Playing Flappy Bird with Reinforcement Learning

Eklavya Sarkar
Department of Computer Science
University of Bath
Bath, BA2 7AY, UK
es2030@bath.ac.uk

Matthew Clarke-Williams
Department of Computer Science
University of Bath
Bath, BA2 7AY, UK
macw21@bath.ac.uk

Abstract

In this project we attempt to develop models which are able to learn to play the game 'Flappy Bird', and ideally surpass human level scores by using Reinforcement Learning techniques. Specifically, we investigate two completely different approaches, tile coding and deep q-learning networks (DQNs), to develop an general overview of the problem and deeper understanding on reinforcement learning techniques. We go through their respective technicalities, and present conclusive results on experiments and hyper-parameter selection, in order to develop an optimal policy.

1 Introduction

Flappy Bird is a mobile game developed in 2013 in which the goal is to navigate a bird through gaps between pairs of randomly generated pipes for as long as possible. The bird can die only if it touches any part of a pipe, or if it falls down to the ground, which it is doing on its own continuously. The player can keep the bird alive by making it 'flap', thus increasing its vertical position for a short time before it starts falling again. The score is the number of pipes the bird has successfully gone through. This simple game was proven to be surprisingly challenging for humans, as it is hard to successfully master the precise position and timing of the bird before going through a pipe. In the sections below, we attempt to implement two different models which can learn to play this game optimally.

2 Tile Coding Approach

2.1 Environment

The first reinforcement learning method we tried to find the optimal solution to flappy bird was by using a linear gradient descent SARSA agent with the use of tile coding. We experimented with the variables that the agent used to learn but decided that the best variables to use would be the distance the bird was away from the bottom pipe in both the horizontal and vertical directions. Other variables that we implemented and then excluded from the final version were the bird's current velocity and also the bird's position above the ground. We decided the latter could be excluded since the vertical distance the bird was above the bottom pipe was essentially a duplicate of this variable (which would take negative values if the bird was below the bottom pipe). We decided that the velocity of the bird could also be ignored as a variable because the bird should still be able to learn only using its position in relation to the pipe. We did this because we wanted the agent to learn as quick as possible and although this may improve the reliability of the bird at harder levels (where the pipes are closer together), the small increments in the birds position should allow the agent enough time to decide whether to jump before hitting the pipe or not.

Project report for CM50270 Reinforcement Learning at the Department of Computer Science, University of Bath, UK. May 2019.

To begin tackling this problem, we first needed to create an environment in which the agent could learn and the physics of the bird could be calculated. We started by creating a class called `flappy_bird()` and made it initialise some variables which would be needed for the game to work. We used screenshots from the real game to work out the right measurements for the ground, pipes, and distances between the pipes. As well as this, we timed how long the bird took to move through the pipes and how long it took for the bird to complete multiple jumps so that we could work out important variables like the birds jumping velocity, the force of gravity, and the birds horizontal moving velocity.

Once these variables had been initialised, we added a method called `make_random_poles()` which when called, would generate a random pipe height from the ground and add the bottom pipe and top pipe to separate lists. We also made a method called `check_poles()` which checked to see whether or not to display the pipes in the graph when printed (the pipes that moved off the screen were not displayed). Next, we needed a method that would check to see if the the game had ended and the bird had reached a terminal state. This would be when the bird had hit into either of the pipes or into the ground. After this, we needed a way to reset the environment every time the game ended and also a method that would print the graph. These methods were called `reset_grid()` and `print_grid()` respectively. Finally, we needed a method called `make_action`, which given an action, updates the state of the environment, like the birds position, velocity and its positional relationship to the pipes.

2.2 Experimentation and Evaluation

Once the flappy bird class had been created, we implemented the linear gradient descent SARSA agent. We experimented with various tile shapes and other hyperparameters. The agent used an ε-greedy policy but we found that although the agent struggled to learn due to its random nature when it was between the pipes. We believe this was because the consequences of doing a random jump are very large while the consequences of randomly doing nothing is rather small. If, due to the random nature of the agent, the bird jumped when it was getting close to a pipe, the bird would take 20 moves of doing nothing to get back to the same spot and by this time, could have easily missed the gap in the pipes. For this reason, we then tried to *turn off* the ε-greedy policy when the agent got close to the pipes.

