

# More About Transformers

Mohammad Taher Pilehvar

Machine Learning 1401

<https://teias-courses.github.io/ml01/>

# Breaking (Transformer) News!

AlphaCode (a pre-trained Transformer-based code generation model) achieved a top 54.3% rating on Codeforces programming competitions (Li et al., 2022)!

<https://alphacode.deepmind.com/>

## AlphaCode Attention Visualization

Hover over tokens in the solution to see which tokens the model attended to when generating the solution. Click a token to select it; clicking in empty space will deselect.

Solutions were selected randomly, keeping at most one correct (passes all test cases in our dataset) and one incorrect sample per problem and language. Note that since our dataset only has a limited number of test cases, passing all tests we have cannot completely rule out false positives (~4%), or solutions that are correct but inefficient (~46%).

Check out selected problems with commentary from World-Class Competitive Programmer Petr Mitrichev: [1566\\_E](#) [1591\\_C](#) [1618\\_B](#) [1618\\_D](#) [1619\\_D](#) [1623\\_B](#)

Read our [paper](#) and [blog post](#) for more.

(annotated) 1566\_E. Buds Re-hanging C++ pass Layer 18 play

Head 1 Head 2 Head 3 Head 4 Head 5 Head 6 Head 7 Head 8 Head 9 Head 10 Head 11 all none

### Problem Description

```
// RATING: 3300
// TAGS: constructive algorithms, greedy, implementation
// LANGUAGE IS cpp
// CORRECT SOLUTION
// A tree is a connected graph without cycles. A rooted tree has a special
// vertex called the root. The parent of a vertex v (different from root) is the
// previous to v vertex on the shortest path from the root to the vertex v.
// Children of the vertex v are all vertices for which v is the parent.
//
// A vertex is a leaf if it has no children. We call a vertex a bud, if the
// following three conditions are satisfied:
//
// * it is not a root,
// * it has at least one child, and
// * all its children are leaves.
//
// You are given a rooted tree with n vertices. The vertex 1 is the root. In one
// operation you can choose any bud with all its children (they are leaves) and
// re-hang them to any other vertex of the tree. By doing that you delete the
// edge connecting the bud and its parent and add an edge between the bud and
```

This solution stands out from other AlphaCode solutions because it inputs a tree and then traverses it with a depth-first search (DFS). Here it most likely borrowed large chunks of code from other dfs-on-tree problems, such as the "dfs" function or the I/O. The sorting of adjacent vertices by depth seems to be unnecessary, fitting the pattern of useless-but-harmless code pieces that we see in other AlphaCode submissions.

There are 5 public sample cases in this problem, but all answers are small (1 or 2), therefore it's quite cool that the solution has managed to figure out the correct logic for the "getans" method with only those examples. I expect a typical human competitor to make at least a few wrong attempts in this problem.

– Petr Mitrichev, Software Engineer, Google & World-Class Competitive Programmer.

### Solution

```
#include <bits/stdc++.h>
using namespace std;
const int N = 2e5 + 10;
vector<int> g[N];
```

# Breaking (Transformer) News!

Pre-Trained Transformer-Based theorem prover sets new state-of-the-art (41.2% vs. 29.3%) on the [mnif2f](https://mnif2f.github.io/) collection of challenging math Olympiad questions (Polu et al., 2022)

<https://alphacode.deepmind.com/>

## PROBLEM 1

*Adapted from AMC12 2000 Problem 5*

Prove that if  $|x - 2| = p$ , where  $x < 2$ , then  $x - p = 2 - 2p$ .

