Connecting the Dots
Interpretable RL Environment for Sequential Planning

Deven Parekh DEVEN.PAREKH@MAIL.MCGILL.CA
McGill University, Canada

Koustuv Sinha KOUSTUV.SINHA@MAIL.MCGILL.CA
McGill University, Canada

Abstract

We introduce a new environment, Turtle Learning Environment (TLE), which is a minimalistic, interpretable and generic environment to learn how to draw images in *connect-the-dots* fashion. We propose this environment as a much more challenging testbed for current reinforcement learning agents to learn how to perform sequential planning, and show the performance of state-of-the-art models. We also show the effect of tuning reward function in the environment and how it affects learning of the agent. Finally, we release the environment as a Gym environment in open source mediums.

1. Introduction

Various environments are available for researchers to test algorithms, however many such environments provide a limited scope to analyze and interpret sequential planning. For example, Sokoban (Junghanns & Schaeffer, 1997) is such an environment where an agent has to push a number of boxes into given target locations. However, the agent is still free to push the boxes in an arbitrary way to achieve the goal, and thus environments do not incentivize intelligent or sample efficient planning. Also, such environments are not inherently interpretable, which also provides less cognitive motivation for the agent to perform a task in an efficient and planned manner. In order to devise such an environment, we took the task of sketching a simple image given some constraints, where the hypothesis is that since the task is challenging, and the agent can be disincentivized to reach the reward by sporadic motion, we can minutely observe how current systems perform in sequential planning and inspect their shortcomings. Recent advancements in Deep Exploration (Osband et al., 2017; 2016; Fortunato et al., 2017) also show the need of such environments which promote efficient planning. Hence, our generalized environment provides such a challenging task of sketching while connecting the dots, which is minimalistic, highly interpretable and extensible. We believe our environment will promote research into sample efficient deep exploration algorithms, and provide a simple yet challenging testbed for evaluation.

Proceedings of COMP 767 Reinforcement Learning Final Project, McGill University, Canada.

2. Turtle Learning Environment

2.1. Initial motivation

We are heavily inspired by the programming language LOGO while developing this environment. LOGO [1] is a Lisp inspired programming language designed to draw shapes in a 2D plane by the use of primitive commands such as *forward* (FD), *backward* (BK) to move *forward* or *backward* by specifying number of pixels to move, and left (LT) and right (RT) to change direction by specifying the degree of rotation. This is ideal for a reinforcement learning agent as we have a definite set of actions to choose from, and the result of those actions are to draw shapes on the screen. The head of the drawing pointer is referred as *Turtle*, and hence the name of our environment. Figure 1 shows some of the basic logo commands.

Initially, we designed a reinforcement learning task in which the agent would learn to draw single digits (0 to 9) in a 28x28 grid using a reward function based

[1] http://el.media.mit.edu/logo-foundation/what_is_logo/logo_programming.html

[3] Image source:http://www.annehelmond.nl/wordpress/wp-content/uploads/2007/11/logo_mit.png

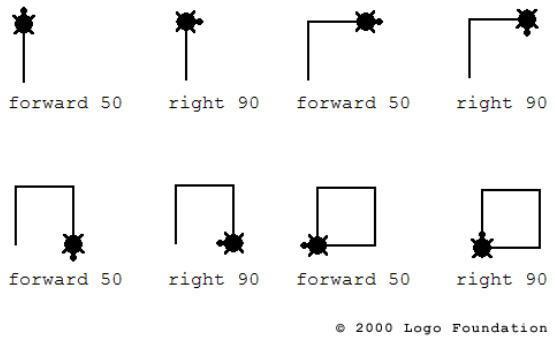


Figure 1. Basic commands in LOGO Programming.[3]

on the confidence score from an MNIST classifier[4]. In this task, the observation or state space is a grayscale grid, where the agent can take the following simple actions:

