Neural-Based Heuristic Search for Program Synthesis

Kavi Gupta

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-135
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-135.html

June 24, 2020

Acknowledgement

The work on Semantic Parsing involved contributions from Richard Shin, Joel Simonoff, Jonathan Wang, Mark Wu, and Xiaouyuan Liu. The work on the Neural Debugger involved contributions from Xinyun Chen and Peter Ebert Christensen. Most of the text and images in Chapter 3 are adapted from a paper that both Xinyun and Peter were involved in writing and editing.

Neural-Based Heuristic Search for Program Synthesis

by

Kavi Gupta

A thesis submitted in partial satisfaction of the

requirements for the degree of

Masters

in

EECS

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Dawn Song, Chair
Professor Koushik Sen

Spring 2020

The thesis of Kavi Gupta, titled Neural-Based Heuristic Search for Program Synthesis, is approved:

Chair _____ Date _____

_____ Date 06/24/20

_____ Date _____

University of California, Berkeley

Neural-Based Heuristic Search for Program Synthesis

Abstract

Neural-Based Heuristic Search for Program Synthesis

by

Kavi Gupta

Masters in EECS

University of California, Berkeley

Professor Dawn Song, Chair

Program Synthesis differs from other domains in machine learning due to the unforgiving nature of the task of writing programs. Since programs are precise sets of instructions, there is a much higher bar before which the results of a program synthesis system are useful. This thesis explores search based techniques that use auxiliary information in order to best improve the performance of systems that solve the task of program synthesis, in both semantic parsing and induction from examples domains.

To my parents, who, among many other contributions, housed me while I was working on my thesis through quarantine.

Contents

List of Figures

List of Tables

Acknowledgments

The work on Semantic Parsing involved contributions from Richard Shin, Joel Simonoff, Jonathan Wang, Mark Wu, and Xiaouyuan Liu. The work on the Neural Debugger involved contributions from Xinyun Chen and Peter Ebert Christensen. Most of the text and images in Chapter 3 are adapted from a paper that both Xinyun and Peter were involved in writing and editing.

Chapter 1

Introduction

Program Synthesis, the task of writing a program given some kind of specification, has been explored by many deep learning techniques. However, it has been difficult to gain the kinds of accuracies on these tasks you might hope to see in other areas such as computer vision. This is in part because program synthesis is less forgiving: it is much more important to get a program exactly right than other kinds of output. As a result in cases where synthesis fails, it is often because a correct program is almost found or found but not selected.

This work seeks to bridge that gap by adapting and improving standard search and repair techniques for the purpose of improving program synthesis. We study a Text-to-SQL domain of Spider and the inductive program synthesis domain of Karel. In Spider we work on improving beam search by retooling ensembling and reranking to work more effectively on this domain and incorporate a language prior. On Karel, we adapt techniques from program repair to iteratively repair programs.

1.1 Program Synthesis

Program synthesis is one of the oldest problems in artificial intelligence, and has been studied from the beginnings of the field.[28, 17] It has been studied in the programming languages community by many from several different perspectives and in recent years it has begun to be studied by many in the machine learning community[13, 1, 2, 26, 16, 21]. Stated simply, the goal of program synthesis is an ambitious one: to automate the process of programming itself. In practice, there are many ways of stating this problem, and each comes with its own challenges.

1.2 Problem Statements

There are many ways of stating the problem of program synthesis, but I will review a few that are more common or are relevant to the work I present in this thesis.

Semantic Parsing

The problem of semantic parsing is probably the most straightforward interpretation of the problem of program synthesis. Typically when humans are given the task of writing a program, they are given an informal and incomplete specification of its behavior in natural language and expected to turn this into precise code that accomplishes the desired task. The problem of semantic parsing is exactly this.

However, while for humans the difficult portion of the task is generally precisely defining the intent of the specification in code, for AI techniques, currently a major component of the task is in understanding the natural language specification. Thus, the term "semantic parsing" is used to refer to the problem of parsing the semantic meaning of natural language. Thus, much of semantic parsing is in domains where the program itself is quite simple, though there are some notable exceptions, including the SQL domain that this work is in.

Inductive Program Synthesis

Inductive program synthesis is the task of writing a program given examples of its behavior (for a more formal description see Section 3.1). The problem of inductive program synthesis can be seen as a compromise between synthesis from formal specification and semantic parsing. Like semantic parsing, it takes inputs in a form that can be easily provided by a human or other system, and like synthesis from a formal specification, there is a mechanically checkable answer as to whether the generated program matches the specification. The approaches in this space tend to be neural [13, 16, 21] as they require the ability to model both a prior on which programs are likely, as well as being able to construct a program from incomplete information. However, some work has been done in the space of inductive program synthesis by Bayesian reasoning and direct search[14].

1.3 Utility

The problem of program synthesis has broad utility in a variety of fields. It has many direct applications, with inductive program synthesis being used to generate formulas in spreadsheets[20] to semantic parsing being used to directly answer questions by converting text questions into SQL queries[31]. Additionally, program synthesis can be used as a component of larger systems like any other tool, as it allows for the creation of programs, which are typically more generalizable and explicable than neural models. Effectively, if this problem were to be better solved, it would allow neural networks to program, and provide similar benefits to allowing humans to program, gaining generalizability and explicability alike.

1.4 Previous Work

Quite a bit of progress has been made over the last decade by applying neural techniques to solving this problem. In semantic parsing, simpler tasks, such as WikiSQL[33], which involves the construction of join-free SQL queries, and ATIS, which focuses on one domain to draw terms for, have largely been solved in the existing literature. However, the more complex Spider[31] task has proven more challenging.

Similarly, in inductive program synthesis, tasks such as FlashFill[20] and Solid Geometry[11] that involve the synthesis of non-loop-containing programs, much progress has been made on solving these with direct approaches as well as approaches that take into account the current state of the program. Attempts to apply these techniques to Karel have also been successful, though they have involved the use of depth first search techniques that are quite expensive.[6]

In general, as the domains get more difficult and the programs that need to synthesize get more complex, this problem of exact match becomes greater and greater. Auxiliary information and the combination of multiple models is more useful in these settings, due to the highly structured nature of the outputs.

1.5 Auxiliary Information

To increase the performance of neural networks on an exact match task, a variety of techniques are used. Two main techniques for the use of auxiliary information in combining the outputs of models are relevant to the work in this thesis. We also explore other techniques that are along these lines.

Beam search is an algorithm that allows for searching the space of a model that outputs the conditional probabilities of each token in a sequence for a sequence that has maximal probability. This is commonly used at evaluation in order to improve the quality of a model at evaluation time and is ignored at training time, though there has been some work in training the entire beam search as well.[29, 18]

Ensembling is a technique in which the outputs of several models are aggregated to produce a single prediction of the output. This allows for an unbiased estimate of the output while also allowing multiple models to make different errors, allowing for a reduction in variance. Some training-time techniques such as dropout are likened to ensembling[2], but ensembling itself is typically only performed at evaluation.

We explore improvements to these techniques as well as different ways of using search that allow us to more effectively employ auxiliary information such as knowledge of language or how a program executed.

Chapter 2

Semantic Parsing

2.1 Background

Spider Domain

The problem of Semantic Parsing, to convert a natural language sentence into a precise program, is a traditional problem in natural language understanding and program synthesis. Tasks in this space include converting a sentence to a logical form [4], constructing short programs [19, 10, 30], and generating SQL queries. [30, 31, 24]

The domain we focus on is specifically SQL queries. There are many datasets for the problem of semantic parsing targeting SQL, as this is a task with direct practical applications. The dataset WikiSQL is composed of tables drawn from Wikipedia, each of which contains queries relating to that table.[33] The queries do not contain `JOIN` statements, however. The GEO[32] and Academic[15] datasets both involve joins across multiple tables, but are single domain. The Spider domain[31], used for this project, has multiple "databases," each of which contains multiple tables, and a set of queries that involve joins on multiple queries. Each domain is additionally different from the others and there is a train/test split on the domains to be used; that is a successful model must be able to generalize across domains.

An example Spider task is to, given the question "Show the stadium name and the number of concerts in each stadium" and the schema shown in figure 2.1, generate a query equivalent to the question. The target, or "gold," query is `SELECT T2.name, COUNT(*) FROM concert AS T1 JOIN stadium AS T2 ON T1.stadium_id = T2.stadium_id GROUP BY T1.stadium_id`. Note that this is more nontrivial since not only do the relevant tables need to be selected but their relationship must also be determined.

