# A Appendix

### Specifics of neural network architectures considered

The original DeepCoder network, summarized in detail in the supplemental material of (Balog et al., 2017), takes as input 5 I/O pairs, padded to a fixed length. Each integer in the network inputs is sent through a trainable embedding layer, represented as 20-dimensional real-valued vectors; these embeddings are then concatenated together and passed through three fully-connected layers of size 256. This yields 5 representations, one for each of the input/output pairs input into the network; these are then averaged, and passed as input into a final sigmoidal activation layer, which outputs the predicted probabilities of the 34 components appearing in the program. Note that the branching factor for a search tree (e.g. depth-first search) is larger than 34, since the lines which contain higher-order functions also require selection of one of the predicate functions; for lines which have two functions, the ranking order is determined by the smaller probability.

In the recurrent neural network, a long short-term memory (LSTM) network is added to produce a per-line heuristic. Most of the architecture is unchanged: as in the original DeepCoder model, the inputs and outputs are sent through an embedding layer, concatenated, passed through three fully-connected layers, and then averaged across the five examples. However, instead of predicting the probabilities of inclusion for each function directly, this representation is instead provided as an input into the LSTM, which outputs a new representation for each line of the program. As before, a final sigmoidal output layer emits probabilities for each of the 34 functions, but now it is applied across each line. The embedding layer in this network is 50-dimensional, and the both the fully connected layers and the LSTM have 200 hidden units. The result is a network which takes as arguments not just the input / output examples, but also a target "number of lines", and then returns estimates of probabilities that a function occurs on a per-line basis, rather than program-wide.

### Influence of the size of the training set

We investigated the effect of training on 90%, 20%, 10% and 1% of the possible programs. All the experiments here were done on programs of length $\ell = 3$, using non-uniform sampling to generate the training and test sets. Although we decrease the number of total unique programs in the training sets, the *total* number of examples remain fixed for each set at $n = 300000$.

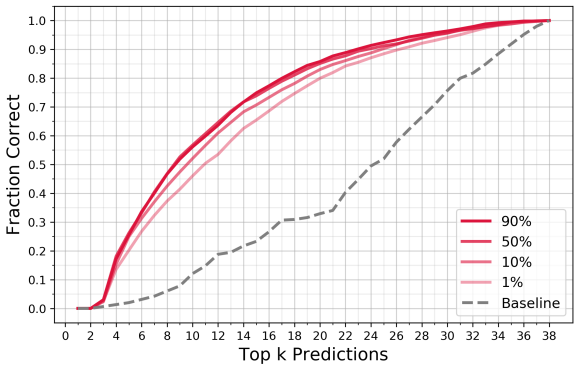


Figure 8: Here we vary the total number of unique programs in training sets and compare the resulting network performance on a fixed, semantically disjoint, test set. All sets were generated on length $\ell = 3$ programs with the non-uniform sampling method.

Although the later sets see less programs, they contain more examples per program.

We observe in Figure 8 that the performance on top-k prediction of the test set programs is not very sensitive to the amount of programs in the test set. In light of this fact, we choose to train with sets containing 10% of the possible programs. This is good news, since in most settings, valid input-output pairs would be cheaper to generate than valid programs.

### Details of Restricted Domain data generation method

A value range for acceptable outputs is specified (initially $[-255, 256]$), as well as the maximum length of the output if it is a list (the length is chosen uniformly at random from one to ten). The program is evaluated backwards, computing for each intermediate variable a 'safe' range for its value that guarantees to have outputs in the target range.

By applying this backward propagation of bounds to the whole program we find a suitable input range. Values are then sampled uniformly from this range to create the input(s), and the output is calculated by evaluating the program. If the resulting valid range for some input is empty or a singleton, then the program is discarded.

