DL4Code
Deep Learning for Programming, Compilers & Code Generation
Agenda

• Background & Context
• DL4Code - Modeling Code
• DL4Code - Compiler Optimizations
• DL4Code - Applications
• DL4Code - Datasets, Challenges & Research Directions
Background and Context
Signs of AI Progress

Driverless cars

Medical Diagnostics

Conversational Systems

Strategic Games
Natural Language to Programming Language

• Today we use DL methods on human language text
  • Natural Language Understanding
  • Natural Language Generation

• Can we use DL methods on Software code/programs/intermediate representation to build software engineering tools?
Naturalness of Software

• Programming languages, in theory, are complex, flexible and powerful
• But "natural" programs, the ones that real people actually write, are mostly simple and rather repetitive
• Programs have usefully predictable statistical properties
• these properties can be captured in statistical language models and leveraged for software engineering tasks
• Repetitiveness occurs in code corpora at *lexical, syntactic, and semantic* levels
Big Code + Naturalness of Software

- Very large publicly available corpora of open-source code
- Growth of Github repositories
- Big Code has “naturalness”
- Amenable to statistical Analysis
- Programmer Intent is built into code
- Use statistical/ML/DL techniques to infer the intent

**Big Code + Naturalness = DL4Code**
Developers implicitly embed **knowledge** in code that may be useful for the same or other projects.

**GitHub**  
internal & external codebases

**Mine** the hidden knowledge to create **smart** software engineering tools.
Some of the Big Code Use Cases

- Idiom mining
- API Mining
- Naming conventions learning
- Bug Localization
- Code Summarization
- Comment Generation
- Code completion
- Code search
- Code fixing
- Compiler Optimizations/heuristics learning
- Code Synthesis
What this workshop covers

• How do we model code using deep learning methods?

• How do we apply DL methods in compiler optimizations?

• How can we build smart software engineering tools using DL?
Modeling Program Representations

Ramakrishna Upadrasta, S. VenkataKeerthy
(along with Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Y. N. Srikant)

PAKDD 2021

May 12th, 2021
Outline

- Necessity of Program Encodings
- Characteristics of a Good Program Encoding
- Facets of Encodings
- Representational models of code: Case studies
- Summary

Program Representations
Program Representations for ML

Big Code: Vechev and Yahav (2016)
Program Representations for ML

Machine Learning based approaches shown to perform better

Similarity Detection, Summarization, Code Suggestion

Automated Tools

Undecidable

Compiler Optimizations

Inlining, Unrolling, Vectorization, Phase Ordering

NPH/NPC Heuristic driven

How to solve?

Big Code
### Program Representations for ML

NLP → word2vec, GloVe, ...
Images → CNN, Alexnet, ...
Programs → ?

#### Good Program Embeddings

<table>
<thead>
<tr>
<th>Independent</th>
<th>Capture</th>
<th>Diverse Applications</th>
<th>Scalable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Language</td>
<td>Syntax</td>
<td>Software Engineering</td>
<td>Faster Training Time</td>
</tr>
<tr>
<td>Target Architecture</td>
<td>Semantics</td>
<td>Compiler Optimizations</td>
<td>Lightweight ML models</td>
</tr>
<tr>
<td></td>
<td>The file size</td>
<td></td>
<td>No Out-Of-Vocabulary</td>
</tr>
</tbody>
</table>
Facets of Encodings

1. Representations
2. Embedding Techniques
3. Applications
Facets of Encodings: Representations

1. Representations

- Lexical Tokens
  - ~ NL Tokens; NLP Techniques
- Abstract Syntax Trees
  - Output by Parser of a language
- Intermediate Representations
- Program Dependence Graphs
Learning Effort  
(Amount of training data, time, ...) 

Semantic richness →  
Syntactic richness ← 

Lexical Tokens  
AST  
Intermediate Representation  
Program Analysis Info  
Recall? Phases of a compiler → 

← Syntactic richness 
Semantic richness → 

Inspired from Prof. Eran Yahav’s talks 

PAKDD 2021 Tutorial : DL4CODE
Intermediate Representation - LLVM IR

```
int sum(int a, int b) {
    return a + b;
}
int main() {
    int a = sum(1, 3);
    return a;
}
```

```
; ModuleID = 'file.c'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
; Function Attrs: nounwind uwtable
define i32 @sum(i32 %a, i32 %b) #0 {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    store i32 %a, i32* %1, align 4
    store i32 %b, i32* %2, align 4
    %3 = load i32, i32* %1, align 4
    %4 = load i32, i32* %2, align 4
    %5 = add nsw i32 %3, %4
    ret i32 %5
}
; Function Attrs: nounwind uwtable
define i32 @main() #0 {
    %entry = block {
...}
```
Some of the supported Languages

- Ada
- C
- C++
- Common Lisp
- Crystal
- CUDA
- DHE
- Delphi
- Fortran
- GO
- OpenGL Shading Language
- Halide

- Haskell
- Julia
- Lua
- Objective-C
- P4
- Python
- R
- Ruby
- Rust
- Scala
- Swift

(Many more Domain Specific Languages ...)

PAKDD 2021 Tutorial : DL4CODE
Facets of Encodings: Embedding Techniques/Learning representations

2. Embedding Techniques

- NLP based token embeddings
- Non-sequential models
- Sequential models
- Graph Neural Networks
- Attention models
Representation Learning

Branch of machine learning
Learns representations of data by automatically extracting the useful features
Widely used in Natural Language Processing

Analogy questions
Man : King :: Woman : ?

