UNSUPERVISED PROGRAM INDUCTION WITH HIERARCHICAL GENERATIVE CONVOLUTIONAL NEURAL NETWORKS

Qucheng Gong and Yuandong Tian and C. Lawrence Zitnick
Facebook AI Research
{qucheng, yuandong, zitnick}@fb.com

ABSTRACT

Automatically inferring a computer program that maps a set of inputs to outputs remains an open and extremely challenging problem. The program induction task is difficult due to the vast search space over possible programs, and the need to handle high-order logic, such as for-loops or recursion. In this paper, we automatically generate programs from input/output pairs using Hierarchical Generative Convolutional Neural Networks (HGCNN). The HGCNN predicts candidate lines of code in a hierarchical manner from which programs can be constructed using standard search techniques. Notably, the model is trained using only randomly generated programs, and can be viewed as an unsupervised approach. We show that the proposed method can generate programs ranging from simple procedures like `Swap` to more complex ones with loops and branches (e.g., finding the max element of an array). We also show promising results for nested-loop programs like Bubble Sort. Compared to baselines such as LSTMs, our approach achieves significantly better prediction accuracy.

1 INTRODUCTION

Computer programming is a difficult and time-consuming activity for humans. Automating this process through automatic code generation or program induction is a long standing topic in Artificial Intelligence [Solomonoff (1964)]. While heuristic search-based techniques have been proposed to address this problem [Koza (1992); Bar-David & Taubenfeld (2003); Mitchell (1982)] program induction remains extremely difficult due to its vast search space and need to reason about high-order logic (e.g., for-loops, recursion).

When programming, humans do not search through the combinatorial space of codes to find the one that addresses their needs. Instead, they typically devise a global plan and progressively refine it. This process relies heavily on past patterns they have learned from coding themselves, or from code written by others. By hierarchically writing code, humans are able to effectively prune away options that are unlikely to be effective, and focus on those likely to solve their task. Inspired by this ability, we propose an hierarchical approach to program induction inspired by recent advances in image generation [Dosovitskiy et al. (2015); Radford et al. (2015)].

In this paper, we use a Hierarchical Generative Convolutional Neural Network (HGCNN) to automatically generate programs from example input and output pairs. Our hierarchical approach takes a set of features extracted from the input and output pairs, and uses hierarchical upsampling to generate a 2D grid representing the instructions and arguments in the program. Within the 2D grid, each row represents a line of code (instruction plus arguments), and each column represents either the instruction or registers used as arguments. The use of hierarchical upsampling allows the algorithm to generate code that is consistent both locally and globally. Programs are generated by sampling the instructions and arguments using simple breath-first search. The inputs and outputs take the form of integer arrays. Since the code required to generate a single input/output pair is ambiguous, multiple examples are provided. Our generated code is written in a customized low-level language similar to assembly language containing 8 types of instructions.

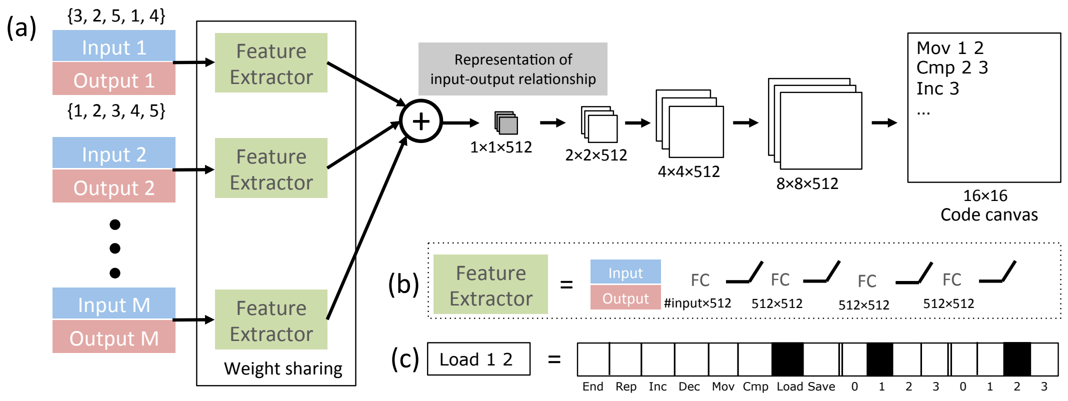

Figure 1: Overview of HGCNN . **(a)** The model takes M pairs of input/output memory maps, extracts features of each pair, and add them together to yield a joint representation of input-output relationship (the grey vertical bar). Then this representation is used to generate code on a 2D canvas using a deep convolutional neural network. In each stage, the spatial dimension of feature maps gets bigger. **(b)** Feature extractor composed of interleaving fully-connected (FC) layer and ReLU nonlinearity. **(c)** Each instruction is encoded as a concatenation of three one-hot representations, one for the operation and two for the operands (register indices).

