# EXPLORATION OF OBSTACLE TOWER USING RANDOM NETWORK DISTILLATION

A PREPRINT

**Kendrick Tan**

May 16, 2019

**ABSTRACT**

The issue of sparse rewards is a major challenge for reinforcement learning problems. Regular epsilon-greedy exploration may be successful in refining policies that have frequent feedback, but when rewards are sparse, as in the case of the Obstacle Tower domain, it is likely that a reward-maximizing policy will never be found. Thus, I will try to implement a type of curiosity-driven exploration, Random Network Distillation, to try and tackle the Obstacle Tower problem. Random Network Distillation adds a bonus intrinsic reward to the DQN, specifically the error of a neural network predicting the features of the observation given by a randomly initialized neural network. This added reward biases the agent towards "surprise", or observations that are less encountered, as sub-goals within the problem. In the case of Obstacle Tower, it will hopefully be able to solve multiple levels, something a vanilla DQN cannot accomplish.

## 1 Introduction

Obstacle Tower is a domain recently released by Unity Technologies, aimed at serving as a Deep Reinforcement Learning benchmark now and in the future

The domain is a game with 25 floors, where the agent starts off at floor 0. There is a room in each floor that has stairs that lead to the next floor, and each room has some sort of problem/task that needs to be solved by the agent. The complexity of of the tasks increase as the agent moves up the tower, and the episode ends when either the agent dies (by falling through a hole, for example), or time runs out. The agent is initially given just enough time to make 600 actions to solve the entire game, but reaching a new floor, and collecting blue orbs laid out throughout the game, grants extra time.

The domain also has the following features:

- **Procedurally generated levels/tasks and graphics:** generalization is needed to consistently perform well in this environment, as each instance of the task has slight variations in layout of rooms, the tasks in each room, as well as the lighting conditions and textures of objects.
- **First-person, partially observable states:** observations are pixels that resemble the state from the perspective of the agent. Thus, the state space is very partially observable, making even simple navigation tasks difficult. It is meaningful to solve problems from the perspective of the agent, as solutions can potentially translate to problems in robotics, where input images come from the perspective of the robots.
- **Large, discrete actions space:** four sub-spaces within the action space result in a total of 54 possible actions.
- **Sparse rewards:** A reward of +1 is given hen the agent reaches the exit stairs of a level, and a reward of +0.1 is given when the agent successfully solves a mini task (opening a door, picking up a key, moving a box to the correct location, etc.)

Of the features listed above, the element of sparse rewards is most important to this paper. Introducing methods that tackle the problem of sparse rewards in a complex domain tackles a critical challenge posed by reinforcement learning. An important element of reinforcement learning is that for every input frame, the target label is not known, which makes

it a machine learning technique that can be applied to more complex problems. However, complex problems often come with very sparse rewards, since in reality, it is often impractical to engineer dense reward functions for every task one wants an RL agent to solve. This makes it very difficult for the neural network to evaluate and refine a policy. In the case of the obstacle tower domain, a vanilla DQN is unable to solve more than the first couple of levels, as rewards are too sparse.

One of the methods developed to deal with sparse rewards is intrinsic motivation. Curiosity-driven methods of intrinsic motivation, in particular, have been shown to perform well on sparse-reward tasks involving 3D environments. Because Obstacle Tower, and any other complex real-world RL problem would likely involve sparse rewards, successfully improving performance in such environments using intrinsic motivation is an important step forward in AI.

Attempting to solve Obstacle Tower using the most basic Deep RL algorithm, DQN, with epsilon-greedy exploration and without any sort of intrinsic reward, poses obvious problems. Epsilon-greedy exploration employed by the most basic DQNs is a model of exploration that falls short in solving the problem of sparse rewards. It encourages an agent to explore as much as possible in earlier episodes, and slowly explores less as training goes on, eventually completely adopting the policy. However, performing random actions early on during training often results in no reward being earned, in the case of the Obstacle Tower environment. When an agent receives no reward, it is unable to learn anything from the episode. This makes learning really slow, and with limited feedback on how the policy is performing, this type of exploration often may not result in meaningful solutions.

