Latent Programmer: Discrete Latent Codes for Program Synthesis

Joey Hong[1] David Dohan[1] Rishabh Singh[1] Charles Sutton[1] Manzil Zaheer[1]

Abstract

A key problem in program synthesis is searching over the large space of possible programs. Human programmers might decide the high-level structure of the desired program before thinking about the details; motivated by this intuition, we consider two-level search for program synthesis, in which the synthesizer first generates a plan—a sequence of symbols that describes the desired program at a high level—before generating the program. We propose to learn representations of programs that can act as plans to organize such a two-level search. Discrete latent codes are appealing for this purpose, and can be learned by applying recent work on discrete autoencoders. Based on these insights, we introduce the *Latent Programmer* (LP), a program synthesis method that first predicts a discrete latent code from input/output examples, and then generates the program in the target language. We evaluate the LP on two domains, demonstrating that it yields an improvement in accuracy, especially on longer programs for which search is most difficult.

1. Introduction

Program synthesis is a longstanding grand challenge in artificial intelligence (Manna & Waldinger, 1971; Summers, 1977). The objective of program synthesis is to automatically write a program given a specification of its intended behavior, such as a natural language description or a small set of input-output examples (Alur et al., 2013; Gulwani et al., 2017). However, program synthesis requires solving a difficult search problem over a large space of possible programs. Search methods that have been explored include top-down search (Lee et al., 2018), bottom up search (Udupa et al., 2013; Odena et al., 2020; Barke et al., 2020), beam search (Devlin et al., 2017), and many others (see Section 2).

Our work is motivated by an intuition about the way people write programs. Synthesis methods often search through programs in an order determined by the token sequence, by a syntax tree, or by a logical solver. In contrast, we imagine that a programmer often starts by thinking about the high-level structure of the desired program — such as what library functions to call, or the overall program structure — and then fills in the details. For example, for a program that processes a list of people's names, a programmer might first plan that the program should extract the person's name followed by the person's last initial, and only then think about details such as which library functions to use.

Based on this intuition, we propose *two-level search* for program synthesis. In two-level search, the synthesizer first produces a plan that describes the desired program, and then synthesizes a program based on that plan. For us, a *plan* is simply a sequence of symbols that describes the code to be generated at a high level, without specifying the syntactic and semantic details. The goal is that plans can provide a way to organize search over programs. In the name-processing example, suppose that an initial plan incorrectly specifies to extract the first instead of the last initial. Searching in plan space could easily find the small change required to yield the correct plan, even if this would correspond to a large change in the program. This allows two-level search to explore a more diverse set of programs, improving the chance of finding the correct one.

A key design decision is defining the space of possible plans. For example, sketches could be used as plans (Nye et al., 2019; Murali et al., 2018). In this work, we explore whether it is possible to learn a representation of programs that is useful for constructing plans that guide search. Instead of having a deterministic heuristic for mapping programs to plans, we let a model discover what plans are useful for representing programs, and how to infer them from the specification. To tackle this problem, we make use of recent work in learning discrete unsupervised representations (van den Oord et al., 2017; Roy et al., 2018; Kaiser et al., 2018). These are self-supervised methods that given a dataset, assign each data item to a *discrete latent code* (a sequence of symbols from an arbitrary set) in such a way that the latent code provides a good description of the data item. The main hypothesis of our work is that discrete latent codes can be used as plans for two-level search.

[*] Equal contribution [1] Google Research, Mountain View, CA, USA. Correspondence to: Joey Hong <jxihong@google.com>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021. Copyright 2021 by the author(s).

This leads us to propose the *Latent Programmer*, a program synthesis method that employs two-level beam search, where the plans are based on discrete latent codes. At training time, a discrete autoencoder based on Kaiser et al. (2018) is used to train three models: one that infers ground-truth discrete latent codes to describe the programs in the training set, one that maps specifications to plans (i.e. discrete latent codes), and one that maps plans to programs. At inference time, Latent Programmer uses a two level beam search, first producing an L-best list of plans, then producing a B/L-best list of programs for each plan. On two different program synthesis domains, we find empirically that the Latent Programmer improves synthesis accuracy by over 10% compared to several baseline synthesis methods, especially on longer programs that are more difficult for search.

2. Background

The goal in program synthesis is to find a program in a given language that is consistent with a specification. Formally, we are given a domain specific language (DSL) which defines a space $\mathcal{Y}$ of programs. The task is described by a specification $X \in \mathcal{X}$ and is solved by an unknown program $Y \in \mathcal{Y}$. For example, each specification can be a set of input/output (I/O) examples denoted $X = \{(I_1, O_1), \dots (I_N, O_N)\}$. Then, we have solved specification X if we found a program Y' (not necessarily Y) which correctly solves all the examples: $Y'(I_i) = O_i, \forall i$. As another example, each specification can be a natural language description of a task, and the corresponding program implements said task. An example synthesis task in the string transformation DSL is shown in Figure 1.

Vector Quantization Our method relies on *discrete autoencoders*, which are unsupervised learning methods that assign each data point to a sequence of symbols, called a *discrete latent code*, in such a way that the code is a good description of the data item. In this section, we describe Vector Quantized Variational Autoencoders (VQ-VAE) (van den Oord et al., 2017; Roy et al., 2018). This will introduce ideas that we use later in the discrete autoencoder for Latent Programmer (Section 3.3). In a VQ-VAE, latent codes are sequences drawn from a discrete set of tokens $\mathcal{T}$ of size $|\mathcal{T}| = K$. Each token with id $k \in [K]$ is associated with a learned embedding $c_k \in \mathbb{R}^D$; these embeddings can be stacked into a matrix $c \in \mathbb{R}^{K \times D}$ called a *codebook*. To generate a code for a data item x, first the data point is passed through a neural network $\mathrm{ec}_\phi(x)$ called an encoder, and the encoder output e is quantized via nearest-neighbor lookup into the codebook. Formally, the quantized token id $\mathrm{qk}(e)$ and embedding $\mathrm{qc}(e)$ are

$$\mathrm{qc}(e) = c_{\mathrm{qk}(e)} \text{ where } \mathrm{qk}(e) = \arg\min_{k \in [K]} ||e - c_k||_2. \quad (1)$$

