university of groningen / faculty of science and engineering

Comparison of Exploration Methods for Connectionist Reinforcement Learning in the game Bomberman

Bachelor's Project Thesis

Joseph Groot Kormelink, s2767058, josephgk@hotmail.nl
Supervisor: dr. M.A. Wiering

Abstract: In this thesis, we investigate which exploration method yields the best performance in the game Bomberman. In Bomberman the controlled agent has to kill opponents by placing bombs. The agent is represented by a multi-layer perceptron that learns to play the game with the use of Q-learning. The learning capabilities of the exploration methods: RandomWalk, Greedy, ϵ-Greedy, Diminishing ϵ-Greedy, Error-Driven, Max-Boltzmann and TD-Error will be compared. Bomberman is represented in a deterministic framework and the agents have been built on top of this framework. The results show that Max-Boltzmann exploration performs the best with a win rate of 88%, which is 2% higher than the second best method, Diminishing ϵ-Greedy. Furthermore, Max-Boltzmann gathers on average 30 points more than Diminishing ϵ-Greedy. Error-Driven exploration outperforms all other exploration methods over the first 80 generations, however this technique also produced unstable behavior.

1 Introduction

The idea that we learn by interacting with our environment is probably the first one that comes to mind when we think about the nature of learning. Every interaction with the environment shapes the way we think about our current state of being. Throughout our lives these interactions are a major source of the knowledge we acquire about the world.

Reinforcement learning methods are computational methods that allow an agent to learn from the interactions that take place in a specific world [1]. After perceiving the current state, the agent reasons about how to proceed to the next optimal state, based on this current state. These environments can be modelled as Markov Decision Processes [2; 3]. Reinforcement learning has already been widely applied to games [4; 5; 6; 7]. This thesis will examine the influence that exploration methods have on the final performance of the agent that is trained with Q-learning using a multi-layer perceptron to store the state-action value function. That every Borel measurable function can be approximated with a single hidden-layer feed forward neural network has already been proven [8]. Thus, it is no surprise that the perfect Bomberman player can be approximated with a neural network. We will explore the influence of specific exploration methods on the training time needed before getting to a near perfect player.

Bomberman is a strategic maze game where the player must kill other players to become the winner. The player controls one of the Bombermen and must, by means of placing bombs, kill the other players. To get to the other players it is necessary to first remove a set of walls by strategically placing bombs.

We aim to contribute to the field of connectionist reinforcement learning and will do this by examining how well exploration methods perform in the game Bomberman. The research question is: "What exploration method performs the best in the game Bomberman?". We will compare the exploration methods: RandomWalk, Greedy, ϵ-Greedy, Diminishing ϵ-Greedy, Error-Driven, Max-Boltzmann and TD-Error. Q-learning will be used to learn the expected state-action value of all moves in a given state. A multi-layer perceptron (MLP) will be used

to represent the Q-value function [9]. The agent will receive the current state as input and then will try to approximate the optimal Q-value for every action. Performance will be measured by using the average amount of points gathered combined with the individual win rate. First, we will describe the framework and then we explain the different exploration methods. After that, we will show the results and discuss them. Finally, we will conclude with respect to the research question and provide a basis for future work.

2 Framework

2.1 Bomberman

Bomberman is a strategic maze-based video game developed by Hudson Soft in 1983. The goal is to finish some assignment by means of placing bombs. Here we focus on the multiplayer variant of Bomberman where the goal is to be the last man standing. At the beginning of the game all four players start in opposing corners of the grid. The Bombermen all have 6 possible moves they can take to transition through the game: up, down, left, right, wait, place bomb. The grid is filled with obstacles. There are two types of obstacles, breakable and not breakable. Before the agent can kill its opponents it first needs to pave a path through the grid. Since the grid is filled with obstacles at the start of the game (see Figure 2.1), players need to break destructible objects in order to reach other players.