Similar to humans, our approach learns from example programs. A set of training programs is randomly generated. For each random program, we randomly generate an input array and execute it to get the corresponding outputs that are used for training. The only help we provide during the learning process is we greedily remove redundant instructions, i.e. instructions that do not change the output. Since no code written by humans is used as training data, we may view the approach as being learned in an unsupervised manner. Our training examples vary from 5 to 10 lines of code.

We demonstrate that our system is capable of generating a wide variety of programs. These programs range from simple procedures such as adding constants to an array, to more complex programs that require loops and branching instructions. For instance, the system is able to generate code for doing vector addition and finding maximal elements. More complex procedures such as sorting, which contain nested for-loops, are still difficult for our approach to generate. However, the system can still predict its instructions with decent accuracy. Compared to common baselines such as LSTM [Hochreiter & Schmidhuber (1997)], our system achieves significantly higher prediction accuracy.

2 Related Work

Traditional methods approach program induction as a combinatorial optimization problem and rely on combinatorial solvers, e.g., Brute-force Search [Bar-David & Taubenfeld (2003)], Genetic Programming [Koza (1992)], Version Space [Mitchell (1982)], etc. Probabilistic graphical models treat program induction as an inference problem in which programs (or the unknown parts of them) are hidden variables to infer from input/output pairs [Jojic et al. (2006)]. Recently `TerpreT` [Gaunt et al. (2016)] proposed a probabilistic programming language that can encode many execution models (e.g., Turing pmachine, assembly, etc) as a probabilistic graphical model and provides four back-end solvers for inference. Probabilistic models have also been applied to inferring LOGO-like programs to draw shapes on a canvas [Ellis et al. (2015)]. Other approaches [Gulwani (2010)] use logic reasoning that turn program induction into a series of constraints that can be solved with off-the-shelf methods, e.g, Satisfiability Modulo Theories (SMT) [De Moura & Bjørner (2008)].

Recently, neural networks have been used to learn a differential mapping between input/ouput pairs. The mappings, which fulfill desired tasks such as sorting, are trained using gradient descent. These methods use a differentiable counterpart to basic discrete operations of Turing machines and external memory [Sukhbaatar et al. (2015); Reed & de Freitas (2015); Graves et al. (2016); Joulin & Mikolov (2015); Graves et al. (2014)]. One limitation is that the learnt program is implicitly encoded in the network parameters and is not easily interpretable by means other than checking its performed operations. Similar ideas have also been applied to code optimization, i.e., representing existing code in a differentiable manner and optimizing for more efficiency [Bunel et al. (2016); Riedel et al. (2016)]. However, as noted by [Gaunt et al. (2016)], to perform inference on the graphical model built for program induction, gradient descent is much slower than traditional combinatorial approaches (e.g, integer linear programming) and heavily relies on weight initialization. In contrast, our approach uses gradient descent only in the training stage. During the inference stage when input/output pairs are given, a simple forward pass performed in milliseconds is sufficient to get candidate instructions.

External annotations can also help program induction. [Solar-Lezama (2008); Gaunt et al. (2016); Riedel et al. (2016)] use human-provided code sketches and infer missing lines in the code. A large codebase with annotations, e.g., a dataset from an online coding competition that contains coding questions and various answers, has also been used to train an RNN model for program generation [Mou et al. (2015)]. In comparison, our model aims to generate programs with only input/output pairs, and is trained with no human annotations.

Our neural network architecture is inspired by [Dosovitskiy et al. (2015)] which generates synthetic images from an abstract concept (e.g., chair), and DCGAN [Radford et al. (2015)] that learns to generate images from hidden representations with adversarial training. In all cases, the networks progressively generate larger "canvases" from an intermediate representation with interleaving convolution and upsampling. Such approaches, including ours, can be regarded as learning the complicated inference step in the graphical model directly [Lin et al. (2015); Zheng et al. (2015)]. To our knowledge, we are the first to apply such architectures and to use randomly generated codebases to train a feed-forward inference model for program induction. We emphasize that for complicated problems such as program induction, hierarchical generative structures are essential for progressing from a global vision to the details of each line of code.

