Play to Grade: Testing Coding Games as Classifying Markov Decision Process

Anonymous Author(s)
Affiliation
Address
email

Abstract

Contemporary coding education often presents students with the task of developing programs that have user interaction and complex dynamic systems, such as mouse based games. While pedagogically compelling, there are no contemporary autonomous methods for providing feedback. Notably, interactive programs are impossible to grade by traditional unit tests. In this paper we formalize the challenge of providing feedback to interactive programs as a task of classifying Markov Decision Processes (MDPs). Each student's program fully specifies an MDP where the agent needs to operate and decide, under reasonable generalization, if the dynamics and reward model of the input MDP should be categorized as correct or broken. We demonstrate that by designing a cooperative objective between an agent and an autoregressive model, we can use the agent to sample differential trajectories from the input MDP that allows a classifier to determine membership: Play to Grade. Our method enables an automatic feedback system for interactive code assignments. Together with Code.org, we curated a dataset of 700k student submissions, one of the largest dataset of anonymized student submissions to a single assignment. This Code.org assignment had no previous solution for automatically providing correctness feedback to students and as such this contribution could lead to meaningful improvement in educational experience for the millions of students who learn through Code.org and beyond.

1 Introduction

The need for high quality education at scale is of critical importance [1]. While delivery of content for millions of students is possible, providing feedback – a cornerstone of high quality education – remains an open challenge. The quality of an online education platform depends on the feedback it can provide to its students. However contemporary coding education has a clear limitation. Students are able to get automatic feedback only up until they start writing interactive programs. When a student authors a program that requires *user interaction*, e.g. where a user interacts with the student's program using a mouse (such as a game), or by clicking on button (such as in a graphical user interface) it becomes exceedingly difficult to grade automatically. Even for well defined challenges, if the user has any creative discretion, or the problem involves any randomness, the task of automatically assessing the work is daunting. This is especially true as coding has become more popular and feedback is required for many students. One popular intro to programming platform, Code.org has over 61 million enrolled students [2]. For their many interactive assignments they have no ability to even identify if a student has a working solution.

Why is providing feedback to interactive programs so hard? The standard way to provide feedback is through human labor. Teachers need to interact with each student's program for a 20 seconds to 10 minutes in order to grade. This may seem small, but for the quantity of unique programs on a platform such as Code.org that amounts to approximately 9.5 years of human effort to provide

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

feedback for a single assignment. Code.org has tried crowdsourcing feedback from hundreds of thousands of teachers which has fallen short for similar reasons: labelling is hard and undesirable work [3]. Interactive assignments can not be auto graded as running the program requires *dynamic* input. Moreover, heavily scaffolded assignments – ones that force students to program their interactive programs in a way that enables standard testing – are considered to be a sub-par learning experience for students and are not common in contemporary classrooms.

There is a long history of work towards providing feedback to students working on open ended assignments which has shown that automatically providing feedback based on code text is a hard machine learning problem [4, 5, 6, 7, 8, 9]. This is true for many reasons. Even for introductory level computer science education, homework datasets have statistical distributions with heavy tails similar to natural language [10]. Moreover, these distributions tend to be highly discontinuous – two solutions which are only slightly different in text can be very different in its behavior. As such, approaches to grading which rely only on reading the text of a students code end up being as complex as understanding a passage of natural language [11]. Finally, because feedback is necessary for the first student working on an assignment, not just the millionth, and new assignments are often introduced, the challenge is inherently few shot.

In this work we show that an algorithm that learns to interact with a students assignment, to *play* with the student work, can enable new capacity for understanding student work. A collaborative system which can simultaneously learn to understand what type of behavior is undesirable, as well as play student's work to actively trigger these undesirable behaviors, is the crucial first step to developing automatic and intelligent feedback for massive online coding education.

Our contributions:

- We introduce the reinforcement learning challenge of *Play to Grade*.
- We propose a collaborative algorithm where an agent simultaneously learns to play a game and recognize what states are bug inducing states.
- Our solution obtains 89%-91.6% accuracy on a real-world coding dataset provided by Code.org. We gained a 15-27 percentage point improvement over grading programs via code text.
- We will release a dataset of over 700k submissions to support further research.

1.1 Related Work

Education feedback The quality of an online education platform depends on the feedback it can provide to its students. Low quality or missing feedback can greatly reduce motivation for students to continue engaging with the exercise [12]. Currently, platforms like Code.org that offers block-based programming use syntactic analysis to construct hints and feedbacks [13]. Using recurrent neural network to understanding programs through code text has also been well-explored, focusing on providing code-level feedbacks or correcting syntactical errors [14, 15]. The current state-of-the-art introduces a method for providing coding feedback that works for assignments up approximately 10 lines of code [10]. The method does not easily generalize to longer and more complicated programs. Our method sidesteps the complexity of static code analysis and instead focus on analyzing the MDP specified by the game environment. Our work is complimentary to these methods because we provide a new tool of analysis for understanding a program's behavior and providing feedback beyond static text analysis.