Random Network Distillation is an algorithm that uses an exploration bonus, in the form of curiosity, as an intrinsic reward. RND is easy to implement, has proven to work well with high-dimensional observations, and can be computed efficiently, as it only requires one single forward pass in a neural network on each batch.

RND implements intrinsic motivation in the form of curiosity, meaning it incentivize the agent to explore states that are unfamiliar to the agent. A state is unfamiliar to the agent if it has not been seen by the agent, and is more unfamiliar if the state is vastly different from previously seen states. Exploring unfamiliar states is beneficial in Obstacle Tower because while most of the environment is kind of similar throughout (brick walls, light), the objectives in the game are always highlighted with color. For instance, as seen in Figure 1, the blue orb that increases the limited amount of time the agent has is significantly different in color from the rest of the environment. It incentivizes this type of exploration by first initializing a random, fixed neural network, called the target network, and then training another neural network, called the predictor network, to emulate the target network. The prediction error of the predictor network is used as an intrinsic reward for the agent, with, along with the extrinsic reward given by the environment, together creates denser feedback. It is desirable for the prediction error to be high, as it means that the state being visited is novel and unlike previously visited states, and thus has not been trained adequately in the predictor network.

Using a model that implements RND on the Obstacle Tower environment, and comparing its performance to a vanilla DQN model that uses epsilon-greedy exploration, shows that RND yields better results in this environment. The DQN model without RND saw extrinsic rewards fluctuate between 0 and 1, meaning the agent either solves no levels or one level, and an average extrinsic reward of 0.11. The average extrinsic reward of 0.11 indicates that the agent solves the first level approximately 10% of the time. The model that implements RND, on the other hand, saw an average extrinsic reward of 0.88, with rewards more consistently reaching over 2 after 1,000 epochs. It is important to note that the models were each trained for approximately 1500 epochs, which could be one reason why the agents were still performing quite poorly. Nevertheless, it was clear that the model that implemented RND performed much better on the Obstacle Tower environment.

## 2 Related Work

There were two common approaches to RL with intrinsic rewards in the form of curiosity:

1. Count-based approach: this approach keeps count of previously visited states, and, and simply gives bigger rewards the fewer number of times a state has been visited.
2. Next-state prediction: in this approach, a model tries to predict the next state, takes an action to move into the next state, and then finds the error of the predicted state. The model tries to minimize prediction error. Initially, the model gives poor predictions to these next states, but eventually the error diminishes for those states, as well as states similar to them, due to the generalization of neural networks. If the prediction error is high, then the state is novel, and should be explored.

The count-based approach because much less effective as the number of possible states grows. In the case of Obstacle Tower, since there are so many possible states, with slight variations to the state space each time the environment is initialized, the inability of this method to generalize across similar states makes it quite ineffective.

Figure 1: A third person view of Obstacle Tower gameplay. Normally, gameplay is through the perspective of the agent, who is on the right side of this photo. The blue orb near the middle of the photo is an object that grants extra time if collected by the agent, and the green door is a door that leads to another room.

The second approach listed above also suffers from a serious drawback, related to model capacity. Attempting to predict the next state of a highly complex environment with a large state space and action space is very difficult, and the model's ability to sufficiently prediction error in most, if not all states, could be compromised. If the model fails at reducing prediction error, the agent could get stuck in areas of the state space that have high prediction errors that cannot be lowered.

## 3 Technical Approach

### 3.1 Random Network Distillation

Random Network Distillation involves two neural networks. First, there is a fixed, randomly initialized target network which defines the prediction problem. Second, there is a predictor network that is trained on the pixel data collected by the agent in Obstacle Tower. The predictor network is trained by gradient descent to minimize the expected MSE $||\hat{f}(x;\theta) - f(x)^2||$, where $\hat{f}$ is the predictor network and $f$ is the target network. This prediction error should be higher for novel states that are unlike states that the predictor has previously trained on.

### 3.2 Sources of Prediction Errors

In general, prediction errors can be attributed to four main factors:

1. Amount of training data. Prediction error is high in states that have not been visited enough. Note that this is what is desired, because it allows us to predict the novelty of a state
2. Stochasticity. Prediction error is high in states with highly variant dynamics.
3. Model misspecification. Prediction error is high when the model lacks the capacity to fit the complexity of the target function.
4. Learning dynamics. Prediction error is high because it is difficult to learn the environment dynamics, and the process fails to converge.

