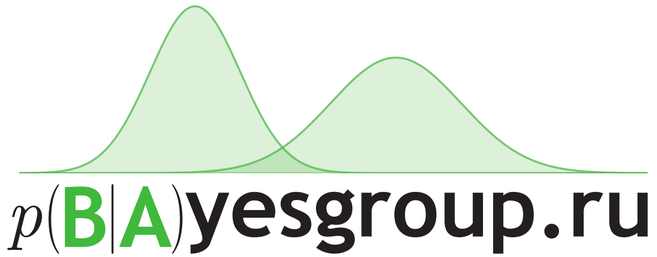
Adapting Natural Language Processing to Source Code Processing: Handling Syntactic Structure and Identifiers

Nadia Chirkova

Higher School of Economics, Samsung-HSE Laboratory
Moscow, Russia

NATIONAL RESEARCH UNIVERSITY

SAMSUNG Research



Deep learning for source code

Code as input	Code as output	Code as both input and output
Predicting bugs (code → bug location) (code → bug fix)	Code completion (code → next token(s))	
Code summarization (code → comment) (code → function name)	Semantic parsing (text → code)	Code translation (Java code → Python code) (updating code for a new library version)
Commit message generation (code changes → commit message)	Code retrieval (text → code from a database)	

What type of data is source code?

```
def masked_softmax(x, mask=None):
    if mask is not None:
        mask = mask.float()
    if mask is not None:
        x_masked = x.masked_fill(1-mask, -1e18)
    else:
        x_masked = x
    x_max = x_masked.max(1)[0]
    x_exp = (x - x_max.unsqueeze(-1)).exp()
    if mask is not None:
        x_exp = x_exp * mask.float()
    return x_exp / x_exp.sum(1).unsqueeze(-1)
```

What type of data is source code?

Text?

```
def masked_softmax(x, mask=None):
    if mask is not None:
        mask = mask.float()
    if mask is not None:
        x_masked = x.masked_fill(1-mask, -1e18)
    else:
        x_masked = x
    x_max = x_masked.max(1)[0]
    x_exp = (x - x_max.unsqueeze(-1)).exp()
    if mask is not None:
        x_exp = x_exp * mask.float()
    return x_exp / x_exp.sum(1).unsqueeze(-1)
```

What type of data is source code?

Text?

✓ Syntactic structure

✓ Invariance to changing identifier names

(and other specifics)

```
1  def masked_softmax(x, mask=None):
2      if mask is not None:
3          mask = mask.float()
4      if mask is not None:
5          x_masked = x.masked_fill(1-mask, -1e18)
6      else:
7          x_masked = x
8      x_max = x_masked.max(1)[0]
9      x_exp = (x - x_max.unsqueeze(-1)).exp()
10     if mask is not None:
11         x_exp = x_exp * mask.float()
12     return x_exp / x_exp.sum(1).unsqueeze(-1)
```

Plan

- Empirical study of syntax-based Transformers for source code
- Handling out-of-vocabulary identifiers in Transformers for source code
- Handling anonymized identifiers in RNNs for source code

Plan

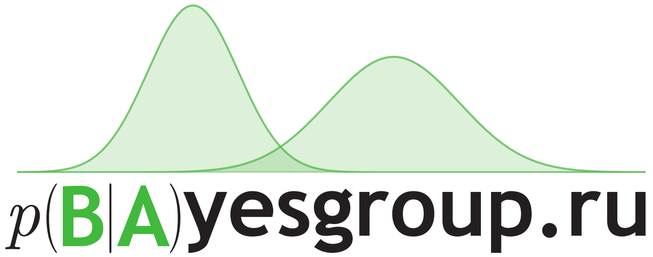
- Empirical study of syntax-based Transformers for source code
- Handling out-of-vocabulary identifiers in Transformers for source code
- Handling anonymized identifiers in RNNs for source code

Empirical Study of Transformers for Source Code

Nadezhda Chirkova, Sergey Troshin

Higher School of Economics, Samsung-HSE Laboratory
Moscow, Russia

SAMSUNG
Research



Motivation

- A lot of recent works adjust Transformer for handling the syntactic structure of the input:

Which one performs better?

Model	Task
Tree positional encodings (Shiv19)	program translation semantic parsing
Tree relative attention (Kim20)	code competion
GGNN-Sandwich (Hellendoorn20)	bug fixing
…	…

Overview

- Goal: understanding whether Transformers are able to utilize syntactic structure and what is the best approach to do it

- 5 syntax-capturing Transformer modifications

  3 tasks: code completion, bug fixing, function naming

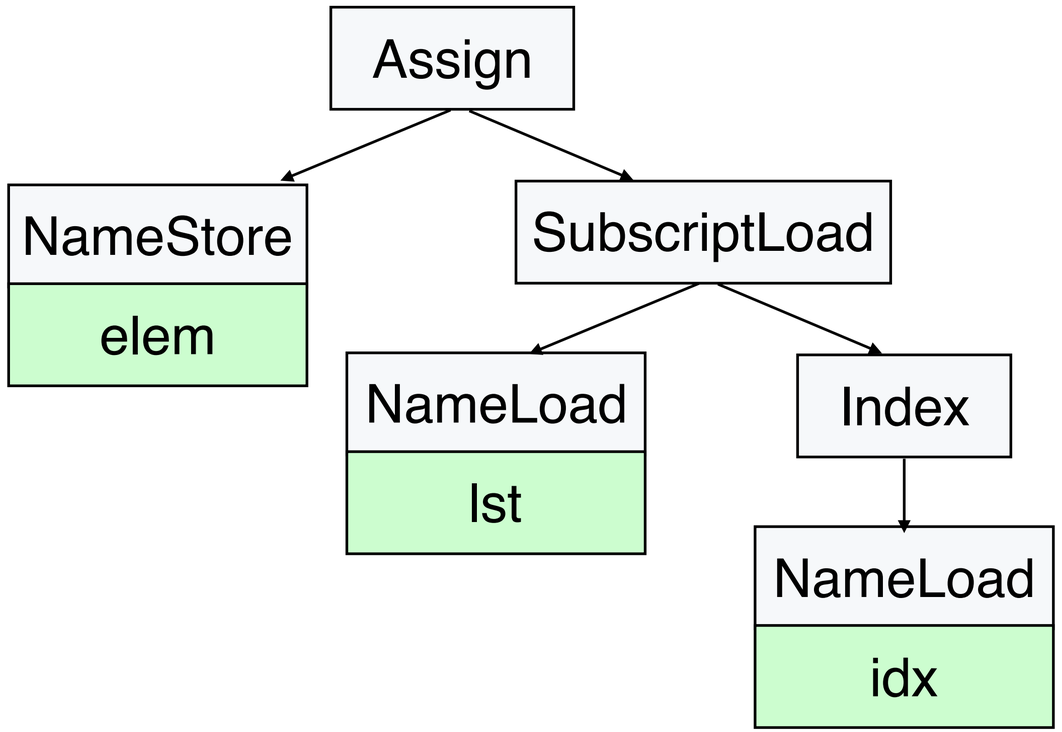
  2 datasets: Python150k and Javascript150k

- Thoughtful experimental setup:
  - repository-based train / test split
  - 2 settings: full data and anonymized data
  - re-implementing all techniques in a unified framework (e. g. same data preprocessing for all techniques)

Abstract syntax tree (AST)

Code: `elem = lst[idx]`

Abstract syntax tree (AST):

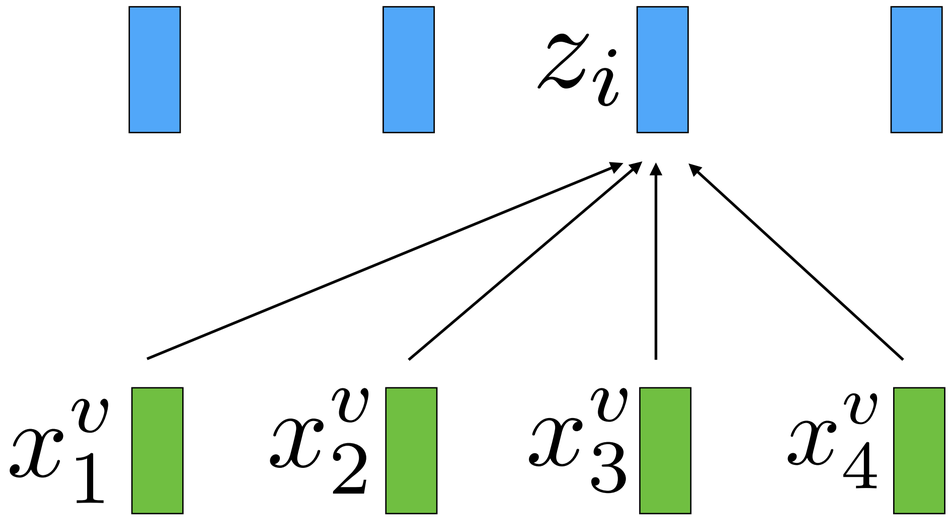


✓ Type in each node

✓ Values in some nodes (identifiers, constants, function names etc.)

✓ Ordered children

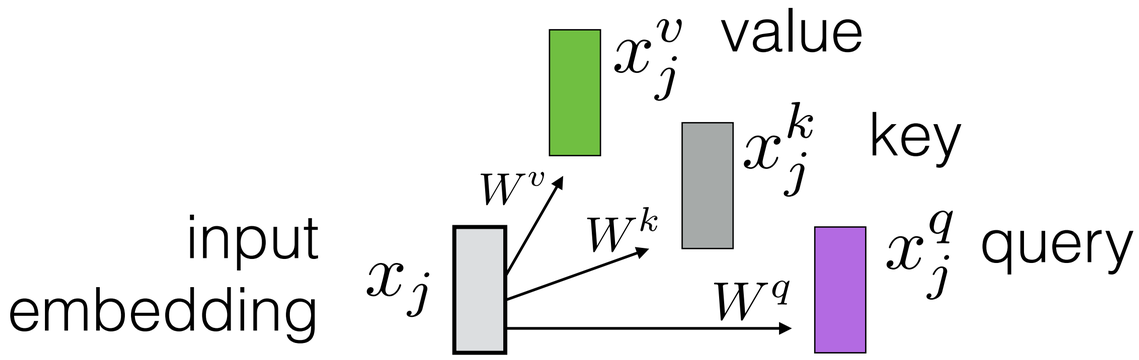
Self-attention mechanism



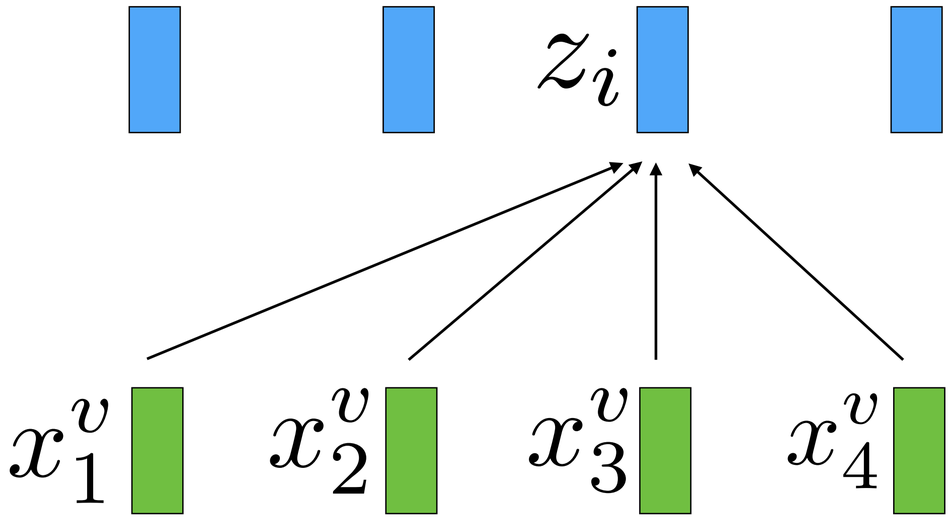
$$z_i = \sum_j \tilde{\alpha}_{ij} x_j^v$$

$$\tilde{\alpha}_{ij} = \frac{\exp(a_{ij})}{\sum_j \exp(a_{ij})}$$

$$a_{ij} = \frac{x_i^q x_j^{k\,T}}{\sqrt{d_z}}$$



Self-attention mechanism



$$z_i = \sum_j \tilde{\alpha}_{ij} x_j^v$$

$$\tilde{\alpha}_{ij} = \frac{\exp(a_{ij})}{\sum_j \exp(a_{ij})}$$

$$a_{ij} = \frac{x_i^q {x_j^k}^T}{\sqrt{d_z}}$$

+multiple heads

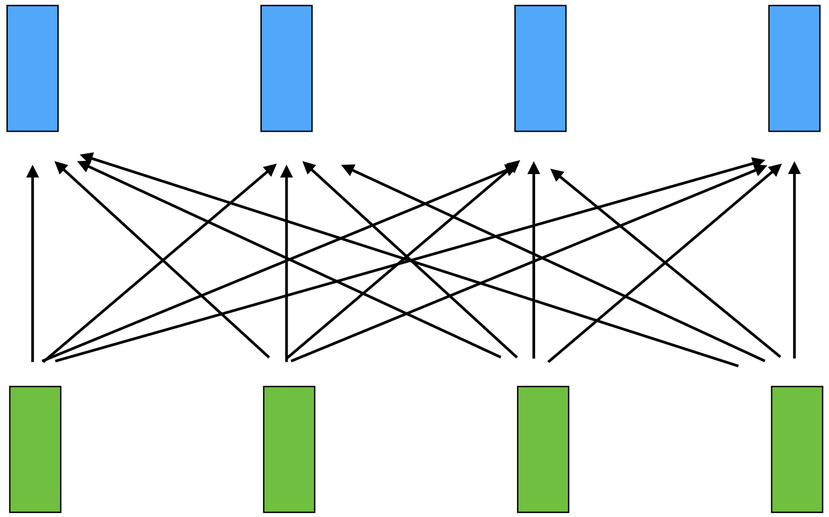
+skip-connection, layer normalization, fully-connected layer

