Deep Reinforcement Learning for Atari Games

Wenqi Jiang, Manqi Yang, Ping Zhu, Tong Yu, Zhongtai Ren
wj2285, my2577, pz2232, ty2387, zr2208

Abstract

We train our agents to play Breakout, one of the most popular Atari games. Several deep reinforcement learning algorithms are implemented and compared: Deep Q Learning, Deep SARSA, Double DQN and Dueling DQN, and each algorithm employs two kinds of convolutional structures: LeNet and VGG-16. Due to the limited computational resources, we set a fixed training step to see which method performs best. Experiment shows that Dueling DQN with LeNet performs the best over other methods.

1. Introduction

Deep Reinforcement Learning (DRL) is one of the most popular fields in Computer Science these years. It combines neural networks, e.g. CNN and RNN, with traditional reinforcement learning algorithms such as Sarsa and Q Learning. The applications of DRL include self-driving, resource management, game-playing, etc.

Playing Atari games is a popular application in reinforcement learning researches [1][2][3]. We use several methods to train the agents. For convolutional neural networks, we use LeNet and VGG-16. For reinforcement learning algorithms, Deep Q Learning, Deep SARSA, Double DQN and Dueling DQN are implemented. Due to the limited amount of computational resources, we decide to set a fixed amount of training steps and see which method performs best. Result shows that Dueling DQN with LeNet performs best while the performance of all methods using VGG-16 are not good as our expectation.

2. Background

2.1. Atari Breakout

Atari was a leading company in arcade games and Breakout is one of the games they developed. The game start with 8 rows of bricks. Players control a platform that can move horizontally to bounce a ball. Each time the ball rebounds, one brick will be eliminated. The purpose of the player in the game is to clear all the bricks within 3 times of falling to bounce the ball. The game will not end automatically after clearing all the bricks unless the player loses 3 times. So we use reinforcement learning to train the robot to play games with 10,000 rebounds per cycle, 174 intervals (cycles) in total.

2.2. Convolutional Neural Networks

Convolutional neural networks (CNN) is among one of the most popular neural network structures. It has been proved to be excellent in tasks such as Image Classification and Object Detection [4][6][7]. Some of the most classic CNN architectures include [5][6][8]. Recent researches on Reinforcement Learning combine deep learning and traditional reinforcement learning: using CNNs or other neural network structures to extract features, such as state value and Q value; and feed these features into algorithms such as Q-Learning to train the model. This combination has been proved successful in multiple fields[1][9][10].

2.3. DRL

Deep Reinforcement learning is the result of applying reinforcement learning using deep neural networks. Reinforcement Learning is a paradigm that learn their own best successful strategies through taking actions and learning through trial-and-error in the interactive environment. The agents will learn their own knowledge through raw inputs. This is realized through deep learning of neural networks.

3. Approach

In this section, we split our approaches into two parts: convolutional neural network structures and deep reinforcement learning methods. The first part mainly focus on CNN structure while the second part combines CNNs with reinforcement learning.

3.1. Neural Network Structures

We will introduce two convolution neural networks used in our project: LeNet [4] and VGG-16 [5]. While we make some adjustment on LeNet, the VGG-16 structure is same as the original paper [5].

3.1.1. LeNet

LeNet is the first convolutional neural network used on image classification task. Even if more complex models such as [6][8] have dominated Computer Vision tasks, these networks requires humongous amount of training data and computation resources. LeNet, though lack of enough expression ability, is one of the most simple convolutional neural networks that only needs few feeding data and training time. Due to our limited time and GPU resources, we choose LeNet as one of our choice.

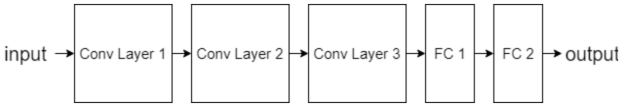


Figure 1. The structure of LeNet

We adjust several aspects of LeNet except its layer number. Because the input image size is larger than original paper, we assume larger filter size, more filter numbers in convolution layers and more neurons in fully-connected layers may lead to a better result. In convolutional layers, we use filter sizes of 8 by 8, 4 by 4 and 3 by 3 separately and corresponding filter numbers are 32, 64, 64. In fully-connected layers, we set the neuron number as 512 in the first two layers, and the output layer dimension is simply decided by action numbers.

