Improved Deep-Coder Implementation

KAVEH ESKANDARI, Tehran Institute for Advanced Studies, Iran
MOHAMMAD HOSSEIN MOTI, Tehran Institute for Advanced Studies, Iran

DeepCoder[1] is a general framework for synthesizing Competitive Programming level problems using a combination of a Neural Network model and Depth First Search/Enumerative search. In this work, we first aim to study the properties of this framework as described in its paper. Additionally, we attempt to implement an improved version of DeepCoder that better suits the technological improvements that have happened in the field of Artificial Intelligence and Deep Learning. To achieve our goal, we modify the Neural Network architecture proposed by the authors as well as make slight changes to the Dataset in order to improve the accuracy and general performance of the Deep Learning model.

ACM Reference Format:
Kaveh Eskandari and Mohammad Hossein Moti. 2021. Improved Deep-Coder Implementation. 1, 1 (July 2021), 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The significant advances that have been made in different fields of Machine Learning and Deep Learning have made it possible for this models to be used in broad ranges of other fields. One such field is Software Synthesis; which, generally speaking, given some specifications, using various methods, strives to find a program in a programming language that satisfies the aforementioned specifications.

A popular and fairly user-friendly choice for describing the program specifications is example based specification, which if used in a synthesizer, is commonly known as Programming By Example(PBE). In this method, we ask the user to provide some number of Input-Output examples to the synthesizer, which in turn, by searching over the program space tries to construct a program that if applied on the set of inputs provided by the user, generates the set of outputs. As an example, consider the Input-Output example $i = [1, 2, 3, 4], o = [4, 8, 12, 16]$. By providing this set of inputs and outputs to the synthesizer, it should, in an ideal case, generate program that applies the operation $4 \times x$ for every member x of a given integer.

However, even for small Domain Specific Languages(DSL), the search space can grow astronomically large. This both limits the efficiency of the search algorithm -which now must search over very large number of function combinations to finally reach a working program.- and its ability to find a program that truly satisfies our needs as it is possible for the search algorithm to find a combination of functions that work on the examples provided but do not capture the intention of the user.
Consider the previous input and example for instance. Given that example, the code below also satisfies the specifications.

Authors' addresses: Kaveh Eskandari, kaveeskandari96@gmail.com, Tehran Institute for Advanced Studies, Tehran, Tehran, Iran; Mohammad Hossein Moti, m.motie@khatam.ac.ir, Tehran Institute for Advanced Studies, Tehran, Tehran, Iran.

```
if input = [1,2,3,4] :
    output = [4,8,12,16]
    return output
```

In order to mitigate such issues, DeepCoder proposes a Neural Network model trained on Input-Output examples, which given some number of Input-Ouput examples, finds a label map corresponding to the number of functions in the programming language. This label map indicates whether a function should be present in the final code generated by the synthesizer. It is clear that if a well performing model can be trained, the search space can potentially be pruned drastically as we can use this information to guide the search towards the functions that our model thinks should be used in the final program. This is exactly what DeepCoder does.

In the next sections, we first briefly describe the inner workings of DeepCoder and changes that we have made in order to make it more dependable.

2 BACKGROUND

We begin by providing some information about the key concepts of the Synthesis field. Then we describe how DeepCoder works briefly. As mentioned in the previous section, the DeepCoder uses Inductive Program Synthesis (IPS). So we will discuss the main components of IPS with respect to DeepCoder. We need to introduce a general approach that DeepCoder follows in its working, Learning Inductive Program Synthesis (LIPS). If we want to describe IPS in one sentence we can say: generating programs that are consistent with given input-output examples. Each IPS system needs to include a domain-specific language, search technique, and ranking approach. In addition to these components, DeepCoder also has a neural network component to do a guided search on program space. We know that DeepCoder has a Neural Network part. We need to discuss how is its dataset structure and generation. Let's investigate each item separately and see how DeepCoder uses these components.

