A Fast and Accurate Machine Learning Autograder for the Breakout Assignment

Evan Zheran Liu
Stanford University, Imbue
Stanford, CA, USA
evanliu@cs.stanford.edu

David Yuan
Stanford University
Stanford, CA, USA
davidy02@stanford.edu

Ahmed Ahmed
Stanford University
Stanford, CA, USA
ahmedah@stanford.edu

Elyse Cornwall
Stanford University
Stanford, CA, USA
cornwall@stanford.edu

Juliette Woodrow
Stanford University
Stanford, CA, USA
jwoodrow@stanford.edu

Kaylee Burns
Stanford University
Stanford, CA, USA
kayburns@stanford.edu

Allen Nie
Stanford University
Stanford, CA, USA
anie@stanford.edu

Emma Brunskill
Stanford University
Stanford, CA, USA
ebrun@cs.stanford.edu

Chris Piech
Stanford University
Stanford, CA, USA
cpiech@stanford.edu

Chelsea Finn
Stanford University
Stanford, CA, USA
cbfinn@cs.stanford.edu

ABSTRACT

We detail the successful deployment of a machine learning autograder that significantly decreases the grading labor required in the Breakout computer science assignment. This assignment — which tasks students with programming a game consisting of a controllable paddle and a ball that bounces off the paddle to break bricks — is popular for engaging students with introductory computer science concepts, but creates a large grading burden. Due to the game's interactive nature, grading defies traditional unit tests and instead typically requires 8+ minutes of manually playing each student's game to search for bugs. This amounts to 45+ hours of grading in a standard course offering and prevents further widespread adoption of the assignment. Our autograder alleviates this burden by playing each student's game with a reinforcement learning agent and providing videos of discovered bugs to instructors. In an A/B test with manual grading, we find that our human-in-the-loop AI autograder reduces grading time by 44%, while slightly improving grading accuracy by 6%, ultimately saving roughly 30 hours over our deployment in two offerings of the assignment. Our results further suggest the practicality of grading other interactive assignments (e.g., other games or building websites) via similar machine learning techniques. Live demo at https://ezliu.github.io/breakoutgrader.

SIGCSE 2024, March 20–23, 2024, Portland, OR, USA

ACM ISBN 979-8-4007-0423-9/24/03
https://doi.org/10.1145/3626252.3630759

CCS CONCEPTS

• **Social and professional topics** → **CS1**.

KEYWORDS

Machine Learning, Autograder, Grading Support, Graphics, CS1, Feedback

ACM Reference Format:
Evan Zheran Liu, David Yuan, Ahmed Ahmed, Elyse Cornwall, Juliette Woodrow, Kaylee Burns, Allen Nie, Emma Brunskill, Chris Piech, and Chelsea Finn. 2024. A Fast and Accurate Machine Learning Autograder for the Breakout Assignment. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024), March 20–23, 2024, Portland, OR, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3626252.3630759

1 INTRODUCTION

Computer science instructors often task students with developing interactive applications such as games and websites. These assignments are particularly engaging and motivating for students [25] and are appearing more and more in computer science curriculum [5], but are particularly time-consuming for instructors to grade. One university CS1 course assignment is to develop the game of Breakout (see Figure 2), which on average takes graders over 8 minutes per student submission to manually play the programmed game and find bugs. As demand for computer science education increases, this grading cost becomes particularly burdensome: the Breakout assignment creates 45+ hours of grading work for a single assignment in a standard university course offering. Automated tools for helping instructors grade faster have the potential to reduce this burden and lead to more widespread adoption of this engaging kind of assignment.

