USP Independent Study Module (ST)

Group Report

Mastering the game of Onitama with Reinforcement Learning

Student Names, Module Code:
Arijit Dasgupta, UIS3901S
Chong Yu Quan, UIS3901S

Student Number:
A0182766R
A0136286Y

Student Emails:
arijit.dasgupta@u.nus.edu
chong.yuquan@u.nus.edu

April 16, 2021

Contents

List of Figures

1 Introduction

Artificial intelligence (AI) is one of the most interesting and fastest developing fields in the world of science today. AI goes by many definitions, one of the most general one being intelligence shown by a machine (such as a computer), as opposed to a living being (such as a human or animal). In the recent years, developments in AI technology have allowed many new applications for AI, such as facial recognition and self-driving cars. While the achievements of AI are magnificent to behold, it is also important to further classify these AIs in order to properly study them. One way to classify AI is to study how their algorithm works. While there are many ways that an algorithm can be implemented, generally a proven algorithm(s) will be implemented and minor tweaks are made as needed to better suit the task at hand. For this project in particular, we will attempt to create an AI that can play the board game Onitama using deep reinforcement learning.

There have been many attempts to create AIs to play board games before. Board games are normally chosen for their well-defined environment which eases the representation of game states, significantly simplifying the code. Two of the earlier algorithms that are commonly used to play board games are the Minimax and Monte-Carlo Tree Search algorithms, which do their task by mapping out possible moves and choosing the best out of those moves. However, the method that this project will be attempting, reinforcement learning, relies more on "training"; the AI will be fed data beforehand, and it will process the data in order to help it play the game in the future. This method has been used to create AIs that surpass human abilities in many games, perhaps most famously AlphaGo defeated Lee Sedol, considered to be the best Go player of all time, in 2016. Perhaps equally important is the rise of new chess engines that rely on reinforcement learning, such as Leela Chess Zero and AlphaZero, which are capable of defeating Stockfish, which for many years reigned unchallenged as the world's most powerful chess engine with its advanced Minimax and pruning algorithms. This showed the potential of reinforcement learning to match and even outperform humans and older algorithms, sparking much interest in the field.

2 Onitama Environment

2.1 Rules

A summary of the rules for Onitama is available in the appendix. The most important aspect to note are the two win conditions for Onitama, as these define the tasks that our AI will aim to accomplish. We refer to the first win condition as 'destroy'; done through capturing the opponent's master (a pawn and the master can both capture the opponent's master), Figure 2.1. The second win condition, 'conquer' is achieved by moving our master to the opponent's temple (the square where the opponent's master starts the game on), Figure 2.2. An implication of these definitions in the code is that if a victory by destroy and conquer is achieved simultaneously (Figure 2.3), it will be treated as victory by destroy, as far as the statistics is concerned (for Chapter 4).

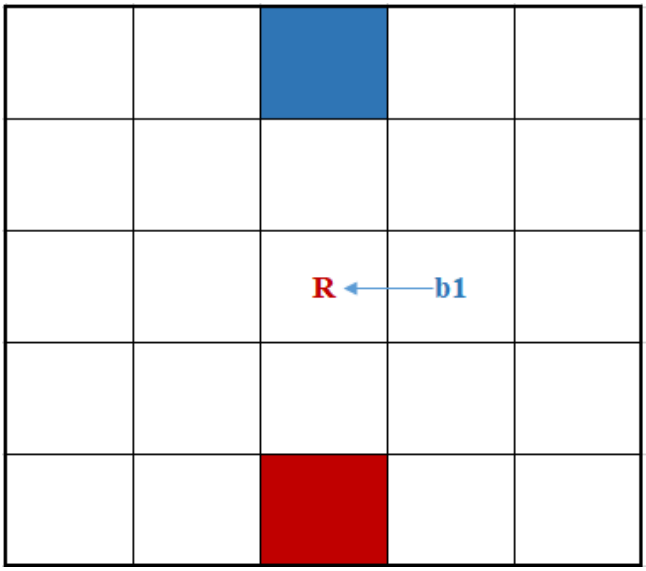


FIGURE 2.1: An example of 'Destroy'; Blue pawn capturing Red master

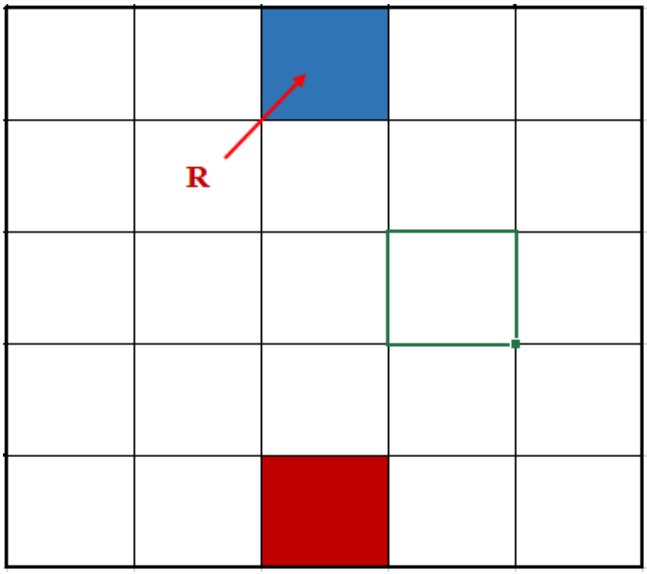


FIGURE 2.2: An example of 'Conquer'; Red master moving into Blue temple

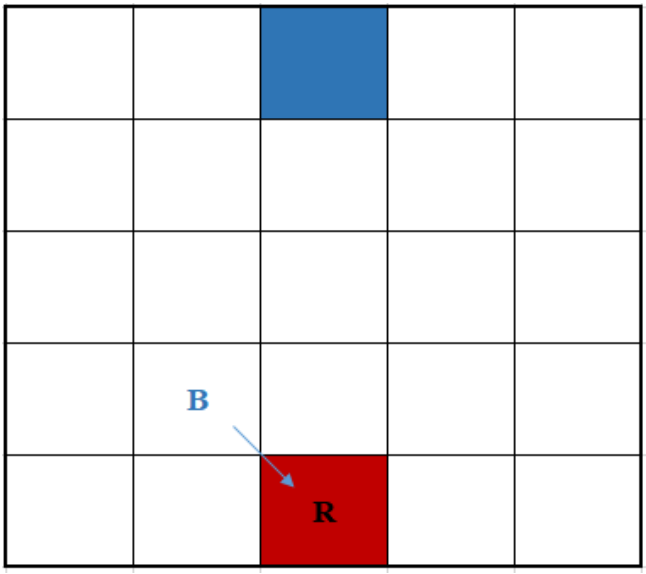


FIGURE 2.3: 'Conquer' and 'Destroy' achieved simultaneously; Blue master capturing Red master, while moving into Red temple

2.2 Code representation

There are two parts of the code that needs to be defined: the game itself and the AI that will be implemented to play the game. Both aspects will be coded using Python. The game will be represented as a class encompassing all the gameplay features (moving, capturing, card distributions, etc.). The board itself is represented using a 2D, 5-by-5 array, each square being represented as a tuple in the form of (i, j) following the Cartesian Coordinates with the origin (0, 0) located on the top-left (blue's side) of the board, as seen in Figure 2.4 below.

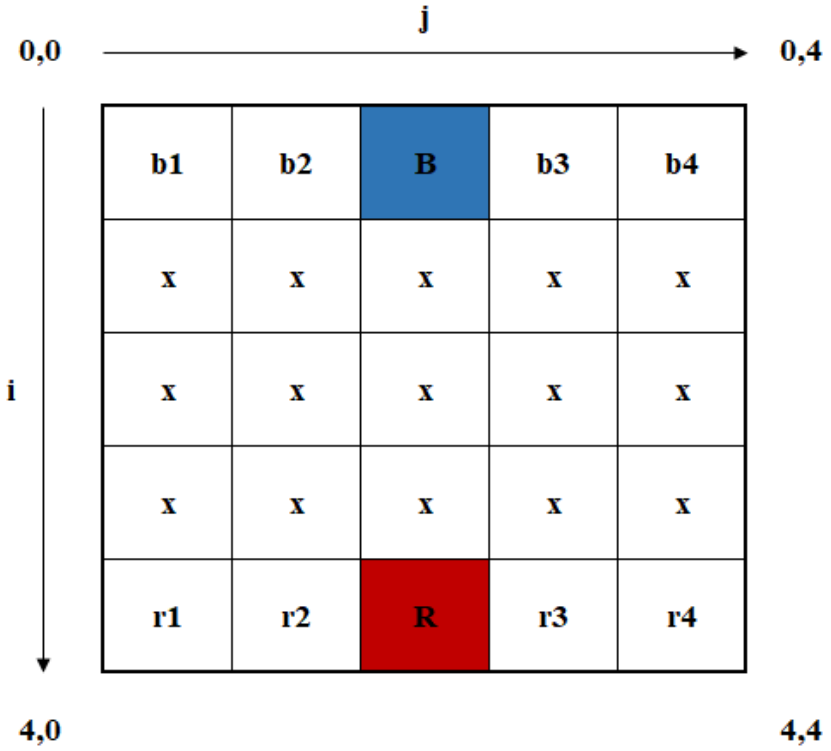


FIGURE 2.4: Coordinate convention for the board and naming convention for the pieces

To select and move the pieces itself, we have named each of the pieces separately, rX and bX representing the 4 pawns on each side while R and B represent the respective colour's master, also displayed in Figure 2.1. Each piece will be stored in the 'piece_state' dictionary with the name of the piece as a key which corresponds to the coordinates of that piece (if it is still alive) or '-1' if it is dead. While it is also possible to not number the pawns, as the pawns are equivalent to each other (e.g. r1 on (1, 1) and r2 on (2, 2) is technically equivalent to r3 on (2, 2) and r4 on (1, 1)), this numbering convention was adopted to make coding the game simpler. This comes at the cost of the efficiency of the AI, as it will evaluate different pawns as distinct entities, instead of copies of a single entity, which dramatically increases the number of positions that will be evaluated when training and making a decision.

Similar to the pieces, all 16 cards will also be saved in a dictionary, with the name of the card as the key that corresponds to the moveset of the card (A, B, C, D), each of which is represented as a change in coordinates. There is a single card with 2 moves (tiger), 10 cards with 3 moves, and 5 cards with 4 moves. Each of the cards also have a colour (red or blue) assigned to them, stored in a third, similar dictionary.

To perform a move, the code requires 3 inputs: the name of a piece, the name of a card, and a letter corresponding to a move. Any invalid input (e.g. dead piece, invalid moves, cards that are not in hand) will be rejected and the user will be prompted to enter another input. Assuming all 3 inputs are valid (e.g. R Tiger A), the move will be executed (in this case, the red master will move 2 steps in the negative i-direction).

The two win conditions (explained earlier) corresponds to their own code representations. 'Conquer' is achieved when the red master (R) is represented by '-1' (dead) in the 'piece_state' dictionary which means that the blue player won. 'Destroy' is achieved when the blue master (B) is represented by (4, 2) (the red temple) in the same dictionary, causing a blue victory. The opposite applies for the red player's victory conditions.

3 Algorithm

3.1 Policy Based Reinforcement Learning

For this project, the team focuses on policy-based learning algorithms, from which most state-of-the-art algorithms are based on. Unlike value-based learning that generates a policy from the approximated value function, policy-based learning directly parametrises the policy. The advantages of policy-based learning over value-based reinforcement learning are that it has better theoretical and empirical convergence properties and the fact that it can handle high dimensional and even continuous action spaces. Besides, it can learn stochastic policies, which is the generalisation of deterministic policies. This is quite a crucial advantage as deterministic policies cannot solve some problems. However, the drawbacks of policy-based methods are that they often converge to a local optimum instead of a global one. Furthermore, evaluating a policy can be very inefficient and high variance. Policy-based methods can be seen as an optimisation problem to find the parameter θ that minimises the cost function $J(\theta)$. Although there are methods that do not use gradients (e.g. hill-climbing, simplex, genetic algorithms), greater efficiency is often possible with gradient-based methods. As a result, the project focuses on policy gradient reinforcement learning methods in particular. Policy gradients algorithms iteratively search for a local minimum by descending the local gradient of policy concerning θ as shown in equation 3.1, where $\nabla_\theta J(\theta)$ is the policy gradient, and α is learning rate.

$$\begin{aligned} \Delta\theta &= -\alpha \nabla_\theta J(\theta) \\ \theta &\leftarrow \theta + \Delta\theta \end{aligned} \tag{3.1}$$

By assuming that the policy, $\pi_\theta(S, A)$ is differentiable when $\pi_\theta(S, A) > 0$ and gradient $\nabla_\theta \pi_\theta(S, A)$ is known, the policy gradient can be computed analytically by using the following identity shown in equation 3.2. where $\nabla_\theta log \pi_\theta(S, A)$ is the score function.

$$\nabla_\theta \pi_\theta(S, A) = \pi_\theta(S, A) \frac{\nabla_\theta \pi_\theta(S, A)}{\pi_\theta(S, A)} = \pi_\theta(S, A) \nabla_\theta log \pi_\theta(S, A) \tag{3.2}$$

The policy gradient theorem generalises to all MDPs, stating that for any differential policy $\pi_\theta(S, A)$ and cost function $J(\theta)$, the policy gradient is as shown in equation 3.3.

$$-\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) * Q_{\pi_\theta}(s, a)] \tag{3.3}$$