Challenges of the problem domain

There are a number of challenges to solving problems in the Spider domain. One is that the highly structured nature of SQL makes the problem difficult to solve in a direct sequence-to-sequence manner. That is why, as a baseline, we built on previous sequence-to-structure

```
CREATE TABLE IF NOT EXISTS "stadium" (
"Stadium_ID" int,
"Location" text,
"Name" text,
"Capacity" int,
"Highest" int,
"Lowest" int,
"Average" int,
PRIMARY KEY ("Stadium_ID"));

CREATE TABLE IF NOT EXISTS "singer" (
"Singer_ID" int,
"Name" text,
"Country" text,
"Song_Name" text,
"Song_release_year" text,
"Age" int,
"Is_male" bool,
PRIMARY KEY ("Singer_ID"));
```

```
CREATE TABLE IF NOT EXISTS "concert" (
"concert_ID" int,
"concert_Name" text,
"Theme" text,
"Stadium_ID" text,
"Year" text,
PRIMARY KEY ("concert_ID"),
FOREIGN KEY ("Stadium_ID") REFERENCES "stadium"("Stadium_ID"));

CREATE TABLE IF NOT EXISTS "singer_in_concert" (
"concert_ID" int,
"Singer_ID" text,
PRIMARY KEY ("concert_ID","Singer_ID"),
FOREIGN KEY ("concert_ID") REFERENCES "concert"("concert_ID"),
FOREIGN KEY ("Singer_ID") REFERENCES "singer"("Singer_ID"));
```

Figure 2.1: An example Spider schema

work that structurally modeled the output[24]. The other major challenge is that the split between the train and test domains makes it difficult for the model to generalize from one to the other without having some general understanding of English built in, but that finetuning a full language model to this task could be quite time consuming. Finally, a more general issue with this domain is that ensembling the results of several models together in ways beyond simple majority vote is difficult as sequential and structured data cannot be naively averaged.

2.2 Reranking

Reranking, the process of using a classification model to rerank multiple outputs of a regression model that outputs multiple results, has been used in the past for other datasets, including GEO and ATIS, text-to-sql datasets [30]. The main idea is to use the existing text-to-sql model to generate training data containing (question, correct query) and (question, incorrect query) pairs, and then train a classification model to be able to tell these apart.

This approach works well when the purpose of reranking is largely to determine logical incoherence, when the predicted query is logically very different from the inputted sentence. However, in Spider there is an additional problem that can occur of domain incoherence, where the model fails to generalize beyond its domain and in doing so returns a query that is clearly intended to be for a different domain. Since these examples are easier to fix and often numerous, the training data is augmented with (question, unrelated query) pairs to increase its size and enable the classification model to be able to do better on this kind of query.

Additionally, to better be able to model language across domains, we use a finetuned version of a pretrained BERT model[7]. Starting with a model that is pretrained on the English language as a whole allows the classifier to be able to bring in knowledge of domains not seen in the training data. Additionally, it is relatively inexpensive to train a classification model for this task versus performing end-to-end training on a BERT-based model for the text-to-sql task itself. Isolating BERT in a reranking model allows for the use of its power without the need for expensive training.

2.3 Ensembling

Background

Model ensembling is a traditional technique for improving the accuracy of a model. It is used in pretty much every machine learning domain from vision and natural language processing to robotics. In general, the purpose of ensembling is to reduce the variance of the models without reducing bias, by aggregating the results of several models into a single model. For classification and single-point regression, this is a relatively simple task, as both majority vote and direct averaging work relatively well.

However, for structured data, neither approach works well. First, the majority vote strategy only really works in situations where multiple models can agree on an entire structured output, which reduces in probability as the number of dimensions of this output increases. Averaging does not work, as even if the structured outputs are of roughly the same shape, averaging ignores the joint probabilities inherent in each sequence: for example "this is a cat" and "a cat is this" cannot be coherently averaged elementwise.

Additionally, for structured data generated by a recurrent model, usually some sort of beam search is utilized to get the results. One side effect of this process is that an entire list of outputs is generated for each model, and it would be useful to perhaps pick one of the outputs that the model found less likely, similar to reranking (see Section 2.2) by the outputs of other models.

Beam Stepwise Ensembling

In a beam search, a conditional model $P(y_{t+1}|x, y_0, \dots y_t)$ is searched for a sequence that maximizes $P(y_1 \dots y_n|x)$ by maintaining a set of hypotheses b_t for every timestep and then setting b_{t+1} to be the top k elements of $\{(y_0 \dots y_t, y_{t+1}) : (y_0 \dots y_t) \in b_t, y_{t+1} \in A\}$ by the scoring function $P(y_0 \dots y_t, y_{t+1}|x) = P(y_{t+1}|y_0 \dots y_t, x)P(y_0 \dots y_t|x)$. Once an end token is found, a beam is considered complete, and once k beams are complete, they can be returned. This process thus takes kn time, where n is the maximal length of the sequence, but can improve inference quality substantially.

If we have M models, and run a beam search on each, we can imagine this as having M different pools of hypotheses, each of which is expanded and then pruned by taking the top

k beams by $P_M(y_0 \dots y_t|x)$. In the ensembling strategies discussed above, at the end, the results are somehow combined.

The idea behind beam stepwise ensembling is that rather than trying to aggregate the overall model outputs $(y_0 \dots y_n)$ or even the final model probabilities $P(y_0 \dots y_n)$ (this is what majority vote does), we can aggregate the model likelihoods at every step.

Instead of having M separate pools of k hypotheses, we keep track of only 1 pool of k hypotheses, and at every step we expand and then take the top k for the next step by

$$\frac{1}{M} \sum_{m=1}^{M} \log P_m(y_0 \dots y_t|x)$$

In this way, we allow each model to fully evaluate each hypothesis, but also for each model's evaluation to influence the search for the other models.

Ensembling By Input

Traditionally, ensembling has been done per model, as described above. But in practice, all of the techniques are really ensembling over a set of model/input pairs (m, x). If there are several inputs that encode the same output, variance reduction can also be accomplished by ensembling over inputs. In the Spider dataset, there are often several questions corresponding to a single output query: these questions are paraphrases of each other.

We explore two modes of combining these information sources: ensembling using Beam Stepwise Ensembling and grouping by providing all the sentences to the model. We are interested in whether a relatively black box inference-only approach like input ensembling can work as well as an approach based on training models to explicitly handle multiple inputs.

Machine Translation

In practice, it is generally an unrealistic assumption to suggest that multiple inputs can be provided by the end-user for a given search problem. However, for tasks where the input is a natural language sentence, and the meaning is what needs to be extracted, it is possible to get multiple phrasings of the question by the use of pivoting[3], in which a pretrained machine translation model is used to generate several paraphrases of a sentence by translating the sentence into a "pivot language" and then translating it out of it. By using several languages, several paraphrases can be made, which can be then ensembled or grouped as inputs. The use of machine paraphrases has been used for the purposes of improving neural tasks, and we evaluate their use in the context of our input ensembling and grouping techniques.

2.4 Results

Reranking

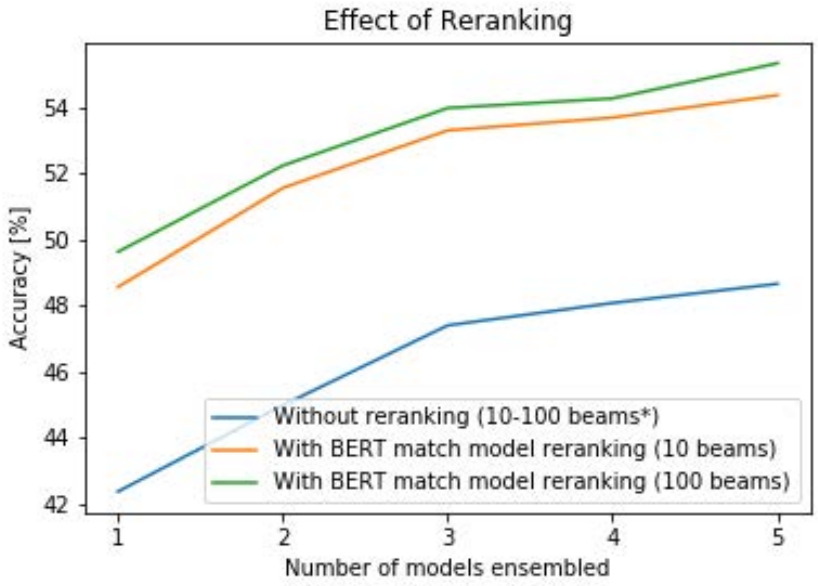

Figure 2.2: The effect of reranking on the overall performance of the model. *The model without reranking performs identically well with 10 or 100 beams.

Reranking using a BERT based model produced dramatic benefits of 6.19%pt when running the model with a beam size of 10, and 7.25%pt when using a beam size of 100. The gains from ensembling and reranking were not entirely additive, but using 100 beams, an ensemble of 5 models, and reranking, we achieved an accuracy of 55.32%, up from 42.36%.

Ensembling

Ensembling using the Stepwise Beam Ensembling technique outperformed majority vote, with ties broken by the model with highest confidence, as well as with the unrealistic setting where ties were broken by the model with the best performance on the entire test set. With 5 models being ensembled, it produced an increase in performance of 6.27% as opposed to 5.32% for best model ensembling and 5.13% for highest confidence ensembling.