When we did this, we noticed a huge difference in the learning of the agent. Despite not performing anywhere near as well as the DQN implementation, tile-coding with a SARSA agent did demonstrate some learning and produced a learning curve where the bird made it through roughly 3 pipes before dying. What was strange to see was that *reducing* the number of tilings from 5 to 1 actually *increased* the performance of the agent. This is probably because this change would reduce the overall number of states and so states would be revisited more.

Furthermore, we found that increasing the number of tiles per tiling had a large benefit on the performance of the agent. Where the previous agents learning had almost converged after 100,000 episodes, using more tiles showed that learning was still taking place and that convergence wasn't reached until around 500,000 episodes (see Figure 1).

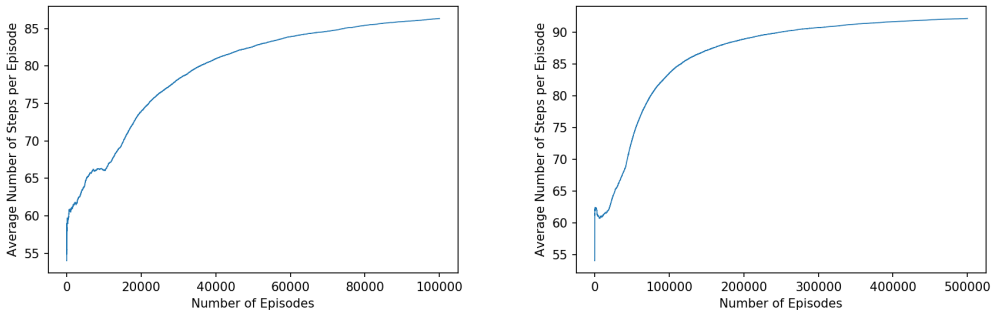

Figure 1: *Left*: Tile coding implementation using 5 tilings of size 16×10 showing convergence after around 100,000 episodes with an end average step size of roughly 86 steps er episode. *Right*: Implementation using 1 tiling of size 30×20 showing convergence after around 500,000 episodes with an end average step size of roughly 2 steps per episode.

For the other hyperparameters, we found that using a decaying epsilon worked best since the agent needed to explore early on but act greedily after enough states had been visited. After various experimentation, we found the best performing discount factor, γ to be 0.99. Having a discount factor of less than 1 ensures that the total reward does not diverge to infinity, however our agent was far away from experiencing this problem. Despite the advice given in Sutton and Barto [1998] where it was recommended to use a learning rate, α of $0.05 \times \frac{0.1}{m}$ where m is the number of tilings, we found this learning rate to be too high and opted for a slightly smaller one. The last hyperparameter we experimented with was λ for which we found optimal at 0.9 although due to time constraints, rigorous experimentation was not done so this may be improved. Again, due to time constraints, curves were only produced for one agent since we believe the improvements made through hyperparameter testing, outweighed the benefits of a smoother learning curve using more agents.

2.3 Personal Experience and Conclusion

The main difficulties we found when implementing the tile-coding SARSA algorithm were with hyperparameter optimisation. Finding values which showed any kind of learning was time consuming and as a consequence, we believe that our solution is still far from what an optimal tile-coding algorithm can achieve. Because the agent struggled a lot to learn, we had to reduce the difficulty of the game a lot by making the gaps in between the pipes $2\times$ larger and making the pipes less than half as wide. Even then, the agent struggled to make it past 3 pipes.

