Universiteit Leiden
The Netherlands

Bachelor Informatica

Neural Networks

for Auto Battlers

Lily Kientz

Supervisors
dr. W.A. Kosters & dr. J.M. de Graaf

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

July 31, 2022

Abstract

This thesis aims to expand on the uses of a Neural Network (NN) for the Auto Battler video game genre. NNs come in various shapes and sizes, each with their own benefits. Because it has not been clearly mapped out when to use which type of NN, we are forced to use trial and error to find the best solution. Fortunately, the decision making process can be sped up by making logical choices. To narrow down the scope of this problem, a skeleton Auto Battler has been made that has the same gameplay as some already existing Auto Battler games, such as *Hearthstone Battlegrounds* and *Super Auto Pets*. Because this skeleton game is limited in its scope, it will focus mostly on a few key characteristics such as health values, attack values, and positioning. This implementation can then later be expanded to the Auto Battler of choice.

Contents

1 Introduction

An Auto Battler is a game genre where multiple people compete against each other, simultaneously preparing their turn, after which they fight. The game that popularized the Auto Battler genre was *Dota Auto Chess* [Jon19] in 2019. To this day Auto Battler games are still widely played, each having their own quirks and requiring specific sets of skills to navigate the game. In this thesis we will look at how a Neural Network (NN) can be used to more efficiently play Auto Battler games.

Every player starts off with an empty field, no units, a set amount of health, resources, and a shop containing purchasable units. During the preparation phase players spend their resources to buy units from the shop to try and improve their field. Other actions including selling units, upgrading units and upgrading one's shop are also done during this phase. After the preparation phase has ended, the combat phase begins. During this phase every player will randomly compete against another player. The players have no input in this phase and the combat will happen automatically. The players that lose this combat phase lose some health, and if their health drops below zero they lose the game.

Figure 1 shows an example of what a game could look like. In this game, the player (green) plays against three different opponents (red). The player has two units (unit 4, unit 5) on their field (yellow) and has three units (unit 1, unit 2, unit 3) available to buy in the shop (purple), each with their own attack (bottom left) and health (bottom right) value. The shop level (blue) has impact on what units show up in the shop. The player also has an amount of resources (orange) to spend for using actions.

The NN that will be utilised in this thesis uses Reinforcement Learning (RL) [SB18, Pla20]. A NN usually needs a large data set to train itself. However, if no adequate dataset is available, RL can be applied. We let the NN play the game in order to make training data in this case. With this strategy we approximate the best action in every position and, depending on whether the player did a good or bad action, the NN will train itself accordingly.

The goal of this thesis is to create a framework to get the best results for a skeleton version for an Auto Battler. This framework can later be expanded and applied to an Auto Battler of choice. Even though this is only a framework, a NN has many parameters and can be built in many different forms. This makes it impossible to try every possible combination of NN. This is why we will mostly talk about the NNs that use RL.

Section 2 outlines the rules of the game; Section 3 discusses related work; Section 4 explains how RL is applied; Section 5 describes the experiments and their outcome; Section 6 concludes and discusses further research. This research was conducted as a bachelor project at LIACS, supervised by W.A. Kosters and J.M. de Graaf.

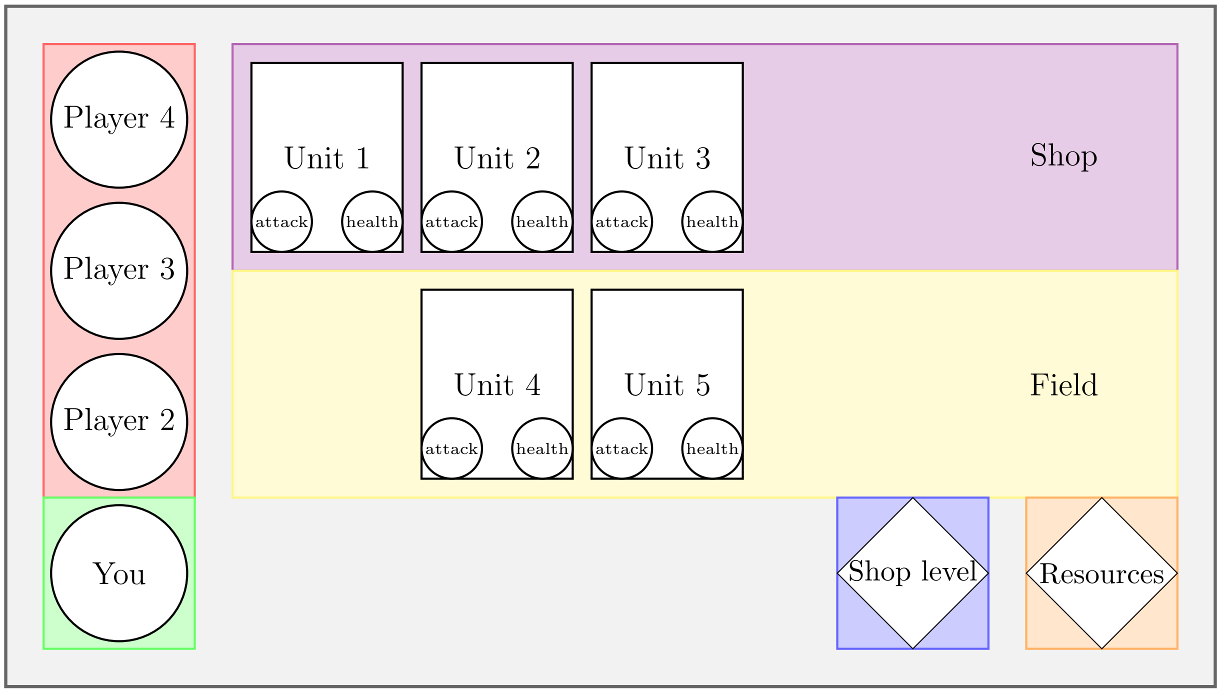


Figure 1: Sample game state from the perspective of a player.

2 Rules of the Game

This section sets forth the environment of the game and the different actions a player can take. An AUTO BATTLER is a game most commonly played on a computer which makes it easy to match players of the same skill level. This enables the usage of a wide variety of play styles, making each game unique. Additionally, it is easy for players to hop into a game whenever and wherever they want. These games generally last somewhere between 5 to 50 minutes, depending on which AUTO BATTLER is being played.

Start of the game First, a group of players is selected to play against each other. Every player has a set amount of starting health, an empty field, and some starting resources.

- $P = \{P_1, P_2, \ldots, P_N\}$, the players where N usually is 8.
  - $H_i \in \mathbb{Z}$ $(1 \leq i \leq N)$, health of player i. Starts at 40.
  - F, field; sorted list that can contain up to 7 units. Initially empty.
  - $R_i \in \mathbb{N}$, resources of player i. Starts at 3 and increments by 1 every turn, up to 10 resources.

The players have a shop to buy units from. The units shown in this shop are usually randomly determined from a predefined pool. Once the shop reaches higher levels, stronger units will be added to this pool.

- S_i, shop that contains units for P_i. Every player has their own shop. Number of units starts at 3.

- $L_i \in \mathbb{N}$, level of shop per player. Starts at 1, and can go up to 6.

- $U = \{U_1, U_2, \ldots, U_M\}$, where unit U_j consists of:
  - $a_j \in \mathbb{N}$, attack. The amount of health it will subtract from the unit it attacks or is attacked by.
  - $h_j \in \mathbb{Z}$, health. If $h_j \leq 0$ the unit is destroyed for that battle.
  - $\ell_j \in \mathbb{N}$, level of unit.

See Figure 2 for a visualisation of the starting position from the view of a single player.

