university of groningen / faculty of mathematics and natural sciences / artificial intelligence

Reinforcement Learning for Playing Othello

Bachelor's Project Thesis

Thijs Eker, s2576597, thijs.eker@gmail.com,
Supervisor: Dr. M.A. Wiering

Abstract: In this thesis we examined the effect of adding more hidden layers, using different activation functions and using a dropout algorithm on the performance of an artificial neural network (ANN) learning to play Othello using Q-learning. While all of these methods have seen promising results in supervised learning, the results for our reinforcement learning experiment were not that spectacular. The activation function with the best results is the sigmoid function. The ReLU and ELU functions do not increase the performance of the agent nor do they speed up the convergence of the ANN. Dropout also does not increase performance. Adding more hidden layers increased the performance and helped speed up the learning process. An ANN with two hidden layers using a sigmoid function and no dropout algorithm yields the best performance in our experiment for playing Othello and wins 95% of the testing games playing against a fixed opponent after learning for two million games.

1 Introduction

Humans are used to interacting and learning from their environment. We see this in all kinds of human activities, from cooking dinner to playing video games. When we see that our actions have brought us closer to our goal we will use them again the next time we encounter a similar activity. For computers this kind of learning is a difficult task. Reinforcement learning is the field of research that concerns itself with these kinds of problems [1].
Reinforcement learning has a wide range of applications. There has been a lot of research into agents learning to play games using reinforcement learning algorithms. The reason researchers use games to experiment with reinforcement learning is that games offer enormous complexity while the problem input or the state remains well-defined. Furthermore games also offer easy performance measures (win, lose and draw). In 1995, Tesauro used the game Backgammon to show the possibilities of reinforcement learning [2] and more recently Google DeepMind used reinforcement learning in the game Go [3]. But reinforcement learning also has been successfully applied in other fields like robot navigation [4] and psychological research [5].

In this research we will apply the concept of reinforcement learning to the game Othello. We created an agent that interacts with Othello and learns from this interaction to become a better Othello player. Reinforcement learning has been used before to play Othello at superhuman levels [6].
Our agent will learn to play Othello by playing against an expert-level fixed opponent. This opponent will always take the same action when presented with the same state. The learning takes place in a neural network that is used to approximate the value of an action. The network is updated using the Q-learning algorithm [7]. The program itself is an extended version of a program used in earlier research on Othello [8].
We would like to see if we can improve the agent's understanding of the game (measured by an average score based on wins/ties/losses) by using other activation functions than the standard sigmoid activation in the neural network. Apart from the sigmoid activation function we will use a Rectified Linear Unit (ReLU) and an Exponential Linear Unit (ELU) activation function. We chose these activation functions for their recent success in the field of supervised learning. The use of rectifier functions has improved supervised networks for speech recognition [9] and image recognition [10]. The rectifier and exponential activation are not prone to a vanishing gradient while the sigmoid activation function is. Therefore these functions should also help to speed up the process of converging [11]. ReLU units sometimes suffer from a problem called an "exploding gradient" [12] resulting in neurons that no longer react to any input. This is one of the reasons why we also selected an ELU function [13]. An ELU function is one of many variations on a ReLU function, that adds a bit of negative output.

This causes the derivative of the activation function to be non-zero for negative input. The non-zero derivative should help a neuron to recover from exploding gradients.
These different activation functions and different hidden layer sizes will be tested with and without a dropout algorithm [14], to see if and to which degree a certain artificial neural network (henceforth ANN) will benefit from such an algorithm.
This thesis attempts to answer the following question: Can we improve the performance (measured by average final score) of an ANN playing Othello by using a combination of the following methods, first using a ReLU or ELU activation function as opposed to the traditional sigmoid activation function, secondly by using two or three hidden layers instead of one, and finally by making use of a dropout algorithm?

