From Programs to Interpretable Deep Models & Back

Eran Yahav
Technion & Codota
## Analysis and Synthesis with “Big Code”

Programming Languages + ML + IR + ...

<table>
<thead>
<tr>
<th>Title</th>
<th>Conference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code2vec: Learning Distributed Representations of Code</td>
<td>POPL’19</td>
</tr>
<tr>
<td>On the Practical Computational Power of Finite Precision RNNs for Language Recognition</td>
<td>ACL’18</td>
</tr>
<tr>
<td>Extracting Automata from Recurrent Neural Networks Using Queries and Counterexamples</td>
<td>ICML’18</td>
</tr>
<tr>
<td>A General Path-based Representation for Predicting Program Properties</td>
<td>PLDI’18</td>
</tr>
<tr>
<td>Synthesis of Forgiving Data Extractors</td>
<td>WSDM’17</td>
</tr>
<tr>
<td>Lossless Separation of Web Pages into Layout Code and Data</td>
<td>KDD’16</td>
</tr>
<tr>
<td>Statistical Similarity of Binaries</td>
<td>PLDI’16</td>
</tr>
<tr>
<td>Estimating Types in Stripped Binaries</td>
<td>POPL’16</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Synthesis from partial programs</td>
<td>OOPSLA’12</td>
</tr>
</tbody>
</table>

https://github.com/tech-srl/  
http://code2vec.org

PRIME  
TRACY  
DIZY  
Like2Drops

https://www.codota.com

---

Sharon Shoham  
(Prof. @TAU)

Yoav Goldberg  
(Prof. @BIU)

Omer Levy  
(FAIR)

Hila Peleg

Dana Drachsler

Shir Yadid

Meital Zilberstein

Adi Omari

Nimrod Partush

Omer Katz

Yaniv David

Uri Alon

Gail Weiss

Yoav Goldberg  
(Prof. @BIU)

Omer Levy  
(FAIR)
Lots of code available on the web

Learn from all the code out there to make software development faster and smarter
Why now?

- Big code
- Static program analysis
- Machine learning
- Computation resources
Augmented Programmer Intelligence

• Predict code (preventive software quality)
  • Automate mundane tasks
  • Keep devs on the main path
  • “likely by construction”

• Check code
  • Standardize on practices learned from code
From programs to models and back

• Neural networks for **predicting program elements**

• What is it that a network **has learned**?
  • A model that provides some explanation
  • A new technique for extracting explanation (DFA) from a recurrent neural network

• What is it that a network **can learn**?
Code2vec [POPL’19]

- A neural network for **predicting program elements from context**
- Learns name & path embeddings, simultaneously learns to aggregate them
- Example: **predicting method names** = ~14M training methods (1 day on a GPU)

Method names task: Suggesting accurate method and class names. [Allamanis, FSE 2015]
How does it work?
Learn tag distribution conditioned on code

\[ P(L | C) \]

- Using syntactic paths in C
Background: Distributed Representations ("embeddings")

"One-hot" vectors

<table>
<thead>
<tr>
<th>Word</th>
<th>hello</th>
<th>world</th>
<th>how</th>
<th>are</th>
<th>you</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector</td>
<td>[0,0,...,0,1,0,0]</td>
<td>[0,0,...,0,1,0,0]</td>
<td>[0,0,...,0,1,0,0]</td>
<td>[0,0,...,0,1,0,0]</td>
<td>[0,0,...,0,1,0,0]</td>
<td>[0,0,...,0,1,0,0]</td>
</tr>
</tbody>
</table>

Distributed representations

<table>
<thead>
<tr>
<th>Word</th>
<th>hello</th>
<th>world</th>
<th>how</th>
<th>are</th>
<th>you</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector</td>
<td>[0,0,...,0,1,0,0]</td>
<td>[0,0,...,0,1,0,0]</td>
<td>[0,0,...,0,1,0,0]</td>
<td>[0,0,...,0,1,0,0]</td>
<td>[0,0,...,0,1,0,0]</td>
<td>[0,0,...,0,1,0,0]</td>
</tr>
</tbody>
</table>

