Generative Grading: Neural Approximate Parsing for Verifiable Automated Student Feedback

Anonymous

Abstract

Open access to high-quality education is limited by the difficulty of providing student feedback at scale. In this paper, we present Generative Grading with Neural Approximate Parsing (GG-NAP): a novel computational 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. With this approach, we achieve feedback prediction accuracy comparable to human experts in many 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.

Introduction

Enabling global access to high-quality education at scale is one of the core grand challenges in education. With recent advancements in machine learning, computer-assisted approaches show promise in providing open access to world-class instruction and a reduction in the growing cost of learning (Bowen, 2012). However, a major barrier to this endeavour has been the need to automatically provide *meaningful and timely* feedback on student work.

Learning to provide feedback has proven to be a hard machine learning problem. Despite extensive research that combines massive education data with cutting-edge deep learning (Basu, Jacobs, and Vanderwende, 2013; Hu and Rangwala, 2019; Liu, Xu, and Zhao, 2019; Piech et al., 2015; Wang et al., 2017; Yan, McKeown, and Piech, 2019), most approaches fall short. Five issues have emerged: (1) student work is highly varied, exhibiting a heavy tailed (Zipf) distribution so that most solutions will not be observed even in large datasets, (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) grading is a precision-critical domain since there is a high cost to misgrading a student, and (5) predictions must be explainable and justifiable to instructors and students. These challenges are typical of many human-centred AI problems, such as diagnosing rare diseases or predicting recidivism rates.

When real instructors provide feedback, they perform the difficult task of classifying a student's misconceptions (y) given their solution (x). In practice, instructors are much more adept at thinking "generatively", $p(x, y)$: they can imagine the misconceptions a student might have, and construct the space of solutions a student with these misconceptions would likely produce. Recently, Wu et al. (2018b) used this intuition to show that if student misconceptions and their corresponding solution set can be decomposed in the form of a probabilistic context free grammar (PCFG), then a neural network trained on samples from this PCFG vastly outperforms data-hungry supervised approaches in classifying student misconceptions. While this work provides a novel paradigm, it is limited by the difficulty of writing cognitive models in the form of just PCFGs. Further, the inference techniques of Wu et al. (2018b) do not scale well to more complex problems and provide no notion of verifiability.

In this paper, we address these limitations by introducing a more flexible class of probabilistic program based grammars (PPGs) for describing student cognitive models. These grammars support arbitrary functional transformations and complex decision dependencies, allowing an instructor to model student solutions to more difficult problems like CS1 programming or short-answer questions. These more expressive grammars present a challenging inference problem that cannot be tackled by prior methods from Wu et al. (2018b). In response, we develop Neural Approximate Parsing (GG-NAP): a novel algorithm that parses a given student solution to find an execution trace of the grammar that produces this solution. Not only does this kind of inference allow for classifying misconceptions (the execution trace can be inspected for which confusions are present), but the provided execution trace of the grammar can serve as a verifiable justification for the model's predictions.

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 prior work across a spectrum of public education datasets: introduction to com-

puter programming, short answers to a citizenship test, and graphics-based programming. 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 *halving* grading time. Moreover, the grading decisions made by our algorithm are auditable and interpretable by an expert teacher due to the provided execution trace. Our algorithm is "zero-shot" and thus works for the very first student. Further, writing a generative grammar requires no expertise, and is orders of magnitude cheaper than manually labelling.

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, to help teachers understand learning ability of students and classrooms, or to help teachers customise curriculums, etc. We see this work as an important stepping stone to scaling automated feedback to student work at the level of introductory classes where instructor resources are especially stretched thin.

Background

The Automated Grading Challenge

In computational education, there are two important machine learning tasks related to "grading" student work. First, we consider *feedback prediction*, or labelling a given student solution with misconceptions. These misconceptions usually represent semantic concepts e.g. a student who manually iterates over a sequence may not understand loop structures.

Unlike most machine learning problems however, we cannot solely judge a computational model by just its accuracy on this predictive task. In a safety-critical domain like education, teachers must be able to verify and justify the claims of a computational agent before providing them to the student. Otherwise, we run the costly risk of providing incorrect feedback; a mistake with potentially devastating impact on student learning. Therefore, the second task we tackle is *verifiable prediction*, in which the algorithm must either return a prediction along with a certificate for correctness, or declare uncertainty (and perhaps still provide a best guess). While many methods have been presented for feedback prediction (Piech et al., 2015; Wang et al., 2017; Wu et al., 2018b), to the best of our knowledge, this work is the first to tackle verifiable prediction for grading student work in education.