2 Method

2.1 Othello

The game Othello is played on a board of 8 by 8 squares. The goal is to occupy as many squares as possible by placing discs on the board. On each turn the player may place a disk. For a move to be valid the disc must be placed adjacent to another disc and an opponent's disc must become enclosed by the move. These enclosed discs are then turned around and become of the player's respective color. If there are no moves that fulfill these conditions, the player must pass. When both players consecutively have to pass, the game is over and the player with the most discs on the board wins. When both players have an equal amount of discs on the board, the game ends in a tie.

2.2 Q-Learning

In this research our agent is playing the game Othello and is trying to find an optimal policy using reinforcement learning. In reinforcement learning there usually is an agent taking actions based on the current state of the problem. For these actions the agent can receive a reward or punishment. The agent starts out performing random actions since it has yet to learn, but will slowly converge to a policy in which it receives the most rewards.

We assume our task satisfies the properties of a Markov Decision Process (MDP). An MDP is defined by:
1. A set of states $s = \{s_1, ..., s_n\}$, 2. A set of actions $a = \{a_1, ..., a_m\}$, 3. A transition function $P(s, a, s')$ mapping state s and action a to a new state s' with probability P, 4. A reward function $R(s, a, s')$ which holds the average reward the agent receives when going for s to s' using a, 5. A discount factor $0 \leq \gamma \leq 1$ which serves the purpose of making closer rewards count more than rewards further away.

Since Othello is a game played by two players, the next state is not only dependent on a player's own action but also on the opponent's move. Because the next state is dependent on both the player's own and the opponent's move we call the problem non-deterministic. For such a problem the Q-value for a state-action pair is given by:

$$Q(s_t, a_t) = E[r_t] + \gamma \sum_{s_{t+1}} T(s_t, a_t, s_{t+1}) \max_a Q(s_{t+1}, a) \tag{2.1}$$

Q-learning is a temporal difference algorithm, it updates the Q-value for a given state-action pair after the opponent has completed its move. The Q-value is updated by bringing the current Q-value a bit closer to the highest Q-value in the next state including the immediate reward in that state. How much the Q-value changes is controlled by the learning rate α. In our experiment the learning rate α is different for every activation function. The update function is given by the following equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{2.2}$$

2.3 Q-learning with Othello

Because the outcome of an action is only known after the opponent made its move, we can not update the Q-value for the chosen state-action pair immediately. This means that in every turn s_t where $t > 1$ we first choose a new action a_t by using the ANN to approximate the Q-value for every action. Then we update the ANN by backpropagating the error for the state-action pair (s_{t-1}, a_{t-1}), which is given by $r_t + \gamma \max_a Q(s_t, a) - Q(s_{t-1}, a_{t-1})$. And finally we execute a_t.

The reward r_n in the final state s_n is 1 when the game is won, 0.5 for a draw and 0 for a lost game.

2.4 Activation Functions

To approximate the Q-values of all actions in state s we use an ANN. Such a network typically consists of a few layers of neurons passing activations. The activation of an individual neuron is dependent on the input from other neurons and its activation function. We tried three different activation functions to see which of them works best, all of them are plotted in Figure 2.1. The sigmoid activation function was already used in previous research on the same game [8] and was therefore an obvious choice. The function converts the summed input from the neurons in the previous layer to an activation between zero and one:

$$f(x) = \frac{1}{1 + \epsilon^{-x}} \tag{2.3}$$

The sigmoid function is a stable function since activations running through the network are not allowed to be higher than one. This on the other hand also slows down convergence of the network, since the gradient used in the back-propagation of the network will be relatively low for very high summed inputs.
In our research we also made use of a rectifier function. A neuron using a rectifier activation function is called a Rectified Linear Unit (ReLU). It has proven to be more effective than the sigmoid function in image recognition and convolutional neural networks [10]. The function is given by this straightforward equation:

$$f(x) = max(0, x) \tag{2.4}$$

A property of the ReLU function is the sparsity of active neurons. When a ReLU gets a summed input $x < 0$ it will have an activation of zero. When we use a sigmoid activation function a small change in the input layer will change the activation of almost all the neurons in the network. With a ReLU function neurons with negative input have an activation of 0, these neurons are not influenced by small changes in the input. This property of a ReLU function improves the information disentangling capabilities of the network [10]. Since there is no function limiting the amount of activation of a ReLU neuron, this can sometimes get out of control and the neuron will render very high or low activations from then on.

