Machine Learning: Project 2017-2018

Florian Van Heghe & Timothy Werquin

May 2017

Abstract

The assignment of this project is to created a machine learning based dots and boxes player. The player has to be able to play on different board sizes without retraining and it should be able to force choosing a move within a given time. We decided that we wanted to create a player that was as general as possible without the least amount of knowledge about the game possible. Our approach to this problem was to combine reinforcement learning with a multidimensional recurrent neural network to generate the Q-values. Finding a good opponent to train against proved to be important, a player that trained successfully against a random opponent will perform very poorly against a human player. Another important decision how to represent the board. Including game specific features in this representation could aid the player but we chose to keep it as general as possible. Different board representations, whether they are symmetrical and how symmetry improves the agent are discussed. After discussing the standard approach the report covers some extensions and tweaks made to this approach such as double Q-learning and prioritised experience replay. Finally agents with different configurations are evaluated and compared.

Contents

1 Problem Statement

The goal of this project is to implement an agent to play dots and boxes using machine learning techniques. The assignment includes some bigger challenges: forcing a move within a certain time and playing on different board sizes without any retraining. Especially the latter provides a real challenge since the agent will need to learn general features such as chains and strategies to be able to play successfully on arbitrary board sizes. When designing a board representation one could try to represent these features to aid the agent in using these features. We chose to do the contrary and try to minimise the included knowledge about the game to keep our board representation and agent as general as possible. The downside of this approach is that the agent has to learn about these features itself without any aid or pointers. The assignment gives some pointers about two possible approaches to solving this problem. The first being Monte Carlo Tree Search and the second being Reinforcement Learning. We decided on a Reinforcement Learning approach but since the action-state space becomes very large very quickly with increasing board size there is a need for a generalisation to predict Q-values. A neural network seems ideal for this kind of task but it does not scale for different input sizes so another scaling technique is needed.

2 Approach

This chapter discusses our approach to creating a dots and boxes player for a variable board size. It discusses the various research questions as well as how symmetry can improve our player and how training against opponents of different playing levels influences the player. The player is coded in Python using Tensorflow as library for neural networks.

2.1 Finding a good opponent

One of the most important factors in training the neural network is what opponent it will be training against. The default player is a fully random player, meaning that every turn it will play a randomly selected legal move. Training against this player and beating it is insignificant because when playing against a player that has an actual strategy, it will lose every game. So to be able to train a player to perform on a (close to) human player level it needs to be trained against a better player. Since training against a human is impossible simply by the number of training games that are required, a better player is needed. A simple yet efficient strategy (especially for smaller boards) is playing greedily, meaning that whenever it is possible to capture a box, capture it and otherwise play a random move.

This greedy player is a good opponent to train against since it has the knowledge of capturing boxes. It also has a lot of randomness which helps with exploring more states. The greedy player is also limited, especially on boards larger than 3x3, because it does not know of the concept of chains and it can also construct boxes for his opponent. A lot of improvement on an opponent player is possible, especially on larger boards but also on smaller boards since the greedy player is still not optimal.

The results of training the Q-learning agent from section 2.2 against a random player and then evaluation against a greedy player are found in table 1. This shows that training against random is not a good strategy for learning, while table 2 shows that the Q-learning agent can learn to beat the greedy player when training against it.

opponent	Win percentage	Loss percentage	Draw percentage
random	72.5%	6.0%	21.5%
greedy	15.5%	43.3%	41.2%

Table 1: Results of training the Q-player on a 2x2 board over 200,000 games and evaluating over 1000 games.

A better player for fixed-size small boards is presented in the next section.

2.2 Implementing Q-learning

To get started we first implemented a naive version of Q-learning. This version uses a hashmap indexed by the state-action pair to store the Q-values. This implementation uses a generic board representation and game class that can be reused later to implement the deep-Q version of the agent. The only rewards that are given are +1 for winning and -1 for losing a game, drawing results in no reward for both player. For each player the other player is regarded as part of the environment. This means that the first player gets a reward only after the second player has made a move.

After training over 100,000 games in 80 seconds this version is capable of beating the greedy player on a 2x2 board. Training on a 3x3 board takes much longer and requires a much larger state table, after training 100,000 games a 11% winrate is reached against the greedy player. Some results from various training sessions can be found in table 2. A limitation of this approach is that it isn't scalable to different board sizes. But it can serve as a benchmark for the deep Q-learning version of the player and as a very strong opponent to train against. The resulting Q-values can also be used in supervised training to bootstrap or test the deep-Q neural network.

