Pretraining to Speed Up Q-Learning for Card Games with Large Feature Spaces

Interim Report – Winter 2021

Supervisor: Michael Guerzhoy
Calvin Kwei Hoi Tan
1003332556

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, thus requiring research in existing or innovative techniques to potentially address these issues.

A technique that found success in accelerating training and 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.

The goal in this research is to explore the principles of pretraining RL agents using a Deep Learning approach 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 pretrain a DNN through supervised learning using thousands of self-play examples generated using 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 with self-play. With the combination of the pretrained DNN initialized onto the DQN, the trained agent will be evaluated against the original baseline agent to test the validity of this pretraining approach.

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:

$$Q_{t+1}(s_t,a_t) \leftarrow Q_t(s_t,a_t) + \alpha_t[r_{t+1} + \gamma \max_{a'} Q_t(s_{t+1},a') - Q_t(s_t,a_t)]$$

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, is that the Gin Rummy environment is still extremely large 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 applicable to a supervised approach. With the application of pretraining to the function approximator, it should help the DQN find a more optimal policy compared to pure randomized initialization.

A paper by Cruz et al. parallels with the current research topic [3]. In their paper, their 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 pretraining method as the primary focus is to on feature learning of the environment.

3 Background: Gin Rummy

Gin Rummy is a two-player card game that uses a standard deck of 52 cards. A game starts with each player starts receiving 10 cards, 1 card face up on the *discard* pile and the remaining deck face down. The goal of 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 **K, Q, J** have value 10 and **A** have 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 play 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 additional points, otherwise known as going *gin*. Lastly, if the *knocking* player had a higher *deadwood*, the *knocking* player *undercuts***,** and additional points are awarded to the opposing player. The first player to 100 points wins the game.

3.1 Environment 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 environment. 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'** action (drawn from deck or pickup discard)
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]
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 not an actual state that occurs during a game of Gin Rummy and instead is merely used to develop an intuition of low-level features for Supervised Learning.

Table 2: State-Action Pair Definition

State	**Available Action Group [Action ID]**	
bpbd	'**draw' [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 (**'draw', 'discard',** and **'knock'**)	

3.2 Data Generation

Data is generated by modifying a two player Gin Rummy tournament [8] to capture the state-action pairs. The inputs are the states, encoded as a 260 (52 cards by 5 planes) long 1-D vector, and the outputs are the actions, encoded as a 110 long 1-D vector. A simple Gin Rummy agent (SimpleGinRummyPlayer) was used to generate the data. The policy of the agent is as follows:

1. Pickup card from discard pile if card can be melded with current hand
2. Discard highest card in hand that is not apart of any melds
3. Knock if deadwood is less than or equal to 10

Batches of games (2K, 6K, 8K, etc.) were generated and denoted as *s_{x}k.npy* and *a_{x}k.npy*, where {x} represents the number of games, for states and actions respectively through self-play. For each state-action pair, 2K, 6K, and 8K batches of games were generated for training, validation and testing purposes. Additional 32K and 40K batches of games for apbd-knock state-action pair were generated to supplement the unbalanced nature of the dataset.

4 Experimental Design

The approach to test pretraining on DQNs can be broken down into the following steps: 1) train the pretrained networks through supervised learning, 2) initialize the pretrained network onto the DQN and train the agent, and 3) evaluate the performance of the pretrained DQN agent against the baseline agent used to train the supervised networks.

5 Pretrained Networks through Supervised Learning

Multiple models for each state-action pair were trained, each tasked in learning the specified action. Some lower-level features like melds were examined to gain better insight.

5.1 Before Pickup, Before Discard (bpbd)

5.1.1 Action Group: draw

The goal of this model is to learn the **'draw'** policy of the SimpleGinRummyPlayer agent, 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).

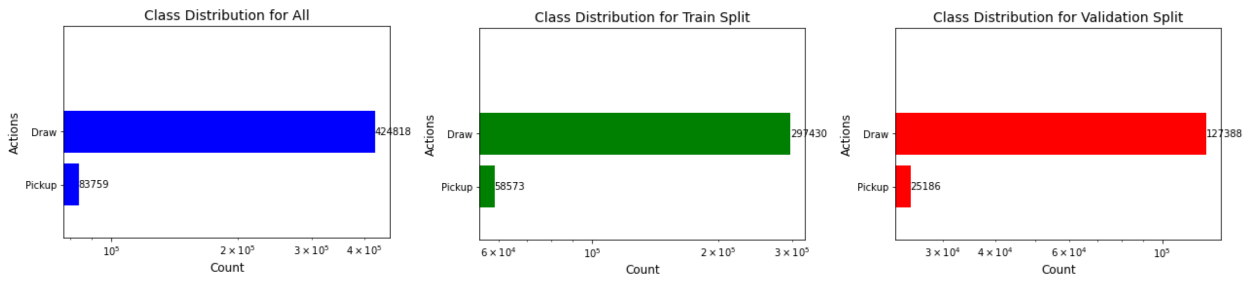


Figure 1: Data Distribution for bpbd_draw – all (left), train (middle), validation (right). Actions (y-axis), Count (x-axis, log-scaled).

The model, **bpbd_draw**, 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)

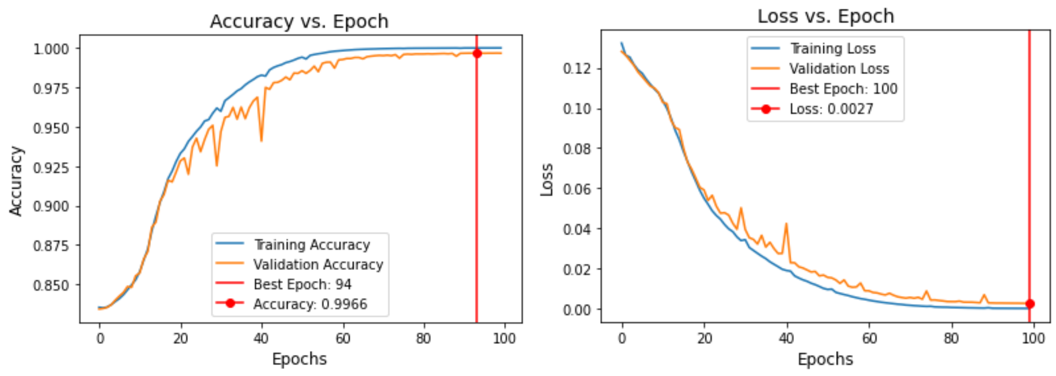


*Figure 2: Training plots for **bpbd_draw** – Accuracy (left), Mean Loss (right)*

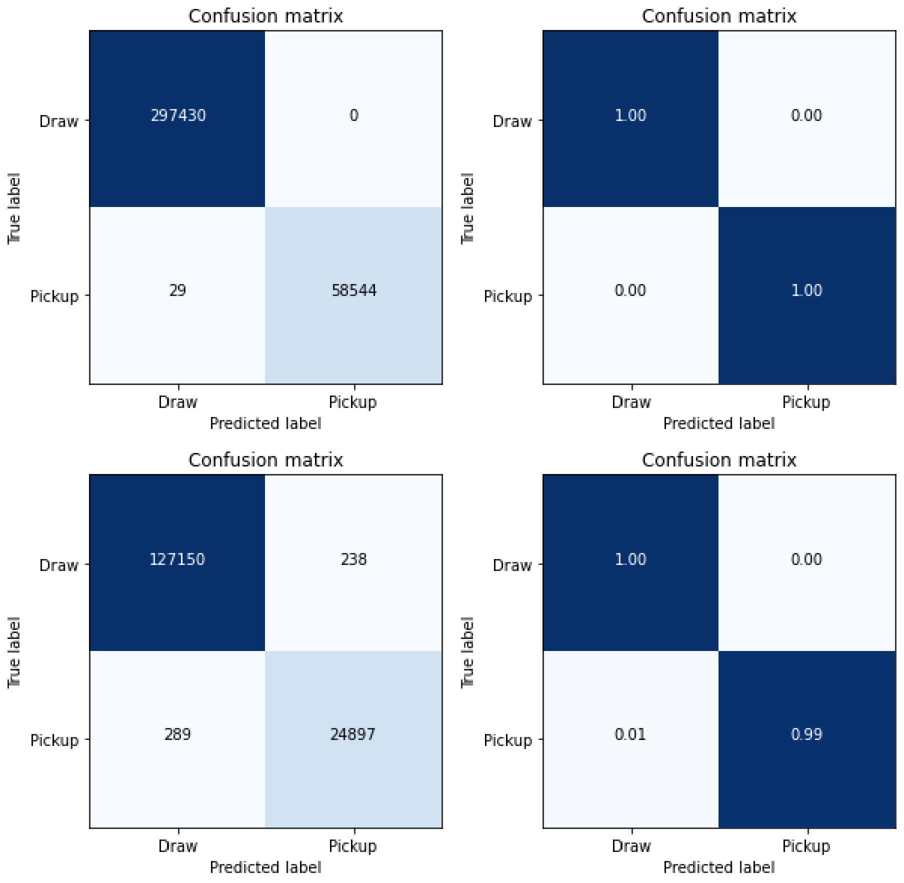


*Figure 3: Confusion Matrices for **bpbd_draw** – Train (top), Validation (bottom), Normalized Values (right). True Label (y-axis), Predicted Label (x-axis). Darker tile = larger number.*

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

5.2 After Pickup, Before Discard (apbd)

5.2.1 Action Group: discard

The goal of this model is to learn the **'discard'** policy of the SimpleGinRummyPlayer agent, 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.

