Technical report

Flatland 2020 – SerpenTrain

Dik van Genuchten, Gijs Pennings, Pieter Voors
serpentrain2020@serpentineai.nl

2020

Contents

1 **Abstract** 2

2 **Introduction** 3

3 **Strategy** 4
- 3.1 Rule based . . . 4
  - 3.1.1 Route reservations . . . 4
  - 3.1.2 Graph algorithms . . . 5
- 3.2 Machine learning . . . 6

4 **Implementation** 8
- 4.1 Rule-based . . . 8
- 4.2 Machine learning . . . 8
  - 4.2.1 Agent . . . 8
  - 4.2.2 Training . . . 8

5 **Results** 10

6 **Discussion** 11

7 **Conclusion** 12

8 **Acknowledgments** 13

1 Abstract

We took part in the Flatland 2020 competition and reached 17th place. In this paper, the approach, results and lessons learned will be discussed. Both a rule-based approach and a reinforcement-learning approach were tried. An introduction to the environment and competition is given, as well as the strategy and implementation details. Machine learning is becoming better in many games and shows potential in a lot of places, however, currently the rule-based method is still superior. While a machine learning method was investigated, this did not succeed. A method of asynchronous training for machine learning is presented and the baselines provided by the competition are lightly touched on.

2 Introduction

Flatland is a competition by the Swiss (SBB), German (DB), and French (SNCF) national railway companies [1]. These companies face the difficult challenge of managing traffic on complex railway networks. For instance, how should trains be routed such that they don't block each other? And, more interestingly, how to reschedule dynamically in case of accidents or breakdowns? Flatland is about finding solutions to these problems in a simplified environment, with the goal of improving or developing solutions for railway networks and logistics in general.

For the competition, the real world is modeled as a 2D grid, as can be seen in Figure 1. The railway network is represented as transitions between neighboring cells. The goal is to minimize the time it takes for each agent (i.e. train) to reach their target location. At every time step, there is a chance an agent will break down for a random length of time. All this information can be accessed via the Flatland environment.

The total score of a submission is the sum of the 'normalized return' for each episode. Each episode involves one environment with stations and a collection of trains, each having a start and target station. The environments are of increasing complexity, i.e. each is larger and contains more stations and trains than the last. The 'return' of an episode is calculated as follows. At each time step, for each agent that has not reached its target station yet, 1 point is subtracted from the total. At the end, the total is normalized to a value in the range $[0, 1]$. In case the time limit is exceeded (10 minutes of initial time, plus 10 seconds per time step) the current episode receives a score of 0.

In a similar competition last year (2019)[1], most solutions took a rule-based approach. This strategy, however, does not scale to the immense multi-agent railway networks in the real world. Therefore, this year's (2020) focus was on reinforcement learning, which usually scales more easily to larger networks. As a starting point, D3QN (Dueling Double Deep Q-Network [2]) and 'Centralized Critic' PPO (Proximal Policy Optimization [3]) baselines were provided to participants [4].

In the next sections, our approach and implementation of it are discussed. In addition, some points for improvement are listed.

