Giving Feedback on Interactive Student Programs with Meta-Exploration

Evan Zheran Liu* Moritz Stephan*
Stanford University Stanford University

Allen Nie Chris Piech Emma Brunskill Chelsea Finn
Stanford University Stanford University Stanford University Stanford University

Abstract

Creating interactive software, such as websites or games, is a particularly engaging way to learn computer science. However, teaching and giving feedback on such software is hard — standard approaches require instructors to hand grade student-implemented interactive programs. As a result, online platforms that serve millions, like Code.org, are unable to provide any feedback on assignments for implementing interactive programs, which critically hinders students’ ability to learn. Recent work proposes to train reinforcement learning agents to interact with a student’s program, aiming to explore states indicative of errors. However, this approach only provides binary feedback of whether a program is correct or not, while students require finer-grained feedback on the specific errors in their programs to understand their mistakes. In this work, we show that exploring to discover errors can be cast as a meta-exploration problem. This enables us to construct a principled objective for discovering errors and an algorithm for optimizing this objective, which provides fine-grained feedback. We evaluate our approach on a set of 700K real anonymized student programs from a Code.org interactive assignment. Our approach provides feedback with 94.3% accuracy, improving over existing approaches by over 17.7% and coming within 1.5% of human-level accuracy.

1 Introduction

Feedback plays a critical role in high-quality education, but can require significant time and expertise to provide [7]. We focus on one area where providing feedback is particularly burdensome: modern computer science education, where students are often tasked with developing interactive programs, such as websites or games (e.g., see Figure 1). While developing such programs is highly engaging [31] and has become ubiquitous in contemporary classrooms [12], these programs can include stochastic or creative elements, so they cannot be graded with traditional unit tests, and must

*Co-first authors. Correspondence to: Evan Z. Liu <evanliu@cs.stanford.edu>.

Figure 1: A learned Play-to-Grade agent for the Bounce programming assignment. The agent tests what happens when the ball is hit into the goal, and finds that the ball incorrectly bounces out instead of scoring a point. Whereas prior work provides coarse feedback of whether the program is correct or not, our goal is to provide fine-grained feedback of the specific mistakes a student has made.
instead be manually graded. However, such manual grading is increasingly infeasible with the growing demand for computer science education and rise of massive online learning platforms. For example, one popular platform, Code.org has enrolled over 72M students [7]. As manually grading a submission can take up to 6 minutes, a single assignment creates decades of grading labor. Consequently, Code.org cannot yet provide feedback about whether an interactive assignment submission is correct or not, let alone more fine-grained feedback.

To alleviate this enormous grading burden, Nie et al. [27] introduce the Play-to-Grade paradigm for automatically providing feedback by training a reinforcement learning agent to grade a program the same way humans do: by interacting or playing with the program. The idea is for the agent to visit states that reveal errors in the program, and then aggregate this information as feedback. Such an agent is trained on a set of training programs labeled with feedback (e.g., provided by an instructor), and the goal is to generalize to new student programs. Figure 1 shows an example learned agent that tests what happens when the ball is hit into the goal, exposing an error where the ball bounces off the goal instead of entering and scoring a point. The state-of-the-art approach in this paradigm provides highly accurate coarse feedback of whether the program is completely correct or not [27]. However, to understand their mistakes, students usually require feedback about what specific errors are in their programs.

Learning an agent to explore and discover the errors in a program to provide such fine-grained feedback is challenging: Most errors cannot be discovered with simple random exploration, and instead require targeted exploration, such as deliberately hitting the ball into the goal. In addition, the agent must be able to adapt its exploration to different programs, which each behave differently and may present unexpected obstacles, such as multiple balls. Our key technical insight is that learning to discover errors connects with the meta-exploration problem in meta-reinforcement learning (meta-RL). This insight enables us to leverage techniques from the meta-exploration literature to construct and optimize a principled objective for producing fine-grained feedback. Specifically, we follow the Play-to-Grade paradigm and assume access to 3500 training programs, labeled with the errors in the program. Then, we formulate the problem as maximizing the mutual information between states visited by our agent and the label. Finally, we use techniques from the DREAM meta-RL algorithm [22] to decompose this objective into shaped rewards that enable learning sophisticated exploration.

Overall, the main contribution of this work is to connect the Play-to-Grade paradigm with the meta-RL literature, and consequently, to provide an effective method for providing fine-grained feedback for interactive student programs, which we call DREAMGRADER. Additionally, we release our code to propose automatic feedback as a new meta-RL benchmark that fulfills an unmet need in the community for a meta-RL benchmark that is simultaneously readily accessible and directly impactful. We evaluate our system on 711,274 anonymized student submissions of the Bounce assignment from Code.org [27]. Trained on 3500 programs, DREAMGRADER achieves an accuracy of 94.3%, which improves over existing approaches by 17.7% and comes within 1.5% of human-level grading accuracy. In addition, our approach can significantly reduce instructor grading burden: while manually grading all student submissions would require 4 years of work, our system can grade the same set of programs in only 3 hours on a single GPU.