The short program described in Figure 9 illustrates the approach and a key limitation of it. The function SCANL1`(*)` on list $A$ outputs a list whose value at position $k$ is the product $\Pi_{i=0}^{k} A[i]$. For lists of length 5 the input range for SCANL1`(*)` that guarantees outputs between $-256$ and 256 is $[-3, 3]$. Pushing this range back through MAP`(+1)` gives a range for `b` of $[-4, 2]$, which is unchanged by FILTER `(%2==1)`. In

| **Example program:** | **Example input, and incremental output:** |
|---|---|
| a ← [int] | a = [-200, 144, 25, 66, -7, 38, -1, 14, 80, 81, 155] |
| b ← FILTER (%2==1) a | b = [25, -7, -1, 81, 155] |
| c ← MAP (+1) b | c = [26, -6, 0, 82, 156] |
| d ← SCANL1 (*) c | d = [26, -156, 0, 0, 0] |

Figure 9: A program with complex restrictions on inputs that will remain in the target range

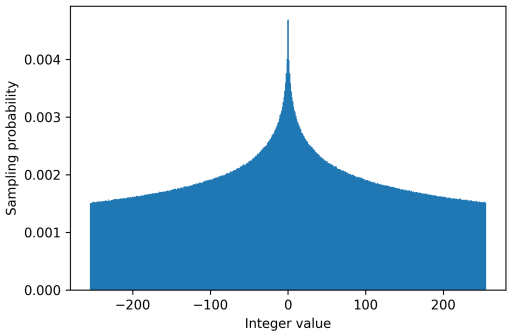


Figure 10: Marginal distribution over proposed integer values in the non-uniform sampling.

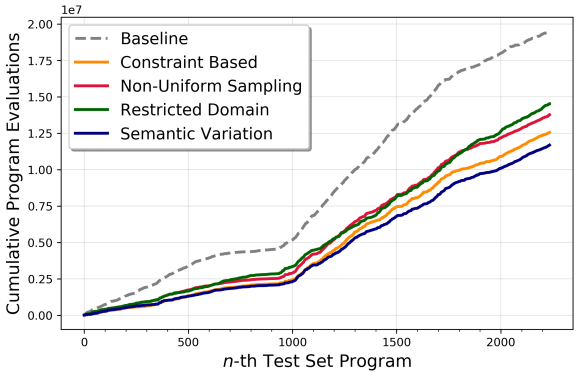


Figure 11: Relative improvement to a search procedure assisted by neural network predictions. The network was trained on the Semantic Variation data, we see how the loss in predictive performance is reflected in increased search effort

fact any even number could be accepted as part of the input, and if the input contains $-1$ at some point then any value appearing subsequently in list can be large.

### Non-uniform Sampling

The marginal distribution over sampled integer values is shown in Figure 10.

**A note about empty outputs** Some programs in the DSL output either null or the empty list on a large number of outputs. While these are informative to some extent, they dominated the examples generated for certain programs. The process of propagating bounds through the program is focussed on the value ranges and does not naturally exclude empty outputs. For example, it is not possible to specify a *range* (other than one containing only a single element) for inputs to FILTER(%2==0) that guarantees a non-empty output. To ensure that empty outputs could not dominate we included post processing for both sampling methods which rejects empty outputs until 90% of the permitted attempts have been made.

### Details of Simple Constraint Based Data Generation

Our training data was made up of 25 sets of 5 examples for each program. Due to the random choice of minimum input length there are 6 versions of the initial SMT problem. Subsequent SMT problems depend on the previous examples produced but because subsequent calls to the solver vary only slightly the examples generated can be very similar. We experimented with randomly adding weak constraints such as setting the first element of an input to be odd or even or positive or negative in order to modify the problem slightly for each call to the solver, however for the experiments reported here the data was produced without any such random constraints and a few examples are indeed repeated across different sets.

### The Effect of Neural Network Performance on Search

Since the intention of training the neural network is to use it to aid a search procedure, we ran a depth first search based on the predictions made by each network. The cumulative time taken to complete the search by each network is shown in Figure 11, showing the effect of reduced prediction accuracy on the time taken to find suitable programs. The network trained on similar data saved around one quarter of the total search time across the test set over the worst performing network. This shows how the benefits of machine learning to programming by example may be overstated when only evaluated on "friendly" artificial data.