◇ FORMAL

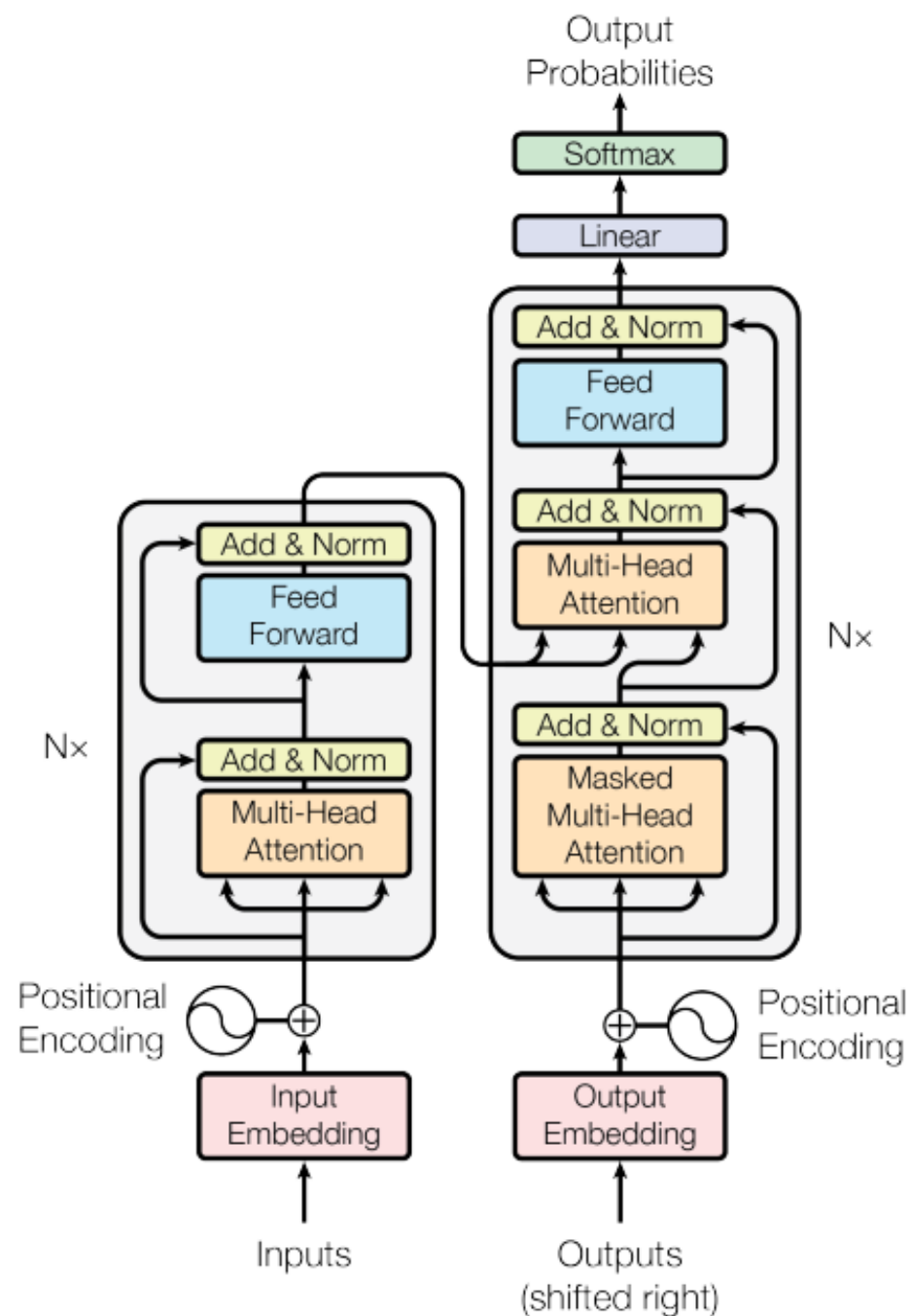
INFORMAL

```
theorem amc12_2000_p5 -- ← theorem name
  (x p : ℝ)           -- ← the statement we want
  (h₀ : x < 2)         -- ← to prove
  (h₁ : abs (x - 2) = p) :
  x - p = 2 - 2 * p :=
begin -- ← formal proof starts here
  -- This first tactic requires that the prover invent
  -- the term: `abs (x - 2) = -(x - 2)`.
  have h₂ : abs (x - 2) = -(x - 2), {
    apply abs_of_neg,
    linarith,
  },
  rw h₁ at h₂,
  -- At this stage the remaining goal to prove is:
  -- `x - p = 2 - 2 * p` knowing that `p = -(x - 2)`.
  linarith,
end
```

