Information and Software Technology 126 (2020) 106350

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Neural networks learn to detect and emulate sorting algorithms from images of their execution traces

Cătălin F. Perticas[a], Bipin Indurkhya[b,*]

[a] Babes-Bolyai University Cluj-Napoca, Romania
[b] Cognitive Science Department Jagiellonian University, Cracow, Poland

ARTICLE INFO

Keywords:
Computer program traces
Program behavior
Program induction
Algorithms
Neural networks
Program visualization
Data representation

ABSTRACT

Context: Recent advancements in the applicability of neural networks across a variety of fields, such as computer vision, natural language processing and others, have re-sparked an interest in program induction methods. (Kitzelman [1], Gulwani et al. [2] or Kant [3].)

Problem: When performing a program induction task, it is not feasible to search across all possible programs that map an input to an output because the number of possible combinations or sequences of instructions is too high: at least an exponential growth based on the generated program length. Moreover, there does not exist a general framework to formulate such program induction tasks and current computational limitations do not allow a very wide range of machine learning applications in the field of computer programs generation.

Objective: In this study, we analyze the effectiveness of execution traces as learning representations for neural network models in a program induction set-up. Our goal is to generate visualizations of program execution dynamics, specifically of sorting algorithms, and to apply machine learning techniques on them to capture their semantics and emulate their behavior using neural networks.

Method: We begin by classifying images of execution traces for algorithms working on a finite array of numbers, such as various sorting and data structures algorithms. Next we experiment with detecting sub-program patterns inside the trace sequence of a larger program. The last step is to predict future steps in the execution of various sorting algorithms. More specifically, we try to emulate their behavior by observing their execution traces. We also discuss generalizations to other classes of programs, such as 1-D cellular automata.

Results: Our experiments show that neural networks are capable of modeling the mechanisms underlying simple algorithms if enough execution traces are provided as data. We compare the performance of our program induction model with other similar experimental results from Graves et al. [4] and Vinyals et al. [5]. We were also able to demonstrate that sorting algorithms can be treated both as images displaying spatial patterns, as well as sequential instructions in a domain specific language, such as swapping two elements. We tested our approach on three types of increasingly harder tasks: detection, recognition and emulation.

Conclusions: We demonstrate that simple algorithms can be modelled using neural networks and provide a method for representing specific classes of programs as either images or sequences of instructions in a domain-specific language, such that a neural network can learn their behavior. We consider the complexity of various set-ups to arrive at some improvements based on the data representation type. The insights from our experiments can be applied for designing better models of program induction.

1. Introduction and motivation

In recent years, there is a renewed interest in modeling the process of software creation, both in terms of how we understand software and programs, as well as how to apply artificial intelligence to write programs (DeepCoder by Balog et al. [6], RobustFill by Devlin et al. [7], Neural Turing Machine by Graves et al. [4]). Using any means of artificial intelligence to learn programs or algorithms from incomplete specifications is commonly referred to as program induction. There are two main paradigms for program induction (Kant [3]): latent program induction and program synthesis.

[*] Corresponding author at: Instytut Filozofii UJ, Ul. Grodzka 52, 31-044 Krakow, Poland.
E-mail addresses: perticascatalin@gmail.com (C.F. Perticas), bipin.indurkhya@uj.edu.pl (B. Indurkhya).

https://doi.org/10.1016/j.infsof.2020.106350
Received 13 October 2019; Received in revised form 27 April 2020; Accepted 17 May 2020
Available online 24 May 2020

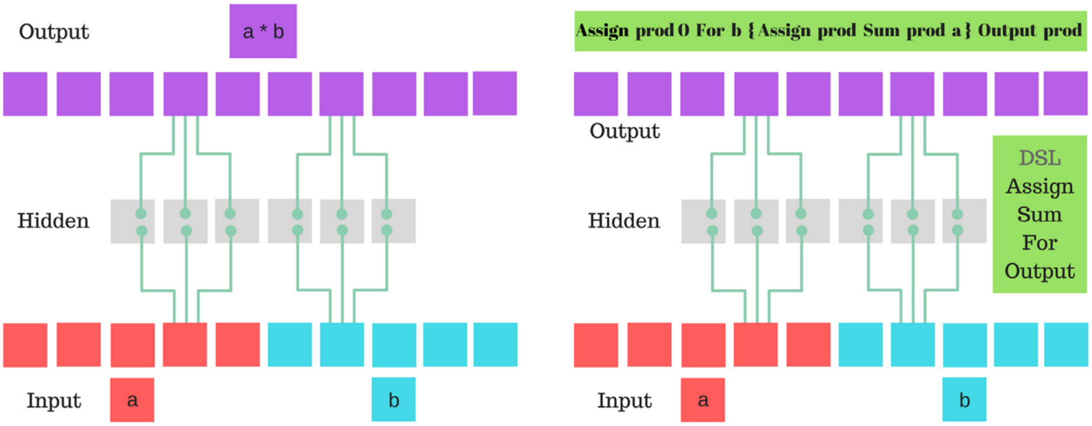


Fig. 1. Left: latent program induction. Right: program synthesis.

When a neural network learns to map input to output, thereby solving a programming task, the program is stored in a network, which is executed through neural activation patterns. This is called **latent program induction**, because the representation of the generated program is not in a human readable form, but resides in the weights and activation patterns of the neural network that emulates the program's input-output behavior.

The second paradigm is to formulate the problem such that the neural network outputs a program in a computer language, which is then executed to get the desired output. This is generally referred to as **program synthesis**. In this case, the network tries to find a program that follows certain constraints, and the generation process is similar to text generation.

Latent programs are written in the language of neural networks, whereas synthesized programs are written in a language of choice (domain specific language). A comparison of the two approaches applied on string transformation problems is presented by Devlin et al. [7]. Fig. 1 exemplifies the differences between the two methods on the simple task of multiplying two numbers. Both methods have pros and cons, but neither of these exploits potentially available knowledge about the intermediate steps of the program which maps the input to the output. The more knowledge a neural network has on how a program works, the less it needs to guess in order to discover the desired program.

In this paper, we propose a new framework for achieving simple programs emulation through basic neural network models. In particular, we develop a visual method for representing programs, which we use as training data for various neural network models (MLP, CNN, RNN - see Krizhevsky et al. [8] and Graves [9]) and tasks (sorting, reversing and manipulating binary search trees). The programs are written in Python and their operations are carried out through swaps. Each time a swap gets executed, another column with the new configuration of values in the array is attached to the image of the program trace. We call this representation Image of an Algorithm (IoA) because it consists of visualizing granular in-memory operations performed during the execution of a program. This visualization method reflects program dynamics.

To test the success of our approach we conducted two experiments to find out if execution traces can be recognized with a neural network. The results confirm this assumption as we are able to get very accurate results for a memory size of 20 and a pool of 20 different algorithms. The third experiment tests whether emulation of these algorithms is possible with neural networks. Based on these results, we discuss some limitations and further steps to follow in this direction for obtaining more interesting and widely applicable results.

Our approach is different from the others in the field of program induction in that we exploit repetitive visual patterns in the execution of simple programs. For instance, Balog et al. [6] use neural networks to directly generate source code for simple problems on lists (filter, map, count, sum) and Devlin et al. [7] use neural networks for inferring string transformations. Both approaches use input-output pairs as training data. However, it is not practically feasible for a neural network to be able to learn a very complex program simply from input-output pairs. To understand why, we can draw a parallel to human problem solving, where a lot of our knowledge comes from mappings learned through step-by-step guidance, rather than by inferring a result from input through many observations. This can be seen as a type of transfer learning. For this reason, we suggest a method for learning the semantics of a program by looking at and recognizing its intermediate steps, which can be inferred from execution traces.

Analysing program traces has been successful in the areas of identifying methods which contribute to an overall's program execution time (Shah & Guyer [10]), detecting software vulnerabilities (Huang et al. [11]), and automated malware classification (Pascanu et al. [12]). The manner in which a program is represented or visualized has an effect on its comprehensibility, which affects how efficiently someone can perform a task on it (Fittkau et al. [13]). One of the goals of our study is to show that program traces are a very natural way for neural networks to understand and emulate programs.

2. Background and related research

To the best of our knowledge, no research has been done particularly on automated classification or detection of computer programs based on execution traces. However, there are some related experiments which apply program induction to a set of tasks. There are two methods reported in the study conducted by Devlin et al. [14]. First, training a model with a large number of input-output pairs belonging to a family of tasks - a meta-learning approach. The second is a portfolio-based method, where the best pre-trained model on a set of related tasks is adapted towards solving a new task using transfer learning. Other experimental set-ups report results on specific tasks for: copying & sorting (Graves et al. [4]), graph traversal (Graves and Wayne [15]), path finding & convex hulls (Vinyals et al. [16]) and lists (Balog et al. [6]).

The experiments conducted in this study are performed both on specific classes of programs (*Program Classification*: twenty program classes, *Algorithm Prediction and Behavior Emulation*: five sorting algorithms), as well as on a family of programs (*Pattern Detection in Sequences*: sorting vs. non-sorting groups). In this sense, the ability to detect previously seen algorithmic patterns can be used as prior knowledge to learn about new programs.

