university of groningen / faculty of science and engineering

Hierarchical reinforcement learning in multiplayer Bomberman

Bachelor's Project Thesis

Remco Pronk, s2533081, r.pronk.1@student.rug.nl
Supervisor: dr. M.A Wiering

Abstract: Experiments have been conducted to compare winrates of an agent obtained with hierarchical reinforcement learning and flat reinforcement learning on the multiplayer mode of the videogame Bomberman. The performance between a single network, two networks and four networks have been compared. Four bombermen are placed together in an arena. This arena contains walls, which are always placed in the same location at the start of each game. Bombermen and some of these walls can be destroyed by exploding bombs. These bombs are placeable by the bombermen. Multilayer perceptrons are used to approximate the utility of each state-action pair used in Q-learning. The exploration strategies used are Error-Driven-ϵ, which is a variant on Diminishing ϵ-Greedy, and Max-Boltzmann. Each trial consisted of a hundred generations of 10,000 training games with 100 test games. A significant difference in results has been found with both exploration strategies. The winrate during the first twenty generations is higher for both the two networks and four networks experiments. After this, the winrate becomes lower in comparison to flat reinforcement learning.

1 Introduction

In recent years, research has been done on reinforcement learning in digital games (Shantia, Begue, and Wiering, 2011; Bom, Henken, and Wiering, 2013). Some of the resulting agents outperform human behaviour (Mnih, Kavukcuoglu, Silver, Graves, Antonoglou, Wierstra, and Riedmiller, 2013). This opens the road for more experimentation in different games.

This thesis builds on a previous bachelor's thesis (Kormelink, 2017). This previous work compares different exploration methods in the multiplayer mode of the game Bomberman. The same framework for the game and learning methodology is used in this research project. This allows for a direct comparison between the data sets collected. A similar research project has been done on the singleplayer mode of this game (Timmers and Mulder, 2014).

The aim of the research project is to compare the results of regular reinforcement learning versus hierarchical reinforcement learning in multiplayer Bomberman. The variable of most importance is the winrate, which will be used to determine the success of the agent. By increasing the complexity of the system through adding extra artificial neural networks for specific stages of the game, the expectation is that winrates would increase.

Two experiments will be run for each exploration method. The first experiment uses two multilayer perceptrons, whereas the second experiment uses four. This allows for specialised networks for different stages of the game. These results will be compared with the data collected from the original experiment.

Winrates and points accumulated will be compared over multiple generations, which consist of training and test games. Training games are used to train the network. During the test games, exploratory actions are no longer performed by the agent. From these test games, data will be collected. Paired t-tests will be used on the collected data to test for significant differences in both winrate and points between generations.

The question that this thesis tries to answer is whether hierarchical reinforcement learning proves to perform better with regards to winrate in multi-

player Bomberman as opposed to flat reinforcement learning.

In section 2, Bomberman will be explained, including the implementation of the game. This section also describes how the experiments are performed. Section 3 discusses the results gathered from the experiments. This includes comparisons between the different sets of experiments. In section 4, comments will be given with regards to the acquired results. Section 5 contains the final conclusion of this research.

2 Methods

2.1 Bomberman

Bomberman is a series of videogames, the first of which was released in 1983 by Hudson Soft and Konami. In this game, the player controls a character called a bomberman. This bomberman is capable of moving on a 2D grid, and is able to place bombs. These bombs can destroy enemies and certain walls. Some walls, however, are indestructible.

For this project, the multiplayer variant of the game is used. In this gamemode, bombermen are placed in each of the four corners of the gamemap. In the initial gamestate, they are surrounded by walls, and have no direct access to the other players. This state can be observed in figure 2.1. To defeat another player, a path has to be cleared first. Once enemies are reachable, they have to be killed by putting them in the blast radius of an exploding bomb.

Each bomberman has six different actions it can perform. These are *move up*, *move down*, *move left*, *move right*, *place a bomb* and *do nothing*. The blast radius of a bomb is two gridspaces in each of the four cardinal directions. Destructible walls and bombermen are removed when they are in the blast radius of an exploding bomb.

Not every action can be performed in every gamestate. A new bomb can only be placed five gamesteps after that player's previous bomb has exploded. Movements can also be invalid if, for example, a bomberman attempts to move through walls or bombs, though bombermen can move through each other. Each valid movement action only moves the bomberman one gridspace.

