Comparing the Effectiveness of Using Deep Reinforcement Learning and Neuroevolution Algorithms for Training a Neural Network to Play Mancala

Research Question: To what extent is either deep reinforcement learning or neuroevolution a more effective algorithm for training a neural network to play the board game mancala?

Subject: Computer Science

Word Count: 3949

Table of Contents

Board games have been a prominent application of artificial intelligence (AI) due to their often complex and nuanced nature. Many methods of designing an AI to effectively play various board games have been developed, from explicitly programming known strategy to much more abstract solutions. This paper will focus on mancala, a strategy board game which dates back thousands of years yet has relatively little modern AI which can play it effectively.

In the case of board games with *perfect information*, where all information about the state of the game is known to all players at all times, the game may be "solved" by recursively finding the best move on every turn and discovering which sequence of moves results in a win (Silver et al., 2016). This is a very feasible solution for simple games such as tic tac toe, where the number of moves per game is very low. Since mancala is a game of perfect information, using a similar method would be theoretically possible; however, the large number of possible moves per game makes such an algorithm far too computationally intensive here, as with other complex games such as chess and Go (Silver et al., 2016).

A modern solution for this problem is the use of neural networks which take a certain board state and output the best move in that situation (Silver et al., 2016). However, most methods for "training" these neural networks to better predict optimal moves entail the use of large databases of games which have already been played. Due to the lack of such data in the case of mancala, there is a need for another algorithm which has no need of training data; many examples even show that such algorithms produce better trained neural networks than those which use training data (Silver et al., 2016). Two such algorithms which are well suited for the case of mancala

are neuroevolution and deep Q learning. This leads to the following question: To what extent is either deep reinforcement learning or neuroevolution a more effective algorithm for training a neural network to play the board game mancala?

The former question will be analyzed through an examination of the theory behind each algorithm, and how its advantages and disadvantages would theoretically translate to the application of mancala. Further, testing of each algorithm will be conducted examining which one produces neural networks that employ better strategy, providing a practical answer on which algorithm was more effective.

Hypothesis

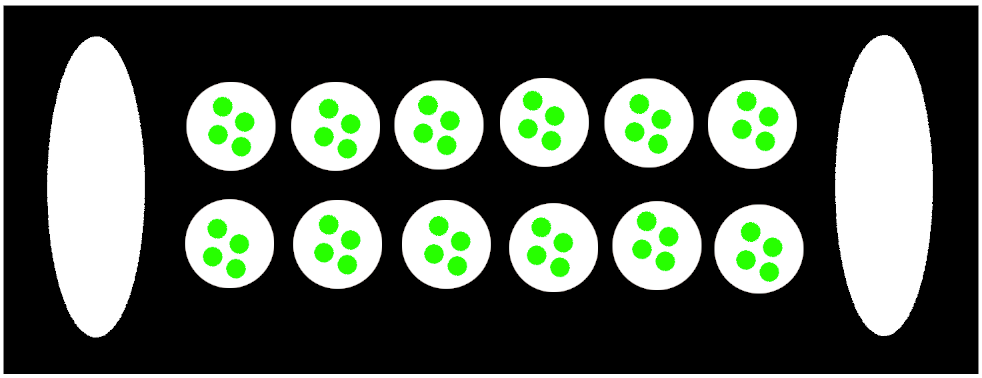
My hypothesis that the ability of deep Q learning to directly teach which moves are advantageous, combined with the inability of neuroevolution to do so, will make deep Q learning a faster and more effective algorithm for training a neural network to play mancala.

Mancala

This paper will focus on Kalah, the most popular modern variant of mancala. It is a game played by two players, who each sit on opposite sides of a rectangular board. The board contains 6 *pockets* on each player's side, for a total of 12 pockets, which all start with 4 *seeds* at the start of the game. Each player also has a *store* on their right side of the board, and the player with the most seeds in their store at the end of the game wins. Figure 1 depicts the starting position of the board.

Figure 1

Initial Board Setup



Rules of the Game

The game proceeds via players taking alternating turns. On each turn, a player chooses one pocket and “sows” the seeds around the board. This entails placing one seed at a time in each pocket on the board (including the player’s store, but not their opponents), starting at the pocket after the chosen one and moving counterclockwise. If the last seed lands in the player’s own store, they can take another turn. If the last seed instead lands in an empty pocket on the player’s own side, then the last seed and all seeds in the opposite pocket are *captured* and put into their store. Play continues until one side of the board is empty, and the rest of the seeds on the non-empty side are placed in the store of the player on that side (*Kalah – Rules,* n.d.).

Kalah with the rules above is a *solved game*, meaning that optimal moves for every board position have already been calculated (Irving et al., 2000, p. 1). However, there are trillions of possible board states, meaning an AI which does not necessitate the storage of all these states is more attractive. Possible applications of such AI would include online mancala games, where utilizing a table of optimal moves would be infeasible.

Although strategy in Kalah can be very deep due to the possibility of long move combinations, there is still general strategy which can be used to infer beneficial moves. These strategies theoretically should arise while training neural networks via algorithms such as neuroevolution of deep reinforcement learning, despite not being explicitly "taught" to the networks (Silver et al., 2016, p. 1).

Neural Networks

Neural networks, at their core, are computing systems "rooted in principles abstracted from human knowledge of neurobiological function" (Munro, 2013). They are modeled after the human brain, with the *weights* and *neurons* in neural networks directly corresponding to similar parts of a biological brain.

One of the most common types of neural networks, and the type which will be examined here, is a perceptron. They are designed to take some form of input, process it in some way, and provide output which classifies the input under some category ("Perceptrons", 2007). In the context of mancala, the input provided to the perceptron is the current state of the board, and the output is the perceived optimal move given that board state. This is analogous to "classifying" the current board state under categories representing the best move to make. For example, one board state might encourage sowing the seeds from the left-most pocket, which would be indicated by the neural network "classifying" the board as favoring that move.

Perceptrons are structured in layers, which are collections of neurons. All perceptrons have an input layer and an output layer, but additional hidden layers allow for modeling nonlinear relationships between input and output ("Perceptrons", 2007). Because choosing an optimal move in mancala is not a linear function, hidden layers

are necessary for an effective AI.

The individual neurons in a neural network each represent “the truth value of a specific hypothesis” (Munro, 2013). They are mathematical functions that output a single scalar value representing the likelihood of some condition being met. For instance, output neurons each indicate the likelihood of a certain move being optimal, while neurons in hidden layers may indicate the likelihood of specific situations, such as whether a certain pocket has many stones opposite to it.

