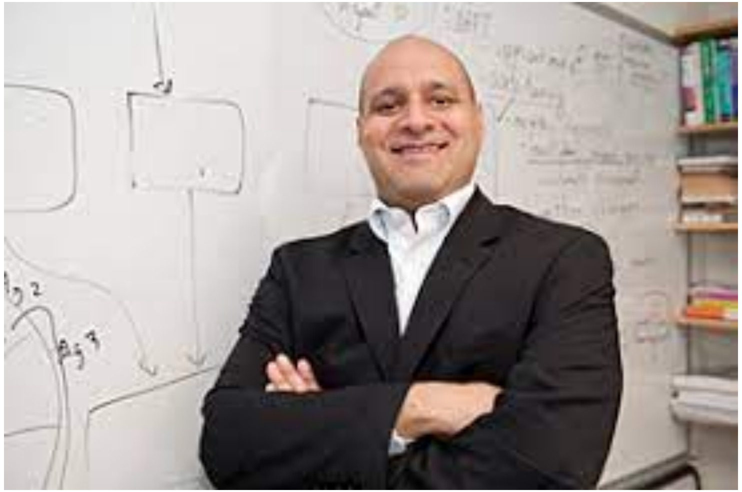
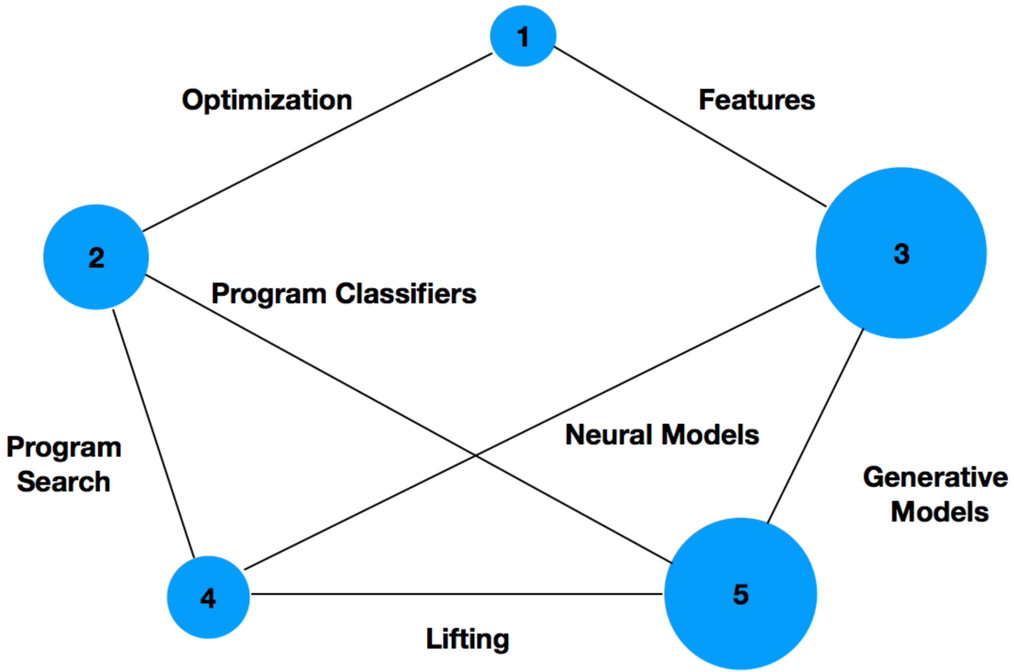
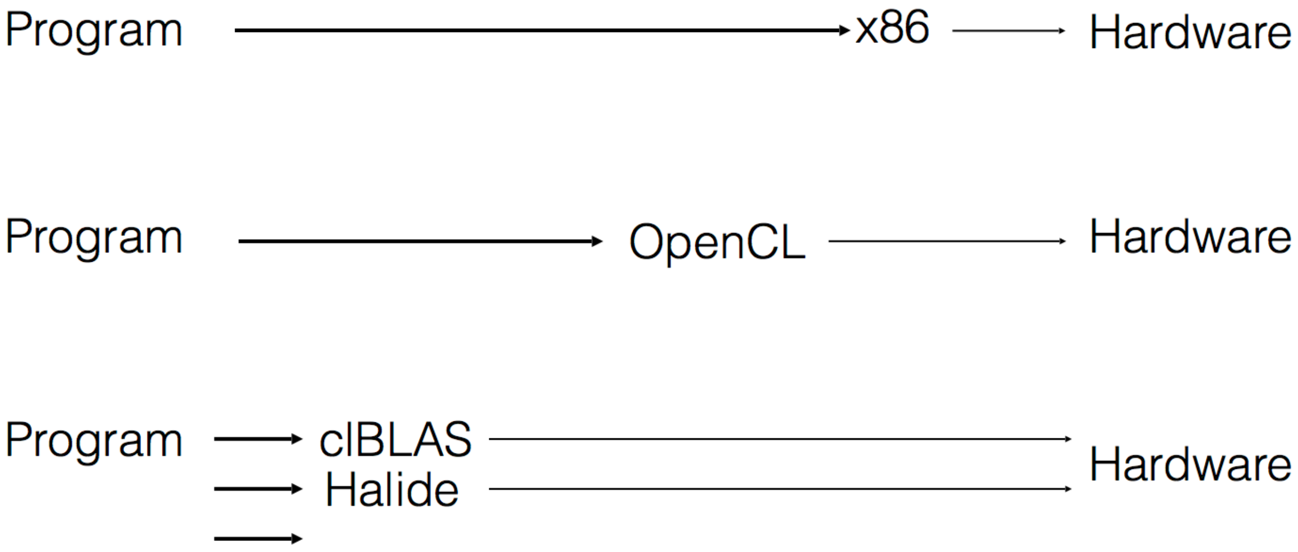
Rethinking Compilation: L5




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


Program
x86
Hardware
Program
OpenCL
Hardware
Program
clBLAS
Halide
Hardware

Overview

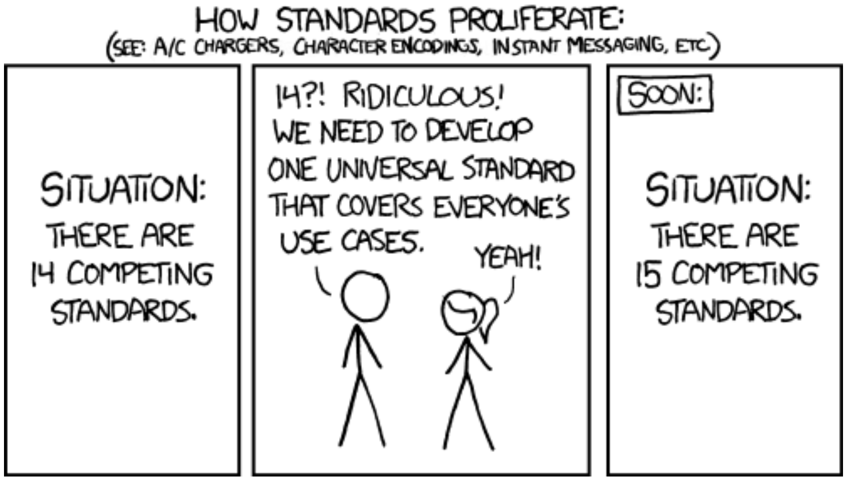
- L1: Motivation and survey of auto-tuning/machine learning for compilers
- L2: Program rewriting schemes - e-graphs and equality saturation
- L3: Program embeddings and Graph Neural Networks
- L4: Program synthesis and neural synthesis
- **L5: Neural Machine Translation,Transformers and Large language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

Lecture Structure

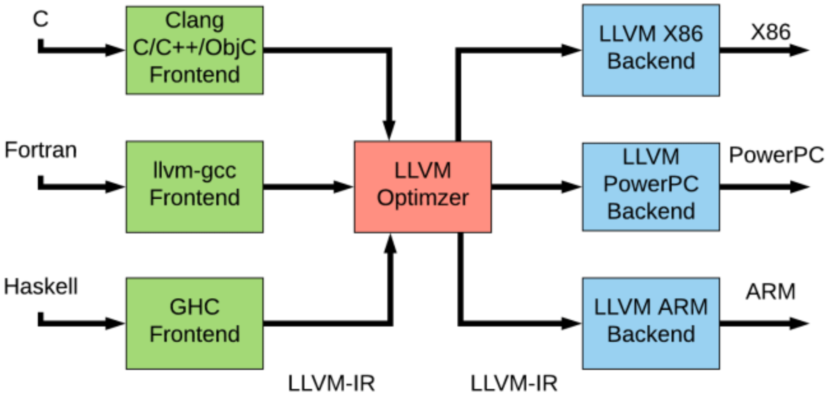
1. The Tower of Babel of Programming Languages
2. Machine Translation of Programming Languages
3. Transformers
4. Unsupervised Translation
5. Translation validation
6. What’s next

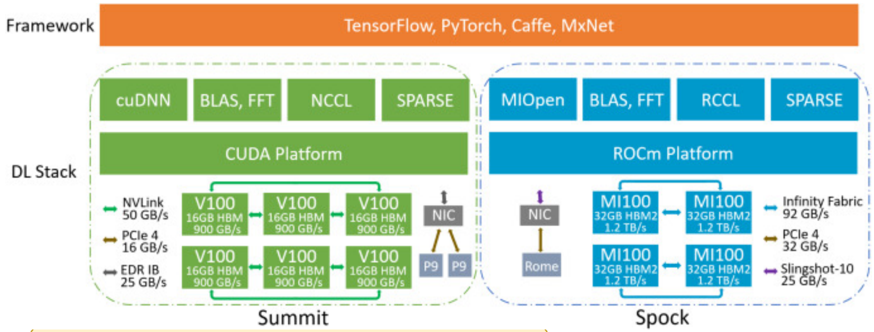
Ultimate language/compiler/backend library to rule them all!

https://xkcd.com/927/



The Tower of Babel of Programming Languages





GPU
- Medium-to-large models, datasets
- Image, video processing
- Application on CUDA or OpenCL

TPU
- Matrix computations
- Dense vector processing
- No custom TensorFlow operations

FPGA
- Large datasets, models
- Compute intensive applications
- High performance, high perf./cost ratio

The Tower of Babel of Programming languages: Automation?

A: Translate/rewrite

- generate x86 code from LLVM IR , or remove dead code

B: Decide when to rewrite given (A)

- Vectorize a loop.

Large potential for automation. ML in compilers used in B, but A

Replacing heuristics is an easy, controllable win for ML

Generating code is way more **challenging** and **dangerous**!

Ways forward to translation automation

A. Compilers, developer tools.
B. Program synthesis.
C. Proof automation, proof repair.
D. **Machine translation**?

+ **Reliable**,
- Less flexible, require manual effort

+ **Flexible**, potentially fully automated
- Less reliable!

Lecture Structure

1. The Tower of Babel of Programming Languages
2. **Machine Translation of Programming Languages**
3. Transformers
4. Unsupervised Translation
5. Translation validation
6. What’s next

Machine Translation of Programming Languages

Translate language S into language T.

