Deep Reinforcement Learning Through Improved Double DQN and Prioritized Experience Replay

Runzhe Wu
ACM Honors Class, Shanghai Jiao Tong University
runzhe@sjtu.edu.cn

Abstract

We implemented an agent receiving only the pixels of the Atari games that achieves a level comparable to that of a professional human player across 6 games. We use double DQN as the main algorithm and equip our agent with a prioritized experience replay. We also implement some other optimizations. Although some of the tried optimizations do not bring better performance, we still introduce them and illustrate their effects on learning to play the Atari game.

1 Introduction

Reinforcement learning is a field of machine learning that is mainly concerned about how agents should take actions in a given environment to maximize their reward. Deep learning is a popular machine learning methods based on artificial neural networks with representation learning.

The idea of combining deep learning and reinforcement learning had been attracting people's attention many years ago, but the rapid development of deep reinforcement learning was not started until DeepMind proposed the deep Q-network [Mnih *et al.*, 2013]. Since then, countless breakthroughs of deep reinforcement learning have been made and the follow-up works appear one after another. Two years later, some improvements were proposed [Mnih *et al.*, 2015] and more effective algorithms were introduced including double DQN [Van Hasselt *et al.*, 2016], dueling network [Wang *et al.*, 2015], rainbow DQN [Hessel *et al.*, 2018] and so on.

We tried some of these algorithms to see their effect and also attempted to enhance them in many ways. Some of our attempts have achieved very good results and we will do our utmost to explain the reason.

2 Preliminaries

2.1 Q-Learning

Q-learning is a reinforcement learning algorithm introduced in 1989 to let an agent learn a policy that tells it what action to take under what states. It does not require a model of the environment, so it's called "model-free".

The famous Q function (Q value) is an evaluation of the quality of a state-action combination:

$$Q : S \times A \to \mathbf{R}. \tag{1}$$

More precisely, based on the Bellman equation, $Q_\pi(s, a)$ is set as the expected sum of future rewards when taking action a in a state s under a given policy π. It is defined mathematically as

$$Q_\pi(s, a) \equiv \mathbf{E}\left[r_1 + \gamma r_2 + \cdots \mid s_0 = s, a_0 = a, \pi\right],$$

where $\gamma \in [0, 1]$ is the discount factor. The optimal value is $Q_*(s, a) = \max_\pi Q_\pi(s, a)$, which is what we always desire. Once we obtain the optimal Q value, we can easily derive the optimal policy by selecting the action with the highest value in each state.

2.2 DQN

However, most environments have a large number of states so that we can hardly learn all action values among all states separately. Instead, we make a little modification to make the Q value a parameterized one $Q(s, a; \theta_t)$ where θ_t is the parameters at time t. Then by standard Q-learning algorithm, after receiving state s_t, taking action a_t and observing reward r_{t+1} and resulting state s_{t+1}, we update the parameters as follows:

$$\theta_{t+1} \equiv \theta_t + \alpha(y_t^{\mathrm{Q}} - Q(s_t, a_t; \theta_t))\nabla_{\theta_t} Q(s_t, a_t; \theta_t).$$

where α can be considered as the learning rate and the target y_t^{Q} is defined as

$$y_t^{\mathrm{Q}} \equiv r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_{t+1}).$$

Intuitively, it updates the current value $Q(s_t, a_t; \theta_t)$ towards the target value y_t^Q by stochastic gradient decent.

As for our agent, we utilize a deep Q network which is a multi-layered neural network that for a given state outputs a vector of action values $Q(s, \cdot; \theta)$ where θ is the parameters of our network. As suggested by DeepMind's research [Mnih *et al.*, 2015], we use an extra target network whose parameters are almost the same as the main network except that it's copied every τ steps from the main network. Let θ_t^- denote the parameters of the target network, then the target is defined as

$$y_t^{\mathrm{DQN}} \equiv r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_{t+1}^-).$$

To break the correlation between consecutive samples, an experience replay is required for DQN, which stores some transitions and samples uniformly from them to update the network. The above is the whole picture of DQN.