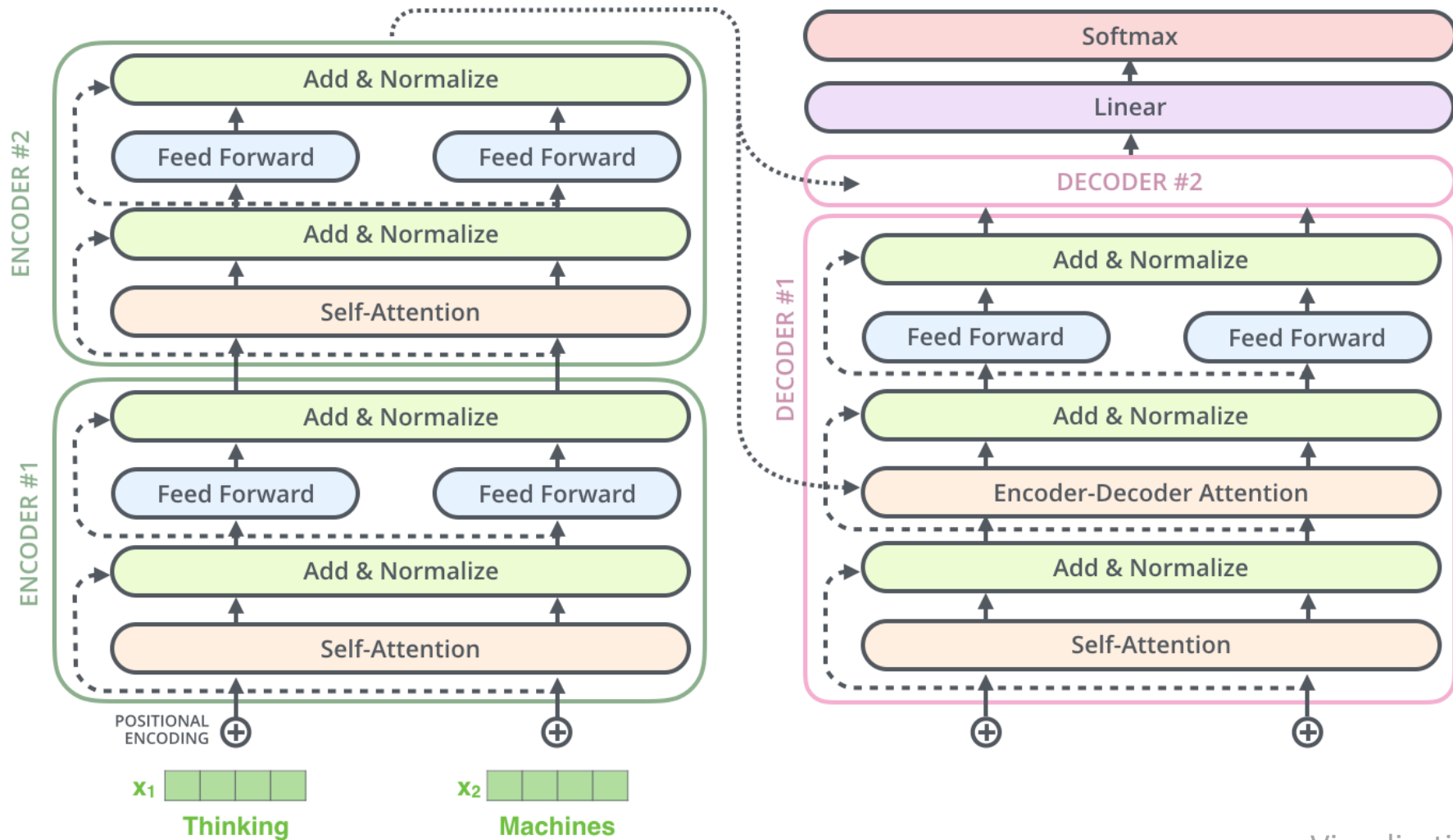
# In this Lecture

- Quick reminder of Transformer model
  - Subword modeling
  - Model types
    - Decoders
    - Encoders
    - Encoder-Decoders
- 
- Homework #2 will be out soon
  - Project proposal is due this week