+ 6 layers

+encoder / decoder

Encoder and Decoder in Transformer

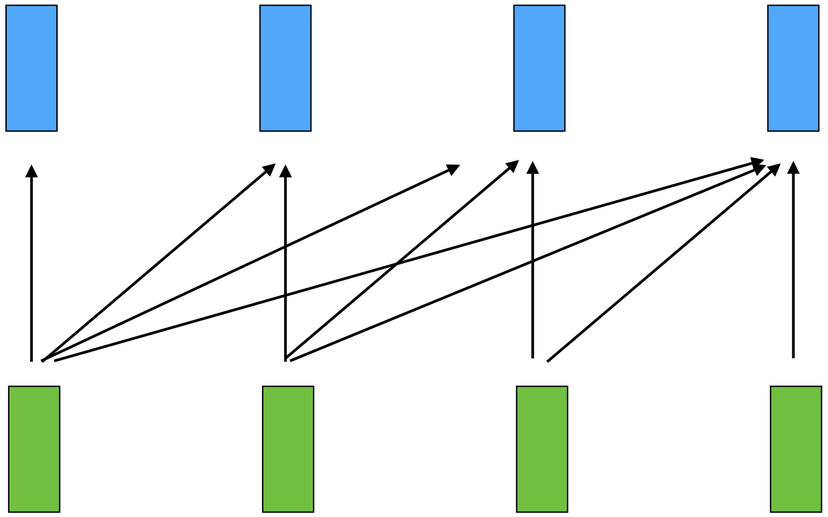
Encoder



Some random short sequence

Decoder

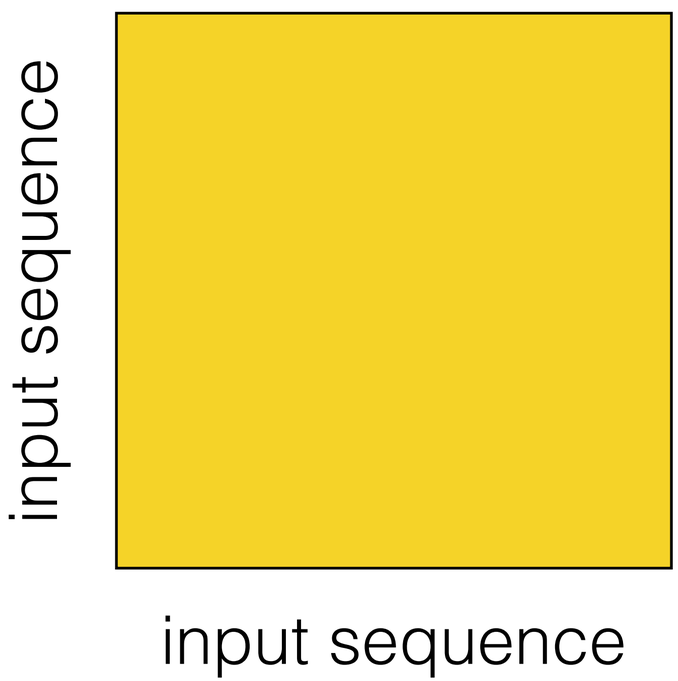
Some random short sequence



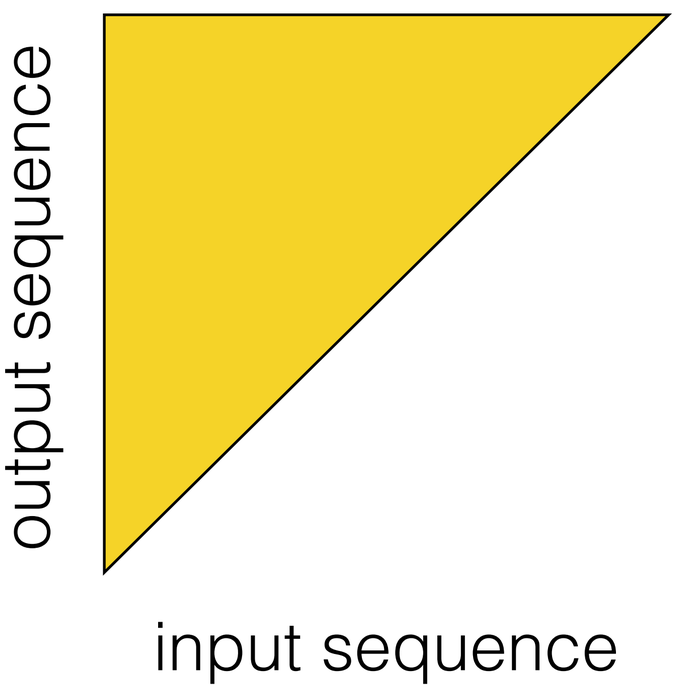
<start> Some random short

Encoder and Decoder: attention masks

Encoder



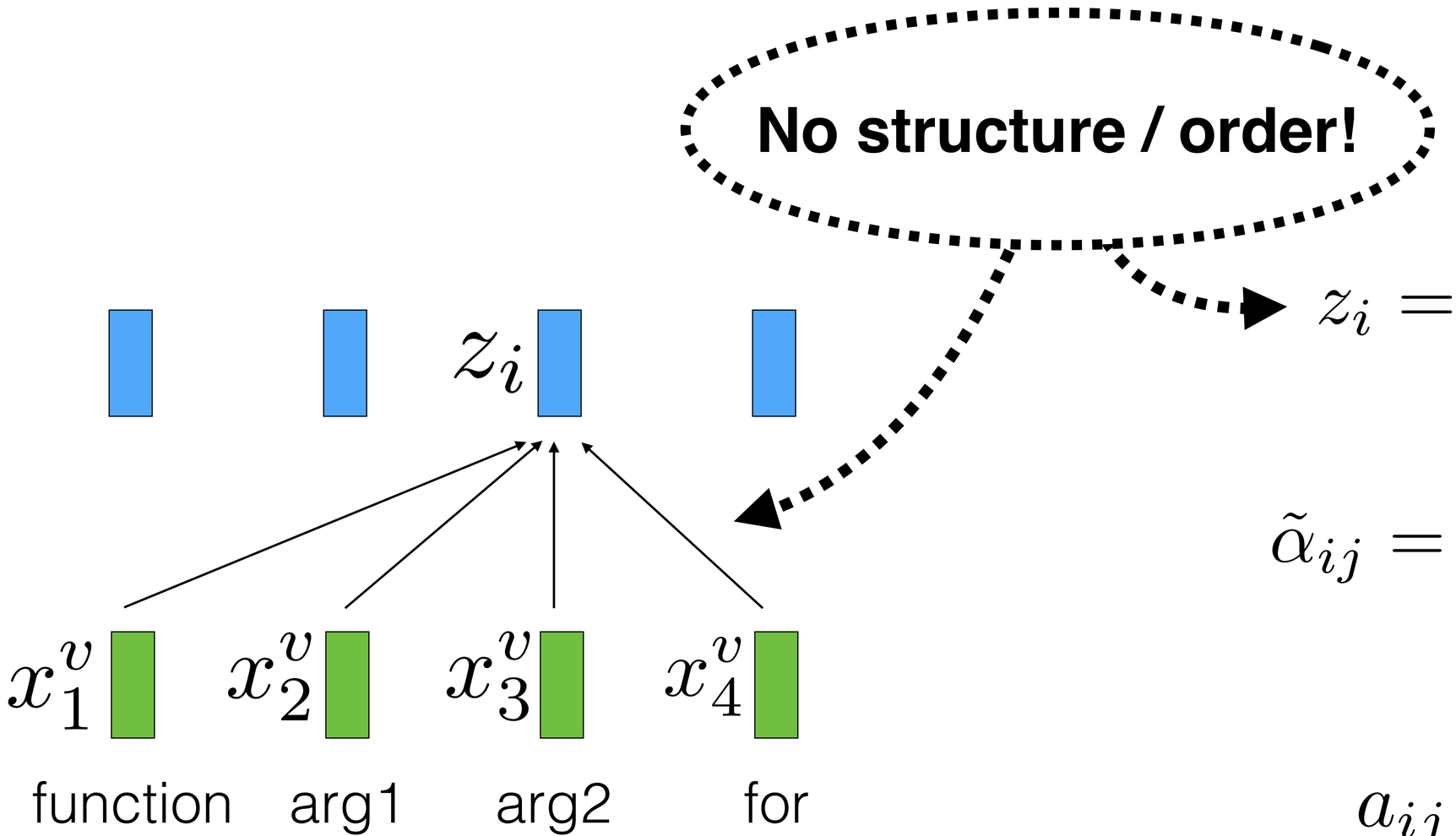
Decoder



Transformer: GIF

https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html

Self-attention: “bag” of elements



$$z_i = \sum_j \tilde{\alpha}_{ij} x_j^v$$

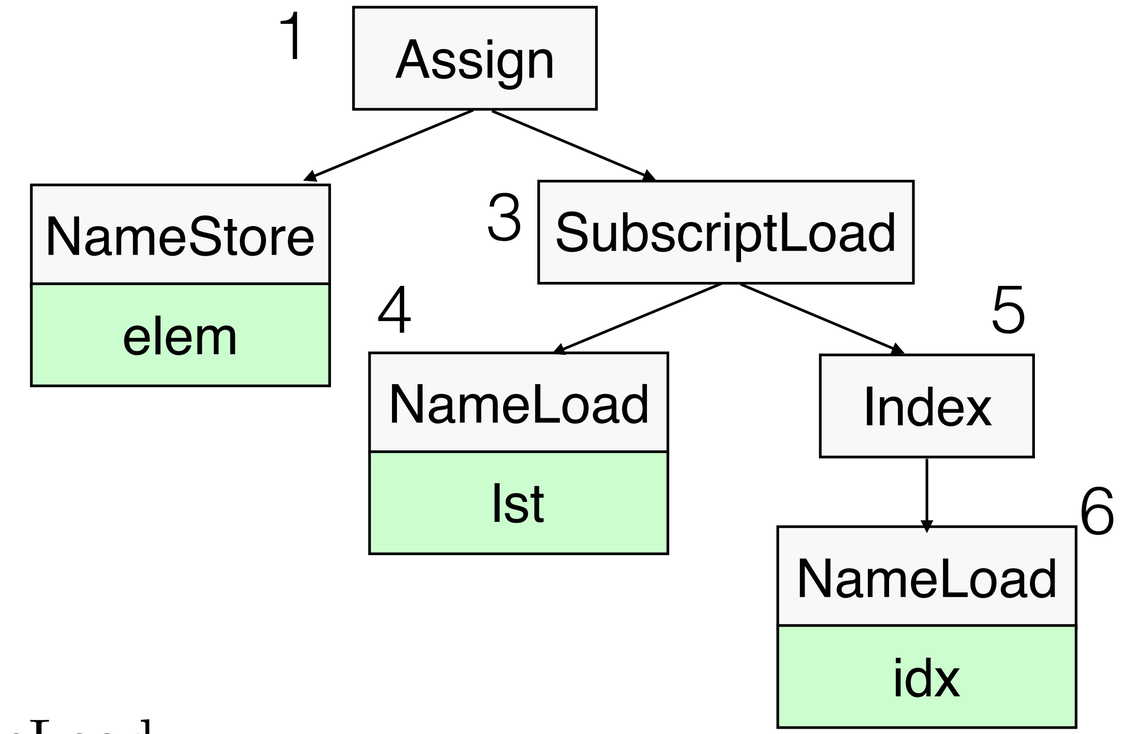
$$\tilde{\alpha}_{ij} = \frac{\exp(a_{ij})}{\sum_j \exp(a_{ij})}$$

$$a_{ij} = \frac{x_i^q {x_j^k}^T}{\sqrt{d_z}}$$

AST depth-first traversal

Code: `elem = lst[idx]`

Abstract syntax ree (AST):



sequence

AST depth-first traversal:

Assign	NameStore	SubscriptLoad	...
⟨empty⟩	elem	⟨empty⟩	...

...	NameLoad	Index	NameLoad
...	lst	⟨empty⟩	idx

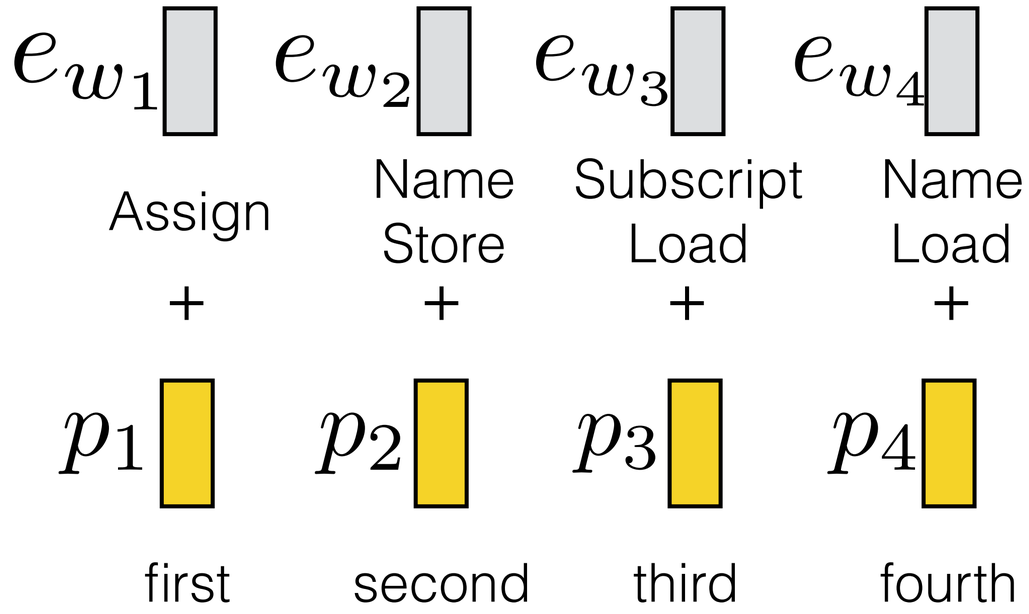
Sequential positional encodings / embeddings

Input layer:

$$x_j = e_{w_j} + p_j$$

Positional embeddings:

$p_j = e_j$ — learnable embeddings

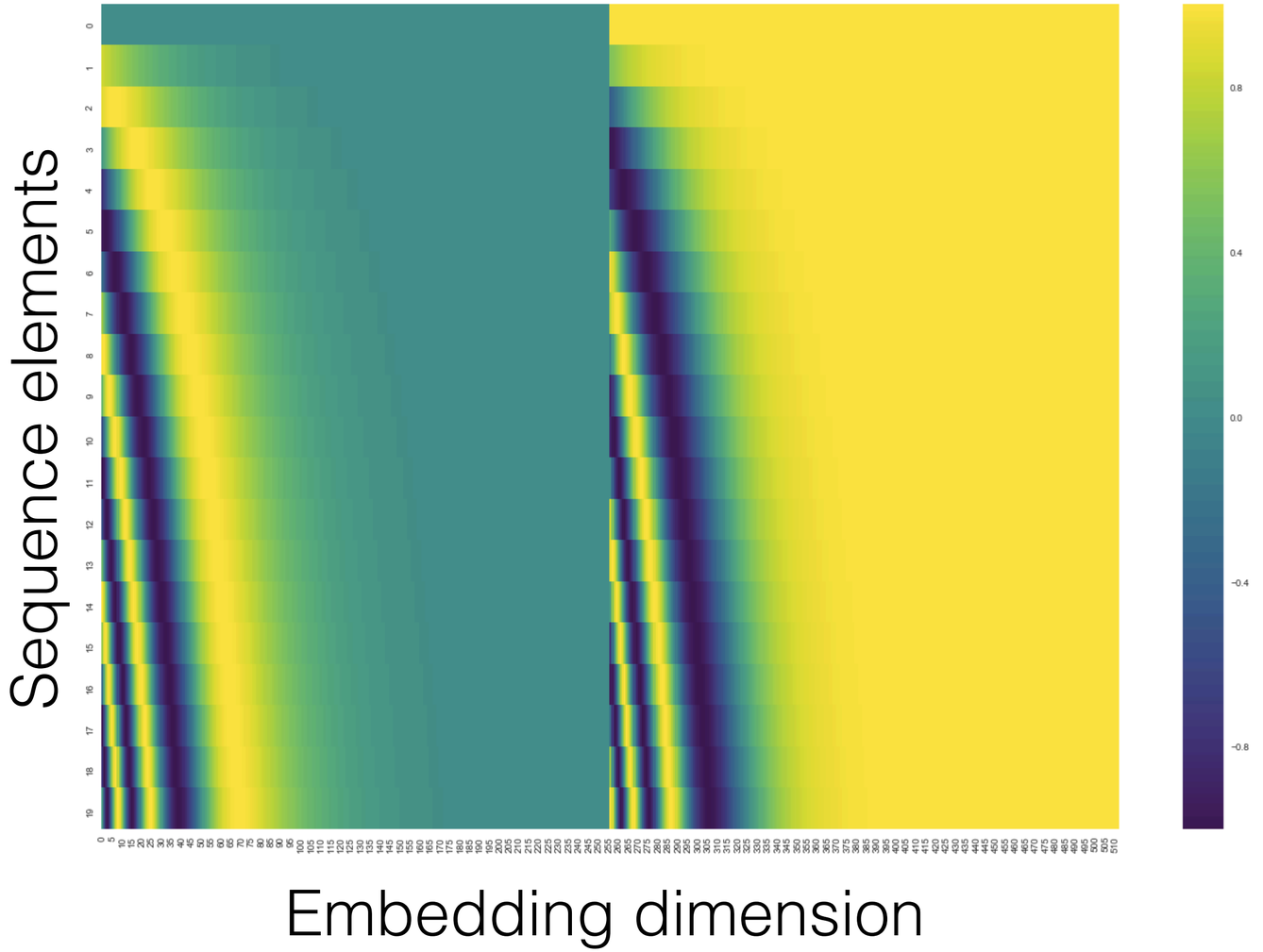


Positional encoding:

$$p_{j,m} = \begin{cases} sin(\omega_k j) & \text{if } m = 2k \\ cos(\omega_k j) & \text{if } m = 2k+1 \end{cases}$$

$$w_k = \frac{1}{10000^{2k/d_x}}$$

Sequential positional encodings: properties



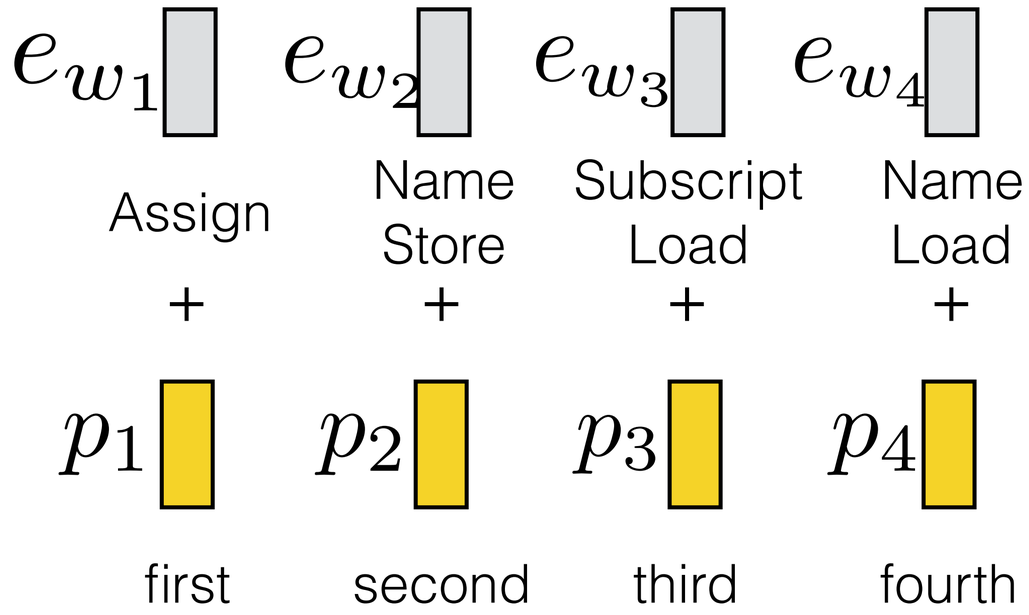
If j_1 and j_2 are two positions with distance ϕ , then their encodings satisfy

$$p_{j_1} = A_\phi p_{j_2}$$

Sequential positional encodings / embeddings

Input layer:

$$x_j = e_{w_j} + p_j$$



Positional embeddings:

Parameters: learnable embeddings

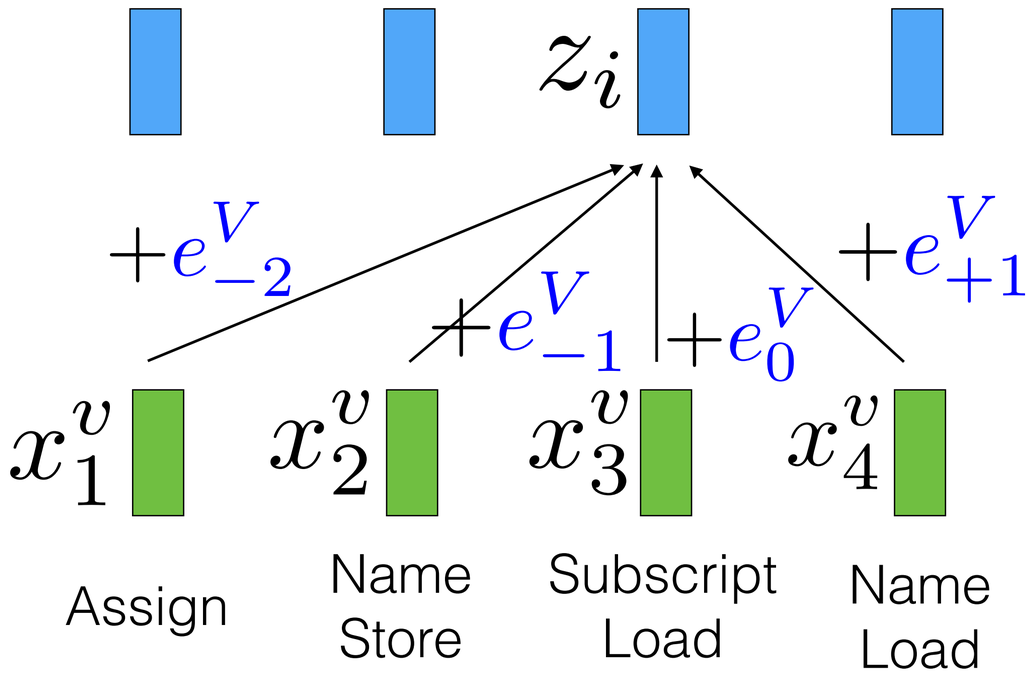
Hyperparameters: none

Positional encoding:

Parameters: none

Hyperparameters: none

Sequential relative attention



embedding of "relation"

$$z_i = \sum_j \tilde{\alpha}_{ij}(x^v_j + e^V_{i-j})$$

$$\tilde{\alpha}_{ij} = \frac{\exp(a_{ij})}{\sum_j \exp(a_{ij})}$$

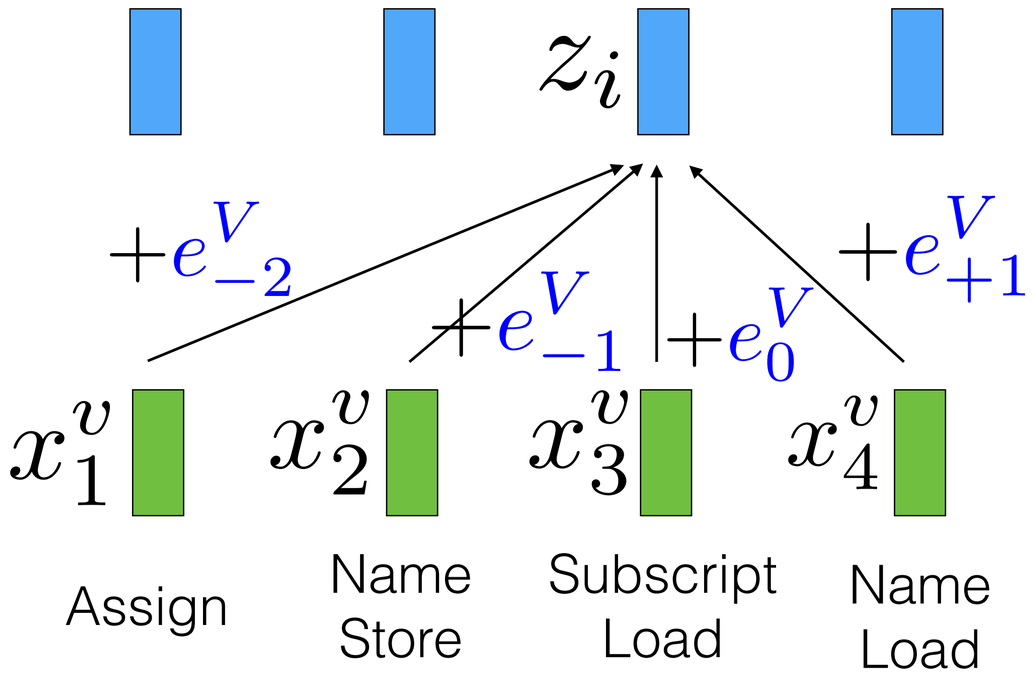
$$a_{ij} = \frac{x^q_i(x^k_j + e^K_{i-j})^T}{\sqrt{d_z}}$$

embedding of "relation"

Learnable embeddings $e^K_{i-j}, e^V_{i-j} \in \mathbb{R}^{d_x}$

Shaw, P., Uszkoreit, J., and Vaswani, A. Self-Attention with Relative Position Representations. NAACL18

Sequential relative attention



Parameters:
embeddings of “relations”

Hyperparameters:
maximum distance between tokens

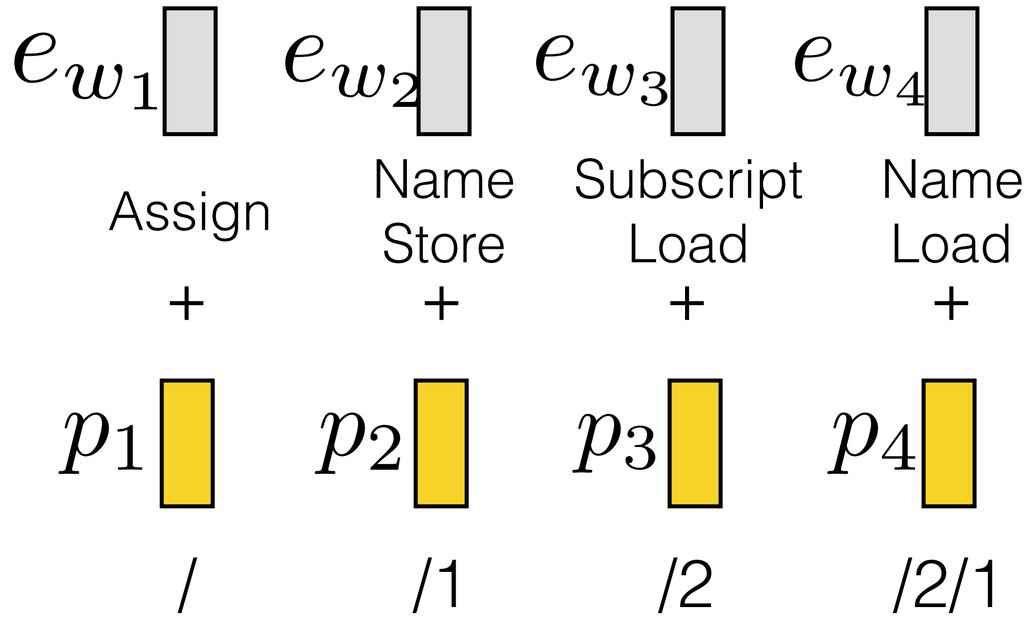
Learnable embeddings $e^K_{i-j}, e^V_{i-j} \in \mathbb{R}^{d_x}$

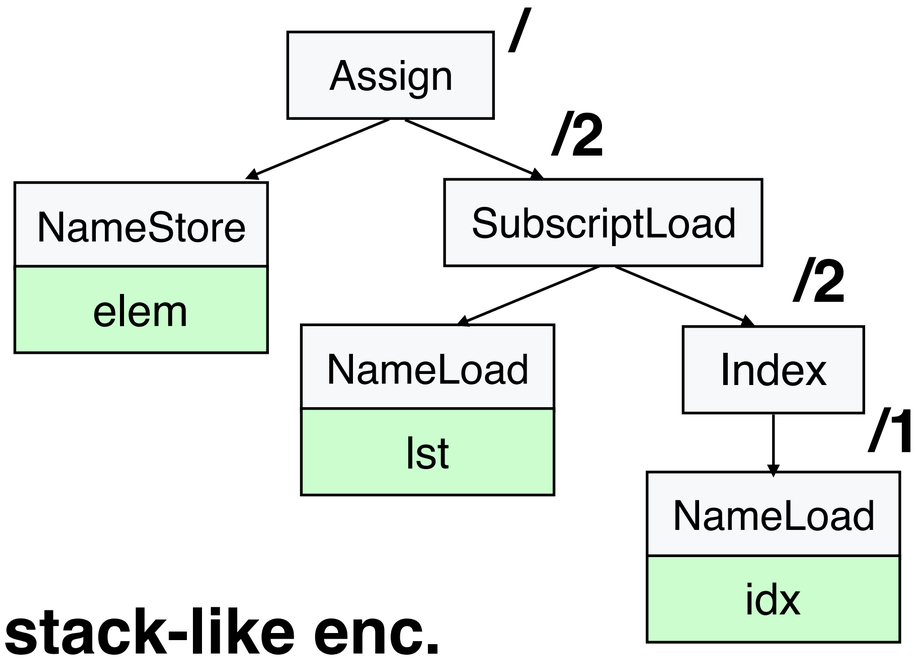
Shaw, P., Uszkoreit, J., and Vaswani, A. Self-Attention with Relative Position Representations. NAACL18

Tree positional encoding

Input layer:

$$x_j = e_{w_j} + p_{\text{path}-\text{to}-j}$$





stack-like enc.

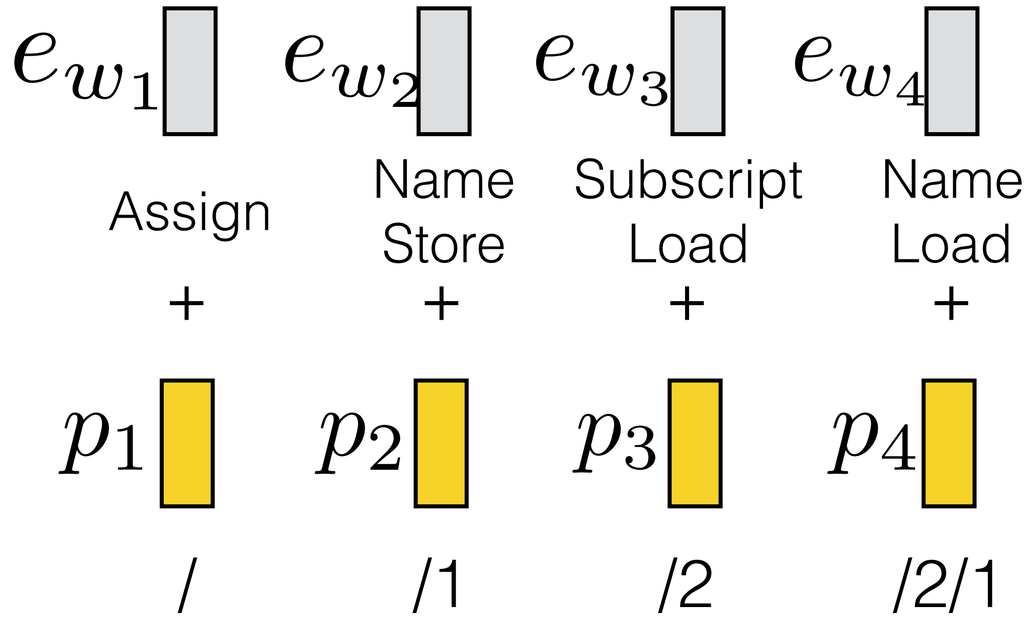
/	**000 000 000**
/1	**100 000 000**
/2	**010 000 000**
/2/1	**100 010 000**
/2/2	**010 010 000**
/2/2/1	**100 010 010**

Shiv, V., and Quirk, C. Novel positional encodings to enable tree-based transformers. NeurIPS19

Tree positional encoding

Input layer:

$$x_j = e_{w_j} + p_{\text{path}-\text{to}-j}$$



If path_1 and path_2 are two positions with path ϕ, then their encodings satisfy

$$p_{\text{path}_1} = A_\phi p_{\text{path}_2}$$

product of "up" and "down" transformation matrices

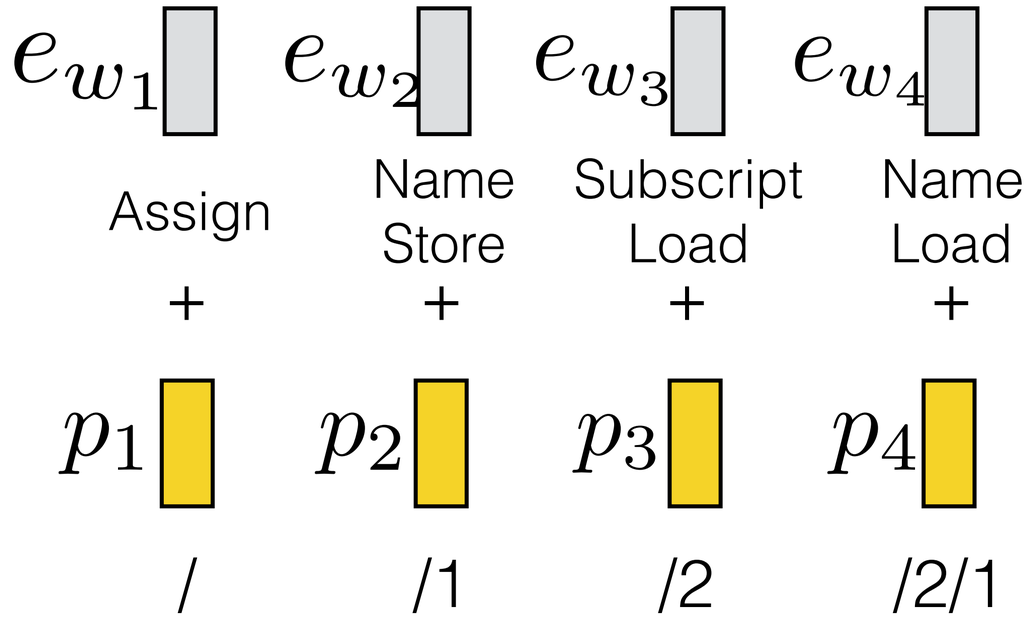
	stack-like enc. →
/	**000 000 000**
/1	**100 000 000**
/2	**010 000 000**
/2/1	**100 010 000**
/2/2	**010 010 000**
/2/2/1	**100 010 010**

Shiv, V., and Quirk, C. Novel positional encodings to enable tree-based transformers. NeurIPS19

Tree positional encoding

Input layer:

$$x_j = e_{w_j} + p_{\text{path}-\text{to}-j}$$



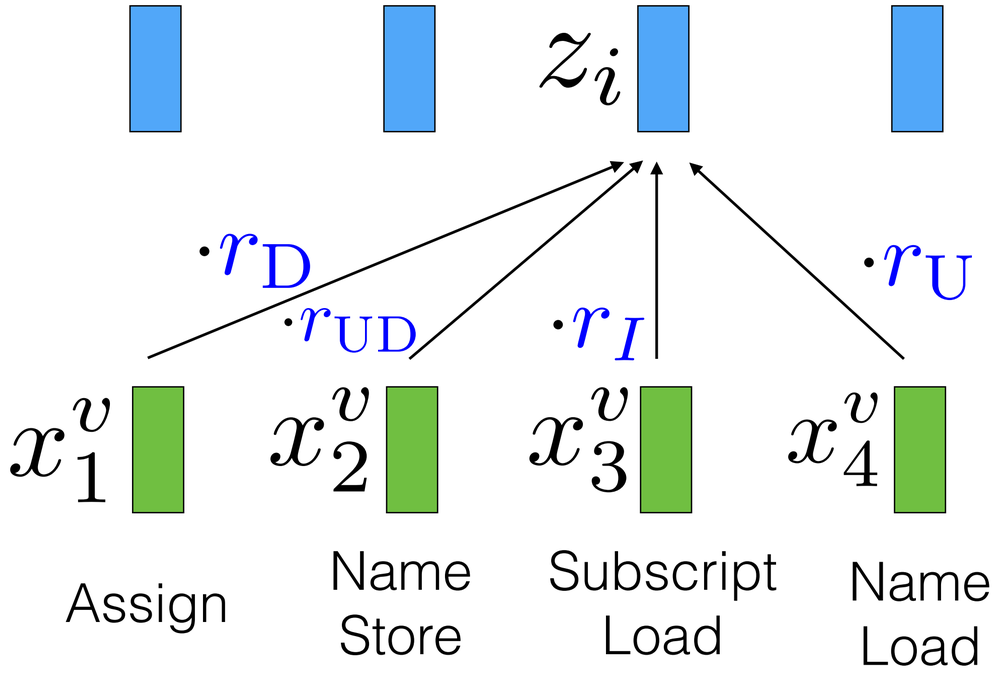
Parameters: p's

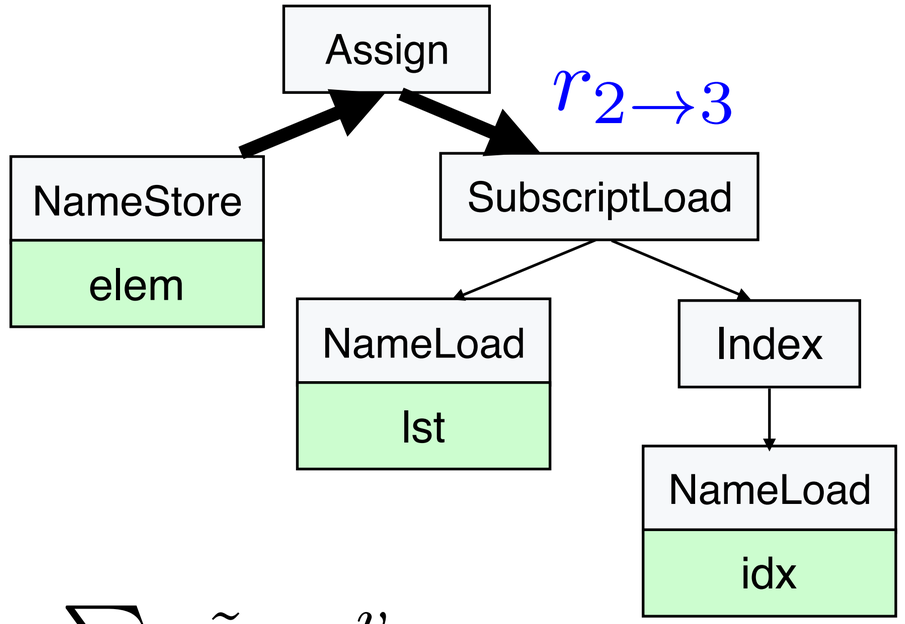
Hyperparameters:
maximum children count
maximum path length

stack-like enc. →

/	**000 000 000**	$\odot[111\ ppp\ p^2p^2p^2]$
/1	**100 000 000**	
/2	**010 000 000**	
/2/1	**100 010 000**	…
/2/2	**010 010 000**	
/2/2/1	**100 010 010**	$\odot[111\ ppp\ p^2p^2p^2]$

Tree relative attention





Learnable 1-dim embeddings $r_{j\to i} \in \mathbb{R}$

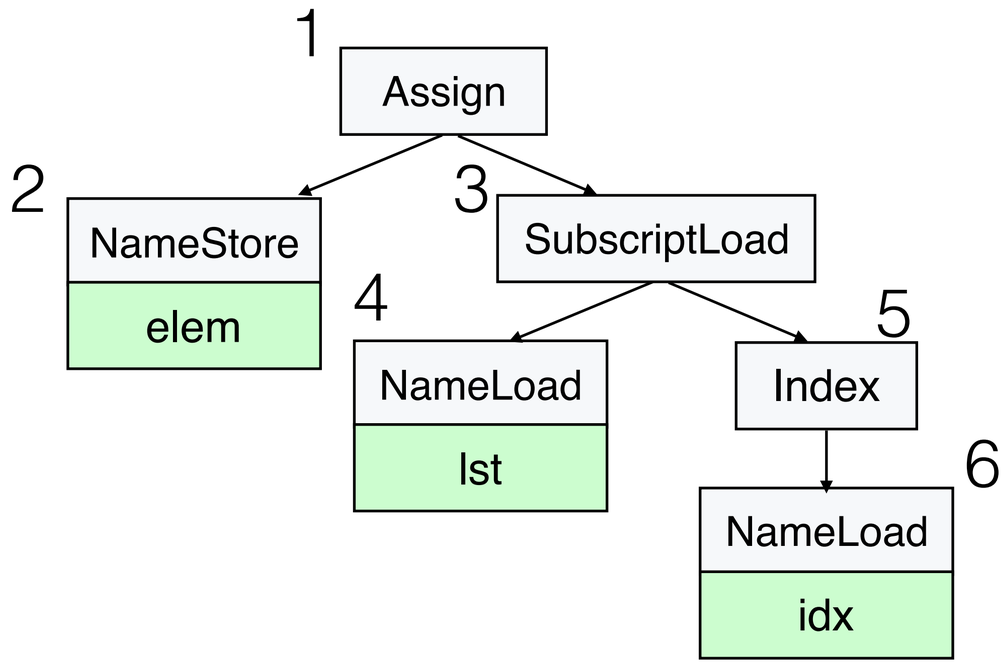
$$z_i = \sum_j \tilde{\alpha}_{ij} x_j^v$$

$$\tilde{\alpha}_{ij} = \frac{\exp(a_{ij} \cdot r_{j\to i})}{\sum_j \exp(a_{ij} \cdot r_{j\to i})}$$

$$a_{ij} = \frac{x_i^q {x_j^k}^T}{\sqrt{d_z}}$$

Kim, S., Zhao, J., Tian, Y., and Chandra, S. Code Prediction by Feeding Trees to Transformers. 2020