Ensembling multiple inputs together also produced improvements, though these were far higher for multiple human paraphrases than machine paraphrases. However, the relative improvements over not ensembling inputs were greater for fewer models being ensembled, suggesting that at least part of the improvement created by input ensembling is due to it

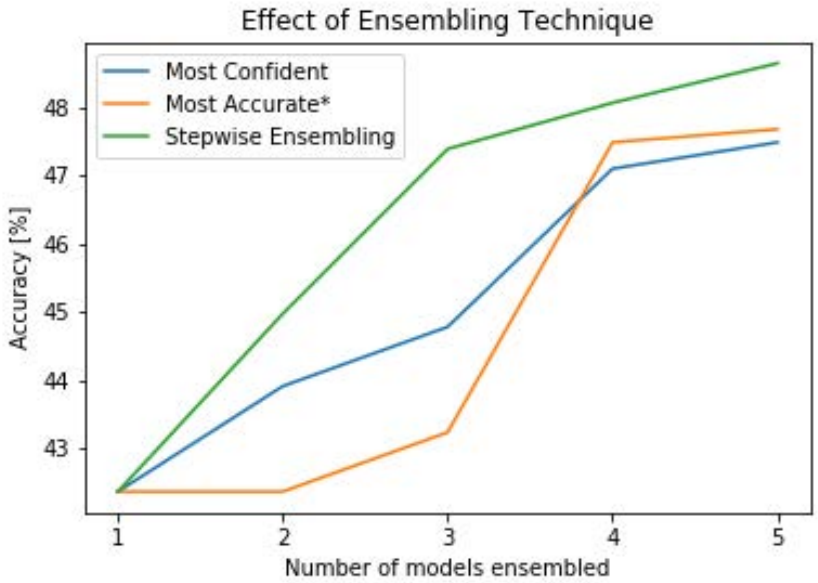

Figure 2.3: The effect of ensembling technique on the overall performance of the model. *the "most accurate" technique is unrealistic as it takes into account the model's performance on the test set

reducing variance by adding more model/input pairs to ensemble. This effect can also be seen with input grouping, which suggests that input grouping also follows a similar pattern. We can also see that with 1 model ensembled, about 25% of the improvement in grouping can be achieved via ensembling, suggesting that this is a technique that makes sense to apply in situations where training a new model is infeasible.

Finally, we can see that machine translation produces fairly disappointing results as compared to human paraphrases. In general, it seems as though the bias introduced by low quality machine translation overwhelms the extra information that can be gained by using them, though they do have some modest benefits when model ensembling is not also used.

2.5 Conclusion

With reranking, we were able to see fairly dramatic gains in performance, that can be attributed to the power of a general purpose understanding of the English language. Additionally, these gains could be made without having to train an entire BERT based model end-to-end for the purpose of constructing SQL queries, instead the model could be trained separately and used to leverage its knowledge directly. This forms a more efficient approach to the integration of complicated external models, more sophisticated than using them for

untrained embeddings but simpler than training them with the main model end-to-end.

With stepwise ensembling, we see that using a beam-search specific ensembling technique for handling sequential data leads to better performance than using traditional majority based techniques that are more general purpose. This technique is no less efficient than running several beam searches and synthesizing the results. Additionally, nothing about this technique is specific to the Spider domain. It is thus potentially generalizable to any domain involving sequential data output. Additionally, since it is purely an inference technique and does not require changing the model in any way, it can be applied to pretty much any ensemble outputting a sequence with minimal effort, and can improve performance nearly for free.

Ensembling inputs is a relatively novel concept, and we see that in practice it can gain about 25% of the improvement that can be achieved by grouping together the inputs for the model. This means that while there are obvious benefits to using grouping, ensembling can be done for any model architecture and does not require changes to the model architecture to be able to handle multiple inputs. However, the gains achieved by ensembling inputs reduce as more models are ensembled, suggesting that it performs the same role as model ensembling, and thus probably can be replaced by it if retraining is computationally infeasible. Future work should probably investigate whether the effect of input ensembling can be directly reproduced by small pertubations to the embeddings of the inputs, and if input ensembling is effectively performing a similar role as dropout or some other kind of internal-to-a-model ensembling technique.

Chapter 3

Neural Debugger

3.1 Background

Inductive Program Synthesis

Neural Inductive Program Synthesis is the problem of synthesizing a program given examples of its input/output behavior. Precisely, there is some latent prior on programs $P(p)$ and on program inputs $P(x|p)$. The prior is such that $p(x)$ is always defined when $x \sim P(x|p)$. The neural synthesizer is a model M that is given input/output examples $(x_1, p(x_1)), \ldots, (x_k, p(x_k))$ where $p \sim P(p)$ and $x_k \sim P(x|p)$ and produces a program p^*. p^* is considered correct if $p^*(x) = p(x)$ for $x \in \{x_1 \ldots x_k\}$ and $p^*(x_{k+1}) = p(x_{k+1})$, where $x_{k+1} \sim P(i|p)$ is drawn independently of each of the other inputs.

In a sense, this problem can be viewed as a generalization of the regression problem, in which there is some dataset $(x, y) \sim D$ and the model M is given training input/output examples $(x, y) \sim D$, and its goal is to create a function that can match independently drawn evaluation samples $(x, y) \sim D$. For example, in linear regression, the model M takes in $(x_1, y_1) \ldots (x_k, y_k)$ and returns a weights vector w that represents the program $x \mapsto w^T x$, which best matches the given data.

The primary conceptual difference is that in this case the distributions (x, y) can have quite distinct characteristics from each other, as they are conditioned on p, and thus the model must be able to infer program structure as well as parameters from the examples. Additionally, since the prior $P(p)$ is unknown, it must be inferred to some extent by the model in order to produce programs that can satisfy the held out examples (for example the program that is a table of every possible input and output is unlikely given the prior).

Karel

The specific domain that my work this semester is in is Karel. Karel is a language designed to teach an introductory programming course at Stanford. In this language, the program represents the actions of a robot that can observe and interact with its immediate

environment. The language has loops and conditionals whose conditions represent given environment states. Figure 3.1 shows an example Karel program, and its corresponding input/output examples can be seen in figure 3.2. As can be seen in this example, relatively nontrivial control flow can be expressed in this language.

```
while(not(rightIsClear)) {
    move
}
turnRight; move; putMarker; turnLeft; turnLeft; move
turnRight
putMarker; move
while(not(rightIsClear)) {
    putMarker
    move
}
```

Figure 3.1: An example Karel program

The two features of this language that make it interesting from a programming languages perspective are the fact that the input and output pairs are entire grids, which allows for an easier inference of program intent from a very small number of examples (usually 5), and the fact that the actual programs are quite complex, often containing loops, which are not present in most other inductive program synthesis domains.

Related Work

The architecture we use to solve the specific task of Karel program synthesis is inspired heavily by the work of Bunel et al [5], which uses convolution on each of the Karel input and output grids in order to produce a task embedding, and then feeds each task embedding to identical RNNs that produce outputs that are maxpooled and used to predict output tokens. I use this architecture with some modifications for all the networks we discuss in this work. Our program editing strategy in Karel is based on that of Shin et. al. [25]: an LSTM that predicts Levenshtein edits to the given program. We use this editor with a modification to allow for the encoding of full execution traces. More generally, program repair has been a longstanding problem in the field [27, 22], and traditionally has been posed as the problem of fixing errors in human written programs. While we are tackling a similar problem, in that we are attempting to fix the output of a programmer, in our case the programmer is itself a neural network and thus has different properties. Namely, it is much easier to generate a dataset of incorrect programs, and we might hope that the errors would be more regular and formulaic, and less socially dependent. The use of an repair as a way to iteratively improve