3 Approach

Our approach uses a Hierarchical Generative Convolutional Neural Network (HGCNN) to generate a program from a set of input/output pairs. In this section, we describe the language for which code is generated, the HGCNN, and how the training dataset is constructed.

3.1 Language Specification

Our programs are written in an assembly-like language. Table 1 shows the basic instructions of the language. It has three types of instructions, **(1)** operations on registers, **(2)** communication between registers and memory, and **(3)** control flow operations for conditionals and loops. Although simple, the language is capable of significant functionality.

The interpreter has a linear memory equal to the size of the input and output arrays. The memory is initialized with the values from the input. The goal is to find a program that updates the memory to be equal to that of the output array. This is accomplished using $K = 4$ registers, r_0, r_1, r_2 and r_3 that may be manipulated using 6 instructions. Before code execution, r_0 contains the length of the array and all other registers are zero. The array is located in the memory MEM, starting from address 0. Values may be read and saved from MEM to the registers using the "Load" and "Save" instructions. Note that MEM is not directly manipulated, and that all manipulations are performed using the registers. This simplifies the language while maintaining its functionality. The total number of possible instruction and register combinations per line of code is $4K^2 - K + 1$. With $K = 4$ there are 61 possibilities, which result in nearly one billion possible programs with just five lines of code.

A pre-compilation step is performed before code execution to match control flow instructions, e.g., match `Rep` and `Cmp` with their corresponding `End` instructions. The matching is performed in a first-in-last-out (FILO) fashion: when scanning the code sequence, the first `End` instruction will match the most recent control flow instruction. This naturally covers the common control structure found in most programs. After one-pass, the control instruction at line i knows which `End` instruction it jumps to, denoted as $\mathrm{CTRL}[i]$. For a program, if a control instruction cannot find a matching `End` instruction, it will jump to the end of the program. Similarly, unmatched `End` instructions are omitted.

Instruction	Description	Action	#Instructions
`Mov a b`	Copy r_b to r_a	$r_a \leftarrow r_b$, $\text{PC} \leftarrow \text{PC} + 1$.	$K(K-1)$
`Inc a`	Increase r_a by 1	$r_a \leftarrow r_a + 1$,$\text{PC} \leftarrow \text{PC} + 1$.	K
`Dec a`	Decrease r_a by 1	$r_a \leftarrow r_a - 1$,$\text{PC} \leftarrow \text{PC} + 1$.	K
`Save a b`	Save r_a to memory at r_b.	$\text{MEM}[r_b] = r_a$, $\text{PC} \leftarrow \text{PC} + 1$.	$K(K-1)$
`Load a b`	Load the memory at r_b to r_a	$r_a = \text{MEM}[r_b]$, $\text{PC} \leftarrow \text{PC} + 1$.	$K(K-1)$
`Cmp a b`	If $r_a \geq r_b$, go to `End`	If $r_a \geq r_b$ then $\text{PC} \leftarrow \text{CTRL}[\text{PC}]$ else $\text{PC} \leftarrow \text{PC} + 1$.	$K(K-1)$
`Rep a`	If $r_a \leq 0$, go to `End` Otherwise decrease r_a by 1	If $r_a \geq 0$ then $r_a \leftarrow r_a - 1$, $\text{PC} \leftarrow \text{PC} + 1$ else $\text{PC} \leftarrow \text{CTRL}[\text{PC}]$.	K
`End`	Used as jump label.		1

Table 1: Language specification. Note that PC is the program counter, MEM represents the memory and r_a ($a = 0, 1, 2, 3$) are the registers. For line i, $\text{CTRL}[i]$ is its pairing `End` instruction precomputed before program execution.

3.2 HGCNN Specification

HGCNN takes as input a set of M input/output pairs and produces a distribution over instructions and arguments for programs up to a certain length. Each line of code is represented using a one-hot representation, as shown in Fig. 1(c). Eight one-hot values represent the instructions. The arguments for the instructions are represented using $2K$ one-hot values, since some instructions use two registers as input. If $K = 4$, each line of code is represented using 16 values. Assuming the programs have at most 16 lines of code, the final output of the HGCNN has 16×16 dimensions.

HGCNN takes as input a set of input/output pairs. A 512 dimensional feature is extracted from each pair and averaged together. The features are computed using a four-layer neural network with ReLU non-linearities, Fig. 1(b). Each layer has 512 hidden units. The input to the feature extraction network is a 20×10 one-hot array representing the 20 entries of the input and output arrays and the 10 possible values for each entry which are assumed to range between 0 and 9.