2.1 Domain Specific Language

The choice or design of a DSL is one of the most vital parts of creating a Synthesis system. It should be restricted to prevent an explosion of search space and be expressive enough to solve provided problems. DeepCoder introduces an attribute function A that maps valid programs in DSL to an attribute vector. Attribute vectors are links between the Neural Network component(machine learning component) and the search component of the system. Some possible attributes are the appearance of high-level functions in the program and control flow templates like the count of loops and conditions. In DeepCoder DSL, there are two major function types: first-order functions like $HEAD, LAST, SORT, SUM$, and higher-order functions that take functions as inputs like MAP and $ZipWith$. This DSL generates a functional language. A program in our DSL is a sequence of function calls. In this work, we will represent the length of the program with T. The program in $figure1$ has four functions $T = 4$. This example takes a list of integer inputs and filters elements lower than zero. Then multiplies each list element with four and does a descending sort.

2.2 Data Generation

The dataset structure is in the form $(P(n), a(n), C(n))$ where P is a program in the DSL, a is its attributes and, C is a list of input-output examples for that program. We must formulate very well to ensure that this approach is feasible to generate a dataset with millions of programs. To generate a dataset, in the first step, we enumerate programs in DSL heuristically and prune the programs with obvious issues. Then we generate appropriate inputs for programs by specifying some output value

```
a ← [int]
b ← FILTER (<0) a
c ← MAP (*4) b
d ← SORT c
e ← REVERSE d
```

Fig. 1. Example Program

bounding integers. These boundaries propagate throughout the program in a backward direction to obtain proper margins for input integers.

2.3 Neural Network

As described above, the idea is to convert a difficult search problem to a supervised machine learning problem. The aim is to calculate the distribution of attributes given input-output examples $q(a|C)$. As attributes have a fixed-size binary vector, we use a neural network with independent *Sigmoid* outputs. We can use RNNs for variable-size vectors. The neural network includes an encoder and decoder to predict the probability of the presence of functions in the program. You can see the block diagram of the model in Figure 2.