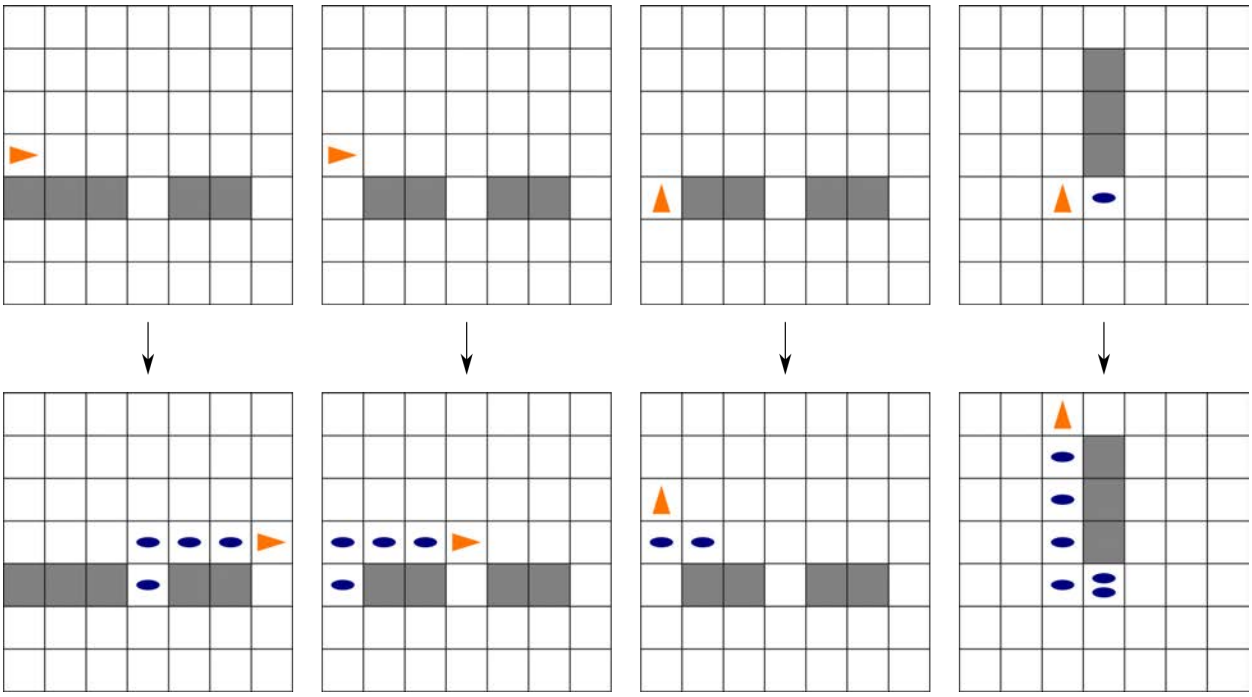

Figure 3.2: Several input output examples corresponding to the program in figure 3.1. The gray blocks represent walls, which Karel cannot travel through, and the blue ovals represent markers, which can be put and picked. The orange triangle represents Karel, who operates like "turtle" in a turtle graphics engine.

a program is explored in [12], although they use it in the context of decompilation. We use a similar iterative editing technique, but with a greater focus on search techniques and in the context of program synthesis from examples rather than alternate representation, which makes the dependency between the specification and the program more complex.

3.2 Execution Information

Unlike in program synthesis from specification or natural language, in the case of synthesis from examples execution information is a piece of the problem. How the problem performs on the input/output set can be a useful guide to solving the problem, as if a program does not succeed on the input/output examples. Thus, search based techniques that seek to find a "reasonable" program that satisfies the input/output examples can produce large improvements in performance.

Partial Program Execution

The standard approach to incorporating execution information is to execute partial programs as part of a search to find the final program. This approach is used in Ellis et al.,[11], who use it in the constructive solid geometry domain, where the task is to generate a sequence of shapes that match a given image. The main idea behind their approach is to have trained code writing and evaluation functions that can be trained in tandem along with search to be able to evaluate the quality of a partial program at every step in the training process. By executing partial programs, they are able to more effectively guide their search, and can use techniques like reinforcement learning more easily. In a more traditional programming setting of a loop free sequential data processing language, Zohar et al use a model that can predict the next program token given the current state and goal of a program. When combined with search, this technique can be used to generate longer programs than previous baselines on this same task. [34]

Partial program execution is particularly well suited to constructive solid geometry as the execution of a partial program will lead to a partially correct result. This property is less well suited to Karel, where loops make the execution of a partial program more complicated. Regardless, Chen et al.,[6] are able to produce improvements using partial program execution. Their method involves training the synthesizer to be able to predict the next token, given a suitably modified input state that represents the results of executing the program up to that point. This enables the use of execution information in most contexts. One downside to this approach, however, is that since they process each token sequentially, they synthesize a loop with only the information about what happens at its first iteration. This means that they cannot take into account the evolution of the state as the loop progresses.

Full Program Execution

Full program execution involves executing an entire program produced by a model. This clearly allows for more accurate information to be collected regarding how a program works, it for example handles loops and other nontrivial control structures implicitly, without having to explicitly model them the way that partial program execution does.

Full program execution has been used by many in the Karel domain as a post-selector on a set of programs created by a model, allowing for the selection of a program that passes all the test cases if one has been created, allowing for an increase in accuracy.[5, 6]

Synthesize, Execute, and Debug

While full program execution for post-selection can prevent missing an already generated correct program, it requires that the model be able to synthesize the correct program immediately, in at least one of its outputs. However, humans program quite differently from this, typically executing their current program, and then using execution information to update their program. This suggests that a different, iterative, approach be used to this problem:

first a candidate program is generated, and is then repeatedly edited until a program is found that passes the test cases.

This general strategy has been used in a few other settings, such as decompilation[12], but it has not been used in environments where program execution contains loops or other complex control flow that makes the relationship between program and outputs complex. Their approach is also less search-focused than the one used in this project.

Additionally, there has been a large body of work on program repair, but this generally focuses on repairing human mistakes, which is a different problem from repairing mistakes made by a neural model, in large part because data availability is a far greater concern with finding bugs in human-generated programs.

The problem to be solved by the neural debugger is then the exact problem of inductive program synthesis but with the additional input of an assumed partially correct program to make edits to. This model has been explored in some prior work, for example see Shin 2018[25].

Synthesizer

Our SED framework is largely agnostic to the choice of the synthesizer, as long as it achieves non-trivial prediction performance, thus it is beneficial to leverage its predicted programs for the debugger component. In particular, SED is compatible with existing neural program synthesis models that largely employ the encoder-decoder architectures [5, 6, 9]. A common model architecture for input-output program synthesis includes an encoder to embed the input-output pairs, which could be an LSTM for string manipulation tasks [9], or a convolutional neural network for our Karel task [5, 6, 23]. Then, an LSTM decoder generates the program based on the input embedding.

Debugger

We present the debugger architecture in Figure 3.4. We follow previous work for Karel domain to use a convolutional neural network for I/O embedding, a bi-directional LSTM to encode the program for debugging, and an LSTM to sequentially generate the edit operation for each input program token [25]. The debugger supports 4 types of edit operations: `KEEP` copies the current program token to the output; `DELETE` removes the current program token; `INSERT`$[t]$ adds a program token t; and `REPLACE`$[t]$ replaces the current program token with t. Therefore, the total number of edit operations is $2|V|+2$, where $|V|$ is the Karel vocabulary size. For `KEEP`, `REPLACE` and `DELETE`, the LSTM moves on to process the next program token after the current edit operation, while for `INSERT`, the next edit operation still based on the current program token, as shown in Figure 3.3.

The input program serves as a cue to the desired syntactic structure; however, it may not be sufficient to reveal the semantic errors. Motivated by the breakpoint support in Integrated Development Environments (IDEs) for debugging, we propose an execution trace embedding

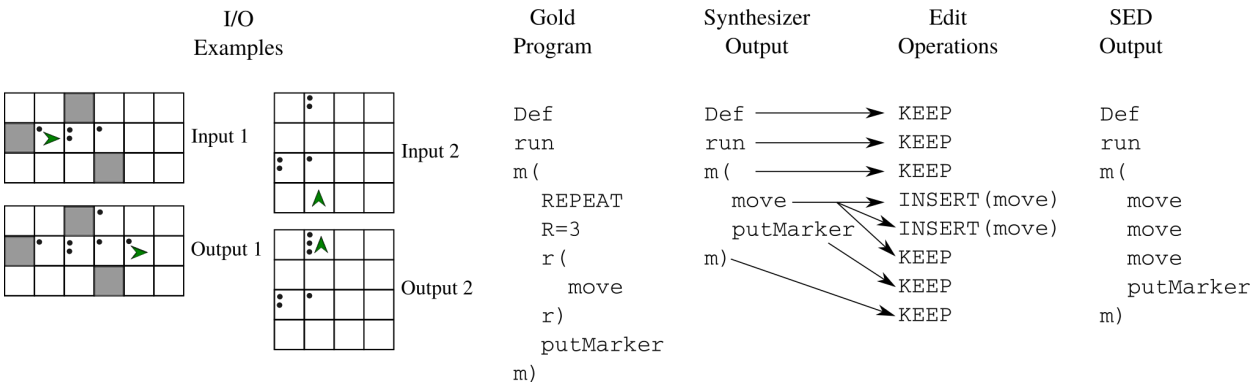

Figure 3.3: A sample debugging process of SED. Given the input-output examples, the synthesizer provides a wrong program that misses the repeat-loop in the ground truth. Our debugger then performs a series of edits, which results in a correct program. Note that the `INSERT` operation does not advance the pointer in the input program, so several edits are applied to the `move` token.

technique and incorporate it into the original debugger architecture, as highlighted in Figure 3.4. Specifically, we first execute the input program on each input grid i_u, and obtain the execution trace $e_{u,0}, ..., e_{u,t}, ..., e_{u,T}$, where $u \in \{1, 2, ..., K\}$. For each state $e_{u,t}$, we use a convolutional neural network for embedding:

$$te_{(u,t)} = \texttt{TraceEmbed}([e_{u,t}; i_u; o_u]) \tag{3.1}$$

Where $[a; b]$ means the concatenation of vectors a and b.

