DeepCoder: Learning to Write Programs

ICLR conference paper by Balog et al., 2017

Presentation by Harry Smith
Background Work & Main Ideas
“A dream of artificial intelligence is to build systems that can write computer programs”

Balog et al., 2017
Program-like neural network models
(Neural Turing Machines)

Generating source code from unstructured text descriptions
(Latent Predictor Networks for Code Generation)

Solving Inductive Program Synthesis
(DeepCoder: Learning to Write Programs)
Neural Turing Machines:

Is it even possible to have a machine learning model represent a program?

Working up to AI-Written Computer Programs

2014
- Program-like neural network models
  (Neural Turing Machines)

2016
- Generating source code from unstructured text descriptions
  (Latent Predictor Networks for Code Generation)

2017
- Solving Inductive Program Synthesis
  (DeepCoder: Learning to Write Programs)
Latent Predictor Networks for Code Generation

```python
class DivineFavor(SpellCard):
    def __init__(self):
        super().__init__("Divine Favor", 3,
                         CHARACTER_CLASS.PALADIN, CARD_RARITY.RARE)

    def use(self, player, game):
        super().use(player, game)
        difference = len(game.other_player.hand) - len(player.hand)
        for i in range(0, difference):
            player.draw()
```

Figure 4: Generation process for the code `init('Tirion Fordring', 8, 6, 6)` using LPNs.
Working up to AI-Written Computer Programs

2014
Program-like neural network models
(Neural Turing Machines)

2016
Generating source code from unstructured text descriptions
(Latent Predictor Networks for Code Generation)

2017
Solving Inductive Program Synthesis
(DeepCoder: Learning to Write Programs)
Main Ideas
Main Ideas

- Learn to induce programs
  - Formulated as a **big data problem**: going from input-output pairs to code
  - Learn strategies that generalize across problems

```python
def rev_sort(in):
    return reversed(sorted(in))
```
The Traditional Differential Interpreter Approach

What is a potential problem with this approach?
Main Ideas

- Integrate NN architectures with search-based techniques
  - No need to replace search!
  - Use the power of differential interpreters for multiple synthesis problems!
Main Ideas

● Learn to induce programs
  ○ Formulated as a **big data problem**
  ○ Learn strategies that generalize across problems

● Integrate NN architectures with search-based techniques
  ○ No need to replace search!
  ○ Use the power of differential interpreters
Inductive Program Synthesis (IPS)
The Basics of IPS

IPS is the problem of taking input-output examples and producing a program that has behavior consistent with the examples.

● How do we find consistent programs?

● How do we choose the best programs when there are multiple options?
The Basics of IPS

IPS is the problem of taking input-output examples and producing a program that has behavior consistent with the examples.

● How do we find consistent programs?
  ○ Requires a well defined set of acceptable programs that creates the search space
  ○ We also need an intelligent search procedure to move through this space

● How do we choose the best programs when there are multiple options?
  ○ Choose the shortest program?
  ○ Choose the first program found, with simple normalizations made?
Need to choose a Domain Specific Language (DSL) in which to synthesize problems

- Language should be useful in a certain domain; otherwise, the synthesis is useless.
- Language should be restricted to limit the search space
  - Can’t search over all C++ programs!
  - Features like loops or ifs make more solutions consistent
Formulating an Approach to IPS: DSLs

DeepCoder’s solution: an SQL-like query language

- Includes ints, int arrays, and booleans* as types
- Program is a sequence of function calls storing results in new variable names
- 34 functions are available, including first-order, higher-order, and lambda functions
  - No explicit control flow
  - The permissible lambdas are finite and enumerated, e.g. (<0) and (*4)

An input-output example:

Input:
[[-17, -3, 4, 11, 0, -5, -9, 13, 6, 6, -8, 11]]

Output:
[-12, -20, -32, -36, -68]
Examples of programs written in the DSL, with the natural language descriptions.