Difficulty of Automated Feedback

Disregarding the requirement of verifiability, feedback prediction alone has been an extremely difficult challenge in education research. Even limited to simple problems in computer science like beginner drag-and-drop programming, automated solutions to providing feedback have been restricted by limited data and lack of robustness. In 2014, Code.org[1] ran an initiative to crowdsource thousands of instructors to label 55,000 student solutions to simple geometric drawing problems in their block programming language. With over 40,228,194 enrolled students, the problem of automating feedback on problems like these is one of the hardest and most impactful challenges they face. Yet, despite having access to an unprecedented amount of labelled data[2], traditional supervised methods failed to perform well on even these "simple" questions. In the broader landscape of education, the situation is worse: there is hardly ever any labelled data and student solutions are Zipfian i.e. the space of correct solutions is simple but the space of incorrect solutions is enormous (see Fig. 1).

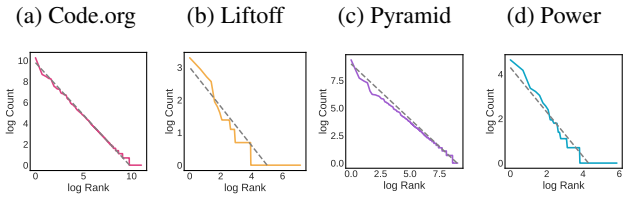


Figure 1: Student solutions (across many domains) exhibit heavy-tailed Zipf distributions, meaning a few solutions are extremely common but all other solutions are highly varied and show up rarely. This suggests that the probability of a student submission not being present in a dataset is high, making supervised learning on a small dataset ineffective.

Generative Grading

Faced with the limitations of traditional supervised approaches, we tackle these grading problems using a "generative" approach. Instead of labelling data, an expert is asked to model the student cognitive process by describing the misconceptions a student might have along with the corresponding space of solutions a student with these misconceptions would likely produce. If we can instantiate these expert beliefs as a real generative model (e.g. probabilistic grammar), then we possess a simulator from which we can sample infinite amounts of "labelled" data, allowing for zero-shot learning. While modelling solutions to large problems is difficult, representing the problem-solving process 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 of student cognition. We refer to this approach as "generative grading".

In previous work, Wu et al. (2018b) represent these student cognition models as instructor-written probabilistic context-free grammars (PCFGs) and use them to generatively grade student submissions to Code.org problems. Although they boast promising results, we find the limitation to context-free grammars excessively restrictive, especially when tackling more complex domains like CS1 programming. Our challenge, then, is to define an expressive enough class of probabilistic models that can capture the complexities of expert priors (and student behaviour), while still being able to do inference and parsing of student solutions.

Neural Parsing for Inference in Grammars

In this section, we define the class of grammars called *Probabilistic Program Grammars* and describe several motivat-

[1]Code.org is one of largest and most widely used online programming resources for beginners in computer science.

[2]Labelling educational data requires expert knowledge, unlike labelling images. For example, 800 student solutions to a block programming problem took 26 hours to label (Wu et al., 2018b).

ing properties that make them useful for generative grading.

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. Perhaps not surprisingly, even with simple programming exercises in Java or Python, this (and more complex) types of conditional execution are abundant.

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 encountered during execution. Here, i_t denotes a unique lexical identifier for each random variable encountered in order and x_{i_t} stores the sampled value. Define the joint distribution (induced by G) over trajectories and productions as $p_G(\tau, y)$.

Given such a grammar, we are interested in *parsing*: this is the task of mapping a production y to the most likely trajectory in the PPG, $\arg\max_\tau p_G(\tau|y)$ 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, CYK) 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.

Neural Inference Engine

The challenge of MAP inference over trajectories is a difficult one. Trajectories can vary in length and contain nonterminals with different support. To approach this, 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:

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

where $\mathbf{x}_{<i_t}$ denotes previous (possibly non-contiguous) nonterminals $(x_{i_1}, \dots, x_{i_{t-1}})$. Eqn. 1 shows that we can learn each posterior $p(x_{i_t}|\mathbf{x}_{<i_t}, y)$ separately. With an autoregressive model $\mathcal{M}$, we can efficiently represent the influence of previous nonterminals $\mathbf{x}_{<i_t}$ using a shared hidden representation over T timesteps. Since the input to $\mathcal{M}$ needs to be fixed dimension, we have to represent all relevant inputs in a consistent manner (see appendix for details).