Given the 512 dimensional extracted features, HGCNN progressively upsamples feature maps using a CNN until a feature map of size 16×16 is obtained, Fig. 1(a). Each upsampling increases the dimensions by $2\times$, and is performed using nearest neighbor. The convolutions of the CNN may be performed in either one or two dimensions. For 1D convolutions, the filters have size $3 \times n$ where n is the width of the current feature maps, i.e., the convolutions capture the regularities between lines of code. We also compare against using standard 2D convolutions [Dosovitskiy et al. (2015); Radford et al. (2015)] using filters with size 3×3. Zero-padding is used for filters that extend beyond the current boundaries. Zero-padding also has the benefit of helping the network determine whether it is near the beginning or end of the program.

Note that while we fix the size of the input and output arrays to 10 for training, but runtime input arrays of arbitrary size may be used. This is unlike neural programming models that only work for sequences up to a limited length [Reed & de Freitas (2015)]

3.3 Random Code Generation

Given the language specification, it is possible to randomly generate arbitrarily long code sequences. Due to its assembly-like nature, any code sequence can be executed to provide a deterministic result. For training, programs are randomly generated on the fly.

When generating random training programs it is possible the code sequence contains redundant instructions that will not alter the output memory. To remove redundant lines of code we use a simple greedy pruning strategy. We iteratively remove lines until no lines can be removed without changing the output result. Note that this does not remove pairs of lines (e.g., `Inc 1` and `Dec 1`) that collectively will not change the functionality. If fewer than 5 lines of code are left, the program is discarded. Programs that form infinite loops, that do not alter the memory, or that do not compile are also discarded. In practice 3% of randomly generated programs remain after pruning and around 2 million training samples are used to train the model.

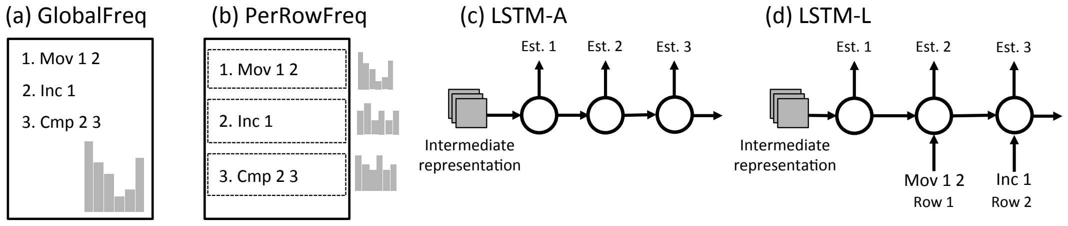

Figure 2: Baseline methods. **(a)** Global frequency instruction predictor. **(b)** Per-row frequency instruction predictor. **(c)** LSTM predictor that outputs each instruction sequentially. **(d)** LSTM predictor that predicts i-th instruction given $1, \ldots, i-1$-th oracle instructions.

4 Experiments

We test our system using 10 hand-coded programs that perform common tasks, and 1,000 randomly generated programs as listed in Tbl. 2. Programs 5-9 contain a single for loop and program 10 contains nested for-loops, which are very challenging. Given 10 input and output pairs, the system outputs a probability for each line of code. Using these probabilities, we can measure the ranking of each correct line of code by its average rank `AverRank` and its recall at the top-k in the candidate list (`Recall@k`). A perfect prediction gives `AverRank` = 1 and `Recall@k` = 100% and no search would be needed to generate the program. Our goal is to predict each line of code as accurately as possible to reduce the burden of search. One limitation of these metrics is that there could be multiple programs with the same functionality; however, these measures could be regarded as a lower bound on the actual performance of the code generator.

Baselines. For comparison, we use several baselines (Fig. 2). **GF.** An instruction predictor from overall instruction frequency. **PRF.** A per-row instruction predictor from per-row instruction frequency. **LSTM-A.** A recurrent neural network with Long Short-term Memory (LSTM [Hochreiter & Schmidhuber (1997)]) with hidden nodes that take intermediate representations as input, and generate programs one line at a time. As a fair comparison, we do not input the ground truth instruction into the LSTM after each instruction is predicted. **LSTM-L.** We also compare against an oracle LSTM that uses the ground truth instructions of rows $1, \ldots, i-1$ to predict the instruction of row i (e.g., language model [Mikolov et al. (2010)]).