Each vector: (0,0,...,0,1,0,0)

sim("hello","hi") > sim("hello","you")
Neural networks in 30 seconds

• A sequence of simple algebraic functions over vectors and matrices

• **Example**: Predict a how positive is a sentence (regression)

\[ \text{vec} \leftarrow \text{vec} - \alpha \frac{\partial \text{loss}}{\partial \text{vec}} \]
\[ w \leftarrow w - \alpha \frac{\partial \text{loss}}{\partial w} \]
AST paths: a general method to represent code in machine learning models

```
while (!done) {
    if (someCondition()) {
        done = true;
    }
}
```

Code snippet represented as a set of all its syntactic paths
Key idea #1: encoding with path-attention

(source1, path, target1)

(source2, path, target2) → (, , )

(source_n, path, target_n)

Bag of contexts

Use embeddings to represent each path as vector in \( \mathbb{R}^{3d} \)
(concatenation of 3 vectors in \( \mathbb{R}^{d} \))

Fully-connected layer
\( \mathbb{R}^{3d} \rightarrow \mathbb{R}^{d} \)

Fully-connected layer
\( \mathbb{R}^{d} \rightarrow \mathbb{R}^{d} \)

prediction
boolean f(Object target) {
    for (Object elem: this.elements) {
        if (elem.equals(target)) {
            return true;
        }
    }
    return false;
}

Object f(int target) {
    for (Object elem: this.elements) {
        if (elem.hashCode().equals(target)) {
            return elem;
        }
    }
    return this.defaultValue;
}

Prediction
contains
matches
equals
containsExact
containsAll

Prediction
get
getProperty
getValue
getElement
getObject
Works great, but

• Monolithic labels
• Monolithic paths
• Huge vocabulary

• How can we do better?
Long Short-Term Memory (LSTM)

- A kind of Recurrent Neural Networks
  - Input: vector
  - Updates its internal memory vector
  - Output: vector

- Input: a sequence of vectors
- Output: a sequence of vectors

- Extremely good at learning sequences
- The basis for (almost) any model that learns sequences (e.g., machine translation, speech recognition, image captioning...)

http://colah.github.io/posts/2015-08-Understanding-LSTMs/
Key idea #2: efficient encoding of paths

```
int countOccurrences(String str, char ch) {
    int num = 0;
    int index = -1;
    do {
        index = str.indexOf(ch, index + 1);
        if (index >= 0) {
            num++;
        }
    } while (index >= 0);
    return num;
}
```

```
int countOccurrences(String source, char value) {
    int count = 0;
    for (int i = 0; i < source.length(); i++) {
        if (source.charAt(i) == value) {
            count++;
        }
    }
    return count;
}
```
Encode paths using an LSTM that "walks" on the AST

k(=200) contexts are randomly sampled every iteration
Code Completion
Recurrent neural networks (RNNs)

- Capture regularities of code in a language model: predict next “letter”
- Sentence can be anything: Character-level, token-level, sequence of API calls

```java
while (!done) {
    if (someCondition()) {
        doStuff();
        done = true;
    }
}
```

- Capture regularities of code in a language model: predict next “letter”
- Sentence can be anything: Character-level, token-level, sequence of API calls
Learning effort
(amount of training data, time, ...)

Implicitly re-learn language syntax

Program analysis effort
(semantic depth)

Language-specific / task specific

Program text

AST based

Semantic analysis

Dependence graphs

...
RNNs are awesome!

• Using LSTMs/GRUs to capture regularity of code in programs
• LSTMs/GRUs work well, but sometimes surprise us

• What do they actually learn?
  • Important to understand, especially when things go wrong
  • E.g., misclassification – can we provide examples that improve the net?

