Artificial Intelligence Project

Florian Kleinicke

February 23, 2018

1 Introduction

Artificial neural networks attain better and better performance. Nowadays it's possible for highly optimized neural networks to beat even the best Go player in the world. [7] But even less optimized networks are really impressive. This way it's possible to create an agent that plays a game, like in this case Snake [3], in a short one-man project for the Artificial Intelligence Lecture, by Prof. Björn Ommer [?].
For this project was an basic implantation of the Q-learning algorithm taken, applied on the OpenAI Gym/PLE Snake environment and improved to deal with the specifics of the problem.
The Q-learning approach shows some progress without being optimized for the problem. To further improve it, we tried different sized neural networks, multiple learning rates and an additional input with calculated features.

2 Q-learning

In traditional Q-learning, every possible state of the environment is stored in a table. There is a certain set of possible actions that lead to a just as many follow-up states. The algorithm wants to take the action, that leads to the best-rated follow-up state. When learning the ratings of the states, the ratings of the current state are changed based on the possible follow-up states and the rewards that are provided. In this way, a positive or a negative reward can propagate back a few states until a kind of stable solution is found. For a more in detail explanation with formulas, please revisit the lecture slides.
The approach with a table only works with a small number of states and actions. Here the number would approximately reach $(64 * 64)^4$ different states since there will be a field of size 64 times 64 with 4 possible values at each position. This would take way too long to be trained. Since every state has to be visited multiple times and different actions have to be tried, this approach cannot scale. For this reason, we use a neural network to replace the table. The network learns the essential elements of the input and tries to derive the best possible action in a more abstract manner, than a table. The

learning performs in batches. In this project, the batch size has been set to 32. Therefore a replay memory is required. This memory has a size much bigger than 32 as this allows for random selection of certain states, thereby learning the reward and follow up state after a certain action.
The benefit of this is that the learning isn't focused only on the last few steps since that training data is highly correlated. In this replay memory the original state, the follow-up state, the action and the reward are stored. When running the network, the last 4 states are provided to the neural network as an input. These should make it easier to estimate the movement that happened in the previous frames. The network reduces this information to 4 floating point numbers as an output. The position of the output with the highest value is taken as the action.
While still learning, random actions are proposed with a decreasing eventuality. Here we start with a probability of random action `EPSILON_START` of 1.0 and slowly approach an `EPSILON_END` of 0.01 after millions of time steps.

3 The environment

We employed the OpenAI Gym [5]. It provides a simple interface to apply AI to certain games. Environments like these make it much easier for programmers to try out AI with less effort for implementing the environment.
OpenAI Gym provides an interface to some few old and simple Atari games (Alien, Breakout,...), classic games and problems that are in common use in the AI community like CartPole and MountainCar. To increase their number, some games that were implemented in the competing environment PyGame Learning Environment (PLE) [2] were also made available in the OpenAI Gym interface. One of these games is Snake, our choice for this project. We decided to use the OpenAI Gym interface instead of PLE since the OpenAI Gym is a more modern approach put forth by a bigger organization. Knowledge in the OpenAI Gym will be more useful in the future than PLE. To run the game, the package `gym-ple` [1] has to be installed and included. After this the Snake environment can be created, like every other environment just by using its name `env = gym.make('Snake-v0')`. By playing around with the environment We learned a few details.

The game itself can be seen in figure 1 and consists of 64 by 64 pixels. At the beginning of the game, the snake (green) is in the center of the screen. The direction of its head is indicated by the red line two (or sometimes three) pixels in front of the body. The snake has three choices at each time step: forward, left or right. But the controls for these are only the global up, down, right and left. If the illegal move (turning back) is chosen, the snake will ignore this and continue straight. There is also another command, going straight, but we will ignore this. The goal is for the snake's head to coincide with one of the salmon-colored pixels. This marks the food and has a size of 5 by 5 pixels. The snake can move along the background which is black. It starts with a length of 3 and grows by 1 every time it gets food. When it hits the border of the game or itself the snake dies. Death returns a reward of -5, finding food a reward of 1.

