Flappy Bird Hack using Deep Reinforcement Learning with Double Q-learning

Jianqiu Kong
University of Illinois
Urbana Champaign
jkong11@illinois.edu

Naman Shukla
University of Illinois
Urbana Champaign
namans2@illinois.edu

Shubham Bansal
University of Illinois
Urbana Champaign
shubham8@illinois.edu

Ziyu Zhou
University of Illinois
Urbana Champaign
ziyuz2@illinois.edu

Zhenye Na
University of Illinois
Urbana Champaign
zna2@illinois.edu

Abstract

This implementation introduces training a software agent via deep reinforcement learning with double deep Q-learning technique. The environment with which the agent is interacting is a game simulation - Flappy Bird. Double Deep Q-learning is used to tackle overestimation of values in Deep Q-learning which arises due to coupling of action selection and evaluation. We present extensive study of sensitivity analysis of hyper-parameters as well as specific model based choices based on consistent environment metric i.e. Flappy Bird game. We also present the analysis on the results obtained by running different experiments for the different model parameters. [1]

1. Introduction

Mnih et al [4] has demonstrated that human-level control can be achieved using Deep Reinforcement Learning via single learning model based on Q-learning. Although, DQN out performs most of the environment specific state of the art algorithms, it still suffers from overestimation issue [2]. Hado van Hasselt et al [7] presented the idea of Double Deep Q-learning to resolve this issue. Here, the aim is to experiment with different network parameters and hyper-parameters to test the idea of Double Deep Q-learning on an environment. [2]

[1] codes for this project could be found here: https://github.com/drl-dql/DQN-Flappy-Bird

[2] experiment results database: https://drive.google.com/drive/folders/1qrXkRJQ3kpdVhvrTNKhWqlAPm3V-52_r?usp=sharing

1.1. Paper Overview

The idea of using double deep networks for Q-learning to solve the overestimation problem of the Q-learning algorithm was proposed by Hado van Hasselt et al in their paper Deep Reinforcement Learning with Double Q-learning [7].

1.1.1 DQN with Target Network

In the DQN, based on the Bellman optimality, the target is defined as,

$$Y_t^{\text{DQN}} = R_{t+1} + \gamma(\max_a(Q(S_{t+1}, a; \theta_t)))$$

where θ_t is the parameters of the network at time t. To update the parameters, we take a Stochastic Gradient Descent (SGD) step,

$$\theta_{t+1} = \theta_t + \alpha(Y_t^{\text{DQN}} - Q(S_t, A_t; \theta))\nabla_\theta Q(S_t, A_t; \theta)$$

This assumes that the Q target Y_t^{DQN} and the Q value $Q(S_t, A_t; \theta)$ use the same parameters and are updated together. It will introduce high correlations between Q target and Q value which makes the training very unstable, meaning that at every step of training, the Q values shift but also the target value shifts.

To break this kind of correlation, Mnih et al. [4] propose to use a separate target network whose parameters are copied every τ steps from the original Q value network and are kept fixed on all other steps. The target network is now,

$$Y_t^{\text{DQN}} = R_{t+1} + \gamma(\max_a(Q(S_{t+1}, a; \theta_t^-)))$$

where θ_t^- represents the parameters of the target network. At every τ steps, the parameters are updated as, $\theta_t^- = \theta_t$.

1.1.2 Problems of DQN

Recall that the target Y_t^{DQN} is defined as

$$Y_t^{\text{DQN}} = R_{t+1} + \gamma(\max_a(Q(S_{t+1}, a; \theta_t^-)))$$

The $\max$ operator in the target network, uses the same parameters to select and evaluate the actions. The problem is that the best action for the next state is not necessarily the action with the highest Q value. As illustrated in Thomas Simonini's blog [5]:

> At the beginning of the training we don't have enough information about the best action to take. Therefore, taking the maximum Q value (which is noisy) as the best action to take can lead to false positives. If non-optimal actions are regularly given a higher Q value than the optimal best action, the learning will be complicated.

