Lutharsanen Kunam
Matteo Brändli

University of Zurich[UZH]

**Department of Business Administration**

**Chair of Quantitative Business Administration**

## Seminar project Deep Reinforcement Learning

# Programming the game 2048 with methods of Deep Reinforcement Learning

**Submitted by Group ShaLuMa:**

Lutharsanen Kunam, 15-700-875

Matteo Brändli, 10-733-566

**Supervisor:** Dr. Robert Leonard Earle

**place, submission date:** Zurich, 12.12.2019

Lutharsanen Kunam
Matteo Brändli

# Table of content

Lutharsanen Kunam
Matteo Brändli

# 0. Contribution of the group members to the project

| | |
|---|---|
| Lutharsanen Kunam | Ideas, discussion, presentation, first tries, got code running, debugging, testing, report and questions |
| Matteo Brändli | Ideas, discussion, presentation, first tries, debugging, methodology, testing report and question |

* A first report version of "Pre-test runs" and the questions has been made by Shabarna, who later dropped out of the project.

# 1. Project- programming 2048 with deep reinforcement learning

## 1.1 Our reinforcement problem

Our objective was to program an algorithm, which solves the game 2048 with Reinforcement Learning methods learned in class. We used python to program. The goal of the game 2048 is to combine two same-value numbered tiles, so that they merge and result in their sum, to finally attain the tile 2048. However, in the game it is possible to attain higher numbers. Nonetheless, after attaining 2048 the player has won the game.

For our project, we have used three methods, namely Q-Learning, SARSA and Deep Q-learning to implement the learning. We used an open AI environment from gym and installed the version gym-2048. This environment has an output of a state space with a 4x4 matrix. We have mostly transformed this 4 x 4 matrix into a list with 16 elements to make it easier to use them in the model. The discrete action space consists of four elements for all the states, even if a particular action would be unfeasible in a specific state. We had to control for this in our tabular models, but the Deep-Q learning appeared to avoid this problem by itself. The four different actions possible were: swipe up, swipe right, swipe down and swipe left. The reward was given by the environment as the sum of merged cells in a step. The goal of our models is to maximize the sum of the rewards to solve the 2048 task. We ran multiple versions of code with very mixed results, even in terms of getting software and code to work. Unfortunately, we could not solve the problem to save agents' progress, i.e. we were not able to import/export Q-tables or neural weightings during this project.

To visualize and interpret the learning process, we are using different graphs and plots. We will divide most of our episodes into groups corresponding to percentiles (always 100 groups). We expect to see an improvement from one batch to the next batch. With our plots we will analyse the win statistics. In each test-run we will define a winning rate - we are going to set a tile number as a goal and count how often the model has achieved this goal. The goal was set at 256 to obtain meaningful win statistics with tabular models. Other plotted statistic will present the maximum value tile reached, the average maximum tile reached, and the average rewards obtained by the interaction with the environment. In all plots, we expect an improvement of the calculated values, especially for the "average reward" statistic (which should be optimized by the models). As a comparison for performance we had an agent act randomly at any point. If they fail to outperform such an agent our models may not have learned much.

## 1.2 Q-Learning

The first model we used to solve the game 2048 was Q-learning. In our opinion, it was theoretically a well-suited model for the 2048 problem. The reasons are that both reward and transitions can be observed and that there is a natural limit to the reward due to the rules of the game. As a result, a Q-Table can be computed, and one should not have to worry too much about discounting. We thus chose discounting at gamma=0.99 to explore the foresight without completely neglecting possible loops.

We started to implement the Q-Learning model by adding a json file to the model, where it should store the Q-table. Due to type issues we stopped this version. At the end, we implemented the Q-table similarly to SARSA. In fact, both models are almost identical but for the updating function. Our goal is to compare both approaches and see whether one is more suitable to our 2048 environment.

Many functions were originally taken from the youtuber Machine Learning with Phil(add hyperlink later…). Much of the code has been modified to cope with our environment and to visualize the data. In the process of developing the Q- model our agent got continuously stuck in the same state where no step is activated (due to the supposed "optimality" of that move by Q considerations) and could not overcome this problem with the computational power at hand. This has been overcome by the choseandcheck function that tracks if the state changed after an action and reduces the corresponding Q value directly in case of an inactive move. Like this we avoid infinite loops and we reused this code snippet for the SARSA model in the next chapter. The function checks, if the state changes and if it doesn't, it reduces the Q-value. With this function we want to make sure, that our model will avoid these steps in future and thus avoid dead-ends.

### 1.2.1 Pre-test-runs

In our first test-run we run 5'000 episodes. This means that each batch contains 50 episodes. We have used alpha as the learning rate with a value of 0.2, as gamma we have used 0.99 and epsilon, which is the exploration rate, decreases from 1 to 0.0002 over time. Our goal in this test-run was 256. Due to a bug, we couldn't produce the average score in this test-run.

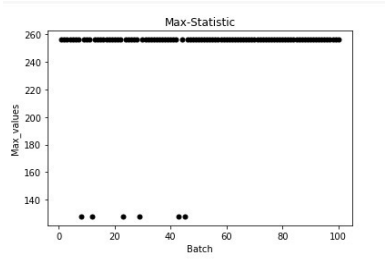



*Figure 1:maximal values statistic*

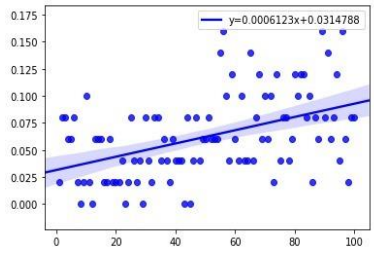
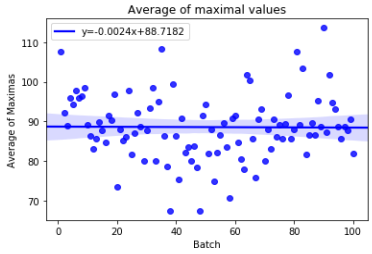
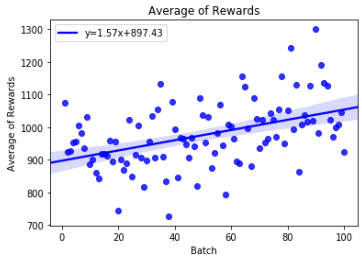



*Figure 2:winning-statistic*

Following our first test-run we can see that there is a positive correlation between the games and the winning statistic. This shows as that our model improves when exploitation is reduced, and further episodes are played. In our next try, we will raise the goal from 256 to 512, to see, if our model can attain higher tiles. The y-axis of Figure 2 stands for the games won.

