PLAY TO GRADE: A REINFORCEMENT LEARNING GENERALIZATION CHALLENGE FOR GRADING INTERACTIVE CODING GAMES

Anonymous authors
Paper under double-blind review

ABSTRACT

Online coding education platforms have been attempting to introduce fun and exciting coding exercises that allow code learners to design and develop an interactive system, such as a game, to cultivate interests and teach basic concepts of programming. Unlike traditional code education projects, these interactive systems require *dynamic* user inputs which makes it difficult to be checked by traditional unit tests. We introduce a new challenge of programming assignment grading: given a student program and a few reference programs provided by a teacher, the task is to determine whether the student program is correct or incorrect. In this paper we formalize this challenge as a novel reinforcement learning generalization task: Play to Grade. Each student's program fully specifies a unique environment where the agent needs to operate and decide if the environment is correct or not. We demonstrate that by experiencing a handful of reference environments millions of times, the agent can form a strong normative expectation that can be used to determine anomaly. Our method drastically reduces the amount of data needed to train an automatic grading system and present a challenge to state-of-the-art reinforcement learning generalization methods. Teamed with Code.org, we curated a dataset of 700k student submissions, one of the largest dataset of anonymized student submissions to a single assignment. The Code.org assignment had no previous solution for providing feedback to students and as such this contribution will lead to meaningful improvement in the educational experience of all future students.

1 INTRODUCTION

The need for high quality education at scale is of critical importance (Bowen, 2012). 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.

Automatically providing feedback based on code text has proven to be a hard machine learning problem (Wu et al., 2019). First of all, even for introductory level computer science education, homework datasets have statistical distributions with heavy tails similar to natural language, making correctly identifying mistakes in tail distribution challenging. Second, hand labeling feedback in programming assignment grading is expensive, rendering supervised solutions that require large amount of training data infeasible. Lastly, each assignment can be considered its own task and many online education platforms design their own coding languages to create an easier learning experience, hindering the ability for text based solution to generalize.

Detecting errors in programs has been a well-established field in software development, from catching syntactic errors when the program is written (Aho et al., 1986), to triggering runtime error by providing the system with random inputs (Duran & Ntafos, 1984; Godefroid et al., 2005), or build a reinforcement learning agent that learns to reach as many state in a game as possible so that the game system will fail (Zheng et al., 2019). These approaches are ill-suited for coding games designed by the online education platforms because they rely on the triggering of system failures, while programming programming assignment grading focuses more on deciding whether the game *behaves* correctly.

We propose the task of programming assignment grading to encourage development of automatic grading solutions, as the ability to automatically provide feedback to students assignment bottlenecks the overall quality of online education. Massive numbers of student sign-up understate the enormous demand for coding education. We created the challenge based on Bounce, a popular coding exercise developed by Code.org, who provides coding education to 40% of U.S. K-12 students. We further curated 711,274 actual student submissions to this assignment and developed a game engine that can faithfully execute student programs.

In this paper we make a first attempt to use deep reinforcement learning to grade interactive student assignments. We name this the *Play to Grade* challenge. We show that even with a handful of reference programs, we can train an agent to form a strong normative expectation of what an environment generated by a correct program should look like. However, treating coding assignment grading as a reinforcement learning task introduces an unique challenge: each student's program specifies a unique Markov Decision Problem (MDP), and the agent needs to be able to operate across all of them. Frequently, students are allowed to be creative in their code (such as changing the background of game, or changing shape/color of game objects). Similar to recent efforts in evaluating reinforcement learning algorithm's ability to generalize (Cobbe et al., 2019; Wang et al., 2020), *Play to Grade* needs to have an agent that generalizes to novel environments.

We design two experiments to evaluate the performance of our algorithm. In the first experiment, we evaluate the agent's performance across environments with visual differences. We show that state-of-the-art RL generalization techniques (Laskin et al., 2020) are insufficient to train an agent that can successfully generalize across our variants. In the second experiment, given an optimal agent, we show that using simple features like sum of total rewards and the hamming distance between expected reward and received reward, we can build a classifier that can differentiate between correct and incorrect programs with 94% accuracy for top-100 programs.