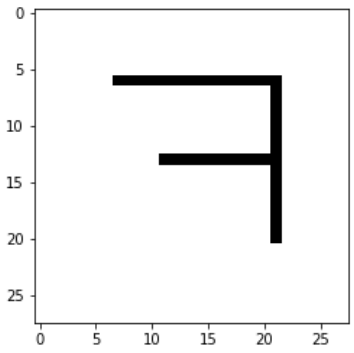
- LT: Turn left by 90 degrees
- RT: Turn right by 90 degrees
- FD1: Move forward by 1 pixel and draw
- FD0: Move forward by 1 pixel without drawing, simulating the *pen-up* mode in LOGO
- STOP: Stop the episode

However, we soon encountered two main problems in using MNIST classifier for the reward function. First, as shown in the Figure 2, the classifier gave a lower confidence score to an image which was fully drawn, while an incomplete image would get a higher score. This poses serious difficulty in using the score in reward function. Ideally, we would like to give higher reward to an image which is closer to the true digit, but the classifier cannot be used for this task. Second, we found that the classifier score is non-deterministic: multiple predictions for same image results in different confidence scores.

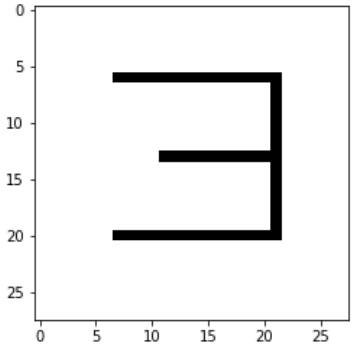
2.2. Generalizing the Environment

We thus conclude that MNIST classifier confidence reward is non-deterministic and is unsuitable to direct an agent to draw. Since drawing itself is a very complex task, we simplified the task by introducing *pivot points*, thus turning the task into a *connect-the-dots* game. This allows us to devise a deterministic reward

[4]We used the MNIST classifier from Github Project: https://github.com/aaron-xichen/pytorch-playground



(a) MNIST Confidence: 0.884



(b) MNIST Confidence: 0.607

Figure 2. Problem with confidence score obtained from the MNIST classifier. The fully drawn image (b), which looks more similar to the digit 3, is given lower confidence score than the incomplete image (a).

function, which have a positive reward for each components connected by drawing a line between two dots. Also, this generalizes the objective by allowing heavy extensibility so that more patterns can be added to the grid to make the drawing more interesting and challenging than drawing digits. We kept the action space same as before, but removed the STOP action.

While this makes the environment simple being a grid-world, we introduce positional markers that will help a function approximator policy like a Convolution Neural Network to pickup relevant signals. We convert the grayscale image into an RGB image with different color markers for tracing a line (cyan), connecting two dots (black), and different colors for rotating the head of the turtle (red, green, blue, yellow). Figure 3 shows initial state of the environment as well as the expected final state which corresponds to the fully connected

image. For simplicity, we chose the pattern of dots representing the digit 3. However, the environment can be easily generalized using a different pattern.

2.2.1. Choice of Reward function

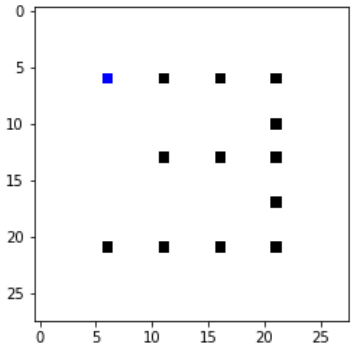
We experimented with various types of reward functions in devising this environment. In general we provide +1 reward for a move action (FD0 or FD1) that results in a connected component, and we can provide either zero or negative reward for each action that does not result in immediate connection. We propose three different reward schemes:

- *Naive Turtle*: Here we provide 0 reward or no penalty for taking any action which does not lead to a connected component. The idea is that since this reward is simplistic, the agent has the motivation to explore the state space more freely.
- *Aggressive Turtle*: Exploring the state space freely might come with a disadvantage to the agent learning to connect two dots after roaming around aimlessly. In order to penalize aimless roaming, we provide a constant negative reward (usually -0.1 or -0.05) to *all actions* that does not lead to two dots being connected.
- *Smart Turtle*: We observed that in the above two schemes the agent is still roaming about aimlessly before connecting a component. To provide motivation to the agent to follow a path, we provide smaller intermediate positive rewards (0.5) whenever the agent reaches a dot irrespective of it leading to connecting the component, while still providing negative reward as the above scheme to penalize unwanted movement. Here the agent should have intrinsic motivation to reach to the dots earlier, which we hypothesize can lead to faster drawing.

