Generative Grading: Neural Approximate Parsing for Automated Student Feedback

Ali Malik[*1], **Mike Wu**[*1], **Vrinda Vasavada**[1], **Jinpeng Song**[1]
John Mitchell[1], **Noah Goodman**[1,2], **Chris Piech**[1]
[1]Department of Computer Science, Stanford University
[2]Department of Psychology, Stanford University
{malikali, wumike, vrindav, jsong5, jcm, ngoodman, piech}@stanford.edu

Abstract

Open access to high-quality education is limited by the difficulty of providing student feedback. In this paper, we present Generative Grading with Neural Approximate Parsing (GG-NAP): a novel approach for providing feedback at scale that is capable of both accurately grading student work while also providing verifiability—a property where the model is able to substantiate its claims with a provable certificate. Our approach uses generative descriptions of student cognition, written as probabilistic programs, to synthesise millions of labelled example solutions to a problem; it then trains inference networks to approximately parse real student solutions according to these generative models. We achieve feedback prediction accuracy comparable to professional human experts in a variety of settings: short-answer questions, programs with graphical output, block-based programming, and short Java programs. In a real classroom, we ran an experiment where humans used GG-NAP to grade, yielding doubled grading accuracy while halving grading time.

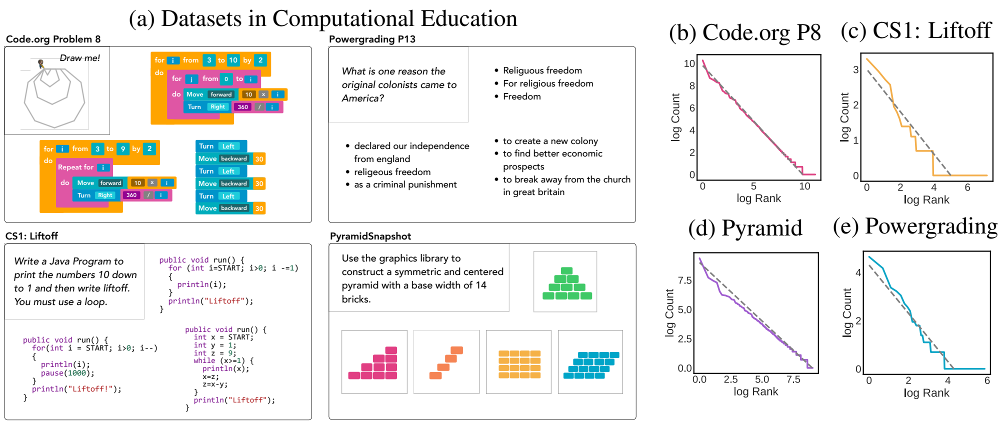
1 Introduction

Computer-assisted education promises open access to world-class instruction and a reduction in the growing cost of learning [2]. A major barrier to this promise of scalable education is the need to automatically *provide feedback* on student work.

Learning to provide feedback has proven to be a hard machine learning problem. Despite dozens of projects that combine massive education data with cutting-edge deep learning in NeurIPS and beyond [16; 1; 28; 23; 15; 10], most approaches fall short. Five issues have emerged: (1) student work is Zipf distributed and as such most incorrect solutions are unique even in large corpora, (2) student work is hard and expensive to label, (3) we want to provide feedback (without historical data) for even the very first student, (4) there is a high human cost to inaccurate predictions, and (5) predictions must be explainable to instructors and students. These challenges are typical of many human-centred AI problems, such as diagnosing rare diseases.

Rather than labelling student solutions, experts are much more adept at thinking "generatively": they can easily imagine the misconceptions a student might have, and construct the space of solutions a student with these misconceptions would produce. Recently, Wu et al. [26] used this intuition to show that a neural network trained on samples from a teacher-written probabilistic context free grammar (PCFG) describing student decisions outperforms a data-hungry supervised neural network and a deep generative model [24]. While groundbreaking, it is difficult for experts to write cognitive models in the form of PCFGs when assignments are complex and open-ended. Further, the inference techniques of [26] do not scale well to very large grammars. The technical contributions of our

[*]Equal contribution.



```
for i from 3 to 10 by 2
do  for j from 0 to i
    do  Move forward 10 x i
        Turn Right 360 / i
```

```
for i from 3 to 9 by 2
do  Repeat for i
    do  Move forward 10 x i
        Turn Right 360 / i
```

```
Turn Left
Move backward 30
Turn Left
Move backward 30
Turn Left
Move backward 30
```

What is one reason the original colonists came to America?

- Religuous freedom
- For religious freedom
- Freedom

- declared our independence from england
- religeous freedom
- as a criminal punishment

- to create a new colony
- to find better economic prospects
- to break away from the church in great britain

Write a Java Program to print the numbers 10 down to 1 and then write liftoff. You must use a loop.

```
public void run() {
  for (int i=START; i>0; i -=1)
  {
    println(i);
  }
  println("Liftoff");
}
```

```
public void run() {
  for(int i = START; i>0; i--)
  {
    println(i);
    pause(1000);
  }
  println("Liftoff!");
}
```

```
public void run() {
  int x = START;
  int y = 1;
  int z = 9;
  while (x>=1) {
    println(x);
    x=z;
    z=x-y;
  }
  println("Liftoff");
}
```

Use the graphics library to construct a symmetric and centered pyramid with a base width of 14 bricks.

Figure 1: (a) We show the prompt and example solutions for 4 problems from programming assignments to history tests. (b)-(e): These datasets all have Zipf-like distributions, as represented by the linear relationship between frequency and rank in log space. This phenomena holds for several modalities: image rendering (Pyramid), natural language (Powergrading), block-based code (Code.org), and Java code (Liftoff).

work are to address these issues by introducing a more flexible generative model class for describing assignment solutions and providing an inference technique able to handle this class.

In this paper we introduce a probabilistic program based grammar; an expressive class that allows for the functional transformations and complex variable dependencies supported by probabilistic programming languages (PPL). However, with added power, this class presents a challenging inference problem. In response, we develop Neural Approximate Parsing (GG-NAP) with two important ideas: (1) to handle complex context-sensitivity, GG-NAP learns to parse as a form of compiled inference, and (2) to handle the long-tailed nature of these generative distributions, GG-NAP is trained via “adaptive" sampling that ensures sufficient resolution in the tails. While we explore it for the education domain, we believe GG-NAP will be useful for many simulation-based modelling applications.

When we apply GG-NAP to open-access datasets we are able to grade student work with close to **expert human-level** fidelity, substantially improving upon the state of the art across a spectrum of public education datasets: introduction to computer programming, short answers to a US citizenship test and graphics-based programs. We show a 50%, 160% and 350% improvement above the state of the art, respectively. When used with human verification in a real classroom, we are able to **double** grading accuracy while reducing by **half** the grading time. Moreover, the grading decisions made by our algorithm are auditable and interpretable by an expert teacher. Our algorithm is “zero-shot" and thus works for the very first student. Since predicted labels correspond to meaningful cognitive states, not merely grades, they can be used in many ways: to give hints to students without teachers, or to help teachers understand their class, etc.

1.1 Datasets

We consider four educational contexts. Fig. 1a shows example solutions for each problem.

Code.org (Block Coding) Wu et al. [26] released a dataset of student responses to 8 exercises from Code.org, involving drawing shapes with nested loops. We take the most difficult problem—drawing polygons with an increasing number of sides—which has 302 human graded responses with 26 labels regarding looping and geometry (e.g. “missing for loop” or “incorrect angle”).

Powergrading (Language) Powergrading [1] contains 700 responses to a US citizenship exam, each graded for correctness by 3 humans. Responses are in natural language, but are typically short (average of 4.2 words). We focus on the most difficult question, as measured by [17]: “name one reason the original colonists came to America". Responses span economics, politics, and religion.