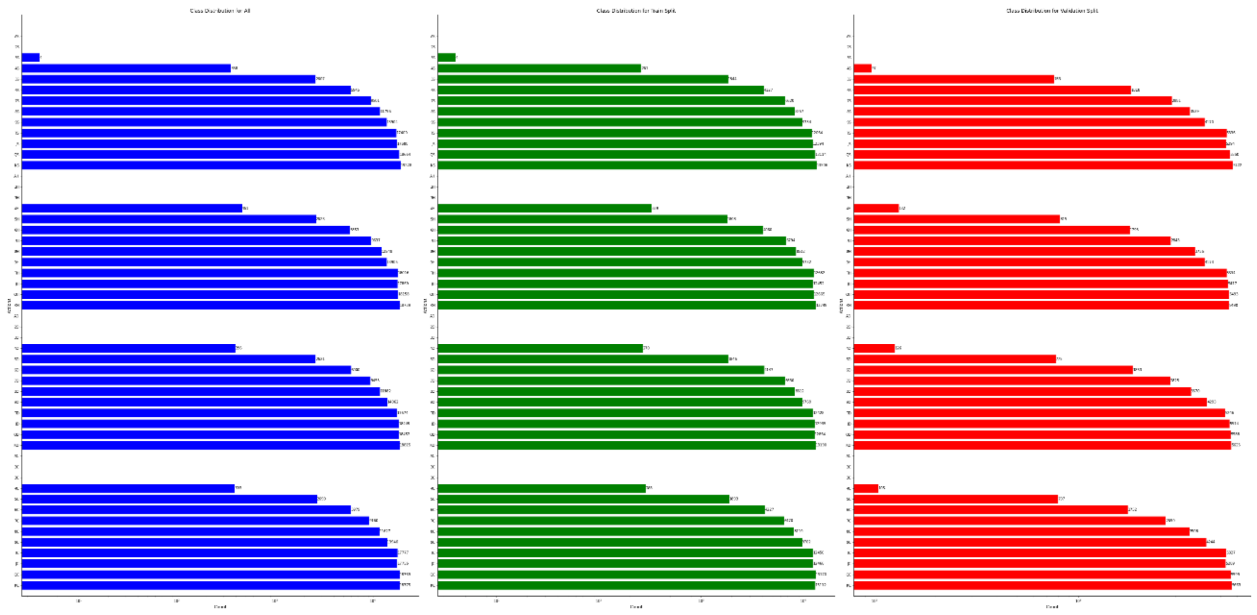
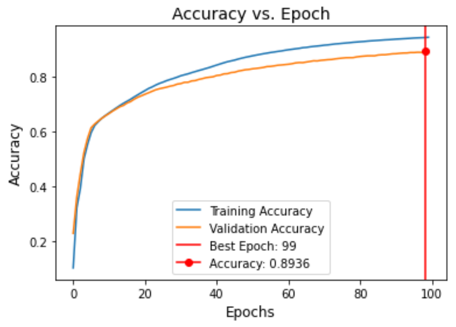


*Figure 4: Data Distribution for **apbd_discard** – all (left), train (middle), validation (right). Actions (y-axis), Count (x-axis, log-scaled).*

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)



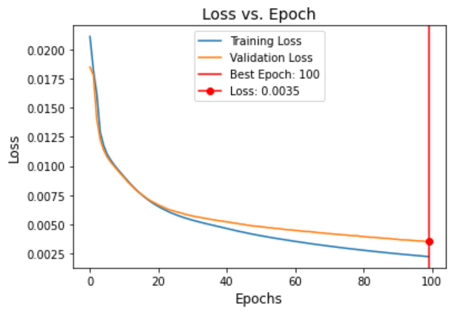


Figure 5: Training plots for ***apbd_discard*** *– Accuracy (left), Mean Loss (right)*

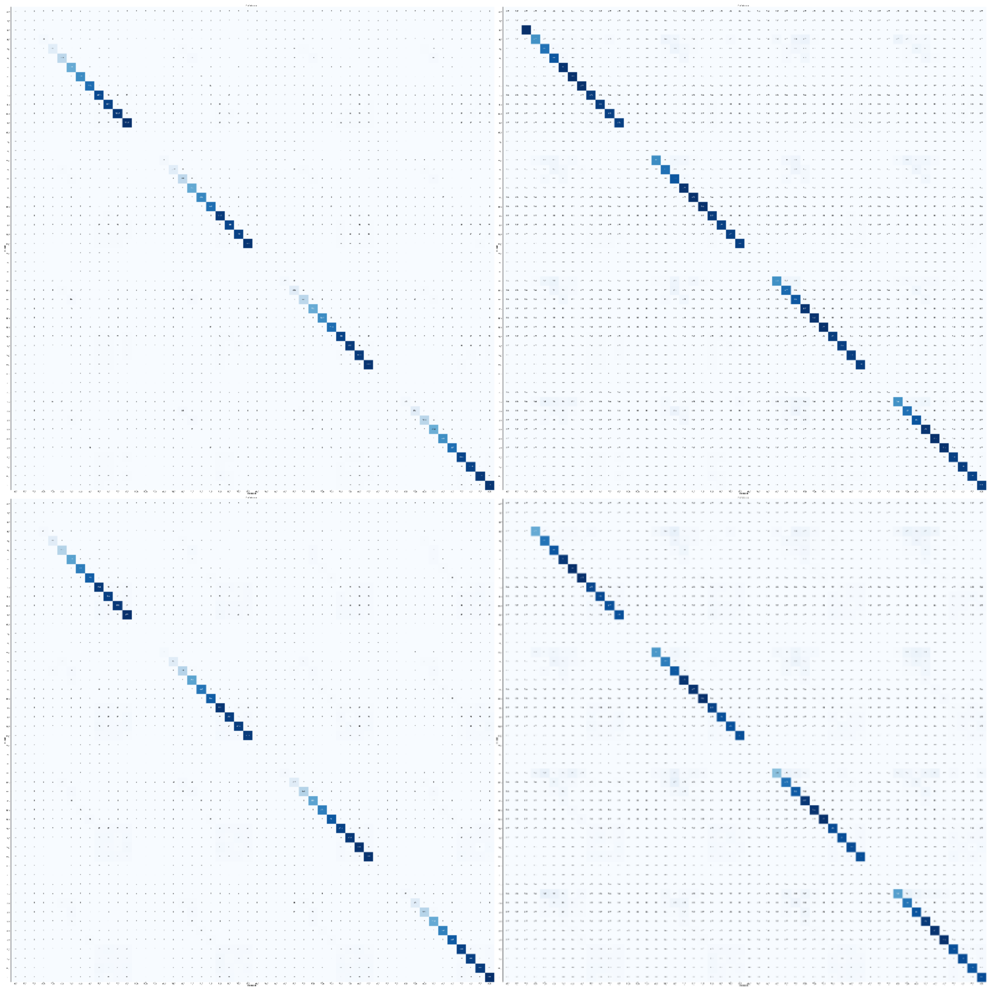


Figure 6: Confusion Matrices for ***apbd_discard*** *– Train (top), Validation (bottom), Normalized Values (right). True Label (y-axis), Predicted Label (x-axis). Darker tile = larger number.*

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.

5.2.2 Action Group: knock

The goal of this model is to learn the **'knock'** policy of the SimpleGinRummyPlayer agent, 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**.

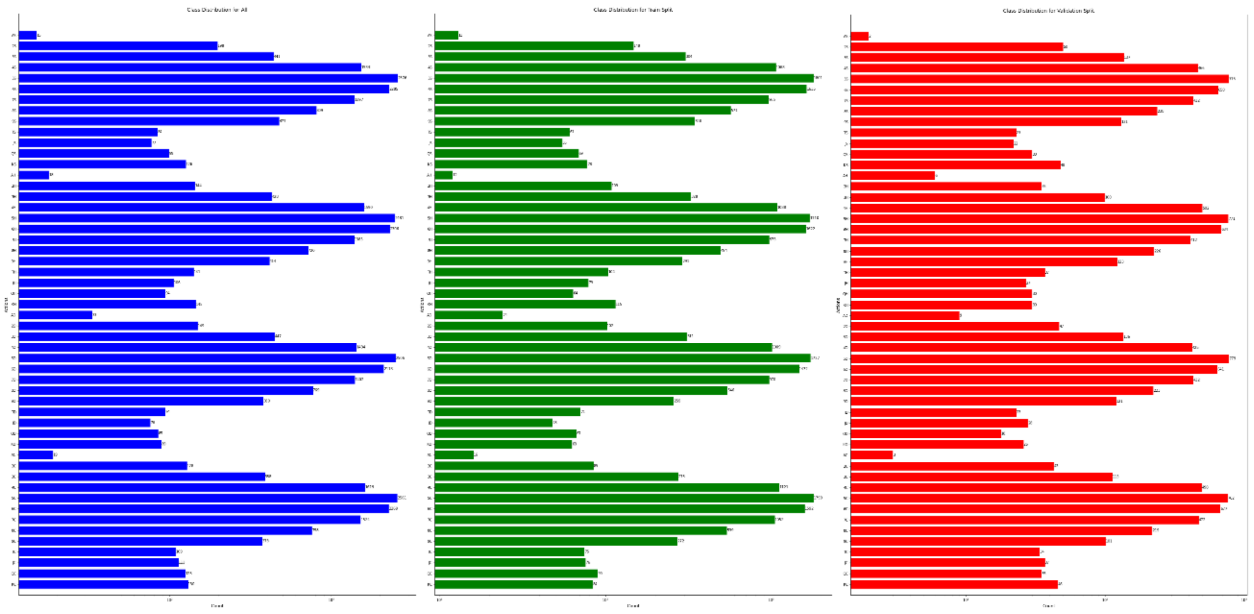


*Figure 7: Data Distribution for **apbd_knock** – all (left), train (middle), validation (right). Actions (y-axis), Count (x-axis, log-scaled).*

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)

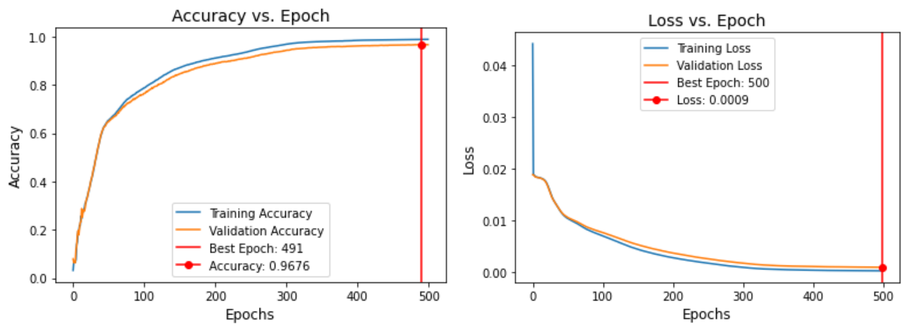


Figure 8: Training plots for ***apbd_knock*** *– Accuracy (left), Mean Loss (right)*

