Automating Flappy Bird using Deep Q-Learning

Radium Zhang

University of Minnesota, Twin Cities, USA

April 15, 2021

1 Introduction

Using the reinforcement learning successfully in situations approaching real-world complexity. Here, we want to design an agent that can automate the flappy bird game based on state vectors instead of images, where the flappy Bird is a game that involves navigating a bird through a bunch of obstacles using reinforcement learning without knowing any information from the environment apriori. Our agents must derive efficient representations of the environment from high-dimensional sensory inputs, and use these to generalize past experience to new situations. This problem is solved by the combination of reinforcement learning and hierarchical sensory processing system. Unlike supervised learning where it has labels for each instances of training example and in unsupervised learning where there are no labels at all, in reinforcement learning, the sparse and time delayed labels-rewards are used to define if the the learning process is good enough or not. Based on the rewards received, the agent learns to behave in a given environment under certain policy. Reinforcement Learning is mainly concerned with the maximization of cumulative reward in return to the agent-environment interaction. Deep Q-Learning is an advanced version of Q-Learning method of Reinforcement Learning where deep neural networks are used to solve Reinforcement Learning problems.

2 Literature Review

One of the top challenges of Reinforcement Learning (RL) is learning to control agents from high-dimensional sensory inputs. The latest breakthroughs in computer vision and speech recognition depend on effective training deep neural network on large training set. The most successful way of direct training from the original input, a lightweight update based on random gradient descent is used. By feeding enough data enters deep neural network, it is usually possible to learn better representation. To strengthen the learning methods we are going to connect the reinforcement learning algorithm to a runnable deep neural network.

Moreover, the training data is processed directly on the RGB image by using random gradient update. By using stochastic gradient updates, connecting a reinforcement learning algorithm to a deep neural network which operates directly on RGB images and efficiently process training data become possible.

2.1 reinforcement learning

Recently, Deep learning in addition to RL has a breakthrough by using a range of neural network architectures which has made it possible to extract high-level features from raw sensory data which can further be used for an agent's learning. However, RL still poses certain challenges from a deep learning perspective which include requirement of large amount of data to train deep learning model whereas in RL the agent must be able to learn from a sparse, noisy and delayed scalar reward signal. Another thing is that in deep learning algorithms, the data samples are assumed to be independent, however, in RL the agent may encounter a sequence of highly correlated states. Also, as the agent learns new behaviours as per the policy, the data distribution might change on go, in contradiction to this, deep learning assumes a fix distribution of data.

2.2 convolutional neural network

In the past, convolutional neural networks are implemented as proven architecture to overcome these challenges by learning policies successfully from raw video frames taken in RL environments. In addition to this, experience relay mechanism helps to resolve the problem of correlated data and non-stationary distribution as it can be used to randomly sample the previous transitions and thereby smooths the training distribution over many past behaviors. Lately, researchers have developed interest in combining deep learning with RL which can be used to extract the information about unseen and unobserved environment.

2.3 neural fitted Q-learning

Similarly, neural fitted Q-learning (NFQ) is another method which has been used in the past that aligns with deep Q-learning which optimizes mean square loss between ground truth Q values and estimated Q values obtained from neural network using RPROP algorithm by updating the Q-network parameters. However, the optimization of loss function in NFQ is computationally expensive since it uses batch update which is proportional to data set size, whereas stochastic gradient updates have lower computational cost and it also scales to large data sets. So far NFQ has been used in simple real world control tasks by first using deep auto encoders to learn low dimensional representation of task and then applying NFQ to it. In contrast to this, deep Q-learning is an end to end RL method which is applied directly to visual inputs to learn features for discriminatory action-values.

2.4 Prior work

Earlier, the problem of automating games like flappy bird have been solved by passing the images of the game through a convolutional neural network and obtaining the states as an output from convolutional neural network. Those states then forms the basis of training using a Deep Q learning algorithm (see section 4.1). Our aim is to train the agent without using images but using the state vectors which we obtain from the OpenAI gym based game engine and evaluate the performance by comparing it with existing image based algorithm which are treated as benchmark during this study. The comparison is also presented in section 5.

