Zero-shot Learning of Hint Policy via Reinforcement Learning and Program Synthesis

Aleksandr Efremov
MPI-SWS
aefremov@mpi-sws.org

Ahana Ghosh
MPI-SWS
gahana@mpi-sws.org

Adish Singla
MPI-SWS
adishs@mpi-sws.org

ABSTRACT

Intelligent tutoring systems for programming education can support students by providing personalized feedback when a student is stuck in a coding task. We study the problem of designing a *hint policy* to provide a next-step hint to students from their current partial solution, e.g., which line of code should be edited next. The state of the art techniques for designing a hint policy use supervised learning approach, however, require access to historical student data containing trajectories of partial solutions written when solving the task successfully. These techniques are limited in applicability when needed to provide feedback for a new task without any available data, or to a new student whose trajectory of partial solutions is very different from that seen in historical data. To this end, we tackle the *zero-shot* challenge of learning a hint policy to be able to assist the very first student who is solving a task, without relying on any data. We propose a novel *reinforcement learning* (RL) framework to solve the challenge by leveraging recent advancements in RL-based neural *program synthesis*. Our framework is modular and amenable to several extensions, such as designing appropriate reward functions for adding a desired feature in the type of provided hints and allowing to incorporate student data from the same or related tasks to further boost the performance of the hint policy. We demonstrate the effectiveness of our RL-based hint policy on a publicly available dataset from Code.org, the world's largest programming education platform.

1. INTRODUCTION

In recent years, there has been an increasing focus on developing educational tools for STEM (science, technology, engineering, and mathematics) and computing. Problem-solving skill, i.e., ability to solve multi-step problems by deductive reasoning, is one of the key ingredient of learning in these domains [14, 25]. For instance, while working on a coding task, a student iteratively writes, tests and refines the code to arrive at the final solution [8, 23, 27, 24].

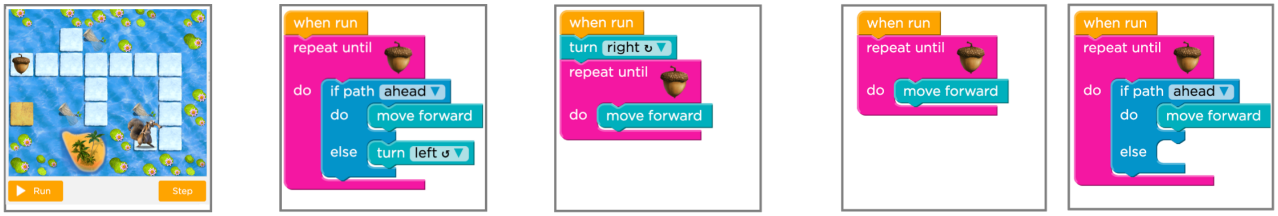
Aleksandr Efremov, Ahana Ghosh and Adish Singla "Zero-shot Learning of Hint Policy via Reinforcement Learning and Program Synthesis" In: *Proceedings of The 13th International Conference on Educational Data Mining (EDM 2020),* Anna N. Rafferty, Jacob Whitehill, Violetta Cavalli-Sforza, and Cristobal Romero (eds.) 2020, pp. 388 - 394

One of the difficulties in designing assistive algorithms for these open-ended coding tasks is that the state space, i.e., the set of partial solutions that students might arrive at when solving the task, is *unbounded*. For instance, for a simple coding task from the *Hour of Code* (HOC) challenge by Code.org [5], the correct solution contains only 5 blocks (see Figure 1), whereas students can create millions of unique partial solutions in the process of solving the task [23]. When solving such tasks, it is evident that students can get stuck at a state (i.e., a partial solution) and do not know how to proceed (i.e., which action/edit to apply). Intelligent tutoring systems empowered by machine learning techniques held a great promise in supporting such stuck students by providing personalized feedback, e.g., explaining misconceptions and giving guidance on what to do next [31, 16, 2, 24].

We focus on the well-studied feedback mechanism in programming education called *next-step* hints: When a student is stuck at a given state, the system suggests the next edit that student should make to their current code to proceed [3, 8, 16, 23, 27, 20]. In the context of block-based languages that are extremely popular in educational tools for visual programming [21, 5, 24], the suggested hints correspond to one of the allowed actions from the student's current code (e.g., adding or removing a block, and changing a conditional in one of the blocks), see Figure 1. Inspired by the work of [3, 23, 20], we refer to the function that provides such hints to the student from any partial solution as *hint policy*.

The key challenge in designing a hint policy is that the space of partial solutions is unbounded even for simple coding tasks and there is a huge variability in students' trajectories of partial solutions [23, 20, 34]. A number of techniques proposed in the literature use a graph representation of the task (with nodes denoting partial solutions and edges denoting single edits that convert one partial solution to another) [3, 9, 23, 35, 27]. These techniques then use historical student data and domain knowledge to capture the editing behavior of capable students (or experts) on this graph. However, these techniques face serious scaling issues as the problem size grows and are only applicable in settings where we have access to large volume of historic data for the task.