We have developed a framework that implements Bomberman in a discrete manner. A graphical depiction of this framework can be seen in Figure 2.1. Every Bomberman is controlled by an agent. The game state is sent to the agent, which can then determine the next move. After an action has been executed the consequences of that action are communicated to the agents in the form of rewards. When all agents have decided on an action, the actions are executed simultaneously so that no agent has an advantage. After a bomb has been placed it will wait 5 turns before it explodes. If a bomb explodes all hits are calculated. A turn consists of: determining the action, executing the action and then calculating hits. If there is a hit with a breakable wall, the wall vanishes. If all players die simultaneously,

Figure 2.1: Starting position of all Bombermen. Brown walls are breakable, grey walls are unbreakable

no one wins. As the game progresses, agents gain more freedom due to the vanishing walls. This results in a lowering difficulty. Thus, Bomberman is a non-converging game. This poses problems because computation time can become infinite without the addition of new knowledge. To tackle this issue we added an element that makes the game converging. After 150 rounds, bombs are placed at random locations. The amount of bombs placed increases every round. This leads to a converging endgame whilst not removing any of the Markov assumptions. The Markov assumption still holds since the next state is only based on the current state and the selected actions.

2.2 State Representation

The game state is transformed into an input vector. The game is divided in grid cells, where every cell represents a position. Every cell has 4 values:

- Is the position free, breakable or obstructed? resp. (1 ,0, -1)
- Does this position contain the player? (1, 0)

- Does this position contain an enemy? (1, 0)
- How dangerous is this position? (-1 ≤ danger ≤ 1)

Danger is measured in $\frac{\text{Time passed}}{\text{Time needed to explode}}$, the danger is negative if the bomb has been placed by the player, and positive if it has been placed by an opponent. The state representation is sent to the MLP, which will be trained using Q-learning as described in the next section.

3 Reinforcement Learning

3.1 Q-learning

Reinforcement learning is a type of machine learning that allows agents to automatically learn the optimal behaviour in a certain context to maximize overall performance. When an agent starts playing it will decide on an action from its action space. What action will be chosen depends on the exploration strategy that is being followed. We will explain more about exploration methods in section 4. After executing an action the agent receives a reward, which is a numerical representation of the consequences of the action it chose. The difference between the received reward plus the next Q-value for the best action and the actual Q-value is the error. The goal of learning is to minimize the error, so the agent can predict the consequences of its actions. The framework created for this study is a model of a Markov Decision Process (MDP) [3]. An MDP is a discrete-time stochastic control process and is defined by the following characteristics:

- A finite set of states S, where $s_t \in S$ is the state at time t.
- A finite set of actions A, where $a_t \in A$ is the action executed at time t.
- A reward function $r(s, s')$ which denotes the reward when transitioning from state s to state s'.
- A transition function $P_a(s, s')$ that gives the probability of ending up in state s' given action a in state s.
- A discount factor $\gamma \in [0, 1]$, which describes how heavy future rewards should be taken into account in the current action selection.

Learning the optimal policy is done by using Q-learning [10]. For every state-action pair a Q-value is learned. The Q-value is the sum of expected rewards obtained by performing action a in state s and following the optimal policy afterwards. The value function of the Q-learning algorithm can be updated according the following rules:

$$\delta_t = r(s_t, s_{t+1}) + \gamma * \max_a Q(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t) \tag{3.1}$$

$$Q_t(s_t, a_t) = Q_t(s_t, a_t) + \alpha * \delta_t \tag{3.2}$$

In equation 3.1 δ_t is the Temporal-Difference Error (TD-Error). The Temporal-Difference Error is the difference between the received reward plus the next Q-Value for the best action and the actual Q-value. In equation 3.2, α is used to regulate how fast the Q-value is pushed in a certain direction. If α is too big an update might overshoot, if α is too small convergences takes long. Q-learning is an off-policy algorithm, which means that it learns independently of the agent's policy. If one would play the game an infinite amount of times, Q-learning with lookup tables would converge to the optimal strategy, and can therefore be used to learn in an MDP [10]. In this thesis, we use an adaptation of the Q-learning algorithm that fits with an MLP as described in section 3.3:

$$Q_t(s_t, a_t) = r(s_t, s_{t+1}) + \gamma * \max_a Q(s_{t+1}, a_{t+1}) \tag{3.3}$$

Where $Q_t(s_t, a_t)$ is the target value for updating the MLP for state s_t and action a_t. We can use this adaptation of the Q-learning rule because the MLP already has a learning rate, thus we can remove this from the formula. This leaves us with equation 3.3 to update the Q-value in an MLP.