2.3 Double DQN

The goal of double Q-learning is to reduce overestimations by decomposing the max operation in the target into action selection and action evaluation as follows

$$y_t^{\text{DoubleQ}} \equiv r_{t+1} + \gamma Q(s_{t+1}, \arg\max_a Q(s_{t+1}, a; \theta_t); \theta_t').$$

Although not fully decoupled, the target network in the DQN architecture provides a natural candidate for the second value function, without having to introduce additional networks. We, therefore, propose to evaluate the greedy policy according to the online network but using the target network to estimate its value. Thus the target in double DQN [Van Hasselt *et al.*, 2016] appears, which can be written as

$$y_t^{\text{DoubleDQN}} \equiv r_{t+1} + \gamma Q(s_{t+1}, \arg\max_a Q(s_{t+1}, a; \theta_t); \theta_t^-).$$

2.4 Prioritized Experience Replay

The central idea of prioritized replay [Schaul *et al.*, 2015] is to assign each transition its importance so that the transition with higher importance will be sampled more often.

It's straightforward to use the magnitude of a transition's TD-error which indicates how far the value is from its estimate. However, TD-error prioritization is not so ideal and some issues have to be handled. First, TD-errors are only recomputed for the transitions that are replayed to avoid expensive computation. Thus, a transition with a low TD-error will rarely be selected although at some points its real TD-error becomes large. For the same reason, it's likely to focus on only a small subset of experience since the errors decrease slowly, which makes the agent over-fitting. Furthermore, it is very sensitive to noise spike.

To overcome these issues, a stochastic sampling method is introduced. It serves as a trade-off between pure greedy prioritization based on TD-error and uniform random sampling. Generally, we ensure that the probability of being sampled is monotonic in a transition's priority, while guaranteeing a non-zero probability even for the lowest-priority transition. Concretely, we define the probability of sampling transition t as

$$P(t) = \frac{p_t^\alpha}{\sum_k p_k^\alpha} \tag{2}$$

where p_t is the priority of transition t and α is a factor. We set $p_t = \delta_t + \epsilon$ where δ_t is the TD-error of transition t and ϵ is a small positive constant.

Another issue of prioritized replay is that it introduces bias because it changes this distribution in an uncontrolled fashion, and therefore changes the solution that the estimates will converge to. We can eliminate this bias by using importance-sampling weights given as

$$w_t = (\frac{1}{N} \cdot \frac{1}{P(t)})^\beta$$

which completely compensates for the bias when $\beta = 1$ if we update the value $w_t \sigma_t$ instead of δ_t. For the sake of stability, we normalize each weight by $1/\max_t w_t$.

2.5 Architecture

The essential architecture of our agent is mainly based on some researches of Google DeepMind [Mnih *et al.*, 2013; Mnih *et al.*, 2015].

The input to the neural network consists of a $4 \times 84 \times 84$ image obtained by concatenating four consecutive 84×84 frames. The first layer convolves 32 filters of 8×8 with stride 4 and applies a rectifier nonlinearity. The second layer convolves 64 filters of 4×4 with stride 2 followed by a rectifier nonlinearity as well. The third layer convolves 64 filters of 3×3 with stride 1 followed by a rectifier. Then, a fully-connected layer with 512 rectifier units follows. Finally, a fully-connected linear layer is used as the output layer.

The whole neural network is equipped with a prioritized experience replay memory[Schaul *et al.*, 2015]. Some optimizations will be mentioned later.

2.6 Hyperparameters

The hyperparameters used in training are listed in Table 2 in the appendix. Some adjustments of the given hyperparameters may give rise to better performance. However, due to the limited training time and computing resources, some better hyperparameters like a big enough replay memory is not realistic as we are not in a lab.

3 Improvements

Now, we will introduce our optimizations on agents. Some optimizations work well while some optimizations may not be effective.

3.1 Huber Loss

As DeepMind [Mnih *et al.*, 2015] suggests to clip the squared error to be between -1 and 1 using an absolute value loss function for errors outside of the $(-1, 1)$ interval, we utilize the Huber loss function, which improves the stability of this algorithm. The Huber loss is defined by

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \le \delta, \\ \delta(|a| - \frac{1}{2}\delta), & \text{otherwise.} \end{cases}$$

where we set $\delta = 1$ as desired.

