Apply DQN on Atari Game: Jellybean World

Xueyang Zhang, Dailun Li, Xiangyu Liu, Kaiwen Xu
McGill University

Abstract

Training an agent to play Atari games is still a challenge in Reinforcement learning. In this paper we try to tackle one of the more difficult ones, Jelly Bean World. In particular, we set up the DQN algorithm to give the agent a direction every time it moves based on its vision. The neural network we used in the DQN is a typical CNN that can convert vision into abstract patterns to help the agent generalize the world. After fine-tuning the model hyperparameters, we get our agent capable of achieving a reward of 646.5. We may improve our model by adopting other algorithms in the future, like DDQN and Priotized Replay.

1. Introduction

Jelly bean world is one of the difficult Atari games. Not only the environment is infinite in 2D, but there is no end to the game. To make the objective better defined and making reward better measured, we make it an episodic task. In other words, we will make the game start with the initial distribution as usual, but forcefully end the game after a fixed 5000 steps. The accumulated reward throughout these steps are seen as the reward of the episode, so naturally we need to design an agent such that it gets as much awards as possible.

At each time step, The agent can choose one of the four directions, North, East, South, and West, and go one step forward in the direction chosen. Since the world has an infinite map, it will always be able to go in any direction without the worry of hitting a wall. There are four different kinds of fruits in the world, each having a different color and scent. Each fruit has its distribution in terms of position in the world, but the agent needs to learn the distribution through input. The agent can get reward by gathering the fruits. Some fruits will give the agent positive reward while others will give negative reward. These are for the agent to learn and plan accordingly.

Luckily, the agent not only knows the spot it is in, but also the situation of the 15*15 square centered by the agent. To be specific, the agent has vision, thus it sees the surrounding 15*15 image in the form of RGB. In each spot, it can either be a kind of fruit, or nothing. In addition, it can perceive the composed scent of all the fruits, which can potentially help the agent planning, especially if it notices difference in scent in different spots.

There are many difficulties in making an agent that can get the most point throughout a fixed number of steps. For example, the agent needs to figure out the effect of moving in different directions in terms of vision and scent.

2. Method & Motivation

2.1 Algorithm and the choice of Model

DQN is famous for its astounding performance in playing atari games. Where sequence of actions are converted into images and then observed by the network. The target of the agent is to maximize its future reward based on the action it takes. For an interval of timesteps, either a random action of probability ϵ or an action with the max Q value output by the network is selected. With a buffer that keeps all these information, the tuples of transitions will then be sampled out as batches. After that, gradient descent is performed by computing the MSE between target and the output of the network. This algorithm will be replayed a number of times to generalize the future reward of each action.

With both Feature and Vision vectors being square matrices that contain position information, a CNN approach seems feasible for our learning process. We derived our CNN based on VGG architecture. However, considering the size of input, we choose to reduce the size of our network. A detailed design of our model is shown in Figure 2. This model will compute and predict the q value received after each action, and back propagate using the real action value from the environment. The kernel size of the first few conv layers are set to 3 since our input has only a size of 15*15, a larger kernel size may result in a lost of information. We also experimented a number of trials on the size of the fully connected(fc) layers by launching the model with fixed seed and running it for 20k time steps. The fc layer with size 128 and 256 performs similarly over the first 20k time steps, and when the number of fc layer increases to 3 and 4, no major improvement in performance is observed. In order to save the computation time, we choose to set the number of fc layers to be 2 and the size of fc layer to be 128.

2.2 Analyzing Inputs

We explored the following aspects of the game:

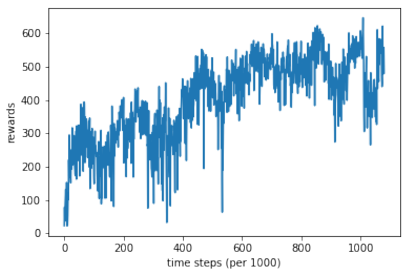
- Feature: Feature is a 1D vector, and it can be seen as a flattened version of 15*15*4 matrix that indicates the position of a particular item based on the sight of our agent. However, we figure out that the vector extracted fails to carry the original information when transposing the feature vector. This is demonstrated in the output of accumulated reward: the reward fluctuates between 20 and 60 with over 100k training steps. With the render method provided in the Jellybean class, we figure out that the agent persistently follows one direction. Thus, we move on to other vectors.
- Scent: The Scent vector contains the information diffused by an item in the 15*15 grid. It appears to be logical to combine the scent vector with one of feature or vision vector to form the network. However, little improvement has been made when the scent vector is concatenated with the output of the Convolutional layers.
- Vision: The vision vector can be seen as an RGB image that contains the position of items surrounding our agent. Therefore one straight forward approach would be to construct a convolutional neural network that accepts RGB images as inputs, filter and detect the features throughout the conv layers.