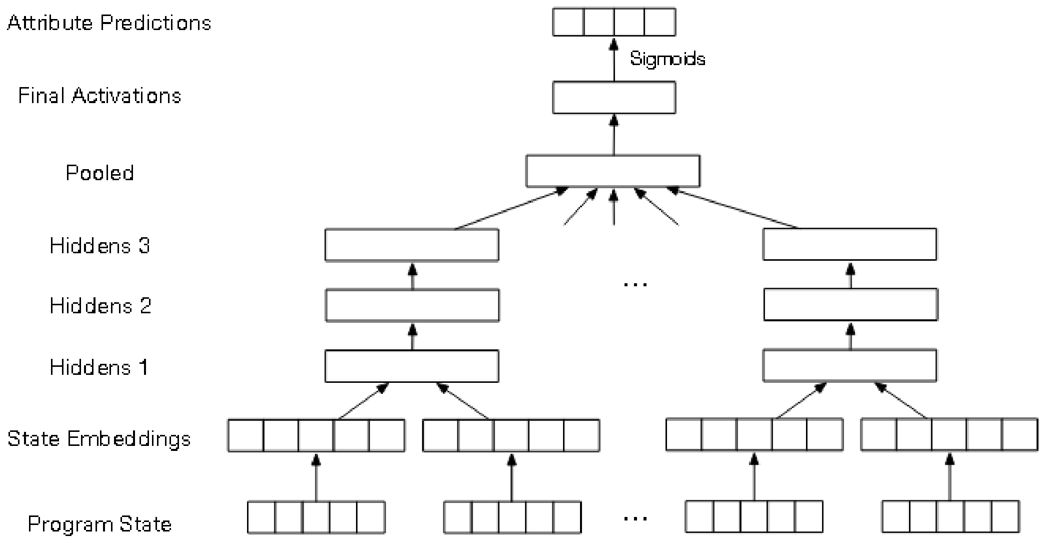


Fig. 2. Original DeepCoder Neural Architecture

As you can see we pass all five input-output examples as input through the neural network. We pad these inputs and outputs to the maximum length of L = 20 using a special NULL character. Then each of them passes through an embedding layer that has E = 20 dimensions. By concatenating each pair of input and output embedding, the neural network passes them through three dense layers with sigmoid as activation function. At the final step, it uses average pooling to get the best information from all of the inputs and pass them through the final dense layer. The result is an attribute vector with a size = 34 that predicts the presence of functions in the final program. The predicted list has the same order as the following functions list. Each element represents the probability of the presence of that function in the output.

2.4 Search Technique

The idea to search is to use DFS. DeepCoder uses an optimized version of DFS that is implemented in the C++ programming language. The output of the neural network will guide DFS by the "Sort and add" technique. This approach will search for programs shorter than the given T, so the maximum height of the DFS tree will be T. The optimized guided version of DFS will create an active set of functions by selecting more probable functions in order using neural network output. Then it will call DFS recursively on each function in the active set. Each time a new function is added to ancestor functions, the algorithm should run the partial program on all of the five given examples. If the generated output was equal to expected outputs then that partial program is a complete program which is also the answer. Otherwise, the algorithm continues until the maximum depth is reached. The complementary approach is to treat functions and their arguments as holes and use Sketch to get values that satisfy given input-output examples constraint for these holes.

3 IMPROVED DEEPCODER

3.1 Dataset Improvement

The original Dataset used by DeepCoder treats each program as an example for the Neural Network. Meaning that if for instance, a program in Dataset has 5 Input-Output examples going along with it, all 5 are considered parts of a single input into the Neural Network model. We argue that this approach has two main drawbacks.

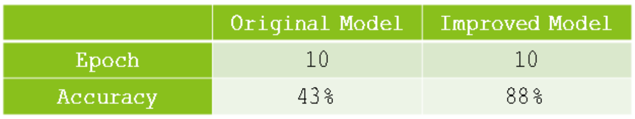
(1) Treating all examples of a program as components of a single input drastically decreases the size of our dataset. Considering that Neural models require large amounts of data to train well, this approach either leaves us with a small dataset which is less than ideal, or forces us to enumerate over a considerably larger number of programs to achieve a sufficiently large dataset.
(2) By concatenating all examples into a single input, we are effectively making it harder for the model to deduce this final input as multiple examples. We believe that this approach makes it harder for the model to learn as it both reduces the number of duplicate label maps and forces it to also learn a mapping from multiple examples to a convoluted input. It is better to keep the input as simple as possible.

In order to solve both of these problems, we construct a second dataset from the original dataset by breaking each item into number of examples in that item. Meaning that each input-output example is fed separately into the model. It is clear that this approach immediately solves the first issue as it multiplies the original dataset size by the number of examples used for each program. However, we argue that our approach to constructing the dataset also mitigates the second issue as now we are only required to concatenate a single input to a single output as the final input to our Neural Model. This should reduce the complexity of our inputs and make it easier for the model to learn label maps.

Additionally, we refrain from shuffling the final dataset in order to keep all examples corresponding to a single program next to each other. We believe that this decision has almost the same effect as having all examples as a single input. However, the impact of this decision on speed and quality of the learning process is not clear and further experiments must be done.

3.2 Neural Network Improvement

As stated earlier, the Neural architecture used in the original DeepCoder exclusively makes use of $Sigmoid(x) = \frac{1}{1+e^{-x}}$ as the non-linearity function for intermediate layers. However, it has since been shown empirically that in most cases, the function $ReLU(x) = Max(0, x)$ yields better results in comparison to Sigmoid function. Hence, we have changed the activation functions of all intermediate dense layers to ReLU. A comparison of performance boost in accuracy can be seen in $figure 3$ which compares the accuracy of the model in 10 epochs using ReLU function with the accuracy of the same model in 10 epochs using Sigmoid function.