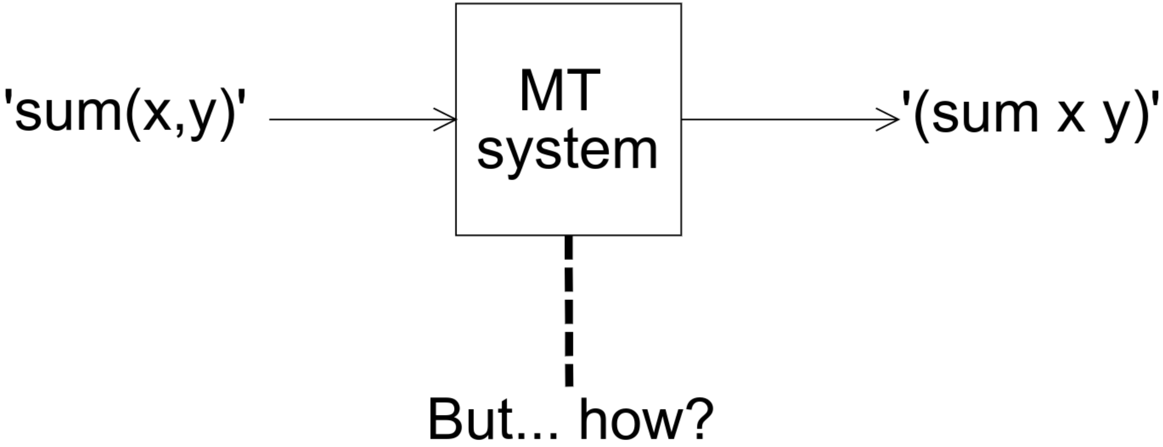
Typically, instances in S and T are expressed as discrete **sequences of tokens**.

$$s = [s_1, \dots s_n] \in S, t = [t_1, \dots t_m] \in T$$

Originally developed for natural languages

Potentially applicable to any **discrete domain**!

Machine Translation of Programming Languages



Machine translation for natural languages:

A. Rule-based MT: Grammars, dictionaries… PL-like approaches
B. Statistical MT: Faster and less data-hungry than NMT
C. **Neural MT:**
    a. Neural networks feature extractors
    b. Ranking translations
    c. **End-to-end**   (i.e. let the model do everything)

Focus on end-to-end NMT.

Tokenization: Assuming a finite vocabulary {‘x’: 0, ‘(‘: 1, ‘)’: 2, ‘,’: 3, ‘y’: 4, ‘sum’: 5}

‘sum(x,y)’ -> [‘sum’, ‘(‘, ‘x’, ‘,’ , ‘y’, ‘)’] -> [5, 1, 0, 3, 4, 2]

Avoid **out-of-vocabulary** words with **finite** vocabulary?

“my_variable_2” identifier

Solution: **subword encoding**

Subword encoding:

Training:

1. Initialize vocabulary with all characters
2. Merge most frequent pairs
3. Until desired vocabulary size reached.

Inference:

1. If token in vocabulary - done
2. Otherwise, express with subtokens

E.g.: ‘my_variable_2’ -> [‘my’, ‘_’, ‘variable’, ‘_’, ‘2’]

List of token indices: [5, 1, 0, 3, 4, 2]

Neural networks need **continuous vectors.**

Solution: introduce **embedding lookup table.**

0: [0.754, -0.25, …]

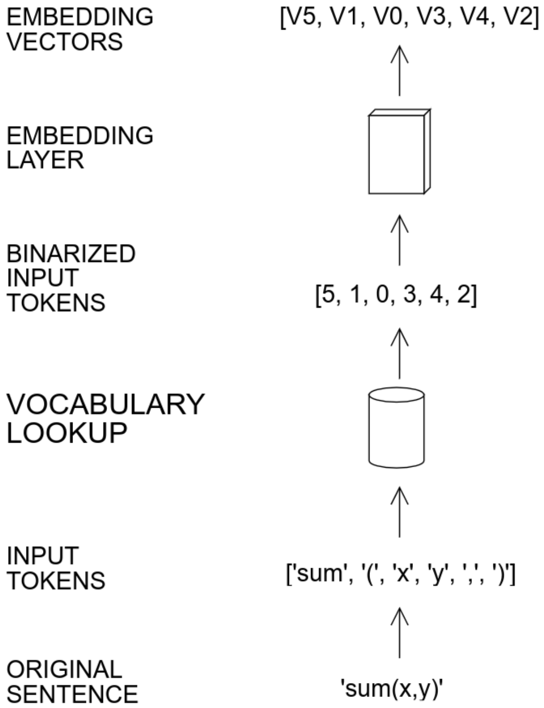
1: [-0.11, …]

2: …

Initialized to **random** values **Optimized end to end** with the rest of the network.

Vectors will get **closer to the vector that minimizes the error** of predictions

Machine Translation of Programming Languages



vectors of floats (NN-friendly).

But **variable-length** list of them (NN-unfriendly).

- multi-layer perceptron (MLP) requires a fixed size vector input.

Several possibilities:

- Apply an MLP to each vector, and aggregate all outputs somehow (mean? sum?)
- Recurrent neural network - LSTM, a well-known RNN architecture
- Other

Seq2seq, an **encoder-decoder** architecture.

- Encoder architecture: **RNN** (Recurrent Neural Network).
- Decoder architecture: another RNN.

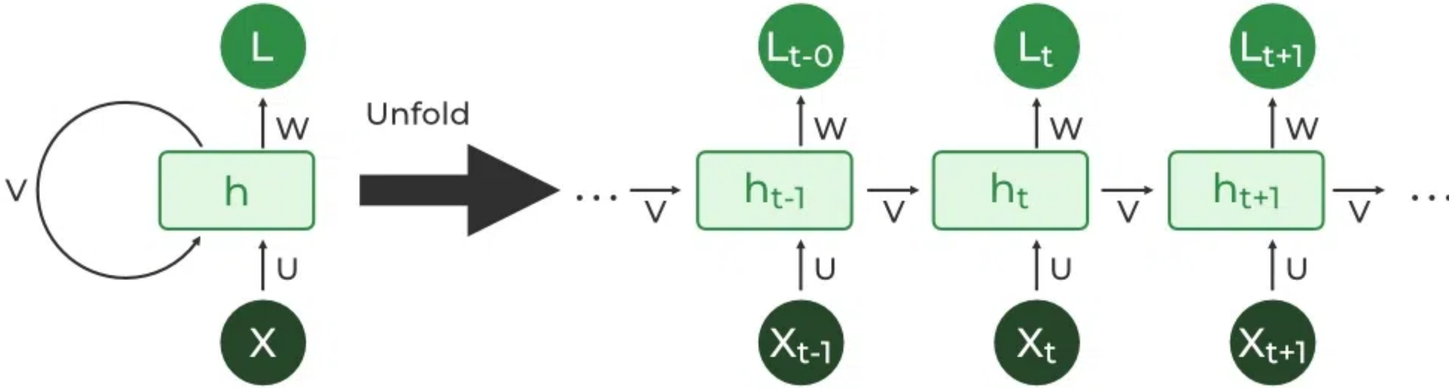
Encoder:

- **variable-length sequence** of embedding vectors
- into a single, **fixed-size vector**.

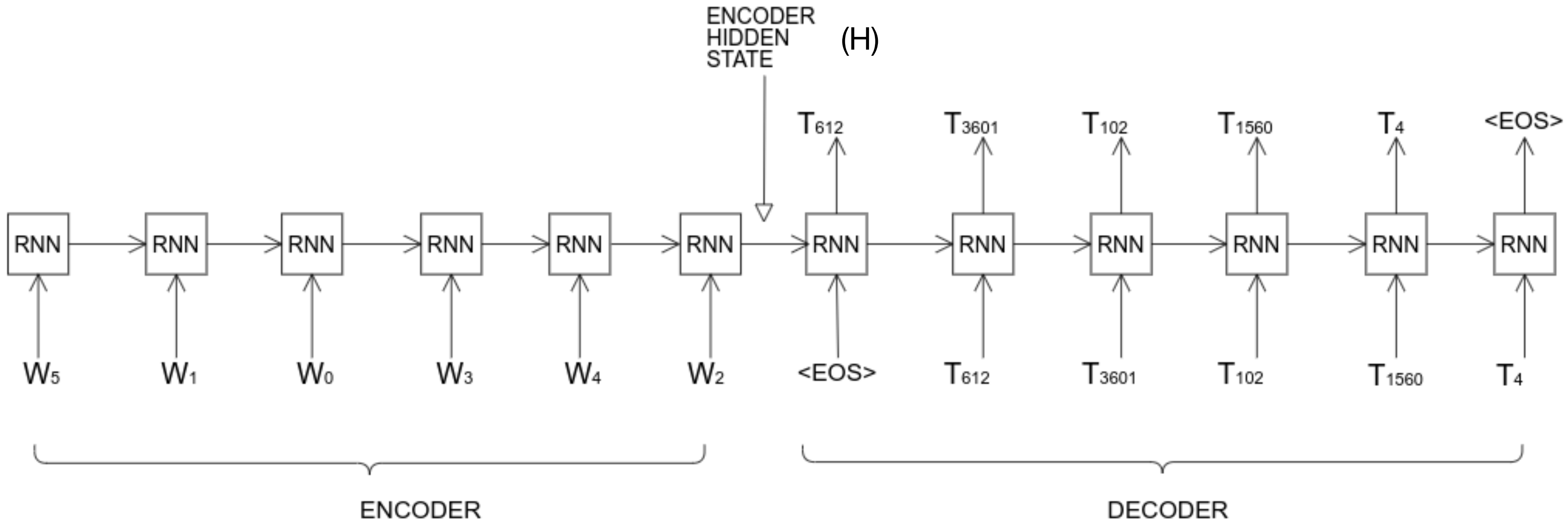
Decoder: **fixed-size vector** (from the encoder) into a variable-length **sequence of target token** indices.

RNN **for each token**

feed the embedding of the token itself and the RNN output from the previous token (**state**/”memory”)



Machine Translation of Programming Languages

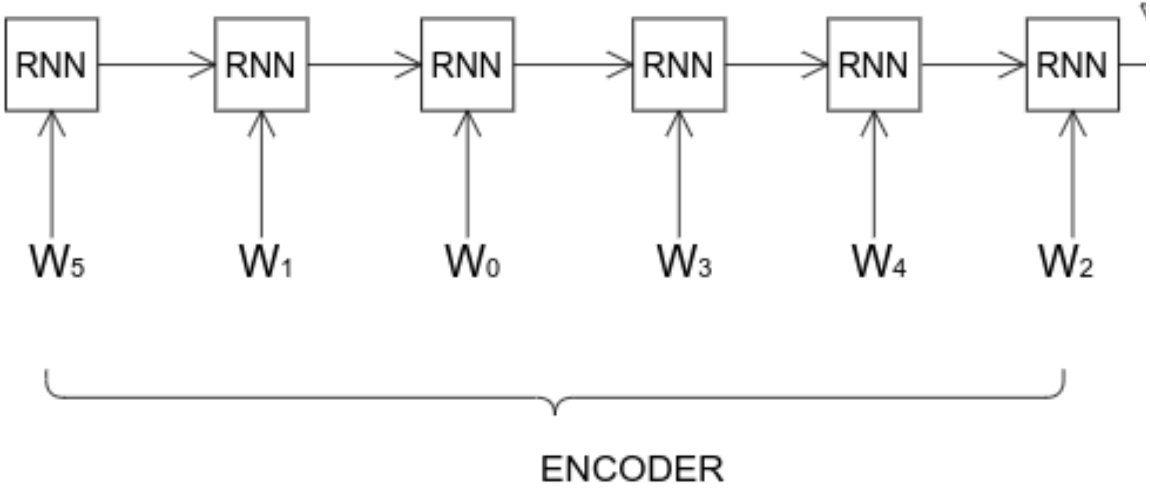


Machine Translation of Programming Languages

`Encoder(source tokens):`

- returns single, fixed-size vector **H** ("**h**idden state") representing the source sequence.

