Pre-Initializing Q-Learning for Card Games with Large Feature Spaces

by
Calvin Tan

Supervisor: Michael Guerzhoy
April 2021

B.A.Sc. Thesis

Division of Engineering Science
UNIVERSITY OF TORONTO

[This page was left intentionally blank]

Pre-Initializing Q-Learning for Card Games with Large Feature Spaces

by
Calvin Tan

Supervisor: Michael Guerzhoy
April 2021

Abstract

Card games with large feature spaces are often unlearnable by a Reinforcement Learning agent through common techniques like Q-Learning, especially when reward signals are sparse. Gin Rummy is such a card game that does not converge when training a Reinforcement Learning agent. An approach to allow an agent to learn these card games is though Pre-Initialization. Deep Q-Networks (DQNs), a branch of Q-Learning, consists of a Neural Network which can be Pre-Initialized through Supervised Learning. First, a Deep Neural Network model was trained on a supervised task of predicting a baseline agent's action provided a state. The goal of the model is to learn lower-level features of the card game environment based on the baseline agent policy. Once the model sufficiently learned this policy, the model's parameters are used as the initialization for the Reinforcement Learning agent. This, in theory, will allow the agent to focus on improving the learned features and policy instead of trying to learn both simultaneously. Multiple experiments were performed to the Pre-Initialized DQN agent including but not limited to layer freezing, varying reward structures, additional top layers, and additional action biases. Finally, these trained agents were evaluated against the baseline agent, and itself, both prior and during the training process, to assess the validity of the approach of Pre-Initializing the Deep Q-Network. Through these experiments, a final agent successfully improved during the DQN training process, supporting this approach to address Q-Learning convergence.

Acknowledgements

First, I would like to thank my supervisor, Professor Michael Guerzhoy, for the endless advice and guidance throughout this thesis. Our weekly meetings helped reinforce my knowledge and understanding of different approaches in tackling this thesis.

Second, I would like to thank Google for providing Google Colaboratory; under no circumstance would I have been able to accomplish so many experiments without the GPU resources.

Third, I would like to thank Anthony Hein for their Gin Rummy Python implementation translated from Told Neller's EAAI challenge. Similarly, numerous amounts of gratitude goes towards RLCard for their Gin Rummy agent DQN Implementation.

Last, I would like to thank my family and friends for supporting me the past five years. I thank my parents for providing me such a strong and determined will. I am forever grateful to you for pushing me to never give up. To my brothers and friends, thank you for all the words of encouragement. You fill up my tank with constant positivity even when I am empty, and everything looks grim. All the countless long drives and conversations we had throughout the years, thank you for listening to my thoughts, whether it be 2:30 in the afternoon or way late into the night, sometimes in different time zones, when everyone should be asleep. Thank you all for your continual support and care from the beginning to the very end of my undergraduate career.

Table of Contents

List of Figures

List of Tables

1 Introduction

Traditional Reinforcement Learning (RL) methods require an Artificial Intelligent (AI) agent to perform an action given the current state and learn either the risk/reward (values/features), a strategy of actions (policy) to perform, or both such that the agent maximizes some reward. One such method is Deep Q-Networks (DQNs), a variant of Q-Learning, which approximates the value of each state and allowable action (state-action pair) using a Deep Neural Network (DNN) [1]. DQNs and other RL methods, however, are time consuming when training. They may not converge to a solution, especially if the agent exists in a large feature space environment containing a large set of actions with sparse extrinsic reward signals, common in card games, thus requiring research in existing or innovative techniques to potentially address these issues.

A technique that found success in convergence in other areas of Machine Learning is pretraining, specifically in Deep Learning. Deep Learning is useful in learning high-level features composed of multiple lower-level features which are not possible with a simple network. With pretraining, the network is initialized in a particular region of parameter space often unreachable through random initialization and has proven to have better results compared to traditional methods of training [2]. A recent success like the Atari 2600 games [3] capitalizes on this strategy by training a Deep Convolutional Neural Network (ConvNet) through supervised learning on human moves and further trained using RL, ultimately demonstrating significant improvements on training time and convergence.

The goal in this research is to explore the principles of pretraining to pre-initialize RL agents through Supervised Learning for the card game Gin Rummy, a high-dimensional environment containing sparse reward signals between each state [4] that does not converge by using only conventional RL techniques like DQNs alone. The approach is to pre-initialize a DNN through Supervised Learning using thousands of self-play examples generated from a baseline agent policy [5]. The Deep Neural Network will be designed and tuned to learn possible lower-level features of the environment. The final weights of the DNN will be used as the initial parameters for the bottom layers of the function approximator in the DQN and undergo additional training through Q-Learning. With the combination of the DNN as pre-initialization on the DQN, the trained agent will be evaluated against a random agent, the original baseline agent, and self-play to test the validity of this pretraining approach in Q-Learning convergence.

2 Literature Review

A branch of RL methods commonly used is Q-Learning, where an agent attempts to learn the reward based on any given state-action pair, otherwise known as the Q-function. The Q-function, or better known as the Bellman Equation, for an optimal policy is defined by the following:

$$Q^*(s,a) = \sum_{s' \in S} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a') \right]$$

where state $s \in S$, action $a \in A$, s' is the resulting state due to action a from state s, $R(s,a,s')$ is the reward function, $T(s,a,s')$ is the transition dynamics, and γ is the discount factor [1]. With the Q-function, the optimal policy π^* can be derived by greedily taking the maximum Q-value at each state: $\pi^*(s) = \arg\max_a Q^*(s,a)$. In Q-Learning, the Q-function is iteratively estimated by allowing the agent to randomly explore the environment. The iterative estimate is as follows:

$$Q_{t'}(s_t,a_t) \leftarrow Q_t(s_t,a_t) + \alpha_t \left[r_{t'} + \gamma \max_{a'} Q_t(s_{t'},a') - Q_t(s_t,a_t) \right]$$

where state s_t, a_t, r_t are the state, action, and reward at time t and α_t is smoothing. Iteratively estimating the Q-function becomes an issue when the agent exists in a large feature space environment such as the Gin Rummy. Convergence may be plausible however will take an extremely long time as a value is required for each unique state-action pair [1]. DQNs address this issue by replacing the Q-function with a function approximator, otherwise known as a Neural Network. The replacement of the Q-function with a NN relieves the agent from exploring the entire environment and utilizes the generalizability a NN provides from the relaxed restriction. An issue, however, are that card games such as Gin Rummy have large environments and the application of DQNs does not guarantee convergence to an optimal policy.

This issue is addressed through the technique of pretraining, which is commonly used to accelerate training Deep Learning networks. As mentioned, DQNs are comprised of a NN as a function approximator which is used to approximate the Q-values of each state-action pair. The NN can be compared to that of a Deep Neural Network, both of which often are very difficult to train. According to Erhan et al., the difficulty in training DNNs is due to the large parameter space that the architectures reside in [2]. Through random initialization alone, a DNN is subject to converging to a local minimum, often of which is undesired. These local minima, described as *basins of attraction*, "trap" the networks and are very difficult for the network to "escape" [2]. With pretraining, the network can be initialized in a more favourable part of the parameter space,

allowing for an easier time for the network to find a better optimum [2]. Although the paper approaches pretraining through an unsupervised approach, the same principles is still applicable. With the application of pretraining to pre-initialize the function approximator, it should help the DQN find a more optimal policy compared to pure randomized initialization.

The paper by Cruz et al. parallels with the current research topic [3]. In their paper, the goal was to demonstrate the effects of supervised pretraining in speeding up Deep RL without the use of human expert data [3]. Using six Atari 2600 games as the environments and a Deep ConvNet as the pretrained network, Cruz et al. were successful in leveraging non-expert human data in speeding up five cases, using the DQN and Asynchronous Advantage Action-Critic (A3C) Deep RL algorithms [3]. They identified two aspects that need to be addressed in order to speed up Deep RL: feature learning, and policy learning. In DQNs, and Q-Learning, both optimal values and policy are learned in each iteration as the function approximator is updated. Cruz et al. addresses the first issue through pretraining a supervised network to learn features and allowed the DQN to focus on optimal policy search, which is also the primary focus of this research. An issue that was encountered in other studies in this topic was highly imbalanced classes due to action sparsity. Although there were no issues in Cruz et al.'s classifier in learning the important features, biases towards the imbalanced class may affect the agent policy [3]. The strategy implored by Cruz et al. will be explored in this research, however an emphasis will be made to verify that features are not subjected to biases toward majority classes.

Last, another technique that is used to improve training on sparse reward signal environments like Gin Rummy is Auxiliary Tasks [6]. By definition, a sparse reward signal is a series of extrinsic rewards obtained by the agent that are often non-positive [6]. Non-positive rewards make it difficult for an agent to learn how to interact with the environment, ultimately causing the agent to fail to learn. Auxiliary Tasks address this issue of reward sparsity by introducing a pseudo-reward function during the training process that help the agent select and learn aspects of the environment (not possible solely with extrinsic reward). Jaderberg et al. has shown to have speedup learning using this technique in the Atari games and Labyrinth [7]. The Auxiliary Task technique simultaneously updates the RL agent as well as the pseudo-reward function during training. From Jaderberg et al., the Auxiliary Task method required them to train a separate policy to maximize activation of each unit of the hidden layer [7]. This concept differs from the proposed pre-initializing method as the primary focus is learning features of the environment.

3 Gin Rummy

Gin Rummy is a two-player card game that uses a standard deck of 52 cards. A game starts with each player receiving 10 cards, a *discard* pile with 1 card faced up on top and the remaining deck faced down, both in the middle. The goal for both players is to create *melds* while minimizing the *deadwood* in their hand. A *meld* can be one of the following: 1) a run – three or more cards of the same suit in a row or 2) a set – three or more cards of the same rank. *Deadwood* is defined as the sum of the rank of all *non-melded* cards, where **J, Q,** and **K** have a rank of 10 and **A** has a rank of 1. Through various actions available in the game, players take turn replacing cards in their hand, either drawing a new card from the deck (*draw*) or picking up the top card from the *discard* pile (*pickup*). If a player has a hand with *deadwood* less than or equal to 10, the player may *knock*, ending the round for both players. Both players reveal their hands and calculate the difference in *deadwood*. The player with lower *deadwood* is awarded the difference as points. If the *knocking* player had a hand with all *melded* cards (zero *deadwood*), they are awarded an additional 25 points, otherwise known as going *gin*. Lastly, if the *knocking* player had a higher *deadwood*, the *knocking* player *undercuts***,** and an additional 25 points are awarded to the opposing player. The first player to 100 points wins the game.

3.1 Game Representation

The state representation (5 feature planes of size 52 cards each) and action space (110 available actions) defined by RLCard [4] was used to represent the Gin Rummy environments for this thesis. The following is a subset of the chosen states, actions, and state-action pairs used to define the current player state and allowable actions within these states:

Table 1: State and Action Definitions

State/Action [s or a]	**Description**
Before Pickup, Before Discard (**bpbd**) [s]	The state a player is in prior to deciding on whether to *pickup* the discarded card or *draw* a new card from the deck
After Pickup, Before Discard (**apbd)** [s]	The state a player is in after performing the *draw* or *pickup* action
After Pickup, After Discard (**apad***) [s]	The state a player is in after discarding a card from their hand
Draw [a]	Draw top card of the deck [Action ID: 2]
Pickup [a]	Pick up top card of the discard pile [Action ID: 3]
Gin [a]	Remove a card from hand and end round [Action ID: 5]
Discard [a]	Remove a card from hand [Action ID: 6-57]
Knock [a]	Remove a card from hand and end round [Action ID: 58-109]

*Note: **apad** is the state after an action is performed and is used to develop an intuition of lower-level features of the Gin Rummy.

Table 2: State-Action Pair Definition

State	**Available Action Group [Action ID]**	
bpbd	**pickup [2,3]**:	either *draw* **[2]** new card from deck or *pickup* **[3]** top card from discarded pile
apbd	**discard [6-57]**: **knock [58-109]**:	remove one of the available (11) cards in player's hand remove one of the available (11) cards in player's hand and end round
apad	**knock_bin**:	binary outcome of the current state to either knock or not knock based on remaining (10) cards in player hand
All States, All Actions (all)	A combination of all available states (**bpbd** and **apbd**) with their respective action groups (**pickup, gin, discard,** and **knock**)	

*Note: **bold** indicates states & action group, *italicized* indicates individual actions

4 Experimental Design

The approach to test the effects of Pre-Initialization on DQNs and subsequently Q-Learning can be broken down into individual environments, with the following steps:

1. Train a model using a baseline agent in a *Supervised Learning Environment*,
2. Initialize the Neural Network portion of the DQN using the weights from the trained network in the previous Supervised Learning step and train the DQN agent in a *Reinforcement Learning (Pre-Initialized DQN) Environment*,
3. Evaluate the performance of the Pre-Initialized DQN agent against the baseline agent, self-play, and random agent in a *Testing Environment.*

The next chapters will go in depth regarding the design and discuss the results and significance of each step/environment wherever applicable.

5 Supervised Learning Environment

This environment encapsulates the entire Gin Rummy gameplay. The primary goal of this environment is to train a functional model with parameters that can be further used to pre-initialize the DQN in the subsequent step. This is achieved by attempting to predict the actions performed by the available baseline agent within this environment, SimpleGinRummyPlayer (SGRAgent). The policy of this baseline agent is as follows:

1. Pick up card from discard pile if the card can be melded with cards in the current hand, otherwise draw a new card from the deck
2. Discard the highest ranked card in the hand that is not apart of any melds
3. Knock if deadwood is less than or equal to 10 (or Gin if deadwood equals 0)

5.1 Data Generation

Data is generated by modifying a two player Gin Rummy tournament provided by [8] to capture the state-action pairs from Table 2. The inputs are the states, encoded as a 260 (52 cards by 5 planes) long 1-D feature vector, and the outputs are the actions, encoded as a 110 long 1-D vector, as represented by [4]. Batches of games are denoted as *s_{x}k.npy* and *a_{x}k.npy*, where {x} represents the number of thousands of games, for states and actions, respectively. For each state-action pair, 2K, 6K, and 8K games were generated through self-play for training, validation, and testing purposes. An additional 32K and 40K batches of games for the **apbd-knock** state-action pair were generated to supplement training due to the unbalanced nature of the dataset previous generated.

5.2 Model Generation

Multiple models were trained and tuned based on the available state-action group pairs and used as baseline parameters in the reinforcement learning environment depending on their performances within the current environment. Each model was tasked in predicting the specified action of the provided action group it was trained on. Some lower-level features like melds were examined to gain better insight in some of these models. See Appendix A for more details in regard to the training results of each model. Additionally, lower-level feature insights for some models were investigated in Appendix A5.

5.3 Model Results

The best performing model, **all_2HL_80K**, was trained using 8,000 games of self-play and a total of 80,000 of pure **knock** actions data. Full training results can be found in Appendix A4.3. The data distribution of the entire dataset can be seen in the figure below. Note that there are approximately 1,000,000 state-action samples with the following distribution:

- 250,000 **pickup** actions (blue)
- 1,000 **gin** actions (black)
- 200,000 **discard** actions (red)
- 50,000 + 450,000 **knock** actions (green)

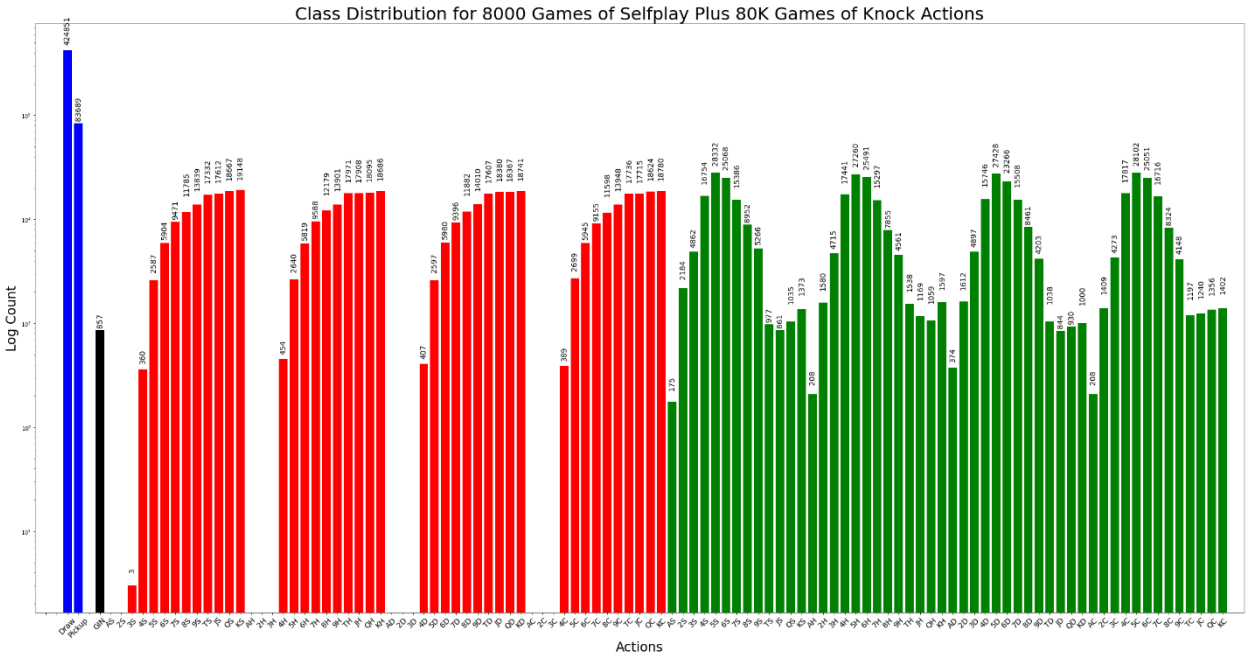

*Figure 5.1: Data Distribution for **all_2HL_80K***
***pickup** (blue), **gin** (black), **discard** (red), **knock** (green). Count (y-axis, log-scaled), Actions (x-axis).*

A typical round of Gin Rummy consists of approximately 10 actions:

- 50% **pickup** action group with a 4:1 *draw* to *pickup* action,
- 40% **discard** action group,
- 10% **knock** action group with a 4:1 *discard* to *knock*

The model was designed to be predict the actions of SGRAgent based on the provided state-action pairs. The 80,000 games of **knock** data was chosen to make the **knock-centric**, ensuring a larger distribution towards the unbalanced **knock** class. This is the preliminary model that will serve as the Pre-Initialization weights for the DQN agent.

6 Reinforcement Learning Environment

This section uses the PyTorch double DQN algorithm implementation from [4] to train DQN agents. Instead of an entire Gin Rummy gameplay like the *Supervised Learning Environment*, this environment is a based on an agent reaching the end of a round. The following are ways for the agent to receive a reward:

1) The agent performs a **gin** action (reward of 1)
2) The agent **knocks** (reward of 0.2 for knock)
3) The agent loses the round (reward of -0.01 for each deadwood differential)

This environment is a simplification of a full gameplay of Gin Rummy as the goal is to allow the agent to focus learning **gin** and **knock** actions within a round instead of learning the entire game. Within this environment, there are three agents that available to train and evaluate the agent performance: random agent, self-play, and SGRAgent. Additionally, each agent will be evaluated based on:

i. Average reward across 100 rounds,
ii. Average number of *turns* across 100 rounds,
   - where 1 *turn* is equivalent to 1 action in the *Supervised Learning Environment*
iii. Conditional Knock Probability across 100 rounds,
   - where $\text{Conditional Knock Probability} = \frac{\sum Agent\ \boldsymbol{Knocks}}{\sum \boldsymbol{Knock}\ action\ available}$
iv. Count of **gin** and **knock** actions across 100 rounds.

Finally, the following experiments were performed to test the agent's ability to learn within this environment.

i. With vs. Without Pre-Initialization (**baseline** agents built on Appendix A4)
ii. Frozen vs. Unfrozen Layers
iii. Various Reward Structures
iv. Additional Top Layer
   a. Randomized Top Layer Weights
   b. Copied Top Layer Weights using an Identity Matrix
   c. Pseudo-Identity Top Layer Weights
v. Knock Layer Biases
vi. Combination of the above

6.1 Deep Q-Network Algorithm

Below is the double DQN Algorithm used to train the agents with the following parameters, similar to the algorithm from [9]:

$D = Replay\ Memory$
$N = Replay\ Memory\ Size$
$N_i = Initial\ Replay\ Memory\ Size$
$\epsilon = Probability\ to\ choose\ random\ action, a_t$
$\epsilon_t = Epsilon\ decay\ factor\ after\ t\ steps$
$\gamma = Discount\ Reward\ Factor$
$B = Batch\ Size$
$T = Train\ every\ T\ steps$
$U = Update\ target\ network\ every\ U\ steps$

Algorithm 1 Double Deep Q-Network

Initialize replay memory $D\ with$ size N
Initialize estimator network $Q_{estimate}$ and target network Q_{target}
Initialize time step $t = 0$
for $episode = 1, \dots M$ **do**
  Initialize the environment with a random initial state s_0
  while $s_t\ is\ not\ terminal\ state$ **do**
    $t = t + 1$
    Generate trajectory $\phi_t(s_t, a_t, r_t, s_{t'})$ using $Q_{estimate}(s_t)$
    Store $\phi_t(s_t, a_t, r_t, s_{t'})$ into $D\ via\ FIFO$
    if $t > N_i\ and\ t\ mod\ T = 0$ **do**
      for $b = 1, \dots B$ **do**
        Randomly sample $\phi_b(s_b, a_b, r_b, s_{b'})$ from D
        Predict next action $a^e_{b'} = \begin{cases} \underset{a}{\text{argmax}}\ Q_{estimate}(s_{b'}), & \Pr(1-(\epsilon-\epsilon_t)) \\ \text{random a}, & \text{otherwise} \end{cases}$
        Store $\mathbf{a}^{\boldsymbol{e}}_{\boldsymbol{B}}[b] \leftarrow a^e_{b'}$
        Calculate $q - value\ q^t_b = \begin{cases} r_b, & s_{b'}\ \text{is terminal state} \\ r_b + \gamma \underset{a^e_{b'}}{\max} Q_{target}(s_{b'}), & \text{otherwise} \end{cases}$
        Store $\mathbf{q}^{\boldsymbol{t}}_{\boldsymbol{B}}[b] \leftarrow q^t_b$
      end for
      Perform gradient descent on $Q_{estimate}$ w. r. t. MSE Loss: $(\mathbf{q}^{\boldsymbol{t}}_{\boldsymbol{B}} - Q_{estimate}(\boldsymbol{s_B}))^2$
      if $t\ mod\ U = 0$ **do**
        $Q_{target} \leftarrow Q_{estimate}$
      end if
    end if
  end while
end for

6.2 Experimental Results

Multiple experiments were performed including but not limited to hyperparameter tuning, single and multi-layer freezing, knock layer bias, varying reward structures, additional top layers, and a combination of the listed. These results can be found in Appendix B. The following are the agents that will be discussed in this section:

- **dqn_baseline:** a baseline agent without Pre-Initialization (see Appendix B1.1 for results)
- **dqn_frozen:** a Pre-Initialized agent using the **all_2HL_80K** as the pre-initialization weights with frozen weights (see Appendix B2 for results)
- **dqn_knock_1:** a Pre-Initialized agent based on **dqn_frozen** using the Knock Layer Bias (see Appendix B5 for additional information about Knock Layer Bias)

After the results of Experiments i. through vi., a final agent, **dqn_knock_1**, was formulated. **dqn_knock_1** incentivizes the agent to perform **knock** actions through the addition of the Knock Layer Bias, a bias that is added to each *knock* action. The Knock Layer Bias is defined by the following equation:

$$y = softmax(\boldsymbol{W}x + \boldsymbol{b})$$

$$where\ y\ = Agent\ Output\ Layer, \qquad x\ =\ Pre-Initialized\ Output\ Layer$$

$$\boldsymbol{W} = \text{Identity Matrix}, \qquad b_i = \begin{cases} b\ +\ b_{knock}, & if\ action\ =\ knock \\ b, & otherwise \end{cases}, \qquad i \in [1{,}110]$$

This bias was required as all agents tend to 'forget' the **knock** action as seen through the negative trend of Conditional Knock Probabilities.

The following are non-default hyperparameters used to train **dqn_ knock_1**:

- Learning Rate = 1e-5 (default = 5e-5)
- MLP = [260, **520, 520**, 110**, 110**] = [*input*, **HL1, HL2,** *output*, **TL**] (*italicized* = default)
- **TL =** Knock Layer Bias $b_{knock} = 1$, followed by Softmax activation
- Sigmoid Activation between layers, Softmax after final layer
- Train Every 10 steps (default = 1)
- Batch Size = **64** (default = 32)
- Initial Memory Size = **1000** (default = 100)
- Number of Episodes = **40000** (default = 20000)

The macro average rewards against the random agent, self-play, and SGRAgent for **dqn_knock_1** are approximately 0.21, -0.02 and -0.04 respectively (Fig. 6.1).

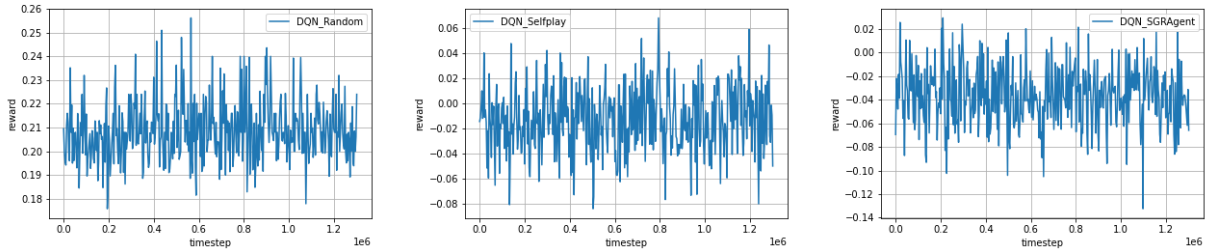

*Figure 6.1: Micro Average Reward plots for **dqn_knock_1** against Random (Left), Self-play (Middle), and SGRAgent (Right)*

Figure 6.2 displays the evaluation results against all three agents.