2 Related Works

Educational feedback. We consider the problem of automatically providing feedback, which plays an important role in student learning and motivation [29]. Though we specifically focus on feedback, other works also leverage machine learning for other aspects of education, including tracking what students know [42, 8, 32], predicting student retention [9, 4, 37, 3], and building intelligent tutoring systems [1, 28]. Work on automatically providing feedback for computer science assignments focuses on two main approaches: analyzing either the code or behavior of a program. Methods that analyze code provide feedback by passing the code through a neural network [53, 6, 23, 47], or by constructing syntax trees [39, 45], e.g., to predict useful next implementation steps [55, 50]. Analyzing code works well for shorter programs (e.g., under 50 lines of code) and has even been deployed in online courses [48]. However, this approach struggles to scale to lengthier or more complex programs. Hence, we instead opt for the second approach of analyzing program behavior, which conveniently does not depend on program length, though it requires the program to be executable.

Arguably, the simplest method of analyzing program behavior is unit testing. Unit testing can provide automatic feedback to some extent when the desired output of each input is known, but this is typically not the case with interactive programs, such as websites or games. Instead, work on automated testing can provide feedback by generating corner-case inputs that reveal errors via input
We consider the problem of automatically providing feedback on programs. During training, we where

\[ I_k \] which errors are made in the program. The feedback function \( \mu \) defines a Markov decision process (MDP) \( \mu = (S, A, T, R) \) with states \( S \), actions \( A \), dynamics \( T \), and rewards \( R \). We assume that the instructor creates a rubric: an ordered list of \( K \) potential errors that can occur in a program. Each program \( \mu \) is associated with a ground-truth label \( y \in \{0, 1\}^K \) of which errors are made in the program. The \( k \)th index \( y_k \) denotes that the \( k \)th error of the rubric is present in the program \( \mu \).

During training, the grading system is given a set of \( N \) labeled training programs \( \{(\mu^n, y^n)\}_{n=1}^N \). The goal is to learn a feedback function \( f \) that takes a program \( \mu \) and predicts the label \( \hat{y} = f(\mu) \) to maximize the expected grading accuracy \( J_{\text{grade}}(f) \) over test programs \( \mu \) with unobserved labels \( y \):

\[
J_{\text{grade}}(f) = \mathbb{E}_{\mu \sim p(\mu)} \left[ \frac{1}{K} \sum_{k=1}^{K} I[f(\mu)_k = y_k] \right],
\]

where \( I \) is an indicator variable. Effectively, \( J_{\text{grade}} \) measures the per-rubric item accuracy of predicting the ground-truth label \( y \). To predict the label \( y \), the feedback function may optionally interact with the MDP \( \mu \) defined by the program for any small number of episodes.

**Bounce programming assignment.** Though the methods we propose in this work generally apply to any interactive programs with instructor-created rubrics, we specifically consider the Bounce programming assignment from Code.org, a real online assignment that has been completed nearly a million times. As providing feedback for interactive assignments is challenging, the assignment currently provides no feedback whatsoever on Code.org, and instead relies on the student to discover their own mistakes by playing their program. This assignment is illustrated in Figure 1. Each student program defines an MDP, where we use the state representation from Nie et al. [27] — each state consists of the \((x, y)\)-coordinates of the paddle and balls, as well as the \((x, y)\)-velocities of the balls. There are three actions: moving the paddle left or right, or keeping the paddle in the current position. In the dynamics of a correct program, the ball bounces off the paddle and wall. When the ball hits the goal or floor, it disappears and launches a new ball, which increments the player score and opponent score respectively. However, the student code may define other erroneous dynamics, such as the ball passing through the paddle or bouncing off the goal. The reward function is \( +1 \) when the player
score increments and $-1$ when the opponent score increments. An episode terminates after 100 steps or if either the player or opponent score exceeds 30.

Our experiments use a dataset of 711,274 real anonymized student submissions to this assignment, released by Nie et al. [27]. We use 0.5% of these programs for training, corresponding to $N = 3556$ and uniformly sample from the remaining programs for testing.

Possible errors in a student program take the form of “when event occurs, there is an incorrect consequence,” where the list of all events and consequences is listed in Table 1. For example, Figure 1 illustrates the error where the event is the ball hitting the goal, and the consequence is that the ball incorrectly bounces off the goal, rather than entering the goal. For simplicity, we consider a representative rubric of $K = 8$ errors, spanning all event and consequence types, listed in Appendix A.

Prior approach for program feedback. This problem is challenging because the agent must explore in a targeted way to discover all potential errors and must be able to adapt to variability in the programs. Prior work by Nie et al. [27] sidesteps this challenge by determining whether a student program is distinct from a reference solution program, which only results in coarse feedback of whether a program is correct or not. In the next section, we present a new approach that instead targets exploration toward uncovering specific misconceptions to effectively provide fine-grained feedback.

