2018-8-27

Speeding-up Deep Reinforcement Learning

A report

邹琮 Cong Zou

zouc15@mails.tsinghua.edu.cn

Abstract

Reinforcement learning is an important machine learning method. It has many applications in the fields of intelligent control robots, analysis and prediction. Currently, deep reinforcement learning is able to solve problems that were previously intractable, such as learning to play video games directly from pixels. However, problem still exists. It always takes a long time to converge. To tackle this problem, I mainly start from two ways: Using deep reinforcement learning methods with higher convergence speed and decomposing neural network with tensor train (TT) decomposition method.

I evaluate these approaches on a video game called flappy bird to show if they can gain higher scores with shorter time.

Table of contents

1 Introduction

In 2013, DeepMind published the article "Playing Atari with Deep Reinforcement Learning"[1] on NIPS, and proposed the DQN (Deep Q Network) algorithm to realize end-to-end learning to play Atari games, that is, learning to play games directly from pixels.

Flappy Bird is an extremely simple and difficult game that is all the rage. A long time ago, someone has achieved to play Flappy Bird using Q-learning. However, it is achieved by getting the specific location information of the bird. [2] finally achieved to implement Nature DQN on it.

According to [2], it first preprocessed the game screens with following steps:

1) Convert image to grayscale
2) Resize image to 80x80
3) Stack last 4 frames to produce an 80x80x4 input array for network

The architecture of the network is shown in the figure below. The first layer convolves the input image with an 8x8x4x32 kernel at a stride of 4. The output is then put through a 2x2 max pooling layer. The second layer convolves with a 4x4x32x64 kernel at a stride of 2. The third layer convolves with a 3x3x64x64 kernel at a stride of 1. The last hidden layer consists of 1600 fully connected nodes. The final output layer has the same dimension as the number of actions. The values of this output layer represent the Q function given the input state for each action.

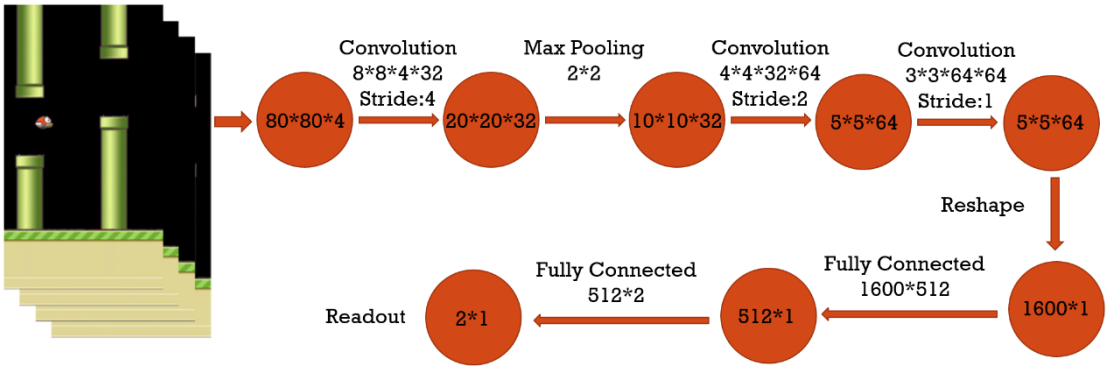


Fig.1. Architecture of the network

However, the problem is that it is too difficult for this network to converge and it will cost a long time training this bird get a high score. The figure below shows the score the bird gained after I have trained fifty thousand time steps. The horizontal axis is the score and the vertical axis is the number of times the score is obtained. The percent of score zero is 0.88889 and the percent of score one is only 0.11111.

Therefore, it is necessary to find some methods to improve its learning speed. I mainly start from two directions. The first is using more efficient deep reinforcement learning methods to improve the convergence speed, and the second is implementing tensor train decomposition on the network to shorten the time spent on each time step.

The report is organized as follows. In Section 2 some more efficient approaches I have

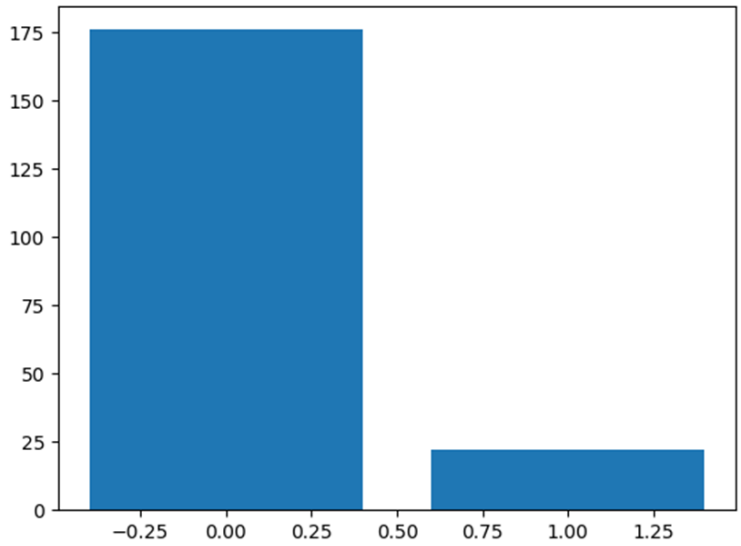


Fig.2. Nature DQN

applied to the flappy bird game are presented, such as Double DQN, Prioritized Experience Replay (PER), Dueling DQN, Asynchronous Advantage Actor-Critic (A3C) and Distributed Proximal Policy Optimization (DPPO). And also In Section 2 it is showed how to decompose the FC layer, the Convolutional layer and finally the whole neural network. I conclude with the summary and the discussion in Section 3.

2 Methods and Results

Overall, my methods can be divided into two modules: (1) using more efficient methods to improve the convergence speed, (2) using tensor train decomposition on the network to shorten the time spent on each time step. Below, I will review the details of each methods and then show how to implement them on the flappy bird game. Finally I will presented the performance of each methods.

2.1 DRL with Higher Convergence Speed

2.1.1 Double DQN

Thrun & Schwartz (1993) pointed out that there was an overestimation problem in Q-learning. Because DQN is essentially based on Q-learning, there must be an overestimation problem in DQN, and it will affect the performance of DQN to some extent. [3] proposed Double DQN, which is aimed to solving this overestimation problem.

The improvement of double DQN is the method of calculating Q target. There are two neural networks in DQN, just as the figure 3 shows below. One for calculating evaluation value and another for calculating target value. And DQN only use the target neural network to estimate the Q target, so the single estimator update rule may overestimate the expected return due to the fuse of the maximum action value as an approximation of the maximum expected action value. But in double DQN, the next action is chosen by the evaluation network and the next Q value is evaluated by the target network.