For input x, the training loss for a VQ-VAE has three terms: a reconstruction loss, a codebook loss that encourages codebook embeddings to be close to their associated encoded inputs $\mathrm{ec}(x)$, and a commitment loss that encourages the encoded input $\mathrm{ec}(x)$ to "commit" to codes i.e. to be close to the discrete code it is quantized to. In total, the loss is

$$\begin{aligned}\mathcal{L}(c, \theta, \phi) = \log p_\theta\left(x \mid \mathrm{qc}(\mathrm{ec}_\phi(x))\right) + ||\mathrm{sg}(\mathrm{ec}_\phi(x)) - c||_2^2 \\ + \beta||\mathrm{sg}(c) - \mathrm{ec}_\phi(x)||_2^2, \quad (2)\end{aligned}$$

where θ, ϕ are the parameters of the decoder and encoder, respectively, $\mathrm{sg}(\cdot)$ is the stop gradient operator that fixes the operand from being updated by gradients, and β controls the strength of the commitment loss. To stabilize training, van den Oord et al. (2017) also proposed removing the codebook loss and set the codebook to an exponential moving average (EMA) of encoded inputs.

Algorithm 1 Program synthesis using two-level search

```
Input: Specification $X$, search functions $S_0, S_1$,
       objective functions $f, g$
plan ← $S_0(X, f)$
$Y' \leftarrow S_1(\mathsf{plan}, X, g)$
return $Y'$
```

3. Synthesis with Discrete Latent Variables

Latent Programmer is an instance of a general framework for two-level search in program synthesis (Section 3.1). After presenting the general framework, we describe the specific architecture (Section 3.2), training objective (Section 3.3), and search method (Section 3.4) used in Latent Programmer.

3.1. Two-level Search

Our approach is based on the generic framework for defining program synthesizers using two-level search in Algorithm 1. This framework is agnostic to the search algorithm and DSL used. The idea is that the algorithm generates a *plan*, which intuitively, provides a high-level, coarse-grained description of a program to organize the search procedure. For example, in string editing, a token in a plan might indicate that the program should extract the first numeric substring. Formally, a plan is simply a sequence of tokens, each drawn from a finite set $\mathcal{T}$ of size $|\mathcal{T}| = K$. We denote tokens in $\mathcal{T}$ as `TOK_1`, `TOK_2`, . . ., `TOK_K`.

To define a concrete synthesizer in this framework, we need to specify concrete choices for the search algorithms S_0 and S_1 and objective functions f and g that should be used. The first-level search function S_0 returns $\mathsf{plan} \in \mathcal{T}^S$ that approximately maximizes the objective $f(\mathsf{plan}, X) \in \mathbb{R}$. Then, the second-level search function S_1 returns a program $Y' \in \mathcal{Y}$ using plan that maximizes $g(Y', \mathsf{plan}, X) \in \mathbb{R}$.

Several previous synthesis methods can be seen as examples of this general framework. For example, SKETCHADAPT

Inputs	Outputs	Program
“Mason Smith"	“Smith M"	`GetToken_PROP_CASE_2 | Const(" ") | GetToken_ALL_CAPS_1`
“Henry Myers"	“Myers H"	
“Barry Underwood"	“Underwood B"	
“Sandy Jones"	“Jones S"	

Figure 1. A string transformation task with four input-output examples and a program in the DSL that is consistent with the examples.

(Nye et al., 2019) can be viewed as an instantiation of this framework where the plans are program sketches, that is, partial programs in which certain subtrees are replaced by a special HOLE token. Also, BAYOU (Murali et al., 2018) can be viewed as another instantiation where the plan is a sketch that abstracts expressions and function calls by their types. Our Latent Programmer approach is a new instantiation of this framework, described next.

3.2. Architecture

Our synthesizer, *Latent Programmer* (LP), is a two-level synthesizer that learns representations of plans using a discrete autoencoder. Because programs are often modular, with components that are reused across tasks, LP is based on the hypothesis that this compositional structure can be leveraged by learning plans as discrete latent codes. We use neural networks to define distributions over both plans and programs, which are then used within Algorithm 1 through having f and g be the log-probabilities defined by those networks. In this section, we describe the architecture of LP at a high-level, deferring details to Appendix B.

Our proposed system consists of three main components: a *latent predictor*, *latent program decoder*, and *program encoder*. Components are parameterized as Transformers, which we use instead of RNNs due to their impressive performance on natural language tasks (Vaswani et al., 2017).

The pipeline of our LP model is summarized in Figure 2, and an end-to-end example is shown in Figure 4. The *latent predictor* $\mathrm{lp}(X)$ predicts a distribution over latent codes $\mathrm{lp}(X) \in \mathbb{R}^{S\times K}$ conditioned on the program specification X. The *latent program decoder* $d(Z,X)$ defines a distribution over programs, and is jointly conditioned on specification X and latent code $Z \in \mathbb{R}^{S\times K}$. The *program encoder* is only used during training and learns useful meanings for the latent tokens in the code. The program encoder $\mathrm{ec}(Y)$ encodes the true program $Y = [y_1, y_2, \dots, y_T]$ into a discrete latent code $Z = [z_1, z_2, \dots, z_S]$, where each $z_i \in \mathcal{T}$. This latent code will then serve as the ground-truth plan for Y, as described in the next section. In this work we let $S = \lceil T/2^\ell \rceil$, where ℓ is the *latent length compression factor* and is tuned during training. This provides temporal abstraction, where the high-level latent tokens roughly map to 2^ℓ program tokens. We emphasize that the program encoder is only used in training. At test time, $\mathrm{lp}(X)$ is used instead of $\mathrm{ec}(Y)$; the latent predictor is unaware of what S is and autoregressively generates latent tokens until an end-of-sequence token is reached.