Prediction Accuracy. Tbl. 3 shows the performance of different approaches. Each number is averaged over 20 experiments, each with 10 input/output pairs. For an instruction, HGCNN rank is computed according to the average confidence of its three components on the canvas. Predictions from LSTMs perform poorly even with representations of increased dimensions. In comparison, the prediction from HGCNN with either 1D or 2D convolutions has lower average rank, and has higher recall compared to all baselines. Using either 1D or 2D convolutions for HGCNN produces similar results. Note even for complicated programs like bubble sort, the overall prediction accuracy is surprisingly high.

In addition to the 10 test programs, we also prepare a test set with 1000 random generated programs. The length of the programs varies from 3 to 8 lines. Tbl. 4 shows the performances of different approaches. On Average, HGCNN predicts significantly better than all baselines. Note that LSTM performs poorly on these tests.

Fig. 3 shows instruction ranks and neuron heatmaps of 4 selected programs based on HGCNN . HGCNN can accurately predict the program `Array Plus One` and `Find Max`. Relatively, it did poorly on certain instructions in `Bubble Sort` and `Load Element`. This may be due to their lower frequency in the training set.

Nearest Neighbor. With a small test set, it is important to makes sure that each test example does not appear in the randomly generated codebase for training. We thus compare the edit distance in terms of instructions between each test program with the training set. The last column in Tbl. 2 shows the results. While simple programs like zeroing an array have an exact match in the codebase, the edit distance is high for more complicated programs. Since the number of programs used for training is on the order of millions, and the generator could generate billions of programs, it is unlikely the network exactly memorized these programs.

No.	Name	Description	Oracle Implementation	ED
1	**Copy Element**	Copy an element	`Dec 0; Mov 1 0; Dec 1; Load 2 1; Save 2 0`	1
2	**Save Element**	Save a designated element to memory	`Dec 0; Load 1 0; Dec 0; Load 2 0; Save 2 1`	1
3	**Load Element**	Load a designated element to memory	`Dec 0; Mov 1 0; Dec 1; Load 2 1; Load 3 2; Save 3 0`	2
4	**Swap**	Swap two elements	`Load 2 0; Load 3 1; Save 2 1; Save 3 0`	2
5	**Zero Array**	Zero the array	`Rep 0; Save 1 0; End`	0
6	**Plus One**	Add all elements by 1	`Rep 0; Load 1 0; Inc 1; Save 1 0; End`	1
7	**Plus One Even**	Add elements at even indexes by 1	`Rep 0; Dec 0; Load 1 0; Inc 1; Save 1 0; End`	1
8	**Find Max**	Find array's maximum	`Rep 0; Load 1 0; Cmp 2 1; Save 2 1; End; End`	2
9	**Reverse**	Reverse the array	`Rep 0; Cmp 1 0; Load 2 0; Load 3 1; Save 2 1; Save 3 0;Inc 1; End; End`	3
10	**Bubble Sort**	Sort with bubble sort	`Rep 0; Mov 1 0; Rep 1; Load 2 0; Load 3 1; Cmp 2 3;Save 2 1; Save 3 0; End; End; End`	5

Table 2: Test programs and their descriptions. ED means Edit Distance to codebase. (Due to precompliation rules, the final `Ends` are optional.)

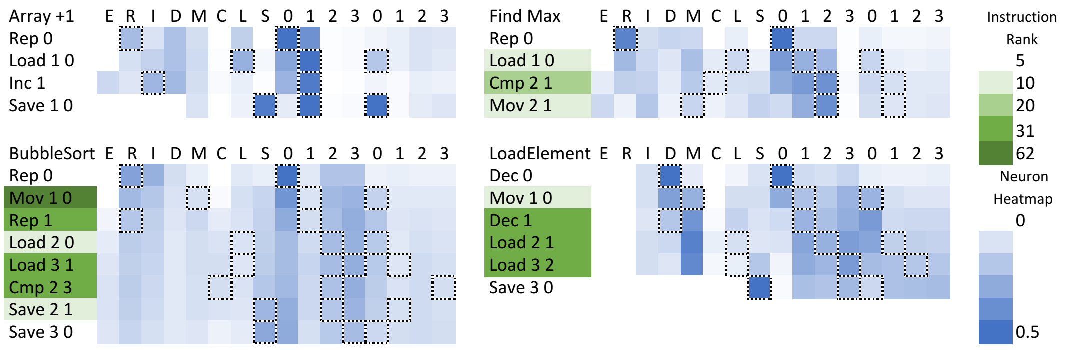

Figure 3: Ranking (green) and probabilities per instruction and argument (blue) for 4 selected programs based on HGCNN . Each instruction is denoted by its first letter in the same order as Fig. 1(c). E=`End`, R=`Rep`, I=`Inc`, D=`Dec`, M=`Mov`, C=`Cmp`, L=`Load`, S=`Save`. The two sets of "0,1,2,3" are its two potential arguments.