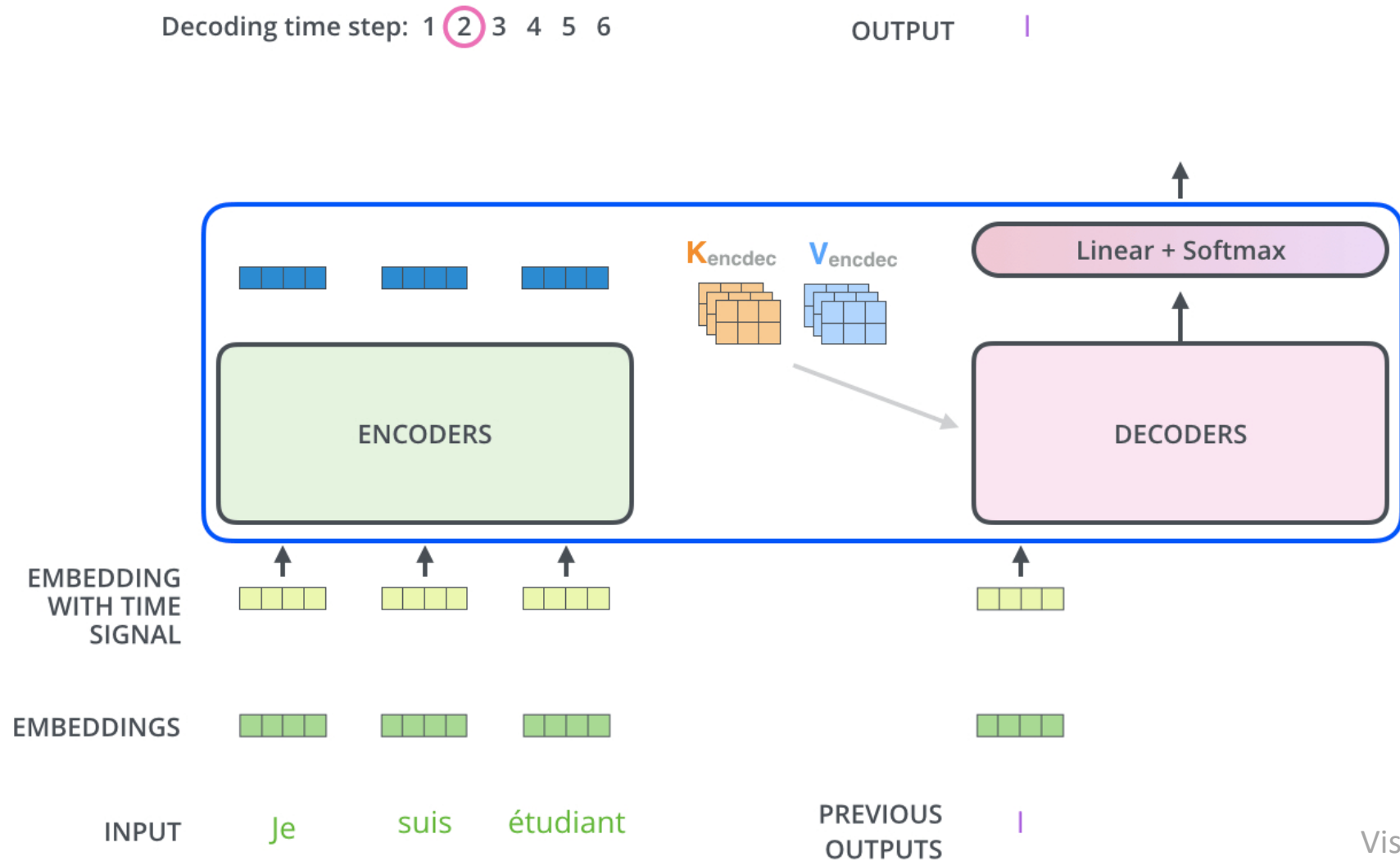
# Transformers



# Transformer with 2 stacked encoders and decoders



# Decoder side



# Tokenizing text

- Simplest: split by spaces

```
["Don't", "you", "love", "😊", "Transformers?", "We", "sure", "do."]
```

```
["Don", "'", "t", "you", "love", "😊", "Transformers", "?", "We", "sure", "do", "."]
```

- [spaCy](#) and [Moses](#) are two popular rule-based tokenizers.

```
["Do", "n't", "you", "love", "😊", "Transformers", "?", "We", "sure", "do", "."]
```



# Tokenizing text

- Hazm (<https://github.com/sobhe/hazm>)






```
>>> sent_tokenize('ما هم برای وصل کردن آمدیم! ولی برای پردازش، جدا بهتر نیست؟')  
['ما هم برای وصل کردن آمدیم!', 'ولی برای پردازش، جدا بهتر نیست؟']  
>>> word_tokenize('ما هم برای وصل کردن آمدیم! ولی برای پردازش، جدا بهتر نیست؟')  
['ما', 'هم', 'برای', 'وصل', 'کردن', 'آمدیم', '!', 'ولی', 'برای', 'پردازش', '،', 'جدا', 'بهتر', '،', 'نیست', '؟']
```

- Dadmatools (<https://github.com/Dadmatech/DadmaTools>)

# Word structure and subword models

Traditionally, we used to have a fixed vocab of hundreds of thousands of words (often built from the training set).