Training games	Win percentage	Loss percentage	Draw percentage
50,000	24%	28%	48%
100,000	30%	17%	53%
200,000	33%	16%	51%

Table 2: Results of training the Q-player on a 2x2 board on increasing amounts of games and evaluating over 1000 games.

2.3 Board representation

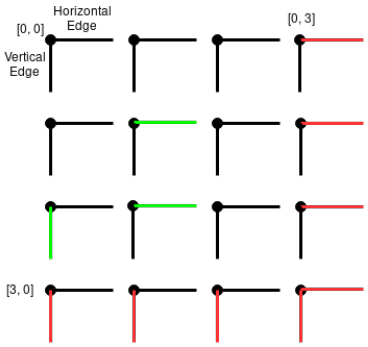
An important choice when designing neural networks is feature selection. Our approach was to give the network as little predefined knowledge as possible and test whether the network could learn important features such as chains on it's own. To ensure the network can learn the necessary features a good board representation is essential. A problem with dots and boxes is that the relevant features are the edges on a grid which means the board can't simply be represented as a matrix or tensor. Throughout the project we implemented three different board representations, the first is the naive representation as used in the provided code. The second and third representation are more thought out and symmetrical.

2.3.1 Naive representation

The first representation uses an $(n + 1) \times (m + 1) \times 2$ tensor to represent a board of size $n \times m$. The first and second dimension represent row and column of the dot and the third dimension represents the vertical or horizontal edge starting in that dot. An element of this tensor is zero for an empty edge and one for a placed edge independently of which player placed that edge, since the player that placed the edges is insignificant. An example board is shown in Figure 1. This representation has a few limitations, there are a lot of edges in the tensor that are outside the board, these edges are represented with a -1 value. Anther limitation is that the representation isn't symmetrical. The symmetry of the representation is important as otherwise the network has to learn all eight transformations of the board separately.

2.3.2 Box representation

As an attempt at a more symmetrical representation we split the grid in it's different boxes and represented a box as an array of four elements indicating the presence of an edge at a certain position in the box. This representation creates a $n \times m \times 4$ tensor for a $n \times m$, an example can be found in Figure 2. This approach has as an advantage that there are no edges that are outside of



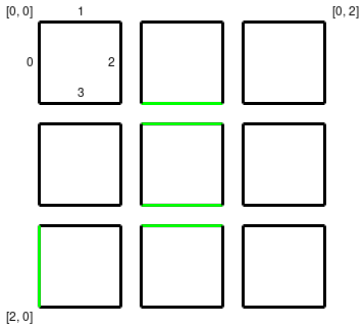
(a) The board

```
[[[ 0, 0], [ 0, 0], [ 0, 0], [ 0, -1]]
 [[ 0, 0], [ 0, 1], [ 0, 0], [ 0, -1]]
 [[ 1, 0], [ 0, 1], [ 0, 0], [ 0, -1]]
 [[-1, 0], [-1, 0], [-1, 0], [-1, -1]]]
```

(b) The resulting tensor

Figure 1: An example board using the naive representation

the board. But some edges are represented twice, to overcome this we take the average of the two edges when evaluating the Q-values. Another limitation is that it still isn't symmetric as a rotation of a board would change the order of the inner edges array as the edges changed position within a box.



(a) The board

```
[[[0, 0, 0, 0], [0, 0, 0, 1], [0, 0, 0, 0]],
 [[0, 0, 0, 0], [0, 1, 0, 1], [0, 0, 0, 0]],
 [[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0]]]
```

(b) The resulting tensor

Figure 2: An example board using the box representation

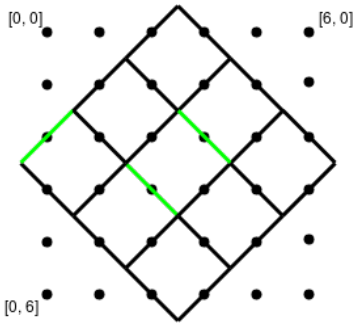
2.3.3 Edge representation

To get a completely symmetrical representation each edge is represented separately, but edges don't form a nice grid so storing them in a matrix is more difficult. To overcome this the board representation is rotated 45° clockwise. This results in a dense edge packing that can be stored in a matrix. For an $n \times n$ board this results in an $(n+m) \times (n+m)$ matrix. This representation is completely symmetrical and has a limited amount of redundant elements in the resulting matrix. An example can be found in Figure 3.

The symmetry of this board means that when using multidimensional recurrent neural networks the network doesn't need to learn all eight dihedral transformations. This should mean that the agent learns eight times as fast as an agent that doesn't take symmetry into account. This is shown in table 3 where the same network is trained for 10,000 games on the box representation and the edge representation. The difference isn't as big because the edge representation has a larger resulting matrix which means the recurrent network needs to learn to store more information in it's state which slows down training.

