment to construct the search tree. On the other hand the RL algorithms do not have explicit access to these transitions. This means that a converged value function would need to encode this information as well, in addition to the usefulness of performing each action.

Algorithm 2 Alpha-Beta Pruning
```
function alphabeta($node, depth, \alpha, \beta,$
        $maximizingPlayer$)
  if $depth = 0$ or node is a terminal node then
    return heuristic value of node
  end if
  if $maximizingPlayer$ then
    $value = -\infty$
    for child of node do
      $value = max(value,$ alphabeta$(child, depth - 1, \alpha, \beta,$ False$))$
      $\alpha = max(\alpha, value)$
      if $\alpha \geq \beta$ then
        break
      end if
    end for
    return $value$ { $\beta$ cutoff }
  else
    $value = +\infty$
    for child of node do
      $value = max(value,$ alphabeta$(child, depth - 1, \alpha, \beta,$ True$))$
      $\alpha = min(\beta, value)$
      if $\beta \geq \alpha$ then
        break
      end if
    end for
    return $value$ { $\alpha$ cutoff }
  end if
end function
```
Algorithm extracted from (Norvig & Intelligence, 2002)

2.3. Performance of baseline SARSA agents

Before we test agents in the non-stationary game environment (Sec. 3) and the stochastic game environment (Sec. 4) we first obtain the best parameters for our agents using grid search. Fig. 2 shows the fraction of wins and of draws for the SARSA agents that use simple features and augmented features with linear functions to approximate the state-action value. For SARSA agent using simple features (basic summary statistics), we note that the amount of information the agent has is almost equivalent to what an AB(1) agent has access to in its heuristic value evaluation, so the RL agent's performance against AB(1) reaches around 50% as expected. As we increase the complexity of the state-action representation by using the augmented features, the performance of the RL agent against AB(1) increases to $80-90\%$. We also note that the drastic increase in the fraction of draws against AB(2) and AB(3) agents reflects that the SARSA agent is still unable to beat the opponents, but it loses less frequently.

Fig. 3 shows the scores obtained by SARSA agents with different initial parameters playing against the AB(1) on test games. During these test games, the weights of the Q-function are frozen. The score is assigned as follows: 1 for wins, 0.5 for draws and 0 for loses. To smooth the score for the learning graph we performing a rolling mean of the score values with a window of 10 games. From the results we decide to continue the experiments with a discount $\gamma = 0.7$ and a learning rate of $\alpha = 0.001$.

3. Efficient learning in non-stationary environments

In this experiment we study the ability of agents to learn in an environment that is non-stationary. Since the values of a standard checkers game are constant across plays, we artificially create non-stationarity by changing the rules of the game at the 1000th game, after which we play for an additional 1000 games. In particular, we alternate between the standard rules and the **LoseToWin** rules, in which the rewards and final state values are negated for the RL agents

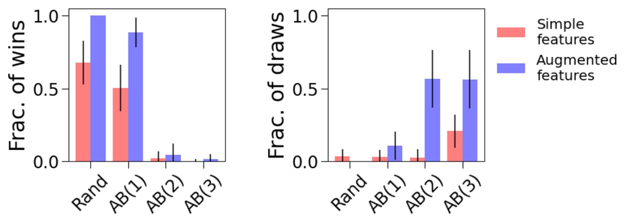

Figure 2. Performance of baseline SARSA agents

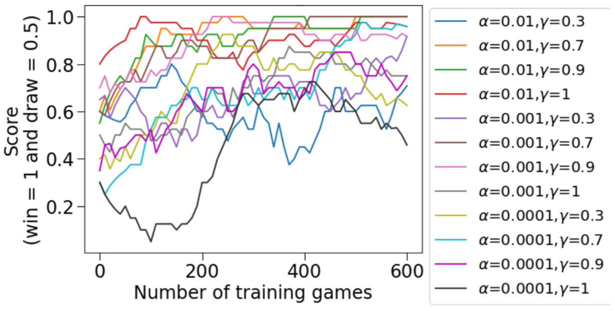

Figure 3. Parameter search: score on test games

and the heuristic values are negated for the AB agents.

3.1. Model Variants and their Performance

We test different variants of the RL models to explore the efficiency of learning in these variants. The variants are as following:

1. Learning algorithm: TD(0) vs. TD(λ). We hope to see if agents with TD(λ) can learn and adapt faster after a rule switch.