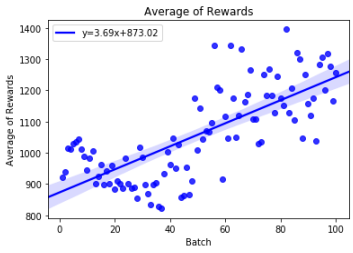
In our second test-run we decreased alpha to 0.1 to see what the effect of a lower alpha is on our model. Intuitively, we would say that a lower learning rate should have a negative impact on the model. But we know from lecture, that there is an issue of overfitting and therefore we want to avoid to overfit our model. We also increased the goal to 512 to see, if our model can achieve this goal too.

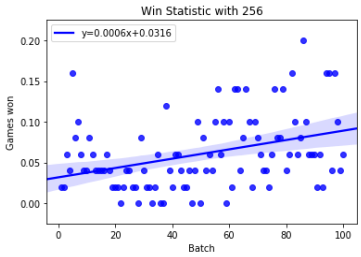



The model never reached this value and therefore the winning statistic didn't make any sense in this test-run. The maximal value from this test-run was 256. In our opinion, the *graph with the average of rewards* is a suited graph to visualize the progress of the network, especially the slope of the graph and the average reward increase to 975.93 suggest faster learning. The regression line of the figure *average of maximal values* shows a constant line.

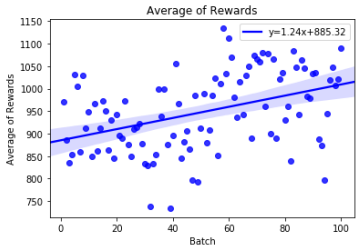
In our third test-run, we increased our alpha to 0.3. Therefore, we wanted to explore the trade-off of a higher or lower learning rate. We have decreased the goal back to 256 to have a better view on how t how often the model reaches the goal. Due to runtime time restriction, we weren't able to able to reach 512 often enough for meaningful interpretations.

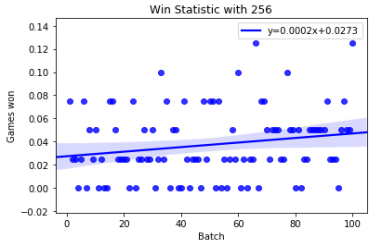



An increase of the learning rate to 0.3 had an enormous impact on the slope of the average rewards graph. On average the reward increases per batch by over 3 units. Overall average 1057.52. This is an increase of over 100% compared to the graph with a learning rate of 0.1. We also see an increase of the percentual win rate over the batches. The maximal value, which was achieved by the model was 256.

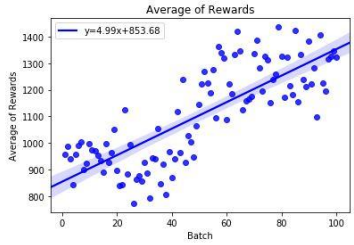
In the fourth test-run we decreased again the learning rate to 0.01 to verify our hypothesis, that the scope of the reward graph will decrease, if we decrease our alpha. There was an additional exception, as we were running only 4000 episodes instead of 5000. All the other parameters stayed the same.

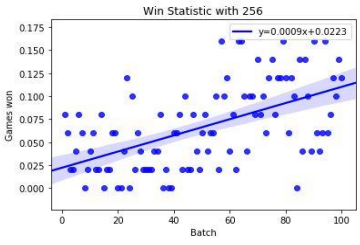



The slope of the reward graph decreased in comparison to the third test-run due to a lower learning rate possibly driven by further reduction of the already limited episodes. Also, the slope of the winning statistic is 3 times smaller than in the previous test-run. The maximal value, which was achieved in this test-run was 256, because, it performs worse than in the previous test-run.

In the fifth test-run, we once again increased the learning rate alpha to 0.5. The maximal value of the test-run was still 256.We expect to have a higher slope in the rewards graph and therefore to have the best possible result so far.




As expected, this test-run outperformed all the previous test-runs. The slope of the reward graph is quite steep, and you could even see a positive correlation between the batches and the reward without a regression line. The reward increases per batch on average by 5 units.

### 1.2.2 Q-Learning test series: experimental design & results

Our Q-Learning agents vary along two dimensions. There are two epsilon schemes; whereas one is a monotonously linearly decreasing epsilon, floored at 0.01 the other scheme is a discontinuous epsilon development with multiple discontinuous jumps of epsilon to a higher level. This should reduce the risk of ending up at a non-optimal solution by continuously varying exploration and exploitation. The other varying dimension are the alphas, chosen at 0.1, 0.5 and 0.8. After 5000 training episodes, there are performance tests with 500 episodes. Results are always expressed in percentiles, thus the average result over the percentiles (100 datapoints, in test mode 5 episodes are pooled to one datapoint).

As a comparison for the performance metrics we also specified 2 random agents, that randomly chose an action at any point for 5000 episodes. Those 2 agents still perform tabular computations of Q values with alpha = 0.5 and 0.8. The performance of these are again tested for 500 episodes with epsilon=0. Therefore, we can compare whether the other parametrizations outperform the ones learning off a random agent to evaluate the parametrizations' effectiveness. Thus, there are 2x3 +2=8 specifications for testing and comparison.