It's the output of the RNN after the last token!

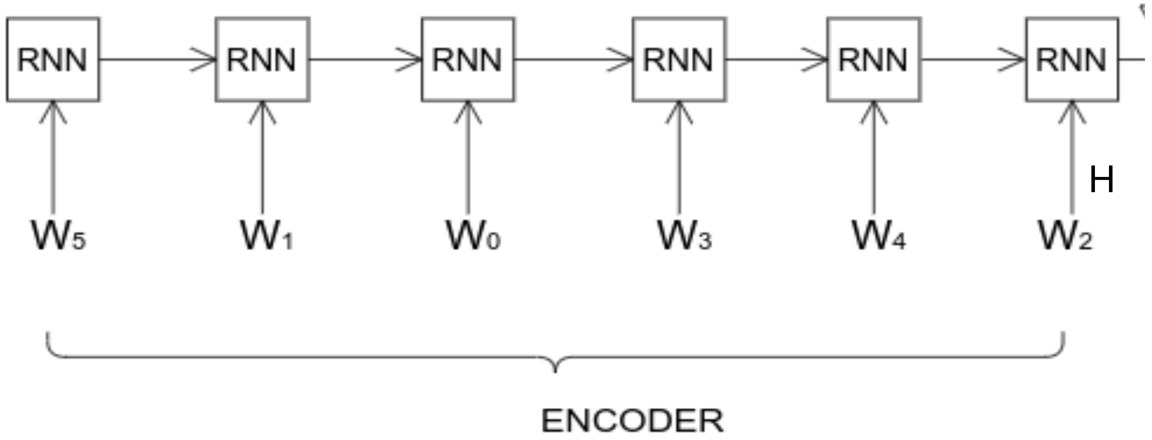


H

Machine Translation of Programming Languages

We have transformed the **original source string into a single vector of floats** (H)!

What do we do now with this H?



Machine Translation of Programming Languages

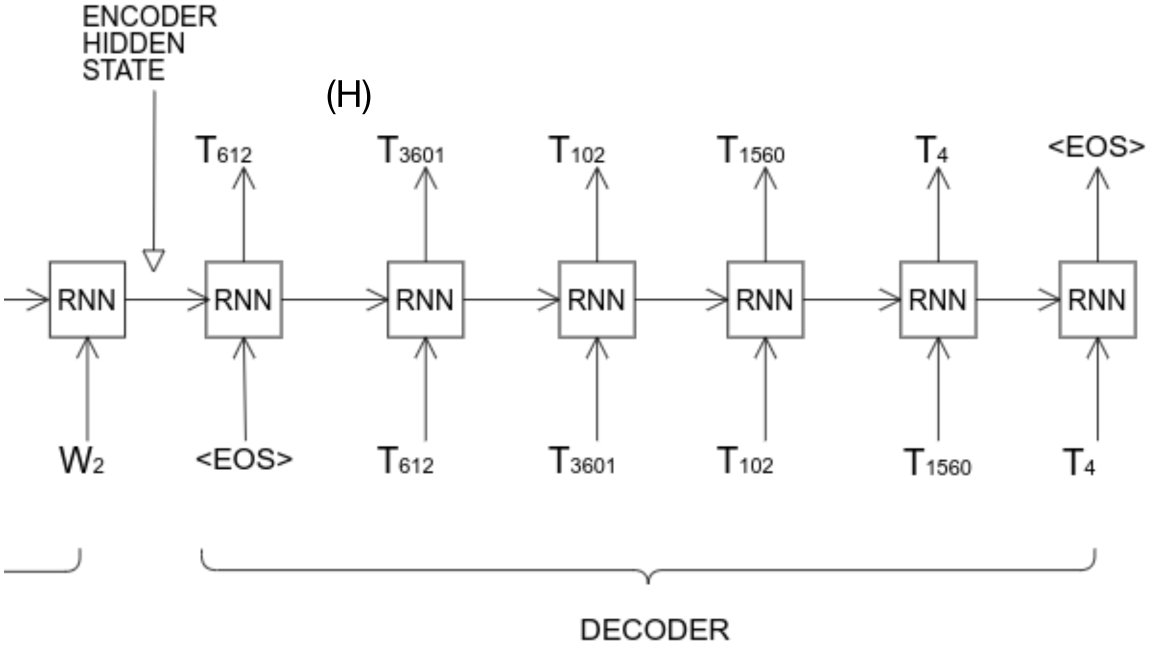
`Decoder(H, current_prediction=[]):`

Returns next target token.

Decoder, also an RNN.

But **not the same RNN!**

It has independent parameters.

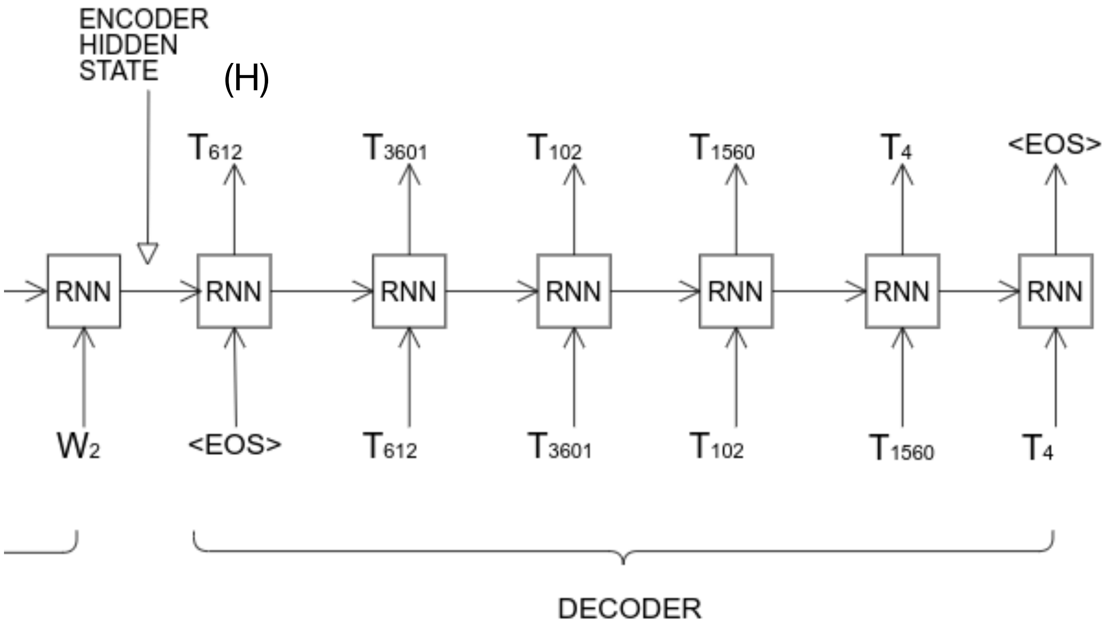


Second “time step” (**T3601**)
decoder reads:

- Encoder final output H.
- Previous Decoder output

Output:

- Useful representation for the next time step.
- Probability over the target vocabulary,
- T3601 has a very high probability



Decoding: inference predictions out of the decoder

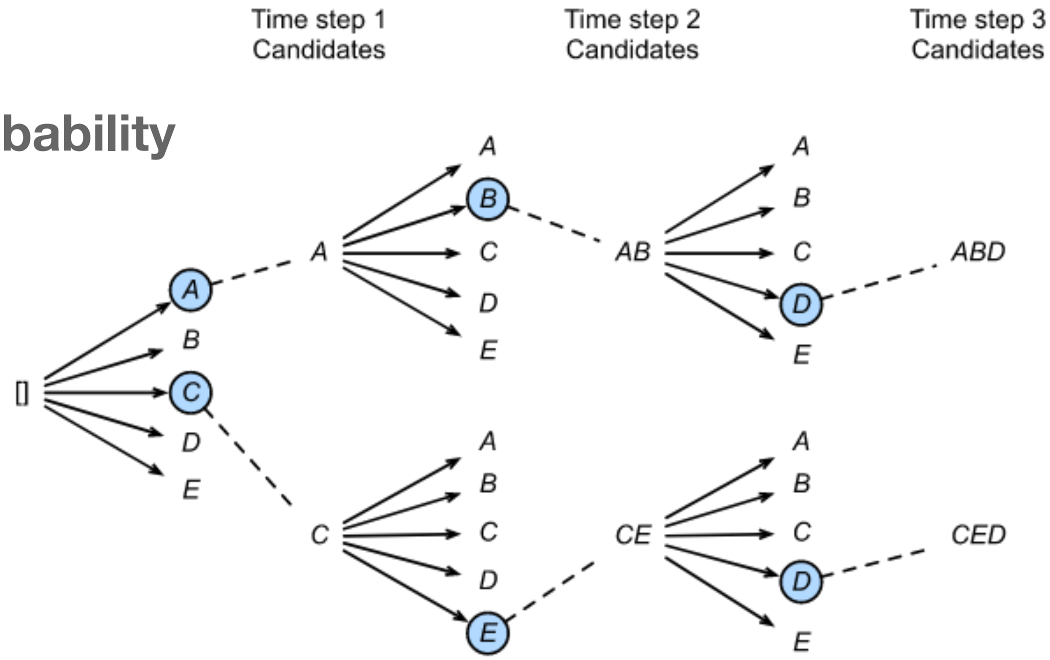
Simplest approach: greedy decoding.

- **Argmax** for each individual token prediction.
- Problem: maximizes local probability, not the global one!

Decoding

Goal: optimize global probability

Solution: beam search.