Method	Win percentage	Loss percentage
box representation	37.0%	63.0%
edge representation	43.2%	56.8%

Table 3: Evaluation results after training for 10,000 games against a greedy player on a 3x3 board using two different board representations.



(a) The board

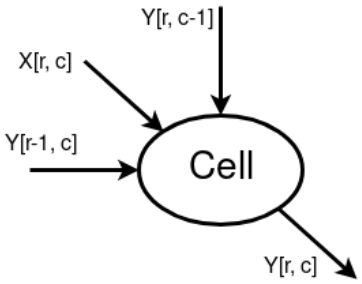
```
[[-1, -1,  0,  0, -1, -1],
 [-1,  0,  0,  0,  0,  0],
 [ 1,  0,  0,  1,  0,  0],
 [ 0,  0,  1,  0,  0,  0],
 [-1,  0,  0,  0,  0, -1],
 [-1, -1,  0,  0, -1, -1]]
```

(b) The resulting matrix

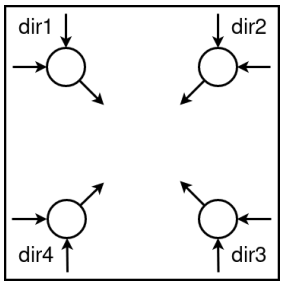
Figure 3: An example board using the edge representation

2.4 Multidimensional Recurrent Neural Networks

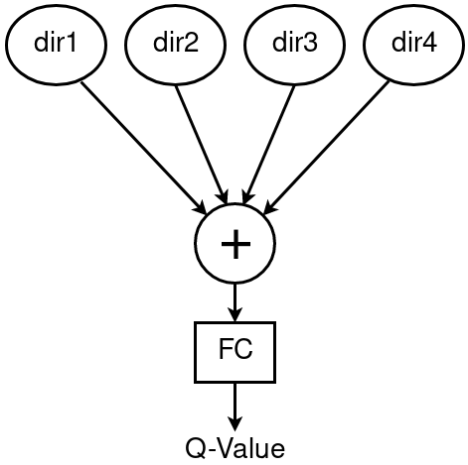
To make our neural network adapt to multiple board sizes we used Multidimensional Recurrent Neural Networks (MDRNN) [3] this is an extension of recurrent neural networks to work over multiple dimensions. In our implementation a 2D RNN was used to swipe over the board representation. This means that a recurrent cell receives the state of the previous cell in both the x and y directions at each point of the grid (see Figure 4a). To get information about the whole board to each cell the cell is ran four times, once for each diagonal direction of the board as illustrated in Figure 4b. Because the board is symmetrical and the result of the neural network should be the same for all rotations, the same cell with the same weights is used in all four directions. As the result should also be the same for diagonal mirroring of the board, the weights of the X and Y state inputs are also the same.



(a) A multidimensional cell with the inputs of the current board element, previous cells state and the cell output



(b) The four cells are each unrolled in a different direction over the board.



(c) The fully connected layer combines the four directions to get the Q-value.

Figure 4: Overview of the MDRNN

The result of these four cells is then combined, as each direction should carry the same weights for symmetry they are summed together. The output of this RNN layer is then passed to a fully connected network that takes as input the state of a RNN cell and outputs the final Q-value for that position in the board, this is shown in Figure 4c. Different network shapes have been tested by varying the RNN cell (simple tanh vs GRU cell) and by varying the depth of the final fully connected layer. Our test network for all following sections uses a simple tanh cell and a fully connected layer with one hidden layer of 64 relu units and a final linear layer providing the Q-values for each board position.

2.5 Deep Q-learning

The final agent is a Deep Q-learning implementation based on the paper by Mnih et al. [2]. The first implementation used a simple fully connected network to produce the Q-values for a fixed board size of 2x2. After this MDRNN were used to make a board-size independent network, this slows down training but when trained on a 3x3 board good results are yielded on a 2x2 board. To improve learning and results various extensions to the standard deep Q-learning algorithm, such as double deep Q-learning and prioritised experience replay, were implemented.

2.5.1 Implementation

During training the agent is asked to make a move, using epsilon-greedy exploration the agent either selects a possible move at random or uses the move with the highest Q-value. After the opponent plays the agent receives a reward which will be zero during most of the game except when the game ended and the agent receives reward +1 for winning or -1 for losing. The exploration probability is reduced during playing on a linear schedule, this ensure the agent is trained on plays it would also make during evaluation. After training for 1000 games the agent is evaluated on 100 games. During evaluation the agent always picks the move with the highest Q-value. Training is always done on a fixed board size as changing the board size would require unrolling the MDRNN again.