PyramidSnapshot (Graphics) PyramidSnapshot is a university CS1 course assignment intended to be a student’s first exposure to variables, objects, and loops. The task is to build a pyramid using Java’s ACM graphics library. The dataset is composed of images of rendered pyramids from intermediary “snapshots" of student work. Yan et al. [28] annotated 12k unique snapshots with 5 categoies representing “knowledge stages" of understanding.

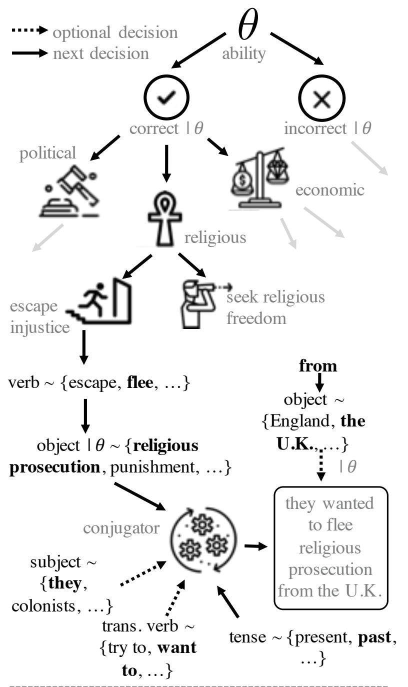
Liftoff (Java) Liftoff is another assignment in a CS1 course. Students write a program that prints a countdown from 10 to 1 followed by the phrase "Liftoff". We use GG-NAP with human verification to grade 176 solutions from a semester of students and measure accuracy and grading time.

1.2 The Grading Task

There are two important machine learning tasks related to grading. First, **auto-predicting feedback**, or labelling a given student solution with meaningful mistakes. Second, **verifiable nearest neighbour**, an alternative when the cost of grading errors is high, in which the algorithm produces a nearest neighbour example whose feedback can be verified with respect to the expert grammar. This system can work with a human-in-the-loop, who focuses on the differences between solutions, to achieve super-human grading precision while reducing grading time.

1.3 Generative Grading

We approach these grading problems by having an expert describe the decisions students make and their resulting answer to an assignment. If we can instantiate these expert priors as a real generative model (e.g. grammar), then we possess a simulator from which we can sample infinite amounts of labelled data, allowing for zero-shot (in terms of real data) learning. While generating solutions to large problems is difficult, representing the prior as a hierarchical set of decisions allows decomposition of this hard task into simpler ones, making it surprisingly easy for experts to express their knowledge. The challenge is then defining a robust enough class of probabilistic models (Sec. 3.2) that can capture the complexities of expert priors (and student behaviour), and constructing the machinery needed to infer student thinking from their solutions (Sec. 3.3). Figure 2 shows a pictorial representation of the generative model we use for the Powergrading task and samples that it produces. Theoretical inspiration for our grammars derives from Brown's "Repair Theory" which argues that the best way to help students is to understand the generative origins of their mistakes [4].



optional decision
next decision
θ
ability
correct | θ
incorrect | θ
political
economic
religious
escape injustice
seek religious freedom
verb ~ {escape, flee, ...}
from
object ~ {England, the U.K., ...}
| θ
object | θ ~ {religious prosecution, punishment, ...}
conjugator
they wanted to flee religious prosecution from the U.K.
subject ~ {they, colonists, ...}
trans. verb ~ {try to, want to, ...}
tense ~ {present, past, ...}

✓ **samples:** (1) their political freedom. (2) the colonists were being politically oppressed.

✗ **samples:** (1) they came to discover gold. (2) i think they were escaping taxes.

Figure 2: A visual representation of a grammar for the Powergrading dataset.

2 Neural Parsing for Inference in Grammars

In this section, we define the class of grammars called *Probabilistic Program Grammars* and describe several motivating properties that make them useful for generative grading.

2.1 Probabilistic Program Grammar

We aim to describe a class of grammars powerful enough to easily encode any instructor's knowledge of the student decision-making process. While it is easy to reason about context free grammars, context independence is a strong restriction that generally limits what instructors can express. As an example, imagine capturing the intuition that students can write a for loop two ways:

```
for (int i = 0; i < 10; i++) { println(10 - i); }  # version 1
for (int n = 10; n > 0; n-=1) { println(n); }      # version 2
```

Clearly, the decision for the for loop header (i < 0; i++), and print statement are dependent on the start index (i = 0) and the choice of variable name (i) as are future decisions like off-by-one. Coordinating these decisions in a context-free grammar requires a great profusion of non-terminals and production rules, which are burdensome for a human to create. Generally, the ability to express arbitrary functional relationships between variables in the grammar is crucial for real world applications. For instance, for capturing method decomposition in programming code or tense and sentence structure in natural language—basic building blocks for a good generative model in education.

We thus introduce a broader class of grammars called Probabilistic Program Grammars (PPGs) that enable us to condition choices on previous decisions and a globally accessible state. A Probabilistic Program Grammar G is more rigorously defined as a subclass of general probabilistic programs, equipped with a tuple (N, Σ, S, D, P) denoting a set of nonterminals, a set of terminals, a start node, a global state, and a set of probabilistic programs, respectively. A production from the grammar is a recursive generation from the start node to a sequence of terminals based on production rules. Unlike PCFGs, a production rule is described by a probabilistic program $\Pi \in P$ so that a given nonterminal can be expanded in different ways based on samples from random variables in Π, the shared state D, and contextual information about other nonterminals rendered in the production. Further, the production rule can also modify the global state D, thus affecting the behaviour of future nonterminals. Lastly, the PPG can transform the final sequence of terminals into an arbitrary space (e.g. from strings to images), to yield the production y. Each derivation is associated with a trajectory $\tau = (x_{i_t}, i_t)_{t=1}^{T}$ of nonterminals[2] encountered during execution. Here, i_t denotes a unique lexical identifier for each random variable encountered in order and x_{i_t} stores the specific value that was sampled. Define the joint distribution (induced by G) over trajectories and productions as $p_G(\tau, y)$. We refer to the procedure of generating a sample $(\tau, y) \sim p_G$ as $\textsc{SampleGrammar}(G)$.

Given such a grammar, we are interested in *parsing*: this is the task of mapping a production y to the most likely trajectory, $\arg\max_\tau p_G(\tau|y)$ that could have produced y. This is a difficult search problem: the number of trajectories grows exponentially even for simple grammars, and common methods for parsing by dynamic programming (Viterbi, CKY) are not applicable in the presence of context-sensitivity and functional transformations. To make this problem tractable, we present deep neural networks to approximate the posterior distribution over trajectories. We call this approach *neural approximate parsing* with generative grading, or GG-NAP.

2.2 Neural Inference Engine

The challenge of doing inference over trajectories is a difficult one. Trajectories can vary in length and contain nonterminals with different support. To approach this with neural nets, we decompose the inference task into a set of easier sub-tasks. The posterior distribution over a trajectory $\tau = (x_{i_t}, i_t)_{t=1}^{T}$ given a yield y can be written as the product of individual posteriors over each nonterminal x_{i_t} using the chain rule:

$$p_G(x_{i_1}, \dots x_{i_T}|y) = p_G(x_{i_T}|y, \mathbf{x}_{<i_T}) p_G(x_{i_{T-1}}|y, \mathbf{x}_{<i_{T-1}}) \cdots p_G(x_{i_1}|y) = \prod_{t=1}^{T} p_G(x_{i_t}|y, \mathbf{x}_{<i_t}), \quad (1)$$

where $\mathbf{x}_{<i_t}$ denotes previous nonterminals $(x_{i_1}, \dots, x_{i_{t-1}})$. Eqn. 1 shows that we can learn each posterior $p(x_i|\mathbf{x}_{<i}, y)$ separately. With an RNN, we efficiently represent the influence of previous nonterminals $\mathbf{x}_{<i}$ autoregressively using a shared hidden representation over T timesteps. To encode the production y, we use standard machinery (e.g. CNNs for images, RNNs for text). To allow for nonterminals with different support, we define three layers for each random variable x_i: (1) an index embedding layer that maps index i to a fixed dimension vector, (2) a value embedding layer that maps the value of x_i to a fixed dimension vector and (3) an inference layer that transforms the RNN hidden state into parameters of the posterior for the next nonterminal x_{i+1}. Thus, the input to the RNN is fixed, being the concatenation of the value embedding, index embedding, and production encoding.