Tree relative attention



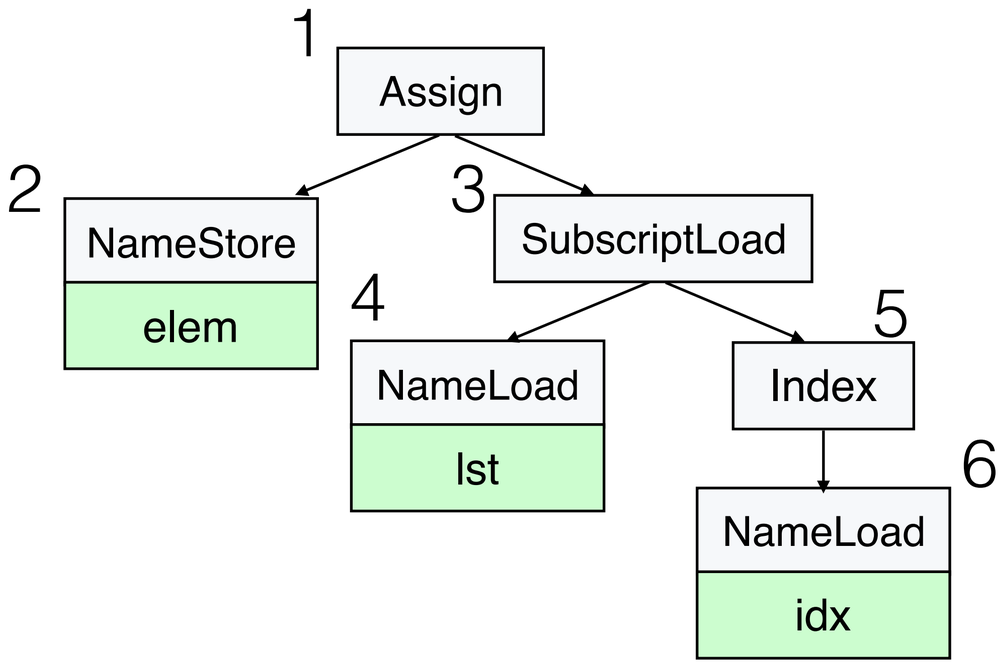
Pairwise relations between AST nodes:

	1	2	3	4	5	6
1	I	D	D	DD	DD	DDD
2	U	I	UD	UDD	UDD	UDDD
3	U	UD	I	D	D	DD
4	UU	UUD	U	I	UD	UDD
5	UU	UUD	U	UD	I	D
6	UUU	UUUD	UU	UUD	U	I

Learn 1-dim embedding for each relation

Kim, S., Zhao, J., Tian, Y., and Chandra, S. Code Prediction by Feeding Trees to Transformers. 2020

Tree relative attention



Pairwise relations between AST nodes:

	1	2	3	4	5	6
1	I	D	D	DD	DD	DDD
2	U	I	UD	UDD	UDD	UDDD
3	U	UD	I	D	D	DD
4	UU	UUD	U	I	UD	UDD
5	UU	UUD	U	UD	I	D
6	UUU	UUUD	UU	UUD	U	I

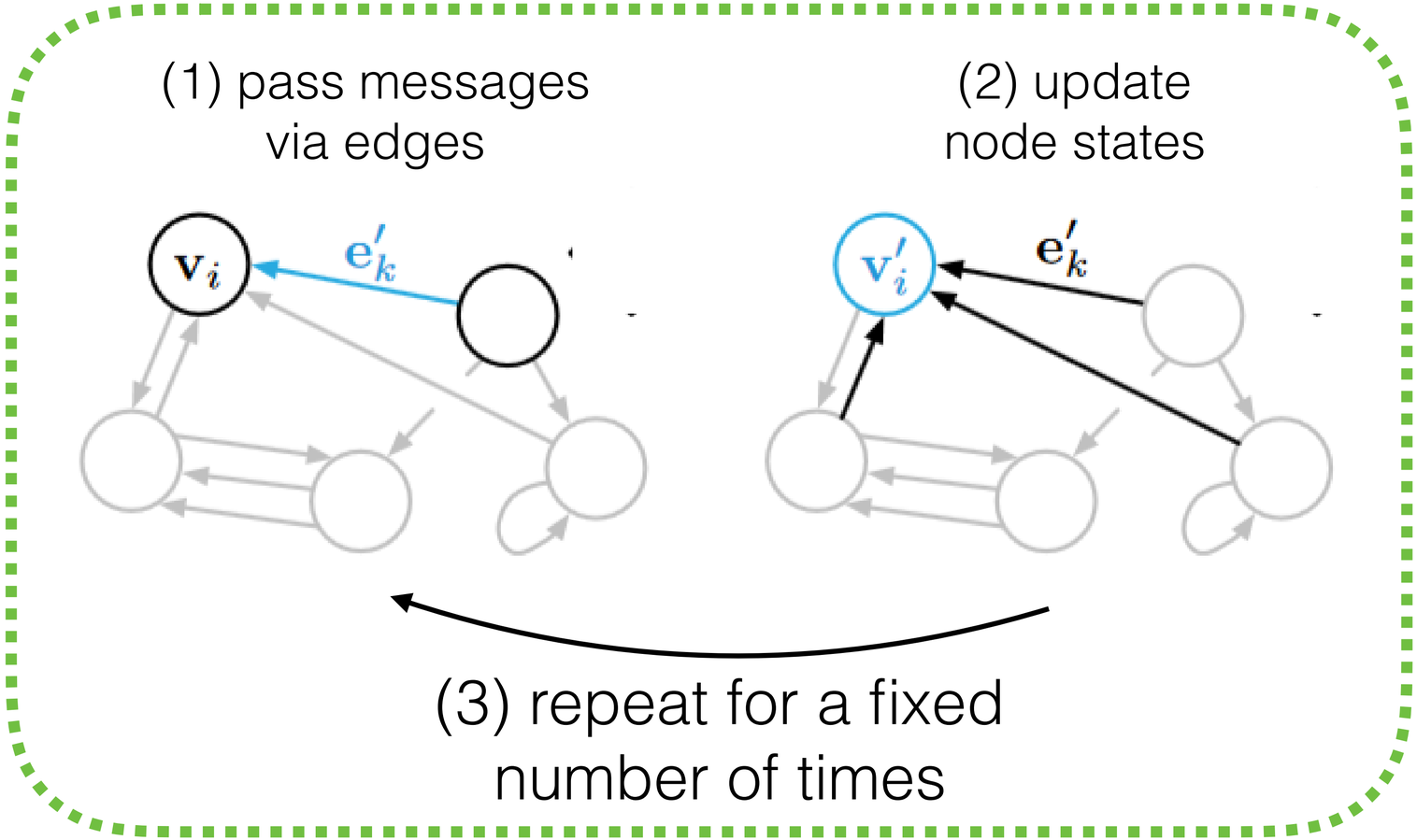
Parameters:
1-dim embeddings
of “relations”

Hyperparameters: none
(or the maximum number of relations)

Kim, S., Zhao, J., Tian, Y., and Chandra, S. Code Prediction by Feeding Trees to Transformers. 2020

GGNN-Sandwich

Graph Gated Neural Network (GGNN):



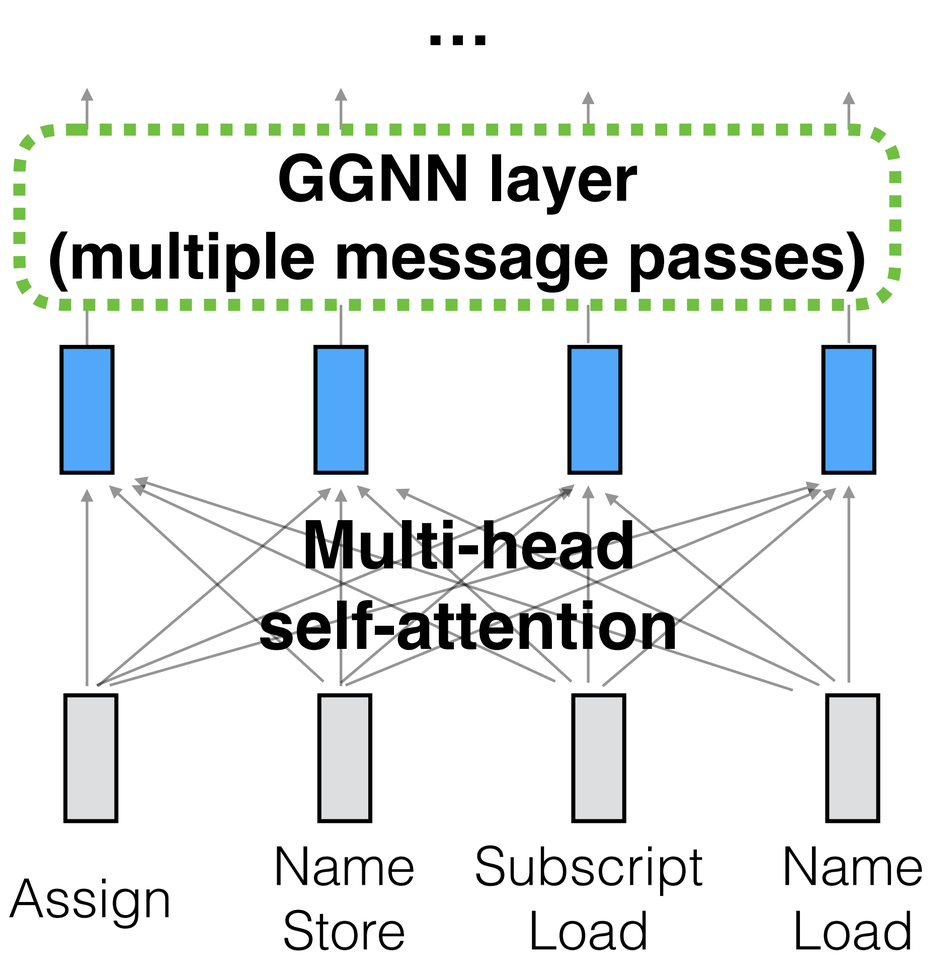
Input:
states of all nodes

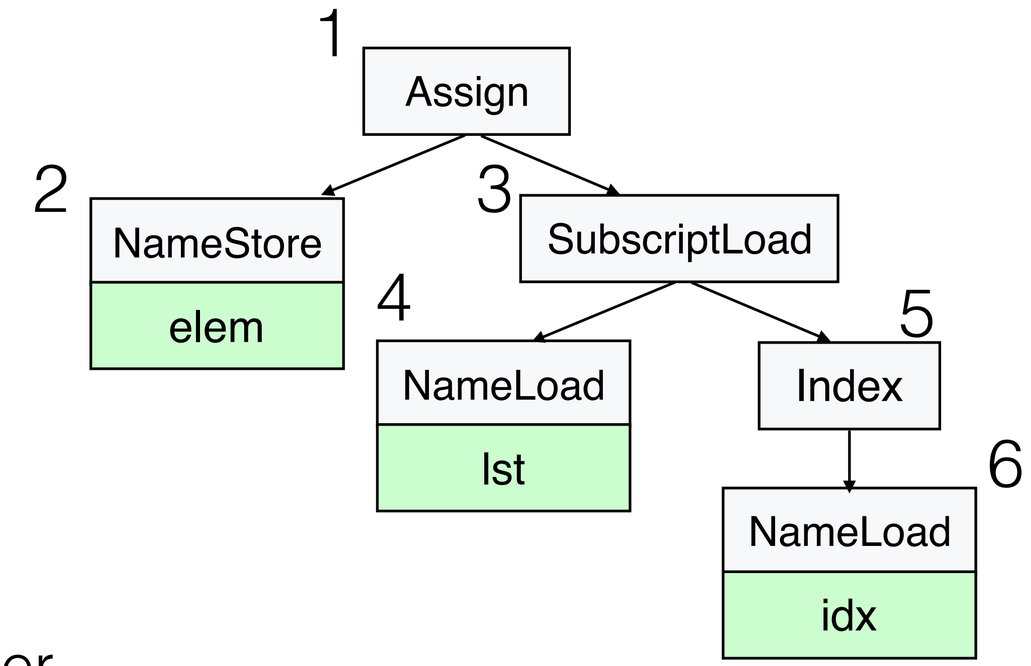
Output:
updated states
of all nodes

Arbitrary graph
structure supprted,
e.g. different edge types

Image from https://aifrenz.github.io/present_file/A%20gentle%20introduction%20to%20graph%20neural%20networks.pdf

GGNN-Sandwich



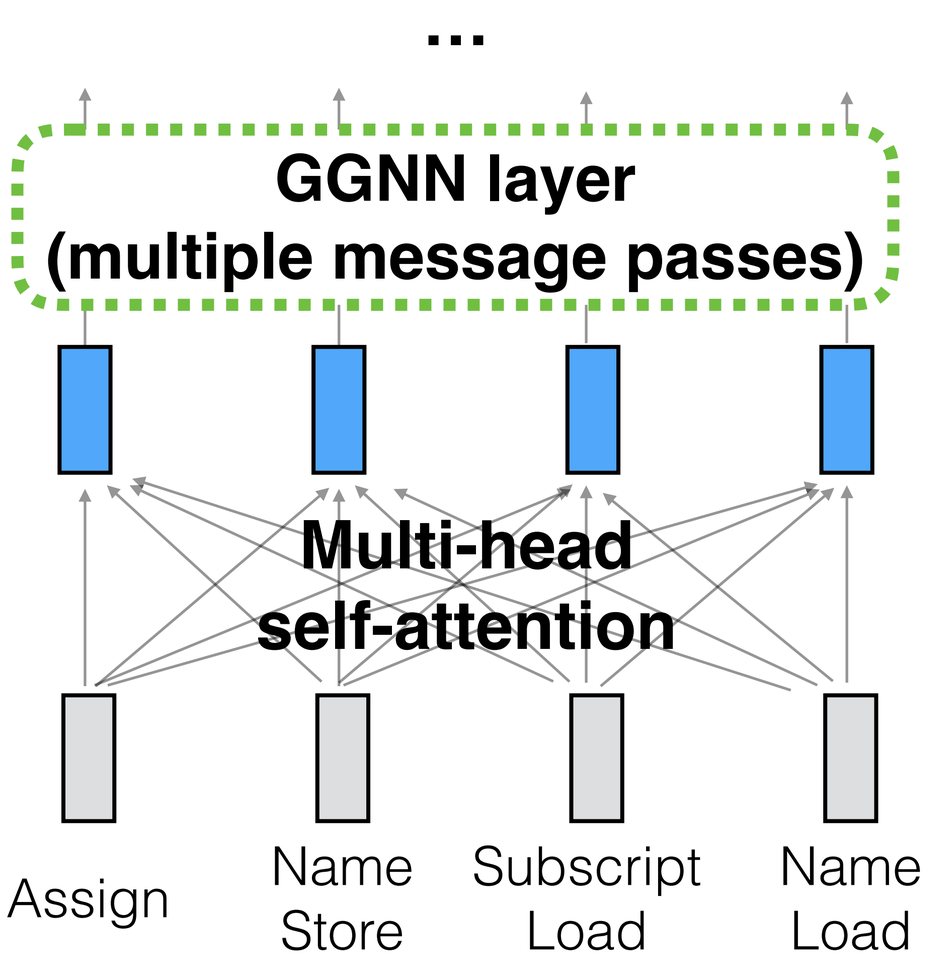


We consider
4 types
of edges:

- Parent (P)
- Child (C)
- Left (L)
- Right (R)

	1	2	3	4	5	6
1		P, L	P			
2	C, R		L			
3	C	R		P, L	P	
4			C, R		L	
5			C	R		P, L
6					C, R	

GGNN-Sandwich



Parameters:
same as n GGNN

Hyperparameters:
same as in GGNN
(e. g. the number
of message passes)

Hard to use
in Transformer decoder

Hellendoorn, V. J., Sutton, C., Singh, R., Maniatis, P., and Bieber, D. Global Relational Models of Source Code. ICLR20

Techniques for capturing AST structure in Transformer

- Sequential positional embeddings (+AST depth-first traversal)
- Sequential relative attention (+AST depth-first traversal)
- Tree positional encoding
- Tree relative attention
- GGNN-Sandwich

Techniques out-of-scope

- Recursive Transformer [1] — not a “global” model, slow
- Transformer with hierarchical accumulation [2] — applied only to texts, future work
- Inferring a tree for the input sequence [3] — other problem statement

[1] Mahtab Ahmed, Muhammad Rifayat Samee, Robert E. Mercer. You Only Need Attention to Traverse Trees, ACL19
[2] Xuan-Phi Nguyen, Shafiq Joty, Steven Hoi, Richard Socher. Tree-Structured Attention with Hierarchical Accumulation. ICLR20
[3] Yau-Shian Wang, Hung-Yi Lee, Yun-Nung Chen. Tree Transformer: Integrating Tree Structures into Self-Attention. EMNLP19

Overview

- Goal: understanding whether Transformers are able to utilize syntactic structure and what is the best approach to do it