For instance, by learning whether a program consists of a sorting or a non-sorting algorithm from a predefined set of programs, the type of a new previously unseen program can be predicted. We can train detectors on various such tasks: a different example would be whether a program uses lists or trees. This could prove relevant in deciding what type of

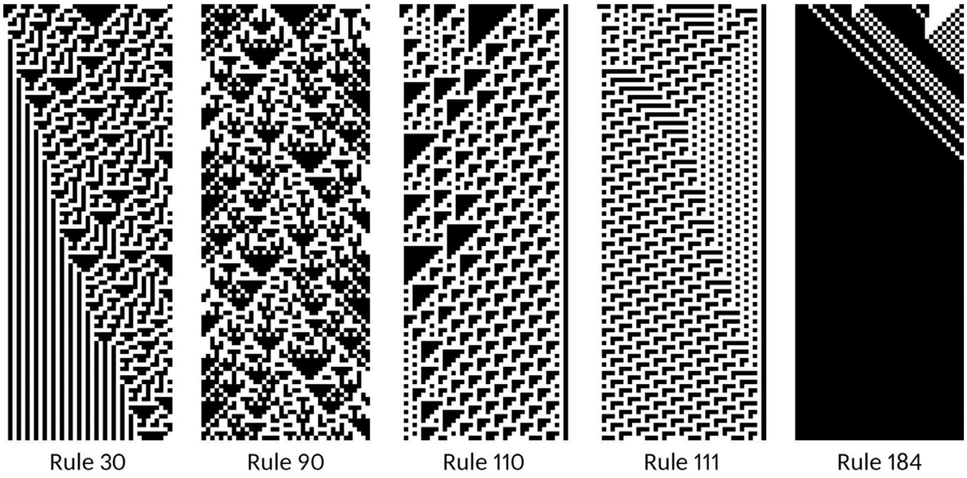


Fig. 2. Images of 1D cellular automata.

problem the program solves and which aspects of the input data have a higher influence on the output data of the program.

The concept of a family of programs arises naturally from how we define a set of programs. In the setup proposed here, the programs are created around the swap operation in an array of numbers. From this point of view, we can relate our approach to the topic of cellular automata.

Cellular Automata (CAs) are a discrete model for simple computations where an array of cells with states (typically *on* and *off*) is evolved over time according to a rule/transition table. Each cell is modified depending on its state and the states of its left and right neighbors. Fig. 2 shows a visual representation of several rules for 1D CAs.

CAs have been extensively studied so far. The main classification is based on whether their structure stabilizes over time and based on the types of patterns that they display. Wolfram [17] puts them into 4 categories:

- Class 1: Stabilizes into a homogeneous structure;
- Class 2: Patterns evolve into stable and oscillating structures;
- Class 3: Seemingly chaotic patterns;
- Class 4: Patterns are very complex and can last a long time with stable local structures.

The evolution of the patterns depends on the initial conditions (the initial state). For this reason the classification above makes the assumption that the computations are performed across several starting states, since there might be exceptional cases when a CA stabilizes given a special input, but in other more general cases it displays a complex behavior.

The concepts behind IoAs are related to those behind CAs - both involve simple programs, a pattern of states and a visual representation. However, one difference between CAs and IoAs is that CAs are computed using the neighborhood-based transition operation, while IoAs were studied in the context of the swap operation. Another difference is that IoAs have cells with continuous values, while CAs have discrete values (0 or 1).

Although CAs are very simple in their nature, they are able to simulate all kinds of behaviors, such as traffic flows (Rawat et al. [18], Guzman et al. [19]) and segmentation of images in regions (Diosan et al. [20]). Similarly, IoAs can be used to simulate programs working on arrays of objects.

3. Method

Like most processes following a set of rules, programs display visible patterns in their execution traces. These traces are helpful to programmers in many ways: when trying to find a bug; when trying to understand a module; or when trying to figure out the order in which certain steps are performed; and so on. We present an approach, called **Images of Algorithms (IoAs),** for visually representing program traces. For our initial study, we chose to focus on programs that work on fixed-length arrays. Without loss of generality, we normalize the array elements to real-valued numbers between 0.0 and 1.0. Then we represent each array as a column of pixels in an image. Brighter pixels represent numbers closer to 1.0, and darker pixels correspond to numbers closer to 0.0. The key idea is to have a visual representation of programs.

3.1. Dataset

All the programs we studied are built on the operation of swapping two elements in the array. A new trace is recorded whenever the swap operation occurs. For experiments, we used 20 classes of programs shown in Table 1: 10 sorting programs and 10 non-sorting programs. Their visual representations are shown in Fig. 3 and their descriptions can be found below.

Altogether, the programs incorporate a variety of concepts relevant in computer science like:

1. **Time-complexity** classes: (bubblesort $O(N^2)$, quicksort $O(NlogN)$, reversing an array $O(N)$). The relation between the IoA length and the time complexity is not in all cases directly proportional because the IoA represents the state of the program only at key places in its execution.
2. **Deterministic vs. non-deterministic** programs: bubblesort vs. randomsort. Some algorithms look more chaotic or unpredictable than others.
3. **Dependency on the input data:** the sorting strategy depends on the input order. For instance, bubblesort, insertionsort, quicksort and randomsort will not perform any swap at all if the array is already sorted. Similarly, heapsort will not perform the part for creating the heap structure if the array already has a heap structure. But will always perform the repetitive procedure of extracting the minimum in order to get the array sorted once the heap structure is established. However, the input data does not have an influence on the strategy of the algorithm while reversing an array or when performing interval swaps. This is because the array values are irrelevant for the decisions made in non-sorting algorithms.
4. **Data structures**: sequential (list - insertionsort and bubblesort, array - quicksort, reversing) and trees (binary search tree, min and max heap);
5. **Ordering principles:** Output is the same permutation of numbers for every input - sorting and binary search tree; output is a permutation that depends on the initial permutation - reversing a list and complementary permutation; output is a permutation that cannot

Table 1
Types of programs used for experiments. Classes 0–9: Sorting Programs. Classes 10–19: Non-sorting Programs.

Program Class & Name	Program Description
0. InsertionSort	Sorts the array elements in increasing order using the **insertion sort** method.
1. InsertionSort	Same method as above, decreasing order.
2. BubbleSort	Sorts the array elements in increasing order using the **bubble sort** method.
3. BubbleSort	Same method as above, decreasing order.
4. HeapSort	Sorts the array elements in increasing order using the **heap sort** method.
5. HeapSort	Same method as above, decreasing order.
6. QuickSort	Sorts the array elements in increasing order using the **quick sort** method.
7. QuickSort	Same method as above, decreasing order.
8. RandomSort	Sorts the array elements in increasing order using the **random sort** method (iteratively picks two random elements and swaps if not in correct order).
9. RandomSort	Same method as above, decreasing order.
10. Reverse	Reverses the entire array.
11. Reverse	Reverses first half, then second half of the array.
12. IntervalSwap	Swaps first half with second half of the array.
13. IntervalSwap	Swap first quarter with second and third quarter with forth of the array.
14. RandomSwap	Randomly swaps N pairs of elements in the array.
15. BinarySearchTree	Orders the elements in the array s. t. they represent a **binary search tree**.
16. MinHeap	Orders the elements in the array s.t. they represent a **min-heap**.
17. MaxHeap	Orders the elements in the array s.t. they represent a **max-heap**.
18. Scrambler	If 3 consecutive array elements are in increasing order, swaps first with second.
19. ComplPerm	Computes the permutation of the array, then orders elements s.t. their permutation is complementary to the initial permutation

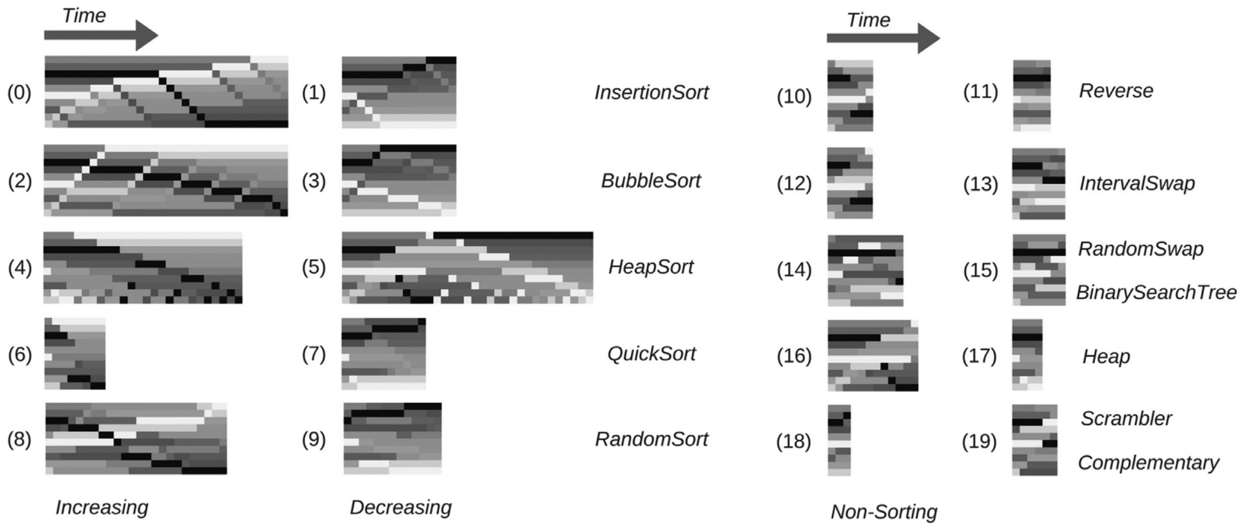


Fig. 3. Images of Algorithms for 20 different programs with a 10-element array input. *Left* (classes 0–9): **Sorting Programs**. *Right* (classes 10–19): **Non-sorting Programs**.

be predicted by the initial permutation without executing the actual program - scrambler and random swap.