<table>
<thead>
<tr>
<th>Program 0:</th>
<th>Input-output example:</th>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>k ← int</td>
<td>Input:</td>
<td>A new shop near you is selling ( n ) paintings. You have ( k &lt; n ) friends and you would like to buy each of your friends a painting from the shop. Return the minimal amount of money you will need to spend.</td>
</tr>
<tr>
<td>b ← [int]</td>
<td>Output:</td>
<td>2, [3 5 4 7 5]</td>
</tr>
<tr>
<td>c ← SORT b</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>d ← TAKE k c</td>
<td>[7]</td>
<td></td>
</tr>
<tr>
<td>e ← SUM d</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program 1:</th>
<th>Input-output example:</th>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>w ← [int]</td>
<td>Input:</td>
<td>In soccer leagues, match winners are awarded 3 points, losers 0 points, and both teams get 1 point in the case of a tie. Compute the number of points awarded to the winner of a league given two arrays ( w, t ) of the same length, where ( w[i] ) (resp. ( t[i] )) is the number of times team ( i ) won (resp. tied).</td>
</tr>
<tr>
<td>t ← [int]</td>
<td>Output:</td>
<td>[6 2 4 7 9], [5 3 6 1 0]</td>
</tr>
<tr>
<td>c ← MAP (*3) w</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>d ← ZIPWITH (+) c t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e ← MAXIMUM d</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program 2:</th>
<th>Input-output example:</th>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a ← [int]</td>
<td>Input:</td>
<td>Alice and Bob are comparing their results in a recent exam. Given their marks per question as two arrays ( a ) and ( b ), count on how many questions Alice got more points than Bob.</td>
</tr>
<tr>
<td>b ← [int]</td>
<td>Output:</td>
<td>[6 2 4 7 9], [5 3 2 1 0]</td>
</tr>
<tr>
<td>c ← ZIPWITH (-) b a</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>d ← COUNT (&gt;0) c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Formulating an Approach to IPS: Search Strategy

Need to solve the problem of how to find the code that satisfies the examples

- The goal of DeepCoder is to interface with an existing solver, using predicted attributes as a guide for search
  - More on this in a moment...
- Once a method of predicting attributes of a program is fixed, DeepCoder can interface with the following:
  - DFS: Try all programs up to a certain length, and add the estimated most likely function at each step
  - “Sort and add”: Try all programs with the estimated most likely k functions, then on failure try again with the k + 1 most likely, etc.
  - Sketch: an SMT-based tool, which can incorporate program attributes into its own “sort and add” strategy
  - $\lambda^2$: a tool that enumerates the search space, using deductive pruning. Designed for small functional programs on data structures! Also leverages “sort and add” from predicted attributes.
A word on Sketch and SMT solvers...

Recall that Satisfiability Modulo Theories (SMT) solvers are SAT solvers with theories like “algebra” or “inequalities”

\[ X < Y; Y < -100 \]

For example, an arbitrary search strategy over all integers might take quite some time to find a satisfying X, Y pair.

An SMT solver with algebra and inequality solves this almost instantly.
A word on Sketch and SMT solvers...

Sketch models an IPS problem as a hole in a code base that needs to be filled in using a set of available functions. Each function is transformed from its original definition to a mapping from constraints:

```python
def plus_one(a):
    return a + 1

plus_one(a_min, a_max, start_inclusive, end_inclusive):
    return a_min + 1, a_max + 1, start_inclusive, end_inclusive
```
A word on Sketch and SMT solvers...

“Constraintified” Sample Program with hole in it

A <- int array  
all of A in [-10, 10]
B <- ???????  
???????????
C <- map (+1) B  
all of C in [-19, 21]

Function Library

plus_one(a_min, a_max, start_inclusive, end_inclusive):
  return a_min + 1, a_max + 1, start_inclusive, end_inclusive

minus_one(a_min, a_max, start_inclusive, end_inclusive):
  return a_min - 1, a_max - 1, start_inclusive, end_inclusive

times_two(a_min, a_max, start_inclusive, end_inclusive):
  return a_min *2, a_max *2, start_inclusive, end_inclusive

Find the function that takes the constraints in step A to the constraints after step C...
Formulating an Approach to IPS: Ranking

Need to pick the “best solution” of those which are discovered in search

IN: [-10, 3, 10]  OUT: 10
IN: [7, 18, 10]  OUT: 18
IN: [1, 4, 0, 3]  OUT: 4
Formulating an Approach to IPS: Ranking