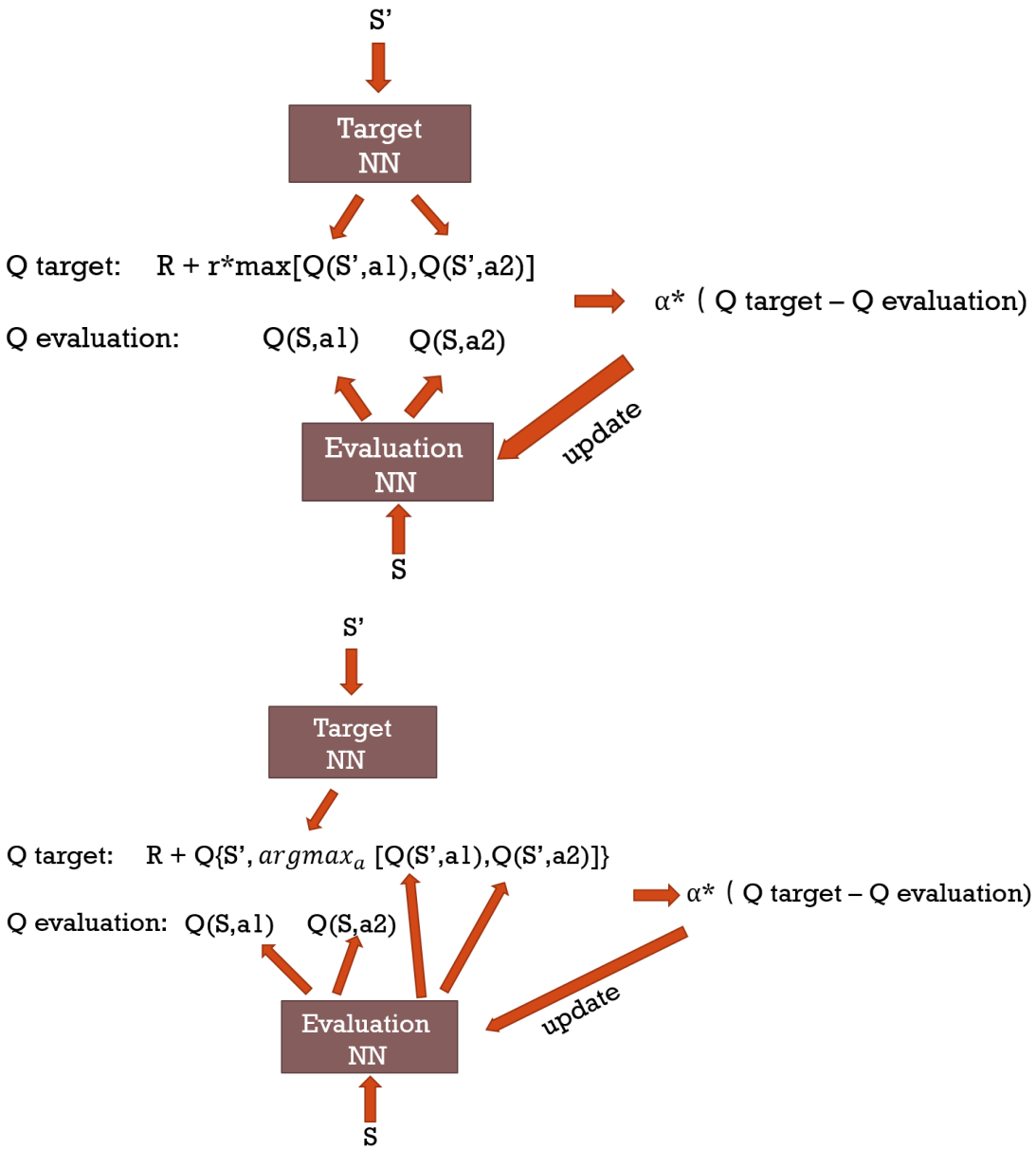


Fig.3. the structure of Nature DQN (top)
and the structure of Double DQN (bottom)

Double DQN has significantly improved the convergence speed so that the scores the Flappy Bird gained are much higher after the same time steps as Nature DQN.

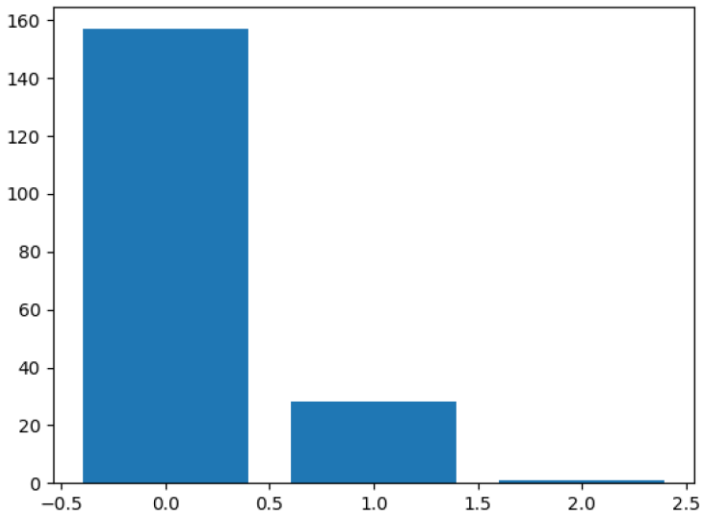


Fig.4. Double DQN

The figure above shows the score the bird gained after I have trained fifty thousand time steps. The percent of score zero is 0.84409, the percent of score one is 0.15054 and the percent of score two is 0.00538. Compared with Nature DQN, it increased by 4.48%

2.1.2 Double PER DQN

Experience Replay used in Nature DQN stores experience in a playback unit and updates by mixing more or less recent experience, which may disrupt temporal correlation, and the rare experience will be used for more than one update. However, this method simply plays back transitions at the same frequency, regardless of their meaning. [4] proposes a method that enables priority playback and enables more frequent playback of more important transitions, which can lead to more efficient learning.

The biggest improvement of this algorithm is that memory batch sampling is not a random sampling; it is based on the sample priority in memory. Therefore, it can be more effective to find the samples we need to learn. The definition of priority is based on TD-error. The bigger the TD-error, the larger room for improvement in our prediction accuracy, which means the higher the priority p. The final step is to sample transitions according to the priority p. However, if each sample needs to sort all experience according to p, it will result in very high computational complexity. So that a method named Sum Tree (where every node is the sum of its children, with the priorities as the leaf nodes) is mentioned in this paper, which can be efficiently updated and sampled from.

In my Flappy Bird game, the rule is that if the bird fly through the pipe, the reward will be positive ten. If the bird crash into the pipe or the upper and lower edges, the reward will be negative one. So in the early stage of training, there will be very few +10 reward in the memory. But the PER method I introduce above will value these few, but worth learning samples.

I combined Double DQN with PER, and the figure above shows the scores the bird gained after I have trained fifty thousand time steps. The percent of score zero is 0.61875; the percent of score one is only 0.36875 and the percent of score two is 0.01250. Compared with Double DQN, it increased by 22.53%

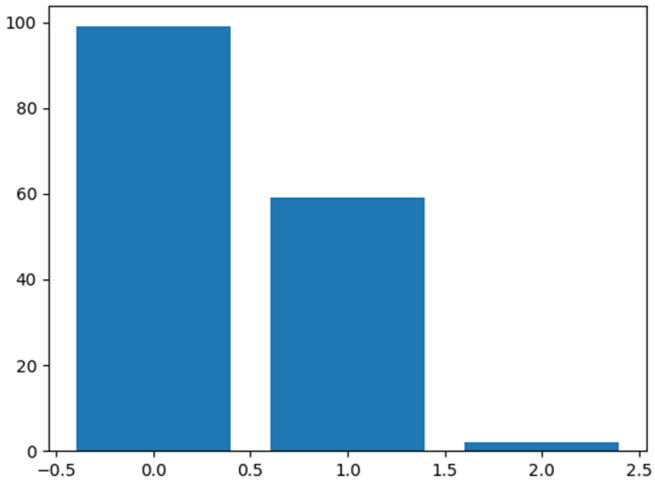


Fig.5. Double PER DQN

2.1.3 Double PER Dueling DQN

In many DRL tasks, the Q-value functions of different state action pairs are different, but in some states, the value of the Q-value function is independent of the action. For example, in my Flappy Bird game, when the bird is far from the pipe as the left figure shows, the advantage stream does not pay much attention to visual input because its action choice is practically irrelevant to the Q-value function. However, in the right figure, the advantage stream pays attentions as there is a pipe immediately in front, making its choice of action very relevant.

