Wild Goose Chase

Tong Hui Kang, Wong Tin Kit, Tey Shi Ying, Mario Josephan

8 August 2021

1 Abstract

In this report, we describe our experience participating in Hungry Geese, a Kaggle Simulation competition. In each game, four geese programmed by the teams will compete with each other. Similar to the Snake game made popular in Nokia phones, the geese extends its length by eating food and dies when its head bumps into any of the snakes. The objective of each goose is to survive 200 steps with the longest length.

Our approach to the competition is to improve on a publicly shared agent AlphaGeese. AlphaGeese follows the approach of AlphaGo, which uses a neural network model with Monte Carlo Tree Search. We have tried alternate model architectures for the neural network model, but none has been proven to be better than the neural network model published on Kaggle. We have finetuned the neural network model by training it against other top rule-based agents. We have also experimented with different ways of improving the Monte Carlo Tree Search.

By using fine-tuned agents and modifying the Monte Carlo Tree Search, we have produced agents that have higher mean rating than the original AlphaGeese agents. Tentatively, we have received a 70th with a rating of 1055.3. We have received a Kaggle Bronze medal for reaching the top 100.

Contents

2 Task Description

Hungry Geese is a Kaggle simulation game, where each participant submit agents that compete with other agents. The rating of the team is the rating of its top agent.

In Hungry Geese, the main objective of each agent, or geese, is to survive. Similar to the Snake game popularised by Nokia phones, eating food will extend the length of the geese, and bumping to any agent will result in death.

2.1 Game Rules

The game board is of size 11 by 7. The game board wraps at the horizontal and vertical side, in other words, travelling right of a rightmost cell will return you to the leftmost cell in the same row. An example of the game board is shown in Figure 1.

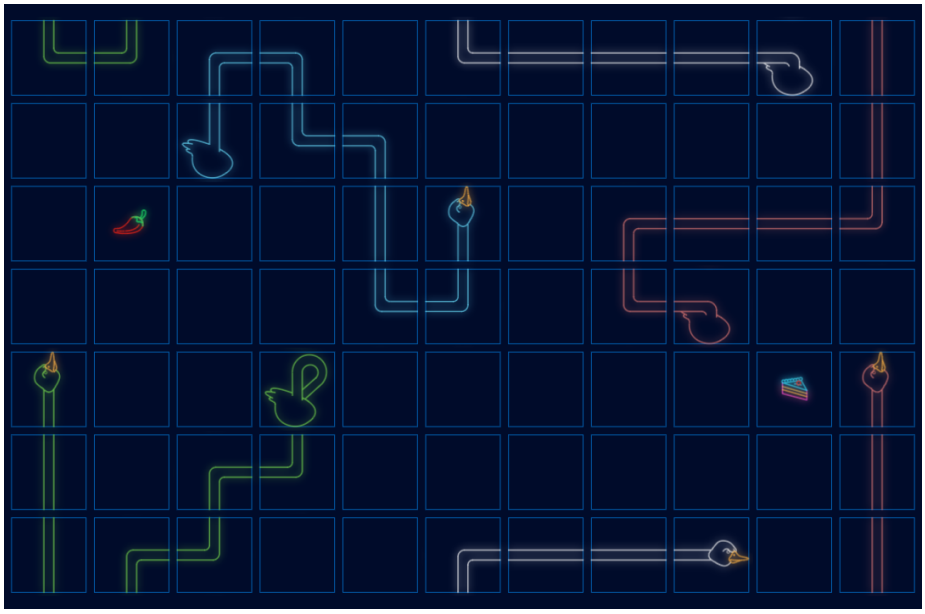

Figure 1: Gameplay Screenshot

Each goose will start with the length of one and they will spawn at a random location on the board. The game will run for a maximum of 200 turns or there is only one goose alive.

In each turn, each goose can choose to move up, down, left or right, relative to the orientation of the board. The goose is not allowed to take the opposite of its previous action even if the length of the goose is below three.

When the goose eats food, its length will increase by one. There is always two pieces of food on the board, and once a food is eaten, another food will spawn at a random unoccupied cell. At every 40 turns, the each goose will lose 1 segment of their body.

The primary objective of each goose is to survive to turn 200. Each goose is primarily ranked by the number of turns it survives. If two or more geese survive to turn 200, or die at the same time, the geese with the longer length will get a higher rank. The rank from each game will update the rating of the agent.

2.2 Competition Rules

The Kaggle competition started on January 26, 2021 and submission closes on July 21, 2021. Our team joined the competition late, making the first submission on June 23, 2021. Throughout the duration of the competition, each team can submit five agents every day, which will play against other agents. The team's placing in the competition is determined by the rating of the team's highest-rated agent.

Each agent will play against other agents of a similar rating. The magnitude of the change of rating will decrease as more games are played. After the close of the submissions, play between agents will continue to allow the agent rating to converge. The results are scheduled to be finalised on August 10, 2021, which is after the deadline of this report.

3 The Metagame and Our Approach

3.1 The Metagame

Currently, there are two popular high performing agents that people have shared during the competition. They are PubHRL [1] and AlphaGeese [2].

Figure 2, which is produced by a top participating team [3], shows the distribution of scores of submitted agents at the close of the submissions. The vertical lines show the required team rating to get a Kaggle Bronze, Silver or Gold medal.

The first peak is heavily contributed by PubHRL [1]. PubHRL was published in February. It has been analysed that the agents shaded in yellow in Figure 2 makes moves exactly suggested by PubHRL, and the agents make up to 17.5% of the submissions [3]. PubHRL is a neural network model that suggest which move to take. We will elaborate on PubHRL in Section 4.