1.1.3 Double DQN

Double DQN addresses the above problem by decoupling action selection and action evaluation. We first use our Q value network, i.e., the original on-line network, to select what is the best action to take for the next state (the action with the highest Q value). Then the target network will be used to calculate the target Q value of taking that action at the next state. Now, Y_t becomes,

$$Y_t^{\text{DoubleDQN}} = R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \theta_t); \theta_t^-)$$

2. Details of Approach

2.1. PyGame Learning Environment

We used PyGame Learning Environment (Figure 1) developed by Tasfi Norman [6] for experimenting. PyGame Learning Environment (PLE) is a learning environment, mimicking the Arcade Learning Environment interface, allowing a quick start to Reinforcement Learning in Python. The goal of PLE is allow practitioners to focus design of models and experiments instead of environment design.

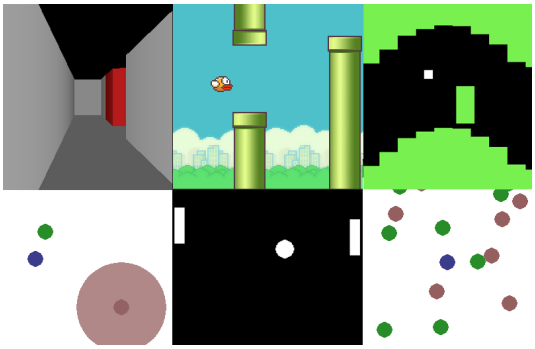

Figure 1: PyGame Learning Environment

2.2. Model Architectures

The Convolution Network used in the experiments is exactly the one proposed by Mnih et al. (2015). Briefly, the network architecture is a Convolution Neural Network (Fukushima, 1988 [1]; LeCun et al., 1998 [3]) with 3 convolution layers and a fully-connected hidden layer (approximately 1.5M parameters in total).

The input to the network is a $84 \times 84 \times 4$ tensor containing a rescaled and gray-scale version of the last four frames. The first convolution layer convolves the input with 32 filters of size 8 (stride 4), the second layer has 64 layers of size 4 (stride 2), the final convolution layer has 64 filters of size 3 (stride 1). This is followed by a Fully-Connected hidden layer of 512 units. All these layers are separated by Rectifier Linear Units (ReLU). Finally, a fully-connected linear layer projects to the output of the network, i.e., the Q-values. The Optimization methods employed to train the network are RMSProp (with momentum parameter 0.95), Adam as well as Stochastic Gradient Descent (with momentum parameter 0.95).

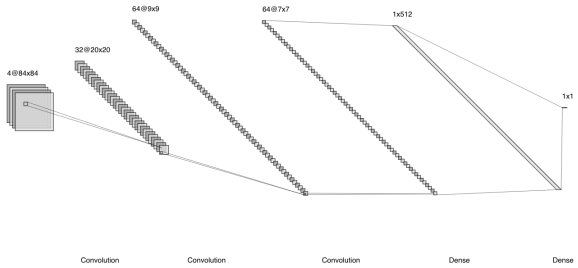

Figure 2: Model architecture

2.3. Hyper-parameter Choices

In the implementation, different combinations of hyper-parameters like learning rate, discount factor, update target frequency, and batch size as well as optimizer have been used as experimental setups. Please refer to the table 1 for more details.

Here are some default parameters:

Hyperparameter	Value
replay buffer size	50000
total episodes	100000
initial epsilon	0.1
minimum epsilon	0.0001
epsilon discount rate	1×10^{-7}
screen size	$84 \times 84 \times 4$
number of saved model	5

2.4. Training Methods

In this implementation, we update policy network after specific number of episodes (determined by Update Target Frequency parameter) and not after certain number of time steps (N^-) within a single episode. Here are several functions in Agent class *Agent.py* explained in detail.

2.4.1 Build Network

Initializes Q and Q_{Target} network based on model architecture declared in *Model.py* and action number (number of valid actions in a specific game). Also, initializes optimizer as specified. Please note that Q and Q_{Target} network have identical network architecture.

2.4.2 Update Target Network

Updates Q_{Target} with parameters of Q network.

2.4.3 Update Q Network

Put Q and Q_{Target} network in evaluation mode. Now, we use current Q network to evaluate action a' based on the new state s':

$$\arg\max_{a'} Q_{\text{Current}}(s', a')$$

Now, use Q_{Target} network to evaluate value using R (current reward) and future rewards based on $s\prime$ and $a\prime$:

$$Y = R + \gamma \times Q_{\text{Target}}(s\prime, a\prime)$$

Using Q_{current}, state and action we determine Q_{value}.