In the REINFORCE algorithm that uses Monte Carlo gradient descent, the return $V_\pi(S)$ from a terminated episode is used as the unbiased sample of $Q_{\pi_\theta}(s,a)$. However, it was discovered that despite using an unbiased value estimate, high variance is still observed in the policy gradient, which slows down learning. Also, the learning process is extremely computationally expensive, requiring millions of episodes for convergence for simple tasks. Given the problems of the Monte Carlo gradient descent, the Actor-Critic policy gradient algorithms are introduced. Actor-Critic policy gradient algorithms maintains two sets of learned parameters as follows: 1) Critic: Updates ω parameters to estimate a value function (e.g. $Q_{\pi_\theta}(s,a) \approx Q_\omega(s,a)$), 2) Actor: Updates θ parameters to estimate policy $\pi_\theta(s,a)$. Actor-Critic policy gradient approaches follow an approximate policy gradient as shown in equation 3.4 below.

$$\begin{aligned} -\nabla_\theta J(\theta) &\approx \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q_\omega(s,a)] \\ \Delta\theta &= \alpha * \nabla_\theta \log \pi_\theta(s,a) Q_\omega(s,a)) \end{aligned} \tag{3.4}$$

3.2 Deep Deterministic Policy Gradients (DDPG)

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

FIGURE 3.1: Pseudocode for DDPG

After the introduction of Actor-Critic policy gradient algorithms in the previous section, the DDPG algorithm can now be elaborated with its pseudocode shown in Figure 3.1. DDPG is an actor-critic,

model-free algorithm based on the deterministic policy gradient that can operate over continuous action spaces developed by Lillicrap et al. [1]. It consists of four fully connected deep neural networks (FCDNN), namely the actor FCDNN, critic FCDNN, target actor FCDNN and the target critic FCDNN, and a replay buffer. The replay buffer takes inspiration from Deep Q-Learning (DQN) from the work of Mnih et al. [2], where the agent's experiences at each time-step, $E_t = (S_t, A_t, R_t, S_{t+1})$, are stored in a replay buffer, $D = E_1, E_2 \dots E_N$. The replay buffer has a finite space (e.g. 1000000 episodes) and is based on a first in first out principle in replacing experiences stored in the replay buffer. During the training process of the FCDNNs, a specified batch (e.g. 512 episodes) of the experiences stored in the replay buffer will be randomly sampled for learning. The actor FCDNN takes in state observations and generates softmax outputs with the dimensions of the action space. On the other hand, the critic FCDNN takes in state observations concatenated with the corresponding actions taken by the agent sampled from the replay buffer to give a single output of the state-action value, $Q_\omega(s, a)$. The target actor FCDNN and target critic FCDNN follows an identical architecture with the actor FCDNN and critic FCDNN, respectively. However, the two target FCDNNs are not utilised during the experience gathering episodes but for the training of the actor and critic FCDNNs. The weights of each hidden unit in the target FCDNNs are updated based on a parameter, τ, for its corresponding FCDNN counterpart as shown in equation 3.5 below.

$$\begin{aligned} \theta'_{target\ actor} &= \tau\theta_{actor} + (1-\tau)\theta_{target\ actor} \\ \omega'_{target\ critic} &= \tau\omega_{critic} + (1-\tau)\omega_{target\ critic} \end{aligned} \tag{3.5}$$

On initialising the FCDNNs, the target FCDNNs are a hardcopy of their original counterparts, i.e. $\tau = 1$. After every subsequent training of the actor and critic FCDNNs, the target FCDNNs undergoes a softcopy update (e.g. $\tau = 0.005$). The training losses of the actor and critic FCDNNs are stated in equation 3.6 below.

$$\begin{aligned} -\nabla_\theta J(\theta) &\approx \frac{1}{N}\sum_i \nabla_A Q_{\omega_{critic}}(S, A_{actor}) \nabla_{\omega_{actor}} \mu_{\omega_{actor}}(S) \qquad (actor) \\ \delta_{\omega_{critic}} &= r + \gamma Q_{\omega_{target\ critic}}(S', A'_{target\ actor}) - Q_{\omega_{critic}}(S, A) \qquad (critic) \end{aligned} \tag{3.6}$$

For the critic losses with experiences from the replay buffer, the action of the target actor FCDNN from S', $A'_{target\ actor}$, is first obtained. From S' and $A'_{target\ actor}$, $Q_{\omega_{target\ critic}}(S', A'_{target\ actor})$ can be obtained from the target critic FCDNN, from which added to reward, r, gives the TD target. The critic loss is hence the mean squared error, δ^2_ω, between the TD target and the state-action value, $Q_{\omega_{critic}}(S, A)$, from the critic FCDNN. For the actor losses, the action selected with an updated policy from the actor FCDNN, A_{actor}, is obtained. The actor losses are then the gradient of the state-action value, $Q_{\omega_{critic}}(s, A_{actor})$, with respect to A_{actor} using S and A_{actor} from the critic FCDNN multiplied with the gradient of actor FCDNN's policy output given S input, $\mu_{\omega_{actor}}(S)$, with respect to the actor FCDNN's parameters, ω_{actor}.

4 Reinforcement Learning Cast

4.1 State

The reinforcement learning cast is a very crucial aspect that one must carefully define. To define the state of the board game, we specifically look at the situation on the board and the cards present in the game-play. To simplify the computational requirements of the training process, we decided to use only 5 of the 16 cards available in Onitama. Each of these 5 cards have a total of 4 possible moves. There are a total of $\binom{16}{5} = 4368$ ways of selecting 5 cards from 16 possible choices. These cards are 'dragon, 'elephant', 'goose', 'rooster' & 'monkey'. Note that the order of these 5 cards as mentioned matter. This will make it significantly easier for the agent to learn to play the game as it does not need to learn which one of the 16 cards are at play. To represent the board state, we propose using 10 layers of a 5×5 grid as show in Figure 4.1.

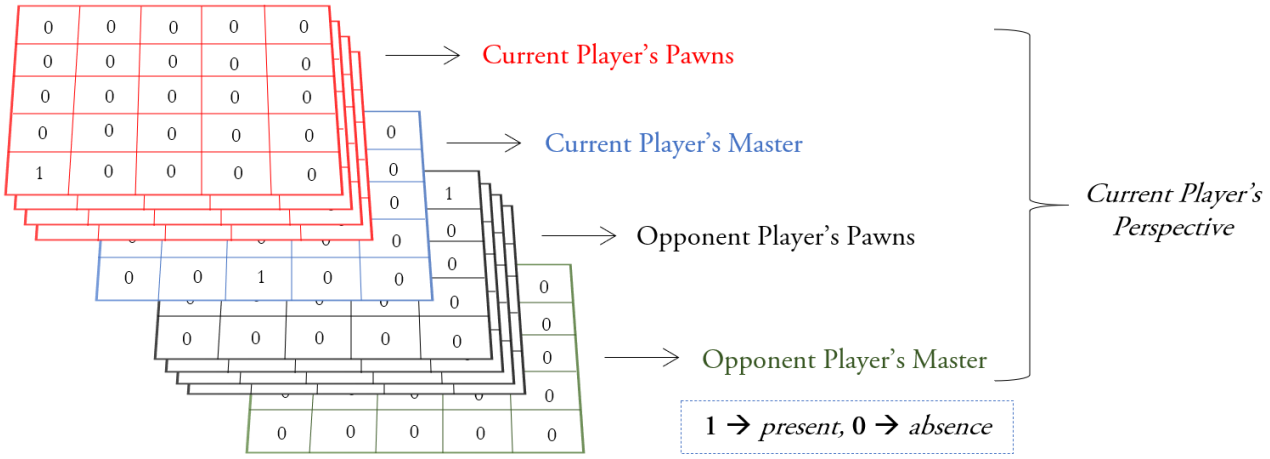


FIGURE 4.1: State representation of the board

The first four layers of the board state represent the current player's pawns (from pawns 1 to 4 in order). The 5×5 grid represents the spatial location of the piece. All spatial locations in the 10 layers are taken from the perspective of the current player. A 1 indicates presence while 0 indicates absence. The fifth layer represents the positions of the current player's master piece. The final five layers follow the same pawns and master positions, but it is for the opponent player instead. This $10 \times 5 \times 5$ representation of the board state can be taken in by a CNN. One may ask, as all four pawns are identical pieces, swapping any two pawns should not change the board, so why split them apart?. This is a valid question, and the reason will be explained in section 4.2. Representing the card state

is easier as it can be represented by a one-hot encoded vector of length 10. Figure 4.2 shows how the card state is represented.

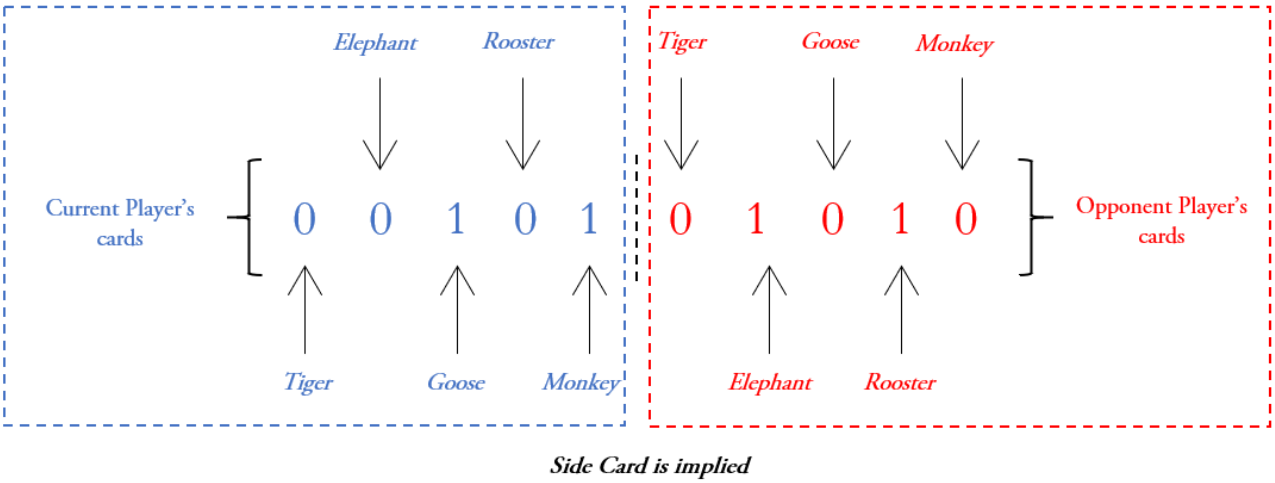


FIGURE 4.2: State representation of the cards

The first 5 inputs of the card state represent the presence of the current player's cards. As any player has 2 cards at all times, two of the 5 inputs will be 1 while the rest are 0. The same rule applies for the next 5 cards which represent the presence of the opponent's cards. As the side card must be the card not represented in either half of the one-hot encoded vector, we consider the choice of the side card implied by the one hot vector. In other words, the 10 inputs in the one-hot encoded vector is sufficient to represent the card state of all 5 cards in the game. Note that the order of the cards in the one-hot encoded vector is always 'dragon, 'elephant', 'goose', 'rooster' & 'monkey', and this order is consistent throughout all training. For instance, this means that the agent is expected to learn that a '1' in the second position always refer to the 'elephant' card. This card state can be used as an input into a fully connected layer.

4.2 Action

When executing a turn in Onitama, the player can have a maximum of 5 pieces on the board, with two cards and a maximum of 4 moves per card. Assuming that all pieces are able to make a valid move with all 4 of the moves in both cards, we have a total of $5 \times 2 \times 4 = 40$ maximum possible moves in one turn. In most cases however, not all combinations of moves and pieces are always valid (moves outside the board or on a piece from the same team), hence usually there are less than 40 possible moves a piece can make in one turn. Nonetheless, the upper limit if this number is important as that will represent the number of neurons in the output of our neural networks. Figure 4.3 illustrates our proposed discrete action space for the agent.

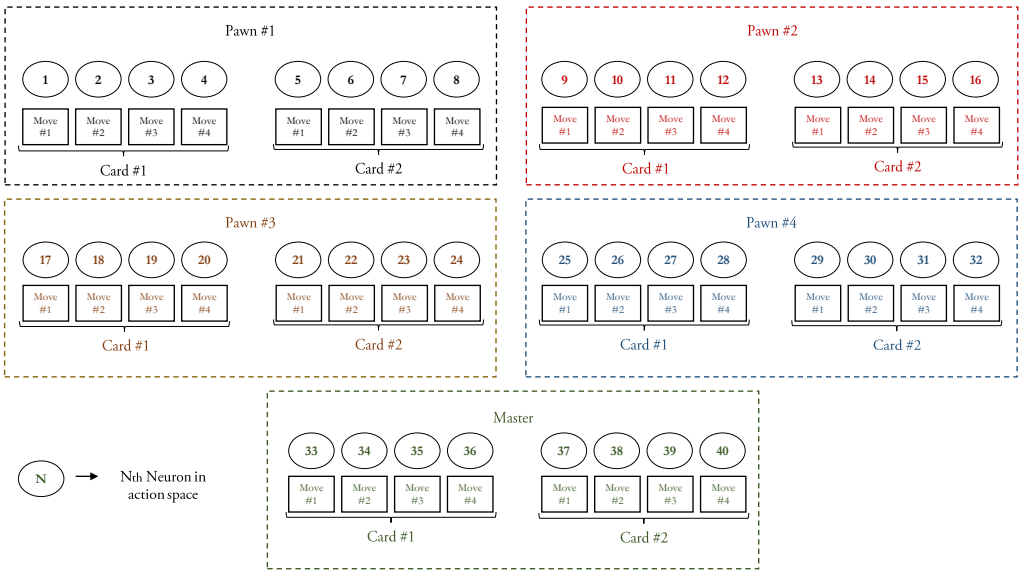


FIGURE 4.3: Representation of the action space of the agent

As Figure 4.3 shows, the 40 actions are split into the 5 pieces and each one of them are split into 2 cards, which have their 4 moves in a consistent order. The order of the pieces are from pawns 1 to 4 and then the master piece. The card 1 and 2 ordering follow the same order as specified in the card state. So if the player has the 'rooster' and 'elephant' cards, card 1 will be 'elephant' and card 2 will be 'rooster'. This order is always consistent and maintain so that it matches the ordering in the card state (Figure 4.2. In our proposed action space, we identify the pawns numbered 1 to 4. We did this so that the agent would know exactly which pawn it should move inside the board and not any random pawn. This is why we needed to separate the pawns in the board state (Figure 4.1) as the agent should be able to tell apart which is pawn 1, or 2 etc. and take the actions specific to those pawns. Imagine if we redefined the board state as shown in Figure 4.4 instead. All the pawns are defined in one layer. Hence, if we swapped any two pawns, it would not change the second layer of the board state. However, when it comes to choosing an action, the agent would not be able to tell which pawn (1 to 4) it should move. The action space output will be different if the two pawns were not swapped for the same exact change on the board. Therefore we stick to the board state representation in Figure 4.1 to go well with our action space for this project.