Clusters

Source: DL Series: NLP and Word Embeddings
Facets of Encodings: Downstream Applications

3. Applications

- Application Dependent Vs. Independent methods
- Software Engineering
  - Code summarization
  - Method name prediction
  - Program recognition/classification/similarity
- HPC/Compiler optimizations
  - Device mapping, Thread coarsening
  - Vectorization, Register allocation, Inlining, ...
- Code Generation
  - Test case generation
  - Code Translation
Representational Models

A high-level overview of the recent approaches
Representational models: codeNN (Iyer et al, ACL 2016)

- Input: Token based representation
- Architecture
  - LSTM based
  - Encoder-Decoder model
  - With Attention
- Combines text and code tokens
- Application dependent
  - Comment like summary
  - C# and SQL

Source: Iyer et al, ACL’16
Representational models: code2vec (Alon et al, POPL 2019)

- Input: AST paths
- Architecture
  - Non sequential model
    - FC
  - With Attention
    - Importance to paths
- Application dependent
  - Method names (Entire name)
  - Java
    - New parsers for extension

Source: code2vec: Learning Distributed Representations of Code - POPL’19
Representational models: code2vec (Alon et al, POPL 2019)
code2vec’s path extractor

```java
int fun(int t)
{
    int a = 90;
    return t + a;
}
```

- METHOD_NAME, (MethodDeclaration ↓ BlockStmt ↓ ExpressionStmt ↓ VariableDeclExpr ↓ VariableDecl), 90
- t, (BinaryOp:Plus), a
Representational models: code2vec (Alon et al, POPL 2019)

Experimental Analysis
Representational models: code2seq (Alon et al, ICLR 2019)

- Input: AST paths
- Architecture
  - sequential model
    - For decoder
  - With Attention
    - Importance to paths
- Application dependent
  - Java - Method names (Token by token)
  - C# - Code captioning

Source: code2seq ICLR’19
Representational models: code2seq (Alon et al, ICLR 2019)

Experimental applications

Source: code2vec: Learning Distributed Representations of Code - POPL’19
Representational models: NCC/inst2vec (Ben Nun et al, NeurIPS ‘19)

- Input: LLVM IR instructions
- Architecture: RNNs (skip gram model); supervised
- Application independent
  - Program classification
  - Device mapping, Thread coarsening

Source: NCC NeurIPS ‘19
Representational models: NCC/inst2vec (Ben Nun et al, NeurIPS ‘19)

double thres = 5.0;
if (x < thres)
  x = y * y;
else
  x = 2.0 * y;
x += 1.0;

(a) Source code
%cmp = fcmp olt double %x, 5.0
br i1 %cmp, label %LT, label %GE
LT:
  %2 = fmul double %y, %y
GE:
  %3 = fmul double 2.0, %y
AFTER:
  %4 = phi double [%2, %LT], [%3, %GE]
  %5 = fadd double %4, 1.0

(b) LLVM IR

(c) Dataflow basic blocks

(d) Contextual Flow Graph

Source: NCC NeurIPS’19
Input: LLVM IR entities

Architecture: Translational model (TransE), Unsupervised, Program analysis based

Application independent
  ○ Program classification
  ○ Device mapping, Thread coarsening
Representational models: IR2Vec (VenkataKeerthy et al, TACO’20)
Initial Representation

```
{ store, "TypeOf", IntegerTy }
{ store, "NextInst", store }
{ store, "Arg1", VAR }
{ store, "Arg2", PTR }

{ ret, "TypeOf", IntegerTy }
{ ret, "Arg1", VAR }
```

```
%a.addr = alloca i32, align 4
%b.addr = alloca i32, align 4
store i32 %a, i32* %a.addr, align 4
store i32 %b, i32* %b.addr, align 4
%0 = load i32, i32* %b.addr, align 4
%1 = load i32, i32* %a.addr, align 4
%add = add nsw i32 %0, %1
ret i32 %add
```
Representational models: IR2Vec (VenkataKeerthy et al, TACO’20)
Initial Representation

```
%a.addr = alloca i32, align 4
%b.addr = alloca i32, align 4
store i32 %a, i32* %a.addr, align 4
store i32 %b, i32* %b.addr, align 4
%0 = load i32, i32* %a.addr, align 4
%1 = load i32, i32* %b.addr, align 4
%add = add nsw i32 %0, %1
ret i32 %add
```
Representational models: IR2Vec (VenkataKeerthy et al, TACO’20)

Initial Representation

```
(\text{store}, "\text{TypeOf}", \text{IntegerTy})
(\text{store}, "\text{NextInst}", \text{store})
(\text{store}, "\text{Arg}_1", \text{VAR})
(\text{store}, "\text{Arg}_2", \text{PTR})
...

(\text{ret}, "\text{TypeOf}", \text{IntegerTy})
(\text{ret}, "\text{Arg}_1", \text{VAR})
```

```
%a.\text{addr} = \text{alloca i32}, \text{align 4}
%b.\text{addr} = \text{alloca i32}, \text{align 4}
\text{store i32 }%a, \text{i32* }%a.\text{addr}, \text{align 4}
\text{store i32 }%b, \text{i32* }%b.\text{addr}, \text{align 4}
%0 = \text{load i32, i32* }%a.\text{addr}, \text{align 4}
%1 = \text{load i32, i32* }%b.\text{addr}, \text{align 4}
\text{add = add nsw i32 }%0, %1
\text{ret i32 }\text{add}
```
\[
\begin{align*}
\text{store } & \text{ "TypeOf", IntegerTy } \\
\text{store } & \text{ "NextInst", store } \\
\text{store } & \text{ "Arg\_1", VAR } \\
\text{store } & \text{ "Arg\_2", PTR } \\
\text{return } & \text{ "TypeOf", IntegerTy } \\
\text{return } & \text{ "Arg\_1", VAR }
\end{align*}
\]