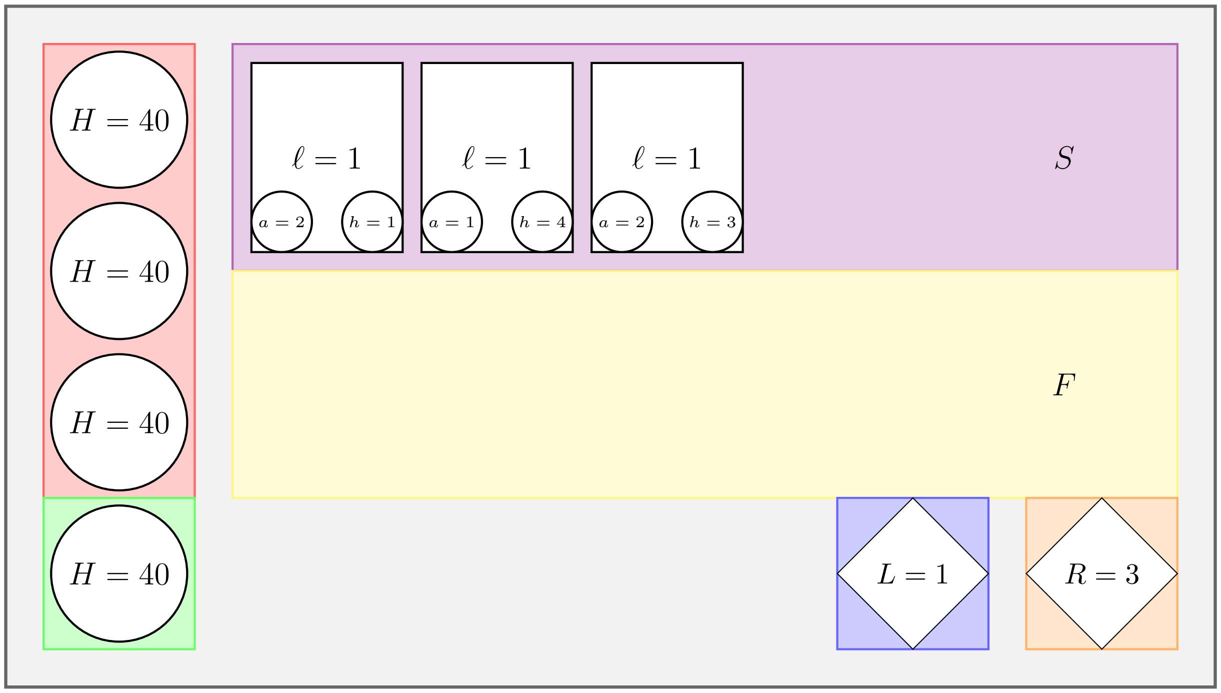


Figure 2: Sample start position from perspective of a player.

Planning phase This phase will last until everyone has passed or when every player has run out of time. During this phase, players attempt to get the best game state with their limited resources and information on their opponents. This can be done through the following actions:

- Buy U from S for R (Figure 3).
- Rearrange F (Figure 4).
- Sell U from F to gain 1 resource (Figure 5).
- Refresh all U's in shop for 1 resource (Figure 6).

- Level-up S for n resources, where n is decided by L, with a discount of 1 for each consecutive turn on current L (Figure 7).
- Freeze U, U in S do not get refreshed at the start of the next preparation phase.
- Pass the turn.

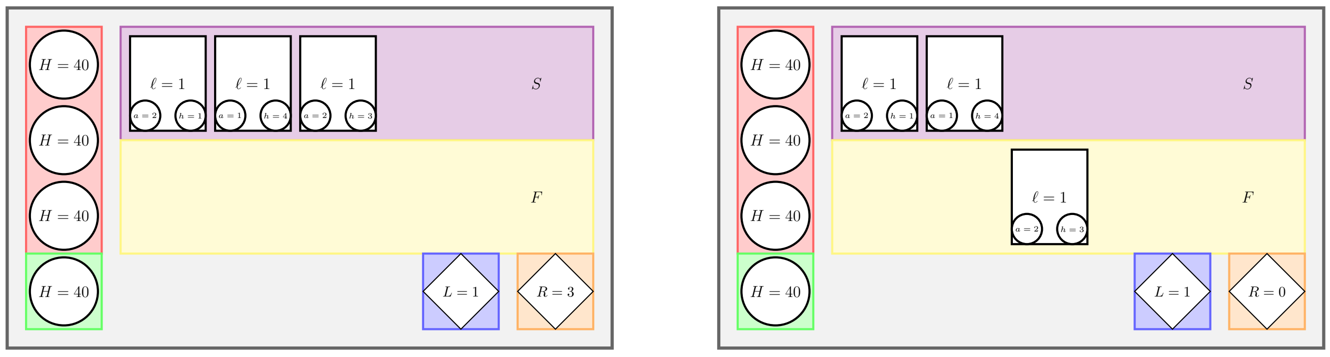
Combat phase In this phase the players have no interaction with the game. Every player is set up against an opponent, and a combat between their units will happen automatically. This fight is played out according to a specific set of rules depending on the Auto Battler. In our case the the following will happen.

First, the starting player is determined by the player with the most units. We will call this player "player one", the other player "player two". In case of a tie, the starting player is randomly selected.

Next, player one's first unit will randomly select and attack one of player two's units. When a unit is attacked, both the attacking and the defending unit will lose health points equal to the opposing unit's attack value.

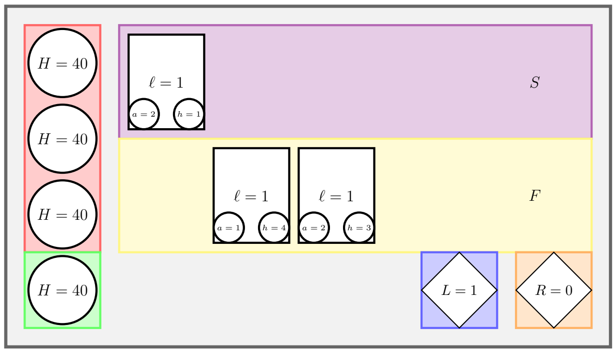
After this, player two's first unit will randomly attack one of player one's units. Each player's units will attack back-and-forth in this manner. Once all of a player's units have attacked, their first unit will attack again. The first player to lose all their units loses this combat phase. The winning player deals damage to the losing player, equal to the winning player's shop level plus the level of every surviving unit.

Winning the game After multiple cycles of planning phases and combat phases, players will slowly start to lose health points. Once a player reaches zero health points they lose and are forced to leave the game. Different Auto Battler games will have different definitions on when a game is won. Some will require you to win a certain number of games, but most define a win as still having health points while half of the other player's health points are depleted.

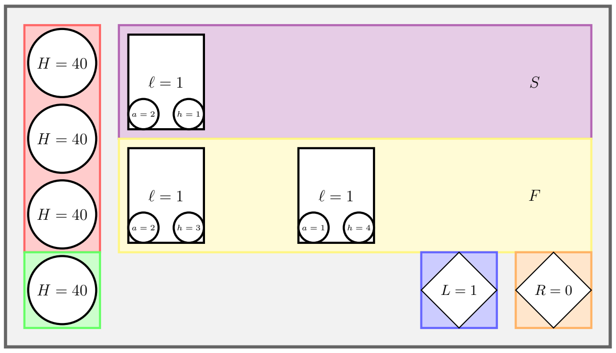


(a) before (b) after

Figure 3: Sample position after a unit is bought.

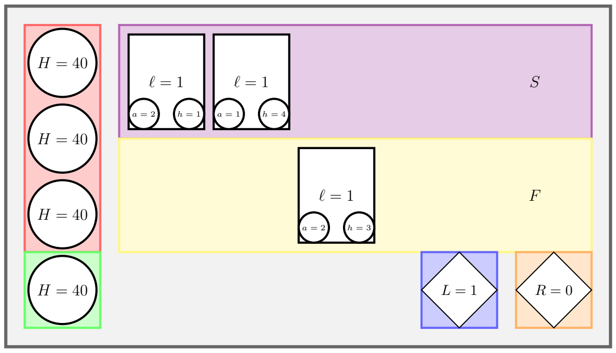


(a) before

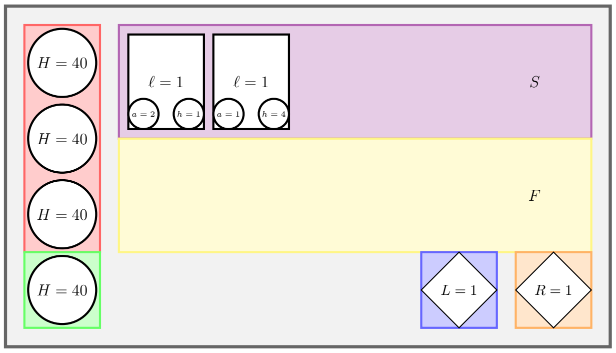


(b) after

Figure 4: Sample position after units are rearranged.

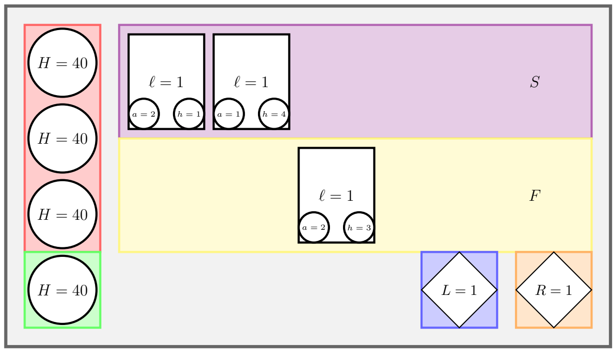


(a) before

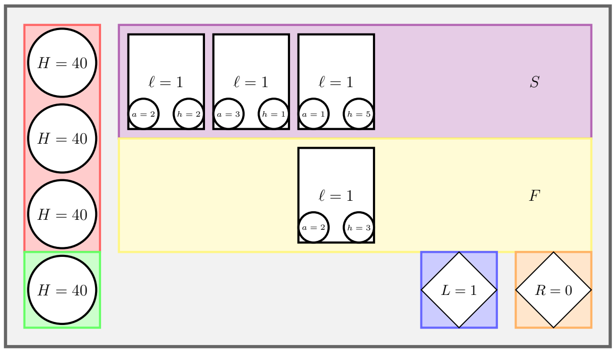


(b) after

Figure 5: Sample position after a unit is sold back to shop.