Excluding input neurons, each neuron in the network derives its value by computing a weighted sum of its inputs (“Perceptrons”, 2007). Each neuron from one layer has weights (scalar values) “connecting” to each neuron in the next layer, and these weights control how the value of one neuron affects a subsequent one. The process of neurons and weights modifying the values of neurons in the next layer continues through the layers of the neural network until the output layer, whose neurons’ values are the outputs of the network.

Training Neural Networks

A major reason why neural networks are used in complex problems is their ability to learn how to produce desired output independent of human action (Munro, 2013). In most situations, neural networks are initialized with completely random weights, but various algorithms may be used to tune those weights for more accurate output over time.

One of the most common algorithms for training neural networks is via *gradient descent*, in which the network is trained to recognize the relationship between input and output via running input with known output (a series of *training examples*) through the

network (Munro, 2013). This technique uses backpropagation, in which the network is run “backwards” by feeding input into the output layer of the network, and collecting output at what would normally be the input layer of the network (Kishore, n.d., p. 161). The weights of the network are iteratively modified based on the errors made by the network in predicting the correct output value.

However, to perform gradient descent, existing training data is required. Mancala, like many other relatively obscure board games, has no readily available database of games which may be used for training. Thus, the need for an alternative algorithm arises—one that either generates training examples on its own or utilizes a completely different method for tuning neural network weights.

Genetic Algorithms and Neuroevolution

Genetic Algorithms

Genetic algorithms are a class of algorithms which model the process of biological evolution in order to solve an optimization problem. They mimic the process whereby biological evolution creates species that are adapted to thrive in their environment; just as natural selection eliminates organisms who are not well suited for survival, genetic algorithms refine a “population of individuals” by eliminating individuals who do not accurately solve the task at hand (“Genetic algorithms”, 2007).

Genetic algorithms search a massive space of solutions by keeping a *population* of individuals, or “candidate solutions to a particular optimization problem” (“Genetic algorithms”, 2007). Just as biological evolution favors individuals with a high fitness (which may entail a variety of advantageous traits), genetic algorithms favor individuals who produce high values from some *fitness function* that rates an individual’s success in

the context of the problem (“Genetic algorithms”, 2007).

Genetic algorithms produce better and better solutions to the optimization problem over time by keeping only the most fit individuals and discarding the rest. Then, the preserved are changed randomly in processes known as *crossover* and *mutation*, which both mimic their respective biological counterparts. Crossover is the creation of “offspring” by combining two “parent” individuals, while mutation is random modification of the offspring (“Genetic algorithms”, 2007). This iterative process of discarding unfit individuals and creating a “new generation” through random modification slowly improves the population’s ability to solve the problem, theoretically causing the emergence of highly fit individuals over time.

Tradeoffs of Genetic Algorithms

Many approaches to solving optimization problems entail some form of gradient descent, in which a single, random solution to a problem is incrementally tuned over time to come to an optimal solution. However, this process is subject to the problem of approaching *local minimum*—a solution that seems like the best possible one, but in reality is inferior to a solution which is relatively unrelated from the former (“Genetic algorithms”, 2007). Approaches using gradient descent have little way to combat this problem, as they are designed to take one approach and refine it iteratively, rather than searching a wide space of solutions. Genetic algorithms, on the other hand, avoid this problem by espousing randomness and exploring unique solutions, even after many generations of refining the population.

However, the advantage of genetic algorithms is also one of its weaknesses. Due to the inherent lack of any “focused” search for a solution, genetic algorithms instead

rely on randomness for discovering a solution. This can be problematic in applications where the solution space is extremely large, and thus finding a solution with high fitness requires a degree of luck. Additionally, extremely nuanced improvements to an already good solution may be difficult for a genetic algorithm because of the randomness central to the algorithm.

Neuroevolution

Neuroevolution is an application of genetic algorithms for the purpose of training neural networks (Risi & Togelius, n.d., p. 2). In the case of neuroevolution, the "solutions" to a certain problem, as discussed earlier, are neural networks with specific weights. The simplest form of neuroevolution, and the one which will be examined here, evolves a population of neural networks with a fixed topology; that is, the number of layers and the number of neurons in each layer are constant throughout the entire process. In each generation, the neural networks are each evaluated using some fitness function which tests their ability to produce accurate output; then, the most fit neural networks are used to create the next generation. The processes of crossover and mutation occur via combining and randomly modifying the weights of the neural networks (Risi & Togelius, n.d., p. 2). At least some degree of the fitness of the parent networks is kept via maintaining the same weights from the parents. However, the continuous randomization of weights allows the algorithm to discover new combinations of weights that better solve the problem.

Advantages of Neuroevolution

Whereas other algorithms (including stochastic gradient descent) may train only one portion of the neural network at a time, neuroevolution possesses the advantage of

being able to improve the network as a whole over time (Floreano, Dürr, & Mattiussi, 2008, p. 47). Rather than tuning specific weights (which may be computationally expensive, depending on the situation), neuroevolution uses random modification of weights to quickly explore how the network can improve as a whole.

Additionally, the central concept of using *individuals* is beneficial because it means that neural networks are evaluated as a whole, rather than based on specific weights. An "individual" neural network will solve the problem with a consistent strategy because its weights are not incrementally changing over time (Stanley, Bryant, & Miikkulainen, 2005, p. 6). This improves the efficiency of the algorithm by allowing it to evaluate performance through holistic fitness functions rather direct mathematical analysis.

Neuroevolution for Mancala

The implementation of neuroevolution for mancala will use the win rate of each network against a *random player* as the fitness function, allowing for a general evaluation of performance rather than direct analysis of performance in single games. A random player is defined here as an opponent that randomly chooses valid moves on their turn. A well-trained neural network will have a high win rate against a random player which uses no strategy to decide which moves to select. Below, the algorithm is defined more explicitly:

1. Initialize a population of 200 neural networks, each with randomized weights having a standard deviation of 0.1 and a mean of 0.
2. Repeat steps 3-7 for generation *i* to *n*:
3. Create 100 pairs of neural networks and make all of the networks play max(*i*,

100) games against a random player.

4. Discard the neural network which wins less games from each pair.
5. Create 50 pairs from the remaining neural networks.
6. For each pair, generate two offspring networks via crossover (randomly combining weights between the networks) and mutation (randomly scale 20% of the weights of each network by a random value whose mean is 0 and standard deviation is 0.1).
7. Combine the parents and offspring to create a new population of 200 individuals.

