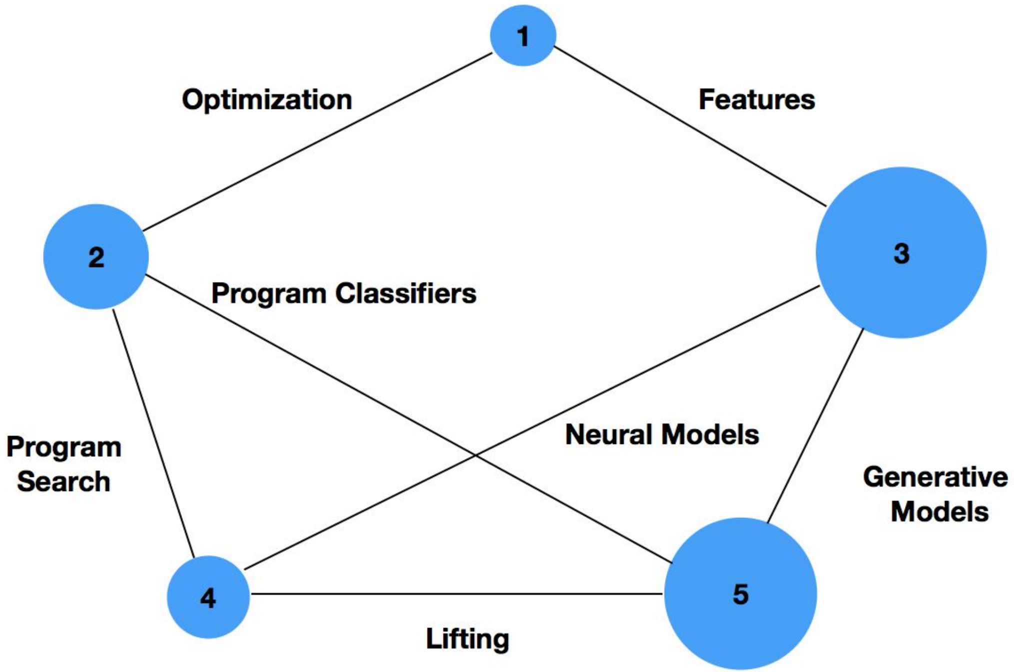
Rethinking Compilation:L3


1
Optimization
Features
2
3
Program Classifiers
Neural Models
Program
Search
Generative
Models
4
5
Lifting

Overview

- L1: Motivation and survey of auto-tuning/machine learning for compilers
- L2: Program rewriting schemes - e-graphs and equality saturation
- This lecture: Program embeddings and Graph Neural Networks
- L4: Program synthesis and neural synthesis
- L5: Neural Machine Translation,Transformers and Large language models

Motivation

Machine Learning on Code

Loop Unrolling

```
int i, j;
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    y[i] += A[i][j]*x[j];
```

→

```
int i, j;
for (i = 0; i < N; i+=4)
  for (j = 0; j < N; j++)
    y[i] += A[i][j]*x[j];
    y[i+1] += A[i+1][j]*x[j];
    y[i+2] += A[i+2][j]*x[j];
    y[i+3] += A[i+3][j]*x[j];
```

Algorithm Detection

```
int i, j, k;
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      C[i][k] += A[i][j]*B[j][k]
```

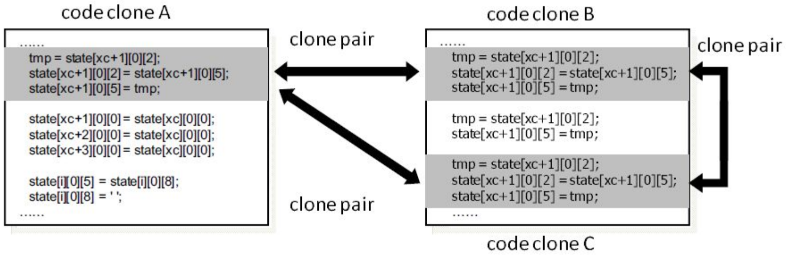
→

```
gemm(A, B, C)
```

Vulnerability Detection

```
i = read(STDIN_FILENO, msg, sizeof(msg)-1);
memcpy( username, msg+2, i-2);
```

Code Clone Detection

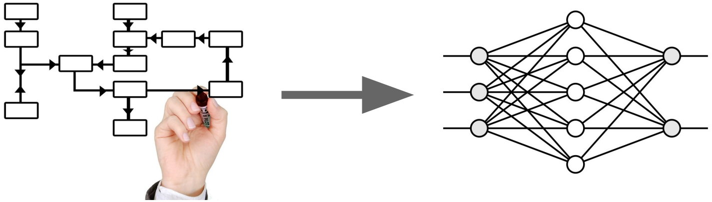


Complex and undecidable problems → Machine Learning techniques

Motivation

Beyond classic machine learning techniques

Automating feature engineering

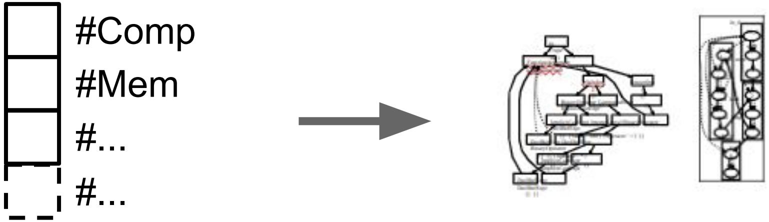


More sophisticated models for harder tasks

Source Code

Transformed Code

Dealing with programs represented as complex datastructures



→ Need for better ML techniques! (this lecture)

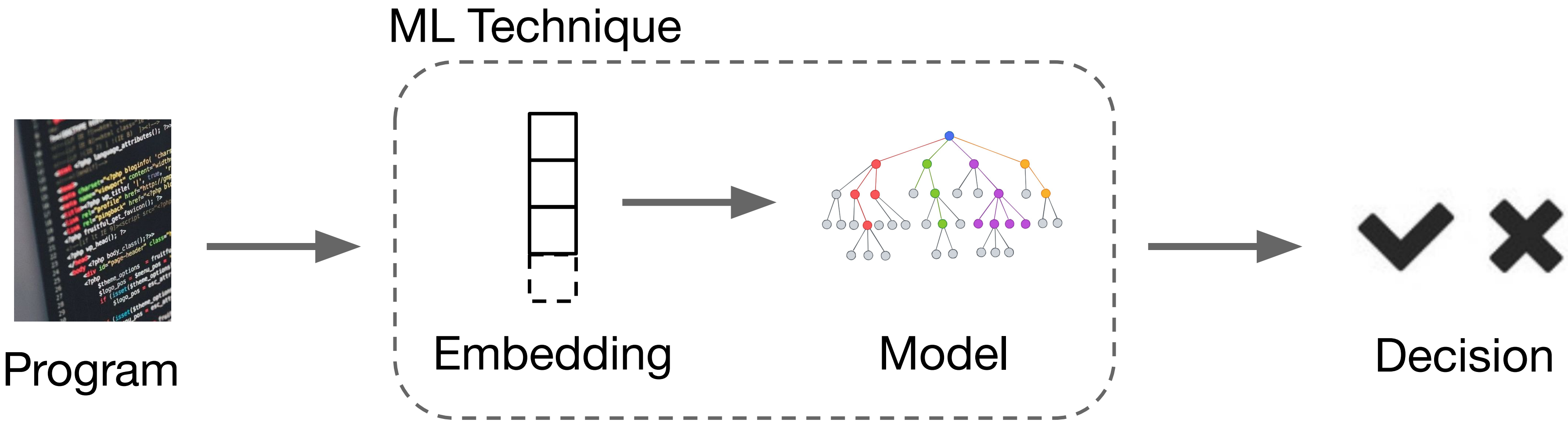
Course content

- Motivation
- Embedding Techniques
  - Feature Engineering
  - Learned Embeddings

- Code Modeling
  - Sequences
  - Graphs
  - Sequences II – Transformers

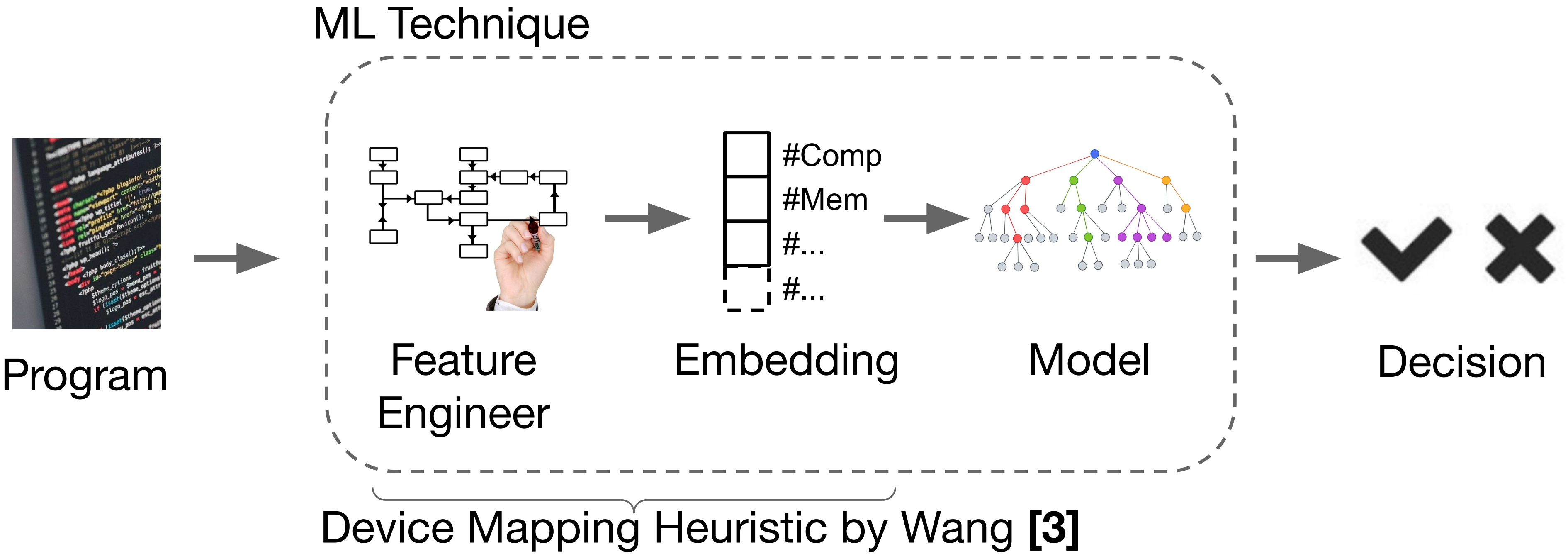
  - Combinations of Sequences & Graphs
  - Interpreters