2, 3, and 4 are undesirable, since we want the prediction error to only predict the novelty of a state, and represent as little of anything else as possible. 2 is not a source of error in Obstacle Tower on RND, as both the environment and the target network are deterministic. Since the target network and the prediction network belong in the same model class, they have the same capacity, so this error does not exist in RND either. Finally, since the target network is fixed, the prediction problem is a stationary learning problem, which tends to converge. This is different from a prediction problem that attempts to predict environment dynamics, which is not stationary. Thus, the desirable source of error is the only one present in RND, which makes the error a good tool in predicting the novelty of a state.

### 3.3 Normalization of Rewards and Observations

A problem with using prediction error as an intrinsic reward is that the scale of the reward can differ at different points in time, and at different instances of the environment, in the case of Obstacle Tower. This makes it difficult to choose hyperparameters that would apply to all observations. RND normalizes the intrinsic reward by dividing it by the standard deviations of the intrinsic rewards. Rewards are then clipped to be between 0 and 1.

### 3.4 Dueling Double DQN

The pseudocode for RND provided by OpenAI is shown in Figure 2. While this implementation uses the Proximal Policy Optimization (PPO) algorithm with RND, I have chosen to use an implementation that uses a Dueling Double DQN instead.

A Dueling Double DQN combines the concepts of a Double DQN and a Dueling DQN. The Double DQN was introduced by Hado van Hasselt to handle the problem of overestimating Q-values. Because no information about what the best action to take is early in training, sometimes, non-optimal actions are given a higher Q value than optimal actions. If this happens often enough, learning becomes complicated, and really slow. Thus, two networks are used for the selection of the Q value. First, a DQN network is used to select what the best action to take is for the next state (the action with the highest Q value). Then, a target network is used to find the target Q value of taking the selected action at the next state. The resulting equation looks like this:

$$Q(s,a) = r(s,a) + \gamma Q(s', argmax_a Q(s',a))$$

$argmax_a Q(s',a)$ is calculated by the DQN Network, choosing the best action, and $\gamma Q(s', argmax_a Q(s',a))$ is calculated by the target network, finding the Q value of taking the selected action at the next state.

Dueling DQNs, on the other hand, decompose Q values into a value function that estimates the value of a state, and another that estimates the advantage of each action from a state. Like this:

$$Q(s,a) = A(s,a) + V(s)$$

It then combines these two elements back together through an aggregate layer in the neural network. The reason for this decomposing the Q values, and then recombining them, is that we can learn which states are valuable without having to learn how valuable each action is from that state, since it's calculating $V(s)$.

With a normal DQN, when a state is bad, the values of each action from that state are still being calculated. But this is not very efficient for learning, since the state is already bad. Thus, by calculating $V(s)$ in the Dueling DQN, we can eliminate this inefficiency, since we can already know that the state is bad.

Using a Dueling DQN to find the best action for the next state, then using another neural network to find the Q value of taking the selected action at the next state, combines both algorithms to form a Dueling Double DQN, which is what is used in my implementation of Random Network Distillation.

## 4 Evaluation

We want to get the agent to solve more levels in the Obstacle Tower domain. One of the main issues of the environment is the issue of sparse rewards, and the goal is to combat this issue using Random Network Distillation, and hopefully perform better than a vanilla DQN model.