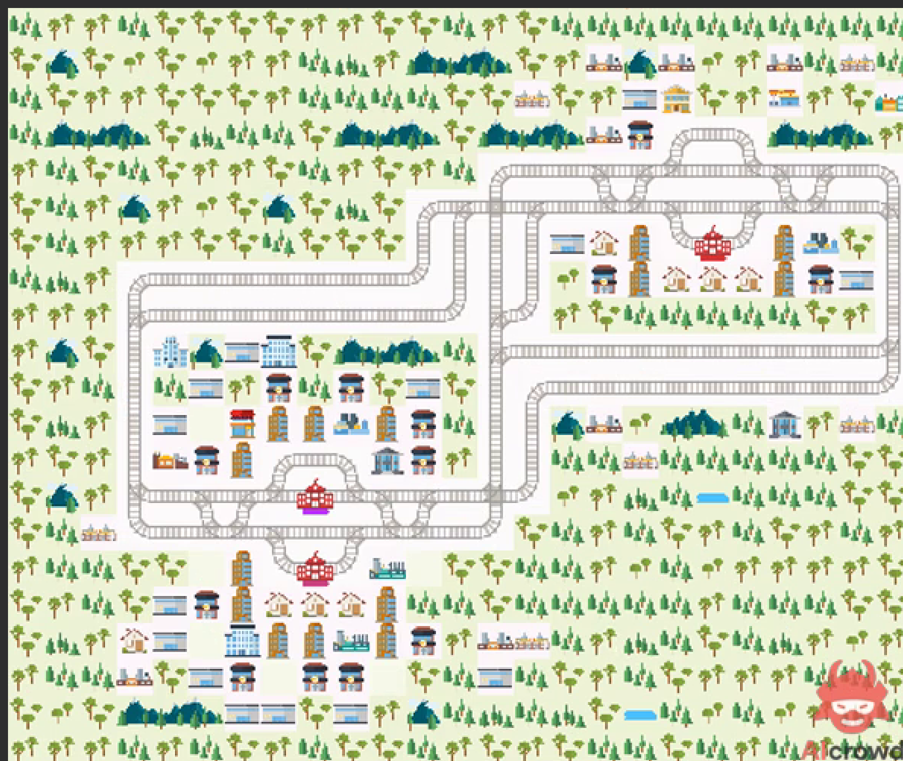

Figure 1: An example of a relatively simple environment

[1] https://www.aicrowd.com/challenges/flatland-challenge

3 Strategy

3.1 Rule based

3.1.1 Route reservations

In order to understand the code base, a simple working version was the first priority. For this, it was decided to first work on a simple rule-based agent to get acquainted with the Flatland framework as well as have a baseline agent and base score.

The very first rule-based agent, the consecutive agent, was designed to be as simple as possible while still being deadlock-free. The deadlock-free property is important, as it is easy to send two trains onto the same track in opposite directions, at which point they are stuck and the episode cannot be finished anymore as trains can only go forward, but are blocking each other. This version realized this by always having only one train driving on the map, following the shortest path to its destination.

Next, this bot was extended to a waiting agent by always driving along its shortest path (simultaneously), unless anywhere on their path another train is driving. This allows for some degree of synchronous driving of the trains while remaining very simple and deadlock-free. For the property of being deadlock-free, it is important that all trains make their move consecutively. That is, once a train decides to move forward, all other trains should be informed of this move immediately, either by actually making the move or by other bookkeeping in code, as otherwise, deadlock scenarios can occur. For example, the scenario in which two trains simultaneously enter a track they deemed free and then end up blocking each other. An example of the result of such a situation can be seen in Figure 2.

Figure 2: Two trains deadlocked.

This strategy of waiting if the shortest path is occupied anywhere can, of course, be improved significantly. The trains must often wait, even if their path is only blocked far away, which would not

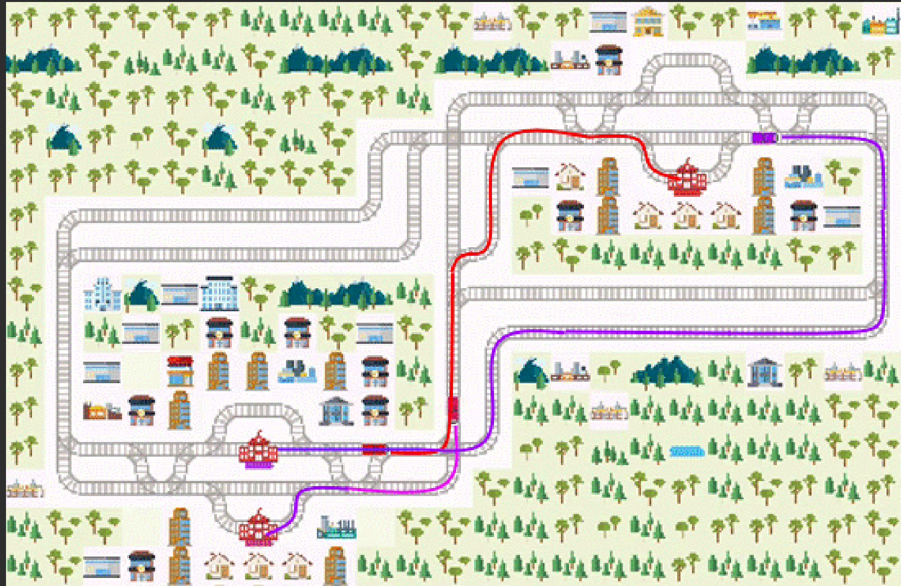

Figure 3: An example of the timed reserving agent with their reserved paths drawn as a line.

