Tackling Obstacle Tower Challenge with Random Network Distillation

Hanchul Choi, Lead Data Scientist at S&P Global, hanschoi@stanford.edu
TA: Jay Whang, jaywhang@cs.stanford.edu

Abstract

This paper is a final report for Stanford CS230 Deep Learning course. In this paper, we try to solve the Obstacle Tower Challenge sponsored by Unity and Google Cloud, by tuning the hyperparameters of a baseline model using Deep Q-Network algorithm, by implementing different neural network architectures inspired by different breakthroughs in computer vision research, and by proposing a simple variant of the Random Network Distillation, which is a recent work by OpenAI that produced state-of-the-art results in the famous Montezuma's Revenge environment.

1. Introduction and Background

1.1. Problem Statement

The main objective for this project is to implement and apply various deep learning architectures and algorithms to solve the Obstacle Tower Challenge. The Obstacle Tower is a reinforcement learning environment created by Unity, where an agent has to climb a tower that becomes increasingly difficult as the level advances. The environment provides a comprehensive benchmark that tests computer vision, locomotion skills, high level planning, and generalization. (Juliani et al., 2019) This year, Unity hosted the Obstacle Tower Challenge at AICrowd, which is a new Kaggle-like platform for data science challenges in the field of machine learning, deep learning and reinforcement learning. (link)

The competition consists of two rounds. Among the 350+ teams who participated in round 1 that lasted from February 11th to April 30th, total of 44 teams advanced to round 2 as finalists. Round 2 is scheduled to finish on July 15th, 2019.

Using the Obstacle Tower Challenge as a content matter, I tried to touch upon two of the three topic types - application, algorithm, and theory - outlined in the CS230 project instructions.

- Application: implement and apply different deep learning architectures learned in the Stanford CS230 Deep Learning course in order to improve the agent.
- Algorithm: propose a new variant to the recently published reinforcement learning algorithm.

1.2. Related Work

Reinforcement learning is a relatively recent field of research, even more so than the machine learning and deep learning field. One of the most notable papers in the past few years was on the Deep Q-Network (Mnih et al., 2015), which effectively combined a deep learning function approximater with value-based reinforcement learning agent. The paper introduced major breakthroughs that enabled stable usage of deep learning in the reinforcement learning context, such as experience replay that broke the correlation between consecutive observations and fixed Q-targets that stabilized the update process. The algorithm performed at or above human-level in more than half of Atari environments.

There has been several improvements on the Deep Q-Network algorithm, such as dueling network (Wang et al., 2015), prioritized experience replay (Schaul et al., 2015), and distributional network. (Bellemare et al.) The Rainbow algorithm combines six extensions of the Deep Q-Network to produce vastly improved performance on various Atari benchmarks. (Hessel et al., 2018) In this project, we use a Deep Q-Network that is combined with three of the six extensions mentioned in the Rainbow algorithm paper as a baseline model, which are prioritized replay, multistep updates, and distributional network. For convenience, we naively refer to the baseline model as a Deep Q-Network (DQN) agent despite the extensions.

Although Deep Q-Network and Rainbow algorithm made great breakthroughs in solving many Atari games, environments with sparse reward structures such as Montezuma's Revenge were difficult to tackle with these algorithms alone. There has been a few proposals that made marginal improvements in performance, but major breakthrough was published only a few months ago by OpenAI. The authors used Random Network Distillation algorithm that scored the state-of-the-art results in the Montezuma's Revenge.

(Burda et al., 2018) The content of the Random Network Distillation will be covered in detail in later sections.

The overall approach of this paper is to first make the most of the Deep Q-Network algorithm by tuning hyperparameters and substituting different neural network architectures. Then, we move on to implement a simple variant of Random Network Distillation that is coupled together with the DQN agent.

1.3. Data Description

As with other reinforcement learning environments, the Obstacle Tower environment also self-generates the training data for the agent. Below is the specification for the environment.