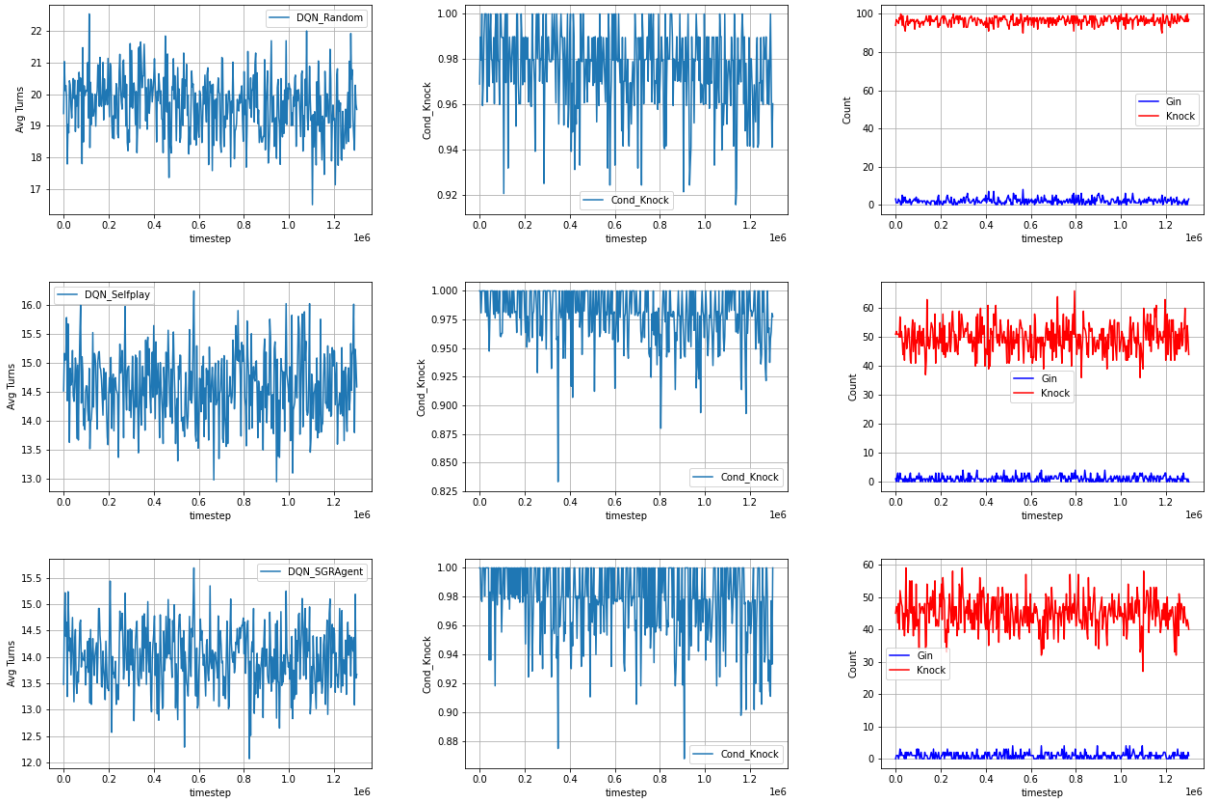

*Figure 6.2: **dqn_knock_1** plots Average Turns **(**Left), Conditional Knock Probability (Middle), and Action Count (Right) against Random Agent (Top), Self-Play (Middle), SGRAgent (Bottom)*

Comparing these results to **dqn_baseline** and **dqn_frozen**, some major takeaways can be drawn:

1. **dqn_knock_1** exhibits stable rewards against SGRAgent (Fig. 6.1, right) that is not observed in **dqn_frozen** (Fig. B9, right)
2. **dqn_knock_1** does not exhibit the 'forget' **knock** action that is reflected in the Conditional Knock Probabilities being above 0.85 (Fig. 6.2, middle)
3. The average number of turns for **dqn_knock_1** does not have a positive trend (Fig. 6.2, left) often observed in other agents
4. The average count of **knock** actions against SGRAgent does not exhibit a non-positive trend with an average count of 45 actions (Fig. 6.2, bottom right), higher than **dqn_frozen** of approximately 35-40 initially (Fig. B10, bottom right)

7 Testing Environment

This environment is the same as the *Supervised Learning* Environment except it is used for evaluation purposes only. Each agent trained in the *Reinforcement Learning Environment* was evaluated for 1000 games against Random Agent, SGRAgent, Baseline Agents, and Pre DQN-Training. See Appendix C for the full evaluation results. The following table will highlight a summary of relevant and the best performing agent, **dqn_knock_1**.

Table 3: Win rate Summary of DQN Agents

Opponent vs. Model			**dqn_baseline**	**dqn_frozen**	**dqn_knock_1**
Random vs. Post DQN-Training			-	100%	100%
SGRAgent	Pre DQN-Training		0%	30.4%[2]	33.5%[2]
	Maximum Reward	Random	0%	23.8%	34.3%[2]
		Self-Play	0%	26.1%	35.8%[2]
		SGRAgent	0%	24.2%	34.3%[2]
	Maximum Knock Probability	Random	0%	31.2%	33.7%[1]
		Self-Play	0%	29.0%	33.5%[1]
		SGRAgent	0%	30.4%	33.5%[1]
	Post DQN-Training		0%	23.1%	37.1%[2]
Baseline vs. Pre DQN-Training			-	49.2%	52.9%[3]
Baseline vs. Post DQN-Training			-	40.8%	55.1%[3]
Pre vs. Post DQN-Training			-	40.8%	52.1%[3] || 54.0%[3][4]

[1] Win rates have no meaning as agent has near 100% Conditional Knock Probability
[2] DQN-Training win rates larger than Pre DQN-Training
[3] Post DQN-Training outperformed Pre DQN-Training/**baseline** agent
[4] 5000 games

To summarize the results of the *Testing Environment*, majority of agents were able to win 100% of the time against the Random Agent with an exception to some agents (see Table 11 & Table 12 of Appendix C). However, as seen during the evaluation in the *Reinforcement Learning Environment* (see Appendix B1.1), all agents tended to perform worse as they went through training. This is evident for **dqn_frozen** against the **SGRAgent**, where the win rate Pre DQN-Training was 30.4% however dropped to 23.1% after training. **dqn_knock_1** exceeded expectations as win rates against **SGRAgent** during and post DQN- Training were larger than pre DQN-Training. Additionally, the post DQN-Training **dqn_knock_1** agent beat the pre DQN-Training **dqn_knock_1** agent 52.1% and 54.0% over 1000 and 5000 games, respectively.

8 Conclusion

Due to the nature of card games with large feature spaces, Q-Learning does not converge through random initialization (see Appendix B1.1). The goal of this thesis was to Pre-Initialize a Deep Q-Network agent with weights trained in a Supervised Learning Environment to see if Pre-Initialization will allow the agent to converge to a more optimal policy through Q-Learning. In the *Supervised Learning Environment*, a two Hidden Layer model, **all_2HL_80K**, was trained to learn the state-action pairs of a baseline agent, SGRAgent, with a high validation accuracy of 98% (see Appendix A4.3). Although achieving a high validation accuracy in predicting the correct actions based on a given state, this model only achieves a win rate of 30.4% against the same baseline agent (see Table 7). Overall, the model learned a suboptimal policy from the baseline agent, however still serviceable. With this suboptimal policy as the Pre-Initialization for the DQN agent, majority of agents outperformed a random agent, something previously unobtainable through random initialization alone. This is undesirable as it does not conclusively suggest the benefits of Pre-Initialization in Q-Learning. Additionally, all agents continued to struggle against SGRAgent, with the best win rate being 37.1% (see Table 3, **dqn_knock_1**). However, **dqn_knock_1** had promising results as the post DQN-Training win rate of 37.1% was larger than pre DQN-Training win rate of 33.5% (see Table 3, **dqn_knock_1**). Additionally, this agent post DQN-Training beat the pre DQN-Training variant with win rates of 52.1% and 54.0% over 1000 and 5000 games, respectively. These results demonstrate that given the Pre-Initialization policy, the agent was able to learn better features through Q-Learning and thus showing moderate success.

8.1 Future Work

8.1.1 Multiple Environments

There are three environments that were used within this thesis: *Supervised Learning Environment*, *Reinforcement Learning Environment*, and *Testing Environment*. The major difference between the environments is the gameplay representation of Gin Rummy; entire game versus round based. This poses an issue of evaluation consistency for agents between these environments as an agent that performs well within the *Reinforcement Learning Environment* does not directly correlate to performing well in the *Testing Environment*. Two such instances would be an *undercut* action due to the difference in deadwood and **gin** actions due to an excessive number of turns taken. The former is not captured in the *Reinforcement Learning Environment* while the latter is a byproduct if this environment not directly penalizing the number of turns to reach an end state. To ensure more consistency, the *Reinforcement Learning Environment* should be changed to represent an entire game of Gin Rummy.

8.1.2 Card Game Exploration

The goal of the thesis was to explore Pre-Initializing Q-Learning for card games, which was accomplished for the card game Gin Rummy. The results show positive effects from pre-initializing an agent prior to Q-Learning, as seen in **dqn_knock_1**. With this moderate success, the concept of Pre-Initialization should be transferrable to other card games containing large feature spaces like this game. Future work should be done to other card games to test this hypothesis.

8.1.3 Supervised Learning Model Investigations

Finally, there currently is no insight behind the lower-level features that are learned by the larger models, as shown Appendix A5. Only the possibility of melds being learnt by the model was investigated with **apad_knock.** All models, especially **all_1HL, all_2HL_40K,** and **all_2HL_80K**, should be investigated to see if melds were one of the lower-level features learnt by the model. Additionally, other lower-level features should also be investigated to further understand and tune these models.

9 References

[1] M. Roderick, J. Macglashan, and S. Tellex, *Implementing the Deep Q-Network*. 2017. [Online] Available: https://arxiv.org/pdf/1711.07478.pdf

[2] D. Erhan, A. Courville, Y. Bengio, and P. Vincent, *Why Does Pre-training Help Deep Learning?* The Journal of Machine Learning Research, 2010. [Online]. Available: http://proceedings.mlr.press/v9/erhan10a/erhan10a.pdf

[3] G. Cruz Jr, Y. Du, and M. Taylor, *Pre-training Neural Networks with Human Demonstrations for Deep Reinforcement Learning*. 2nd ed, 2019. [Online]. Available: https://arxiv.org/pdf/1709.04083.pdf

[4] "Gin Rummy", *Games in RLCard*. DATA Lab at Texas A&M University, 2020. [Online]. Available: https://rlcard.org/games.html#gin-rummy

[5] T. Neller, *Gin Rummy EAAI Undergraduate Research Challenge*. Gettysburg College: Department of Computer Science, 2019. [Online]. Available: http://cs.gettysburg.edu/~tneller/games/ginrummy/eaai/

[6] J. Hare, *Dealing with Sparse Rewards in Reinforcement Learning*. 2nd ed, 2019. [Online]. Available: https://arxiv.org/pdf/1910.09281.pdf

[7] M. Jaderberg, V. Mnih, W.M. Czarnecki, et al, *Reinforcement Learning with Unsupervised Auxiliary Tasks*. 1st ed, 2016. [Online]. Available: https://arxiv.org/pdf/1611.05397.pdf

[8] Hien, *GinRummyEAAI Python*, Git code 2020. [Online]. Available: https://github.com/AnthonyHein/GinRummyEAAI-Python

[9] V. Mnih, K. Kavukcuoglu, D. Silver, et al, *Playing Atari with Deep Reinforcement Learning*, 2013. [Online]. Available: https://arxiv.org/pdf/1312.5602.pdf

10 Appendix

Appendix A: Supervised Learning Training Results

A1: State – Before Pickup, Before Discard (bpbd)

A1.1: Action Group – pickup

The goal of this model is to learn the pickup **policy** of the SGRAgent, which is to pick up the discarded card only if it can form a meld with the current cards in the agent's hand. Below is the data distribution of *draw* and *pickup* actions for 8K games of self-play, using an 80:20 train:validation split. Note that the counts (x-axis) are log-scaled. It can be observed that majority of the actions are *draw* (approximately 4 *draws*:1 *pickup*).

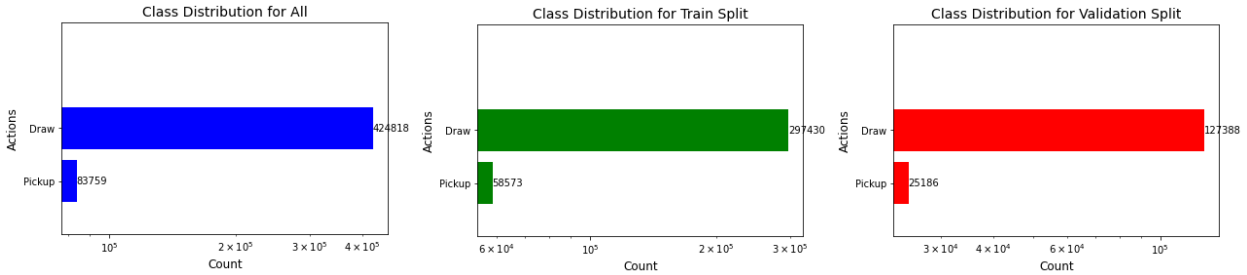

*Figure A1: Data Distribution for **bpbd_pickup***
all (left), train (middle), validation (right). Actions (y-axis), Count (x-axis, log-scaled).

The model, **bpbd_pickup**, was trained using the following hyperparameters and the following page are the results of the network:

- Batch Size = 1000
- Learning Rate = 0.001
- Epochs = 100
- MLP = [260, 520, 2] = [input, HL1, output]
- Sigmoid Activation between layers
- Softmax Outputs
- MSE Loss, Adam Optimizer (PyTorch default settings)

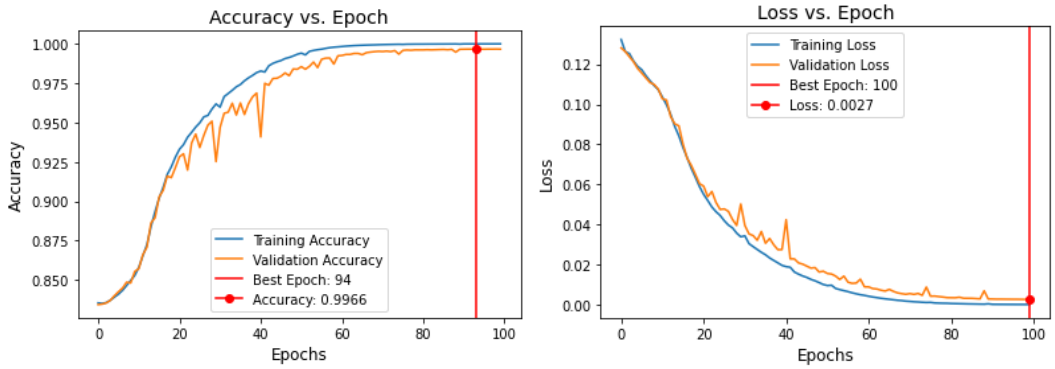

*Figure A2: Training plots for **bpbd_pickup***
Accuracy (left), Mean Loss (right)

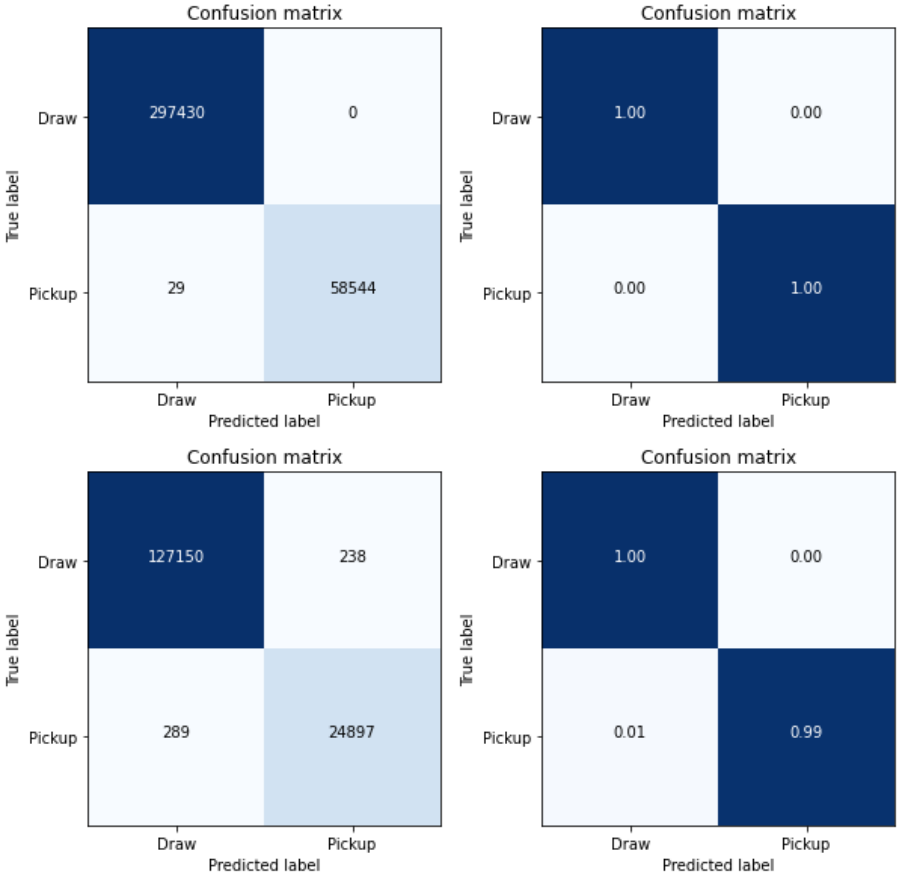

*Figure A3: Confusion Matrices for **bpbd_ pickup***
Train (top), Validation (bottom), Normalized Values (right).
True Label (y-axis), Predicted Label (x-axis). Darker tile = larger number.

From the results of the training, **bpbd_pickup** successfully learned to differentiate between the *draw* and *pickup* actions. In validation, majority of the samples were correctly classified, with minimal samples being misclassified.

A2: State – After Pickup, Before Discard (apbd)

A2.1: Action Group – discard

The goal of this model is to learn the **discard** policy of the SGRAgent, or which card to discard in the agent's hand. Below is the data distribution of *discard* actions for 8K games of self-play, using an 80:20 train:validation split. Note that the counts (x-axis) are log-scaled, and the cards (y-axis) are in descending rank order by the following suits: **S, H, D, C**. Additionally, it can be observed that the data contains no samples of **A, 2** ranks for all suits, very few samples of **3** ranks, and an increasing number of samples as the rank increases, as well as symmetry in distribution between suits, which is indicative of the agent that was used to generate the data.

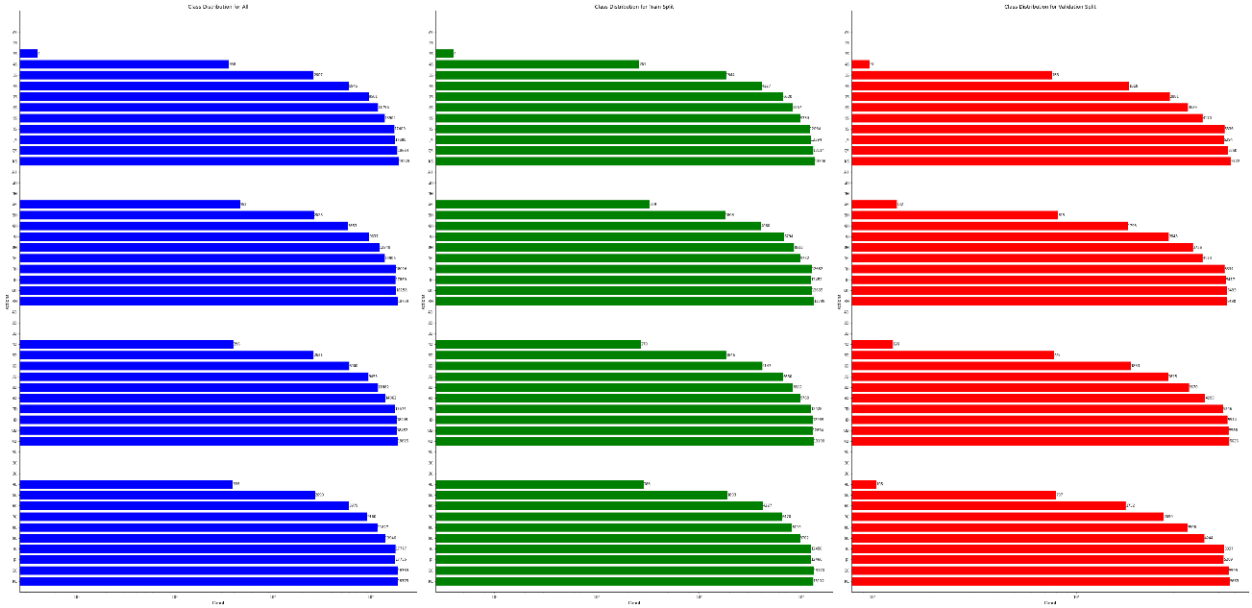

*Figure A4: Data Distribution for **apbd_discard***
all (left), train (middle), validation (right). Actions (y-axis), Count (x-axis, log-scaled).

The model, **apbd_discard**, was trained using the following hyperparameters and the following page are the results of the network:

- Batch Size = 1000
- Learning Rate = 0.001
- Epochs = 100
- MLP = [260, 520, 52] = [input, HL1, output]
- Sigmoid Activation between layers
- Softmax Outputs
- MSE Loss, Adam Optimizer (PyTorch default settings)

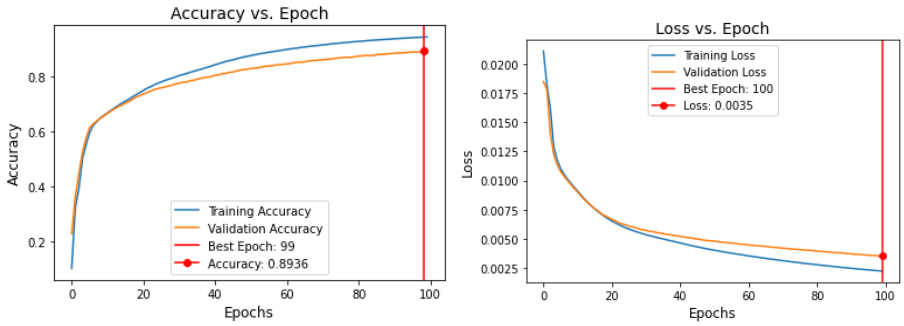

*Figure A5: Training plots for **apbd_discard***
Accuracy (left), Mean Loss (right)

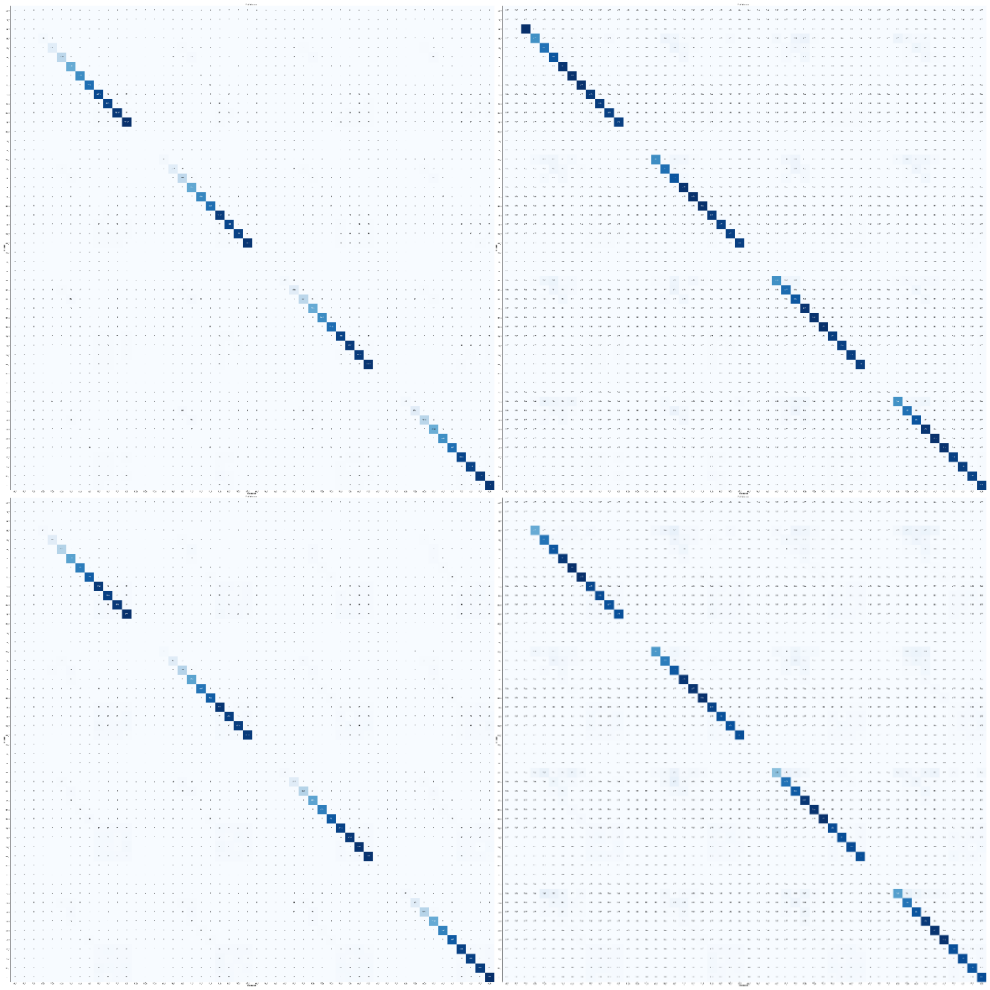

*Figure A6: Confusion Matrices for **apbd_discard***
Train (top), Validation (bottom), Normalized Values (right).
True Label (y-axis), Predicted Label (x-axis). Darker tile = larger number.

From the results of the training, four distinct sections can be observed in the confusion matrices, one for each suit. In majority of the samples, higher ranked cards were predicted more accurately (right). Lower ranked cards in validation had moderate prediction success compared to train. Overall, **apbd_discard** successfully learned to discard higher ranked cards to a high degree and moderate success with lower ranked cards.

A2.2: Action Group – knock

The goal of this model is to learn the **knock** policy of the SGRAgent, or which card to discard in the agent's hand prior to knocking. Below is the data distribution of *knock* actions for 8K games of self-play, using an 80:20 train:validation split. Note that the counts (x-axis) are log-scaled, and the cards (y-axis) are in descending rank order by the following suits: **S, H, D, C**. Additionally, it can be observed that the data contains symmetry in distribution between suits, with larger distribution of samples with cards of rank **4**, **5** (largest for each suit), and **6**.

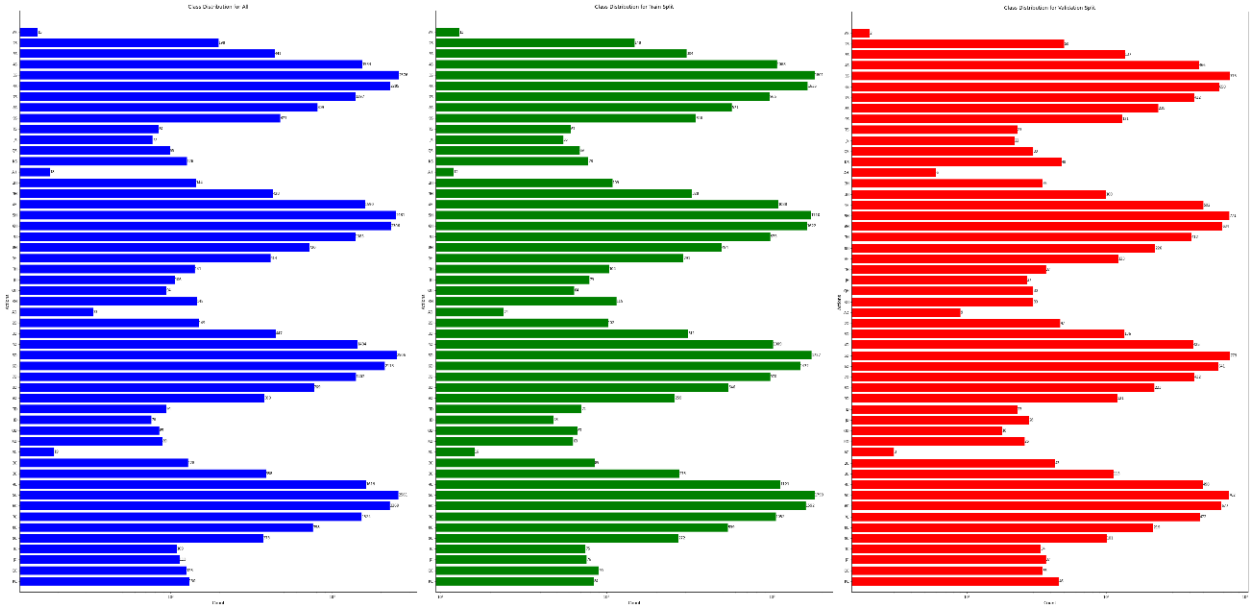

*Figure A7: Data Distribution for **apbd_knock***
all (left), train (middle), validation (right). Actions (y-axis), Count (x-axis, log-scaled).