2 The Play to Grade Challenge

In this section we are going to introduce a novel challenge for the machine learning community: how can an algorithm learn how to "play" with a student submission in order to understand the ways in which it might be buggy. The challenge applies to the task of giving feedback for any interactive student work. In this paper we will focus on games typical of introduction to programming courses.

We formulate the challenge with constraints that are often found in the real world. Given an interactive coding assignment, a teacher often has at least one but possibly a few solution implementations of the assignment. We also assume that the teacher can prepare a few incorrect implementations that represent their "best guesses" of what a wrong program should look like. This sets up a few shot challenge where an algorithm must learn to generalize from the few teacher examples, to the

tens-of-thousands (or more) unlabeled combinations of mistakes and invariances found in natural student code.

Definition 2.1. *A deterministic **Markov Decision Process (MDP)** is a 4-tuple M = <$\mathcal{S}, \mathcal{A}, T, R$>, where $\mathcal{S}$ is a set of states; $\mathcal{A}$ is a set of actions; $T : \mathcal{S} \times \mathcal{A} \rightarrow S$ is the transition dynamics; and $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the reward function.*

We can consider each student implementation of the interactive assignment as a separate MDP. Shared structures between these MDPs can be leveraged to train a grading algorithm. The first assumption that we take is that all MDPs share the same action space $\mathcal{A}$. This is not a difficult requirement to satisfy because specifying how a game should be played is often part of the instruction for the homework. The second assumption is that all MDPs in this set share the same state space $\mathcal{S}$. This can be satisfied in two ways: 1) If the algorithm takes game objects positional and dynamics as input, we can design the state space to include as much information as possible (i.e., grab all objects on the canvas); 2) If the algorithm takes screenshots/images of game play, then the state space is trivially the same, up to a scaling factor.

To formalize this setting, we consider a set of programs, each fully specifies an environment and its dynamics: $\mathcal{D} = \{(M_n = (\mathcal{S}, \mathcal{A}, T_n, R_n), y_n) : n = 1, 2, 3, ...\}$, where $y_n \in \{0, 1\}$ is a binary label for the MDP M_n, and a set of programs similarly specified to be tested in $\mathcal{D}_{\text{test}}$. We note that n is often a small number, usually less than 20, significantly fewer from other supervised learning tasks. $y = 0$ means the game has no bug, and $y = 1$ means the game has at least one bug. The challenge requires the grading algorithm to correctly assign a label $\hat{y}_n$ for each M_n. Though our work focuses on the binary labeling of MDPs, our proposed algorithms are not inherently limited by this. We leave multi-class labeling of MDPs for future work and note they can provide even larger impacts on generating fine-grained feedbacks to student assignments.

2.1 Equivalence Relations in MDP

The study of equivalence relations in MDPs and labelled transition systems has a long history. Previous works focus on determining equivalence relations of states in order to condense state representations for easier policy learning [16, 17, 18, 19]. The general notion is that if two states are equivalent, they should receive similar reward and lead to a similar next state. Therefore, a distance measure d can be defined to judge whether two states are equivalent.

In our work, instead of considering whether two states are equivalent, we are considering whether two MDPs are the same. If the reference MDP has reward function R and transition function T, we only need to know if $\tilde{T}$ and $\tilde{R}$ from the new MDP are the same as the reference MDP. Unfortunately, since we cannot directly compare the parametrized functional forms, we extend the distance metric developed by [17] from state equivalence relations to MDP equivalence relations by sum over all state in Eq 1. We can use any norm to measure distance of next state for continuous state space, and use an indicator function if the state is discrete. We can use α and β to decide how much we weigh the difference of each.

$$D(M, \tilde{M}) = \sum_{s \in \tilde{\mathcal{S}}} \max_a \alpha |R(s, a) - \tilde{R}(s, a)| + \beta \|T(s, a) - \tilde{T}(s, a)\| \tag{1}$$

However, we will not be able to compute $D(M, \tilde{M})$ exactly for non-tabular MDPs because we cannot efficiently enumerate over all possible states. Our goal is to find some kind of difference between the two MDPs so that we can confidently reject the notion that the given MDP $\tilde{M}$ is the same as our reference MDP M. Therefore, instead of calculating the difference between the two MDPs over all states, we only need a subset of states where the difference is the most significant. As long as we can discover a subset of states, or even one state where a significant difference occurred, we can decide that two MDPs are not the same. We call these states the **differential states**. The difficulty of The challenge is determined by the cardinality of the differential states.