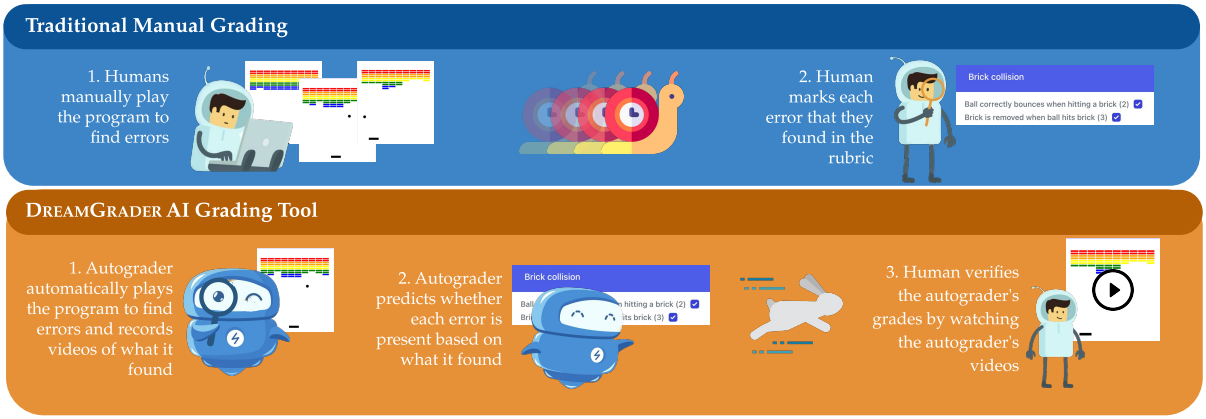

Figure 1: Grading Breakout traditionally requires manually playing each student program to find the errors, which is extremely time consuming. Instead, our autograder to automatically plays and find errors in each program, saving a lot of time. The autograder further automatically assigns grades based on what it found, and submits the predicted grades, as well as videos of what it observed to a human for verification.

Many existing automated tools for grading programming assignments are designed for analyzing student code directly [2, 15, 28–31], e.g. by comparing student code with solution code or other graded programs. Such tools are best suited for short student code snippets, and thus would be challenging to extend to the substantially longer and more open-ended code that underlies interactive games and websites. Alternatively, approaches that find mistakes by *interacting* with the student-programmed application have shown promising results in offline research studies [13, 21], but have not yet shown success in live use-cases.

In this work, we choose to build on one of such methods, specifically DreamGrader [13], which uses example graded assignments to train a neural network agent that interacts with and finds mistakes in games, and which focused on the Bounce assignment from Code.org [4]. Bringing DreamGrader from a research project into a live deployment in a university course is non-trivial for multiple reasons. First, university courses typically assign more complex games than Bounce; we will focus on the Breakout assignment in Python. Second, unlike the Code.org assignment, the Python-based Breakout assignment does not have any existing data of graded assignments. Because of the lack of data, we programmatically generate student programs with different combinations of mistakes, leading to a representative dataset of 16386 synthetic programs. Finally, assigning grades to real students has high stakes, and only a human-in-the-loop tool can guarantee high-quality grades. Therefore, we develop a grading interface that, for each rubric item, allows course assistants to leverage both the predicted grade and a video of item-relevant program behavior generated by DreamGrader.

The main contribution of this paper is the development and deployment of an artificial intelligence (AI) based grading tool for the Breakout assignment. Our autograding interface uses a machine learning model to help instructors in two different ways. For each rubric item, it first displays a video of the machine learning model playing the game and exposing whether or not an error was made. Second, it pre-fills the rubric with the machine learning model's predictions, while allowing instructors to modify the rubric grades. We deployed the tool in an university introductory computer science course in Spring 2023, and at the request of the instructor, in the Summer 2023 course offering as well. Beyond the deployment, the results of our randomized A/B test indicate that the tool reduces grading time by 44% while improving grading accuracy by 6%. These results suggests that the tool saved roughly 30 hours of course assistant time over the two deployments. We plan to open-source our complete autograding tool and hope that it, and similar autograders developed for other interactive assignments, will make it practical for more courses to offer engaging, interactive assignments.

2 RELATED WORK

Game-based assignments. Highly interactive programs are a common element of many computer science curricula. There is a lot of research into the benefit of building games in CS education [1, 10, 12, 20]. Well-designed games have also been shown to be an effective pedagogy for both men and women in the research on girls, computer science, and games [3]. Breakout, in particular is a classic assignment in CS education. First presented as a "nifty" assignment in SIGCSE 2006[1] [24], it has become a common assignment both in courses [22, 26, 27] and research [33].

AI-based autograding and grading assistance. Autograding has been a core focal point of CS education for reducing human grading burden [7, 23]: with many newer systems incorporating AI-based technology such as deep learning [16, 32]. There has also been interest in using AI to help human instructors efficiently provide richer forms of feedback to students in introductory programming courses (e.g. [6, 9]).

Recently, AI for grading interactive work has been an active area of research in the reinforcement learning literature. The problem of playing student work was first posed in 2021 [21] and was expanded on in 2022 [13]. Both of these works were developed on simple assignments from Code.org, and presented exciting proof-of-concept results. This paper extends those results to a CS1 level assignment and deploys the grading system with real graders in a live course.