3.2 Reward Function

In reinforcement learning, interaction with the environment is important. Therefore, we need a way to transform action consequences into something that Q-learning can use to learn the Q-function. We do this by giving in-game events a numerical reward. The in-game events and rewards are stated in Table 3.1. These rewards have been carefully chosen to clearly distinct between good and bad actions. Dying is by far the worst thing, so this is represented by a very negative reward. For every bomb

Table 3.1: Ingame rewards

event	reward
Perform action	-1
Perform impossible action	-2
Break a wall	30
Kill a player	100
Die	-300

placed, it is tracked who placed it. The reward of killing a player is only rewarded to the player that actually placed the bomb. The rest of the rewards are chosen to promote fast convergence and active behaviour.

3.3 Multi-layer Perceptron

A problem that occurs when using Q-learning with lookup tables is that a large state space requires a large amount of memory. Every state needs to have its own Q-value for every action. Since we have a lot of states we need a lot of memory to store those. Furthermore when using lookup tables, Q-learning needs to explore all states before it can make educated decisions on what action to take.

To solve these issues regarding space and time complexity we use an MLP. An MLP is a feed-forward artificial neural network that maps an input vector that represents the state, to an output vector, that represents the Q-value for all actions. An MLP consists of a set of nodes. These nodes are ordered in layers and every layer is fully connected to the next layer. Every time a node outputs a value, it is sent through connections to the next nodes. In a connection the value is multiplied with a weight assigned to that connection. Then, in the next nodes, all values are summed and transformed by some activation function. We use the sigmoid function as activation function for the hidden-layer:

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

We use a linear output function for the output layer, so we can also predict values outside of the [0,1] scope. As input we give the complete game state to the MLP. The output of the MLP is a 6 value vector, where every value represents a Q-value for a corresponding action. The MLP is initialized randomly, which means that it needs to learn what Q-values correspond to which state-action pairs. We do this by backpropagating the error through the MLP to update the output in the direction of the target Q-values. After training the MLP we can map the appropriate Q-values without having to store all different Q-values of all states. The MLP does this by generalizing over the input state and transforming it into output. Because of this our MLP is also able to make intelligent decisions in states that it has not encountered yet. The use of an MLP results in both a decrease in space and time complexity.

4 Exploration Methods

If we humans engage in an activity, we always want to gain the optimal reward. Imagine you are going out for dinner, you look for the best food for the cheapest price. The problem is, you cannot be certain you've had the best before you've tried all the food. In reinforcement learning this is not different: we want the best policy. Exploration methods describe the policy one should use to learn the optimal solution while minimizing the cost. If we reflect exploration methods on the game Bomberman we want to learn how to kill our opponents as fast as possible. The problem of balancing exploration and exploitation is known as the exploration/exploitation dilemma [11]. This dilemma describes the difficulty of deciding whether to get more knowledge or to act upon current knowledge. It describes the trade-off of being stuck in a local optimum versus exploring insignificant parts of the state space.

We are going to compare 7 exploration methods and finally determine which works best. The exploration methods will be introduced and explained in sections 4.2-4.8. The best performing method is the method that gathers the most points and has the highest final win rate. Points are gathered during the game, where the total amount of points equals the sum of all individual rewards. For every algorithm we have two phases: we have a training phase and a testing phase. In the training phase we take the actions suggested by the exploration method. In the testing phase we take the actions with the highest Q-value. We are testing the performance of the MLP after training a certain amount of time with a specific exploration method. We cannot reliably

test the performance of the MLP with randomness involved and therefore the agent always takes the action with the highest Q-value. Since we want to test how an MLP learns over time, one training and one testing phase might not be sufficient. Therefore we introduce generations. A generation consist of one training and one testing phase. With generations we can show how the MLP learns relative to the amount of games played.

4.1 Opponents

To evaluate how good our exploration methods perform, we need a baseline. For this we introduce a hard coded opponent algorithm. All 3 opponents use this algorithm. These opponents do not learn, but instead are guided by properties that induce intelligent behaviour. The algorithm consists of 3 elements. The first element is that the agent always search for cover in the neighbourhood of a bomb. In Algorithm 4.1 we can see this in the first conditional statement, where the agent will search for cover in the neighbourhood of a bomb. The agent searches for cover by calculating the utility of every action. It does this by iterating through all bombs that are within hit-range of the Bomberman. If a Bomberman is within hit-range of a bomb, an utility value is calculated for every action. The utility is calculated by taking the root of the absolute axis dependant distance between the bomb and the Bomberman given a certain action is executed. The absolute axis dependent distance is calculated by adding the root of the absolute x-distance to the root of the absolute y-distance. Because we separate the x/y-axis and take the root of the difference, actions that make sure the Bomberman and the bomb are no longer on the same x/y axis get a higher utility. Secondly, if an agent is surrounded by 3 walls it will place a bomb. If the agent is surrounded by 3 walls, there has to be at least one breakable wall. The combination of placing bombs when surrounded by walls and searching cover in the neighbourhood of bombs works well because it shows incentive of opening up paths, while staying clear of bombs. If there are no bombs and not enough walls the algorithm produces random behaviour. This algorithm is called semi-random because the behaviour is mostly guided, but random at times.