In recent years, new techniques have been developed using a supervised learning approach. These techniques leverage code embeddings to compactly represent the space of partial solutions and can provide hints to students with trajectories that have never been observed in the historical data [22, 20].



(a) HOC-18 task (b) Correct solution (c) Student's current code (d) Two subsequent next-step hints by the hint policy

Figure 1: Illustration of next-step hints feedback by our hint policy. **(a)** shows the HOC-18 task from the *Hour of Code* (HOC) challenge by Code.org [5]. A student solves the task by starting from empty code and builds up the solution using blocks available in the visual interface, also see [23]. **(b)** shows the correct solution—this is the code that solves the task with minimal number of 5 blocks. **(c)** shows the current partial solution of a student who is solving this task. **(d.left)** shows the next-step hint by our hint policy that will be provided as feedback to the student. **(d.right)** shows the subsequent next-step hint by the policy if student were to ask another hint after receiving the first hint.

In particular, the state of the art technique by [20], *Continuous Hint Factory* (CHF), learns a *regression function* as the hint policy which can identify the most likely hint as a vector in an embedding space and then translates this vector back into a human-readable edit. In comparison to techniques using graph representations, CHF is more scalable and requires access to fewer samples of student data (just enough to learn the generic editing behavior of capable students or experts for the task).

While these state of the art supervised learning techniques are less data-hungry and computationally more powerful, they are still limited in applicability when needed to provide feedback for a new task without any available data, or to a new student whose trajectory of partial solutions is very different from that seen in historical data. Especially with intelligent tutoring systems having the ability to generate tasks on the fly [28, 1, 12], the problem of providing feedback to the very first student on a task is increasingly important. In this paper, we tackle the following *zero-shot* learning challenge: *Can we design a hint policy for a task to provide hints to the very first student solving the task?*

1.1 Our Approach and Contributions

Our approach towards zero-shot learning of hint policy is based on the *reinforcement learning* (RL) framework. In the RL terminology, the set of all possible partial solutions corresponds to the state space, the possible edits from a partial solution defines the state-dependent actions and transition dynamics, and reaching the correct solution quickly yields higher reward (we refer the reader to [26, 29] for a background on RL). Our framework is inspired by recent works [4, 11] that have shown that deep-RL techniques applied to neural embeddings of the code are effective in learning policies to synthesize new programs and to do program repair even if no/minimal training data is available for the task. Intuitively, the problem of providing a hint from a current partial solution is equivalent to one-step of synthesizing the program from this partial solution [17, 10]. However, we note that learning hint policy using RL poses its own practical challenges because the policy needs to provide hints from *any* partial solution which could be arbitrarily bad—this is in contrast to program synthesis and program repair where the initial starting states for RL are limited to either an empty code [4] or a set of partial solutions which are close to the correct solution [11], respectively. The idea of using RL for designing hint policy is also inspired by the seminal work on *Hint Factory* [3]; however, unlike [3] which relies on historical student data and uses the graph representation of partial solutions, our RL framework uses code embedding and a neural network policy for efficient training.

One might ask what are the advantages of using RL compared to supervised learning techniques for zero-shot challenge. First and foremost, RL enables an effective self-exploration of the solution space by leveraging reward signals (such as receiving higher rewards when a policy can synthesize the correct solution in a fewer steps or can reduce compiler errors). Furthermore, the RL framework is amenable to several extensions for boosting the performance. For instance, if additional student data if available from the same or related tasks, it is possible to bootstrap by combining techniques from imitation learning within RL framework [19, 11]. Also, we can easily incorporate additional human knowledge or features into the policy by designing appropriate reward functions [4, 11]. In summary, this power and flexibility of the RL framework makes it especially suitable for zero-shot learning as it gives us the following ingredients: (i) automatically exploring the solution space or generating synthetic training data [32, 11, 34], (ii) incorporating any available data or expert knowledge to bootstrap and boost the performance [15, 34], and (iii) transferring knowledge from one task to another [18, 6, 7]. Below, we summarize our main contributions:

- We introduce the zero-shot challenge for learning a hint policy to provide next-step hints to the very first student working on a coding task.
- We propose RL framework for zero-shot learning of hint policy. Our framework leverages the representation power of code embedding and a neural network policy for efficiently learning to provide hints. The framework is amenable to several important extensions, e.g., bootstrapping via additional data if available.
- We evaluate the performance of our RL-based hint policy on a publicly available dataset from Code.org, the world's largest programming education platform [5, 23]. We show significant improvements in next-step hint accuracy w.r.t. the state of the art supervised learning technique.

2. PROBLEM FORMULATION

In this section, we formalize the problem of learning next-step hint policy for programming education.

