ISSN 2616-5767 Vol.4, Issue 3: 52-57, DOI: 10.25236/AJETS.2021.040306

Fetching Policy of Intelligent Robotic Arm Based on Multiple-agents Reinforcement Learning Method

Xiao Huang[1, *]

[1]*The Stony Brook School, Stony Brook, NY, USA*
**Corresponding author*

Abstract: *Traditionally, robotic arms on production lines perform actions by computing rotational matrices, which control the movement of each joint and repeat their work by following commands that were previously programmed. Although with accuracy and efficiency, traditional robotic arms are unable to complete their tasks when the preset conditions change. In this paper, we propose a method using novel Q learning method with residual neural network to train robotics arms. Comparing to traditional models, this novel method results in a better performance for robotic arms after training. The environment is abstracted into the fetch and place problem. The agent we trained could make a policy to fetch various objects, with an accuracy of 92.37%. As it takes only about an average of 32 consecutive commands to complete a task, it is more efficient in execution than any other agents trained only by the usual reinforcement method.*

Keywords: *Robotic Arm, Reinforcement Learning, Deep Learning*

1. Introduction

Reinforcement Learning (RL) [1] is a branch of machine learning in which an agent learns from interacting with an environment. Before an agent or robot (software or hardware) can select an action, it must have a good representation of its environment [2]. Thus, perception of the environment is one of the key problems that must be solved before the agent can decide to select an optimal action to take. Representation of the environment might be given or might be acquired. In reinforcement learning tasks, a human expert usually provides features of the environment based on his knowledge of the task. However, for realistic applications, this work should be done automatically, since automatic feature extraction will provide much more accuracy. There are several solutions addressing the challenge, such as Mont Carlo Tree search [3], Hierarchical Reinforcement Learning [1] and function approximation [4]. Therefore, we look for better representation of environment for better performance of the robotic arms.

The most well-known reinforcement learning algorithm which uses neural networks (but no deep nets, i.e., there is only one hidden layer) is the world-class RL backgammon player called TD-Gammon, which gained a score equal to human champions by playing against itself [4]. TD-Gammon uses TD (lambda) algorithm to train a shallow neural net to learn to play backgammon game. However, later attempts to use the same method for other games such as chess, Go and checkers were not successful. With riving interest in research works on deep learning among 2000s, the promise, to use neural networks as function approximator both for the state value function $V(s)$ and the action-value function $Q\ (s, a)$ in visual based RL tasks, returned.

Deep learning attempts to model high-level abstractions in data using deep networks of supervised and/or unsupervised learning algorithms, in order to learn from multiple levels of abstractions. For example, it learns hierarchical representations in deep architectures for classification. In recent years, deep learning has gained huge attraction not only in academics (such as pattern recognition, speech recognition, computer vision and natural language processing), but also been used successfully in industry products such as voice search engines. Recent research has also shown that deep learning techniques can be used to learn useful representations for reinforcement learning problems [5]. Combining RL and deep learning techniques enables RL agent to have a better perception of its environment.

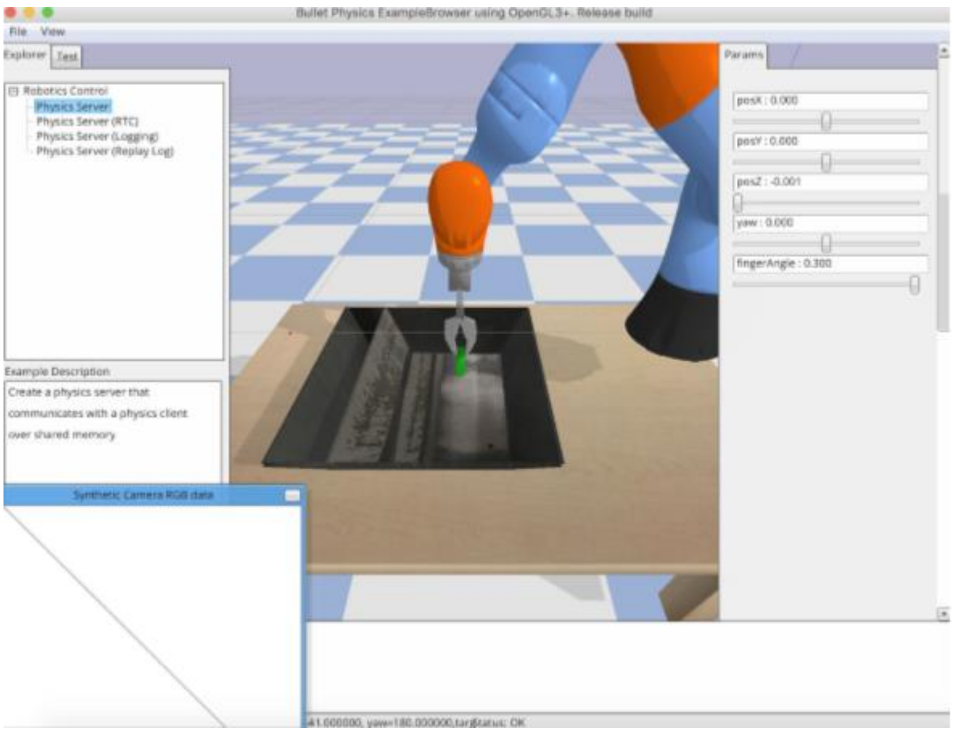
Other related works are about robotic arms controlling methods. The mature methods used in the industry are different variants of PID controlling. The cost for this method is cheap, and it fits the actual needs of the traditional factory, which only produces products of limited types. However, large amount of refinement of parameters of the PID controller is required after modeling the robotic arm in the specific industrial environment. Further advanced techniques take advantages of visual recognition methods, which could interpret the visual information taken by the normal camera. Similar methods like object detection are widely used. However, this only provides front-end method about the target, which offers no solution to the controlling method. Others who try to solve the controlling method may use traditional controlling methods and reinforcement learning. In this paper, I will introduce a single-agent algorithm, which shows significant end-to-end effects, as it takes raw camera image as its input and outputs controlling commands sequence, which could help to control the arm to perform target task.