- 5 syntax-capturing Transformer modifications

  3 tasks: code completion, bug fixing, function naming

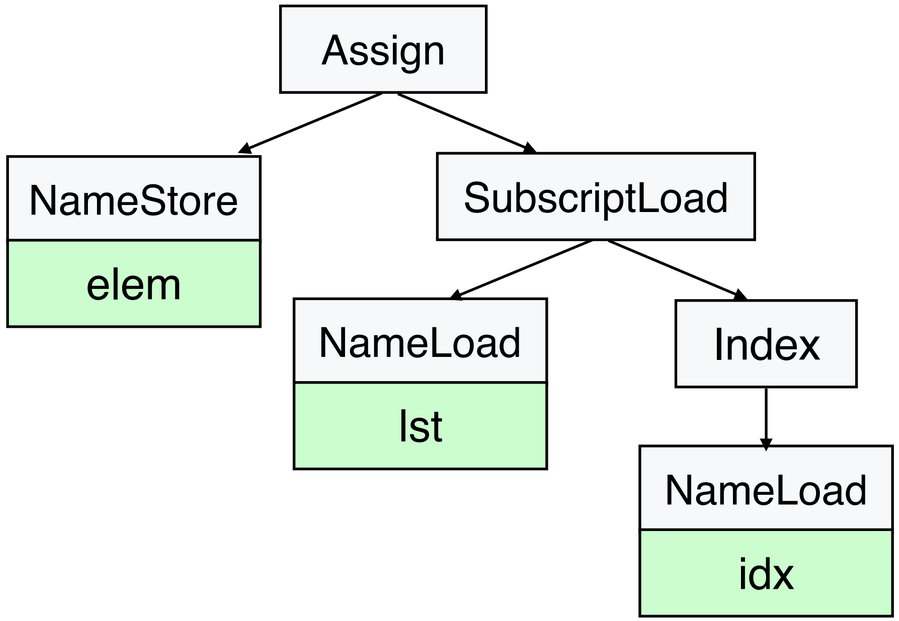
  2 datasets: Python150k and Javascript150k

- Thoughtful experimental setup:
  - repository-based train / test split
  - 2 settings: full data and anonymized data
  - re-implementing all techniques in a unified framework (e. g. same data preprocessing for all techniques)

Some technical details

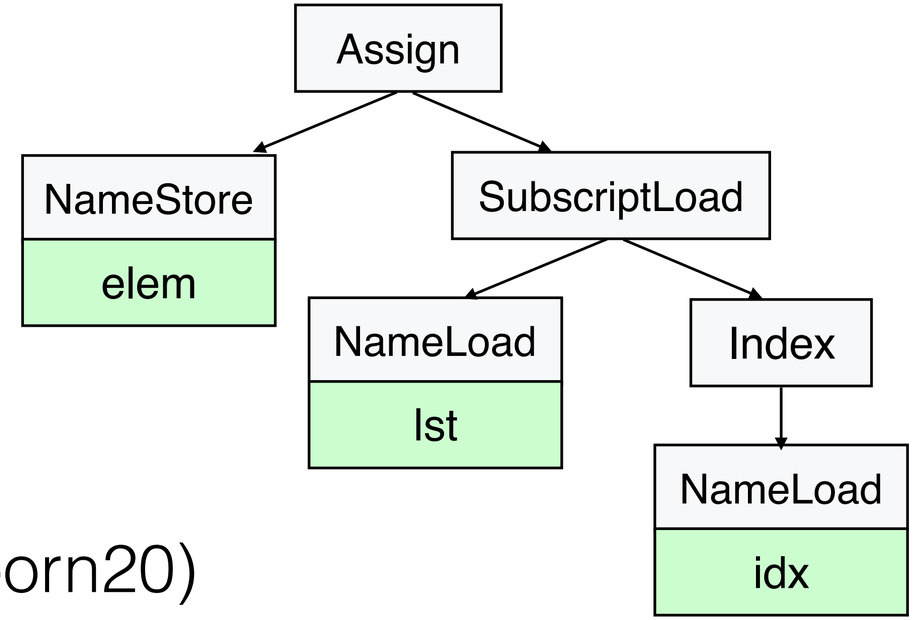
Each AST node contains type,
but not all nodes contain values

How to define type and value vocabularies?



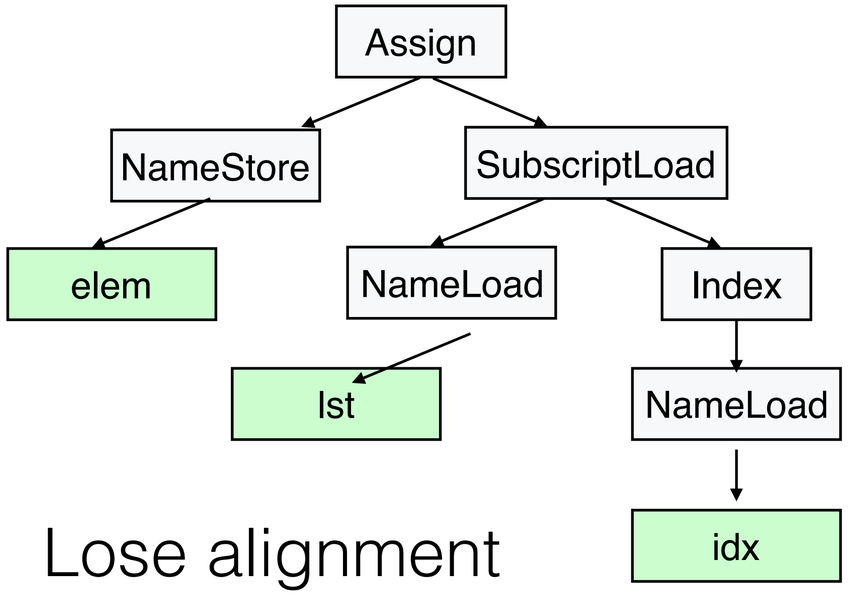
Some technical details

Each AST node contains type,
but not all nodes contain values



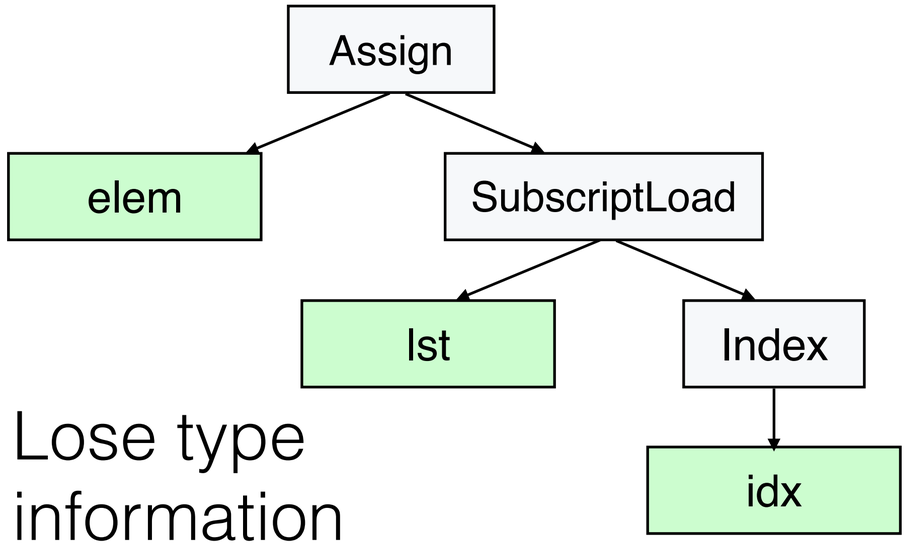
1 node — 1 token strategies:

(Kim20) Re-assign values:



- Lose alignment
- Longer sequences

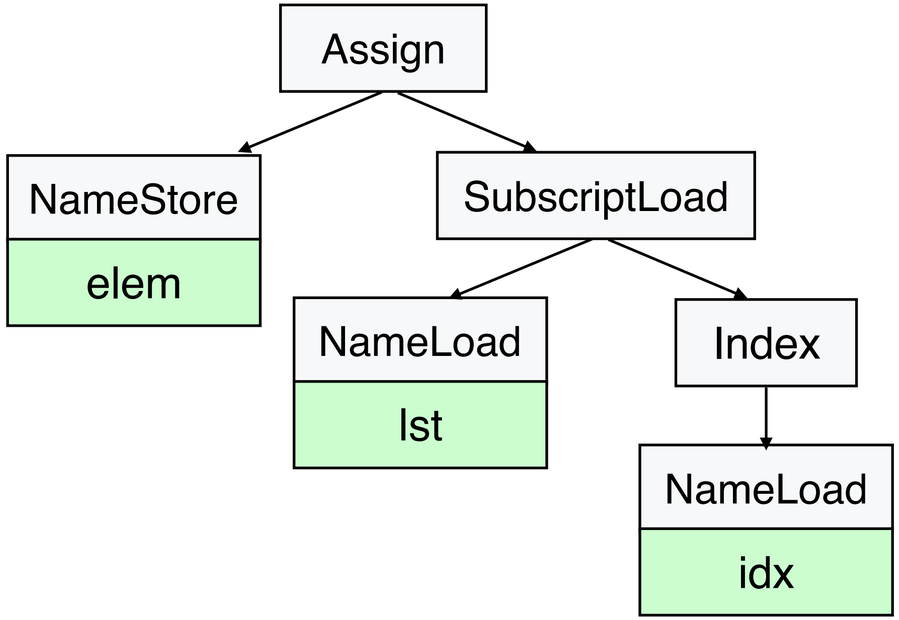
(Hellendoorn20)
Omit types:



- Lose type information

Some technical details

Each AST node contains type,
but not all nodes contain values



Our strategy (borrowed from (Li18)):

Use <empty> values, sum type embedding and value embedding:

Assign	NameStore	SubscriptLoad	...
⟨empty⟩	elem	⟨empty⟩	...

...	NameLoad	Index	NameLoad
...	lst	⟨empty⟩	idx

+ No information lost
+ Preserve input length
- Small redundancy

Some technical details

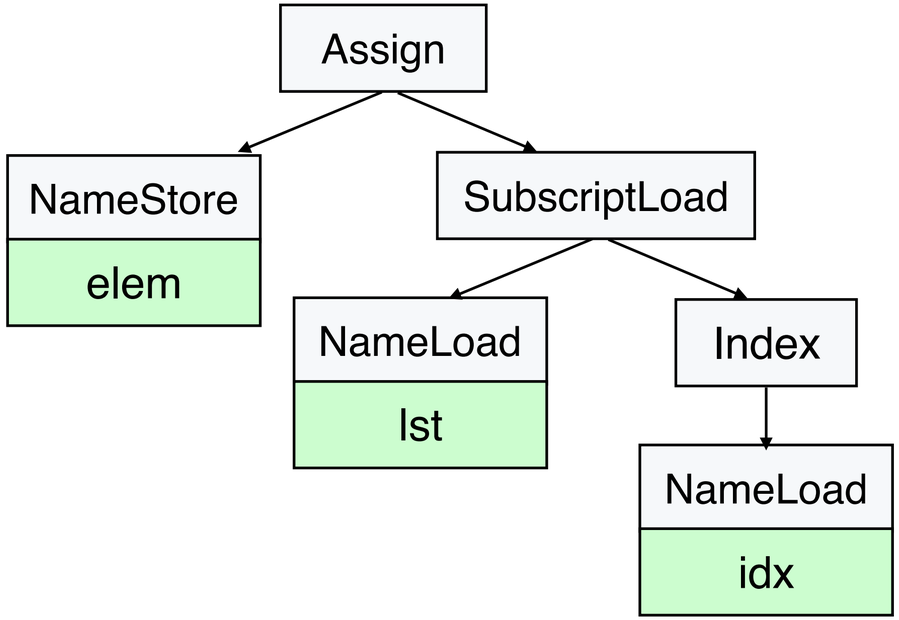
- Use <empty> values,
  sum type embedding and value embedding

- Vocabulary preprocessing:
  - for code: no token splitting

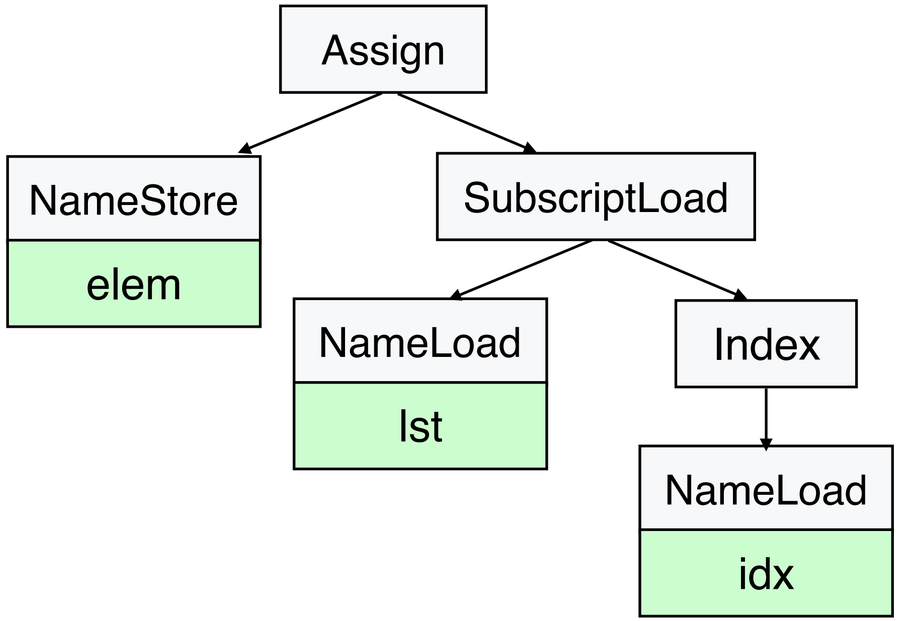
    `very_long_identifier` used “as is”

  - for function names: split by tokens

    `very long function name`



Some technical details



- Use <empty> values,
  sum type embedding and value embedding

- Vocabulary preprocessing:
  - for code: no token splitting

    `very_long_identifier` used “as is”

  - for function names: split by tokens

    `very long function name`

- Hyperparameters for tree pos. encoding, seq. rel. attention and GGNN-Sandwich were chosen using grid search

Overview

- Goal: understanding whether Transformers are able to utilize syntactic structure and what is the best approach to do it

- 5 syntax-capturing Transformer modifications

  3 tasks: code completion, bug fixing, function naming

  2 datasets: Python150k and Javascript150k

- Thoughtful experimental setup:
  - repository-based train / test split
  - 2 settings: full data and anonymized data
  - re-implementing all techniques in a unified framework (e. g. same data preprocessing for all techniques)

Tasks

Code completion:

```
import sys
from utils.parsing import parse
if __name__ == '__main__':
    argmnents = parse(
        sys.argv[1:]
    )
    print( next token?
```

Variable misuse

```
def count_lines(filename):
    with open(filename) as fin:
        return len(
            filename.read()
                .split("\n")
        )
```

A bug? Fix?

Function naming:

```
def <fun_name>(filename):
    with open(filename) as fin:
        return len(
            fin.read()
                .split("\n")
        )
```

Function name?

Tasks

Code completion:

```
import sys
from utils.parsing import parse
if __name__ == '__main__':
    argmnents = parse(
        sys.argv[1:]
    )
    print( argmnents
```

Kim, S., Zhao, J., Tian, Y., and Chandra, S. Code Prediction by Feeding Trees to Transformers. 2020

Variable misuse

```
def count_lines(filename):
    with open(filename) as fin:
        return len(
            filename.read()
                .split("\n")
        )
```

bug → filename

fix → fin

Function naming:

```
def <fun_name>(filename):
    with open(filename) as fin:
        return len(
            fin.read()
                .split("\n")
        )
```

count_lines

Hellendoorn, V. J., Sutton, C., Singh, R., Maniatis, P., and Bieber, D. Global Relational Models of Source Code. ICLR20

Tasks

Code completion:

```
import sys
from utils.parsing import parse
if __name__ == '__main__':
    argmnents = parse(
        sys.argv[1:]
    )
    print( argmnents
```

Decoder only

Variable misuse

```
def count_lines(filename):
    with open(filename) as fin:
        return len(
            filename.read()
                .split("\n")
    )
```

bug → `filename`

fix → `fin:`

Encoder only

Function naming:

```
def <fun_name>(filename):
    with open(filename) as fin:
        return len(
            fin.read()
                .split("\n")
    )
```

count_lines

Code encoder + Text decoder

Tasks: metrics

Code completion:

```
import sys
from utils.parsing import parse
if __name__ == '__main__':
    argmnents = parse(
        sys.argv[1:]
    )
    print( argmnents
```

MRR

Variable misuse

```
def count_lines(filename):
    with open(filename) as fin:
        return len(
            filename.read()
                .split("\n")
    )
```

bug → `filename` (fix → `fin`)

Joint loc. and repair accuracy

Function naming:

```
def <fun_name>(filename):
    with open(filename) as fin:
        return len(
            fin.read()
                .split("\n")
    )
```

count_lines

F-measure

Overview

- Goal: understanding whether Transformers are able to utilize syntactic structure and what is the best approach to do it

- 5 syntax-capturing Transformer modifications

  3 tasks: code completion, bug fixing, function naming

  2 datasets: Python150k and Javascript150k

- Thoughtful experimental setup:
  - repository-based train / test split
  - 2 settings: full data and anonymized data
  - re-implementing all techniques in a unified framework (e. g. same data preprocessing for all techniques)

Datasets

- Python150k / JavaScript150k (100K train files + 50K test files)

  (~10^8 train chars, ~$5 \cdot 10^7$ test chars)

- Custom train / test split:
  - repository-based
  - deduplication

Overview

- Goal: understanding whether Transformers are able to utilize syntactic structure and what is the best approach to do it

  - 5 syntax-capturing Transformer modifications

    3 tasks: code completion, bug fixing, function naming

    2 datasets: Python150k and Javascript150k

  - Thoughtful experimental setup:
    - repository-based train / test split
    - 2 settings: full data and anonymized data
    - re-implementing all techniques in a unified framework (e. g. same data preprocessing for all techniques)

What does this code snippet do?

```
var78 = []
for var5 in var12:
    var31 = []
    for var1 in var5:
        if var1 in var25:
            var31 += [var1]
    var78 += [var31]
```

What does this code snippet do?

```
var78 = []
for var5 in var12:
    var31 = []
    for var1 in var5:
        if var1 in var25:
            var31 += [var1]
    var78 += [var31]
```

```
filtered_seqs = []
for seq in seqs:
    new_seq = []
    for word in seq:
        if word in vocab:
            new_seq += [word]
    filtered_seqs += [new_seq]
```

Anonymized setting vs Full data setting

```
var78 = []
for var5 in var12:
    var31 = []
    for var1 in var5:
        if var1 in var25:
            var31 += [var1]
    var78 += [var31]
```

Only one source of information
— syntactic structure

var78, **var5**, **var12** etc
— anonymized identifiers

```
filtered_seqs = []
for seq in seqs:
    new_seq = []
    for word in seq:
        if word in vocab:
            new_seq += [word]
    filtered_seqs.append(new_seq)
```

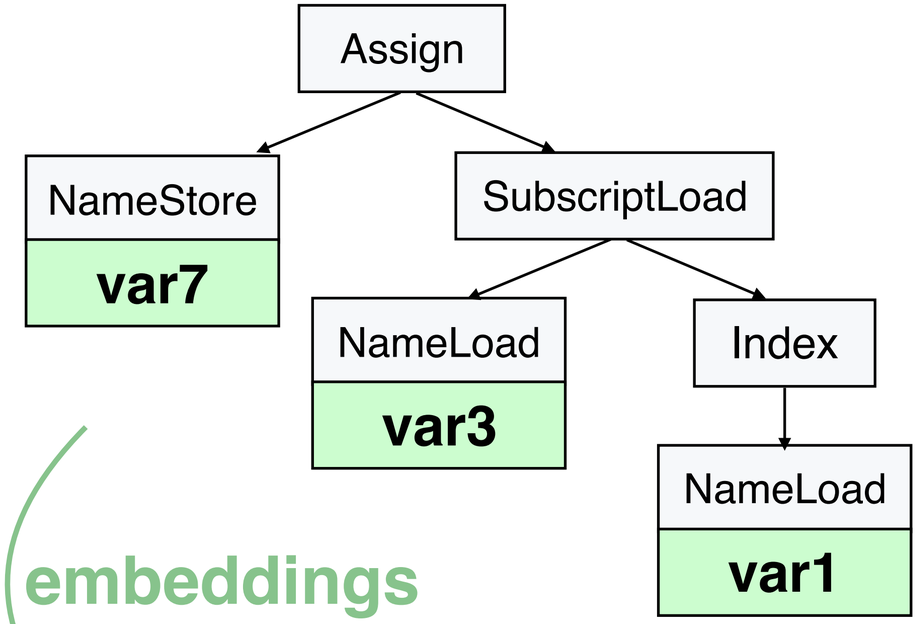
Two sources of information:

- Syntactic structure
- Identifier names (**seq**, **word** etc)

Anonymization procedure

```
var78 = []
for var5 in var12:
    var31 = []
    for var1 in var5:
        if var1 in var25:
            var31 += [var1]
    var78 += [var31]
```

parse AST



embeddings

Assign	<empty>
NameStore	var7
SubscriptLoad	<empty>
NameLoad	var3
Index	<empty>
NameLoad	var1

- Random subset of **var1** … **var1000**
- Save equality/inequality information
- An identifier may have different anonymizations in different input snippets

Overview

- Goal: understanding whether Transformers are able to utilize syntactic structure and what is the best approach to do it

- 5 syntax-capturing Transformer modifications

  3 tasks: code completion, bug fixing, function naming

  2 datasets: Python150k and Javascript150k

- Thoughtful experimental setup:
  - repository-based train / test split
  - 2 settings: full data and anonymized data
  - re-implementing all techniques in a unified framework (e. g. same data preprocessing for all techniques)

Research questions

- Are Transformers generally capable of utilizing syntactic information represented via AST?