To better represent the correspondence between each program token and the execution states it affect, we construct a bipartite graph $E \subseteq W \times I$ where W is the set $\{(u, t)\}$, and I is the set of program token indices. We set $((u, t), i) \in E$ iff the program token p_i was either executed to produce $e_{u,t}$; or p_i initiates a loop, e.g., `repeat`, and the body of that loop produces $e_{u,t}$ when executed. For each program token p_i, let $w_{u,t,i} = \frac{1}{|\{(u',t'):((u',t'),i)\in E\}|}$ be the weight of each edge connecting p_i and its associated execution state, we compute a vector representation of its related execution states below:

$$q_i = \sum_{(u,t):((u,t),i)\in E} w_{u,t,i} e_{u,t} \tag{3.2}$$

Finally, the program token representation fed into the edit decoder is $ep'_i = [ep_i||q_i]$, where ep_i is the original program token embedding computed by the bi-directional program encoder.

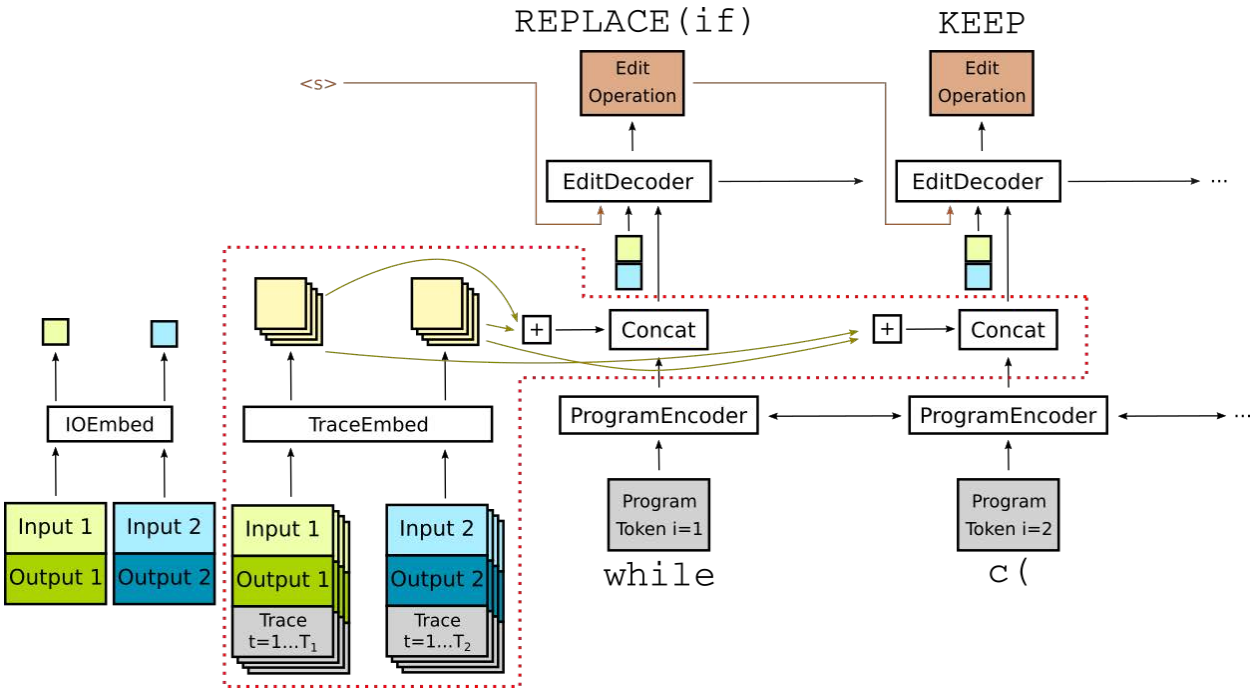

Figure 3.4: The neural debugger model in SED. The encoder consists of three parts: (1) `IOEmbed` for I/O embedding; (2) `TraceEmbed` that convolves the traces with their corresponding I/O pairs; and (3) `ProgramEncoder` that jointly embeds each program token with its corresponding execution steps in the trace. `EditDecoder` is used for generating edits. We outline our proposed `TraceEmbed` component in red dots, which is the key architectural difference compared to [25].

Training

We design a two-stage training process for the debugger, as discussed below.

Stage 1: Pre-training with synthetic program mutation.

We observe that if we directly train the debugger with the predicted programs of the synthesizer, the training hardly makes progress. One main reason is because a well-trained synthesizer only makes wrong predictions for around $15\% - 35\%$ training samples, which results in a small training set for the debugger model. Although the synthesizer could produce a program close to one that satisfies the input-output specification, it is mostly distinct from the annotated ground truth program, as indicated in our evaluation. Therefore, we build a synthetic program repair dataset in the same way as [25] to pre-train the debugger. Specifically, for each sample in the original Karel dataset, we randomly apply several mutations to generate an alternative program P' from the ground truth program P. Note that the muta-

tions operate on the AST, thus the edit distance between program token sequences of P and P' may be larger than the number of mutations, as shown in Figure 3.6. We generate an edit sequence to modify from P' to P, then train the debugger with the standard cross-entropy loss using this edit sequence as supervision.

Fine-tuning with the neural program synthesizer.

After pre-training, we fine-tune the model with the incorrect programs produced by the neural program synthesizer. Specifically, we run the decoding process using the synthesizer model on the Karel training set, then use those wrong predictions to train the debugger.

Inference Procedure

Algorithm 1 Greedy search algorithm

```
1: function GREEDY-SEARCH_k((e))
2:     c ← arg max_{p∈M(e)} T(p, e)
3:     S ← {}                                  ▷ Already expanded programs
4:     if T(c, e) = 1 then
5:         return c                            ▷ Success
6:     end if
7:     for i ∈ {1 ... k} do
8:         c ← arg max_{p∈D(c,e)\S} T(p, e)
9:         S ← S ∪ {c}
10:        if T(c, e) = 1 then
11:            return c                        ▷ Success
12:        end if
13:    end for
14:    return c                                ▷ Failure
15: end function
```

During inference, we achieve the best results using a best first search, as described in Algorithm 2. In the algorithm, we denote the synthesizer model as $M(e)$, which produces a list of candidate programs for input-output examples e. The debugger model $D(p, e)$ produces a list of candidate programs given the input program p. The function $T(p, e)$ executes the program p on the examples e, and returns a value in $[0, 1]$ representing the proportion of input-output examples that are satisfied.

Within our SED framework, we view program synthesis from specification as a search on an infinite tree with every node except the root node being annotated with a program c, and having children $D(c, e)$. Our goal is to search for a program p satisfying $T(p, e) = 1$. While $T(p, e) = 1$ does not ensure that the generated program is semantically correct, as e

Algorithm 2 Best first search

1: **function** BEST-FIRST-SEARCH$_k((e))$
2: $F \leftarrow M(e)$ ▷ "Fringe" of the search space: programs yet to be expanded
3: $S \leftarrow \{\}$ ▷ Already expanded programs
4: **for** $i \in \{1 \dots k\}$ **do**
5: $c \leftarrow \arg\max_{p \in F \setminus S} T(p, e)$
6: $S \leftarrow S \cup \{c\}$
7: **if** $T(c, e) = 1$ **then**
8: **return** c ▷ Success
9: **end if**
10: $F \leftarrow F \cup D(c, e)$
11: **end for**
12: **return** $\arg\max_{p \in F} T(p, e)$ ▷ Probable failure, unless the program was found on the final step
13: **end function**

does not include held-out test cases, it is a necessary condition, and we find it sufficient as the termination condition of our search process.

We design two search algorithms for SED. Our first algorithm is a *greedy* search, which iteratively selects the program from the beam output of the previous edit iteration that passes the greatest number of input-output examples (and has not yet been further edited by the debugger), and returns the edited program when it passes all input-output examples, or when it reaches the maximal number of edit operations allowed, denoted as k. See Algorithm 1 for more details.

A more effective scheme employs a *best-first* search. Compared to the greedy search, this search algorithm keeps track of all the programs encountered, as shown in line 10 of Algorithm 2, so that it can fall back to the debugger output from earlier edit iterations rather than get stuck, when none of the programs from the current edit iteration is promising.

3.3 Results

In this section, we demonstrate the effectiveness of SED for Karel program synthesis and repair. We first discuss the evaluation setup, then present the results.

Evaluation Setup

The Karel benchmark [8, 5] is one of the largest publicly available input-output program synthesis dataset that includes 1,116,854 samples for training, 2,500 examples in the validation set, and 2,500 test examples. Each sample is provided with a ground truth program, 5 input-output pairs as the specification, and an additional one as the held-output test example. We follow prior work [5, 23, 6] to evaluate the following metrics: (1) **Generalization.**

The predicted program P is said to generalize if it passes the all the 6 input-output pairs during testing. This is the primary metric we consider. (2) **Exact match.** The predicted program is an exact match if it is the same as the ground truth.