3.2. Training & testing methodology

The dataset described in the previous subsection is used for conducting a series of experiments designed to test the applicability of machine learning models to visual representations of program execution traces. As a general rule, the experiments are performed as follows:

- A synthetic training dataset is generated by running various algorithms on random inputs. Using the ground-truth class/property of the program, the models are trained to infer the correct class/property.
- Similarly, we generate a test dataset, which is unknown to the models. Then the models need to infer the correct class/property without access to the ground-truth. This way, we evaluate the capability of the models to learn about programs from their execution traces.

3.3. Experimental methodology

The experiments are grouped in three sections:

- Section 4 **Program Classification**: programs are run on various inputs, thus generating an execution trace. One of the tasks is to train a model to recognize which program generated a given previously unseen execution trace. Other variations of this task are included: inferring properties of the program generating the execution trace, such as whether it is a sorting or a non-sorting program. The motivation behind this set of experiments is to validate that program properties and classes can be inferred based on the isualizations of their execution traces.
- Section 5 **Pattern Detection in Sequences**: different programs are run in a sequence on a random input. The task of the learning model is to split the sequence of execution traces into sub-sequences belonging to the execution of a single program. The purpose of this experiment is to validate that program classification can be extended to program detection, which would get us closer to applying machine learning models to real life cases, where we typically see numer-

Table 2
Models used in the program classification experiments.

Model	Description	Input Representation
MLP1	Multi-Layer Perceptron: one hidden layer of 1.024 neurons.	No correlation between pixels in IoAs - columns of pixels are concatenated into a single vector (**order of pixels is irrelevant**).
CNN11	Convolutional Neural Network: one layer of 32 convolutions (size 2x2) and maxpool (size 2x2), and one fully-connected layer of 1.024 neurons.	Spatial correlation in the input (**structural patterns**) - 2D structure of the image is retained.
LSTM1	Long Short-Term Memory: one layer of 64 LSTM cells.	Temporal correlation in the input (**dynamical patterns**) - the image becomes a time-series of rows of pixels.

ous programs or routines performed one after another, with no clear boundary between semantically different parts.

- Section 6 **Algorithm Prediction and Behavior Emulation**: models need to learn how to replicate the execution of an algorithm for a given input. This is done in two steps: first a model learns to predict the next step in the execution of an algorithm. Then these predictions are used to replicate the algorithm's execution process.

3.4. Comparison methodology

We compare the results obtained in the experimented conducted by us with other approaches in the literature that use similar setups. This is done in Section 7 **Discussion and Comparison to Other Approaches**. The comparison is not so straightforward because the inputs, outputs, and the methodology are different in different approaches, but we do take into account as many details as possible.

4. Program classification

Automated classification of programs is a new topic in machine learning, so we extend here the classification experiments reported in our earlier work on IoAs (Perticas et al. [21]). A few more algorithms are added for which patterns are learned using neural networks, an additional learning model is tested, and two more classification experiments are conducted.

We made two choices for designing learning models. The first is to have a neural-network based architecture to allow the model to learn the features so that they do not need to be handcrafted. The second choice is to have models that assume different correlation types in the input, as shown in Table 2.

These models use the *ReLu* activation function and the *Adam* optimizer. The datasets consist of 1.000 samples per class ⇒ 10.000 – 20.000 samples (depending on the experiment performed), out of which 80% are used for training and 20% are used for validation. The number of training epochs is 20 and a batch contains 20 samples.

Two datasets are generated: one with 10-element arrays - used for the initial experiments; and another one with 20-element arrays - for checking how the classification scales up in terms of time complexity.

- **10-element arrays**: Represented by IoAs with 10 rows and 42 columns (equal to maximum number of time steps).
- **20-element arrays**: Represented by IoAs with 20 rows and 300 columns (equal to maximum number of time steps).

Different traces/IoAs can have different lengths - depending on the number of swaps performed, but in order to apply statistical learning on data, all the samples in the dataset must have the same size. Therefore, we use either padding with 0.0 values at the beginning or at the end until the sample reached the maximum length for data standardization.

4.1. Sorting vs. non-sorting program classification (SNSC)

In this experiment we are interested in determining if the trace of a program is that of a sorting algorithm (increasing or decreasing order, classes 0–9) or that of a non-sorting algorithms (classes 10–19). Thus, we have only two categories for this classification problem and the model should learn to distinguish a sorting program from a non-sorting program based on their IoAs.

Table 3
Accuracy results for **sorting vs. non-sorting program classification** by model and array size. First: 0-before-padding. Second: 0-after-padding.

Model	Acc 10	Acc 20	Acc 10	Acc 20
MLP1	99.6%	99.8%	99.6%	99.6%
CNN11	98.9%	99.9%	99.5%	99.9%
LSTM1	99.5%	99.9%	99.2%	100.0%

Table 4
Accuracy results for **sorting algorithm classification** by model and array size. First: 0before-padding. Second: 0after-padding.

Model	Acc 10	Acc 20	Acc 10	Acc 20
MLP1	94.3%	99.7%	95.0%	98.5%
CNN11	95.4%	99.9%	97.5%	99.8%
LSTM1	94.7%	99.5%	92.6%	98.1%

Accuracy report for classification on the validation dataset by model is shown in Table 3.

An interesting experiment on top of this problem would then be to test the network on new sorting and non-sorting programs to see if the model is able to generalize its internal concepts about what sorting algorithms mean. Randomly choosing a class obtains 50% accuracy.

4.2. Sorting algorithm classification (SAC)

The goal here is to check how well a model can learn to label different sorting algorithms (5 sorting algorithms and 2 sorting orders - classes 0–9). We report accuracy of classification on the validation dataset by model in Table 4. Randomly choosing a class obtains 10% accuracy.

4.3. General program classification (GPC)

The goal of this experiment is to validate if the patterns of various programs (20 programs, classes 0–19) can be recognized and labeled with high accuracy by a single model. This way, we check how well classification of IoAs can scale up with the increase in the number of program classes. Our experiments show that the accuracy does not drop significantly, except in the case of the recurrent model. Accuracy report for classification on the validation dataset by the model is shown in Table 5. Randomly choosing a class obtains 5% accuracy.

An interesting problem is whether it is possible to classify IoAs with different levels of specificity - hierarchical classification. For instance, BubbleSort is considered a different algorithm if it sorts in increasing vs. decreasing order, but we could consider families of algorithms - composed of the same sorting algorithm for various orderings; and semantically related algorithms - all sorting algorithms, algorithms that use tree-like organization of data, etc. This way, whenever we encounter

Table 5
Accuracy results for **general program classification** by model and array size. First: 0before-padding. Second: 0after-padding.

Model	Acc 10	Acc 20	Acc 10	Acc 20
MLP1	95.0%	99.7%	97.0%	98.6%
CNN11	95.7%	99.7%	97.5%	99.9%
LSTM1	92.6%	91.7%	86.5%	78.9%

an algorithm which was not seen before, we can classify it at a level of specificity where the model has high confidence.

4.4. Results discussion

Two trends are noticeable: the convolutional network seems to be the most robust; it generally performs the best, and the results do not drop below 95% accuracy under the tested settings. This could be attributed to the architecture of the network, which exploits the spatial structure of the data. However, this model is the most time consuming to train on CPUs. The second trend seen in the performance data of our models is that they perform better on the dataset with larger arrays. This comes as a surprise, but an intuitive explanation is that larger arrays imply more data to learn from, both in the number of time steps used by each program, as well as in the number of elements swapped.

The multi-layer perceptron (MLP) and the convolutional network (CNN) display similar results, with the CNN having an edge over the MLP (typically between 0.0% and 2.0% increase in accuracy). The type of padding performed does not influence the result in their case. However, for the recurrent network, the results drop significantly when padding is done after (between 0.3% and 13.0% decrease in accuracy), confirming the fact that recurrent networks are sensitive to the recency factor in the data - they ”forget”.

4.5. Scaling discussion

In the selected algorithms, the worst-case scenario for time complexity is expected to be $O(N^2)$. Although more than half of them should belong to $O(N)$ or $O(Nlog(N))$, after standardization their time-step wise length grows to $O(N^2)$. This means that when we double the length of the array (example: from 10 elements to 20 elements), we should expect the number of columns in the IoA to double (from N to $2N$) and the number of lines in the IoA to quadruple (from N^2 to $(2N)^2 = 4N^2$).

The overall increase factor in the data is the product between the *element-wise factor* = 2 and the *time-step-wise factor* = 4, which is 8. In practice, the data shows a size increase factor of 7.1 (from 420 values to 3.000 values per sample).

4.6. Remarks and limitations

The results from this section show that neural networks can learn the patterns of algorithms. We were careful to check that the learned models do not rely on simple observations such as the number of time-steps (for distinguishing between sorting algorithms) or the final order of the array elements (for distinguishing sorting from non-sorting programs). One basis for this conclusion is that we obtain similar results for different types of padding. Another reason is that the general program classification works well even though some programs require almost the same number of time-steps to finish. Although the array sizes are fixed, there are ways to work around this issue. For instance, if the array used by a program is larger than the one on which the network was trained, we can select a subset of the array's elements or intervals of length equal to the one used for training.