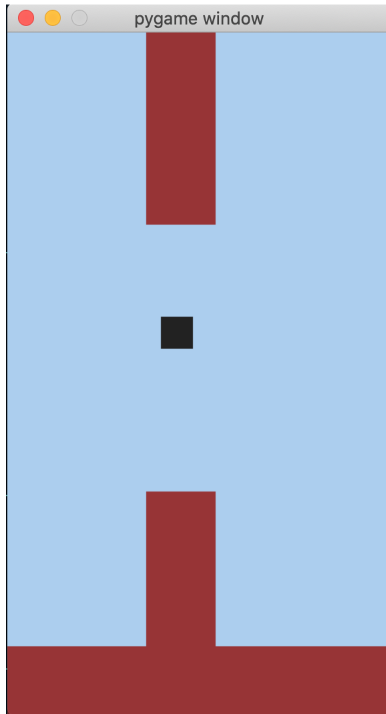


Figure 1: Flappy Bird Environment

3 Game Environment Design

3.1 Description

The goal of this project is to apply and implement modern Deep Reinforcement Learning technique to automate 'Flappy Bird' game where the bird is our agent and the environment is made up with obstacles in form of pipes with different heights. Besides, the snap shot of the Flappy Bird game can be seen in Figure 1, where the black block represents the bird and the brown blocks are obstacles that the bird cannot cross. In this game, the aim is to prevent the bird from dying with collision on pipes, ground or ceiling and help it to learn to go through them to maximize its score in the end. The action space consists of two actions only, one is a tap on screen to give the bird a little flap to gain some height and stay in the air and the other action is doing nothing, which will make the bird lose some height automatically because of the gravity. The bird scores when it successfully passes a pipe which gives it a motivation to increase its score.

3.2 Implementation

The game engine is implemented using OpenAI Gym and Pygame, which can be seen in Figure 1, where the way we build this game engine is to specify positions of the bird and obstacles based on vector first and then fill up with different colors. The aim is to create the agent which is able to learn how to keep the bird alive practically forever, and this is a continuous task since the terminal state is defined as when the bird collides with pipes, ground or ceiling.

The agent will not be provided with any environment specific knowledge as it should be able to learn itself with continuous interactions with the environment at each time step t. Furthermore the network architecture and hyper-parameters are kept constant during one complete training

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    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 D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 2: Deep Q-learning algorithm

exercise. Effect of different network architectures and hyper-parameters is compared and results are shown in section 5.

4 Methods and Algorithms

The motivation behind this approach is that by feeding sufficient large data into deep neural networks it is possible to learn better than handcrafted features. The goal here is to combine Reinforcement Learning with deep neural networks which operate on state vector representation and efficiently process the training data of states observed by the agent by using stochastic gradient updates.

4.1 Deep Q-learning algorithm

First, the full algorithm is presented in Figure 2 [4]. The neural network here is used to estimate the action-value function by updating the parameters of the network using samples of experience of agent with environment as $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$. But the approximation of Q-values using non-linear functions (neural networks) is not very stable. Therefore, a technique known as experience

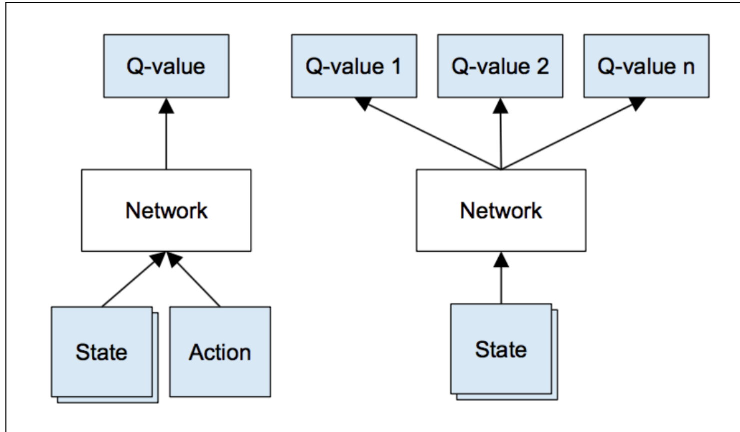


Figure 3: Left: Naive formulation of deep Q-network. Right: More optimized architecture of deep Q-network

replay D is used to store the experiences $(e_1, e_2, e_3, ...e_N)$ of agent over a pool of episodes with $e_t = (s_t, a_t, r_t, s_{t+1})$.