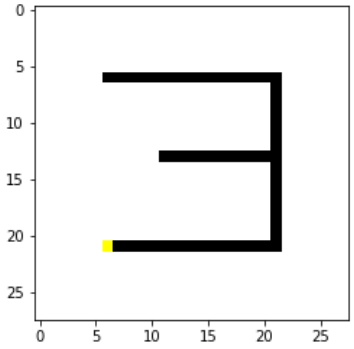
2.2.2. Termination Condition

We terminate the episode when all the components are connected. However, since the state space is large (28x28 grid with 4 possible actions), we observed an episode takes longer than 10,000 timesteps to connect all the dots. For simplicity and training time constraints, we fixed the max timesteps of an episode to 6,000.

We finally release our source code in our Github repository [5]. The codebase is highly extensible so any number of connected components can be made from the environment, and the API is fully compatible to OpenAI Gym environment (Brockman et al., 2016).



(a) Initial State



(a) Expected Final State

Figure 3. Initial and Expected Final state of the Connected Dots Environment

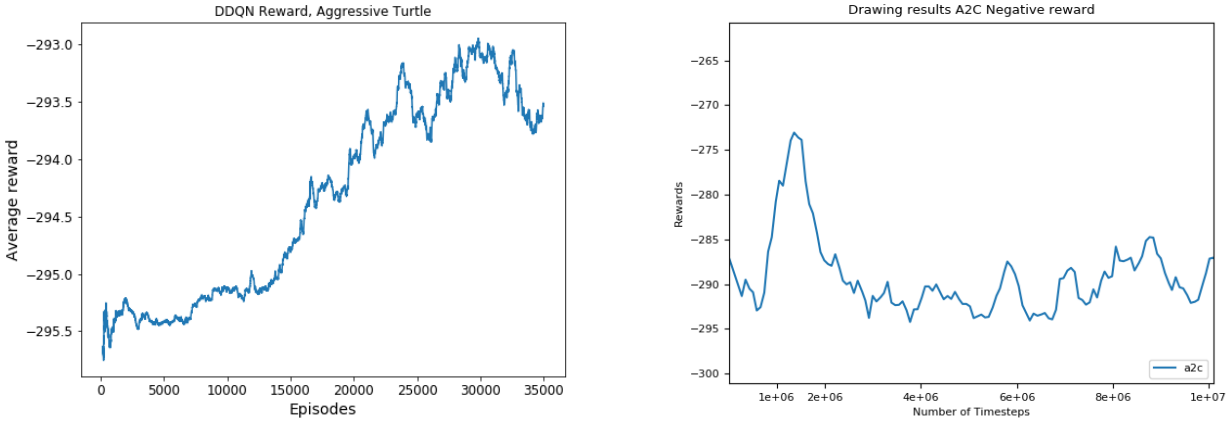
3. Benchmark Results

We aim to provide some benchmark results with respect to current state-of-the-art algorithms. Our motivation to provide these results is to show even in a simplistic 2D environment how challenging it is for the agent to learn to draw. Also, we want to cognitively understand the learned agent's policy in order to open questions regarding possible improvements of exploring in the state space.

We compare two model-free algorithms, a Q-learning algorithm Double Deep Q-Learning (Hasselt, 2010) and a policy-gradient method Advantage Actor Critic (Mnih et al., 2016) 's synchronous version A2C, since we are primarily using a function approximator Convolutional Neural Network policy. The policy is a simple two-layer CNN similar to traditional MNIST classifier.