2. Method

2.1 The simulation of the Robotic Arms

There are several practical issues we face to train the actual robotic arms in experiments. The cost would be significantly high experimenting with real robotic arms as they consume large amounts of energy performing actions required for the training process. Errors such as out-of-bound data always appear during the training process, which causes potential damage to the robotic arms and other equipment while training. Besides, it takes a longer time for the physical machines to complete their tasks, and extra time is wasted to reset the agent and the environment after a set of actions was performed.

The whole simulation is based on pybullet physical engine. The main framework of this environment is an industrial robot arm, which aims to grasp object in a tray. The only feedback is the image taken above the tray. The major reward is given in the end when the robot grasped the object above a certain height. There are also some tiny awards happen during each step.



Create a physics server that communicates with a physics client over shared memory

posX : 0.000
posY : 0.000
posZ : -0.001
yaw : 0.000
fingerAngle : 0.300

41.000000, yaw=180.000000,targStatus: OK

Figure 1: Details of Reinforcement Learning

We consider tasks in which an agent interacts with an environment E, in this case the robot arm emulator, in a sequence of actions, make observations and get the rewards. At each time-step the agent selects an action from the set of legal game actions, $A = \{1, \ldots, K\}$. The action is passed to the emulator and modifies its internal state and the game score. In general, E is stochastic where the object to be picked up is generated at a random place on the tray. The emulator's internal state is not observed by the agent; instead, it observes an image, $x_t \in R_t$, from the emulator, which is a vector of raw pixel values representing the current screen. Moreover, it receives a reward r_t representing the change in game score. As the game score may depend on the whole prior sequence of actions and observations, the feedback of an action may only be received after thousand time-steps. Since the agent only observes images of the current screen, the task is partially observed and many emulator states are perceptually aliased, i.e. it is impossible to fully understand the current situation from only the current screen x_t. Thus, we consider the sequences of actions and observations $(s_t = x_1, a_1, x_2, \dots, a_{t-1}, x_t)$ and we learn the game strategies that depend on these sequences. All sequences in the emulator are assumed to terminate in a finite number of time-steps. This formalism gives rise to a large but finite Markov decision process (MDP) in which each sequence is a distinct state. As a result, we can apply standard reinforcement learning methods for MDPs, simply by using the complete sequence s_t as the state representation at time t. The goal of the agent is to interact with the emulator by selecting actions in order to maximizes future rewards. We make the standard assumption that future rewards are discounted by a factor of γ per time-step, and define the future discounted return at time t as $R_t = P_T$ where $t_0 = t\,\gamma_{t_0} - tr_{t_0}$, and T is the time-step at which the game terminates. We define the optimal action-value function $Q*(s, a)$ as the maximum expected return achievable by following any strategy, after seeing some sequence s and then taking some action $a, Q*(s, a) = max\pi\, E[R_t \,|\, s_t = s, a_t = a, \pi]$, where π is a policy mapping sequences to actions (or distribution over actions). The optimal action-value function follows the Bellman equation. This is due on the following reason: if the optimal value $Q*(s_0, a_0)$ of the sequence s_0 at the next time-step was known for all possible actions a_0, then the optimal strategy is to select the action a' maximizing the expected value of $r + \gamma\, Q*(s_0, a_0)$,