Grading user experience. There is a long history of research into into how to present feedback to students and graders in SIGCSE and beyond [8, 11, 17], including work to improve the grading

[1] http://nifty.stanford.edu/2006/roberts-breakout/

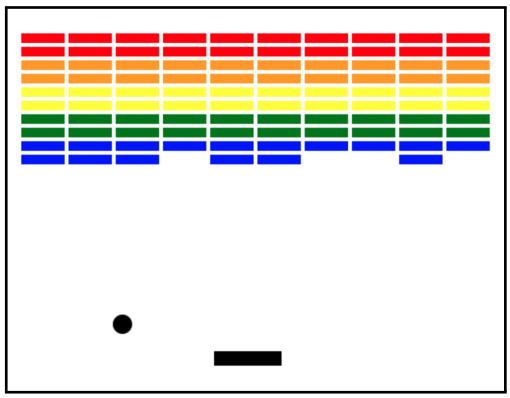

Figure 2: The Breakout game. Several colored bricks have already been broken by bouncing the ball off of the paddle to hit the bricks.

experience [18, 19, 19, 34], which inspire our design choices to make an experience that is usable, accessible and fair.

3 THE BREAKOUT ASSIGNMENT

We now detail the Breakout assignment for which we have built a new autograder. In this assignment, students are tasked with programming the classic Breakout arcade game in Python, displayed in Figure 2. A correct implementation should include a mouse-controllable paddle that can move left or right, a ball, and several rows of bricks. The ball should bounce off the paddle and walls, and break bricks upon collision. Breakout has gained popularity as a fun and engaging way to learn introductory computer science concepts, such as decomposition, control flow operations, and basic graphics. Consequently, it has been offered at multiple universities, as well as high school classrooms, though the associated immense grading burden has prevented its further widespread adoption.

Grading challenges. Breakout is typically graded according to an instructor-written rubric, which lists a set of errors detailed below. Graders deduct points for each error present in a student's program. The main bottleneck of grading is that there have been historically no effective ways for automatically testing for these errors.

Breakout defies traditional input-output unit tests, due to its interactive and stochastic nature (e.g., the initial velocity of the ball is randomized). There have been attempts to automatically test program behavior via hardcoded algorithms of controlling the paddle, but the resulting autograders are extremely brittle, breaking if the student even slightly misplaces the paddle. Furthermore, student implementations range between 200 and 400 lines of code, which is equivalent to roughly 3-6 pages of text. This makes grading by simply reading the code infeasible.

Consequently, the prevailing method for grading is to manually interact with each student submission and exhaustively check if each error in the rubric is present. Unfortunately, this can be extremely time consuming and error prone, as certain game states can be difficult to reach. For example, one common error is "paddle skewering," where the ball can become skewered if it is struck by the paddle on the side. As the ball is small and falls quickly, manual testing can take many attempts. Another example is checking if the program correctly terminates if all bricks are broken, as this requires either manually breaking all of the bricks, or modifying the student's code to create fewer total bricks.

Grading rubric. To understand the types of errors that graders check for, Table 1 outlines the rubric of errors that graders uses.

4 THE DREAMGRADER AUTOGRADER

In this section, we detail the machine learning techniques behind our autograder and how instructors interact with it to grade more quickly. Our autograder builds on the insight that the time-consuming aspect of grading is manually entering game states that indicate the presence or absence of errors in the student program. In contrast, determining whether the correctness of the game's behavior at such states is relatively simple and fast.

Hence, the autograder is designed to automatically and robustly reach the game states that a human would normally have to reach manually. Then, it assigns grades to the rubric items based on the observed behavior at these states. Additionally, to mitigate concerns about fully-automatic grading, the autograder also presents short videos of the observed behaviors to human graders, who can then quickly verify the correctness of the autograder's grades. This vastly accelerates the grading process by automating the slow component of playing the game (see Figure 1 for an illustration).

In the next three sections, we first provide an overview of the machine learning techniques that enable the autograder to robustly reach the necessary game states, even in student programs with unexpected or incorrect behavior (Section 4.1). Second, we describe how we apply these techniques to specifically train a Breakout autograder (Section 4.2). Finally, we detail the interface through which human graders interact with the autograder (Section 4.3).

4.1 Background on DreamGrader

We build our autograder on top of the DreamGrader framework [13] for training reinforcement learning (RL) agents to discover bugs in interactive programs. DreamGrader assumes that an instructor has already defined a rubric and uses that rubric and a dataset of programs with filled-in rubrics to train an autograder. The dataset should be representative of the kinds of buggy programs that students will write. The resulting DreamGrader autograder is a collection of agents that seek out bugs corresponding to the rubric items by interacting with the program. DreamGrader further assigns grades to each rubric item based on its observed interaction.