4 Automatically Providing Fine-Grained Feedback with DREAMGRADER

In this section, we detail our approach, DREAMGRADER, for automatically providing fine-grained feedback to help students understand their mistakes. From a high level, DREAMGRADER learns two components that together form the feedback function $f$:

(i) An exploration policy $\pi$ that acts on a program $\mu$ to produce a trajectory $\tau = (s_0, a_0, r_0, \ldots)$.

(ii) A feedback classifier $g(y \mid \tau)$ that defines a distribution over labels $y$ given a trajectory $\tau$.

The idea is to explore states that either indicate or rule out errors with the exploration policy, and then summarize the discovered errors with the feedback classifier. To provide feedback on a new program $\mu$, we first roll out the exploration policy $\pi$ on the program to obtain a trajectory $\tau$, and then obtain the predicted label $\arg \max_y g(y \mid \tau)$ by applying the feedback classifier. Under this parametrization of the feedback function $f$, we can rewrite the expected grading accuracy objective in Equation 1 as:

$$J_{\text{DREAMGRADER}}(\pi, g) = E_{\mu \sim p(\mu), \tau \sim \pi(\mu)} \left[ \frac{1}{K} \sum_{k=1}^{K} \mathbb{I}[\arg \max_y g(y \mid \tau)_k = y_k] \right], \quad (2)$$

where $\pi(\mu)$ denotes the distribution over trajectories from rolling out the policy $\pi$ on the program $\mu$.

After this rewriting of the objective, our approach is conceptually straightforward: we learn both the exploration policy and classifier to maximize our rewritten objective. We can easily learn the feedback classifier $g$ by maximizing the probability of the correct label given a trajectory generated by the exploration policy (i.e., cross-entropy loss), but learning the exploration policy $\pi$ is more challenging. Note that we could directly optimize our objective in Equation 2 by treating the inside of the expectation as a reward received at the end of the episode and use this to learn the exploration policy $\pi$ with reinforcement learning. However, this reward signal makes credit assignment difficult for learning the exploration policy, since it is given at the end of the episode, rather than at the states where the exploration policy discovers errors. Consequently, we empirically find that learning from this reward signal struggles to adequately explore (Section 5).

Hence, our goal is instead to construct a reward signal that helps assign credit for the exploration policy and provides rewards at the states that indicate or rule out errors in the program. To do this, we propose an alternative objective that is sufficient to maximize the Play-to-Grade objective, but can be decomposed into per-timestep rewards that correctly assign credit (Section 4.1). Intuitively, these rewards leverage the feedback classifier to provide high reward when an action leads to a new state that causes the classifier to become very certain about whether an error is present. After, we
**Error type:** Does hitting the ball into the goal increment the score?

Figure 2: **DREAMGRADER** provides credit assignment for learning exploration by leveraging the feedback classifier \( g(y \mid \tau) \). Here, we consider the error “when the ball enters the goal, the player score does not increment.” At \( 0 \) and \( 4 \), no information is observed that either rules out or indicates the error. Hence, no exploration reward \( r_t^{\text{exp}} \) is provided, and the classifier assigns 0.12 probability that the error is present, reflecting the prior that 12% of the training programs have this error. At \( 4 \), the ball enters the goal, but does not score a point and a new ball is launched. This indicates that the error is present, so the classifier updates, which creates high exploration reward and credit assignment for learning exploration. At \( 3 \), the ball enters the goal again. However, no additional information is gained, so the classifier does not change, and no reward is given. Overall, this enables learning effective exploration that purposely hits the ball into the goal once.

4.1 Assigning Credit to Learn Exploration

We now obtain a reward signal to help assign credit for learning exploration. From a high level, we first propose to maximize a mutual information objective that is sufficient to maximize our objective \( J_{\text{DREAMGRADER}}(\pi, g) \) in Equation 2. We then rewrite our mutual information objective in terms of per-timestep exploration rewards related to the information gain of the feedback classifier on the true label \( y \) when it observes the transition \((s_t, a_t, r_t)\). This helps assign credit for learning the exploration policy, as the transitions that either indicate or rule out errors in the program are exactly those that have high information gain. Figure 2 illustrates an example of the derived exploration rewards.

**Objective.** Intuitively, we want our exploration policy to visit states that either indicate or rule out errors. We can formalize this intuition by maximizing the mutual information \( I(\tau; y) \) between the trajectories \( \tau \sim \pi \) visited by the policy and the feedback label of the program \( y \), which causes the policy to visit states that make the feedback label highly predictable. Importantly, maximizing this objective is sufficient to maximize the expected grading accuracy \( J_{\text{DREAMGRADER}}(\pi, g) \) in Equation 2. Let \( p(y \mid \tau) \) be the true posterior over labels given trajectories \( \tau \sim \pi \) sampled from the policy. Maximizing the mutual information \( I(\tau; y) \) produces trajectories that maximize the probability of the label under the true posterior \( p(y \mid \tau) \). Then, if the feedback classifier is learned to match the true posterior \( g(y \mid \tau) = p(y \mid \tau) \) while \( I(\tau; y) \) is maximized, the expected grading accuracy \( J_{\text{DREAMGRADER}}(\pi, g) \) is also maximized.