Note: the number of games played against a random player is max(*i*, 100) to allow for more selectivity as time goes on; with more nuanced play, 100 games may no longer be enough to thoroughly evaluate the performance of a network.

Reinforcement Learning and Deep Q Learning

Reinforcement Learning

Reinforcement learning is a method of training an *agent* to solve a problem without explicitly specifying how to achieve the task. Instead, the agent is punished or rewarded depending on how well it accomplishes its task (Kaelbling, Littman, & Moore, 1996). Thus, this is another algorithm which has no need of training data—instead, the agent is taught through a series of “trial-and-error interactions with a dynamic environment” (Kaelbling, Littman, & Moore, 1996).

Every time the agent must choose an action, it uses an *observation* of the current state of the environment to choose what to do. After taking this action, the agent receives a positive or negative *reward* depending on whether the action was beneficial. Over time, the agent should develop a *policy* (or strategy) which defines a relationship

between observations and actions that maximize the reward over time (Kaelbling, Littman, & Moore, 1996). Although immediate rewards are prioritized by the agent, *discounted rewards* which represent theoretical rewards which may come later are also considered. Thus, reinforcement learning is especially well suited for applications such as games where strategy entails consideration of the future.

Q Learning

Q learning is a type of reinforcement learning that allows for developing an optimal policy independent of what actions the agent takes. In this algorithm, the goal is to learn the optimal *Q values* which each represent the expected reward over time given a specific state of the environment and a corresponding action (Watkins & Dayan, 1992, p. 56). Thus, Q learning is tailored towards problems which require consideration of the long-term effects of actions, as immediate rewards are sometimes insignificant compared to large rewards which can come later.

Deep Q Learning

Deep Q learning is an application of reinforcement learning in the context of neural networks. Instead of directly computing the Q values for a given state and action via an explicit function, a neural network is trained to approximate the Q values (Mnih et al., 2013). This also means that the neural network is trained to recognize which actions yield the highest rewards, as predicting Q values is equivalent to predicting reward over time.

The network itself is trained via repeatedly attempting to solve the problem at hand, and saving states, actions, and their corresponding rewards as it attempts to solve the problem (Mnih et al., 2013). These become artificially created training

examples for the neural network, meaning gradient descent can then be performed for tuning the weights of the network.

Deep Q Learning for Mancala

In the case of mancala, the Q values which are estimated by the neural network correspond to how "good" a move is in a given board state. The deep Q learning algorithm should theoretically teach the neural network the optimal Q values over time, meaning that the network should more accurately predict the value of making certain moves over others. This lends itself to a more move-by-move approach compared to neuroevolution, which evaluated neural networks based on performance in entire games. To cater to this, the reward which the agent will be given was chosen to be the difference between the number of seeds in the agent's mancala and the opponent's mancala. Additionally, to encourage some degree of exploration of new strategies, the agent will make a random move instead of the perceived optimal one with probability 0.1. The specific algorithm used for testing is outlined below:

1. Repeat steps 2-4 for *n* steps, restarting the game whenever a win condition is met:
2. Input the current board state into the neural network, and make either the move with the highest corresponding Q value with probability 0.9, or a random valid move with probability 0.1.
3. Save the state, action, and corresponding reward as a *transition* in a *replay buffer*.
4. Choose a random batch of transitions to perform gradient descent and update the neural network's weights.

Testing Methodology

To perform a fair comparison of neuroevolution and deep Q learning, the following were kept constant:

1. Both algorithms trained neural network(s) with three hidden layers, having 300 neurons each.
2. The input to each neural network was the current board state, which was defined as a vector with 290 items. The first 288 represent whether each of the 12 pockets on the board had *n* seeds in them, where $0 \leq n \leq 23$ (resulting in 12 * 24 = 288 possible combinations). The final 2 values stored the number of seeds in each store.
3. All algorithms were run with the Python programming language, on the same computer.

Each algorithm was run for approximately 150 minutes, and periodically the weights of the resulting neural networks were written to the disk. After training was complete, the following metrics were recorded for each saved network:

1. Win rate against a random player (deep Q learning)
2. Highest and average win rate against a random player across the entire population (neuroevolution)
3. Highest and average win rate of neuroevolution-trained networks against deep Q learning-trained network across the entire population

Refer to the appendix for the exact values of these metrics.

Figure 2

Highest/average win rate of a population of neuroevolution-trained neural networks against a random player and win rate of a deep Q learning-trained network against a random player after various time intervals