Q-Learning summary table of parametrizations and main performance measures after 5000 training episodes[1]. The random agents' performance is measured from 5000 training episodes.

| alpha | epsilon | Win percentage | Avg max tile | Avg reward |
|---|---|---|---|---|
| Alpha = 0.5* | 1, training | 0.047 | 94.97 | 955.65 |
| | Test with eps=0 | 0.104 | 106.61 | 1213.52 |
| Alpha = 0.8* | 1, training | 0.045 | 94.40 | 959.35 |
| | Test with eps=0 | 0.086 | 107.60 | 1281.7 |
| Alpha = 0.1 | Monotonous | 0.044 | 86.99 | 1017.93 |
| Alpha = 0.1 | Nonmonotonous | 0.067 | 93.18 | 1037.97 |
| Alpha = 0.5 | Monotonous | 0.094 | 108.61 | 1256.05 |
| Alpha = 0.5 | Nonmonotonous | 0.115 | 107.11 | 1240.44 |
| Alpha = 0.8* | Monotonous | 0.126** | 109.64 | 1273.13 |
| Alpha = 0.8* | Nonmonotonous | 0.100** | 109.42 | 1260.52** |

*The automatic average performance calculations have been added in time for the models with alpha = 0.8 and the random agents. Those numbers do not contain compounded rounding errors. **These estimates differ from the manual calculations, by -0.024 (monotonous win %) and +0.01 / -0.23 (non-monotonous win % / avg reward). The effect on the main variable avg reward is negligible, whereas the win % may have lost some comparative power as statistic.

### 1.2.3 Q-Learning test series: comparison

Agents with alpha = 0.1 fail to clearly outperform the random agent (epsilon=1, training) and only slightly outperform randomness by the average reward measure. Nevertheless, Q learning off the random agent outperformed those low alpha models. Alpha = 0.5 and Alpha= 0.8 performed comparably well and both outperformed random agent and Q learning off random agent in all measures. Tabular Q learning does not seem suitable to low levels of alpha, like 0.1, at least early in the learning process.

There does not seem to be a clear favourite among the epsilon schemes. The non-monotonous schemes tend to perform better in low alpha specifications. whereas the monotonous scheme seems superior at alpha = 0.8. At alpha = 0.5, there is conflicting evidence: the monotonous scheme slightly outperforms regarding average max tile and average reward but falls short of the non-monotonous scheme in the probability to reach the tile 256 (win %). Nevertheless, the differences are rather small and further testing would be needed for conclusive results.

Rather surprisingly to us, Q Learning off a random agent is quite successful and with alpha=0.8 is the best performing Q-Learning model. This might suggest that an epsilon decay scheme at a slower rate might be a successful alternative specification, as high epsilons only seem to decrease training performance, but affects less the testing performance with full exploitation.

[1] We initially forgot to specify those measures in the code and thus extrapolated the average of the test performance as intercept + 50*slope of each statistic in the performance test. Rounding errors may have occurred

### 1.2.4 Q-Learning test series: conclusion

The main problem of Q Learning (or tabular learning in general) in the 2048-game is the vast state space that needs quite some computational power, especially as the game progresses, to keep track of the tabular information. If agents were to reach higher tiles, the possible state space increases exponentially, and our hardware would probably reach its limit.

Nevertheless, we gained some insight about suitable alpha levels and possible scopes of epsilon schemes for tabular Q-Learning. Very low alphas tend to perform poorly, failing to beat a random agent after 5000 episodes. Epsilon schemes might even decrease at very low rates and still obtain good results, as the test results from learning off random agents suggest decent results from pure exploration as a start. Our specifications with alpha = 0.5 and 0.8 fared comparably well. As a comparison for DQN we will later on take the best performing agent (alpha = 0.8, monotonous epsilon scheme) as additional baseline for performance evaluation.

## 1.3 SARSA

We used SARSA, because our environment fits to the requirements of it. The reason is, that our environment has a limited space of states. Therefore, it is possible for the agent to know the state at the specific point and calculate the perfect action for the given state.

Our initial goal was to program an agent, where it was possible to attain the tiles up to $2^{15}$. However, due to missing disk memory and computing power of our machines it wasn't possible. Therefore, we lowered the highest attainable tile to $2^{8}$, which is equal to 256. We wanted to store the whole space of possible states in a text file, but the file, where we saved our data, was bigger than 40 GB before running any computation on it! As mentioned before, in the original version of 2048, it is possible to attain higher number than 2048. As a template we have used a tutorial from the youtuber Machine Learning with Phil (Machine Learning with Phil (2019b)). He has created a SARSA model for the game carpole-v0, which is also set in an open AI gym environment. We adjusted his code to our environment, which was in our view more complex than his environment. We have also added a get_max () function and some plots, so the analysis of the results can be visualized and more meaningful results presented. In the end, the code is virtually the same for SARSA and Q-Learning, apart from the expectations upon which the updates happens.

Compared to the version of Phil, we dynamically updated our Q-table. We started to generate the whole Q-table at the beginning too, but we never managed to fully create it. Therefore, we started to create a list of all possible states and a Q-table dynamically, by adding states as they were first observed. The SARSA agents use the choseandcheck function too, to check for loops. In order to make the results from SARSA and Q-Learning testing more comparable, we used the identical hyperparameters and the same size of batches, goals and episodes as in the prior test series.

### 1.3.1 Pre-test runs

As in the Q learning model, we first started with an alpha of 0.2, a gamma of 0.99 and an epsilon, from 1 to 0.0002. The goal was still to attain 256. Comparing with the Q-Learning model, it can be concluded that, there is a positive correlation between the batches and winning statistics too, like in the Q learning model. Therefore, the model was improving from game to game. It never reached values above 256, however the next step of our model testing would consist of increasing the goal to 512.