Main contributions in this paper:

1. We introduce the reinfocement learning challenge of *Play to Grade* and will release an open-access dataset for evaluation of future solutions.
2. We propose a baseline solution where an agent learns to play a game and while interacting with a student solution, contrasts its expectations with observations in order to determine if the program is correct.
3. We demonstrate high performance (94%) which will allow for a deployable solution – giving feedback to thousands, or perhaps millions of future students.

2 THE PLAY TO GRADE CHALLENGE

Our task consists of a set of programs, each defines a unique environment. Similar to the training and testing split of supervised learning, we want to separate two types of programs: reference programs and student submitted programs. Reference programs are all implemented by teachers or submitted student programs that are already graded by teachers. Correct reference programs can be regarded as the "correct" implementation of an environment. Beside correct reference programs, we also allow a small handful of reference incorrect programs that are "best guesses" from the teachers that the type of errors that might appear in student submitted programs. Each run of the environment will have a randomly sampled initial position and speed for the ball.

Consider an agent whose behavior model is given by a MDP tuple $(\mathcal{S}^\star, \mathcal{A}^\star, P^\star, R^\star, \gamma)$, we use * to denote that the MDP is specified by the correct reference program. A submitted program specifies a

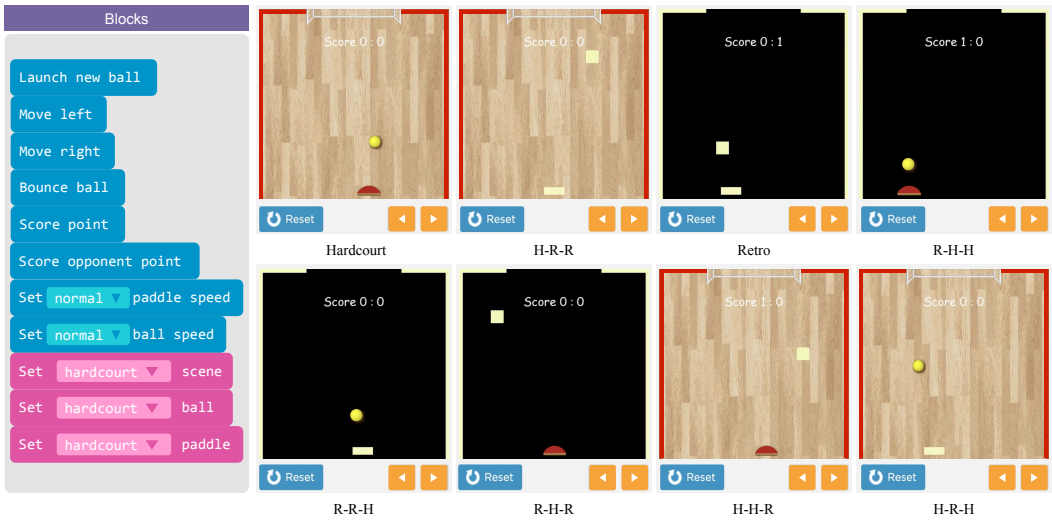


Figure 1: Bounce can have different "themes" for the background, paddle, and ball. There are two themes to choose from: "hardcourt" and "retro". We show the complete eight different combinations of themes and what the game would look like under these settings.

MDP $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$. In our setting, we assume that since this is a homework assignment, students are expected to deliver a product that matches teachers' expectation. Therefore $\mathcal{A}^\star = \mathcal{A}$. However, the dynamics P and $P^\star$ can be different, as the submitted programs might contain errors.

Observable State Space In an ideal setting, students will implement API in their programs so the internal state of the game can be accessed as the state space for the agent (i.e., coordinates of game objects). However, designing such API that can capture game's fully behavior can be cumbersome on the behalf of the instructor. Implement an API for the purpose of grading is not a fun experience for the students, tangential to the goal of teaching, and such API can be implemented wrongly — the submitted game is correct while the student API provided the wrong object's state.

In order to reduce frustration on behalf of the student and the instructor, we choose the state space to be the pixel output of the game. This assumption imposes the least amount of modification work on hundreds of thousands of games that teachers have created for students over the years. It also allows us to draw inspirations from recent reinforcement learning advances made in pixel state space (Mnih et al., 2013).

This decision poses challenges to our agent. When teachers specify a game as an assignment, the game mechanisms are the criteria on whether the game is correct. Other features such as background pictures, shape or color of the game objects, moving speed, can all be consider as "creative expressions" from the students. Some of these creative expressions only affect game aesthetics, but some will affect how the game is played (i.e., changing the speed of an object). When we use pixel state space, both may constitute as major problems for our agent.