**Optimization.** We can efficiently maximize our objective \( I(\tau; y) \) by maximizing a variational lower bound \( \mathcal{L} \) and decomposing the lower bound into a per-timestep reward that helps assign credit for learning the exploration policy \( \pi \). We derive this for the case of learning only a single exploration policy \( \pi \) to uncover all errors, though we will later discuss how we can factorize this to learn \( N \) exploration policies \( \{\pi_i\}_{i=1}^N \) that each explore to uncover a single error type.

\[
I(\tau; y) = H[y] - H[y \mid \tau] \\
= H[y] + \mathbb{E}_{\mu \sim p(\mu), \tau \sim \pi(\mu)} \log p(y \mid \tau) \\
\geq H[y] + \mathbb{E}_{\mu \sim p(\mu), \tau \sim \pi(\mu)} \log g(y \mid \tau) \\
= H[y] + \mathbb{E}_{\mu \sim p(\mu), \tau \sim \pi(\mu)} \left[ \log g(y \mid s_0) + \sum_{t=0}^{T-1} r_t^{\text{exp}} \right],
\]

where \( r_t^{\text{exp}} = \log g(y \mid \tau_{t+1}) - \log g(y \mid \tau_t) \) and 

\( T \) is the length of the trajectory \( \tau = (s_0, a_0, r_0, \ldots, s_T) \).
The inequality in (5) holds for replacing the true posterior \(p(y | \tau)\) with any distribution, and (6) comes from expanding a telescoping series as done by DREAM [22], where \(\tau_t = (s_0, a_0, r_0, \ldots, s_t)\) denotes the the trajectory up to the \(t\)th state.

This derivation provides a shaped reward function \(r_t^{exp}\) for learning the exploration policy. Intuitively, this reward captures how much new information the transition \((s_t, a_t, \tau_t, s_{t+1})\) provides to the feedback classifier \(g\) on what errors are in the program: The reward is high if observing this transition either indicates an error (e.g., the ball enters the goal, but does not score a point) or rules out an error (e.g., the ball enters the goal and scores a point), and is low otherwise.

Additionally, we now have a recipe for maximizing the mutual information \(I(\tau; y)\) to learn both the policy \(\pi\) and feedback classifier \(g\). Only the second term in (6) depends on \(\pi\) and \(g\). Hence, we can maximize this lower bound on \(I(\tau; y)\) by maximizing the log-likelihood of the label with respect to the feedback classifier \(J_{\text{feedback}}(g) = \mathbb{E}_{y \sim p(y), \tau \sim \pi(\mu)} [\log g(y | \tau)]\), and maximizing the rewards \(r_t^{exp} = \log g(y | \tau_{t+1}) - \log g(y | \tau_t)\) with respect to the policy \(\pi\) via reinforcement learning.

**Factorizing our objective.** So far, our approach learns a single exploration policy that must uncover all error types in the rubric. However, learning such a policy can be challenging, especially if uncovering different error types requires visiting very different states. We instead propose to learn a separate exploration policy \(\pi_k\) for each error index of the rubric \(k = 1, \ldots, K\). We can accomplish this by observing that maximizing the mutual information \(I((\tau_1, \ldots, \tau_K); y)\) between \(K\) trajectories \(\tau_i \sim \pi_i\) is also sufficient to maximize the expected grading accuracy. Furthermore, maximizing the mutual information with each dimension of the label \(I(\tau_k; y_k)\), where \(\tau_k \sim \pi_k\) for each \(k\), is sufficient to maximize the mutual information with the entire label \(I((\tau_1, \ldots, \tau_K); y)\). We therefore can derive exploration rewards for each term \(I(\tau_k; y_k)\) to learn each policy \(\pi_k\) with rewards \(r_t^{exp} = \log g(y_k | \tau_{t+1}) - \log g(y_k | \tau_t)\). We find that this improves grading accuracy in our experiments (Section 5) and enables parallel training and testing for the \(K\) exploration policies.

### 4.2 A Practical Implementation

Overall, DREAMGRADER consists of a feedback classifier \(g\) and \(K\) exploration policies \(\{\pi_k\}_{k=1}^K\), where the \(k\)th policy \(\pi_k\) tries to visit states indicative of whether the \(k\)th error type is present in the program. We learn these components by repeatedly running training episodes for each policy \(\pi_k\) with Algorithm 1. We first sample a labeled training program and follow the policy on the program (lines 1–2). Then, we maximize our mutual information objective by updating the policy with our exploration rewards (lines 3–4), and by updating the classifier to maximize the log-likelihood of the label (line 5).