\[
\begin{align*}
\text{\%a.addr} & = \text{alloca i32, align 4} \\
\text{\%b.addr} & = \text{alloca i32, align 4} \\
\text{store i32} & \text{\%a, i32* \%a.addr, align 4} \\
\text{store i32} & \text{\%b, i32* \%b.addr, align 4} \\
\text{\%0} & = \text{load i32, i32* \%a.addr, align 4} \\
\text{\%1} & = \text{load i32, i32* \%b.addr, align 4} \\
\text{\%add} & = \text{add nsw i32 \%0, \%1} \\
\text{ret i32} & \text{\%add}
\end{align*}
\]
\[
\begin{align*}
\text{TransE} &\quad \rightarrow \quad h, r, t \\
\text{Seed Embedding Vocabulary} &\quad \rightarrow \quad \text{IntegerTy}, \text{ret}, \text{store}, \text{VAR}, \text{PTR} \\
\end{align*}
\]
Symbolic Encodings

PAKDD 2021 Tutorial: DL4CODE

\[ W_o(\text{Seed}) + W_t(\text{Seed}) + W_a(\text{Seed}) \]

\[ \langle \text{store, "TypeOf", IntegerTy} \rangle \]
\[ \langle \text{store, "NextInst", store} \rangle \]
\[ \langle \text{store, "Arg1", VAR} \rangle \]
\[ \langle \text{store, "Arg2", PTR} \rangle \]
\[ \langle \text{ret, "TypeOf", IntegerTy} \rangle \]
\[ \langle \text{ret, "Arg1", VAR} \rangle \]
IR2Vec Representation

Seed Embedding Generation → Vector Generation → Downstream Tasks

- Seed Embedding Vocabulary
  - n-d vectors of h, r, t
- Triplets
- LLVM IR
- Source Program
- C, C++, Fortran...
- Use-Def + Reaching Def
- Live Inst.
- Call Inst.
- Instruction Vector
- BasicBlock Vector
- Function Vector
- Program Vector
- PAKDD 2021 Tutorial : DL4CODE
Flow-Aware: Intra Basic block Instruction Vector

Instructions

11: %arr.addr = alloca i32
12: %size.addr = alloca i32
13: %i = alloca i32
14: store i32 *%arr, i32 *%arr.addr
15: store i32 %size, i32 *%size.addr
16: store i32 0, i32 *%i
17: br label %for.cond

Embeddings

\[ W_o([\text{alloca}]) + W_t([\text{PointerTy}]) \Rightarrow [11] \]
\[ W_o([\text{alloca}]) + W_t([\text{IntegerTy}]) \Rightarrow [12] \]
\[ W_o([\text{alloca}]) + W_t([\text{IntegerTy}]) \Rightarrow [13] \]
\[ W_o([\text{store}]) + W_t([\text{PointerTy}]) + W_o([\text{PTR} ]+ [11]) \Rightarrow [14] \quad \text{KILL and UPDATE [11]} \]
\[ W_o([\text{store}]) + W_t([\text{IntegerTy}]) + W_o([\text{VAR} ]+ [12]) \Rightarrow [15] \quad \text{KILL and UPDATE [12]} \]
\[ W_o([\text{store}]) + W_t([\text{IntegerTy}]) + W_o([\text{CONST} ]+ [13]) \Rightarrow [16] \quad \text{KILL and UPDATE [13]} \]
\[ W_o([\text{br}]) + W_t([\text{VoidTy}]) + W_t([\text{LABEL}]) \Rightarrow [17] \]
Inter BasicBlock Data Flow Info Propagation

I1: %i = alloca i32
I2: %j = alloca i32

I3: store i32 5, i32* %j

I4: %t0 = load i32, i32* %j
I5: %t1 = load i32, i32* %i

I6: %x = add nsw i32 %t1, 2
I7: store i32 %x, i32* %t1
I8: %y = add nsw i32, %t0, 3
I9: store i32 %y, i32* %t0

\[ [I_2] = W_0([\text{alloca}]) + W_2([\text{IntegerTy}]) \]
Inter BasicBlock Data Flow Info Propagation

I1: %i = alloca i32
I2: %j = alloca i32

I3: store i32 5, i32* %j

I4: %t0 = load i32, i32* %j
I5: %t1 = load i32, i32* %i

I6: %x = add nsw i32 %t1, 2
I7: store i32 %x, i32* %t1
I8: %y = add nsw i32, %t0, 3
I9: store i32 %y, i32* %t0

\[
\begin{align*}
[I_2] &= W_0([\text{alloca}]) + W_f([\text{IntegerTy}]) \\
[I_4] &= W_0([\text{load}]) + W_f([\text{PointerTy}]) + W_0([I_2] + [I_3] + [I_9])
\end{align*}
\]
Inter BasicBlock Data Flow Info Propagation

\[ I_2 \] = \[ W_0 \cdot \text{alloca} \] + \[ W_t \cdot \text{IntegerTy} \] \\
\[ I_4 \] = \[ W_0 \cdot \text{load} \] + \[ W_t \cdot \text{PointerTy} \] + \[ W_a \cdot (\text{alloca} + \text{load}) \] \\
\[ I_5 \] = \[ W_0 \cdot \text{load} \] + \[ W_t \cdot \text{PointerTy} \] + \[ W_a \cdot (\text{alloca} + \text{load}) \]
Code Vector

How vectors are composed!

Find a function to effectively compose Instruction vectors
Code Vector

How vectors are composed!