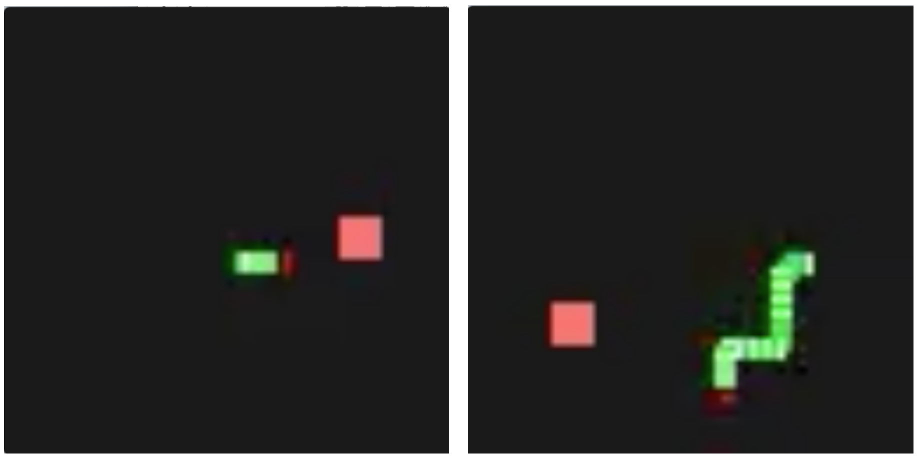


Figure 1: The snake at the beginning and in an advanced state the game

To simplify the input for the learning algorithm we reduced the number of color channels to 1. The second (green) color channel has unique values for each of the 4 elements:
direction indicator: 0, background: 25, food: 25 and snake body: 255.

There was also some weird stuff, we found out. Calling `env.reset()` at the beginning of the game returns a black image with a computer mouse on it. This might confuse the algorithm but were unable to avoid this. After the first step with the command `env.step(action)` the screen behaved as expected and can be seen in the previous figure.
After gaining an understanding of the environment we applied the Q-learning approach described in chapter 2.
Nevertheless, there was a problem with the implementation. The input of the network was the image as can be seen in figure 1. We used a neural network with two convolutional and two fully-connected layers. However, there was no visible learning progress. After days of searching for the problem, we decided to look for another implementation, that has all the basic functions pre-implemented.

4 Second approach

For the second try, we used a running implantation [4] of the Q-learning algorithm as a basis. The benefits of this implementation are that there already was a functional way to save and load the network. Also, the training was much faster because it didn't display the game while learning. There is an extra function, called `play` that displays the game played by a selected neural network.
We managed to modify the code to be useful for my purpose and run Snake with certain arguments.
The model we used consists of 3 convolutional layer and 2 fully connected layers. Each of the fully connected layers uses a ReLU as an activation function. ADAM is used as

the optimizer and `smooth_l1_loss` is used to calculate the loss function. In chapter 6 the model is modified to get improved results or a faster performance.

5 Improvements

The snake is able to learn just with the image as an input value. In general, this is exactly what is wanted. The neural network should be able to infer everything out of the image as an input. But how much could the results be improved when we help the network with additional information.

In total, we handed over 12 different numbers. 2 2D coordinates of the upper left corner and the bottom right corner of the food, the 2D position of the head of the snake, the direction the snake has to go, a direction the snake is able to go, depending on the current direction the snake is looking and finally the distances of the snakes head to the previous calculated food corners.
To calculate the position of the head, we searched for the pixel, that indicates the direction of the snake. After finding these we searched for the body of the snake in it's 2- and 3-pixel environment. The position of the central front pixel of the 3-pixel wide snake had been marked as the head position.

The find the direction the snake has to go, the distance between the head and the 4 corners of the food for each dimension is found. The guessed direction it the direction that is furthest off to any of the corners.
The further improve this result the current direction of the snake is taken into account. If the guessed direction is in the opposite direction to the snake it's recommended to turn around in two steps.