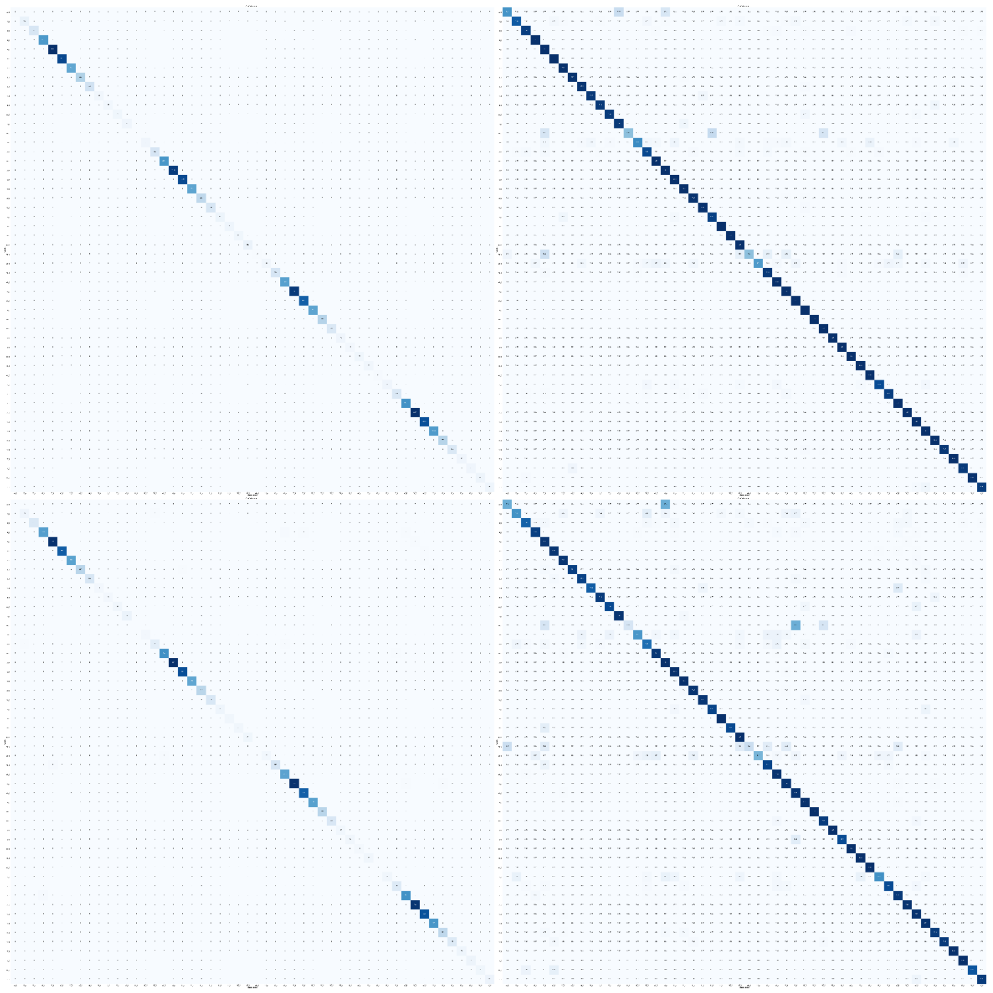


Figure 9: Confusion Matrices for ***apbd_knock*** *– Train (top), Validation (bottom), Normalized Values (right). True Label (y-axis), Predicted Label (x-axis). Darker tile = larger number.*

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

5.3 After Pickup, After Discard (apad)

5.3.1 Action Group: knock_bin

The goal of this model is to learn the **'knock'** policy of the SimpleGinRummyPlayer agent, 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).

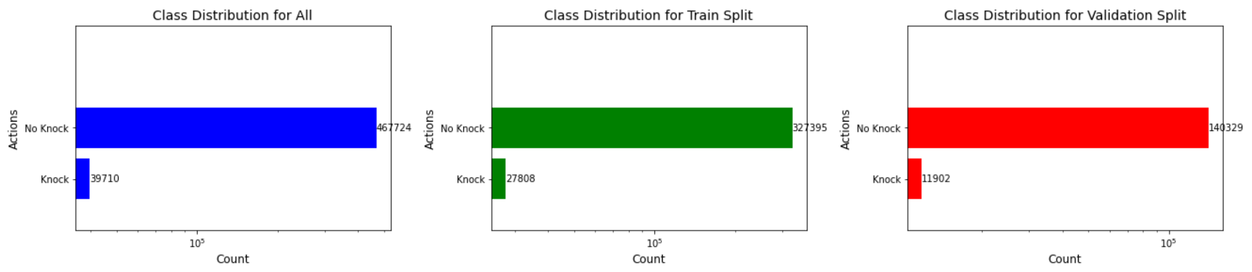


*Figure 10: Data Distribution for **apad_knock** – all (left), train (middle), validation (right). Actions (y-axis), Count (x-axis, log-scaled).*

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)

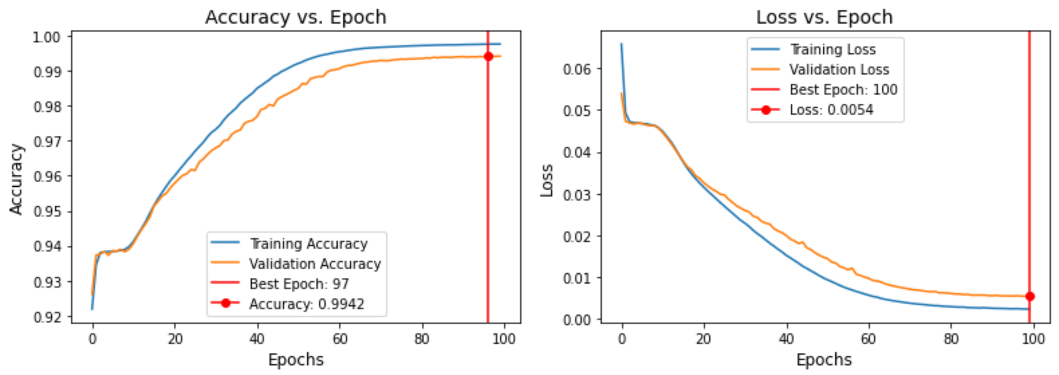


*Figure 11: Training plots for **apad_knock** – Accuracy (left), Mean Loss (right)*

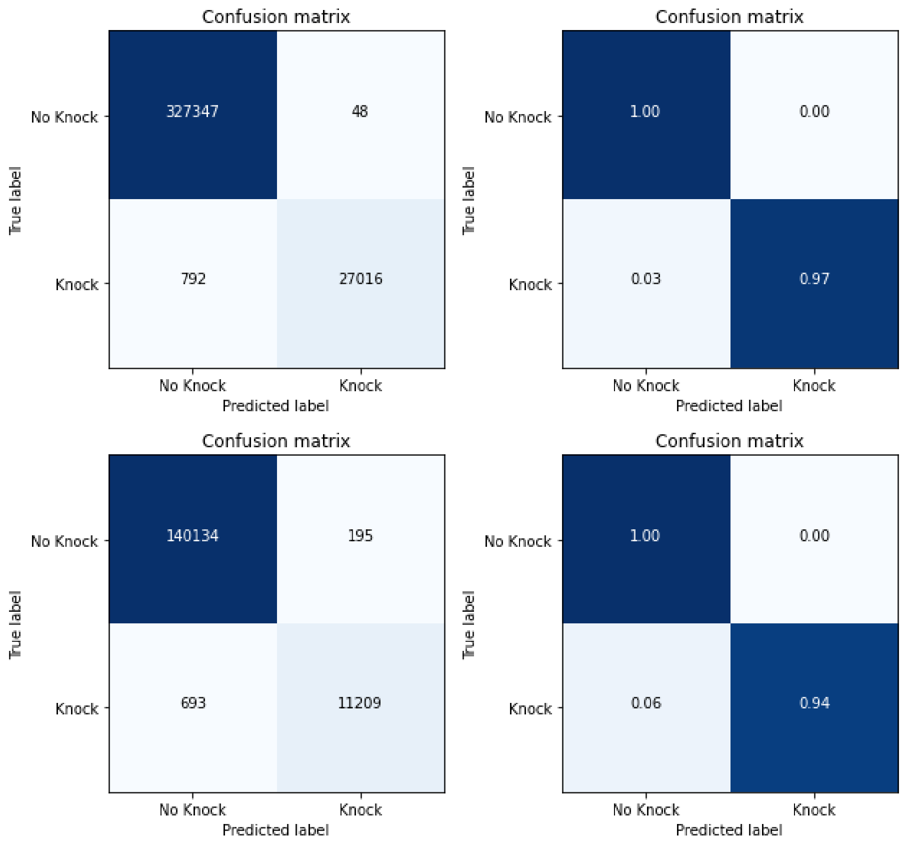


Figure 12: Confusion Matrices for ***apad_knock*** *– Train (top), Validation (bottom), Normalized Values (right). True Label (y-axis), Predicted Label (x-axis). Darker tile = larger number.*

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.

5.4 All States, All Actions (all)

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

- **all:** baseline model using the same hyperparameters as previous models
- **all_2HL_40K:** adapted version of **all_baseline**, with an extra hidden layer and more data
- **all_2HL_80K:** adapted version of **all_baseline**, with an extra hidden layer and even more data

5.4.1 all_1HL

The goal of this model is to learn create a baseline model for all the state-action pairs generated from the SimpleGinRummyPlayer agent. 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 the distribution (half) 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 10 **discards**:1 **knock**.

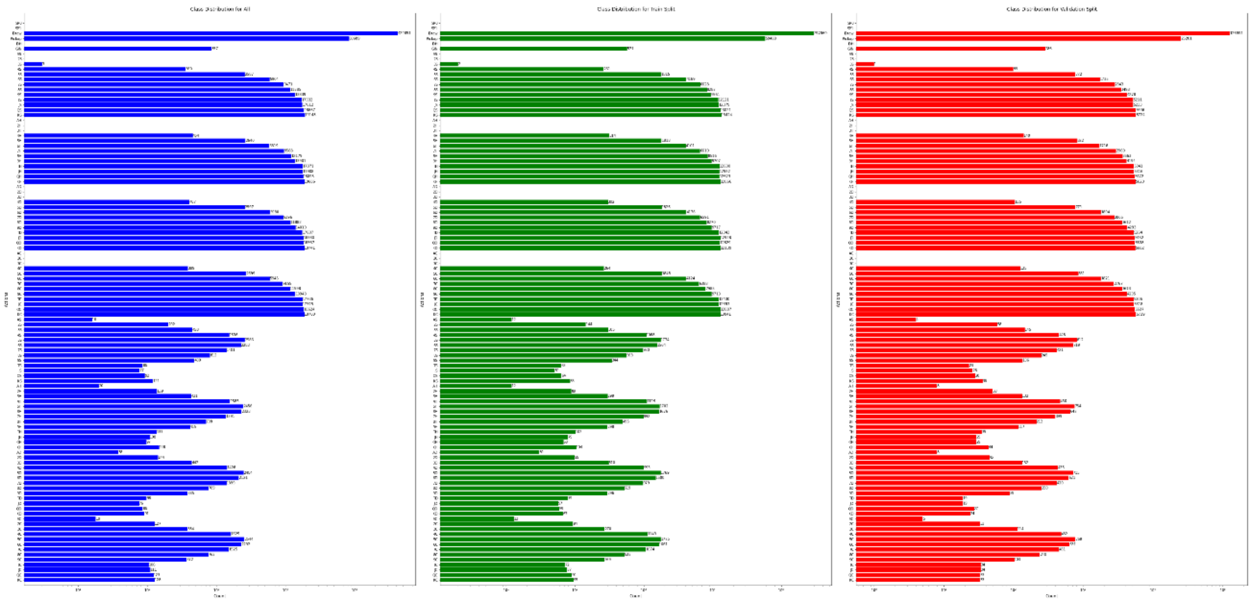


*Figure 13: Data Distribution for **all_1HL** – all (left), train (middle), validation (right). Actions (y-axis), Count (x-axis, log-scaled).*

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.