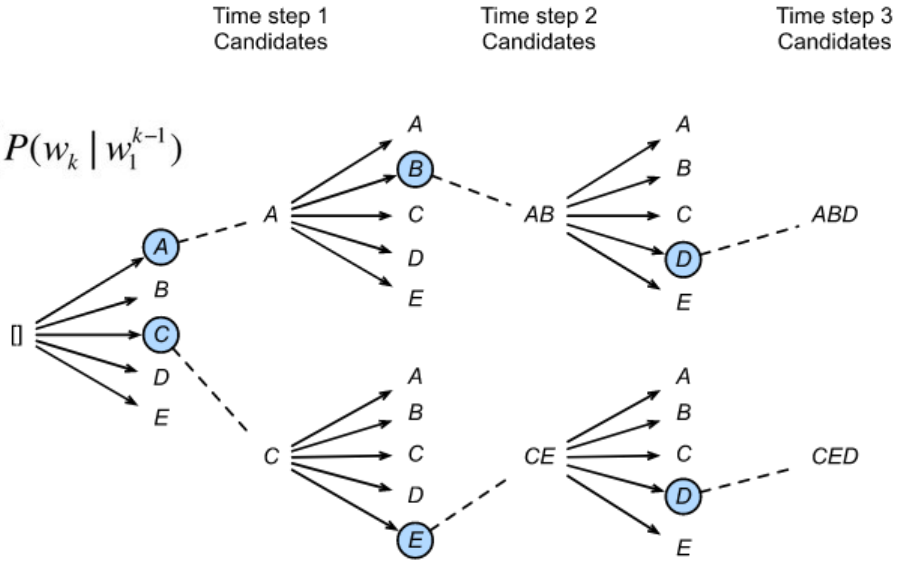


Probability of the sequence:

$$P(w_1^n) = P(w_1)P(w_2 \mid w_1)P(w_3 \mid w_1^2)...P(w_n \mid w_1^{n-1}) = \prod_{k=1}^{n} P(w_k \mid w_1^{k-1})$$

Beam search with `beam_size=k`

We **keep the top k most probable** sentences along all paths.



The information bottleneck:

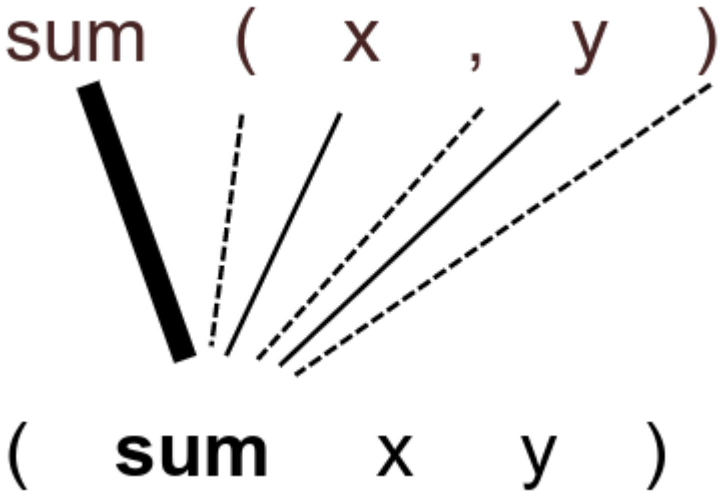
Encode the source with a single, fixed-size vector without losing information?
No

Solution: **attention**, i.e. **weighted average over source** hidden states.

Weights are dynamic (depend on the current decoder state).

Learned end-to-end with the rest of the neural network

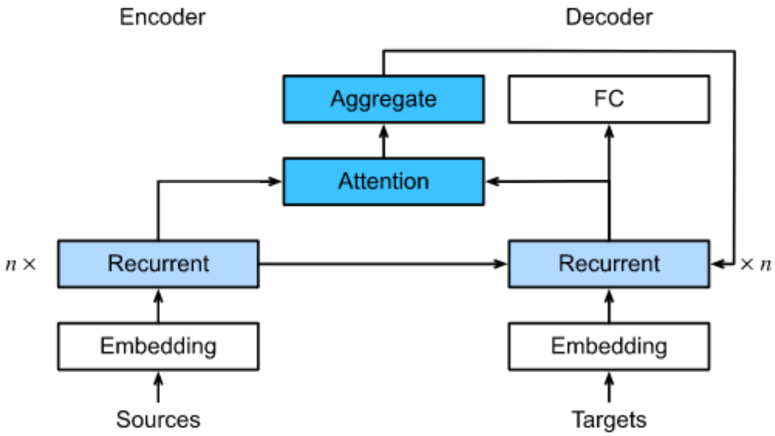
Attention



Bahdanau attention: Seq2seq ++

Encoder identical to Seq2seq.

Weighted average of all source hidden states. The weights depend on each decoder state.



Can we do better than that?

Transformers

Lecture Structure

1. The Tower of Babel of Programming Languages
2. Machine Translation of Programming Languages
3. **Transformers**
4. Unsupervised Translation
5. Translation validation
6. What’s next

Transformers

Transformers are eating the world [4]

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

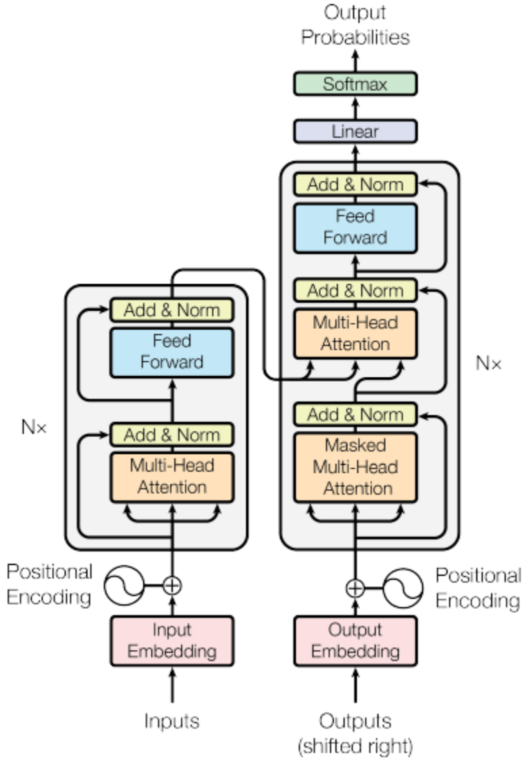
Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com



Especially in the form of large language models!

Language Models are Few-Shot Learners

Tom B. Brown[*] Benjamin Mann[*] Nick Ryder[*] Melanie Subbiah[*]

Jared Kaplan[†] Prafulla Dhariwal Arvind Neelakantan Pranav Shyam Girish Sastry

Amanda Askell Sandhini Agarwal Ariel Herbert-Voss Gretchen Krueger Tom Henighan

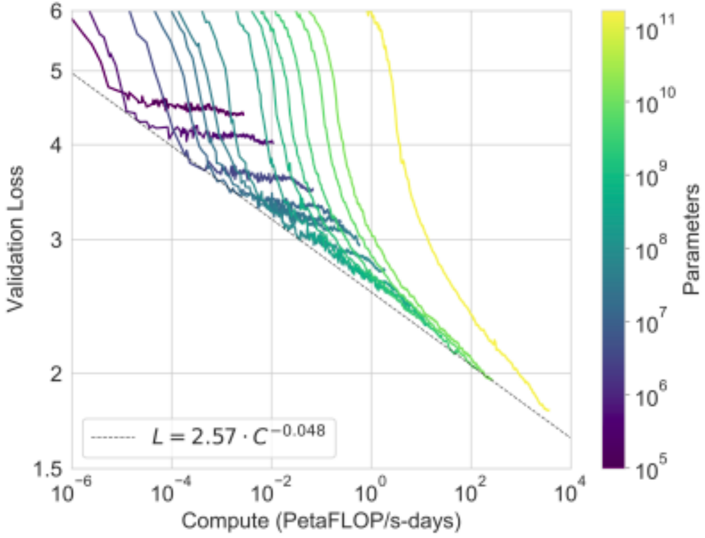
Rewon Child Aditya Ramesh Daniel M. Ziegler Jeffrey Wu Clemens Winter

Christopher Hesse Mark Chen Eric Sigler Mateusz Litwin Scott Gray

Benjamin Chess Jack Clark Christopher Berner

Sam McCandlish Alec Radford Ilya Sutskever Dario Amodei

OpenAI



Transformers

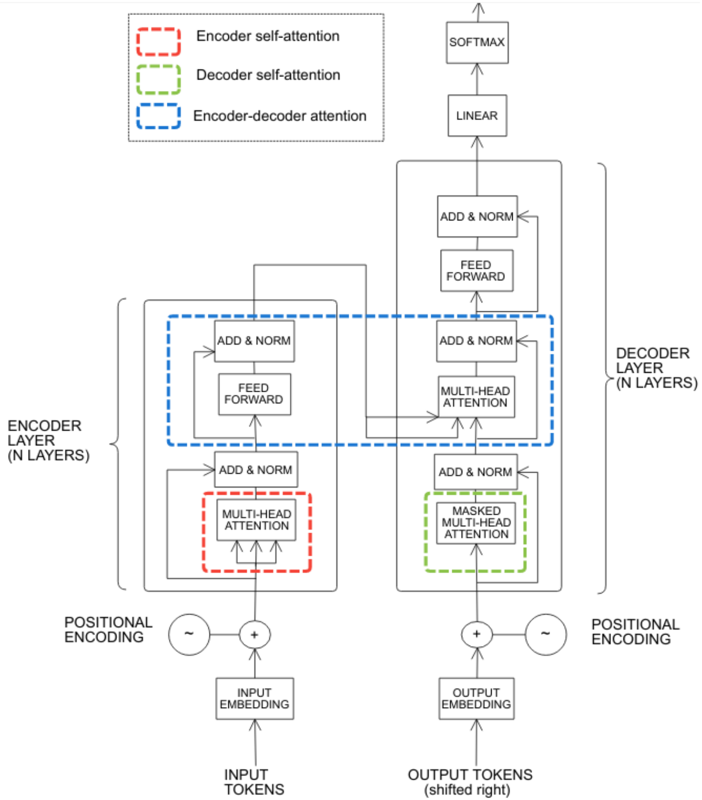
Transformers “all-attention” **encoder-decoder** NN (**no recurrence**).

High-level - similar Seq2seq.

Low-level - different

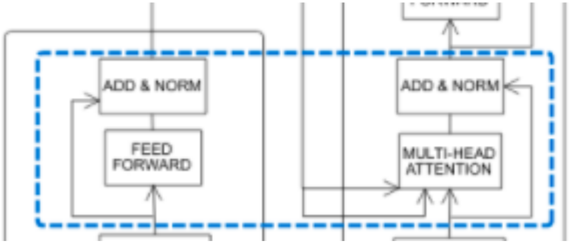
- parallel, pair-wise attention instead of sequential, recurrent network

Transformers



Seq2seq++ **encoder-decoder attention**, attend to the source sequence wrt the target sequence.

The Transformer **also** has encoder-decoder attention, no fundamental difference here!



Implementation is slightly more complicated than Bahdanau attention.

Machine Translation of Programming Languages

Replaces RNNs with self-attention

- in the encoder and the decoder
- attend to the source sequence wrt the source sequence itself, in the encoder,
- attend to the target sequence wrt the target sequence itself, in the decoder.