Firstly, to encode the production y, we use standard machinery (e.g. CNNs for images, RNNs for text) with a fixed output dimension. To represent the nonterminals with different support, we define three layers for each random variable x_{i_t}: (1) a one-hot embedding layer that uses the index i_t to lexically identify the random variable, (2) a value embedding layer that maps the value of x_{i_t} to a fixed dimension vector and (3) an value decoding layer that transforms the hidden output state of $\mathcal{M}$ into parameters of the posterior for the next nonterminal x_{i_t+1}. Thus, the input to the $\mathcal{M}$ is a fixed size, being the concatenation of the value embedding, index embedding, and production encoding.

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

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

where θ are all trainable parameters and $p_\theta(\tau|y)$ represents the posterior distribution defined by the inference engine[3]. At test time, given only a production y, GG-NAP recursively samples $x_{i_t} \sim p_\theta(x_{i_t}|y, \mathbf{x}_{<i_t})$ for $t = 1, \dots, T$ and uses each sample as the input to the next step in $\mathcal{M}$, as in usual sequence generation models (Graves, 2013).

Note that inference over trajectories is much more difficult than just classification. Previous work in generative grading (Wu et al., 2018b) only learned to classify an output program to a fixed set of labels. To draw the distinction, GG-NAP produces a distribution over possible parses where each nonterminal is associated with one or more labels.

Relationship to Viterbi Parsing

To check that neural approximate parsing is a sensible approach, we evaluate it on a simple class of grammars where

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

exact parsing (via dynamic programming) is possible. In (Wu et al., 2018b), the authors released PCFGs for two exercises from Code.org (P1 and P8) that produce block code. These grammars are large: P1 has 3k production rules whereas P8 has 263k. Given a PCFG, we compare GG-NAP

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

Table 1: Agreement between Viterbi and Neural Parsing

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.

Using 5,000 generated 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 and can even parse outputs not in the the grammar to a plausible nearest in-grammar neighbour. Finally, it is *orders of magnitude* faster than Viterbi: 0.3 vs 183 sec for P8 (see appendix).

Verifiable Nearest Neighbour Retrieval

If we can parse a student solution to a trajectory of nonterminals, then we can sample the grammar production from this trajectory—if this sample is equal to the original solution, then that is a proof that the parse was correct. In the case that the sample is not an exact match, we can treat the parsed production as a "nearest in-grammar neighbour" of the original solution, which is still useful in downstream tasks.

More formally, assume we are given a production y from a grammar G. Let the sequence $\widehat{\tau} = \{\widehat{x}_{i_t}\}$ refer to the inferred trajectory for y and $\tau = \{x_{i_t}\}$ refer to the true (unknown) trajectory. If we repeatedly generate from the grammar 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.

In education, verifiable prediction adds an important ingredient of interpretability, whereby teachers can be confident in the feedback that models provide. Furthermore, with intelligent grading systems, the nearest neighbour $\pi(\widehat{\tau})$, along with its known labels, $\widehat{\tau}$, can greatly assist human grading. A grader can "grade the diff" by comparing the real

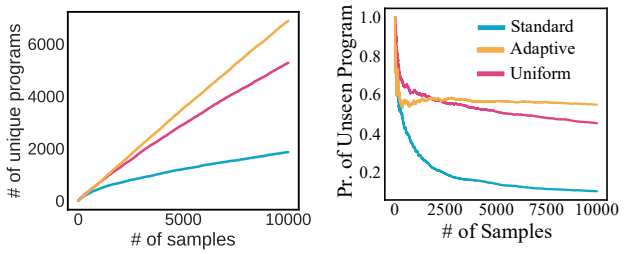


Figure 2: Efficiency of different sampling strategies for Liftoff grammar. (left) Number of unique samples vs total samples so far. (right) Good-Turing estimates: probability of sampling a unique next program given samples so far.

solution with this nearest neighbour and adjusting the labels accordingly. In our experiments, we show this to achieve super-human grading precision while reducing grading time.