3.1.2. VGG-16

VGG-16 is a big step for CNN: it proves the larger models, if carefully designed, can perform better than small models when training data is enough. The VGG-16 network consists of 13 convolutional layers and 3 fully-connected layers. The max filter number of convolution layers is 512, which contains much more parameters than LeNet, leads to a better expression capacity. However, the humongous amount of parameters may not be an advantage when training samples are insufficient. Also, it takes much more time to exert the full capacity of VGG-16 network.

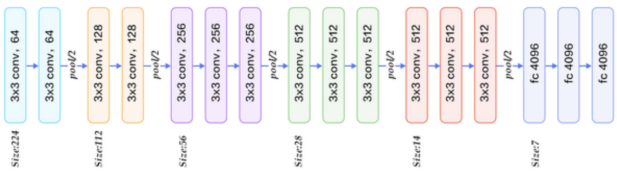


Figure 2. The structure of VGG-16 network

3.2. Deep Reinforcement Learning

3.2.1. Implement DNN in RL

When combining DNN with Q-Learning, we use a DNN $f_{Q^*}(s,a;\theta)$ to represent $Q^*(s,a)$. For TD algorithm, we need firstly initialize θ arbitrarily, then iterate until converge.

(1) Take action a from s using epsilon greedy policy derived from f_{Q^*}
(2) Then define loss function as:

$$L(\theta) = E[(R(s,a,s') + \gamma max_{a'} f_{Q^*}(s',a',\theta) - f_{Q^*}(s,a,\theta)^2]$$

Where

$$Target = Bellmanequation = R(s,a,s') + \gamma max_{a'} f_{Q^*}(s',a',\theta)$$

(3)Then the gradient of L with respect to θ can be reached:

$$\frac{\partial L(\theta)}{\partial \theta} = E[(r + \gamma max_{a'} Q(s',a',\theta) - Q(s,a,\theta))(\frac{\partial Q(s,a,\theta)}{\partial \theta})]$$

(4)SGD will be used to update θ and get the best Q value:

$$\theta \leftarrow \theta - \eta \nabla_\theta L$$

3.2.2. Techniques for Convergence

The problem is, naïve TD algorithm is easy to diverge due to the correlated samples and moving target. Moving target means when we update Q, the target is also changing, which is a moving target, thus hard to converge. The famous paper [1] proposed two stabilization techniques: for correlated samples it uses Experience replay to solve that, and moving target problem is solved through delayed target network.

(a) Experience Replay
Firstly use a replay memory D to store recently seen transitions (s,a,r,s')'s.
Secondly sample a mini-batch from D and update . Since the sample is random, the samples could much more possible to be independent identically distributed (i.i.d). Thus, the procedure of TD algorithm is:

(1) Take action a from s using epsilon greedy policy derived from f_{Q^*}.
(2) Observe s and reward R, add (s,a,r,s') to D.
(3) Sample a mini-batch of $(s^i, a^i, r^i, s'^{i+1})'s$.
(4) Re-define the loss function as the sum of samples of a mini-batch:

$$L(\theta) = \sum [(R^i + \gamma max_{a'} f_{Q^*}(s^{i+1}, a^{i+1}, \theta) - f_{Q^*}(s^i, a^i, \theta))^2]$$

(5) SGD will be used on each mini-batch to update and get the best Q value: $\theta \leftarrow \theta - \eta \nabla_\theta L$

(b) Delayed Target Network
To avoid chasing a moving target, set the target value at network output parametrized by old θ: θ^-
Then re-define the loss function again:

$$L(\theta) = \sum[(R^i + \gamma max_{a'} f_{Q^*}(s^{i+1}, a^{i+1}, \theta) - f_{Q^*}(s^i, a^i, \theta))^2]$$

And update $\theta^- \leftarrow \theta$ every k iterations.

(c) Other Tricks
Consider using reward clipping or batch normalization (better) to stabilize gradients.
Consider changing the optimizer, i.e. RMSProp, since it has an adaptive learning rate.

3.2.3. DQN

In Atari Games, the states are raw pixels, the actions are 18 combinations of joystick and button positions, and the rewards are scores.