The second peak is suspected to be heavily contributed by AlphaGeese [2]. AlphaGeese was published in April. AlphaGeese model uses PubHRL model inferences but it improves the performance by adding Monte Carlo Tree Search

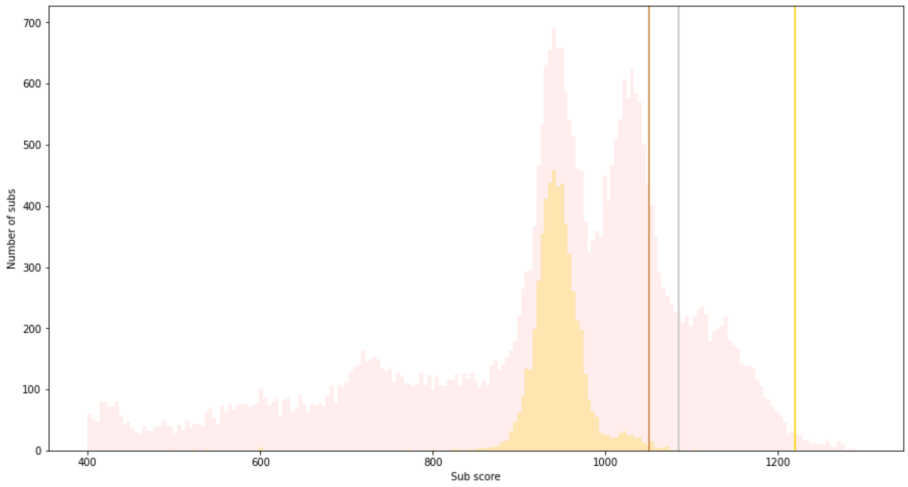

Figure 2: Distribution of Scores of Submitted Agents

(MCTS) algorithm on top of it. This approach is similar to AlphaGo [4], and we will elaborate this in Section 5.

3.2 Our Approach

When we entered the competition, we have decided to improve on the best-performing approach that has been published, which is AlphaGeese. There are two aspects we can improve on AlphaGeese. One aspect is to improve the underlying neural network model in terms of performance or time, which we will explore in Section 4. The neural network model can be improved by changing the model architecture, which we will describe in Subsection 4.4 or changing agents it is being trained against, which we will describe in Subsection 4.5. The other aspect is to improve how Monte Carlo Tree Search is done, which we will explore in Section 5.

Then in Section 6, we describe and analyse the results. We will also show how to set up a GUI to play against specified agents in Section 8.

4 Neural Network Model

In this section, we first explain the input and structure of PubHRL, the neural network AlphaGeese is built upon. We explain how we attempted to improve the neural network model by changing the model architecture and the agents we train against. Finally we analyse some of the weakness of the neural network models.

4.1 PubHRL Neural Network Model

In this subsection, we explain the input and structure of PubHRL model [1] to provide context to the improvements we attempt to make. The neural network model takes in a matrix of values, and returns four action-values and one state-value.

4.1.1 Processing the Game State into the Model Input

The PubHRL takes an input of a binary matrix of size (17,11,7). Each cell in the 11 $\times$ 7 game board is represented by 17 binary variables. Each "slice" of the matrix input encodes a different part of the information from the game board.

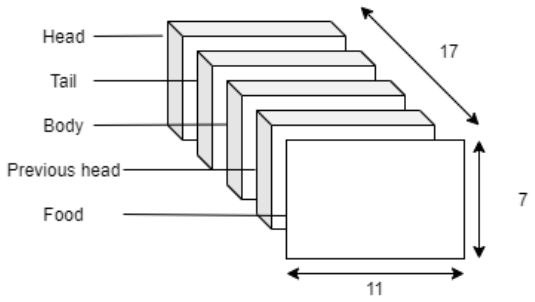

Figure 3: Input block

- The first four slices encode the position of the head, with one slice for each goose. The value in the matrix is 1 if the head belongs to the geese. Each slice is one-hot as there is at most one non-zero value. If the goose is dead, the slice will have values of all zeroes.
- The next four slices encode the position of the tail in a similar manner. Each slice of the tail encoding is one-hot.
- The next four slices encode the position of the body. The value in the matrix is 1 if the body belongs to the geese. These slices are not one-hot.
- The next four slices encode the position of the previous head position.
- The last slice encode the position of the food.

There is sufficient information encoded in the input. The model can infer the length and shape of the goose from the position of the head, body and tail. Likewise, the direction of the previous move can be deduced from the difference between the position of the current head and the previous head. The input also takes into account the position of the other goose as well so that the goose can avoid colliding with them.

4.1.2 The Torus Block

The Torus Block consists of a two-dimensional convolution, ReLU and residual connections. A diagram of the architecture of the Torus Block can be seen as follows.

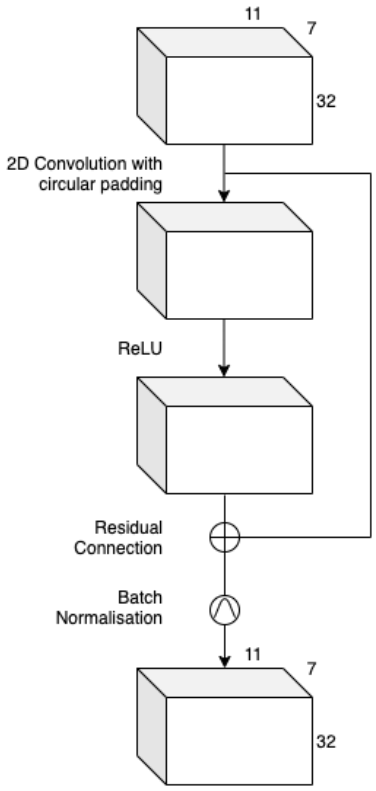

Figure 4: Torus Block Architecture

The two-dimensional convolution uses filter size of 3×3 and has circular padding of size one to model the wrapping game board. The shape of the variables before and after the convolution is the same.

The output of the convolution is passed into ReLU. A non-linear function is used so that the neural can learn and produce a nonlinear decision boundary with non-linear combinations of the weight and inputs.

The output of ReLU is added with the input of the Torus Block. This summation is a residual connection and allows the model to address the vanishing gradient problem. In training, batch normalisation is also applied to improve gradient flow.

4.1.3 PubHRL Model Architecture

The PubHRL model architecture is made up of Torus Blocks and fully connected layers.

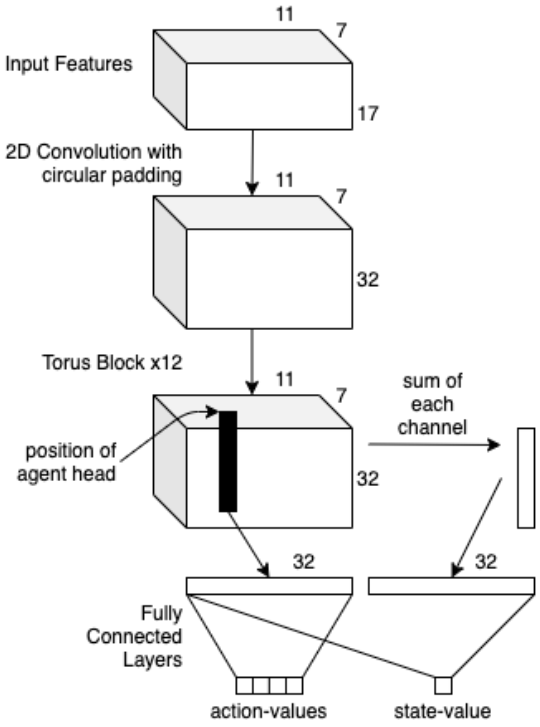

Figure 5: PubHRL Architecture

One convolution with circular padding transforms the input of size (17,11,7) to intermediate features of size (32,11,7). Then 12 Torus Blocks are applied on the intermediate features, which does not change the variable size of (32,11,7).