While applying Q-learning updates, a random batch of samples of experiences e, which is called the minibatch, is drawn from the stored samples in D. Based on this, the agent selects and executes an action according to epsilon-greedy policy. One of the advantages of deep Q-learning method over standard Q-learning is that using experience replay helps to make weight updates of neural network more efficiently. Also, as discussed above there is a correlation between consecutive samples, which makes it inefficient to directly learn from them. To resolve this issue, random sample selection from experience replay is done to reduce the correlation between the samples and variance in the updates. Also, there might occur a sudden change in distribution of data in RL due to change in maximizing action. This could result in parameters to get stuck at poor local minimum/maximum or even diverge abruptly. This issue can also be solved using experience replay as it averages the agent's behaviour distribution over number of previous states which smooths the learning and avoids divergence or oscillations of parameters around local minimum/ maximum [3].

Another method to improve the stability is to use a separate network for generating the targets y_j in the Q-learning update. More precisely, every C updates we clone the network Q to obtain a target network $\hat{Q}$ and use $\hat{Q}$ for generating the Q-learning targets y_j for the following C updates [4]. Besides, it can make divergence and oscillations much more unlikely. Currently, we take C=1 and train the flappy bird for simplicity. But in the future, we will tune the value of C and compare the results to find out the influence of it.

4.2 Model architecture

We could have represented our Q-function with a neural network that takes the state and action as input and outputs the corresponding Q-value. Alternatively we can take only the state as input and output the Q-value for each possible action (See Figure 3). This approach has the advantage, that if we want to perform a Q-value update or pick the action with the highest Q-value, we only have to do one forward pass through the network and have all Q-values for all actions available immediately [4].

5 Results

	Training Parameters 1	Training Parameters 2	Training Parameters 3	Training Parameters 4	Training Parameters 5
NN Layer 1	Linear(6,10) + ReLU	Linear(6,20) + ReLU	Linear(6,10) + ReLU	Conv1d(4, 32, 2, 2) + ReLU	Conv1d(4,16, 2, 2) + ReLU
NN Layer 2	Linear(10,10) + ReLU	Linear(20,20) + ReLU	Linear(10,10) + ReLU	Conv1d(32, 64, 2 , 1) + ReLU	Conv1d(16, 16, 3, 1) + ReLU
NN Layer 3	Linear(10,2)	Linear(20,20) + ReLU	Linear(10,10) + ReLU	Conv1d(64, 64 ,2, 1) + ReLU	nn.Linear(16, 16) + ReLU
NN Layer 4	None	Linear(20,2)	Linear(10,2)	nn.Linear(64, 16) + ReLU	Linear(16, 2)
NN Layer 5	None	None	None	Linear(16, 2)	None
Max Iterations	2000000	1000000	1500000	1000000	2000000
Replay Memory	100000	100000	300000	300000	300000
Minibatch size	32	32	32	64	32

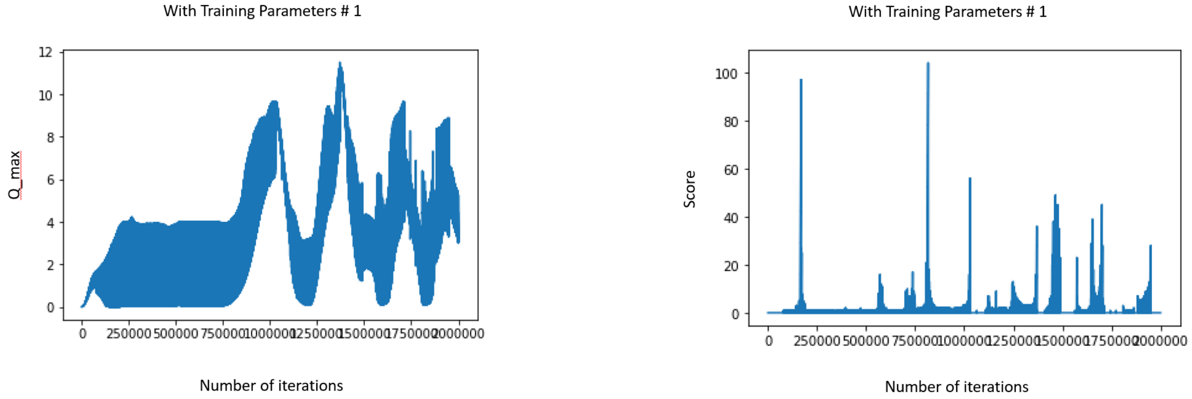
Figure 4: Neural Network Architectures for Training through Deep Q-Learning

5.1 Simplified Environment

To begin with, we simplify the environment from random position of pipes to fixed position which will shorten the learning time and help us understand how parameters in Fig. 4 influence the agent better. Then we will change the environment back and train the agent again to see if we can make the agent play the game succesfully.