To train the GG-NAP, we optimize the objective,

$$\mathcal{L}(\theta) = \mathbb{E}_{p_G(\tau, y)}[\log p_\theta(\tau|y)] \approx \frac{1}{M} \sum_{m=1}^{M} \log p_\theta(\tau^{(m)}|y^{(m)}) \quad (2)$$

where θ are all trainable parameters and p_θ represents the posterior distribution defined by the inference engine[3]. The second equality is a Monte Carlo estimate using a dataset of samples $\{\tau^{(m)}, y^{(m)}\}$ from G. At test time, given only a production y, GG-NAP recursively samples $x_{i_{t-1}} \sim p_\theta(x_{i_{t-1}}|y = y, \mathbf{x}_{>i_t} = \mathbf{x}_{>i_t})$ for $t = T, ..., 1$ and uses this sample as the input to the next RNN step, like in usual sequence generation models [8].

[2]Note that the length of the trajectory T can vary for different y.

[3]Since we are given $p_G(\tau|y)$, we can parameterise $p_\theta(\tau|y)$ to be from the correct distributional family.

2.3 Relationship to Viterbi Parsing

In [26], the authors released PCFGs for two exercises from Code.org (P1 and P8) that produce code. These grammars are large: P1 has 3k production rules whereas P8 has 263k. Given a PCFG, we compare GG-NAP to Viterbi (CYK) in terms of retrieving the correct trajectory for productions from the grammar. We measure *trajectory accuracy*: the fraction of nodes that are in both parses.

Table 1: Comparison of Inference and Cost between Viterbi and Neural Parsing

PCFG	Trajectory Acc.
Code.org P1 (MAP)	0.943
Code.org P1 (best-of-10)	0.987
Code.org P8 (MAP)	0.917
Code.org P8 (best-of-10)	0.921

PCFG	Parser	# Production Rules	Cost (Sec.)
Code.org P1	Viterbi	3k	0.79 ± 1.2
Code.org P1	NAP	3k	0.17 ± 0.1
Code.org P8	Viterbi	263k	182.8 ± 40.2
Code.org P8	NAP	263k	0.25 ± 0.2

Using 5k samples from each PCFG, we found trajectory accuracies of 94% and 92% for P1 and P8 respectively, meaning that Viterbi and GG-NAP agree in almost all cases. Further, if we draw multiple samples from the GG-NAP posterior and take the best one, we find improvements of up to 4%. In exchange for being approximate, GG-NAP is not restricted to PCFGs, can invert transformations on productions, and is orders of magnitude faster than Viterbi (0.3 vs 183 sec).

2.4 Verifiable Nearest Neighbour Retrieval

Given a production y from a grammar G, the GG-NAP algorithm can provide a verifiable certificate for its predicted parsing. Let $\widehat{\tau} = \{\widehat{x}_i\}$ refer to the inferred trajectory for y and $\tau = \{x_i\}$ refer to the true (unknown) trajectory. If we repeatedly call $\textsc{SampleGrammar}(G)$ while fixing the values for each encountered random variable to $\widehat{x}_i$, then we should be able to generate the exact production y, showing with certainty that $\widehat{\tau} = \tau$. In practice, very few samples are needed to recover y. On the other hand, if an observation y is not in the grammar G (like some real student programs), τ is not well-defined and the inferred trajectory $\widehat{\tau}$ will be incorrect. However, $\widehat{\tau}$ will be still specify a production $\widehat{y}$ that we can interpret as an approximate nearest neighbour to y in G. Intuitively, we expect $\widehat{y}$ and y to be "similar" semantically as specified by the nonterminals in G. In practice, we can measure a domain-specific distance between $\widehat{y}$ and y e.g. token edit distance for text.

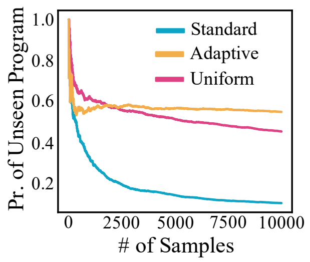
2.5 k-Nearest Neighbour Baseline

We present a strong baseline that is also capable of performing verifiable approximate parsing. This algorithm is simply a k-nearest neighbour classifier: we generate and store a dataset $\{\tau^{(m)}, y^{(m)}\} \sim p_G$ with hundreds of thousands of unique productions as well as their associated trajectories. At test time, given an input to parse, we can find its nearest neighbour in the stored samples and return its associated trajectory. If the neighbour is an exact match, the prediction is verifiable. We refer to this baseline as GG-kNN. Depending on the grammar, y will be in a different output space (images, code, text) and thus the distance metric used for GG-kNN will be domain dependent.

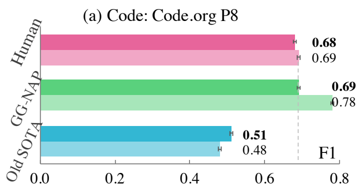
2.6 Adaptive Sampling

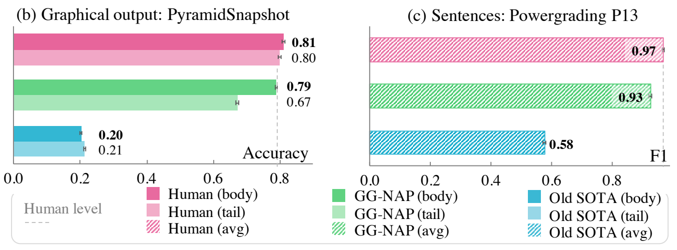
As both GG-kNN and GG-NAP require a dataset of samples for training, we must be able to generate unique productions from a grammar efficiently. For GG-kNN specifically, the number of unique productions strictly defines the quality of the model. However, due to the nature of Zipfs, generating unique data points can be expensive due to over-sampling of the most common productions. Furthermore, a second concern is that we do not want to completely ignore the prior distributions defined by the grammar. Otherwise we would sample very unlikely (albeit unique) productions that do not describe student behaviour.

Figure 3: Good-Turing Estimates



To balance competing interests, we present a novel method called Adaptive Grammar Sampling that downweights the probabilities of decisions proportional to how many times they lead to duplicate productions. This algorithm has many useful properties and is based on Monte-Carlo Tree Search





Model	Body F1	Tail F1
Output CNN [26]	0.10	0.10
Program RNN [23]	0.27	0.22
MVAE [24]	0.38	0.26
Rubric Sampling [26]	0.51	0.48
GG-kNN	0.31	0.33
GG-NAP	**0.69**	**0.78**
Human	0.68	0.69

Model	Body Acc	Tail Acc
kNN [28]	0.20	0.12
NeuralNet [28]	0.20	0.21
GG-kNN	timeout	timeout
GG-NAP	**0.79**	**0.67**
Human	0.81	0.80

Model	Avg F1	Tail F1
Handcrafted [6]	0.58	-
T&N Best [17]	0.55	-
GG-kNN	0.78	0.63
GG-NAP	**0.93**	**0.76**
Human	0.97	0.90

Figure 4: Summary of results for three datasets. GG-NAP outperforms the old state of the art (SOTA).

and the Wang-Landau algorithm from statistical physics. We consider this an interesting corollary and refer the reader to the supplement. Fig. 7 shows an example of how much more efficient this algorithm is compared to simply sampling naively from the Liftoff grammar by plotting the Good-Turing estimates (probability of encountering an unseen program) over the number of samples made so far. In practice, adaptive sampling has a parameter that can be toggled to control how fast we explore the Zipf, allowing us to preserve likely productions from the head and body.