Algorithm 4.1 Semi Random Opponent

```
possibleA = ReturnPossibleActions(player)
bombList = SurroundingBombs(player)
if bombList.NotEmpty() then
  utilityList[] = possibleA.Size()
  for a : possibleA do
    for bomb : Bomblist do
      possiblePos = MakeAction(a, player)
      curUtility = Dist(bomb, possiblePos)
      utilityList[a] += curUtility
    end for
  end for
  bestUtility = IndexMax(utilityList)
  return(possibleA[bestUtility])
end if
SBT(obj) = SurroundedByThreeWalls(obj)
if SBT(player) == TRUE then
  return(placeBomb)
end if
return(RandomAction())
```

4.2 RandomWalk

The RandomWalk exploration method executes a randomly chosen action every turn. This method produces completely random behaviour, and is therefore good as baseline method. Therefore with this method it is possible to show the difference between guided and completely random behaviour when learning a policy.

4.3 Greedy

The Greedy method is the complete opposite of the RandomWalk exploration strategy. We assume the knowledge represented in the current Q-function is completely correct and therefore every action is based on exploitation. We always take the action with the highest Q-value because we assume that this is always the best action. Greedy tries to solve some problems of RandomWalk: if the agent dies constantly in the early game, the agent will not get to explore the later part of the game. This could be solved by taking no bad actions and this could be achieved by only taking actions with the highest Q-value. Because this method does not explore it is susceptible to local optima. The pseudo-code is stated in Algorithm 4.2

Algorithm 4.2 Greedy()

$state = CalculateState()$
$action = GetBestAction(state)$
$return(action)$

4.4 ϵ-Greedy

ϵ-Greedy exploration is one of the most used and simplest methods. It has 1 parameter, ϵ, that determines what percentage of the actions is randomly selected. The parameter falls in the range $0 \leq \epsilon \leq 1$ where 0 translates to no exploration and 1 to only exploration. The action with the highest Q-value is chosen with the probability $1 - \epsilon$ and a random action is selected otherwise. Since the MLP is initialized randomly initially the Q-values do not translate to the actual sum of expected rewards. In the Greedy method this could lead to the problem of always taking a specific sub-optimal action in a state. ϵ-Greedy solves this problem of the Greedy method by allowing for some randomness in the action selection which allows for exploring the effects of different actions. The pseudo-code is stated in Algorithm 4.3.

Algorithm 4.3 $\epsilon - Greedy(\epsilon)$

$rand = RandomValue(0, 1)$
if $rand < \epsilon$ **then**
  $return(RandomAction())$
else
  $return(Greedy())$
end if

4.5 Diminishing ϵ-Greedy

Diminishing ϵ-Greedy is a variant of ϵ-Greedy in which ϵ decreases relative to the amount of games that have been played. ϵ-Greedy explores the same amount in the beginning as in the end. This is not optimal since we assume the agent is improving its behaviour and thus over time needs less exploration. This will lead to less exploration if we play more games. We also incorporate a minimal exploration value, to make sure the algorithm is still exploring in the end. As can be seen in Algorithm 4.4 the pseudo-code is similar to that of ϵ-Greedy, only now we calculate ϵ adaptively.

Algorithm 4.4 Diminishing ϵ-Greedy

$curExplore = \epsilon * (1 - \frac{currentGen}{totalGens})$
if $curExplore < 0.05$ **then**
  $curExplore = 0.05$
end if
$return(\epsilon{-}Greedy(curExplore))$

4.6 Error-Driven

The problem of Diminishing ϵ-Greedy is that you have to specify beforehand how much the agent is going to explore over time. To solve this problem, Error-Driven exploration bases the exploration rate on the difference in error between the previous two generations. The current exploration can be calculated with: if $t \geq 3$

$$max(\frac{min(error_{\text{t-1}}, error_{\text{t-2}})}{max(error_{\text{t-1}}, error_{\text{t-2}})}, minExploration) \tag{4.1}$$

Where t is the amount of generations and the error is calculated as the sum of TD-Errors in a game, averaged over a generation. If $t < 3$ the exploration chance is the standard value of ϵ-Greedy. The exploration baseline exists to ensure that some exploration occurs . The pseudo-code of this exploration method is stated in Algorithm 4.5. The "GetErrorBasedExploration" function gets the error using equation 4.1.