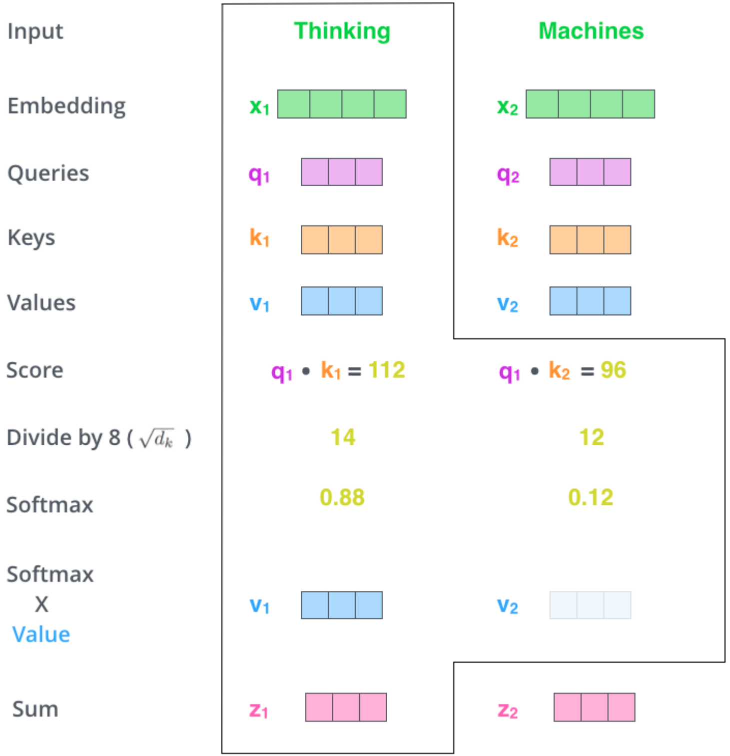
Attention implementation: **scaled dot-product attention**

Queries: “Is X information present?”

Keys: “Is information present in Y word?”

Use **query · value dot product** as the importance weight in the **weighted average**.

Value: “Now that we know the weight importance, what’s the information that we wanted to transfer?”



All attention implemented identically.

Difference is **where queries, keys, and values come from**:

- Encoder: the source sequence.
- Decoder: the target sequence.
- Encoder-decoder: Queries come from the **target** sequence. Keys and Values come from the **source** sequence.

Encoder attention can be applied **in parallel**

Decoder **cannot run in parallel** in inference

Transformers scale better than previously thought.

Quadratic complexity with respect to the sequence length **n**.

But

- As **embedding dimension d increase.** n becomes less important.
- Some tasks (??) can be solved in **relatively small sequence length n**.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \times d)$	$O(1)$	$O(1)$
Recurrent	$O(n \times d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \times n \times d^2)$	$O(1)$	$O(\log_k(n))$

Lecture Structure

1. The Tower of Babel of Programming Languages
2. Machine Translation of Programming Languages
3. Transformers
4. **Unsupervised Translation**
5. Translation validation
6. What’s next

Unsupervised translation

NMT large **parallel datasets S, T.**

In NLP   pairs (S,T) with the same meaning.

- Subtitled movies or European Parliament **transcriptions**
- Using **sentence alignment** algorithms.

PL more difficult:

- need **function-to-function** parallel datasets
- no natural occurrences of them (**Exception: Code-assembly pairs**).

Unsupervised translation more important in programming languages!

Unsupervised learning

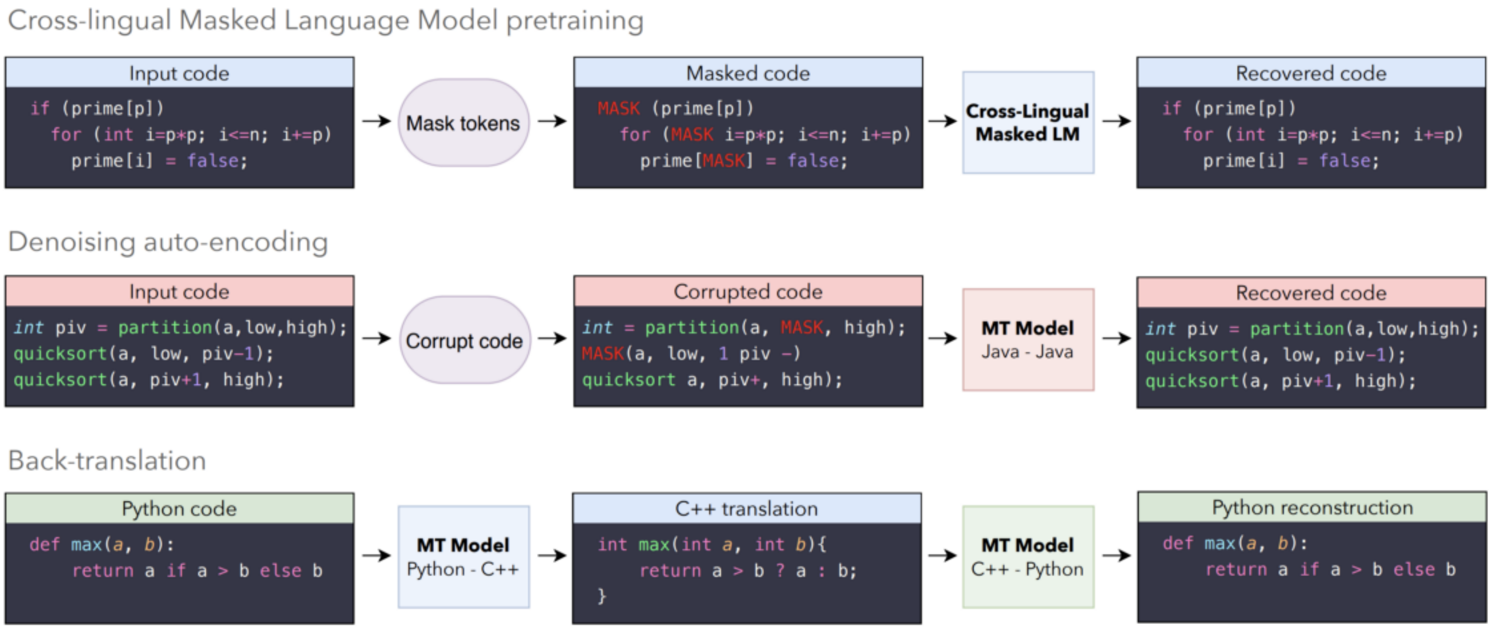
- Unsupervised algorithms?
- **Apply supervised algorithms to unlabelled data.**

Basic idea:

- automatically **create a low-quality, synthetic parallel dataset**
- **bootstrap** the learning.
- **Iteratively improve.**

Unsupervised translation

Unsupervised Translation of Programming Languages (Transcoder)[7]



Create a synthetic dataset.

- Step 1: Pretrain an **encoder**

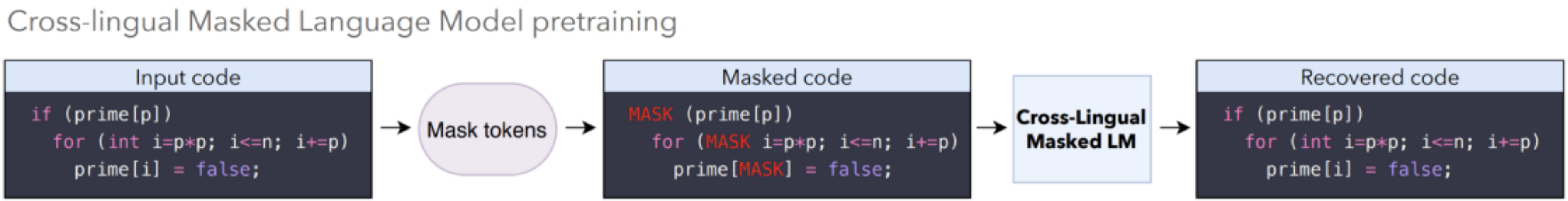
Predict masked tokens.

E.g. “def **<MASK>**(x,y): return x + y” -> predict “sum”.

Randomly replace tokens from the functions with a special token <MASK>.

- Train the encoder to predict the masked tokens
- (Similar to word2vec in L3)

Unsupervised translation



Unsupervised translation

Randomly mask functions (**Unsupervised**).

- Pretraining task forces the encoder to learn useful embeddings

Learn **aligned embeddings**,

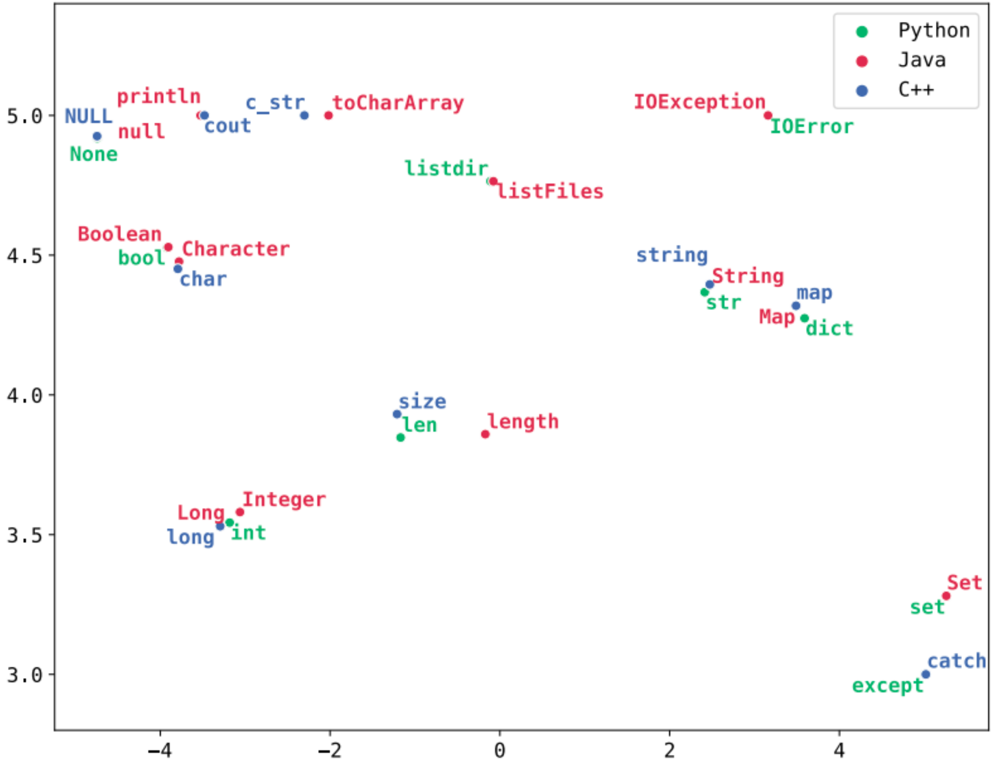
- **same** encoder to simultaneously demask functions in language A and B
  shared vocabulary and embedding table.

Same model for both,

- forces alignment
- tokens in A occur in **semantically similar contexts** in B

Unsupervised translation

Tokens used in similar contexts cluster together even if they belong to different languages.



Step 2: Use pre-trained, multilingual encoder to **initialize an encoder-decoder Transformer.**

- Encoder identical to the pretrained one.