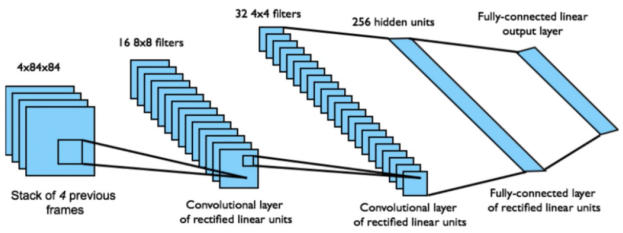


Figure 3. Original Network Structure of DQN.

The input is state s, which is a stack of raw pixels from 4 previous frames. The output is 18 corresponding to 18 actions. Furthermore, this architecture is uniform and can be used in all Atari games.

At each time step, the agent selects an action, pass it to the Atari emulator, and get a game score. Actually, the agent cant observe the internal state of the emulator, it can only observe an image from the emulator[11], which is the reason why we consider using VGG16 as our DNN model. The image is a screenshot of the current screen, and can present some visual features through raw pixel values.

The goal of the agent is to select actions that maximize the future rewards. For the forward process, the observation is the image feature of current screen, after resizing and converting it into gray scale, we store the image as an array. Based on the last/current observation, we get the current state, which is a list of last observations. Then based on the state, we can compute Q values, and use this Q value to select next action. For training part, the epsilon greedy policy is used to select the action, and for testing part, we choose greedy policy. For epsilon greedy policy, we initialize $\epsilon = 1$, then during the 1,000,000 step training, decrease the ϵ. Thus the intuition of the whole procedure is that the agent firstly explore the environment a lot then gradually stick to what it knows and take more exploitation.

For the backward process, firstly we store the most recent experience in memory, including recent observation, action, reward, etc. During training, we use SGD to optimize the model and use combined metrics of Mean_q, MAE loss, episode reward. There are two kinds of target model update method, and the parameter target-model-update will control how often the target network is updated: target-model-update should be greater than zero:
a) If target-model-update≥ 1, the target model is updated every target-model-update -th step. Different game may set a different initial value for this parameter. For Atari, we set target-model-update = 10000, which means, the target model will be updated on step 10000, 20000, and so on.

b) If target-model-update < 1, we use something called soft updates. The idea here is similar but instead of updating the entire target model (hard update), we gradually adopt changes like following:
target-model = target-model-update * new-target-model + (1 - target-model-update) * old-model

Using a target model stabilizes learning significantly and was one of the core contributions of the initial DQN paper (Mnih 2013). Also, using a target model will give us a stable target that can be optimized. [5]

Algorithm: DQN with Experience Replay
Initialize memory D and capacity N
Initialize action-value function Q with random weights
For episode = 1, M do
  Initialize sequence $s_1 = \{x_1\}$ and processed sequenced $\phi_1 = \phi(s_1)$
  For t = 1, T do
    With probability ε select a random action a_t
    Otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    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 mini-batch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_i = \begin{cases} r_j & for\ terminal\ \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & for\ non-terminal\ \phi_{j+1} \end{cases}$
    Perform SGD on loss function to update θ
  End
End

3.2.4. DOUBLE DQN

Though DQN performs higher accuracy and more stability than others algorithms, there is still a problem that q value is sometimes overestimated, which leads to a wrong action pick if any action is estimated with a higher q value than the optimal action. Q-learnings overestimations were first

investigated by Thrun and Schwartz (1993), who showed that if the action values contain random errors uniformly distributed in an interval [-,], then each target is overestimated up tom-1m+1 , where m is the number of actions. Noticing overoptimism due to estimation errors, Google DeepMind came up with a novel algorithm - Double DQN[2] to to reduce overestimations by decomposing the max operation in the target into action selection and action evaluation. In Double DQN, target network is built besides main network to give a more accurate q value. Then the action with the most q value is selected in main network to avoid choosing sub-optimal action. Its update is the same as for DQN, but replacing the target Y_t^{DQN} with

$$Y_t^{DoubleQ} \equiv R_{t+1} + \gamma Q(S_{t+1}, argmax_a Q(S_{t+1}, a; \theta_t); \theta_t')$$

In comparison to Double Q-learning, the weights of the second network tare replaced with the weights of the target network t-for the evaluation of the current greedy policy. The update to the target network stays unchanged from DQN, and remains a periodic copy of the online network. This version of Double DQN is perhaps the minimal possible change to DQN towards Double Q-learning. The goal is to get most of the benefit of Double Q-learning, while keeping the rest of the DQN algorithm intact for a fair comparison, and with minimal computational overhead.