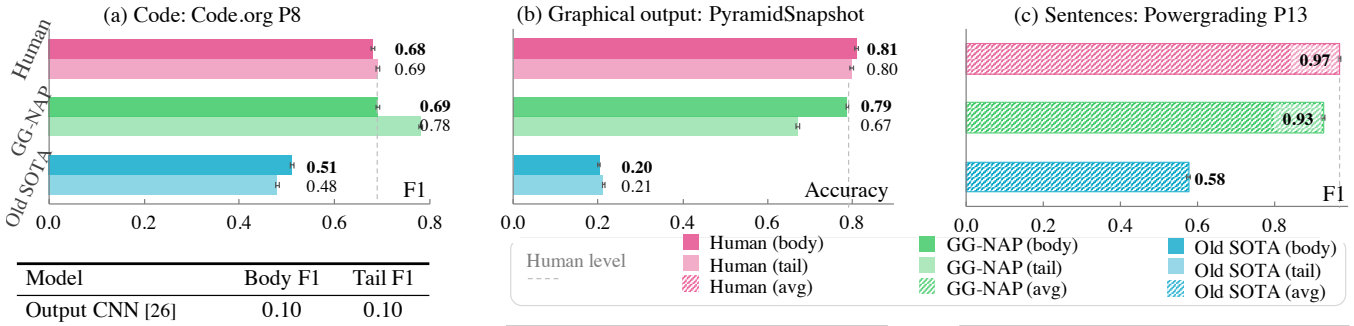
k-Nearest Neighbour Baseline

As a strong baseline for verifiable prediction, we simply use 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 using a linear search of 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 (image, text) and thus the distance metric used for GG-kNN will be domain dependent. Note that GG-kNN is much more costly than GG-NAP in memory and runtime as it needs to store and iterate through all samples.

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.

To make sampling more efficient, we present a novel method called Adaptive Grammar Sampling that down-weights 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 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. 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.



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

Experiments

We test GG-NAP on a suite of public education datasets focusing on introductory courses either from online platforms or large universities. In each, we compare against the existing state-of-the-art (SOTA) model. First, we briefly introduce the datasets, then present results, focusing on a real classroom experiment we conducted. In summary, we find that GG-NAP beats the previous SOTA by a significant margin *in all four educational domains*. Further, it approaches (or surpasses in one case) human performance (see Fig. 3).

Datasets

We consider four educational contexts. Refer to the supplement for example student solutions for each problem.

Code.org (Block Coding) Wu et al. (2018b) 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 (Text) Powergrading (Basu, Jacobs, and Vanderwende, 2013) 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 (Riordan et al., 2017): "name one reason the original colonists came to America". Responses span economic, political, and religious reasons.

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, McKeown, and Piech, 2019) annotated 12k unique snapshots with 5 categories representing "knowledge stages" of understanding.

Liftoff (Java) Liftoff is a second assignment from an university CS1 course that tests looping. Students are tasked to write a program that prints a countdown from 10 to 1 followed by the phrase "Liftoff". We measure the performance of verifiable prediction with GG-NAP and a human-in-the-loop to grade 176 solutions from a semester of students and measure accuracy and grading time.

Results for Feedback Prediction

In each domain except Liftoff, we are given a small test dataset of student programs and labelled feedback. By design, we include each of the labels as a nonterminal in the grammar[4], thereby reducing prediction to parsing. 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 trivially memorised, we focus on the body and tail.

Code.org GG-NAP sets the new SOTA, beating (Wu et al., 2018b) in both the body and tail, and surpassing human performance (historically measured as F1). This is a big improvement over previous work involving supervised classifiers (Wang et al., 2017; Wu et al., 2018b) as well as zero-shot approaches like Wu et al. (2018b), which perform significantly below human quality. By removing restrictions of context-dependence, we are able to easily write richer grammars; combining this with the better predictive power of neural parsing leads to the improved performance. The potential impact of a human-level autonomous grader is large: Code.org is used by 610 million students worldwide, and

[4]In generality, we only require that the labels can be derived deterministically from the nonterminals.

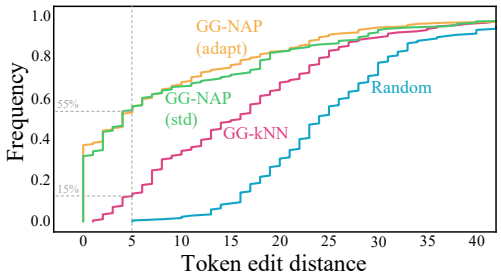


Figure 4: CDF of edit distance between student programs and nearest-neighbours using various strategies.

using GG-NAP could save thousands of human hours for teachers by providing 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, surpassing earlier work that used hand-crafted features (Daxenberger et al., 2014) and supervised neural networks (Riordan et al., 2017). We also note that, since the Powergrading responses contain (simple) natural language, we find these results to be a promising signal that GG-NAP could generalise to domains beyond just computer science classes.