Overview



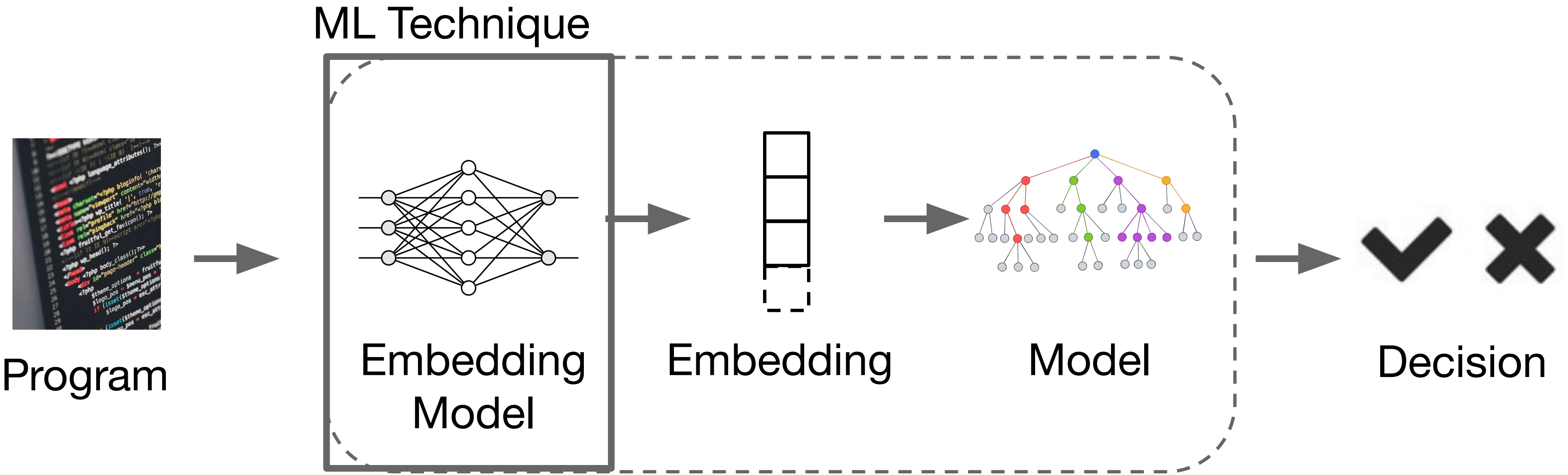
Embedding Techniques

Feature Engineering



Problem: Features not optimal and engineering time-consuming

Learned Embeddings

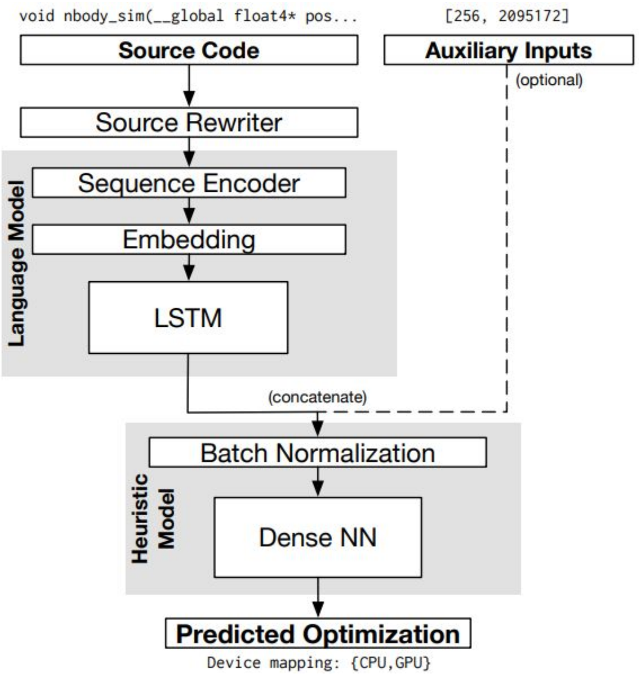


Use models to learn embeddings automatically!

Learned Embeddings

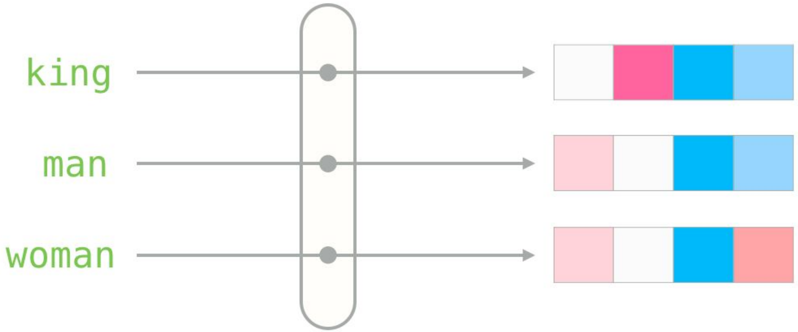
- Pre-compute feature vectors ("Embeddings")

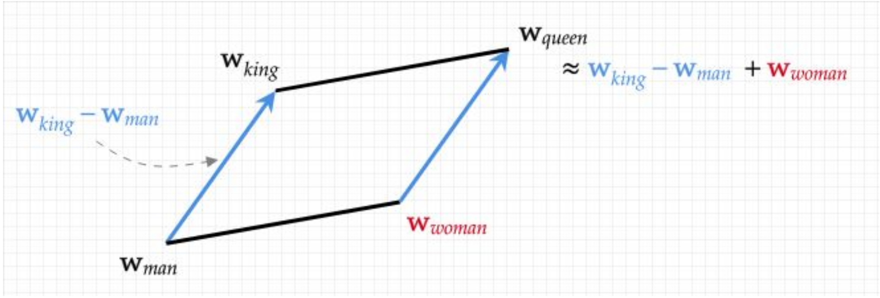
- DeepTune **[4]**
  - Training on task A with large dataset →Embedding
  - Re-using Language Model of task A for task B, then train on small-scale dataset



Embedding Techniques

Learned Embeddings



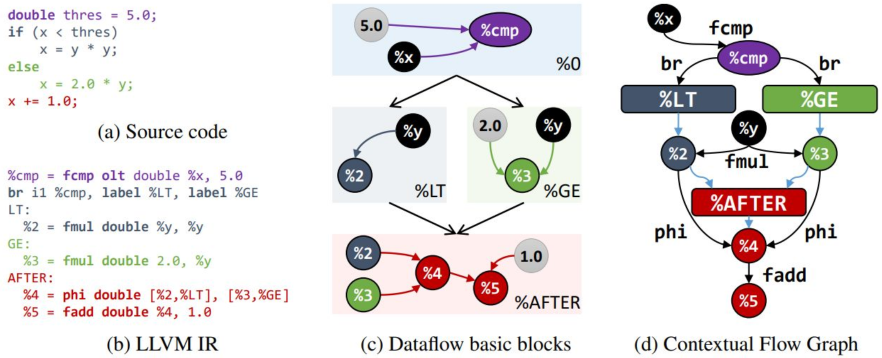


- Word2vec **[1]**
  - Self-supervised training with contextual similarity objective: Tokens with similar context should have similar embedding
  - Multi-layer Perceptron (MLP) model predicts context
  - MLP later used for vector lookup

Learned Embeddings

- Word2vec on Code: Inst2vec **[2]**
  - Pre-train word2vec style embeddings on LLVM IR graphs with contextual similarity objective
  - Predict types of neighbouring nodes in graph
  - Trained on 50 mio. lines of code

  - Significant performance gain vs. no pre-trained embeddings



(a) Source code
(b) LLVM IR
(c) Dataflow basic blocks
(d) Contextual Flow Graph

Course content

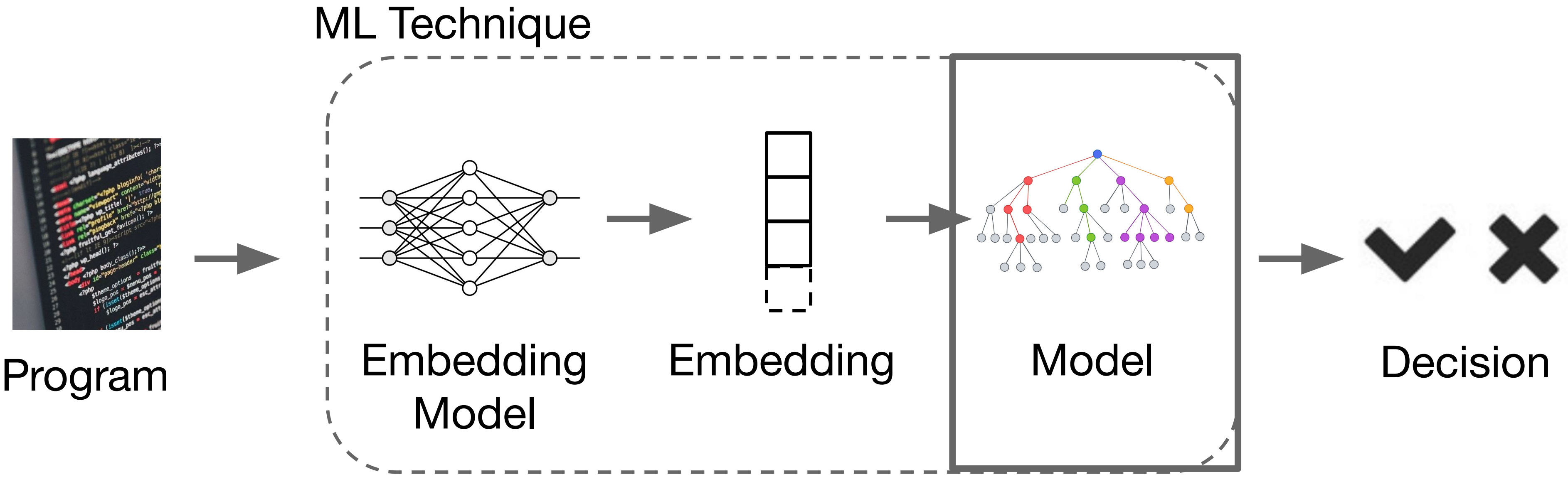
- Motivation
- Embedding Techniques
  - Feature Engineering
  - Learned Embeddings