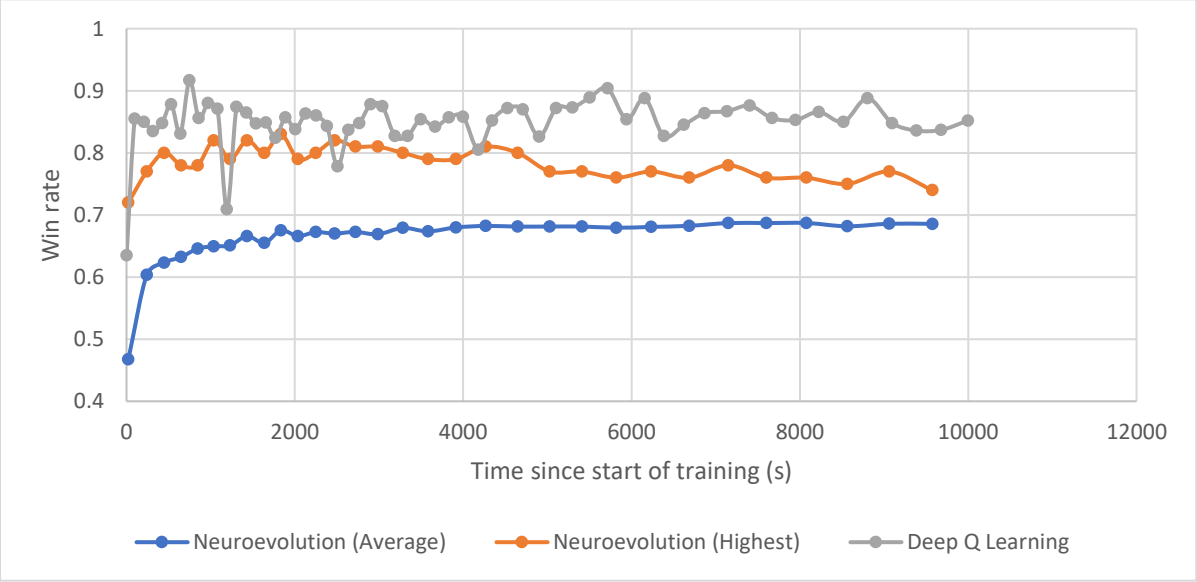
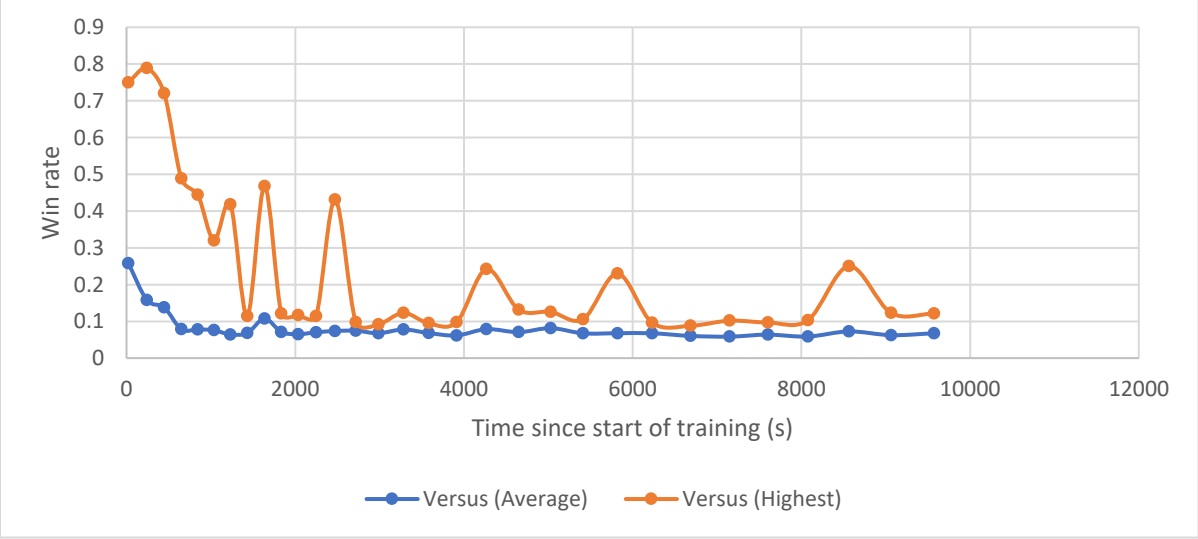


Figure 3

Highest/average win rate of a population of neuroevolution-trained neural networks versus a deep Q learning-trained network after various time intervals



Analysis of Results

In the case of neuroevolution, training was relatively effective in the first 1000 seconds, causing the average win rate against a random player to quickly approach 0.65. However, training largely stagnated as time went on, indicating an inability to develop nuanced strategy. The highest win rate even went down over time; though it occasionally increased again, the peak win rate across the entire population never increased beyond 0.8 after 5000 seconds of training. This confirms the weakness of neuroevolution, being that there was no focused training once some basic strategy had developed.

The neural network trained via deep Q learning, on the other hand, was able to not only quickly grasp basic strategy and develop a win rate greater than 0.7, it also continued to refine the strategy over time, approaching a win rate of 0.9 throughout training. Though it did not achieve superhuman play, the high win rate produced by this algorithm indicates that it was able to effectively train the neural network. Additionally, the fluctuations in win rate as training progressed indicate there was some degree of exploration for new strategies, something which neuroevolution seemed to overuse.

The win rates of the neural networks from both algorithms playing each other corroborates the earlier results. Even after relatively few iterations of both algorithms, neuroevolution-trained neural networks were winning less than 20% of games; this quickly approached 5% as time went on. Thus, the data indicates that deep Q learning was a more effective algorithm than neuroevolution, as it was able to produce a neural network capable of strategic play more quickly than neuroevolution. Additionally, deep Q learning also proved to either maintain or improve strategy over time rather than

worsening it.

Conclusion

As indicated by the recorded data, deep Q learning proved to be a much more effective algorithm to train a neural network to play mancala. The hypothesis that searching a large solution space using neuroevolution would not provide a significant enough advantage over the direct training approach of deep Q learning was correct. Although both algorithms were able to achieve a decent win rate above 0.6 relatively quickly, nuanced tuning of specific strategy was vital to achieve better play. Deep Q learning was able to directly refine the neural network, leading to more effective training.

Although here the results indicated a clear winner in deep Q learning, the hyperparameters of each algorithm (such as number of neurons in each layer, or the probability of mutation in neuroevolution) have a large impact on the performance of each algorithm. Analysis of both algorithms with different hyperparameters may have indicated different effectiveness, and so it should be noted that the results obtained here mainly apply to the specific network structure used.

In the future, the training techniques used here could be expanded upon by utilizing modern, specialized variants of each algorithm. One example of this is neuroevolution algorithms which modify network structure along with the weights, which often provides the benefit of producing neural networks better structured for their specific task (Stanley, Bryant, & Miikkulainen, 2005, p. 6). Since solving board games is often complex and requires unconventional approaches, further research that examines algorithms such as these may produce valuable results.

However, as indicated by the testing, deep Q learning is far more effective than

traditional neuroevolution for training neural networks to play mancala.

References

Floreano, D., Dürr, P., & Mattiussi, C. (2008). Neuroevolution: From architectures to learning. *CiteSeerX*. https://doi.org/10.1.1.182.1567

Genetic algorithms. (2007). In *World of Computer Science*. Gale.

Irving, G., Donkers, J., & Uiterwijk, J. (2000, September). *Solving kalah*. https://naml.us/paper/irving2000_kalah.pdf

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, *4*.

Kalah - rules of the game. (n.d.). The University of Manchester. Retrieved August 30, 2020, from http://syllabus.cs.manchester.ac.uk/ugt/2015/COMP34120/KalahRules.htm

Kishore, R. (1996). The backpropagation algorithm. *International Journal of Scientific & Engineering Research*, *3*(6).

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013, December). *Playing Atari with deep reinforcement learning*. https://arxiv.org/pdf/1312.5602.pdf

Munro, P. (2013). Neural networks. In K. L. Lerner & B. W. Lerner (Eds.), *Computer Sciences* (2nd ed.). Retrieved from Gale in Context: Science database.

Perceptrons. (2007). In *World of computer science*. Retrieved from Gale in Context: Science database.

Risi, S., & Togelius, J. (2015). Neuroevolution in games: State of the art and open challenges. *dblp*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*(7587), 484+.