2.1 Coding Task, Partial Solutions, and Edits

We define the problem in the context of a fixed coding task (e.g., HOC-18 task as shown in Figure 1a). We assume that the correct solution for the task is known. For brevity of presentation, we consider that the correct solution is unique (in fact, the uniqueness holds for HOC-18 task, see Figure 1b). We denote all possible partial solutions for the task by the set S. Note that S is a countable, infinite set. A partial solution $s \in S$ is a piece of code, e.g., as shown in Figure 1, and we denote the correct solution by $s^* \in S$. For any $s \in S$, we define the set of edits that can be applied to s by the action set A_s. In block-based languages, the set A_s corresponds to adding or removing a block in s, editing a conditional for one of the blocks in s, or moving blocks within s. For a partial solution $s \in S$ and an edit $a \in A_s$, the next partial solution obtained by applying a to s is denoted as $s \oplus a$.

2.2 Hint Policy for Next-step Edits

When a student attempts the task, they generate a trajectory of partial solutions denoted as $\xi = (s_0, s_1, s_2, \ldots, s_k)$ where k is the trajectory length. Here, s_0 is the empty code, and s_k is the student's latest/current partial solution. Upon reaching s_k, the student might be stuck and is unable to decide how to proceed. Our goal is to help this student by providing feedback as the next-step hint $a \in A_{s_k}$ in the form of an edit that allows the student to make progress. Figure 1c shows one such partial solution s_k and Figure 1d (left) shows the next-step hint that could be provided.

Formally, the next-step hint policy $\pi(\cdot \mid \xi)$ provides a probability distribution with support over actions A_{s_k} where s_k is the last partial solution in the trajectory ξ. Note that when a policy depends on the whole trajectory ξ, then it can infer the knowledge of the student based on this trajectory and can provide personalized hints. However, the existing hint policy techniques discussed in Section 1 consider myopic policies. A myopic policy $\pi(\cdot \mid s_k)$ provides a probability distribution with support over actions A_{s_k} and takes as argument only the last partial solution s_k (i.e., ignoring the trajectory of how student reached s_k). In our work, we also focus on learning such a myopic hint policy.

2.3 Evaluation Criteria

As an evaluation criterion, we use the standard approach in literature (e.g., see [23, 20]) where the performance of a hint policy is measured in terms of prediction accuracy. We assume access to a set of expert annotations given by $D_{\text{hints}} = \{(s_i, N(s_i)\}_{i=1,2,\ldots,n}$: here, for a partial solution $s_i \in S$, the experts have annotated that the next partial solution where a student should transition to should be among the set $N(s_i) \subseteq S$. In our experiments, we will use the publicly available annotation dataset from [23] for evaluating hint policy on HOC tasks. For a policy π, we use the following notion of *unweighted* prediction accuracy:

$$\frac{1}{n}\sum_{i=1}^{n}\Big(\sum_{a\in A_{s_i}} \pi(a \mid s_i)\cdot \mathbb{1}\big(s_i \oplus a \in N(s_i)\big)\Big) \qquad (1)$$

where $\mathbb{1}(.)$ represents an indicator function (cf. [23] which uses a notion of accuracy *weighted* by frequency). Note that, this measure of prediction accuracy does not capture the long-term pedagogical value of providing hints to students, and we further discuss this as future work in Section 5.

3. LEARNING HINT POLICY USING RL

In this section, we present our approach to zero-shot learning of hint policy via reinforcement learning (RL) framework.

3.1 RL Framework

3.1.1 Hint policy learning environment as an MDP

In reinforcement learning, a learning algorithm (agent) interacts with an environment typically modelled as a Markov Decision Process (MDP). Here, we present the MDP corresponding to the problem of learning hint policy. We define the MDP $M = (S, A, P, R, S_0)$ as follows:

- S corresponds to the set of partial solutions;
- $A = \cup_{s\in S} A_s$ is the set of all possible actions, and A_s is the set of actions or edits possible in state s;
- $P : S \times A \times S \to \mathbb{R}$ denotes the transition dynamics. $P(s' \mid s, a)$ is defined only for $a \in A_s$. We have $P(s' \mid s, a) = 1$ for $s' = s \oplus a$, and 0 otherwise.
- $R : S \times A \to \mathbb{R}$ denotes the reward function. $R(s, a)$ is defined only for $a \in A_s$. A simple reward function could be to set a small negative reward for every action taken and a high reward for reaching the correct solution termed as "goal" (i.e., when $s \oplus a = s^*$). We will discuss more about designing rewards in Section 3.3.
- $S_0 \subseteq S$ is the set of initial states. This corresponds to the states which would be used to initialize an episode when training the hint policy. One way to pick set S_0 is to randomly sample states from S, limited to some upper limit on the code size.

We consider an episodic, finite horizon learning setting [29, 26]: A learning episode starts with an initial state s_0 sampled at random from the set S_0, then the agent interacts with the environment over discrete time steps t, and the episode ends when one of the following happens: (i) either the agent reaches goal state s^*, or the episode length exceeds a pre-specified threshold (set to 20 in our experiments).