During the implementation, different architectures of neural networks, which can be seen in Figure 4, were trained and tested to have a comparative analysis of training performance of the agent using deep Q-learning approach. The weights of neural networks were initialized uniformly at random in the range -0.01 to 0.01 with bias equals to 0.01. The set of actions which the agent can take is either 0 or 1 where 1 means flap and 0 means doing nothing. For the hyper parameters of neural networks which have been tried while training the agent using γ equals to 0.99, initial ϵ equals to 0.1 which is linearly decreased per iteration up to final ϵ as 0.0001.

First, let's focus on the general hyper-parameters and then analyze the neural network architectures, i.e. the combinations of layers of the neural network. To begin with, as per the past studies the discount factor γ, γ needs to be in the range $[0, 1]$ where 1 means giving more importance to the future return instead of immediate reward. Then since the agent's main goal is to pass through as many pipes in the future as it can to accumulate the maximum return, γ is chosen close to 1. Also, to maintain a balance between exploration vs exploitation, the ϵ values are kept a bit high in the beginning so that the agent takes random action initially to explore the possible values of action value pairs $Q(s, a)$ and then as the number of iterations goes up it only takes the actions which maximizes $Q(s, a)$ for all s according to ϵ-greedy approach since the agent want to apply the learned policy when it has learned a lot from past experience instead of keeping trying new random



(a) Q(s,a) per iteration for Training Parameter 1

(b) score per iteration for Training Parameter 1

Figure 5: Parameter 1 result

actions.

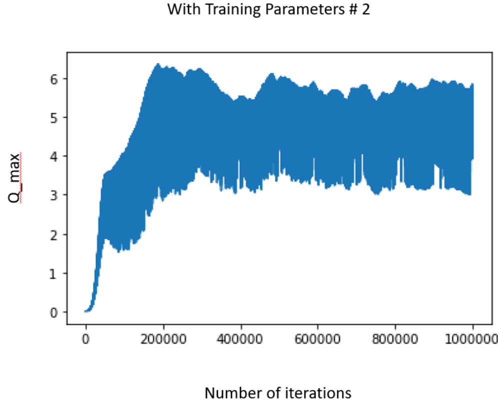
Then in Fig. 4, we have trained neural networks for different large number of iterations and size of experience replay with various minibatch sizes to optimize the training speed for the agent. The major challenge encountered while implementation of the project was training speed of the agent. While we have tried to run the training process for at least 30% of total number of iterations and we found that the training speed of the agent was very well handled by using minibatch sizes from 32 to 64.

It was observed that the training speed is increased as the minibatch size of experience samples is increased since these are the samples which are randomly selected every time from experience replay to use the memory of the actions taken by the agent in the past and to average out the agent's behaviour distribution. So the larger minibatch size we choose, the longer computation time we would have. The reason why the minibatch is helpful is for the case when there is sudden change in the distribution due to change in selection of action which maximizes the action-value pair.

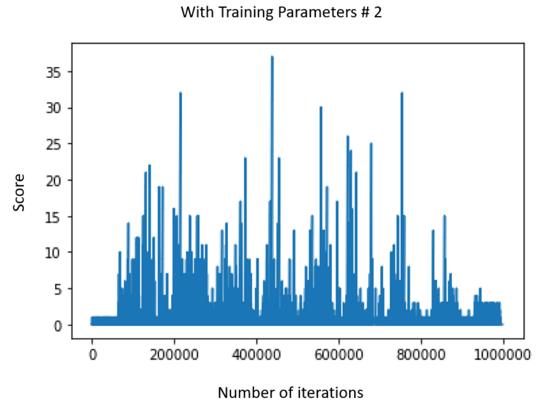
Then, let's see what is influence of different neural network architectures referring to Figure 4. Figure 5 to Figure 9 are plots for $Q(s,a)$ vs number of iterations and score of agent vs iterations where the score implies number of pipes crossed by the agent without dying per episode and $Q(s,a)$ is the maximum Q-value according to the chosen action per iteration.