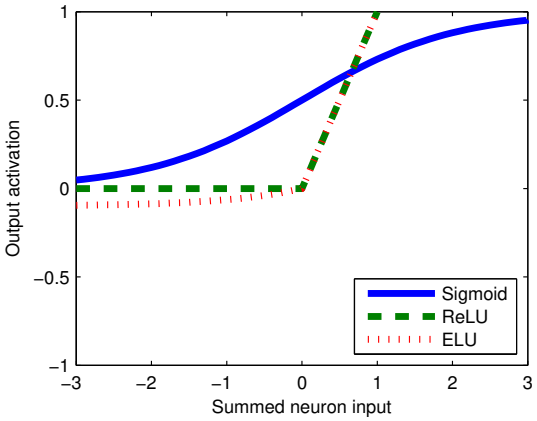

Figure 2.1: Plot of all activation functions used in the paper

The last function we consider in this paper is the Exponential Linear Unit (ELU). For $x > 0$ its output is the same as for a ReLU. In contrast with the ReLU, the ELU function allows for a small activation when $x < 0$. This gives the neurons a chance to recover when the gradients get too high. The ELU used in this paper has an α of 0.1. When $\alpha > 0$ the function is given by:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(exp(x) - 1) & \text{if } x \leq 0 \end{cases} \tag{2.5}$$

It is important to note that while all hidden layers make use of the specified activation function, the output layer always uses a sigmoid function.

2.5 Hidden Layer Sizes

All the activation functions were tested with three different neural networks. The networks all have an input layer of 65 neurons, one for each square on the board and one extra which always holds the bias value. The Q-learning algorithm works with an output neuron for each action, so the output layer is 64 neurons big in every network, again one neuron for every square. The first network we tested uses only one hidden layer of 250 neurons. Therefore this network has $65*250+250*64 = 32250$ weights. The second network makes use of two hidden layers both

containing 100 neurons. This network has $65*100+100*100+100*64 = 22900$ weights. Our last and deepest network uses three layers of all 75 units, with a total of $65*75+75*75+75*75+75*64 = 20925$ weights.

2.6 Dropout

Dropout is a technique used primarily in supervised learning networks. It is a method to reduce overfitting, in other words dropout makes the network more applicable to data outside of the training set. In the training phase the dropout algorithm will conventionally switch off half of the neurons (chosen at random) of an ANN every time a new input is presented [14]. This was not an option with the Q-learning algorithm, since Q-learning learns from the difference between its current and its last state. If we switch neurons on and off between every move, this will have too much influence on the difference between the Q-values. Therefore we chose to make a new selection of active neurons after every finished game. In the testing phase all the neurons will be active, but their outgoing weights will be halved to compensate for twice as many neurons being active.

3 Results

3.1 Experimental setup

In total we ran 18 different experiments, every experiment was repeated 6 times. The results shown here are the averages of these six runs for each experiment. The final score is calculated by averaging the outcome of the games in the last testing round (1 for a win, 0.5 for a tie and 0 for a loss). Every activation function was combined with every neural network size and we ran these experiments once with and once without the dropout algorithm. The indicated activation function is used in every layer except for the output layer which always uses a sigmoid activation function. Tests using different activation functions in the output layer never seemed to improve performance and since the rewards are 0, 0.5 or 1 a sigmoid function is also the logical option. We did a parameter sweep to find the best learning rates for the different activation functions. For the networks with a sigmoid activation function we had the best results with a learning rate of 0.02, for ELU 0.005 and for ReLU 0.00001.
The weights of the ANN are initially set to random values between -0.5 and 0.5. The exploration rate ε starts at 0.1 and decreases linearly to 0 over the course of 2 million training games. This means that for every move in the first game there is a 10% chance that the move will be randomly chosen. Since there are no intermediate rewards we set the discount factor γ to 1.
We train the network for a total of 2 million games. Since our opponent is deterministic, our network would normally almost always encounter the same series of states. To make sure our network is also exposed to different situations, we initialize the board to one of 236 unique states that can be reached within four turns. After every 20,000 games the network plays 472 test games. The test games consist of 2 games per starting position, the agent plays every position once as white and once as the black player. After every test phase we calculate the average score by averaging the outcomes. All training and testing games are played against the same fixed opponent. The fixed opponent takes the action that results in a board with the highest evaluation. The evaluation function simply sums all the positional values for its own discs minus the values for the agents discs. Figure 3.1 shows the positional values that are used by the fixed opponent. The positional values we used were found in a paper describing the application of reinforcement learning to Othello [6].