The whole algorithm workflow is as below:
As shown in the explanation above and in the final results, compared to DQN, Double DQN successfully reduces overoptimism in DQN, finds better policies, resulting in more stable and reliable learning.

Algorithm: Double DQN Algorithm

Input: D - empty replay buffer; θ - initial network parameters, θ^- - copy of θ
Input: N_r- replay buffer maximum size; N_b-training batch size; N^--target network replacement freq,
For episode = 1 to M **do**
  Initialize frame sequence **x** ← ( )
  For t = 1 to T do
    Set state s ← x, sample action a ~ π_B
    Sample next frame x^t from environment ϵ given (s, a) and receive reward r, and append x^t to x
    If |x| > N_f **then** delete oldest frame $x_{t_{min}}$ from x **end**
    Set s' ← x, and add transition tuple (s, a, r, s') to D replacing the oldest tuple if |D| ≥ N_r
    Sample a mini-batch of N_b tuples (s, a, r, s') ~ Unif(D)
    Construct target values, one for each of the N_b tuples:
    Define a^{max}(s'; θ) = arg $max_{a'}$ Q(s',a'; θ)
    $y_j = \begin{cases} r & \text{if } s' \; is \; terminal \\ r + \gamma Q(s', a^{max}(s'; \theta); \theta^-) & Otherwise \end{cases}$
    Do a gradient descent step with loss $||y_j$ - Q(s, a; θ)$||^2$
    Replace target parameters $\theta^- \leftarrow \theta$ every N^- steps
  End
End

3.2.5. DEEP SARSA

While Q-learning is an off-policy method, SARSA is an on-policy method. It means when updating the current state-action value, the next action a will be taken. But in Q learning, the action a is completely greedy. Given such analysis, the update equation of state-action value can be defined as

$$Q(s, a) \leftarrow Q(s, a) + [r + \gamma Q(s', a') - Q(s, a)]$$

Actually, the difference between SARSA learning and Q learning lies in the update equations. In SARSA learning, the training data is quintuple-(s, a, r, s', a'). In every update process, this quintuple will be derived in sequence. However, in Q-learning a' is just for estimation and will not be taken in fact.

By replacing Q-learning with SARSA, also using CNN as the network, we could generate Deep SARSA network. Theoretically, Deep SARSA increases a little slower than DQN after a few intervals and converge much faster in the end.

3.2.6. DUELING DQN

Dueling DQN is an improved version of DQN. The dueling architecture separates the representation of state values and action advantages. As shown in Figure 4, both state and action values share the same convolution layer outputs. Nevertheless, the architecture compute them separately using fully-connected layers of different weights and dimensions. The network then combine these two parts using formula below.

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

Intuitively, the advantage of dueling architecture is that it can learn which states are valuable or useless. In this case, the network may not need to learn the effect of each action for each state. Results show that Duel DQN significantly outperforms DQN in most of 57 given Atari games.

4. Experiment Results

4.1. Metrics

A metric function is similar to a loss function, except that the results from evaluating a metric are not used when training the model. Any of the loss functions can be used as a metric function.

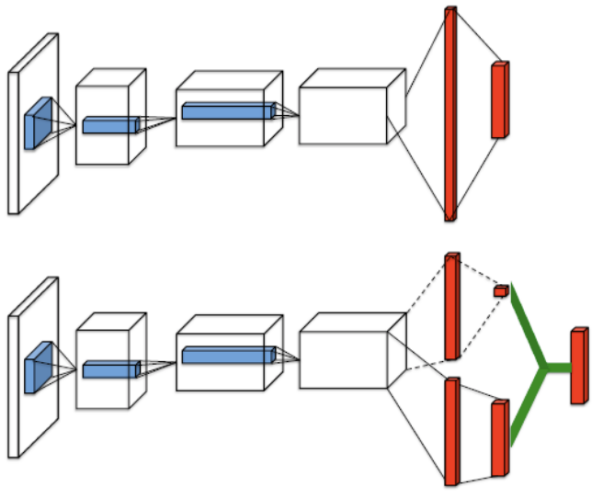
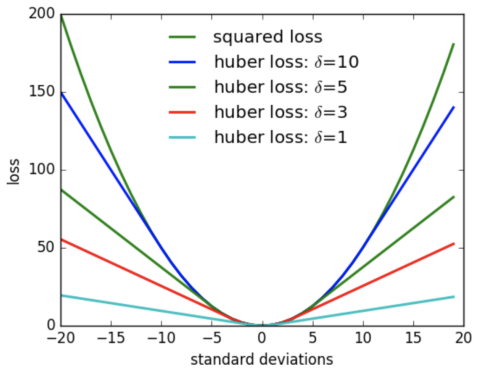