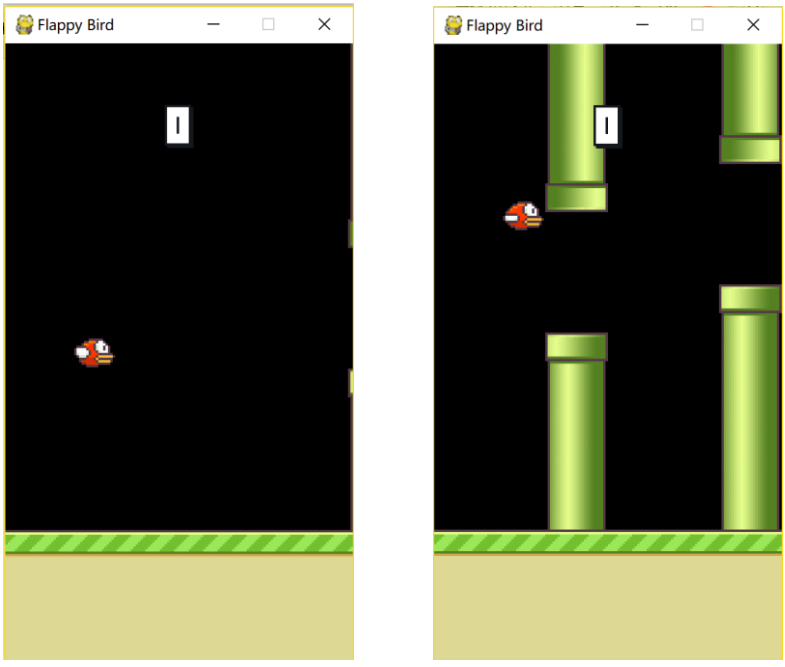


Fig.6. Flappy Bird

Based on the above ideas, Wang et al.[5] proposed a dueling network as the network model of DQN. The improvement of dueling DQN can be illustrated by the figure below. The dueling network has 2 streams to separately estimate (scalar) state-value and the advantages for each action; the green output module implements equation (1) to combine them. Both networks output Q-values for each action.

$$\mathrm{Q}(\mathrm{s},\mathrm{a};\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|A|}\sum_{a'} A(s,a';\theta,\alpha)\right) \quad (1)$$

It benefits from a single baseline and easier-to-learn relatives value rather than having to calculate the accurate Q values for all actions.

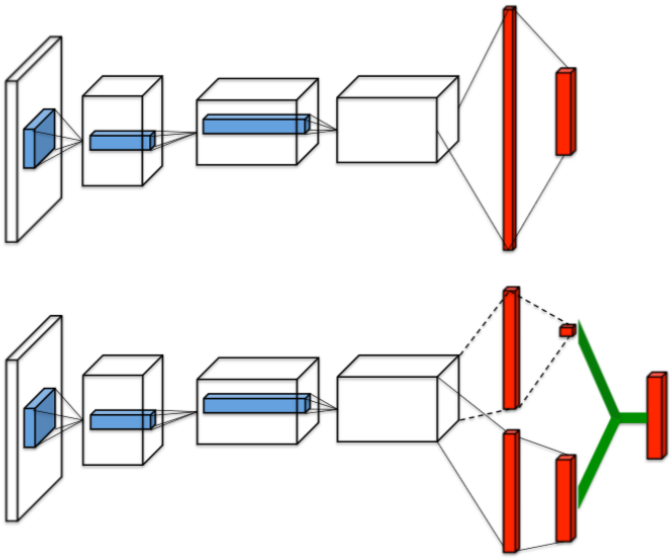


Fig.7. A popular single stream Q-network (top)
and the dueling Q-network (bottom).

I combined Double DQN and PER with Dueling DQN, and the figure above shows the scores the bird gained after I have trained fifty thousand time steps. The percent of score zero is 0.54545 and the percent of score one is only 0.45454. Compared with Nature DQN, it increased by 7.33%

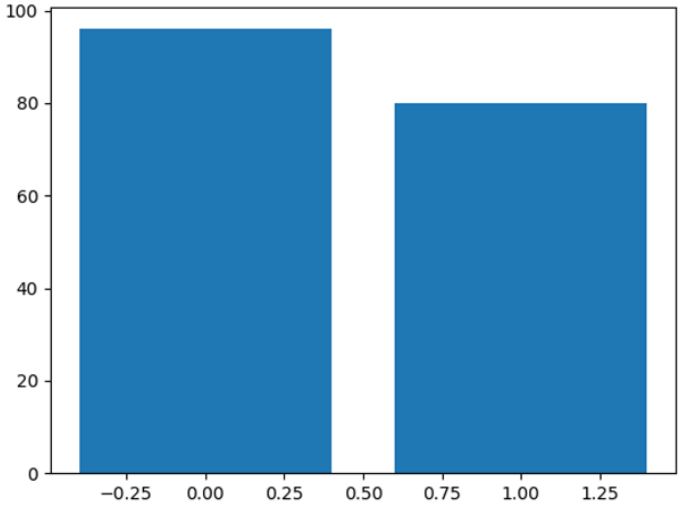


Fig.8. Double PER Dueling DQN

2.1.4 A3C

Actor-critic is a method that combines the advantages of the two major branches of RL: value-based method and policy-based method. Actors choose behavior based on probability. And Critic judges the score of behavior based on the behavior chosen by the Actor. Then Actor modifies the probability of behavior based on Critic's score.
But unfortunately, Actor-Critic involves two neural networks, and each time the parameters are updated in a continuous state, so that there is always a correlation before and after each parameter update, which causes the neural network to treat the problem unilaterally.
To solve this problem, [6] proposes a method named Asynchronous Advantage Actor-Critic (A3C). The main idea of it is running several agents in parallel, each with its own copy of the environment, and updating the parameters of the shared model together. There several advantages of these parallel agents:

1) Different agents will likely experience different states and transitions, thus avoiding the correlation
2) This approach needs less memory because it doesn't need to store the samples
3) This approach needs less time because of the parallel agents

There are two improvements of this method. The first is the asynchronous update, which can be illustrated by the figure below. Each thread has its own agent, each step generates the gradient of parameters, and the shared parameters will be updated after a certain number of steps. The different agents runs on different CPU threads of a single machine so that it can save much more time.