3 Results

For the task of inferring student understanding, we find that GG-NAP beats the previous state-of-the-art (SOTA) by a significant margin *in all four educational domains*. Further, it approaches (or surpasses) human performance (see Fig. 4). Below, we first describe GG-NAP's performance on labelled datasets as compared to previous work followed by its performance when used for grading student code in a real classroom.

To evaluate our models, we separately calculate performance for different regions of the Zipf: we define the *head* as the most popular solutions, the *tail* as solutions that appear only once or twice, and the *body* as the rest. As solutions in the head can be memorised, we focus on the body and tail.

3.1 Autonomous Grading

In each domain, we are given a dataset of student programs and labelled feedback. By design, we include each of the labels as a nonterminal in the grammar, thereby reducing prediction to parsing.

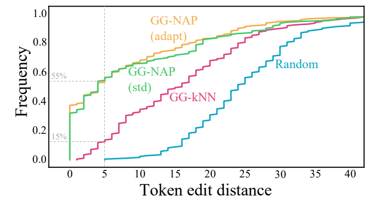
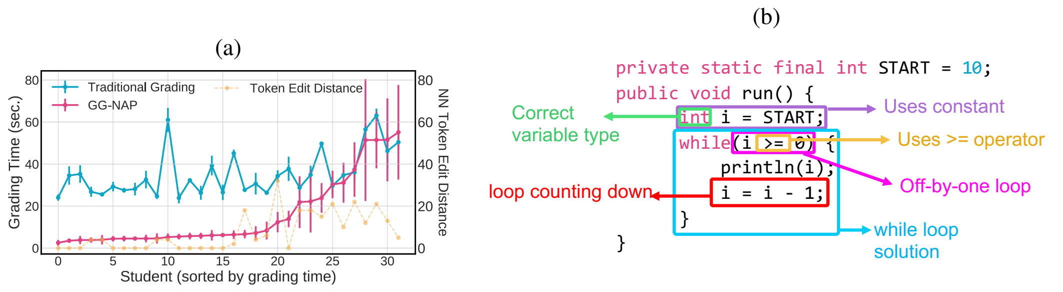


Figure 5: Cumulative distribution of token edit distance between student programs and nearest-neighbours produced by various strategies. GG-NAP has 30% exact matches and 55% in 5 token edits. GG-kNN only captures 15%.

Code.org GG-NAP sets the new SOTA on the dataset, beating [26] in both the body and tail, and surpassing human performance (historically measured as F1). There is a rich history of previous work involving supervised classifiers [26; 23] that struggled with the tiny amount of labelled data, resulting in poor performance. Even some zero-shot approaches like [26], which trains an RNN on synthetically labelled samples from an expert-designed PCFG, are significantly below human quality. The potential impact of a human-level autonomous grader is large: Code.org is used by 610 million students worldwide, and has *un*successfully launched initiatives in the past to crowd-source feedback for student solutions. Instead of thousands of human hours of teacher work, GG-NAP could provide the same quality of feedback at scale.

Powergrading For this open dataset of short answer responses, GG-NAP outperforms the previous SOTA with an F1 score of 0.93, an increase of 0.35 points. We close the gap to human performance, measured to be F1 = 0.97 (which generously considers the majority of the three raters to be the gold label). Earlier work either used hand-crafted features for natural language [6] or the latest supervised



```
private static final int START = 10;
public void run() {
    int i = START;
    while(i >= 0) {
        println(i);
        i = i - 1;
    }
}
```

Figure 6: (a) Plot of average time taken to grade 30 student solutions to Liftoff. GG-NAP convincingly reduces grading time for 26/30 solutions. The amount of time saved correlates with the token edit distance (yellow). (b) GG-NAP allows for automatically associating student work with fine-grained automated feedback.

neural network architecture [17]. With 700 labelled data points, these methods heavily overfit to the training set. Further, since the Powergrading task is unique in that it contains natural language, the PPG we designed had to explain variations both in writing style and in semantic understanding. The strong performance of GG-NAP suggests that even beyond education, the idea of representing expert priors as expressive simulators can be generalised to many domains.

PyramidSnapshot As in the last two cases, GG-NAP is the new SOTA, out-performing baselines (kNN between images and a VGG classifier) from [28] that are trained on 200 labelled images by about a 50% gain in accuracy. Unlike other datasets, PyramidSnapshot includes student's intermediary work, showing stages of progression through multiple attempts at solving the problem. With our near-human level performance, instructors could use GG-NAP to measure student cognitive understanding over time *as* students work. This builds in a real-time feedback loop between the student and teacher that enables a quick and accurate way of assessing teaching quality and characterising both individual and classroom learning progress. From a technical perspective, since PyramidSnapshot only includes rendered images (and not student code), GG-NAP was responsible for parsing student understanding from unstructured images, a feat not possible with simpler grammars like PCFGs.

3.2 Human Guided Grading

While good performance on benchmark datasets is promising, a true test of an algorithm is its effectiveness in the real world. For GG-NAP, we would like to gauge its impact on grading accuracy and speed in a real classroom setting. We hired a cohort of expert graders (teaching assistants from a large private university with similar experience) to each grade 30 real student solutions to Liftoff, a university course assignment. For each student solution, we also retrieve the auto-graded nearest neighbour using GG-NAP. (As an aside, GG-NAP excels at finding semantically relevant neighbours compared to baseline methods. Fig. 5 compares the token edit distance between the student program and the nearest neighbours retrieved by GG-NAP versus GG-kNN, finding significantly better matches with the former.) For control, half the graders proceed normally, assigning a set of feedback labels measuring understanding of looping concepts by analysing student solutions. The other half of graders additionally have access to (1) the feedback assigned to the nearest neighbour by GG-NAP and (2) a code diff between the student program and the nearest neighbour. Some example feedback labels include “off by one increment", “uses while loop", or “confused > with <". All grading is done on a web application that keeps track of the time taken for the grader to grade a problem.

We found that the average time (to grade 30 problems) for graders with GG-NAP is **507 sec**. Without GG-NAP, the average time is **1130 sec**, a more than double increase. With GG-NAP, **3 grading errors** were made with respect to gold-standard feedback given by the course Professor. Without GG-NAP, **8 errors** were made. By halving both the number of errors and the amount of time, GG-NAP can have a large impact in classrooms today, saving instructors and teaching assistants unnecessary hours and worry over grading assignments.

4 Related Work

“Rubric sampling” [26] first introduced the concept of encoding expert priors in grammars of student decisions, and was the inspiration for our work. The authors design PCFGs to curate synthetically labelled datasets to train a supervised classifier. Our approach builds on this, but GG-NAP operates on a more expressive family of grammars that are context sensitive. Due to this complexity, new innovations were required to effectively do inference. From Code.org, we see that expressivity is responsible for pushing GG-NAP past human level performance. Further, our paradigm adds an important notion of verifiability lacking in previous work. Rubric sampling as previously presented suffers from the black-box nature of neural networks.

Inference over grammar trajectories is similar to “compiled inference" for execution traces in probabilistic programs. As such, our inference engine shares similarities to PPL literature [14]. By limiting ourselves to a class of grammars, we get a nice interpretation of compiled inference as a parsing algorithm. Further, we show the promise of compiled inference in much larger probabilistic programs (with skewed prior distributions). Previous work [14; 27; 13] usually involve 4 or 5 random variables whereas our grammars grow to hundreds.