3.1.2 Policy gradient methods

While a variety of RL algorithms can be used to learn a policy, we consider policy gradient methods which have proven to be highly effective for dealing with large-scale problems [29, 11, 4]. These methods learn a parametrized policy $\pi_\theta(a \mid s)$ where θ represents the parameters; then, a gradient ascent method is employed to update parameters that would increase the expected reward of the policy in the MDP. In our work, we use a neural network to learn the policy, i.e., θ represents the weights of the network. Given a state s and action a, the policy network parametrized by θ outputs a score $H_\theta(a \mid s)$. Using these scores, we define the policy by the following softmax distribution: $\pi_\theta(a \mid s) = \frac{\exp^{H_\theta(a \mid s)}}{\sum_{a'\in A_s}\exp^{H_\theta(a' \mid s)}}$.

We use the classic REINFORCE policy gradient method (see [29, 33]) to update the weights of the network. In an episode, the RL agent performs an update as follows. First, an initial state s_0 is sampled, and then the policy π_θ is executed until the episode ends, thereby generating a sequence of experience given by $(s_t, a_t, r_t)_{t=0,1,2,\ldots L}$ where L represents the episode length. Then, in this episode, for each

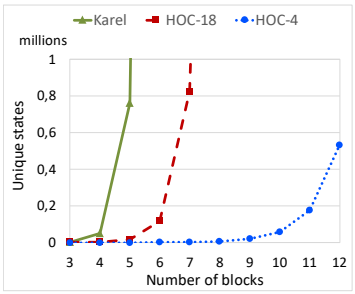
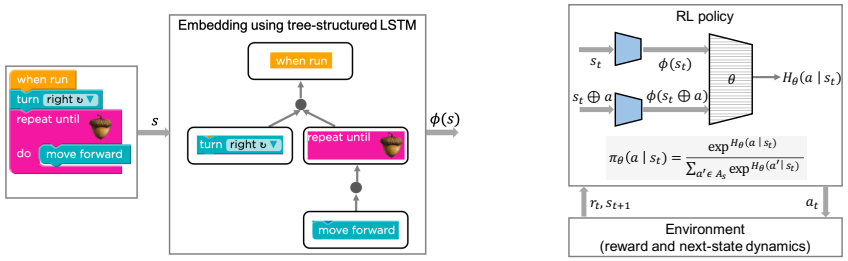


Figure 2: Number of states (i.e., unique partial solutions) grows exponentially w.r.t. the size (i.e., number of code blocks).



(a) Code embedding using tree-structured LSTM (b) Neural architecture of our hint policy

Figure 3: Illustration of our approach to learning hint policy using RL. **(a)** shows the module for generating neural code embedding using tree representation of code. For a state s, the module outputs $\phi(s)$. **(b)** shows the module for our hint policy which uses code embedding for state representation. Details are provided in Section 3.1 and 3.2.

$t \in [0, L]$, we use the following gradient update with η as learning rate:

$$\theta \leftarrow \theta + \eta \cdot \underbrace{\left(\sum_{\tau=t}^{L} r_\tau\right) \cdot \left(\nabla_\theta \log\left(\pi_\theta(a_t \mid s_t)\right)\right)}_{\text{gradient at time step } t \text{ in an episode}} \quad (2)$$

This gradient update can be computed efficiently for our setting—we refer the reader to [33, 29] for detailed discussion. We provide the implementation details in Section 4.2.

3.2 Efficient Learning of Hint Policy

3.2.1 *Dealing with infinite state space*

Figure 2 shows the number of states (unique partial solutions) w.r.t. the size (number of blocks) of a partial solution. Here, for reference, we also show number of states for a more complex language Karel [21]. Note that even if the correct solution is of small size (e.g., 5 for HOC-18 and HOC-4 tasks), the struggling students end up writing large partial solutions even up to 50 blocks length [23]. To deal with this computational challenge of a very large state space, we rely on code embeddings to have a featurized state representation. In our work, we train code embedding inspired by recent developments in using structured RNNs for embeddings, in particular Tree-RNN model by [22] used for HCO-18 embeddings and Tree-LSTM model by [30]. We represent the code as an Abstract Syntax Tree (AST) as shown in Figure 3a, and then this tree structure is used to process the blocks. When training, we require syntactic edit distance between raw states to be preserved after the embedding. In Section 4.2, we provide a more detailed description of the process used to learn the code embedding.

3.2.2 *Dealing with state-dependant action sets*

In a typical RL setting, the action set A is finite, and the standard architecture for training the network is to have $\phi(s)$ as input and the scores $H_\theta(a \mid s)\ \forall\ a \in A$ as output (i.e., output layer has $\mathbb{R}^{|A|}$ size). In our setting, the action set A is infinite, and the allowable actions from a state s given by set A_s are state-dependant. To tackle this challenge, we use the neural architecture as illustrated in Figure 3b. We train a network which takes as input both $\phi(s)$ and $\phi(s \oplus a)$. To evaluate the probability of taking action a from state s, we first compute scores $H_\theta(a' \mid s)$ for all $a' \in A_s$, and then probability of action $\pi_\theta(a \mid s)$ is given by the softmax distribution.