These additional parameters are passed next to the image as an input parameter for the neural network. In the forward module, these parameters are transformed to the right shape and are appended to the output of the last convolutional layer.
The additional parameters were also included in the training process and saved into the Replay Memory. Therefore we had to extend the transition states by the current additional state and the next additional state.

6 Models

At this point, we had two different options. We could spend some time with more advanced methods than the Q-learning approach, to compare the difference between the possible approaches. This would have been interesting to be seen and had been done by some other groups from this lecture.
Never the less we decided to stop here and to run a view experiments with the given algorithm. We tried several models and compared them to figure out what seems to be the important part of the model, what can be stripped away and what shouldn't be left out. We tried to make the network bigger to see faster performance gains (per training step) and smaller for faster training. The challenge is that every run is different, and there is no monotony in performance increases.

Name	conv	out channels	fc	in features	specialty
`DQN`	3	32, 64, 64	2	1024, 512	standard we started with
`DQN_smaller`	2	32, 64	2	2304, 512	
`DQN_verysmall`	2	32, 32	1	1152	
`DQN_tiny`	3	16, 16, 16	1	64	
`DQN_other`	2	16,32	2	1152, 256	We used this in the first try
`DQN_big`	3	64, 128, 128	3	8192, 2048, 512	didn't run well, used the next one
`DQN_big2`	3	8, 32, 32	2	288, 64	
`DQN_test1`	2	8, 32	2	6272, 256	0 padding, crashed fast
`DQN_test2`	2	8, 32	2	7200, 256	1 padding in first conv
`DQN_insp`	4	32, 32, 32, 32	1	512	didn't learn anything
`DQN_fully`	0		2	16384, 256	caused overflow

Table 1: The different models we tested

All these models can be found in the file `dqn_model.py`. The base model uses 3 convolutional layers with up to 64 different channels, strides between 4 and 1 and a kernel size between 8 and 3. Afterwards, the two fully connected layer reduces the number of features from 1024 to 4. As described before the output with the maximum value will be the guess from the network for the next move. All the used models can be found in table 1. In there the size of the convolutional and fully connected layer is described. But the kernel size and stride also varies with every convolutional layer and model. To see all the details, please look at the source code.

7 Experiments

We called the program with different models by using a command like:

```
python dqn_snake.py --add False --folder 7 --lr 0.0001 --model dqn
```

That can be modified slightly to run all the different models with and without additional features. The here used name of the model is the name that is used behind the `DQN_`. This mean the model from the table 1 `DQN_smaller` can be called by `--model smaller`. The results of the example above will be saved in the folder `dqn_checkpoints_7_False_dqn_0.0001`. We also tested different learning rates to figure out the right learning rate. All the experiments were run on the CIP Pool computer provided by the university and took several hours. After the training on the server, the trained models got downloaded and the resulting performance could be run by a command like:

```
python dqn_snake.py --add True --model dqn --mode play --record
 --checkpoint dqn_checkpoints_7_False_dqn_0.0001/chkpoint_dqn_8.pth.tar
```

This shows the result and to creates a video from it. The 8, in the end, stands for the total reward this saved neural network will receive in a certain test environment. The main problems for the snake were that in the end it usually killed itself without any reason, or it ended in a loop where it repeated the same action over and over again. The program automatically stops after 2500 steps. In all results the snake performs unusual behavior taking a way to long route to the food. Sometimes it prefers doing a very long loop before approaching nearby food.

We decided to plot the training progress to compare what methods learn how fast. After