DreamGrader trains two key components for each rubric item, each represented with a deep neural network: First, a game-playing *policy* takes the game state as input and outputs actions; this policy is trained automatically via RL to interact with a student's program and visit states indicative of whether the rubric item is correct or not. The policy produces a *trajectory* of game states and actions. Second, a *rubric classifier* takes the trajectory generated by the policy as input and predicts the rubric item label. This classifier is trained via supervised learning using the labeled programs.

In RL, the policy learns, through trying things out, to select actions that are expected to yield a large amount of cumulative reward over a trajectory. One intuition behind the benefit of RL is that it can be easier to specify a reward function and then automatically (machine) learn a policy to optimize that function (in this case, grade the assignment well), rather than manually write down

Table 1: Outline of the rubric items that graders check to grade Breakout.

Category	Assignment Specification	Example Error(s)
Game Display	The game must display 10 rows of 10 bricks, a ball, and a paddle. The color, size, and location of each of these objects must match the assignment handout.	The number of bricks is incorrect. Changes to constants are not correctly handled.
Paddle Movement	The paddle must track the left and right mouse movements of the player.	The paddle is not centered on the mouse. The paddle follows the mouse off screen.
Ball Movement	The ball must correctly bounce off bricks, walls, and the paddle.	The ball becomes "skewered" on the paddle, when struck from the side.
Brick Collision	Bricks should break when hit by the ball.	The ball does not bounce when breaking a brick. Bricks break before collision with the ball.
Game Termination	The game should terminate when all lives are lost, or all bricks are broken. A life is lost when the ball falls to the bottom of the screen.	The game continues even when all bricks are broken. The player has too many or too few lives.

a decision policy that can effective grade a student's program by playing it and finding bugs. Hence, our work uses RL, rather than asking expert graders to manually program a grading strategy or to demonstrate how they grade sample programs, which is expensive and time-consuming to gain the detail and coverage necessary to generalize to the full range of student submissions.

The key challenge of our setting then lies in defining a reward function that incentivizes the grading policy to visit states informative of the rubric label. This could be implemented as an end-of-episode reward corresponding to the rubric classifier's accuracy; however, this creates a sparse reward signal, which can hinder optimization. Based on Dream [14], DreamGrader instead formulates a reward function that estimates information gain at each timestep in the trajectory, which corresponds to the improvement in the rubric classifier's accuracy after observing the current timestep. Once each policy and rubric classifier are trained using the labeled programs, they can be used to fill in rubrics for new programs. During deployment for a given rubric item, DreamGrader runs the learned policy to generate a trajectory in the game and then passes that trajectory to the classifier to predict the rubric item label.

4.2 Adapting DreamGrader to Breakout

From a high level, training a DreamGrader autograder on Breakout requires three main components: First, we must define how the autograder interfaces with the Breakout game, namely, what the autograder receives as input and what actions in the game it outputs. Second, we need a set of graded programs labeled with the errors contained within to train the autograder. Finally, as student programs typically take mouse movements as input, we need a way to practically hook up the autograder with student code. We describe how we tackle these three components below.

Modeling Breakout as a Markov decision process. The RL techniques that DreamGrader builds upon relies on formalizing the domain as a Markov decision process, which roughly defines the inputs and outputs of the autograder, and what happens as a result of the autograder's outputs. Formally, we let the *state* (i.e., the autograder's inputs) at each timestep be:

- The bounding boxes of the paddle and ball: i.e., the (x, y)-coordinates of the top left corner, the width and the height.
- The x and y-velocities of the ball, as well as binary indicators for whether they have flipped signs at this timestep (e.g., from bouncing off something).
- The remaining number of bricks.

We let the possible *actions* at each timestep be to move the paddle left or right by 10 pixels. Finally, each Breakout program defines the *dynamics* or what happens on the next timestep as a consequence of each action (e.g., how the ball bounces off the paddle and bricks).

Obtaining training data. Ideally, a set of training programs should maximally cover the possible behaviors the autograder may see when grading real student programs. Since manually curating such a set is time-consuming and error prone, we instead automatically generate programs by taking a reference correct program and injecting errors into it. Specifically, we take 16 of the most common implementation errors and create the set of $2^{16} = 65356$ programs of all possible combinations of these errors. The presence of some errors can overshadow the behaviors of other errors, so there are only $2^{14} + 2 = 16386$ uniquely-behaving programs. Generating programs this way is convenient, because we then know which errors are present in each program, which is required for training.