2.2 Play to Grade Objective

The objective of an ideal solution is to discover and reach differential states so that we can confidently determine $p(D(M, \tilde{M}) > \nu) > 0$, where $\nu \geq 0$. ν is chosen by the teacher, and can be thought of as some fault tolerance level. Because $d(M, \tilde{M})$ is a summation over norms, identifying some or any differential states would trivially satisfy this objective. We can rewrite this existential search objective

as an optimization instead. We can define a new MDP where the reward function is the difference on (s, a), and the rest are the same as $\tilde{M}$. We can obtain a policy that learns to maximize this reward, which is equivalent to searching for differential states by sampling **differential trajectories** between two MDPs:

$$d(s, a) = \alpha|R(s, a) - \tilde{R}(s, a)| + \beta\|T(s, a) - \tilde{T}(s, a)\| \tag{2}$$

$$\pi = \arg\max_{\pi} \mathbb{E}_{(s,a)\sim\pi}\Big[\mathbb{1}(d(s, a) > \delta)\Big] \tag{3}$$

Now that we have a clear optimization objective that we can solve it using any reinforcement learning algorithm, we must consider another difficulty. Even though the MDP framework provides a simple abstraction to the problem we face, we do not have real access to the underlying reward and transition function. Given a state s from $\tilde{M}$ and action a from $\pi(s)$, we are unable to directly query the reference MDP M for $T(s, a)$ and $R(s, a)$. This is because we are in an episodic setting, where direct access to transition function and reward function is not possible. Also, we generally assume that each episode will have a random initial state $s_0 \sim p(s_0)$ (i.e., in breakout, the ball gets launched at a different downward direction each time you play). This randomness is defined internally and does not allow for outside control (such as seeding). This means we can't simply use one policy π to sample two trajectories from both MDPs and directly compare the reward and the next state.

3 Recognizing Bugs

The main challenge of solving is task is: if we know what differential states look like (i.e., have a good $d(s, a)$), we can directly train an agent to reach these states; conversely, if we have a good agent that can reliably reach these differential states all the time, we can find a good $d(s, a)$. The fact that we have neither, is the *cold-start* problem of this challenge.

We will address the *cold-start* problem in the next section, but first, if we assume that we do have a good policy that can generate differential trajectories, how should we parametrize and learn a distance function $\hat{d}(s, a)$ that recognizes (s, a) does not belong to the reference MDP? We introduce a few baselines and propose two methods that have a good inductive bias to recognize differential states.

3.1 Baseline: Noisy Supervised Learning

We can learn a good distance function in a parametrized function form trained with noisy supervised learning. Define a function $\phi : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \{0, 1\}$. $\phi(\tilde{s}, \tilde{a}, \tilde{s}') = 0$ if $(\tilde{s}, \tilde{a}, \tilde{s}')$ are sampled by evaluating π on M, otherwise they are sampled from $\tilde{M}$. Since we have a small set of labelled MDPs in our training dataset $\mathcal{D}$, a naive approach to learn ϕ is to use a policy π to collect τ and $\tilde{\tau}$ from M and $\tilde{M}$. Given $(M, y = 0)$ and $(\tilde{M}, y = 1)$, we can label each triple using the label on the MDP itself, treating them as either all bug-inducing states or not bug-inducing states at all: $\mathcal{D}_\tau = \{((s_0, a_0, s_1), y = 0), ((s_1, a_1, s_2), y = 0), ..., ((\tilde{s}_0, \tilde{a}_0, \tilde{s}_1), y = 1), ((\tilde{s}_1, \tilde{a}_1, \tilde{s}_2), y = 1), ...\}$.

The labeling on triples from M is trivially correct. However, the same cannot be said for triples collected from $\tilde{M}$. It is easy to imagine that not all states in a trajectory from $\tilde{M}$ are bug-triggering states, which result in a high amount of erroneously labeled triples in this dataset.

3.2 Baseline: Unsupervised Learning

Another simple idea to learn the distribution of (s, a) sampled from M. We can train a generative model on triples collected from $(M, y = 0)$. The model can optimize for $p_\theta((s, a, s'))$, and let $\phi(\tilde{s}, \tilde{a}, \tilde{s}') \triangleq p_\theta(\tilde{s}, \tilde{a}, \tilde{s}') \geq \sigma$. We can adjust σ based on the performance over $\mathcal{D}$. We compare a Gaussian mixture model and a variational autoencoder [20] as our generative model of choice. For both baseline algorithms, we can assign a label to the new MDP $\tilde{M}$ by computing $\hat{D}(M, \tilde{M}) = \mathbb{E}_{(s,a)\sim\pi,\tilde{M}}[\phi(s, a, \tilde{T}(s, a))]$ and set a decision threshold $\hat{D}(M, \tilde{M}) \geq \nu$.