- What components of AST (structure, node types and values) do Transformers use in different tasks?

- What is the most effective approach for utilizing AST *structure* in Transformer?

Capability of Transformers to utilize syntactic information

Syntax+Text* vs *Text only*

1	Assign	<empty>
2	NameStore	elem
3	SubscriptLoad	<empty>
4	NameLoad	lst
5	Index	<empty>
6	NameLoad	idx

1	elem
2	lst
3	idx

Does using AST in addition to identifier names improve the performance?

*Syntax** only vs *Dummy*

1	Assign	<empty>
2	NameStore	var7
3	SubscriptLoad	<empty>
4	NameLoad	var3
5	Index	<empty>
6	NameLoad	var1

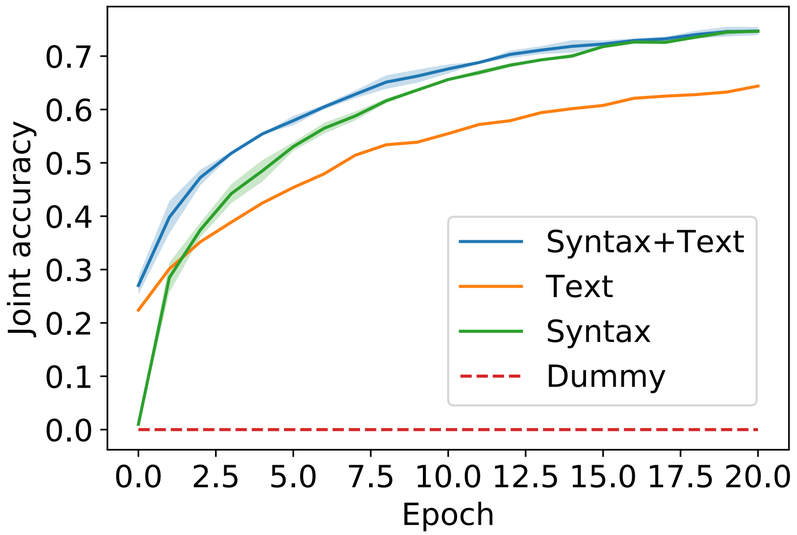
Most frequent *constant* prediction

Does using only AST perform any better than a Dummy baseline?

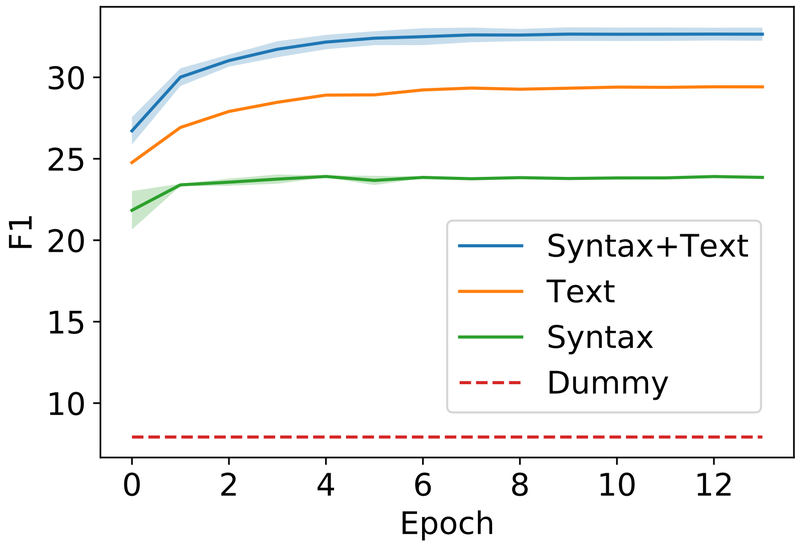
* AST depth-first traversal + standard sequential positional embeddings

Capability of Transformers to utilize syntactic information: results

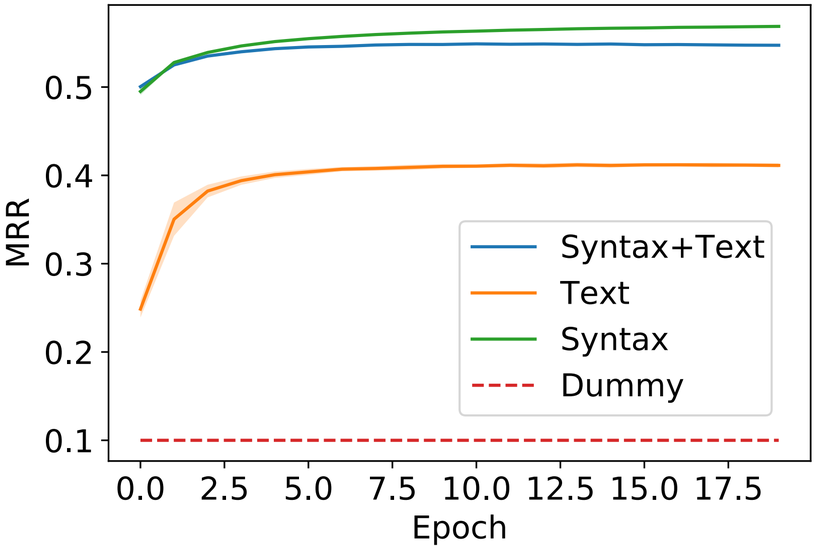
Variable misuse



Function naming



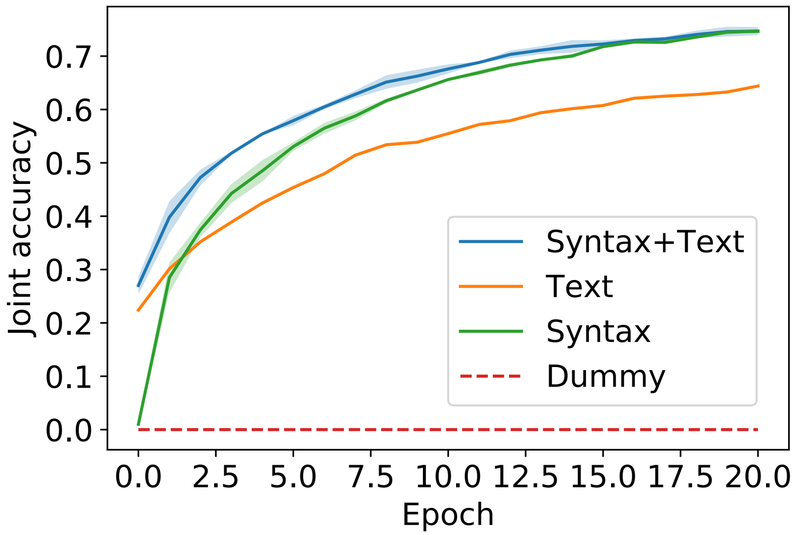
Code completion (values)



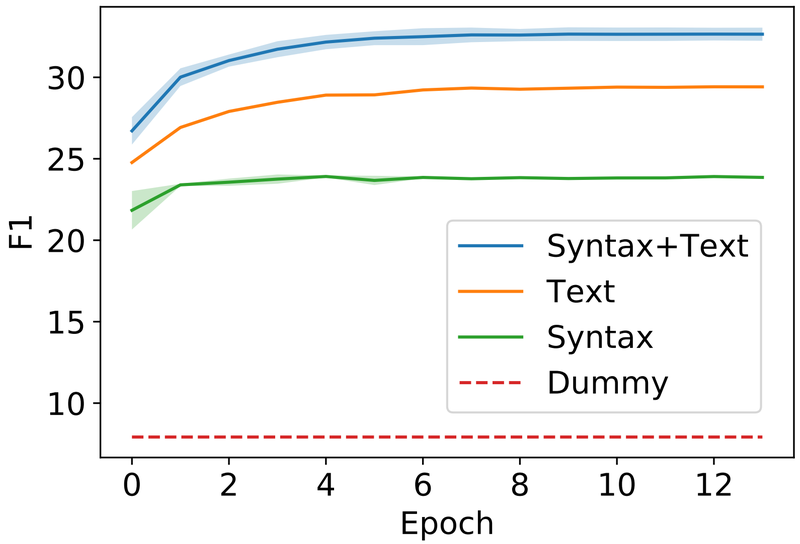
- *Syntax + Text* > *Text* only
- *Syntax* only > *Dummy*

Capability of Transformers to utilize syntactic information: results

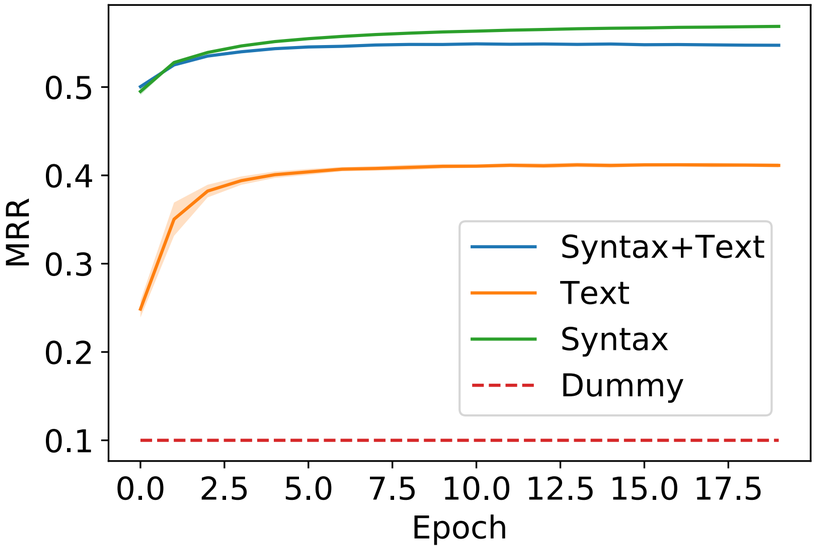
Variable misuse



Function naming



Code completion (values)



- *Syntax + Text* > *Text* only
- *Syntax* only > *Dummy*
- Interestingly, sometimes *Syntax* only ⩾ *Syntax + Text*

Code completion: demo

```
def get_or_create_bucket(s3_connection):
    bucket = s3_connection.get_bucket(
        settings.S3_BUCKET_NAME
    )
    if bucket is None:
        bucket = s3_connection.
```

(a) **S+T**: [create_bucket get_bucket get_key]
S: [get_bucket S3_BUCKET_NAME var107]
T: [bucket get_bucket settings]
Dummy: [self 0 None]
Gold: create_bucket

```
import sys
from utils.parsing import parse
if __name__ == '__main__':
    argmnents = parse(
        sys.argv[1:]
    )
    print(
```

(b) **S+T**: [Usage: done. \n]
S: [argmnents var440 var289]
T: [sys outfn len]
Dummy: [self 0 None]
Gold: argmnents

Function naming: demo

```
def <fun_name>(seqs):
    dt = {}
    for seq in seqs:
        for word in seq.split():
            if not word in dt:
                dt[word] = 1
            else:
                dt[word] += 1
    return dt
```

(c) **S+T**: get dict
S: get all
T: get split
Dummy: init
Gold: get dictionary

```
def <fun_name>(filename):
    with open(filename) as fin:
        return len(
            fin.read()
                .split("\n")
        )
```

(d) **S+T**: read file
S: read
T: read file
Dummy: init
Gold: count lines

Research questions

- Are Transformers generally capable of utilizing syntactic information represented via AST?

- What components of AST (structure, node types and values) do Transformers use in different tasks?

- What is the most effective approach for utilizing AST *structure* in Transformer?

Ablation study of mechanisms for utilizing syntactic information in Transformer

3 syntax-capturing mechanisms:

- types
- syntactic structure (AST edges)
- information about value equality/inequality

What mechanisms are essential for achieving the benefit in quality?

Ablation study of mechanisms for utilizing syntactic information in Transformer

3 syntax-capturing mechanisms:

- types
- syntactic structure (AST edges)
- information about value equality/inequality

(already ablated in the previous experiment)

a bag of (type, (an) value) pairs — turn off seq. pos. embeddings

Assign	<empty>
NameStore	var7 / elem
SubscriptLoad	<empty>
NameLoad	var3 / lst
Index	<empty>
NameLoad	var1 / idx

a sequence of types

1	Assign
2	NameStore
3	SubscriptLoad
4	NameLoad
5	Index
6	NameLoad

Ablation study of mechanisms for utilizing syntactic information in Transformer: results

	Full data			Anonymized data		
	Var. misuse	Fun. naming	Comp. (val.)	Var. misuse	Fun. naming	Comp. (val.)
Full AST	**74.59** %	**32.83**±0.35%	**54.89** ±0.07%	**74.71**%	**23.92** ±0.004%	**56.87** ±0.09%
AST w/o struct.	26.1%	**33.0** ±0.08%	54.2 ±0.1%	12.12%	23.28 ±0.02%	55.07 ±0.05%
AST w/o an.val.	N/A	N/A	N/A	32.1%	**24.00** ±0.15%	N/A

- Var. misuse, Code completion: all mechanisms important
- Function naming: mostly *types* contribute

Full AST

1	Assign	<empty>
2	NameStore	var7 / elem
3	SubscriptLoad	<empty>
4	NameLoad	var3 / lst
5	. Index	<empty>
6	NameLoad	var1 / idx

AST w/o struct.

Assign	<empty>
NameStore	var7 / elem
SubscriptLoad	<empty>
NameLoad	var3 / lst
Index	<empty>
NameLoad	var1 / idx

AST w/o an. val.

1	Assign
2	NameStore
3	SubscriptLoad
4	NameLoad
5	Index
6	NameLoad

Research questions

- Are Transformers generally capable of utilizing syntactic information represented via AST?

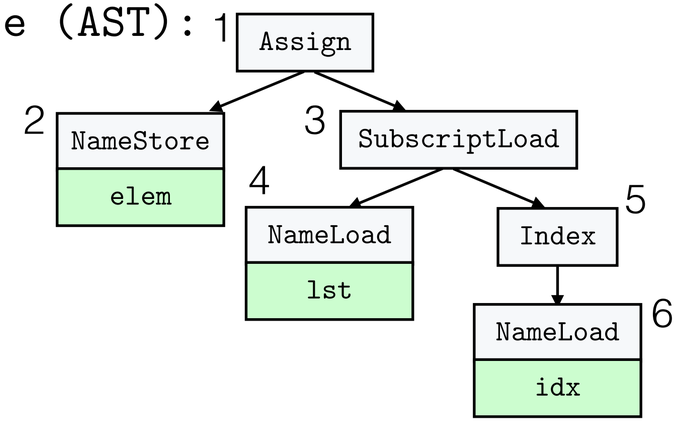
- What components of AST (structure, node types and values) do Transformers use in different tasks?

- What is the most effective approach for utilizing AST *structure* in Transformer?

Comparison of approaches for utilizing syntactic structure in Transformer

(a) Code: `elem = lst[idx]`

(b) Abstract syntax tree (AST):



(c) AST depth-first traversal:

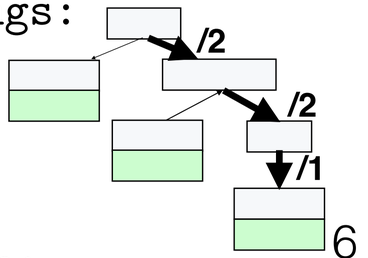
Assign	NameStore	SubscriptLoad	...
⟨empty⟩	elem	⟨empty⟩	...

...	NameLoad	Index	NameLoad
...	lst	⟨empty⟩	idx

- Sequential positional embeddings
- Sequential relative attention

(d) Tree positional encodings:

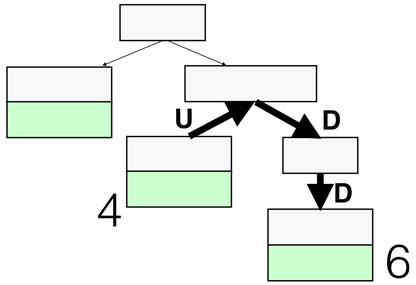
1	2	3	4	5	6
/	/1	/2	/2/1	/2/2	/2/2/1



stack-like enc.