- Episode: the environment consists of 25 floors during Round 1 and 100 floors during Round 2, with the agent starting at level 0. There is a limited time given to the agent which is extended by a small amount when the agent clears each floor, and the episode ends when the remaining time is 0 or the agent dies for various reasons.
- Observation Space: the observation is an image rendered in 168 x 168 x 3 RGB. Note that visual appearance and floor layout is generated in a stochastic manner, making the training more difficult.
- Action Space: the action space is a four-dimensional 3 x 3 x 3 x 2 multi-discrete space, which consists of [forward/backward/no-op, left/right/no-op, clockwise/counterclockwise/no-op, jump/no-op]. For each step, the agent takes a combination of these four dimensions, picking one from each dimension.
- Reward Function: the environment is a sparse-reward environment. A reward of +1.0 is given when the agent completes a floor, and +0.1 is given for obtaining a key, solving puzzles, or opening doors.

1.4. Data Preprocessing

I performed following preprocessing on the environment.

- Observation space: with the default function provided by the Unity, the observation space was downscaled from 168 x 168 x 3 RGB to 84 x 84 x 3 RGB. I further processed the image to 84 x 84 x 1 by converting it to greyscale.
- Action reshaping: in the default setting, there are 3 x 3 x 3 x 2 = 54 available actions for the agent. After running the agent in real-time mode, I realized that vast majority of the actions are not necessary. For example, backward operation, move left operation, and move right operation were all unnecessary since the agent can simply go forward while turning the camera clockwise or counter-clockwise to alter its directions. In addition to these operations, jumping operations combined only with camera turning operations were removed. As a result, I reduced the action space from 54 to 12 without much loss of performance while improving the training speed by almost threefold.
- Reward adjustment: the default reward structure for the environment had +0.1 reward for obtaining a key. Since obtaining the key was crucial to move up the floors, I increased the reward for the key to +0.8.

2. Methods and Experiments

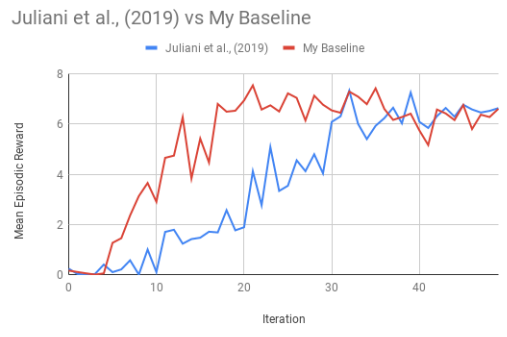
2.1. Overview

Below is an overview of steps I took for the project.

1. Establish human performance standard by manual play.
2. Preprocess the environment through reshaping observation space, action space, and reward function.
3. Implement baseline model in Deep Q-Network using three extensions.
4. Tune hyperparameters for the baseline model.
5. Improve the computer vision of the agent by implementing different convolutional neural network architectures from classic computer vision papers.
6. Substitute policy network with sequential models.
7. Implement a variant of the Random Network Distillation algorithm.

2.2. Baseline Model and Evaluation Criteria

Performance of the agent is evaluated based on the mean episodic reward obtained by the agent. As outlined in the data description, going up a floor is the largest source of reward for the agent. For reference regarding the floor, getting to the 5th floor gives an average of +6.8 mean episodic reward. For the human performance standard, I was able to get to the 25th floor without much difficulty after playing it for several times. For the baseline model, I modified Google's Dopamine library (Castro et al., 2018) to customize a Deep Q-Network agent (Mnih et al., 2015) with priortized replay, distributional network, and multistep updates.