Decoder identical to the encoder

- except randomly initialize the cross-attention modules

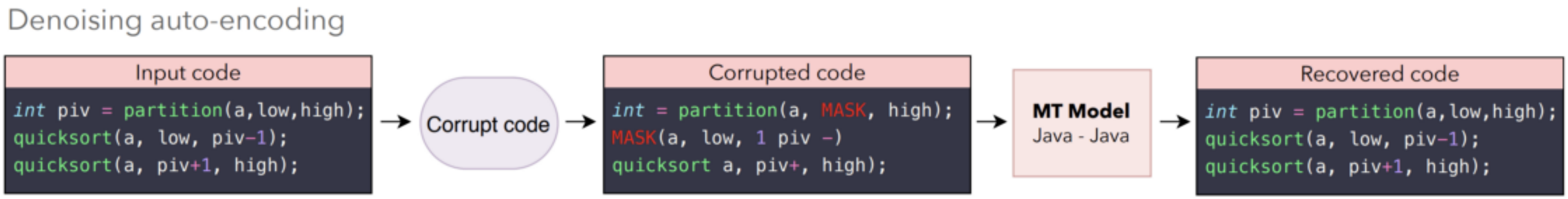
Train this encoder-decoder model to **demask** the original sentence

But using **special tokens** to denote the desired target (e.g., <TO_PYTHON>).

Unsupervised translation

Example:

“def **<MASK>** (x, y): x+y” -> encoder + **<TO_PYTHON>** -> “def **sum**(x, y): x+y”



Step 3: **Zero-shot translation** (unseen directions).

Trained Python to Python and Java to Java.

- After feeding a **Java** function can we inject **<TO_PYTHON>** instead of **<TO_JAVA>** ?

Trained the same model on Java and on Python.

- Can we translate without having done it in training?

Yes the **embeddings are aligned**

- the model is going to extrapolate to perform a poor-quality translation!

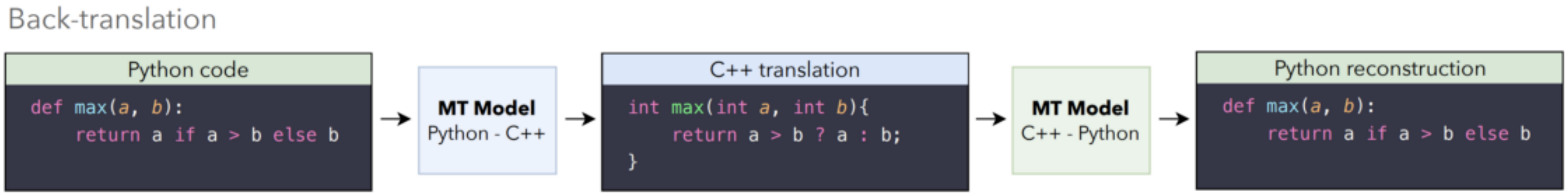
Unsupervised translation

Poor-quality Java-Python translations are **not useful on their own**

But can build a **parallel synthetic dataset**

For every Python sentence:

1. Apply (bad) translator to get C++ **-ish**
2. Use the reverse direction to build a synthetic dataset **back translation**



Use synthetic dataset to learn translator using supervised learning.

iteratively improve

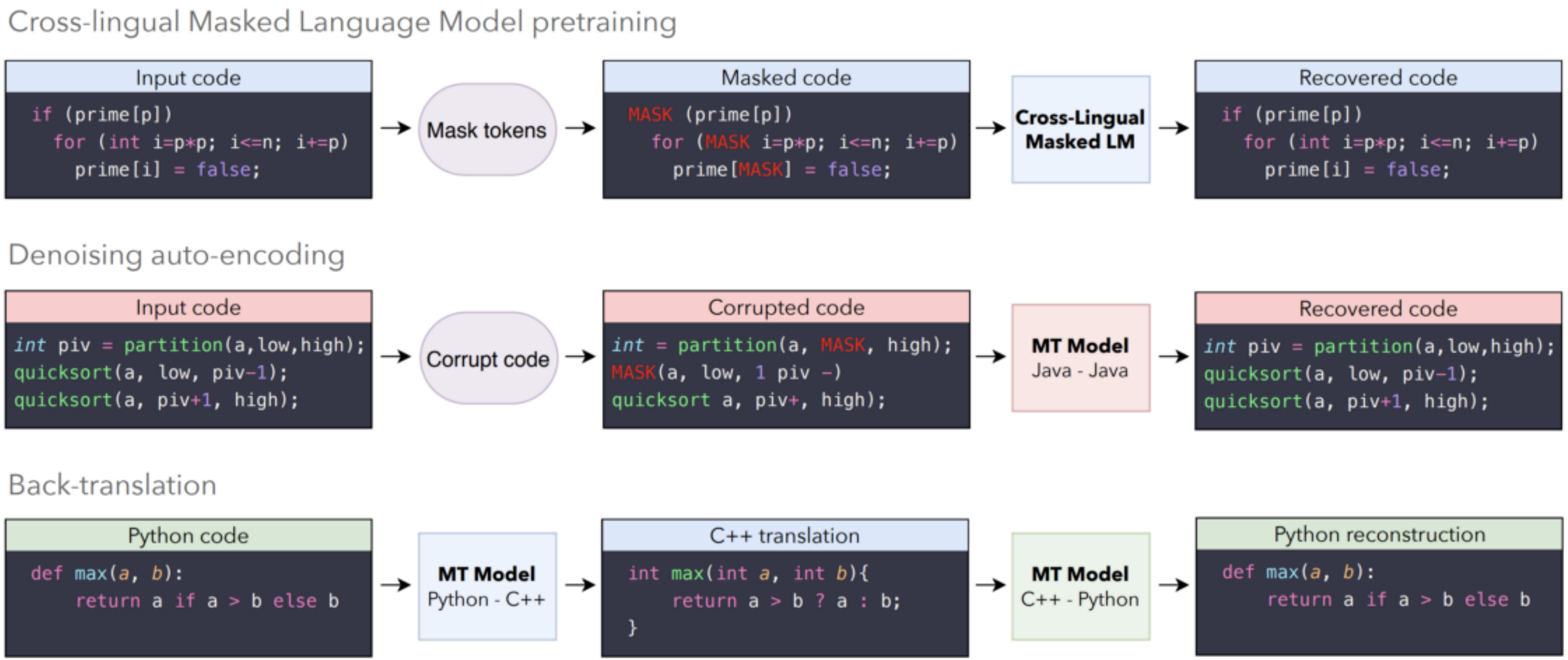
Use the new translator (better than the zero-shot one)

Better synthetic dataset and train an even better translator.

Unsupervised translation will be key for PL/compilers

Unsupervised translation

Unsupervised Translation of Programming Languages (Transcoder) [7]



```python
def SumOfKsubArray(arr, n, k):
  Sum = 0
  S = deque()
  G = deque()
  for i in range(k):
    while (len(S) > 0 and arr[S[-1]] >= arr[i]):
      S.pop()
    while (len(G) > 0 and arr[G[-1]] <= arr[i]):
      G.pop()
    G.append(i)
    S.append(i)
  for i in range(k, n):
    Sum += arr[S[0]] + arr[G[0]]
    while (len(S) > 0 and S[0] <= i - k):
      S.popleft()
    while (len(G) > 0 and G[0] <= i - k):
      G.popleft()
    while (len(S) > 0 and arr[S[-1]] >= arr[i]):
      S.pop()
    while (len(G) > 0 and arr[G[-1]] <= arr[i]):
      G.pop()
    G.append(i)
    S.append(i)
  Sum += arr[S[0]] + arr[G[0]]
  return Sum
```

```cpp
int SumOfKsubArray(int arr[], int n, int k){
  int Sum = 0;
  deque <int> S;
  deque <int> G;
  for(int i = 0; i < k; i ++){
    while((int) S.size() > 0 && arr[S.back()] >= arr[i])
      S.pop_back();
    while((int) G.size() > 0 && arr[G.back()] <= arr[i])
      G.pop_back();
    G.push_back(i);
    S.push_back(i);
  }
  for(int i = k; i < n; i ++){
    Sum += arr[S.front()] + arr[G.front()];
    while((int) S.size() > 0 && S.front() <= i - k)
      S.pop_front();
    while((int) G.size() > 0 && G.front() <= i - k)
      G.pop_front();
    while((int) S.size() > 0 && arr[S.back()] >= arr[i])
      S.pop_back();
    while((int) G.size() > 0 && arr[G.back()] <= arr[i])
      G.pop_back();
    G.push_back(i);
    S.push_back(i);
  }
  Sum += arr[S.front()] + arr[G.front()];
  return Sum;
}
```

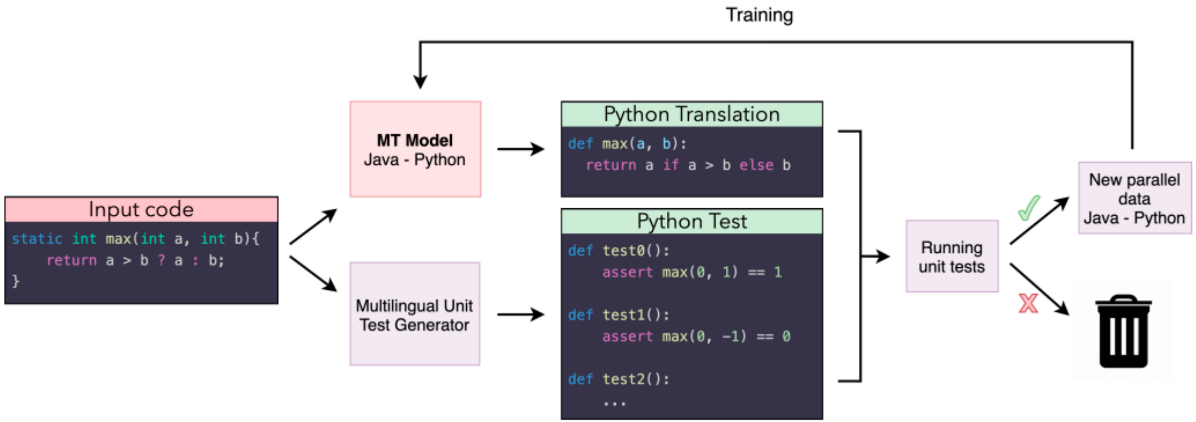
24 to 60% accuracy

Unsupervised translation

Bonus: select the synthetic functions

- **discard the bad ones** by running the original **unit tests** on them.

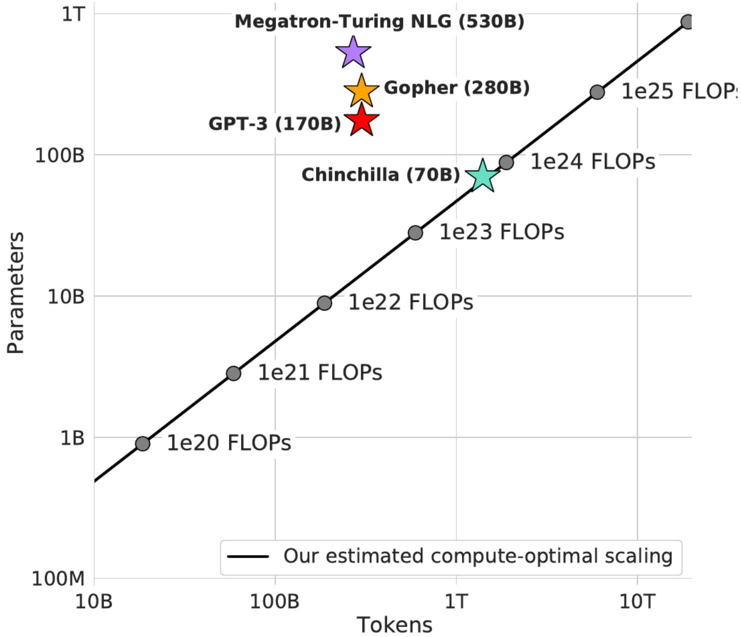
- **discard the syntactically incorrect** synthetic functions.