- Code Modeling
  - Sequences
  - Graphs
  - Sequences II – Transformers

  - Combinations of Sequences & Graphs
  - Interpreters

Code Modeling

Overview



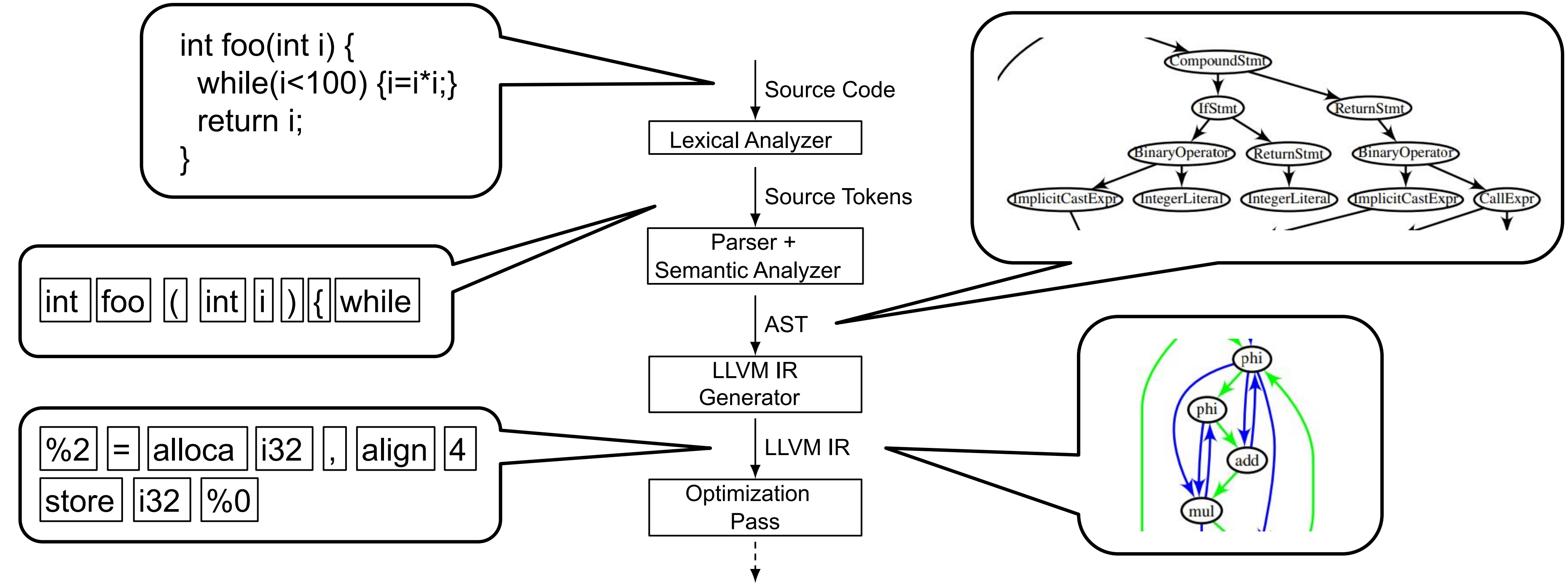
After extracting embeddings, learn models of code

Code Modeling

How to represent programs?

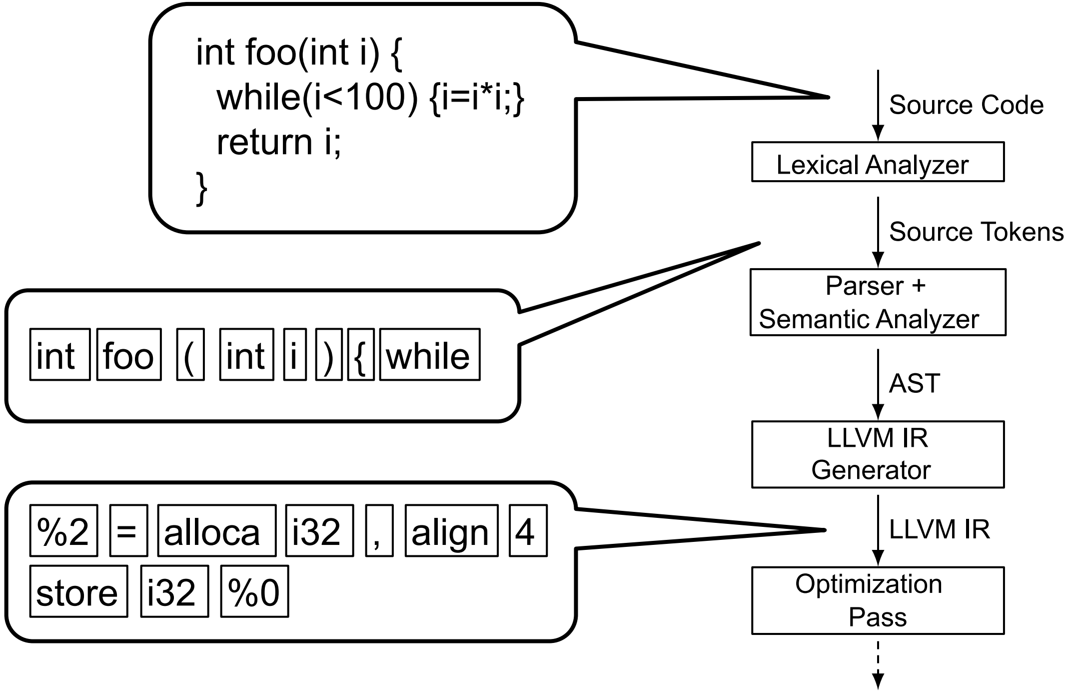
Sequences

Graphs



Code Modeling

Sequence Program Representations

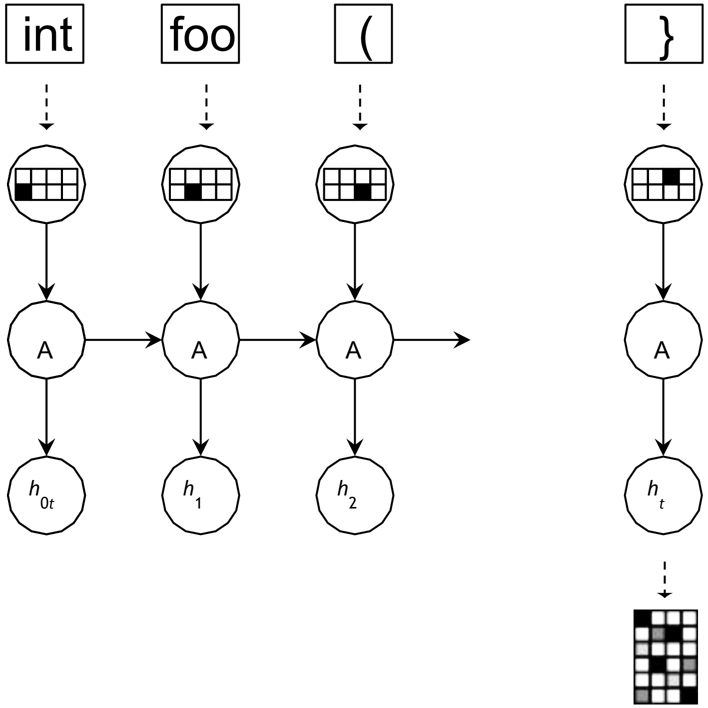


- Program as token sequence
- Different abstraction levels
  - Characters
  - Source language
  - Compiler internal representations (IRs)
- Normalization of identifiers helps generalization

Code Modeling

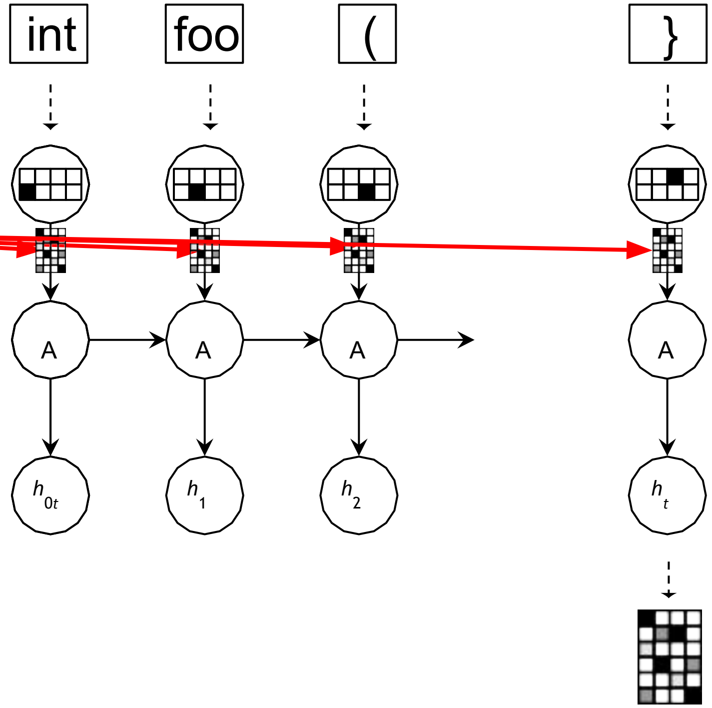
Sequence Models – DeepTune [4]

- Input: Sequence of C tokens
- One-hot encoding of tokens
- Recurrent Neural Network
  - Processes tokens one-by-one
  - Captures sequential dependencies
- Output: Hidden states
- Final hidden state used for prediction



Sequence Models – inst2vec [2]

- Input: Sequence of C tokens
- One-hot encoding of tokens
- Lookup of embedding
- Recurrent Neural Network
  - Processes tokens one-by-one
  - Captures sequential dependencies
- Output: Hidden states
- Final hidden state used for prediction



Code Modeling

Sequence Models – inst2vec **[2]**

Table 4: Heterogeneous device mapping results

Architecture	Prediction Accuracy [%]				
	GPU	Grewe et al. [29]	DeepTune [18]	inst2vec	inst2vec-imm
AMD Tahiti 7970	41.18	73.38	83.68	82.79	**88.09**
NVIDIA GTX 970	56.91	72.94	80.29	82.06	**86.62**
	Speedup				
	GPU	Grewe et al.	DeepTune	inst2vec	inst2vec-imm
AMD Tahiti 7970	3.26	2.91	3.34	3.42	**3.47**
NVIDIA GTX 970	1.00	1.26	1.41	1.42	**1.44**

Table 5: Speedups achieved by coarsening threads

Computing Platform	Magni et al. [46]	DeepTune [18]	DeepTune-TL [18]	inst2vec	inst2vec-imm
AMD Radeon HD 5900	1.21	1.10	1.17	**1.37**	1.28
AMD Tahiti 7970	1.01	1.05	**1.23**	1.10	1.18
NVIDIA GTX 480	0.86	1.10	**1.14**	1.07	1.11
NVIDIA Tesla K20c	0.94	0.99	0.93	**1.06**	1.00