The general trend observed for $Q(s,a)$ per iteration is increasing which was the desired case, however, the dips in the $Q(s,a)$ values can be seen in between which might be the case when the past experiences replay was fully exploited by the agent and it now has no memory of past experiences for the actions taken in a state due to which it starts appending new experiences in the replay for its knowledge of environment. One more reason which might explain the oscillations in $Q(s,a)$ graph is because of smaller experience replay since using an older set of parameters adds a delay between the time an update to Q is made and the time the update affects the target which makes divergence or oscillations much more unlikely.

Figure 5a shows the $Q(s,a)$ per iteration for the neural network architecture with Training Parameters 1 and max iterations 2 million. It can be seen that for the first 0.75 million iterations, the agent tries to learn and gain some information about the environment and takes actions in such a way that $Q(s,a)$ is always increasing and then as the experience replay is utilized the action-value shows a dip as agent will have to learn new experiences for the future actions.The maximum value

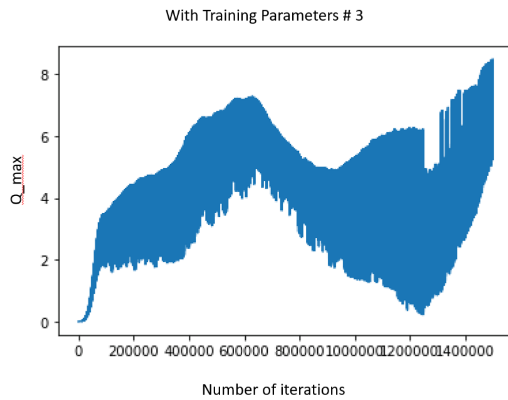


(a) Q(s,a) per iteration for Training Parameter 2

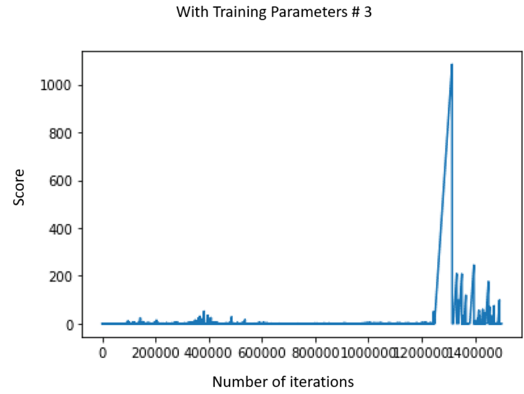


(b) score per iteration for Training Parameter 2

Figure 6: Parameter 2 result



(a) Q(s,a) per iteration for Training Parameter 3



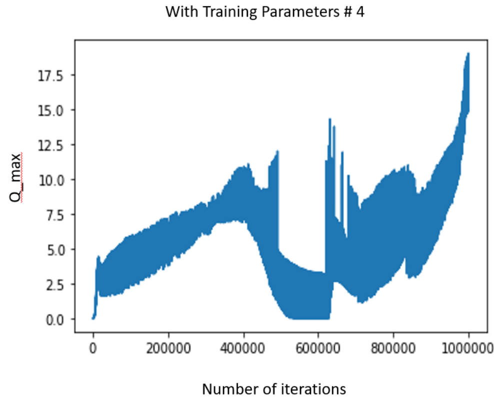
(b) score per iteration for Training Parameter 3

Figure 7: Parameter 3 result

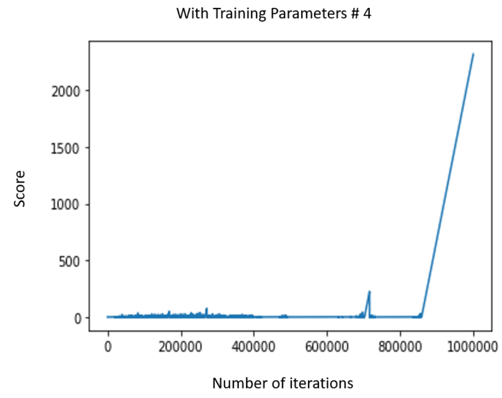
for $Q(s, a)$ recorded for this neural network architecture was approximately 11 and the highest score in terms of pipes crossed by the agent was around 110 from Figure 5b.