Figure 4. Dueling architecture

4.1.1. Loss

Here we use Huber loss. Huber loss is proposed to enhance the robustness to noise (or outliers) of squared loss function, and be less sensitive to outliers.

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



we set $\delta = 1$ in this paper.

4.1.2. Other metrics

Mean absolute error: Similar to loss function, but instead of Huber loss, we use mean absolute error as one of the metrics to compute the distance between target and current Q value.

Mean-q: For each episode, there are many steps. As mentioned before, every 4 frames will output a (18,) vector with 18 corresponding $Q^*(s, a)$. We firstly compute the max Q value among these 18 Q values, then calculate the mean Q value of the whole episode.

Mean-eps: As demonstrated above, for epsilon greedy policy, we initialize $\epsilon = 1$, then during the 1,000,000 step training, decrease the ϵ. Thus take more exploration at beginning and more exploitation at the end.

Episode-reward: For each action, can get a reward from the environment, we sum it up for each episode.

4.2. Analysis of results using LeNet

4.2.1. DQN

The DQN serves as our baseline model. In fact, DQN achieves the second best performance in this experiment: it performs better than Double DQN and Deep SARSA, but is not as good as Dueling DQN. One reason of the good performance is the small amount of trainable parameter number: it is less than Double DQN and all methods using VGG-16 network. With limited training steps, a smaller model may converge faster than others, even it has modest expression capacity. Another reason is the aggressive choices DQN make: unlike SARSA which always makes conservative choices, DQN may take risks to find the optimal actions.

4.2.2. Double DQN

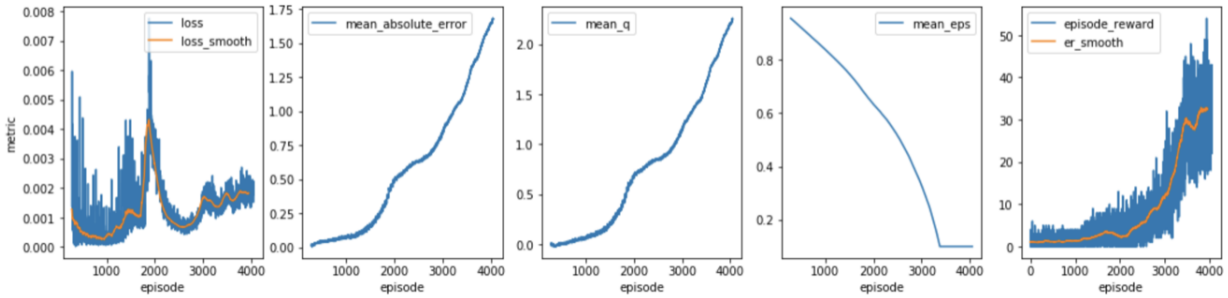
As we state before, Double DQN chooses actions with highest Q values without overoptimism, performs more stable and reliable learning. During our experiment, we can see clearly from the plots that Double DQN has almost the same trending as DQN goes, but has some slight differences when looking into this. Double DQN has slightly worse rewards comparing to DQN (4.95% down). However, it has better stability and reliability with a clearer trend to converge when the graphs fluctuate during the training process. This result confirms the theory that when using the same neural network, Double DQN has the same process of performing on-policy network. The only difference is when choosing the optimal action. Double DQN uses a target network to reduce overoptimism of calculating Q value, resulting in better stability and reliability, but a slightly slower speed and a slightly lower episode reward due to extra parameters in target network.