Reward Difference We further assume that the game has a score-keeping mechanism and this can be accessed both in reference programs and submitted programs. Even if the submitted programs did not provide API for scores, most games display the score on the screen. One can easily use a transcription algorithm to transcribe visually displayed scores to actual numbers.

However, being able to access reward does not mean $R = R^\star$. A particular type of error that the student can make is to give reward at wrong places. For example, in the game of breakout, instead of earning a point after hitting a brick, the student can let players win points after the ball bounces on the paddle or on the wall, making $R^\star(s) \neq R(s)$.

Another possible scenario is that the student assigns reward on a completely different numerical scale compared to the reference programs. For example, "shooting a ball into the goal" can equal to 100 points specified by students, while the reference program only gives 1 point.

Classification Given a submitted program that fully specifies an MDP $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$, and a policy $\pi^\star$, we can evaluate this policy on this MDP and generate a trajectory $\boldsymbol{\tau} = (s_0, a_0, r_0, s_1, a_1, r_1, ...)$. Note that this trajectory is generated by $a_t \sim \pi^\star(\cdot|s_t)$, $r_t = r(s_t, a_t)$, and $s_{t+1} \sim P(s_{t+1}|s_t)$.

Sometimes, we also have access to a value function $V^\star(s_t)$. We will describe how we train the policy and the value function in later section. Using τ and other estimations our RL algorithm provides, we build an additional classifier on top of these features to render a decision on whether the submitted program is correct or broken.

This definition of task is broader than detecting bugs in educational games. It also fully captures a range of other real-world tasks that include an MDP and a classification decision. For example, in quality assurance, the task of using a robotic arm to interact with a gadget to determine if it is faulty can be described in this framework as well.

2.1 Bounce Game Simulator

We pick Bounce as the coding game of our choice. Bounce is a block-based educational game created to help kids understand if-then style conditionals[1]. We show actual game scenes in Figure 1, and we show the coding area below:

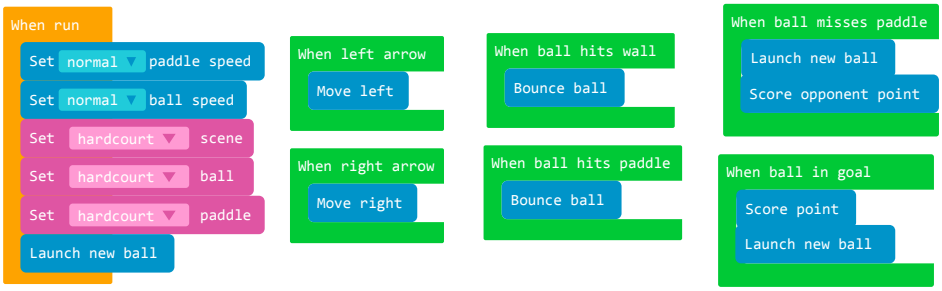


Figure 2: This is the drag-and-drop style code editor that students use to create the game Bounce. Conditions marked by orange or green, such as "when run" or "when left arrow", are fixed in space. Students place commands marked by blue or purple blocks, such as "score point", under each conditions.

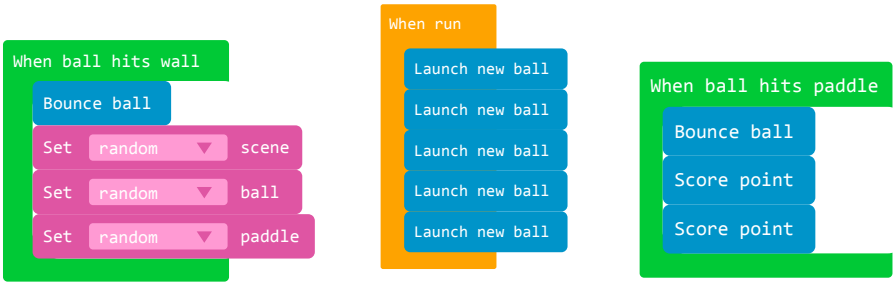
We pick this game for three reasons: first of all, this game is a very popular coding game in Code.org. The popularity gives us an abundance of real student submissions over years. Secondly, a block-based program can be easily represented in a structured format, eliminating the need to create complicated game engine that can parse and understand custom scripts. Last reason is that in order to make a realistic benchmark that can measure progress in Play to Grade, we need to be able to provide gold labels for each submission. If students programs were written in industry standard language (like Java, Python or UnrealScript), we will not be able to create gold labels programmatically. However, with Bounce, we can just specify a list of legal and illegal commands under each condition.