I evaluate the performance of the RND model using the average extrinsic reward of the first 1500 episodes or so. Comparing average extrinsic rewards allowed me to compare the performances of the RND model and then vanilla DQN model. On top of that, video recordings of the agent after 1500 epochs were compared to verify the results indicated

**Algorithm 1** RND pseudo-code

$N \leftarrow$ number of rollouts
$N_{\text{opt}} \leftarrow$ number of optimization steps
$K \leftarrow$ length of rollout
$M \leftarrow$ number of initial steps for initializing observation normalization
$t = 0$
Sample state $s_0 \sim p_0(s_0)$
**for** $m = 1$ **to** $M$ **do**
  sample $a_t \sim \text{Uniform}(a_t)$
  sample $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$
  Update observation normalization parameters using $s_{t+1}$
  t += 1
**end for**
**for** $i = 1$ **to** $N$ **do**
  **for** $j = 1$ **to** $K$ **do**
    sample $a_t \sim \pi(a_t|s_t)$
    sample $s_{t+1}, e_t \sim p(s_{t+1}, e_t|s_t, a_t)$
    calculate intrinsic reward $i_t = \|\hat{f}(s_{t+1}) - f(s_{t+1})\|^2$
    add $s_t, s_{t+1}, a_t, e_t, i_t$ to optimization batch $B_i$
    Update reward normalization parameters using $i_t$
    t += 1
  **end for**
  Normalize the intrinsic rewards contained in $B_i$
  Calculate returns $R_{I,i}$ and advantages $A_{I,i}$ for intrinsic reward
  Calculate returns $R_{E,i}$ and advantages $A_{E,i}$ for extrinsic reward
  Calculate combined advantages $A_i = A_{I,i} + A_{E,i}$
  Update observation normalization parameters using $B_i$
  **for** $j = 1$ **to** $N_{\text{opt}}$ **do**
    optimize $\theta_\pi$ wrt PPO loss on batch $B_i, R_i, A_i$ using Adam
    optimize $\theta_{\hat{f}}$ wrt distillation loss on $B_i$ using Adam
  **end for**
**end for**

Figure 2: Pseudocode for Random network Distillation, found in the OpenAI paper.

by the average extrinsic rewards. This comparison highlights the effectiveness of Random Network Distillation in improving the training results of agents in the Obstacle Tower environment.

The results of this experiment are shown in Figure 3. Average (extrinsic) reward for the vanilla DQN model was approximately 0.11, while average reward for the Random Network Distillation model was approximately 0.88. With the vanilla DQN model, the agent successfully completed two levels once in 1500 epochs. With the RND model, the agent successfully complete two levels pretty consistently after about 1200 epochs. Note that the extrinsic reward is never at exactly 2, as completing the second level requires the successful completion of at least one obstacle/task, which gives a reward of +0.1 each. From the graphs, we can see that after 1500 epochs, while the RND model pretty consistently solves 1 and sometimes 2 levels in the environment, the vanilla DQN model does not exhibit much improvement over time, solving the first level approximately 10% of the time.

Qualitatively, I noticed that even after 1500 epochs, the agent trained on the vanilla DQN model made movements that seemed pretty much random, similar to how the agent made decisions when training began. This validates the quantitative data showing that the average extrinsic reward obtained by the agent stays relatively constant through the first 1500 epochs (the orange line in Figure 3). On the other hand, the agent trained on the RND model, while making obviously suboptimal actions, noticeably worked its way towards the exit stairs of the first level each time, getting there

