An AI for Game *Tokkun'99*
CS 221 Course Project
Final Report

Yilong Li*, Shiyu Liu†, Zhefan Wang†
*Department of Computer Science, Stanford University
†Department of Electrical Engineering, Stanford University
{yilong, shiyuliu, zwang141}@stanford.edu

I. Game Introduction

Game playing has been a major topic of artificial intelligence since its very beginning, since it is highly related to "intelligence" and has well-defined states, rules and metrics. Recent progress of deep learning and reinforcement learning has made AI for games that contains a large number of states, like the DeepMind AlphaGo, or games AI for general game playing using only pixel points, like the Atari Agent, possible. Here we are focusing on the game *Tokkun'99*, a game which requires quick reactions for human.

Tokkun'99 is a Japanese desktop game adopted from a 1980's NEC PC-8801 computer game and it was popular in early 2000's on Chinese internet.

The game's only objective is to control the airplane to dodge bullets and missiles coming from all directions for as long as possible. The player can only control the airplane to go along eight directions on a 2D map and once the airplane is hit by any of the bullet, the game ends. There are several different types of bullets: most of them move in a straight line with constant velocity, but some of them ("special bullets") can follow the player, move along a parabola, or move at high speed. And there can be from 50 to at most 200 bullets in a 320x240 game window, which makes the game very difficult for human to play.

Creating this agent is challenging, partly because the game is closed source so we need to use the raw pixels or extract the game information from game image. And the only elements in the game screen are dots of the same shape and different colors, which makes the state space very large and makes it harder for machine perception.

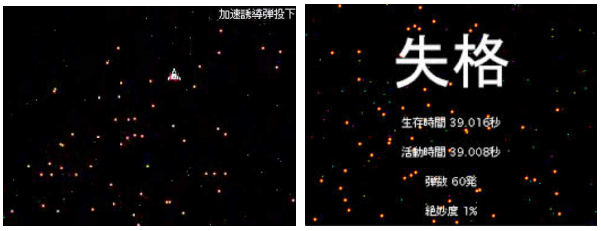


Fig. 1: The game screenshot. Left is the playing game and right is the game-over scene.

II. Related Work and Projects

Reinforcement learning is widely used in game playing. The backgammon agent TD-Gammon [1] is the first reinforcement-learning-based agent using neural networks to achieve an expert level in backgammon game playing, after which games like Minesweeper [2] and Tetris [3] were also solved in reinforcement framework.

For general and domain-independent game playing, Bellemare et al. [4] extracted some game independent features like Basic, BASS, DISCO and used SARSA to play Atari games. In 2013, Mnih et al. presented the first deep learning model to learn control policies directly from image input using a variant of reinforcement learning named "Deep Q-Learning" which achieves expert level on three of the Atari 2600 games [5] and they achieved better results in 2015 using the Deep Q Network(DQN) [6].

Several improvements are proposed to improve the DQN algorithm. The method of Double Q-learning [7], which is proposed to solve the problem of overestimations of action values for Q-learning, can be adapted to DQN algorithms to reduce the overestimation and lead to better performance on several Atari games [8].

For models with partially observable MDPs, Deep Recurrent Q-Network (DRQN) algorithm, which utilizes recurrent LSTM and replicates DQN's performance on standard Atari games, was proposed by Hausknecht et al. [9]

Similar CS 221 projects applies reinforcement learning to agent-based games like Flappy Bird, Chrome Dinosaur Game and Snake. Bojanczyk et al. [10] and Quinn et al. [11] used vanilla Q-learning to play Flappy Bird; H. Wei et al. used DQN to play Chrome Dinosaur Game [12]; Menon compared vanilla DQN and policy-based Asynchronous Advantage Actor-Critic (A3C) methods in Atari Pong game [13]; H. Xiong et al. improved the A3C method for long chain game like Atari game Tutankham [14].