- If $\text{LIVE}[BB_i]$ and $\text{KILL}[BB_i]$ live and kill instruction set of the basic block $BB_i$,
  \[
  \text{LIVE}[BB_i] = \bigcup I - \text{KILL}[BB_i]
  \]
  \[
  [[BB_i]] = \sum \text{LIVE}[BB_i]
  \]

- For a function $F$ with the basic blocks, $BB_0, BB_1, \ldots, BB_b$
  \[
  [[F]] = \sum_{i=1}^{b} [[BB_i]]
  \]

- For a program $P$ with functions, $F_1, F_2, \ldots, F_n$
  \[
  [[P]] = \sum_{i=1}^{n} [[F_i]]
  \]
Experimentation: Data Collection

- Eliminate unwanted information
  - Collect information only within a Basic block

- Data collection is done on entire SPEC CPU 2017 and Boost libraries
  - ~13k files
  - Converted C/CPP files to LLVM IR
  - Each IR file ranging from 100s to 20k lines of code

- An LLVM pass is run to extract data, convert it to required format
Instruction clusters

![Graphs showing instruction clusters](Image)
Instruction Analogies

<table>
<thead>
<tr>
<th>Syntactic</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>zext : integerTy :: fpext : floatTy</td>
<td>sdiv : lshr :: mul : shl</td>
</tr>
<tr>
<td>trunc : fp trunc :: icmp : fcmp</td>
<td>fptoui : uitofp :: fptosi : sitofp</td>
</tr>
<tr>
<td>load : pointer :: store : constant/variable</td>
<td>sdiv : srem :: udiv : urem</td>
</tr>
<tr>
<td>fdiv : sdiv :: fmul : mul</td>
<td>shl : lshr :: or : and</td>
</tr>
<tr>
<td>call : ret :: switch : label</td>
<td>shl : ashr :: mul : div</td>
</tr>
</tbody>
</table>

- Queries: \( a : b :: c : ? \)
- \( ? = b - a + c \)
- Closest neighbour
  - Using Euclidean distance
Experimentation

- Detecting optimal device mapping in a heterogeneous system
  - CPU Vs. GPU
  - Gradient boosting
- Predicting optimal thread coarsening factor
  - Number of threads to fuse/coarsen
  - Gradient boosting
- Program Classification
Some of the models that we didn’t cover …

- Flow2Vec: Value-Flow-Based Precise Code Embedding
  - Sui et al, OOPSLA 2020

- Learning Semantic Program Embeddings with Graph Interval Neural Network
  - Wang et al, OOPSLA 2020

- ProGraML: Graph-based Deep Learning for Program Optimization and Analysis
  - Cummins et al, Preprint 2020

- Compiler-based graph representations for deep learning models of code
  - Brauckmann et al, CC 2020
Comparison of various models: OOV Perspective
Comparison of various models: Representations Perspective

Learning Effort
(Amount of training data, time, ...)

Program Analysis
Effort/Complexity

Semantic richness →  Syntactic richness

Lexical Tokens
AST
Intermediate Representation
Program Analysis Info

Program Analysis
Effort/Complexity

Phases of a compiler ↗

Somewhere here: NCC, IR2Vec

deepTune, CodeNN, ...
code2vec, code2seq, ...

PAKDD 2021 Tutorial : DL4CODE
## Comparison of various models: Summary

<table>
<thead>
<tr>
<th>Model</th>
<th>Type of Embeddings</th>
<th>Vocabulary Size</th>
<th>Training Time</th>
<th>Application</th>
<th>Specific</th>
</tr>
</thead>
<tbody>
<tr>
<td>code2vec</td>
<td>AST Paths</td>
<td>1.5 M Token Embeddings</td>
<td>Vocab: N/A Task: 1.5 Days</td>
<td>Application Specific</td>
<td></td>
</tr>
<tr>
<td>NCC</td>
<td>IR Instructions</td>
<td>8,565 Stmt Embeddings</td>
<td>Vocab: 8 Days Task: 10 Hours</td>
<td>Application Independent</td>
<td></td>
</tr>
<tr>
<td>DeepTune</td>
<td>Hybrid approach</td>
<td>128 Tokens &amp; Characters</td>
<td>Vocab: N/A Task: 10 Hours</td>
<td>Application Specific</td>
<td></td>
</tr>
<tr>
<td>IR2Vec</td>
<td>Entities</td>
<td>64 Entity Embeddings</td>
<td>Vocab: 20 Minutes Task: 5 Seconds</td>
<td>Application Independent</td>
<td></td>
</tr>
</tbody>
</table>
DL-based Compiler Optimizations
Agenda

• Brief Background
• DeepTune- End-to-end Deep Learning of Optimization Heuristics
• Ithemal : Basic Block Throughput Prediction
• Graph-Coloring based Register Allocation via LSTMs
• Compiler Auto-Vectorization using Imitation Learning
• Conclusion and Way Forward
Brief Background

Compiler Optimization Pass

Program → IR → IR → Optimized Machine Code

Analysis → Transformation → Optimized program in target language → Code Generation
• Lots of compiler optimization depend on carefully constructed heuristics
  • Which optimization order to apply
  • How to inline functions, which functions should be inlined
  • Scheduling and register allocation
  • How much to unroll loops

• Heuristics are developed and maintained by compiler engineers
  • Not always scalable
  • Difficult to adapt to different hardware
  • Fragile in many cases
  • The space of possible optimizations is also vast, making it very hard for compiler writers to design heuristics that take all these considerations into account
    • As a result, many compiler optimizations are out of date or poorly tuned
Brief Background (Cont’d)

• Given a program, compiler writers would like to know what compiler heuristic or optimization to apply in order to make the code better
  • Execute faster
  • Smaller code footprint
  • Reduced power