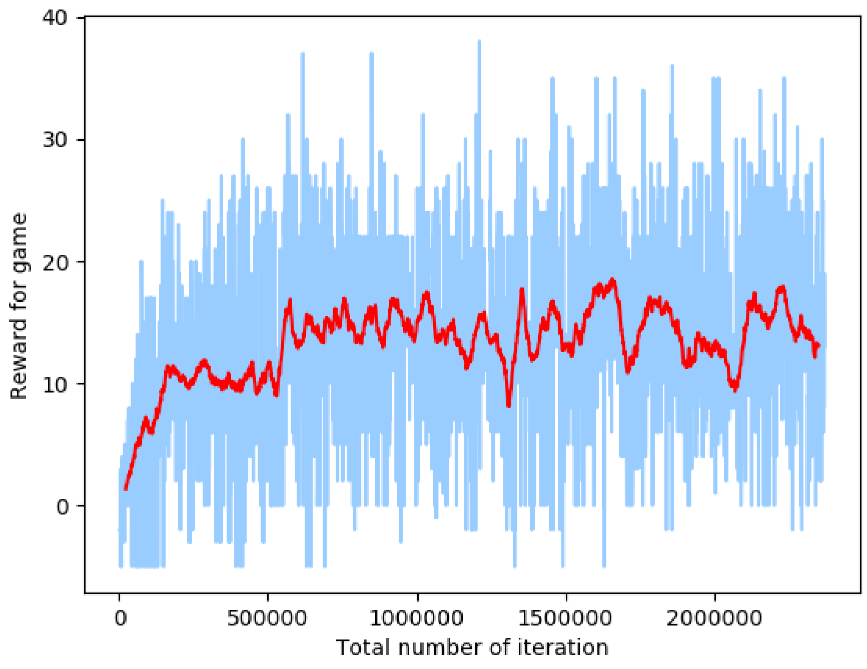


Figure 2: Results of a training with the `dqn` network. Redline is the average, blue line is the single results

900 steps of training, the algorithm takes a short break and tests the network. In the test, the order for the position of the food stays the same in every game. This means the tests are comparable. The problem is, that in reinforcement learning the learning progress isn't monotone and can have huge differences since there isn't an absolute right solution. In figure 2 you can see how the performance can vary between single test sequence. The blue line symbols the reward from every single test. The snake can perform very well, at in one test, but when there are a few changes made to the network by new training data the snake can already die before finding the first food. The performance of the snake is never about the best strategy to find the food since it doesn't get long enough. It's just about how far can the snake go, before accidentally killing itself. This error can occur right at the beginning, or at the end after finding the food more than 50 times.

The red line is the average over the last 50 play results. You can see that it improves at the beginning but stops rising after a short while. The average isn't the best indicator of training success, but in general, if the average is high, the best-reached scores are also higher.