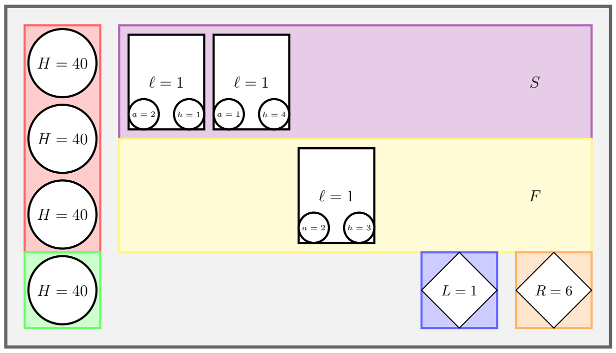


(a) before

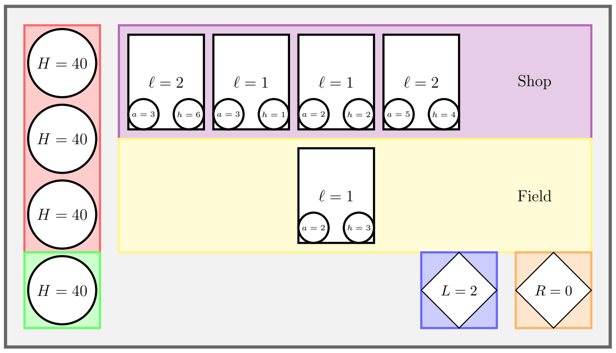


(b) after

Figure 6: Sample position after the shop is refreshed.



(a) before



(b) after

Figure 7: Sample position after a player levels-up the shop and refreshes it.

3 Related Work

Since Auto Battler games are relatively new, there have not been many papers covering this subject. However, there is still a small body of work to build upon. In the paper "Hearthstone Battleground: An AI Assistant with Monte Carlo Tree Search" [ZJDR21], Zolboot et al. built an agent to assist the player by showing them their optimal gameplay. This was done in the programming language Python [Pyt]. They constructed a *Hearthstone Battlegrounds* simulator containing pools of two players. Monte Carlo Tree Search (MCTS) [BPW+12] was implemented to find the best move using selection, expansion, simulation, and backpropagation. There is however one problem with MCTS and Auto Battler games: during later turns when the player is given more resources, the agent will take an exponentially longer amount of time to find the best path. Because of this, their model focuses only on battles up to turn 6. This brings up a major flaw with their agent. Normally games of *Hearthstone Battlegrounds* will take somewhere around 10 to 14 turns. Only being able to play 6 turns will drastically impact the play style, favoring quick benefits over late-game strategy. Therefore, this paper is not representative of normal gameplay. As discussed in Section 4, a solution to this long execution time could be using a Neural Network (NN) instead of MCTS. A NN will approximate what action should be taken instead of traversing a tree, making it more applicable to problems with many different game states.

In the paper "Lineup Mining and Balance Analysis of Auto Battler" [XCZW20] Xu et al. attempt to solve balance adjustments for Auto Battler games. First, a lineup measurement system is made: a lineup evaluation model uses a full connection NN to determine the win rate. Second, a lineup mining algorithm is used to test the different constraints of the problem. This is done with a Genetic Algorithm, because it can quickly complete lineup mining without training data, while also being able to handle multiple constraints, which makes it better than MCTS. In this second step, the strongest Field is always chosen. The authors mention that obtaining optimal lineup data is cumbersome, so a NN was difficult to use. Last, statistics are used to determine how to adjust the individual units.

As mentioned above, one of the reasons a NN was only used for lineup evaluation is because of a lack of data. To let a NN train itself, it requires quite some data to allow it to tune itself to a specific problem. However, this data is not always available. A solution to this would be using Reinforcement Learning (RL) [AABB15]. With the use of RL it is not necessary to obtain data beforehand, since training data will be generated by playing the game as the NN learns. How this exactly works will be explained further in Section 4.

4 Reinforcement Learning

Reinforcement Learning (RL) [SB18, Pla20] is applied to a problem when there is no available data set to teach an agent how to interact in an environment. At first, only a few things are known about the environment: which state the agent is in, and the rewards or punishments for certain states and actions. When the value for every end state is known, backtracking can be applied to calculate the maximum value for the previous states. This way a map can be created to navigate through the environment by taking the best path. There is a downside, however. For some problems, the number of states is too large to efficiently calculate every possible path. Since this is the case here [ZJDR21], a Neural Network (NN) will be used to learn about the states of the environment, which will remove this bottleneck.

4.1 Bellman Equation

To calculate the value of a given state, the *Bellman Equation* [BI89] can be used. This equation is used when the objective for an optimization problem is to minimize travel time and cost, while maximizing profits. It does so by knowing the values of some specific states and calculating the neighbouring states via the following equation:

Equation

$$V(s) = \max_{s \xrightarrow{a} s'} (R(s,a) + \gamma V(s'))$$

where:

$V(s)$ = the value of a state s
$R(s,a)$ = reward for performing action a in state s
γ = discount factor
$s \xrightarrow{a} s'$ = transition rule between states s and s' using action a

Game state A state is the position of the game at a given moment in time. Usually when a single parameter changes, such as when one of the actions during the planning phase is carried out, the game will find itself in a new state. One could instead choose to calculate states between the end of every phase, if this works better in the specific optimization problem. Every state has a value that can be calculated if a reachable end state value is known. Figure 8 is a representation of what such a game could look like, where we have a starting state (blue), some unknown states (black), and a few end states (green and red).

Rewards Rewards are assigned to the agent for entering specific states. This could be for winning a game or knocking out an opponent. These rewards can be negative in case of a bad

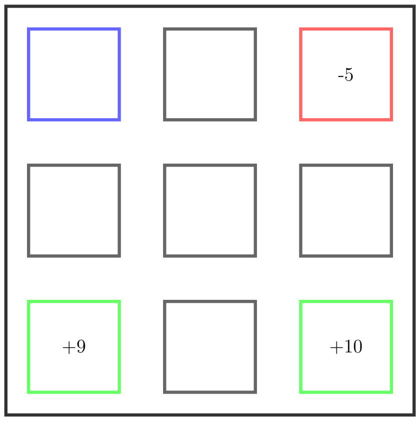


Figure 8: Game states with rewards.

outcome. Figure 9 shows how to traverse the collection of states to get to the best possible reward if the rewards are end states of the game. The rewards which we use in the skeleton game are:

- $10 - 2\frac{1}{2}x$ where x is the number of players that are still alive when one loses the game.
- $-2\frac{1}{2}$ for finishing in the bottom half.
- $\frac{1}{10}x$ where x is the amount of damage dealt.
- $-\frac{1}{10}x$ where x is the number of health points lost.

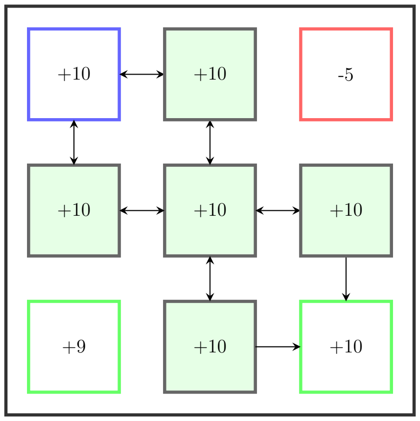


Figure 9: Best path with Rewards.

Discount factor A discount factor is applied to prevent alternating between two states and ensures the agent converges to a solution. The shortest path between two routes that end with the same value will in this case be taken. This will lead to less computational power being spent for reaching a specific state. As shown in Figure 10, the best path now leads to a different end state. Since this end state is closer to the begin state, the leftover value after traversing the new path is bigger than when one would take the previous path.

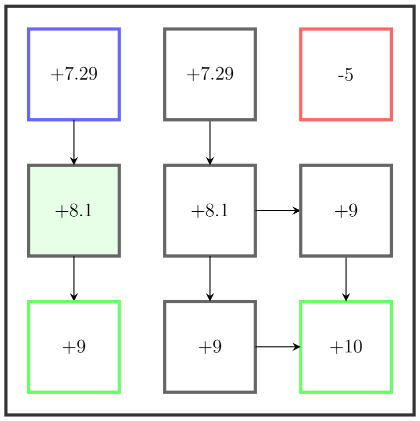


Figure 10: Best path with rewards and discount factor of 0.9.

4.2 Neural Network

The main purpose of a Neural Network (NN) is to approximate a function when this function is unknown. It does so by feeding input values to the input neurons of the NN, then letting one or multiple layers of activation functions and weights transform those input values to one or multiple output values.