Even if the correct program exists in the training set, one cannot efficiently perform a search to retrieve such programs. This is because each training program would have to run on the input/output pairs to determine whether the correct outputs were achieved. Caching the input/output pairs of training programs would also be infeasible since their exists an exponential number of inputs. Therefore, nearest neighbor is a good measure of the similarity between the training and testing set, but not a practical baseline.

Generating the Program. Based on the ranking result provided, we perform a breath first search (BFS) to extract the program that can map the input to the output. Tbl. 5 shows the time spent to rediscover each test program. All experiments use a single thread on one Intel Xeon E5-2680 v2 at 2.80GHz. We see the time spent using the candidates suggested from HGCNN is much shorter than LSTM-A. Note that HGCNN outputs candidate instructions with one pass, while LSTM-L has to be evaluated each time a search node is expanded. As a result, even if LSTM-L gives the same average rank, it is substantially slower in terms of search. TerpreT [Gaunt et al. (2016)] also

Metric	No.	GF	PRF	LSTM-A			LSTM-L			HGCNN			
				128	256	512	128	256	512	128 2D	256 2D	512 2D	512 1D
Average Rank	1	21.2	16.6	28.2	22.2	25.2	29.6	29.2	32.3	8.1	7.9	7.2	**6.6**
	2	26.8	24.0	29.8	29.2	32.6	38.8	30.3	26.0	12.1	10.6	10.5	**6.1**
	3	25.0	23.7	23.3	29.7	33.2	30.0	25.4	38.8	17.0	15.5	18.2	**14.5**
	4	36.3	27.5	29.6	35.5	20.7	22.8	25.1	21.4	**15.4**	19.4	18.4	19.7
	5	26.5	18.0	26.4	24.2	24.0	34.0	26.5	30.6	4.1	**1.5**	4.5	5.2
	6	25.3	25.8	31.5	27.8	25.6	36.0	40.9	37.1	3.6	**2.1**	2.4	3.1
	7	16.2	12.8	28.1	29.1	30.6	39.2	41.1	34.9	9.8	7.4	7.6	**7.1**
	8	32.5	34.0	45.9	34.9	41.9	28.0	30.5	38.0	14.4	12.9	**10.5**	12.3
	9	26.3	26.3	35.1	27.8	29.5	37.0	34.5	29.4	22.6	22.3	**15.6**	18.6
	10	33.0	27.6	38.4	34.0	26.7	35.2	30.0	32.2	25.2	**18.9**	20.5	26.2
Recall @5	1	20%	40%	20%	20%	15%	0%	0%	0%	62%	63%	64%	**80%**
	2	40%	40%	0%	0%	8%	0%	0%	20%	44%	45%	42%	**64%**
	3	17%	33%	16%	17%	7%	0%	0%	1%	33%	28%	40%	**45%**
	4	0%	0%	0%	**25%**	0%	0%	0%	3%	0%	20%	**25%**	23%
	5	0%	0%	0%	10%	0%	0%	0%	0%	55%	**100%**	50%	50%
	6	0%	0%	0%	12%	13%	0%	0%	0%	60%	64%	73%	**84%**
	7	60%	60%	0%	0%	0%	0%	0%	5%	55%	**73%**	66%	59%
	8	0%	0%	0%	10%	5%	0%	0%	0%	28%	30%	**48%**	39%
	9	29%	29%	0%	14%	8%	2%	2%	13%	22%	23%	**40%**	23%
	10	0%	0%	13%	7%	13%	3%	0%	0%	**25%**	20%	**25%**	19%
Recall @10	1	20%	60%	23%	22%	25%	9%	6%	15%	74%	68%	76%	**80%**
	2	40%	40%	0%	20%	17%	0%	6%	20%	52%	57%	62%	**79%**
	3	17%	50%	24%	17%	17%	2%	6%	16%	42%	**52%**	40%	**52%**
	4	0%	25%	25%	25%	**50%**	42%	28%	47%	15%	**50%**	25%	45%
	5	0%	50%	33%	33%	33%	0%	10%	10%	**100%**	**100%**	**100%**	75%
	6	25%	25%	0%	20%	50%	0%	0%	0%	90%	**100%**	**100%**	95%
	7	60%	60%	3%	20%	22%	0%	0%	17%	68%	77%	**78%**	67%
	8	0%	25%	0%	15%	10%	0%	0%	0%	48%	42%	**73%**	50%
	9	29%	29%	0%	36%	16%	14%	12%	17%	29%	27%	**47%**	39%
	10	12%	25%	13%	17%	35%	13%	11%	10%	37%	27%	**38%**	35%