3.2 Improved Annealing Function

The original annealing function given by DeepMind's research is

$$\epsilon(x) = \begin{cases} -0.9T^{-1} + 1 & \text{for } 0 \le x \le T, \\ 0.1 & \text{for } x > T. \end{cases}$$

which is a piecewise function with two pieces. However, we make it more elaborate like this:

$$\epsilon'(x) = \begin{cases} -0.9T_1^{-1} + 1 & \text{for } 0 \le x \le T_1, \\ -0.09(T_2 - T_1)^{-1}(x + T_1) + 0.1 & \text{for } T_1 < x \le T_2, \\ 0.01 & \text{for } x > T_2. \end{cases}$$

Figure 1 shows the original annealing function ϵ and the modified one ϵ'. They have no difference at the beginning but the original one stops once it reaches 0.1 while our modified one

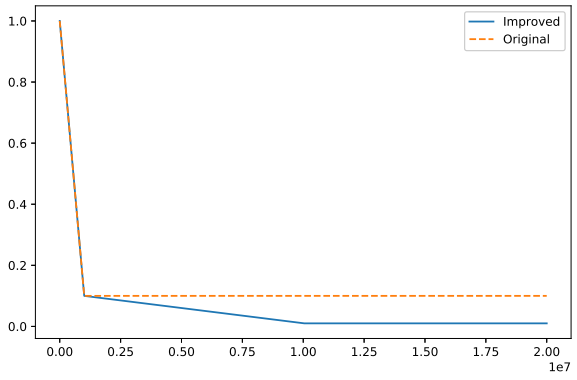

Figure 1: Comparison of Two Annealing Functions

slowly decreases to 0.01 when it reaches 0.1 and finally stabilizes at 0.01.

Our goal of modifying the original annealing function is to make the learning process more stable as some games like *Atlantis* and *Beam Rider* are so sensitive to network parameters that even a small change on our network will result in a huge performance difference. Thus, to void missing some excellent network parameters, we reduce the learning rate to 0.01.

3.3 Optimality Tightening

Optimality tightening [He *et al.*, 2016] is an interesting approach to accelerate reward propagation, and thus, dramatically reduce the training time.

From the Bellman equation we know that the following series of equalities hold for the optimal Q function Q_*:

$$\begin{aligned} &Q_*(s_j, a_j) = r_j + \gamma \max_a Q_*(s_{j+1}, a) \\ &= r_j + \gamma \max_a \left[r_{j+1} + \gamma \max_{a'} \left[r_{j+2} + \gamma \max_{a''} Q_*(s_{j+1}, a'') \right] \right]. \end{aligned}$$

However, we cannot expect to use it to update the network since the enumeration of intermediate states s_{j+i} increases exponentially with i. But life is not so bad if we notice that the following inequalities also hold for optimal Q function:

$$\begin{aligned} &Q_*(s_j, a_j) = r_j + \gamma \max_a Q_*(s_{j+1}, a) \\ &\geq \cdots \geq \sum_{i=0}^{k} \gamma^i r_{j+i} + \gamma^{k+1} \max_a Q_*(s_{j+k+1}, a) =: L^*_{j,k}. \end{aligned}$$

Then $L^*_{j,k}$ can be considered as a lower bound of $Q(s_j, a_j)$. Analogously, we can get a upper bound by the following inequality:

$$\begin{aligned} &Q_*(s_j, a_j) \leq \\ &\gamma^{-k-1} Q_*(s_{j-k-1}, a_{j-k-1}) - \sum_{i=1}^{k} \gamma^{i-k-1} r_{j-k-1+i} =: U^*_{j,k} \end{aligned}$$

Traditionally, we update the Q function with the parameter θ_j by optimizing the following program

$$\min_\theta \sum_{j \in \mathcal{B}} (Q(s_j, a_j; \theta_j) - y_j)^2.$$

where $\mathcal{B}$ represents the replay memory. But if we take the lower bound and the upper bound into account, we can reformulate the program into

$$\begin{aligned} \min_\theta \sum_{j \in \mathcal{B}} &[(Q(s_j, a_j; \theta_j) - y_j)^2 + \lambda (L_j^{\max} - Q_\theta(s_j, a_j))_+^2 \\ &+ \lambda (Q_\theta(s_j, a_j) - U_j^{\min})_+^2]. \end{aligned} \tag{3}$$

where $(x)_+ := \max(0, x)$, λ is a penalty parameter and

$$\begin{aligned} L_j^{\max} &:= \max_{1 \leq k \leq K} L_{j,k}, \\ U_j^{\min} &:= \max_{1 \leq k \leq K} U_{j,k}. \end{aligned}$$

If we use (3) as the loss function, according to the previous discussion, we should be able to train faster with the help of the lower bound and the upper bound. But in fact, this optimization does not affect well and significantly adds to the training time (because we need to compute those two bounds). Maybe that is just because we did something wrong. However, it is removed from our final version. We will show its effect in the *Result* section.