100	-25	10	5	5	10	-25	-100
-25	-25	2	2	2	2	-25	-25
10	2	5	1	1	5	2	10
5	2	1	2	2	1	2	5
5	2	1	2	2	1	2	5
10	2	5	1	1	5	2	10
-25	-25	2	2	2	2	-25	-25
100	-25	10	5	5	10	-25	100

Figure 3.1: Positional values for the fixed opponent

3.2 Dying networks

For almost all the experiments using a ReLU function and some of the experiments with an ELU

function, the network stops learning during the experiment and starts rendering output that makes no sense. The cause of this behaviour are neurons that no longer react to input. These dead neurons will always output the same value and are the results of exploding gradients. In a network with a rectifier activation function and multiple hidden layers gradients can grow very fast [12]. When we update according to such a gradient the weights connected to the neuron will become so big or small that the actual input becomes irrelevant. For a majority of the ReLU experiments this happens within the first 20,000 games, but sometimes the network starts out learning normally only to fail halfway through. The averages used in the plots are only from the runs that did not die, so it is important to notice that some of the combinations are less reliable than others, while this does not show in the plots. Table 3.1 shows how many of the experiments did not die. For the experiment with a ReLU function and a single hidden layer using dropout, two runs made it. Four did not, so even when ReLU succeeds it is not very trustworthy. For the experiments using an ELU function, multiple hidden layers and no dropout, only half of the runs succeed.

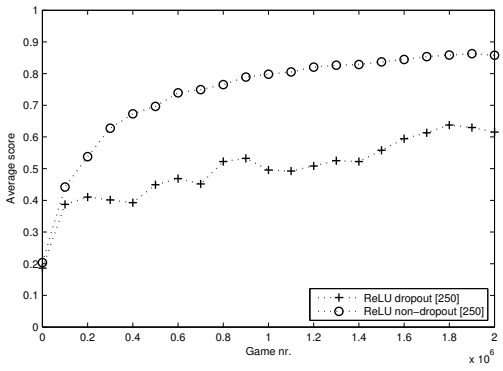

Figure 3.2: Plot of winrate over time for the ReLU function both with and without using dropout

3.3 ReLU results

For ReLU there are only sensible results for the networks with one layer, see Figure 3.2. When we do not use a dropout algorithm, the results are the best. The network with one layer and no dropout

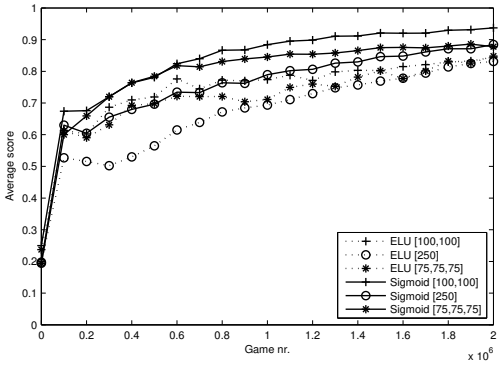

Figure 3.3: Plot of winrate over time for sigmoid and ELU functions without using dropout