We use PyGame, a popular gaming library in Python as the rendering and control backend. We simulate physical events, such as collision, bouncing, moving, with PyMunk, a 2D rigid body physics library. We represent the block-based programs as JSON strings.

The Bounce exercise does not have a bounded solution space, similar to other exercises developed at Code.org. This means that the student can produce arbitrarily long programs, such as repeating the same command multiple times (Figure 3(b)) or changing themes whenever a condition is triggered (Figure 3(a)). These complications can render very different game settings depicted in Figure 4.

Reward Design and Termination In order to train an RL agent in this environment, we need to design the reward space and game ending condition. We set the termination condition to be: after winning or losing 1 point, the game terminates. The reward schema is that if the agent wins 1 point,

[1] `https://studio.code.org/s/course3/stage/15/puzzle/10`



(a) Hit wall change to random theme

(b) Multiple balls when run

(c) Hit paddle wins two points

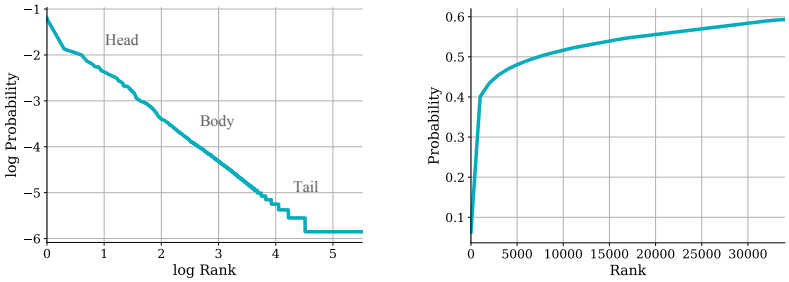
Figure 3: We demonstrate three examples of how Bounce can be programmed freely by allowing infinite repetition of the same command and allow commands to be placed under all conditions. Only (a) is considered correct since theme change does not affect game play. Both (b) and (c) are considered wrong. (c) represents a reward design error (the game should only give player points when they successfully shoot the ball into the goal).

Figure 4: The left two panels show how theme change is triggered after the ball hits the wall, corresponding to the code in Figure 3(a). The rightmost panel corresponds to code in Figure 3(b) where at run time, five balls were launched.

it will translate to a +20 reward, and if the game loses 1 point, it will translate to -10 reward. The asymmetry is designed to encourage the agent to focus on shooting the ball through the goal. Every other state in this game has 0 reward. In evaluation, we measure if the agent can score 5 points and win the game, with a final winning provides 100 reward and losing game provides -100 reward.

Reference programs We provide reference programs for the Bounce environment. These programs can be regarded as instructor's implementations of the assignment. They can be correct or incorrect. We use these reference programs to teach the agent how to play the game and how to differentiate between correct and broken programs. The selection of the reference programs is paramount to the success of the algorithm. We separate the reference programs into four groups:

- **Default program**: the simplest working program that is considered correct. The speed setting is "normal", and the theme setting is "hardcourt". This is the default setting (when students choose not to use any theme setting or speed setting). This program is displayed in Figure 2.
- **Mixed-theme program**: the program will randomly sample one of 8 theme variations displayed in Figure 1. The speed setting is "normal".
- **Mixed-speed program**: the program will randomly sample one of 25 speed variations, the theme is set as "hardcourt".
- **Error programs**: programs that are written by the instructor to demonstrate one or more implementation errors.



(a) Probability mass function (PMF) log-log plot

(b) Cumulative distribution function (CDF) plot

Figure 5: We show that the probability distribution of student submissions conform to a Zipf distribution, evidenced by the log-linear relationship between the program's rank and its frequency.

We assume that the instructor will always have a default program as reference implementation. Error programs are a small number of programs written by the instructor. In our task, we have 10 reference error programs. We will conduct ablation experiment to show the influence of the rest of the reference programs on the RL agent's performance. We regard this as an RL generalization challenge.