lead to any problems if they would continue forward. However, it is not as simple as only stopping if the next piece of track is occupied, as it may result in cases where both trains can only go one way or multiple trains come together, and still get stuck. A simple improvement can be made by making trains still go forward when they are already on a piece of track and not near any intersection, hereby creating a smarter waiting agent. Only when they are at an intersection, the check for trains on its path is done.

Instead of making the trains wait when their path is occupied, it is also possible to take an approach of reserving tracks. A basic strategy of such a reserving agent would be the strategy of reserving all tracks on the shortest path of a train, and then moving along this path entirely, or waiting with the reservation and movement if any part of the path is already reserved. This strategy has the additional property that once a train starts moving, it does not have to stop until it is at its destination.

To improve the strategy of the reserving agent, a first optimization is to cancel the reservation of tracks as soon as the train leaves them, instead of once the train arrives at its destination, creating a timed reserving agent. This still reserves a lot of track ahead of the trains, resulting in unnecessary waiting of other trains and limited synchronous driving. The only reason you want to reserve tracks is to prevent multiple trains from driving on the same track at the same time. Thus, it suffices to reserve the tracks only for the time intervals in which the train is on that piece of track. This massively improves how many trains can drive at the same time. An example of this can be seen in Figure 3, where multiple trains will be driving on the same track segment, but at different non-overlapping time intervals. One problem with this, however, is the fact that in the Flatland environment, trains can break down. In real life, trains have accidents, breakdowns, or other unexpected interruptions which cannot be accurately predicted. Therefore, the exact arrival times of trains at pieces of track is not 100% accurate, which in the Flatland environment can thus still lead to deadlocks when you need to re-time the reservations. One way to get around this is to have all trains stop when any train breaks down and restart the entire reservation when all trains are operation again, but this is obviously very inefficient when trains break down often.

3.1.2 Graph algorithms

Besides thinking about planning strategies, there was also some discussion on techniques for finding paths. Since the environment often requires you to reschedule, it is useful to always have the distance to the target from any position available. Note that with this information, you can also efficiently find the next move in the shortest path by selecting the neighbor that itself is the smallest distance away

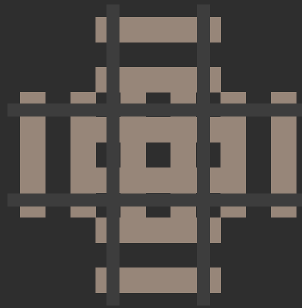

Figure 4: Diamond crossing

from your target. For this, algorithms such as the Floyd-Warshall all-pairs shortest path algorithm can be used to calculate the distances between any two points beforehand [5]. However, it is not (necessarily, depending on your main strategy) needed to know the shortest path to any point, but only to the targets. It may in practice also be quicker to run single-path algorithms such as Dijkstra's algorithm [6] or $A^\star$ [7].

To find the shortest path or, in general, apply graph algorithms on the railway network, the network should be expressed as a graph. However, that is not as simple as creating a node for each intersection and edges between any two intersections that have a rail section between them. This is because, at intersections, it is not always possible to turn in any direction. For example, at a straight crossing, you can either go from top to bottom or from left to right but not from left to the top rail, as shown in Figure 4. In essence, in Flatland the direction you can leave a node at depends on the direction you arrived from. While it is possible to carefully adapt your algorithms to take into account the directions the rails go at intersections, adjusting the graph to remove this restriction would allow for easier applying of various graph algorithms. One way to do this is to create four nodes for any intersection, one for each direction of the intersection. Then, for each pair of incoming and outgoing directions, you add an edge if you can traverse the intersection via it. Then, if you add for every rail section in the environment an edge to your graph connected to the corresponding direction nodes, you obtain a graph to which you can apply default graph algorithms. Such a graph then contains $2S \leq 4I$ vertices and at most $6I + S$ edges, where I and S denote the number of intersections and rail sections in the environment respectively.