Stanley, K., Bryant, B., & Miikkulainen, R. (2005). Real-time neuroevolution in the NERO video game. *IEEE Transactions on Evolutionary Computation*, *9*(6). https://doi.org/10.1109/TEVC.2005.856210

Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*. https://doi.org/10.1007/BF00992698

Appendix

Table 1

Highest and average win rate of a population of neuroevolution-trained neural networks against a random player and against a deep Q learning-trained network after various time intervals

Time since training start (s)	Average win rate against random (%)	Highest win rate against random (%)	Average win rate against deep Q learning network	Highest win rate against deep Q learning network
22	0.4676	0.72	0.2580	0.750
241	0.6035	0.77	0.1583	0.789
447	0.6233	0.80	0.1378	0.720
650	0.6324	0.78	0.0785	0.489
846	0.6458	0.78	0.0782	0.444
1037	0.6496	0.82	0.0759	0.320
1232	0.6511	0.79	0.0643	0.418
1432	0.6661	0.82	0.0684	0.114
1636	0.6550	0.80	0.1073	0.468
1836	0.6750	0.83	0.0713	0.121
2037	0.6661	0.79	0.0651	0.117
2248	0.6724	0.80	0.0703	0.114
2473	0.6701	0.82	0.0738	0.431
2719	0.6728	0.81	0.0746	0.098
2988	0.6690	0.81	0.0678	0.092
3282	0.6792	0.80	0.0779	0.123
3582	0.6736	0.79	0.0683	0.095
3914	0.6799	0.79	0.0617	0.098
4268	0.6824	0.81	0.0789	0.242
4648	0.6813	0.80	0.0706	0.132
5028	0.6814	0.77	0.0816	0.126
5413	0.6814	0.77	0.0674	0.106
5821	0.6795	0.76	0.0678	0.230
6233	0.6808	0.77	0.0676	0.096
6688	0.6826	0.76	0.0604	0.088
7150	0.6870	0.78	0.0585	0.102
7603	0.6870	0.76	0.0639	0.097
8077	0.6871	0.76	0.0587	0.103
8564	0.6818	0.75	0.0727	0.250
9064	0.6859	0.77	0.0625	0.123
9574	0.6857	0.74	0.0677	0.121

Table 2

Win rate of deep Q learning-trained neural network against a random player after various time intervals

Time since training start (s)	Win rate against random (%)
1	0.635
99	0.855
206	0.850
316	0.835
423	0.848
531	0.878
640	0.831
749	0.917
859	0.856
969	0.880
1081	0.871
1194	0.709
1308	0.874
1424	0.865
1539	0.847
1655	0.849
1770	0.824
1887	0.857
2005	0.838
2129	0.863
2255	0.860
2381	0.843
2508	0.778
2636	0.837
2767	0.848
2900	0.878
3040	0.875
3189	0.827
3339	0.827
3495	0.854
3666	0.842
3831	0.857
4000	0.858
4179	0.805
4344	0.852
4525	0.872

Time since training start (s)	Win rate against random (%)
4708	0.870
4903	0.826
5105	0.872
5298	0.873
5505	0.889
5715	0.904
5942	0.854
6158	0.888
6384	0.827
6623	0.845
6869	0.864
7136	0.867
7404	0.876
7670	0.856
7950	0.853
8225	0.866
8518	0.850
8804	0.888
9097	0.848
9385	0.836
9680	0.837
9996	0.852

Program Code

All the program code below was written by me and used for running and testing the two algorithms, neuroevolution and deep Q learning. All programs were run on the same computer with a Ryzen 5 1600 processor.

