Deep Grade: A visual approach to grading student programming assignments

Lisa Yan
Stanford University
yanlisa@stanford.edu

Nick McKeown
Stanford University
nickm@stanford.edu

Chris Piech
Stanford University
piech@cs.stanford.edu

1. Introduction

As classes get larger, the process of evaluating student performance becomes more time-consuming. While content delivery can be scaled to large classrooms in the form of webcasted lectures or online materials, grading and assessing students requires human grader input to identify areas of misunderstanding and provide individual student feedback.

In computer science courses, student programming assignments can sometimes be autograded with delicate unit tests. However, these unit tests are time-consuming to design and are specifically tailored to the assignment; they cannot be extended to other homeworks. Furthermore, unit tests are often low-level in nature and cannot handle more qualitative checks. In particular, introductory computer science programming assignments often involve interactive, open-ended graphics. Unit tests for such graphics-based assignments are consequently particularly difficult to design.

More open-ended approaches to grading computer programs for functionality often focus on the student code itself, parsing the student programs to produce abstract syntax trees (ASTs), after which the programs can be compared and evaluated [2, 4]. Yet these representations become impossibly complex when student solutions grow longer than twenty lines or contain diverse variable names.

In this project, we present Deep Grade, a deep learning approach to grading graphics-based programming assignments relying on neither the student code nor unit test development. Inspired by the Deep Mind project, a deep Q-learner for Atari games [1], we aim to develop an assistive grading system that requires only raw pixels as input for suggesting grades. We focus on Breakout, a classic brick break game used in Stanford's introductory CS 1 course [3]. Deep Grade takes video playthroughs of student-coded programs and outputs a grade according to a given rubric; it also annotates sequences of frames corresponding to each reported rubric item.

The goal of Deep Grade is to develop a detection system that can be used by human graders as an assistive technology for grading. For example, the system should detect that the bricks are correct aligned and that the paddle stays on-screen in the left frame of Breakout in Figure 1 (whereas the other frame shows incorrectly aligned bricks and a slightly off-screen paddle); it can justify its claims with timestamped output so that human graders can make the final decision. Students can also use this system as a visual medium for understanding their homework grade.

Because our system requires only a temporal, raw pixel representation of a student program run, it can be generalized and applied to many other homework assignments. The assignments supported by our system can range from animation assignments like Breakout to computer systems assignments, such as designing an Internet router, where the output can be represented graphically over time (e.g., raw network packet bits transmitted with time delays).

This abstract has three components: we define a novel problem at the intersection of computer vision and computer science education, describe a zero-shot learning method by training on a *synthetically* generated dataset of student code, and present preliminary work on a supervised learning implementation of the Deep Grade system.

2. Problem Statement

The objective of the Breakout homework assignment in Stanford's CS 1 course is to design and program a brick break game in Java using the ACM graphics library. Students apply concepts like animation loops and event handlers to move bricks, a ball, and a paddle according to simplified laws of physics. Each human grader runs a student program multiple times; in each run, the grader interacts with the program by supplying mouse input to move the

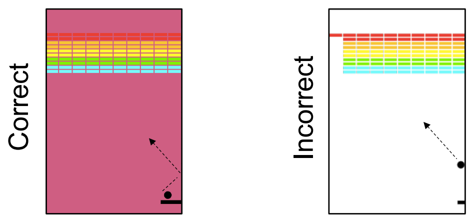
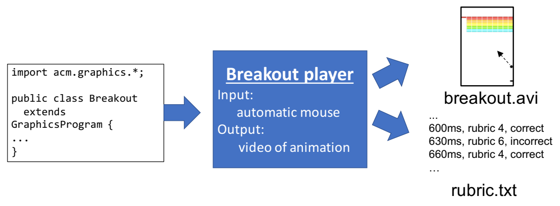


Figure 1. Examples of synthetic student Breakout programs illustrating wall bounce, where the ball leaves the paddle and bounces against the rightmost wall. Deep Grade is insensitive to ungraded image components, such as background color.



```
import acm.graphics.*;

public class Breakout
  extends
GraphicsProgram {
...
}
```

```
...
600ms, rubric 4, correct
630ms, rubric 6, incorrect
660ms, rubric 4, correct
...
```

Figure 2. Breakout playthrough and video/rubric log generation.

paddle and observes the animated object interactions. Out of 55 items in the Breakout rubric, 13 items are *static*; i.e., observable from the initial frame prior to any mouse movement. Detecting the remaining rubric items requires playing the game for as long as several minutes.