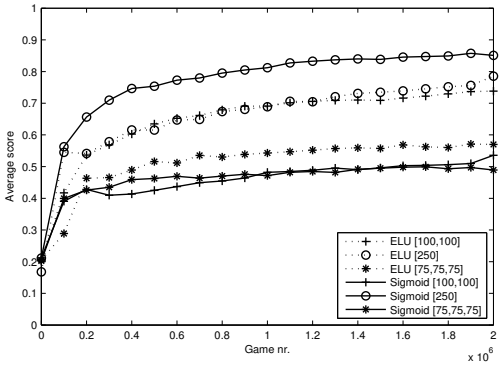

Figure 3.4: Plot of winrate over time for sigmoid and ELU functions with a dropout algorithm

eventually has an average score of 0.87. The single layer network that uses a dropout algorithm performs worse and is also prone to dying neurons. The dropout network only completed the two million games in two out of six runs.

3.4 Results without dropout

Figure 3.3 shows the results for both the ELU and sigmoid activation functions without dropout. This is where we see the best results. Especially the network with two layers and a sigmoid function performs really well: after two million games it has an average score of 0.95. This is the best result we obtained in our research.
The other two sigmoid networks both end with an average score of 0.89. The network with three layers

Function	Layers	Dropout	Runs	Final Score
sigmoid	1	No	6/6	0.89 ± 0.01
sigmoid	2	No	6/6	**0.95 ± 0.01**
sigmoid	3	No	6/6	0.89 ± 0.02
sigmoid	1	Yes	6/6	0.86 ± 0.02
sigmoid	2	Yes	6/6	0.54 ± 0.05
sigmoid	3	Yes	6/6	0.49 ± 0.02
ELU	1	No	6/6	0.84 ± 0.03
ELU	2	No	3/6	0.85 ± 0.03
ELU	3	No	3/6	0.86 ± 0.05
ELU	1	Yes	6/6	0.79 ± 0.01
ELU	2	Yes	6/6	0.75 ± 0.02
ELU	3	Yes	6/6	0.58 ± 0.02
ReLU	1	No	6/6	0.87 ± 0.02
ReLU	2	No	0/6	NA
ReLU	3	No	0/6	NA
ReLU	1	Yes	2/6	0.62 ± 0.13
ReLU	2	Yes	0/6	NA
ReLU	3	Yes	0/6	NA

Table 3.1: Table showing completed runs, final scores and standard deviations for every combination

seems to learn quite a bit faster than the network with only one layer. The networks using ELU activation functions are less stable than the sigmoid networks. They also perform less well, all end with an eventual score of around 0.85. Just like we saw with the sigmoid networks we can see that extra layers seem to speed up the learning process.

3.5 Results with dropout

The networks utilising a dropout algorithm perform generally less good than their counterparts without dropout. In Figure 3.4 the sigmoid network with only one layer performs best, the other sigmoid networks with more than one layer perform less good with final scores that are all lower than 0.55. The networks with an ELU activation function perform less good than without a dropout algorithm, but interestingly all of the networks kept learning and none died.

4 Conclusion

In this research we have tried to find out what the influence of using different activation functions, different amounts of hidden layers and using a dropout algorithm is on the performance of an ANN learning to play Othello. We found that after two million training games an ANN with a sigmoid activation function, two hidden layers and no dropout algorithm obtained the best results.

4.1 Performance of the networks

The most basic network with a sigmoid function, one layer and no dropout already ended with an average score of 0.89, the only network with better performance is exactly the same but uses two hidden layers as already mentioned above. For other activation functions two hidden layers do not seem to improve performance though. When using dropout, extra layers do even seem to make the performance worse, especially in combination with the sigmoid activation function. The effect of adding more layers in combination with a ReLU activation function is even more drastic since none of these combinations survived until the two millionth game. Using three hidden layers never increased performance compared to using one hidden layer. Both the ReLU and ELU function do not make the network perform better, the same is true for the use of a dropout algorithm.
To summarize our findings: the performance of an ANN learning to play Othello can be improved by adding an extra hidden layer, but not by using a different activation function or using a dropout algorithm.