In practice, we parametrize the exploration policies and feedback classifier as neural networks. Since the exploration rewards \(r_t^{exp}\) depend on the past and are non-Markov, we make each exploration policy \(\pi_k\) recurrent: At timestep \(t\), the policy \(\pi_k(a_t | (s_0, a_0, r_0, \ldots, s_t))\) conditions on all past states, actions, and observed rewards for each \(k\). We parametrize each policy as a deep dueling double Q-networks [25][46][41]. Consequently, our policy updates in line 4 consist of placing the trajectory in a replay buffer with rewards \(r_t^{exp}\) and sampling from the replay buffer to perform Q-learning updates. We parametrize each dimension of the feedback classifier \(g(y_k | \tau)\) for \(k = 1, \ldots, K\) as a separate neural network. We choose not to share parameters between the exploration policies and between the dimensions of the feedback classifier for simplicity. See Appendix B for full architecture and model details.

### 4.3 Play-to-Grade as Meta-Exploration

Our choice to optimize the mutual information objective \(I(\tau; y)\) and decompose this objective into per-timestep rewards using techniques from the DREAM meta-RL algorithm stems from the fact that the Play-to-Grade paradigm can be cast as a meta-exploration problem. Specifically, meta-RL aims to learn agents that can quickly learn new tasks by leveraging prior experience on related tasks. The standard few-shot meta-RL setting formalizes this by allowing the agent to train on several MDPs (tasks). Then, at time time, the agent is presented with a new MDP and is allowed to first explore the new MDP for several episodes to gather information, before it must solve the MDP and maximize returns. Learning to efficiently spend these allowed few exploration episodes to best solve the test MDP is the meta-exploration problem.
In our setting, we can view each student program as a 1-step task of predicting the feedback label. To predict this label, the Play-to-Grade paradigm first explores the program for several episodes to discover states indicative of errors, which is exactly the meta-exploration problem. This bridge between identifying errors in programs and meta-exploration suggests that techniques from each body of literature could mutually benefit each other. Indeed, DREAMGRADER leverages ideas from DREAM and future work could explore other techniques to transfer across the two areas. Additionally, this connection also offers a new benchmark for meta-exploration and meta-RL research. As discussed in Section 4.3, while existing meta-RL benchmarks tend to be either readily accessible or impactful and realistic, automatically providing feedback simultaneously provides both, and we release code for a meta-RL wrapper of the Bounce programming assignment to spur further research in this direction.

5 Experiments

In our experiments, we aim to answer five main questions: (1) How does automated feedback grading accuracy compare to human grading accuracy? (2) How does DREAMGRADER compare with Nie et al. [27], the state-of-the art Play-to-Grade approach? (3) What are the effects of our proposed factorization and derived exploration rewards on DREAMGRADER? (4) How much human labor is saved by automating feedback? (5) Interactive programs can be particularly challenging to grade because they can include unexpected creative elements not seen during training — how well do automated feedback systems generalize to such creative elements? To answer these questions we consider the dataset of 700K real anonymized Bounce student programs, described in Section 3.

Below, we first establish the points of comparison to necessary answer these questions (Section 5.1). Then, we evaluate these approaches to answer the first four questions (Section 5.2). Finally, we answer question (5) by evaluating DREAMGRADER on variants of Bounce student programs that modify the ball and paddle speeds, including speeds not seen during training (Section 5.3).

5.1 Points of Comparison

We compare with the following four approaches. Unless otherwise noted, we train 3 seeds of each automated approach for 5M steps on $N = 3556$ training programs, consisting of 0.5% of the dataset.

**Human grading.** To measure the grading accuracy of humans, we asked for volunteers to grade Bounce programming assignments. We obtained 9 volunteers consisting of computer science undergraduate and PhD students, 7 of whom had previously instructed or been a teaching assistant for a computer science course. Each volunteer received training on the Bounce programming assignment and then was asked to grade 6 randomly sampled Bounce programs. See Appendix C.1 for details.

Nie et al. [27] extended to provide fine-grained feedback. We extend the original Play-to-Grade approach, which provides binary feedback about whether a program is completely correct or not, to provide fine-grained feedback. Specifically, Nie et al. [27] choose a small set of 10 training programs, curated so that each program exhibits a single error, and together, they span all errors. Then, for each training program, the approach learns (i) a distance function $d(s, a)$ that takes a state-action tuple $(s, a)$, trained to be large for tuples from the buggy training program and small for tuples from a correct reference implementation; and (ii) an exploration policy that learns to visit state-actions where $d(s, a)$ is large. To provide feedback to a new student program, the approach runs each exploration policy on the program and outputs that the program has an error if any of the distance functions is high for any of the tuples visited by the exploration policies.

We extend this approach to provide fine-grained feedback by following a similar set up. We follow the original procedure to train a separate policy and distance function on $K = 8$ curated training programs, where the $k$th program exhibits only the $k$th error from the rubric we consider. Then, to provide fine-grained feedback on a new program, we run each policy on the new program and predict that the $k$th error is present (i.e., $y_k = 1$) if the $k$th distance function is high on any state-action tuple. We use code released by the authors without significant fine-tuning or modification. We emphasize that this approach only uses 8 curated training programs, as opposed to the $N = 3556$ randomly sampled programs used by other automated approaches, as this approach is not designed to use more training programs, and furthermore cannot feasibly scale to many more training programs, as it learns a distance function and policy for every training program.