Need to pick the “best solution” of those which are discovered in search

\[
\begin{align*}
\text{IN: } [4, 3, 10] & \quad \text{OUT: } 10 \\
\text{IN: } [7, 18, 10] & \quad \text{OUT: } 18 \\
\text{IN: } [1, 4, 0, 3] & \quad \text{OUT: } 4
\end{align*}
\]

```
def f(arr): return max(arr)
```

```
def f(arr): return 1 + 1 + 1 + 1 + min(arr) + min(arr)
```

```
def f(arr): return 4 + min(arr) * 2
```
DeepCoder focuses on the search aspect of IPS, and does not define their ranking strategy.

- Possible candidates:
  - Shortest is best
  - Max-margin prediction
    - Assign scores so that ground-truth programs are always scored higher than induced examples.
  - Others?
Learning Inductive Program Synthesis (LIPS)
Recall that we need to link machine learning and search to efficiently synthesize programs.

How do we determine program attributes?

Program Attributes

Search

A <- SORT(X)
B <- REVERSE(A)
C <- ???

A <- SORT(X)
B <- REVERSE(A)
C <- MAP (+ 1) B
How do we determine program attributes?

From the input-output pairs!

Formally, we want to define an attribute function $A$ which maps programs $A$ into finite attribute vectors $a$:
How do we determine program attributes?

From the input-output pairs!

This way, given a set of input-output examples $E$, we can compute a distribution $q(a \mid E)$. Then, we search over programs $P$ ordered by $q(A(P) \mid E)$. 
Given a set of input-output examples $E$, we can compute a distribution $q(a \mid E)$. Then, we search over programs $P$ ordered by $q(A(P) \mid E)$. 

![Diagram showing distribution $q(a \mid E)$ with programs $P$, $P'$, and $P''$, and their corresponding distributions $q(a = A(P') \mid E)$, $q(a = A(P'') \mid E)$, and $q(a = A(P) \mid E)$]
Turning Programs into Attributes

So what are these attributes?

- The identity function: $A(P) = P$?
- Control Flow Templates?
  - # of loops
  - # of conditionals
- Presence or absence of high-level functions?
  - Does program $P$ ever use “SORT”?
  - Does program $P$ end with a call to “MAP”?

An input-output example:

**Input:**

$$[-17, -3, 4, 11, 0, -5, -9, 13, 6, 6, -8, 11]$$

**Output:**

$$[-12, -20, -32, -36, -68]$$
Turning Programs into Attributes: Data Generation

How do we formulate this as a big data problem?

- Data Generation procedure:
  - Enumerate programs in the DSL
    - Prune those with redundant variables/instructions
  - Generate valid inputs for a program
    - Enforce constraints on the output value(s)
    - Propagate constraints backwards up to input
  - Select inputs from the valid range and execute the program to get outputs.
    - If the input set is empty, discard the program
  - Read off the attribute vector from the program itself
A <- int array
B <- map (+1) A
C <- map (*4) B
D <- C

all of A in []
all of B in []
all of C in []
all of D in [1, 3]
Example of Data Generation: Rejecting an example program

\[
A \leftarrow \text{int array} \\
\text{all of } A \text{ in } [-2, 2] \\
B \leftarrow \text{filter } (>0) A \\
\text{all of } B \text{ in } [-2, 2] \\
C \leftarrow \text{map } (*4) B \\
\text{all of } C \text{ in } [-2, 2] \\
D \leftarrow C \\
\text{all of } D \text{ in } [-10, 10]
\]

\[
A \leftarrow [1, 1, 0, 2] \\
B \leftarrow \text{filter } (>0) A \\
[1, 1, 2] \\
C \leftarrow \text{map } (*4) B \\
[4, 4, 8] \\
D \leftarrow C \\
\text{output } = [4, 4, 8]
\]

<\text{has_filter} : 1, \text{has_map} : 1, \text{has}_{>0} : 1, \text{has}_{*4} : 1>
How are the attributes learned?