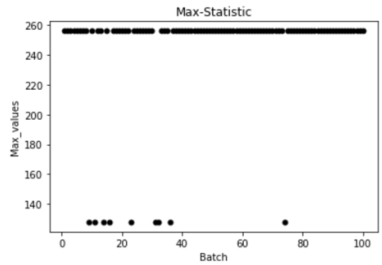



*Figure 3: Max-statistics*

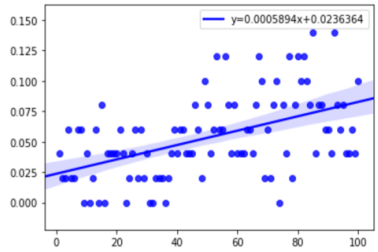
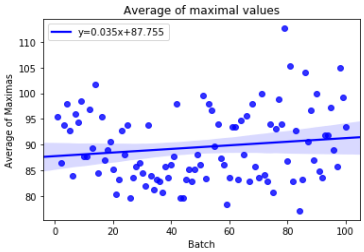
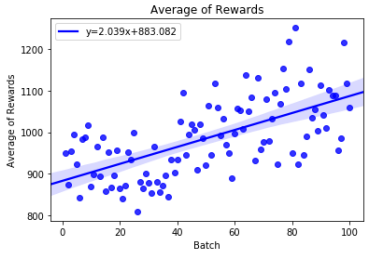



*Figure 4: Win statistics*

If we compare the win statistics model from Q-Learning and SARSA, we see that Q-Learning performed slightly better than the one from SARSA. Regarding the max value statistic, both models performed equally well.

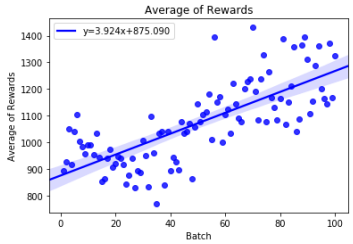
In our second test-run we also decreased the learning rate alpha to 0.1. Similarly, to the Q-learning model, we increased the goal to 512 too.

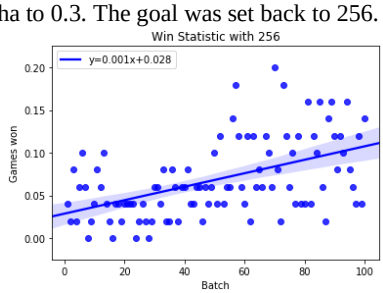



Like in the Q-learning model there is a positive correlation between the number of the batches and the average reward per batch. With these hyperparameters and graphs, we see that the SARSA model performs better than the Q-learning model. One indicator is the higher slope of the reward graph and the other indicator is the positive slope of the average of maximal values graph.

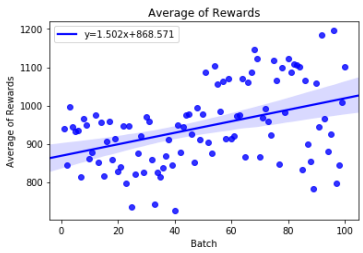
In the third test-run, we set the learning rate alpha to 0.3. The goal was set back to 256.

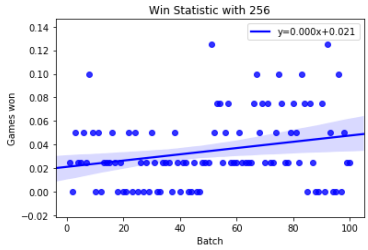


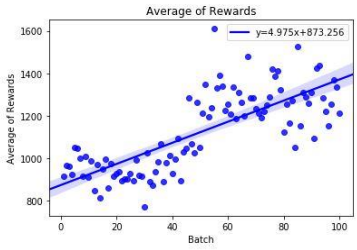

Looking at the figures, it becomes clear, that the reward and the win statistic have increased. The average in rewards have increased almost twice as much as in the second test-run. The Win-statistics doubled in comparison to the first test-run. In comparison to the Q-Learnings, SARSA slightly better in the Win Statistics and Average of Rewards.

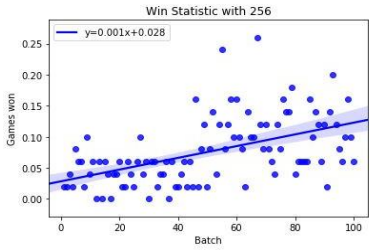
We lowered the alpha to 0.01 and did a test-run with 4000 episodes. Clearly the rewards were almost three times lower than in the third test-run. The same applies to the win statistics. Comparing with the Q-Model, the average of Rewards was slightly increased and the Win statistics slightly decreased in the SARSA model.



As lowering the alphas did not improve the performance of the model, we raised the alpha again to 0.5. We did a test-run with 5000 episodes. Again, the rewards were increasing quicker with the number of episodes than in fourth test-run. The same applies to the win statistics. The reward increased per batch on average by 5 units. In comparison to the Q-Learnings, SARSA identically concerning the average of rewards, however had superior win statistics.




### 1.3.2 SARSA test series: experimental design & results

Analogous to Q-Learning, our SARSA agents vary along two dimensions. There are two epsilon schemes; whereas one is a monotonously linearly decreasing epsilon, floored at 0.01, whereas the other scheme is a discontinuous epsilon development with multiple discontinuous resets of epsilon to a higher level. This should reduce the risk of ending up at a non-optimal solution by continuously varying exploration and exploitation. The other varying dimension are the alphas, chosen at 0.1, 0.5 and 0.8. After 5000 training episodes, there are performance tests with 500 episodes. Results are mostly expressed in percentiles, thus the average result over the percentiles (100 datapoints, in test mode 5 episodes are pooled to one datapoint) are presented[2].

Analogous to Q-Learning we also specified 2 random agents with alpha = 0.5 and 0.8 respectively to evaluate performance of the different parametrizations' effectiveness. Thus, there are 2x3 +2*2=10 specifications for testing and comparison.