Course content

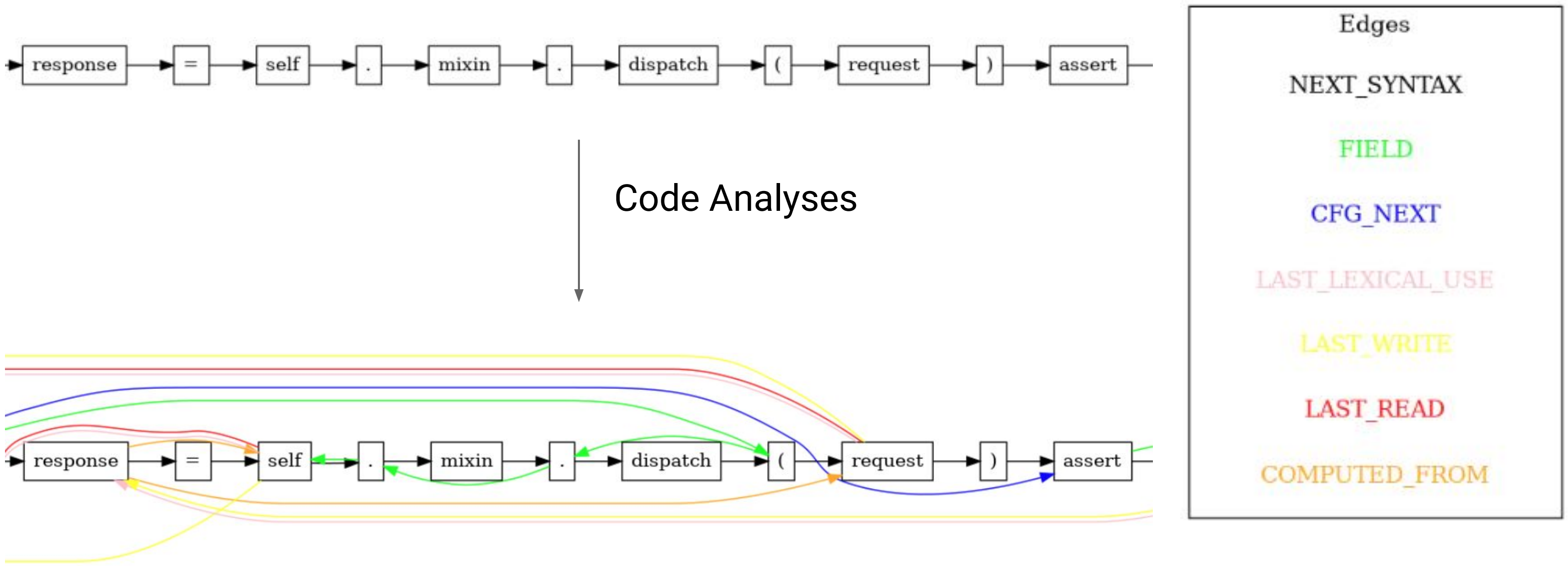
- Motivation
- Embedding Techniques
  - Feature Engineering
  - Learned Embeddings

- Code Modeling
  - Sequences
  - Graphs
  - Sequences II – Transformers

  - Combinations of Sequences & Graphs
  - Interpreters

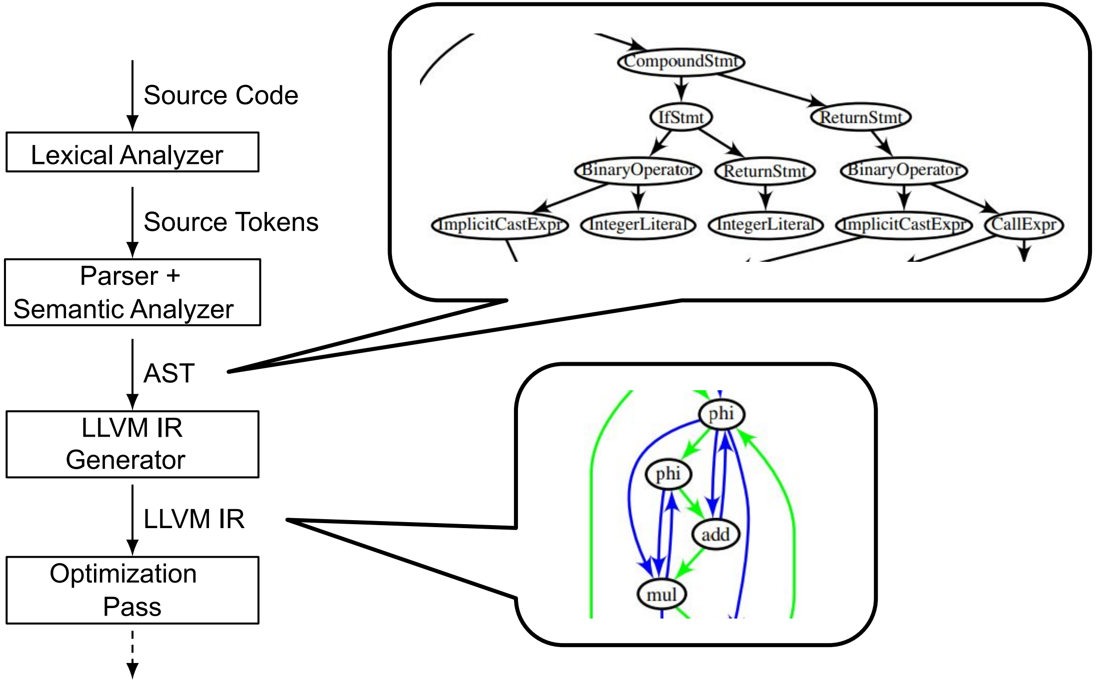
Graph Models

- Idea: Learn a model on known code structures



Graph Program Representation

- Represent programs as graphs
- Compiler-internal information represented as edges
    - Control-flow
    - Use-def
- Normalized by construction

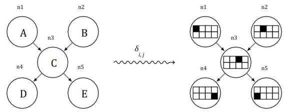


Graph Models

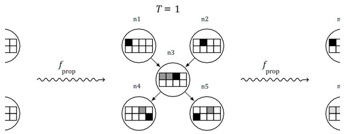
- Graph Neural Networks
  - Input: Nodes, Edges
  - Output: Graph embedding h_G

- Propagation Style
  - Graph-based
- Modeling Capability
  - Relations

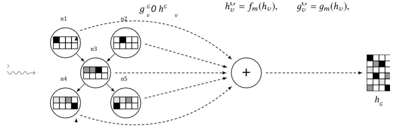
Phase 0: Initialization



Phase 1: Message Passing

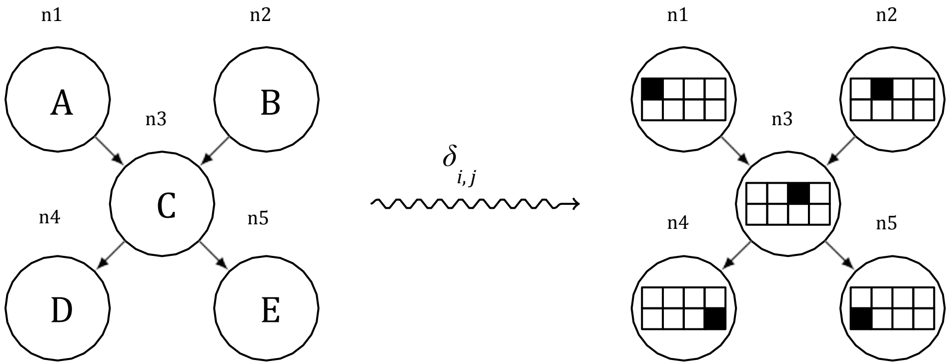


Phase 2: Embedding Aggregation



Graph Models

Phase 1: Initialization



- Initialize each node with hidden state
  - One-hot encoded
  - Produced by learned function

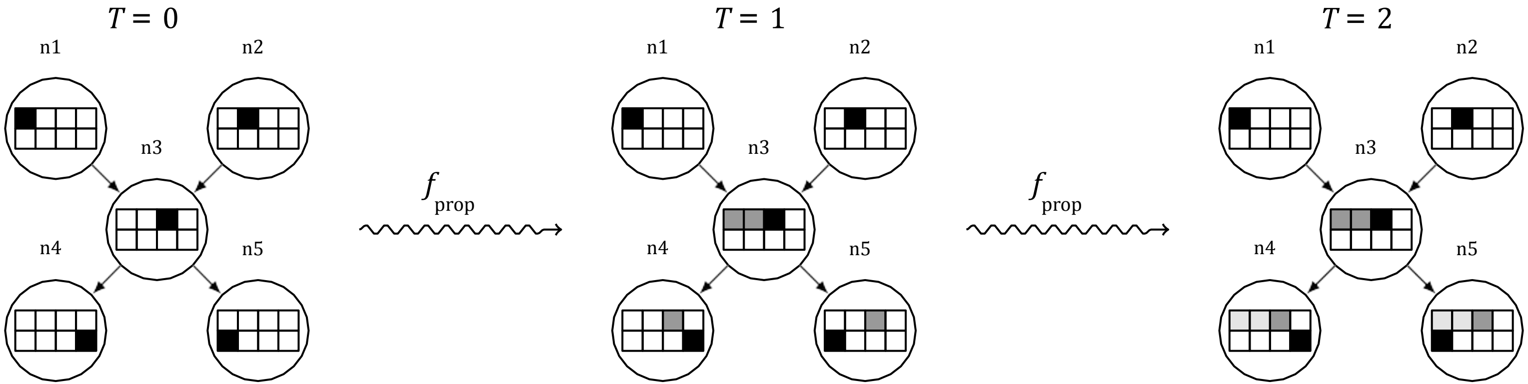
Code Modeling

Graph Models

Phase 2: Message Passing

$$f_{\text{msg}}(h_v, e_t) = A_{e_t} \cdot h_v + b_{e_t},$$

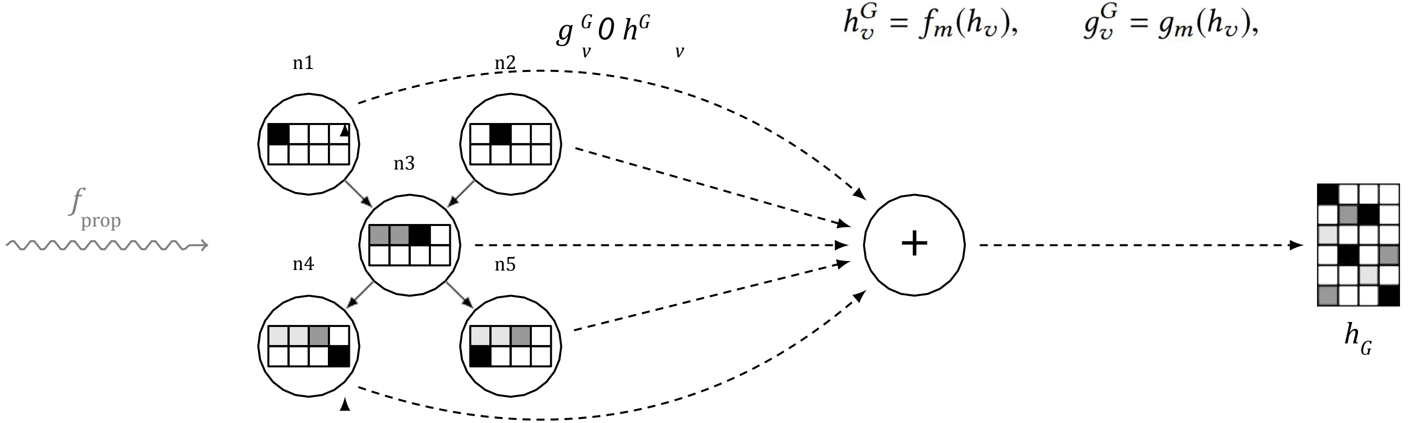
$$a_v = \sum_{u:(u,v)\in E} f_{\text{msg}}(h_v, e_t), \qquad h'_v = f_{\text{prop}}(a_v, h_v)\ \forall v \in V$$