	Original Model	Improved Model
Epoch	10	10
Accuracy	43%	88%

Fig. 3. Comparison of Activation Functions

Next, although larger models are somewhat more prone to over-fitting, they also boast higher learning power, meaning that they can better learn to map complex inputs to labels. Due to this, we have decided to to change the number of hidden units at each intermediate layers to 512 from the previous 256 hidden units.
The final architecture of our model can be seen in $figure 4$.

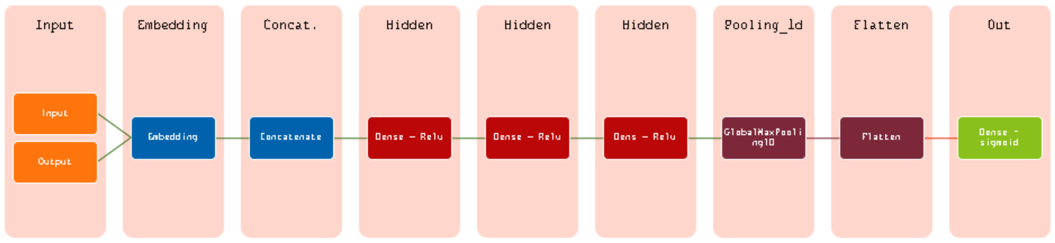
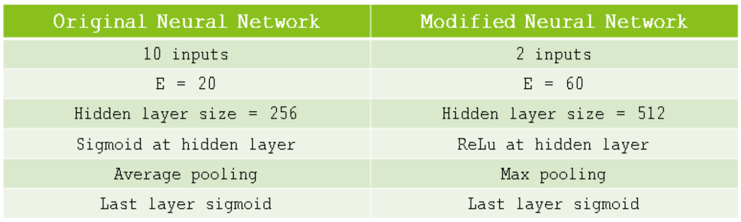


Fig. 4. Neural Network Architecture

Furthermore, as we are using single Input-Output example per dataset element, usage of Average Pooling is not justified anymore and thus we change our pooling layer to Max Pooling to capture the most important features of the final vector.
Note that the final Sigmoid layer stays the same as we are doing classification on multiple possible *True* labels so we need independent probability distribution over each of them. A comparison of the original model and our proposed model can be viewed in $figure 5$.



Original Neural Network	Modified Neural Network
10 inputs	2 inputs
E = 20	E = 60
Hidden layer size = 256	Hidden layer size = 512
Sigmoid at hidden layer	ReLu at hidden layer
Average pooling	Max pooling
Last layer sigmoid	Last layer sigmoid

Fig. 5. Comparison of the Original Model and the Proposed Model

3.3 Search Technique

The search method used is similar to the depth based enumeration method seen in the original paper. We first run the examples provided by the user through the Neural model, yielding n label maps for n examples provided. Next, in order to mitigate prediction failures by the model, we do an element-wise average on all the label maps, achieving a single final label map to be used in the search process.
The aforementioned label map corresponds to the list of functions that are available through the DSL. Then, for a predetermined depth T, we enumeratively generate functions with respect to their ranking in the label map provided such that if at any level, we reach a first-order function, we evaluate the current partial program against a single example from the examples provided by the user. Else, we continue the search on the next level. If at any point, our partial program satisfies the example, it is regarded as the solution and returned to the user. If all possible combinations are searched with no satisfying results, *None* is returned.