5. Pattern detection in sequences

Now that we have demonstrated that neural networks can distinguish between sorting and non-sorting algorithms based on the Image of Algorithm, the next challenge is to see if program classification can be extended to program detection. For this, we explore the potential of sequential learning models like the recurrent neural network for splitting and labeling parts of IoAs in intervals of computations which are semantically similar. The program classification results indicated that IoA detection could be feasible in longer sequences of memory traces. For an IoA with a succession of patterns representing various programs, we want to split it into separate logical blocks and specify the pattern to which they belong. If we are able to achieve this, it would bring us closer to applying machine learning models to real life cases, where we typically see numerous programs or routines performed one after another, with no clear boundary between semantically different parts

A sequential learning model can label the program being executed at any given time step by looking back in time a certain number of steps. Based on this information, the model is able to tell the state of the program. Specifically, we slide a window through the sequence and for each position we generate a label corresponding to the sequence covered by the window. These labels are then used to decide which are the logical chunks being executed by a single program and what is that program. Next we describe specific experiments elaborate on this approach.

5.1. Sorting vs. non-sorting sequences (SNSS)

We consider the task of splitting the execution trace of one program into segments which display a **sorting pattern** and segments which do something else - **non-sorting pattern**. For this experiment we use 10 different algorithms (5 sorting algorithms and 5 non-sorting algorithms, selected from the initial pool of 20 program classes).

Sorting Algorithms: *InsertionSort* (class 0), *BubbleSort* (class 2), *HeapSort* (class 4), *QuickSort* (class 6), *RandomSort* (class 8). All sorting algorithms use increasing order.

Non-sorting Algorithms: *Reverse* (class 10), *Reverse* (class 11), *IntervalSwap* (class 12), *IntervalSwap* (class 13), *RandomSwap* (class 14).

Detection Steps Example:

Fig. 4 shows the sequence of steps in the detection of IoA. First we have the raw input sequence (*RAW*), then we have the ground-truth segmentation of the sequence - the delimitation of intervals where a single algorithm is applied (*SEGMENTED*). *EXPLICIT* shows the category of algorithms applied to generate the input sequence. Each algorithm takes as input the output array of the previous algorithm and runs a few iterations.

Explanation

Notice that there are several consecutive different sorting patterns of various lengths. However, we would like to map all of them to a single sort segment. The same rational is applied to non-sort segments. The desired output from the detector is a list of 4 intervals:

- InsertionSort, BubbleSort → Sorting
- Reverse, IntervalSwap → Non-sorting
- HeapSort, QuickSort, RandomSort → Sorting
- Reverse, IntervalSwap → Non-sorting

5.2. Step labeling

Suppose A is the IoA on which we want to perform the sequencing algorithm - split into intervals with similar semantic. $A_{i,t}$ is the i-th element of the array at time-step t. $A_{i,0}$ is the initial array.

- N - Length of the array, number of rows in IoA.
- T - Number of time-steps, number of columns in IoA.
- L - Chunk size.

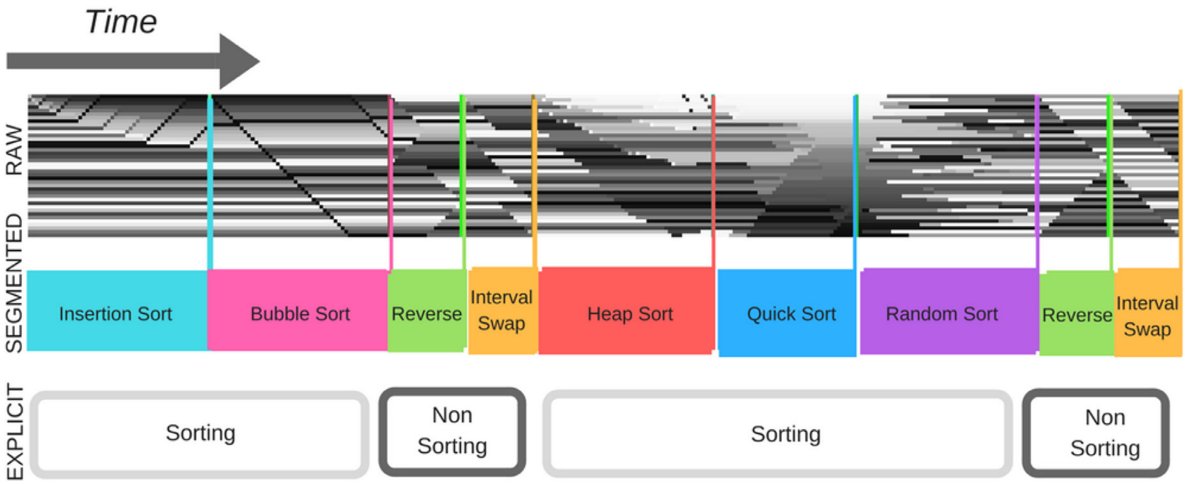


Fig. 4. Sequence of alternating programs on an array with 20 elements. *RAW*: the input IoA sequence. *SEGMENTED*: the segmentation into separate programs. *EXPLICIT*: detection of program categories: sorting vs. non-sorting.

Table 6
Time Step labeling performance by array size and chunk size.

Array Size	Chunk Size	Train Accuracy	Test Accuracy
10	5	99.5%	88.7%
10	10	99.7%	88.2%
20	5	99.7%	81.0%
20	10	99.8%	86.2%

Then our model generates a sequence B_t for each time-step $t >= L - 1$, indicating a label associated to step t in the sequence. Sorting steps are labeled with 1 and non-sorting steps are labeled with 0. For a sequence containing the initial array and $T = 20$ iterations, with a chunk size $L = 5$, [5 iterations *Reverse*, 5 iterations *BubbleSort*, 5 iterations *InsertionSort*, 5 iterations *IntervalSwap*] - the model outputs a sequence:

- time-step = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
- labels = [-, -, -, -, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0]

The model is trained with 20.000 chunks representing sequences of operations from the IoA sequence. We vary the array size (10 and 20) and the chunk length (5 and 10). Another set of 20.000 chunks are used to validate the model - Table 6. The programs in the IoA sequence contain a variable number of iterations (up to 50 consecutive iterations per program). The learning model is made of a standard LSTM with 100 units and a softmax layer, trained in batches of 20 for 100 epochs. We used recurrent dropout to further improve the results.

5.3. Sequence reconstruction

The predicted time-step labels can be used to reconstruct the initial sequence. This means that we need to specify which are the sorting and non-sorting intervals and how many iterations they contain. Since small time-step errors can contribute considerably to the difficulty of correctly reconstructing the sequence, we consider three measures of comparison against the correct segmentation:

Sub-basic Match: The sequence is split into the correct number of intervals.

Basic Match: A sub-basic match and the correct label prediction for each interval.

Perfect Match: A basic match and the correct number of iterations for each program in the sequence.

5.4. Results and discussion

The results for time-step labeling (training and validation) are presented in Table 6. For reconstructing the sequence of executed programs we use 100 new IoA sequences with 100 iterations. These results are presented in Table 7.

These results can be further improved via post-processing methods, for instance, by approximating the over-segmentation level based on the time-step accuracy, we can estimate a threshold for the minimum number of occurrences of a time-step label to be considered reliable in a larger sequence of labels.

This seems to be a rather heuristic method to deal with the issue of detecting program transitions. However, this fact suggests that in the language of neural networks, programs or algorithms are not clearly separated from one another. Rather, they belong to what one could imagine to be a scenery or landscape of programs where similar programs display spatio-temporal correlations, which can be easily observed through the images of algorithms taken from the execution traces.

6. Algorithm prediction and behavior emulation

As opposed to discriminative models, generative models need to be able to reproduce patterns in data, instead of just recognizing them. We now tackle the task of predicting the behavior of a few algorithms based on their representation as IoAs. This task can be split into two sub-problems. First, we train a model to predict the next steps of the algorithm given the ground-truth. Then we use the predictor to generate the whole sequence of steps using the starting state of the algorithm and its own predictions.

We can emulate the patterns exhibited by a certain algorithm using this approach. Our target algorithms are sorting algorithms (classes 0–9) - this choice was based on the fact that sorting algorithms are simple, but at the same time they have more intermediate steps than other simple programs. They also display a variety of patterns on which we can test the performance of such a generative model.

6.1. Sorting algorithm prediction (SAP)

The first problem in creating an IoA pattern emulator is to design a model that can reliably predict what operations the algorithm is going to perform based on its last few steps. For this purpose, we collected 20.000 intervals of 10 consecutive swapping operations performed by the algorithm. We divided these into **8 swaps** that are known to the model and **2 swaps** that it has to predict - see Fig. 5. Using this data we

Table 7
Sequence segmentation results by array size and chunk size.

Array Size	Chunk Size	Time-step Accuracy	Sub-basic Match	Basic Match	Perfect Match
10	5	89.1%	25%	25%	15%
10	10	91.0%	52%	52%	33%
20	5	81.4%	9%	9%	5%
20	10	85.5%	25%	24%	17%

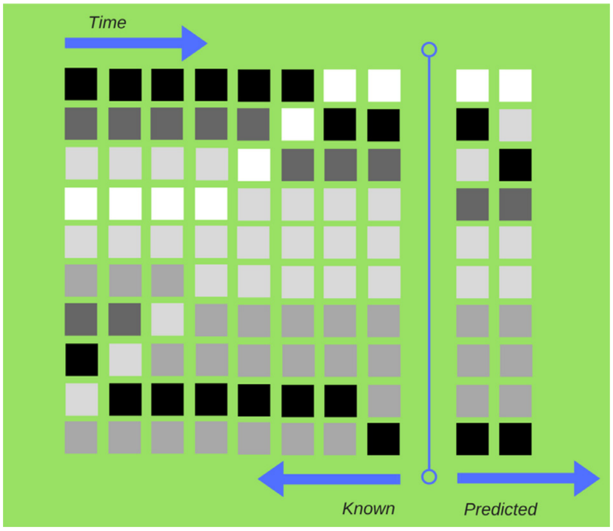


Fig. 5. Prediction of the next 2 swaps for sorting algorithms given the state of the last 8 swaps.

trained a long short-term memory model to guess the next two operations by regression. Our experimental setup is with IoAs for arrays with 10 elements, so the model needs to estimate the values for 2 operations x 10 elements = 20 elements. The model has 3 hidden layers, out of which 2 with 64 LSTM cells and a fully-connected one with 20 cells. It is then trained for up to 40 epochs (about 20 mins) using batches of 20 samples. The optimization method used for regression is *RMSprop*.