• Machine learning can be used to build a model used within the compiler, that makes such decisions for any given program
  • Early work concentrated on feature engineering (ex: Loop Unrolling)
  • More recent work applies DL, RL techniques

* Figure taken from: Machine Learning in Compiler Optimisation, Zheng Wang and Michael O’Boyle
DeepTune- End-to-end Deep Learning of Optimization Heuristics
DeepTune

- Used for optimal mapping of heterogeneous parallelism
  - Should we run an OpenCL kernel on a CPU or a GPU?
- End2End DL of optimization heuristics
  - One of the earliest works to test this strategy
DeepTune Components

- Language Model
- Source Rewriter
- Sequence Encoder

```c
__kernel void memset_kernel(__global char *mem_d,  
  short val, int num_bytes) {
        const int thread_id = get_global_id(0);  
    mem_d[thread_id] = val;
}

__kernel void A(__global char *a, short b, int c) {
        const int d = get_global_id(0);
    a[d] = b;
}
```

<table>
<thead>
<tr>
<th>idx</th>
<th>token</th>
<th>idx</th>
<th>token</th>
<th>idx</th>
<th>token</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>_kernel</code></td>
<td>10</td>
<td><code>'\n</code></td>
<td>19</td>
<td>`'const'</td>
</tr>
<tr>
<td>2</td>
<td><code>'</code></td>
<td>11</td>
<td><code>'short'</code></td>
<td>20</td>
<td><code>'d'</code></td>
</tr>
<tr>
<td>3</td>
<td><code>'void'</code></td>
<td>12</td>
<td><code>'b'</code></td>
<td>21</td>
<td><code>'='</code></td>
</tr>
<tr>
<td>4</td>
<td><code>'A'</code></td>
<td>13</td>
<td><code>'int'</code></td>
<td>22</td>
<td><code>'get_global_id'</code></td>
</tr>
<tr>
<td>5</td>
<td><code>(</code></td>
<td>14</td>
<td><code>'c'</code></td>
<td>23</td>
<td><code>'0'</code></td>
</tr>
<tr>
<td>6</td>
<td><code>'__global'</code></td>
<td>15</td>
<td><code>')'</code></td>
<td>24</td>
<td><code>'.'</code></td>
</tr>
<tr>
<td>7</td>
<td><code>'char'</code></td>
<td>16</td>
<td>`'</td>
<td>'`</td>
<td>25</td>
</tr>
<tr>
<td>8</td>
<td><code>'\n'</code></td>
<td>17</td>
<td><code>'\n'</code></td>
<td>26</td>
<td><code>'.'</code></td>
</tr>
<tr>
<td>9</td>
<td><code>'a'</code></td>
<td>18</td>
<td><code>'\n'</code></td>
<td>27</td>
<td><code>'.'</code></td>
</tr>
</tbody>
</table>

PAKDD 2021 Tutorial : DL4CODE 63
DeepTune Components

- Embedding
  - Converts sparse to dense representations
- LSTM
- Heuristic Model

```python
def init(self, seed: int):
    from keras.layers import Input, Embedding, LSTM, Dense
    from keras.layers.merge import Concatenate
    from keras.layers.normalization import BatchNormalization
    from keras.models import Model
    np.random.seed(seed)

    # Language model. Takes as inputs source code sequences.
    code_in = Input(shape=(1024,), dtype="int32", name="code_in")
    x = Embedding(input_dim=atomizer.vocab_size + 1, input_length=1024, output_dim=64, name="embedding")(code_in)
    x = LSTM(64, implementation=1, return_sequences=True, name="lstm_1")(x)
    x = LSTM(64, implementation=1, name="lstm_2")(x)
    langmodel_out = Dense(2, activation="sigmoid")(x)

    # Auxiliary inputs. wgsize and dsize.
    auxiliary_inputs = Input(shape=(2,))
    x = Concatenate()([auxiliary_inputs, x])
    x = Dense(32, activation="relu")(x)
    out = Dense(2, activation="sigmoid")(x)
    self.model = Model(inputs=[auxiliary_inputs, code_in], outputs=[out, langmodel_out])
    self.model.compile(optimizer="adam", metrics=['accuracy'], loss=["categorical_crossentropy", "categorical_crossentropy"], loss_weights=[1., .2])

    return self
```

https://github.com/ChrisCummins/paper-end2end-dl/blob/master/code/Case%20Study%20A.ipynb
Visualizing DeepTune

* Figure taken from: End-to-end Deep Learning of Optimization Heuristics. Chris Cummins, Pavlos Petoumenos, Zheng Wang, Hugh Leather
• The optimal static mapping achieves 58% accuracy
• Grewe et al. (hand-crafted features + ML) achieves 73% accuracy
• DeepTune predictive models achieve accuracy of 84%

* Figure taken from: End-to-end Deep Learning of Optimization Heuristics. Chris Cummins, Pavlos Petoumenos, Zheng Wang, Hugh Leather
Performance

- DeepTune Token-based IR encoding vs IR2Vec
Ithemal : Basic Block Throughput Prediction
Estimating Basic Block Throughput

- The problem:
  - *Given a basic-block of x86 instructions estimate the throughput in terms of clock cycles*
  - Analytical models (llvm-mca, IACA) are usually used in such scenarios

Diagram:

- Code 1:
  - `lea r14, [rbx-0x40]`
  - `lea rdx, [rbp+0x48]`
  - `cmp rdi, rax`

- Code 2:
  - `lea r14, [rbx-0x40]`
  - `lea rdx, [rbp+0x48]`
  - `cmp rdi, rax`
  - `sub rbp, 0x60`