The values at the head position of the player's agent are extracted from the matrix, forming an array of size (1,32). The values at each of the 32 channels are summed, forming another array of size (1,32).

A fully connected layer transforms the values of the first array to the action-values. Another fully connected layer transforms the concatenation of the first and second array to the state-values. The fully connected layers are configured without bias.

This is how the model produces four action-values and one state-value. The model can be used directly to make inferences on which move to make, just by taking the action with the highest action-value.

4.2 Dataset, Loss, and Training Architecture

As this is a reinforcement learning problem, dataset and loss functions are defined differently compared to supervised training. The objective of the training is to train the model to produce a good estimate of the policy-value and state-value.

The state-value $V^{\pi}(s)$ is the expected discounted return for a given state s following policy π that is assumed to be optimal. The action-value $Q^{\pi}(s, a)$ is the expected discounted return for taking action a from a given state s following policy π that is assumed to be optimal [5].

As the number of possible game states is large, it is not possible to calculate the exact expectation for every state-value and action-value. Therefore state-value and action-value is approximated with a function, and the function is a neural network. The neural network is trained on squared loss [5] between the predicted values and estimated values generated from the simulating the game.

One way to estimate the action-value and state-value without enumerating all possible states is to randomly simulate the actions until it reaches a terminal node in each complete episode. The payoff associated with the terminal node is backpropagated, thereby providing action-value and state-value for training. This method is known as the Monte Carlo (MC) method [6].

It is possible to learn from incomplete episodes with Temporal Difference (TD) learning [6]. TD learning methods update targets with value estimates rather than solely relying on actual rewards from complete episodes as in MC methods.

It is possible to scale up training to achieve a very high throughput with the Importance Weighted Actor-Learner Architecture (IMPALA) framework [7] [8]. In the IMPALA framework, actors generate experience trajectories in parallel, while the learner optimizes the neural network parameters using generated experience. Actors update their parameters from the learner periodically. Because acting and learning are decoupled, we can generate many episodes with many actors in parallel and train the model more quickly. IMPALA also uses an off-policy gradient algorithm [9] and TD learning to allow the decoupling and learning from past episodes for better sample efficiency and exploration [8].

HandyRL [10] is an implementation of the IMPALA framework. We have used the training hyperparameters shared by PubHRL authors [11].

4.3 Training Setup

The IMPALA framework allows the use of distributed computing resources to allow a high throughput for reinforcement learning training. For this to be possible, we need access to many virtual CPUs and a GPU. Therefore, we have

trained our models on the Google Cloud Platform. Figure 6 shows a screenshot of our training setup.

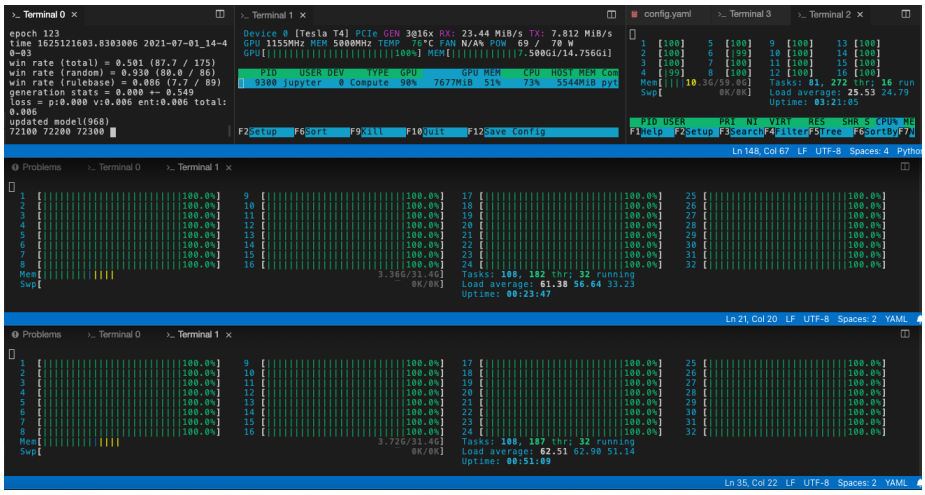

Figure 6: Screenshot of Training Setup

The actors are run on two preemptible `e2-highcpu-32` instances. As the actors do not require much memory, a `highcpu` instance subclass is chosen for some cost savings. As training is decentralised, the instances are also preemptible to save up to 80% of the cost. The actors cost around US$0.71 per hour.

The learner is run on one `n1-highmem-16` instance with a T4 GPU. We need a instance with high memory to store the large number of episodes for experience replay. The instance will cost around $1.30 per hour [12].

It takes 24 hours to train the PubHRL model with 2.1 million generated episodes over 4200 epochs. The training curve is shown in Figure 7. The $300 Google Cloud welcome credits has helped to alleviate our computation costs.

4.4 Alternate Model Architectures and Hyperparameters

Modifying the model architecture has the potential to improve the model. A model can be considered to have improved if it provides better suggestions on which move to take, or takes a shorter time to make an inference.

There is a performance-cost trade-off when we design the model architecture. A larger model may have a better performance, however, a larger model takes a longer time for inference. The larger model with a better performance may not produce a better agent as the depth of the tree search is shallower.

This section documents the different model architectures that we have tried. Figure 7 shows the training curve against the default rule-based agent by Kaggle [13]. The objective function value is defined to be 1.00 if the model gets a first

place, 0.67, 0.33 and 0.00 if the model gets second, third or fourth place. The score plotted is the average of the 500 games simulated per epoch.