DREAMGRADER (direct max). To study the effect of our derived exploration rewards $r^{exp}_{i}$, we consider the approach of directly maximizing the DREAMGRADER objective in Equation 2 described at the beginning of Section 4.1. This approach treats the inside of the expectation as end-of-episode
returns, and does not provide explicit credit assignment. This approach is equivalent to maximizing the DREAMGRADER objective with the RL\(^2\) meta-RL algorithm [10, 43].

**DREAMGRADER (unfactorized).** Finally, to study the effect of our proposed factorization scheme, described at the end of Section 4.1, we consider a variant of DREAMGRADER where we do not factorize the objective and instead only learn a single exploration policy to uncover all errors.

### 5.2 Main Results

We compare the approaches based on grading accuracy, precision, recall, and F1 scores averaged across the 8 error types in the rubric. Table 2 summarizes the results. Overall, we find that DREAMGRADER achieves the highest grading accuracy of the automated grading approaches, providing feedback with 17.7% greater accuracy than Nie et al. [27]. Furthermore, DREAMGRADER comes within 1.5% of human-level grading accuracy and actually achieves slightly superhuman F1. DREAMGRADER achieves this by learning exploration behaviors that probe each possible error event (see Appendix D for visualizations of the learned behaviors).

#### Analysis
To further understand these results, we plot the training curves of the grading accuracy vs. the number of training steps for DREAMGRADER with 1-stddev error bars. We plot the final grading accuracies of the other approaches as horizontal lines.

Figure 3: Average grading accuracy for each error type vs. number of training steps for DREAMGRADER with 1-stddev error bars. We plot the final grading accuracies of the other approaches as horizontal lines.
the “no new ball after goal” error when multiple events launch balls. Human grading was able to circumvent this issue, but humans required playing the program for up to 40 episodes, while we limited DREAMGRADER to a single episode per error type.

**Reduction in human grading burden.** We found that our grading volunteers required between 1–6 minutes to grade each program, averaging around 3 minutes. At this grading speed, grading all of the 700K Code.org submissions would take close to 4 years of human labor. In contrast, automatic grading with DREAMGRADER requires only 1 second per program on a single NVIDIA RTX 2080 GPU, which grades all 700K submissions in only 3 hours.

### 5.3 Feedback in the Face of Creativity

A key challenge in providing feedback for interactive programs is that student programs may include unexpected creative elements, requiring generalization to unseen behaviors at test time. The Bounce assignment provides one simple way for students to express creativity in their programs by selecting the speeds of the ball and paddle. There are five possible settings for the ball and paddle speeds: very slow, slow, normal, fast, and very fast. To test the ability of DREAMGRADER to generalize to unseen behaviors at test time, we train DREAMGRADER on programs where we hold out the normal ball and paddle speeds, and then test DREAMGRADER on programs with the held out speeds. Specifically, we use the same $N = 3556$ training programs as before, but we uniformly randomize the ball and paddle speeds independently to be one of the four non-held out speeds. Then we evaluate on test programs with (1) held out ball and paddle speeds; (2) held out ball speed with random training paddle speed; (3) held out paddle speed with random training ball speed; and (4) training ball and paddle speeds.

DREAMGRADER successfully generalizes to unseen ball and paddle speeds at test time. Compared to the standard training in the previous section, where all ball and paddles had the “normal” speed, performance drops. However, accuracy remains relatively high, and DREAMGRADER performs about the same on test programs regardless of whether the ball and paddle speeds were seen during training or not. This indicates some ability to generalize to unseen behaviors at test time. Table 3 displays the full results.

### 6 Conclusion

In this work, we introduced DREAMGRADER, an automatic grading system for interactive programs that provides fine-grained feedback at near human-level accuracy. The key insight behind our system is connecting the problem of automatically discovering errors with the meta-exploration problem in meta-RL, which yields important benefits for both sides. On the one hand, this connection offers a powerful and previously unexplored toolkit to computer science education, and more generally, to discovering errors in websites or other interactive programs. On the other hand, this connection also opens impactful and readily accessible applications for meta-RL research, which has formerly primarily focused on synthetic tasks due to the lack of more compelling accessible applications.

While DREAMGRADER nears human-level grading accuracy and even achieves superhuman grading accuracy on many error types, we caution against blindly replacing instructor feedback with automated grading systems, such as DREAMGRADER, which can lead to potentially negative educational and societal impacts without conscientious application. For example, practitioners must ensure that automated feedback is equitable and not biased against certain classes of solutions that may be correlated with socioeconomic status. One option to mitigate the risk of potential negative consequences while still reducing instructor labor is to use automated feedback systems to assist instructors, e.g., by querying the instructor on examples where the system has low certainty, or presenting videos of exploration behavior from the system for the instructor to provide the final feedback.