4.2.3. Dueling DQN

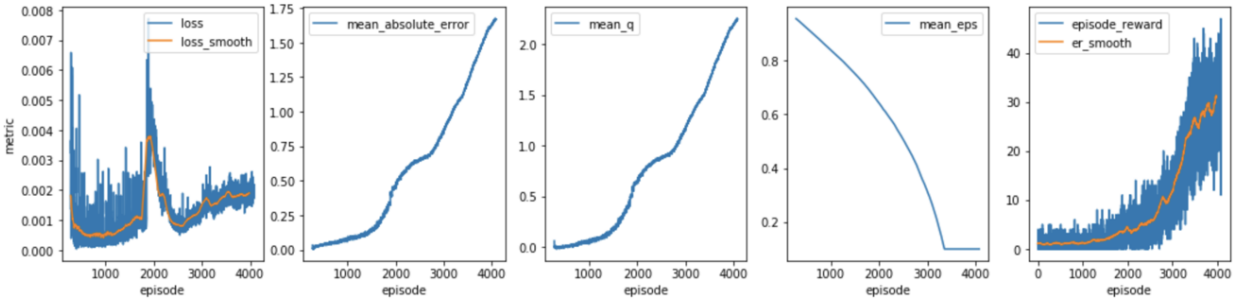
Dueling DQN achieves the best performance within the given steps in our experiment: the average reward of last 100 episode is 35.02, which outperforms DQN by 7.69%. The main advantage of Dueling DQN is that it splits state values and action advantages without adding much parameters.

.