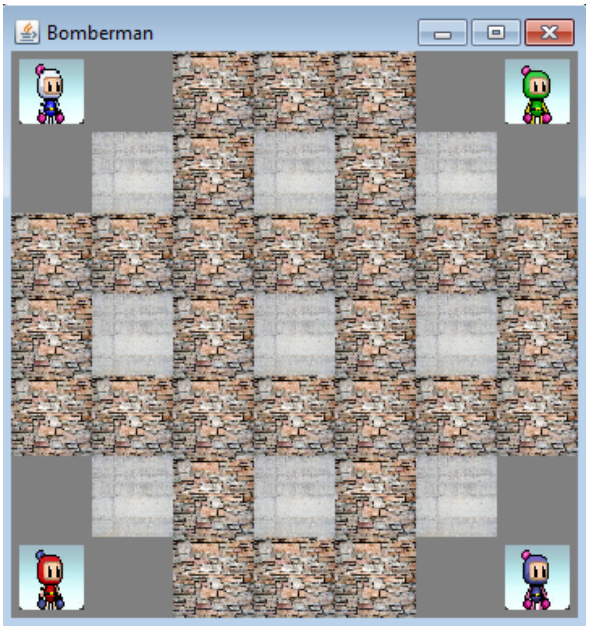

Figure 2.1: The initial state of a game. Brown walls are destructible. Grey walls are indestructible.

2.1.1 Game loop

A run of the game is controlled by the game loop. This loop repeats a set of steps until the game is ended. Player-specific steps are performed for each bomberman in the game, such as updating the bomb cooldown. These steps are:

1. Calculate and add the next move to the move buffer;
2. Perform the move from the move buffer;
3. Update all bombs and the bomb cooldown;
4. Award points after bomb explosion;
5. Update the weights of the network;
6. Remove dead bombermen;
7. Random bombs can get placed after 150 gamesteps.

The rewards for the agent change according to the current strategy used. The two strategies used are named *pathfinding* and *attacking*. These strategies determine which reward function and networks

Algorithm 2.1 addMoveToBuffer()

```
enemyCount ← checkPathToEnemies()
if enemyCount > 0 then
  changeStrategyRewards(attacking)
else
  changeStrategyRewards(pathfinding)
end if
move ← calculateMove()
addToMoveBuffer(move)
```

are used. The pseudocode for this can be seen in algorithm 2.1. The path calculation is done by the A^* algorithm. The algorithm is modified to return the amount of enemies that are accessible. The actual path is not important, since the agent has to find a path itself.

Move calculation is done by executing a forward pass by feeding the corresponding network the gamestate. The index of the highest value in the output layer corresponds with a move. Either this move, or a semi-random move is performed. An invalid move will still be tried, but will result in the agent not performing an action. How the random move is determined differs between exploration methods, and will be explained in section 2.2. Performing an action will directly result in its penalty to be subtracted from the agent's total points.

Bombs get updated by decrementing their timer. When the timer reaches zero, the bomb will explode. The only source for punishments and rewards, besides the penalty for performing an action, are from events occurring after a bomb explodes. An enemy can die, the agent can die and walls can be destroyed. After a bomb explodes, these corresponding rewards are awarded to the bomberman who placed the bomb.

Exploded bombs then get removed from the game, after which the cooldown for placing a new bomb is updated for all bombermen.

Weights of the multilayer perceptron are updated by first saving a copy of the network corresponding to the current strategy. The output layer of this network is saved as the target. A forward pass is then done on the network, and the agent takes its action. A copy of the output layer of this network is stored. The node corresponding to the action in the target set is updated. This is done by adding the reward and the discounted future reward. This discounted future reward is the highest value found in the output layer of the resulting network after the forward pass on the next state, multiplied by the discount factor. A backward pass is then done by using the target set with the initially stored network. These values found in the output layer are Q-values, which are explained in section 2.2.

To prevent stalemates, random bombs can be placed after 150 gamesteps. This happens as a final step in the gameloop. The specifics can be found in algorithm 2.2.

Algorithm 2.2 randomBomb()

```
Require: gameRound > 150
  bombProbability ← (gameRound − 150)/(50 ∗ gameRound)
  for all position : positionList do
    if emptySpace(position) then
      r ← random(0, 1)
      if bombProbability > r then
        placeBomb()
      end if
    end if
  end for
```

2.1.2 Gamestate representation

The arena in which the game is played has a dimension of seven by seven gridspaces. The gamestate is represented as a vector. Each coordinate of the game corresponds to four consecutive values in the vector. This vector thus holds 196 values.

The first input corresponds to the floortype. Possible floortypes are *hardwall*, *softwall* and *empty*, where *softwalls* are destructible walls. The values used to represent these types are $\{-1, 1, 0\}$ respectively.

The second input holds the danger level. It is calculated as $\frac{Elapsedtime}{Initialtimer}$, with regards to the blastradius of a bomb. To differentiate between bombs set by the agent and enemies, danger created by owned bombs is represented as a negative value. Its representation is $-1 \leq dangerlevel \leq 1$.

The third input tracks the location of enemies. If an enemy is present on the current coordinate, it will be represented with the value 1. Possible values thus are $\{0, 1\}$.