The design of PPGs also draws on many influences from natural language processing. For starters, our neural inference engine can be viewed as an encoder (or “inference network") in a RNN-based variational autoencoder [3] that specifies a posterior distribution over many categorical variables. Further, the index embedding layer serves as a unique identifier similar to the positional encoding in transformers [21]. Finally, the verifiable properties of GG-NAP have strong ties to explainable AI [19; 9; 12], especially in the healthcare domain [25; 18] where interpretability is paramount.

5 Discussion

Highlighting feedback in student solutions Rather than predicting feedback labels, it would be even more useful to provide “dense" feedback that highlights the section of the code or text responsible for the student misunderstanding. To achieve this, we use GG-NAP to infer a trajectory, $\tau = \{x_i\}$ for a given production y. For every nonterminal x_i, we want to measure its impact on y. If for each x_i we have an associated production rule with an intermediate output β, then highlighting amounts to finding the part of y which β was responsible for. Fig. 6 shows a random program with automated, segment-specific feedback given by GG-NAP. This level of explainability is sorely needed in both online education and AI and could revolutionise how students are given feedback at scale.

Automatically improving grammars Building PPGs is an iterative process, requiring time for improvements in design. A user wishing to improve a PPG would like a sense of where their grammar is lacking. Fortunately, given a set of difficult examples where GG-NAP does poorly, we can deduce the set of nodes in the PPG that consistently led to mistakes. To illustrate this, we took the Liftoff PPG which crucially contains a node that decides between incrementing up or down in a “for" loop, and removed the option of incrementing down. If we train GG-NAP on the smaller PPG, we will fail to parse examples that “increment down". In this case, the set of nodes that consistently led to mistakes all related to incrementation. At this time, an expert can quickly diagnose the issue.

Need for probabilistic program grammars In practice, we have experienced the benefits of having a grammar which allows for the full expressivity of a computer program. One non-obvious benefit of having state is the ability to break independence assumptions between mistakes. If a PCFG describes N different places where a student could err, as N tends towards infinity it will be increasingly improbable to produce a sample with only one mistake, which we know to be a very common case among students. A PPG allows for a natural way to have a continuous ability for students which can model the binomial phenomena of either making many mistakes or only a few. Adding ability as a state alone increased the F1 scores for Code.org by 0.1 points.

Not only experts can write good grammars. Writing a good grammar does not require immense experience. For instance, the PyramidSnapshot grammar that sets the new SOTA was written by a first-year undergraduate. Further, grammars are re-usable: similar assignments will share nonterminals and some invariances (e.g. all the ways of writing i++ are the same everywhere).

6 Conclusion

In this paper we make novel contributions to the task of providing automated student feedback that beats numerous state-of-the-art approaches and shows significant impact when used in practice. The ability to finely predict student decisions opens up many doors in education. This work could be used to automate feedback, visualise student approaches for instructors, and make grading easier, faster, and more consistent. Although more work needs to be done on making powerful grammars easier to write, we believe this is an exciting direction for the future of education and a huge step in the quest for combining machine learning and human-centred artificial intelligence.

References

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

[2] W. G. Bowen. The ‘cost disease’in higher education: is technology the answer? *The Tanner Lectures Stanford University*, 2012.

[3] S. R. Bowman, L. Vilnis, O. Vinyals, A. M. Dai, R. Jozefowicz, and S. Bengio. Generating sentences from a continuous space. *arXiv preprint arXiv:1511.06349*, 2015.

[4] J. S. Brown and K. VanLehn. Repair theory: A generative theory of bugs in procedural skills. *Cognitive science*, 4(4):379–426, 1980.

[5] H. S. Chang, M. C. Fu, J. Hu, and S. I. Marcus. An adaptive sampling algorithm for solving markov decision processes. *Operations Research*, 53(1):126–139, 2005. doi: 10.1287/opre.1040.0145.

[6] J. Daxenberger, O. Ferschke, I. Gurevych, and T. Zesch. Dkpro tc: A java-based framework for supervised learning experiments on textual data. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 61–66, 2014.

[7] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[8] A. Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013. URL `http://arxiv.org/abs/1308.0850`.

[9] B. Hancock, P. Varma, S. Wang, M. Bringmann, P. Liang, and C. Ré. Training classifiers with natural language explanations. *arXiv preprint arXiv:1805.03818*, 2018.

[10] Q. Hu and H. Rangwala. Reliable deep grade prediction with uncertainty estimation. *arXiv preprint arXiv:1902.10213*, 2019.

[11] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[12] P. W. Koh and P. Liang. Understanding black-box predictions via influence functions. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1885–1894. JMLR. org, 2017.

[13] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

[14] T. A. Le, A. G. Baydin, and F. Wood. Inference compilation and universal probabilistic programming. *arXiv preprint arXiv:1610.09900*, 2016.

[15] J. Liu, Y. Xu, and L. Zhao. Automated essay scoring based on two-stage learning. *arXiv preprint arXiv:1901.07744*, 2019.

[16] C. Piech, J. Bassen, J. Huang, S. Ganguli, M. Sahami, L. J. Guibas, and J. Sohl-Dickstein. Deep knowledge tracing. In *Advances in neural information processing systems*, pages 505–513, 2015.

[17] B. Riordan, A. Horbach, A. Cahill, T. Zesch, and C. M. Lee. Investigating neural architectures for short answer scoring. In *Proceedings of the 12th Workshop on Innovative Use of NLP for Building Educational Applications*, pages 159–168, 2017.

[18] A. S. Ross and F. Doshi-Velez. Improving the adversarial robustness and interpretability of deep neural networks by regularizing their input gradients. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

[19] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 618–626, 2017.

[20] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[22] F. Wang and D. Landau. Efficient, multiple-range random walk algorithm to calculate the density of states. *Physical review letters*, 86:2050–3, 04 2001. doi: 10.1103/PhysRevLett.86.2050.

[23] L. Wang, A. Sy, L. Liu, and C. Piech. Learning to represent student knowledge on programming exercises using deep learning. In *EDM*, 2017.

[24] M. Wu and N. Goodman. Multimodal generative models for scalable weakly-supervised learning. In *Advances in Neural Information Processing Systems*, pages 5575–5585, 2018.

[25] M. Wu, M. C. Hughes, S. Parbhoo, M. Zazzi, V. Roth, and F. Doshi-Velez. Beyond sparsity: Tree regularization of deep models for interpretability. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[26] M. Wu, M. Mosse, N. Goodman, and C. Piech. Zero shot learning for code education: Rubric sampling with deep learning inference. *arXiv preprint arXiv:1809.01357*, 2018.

[27] Y. Wu, L. Li, S. Russell, and R. Bodik. Swift: Compiled inference for probabilistic programming languages. *arXiv preprint arXiv:1606.09242*, 2016.

[28] L. Yan, N. McKeown, and C. Piech. The pyramidsnapshot challenge: Understanding student process from visual output of programs. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, pages 119–125, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-5890-3. doi: 10.1145/3287324.3287386. URL `http://doi.acm.org/10.1145/3287324.3287386`.

A Model Hyperparameters

For reproducibility, we include all hyperparameters used in training GG-NAP. Unless otherwise stated, we use a batch size of 64, train for 10 or 20 epochs on 100k samples from a PPG. The default learning rate is 5e-4 with a weight decay of 1e-7. We use Adam [11] for optimization. If the encoder network is an RNN, we use the Elman network with 4 layers, a hidden size of 256, and a probability of dropping out hidden units of 1%. If the encoder network is a CNN, we train VGG-11 [20] with Xavier initialization [7] from scratch. For training VGG, we found it important to lower the learning rate to 1e-5. The neural inference engine itself is an unrolled RNN: we use a gated recurrent unit with a hidden dimension of 256 and no dropout. The value and index embedding layers output a vector of dimension 32. These hyperparameters were chosen using grid search.