neuroevolution.py

```
"""

  This module trains a population of neural networks to play mancala using neuroevolution.

  The training time and average/best win rate against a random player for each generation are
output to the results folder:
  - neuroevolution-average-winrate.txt
  - neuroevolution-best-winrate.txt
  - neuroevolution-time.txt

  The weights of the neural networks are also saved in the neuroevolution-
networks folder after each generation.

"""

import numpy as np
from numpy import random
from itertools import chain
import json
import multiprocessing
import time
import pickle
from joblib import Parallel, delayed

num_cores = multiprocessing.cpu_count()

class NeuralNetwork:
  def __init__ (self, weights, biases):
    self.weights = weights
    self.biases = biases
  def feedforward(self, activations):
    for weight_matrix, bias_vector in zip(self.weights, self.biases):
      activations = self.sigmoid(np.dot(weight_matrix, activations) + bias_vector)
    return activations
  def sigmoid(self, x):
    return 1 / (1 + np.exp(-x))

class GameResult:
  def __init__(self, winner, scores):
    self.winner = winner
    self.scores = scores
```

```
class RandomPlayer:
  def __init__ (self):
    pass
  def feedforward(self, activations):
    return random.rand(7)

population_size = 200

random_player = RandomPlayer()

def genetic_algorithm(layer_size, layer_size_2, layer_size_3,
                      weight_init_stdev, weight_mutation_chance, weight_mutation_stdev,
                      bias_init_stdev, bias_mutation_chance, bias_mutation_stdev):
  def simulate():
    population = []
    for _ in range(0, population_size):
      weights = [np.random.normal(0, weight_init_stdev, (layer_size, 291))]
      if layer_size_2 != 0:
        weights.append(np.random.normal(0, weight_init_stdev, (layer_size_2, layer_size)))
      if layer_size_3 != 0:
        weights.append(np.random.normal(0, weight_init_stdev, (layer_size_3, layer_size_2)))
        weights.append(np.random.normal(0, weight_init_stdev, (7, layer_size_3)))
      elif layer_size_2 != 0:
        weights.append(np.random.normal(0, weight_init_stdev, (7, layer_size_2)))
      else:
        weights.append(np.random.normal(0, weight_init_stdev, (7, layer_size)))

      # rows of matrix are hidden/outputs, while columns are inputs

      biases = [np.random.normal(0, bias_init_stdev, layer_size)]
      if layer_size_2 != 0:
        biases.append(np.random.normal(0, bias_init_stdev, layer_size_2))
      if layer_size_3 != 0:
        biases.append(np.random.normal(0, bias_init_stdev, layer_size_3))

      biases.append(np.random.normal(0, bias_init_stdev, 7))

      population.append(NeuralNetwork(weights, biases))

    def play_game(player1, player2):
      # board structure: start counting at leftmost pocket on player's side
      # go counterclockwise, and end at opponent's mancala
      board = np.array([4, 4, 4, 4, 4, 4, 0, 4, 4, 4, 4, 4, 4, 0])
      currentPlayer = player1 if random.choice([True, False]) else player2

      turn = 0
      pie_rule_available = False

      while not np.all(board[0:6] == 0) and not np.all(board[7:13] == 0):
        currentPlayer = player1 if currentPlayer == player2 else player2
        # flip board orientation every turn
        board = np.concatenate([board[7:14], board[0:7]])
        turn += 1
        while True:
```

```
          observation = [0] * 288
          for i, pieces in enumerate(list(board[0:6]) + list(board[7:13])):
            observation[pieces + (i * 24)] = 1
          observation += [board[6], board[13], 1 if turn == 1 else 0]

          outputs = currentPlayer.feedforward(observation)
          move = 0
          move_confidence = -1
          for i in range(0, 7):
            if outputs[i] > move_confidence:
              if i == 6:
                if turn == 2 and pie_rule_available:
                  move = i
                  move_confidence = outputs[move]
              elif board[i] != 0:
                move = i
                move_confidence = outputs[move]

          def resolve_move(board, pie_rule_available):
            if move == 6:
              board = np.concatenate([board[7:14], board[0:7]])
              return False
            if turn == 2:
              pie_rule_available = False
            pocket = move
            while True:
              pieces = board[pocket]
              if pieces == 0:
                # Invalid move, assume game is over because all pockets are empty
                return False
              board[pocket] = 0
              for _ in range(pieces):
                pocket = pocket + 1 if pocket < 12 else 0
                board[pocket] = board[pocket] + 1
              if pocket == 6:
                return True
              if board[pocket] == 1:
                if pocket < 6:
                  board[6] += board[12 - pocket] + 1
                  board[pocket] = 0
                  board[12 - pocket] = 0
                return False

          free_move = resolve_move(board, pie_rule_available)
          if free_move:
            continue
          else:
            break

      mancala_to_fill = 13 if np.all(board[0:6] == 0) else 6
      board[mancala_to_fill] += np.sum(np.concatenate([board[0:6], board[7:13]]))
      winner_player = currentPlayer if board[6] > board[13] else (player1 if currentPlayer ==
player2 else player2)
      winner = player1 == winner_player
```

```
      scores = [board[6] if currentPlayer == player1 else board[13], board[6] if currentPlayer
 == player2 else board[13]]
      return GameResult(winner, scores)

    generations = 1000

    # used later to evaluate fitness
    def play_against_random(player, games):
      won = 0
      for _ in range(games):
        game = play_game(player, random_player)
        if game.scores[0] > game.scores[1]:
          won += 1
      return won

    start = time.time()

    open("results/neuroevolution-time.txt", "w").close()
    open("results/neuroevolution-average-winrate.txt", "w").close()
    open("results/neuroevolution-best-winrate.txt", "w").close()

    for generation in range(0, generations):

      shuffled_population = population.copy()
      random.shuffle(population)
      # the networks play games against a random player
      # the networks that win more games move on to become parents of the next generation
      fitness_games = max(100, generation)

      def select_from_pair(player1, player2):

        score1 = play_against_random(player1, fitness_games)
        score2 = play_against_random(player2, fitness_games)

        return player1 if score1 > score2 else player2, score1, score2

      pairs = []
      for _ in range(population_size // 2):
        pairs.append((shuffled_population.pop(), shuffled_population.pop()))
      results = Parallel(n_jobs=num_cores)(delayed(select_from_pair)(pair[0], pair[1]) for pai
r in pairs)
      winners, scores1, scores2 = [list(a) for a in zip(*results)]

      if generation % 10 == 0:
        scores = scores1 + scores2

        generation_time = time.time() - start
        average_score = sum(scores) / population_size / fitness_games
        best_score = max(scores) / fitness_games

        with open("results/neuroevolution-time.txt", "a") as f:
          f.write(str(generation_time) + "\n")
        with open("results/neuroevolution-average-winrate.txt", "a") as f:
          f.write(str(average_score) + "\n")
```

```
        with open("results/neuroevolution-best-winrate.txt", "a") as f:
          f.write(str(best_score) + "\n")
        with open(f"neuroevolution-networks/generation-{generation}", 'wb+') as f:
          pickle.dump(population, f)

      # breeding (crossover/mutation)
      population = winners.copy()
      for _ in range(2):
        winners_copy = winners.copy()
        random.shuffle(winners_copy)
        pairs = []
        while winners_copy:
          pairs.append((winners_copy.pop(), winners_copy.pop()))
        def crossover_and_mutate(parent1, parent2):
          child_weights = []
          child_biases = []
          for i in range(len(parent1.weights)):
            weights_shape = parent1.weights[i].shape
            weights_crossover = random.choice([True, False], weights_shape)
            child_weights.append(parent1.weights[i].copy())
            child_weights[i][weights_crossover] = parent2.weights[i][weights_crossover]
            weights_randomized = random.normal(0, weight_mutation_stdev, weights_shape)
            weights_mutated = random.choice([True, False], weights_shape,
                                            p=[weight_mutation_chance, 1 - weight_mutation_cha
nce])
            child_weights[i][weights_mutated] += weights_randomized[weights_mutated]

            biases_shape = parent1.biases[i].shape
            biases_crossover = random.choice([True, False], biases_shape)
            child_biases.append(parent1.biases[i].copy())
            child_biases[i][biases_crossover] = parent2.biases[i][biases_crossover]
            biases_randomized = random.normal(0, bias_mutation_stdev, biases_shape)
            biases_mutated = random.choice([True, False], biases_shape,
                                            p=[bias_mutation_chance, 1 - bias_mutation_chance]
)
            child_biases[i][biases_mutated] += biases_randomized[biases_mutated]
          return NeuralNetwork(child_weights, child_biases)
        children = Parallel(n_jobs=num_cores)(delayed(crossover_and_mutate)(pair[0], pair[1])
for pair in pairs)
        population += children
      print(f"Generation {generation} complete.")
      if False:
        generation_games = 1000
        result = Parallel(n_jobs=num_cores)(delayed(play_against_random)(player, generation_ga
mes) for player in population)
        generation_win_percentage = sum(result) / generation_games / population_size
        print(f"Generation {generation}: " + str(generation_win_percentage))
        print("Best performing individual: " + str(max(result) / generation_games))

    # below computes total_won after all generations are complete
    trained_games = 1000
    trained_won_games = sum(Parallel(n_jobs=num_cores)(delayed(play_against_random)(player, tr
ained_games) for player in population))
    trained_win_percentage = trained_won_games / trained_games / population_size
```

```
    print("After training: " + str(trained_win_percentage))
    return trained_win_percentage
  simulated_win_percentage = simulate()
  return simulated_win_percentage

if __name__ == "__main__":
  # Hyperparameters used for neuroevolution
  layer_1_size = 300
  layer_2_size = 300
  layer_3_size = 300
  weight_init_stdev = 0.1
  weight_mutation_chance = 0.2
  weight_mutation_stdev = 0.1
  bias_init_stdev = 0.1
  bias_mutation_chance = 0.2
  bias_mutation_stdev = 0.1

  genetic_algorithm(layer_1_size, layer_2_size, layer_3_size,
                    weight_init_stdev, weight_mutation_chance, weight_mutation_stdev,
                    bias_init_stdev, bias_mutation_chance, bias_mutation_stdev)
```