Using this Q_{value} and Y (from above), we calculate mean squared loss (MSE). This loss is used for backward propagation and eventually in taking an optimizer step.

2.4.4 Take Action

Puts Q_{current} in evaluation mode. Returns an action based on $\epsilon-$greedy algorithm, which is with ϵ probability choose random action else choose $\arg\max Q$ estimate action.

2.4.5 Update Epsilon

If $\epsilon_{\text{value}} > \epsilon_{\text{min value}}$, reduce ϵ_{value} by Discount Rate(ϵ).

3. Results and Analysis

We performed multiple different experiments on different hyperparameters with all other parameters fixed to see the influence of the single hyperparameter. All the experiments are done with a time limit of 48 hours of training.

3.1. Batch Size Analysis

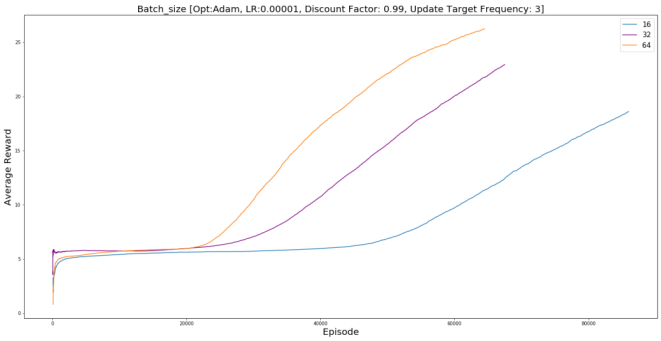

Figure 3: Batch Size Rewards

Batch size refers to the number of training examples utilized in one iteration. We may assume that larger batch size lead to larger average reward and higher stability.

We have experimented on three batch sizes, 16, 32 and 64. As we can see in Figure 3, the reward is increasing almost linearly with the batch size after the first 20000 episodes. As a larger batch size allows more gradient updates per episode, the policy network is able to learns faster than a smaller batch size within the same number of episodes.

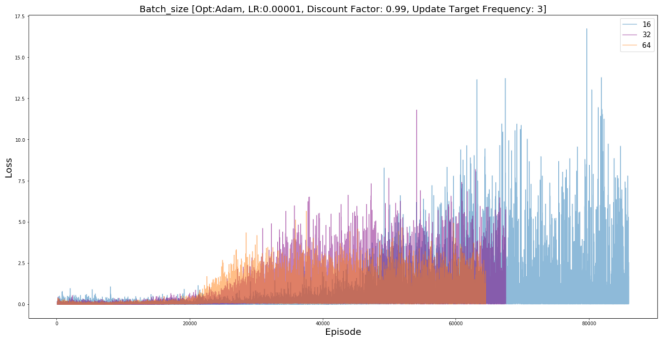

Figure 4: Batch Size Loss

The stability information may also be revealed in Figure 4, the loss plot of different batch sizes. The error incurred tend to be more stable and smaller with the increase in batch size. As the training time is fixed, it's reasonable that a larger batch size will need more time to train one episode and given same number of episodes, experiments with larger batch size result in higher average rewards.

3.2. Discount Factor Analysis

The discount factor (γ) determines the importance of future rewards. A factor of 0 makes the agent "opportunistic" by only considering current rewards, while a factor approaching 1 will make it strive for a long-term high reward.

Algorithm 1 Double DQN Algorithm.

Input
  $\mathcal{D}$ empty replay buffer
  θ initial network parameters
  θ^- copy of θ
  N_r replay buffer maximum size
  N_b training batch size
  N^- target network replacement frequency
for episode $e \in \{1, 2, \ldots, M\}$ **do**
  Initialize frame sequence $\mathbf{x} \leftarrow ()$
  for $t \in \{0, 1, \ldots\}$ **do**
    Set state $s \leftarrow \mathbf{x}$, sample action $a \sim \pi_{\mathcal{B}}$
    Sample next frame x^t from environment ϵ given (s, a) and receive reward r, and append x^t to $\mathbf{x}$
    If $|\mathbf{x}| > N_f$, delete oldest frame x_{tmin} from $\mathbf{x}$
    Set $s' \leftarrow \mathbf{x}$, and add transition tuple (s, a, r, s') to $\mathcal{D}$,
      replacing the oldest tuple if $|\mathcal{D}| > N_r$
    Sample a minibatch of N_b tuples $(s, a, r, s') \sim \text{Unif}(\mathcal{D})$
    Construct target values, one for each of the N_b tuples:
      Define $a^{\max}(s'; \theta) = \arg\max_{a'} Q(s', a'; \theta)$
      $y_j = \begin{cases} r & \text{if } n \text{ if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{\max}(s'; \theta); \theta^-) & \text{otherwise} \end{cases}$
    Do a gradient step with loss $\| y_j - Q(s, a; \theta)$
    Replace target parameters $\theta^- \leftarrow \theta$ every N^- steps
  end for
end for

