Boosting Exploration of Low-Dimensional Game Spaces with Stale Human Demonstrations

Kenneth Chang
University of California, Santa Cruz
Santa Cruz, CA
kchang44@ucsc.edu

Adam M. Smith
University of California, Santa Cruz
Santa Cruz, CA
amsmith@ucsc.edu

***Abstract*—Automated game exploration methods benefit from having human demonstration data, but this kind of data is not available for each incremental build of a videogame. If we want to make use of exploration inside of continuous integration (CI) workflows, we need to leverage stale human data (from a recent version of the game). In this paper, we show how to train a goal-conditioned action policy from stale human data used in the context of RRT-based exploration of a modestly changed game version. We demonstrate the benefit of this transfer with experiments in the MiniGrid environment (which has a three-dimensional agent configuration space).**

I. Introduction

Automated testing is used in continuous integration (CI) pipelines for many major videogames [1], [2], however these tests usually do not attempt to map the playable space of games or report how that space has changed, which might reveal unexpected impacts of small code changes [3]. To make this kind of machine playtesting usable within CI processes (where it will be applied to each incremental software change), it needs to be made efficient and *completely* automated. Our previous work showed how to boost the efficiency of exploration using human player data [4], but this is exactly the kind of data that is not available after each software change. We believe that there is useful information to be extracted from stale data (e.g. from a human playtest of a recent build of the game), and that it could be applied to boost machine playtesting. In particular, we propose to use behavior cloning on the stale human data to train an action-selection policy.

We consider exploration algorithms based on rapidly-exploring random trees (RRT) [5], an algorithm from the robotics literature designed for exploring low-dimensional continuous state spaces that has since been applied to exploring videogame spaces [6]. RRT makes use of a subsystem for selecting actions that are intended to make progress towards a given goal state. Recent previous work used very simple policies that were oblivious to the intended goal and even the current state [7]. We propose to use machine learning to fit a goal-conditioned action policy to expert-relabeled human demonstration data (using another technique borrowed from robotics). We offer initial evidence of the usefulness of this approach in the MiniGrid [8] interaction environment, which has a low-dimensional state space with known bounds (similar to the setting for robotics applications).

978-1-6654-3886-5/21/$31.00 ©2021 IEEE

II. Background

A. RRT for Exploratory Game Playing

The Rapidly-Exploring Random Trees algorithm (RRT) has roots in robotics, and its original purpose was to guide a robot in mapping an unknown environment (i.e. *unmotivated* search [9]). During search, RRT builds a tree data structure recording how each newly discovered state was first reached from a previously reached state. To encourage rapid exploration, the algorithm selects goal points uniformly from the low-dimensional state space (e.g. five dimensions for a robot arm with five angular joints) and attempts to reach that goal from the closest existing node.

Outside of robotics, RRT has been used to explore game spaces [6]. Doing so requires a way of mapping a game's true state space (often very high-dimensional with discrete elements) into a form more similar to the robotics setting. Zhan et al. [10] proposed the use of manifold learning techniques to train a state embedding function in the form of a convolutional neural network. The RRT algorithm assumes there is a way to take actions from one state that make progress towards the selected goal state. In prior work [7], the action selection policies were not sensitive to the goal and were even insensitive to the current state. Instead they used a fixed action distribution. This strategy is also used in non-RRT exploration methods like Go-Explore [11]. While our present work uses a hand-crafted state embedding function, we focus our attention on the use of machine learning for fitting a useful and transferable goal-conditioned action policy for RRT.

B. Goal Conditioned Policies for Reinforcement Learning

The idea of learning to take actions in a way that depends on the agent's current state with explicit consideration for the agent's current goal has already been explored in the world of reinforcement learning (RL). Further, it has been shown how to distill unguided demonstration data into goal-conditioned action policies using a strategy called *expert relabeling* [12]. Without making any assumptions on the overall goal being pursued in the demonstration data, the relabeling technique defines a training data set (pairing current and goal states with an action that makes progress towards the goal) by simply

defining the goal to be some other state seen later in the trajectory.

In reinforcement learning research aimed at videogames (e.g. playing StarCraft or Atari games), it is common to evaluate policies on the very same environments used train them (in effect, training on the test data). As a result, trained policies can memorize a single brittle solution rather than capture robust playing styles [13]. In our project, we require that policies accelerate exploration of a somewhat different environment from the one used to source the human demonstration training data.