Algorithm 4.5 Error-Driven

$curExplore = GetErrorBasedExploration()$
$return(\epsilon{-}Greedy(curExplore))$

4.7 Max-Boltzmann

One drawback of all the previous methods mentioned is that all explorative actions are chosen randomly, which means that the second best action is chosen as likely as the worst action. The Boltzmann exploration method tries to solve this problem by assigning a probability to all actions, ranking best to worst [11] .

The probabilities are assigned using a Boltzmann distribution function. The probability $\pi(s, a)$ for

action a in state s is:

$$\pi(s,a) = \frac{e^{(Q(s,a)-\max_a Q(s,a))/T)}}{\sum_i^M e^{(Q(s,a^{i=1})-\max_a Q(s,a)/T)}}$$

Where M is the amount of possible actions and T is the temperature parameter in the function above. A high T translates to a lot of exploration. If we divide the Q-values by a high T the probabilities of the actions will be approximately equal, which means that all actions are chosen equally. We decrease T over time until we reach a value of 1. However, Boltzmann exploration did not provide any good results because it produced either random or Greedy behaviour. We have adapted the algorithm and now use Max-Boltzmann. Max-Boltzmann produces the same behaviour as ϵ-Greedy, however the exploration is now done according the Boltzmann distribution function [12; 13]. The greedy action will be taken with probability $1 - \epsilon$ and otherwise the action will be chosen according to the Boltzmann distribution function, as shown in Algorithm 4.6.

Algorithm 4.6 Max-Boltzmann(ϵ)

```
rand = RandomValue(0, 1)
if rand > ε then
  return(Greedy())
end if
state = GetState()
t = GetTemperature()
qValues = GetQValues(state)
probs = QvalueToProbability(Qvalues, t)
action = ChooseAction(probs)
return(action)
```

4.8 TD-Error

The Temporal-Difference Error (TD-Error) is an exlporation strategy that uses the error range of the Q-value estimates, next to the prediction of the Q-values [14]. This method is based on Kaelbling's Interval Estimation [15]. Kealbling's Interval Estimation is used to assess how reliable a Q-value is. The MLP has now 12 instead of 6 outputs, where the first 6 outputs represent the Q-value and the other 6 outputs represent the expected absolute Temporal-Difference . We take the next action based on the action that has the highest possible reach in the Q-value. The reach is the highest possible Q-value given the current Q-value prediction and the error interval. We calculate the reach by adding the $\sqrt{interval}$ to the expected Q-value. Furthermore, with a probability of ϵ it chooses a random action. The pseudo-code of this exploration method is stated in Algorithm 4.7.

Algorithm 4.7 TD-Error(ϵ)

```
rand = RandomValue(0, 1)
if rand < ε then
  return(RandomMove())
end if
state = GetState()
qValues = GetQValues(state)
range = GetErrorRange(state)
maxReach = −∞
bestAction = NULL
for (action : Actions) do
  reach = qValues[action] + √range[action]
  if reach > maxReach then
    maxReach = reach
    bestAction = action
  end if
end for
return(bestAction)
```

5 Experiments and Results

To answer the research question we will test the exploration methods in combination with an MLP and Q-learning. Every method will be trained for 100 generations. A generation consists of 10,000 training games and 100 testing games. 100 generations of training is called 1 simulation. The results were obtained by running 20 simulations and taking the average of the results. The averaging is done to ensure the validity of the data. For every algorithm we are going to examine what percent of the games the methods win, and how many points they are able to gather. The amount of points they gather is the average sum of rewards gathered in a game. The input length of our MLP on a 7×7 grid is $7 \times 7 \times 4 = 196$. However, in a 7×7 grid we have 9 indestructible walls, so the true dimensionality of the input is $7 \times 7 \times 4 - 9 \times 4 = 160$. We use a single hidden-layer MLP with 100 hidden nodes and

6 output nodes, 1 output node for every possible action. For this experiment, the MLP is initialized randomly with values between -0.5 and 0.5, this to make sure saturation is not likely to occur. After running multiple preliminary experiments, 100 hidden nodes was found to be sufficient to produce intelligent behaviour for a grid size of 7×7. For this research there has also been experimented with different amounts of nodes, but removing nodes would decrease performance and increasing nodes would only add computation time with no performance increase. Adding more layers has also been investigated but this only decreased the performance.