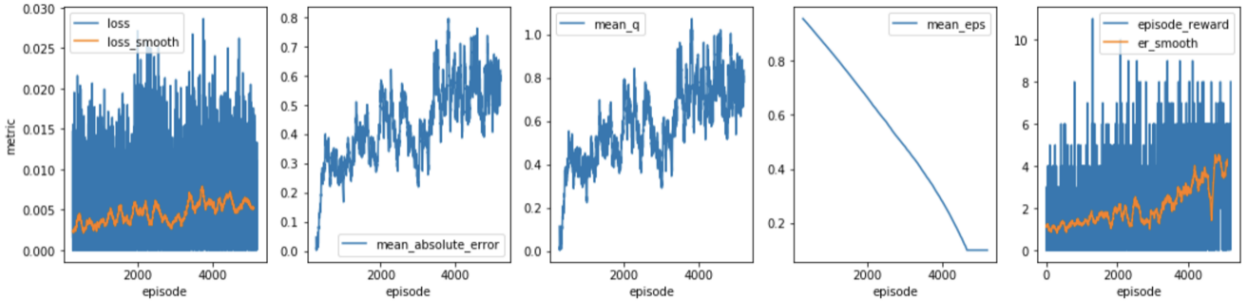
DQN



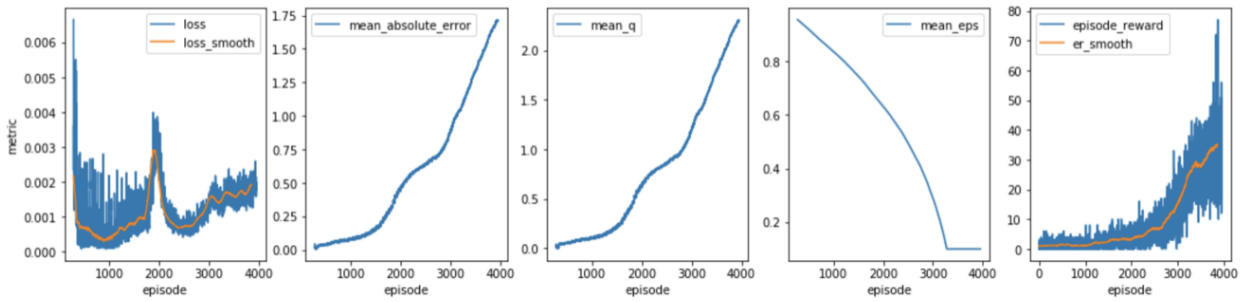
Double_DQN



SARSA



Dueling_DQN



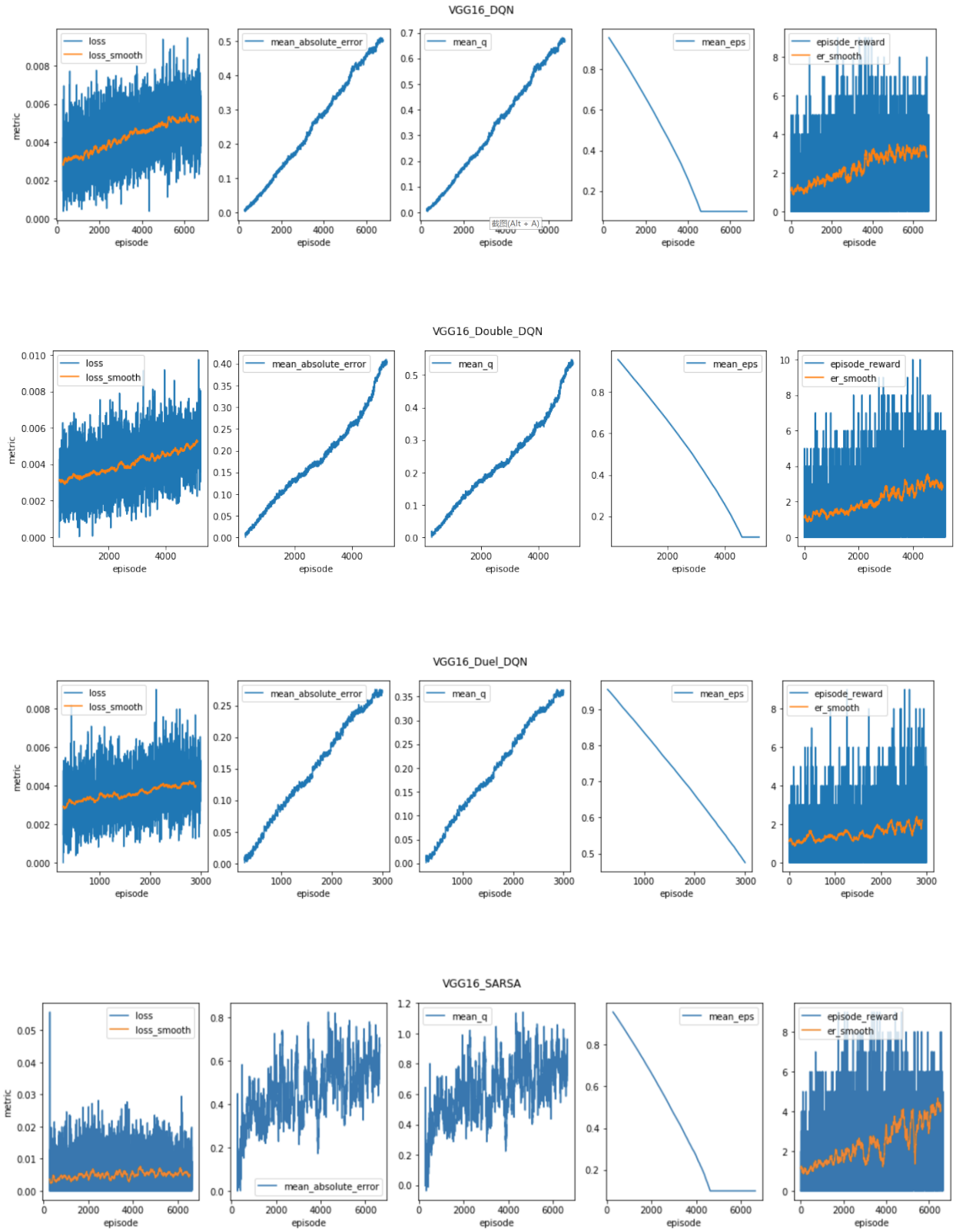


Figure 5. Training results of 8 methods

In this case, the training process of Dueling DQN can be as fast as DQN while keeping its advantage. Thus, it is not surprising that Dueling DQN performs the best in our experiment within a fixed amount of training steps.