3.3. Training

To learn the plan representations, we use discrete latent codes from an autoencoder, based on the work of Kaiser et al. (2018) in natural language, which combines a VQ-VAE with a sequence-to-sequence learning objective. The loss function has three parts.

First, the *autoencoder loss* ensures that the latent codes contain information about the program, and that the latent program decoder can recover the true program given the specification and the latent code. This loss is similar to the loss function of a VQ-VAE as in equation 2, but also depends on a specification X. Like in Roy et al. (2018), the codebook is not trained but set to the EMA of encoder outputs. Second, the *latent prediction loss* ensures that latent codes can be predicted from specifications. This loss treats the discrete latent sequence $\mathrm{qk}(\mathrm{ec}(Y))$ of the true program as the ground-truth plan, and trains the latent predictor $\mathrm{lp}(X)$ to generate it using just the program specification X. Finally, the *end-to-end loss* ensures that programs can be predicted from specifications. This is needed because when computing the autoencoder loss, the latent code arises from encoding the correct program $\mathrm{ec}(Y)$, but at test time, we have only the specification X. This can result in mistakes in the generated program since the decoder has never been exposed to noisy results from the latent predictor. The end-to-end loss alleviates this issue. To make this differentiable, the end-to-end loss is probability of the correct program Y when predicted from a soft-quantized latent code, given by $\mathrm{lp}(X)^T c$. In summary, the full loss for a training instance is

$$\begin{aligned} &\mathcal{L}(c, \theta, \phi, \psi) \qquad (3)\\ &= \underbrace{\log p_\theta\left(Y \mid \mathrm{qc}(\mathrm{ec}_\phi(Y)), X\right) + \beta||\mathrm{sg}(c) - \mathrm{ec}_\phi(Y)||_2^2}_{\text{autoencoder}} \\ &+ \underbrace{\log p\left(\mathrm{qk}(\mathrm{ec}_\phi(Y)) \mid \mathrm{lp}_\psi(X)\right)}_{\text{latent prediction}} + \underbrace{\log p_\theta\left(Y \mid \mathrm{lp}_\psi(X)^T c, X\right)}_{\text{end-to-end}} \end{aligned}$$

where we explicitly list out θ, ϕ, and ψ representing the parameters of the latent program decoder, program encoder, and latent program decoder respectively.

Finally, for the first 10K steps of training, we give embed-

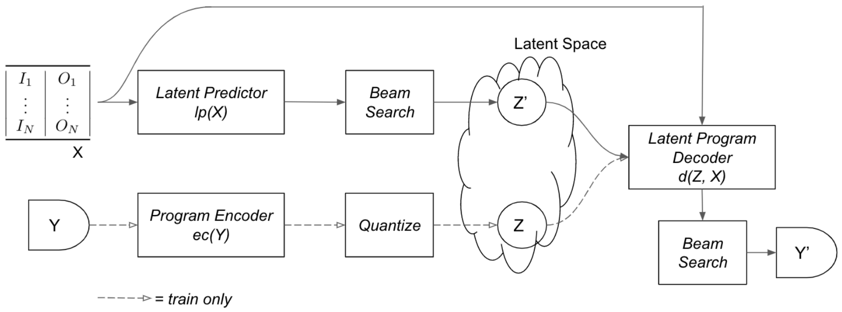

Figure 2. High-level architecture for the Latent Programmer system. The latent predictor generates probabilities over latent sequences, which can be decoded into a predicted latent sequence Z'. Z' is fitted to a ground-truth latent sequence Z generated by a program encoder, and used during decoding to by the latent program decoder to generate programs.

dings of the ground-truth program Y, averaged over every 2^ℓ tokens, as the latent code instead of $\mathrm{ec}(Y)$. This pre-training ensures that when we start training on the full objective, the latent code already contains information about the program that can aid in training the latent program decoder. We found empirically that this prevented the bypassing phenomenon where the latent code is ignored by the decoder (Bahuleyan et al., 2017).

3.4. Two Level Beam Search

During inference, we use two-level beam search, i.e., in Algorithm 1, both S_0, S_1 are beam search, f is the log probability from the latent predictor, and g the log probability from the latent program decoder. Standard beam search returns the top-B most likely programs according to the model, from which we return the first one (if any) that is consistent with the specification (Parisotto et al., 2017; Devlin et al., 2017). In our case, S_0 performs beam search to return L sequences of discrete latent codes, then S_1 returns $\lfloor B/L \rfloor$ programs for each latent sequence. During inference, the latent predictor will continue to generate latent tokens until an end-of-sequence token is produced, so the generated latent sequence does not necessarily have length $\lceil T/2^\ell \rceil$ as during training; however, we found the latent sequence lengths during training and evaluation to be close in practice. Setting $L = B$ allows for the maximum exploration of the latent space, while setting $L = 1$ reduces our method to standard beam search, or exploitation of the most likely latent decoding. We choose $L = \sqrt{B}$ in our experiments, but explore the effect of L in Section 5.2.