2.2 CODE.ORG BOUNCE DATASET

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 323,516 unique programs were submitted.

Evaluation Metric In an unbounded solution space, the distribution of student submissions incur a heavy tail, observed by Wu et al. (2019). We show that the distribution of submission in dataset conforms to Zipf distribution. This suggests that we can partition this dataset into three sections: we define the *head* of the Zipf distribution as the top 100 most frequent programs, *tail* as any program that has less than 5 submissions, and *body* as everything in between. We mark the segmentation on Figure 5(a).

We evaluate on two metrics. One is whether the trained RL agent can perform in all variations of the game setting, as well as action-triggered theme change, or learn to play with multiple balls on screen. The other metric is how accurate we can determine the submitted program to be correct or in correct. We will report three accuracy scores for the programs in the head, body, and we will measure tail accuracy by sampling 1000 programs from the tail distribution.

Gold Annotations We generate the gold annotations by defining illegal commands under each condition. For example, having more than one "launch new ball" under "when run" is incorrect. Placing "score opponent point" under "when run" is also incorrect. Abiding by this logic, we put down a list of legal commands and a list of illegal commands for each condition. Thematic changes such as "set 'hardcourt' scene" under any condition is considered legal, since they do not alter game play. Speed changes under any condition is also considered legal, but they may result in unplayable conditions, such as "very slow paddle" and "very fast ball".

3 RELATED WORK

Education feedback Feedback from visual output Yan et al. (2019).

Mike Wu's paper Wu et al. (2019)

Borrow the citations from Mike's paper? http://stanford.edu/ cpiech/bio/papers/zeroShotCode.pdf

Automated Testing in Software Design There are other works in detecting bugs in the field of automated software testing. One of the most noticeable recent work uses multi-objective evolutionary deep reinforcement learning techniques to explore as many states in a game as possible to trigger potential bugs Zheng et al. (2019). There are several problems of this approach in our setting: they have access to a discrete representation of game states (as a count) where they can use as reward, while our setting does not have such state representation. The sole objective of the algorithm is to "trigger" a bug (reaching the abnormal state) which is defined apriori by programmers; while in our setting, we do not know explicitly which state corresponds to a bug.

Gneralization in RL There are similarities between our environments and procedurally generated environments,

4 METHOD

In order to determine if a game is implemented correctly or not, an agent needs to be able to interact with it and generate meaningful observations. Then based on these observations and estimates from the RL algorithm, we need to determine whether the submitted program is correct. Therefore, we break the method section down to two parts.

4.1 LEARNING TO PLAY

We use Policy Proximal Optimization (PPO), a state-of-the-art on-policy RL algorithm that can quickly learn a discrete control policy (Schulman et al., 2017).

Given an observation s_t, we first use a convolutional neural network (CNN), the same as the one used in Mnih et al. (2013) as feature extractor over pixel observations. To accumulate multi-step information, such as direction of movement and velocity, we use a Long-short-term Memory Network (LSTM).

PPO utilizes actor-critic style training by training for an estimation for value function: $V_\phi^\pi(s_t)$ as well as an estimation for policy function: $\pi_\theta(a_t|s_t)$. The value function of an MDP is defined as:

$$V^\pi(s_t) = \mathbb{E}_{\tau\sim\pi_\theta}\big[\sum_{t=0}^{T}\gamma^t r_t | s_0 = s_t\big] \tag{1}$$

PPO waits until the trajectory is completed before computing the value for each step of the trajectory. This empirical sum of discounted reward is computed on one trajectory sampled from π_θ. We call this V_t^{target}. The loss is computed as follow:

$$\mathcal{L}_V(\phi) = \mathbb{E}_{\tau\sim\pi_\theta}\big[(V_\phi^\pi(s_t) - V_t^{target})^2\big] \tag{2}$$

Using the value function estimator defined above, PPO estimates the advantage function $A_t = A^\pi(s_t, a_t)$ using Generalized Advantage Estimator (GAE) (Schulman et al., 2015) and then use it to compute the policy gradient and update the policy network π_θ.

The main challenge is to train an agent that can operate in environments specified by student programs. This is a generalization challenge. Using the reference programs provided by teachers, we can formulate the following training strategies and measure the corresponding agents' generalization ability.