A NN can be built out of many neurons, each receiving their input from previous neurons or receiving input from outside the NN. A network can be fully connected, which means that every neuron of a layer will give its output to every neuron of the next layer. A network can also be partially connected, where only some neurons receive this output value. Sometimes a NN has more layers than there are neurons per layer, but other times it is the other way around. There is no single NN that fits every solution. Every problem requires a specific architecture to get the fastest working NN that gives the most precise answers.

A single neuron can have multiple parameters. There exist many different activation functions, each serving different purposes. Additionally, when a neuron is told to re-evaluate its weights, the learning rate and learning momentum are considered when deciding on how much the weight needs to be changed.

Everything mentioned above makes it nearly impossible to construct an optimal NN, which also means it is impossible to get a NN that gives perfect answers. This is why a NN will always be approximating an outcome. One has to evaluate the outputs themselves to find if the NN is up to their standards. Adding more neurons to a network will most likely make it better at approximating, but after a certain number of neurons the computational costs will outweigh these benefits. The computational costs mostly lie in the training of the NN. Once a NN is fully trained you one save the parameters to load these somewhere else without having to re-train it.

4.2.1 Types of Neural Networks

A *value-NN* (Figure 11), also known as Q-network, allows the agent to evaluate every state without having to traverse the whole set of states. To train this type of NN, one takes a randomly generated game state as input and lets the NN calculate the output. After this is done, the real state value according to the *Bellman equation* is calculated via its neighbouring states. Lastly, the NN gets some minor tweaks to better calculate the correct state value. Close states will most likely have similar values. Knowing this, the NN will approximate the value in every state while not having to traverse all of them. If one keeps feeding new states to the NN, it will eventually approximate every state correctly. After the NN is fully trained, the agent can walk the best path by only looking at its own state.

A *policy-NN* (Figure 12) evaluates which action to take in every state. This type of NN skips calculating the value of the state, and goes straight to outputting the actions the agent has to take. To get the actions on which one trains, we first take a random path. Once at the end of this path one can calculate, via the *Bellman equation*, for every action, if it was a good or bad one. These policies with their respective states can then be fed to the NN, after which the NN is slightly tweaked to make good policies more likely and bad policies less likely to occur. Since close states will probably have similar policies, just like in the *value-NN*, we do not have to train every state to get a good approximation.

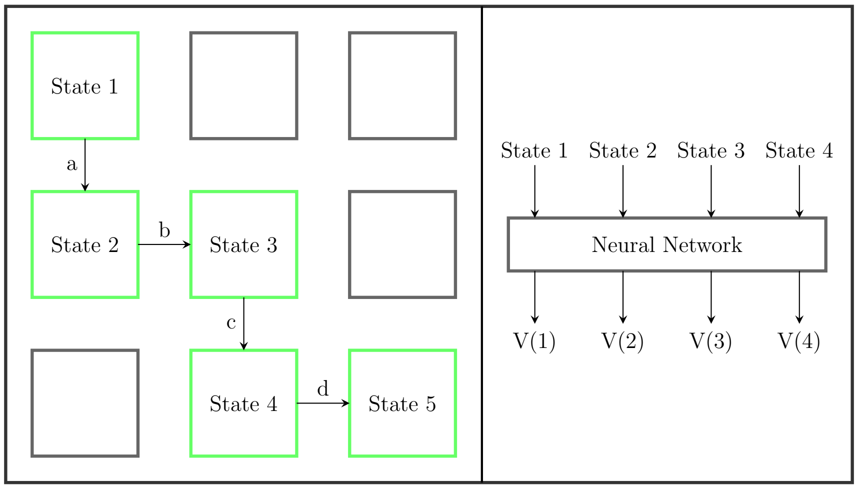


Figure 11: Value Neural Network.

Another choice that needs to be made is deciding whether the NN is deterministic or stochastic. A Deterministic NN will always find the best current known path, but might get stuck in a local optimum. This is called exploitation. A Stochastic NN will sometimes make random decisions and will not always get the best solution. However, it will sometimes find even better solutions than the ones previously found because it is increasing its search range. This is called exploration. A NN can be implemented in such a way that it can switch between exploitation and exploration to find an optimal solution.

4.2.2 Inner workings of a Neural Network

The most simple NN is the one that consists of only a single neuron. The weight of an input shows the neuron how important that input is. A weight further away from zero means more of the neuron's output will be dependent on this input, a weight closer to zero means this input is less

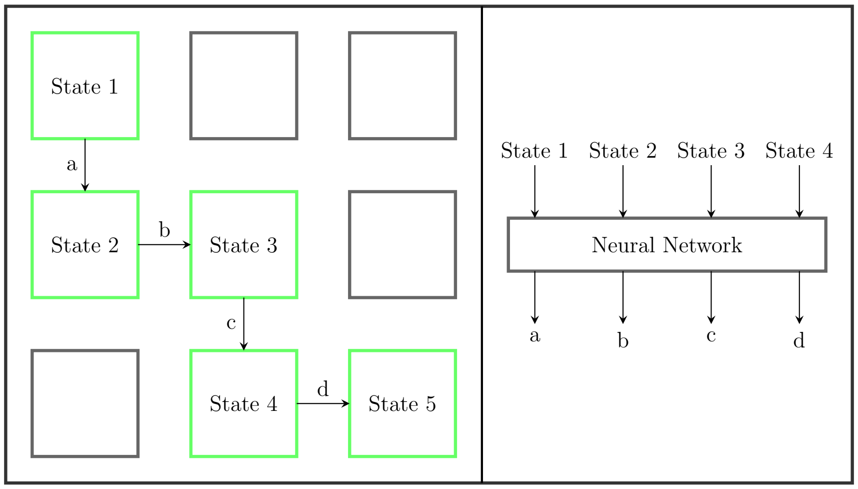


Figure 12: Policy Neural Network.

important, as seen in Figure 13. The weight of 1 is an identity function while the weight of -1 is the complement of an identity function. After the neuron gets its input, an activation function is used to normalize the output. To do this, we use a hyperbolic activation function which can be seen in Figure 14. If a positive input is further away from zero, the output of this neuron will get closer to 1. An input below zero will converge to -1 instead. This output can be interpreted to the programmer's liking.

The NN is initialized with random weights, which means that before any training has been done, random outputs will be given. To get the output one wants to have, one first needs to train the NN. To do this, a big data set with inputs and their respective output is required. The input is fed to the NN and the output is checked. Depending on the error between the given output and the correct output, there are two things that can be changed to increase the accuracy of the output:

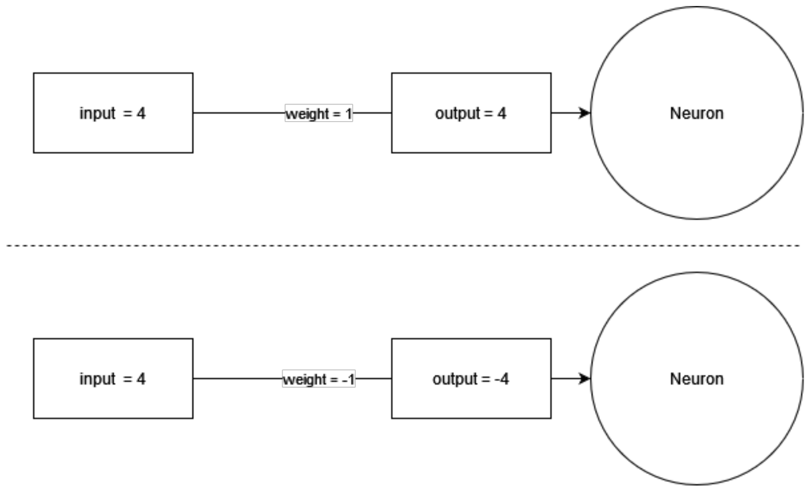


Figure 13: Simplest NN with input.

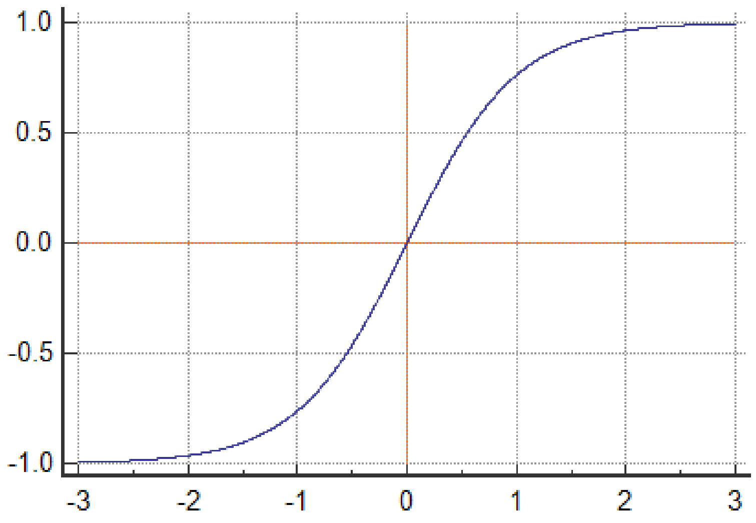


Figure 14: Tangent hyperbolic (tanh) activation function.