4. Related Work

Program Synthesis Our work deals with *program synthesis*, which involves combinatorial search for programs that match a specification. Many different search methods have been explored within program synthesis, including search within a version-space algebra (Gulwani, 2011), bottom-up enumerative search (Udupa et al., 2013), stochastic search (Schkufza et al., 2013), genetic programming (Koza, 1994), or reducing the synthesis problem to logical satisfiability (Solar-Lezama et al., 2006). *Neural program synthesis* involves learning neural networks to predict function distributions to guide a synthesizer (Balog et al., 2017), or the program autoregressively in an end-to-end fashion (Parisotto et al., 2017; Devlin et al., 2017). SKETCHADAPT (Nye et al., 2019) combined these approaches by first generating a program sketch with holes, and then filling holes using a conventional synthesizer. BAYOU (Murali et al., 2018) trained on a different form of program sketches that abstracted names and operations by their type. DreamCoder (Ellis et al., 2020) iteratively built sketches using progressively more complicated primitives though a wake-sleep algorithm. Our work is closely related in spirit but fundamentally differs in two ways: (1) our sketches are comprised of a general latent vocabulary that is learned in a simple, self-supervised fashion, and (2) our method avoids enumerative search, which is prohibitively expensive for large program spaces. Another related avenue of research is using idiom mining to learn high-level concepts of a program (Shin et al., 2019; Iyer et al., 2019). However, the idioms considered are always based on syntactic structure, i.e. subgraphs of the AST of the program, whereas tokens of our latent codes need not be so localized; also, idioms are extracted by a preprocessing step, whereas our training method learns the semantics of the latent tokens end-to-end. Finally, there is a line of work that deals with learning to process partial programs in addition to the specification. In *execution-guided program synthesis*, the model guides iterative extensions of the partial programs until a matching one is found (Zohar & Wolf, 2018; Chen et al., 2019; Ellis et al., 2019). Balog et al. (2020) proposed training a differentiable fixer to edit incorrect programs. We treat these works as complementary, and can be combined with ours to refine predictions.

Discrete Autoencoders. Variational autoencoders (VAE) were first introduced using continuous latent representations (Kingma & Welling, 2014; Rezende et al., 2014). Several approaches were proposed to use discrete latent codes instead, such as continuous relaxations of categorical distributions i.e. the Gumbel-Softmax reparametrization trick (Jang et al., 2017; Maddison et al., 2017). VQ-VAEs (see Figure 2 for more details) achieved impressive results almost matching continuous VAEs (van den Oord et al., 2017; Roy et al., 2018). In natural language processing, discrete bottlenecks have also been used for sentence compression (Miao & Blunsom, 2016) and text generation (Puduppully et al., 2019), but these works do not use an autoencoder to learn the semantics of the latent codes, like our work does. Within the domain of synthesis of chemical molecules, (Gómez-Bombarelli et al., 2018) have applied Bayesian optimization within a continuous latent space to guide this structured prediction problem. Learning to search has also been considered in the structured prediction literature (Daumé et al., 2009; Chang et al., 2015; Ross et al., 2011), but to our knowledge, these works do not consider the problem of learning a discrete representation for search. Notably, VQ-VAE methods have been successfully used to encode natural language into discrete codes for faster decoding in machine translation (Kaiser et al., 2018). The key novelty behind our work is in proposing two-level search over a learned latent discrete space; using a VQ-VAE as Kaiser et al. (2018) did enabled us to do so.

5. Experiments

We now present the results of evaluating our Latent Programmer model in two test domains: synthesis of string transformation programs from examples and code generation from natural language descriptions. We compare our LP model against several strong baselines.

RobustFill [LSTM] is a seq-to-seq LSTM with attention on the input specification, and trained to autoregressively predict the true program. The architecture is comparable to the RobustFill model designed originally for the string transformation tasks in our first domain (Devlin et al., 2017), but easily generalizes to all program synthesis domains. We detail the architecture in Appendix A.

RobustFill [Transformer] alternatively uses a Transformer architecture, equivalent in architecture to the latent planner in our LP model, also trained to autoregressively predict the program. Transformers were found to perform much better than LSTMs in language tasks because they process the entire input as a whole, and have no risk of forgetting past dependencies (Vaswani et al., 2017). This baseline can be also be considered an ablation of LP without latent codes.

To test the hypothesis that the discrete latent code is helping to organize search, rather than simply increasing the capacity of the model, we compare to two ablations that use continuous autoencoders rather than discrete ones. Because for these methods the latent space is continuous, combinatorial search algorithms such as beam search cannot search over the latent space.

Method	Accuracy		
	B = 1	10	100
RobustFill [LSTM]	45%	49%	61%
RobustFill [Transformer]	47%	51%	61%
Latent RobustFill [AE]	47%	50%	60%
Latent RobustFill [VAE]	46%	51%	62%
Latent Programmer	**51**%	**57**%	**68**%

Table 1. Accuracy on string transformation domain.

Latent RobustFill [AE] replaces the VQ-VAE component of our LP model with a generic autoencoder. This makes the latent code a sequence of continuous embeddings. The latent prediction loss in equation 3 is simply replaced by a squared error between the output of the autoencoder and the latent predictor. Performing beam search over the continuous latent space is intractable, so during inference we generate only one latent code per task; this is equivalent to two-level beam search described earlier with $L = 1$. In addition, because we cannot define an end-of-sequence token in the latent space, this baseline must be given knowledge of the true program length even during inference, and always generates a latent code of length $\lceil T/2^\ell \rceil$.

Latent RobustFill [VAE] substitutes the VQ-VAE component with a VAE (Kingma & Welling, 2014). This again produces a continuous latent space, but regularized to be distributed approximately as a standard Gaussian. Performing beam search is still intractable, but we can sample L latent codes from the output of the VAE, and perform beam search on the programs afterwards. Again, we assume that the true program length is known during inference.

5.1. String Transformation

The first test domain is a string transformation DSL frequently studied in the program synthesis literature (Parisotto et al., 2017; Devlin et al., 2017; Balog et al., 2020). Tasks in this domain involve finding a program which maps a set of input strings to a corresponding set of outputs. Programs in the DSL are a concatenation of expressions that perform regex-based string transformations (see Appendix A).