DDQN is a simple extension to DQN to tackle over-

[5] https://github.com/rllabmcgill/rl_final_project_turtle



(a) DDQN

(b) A2C

Figure 4. Average Rewards for Aggressive Reward, 3000 timesteps per episode

estimation problem in Q-learning, which uses a max operator on the same values to both select and use an action, which leads to overestimation. DDQN thus uses two such networks, one *policy* network to evaluate greedy policy and one *target* network to estimate its value, and they are interchanged after n updates. These algorithms typically use a replay memory to accelerate learning by sampling from past trajectories. On Atari games of the Arcade Learning Environment (ALE) (Bellemare et al., 2013), DDQN has been found to get better policies than DQN.

A3C, on the other hand, is an actor-critic algorithm which learns both the policy and the state-value function, and the value function is used to reduce the variance and accelerate the learning. Here, different parallel actors employ different exploration policies, so replay memory is not utilized. For Atari games A3C is reported to run much faster and performing better or comparably with DQN variants.

3.1. Evaluation Methods

Traditionally the performance of game environments such as Atari games are evaluated by the highest cumulative rewards per episode. In our environment we can also do so, but additionally we also measure the number of components connected per episode. Higher the average connected components are for a learning algorithm the faster it finds all dots to connect. However, we hypothesize that connecting all the components should not be the only evaluation criteria, we should also look at how much the agents deviate from the optimal desired behaviour as a benchmark, which we will study in future work.

3.2. Results & Discussion

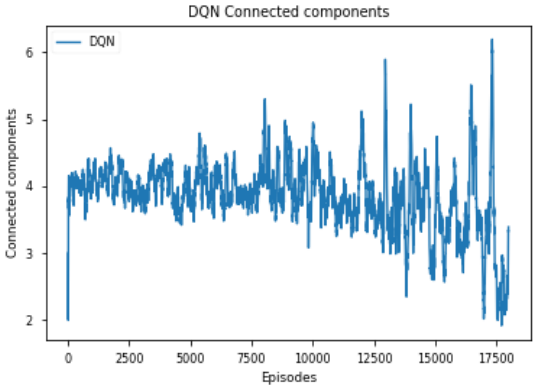
3.3. A2C vs DDQN

We first compared learning performance with A2C and DDQN on our environment. We initially constrained the number of max timesteps for the environment to be 3000, and did a similar comparision with both the methods on *Aggressive* rewards, because we started with this hypothesis that *Aggressive* rewards will motivate the agent to connect all components faster.

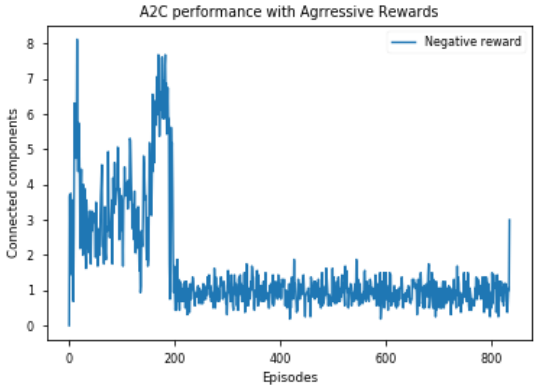
We find that DDQN (Figure 4) has comparable performance with A2C, although A2C learns much faster. However, we notice that 3000 timesteps are not enough for the agent to connect to all dots so we ran a longer experiment with 6000 timesteps per episode and compare the average connectivity, which we show in Figure 5.