- Code n:
  - `lea r14, [rbx-0x40]`
  - `lea rdx, [rbp+0x48]`
  - `cmp rdi, rax`

- High-Level Code
- Optimizing Compiler

Cycle Times:

- 40 cycles
- 44 cycles
- 36 cycles
Accurate Modeling of Processor Core is Complex

• Modeling the micro-architectural details of a complex core is a very hard problem
  • Very easy to omit details
  • Specs are not always accurate
  • Some details are proprietary
Ithemal: A Data-driven approach

- Run/train a DL model using many samples of x86 BBs and their corresponding cycle count
  - Measured on real hardware
  - A new hardware just means re-training the new model OR some form of transfer learning
- High accuracy and ease of portability

*Figure taken from: Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks, C. Mendis et al.*
Ithemal DL model

- Hierarchical LSTMs
  - 2-layers
  - Layer-1 for the sequence of operands of each instruction
  - Layer-2 for the sequence of instructions

*Figure taken from: Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks, C. Mendis et al.*
Ithemal Comparative Performance

- Ithemal delivers better throughput accuracy compared to analytical models
- Portable and robust
- Github resource: https://github.com/ithemal/Ithemal/tree/master/learning/pytorch

*Figure taken from: Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks, C. Mendis et al.*
DL-based Graph Coloring Register Allocation
Graph Coloring

- Graph-coloring is an important problem in CS with numerous applications
- A graph coloring is an assignment of labels, called colors, to the vertices of a graph such that no two adjacent vertices share the same color
- The chromatic number $X(G)$ of a graph $G$ is the minimal number of colors for which such an assignment is possible
- Solved using heuristics

A graph coloring for a graph with 6 vertices. It is impossible to color the graph with 2 colors, so the graph has chromatic number 3.
Register Allocation as a Graph-Coloring Problem

- Register Allocation is an important problem in the area of compiler code generation.
- Usually the number of registers available < number of variables used.
- Create what is known as the *interference graph* which models registers which need to be *live* at the same time.

Example:

```
live-in:  k  j
      g := mem[j+12]
      h := k - 1
      f := g * h
      e := mem[j+8]
      m := mem[j+16]
      b := mem[f]
      c := e + 8
      d := c
      k := m + 4
      j := b

live-out:  d  k  j
```

Interference Graph of Live Ranges
Modeling Graph Coloring using LSTMs

- Viewed as a sequence-2-sequence translation via LSTMs
- An input sequence where each item of the sequence corresponds to a node of the graph
- The output sequence is of the same length as the input sequence (number of nodes of the graph)
- Trained using random graphs
Inference and Color-Correction phase

- Difficult to encode constraints in LSTM that two adjacent nodes cannot have some color
- *Invalid* edges may appear during inference
- Rectify these edges using a post-inference color-correct pass

- Forest-Fire graph
  - has a chromatic number of 5
- LSTM-based model colors with 4 colors resulting in 2 invalid edges
  - These edges are \( <v_2, v_3> \) and \( <v_1, v_5> \)
  - \( v_5 \) can reuse the color \( c_3 \) as none of its neighboring nodes use \( c_3 \).
  - \( v_2 \) requires a new color \( c_5 \) as both \( v_2 \) and \( v_3 \)'s neighbors use all the colors
- So finally we also get 5 colors 😊
DL-model vs LLVM’s Greedy Register Allocator (GRA)

- Collected the interference graphs for the functions of certain SPEC CPU® 2017 benchmarks
  - Use these graphs to predict colors using the DL-model
- Collect the actual register count of each function after codegen from LLVM
- Comparison shows DL-model performing better than GRA

https://github.com/dasdibye/DL4RegAlloc
Compiler Auto-vectorization using Imitation Learning
Auto-Vectorization

The problem:
- Solve the SLP (Superword-Level Parallelism) using ML
- SLP is a superset of Loop Vectorization whereby straight-line code can also be vectorized
- Naïve ML strategies may lead to correctness issues as only isomorphic statements can be vectorized

Scalar Code


Vector Code

Auto-Vectorization

• Naïve ML strategies may lead to correctness issues as only isomorphic statements can be vectorized

Scalar Code

Neural Machine Translation

Vector Code


MDP Formulation

- The pairwise instruction packing problem formulated as a MDP by iteratively forming one vector pack at a time following a particular instruction traversal policy – Bottom-Up or Top-Down
MDP Formulation

– State details

• Nodes: 5 types of nodes to encode the graph features of a state:
  • Instruction Node: correspond to each instruction with at least one valid packing opportunity or instructions which are already packed
  • Pack Node: common node representing overhead packing instructions
  • Unpack Node: common node representing overhead unpacking instructions
  • Constant Node: common node representing any constant value used by instructions
  • Focus Node: connected to the instruction that is considered for packing

• Edges: Following are the 4 types of edges connecting the above nodes:
  • Dependency Edge: encodes if an instruction must be executed after another
  • Possible Pack Edge: encodes whether two instructions can be packed together
  • Packed Edge: encodes instructions that are already packed together
  • Focus Edge: the focus edge connects the focus node to the instruction node that is considered for packing

* Figure taken from: Compiler 2.0: How to Modernize Compiler Technology, Saman Amarasinghe et al.
Imitation Learning

• goSLP solves the SLP problem *exactly* using ILP solvers
• Use such a solution to imitate/mimic the action space
• Use actual runtimes/estimates as cost/reward function
• Gated Graph Neural Network (GGNN) used as part of the policy network modeling to make packing decisions for each state