We train DreamGrader on these programs, with a separate game-playing policy and rubric classifier for each rubric item. Then, we can grade a new student program by running each policy and classifier pair. Critically, each pair needs to train on the full set of training data in order to be able to robustly handle student programs. For example, a paddle skewering policy needs to be able to test skewering the ball even in student programs with other errors, such as an incorrectly-sized paddle or incorrect bouncing mechanics.

However, we find that directly training each pair on all 16386 programs via RL often fails because the specific error that each pair is trained detect can be difficult to identify from such a diverse set of program behaviors. We mitigate this by first training each pair only on 2 programs: (1) a reference correct program; and (2) a program that contains only the error that the pair is designed to detect and no other errors. This serves as a good initialization that then enables training on all 16386 programs. Each rubric item takes between 3 to 74 hours of training on a single GTX3080.

Injecting the autograder into student code. Applying the autograder to real student code requires two components: A way for the autograder to (1) read the state from the program; and (2) feed its actions into the program.

The Breakout assignment tasks students with programming Breakout on top of a `tkinter Canvas` object, which enables the student to draw the bricks, paddle, and ball with the `create_rectangle` and `create_oval` functions. To extract the game's state from the

program, we create our own `Canvas` object that holds the autograder, and change the student's code dependencies to use ours instead at run time. Our `Canvas` object then keeps track of all of the relevant objects added and feeds this information as the state to our autograder. We also use this `Canvas` to render the videos that the autograder displays to human graders.

To track the mouse's position with the paddle, students use the `get_mouse_x()` and `get_mouse_y()` functions from the `tkinter` `Canvas` object, which returns the mouse's (x, y)-coordinates. We inject the autograder's actions into student's code by monkey patching these functions so that each `get_mouse_x()` or `get_mouse_y()` function call instead queries the autograder to determine what paddle movement to make. Then, instead of relying on the student code to move the paddle appropriately, we directly move the paddle ourselves with our `Canvas` object, so that the autograder continues to work, even if the student paddle movement implementation is broken. Accordingly, we also override any other paddle movements that the student code makes in our `Canvas`.

A final wrinkle in interfacing the autograder with student code lies in dependent constants. Some rubric items require testing the student's code after changing certain constants in the code, such as `NUM_BRICK_ROWS` or `BALL_RADIUS`. We handle these by loading the student's code and then changing these values at run time. However, students occasionally define their own dependent constants, such as `HALF_NUM_BRICK_ROWS = NUM_BRICK_ROWS // 2`, which would fail to be updated in our changes to the constants. We update these values by extracting dependencies from the abstract syntax tree and propagating all constant changes.

Special case errors. The autograder handles two categories of errors separately: First, game display errors do not require any interaction with the game, and only need to test whether the paddle, ball, and bricks are drawn correctly. To handle these, we register all the objects created via our `Canvas` object and check that their rendering matches a reference solution. Second, programmatically generating mouse movements via Python to test that the paddle correctly tracks the mouse is highly non-trivial. However, manually testing this is extremely fast and simple: graders only need to move the mouse around and check that the paddle is centered on the mouse, so we leave this to always be manually graded.

4.3 Interface for Graders

Figure 3 displays the interface through which graders use the autograder. Graders are presented with all of the rubric items on the left, which are grouped into folds and pre-populated with the predicted grades from the autograder. Graders can either confirm the pre-populated grades or overrule them by watching the videos justifying the autograder's grades on the right. Opening a new fold automatically loads the video associated with the corresponding rubric items. Additionally, graders can click on the "Code" tab to directly read or modify the student's code, as well as the "Demo" tab, to manually interact with the game defined by the student program. As videos are quite short (less than 10s), ideally grading a single student should only take a few minutes to watch each video and confirm all of the grades.

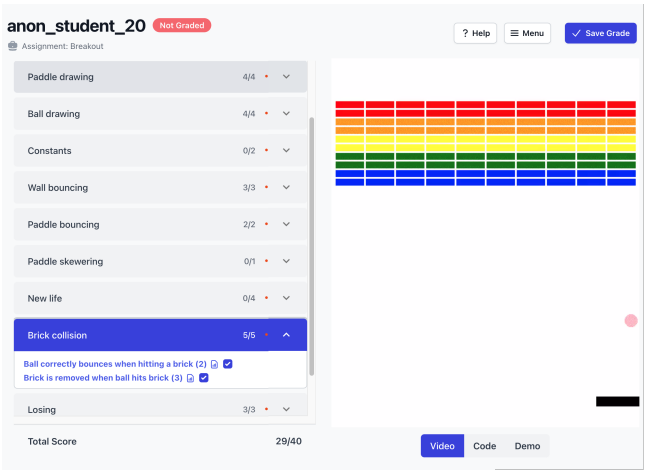

Figure 3: The user interface for our grading tool.