If the discount factor meets or exceeds 1, the Q values may diverge. It somehow measures how far ahead in time the algorithm looks.

We have compared the influence discount factor makes to the average rewards and loss of multiple training results. In order to do this, we fixed all the other parameters, only changed the value of discount factor γ ranging from 0.9 to 0.99 to test.

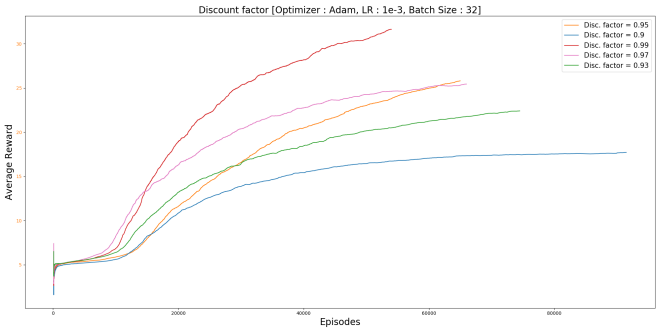

Figure 5: Discount Factor - Average Rewards

As we can see in Figure 5, this is the how discount factor influenced average reward. This is exactly what we expected to see. In this figure, from top to bottom, the discount factor are 0.99, 0.97, 0.95, 0.93 and 0.90 in order. Since we consider for a long-term observation of policies, the average reward of larger discount factor performs better than experiments with smaller discount factor.

As for the loss during training in figure 6, even though the losses of training deep reinforcement learning does not tell much, we can still observe that there is a trend that experiments with larger discount factor has a higher or more fluctuating loss, while smaller discount factor has a more stable loss during training.

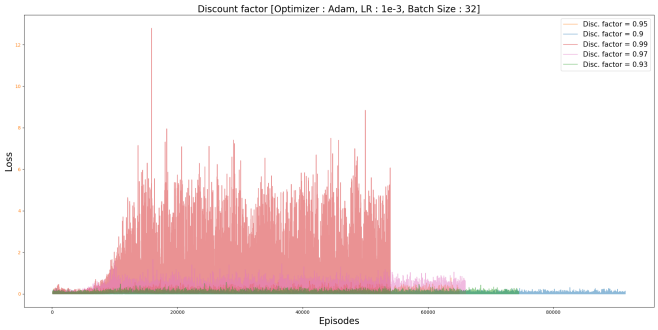

Figure 6: Discount Factor - Loss

3.3. Learning Rate Analysis

Learning rate is the hyper-parameter that controls how much we are adjusting the weights of the network with respect to the loss gradient. It is important to figure out the appropriate learning rate for a certain network architecture.

From figure 7, one interesting point to notice is that the policy network barely learns anything with either too large or too small learning rates. A learning rate of 0.01 is too fast to update the weights of the network and 0.00001 is too slow to update the network weights. For those values with a reasonable average reward, the reward plot shows a similar relationship as to the batch size reward plot. Learning rates of 0.0001, 0.00015 and 0.00025 are of the same magnitude and an approximate linear relationship of the average reward increase vs. learning rates can be observed. With a larger learning rate, the network updates the weights with a faster speed and thus incur faster reward increase than smaller learning rate within a reasonable range.