We perform experiments on a synthetic dataset generated by sampling programs from the DSL, then the corresponding I/O examples using an heuristic similar to the one used in NSPS (Parisotto et al., 2017) and RobustFill (Devlin et al., 2017) to ensure nonempty output for each input. We consider programs comprising of a concatenation of up to 10 expressions and limit the lengths of strings in the I/O to be at most 100 characters. All models have an embedding size of 128 and hidden size of 512, and the attention layers consist of 3 stacked layers with 4 heads each. For the LP model,

Method	Accuracy
DeepCoder (Balog et al., 2017)	40%
SketchAdapt (Nye et al., 2019)	62%
Latent Programmer	**67%**

Table 2. Accuracy on string transformation domain of Nye et al. (2019) using $B = 100$. SKETCHADAPT and DEEPCODER results are from Nye et al. (2019) using $3,000$ and $300,000$ synthesized programs, respectively (similar wall clock time).

we used a latent compression factor $\ell = 2$ and vocabulary size $K = 40$. The models are trained on roughly 25M tasks, and evaluated on 1K held-out ones.

In Table 1, we report the accuracy — the number of times a program was found conforming to the I/O examples — of our method against the baselines. Across all beam sizes, our LP model performed 5-7 percentage points better (over 10% of baseline accuracy) than the next best model. From our ablative study, we see that having two-level using discrete latent codes was important, as the baselines over continuous latent spaces performed comparably to baseline RobustFill.

SketchAdapt As alluded to earlier, two-level search was also proposed by Nye et al. (2019) as SKETCHADAPT, which learned programs with a `HOLE` token, then filled in the holes using enumerative search. To compare our proposed method with SKETCHADAPT, we evaluate our LP model on samples generated according to Nye et al. (2019), which slightly modifies the DSL to improve the performance of synthesizers. We report results in Table 2. Since enumeration can be done more quickly than beam search, we let SKETCHADAPT synthesize $3,000$ programs using $B = 100$ top-level beams, whereas our LP model can only generate B programs. We also reported results for DEEPCODER (Balog et al., 2017), which synthesizes $300,000$ programs without the high-level beam search. We chose the number of synthesized programs so that all methods have similar wall clock time. Our LP model is able to outperform both methods in this modified DSL.

5.2. Analysis

We conduct extensive analysis to better understand our LP model, the ability to generate long programs, and diversity in the beams. All results are reported with beam size $B = 10$.

Model Size Our LP model uses an additional latent code for decoding, which introduces additional parameters into the model than the baseline RobustFill model. To make a fair comparison, we vary the embedding and hidden dimension of all of our evaluated methods, and compare the effect of the number of trainable parameters on the accuracy. Figure 3 shows that all methods respond well to an increase in model size. Nevertheless, we see that even when normalized for size, our LP model significantly outperforms baselines.

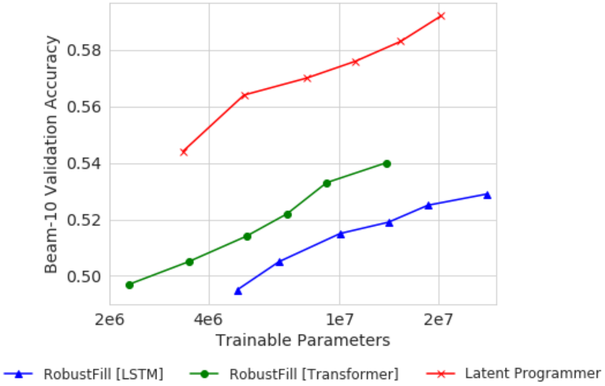

Figure 3. Influence of hidden size on beam-10 accuracy.

Length	RobustFill Acc.	LP Acc.
1	**94.5**%	94.0%
2	83.9%	**84.6**%
3	**72.8**%	72.2%
4	63.1%	**66.1**%
5	47.1%	**49.8**%
6	40.6%	**43.0**%
7	30.2%	**34.6**%
8	22.7%	**28.4**%
9	18.6%	**27.0**%
10	14.4%	**25.6**%

Table 3. Beam-10 accuracy of baseline transformer and LP by ground truth program length

Program Length Prior work has shown that program length is a reasonable proxy measure of problem difficulty. We hypothesize that using latent codes is most beneficial when generating long programs. Table 3 shows how ground-truth program length affects the accuracy of our LP model compared to RobustFill, which lacks latent codes. As expected, accuracy decreases with problem complexity. Perhaps surprisingly, though, we see a large improvement in our LP model's ability to handle more complex problems. This supports our hypothesis two-level search can organize and improve search over more complex tasks, because we see a greater improvement in accuracy precisely for the examples in which traditional search is most difficult.

Latent Beam Size In two-level beam search of beam size B, first L latent beams are decoded, then $\lfloor B/L \rfloor$ programs per latent code. The latent beam size L controls how much search is performed over latent space. We theorize that higher L will produce more diverse beams; however, too high L can be harmful in missing programs with high joint log-probability. We show the effect of latent beam size on both the beam-10 accuracy and a proxy measure for diversity. Diversity is important to measure because increased di-

<table>
<tr><th>Inputs</th><th>Outputs</th><th>Program</th></tr>
<tr><td>“Jacob,Ethan,James 11"</td><td>“11:J.E.J."</td><td><code>GetToken_NUMBER_1 | Const(:) |</code></td></tr>
<tr><td>“Elijah,Daniel,Aiden 3162"</td><td>“3162:E.D.A"</td><td><code>GetToken_ALL_CAPS_1 | Const(.) |</code></td></tr>
<tr><td>“Rick,Oliver,Mia 26"</td><td>“26:R.O.M."</td><td><code>GetToken_ALL_CAPS_2 | Const(.) |</code></td></tr>
<tr><td>“Mark,Ben,Sam 510"</td><td>“510:M.B.S."</td><td><code>GetToken_ALL_CAPS_3 | Const(.)</code></td></tr>
</table>