Use Imitation Learning
The goal is to check learnability

State New State
{a[1],a[2]}, {a[2],a[3]}, {a[3],a[4]}

Choose a “valid” action
{a[1],a[2]}, {a[2],a[3]}, {a[3],a[4]}
{a[1],a[2]}
Some other work we did not cover

- **Static Neural Compiler Optimization via Deep Reinforcement Learning** – R. Mammdil et al. LLVM-HPC Workshop (2020) co-located with Supercomputing 2020
  - Solving the phase-ordering problem for the LLVM compiler pass-pipeline
  - Predict the most beneficial VFs(Vectorization Factor) for Loop Vectorization
  - Replacing the heuristics-based inlining-for-size optimization in LLVM with machine learned models
- **CompilerGym - Facebook AI.** [https://github.com/facebookresearch/CompilerGym](https://github.com/facebookresearch/CompilerGym)
  - CompilerGym is a toolkit for exposing compiler optimization problems for reinforcement learning. It allows machine learning researchers to experiment with program optimization techniques without requiring any experience in compilers, and provides a framework for compiler developers to expose new optimization problems for AI
DL-based Compiler Optimizations
– Wrapping it up
Conclusion and Future Directions

- We are just at the cusp of utilizing DL techniques for compiler analysis/transform
- A big challenge is to ensure correctness of the transformed code
- Some of the proposed work consists of a mix of NLP-like techniques and RL
- No consensus yet on standardized program representations though techniques along the lines of word2vec seem to be gaining traction
- No consensus yet on representing control and data-flow though Graph Neural Networks (GNNs) are being used in some cases
Conclusion and Future Directions

• Some authors like Cummins & Leather suggest OaaS
  • Optimization as a service
  • All optimizations implemented as some kind of an RL-based model

• Personal opinion:
  • RL too costly to be introduced for all optimization passes
  • Compile time may blow up and we may need sophisticated HW resources
  • Optimizations of the future may be a mix of NLP-like and RL models

• Open questions:
  • Can we generate *semantically correct* transformed code using a DL model?
  • Would the optimizations of the future be a mix of DL models + Fast correction algorithms?
    • Get most of it right using a DL model and then apply a quick and simple correction pass
  • Can these models be really portable – across compilers and across hardware?

* Figure taken from: Machine Learning in Compilers: Past, Present and Future, Chris Cummins and Hugh Leather
The Future?

Learnt Compiler Optimization Pass

Program → IR (Learnt Analysis Model) → IR (Learnt Transformation Model) → Optimized Machine Code

Optimized program in target language → IR

Code Generation
DL4Code Applications
DL4Code – DL for Smart Software Tools

- DeepBugs - Finding Bugs Automatically
- DeepFix - Fixing Bugs Automatically
- Deep Code Search
- Code Clone Detection
- Code Summarization
- Comment Generation
Learning to find bugs

• Use deep learning to distinguish correct code and buggy code

• How do we create training data for this problem?
• How should the code be represented?
What is wrong with this code?

```javascript
function setPoint(x, y) {
  ...
}

var x_dim = 23;

var y_dim = 5;

setPoint(y_dim, x_dim);
```
Name related bugs

What’s wrong with that code?

```java
for (j = 0; j < params; j++) {
    if (params[j] == paramVal) {
        ...
        Should be params.length
    }
}
```
DeepBugs

1. Code corpus → Generate training data
   - Correct code → Represent code as vectors
   - Buggy code → Represent code as vectors
2. Correct vectors → Train classifier
3. Buggy vectors → Train classifier
   - Classifier → Predict bugs in new code
4. New code → Predict bugs in new code
   - Bugs
How to generate training data?

• Use simple code transformations to generate artificial bugs in code

• Argument swapping
  • SetPoint (X, Y) $\rightarrow$ SetPoint(Y, X)

• Wrong Binary Operator
  • Replace an existing operator with a randomly selected operator
  • for (; i <= length;...) $\rightarrow$ for (; i % length, i++)

• Wrong Operand
  • Replace an existing operand with a randomly selected operand
    • Value << 2 $\rightarrow$ value << next
How does DeepBugs represent code?

- Concatenate word embeddings of tokens to generate code vectors
  - Intuition – natural language identifiers in code carry semantics
- For each argument, add type and formal parameter name
- For each operand, add name, type, AST parent and grand parent node
Evaluation of DeepBugs

• 68 millions of Javascript code
• Gets close to 68% precision
• But bugs of only the 3 types below identified

<table>
<thead>
<tr>
<th>Bug detector</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Training</td>
</tr>
<tr>
<td>Swapped arguments</td>
<td>1,450,932</td>
</tr>
<tr>
<td>Wrong binary operator</td>
<td>4,901,356</td>
</tr>
<tr>
<td>Wrong binary operand</td>
<td>4,899,206</td>
</tr>
</tbody>
</table>
Fixing Bugs Automatically

Erroneous program → Error localization → Ranked list of potential faulty statements → 
Mutation → Candidate fixes → Validation → Test suite → Fixed program

Fixing Bugs Automatically
Common Programming Errors

• Often arise due to programmer oversight
• Surface level problems rather than semantic/program logic issues
• Can be due to inexperience with a specific programming language
• Include
  • missing code block delimiter
  • use of incompatible operators
  • missing variable declarations
• Compiler errors are not often intuitive
• Can DL tools learn about such issues and repair them automatically?
DeepFix – probabilistic program repair

• Models $P(\text{fixed code/input program})$ as an attention based seq-seq network
Challenges

• Unlike machine translation or summarization, predicting a fix requires all the sequence tokens to be exactly correct.
• Requires handling long term dependencies correctly.
• Since we need to generate correct executable code, we need to get identifier names exactly correct.
• Verifying the correctness of the generated fix.
DeepFix block diagram