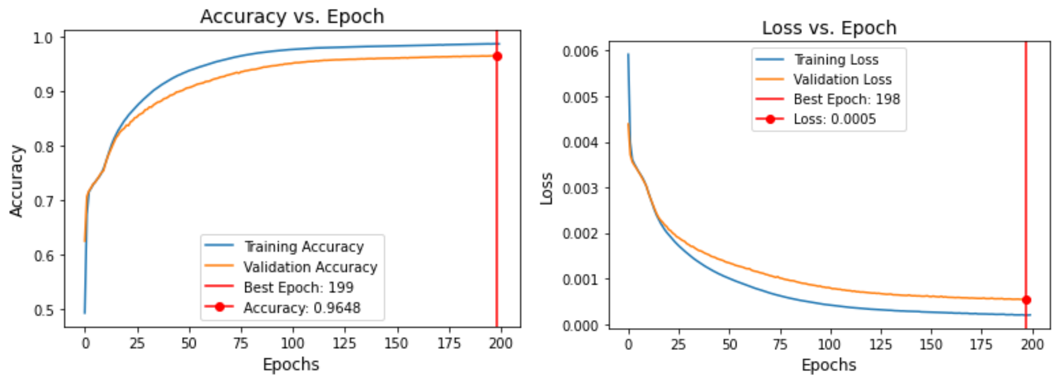


Figure 14: Training plots for ***all_1HL*** *– Accuracy (left), Mean Loss (right)*

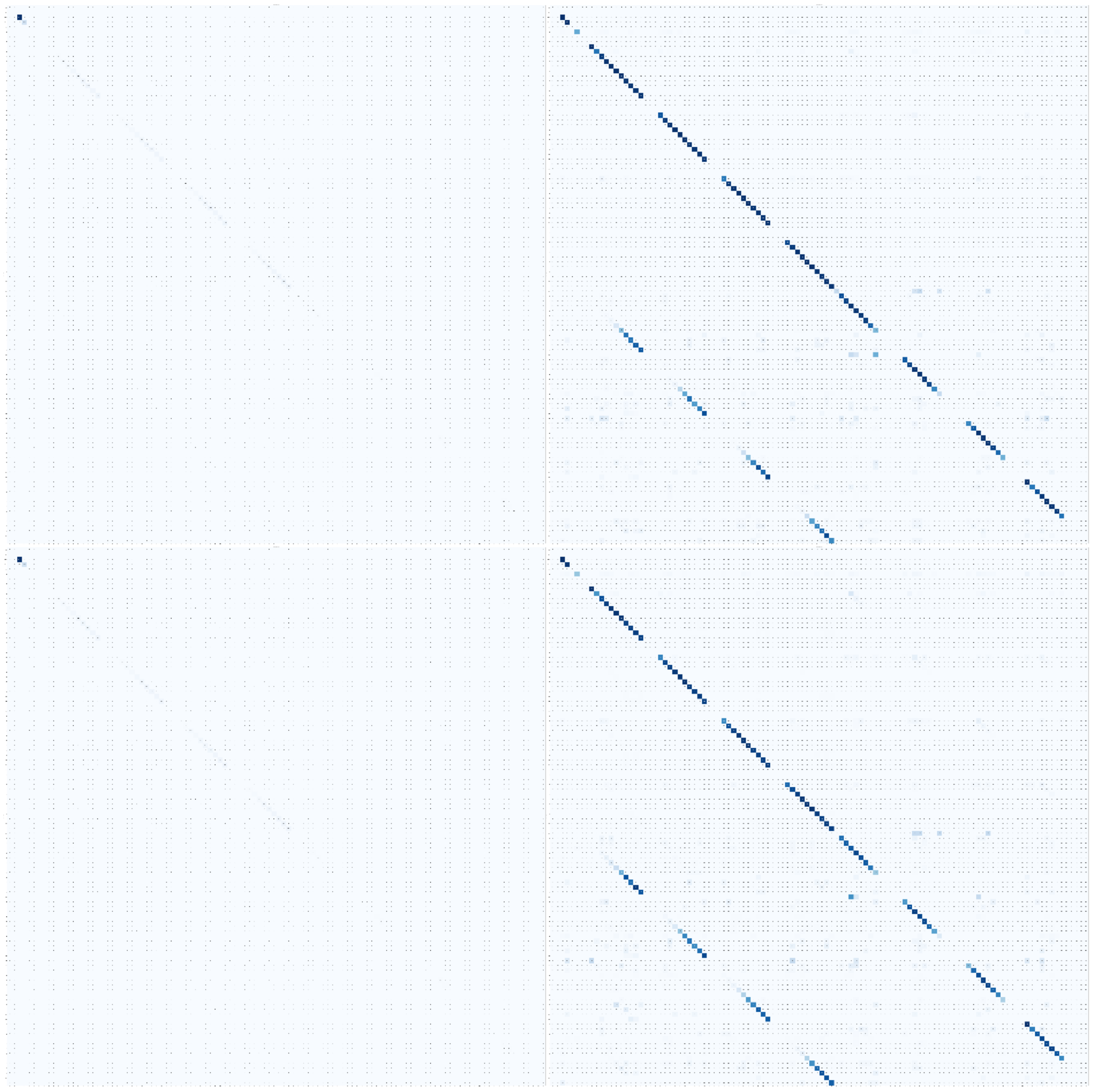


Figure 15: Confusion Matrices for ***all_1HL*** *– Train (top), Validation (bottom), Normalized Values (right). True Label (y-axis), Predicted Label (x-axis). Darker tile = larger number.*

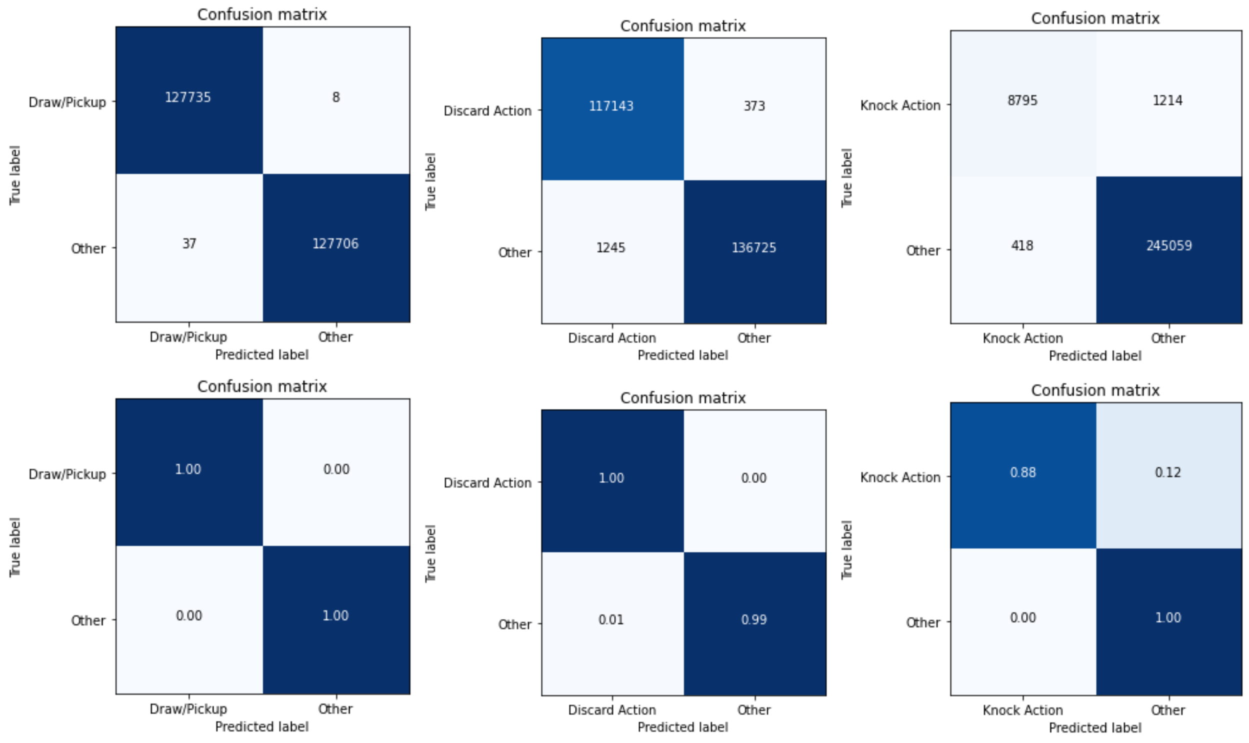


Figure 16: Confusion Matrices for **all_1HL** based on state-action groups – **'draw'** (left), **'discard'** (middle), **'knock'** (right), Normalized Values (bottom). True Label (y-axis), Predicted Label (x-axis). Darker tile = larger number. *2K Games Test Set*

From the results of the training, **all_1HL**, a final validation accuracy of 96.48% was achieved. The confusion matrices from Figure 15 illustrate the following:

1) Majority of the samples are within the **'draw'** action group (Fig. 15, left),
2) All three state-action groups can be identified by 'quadrants' (Fig. 15, right)
   - the upper left section is **'draw'**,
   - the middle section is **'discard'**,
   - the lower right section is **'knock'**
3) **'knock'** actions are being misclassified to **'discard'** action (Fig. 15, right)
   - More specifically, a card of certain suit and rank that was intended to be knocked is being discard

The confusion matrices from Figure 16 also illustrates these issues. When evaluated based on state-action groups, both **'draw'** and **'discard'** have a high degree of accuracy predicting the true labels however only an 88% accuracy for **'knock'** (Fig. 16, right).

5.4.2 all_2HL_40K

Due to the poor performance of **‘knock’** prediction in the **all_1HL** model, an additional 40K games of **‘knock’** data was generated and 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. There are approximately 2 **discards**:1 **knock**.