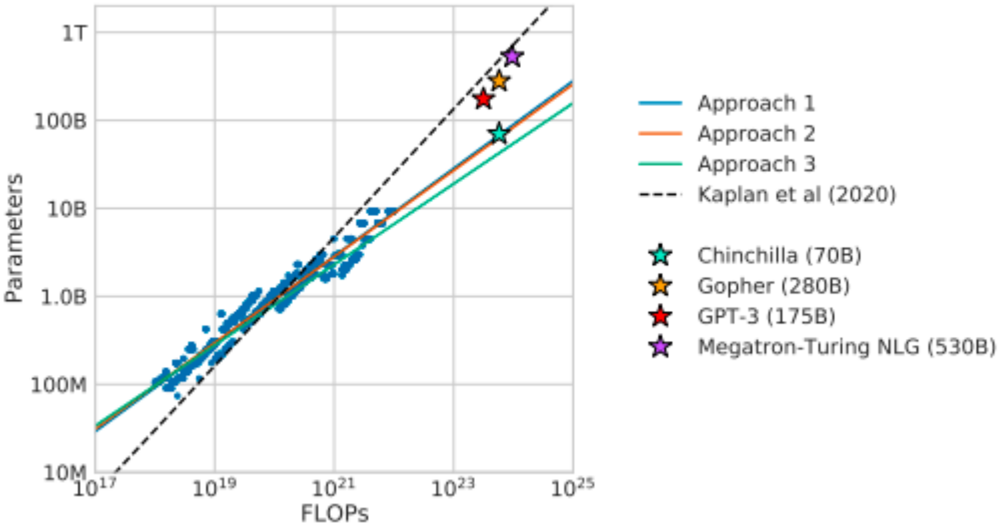


Language models

Neural **Scaling Laws**: neural language models scale smoothly with optimally allocated compute.

No diminishing returns yet?!





Lecture Structure

1. The Tower of Babel of Programming Languages
2. Machine Translation of Programming Languages
3. Transformers
4. Unsupervised Translation
5. **Translation validation**
6. What’s next

Validation/evaluation

Evaluating text generation **hard**

Natural languages, require **human evaluation**

- **slow** and **expensive**.

Solution: **similarity** metrics **correlate** with human evaluation.

BLEU (n-gram match) in academia and **edit similarity** in the industry.

Validation/evaluation

BLEU in Action

(Source Original) 枪手被警方击毙.

(Reference Translation) **The gunman was shot to death by the police**

The gunman was police	#1
Wounded police jaya of	#2
The gunman was shot dead by the police	#3
The gunman arrested by police kill	#4
The gunmen were killed	#5
The gunman was shot to death by the police	**#6**
Gunmen were killed by police ?SUB>0 ?SUB>0	#7
Al by the police	#8
The ringer is killed by the police	#9
Police killed the gunman	#10

Best BLEU Score

green=4-gram match	**(very good!)**
grey=3-gram match	**(good)**
black=1 or 2 gram match	**(Ok)**
red=word not matched	**(bad!)**

Source: homepages.inf.ed.ac.uk/pkoehn/publications/autoeval2004.ppt

Validation/evaluation

What about code translation?

Disadvantage: **hard/binary correctness**, a single character may corrupt an otherwise perfect solution.

Advantage: well-defined **syntax** and **semantics**

Validation/evaluation

- **Exact match**: wrt ground truth translation.
- **CodeBLEU**: BLEU with **AST** features.
- **IO/unit tests**: correct iff passes unit tests (“**observational equivalence**”)
- **Formally verified**: equivalent to source sequence.

Validation/evaluation

Compiler validation vs **translator** validation:

- Compilers/synthesizers are **correct by design**/construction
- Translator validation >>> compiler validation
- Growing interest in neural network interpretability/verifiability

Lecture Structure

1. The Tower of Babel of Programming Languages
2. Machine Translation of Programming Languages
3. Transformers
4. Unsupervised Translation
5. Translation validation
6. **What’s next**

What's next

Methods are **generic**: apply to any sequence

Add i**nductive biases** from PL?

- **graph neural networks**. Good at code classification, but **underperform Transformers at generative tasks**!
- "bitter lesson" for ML for code: "just" scaling up outperforms everything?

Can we learn from program synthesis and compilers ?

What's next

Code inductive biases are likely to inform future designs

Best changes will not necessarily be in the architecture level:

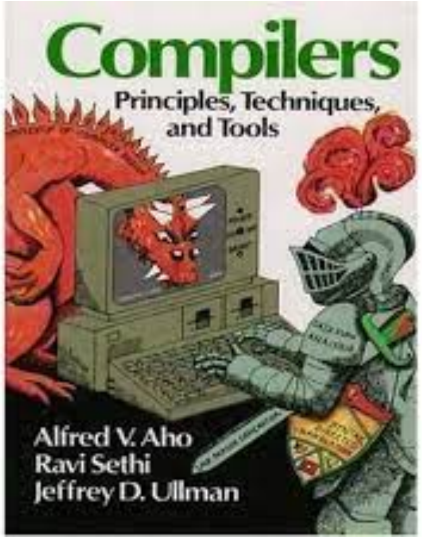
- **Data**: can we preprocess data differently/better?
- **Training objectives**: can we find better loss functions for the code domain?
- **Hybrid methods**: can we effectively combine NMT with program synthesis?
- **Input/output pairs**: can we use input/output pairs to guide the translation?

David vs Goliath: Decompilation and ChatGPT

(SLaDe: A Portable Small Language Model Decompiler for Optimized Assembler)

[CGO2024]

Decompilation: x86->C

C ← 

← x86

Difficult to run compilers backwards

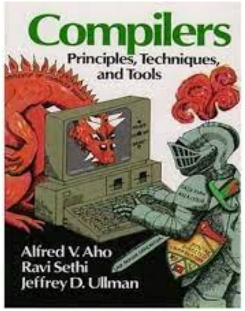
Decompilation: x86->C

Explored for 50 years

- Used as precursor for binary translation
- Surprising lack of evaluation on correctness
- Often produce mangled code

Several commercial and open-source tools

Many person-years of effort
- Retdec over 25 person-years



State-of-the-art GHIDRA

C ← GHIDRA ← x86

Decompiler from NSA
Many person-years effort

GHIDRA

Useful for:

- Security purposes!
- Porting legacy code.
- Lifting?

Excellent rule-based decompiler

… but produces hard-to-read code!

GCC O3:

4

```
 .globl add
 .type add, @function
add:
.LFB0:
 .cfi_startproc
 endbr64
 movq %rdi, %rcx
 testl %edx, %edx
 jle .L1
 leal -1(%rdx), %eax
 cmpl $2, %eax
 jbe .L6
 movq %rdi, %rax
 movl %edx, %edi
 movd %esi, %xmm2
 shrl $2, %edi
 pshufd $0, %xmm2,
       %xmm1
 subl $1, %edi
 salq $4, %rdi
 leaq 16(%rcx,%rdi),%rdi
 .p2align 4,,10
 .p2align 3
.L4:
 movdqu (%rax), %xmm0
 addq $16, %rax
 paddd %xmm1, %xmm0
 movups %xmm0, -16(%rax)
 cmpq %rdi, %rax
 jne .L4
 movl %edx, %edi
 andl $-4, %edi
 testb $3, %dl
 je .L9
.L3:
 movslq %edi, %rax
 leal 1(%rdi), %r8d
 salq $2, %rax
 addl %esi, (%rcx,%rax)
 cmpl %r8d, %edx
 jle .L1
 addl $2, %edi
 addl %esi, 4(%rcx,%rax)
 cmpl %edi, %edx
 jle .L1
 addl %esi, 8(%rcx,%rax)
.L1:
 ret
 .p2align 4,,10
 .p2align 3
.L9:
 ret
.L6:
 xorl %edi, %edi
 jmp .L3
 .cfi_endproc
```

What does this code do?

Generates structured code
- but difficult to read

Non-intuitive variable names

Use of shifts and masks

Follows O3 control-flow mangling

1

```
void add(int *param_1, int param_2,
         uint param_3) {
  uint uVar1;
  int *piVar2;
  int *piVar3;

  if (0 < (int)param_3) {
    if (param_3 - 1 < 3) {
      uVar1 = 0;
      do {
        param_1[(int)uVar1] =
          param_1[(int)uVar1] + param_2;
        uVar1 = uVar1 + 1;
      } while ((int)uVar1 < (int)param_3);
      return;
    } else {
      piVar2 = param_1;
      do {
        piVar3 = piVar2 + 4;
        *piVar2 = *piVar2 + param_2;
        piVar2[1] = piVar2[1] + param_2;
        piVar2[2] = piVar2[2] + param_2;
        piVar2[3] = piVar2[3] + param_2;
        piVar2 = piVar3;
      } while (piVar3 = param_1 +
        ((ulong)param_3 >> 2) * 4,
        piVar2 != piVar3);

      uVar1 = param_3 &
        0xfffffffc;
      if ((param_3 & 3) == 0) {
        return;
      }
      param_1[(int)uVar1] =
        param_1[(int)uVar1] + param_2;
      if ((int)(uVar1 + 1) <
          (int)param_3) {
        param_1[(int)(uVar1 + 1)] =
          param_1[(int)(uVar1 + 1)] +
          param_2;
        if ((int)(uVar1 + 2) <
            (int)param_3) {
          param_1[(int)(uVar1 + 2)] =
            param_1[(int)(uVar1 + 2)] +
            param_2;
          return;
        }
      }
    }
  }
}
```

ChatGPT

3

```c
void add(int* dest, int* src, int n) {
  if (n <= 0) {
    return;
  }
  // If n is smaller than 2, use a loop
  // to add the values
  if (n < 2) {
    for (int i = 0; i < n; i++) {
      dest[i] += src[i];
    }
    return;
  }

  // Align dest pointer to
  // 16-byte boundary
  int* aligned_dest = (int*)
     (((uintptr_t) dest + 15) & ~15);

  // Calculate the number of elements
  // before the aligned dest pointer
  int pre_count = (aligned_dest - dest);

  // Add the unaligned elements
  for (int i = 0; i < pre_count; i++) {
    dest[i] += src[i];
  }
```

```c
  // Add the aligned elements
  int count = (n - pre_count) / 4;
  __m128i* sse_src =
        (__m128i*) (src + pre_count);
  __m128i* sse_dest =
        (__m128i*) (aligned_dest);
  __m128i xmm1 =
        _mm_loadu_si128(sse_src);
  for (int i = 0; i < count; i++) {
    __m128i xmm0 =
      _mm_load_si128(sse_dest);
    xmm0 = _mm_add_epi32(xmm0, xmm1);
      _mm_store_si128(sse_dest, xmm0);
    sse_src++;
    sse_dest++;
  }

  // Add the remaining elements
  int remaining = n -
    (pre_count + count * 4);
  for (int i = 0; i < remaining; i++) {
    dest[pre_count + count * 4 + i] +=
       src[pre_count + count * 4 + i];
  }
}
```