<table>
<tr><td>RobustFill</td><td><code>GetAll_NUMBER | Const(:)| GetToken_ALL_CAPS_2 | Const(.)</code></td></tr>
<tr><td>LP</td><td><code>GetAll_NUMBER | Const(:) | GetToken_ALL_CAPS_1 | Const(.) |</code>
<code>GetToken_ALL_CAPS_2 | Const(.) | GetToken_ALL_CAPS_-1 | Const(.)</code></td></tr>
<tr><td>LP Latent</td><td><code>TOK_14 | TOK_36 | TOK_36 | TOK_36</code></td></tr>
</table>

Figure 4. Illustrative string transformation problem where the ground-truth program was long but had repetitive structure. The baseline Transformer was unable to generate the program but our LP model, which first predicts a coarse latent code, was able to.

Beam Size	Accuracy	Distinct n-Grams			
		n = 1	2	3	4
L = 1	52%	0.13	0.23	0.26	0.28
2	55%	0.13	0.24	0.26	0.28
3	**57**%	0.14	0.25	0.28	0.31
5	57%	0.14	0.26	0.29	0.32
10	56%	**0.14**	**0.26**	**0.30**	**0.33**

Table 4. Effect of latent beam size on beam-10 accuracy and number of distinct n-grams (normalized by total number of tokens).

versity suggests that two-level search is better exploring the space of possible programs. Following prior work, we measure diversity by counting the number of distinct n-grams in the beams, normalized by the total number of tokens to bias against long programs (Vijayakumar et al., 2018). We report the results varying L for $B = 10$ in Table 4. As expected, increasing the latent beam size L improves diversity of output programs, but excessively large L harms the final accuracy. An important observation is that the $L = 1$ case effectively corresponds to single-level search, and performs similarly to baseline RobustFill. This is further evidence that explicitly having two-level search is critical to the LP model's improved performance.

2^ℓ	Accuracy
2	52%
4	**55**%
8	49%

K	Accuracy
10	48%
40	**55**%
100	51%

Figure 5. Effect of ℓ, K.

Latent Code Dimension We also measured the effect of the expressiveness of our latent code, specifically by varying the latent length compression factor ℓ, and size of latent vocabulary K, on overall performance. If c is too small, the latent space becomes too large to search; on the other hard, too large c can mean individual latent tokens cannot encode enough information to reconstruct the program. Similarly, we expect that too small of a vocabulary K can limit the expressiveness of the latent space, but too large K can make the latent space too complex, and predicting the correct latent code difficult. Figure 5 confirms this.

Latent Interpretability A key hypothesis of our work is that searching over latent codes organizes search over programs; it is crucial that the latent codes be informative of the synthesized program. In Figure 4, we also show an illustrative example in the domain where our LP model found a valid program whereas the RobustFill model did not (more examples are in Appendix D). In the example, the ground-truth program was long but had a repetitive underlying structure. Our LP model correctly detected this structure, as evidenced by the predicted latent code. However, due to the complexity of our DSL and size of latent space, it is difficult to find explicit meaning behind individual tokens.

Thus, to better investigate interpretability, we created a toy DSL using only the `GetSpan` expression from the RobustFill DSL. This expression allows us to grab arbitrary ranges defined by a regex and its index of appearance, so sufficiently complex programs can still be generated (see Appendix C for full DSL). We trained a LP model with $\ell = 2$ and $K = 10$ on the toy DSL, and recorded examples of predicted programs and their corresponding latent codes in Appendix C. From these examples, we can see a pattern of LP associating particular latent tokens with high-level operations. For example, `TOK_7` and `TOK_4` were extracting the first and last number in the string, and `TOK_6`, `TOK_3` the first and last word. As further evidence, in Figure 6, we chose six high-level operations and recorded the percentage of times each was mapped to a specific latent token. Specific high-level operations were clearly biased towards particular latent tokens, further suggesting that the latent codes were specifying high-level components of the program. In addition, since there are multiple syntactic ways of expressing the same operation, latent tokens were more likely capturing high-level semantics over syntax.

	TOK_3	TOK_4	TOK_5	TOK_6	TOK_7	TOK_8	TOK_9
Get First Number	12%	5%	0%	9%	70%	6%	0%
Get Last Number	22%	49%	0%	11%	8%	8%	0%
Get First Word	10%	20%	0%	56%	7%	9%	0%
Get Last Word	75%	4%	0%	6%	9%	6%	0%
Get First Alphanum	11%	3%	0%	35%	42%	9%	0%
Get Last Alphanum	45%	29%	0%	22%	0%	4%	0%

Figure 6. Percentage of time each high-level operation was associated with a particular latent token on toy DSL. Note that tokens $0, 1, 2$ are reserved for padding, and start and end of sequences, respectively.

Method	BLEU		
	B = 1	10	100
Base (Wei et al., 2019)	10.4	-	-
Dual (Wei et al., 2019)	12.1	-	-
RobustFill [LSTM]	11.4	14.8	16.0
RobustFill [Transformer]	12.1	15.5	17.2
Latent Programmer	**14.0**	**18.6**	**21.3**

Table 5. BLEU score on code generation task.

5.3. Python Code Generation

Our next test domain is a Python code generation (CG) task, which involves generating code for a function that implements a natural-language specification. The dataset used consists of 111K python examples, which consist of a docstring and corresponding code snippet, collected from Github (Wan et al., 2018). An example docstring and program from the dataset is shown in Figure 7.