5.1 Parameters

To find the best hyperparameters a lot of testing has been done. Since the parameters are continuous, one can never be sure the optimal parameters have been found. All neural networks have been trained with a learning rate of 0.0001, and a γ of 0.95. These values were selected after some preliminary experiments and are the same for all exploration methods. Table 5.1 shows the parameters for all methods where ϵ equals the exploration chance, min-ϵ is the minimal exploration chance and T is the Temperature.

Table 5.1: The parameter settings in training

Settings	ϵ	**min-ϵ**	T
RandomWalk	/	/	/
Greedy	/	/	/
ϵ-Greedy	0.3	/	/
Diminishing ϵ-Greedy	0.3→0.05	0.05	/
Error-Driven	/	0.05	/
TD-Error	0.2	/	/
Max-Boltzmann	0.3	/	200→1

5.2 Results

Figure 5.1 shows the exploration methods' win rate over time. In Figure 5.1 one can see that there is a big difference between the methods that use an exploration/exploitation trade-off and the methods that do not (Greedy, Randomwalk). The trade-off methods obtain quite good performances, although they do not seem to improve after 20 generations. Error-Driven outperforms all other methods for the first 80 generations, but eventually gets surpassed by Diminishing ϵ-Greedy and Max-Boltzmann.

In Figure 5.2 we show for every exploration method the average amount of points it gathered. The two methods without exploration/exploitation trade-off converge to the same value, while the other methods perform much better. All methods with the exploration/exploitation trade-off initially follow the same curve and learning process, but one can see that after 100 generations all methods converge to different values.

Table 5.2 shows the results over the last 2000 games for every exploration method. We show the mean percentage of the games that were won and the standard error. It can be seen that Max-Boltzmann performs the best while Error-Driven and Diminishing ϵ-Greedy perform second best.

Table 5.2: Mean and standard error of the win rate over the last 2000 games

	Mean win rate	**SE**
RandomWalk	0.08	0.006
Greedy	0.58	0.033
ϵ-Greedy	0.79	0.014
Diminishing ϵ-Greedy	0.86	0.022
Error-Driven	0.86	0.026
TD-Error	0.82	0.027
Max-Boltzmann	0.88	0.015

Table 5.3 shows the average amount of points gathered and the standard error for every exploration method. This data was gathered over the last 2000 games. This table shows that Max-Boltzmann performs far better than the other methods, scoring on average 30 points more than the second best method. There was a significant difference between Diminishing ϵ-Greedy and Max-Boltzmann, $p<0.0001$.

Table 5.3: Mean and standard error of the gathered amount of points over the last 2000 games

	Mean points	**SE**
RandomWalk	-336	0.2
Greedy	-346	1.1
ϵ-Greedy	3	1.2
Diminishing ϵ-Greedy	66	1.1
Error-Driven	32	1.5
TD-Error	55	1
Max-Boltzmann	96	1.3