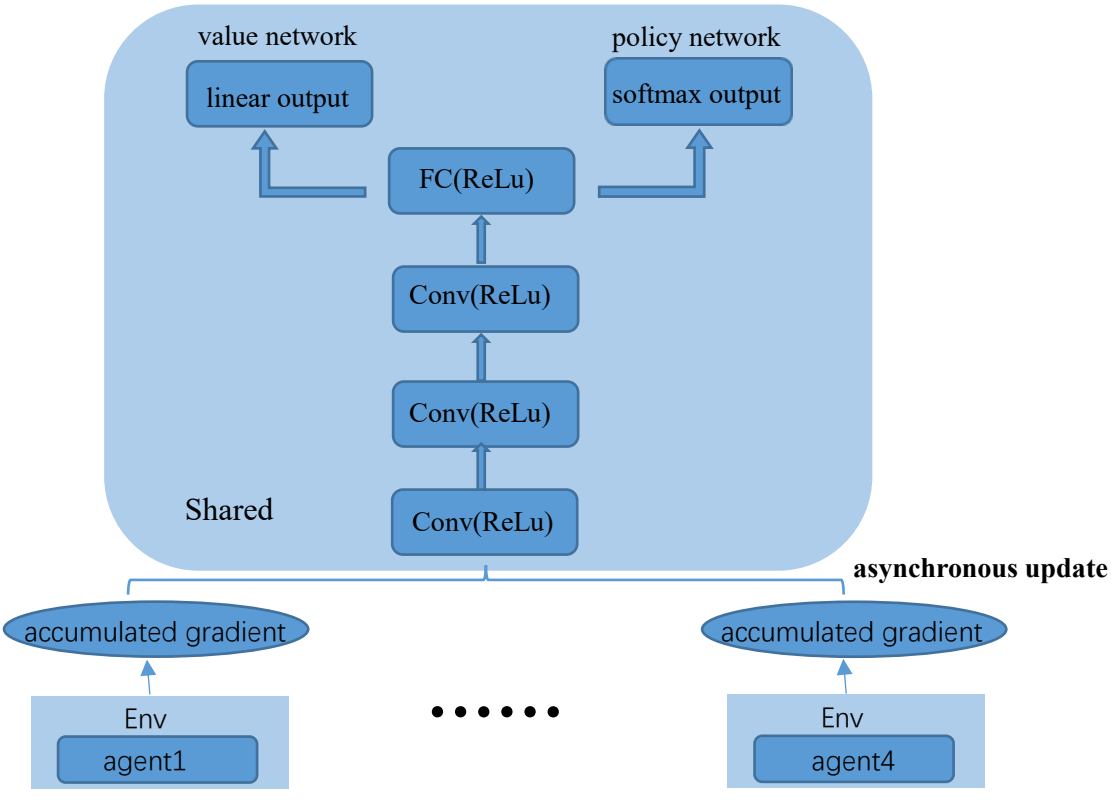


Fig.9. the structure of A3C network

The second is the N-step return. For the Nature DQN, the return is only one step like equation (2); but A3C uses a N-step return as equation (3).

$$V(s_0) \rightarrow r_0 + \gamma V(s_1) \quad (2)$$
$$V(s_0) \rightarrow r_0 + \gamma r1 + \cdots + \gamma^n V(s_n) \quad (3)$$

The advantage is that the change is propagated n steps backwards each iteration, which will improve the training speed.
The figure below shows the scores the bird gained after I had trained the same time as the Nature DQN network. The percent of score zero is 0.53205; the percent of score one is 0.42308; the percent of score two is 0.03846 and the percent of score three is 0.00641. Compared with Nature DQN, it increased by 35.68%

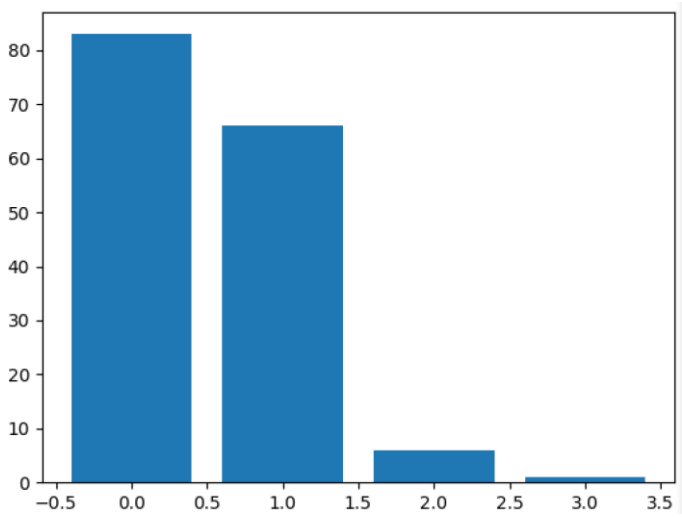


Fig.10. A3C

The A3C method is based on the Actor-Critic method, which is not as efficient as the Double PER Dueling DQN method proposed above. So if the parallel agents and the asynchronous update can be implemented on the Double PER Dueling DQN, I think there will be a huge acceleration because of the multi-threaded simultaneous training.

2.1.5 PPO

For A3C and most Policy gradient methods, the most commonly used gradient estimator has the form in equation (4) which is very commonly used but often leads to destructively large policy updates and make it difficult to converge.

$$L^{PG}(\theta) = \hat{E}_t\left[log\pi_\theta(a_t|s_t)\hat{A}_t\right] \quad (4)$$

Where $\hat{A}_t$ is the advantage function.
But the trust region method implements equation (5) to maximize objective function

subject to a constraint on the size of the policy update, and it modifies the new policy based on old policy.

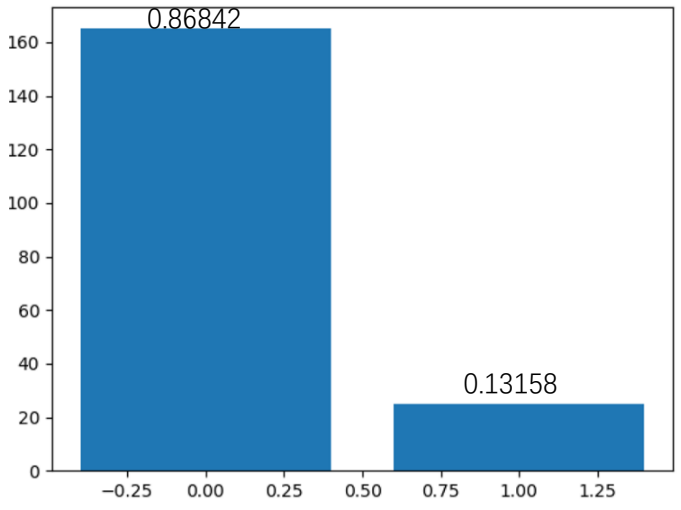
$$maximize_\theta \ \hat{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right]$$

$$subject\ \ to\ \ \hat{E}_t\left[KL[\pi_{\theta_{old}}(\cdot\,|s_t), \pi_\theta(\cdot\,|s_t)]\right] \le \delta \qquad (5)$$

To solve this problem with a first-order algorithm, the OpenAI team proposed a method named Proximal Policy Optimization Algorithms (PPO) in [7]. When updating the actor network, the paper explains an approach named Clipped Surrogate Objective. The main objective they proposed is the following equation: r is the probability ratio and this term removes the incentive for moving r_t outside of the interval.

$$L^{CLIP}(\theta) = \hat{E}_t\left[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right] \qquad (6)$$

Then the final objective is a lower bound on the unclipped objective. This method not only avoids an excessively large policy update but also emulates the monotonic improvement of Trust Region method and is much simpler to implement.
I trained the A3C network and the A3C + Trust Region network 2 hours respectively. The figure below shows the results of this two networks, and there is a slightly improvement with trust region method.