2. State representation: simple features vs. augmented features. In models with MLP, we also directly provide the agent with vectorized raw board positions, or raw board position plus augmented features. We aim to study how the complexity of state representation affects learning or generalization across different game rules.

3. Function approximator: linear function vs. MLP (2-layer, with 128 and 64 hidden units in each layer). We would like to see if using more complex, non-linear functions can improve the performance and learning efficiency.

Fig. 4 and Fig. 7 show the scores during learning (combined all training games against the 4 types of opponents) and upon testing (separating games against the 4 opponents) for different model variants, first trained on 1000 standard games and then switched to 1000 LoseTowin games. Fig. 5 shows the changes of model weights for one RL agent using linear approximator with augmented features (33 weights) in one of the rule-switching experiments. For each model variants, 10 agents with different initialization of the weights were trained and tested. RL agents using linear function approximators with augmented features have the highest performance before rule switch, and recovered from the switch after 200 games. Surprisingly, we do not observe differences in learning speed between models using TD(0) vs. TD(λ) learning algorithms. Models using MLPs with augmented features or augmented features plus raw board positions achieve slightly lower performance before rule switch, but are unable to recover to the pre-switch performance after rule switch. Models using linear function with simple features as well as models using MLPs with raw board positions show the lowest stable performance in standard games, and fail to recover after switching to LoseToWin games.

Interestingly, we observe asymmetric stable performance of these models in standard games vs. LoseToWin games. For example, in Fig. 7 we see that the post-switch stable performance for models with linear approximators and augmented features against AB(2) is higher than their stable performance in pre-switch periods. To ensure the differences in the asymmetric performance do not interference with their recovery from rule switch, we perform another set of experiments where we train the models first on LoseToWin games and switch to standard games, as shown in Fig. 7. Comparing the stable performance of both rules pre- and post-switch, we see that the above-mentioned models that fail to switch in standard-to-LoseToWin experiments indeed fail to achieve the stable performance in the LoseToWin games when they are trained from scratch, and vice-versa.

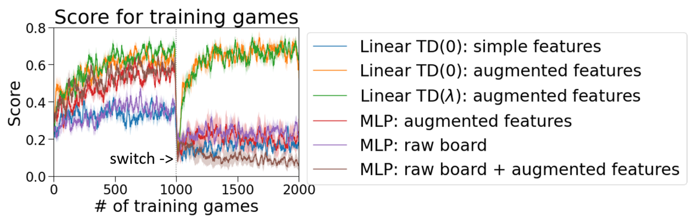

Figure 4. Standard game rule to LoseToWin

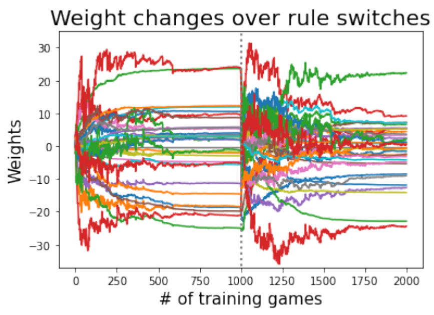

Figure 5. Weight changes over rule switch

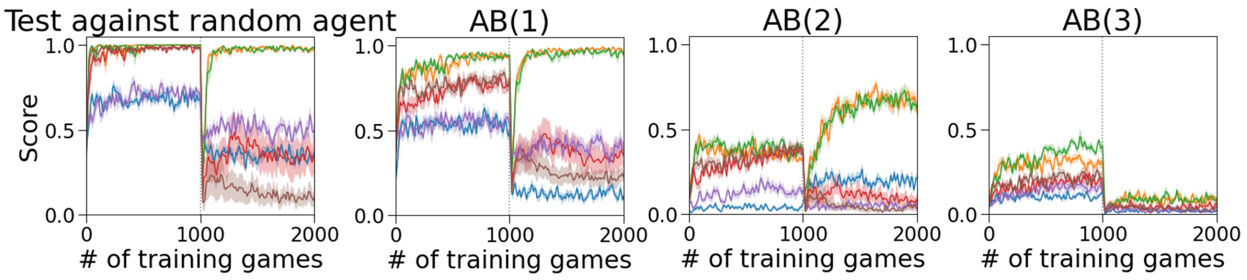

Figure 6. Testing results across all agents for the transition from the Standard to the LoseToWin game