If we were to do it differently next time, we would probably use a pre-made flappy bird environment instead of coding our own. This would have allowed us more time for hyperparameter experimentation and may offer a better way to capture the video performance of the agent. The problem with ours is that it printed out a new graph of the game every step the bird moved and so it was not easy to capture any footage of the bird playing. However, the benefit of making our own flappy bird environment was that we could easily add any features we wanted since we were designing the code from scratch. Sometimes modifying someone else's code can take more time than just writing your own from scratch.

3 Deep Q-Learning Network Approach

The goal with this approach was to attempt to implement a method with a fundamentally different approach to tile coding function approximation. By employing Deep Q-Learning Network (DQN), we wished to showcase how computer vision and deep neural networks such as convolutional neural networks (CNNs) can be used in the context of reinforcement learning as well. Essentially, we implement a DQN which attempts to *approximate* the Q-function, rather than directly use the traditional Q-Learning method. We also wanted to employ 'experience replay' in our model, as summarised by Mnih et al. [2013], in their pseudocode (given on Figure 5). For this method, I chose to use PyGame's Flappy Bird environment implementation (PyGame), which I watered-down to simplify the training process.

3.1 MDP Representation

In terms of an Markov Decision Process (MDP), the game can be summarised in *actions*, *states*, and *rewards*. The actions here are simply flap (a = 1), or not flap, i.e. do nothing, (a = 0). The state representation in this case is the sequence of frames of the game itself, which depends on the current time frame t and its corresponding 'screen shot'. The discount factor can be formulated as $\gamma = 0.99$. Finally, defining the rewards of the game is perhaps the key to faster learning and convergence. As explained by Chen [2015], a model tends to learn much faster if there are more rewards in addition to the original one, incremented every time a bird passes through pipes ($\texttt{pipeReward} = +1$) . First of all, there needs to be a negative reward ($\texttt{deathReward} = -1$) every time the bird dies, in order to de-incentivise dying. However, this would still imply that a bird dying right before entering a pipe has the same amount of reward as one which dies instantly. Thus, an additional smaller reward ($\texttt{aliveReward} = +0.1$) can be used in order to further instigate the agent to stay alive as long as possible.

3.2 Network Architecture

The architecture of the DQN is heavily based on the previous work by Mnih et al. [2013] and Chen [2015]. The input to the first layer is essentially the screenshot image of the game, defined as 80x80. The image is converted to greyscale, and downsized to the desired dimensions using `cv2` module. The first convolutional layer contains 32 8x8 filters and a stride of 4. Similarly, the second and third layers contain 64 filters each of size 4x4 and 3x3 respectively, and a corresponding stride of 2 and 1. All three convolutional layers apply a rectified non-linearity (ReLU). Finally, we implement a fully connected output layer with a single output for each action, the best one selected as $a_{\text{best}} = \arg\max_{a'} Q(s, a')$.

3.3 Experiments

There is a large number of hyperparameters we could potentially modify and then evaluate, however this would be too elaborate for the scope of this project. Hence, we simply focused on implementing a working model of network described above on a *chosen difficulty*, which can be defined by the gap size between pipes. We narrowed down the difficulties into 3 broad categories defined as easy (`PipeGapSize` $= 200$), medium (`PipeGapSize` $= 150$) and hard (`PipeGapSize` $= 100$). We decided to train our model on medium difficulty until convergence, and then test the same model on the other two difficulties.

3.4 Training

We initialised our weights to random values of a normal distribution with a standard deviation of 0.01 and a mean of 0. I also chose to decay my epsilon, starting at $\epsilon_i = 0.01$ and ending at $\epsilon_f = 0.00001$. My reasoning was that an $\epsilon_i = 1$ of would make the bird flap at every time iteration, and it would thus always go to the top of the screen and eventually die by hitting a pipe. However, my model with $\epsilon_i = 0.01$ also had the same initial behaviour, but was nonetheless able to converge faster. I chose to perform the gradient descents during training with `AdamOptimizer` using a learning rate of 1×10^{-6}. We chose 50000 as the size of our memory replay, and 32 for the size of the mini-batches. We chose to fill in the experience replay with 10000 number of observations before starting the training. Once past this number, we trained on two batches of around 300000 iterations each, by using Tensorflow's `checkpoint` feature, which allowed us to save the model at regular intervals.