1. The weights can be increased or decreased depending on the output values given by the neurons in the previous layer.

2. The outputs of the previous layer can be modified. To do this one needs to go over the same two steps until one reaches the first layer.

By starting in the last layer and working its way backwards, the NN uses backpropagation [Gra13]. Every weight is changed slightly to get closer to the desired output. The amount the weights are changed depends on the hyper parameters combined with the desired impact of the change. One only wants to nudge the output in the right direction, as to not overshoot the desired outcome. This results in an output closer to the correct output the next time a similar input is given to the NN.

Since this is only a very basic NN, it will only be able to solve a few linear problems, such as checking if the temperature is above a certain degree. To add more functionality, more things need to be added. First, a bias input, which is needed to shift the output without it being affected by any other inputs. Second, more neurons can be added to make the NN approximate more complex functions. The number of neurons in the NN and the architecture of how these neurons are connected are highly dependent on the problem.

4.3 Neural Network for main choices

As discussed in Section 3, in the Auto Battler genre there exist complex games and these can have too many game states to calculate the optimal path with a Monte Carlo Tree Search approach. Because of this, we will use a NN to approximate which action to take in every state. To do this, we first need to define what we use as input for this NN and what we desire as output.

Neural Network input Our skeleton game distinguishes between seven different main choices. These choices can easily be defined as policies, which is why we will use a policy-NN. This is further explained in the planning phase of Section 2. The number of neurons the first layer of the NN has, is decided by the amount of information known to the player. This is also further expanded upon in Section 2. Every input neuron can be fed one floating point number. In order to make this compatible with the game, we first need to decide how we transform our information into the right numbers before we can feed it to the NN. We use the information in the following way:

- The first piece of information that can be fed to a neuron is the number of players left in the game. For each player, we need a neuron that takes the number of health points as input. We also need a neuron for the number of resources one has.
- The information about the shop can be fed into two neurons, one for the shop level and one for the resource cost to level up the shop.
- Every unit visible to the player requires three neurons as well: one for unit level, one for attack, and one for health.

Since the NN is able to upscale or downscale the input by itself, it is not necessary to change any of the visible values. It will, however, take the NN longer to train due to the small changes when backpropagating need to be made even smaller.

Neural Network output The NN needs to have seven output neurons, for the seven different actions of the corresponding planning phase as described in Section 2. When an output neuron has a higher value, the possibility that action is taken will increase. The NN is stochastic at the start of a training session. This means that if an output value is 60% of the total output value, the agent has 60% chance of choosing that action. Once the NN has trained for some time, it will slowly change into a Deterministic NN by increasing chances of picking large output values and decreasing chances of picking small output values. The largest output value will always be chosen in the end. A visualisation of this NN can be seen in Figure 15.

4.4 Neural Network for sub-choices

A NN works best when only making one decision at a time; a single NN can sometimes not be sufficient when making a decision requiring multiple steps. One way of making these additional choices is by making a second agent: this agent can then be implemented to make smart or random decisions. This agent can also make use of a second NN. This can be implemented in the following manner:

Neural Network input For position swapping, we started by only using the information on one's own field. As mentioned above, this only requires three input neurons per unit.

Neural Network output Choosing outputs can be done in several different ways:

1. Having one output neuron per unit and letting the units swap if their output neurons have the highest output values. The output value for each neuron will signal whether their unit is in the right position. An output of $+1$ means that that the unit wants to be swapped while an output of -1 means that the unit does not want to be swapped.

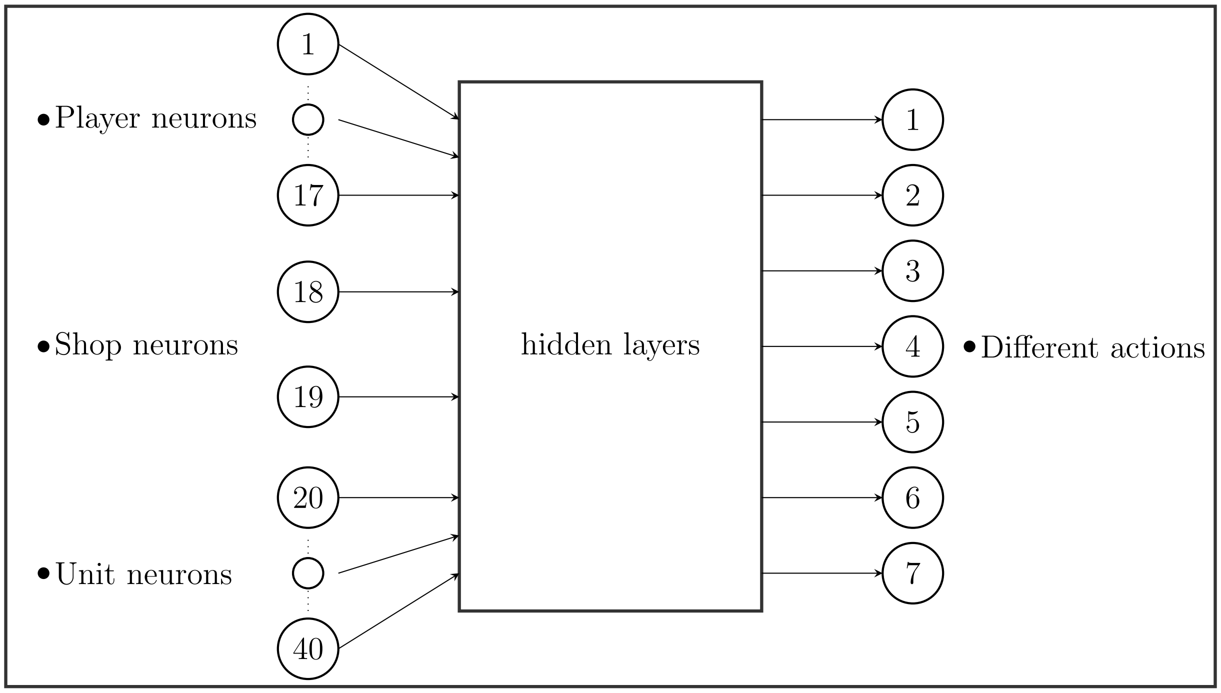


Figure 15: Neural Network for main choices.

2. Having one neuron per unit and swapping the unit with the highest output with the unit with the lowest output. Output values close to zero mean that a unit is in the right position. Output values closer to -1 mean that the unit wants to move further back in the attack order, and output values closer to $+1$ mean that the unit wants to move up in the attack order. The issue with this style of reading outputs is that one cuts the range of readable outputs in half. The NN will be less precise when it is trying to converge to a solution.

3. Having two neurons and dividing the output value between the units. The unit will be selected if the output value is part of that unit. This has the same issue as the previously mentioned option, but by cutting the output value in smaller pieces it will be even more difficult to accurately approximate the best solution.

We start by testing option 1, because we want the most precise actions. If this takes too much time we can try out option 2 and 3 to see how they compare to each other in computational time and accuracy. A visualisation of the first NN can be seen in Figure 16.

5 Experiments

We divide our problem into two parts for the experiments. Part one will contain every sub-choice that is made when one of the main choices is picked. The second part consists of playing a full game. This can again be divided into two different kinds of games where one game uses a second Neural Network (NN) to make sub-choices and the other makes smart decisions for sub-choices without a second NN. Whenever a NN is implemented, an already existing library *Tensor Flow*

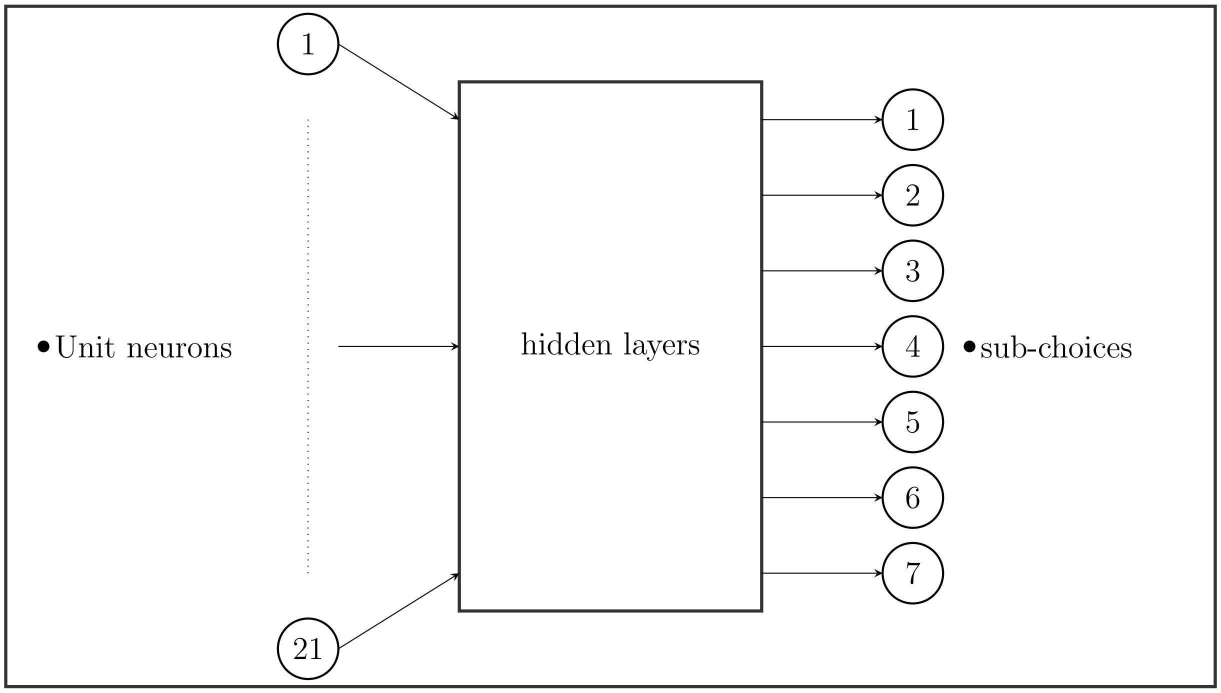


Figure 16: Neural Network for sub-choices.