SARSA summary table of parametrizations and main performance measures after 5000 training episodes[3]

| alpha | epsilon | Win percentage | Avg max tile | Avg reward |
|---|---|---|---|---|
| Alpha = 0.5 random agent | 1, training | 0.046 | 95.72 | 964.91 |
| | Test with eps=0 | 0.08 | 101.01 | 1184.7 |
| Alpha = 0.8 random agent | 1, training | 0.045 | 95.77 | 966.65 |
| | Test with eps=0 | error | 105.54 | 1234.28 |
| Alpha = 0.1 | Monotonous | 0.034 | 87.73 | 996.02 |
| Alpha = 0.1 | Nonmonotonous | 0.046 | 95.07 | 1030.42 |
| Alpha = 0.5 | Monotonous | 0.105 | 111.10 | 1299.20 |
| Alpha = 0.5 | Nonmonotonous | 0.151 | 115.94 | 1332.19 |
| Alpha = 0.8 | Monotonous | 0.076 | 110.00 | 1266.85 |
| Alpha = 0.8 | Nonmonotonous | 0.090 | 105.29 | 1233 |

[2] Averaging episodes performance or averaging percentile's average performance leads to the same mean results, which have been used in the tables.

[3] We initially forgot to specify those measures in the code and thus extrapolated the average of the test performance as intercept + 50*slope of each statistic in the performance test. Rounding errors may have occurred

### 1.3.3 SARSA test series: comparison

Considering the alpha, which is the learning rate, agents with alpha = 0.1, irrespective of the epsilon decay, failed to beat the random agent across all measures. Only for average reward, those models slightly outperformed the random strategy. The model with alpha 0.5 and 0.8 managed to beat the random agent across all measures and outperformed most of the SARSA models learning off the random agent. With alpha=0.8 the results were comparable to the learning off random agents. The agents with alpha = 0.5 outperformed with respect to all measures. So, we expect the globally optimal value of alpha for the SARSA model around that value. Given that alpha 0.8 does not perform much worse, the actual optimal value might be somewhere between, i.e. in the range (0.5, 0.8). Further testing is needed to specify the optimal range.

Comparing the results of the non-monotonous epsilon decay and the monotonous epsilon decay, non-monotonous epsilon decay outperforms monotonous epsilon decay, where learning state is increasing and, in a state, where the learning state is decreasing the monotonous epsilon decays the non-monotonous epsilon. It is difficult to make a conclusion about the alpha level 0.5. In our experiment it is the best performing, but if there was more time, there is a possibility that the global maximum is at an intermediate level the varying decay schemes effect is unclear. In our case, the non-monotonous scheme appears marginally superior Due to time restrictions we unfortunately couldn't test this hypothesis, but this would be our best guess.

Thus, we can conclude that the learning rate alpha has a much bigger influence on the result of the model than the decay scheme of epsilon.

### 1.3.4 Q-Learning test series: conclusion

Analogous to the Q-Learning models, tabular SARSA is challenging due to the vast state space. Nevertheless, we gained some insight about suitable alpha levels and possible scopes of epsilon schemes for SARSA. Very low alphas tend to perform poorly, failing to beat a random agent after 5000 episodes of learning. Epsilon schemes might even decrease at very low rates and still obtain good results, as the test performance from learning off random agents suggest decent results from pure exploration as a start. As a comparison for DQN we will later on take the best performing agent (alpha = 0.5, non-monotonous epsilon scheme) as additional baseline for performance evaluation.

## 1.4 Comparing Q-Learning and SARSA

Neglecting the specifications with alpha =0.1, which performed poorly with both methods, there is little evidence for actual superiority of one. The results from SARSA and Q-Learning are very similar. Asa a tendency, SARSA seems more challenged by high alphas in the range of 0.8 and might therefore have a lower optimal alpha level than Q-Learning. SARSA is a somewhat richer model that takes into consideration the momentaneous exploration rate when building its expectations of future reward (i.e., Q Value). This can also be seen by the slight outperformance of SARSA when learning off a random agent. Q-Learning seems less successful for the problem at hand. To our surprise, the best test performance resulted when learning with a=0.8 off the random agent. This result might be driven by our earlier observation that observing and learning from random strategies might be a good starting point for initial learning.

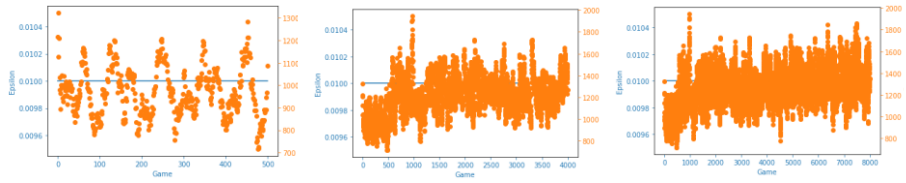
## 1.5 Deep Q-Learning

Deep Q-Learning Network (DQN) uses neural networks to approximate the Q-value. In contrast to Q-Learning, only the state is given as an input. Nonetheless, the Q-value of all possible actions is the outcome (Choudhary, A. (2019)). This is the reason, why we chose DQN as one of the methods for our project.

We programmed the DQN model with Keras with an Adam optimizer. Once again, we used the tutorial of the youtuber Machine Learning with Phil (2019a) as a template. He created a DQN model for the open gym ai environment called LunarLander v2. This environment was less complex than ours and can be solved in less than 500 episodes. The neural network, which Phil created suited quite well to our environment after some modifications. We had to transform the 4 x 4 matrix into a list to be able to use the deep q network of Phil. Later we realised that we forgot to apply a function like choseandcheck from the previous chapter. But during our test-runs the model never got stuck in a loop. We assumed that the neural network learned it by itself to handle this problem[4].