Finally, this work takes an important step to reduce teaching burden and improve education, but we also acknowledge DREAMGRADER still has important limitations to overcome. Beyond the remaining small accuracy gap between DREAMGRADER and human grading, DREAMGRADER also requires a substantial amount of training data. Though we only use 0.5% of the Bounce dataset for training, it still amounts to 3500 labeled training programs, and labeling this many programs can be
prohibitive for smaller-scale classrooms, though feasible for larger online platforms. We hope that future work on meta-exploration via our released benchmark can help overcome these limitations.

Acknowledgments and Disclosure of Funding

We thank Kali Stover from the Institutional Review Board (IRB) and Ruth O’Hara and Tallie Wetzel from the Student Data Oversight Committee (SDOC) for reviewing our process of asking humans to grade programs.

We thank accountability buddies Annie Xie and Sahaamble Suri for thwarting EL’s best attempts at procrastination, without whom, this work would not have been completed in time. Icons in this work were made by FreePik from flaticon.com.

EL is supported by a National Science Foundation Graduate Research Fellowship under Grant No. DGE-1656518. This work was also supported in part by Google.

References


A Dataset Details

We use the Bounce programming assignment dataset from Code.org, released by Nie et al. [27]. We release code that packages this dataset as an easy-to-use meta-RL benchmark.

Rubric. Our rubric contains the following 8 error types, spanning all of the events and consequences listed in Table 1.

1. When the ball hits the goal, it incorrectly bounces off.
2. When the ball hits the goal, the opponent score is incorrectly incremented.
3. When the ball hits the goal, no new ball is launched.
4. When the ball hits the floor, the opponent score is not incorrectly incremented.
5. When the ball hits the wall, the opponent score is incorrectly incremented.
6. When the left or right action is taken, the paddle moves in the wrong direction.
7. When the ball hits the paddle, the player score is incorrectly incremented.
8. When the program starts, no ball is launched.

Some errors in the program make it impossible to uncover other errors. For example, if no ball is launched at the start of the program, it is impossible to check whether error 7 is present in the program, because there is no ball to hit on the paddle. In these situations, we label all of the impossible to check errors as not present in the program.

Future work could investigate iteratively providing feedback on a program to a student in order to provide feedback about all errors, including those that are initially impossible to uncover. Such a system could work as follows: First, the grading system provides feedback about all of the errors it can currently find in the program. Then, the student updates their program to fix all of the errors found by the program. Finally, the student re-submits their updated program to the grading system for another round of evaluation. This process continues until there are no more errors in the program.

Statistics. The dataset consists of 711,274 submissions from 453,211 students — some students created multiple submissions. Amongst the 711,274 submissions, there are 111,773 unique programs. The error labels y assigned to each program were programmatically generated by Nie et al. [27].

B DREAMGrader Details

Our implementation of DREAMGrader builds off the DREAM code released by Liu et al. [22] at https://github.com/ezliu/dream

B.1 Model Architecture

DREAMGrader consists of K recurrent exploration policies \{π_k\}_{k=1}^K and a feedback classifier g(y | τ).

Exploration policies. Each exploration policy π_k is parametrized as a double dueling deep Q-network [25, 46, 41], consisting of a recurrent Q-function Q(τ_t, a_t) and a target network Q_target(τ_t, a_t) with the same architecture as the Q-function, where τ_t = (s_0, a_0, r_0, ..., s_t) denotes the trajectory so far up to timestep t. To parametrize these Q-functions, we first embed the trajectory τ_t as e(τ_t) ∈ R^{64}. Then, we apply two linear layers with output size 1 and |A| respectively. These represent the state-value function V(τ_t) and advantage A(τ_t, a_t) respectively. Finally, following the dueling architecture [46], we compute the Q-value as:

\[
Q(τ_t, a_t) = V(τ_t) + A(τ_t, a_t) - \frac{1}{|A|} \sum_{a \in A} A(τ_t, a). \tag{7}
\]

To compute the trajectory embedding e(τ_t), we embed each tuple (s_{t'}, a_{t'}, r_{t'}, s_{t'+1}) for t' = 0, ..., t - 1 and then pass an LSTM [12] over the embeddings of the tuples. The embedding of (s_{t'}, a_{t'}, r_{t'}, s_{t'+1}) is computed by embedding each component and applying a final linear layer with output dimension 64 to the concatenation of the embedded components. We embed s_{t'} and s_{t'+1} with the same network, using the architecture from Nie et al. [27], consisting of two linear layers with output dimensions 128 and 64, respectively, with an intermediate ReLU activation. We embed
the action $a_t$ with an embedding matrix with output dimension 16. We embed the scalar reward $r_t$ with a single linear layer of output dimension 32.