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s', a') \middle| s, a\right]$$

The basic idea behind many reinforcement learning algorithms is to estimate the action value function, by using the Bellman equation as an iterative update, $Qi + 1(s, a) = E[r + \gamma\, maxa'\, Qi(s_0, a_0)|s, a]$. Such value iteration algorithms converge to the optimal action value function, $Qi \to Q*$ as $i \to \infty$ [23]. In practice, this basic approach is totally impractical, because the action-value function is estimated separately for each sequence without any generalization. Instead, it is common to use a function approximator to estimate the action-value function, $Q(s, a;\, \theta) \approx Q*(s, a)$. In reinforcement learning, this is a typical linear function approximator, but sometimes a non-linear function approximator is used instead, such as a neural network. We refer to a neural network function approximator with weights θ as a Q-network. A Q-network can be trained by minimizing a sequence of loss functions $Li(\theta_i)$ that changes at each iteration i,

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[\left(y_i - Q(s,a;\theta_i)\right)^2\right],$$

where $y_i = Es_0 \sim E\,[r + \gamma\, maxa_0\, Q(s_0, a_0;\, \theta_i - 1)|s, a]$ is the target for iteration i and $\rho(s, a)$ is a probability distribution over sequences s and actions a that we refer to as the behavior distribution. The parameters from the previous iteration $\theta_i - 1$ are held fixed when optimizing the loss function $Li\,(\theta_i)$. Note that the targets depend on the network weights; this is in contrast with the targets used for supervised learning, which are fixed before learning begins. Differentiating the loss function with respect to the weights we have the following gradient,

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

Rather than computing the full expectations with the above gradient function, it is easier to optimize the loss function by stochastic gradient descent. If the weights are updated after every time-step, and the expectations are replaced by single samples from the behavior distribution ρ and the emulator E respectively, then we arrive at the familiar Q-learning algorithm [26]. Note that this algorithm is model-free: it solves the reinforcement learning task by directly using samples from the emulator E, without explicitly constructing an estimate of E. It is also off-policy: it learns by following the greedy strategy $a = maxa\, Q(s, a;\, \theta)$, while following a behavior distribution that ensures adequate exploration of the state space. In practice, the behavior distribution is often selected by a Ɛ-greedy strategy that follows the greedy strategy with probability $1-\varepsilon$ and selects a random action with such probability.

2.2 The Policy Model

In reinforcement learning, the neural network is usually used as policy, which could take the current observation as its input and output the calculated instructions for actions. In this project, the instructions are commands of the robotic arm. The policy is designed to be stateless, which does not contain the information of the past. We aim to find more effective models for the training process to improve the overall performance while reducing the training time.Therefore, we used the residual model. The structure of the model we used could be seen from the following figure. In this project, we take a shallow version of the residual model because of the limitation of the hardware. The hardware needs to simulate the robotic arm in real time and train the model at the same time, meaning that with too many parameters it may cause out of memory model.

2.3 Action space

The action space also influences the policy model. One way to represent the vector space is to treat the problem as a multi agent problem as the robotic model has three joints, which could all be regarded as individual agents and each of them represent an individual vector. However, this representation could lead to complex structure of the training model. It is better to represent the entire robotic arm with three joints as a single agent which has action space with a size of 2^{m*n}, where m is the number of joints while the n is the number of action choice of single joints. However, the disadvantage of this implementation is also obvious: it may face dimension explosion as the number of joints increase.

2.4 The training of the policy model

To fit the policy that the agent uses for choice with our neural network, we train our model using backpropagation. When every time it finishes the process of trials to complete the assignment, the weights, parameters in the neural network deciding how the agent takes actions in the actions set, are updated so that our model better fits the policy demanded. We have adjusted the parameters of the neural network model several times for the better performance of the robotic arms with a shorter training time. For complicated models with more convolution layers, it significantly increased the training time to achieve what we want, considering the equipment we have for this project.

An agent (object, here, the robotic arm)'s observation of the environment (the agent's surroundings and its current status) allows it to complete the tasks through the policy (the process of selecting the best action from the action sets). For each action taken by the agent, the environment returns feedback known as a reward, which reflects how good the action is completing the assigned task. In this project, the agent is the simulated robotic arm; the environment includes the item to be picked up, the board, and the agent itself; the observation is the camera pixels of an image of the environment that will be perceived by the agent; policy refers to the agent's choice to complete the task, and it mostly represents the process that the robotic arms both choose and perform the actions. The robotic arm would choose by selecting the actions it will take to pick up an object based on an image of the entire environment. Our goal is to train the robotic arm's policy so that it makes better choices to complete the assignment. After each trial is complete, a reward is given as a float if the object is grasped to a certain height. The closer it gets to the top of the robotic arm, the higher the reward returned.