The model, **apbd_knock**, was trained using the following hyperparameters and the following page are the results of the network:

- Batch Size = 1000
- Learning Rate = 0.001
- Epochs = 100
- MLP = [260, 520, 52] = [input, HL1, output]
- Sigmoid Activation between layers
- Softmax Outputs
- MSE Loss, Adam Optimizer (PyTorch default settings)

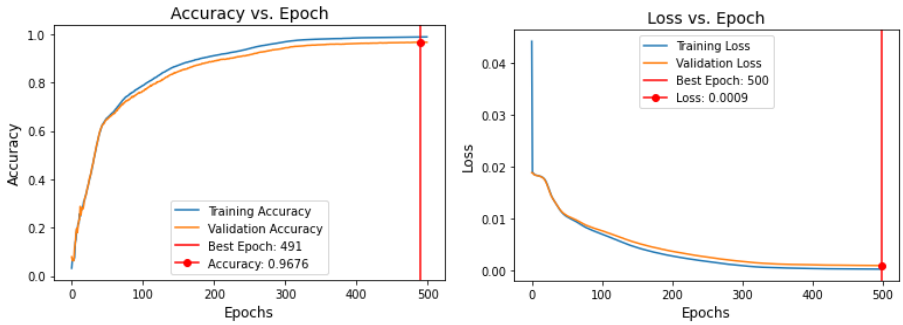

*Figure A8: Training plots for **apbd_knock***
Accuracy (left), Mean Loss (right)

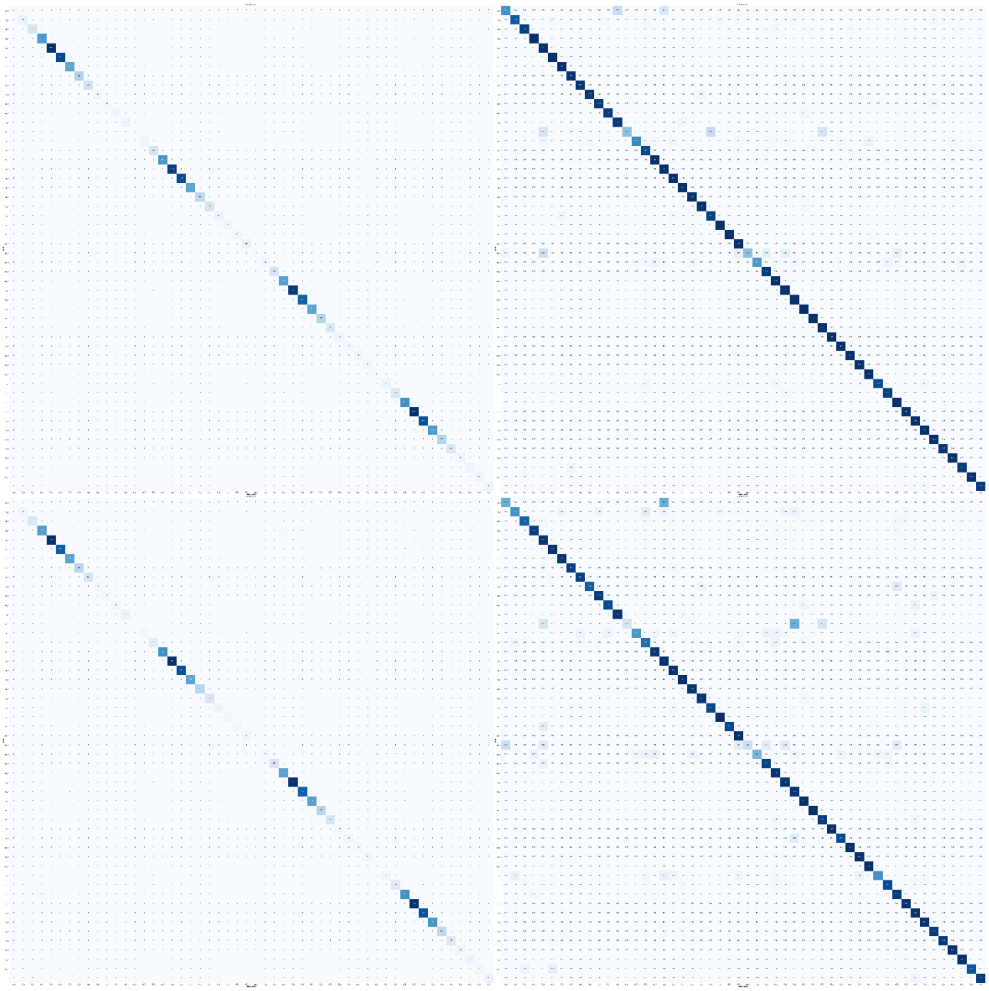

*Figure A9: Confusion Matrices for **apbd_knock***
Train (top), Validation (bottom), Normalized Values (right).
True Label (y-axis), Predicted Label (x-axis). Darker tile = larger number.

From the results of the training, four distinct sections can be observed in the confusion matrices, one for each suit (Fig. A9, left). In majority of the samples, higher ranked cards were predicted more accurately (Fig. A9, right). For lower ranked cards, specifically **A** and **2,** the model had difficultly in classifying them in both sets. Overall, **apbd_knock** successfully learned to knock higher ranked cards to a high degree and moderate to low success with low ranked cards.

A3: State – After Pickup, After Discard (apad)

A3.1: Action Group – knock_bin

The goal of this model is to learn the **knock** policy of the SGRAgent, or which is to whether the agent will knock given the state **after** the agent has discarded. Below is the data distribution of *knock* and *no-knock* actions for 8K games of self-play, using an 80:20 train:validation split. Note that the counts (x-axis) are log-scaled. It can be observed that majority of the actions are *no-knocks* (approximately 10 *no knocks*: 1 *knock*).

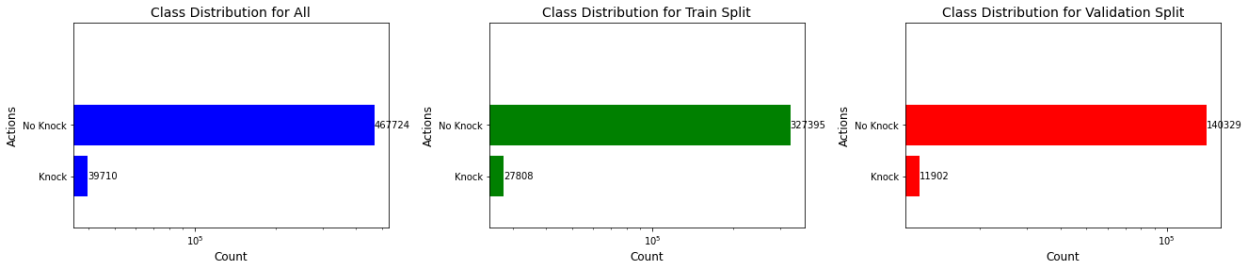

*Figure A10: Data Distribution for **apad_knock***
all (left), train (middle), validation (right). Actions (y-axis), Count (x-axis, log-scaled).

The model, **apad_knock**, was trained using the following hyperparameters and the following page are the results of the network:

- Batch Size = 1000
- Learning Rate = 0.001
- Epochs = 100
- MLP = [260, 520, 2] = [input, HL1, output]
- Sigmoid Activation between layers
- Softmax Outputs
- MSE Loss, Adam Optimizer (PyTorch default settings)

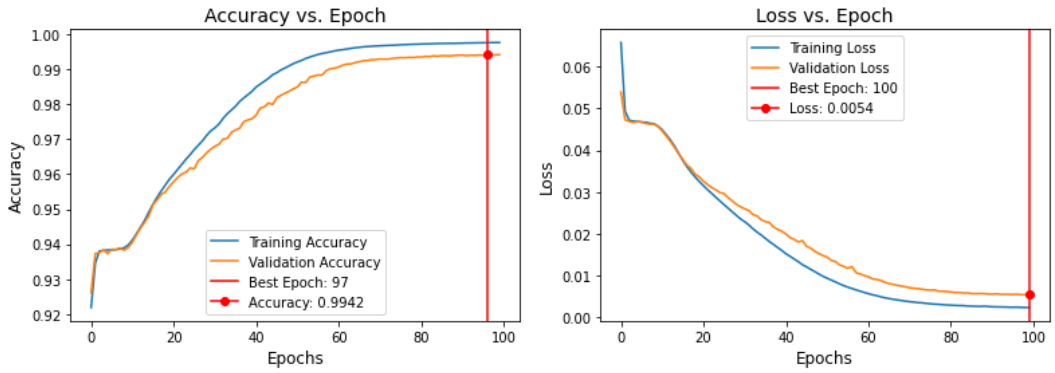

*Figure A11: Training plots for **apad_knock***
Accuracy (left), Mean Loss (right)

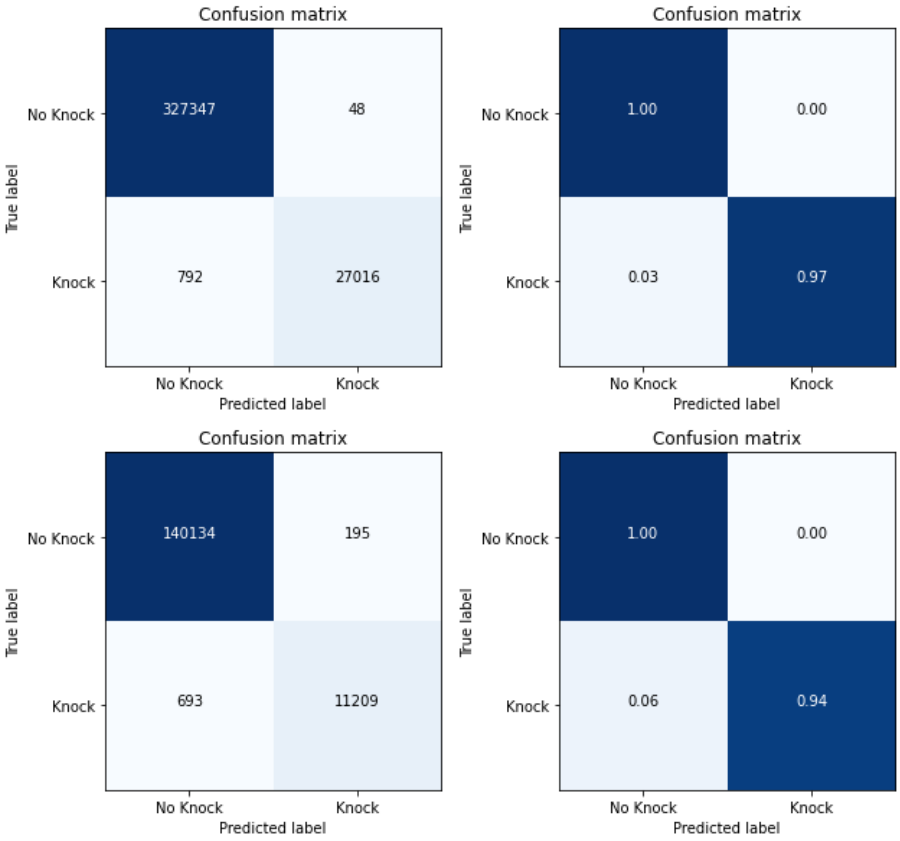

*Figure A12: Confusion Matrices for **apad_knock***
Train (top), Validation (bottom), Normalized Values (right).
True Label (y-axis), Predicted Label (x-axis). Darker tile = larger number.

From the results of the training, **apad_knock**, there are minor issues with the model learning to distinguish knocking from not knocking. The true prediction of knock on the train set was 97% accurate whereas the true prediction on the validation set was 94%, a large drop in knock prediction accuracy.

A4: State – All States, All Actions (all)

This section contains models trained with both **bpbd** and **apbd** states combined as inputs, alongside all possible action groups (**pickup, gin, discard, knock**) as outputs. These models serve as the culmination of previous models and their insights and will be used primarily in the pre-initialization of parameter weights in the DQN step. Three models will be highlighted:

all_1HL: baseline model using the same hyperparameters as previous models

all_2HL_40K: adapted version of **all_1HL**, with an extra hidden layer and more data

all_2HL_80K: adapted version of **all_1HL**, with an extra hidden layer and even more data

A4.1: all_1HL

The goal of this model is to learn a baseline model for all the state-action pairs generated from the SGRAgent. Below is the data distribution of these state-action pairs for 8K games of self-play, using an 80:20 train:validation split. Note that the counts (x-axis) are log-scaled. It can be observed that majority of the distribution (50%) of samples occurs in the top two bars, otherwise known as *draw* or *pickup* actions, which is expected as each turn requires one *draw* action from **bpbd** state and one action from **apbd** state, whether it be *discard* or *knock***.** Another observation, which is also present in the differences in data counts between **apbd_discard** and **apbd_knock** as well as **apad_knock,** is the ratio of **discard** samples compared to **knock**. There are approximately 5 *discards*:1 *knock*. Overall, there are approximately 10 actions/round.

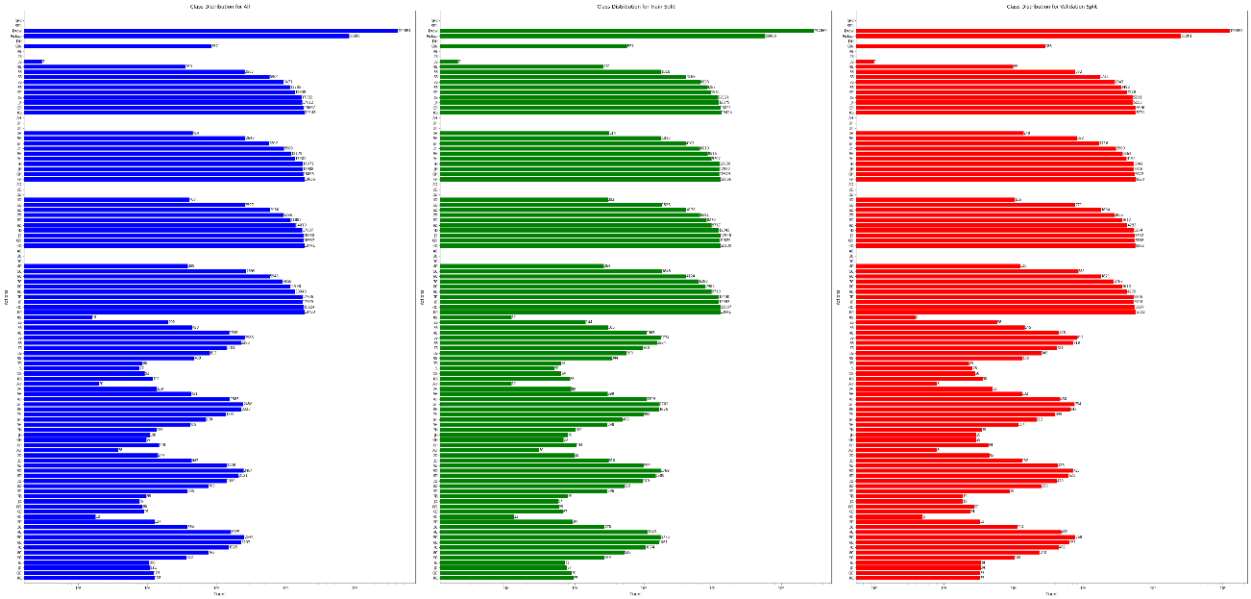

*Figure A13: Data Distribution for **all_1HL***
all (left), train (middle), validation (right). Actions (y-axis), Count (x-axis, log-scaled).

The model, **all_1HL**, was trained using the following hyperparameters and the following page are the results of the network:

- Batch Size = 1000
- Learning Rate = 0.001
- Epochs = 200*
- MLP = [260, 520, 110] = [input, HL1, output]
- Sigmoid Activation between layers
- Softmax Outputs
- MSE Loss, Adam Optimizer (PyTorch default settings)

***Note: Number of epochs were doubled as **all_1HL** had slower convergence.

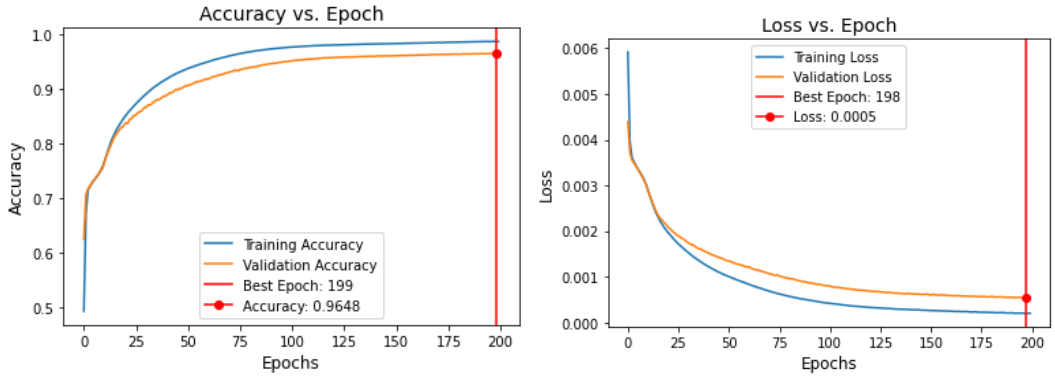

*Figure A14: Training plots for **all_1HL***
Accuracy (left), Mean Loss (right)

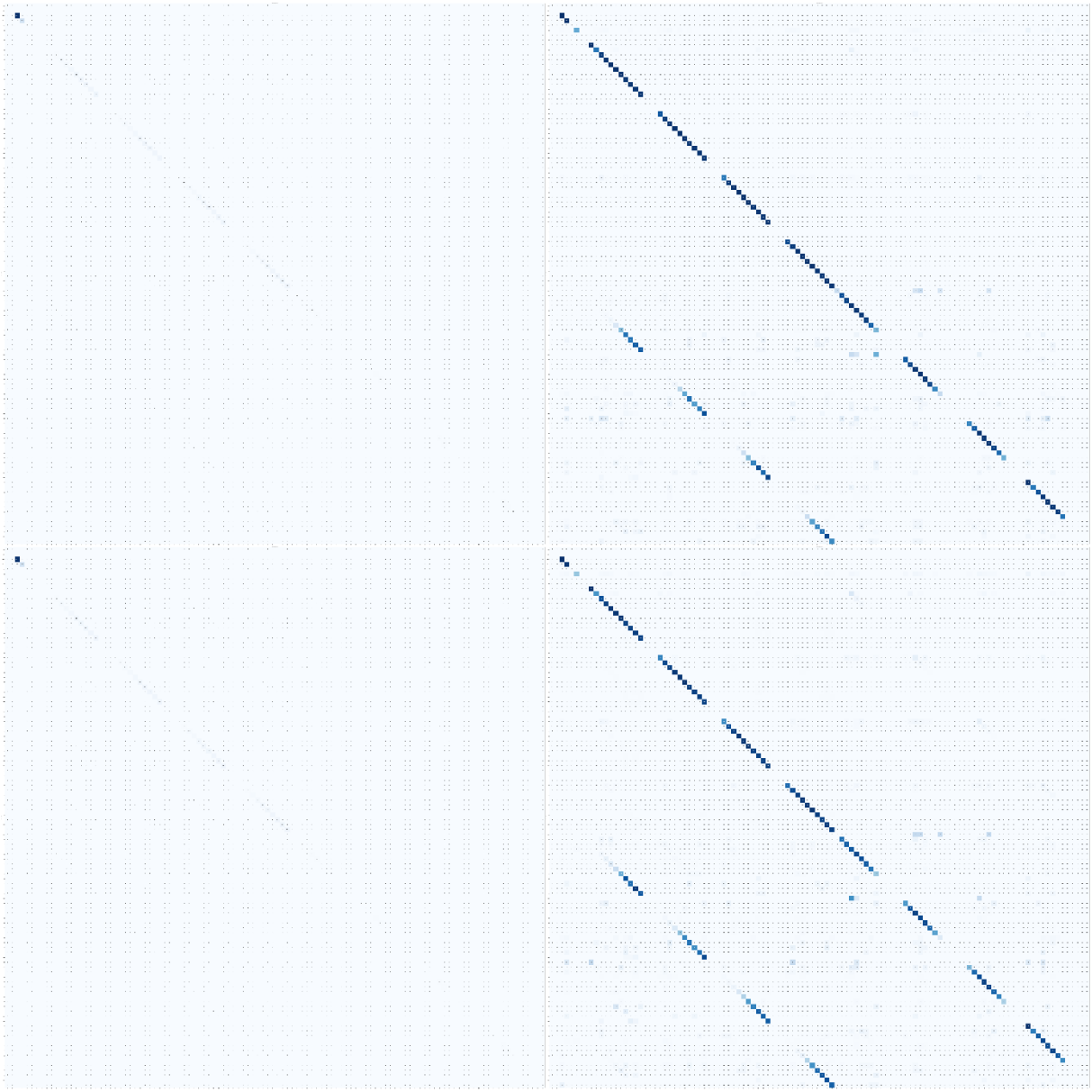

*Figure A15: Confusion Matrices for **all_1HL***
Train (top), Validation (bottom), Normalized Values (right).
True Label (y-axis), Predicted Label (x-axis). Darker tile = larger number.

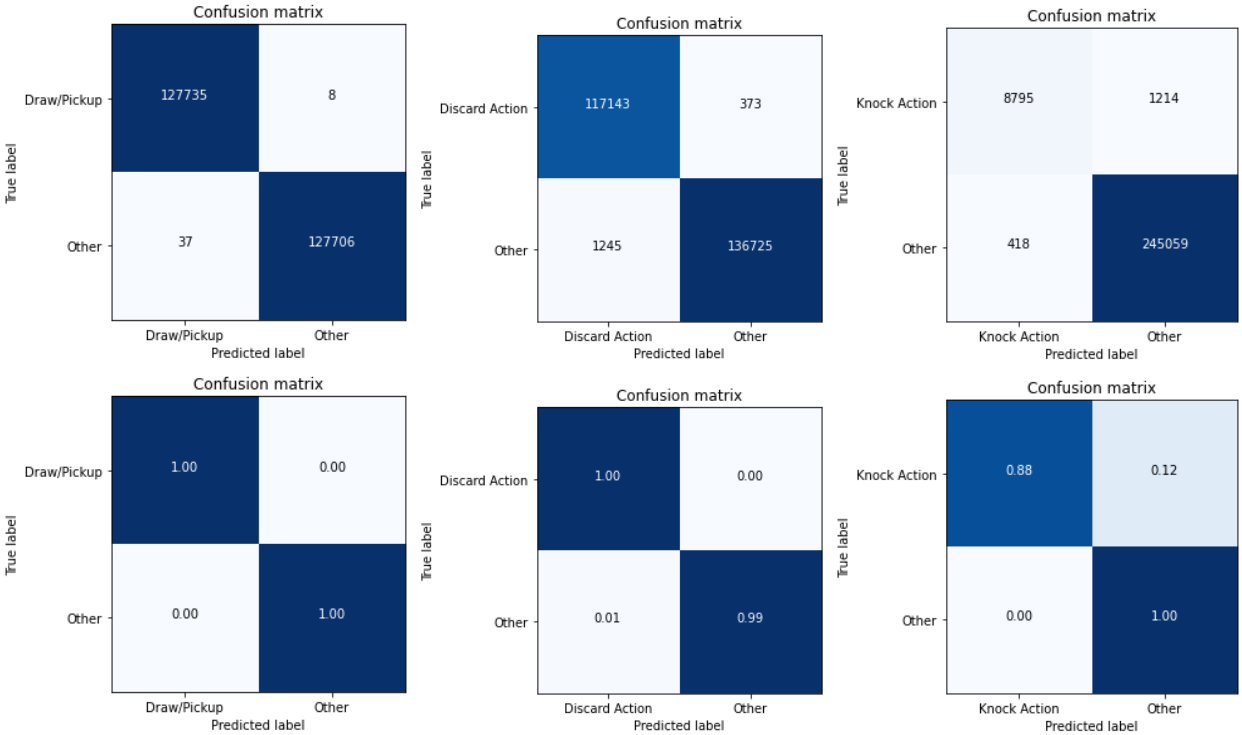

Figure A16: Confusion Matrices for **all_1HL** based on state-action groups*
***'draw'** (left), **'discard'** (middle), **'knock'** (right), Normalized Values (bottom).*
*True Label (y-axis), Predicted Label (x-axis). Darker tile = larger number. *2K Games Test Set*

From the results of the training, **all_1HL**, a final validation accuracy of 96.48% was achieved. The confusion matrices from Figure A15 illustrate the following:

1) Majority of the samples are within the **pickup** action group (Fig. A15, left),

2) All three state-action groups can be identified by 'quadrants' (Fig. A15, right)

   - the upper left section is **pickup**,
   - the middle section is **discard**,
   - the lower right section is **knock**

3) **knock** actions are being misclassified to discard action (Fig. A15, right)

   - More specifically, a card of certain suit and rank that was intended to be knocked is being predicted as discard

The confusion matrices from Figure A16 also illustrates these issues. When evaluated based on state-*action groups* pairs, both **pickup** and **discard** have a high degree of accuracy predicting the true labels however only an 88% accuracy for **knock** is achieved (Fig. A16, right).

A4.2: all_2HL_40K

Due to the poor performance of **knock** prediction in the **all_1HL** model, a total of 40K games of **knock** data was used to supplement training. Below is the data distribution of these state-action pairs for 8K games of self-play with the supplemented 40K **knock** data, using an 80:20 train:validation split. Note that the counts (x-axis) are log-scaled. With the supplemented data, there are approximately 1 *discard*:1 *knock*.

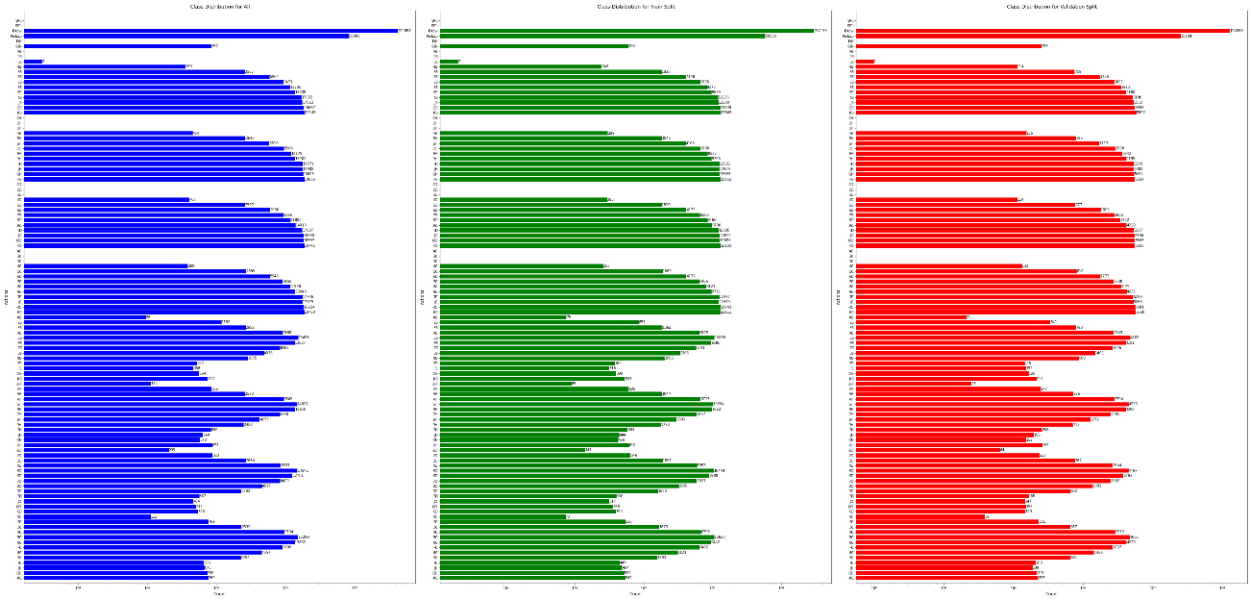

*Figure A17: Data Distribution for **all_2HL_40K***
all (left), train (middle), validation (right). Actions (y-axis), Count (x-axis, log-scaled).