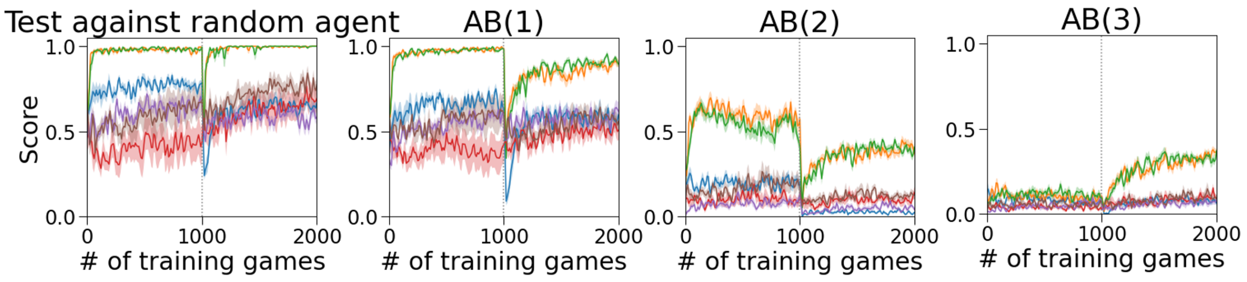

Figure 7. Testing results across all agents for the transition from the LoseToWin to the Standard game

3.2. Automatic Detection and Dynamic Adjustment of Learning Parameters in Non-stationary Environment

In true online learning setting, the agent normally starts with high learning and exploration rate, and then gradually decreases those parameters overtime as the agent learns so that the policy converges to near-deterministic as learning slows down. For our training, we also deploy learning rate α and exploration rate ϵ (for the ϵ-greedy action selection) scheduling so these values decay exponentially over training games. However, this can be disadvantageous in non-stationary environment if the agent is unaware of changes in the environment and cannot ”re-initiate” learning.

In previous experiments for switching game rules, we manually tune the learning rate and exploration rate to initial values to allow re-learning after rule switches. In this section, we aim to automate this process. For an agent agnostic to explicit changes in the environment, it is possible that the agent can still detect the non-stationarity of the environment by monitoring its learning history. We develop a way to automatically detect major changes in the environment based on monitoring the values of average episodic temporal difference over the recent learning history, and then to ramp up learning rate and exploration accordingly to “re-learn”.

Specifically, we allow the agent to maintain the memory of the average values of temporal differences for all steps in each episode (game) over a long time window (e.g. 100 previous episodes) and a short time window (e.g. 10 previous episodes), and compare the mean temporal difference for all episodes over the short time window μ_s to the distribution of the temporal differences M over the long time window. When the value of μ_s falls out of a certain interval of M parametrized by a threshold t, such as mean$\pm t\cdot$SD, it is indicative of some abrupt changes in the rewards or the environment structure. We then increase the value of learning rate α and exploration rate ϵ by either a fixed factor or a scaled factor proportional to the amount of deviation. We call this mechanism ”double window error monitoring and dynamic learning adjustment”.

Fig. 8 shows how the average TD values over long time window and the distribution change over training games relative to the average TD values over the short time window. In this case, the threshold is set at $0.5 \cdot SD$. After rule switch at game 500, the mean TD over short time window falls out of the interval, and the learning rate (Fig. 9) and well as the exploration rate increase accordingly. In Fig. 10, we compare the recovery of performance of the same RL agents (linear function plus augmented features) with different methods of learning parameter adjustment upon rule switch from standard games to LoseToWin games. Models with automatic error detection and dynamic adjustment of learning parameter recover from rule switch with a slightly slower but comparable speed compared to the ones with manual adjustment, whereas models without any learning parameter adjustment fail the re-learn after rule switches.