Table 3: Average rank and recall at top 5/10 candidates provided by different methods, with different dimensions. No. refers to the program number from Tbl. 2. 1D and 2D refer to the type of convolution used by HGCNN . Note that `End` at the rear of the program is optional and is not included in the average rank computation.

Metric	GF	PRF	LSTM-A			LSTM-L			HGCNN			
			128	256	512	128	256	512	128 2D	256 2D	512 2D	512 1D
Average Rank	15.5	13.3	32.1	33.4	31.8	33.5	35.0	29.1	12.6	12.5	11.8	**11.7**
Recall@5	25%	36%	5%	6%	6%	5%	5%	10%	47%	48%	**51%**	50%
Recall@10	45%	52%	10%	12%	13%	9%	11%	19%	63%	63%	**65%**	**65%**

Table 4: Averaged rank and recall at top 5/10 for 1000 randomly generated programs with 3 to 8 lines of code.

provides execution times for some programs, but their experimental setup is different. While their timing results may not be directly comparable, their approach tends to be slower perhaps despite their complicated inference techniques.

Other Search Techniques. We also tried other more advanced search techniques, e.g., Monte Carlo Tree Search (MCTS) [Browne et al. (2012)] that dynamically assigns likelihoods to each expanded search branch based on previous search results. However, it turns out that the value function of code generation is very different from games like Go [Silver et al. (2016); Tian & Zhu (2016)], in which past search results can inform the future. In code generation past experience is often misleading and might actually hurt the performance, if not used properly. For example, if the goal is to set all array elements to one, a tempting solution could be to set the first element to one, which may receive a partial reward. This prevents switching to a better solution that starts with a loop first.

	HGCNN	LSTM-A	TerpreT (Assembly)
Zero Array	3 seconds	3 minutes	N/A
Access	3 seconds	3 minutes	3.8 seconds
Swap	13 minutes	4 hours	N/A
Array Plus One	20 seconds	-	N/A
Array Minus One	30 seconds	-	69.4 seconds
Find Max	2 hours	-	N/A
Bubble Sort	-	-	N/A

Table 5: Time spent in search to generate the code. "-" means that the time spent is greater than 6 hours. "N/A" means that the result is not available. TerpreT is a probabilistic programming language from [Gaunt et al. (2016)]. Note that the experiment setup is very different and the time spent listed here is for reference.

5 Discussion

This paper serves as a first step towards utilizing hierarchical deep learning techniques for program induction. Future direction include:

Understanding the internal representations. The hierarchical generative process produces intermediate results in addition to the final instruction candidates. It is interesting to analyze these representations, e.g., with clustering, and check for the existence of any regularities. In particular, how the program encodes for-loop and branches. Humans commonly program in a hierarchically manner: a high-level plan is constructed and code is written and debugged. Knowledge is gained through the debugging process. Whether machines could mimic this progress is yet to be explored.

Dynamic knowledge building. In practice, humans never write complicated programs from scratch. Humans use basic libraries, can quickly pick up new ones, and can apply them to generate powerful programs. For example, learning to write bubble sort with an additional `Swap` function. This requires a general representation for arbitrary functions that can be easily learned and incorporated into the system.

Instruction-independent Representation. In this paper, we specify eight types of instructions and use them as the targets of the hierarchical generative model. However, a mapping should ideally only depend on the transformation of input/output pairs and is independent of the specific language being used. With the instruction-independent representation, one could pick up any language to write code without retraining the model from sketch.

6 Conclusion

In this paper, we propose a novel hierarchical generative framework for program induction. Given input and output pairs, the HGCNN automatically gives candidate instructions for each line of the program. The correct candidates generally have high probability, which narrows the choices when searching for an effective program. Our model is trained using randomly generated programs based on a custom designed and simplified assembly-like language. No human labels are used for training. The proposed system can write programs with simple for-loops (e.g., finding the maximum element of an array), and shows promising prediction accuracy for more complicated programs like Bubble Sort, and is significantly better than frequency and LSTM baselines.