The model, **all_2HL_40K**, was trained using the following hyperparameters and the following page are the results of the network:

- Batch Size = 1000
- Learning Rate = 0.001
- Epochs = 200
- MLP = [260, 520, **520**, 110] = [input, HL1, **HL2**, output]*
- Sigmoid Activation between layers
- Softmax Outputs
- MSE Loss, Adam Optimizer (PyTorch default settings)

***Note: An additional hidden layer was found to increase model performance on **knock** predictions

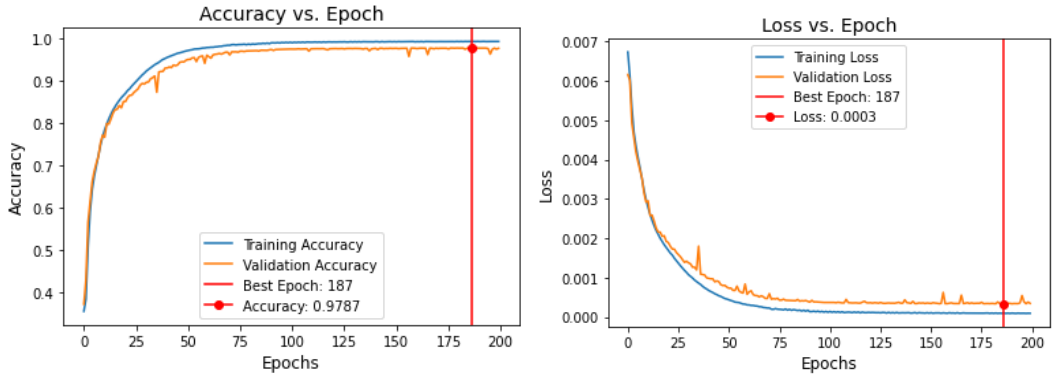

*Figure A18: Training plots for **all_2HL_40K***
Accuracy (left), Mean Loss (right)

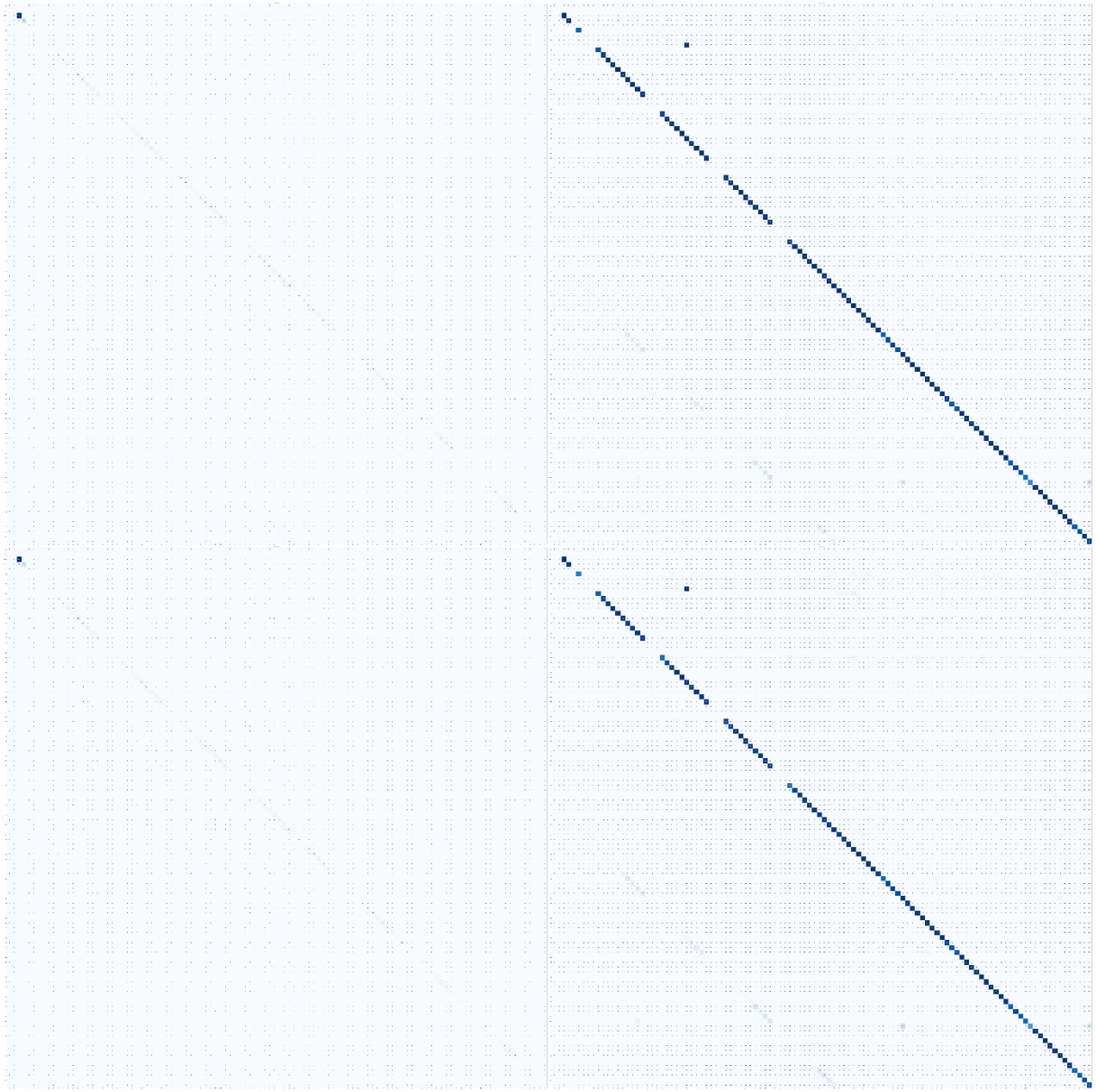

*Figure A19: Confusion Matrices for **all_2HL_40K***
Train (top), Validation (bottom), Normalized Values (right).
True Label (y-axis), Predicted Label (x-axis). Darker tile = larger number.

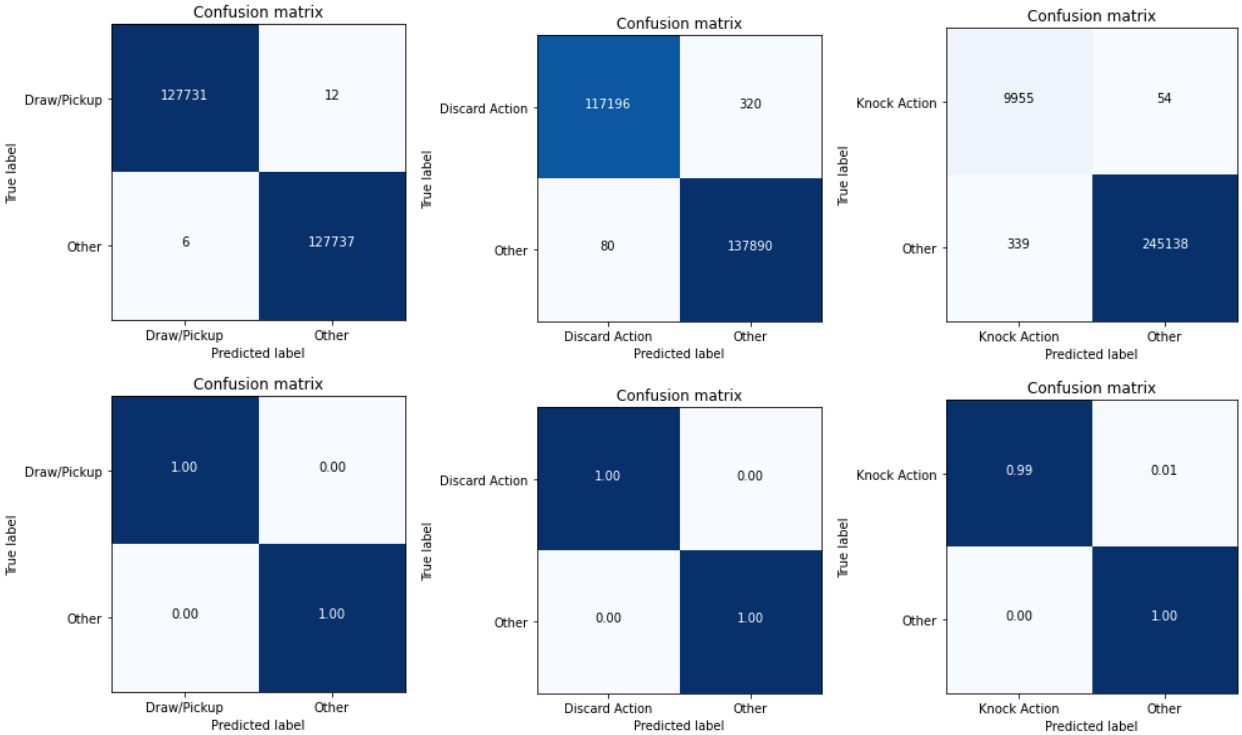

Figure A20: Confusion Matrices for **all_2HL_40K** based on state-action groups*
***'draw'** (left), **'discard'** (middle), **'knock'** (right), Normalized Values (bottom).*
*True Label (y-axis), Predicted Label (x-axis). Darker tile = larger number. *2K Games Test Set*

From the results of the training, **all_2HL_40K**, a final validation accuracy of 97.87% was achieved. The confusion matrices from Figure A19 illustrate the following:

1) **knock** actions are not as often being misclassified as **discard** action (Fig. A19, right) compared to prior (Fig. A15, right). It can be seen, however, that there still exists a faint diagonal within the True **knock**, Predicted **discard** section of the confusion matrix (Fig. A19, bottom right, bottom middle of the confusion matrix)

2) *discard 3S* is being misclassified as *discard 5H*

From the data distribution, it can be noted that there are only 3 samples of **3S** (1 sample in Validation Set), as the SimpleGinRummyPlayer agent policy is to avoid discarding low ranked cards

The confusion matrices from Figure A20 show drastic improvements compared to the **all_1HL** model, with all state-action groups pairs having a high degree of accuracy predicting the true labels (Fig. A20, bottom).

A4.3: all_2HL_80K

A total of 80K games of **knock** data was used to supplement training on top of the **all_1HL**. Below is the data distribution of these state-action pairs for 8K games of self-play with the supplemented 80K **knock** data, using an 80:20 train:validation split. Note that the counts (x-axis) are log-scaled. With the supplemented data, there are approximately 1 *discard*:2 *knock*. The surplus of knock data is desired as a knock-centric model would be desired in the DQN environment.

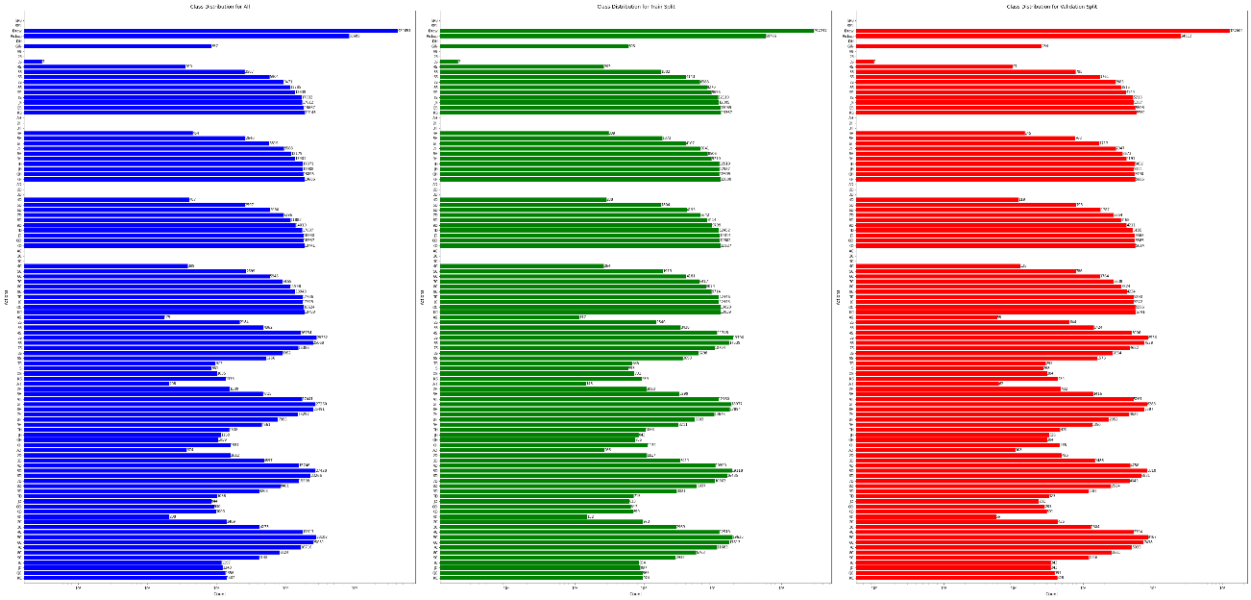

*Figure A21: Data Distribution for **all_2HL_80K***
all (left), train (middle), validation (right). Actions (y-axis), Count (x-axis, log-scaled).

The model, **all_2HL_80K**, was trained using the following hyperparameters and the following page are the results of the network:

- Batch Size = 1000
- Learning Rate = 0.001
- Epochs = 200
- MLP = [260, 520, **520**, 110] = [input, HL1, **HL2**, output]*
- Sigmoid Activation between layers
- Softmax Outputs
- MSE Loss, Adam Optimizer (PyTorch default settings)

*****Note: An additional hidden layer was found to increase model performance on **knock** predictions

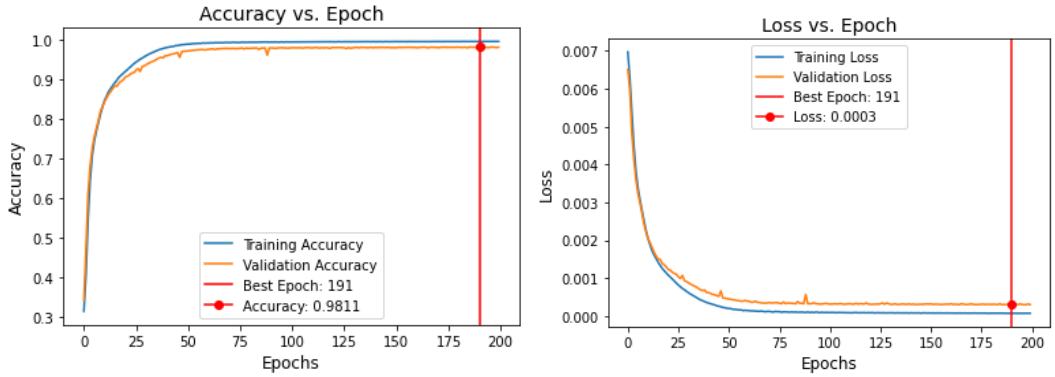

*Figure A22: Training plots for **all_2HL_80K***
Accuracy (left), Mean Loss (right)

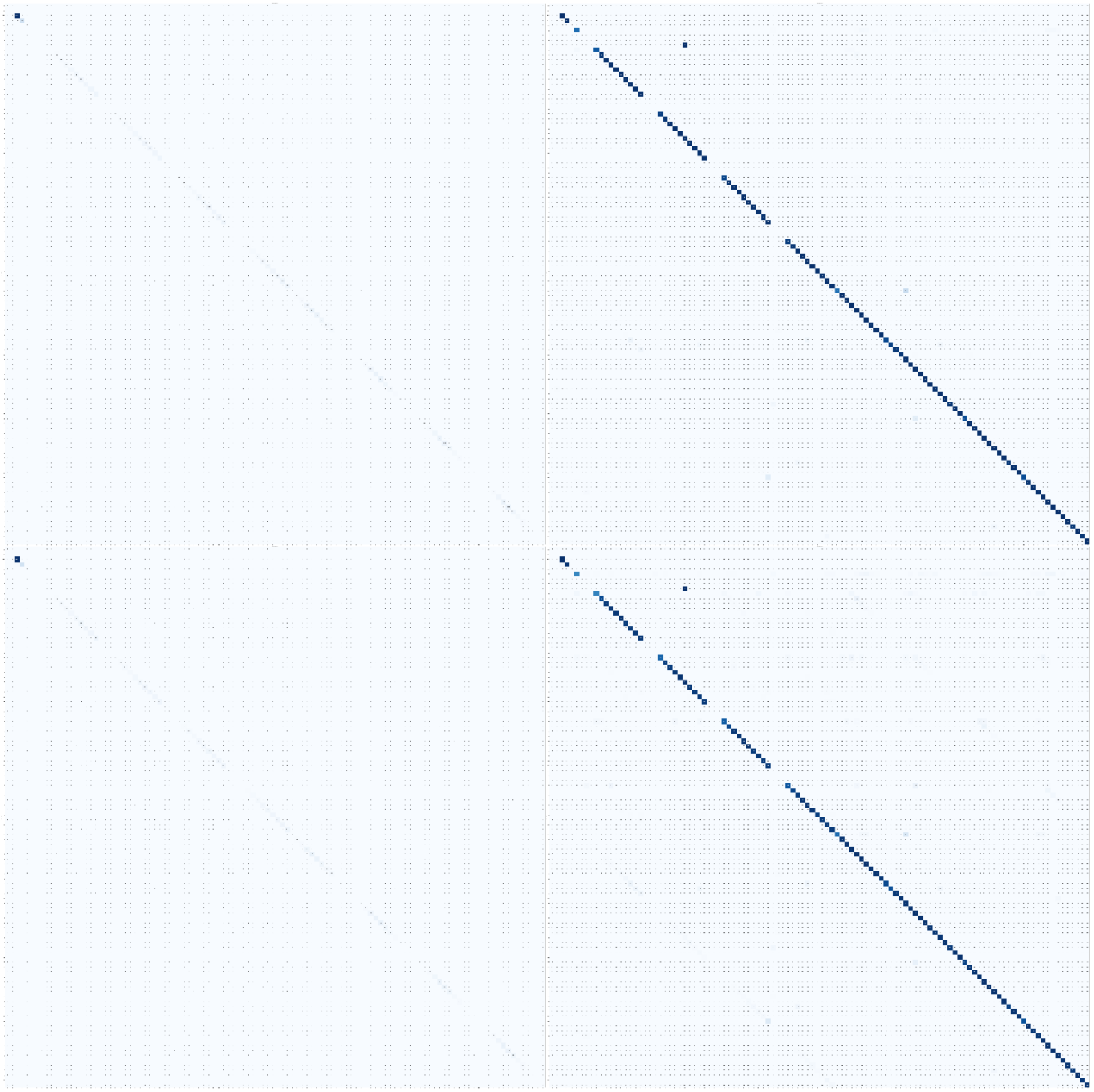

*Figure A23: Confusion Matrices for **all_2HL_80K***
Train (top), Validation (bottom), Normalized Values (right).
True Label (y-axis), Predicted Label (x-axis). Darker tile = larger number.

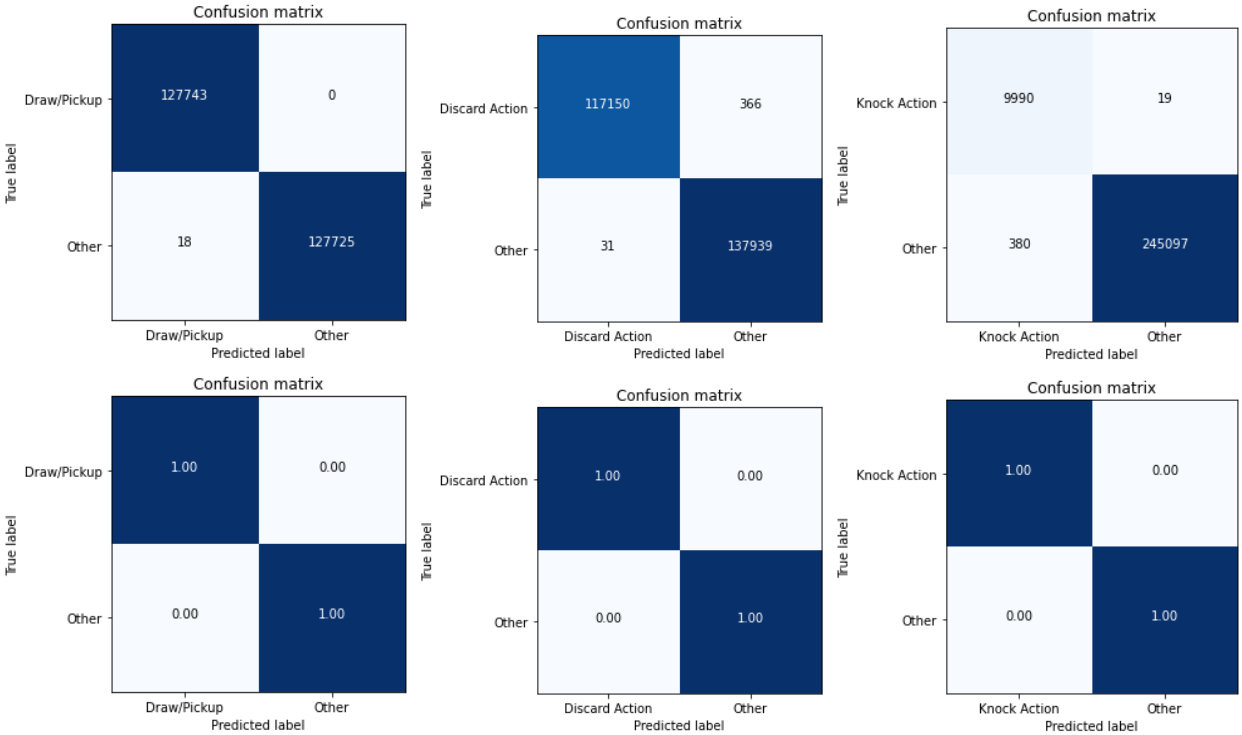

Figure A24: Confusion Matrices for **all_2HL_80K** based on state-action groups*
***'draw'** (left), **'discard'** (middle), **'knock'** (right), Normalized Values (bottom).*
*True Label (y-axis), Predicted Label (x-axis). Darker tile = larger number. *2K Games Test Set*

From the results of the training, **all_2HL_80K**, a final validation accuracy of 98.1% was achieved, a slight improvement to the previous model. The confusion matrices from Figure 23 illustrate the following:

1) **knock** actions are not rarely classified to **discard** action (Fig. A24, right) compared to prior (Figs. A15 & A19, right). The faint diagonal within the True **knock**, Predicted **discard** section of the confusion matrix (Fig. A19, bottom right, bottom middle of the confusion matrix) is no longer visible, indicating better classification of true **knock**.

2) *discard 3S* is being misclassified as *discard 5H*

- From the data distribution, it can be noted that there are only 3 samples of **3S** (1 sample in Validation Set), as the SimpleGinRummyPlayer agent policy is to avoid discarding low ranked cards

The confusion matrices from Figure A24 also show drastic improvements compared to the **all_1HL** model, with all state-action groups having a high degree of accuracy predicting the true labels (Fig. A24, bottom). It can be noted, however, that the number of **discard** misclassification increased (Fig. A24, top middle).

A5: Supervised Network Testing

This section is dedicated to understanding the lower-level features that the Supervised Neural Networks have learned.

A5.1: Meld Testing

One lower-level feature that may be learned by the models are different combination of melds. A strategy in Gin Rummy is to ensure to not break meld sets within the agent's hand as doing so will increase deadwood. As indicated prior, there are two types of melds:

1) a run – three or more cards of the same suit in a row,

2) a set – three or more cards of the same rank.

In total, there are a possible of 65 different sets (13 ranks * (4 suits **C** 3 cards = 4) + 13 ranks * (4 suits **C** 4 cards = 1)) and 240 different runs ((11 runs of 3 cards + 10 runs of 4 + … + 5 runs of 9 + 4 runs of 10 = 60 different runs for 1 suit) * 4 suits) for a total of 305 different melds.

In the **apad_knock** model, the output of the model is a binary classification; knock or no knock. To verify that **apad_knock**, and additional models have learned this feature, a random **knock** state can be provided to the network to check the activation/weights of the units. Sets of hidden units with high weights corresponding to the particular cards that form a meld would likely indicate a correlation between the network learning the melds. This approach can be generalized to all possible single sets of melds. Additionally, given the **knock** state, there exists at least one meld in the player's hand. Replacing a card within the meld would break the meld, which should be reflected in the unit weights and model output. Some work has been done on this section, testing a simpler model (**apad_knock** but 1 feature plane of current hand only instead of 5) with promising results suggesting that the model was able to learn these melds.

Appendix B: Reinforcement Learning Environment

The following are non-default hyperparameters used to train the agents unless otherwise stated:

- Learning Rate = 5e-5
- MLP = [*260*, **520, 520**, *110*] = [*input*, **HL1**, **HL2**, *output*] (*italicized* are default sizes)
- Sigmoid Activation between layers, Softmax after final layer
- Train Every 10 steps (default = 1)
- Batch Size = 64 (default = 32)
- Initial Memory Size = 1000 (default = 100)
- Number of Episodes = 20000

B1: With vs. Without Pre-Initialization

This section focuses on the discussing the results between a DQN agent without Pre-Initialization (random initialization) and three baseline DQN agents initialized using the models trained in Appendix A4. Each agent was trained against the SGRAgent and evaluated against the random agent, SGRAgent and self-play whenever possible.

B1.1: Without Pre-Initialization

dqn_baseline serves as a baseline for all DQN agents. The macro average reward against the random agent and SGRAgent is approximately -0.6 for both, which serves as a benchmark when comparing future training iterations. Note that this agent was not trained against self-play as the agent was unable to be evaluated.

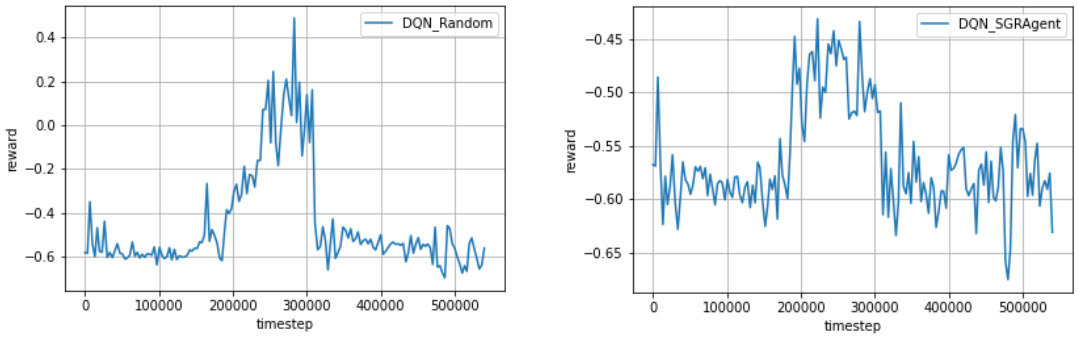

*Figure B1: Micro Average Reward plot for **dqn_baseline** against Random (Left), SGRAgent (Right)*

It can be noted that there is an increase in reward midway through training **dqn_baseline** against the both agents, suggesting learning occurred. However, further investigation into the number of turns and count of actions provide a better insight to this occurrence, seen in Figure B2. The average number of turns required by **dqn_baseline** against the random agent is approximately 80 turns (Fig. B2, top left), a large difference compared to the average number of turns in the *Supervised Learning Environment* of 10. Additionally, the evaluation against SGRAgent shows no noteworthy performance in terms of **gin** or **knock** actions (Fig. B2, bottom right).