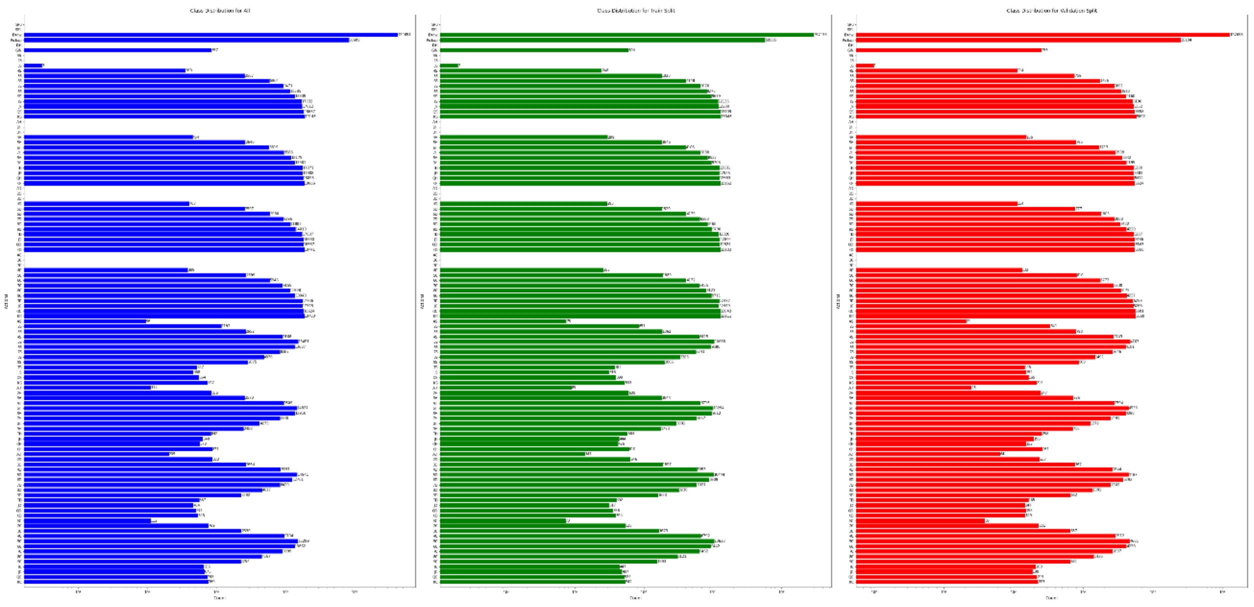


*Figure 17: Data Distribution for **all_2HL_40K** – all (left), train (middle), validation (right). Actions (y-axis), Count (x-axis, log-scaled).*

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

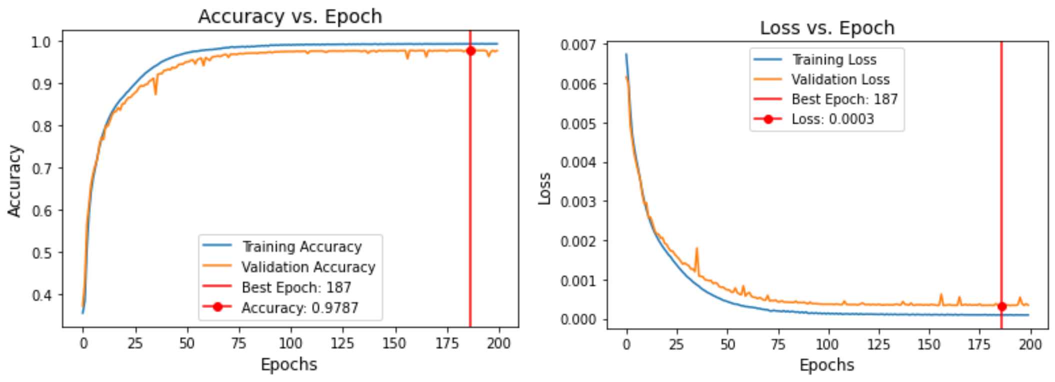


*Figure 18: Training plots for **all_2HL_40K** – Accuracy (left), Mean Loss (right)*

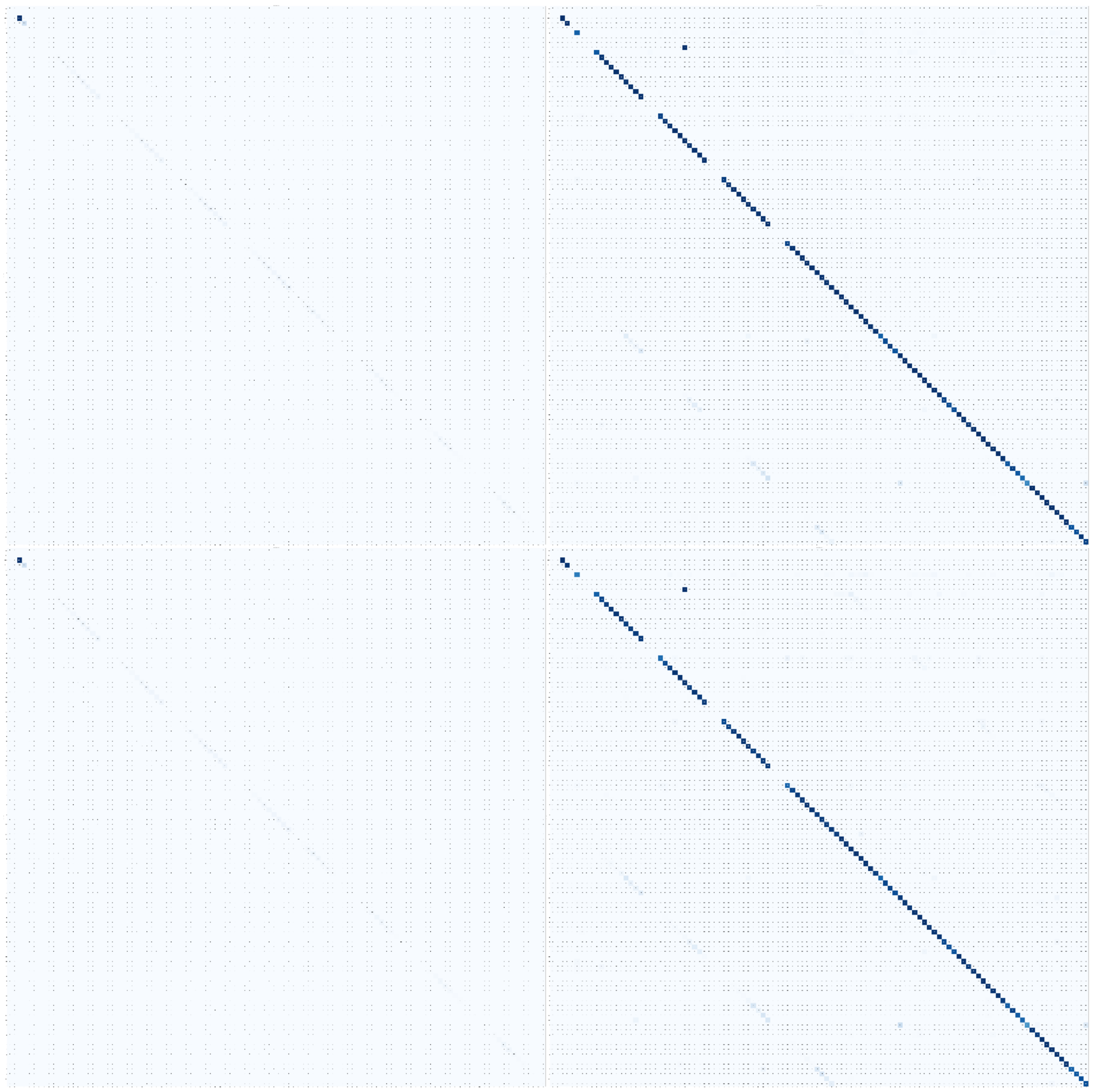


*Figure 19: Confusion Matrices for **all_2HL_40K** – Train (top), Validation (bottom), Normalized Values (right). True Label (y-axis), Predicted Label (x-axis). Darker tile = larger number.*

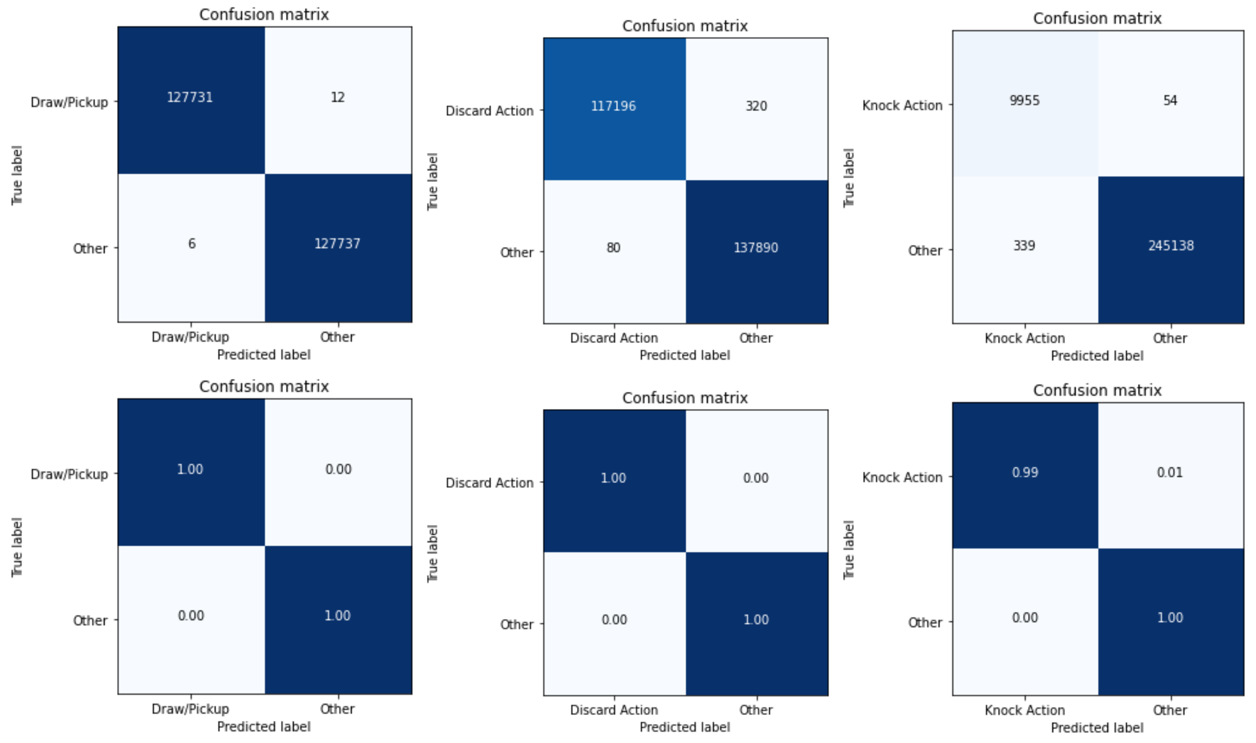


Figure 20: Confusion Matrices for* ***all_2HL_40K*** *based on state-action groups –* ***'draw'*** *(left),* ***'discard'*** *(middle),* ***'knock'*** *(right), Normalized Values (bottom). True Label (y-axis), Predicted Label (x-axis). Darker tile = larger number. *2K Games Test Set*

From the results of the training, **all_2HL_40K**, a final validation accuracy of 97.87% was achieved. The confusion matrices from Figure 19 illustrate the following:

1) **'knock'** actions are not being misclassified as often to **'discard'** action (Fig. 19, right) compared to prior (Fig. 15, right). It can be seen that there still a faint diagonal within the True **'knock'**, Predicted **'discard'** section of the confusion matrix (Fig. 19, bottom right, bottom middle of the confusion matrix)

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

   o 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 20 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. 20, bottom).