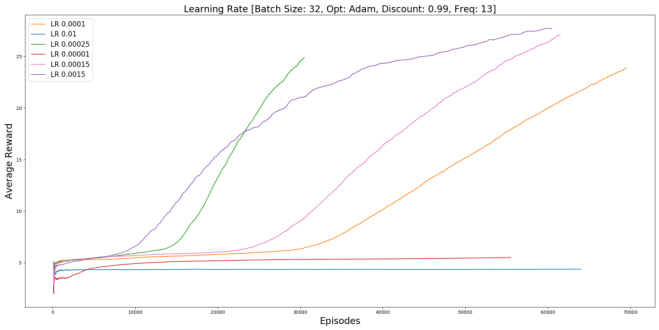

Figure 7: Learning rate - Average Rewards

Figure 8: Learning rate - Loss

3.4. Optimizer Analysis

It takes a long time to train a reinforcement agent and the training stability itself could be a point of consideration, the choice of optimizer plays a key role in learning. We have considered two different optimizers for loss minimization - RMS prop and Adam.

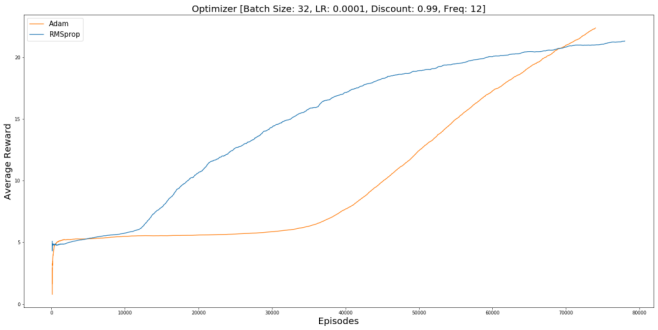

Figure 9: Optimizer selection - Average Rewards

Figure 9 demonstrates that the trajectory of average reward is quite different in both cases. The rate of change of the marginal gain in average reward is negative. This perhaps makes the agent to receive asymptotic average reward as the number of episodes increases. On the other hand, Adam pretends to be far slower in learning but the rate of change of marginal gain in average reward is positive. This implies the agent might not get saturated rewards after a certain stage of training.

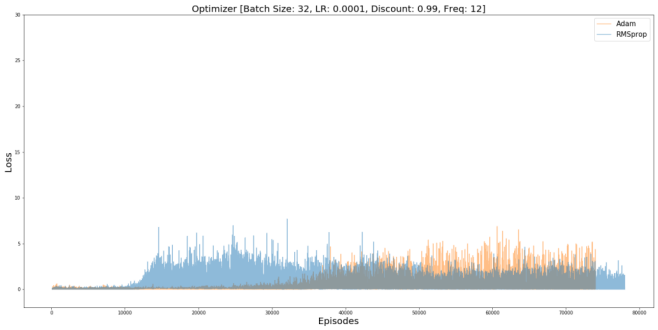

Figure 10: Optimizer selection - Loss

The loss values present in figure 10 are showing similar trend as figure 9. The initial loss values for RMS prop is dominant over Adam. Eventually, Adam loss values surpasses RMSprop's loss values. This is consistent with the previous observation of higher loss corresponding to higher rewards.

3.5. Update Target Frequency Analysis

Updating the target network after a certain number of iteration is referred as update target frequency. This hyper-parameter is specific to DDQN architectures. Here, we have attempted 3, 7, 9, 10, 12, 13 and 15 frequency values and observed the performance of the agent.

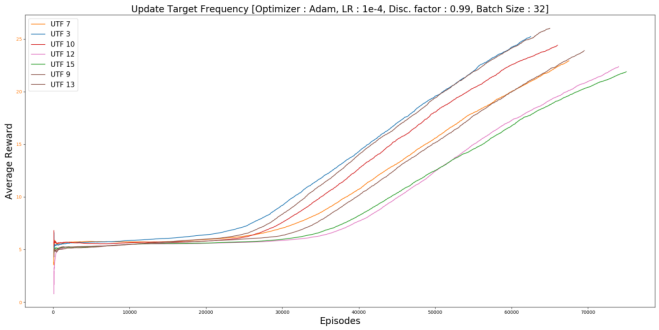

Figure 11: Update Target Frequency - Average Rewards

The overall trend observed in figure 11 is such that the curves corresponding to different values of hyper-parameter differ very slightly. Nevertheless, according to figure 12 the agent receives higher average reward for lower update frequency value at a certain stage of training. Also, the initial rewards are indistinguishable for different frequency values. This demonstrate that the untrained/semi-trained networks would not have coupling issues as the reward signal is low due to random weights and biases of the target and action selection network.