3.3 HoareLSTM

Since the policy samples sequential observations from an MDP, it is natural to calculate distance as a form of unexpectedness between predicted state and observed state. In fact, the programming language community has been looking into a triple of pre/post-condition along with an action input for a long time. These are called *Hoare triples* [21]. They are also been extended to measure state

equivalence in MDP [22]. Previous work in computational education has used them to learn program embeddings for generating feedbacks [14].

We can approximate the distance function $d(s, a)$ in Eq 2 by learning a transition and reward model. Unlike model-based reinforcement learning [23], where an accurate model of transition model T and reward function R must be learned for policy optimization, we only need to optimize the models to the extent that we can confidently make decisions about $\tilde{M}$'s label.

As explained before, we only have access to either (T, R) or $(\tilde{T}, \tilde{R})$, but not both. We can learn a predictive model $R_\theta(s, a)$ and $T_\theta(s, a)$ for the reference MDP's (T, R). We note that the model does not need to be autoregressive, but we choose to parametrize them as an autoregressive long-short-term memory (LSTM) network [24]. We choose f and g to be fully connected neural networks with separate parameters. Since we are predicting Hoare triples, we call this the HoareLSTM model.

$$h_t = \text{LSTM}(s_t, a_t, h_{t-1}) \tag{4}$$

$$\hat{s}_{t+1} = f(h_t); \hat{r}_t = g(h_t) \tag{5}$$

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a)\sim\pi,M}\left[|\hat{r}_t - R(s, a)| + \|\hat{s}_{t+1} - T(s, a)\|\right] \tag{6}$$

If we learn to T_θ and R_θ to approximate T and R from the reference MDP M, we can compute a state-action tuple distance function from $\tilde{M}$ to M, given any new MDP $\tilde{M}$. We denote this approximate distance function as $\hat{d}_M$ that takes $\tilde{T}$ and $\tilde{R}$ as input:

$$\hat{d}_M(s, a; \tilde{T}, \tilde{R}) = \alpha|\hat{r}_t - \tilde{R}(s, a)| + \beta\|\hat{s}_{t+1} - \tilde{T}(s, a)\| \tag{7}$$

$$\hat{D}(M, \tilde{M}) = \mathbb{E}_{(s,a)\sim\pi,\tilde{M}}\left[\hat{d}_M(s, a)\right] \tag{8}$$

We use $\hat{D}(M, \tilde{M})$ to compute the difference between the predicted trajectory by approximating T and R from MDP M and observed trajectory by actually sampling from $\tilde{T}$ and $\tilde{R}$. At last, we can determine if $\tilde{M}$ is the same as M by checking if $\hat{D}(M, \tilde{M}) \geq \nu$.

3.4 Contrastive HoareLSTM

The distance we calculate is highly reliant on how good our model can approximate the system dynamics of state evolution. However, our model has inherent approximation error. We present the idea of a contrastive distance to potentially mitigate this error.

Instead of learning to approximate one reference MDP's (T, R), we can learn to approximate a correct MDP M^+ with (T^+, R^+), and a bug MDP M^- with (T^-, R^-). Given a new test MDP's $(\tilde{T}, \tilde{R})$, we can perform a contrastive comparison to determine the distance function $\hat{d}_{M^+,M^-}$:

$$\hat{d}_{M^+,M^-}(s, a; \tilde{T}, \tilde{R}) = \alpha|\tilde{R}(s, a) - \hat{R}^+_\theta(s, a)| - \alpha|\tilde{R}(s, a) - \hat{R}^-_\theta(s, a)| + \tag{9}$$

$$\beta\|\tilde{T}(s, a) - \hat{T}^+_\psi(s, a)\| - \beta\|\tilde{T}(s, a) - \hat{T}^-_\psi(s, a)\| \tag{10}$$

We still use policy π to sample (s, a) from the new input MDP $\tilde{M}$, but instead of directly comparing it with one MDP M, we are now comparing it with two MDPs M^+ and M^-, and see which transition and reward model better predicts the behavior of the new input MDP.

The main advantage of using a difference comparison is that we are likely to be more fault tolerant. If we look at how we decide if a MDP has a bug: $\hat{D}(M, \tilde{M}) \geq \nu$, the choice of ν needs to reflect our approximation error when we try to learn T and R so that we are not classifying an correct MDP as a bug MDP if our model is not able to recreate the transition dynamics accurately. If we approximate both (T, R) and $(\tilde{T}, \tilde{R})$ and use the same family of models with similar parameter size, the approximation errors would cancel out to some extent in the contrastive setting. The disadvantage is that now we need to train two models, and the distance can be influenced by how well each model is optimized.