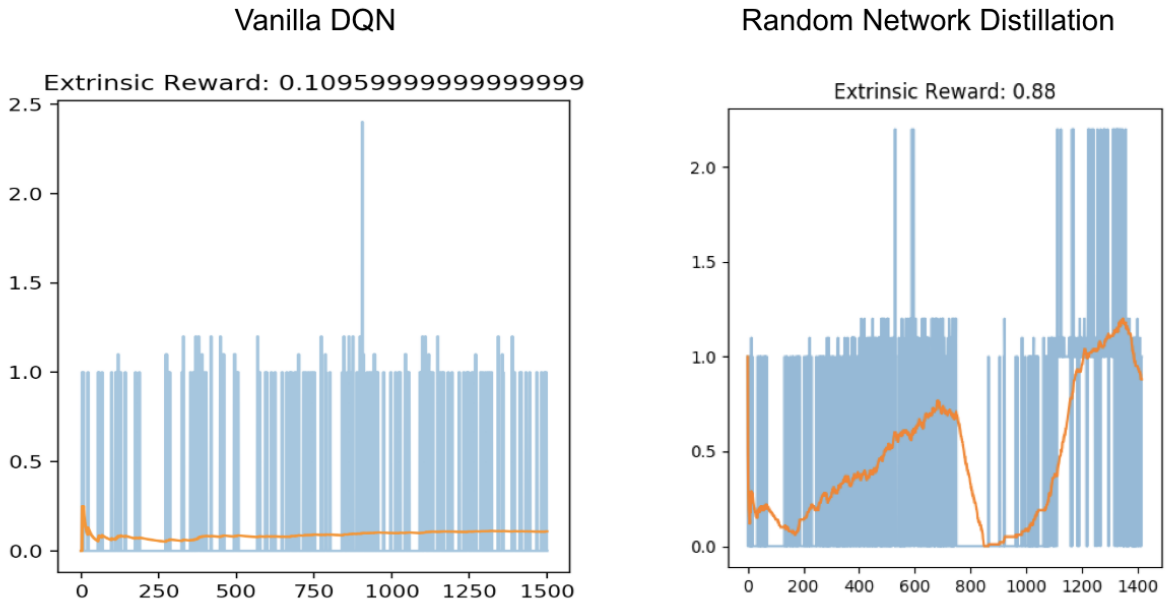


Figure 3: Graphs showing the extrinsic reward obtained each episode. The y-axes represent the extrinsic reward, and the x-axes represent the epoch. The average extrinsic reward for each model is displayed at the top, and the orange line depicts the average extrinsic reward at each epoch.

most of the time before time ran out. It was clear that the agent found it desirable to reach the exit stairs, through its movements.

## 5 Conclusion

I compare the performances of a vanilla DQN model with a model that implements RND on the Obstacle Tower environment. Obstacle Tower is a highly complex environment that demands generalization and has very sparse rewards. Random Network Distillation, which is a method of implementing intrinsic motivation in the form of curiosity, combats the problem of sparse rewards by providing feedback to the agent at each state, encouraging the agent to explore states that are dissimilar to already explored states. The model that implements RND performs considerably (approximately 8 times) better than the vanilla DQN model after 1500 epochs.

The Obstacle Tower domain poses many more difficult problems that remain unsolved in Reinforcement Learning. After 1500 epochs, RND was only able to solve 1-2 levels consistently. Seeing that successful results observed by the OpenAI paper on RND necessitated 200,000 to 400,000 epochs, it is clear that more training is needed to potentially observe better results.

Another glaring problem in the environment was that the action space is too large, and after looking at video recordings of gameplay, I observed that the agent was jumping and turning too frequently, even when the environment did not demand any kind of jumping at all. Under the current action space, half of the possible 54 actions involve jumping, which is definitely not a smart way to explore the environment, especially in the earlier levels where jumping is not necessary at all. Thus, reducing/modifying the action space such that actions that involve jumping are much more limited, and thus happen at much lower frequencies, could potentially be a quick fix that could see noticeable improvement in performance.

One area of research that could be explored is hierarchical control. In the Obstacle Tower environment, the goal of the agent is to reach as high of a level as possible. This goal can naturally be separated into at least two levels of tasks: solving the puzzles in the rooms, and navigation between the rooms. THe structure of the environment is a good opportunity to explore hierarchical control methods, such as the use of a high-level planner to solve floors, and a low-level controller that navigates obstacles within a room. Moreover, since many of the skills and tasks in the game do repeat, such as opening doors, collecting orbs, navigation, and collecting keys, the acquisition of options, or skills,

that represent high-level plans to solve such tasks, could be a promising area of research to consider for improved performance in this domain.

## References

[1] Arthur Juliani and Ahmed Khalifa and Vincent-Pierre Berges and Jonathan Harper and Hunter Henry and Adam Crespi and Julian Togelius and Danny Lange. Obstacle Tower: A Generalization Challenge in Vision, Control, and Planning In *arXiv preprint arXiv:1902.01378*, 2019.

[2] Yuri Burda, Harrison Edwards, Amos Storkey, Oleg Klimov. Exploration by Random Network Distillation In *arXiv preprint arXiv:1810.12894*, 2018.

[3] Hado van Hasselt, Arthur Guez, David Silver. Deep Reinforcement Learning with Double Q-learning In *arXiv preprint arXiv:1509.06461*, 2018.