B Adaptive Grammar Sampling

In the text, we introduced a nearest neighbour baseline (KNN) for verifiable parsing. The success of KNN is highly dependent on storing a set of unique samples. With Zipfs, i.i.d. sampling often over-samples from the head of the distribution, resulting in a low count of unique samples and poor performance. To build a strong baseline, we must sample uniques more efficiently.

Algorithm 1 Adaptive Sampling

Input: Probabilistic program grammar $G = (N, \Sigma, R, S, \mathcal{X}, F)$, decay factor d, reward r, and desired size of dataset M.
Output: Dataset of M *unique* samples from the grammar: $D_G = \{(\tau^{(m)}, y^{(m)})\}_{m=1}^{M}$.

```
1:  procedure AdaptiveSample($G$, $d$, $r$, $M$)
2:      $D_G \leftarrow \{\}$
3:      while $|D_G| < M$ do
4:          $\tau, y \leftarrow$ SampleGrammar($G$)
5:          if $(\tau, y) \notin D_G$ then
6:              $D_G \leftarrow D_G \cup \{(\tau, y)\}$
7:          for $i \leftarrow 0$ to $|\tau|$ do
8:              $x_i \leftarrow \tau[i]$        ▷ get $i$-th node in trajectory, $\tau = \{x_i\}_{i=1}^{T}$, of length $T$
9:              $p(x_i|\mathbf{x}_{<i}) \leftarrow \frac{p(x_i|\mathbf{x}_{<i})}{r + d^{|\tau|-i} \cdot p(x_i|\mathbf{x}_{<i})}$
10:             $p(x_i|\mathbf{x}_{<i}) \leftarrow$ Normalise($p(x_i|\mathbf{x}_{<i})$))
```

Further, training the neural inference engine requires sampling a dataset D_G from a PPG G. These samples need to cover enough of the grammar to allow the model to learn meaningful representations and, moreover, they again need to be *unique*. The uniqueness requirement is paramount for Zipfs since otherwise models would be overwhelmed by the most probable samples.

Naively, we can i.i.d. sample a set of M *unique* observations and use it train NAP. However, again, due to the Zipfian nature, generating M unique data points can be expensive as M gets large due to having to discard duplicates. To sample efficiently, a simple idea is to pick each decision uniformly (we call this *uniform sampling*). Although this will generate uniques more often, it has two major issues: (1) it disregards the priors, resulting in very unlikely productions, and (2) it might not be effective as multiple paths can lead to the same production.

Ideally, we would sample in a manner such that we cover all the most likely programs and then smoothly transition into sampling increasingly unlikely programs. This would generate uniques efficiently while also retaining samples that are relatively likely. To address these desiderata, we propose a method called Adaptive Grammar Sampling (Alg. 1) that downweights the probabilities of decisions proportional to how many times they lead to duplicate productions. We avoid overly punishing nodes early in the decision trace by discounting the downweighting by a decay factor d. This method is inspired by Monte-Carlo Tree Search [5] and shares similarities with Wang-Landau from statistical physics [22].

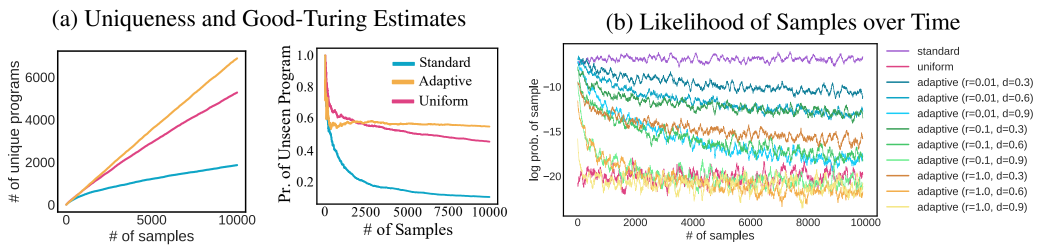


Figure 7: Effectiveness of sampling strategies for Liftoff. Left/Middle: Number of unique programs generated (left) and Good-Turing estimate (middle) as a function of total samples. Right: Likelihood of generated samples over time for various sampling strategies. In particular, we note the effect of reward r and decay d on the exploration rate. The ideal sampling strategy for Zipfs first samples from the head, then body, and finally the tail.

B.1 Properties of Adaptive Sampling

In the main text, we expressed the belief that adaptive grammar sampling increases the likelihood of generating unique samples. To test this hypothesis, we sampled 10k (non-unique) Java programs using the Liftoff PPG and track the number of uniques over time. Fig. 7a shows that adaptive sampling has linear growth in number of unique programs compared to sublinear growth with i.i.d. or uniform sampling. Fig. 7b compute the Good-Turing estimate, a measure for the probability of the next sample being unique, and found adaptive sampling to "converge" to a constant while other sampling methods approach zero. Interestingly, adaptive sampling is customisable. Fig. 7c show the log probability of the sampled trajectories over time. With higher reward r or a smaller decay rate d, adaptive sampling will sample less from the head/body of the Zipf. In contexts where we care about the rate of sample exploration, adaptive sampling provides a tune-able algorithm to search a distribution.

C Grammar Descriptions

We provide an overview of the grammars for each domain, covering the important choices.

Code.org P8 This PPG contains 52 decisions. The primary innovation in this grammar decision is the use of a global random variable that represents the ability of the student. In this turn will affect the distributions over values for nonterminals later in the trajectory such as deciding the loop structure and body. The intuition this captures is that high ability students make very few to no mistakes whereas low ability students tend to make many correlated misunderstandings (e.g. looping and recursion).

CS1: Liftoff This PPG contains 26 decisions. It first determines whether to use a loop, and, if so, chooses between "for" and "while" loop structures. It then formulates the loop syntax, choosing a condition statement and whether to count up or count down. Finally, it chooses the syntax of the print statements. Notably, each choice is dependent on previous ones. For example, choosing an end value in a for loop is sensibly conditioned on a chosen start value.

Powergrading: Short Answer This PPG contains 53 nodes. Unlike code, grammars over natural language need to explain variance in both semantic meaning and prose. This is not as difficult for short sentences. In designing the grammar, we inspect the first 100 responses to gauge student thinking. Procedurally, the grammar's first decision is choosing whether the production will be correct or incorrect. It then chooses a subject, verb, and noun. These three choices are dependent on the correctness. Correct answers lead to topics like religion, politics, and economics while incorrect answers are about taxation, exploration, or physical goods. Finally, the grammar chooses a writing style to craft a sentence. To capture variations in tense, we use a conjugator[4] as a functional transformation F on the output.

PyramidSnapshot The grammar contains 121 nodes, the first of which decides between 13 "strategies" (e.g. making a parallelogram, right triangle, a brick wall, etc.). Each of the 13 options leads its

[4]Python's mlconjug library: `https://pypi.org/project/mlconjug`.

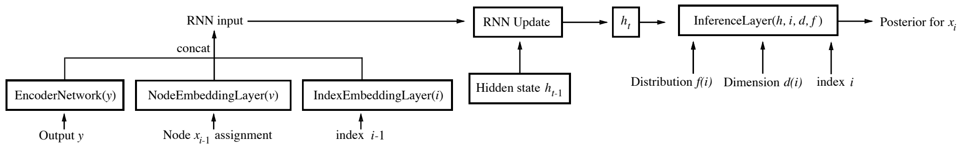


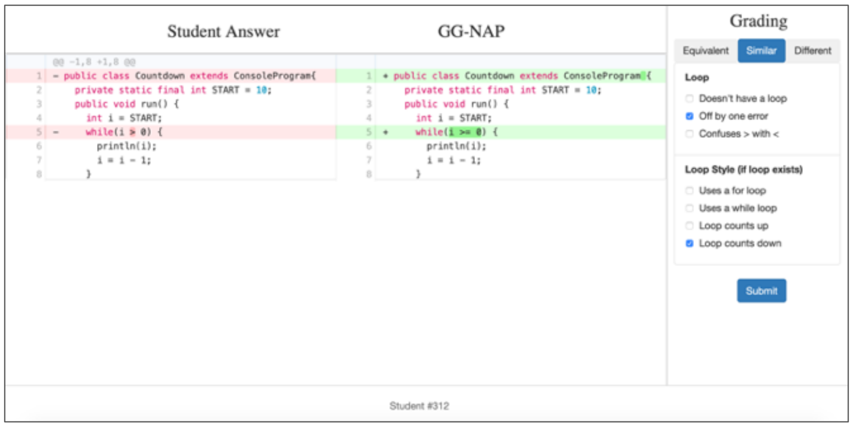
Figure 8: Architecture of the neural inference engine. We show a single RNN update to parameterize $p(x_i|\mathbf{x}_{<i}, y)$. This procedure is repeated for each T, the length of the trajectory.

own set of nodes that are responsible for deciding shape, location, and colour. Finally, the trajectory of decisions is used to render an image. The first version of the grammar was created by peaking at 200 images. A second version was updated by viewing 50 more.