• What can they actually learn?
What has a network actually learned?
What has a network learned?

valid email addresses
40,000 training samples
2,000 test samples

Training
(100% accuracy on train, reached 100% also on test)

RNN

abc@sc.net
blbl@df.com
sf.se@sdf.co.uk
..@.co.
dasd@@.vim
b.net

But has it really learned to recognize valid email addresses?
What has a network learned?

valid email addresses
40,000 training samples
2,000 test samples

Training
(100% accuracy on train, reached 100% also on test)

RNN

abc@sc.net
blbl@df.com
sf.se@sdf.co.uk
..@.co.
dasd@@.vim
b.net

25.net
5x.nem
2hs.net
How can we find these flaws?

Valid email addresses
40,000 training samples
2,000 test samples

Training
(100% accuracy on train, reached 100% also on test)

RNN

abc@sc.net
blbl@df.com
sf.se@sdf.co.uk
..@.co.
dasd@@.vim
b.net

25.net
5x.nem
2hs.net
Setting: back to basics

• What happens if we train on a regular source?

• Learning regular languages
  • Train network for classification of words in a regular language
  • Try to extract a DFA that captures what the network has learned
Idea #0: Exhaustively explore net from initial state

- Consider outputs of recurrent units as “states”
- Explore net from initial state under inputs of increasing length
- Hope that (continuous) “states” start repeating
  - Net was trained on words from DFA
Exploring toy example

**Initial State**

- **Input = aab**
- Converted to one-hot-vectors: (1,0),(1,0),(0,1)
Exploring toy example

Input = aab
converted to one-hot-vectors: (1,0),(1,0),(0,1)
Exploring toy example

Input = aab
converted to one-hot-vectors: (1,0),(1,0),(0,1)

Initial State: (0,0)

Input alphabet: {a,b}
Exploring toy example

Input alphabet: {a,b}

Initial State: (0,0)

Input = aab
converted to one-hot-vectors: (1,0),(1,0),(0,1)
Exploring toy example

Initial State

RNN cell

RNN cell

RNN cell

Input = aab
converted to one-hot-vectors: (1,0),(1,0),(0,1)

Input alphabet: {a,b}

Initial State: (0,0)
Exploring toy example

Input = aab
converted to one-hot-vectors: (1,0),(1,0),(0,1)

Initial State

RNN cell
0

RNN cell
0

RNN cell
1,0

RNN cell
1,0

Input alphabet: {a,b}

Initial State: (0,0)
Exploring toy example

Initial State

Input = aab
converted to one-hot-vectors: (1,0),(1,0),(0,1)

Label Accept/Reject using hidden-to-output cell
Ideally...

Input alphabet: \{a, b\}
Initial State: (0,0)
In reality... potentially infinite unrolling

Values of state vectors do not actually repeat...
Hopefully, these can be **abstracted** into equivalent states.
Idea #1: fixed quantization [Omlin&Giles ‘92]

- Partition space and consider “close values” for outputs as same state

- Abstraction by quantization constant $q$, partitioning each dimension to equal parts
  - $q = 3$ induces $3^N$ discrete states ($N$ neurons)
  - Easy to see where this is headed for large values of $N$

- Induces an “abstract automaton”
Exploring abstract automaton (fixed quantization abstraction)
Exploring abstract automaton (fixed quantization abstraction)

Input alphabet: \{a,b\}
Initial State: (0,0)
Exploring abstract automaton (fixed quantization abstraction)

Input alphabet: \{a,b\}
Initial State: (0,0)
Can easily go wrong...

This unrolled automaton will not represent the network at all well.
Refinement quickly leads to explosion

Input alphabet: \{a,b\}
Initial State: \((0,0)\)

Not practical
Our approach: Use L* with abstract teacher

• Learner maintains a hypothesized DFA $A$ and refines it by posing queries to the teacher

• Teacher has to answer two kinds of queries
  • Membership query: $w \in L$ ?
  • Equivalence query: $L(A) = L$ ?