000 000 000
100 000 000
010 000 000
100 010 000
010 010 000
100 010 010

(e) Tree relative attention:



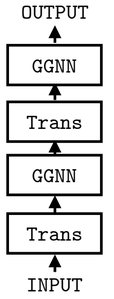
	1	2	3	4	5	6
1	I	D	D	DD	DD	DDD
2	U	I	UD	UDD	UDD	UDDD
3	U	UD	I	D	D	DD
4	UU	UUD	U	I	UD	UDD
5	UU	UUD	U	UD	I	D
6	UUU	UUUD	UU	UUD	U	I

(f) GGNN Sandwich:

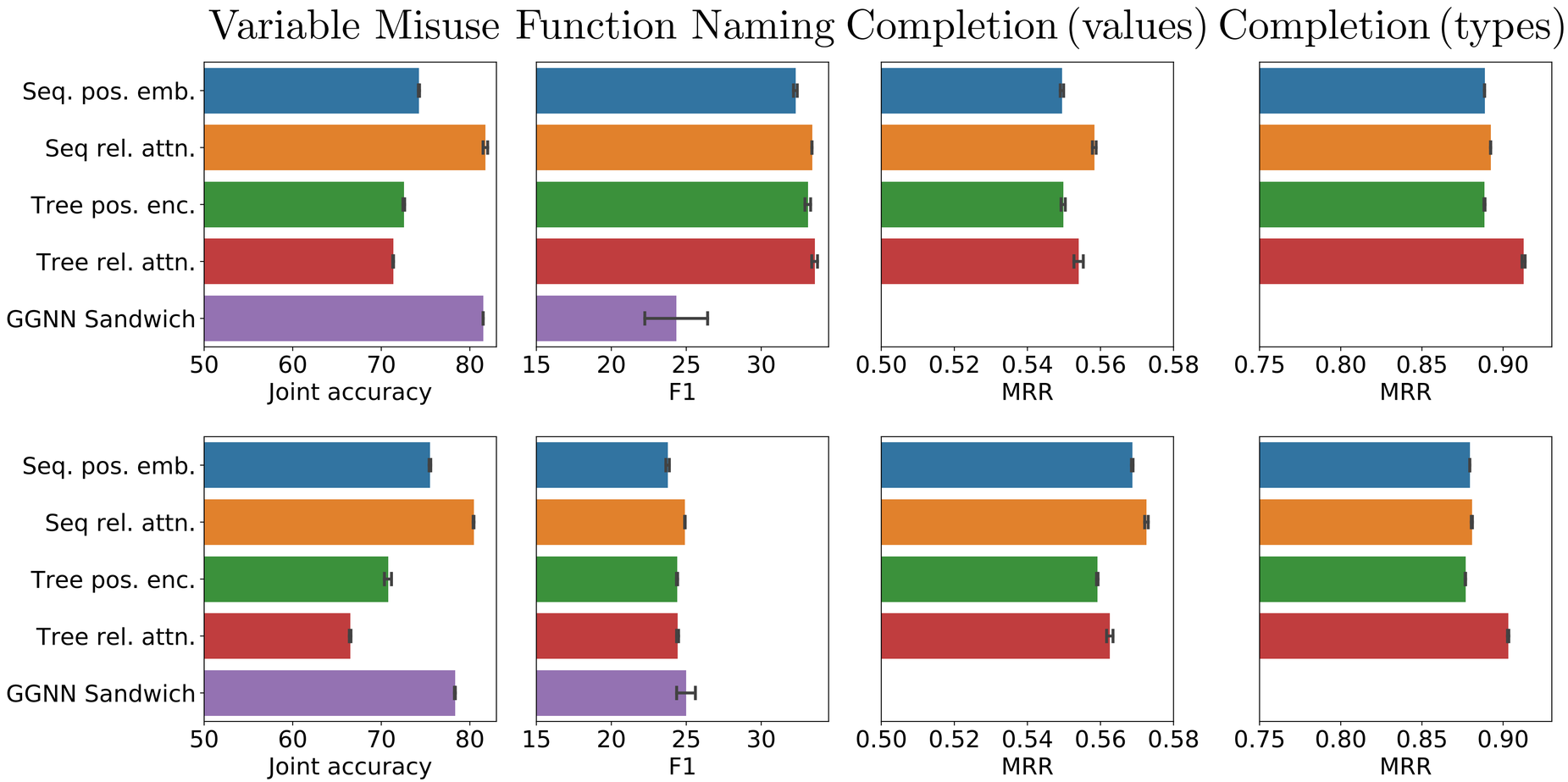
4 types of edges:

- Parent (P)
- Child (C)
- Left (L)
- Right (R)

	1	2	3	4	5	6
1		P, L	P			
2	C, R		L			
3	C	R		P, L	P	
4			C, R		L	
5			C	R		P, L
6					C, R	



Comparison of approaches for utilizing syntactic structure in Transformer: results



	Train time (h/ep)	Add. tr. data size
Seq. pos.	2.3	0
Tree. pos.	2.5	0.3
Seq. rel.	2.7	0
Tree rel.	3.9	18
GGNN sand.	7.2	0.35

- *Seq. rel. attn. the best overall*

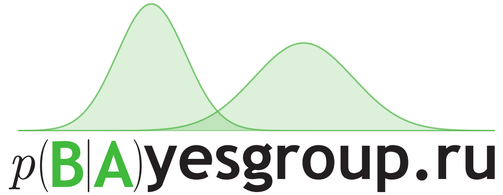
- *GGNN-Sandwich fine for VM*
- *Tree.rel. attn. fine for CC-types*

Combining AST-capturing techniques

Model	VM	FN	CC (val.)
SRA	81.54 %	33.38 %	55.79
SRA + Seq. pos. emb.	80.31 %	31.78 %	55.73
SRA + Tree rel. attn.	80.59 %	33.01 %	**56.26**
SRA + Tree pos. enc.	81.74 %	32.52 %	55.98
SRA + GGNN sand.	**84.14** %	27.67 %	N/A

- Other techniques may be beneficial in combination with sequential relative attention

Ensembling of syntax-based models



Models	VM	FN	CC (typ.)	CC (val.)
ST	81.54 %	**33.38**%	89.1 %	55.79 %
ST & ST	83.15 %	32.57 %	**89.4** %	56.86 %
ST & S	**85.73**%	29.42%	88.51 %	**59.87** %

- Ensembling *Syntax+Text* and *Syntax* outperforms ensembling two *Syntax+Text* (standard) models in several tasks

ST: *Syntax* + *Text*
S : *Syntax* only
& denotes ensembling

Summary

- Using syntactic structure in Transformer increases the quality in all three tasks. However, in the function naming task, mostly type information contributes to the increase, while syntax itself does not.

- Sequential relative attention outperforms other techniques in almost all cases, with less computational overhead.
  This baseline is missing in a lot of works.

- Anonymization increases the quality of the ensemble
  and sometimes increases even the quality of a single model.

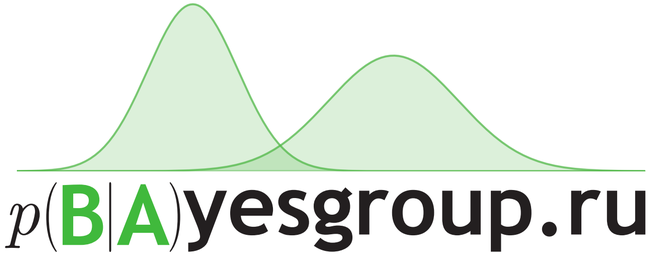
A Simple Approach for Handling Out-of-Vocabulary Identifiers in Deep Learning for Source Code

Nadezhda Chirkova*, Sergey Troshin*

Higher School of Economics, Samsung-HSE Laboratory
Moscow, Russia

NATIONAL RESEARCH UNIVERSITY

SAMSUNG Research



Motivation

Advantages of identifier anonymization:

- Uniform token frequencies — dealing with rare or misspelled identifiers
- Other form of same data — beneficial for ensembling, (possibly) regularization

```
var78 = []
for var5 in var12:
    var31 = []
    for var1 in var5:
        if var1 in var25:
            var31 += [var1]
    var78 += [var31]
```

- Disadvantage: losing semantic information stored in variable names

Motivation

Advantages of identifier anonymization:

- Uniform token frequencies — dealing with rare or misspelled identifiers
- Other form of same data — beneficial for ensembling, (possibly) regularization

```
var78 = []
for var5 in var12:
    var31 = []
    for var1 in var5:
        if var1 in var25:
            var31 += [var1]
    var78 += [var31]
```

- Disadvantage: losing semantic information stored in variable names

Proposed method

Preprocessing steps:

1. Select the vocabulary of frequent identifiers, e. g. 50K
2. Anonymize out-of-vocabulary identifiers

Vocabulary: {`np`, `sin`}

Input:

```
my_y  = np.sin(my_x) + my_x
```

Standard OOV processing procedure:

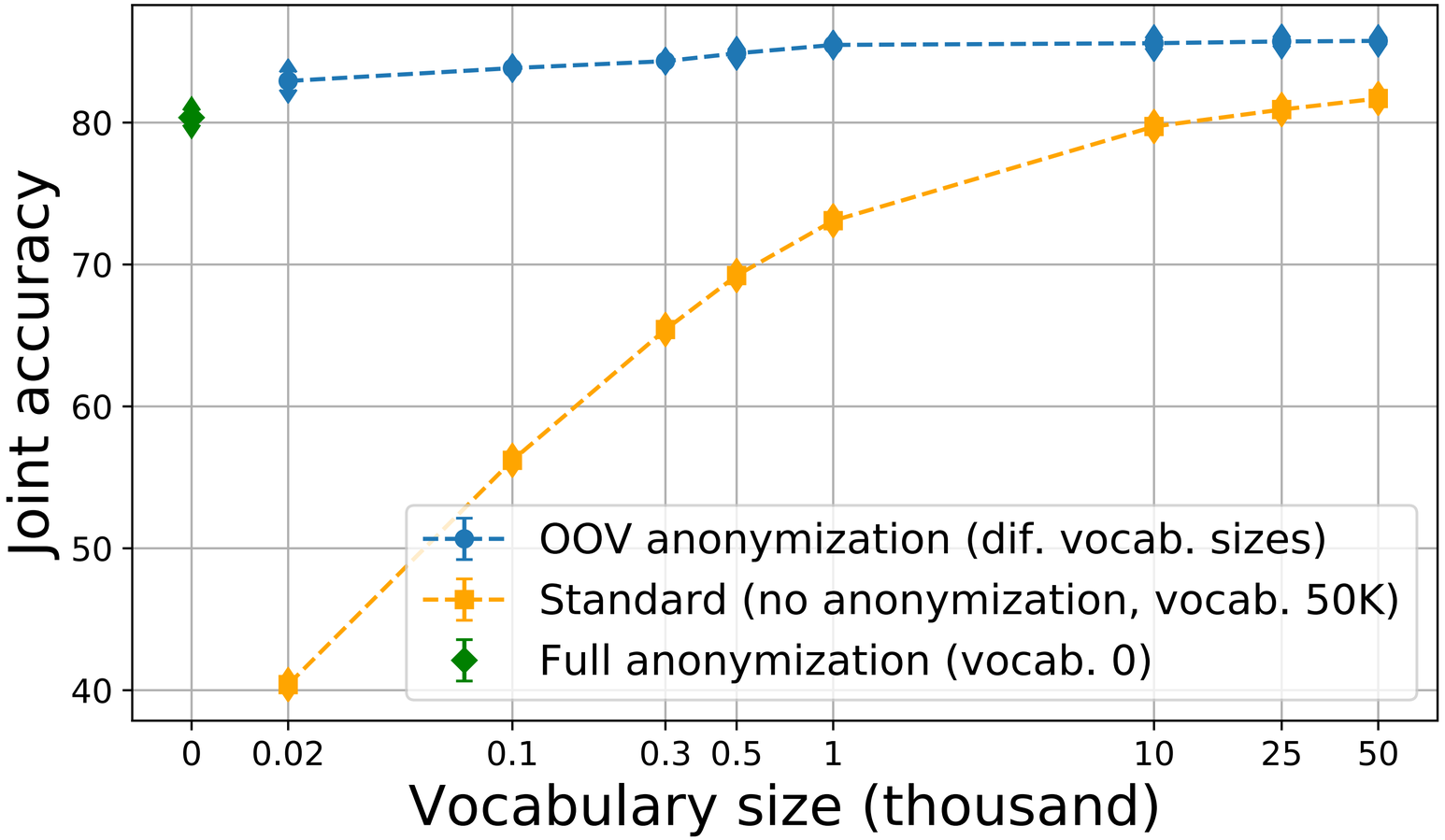
```
UNK = np.sin(UNK) + UNK
```

Proposed OOV anonymization procedure:

```
VAR1 = np.sin(VAR2) + VAR2
```

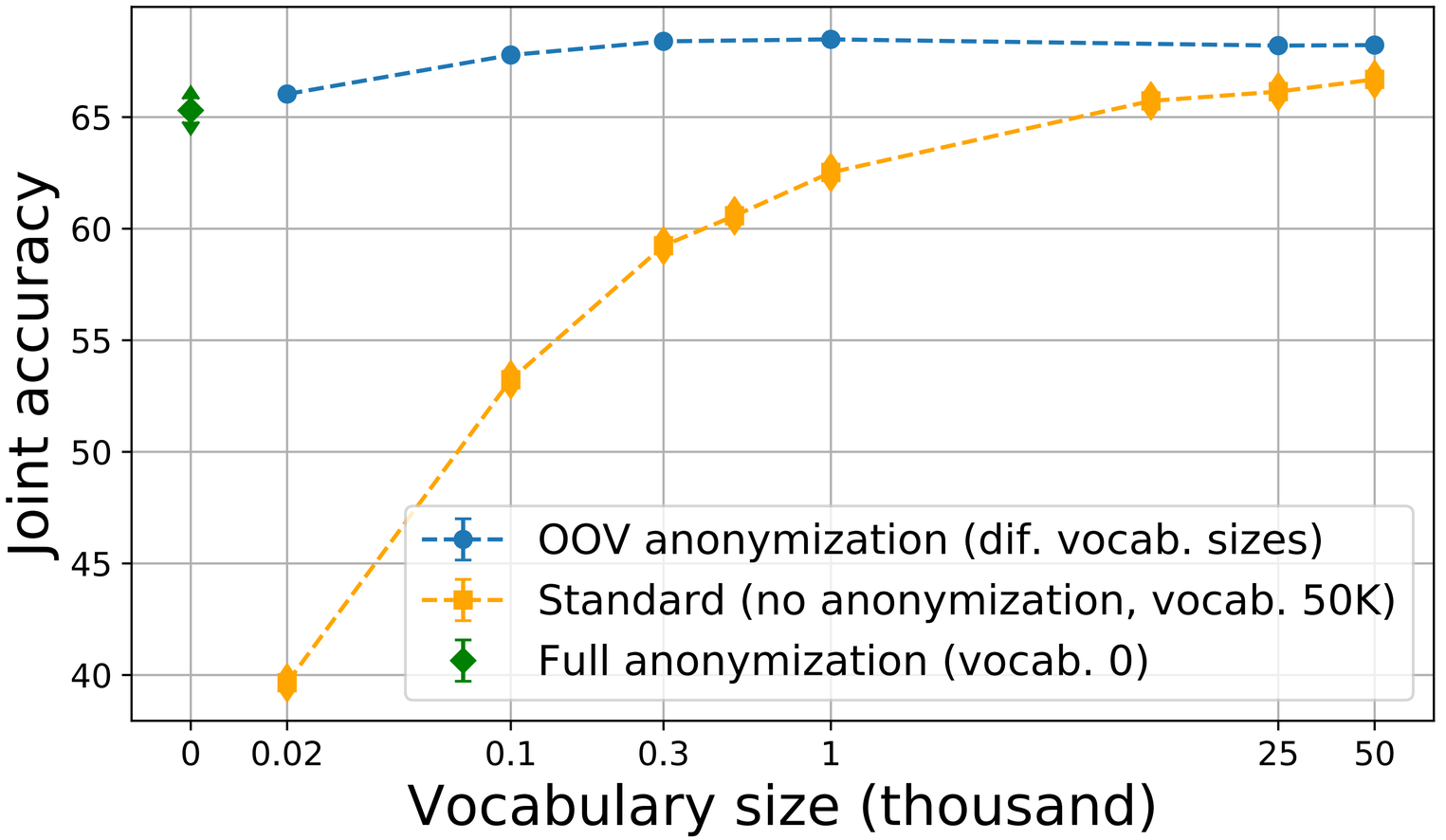
Results: variable misuse task, Python150k