5 COMPARISON WITH MANUAL GRADING

To understand the practical deployability of our autograder, we compared our autograder with manual grading, aiming to answer three main questions:

(1) How does the autograder impact grading time?
(2) How does the autograder impact grading accuracy: i.e., the fraction of rubric items correctly graded?
(3) How do graders feel about using the autograder?

Experimental setup. To answer these questions, we obtained 15 volunteer graders from a pool of university computer science teaching assistants. We compared their grading with and without the autograder on 60 real student submissions to the Breakout assignment. To ensure a fair comparison between the autograder and manual grading, we controlled for three effects:

- *Submission grading difficulty.* Due to differences in the number of errors and implementation, some student submissions are more difficult to grade than others. To control for this effect, we had each submission graded twice by separate graders, once manually, and once with the autograder,
- *Grading ability.* Graders vary in their grading ability. To control for this effect, we tasked each grader with grading both manually and with the autograder, in random order.
- *Breakout grading experience.* Graders improve as they gain more experience with an assignment. To control for this effect, we randomized whether each grader graded with the autograder or manually first.

Altogether, each of the 60 submissions was graded once manually and once with the autograder. Each grader graded 4 submissions manually and then 4 with the autograder, or in the reverse order, with the order of manual vs. assisted determined at random. Graders were not permitted to revisit previously-graded submissions to prevent them from using insights gained from the autograder for previously manually-graded assignments or vice versa.

We recorded the amount of time each grader spent on each submission. To measure accuracy, we obtained ground-truth grades from an expert grader who had previously graded over 100 submissions, and double-checked cases where the expert disagreed

Table 2: Survey questions presented to graders and their average responses. Graders were asked to rate the first five statements on a Likert Scale, and then provide a Net Promoter Score for how much they would recommend using the autograder.

Likert Scale (*Strongly Disagree* = 1, *Disagree* = 2, *Neutral* = 3, *Agree* = 4, *Strongly Agree* = 5) **Statement**	**Avg. Score**
Using the autograder is easier than manually grading.	4.5
Using the autograder is faster than manually grading.	4.5
Using the autograder is more accurate than manually grading.	3.9
The autograder's grades were useful to me.	4.4
I enjoyed using the autograder.	4.6
Net Promoter Score (0 - 10 inclusive)	
How much would you recommend using the autograder over manually grading in the future?	9.0

Table 3: Grading time and accuracy results of manual grading, human-in-the-loop autograding, and human-free autograding.

Grading Scheme	Human Grading Time	Grading Accuracy
Manual	8 min 35s ± 6 min 47s	86.4% ± 8.9%
Autograder with human	**4 min 49s ± 2 min 5s**	**92.3% ± 7.6%**
Autograder only	—	90.1% ± 11.0%

with the other graders. Additionally, to answer question (3), we also asked each grader to fill out a short survey displayed in Table 2.

Grading time and accuracy results. As shown in Table 3, we found that the autograder overall significantly decreased grading time by 44% on average, while also increasing grading accuracy by 6%. Grading accuracy with the autograder matched or improved upon manual grading in every single rubric item, with the largest accuracy improvements arising in rubric items requiring long or complex manipulations, such as terminating the game by winning or losing, ensuring that the game reacts appropriately to modified constant values, and paddle skewering.

While we mainly use the autograder with a human in the loop, the autograder itself directly predicts grades for each rubric item. We also measured the accuracy of these grades without any human intervention under "Autograder only" in Table 1. Promisingly, these grades already surpass manual grading accuracy, with *no human time required.* Further, we found that these grades actually match or even improve on human-in-the-loop autograding in most rubric items, except for brick collisions errors: The autograder failed to catch errors where multiple bricks were removed at the same time, likely because its training data included no such programs. Notably, we found that graders not infrequently overruled autograder grades to give out incorrect grades. These results emphasize the importance of obtaining sufficient coverage in the training data, and suggest the possibility of surpassing even autograder-assisted humans with only the autograder via improved training data.

Survey results. Table 2 reports the average responses of graders to each survey question. Graders overwhelmingly found the autograder to be easier and faster to use than manually grading, as well as useful and enjoyable, with an average response between *Agree* and *Strongly Agree*. They also found the autograder to be more accurate than manual grading, with an average response near *Agree*. Finally, graders unanimously recommended using the autograder over manual grading, with an average score of 9 out of 10.