The fourth input is used to track the agent's location. As with the previous value, possible values are $\{0, 1\}$, where a 1 represents the current location of the agent.

2.1.3 Opponents

All enemy bombermen follow a set of rules to determine their movement. The same preprogrammed enemies have been used in the previous experiment. This serves as a baseline that allows comparison over games played.

Moves are determined using algorithm 2.3 (Kormelink, 2017). Utility is calculated using equation 2.1. The x and y represent coordinates, where x_{man} and y_{man} represent the coordinates of the bomberman after the move would have been made.

$$utility = \sqrt{|(x_{man} - x_{bomb})|} + \sqrt{|(y_{man} - y_{bomb})|} \tag{2.1}$$

Algorithm 2.3 Move calculation opponents

```
possibleActions ← returnPossibleAction()
bombList ← surroundingBombs()
if bombList.notEmpty() then
  utilityList[] ← possibleActions.size()
  for all a : possibleActions do
    for all b : bombList do
      possiblePosition ← makeAction(a)
      currentUtility ← distance(b, possiblePosition)
      utilityList[a] ← utilityList[a] + currentUtility
    end for
  end for
  bestUtility ← indexMax(utilityList)
  return indexMax(possibleActions[bestUtility])
end if
if surroundedByThreeWalls() then
  return placeBomb()
end if
return randomAction()
```

2.2 Experiment

Two sets of experiments on hierarchical reinforcement learning have been run. The first set of experiments uses the exploration method Error-Driven-ϵ, the second set uses Max-Boltzmann. All other parameters for each experiment are identical. Each set consists of two experiments, which use a different amount of networks. In the first experiment, two networks are used. One network is used during the *pathfinding* strategy, the other during the *attacking* strategy. The other experiment uses four networks. The first network is again used during the *pathfinding* strategy. The remaining three networks are used in the case of one, two or three enemies being accessible, respectively.

Table 2.1: Rewards/punishments in points, for both strategies

Event/Strategy	Pathfinding	Attacking
Action performed	-1	-1
Player death	-300	-300
Wall destruction	30	0
Enemy kill	0	100

Table 2.2: Parameters used in both experiments

Parameter	Value
Exploration rate	0.3
Learning rate	0.0001
Discount factor	0.9

Each strategy uses its own set of rewards. These can be seen in table 2.1. Although the rewards change between strategies, the reward function remains the same. When, for example, a wall gets destroyed by a bomb of the agent during the *attacking* strategy, the agentis simply rewarded with zero points, but gets points for the gathered points statistic, which will be reported in the experiments. Gathered points are set back to zero at the start of each game. The parameters used in both experiments can be seen in table 2.2.

A hundred generations have been run for each experiment. All individual experiments have been run ten times. A generation consists of training games and test games. Training games are used to train the network. Test games are used to gather results. The agent is not able to make exploratory moves during these test games. Each generation consists of 10,000 training games and 100 test games.

All networks used in a set of experiments share the same properties. The exploration method used

in the first set of experiments is Error-Driven-ϵ, which is a variant on Diminishing ϵ-Greedy (Kormelink, 2017). The implementation can be seen in algorithm 2.4. The exploration rate is determined by the error of the two previous generations. After two generations, the exploration chance becomes the lowest value between the exploration rate and the relative error. The exploration rate parameter is thus a maximum exploration chance.

Algorithm 2.4 semiRandomMove()

```
r ← random(0, 1)
if generation < 3 then
  relativeError ← explorationRate
else
  error_{t−1} ← getGenError(generation_{t−1})
  error_{t−2} ← getGenError(generation_{t−2})
  if error_{t−1} > error_{t−2} then
    relativeError ← error_{t−2}/error_{t−1}
  else
    relativeError ← error_{t−1}/error_{t−2}
  end if
  relativeError ← max(explorationRate, relativeError)
end if
if relativeError > r then
  move ← randomMove()
end if
```

The second set of experiments uses the Max-Boltzmann exploration method (Wiering, 1999). This exploration method uses the Boltzmann distribution to assign probabilities, based on the Q-value, to each action. The Boltzmann distribution function can be seen in equation 2.2. The parameter T is the temperature, which determines the exploration. A is the set of all actions.

$$P(a|s) = \frac{e^{Q(s,a)/T}}{\sum_{i \in A} e^{Q(s,i)/T}} \tag{2.2}$$

The implementation of Max-Boltzmann can be seen in algorithm 2.5. The temperature parameter decreases evenly over generations. It starts out at 200, and decreases to 1.