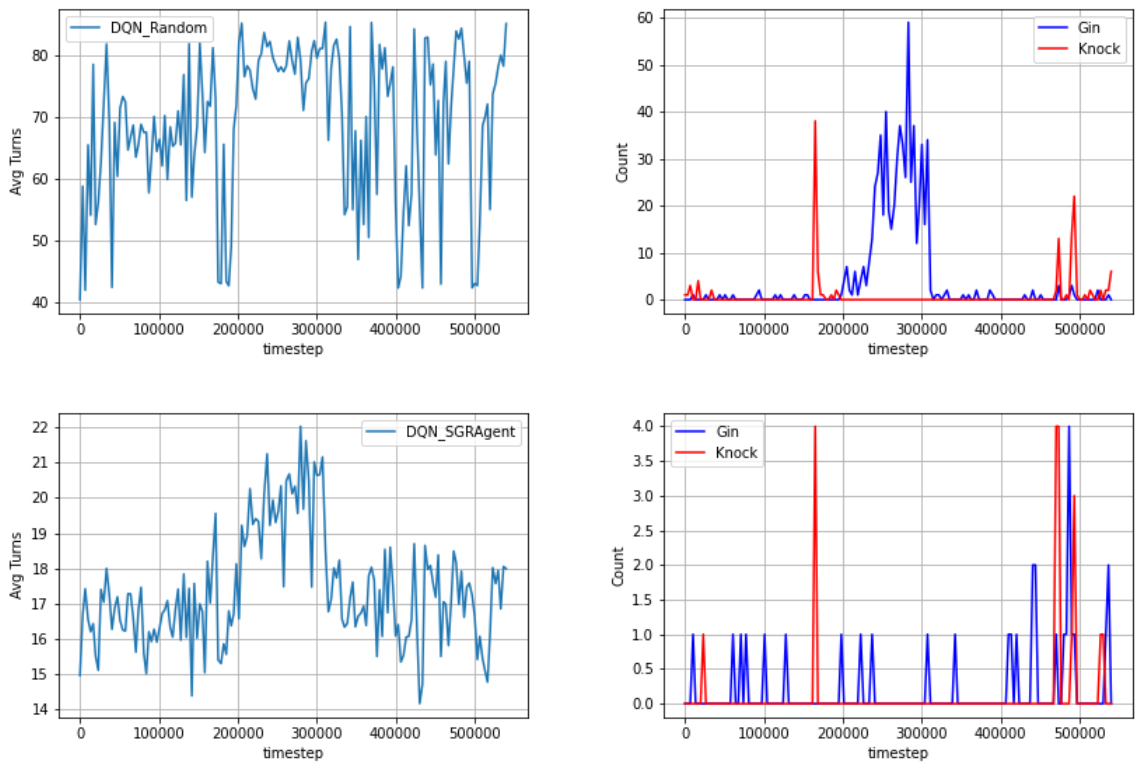

*Figure B2: **dqn_baseline** plots*
*Average Turns **(**Left) and Action Count (Right)*
against Random Agent (Top) and SGRAgent (Bottom)

B1.2: With Pre-Initialization

Using the three **all state, all action** models produced in Appendix A5 (**all_1HL, all_2HL_40K, all_2HL_80K**), the network weights were used to pre-initialize the DQN Neural Network parameters. Layer weights were kept unfrozen to see the evolution of each model during the training process in this environment without further assistance. This was to test the macro average reward of each DQN agent and provide a baseline for each of these models.

B1.2.1: dqn_1HL

The final micro average rewards against the random agent, self-play, and SGRAgent for **dqn_1HL** are approximately 0.25, 0.4 and -0.15 respectively. The reward against the random agent (Fig. B3, left) is slightly above the threshold of 0.2 required to **knock**. This indicates that **dqn_1HL** is able to **knock** and **gin** each round against the random agent. The reward plot against self-play suggests that there is learning as **dqn_1HL** (Fig. B3, middle) is obtaining a reward larger than the prior to training. However, the reward plot against SGRAgent (Fig. B3, right) suggests otherwise, as there is a downward trend. Further investigation in the other evaluation metrics provide more insight.

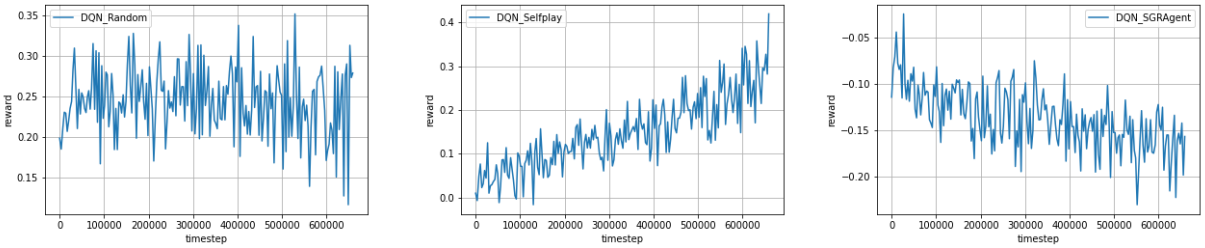

*Figure B3: Micro Average Reward plots for **dqn_1HL***
against Random (Left), Self-play (Middle), and SGRAgent (Right)

The following are non-default hyperparameters used to train **dqn_1HL**:

- MLP = [260, 520, 110] = [input, HL1, output] (italicized are default sizes)

Figure B4 displays the same trends observed in **dqn_baseline**. As the agent interacts more within the *DQN Environment*, the average number of turns increases, with a drop of the ability to perform **knock** actions, as observed in Figure B4, middle and right. Additionally, the agent tends to perform more **gin** actions, which may be an indication that the agent is able to receive the **gin** reward.

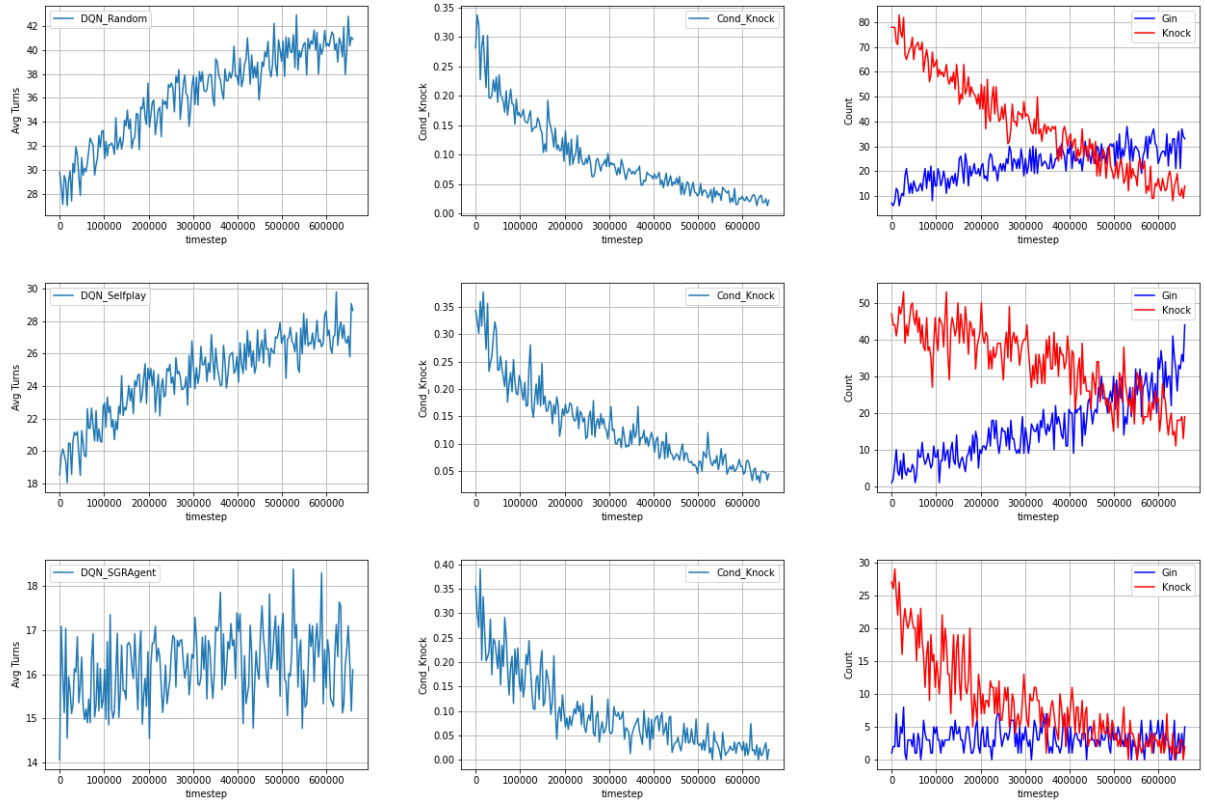

*Figure B4: **dqn_1HL** plots*
*Average Turns **(**Left), Conditional Knock Probability (Middle), and Action Count (Right)*
against Random Agent (Top), Self-Play (Middle), SGRAgent (Bottom)

B1.2.2: dqn_2HL_40K

The final micro average rewards against the random agent, self-play, and SGRAgent for **dqn_2HL_40K** are approximately 0.3, 0.1 and -0.15 respectively (Fig. B5). Against both Random and Self-play, the average reward saw a positive trend however the reward against SGRAgent trends negatively, similar to the results of **dqn_1HL**.

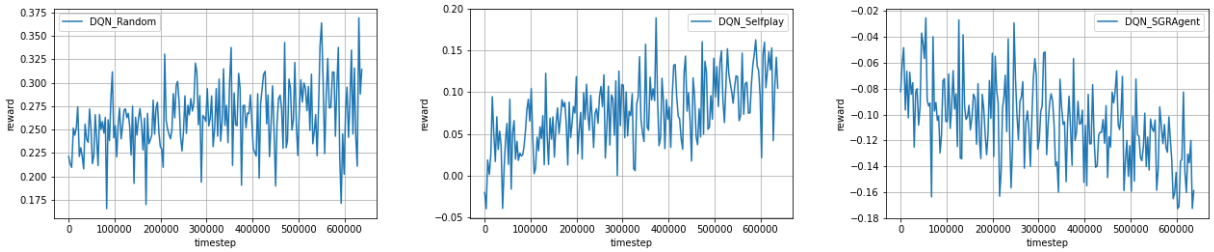

*Figure B5: Micro Average Reward plots for **dqn_2HL_40K** against Random (Left), Self-play (Middle), and SGRAgent (Right)*

Figure B6 displays the same trends observed in **dqn_1HL**. However, this agent requires fewer average turns and retained some ability to **knock** (Fig. B6, right). Note that against SGRAgent, the initial count of **knock** are approximately 30-35 (Fig. B6, bottom right).

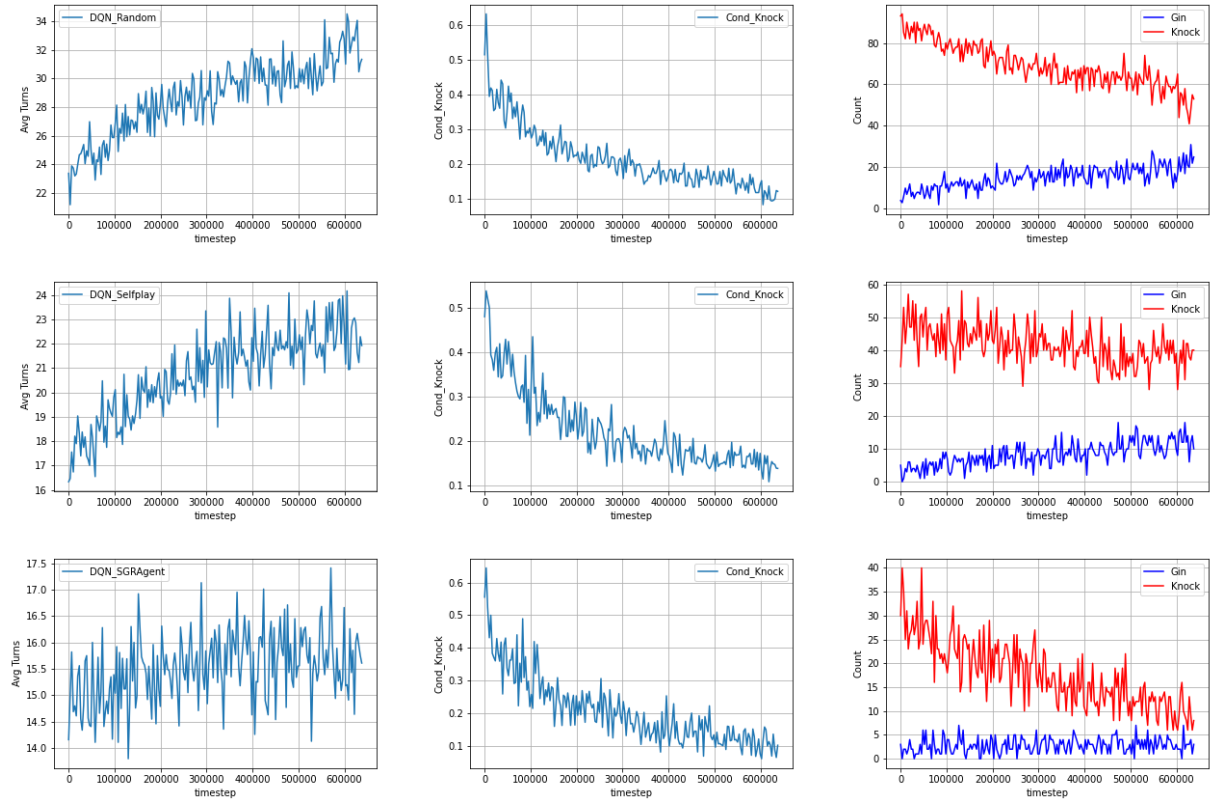

*Figure B6: **dqn_2HL_40K** plots*
*Average Turns **(**Left), Conditional Knock Probability (Middle), and Action Count (Right) against Random Agent (Top), Self-Play (Middle), SGRAgent (Bottom)*

B1.2.3: dqn_2HL_80K

The final micro average rewards against the random agent, self-play, and SGRAgent for **dqn_2HL_80K** are approximately 0.3, 0.1 and -0.125 respectively (Fig. B7). These results are similar to both **dqn_1HL** and **dqn_2HL_40K**.

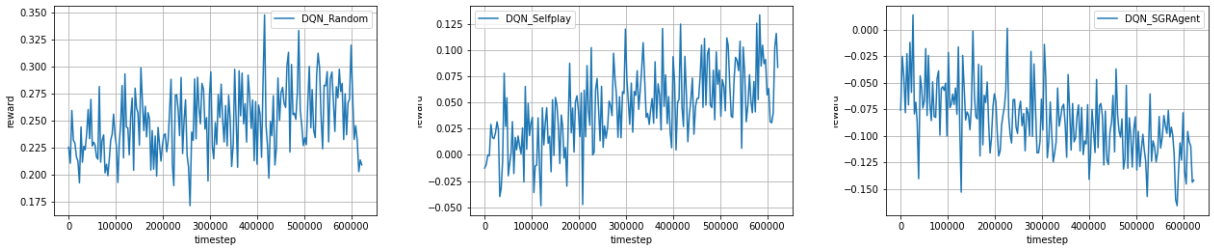

*Figure B7: Micro Average Reward plots for **dqn_2HL_80K** against Random (Left), Self-play (Middle), and SGRAgent (Right)*

Figure B8 displays the same trends observed in **dqn_2HL_40K**. Note that against SGRAgent, the initial count of **knock** actions are approximately 35-40 (Fig. B8, bottom right).

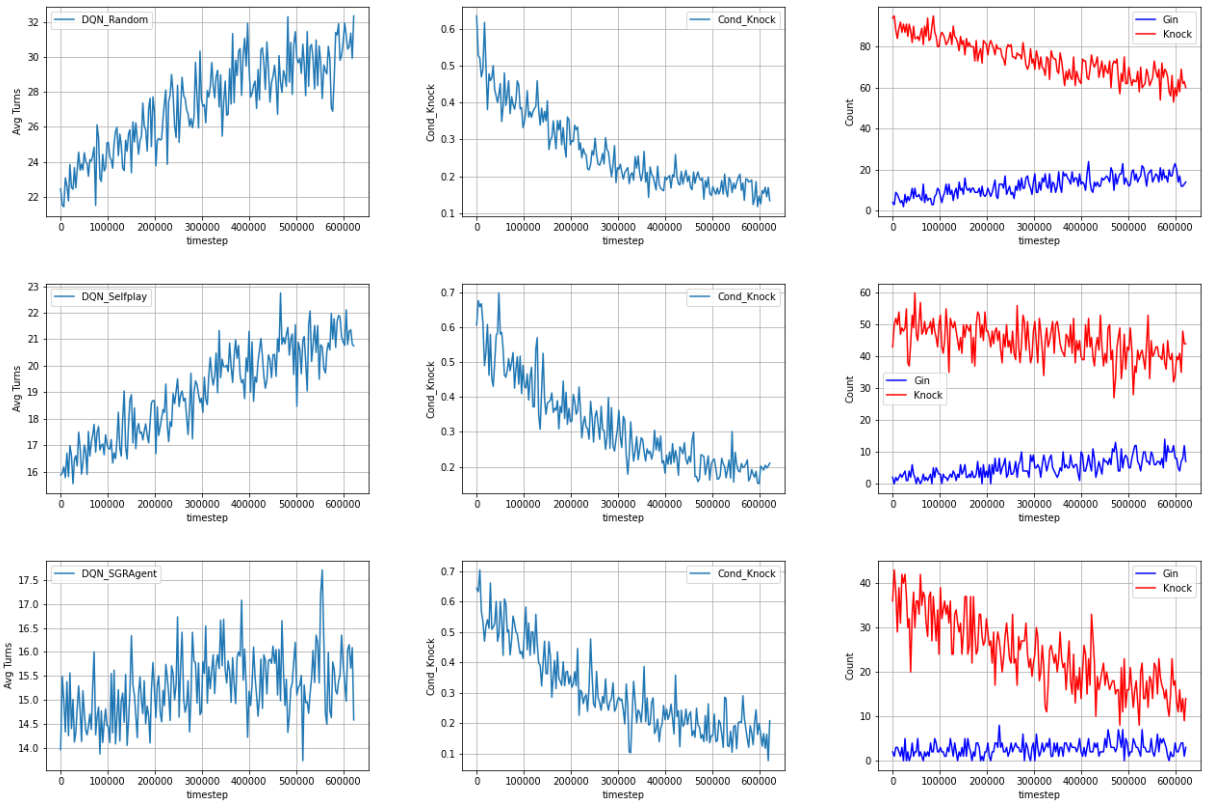

*Figure B8: **dqn_2HL_80K** plots*
*Average Turns **(**Left), Conditional Knock Probability (Middle), and Action Count (Right) against Random Agent (Top), Self-Play (Middle), SGRAgent (Bottom)*

B1.3: Baseline Results Summary

Important conclusions that can be drawn through the Pre-Initialization baseline models are:

- Continual training within the *Reinforcement Environment* improves the performance of the agent against the Random Agent and Self-play (Fig. B3, B5 & B7, left & middle) however decreases the performance against SGRAgent (Fig. B3, B5 & B7, right)
- The Average number of turns has a positive trend (Fig. B4, B6 & B8, left) while the Conditional Knock Probability has a negative trend (Fig. B4, B6 & B8, middle) for all sets of evaluation
- The count of **gin** actions increase while the count of **knock** actions decrease during training (Fig. B4, B6 & B8, right)
- **dqn_2HL_80K** had the best performance against the SGRAgent, with an initial **knock** count of 40

With this information, **dqn_2HL_80K** and its Pre-Initialization model **all_2HL_80K** will be the primary baseline for all subsequent experiments.

B2: Frozen vs. Unfrozen Layers

This section will highlight the difference in performance between the Pre-Initialized DQN agent with bottom layers frozen and unfrozen.

B2.1: Unfrozen Layers

Refer to **dqn_2HL_80K** of Appendix B1.2.3.

B2.2: Frozen Layers

The macro average rewards against the random agent, self-play, and SGRAgent for **dqn_frozen** are approximately 0.25, 0.025 and -0.08 respectively (Fig. B9). These results are comparable to the rewards from the unfrozen model shown in Figure B9.

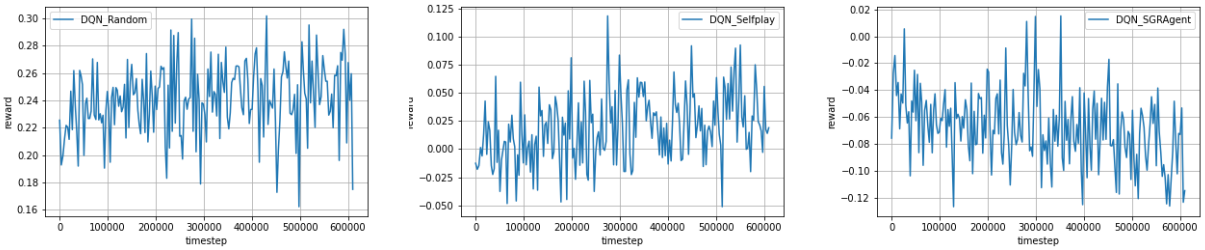

*Figure B9: Micro Average Reward plots for **dqn_frozen** against Random (Left), Self-play (Middle), and SGRAgent (Right)*

The following are non-default hyperparameters used to train **dqn_frozen**:

- MLP = [260, **520, 520**, 110] = [*input*, **HL1, HL2,** *output*] (italicized are default sizes)
- Frozen bottom layers, unfrozen top (output) layer

Figure B10 displays the evaluation results against all three agents. Note that training **dqn_frozen** also exhibits the same trends seen in prior agents, with the increase in number of turns taken, decrease in conditional knock probability, and increase in the count of **gin** actions with a decrease in **knock** actions.

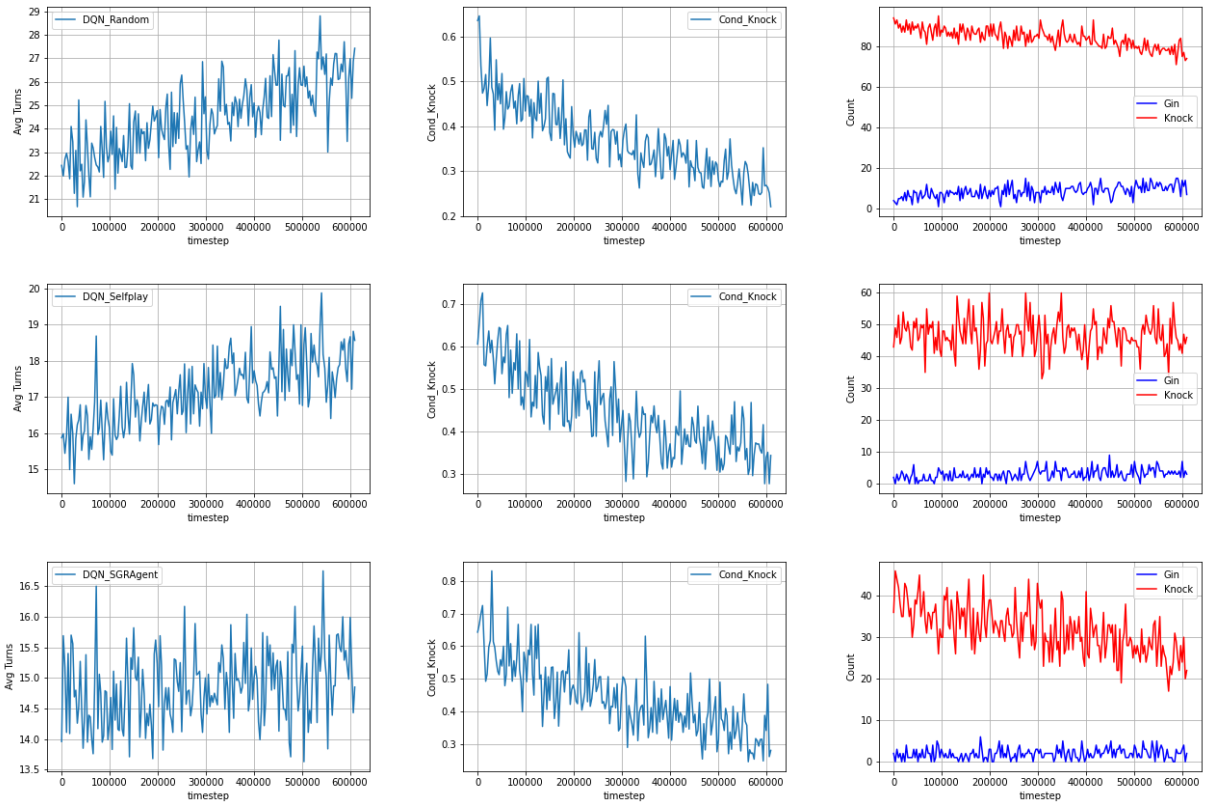

Figure B10: ***dqn_frozen*** *plots*
Average Turns ***(****Left), Conditional Knock Probability (Middle), and Action Count (Right) against Random Agent (Top), Self-Play (Middle), SGRAgent (Bottom)*

B2.3: Frozen Layers Summary

Comparing the results between the unfrozen agent, **dqn_2HL_80K**, and the frozen agent, **dqn_frozen**, shows that both agents suffer from learning within the environment when it pertains to evaluating against the SGRAgent (Fig. B7 & B9, right). Moreover, both agents tend to lose their ability to **knock**, which can be observed in all agents up to this point, however **dqn_frozen**, given the same training parameters, has a slower descent (Fig. B8 vs B10, bottom middle), thus agents moving forward will be trained using frozen bottom layers.

B3: Various Reward Structures

This section will highlight the various rewards used to train the baseline agent **all_2HL_80K** with **frozen layers** listed in the table below. Note that the rewards are based around incentivizing the agent to **knock** as all prior agents have a decreasing conditional knock probability. Increasing the reward for a **knock** action should positively reinforce the policy that the agent used to arrive to this action more significantly compared to a lower **knock** reward.

Table 4: Environment Rewards and Agent Names

agent name	Gin	Knock	Other
dqn_frozen	1	0.2	- 0.01 / Deadwood
dqn_2xKnock	1	0.4	- 0.01 / Deadwood
dqn_5xKnock	1	1	- 0.01 / Deadwood
dqn_50xKnock	1	10	- 0.01 / Deadwood
dqn_50xKnock_pos	1	10	0
dqn_50xKnock_only	0	10	0

B3.1: dqn_2xKnock

The final micro average rewards against the random agent, self-play, and SGRAgent for **dqn_2xKnock** are approximately 0.4, 0.125 and -0.05 respectively (Fig. B11). Note that these results are approximately double the rewards of the baseline reward agent, **dqn_frozen**.

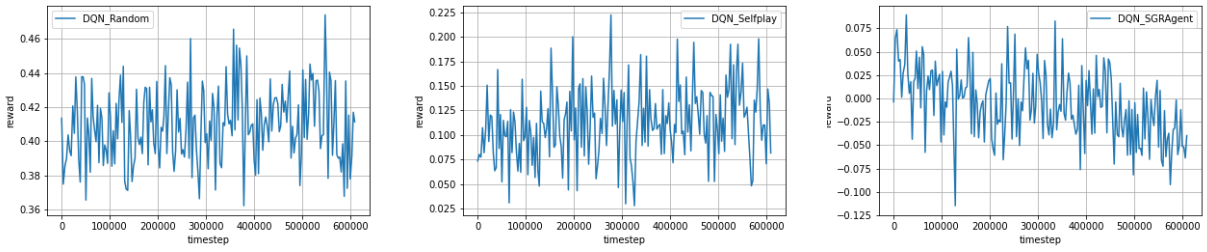

*Figure B11: Micro Average Reward plots for **dqn_2xKnock** against Random (Left), Self-play (Middle), and SGRAgent (Right)*

Figure B12 displays the evaluation results against all three agents. Note that despite the additional reward on **knock, dqn_2xKnock** still exhibits the same trends seen in **dqn_frozen**, with the increase in number of turns taken, decrease in conditional knock probability, and increase in the count of **gin** actions with a decrease in **knock** actions.