PyramidSnapshot As in the last two cases, GG-NAP is the new SOTA, out-performing baselines (kNN and VGG classifier) from Yan, McKeown, and Piech (2019) by about a 50% gain in accuracy.[5] 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 solutions from just images alone, a feat not possible without the functional transformations allowed in PPGs.

Human Guided Grading in a Classroom Setting

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 investigated its impact on grading accuracy and speed in a real classroom setting. To do this, we created a human-in-the-loop grading system using GG-NAP: for each student solution, a grader is presented with the student solution to grade, as well as a diff to the nearest in-grammar neighbour found using GG-NAP (see Fig. 9 in appendix). This nearest neighbour already has associated labels, and the grader adjusts these labels based on the diff to determine grades for the real solution.

As an experiment, we hired a cohort of expert graders (teaching assistants with similar experience from a large private university) who graded 30 real student solutions to Liftoff. For control, half the graders proceeded traditionally, assigning a set of feedback labels by just inspecting the student solutions. The other half of graders additionally had access to (1) the feedback assigned to the nearest neighbour by GG-NAP and (2) a code differential[6] between the student program and the nearest neighbour. Some example feedback labels included "off by one increment", "uses while loop", or "confused $>$ with $<$". All grading was done on a web application that kept track of the time taken to grade a problem.

We found that the average time for graders with GG-NAP was *507 seconds* while the average time using traditional grading was *1130 seconds*, a more than double increase. Moreover, with GG-NAP, only *3 grading errors* (out of 30) were made with respect to gold-standard feedback given by the course Professor, compared to the *8 errors* made with traditional grading. The improved performance stems from the semantically meaningful nearest neighbours provided by GG-NAP; compared to the GG-kNN baseline, the quality of nearest neighbours of the former are noticeably better (see Fig. 4). Having access to graded nearest neighbours that are semantically similar to the student solution helps increase grader efficiency and reliability by allowing them to focus on only "grading the diff" between the real solution and the nearest neighbour. 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.

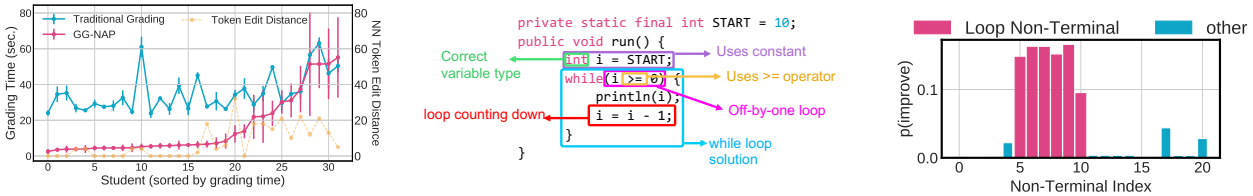
Related Work

"Rubric sampling" (Wu et al., 2018b) 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 supervised classifiers. Our approach builds on this, but GG-NAP operates on a more expressive family of grammars that are context sensitive and comes with new innovations that enable effective inference. From Code.org, we see that expressivity is responsible for pushing GG-NAP past human level performance. Furthermore, our paradigm adds an important notion of verifiability lacking in previous work, opposing the typical 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 (Le, Baydin, and Wood, 2016). With PPGs, we get a nice interpretation of compiled inference as a parsing algorithm. We also show the promise of compiled inference in much larger probabilistic programs (with skewed prior distributions). Previous work usually involved less than ten random variables whereas our grammars grow to hun-

[5]These baselines were trained on 200 labelled images.

[6]The differential is in the style of Github. See appendix .



(a) Classroom Experiment Results

(b) Automated Dense Feedback

(c) Auto-improving Grammars

Figure 5: (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) to the GG-NAP nearet neighbour. (b) GG-NAP allows for automatically associating student work with fine-grained automated feedback. (c) Given a Liftoff grammar that can only increment up, we can track nonterminals where inference often fails and use that to estimate where the grammar need improvement. The height of each bar represents the likelihood that improvements are needed for that nonterminal.

dreds (Lake, Salakhutdinov, and Tenenbaum, 2015; Le, Baydin, and Wood, 2016; Wu et al., 2016).