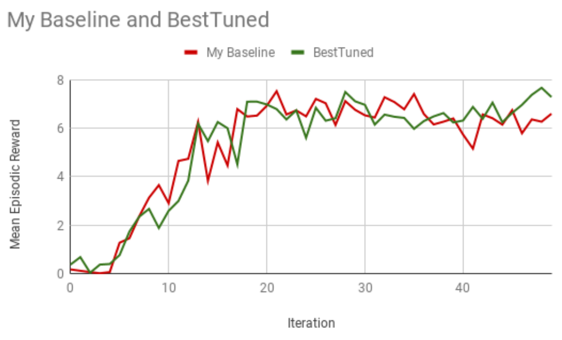


Blue plot is the mean episodic reward of the baseline model following the Obstacle Tower environment white paper (Juliani et al., 2019), and red plot is my implemented baseline model. By preprocessing the observation space, reducing the action space, and reshaping the reward values, I was able to achieve faster convergence without the loss of performance. The baseline model showed an average reached floor of 5.40 with mean episodic reward of 6.61.

2.3. Hyperparameter Tuning

I tuned the following hyperparameters of the baseline agent:

- Learning rate of the Adam optimizer
- Type of optimizer: Adam or RMSProp
- Number of atoms for distributional Q-values
- Minimum required experience replay history
- Experience replay batch size
- Epsilon end value and its decay rate for exploration



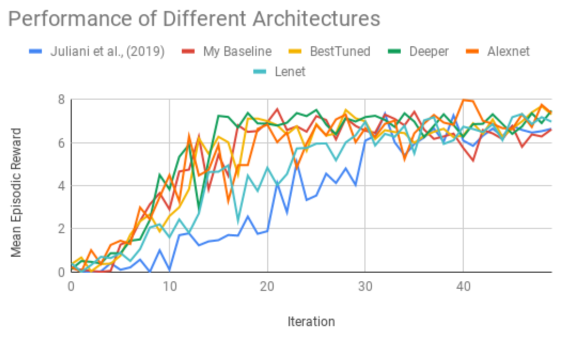
I was only able to try out 10 different combinations of these hyperparameters since training the agent on the Obstacle Tower environment was a very computationally expensive process that took at least 4 days on Nvidia Tesla K80. Unfortunately, all 10 combinations showed almost no improvement over the baseline model. The best tuned combination in the above plot only shows a negligible amount of improvement.

2.4. Improving Neural Network Architecture

Next, I tried to improve the neural network architecture of the agent to see if better computer vision could improve the agent's performance. I implemented eight different neural network architectures that I learned in CS230 course. For architecture borrowed from existing papers, I only used the main underlying concept rather than using the exact same layers due to the computational cost of training the agent as well as different pixel size requirements. For example, for Alexnet, I borrowed the concept of max pooling layer but removed a few repetitive 3x3 layers to ensure the training ran on time. Below is a list of algorithms I tried.

- Basic model: the default model used the same architecture as the one in the Deep Q-network paper published in the Nature, that has three convolutional layers followed by two fully connected layers.
- Deeper model: added a few more convolutional layer and increased the hidden unit size.
- Lenet: similar to Lenet, I incorporated average pooling layers into the network.
- Alexnet: similar to Alexnet, I incorporated max pooling layers into the network.
- VGG: similar to VGGnet, I tried making the network much deeper, although not as deep as the original VGG.
- Resnet*: customized the Resnet50 architecture.
- Inception*: customized the Inception architecture.
- Inception-Resnet*: customized the Inception-Resnet architecture.
  (*modified Tensorflow Slim models)

Main problem with trying out deeper neural network architecture was the computational cost. The basic model, for example, took 93 minutes on average per iteration, totaling 78 hours to train. Neural network architecture similar to Alexnet took 130 minutes per iteration, totaling 108 hours. The simplfied VGG network took 178 minutes per iteration, totaling 148 hours. When I tried to train neural networks based on Resnet, Inception, and Inception-Resnet, it was impossible to finish the run within a week. Therefore, I had to stop halfway after monitoring the interim performances.



More importantly, and unfortunately as we can see above, all these architectures did not show any significant improvements. All three deeper architectures did show higher mean episodic reward in general over the baseline model, but the improvement was negligible to the point that they were not differentiable from the hyperparmeter-tuned model. Therefore, I concluded that computer vision of the agent was not the main problem why the agent was stuck at the fifth floor.