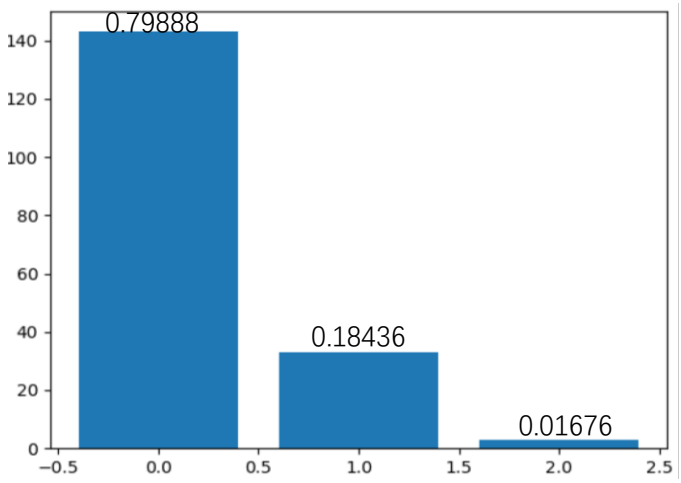


Fig.11. A3C(top) and A3C+Trust Region (bottom)

2.2 Tensor Train Decomposition

2.2.1 Decompose FC Layer

The fully-connected layer consists in a linear transformation of a high-dimensional input signal to a high-dimensional output signal with a large dense matrix defining the transformation. So when the input and output dimensions are high, the dense matrix will be very large, which will result in high computational complexity and large storage space. To save time and memory, [8] proposes to use Tensor-Train to represent the dense weight matrix of the fully-connected layers using few parameters while keeping enough flexibility to perform signal transformations.

Firstly, I will review Tensor Train (TT) format. A d-dimensional tensor W is said to be represented in the TT-format if all the elements of W can be computed as the following matrix product: All the matrices $G_k[j_k]$ related to the same dimension k are restricted to be of the same size $r_{k-2} \times r_{k-1}$. The values r_0 and r_d equal one in order to keep the matrix product of size 1×1

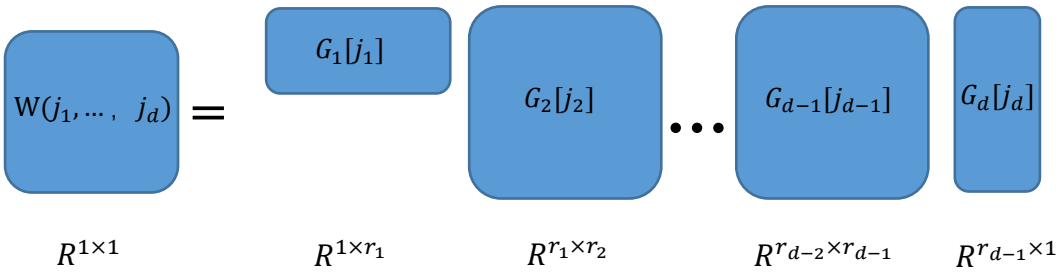


Fig.12. TT-Format

Therefore, the number of elements required to store reduces from $\prod_{k=1}^{d} n_k$ to $\prod_{k=1}^{d} n_k\, r_{k-1} r_k$. Thus, the TT-format is very efficient in terms of memory if the ranks are small.

To be able to efficiently work with large vectors $\mathrm{b} \in R^N$ and matrices $\mathrm{W} \in R^{M \times N}$, we can decompose $\mathrm{N} = \prod_{k=1}^{d} n_k$ and $\mathrm{M} = \prod_{k=1}^{d} m_k$ to reshape the vector and matrix to tensor, then use the Tensor Train decomposition. In this way, the computational complexity of an M × N fully-connected layer (FC) will reduce from O(MN) to $\mathrm{O}(\mathrm{d}r^2 m \cdot \max\{M, N\})$ where $\mathrm{m} = max_{k=1\ldots d} m_k$ and r is the maximal TT-rank, which can be illustrated by the following equation and figure 13.

$$\mathrm{y}(i_1, \ldots, i_d) = \sum_{j_1, \ldots, j_d} G_1[i_1, j_1] \ldots G_d[i_d, j_d] X(j_1, \ldots, j_d) \qquad (7)$$

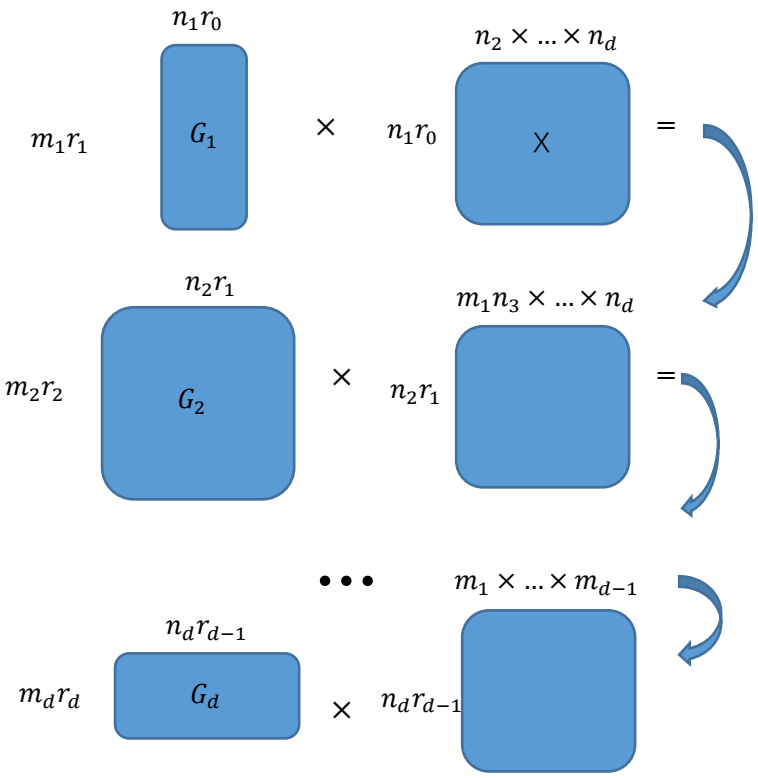


Fig.13 the process of multiplication of tt-matrix and vector

Because the acceleration with Tensor Train layer will be more obvious, I changed the architecture of the network as the figure below. The first convolution layer's stride is changed from four to two and the size of the fully connected layer is changed to 6400*1600, and then I decompose this FC layer in the network of Nature DQN.

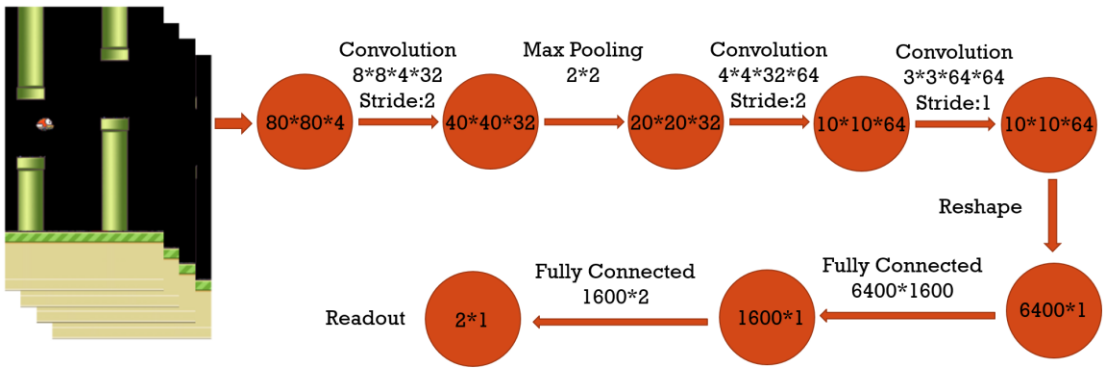


Fig.14. the architecture of the changed network

I decompose the row and column dimension into six factors respectively and fix the tt rank as 2. So that the TT cores are just like figure 15. The number of elements used to store is reduced from 10240000 to 326, and the compression ratio is 0.000032. And the time consumed after 10000 time steps is reduced from 4940.94s to 4213.24s. Actual acceleration is not as big as theoretical analysis, which may be caused by too many hyper parameters, such as the value of TT ranks and how MN is decomposed. These hyper parameters also affect the convergence speed. If the rank is too small, the