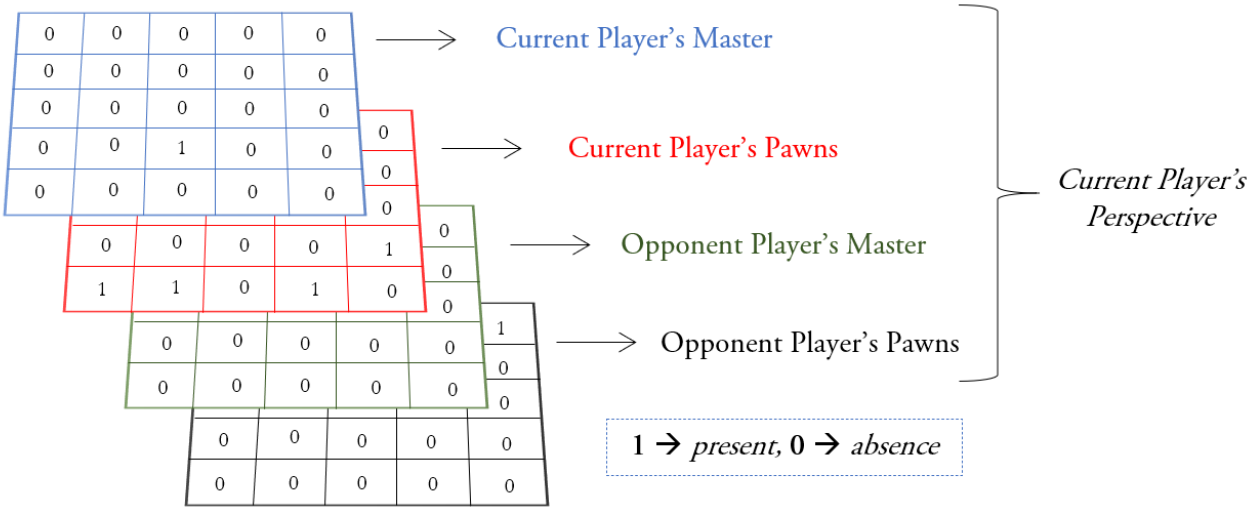


FIGURE 4.4: Alternative board state representation (**not used**)

4.3 Reward & defining the environment rollout

We used an extremely simple reward structure for this project. The reward is 1 if the agent wins after making its move, -1 if it loses after making its move and 0 otherwise. The more interesting discussion is the manner in which we define a rollout of the environment. This is crucial as the points in which we gather S_t and S_{t+1} of the board and card state changes everything about what the agent is learning. At the start of the project, we took S_t right before the agent makes its move and S_{t+1} immediately after the agent makes its move. This seemed intuitive at first, after the agent makes its move, the board and card states change and therefore one should record the change. When we implemented this method, we found that it was impossible for the agent to learn to beat any AI it was training against. We realised that the way we defined the rollout of the environment was completely wrong. The rollout of the environment should not only include the change the agent makes on the board, but the move made by the opponent AI as well. As far as the agent is concerned, the opponent AI is part of the environment. What the agent was learning was not beating the opponent, but the change it had on the board. Therefore S_t should be recorded before the agent makes the move and S_{t+1} should be taken before the agent makes a move in the **next** turn. We found a big difference in how the agent was performing after making this change

5 Methodology

5.1 The issue of validity

The first aspect one must tackle for the training process is how we manage invalid actions. As mentioned in section 4.2, there may be a maximum of 40 possible moves but most of them may not be valid. To illustrate this clearly, Figure 5.1 shows the distribution of the number of valid moves in each turn, taken from a dataset of 71789 turns. The number of valid turns was measured in both agent's and opponent's turns.

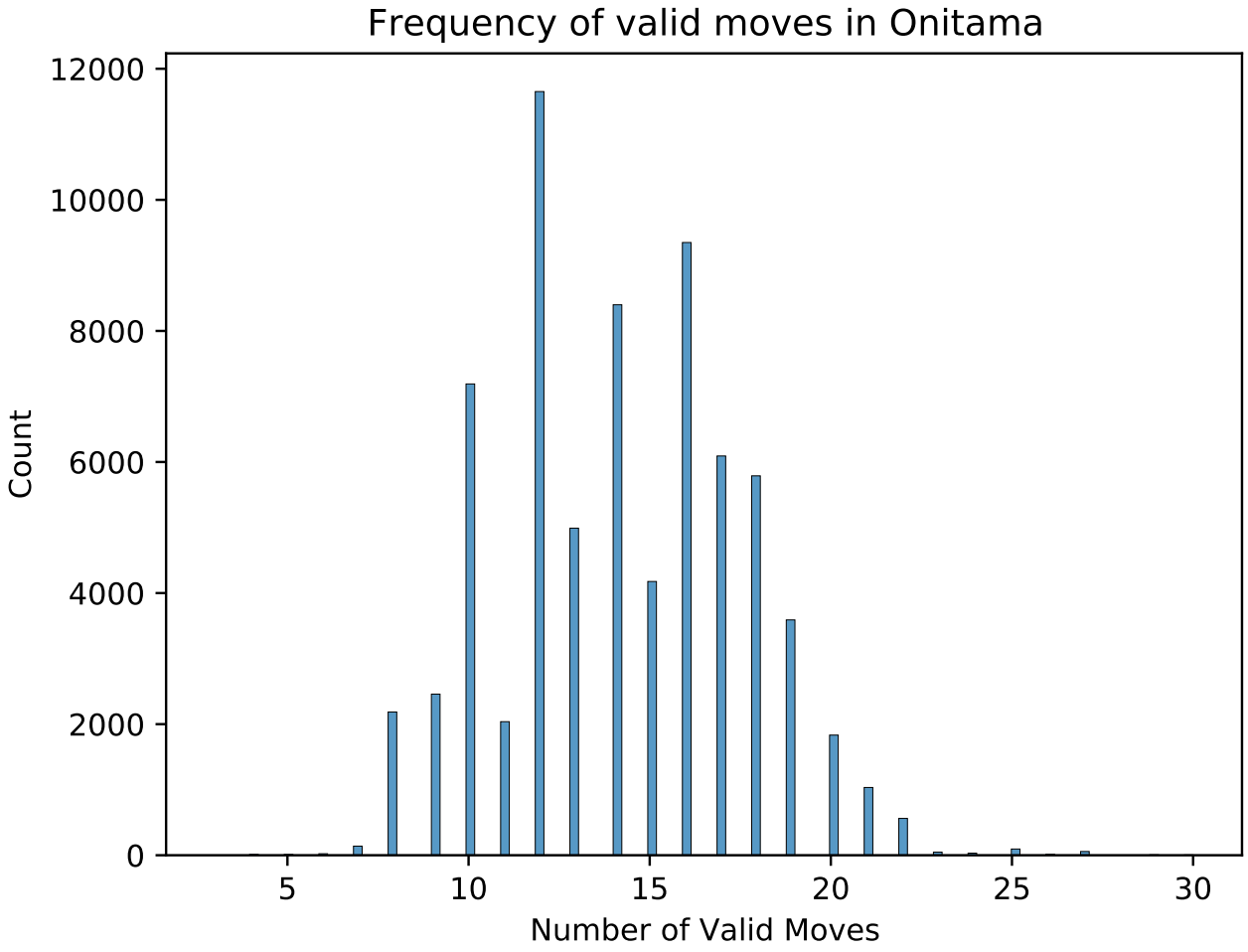


FIGURE 5.1: Distribution of valid moves per turn from a dataset of 71789 turns

It is clear that the board game hardly ever provides the ability for a player to have all 40 choices. In fact the maximum number of valid moves recorded was 30. The average and median are 14.2496 and

14 respectively. The most common value recorded is 12. Note that this distribution was generated when the untrained agent played against a Minimax AI of depth 1, hence different strengths in players may produce slight variations in the values.

The fact that only 14 of 40 output neurons link to valid moves mean that it is necessary for the agent to learn to make valid moves. In the early stages of training however, it is possible for the agent to output an invalid move as the most probably choice. In this case, we implement a 'hand of god' intervention where we force the agent to make a valid move. For each turn, we first extract an array of length 40 that indicates the validity of each move from the board game. A 1 indicates the move is valid and 0 otherwise. We call this array the valid mask. Afterwards we do an element-wise multiplication of the mask onto the action space output. Note that the action space output is a softmax layer, hence all the values of the neurons add to 1. To preserve this property, we re-normalise the values so that the sum of the neurons add up to one. With this, it becomes impossible for the agent to take an invalid action. However, this is not satisfactory as we do not want the agent to rely on this valid mask. If the agent was originally giving an invalid move high values before the valid mask, then the agent must still learn that it made a bad choice.

With this added dimension of validity, the project has two aims. The first is to learn good moves to beat the opponent and the second is to learn valid moves. The best case scenario is if the agent can achieve both. We believe that learning valid moves will improve the first goal of learning better moves as the action space will become constrained, allowing the agent to choose good moves with a higher probability.

5.2 First segment of the neural network

To tackle the issue of validity, we have designed 8 different neural network architectures for testing. Note that only the actor and the target actor networks are varied, while the critic network remains the same. Before delving into the networks to determine branches for validity and actions, we first define preprocess blocks that take in the board state & card state (and actions for critic) and processes them into a 256×1 vector. Figure 5.2 illustrates the 'Actor_Preprocess' block we designed. The board state goes through two layers of convolution and max-pooling. During each convolution, padding is applied to keep the shape, which them gets altered by max pooling. Each layer is also batch normalised. This leads to the data having a shape of (128,1,1) which then gets flattened to a (128,1) vector. On the other side, the card state goes through one fully connected layer to the same size of (10,1) before concatenating with the (128,1) vector. This new vecotr of shape (138,1,1) then goes through two fully connected layers until it has an ultimate shape of (256,1).

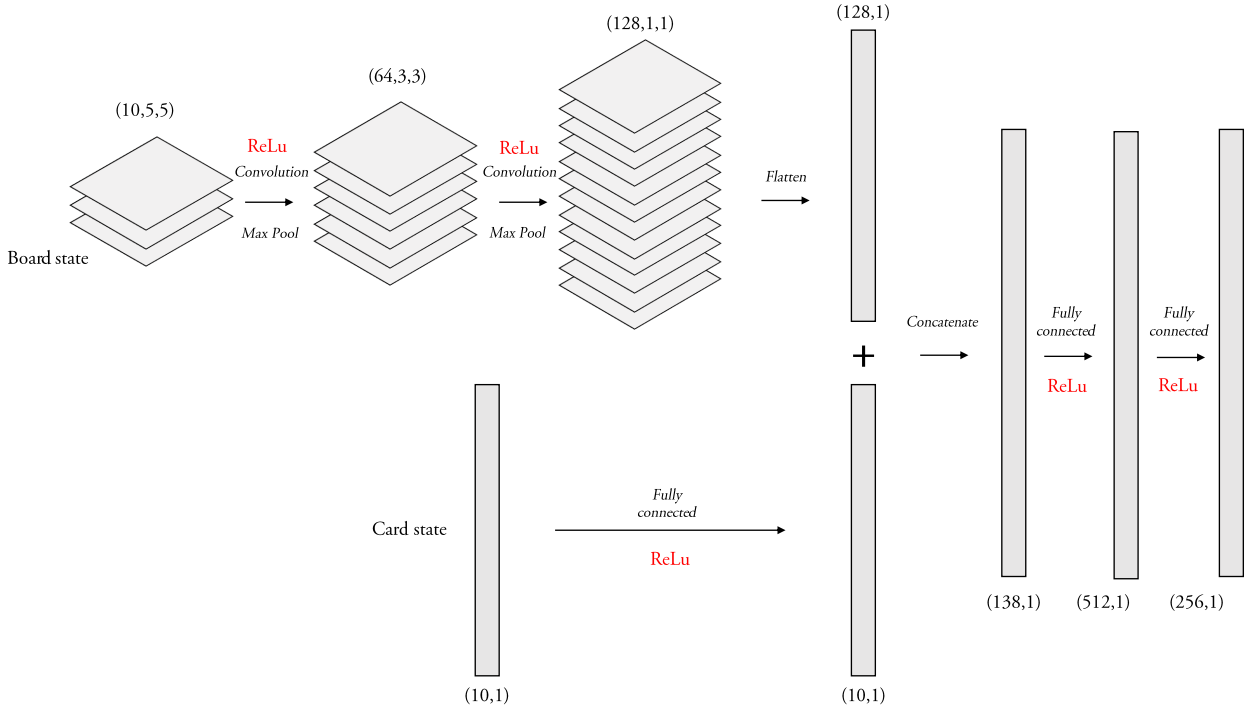


FIGURE 5.2: The 'Actor_Preprocess' Block

The 'Critic_Preprocess' block is almost identical to the 'Actor_Preprocess' block with the only difference being that the former also inputs the actions from the Actor. Note in Figure 5.3, that an extra (4,1) vector of actions goes through one fully connected layer to the same shape before concatenating with the other vectors to form a (178,1) vector.