3.5 Evaluation

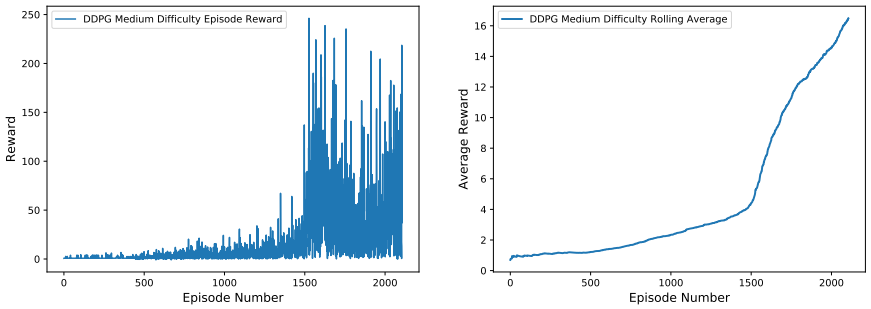

Figure 2: The evolution of rewards in function of episodes while training in medium difficulty. *Left*: Total rewards. *Right*: Rolling average of rewards. Note that both of these include the rewards during the observations phase, in which we store experiences to the replay memory, as well as the training phase, which on 300k time frames.

First, we trained our model on around 300k time frames on medium difficulty. As we see in our results, shown on Figure 2, the agent clearly continues to learn during training as the average rewards steadily increase in function of the number of episodes the agent plays. As this curve does not seem

to converge to a specific average return, we can perhaps conclude that the agent has successfully learnt to play the game at this difficulty, and the episodic rewards would continue to increase.

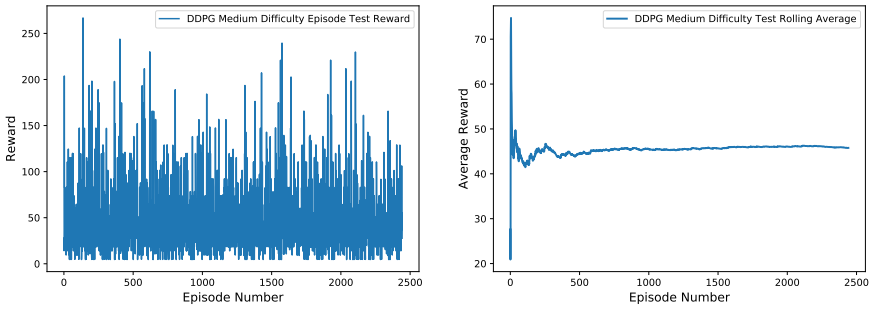

Figure 3: The evolution of rewards in function of episodes while testing in medium difficulty, having trained on 300k time frames. *Left*: Overall rewards. *Right*: Rolling average of rewards. Note that no training occurs in this part, as we simply run the model and choose the actions based on the trained model's Q-Network.

Figure 3 on the other hand shows us the *test* rewards of the agent per episode, which are similar to the highest rewards during the *training* phase. Unlike the training phase, we can clearly see a convergence in the rolling average of the rewards at around a value of $r = 45$, which is due to the fact that the agent does not learning during testing. This convergence is further highlighted in Figure 4, which shows the evolution of the overall 'best score' in the game over all the episodes. We were curious to see if the best score on this medium difficulty would increase if we trained our model for a longer amount of time. To this end, we trained our 300k model over another 420k time frames, using Tensorflow's 'checkpoints' feature, to arrive at a model trained over a total of 720k time frames. As shown on Figure 12, it reaches a high score of 1400, which is a significant increase over the 300k model's high score of 235.