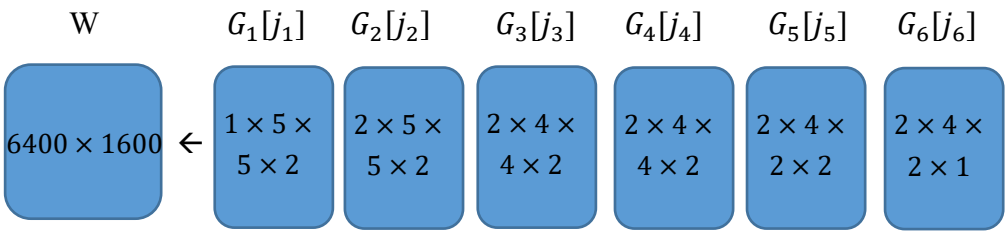


Fig.15. TT cores of the dense matrix W

convergence speed may be slowed down. Therefore, there is a contradiction here: these hyper parameters need to be carefully adjusted to make both the spent time of each time step and the convergence speed better.

2.2.2 Decompose Convolutional Lay

Convolutional layer is also a really important part of my Flappy Bird neural network. However, it also requires millions of floating point operations to process an image, contains millions of trainable parameters, and consumes hundreds of megabytes of storage. To address the storage and memory requirements of neural networks, [9] proposes a tensor factorization based method to compress convolutional layers: it reshapes the 4-dimensional kernel of a convolution into a multidimensional tensor to fully utilize the compression power of the Tensor Train decomposition.

The convolutional layer transforms the 3-dimensional input tensor X into the output tensor Y by convolving X with the kernel tensor K by the following equation:

$$\mathrm{Y}(\mathrm{x},\mathrm{y},\mathrm{s}) = \sum_{i=1}^{l}\sum_{j=1}^{l}\sum_{c=1}^{C} K(i,j,c,s)X(x+i-1, y+j-1, c) \qquad (8)$$

To improve the computational performance, it reformulates convolution as a matrix-by-matrix multiplication $\mathrm{Y} = \mathrm{XK}$. The specific method can be illustrated by the figure below:

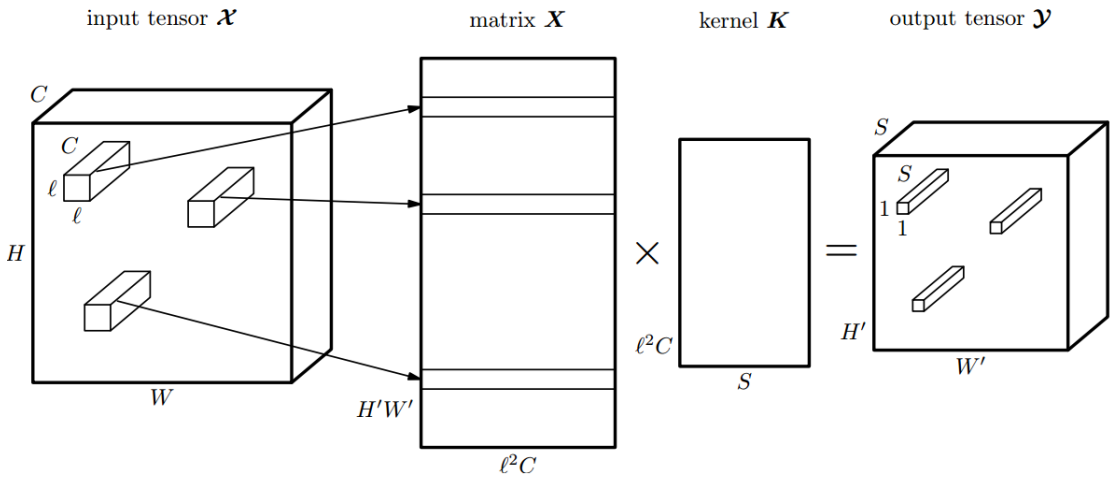


Fig.16. Reducing convolution to a matrix-by-matrix multiplication

Firstly, reshape the output tensor $\mathcal{Y} \in R^{W\prime\times H\prime\times S}$ into a matrix Y of size $W'H' \times S$ in the following way

$$\mathcal{Y}(\mathrm{x},\mathrm{y},\mathrm{s}) = \mathrm{Y}(\mathrm{x} + \mathrm{W}'(\mathrm{y}-1),\mathrm{s}) \qquad (9)$$

Where H' = H - ℓ + 1 and W' = W - ℓ + 1. Then the matrix X of size $W'H' \times \ell^2 C$ can be obtained by the following equation, the k-th row of which corresponds to the $\ell \times \ell \times C$ patch of the input tensor that is used to compute the k-th row of the matrix Y

$$\mathcal{X}(\mathrm{x}+\mathrm{i}-1,\mathrm{y}+\mathrm{j}-1,\mathrm{c}) = \mathrm{X}\big(\mathrm{x}+\mathrm{W}'(\mathrm{y}-1),\mathrm{i}+\ell(\mathrm{j}-1)+\ell^2(c-1)\big) \qquad (10)$$

Finally, the kernel tensor $\mathcal{K}$ can be reshaped into a matrix K of size $\ell^2 C \times S$:

$$\mathcal{K}(\mathrm{i},\mathrm{j},\mathrm{c},\mathrm{s}) = \mathrm{K}(\mathrm{i}+\ell(\mathrm{j}-1)+\ell^2(c-1),s) \qquad (11)$$

After getting the matrix-by-matrix multiplication format of convolution, the matrix TT-format can be applied to the matrix K, i.e. reshape it into a tensor K and convert it into the TT-format, which can just be treated as FC layer. Then it can transform the input tensor X into the output tensor Y.

I used this method to decompose the second and third convolution layers. The process can be illustrated by the following figure, the TT cores of the second conv layer are $(1 \times 4 \times 4 \times 10), (10 \times 4 \times 4 \times 10), (10 \times 2 \times 4 \times 1)$ and the TT cores of the third layer are $(1 \times 4 \times 4 \times 10), (10 \times 4 \times 4 \times 10), (10 \times 4 \times 4 \times 1)$.