4.2.4. SARSA

In our experiment, SARSA performs significantly worse than other algorithms: after 1,750,000 steps of training, it only achieves a reward value about 4.31 in the last 100 episodes, while other algorithms are able to achieve values between 30 to 40 approximately. The terrible result may attribute to the following reasons. First, SARSA is the most conservative algorithm in our project: if there exist risk of a large negative reward close to the optimal path, SARSA would avoid it and choose safer but not optimal actions instead, while Q Learning may take some risk to achieve the optimal result. Second, when learning optimal policy, SARSA employs ϵ-greedy algorithm with a decay rate. These hyperparameters can lead to performance fluctuation. Thus, in a fast-iterating environment, SARSA may not be the best choice, especially when training time is limited.

4.3. Analysis of results using VGG16

All the results using VGG-16 network are not satisfying enough: the average rewards of last 100 episodes range from 2.78 to 4.04, while DQN achieves 32.52. The unsatisfying results may mainly contribute to the large scale of VGG-16 network. The number of trainable parameters in VGG-16 is 52,466,816, which is 172.77 times of our adjusted LeNet. In this case, methods employing VGG-16 may not be able to achieve comparable result as LeNet within limited short training steps.

However, this result does not mean that VGG-16 is a secondary choice comparing to LeNet. Theoretically, VGG-16 network has a better expression capacity than LeNet. Once enough training data and steps are given, VGG-16 can get rid of serious underfitting and may performance better than LeNet.

5. Conclusion

In this paper, we first give a brief introduction of Deep Learning, Reinforcement Learning and their combination - Deep Reinforcement Learning. Then we continue to focus on one specific game in Atari 2600 game series - Breakout. In this project, we implement and compare several Deep Reinforcement Learning algorithms: Deep Q Learning, Sarsa, Double DQN and Dueling DQN, using LeNet or VGG-16 on CNN part. After we take a deep look into different Reinforcement Learning methods, we implement them and achieve final results.

We analyze the results and find that among the methods we use, Dueling DQN has the best episode reward and better stability in loss and episode reward. DQN and Double DQN have approximately the same performance, but the latter one has better stability and reliability. SARSA is much worse than the other methods due to limited training time but has clearer trend to converge. When using VGG-16, due to the huge CNN network, we spend tremendous amount of time training the model. However, because of the limited time, we achieve an underfitting network which lead to poor results.

In the future, we could make some improvements:
First, we can training longer for better results. In our project, due to the lack of computational resources as well as limited time, all the Reinforcement Learning methods are not given enough time to converge in the end. Results would be further improved and will be clearer to be shown if we continue to train these models. Second, in our project, we can implement more latest methods in recent Reinforcement Learning researches. For improvement, we will implement other up-to-date methods as well as making some improvement on existing methods for better performance.

6. Contributions

Manqi Yang: Configurate GCP environment; Implement DQN and Double DQN and write the relevant report. Plot and analyze different results from other members.

Wenqi Jiang: Configurate GCP environment and provide materials about Linux; Implement VGG network and SARSA, write the relevant report.

Ping Zhu: Implement Dueling Network and SARSA, and write the relavant report.

Tong Yu: Configurate GCP environment; Implement VGG16_DQN, write the relevant report, and realize LaTex Layout.

Zhongtai Ren: Configurate GCP environment; Train part of the model and write the introduction and background of the report.

References

[1] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529.

[2] Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning." AAAI. Vol. 2. 2016.

[3] Wang, Ziyu, et al. "Dueling network architectures for deep reinforcement learning." arXiv preprint arXiv:1511.06581(2015).

[4] LeCun, Yann. "LeNet-5, convolutional neural networks." URL: http://yann. lecun. com/exdb/lenet (2015): 20.

[5] Long, Jonathan, Evan Shelhamer, and Trevor Darrell. "Fully convolutional networks for semantic segmentation." Proceedings of the IEEE conference on computer vision and pattern recognition. 2015.

[6] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

[7] Ren, Shaoqing, et al. "Faster r-cnn: Towards real-time object detection with region proposal networks." Advances in neural information processing systems. 2015.

[8] He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

[9] Silver, David, et al. "Mastering the game of Go without human knowledge." Nature 550.7676 (2017): 354.

[10] Kaelbling, Leslie Pack, Michael L. Littman, and Andrew W. Moore. "Reinforcement learning: A survey." Journal of artificial intelligence research 4 (1996): 237-285.

[11] https://www.cs.toronto.edu/ vmnih/docs/dqn.pdf