We used a language-independent tokenizer jointly on data (Kudo & Richardson, 2018), and processed the dataset into a vocabulary of 35K sub-word tokens. Furthermore, following Wei et al. (2019), we set the maximum length of the programs to be 150 tokens resulting in 85K examples. Across all models, we set the embedding size to be 256 and hidden size to be 512, and the attention layers consist of 6 stacked layers with 16 heads each, similar to in neural machine translation (Vaswani et al., 2017). For the LP model, we used a latent compression factor $c = 2$ and vocabulary size $K = 400$ after a hyperparameter search. The models are evaluated on 1K held-out examples. We initially found that it was difficult for the program encoder to find a latent structure in the ground-truth programs due to the wide variety of variable names. To remedy this, we replace the i-th function argument and variable appearing the program with the token `ARG_i` and `VAR_i`, respectively. This was only used in training the program encoder.

In this domain, we are not given I/O examples as specification. In addition, the programs in the dataset are often not executable due missing dependencies, or having complex objects as arguments. Hence, we cannot measure accuracy by evaluating programs on test cases as before. Instead, we evaluate performance by computing the best BLEU score among the output beams (Papineni et al., 2002). This is a natural metric, as we can imagine that in practice, a user would examine the candidate programs to select one that best matches their intent. We computed BLEU as the geometric mean of n-gram matching precision scores up to $n = 4$. Table 5 shows that our LP model outperforms the baselines. From the results, it can be seen that this is a difficult task, which may be due to the ambiguity in specifying code from a short docstring description. As evidence, we additionally include results from a recent work that proposed seq-to-seq CG models on the same data that performed similar to our baselines (Wei et al., 2019). These results show that improvements due to the LP model exist even in difficult CG domains. For example docstrings and generated code, refer to Appendix D.

Finally, we investigated interpretability in this domain. For each latent token, we collected the set of programs associated with that token, and for each of those sets, we ranked the program tokens by TF-IDF (Salton & McGill, 1986). In Figure 8, we list several latent tokens where the top-5 program tokens have a common semantic interpretation. For example, the first one seems to exhibit a latent state learning high-level concepts about file manipulation. However, due to the scale and noisiness of the dataset, it was difficult to see strong semantic clustering among all latent tokens.

6. Conclusion

In this work we proposed the Latent Programmer (LP), a novel neural program synthesis technique that leverages a structured latent sequences to guide search. The LP model consists of a latent predictor, which maps the input specification to a sequence of discrete latent variables, and a latent program decoder that generates a program token-by-token while attending to the latent sequence. The latent predictor was trained via a self-supervised method in which a discrete autoencoder of programs was learned using a discrete bottleneck, specifically a VQ-VAE (van den Oord et al., 2017),

Docstring	Program

return a list of the words in the string s	`def split(s, sep=None, maxsplit=-1):`
	`  return s.split(sep, maxsplit)`

Figure 7. Example problem from the Python code generation dataset.

0	_files	dirname	glob	isdir	makedir
1	server	_port	_socket	_password	host
2	pip	package	wheel	install	sudo
3	dt	interval	seconds	time	timestamp
4	timeout	_timeout	handle	future	notifier

Figure 8. Example latent tokens and top-5 program tokens ranked by TF-IDF score.

and the latent predictor tries to predict the autoencoded sequence as if it were the ground-truth. During inference, the LP model first searches in latent space for discrete codes, then conditions on those codes to search over programs. Empirically, we showed that the Latent Programmer outperforms state-of-the-art baselines as Robustfill (Devlin et al., 2017), which ignore latent structure. Exciting future avenues of investigation include achieving better performance by grounding the latent vocabulary and generalizing our method to other tasks in structured prediction.

References

Alur, R., Bodík, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pp. 1–8. IEEE, 2013.

Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations (ICLR)*, 2016.

Bahuleyan, H., Mou, L., Vechtomova, O., and Poupart, P. Variational attention for sequence-to-sequence models. *CoRR*, abs/1712.08207, 2017.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. Deepcoder: Learning to write programs. In *International Conference on Learning Representations (ICLR)*, 2017.

Balog, M., Singh, R., Maniatis, P., and Sutton, C. Neural program synthesis with a differentiable fixer. *CoRR*, abs/2006.10924, 2020. URL https://arxiv.org/abs/2006.10924.

Barke, S., Peleg, H., and Polikarpova, N. Just-in-Time learning for Bottom-Up enumerative synthesis. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2020.

Chang, K.-W., Krishnamurthy, A., Agarwal, A., Daume III, and Langford, J. Learning to search better than your teacher. In *International Conference on Machine Learning (ICML)*, 2015.

Chen, X., Liu, C., and Song, D. Execution-guided neural program synthesis. In *International Conference on Learning Representations (ICLR)*, 2019.

Daumé, III, H., Langford, J., and Marcu, D. Search-based structured prediction. *Machine Learning Journal*, 2009.

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A., and Kohli, P. Robustfill: Neural program learning under noisy I/O. *CoRR*, abs/1703.07469, 2017. URL http://arxiv.org/abs/1703.07469.

Ellis, K., Nye, M. I., Pu, Y., Sosa, F., Tenenbaum, J., and Solar-Lezama, A. Write, execute, assess: Program synthesis with a REPL. In *Neural Information Processing Systems (NeurIPS)*, 2019.

Ellis, K., Wong, C., Nye, M., Sable-Meyer, M., Cary, L., Morales, L., Hewitt, L., Solar-Lezama, A., and Tenenbaum, J. B. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *CoRR*, abs/2006.08381, 2020. URL https://arxiv.org/abs/2006.08381.

Gómez-Bombarelli, R., Wei, J. N., Duvenaud, D., Hernández-Lobato, J. M., Sánchez-Lengeling, B., Sheberla, D., Aguilera-Iparraguirre, J., Hirzel, T. D., Adams, R. P., and Aspuru-Guzik, A. Automatic chemical design using a Data-Driven continuous representation of molecules. *ACS Cent Sci*, 4(2):268–276, February 2018.