Each multilayer perceptron has one input layer, a single hidden layer and an output layer. The input layer has 196 nodes, the hidden layer has 100, and the output layer has six. The activation function used for the hidden layer is a sigmoid function

Algorithm 2.5 Max-Boltzmann(s)

```
r ← random(0, 1)
if r > explorationRate then
  return highestActivationAction(s)
else
  for all a : Actions do
    cumulativeProbabilities[a] ← cumulate(boltzmann(a, s))
  end for
  r ← random(0, 1)
  for all e : cumulativeProbabilities do
    if checkBetweenBounds(r, e) then
      return e
    end if
  end for
end if
```

with default parameters (Gurney, 1997), as seen in equation 2.3.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

The output layer uses a linear output function, which is defined in equation 2.4.

$$f(x) = x \tag{2.4}$$

Markov decision processes (MDPs) are used to model dynamic systems which are partly stochastic and partly under control of an agent (Ibe, 2013; Markov and Schorr-Kon, 1961). MDPs have some requirements:

- A finite set of states: S;
- A finite set of actions for each state: A_s;
- A probability function that calculates the probability of state s leading to state s' after action a: $P_a(s, s')$;
- A reward function that gives the reward after reaching state s' after executing action a in state s: $R_a(s, s')$;
- A discount factor, which is used to determine the importance of future rewards versus immediate rewards: $\gamma \in [0, 1]$;

The implementation of multiplayer Bomberman is in correspondence to these requirements. It has a

finite set of states and actions in those states. The reward, probability functions and discount factor are built into the implementation of reinforcement learning where Q-learning is used (Watkins, 1989).

The multilayer perceptrons are used to approximate the Q-values for all state-action pairs. Simply calculating the Q-values for every state-action pair induces memory constraints, since the state space is large. These Q-values represent the expected utility when action a is taken in state s, including expected future rewards discounted by the discount factor (Russell and Norvig, 2010). The update rule of Q-learning can be seen in equation 2.5, wherein α represents the learning rate.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \quad (2.5)$$

The code for these experiments can be found at https://github.com/Remco32/Bomberman_2.

3 Results

Mean winrates and points have been gathered for each generation for every run of all experiments.

3.1 Error-Driven-ϵ

In figure 3.1, the mean winrate of the two network experiment minus the mean winrate of the single network experiment is shown. In figure 3.2 this is done for the four network experiment with regard to the single network experiment. Winrates gathered from the experiment have a range between 0 and 1, inclusive.

The mean winrates over all runs of the experiments have been plotted in a single plot. This can be seen in figure 3.3.

Figure 3.4 shows the mean points obtained by the agent using the two network experiment minus that of the single network experiment. The mean points difference between the four network experiment and the single network experiment is shown in figure 3.5.

Significance has been tested for both experiments. The mean winrates for every generation have been tested between the two network experiment against the single network, and between the

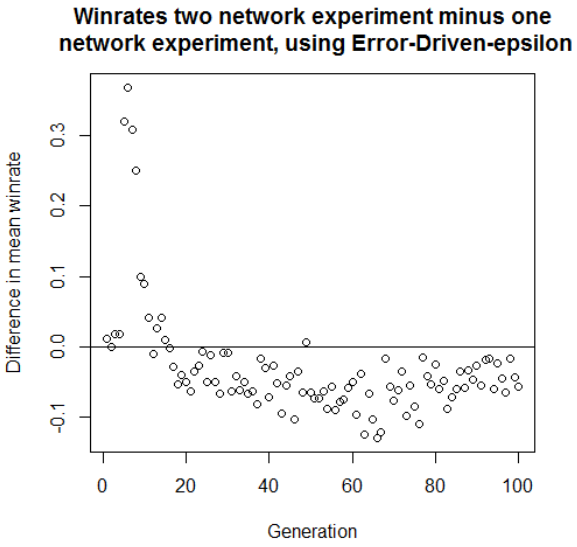

Figure 3.1: Difference in winrate between the two network experiment and previous data, using Error-Driven-ϵ.

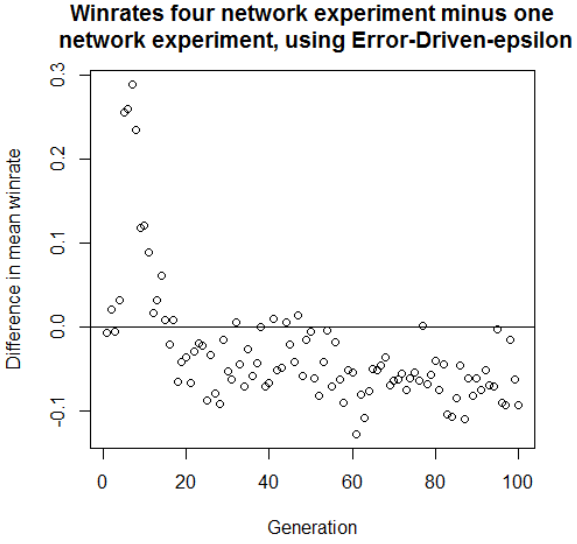

Figure 3.2: Difference in winrate between the four network experiment and previous data, using Error-Driven-ϵ.