With the general assumption that the IoA is given by $A_{t,i}$ known for $t < S$ and $0 \leq i < N$, where S is the number of time-steps known (8 in our case) and N is the length of the array (10 in our case), then the problem is to predict $A_{t,i}$ for $S \leq t < S + P$ and $0 \leq i < N$, where P is the number of time-steps we want to predict (2 in our case). This translates into finding an approximate h such that $h(A_{t<S,i}) \approx A_{S\leq t<S+P,i}$.

This problem can also be modeled as a classification task, but then the IoA representation needs to be replaced with one that puts emphasis on the position of an element rather than its value - a permutation for instance. This approach should make it easier for the model to learn and we will discuss it in a later section. Thus, the regression setup we used to model this learning problem is a more challenging one because the model needs to learn to retain the values from the array and to perform the swaps in the same way as the emulated algorithm.

Although the values do not change in the case of sorting algorithms, this is how we would have to emulate a program when the exact operations or domain specific language are not known in advance.

The model minimizes the mean absolute error computed on the 20 values that need to be predicted. This metric is relative to the values of the array that are sampled uniformly from the interval [0.0, 1.0]. The results of predicting the next two operations for various algorithms after several epochs (up to 40) are shown in Fig. 6. After 10 epochs, the decrease in the error becomes very small, but it is at that point that the emulation starts to resemble the target algorithm.

The graphs display three trends: *QuickSort* obtains the lowest error for all measures. This could be because it is the most efficient method, so the predictor does not have to fit as many intermediate states as for the other algorithms. *RandomSort* has the least improvement across training epochs. We expect this since it is very hard to predict random behavior. *BubbleSort* has the worst starting error. Bubble sorting is different from the other methods because several possible swaps are valid if the location of the iterator is not known. The predictor needs to estimate its location based on the known operations in order to choose the correct swap.

A noticeable issue is the discrepancy between sorting in increasing and decreasing order for the same sorting algorithm in the case of MAPE loss. This might have to do with the fact that we ignore the error for predicted values close to 0.0, where MAPE is not defined. This is a known disadvantage of this error measure. However, this is the only error measure we computed that is not relative to the values in the array.

6.2. Sorting algorithm emulation (SAE)

The second problem we investigated is the actual algorithm emulation using the IoA representation. With the predictors trained in the previous section, we can generate images of the same size as the target images and compare them. Fig. 7 shows the progress in generating images of algorithms by the number of epochs used to train the predictors. The generative model begins predicting based on a few known initial steps and then uses its own predictions to generate the next steps. That is why the trained predictor needs to be very accurate. A small error in the prediction can lead to increasingly larger errors as the model relies more and more on its own output.

Similarly to the prediction task, the IoA is given by $A_{t,i}$ known for $t < S$ and $0 \leq i < N$, where S is the number of time-steps known and N is the length of the array, then the problem is to predict $A_{t,i}$ for $S \leq t < T$ and $0 <= i < N$, where T is the expected number of time-steps after which the pattern E ends, with the initial conditions $A_{t\,<\,S,i} \in E$ using h learned from a subset of E. The emulation is computed iteratively:

$A_{x\cdot P\leq t<S+x\cdot P,i} = concat(A_{x\cdot P\leq t<S+(x-1)\cdot P,i}, h(A_{(x-1)\cdot P\leq t<S+(x-1)\cdot P,i})), x = 1,\ldots,X$, until $S + X \cdot P > T$.

The learned emulations of sorting algorithms look convincing for *InsertionSort, BubbleSort* and *HeapSort* - the emulated images exhibit a similar pattern to the original algorithm. In the case of *Quicksort*, although the error metrics seem to point out that is should behave the best, the visual results are not convincing. As it uses only a few time-steps to sort, there is no time for it to create a noticeable pattern. *RandomSort* displays no pattern, so the model only learns how to sort and how to retain its original values (up to a point). An interesting fact is that more epochs of training lead to a trend in the improvement of the generated image, but there are exceptions when the emulated image resembles more the target image with less epochs of training.

6.3. Swap indexes prediction (SIP)

When modeling the problem of classification or prediction of IoAs, we used a visual representation of the process involved in executing a specific algorithm. However, we are dealing with programs that are based on the swap operation, so an alternative is to predict which indexes will be swapped. This task can also be solved in three ways, we

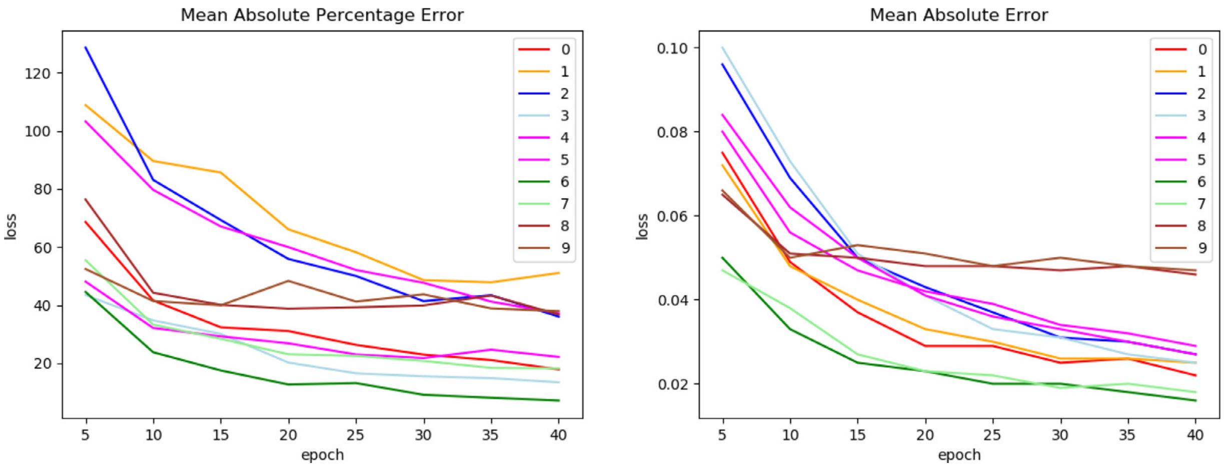


Fig. 6. Mean average percentage error (MAPE) and mean average error (MAE) for prediction of next 2 operations of 10 sorting algorithms (classes 0–9), values plotted by epoch.

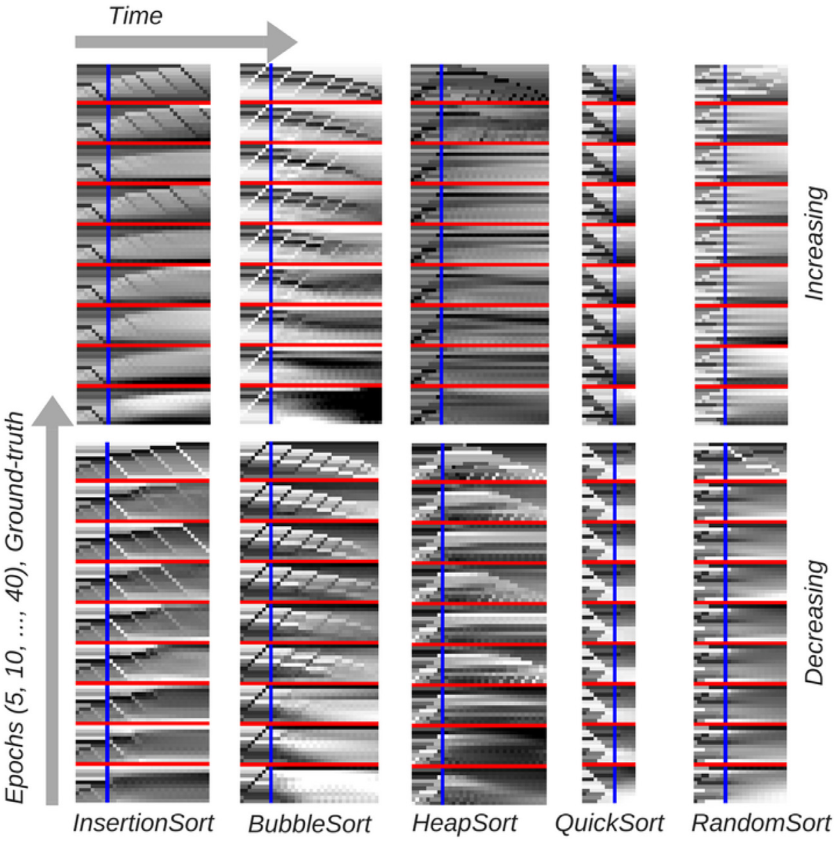


Fig. 7. Pattern emulation by algorithm, the images at the top of all sequences represent the ground-truth - how the sorting algorithm performed the task, the rest are the artificially generated patterns by the learning model after 5, 10, 15, 20, 25, 30, 35 and 40 epochs of training. The red lines separate the different emulations. The blue lines delimit the known values from the predicted/generated values. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

can use a sequence-to-sequence model, classify the indexes using the one-hot encoding with separate models, or estimate the position of the indexes via regression and discretization of the resulting values.