4.2 Reliability of the networks

The sigmoid function is by far the most reliable since all runs were completed with rather good results. Simpler networks with only one hidden layer also proved to be more reliable than bigger networks. Especially more than one consequent ReLU layer make the gradients explode very quickly. In the networks using an ELU activation function, dropout seems to increase the reliability.

5 Discussion

One of the major problems we encountered in this research was the fact that a considerable amount of runs died before completion. We did

not consider these dying runs in our plots, since we wanted to examine the learning capabilities of the different activation functions and taking these runs into account would have blurred the results of the good runs. However it might be interesting to find a way to avoid these dying runs. One possible way to improve the reliability of the failing ReLU networks could be to even further lower the learning rates. In this research learning rates as small as 0.000005 have been tried with no success, but it could very well be that even smaller learning rates could perform better. Another way to fix the problems with the ReLU function might be using a bounded ReLU variant [15]. With bounded ReLU the activation is limited to a certain threshold. Instead of limiting the activation itself, it would also be possible to clip the derivative of the activation. This method is called gradient clipping and has been used to control the exploding gradient in recurrent neural networks [16].

Looking back, the use of a dropout algorithm might not have been an obvious choice. Dropout is used to prevent over-fitting on the training set and makes the network better applicable to other data. In our research the training and test set are basically the same. This means over-fitting on the training set would even be desirable. The fact that our opponent will always make the same fixed moves also encourages over-fitting. Trying to make the network more general should result in worse performance, as can be seen in our results. It should not be hard to come up with a reinforcement learning problem where the training and test set would be different. Then we might be able to really test the dropout algorithm's capabilities. One way to make the current problem better suitable for dropout could be implementing a less predictable opponent. One final remark might be that looking at figures 3.2-3.4 shows that some of the networks might not have fully converged to their optimal policies. Therefore training over more games could increase their final average score.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge: The MIT press, 1998.

[2] G. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38(3), p. 5868, 1995.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 518, p. 529533, 2016.

[4] N. Altuntas, E. Imal, N. Emanet, and C. Ozturk, "Reinforcement learning-based mobile robot navigation," *Turkish Journal of Electrical Engineering and Computer Sciences*, vol. 24(3), pp. 1747–1767, 2016.

[5] A. D. Redish, S. Jensen, A. Johnson, and Z. Kurth-Nelson, "Reconciling reinforcement learning models with behavioral extinction and renewal: Implications for addiction, relapse, and problem gambling," *Psychological Review*, vol. 114(3), pp. 784–805, 2007.

[6] N. J. van Eck and M. van Wezel, "Application of reinforcement learning to the game of othello," *Computers and Operations Research*, vol. 35(6), p. 19992017, 2008.

[7] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8(3), p. 279292, 1992.

[8] M. van der Ree and M. Wiering, "Reinforcement learning in the game of othello: Learning against a fixed opponent and learning from self-play," *Proceedings of IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning: ADPRL*, 2013.

[9] M. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, and G. Hinton, "On rectified linear units for speech processing," *IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 2013.

[10] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," *In International Conference on Artificial Intelligence and Statistics*, p. 315323, 2011.

[11] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *CoRR*, vol. abs/1505.00853, 2015.

[12] K. Kurach, M. Andrychowicz, and I. Sutskever, "Neural random-access machines," *CoRR*, vol. abs/1511.06392, 2015.

[13] D. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units," *CoRR*, vol. abs/1511.07289, 2015.

[14] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *CoRR*, vol. abs/1207.0580, 2012.

[15] S. S. Liew, M. Khalil-Hani, and R. Bakhteri, "Bounded activation functions for enhanced training stability of deep neural networks on visual pattern recognition problems," *Neurocomputing*, vol. 216, p. 71834, 2016.

[16] R. Pascanu, T. Mikolov, and Y. Bengio, "Understanding the exploding gradient problem," *CoRR*, vol. abs/1211.5063, 2012.