III. Methods

A. MDP Modeling

The game can be considered as a Markov Decision Process (MDP), since the outcomes of the game, and the movement of other bullets are partly random and partly under the control of a decision maker.

We can define the action space as a set of 8 different directions and *noop* (no operation): {noop, up, left, down, right, up-left, up-right, down-left, down-right}, so the action space is discrete, which makes it easier to solve using reinforcement learning algorithms.

For the state space, we have different ways for feature extraction: We can use hand-crafted features, including the player's and bullets' positions, or histograms of neighboring bullets. We can also extract some general features from the game image. And we can use Convolutional Neural Network (CNN) architectures to extract some features from the game pixels by training the network. Details will be discussed in Section IV and VI.

For each step, we set the discount factor determining the importance of future rewards to $\gamma = 0.95$. And we need to define the transition probability inside the game: For each step, if the player is hit by a bullet (game over), $r = -100$; otherwise $r = 1/\text{FPS}$.

B. Game Simulation

The game is written for Windows and is closed source so it is time-consuming for each episode and it is not easy to extract the states. We implemented a game simulator in Python using Pygame Learning Environment (PLE) (See Fig. 2) which can run in headless mode and return states, rewards and RGB images for each step. The maximum number, speed and behavior of bullets are all adjustable, which makes it easier for us to evaluate our learning algorithms.

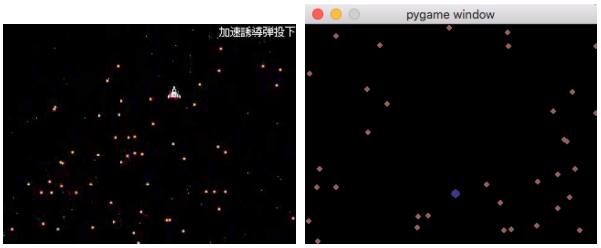


Fig. 2: Real game (left) and simulated game (right)

C. Q-Learning Framework

The objective of reinforcement learning is to maximize the expected utility function. Q-learning estimated the expected utility on an optimal policy, i.e. to maximize

$$Q_{opt}(s,a) = \sum_{s'} T(s,a,s')[r(s,a,s') + \gamma \max_{a' \in A}(Q_{opt}(s',a')]$$

The basic framework of Q learning can be described as follows:

1) Initialize Q function.
2) Choose action from current state s using function $Q(s,a)$.
   To solve the exploration-exploitation problem, when choosing actions we can use ϵ-greedy strategy:
   $$a = \begin{cases} \arg\max_{a' \in \text{Actions}} Q(s,a'), & \text{rand}() > \epsilon, \\ \text{rand}(0, \|\text{Actions}\|), & \text{otherwise.} \end{cases}$$
   or softmax strategy, where the probability of choosing action a is
   $$P(a|s,Q) = \frac{\exp(Q(s,a))}{\sum_{a'} \exp(Q(s,a'))}$$
3) Perform the action in the game and we can get a new state s' and measure our reward r.
4) Update Q function. Using the (s,a,r,s') tuple, we can update the estimation of Q function using the Bellman equation:
   $$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q_(s',a) - Q(s,a))$$
5) We can iteratively update our estimation of Q function by repeating step 2, 3 and 4.

So the two key problems are how to define our state s, i.e. how to extract game features, and how to express the Q function and learn it from game playing.

IV. Feature Extraction

A. Hand-labeled Feature

The initial idea is that we can use all the features given, including the player's position, player's velocity, and all the bullets' position and type, to build a feature vector. But the problem is the number of bullets are not fixed, and the state space is too large. Thus, we consider using some of the local information. We tried two methods for feature labeling:

1) Histograms of neighboring bullets (Fig. 3(b)): For each timestep, if the player is at (x, y), we only consider bullets that in between the range $(x \pm r, y \pm r)$. We divide the 2D range into $b \times b$ bins and count bullets in each bin, so we can get a $b \times b$-dimension vector. We refer to it as HIST(b, r).