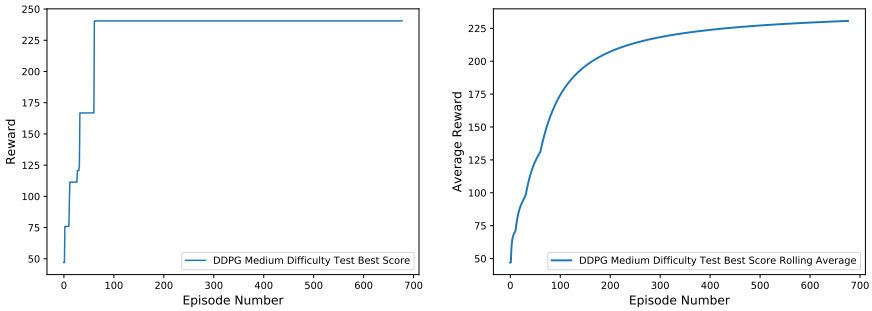

Figure 4: The evolution of the high score in function of episodes while testing in medium difficulty, having trained on 300k time frames. *Left*: Actual high scores. *Right*: Rolling average of high scores.

Finally, we attempted to have our strongest 720k version model play on easy and hard difficulties. The former proved to be of no challenge for our network, and it would keep on playing the first test episode till 'infinity' by virtue of simply not dying. However, having the model play on the hard difficulty while only being trained on the medium difficulty proved to be a bigger challenge than expected. In fact, as shown on Figure 14, the model obtained a poor high score of only 10.

Table 1: The high scores of our DQN model at different difficulties

Testing difficulty	DQN
Easy	Infinity
Medium	1400
Hard	10

As can be observed on Table 1, our DQN model is capable of matching and surpassing human level performance on easy and medium difficulties (a high score of 1400 can be categorised as infinity for this purpose). However, it is unable to beat humans, who are be able to have a better score than 10 in the hard mode. The logical next step would be to train our model over a similar number of iterations on the hard difficulty and then again compare the results. Based on our experiments, we can assume the model would be able to maintain, if not improve, its performance on the easier difficulties, and significantly increase its hard difficulty high score.

3.6 Personal Experience

From a personal point of view, we initially struggled with some technical problems relating to the environment, which was created by PyGame, and the latest Mac OS X. This was solved by installing the official OSX Python repository, rather than using Homebrew's one. We also had some difficulties conceptualising and understanding how the deep q-network worked, but fortunately were able to refer to Gue's excellent clarification and Emami's implementation examples. Overall, we were pleasantly surprised that the agent was indeed able to learn to effectively play the game with the DQN implementation, as we wasn't fully convinced of the efficiency of an algorithm which 'reads' the screen of an emulator and its those frames as inputs. As the game is relatively non-complex (only 2 actions), one can actually see the network trying to improve performance through the emulator, which was an enjoyable experience in real time.

3.7 Conclusion

We have shown that it is possible for this game to be solved through deep networks, and that its learning can indeed be optimised by using a `aliveReward` reward as it provides an incentive which is correlated to the main goal, as explained by Chen [2015]. We have also implemented experience replay which improves the process further. Additionally, we have seen that an agent can successfully play on easier levels if it trains on a harder one. In our case, to reach human level performance, we would have to train our model on a harder difficulty, and then compare the results.

4 Alternative Solution Methods

In this paper we actually take the time to investigate and present two completely different approaches, which cover a lot of ground in reinforcement learning techniques. Further methods could thus simply be more iterative, and build on these two, such as using Q-Learning on tile coding, or perhaps experimenting further with the architecture of the DQN to make it more learn faster.

References

Timothy P. Lillicrap, Jonathan J. Hunt, Alexand er Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv e-prints*, art. arXiv:1509.02971, Sep 2015.

Kevin Chen. Deep reinforcement learning for flappy bird. `http://cs229.stanford.edu/proj2015/362_report.pdf`, 2015.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL `http://arxiv.org/abs/1312.5602`.

Guest post (part i): Demystifying deep reinforcement learning - intel ai. `https://www.intel.ai/demystifying-deep-reinforcement-learning/#gs.9pzf53`. (Accessed on 05/05/2019).

Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 0262193981.

Patrick Emami. Deep deterministic policy gradients in tensorflow. `https://pemami4911.github.io/blog/2016/08/21/ddpg-rl.html`. (Accessed on 05/05/2019).

PyGame. Flappybird — pygame learning environment 0.1.dev1 documentation. `https://pygame-learning-environment.readthedocs.io/en/latest/user/games/flappybird.html`. (Accessed on 05/05/2019).

5 Appendix - DQN

5.1 Video

A short video of the DQN implementation can be seen at `https://youtu.be/db9bNbZNRjw`.

5.2 Hyperparameter choices

Table 2: Hyperparameter choices for training DQN at medium difficulty

Hyperparameter	Value
Gamma γ	0.99
Adam Learning Rate	1×10^{-6}
Number of observations	1×10^{4}
Number of explorations	3×10^{5}
Initial epsilon ϵ_i	0.01
Final epsilon ϵ_f	1e-05
Replay Memory Size	5×10^{4}
Batch size	32
Frames per action	1
Pipe gap size	150

5.3 Deep Q-learning Pseudocode

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 5: Deep Q-Learning with Experience Replay pseudocode as given by Mnih et al. [2013]

5.4 Train on medium difficulty with model trained on 300k time frames

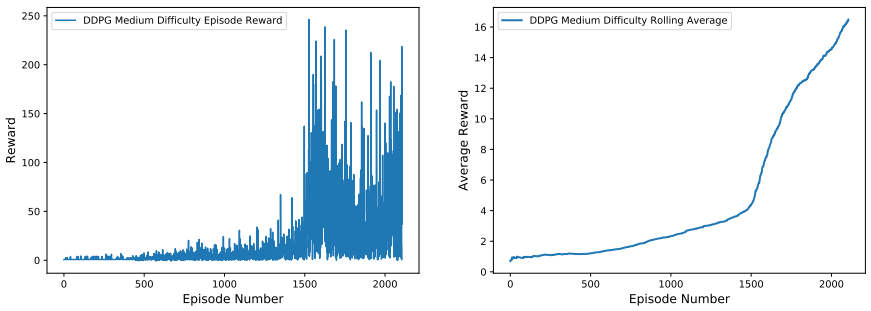

Figure 6: The evolution of rewards in function of episodes while training in medium difficulty, having trained on 300k time frames. *Left*: Overall rewards. *Right*: Rolling average of rewards.

5.5 Tests on medium difficulty with model trained on 300k time frames

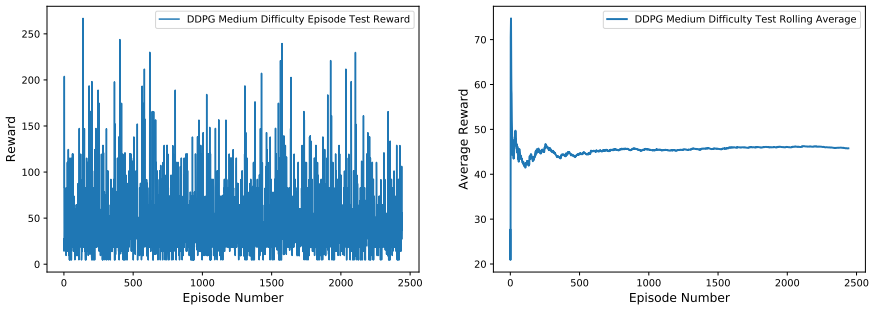

Figure 7: The evolution of rewards in function of episodes while testing in medium difficulty, having trained on 300k time frames. *Left*: Overall rewards. *Right*: Rolling average of rewards.