2.5. Human Play Manual Analysis

In order to find out why the agent was stuck at the fifth floor, I decided to examine the environment more thoroughly by playing it several times. Below is what I observed.

- From the 5th floor, the agent has to obtain the key to get to the next floor.
- From the 7th floor, the key is usually in the air so the agent has to jump to obtain it.
- From the 10th floor, the agent has to move a box toward a designated space to open the door.
- From the 18th floor, the key is on a vertically moving tile so the agent has to jump at the right timing.
- From the 19th floor, there is a revolving obstacle that may push the agent away.
- From the 22nd floor, the agent could die from falling into the pit.

In summary, the agent was not able to pass the fifth floor because it couldn't make an inference on the relationship between getting the key and opening the locked door. This was clearly not a computer vision problem, but rather a planning problem. Although I stacked four consecutive frames in the observation space to incorporate the temporal information following the same methodology in the original Deep Q-network paper (Minh et al., 2015), apparently it was not enough for this issue. Therefore, I tried the sequential models.

2.6. Sequential Models

At this point, I implemented the Proximal Policy Optimization (PPO) algorithm by customizing OpenAI's Baseline library, and found that the training time of a naive PPO agent is half of the time required for the baseline DQN agent. Therefore, I combined Long short-term memory (LSTM) policy network with PPO agent instead of DQN agent. Naively using the LSTM policy did not produce a good result, since it was difficult to efficiently extract features from the 84 by 84 pixel space without using the CNN. Therefore, I combined CNN with LSTM.

The results showed a good improvement over the existing CNN-only implementations, with agent getting to the sixth floor for the first time with approximately 1.00 to 1.50 more mean episodic rewards than the baseline model. I also tried normalizing the layers of the LSTM network, but that did not produce better results. Full result is shown in the performance result section due to page limitations.

2.7. Random Network Distillation

While I was running the sequential models, I performed another literature review. The Obstacle Tower only gives out +1.0 reward once the agent reaches the next floor, so I tried to find a good algorithm that can tackle the sparse reward problem. I found that OpenAI published a paper last winter on using Random Network Distillation, and the algorithm scored a state-of-the-art performance in Montezuma's Revenge, which is another challenging environment with sparse rewards. (Burda et al., 2018)

The problem of sparse reward is that the agent loses the motivation to explore new states in the absence of immediate or at least near-immediate reward of doing so. Random Network Distillation solves this problem by introducing a synthetic reward that is given to the agent when it explores a novel state. This synthetic reward is called an intrinsic reward as opposed to the extrinsic reward given by the environment. (Burda et al., 2018)

In a nutshell, the authors of the paper added two neural networks - target and predictor network - to calculate the intrinsic reward. At each step, the observation goes through the target network whose parameters are frozen after being initialized randomly. In other words, the target network produces an embedding of the observation space. Goal of the predictor network is to receive the same observation space as an input and output an embedding that is as close as possible to the output of the target network. As the training continues, predictor network becomes better at producing a similar output to the target network. If the given observation space has already been seen by the predictor network, the loss defined by mean squared error between the outputs of two networks will be small, while a novel state that was never seen before by the predictor network will have a big loss. Therefore, the loss itself can be used as an intrinsic reward. (Burda et al., 2018)

2.8. New Variant Implementation

The initial paper published by OpenAI addressed how the Random Network Distillation can be applied to policy gradient methods. The authors (Burda et al., 2018) explicitly mention that the method "can be used with any policy optimization algorithm," (p.2) but does not address whether it will work on value-based agents such as the Deep Q-Network. None of the 30 papers that cited this paper addressed the application of this algorithm on value-based agent, so I decided to try out this new variant of Random Network Distillation algorithm on the Deep Q-Network.

I initially approached this problem by naively following the paper, by implementing target and predictor networks to produce the intrinsic reward. After training the agent for couple days, I realized that the agent's performance actually deteriorated a lot to the point where it could hardly get to the second floor. I thought there was a bug in my code but extensive debugging did not find anything. After a few days, I realized that it was due to the experience replay feature of the Deep Q-Network.