- A feed-forward neural network learns the mapping from input-output examples to attributes.
- The FF-NN contains two fundamental components
  - An encoder: a differential mapping from $M$ input-output examples to a latent, real-valued vector
  - A decoder: a differential mapping from the latent vector to predictions on the ground truth’s attributes
The Encoder

1. Represent the input/output types (singleton or array) as one-hot vectors
   ○ // E.g. <{0,1}, {1,0}> for a function that takes in an array and outputs an integer.

2. Pad all inputs and outputs to a maximum length $L$ with null value

3. Map all integers to an $E = 20$ dimensional embedding
   ○ // If the input to a problem is a length $L$ array, the dimensionality of the input becomes $E*L$

4. For each input-output example...
   ○ ...concatenate the embeddings of input types, output types, inputs, and outputs into a single vector
   ○ ...pass this vector through $H = 3$ hidden layers of $K = 256$ sigmoid units each

5. Take the arithmetic mean of all output vectors (one for each input-output pair) as the output of the encoder
   ○ // note that this output vector has dimensionality $K$
Embedding of integers

- No justification is given for the choice of $E = 20$ as the dimension for the embedding of the integers...
  
  - This is less than the total number of functions available
  - This is four times larger than the length of the programs on which they ran their experiments
  - Perhaps the break between 20 and 21 is where the NN was “no longer simple to train”

- Balog et al. initially experimented with values of $E = 2$ for programs of length $T = 1$ and found the result at right:

Figure 8: A learned embedding of integers $\{-256, -255, \ldots, -1, 0, 1, \ldots, 255\}$ in $\mathbb{R}^2$. The color intensity corresponds to the magnitude of the embedded integer.

The Decoder

1. Pre-multiply the averaged vector \((K \times 1)\) by a final “decoding” matrix of dimension \(C \times K\), where \(C = 34\), the number of functions in the DSL.

2. Interpret the \(C \times 1\) resulting vector as the log-unnormalized probabilities of each function appearing in the source code.

Figure 2: Neural network predicts the probability of each function appearing in the source code.
The full architecture

Figure 7: Schematic representation of our feed-forward encoder, and the decoder.
Figure 7: Schematic representation of our feed-forward encoder, and the decoder.

The full architecture (dimensionality tracked in orange)
How are the attributes learned?

- The NN is trained using negative cross entropy loss
- The outcome is the joint distribution $P(a \mid E)$ of all attributes given the input-output examples
  - The distribution is only over single function presence or absence
  - Likely some power lost by ignoring the correlations among the functions
Experimental Results
Experiment #1.1

1. Train the neural network on programs of length $T = 3$.
2. Create a test set of $P = 500$ programs, each of length $T$.
   a. For each generated program, create 5 input-output examples
   b. Ensure that each program in the test set is semantically distinct from all other examples
3. Produce attribute vectors for each program using the input-output pairs.
4. Use the attribute vectors with different search strategies to induce programs
   a. Time the search process
   b. Timeout after $10^4$ seconds
   c. Search space $\sim 10^6$
5. Compare the speeds of the attribute-guided searches to a baseline
   a. Baseline strategy is considering the probability of each function appearing to be the overall probability of that function appearing in the dataset
Experiment #1.1

Table 1: Search speedups on programs of length $T = 3$ due to using neural network predictions.

<table>
<thead>
<tr>
<th>Timeout needed to solve</th>
<th>DFS</th>
<th>Enumeration</th>
<th>$\lambda^2$</th>
<th>Sketch</th>
<th>Beam</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20%</td>
<td>40%</td>
<td>60%</td>
<td>20%</td>
<td>40%</td>
</tr>
<tr>
<td>Baseline</td>
<td>41ms</td>
<td>126ms</td>
<td>314ms</td>
<td>80ms</td>
<td>335ms</td>
</tr>
<tr>
<td>DeepCoder</td>
<td>2.7ms</td>
<td>33ms</td>
<td>110ms</td>
<td>1.3ms</td>
<td>6.1ms</td>
</tr>
<tr>
<td>Speedup</td>
<td>15.2×</td>
<td>3.9×</td>
<td>2.9×</td>
<td>62.2×</td>
<td>54.6×</td>
</tr>
</tbody>
</table>

If the time taken for four programs is 3s, 2s, 1s, 3s, then the timeout needed to solve 50% of problems is 2s.
Experiment #1.2