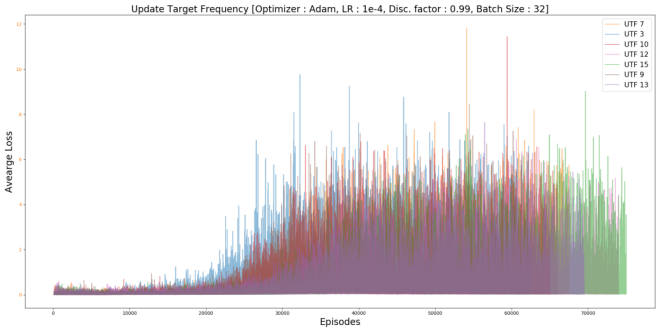

Figure 12: Update Target Frequency - Loss

The loss values observed from figure 12 follows the similar trend as the average reward values. Although the loss values are noisy, there exists a high overlap between the loss values corresponding to the different set of hyper-parameters.

4. Computational Hours

We have fully utilized Blue Waters as well as Google Colab for training.

Platform	Computation hours used
Blue Waters	$\sim$ 1300
Google Colab	$\sim$ 30 (for training and environment pre-configuration)

5. Comparison with Paper's Results

The results provided by Hado van Hasselt et al on various Atari games are in comparison with the DQN through normalized scores. Although the objective of our implementation is different from the original paper baseline comparison, we observed similar trend in loss function (a bit noisy than expected) and average reward as Wizard of Wor and Asterix as mentioned in the original paper.

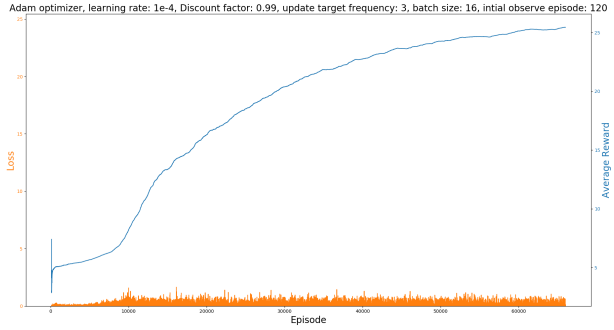

Figure 13: Best Performing Agent

In the original paper, the robustness is judged by executing training at various starting point and observing the generalization of DDQN. In our environment i.e. Flappy Bird, we don't have to explicitly attempt different starting points for the game since this randomness is the part of the game itself. However, the observation is consistent with original paper's conclusion about the robustness of the model i.e. the agents performance is independent of the stating point.

6. Conclusion

We proposed and implemented Double DQN algorithm (with slight modifications) using PyTorch, which uses a convolution neural network architecture proposed by Mnih et al. (2015). We developed and ran 50 different experiments based on different sets of hyper-parameters. The results of these experiments helped us analyze affect of different hyper-parameters on quality of policy learned by the agent. This was achieved by running sets of experiments where only one hyper-parameters varied and rest remained constant. Our study yielded interesting insights about impact of batch size, optimizer, learning rate, discount factor, update target frequency on cumulative average rewards as explained in Section 3.

7. Statement of individual contribution

Jianqiu Kong

Jianqiu along with Zhenye build CNN used for Q-value. She also did unit testing and final visualization for the application.

Naman Shukla

Naman was responsible for configuring the environment in Google Colaboratory as well as in Blue Waters. Along with Shubham and Zhenye, he built the final application.

Shubham Bansal

Shubham was in charge of running majority of the experiments. He developed DDQN agent network with Ziyu Zhou and helped Naman in integrating the modules into a single application.

Ziyu Zhou

Ziyu took responsibility of building DDQN agent network along with Shubham. Also, she wrote majority of utility functions for the application.

Zhenye Na

Zhenye built the CNN along with Jianqui. He also architected the entire pipeline for the final application integration using different modules.

References

[1] K. Fukushima. Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural Networks*, 1:119–130, 1988.

[2] H. V. Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems*, pages 2613–2621, 2010.

[3] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, volume 86, pages 2278–2324, 1998.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[5] T. Simonini. Improvements in deep q learning: Dueling double dqn, prioritized experience replay, and fixed..., Jul 2018.