2) K nearest Bullets (Fig. 3(c)): For each timestep, we sort all the bullets by the distance between the player and the bullet in ascending order, and keep only first K of them and use all the relative offsets $(x - x_i, y - y_i)$ as features so we can get a $2K$ dimension vector. We refer to it as KN(K).

In order to increase the importance of bullets closer to the player, for point each (x, y) in KN feature, we transform it to $(x/(x^2 + y^2), y/(x^2 + y^2))$ to make the feature value of points closer to the player much larger. We refer to this method as KN-INV(K).

In order to cover velocity information for enemy bullets, we also add the historical states into our state representation. We concatenate states of R frames together to create a state with history, which is named KN-MEM(K, R).

(a)

0	1	0	1
2	0	0	0
1	1	1	0
1	0	1	0

(b)

(15, 0)
(-20, 6)
(21, 5)
(-30, 0)
(-20, 30)

(c)

Fig. 3: Hand labeled features from game image (a)

B. General Feature from Game Image

Aside from hand-labeled features, we can also use some general features directly extracted from the game image. Bellemare et al. [4] proposed some methods to extract features from Atari games, including BASS features, Locally Sensitive Hashing(LSH) features, etc. Here we used the BASS Method to extract some general features.

BASS is the abbreviation of Basic Abstraction of the ScreenShots. This method first removes the background from the image, then maps all the color pixels into a K-palette, and divides it into a grid of $m \times n$ blocks, where for each block $B_{i,j}$ it has an K-bit vector of which each bit represents if a color exists in this block, as shown in Fig. 4.

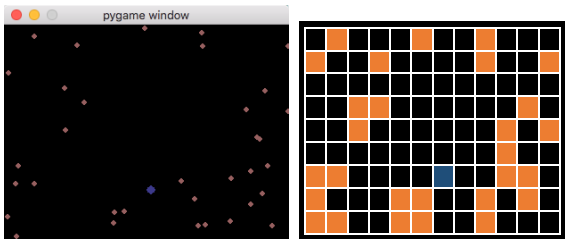


Fig. 4: Game image (left) and visualization of BASS Feature (right), from [15]

For our simulated game, it just splits the color bitmap of bullets and the color bitmaps of the player, creating an $m \times n \times 2$ dimension state vector for each state.

V. Function Approximation

When the dimension of the state space is very large for our hand-labeled features, it is impossible to enumerate over all the state space in order to implement table-lookup methods for Q-learning. So we should use function approximation, i.e. a function $Q(\theta; s, a)$, to estimate the value function $Q(s, a)$, where the θ is the parameter. To train the function approximator, we consider differentiable approximators like linear combinations of features or neural network, which are suitable for non-stationary, non-iid data.

A. Linear Function Approximation

Assume the features can be described as $\phi(s, a) = [\phi_0(s, a) \ \phi_1(s, a) \ \ldots \ \phi_N(s, a)]$, the approximated Q function will be

$$\hat{Q}(\theta; s, a) = \phi(s, a) \cdot \theta = \sum_{i=1}^{N} \theta_i \phi_i(s, a).$$

And we need to approximate the target function

$$Q^+(s, a) = R(s, a, s') + \gamma \max_{a'} Q^+(s', a'),$$

So we can define our squared error loss function as

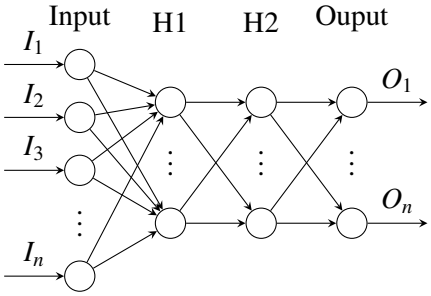
$$L(\theta) = \frac{1}{2}[Q^+(s,a) - Q(\theta; s, a)]^2$$