It can be seen that the agent was not well-learned with parameters 1 and we supposed that it's because the neural network is not large enough to contain all information needed, which may explain why there is many oscillations in the trend for $Q(s, a)$. Therefore, we increased the parameters in neural network to training parameters 2 as referred to Figure 4. Then, the more general trend for $Q(s, a)$ can be seen for training parameters 2 in Figure 6a, as it shows increasing behaviour till the end of the process and the highest value recorded for $Q(s, a)$ is 6 in this case with highest score equals to around 40 in Figure 6b. From Figure 7a and Figure 7b, it can be observed that as we increase the experience replay from 10% to 20% of total number of iterations as seen in Figure 4 for Training Parameters 3, the behaviour achieved by agent is miraculous where it has crossed approximately up to 1200 pipes without dying with $Q(s, a)$ increasing per iteration with maximum of 8.

The next step was to try convolutional neural network(CNN) to see the performance over fully connected neural networks (FCNN) with linear layers and ReLU activation function. Even though

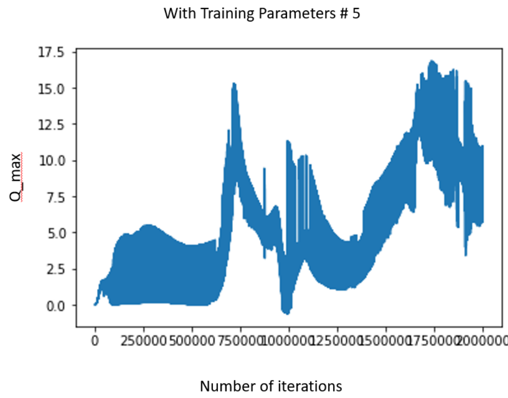


(a) Q(s,a) per iteration for Training Parameter 4

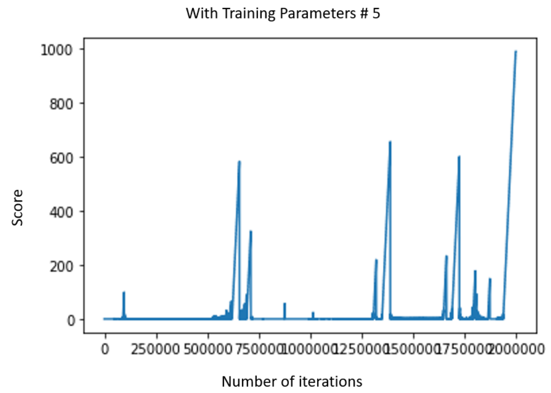


(b) score per iteration for Training Parameter 4

Figure 8: Parameter 4 result



(a) Q(s,a) per iteration for Training Parameter 5



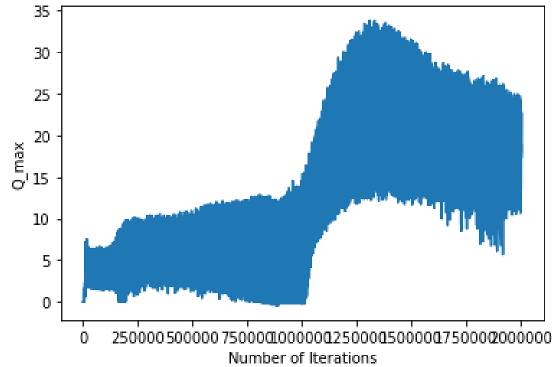
(b) score per iteration for Training Parameter 5

Figure 9: Parameter 5 result

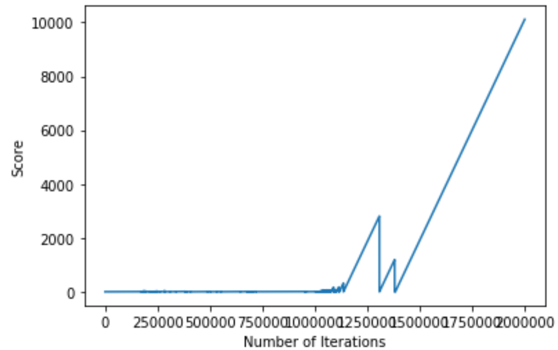
in general CNN deals with image-based problems and we have the vector based problem here, but we can stack the four consecutive states in a trajectory together to get a matrix which can be given as input to CNN input layer and processed then. Through this way, the agent will be able to know the ordered information in a small time interval and we expected to have a better result.