D NAP Architecture

Fig. 8 visualizes the architecture for the neural inference engine in NAP. Critically, NODEEMBEDDINGLAYER, INDEXEMBEDDINGLAYER, and InferenceLayer are specific to each nonterminal x_i to support arbitrary dimensionality and distributions for random variables. The ENCODERNETWORK is responsible for transforming unstructured images and text to vector space.

E Grading UI



```
@@ -1,8 +1,8 @@
- public class Countdown extends ConsoleProgram{
    private static final int START = 10;
    public void run() {
      int i = START;
-     while(i > 0) {
        println(i);
        i = i - 1;
      }
```

```
+ public class Countdown extends ConsoleProgram {
    private static final int START = 10;
    public void run() {
      int i = START;
+     while(i >= 0) {
        println(i);
        i = i - 1;
      }
```

Loop

- [ ] Doesn't have a loop
- [✓] Off by one error
- [ ] Confuses > with <

Loop Style (if loop exists)

- [ ] Uses a for loop
- [ ] Uses a while loop
- [ ] Loop counts up
- [✓] Loop counts down

Figure 9: Grading UI based on GG-NAP

We show an image of the user-interface used in the field experiment. This is the view a grader (with access to NAP) would see. The real student response is give on the left and the nearest neighbour given by GG-NAP on the right. A differential between the two images is provided, inspired by Github design. On the very right is a set of labels that the grader is responsible for assigning values to.

F Improving the Grammar

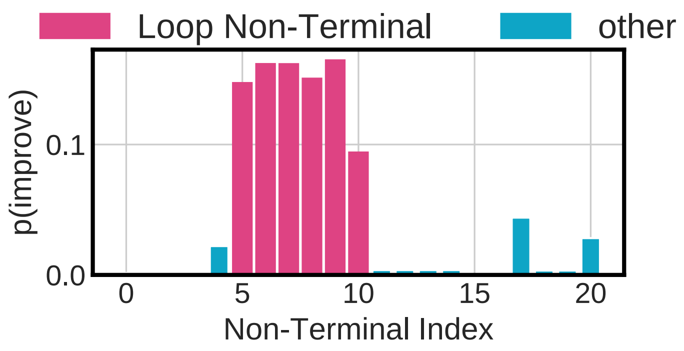


Figure 10: Given a Liftoff grammar that can only increment up from 1 to 10 (e.g. i++), if we attempt inference on an unseen program that increments down from 10 to 1 (e.g. i–), we can track at which nonterminals inference fails, and use that to estimate where we need to add additional nodes, thereby helping the user improve the grammar. The height of each bar represents the likelihood that improvements are needed for that nonterminal, the highest of which are all related to looping.

In the discussion of the main text, we introduced an experiment to test if we could detect nodes at which we were failing to parse out-of-distribution examples: we took the Liftoff PPG (which crucially contains a node that decides between incrementing up or down in a “for" loop), and removed the option of incrementing down. If we train GG-NAP on the smaller PPG, we will fail to parse examples that “increment down". In this case, the set of nodes that consistently led to mistakes all related to incrementation. Fig. 10 shows the distribution over which nodes GG-NAP believes to be responsible for the failed parse. The top 6 nonterminals that GG-MAP picked out related to looping and incrementation. As an expert, this is enough of a diagnosis to improve the grammar.

G Grammar Sample Zoo: Powergrading

they left to pursue freedom of religion
i learned, penal colony
the colonists were spreading religion?
maybe they flee from religious oppression.
as penal colony.
freedom to practice religion
religion.
political persecution from their king and queen.
i learned, to explore the us
the colonists escaped taxation
farmers.
the colonists practiced freedom?
politically persecuted.
i think the original colonists left to explore the us.
the original colonists left to pursue religion.
economic opportunity
the colonists had wanted to flee from their political persecution from their king and queen
the colonists fled political oppression.
i learned, they wanted to gain freedom of religion in america?
land.
tobacco?
puritans.
i learned, the english wanted to avoid their religious tyranny
their political beliefs in the colonies.
a colonist?
the original colonists were spreading their religion
the colonists came to settle the land
for tobacco plantations
i think the original colonists had wanted to pursue their political freedom?
political beliefs
i think more freedom of religion.
to worship their religion.
they came to discover .
i think they were escaping taxes
i think political prosectuion from england.
the colonists were obtaining freedom of religion
the colonists escape taxes?
the original colonists left to travel america.
i learned, tobacco
for land
politically oppressed.
they had got away from england?
maybe the pilgrims came to worship freely?
i think they had left
the original colonists avoid their religious persecution
they had escaped taxes from britain.
they had wanted to find religious freedom.
i learned, the colonists escaped taxation?
as penal colony?
the pilgrims had wanted to worship their religion.
maybe plantations?
i think penal colony.
the colonists had obtained better economic opportunity in the colonies.
colonists
the pilgrims were searching for economic prospects
i learned, for land.
the original colonists were escaping taxation.
the english came to spread their religion
i think to flee their political prosectuion from the uk?
i think the original colonists came to seek their beliefs
to flee political prosectuion from great britain
the colonists escaped taxes.
maybe gold
criminals?
the colonists came to travel the us.
the colonists left to escape taxes.
i think they wanted to tour
the original colonists had wanted to leave ?
i learned, plantations
i learned, politically persecuted?
the pilgrims left to escape their political persecution from their king and queen?
i think the colonists were gaining liberty.
their religious freedom in america
i learned, taxation from the uk
i think the pilgrims were being criminals
they wanted to discover
i think the pilgrims had wanted to break away from the church
money
the original colonists fled tyranny from britain?
maybe oppression.
maybe their freedom in the us.
maybe the colonists left to flee their oppression from the uk?
gold.
a colonist.
the original colonists wanted to escape taxation?
i think the colonists get away ?
the colonists wanted to flee their religious prosectuion
economic prospects in america?
their political freedom?
i think liberty in the us?
to flee their religious tyranny?
the pilgrims gain political beliefs
gold
the pilgrims had wanted to settle the land.
i learned, they wanted to explore the colonies
i think to worship their religion.
economic possibilities in america?
the colonists left to search for economic prospects
the pilgrims were religiously persecuted?
their tyranny from the uk.
i learned, their religious oppression.
i learned, the colonists were being politically persecuted
i learned, the original colonists had fled political oppression.
i think land?
the pilgrims had fled religious oppression from england.
tobacco plantations.
maybe the colonists came to spread religion.
maybe they wanted to get away ?
i learned, the english had explored ?
maybe freedom.
a penal colony?
their religious tyranny.
i learned, promised cheap land?
the english came to get away from the british
freedom in america
better economic prospects
i learned, oppression from the uk.
i learned, to worship their religion.
i learned, tyranny from the british?
maybe the colonists left from england
plantations?