5.4.3 all_2HL_80K

An additional 80K games of **‘knock’** data was generated and 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. There are approximately 1 **discard**:1 **knock**.

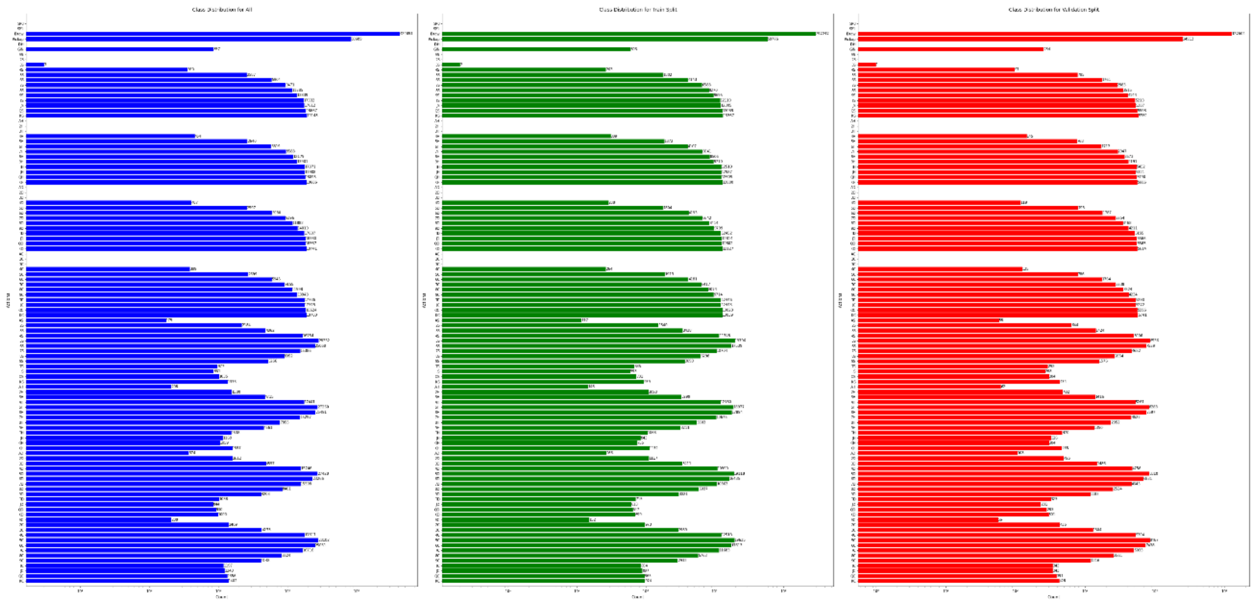


*Figure 21: Data Distribution for **all_2HL_80K** – all (left), train (middle), validation (right). Actions (y-axis), Count (x-axis, log-scaled).*

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

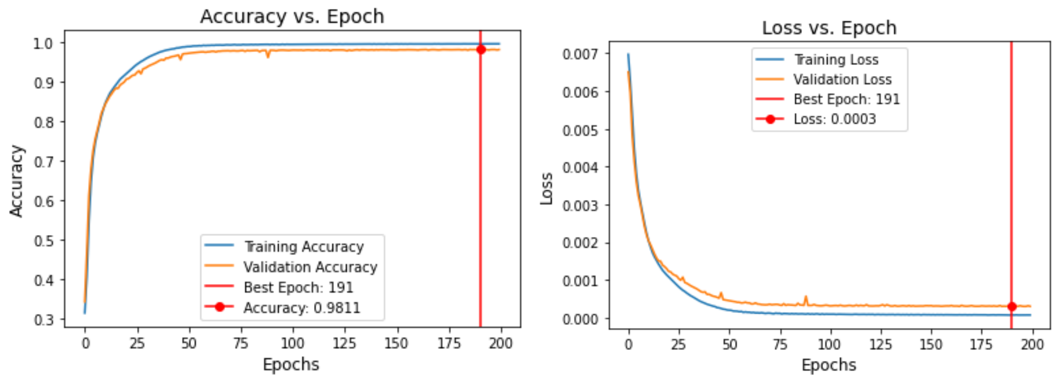


*Figure 22: Training plots for **all_2HL_80K** – Accuracy (left), Mean Loss (right)*

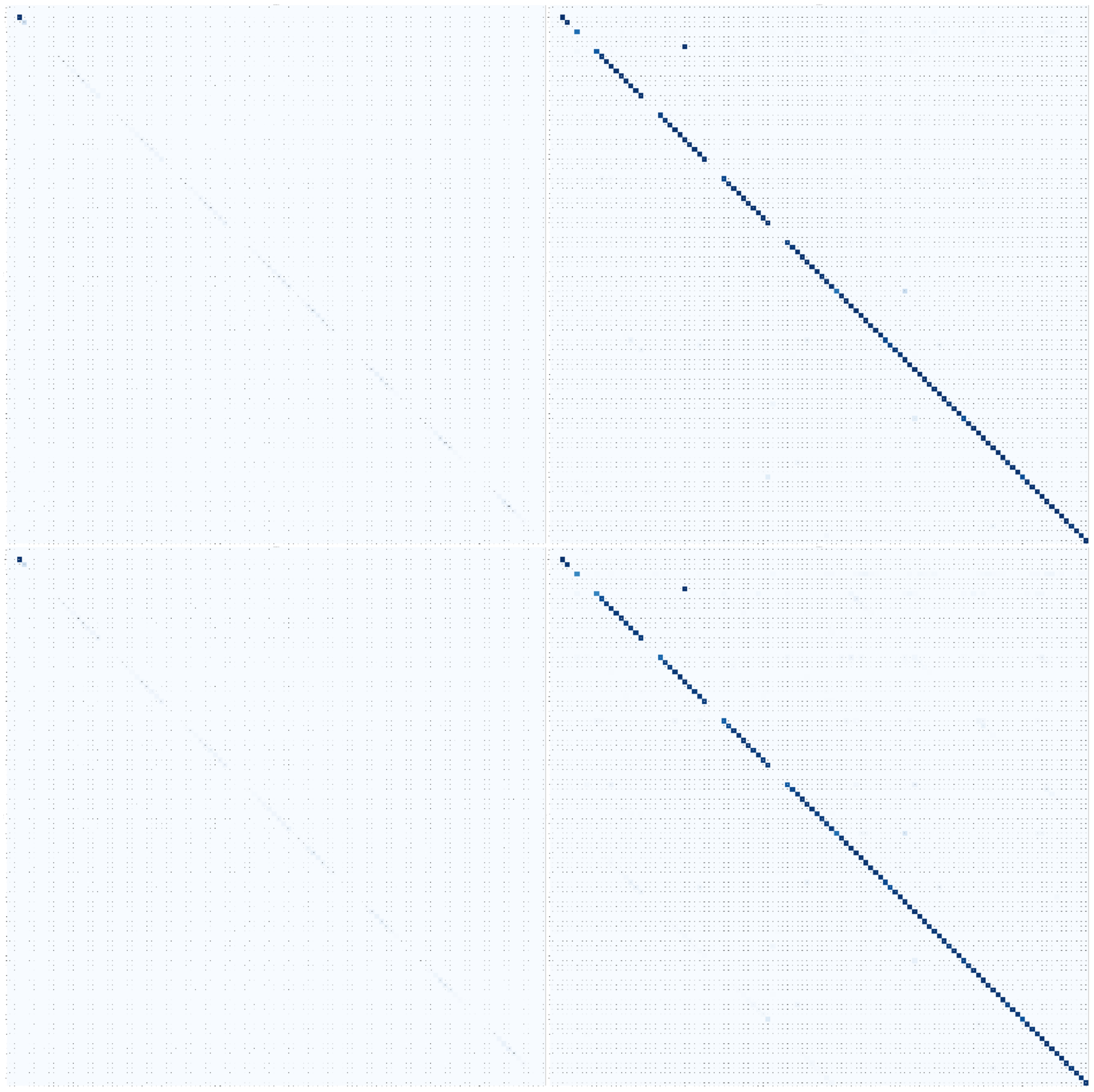


*Figure 23: Confusion Matrices for **all_2HL_80K** – Train (top), Validation (bottom), Normalized Values (right). True Label (y-axis), Predicted Label (x-axis). Darker tile = larger number.*