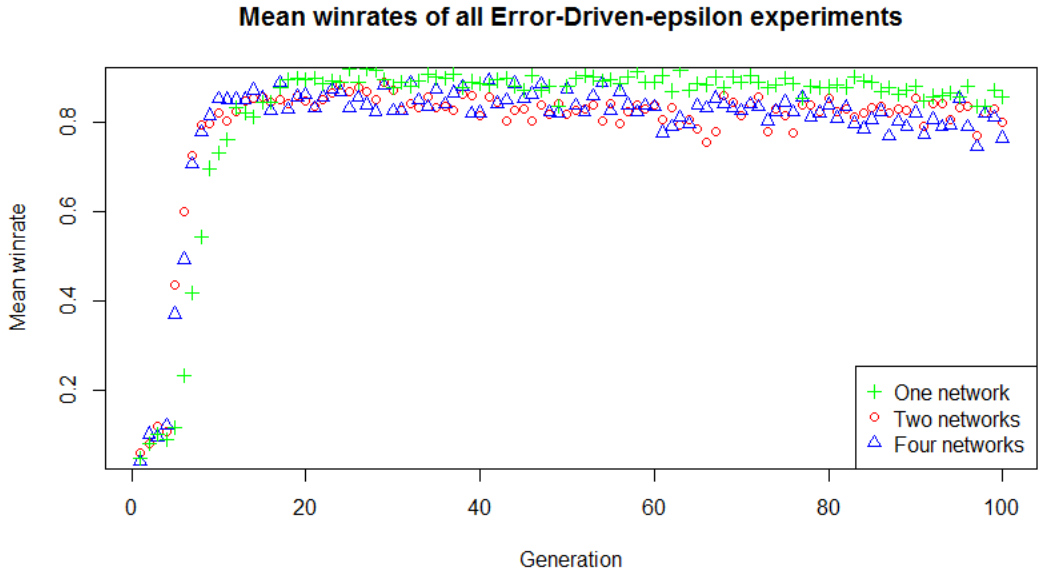

Figure 3.3: Winrates of all experiments, over generations, using Error-Driven-ϵ.

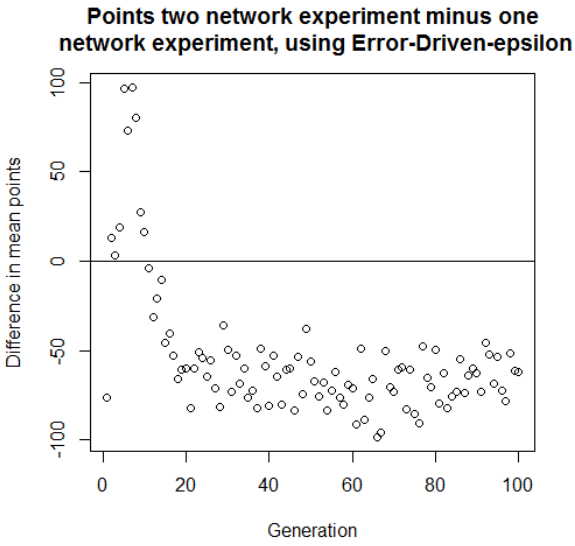

Figure 3.4: Difference in points between the two network experiment and previous data, using Error-Driven-ϵ.

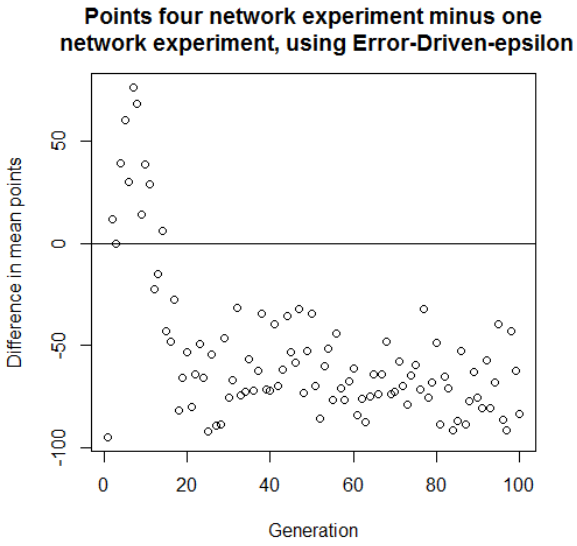

Figure 3.5: Difference in points between the four network experiment and previous data, using Error-Driven-ϵ.

Table 3.1: Mean results from the final generation between experiments with standard error, using Error-Driven-ϵ.