3.3 Incorporating Additional Knowledge

3.3.1 *Designing rewards*

We can easily incorporate additional human knowledge or features into the policy by designing appropriate reward functions [4, 11]. For instance, by changing the reward values $R(s, a)$ based on the type of action a (e.g., deleting an existing block vs. adding a new block), we can train a hint policy that favours certain types of hints. One can further incorporate more complex criterion such as suggesting hints at the last line in the code to capture students' current focus of attention which is important for better interpretability of hints [20]. Reward design also allows us to incorporate intermediate partial solutions that serve as milestones toward the final correct solution. By providing positive rewards for such states representing milestones, our hint policy would automatically learn to steer the students towards such states. Furthermore, this approach can also help in speeding-up the learning process of the RL algorithm by dealing with sparse reward problem (see Section 4.2 on how we use this idea to speed up the learning).

3.3.2 *Bootstrapping from data when available*

While we introduced RL framework to tackle the zero-shot challenge, the proposed framework allows one to bootstrap from additional student data if available from the same or related tasks. In fact, the existing RL-based techniques used in program synthesis and repair (see [4, 11]) have shown that substantial performance gain and convergence speed-up can be obtained by bootstrapping from available data.

We incorporate student data to bootstrap RL-based hint policy as follows. Consider we have access to a set of trajectories of successful students or experts who solved the task, given by $\Xi = \{\xi_j\}_{j=1,2,\ldots}$. From these trajectories, we can obtain dataset of edits made by successful students, represented as $D_{\text{train}} = \{(s_i, s_i \oplus a_i)\}_{i=1,2,\ldots}$. The RL policy network can be bootstrapped by additionally training from D_{train} using cross-entropy loss. The gradient update is given below where η' represents the learning rate:

$$\theta \leftarrow \theta + \eta' \cdot \underbrace{\left(\sum_{(s, s\oplus a) \in D_{\text{train}}} \nabla_\theta \log\left(\pi_\theta(a \mid s)\right)\right)}_{\text{gradient for cross-entropy loss}} \quad (3)$$

Further implementation details are provided in Section 4.2.

4. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of our RL-based hint policy on a publicly available dataset from Code.org [5].

4.1 Hour of Code Tasks

We consider HOC-18 and HOC-4 tasks from the *Hour of Code* (HOC) challenge by Code.org, the world's largest programming education platform [5, 23]. HOC-18 task, shown in Figure 1, is an advanced task in HOC challenge with 7 different types of blocks ("move forward", "turn left", "turn right", "repeat until", and "IF ELSE" with three different types of conditionals). HOC-4 is a simpler task with only 3 types of blocks ("move forward", "turn left", "turn right"). For the zero-shot setting, we do not require availability of any student data for comparing different hint-policy techniques (see Figures 4a and 4c for $x = 0$ on the x-axis). Beyond zero-shot setting, we also evaluate the performance when additional data becomes available (see Figures 4a and 4c for $x = 9, 12, 15$ on the x-axis). For these experiments, we use the trajectories of successful students from the dataset provided by [5, 23]. We refer the reader to [5, 23] for further details about these tasks and the available dataset.

4.2 Implementation Details

Here, we briefly provide implementation details for the following: (i) code embedding, (ii) RL-based hint policy, and (iii) three baselines. Some details are omitted because of lack of space—the source code would be made publicly available with the final version of the paper for reproducibility.

4.2.1 Code embedding

We learn a separate embedding for HOC-18 and HOC-4 tasks, and the code embedding is learnt prior to training the hint policy. We begin by sampling 400 random states limited to a size up to 6 blocks, and then use pairwise syntactic edit-distance between these states to generate a training dataset containing triplets of the form $\big(s, s', d_{\text{synt}}(s, s')\big)$: here $d_{\text{synt}}(s, s')$ represents the syntactic edit-distance between s and s' in terms of the number of edits required to convert s to s'. Given these triplets, we train a neural embedding $\phi(.)$ so the $||\phi(s) - \phi(s')||_2 \approx d_{\text{synt}}(s, s')$. As shown in Figure 3a, we use the Abstract Syntax Tree (AST) representation of a state s which is then traversed in a preorder depth-first way to produce a sequence of blocks. The resulting sequence is passed through bi-directional LSTM where each unique block of the HOC language is encoded differently (cf., Tree-RNN model of [22] and Tree-LSTM model of [30]). The size of the feature representation used for our experiments is given by $dim_\phi = 40$, i.e., $\phi(s) \in R^{|40|}$.