Our implementation is noteworthy from two aspects.
Firstly, note that the order in which the search is done is based on the probabilities generated by the Neural Network model, naturally causing the functions that are more likely to appear be tested first. With a strong machine learning model, this can dramatically decrease the search space as it is highly probable that the few first functions explored are also the answer to the problem.
Secondly, note that we test the generated program against a single example rather than all of the examples. This approach heavily relies on the performance of the Neural Network model. The logic behind this course of action is that by generating a label map for all the examples provided by the user, the label map generated naturally corresponds correctly with the functions needed to satisfy all examples. Thus we believe that given a strong Neural Network model, the first complete program to satisfy a single example is also the program that can satisfy all other examples. This approach is not complete with regards to other examples as an error in predicting the label map can cause the search technique to yield an incorrect solution. However, the speed gain is significant as effectively $Num(programs) \times Num(examples)$ evaluations is reduced to only $Num(programs)$ evaluations and with a strong enough Neural Network model, this approach can also be almost complete as the predictions will be adequately accurate.

3.4 Examples

The model that we did the tests on has been trained on programs with the depth of at most $T = 2$. The reason behind this choice is that the effectiveness of the Neural Network model decreases as examples get more complex, converging into a semi-guided enumerative search as T increases. Furthermore, our assumption that the first program to satisfy a single example is also the correct program does not hold anymore due to the degradation of performance by our Deep Learning model. We believe that this assumption can still be used for more complex examples, but a stronger prediction model is required to make it work.

First, the outputs of the Deep Learning models, one using *Sigmoid* activation functions and the other using *ReLU* ones are compared for the following program in *Figure*6.

```
LIST|INT|ACCESS(INT,LIST)
```

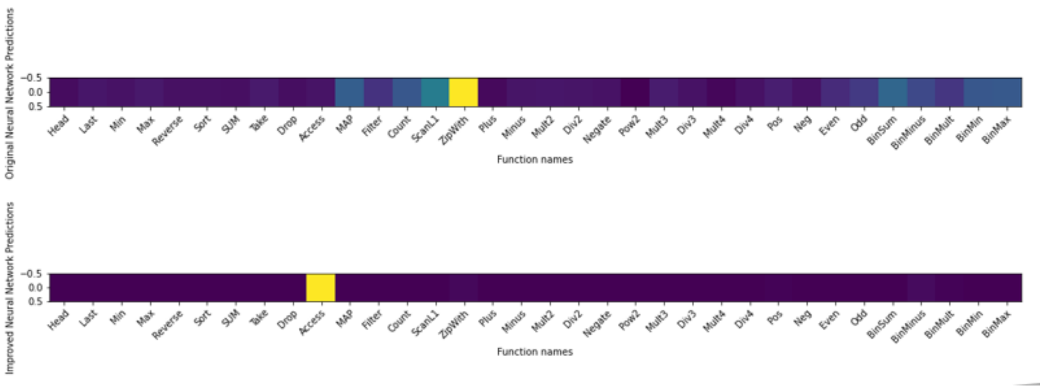


Fig. 6. Prediction Results

It is observable that the model trained on *ReLU* function outperforms the model trained on *Sigmoid* function as the former correctly predicts the presence of a single *ACCESS* function while the latter fails to do so, predicting the unrelated function *ZIPWITH* as the most probable one.

Next, consider the program

```
LIST|LIST|ZIPWITH|Min 0, 1
```

By providing the the Neural Network model with 5 input-output examples

```
{
        "inputs": [
           [194, -125, 9, 31, 74, -1, -201, -167, -153, -54, 151, 224, -16, -113, 82,
                -136, -83, 98, -150],
           [-250, -101, -115, -167, 239, 10, 195, 78]
        ],
        "output": [-250, -125, -115, -167, 74, -1, -201, -167]
     },
     {
        "inputs": [
           [239, 87, -29, 47, 130, 0, -78, -16],
           [75, -153, -52]
        ],
        "output": [75, -153, -52]
     },
     {
        "inputs": [
           [-36, 103, 150, 70, -240, 148, -226],
           [9, 237, -189, 163, -7, 25, 160, 204, 164, 143, -182, -174, -236, -63, -188]
        ],
        "output": [-36, 103, -189, 70, -240, 25, -226]
     },
     {
        "inputs": [
           [-242, -205, -85, -179, -115, 104, -220, -153, 136],
           [-27, 98, -77, 170, 182, 247, -101, -77, -50, -228, -247, -53, 75, 70, 210,
                16, 128, 50, 27]
        ],
        "output": [-242, -205, -85, -179, -115, 104, -220, -153, -50]
     },
     {
        "inputs": [
           [32, -202, 152, 94, 30, 160, -97, -251, -101, 103, 7, -233, -228, 18, 235,
                9, 237],
           [-135, 219]
        ],
        "output": [-135, -202]
     }
```