### 1.5.1 Pre-test runs

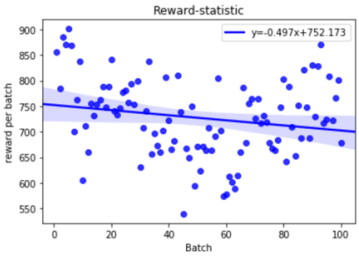
In our first test-run with 8000 episodes, our model achieved an impressive value of 512. The learning-rate alpha was 0.0005 and epsilon decay was 0.096. The epsilon started at 1 and decayed till 0.01. To plot this test-run, we used the graph from Phil. While this plot worked for his model, it gets less clear if our DQN really learnt during these 8'000 episodes. We can see a slow positive scope however there is too much variation in the data. Therefore, we are going to use our plots again for the next test and once again divide the episode to 100 batches. Therefore, the learning becomes more evident,

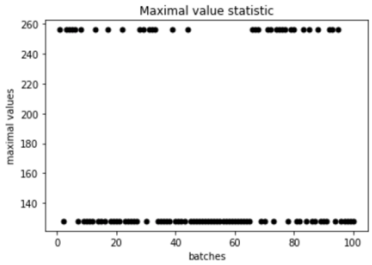


*ep. 500* *ep. 4000* *:ep. 800*

Remarkable is that the updated DQN version was much quicker than the SARSA model. Almost twice as quick. With a goal of 2048, an alpha of 0.025 and 5000 episodes the model didn't do that well either. There was a slightly negative correlation between the batch and the reward of the batches. The highest value the model hit was 256, however only once. Mostly 32 and 64 were hit. The performance was therefore quite poor.

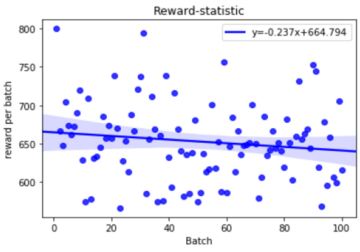
[4] Note: Contrary to our previous assumption, the DQN agent does not immediately get out of "unfeasible move loops" itself! The agents had to randomize multiple times. This can be seen by the "penalty" count at end of DQN notebooks, that count the number of moves without resulting changes to the state space.

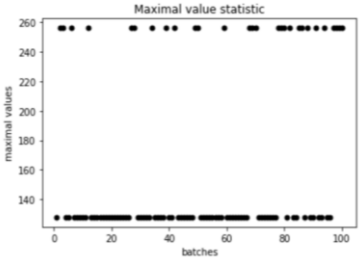


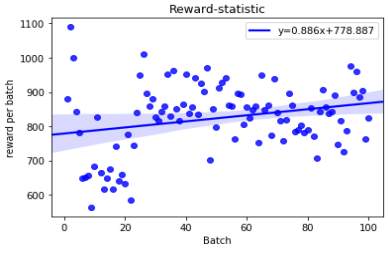

As the results were not satisfying, we still made another test-run with the same parameters. However, this time, the model performed slightly better. The model mostly achieved tiles of 128 and 256.Nonetheless, the regression line shows that the batch and reward per batch were still negatively correlated.

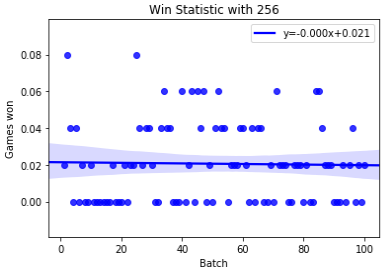
As we weren't satisfied with the results, we searched for projects, which implemented DQN for the game 2048. We found the paper of *Kaundinya, V., Jain, S., Saligram, S., Vanamala, C.K, Avinash, B. (2018)*, who did a similar analysis as we did. In this fourth test-run we used the parameter of this paper, to see if their parameter would enhance our model. Therefore, we defined alphas as 0.009 and gamma as 0.9. The number of episodes remained 5000. Similar to the second part of the third test-run, there is negative correlation and maximal values reached where mostly 128 and some 256. However, comparing the reward statistic, with the second test-run, it was only half as high.



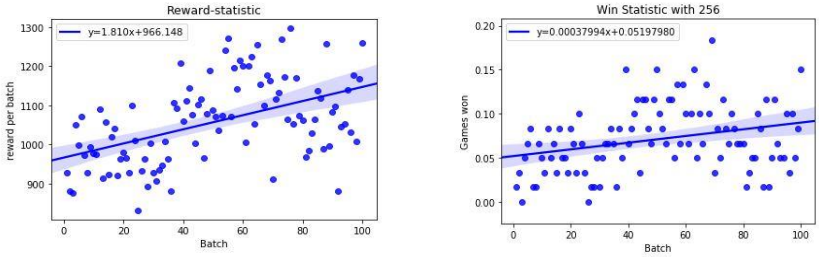




Comparing the last few test-runs ha the suspicion that a lower alpha resulted in a better performance. Therefore, again we defined an alpha of 0.005 for the fourth test-run but this time with our statistics visualization rather than Phil's. Again running 5000 episodes. The maximum goal was set at the tile 256. Again, the computations ran remarkably slow, but the reward statistics were much better than previous test-runs.



To confirm our assumption that a lower alpha improves the model, we did a fifth test-run with an alpha of 0.001, 6000 episodes and a maximum goal of 512 tiles. Our assumption was validated again. The reward statistic shows a steeper line, than in the fourth test-run as well as an increase of the intersect. Therefore, the positive correlation between the batch and reward per batch is higher. A possible explanation for the better performance with a lower alpha may be the very vast state space, such that opinions, or in this case neural net approximators, should only be adapted slowly.




### 1.5.2 DQN test series: experimental design & results

We vary our specifications along three dimensions: neural architecture, epsilon decay scheme, and alpha. The neural architecture is always two-layered, where the specifications are (256,256), (64,32), and (8,4). We therefore compare a big symmetric network to two smaller funnelled layers. Epsilon decay is always linearly decreasing and floored at 0.1, but two different rates are applied: 0.096 as suggested by Phil and our specification 1/5000 where the floor will only be hit after 4500 episodes, i.e. 90% of our training episodes. We therefore have 3*2*2=12 DQN specification for direct comparison and evaluation. Furthermore, we include the model with the best performance from Q Learning, SARSA, and random agent.

Given that we do not have access to a personal GPU, we did the calculations for our DQN on Google Colab. In the workspace folder shared below you will find 12 specifications of Jupyter notebooks, varying along three dimensions: epsilon decay, alpha and the dimensions of the neural network.