Algorithm 1: Collaborative Reinforcement Learning

TRAINDIFFERENTIALPOLICY $(M, \tilde{M}, \pi_0, \hat{d}_\theta)$
$\delta = 0.1$
for $n \leftarrow N$ **do**
  $\mathcal{D}_\tau$ = `CollectTrajectory`$(M, \tilde{M}, \pi_0)$
  $\hat{d}_\theta$ = `TrainDistFunc`$(\mathcal{L}(\theta), \mathcal{D}_\tau)$
  δ = `AdjustDelta`$(M, \tilde{M}, , \pi_0, \delta, \hat{d}_\theta)$
  Let $r^\star = \mathbb{1}(\hat{d}_\theta(s, a) > \delta)$
  π_n = `TrainDQN`$(\pi_0, \tilde{M}, r^\star)$
end
return $\pi_N, \hat{d}_\theta, \delta$

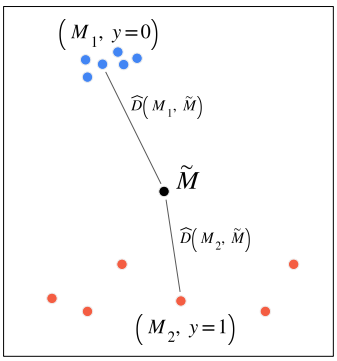


Figure 1: Grading interactive coding assignments as pairwise distance comparisons between MDPs, similar to few-shot learning.

4 Collaborative Training

We have talked about how to compute the distance between two MDPs by calculating distances over differential states. We also showed that by using the distance function as a reward function, we can train an agent that can sample MDPs effective to produce differential trajectories. But we haven't discussed how to solve the *cold-start* problem. We need to simultaneously learn to play the game and find bugs. Even though we have specified an approximate distance function and propose to use it as the reward to train our policy, at the beginning, neither policy nor our approximate distance function were very good. We discuss some simple ideas and delegate more detailed investigations to future work.

4.1 Initialize Agent: Pretraining or Random

An obvious solution is to pre-train either the policy or the parametrized distance function. If the reference MDP has a clearly defined reward function that gives reward when a bug is found, then we can train a policy π that simply learns to maximize the episodic total reward. This is a very useful pre-training to hot-start the policy, especially useful in games with long horizon and easy to fail. After obtaining a pre-trained policy, we can use it to sample trajectories and train the parametrized distance function to capture the reference MDP's dynamics.

The problem with this approach is that the policy is biased toward finishing the original objective in the coding game, it might not lead to any bug discovery. Another problem is that some coding games do not have clearly defined reward. But as we will show in the experiment section, pre-training agent to maximize reward in the original environment is often good enough at sampling sufficient differential trajectories to train a good distance function.

4.2 An Iterative Algorithm

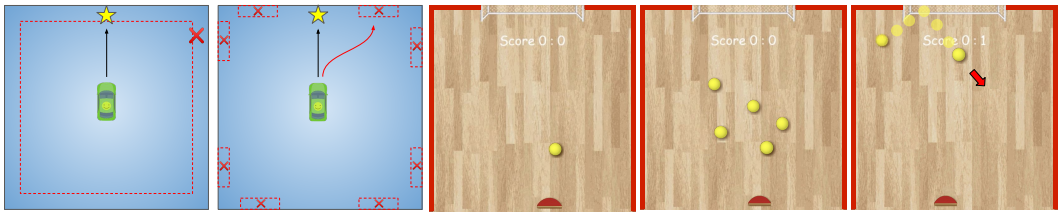
After having the initial agent, we can start to collect trajectories that we can use to train the underlying parameters of our distance function. All of our proposed models are generative models that try to estimate the joint distribution of the observed data. Therefore, whether the trajectories collected are differential for two MDPs or not, it does not affect the inherent optimization of the parameters. However, what it will affect is how difficult is it for us to find δ, which we use to determine if a (s, a) is a bug-inducing state or not, further influencing the reward for our agent in future iterations.

Empirically, we find that for our environments, we do not need to more than 1 iteration of collaborative training. Therefore, we are able to adjust δ at the end of each iteration manually by looking at the distance-to-self $\hat{D}(M, M)$ – if we take (s, a) from the reference MDP and use $\hat{d}$ to calculate the difference to itself, we can choose a δ so that all (s, a) can be labeled as correct. We leave the exploration of how to automatically adjust δ to future work and more complicated environments.