Another option to create such graphs is to create a node for every section of railway and connecting these sections with an edge wherever there is an intersection. When creating these edges, you only add edges for an intersection if that intersection in the Flatland environment allows you to enter the intersection from the first rail section and exit the intersection at the second rail section. This then results in a graph that you can apply default algorithms to, containing at S vertices and at most $6I$ edges. Note that this graph is smaller than (or as large as) the previous graph. However, the edges and vertices are 'inverted' compared to a normal graph representation, which can make the development of algorithms for it harder. Moreover, it also makes it harder to prevent multiple trains from attempting to enter the same intersection at the same time, as one intersection is now represented by multiple edges.

3.2 Machine learning

As Serpentine members, we are especially interested in the machine learning aspect of this competition. Therefore, machine learning techniques were applied, starting with a Deep Q Network (DQN) [8]. However, Q-learning requires a considerable amount of training data, and to optimally make use of the computational power that was available, asynchronous training was a must. The solution used was a server-based approach, consisting of three main cornerstones: the actor, trainer, and buffer server.

The actor server is responsible for generating experiences by playing in the environment. The buffer server is responsible for storing, aggregating and sampling the experiences from the actor server. The trainer server is responsible for updating the DQN using the experiences sampled from

the buffer, and running evaluation runs. The visualization of the training is also done in the trainer server. This design allows for saturation of the GPU of the trainer server with a continuous stream of new samples by scaling the number of actors. However, in the end, this was not achieved due to unexpected (CPU) bottlenecks.

4 Implementation

4.1 Rule-based

In order to be flexible in strategy, importance was placed on making sure that different versions could be easily implemented. It was decided to use a controller-agent structure; each step, a controller is called, which in turn calls each agent for their action. This way, agents can make decisions on agent-specific state information, while the controller can influence the general plan of the group. By making the agent and controller models abstract, different versions of them could easily be implemented and swapped out anywhere. In practice, for the strategies that were implemented in the end, only controllers were necessary. The implementation follows from the strategy described in section 3.

4.2 Machine learning

4.2.1 Agent

The DQN was chosen because it is a relatively easy algorithm that can be expanded by various techniques. This allowed us to verify the other parts of our training, i.e. the server-based training. The initial reward function was a discounted reward, consisting of a combination of the global score ($-1 \cdot$ #agents) and an agent-specific (-1). As input, the rail network was converted to a tree, with at each junction a node with 3 children, where the distance to the next node was its value. For directions that were not possible, a high value was chosen.

4.2.2 Training

To optimally make use of the workstation that was provided, the PyTorch multiprocessing package was used to handle the initialization and communication between each 'server'. The inter-server communication was done via queues. Although this was not optimal, it was initially good enough, since it was not the limiting factor within the training.

The buffer server is a single process. It continuously pulls data from the actor-buffer queue to process it in its experience replay buffer. Then, it fills the buffer-trainer queue with a combination of randomly sampled old and new experiences, to be trained on. The specific implementation of buffering and sampling can easily be changed, as long as it pulls experience and pushes training samples.

The trainer server is also a single process. It first trained the model using the samples from the buffer-trainer queue, and after n training steps it ran m evaluation rounds. (The training rounds were not visualized as saving an image of each step took more than two seconds per step). To still get some insight into what the model was doing, various statistics were logged. E.g. how often the model took a certain action. This gave a clear picture that our model was not working as intended, as the model was consequently returning either invalid (`NaN`) or the same action, regardless of input. However, due to the time constraint, it was not investigated further.

The actor server is composed of two main components. The so-called brain process, which is responsible for running a single model on the GPU, and the worker processes, which are responsible for running a single game, requesting actions from the central brain. This was done by putting an observation and ID tuple into the worker-brain queue. The brain then pulls a batch from the queue, storing which ID belongs to which element in the batch. After a forward pass, each resulting action was returned via a shared dictionary with the ID as key. Each worker could then wait for their ID to show up in the shared dictionary. Writing to this shared dictionary was extremely slow compared to all other operations. However, no proper solution was implemented within the time frame of the competition. A shared array is expected to be faster as it would not require a lock on the entire array when writing to an index, which is the case for the dictionary implementation.