Thus we can use stochastic gradient descent method to update our parameters θ:

$$\begin{aligned} \theta_{\text{new}} &\leftarrow \theta - \alpha \nabla L(\theta) \\ &= \theta - \alpha \nabla \frac{1}{2}[Q^+(s,a) - Q(\theta; s, a)]^2 \\ &= \theta - \alpha[Q^+(s,a) - \phi(s,a) \cdot \theta] \cdot \phi(s,a) \end{aligned}$$

B. Neural Network

We can train neural networks to approximate the target function $Q^+(s,a)$, where the coefficients of the function decomposition are automatically obtained from the input-output data pairs. Here we tried a simple 3-layer MLP for function approximation:



Using the method proposed by Mnih et al. [6], we keep a memory of size N storing the N most recent (s, a, r, s') tuples, which is the previous experiences, and after each action taken, we draw a mini-batch of experiences from the memory to perform the update step.

VI. Deep Q Learning

The Deep Q Learning, first described in [16] and [6] by Mnih et al., is a method which can estimate Q function directly from the image pixels, which merges the feature extraction and learning process together. This method performs over human on many Atari 2600 games like Pong, Space Invaders and Breakout, in some of which (like Pinball, Boxing and Breakout) the performance level is over 10×.

A. Network

This method uses a convolutional neural network, named Deep Q Network (DQN), to estimate Q values directly from the high dimensional sensory input. For the Atari Game, the input, a preprocessed $84 \times 84 \times 4$ image, will be fed into the network. The first layer is a convolution layer containing 8×8 filters with stride 4, followed by ReLU units; the second layer is also a convolution layer containing 4×4 filters with stride 2, followed by ReLU units; the third layer is a convolution layer containing 3×3 filters with stride 1, followed by ReLU units. After three convolution layers there are two fully connected layers, producing outputs corresponding to the predicted Q-values for the individual actions for the input state, which can be used for the optimal policy selection.

B. Implementation

Historical data are used throughout the Deep Q Learning process. In state of each timestep, pixels of 4 continuous frames are included in order to extract temporal information using convolutional neural networks.

For long term learning, a problem in traditional Q learning is that experiences of consecutive frames of the same episode are very correlated, which may cause inefficient training. Both [16] and [6] use the experience replay technique, storing previous experiences, including s, a, r and s', in a fixed-size memory. For each timestep, after the agent performs the action, the oldest experience will be replaced by the latest one. Then a minibatch of M samples will be extracted from the memory and get updated.

Article [6] also improved the stability, using a separate network for generating target action-value function $Q^+(s,a)$ in update phase, and the Q^+ network synchronizes with the Q network every C timesteps. This makes the algorithm more stable for neural network approximation, since when we update $Q(s,a)$ the value of $Q(s',a)$ will also change, which can lead to oscillations and divergence. And it uses the error clipping technique, limiting the (un-squared) error term to be between -1 and 1, to improve the stability of the algorithm.

C. Asynchronous Learning

In 2016, Mnih et al. [17] proposed a reinforcement learning framework which uses asynchronous gradient to optimize the deep neural network controllers, which surpasses previous methods in both time and performance.

In Asynchronous Q-learning, all threads share a same Q network $Q(s,a;\theta)$. The only difference between asynchronous Q-learning and deep Q learning

described in Section VI is that in this method, for each thread, it keeps gradients w.r.t θ: $d\theta$, which gets updated every step and will be updated to the global parameters θ every $I_{\text{AsyncUpdate}}$ steps.

VII. Results

A. Testing parameters

For the game simulator, it runs at 15 frames per second in a 320x240 window, where there are 30 bullets in total. The agent makes decision at each frame.

For the MDP, the discount factor is fixed at 0.95 and the reward is defined as

$$r(s, a, s') = -100[\text{IsEnd}(s')] + 1[\neg\text{IsEnd}(s')]$$

For the Q-learning framework, all methods use learning rate $\alpha = 0.001$ and ϵ-greedy policy with ϵ decays from $\epsilon_{max} = 1$ to $\epsilon_{min} = 0.01$.