• Algorithm guaranteed to return minimal DFA

Synthesis of interface specifications for Java classes [Alur et al., POPL 05]
L* [Angluin, 1987] Refresher

Concept known to teacher $\varepsilon | a(a|b)^* a|b(a|b)^* b$
Exact learning with **abstract teacher**

- *Abstraction of the neural net as the teacher* (oracle)
  - Teacher *may be wrong due to abstraction*

- Membership queries are easy
  - *Run the network* on the given example and return accept/reject
  - Membership queries are *precise*
    - use the network directly
    - no abstraction

- Equivalence queries?
Equivalence queries

• Check equivalence between hypothesized automaton A and abstract automaton

• Use hypothesized automaton A to **guide construction of abstract automaton from the RNN**

• Two possible outcomes
  • Exploration terminates **without conflict** between network and A – the hypothesized automaton A is **equivalent** to the RNN representation
  • A **conflict** is detected between a state in A and a state of the network
Our Approach - Equivalence Queries

Check equivalence of partitioning-induced DFA and of L* DFA

Our Approach - Equivalence Queries

Check equivalence of partitioning-induced DFA and of L* DFA
Our Approach - Equivalence Queries

Check equivalence of partitioning-induced DFA and of $L^*$ DFA

Disagreements are checked by RNN for ground truth
Our Approach - Equivalence Queries

Check equivalence of partitioning-induced DFA and of \( L^* \) DFA

Disagreements are checked by RNN for ground truth
Check equivalence of partitioning-induced DFA and of L* DFA

Disagreements are checked by RNN for ground truth

Our Approach - Equivalence Queries

Check equivalence of partitioning-induced DFA and of L* DFA

Disagreements are checked by RNN for ground truth
Our Approach - Equivalence Queries

Check equivalence of partitioning-induced DFA and of L* DFA

Disagreements are checked by RNN for ground truth
Our Approach - Equivalence Queries

Check equivalence of partitioning-induced DFA and of L* DFA

Disagreements are checked by RNN for ground truth
If RNN and L* disagree, a counterexample is returned.
Our Approach - Equivalence Queries

Check equivalence of partitioning-induced DFA and of L* DFA
Disagreements are checked by RNN for ground truth
If RNN and L* disagree, a counterexample is returned.
Our Approach - Equivalence Queries

Check equivalence of partitioning-induced DFA and of $L^*$ DFA

Disagreements are checked by RNN for ground truth.
If RNN and $L^*$ disagree, a counterexample is returned.
Our Approach - Equivalence Queries

Check equivalence of partitioning-induced DFA and of L* DFA
Disagreements are checked by RNN for ground truth
If RNN and L* disagree, a **counterexample** is returned.
Otherwise, the partitioning is refined
Our Approach - Equivalence Queries

Check equivalence of partitioning-induced DFA and of L* DFA
Disagreements are checked by RNN for ground truth
If RNN and L* disagree, a **counterexample** is returned.
Otherwise, **the partitioning is refined**
Our Approach - Equivalence Queries

Check equivalence of partitioning-induced DFA and of $L^*$ DFA
Disagreements are checked by RNN for ground truth
If RNN and $L^*$ disagree, a **counterexample** is returned.
Otherwise, the partitioning is refined
Our Approach - Equivalence Queries

Check equivalence of partitioning-induced DFA and of L* DFA

Disagreements are checked by RNN for ground truth
If RNN and L* disagree, a **counterexample** is returned.
Otherwise, the partitioning is refined
Our Approach - Equivalence Queries

Check equivalence of partitioning-induced DFA and of $L^*$ DFA

Disagreements are checked by RNN for ground truth
If RNN and $L^*$ disagree, a counterexample is returned.
Otherwise, the partitioning is refined
Our Approach - Equivalence Queries

Check equivalence of partitioning-induced DFA and of $L^*$ DFA

Disagreements are checked by RNN for ground truth
If RNN and $L^*$ disagree, a counterexample is returned.
Otherwise, the partitioning is refined
Our Approach - Equivalence Queries

Check equivalence of partitioning-induced DFA and of $L^*$ DFA

Disagreements are checked by RNN for ground truth

If RNN and $L^*$ disagree, a **counterexample** is returned.