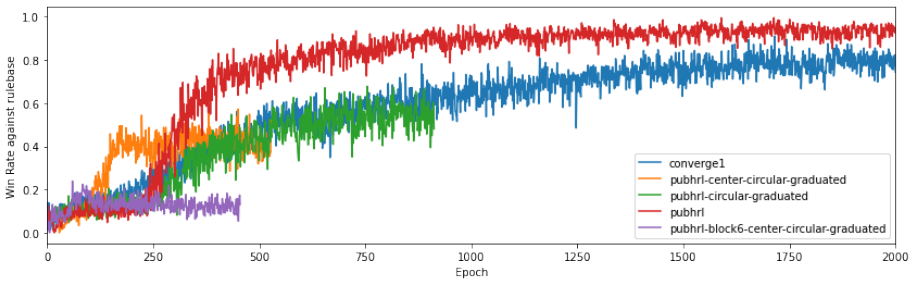

Figure 7: Training Trajectories of Various Models

`pubhrl`
We attempt to reproduce the training with the original model architecture shared by HandyRL. We use the training hyperparameters shared by the authors [11]. Notably, the performance of the model only start to significantly improve after the 250th epoch. This suggests the volatile nature of training reinforcement learning models.

`pubhrl-circular-graduated`
In the input to the original model, the orientation of the body is not well-defined. The main change in this variant is to modify the input to better define the orientation of the body. Instead of having binary input for the four body slices, we used `2/math.log(i,2)` where `i` is 4 if it is the tail, and an increment for every unit of body from the tail. The tail will have a value of 1, and it decreases asymptotically to zero further away from down the tail. The intention is to encode the arrangement of the body in the hopes that the model will learn the arrangement. A high value implies that the location will soon be unoccupied.

We also changed the code that builds the convolutional layers, even though the behaviour is the same. The original PubHRL code makes copies of the array to pad feature matrix. We avoided the copying by using PyTorch `nn.Conv2d` with the initialisation parameter `padding_mode` as `"circular"`.

`pubhrl-center-circular-graduated`
In addition to the previous model, we transpose the input such that the head is always at the coordinate `(0,0)`. While we a steep improvement in training objective score at the at around the 100th epoch, the performance plateaued.

`pubhrl-block6-center-circular-graduated`
In addiditon to the previous model, we attempt to train a smaller model and see if is possible to achieve comparable performance. The original model architecture has 12 layers of Torus Blocks, we have reduced to 6 in this experiment.

However, we have given up on the training as we do not see the model performance improve at the 400th epoch.

`converge-1`
This model architecture is inspired by the ResNet architecture [14]. After every residual block the the ResNet, a pooling operation is applied which reduces the dimension while increasing the number of channels. This allows more abstract features to be learnt. While the training trajectory is promising, it does not perform better at the training objective.

The alternate model architectures we have experimented did not produce superior performance on the training objective compared to the original model shared by the authors. Therefore, in our future experiments, we continued used the original architecture. After the competition, one of the top teams reported that drastically different models, such as multi-layer perceptrons and minified versions of ResNet and EfficientNet, have not shown benefit [15].

4.5 Training against Different Agents

Instead of modifying the model architecture, there is an opportunity to improve the performance of the model by changing the agents that the model is trained against. In the Figure 7, the model is trained against the default rule-based agent provided by the repository. In this subsection, we trained the model against better rule-based agents shared by other participants on Kaggle. The training logs and the code to plot the logs are available in an repository described in Section 9.2.

As we use the same model architecture as PubHRL shared by the HandyRL authors, we begin our training from their pretrained parameters.

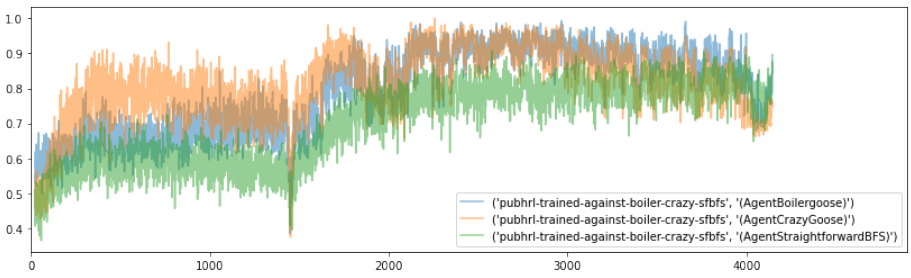

Figure 8: Training Trajectories against three rule based agents

`pubhrl-trained-against-boiler-crazy-sfbfs`
We finetune the PubHRL parameters by training against three agents - namely Boilergoose [16], Crazy Goose [17] and Straightforward BFS [18]. These three agents are selected from because their general good performance against other rule-based agents and against PubHRL [18].

The trajectory of the training objective is plotted in Figure 8. As we use the pretrained parameters, the training objective value starts at around 0.5, rather than zero when trained from scratch. There is an unexplained dip in training objective value at around 1500th epoch, after which the model improved its performance to some peak. However, at the end of training, the training objective value declined.

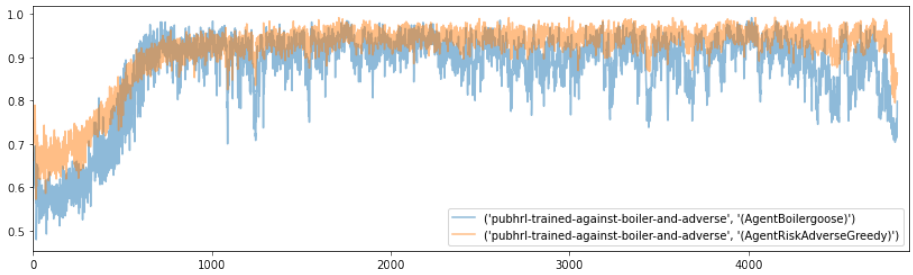

Figure 9: Training Trajectories against two rule based agents

`pubhrl-trained-against-boiler-and-adverse`
We decide the reduce the number of agents that we are training against. We choose to train the model against Boilergoose [16] and Risk Adverse Greedy Goose [19] because of their very different strategies but similarly high performance. We managed to train the model to score at around 0.9 against both models. After 4000 epochs, the training objective value declined. Eventually, we used the model checkpoint at the 3600 epoch.

4.6 Weaknesses of the Neural Network Model

The neural network may produce suggested moves that are certain to be bad. We show two examples from the neural network fine-tuned by us.

4.6.1 Tail Strike

It is usually permissible to enter the location of the tail of another geese, because the location is unoccupied as the other geese moves to the next step. However, if the other geese happens to capture a food in the next step, the location of the tail is not unoccupied. This results in the elimination of the geese. Due to the rarity of such occurrence, the model might not be able to train on this scenario well enough.