3.4 Frame Skipping

As some previous researches pointed out, a simple frame-skipping technique is efficient for learning and playing games. Our agent sees and selects actions on every kth frame instead of every frame, and its last action is repeated on those skipped frames. Because it requires much less computation than having the agent select an action for every frame, this technique allows the agent to learn and play approximately k times faster. Just as suggested by some previous work, we set $k = 4$ for all 6 games.

4 Methods

We utilize the standard atari-py environment provided by OpenAI [Palanisamy, 2018] so that we can only focus on the design of the agent regardless of some interfaces. Our model is implemented in TensorFlow 2.1.

Experiments were performed on 6 Atari games that consists of *Krull*, *Freeway*, *Pong*, *Tutankham*, *Atlantis* and *Beam Rider*. A different network was trained on each game. The same network architecture, learning algorithm and hyperparameters(see Table 2) were used across all games, showing that our approach is robust enough to work on a variety of games.

5 Result

5.1 Improved Annealing Function

The details of the improved annealing function have been explained in the previous section. Now we should see whether it

can give rise to better performance. Since there is not any difference between the original annealing function and the improved one in the first piece of the function, an experiment on a game with a relatively long training time is needed. Finally, we selected the game *Atlantis* since it takes the longest training time.

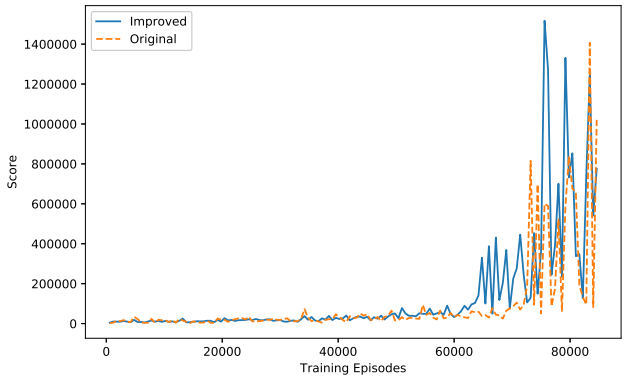

Figure 2: Comparison of Two Functions on *Atlantis*

Figure 2 shows the result when applying these two different annealing functions in the training of *Atlantis*. The training took 84600 episodes and the evaluation was performed every 600 episodes. We evaluated the agent on 20 episodes and took the average as the score.

The improved one reached the maximum score of 1517080 in 75600 episodes of training, and the learning speed of the improved one is also slightly higher than the original one. It seems that the improved function is more efficient, but we admit that randomness may play an important role in it.

5.2 Optimality Tightening

As discussed earlier, optimality tightening is expected to accelerate the learning. To see whether it works, we experimented on the Atari game *Pong*.

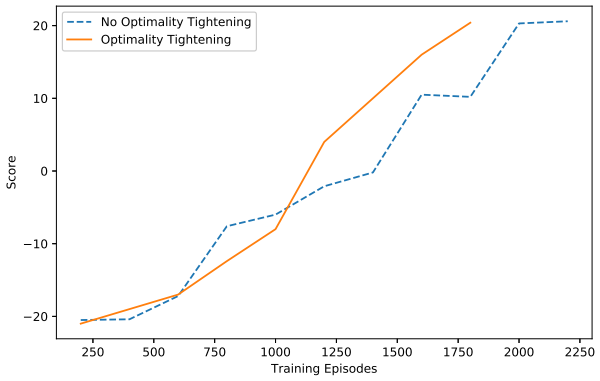

Figure 3: Comparison of Whether to Use Optimality Tightening on *Pong*

As shown in Figure 3, the two training curves, one of which is applied optimality tightening and the other is not, reach the score $20 \sim 21$ in $1750 \sim 2000$ episodes of training. Precisely, the one with optimality tightening reaches the score after 1800 episodes of learning while the one without it gets the same score after 2000 episodes.