	Winrate	**SE**
Single network	0.856	0.026
Two networks	0.800	0.024
Four networks	0.763	0.027
	Points	**SE**
Single network	32.2	15.1
Two networks	-29.9	9.0
Four networks	-51.4	14.4

four network experiment against the single network experiment. A significant difference between the sets have been found in both experiments. The two network experiment shows a significant difference with $t(1, 99) = -3.7153, p \approx 0.0003$. The four network experiment also shows a significant difference with $t(1, 99) = -3.9492, p \approx 0.0001$.

A significant difference has also been found when comparing accumulated points. The two network experiment compared to the data from the single network experiment shows a significant difference between the data ($t(1, 99) = -14.455, p < 2.2e - 16$). The same shows when comparing the four network experiment with the original data ($t(1, 99) = -14.874, p < 2.2e - 16$).

The single network experiment shows better performance in both winrate and points accumulated over both experiments, as will be discussed further in section 4.

A table comparing the mean winrates, mean points and standard error over the last generation between the experiments is shown in table 3.1.

3.2 Max-Boltzmann

Results have been compared for the Max-Boltzmann experiments in the same way as described in the previous section. The compared winrates between the multiple experiments can be seen in figures 3.6, 3.7 and 3.8.

The points are compared between the multiple experiments, and can be seen in figures 3.9 and 3.10.

Significant results have been found for winrate when compared to the one network experiment with $p = 3.892e{-}10$ and $p = 2.614e{-}05$ for the two

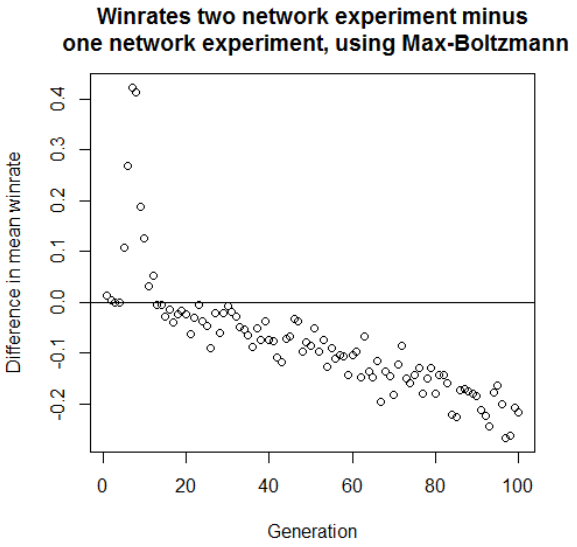

Figure 3.6: Difference in winrate between the two network experiment and previous data, using Max-Boltzmann.

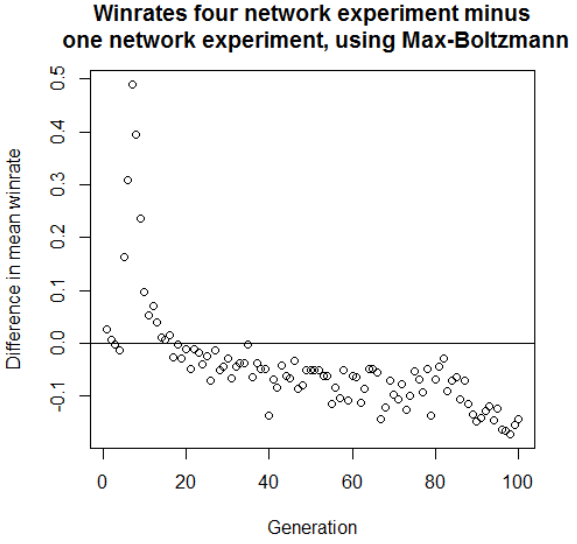

Figure 3.7: Difference in winrate between the four network experiment and previous data, using Max-Boltzmann.