Standard Training We train the agent using the game specified by the default program. This program is displayed in Figure 2 and the corresponding game environment is displayed in the top left-most panel in Figure 1. We train the agent in this environment with reward for bouncing the ball into the goal, and penalty for failing to catch the ball with the paddle. We train the agent until it solves the game. Details of the reward design and training parameters are described in the Appendix.

Mixed-theme Training If the agent has only seen pixel observations from the default program, there is no reason to expect that it will perform well in the other 7 thematic variations. We continue training the agent that has completed the standard training by using the mixed-theme reference program, where at the beginning of game, one of the 8 thematic variations will be randomly sampled. We train this agent until it solves the mixed-theme initialized game.

Even though it is unlikely that a teacher will have a reference program that can emulate *all* of students' creative choices, we propose this training strategy as an upper-bound to the performance of an agent that can generalize to the entirety of the observation space.

Generalizable Reinforcement Learning Reinforcement learning with data augmentation (RAD) is the state-of-the-art algorithm on learning generalizable representations from pixel observation space Laskin et al. (2020). They proposed a domain-agnostic technique that allows agents to generalize to scenes that have not been observed in the training environments. These techniques consist of image augmentation techniques. We adopt the top-performing augmentations (**cutout**, **cutout-color**, **color-jitter**, **gray-scale**) to our agent that solved the standard game and train the agent till it solves the new augmented game. The augmentation is applied stochastically — agent will observe augmented images for 50% of the time, and original images for 50% of the time. We describe these augmentations in the Appendix.

4.2 Learning to Grade

Since training an agent that can generalize to novel environments in itself is a challenging task, in order to avoid the epistemic uncertainty around whether the observation comes from an incorrect programs or the observation comes from the failure of the agent to correctly interact, in this section, we always assume that we have a near-optimal agent that can operate successfully in most correct programs. We will show that by training with a mixed-theme strategy, such near-optimal agent is obtainable for our restricted domain.

We build a feature-based classifier that is trained on data collected from 23 reference correct programs and 10 reference incorrect programs. We evaluate the performance of this classifier on student submitted programs. When we evaluate our agent on the environment generated by a program, we obtain a full trajectory of a complete run: $\boldsymbol{\tau} \sim \pi$ where $\boldsymbol{\tau} = (s_0, a_0, r_0, s_1, a_1, r_1, ...)$, and we also have estimated value of each state: $\boldsymbol{v} = (V^\pi(s_0), V^\pi(s_1), V^\pi(s_2), ...)$. The estimated value of a given state is computed from Equation 1 and represent a sum of expected discounted future reward. In a near-optimal agent, a upward trend in $V^\pi(s_t)$ indicates that the agent anticipates a reward in the near future. We show this relationship in Figure X.

We can build a simple N-th order Markov sequence generative model to reconstruct the observed reward and we train by negative log-likelihood. Let $\boldsymbol{r} = (r_0, r_1, r_2, ...)$ be the sequence of reward returned by the environment. For simplicity, we categorize the reward to: $r_t \in \{-1, 1, 0\}$ that represent negative, positive and no reward. This model decomposes the sequence probability $p(\boldsymbol{r}|\boldsymbol{v})$ by the N-th order Markov assumption:

$$p(r_0, r_1, r_2...|\boldsymbol{v}) = p(r_0) \prod_{t=1} p(r_t|v_{<t-1}) \approx p(r_0) \prod_{t=1} p(r_t|v_{t-n+1}, ..., v_{t-1}) \tag{3}$$

With this model, we can easily reconstruct $\boldsymbol{r}$ by choosing reward that maximizes this distribution: $\hat{\boldsymbol{r}} = \arg\max_{\boldsymbol{r}} p(\boldsymbol{r}|\boldsymbol{v})$. We train this model only on the reference correct programs to learn how agent anticipates reward in the correct environment. For submitted programs, we use the $\boldsymbol{v}$ collected in the environment, reconstruct expected reward $\hat{\boldsymbol{r}}$ and compute the hamming distance between the reward returned by student submitted program's environment. Therefore, two features we used are:

$$\phi_r(\boldsymbol{\tau}) = \sum_t r_t \tag{4}$$

$$\phi_{v,r}(\boldsymbol{r}, \boldsymbol{v}) = \text{distance}(\boldsymbol{r}, \hat{\boldsymbol{r}}) / \max(|\boldsymbol{r}|) \tag{5}$$

By predicting what the observed reward should look like and compare with student submitted program's actual returned reward, we built a feature that can catch reward design errors — errors like

"score point" under "when hit wall". Such error can hardly be distinguished by total reward but will result in high hamming distance between $\hat{\boldsymbol{r}}$ and $\boldsymbol{r}$.