References

Bar-David, Yoah and Taubenfeld, Gadi. Automatic discovery of mutual exclusion algorithms. In *International Symposium on Distributed Computing*, pp. 136–150. Springer, 2003.

Browne, Cameron B, Powley, Edward, Whitehouse, Daniel, Lucas, Simon M, Cowling, Peter I, Rohlfshagen, Philipp, Tavener, Stephen, Perez, Diego, Samothrakis, Spyridon, and Colton, Simon. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

Bunel, Rudy, Desmaison, Alban, Kohli, Pushmeet, Torr, Philip HS, and Kumar, M Pawan. Adaptive neural compilation. *arXiv preprint arXiv:1605.07969*, 2016.

De Moura, Leonardo and Bjørner, Nikolaj. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer, 2008.

Dosovitskiy, Alexey, Tobias Springenberg, Jost, and Brox, Thomas. Learning to generate chairs with convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1538–1546, 2015.

Ellis, Kevin, Solar-Lezama, Armando, and Tenenbaum, Josh. Unsupervised learning by program synthesis. In *Advances in Neural Information Processing Systems*, pp. 973–981, 2015.

Gaunt, Alexander L, Brockschmidt, Marc, Singh, Rishabh, Kushman, Nate, Kohli, Pushmeet, Taylor, Jonathan, and Tarlow, Daniel. Terpret: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428*, 2016.

Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Graves, Alex, Wayne, Greg, Reynolds, Malcolm, Harley, Tim, Danihelka, Ivo, Grabska-Barwińska, Agnieszka, Colmenarejo, Sergio Gómez, Grefenstette, Edward, Ramalho, Tiago, Agapiou, John, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 2016.

Gulwani, Sumit. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pp. 13–24. ACM, 2010.

Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Jojic, Vladimir, Gulwani, Sumit, and Jojic, Nebojsa. Probabilistic inference of programs from input/output examples. 2006.

Joulin, Armand and Mikolov, Tomas. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in Neural Information Processing Systems*, pp. 190–198, 2015.

Koza, John R. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

Lin, Guosheng, Shen, Chunhua, Reid, Ian, and van den Hengel, Anton. Deeply learning the messages in message passing inference. In *Advances in Neural Information Processing Systems*, pp. 361–369, 2015.

Mikolov, Tomas, Karafiát, Martin, Burget, Lukas, Cernockỳ, Jan, and Khudanpur, Sanjeev. Recurrent neural network based language model. In *Interspeech*, volume 2, pp. 3, 2010.

Mitchell, Tom M. Generalization as search. *Artificial intelligence*, 18(2):203–226, 1982.

Mou, Lili, Men, Rui, Li, Ge, Zhang, Lu, and Jin, Zhi. On end-to-end program generation from user intention by deep neural networks. *arXiv preprint arXiv:1510.07211*, 2015.

Radford, Alec, Metz, Luke, and Chintala, Soumith. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

Reed, Scott and de Freitas, Nando. Neural programmer-interpreters. *International Conference on Learning Representations (ICLR)*, 2015.

Riedel, Sebastian, Bošnjak, Matko, and Rocktäschel, Tim. Programming with a differentiable forth interpreter. *arXiv preprint arXiv:1605.06640*, 2016.

Silver, David, Huang, Aja, Maddison, Chris J, Guez, Arthur, Sifre, Laurent, Van Den Driessche, George, Schrittwieser, Julian, Antonoglou, Ioannis, Panneershelvam, Veda, Lanctot, Marc, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

Solar-Lezama, Armando. Program synthesis by sketching. *PhD thesis*, 2008.

Solomonoff, Ray J. A formal theory of inductive inference. part i. *Information and control*, 7(1): 1–22, 1964.

Sukhbaatar, Sainbayar, Weston, Jason, Fergus, Rob, et al. End-to-end memory networks. In *Advances in neural information processing systems*, pp. 2440–2448, 2015.

Tian, Yuandong and Zhu, Yan. Better computer go player with neural network and long-term prediction. *International Conference on Learning Representation*, 2016.

Zheng, Shuai, Jayasumana, Sadeep, Romera-Paredes, Bernardino, Vineet, Vibhav, Su, Zhizhong, Du, Dalong, Huang, Chang, and Torr, Philip HS. Conditional random fields as recurrent neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1529–1537, 2015.