DQN summary table of parametrizations and main performance measures after 5000 training episodes

| Model | Training | Neural dims | Epsilon decay | Alpha | Win % | Avg max tile | Avg reward |
|---|---|---|---|---|---|---|---|
| Q, a=0.8, mono | 5000 | - | - | - | 0.126 | 109.64 | 1273.13 |
| SARSA, a= 0.5, nonmono | 5000 | - | - | - | 0.151 | 115.94 | 1332.19 |
| Best random performance* | 5000 | | | | 0.047 | 95.77 | 966.65 |
| DQN1 | 5000 | 256,256 | 0.096 | 0.001 | 0.078 | 103.14 | 1127.57 |
| DQN2 | 5000 | 256,256 | 1/5000 | 0.0001 | 0.563 | 204.25 | 2120.86 |
| DQN3 | 5000 | 256,256 | 0.096 | 0.0001 | 0.572 | 159.04 | 2218.79 |
| DQN4 | 5000 | 256,256 | 1/5000 | 0.001 | 0.064 | 96.47 | 1054.54 |
| DQN5 | 5000 | 64,32 | 1/5000 | 0.0001 | 0.326 | 156.41 | 1694.18 |
| DQN6 | 5000 | 64,32 | 0.096 | 0.0001 | error | 156.74 | 1703.26 |
| DQN7 | 5000 | 64,32 | 0.096 | 0.001 | 0.062 | 93.33 | 1039.31 |
| DQN8 | 5000 | 64,32 | 1/5000 | 0.001 | 0.096 | 102.66 | 1107.65 |
| DQN9 | 5000 | 8,4 | 0.096 | 0.0001 | 0.198 | 133.01 | 1479 |
| DQN10 | 5000 | 8,4 | 1/5000 | 0.0001 | 0.216 | 132.15 | 1499.11 |
| DQN11 | 5000 | 8,4 | 0.096 | 0.001 | 0.016 | 75.02 | 841.13 |
| DQN12 | 5000 | 8,4 | 1/5000 | 0.001 | 0.064 | 91.36 | 1041.59 |

*Best performance measures across all random agent training sets

Given our finding above that alpha= 0.001 tends to do better, we incorporated a further reduced alpha equal to 0.0001 which even tended to do better. After 5000 training episodes, the further reduced alpha seems to be superior in terms of our performance measures. Maybe this decrease by a factor of 10 could slow down the learning too much and result in underfitting, as presented in class. Nevertheless, the strong performance of this parametrizations does not support this idea yet. Maybe this could become more evident after more training.

### 1.5.3 DQN test series: comparison

The neural architecture seems quite important for the performance of the DQN: bigger networks with more nodes tend to perform better. The deviation to the rule is DQN8 that outperformed DQN4, ceteris paribus, with a sparser network. At the same time, bigger networks needed more time for the computations. The question, which one is most efficient remains open.
The further reduced alpha seems to be superior in terms of our performance measures, compared to the a=0.001 after pre-test runs. Maybe this decrease by a factor of 10 could slows down the learning too much and results in underfitting, as presented in class. Nevertheless, the strong performance of this parametrizations does not support this idea yet. Maybe this could become more evident after more training. Even with the sparsest architectures, agents performed quite well when alpha=0.0001 (see DQN9&10). All the alpha specifications, even with the smallest neural network, outperformed all of the previous tabular agents. As a caveat, those very small alphas increased the runtime considerably, such that we ran into backend problems with google colab. Multiple restarts where needed.
Regarding the epsilon decay, the effect seems to depend on the specification of the neural architecture. Whereas faster decay outperforms in case of the densest architectures (see DQN

1-4), the reduced decay is slightly superior for most of the rest (DQN 7-12). In the case of DQN's 5 & 6 there is hardly any difference.

Comparing the results to the best performing tabular agents', one can see the superiority of most DQN specifications to Q-Learning and SARSA. Only the worst few agents underperform compared to the best random agent result from previous chapters, with very sparse architecture (8,4) and relatively high alpha of 0.001.

#### 1.5.4 DQN test series: conclusion

For a game like 2048 with its very vast state space, neural approximation seems to fare much better in terms of performance compared to the tabular approaches. Nevertheless, quite some computational power is needed to implement a DQN. Performance differential to the random agent increased considerably, compared to the tabular models, and as such we conclude that the learning process is more pronounced. Having said that, with additional training and better performance, the tabular storage of information should also increase exponentially. This might eventually become a computational problem itself that might surpass the need of DQNs.

In conclusion, for 2048, we would suggest a rather large neural architecture, possibly denser than (256,256), with low alpha. The epsilon decay does not seem to have such a big influence, but above results rather suggests a faster epsilon decay when coupled with dense neural architecture. The problem with such a model will be the great need computational capacities needed.

### 1.6 Overall conclusion

DQN seems vastly superior in our case with very extensive state spaces, where the game dynamics are not purely deterministic and the average reward results are very promising since they doubled in the best performing specifications compared to the random agent performance. Prolonged training should continue to increase the performance, such that these models should eventually reach the tile of 2048, provided enough training.

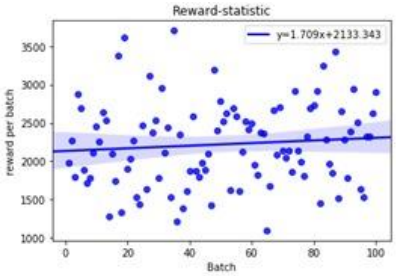
In the end, our performance is limited by our inability to store observations and learning progress. We tried to implement "a save" of the agents as to let them continue to learn rather than starting from the beginning. Coupled with the computational requirements of (deep) reinforcement learning our progress was quite slow. Most of the project time was actually needed for creating an executable code and as such, relatively little time was left in the end for "meta-troubleshooting".