5 Experiment

5.1 Car Environment

We use a toy example to show the properties of collaborative reinforcement learning algorithms. In this toy example, we continue our intuition that the goal is to reach and comprehend the bug-triggering



Car-Easy | Car-Hard | Bounce | Bounce-Bug: **Multiple Balls When Run** | Bounce-Bug: **Ball Bounces On Goal Gate**

Figure 2: We show the illustration of the car environment and two bug difficulty settings on the left, and bounce environment and its two incorrect implementations on the right. For the car environment, bug states are indicated by the dotted red lines and the error signs inside. Note that the correct car environment only has the car and the gold star, without any dotted line or error sign.

state, and reward agent for reaching that state. This is the most generalized representation of the underlying problem. The training of the collaborative reinforcement learning algorithm is illustrated by how quickly the bug classifier can correctly identify the bug state and how quickly the agent learns the shortest route to reach these states. We define difficulty not by how easy it is for the agent to navigate the environment, but by how fast the agent and classifier can collaborate and find the optimal solution. Therefore, the difficulty is arranged by how easy/frequent it is for a random agent to reach the bug state. We explain how these three toy environments will help us understand the challenge of collaborative reinforcement learning below.

In all car environments, the agent can finish the game by reaching the gold star. Let the center of the environment to be $(0, 0)$, the opposite boundaries are -10 and 10. The car's initial x-y coordinates are uniformly sampled from $[-2, 2]$. There are four discrete actions for the agent to take, each action applies an acceleration or deceleration of 0.2 to the velocity along the corresponding direction, with max speed of any direction capped at 1. This allows the car to have more complex trajectories than zig-zag lines.

When a bug-state is reached, the car's physical movement is altered. The velocity is first clipped to be in $[-0.5, 0.5]$. At each step, instead of responding to an action from the agent, the car simply flips the sign of the previously clipped velocity, resulting in back-and-forth movement around the same location – reflecting the idea that the car is "stuck".

5.1.1 Car-Easy

Car-Easy is the setting where almost all agents can easily reach the bug state regardless of whether they adopt the collaborative reinforcement learning objective or not (unless this agent is truly adversarial that just spins around near the initial position). In this setting, the main goal is to test how well bug classifiers perform under with an optimal or near-optimal agent; and whether they can capture the nuance of trajectories between correct and bugged environments.

We test how bug classifiers work under two near-optimal agents in Car-Easy. One agent is a single-direction agent: at every step, it only gives the action that corresponds to the direction of the gold star. Another agent is a random direction agent: at every step, it uniformly samples one out of four available actions. Both agents will reach the gold star, and both agents will enter bug state space. However, the trajectory generated by a random policy is much harder to learn to predict and memorize compared to a single direction policy.

We can see that in Table 1, all models including baseline models are able to predict bug states accurately in a single direction agent setting. However, only our proposed methods: HoareLSTM and Contrastive HoareLSTM are able to learn well by predicting next states in an autoregressive manner. We report the model specification and optimization details in supplementary material.

5.1.2 Car-Hard

Given the knowledge that we have learned from the previous experiment, we use Car-Hard setting to test what happens if the agent only encounters bug states in some of the episodes (trajectories) but not in others. This presents a major challenge for some bug classifiers to correctly the bug state. Unlike Car-Easy, in Car-Hard environment, if the agent drives the car to the goal state, it will not encounter

	Single Direction Agent			Random Agent		
	Accuracy	Precision	Recall	Accuracy	Precision	Recall
Baselines						
Noisy Supervised Learning	91.0 ± 4.7	90.3 ± 5.1	98.6 ± 1.8	60.3 ± 11.8	48.0 ± 14.8	84.6 ± 17.8
Gaussian Mixture Model	25.3 ± 12.5	60.6 ± 26.0	23.8 ± 15.7	17.0 ± 10.6	18.0 ± 11.0	7.3 ± 6.8
Variational Autoencoder	83.8 ± 4.4	84.9 ± 5.0	95.7 ± 1.1	69.5 ± 6.4	51.6 ± 10.9	86.0 ± 8.5
Our methods						
HoareLSTM	99.3 ± 1.9	99.0 ± 2.7	100.0 ± 0.0	98.5 ± 1.1	99.6 ± 1.0	94.1 ± 6.0
Contrastive HoareLSTM	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	94.5 ± 1.8	94.4 ± 2.8	86.0 ± 5.2

Table 1: Bug-state level result of the Car-Easy environment. Randomly agent result we report average over 5 runs. For non-sequential models, we stack the previous 4 states with the current state. We measure the accuracy of correctly classifying bug or non-bug states. Recall and precision are measured only for bug states. We report 95% confidence interval.