Fig. 8 shows the kind of emulations that can be made if we predict the indexes of the next swap, instead of the array values. We make the prediction based on the last eight arrays (steps), as in the SAP experiment. The model setup is the same, except for the output layer. Table 8 shows the accuracies obtained for sequence to sequence, classification and regression modeling. The results are reported for individual indexes and for one swap (2 indexes, *A* and *B*).

This study is carried only on algorithms for sorting in increasing order. The approach of predicting the swapped indexes has the advantage of 100% value preservation and a single swap per time-step in the emulated algorithm.

6.4. *Evaluation of generated patterns*

So far we have investigated the loss function values (MAPE and MAE) and the visual aspect of IoA emulations in order to get a sense of how good the generated images are. Next we aim to explore a number of metrics which should reflect how accurate the generative algorithm really is. For this reason, we investigated four metrics for each line in the emulated IoA - to check how the error increases as the generative model bases its predictions more and more on its own output:

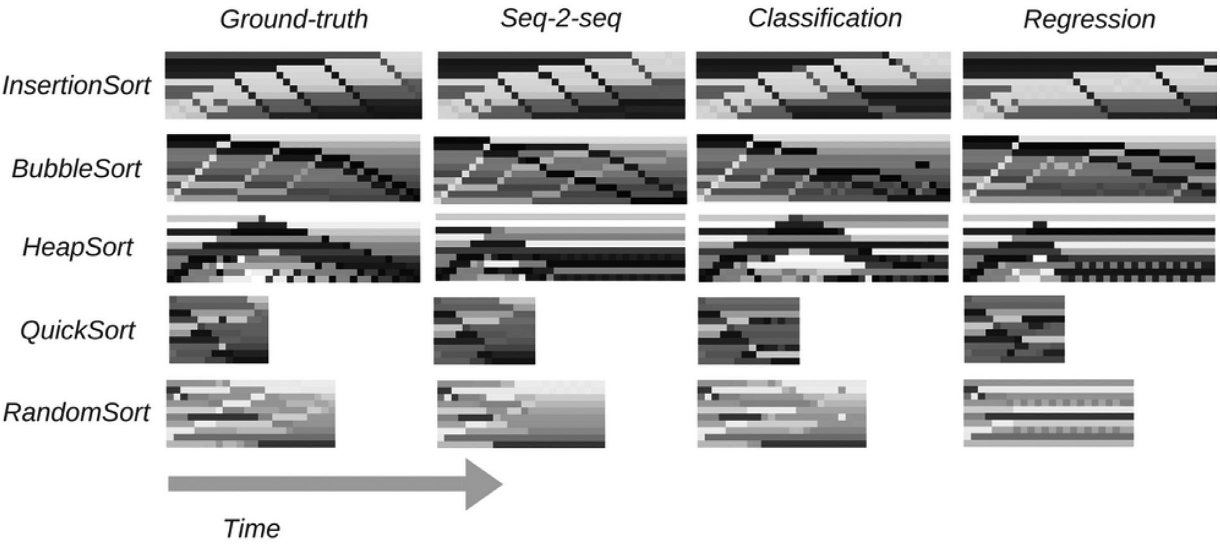


Fig. 8. Images of emulated IoAs by sequence-to-sequence, classification and regression of swapped indexes.

Table 8
Seq-2-seq vs. Classification vs. Regression. Accuracy for prediction of the next swap for sorting algorithms.

Algorithm	A	B	Swap	A	B	Swap	A	B	Swap
InsertionSort	97.8%	97.8%	97.5%	96.3%	95.3%	93.1%	90.3%	90.3%	90.1%
BubbleSort	91.0%	91.0%	89.9%	94.0%	94.0%	90.3%	80.9%	80.9%	80.3%
HeapSort	97.1%	92.6%	91.0%	97.0%	93.7%	91.6%	95.4%	89.5%	87.0%
QuickSort	91.8%	90.4%	86.0%	94.6%	93.9%	89.8%	77.9%	72.0%	65.8%
RandomSort	40.7%	39.9%	18.1%	38.0%	39.4%	15.4%	24.5%	24.8%	10.9%

Absolute Error: We compute the absolute pixel-wise difference between the ground-truth and the predicted images for each line of the target *R* and generated *G* images. If *i* is the index of the current line evaluated and *N* the array length, then:

$E_{absolute} = \frac{\sum_{j=0}^{N-1} |G_{i,j} - R_{i,j}|}{N}$

Value Preservation: To estimate how well the values from the target image *R* are preserved in the generated image *G*, we define a correspondence function *crsp* on G_i with values in R_i for line *i*:

$crsp(G_{i,j}) = R_{i,k}$ if $arg(G_{i,j}, sorted(G_i)) = arg(R_{i,k}, sorted(R_i))$, where $arg(v, V) = p$ if $v = V_p$

In other words, two elements in the target and generated images lines correspond if they appear in the same position in the sorted lines.

$E_{preserve} = \frac{\sum_{j=0}^{N-1} |G_{i,j} - crsp(G_{i,j})|}{N}$

Swap Accuracy: Number of swaps: All of the target emulated algorithms perform one swap per line in an IoA. However, the emulated algorithms tend to perform more swaps per line. This metric helps us identify their number. We use again the correspondence function, but between consecutive lines in the same image:

$crsp(G_{i,j,G_{i-1}}) = G_{i-1,k}$ if $arg(G_{i,j}, sorted(G_i)) = arg(G_{i-1,k}, sorted(G_{i-1}))$

$A_{numSwaps} = \sum_{j=0}^{N-1} isSwapped(G_{i,j})$

$isSwapped(G_{i,j}) = 1$, if $arg(G_{i,j}, sorted(G_i)) > arg(crsp(G_{i,j}, G_{i-1}), sorted(G_{i-1}))$

$isSwapped(G_{i,j}) = 0$ otherwise.

Swap Accuracy: Distance between swaps: Using the swaps identified with the previous metric, we now compute the minimum distance between them and the ground-truth swap. For an array with ten elements, this is four in the worst case scenario.

$A_{dstSwaps} = \min(\frac{|a-a'| + |b-b'|}{2})$

$a = arg(G_{i,j}, sorted(G_i)), b = arg(crsp(G_{i,j}, G_{i-1}), sorted(G_{i-1}))$ for *j* s.t. $isSwapped(G_{i,j}) = 1$

$a' = arg(R_{i,j}, sorted(R_i)), b' = arg(crsp(R_{i,j}, R_{i-1}), sorted(R_{i-1}))$ where $isSwapped(R_{i,j}) = 1$

The following two metrics are for discrete representations used for swapping indexes prediction and for comparison with similar experiments:

Over. Accuracy: how well the array is sorted overall.

Sort Accuracy: in how many cases the array is sorted correctly at the end.

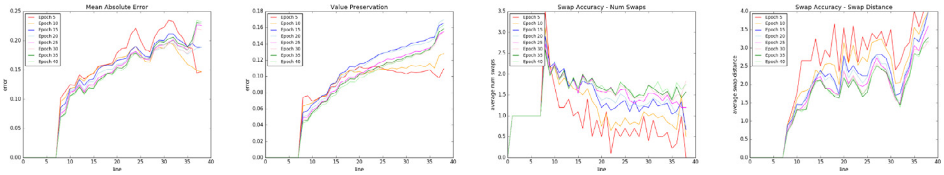
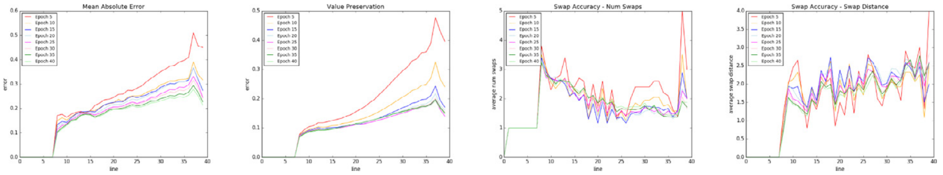
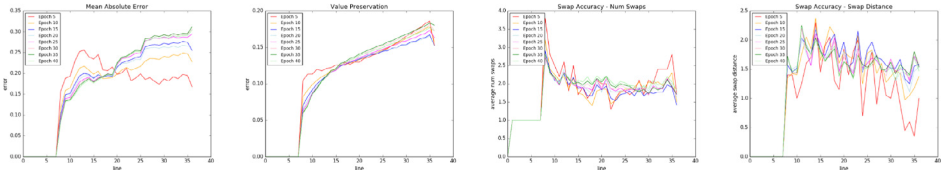
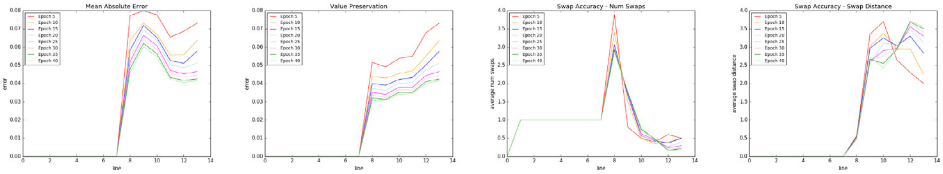
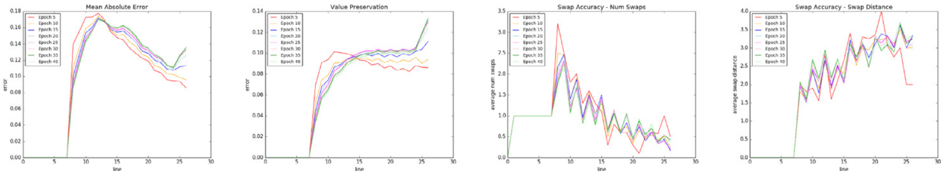
6.5. Analysis of results

According to the graphs in Table 9, the absolute error and value preservation are related metrics: both display a smooth function and diverge from the target value as the number of generated lines increases - though the convergence stabilizes after a number of steps. The number of swaps and the distance between swaps display many spikes and, although they have a general trend to fluctuate around certain values, they show unexpected jumps.