As in the Deepmind paper [2] a separate Q-value and target network are used. The Q-value network is used to select the next move while the target network is used when updating the network to get the maximum reward of the next state. This prevents oscillation in changing both the predicted Q-value and the target Q-value at the same time. The target network is updated after a fixed number of training steps (1000 plays in our case).

Only training on the last action would mean the network overfits on it's most recent policy. To prevent this experience replay is used. Every time the agent receives a reward the state, action, next state and reward are stored in a queue of a fixed size. While training a random batch is sampled from this queue and trained upon. This removes the correlation between subsequent actions in the training batch.

Training for 1000 games on a 3x3 board takes about 400 seconds on a regular laptop cpu (i7-5600U). Evaluating is very fast as per move the network needs to be evaluated forwards just once, one move takes on average 4ms on a 3x3 board, while playing a game takes about 80ms. Because evaluating is this fast there is no need to force a decision within a given time. The total weights take about 70 KiB on disk and could further be reduced by only saving the q-value

network and not the target network.

2.5.2 Double Q-learning

To further improve performance the double Q-learning algorithm was used, this algorithm uses the value network to select the maximum next play but uses the target network for the actual Q-value estimation. This prevents overestimating the Q-values as random noise in the estimation causes the Q-values to increase as detailed in the paper by van Hasselt et al. [5]. By using a separate network to select the maximum and evaluate the maximum the influence of this noise is reduced.

However in our results the double Q-learning approach doesn't gain much advantage during training over the regular Q-learning as can be seen in figure 5. Yet the final evaluation results are better with double Q-learning as seen in table 4. The difference in final evaluation result could be explained by the randomness in evaluation games.

Method	Win percentage	Loss percentage
regular Q	43%	57%
double Q	58%	42%

Table 4: Evaluation results after training for 30,000 games against a greedy player on a 3x3 board.

2.5.3 Prioritised experience replay

Selecting a random batch from the experience queue uniformly means that some actions are seen more by the agent then other. For example the initial state will be much more prevalent in the queue than any other state. To prevent overfitting on these states prioritised experience replay can be used [4]. This method weights experiences by the Q-value estimation error. This ensures unexpected experiences will be seen more during training by the agent.

As can be seen in figure 6 the priority experience replay doesn't have any meaningful influence on the average reward during training. After training for 30,000 games the evaluation results are given in table 5, this indicates that prioritised experience learning doesn't give any advantage in our use case.

Method	Win percentage	Loss percentage
no priority	58%	42%
priority	54%	46%

Table 5: Evaluation results after training for 30,000 games against a greedy player on a 3x3 board.

2.6 Gated Recurrent Units

A final test was to use a Gated Recurrent Unit (GRU) [1] as the cell in the MDRNN. This resulted in a drop of the final evaluation after 30,000 games of training on a 3x3 board. As a GRU cell should be better at remembering over longer distances, the performance should be better with larger board sizes. After testing this also wasn't the case, which is why we didn't use a GRU cell in our final model.

2.7 Evaluation

Our best model was the plain double Q-learning agent which was trained for 30,000 games against a greedy player on a 3x3 board of which the avgerage reward in the last 100 games during training can be seen in figure 5.

Opponent	Win percentage	Loss percentage
random	99%	1%
greedy	58%	42%

Table 6: Evaluation results after training for 30,000 games against a greedy player on a 3x3 board.

The network can be evaluated against different opponents on a 3x3 board as shown in table 6. This shows that the agent hasn't learned to win in all cases against a greedy player but can win in almost all cases against a random player.

As the model should work on multiple board sizes we can also evaluate on a 2x2 board against various opponents. The results of this are given in table 7. The agent has developed a good policy on a 2x2 board which allows a high win rate against a random opponent. Even against a greedy player the agent can win in a lot of cases. The tabular Q-player from section 2.2 is deterministic and can't win against the deep Q-player, which means it will always lose.

And finally evaluating on a larger board is done on a 4x4 board with results in table 8. This shows that while the network learned some strategies as shown in the win rate against random play, it didn't generalise very well as it still loses a lot against a greedy player.

Opponent	Win percentage	Loss percentage	Draw percentage
random	78%	2%	20%
greedy	25%	27%	48%
tabular Q-player	100%	0%	0%

Table 7: Results of the final agent on a 2x2 board after only training on a 3x3 board. The tabular Q-player is the player from section 2.2 trained for 200,000 games against a greedy player.