4.2.2 RL-based hint policy

For the policy network, we use a 5-layer fully connected neural network with the following architecture: (i) the input layer has $2 \times dim_\phi$ units for $\phi(s)$ and $\phi(s \oplus a)$; (ii) the first three hidden layers have 128 hidden units and the fourth hidden layer has dim_ϕ hidden units; and (iii) the output layer linearly aggregates dim_ϕ values from the last hidden layer to produce the score $H(a \mid s)$. All hidden units use ReLU activations with a dropout rate of 0.1, and we use ADAM optimizer for training [13]. The policy actions are taken using a softmax distribution as discussed in Section 3. Below, we discuss the rewards and stopping criteria used for training, separately for zero-shot setting and when bootstrapping from available student data:

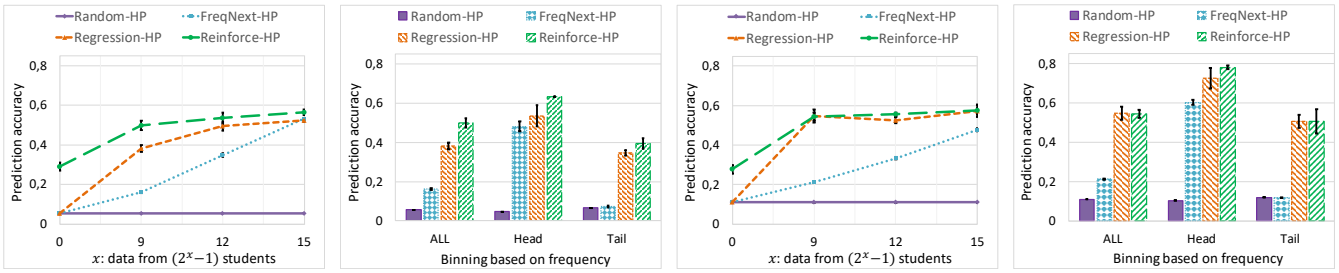
- *Zero-shot learning setting:* We set reward $R(s, a)$ as $+100$ when $s \oplus a = s^*$, and -1 otherwise. The training is done until the average reward of the policy is saturated. To further speed up the convergence, we use intermediate rewards in the training process by adding an additional term of $-d_{\text{synt}}(s \oplus a, s^*)$ to the reward. Here, d_{synt} represents the syntactic edit-distance between two states (same function as used in generating training data for code embedding). These intermediate rewards during the training process allowed us to speed up the convergence by order of magnitude, without effecting the overall performance of the trained policy. After this speed up, the number of episodes required until convergence varied from 5000 to $20,000$.
- *Additional student data is available:* We first pre-train the network using cross-entropy loss with the data sampled from D_{train}. This pre-training is done for 20 epochs where each epoch consists of multiple gradient updates as follows: A batch of data is sampled from D_{train} of size given by $batchsz = 32$ and a gradient update is performed using this batch as per Eq. 3; this process is repeated $\frac{|D_{\text{train}}|}{batchsz}$ within an epoch. After this pre-training phase, we train the policy network using rewards for 2000 episodes using the gradient updates in Eq. 2. Given that the pre-training phase already provides a good initialization of the policy network, we used modified reward signals in this case as compared to the zero-shot setting: (i) we set reward of $+20$ for reaching the goal instead of $+100$ and (ii) we reduced the value of intermediate rewards and set it to $-0.05 \cdot d_{\text{synt}}(s \oplus a, s^*)$ by scaling it down.

4.2.3 Baselines

We compare our RL-based hint policy REINFORCE-HP w.r.t. several baselines as discussed below. In particular, we consider baseline techniques which can be implemented efficiently, without requiring any explicit graph representation of the state space which is computationally intractable (also, see Figure 2).

As a simple benchmark, we use RANDOM-HP: a baseline policy that simply selects an action $a \in A_s$ randomly when providing a hint for state s. As another natural baseline, we consider FREQNEXT-HP which uses historical student data as follows. Based on the available data, a frequency count $\text{count}(s, a)$ is maintained for each (s, a) pair counting the number of times action a was taken from state s by students in the historical data. Then, when providing hint for a state s, the hint is chosen from a distribution given by the following softmax distribution: $P(a|s) = \frac{\exp\big(1+\text{count}(s,a)\big)}{\sum_{a' \in A_s} \exp\big(1+\text{count}(s,a')\big)}$.

Next we discuss a baseline based on the state of the art technique of *Continuous Hint Factory* (CHF) [20] that uses supervised learning approach. We adapt the key ideas of CHF to our setting and refer to the resulting hint policy as REGRESSION-HP—this adaption allows us to directly compare REINFORCE-HP with REGRESSION-HP as both these hint policies use the same embedding and same historical data when available. Below, we discuss three key steps required in training REGRESSION-HP:



(a) HOC18: Prediction accuracy with increasing data

(b) HOC18: Prediction accuracy based on frequency

(c) HOC4: Prediction accuracy with increasing data

(d) HOC4: Prediction accuracy based on frequency

Figure 4: **(a)** shows prediction accuracy for HOC-18 task. $x = 0$ on the x-axis corresponds to the zero-shot setting. REINFORCE-HP achieves over 20% absolute improvement in the prediction accuracy compared to baselines. **(b)** shows results for HOC-18 task when states are binned into "Head" and "Tail" based on frequency counts, and a moderate amount of historical student data is available (see details in Section 4.3). REINFORCE-HP performance on low-frequency states is even higher than the overall performance of any of the baselines. **(c, d)** shows the results for simpler task of HOC-4.