These two categories of metrics display two important trends: the value-based metrics display lower error values by increasing the number of training epochs, which means the model learns to preserve initial values across its training; the swap-based metrics oscillate less by increasing the number of training epochs, meaning that the model is finding some strategy for swapping. However, the number of swaps is typically larger than one.

This issue is overcome by re-formulating the problem as swap index prediction, which excels in the overall accuracy and final result correctness for some of the algorithms in our experimental set (very well for InsertionSort and fairly well for BubbleSort). In the case of HeapSort the predictions are correct only for short time ranges because of error accumulation. QuickSort does not have this problem due to its short length of execution, but immediate predictions are not as good as for the rest of the algorithms.

Table 9
Error metrics for emulation by number of epochs trained for SAE.

Method	Metrics Plot
InsertionSort	
BubbleSort	
HeapSort	
QuickSort	
RandomSort	

A considerable influence on the accuracy of the predicted results is posed by the difference in how the algorithms work. This is mostly evident for Heapsort, which organizes data into a tree structure. When using basic machine learning models, it is more difficult to infer the process used to reorganize data into a heap (tree/hierarchical structure) than it is for reorganizing data inside lists - eg. InsertionSort and BubbleSort. The reason for this is because trees are inherently more complex structures than lists, and so the models behind tree algorithms are represented by higher-order functions than the models for list algorithms.

6.6. Conclusions from the experiment

Based on our analysis of the experimental results, we can draw some conclusions about to what extent basic machine learning models can be used to infer the behaviour of programs from execution traces. There are four parameters about which we comment below as to whether our results are favorable or further research is needed.

- 1. Local patterns **YES**
- 2. Trend patterns **YES**
- 3. Good abstraction **NO - solution provided**
- 4. Error propagation **NO - possible solutions discussed**

1. Local patterns are learned well for most sorting algorithms (except RandomSort). This can be observed by looking at a short range across the time dimension in the emulated algorithm images - Figs. 7 and 8. In addition to this, the task of predicting a small number of future steps performed by a sorting algorithm is carried out successfully by basic neural models based on the obtained validation accuracy levels in Table 8.
2. Trend patterns are learned well in the case of InsertionSort and BubbleSort though with temporal distortions. Emulated RandomSort with swap prediction sorts the array at the end. The regression on memory values converges to a sorted sequence of numbers.
3. A better model would use the prior knowledge or simply observe better that the transitions between lines are represented by exactly one swap at a time step. This concern is addressed in the *Swap Indexes Prediction* set-up, where instead of predicting the in-memory values, the neural network model is re-designed to predict the indexes of the swapped cells. However, this approach is not very general and it would be desirable to leverage a model which is possible to re-use on programs defined on a variable set of granular operations such as swaps. Other models from related work perform similar predictions on bytes (copying, and sorting with an attention network accessing memory by Graves et al. [4] and Vinyals et al. [5]). Further improvements can be obtained by designing models with certain biases such as the relational neural networks for inferring answers to relational questions presented by Santoro et al. [22] and Hudson & Manning [23].
4. Error propagation is the process through which prediction errors accumulate as the neural program makes more moves based on its faulty predictions - for instance the emulated Heapsort in Fig. 8 does not get the list sorted, but the one in Fig. 7 learns to converge to a sorted state. This difference comes from the optimization procedure - what error to minimize (low level vs. high level outputs). Reducing error propagation can also be addressed as a matter of model representation. A faulty prediction leads the model to an input state which is more likely to not have been observed in the training data.

Task	Input Output Ratio	Data	Induction Type	Model	Over. Acc.	Sort Acc.
PrioritySort - [4]	160 (8x20) - 128 (8x16)	$\{0, 1\}$ - bits	Input-Output	NTM	**87.5**	?
PrioritySort - [4]	160 (8x20) - 128 (8x16)	$\{0, 1\}$ - bits	Input-Output	LSTM	62.5	?
InsertionSort - SIP	80 (10x8) - 100 (10x10)	$[0, 1]$ - unif.	Execution Trace	LSTM	**89.6**	**yes**
InsertionSort - SIP	80 (10x8) - 160 (10x16)	$[0, 1]$ - unif.	Execution Trace	LSTM	84.0	yes
InsertionSort - SIP	80 (10x8) - 200 (10x20)	$[0, 1]$ - unif.	Execution Trace	LSTM	80.0	yes
Sorting - [5]	10 (10x1) - 10	$[0, 1]$ - unif.	Input Output E2E	PtrNet	?	28.0
Sorting - [5]	10 (10x1) - 10	$[0, 1]$ - unif.	Input Output E2E	RPWNet	?	44.0
Sorting - [5]	10 (10x5) - 10	$[0, 1]$ - unif.	Input Output E2E	**RPWNet**	?	**57.0**
Sorting	10 (10x3) - 10	$[1, 50]$	Feed-forward E2E	MLP 3	98.0	86.0
Sorting	10 (10x2) - 10	$[1, 50]$	Feed-forward E2E	MLP 2	100.0	100.0
Sorting	10 (10x1) - 10	$[1, 50]$	Feed-forward E2E	**MLP 1**	**95.0**	**57.0**

Fig. 9. Comparison of our best performing emulated algorithm - **InsertionSort** LSTM against **PrioritySort** in [4]. Comparison with direct task learning (sorting) - feedforward MLP and sequential in [5]. **PrtNet**: Pointer Network, **RPWNet**: Read-Process-Write Attention and Memory Network. **NTM**: Neural Turing Machine (memory and controller). **LSTM**: Long short-term memory. **MLP**: Multilayer Perceptron (feedforward). **E2E** End-to-end result prediction (eg. direct sorted order from input sequence). **Input Output** models are all sequential. Compared against **feed-forward** and **execution trace** - sequential. **Over. Acc**: overall accuracy across multiple steps. **Sort Acc**: accuracy of correct results at the end - whether array sorted. **Input Output Ratio**: number of inputs and outputs in the setup. Number of times processed/Number of layers in feed-forward: x N.

However, models equipped with memory or prior knowledge in the form of DSL can be used to prevent distorting input data (generate less unobserved data) and restrain search space (reduce chances of getting lost).

7. Discussion and comparison with other approaches

Replicating complex dynamical behavior generated from simple patterns (such as CAs or IoAs) using neural networks is a hard task in itself. First of all there are hard theoretical constraints which have an impact on the limitations of program induction techniques - eg. Turing completeness of NN models in Perez et al. [24]. The paper compares properties of two NN models (Universal Transformer and Neural GPU) designed for learning algorithms from examples.

Then there are the practical limitations which can be understood by looking at different similar experiments performed on synthetic datasets. Graves et al. [4] performs computer memory manipulation to infer/predict simple computer programs (copying, sorting). The operations of copying and sorting an array are performed with a fully differentiable network connected to an eternal memory via attention. The model is further elaborated by Graves and Wayne [15] and used to solve problems involving trees and graphs, which are considerable more challenging than those involving lists.

In another approach, Vinyals et al. [5] presents a sorting experiment for sets of numbers using a sequence to sequence model working through read and write processes, an extension of the pointer network by Vinyals et al. [16] - a neural programming approach for solving path finding and geometry problems.

A comparison of these experiments is presented in Fig. 9.

A third piece of research is by Balog et al. [6], which experiments with simple programs involving lists and synthesizes solutions from input-output pairs with a domain-specific language. Yet another approach from Parisotto et al. [25], which makes a case for program synthesis applied to string transformations by learning to expand partial programs in a context-free grammar.

These cases do not typically report any results on setups where the observed variables (data points) exceed 10–20 elements. We ran some additional tests which are not presented in this paper, and it seems that predicting the sorted order of an array works accurately for up to 30 elements if the neural network is properly engineered.

Finally, Devlin et al. [7] explores the difference in formulating string transformations as a program synthesis vs. a program induction task and Kraska et al. [26] takes the problem of program induction to the domain of data structures and provides an approach for implementing them using deep learning, leading to increased efficiency based on the properties of the data distribution.

Using intermediate steps as data points, such as described by Ling et al. [27], can boost the performance of neural program induction. The recursive compositionality model by Ling et al. [28] can be used to process recursive algorithm traces and their relation to neural networks.

Jaeger [29] performs function approximations with RNNs on chaotic attractors and body movement patterns (walking, running, dancing). Then introduces Conceptors as a compressed form of pattern representation which can drive and combine the execution of learned behavior.

In some cases, simply learning the answer to a problem is not good enough - either because there is the need of understanding the principles behind the decisions made or because the end-to-end solution is too difficult to learn. This can be done by learning the sequence of atomic operations that lead to solving the problem, and it differs from end-to-end program induction such as learning a program directly from its input-output pairs. We use a similar approach when training neural networks to sort an array based on intermediate operations.

8. Conclusions

Our experiments show that neural networks are capable of modeling the mechanisms underlying simple algorithms if enough execution traces are provided as data. Moreover, we were able to demonstrate that programs can be treated both as images displaying spatial patterns, as well as sequential instructions in a domain specific language - swap, first position and second position. We tested our approach on three types of increasingly harder tasks: detection, recognition and emulation.