## 2. Question 1

*Larry, Moe, and Curly are fighting as usual, but this time about reinforcement learning. Larry claims that whether exploring or exploiting, learning can always occur as long as the agent encounters something unexpected. Moe shouts, "Larry, you are wrong as usual! Learning can occur whether expectations are met or not! But only during exploration!" Curly yells, "No! You are both wrong! Learning can occur under any circumstances, but you have to expand the state space!" At this point, the police arrive, arrest them for disturbing the peace, and take them to the judge. Please tell the judge who (of Larry, Moe, and Curly) is right about what and why.*

A problem the agent faces in Reinforcement Learning is the trade-off between trial and error. The agent learns through trial and error, put in other words with the interaction with the environment. In order to maximize the rewards, the agent exploits actions tried in the past that were effective in attaining reward. However, to discover such actions, the agent needs to explore actions untried beforehand. In other words, on the one side the agent exploits his knowledge from past experience to maximize rewards but has to explore the environment to find even better strategies. Therefore, the agent can't learn only through exploration or only through exploitation without failing. During exploration, where he tries various actions, he needs to think forward and be able to favour the one which appear to be the best.

Thus, Moe's claim is wrong. One needs to find the most efficient balance between exploitation and exploration in order to achieve the best learning results (Sutton, R.S. and Barto, A.G. (2017)). We can see this, for example, in our DQN3 notebook, where the agent continued to improve performance during the test phase, where epsilon=0, i.e. without any exploration.




Curly's claim is wrong too. Expanding the state space can be even problematic. The larger the space, the longer the time needed understand the environment and learn. It is almost impossible find the "perfect strategy" with Q-Learning method, when the state space is large. Traying all state spaces will cost a lot of time (Hu, J. (2016)).

Therefore, Larry's claim was right. It fits the above-described explanation of Reinforcement Learning adequately. If expectations are met, Q are not updated, no matter what the alpha is. If expectations are not met, there will be an update. The only case where Larry's claim would be wrong was with an alpha of 0, but this is not a sound approach to RL, as any learning would be blocked, no matter what the agent encounters.

## 3. Question 2

*Consider a dynamic problem with states, actions, and rewards. After a random number of episodes (say 5,000 to 10,000), the parameters controlling the problem go through a regime shift. So, for example, in the box world this might mean the rewards would change, and/or the transition probabilities would change. There are a finite number of regimes that can occur, but which regimes occur is dependent on a probability distribution. Is it possible to come up with a modified Q-Learning algorithm that could develop a good policy for the dynamic problem? If yes, please describe such an algorithm. If no, describe why it is not possible.*

Q learning predicts the best action to maximize the cumulative rewards. Q learning is already an iteration, which evaluates his strategy step-by-step after each action. However its values iteration is limited. At least the states are known to compute the Q -table.

$$Q(s,a) = R(s) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q(s',a')$$

For Q-learning $R(s)$ and $T$ are not needed for the computations. The learning rate is updated based on the transition we have seen. Therefore, a modified Q-Learning isn't needed, to evaluate the change in rewards or transition probability.

SARSA might be a solution, as it builds the expectations while considering the impact of the applied exploration scheme on the expected rewards and transition probabilities.

## 4. Question 3

*Please explain in terms your grandmother/9-year-old nephew/CEO can understand the differences and similarities between supervised learning and reinforcement learning.*

Supervised learning and reinforcement learning are both paradigms of machine learning. There are three paradigms in total: supervised learning, unsupervised learning and reinforcement learning. In machine learning you use computer algorithms, which is a piece of logic written in computer code that the computer can compile(understand). The algorithms of machine learning are based on statistical or mathematical model, which maximize a certain value (prediction) or search for patterns, similarities or characteristics (classifier). All algorithms of machine learning have a high computing intensity and regularly need extended runtimes. Both methods can use neural networks approximators or not (then it becomes deep learning).

The difference between supervised learning and reinforcement learning is that supervised learning receives two data sets of two value types. The first type is a x-value and the second type is a y-value. Each x-value is clearly connected to a single y-value (or maybe a y-value-tuple). Then there are two types of sets: a training set and a test set. The data from the training set you use to train the model. By giving the model a big set of training data, it can easily figure out the relationship between x and y. Then you use the test data's x-values to challenge your model, if it learned appropriately. You always predict a y-value using a x-value from the test set. Then you compare the predicted y with the real y from your test set and calculate an accuracy. The accuracy should rate your model and describe how precise it recognizes the pattern resp. relationship between x and y.

Reinforcement learning on the other side doesn't need structured data sets. There, you have agents and environments interacting, where actions, rewards and states result out of this interaction. The agent, which is usually a model, tries to explore his environment by trial and error. An environment is usually a game, but it can be also something else. Then agents memorize states and actions and transitions resulting. For each action a certain reward is given by the environment. Depending on the environment, the reward could even be negative, but the agent always maximizes reward (alternatively it may minimize costs to reach certain reward levels. The dynamics, nevertheless, remain very similar).

So, the main difference is that the learning in the supervised learning is limited to the input provided by the data sets. The model can only learn as good as his training data are. In reinforcement learning a model can adapt to changes in the environment and continue to improve performance by interaction with the environment, without the need of external inputs like training sets.

## 6. Bibliography

Choudhary, A. (2019) *A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python.* Accessed on November 22, 2019, from https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-Q-Learning-python/.

Hu, J. (2016). *Reinforcement learning explained.* Accessed on November 22, 2019, https://www.oreilly.com/radar/reinforcement-learning-explained/.

Kaundinya, V., Jain, S., Saligram, S., Vanamala, C.K, Avinash, B. (2018) Game Playing Agent for 2048 using Deep Reinforcement Learning, *NCICCNDA*, 363-370. doi: https://doi.org/10.21467/proceedings.1.57

Machine Learning with Phil (2019a) *Deep Q Learning is Simple with Keras | Tutorial*. Accessed on November 22, 2019, from https://www.youtube.com/watch?v=5fHngyN8Qhw.

Machine Learning with Phil (2019b) *Reinforcement Learning in the OpenAI Gym (Tutorial) - SARSA.* Accessed on November 30, 2019,from https://www.youtube.com/watch?v=P9XezMuPfLE&list=PL-9x0_FO_lglnlYextpvu39E7vWzHhtNO&index=4.

Sutton, R.S. and Barto, A.G. (2017) Reinforcement Learning: an Introduction. Accessed on November 22, 2019, http://incompleteideas.net/book/bookdraft2017nov5.pdf.