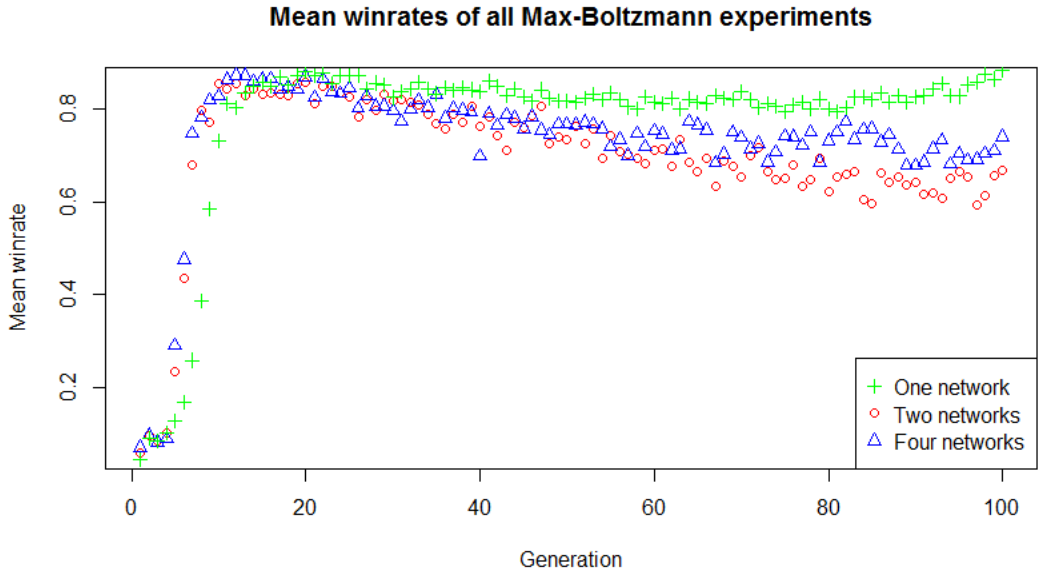

Figure 3.8: Winrates of all experiments, over generations, using Max-Boltzmann.

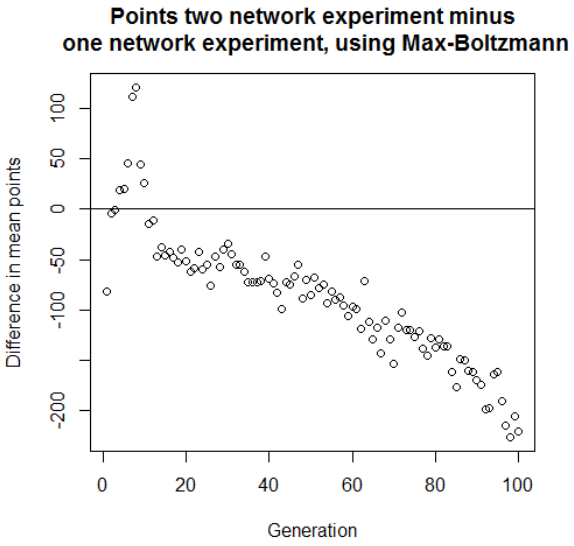

Figure 3.9: Difference in points between the two network experiment and previous data, using Max-Boltzmann.

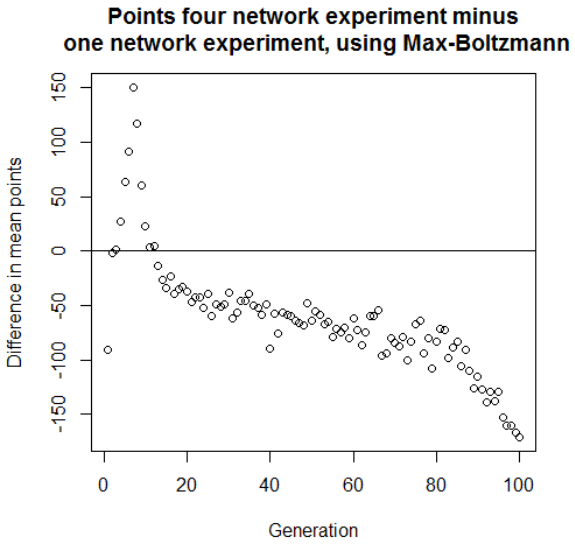

Figure 3.10: Difference in points between the four network experiment and previous data, using Max-Boltzmann.

Table 3.2: Mean results from the final generation between experiments with standard error, using Max-Boltzmann.

	Winrate	**SE**
Single network	0.885	0.015
Two networks	0.668	0.033
Four networks	0.740	0.038
	Points	**SE**
Single network	96.3	2.9
Two networks	-123.7	18.9
Four networks	-74.3	18.1

network and four network experiments respectively.

A significant difference has also been found when comparing accumulated points. $p < 2.2e-16$ between both the two network experiment and the four network experiment as compared with the one network experiment.

The single network experiment shows better performance again in both winrate and points accumulated over both experiments.

The table comparing the mean winrates, mean points and standard error over the last generation between the experiments is shown in table 3.2.

4 Discussion

4.1 Error-Driven-ϵ

The results show that there is a significant difference between the single network experiment as compared with both the two and four network experiments.

When looking at the data on the winrate, as visualised in figures 3.1 and 3.2, a trend can be seen. During the first approximately twenty generations, performance is higher than that of a single network. This holds for both the two and four network experiments. This could be explained by the fact that each network is specialised. The single network has to account for both stages of the game: *pathfinding* and *attacking*, whereas the multiple networks can specialise themselves for a certain strategy. During training this is observable by the increased performance.

After these approximately twenty generations, the multiple networks perform worse. When these last generations are looked at in isolation and compared to the single network, the results are still significantly different ($t(1, 80) = -17.363, p < .2e-16$ and $t(1, 80) = -15.699, p < .2e-16$, for two and four networks respectively). The significance found shows a significant decrease in winrate over these generations.