We are returned the label map

```
[0.04112128317137831, 0.00018591539023295593, 3.3414251706589323e-06,
    4.7933140412114285e-05, 9.257456026882475e-05, 0.0004583842929791378,
    0.005295833002224508, 8.607594799573774e-05, 2.1163176578891588e-05,
    0.0006085693427557715, 1.3210548615166795e-06, 7.96624299964006e-06,
    0.010354584771178286, 3.166187525494024e-05, 0.4098999959727128,
    5.342804433894344e-05, 6.468804549507092e-06, 6.699733033229312e-05,
    6.4707995382680865e-06, 2.3196330785254147e-06, 4.764181726257948e-10,
    2.8567555154323007e-05, 2.1431186098652688e-05, 5.9249342343710896e-05,
    5.4615117047770535e-05, 0.0023860770365177566, 0.0014578700746708019,
    0.002336806784074174, 0.0001492104801431894, 0.09499136255302194,
    0.04606447430948416, 0.00014376110501264824, 0.3184212847433325,
    0.0014034292838308222]
```

in which each element is the probability of a single function.
By providing this mapping to the search function, along with a single example from the example set, we are returned the function list

```
[<function ZipWith at 0x7f303c67f0e0>, <function BinMin at 0x7f3038c48950>]
```

Which indicates that the higher-order function *ZipWith* should be called with the first-order function *BinMin* as a parameter to it. Checking this result with the above ground truth program, we see that it is indeed correct and two programs conform to each other.
By analyzing the label map returned by the Neural model ($figure7$), we observe that the model correctly predicts that functions *ZipWith* and *BinMin* should be in the final program. Some errors are present however, as some other functions also have rather high probabilities.

More examples can be found in the dataset as well as the presentation slides along with their generated programs.

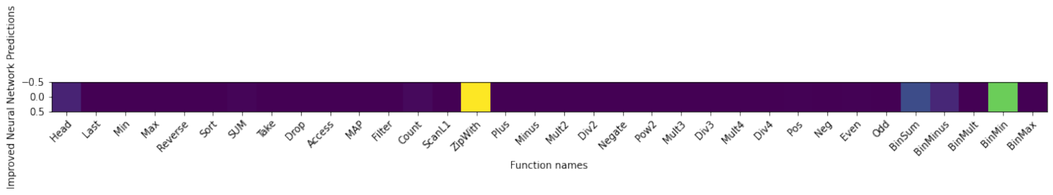


Fig. 7. Example Label Map

4 CONCLUSION

DeepCoder has shown significant potential as it showcases the capacity of even simple Deep Learning models to guide and quicken the process of inferring programs from examples. And as the field of Deep Learning is improving rapidly, there are many ideas that can be used along with Program Synthesis methods to solve open synthesis problems. Owing to this, further research can be done on usage of modern, large pre-trained models and their combination with synthesis approaches in order to further increase the accuracy of Synthesizers.

5 REFERENCES

[1] Balog, Matej, et al. "DeepCoder: Learning to Write Programs." ArXiv:1611.01989 [Cs], Mar. 2017. arXiv.org, http://arxiv.org/abs/1611.01989.

6 APPENDIX

- The code is currently available at Google Colab. However, it should be cleaned and moved to Github in a few days along with an easy to use user interface. The link to the Github containig the code will also be added to Google Colab Notebook.
- The generated dataset containing code examples can be downloaded from Link