H Grammar Sample Zoo: Code.org

```
For(1, 100, 10){
  MoveForward(x)
  TurnLeft(360 / x)
}
```

```
MoveForward(30)
For(3, 4, 3){
  Repeat(x){
    TurnRight(360 / x)
    MoveForward(x * 10)
  }
}
```

```
For(3, 6, 1){
  Repeat(x){
    MoveForward(x * 10)
    TurnRight(360 / x)
  }
}
```

```
For(30, 120, 30){
  Repeat(3){
    MoveForward(70)
    TurnRight(360 / x)
  }
}
```

```
For(3, 10, 2){
  Repeat(x){
    MoveForward(x * 10)
    TurnRight(360 / x)
  }
}
```

```
For(3, 9, 2){
  Repeat(x){
    MoveForward(360 / x)
    TurnRight(360 / x)
  }
}
```

```
Repeat(9){
  MoveForward(90)
  TurnRight(40)
}
Repeat(6){
  MoveForward(68)
  TurnRight(55)
}
```

```
For(1, 11, 2){
  Repeat(x){
    MoveForward(100)
  }
}
```

```
For(3, 9, 3){
  Repeat(3){
    TurnRight(120)
  }
}
```

```
Repeat(9){
  MoveForward(90)
  TurnRight(40)
}
MoveForward(99)
TurnRight(336)
Repeat(6){
  MoveForward(70)
  TurnRight(51)
}
```

```
MoveForward(100)
```

```
For(3, 9, 2){
  Repeat(x){
    MoveForward(70)
    TurnRight(x)
  }
}
```

```
For(1, 100, 10){
  Repeat(9){
    MoveForward(30)
    TurnRight(72)
    MoveForward(75 + x)
  }
}
```

```
MoveForward(70)
```

```
MoveForward(90)
TurnRight(20)
MoveForward(90)
```

```
For(10, 400, 23){
  Repeat(x){
    MoveForward(100)
  }
}
```

```
For(100, 200, 100){
  Repeat(x){
    MoveForward(x * 10)
    TurnLeft(90)
  }
}

)
```

```
For(25, 100, 20){
  Repeat(9){
    MoveForward(90)
    TurnRight(40)
    MoveForward(70)
    TurnRight(51.5)
    MoveForward(50)
    TurnRight(72)
    MoveForward(30)
    TurnRight(120)
    MoveForward(30)
    TurnRight(120)
    MoveForward(30)
  }
  MoveForward(75)
  TurnRight(60)
}
```

```
For(3, 9, 2){
  Repeat(x){
    MoveForward(x * 10)
  }
}
```

```
For(1, 15, 3){
  Repeat(x){
    MoveForward(x * 10)
    MoveForward(10 * x)
    TurnRight(360/5)
  }
}
```

```
Repeat(x * 0){
  MoveForward(x * 4)
  TurnRight(90)
  MoveForward(x * 10)
}
TurnRight(x)
```

```
For(4, 7, 3){
  Repeat(x * 0){
    TurnRight(x / 360)
  }
}
```

```
For(5, 2, 3){
  Repeat(x){
    MoveForward(75 + x)
    TurnRight(40)
  }
}
```

```
For(25, 100, 20){
  MoveForward(100)
  Repeat(x){
    MoveForward(60)
  }
}
```

```
For(3, 5, 2){
  MoveForward(x * 10)
  TurnRight(360 / x)
  MoveForward(x * 10)
}
```

```
MoveForward(10)
```

```
For(3, 9, 2){
  Repeat(x){
    MoveForward(30)
    TurnRight(90)
  }
}
```

```
MoveForward(90)
```

```
MoveForward(49)
```

```
For(3, 9, 1){
  Repeat(x){
    MoveForward(10 * x)
    TurnRight(360 / x)
  }
}
```

```
Repeat(9){
  MoveForward(90)
  TurnRight(40)
}
Repeat(6){
  MoveForward(70)
  TurnRight(51.32)
}
```

```
For(3, 6, 10){
  Repeat(3){
    MoveForward(x + 75)
    TurnRight(360 / x)
  }
}
```

```
Repeat(3) {
  MoveForward(90)
  TurnRight(120)
}
```

```
MoveForward(90)
TurnRight(40)
MoveForward(90)
TurnRight(40)
MoveForward(100)
```

I Grammar Sample Zoo: Liftoff

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    for(int i = START; i > 0; i--) {
      println(i);
    }
    println("Liftoff");
  }
}
```

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    for(int i = START; i >= 1; i = i - 1) {
      println(i);
    }
    println("Liftoff");
  }
}
```

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    for(int i = START; i > 0; i -= 1) {
      println(i);
    }
    println("LIFTOFF");
  }
}
```

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    for(int i = 10; i > 0; i -= 1) {
      println(i);
    }
  }
}
```

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    println("Liftoff!");
  }
}
```

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    for(int i = START; i > 0; i--) {
      println(i);
    }
    println("Lift off");
  }
}
```

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    double i = START;
    while(i > 0) {
      println(i);
      i--;
    }
    println("Liftoff!");
  }
}
```

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    int x = 10;
    for(double START = 0; START < START; START++) {
      println(x);
      x--;
    }
    println("liftoff!!!");
  }
}
```

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    double START = 10;
    while(START >= 1) {
      println(START);
      START = START - 1;
    }
  }
}
```

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    for(int i = 0; i < START + 1; i += 1) {
      int x = START - i;
      println(x);
    }
    println("liftoff");
  }
}
```

```
public class Countdown extends ConsoleProgram {
  public void run() {
  }
}
```

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    double i = START;
    while(i > 0) {
      println(i);
      i -= 1;
    }
    println("Liftoff !");
  }
}
```

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    double i = 10;
    while(i > 1) {
      i -= 1;
    }
  }
}
```

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    int START = 10;
    while(START > 0) {
      println(START);
      START--;
    }
  }
}
```

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    println("10");
    println("9");
    println("8");
    println("7");
    println("6");
    println("5");
    println("4");
  }
}
```

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    int x = 0;
    for(double i = 0; i != START + 1; i++) {
      temp = 10 - i;
      println(temp)
    }
    println("liftoff!");
  }
}
```

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    int START = START;
    println(START);
    while(START >= 3) {
      START--;
      print(START);
    }
    print("LiftOff");
  }
}
```

```
public class Countdown extends ConsoleProgram {
  public void run() {
    int x = 0;
    for(int i = 0; i != START; i += 1) {
      temp = 10 - i;
      println(temp)
    }
    print("Liftoff !!!");
  }
}
```

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    int x = START;
    for(double START = 0; START > START - 1; START++) {
      x--;
    }
  }
}
```

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    println(START);
    double i = 10;
    while(i >= 2) {
      i--;
      println(i);
    }
    print("Liftoff!");
  }
}
```

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    int START = 10;
    println(START);
    while(START > 2) {
      START--;
      println(START);
    }
    println("LIFTOFF");
  }
}
```

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    int x = START;
    for(int START = 0; START != START; START += 1) {
      x--;
    }
  }
}
```

```
public class Countdown extends ConsoleProgram {
  public void run() {
    println("10");
    println("9");
    println("8");
  }
}
```

```
public class Countdown extends ConsoleProgram {
  public void run() {
    println(START);
    double i = 10;
    while(i >= 3) {
      i = i - 1;
      print(i);
    }
  }
}
```

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    for(double START = START; START <= 1; START -= 1) {
      print(START);
    }
  }
}
```

```
public class Countdown extends ConsoleProgram {
  public void run() {
    int START = 10;
    print(START);
    while(START != 0) {
      START--;
      print(START);
    }
  }
}
```

```
public class Countdown extends ConsoleProgram {
  private static final int START = 10;
  public void run() {
    println(START);
    double i = START;
    while(i != 0) {
      i -= 1;
    }
  }
}
```

J Grammar Sample Zoo: PyramidSnapshot