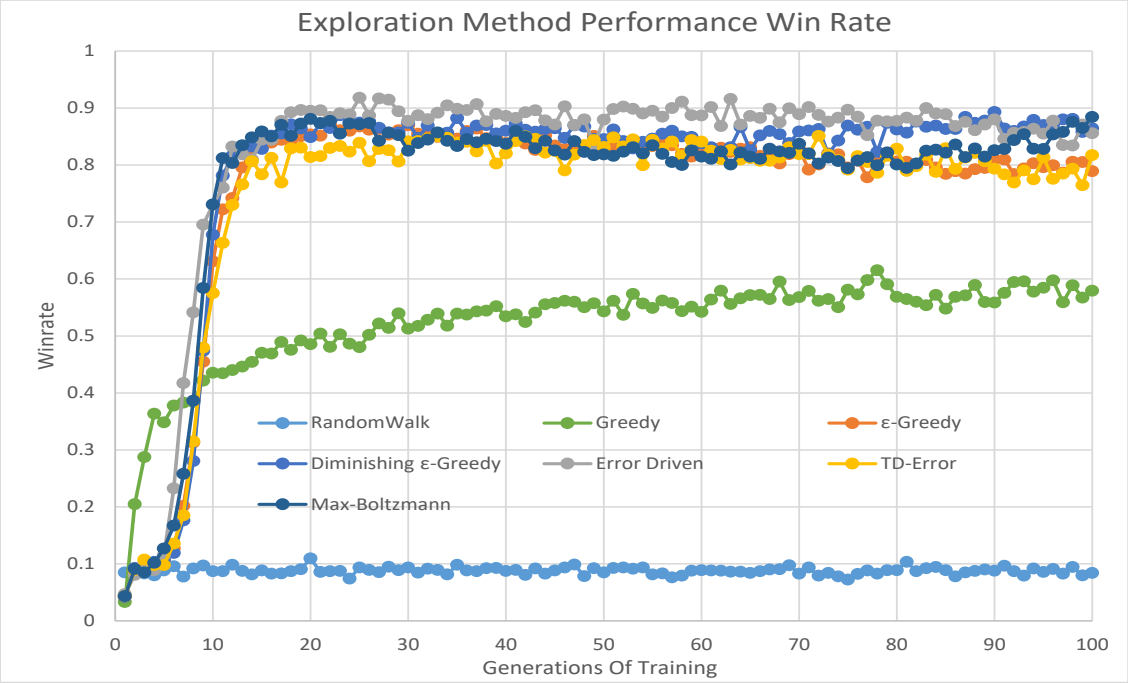

Figure 5.1: Win rate of the exploration methods, where a generation consist of 10,000 training games and 100 testing games

5.3 Discussion

From the results it can be concluded that Max-Boltzmann performs better than the other methods. Max-Boltzmann gathers on average 30 points more than the second best method and has a 2% higher win rate. Especially the high amount of points is important. We are maximizing the amount of points and a high win rate does not always correspond to a high amount of points. That a high win rate does not directly translate to a high amount of points becomes clear when comparing Greedy to RandomWalk. Greedy has a much higher win rate overall than RandomWalk whereas it gathers less points. In the first 60 epochs the T of Max-Boltzmann is relatively high, which means that it produces approximately equal behaviour to ϵ-Greedy. During the last 40 epochs the exploration gets more guided, we can see that this results in the average amount of points starting to increase compared to ϵ-Greedy. It is worth noting that Error-Driven exploration outperforms all other methods in the 10-80 generations interval. However this method produces unstable behaviour. This claim is supported by the relative high standard error in Table 5.3 and can be seen in Figure 5.1.

We can answer our research question by stating that Max-Boltzmann performs better than all other methods. The only problem with Max-Boltzmann is that it takes a lot of time before it outperforms the other methods. In Figure 5.1 and 5.2 we can see that only in the last 10 generations Max-Boltzmann starts to outperform the other methods. Looking at the tables, it is clear that the trade-off between exploration and exploitation is important. All methods that have this exploration/exploitation trade-off perform significantly better than the methods that have either only exploration or exploitation.

6 Conclusions

This thesis examined exploration methods in connectionist reinforcement learning in Bomberman.