For the Deep Q Learning algorithm, we used a replay memory with $R = 2000$ experiences. Every time we do updates in mini-batches of 32 experiences. And we started learning and exploitation after R frames after the training begins. Our network is

B. Overall performance

1) Maximum playing time: We counted the maximum playing time (frames) to evaluate the limitation of performance of each method.

The results of our experiments are shown in Table I. For most methods except BASS, it is possible to obtain a fairly long playing time.

Feature	Linear Approx	Neural Network
HIST(25, 5)	500/500	498/500
HIST(30, 6)	500/500	499/500
KN(5)	500/500	994/1000
KN-INV(5)	1181/2000	998/1000
KN-MEM(5)	1320/2000	971/1000
BASS	355/1000	
DQN		1000/1000
Random		178/1000
Static		226/1000

TABLE I: Maximum time of the last 1000 episodes. Unit: frame (game is played in 15fps)

2) Average playing time: We used average playing time (frames) over recent 1000 episodes to evaluate the overall performance of each method.

The results of our experiments are shown in Table II, Fig 5, Fig 6, and Fig 8.

Feature	Linear Approx	Neural Network
HIST(25, 5)	89.37	61.28
HIST(30, 6)	89.34	80.50
KN(5)	85.76	90.13
KN(10)	58.55	62.98
KN-INV(5)	85.37	108.40
KN-MEM(5)	126.17	100.12
BASS	29.18	
DQN		175.10
Random		28.00
Static		31.78

TABLE II: Average playing time of the last 1000 episodes. Unit: frame (game is played in 15fps)

From the plots we can find that there are several methods which have achieved some good results. For non-DQN methods, the K-nearest features with linear function approximation has achieved the best result, which is 126.17 frames, much higher than the approaches without local history. As of learning methods, we can see from Fig 6 that neural network approximation methods usually converges slower than corresponding linear approximation methods. Though they can achieve some good results in a short time, they suffer a lot from non-convergency. Deep Q Learning converges much slower than other methods, but it achieves the best result among all the methods. The general feature — BASS Feature — doesn't work well in our experiments, the reason is partly that the dimension of BASS features is too high, making it difficult to be estimated by a linear approximator.

C. Parameter choices

1) Frame skip: Frame skip, i.e. how many frames should an action last, also affects the learning performance. We tried different frame skips by changing the game's frame rate and update at each frame. We measured the average of recent 1000 playing times using three linear approximation methods, as shown in Table III.

If the frame skip is too small, the action may not affect the agent too much before the next action; if the frame skip is too large, the agent may not move promptly so it will be more likely to be hit by bullets. Thus we finally chose 15fps as the frame rate of the game.

2) Learning rate: Learning rate affects how fast the linear model / neural network model tries to converge. If the learning rate is too low, it would take longer time to train and make the agent trapped in suboptimal

FPS	Method	Frames	Time(s)
8	KN(5)	20.13	2.52
	KN-INV(5)	18.16	2.27
	KN-MEM(5,1)	61.93	7.74
15	KN(5)	85.76	5.72
	KN-INV(5)	85.37	5.69
	KN-MEM(5,1)	126.17	8.41
30	KN(5)	52.94	1.76
	KN-INV(5)	136.21	4.54
	KN-MEM(5,1)	231.96	7.73

TABLE III: Comparison of different frame rates. Unit: frame

states. If the learning rate is too large, the weights of linear model or neural network may begin exploding.

For linear models, we tried the following learning rates for the following features, and measured the average of recent 1000 playing times after 20000 episodes. The results are shown in Table IV.