C. Machine Playtesting

The end goal of our work is to produce a method that can assist in the playtesting of videogames during development. Ideally, developers would be able to immediately see the effects of changing gameplay mechanics, such as the behavior of an enemy or the number of times an ability can be used. For some games, it might be reasonable to directly enumerate the entire space of reachable states [14], however this is unrealistic for more complex games. Current industry practice has evolved to automate many parts of the quality assurance pipeline, and has begun to moved beyond executing fixed scripts (e.g. pressing pre-defined button sequences to traverse menus [1]) towards robust, search-based testing using agents trained on human demonstration data [15]. These intelligent agents are intended to imitate human players playing the game, providing valuable gameplay traces that can be used to analyze the state of the game's mechanics [16], [17].

III. Training a State and Goal-Sensitive Policy

In the setting for machine playtesting for games, we assume we have access to human demonstration data for a recent version of the game. In this data, we record pairs of game states and the human-selected action at that state. A typical human demonstration in our experiments involves 200 steps of gameplay in a turn-based interaction. We transform this into training data for a goal-conditioned action policy using expert relabeling. Concretely, for each point in the demonstration data, we define the goal to be whichever state was reached 1-20 steps further into the demonstration (sampling several random goal offsets). Considering expert relabeling as a form of data augmentation, this typically yields a dataset of about 6,000 examples.

For this early-stage work, our goal selection policy is a simple multi-layer perceptron (MLP) with three hidden layers, illustrated in Figure 1. The inputs to the network are the concatenated state and goal vectors (of three integer dimensions representing the agent's world position and orientation). The output of the network is a distribution over the few discrete actions available (up, down, left, right, and door-toggle). The networks typically have about 40,000 trainable parameters. The training of this model on the data described above using the Adam optimizer proceeds to convergence unremarkably. To create goal-oblivious or even state-oblivious ablations of

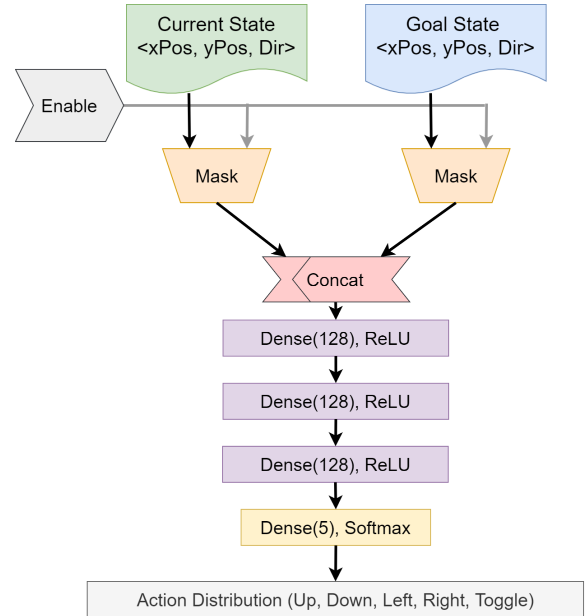


Fig. 1: Neural network architecture used to train an action policy. Masks represent a gate used to hide specific training data.

this network, we mask (scale by zero) the input respective input vectors at both training and evaluation time.

IV. Exploring with RRT

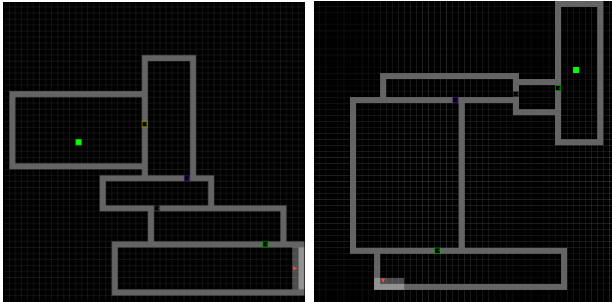
To explore a game space using RRT, we apply the algorithm described in Algorithm 1. For the MiniGrid environment, we implement `sample_goal(configuration_space)` by sampling a grid location and agent orientation using a uniform distribution. We implement `policy(state, goal)` by asking the neural network for a distribution over actions, which we then sample. When moving from one version of a game to the next, the shape of the configuration space and the details of the initial state may change along with the details of the simulator. In our experiments `simulate(state, action` is always implemented by executing the same MiniGrid envronment rules, just with a different level design. We imagine that for many incremental changes, the inputs and outputs data types of the policy function do not change significantly (i.e. the game does not often add or remove spatial dimensions or actions).