ChatGPT

Compared to Ghidra

+ More readable code than Ghidra
+ More meaningful variable names
+ Even adds comments!
+ Compiles and executes

However

- complex code
- introduces x86 intrinsics
- **gives incorrect results**

3

```
void add(int* dest, int* src, int n) {
  if (n <= 0) {
    return;
  }
  // If n is smaller than 2, use a loop
  // to add the values
  if (n < 2) {
    for (int i = 0; i < n; i++) {
      dest[i] += src[i];
    }
    return;
  }

  // Align dest pointer to
  // 16-byte boundary
  int* aligned_dest = (int*)
     (((uintptr_t) dest + 15) & ~15);

  // Calculate the number of elements
  // before the aligned dest pointer
  int pre_count = (aligned_dest - dest);

  // Add the unaligned elements
  for (int i = 0; i < pre_count; i++) {
    dest[i] += src[i];
  }

  // Add the aligned elements
  int count = (n - pre_count) / 4;
  __m128i* sse_src =
        (__m128i*) (src + pre_count);
  __m128i* sse_dest =
        (__m128i*) (aligned_dest);
  __m128i xmm1 =
        _mm_loadu_si128(sse_src);
  for (int i = 0; i < count; i++) {
    __m128i xmm0 =
      _mm_load_si128(sse_dest);
    xmm0 = _mm_add_epi32(xmm0, xmm1);
      _mm_store_si128(sse_dest, xmm0);
    sse_src++;
    sse_dest++;
  }

  // Add the remaining elements
  int remaining = n -
    (pre_count + count * 4);
  for (int i = 0; i < remaining; i++) {
    dest[pre_count + count * 4 + i] +=
       src[pre_count + count * 4 + i];
  }
}
```

SLaDe: both correct and readable!

Original Source

2

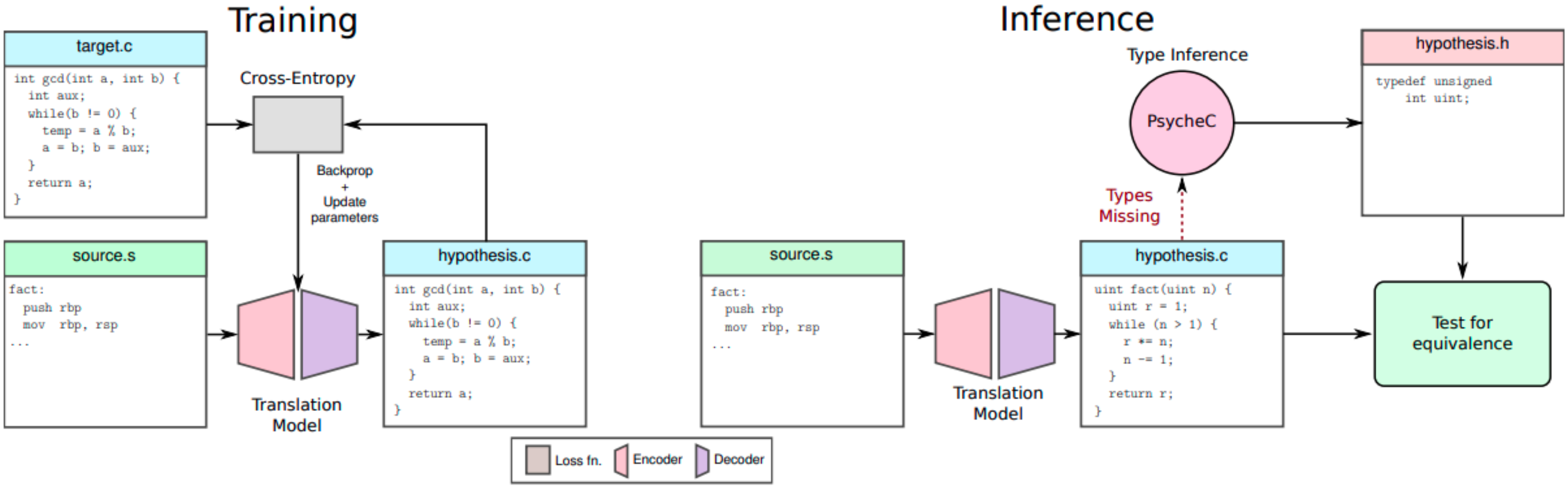
```
void add(int *list, int val, int n) {
  int i;
  for (i = 0; i < n; ++i) {
    list[i] += val;
  }
}
```

```
void add(int a[], int x, int y) {
  int i;
  for (i = 0; i < y; i++) {
     a[i] += x;
  }
}
```

6

SLaDe

SLaDe architecture: small Transformer + Type Inference



200M parameters (David) vs ChatGPT(Goliath): O(10^3) larger

Datasets: AnghaBench (large-scale training and evaluation)

AnghaBench: a Suite with One Million Compilable C Benchmarks for Code-Size Reduction

Anderson Faustino da Silva
Department of Informatics
UEM, Brazil
anderson@din.uem.br

Bruno Conde Kind
Department of Computer Science
UFMG, Brazil
condekind@dcc.ufmg.br

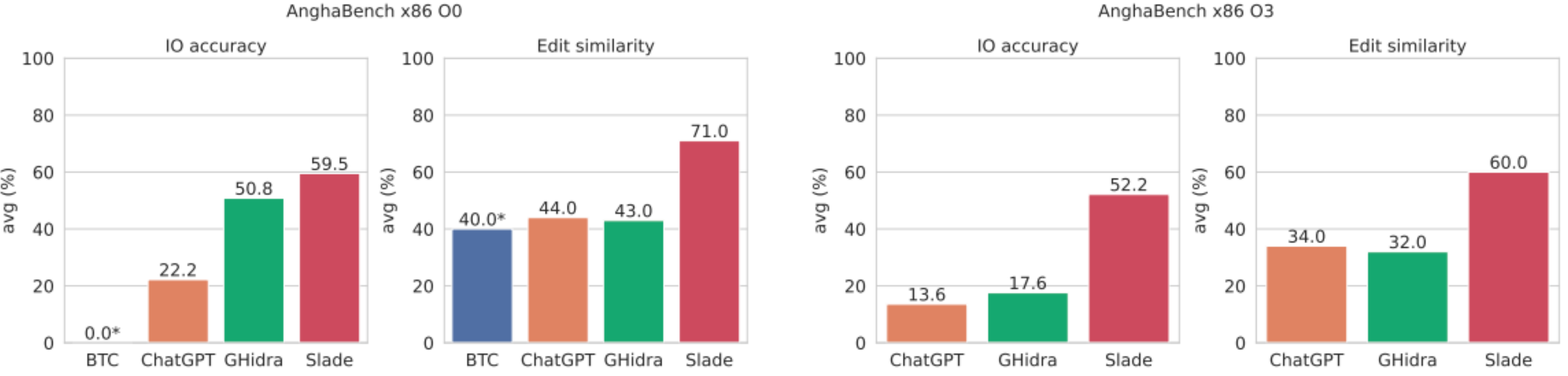
José Wesley de Souza Magalhães
Department of Computer Science
UFMG, Brazil
josewesleysouza@dcc.ufmg.br

Jerônimo Nunes Rocha
Department of Computer Science
UFMG, Brazil
jeronimonunes@dcc.ufmg.br

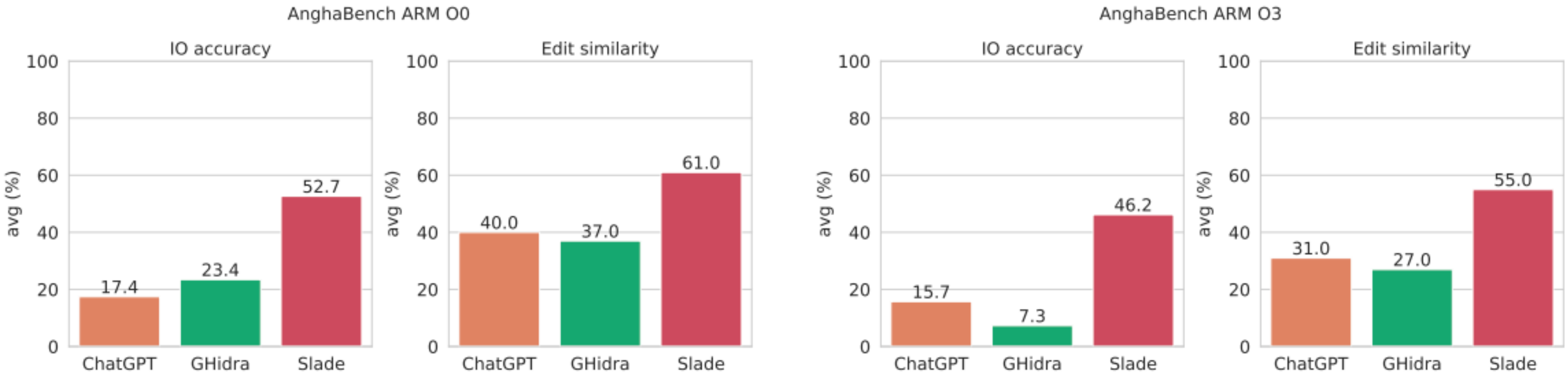
Breno Campos Ferreira Guimarães
Department of Computer Science
UFMG, Brazil
brenosfg@dcc.ufmg.br

Fernando Magno Quintão Pereira
Department of Computer Science
UFMG, Brazil
fernando@dcc.ufmg.br

x86: 3x improvement on O3



ARM: 3x to 6x improvement on O3



Analysis

Ghidra
- code complexity causes problems
- fundamentally cannot predict types of external functions

ChatGPT
- performs well on x86 O0 but O3 causes significant difficulty
- worryingly produces compilable code that is wrong

SLaDe
- rarely produces compilable code that is incorrect
- can be improved with program analysis interacting with decoder

Conclusion

To adapt to a world of language/hardware innovation

We need to rethink compilation

Lots of unusual technology - great time for research!

mob@inf.ed.ac.uk

Rethinking Compilation

Alexander Braukmann, Jordi Armengol Estape, Jose Wesley Magalhaes. Michael O’Boyle, Jackson Woodruff

Bibliography

[1] Phrase-Based Statistical Translation of Programming Languages: S. Karaivanov, Veselin Raychev, Martin Vechev (2014)

[2] Sequence to Sequence Learning with Neural Networks: Ilya Sutskever, Oriol Vinyals, Quoc V. Le (2014)

[3] Neural Machine Translation by Jointly Learning to Align and Translate: Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio (2014)

[4] Attention Is All You Need: Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin (2017).

[5] BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding: Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova (2018).

[6] Language Models are Unsupervised Multitask Learners: Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever (2018).

[7] Unsupervised Translation of Programming Languages: Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, Guillaume Lample (2020).

[8] SLaDe: A Portable Small Language Model Decompiler for Optimised Assembler, Jordi Armengol-Estape, Jackson Woodruff, Chris Cummins , Michael FP O’Boyle