3.4. Effect of Reward policies

From Figure 5 we see that A2C is sensitive to negative rewards (Aggressive reward method described earlier). The learning abruptly becomes much worse after it hits a global maximum. We hypothesize that due to high cumulative negative rewards (-300), the value function approximator of the actor-critic algorithm is failing to estimate correct rewards. To test our hypothesis, we ran a comparison between A2C with our Aggressive reward method and Naive reward method, where we only provide 0 reward, which we see in Figure 6. We conclude that our hypothesis is indeed correct and when



(a) DDQN



(b) A2C

Figure 5. Average Connectivity for Aggressive Reward, 6000 timesteps per episode

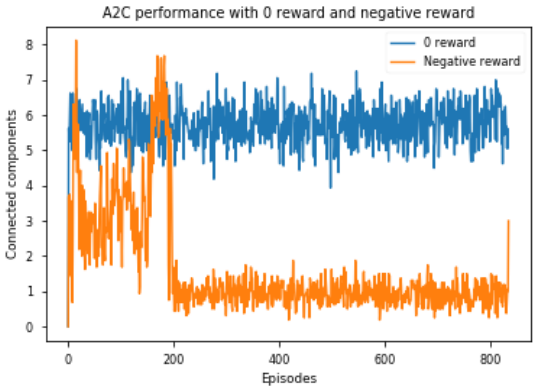


Figure 6. A2C Comparison between Aggressive and Naive reward methods

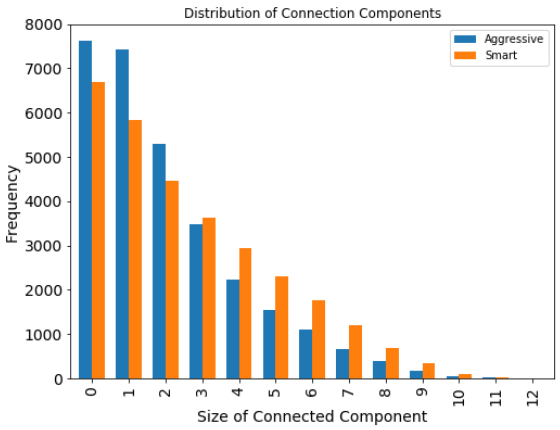


Figure 7. Number of connected components observed while training for Aggresive and Smart reward methods.

our rewards are clipped within 0 and +13 for the connected components the algorithm is much stable.

Similarly, we compare between Aggressive and Smart rewards, this time with DDQN. Figure 7 shows frequency of connected components observed during training. We find a marked increase in the connected components with the change of reward function, thus displaying the effectiveness of choosing a proper reward function.

3.5. Learning Progression

Since our environment is minimalistic and highly interpretable, this provides us an opportunity to actually observe how the agent is learning to connect the dots. We take our learned DDQN model and plot our environment grid for two timesteps around beginning and end of training. Interestingly, we see a natural behaviour emerge where the agent is trying to draw longer lines after it has learned to maximize the rewards, which provides a good insight into the cognitive behaviour of the agent. We provide details in Appendix A (Figure 8). Algorithms using this environment can thus easily compare their learned agents and discover better strategies for learning.

3.6. Conclusion & Future Work

In this work we present a simple, interpretable, extensible yet highly challenging environment. We describe simplicity of the environment (observation and action spaces) and we open-source our code to show it is highly extensible to any patterns of dots. We found

that it is a challenging domain by showing the performance of two different model-free approaches on the environment. We also further show its interpretabilty by plotting the intermediate states of the grid and we try to discover agent's learning patterns and strategies.

There is a host of related work which can be done on this environment. Specifically, we propose conditional generation as a future work where the agent should have the true (expected) image as a reference to connect the dots, which we hypothesize that it would make the training faster. It could also facilitate transfer learning, as after training for several patterns with the actual reference image, it would serve as a learned pre-training of the weights to pick up a new skill.

We hope that our environment will spur more research into developing efficient sequential planning algorithms.

3.7. Contribution

Deven Parekh contributed on development of the environment and DDQN implementation. Koustuv Sinha contributed on testing environment and A2C implementation. Both equally contributed on writing the paper.

References

Bellemare, Marc G, Naddaf, Yavar, Veness, Joel, and Bowling, Michael. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.(JAIR)*, 47:253–279, 2013.