- f_msg forms messages, based on node state
- f_prop computes new node state, based on aggregated messages
- f_msg and f_prop are learned functions

Graph Models

Phase 3: Embedding Aggregation

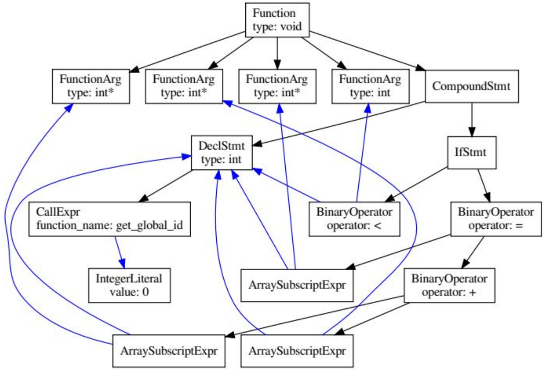


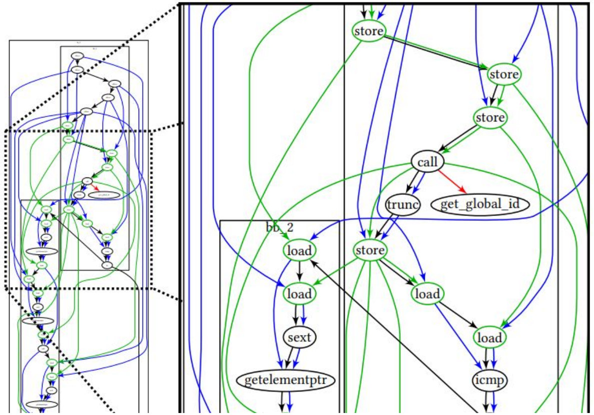
$$h_v^G = f_m(h_v), \qquad g_v^G = g_m(h_v),$$

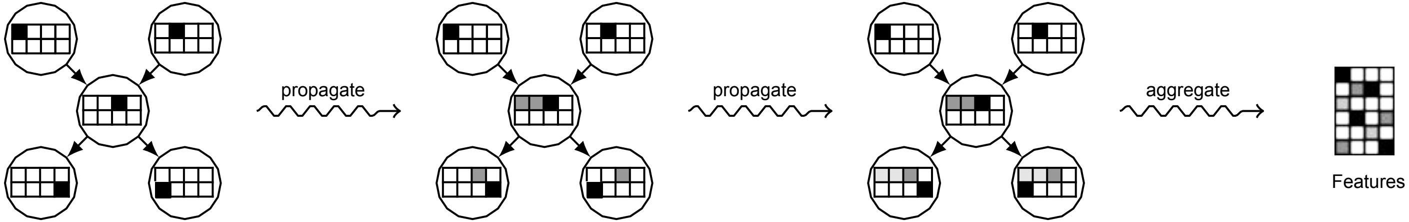
- Aggregate node embeddings to a graph embedding
- f_m and g_m are learned functions

Graphs – GNNs4Compilers [5]

- Program Representation
  - Clang AST + Use-Def
  - LLVM IR Graphs
- Code Model
  - GNN







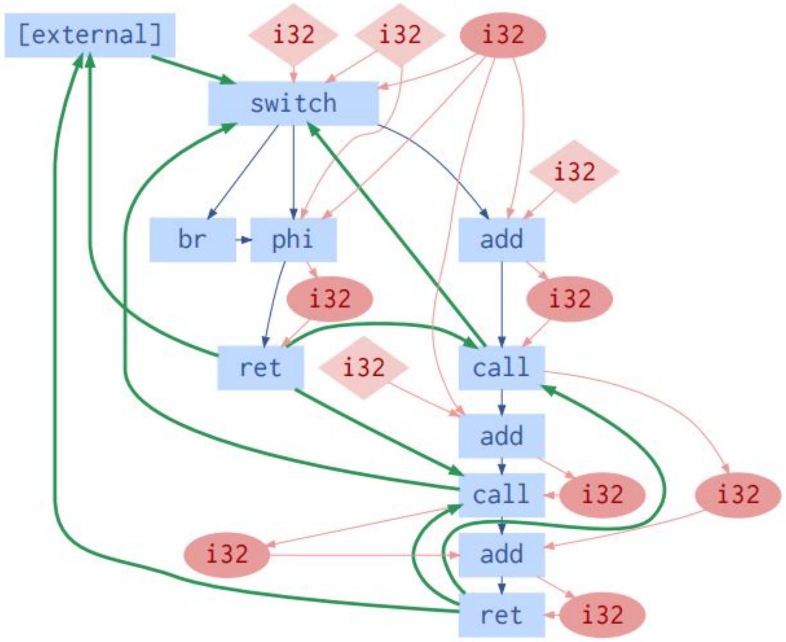
Code Modeling

Graphs – ProGraML [6]

- Program Representation
  - LLVM IR Graphs

- Embedding
  - Pre-trained inst2vec embeddings

- Code Model
  - GNN



Code Modeling

Graphs – ProGraML [6]

	Accuracy	Precision	Recall	F_1
Static Mapping	58.8%	0.35	0.59	0.44
DeepTune [23]	71.9%	0.72	0.72	0.72
$\text{DeepTune}_{\text{IR}}$	73.8%	0.76	0.74	0.75
NCC [7]	80.3%	0.81	0.80	0.80
ProGraML	**86.6%**	**0.89**	**0.87**	**0.88**

(a) AMD

	Accuracy	Precision	Recall	F_1
Static Mapping	56.9%	0.32	0.57	0.41
DeepTune [23]	61.0%	0.69	0.61	0.65
$\text{DeepTune}_{\text{IR}}$	68.4%	0.70	0.68	0.69
NCC [7]	78.5%	0.79	0.79	0.79
ProGraML	**80.0%**	**0.81**	**0.80**	**0.80**

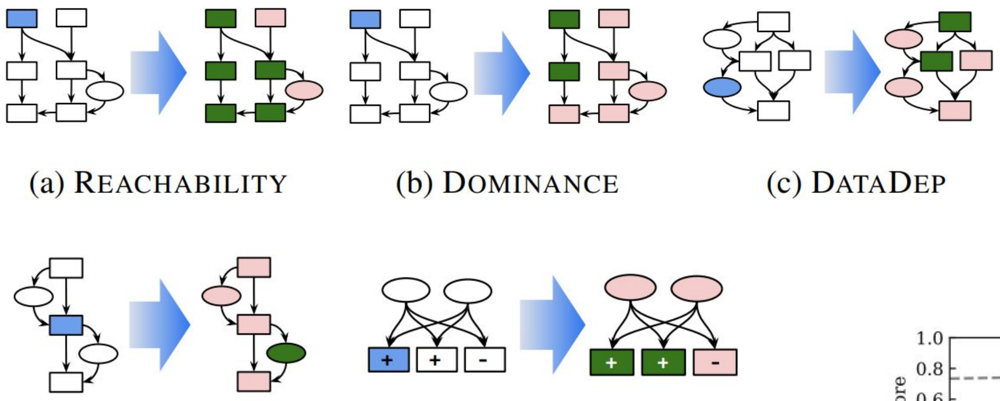
(b) NVIDIA

Table 5: Five approaches to predicting heterogeneous device mapping: (a) Static Mapping (b) DeepTune [23], a sequential model using tokenized OpenCL, (c) $\text{DeepTune}_{\text{IR}}$, the same model adapted for tokenized LLVM-IR, (d) NCC, which uses pre-trained statement embeddings, and (e) PROGRAML, our approach.

GNNs significantly outperform RNNs (+ inst2vec embeddings)

GNNs yield best performance

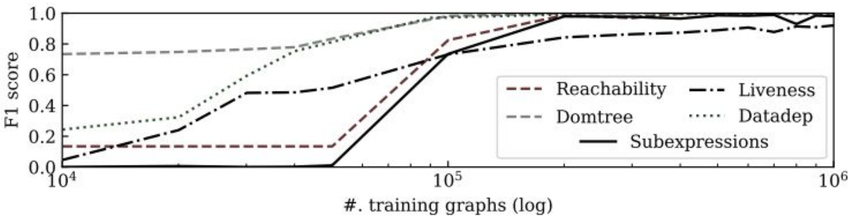
Graphs – Compiler Analyses [6]



(a) REACHABILITY (b) DOMINANCE (c) DATADEP

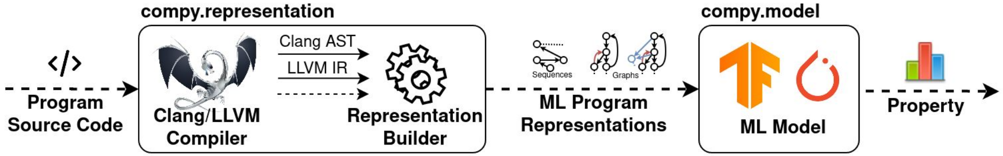
(d) LIVENESS (e) SUBEXPRESSIONS

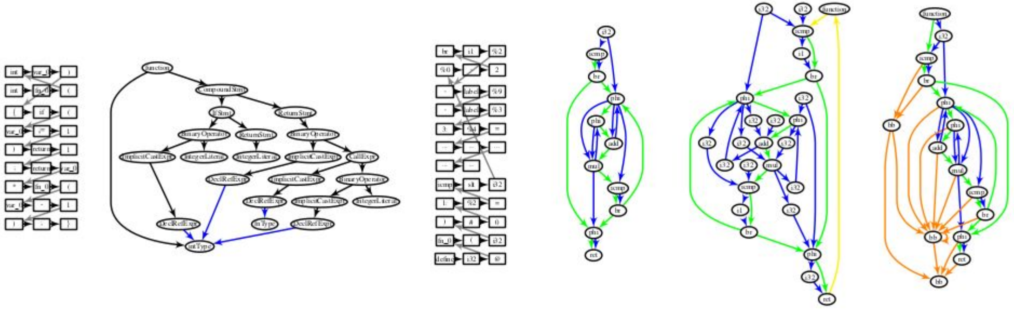
- GNNs can learn classic compiler analyses
- RNNs perform poorly, as they don't represent structure explicitly



Graphs – ComPy-Learn [7]

- Designing own, task-specific code representations, based on Clang/LLVM
- Auto-extracting these from C/C++ code, using just Python
- Learning models of code (RNNs, GNNs) on graph structures
- https://github.com/tud-ccc/compy-learn