This way it also helps to hold the information about a trajectory followed by the agent. It was seen that CNN is little bit faster as compared to FCNN and it outperforms all previous approaches. As per Training Parameters 4 in Figure 4, Figure 8a explains the performance of the agent where the $Q(s, a)$ is seen increasing with iteration and the dip in its value is might be because of the reason as explained earlier. The maximum value for $Q(s, a)$ in this case was around 19.

Also, from Figure 8b we can see that for upto 0.8 miliion iterations the agent just learns with its experiences and then it has crossed approximately 2500 pipes in an episode which is more than what we got for FCNN architectures. Finally in Figure 9a, for CNN as the experience replay is reduced it can be seen that the maximum value attained for $Q(s, a)$ is reduced to upto 17 and score attained by the agent is up to 1000 as in Figure 9b.



(a) Q(s,a) per iteration for Training Parameter 6



(b) score per iteration for Training Parameter 6

Figure 10: Parameter 6 result

5.2 Complex Environment

The parameter4 in Fig. 4 gives us the best result for the simplified environment as described in the previous section. Then after changing the positions of pipes to be randomly instead of fixed, we use the same parameter4 in Fig. 4 expect for increasing the size of replay memory to 1 million to decrease the presentations of dips as explained above and increasing the number of iterations to 2 million to give the agent enough time to learn the environment. The other reason to increase the size of replay memory is because the agent needs more memory to store a more complex environment. As seen in Fig. 10a and Fig. 10b, the maximum score reaches 10000 and the $Q(s, a)$ reaches up to 35 this time. Besides, $Q(s, a)$ has a good increasing trend without many dips. From these training results, we can conclude that the agent we trained is able to play the flappy bird game with excellent performance, which will exceed almost every human's performance.

During the training procedure of the agent, we keep storing the model we used and then we can test the agent with the pre-trained model. The deep Q learning code and the flappy bird game engine code are available on https://github.umn.edu/zhan5821/CSCI5512_Project. Besides, we have recorded the video in the previous URL to show that how will the bird passes random pipes practically forever. For the code, we have modified the game engine made available on Github at [1] and trained with different network architecture, and then compared with Deep Q learning implementation for image based environment at [2].

The training speed of our algorithm is still very slow. It takes around 10 hours to train for two million iterations on a standard PC with 16 GB RAM and NVIDIA GPU. Nevertheless, the training speed is faster than that of image based deep Q learning algorithm which takes roughly 15-16 hours to train for two million iterations on same PC.

6 Conclusions and Future Work

In conclusion, we attempted to automate the game flappy bird without giving the agent any environment related information except for rewards that it gets for different actions. The action value function was represented using a neural network and the network was trained using deep Q learning algorithm to store the Q values. Simple linear Neural network performed poorly and changing the number of neurons and layers had little effect on the learning. We switched to 1D convolutional neural network which performed better than linear neural networks. During training for the simplified environment, maximum score achieved by the agent was around 2500 pipes. And then

we use the parameter with best performance for the simplified environment to learn the complex environment, the maximum score achieved by the agent is about 10000. Then the same model was used for testing which made the agent to go on without dying practically indefinitely.

Besides, we found out that our vector-based learning for flappy bird game is a little bit faster than the image-based one, which may give the intuition that we may decrease the information contained in the image further to get a better training speed since the image is constituted with vectors. So if we can find a proper method to do pre-processing on the image to keep the necessary information before training, we will have a better learning speed, and this is the work we want to study in details in the future.

In future work, we plan to increase training speed by tuning the values of C or by using simpler neural network. We also plan to explore what effect changing the profile of decay of epsilon has on overall training exercise.

References

[1] jmathison/gym-simpleflappy. https://github.com/jmathison/gym-simpleflappy.git, December 2019.

[2] nevenp/dqn_flappy_bird. https://github.com/nevenp/dqn_flappy_bird, December 2019.

[3] 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.

[4] 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, 2015.