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 in a RNN-based variational autoencoder (Bowman et al., 2015) that specifies a posterior distribution over many categorical variables. Further, the index embedding layer serves as a unique lexical identifier, similar to the positional encoding in transformers (Vaswani et al., 2017). Finally, the verifiable properties of GG-NAP have strong ties to explainable AI (Hancock et al., 2018; Koh and Liang, 2017; Ross and Doshi-Velez, 2018; Selvaraju et al., 2017; Wu et al., 2018a).

Discussion

Highlighting feedback in student solutions Rather than predicting feedback labels, it would be more useful to give "dense" feedback that highlights the section of the code or text responsible for the student misunderstanding. This would be much more effective for student learning than vague error messages currently found on most online education platforms. To achieve this, we use GG-NAP to infer a trajectory, $\tau = \{x_{i_t}\}$ for a given production y. For every nonterminal x_{i_t}, we want to measure its "impact" on y. If for each x_{i_t} 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 (via string intersection). Fig. 5a shows a random program with automated, segment-specific feedback given by GG-NAP. This level of explainability is sorely needed in both education and AI and could revolutionise how students are given feedback at scale.

Cost of writing good grammars. Writing a good grammar does not require special expertise and can be undertaken by a novice in a short time. For instance, the PyramidSnapshot grammar that sets the new SOTA was written by a first-year undergraduate within a day. Furthermore, many aspecst of grammars are re-usable: similar problems will share nonterminals and some invariances (e.g. the nonterminals that capture different ways of writing i++ are the same everywhere). This means every additional grammar is easier to write since it likely shares a lot in structure with existing grammars. Moreover, compared to weeks spent hand-labelling data, the cost of writing a grammar is orders of magnitude cheaper and leads to much better performance.

Automatically improving grammars Building PPGs is an iterative process; a user wishing to improve their grammar would want a sense of where it is lacking. Fortunately, given a set of difficult examples where GG-NAP does poorly, we can deduce the nodes in the PPG that consistently lead to mistakes and use these to suggest components to improve. To illustrate this, we took the Liftoff PPG which contains a crucial node that decides between incrementing up or down in a "for" loop, and removed the option of incrementing down. Training GG-NAP on the smaller PPG, we fail to parse student solutions that "increment down". Given such a solution, to compute the probability that a nonterminal is "responsible" for the failure, we find its GG-NAP nearest neighbour and associated trajectory. Then, for each nonterminal in this trajectory, we can associate it with its substring in the solution (via highlighting). By finding the nonterminals where the substring often differs between the neighbour and the solution, we can identify nonterminals that often causes mismatches. Fig. 5c shows the distribution over which nodes GG-NAP believes to be responsible for the failed parses. The top 6 nonterminals that GG-MAP picked out all rightfully relate to looping and incrementation.

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

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

Bowen, W. G. 2012. The cost diseasein higher education: is technology the answer? *The Tanner Lectures Stanford University*.

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

Chang, H. S.; Fu, M. C.; Hu, J.; and Marcus, S. I. 2005. An adaptive sampling algorithm for solving markov decision processes. *Operations Research* 53(1):126–139.

Daxenberger, J.; Ferschke, O.; Gurevych, I.; and Zesch, T. 2014. 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*, 61–66.

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

Graves, A. 2013. Generating sequences with recurrent neural networks. *CoRR* abs/1308.0850.

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

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

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

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

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

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

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

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

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

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

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

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

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

Wang, F., and Landau, D. 2001. Efficient, multiple-range random walk algorithm to calculate the density of states. *Physical review letters* 86:2050–3.

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

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

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

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

Yan, L.; McKeown, N.; and Piech, C. 2019. 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, 119–125. New York, NY, USA: ACM.

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 (Kingma and Ba, 2014) 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 (Simonyan and Zisserman, 2014) with Xavier initialization (Glorot and Bengio, 2010) 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.

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: $D_G = \{(\tau^{(m)}, y^{(m)})\}_{m=1}^{M}$.

procedure ADAPTIVESAMPLE(G, d, r, M)
  $D_G \leftarrow \{\}$
  while $|D_G| < M$ **do**
    $\tau, y \leftarrow$ SAMPLEGRAMMAR(G)
    if $(\tau, y) \notin D_G$ **then**
      $D_G \leftarrow D_G \cup \{(\tau, y)\}$
    for $i \leftarrow 0$ to $|\tau|$ **do**
      $x_i \leftarrow \tau[i]$ ▷ get i-th node in trajectory, $\tau = \{x_i\}_{i=1}^{T}$, of length T
      $p(x_i|\mathbf{x}_{<i}) \leftarrow \frac{p(x_i|\mathbf{x}_{<i})}{r + d^{|\tau|-i} \cdot p(x_i|\mathbf{x}_{<i})}$
      $p(x_i|\mathbf{x}_{<i}) \leftarrow$ 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 (Chang et al., 2005) and shares similarities with Wang-Landau from statistical physics (Wang and Landau, 2001).