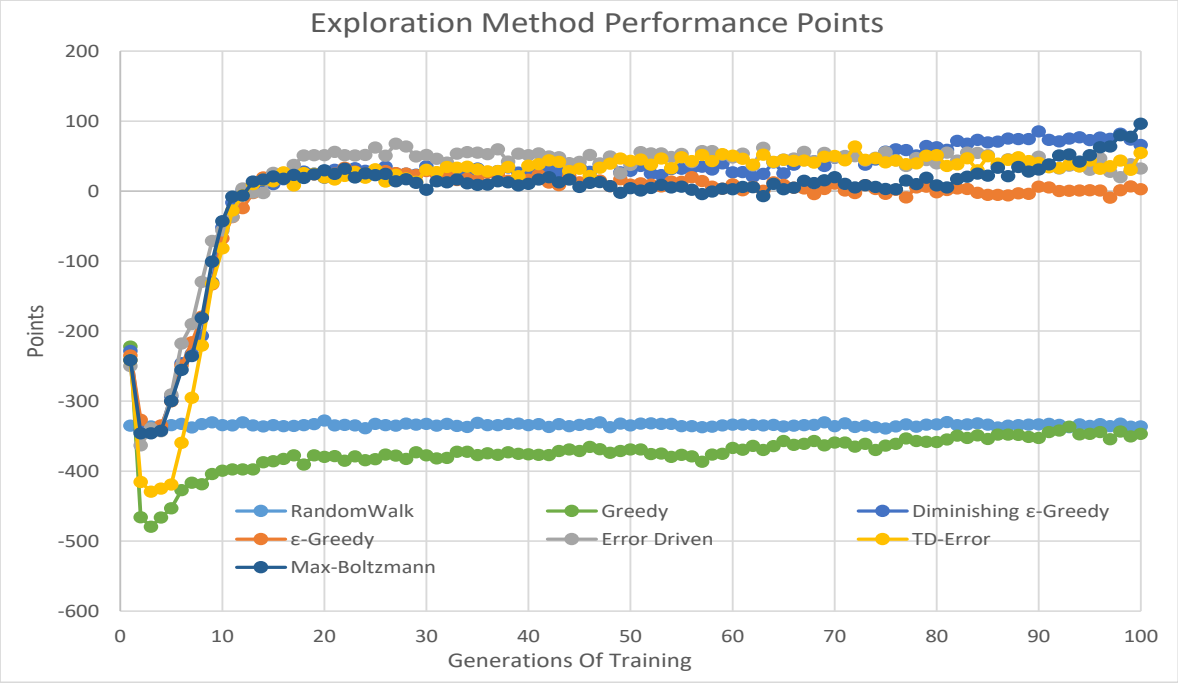

Figure 5.2: Points gathered by the exploration methods, where a generation consist of 10,000 training games and 100 testing games

We have explored multiple exploration methods and can conclude that Max-Boltzmann outperforms the other methods on both win rate and points gathered. The only aspect where Max-Boltzmann is being outperformed, is the learning curve. Error-Driven learns faster, but produces unstable behaviour. Max-Boltzmann takes longer to reach a high performance than some other methods. For further research, it might be interesting to combine Error-Driven exploration with Max-Boltzmann. Both methods perform well and a combination of the two might solve the problem where Max-Boltzmann acts as ϵ-Greedy early on.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning : An Introduction*. Cambridge: Bradford Books, Mar 6, 2015.

[2] M. van Otterlo and M. Wiering, "Reinforcement Learning and Markov Decision Processes," in *Reinforcement Learning: State-of-the-Art* (M. Wiering and M. van Otterlo, eds.), Springer Berlin Heidelberg, 2012.

[3] R. Bellman, "A markovian decision process," *Journal of Mathematics and Mechanics*, vol. 6, no. 5, pp. 679–684, 1957.

[4] L. Bom, R. Henken, and M. Wiering, "Reinforcement learning to train Ms. Pac-Man using higher-order action-relative inputs," in *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pp. 156–163, 2013.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[6] I. Szita, "Reinforcement learning in games,"

in *Reinforcement Learning: State-of-the-Art* (M. Wiering and M. van Otterlo, eds.), pp. 539–577, Springer Berlin Heidelberg, 2012.

[7] A. Shantia, E. Begue, and M. Wiering, "Connectionist reinforcement learning for intelligent unit micro management in starcraft," in *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pp. 1794–1801, IEEE, 2011.

[8] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.

[9] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1," ch. Learning Internal Representations by Error Propagation, pp. 318–362, Cambridge, MA, USA: MIT Press, 1986.

[10] C. J. C. H. Watkins and P. Dayan, "Technical note: Q-learning," *Machine Learning*, vol. 8, p. 279, May 1, 1992.

[11] A. D. Tijsma, M. M. Drugan, and M. A. Wiering, "Comparing exploration strategies for Q-learning in random stochastic mazes," in *Computational Intelligence (SSCI), 2016 IEEE Symposium Series on*, pp. 1–8, 2016.

[12] M. Wiering and J. Schmidhuber, "HQ-learning," *Adaptive Behavior*, vol. 6, no. 2, pp. 219–246, 1997.

[13] M. A. Wiering, *Explorations in efficient reinforcement learning*. PhD thesis, University of Amsterdam, 1999.

[14] M. White and A. White, "Interval Estimation for Reinforcement-Learning Algorithms in Continuous-State Domains," in *Advances in Neural Information Processing Systems 23* (J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, eds.), pp. 2433–2441, Curran Associates, Inc., 2010.

[15] L. Kaelbling, *Learning in Embedded Systems*. A Bradford book, MIT Press, 1993.