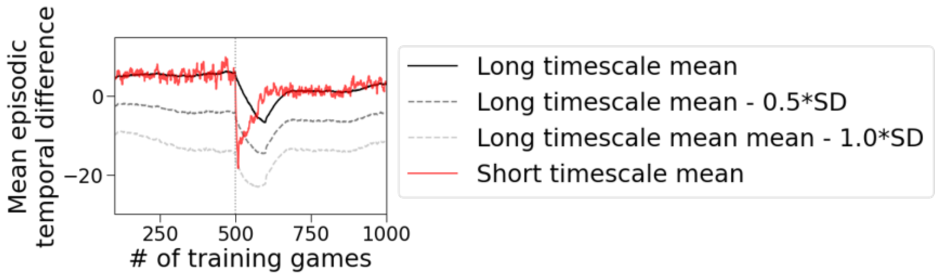

Figure 8. Changes in mean episodic temporal distance over long vs. short time window in rule-switching experiments

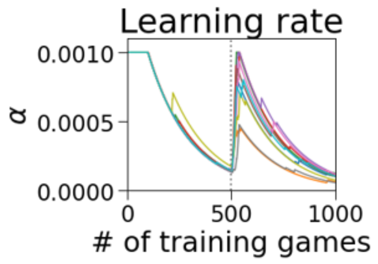

Figure 9. Dynamic learning rate adjustment for 10 models with different initialization

4. Robust learning in a stochastic Game Environment

In a stochastic game the next state of an agent is selecting randomly with probability p out of the available states.

This setup aims at simulating the real-life scenario in which agents learn in a deterministic environment but are then made to operate in the real world where all actions have a latent stochasticity.

We explore how the performance of the SARSA agent changes in this environment: We train and then test SARSA agents in environments with random move probabilities from 0.2 to 0.6 with 0.1 increments. During training we sample a random agent from: the random agent, AB(1), AB(2), AB(3). Every 100 games we stop training and test 10 times against each opponent. When not learning, the RL agents are deterministic which means that in order for us obtain an average performance across the 10 testing games we need to have the agents start in different states. To do that we initialize the board by making $k \leq 3$ random moves by each opponent (similarly to how we do in Sec. 3). Afterwards we let our agents play on the resulting state as before.

In order to obtain confidence intervals for the score we repeat the above process 10 times with different random seeds. Finally these allow us to obtain the plots in Fig. 11, Fig. 13, Fig. 13.

We conclude the following:

1. The agent's ability to learn is not hindered by the stochasticity of the environment.
2. If an agent perform well, increasing the stochasticity decreases its performance and vice-versa. In the limit of $p \to 1$ the score will be tend towards 0.5, representing the probability of one random agent winning against another one (provided there are no draws).
3. One way to determine if the RL agent is more robust is finding an equally strong AB opponent in a deterministic setting, introducing stochasticity and observing the change in performance. Unfortunately it is difficult to find such an agent.

5. Discussion

In this project, we aim to build RL-based checkers AI and study the learning efficiency in an online setting when facing non-stationary or stochastic environments. Although these two scenarios may not appear realistic in terms of playing checkers, we hope to use these models as a proof-of-concept demonstration and extend the work to other real-life application of online reinforcement learning task.

In the first part, we introduce non-stationarity by alternating the standard games and LoseToWin games during learning and study what factors can affect the recovery of performance upon a rule switch. RL agents using linear function approximators plus hand-crafted augmented features adapt to rule switch with a reasonable speed and achieve better stable performance compared to models using simple summary statistics of the board positions as those commonly used for construction of heuristic evaluation functions for tree search-based algorithms such as Alpha-beta pruning agents used in this study as opponents. To our surprise, changing the learning algorithms from $TD(0)$ to $TD(\lambda)$ wouldn't improve the learning speed. We also attempt to test if increasing the complexity of the function approximators with neural networks can lead to higher performance, since the

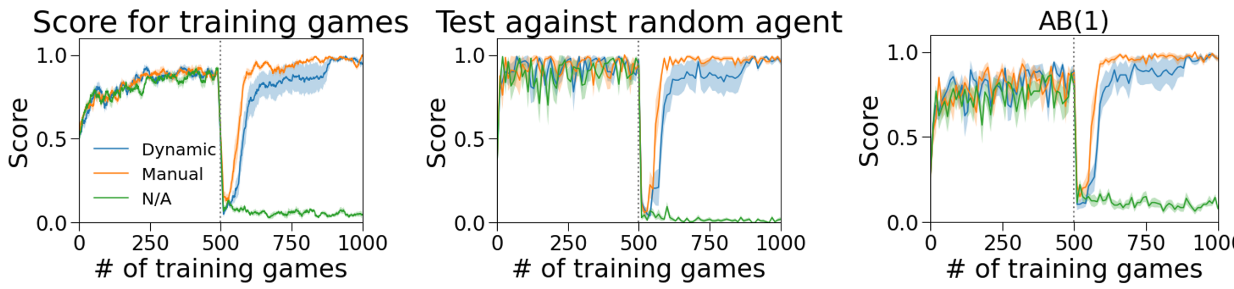

Figure 10. Comparison of performance recovery with different adjustment methods for learning parameters