These three servers only need minimal communication between each other and, therefore, could theoretically run on different nodes within a cluster. To make it usable, a better way of communicating

for the actor server needs to be found, or the central GPU accelerated model needs to be replaced by a (slower) CPU model for each worker, removing the benefit of doing batch inference and taxing the already taxed CPU even more.

5 Results

The machine learning model yielded unusable results that could not be fixed before the deadline. Therefore, the rule-based reserving agent as described in section 3.1.1 was our main submission. Not much time was spent on it since the focus was put on machine learning. In the end, we finished 17th, which can be seen Table 1).

The time reserving agent follows a simple set of rules that always ensures progress, although sometimes at a slow pace. An example of a situation causing such a slow pace can be found in Figure 5, where an agent is waiting for a reserved path to be freed, even though it has room to move forward without creating (a possibility for) deadlocks. In the figure, the red train (left) is moving towards the station on the right, while the pink train (right) is trying to move to the one on the left. However, the left train has reserved its current path (including the 'roundabout'). Since the pink train's path partly overlaps with it, the train stops moving completely, even though it could still move on the bottom track without risk of collisions. This shows one of the limitations of the time reserving agent. In other cases, the time reserving agent performs very well, allowing many trains to move at once, as can be seen in Figure 6.

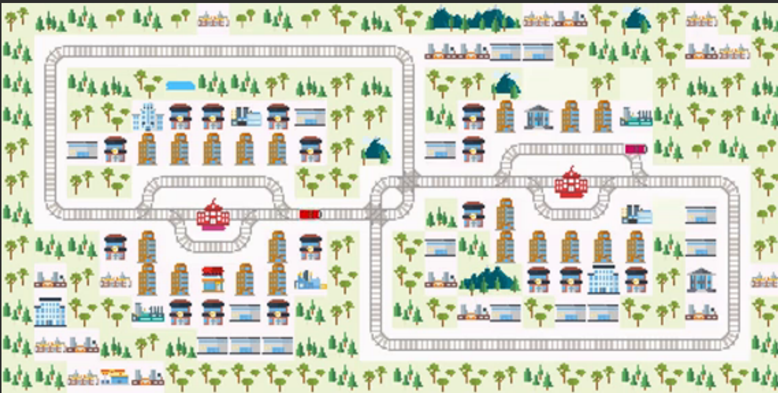

Figure 5: Example scenario where a train halts because of another train on its path.

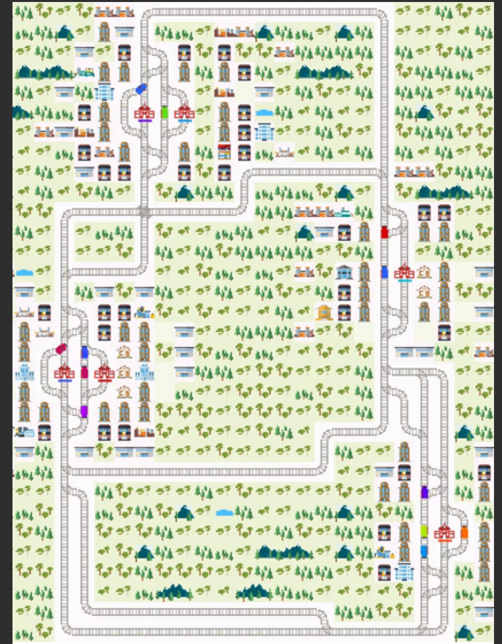

Figure 6: Example scenario where 7 trains can move at once.

Agent	Ranking	Score	% done	Running time
Reserving Agent	17	99.813	70.1	$O(n\,v\,\log v)$
CCPPO with Transformer	23	81.237	67.9	$O(n)$
Double Dueling DQN	32	54.646	53.5	$O(n)$

Table 1: Comparison between the reserving agent and the provided baselines, where n is the number of agents and v the number of intersections in the rail network.