- *Embedding* (cf. Section 3.1 of [20]): For REGRESSION-HP, we use the same code embedding as used for REINFORCE-HP. Our code embedding is similar to the Euclidean embedding space used by [20] which was obtained by preserving syntactic distances between raw states.

- *Regression function* (cf. Section 3.2 of [20]): Then, we learn a regression function in the embedding space which can identify the most likely hint as a vector in this space. This step makes use of available student data D_{train} as discussed in Section 3.3, and learns a function f_{reg} that can map $\phi(s)$ to $\phi(s \oplus a)$ for $(s, s \oplus a) \in D_{\text{train}}$. We use neural network to learn this function f_{reg}, in contrast, [20] used Gaussian process regression. We use a 4-layer neural network to learn f_{reg} with essentially the same architecture as the one used to learn REINFORCE-HP, except that (i) the input layer has dim_ϕ units for $\phi(s)$, (ii) the output layer here has dim_ϕ units to produce $\phi(s \oplus a)$ (this corresponds to what was the last hidden layer in REINFORCE-HP neural architecture).

- *Human-readable hint* (cf. Section 3.3 of [20]): Finally, when providing hint for a state s, we first compute the hint in embedding space as $f_{\text{reg}}(\phi(s))$ and then convert this to an editable hint $a \in A_s$ as the one that minimizes $||f_{\text{reg}}(\phi(s)) - \phi(s \oplus a)||_2$.

4.3 Results

Figure 4 shows the results for HOC-18 and HOC-4 tasks, averaged over 3 runs of all the hint policies. Figures 4a and 4c show the overall average prediction accuracy. The $x = 0$ point in these plots corresponds to the zero-shot setting and measures the prediction accuracy of next-step hint for the very first student who is attempting the task. For both HOC-18 and HOC-4 tasks, our RL-based policy REINFORCE-HP has a significant improvement over baselines by about 20% gain in absolute accuracy. For the HOC-18 task, even when a moderate amount of data becomes available (e.g., see $x = 9$ on the plot which is equivalent to data of 511 students), REINFORCE-HP improves w.r.t. REGRESSION-HP by 10% gain in absolute accuracy.

In Figures 4b and 4d, we further analyze the performance of different hint policies when training using a moderate amount of available data (corresponding to $x = 9$ in Figures 4a and 4c which is equivalent to data of 511 students). In these plots, states are binned into "Head" and "Tail" based on frequency counts as available in the dataset obtained from [23]. Here, the bin "Head" corresponds to top 40 states and "Tail" corresponds to bottom 40 states based on frequency. For HOC-18 task, REINFORCE-HP performance on low frequency states is even higher than the overall performance of any of the baselines: this hightlights the power of RL framework that allows an efficient self-exploration of the solution space when learning the hint policy. These plots also illustrate that techniques such as FREQNEXT-HP that rely on frequency counts can have much worse performance on the tail segment of states compared to head segment of states.

In summary, these results demonstrate that the proposed RL framework enables us to learn an effective hint policy in the zero-shot setting, and the performance can be further improved with the availability of student data.

5. CONCLUSIONS AND FUTURE WORK

We tackled the challenge of zero-shot learning of hint policy to be able to provide hints for the very first student working on a coding task. Building on the recent advances in RL-based neural program synthesis, we proposed an RL framework for learning hint policy. Using a publicly available dataset from Code.org, we showed that our policy achieves significant improvements over state of the art supervised learning techniques when no or very limited data is available. Furthermore, the results demonstrated that our proposed framework is easily amendable, e.g., it can benefit from historical student data to further boost the performance.

There are several research directions for future work. As an evaluation criterion, we used the prediction accuracy of next-step hints based on expert annotations. In future work, it would be important to do user studies and understand the pedagogical value of these hints. In this work, our hint policy provided hints based on only the current partial solution of the student. It would be important to learn a richer hint policy that can provide personalized hints by accounting for the whole trajectory of the student. Finally, it would be interesting to apply our framework to more complex learning scenarios (e.g., with more complex coding tasks or with a more complex language involving additional concepts such as the ability to declare variables).

6. REFERENCES

[1] U. Z. Ahmed, S. Gulwani, and A. Karkare. Automatically generating problems and solutions for natural deduction. In *IJCAI*, 2013.