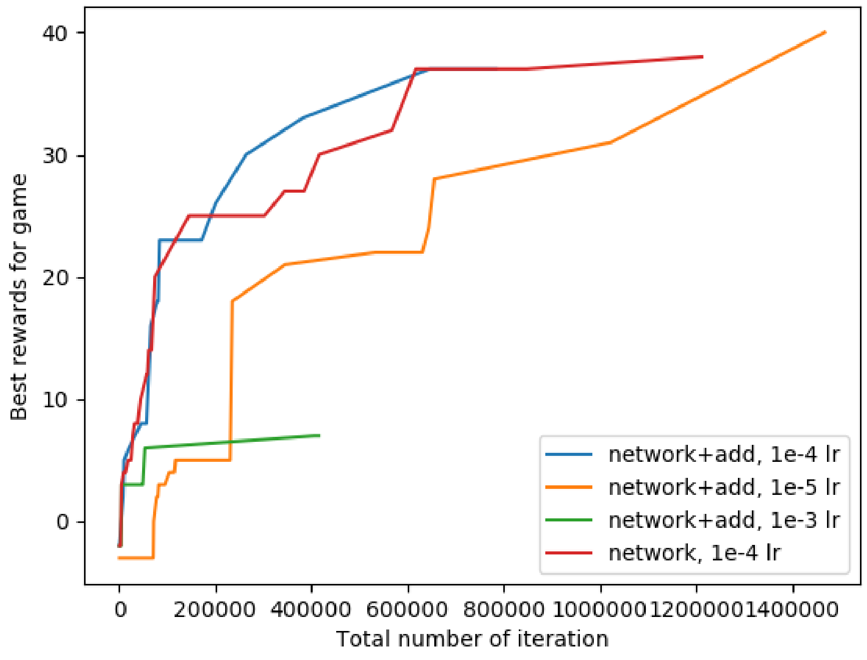


Figure 3: The best results for different learning rates

In the beginning, we tried to find the best learning rate. In figure 3 we plotted 3 different learning rates from `0.001` over `0.0001` to `0.00001` with the additional features described in chapter 5 activated. The rewards are shown in the plot where only updated for a new best-achieved reward. This is the essential measurement since this network is saved and can be used later on. You can clearly see one other problem we had. The best possible scores of the networks rise over time, and the longer they run the better is the score. Usually, the best possible score that can be reached anyhow is a good measurement for a network, but here we can't train long enough to reach its maximum.
Another problem is, that they perform slightly different after each run. So don't pay attention to minor differences in the plots since they might change when you run the network for the second time.
The three learning rates had different effects. If the learning rate is too high (`0.001`) the network can't make a good progress and stays at a very low level. The results for the learning rate `0.0001` were as expected. At first, the results improved relatively fast, but after some time the improvements became smaller and smaller. The smaller learning rate `0.00001` improves as slower, but reaches, in the end, the same level. It is hard to guess from that plot which of them would further increase in the future, so we continued that experiment after the poster session. As shown in the appendix in figure 6b the rewards continue to improve really slowly with no clear winner. This means that the learning rate `0.0001` performs better for short training periods and `0.00001` performs similar well

for longer training periods. We decided to use the learning rate `0.0001` for the future experiments since it reaches good results much faster.
In the same figure, there is a comparison plot for the learning rate `0.0001` without the use of additional information. As described before the results will look different with every run, so the minor differences between the two curves with the same learning rate mean that there seems to be no measurable advantage of the additional information. This is surprising since this should really help the snake to go in the right direction. Nevertheless using the additional features takes more time to train, it's not used anymore for most of the tests that follow.