Input program

\[(l_1, s_1), (l_2, s_2), \ldots, (l_k, s_k)\]

Seq2Seq attention network

Fix

\[(l_i, s'_i)\]

Oracle

Updated program

\[(l_1, s_1), \ldots, (l_i, s'_i), \ldots, (l_k, s_k)\]

Accept

Reject or stop

iterative repair strategy
DeepFix Evaluation

• Student written C programs
• Program length up to 400 tokens,
• Training data generation
  • seed errors into correct submissions to generate training examples
• Testing on two types of data
  • *Raw dataset*: real incorrect submissions
  • *Seeded dataset*: seed errors into correct submissions
  • *For the raw dataset, 32% fixed messages*
  • *For seeded dataset, 63% fixed messages*
Deep Code Search

• Programming is hard 😊
• Often developers search for code online for specific issues

• Code search engines such as Github search
• General purpose search engines not optimal for code search
• Code search engines have limited capabilities for semantic search
  • Typically key word/phrase matched search
• Standard techniques are based on information retrieval
IR based Code Search

- Representation Mismatch between NL query and Search object in PL
- Query – How to read an object in XML

```java
public static <S> S deserialize(Class c, File xml) {
    try {
        JAXBContext context = JAXBContext.newInstance(c);
        Unmarshaller unmarshaller = context.createUnmarshaller();
        S deserialized = (S) unmarshaller.unmarshal(xml);
        return deserialized;
    } catch (JAXBException ex) {
        log.error("Error-deserializing-object-from-XML", ex);
        return null;
    }
}
```
Unifying code and natural language query

Joint Embedding of both Code and Natural Language into a unified vector representation

“read a text file line by line”

“read an object from an xml file”
Deep Learning based Code Search

Offline Training

Training Instances

aspect extraction

code snippets (Java methods)

Commented Code Snippets

natural language descriptions

Query

CODEnn Model

embedding

0101010

Query Vector

Recommended Code

Similarity Lookup

embedding

010110
110011
101000
0001

Code Vectors

Search Codebase

Offline Embedding

PAKDD 2021 Tutorial : DL4CODE
Code-Description Embedding Neural Network (CODEnn)

- Code Embedding Network (CoNN)
- Description Embedding Network (DeNN)
- Similarity Module
Deep Code Search Model
Code Clone detection

• Code clones are one or more code fragments that are similar
• Degree of similarity defines different types of clones
  • T1 – code fragments are similar except for white space, layout & comments
  • T2 – syntactically identical except for identifier names and literals
  • T3 – syntactically similar that can differ at statement level. The fragments have statements added, modified and/or removed
  • T4 - Syntactically dissimilar code fragments that implement the same functionality

• T1 easiest to detect, T4 is the hardest clone type to detect
Code Clone Detection

• Compare the fragments at text level – string comparison
• Lexical comparison by comparing the fragments at token level
• Tree based – measure AST tree similarity
• Graph – Compare semantic representation of the code fragments
• Type 1 and Type 2 clones can be covered by lexical/string comparison
• Type 3 – typically tree based comparison
• Type 4 needs semantic level comparison
DL based Clone Detection

• Earlier approaches used either identifier information or structural (tree) information
• ML approaches used hand-crafted features which do not generalize across code bases
• End to end DL approaches can automatically learn from characteristics common across code bases to detect clones
• Considerable work on text similarity/paraphrasing can be leveraged for this problem
Deep Learning Code Fragments for Code Clone Detection
Other Applications - DeepAPI

Use DL to identify API usage sequence

User need - Parse XML Files
- `DocumentBuilderFactory.newInstance()`
- `DocumentBuilderFactory.newDocumentBuilder()`
- `DocumentBuilder.parse()`

Earlier approaches used IR approaches
- From User query to API sequence

Treats it as a seq-to-seq problem

Represents user NL query as a context vector using an encoder

Generate API sequence as output using a decoder
Other Applications

• **Learning naming conventions/coding conventions**
  • suggest natural identifier names and formatting conventions

• **Comment Generation**
  • Model comment generation as a machine translation problem
  • Handle out of vocabulary tokens

• **Automatically generating commit messages from diffs**
  • Using neural machine translation

• **Intelligent Code Review**

• **Code Summarization**
  • **Learning to generate pseudo code from code**
  • **Convolutional attention network for code summarization**
Datasets, Challenges & Research Directions
Datasets for DL4Code

• Large unlabelled repositories from github
  • Linux kernel
  • LLVM applications test suite, SPEC benchmark
  • Java applications on github
  • Typically used for building language models of code
  • Only unsupervised learning can be applied
  • No common benchmarking across these datasets

• Datasets for DL4Code Applications
  • Smaller task specific datasets often are used
  • BigCloneBench – for Code clone detection
  • Defects4j – Java based, buggy and correct code from Java projects
  • Bugswarm: mining and continuously growing a dataset of reproducible failures and fixes
DL4Code – Challenges

• Collecting labeled data requires massive annotation work.
• Different tasks will require different labels
• Often expertise needed in annotation unlike standard DL tasks
• Need for proper evaluation metric that can incorporate both semantic meaning, grammatical and execution correctness
• DL models for code are complex
  • Tasks which humans find simpler (naming conventions) are difficult for machines
  • Handling out of vocabulary tokens
DL4Code – Research Directions

• Improving data efficiency
  • Use of pretrained models
  • Multi-task learning for DL4Code tasks
  • Use of semi-supervised, unsupervised methods

• Discrete and symbolic representations

• External knowledge Injection

• Real world application & evaluation

• Ensuring correctness
References


References


Thanks
Backup slides