5 RESULT

5.1 LEARNING TO PLAY IN GENERALIZED ENVIRONMENTS

Thematic generalization We evaluate the generalization performance of agents trained under different strategies. We provide a full set of hyperparameters and model architecture in Appendix. Each agent is trained in total of 6M steps. Except for "Mixed-Theme" training strategy, the rest of the agents are trained only on the "Hardcourt" setting (H-H-H), with RL generalization strategies providing random data augmentations on the observations.

In Table 1, we organize the game setting from most similar to standard program specified game to the least similar to standard program specified game. We use "H" to refer to "Hardcourt" setting, and "R" to refer to "Retro" setting. The evaluation environments only have thematic differences and are all considered as correct programs.

We give 20 points for each ball in goal, and -10 points for each ball that missed the paddle. In the end, if the agent can shoot 5 balls in, it wins 100 extra points; if the agent loses 5 balls, it gets -100 points. This conveniently puts the reward range to be [-150, 200].

	Standard	Mixed-Theme	Color-jitter	Gray-scale	Cutout	Cutout-color
Hardcourt	132.0 ± 60.3	157.0 ± 50.7	-21.0 ± 68.7	**195.0** ± 3.1	153.0 ± 39.1	150.0 ± 38.2
H-H-R	-140.0 ± 6.1	**174.0** ± 22.2	-138.0 ± 8.1	-140.0 ± 8.2	-138.0 ± 14.7	-109.0 ± 28.5
H-R-H	-113.0 ± 25.3	**80.0** ± 75.2	-124.0 ± 14.5	-132.0 ± 6.6	-124.0 ± 9.5	-95.0 ± 54.2
H-R-R	-130.0 ± 12.2	**92.0** ± 76.6	-144.0 ± 5.6	-134.0 ± 10.7	-130.0 ± 7.7	-134.0 ± 7.3
R-H-H	-124.0 ± 13.4	**139.0** ± 59.5	-142.0 ± 6.0	-130.0 ± 10.9	-120.0 ± 11.3	-136.0 ± 9.5
R-H-R	-138.0 ± 8.1	**149.0** ± 51.8	-140.0 ± 6.1	-132.0 ± 8.6	-132.0 ± 11.5	-138.0 ± 6.0
R-R-H	-142.0 ± 6.0	**131.0** ± 71.1	-138.0 ± 8.1	-136.0 ± 12.3	-124.0 ± 12.3	-132.0 ± 11.5
Retro	-107.0 ± 54.9	**113.0** ± 64.3	-136.0 ± 7.8	-142.0 ± 8.1	-120.0 ± 13.7	-134.0 ± 15.3

Table 1: We show the average total reward for 10 runs from agent trained with different strategies. The confidence interval is computed with 90% confidence level. We evaluate the agent with eight theme variations. We refer to other settings in the order of background, paddle, ball (i.e., H-H-R refers to Hardcourt background, Hardcourt paddle, Retro ball).

As seen in Table 1, under each training strategy, except for **color-jitter**, agents can reach a consistently high performance on the original Hardcourt setting. However, even with data augmentation over pixel observations, agents fail to generalize to new environments. This suggests that generalization achieved through image augmentations might not have helped agent learn the true state space that's underlying pixel-level observations.

Speed generalization Other than changing themes of game objects, Bounce also allows the change of game object's speed. THere are five speed settings for each object. We trained all our agents under the "normal paddle" speed and "normal ball" speed setting. Since we have established that mixed-theme trained agent is best performing for all thematic differences, here we evaluate its ability to generalize to other speed settings.