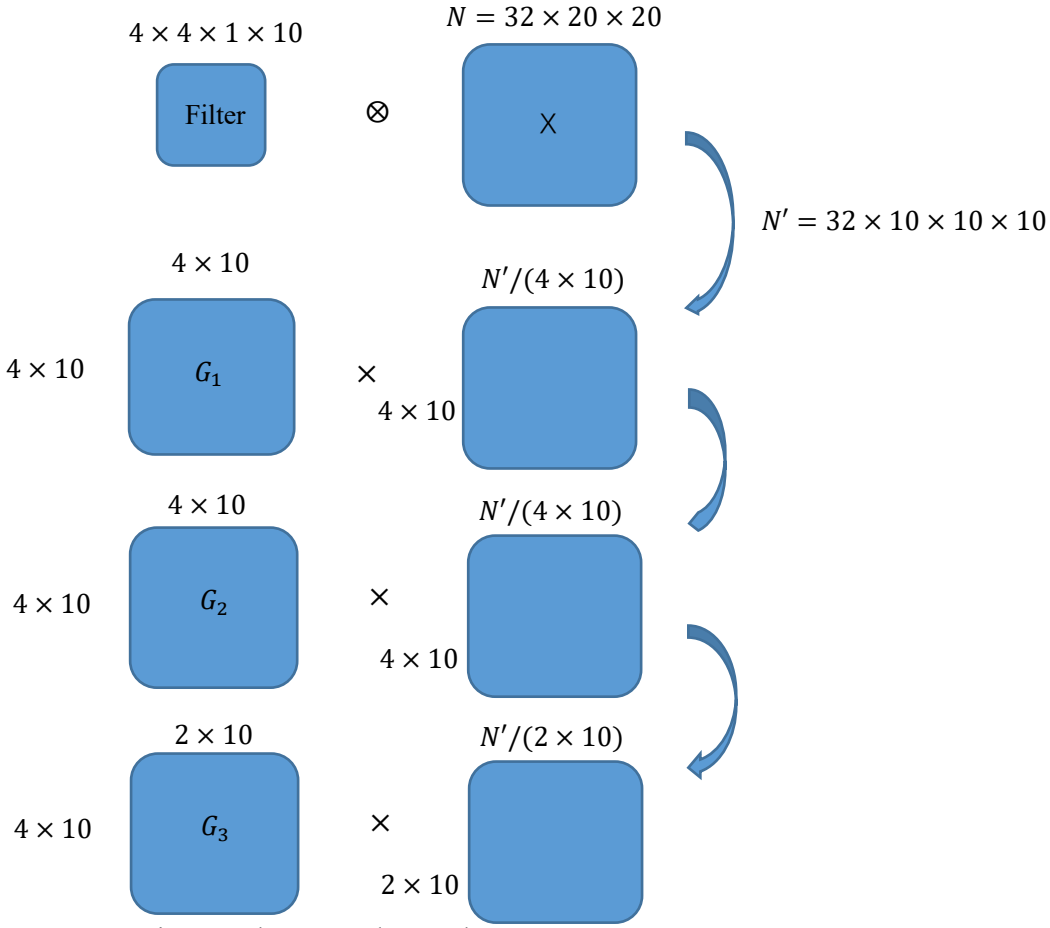


Fig.17. The second conv layer

The compression rate of these two convolution layers is 0.054, but it's a pity that I have changed the value of rank several times and found that this method can not save time.
I think there is a another method to use TT format in the convolution layer, that is to convert the convolution operation from the time domain to the frequency domain, then it will become matrix multiplication, which can be directly seemed as a FC layer and is convenient to use the TT format. But I have not yet implemented it, and I think the challenge is doing Fourier transform and inverse Fourier transform of TT tensor, which will also produce high computational complexity.

2.2.3 Decompose the Whole Network

I think one of the reasons for why the two methods above unable to achieve substantial acceleration is that they both need to decompose the tensors to Tensor Train, reshape the tensors while doing the multiplication to obtain a full tensor in the end. So in my opinion, if I can decompose the whole network, which means that I just use Tensor Train to train all of the network without convert between Tensor Train and full tensor, then it may save much more time.
To solve this problem, the main challenge I met is to make Tensor Train go through the nonlinear ReLu layer, but Tensor Train can only be directly used in linear layer such as FC layer, convolutional layer and recurrent layer.
I have tried two methods. The first is to convert the TT-tensor back to the full tensor and make it go through the ReLu layer, then convert it back to TT-tensor again, which can be illustrated be the figure bellow.

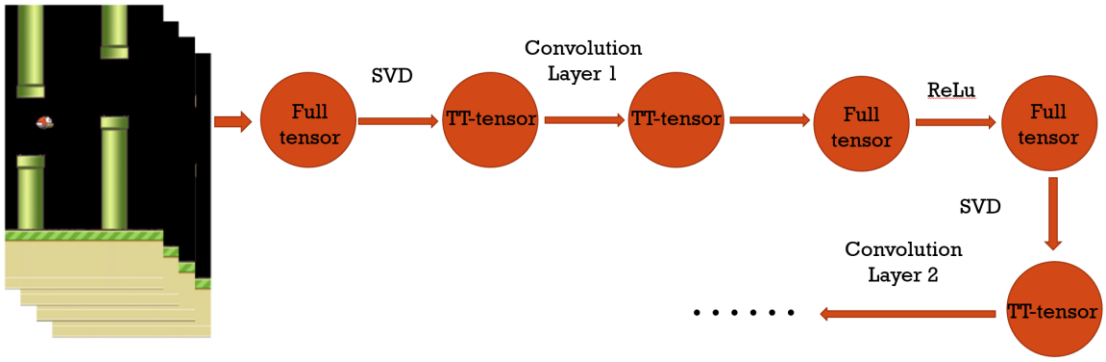


Fig.18. Method 1

However, the problem of this method is that every time converting the full tensor to TT-tensor needs SVD decomposition with high computation complexity, which cannot achieve the purpose of acceleration. Moreover, SVD will make the backpropagation process very complicated, so it is not an ideal method.
The second method is trying to obtain a ReLu function for TT-tensor and the output will still be a TT-tensor, which can be illustrated by the following figure:

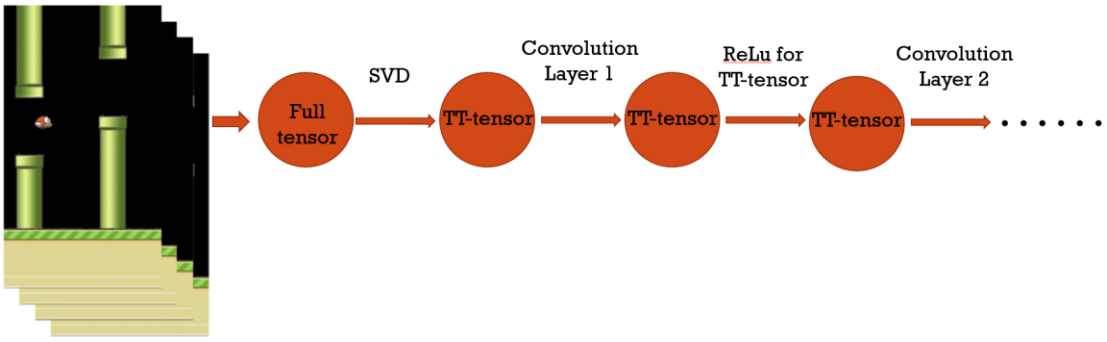


Fig.19. Method 2

The core of this method is TT-cross, which is a fast interpolation method to approximate the given tensor by using only a small number of its elements. There are several TT-cross methods and I have tied two of them, which are DMRG[11] and AMEN[14]. Next, I will introduce DMRG in detail. AMEN is similar to it, so I will not repeat to introduce it.

DMRG is mainly based on the Density Matrix Renormalization Group (DMRG) scheme and the Maximum-volume principle. Before introduce DMRG scheme, it's necessary to explain ALS first.