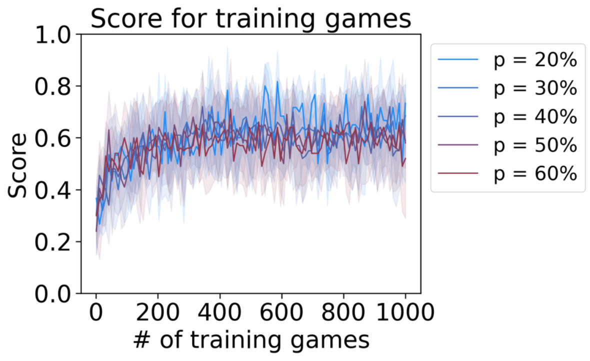

Figure 11. Training performance graph for the stochastic environment

networks may learn to extract features directly from the raw board positions as more useful state representations, beyond what have been proposed in the literature as hand-crafted features. However, our RL agents using MLPs with raw board positions do not achieve superior stable performance prior to rule switches, as well as failing to recover from the switches. There are several possible reasons: (1) the number of weights in MLPs is 2 orders of magnitude larger than the number of weights in linear function approximators. We didn't find the ideal hyper-parameters for the training, such as learning rate, regularization, or total number of training episodes to achieve asymptotic performance. (2) The gradient from each step of state transition within an episode is too noisy, leading to unstable update of the weights in a non-convex regimes. (3) MLPs using vectorized raw board position is not an efficient way for extracting useful features; other network architectures, such as convolutional neural networks (CNNs) may be advantageous as they are more powerful models for extracting features from 2D images. Indeed, other AIs for board games, such as AlphaGo or Atari agents, use CNNs as part of their feature extractor for finding efficient state representations (David Silver, 2016).

Besides, we consider a more realistic scenario for online learning in non-stationary environment in which the agents use their learning history to detect changes in the environment, and respond with increasing learning rate and exploration. Based on these principles, we develop an algorithm that allows the agent to keep track of the temporal differences over learning history with different time windows and dynamically adapts learning parameters when recent changes in TD errors are detected, indicative of changes in the environment. With this algorithm, our agents are able to adjust their learning parameters upon a rule switch and learn to play the new games without any manual tuning. One caveat of this algorithm is that it may only work well when the non-stationarity comes in the form of sudden changes; if the changes in the environment are gradual and subtle, our algorithms may fail to detect those changes.

In the second part of our work, we explore the robustness of RL agents vs. the Alpha-beta pruning agents in stochastic environments. As we expected, adding stochasticity into the environments brings the model performance closer to that of a random agent. However, the stochasticity is more detrimental for models that rely heavily on the knowledge of the deterministic state transition in a noise-free environment, such as AB(3) that plans three steps ahead before evaluating the values of each move, whereas the RL agents learn to estimate a Q function for current state-action pairs which implicitly incorporates the randomness of the environment through experience. Thus the performance of RL agents against AB(3) increases as the stochasticity increases. Our experiments demonstrate the potential advantage of using RL-based agents in real-life, noisy environments compared to a well-engineered, non-learning agent built upon knowledge of noise-free, deterministic model of the environment.

6. Conclusions

1. We have implemented online learning reinforcement learning agents to play checkers in non-stationary and stochastic game environments.

2. Agents with linear function approximators on augmented features adjusted to rule switches better than the model using simple features or the ones with MLPs.