Experience replay feature enables the Deep Q-Network agent to learn from memory, by storing recent experiences into a memory buffer and sampling a batch to perform an update. (Mnih et al., 2015) This greatly helps the agent's stability by effectively breaking the correlation among consecutive steps of experiences, but it was working as a main culprit for the Random Network Distillation algorithm's failure. Briefly speaking, when I save the [state, reward, action, next state] pairs to the experience replay, the intrinsic reward that has been added to the reward value was representing the novelty of the state at the time of the memory save. By the time the experience was sampled from the replay buffer for an update, the state may or may not be novel anymore.

To fix the issue, I first tried minimizing the experience replay buffer capacity but that only had a marginal effect. Then, I rewrote the implementation to calculate and add the intrinsic reward after the experience has been sampled from the experience replay buffer, rather than when the experience is entered into the buffer. In addition, I normalized the observation space with running mean and standard deviation, and normalized the intrinsic reward by dividing it with its running standard deviation just as the authors of the Random Network Distillation did. (Burda et al., 2018) This led to a working Random Network Distillation with Deep Q-Network.

3. Performance Result

Computation cost was a big hurdle for this project, since full training took about four days for naive baseline models. For complex models such as the Random Network Distillation with Deep Q-Network, it took more than ten days to get the interim results. (I used about $1,700 credits in Google Cloud Platform in addition to using two RTX 2080 Ti GPUs on the personal computer) Therefore, some implementation were halted in the middle if it did not show much promise in the early stage of training, and some others are still running at the time of writing this report.

As the average mean reward value for five episodes were not stable, I used 50 episodes to calculate the final performance numbers reported here. Tuning the hyperparameters and using different CNN architectures showed marginal improvement over the baseline model. Using only the sequential model, LSTM, as the policy network failed to deliver any performance while using CNN feature extraction with LSTM greatly outperformed any runs that only used CNN. For the Random Network Distillation implementation, not normalizing the intrinsic reward or the observation space did not show much promise, while normalizing both the intrinsic reward and the observation space is showing great improvement over all other algorithms even though it is still in the interim stage of learning.

Algorithm	Architecture	LR	Action	Performance
Default Paper Rainbow	Nature	0.0000625	54	6.64
Baseline DQN	Nature	0.0000625	12	6.61
Best Tuned DQN	Nature	0.0001	10	7.31
Deeper DQN with one more conv layer	Nature	0.0001	12	7.48
Lenet-like DQN with avg pooling	Lenet	0.0001	12	6.97
Alexnet-like DQN with max pooling	Alexnet	0.0001	12	7.37
VGG-like like	VGG	0.0001	12	N/A
Resnet-like like with residual blocks	Resnet	0.0001	12	N/A
Inception-like like with inception blocks	Inception	0.0001	12	N/A
Inception-resnet-like like	Inc-Resnet	0.0001	12	N/A
RND DQN, raw IR and states	Nature	0.0001	12	0.52
RND DQN, normalized IR	Nature	0.0001	12	3.14*
RND DQN, normalized IR and states	Nature	0.0001	12	8.31*
Baseline PPO	CNN	0.00025	12	6.78
PPO with reduced action space	CNN	0.00025	8	6.72
PPO with tuned with stacked frame	CNN	0.00025	10	6.98
PPO with LSTM	LSTM	0.00025	10	1.41
PPO with CNN and LSTM	CNN/LSTM	0.00025	10	8.27
PPO with CNN and LN-LSTM	CNN/LNLSTM	0.00025	10	7.01
				* Interim Results

4. Conclusion and Future Work

In this project, we first implemented a baseline model using Deep Q-Network and tuned the hyperparameters. Then, we implemented different CNN architectures for the agent. The results only showed marginal improvement, showing that the agent's inability to move beyond the fifth floor did not originate from its computer vision capacity, but rather from its inability for long-term planning in the absence of dense rewards. Using a sequential LSTM model with CNN extractor to solve this issue showed a good improvement.