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.

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

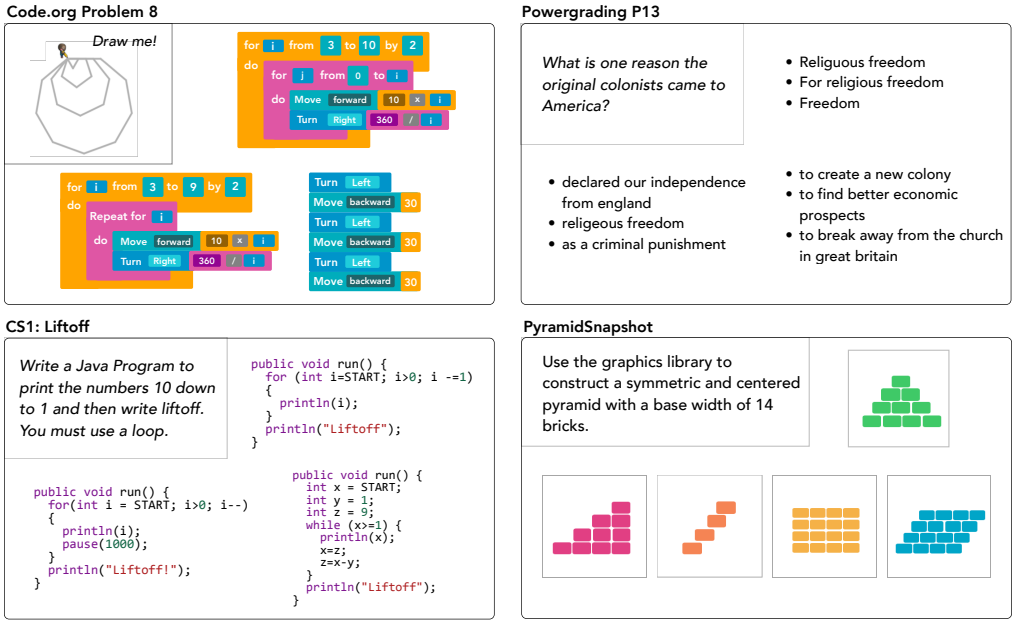


Figure 6: We show the prompt and example solutions for 4 problems from programming assignments to history tests.

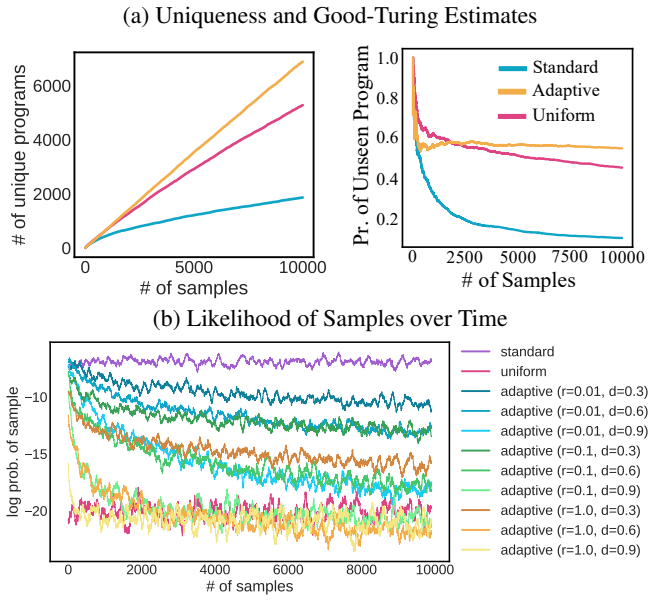


Figure 7: Effectiveness of sampling strategies for Liftoff. Left/Middle: Number of unique programs generated (left) and Good-Turing estimate (middle) as a function of total samples. Right: Likelihood of generated samples over time for various sampling strategies. In particular, we note the effect of 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.

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[7] 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 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.