From Table 2, we note that unlike human players, who find more challenging to play when both ball and paddle moves "very fast", the agent can generalize easily to environments when both ball and

	very slow ball	slow ball	normal ball	fast ball	very fast ball
very slow paddle	-108.0 ± 10.2	-106.0 ± 30.3	-99.0 ± 57.3	-75.0 ± 50.6	-111.0 ± 28.7
slow paddle	-13.0 ± 51.1	-10.0 ± 75.9	-1.0 ± 73.1	54.0 ± 84.7	-2.0 ± 68.8
normal paddle	146.0 ± 39.0	184.0 ± 5.6	160.0 ± 47.2	103.0 ± 77.3	180.0 ± 4.7
fast paddle	147.0 ± 46.3	135.0 ± 58.2	125.0 ± 43.8	171.0 ± 31.4	197.0 ± 3.9
very fast paddle	163.0 ± 38.3	179.0 ± 23.4	152.0 ± 52.4	170.0 ± 25.6	149.0 ± 45.3

Table 2: We show the 10-run average reward for the mixed-theme trained agent under different speed variations.

paddle move faster than the speed it was trained on. This might indicate that generalization in RL is not just about visual generalization. Slower paddle and ball means the agent needs to foresee longer horizon in order to anticipate events in more distant future.

5.2 Grading Performance

We train a grading classifier using the reference correct and incorrect programs. There are 23 correct reference programs and 10 incorrect reference programs. The feature-based classifier is trained using data collected by evaluating the agent on these programs. In order to create a balanced dataset, we evaluate each incorrect reference program 16 times, and each correct reference program 8 times.

For the sequence generative model, we use a simple multi-class logistic regression that is trained on a sliding window of past $\boldsymbol{v}$. For the feature classifier, we use a fully connected neural network with one hidden layer of size 100 and tanh activation function. As a baseline comparison, we additionally build a code-as-text classifier, where we represent the program as a count-based vector, where we count the number of coding blocks placed under every condition.

	Student Program (N=100, Head)	Student Program (N=7958, Body)	Student Program (N=500, Tail)
Logistic Regression			
Code-as-text	94.0%	82.6%	70.4%
Total-reward	90.0%	86.4%	76.6%
Total-reward + Trajectory	93.0%	91.1%	82.8%
Fully Connected Network			
Code-as-text	74.0%	71.6%	62.2%
Total-reward	94.0%	92.0%	84.0%
Total-reward + Trajectory	**94.0%**	**93.1%**	**84.0%**

Table 3:

6 Discussion

In this paper, we showed even under an egregiously simplified setting, this task is still extremely difficult even in the presence of state-of-the-art solutions. One promising direction is to go beyond a binary classification — provide a fine-grained diagnoses of behavioral errors for each submitted program.

Another direction is to allow the agent to learn to play the game that will maximally expose potential errors — not just to learn to reach each state uniformly, but to intentionally go to states that will most likely trigger an error. This direction is crucial for games where the physics engine is also

programmed by students. Errors such as "sticky actions", will require the agent to use the paddle to catch the ball at different impact angles. This type of behavior is well beyond typical game play and must be trained intentionally.

Discoverability Some bugs are hard to discover. Letting the agent interact with the environment with a pre-determined number of steps and assigning a specific reward is a human decision after all. There are settings where the reward distribution and reward trajectory do not reveal a difference between a incorrect program and correct program. In this case, other algorithms that can automatically adjust hyper-parameters that controls the agent's behavior (including how many steps to take) need to be implemented to ensure maximal success.

7 Conclusion

Overall, by formalizing educational game grading, we illustrated how challenging this task is, and

References

Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

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

Karl Cobbe, Christopher Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning. *arXiv preprint arXiv:1912.01588*, 2019.

Joe W Duran and Simeon C Ntafos. An evaluation of random testing. *IEEE transactions on Software Engineering*, (4):438–444, 1984.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pp. 213–223, 2005.

Michael Laskin, Kimin Lee, Adam Stooke, Lerrel Pinto, Pieter Abbeel, and Aravind Srinivas. Reinforcement learning with augmented data. *arXiv preprint arXiv:2004.14990*, 2020.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Rui Wang, Joel Lehman, Aditya Rawal, Jiale Zhi, Yulun Li, Jeff Clune, and Kenneth O Stanley. Enhanced poet: Open-ended reinforcement learning through unbounded invention of learning challenges and their solutions. *arXiv preprint arXiv:2003.08536*, 2020.

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, pp. 782–790, 2019.

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*, pp. 119–125, 2019.

Yan Zheng, Xiaofei Xie, Ting Su, Lei Ma, Jianye Hao, Zhaopeng Meng, Yang Liu, Ruimin Shen, Yingfeng Chen, and Changjie Fan. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 772–784. IEEE, 2019.

A APPENDIX

You may include other additional sections here.