[AABB15] can be used to support building the NN. However, this was not done in this research, since we wanted to grow more familiar with the inner workings of a NN. That is why we built our NN from the ground up.

To evaluate the NN, an extra function is written that calculates the Root Mean Square Error (RMSE) [MWL17] for the output neurons. The RMSE along with the number of iterations it will take to reach a minimum value is plotted. The NN, the Auto Battler, and the evaluation function are all written in C++.

5.1 Sub-choices

To test the NN for the different sub-choices, we isolate the game states in which these actions appear. To do this, we generate random states in which we force a NN to make one of the seven specific actions. For each sub-choice we use "random grid search" [BB12] to find the hyper parameters that gives us the best solutions. The grid of hyper parameters that we search with consists of the following:

- Number of units to choose between: $[3, 5, 7]$
- Learning rate: $[0.01, 0.1, 0.5]$
- Learning momentum: $[0.01, 0.1, 0.5]$
- Number of hidden layers: $[1, 3, 5]$

- Number of nodes per hidden layer: $[3, 6, 9]$

There are many different permutations, therefore we will only mention/plot some of the more interesting ones.

Which unit to buy To determine which unit needs to be bought, the output neuron with the highest output value is selected. As input we only need to connect the units available in the shop to a neuron. However, if we want the NN to adapt to more specific situations, we can add more input neurons to input more data visible to the player. If the NN would randomly assign -1 and $+1$ to the output neurons, on average half the outputs given will not give the desired output, and the other half will. This is because every neuron has 50% chance to guess correctly. If this were to happen, the RMSE would be 1.414. Any RMSE below the value of 1.414 means the NN has found a better strategy, while any RMSE above this value means the NN found a bad strategy on purpose. In case of this bad strategy, we can invert the output of the neurons to get to a better strategy. The NN selecting one unit at random gives the following RMSE depending on how many units it can pick from:

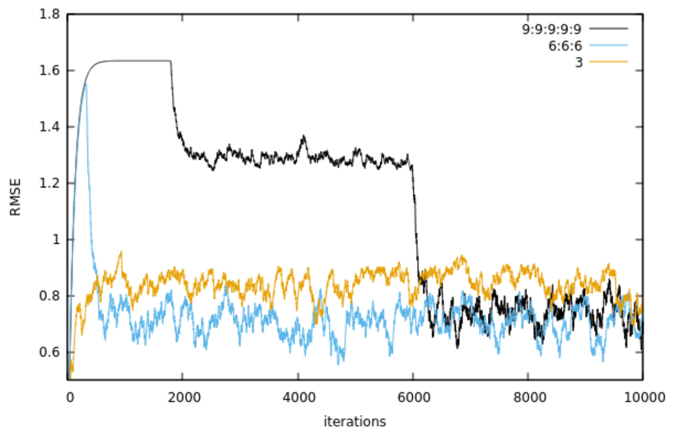
- 3 units: 1.0886
- 5 units: 1.0119
- 7 units: 0.916

Getting a lower RMSE than the above mentioned values would mean the NN has found even better solutions. In Figure 17 a subgroup of the grid of hyper parameters is shown to give a general idea on how our NN is evaluated. Figure 18, Figure 19 and 20 show the change in RMSE over 10,000 iterations during training of the NN. The graphs are made with gnuplot [Wea].

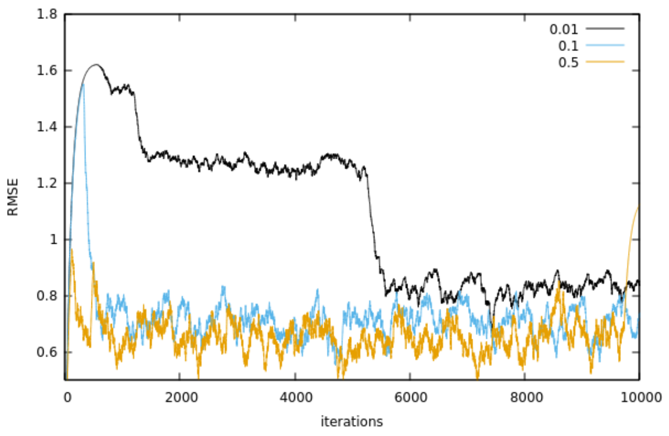
RMSE						
learning rate	learning momentum	Layers	Neurons per layer	3 units	5 units	7 units
0.1	0.1	1	3	0.78	0.65	0.70
0.1	0.1	3	6	0.65	0.63	0.70
0.1	0.1	5	9	0.67	0.63	0.72
0.01	0.01	3	6	0.82	0.80	0.89
0.5	0.5	3	6	1.11	0.81	1.02

Figure 17: RMSE for buying units.

Which unit to sell In this section, the NN has to select one unit on the field that will be sold back to the shop. The unit is selected by being connected to the output neuron with the highest value. Similar to buying a unit, the worst RMSE is 1.414. The RMSE threshold for randomly selecting a single unit to sell are the same as for randomly picking a single unit to buy. In the same manner, Figure 21 shows the RMSE values after training for a set amount of iterations. Since buying and selling units carries similar results, we will not plot the way the NN learned over time.

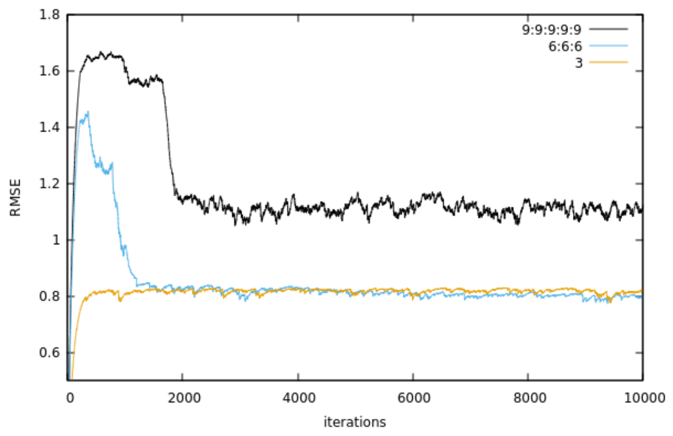


(a) NN architecture.

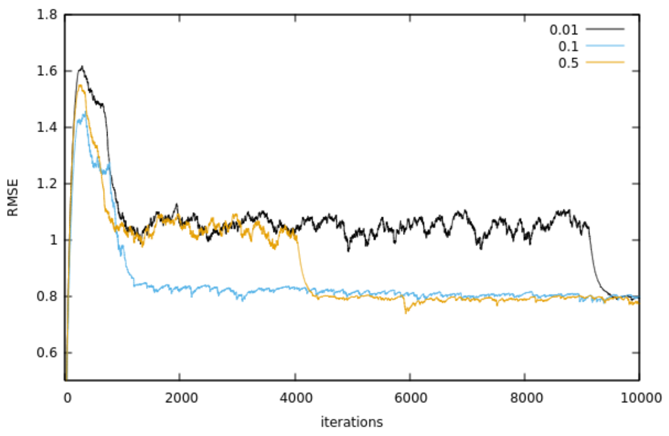


(b) Learning rate.

Figure 18: RMSE when learning for buying; out of 3 units.