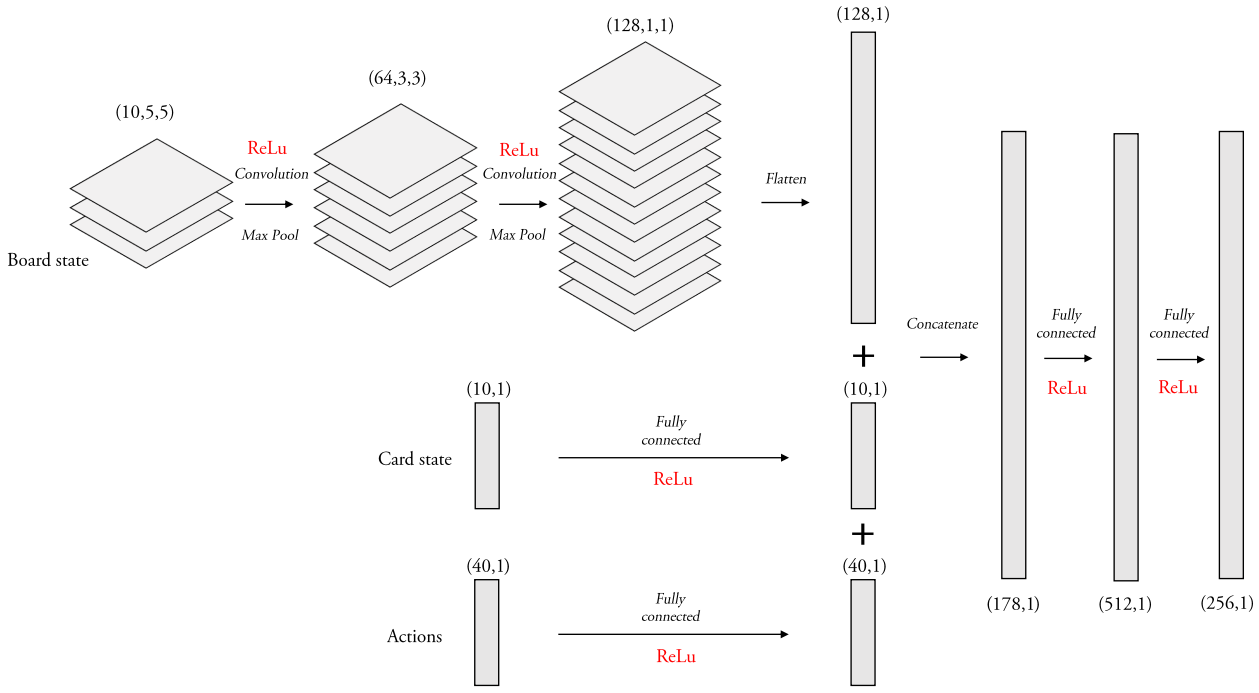


FIGURE 5.3: The 'Critic_Preprocess' Block

The remainder of the critic's network is relatively simple as shown in Figure 5.4. The output of the 'Critic_Preprocess' block only goes to a layer with a single neuron. This neuron has a linear activation function and is made to approximate the Q value.

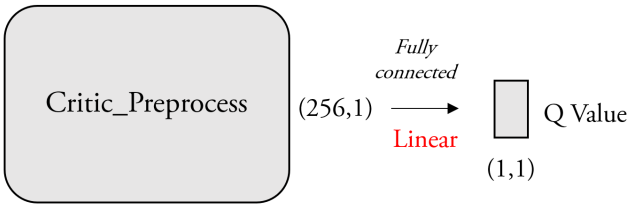
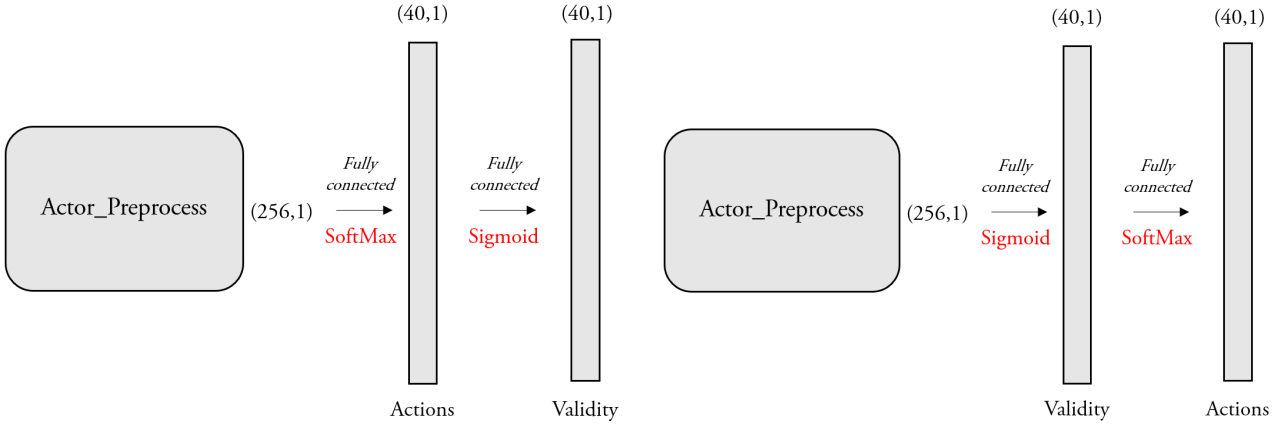


FIGURE 5.4: The DDPG Critic Network

5.3 Branched vs Linear Networks Architectures

To accommodate validity and actions, we embrace two different philosophies in our architectures. The first philosophy is what we call "no_multiply". The first method is the linear network approach which attaches a layer next to the actions Softmax output layer to learn the valid mask. A mean squared error loss is applied for any layer that tries to learn the valid mask in this project. We shall refer to any layer that tries to approximate the valid mask as the validity layer. As the mask is a value that is either 0 or 1, we felt that it was appropriate to use a sigmoid activation function. This method may be considered a softer implementation as it does not directly force the actions layer to learn valid actions. The hope is that, as there is a layer beside that is learning valid actions, the actions layer will resemble the magnitudes set by the validity layer. This may not fully work as the layers are fully connected, so there is no guarantee that the magnitude (high or low) of the first neuron of the actions layer will follow that of the first neuron of the validity layer. Nonetheless, we think it is worth testing this architecture.



(A) The 'val_after_actions' Network

(B) The 'actions_after_val' Network

FIGURE 5.5: Linear Networks

There are two ways that we can put the validity layer beside the actions layer. Figure 5.5 shows both approaches. In Figure 5.5a illustrates the 'val_after_actions' network where the output from the

'Actor_Preprocess' block goes into the actions layer and then into the validity layer. Figure 5.5b simply shows the swap between both layers, and we call this the 'actions_after_val' network.

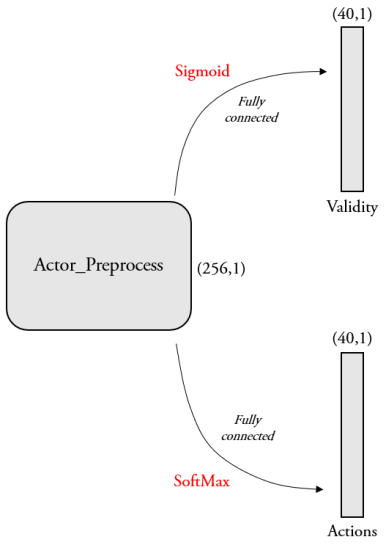


FIGURE 5.6: The 'val_branch_actions' Network

The other representation is to branch out each layer from the 'Actor_Preprocess' block instead of putting them beside each other, hence called the branched netork. Figure 5.6 illustrates this network. We call this the 'val_branch_actions' network.

5.4 Combining Validity & Action branches

The second philosophy in handling validity and actions is what we call the "multiply" method. This approach quite literally multiplies the actions and validity layers together in an element-wise fashion. This is a much harder implementation that attempts to force the actions to be valid instead of expecting it to learn passively. We can this of the agent creating its own knowledge of validity and simply applying it to its predicted actions to only output the valid actions. Note that the output of the multiplication is re-normalised as the actions output is to be a probability distribution.

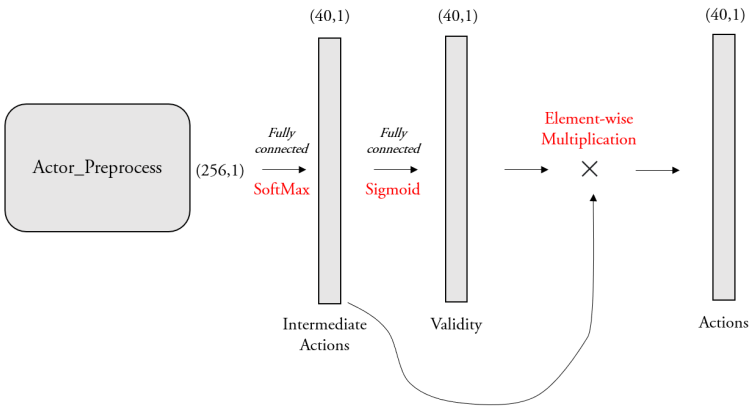


FIGURE 5.7: The 'val_after_actions_multiply' Network

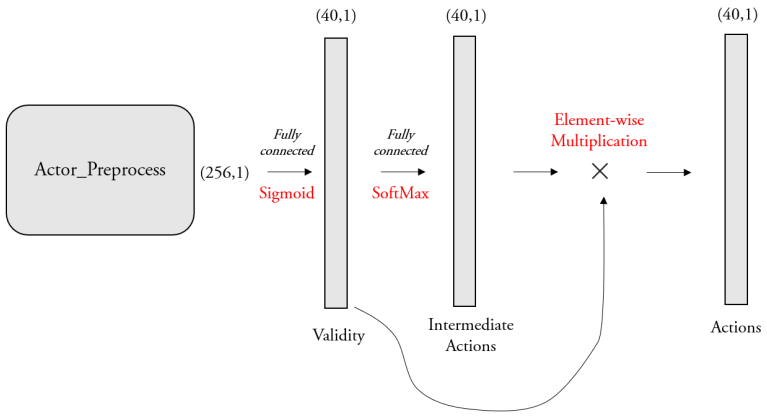


FIGURE 5.8: The 'actions_after_val_multiply' Network

We can apply this multiply method to each of the three architectures described in the previous section. Figures 5.7 and 5.8 illustrate this multiplication and we refer to them as the 'val_after_actions_multiply' & 'actions_after_val_multiply' networks.

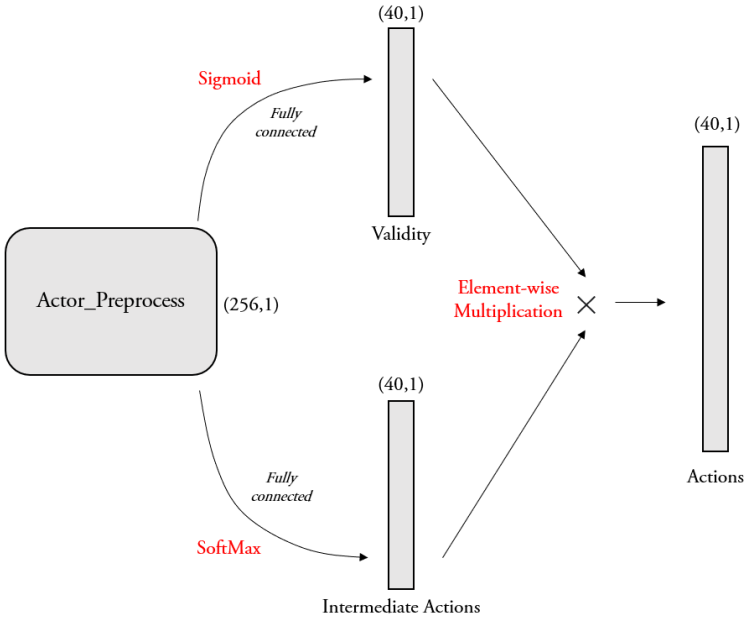


FIGURE 5.9: The 'val_branch_actions_multiply' Network

Finally, we do the same with the 'val_branch_actions' network and multiply the two layers as seen in Figure 5.9. Predictably, we call this the 'val_branch_actions_multiply' network.

5.5 Dual networks for Validity & Actions

Given that we see the task of learning validity and learning good actions as both necessary, we can even split them up to two different networks. In all the earlier models, there is a form of weight sharing where the validity layer and actions layer share the same weights of the 'Actor_Preprocess' block. It is possible that while trying to improve the performance of the validity layer, it may in turn harm the performance of the actions layer and vice-versa. Even though we do not expect it to be the case, there

is no guarantee that both loss functions for the actions and validity will work to reduce each others' losses, that is still an assumption. To test this idea, we came up with the 'dual' network as illustrated in Figure 5.10.