Program repair. In addition to the Karel program synthesis task introduced above, we also evaluate our debugger component on the mutation benchmark in [25]. Specifically, to construct the test set, for each sample in the original Karel test set, we first obtain 5 programs $\{P'_i\}_{i=1}^{5}$ by randomly applying 1 to 5 mutations starting from the ground truth program P, then we generate 5 test samples for program repair, where the i-th sample includes P'_i as the program to repair, and the same input-output pairs and ground truth program P as the original Karel test set.

Synthesizer Details

We consider two choices of synthesizers. The first synthesizer is **LGRL** [5], which employs a standard encoder-decoder architecture as discussed in Section 3.2. During inference, we apply a beam search with beam size $B = 32$. We also evaluate a variant that performs the greedy decoding, i.e., $B = 1$, denoted as **LGRL-GD**. The second synthesizer is the execution-guided neural program synthesis model proposed in [6], denoted as **EGNPS**. The model architecture of EGNPS similar to LGRL, but it leverages the intermediate results obtained by executing partial programs to guide the subsequent synthesis process. We present the performance of these synthesizers in the first row ("Synthesizer only") of Table 3.1.

Debugger Details

We compare our debugger architecture incorporated with the trace embedding component to the baseline in [25], and we refer to ours and the baseline as *TraceEmbed* and *No TraceEmbed* respectively. For the program synthesis task, all models are pre-trained on the training set of the synthetic mutation benchmark with 1-3 mutations. For the fine-tuning results, LGRL and LGRL-GD are fine-tuned with their synthesized programs, as discussed in Section 3.2. For EGNPS, we evaluate the debugger fine-tuned with LGRL, because EGNPS decoding executes all partial programs generated in the beam at each step, which imposes a high computational cost when evaluating the model on the training set.

Results

Mutation benchmark for program repair. Figure 3.5 shows the results on the mutation benchmark. For each debugger architecture, we train one model with programs generated using 1-3 mutations, and another one with 1-5 mutations. Our most important observation is that the debugger demonstrates a good out-of-distribution generalization performance. Specifically, when evaluating on 4-5 mutations, although the performance of models trained only on 1-3 mutations are worse than models trained on 1-5 mutations, they

can already repair around 70% programs with 5 mutations, which is desirable when adapting the model for program synthesis. On the other hand, each model achieves better test performance when trained on a similar distribution. For example, models trained on 1-3 mutations achieve better performance when evaluating on 1-3 mutations than those trained on 1-5 mutations.

Meanwhile, for each number of mutations, the lowest error is achieved by the model with our TraceEmbed component, demonstrating that leveraging execution traces is helpful. However, such models tend to overfit more to the training data distribution, potentially due to the larger model sizes.

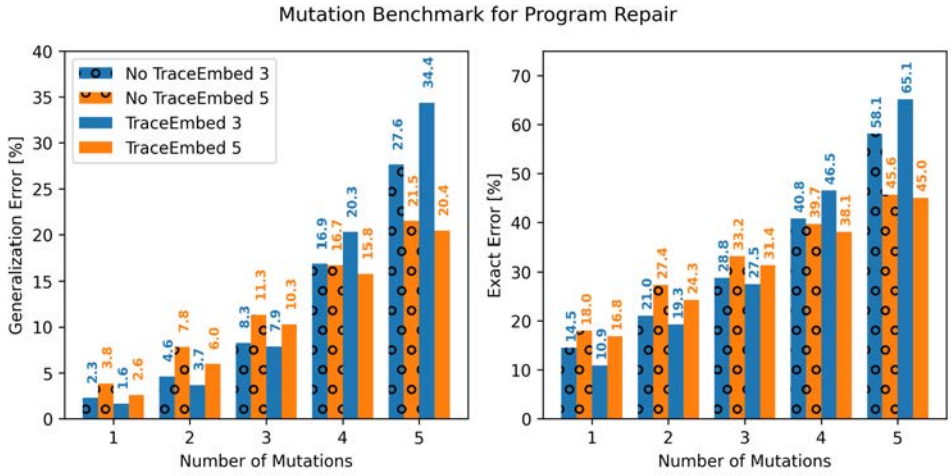

Figure 3.5: Results on the mutation benchmark, where x-axis indicates the number of mutations to generate the programs for repair in the test set. In the legend, "3" refer to models trained on 1-3 mutations, "5" refer to models trained on 1-5 mutations.

Table 3.1: Results on the test set for Karel program synthesis, where we present the generalization error with exact match error in parentheses for each synthesizer / debugger combination.

Synthesizer+Debugger	LGRL-GD	LGRL	EGNPS
Synthesizer only	39.00% (65.72%)	22.00% (63.40%)	19.40% (56.24%)
No TraceEmbed+No Finetune	18.72% (63.12%)	14.92% (62.80%)	11.68% (54.16%)
No TraceEmbed+Finetune	**16.20%** (**60.92%**)	14.32% (**62.48%**)	11.52% (53.68%)
TraceEmbed+No Finetune	18.80% (63.76%)	14.60% (62.88%)	11.48% (54.12%)
TraceEmbed+Finetune	16.32% (61.28%)	**14.28%** (62.68%)	**11.36%** (**53.52%**)

The Karel program synthesis benchmark. Table 3.1 presents our main results for program synthesis, where the debugger runs 100 edit steps. Firstly, SED consistently boosts the performance of the neural program synthesizer it employs, reducing the generalization error by at least 7.5%. In particular, with LGRL-GD as the synthesizer, SED significantly outperforms LGRL without the debugger, which shows that the iterative debugging performed by SED is more effective than the standard beam search. Meanwhile, with EGNPS as the synthesizer, even if the synthesizer already leverages the execution information to guide the synthesis, SED still provides additional performance gain, which confirms the benefits of incorporating the debugging stage for program synthesis.

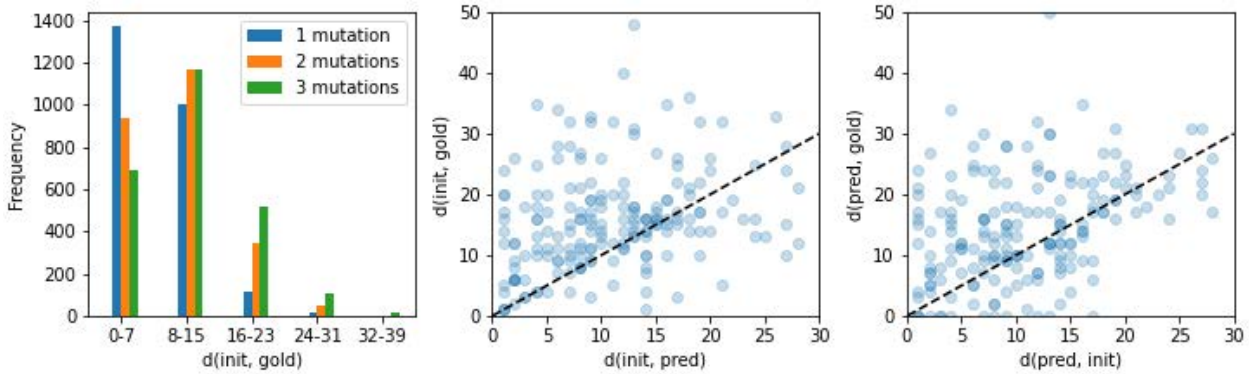

Figure 3.6: Left: The distribution of edit distances for the mutation benchmark by number of mutations. Middle and right: The joint distributions of the edit distances between the initial program predicted by the LGRL synthesizer (`init`), the `gold` program, and the program predicted by SED that passes all IO cases (`pred`). Dashed lines correspond to $x = y$.

To understand how SED repairs the synthesizer predictions, Figure 3.6 demonstrates the distribution of the edit distances between the initial and ground truth programs in the pre-training dataset (leftmost), and several distributions of edit distances among the ground truth, the predicted programs by the synthesizer, and the repaired programs by SED that is semantically correct (middle and right). The debugger is a TraceEmbed model without fine-tuning, and it performs 100 editing steps. Firstly, we observe in the middle graph that SED tends to repair the initial program towards a correct one that is closer to the prediction than the ground truth, and we also provide an example in Figure 3.3. The rightmost graph further shows that the repaired programs are generally closer to the initial programs than the gold ones, which is also the reason why the improvement of exact match achieved by SED is much smaller than the generalization metric. Comparing these distributions to the leftmost graph, we note that without fine-tuning, SED is already able to repair initial programs not only include semantic errors that might not correspond to the mutations it is trained on, but also with larger edit distances to the ground truth than the training samples, which again demonstrates the generalizability of SED.