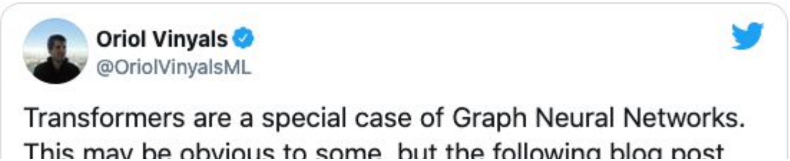
Course content

- Motivation
- Embedding Techniques
  - Feature Engineering
  - Learned Embeddings

- Code Modeling
  - Sequences
  - Graphs
  - Sequences II – Transformers

  - Combinations of Sequences & Graphs
  - Interpreters

Sequences II – Transformers



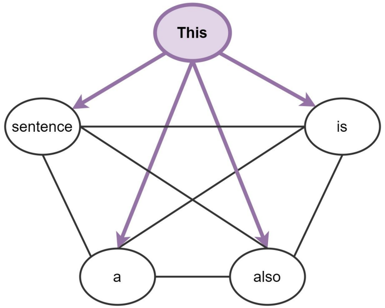
Transformers are GNNs

- on fully-connected graphs
- with learned attention weights

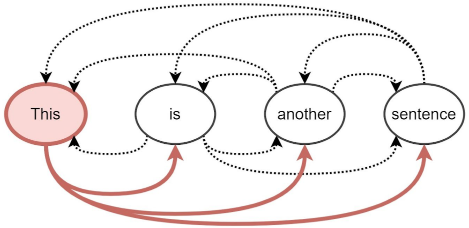
Intuition

- Transformers operate on graphs with "soft edges"

Graph Neural Networks



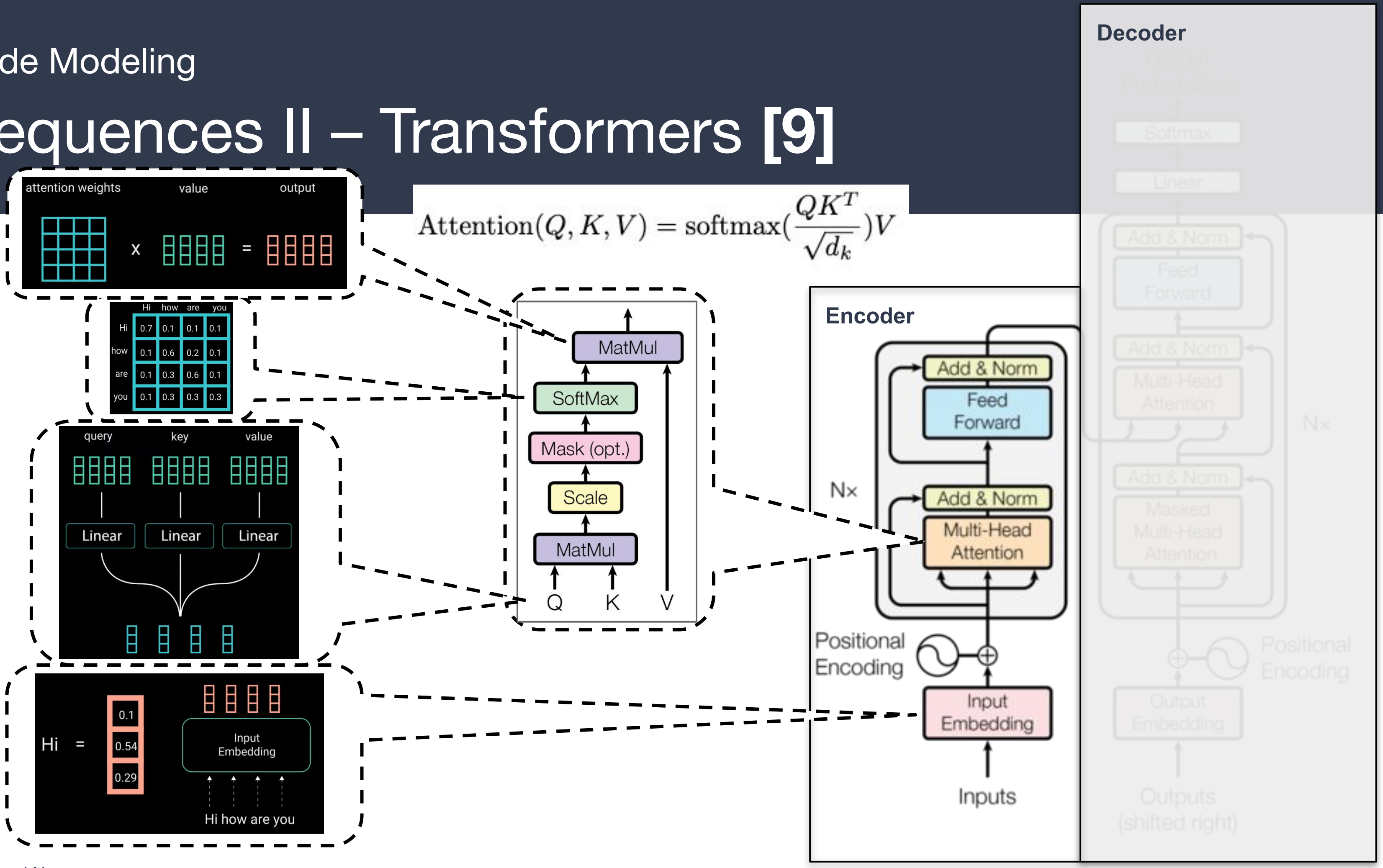
Transformers



Code Modeling

Sequences II – Transformers [9]

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$



Figures: Michael Nguyen

Course content

- Motivation
- Embedding Techniques
  - Feature Engineering
  - Learned Embeddings

- Code Modeling
  - Sequences
  - Graphs
  - Sequences II – Transformers

  - Sequences & Graphs
  - Interpreters

RNNs, GNNs and Transformers

	RNNs	GNNs	Transformers
Propagation Style	Sequential	Graph-based	Graph/Attention-based
Modeling capabilities	Sequential Dependencies	Relations	Sequential Dependencies & Relations
Dataset size requirements	+	o	++

Complementary properties! → Combine models?

Course content

- Motivation
- Embedding Techniques
  - Feature Engineering
  - Learned Embeddings

- Code Modeling
  - Sequences
  - Graphs
  - Sequences II – Transformers

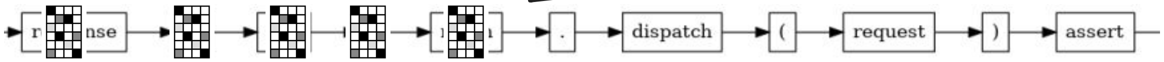
  - Combinations of Sequences & Graphs
  - Interpreters

Sequences & Graphs – Sandwich Models **[10]**

Alternate RNNs/Transformers with GNNs

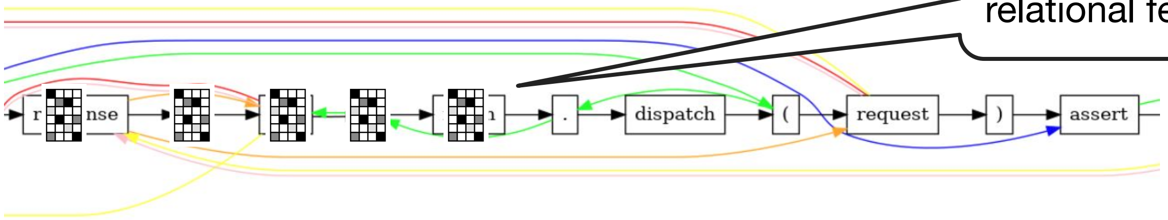
[RNN/Transformer,

States with global features

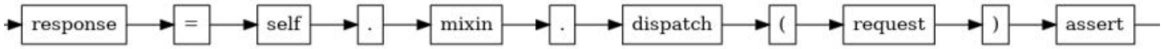


GNN,

States with global + relational features



RNN/Transformer]



Sequences & Graphs – “GREAT” Model [10]

Bias query of attention mechanism to relations

Attention score

$$\alpha_{ij} \sim \mathbf{q}_i \mathbf{k}_j^{\top} / \sqrt{N}$$

$$\alpha_{ij} \sim (\mathbf{q}_i + b_{ij}) \mathbf{k}_j^{\top} / \sqrt{N}$$

Additional bias

$$b_{ij} = \begin{cases} W_\tau \mathbf{e}_\tau, & \exists\, edge(i,j,\tau) \\ 0, & \text{otherwise} \end{cases}$$

Learnable edge type embedding

```
def validate_sources(sources):
  object_name = get_content(sources, 'obj')
  subject_name = get_content(sources, 'subj')
  result = Result()
  result.objects.append(object_name)
  result.subjects.append(object_name)
  return result
```

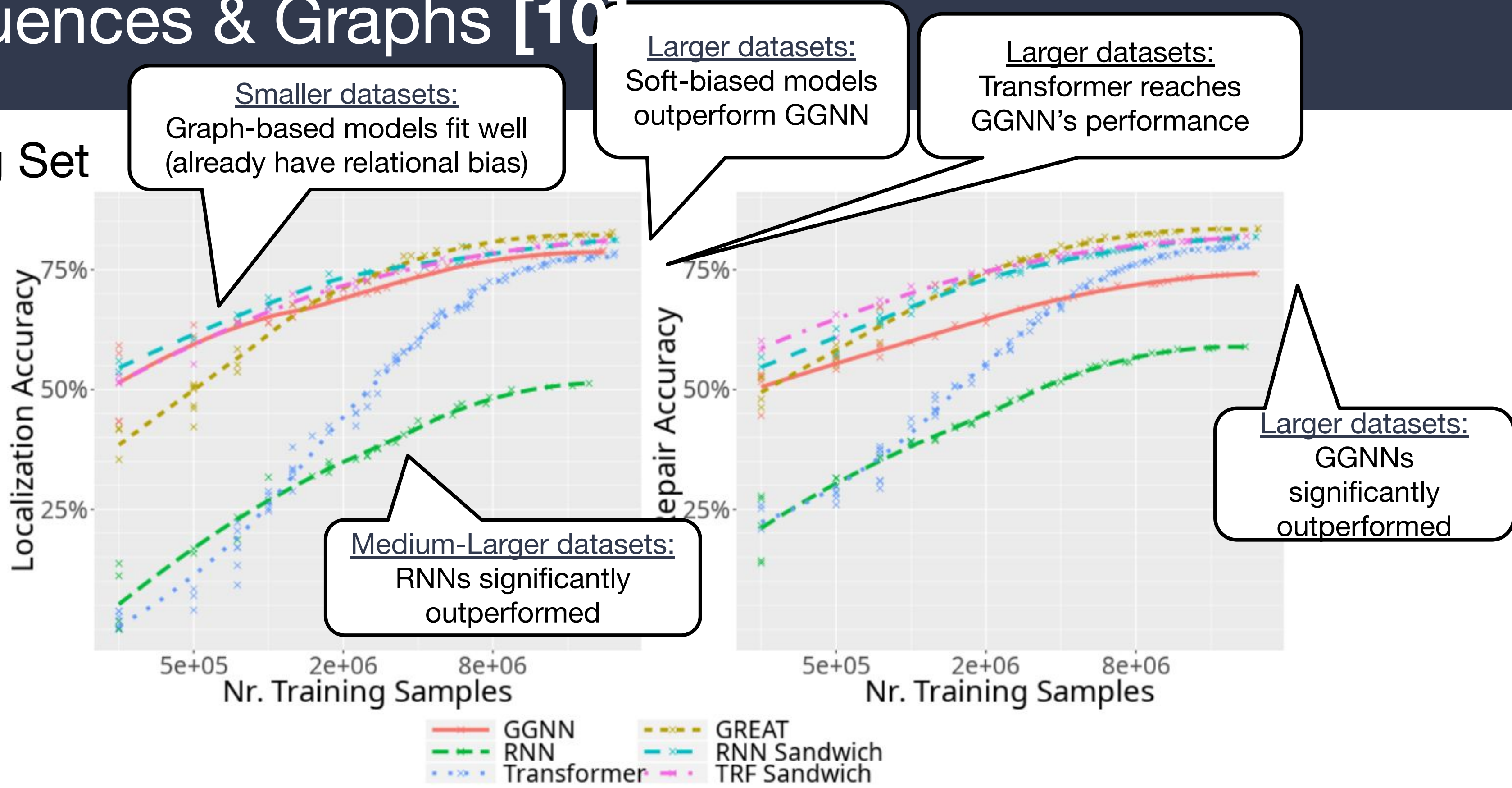
“Soft” relational bias

- Free to attend to any token, even ones that are not in explicit in input data structure (including global information)