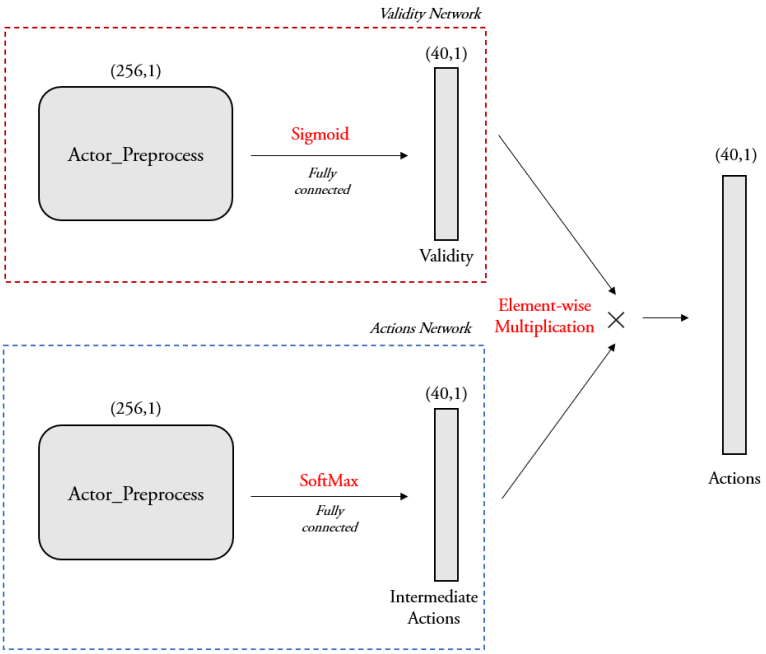


FIGURE 5.10: The 'dual' Network

In the dual network, we have a validity network and an actions network. Each network has their own sets of the 'Actor_Preprocess' block to prevent any form of weight sharing. In the end, the multiply method is once again used to element-wise multiply the validity and actions layers. The idea behind this method is to freeze all the weights of the validity network once a certain acceptable valid rate is reached. For instance, if the agent is choosing valid actions 95% of the time, we can freeze the validity network weights and let the actions network continue training to improve the intermediate actions layer to take better actions. The hope is that the network training will then become less computational and we can have a knowledge of the valid mask that we can fix as its weights are no longer chared with the actions layer.

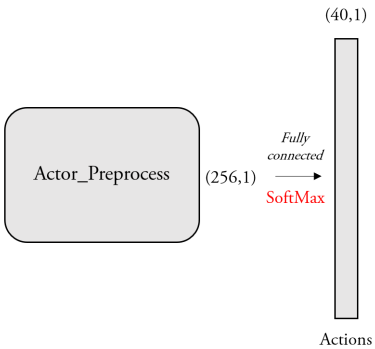


FIGURE 5.11: The 'actions_only' Network

Finally, just to see what happens if we don't bring validity into the picture, we also have the 'actions_only' network that has nothing to do with validity. The network is shown in Figure 5.11. Testing this model will allow us to verify if learning validity does indeed allow the agent to learn better moves faster (irregardless of whether the moves are valid).

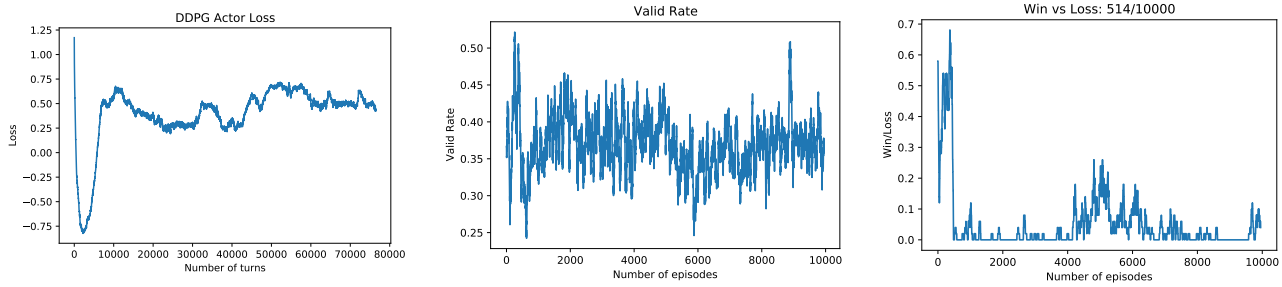
5.6 Training Process

To train the agent to play Onitama, it will face against a computational opponent. It is too time consuming and impractical for a human to sit & play with the agent to improve it. Instead, we let it play with a very powerful AI, the Minimax. We adopt a curriculum approach where its starts by training against Minimax depth 1. If it can beat the Minimax in at least 60 of the last 100 games (during training), the Minimax will advance to the next depth and this process can repeat for as long as necessary. The 60% is simply a hyperparameter that can be adjusted if necessary.

We will conduct a simple hyperparameter tuning and then train each one of the 8 architectures for 10,000 episodes each. Out of the 8 architectures, we let the different models play with each other and we use a ELO rating system to deterimine the strongest agent. This will also include a random AI to see how it fares. Based on that, we will train the strongest AI for a longer period of 100,000 episodes. We will also test the best architecture on a random AI and sees how it performs. To add some noise for exploration during the training process, the agent chooses the action, not with a greedy behaviour policy, but by using the actions layer probability output and using those probabilities to decide on the action. Hence, even if the agent gives a certain action a value of 0.8, it has a 20% chance of not being selected. This probabilistic behaviour policy contributes to the exploration aspect of this training process. Finally, note that all plots shown with the horizontal axis as either "turn" or "episodes" uses the moving average of the last 50 turns and episodes respectively for a slight smoothing effect.

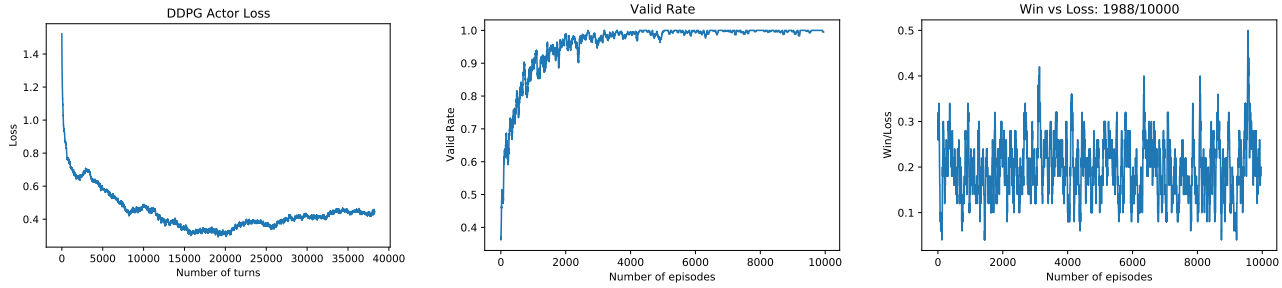
6 Results

6.1 Preliminary Model Selection



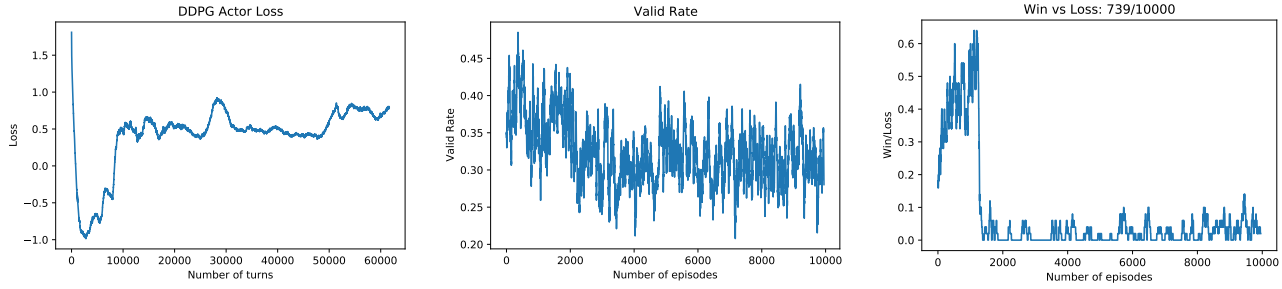
(A) Actor loss (B) Valid rate (C) Win rate

FIGURE 6.1: Metrics for "actions_after_val"



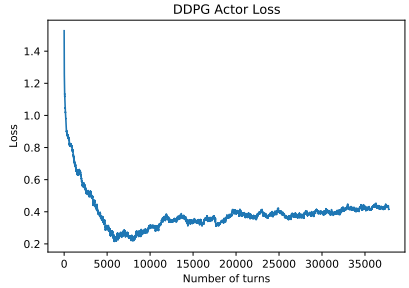
(A) Actor loss (B) Valid rate (C) Win rate

FIGURE 6.2: Metrics for "actions_after_val_multiply"

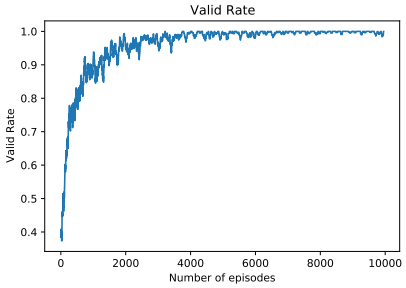


(A) Actor loss (B) Valid rate (C) Win rate

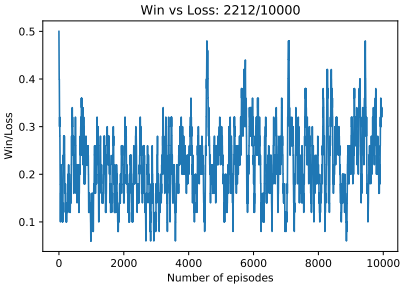
FIGURE 6.3: Metrics for "val_after_actions"



(A) Actor loss



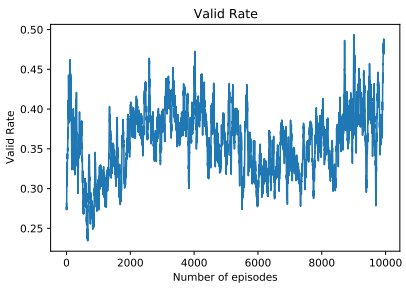
(B) Valid rate



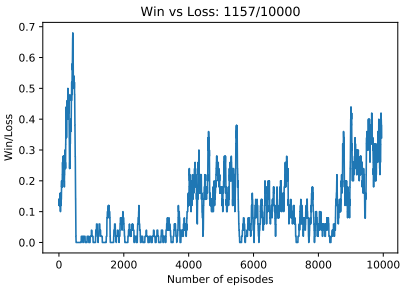
(C) Win rate

FIGURE 6.4: Metrics for "val_after_actions_multiply"

(A) Actor loss

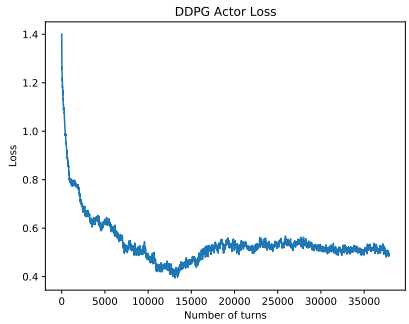


(B) Valid rate

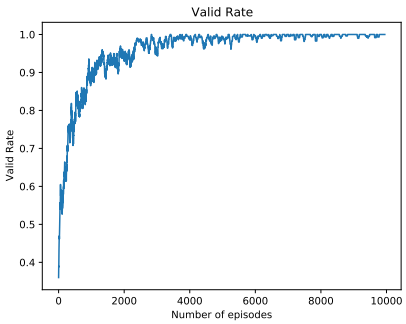


(C) Win rate

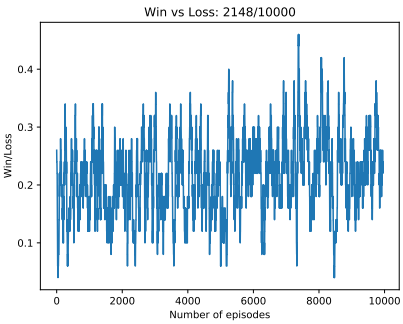
FIGURE 6.5: Metrics for "val_branch_actions"



(A) Actor loss

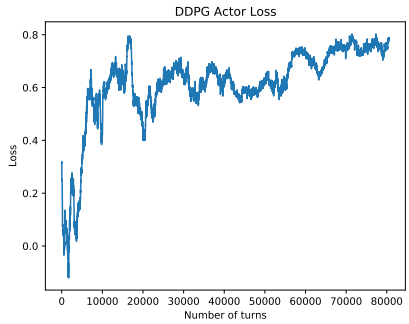


(B) Valid rate

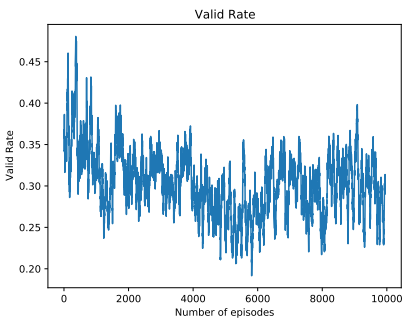


(C) Win rate

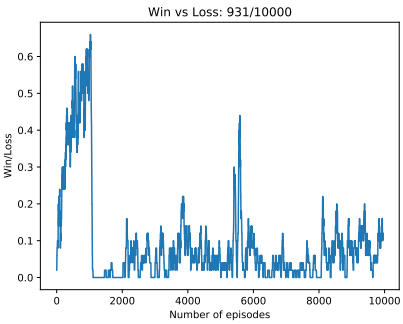
FIGURE 6.6: Metrics for "val_branch_actions_multiply"