[6] N. Tasfi. Pygame learning environment. `https://github.com/ntasfi/PyGame-Learning-Environment`, 2016.

[7] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

Table 1: Experimental Setups

Account used	Configuration #	Optimizer	LR	Disc. Factor	Update target frequency	Batch size	Initial observe episode
Naman's Account	0	Adam	1.00E-04	0.99	7	32	80
Naman's Account	1	Adam	1.00E-04	0.99	3	32	120
Naman's Account	2	Adam	1.00E-04	0.99	10	32	100
Naman's Account	3	Adam	1.00E-04	0.99	10	16	100
Naman's Account	4	Adam	1.00E-04	0.99	12	32	100
Naman's Account	5	Adam	1.00E-04	0.99	12	16	200
Naman's Account	6	Adam	1.00E-04	0.99	15	32	100
Naman's Account	7	Adam	1.00E-04	0.99	10	16	150
Naman's Account	8	Adam	1.00E-04	0.99	7	16	180
Naman's Account	9	Adam	1.00E-04	0.99	12	16	180
Naman's Account	10	Adam	1.00E-04	0.99	9	32	120
Naman's Account	11	Adam	1.00E-04	0.99	9	16	100
Naman's Account	12	Adam	1.00E-04	0.99	15	16	120
Naman's Account	13	Adam	1.00E-04	0.99	3	16	120
Naman's Account	14	Adam	1.00E-04	0.99	15	64	100
Naman's Account	15	Adam	1.00E-04	0.99	7	64	120
Naman's Account	16	Adam	1.00E-04	0.99	3	64	120
Naman's Account	17	Adam	1.00E-04	0.99	9	64	150
Naman's Account	18	Adam	1.00E-04	0.99	13	64	150
Naman's Account	19	Adam	1.00E-04	0.99	13	32	120
Naman's Account	20	Adam	1.00E-03	0.95	13	32	120
Naman's Account	21	Adam	1.00E-03	0.9	13	32	120
Naman's Account	22	Adam	1.00E-03	0.99	13	32	120
Naman's Account	23	Adam	1.00E-03	0.97	13	32	120
Naman's Account	24	Adam	1.00E-03	0.93	13	32	120
Naman's Account	25	Adam	1.00E-04	0.95	13	32	120
Naman's Account	26	Adam	1.00E-04	0.9	13	32	120
Naman's Account	27	Adam	1.00E-04	0.97	13	32	120
Naman's Account	28	Adam	1.00E-04	0.93	13	32	120
Naman's Account	29	Adam	1.00E-03	0.9	10	32	120
Shubh's Account	30	Adam	1.00E-02	0.99	13	32	120
Shubh's Account	31	Adam	2.50E-04	0.99	13	32	120
Shubh's Account	32	Adam	1.00E-05	0.99	13	32	120
Shubh's Account	33	Adam	1.50E-04	0.99	13	32	120
Shubh's Account	34	Adam	1.50E-03	0.99	13	32	120
Zhenye's Account	35	RMSProp	0.00025	0.99	4	32	100
Kong's Account	36	RMSprop	1.00E-04	0.99	7	32	80
Kong's Account	37	RMSprop	1.00E-04	0.99	3	32	120
Kong's Account	38	RMSprop	1.00E-04	0.99	10	32	100
Kong's Account	39	RMSprop	1.00E-04	0.99	10	16	100
Kong's Account	40	RMSprop	1.00E-04	0.99	12	32	100
Kong's Account	41	RMSprop	1.00E-04	0.99	12	16	200
Zhenye's Account	42	RMSprop	1.00E-05	0.99	10	32	100
Zhenye's Account	43	RMSprop	1.00E-04	0.99	10	32	100
Zhenye's Account	44	RMSprop	0.0002	0.99	4	32	100
Kong's Account	45	SGD	1.00E-03	0.99	7	16	180
Kong's Account	46	SGD	1.00E-03	0.99	9	16	100
Kong's Account	47	SGD	1.00E-03	0.99	9	32	120
Kong's Account	48	SGD	1.00E-03	0.99	10	16	150
Kong's Account	49	SGD	1.00E-03	0.99	12	16	180
Kong's Account	50	SGD	1.00E-03	0.99	15	32	100