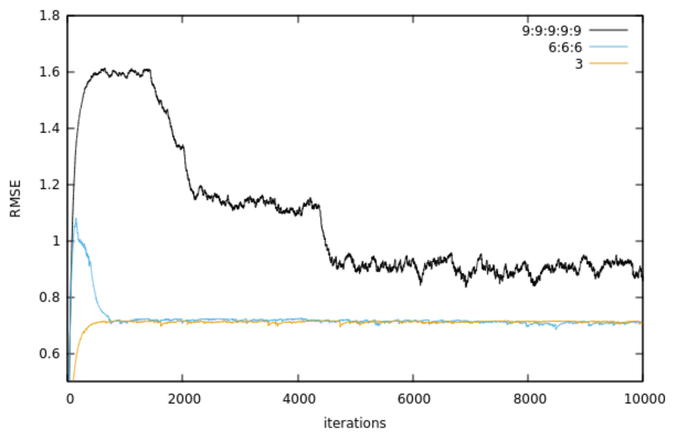


(a) NN architecture.

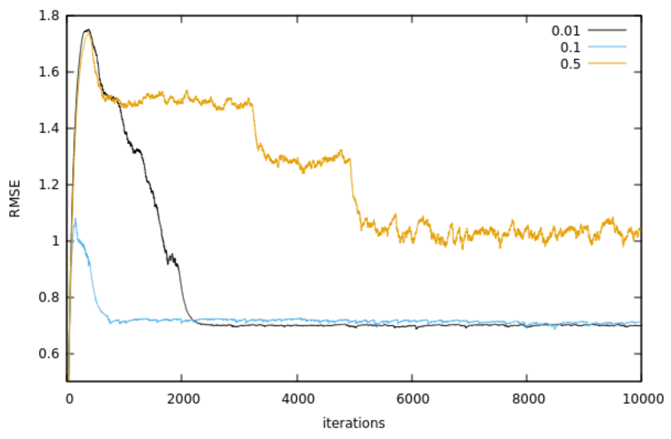


(b) Learning rate.

Figure 19: RMSE when learning for buying; out of 5 units.



(a) NN architecture.



(b) Learning rate.

Figure 20: RMSE when learning for buying; out of 7 units.

RMSE						
learning rate	learning momentum	Layers	Neurons per layer	3 units	5 units	7 units
0.1	0.1	1	3	0.62	0.81	0.70
0.1	0.1	3	6	0.70	0.65	0.69
0.1	0.1	5	9	0.73	0.83	0.70
0.01	0.01	3	6	0.84	0.80	0.82
0.5	0.5	3	6	1.12	0.90	1.01

Figure 21: RMSE for selling units.

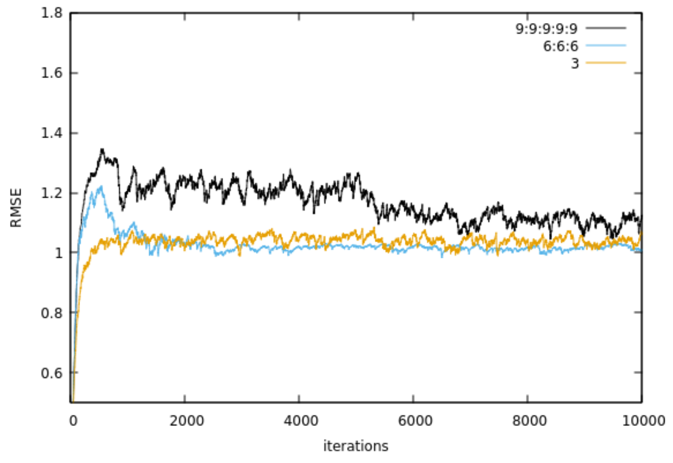
Position swapping In this part, we require the NN to pick two units on the field. The two units selected by picking the highest output values of the NN should give the highest win percentage. Just like in the last two cases, randomly assigning minus one or plus one to the output neurons gives an average RMSE of 1.414. Randomly assigning two neurons the plus one output and the rest a minus one output gives the following RMSE:

- 3 units: 1.0886
- 5 units: 1.2889
- 7 units: 1.2289

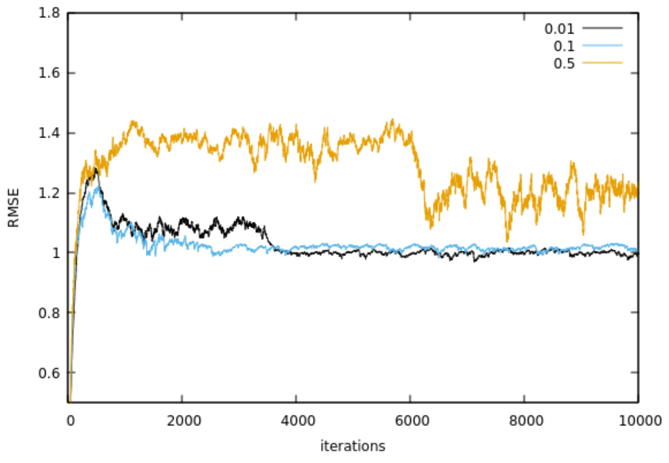
In Figure 22 more RMSE values are shown when swapping units. Figure 23, Figure 24 and Figure 25 show us how the NN developed over time.

RMSE						
learning rate	learning momentum	Layers	Neurons per layer	3 units	5 units	7 units
0.1	0.1	1	3	1.04	1.09	0.93
0.1	0.1	3	6	1.00	0.97	0.92
0.1	0.1	5	9	1.02	1.07	1.29
0.01	0.01	3	6	1.00	1.39	0.87
0.5	0.5	3	6	1.21	1.41	1.41

Figure 22: RMSE for swapping units.



(a) NN architecture.



(b) Learning rate.

Figure 23: RMSE when learning for swapping; choosing between units.

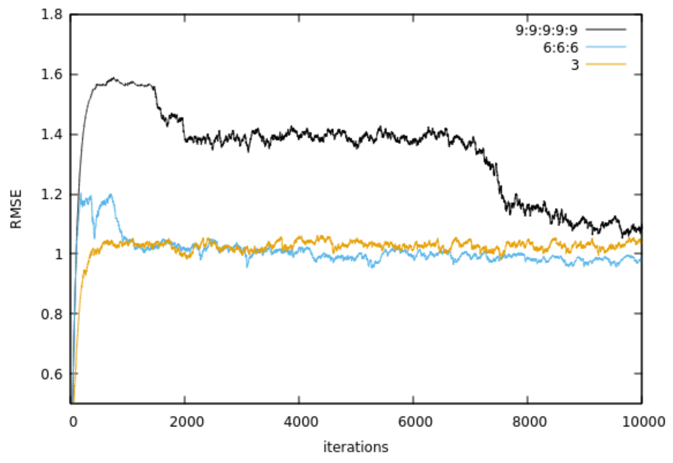
5.2 Main choices

To train the NN in playing a complete game, we first let the agent play against random players. If this is successful, we let the agent play against other agents that use a 'smart' play style. The smart agent can be implemented by applying some basic rules to the random player:

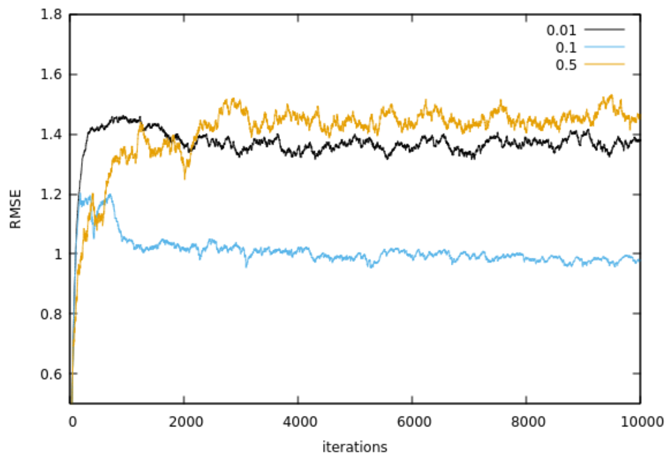
- Only sell units if the maximum amount of units on the field has been reached.
- Always level-up the shop if the last battle was won and there are enough resources.
- Only spend resources to refresh the shop if one does not have enough resources to buy a unit.

If the NN beats this agent, we can let the NN play against agents that also use a NN to make their choices.

Random opponents Playing an entire game is very complex and requires many nodes in the NN. As seen in Figure 26, even when the NN gets ten times more iterations than it did for one of the sub-choices, it still does not converge to a good solution. The NN got slightly better around

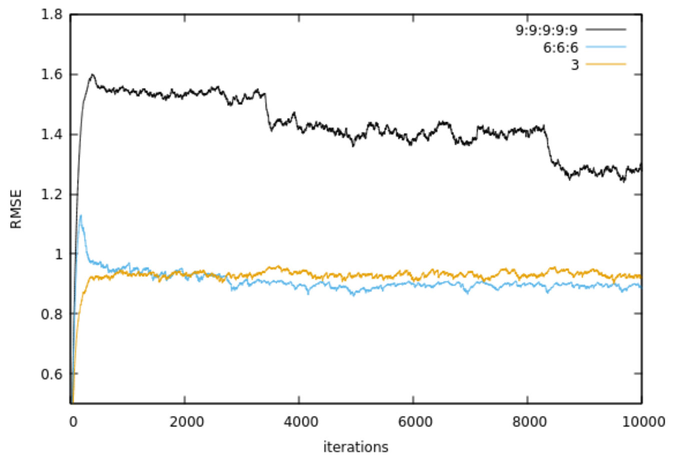


(a) NN architecture.