(A) Actor loss

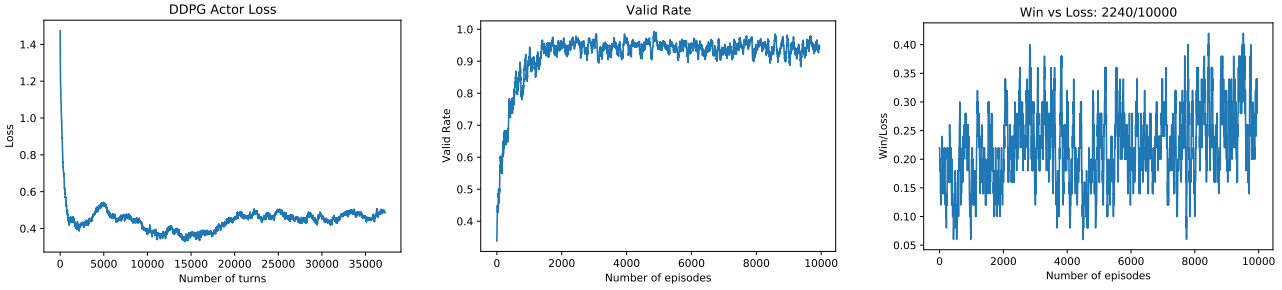


(B) Valid rate



(C) Win rate

FIGURE 6.7: Metrics for "actions_only"



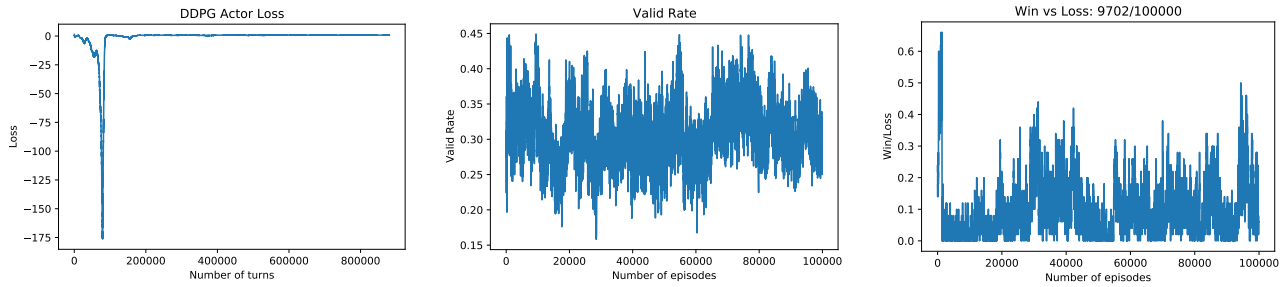
(A) Actor loss (B) Valid rate (C) Win rate

FIGURE 6.8: Metrics for "dual"

From the stated eight models devised, a preliminary model selection test was performed with the same hyperparameters for 10000 episodes in order to identify the best performing model to be trained for an extended period of time. In the scenario where the model achieves a greater win rate than the desired win rate of 0.6, the Minimax depth would be boosted by 1. Figure 5.5b to 6.8 highlights the results of the preliminary based on the following metrics: actor loss, valid rate, win rate, from which are obtained based on a moving average of 50. By examining the stated figures, one observation with regards to the models would be the fact the models with the element-wise multiplication between the validity output and the policy output (e.g. "dual", "actions_after_val_multiply") generally have a valid rate that quickly rises and plateaus at 1, implying that model has generally learned how to take valid moves. However, despite having learned a sufficiently good validity mask, the models struggles in general against the Minimax algorithm at depth 1 with typical fluctuating win rates between 0.1 and 0.5, but never above 0.6.

On the other hand, for models without the element-wise multiplication between the validity output and the policy output (e.g. "actions_only", "actions_after_val"), it can observed that the models in general learned to defeat Minimax at depth 1 reasonably quickly. The instance of the boost in Minimax depth to 2 is readily apparent in the sharp dip in win rate of the models as shown in Figure 6.5c for example. However, upon reaching Minimax at depth 2, it can be observed that there is a significant dip in the win rate of the models given the tougher Minimax at depth 2. This is corroborated with the general increase in actor losses for the models upon reaching facing Minimax at depth 2. Nevertheless, despite the models finding success against Minimmax at depth 1, the valid rates of the stated models are generally poor, typically hovering between the range of 0.1 and 0.65.

6.2 Extended training



(A) Actor loss (B) Valid rate (C) Win rate

FIGURE 6.9: Metrics for "val_branch_actions" for extended training

After examining the metrics of the eight models, the team agreed came to the conclusion that the "val_branch_actions" is the model that showed the most promise as the valid rate and win rate from Figure 6.5b and 6.5c appears to be on a steady upward trend during $80000-100000$ number of turns as well as a general upward trend for both the valid rate and the win rate. Hence, the "val_branch_actions" model was trained for 100000 episodes and the results are as shown in Figure 6.9. It can be observed that unfortunately, despite being given 10 times the number of training episodes from the preliminary model selection test, the model was unable to reach the desired win loss ratio of 0.6 against the depth 2 Minimax, coming close with a win rate of approximately 0.5 as shown in Figure 6.9c. Furthermore, it can be observed from Figure 6.9b that the valid rate does not appear to be improving, fluctuating erratically between the range of $0.2-0.45$ approximately. Given the results, it is apparent that the model is struggling against Minimax at depth 2 despite doing well against Minimax at depth 1. As possible explanation for the model's struggle would be the fact that the disparity between in strength between depth 2 and 1 for Minimax is too large a jump in difficulty for the model to handle. It can be seen from the models that managed to face Minimax at depth 2 in the preliminary model selection as well as in Figure 6.9c that the win lost rate dips extremely sharply to almost 0 upon the increment of the depth of Minimax. Hence, it can be inferred that a more gradual learning curriculum in terms of difficult may significantly help the model to learn better overtime. Given that the depth size of Minimax are integer increments, other alternatives can be such as the Monte Carlo Tree Search (MCTS) algorithm can be considered to be used as the competing AI in the learning curriculum instead of Minimax.

6.3 ELO Round Robin Tournament

Using the 8 architecture models trained fro 10,000 episodes, we pit them against each other in a round-robin style tournament. We also include the random AI as part of this round-robin. We define one round-robin as the set of matches where each one of the 9 models pit against one another once, just like the style used in sporting tournaments. Hence in one round robin, there are a total of $\binom{9}{2} = 36$ games and each model plays a total of 8 games. For our tournament simulation, we simulate a total of 2500 round robins. Hence there are a total of 90,000 games and each model plays a total of 20,000 games. Any two of the models will hence face each other 2500 times. To add another dimension

of determining the best AI, we also record the ELO scores (used famously in games like chess) as a way to quantify the performance of each model. We also record the overall win rates and pairwise comparison on the net number of wins and average number of turns. To describe the ELO system, every model starts with an ELO score of 1000. When any two players play a match, the result of the match affects the ELO score as shown in Equations 6.1 and 6.2.

$$\begin{aligned} E_A &= \frac{1}{1 + 10^{\frac{R_B - R_A}{400}}} \\ E_B &= \frac{1}{1 + 10^{\frac{R_A - R_B}{400}}} \end{aligned} \tag{6.1}$$

$$\begin{aligned} R'_A &= R_A + K(S_A - E_A) \\ R'_B &= R_B + K(S_B - E_B) \end{aligned} \tag{6.2}$$

Equation 6.1 calculates the expected value of player A and B winning. If player A wins, $S_A = 1$ & $S_B = 0$, but if player B wins, $S_A = 0$ & $S_B = 1$. IF both players reach a draw, $S_A = S_B = 0.5$. Then Equation 6.2 determines the update based on a scaling factor K. For this project, we follow a commonly K value of 32. To determine a draw in a Onitama, if the number of turns reach 200 and there is no winner, then we consider the game to be a draw. This also helps to prevent unreasonably long games from happening. Figure 6.10 shows the plot of the ELO scores of all 9 models during the 2500 round robins.

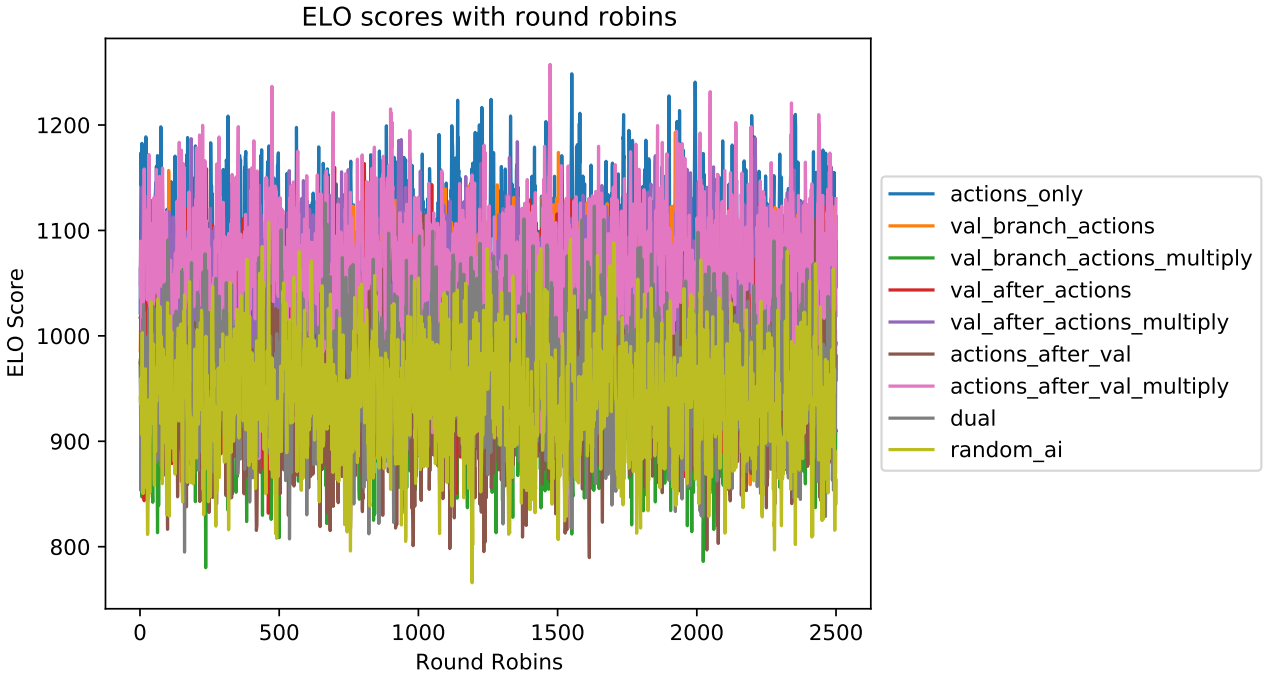


FIGURE 6.10: ELO scores during 2500 round robins of all 9 models

The general trend of each model's ELO does not diverge, converge or change with time, indicating that the ELO scores have stabilised to their bands. There are a lot of fluctuations for each model

and the gap between each model is not very high. This signals that all of the models are relatively competitive with each other and there is no big obvious winner or loser. Nonetheless, the yellow band at the lower half shows that the random AI generally performs poorly. The pink and blue regions at the top are the 'actions_after_val_multiply' and 'actions_only' architectures and they seem to perform the best throughout the entire duration of the round robins. In fact, Figure 6.11 confirms this as the 'actions_after_val_multiply' model ends the tournament with the highest ELO score of 1130, while the random AI ends with the lowest ELO score of 862.

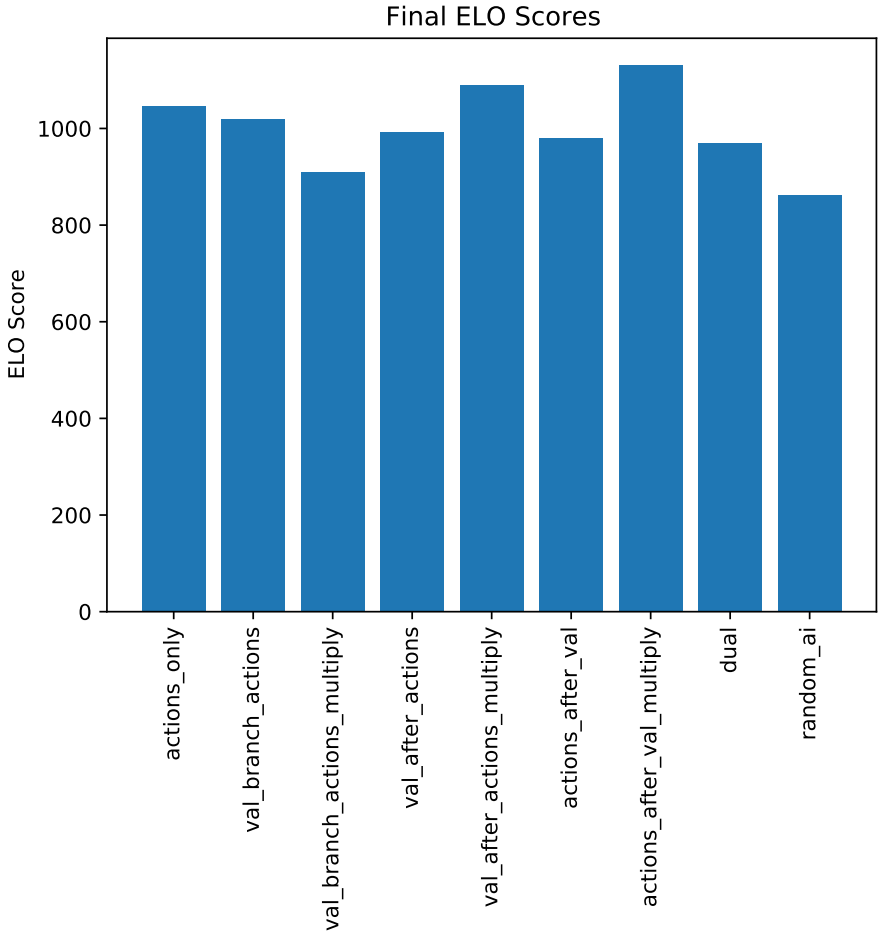


FIGURE 6.11: Final ELO scores of all 9 models at the end of 2500 round robins

Figure 6.12 illustrates the win rate after the tournament for each model. Interestingly the highest win rate is the 'actions_only' model, instead of the 'actions_after_val_multiply' model. This would seem surprising as the 'actions_only' model does not have the highest final ELO score, in fact it is only the third highest. This has to do with the way the ELO score is calculated. The ELO increases quickly when one wins against a much stronger opponent and drops quickly when losing to a much weaker opponent. Likewise, beating weaker opponents non-stop may increase one's win rate, but not their ELO significantly. What is likely happening is that the 'actions_only' model is beating weaker opponents at a much higher rate than the 'actions_after_val_multiply' model, but the the 'actions_after_val_multiply' model still triumphs over the 'actions_only' model. Hence, the

'actions_only' model may win more often, but it is not with stronger opponents, hence its ELO may not reflect this.