Deep Grade has two phases for grading a Breakout program: (1) record a playthrough video via automated mouse inputs, and (2) analyze video frames to suggest a rubric grade. The playthrough in phase one should be designed to observe all rubric items. In phase two, Deep Grade slides a temporal window over the input video and returns all rubric items—and their correctness—that were detected within each sliding window. For example, video sequences like those in Figure 1 passed through the system will have a wall bounce rubric item labeled as either *correct*, *incorrect*, or *none observed*. Finally, the system returns the detected sequences of frames along with a prediction of the rubric grade. These video clips can be used by a human grader to decide the final student grade.

3. Preliminary work

Dataset generation. We construct a real dataset by collecting over 5000 fully graded historic student submissions and recording 20 human grader interaction patterns from the Winter 2017 offering of CS 1. However, in order to achieve temporal detection of rubrics, we need more than one grade label per video; we require a fine-grained labeling of *when* each rubric item is observed. We therefore train our classifier on *synthetically generated* student Breakout code. We have created over 10^{18} instances of synthetic programs corresponding to all possible rubric grades. Purely synthetic data has two main benefits: first, we can achieve zero-shot learning prior to seeing any real student code; second, it allows us to open-source our entire synthetic student program dataset for other researchers to use.

Automatic Breakout play. Figure 2 shows a preliminary workflow for the first phase of Deep Grade. We implement the simplest Breakout player, where the paddle always tracks the ball motion. We insert logging functions into each synthetic program so that each playthrough generates both an animation video and a timestamped log of which rubric items are observable in each video frame.

Classifier design. The second phase of Deep Grade involves a preprocessor and classifier for grading automated Breakout playthroughs. The preprocessor compresses each sliding window into a single image by finding optical flows in the animation of the ball, paddle, and bricks. However, conventional trajectory detection algorithms for real-life video are suboptimal, as Breakout animation often involves sudden appearance (of the ball upon game reset) or disappearance (of a brick upon ball contact) between sequential frames. We therefore perform a simple weighted averaging of the window as an initial step.

We design a recurrent neural network (RNN) classifier that takes a weighted average image as input at each timestep, generates a CNN representation, and outputs a probability distribution for each rubric item. The final probability distribution is a normalized maximum of all sliding windows in the recorded video. We optimize for cross-entropy loss.

Preliminary results. In preliminary work, we operate only on *static* rubric items, which are observable from a single frame at the start of each video. A CNN model with two convolution layers and two fully connected layers trained on 10,000 instances achieves an average F1/precision/recall score of 0.85/0.87/0.87 per rubric, outperforming both a baseline logistic regression approach (F1/P/R: 0.76/0.81/0.75) and random chance (F1: 0.33).

Future work. While we are not overfitting on our training set, we expect to improve our static CNN model performance with a deeper network and further debugging of the synthetic dataset. We also would like to design a more complex loss function that accounts for co-dependency between rubric items prior to beginning work on an RNN implementation that can classify temporal data. Our large dataset allows us to increase our training set size to support robustness of future complex models.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing Atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[2] A. Nguyen, C. Piech, J. Huang, and L. Guibas. Codewebs: Scalable homework search for massive open online programming courses. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 491–502, New York, NY, USA, 2014. ACM.

[3] N. Parlante, S. A. Wolfman, L. I. McCann, E. Roberts, C. Nevison, J. Motil, J. Cain, and S. Reges. Nifty assignments. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '06, pages 562–563, New York, NY, USA, 2006. ACM.

[4] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. J. Guibas. Learning program embeddings to propagate feedback on student code. *CoRR*, abs/1505.05969, 2015.