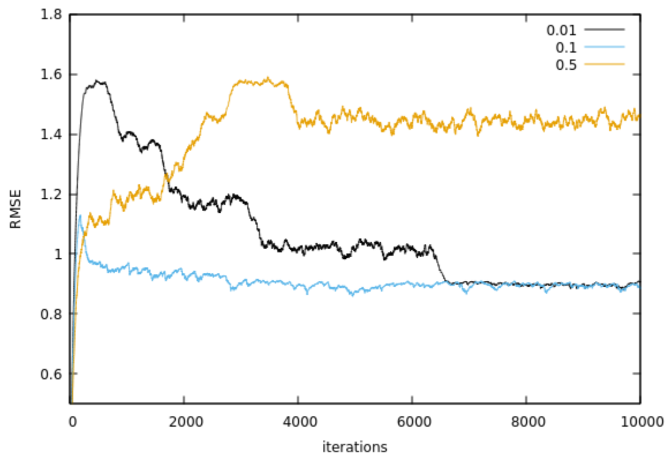


(b) Learning rate.

Figure 24: RMSE when learning for swapping; choosing between 5 units.



(a) NN architecture.



(b) Learning rate.

Figure 25: RMSE when learning for swapping; choosing between 7 units.

25,000 iterations during a single training session, but even this small improvement does not occur every time the network is trained. Sadly, because it is beyond of the scope of this bachelor project to add more efficient ways to let the NN learn, this is where we decided to stop our experiments.

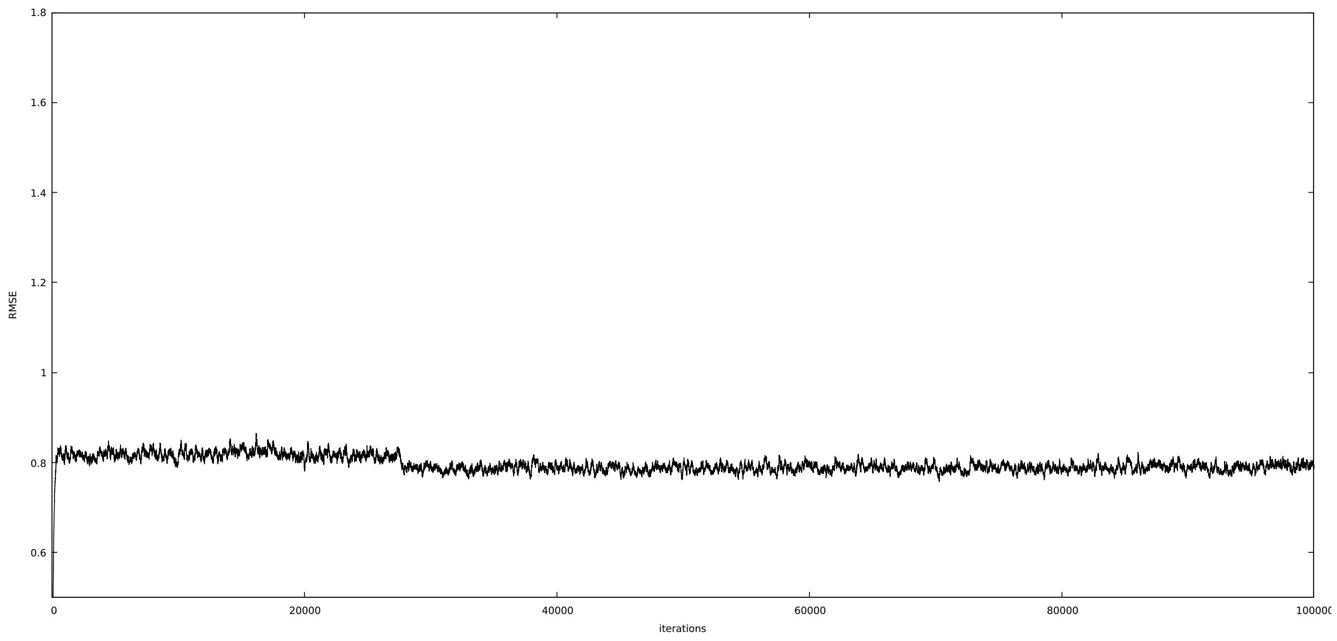


Figure 26: RMSE for playing the game.

6 Conclusions and Further Research

This thesis lookes into the usage of a Neural Network (NN) to improve gameplay in a skeleton build of an AUTO BATTLER as an alternative to using Monte Carlo Tree Search. The NN is connected to the skeleton AUTO BATTLER and tells us the actions with the highest chances of winning. It does so by utilizing Reinforcement Learning and the *Bellman equation*. Experiments are conducted to calculate the effectiveness of the NN.

Firstly, for sub-choices we can conclude from Figure 18, 19, 20, 23, 24, and 25 that the NN will most of the time converge to a better solution than when assigning random values. It also functions better than when one would assign "smart" values. Only when the learning rate and learning momentum in which the NN learns are set to a value that is too high, the NN has problems with learning. Setting the learning rate and learning momentum to a lower value will increase the accuracy on more complex problems, but also vastly increases training time.

The second conclusion we can make is that having more neurons in the NN does not necessarily mean it can better approximate the right action. When a problem is complex, more neurons can better approximate that complexity. However, at a certain point the extra neurons will only add to the total approximation error caused by the neurons. This is why it is important to start building your network with as few neurons as possible and adding more if one thinks the extra training time is worth the extra accuracy.

A third conclusion we can make is that a less complex architecture for the NN results in a more stable Root Mean Square Error (RMSE). Having fewer neurons or layers means there are fewer small changes which adds to a smaller overall change in the NN.

For playing a full game we only see a small improvement in Figure 26, which costs a lot of iterations. The NN gets stuck more easily because of the number of nodes it requires to function. A NN that is not fully connected but has a specific layout could make it more easy to cater to the needs of this specific problem, which might help in finding better solutions more quickly.

6.1 Further research

In future research the sub-choices can be trained while training the main choices to more accurately learn multiple-turn strategies. This will increase training time, but hopefully also increase accuracy when playing a game. Another thing to expand upon is building a NN with already existing libraries such as TensorFlow [AABB15] instead of building a NN from the ground up, which was done in this research. Using TensorFlow or other libraries vastly expands the flexibility and utility of the NN, allowing it to better adapt to its environment. The last thing to improve upon is the training time of the NN. This can be done by having better hardware. This research was conducted on a single CPU on a standard desktop computer. Being able to train multiple agents at the same time across multiple CPUs will decrease training time and leave more time to tweak the parameters.

References

[AABB15] M. Abadi, A. Agarwal, P. Barham, and E. Brevdo. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from tensorflow.org.

[BB12] J. Bergstra and Y. Bengio. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research*, 13(10):281–305, 2012.

[BI89] E.N. Barron and H. Ishi. The Bellman Equation for Minimizing the Maximum Cost. *Nonlinear Analysis: Theory, Methods & Applications*, 13:1067–1090, 1989.

[BPW$^+$12] C. Browne, E. J. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. P. Liebana, S. Samothrakis, and S. Colton. A Survey of Monte Carlo Tree Search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4:1–43, 2012.

[Gra13] D. Graupe. *Principles of Artificial Neural Networks*, volume 7 of *Advanced Series on Circuits and Systems*. World Scientific, Singapore, third edition, 2013.

[Jon19] A. Jones. Dota auto chess player count tops eight million, 30.04.2019. https://www.pcgamesn.com/dota-2/dota-auto-chess-player-count.

[MWL17] S. Miao, J. Z. Wang, and R. Liao. Convolutional Neural Networks for Robust and Real-Time 2-D/3-D Registration. In *Deep Learning for Medical Image Analysis*, pages 271–296. Academic Press, 2017.

[Pla20] A. Plaat. *Learning to Play — Reinforcement Learning and Games*. Springer, 2020.

[Pyt] Python Software Foundation. *Python documentation*. https://docs.python.org/3/.

[SB18] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[Wea] T. Williams and C. Kelley et al. Gnuplot 4.6: An Interactive Plotting Program. http://gnuplot.info/.

[XCZW20] J. Xu, S. Chen, L. Zhang, and J. Wang. Lineup Mining and Balance Analysis of Auto Battler. In *2020 IEEE 6th International Conference on Computer and Communications (ICCC)*, pages 2300–2307, 2020.

[ZJDR21] N. Zolboot, Q. Johnson, S. Dakun, and A. Redei. Hearthstone Battleground: An AI Assistant with Monte Carlo Tree Search. Master's thesis, Central Michigan University, 2021. https://www.researchgate.net/publication/357898736_Hearthstone_Battleground_An_AI_Assistant_with_Monte_Carlo_Tree_Search.