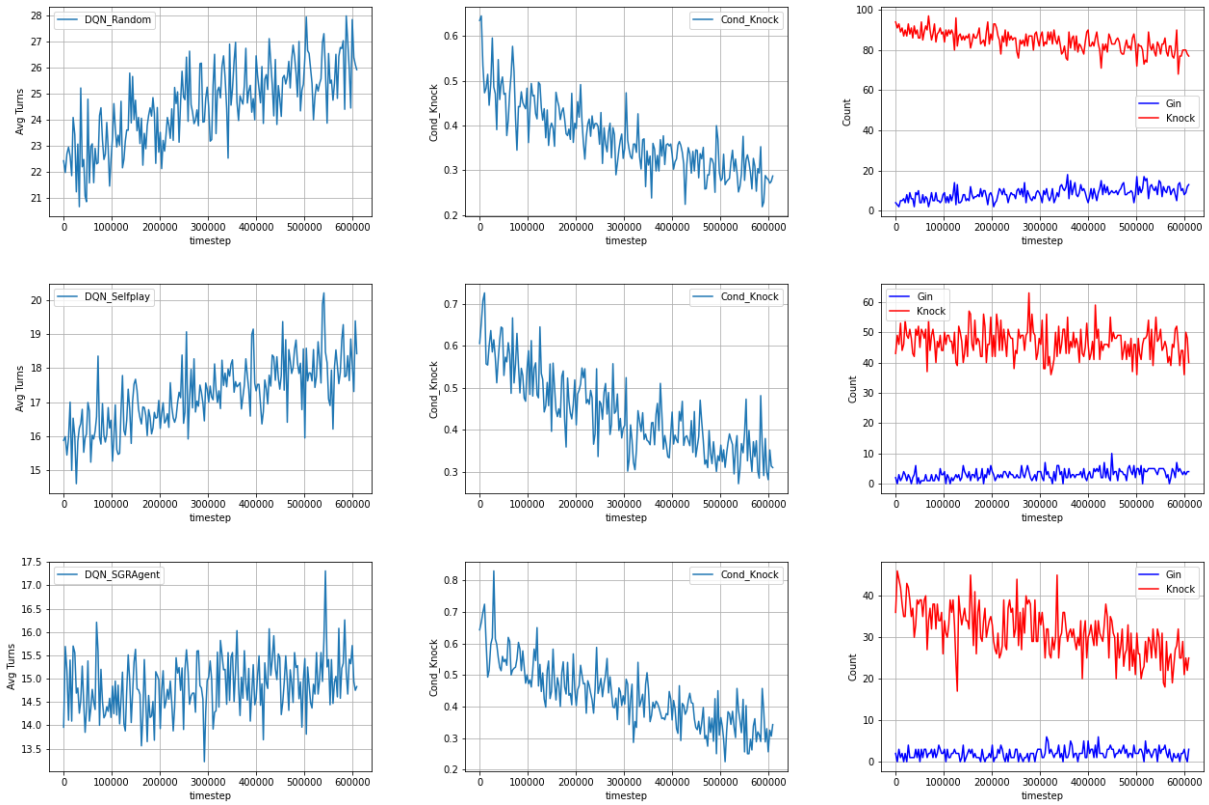

*Figure B12: **dqn_2xKnock** plots*
*Average Turns **(**Left), Conditional Knock Probability (Middle), and Action Count (Right)*
against Random Agent (Top), Self-Play (Middle), SGRAgent (Bottom)

B3.2: dqn_5xKnock

The final micro average rewards against the random agent, self-play, and SGRAgent for **dqn_5xKnock** are approximately 0.9, 0.4 and 0.1 respectively (Fig. B13).

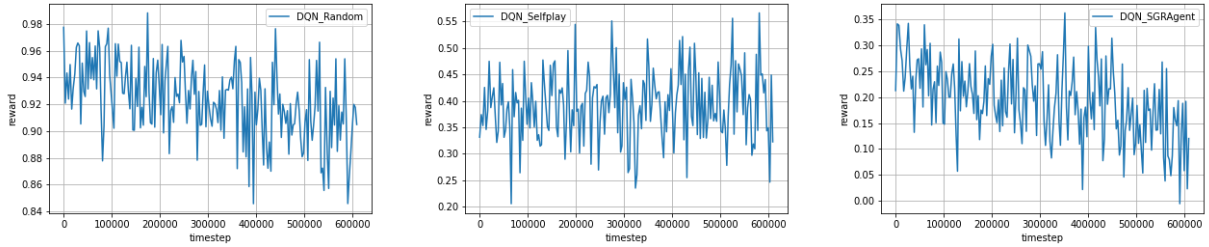

*Figure B13: Micro Average Reward plots for **dqn_5xKnock***
against Random (Left), Self-play (Middle), and SGRAgent (Right)

Figure B14 displays the evaluation results against all three agents. Note that despite the additional reward on **knock, dqn_5xKnock** exhibits the same trends seen in **dqn_frozen** and **dqn_2xKnock**, with the increase in number of turns taken, decrease in conditional knock probability, and increase in the count of **gin** actions with a decrease in **knock** actions.

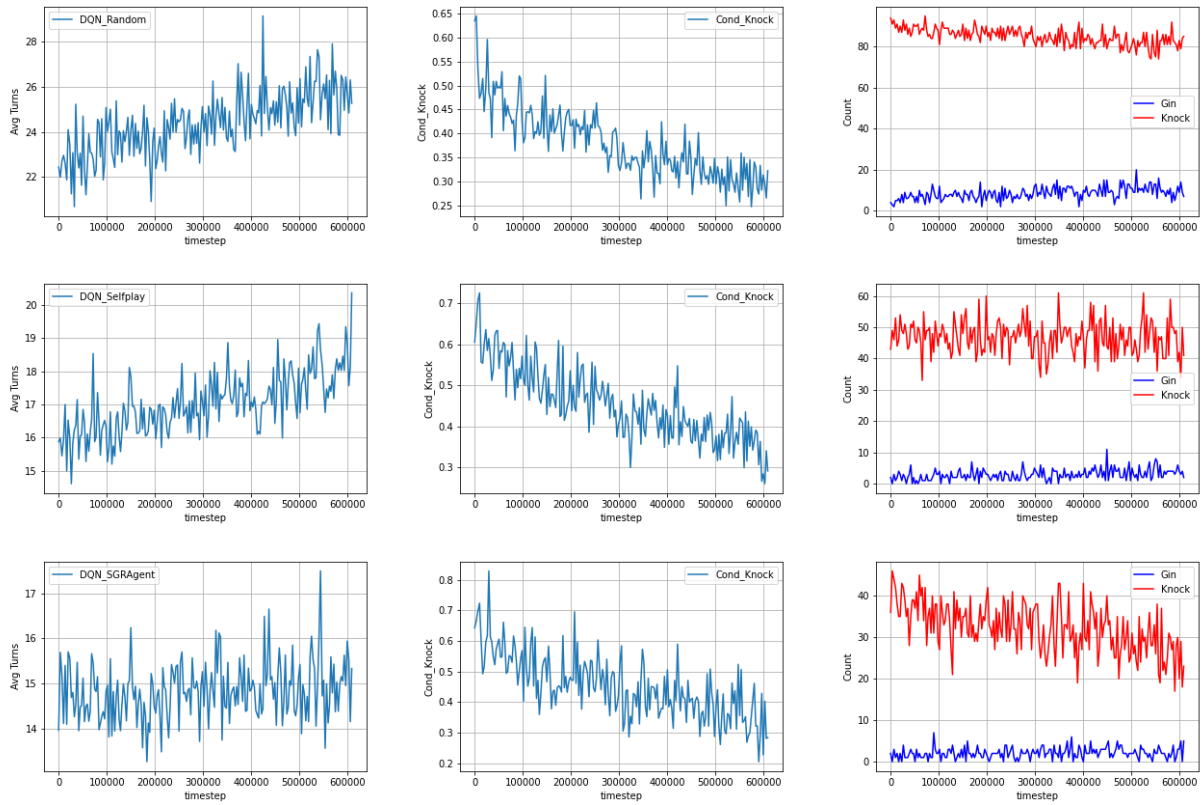

*Figure B14: **dqn_5xKnock** plots*
*Average Turns **(**Left), Conditional Knock Probability (Middle), and Action Count (Right)*
against Random Agent (Top), Self-Play (Middle), SGRAgent (Bottom)

B3.3: dqn_50xKnock

The final micro average rewards against the random agent, self-play, and SGRAgent for **dqn_50xKnock** are approximately 8.25, 4.0 and 1.25 respectively (Fig. B15). Note that these results are approximately 10 times the rewards of **dqn_5xKnock**. Additionally, all rewards exhibit a negative trend, demonstrating that the agent prefers other actions over **knock**.

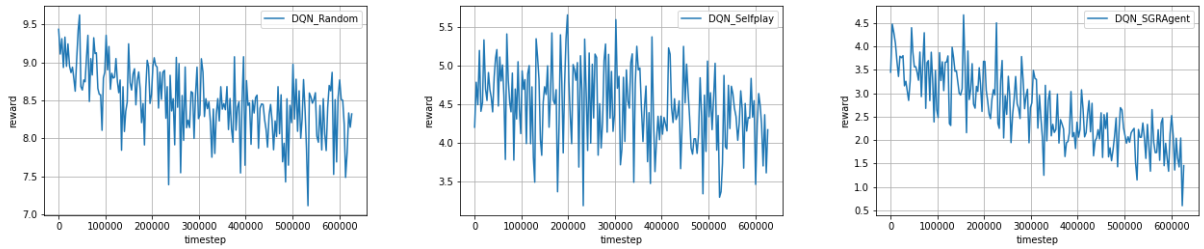

*Figure B15: Micro Average Reward plots for **dqn_50xKnock***
against Random (Left), Self-play (Middle), and SGRAgent (Right)

Figure B16 displays the evaluation results against all three agents. Note that despite the additional reward, **dqn_50xKnock** exhibits the same trends seen in all previous agents.

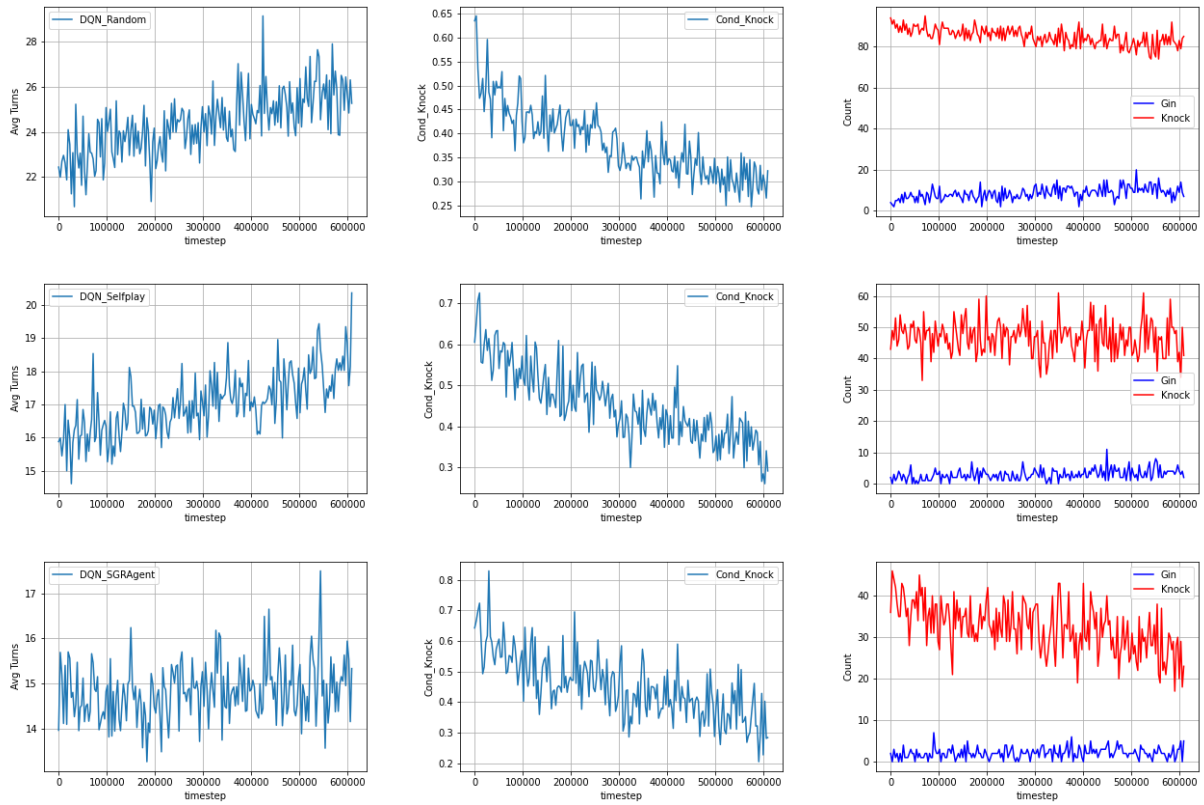

Figure B16: ***dqn_50xKnock*** *plots*
Average Turns ***(****Left), Conditional Knock Probability (Middle), and Action Count (Right)*
against Random Agent (Top), Self-Play (Middle), SGRAgent (Bottom)

B3.4: dqn_50xKnock_pos

The goal of this agent is to remove all negative rewards that may be used to penalize the agent. The final micro average rewards against the random agent, self-play, and SGRAgent for **dqn_50xKnock_pos** are approximately 8.25, 4.0 and 1.25 respectively (Fig. B17). Note that there are little differences between this agent and **dqn_50xKnock** in terms of rewards.

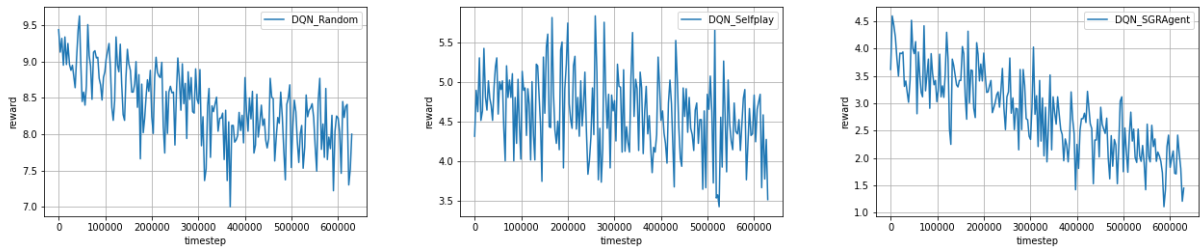

Figure B17: Micro Average Reward plots for ***dqn_50xKnock_pos***
against Random (Left), Self-play (Middle), and SGRAgent (Right)

Figure B18 displays the evaluation results against all three agents. Note that despite the removal of negative rewards, **dqn_50xKnock_pos** exhibits the same trends seen in all previous agents.

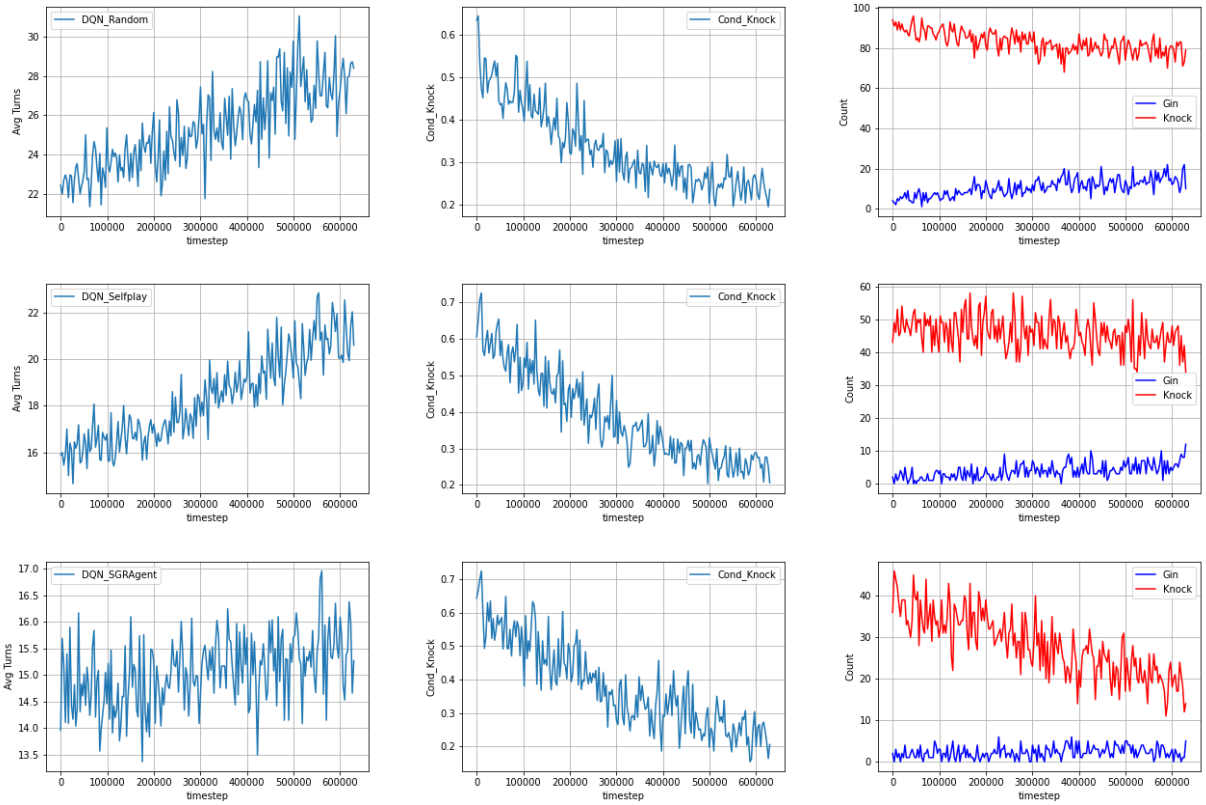

*Figure B18: **dqn_50xKnock_pos** plots*
*Average Turns **(**Left), Conditional Knock Probability (Middle), and Action Count (Right)*
against Random Agent (Top), Self-Play (Middle), SGRAgent (Bottom)

B3.5: dqn_50xKnock_only

The goal of this agent is to remove all non-**knock** rewards to see if these rewards have any influence on the agent. The final micro average rewards against the random agent, self-play, and SGRAgent for **dqn_50xKnock_only** are approximately 8.25, 4.0 and 1.0 respectively (Fig. B19). Note that there are little reward differences between this agent and **dqn_50xKnock**.

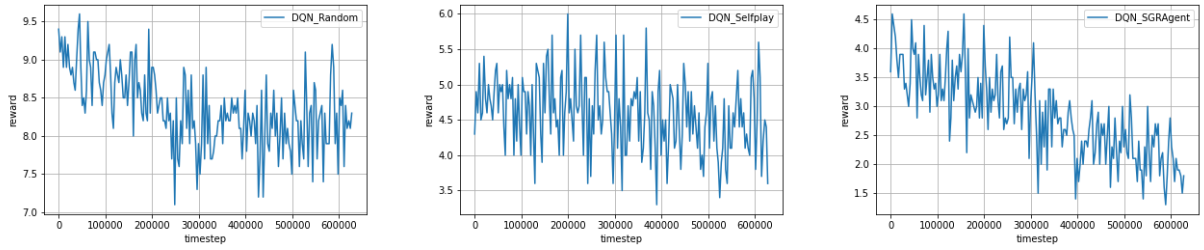

*Figure B19: Micro Average Reward plots for **dqn_50xKnock_only***
against Random (Left), Self-play (Middle), and SGRAgent (Right)

Figure B20 displays the evaluation results against all three agents. Note that despite the removal of all non-**knock** rewards, **dqn_50xKnock_only** exhibits the same trends seen in all previous agents.

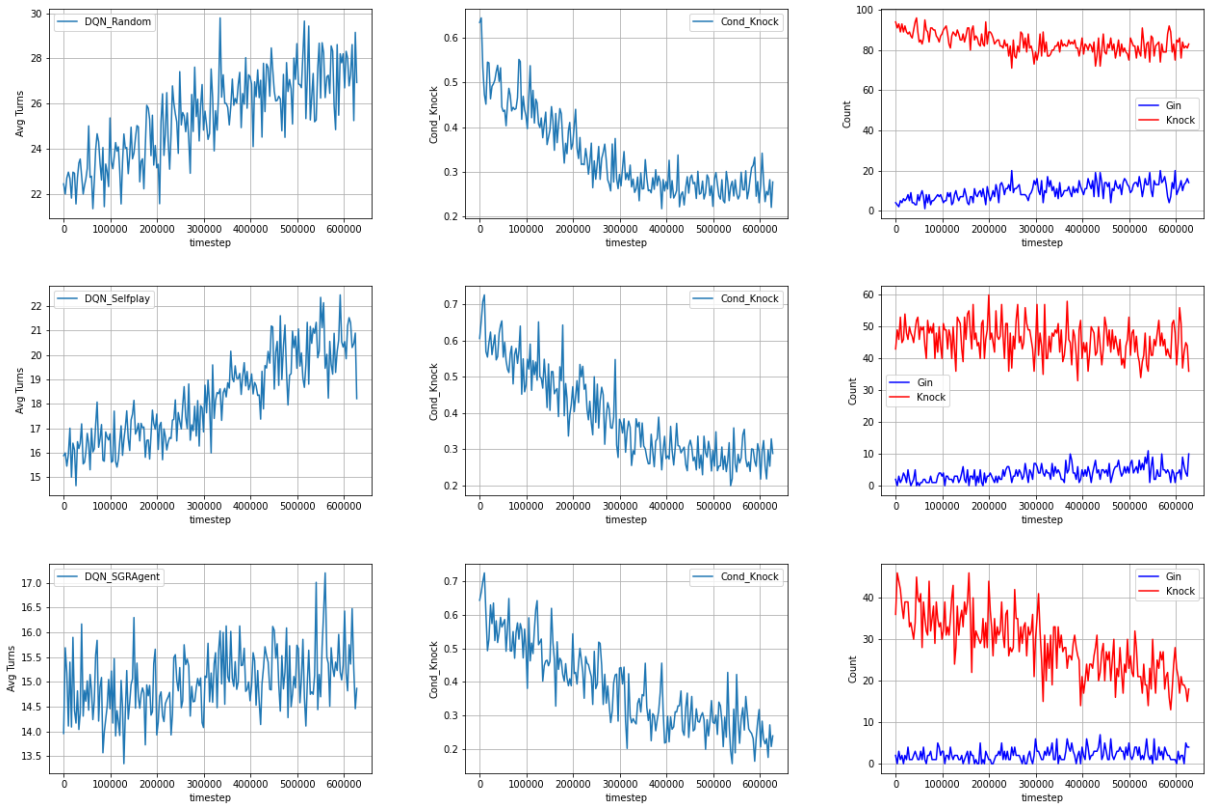

Figure B20: ***dqn_50xKnock_only*** *plots*
Average Turns ***(****Left), Conditional Knock Probability (Middle), and Action Count (Right)*
against Random Agent (Top), Self-Play (Middle), SGRAgent (Bottom)

B3.6: Various Rewards Summary

Varying the reward structure in the *Reinforcement Environment* played a minor to no role in contributing to the training of the agents. All agents consistently were unable to achieve a higher reward against the SGRAgent post-training. All agents had a negative trend for conditional knock probability while increased the number of turns required to reach an end state.

B4: Additional Top Layer

This section will highlight the different types of top layers that were added on top of the Pre-Initialized DQN agent. The goal is to determine if a top layer can be learned by the agent.

B4.1: Random Top Layer

The final micro average rewards against the random agent and SGRAgent for **dqn_random** are approximately -0.225 and -0.375 respectively (Fig. B21). Note the positive trend for rewards against both agents and self-play was not evaluated against itself was not computable.

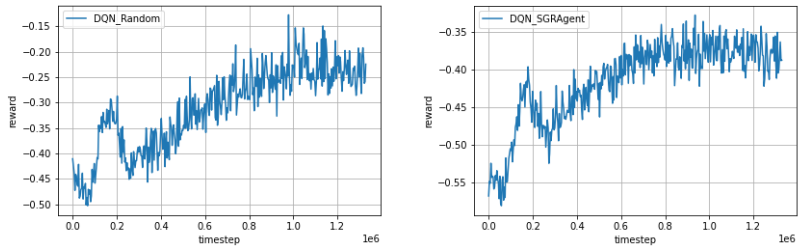

*Figure B21: Micro Average Reward plots for **dqn_random** against Random (Left), SGRAgent (Right)*

The following are non-default hyperparameters used to train **dqn_random**:

- MLP = [260, **520, 520**, 110**, 110**] = [*input*, **HL1, HL2,** *output*, **TL**] (*italicized* = default)
- Number of Episodes = **40000** (default = 20000)

Figure B22 displays the evaluation results against all random and SGRAgent. Atypical to previous agent results, there is a decrease in number of turns taken by this agent, however, similarly, a decreasing (in this case, 0%) conditional knock probability.

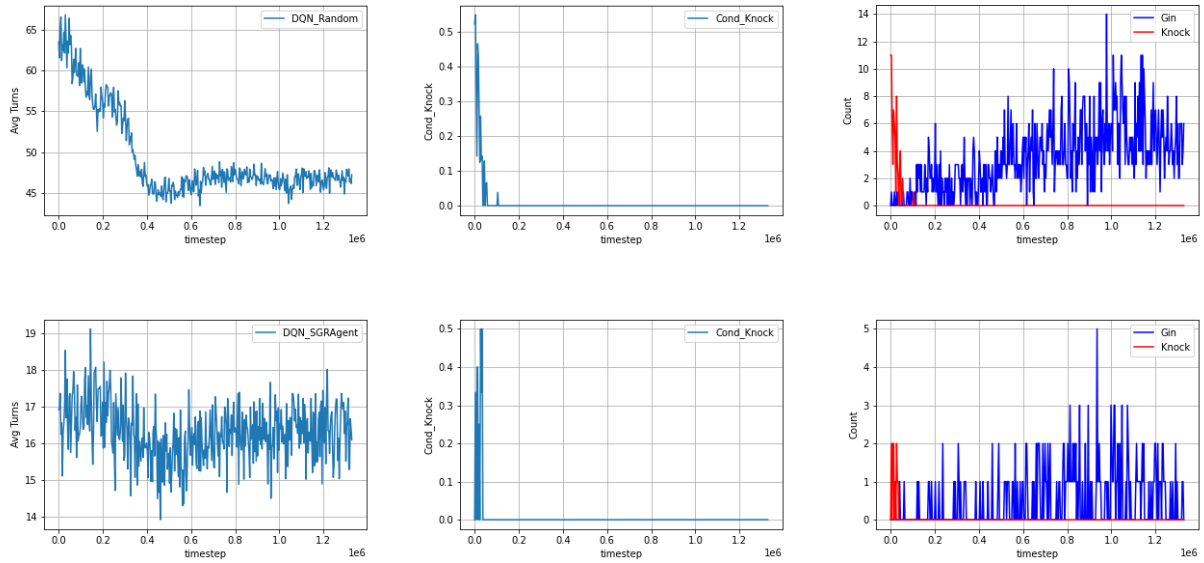

*Figure B22: **dqn_random** plots*
*Average Turns **(**Left), Conditional Knock Probability (Middle), and Action Count (Right) against Random Agent (Top), SGRAgent (Bottom)*

B4.2: Copied Top Layer

The goal of this experiment is to initialize the additional top layer with the identity matrix. This will allow the agent to initially perform similarly to the **dqn_frozen** agent, while allowing the additional top layer to be learned. The final micro average rewards against the random agent and SGRAgent for **dqn_copy** are approximately -0.1 and -0.375 respectively (Fig. B23). Note that the reward displays a negative trend.