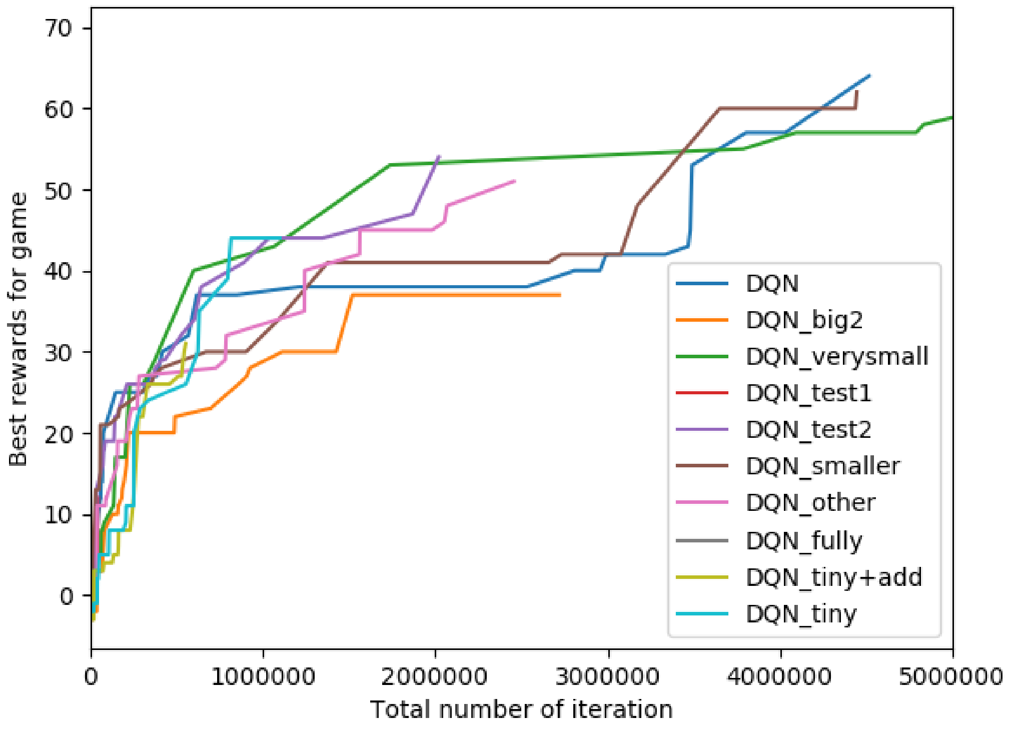


Figure 4: Best rewards of all the different tested models

the best test results of all the different in 1 proposed models are shown in figure 4. They all seem to perform really similar, with some exceptions. Except for the model all the other parameters were kept the same for each of the models.
`DQN_big2` doesn't seem to perform better than the smaller models. Therefore we experimented more with the smaller models. Generally speaking, the smaller the model, the faster the training. This means a smaller model could be trained with more iterations in the same period of time. We tried multiple changes to affect the training speed positively or negatively. In the end, the rewards the networks could archive with a given number of

iterations are all really similar. These smaller networks are `DQN_smaller`, `DQN_verysmall` and `DQN_tiny`. We reduced the number of convolutional and fully connected layer to a minimum, without seeing a performance decrease. But having more convolutional layer would lead to less fully connected nodes. So with `DQN_tiny` we reduced the number of channels in the fully connected layer to only 16. In the network we started with, the number of channels went up to 64 different channels.
The networks `DQN_test1` and `DQN_test2` were designed to compare the choice of the padding. The `DQN_test2` network performed similar to all the others, considering we used 1 padding for the first convolutional layer. In `DQN_test1` we used 0 padding, like in all the other models. But the network had some issues with overflows and crashed in multiple tries between 40000 and 60000 iterations. `DQN_fully` crashed after a similar amount of iterations. We thought that it might be possible to use only fully connected layer. A similar issue occurred to the `DQN_big` network. This one isn't even shown on the plot.
In the last comparison, we tried again to use the additional parameter, this time in combination with the `DQN_tiny` model. Here the additional features got a small additional fully connected layer before it got appended with the output of the convolutional layer. It had just as little success as before. There was no faster learning measure able.
We created a few more comparisons and put them in the appendix. In figure 6a the (200 tests) averages reward of every model are shown. In figure 5b the long-time version of figure 4 is shown. This can be compared to figure 5a the shows the same, compared to the required time instead of iterations. There the network `DQN_verysmall` has a small advantage, due to its size.
All plots were done with Matplotlib [6].

8 Summary and Outlook

We use a Q-learning approach to learn the game Snake from scratch. Therefore we create multiple different models to compare their performance. There is no improvement visible when exchanging models since the model seems to be good enough for the given Q-learning algorithm. 3 different learning rates are tried and we choose the learning rate that has the best result considering reward and training time.
The only input for these models is the game field provided by the game. To add more information to the network we calculated some features, that can be used by the network. These features don't improve the results of the tests.
The main bottleneck seems to be the Q-learning algorithm. Other approaches lead to much better results. Long short-term memory and actor-critic models can further improve the performance of the models. So the next steps would be to implement them and apply them to the modules. Therefore only the training module of the code has to be enhanced. The other parts of the code can stay in place.

References

[1] gym-ple. https://github.com/lusob/gym-ple.

[2] Pygame learning environment (ple). https://github.com/ntasfi/PyGame-Learning-Environment.

[3] Snake environment. https://gym.openai.com/envs/Snake-v0/.

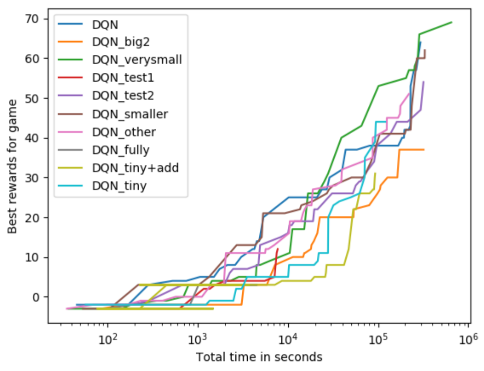
[4] AndersonJo. Deep q-learning with pytorch. https://github.com/AndersonJo/dqn-pytorch/blob/master/dqn.py.