The final winrates for the multiple network strategies are lower as well, as seen in table 3.1.

When looking at the points gathered instead of the winrates, the same pattern can be seen emerging. The first approximately twenty generations show more points accumulated, after which it drops of. Oscillation can be seen after these initial generations more clearly in figures 3.4 and 3.5, which show the difference in points gathered. This can be explained by the characteristics of the exploration strategy. The chance of performing an exploratory move is unstable, since it is based on the error of the network. When this error doesn't stabilise, the exploration chance will become unstable as well as a result.

4.2 Max-Boltzmann

A significant difference has been found between the one network experiment compared to the two and four network experiments. When looking at the last eighty generations, a significant difference is still found in both experiments ($p < 2.2e-16$).

As with the Error-Driven-ϵ exploration method, the hierarchical networks performs better with regards to winrate during the first approximately twenty generations, after which performance drops of. However, using the Max-Boltzmann exploration method, winrate continues to decline over generations, as seen in figure 3.8. This could be explained by the characteristics of Max-Boltzmann. As explained in section 2.2, a distribution is used to determine the probabilities of actions being taken. It is possible that through repeatedly selecting subpar actions, a (local) optima is never reached.

Figure 3.8 also shows that the four network experiment has a less steep downward trend with regards to winrate over generations. This could be explained by the fact that four networks have to be trained, instead of just two. These networks are only trained in specific game situations. By using more networks, an individual network has less training time as compared to using less networks. The cause of the downward trend in winrate would

be affecting more networks, resulting in the effect emerging slower.

A limitation of this research is the absence of experimentation with more exploration strategies. Only two exploration strategies have been used to compare data between the hierarchical and flat reinforcement learning approach.

Expected improvement in results has been found. However, this only applies to the initial generations of the experiments. Therefore, it can be concluded that the approach of specialised networks doesn't necessarily improve performance in the game of multiplayer Bomberman, using the aforementioned exploration methods.

5 Conclusion

In this thesis, an agent has been trained to play multiplayer Bomberman. Hierarchical reinforcement learning has been used with multilayer perceptrons. These networks are trained to approximate the Q-values for each state-action pair.

The results show worse performances when using the approach of multiple networks for different strategies in the game, for both tested exploration methods. The winrate and points accumulated are higher during the first approximately twenty generations, which translates to approximately 20,000 single training games. After this stage, both winrate and points are lower for both the two and four network experiment.

It can be concluded that using multiple networks to specialise in strategies used during certain game stages of Bomberman, result in declined performance over time. One reason could be that the agent with the attacking strategy is forced to fight other bombermen due to the used reward function. It could be a better strategy to blow up walls and let other opponents fight against each other. More research can therefore be done on the effects of the reward function on the game-play performance.

More research can also be done examining different exploration strategies, before definitive conclusions can be drawn about hierarchical reinforcement learning in Bomberman. The previous thesis provided other exploration strategies, which could be tested against the hierarchical reinforcement learning approach (Kormelink, 2017).

References

L. Bom, R. Henken, and M. Wiering. Reinforcement learning to train Ms. Pac-Man using higher-order action-relative inputs. In *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning, ADPRL*, pages 156–163, 2013. ISBN 2325-1824. doi: 10.1109/adprl.2013.6615002.

Kevin Gurney. *An Introduction to Neural Networks*. Taylor & Francis Ltd, 1997. ISBN 0-203-45151-1. doi: 10.4324/9780203451519.

Oliver C. Ibe. *Markov Processes for Stochastic Modeling.*, volume 2nd edition. Elsevier, 2013. ISBN 9780124077959.

Joseph Groot Kormelink. Comparison of exploration methods for connectionist reinforcement learning in the game Bomberman. Bachelor's thesis, University of Groningen, 2017.

A. A. Markov and Jacques J. Schorr-Kon. *Theory of algorithms*. publ. for the National Science Foundation, Washington, D.C. and the Department of Commerce, USA by the Israel Program for Scientific Translations, Jerusalem, 1961.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Stuart Russell and Peter Norvig. *Artificial intelligence: A Modern Approach*. Pearson, Boston, 3rd ed. edition, 2010. ISBN 0132071487 9780132071482.

Amirhosein Shantia, Eric Begue, and Marco Wiering. Connectionist reinforcement learning for intelligent unit micro management in StarCraft. In *The 2011 International Joint Conference on Neural Networks*. Institute of Electrical and Electronics Engineers (IEEE), jul 2011. doi: 10.1109/ijcnn.2011.6033442.

Rik Timmers and Anton Mulder. Using reinforcement learning to play Bomberman. January 2014.

Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

Marco Wiering. *Explorations in Efficient Reinforcement Learning*. PhD thesis, 1999.