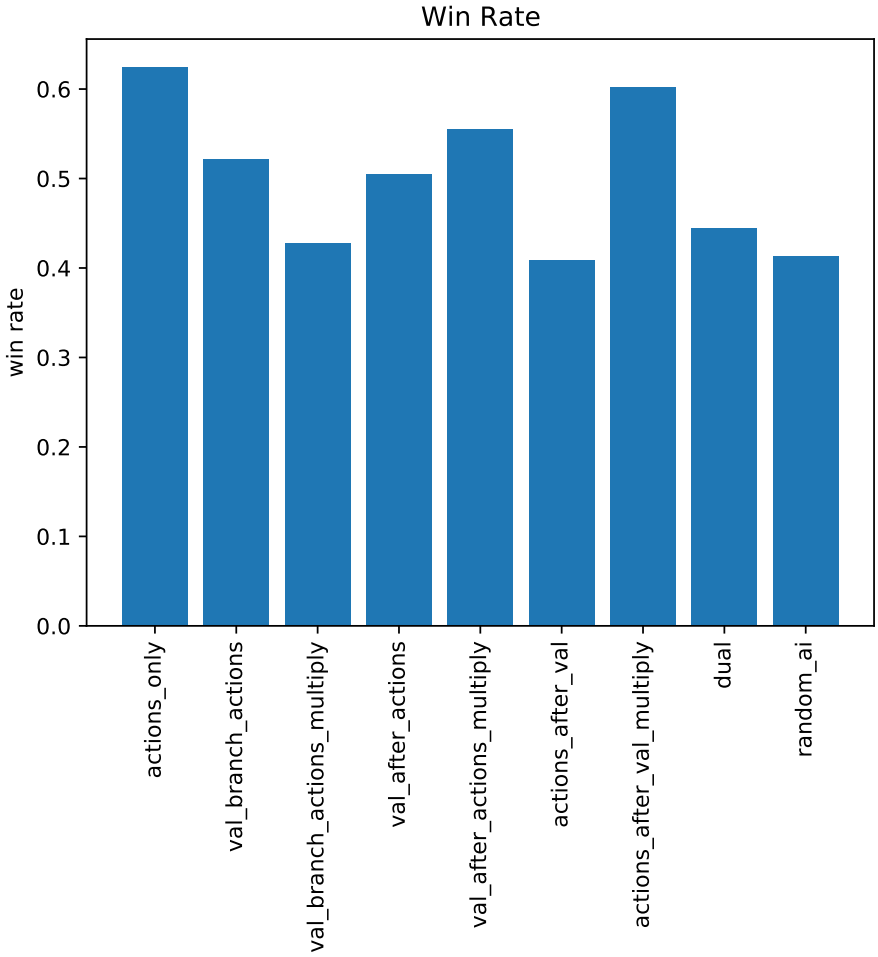


FIGURE 6.12: Final win rates of all 9 models

The pairwise net win matrix in Figure 6.13 actually shows this very case. The value in the cell is the net number of wins of the model in the x axis against the model in the y axis. Hence if the cell value is 250, the model in the x axis has a net win of 250 games over the model on the vertical axis. The maximum and minimum possible values are +2500 and -2500 as 2500 round robins were simulated. The matrix shows that the 'actions_only' model wins extremely often against the 'actions_after_val' and 'val_after_actions_multiply' models (+1234 +1184 respectively) while the 'actions_after_val_multiply' model has net wins over those same models of +428 & +270 (note that the signs are flipped as we are reading from the y axis perspective) respectively. However, on closer inspection, the 'actions_after_val_multiply' model beats the 'actions_only' model by 304 net games won. In fact, the 'actions_after_val_multiply' model beat every single AI. This example shows why win rates are not enough to revel the full picture and hence the need for this ELO scoring system. As it beats all other models, we consider the 'actions_after_val_multiply' model to be the best performing when pit against all the other agents and random AI. This is despite the fact that the 'val_branch_actions' model performed with the most promise against the Minimax opponent for 10,000 episodes. This

shows that simply learning to play the best against the Minimax does not mean it will perform the best overall.

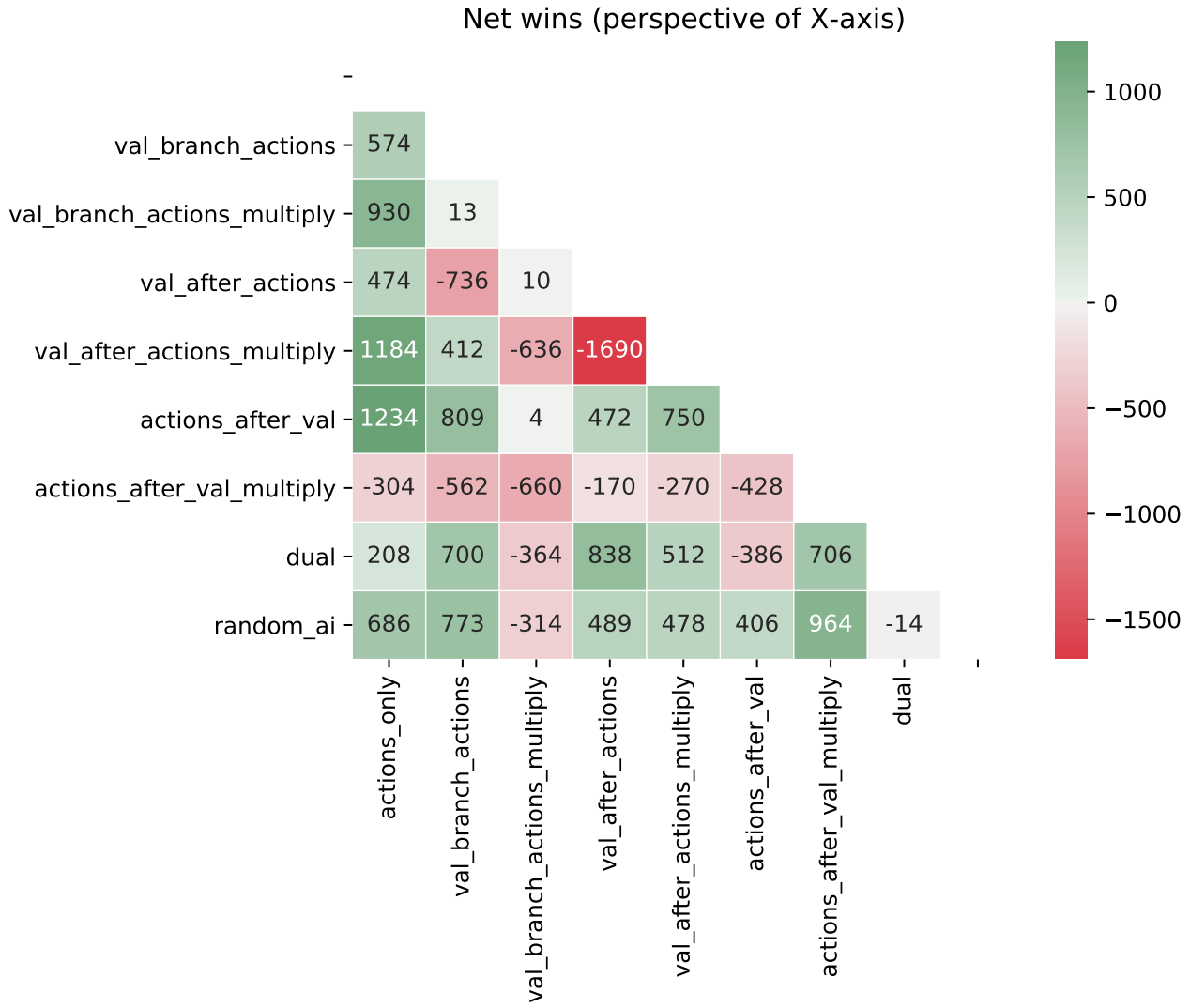


FIGURE 6.13: Pairwise net win matrix of all 9 models

In fact, not learning to play against a random AI does become apparent in this round robin tournament. The 'dual' and 'val_branch_actions_multiply' models both failed to beat the random AI on average (Figure 6.13). This would seem shocking as not being able to beat the random AI would indicate the agents learnt rubbish. However, we think that may not be the case. These 8 models (not including random AI) were trained only with Minimax. This means that the agents have only seen the opponent make good and sensible moves (As far as Minimax depth 2 can go). They have not yet been exposed to the random AI moves, which could be why they did not perform against the random AI. This indicates that 10,000 episodes is too few for the agent to have seen all possible moves made by the opponent AI. Additionally, as there are only 14 valid moves on average (Figure 5.1), the random AI is picking from 14 available moves in each turn. If 2 or 3 of these moves are good moves, then the random AI may have a decent probability of choosing a good move. Hence, we cannot simply assume that the random AI will make bad moves almost all the time.

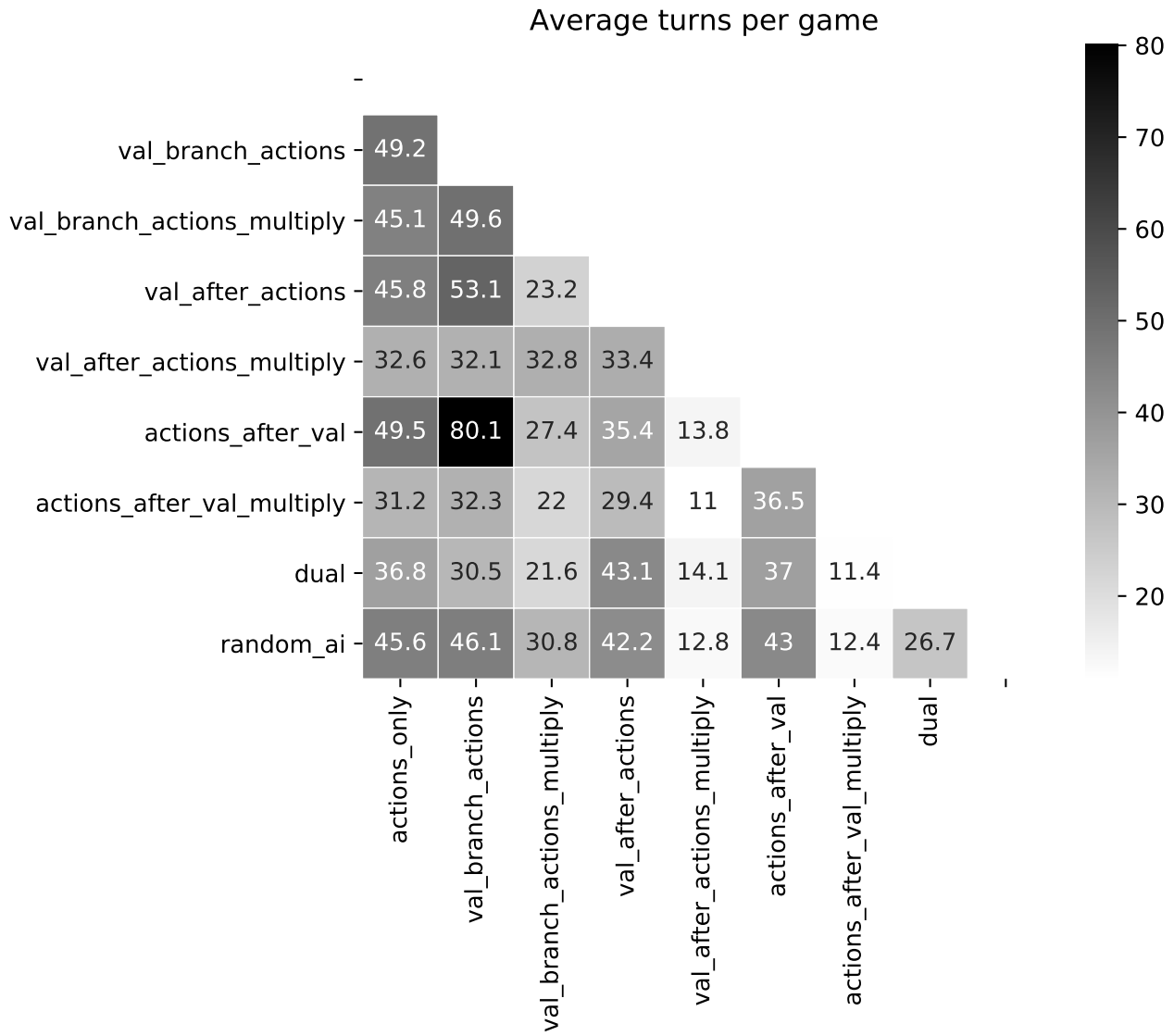


FIGURE 6.14: Pairwise average number of turns for all 9 models

Finally, Figure 6.14 shows the pairwise matrix of the average number of turns taken per game. This was generated to see if there was a correlation between two models having similar strengths leading to a high number of turns. We also wanted to see if one model is significantly better than another, does it correlate to much shorter games? The highest average number of turns is 80.1 with 'actions_after_val' pit against 'val_branch_actions'. However, Figure 6.13 for this pairwise comparison shows that the 'val_branch_actions' model performed much better with a net win of 809 games. Hence there is no correlation there. Figure 6.13 also shows that 'val_after_actions_multiply' had the biggest triumph over 'val_after_actions' with +1690 net games won. Yet the average number of turns is at a moderate 33.4, showing that it is not possible to determine an obvious correlation.