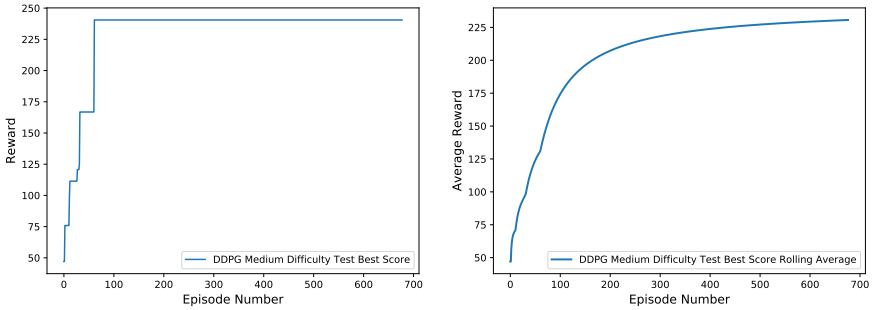

Figure 8: The evolution of the high score in function of episodes while testing in medium difficulty, having trained on 300k time frames. *Left*: Actual high scores. *Right*: Rolling average of high scores.

5.6 Train on medium difficulty with model trained on 720k time frames

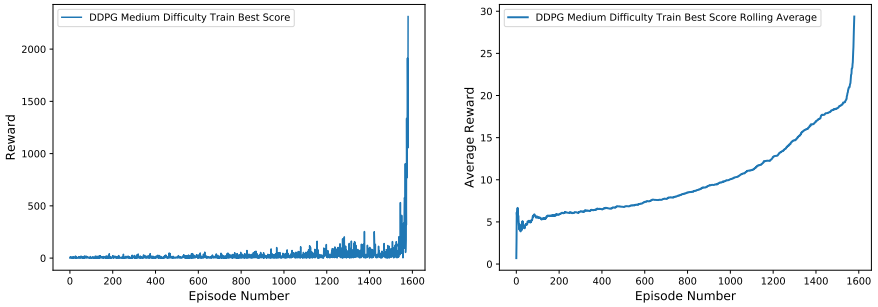

Figure 9: The evolution of rewards in function of episodes while training in medium difficulty on 720k time frames. *Left*: Overall rewards. *Right*: Rolling average of rewards.

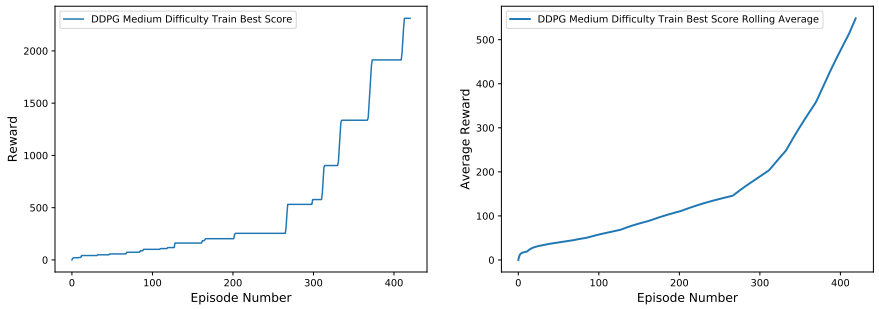

Figure 10: The evolution of the high score in function of episodes while training in medium difficulty on 720k time frames. *Left*: Actual high scores. *Right*: Rolling average of high scores.

5.7 Tests on medium difficulty with model trained on 720k time frames

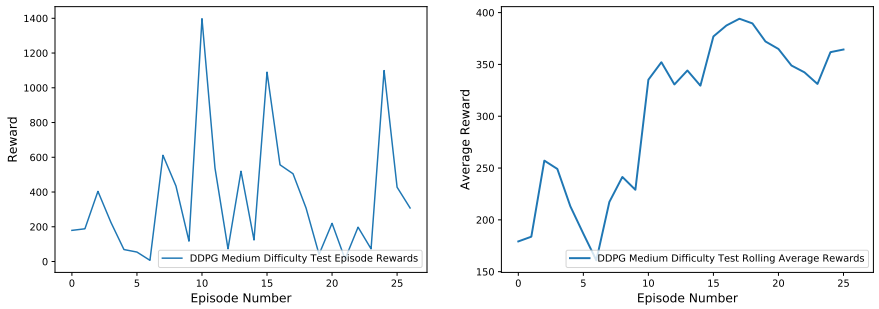

Figure 11: The evolution of rewards in function of episodes while testing in medium difficulty, having trained on 720k time frames. *Left*: Overall rewards. *Right*: Rolling average of rewards.