Figure 10 shows a scenario of a tail strike. Our agent blue, is considering two feasible moves, left or down.

Direction	Up	Down	Left	Right
Model Inference Probability	0.0000	0.0431	0.9569	0.0000

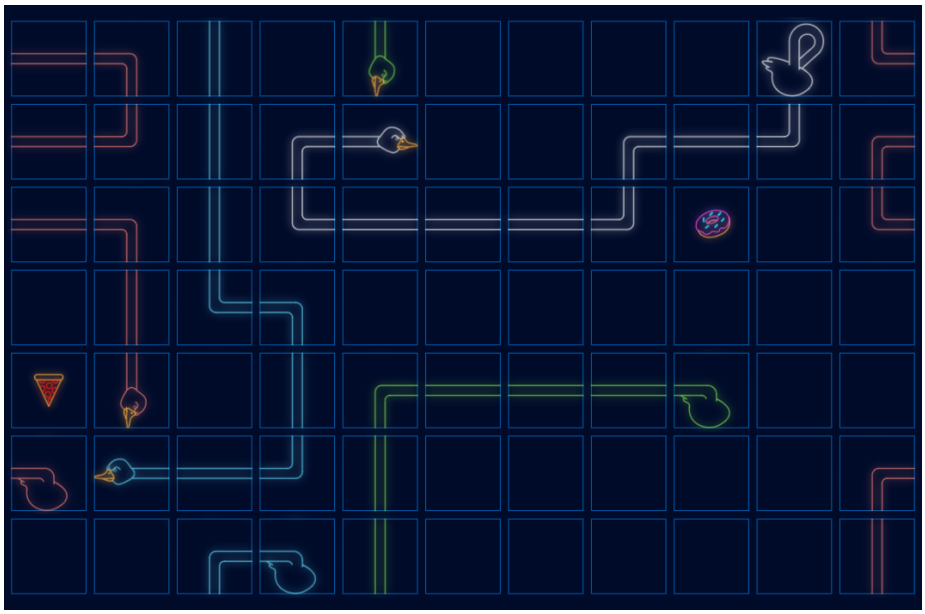

Figure 10: Scenario of Tail Strike (Our agent is blue)

In this scenario, the model make a strong suggestion to step into the location of the tail of probability 0.9569. However, the only move that red agent can make to survive is to go left to eat the food. If we follow the action suggested by the model, our agent will be eliminated.

4.6.2 Length Ignorance

When two heads collide, both geese will be eliminated at the same time. The geese with the longer length will get a higher position, while the geese with the shorter length will be relegated to the lower position. However, surviving geese will be ranked better than the pair of geese which collided head-on.

Figure 11 shows a scenario of a length ignorance. Our agent red, is considering three feasible moves. The agent can move left to capture the food, or move up or down to avoid the food. Our red agent is 10 units long, one unit shorter than our rival which is 11 units long. If we collide with the blue agent over the food, we get the worst possible outcome.

Direction	Up	Down	Left	Right
Model Inference Probability	0.0007	0.0031	0.9962	0.0000

Neural Network models may also make other move suggestions that may result in the worst possible outcome. A top team [15] shared about how they use rule-based oversight to avoid getting into traps and corridors, as well as trapping

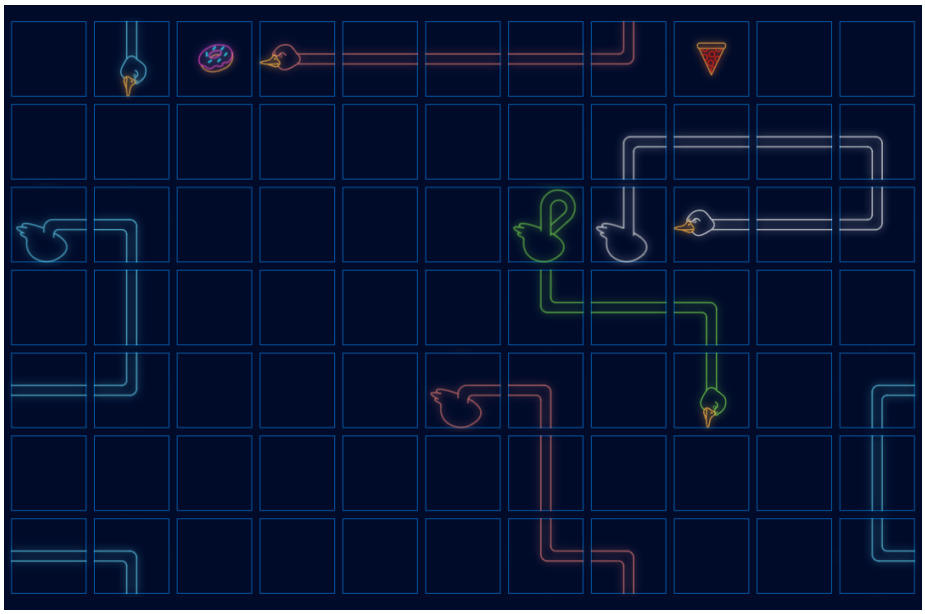

Figure 11: Scenario of Length Ignorance (Our agent is red)

other agents.

5 Monte Carlo Tree Search

As described in subsection 4.6, the neural network model may not always provide the best action-values. This can be addressed by simulating multiple steps of the game to improve the quality of move suggestions.

The simulation is considered a tree search as we consider the possible moves made by each agent simultaneously. Each board state can be considered as a node. A directed edge between two nodes exists $i \to j$ if there is a set of valid actions from each agent that can cause a transition from state i to state j.

As there is a maximum of four agents each making either of three possible moves, the branching factor of the game tree is 81. The branching factor is likely to decrease on average as the game progresses due to lesser geese and more crowded board space.

An exhaustive tree search is not possible due to the number of steps in the game (up to 200) as well as the large branching factor of 81. Therefore the original AlphaGeese agent uses a Monte Carlo tree search (MCTS) as a heuristic tree search algorithm. In the pure MCTS algorithm, random rollouts were made until a terminal state is reached, and what follows is the backpropagation of the payoff in the game tree. However, as it is possible for the game to progress until

step 200, random rollouts are inefficient and ineffective. AlphaGo replaces the random rollouts with a neural network that infer the expected payoff at each non-terminal state [20].