Each exploration policy $\pi_k$ is trained to maximize the expected discounted exploration rewards via standard DQN updates:

$$J_{\text{exp}}(\pi) = \mathbb{E}_{\mu \sim p(\mu), \tau \sim \pi(\mu)} \left[ \sum_{t=0}^{T} r_t^{\text{exp}} \right],$$

where $r_t^{\text{exp}} = \log g(y_k | \tau_{t+1}) - \log g(y_k | \tau_t)$. (9)

**Feedback classifier.** The feedback classifier $g(y | \tau)$ outputs a distribution over predicted labels $y \in \{0, 1\}^K$. We parametrize each dimension of the feedback classifier $g(y_k | \tau)$ with a separate neural network for simplicity. To parametrize $g(y_k | \tau)$, we embed the trajectory $\tau$ as $e(\tau)$ using a network similar to the one for the exploration policy. Then, we apply three linear layers with output dimensions 128, 128, and 2 respectively to $e(\tau)$ with intermediate ReLU activations. Finally, we apply a softmax layer to the output of the linear layers, which forms the distribution over $y_k \in \{0, 1\}$.

To embed the trajectory $\tau$ as $e(\tau)$, we embed each $(s_t, a_t, r_t, s_{t+1})$-tuple for each timestep $t$ in the trajectory $\tau$. Then, we pass an LSTM with output dimension 128 over the embeddings of the tuples, and take the last hidden state of the LSTM as $e(\tau)$. To embed each $(s_t, a_t, r_t, s_{t+1})$-tuple, we embed each component separately, and then apply two final linear layers with output dimensions 128 and 64 respectively and an intermediate ReLU activation. We use the same networks architectures used in the exploration policy trajectory embeddings to embed the state, action, and reward components.

Each dimension of the feedback classifier $g(y_k | \tau)$ is trained to maximize:

$$J_{\text{feedback}}(g) = \mathbb{E}_{\mu \sim p(\mu), \tau \sim \pi(\mu)} [\log g(y_k | \tau)].$$

**B.2 Hyperparameters**

We use the hyperparameters listed in Table 4 for all of our experiments. We chose values based on those used in Liu et al. [22] used for DREAM, and did not tune these values. We optimize the objectives written above using the Adam optimizer [21]. During training, we anneal the $\epsilon$ for $\epsilon$-greedy exploration from 1 to 0.01 over 250,000 steps. We use $\epsilon = 0$ during evaluations.

**C Experiment Details**

**C.1 Human Grading**

**Grading details.** We obtained 9 volunteers to measure the grading accuracy of humans by soliciting volunteers from a university research group, consisting of computer science undergraduate, master’s and PhD students involved in machine learning research. Each volunteer first received training about the Bounce programming assignment by reading a document describing the behavior of a correct Bounce program and all potential errors that may occur in incorrect implementations. Additionally, each volunteer was allowed to play a correct implementation for as long as they desired. After receiving training, each volunteer was presented with the same set of 6 randomly sampled Bounce programs, but in a randomized order. For each program, each volunteer was allowed to play the program for as long as they needed and was asked to list the errors they found in the program on a checklist including all possible errors. The reported human grading accuracy in Table 2 is the mean accuracy of the 9 volunteers on these 6 programs.

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discount Factor $\gamma$</td>
<td>0.99</td>
</tr>
<tr>
<td>Learning Rate</td>
<td>0.0001</td>
</tr>
<tr>
<td>Replay buffer batch size</td>
<td>32</td>
</tr>
<tr>
<td>Target parameters syncing frequency</td>
<td>5000 updates</td>
</tr>
<tr>
<td>Update frequency</td>
<td>4 steps</td>
</tr>
<tr>
<td>Grad norm clipping</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 4: Hyperparameters used for DREAMGRADER.
The exploration behaviors learned by DREAMGRADER. These exploration behaviors probe all of the possible events associated with errors, enabling DREAMGRADER to uncover all errors. Crucially, these exploration behaviors are robust to different programs. For example, the illustrated “score goal” exploration behavior succeeds in hitting a ball into the goal, even when there are multiple balls.

**Compensation.** Volunteers took 30–45 minutes total to familiarize themselves with the Bounce programming assignment and to grade the 6 programs. They were each compensated with a $10 gift card, amounting to a compensation of roughly $15 per hour.

**Institutional Review Board.** The grading process did not expose grading volunteers to any risks beyond that of normal life. It was reviewed by the Institutional Review Board (IRB) and it was determined that it did not constitute human subjects research and did not require IRB approval. Below is the final determination of the IRB.

After further review, the IRB has determined that your research does not involve human subjects as defined in 45 CFR 46.102(f) and therefore does not require review by the IRB.

**D Additional Results**

**Visualizations of DREAMGRADER exploration behavior.** We visualize the exploration behaviors learned by DREAMGRADER’s exploration policies in Figure 4. Qualitatively analyzing these behaviors shows that DREAMGRADER exploration behaviors that probe each possible event type listed in Table 1. Specifically, DREAMGRADER learns to hit the ball into the goal; hit the ball into the wall; hit the ball with the paddle; deliberately miss the ball; and move the paddle in various directions. Crucially, we find that behaviors are fairly robust to different programs. For example, we find that the exploration policies for bugs related to hitting the ball into the goal still successfully hit the ball into the goal most of the time, even when there are multiple balls, or when the actions are reversed, so that the left action moves the paddle right, and vice-versa. This indicates that DREAMGRADER can handle the key challenge of our problem setting: learning diverse and adaptable exploration behaviors.