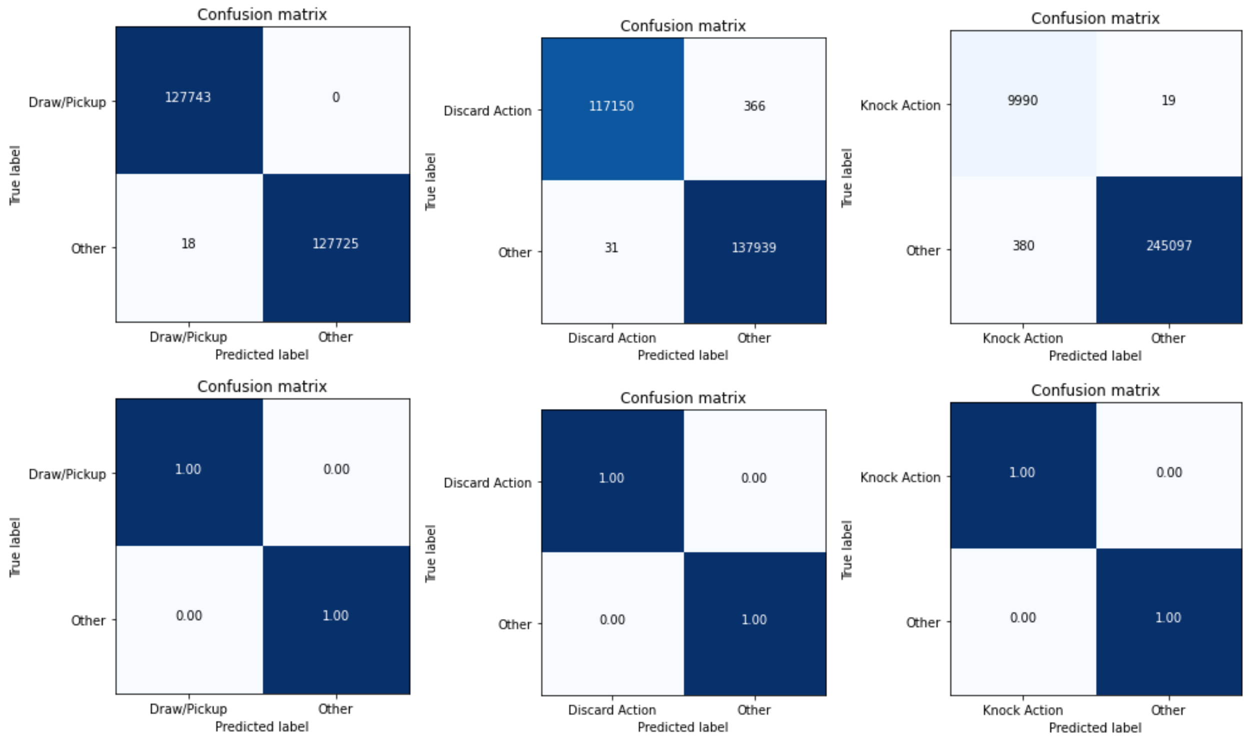


Figure 24: Confusion Matrices for **all_2HL_80K** based on state-action groups – **'draw'** (left), **'discard'** (middle), **'knock'** (right), Normalized Values (bottom). True Label (y-axis), Predicted Label (x-axis). Darker tile = larger number. *2K Games Test Set*

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. 24, right) compared to prior (Figs. 15, 19, right). The faint diagonal within the True **'knock'**, Predicted **'discard'** section of the confusion matrix (Fig. 19, 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 24 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. 24, bottom). It can be noted, however, that the number of **'discard'** misclassification increased (Fig. 24, top middle).

5.5 Pretrained Network Testing

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

5.5.1 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. Further investigation will be performed on the Deeper Neural Networks to verify the learnt lower-level feature.

6 Pretrain-Initialized DQN

This section uses the PyTorch DQN algorithm from [4] to train DQN agents which will be later used to evaluate its performance against the previous SimpleGinRummyPlayer agent. There are numerous ways for an agent to receive the reward:

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

* Note that a **GIN** action was not trained for in the Supervised Learning portion as these action occurrences are extremely rare and serves little differences from a **knock** action.

6.1 DQN Agent vs. Random Agent

DQN agents are trained against a random agent, where each action is picked randomly.

6.1.1 Baseline Agent – Randomly-Initialized Agent

dqn_baseline:

dqn_baseline serves as a baseline for all DQN agents. The macro average reward is approximately -0.55, which serves as a benchmark when comparing future training iterations.

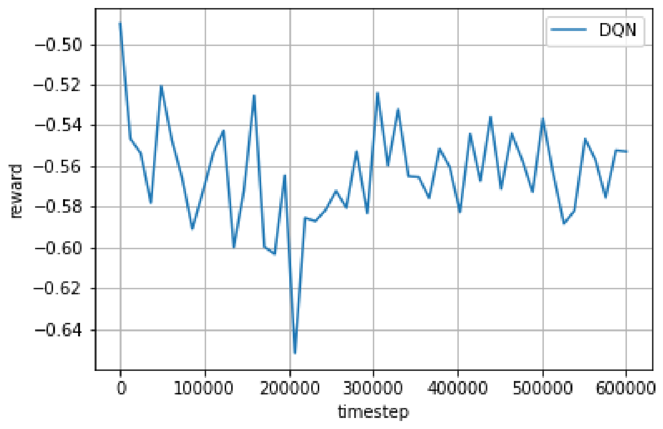


*Figure 25: Micro Average Reward plot for **dqn_baseline***

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

- Learning Rate = 0.00005
- MLP = [*260*, **520**, *110*] = [*input*, **HL1**, *output*] (*italicized are default sizes*)
- Sigmoid Activation between layers
- # Episodes = 5000

6.1.2 Pretrain-Initialized Agent

6.1.2.1 Pretrained Baseline Agents

Using three **all state, all action** models produced in 4.1.4 (**all_1HL, all_2HL_40K, all_2HL_80K**), the network weights were used to initialize the DQN Neural Network parameters. No additional top layer was initialized in the DQN Neural Network and the network parameters were frozen. This was to test the macro average reward of each DQN agent prior to any additional training from the DQN algorithm.

dqn_all_1HL_baseline:

The macro average reward for **dqn_all_1HL_baseline** is around 0.22, which is slightly above the threshold of 0.2 required to **knock**. This indicates that the **dqn_all_1HL_baseline** agent, or the **all_1HL** model, is able to **knock** with positive deadwood each turn against the random agent.

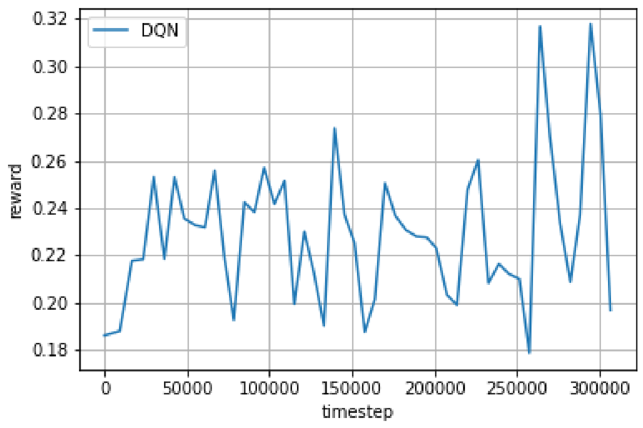


*Figure 26: Micro Average Reward plot for **dqn_all_1HL_baseline***

The following are non-default hyperparameters used to train **dqn_all_1HL_baseline**:

- Learning Rate = 0.00005
- MLP = [*260*, **520**, *110*] = [*input*, **HL1**, *output*] (*italicized are default sizes*)
- Sigmoid Activation between layers
- Frozen bottom layers
- # Episodes = 5000

dqn_all_2HL_40K_baseline:

The macro average reward for **dqn_all_2HL_40K_baseline** is also around 0.22.

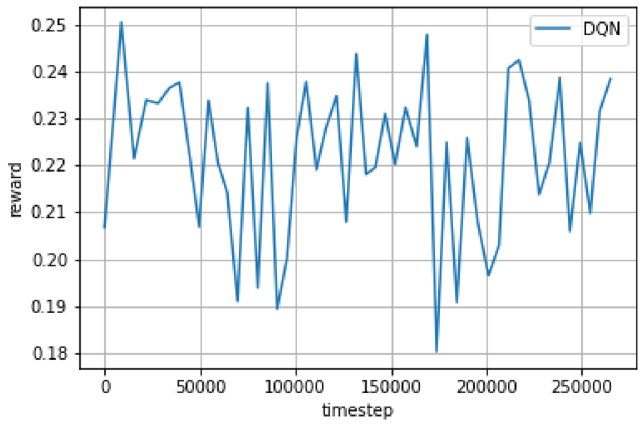


Figure 27: Micro Average Reward plot for ***dqn_all_2HL_40K_baseline***

The following are non-default hyperparameters used to train **dqn_all_2HL_40K_baseline**:

- Learning Rate = 0.00005
- MLP = [*260*, **520, 520**, *110*] = [*input*, **HL1, HL2**, *output*] (*italicized are default sizes*)
- Sigmoid Activation between layers
- Frozen bottom layers
- # Episodes = 5000

dqn_all_2HL_80K_baseline:

The macro average reward for **dqn_all_2HL_80K_baseline** is also around 0.22 to 0.23.

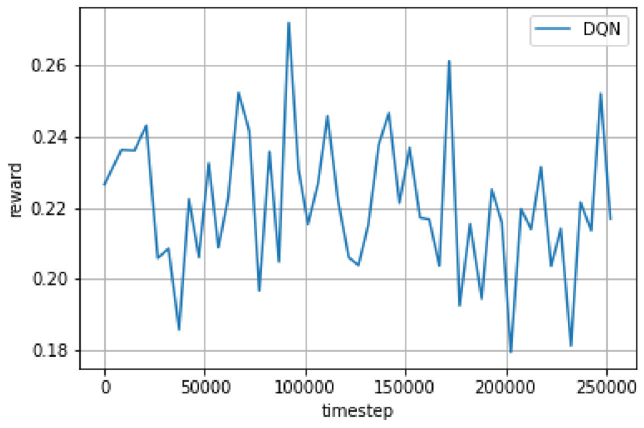


Figure 28: Micro Average Reward plot for ***dqn_all_2HL_80K_baseline***

The hyperparameters used to train **dqn_all_2HL_80K_baseline** are the same as the parameters used to train **dqn_all_2HL_40K_baseline**.

6.1.2.2 Pretrained Agents with Top Layer

The three **all state, all action** models were initialized as the bottom layers of a DQN agent with a single top layer.

dqn_all_1HL:

The average reward for **dqn_all_1HL** begins at around -0.55, which is expected as the top layer weights are randomized, similar to that of **dqn_all_1HL_baseline**. The final average reward oscillates around 0. This is below the baseline reward achieved from **dqn_all_1HL_baseline** of around 0.22.