Otherwise, the partitioning is refined
Applications

• Concise, **exact models** from noisy, **partial data** (didn’t previously exist!)
• Test trained RNNs (**adversarial examples**)
• **Knowledge Extraction** (e.g., for NLP)
Results: concise, exact models in short time

```
def target(w):
    if len(w)==0:
        return True
    return w[0]==w[-1]

alphabet = "abcd"
```

Training

(4,400 samples to 100% accuracy)

RNN

Extraction

0.2 sec
Results: adversarial examples

• **GRU** trained to accuracy 100% on training set containing balanced parentheses up to depth 11, over alphabet (,a,b,c,...,z

```plaintext
)) (0.4s)
((i)ma (32.6s)
))) (1.1s)
(() (1.2s)
(((() (2.1s)
((((() (3.1s)
(((((() (3.8s)
(((((())) (9.2s)
((((((v())))) (10.7s)
((((((a()z)))))) (8.3s)
```
What do RNNs actually learn?

• Novel extraction algorithm using abstraction and exact learning to interpret behaviour of RNNs
  • Dynamic abstraction guided by interaction with exact learning algorithm, L*
  • First application of exact learning to a trained RNN

• Creates accurate and concise automata for trained RNNs

• Quickly discovers adversarial inputs for seemingly perfect RNNs (this happens frustratingly often...!)
What can RNNs actually learn?

**Proof requires infinite precision**
"push 0 into stack": \( g = g/4 + 1/4 \)
this allows pushing 15 zeros when using 32 bit floating point.

**Construction requires complex integration of carefully crafted components**

can this really be reached by gradient methods?

**Construction requires extra processing time at the end of the sequence**

we use "real time" RNNs in practice

---

**Classical result: RNNs are Turing complete**
(a) $a^n b^n$-LSTM on $a^{1000} b^{1000}$

(b) $a^n b^n c^n$-LSTM on $a^{100} b^{100} c^{100}$

(c) $a^n b^n$-GRU on $a^{1000} b^{1000}$

(d) $a^n b^n c^n$-GRU on $a^{100} b^{100} c^{100}$
LSTM equations

\[
\begin{align*}
  f_t &= \sigma(W^f x_t + U^f h_{t-1} + b^f) \\
  i_t &= \sigma(W^i x_t + U^i h_{t-1} + b^i) \\
  o_t &= \sigma(W^o x_t + U^o h_{t-1} + b^o) \\
  \tilde{c}_t &= \tanh(W^c x_t + U^c h_{t-1} + b^c) \\
  c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\
  h_t &= o_t \odot g(c_t)
\end{align*}
\]

Can make 1 via sigmoid

Exposes counter value
GRU Equations

\[
z_t = \sigma(W^z x_t + U^z h_{t-1} + b^z)
\]

\[
r_t = \sigma(W^r x_t + U^r h_{t-1} + b^r)
\]

\[
\tilde{h}_t = \tanh(W^h x_t + U^h (r_t \circ h_{t-1}) + b^h)
\]

\[
h_t = (z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t)
\]
In the paper [ACL’18]

• RNNs as simplified k-counter machines
• Results comparing different architectures
  • SRNN/IRNN
  • GRU/LSTM
• Proof that SRNN/GRUs cannot implement a binary counter
Summary

code2vec
A (somewhat) interpretable model for predicting program properties

What can RNNs learn?

Extraction from RNN to DFA

boolean Object (target) {
  for (Object elem: this.elements) {
    if (elem.equals(target)) {
      return true;
    }
  }
  return false;
}