Brockman, Greg, Cheung, Vicki, Pettersson, Ludwig, Schneider, Jonas, Schulman, John, Tang, Jie, and Zaremba, Wojciech. Openai gym, 2016.

Fortunato, Meire, Azar, Mohammad Gheshlaghi, Piot, Bilal, Menick, Jacob, Osband, Ian, Graves, Alex, Mnih, Vlad, Munos, Remi, Hassabis, Demis, Pietquin, Olivier, et al. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

Hasselt, Hado V. Double q-learning. In *Advances in Neural Information Processing Systems*, pp. 2613–2621, 2010.

Junghanns, Andreas and Schaeffer, Jonathan. Sokoban: A challenging single-agent search problem. In *In IJCAI Workshop on Using Games as an Experimental Testbed for AI Reasearch*. Citeseer, 1997.

Kostrikov, Ilya. Pytorch implementations of reinforcement learning algorithms. https://github.com/ikostrikov/pytorch-a2c-ppo-acktr, 2018.

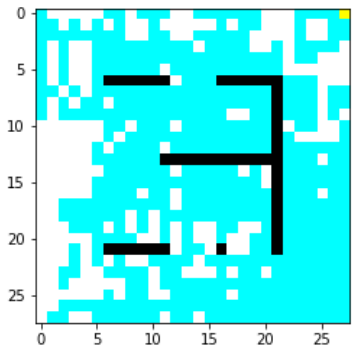
Mnih, Volodymyr, Badia, Adria Puigdomenech, Mirza, Mehdi, Graves, Alex, Lillicrap, Timothy, Harley, Tim, Silver, David, and Kavukcuoglu, Koray. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pp. 1928–1937, 2016.

Osband, Ian, Blundell, Charles, Pritzel, Alexander, and Van Roy, Benjamin. Deep exploration via bootstrapped dqn. In *Advances in neural information processing systems*, pp. 4026–4034, 2016.

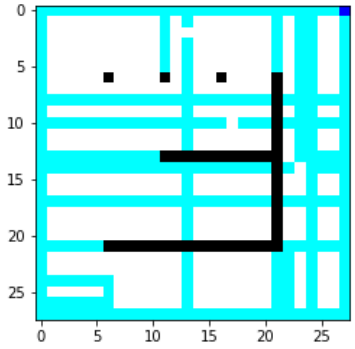
Osband, Ian, Russo, Daniel, Wen, Zheng, and Van Roy, Benjamin. Deep exploration via randomized value functions. *arXiv preprint arXiv:1703.07608*, 2017.

A. Learning Progression

We show the learning progression in the Figure 8 for DDQN algorithm. We observed that in the beginning of the training, the model draws shorter lines, while after it has trained it discovers that drawing a longer straight line often maximizes the scores and connects more components faster.



(a) Around beginning of Training



(b) End of training

Figure 8. Learning progression for DDQN

B. Hyperparameters

B.1. A2C

To train A2C, we followed popular Pytorch(Kostrikov, 2018) implementation, where we trained the agent with nsteps 1, 5 and 10, where we observed nsteps 5 has the highest cumulative rewards. We used the following hyperparameters : learning rate 0.0007 with RMSProp optimizer (default epsilon 1e-5 and alpha 0.99), discount factor 0.99, entropy coefficient 0.01, value loss coefficient 0.5, parameter clipping 0.2, and ran the algorithm on 16 concurrent processes. The policy network is a two layer CNN consisting of 16 and 32 channels, followed by a dropout layer and a dense layer to predict the action, and the critic network is a linear layer on top of the image representation.

B.2. DDQN

To train DDQN, we used 50,000 frames as replay buffer, discount factor 0.99, decreasing epsilon exploration till 0.0001, and Adam optimizer with 0.001 learning rate. Here, we update the target model every 100 episodes, and we train it on GPU with batch size 32. We use a similar 2 layer CNN with 16 and 32 channels, followed by a dense layer.