6 LIVE DEPLOYMENT

We first deployed our autograder for the Breakout assignment in the Spring 2023 semester of a university CS1 course of 323 students. The teaching staff enjoyed using the autograder so much that another instructor asked us to deploy the autograder again for the Summer 2023 semester offering of the same course, with 199 students.

We could not obtain accurate measurements of grading accuracy or time, as each assignment was only graded once, and graders paused grading at their own leisure. However, based on the results from our comparison with manual grading in the previous section, our two deployments of the autograder saved roughly 32 hours.

Graders in the live deployment optionally left feedback via the same survey in Table 2, along with a space for free-form feedback. Similarly to the comparison with manual grading, graders generally recommended using the autograder with an average score of 8.6 out of 10. Interestingly, the free-form feedback primarily focused on improving the UI of the autograder, which can significantly improve grader experiences and is relatively simple to address.

7 CONCLUSION

We presented an AI-based grading tool for the Breakout assignment, and detailed its successful deployment in two offerings of a university computer science course. Our results indicated that the tool saved grading time by 44% while improving grading accuracy by 6% compared to manual grading. In the live deployment, graders left positive feedback, almost unanimously recommending it for future use. We fully open source our autograder and host a live demo at https://ezliu.github.io/breakoutgrader. We hope that such an autograder will lower the barrier for instructors to incorporate the Breakout assignment into their course in the future, especially at schools with limited manual grading capacity.

There are numerous interesting directions for future work. First, it would be interesting to explore whether students can more directly benefit from such an autograding tool, since it may provide immediate feedback that is infeasible for instructors to provide. Second, our work shows the practicality of using DreamGrader techniques for grading interactive assignments. Similar tools could in principle be built for other interactive assignments, including web design assignments, and possibly using a single generalist autograding agent, which would open the door for a more expansive array of possible interactive assignments. Finally, as our analysis indicates that the autograder alone already surpasses human-in-the-loop autograder for many rubric items, future research should investigate new human-AI interfaces that can better combine automated predictions with human expertise.

ACKNOWLEDGMENTS

We thank the volunteer graders and CS106A teaching assistants, listed at https://ezliu.github.io/breakoutgrader. EL is supported by a NSF Graduate Research Fellowship under Grant No. DGE-1656518. CF is a CIFAR Learning Machines and Brains Fellow. This work was supported by Intel and a Stanford Hoffman Yee grant. Icons in this work were made by Freepik and imaginationlol from Flaticon, and ilyakalinin from Adobe Stock Photo.

REFERENCES

[1] Paulo Battistella and C Gresse von Wangenheim. 2016. Games for teaching computing in higher education–a systematic review. *IEEE Technology and Engineering Education* 9, 1 (2016), 8–30.

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

[3] Gail Carmichael. 2008. Girls, computer science, and games. *ACM SIGCSE Bulletin* 40, 4 (2008), 107–110.

[4] Code.org. 2022. Code.org. https://code.org/about.

[5] Jeffrey E Froyd, Phillip C Wankat, and Karl A Smith. 2012. Five major shifts in 100 years of engineering education. *Proc. IEEE* 100, 0 (2012), 1344–1360.

[6] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. 2015. OverCode: Visualizing variation in student solutions to programming problems at scale. In *Conference on Human Factors in Computing Systems (CHI)*. 1–35.

[7] Georgiana Haldeman, Andrew Tjang, Monica Babeş-Vroman, Stephen Bartos, Jay Shah, Danielle Yucht, and Thu D Nguyen. 2018. Providing meaningful feedback for autograding of programming assignments. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. 278–283.

[8] Qiang Hao, David H Smith IV, Lu Ding, Amy Ko, Camille Ottaway, Jack Wilson, Kai H Arakawa, Alistair Turcan, Timothy Poehlman, and Tyler Greer. 2022. Towards understanding the effective design of automated formative feedback for programming assignments. *Computer Science Education* 32, 1 (2022), 105–127.

[9] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*. 89–98.

[10] Stan Kurkovsky. 2009. Engaging students through mobile game development. *ACM SIGCSE Bulletin* 41, 1 (2009), 44–48.

[11] Abe Leite and Saúl A Blanco. 2020. Effects of human vs. automatic feedback on students' understanding of AI concepts and programming style. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 44–50.

[12] Scott Leutenegger and Jeffrey Edgington. 2007. A games first approach to teaching introductory programming. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education*. 115–118.