1. Train the neural network on programs of length $T = 4$.

2. Create a test set of $P = 100$ programs, each of length $T = 5$.
   a. For each generated program, create 5 input-output examples
   b. Ensure that each program in the test set is semantically distinct from all other examples

3. Produce attribute vectors for each program using the input-output pairs.

4. Use the attribute vectors with different search strategies to induce programs
   a. Time the search process
   b. Timeout after $10^4$ seconds
   c. Search space $\sim 10^6$

5. Compare the speeds of the attribute-guided searches to a baseline
   a. Baseline strategy is considering the probability of each function appearing to be the overall probability of that function appearing in the dataset
### Experiment #1.1

<table>
<thead>
<tr>
<th>Timeout needed to solve</th>
<th>DFS</th>
<th>Enumeration</th>
<th>$\chi^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20%</td>
<td>40%</td>
<td>60%</td>
</tr>
<tr>
<td>Baseline</td>
<td>163s</td>
<td>2887s</td>
<td>6832s</td>
</tr>
<tr>
<td>DeepCoder</td>
<td>24s</td>
<td>514s</td>
<td>2654s</td>
</tr>
<tr>
<td>Speedup</td>
<td>$6.8\times$</td>
<td>$5.6\times$</td>
<td>$2.6\times$</td>
</tr>
</tbody>
</table>
Figure 5: Number of test problems solved versus computation time.
Experiment #1 Takeaways

- Enumeration enjoys a better speedup than DFS
  - Hypothesized that the attribute vectors are more useful for a sort-and-add strategy than DFS anyway
  - The authors cite other research to suggest that sort-and-add does not lose much or any power from ignoring correlations/positional dependence
  - DFS is penalized quite heavily by making the wrong choice at the beginning, so it matters a lot more which function actually comes first
- The choice of a simple decoder (learned matrix) provides better results than an RNN decoder
  - Authors acknowledge that their strategy for training the RNN was somewhat unsuccessful
“Intuitively, the i-th row of this matrix shows how the presence of attribute i confuses the network into incorrectly predicting each other attribute j.”
Experiment #2

1. Train the neural network on programs of lengths $T = 1 \ldots 4$.
2. Test the network on test sets where all programs are of length $1 \ldots 5$
   a. N.b. the above two steps lead to 20 train/test pairs
3. Run the sort-and-add enumerative search with all combinations of training and testing data until 20% of programs are solved
   a. Compare the search with the learned attributes against the baseline priors as before
Experiment #2

![Graph showing speedup vs length of test programs for different training times $T_{train}$.

- The $y$-axis represents speedup on a logarithmic scale, ranging from $10^0$ to $10^3$.
- The $x$-axis represents the length of test programs $T_{test}$, ranging from 1 to 5.
- Different colors and markers represent different values of $T_{train}$.

For each length of test programs, speedup increases as $T_{train}$ increases.

- $T_{train} = 1$ (red markers, lowest speedup) shows minimal improvement.
- $T_{train} = 2$ (green markers) shows moderate improvement.
- $T_{train} = 3$ (orange markers) shows significant improvement.
- $T_{train} = 4$ (purple markers) shows the highest improvement.

At $T_{train} = 4$, the speedup is maximized across all lengths of test programs.

The graph illustrates the relationship between training time and performance improvement in the context of test programs.
Experiment #2 Takeaway

- Neural networks are able to generalize beyond programs of the same length that they were trained on
  - This is a result of having a search on top of attribute learning: the search can correct for the bad assumptions of the network.
Further Research
More explicit guiding for LIPS

- Menon et al. (2013) takes a similar approach to this paper, but incorporates explicitly defined “clues” that can be gleaned from input-output examples
  - E.g. input is a permutation of the output
  - Clues cause a reweighting of probabilities
- Domains are slightly off:
  - Smaller training/testing corpus
  - DSL that they chose lends itself towards these clues much more directly

Improving the data structures in the DSL

- Li et al. (2016) attempt to predict logical features of the program
  - Instead of presence/absence of certain functions alone
  - Employs a GNN instead of the simpler feed-forward architecture
- Allows the learning to focus on data structure shape and growth instead of simply tracking data changes.