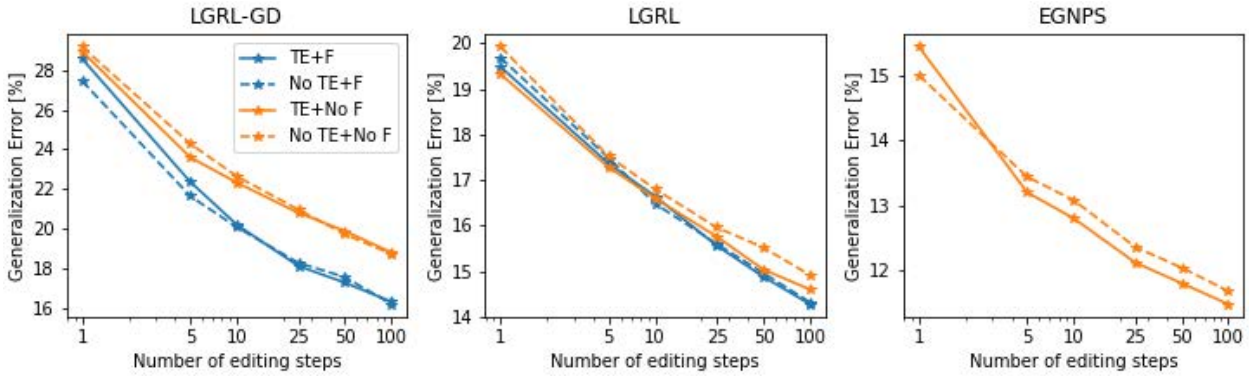

Figure 3.7: Comparison of different architectures and training process for program synthesis. TE refers to TraceEmbed, and F refers to fine-tuning on the data generated by the same synthesizer. Note the logarithmic scale of the x axis.

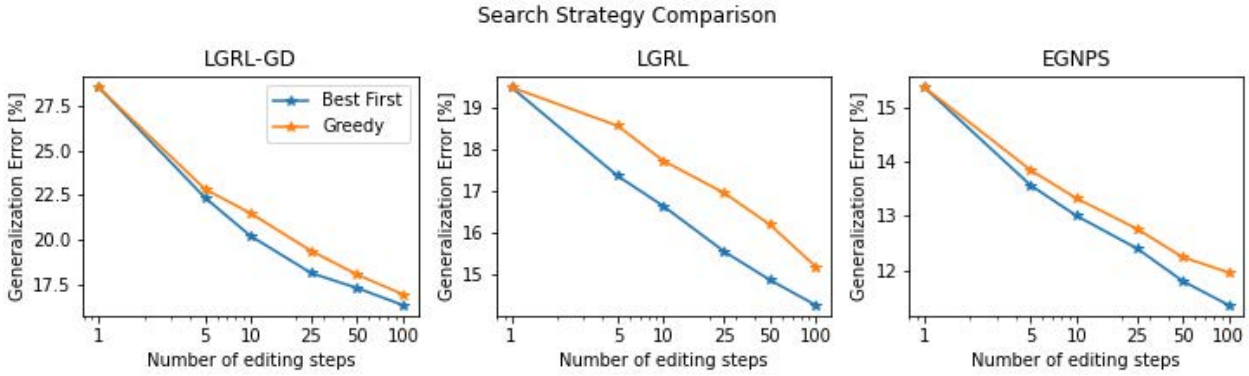

Figure 3.8: Comparison of best first and greedy search strategies. All models use TraceEmbed+Finetune as defined in Table 3.1.

Next, we discuss the effects of our trace embedding component and fine-tuning, and we further present the results with different number of edit steps in Figure 3.7. We observe that fine-tuning improves the results across the board, and has a particularly pronounced effect for LGRL-based models, where the data source for fine-tuning comes from the same synthesizer. Meanwhile, the debugger accompanied with the trace embedding mostly achieves better performance, especially when fine-tuning was not performed. This is potentially because the trace embedding component introduces additional model parameters, thus the model could suffer more from overfitting due to the small training set for fine-tuning.

Figure 3.8 compares the best first and greedy search strategies. We see that best first search always outperforms greedy search, often being able to achieve a similar performance in half the number of edit steps. This effect is more pronounced for LGRL and EGNPS

synthesizers, as they provide more than one program to start with, which best first search can more effectively exploit.

3.4 Conclusion

Program synthesis and program repair have typically been considered as largely different domains. In this work, we present the SED framework, which incorporates a debugging process for program synthesis, guided with execution results. The iterative repair process of SED outperforms the beam search when the synthesizer employs the greedy decoding, and it significantly boosts the performance of the synthesizer alone, even if the synthesizer already employs a search process or incorporates the execution information. Additionally, we found that even though there is a program aliasing problem for supervised training, our two-stage training scheme alleviates this problem, and achieves strong generalization performance. Our SED framework could potentially be extended to a broad range of specification-guided program synthesis applications, and we consider it as future work.

One thing of note is that full execution information with traces seems to be the most valuable within the context of short run edits where presumably it is useful in identifying the exact issue to be fixed, whereas a more general model is valuable in finding several reasonable edits that can then be filtered for accuracy. On the other hand, the search strategy seems to make a massive difference, suggesting that the expression of this problem as a search problem is a reasonable framing. We also see that finetuning leads to improvements in performance, suggesting that while mutation provides easy to access and construct data that covers a large space of possible edits, the distributional shift does play a part in depressing the results of the model when run in practice.

We are able to show state of the art performance for unensembled models, with our best result of 11.36% error beating the state of the art [6] model using RL, which achieves 13.96% error, but without the need for expensive depth first traversal of the search space[1].

3.5 Future Work

Several extensions to this project might prove interesting. Several presented below are ones that we worked on as part of this project. We were unable to get them to produce performance gains, but did not explore them thoroughly enough to comfortably rule them out.

One area of potential improvement involves the ensembling of several debugger networks together. There are many forms this could take, from simply running all the networks and combining together their beams at each step of the process to something more sophisticated like Beam Stepwise Ensembling (see section 2.3). Using the first technique of dovetailing the beams for 3 debuggers together, we were able to gain 0.5% reduction in error. This was

[1]The discrepancy between the best accuracy in that paper and our result was that we based our model on a non-RL-trained version of [6] model to get EGNPS

smaller than the results that we have seen in other domains and perhaps evidence that there is potentially a better strategy or different ways to generate more diverse models.

We additionally explored techniques that could determine whether a program was "overfit," that is, if it passed the given but not the held out test cases, in order to improve that performance. Specifically, the current model cannot recover at all if the synthesizer produces a program that passes the given but not the held out test cases. Being able to detect probably overfit programs would be quite useful. However, we found that when we tried training a simple classifier model on top of the encoder architecture discussed above to do this, we saw no improvements. A more purpose-built model might be better suited to the task and provide benefits.

Finally, one area that we briefly considered but abandoned involves the ensembled finetuning of the debugger, in which several debugger models are trained and then evaluated on the training set to produce examples of correct but non-gold programs that can then be used as ground truth programs to train the remainder of the debuggers. This strategy does not work for a single model particularly well, since it is clearly already capable of producing a correct program that it does in fact produce[2], but hopefully several models will have different errors enough to make this a reasonable data collection strategy. Unfortunately, we found that the number of data points we were able to gather from three models was too small to produce a good finetuning dataset. A more sophisticated approach involving the combination of some of the original gold programs with these generated programs might be beneficial, as might the use of more models to generate the ensemble.

One important area of future work that I did not attempt as part of this project is to generalize this approach to other inductive program synthesis domains that involve loop-containing programs. Unfortunately, most inductive program synthesis domains involve sequential rather than loop containing programs and this technique is likely to be less useful in these domains.

[2]Though technically there could be some learning signal as the program is not necessarily the MLE according to the model if it is found in a non-top beam, this signal is likely to be sparse and also lead to overfitting as the model will become more confident in answers it already thought were at least somewhat likely

Chapter 4

Conclusion

4.1 Application of specific results

In the course of the work of this thesis I think there are several methods that are generally applicable when doing program synthesis. Many of these techniques are capable of boosting the performance of models without too much additional computational power or modifications to the architecture.

Stepwise Beam Ensembling showed some fairly dramatic benefits over other modes of ensembling in Spider, but it is not in any way specific to Spider as a domain, and can probably be used in a variety of domains that involve the output of some sequential data. It also not more expensive than other forms of ensembling in terms of number of operations, though in concept it is somewhat more difficult to parallelize.

Input ensembling is probably a useful way to get at least some of the advantages of training a model with multiple inputs without having to retrain a model, though it does not provide all the improvements. More work in this area might reveal the best way to use this kind of ensembling to best improve results.

Iterative full-program-execution based search provides fairly dramatic improvements over existing models without too much time spent in the search in Karel, and this might generalize to other domains that involve nontrivial control flow.

Using highly synthetic data for program repair and then finetuning on more realistic data does seem to provide advantages that neither technique can on its own. We test this in the environment of repairing bugs introduced by an imperfect neural program synthesis module, but perhaps some of these same techniques could be applied to the more traditional problem of repairing human-created errors in programs. Additionally, the ability to zero-shot generalize from entirely synthetic issues in programs to issues created by a faulty program synthesis network suggest that potentially program synthesis-introduced errors could be used as a reasonable source of infinite synthetic data to train program repair networks to solve the problem of human-introduced errors directly.