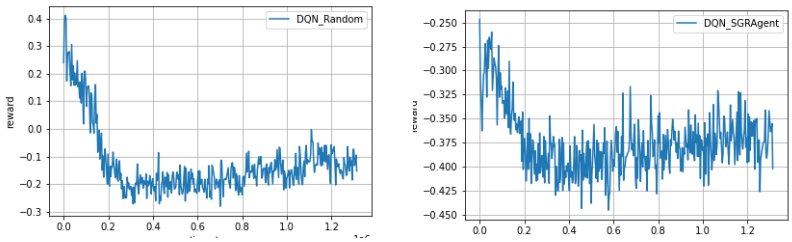

*Figure B23: Micro Average Reward plots for **dqn_copy** against Random (Left), SGRAgent (Right)*

The following are non-default hyperparameters used to train **dqn_copy**:

- MLP = [260, **520, 520**, 110**, 110**] = [*input*, **HL1, HL2,** *output*, **TL**] (*italicized* = default)
- **TL =** Identity Matrix, with bias 110-D vector of 0s, followed by Softmax activation
- Number of Episodes = **40000** (default = 20000)

Figure B24 displays the evaluation results against all random and SGRAgent. The steady state results of this agent is similar to **dqn_random**, exhibiting an average of 45 turns against Random agent (Fig B22, top left), with 0% conditional knock probability against both (Fig B22, middle).

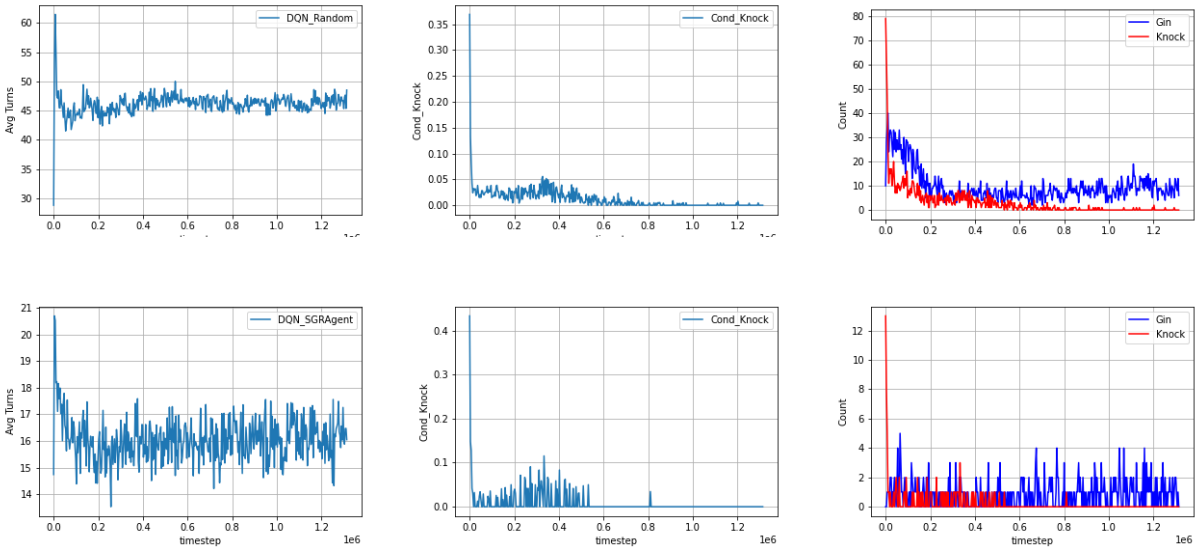

*Figure B24: **dqn_copy** plots*
*Average Turns **(**Left), Conditional Knock Probability (Middle), and Action Count (Right) against Random Agent (Top), SGRAgent (Bottom)*

B4.3: Pseudo-Identity Top Layer

This subsection will go in depth between various experiments performed using a Pseudo-Identity Top Layer. The Pseudo-Identity Matrix is defined by the following linear constraint:

$$Wy = y$$

$$where y = modeloutputlayer, w_{i,j} \in [-0.5, 0.5], i, j \in [1, 110]$$

As seen in section B4.2, the identity matrix of **dqn_copy** does not retain any information of the Pre-Initialization and converges to **dqn_random**. A possibility for this outcome are the parameters initialized to 0, which are the non-diagonal entries of the top layer. After some iterations of training the agent, the 0-value weights will become non-zero, activating other parameters that may not have been desired. The Pseudo-Identity Matrix will address the instability of using an Identity matrix as the Top Layer Initialization. To solve this linear constraint, a single layered network (no activation) was used to approximate the matrix, minimizing the mean squared error:

$$W = \underset{W}{argmin} \sum (Wy + b - y)^2$$

The following are the agents that will be used in the next set of experiments.

Table 5: Pseudo-Identity Agent Names

agent name	Dataset	Number of Samples Used
dqn_pseudo_all	*all /8K*	~500K (100%)
dqn_pseudo_10pct	*all /8K*	~50K (10%)
dqn_pseudo_5K	*all /8K*	5K (1%)

B4.3.1: dqn_pseudo_all

The entire 8K dataset was used to approximate the weight matrix, constraining the parameters every training loop. Below is the distribution after 200 iterations (Fig. B25). It can be observed that this weight matrix is very similar to the identity matrix, with the diagonals value at 0.5, the maximum allowable value.

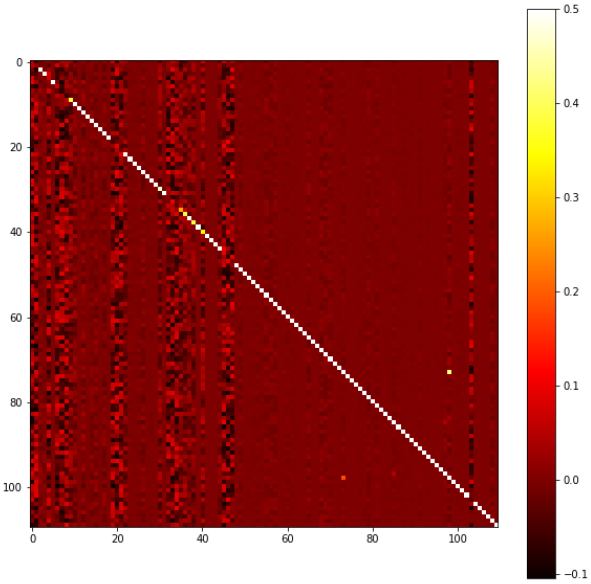

*Figure B25: Pseudo-Identity Matrix Distribution for **dqn_pseudo_all***

The final micro average rewards against the random agent, self-play, and SGRAgent for **dqn_pseudo_all** are approximately 0.2, 0 and -0.15 respectively (Fig. B26).

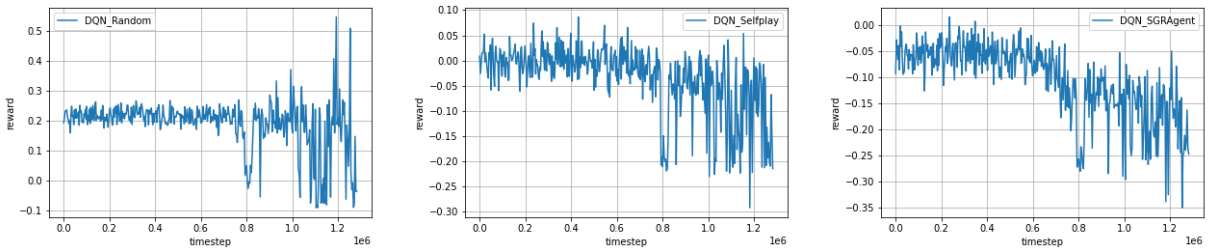

*Figure B26: Micro Average Reward plots for **dqn_pseudo_all** against Random (Left), Self-play (Middle), and SGRAgent (Right)*

The following are non-default hyperparameters used to train **dqn_pseudo_all**:

- MLP = [260, **520, 520**, 110**, 110**] = [*input*, **HL1, HL2,** *output*, **TL**] (*italicized* = default)
- **TL =** Pseudo-Identity Matrix, followed by Softmax activation
- Number of Episodes = **40000** (default = 20000)

Figure B27 displays the evaluation results against all three agents. Note that similar trends of increasing number of turns and decreasing conditional probability can be observed against all three agents.

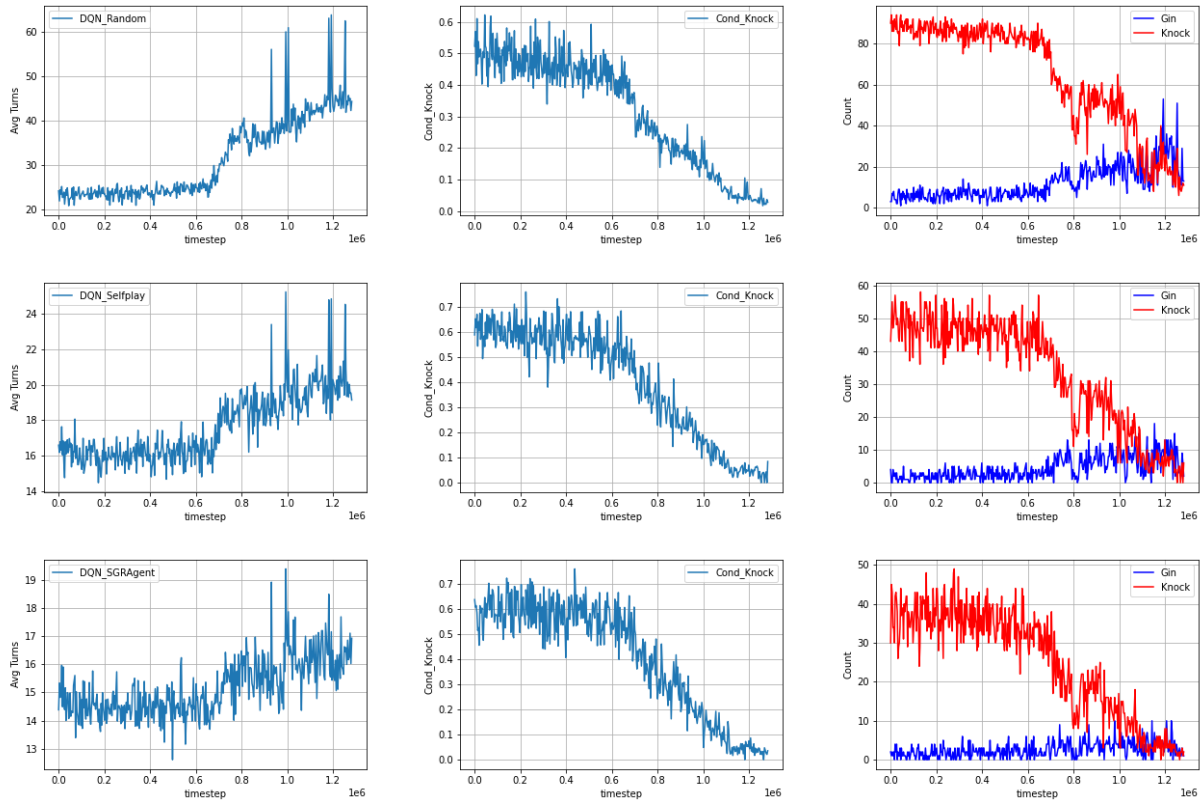

Figure B27: ***dqn_pseudo_all*** *plots*
Average Turns ***(****Left), Conditional Knock Probability (Middle), and Action Count (Right) against Random Agent (Top), Self-Play (Middle), SGRAgent (Bottom)*

B4.3.2: dqn_pseudo_10pct

Ten percent of the entire 8K dataset was used to approximate the weight matrix, constraining the parameters every training loop. Below is the distribution after 200 iterations (Fig. B28). It can be observed that this weight matrix does not resemble the identity matrix as closely as Figure B25. The parameters that approximated this data are bounded between [-0.1, 0.5].

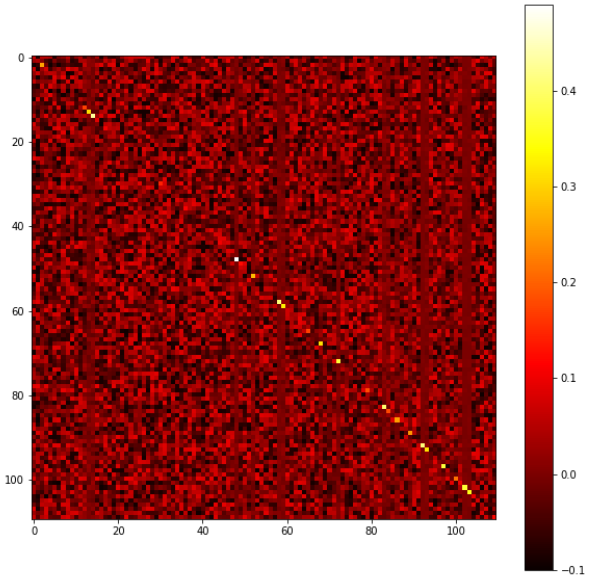

*Figure B28: Pseudo-Identity Matrix Distribution for **dqn_pseudo_10pct***

The final micro average rewards against the random agent, self-play, and SGRAgent for **dqn_pseudo_10pct** are approximately -0.15, -0.15 and -0.3 respectively (Fig. B29), all with a positive trend.

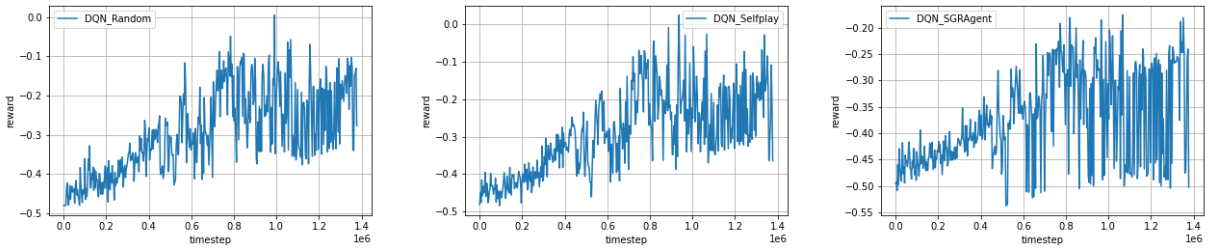

*Figure B29: Micro Average Reward plots for **dqn_pseudo_10pct** against Random (Left), Self-play (Middle), and SGRAgent (Right)*

The following are non-default hyperparameters used to train **dqn_pseudo_10pct**:

- MLP = [260, **520, 520**, 110**, 110**] = [*input*, **HL1, HL2,** *output*, **TL**] (*italicized* = default)
- **TL =** Pseudo-Identity Matrix, followed by Softmax activation
- Number of Episodes = **40000** (default = 20000)

Figure B30 displays the evaluation results against all three agents. Note that similar trends of increasing number of turns and decreasing conditional probability can be observed against all three agents.

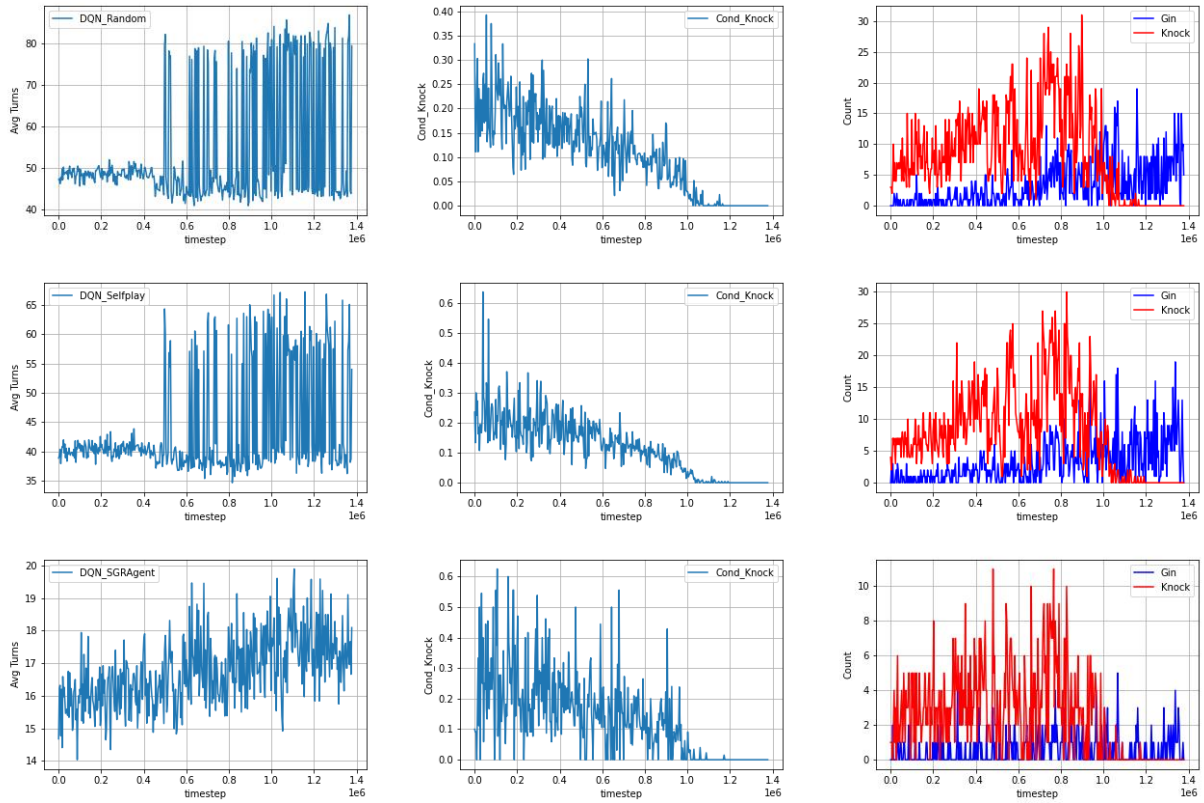

Figure B30: ***dqn_pseudo_10pct*** *plots*
Average Turns ***(****Left), Conditional Knock Probability (Middle), and Action Count (Right)*
against Random Agent (Top), Self-Play (Middle), SGRAgent (Bottom)

B4.3.3: dqn_pseudo_5K

Of the entire 8K dataset, 5000 samples were used to approximate the weight matrix, constraining the parameters every training loop. Below is the distribution after 200 iterations (Fig. B31). It can be observed that the parameters are much more distributed compared to the previous two matrices. The parameters that approximated this data are bounded between [-0.15, 0.25].

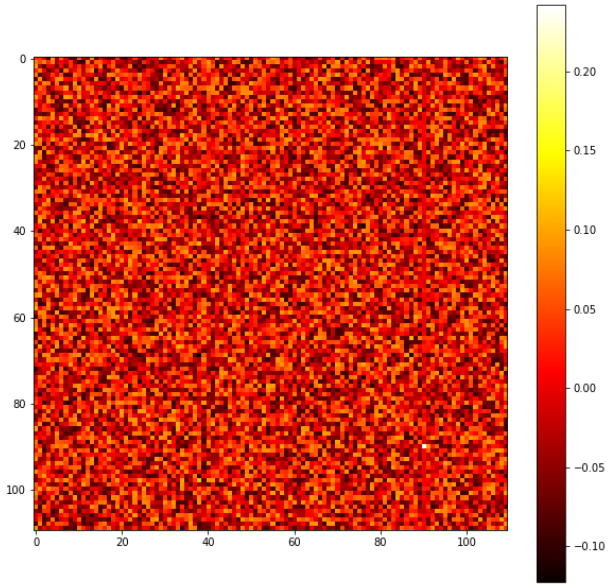

*Figure B31: Pseudo-Identity Matrix Distribution for **dqn_pseudo_5K***

The final micro average rewards against the random agent and SGRAgent for **dqn_pseudo_5K** are approximately -0.15, and -0.3 respectively (Fig. B32), all with a positive trend. Note that this agent was unable to be evaluated against Self-play.

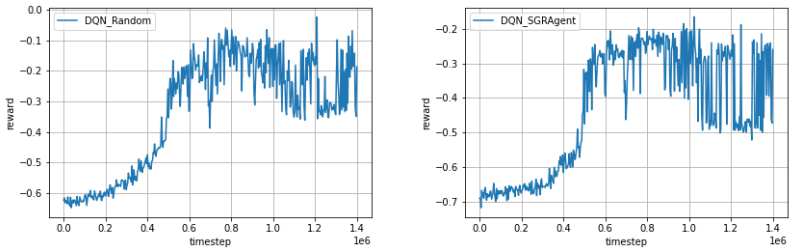

*Figure B32: Micro Average Reward plots for **dqn_pseudo_5K** against Random (Left), and SGRAgent (Right)*

The following are non-default hyperparameters used to train **dqn_pseudo_5K**:

- MLP = [260, **520, 520**, 110**, 110**] = [*input*, **HL1, HL2,** *output*, **TL**] (*italicized* = default)
- **TL =** Pseudo-Identity Matrix, followed by Softmax activation
- Number of Episodes = **40000** (default = 20000)

Figure B33 displays the evaluation results against the agents. Note that similar trends of increasing number of turns and decreasing conditional probability can be observed against all three agents.

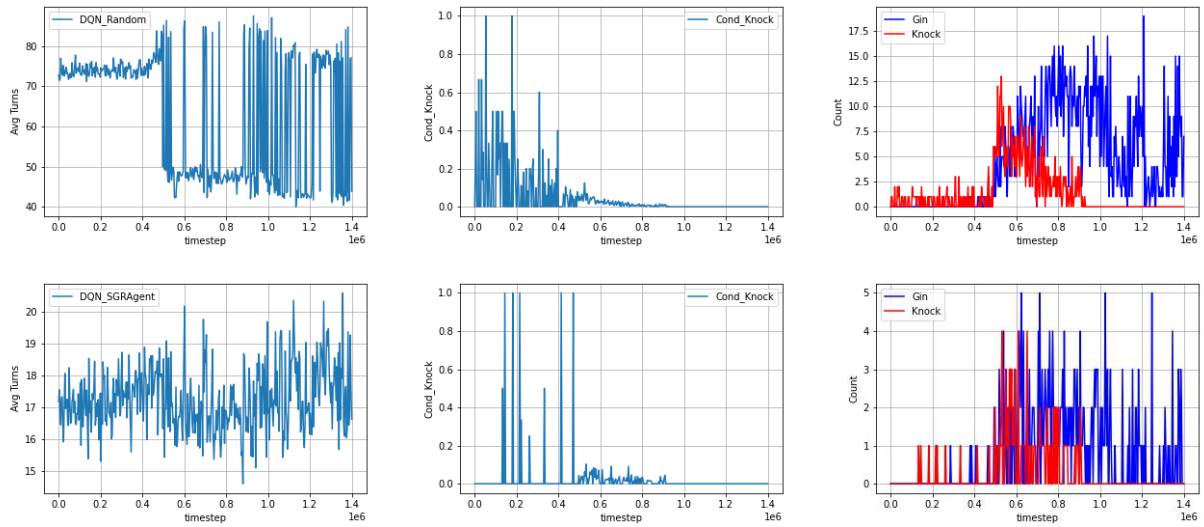

*Figure B33: **dqn_pseudo_5K** plots*
*Average Turns **(**Left), Conditional Knock Probability (Middle), and Action Count (Right) against Random Agent (Top), SGRAgent (Bottom)*

B4.4: Additional Top Layer Summary

The following are conclusions that can be drawn from applying an additional Top Layer:

- **dqn_random** and **dqn_copy** both exhibit similar steady state results
- the Pseudo-Identity Matrix does not provide much stability in training
- The Conditional Knock Probability for all agents tended towards 0%, with majority of the agents' average number of turns increasing as training progressed (Fig. B22, B24, B26, B28 & B30, middle)
- The count of **gin** actions increase while the count of **knock** actions decrease during training for all agents (Fig. B22, B24, B26, B28 & B30, right)
- An additional Top Layer seems unfeasible to be learned given the current Pre-Initialization

B5: Knock Layer Biases

This subsection will go in depth between various experiments performed using by adding a knock layer bias. The Knock Layer Bias is defined by the following:

$$y = softmax(\boldsymbol{W}x + \boldsymbol{b})$$

$$where\ y\ = Agent\ Output\ Layer, \qquad x\ =\ Pre-Initialized\ Output\ Layer$$

$$\boldsymbol{W} = \text{Identity Matrix}, \qquad b_i = \begin{cases} b\ +\ b_{knock}, & if\ action\ =\ knock \\ b, & otherwise \end{cases}, \qquad i \in [1,110]$$

The Knock Layer Bias was introduced as **all** agents tend to 'lose' their ability to perform the **knock** action. The following are the agents that will be used in the next set of experiments.

Table 6: Knock Layer Bias Agent Names

agent name	**Knock Bias, b_{knock}**
dqn_knock_pt002	0.002
dqn_knock_pt02	0.02
dqn_knock_pt2	0.2
dqn_knock_2	2

Note that the Pre-Initialized Output Layer x has already been normalized from the final Softmax activation to achieve Probabilities for each action. Given a total of 52 **knock** actions available, The knock bias values were centered around increasing the entire **knock** action-group by 50% ($b_{knock} = 0.02$) and incremented in both directions by a factor of 10.

B5.1: dqn_knock_pt002

The final micro average rewards against the random agent, self-play, and SGRAgent for **dqn_knock_pt002** are approximately 0.24, -0.05 and -0.08 respectively (Fig. B34).

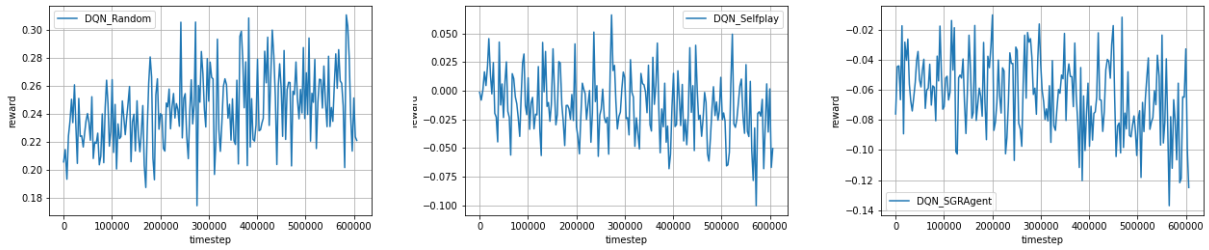

*Figure B34: Micro Average Reward plots for **dqn_knock_pt002** against Random (Left), Self-play (Middle), and SGRAgent (Right)*

The following are non-default hyperparameters used to train **dqn_ knock_pt002**:

- MLP = [260, **520, 520**, 110**, 110**] = [*input*, **HL1, HL2,** *output*, **TL**] (*italicized* = default)
- **TL =** Knock Layer Bias $b_{knock} = 0.002$, followed by Softmax activation

Figure B35 displays the evaluation results against all three agents. Note that **dqn_knock_pt002** has similar performances in terms of increasing average number of turns and decreasing conditional knock probability, despite the increased knock bias (Fig. B35, middle).