5.1 Algorithm Implemented by AlphaGeese

AlphaGeese [2] follows an implementation of MCTS [21] [22]. The implementation of MCTS tracks the following variables.

- $P(s, a, i)$ The prior probability agent i of taking an action a from a state s according to the neural network. This is the softmaxed value of the action-values inferred by the neural network.
- $N(s, a, i)$ The number of times we explore the action a taken by agent i from a state s when we are searching the tree.
- $Q(s, a, i)$ The expected reward for taking the action a by agent i from a state s. This is initialised with the state-value inferred by the neural network. $Q(s, a, i)$ is the average of the state-values of the explored nodes in its subtree.

The action with the highest upper confidence bound $U(s, a)$ is explored.

$$U(s, a, i) = Q(s, a, i) + c_{cpuct} P(s, a, i) \frac{\sqrt{\sum_b N(s, b, i)}}{1 + N(s, a, i)}$$

Different implementations of MCTS may have different expressions for the upper confidence bound. The constant c_{cpuct} is a search hyperparameter that controls the degree of exploration.

At every unexplored node, the neural network makes an inference providing the state-value and the action-values. The state-value is backpropagated in the search tree, updating $Q(s, a, i)$. If the unexplored node is a terminal state where the agent is eliminated or the game reaches 200 moves, the actual reward is backpropagated instead. The action values are softmaxed and stored as $P(s, a, i)$.

At the root node s', the most explored action $\max_a N(s', a, 0)$ is the returned result of our search strategy.

5.2 Approaches at Improving the Search

In this section, we describe how we attempt to improve the search and the intention behind the improvements. The code is described in Section . In the following section, Section 6, we describe and interpret the results.

5.2.1 V13 - Use of Different Agents in Search

In the original AlphaGeese notebook, the PubHRL model is used to make model suggestions at every node. We replaced the pretrained parameters shared by HandyRL with our finetuned parameters.

To better simulate the opponent moves, when we are making a neural network inference for opposing agents, we use the original PubHRL parameters. This approach gave our first agents that perform better than the original AlphaGeese.

5.2.2 V14 - Assuming Rivals follow PubHRL Deterministically

In the MCTS, we assume that our opponents' moves are deterministic and follow the inferences of PubHRL without modification. As this reduces the branching factor, we are also able to search more deeply into the tree. This is done by setting the setting exploration constant of other agents to zero.

5.2.3 V17 - Use of Overage Time to Tie Break

Each agent is given one second to make an inference for every move. However, there is one fine detail of this rule - each agent is allowed to exceed one second for a total of one minute in each game. The original AlphaGeese model does not make use of this overage time.

One way to use the overage time is to extend the allowed duration of each move by 0.3 seconds on average. We have decided to use this overage time more strategically by spending it on moves where are current probabilities are close.

In our implementation, each move is allowed to use an overage time of up till the four times the remaining overage time if shared equally by remaining steps. We employ the overage time if the current tree search probability is under a certain threshold. As move probabilities usually increase as the board gets more crowded, the threshold is set to be 0.5 at the start, which linearly increases to 0.9 at move 200. To allow sufficient time to make the remaining moves, we do not use overage time if the remaining overage time is less than 10 seconds.

5.2.4 V32 - Soft Masking of Undesirable Moves

As described in Subsections 4.6.1 and 4.6.2, the neural network may provide bad inference in certain scenarios. In an extreme case, the model may produce a probability very close to one for a move that is certain to result in the worst possible outcome.

We multiply the action-values produced by the neural network model with the following coefficients if the scenarios apply. The values of the coefficients are arbitrarily chosen.

- 0.000 if the move is illegal. This is already present in the original AlphaGeese notebook [2].
- 0.101 if the move results in a potential tail strike as described in Section 4.6.1.
- 0.111 if there is a potential of a head collision with another goose of greater length as described in Section 4.6.2.
- 0.888 if the move results in a head collision with another goose of shorter or equal length while there are at least three remaining agents in the game.
- 3.333 if there is are only two remaining agents in the game and you are the longer goose. In this case, a head collision secures first place.

This modifies the value of $P(s, a, i)$, which is obtained after softmaxing the action-values. We hope that this encourages the MCTS to start the search from a more reasonable, in the hopes that it will return a better result.

6 Results and Discussion

6.1 Agent Performance and Analysis

Table 1 shows the performance of our agents on the leaderboard. The version refers to the notebook version. The count refers to how many of the same agents have we submitted. The mean, standard deviation and maximum ratings of the agents submitted are also tabulated. The number of wins of the agent against three copies of the original AlphaGeese agent over 100 matches is recorded under column AG.

Model	Ver	Count	AG	Mean	Std	Max
AlphaGeese without edits	1	35	26	1020	10.4	1043
Use Finetuned Agent	13	11	**35**	1024	13.6	1045
Assume Deterministic Rivals	14	10	28	**1037**	10.4	**1055**
Use Overage Time to Tie Break	17	8	25	1030	10.0	1043
Softmasking Dangerous Moves	32	8	30	1001	20.3	1025

Table 1: Comparison of Submitted Agents

The performance distribution of AlphaGeese closely matches the second peak illustrated in Figure 2. Notably, submitting sufficient copies of AlphaGeese [2] can bring the participant a very good chance at winning a Kaggle Bronze medal, which requires a cutoff of 1050 (to be determined).

The model with the best mean rating is Version 14 that assumes that the opponent is deterministic. At the close of submissions before the scores are allowed to converge further, copies of this agent has a small standard deviation in rating. The high mean and low variance score might be due to its consistent superior

performance against the PubHRL submissions, while under-performing against better agents also due to the same absolute assumption of its rivals.

While our scores converge, our best agent frequently rotates between Versions 1, 13, 14 and 17. Version 14 and Version 17 has a higher and similar mean score and it expected that they deliver our top agent. Version 1 and Version 13 has a lower mean score, but is still our top agent at times. For Version 1, this is because we submitted many of its copies than the other. For Version 13, this is because the standard deviation of its rating is higher.

Regrettably for Version 32, soft-masking dangerous moves produces an agent worse than the original AlphaGeese agent. While the soft-masking addresses scenarios where the neural network model performs badly, encouraging more alternate moves may result in shallower search depth.

6.2 Survey of Top Performing Strategies

This section summarises the successful strategies shared by the top participants. These strategies were shared when the submissions for the competition closed.