Gulwani, S. Automating string processing in spreadsheets using input-output examples. In *PoPL'11, January 26-28, 2011, Austin, Texas, USA*, 2011.

Gulwani, S., Polozov, O., and Singh, R. Program synthesis. *Foundations and Trends in Programming Languages*, 4 (1-2):1–119, 2017. doi: 10.1561/2500000010. URL https://doi.org/10.1561/2500000010.

Iyer, S., Cheung, A., and Zettlemoyer, L. Learning programmatic idioms for scalable semantic parsing. In *EMNLP-IJCNLP*, 2019.

Jang, E., Gu, S., and Poole, B. Categorical reparameterization with gumbel-softmax. In *International Conference on Learning Representations (ICLR)*, 2017.

Kaiser, Ł., Roy, A., Vaswani, A., Parmar, N., Bengio, S., Uszkoreit, J., and Shazeer, N. Fast decoding in sequence models using discrete latent variables. In *International Conference on Machine Learning (ICML)*, 2018.

Kingma, D. P. and Welling, M. Auto-encoding variational bayes. In *International Conference on Learning Representations (ICLR)*, 2014.

Koza, J. R. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112, 1994.

Kudo, T. and Richardson, J. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 66–71, November 2018.

Lee, W., Heo, K., Alur, R., and Naik, M. Accelerating search-based program synthesis using learned probabilistic models. In *Conference on Programming Language Design and Implementation (PLDI)*, pp. 436–449, June 2018.

Maddison, C. J., Mnih, A., and Teh, Y. W. The concrete distribution: A continuous relaxation of discrete random variables. In *International Conference on Learning Representations (ICLR)*, 2017.

Manna, Z. and Waldinger, R. J. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, 1971.

Miao, Y. and Blunsom, P. Language as a latent variable: Discrete generative models for sentence compression. *CoRR*, abs/1609.07317, 2016. URL http://arxiv.org/abs/1609.07317.

Murali, V., Qi, L., Chaudhuri, S., and Jermaine, C. Neural sketch learning for conditional program generation. In *International Conference on Learning Representations (ICLR)*, 2018.

Nye, M. I., Hewitt, L. B., Tenenbaum, J. B., and Solar-Lezama, A. Learning to infer program sketches. In *International Conference on Machine Learning (ICML)*, 2019.

Odena, A., Shi, K., Bieber, D., Singh, R., Sutton, C., and Dai, H. BUSTLE: Bottom-Up program synthesis through learning-guided exploration. In *International Conference on Learning Representations (ICLR)*, September 2020.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pp. 311–318. Association for Computational Linguistics, 2002.

Parisotto, E., Mohamed, A., Singh, R., Li, L., Zhou, D., and Kohli, P. Neuro-symbolic program synthesis. In *International Conference on Learning Representations (ICLR)*, 2017.

Puduppully, R., Dong, L., and Lapata, M. Data-to-text generation with content selection and planning. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pp. 6908–6915. AAAI Press, 2019. doi: 10.1609/aaai.v33i01.33016908. URL https://doi.org/10.1609/aaai.v33i01.33016908.

Rezende, D. J., Mohamed, S., and Wierstra, D. Stochastic backpropagation and approximate inference in deep generative models. *CoRR*, abs/1401.4082, 2014. URL https://arxiv.org/abs/1401.4082.

Ross, S., Gordon, G., and Bagnell, D. A reduction of imitation learning and structured prediction to No-Regret online learning. In Gordon, G., Dunson, D., and Dudík, M. (eds.), *Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 15 of *Proceedings of Machine Learning Research*, pp. 627–635, Fort Lauderdale, FL, USA, 2011. PMLR.

Roy, A., Vaswani, A., Neelakantan, A., and Parmar, N. Theory and experiments on vector quantized autoencoders. *arXiv*, May 2018.

Salton, G. and McGill, M. J. Introduction to modern information retrieval. 1986.

Schkufza, E., Sharma, R., and Aiken, A. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’13, pp. 305–316, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318709.

doi: 10.1145/2451116.2451150. URL https://doi.org/10.1145/2451116.2451150.

Shin, R., Allamanis, M., Brockschmidt, M., and Polozov, O. Program synthesis and semantic parsing with learned code idioms. In *Neural Information Processing Systems (NeurIPS)*, 2019.

Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S. A., and Saraswat, V. A. Combinatorial sketching for finite programs. In *Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pp. 404–415. ACM, 2006.

Summers, P. D. A methodology for lisp program construction from examples. *Journal of the ACM (JACM)*, 24(1): 161–175, 1977.

Udupa, A., Raghavan, A., Deshmukh, J. V., Mador-Haim, S., Martin, M. M. K., and Alur, R. TRANSIT: Specifying protocols with concolic snippets. In *Conference on Programming Language Design and Implementation (PLDI)*, pp. 287–296. Association for Computing Machinery, 2013.

van den Oord, A., Vinyals, O., and Kavukcuoglu, K. Neural discrete representation learning. In *Neural Information Processing Systems (NeurIPS)*, 2017.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *Neural Information Processing Systems (NeurIPS)*, 2017.

Vijayakumar, A. K., Cogswell, M., Selvaraju, R. R., Sun, Q., Lee, S., Crandall, D. J., and Batra, D. Diverse beam search: Decoding diverse solutions from neural sequence models. In *AAAI*, 2018.

Wan, Y., Zhao, Z., Yang, M., Xu, G., Ying, H., Wu, J., and Yu, P. S. Improving automatic source code summarization via deep reinforcement learning. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 397–407, 2018.

Wei, B., Li, G., Xia, X., Fu, Z., and Jin, Z. Code generation as a dual task of code summarization. In *Neural Information Processing Systems (NeurIPS)*, 2019.

Zohar, A. and Wolf, L. Automatic program synthesis of long programs with a learned garbage collector. In *Neural Information Processing Systems (NeurIPS)*, 2018.