Hence, it seems that optimality tightening brings a 10% reduction in the number of training frames. However, the fact is worse than that. As it reduces the training frames, it dramatically increases the computing time for each frame. That means in term of total training time, it is actually worse. So eventually we removed it from the final version of our agent as it is worthless.

We guess maybe there is some better implementations so that the trade-off between the reduction of training frames and the increment of computing time can be better, or its failure on *Pong* is just because the model *Pong* does not fit in with this optimization. We believe it worth further considering.

5.3 Overall Result

For each game, we selected the agent with the highest score in the evaluation during training. Their final evaluation scores are listed in Table 1. The evaluation was performed on 30 trials for each game. As we can see, it plays much better than human[1].

Game	Average	Maximum	Human
Atlantis	2303165.0	2734600	29028
Beam Rider	13557.4	35380	5775
Freeway	31.6	32	29.6
Krull	8830.4	9684	2395
Pong	20.0	20	9.3
Tutankham	262.7	351	167.6

Table 1: Evaluation Results

The learning curves of all 6 games are shown in the appendix.

6 Conclusion

This report made a summary about an agent we implemented using double DQN and prioritized experience replay, and demonstrated its ability to play six Atari computer games using only raw pixels as input.

We also introduced some optimizations including the improved annealing function and the optimality tightening, and tested their performances. We found that not all of them can give rise to a better performance in the experiments. Two main reasons are believed to be considered. First, it is difficult to rule out the impact of randomness during learning. Therefore, more experiments are favorable so that we can eliminate the effect of randomness. Second, the difference between models is also a vital factor. Maybe some optimizations work on one model but reduce its performance on another. However, we believe the optimality tightening tech-

[1]The human scores are from DeepMind's early research [Mnih *et al.*, 2015].

nique is creative and should be useful, so some future works can be done based on it.

7 Acknowledgments

I would like to thank TA Weizhe Chen for his good guidance and Asst. Prof. Weinan Zhang for his helpful reinforcement learning related lectures in the Boyu learning platform. Also, special thanks go to those who explored reinforcement learning and Atari games with me and share the experience.

References

[He *et al.*, 2016] Frank S He, Yang Liu, Alexander G Schwing, and Jian Peng. Learning to play in a day: Faster deep reinforcement learning by optimality tightening. *arXiv preprint arXiv:1611.01606*, 2016.

[Hessel *et al.*, 2018] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[Mnih *et al.*, 2013] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[Mnih *et al.*, 2015] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[Palanisamy, 2018] Praveen Palanisamy. *Hands-On Intelligent Agents with OpenAI Gym: Your guide to developing AI agents using deep reinforcement learning*. Packt Publishing Ltd, 2018.

[Schaul *et al.*, 2015] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[Van Hasselt *et al.*, 2016] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[Wang *et al.*, 2015] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

A Hyperparameters

Hyperparameters	Value	Description
minibatch size	32	The number of training cases during each training process.
replay memory size	200000	The size of replay memory. Limited by poor computing resources.
agent history length	200000	The number of consecutive frames to concatenate as input.
target network update frequency	30000	The interval between two target network updates.
discount factor	0.99	Discount factor γ in Q-learning algorithm.
frame skip	4	Skip certain number of frames, and repeating same action during skipping.
update frequency	4	The interval between two successive updates.
learning rate	0.00025	The learning rate used by RMSProp.
gradient momentum	0.95	The gradient momentum used by RMSProp.
min squared gradient	0.01	The epsilon used by RMSProp.
initial exploration rate	1	Initial value of ϵ-greedy algorithm.
turning exploration rate	0.1	The turning value of ϵ-greedy algorithm.
turning exploration frame	1000000	The number of frames over which the initial value ϵ is linearly annealed to its turning value.
terminal exploration rate	0.01	The final value of ϵ-greedy algorithm.
terminal exploration frame	10000000	The number of frames over which the turning value ϵ is linearly annealed to its terminal value.
replay start size	50000	The number of frames before learning starts.
alpha	0.6	The parameter α used by prioritized experience replay memory.
initial beta	0.4	The initial value of value β used by prioritized replay memory.
beta increment	0.000024	The increment of β each time before it reaches 1.