[2] V. Aleven, I. Roll, B. M. McLaren, and K. R. Koedinger. Help helps, but only so much: Research on help seeking with intelligent tutoring systems. *International Journal of Artificial Intelligence in Education*, 26(1):205–223, 2016.

[3] T. Barnes and J. Stamper. Toward automatic hint generation for logic proof tutoring using historical student data. In *International conference on intelligent tutoring systems*, pages 373–382. Springer, 2008.

[4] R. Bunel, M. J. Hausknecht, J. Devlin, R. Singh, and P. Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *ICLR*, 2018.

[5] Code.org. Code.org: Learn computer science. `https://code.org/research`.

[6] J. Devlin, R. Bunel, R. Singh, M. J. Hausknecht, and P. Kohli. Neural program meta-induction. In *NIPS*, pages 2080–2088, 2017.

[7] A. Ghosh, S. Tschiatschek, H. Mahdavi, and A. Singla. Towards deployment of robust cooperative ai agents: An algorithmic framework for learning adaptive policies. In *AAMAS*, 2020.

[8] S. Gross, B. Mokbel, B. Paassen, B. Hammer, and N. Pinkwart. Example-based feedback provision using structured solution spaces. *International Journal of Learning Technology 10*, 9(3):248–280, 2014.

[9] S. Gross and N. Pinkwart. How do learners behave in help-seeking when given a choice? In *International Conference on Artificial Intelligence in Education*, pages 600–603. Springer, 2015.

[10] S. Gulwani, O. Polozov, R. Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[11] R. Gupta, A. Kanade, and S. K. Shevade. Deep reinforcement learning for syntactic error repair in student programs. In *AAAI*, pages 930–937, 2019.

[12] B. Kartal, N. Sohre, and S. J. Guy. Data driven sokoban puzzle generation with monte carlo tree search. In *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2016.

[13] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

[14] J. Krajcik. Three-dimensional instruction: Using a new type of teaching in the science classroom. *Science Scope*, 39(3):16, 2015.

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

[16] T. Lazar and I. Bratko. Data-driven program synthesis for hint generation in programming tutors. In *International Conference on Intelligent Tutoring Systems*, pages 306–311. Springer, 2014.

[17] Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.

[18] J. Oh, S. Singh, H. Lee, and P. Kohli. Zero-shot task generalization with multi-task deep reinforcement learning. In *ICML*, pages 2661–2670, 2017.

[19] T. Osa, J. Pajarinen, G. Neumann, J. A. Bagnell, P. Abbeel, J. Peters, et al. An algorithmic perspective on imitation learning. *Foundations and Trends® in Robotics*, 7(1-2):1–179, 2018.

[20] B. Paaßen, B. Hammer, T. W. Price, T. Barnes, S. Gross, and N. Pinkwart. The continuous hint factory - providing hints in continuous and infinite spaces. *Journal of Educational Data Mining*, 2018.

[21] R. E. Pattis. *Karel the robot: a gentle introduction to the art of programming*. John Wiley & Sons, Inc., 1981.

[22] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. J. Guibas. Learning program embeddings to propagate feedback on student code. In *ICML*, pages 1093–1102, 2015.

[23] C. Piech, M. Sahami, J. Huang, and L. J. Guibas. Autonomously generating hints by inferring problem solving policies. In *Conference on Learning @ Scale, L@S*, pages 195–204, 2015.

[24] T. W. Price, Y. Dong, and D. Lipovac. isnap: towards intelligent tutoring in novice programming environments. In *SIGCSE*, pages 483–488, 2017.

[25] B. Priemer, K. Eilerts, A. Filler, N. Pinkwart, B. Rösken-Winter, R. Tiemann, and A. U. Zu Belzen. A framework to foster problem-solving in stem and computing education. *Research in Science & Technological Education*, pages 1–26, 2019.

[26] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., 1st edition, 1994.

[27] K. Rivers and K. R. Koedinger. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education*, 27(1):37–64, 2017.

[28] R. Singh, S. Gulwani, and S. Rajamani. Automatically generating algebra problems. In *AAAI*, 2012.

[29] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[30] K. S. Tai, R. Socher, and C. D. Manning. Improved semantic representations from tree-structured long short-term memory networks. In *ACL*, pages 1556–1566, 2015.

[31] K. Vanlehn. The behavior of tutoring systems. *International journal of artificial intelligence in education*, 16(3):227–265, 2006.

[32] V. K. Verma, G. Arora, A. Mishra, and P. Rai. Generalized zero-shot learning via synthesized examples. In *CVPR*, pages 4281–4289, 2018.

[33] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[34] M. Wu, M. Mosse, N. Goodman, and C. Piech. Zero shot learning for code education: Rubric sampling with deep learning inference. In *AAAI*, volume 33, pages 782–790, 2019.

[35] K. Zimmerman and C. R. Rupakheti. An automated framework for recommending program elements to novices (n). In *International Conference on Automated Software Engineering, ASE*, pages 283–288, 2015.