dql.py

```
"""

This module trains a neural network to play mancala using deep Q learning.
The algorithm is mostly just set up here via method calls to external libraries.
The games are played and rewards are calculated in Env.py.

The weights of the neural network are saved to the dql-networks folder every 10,000 steps.

"""

import gym
from Env import MancalaEnv

from keras.models import Sequential
from keras.layers import Dense, Flatten
from keras import Input
from keras.optimizers import Adam

from rl.agents.dqn import DQNAgent
from rl.policy import LinearAnnealedPolicy, EpsGreedyQPolicy
from rl.memory import SequentialMemory
from rl.core import Processor
from rl.callbacks import FileLogger, ModelIntervalCheckpoint

env = MancalaEnv(use_random_opponent=True, testing=True)

model = Sequential()
model.add(Flatten(input_shape=(1, 291)))
model.add(Dense(300, activation="relu"))
```

```
model.add(Dense(300, activation="relu"))
model.add(Dense(300, activation="relu"))
model.add(Dense(7, activation="relu"))

print(model.input_shape)

policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr="eps", value_max=1, value_min=0.1,
                              value_test=0, nb_steps=50000)

memory = SequentialMemory(limit=1000000, window_length=1)

dqn = DQNAgent(model=model, nb_actions=7, policy=policy, memory=memory,
               gamma=0.99, train_interval=1, delta_clip=1.)

dqn.compile(Adam(lr=0.001), metrics=["mae"])

callbacks = [ModelIntervalCheckpoint("dql-networks/step-{step}.h5f", interval=10000)]

dqn.fit(env, callbacks=callbacks, nb_steps=1000000, log_interval=10000, nb_max_start_steps=0)

dqn.save_weights("dql-networks/final.h5f", overwrite=True)
```

Env.py

```
"""

This module defines an OpenAI Gym environment for mancala.
The board state is stored here and moves are carried in the 'step' method.
Rewards are also calculated for use with the deep Q learning algorithm.

Time taken for each set of 5,000 steps are output to the results folder (dql-time.txt).

"""

import gym
from gym import spaces
import numpy as np
import random
import time

VISUALIZE_MOVES = False

class MancalaEnv(gym.Env):
  metadata = {'render.modes': ['human']}

  def __init__(self, use_random_opponent, testing):
    super(MancalaEnv, self).__init__()
    self.start = time.time()
    self.current_step = 0
    self.games_won = 0
    self.games_tied = 0
    self.games = 0
    self.reward_range = (-10000, 100)
```

```
    self.action_space = spaces.Discrete(7)
    self.use_random_opponent = use_random_opponent
    if self.use_random_opponent:
      open("results/dql-time.txt", "w").close()
    self.testing = testing
    self.observation_space = spaces.Box(low=0, high=50, shape=(291,), dtype=np.uint8)

  def step(self, move):
    # Execute one time step within the environment

    if self.current_step % 5000 == 0 and self.use_random_opponent:
      with open("results/dql-time.txt", "a") as f:
        f.write(str(time.time() - self.start) + "\n")

    self.current_step += 1
    reward = 0

    while True:
      if self.current_player:
        reward = self.move(move)
      elif self.use_random_opponent:
        self.move(self.get_random_action())
      else:
        self.move(move)

      # if one side is completely empty, then the game is over - transfer remaining pieces to
the corresponding player's mancala
      if np.all(self.board[0:6] == 0) or np.all(self.board[7:13] == 0):
        mancala_to_fill = 13 if np.all(self.board[0:6] == 0) else 6
        self.board[mancala_to_fill] += np.sum(np.concatenate([self.board[0:6], self.board[7:13
]]))
        self.done = True

        player_mancala = 6 if self.current_player else 13
        opponent_mancala = 13 if self.current_player else 6

        reward = (self.board[player_mancala] - self.board[opponent_mancala])

        if VISUALIZE_MOVES:
          print("\nPlayer total: " + str(self.board[player_mancala]))
          print("Opponent total: " + str(self.board[opponent_mancala]))

        self.won = self.board[player_mancala] > self.board[opponent_mancala]

        if VISUALIZE_MOVES:
          self.games += 1
          if self.board[player_mancala] > self.board[opponent_mancala]:
            self.games_won += 1
          elif self.board[player_mancala] == self.board[opponent_mancala]:
            self.games_tied += 1
          if self.games % 1000 == 0:
            print("\nWin percentage:", self.games_won / self.games)
            print("Tie percentage:", self.games_tied / self.games)
            print(self.games)
```

```
            self.games = 0
            self.games_won = 0
            self.games_tied = 0

      if self.current_player or self.done or not self.use_random_opponent:
        return self.get_observation(), reward, self.done, {}

  def move(self, move):
    if move == 6:
      if self.turn == 2 and self.pie_rule_available:
        if not self.current_player:
          self.used_pie_rule = True
        # pie rule: switch sides
        self.flip_board()
        self.switch_turn()
        return self.board[13] - self.board[6]
    elif self.turn == 2:
        self.pie_rule_available = False

    pocket = move
    while True:
      if self.testing and VISUALIZE_MOVES:
        print(self.board, pocket, self.current_player)
      # distribute pieces from the chosen pocket
      pieces = self.board[pocket]
      self.board[pocket] = 0
      for _ in range(pieces):
        pocket = pocket + 1 if pocket < 12 else 0
        self.board[pocket] += 1

      # if the last piece lands in the player's mancala, then go to the next step without chan
ging turns
      if pocket == 6:
        return self.board[6] - self.board[13]

      if self.board[pocket] == 1:
        # if the last piece lands in an empty mancala on the player's side, then transfer the
pieces in the opposite mancala to the player's mancala
        if pocket < 6:
          self.board[6] += self.board[12 - pocket] + 1
          self.board[pocket] = 0
          self.board[12 - pocket] = 0
        # regardless of whether the empty pocket is on the player's or opponent's side, the tu
rn is over
        self.switch_turn()
        return self.board[13] - self.board[6]

  def switch_turn(self):
    self.current_player = not self.current_player
    self.turn += 1
    self.flip_board()

  def flip_board(self):
    self.board = np.concatenate([self.board[7:14], self.board[0:7]])
```

```
  def get_observation(self):
    observation = [0] * 288
    for i, pieces in enumerate(list(self.board[0:6]) + list(self.board[7:13])):
      observation[pieces + (i * 24)] = 1
    observation += [self.board[6], self.board[13], 1 if self.turn == 1 else 0]
    return observation

  def get_random_action(self):
    while True:
      random_move = random.randint(0, 6)
      if random_move == 6:
        if self.turn == 2 and self.pie_rule_available:
          return random_move
      elif self.board[random_move] != 0 or self.done:
        return random_move

  def get_best_action(self, values):
    max_value = -1
    best_action = None
    for i in range(len(values)):
      if values[i] > max_value:
        if i == 6:
          if self.turn == 2 and self.pie_rule_available:
            max_value = values[i]
            best_action = i
        elif self.board[i] != 0:
          max_value = values[i]
          best_action = i
    return best_action

  def reset(self):
    # Reset the state of the environment to an initial state
    self.board = np.array([4, 4, 4, 4, 4, 4, 0, 4, 4, 4, 4, 4, 4, 0])
    self.current_player = bool(random.getrandbits(1))
    self.turn = 1
    self.pie_rule_available = True
    self.done = False
    self.used_pie_rule = False
    return self.get_observation()

  def render(self, mode='human', close=False):
    # "render" the environment to the screen
    print("Current player: " + str(self.current_player))
    print("Board: " + str(self.board))
```