6.2.1 Imitating the Top Agents

Match replays of submitted agents are publicly available [23]. A model can be trained by learning the moves of top-rated models. The score of models trained by imitation learning is reported to be rated at around the mid-1100s, which is more than sufficient for a Kaggle Sliver medal [15]. The AlphaGeese author used the same PubHRL model architecture but trained with imitation learning obtained a score very near the Kaggle Gold medal boundary [24].

6.2.2 Optimising Model Inference Speed

One top participant [15] exported the PyTorch model into and an ONNX Runtime. ONNX Runtime parses the model to identify optimization opportunities and provides access to the best hardware acceleration available [25]. The optimisation has improved the model inference time from 3.42 milliseconds to 0.77 milliseconds [15]. By improving model inference times by approximately four times, it is possible to search four times as many more nodes and allow a better performance without any tradeoff.

6.2.3 Differentiating the Agent Behaviour

While the agents are denied internet connection, the agents have access to the system date and time. It is possible to program the agent to have different behaviour after a predefined time. One top participant [15] implemented a 90-minute timer so that when the agent is competing with lower-ranked agents right after when the agent is submitted, the agent will not make risky moves. This submission strategy is in response to the observation that agents rated below

1000 can make dangerous moves. This helps the agent to climb the ratings quickly.

As observed in Table 1, our agents have a different standard deviation in ratings even though their mean is similar. There is a possibility that some of our agents perform better at higher ratings if given a chance to reach the rating band.

The extension to this idea is for the agent to use a fake strategy until the close of submissions. This way, the team can submit more duplicate agents for a better chance at a higher maximum rating, while reducing imitation attacks.

6.2.4 Building an Evaluation System

Another popular practice of top teams is to implement a systematic procedure to benchmark their agents. Submitting the agent into the competition is the most convenient method of getting accurate feedback for how well the agent is performing. However, each team can only submit up to 5 agents each day, and it will take around a day to converge to a stable score. The stable score it converges to will still have a large standard deviation. Moreover, submitting your agent onto the leaderboard leaves the agent prone to imitation by other competitors.

The evaluation system implemented by top teams is similar to how we evaluate our agents against the original AlphaGeese agent as describe in Subsection 6.1, but at a larger scale and in parallel. Some teams went on to implement their own "league of agents" to estimate the rating of their agents [26]. A robust and efficient evaluation system will allow more variants of agents to be developed and tested.

7 Conclusion

This is our first project at reinforcement learning and we have glad to be able to make improvements on the agents published during the competition.

There are a few takeaways for future reinforcement learning projects. It is important to start from the state-of-the-art when designing the neural network model. The PubHRL neural network model is inspired by AlphaGo, and has proven to have good performance. The time-complexity tradeoff is also needed to be considered in the design of the neural network model, as more time-consuming model will result in a shallower search. We also sense that reinforcement learning models favors simplicity in both the design of the neural network model and the Monte Carlo Tree Search. It is also important to build a validation process to conduct ablation tests on improvements that you are experimenting. If you have access to plays by the top agents, imitation learning is able to provide strong results with a simple setup. We hope these takeaways can help participants get started in future reinforcement learning projects.

8 GUI Setup and Demonstration

Following contains the steps to set up the GUI

1. Clone the Github repository
   `git clone https://github.com/tinkitwong/kaggle-environments`.
2. Navigate into directory of the repository
3. Create a python Virtual Environment `python3 -m venv env`
4. Activate Virtual Environment `source env/bin/activate`
5. Install Dependencies `pip install requirements.txt`
6. Start the Jupyter server `./env/bin/jupyter notebook`
7. Navigate into notebook `playtest.ipynb`
8. Run the first two cells of the notebook (Figure 12)

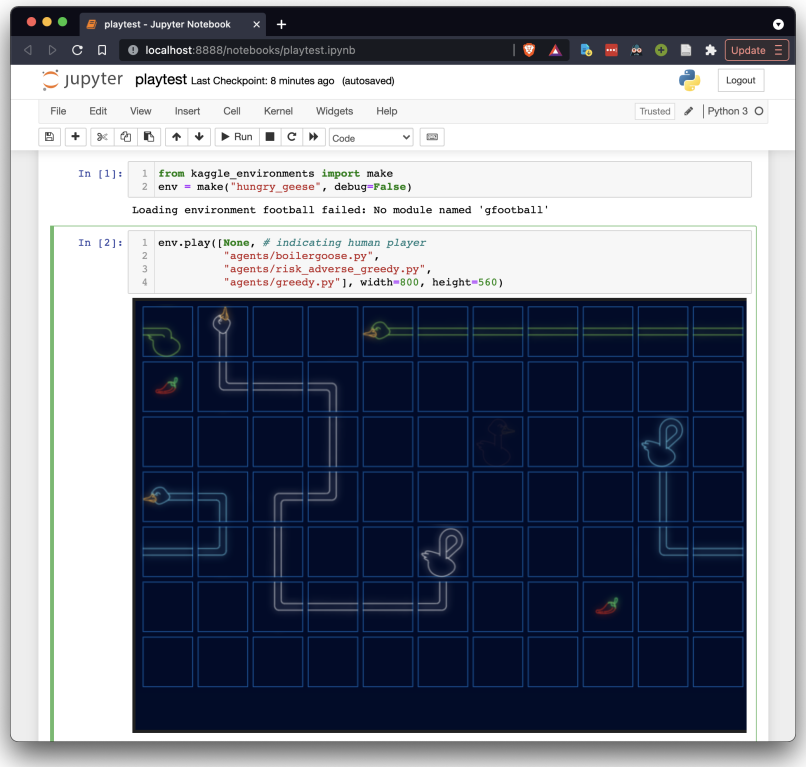

Figure 12: Screenshot of our GUI to play against the agents

When the game screen has loaded like what is seen in Figure 12, use your keyboard arrow keys (up, down, left, right) to move your agent. Your agent is the white goose.

You may choose to play against other agents by changing the arguments passed into `env.play()` This list of agents are inside the `./agents` folder. You may also add your own custom agent to play against. Playing against agents using Monte Carlo Tree Search is likely to be slow.

9 Our Code

This section documents the code that we have written for this project. Most of the code that we have used is either a fork of another repository on Github or a fork of another notebook on Kaggle. We also briefly describe the source of our code, how did we edit the code, and what the code is used for.

You may see the specific changes we made in the original versions. For Github, you can see the commit history of our forked repositories. For Kaggle, you can click on the version history of the notebooks, and compare the difference between the two versions in the history.