In addition, we implemented a simple variant of the Random Network Distillation algorithm that was originally applied to the policy gradient methods, by applying it to value-based Deep Q-Network. The training computation has not yet finished despite running for more than ten days on 16 CPUs with 2 Nvidia Tesla K80 GPUs, but interim results already exceeded all previous methodologies that were implemented in this project.

The competition is expected to end on July 15th, so I will continue my efforts to improve my agent. The next step would be to tune the hyperparameters of the Random Network Distillation model. Furthermore, I am currently reviewing the paper on the Go-Explore, (Ecoffet et al., 2019) which was published recently to beat the performance of the Random Network Distillation on Montezuma's Revenge.

5. Contributions

I was the only member of my team. Since I forked two different repositories, detailed explanation of which part is my own code is supplied in the README.md file here: https://github.com/hanschoi86/cs230

References

Bellemare, M. G., Dabney, W., & Munos, R. (2017, August). A distributional perspective on reinforcement learning. In Proceedings of the 34th International Conference on Machine Learning-Volume 70 (pp. 449-458). JMLR. org.

Burda, Y., Edwards, H., Storkey, A., & Klimov, O. (2018). Exploration by random network distillation. arXiv preprint arXiv:1810.12894.

Castro, P. S., Moitra, S., Gelada, C., Kumar, S., & Bellemare, M. G. (2018). Dopamine: A research framework for deep reinforcement learning. arXiv preprint arXiv:1812.06110.

Ecoffet, A., Huizinga, J., Lehman, J., Stanley, K. O., & Clune, J. (2019). Go-Explore: a New Approach for Hard-Exploration Problems. arXiv preprint arXiv:1901.10995.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., ... & Silver, D. (2018, April). Rainbow: Combining improvements in deep reinforcement learning. In Thirty-Second AAAI Conference on Artificial Intelligence.

Juliani, A., Khalifa, A., Berges, V. P., Harper, J., Henry, H., Crespi, A., ... & Lange, D. (2019). Obstacle Tower: A Generalization Challenge in Vision, Control, and Planning. arXiv preprint arXiv:1902.01378.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Petersen, S. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529.

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized experience replay. arXiv preprint arXiv:1511.05952.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.

Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., & De Freitas, N. (2015). Dueling network architectures for deep reinforcement learning. arXiv preprint arXiv:1511.06581.

Libraries

For repositories where authors requested their white paper to be cited, the white paper is cited with github link.

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Kudlur, M. (2016). Tensorflow: A system for large-scale machine learning. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16) (pp. 265-283). https://github.com/tensorflow/tensorflow

Castro, P. S., Moitra, S., Gelada, C., Kumar, S., & Bellemare, M. G. (2018). Dopamine: A research framework for deep reinforcement learning. arXiv preprint arXiv:1812.06110. https://github.com/google/dopamine

Chollet, F. (2015) Keras, GitHub. https://github.com/fchollet/keras

Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., ... & Zhokhov, P. Openai baselines (2017). https://github.com/openai/baselines.

Hill, A., Raffin, A., Ernestus, M., Gleave, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plap-pert, M., Radford, A., Schulman, J., Sidor, S., & Wu, Y. (2018) Stable baselines. https://github.com/hill-a/stable-baselines

Juliani, A., Khalifa, A., Berges, V. P., Harper, J., Henry, H., Crespi, A., ... & Lange, D. (2019). Obstacle Tower: A Generalization Challenge in Vision, Control, and Planning. arXiv preprint arXiv:1902.01378. https://github.com/Unity-Technologies/obstacle-tower-env

McKinney, W. (2010, June). Data structures for statistical computing in python. In Proceedings of the 9th Python in Science Conference (Vol. 445, pp. 51-56). https://github.com/pandas-dev/pandas

Van Der Walt, S., Colbert, S. C., Varoquaux, G. (2011). The NumPy array: a structure for efficient numerical computation. Computing in Science Engineering, 13(2), 22. https://github.com/numpy/numpy