Opponent	Win percentage	Loss percentage	Draw percentage
random	81%	15%	4%
greedy	0%	100%	0%

Table 8: Results of the final agent on a 4x4 board after only training on a 3x3 board.

3 Conclusion

The results show that it is possible to use deep Q-learning to win at dots-and-boxes and to learn some features such chains but the current approach doesn't extend to larger board sizes than those which were trained upon. A better approach for board games would be Monte Carlo tree search as games can be rolled out. Q-learning is more applicable in environments where this isn't possible such as the Atari games of the original deep Q-learning paper.

In total we each spent about 70 hours on this project. The time was divided over different parts in the following way: 30 hours on implementation of the agent and all extensions, 30 hours on training, tuning, experimenting and testing and 10 hours on this report.

References

[1] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *ArXiv e-prints*, Dec. 2014.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with Deep Reinforcement Learning. *ArXiv e-prints*, Dec. 2013.

[3] T. Schaul and J. Schmidhuber. A scalable neural network architecture for board games. In *2008 IEEE Symposium On Computational Intelligence and Games*, pages 357–364, Dec 2008. doi: 10.1109/CIG.2008.5035662.

[4] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized Experience Replay. *ArXiv e-prints*, Nov. 2015.

[5] H. van Hasselt, A. Guez, and D. Silver. Deep Reinforcement Learning with Double Q-learning. *ArXiv e-prints*, Sept. 2015.

Appendix

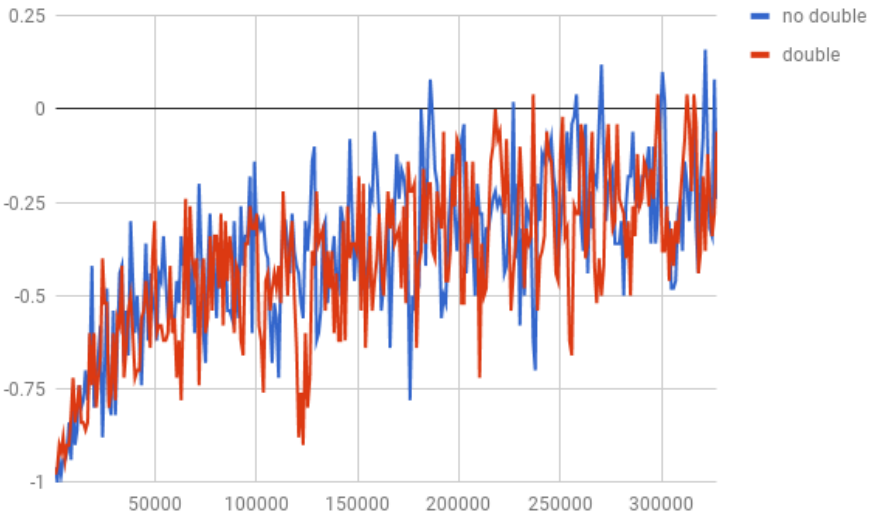


Figure 5: The average reward of the last 100 games while training with and without double Q-learning against a greedy player on a 3x3 board.

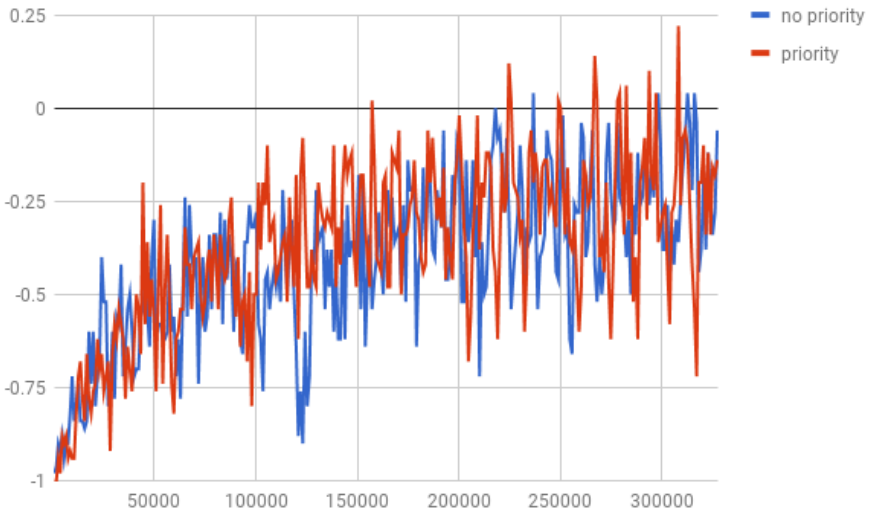


Figure 6: The average reward of the last 100 games while training with and without prioritised experience replay against a greedy player on a 3x3 board.