Same setting as below, sequential relative attention

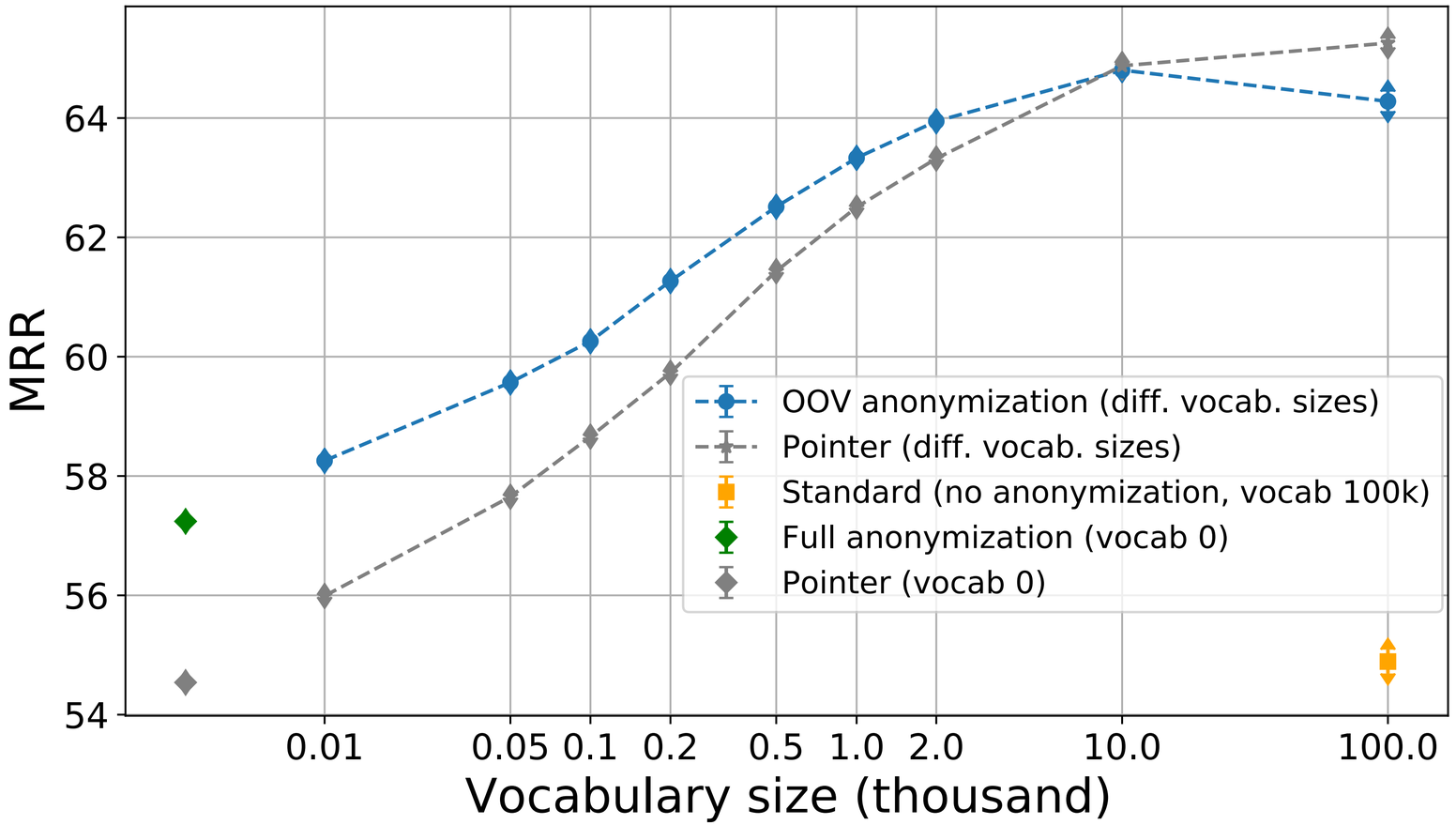


Results: variable misuse task, Javascript150k

Same setting as below, sequential relative attention

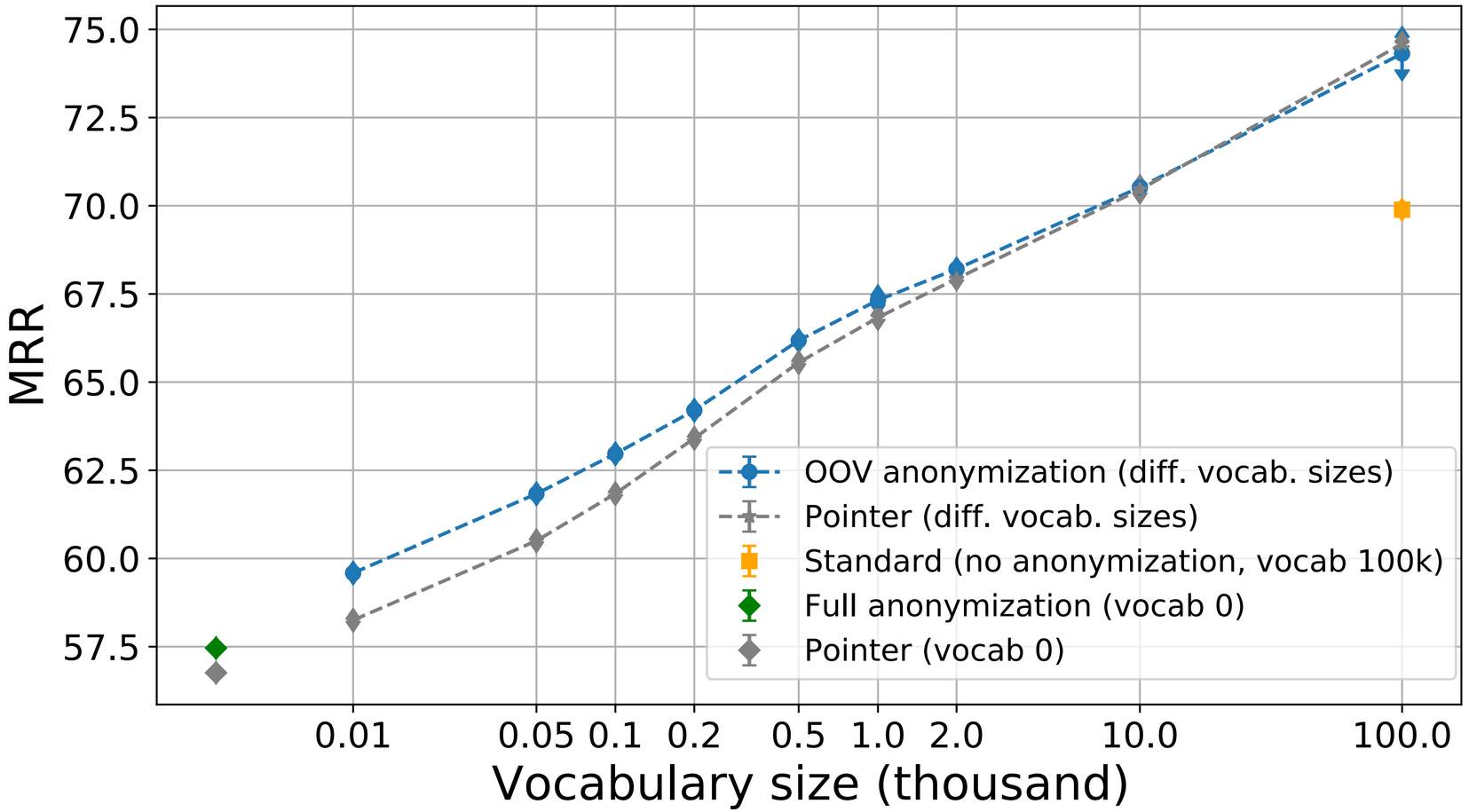


Results: code completion task, Python150k



The proposed OOV anonymization is much simpler in implementation than the pointer mechanism

Results: code completion task, Javascript150k



The proposed OOV anonymization is much simpler in implementation than the pointer mechanism

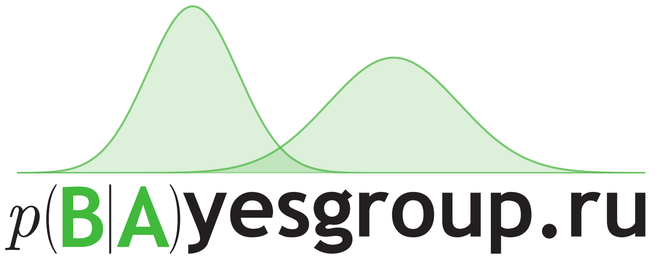
Neural Code Completion with Anonymized Variable Names

Nadezhda Chirkova

Higher School of Economics, Samsung-HSE Laboratory
Moscow, Russia

NATIONAL RESEARCH UNIVERSITY

SAMSUNG Research



Motivation

- Two different anonymizations result in (formally) different predictions:

Source:

```
elem = lst[idx]
```

Anonymization 1:

```
var87 = var34[var7]
```

embedding of `var87` | embedding of `var34` | embedding of `var7`

Anonymization 2:

```
var34 = var11[var3]
```

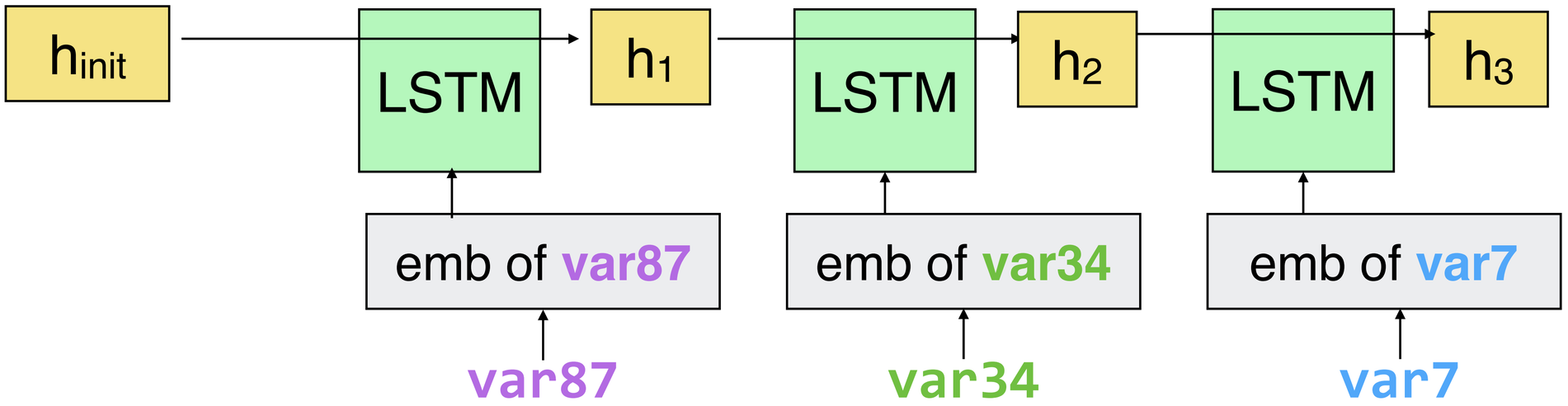
embedding of `var34` | embedding of `var11` | embedding of `var3`

- OK for Transformers, not OK for RNNs

Goal

- Goal: to develop RNN model that is invariant to identifier renaming,
  i. e. renaming all the identifiers does not change the model's prediction

- Baseline: static embeddings (conventional embedding layer)
  — does not fulfill the invariance property



Goal

- Goal: to develop RNN model that is invariant to identifier renaming,
  i. e. renaming all the identifiers does not change the model's prediction

- Baseline: static embeddings (conventional embedding layer)
  — does not fulfill the invariance property

- Our model: dynamic embeddings to replace the conventional embedding layer

Dynamic embeddings: general idea

Initialize the embedding of each variable with $DE_{initial}$, update the embedding after each use of the variable

```
var78 = []
for var5 in var12:
    var31 = []
    for var1 in var5:
        if var1 in var25:
            var31 += [var1]
    var78 += [var31]
```

Input	Dynamic embeddings	Hidden state
		$H_0 = H_{init}$
var5	$DE_{var5} = DE_{initial}$	$H_1 = UpdH(DE_{var5}, H_0)$
var12	$DE_{var12} = DE_{initial}$	$H_2 = UpdH(DE_{var12}, H_1)$
var31	$DE_{var31} = DE_{initial}$	$H_3 = UpdH(DE_{var31}, H_2)$
var1	$DE_{var1} = DE_{initial}$	$H_4 = UpdH(DE_{var1}, H_3)$
var5	$DE_{var5} = UpdDE(H_4, DE_{var5})$	$H_5 = UpdH(DE_{var5}, H_4)$
var1	$DE_{var1} = UpdDE(H_5, DE_{var1})$	$H_6 = UpdH(DE_{var1}, H_5)$
var25	$DE_{var25} = DE_{initial}$	$H_7 = UpdH(DE_{var25}, H_6)$
…	…	…

Dynamic embeddings: algorithm

Input data:

$[(t_1, \tilde{v}_1), \ldots, (t_L, \tilde{v}_L)]$

— sequence of
(type, anon. value) pairs

1	`Assign`	`<empty>`
2	`NameStore`	`var7`
3	`SubscriptLoad`	`<empty>`
4	`NameLoad`	`var3`
5	`Index`	`<empty>`
6	`NameLoad`	`var1`

Dynamic embeddings: algorithm

Input data:

$[(t_1, \tilde{v}_1), \ldots, (t_L, \tilde{v}_L)]$

— sequence of
(type, anon. value) pairs

Notation:

$e_{\tilde{v}_i, j}$ — dyn. emb. of an.value $\tilde{v}_i$ at timestep j

e_{t_i} — (static) emb. of type t_i

h_i — hidden state as timestep i

Initialization

Dyn. embs.: $e_{\tilde{v}_i, 0} = e_{\text{init}}, \quad i = 1, \ldots, L$

RNN hidden state: $h_0 = h_{\text{init}}$

Dynamic embeddings: algorithm

Input data:

$[(t_1, \tilde{v}_1), \ldots, (t_L, \tilde{v}_L)]$

— sequence of
(type, anon. value) pairs

Notation:

$e_{\tilde{v}_i, j}$ — dyn. emb. of an.value $\tilde{v}_i$ at timestep j

e_{t_i} — (static) emb. of type t_i

h_i — hidden state as timestep i

One timestep $i-1 \to i$:

Update current dyn. emb.: $e_{\tilde{v}_i, i} = \mathrm{LSTM}_{\mathrm{dyn}}(h_{i-1}, e_{t_i} \ ; \ e_{\tilde{v}_i, i-1})$

Other dyn. embs. stay the same: $e_{\tilde{v}, i} = e_{\tilde{v}, i-1}, \quad \tilde{v} \neq \tilde{v}_i$

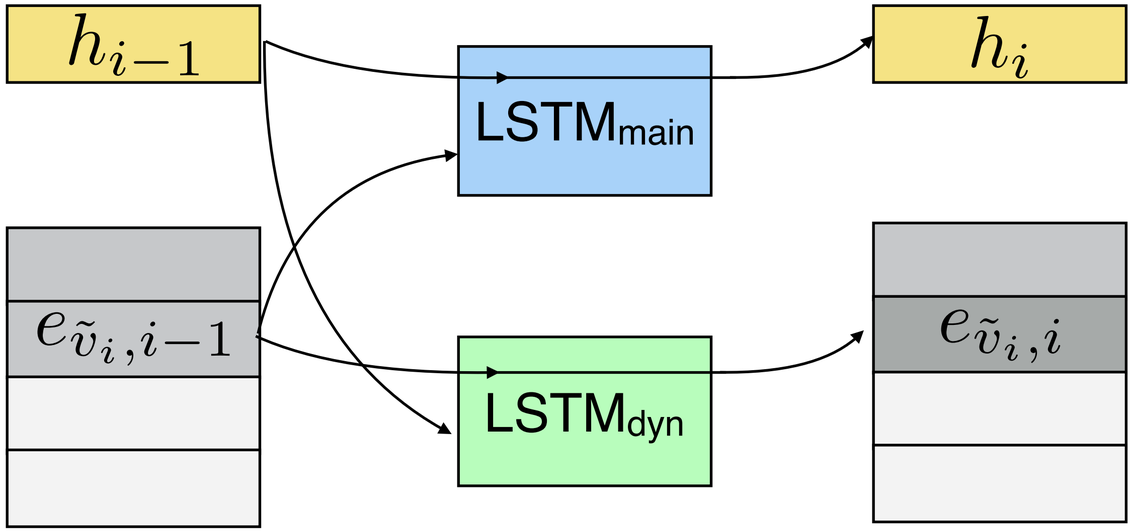
Update RNN state: $h_i = \mathrm{LSTM}_{\mathrm{main}}(e_{\tilde{v}_i, i-1}, e_{t_i} \ ; \ h_{i-1})$

Dynamic embeddings: algorithm

Input data:

$[(t_1, \tilde{v}_1), \ldots, (t_L, \tilde{v}_L)]$

— sequence of
(type, anon. value) pairs



One timestep $i-1 \to i$:

Update current dyn. emb.: $e_{\tilde{v}_i,i} = \mathrm{LSTM}_{\mathrm{dyn}}(h_{i-1}, e_{t_i} \ ; \ e_{\tilde{v}_i,i-1})$

Other dyn. embs. stay the same: $e_{\tilde{v},i} = e_{\tilde{v},i-1}, \quad \tilde{v} \neq \tilde{v}_i$

Update RNN state: $h_i = \mathrm{LSTM}_{\mathrm{main}}(e_{\tilde{v}_i,i-1}, e_{t_i} \ ; \ h_{i-1})$

Results: code completion, anonymized setting

3 variants of base architecture: plain LSTM, LSTM+attention, LSTM+pointer

2 datasets: Python150k (Py) and Javascript150k (JS)

Metric: accuracy

Setup of (Li18) + custom train / test split

Type embedding: 300

Value embedding: 1200 (standard, stat. emb)
500 (dyn. emb.)

Model	**Py:**	**LSTM**	**LSTM+at**	**LSTM+pt**	**JS:**	**LSTM**	**LSTM+at**	**LSTM+pt**
W/o vars.		47.34	47.33	60.65		45.22	45.21	58.80
Stat. emb.		55.76	59.74	60.28		51.80	56.26	57.67
Dyn. emb.		**66.35**	**66.79**	**66.90**		**61.69**	**62.86**	**62.85**
Standard		61.62	63.73	64.69		62.03	64.28	65.05

Jian Li, Yue Wang, Michael R. Lyu, Irwin King. Code Completion with Neural Attention and Pointer Networks. IJCAI18

Results: code completion, both settings

3 variants of base architecture: plain LSTM, LSTM+attention, LSTM+pointer

2 datasets: Python150k (Py) and Javascript150k (JS)

Metric: accuracy

Setup of (Li18) + custom train / test split

Type embedding: 300

Value embedding: 1200 (standard, stat. emb)
500 (dyn. emb., mixed emb.)

Model	**Py:**	**LSTM**	**LSTM+at**	**LSTM+pt**	**JS:**	**LSTM**	**LSTM+at**	**LSTM+pt**
W/o vars.		47.34	47.33	60.65		45.22	45.21	58.80
Stat. emb.		55.76	59.74	60.28		51.80	56.26	57.67
Dyn. emb.		**66.35**	**66.79**	**66.90**		**61.69**	**62.86**	**62.85**
Standard		61.62	63.73	64.69		62.03	64.28	65.05
Mixed. emb.		**67.18**	N/A	**68.61**		**63.96**	N/A	**65.56**

Results: code completion, OOV anonymization

Standard embeddings for in-vocabulary identifiers
and dynamic embeddings for OOV identifiers

Type embedding: 300

Value embedding: 1200 (standard)
500 (dyn. emb. — OOV)

Model	Py	JS
Standard	64.69	65.05
Proposed	**66.01**	**65.26**

Vocabulary: {`np`, `sin`}

Input:

```
my_y  = np.sin(my_x) + my_x
```

OOV anonymization:

```
VAR1 = np.sin(VAR2) + VAR2
```

Results: variable misuse, anonymized setting

Base architecture: bidirectional LSTM (dyn. emb. are also bidirectional)

2 datasets: Python150k (Py) and Javascript150k (JS)

Metric: joint loc. and repair accuracy

Type embedding: 300 Value embedding: 1200 (standard, stat. emb)
500 (dyn. emb)

Model	**Py**	**JS**
Stat. emb.	24.20	23.89
Dyn. emb.	**54.39**	**46.38**
Standard	49.38	41.59

Summary

- Dynamic embeddings: invariant to any renaming identifiers
- Dynamic embeddings significantly outperform static embeddings in the anonymized setting
- Dynamic embeddings in the full data setting: mixed embeddings outperform standard model; dynamic embeddings help to process OOV identifiers more effectively