V. Experiments with Minigrid Environment

In this early-stage experiment with learning from stale human demonstration data, we focus on the MiniGrid interaction environment [8] (which has been used extensively in recent reinforcement learning research [18]). Figure 2 shows two examples of worlds seen in this environment. To model an incremental change to a game design, we sample two different procedurally generated MiniGrid maps and only provide human demonstration data for the first one. In particular, one of the authors recorded about 200 steps of navigation between

Algorithm 1: rrt_explore(configuration_space, initial_state, simulate, policy, max_steps)

```
tree = new Tree(initial_state)
forall t in range(max_steps) do
    goal = sample_goal(configuration_space)
    state = tree.find_nearest(goal)
    action = policy(state, goal)
    result = simulate(state, action)
    tree.add_edge(state, action, result)
end
return tree
```



(a) Map A (b) Map B

Fig. 2: Two different map designs in the MiniGrid environment. Human demonstration data is made available only for Map A as if Map B had just been produced as an incremental game design change and had not yet been seen by human playtesters. Players move the oriented red triangle through rooms and doors to reach the green exit tile.

the various rooms of Map A, covering some but not all of the total reachable area of the map. The goal of our work is to use the human demonstration data from Map A to improve the efficiency of machine exploration of Map B. Note that the human demonstration data cannot be superficially transferred to the new map because (1) many paths in the first map are not possible in the new map because they would cross walls and (2) the demonstration touches very few tiles in the original map anyway. Our choice of simplified interaction environment and use of level generation to introduce variation is intended as a *computational caricature* [19] of incremental changes to navigation-oriented videogames.

We consider several versions of our RRT-based exploration agent:

- **RRT-Uniform**: Acts using a fixed uniform distribution.
- **RRT-Stateless**: Masking both the current and goal vectors, this model uses a fixed non-uniform distribution fit to the human demonstration data.
- **RRT-StateOnly**: Masking only the goal vector, this model can adapt its action distribution to the current state.
- **RRT-StatesAndGoals**: The full goal-conditioned action policy trained via imitation learning.

These variations were chosen to demonstrate the relative benefit of using stale human data to biasing the action distribution as well as the benefit of considering the current and goal states when doing so.

To measure exploration efficiency, we recorded the number of unique state vectors seen during an exploration run as a function of the number of environment interaction steps. Each exploration algorithm is run three times to reduce the effect of randomness from RRT. Better exploration algorithms will touch many unique state without spending many steps revisiting old states. This metric metric for MiniGrids is comparable to the "tiles touched" metric used for more complex games like Mario and Zelda seen in previous exploration work [4]. We run each variation of RRT for 10,000 interaction steps and report the results in the following section.

VI. Results

In Figure 3a, the RRT agent trained with goal and current state data explores more unique tiles than RRT missing one or both or one of these inputs. As a result of overfitting to the human demonstration data that does not touch many states, the RRT-StateOnly model is unable to outperform RRT-Uniform (a policy that requires no access to human demonstration data). Although none of the exploration algorithms reached the maximum number of tiles that could have been explored within 10,000 steps,[1] it is clear that RRT boosted with goal and current state information can find unexplored areas in a shorter amount of time than the other methods.

When asked to generalize to a fresh map, Figure 3b indicates that both RRT-Stateless and RRT-StateOnly underperform the RRT-Uniform benchmark. Only the combined model RRT-StatesAndGoals, which is trained to mimic the human's navigation style rather than their specific navigation path, is able to beat the benchmark. From this, we conclude that knowledge extracted from stale human data *can* be misleading, an appropriate model architecture and training regime can extract *transferable* knowledge that demonstrably improves exploration efficiency on incrementally changed game designs for which no human demonstrations are available.

VII. Future Work

Where previous work trained a state embedding function from past gameplay data and used a fixed action selection policy, this work used fixed embedding and trained action selection policy. Under the assumption the game design changes incrementally over time, we believe both elements could be fruitfully learned simultaneously, even from collections of partially-stale data. Doing so might extend the benefit demonstrated in this paper to more complex games.

Future work should also seek a source of more realistic incremental software changes to videogames (e.g. commit logs for open source game projects) to understand the types and degrees of changes that would likely be seen in continuous integration (CI) pipelines. This would shed light on the validity of assumptions about the nature of incremental changes used

[1] Experimentally, we confirmed that all methods, including RRT-Uniform, were able to reach the maximum number of unique tiles when given enough time to search (e.g. 100,000 steps).