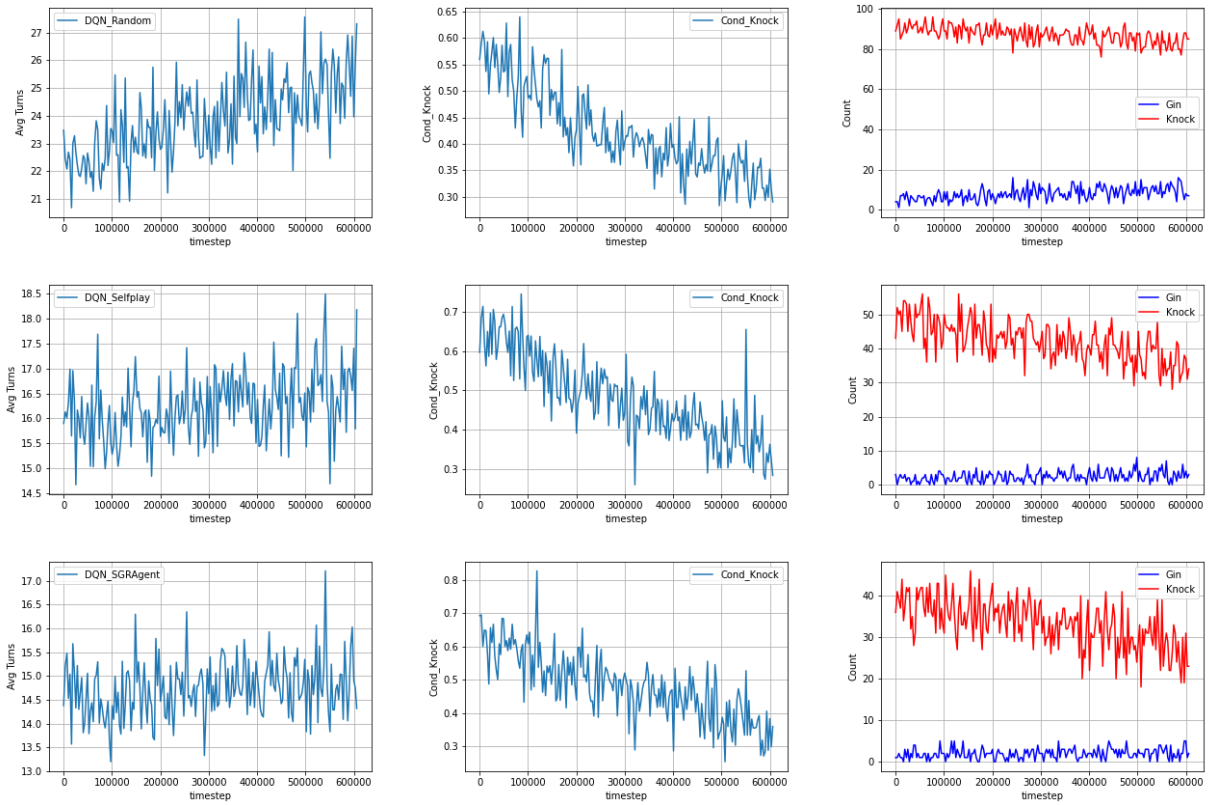

*Figure B35: **dqn_knock_pt002** plots*
*Average Turns **(**Left), Conditional Knock Probability (Middle), and Action Count (Right) against Random Agent (Top), Self-Play (Middle), SGRAgent (Bottom)*

B5.2: dqn_knock_pt02

The final micro average rewards against the random agent, self-play, and SGRAgent for **dqn_knock_pt02** are approximately 0.26, -0.02 and -0.1 respectively (Fig. B36).

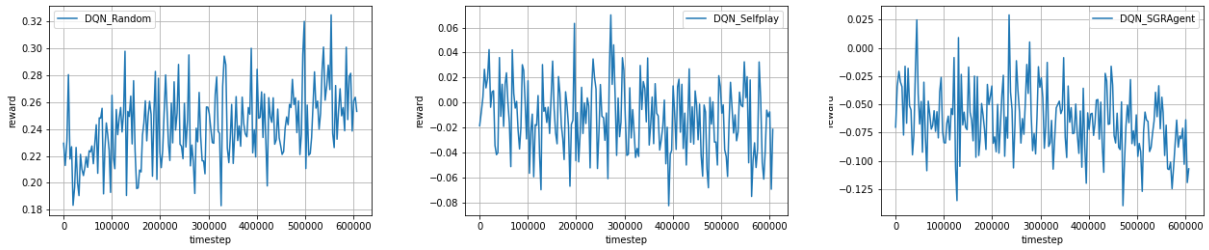

*Figure B36: Micro Average Reward plots for **dqn_knock_pt02***
against Random (Left), Self-play (Middle), and SGRAgent (Right)

The following are non-default hyperparameters used to train **dqn_ knock_pt02**:

- MLP = [260, **520, 520**, 110**, 110**] = [*input*, **HL1, HL2,** *output*, **TL**] (*italicized* = default)
- **TL =** Knock Layer Bias $b_{knock} = 0.02$, followed by Softmax activation

Figure B37 displays the evaluation results against all three agents. Note that **dqn_knock_pt02** has similar performances in terms of increasing average number of turns and decreasing conditional knock probability, despite the increased knock bias (Fig. B37, middle).

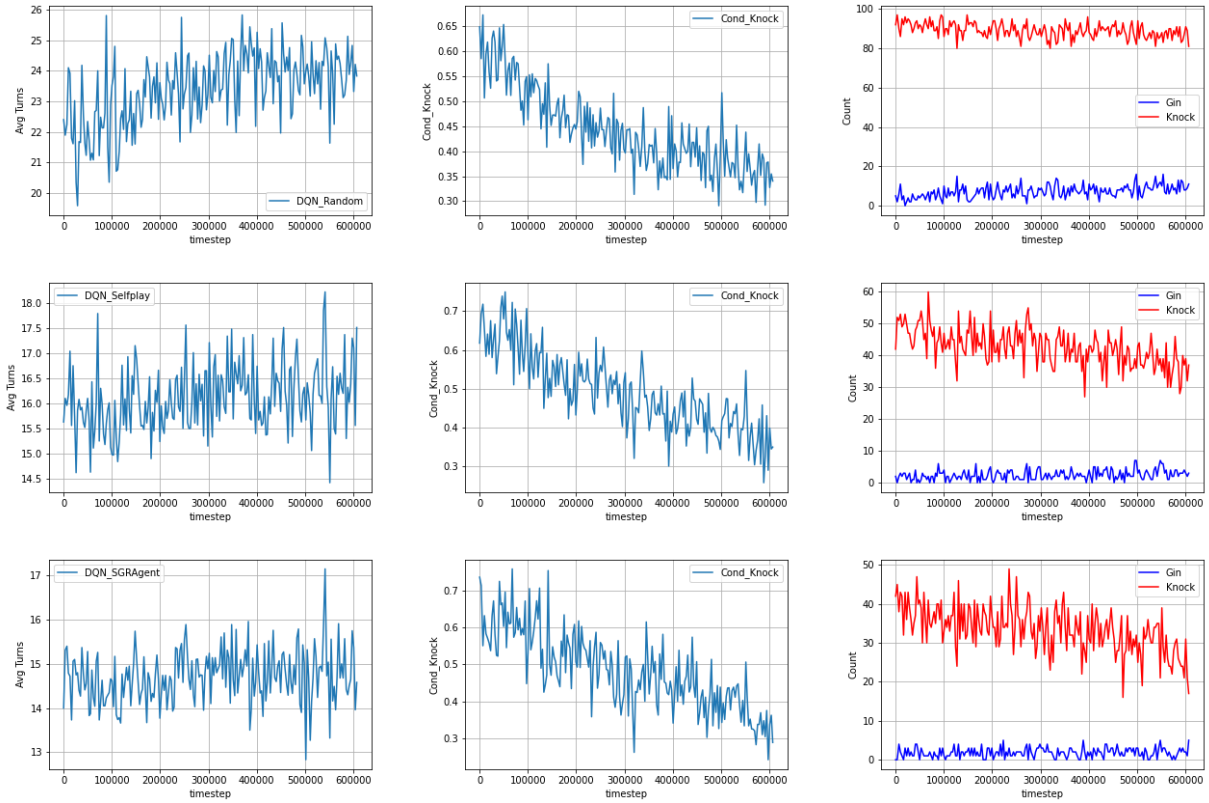

*Figure B37: **dqn_knock_pt02** plots*
*Average Turns **(**Left), Conditional Knock Probability (Middle), and Action Count (Right)*
against Random Agent (Top), Self-Play (Middle), SGRAgent (Bottom)

B5.3: dqn_knock_pt2

The final micro average rewards against the random agent, self-play, and SGRAgent for **dqn_knock_pt2** are approximately 0.27, -0.02 and -0.1 respectively (Fig. B38).

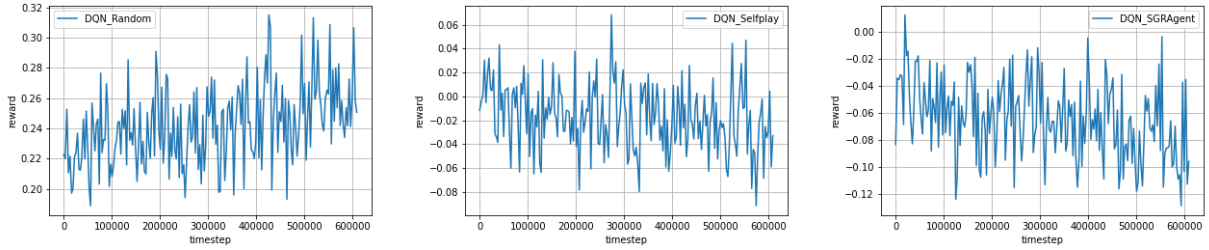

*Figure B38: Micro Average Reward plots for **dqn_knock_pt2***
against Random (Left), Self-play (Middle), and SGRAgent (Right)

The following are non-default hyperparameters used to train **dqn_ knock_pt2**:

- MLP = [260, **520, 520**, 110**, 110**] = [*input*, **HL1, HL2,** *output*, **TL**] (*italicized* = default)
- **TL =** Knock Layer Bias $b_{knock} = 0.2$, followed by Softmax activation

Figure B39 displays the evaluation results against all three agents. Note that **dqn_knock_pt2** has similar performances in terms of increasing average number of turns and decreasing conditional knock probability, despite the increased knock bias (Fig. B39, middle).

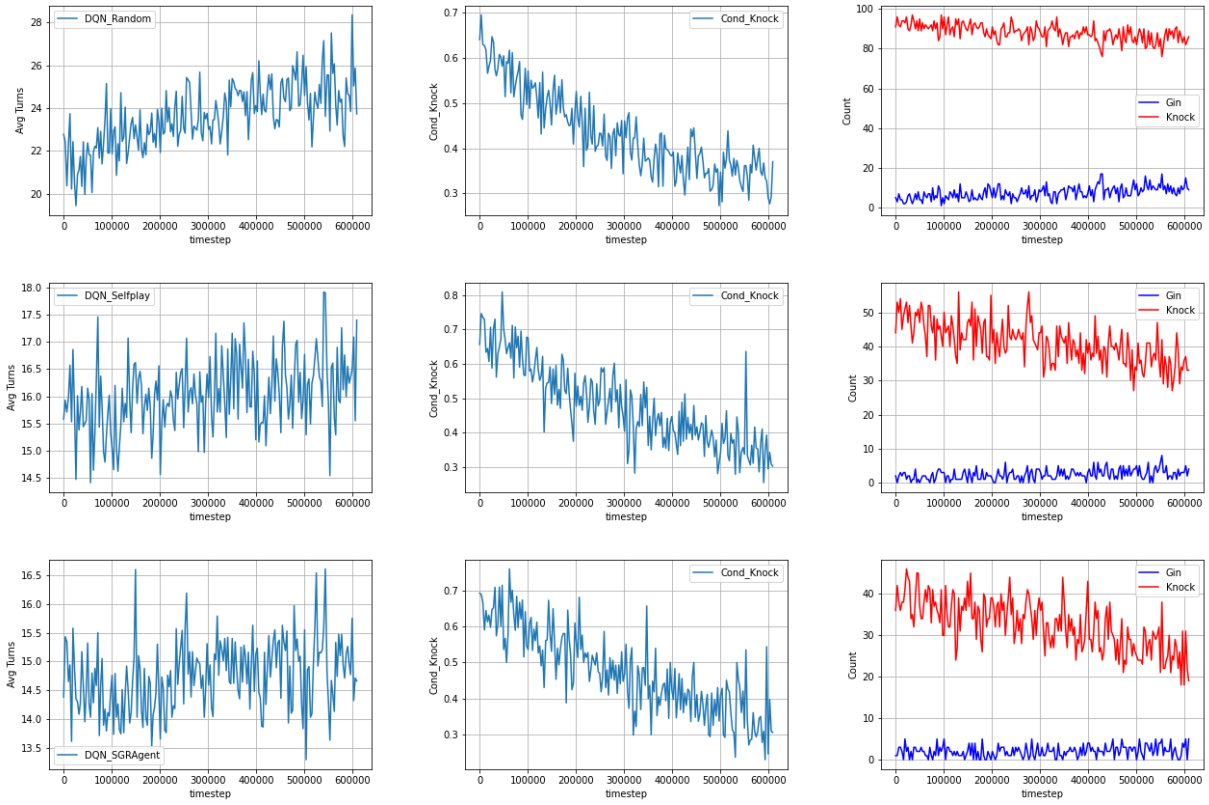

*Figure B39: **dqn_knock_pt2** plots*
*Average Turns **(**Left), Conditional Knock Probability (Middle), and Action Count (Right)*
against Random Agent (Top), Self-Play (Middle), SGRAgent (Bottom)

B5.4: dqn_knock_2

The final micro average rewards against the random agent, self-play, and SGRAgent for **dqn_knock_2** are approximately 0.21, -0.025 and -0.06 respectively (Fig. B40).

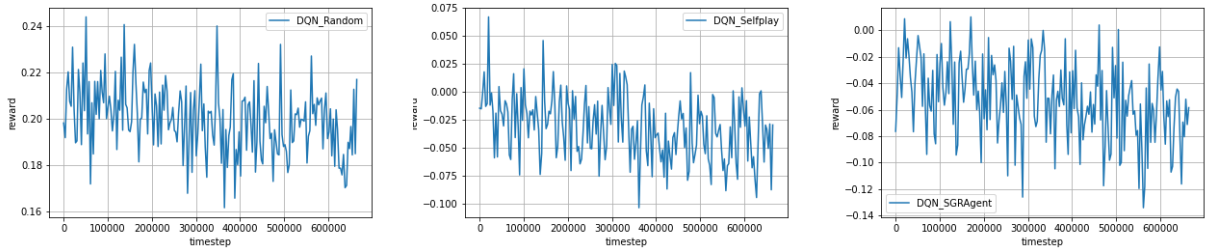

*Figure B40: Micro Average Reward plots for **dqn_knock_2***
against Random (Left), Self-play (Middle), and SGRAgent (Right)

The following are non-default hyperparameters used to train **dqn_ knock_2**:

- MLP = [260, **520, 520**, 110**, 110**] = [*input*, **HL1, HL2,** *output*, **TL**] (*italicized* = default)
- **TL =** Knock Layer Bias $b_{knock} = 2$, followed by Softmax activation

Figure B41 displays the evaluation results against all three agents. Note that **dqn_knock_2** has similar performances in terms of increasing average number of turns however the agent finally achieves a 100% conditional knock probability, (Fig. B41, middle).

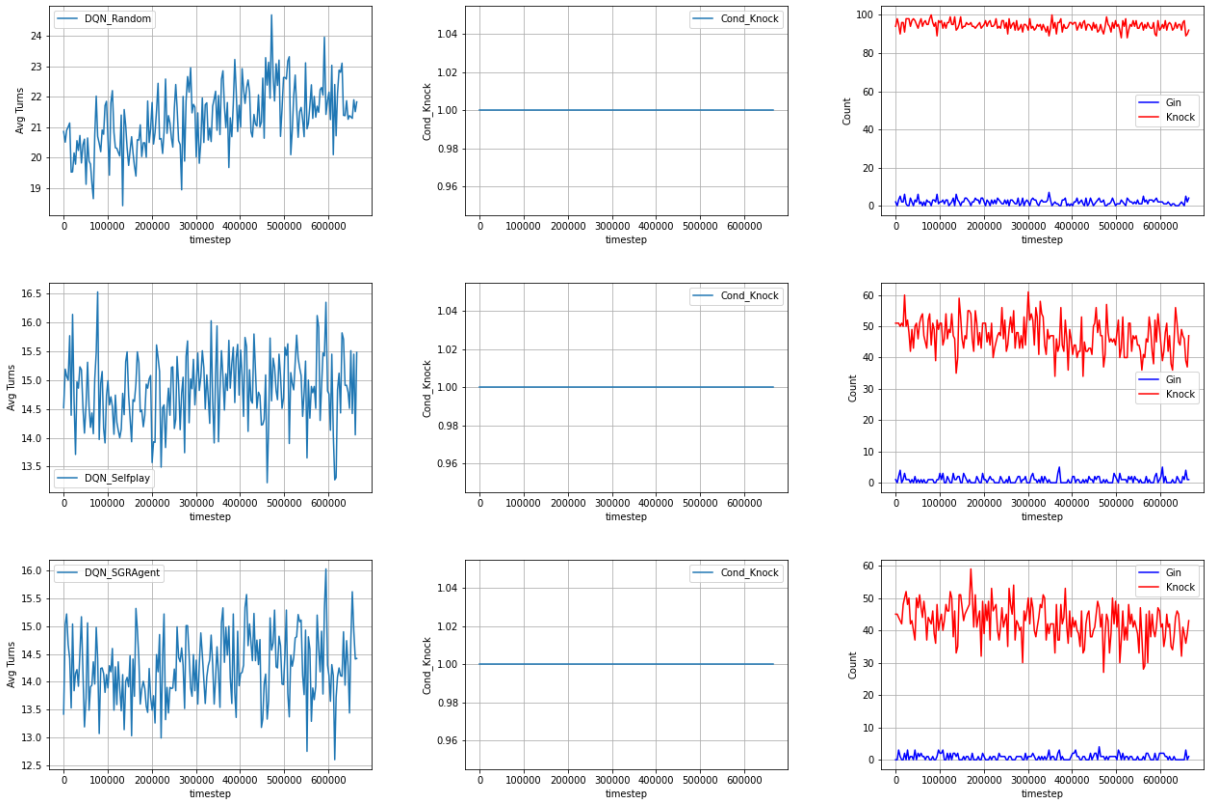

*Figure B41: **dqn_knock_2** plots*
*Average Turns **(**Left), Conditional Knock Probability (Middle), and Action Count (Right)*
against Random Agent (Top), Self-Play (Middle), SGRAgent (Bottom)

B5.5: Knock Layer Bias Summary

The application of the knock layer bias has little to no efforts on the overall performance of the agents. Seen in the reward plots of Figures B34, 36, 38, and 40, all agents had similar performances. Additionally, agents **dqn_knock_pt002, dqn_knock_pt02,** and **dqn_knock_pt2** still exhibit the tendency to 'forget' the **knock** action (Fig. B35, B37 & B39, middle). However, **dqn_knock_2** does not exhibit this as the Conditional Knock Probability for this agent is constantly 1 (Fig. B41, middle). It can be noted that the Knock Layer Bias does slightly incentivise the agent to perform more **knock** actions against SGRAgent with an initial increase of approximately 38 **knocks** with **dqn_knock_pt002** to approximately 45-50 **knocks** with **dqn_knock_2** (Fig. B35 & B41, bottom right), an increase compared to the baseline **dqn_frozen** with approximately 35-40 (Fig. B8, bottom right).

B6: Combination of Experiments

Refer to https://github.com/tancalv2/PretrainGinRummy/tree/main/plots/dqn for combinations of experiments conducted. Majority of the results exhibit the same patterns all agents experience:

- Increasing Number of Turns
- Decreasing Conditional Knock Probability
- Decreasing **knock** actions, Increasing **gin** actions

Appendix C: Testing Environment

The following are the results of evaluating each agent from the *Reinforcement Learning Environment* within the *Testing Environment*. Each agent was evaluated for 1000 games against the following (if applicable):

- **Random Agent**
  - Post DQN-Training Agent
- **SGRAgent**
  - Pre DQN-Training Agent
  - Maximum Reward **and** Conditional Knock Probability
    - Against Random
    - Against Self-play
    - Against SGRAgent
  - Post DQN-Training Agent
- **Baseline Agent** (see Appendix A4)
  - Pre DQN-Training Agent
  - Post DQN-Training Agent
- **Pre DQN-Training Agent**
  - Post DQN-Training Agent

Note: **Bold** indicates the opponent agent, and each subpoint indicates an instance of the DQN agent during the DQN training

C1: With vs. Without Pre-Initialization

Table 7: Win rate for Baseline Agents

Opponent vs. Model			**dqn_baseline**	**dqn_1HL**	**dqn_2HL_40K**	**dqn_2HL_80K**
Random vs. Post DQN-Training			-	100%	100%	100%
SGRAgent	Pre DQN-Training		0%	13.8%[1]	28.0%[1]	30.4%[1]
	Maximum Reward	Random	0%	7.4%	16.1%	20.5%
		Self-Play	0%	7.3%	18.1%	19.4%
		SGRAgent	0%	13.7%	22.0%	30.8%[1]
	Maximum Knock Probability	Random	0%	13.5%	28.2%[1]	30.4%
		Self-Play	0%	14.1%[1]	28.2%[1]	27.3%
		SGRAgent	0%	13.8%	28.2%[1]	32.3%[3]
	Post DQN-Training		0%	8.0%	16.7%	19.4%
Baseline vs. Pre DQN-Training			-	32.1%	53.9%[2]	49.2%
Baseline vs. Post DQN-Training			-	15.9%	33.2%	35.0%
Pre vs. Post DQN-Training			-	28.0%	33.2%	35.0%

[1] Pre DQN-Training was outperformed marginally by agents during training which can be attributed to minor improvements within the agent parameters or outlier game samples

[2] **dqn_2HL_40K** Pre DQN-Training outperforming the baseline **all_2HL_40K** is an aberration as both agents have the initialization

[3] **dqn_2HL_80K** Pre DQN-Training was outperformed by a slightly larger margin against SGRAgent – Maximum Knock Probability

C2: Frozen vs. Unfrozen Layers

Table 8: Win rate for Frozen Agents

Opponent vs. Model			**dqn_unfrozen (dqn_2HL_80K)**	**dqn_frozen**
Random vs. Post DQN-Training			100%	100%
SGRAgent	Pre DQN-Training		30.4%	30.4%[1]
	Maximum Reward	Random	20.5%	23.8%
		Self-Play	19.4%	26.1%
		SGRAgent	30.8%	24.2%
	Maximum Knock Probability	Random	30.4%	31.2%[1]
		Self-Play	27.3%	29.0%
		SGRAgent	32.3%	30.4%
	Post DQN-Training		19.4%[2]	23.1%[2]
Baseline vs. Pre DQN-Training			49.2%	49.2%
Baseline vs. Post DQN-Training			35.0%[2]	40.8%[2]
Pre vs. Post DQN-Training			35.0%[2]	40.8%[2]

[1] Pre DQN-Training was outperformed marginally by agents during training which can be attributed to minor improvements within the agent parameters or outlier game samples

[2] **dqn_frozen** outperforms **dqn_unfrozen** Post DQN-Training

C3: Various Reward Structures

Table 9: Win rate for Various Reward Agents – Scaled Knock Rewards

Opponent vs. Model			**dqn_frozen**	**dqn_2xKnock**	**dqn_5xKnock**	**dqn_50xKnock**
Random vs. Post DQN-Training			100%	100%	100%	100%
SGRAgent	Pre DQN-Training		30.4%	30.4%[1]	30.4%[1]	30.4%[1]
	Maximum Reward	Random	23.8%	23.7%	27.9%	28.9%
		Self-Play	26.1%	26.7%	25.0%	28.0%
		SGRAgent	24.2%	30.3%	24.7%	27.2%
	Maximum Knock Probability	Random	31.2%	31.2%[1]	31.2%[1]	31.2%[1]
		Self-Play	29.0%	29.0%	29.0%	28.9%
		SGRAgent	30.4%	30.4%	30.3%	28.9%
	Post DQN-Training		23.1%	24.3%	25.7%	11.8%[2]
Baseline vs. Pre DQN-Training			49.2%	49.2%	49.2%	49.2%
Baseline vs. Post DQN-Training			40.8%	40.8%	43.8%	26.0%[2]
Pre vs. Post DQN-Training			40.8%	40.8%	43.8%	26.0%[2]

Table 10: Win rate for Various Reward Agents – Modified Non-Knock Rewards

Opponent vs. Model			**dqn_50xKnock**	**dqn_50xKnock_pos**	**dqn_50xKnock_only**
Random vs. Post DQN-Training			100%	100%	100%
SGRAgent	Pre DQN-Training		30.4%[1]	30.4%[1]	30.4%
	Maximum Reward	Random	28.9%	28.9%	28.9%
		Self-Play	28.0%	27.1%	25.7%
		SGRAgent	27.2%	31.2%	31.2%
	Maximum Knock Probability	Random	31.2%[1]	31.2%[1]	31.2%[1]
		Self-Play	28.9%	28.9%	28.9%
		SGRAgent	28.9%	28.9%	28.9%
	Post DQN-Training		11.8%[2]	11.3%[2]	13.1%[2]
Baseline vs. Pre DQN-Training			49.2%	49.2%	49.2%
Baseline vs. Post DQN-Training			26.0%[2]	24.1%[2]	27.8%[2]
Pre vs. Post DQN-Training			26.0%[2]	24.1%[2]	27.8%[2]

[1] Pre DQN-Training was outperformed marginally by agents during training which can be attributed to minor improvements within the agent parameters or outlier game samples

[2] **dqn_50xKnock_XXX** agent have much poorer performances Post DQN-Training

C4: Additional Top Layer

Table 11: Win rate for Random and Copy Top Layer Agents

Opponent vs. Model			**dqn_frozen**	**dqn_random**	**dqn_copy**
Random vs. Post DQN-Training			100%	99.0%[1]	99.7%[1]
SGRAgent	Pre DQN-Training		30.4%	0%	6.3%
	Maximum Reward	Random	23.8%	0%	0%
		Self-Play	26.1%	-	-
		SGRAgent	24.2%	0%	6.3%
	Maximum Knock Probability	Random	31.2%	0%	6.3%
		Self-Play	29.0%	-	-
		SGRAgent	30.4%	0%	6.3%
	Post DQN-Training		23.1%	0%	0%
Baseline vs. Pre DQN-Training			49.2%	0%	17.6%
Baseline vs. Post DQN-Training			40.8%	0%	0%
Pre vs. Post DQN-Training			40.8%	-	0.1%

Table 12: Win rate Pseudo-Identity Top Layer Agents

Opponent vs. Model			**dqn_pseudo_all**	**dqn_pseudo_10pct**	**dqn_pseudo_5K**
Random vs. Post DQN-Training			100%	97.6%[1]	86.5%[1]
SGRAgent	Pre DQN-Training		0%	0%	0%
	Maximum Reward	Random	0%	0%	0%
		Self-Play	-	-	-
		SGRAgent	0.1%	0%	0%
	Maximum Knock Probability	Random	0%	0%	0%
		Self-Play	-	-	-
		SGRAgent	0%	0%	0%
	Post DQN-Training		0%	0%	0%
Baseline vs. Pre DQN-Training			0.1%	0.1%	0%
Baseline vs. Post DQN-Training			0%	0%	0%
Pre vs. Post DQN-Training			1.0%	-	-

[1] Some Post DQN-Trained agents do not beat Random agent 100% of the time

[2] Pseudo-Identity agents have no performance against SGRAgent or **baseline** agents, very similar results to a **dqn_random**

C5: Knock Layer Biases

Table 13: Win rate for Knock Layer Bias Agents

Opponent vs. Model			**dqn_knock_pt002**	**dqn_knock_pt02**	**dqn_knock_pt2**	**dqn_knock_2**
Random vs. Post DQN-Training			100%	100%	100%	100%
SGRAgent	Pre DQN-Training		30.9%	30.9%	31.0%[1]	33.4%[1]
	Maximum Reward	Random	23.8%	28.5%	27.6%	33.9%[1]
		Self-Play	27.3%	27.5%	28.1%	33.6%
		SGRAgent	27.0%	24.7%	31.5%[1]	32.2%
	Maximum Knock Probability	Random	28.0%	30.6%	31.0%	33.4%[3]
		Self-Play	29.3%	30.1%	29.4%	33.4%[3]
		SGRAgent	27.4%	29.3%	30.5%	33.4%[3]
	Post DQN-Training		23.3%	27.8%	30.1%	21.9%
Baseline vs. Pre DQN-Training			49.6%	49.4%	50.8%[2]	52.8%[2]
Baseline vs. Post DQN-Training			44.7%	42.7%	46.9%	39.1%
Pre vs. Post DQN-Training			45.0%	42.9%	44.3%	34.2%

[1] Pre DQN-Training was outperformed marginally by agents during training which can be attributed to minor improvements within the agent parameters or outlier game samples

[2] Pre DQN-Training is better than baseline since the agent is more likely to knock compared to baseline agent

[3] Win rates have no meaning as agent has 100% Conditional Knock Probability

[This page was left intentionally blank]