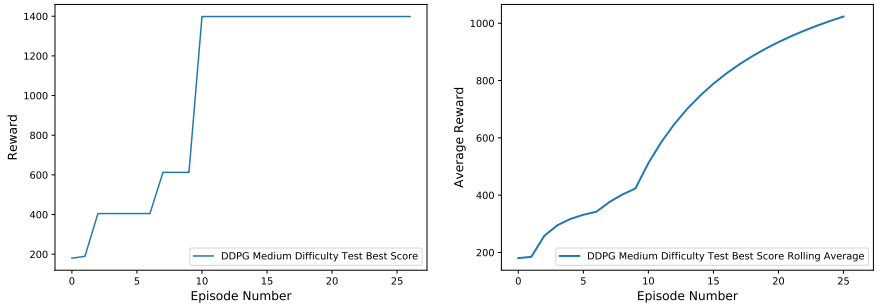

Figure 12: The evolution of the high score in function of episodes while testing in medium difficulty, having trained on 720k time frames. *Left*: Actual high scores. *Right*: Rolling average of high scores.

5.8 Tests hard difficulty with model trained on 720k time frames

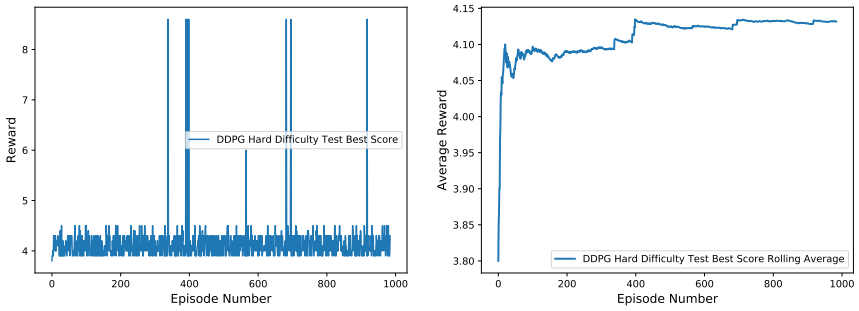

Figure 13: The evolution of rewards in function of episodes while testing in hard difficulty, having trained on 720k time frames. *Left*: Overall rewards. *Right*: Rolling average of rewards.

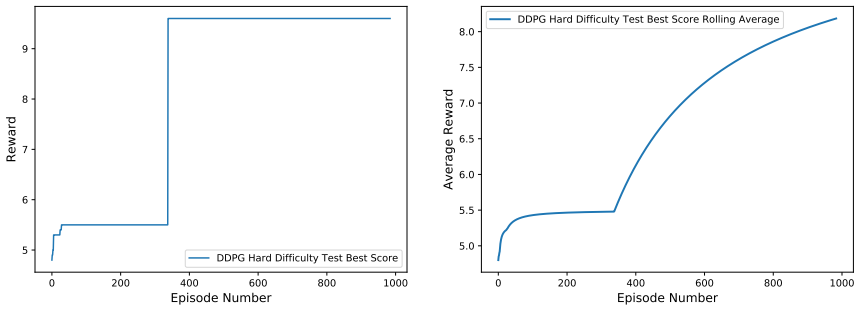

Figure 14: The evolution of the high score in function of episodes while testing in hard difficulty, having trained on 720k time frames. *Left*: Actual high scores. *Right*: Rolling average of high scores.

5.9 Results

Table 3: The high scores of our DQN model at different difficulties.

Testing difficulty	DQN
Easy	Infinity
Medium	1400
Hard	10