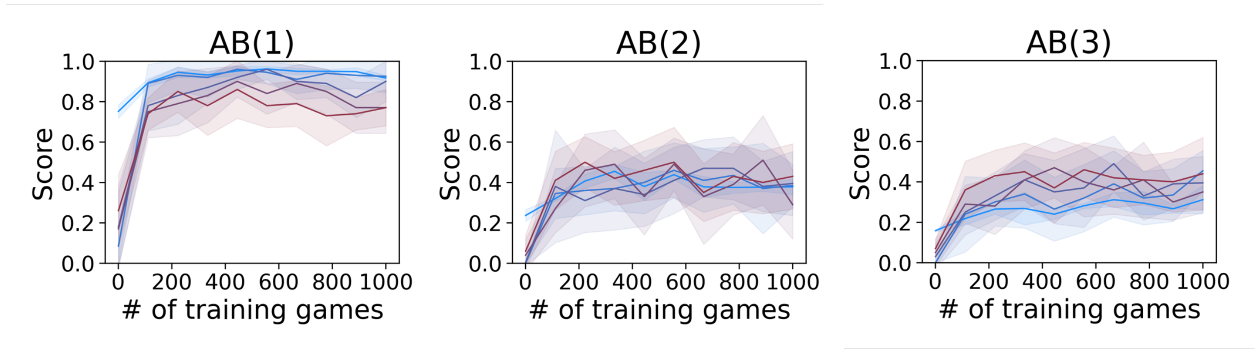

Figure 12. Testing performance graph for the stochastic environment

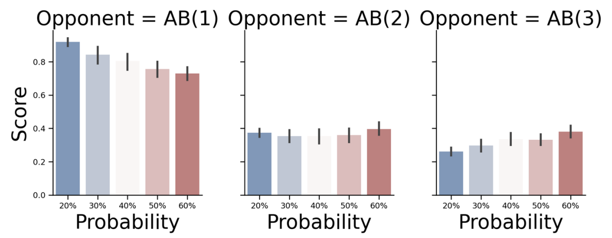

Figure 13. Average testing performance boxplots for the stochastic environment

3. We have developed the "double window" error monitoring mechanism for automatic learning parameter adjustment in non-stationary environment with abrupt reward change.

4. The agent's ability to learn is not hindered by the stochasticity of the environment, while the score tends to 0.5 as p increases.

5. One way to determine if the RL agent is more robust is finding an equally strong AB opponent in a deterministic setting, introducing stochasticity and observing the change in performance. We will attempt this in future work.

6. Other future directions also include using convolutional neural networks for feature extractions from the raw board positions, potentially pre-trained on expert moves in various game variants to learn generalizable features.

Software and Data

For this project we used a python implementation of a checkers board by (Ragusa, 2017) and a starting implementation of the AB and Q-learning agents by (Raval, 2018).

All modifications and additions to the code were done in python with the use of the **numpy** library for vector manipulations and **autograd** library for gradient control of the MLP. The code can be found in this Github repository: `https://github.com/sytseng/STAT234_Project`

The data used for training and testing was obtained by having the agents interact with the game environment.

Acknowledgements

We thank the teaching staff for useful comments and suggestions. In particular we thank professor Susan Murphy for the initial discussion about potential final projects and the subsequent discussion about ways to expand current results in the field. Both our explorations on efficient learning in non-stationary environments and learning in a stochastic environment are due to those fruitful conversations. We also thank Kelly Wang for her review of our outline, multiple meetings and useful suggestions and Peng Liao for his review and helpful questions during the presentation.

References

Caixeta G.S., d. S. J. R. A draughts learning system based on neural networks and temporal differences: The impact of an efficient tree-search algorithm. *Advances in Artificial Intelligence*, 3(3):73–82, 2008.

David Silver, Aja Huang, C. J. M. A. G. L. S. G. v. d. D. J. S. I. A. V. P. M. L. S. D. D. G. J. N. N. K. I. S. T. L. M. L. K. K. T. G. . D. H. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

Lynch, M. and Griffith, N. Neurodraughts: the role of representation, search, training regime and architecture in a td draughts player. In *Eighth Ireland conference on artificial intelligence*, pp. 64–72. Citeseer, 1997.

Norvig, P. R. and Intelligence, S. A. *A modern approach*. Prentice Hall Upper Saddle River, NJ, USA:, 2002.

Ragusa, S. Checkers-reinforcement-learning.

https://github.com/SamRagusa/Checkers-Reinforcement-Learning, 2017.

Raval, V. Checkers-ai. https://github.com/VarunRaval48/checkers-AI, 2018.

Samuel, A. L. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

Schaeffer, J., Lake, R., Lu, P., and Bryant, M. Chinook the world man-machine checkers champion. *AI Magazine*, 17(1):21–21, 1996.

Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P., and Sutphen, S. Checkers is solved. *science*, 317(5844):1518–1522, 2007.

Sutton, R. S. and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018.