	Random Agent			CT - Iteration 1			CT - Iteration 2		
	Acc	Prec	Rec	Acc	Prec	Rec	Acc	Prec	Rec
Baselines									
Noisy Supervised Learning	55.5	24.0	90.3	86.3	86.3	95.6	66.3	48.0	57.9
Gaussian Mixture Model	11.7	12.3	29.4	14.3	23.6	4.6	6.4	0	0
Variational Autoencoder	52.8	21.7	90.7	81.6	77.3	97.0	76.0	71.0	97.5
Our methods									
HoareLSTM	98.9	94.4	93.0	100	100	100	99.8	99.7	100
Contrastive HoareLSTM	98.5	100	90.1	98.9	100	97.5	99.0	100	98.2

Table 2: Bug-state level result of the Car-Hard environment. We show how collaborative iterative training can help classifiers get higher performance. We also average over three runs. We measure the accuracy of correctly classifying bug or non-bug states. Recall and precision are measured only for bug states.

any bugs. However, by adding some epsilon greedy exploration to the agent's action, the agent can still reach bug states.

We use this to show that letting the classifier and the agent train iteratively can improve the performance of both classifier and agent. In Table 2, we show that by applying collaborative training (CT), we allow the agent to become much better at reaching bug states, and in return, allow us to train a much stronger bug classifier even after one iteration. We argue that CT is necessary when bug states are far away from the goal and that is the reason why recall is increased on most models after CT.

5.2 Bounce

Data source *Code.org* is an online computer science education platform that teaches beginner programming. They designed a drag-and-drop interface to teach K-12 students basic programming concepts. Our dataset is compiled from 453,211 students. Each time a student runs their code, the submission is saved. In total, there are 711,274 submissions, where 73,611 unique programs were submitted. Since the data are student programs, there is no personally identifiable information in our dataset. Students agree to share their program submissions as part of the terms of service[1].

Assignment detail Students use a drag-and-drop style code editor to create the game Bounce. Conditions such as "when run" or "when left arrow" are fixed in space. Students place commands such as "Score point", under each condition. There is no limit on how many blocks can be placed under one condition, therefore allowing freedom to add five "Launch new ball" under one condition, or by placing "bounce ball" under "when ball in goal" to prevent ball from passing through the goal gate. These are visualized in Figure 2.

[1] `https://code.org/tos`

	Accuracy	Correct Program Precision	Correct Program Recall	Broken Program Precision	Broken Program Recall
Body Student Programs (607K submissions, 3032 unique)					
Majority Class	50.0	—	—	—	—
Code-as-text	74.2	84.6	59.2	68.6	89.2
Contrastive HoareLSTM + PTW	89.0	82.2	99.6	99.5	78.4
Tail Student Programs (103K submissions, 70579 unique)					
Majority Class	50.0	—	—	—	—
Code-as-text	68.4	85.9	44.0	62.4	92.8
Contrastive HoareLSTM + PTW	91.6	87.1	97.6	97.3	85.6

Table 3: We report the pre-trained "play-to-win" (PTW) agent's performance with our proposed methods on this task. For both body and tail distribution, we sample 250 correct and 250 broken programs for a balanced evaluation.

Evaluation In an unbounded solution space, the distribution of student submissions incur a heavy tail, as observed by [10]. We show that the distribution of submission in dataset conforms to a Zipf distribution. This suggests that we can partition this dataset into two sections. **Body**: we count any unique program submitted more than 10 submissions as the "body" of the distribution. It accounts for 85.5% of the total submissions (607,891). This set contains 419 unique correct programs and 2,613 unique incorrect programs. **Tail**: This set represents any programs that are submitted less than 10 times. There are 70,579 unique programs in this set and 53,200 of them (72.3%) were only submitted once. For both Body and Tail distribution, we sample 250 correct and 250 incorrect programs uniformly from each set for evaluation.

Analysis We evaluate our algorithm on Bounce programs. Bounce, unlike Car environment, has most bug states close to the goal state; for example, if the wall or paddle does not bounce the ball, a game cannot be complete. Therefore, we focus on examining whether our model can grade using the parametrized distance functions alone. We have 10 incorrect programs and 1 correct program in our training data. From our toy environment, we see that only HoareLSTM and Contrastive HoareLSTM are strong candidates for an ideal solution. We report the result of Contrastive HoareLSTM trained with trajectories sampled by a pre-trained play-to-win agent Bounce. The final grading label is determined by a majority vote with 10 contrastive distance functions. This shows the potential of our algorithm in grading real-world assignment using trajectories collected by agents and can be expanded in future work.

6 Discussion

What if game internal states can't be accessed? There are many situations where the internal object representations (such as velocity, position) can't be accessed easily from outside; in this case, often all the information we have would be the image snapshots of the game state. It is true that the iterative learning paradigm we proposed would be more compute intensive, but the model components we explored in this paper are capable of learning and operating on pixels [25].

Does this work for non-game programming assignments? Many student assignments might not be a "game" and the reward might not be clearly defined. However, since our algorithm does not rely on the internal reward of the game – we only rely on the classifier to discover the most differential trajectories between correct and incorrect programs, our solution should generalize to other types of programming assignments that requires interactive user inputs.