dql-test.py

```
"""

This module tests the neural network trained using deep Q learning.
It calculates the network's win rate against a random player after every 5,000 steps.
```

```
This win rate is output to the results folder (dql-winrate.txt).

"""

from keras.models import load_model
from keras.models import Sequential
from keras.layers import Dense, Flatten
from keras import Input
from keras.optimizers import Adam

import numpy as np
import glob

from Env import MancalaEnv

from copy import deepcopy

GAMES = 1000

model = Sequential()
model.add(Flatten(input_shape=(1, 291)))
model.add(Dense(300, activation="relu"))
model.add(Dense(300, activation="relu"))
model.add(Dense(300, activation="relu"))
model.add(Dense(7, activation="relu"))

env = MancalaEnv(use_random_opponent=True, testing=True)

open("results/dql-winrate.txt", "w").close()

for f in glob.glob("dql-networks/*.h5f"):
  model.load_weights(f)
  won = 0
  for episode in range(GAMES):
    # Obtain the initial observation by resetting the environment.
    observation = deepcopy(env.reset())

    done = False
    while not done:
      action_values = model.predict(np.array([[observation]]))[0]
      action = env.get_best_action(action_values)
      observation, r, done, info = env.step(action)
      observation = deepcopy(observation)
    if env.won:
      won += 1
  with open("results/dql-winrate.txt", "a") as f:
    f.write(str(won / GAMES) + "\n")
```

versus.py

```
"""
```

```
  This module tests the neural networks trained by neuroevolution and deep Q learning by makin
g them
play mancala games against each other.

  For every saved generation of the population of neuroevolution-
trained networks, the weights of the deep
Q learning network from around the same time after training are extracted and paired up with t
hat generation.

  Then, the population of neuroevolution-
trained networks play mancala games against the deep Q
learning network, and the average/best win rate across the population are saved to the results
 folder:
  - versus-average-winrate.txt
  - versus-best-winrate.txt

"""

from keras.models import load_model
from keras.models import Sequential
from keras.layers import Dense, Flatten
from keras import Input
from keras.optimizers import Adam

import multiprocessing
from joblib import Parallel, delayed

import numpy as np
import pickle

from Env import MancalaEnv

from copy import deepcopy

import random

GAMES = 1000

class NeuralNetwork:
  def __init__ (self, weights, biases):
    self.weights = weights
    self.biases = biases
  def feedforward(self, activations):
    for weight_matrix, bias_vector in zip(self.weights, self.biases):
      activations = self.sigmoid(np.dot(weight_matrix, activations) + bias_vector)
    return activations
  def sigmoid(self, x):
    return 1 / (1 + np.exp(-x))

indices = []

with open("results/neuroevolution-time.txt", "r") as f:
  for line in f:
```

```
    time = float(line.strip("\n"))
    latest = 0
    with open("results/dql-time.txt", "r") as f2:
      for i, line2 in enumerate(f2):
        if float(line2) <= time:
          latest = i
        else:
          break
    indices.append(latest)

print(indices)

open("results/versus-average-winrate.txt", "w").close()
open("results/versus-best-winrate.txt", "w").close()

def test_network(network, step):
  model = Sequential()
  model.add(Flatten(input_shape=(1, 291)))
  model.add(Dense(300, activation="relu"))
  model.add(Dense(300, activation="relu"))
  model.add(Dense(300, activation="relu"))
  model.add(Dense(7, activation="relu"))
  model.load_weights(f"dql-networks/step-{step * 10000 + 10000}.h5f")
  env = MancalaEnv(use_random_opponent=False, testing=True)
  network_won = 0
  for _ in range(GAMES):
    # Obtain the initial observation by resetting the environment.
    observation = deepcopy(env.reset())
    done = False
    while not done:
      if random.random() > 0.1:
        if env.current_player:
          action_values = network.feedforward(observation)
        else:
          action_values = model.predict(np.array([[observation]]))[0]
        action = env.get_best_action(action_values)
      else:
        action = env.get_random_action()

      observation, r, done, info = env.step(action)
      observation = deepcopy(observation)
    if env.won:
      network_won += 1
  return network_won

for neuroevolution, reinforcement in enumerate(indices):
  if reinforcement * 10000 + 10000 > 620000:
    break

  with open("neuroevolution-networks/generation-" + str(neuroevolution * 10), "rb") as f:
    population = pickle.load(f)

  # 'won' refers to how many neuroevolution won
```

```
  results = Parallel(n_jobs=12)(delayed(test_network)(network, reinforcement) for network in p
opulation)

  average_winrate = sum(results) / len(population) / GAMES
  best_winrate = max(results) / GAMES
  with open("results/versus-average-winrate.txt", "a") as f:
    f.write(str(average_winrate) + "\n")
  with open("results/versus-best-winrate.txt", "a") as f:
    f.write(str(best_winrate) + "\n")
  print(f"Generation {neuroevolution * 10} done")
```