6 Discussion

In the first half of the contest, the rule-based approach was worked on. Only relatively simple ideas were implemented, which were expected to get quite low scores, but it was surprising how well these simple rules worked. The bot was not very efficient, but generally robust. However, it sometimes ran into deadlocks of which the cause could not be found. This caused quite some time to be spent on debugging, although one of the benefits of using simple rules for a bot is generally that it is easier to understand and as such also easier to debug. Unfortunately, for the implementation that was not always the case. Still, the simple rules gave surprisingly good results and worked well to get accustomed to the contest, the code and the Flatland environment.

Instead of continuing with the promising rule-based approach, it was chosen to focus on machine learning. It was expected that with machine learning, the achieved score would not be as high as with a rule-based approach. This turned out to be true, but even more significantly than was expected. Unfortunately, the final machine learning bot did not get a positive score. The majority of the development time was spent on a version that could not properly be tested, which then resulted in a lot of debugging when all parts were put together. At that point, not a lot of visualization or data-tracking of how the algorithm was performing was created, which made it even harder. In a next iteration of the contest, more focus should be put into getting a visualization and/or performance data out of the algorithm. Moreover, when everything was combined, it could only be run on the remote server, which made testing and visualizing even harder. This shows the importance of testing the base systems locally and in smaller pieces, before combining them and uploading them to an external compute server.

At the end, the rule-based solution is superior. Additionally, it took less time to create and its correctness is easier to prove. Compare that to a machine learning approach, especially the black box models used here, where it is practically impossible to prove its correctness for all cases (indeed, the unpredictable breakdowns of the trains make it a complex problem). In the reserving agent all actions were recalculated every step, and thus it was possible to react to this dynamically. A machine learning approach has the same benefit and, in addition, it might be able to take breakdowns more into account when sending a lot of trains down a long path. Therefore, we expect that, for the near-future, rule-based is still a better choice for this problem. Machine learning is particularly suited to problems in which domain knowledge is important, problems for which solutions are very difficult to implement, or strategic planning, in which a solution needs to maintain a balance between short-term and long-term rewards.

Furthermore, it is interesting to note that all top participants used a rule-based (also known as OR, operations research) approach as opposed to machine learning[2]. This suggests that for such complex environments, classical methods still outplay machine learning approaches. For example, even the reserving agent beat the provided machine learning baselines.

However, rule-based has an important disadvantage compared to machine learning: it scales much less well to larger environments. For instance, all strategies described in section 3.1 used a shortest path algorithm, which depends on the size of the network. On the other hand, running a machine learning model usually has a fixed (although often large) cost per agent, scaling linearly in the number of agents, without depending on the size of the graph. The observation used as input for the model still has to be calculated, but the cost of this is typically low compared to the running time of rule-based approaches.

[2] https://www.aicrowd.com/challenges/neurips-2020-flatland-challenge/leaderboards

7 Conclusion

Rule-based methods still have many benefits over their machine learning counterparts. They are more explainable, provable and generally better at solving the task at hand. Basic machine learning methods are still beaten by simple rule-based methods. However, new and improved machine learning approaches are invented which can be applied to a broad range of tasks. During the competition, some participants tried and succeeded in applying machine learning methods to the problem, with the highest-ranking machine learning approach this year ending 7th place.

Approaching problems with a machine learning algorithm requires mostly the same approach as rule-based (i.e. first getting a basic algorithm working, then improving it and understanding what your agent does). However, in machine learning understanding 'what the model does' requires more effort. Therefore, visualization during training and gaining insights into how the model evolves over time is very important.

8 Acknowledgments

This project was done within Serpentine, the student team of the Eindhoven University of Technology (TU/e) that competes in AI programming contests. We would like to thank the TU/e, as well as our industry partners VBTI and Flowserve for supporting our work.

References

[1] SBB, DB & SNCF, "Neurips 2020: Flatland challenge." `https://www.aicrowd.com/challenges/neurips-2020-flatland-challenge`, 2020. Accessed: April 22, 2021.

[2] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *International conference on machine learning*, pp. 1995–2003, 2016.

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv:1707.06347*, 2017.

[4] SBB, DB & SNCF, "Flatland baselines source code." `https://gitlab.aicrowd.com/flatland/neurips2020-flatland-baselines`, 2020.

[5] Robert W. Floyd, "Algorithm 97: Shortest path," in *Communications of the ACM*, p. 5(6):345, 1962.

[6] E. W. Dijkstra *et al.*, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[7] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.