Code Modeling

Sequences & Graphs [10]

Training Set



Code Modeling

Sequences & Graphs [10]

Test Set

Transformers on-par with GGNN (Baseline models)

Maximum tokens / sample

Model Family	Class. Accuracy ≤ 250	Class. Accuracy ≤ 1000	Loc & Rep Accuracy ≤ 250	Loc & Rep Accuracy ≤ 1000	Parameters
RNN[1]	71.8%	70.6%	44.4%	42.5%	4.3M
Transformer	75.9%	73.2%	67.7%	63.0%	3.7M
GGNN	81.4%	79.2%	64.0%	60.9%	5.5M
RNN Sandwich	**82.5%**	**81.9%**	75.8%	**73.8%**	12.6M
Transformer Sandwich	81.1%	78.1%	74.5%	71.4%	10M
GREAT	80.1%	76.9%	**76.4%**	73.1%	7.9M

RNN Sandwich best

GREAT and RNN Sandwich best

Overview

Course content

- Motivation
- Embedding Techniques
  - Feature Engineering
  - Learned Embeddings

- Code Modeling
  - Sequences
  - Graphs
  - Sequences II – Transformers

  - Combinations of Sequences & Graphs
  - Interpreters

Interpretation – Instruction Pointer Attention GNN **[11]**

n	Source	Control flow graph
0	v0 = 407	
1	if v0 % 10 < 3:	
2	v0 += 4	
3	else:	
4	v0 -= 2	
5	<exit>	

- Idea: Following the principle of an interpreter

Interpretation – Instruction Pointer Attention GNN **[11]**

n	Source	Control flow graph	Line-by-Line RNN
0	`v0 = 407`		
1	`if v0 % 10 < 3:`		
2	`  v0 += 4`		
3	`else:`		
4	`  v0 -= 2`		
5	`<exit>`		

$$h_t = \mathrm{RNN}\left(h_{t-1}, \mathrm{Embed}\left(x_{n_{t-1}}\right)\right)$$

$$n_t = t$$

h_t : Hidden State

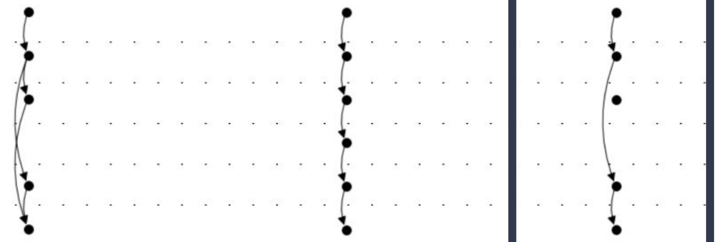
n_t : Instruction Pointer

x_{n_t} : Statement at n_t

- RNN is a natural fit for execution
  - Statement-by-statement
- Problem: Branches / Non-linear control flow

Interpretation – Instruction Pointer Attention GNN [11]

n	Source	Control flow graph	Line-by-Line RNN	Trace RNN
0	`v0 = 407`			
1	`if v0 % 10 < 3:`			
2	`  v0 += 4`			
3	`else:`			
4	`  v0 -= 2`			
5	`<exit>`			



$$h_t = \mathrm{RNN}\left(h_{t-1}, \mathrm{Embed}\left(x_{n_{t-1}}\right)\right)$$

$$n_t = t \qquad n_t = n_t^*$$

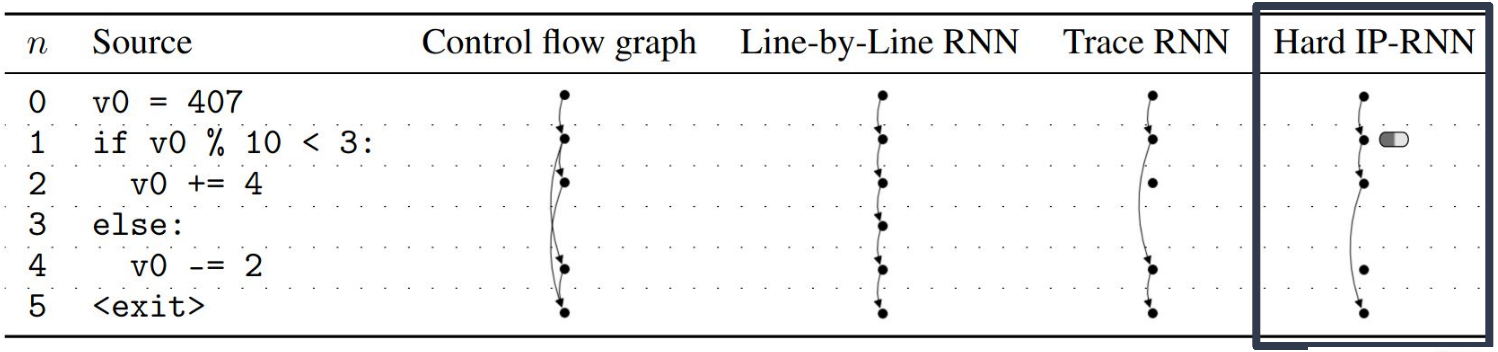
h_t : Hidden State

n_t : Instruction Pointer

x_{n_t} : Statement at n_t

- RNN on execution trace (→ Trace RNN)
- Problem: Not static as execution trace requires execution

Interpretation – Instruction Pointer Attention GNN [11]



$$n_t = t \qquad n_t = n_t^* \qquad n_t = N_{\text{out}}(n_{t-1}) \mid j$$

where $j = \text{argmax Dense}(h_t)$

$$h_t = \text{RNN}\left(h_{t-1}, \text{Embed}\left(x_{n_{t-1}}\right)\right)$$

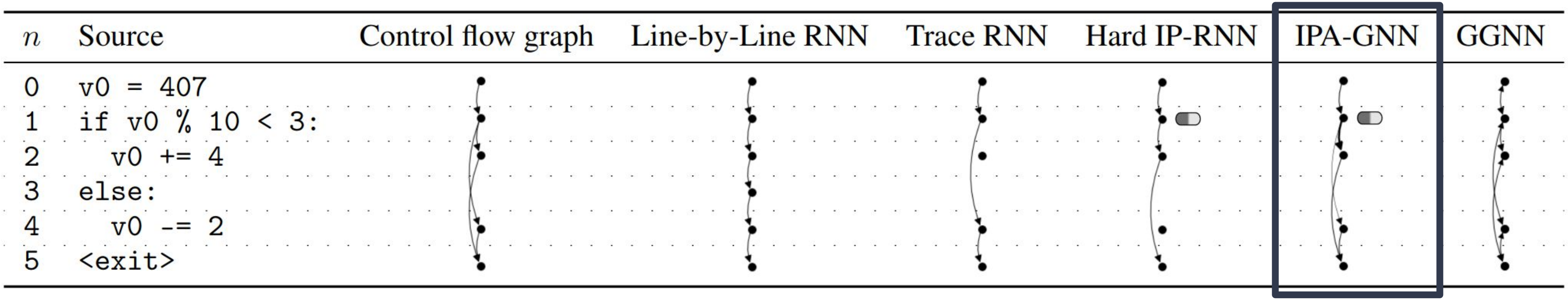
h_t : Hidden State

n_t : Instruction Pointer

x_{n_t} : Statement at n_t

- Predict what branch to take with a Neural Network and update IP accordingly
- Problem: Not differentiable (model makes discrete decisions)

Interpretation – Instruction Pointer Attention GNN [11]



$$h_{t,n} = \sum_{n' \in N_{\text{in}}(n)} p_{t-1,n'} \cdot b_{t,n',n} \cdot a^{(1)}_{t,n}$$

h_t : Hidden State

$a^{(1)}_{t,n}$: State proposal

$b_{t,n',n}$: Branch decision

$p_{t,n}$: Soft instruction pointer

- Soft branch decision (distribution over current statements branches)
- Soft instruction pointer (distribution over all edges)

→ Supports Branches, Differentiable

Interpretation – Instruction Pointer Attention GNN [11]

$$h_{t,n} = \sum_{n' \in N_{\text{in}}(n)} p_{t-1,n'} \cdot b_{t,n',n} \cdot a^{(1)}_{t,n}$$

h_t : Hidden State

$a^{(1)}_{t,n}$: State proposal

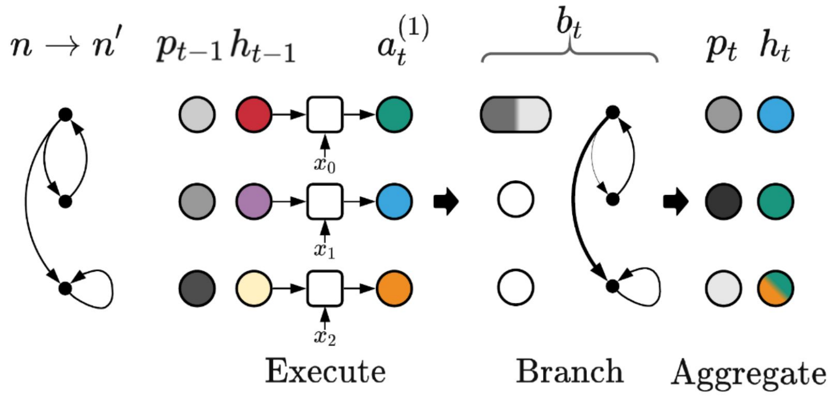
$b_{t,n',n}$: Branch decision

$p_{t,n}$: Soft instruction pointer

$$a^{(1)}_{t,n} = \text{RNN}\left(h_{t-1,n}, \text{Embed}\left(x_n\right)\right)$$