Given a current approximation of the following form

$$\mathrm{A}(i_1, \ldots, i_d) \approx \mathrm{B}(i_1, \ldots, i_d) = B_1(i_1) \ldots B_d(i_d) \qquad (12)$$

ALS fixes all cores except the one with the number k, so that the least squares problem $||A - B||_F \to min$ is then reduced to the linear system with $r_{k-1} n_k r_k$ unknowns. However, there is a disadvantage of ALS, which is the requirement to specify all the TT ranks in advance. So the DMRG method is proposed, the optimization of which is performed over two cores B_k, B_{k+1} simultaneously, so that a new supercore is introduced as follows:

$$W_k(i_k, i_{k+1}) = B_k(i_k) B_{k+1}(i_{k+1}) \qquad (13)$$

The advantage of DMRG is that it removes the information about r_k, so that ranks can be determined adaptively according to the desired accuracy.

In fact the k-th step of the DMRG is equivalent to the ALS step, in addition to the dimension is reduced by one. Thus all formulas can be derived in terms of an ALS step, and then plugged into the DMRG scheme. The approximation problem is then a linear least squares problem for B_k, and the process can be explained by the figure bellow:

$$||A - B||_F \to min$$

$$\mathrm{U}(i_1, \ldots, i_{k-1}) B_k(i_k) \mathrm{V}(i_{k+1}, \ldots, i_d) \approx \mathrm{A}(i_1, \ldots, i_d)$$

$\mathrm{N} \times r_{k-1}$ matrix $\qquad$ $r_{k+1} \times \mathrm{M}$ matrix

$n_1 \times \ldots \times n_{k-1} \qquad n_{k+1} \times \ldots \times n_d$

Fig.20. DMRG

The matrix U and V can be kept orthogonal, so that the core B_k can be obtained by

$$B_k(i_k) = U_k^T A_k(i_k) V_k^T \qquad (14)$$

where an N × M matrix $A_k(i_k)$ in position (I,J) has element $A(I, i_k, J)$. But the standard DMRG/ALS algorithm requires all elements of tensor to be computed. For high dimensions it is not possible. Thus, some alternative has to be presented, that is the cross approximation techniques, which interpolate a given low-rank matrix from a small number of so-called crosses.

$$B_k(i_k) = \hat{U}_k^T \hat{A}_k(i_k) \hat{V}_k^T \qquad (15)$$

Now B_k can be a matrix with smaller size p × q. If an adequate index set I with p elements, and J with q elements are found, then in the exact case B_k can be recovered by the above formula, where $\hat{U} = U(:, I), \hat{V} = V(J, :), \hat{A} = A(I, J)$. The method to find adequate index set is just Maximum-volume principle.

Thus, in summary, the approximation TT-tensor $\mathrm{B}(i_1, \ldots, i_d) = B_1(i_1) \ldots B_d(i_d)$ can be computed only by part of the elements of the full tensor $\mathrm{A}(i_1, \ldots, i_d)$. In theory, this method will have a faster speed.

However, unfortunately, to get the desired accuracy, there are two things to do. Firstly, I need to iteratively find the adequate index set by the Maximum-volume principle, which will take a long time. Secondly, because of the nonlinear of ReLu, the ranks of the approximation TT-tensor will become larger, which will lead to high computation complexity. So in practice, this method cannot speed up also, it is even slower than the first method.

I tried these two methods on the first FC layer and computed the time of their forward propagation respectively. Following is my experiment results.

Method1

TIME	SIZE
0.005s	**51200**

Method2_DMRG

RANK	ERROR	TIME	SIZE
[1,12,102,64,8,2,1]	**6.12e-15**	**5.72s**	**68748**
[1,12,62,62,8,2,1]	**0.1869**	**27.03s**	**42348**
[1,12,22,22,8,2,1]	**0.4517**	**9.92s**	**8108**

Method2_AMEN

RANK	ERROR	TIME	SIZE
[1,10,100,64,8,2,1]	**2.21e-15**	**10.59s**	**65464**

[1,12,55,55,8,2,1]	**0.3531**	**4.18s**	**34508**
[1,12,20,20,8,2,1]	**0.6171**	**1.51s**	**7068**

The time spent on method2 is much longer than that on method1, so it isn't an ideal approach, too.

3 Conclusion

Now the biggest challenge is how to find an effective method to make the data in Tensor Train format directly through a nonlinear layer such as relu and sigmoid. If this problem can be solved, all of the data in the neural network can be represented in Tensor Train format. If combine it with an efficient reinforcement learning method such as Double PER Dueling DQN I introduced above, then I think a huge acceleration can be achieved, because the multiplication of two TT-tensor is really quicker than either the multiplication of two matrices or the multiplication of a TT-tensor and a matrix.
All of the codes can be found at https://github.com/zouc15/Speeding-up-reinforcement-learning/tree/master.

4 References

[1] Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Graves, Alex, Antonoglou, Ioannis, Wierstra, Daan, and Riedmiller, Martin. Playing atari with deep reinforcement learning. In NIPS Deep Learning Workshop. 2013.
[2] yenchenlin1994/DeepLearningFlappyBird
[3] Van Hasselt, Hado, Guez, Arthur, and Silver, David. Deep reinforcement learning with double q-learning. arXiv preprint arXiv:1509.06461, 2015.
[4] Schaul, Tom, Quan, John, Antonoglou, Ioannis, and Silver, David. Prioritized experience replay. arXiv preprint arXiv:1511.05952, 2015.
[5] Wang, Z., de Freitas, N., and Lanctot, M. Dueling Network Architectures for Deep Reinforcement Learning. ArXiv e-prints, November 2015.
[6] Volodymyr Mnih, Adri`a Puigdom`enech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016, pages 1928–1937, 2016.
[7] Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.
[8] Novikov, Alexander, Podoprikhin, Dmitry, Osokin, Anton, and Vetrov, Dmitry. Tensorizing neural networks. arXiv preprint arXiv:1509.06569, 2015.

[9] Garipov, Timur, Podoprikhin, Dmitry, Novikov, Alexander, and Vetrov, Dmitry. Ultimate tensorization: compressing convolutional and fc layers alike. arXiv preprint arXiv:1611.03214, 2016.

[10] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. arXiv preprint arXiv:1412.6553, 2014.

[11] D.V. Savostyanov, I.V. Oseledets, Fast adaptive interpolation of multidimensional arrays in tensor train format, in: Proceedings of 7th International Workshop on Multidimensional Systems (nDS), IEEE, 2011

[12] I. Oseledets and E. Tyrtyshnikov, TT-cross approximation for multidimensional arrays, Linear Algebra Appl., 432 (2010), pp. 70–88.

[13] S. Dolgov and D. Savostyanov, “Alternating minimal energy methods for linear systems in higher dimensions. part i: SPD systems,” arXiv preprint arXiv:1301.6068, 2013.

[14] ——, “Alternating minimal energy methods for linear systems in higher dimensions. part ii: Faster algorithm and application to nonsymmetric systems,” arXiv preprint arXiv:1304.1222, 2013

[15] M. V. Rakhuba and I. V. Oseledets. Fast multidimensional convolution in low-rank tensor formats via cross approximation. SIAM J. Sci. Comput., 37, 2015.

[16] I.V. Oseledets, TT Toolbox 1.0: Fast multidimensional array operations in MATLAB, Preprint 2009-06, INM RAS, August 2009.