4.2 General Conclusions

In general, neural architectures are one of the main areas of focus for program synthesis, along with finding good training data. However, I think that the results of this thesis show that the search algorithms and integration of different networks together are also an important facet of the solution to this problem. In particular, they seem well suited to program synthesis because of the unforgiving nature of the task, where results are both high dimensional and graded on what is effectively a form of exact match accuracy (though in semantic rather than syntactic space). It is thus often useful to explore several possibilities to have a good chance of funding a single usable program.

4.3 Future Work

In general, I think that the most interesting areas for future work involve explorations of different ways in which iterative techniques and reranking can be used in other domains within program synthesis. Additionally, there are probably many improvements that can be made in the way that training data is generated for the debugger, as it is largely being trained with the assumption that it is supposed to construct the correct program in a single shot when that is not in practice how it is being used. Some kind of policy gradient or perhaps value-based approach might be useful for determining the best course of action to take when constructing edits. Alternatively the best approach may be better ways to construct training data.

Additionally, I think that ensembling is a fascinating problem that can probably be approached from several different angles. The algorithms typically used in machine learning assume little similarity between the models except their input and output spaces, but in practice there are probably many more similarities that can be exploited, as their architectures are often quite similar.

Finally, I think that currently in the field of program synthesis from examples the relatively unlimited nature of the data is underutilized as a conceptual tool. In concept, having an infinite source of data makes overfitting impossible, and this should open the door up to many strategies that would be unheard of in fields where sample efficiency is required.

Bibliography

[1] Rajeev Alur et al. *Syntax-guided synthesis*. IEEE, 2013. URL: https://escholarship.org/content/qt1g67m7hp/qt1g67m7hp.pdf.

[2] Jimmy Ba and Brendan Frey. "Adaptive dropout for training deep neural networks". In: *Advances in neural information processing systems*. 2013, pp. 3084–3092. URL: http://papers.nips.cc/paper/5032-adaptive-dropout-for-training-deep-neural-networks.pdf.

[3] Colin Bannard and Chris Callison-Burch. "Paraphrasing with bilingual parallel corpora". In: *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics. 2005, pp. 597–604. URL: https://dl.acm.org/ft_gateway.cfm?ftid=964485&id=1219914.

[4] Jonathan Berant and Percy Liang. "Semantic Parsing via Paraphrasing". In: *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Baltimore, Maryland: Association for Computational Linguistics, June 2014, pp. 1415–1425. DOI: 10.3115/v1/P14-1133. URL: https://www.aclweb.org/anthology/P14-1133.

[5] Rudy Bunel et al. "Leveraging grammar and reinforcement learning for neural program synthesis". In: *arXiv preprint arXiv:1805.04276* (2018). URL: https://arxiv.org/pdf/1805.04276.pdf.

[6] Xinyun Chen, Chang Liu, and Dawn Song. "Execution-guided neural program synthesis". In: (2018). URL: https://openreview.net/pdf?id=H1gfOiAqYm.

[7] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018). URL: https://arxiv.org/pdf/1810.04805.pdf.

[8] Jacob Devlin et al. "Neural program meta-induction". In: *Advances in Neural Information Processing Systems*. 2017, pp. 2080–2088.

[9] Jacob Devlin et al. "RobustFill: Neural Program Learning under Noisy I/O". In: *ICML*. 2017.

[10] Li Dong and Mirella Lapata. "Coarse-to-fine decoding for neural semantic parsing". In: *arXiv preprint arXiv:1805.04793* (2018). URL: https://arxiv.org/pdf/1805.04793.pdf.

[11] Kevin Ellis et al. "Write, execute, assess: Program synthesis with a repl". In: *Advances in Neural Information Processing Systems*. 2019, pp. 9165–9174. URL: https://arxiv.org/pdf/1906.04604.pdf.

[12] Cheng Fu et al. "A Neural-based Program Decompiler". In: *arXiv preprint arXiv:1906.12029* (2019). URL: https://arxiv.org/pdf/1906.12029.pdf.

[13] Sumit Gulwani, William R Harris, and Rishabh Singh. "Spreadsheet data manipulation using examples". In: *Communications of the ACM* 55.8 (2012), pp. 97–105. URL: https://dl.acm.org/doi/pdf/10.1145/2240236.2240260.

[14] Emanuel Kitzelmann and Ute Schmid. "Inductive synthesis of functional programs: An explanation based generalization approach". In: *Journal of Machine Learning Research* 7.Feb (2006), pp. 429–454. URL: http://www.jmlr.org/papers/volume7/kitzelmann06a/kitzelmann06a.pdf.

[15] Fei Li and HV Jagadish. "Constructing an interactive natural language interface for relational databases". In: *Proceedings of the VLDB Endowment* 8.1 (2014), pp. 73–84. URL: https://dl.acm.org/doi/pdf/10.14778/2735461.2735468.

[16] Dianhuan Lin et al. "Bias reformulation for one-shot function induction". In: (2014). URL: https://dspace.mit.edu/bitstream/handle/1721.1/102524/Tenenbaum_Bias%20reformulation.pdf?sequence=1&isAllowed=y.

[17] Zohar Manna and Richard J Waldinger. "Toward automatic program synthesis". In: *Communications of the ACM* 14.3 (1971), pp. 151–165. URL: https://dl.acm.org/doi/pdf/10.1145/362566.362568.

[18] Renato Negrinho, Matthew Gormley, and Geoffrey J Gordon. "Learning beam search policies via imitation learning". In: *Advances in Neural Information Processing Systems*. 2018, pp. 10652–10661. URL: http://papers.nips.cc/paper/8264-learning-beam-search-policies-via-imitation-learning.pdf.

[19] Yusuke Oda et al. "Learning to generate pseudo-code from source code using statistical machine translation (t)". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 574–584. URL: https://dl.acm.org/ft_gateway.cfm?id=3343959&type=pdf.

[20] Emilio Parisotto et al. "Neuro-symbolic program synthesis". In: *arXiv preprint arXiv:1611.01855* (2016). URL: https://arxiv.org/pdf/1611.01855.pdf.

[21] Oleksandr Polozov and Sumit Gulwani. "FlashMeta: a framework for inductive program synthesis". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2015, pp. 107–126. URL: https://dl.acm.org/doi/pdf/10.1145/2814270.2814310.

[22] Veselin Raychev, Martin Vechev, and Andreas Krause. "Predicting program properties from" big code"". In: *ACM SIGPLAN Notices* 50.1 (2015), pp. 111–124.

[23] Eui Chul Shin, Illia Polosukhin, and Dawn Song. "Improving neural program synthesis with inferred execution traces". In: *Advances in Neural Information Processing Systems*. 2018, pp. 8917–8926.

[24] Richard Shin. "Encoding database schemas with relation-aware self-attention for text-to-SQL parsers". In: *arXiv preprint arXiv:1906.11790* (2019). URL: https://arxiv.org/pdf/1906.11790.

[25] Richard Shin, Illia Polosukhin, and Dawn Song. "Towards specification-directed program repair". In: (2018). URL: https://openreview.net/pdf?id=B1iZRFkwz.

[26] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. "Automated feedback generation for introductory programming assignments". In: *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 2013, pp. 15–26. URL: https://dl.acm.org/doi/pdf/10.1145/2491956.2462195.

[27] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. "From program verification to program synthesis". In: *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2010, pp. 313–326. URL: https://dl.acm.org/doi/pdf/10.1145/1706299.1706337.

[28] Richard J Waldinger and Richard CT Lee. "PROW: A step toward automatic program writing". In: *Proceedings of the 1st international joint conference on Artificial intelligence*. 1969, pp. 241–252. URL: http://papers.nips.cc/paper/5032-adaptive-dropout-for-training-deep-neural-networks.pdf.

[29] Sam Wiseman and Alexander M Rush. "Sequence-to-sequence learning as beam-search optimization". In: *arXiv preprint arXiv:1606.02960* (2016). URL: https://arxiv.org/pdf/1606.02960.

[30] Pengcheng Yin and Graham Neubig. "Reranking for neural semantic parsing". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 2019, pp. 4553–4559. URL: https://www.aclweb.org/anthology/P19-1447.pdf.

[31] Tao Yu et al. "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task". In: *arXiv preprint arXiv:1809.08887* (2018). URL: https://arxiv.org/pdf/1809.08887.

[32] John M Zelle and Raymond J Mooney. "Learning to parse database queries using inductive logic programming". In: *Proceedings of the national conference on artificial intelligence*. 1996, pp. 1050–1055. URL: https://www.aaai.org/Papers/AAAI/1996/AAAI96-156.pdf.

[33] Victor Zhong, Caiming Xiong, and Richard Socher. "Seq2sql: Generating structured queries from natural language using reinforcement learning". In: *arXiv preprint arXiv:1709.00103* (2017). URL: https://arxiv.org/pdf/1709.00103.pdf.

[34] Amit Zohar and Lior Wolf. "Automatic program synthesis of long programs with a learned garbage collector". In: *Advances in Neural Information Processing Systems*. 2018, pp. 2094–2103. URL: https://arxiv.org/pdf/1809.04682.pdf.