It should be noted that though the three different problem setups have the same state space, they are conceptually different from each other. They reveal how the same object (the state space of an algorithm) exhibits characteristics with varying degrees of difficulty in static (detection), temporal (recognition in a sequence), and generative (emulation) environments.

Out of these, detection seems to be the easiest to learn in a setup with fixed dimensions; and scales well given enough computational time. Recognition of algorithms within longer sequences of execution traces poses problems regarding the length of algorithm blocks, and how to treat sub-sequences containing more than one algorithm; for instance when one algorithm is stopped and a second one is started. Finally, emulation of algorithms shows a more complex behavior and seems to be the hardest to model as an optimization task. As a result, certain loss functions perform well for some classes of algorithms, but poorly for others.

The representation of the data used in learning shows duality: the use of the swapping conceptualization makes the network very effective in the short run, but exhibits worse long-term results than the raw data.

Given our program induction set-up, basic neural networks seem bad at tough decisions (decisions where very exact predictions are required).

The take-away message here is that emulation is too complex for a neural network to learn completely: some aspects of it can be learned, but not precisely and not at the same time. Answers could be found by modeling aspects of memory manipulation in a more granular manner - like moving a pointer, performing swap etc, then trying to re-create these simpler operations.

The conceptualization of swaps for the neural network can improve results, but only in conjunction with additional information. Another aspect of this is that the raw data provides information that the neural network cannot capture from knowing the correct swaps (the end result or the target of its learning process). So tackling the problem of generating more general algorithms exhibiting a given behavior still poses challenges, including how to evaluate an approximate statistically learned algorithm.

The framework and the experiments presented for emulating images of execution traces provide results which are in some cases more accurate than those obtained with more complex machine learning models, leading to the conclusion that observing an algorithm's behavior along its execution timeline can overcome barriers in program induction that remain challenging even with very complex machine learning models. However, these models show further methods which might be used to improve the results of the execution trace approach. This is why the report focuses on results in the direction of problem formulation. Our findings indicate that efficient neural network models for this domain would have to obey principles of problem formulation (input-output, execution trace), specific problem solving concepts (eg. pointers, relations, attention, memory, operations) and design (sequential, feed-forward models). Putting these principles to work inside a machine learning model, might then generalize the ability to learn programs from noisier or more complex execution traces.

9. Future work and applications

One way to create novel sorting algorithms is by training a model with several sorting algorithms. For instance, we could train a neural network with InsertionSort and BubbleSort to obtain a hybrid sorting method that combines the two. By constraining the model to behave like a more efficient algorithm depending on its current state, the hybrid emulation could result in a faster sorting algorithm. This can be achieved by manipulating the training data. We can combine the data for which InsertionSort is efficient with the data for which BubbleSort is efficient at sorting. Such experiments have been done by Jaeger [29] - applications in maths and robotics and by Gatys et al. [30] - applications in art and computer vision.

A more general application where our framework could serve as background would be represented by the recorded actions of a programmer debugging code. Little changes in the source code can have a high impact on the trace left by the programmer debugging the application. It is rather through this tracing representation and where it points to occur in the source code that we find faulty definitions in a program than by simply looking through the source code. We could expect an intelligent programming agent to learn more efficient representations by using interactions (watching traces) in a virtual environment than by simply using the code definition (source code) and its end results. This is by assuming the agent does not have prior knowledge of how source code translates to machine code, which would be very difficult to integrate in an agent.

Consider the actions of a reinforced learning model as emulations of some previously learned algorithms. The actions of the agent would be to behave like a specific algorithm for a chosen number of steps. Based on the rewards received, the agent could optimize certain tasks by correctly deciding the order and the subset of algorithms to apply. An example of such situation is when an almost increasingly sorted array needs to be sorted in the decreasing order. Here, a very effective strategy would be to reverse the array first and then apply a sorting algorithm in the decreasing order.

Classification and detection of programs have applications in malware detection by Pascanu et al. [12]. Emulation of algorithms shows a potential for program optimization according to Kraska et al. [26], data manipulation and stable program execution. For instance, hybridization and reinforced emulation could save computational time by heuristic assessment of the best strategy based on the distribution of the input data. Simple emulation allows running an algorithm from a break-point with potentially different parameters without having to ”re-start” the algorithm. Managing programs this way could have applications in situations when fatal and unexpected errors occur in the middle of the execution of a program. Classification and detection of programs could be used for bug detection and malicious software recognition. Moreover, hierarchical clustering of programs based on semantics offers insight into the kind of issue presented by the analyzed programs.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Cătălin F. Perticas: Writing - original draft, Visualization, Validation, Software, Investigation, Data curation. **Bipin Indurkhya:** Conceptualization, Methodology, Supervision, Writing - review & editing.

Acknowledgement

This work was supported by the European Regional Development Fund and the Romanian Government through the Competitiveness Operational Programme 2014–2020, project ID P_37_679, MySMIS code 103319, contract no. 157/16.12.2016.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.infsof.2020.106350.

References

[1] E. Kitzelman, Inductive programming: a survey of program synthesis techniques, International Workshop on Approaches and Applications of Inductive Programming, 2009.
[2] S. Gulwani, O. Polozov, R. Singh, Program synthesis, Found. Trends Program. Lang. 4 (1–2) (2017).
[3] N. Kant, Recent advances in neural program synthesis (2018).
[4] A. Graves, G. Wayne, I. Danihelka, Neural Turing Machines (2014).
[5] O. Vinyals, S. Bengio, M. Kudlur, Order Matters: Sequence to Sequence for Sets, in: 4th International Conference on Learning Representations (ICLR), 2016.
[6] M. Balog, A.L. Gaunt, M. Brockschmidt, S. Nowozin, D. Tarlow, DeepCoder: learning to write programs, in: 5th International Conference on Learning Representations (ICLR), 2017.
[7] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, P. Kohli, RobustFill: neural program learning under noisy I/O, in: 34th International Conference on Machine Learning (ICML), 2017.
[8] A. Krizhevsky, I. Sutskever, G. Hinton, Imagenet classification with deep convolutional neural networks, Adv. Neural Inf. Process. Syst.25 (NIPS) (2012).
[9] A. Graves, Generating sequences with recurrent neural networks (2013).
[10] M.D. Shah, S.Z. Guyer, Critical section investigator: building story visualizations with program traces, in: IEEE Working Conference on Software Visualization (VISSOFT), 2016.
[11] H.N. Huang, E. Verbeek, D. German, M. Storey, M. Salois, Atlantis: improving the analysis and visualization of large assembly execution traces, in: IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017.
[12] R. Pascanu, J.W. Stokesy, H. Sanossianz, M. Marinescu, A. Thomas, Malware classification with recurrent networks, in: 40th IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2015.
[13] F. Fittkau, S. Finke, W. Hasselbring, J. Waller, Comparing trace visualizations for program comprehension through controlled experiments, in: IEEE 23rd International Conference on Program Comprehension (ICPC), 2015.

[14] J. Devlin, R. Bunel, R. Singh, M. Hausknecht, P. Kohli, Neural Program Meta-Induction, in: 31st Conference on Neural Information Processing Systems (NIPS), 2017.
[15] A. Graves, G. Wayne, Hybrid computing using a neural network with dynamic external memory, Nature (2016).
[16] O. Vinyals, M. Fortunato, N. Jaitly, Pointer Networks (2016).
[17] S. Wolfram, Statistical mechanics of cellular automata, Rev. Mod. Phys. (1983).
[18] K. Rawat, V. Katiyar, P. Gupta, Two-lane traffic flow simulation model via cellular automaton, Int. J. Veh. Technol. (2011).
[19] H. Guzmán, M. Lárraga, L. Alvarez-Icaza, F. Huerta, A realistic two-lanes traffic simulation model based on cellular automata, Eur. Model. Symposium (2014).
[20] L. Dioşan, A. Andreica, I. Boros, I. Voiculescu, Avenues for the use of cellular automata in image segmentation, in: European Conference on the Applications of Evolutionary Computation, 2017.
[21] C.F. Perticas, B. Indurkhya, R.V. Florian, L. Csato, Finding patterns in visualizations of programs, Proc. Psychol. Program. (2017).
[22] A. Santoro, D. Raposo, D. Barret, M. M., R. Pascanu, P. Battaglia, T. Lillicrap, A simple neural network module for relational reasoning (2017).
[23] D. Hudson, C. Manning, Compositional attention networks for machine reasoning, in: 6th International Conference on Learning Representations (ICLR), 2018.
[24] J. Pérez, J. Marinković, P. Barceló, On the turing completeness of modern neural network architectures, in: 7th International Conference on Learning Representations (ICLR), 2019.
[25] E. Parisotto, A. Mohamed, R. Singh, L. Li, D. Zhou, P. Kohli, Neuro-symbolic program synthesis, in: 5th International Conference on Learning Representations (ICLR), 2017.
[26] T. Kraska, A. Beutel, E. Chi, J. Dean, N. Polyzotis, The case for learned index structures, in: Proceedings of the 2018 International Conference on Management of Data Pages, 2018, pp. 489–504.
[27] W. Ling, D. Yogatama, C. Dyer, P. Blunsom, Program Induction by Rationale Generation: Learning to Solve and Explain Algebraic Word Problems (2017).
[28] R. Socher, B. Huval, C.D. Manning, A. Ng, Semantic compositionality through recursive matrix-vector spaces, in: Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural, 2012.
[29] H. Jaeger, Controlling recurrent neural networks by conceptors, Technical Report 31, Jacobs University, 2014.
[30] L. Gatys, A. Ecker, M. Bethge, A neural algorithm of artistic style (2015).