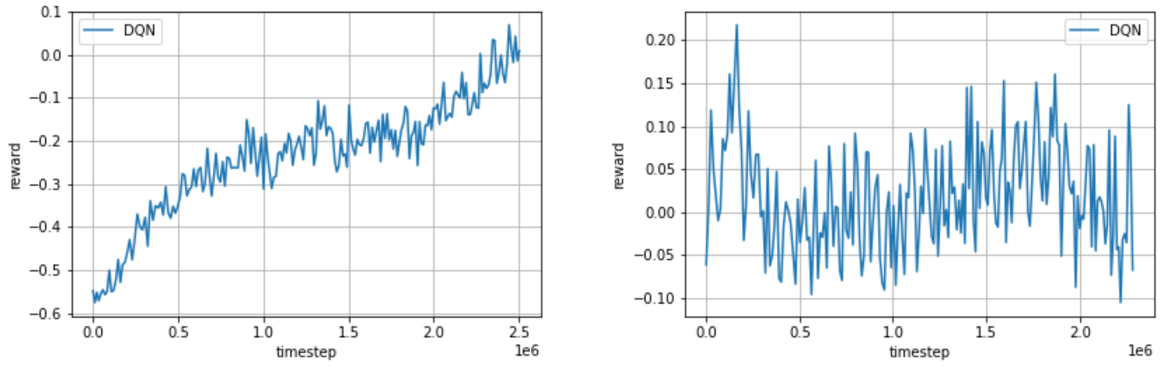


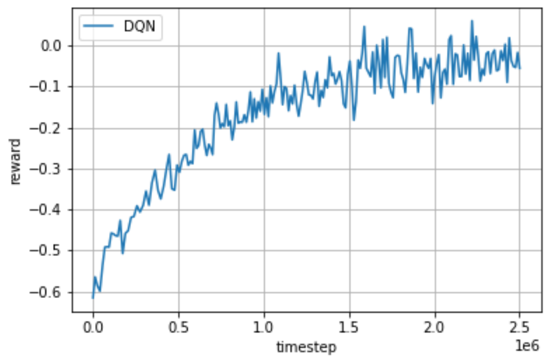
*Figure 29: Micro Average Reward plot for **dqn_all_1HL**. Frozen weights (left), Additional training, unfrozen weights (right)*

The following are non-default hyperparameters used to train **dqn_all_1HL**:

- Learning Rate = **0.00001**
- MLP = [*260*, **520, 520, 110**, *110*] = [*input*, **HL1, HL2, TL1**, *output*]
- Sigmoid Activation between layers
- Randomly Initialized Top Layer Weights (left), Copied Model Parameters (right)
- Frozen bottom layers (left), Unfrozen (right)
- # Episodes = 20000 and 20000

dqn_all_2HL_40K:

The average reward for **dqn_all_2HL_40K** also begins at around -0.55, which is expected. The average reward peaks at around 0.2, which is the baseline result for a pretrained, no top layer agent however the final average reward dips to a negative value of around -0.2.



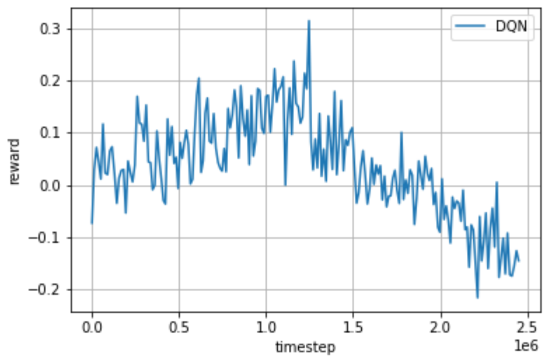


Figure 30: Micro Average Reward plot for ***dqn_all_2HL_40K****. Frozen weights (left), Additional training, unfrozen weights (right)*

The following are non-default hyperparameters used to train **dqn_all_2HL_40K**:

- Learning Rate = **0.00001**
- MLP = [*260*, **520, 520, 110**, *110*] = [*input*, **HL1, HL2, TL1**, *output*]
- Sigmoid Activation between layers
- Randomly Initialized Top Layer Weights (left), Copied Model Parameters (right)
- Frozen bottom layers (left), Unfrozen (right)
- # Episodes = 20000 and 20000

dqn_all_2HL_80K:

Due to poor performance in both **dqn_all_1HL** and **dqn_all_2HL_40K**, the top layer weights and biases of **dqn_all_2HL_80K** were initialized as the previous output weights of the pretrained network. From initial observations, the average reward for **dqn_all_2HL_80K** is about 0.2, which is to be expected as that is the average reward for **dqn_all_2HL_80K_baseline**. After the first 20K episodes of training, the agent reached an average reward of approximately 0.3, a performance better than the baseline. An additional 20K episodes of training, with unfrozen weights yielded little results, with a macro average reward oscillating around 0.25. This macro average reward is better than the baseline agent counterpart.

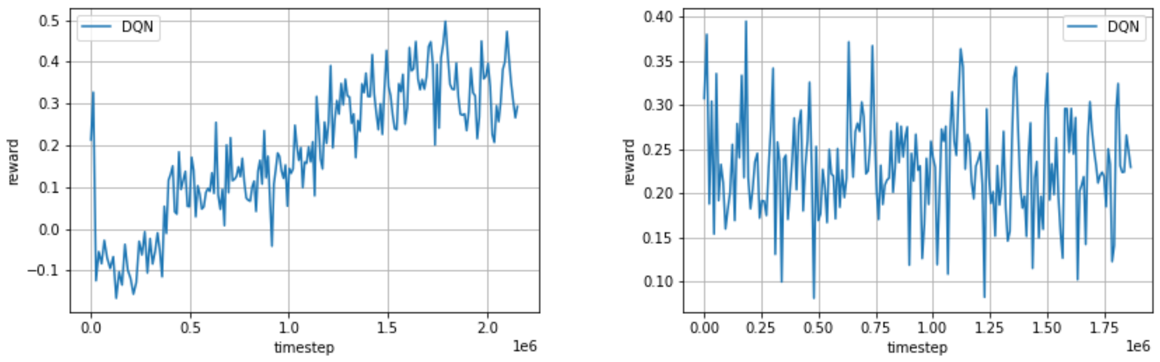


Figure 31: Micro Average Reward plot for ***dqn_all_2HL_80K****. Frozen weights (left), Additional training, unfrozen weights (right)*

The following are non-default hyperparameters used to train **dqn_all_2HL_80K**:

- Learning Rate = **0.00001**
- MLP = [*260*, **520, 520, 110**, *110*] = [*input*, **HL1, HL2, TL1**, *output*]
- Sigmoid Activation between layers
- Hardcoded Top Layer Weights (left), Copied Model Parameters (right)
- Frozen bottom layers (left), Unfrozen (right)
- # Episodes = 20000 and 20000

7 Agent Performance Evaluation

The three **all state, all action** models produced in 4.1.4 (**all_1HL, all_2HL_40K, all_2HL_80K**), and the three **dqn** models produced in 4.2.1.2 (**dqn_all_1HL, dqn_all_2HL_40K, dqn_all_2HL_80K**) are evaluated against a random agent, and the SimpleGinRummyPlayer Agent to test the performance of training from Supervised Learning and the DQN algorithm.

Table 3: Win/Loss Against Random Agent

Model	**Win-Loss Record**
all_1HL	**2000-0**
all_2HL_40K	**2000-0**
all_2HL_80K	**2000-0**
dqn_all_1HL	**-**
dqn_all_2HL_40K	**1996-4**
dqn_all_2HL_80K	**-**

Table 4: Win/Loss Against SimpleGinRummyPlayer Agent

Model	**Win-Loss Record**
all_1HL	**130-870**
all_2HL_40K	**197-803**
all_2HL_80K	**246-754**
dqn_all_1HL	**-**
dqn_all_2HL_40K	**2-998**
dqn_all_2HL_80K	**6-994**

From the preliminary results, the both the pretrained networks and the dqn agents were well below the 0.500 mark.

8 Future Work

8.1 Pretrained Network Testing

Currently, no insight behind the lower-level features is being investigated in the larger models, as outline in Section 5.5.1. The melds testing has only been applied to the **apad_knock** model however should be expanded upon to all models, especially **all_1HL, all_2HL_40K,** and **all_2HL_80K.** Additional lower-level features should also be investigated if time permits. Another insight that would be useful is to investigate the amount of training data required to train the supervised network. Shown in Section 5.4, doubling the number of **'knock'** training samples from 40K to 80K made a significant difference in the classification of **'knock'**. The rationality could be that the samples of each class group is balanced, as **all_2HL_80K** had a 1:1 **'discard':'knock'** ratio. There currently are limitations with increasing the number of training samples however the same ideology can be applied in reverse.

8.2 DQN Training

As seen in training **dqn_all_1HL** and **dqn_all_2HL_40K**, the agents were unable to reach the same level of reward as their baseline counterparts. Possible issues that may cause this are:

- Too large learning rate – a desirable local minima cannot be found
- Weight fragility – parts of the pretrained network is already in an optimal part of the parameter space and any deviation will end in loss of progress
- Feature biases – **dqn_all_1HL** may be biased towards certain actions due to class imbalance, causing little progression in minority class weights

To address these issues, hyperparameter searching and weight analyses of the hidden units are required to better understand what is occurring during training. A technique that might be beneficial in debugging the weights is tracking the evolution of the change in weights throughout training. Another possible technique to achieve this is freezing the weights of one network while allowing another network to freely explore the parameter space. This allows both exploration and exploitation of the pretrained network and any potential learning for the agent.

Additionally, the DQN algorithm currently trains against a random agent. The random agent may be too simplistic to gain any meaningful learning for the DQN agents. A next step would be to implement self-play in the DQN environment to allow the DQN agents to learn against itself. Another agent that should be considered, if time permits, is to implement the SimpleGinRummyPlayer agent into the DQN agent.

8.3 Agent Performance Evaluation

As seen in the results in Section 7, all DQN agents perform terribly against the SimpleGinRummyPlayer agent that was used to train the pretrained network. Additionally, the pretrained networks cannot achieve a 0.500 record against the same agent, with the best results being 0.246. Both results are worrying as the pretrained networks are designed to predict the actions that will subsequently follow a given state, achieving near 100% accuracy and the DQN agents are supposedly trained longer. A plausible explanation for the DQN agents is that the learning was lost while training against a random agent. Further work should be done to debug the issues as there may be a flaw in the implementation of the agent in the tournament environment.

8.4 Work Plan

The following is the tentative work plan used to schedule deliverables and milestones for the remaining part of the thesis project.

Feb. 1	• Thesis Interim Report Submission
Feb. 2 to Feb. 15	• DQN Agent self-play implementation (8.3) • DQN weight fragility/hyperparameter investigation (8.2)
Feb. 16 to Mar. 1	• Meld Feature Investigation (8.1) • Additional Features and Network Data Investigation (8.1)
Mar. 2 to Mar. 15	• Meld Feature Investigation (8.1) • Additional Features and Network Data Investigation (8.1)
Mar 16 to Apr. 7	• Thesis Final Presentation • Thesis Final Report
Apr. 7	• Thesis Final Presentation Submission • Thesis Final Report Initial Draft
Apr. 16	• Thesis Final Report Submission

9 Appendix

[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