α	Method	Frames
10^{-2}	KN(5)	28.14
	KN-INV(5)	76.972
	KN-MEM(5,1)	101.78
5×10^{-3}	KN(5)	28.89
	KN-INV(5)	90.54
	KN-MEM(5,1)	85.18
10^{-3}	KN(5)	85.76
	KN-INV(5)	85.37
	KN-MEM(5,1)	126.17
5×10^{-4}	KN(5)	28.30
	KN-INV(5)	76.66
	KN-MEM(5,1)	74.19
10^{-4}	KN(5)	23.05
	KN-INV(5)	129.86
	KN-MEM(5,1)	62.80

TABLE IV: Comparison of different learning rates. Unit: frame

For nonlinear models like the 3-layer MLP and Deep Q Network, we only used $\alpha = 0.001$ as the parameter. It could be chosen using a binary search algorithm to find the best learning rate.

3) Image size: The first step of Deep Q Learning is to resize the image to the network's input size. In the original paper, they resized the game pixels to 80x80 grayscale images [16]. We tried two image sizes in our implementation: 80x80 and 120x120, and evaluated them using the average of recent 1000 playing times. The result is shown in Table V and Fig 7 .

The possible reason for the performance boost is that the bullets and the player are expressed as small dots in the screen, when the image is resized, the size of dots may decrease drastically and making it difficult to be distinguished. These dots may disappear after convolutional filters if the image is too small. But as the size of image increases, the network will correspondingly become larger, making it slower to train until convergence. But from the Figure we can see that when the image size becomes larger, there are several drastic increases and decreases when the image size is 120x120, a possible reason is that the Q function is still not stable and we need more episodes to train the network.

Size	Time
80x80	115.81
120x120	175.10

TABLE V: Comparison of different image sizes in DQN. Unit: frame

D. Comparison with human players

In our proposal, three human players has tested the game, the result of which is shown in Table VI.

	Worst(s)	Best(s)	Average(s)	Deviation(s)
Oracle 1	2.064	22.064	9.103	21.926
Oracle 2	1.984	10.400	5.225	2.456
Oracle 3	2.080	9.568	4.243	1.998

TABLE VI: Human players' results for *Tokkun'99*

Currently for the Deep Q Learning method, our method can achieve over 1000 frames (over 66 seconds) maximum and 175 frames (11.67 seconds) on average, which performs better than the human players. But the deviation of the agent is far larger than human players since it converges very slowly when training the network.

VIII. Conclusion

We implemented a game agent of the game *Tokkun'99* using reinforcement learning algorithms, including linear and neural network approximated Q-learning with different feature extractors, and Deep Q Learning which calculates $Q(s, a)$ function values directly from game pixels. For simple games (without different types of bullets, with only a small number of bullets) the agent outperforms the human player. Our results show that reinforcement learning, especially deep reinforcement learning, has great potential for further application in game playing.

References

[1] G. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[2] L. P. Castillo and S. Wrobel, "Learning minesweeper with multirelational learning," in *Proceedings of IJCAI-03*, 2003, pp. 533–540.

[3] I. Szita and A. Lörincz, "Learning tetris using the noisy cross-entropy method," *Neural Computation*, vol. 18, no. 12, pp. 2936–2941, 2006.

[4] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *J. Artif. Intell. Res.*, vol. 47, pp. 253–279, 2013.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, and A. A. Rusu, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[7] H. van Hasselt, "Double Q-learning," in *Proceedings of NIPS 2010*, 2010, pp. 2613–2621.

[8] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, 2016, pp. 2094–2100.

[9] M. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," in *AAAI Fall Symposium on Sequential Decision Making for Intelligent Agents (AAAI-SDMIA15)*, Arlington, Virginia, USA, November 2015.

[10] K. Bojanczyk, Z. Hu, and C. Xu, "Fly, flappy, fly: A q-learning decision system for obstacle detection and avoidance," *CS 221 Project*, 2016.

[11] G. F. Reis and M.-A. Quinn, "Automatic flappy bird player," *CS 221 Project*, 2016.