Table 2: List of Hyperparameters and Their Values

B Learning Curves

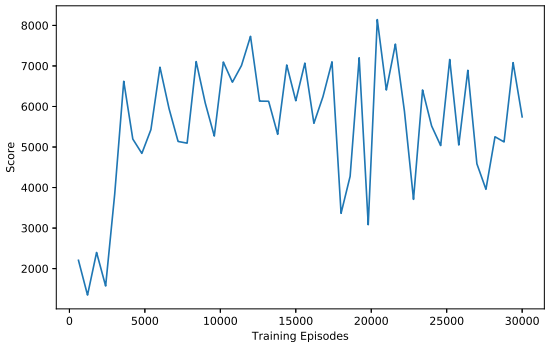

Figure 4: *Krull*

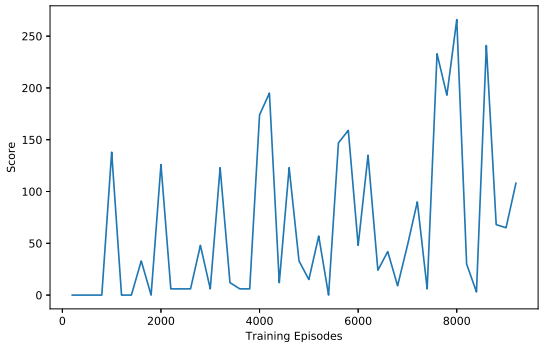

Figure 5: *Tutankham*

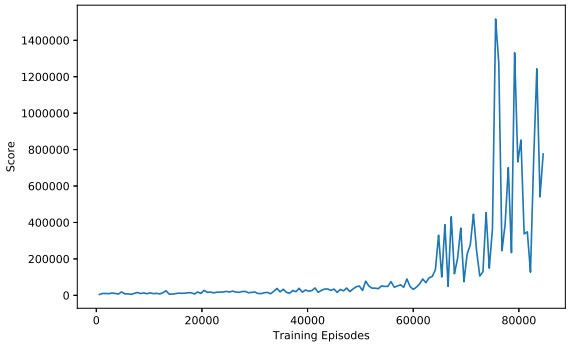

Figure 6: *Atlantis*

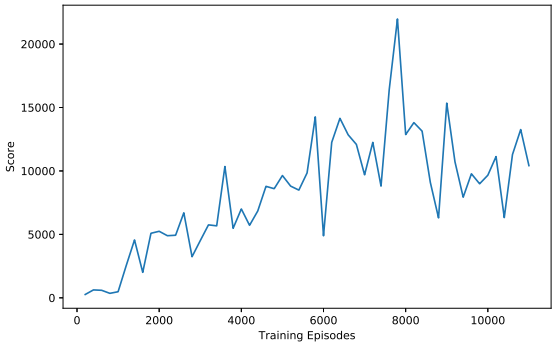

Figure 7: *Beam Rider*

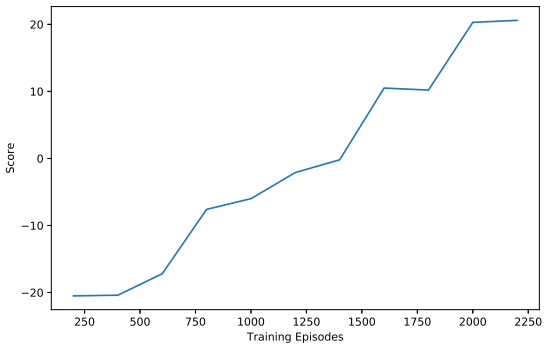

Figure 8: *Pong*

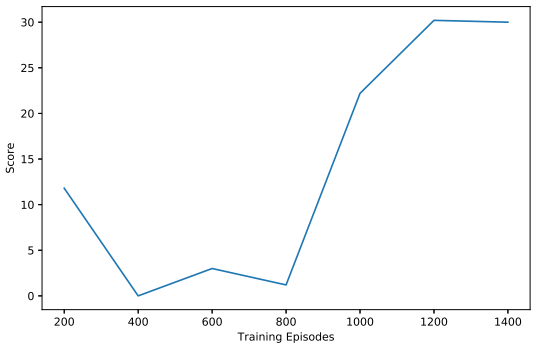

Figure 9: *Freeway*