NAP Architecture

Fig. 8 visualises the architecture for the neural inference engine in NAP. The PRODUCTIONENCODER network is responsible for transforming unstructured images and text to a fixed vector space representation, using a domain specific architecture like a CNN for images or RNN for text. The

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

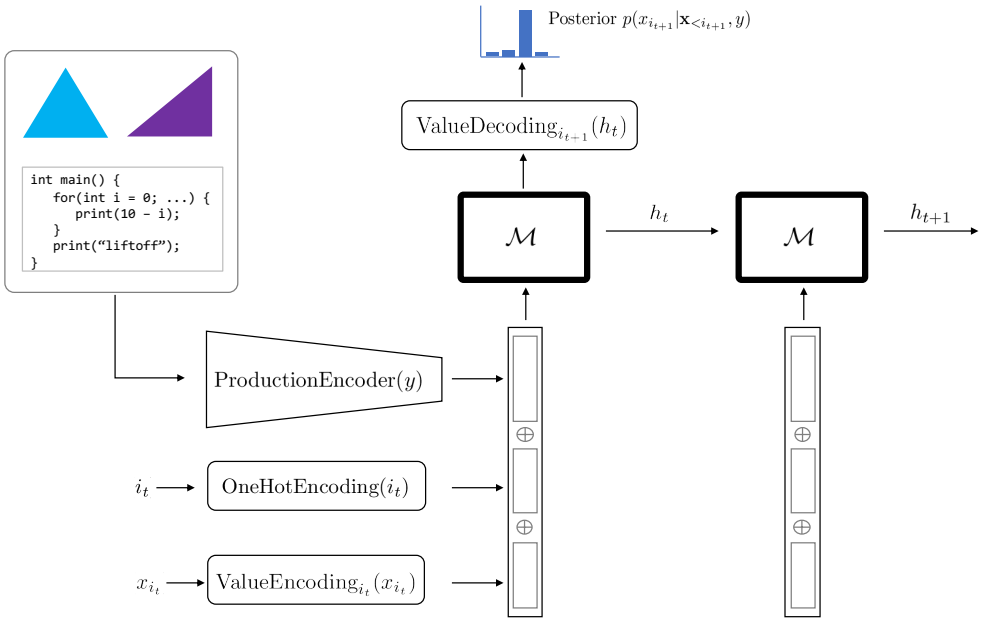


Figure 8: Architecture of the neural inference engine. We show a single RNN update to parameterize $p(x_{i_{t+1}}|\mathbf{x}_{<i_{t+1}}, y)$. This procedure is repeated for each T, the length of the trajectory.

lexical index of the current random variable, i_t, is encoding using the ONEHOTENCODING transformation and its current value, x_{i_t}, is encoded to a fixed dimension using a VALUEENCODING$_{i_t}$ layer that is specific to this random variable. To get a posterior distribution over the next random variable, the VALUEDECODING$_{i_{t+1}}$ transformer specific to the next random variable maps from the hidden state of $\mathcal{M}$ to a distribution over values of the next random variable.

At train time, the inputs to the autoregressive model, $\mathcal{M}$, at each timestep, t, are the true values of x_{i_t} from the data. We train the model and all encoding/decoding layers end-to-end by backpropogating per-timestep gradients using the cross-entropy loss of the posterior distribution $p(x_{i_{t+1}}|\mathbf{x}_{<i_{t+1}}, y)$ output by the model and the true value taken on by $x_{i_{t+1}}$.

At inference time, we do not have a true value for $x_{i_{t+1}}$ to use in the next timestep so we sample this value from the posterior produced by $\mathcal{M}$. This sample is then fed to the next timestep of $\mathcal{M}$ and the process is repeated until the trajectory is completely determined.

Grading UI

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.

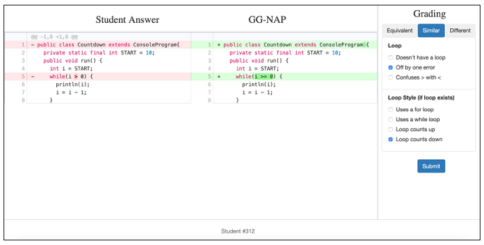


Figure 9: Grading UI based on GG-NAP

GG-NAP and Viterbi Cost Comparison

Table 2 compares the wall clock cost of Viterbi and GG-NAP on very large PCFGs. We can see significant time savings (of 700x).

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

Table 2: Inference Cost of Viterbi and Neural Parsing

Grammar Sample Zoo

In the following, we show many generated samples from the PPGs for Powergrading, Code.org, Liftoff, and Pyramid-Snapshot (in that order).

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?

```
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)
```

```
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;
    }
  }
}
```