[5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. https://github.com/openai/gym, 2016.

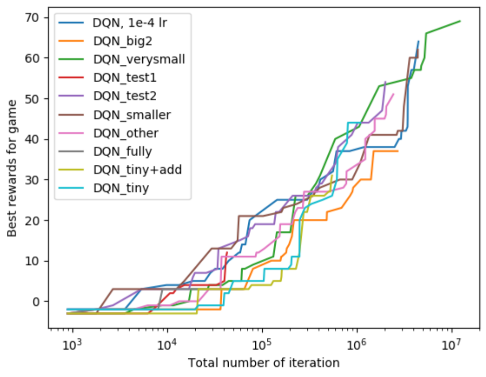
[6] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[7] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 10 2017.

Appendix

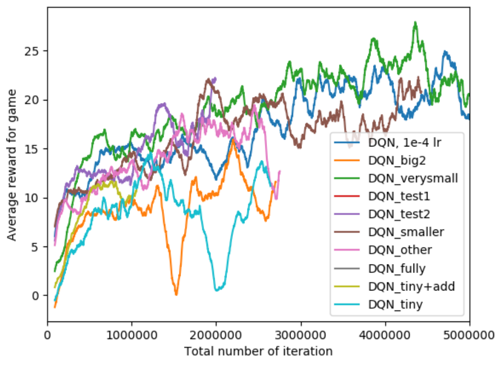


(a) The best models shown over an longer period than before, in seconds

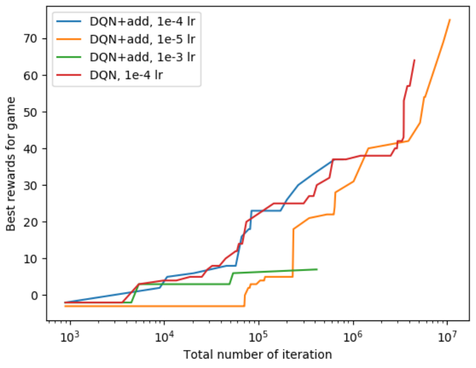


(b) The best models shown over an longer period than before, in iterations

Figure 5: Comparison time and iterations

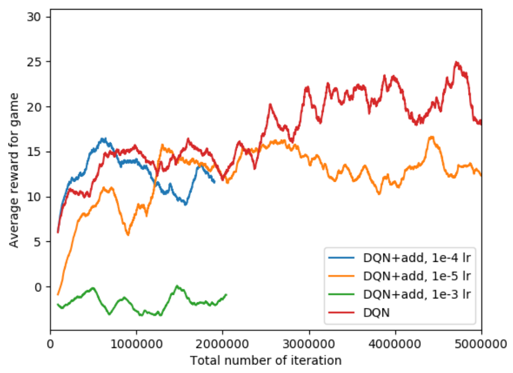


(a) Average reward for all the tested models

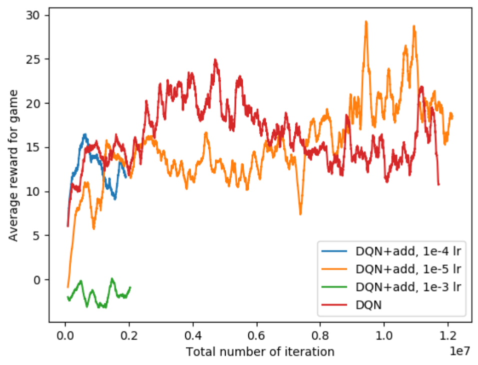


(b) The experiments from figure 3 run a little bit longer and plotted on a log scale

Figure 6



(a) The experiments from figure 3 with the average rewards shown



(b) The experiments from figure 3 run a little bit longer. Compare this to figure 6b. At 1e7 the orange average is high and a new record is set.

Figure 7: Comparison of the same plots. The left one has a smaller range.