[13] Evan Liu, Moritz Stephan, Allen Nie, Chris Piech, Emma Brunskill, and Chelsea Finn. 2022. Giving Feedback on Interactive Student Programs with Meta-Exploration. *Advances in Neural Information Processing Systems* 35 (2022), 36282–36294.

[14] Evan Zheran Liu, Aditi Raghunathan, Percy Liang, and Chelsea Finn. 2021. Decoupling Exploration and Exploitation for Meta-Reinforcement Learning without Sacrifices. In *International Conference on Machine Learning (ICML)*.

[15] Ali Malik, Mike Wu, Vrinda Vasavada, Jinpeng Song, Madison Coots, John Mitchell, Noah Goodman, and Chris Piech. 2021. Generative Grading: Near Human-Level Accuracy for Automated Feedback on Richly Structured Problems. *International Educational Data Mining Society* (2021).

[16] Ali Malik, Mike Wu, Vrinda Vasavada, Jinpeng Song, John Mitchell, Noah Goodman, and Chris Piech. 2019. Generative grading: Neural approximate parsing for automated student feedback. *arXiv preprint arXiv:1905.09916* (2019).

[17] Jessica McBroom, Irena Koprinska, and Kalina Yacef. 2021. A survey of automated programming hint generation: The hints framework. *ACM Computing Surveys (CSUR)* 54, 8 (2021), 1–27.

[18] Alexandra Ann Milliken. 2021. *Redesigning How Teachers Learn, Teach, and Assess Computing with Block-Based Languages in Their Classroom.* North Carolina State University.

[19] Divyansh Shankar Mishra. 2023. *The Programming Exercise Markup Language: A Teacher-Oriented Format for Describing Auto-graded Assignments.* Ph. D. Dissertation. Virginia Tech.

[20] Briana B Morrison and Jon A Preston. 2009. Engagement: Gaming throughout the curriculum. *ACM SIGCSE Bulletin* 41, 1 (2009), 342–346.

[21] Allen Nie, Emma Brunskill, and Chris Piech. 2021. Play to Grade: Testing Coding Games as Classifying Markov Decision Process. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[22] Isabela Ortiz Jaramillo et al. 2023. A summer introductory programming course with diversity awareness. (2023).

[23] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated assessment in computer science education: A state-of-the-art review. *ACM Transactions on Computing Education (TOCE)* 22, 3 (2022), 1–40.

[24] Nick Parlante, Steven A Wolfman, Lester I McCann, Eric Roberts, Chris Nevison, John Motil, Jerry Cain, and Stuart Reges. 2006. Nifty assignments. In *Proceedings of the 37th SIGCSE technical symposium on Computer science education*. 562–563.

[25] Jay A Pfaffman. 2003 2003. *Manipulating and measuring student engagement in computer-based instruction.* Ph. D. Dissertation. Vanderbilt University.

[26] Christopher Piech, Ali Malik, Kylie Jue, and Mehran Sahami. 2021. Code in place: Online section leading for scalable human-centered learning. In *Proceedings of the 52nd acm technical symposium on computer science education*. 973–979.

[27] Chris Piech, Lisa Yan, Lisa Einstein, Ana Saavedra, Baris Bozkurt, Eliska Sestakova, Ondrej Guth, and Nick McKeown. 2020. Co-teaching computer science across borders: Human-centric learning at scale. In *Proceedings of the Seventh ACM Conference on Learning@ Scale*. 103–113.

[28] Kelly Rivers and Kenneth R Koedinger. 2017. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education* 27 (2017), 37–64.

[29] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 15–26.

[30] Ke Wang, Benjamin Lin, Bjorn Rettig, Paul Pardi, and Rishabh Singh. 2017. Data-driven feedback generator for online programing courses. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*. 257–260.

[31] Mike Wu, Noah Goodman, Chris Piech, and Chelsea Finn. 2021. Prototransformer: A meta-learning approach to providing student feedback. *arXiv preprint arXiv:2107.14035* (2021).

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

[33] Lisa Yan, Nick McKeown, Mehran Sahami, and Chris Piech. 2018. TMOSS: Using intermediate assignment work to understand excessive collaboration in large classes. In *Proceedings of the 49th ACM technical symposium on computer science education*. 110–115.

[34] Jeremy K Zhang, Chao Hsu Lin, Melissa Hovik, and Lauren J Bricker. 2020. GitGrade: A Scalable Platform Improving Grading Experiences.. In *SIGCSE*. 1284.