[12] H. Wei, J. Ke, and Y. Zhao, "Ai for chrome offline dinosaur game," *CS 221 Project*, 2016.

[13] H. Menon, "Learning to play pong," *CS 221 Project*, 2016.

[14] H. Xiong, Y. Ma, and C. Tian, "Deep reinforcement learning for atari game tutankham," *CS 221 Project*, 2016.

[15] Y. Naddaf, "Game-independent ai agents for playing atari 2600 console games," 2010.

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," in *NIPS Deep Learning Workshop*, 2013.

[17] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning*, 2016, pp. 1928–1937.

Appendix

For all the plots, the X axis is #episode/10, and the Y axis is the number of frames (15fps).

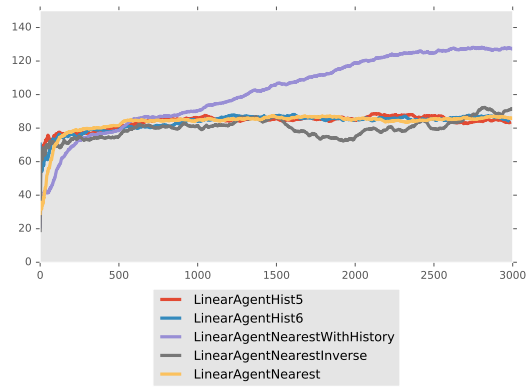


Fig. 5: Comparison of different feature types.

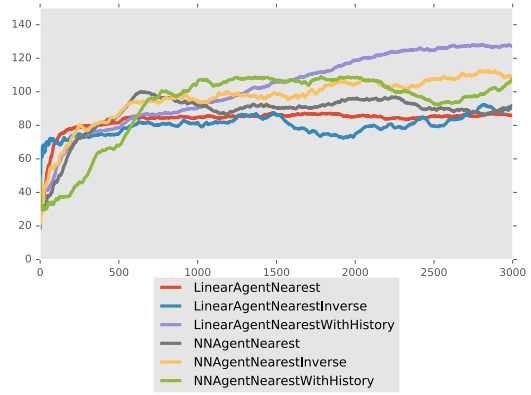


Fig. 6: Comparison of different approximation methods on K-nearest features.

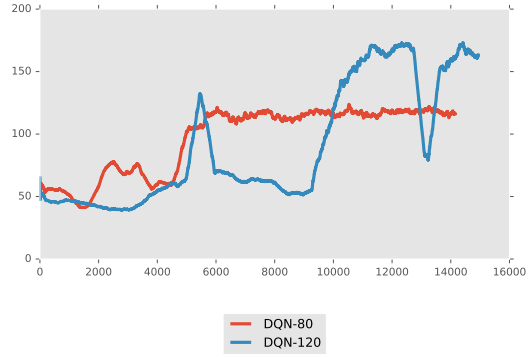


Fig. 7: Comparison of image size of DQN.

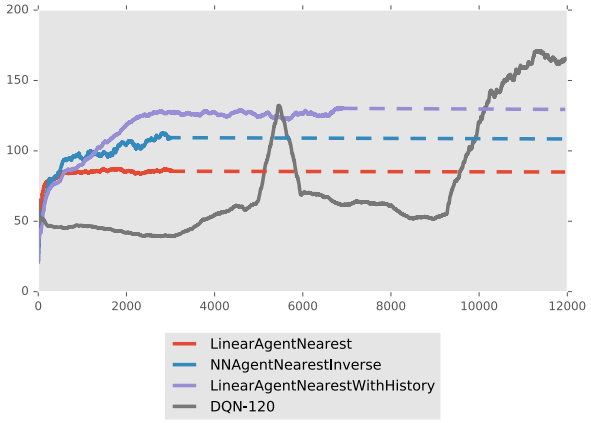


Fig. 8: Comparison of the top 4 scored methods.