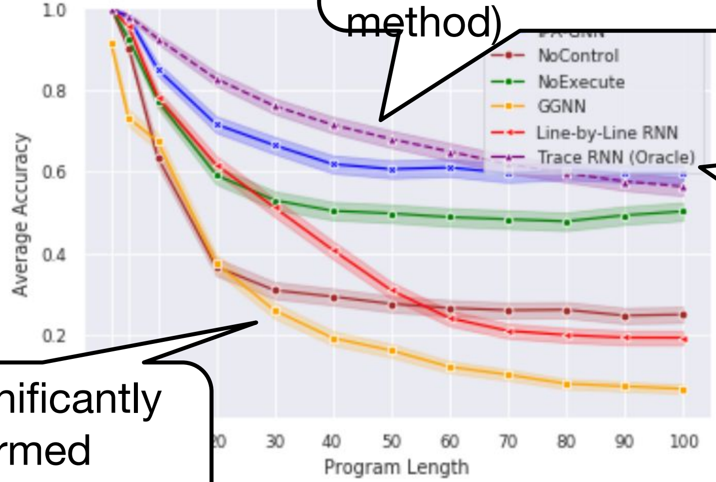
$$b_{t,n,n_1}, b_{t,n,n_2} = \text{softmax}\left(\text{Dense}\left(a^{(1)}_{t,n}\right)\right)$$

$$p_{t,n} = \sum_{n' \in N_{\text{in}}(n)} p_{t-1,n'} \cdot b_{t,n',n}$$



Interpretation – Instruction Pointer Attention GNN [11]

- Learning-to-Execute task
  - Input: Program
  - Output: Result



TraceRNN has best performance (but not static method)

IPA-GNN on-par with TraceRNN on larger programs

GNN significantly outperformed (steps are limiting)

Interpretation – Instruction Pointer Attention GNN [11]

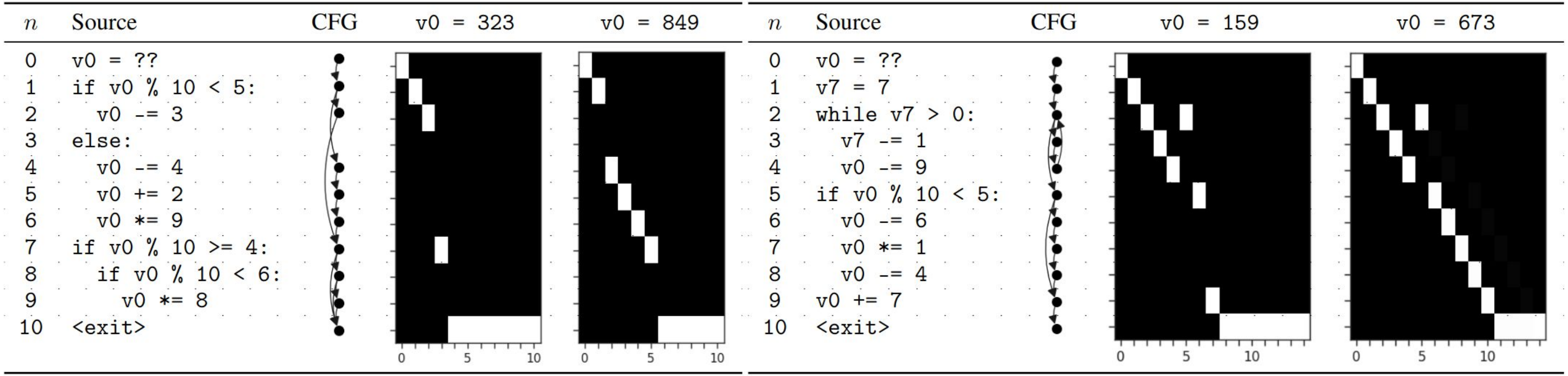
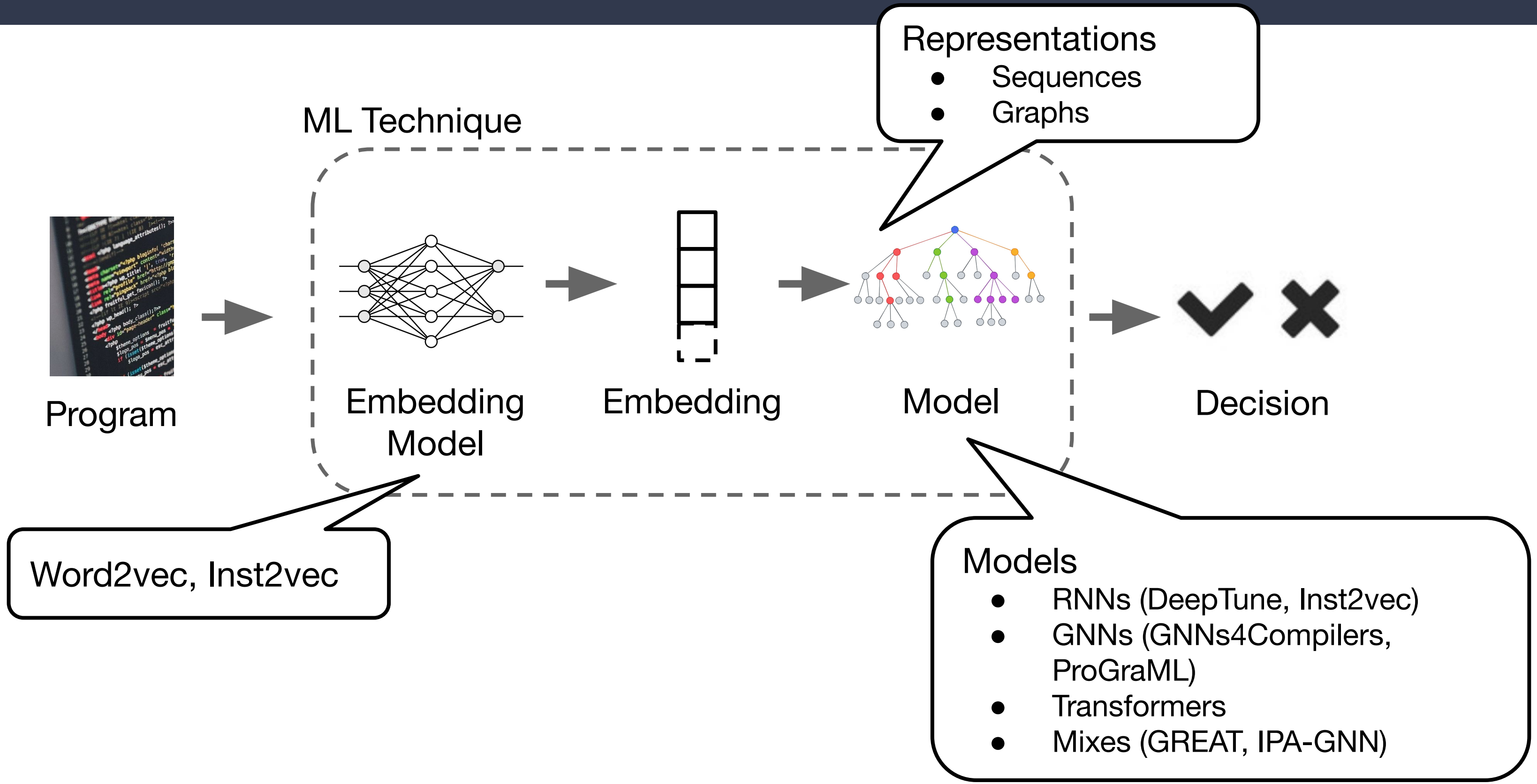


Figure 5: **Instruction Pointer Attention.** Intensity plots show the soft instruction pointer $p_{t,n}$ at each step of the IPA-GNN on two programs, each with two distinct initial values for v0.

- IPA-GNN learns instruction pointer mechanism itself

Summary


Representations
● Sequences
● Graphs
ML Technique
Program
Embedding Model
Embedding
Model
Decision
Word2vec, Inst2vec
Models
● RNNs (DeepTune, Inst2vec)
● GNNs (GNNs4Compilers, ProGraML)
● Transformers
● Mixes (GREAT, IPA-GNN)

Overview

- L1: Motivation and survey of auto-tuning/machine learning for compilers
- L2: Program rewriting schemes - e-graphs and equality saturation
- L3: Program embeddings and Graph Neural Networks
- L4: Program synthesis and neural synthesis
- L5: Neural Machine Translation,Transformers and Large language models

Bibliography

Surveys

1. Wang, Zheng, and Michael O'Boyle. "Machine learning in compiler optimization." Proceedings of the IEEE 106, no. 11 (2018): 1879-1901.
2. Allamanis, Miltiadis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. "A survey of machine learning for big code and naturalness." ACM Computing Surveys (CSUR) 51, no. 4 (2018): 1-37.

Papers

1. Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. "Efficient estimation of word representations in vector space." arXiv preprint arXiv:1301.3781 (2013).
2. Ben-Nun, Tal, Alice Shoshana Jakobovits, and Torsten Hoefler. "Neural code comprehension: A learnable representation of code semantics." Advances in Neural Information Processing Systems 31 (2018).
3. Wang, Zheng, Dominik Grewe, and Michael FP O'Boyle. "Automatic and portable mapping of data parallel programs to opencl for gpu-based heterogeneous systems." ACM Transactions on Architecture and Code Optimization (TACO) 11, no. 4 (2014): 1-26.
4. Cummins, Chris, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. "End-to-end deep learning of optimization heuristics." In 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 219-232. IEEE, 2017.

Bibliography

5. Brauckmann, Alexander, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. "Compiler-based graph representations for deep learning models of code." In Proceedings of the 29th International Conference on Compiler Construction, pp. 201-211. 2020.
6. Cummins, Chris, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, and Hugh Leather. "Programl: Graph-based deep learning for program optimization and analysis." arXiv preprint arXiv:2003.10536 (2020).
7. Brauckmann, Alexander, Andrés Goens, and Jeronimo Castrillon. "ComPy-Learn: A toolbox for exploring machine learning representations for compilers." In 2020 Forum for Specification and Design Languages (FDL), pp. 1-4. IEEE, 2020.
8. Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." Advances in neural information processing systems 30 (2017): 5998-6008.
9. Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." Advances in neural information processing systems 30 (2017): 5998-6008.

Bibliography

10. Hellendoorn, Vincent Josua, Petros Maniatis, Rishabh Singh, Charles Sutton, and David Bieber. "Global relational models of source code." (2020).
11. Bieber, David, Charles Sutton, Hugo Larochelle, and Daniel Tarlow. "Learning to execute programs with instruction pointer attention graph neural networks." Advances in Neural Information Processing Systems 33 (2020): 8626-8637.