Grading as formative feedback A potential negative use of our technology is to grade the AP computer science exam or any kind of exam. We did not develop an assessment tool. Giving students formative feedback can spark interests in learning and improve student's understanding of content in the long run. We argue the societal benefits of our algorithm outweighs potential misuse.

References

[1] William G Bowen. The 'cost disease'in higher education: is technology the answer? *The Tanner Lectures Stanford University*, 2012.

[2] Code.org. About us.

[3] Add hints to code.org tutorials.

[4] Chris Piech, Jonathan Bassen, Jonathan Huang, Surya Ganguli, Mehran Sahami, Leonidas J Guibas, and Jascha Sohl-Dickstein. Deep knowledge tracing. In *Advances in neural information processing systems*, pages 505–513, 2015.

[5] Sumit Basu, Chuck Jacobs, and Lucy Vanderwende. Powergrading: a clustering approach to amplify human effort for short answer grading. *Transactions of the Association for Computational Linguistics*, 1:391–402, 2013.

[6] Lisa Yan, Nick McKeown, and Chris Piech. The pyramidsnapshot challenge: Understanding student process from visual output of programs. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, pages 119–125, New York, NY, USA, 2019. ACM.

[7] Lisa Wang, Angela Sy, Larry Liu, and Chris Piech. Learning to represent student knowledge on programming exercises using deep learning. In *EDM*, 2017.

[8] Jiawei Liu, Yang Xu, and Lingzhe Zhao. Automated essay scoring based on two-stage learning. *arXiv preprint arXiv:1901.07744*, 2019.

[9] Qian Hu and Huzefa Rangwala. Reliable deep grade prediction with uncertainty estimation. *arXiv preprint arXiv:1902.10213*, 2019.

[10] Mike Wu, Milan Mosse, Noah Goodman, and Chris Piech. Zero shot learning for code education: Rubric sampling with deep learning inference. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 782–790, 2019.

[11] Ali Malik, Mike Wu, Vrinda Vasavada, Jinpeng Song, Madison Coots, John Mitchell, Noah Goodman, and Chris Piech. Generative grading: Near human-level accuracy for automated feedback on richly structured problems. *Educational Data Mining*, 2021.

[12] Eleanor O'Rourke, Christy Ballweber, and Zoran Popovií. Hint systems may negatively impact performance in educational games. In *Proceedings of the first ACM conference on Learning@ scale conference*, pages 51–60, 2014.

[13] Thomas W Price and Tiffany Barnes. Position paper: Block-based programming should offer intelligent support for learners. In *2017 IEEE Blocks and Beyond Workshop (B&B)*, pages 65–68. IEEE, 2017.

[14] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. Learning program embeddings to propagate feedback on student code. In *International conference on machine Learning*, pages 1093–1102. PMLR, 2015.

[15] Sahil Bhatia and Rishabh Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks. *arXiv preprint arXiv:1603.06129*, 2016.

[16] Balaraman Ravindran and Andrew G Barto. Symmetries and model minimization in markov decision processes, 2001.

[17] Norm Ferns, Prakash Panangaden, and Doina Precup. Metrics for finite markov decision processes. In *UAI*, volume 4, pages 162–169, 2004.

[18] Lihong Li, Thomas J Walsh, and Michael L Littman. Towards a unified theory of state abstraction for mdps. *ISAIM*, 4:5, 2006.

[19] Elise van der Pol, Thomas Kipf, Frans A Oliehoek, and Max Welling. Plannable approximations to mdp homomorphisms: Equivariance under actions. *arXiv preprint arXiv:2002.11963*, 2020.

[20] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[21] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[22] Pablo Samuel Castro, Prakash Panangaden, and Doina Precup. Equivalence relations in fully and partially observable markov decision processes. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[23] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[24] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[25] Aaron Van Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. In *International Conference on Machine Learning*, pages 1747–1756. PMLR, 2016.

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] in Discussion.
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] We discussed this in Section 7. In short, we intend our algorithm to provide formative feedback, not used for standardized testing assessment. The societal benefit of developing this technology far outweighs the harm at this point.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes] .
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] In supplementary material.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] In supplementary material.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] We reported error bars in main paper and/or in supplementary materials.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] Experiments are run on one Nvidia RTX 3070 GPU. The training takes less than 30 minutes and evaluation takes less than 5 hours for our sampled set of programs.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [N/A]
   (b) Did you mention the license of the assets? [No] The data is currently proprietary, though we do have a plan to open-source it.
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [Yes] See Section 6.3 Data source.
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [Yes] See Section 6.3 Data source.
5. If you used crowdsourcing or conducted research with human subjects...

(a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]