9.1 HandyRL Fork

`https://github.com/tonghuikang/handyRL`

This repository contains the attempted edits to train the model from scratch. We attempted to train the model with different neural network architecture, or with against different agents. This repository is a fork of a distributed reinforcement learning library from the PubHRL authors [10].

The variations of the model architecture listed in Subsection 7 is recorded in different branches. The branches can be accessed here.
`https://github.com/tonghuikang/HandyRL/branches/all`

9.2 Training Logs

`https://github.com/tonghuikang/hungry-goose-training-logs/`

The `logs/` folder contains the training logs for Section 4.4 and 4.5. `analysis.ipynb` contains the code to make the plots in Figure 7, 8 and 9. The folder `strings` contains base64 model paramters.

9.3 Submitted Agents

`https://www.kaggle.com/huikang/hg-alphageese-baseline`

In order to submit to the competition, you need to produce a notebook on

Kaggle that generates a Python file after it runs. The Kaggle simulator will call the last function defined in your Python file to query the action given the environment.

This notebook is a fork of the notebook from the AlphaGeese author [2]. The initial parameters of the neural network model are from the PubHRL authors [1].

Most of our Kaggle submissions are made from this notebook. To keep our notebook clean, and to adhere to the notebook file size limit, model parameters are downloaded from loaded from our training logs repository described in Subsection 5.2.

9.4 Agent Evaluation

`https://www.kaggle.com/huikang/hg-agents-comparison`

We use this notebook to evaluate the win-rate against AlphaGeese, as described in Subsection 6.1. This will give us a good sense of the model performance. This is a fork of a notebook from user `ihelon` [18].

9.5 GUI Notebook

`https://github.com/tinkitwong/kaggle-environments/`

This contains the code to run a GUI to play against the agents. The instructions to set up and run the GUI has been described in Section 8.

This is a fork of an official repository by Kaggle [13]. We have fixed some lines of code to enable the game logic to take keyboard input, and not stall the game if another agent has died. We have also added popular agents [18] into the repository for the GUI user to play against.

10 References

References

[1] yuricat. Smart Geese Trained by Reinforcement Learning. `https://www.kaggle.com/yuricat/smart-geese-trained-by-reinforcement-learning`, January 2021. [Kaggle Notebooks; accessed 9-Aug-2021].

[2] shoheiazuma. AlphaGeese Baseline. `https://www.kaggle.com/shoheiazuma/alphageese-baseline`, April 2021. [Kaggle Notebooks; accessed 9-Aug-2021].

[3] robga. Distribution of Scores of Submitted Agents. `https://www.kaggle.com/c/hungry-geese/discussion/230794#1404158`, July 2021. [Kaggle Discussions; accessed 9-Aug-2021].

[4] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[5] Lilian Weng. A (Long) Peek into Reinforcement Learning, 2018. [Online; accessed 9-Aug-2021].

[6] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[7] Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. *CoRR*, abs/1802.01561, 2018.

[8] Lilian Weng. Policy Gradient Algorithms, 2018. [Online; accessed 9-Aug-2021].

[9] Thomas Degris, Martha White, and Richard S. Sutton. Off-policy actor-critic. *CoRR*, abs/1205.4839, 2012.

[10] DeNA. HandyRL: Python and PyTorch for Distributed Reinforcement Learning. `https://github.com/DeNA/HandyRL/`, 2021. [Github; accessed 9-Aug-2021].

[11] kyazuki. [HandyRL] Enjoy Distributed Reinforcement Learning (Rating 1100, February 10). `https://www.kaggle.com/c/hungry-geese/discussion/255931`, January 2021. [Kaggle Discussions; accessed 9-Aug-2021].

[12] Google Cloud. Compute Engine pricing. `https://cloud.google.com/compute/all-pricing`, 2021. [Online; accessed 9-Aug-2021].

[13] Kaggle. Kaggle Environments. `https://github.com/Kaggle/kaggle-environments`, 2021. [Github; accessed 9-Aug-2021].

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[15] robga. Top ten solution: RL with LookAhead, Floodfill, and Rules. `https://www.kaggle.com/c/hungry-geese/discussion/255931`, July 2021. [Kaggle Discussions; accessed 9-Aug-2021].

[16] superant. Mighty BoilerGoose with Flood fill. `https://www.kaggle.com/superant/mighty-boilergoose-with-flood-fill`, February 2021. [Kaggle Notebooks; accessed 9-Aug-2021].

[17] gabrielmilan. Crazy Goose. https://www.kaggle.com/gabrielmilan/crazy-goose, February 2021. [Kaggle Notebooks; accessed 9-Aug-2021].

[18] Yaroslav Isaienkov. Hungry Geese - Agents Comparison. https://www.kaggle.com/ihelon/hungry-geese-agents-comparison, January 2021. [Kaggle Notebooks; accessed 9-Aug-2021].

[19] ilialar. Risk Averse Greedy Goose. https://www.kaggle.com/ilialar/risk-averse-greedy-goose, February 2021. [Kaggle Notebooks; accessed 9-Aug-2021].

[20] Surag Nair. A Simple Alpha(Go) Zero Tutorial, December 2017. [Online; accessed 9-Aug-2021].

[21] Surag Nair. Alpha Zero General, any game, any framework. https://github.com/suragnair/alpha-zero-general/blob/master/MCTS.py, 2019. [Github; accessed 9-Aug-2021].

[22] Shantanu Thakoor, Surag Nair, Megha Jhunjhunwala. Learning to Play Othello Without Human Knowledge. *Stanford University CS238 Final Project Report*, December 2017.

[23] robga. Simulations Episode Scraper Match Downloader. https://www.kaggle.com/robga/simulations-episode-scraper-match-downloader, March 2021. [Kaggle Notebooks; accessed 9-Aug-2021].

[24] shoheiazuma. AlphaGoose Simple Solution. https://www.kaggle.com/shoheiazuma/alphageese-simple-solution/, April 2021. [Kaggle Notebooks; accessed 9-Aug-2021].

[25] Microsoft. ONNX Runtime | About. https://www.onnxruntime.ai/about.html, 2021. [Online; accessed 9-Aug-2021].

[26] ironbar. Deep Q* Learning with simplified game 1200. https://www.kaggle.com/c/hungry-geese/discussion/255440, July 2021. [Kaggle Discussions; accessed 9-Aug-2021].