3. Experiment & Results

The simulation of the robotic arm is based on the physical engine, pybullet. The other code framework is under openAI, which is a platform providing packages to deal with deep reinforcement learning problems. It abstracts the concepts of environment, agent and the rewards and provides standard interface to implement.

The rewards granted to the model increase over time during the training.

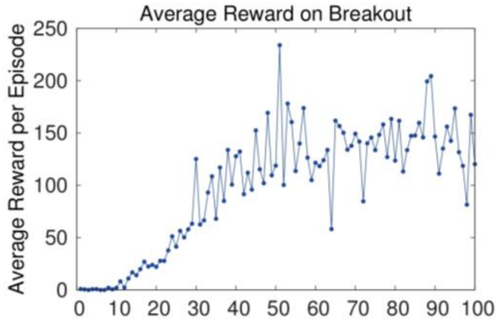


Figure 2: Average Reward

In the first experiment, we tested the computation efficiency of models with different layers. We utilized the model with 18, 34, and 52 layers. With more layers, the model will be slower to train. The result could be seen from the following table: model with more layers will approach the target with less average steps, which means the decision of each step is much more valuable, while model with less layers could compute with less computation pressure. The model with further parameters would fail because demanded memory needs.

Table 1: Layers and steps

Numbers of layers	memory consumption	average fetching steps	Average policy generation time for each step	Average Time to complete a task
18	34.8m	59	0.82s	48.48s
34	67.2m	46	1.13s	51.98s
52	112m	41	1.35s	55.35s

Although the model with more layers results in a “smarter” machine as less steps of actions are required to complete the assigned task, it takes comparatively longer time in total to complete the same task. Overall, with limited hardware, using a shallower structure of residual network with less layers would result in better controlling for practical usage, and it reduces the potential of crashes during the training process due to memory consumption.

The other experiment is between single action space and vectorized action space. The single action space uses Q learning method, while the vectorized action space needs PPO2 method.

Table 2: Action space

action space	average step	training time
single action space	46	7.5h
vectorized space	73	13.5h

From the chart above, it could be seen that the single action space outperforms the vectorized action space in both training time and performance, because it is much easier to control one agent comparing to the coordination of three independent agent vectors.

The final experiment is the memory replay mechanism. The memory replay mechanism is to utilize the random pieces of past instructions and rewards to train the policy model. The memory replay mechanism will be conducted on another threads, which reduced the training time and stabilized the training process by using a diverse distribution of data.

Table 3: Final experiment

memory replay	converging time	threads num
true	7.5h	2
false	17h	1

4. Conclusion

From the data obtained from experiments, using residual neural network as policy model based on Q learning method shows a better performance of robotic arm while having a significantly shorter training time, with limited hardware. The residual network enables the machine to learn the information from the past steps, which highly improve the efficiency of the training process as it makes sure that the new policy updated during the training process would be better than the old ones, comparing to other models which might show obvious fluctuation of rewards during the training process.

However, the algorithm we proposed based on the premise of limited hardware, meaning that other models or methods might show different results when the preset condition changed. For example, with hardware that could execute much faster calculation, the policy generating time could be reduced using vectorized space with three agents, which might minimize the significance of the simplified method using single agent. In future, we might conduct similar experiments on better machines or different environments, and test if the same method still shows better results than other methods. If not, we would love to improve the structure of the residual network and/or combine it with other reinforcement learning algorithm.

References

[1] Kaelbling, L. P., Littman, M. L., Moore, A. W., 199s6. Reinforcement learning: A survey. Journal of artificial intelligence research 4, 237–285.
[2] Kober, J., Bagnell, J. A., Peters, J., 2013. Reinforcement learning in robotics: A survey. The International Journal of Robotics Research 32 (11), 1238–1274.
[3] Vien, N. A., Ertel, W., Dang, V.-H., Chung, T., 2013. Monte-carlo tree search for bayesian reinforcement learning. Applied intelligence 39 (2), 345–353.
[4] Sutton, R. S., Barto, A. G., 1998. Introduction to reinforcement learning. Vol. 135. MIT press Cambridge.
[5] Mattner, J., Lange, S., Riedmiller, M., 2012. Learn to swing up and balance a real pole based on raw visual input data. In: International Conference on Neural Information Processing. Springer, pp. 126–133.