2.3 Hyperparameter Tuning

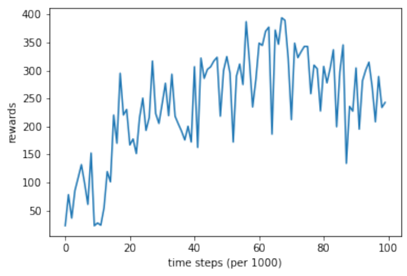
We let the epsilon start from 1 and uniformly decay to 0.02 after 10k time steps to balance exploration and exploitation. In this way, the model will initially explore and try different actions to gather a basic understanding of the environment. After 10k timesteps, the model will mostly perform exploitation. This would help more when the inner parameters of the model needs to be fine-tuned to search over the performance grid. We initially set the learning rate of the adam optimizer to be 1e-2. However, the performance (measured by accumulative reward) fluctuates between 80 and 500 even after 100k steps, which means the learning rate is to high for the model to converge. We then set the learning rate to be 1e-3, with beta1 and beta2 to be 0.99, 0.999 respectively. In this time, the model's performance gradually increased over the 1M steps. The maximum result is 646.5 when the evaluation frequency is 1000 steps.

3. Results Discussion

The result is analyzed by performance and sample efficiency below.



(a) acc reward in 1M timesteps



(b) acc reward in 100K timesteps

Figure 1: Performance and Sample Efficiency Plot

3.1 Performance

For the first 20000 steps, the reward versus accuracy varied since it is searching for the local maxima of the optimal weights. After that, the general reward received steadly increases. Although in certain timesteps the reward may rapidly drops, it would correct itself immediately afterwards, demonstrating the effectiveness of our algorithm. The highest accumulative reward reached is 646.5, at timestep 1,009,000. To save the computation time, the algorithm terminates after 1.1M timesteps, as the reward begins to fluctuate over 800K timesteps.

3.2 Sample Efficiency

As we can see from Figure 1(b), the accumulated reward in the first 100k time steps is also generally increasing. What's more, the highest reward reaches 400, which is not far at all from the highest accumulated reward we would finally get. Similarly, we also notice that the accumulated reward took a big dip in some steps but goes back to the original high level very soon, as we have seen throughout the training process.

4. Conclusion

In this project, we explored the never-ending environment of Jelly bean World(JBW) and designed an agent whose goal is to maximize the rewards collected from the environment. Based on the fact that the state space is continuous and massive, while the action space is discrete and simple, we choose to adopt the Deep Q-Learning(DQN) algorithm, which uses a neural network to store the action values(q-value) and a buffer to reply the experience. After experimenting with using features+scents as input, we find that the agent acts poorly even after a long training time, which drives us to use vision space instead. Our DQN model uses a Convolutional Neural Network as the q-net and target net, and the vision space as the only input. The trained agents can get 577 as the mean accumulative rewards over 20 episodes.

5. Future Improvements

First, the input can undoubtedly be further improved by combining the vision and scents instead of vision only since the observed space is pretty limited(15*15). Sometimes, we cannot extract enough information just from the vision space. A naive implementation uses the concatenation of the scents vector and the processed vision space (output of the convolutional layers) as the input for the fully connected layers. We believe the full use of comprehensive information can increase model performance and sample efficiency.

Second, we could try some other advanced algorithms to solve the flaws naturally existing in the Deep Q-Learning algorithm. An improved version of DQN would be the Dueling DQN representing[1] two separate estimators: one for the state value function and one for the state-dependent action advantage function, which could help avoid high Q-value to train on. Another implementation to try is Prioritized Replay[2]. Instead of randomly choosing a batch from the memory to train on, it uses the TD-error to compute the probability of a sample being chosen, which can significantly enhance the learning efficiency.

Besides the choice of the input and model, hyperparameters can also be more precisely tuned to achieve a better balance between the model performance and computing resources. For example, a larger batch size can improve the learning process while requiring more memory space. Meanwhile, the initial exploration can also be further optimized to accumulate more high-quality samples at an early stage, avoiding a mass of repetitive and redundant data.

References

[1] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot and Nando de Freitas. Dueling Network Architectures for Deep Reinforcement Learning, 2015; arXiv:1511.06581.

[2] Tom Schaul, John Quan, Ioannis Antonoglou and David Silver. Prioritized Experience Replay, 2015; arXiv:1511.05952.

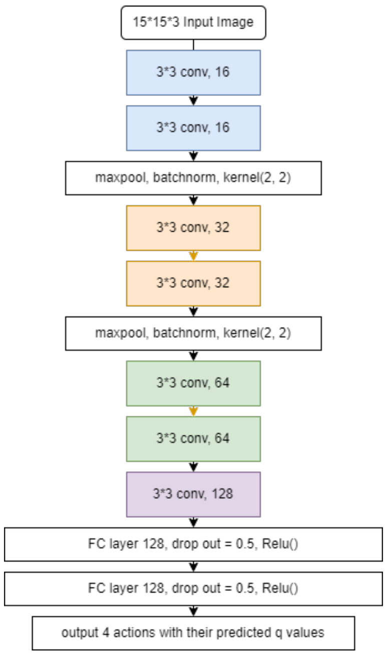

15*15*3 Input Image
3*3 conv, 16
3*3 conv, 16
maxpool, batchnorm, kernel(2, 2)
3*3 conv, 32
3*3 conv, 32
maxpool, batchnorm, kernel(2, 2)
3*3 conv, 64
3*3 conv, 64
3*3 conv, 128
FC layer 128, drop out = 0.5, Relu()
FC layer 128, drop out = 0.5, Relu()
output 4 actions with their predicted q values

Figure 2: the structure of our network

.