7 Conclusion

7.1 Summary

In conclusion, the team has managed to deploy eight different variants of the actor network for the DDPG algorithm with the intention for the model to learn the ability to select valid moves and building upon that, the ability to select good moves as the model trains. The eight different models were placed on a preliminary model selection test where the models were training using a standard set of hyperparameters for 10000 episodes in order to determine the best performing algorithm to be trained for an extended period of episodes. The model with the most potential, "val_branch_actions" was selected to for extended training with 100000 episodes, where it came close to the desired win rate of 0.6 with 0.5. In addition, the models trained from the preliminary model selection pitted against each other in an ELO Round Robin Tournament in order for the team to better understand the relative differences in strengths of models after being trained with the same set of parameters.

7.2 Limitations

One of the major issue identified on hindsight is the loss function used for the validity loss for the model. The loss function used for all the experiments in the project current uses the mean squared error (MSE) loss function as shown in equation 7.1, where N is the batch size, Y_i is the true value and $\hat{Y}_i$ are the predicted values.

$$L_{MSE}(Y, \hat{Y}) = \frac{1}{N}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{7.1}$$

For the case of the MSE loss function, the gradient of MSE loss with respect to the predicted values, $\frac{\delta L_{MSE}}{\delta \hat{Y}}$, is essentially $2\hat{Y}$. Consider now the binary cross entropy loss function as shown in equation 7.2

$$L_{BCE}(Y, \hat{Y}) = -\frac{1}{N}\sum_{i=1}^{n} Y_i log(\hat{Y}_i) + (1 - Y_i) log(1 - \hat{Y}_i) \tag{7.2}$$

The gradient of BCE loss with respect to the predicted values, $\frac{\delta L_{BCE}}{\delta \hat{Y}}$ is $\frac{Y}{\hat{Y}} + \frac{1-Y}{1-\hat{Y}}$. From the gradients with respect to the predicted values, it can be seen that when the error between $\hat{Y}$ and Y is large (in the context of binaries), the value of the gradient approaches infinity while that is not the case for the MSE loss. Hence, it can be inferred that the BCE loss is more aggressive in ensuring that the each out

7.3 Possible Improvements

The first improve we would naturally suggest is to change the loss function for validity to use a binary cross entropy instead of mean squared error. Additionally, we would like to think of a faster and more efficient way of representing the action space. for instance one could think of a way to represent the actions by the change in location made instead of having different parts of the action space dedicated to specific pawns. This will allow the agent to not think of the pieces strictly and there is no need to individualise the pawns. There is no guarantee however that this method will be any better. One important suggestion that can be made is to design a faster algorithm to generate the valid mask. This may require the board game to always have an understanding of the valid moves everytime a turn is made. This way the training code does not need to constantly check all 40 actions for validity.

Regarding the training itself, one may consider engaging in self play to train. We create two agents and we let them play against one another. Both will improve and challenge the other further, leading to both agents building their performance on one another. Otherwise, once can use a Monte Carlo Tree Search (MCTS) algorithm to act as the opponent AI. The competence of MCTS can be easily tuned and controlled hence we can avoid the huge jumps in competency like that from depth 1 in Minimax to depth 2. We could have also tried other algorithms like Advantage Actor Critic (A2C) or Proximal Policy Optimisation (PPO). Another possible improvement is to speed up the training with a mix of imitation learning. This can be done by saving the replay memory of the opponent and using that replay as part of the agent's replay memory.

Ultimately, there were things we were unable to time due to time constraints, but we learnt a lot from this project nonetheless. We are happy and proud of how much we have learnt and will continue to pursue the applications of reinforcement learning and AI. Finally, we are thankful to Prof Guillaume Sartoretti for his supervision during the duration of this ISM. We conclude our project.

8 Appendix

8.1 Rules of Onitama

This section is taken from the Onitama Rulebook

8.1.1 Setting up

The board is set up so that there is a temple in front of each player. Each Master piece is then placed on the temple of the same colour. Similarly, four pawns are then placed in the same row as the Master piece of the same colour, two on each side of the Master piece. The starting board should look like Figure 8.1.

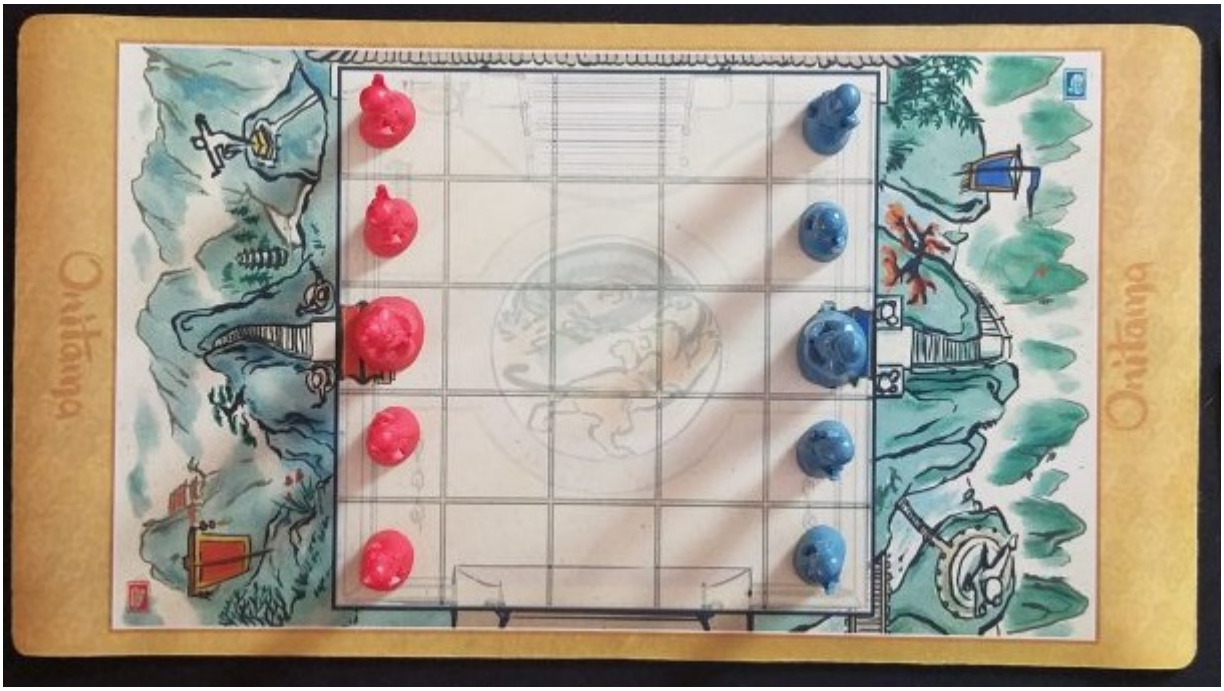


FIGURE 8.1: The starting board of Onitama

The 16 cards are then shuffled. Deal the top two cards, face-up, to one player and deal the next top two cards, also face-up, to the other player. Finally, deal the next cards, face-up, to the side of the board, between the two players. The player whose pieces are the colour of this card, indicated in Figure 8.2, will make the first move. The other 11 cards are set aside and will not be used for the current game.

FIGURE 8.2: The colour of the card (circled). Note that the colour of the moveset itself may not be the colour of the card.

At the end of the setup, the board should look like Figure 8.3:

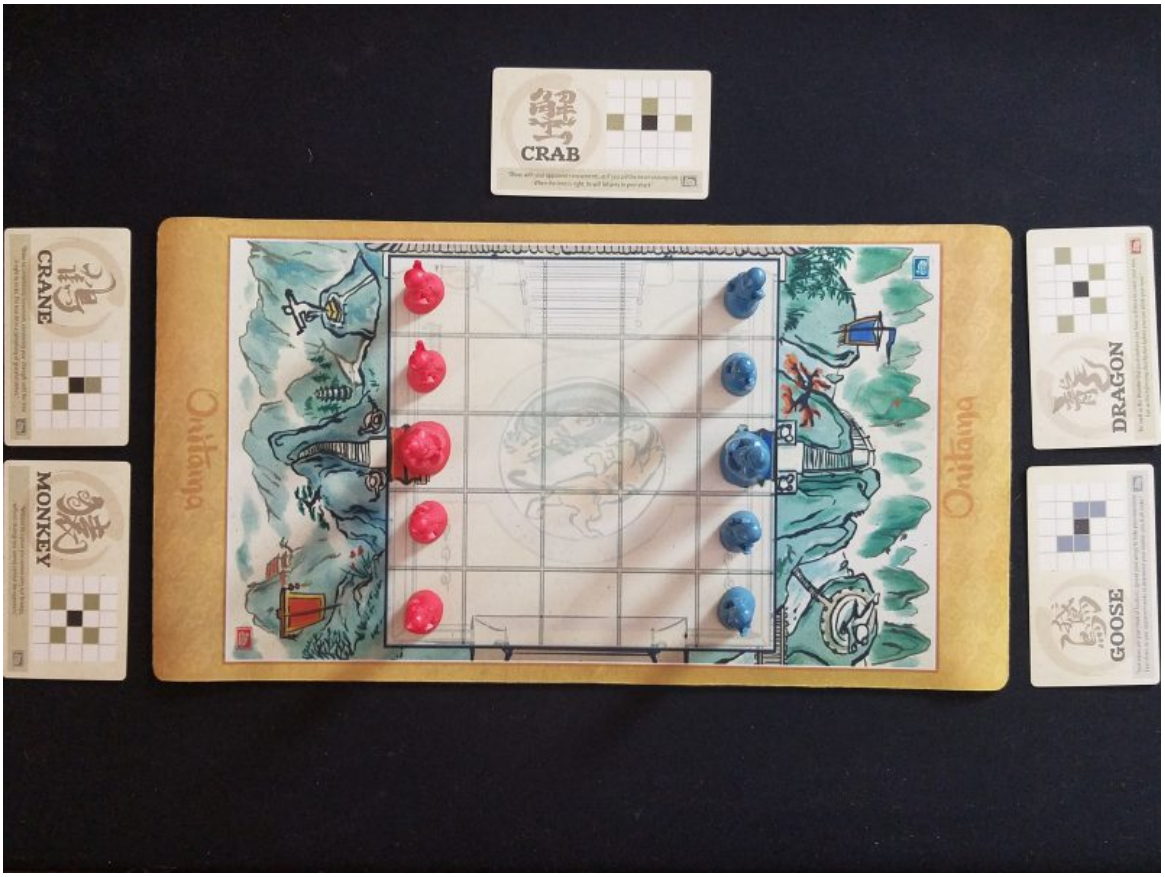


FIGURE 8.3: An example of the Onitama board at the end of the setup, ready to begin a game

8.1.2 Movement

After determining the first player to move, the game can begin. The movement phase can be separated into two parts: pieces movement phase and cards movement phase, which must be resolved in that particular order.

Pieces movement phase

Pieces can move based on the moveset of the cards their owner possess. For example, in the Frog example in Figure 8.2, the player with possession of that card can move their piece two squares to the left, one square diagonally to their top-left, or one square diagonally to their bottom-right.

To make a move, the turn player must declare three things:

- The piece they wish to move
- The card they wish to use
- The move in the card they wish to perform

In determining the validity of the move, three conditions must be fulfilled:

- The piece cannot land outside of the board
- The piece cannot land on a square occupied by another piece of the same colour
- The card chosen must, obviously, be in the turn player's possession

After the turn player declares a valid move, the piece is moved accordingly. If the moved piece lands on a square occupied by another piece of the opposite colour, that piece is removed and the moved piece takes its place on the board. If any of the victory conditions (see below) is fulfilled at this point, the game ends. Otherwise, this is the end of the pieces movement phase and the cards movement phase begins.

Cards movement phase

If the game does not end after the pieces movement phase, the turn player swaps the card on the side of the board with the card they used for moving the piece. The turn ends and the opposing player can start their turn.

8.1.3 Winning

There are two winning conditions. The first is to eliminate the opposing Master piece. This can be done by moving either your own Master or pawn piece to the square that the opposing Master piece is currently occupying, thereby eliminating him. The second win condition is to move your own Master piece to the square that the opponent's Master piece started on (their temple). If either of these conditions occur, the game ends and the player who achieved either (or both) of these objectives is declared the winner.

Bibliography

[1] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.