All *novel* words seen at test time are mapped to a single *UNK*.

	word		vocab mapping	embedding
Common words	hat	→	hat	
	learn	→	learn	
Variations	taaaaasty	→	UNK	
misspellings	laern	→	UNK	
novel items	Transformerify	→	UNK	

# Word structure and subword models

Fixed vocab makes *less* sense for languages with rich morphology (more word types, occurring less frequently)

گردایش فارسی							
				دیدن	بن واژه		
				دید	بن ماضی		
				بین	بن مضارع		
جمع			مفرد			شخص	
سوم شخص	دوم شخص	اول شخص	سوم شخص	دوم شخص	اول شخص		
آنها/ایشان	شما	ما	او/آن	تو	من	گذشته	
دیدند	دیدید	دیدیم	دید	دیدگی	دیدم		
می‌دیدند	می‌دیدید	می‌دیدیم	می‌دید	می‌دیدگی	می‌دیدم		
دیده‌بودند	دیده‌بودید	دیده‌بودیم	دیده‌بود	دیده‌بودگی	دیده‌بودم		
دیده‌باشند	دیده‌باشید	دیده‌باشیم	دیده‌باشد	دیده‌باشی	دیده‌باشم		
داشتند می‌دیدند	داشتید می‌دیدید	داشتیم می‌دیدیم	داشت می‌دید	داشتی می‌دیدگی	داشتم می‌دیدم		
آنها/ایشان	شما	ما	او/آن	تو	من	حال	
بینند	بینید	بینیم	بیند	بینی	بینم		
می‌بینند	می‌بینید	می‌بینیم	می‌بیند	می‌بینی	می‌بینم		
دیده‌اند	دیده‌اید	دیده‌ایم	دیده‌است/دیده	دیده‌ای	دیده‌ام		
دارند می‌بینند	دارید می‌بینید	داریم می‌بینیم	دارد می‌بیند	داری می‌بینی	دارم می‌بینم		
بینند	بینید	بینیم	بیند	بینی	بینم		
آنها/ایشان	شما	ما	او/آن	تو	من	آینده	
خواهند دید	خواهید دید	خواهیم دید	خواهد دید	خواهی دید	خواهم دید	دستوری	
-	شما		-	تو	-		
	بینید			بینی			
	نبینید			نبینی			

# Word structure and subword models

Big vocab size:

- Enormous embedding matrix (input and output layers)
- Increased memory
- Increased time complexity

In general, transformers models rarely have a vocabulary size greater than 50,000, especially if they are pretrained only on a single language.

# Word structure and subword models

[https://huggingface.co/docs/transformers/tokenizer\\_summary](https://huggingface.co/docs/transformers/tokenizer_summary)



**Hugging Face**

Subword tokenization algorithms rely on the principle that frequently used words should not be split into smaller subwords, but rare words should be decomposed into meaningful subwords.

Examples and demo:

<https://colab.research.google.com/drive/1E86l5oGlyZi7vIbzz0cC1RLgbTl8JVzm?usp=sharing>

[https://github.com/microsoft/SDNet/blob/master/bert\\_vocab\\_files/bert-base-uncased-vocab.txt](https://github.com/microsoft/SDNet/blob/master/bert_vocab_files/bert-base-uncased-vocab.txt)

# Word structure and subword models

## Byte-Pair Encoding (BPE)

1. Split the training data into words. Create a list of unique words with their frequency.

```
("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)
```

2. Create a base vocabulary consisting of all symbols that occur in the set of unique words

```
("h" "u" "g", 10), ("p" "u" "g", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "u" "g" "s", 5)
```

3. Learn merge rules to form a new symbol from two symbols of the base vocabulary (until a vocabulary of certain size is reached).

```
("h" "ug", 10), ("p" "ug", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "ug" "s", 5)
```

```
("hug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("hug" "s", 5)
```

# Word structure and subword models

## WordPiece

- Very similar to BPE
- In contrast to BPE, WordPiece does not choose the most frequent symbol pair, but the one that maximizes the likelihood of the training data once added to the vocabulary.
- Referring to the previous example, maximizing the likelihood of the training data is equivalent to finding the symbol pair, whose probability divided by the probabilities of its first symbol followed by its second symbol is the greatest among all symbol pairs.
  - u followed by g would have only been merged if the probability of ug divided by u (followed by) g would have been greater than for any other symbol pair

WordPiece



# Word structure and subword models






## SentencePiece

- Not all languages use spaces to separate words
- SentencePiece treats the input as a raw input stream, thus including the space in the set of characters to use.
- It then uses the BPE or other algorithms to construct the appropriate vocabulary.

# Word structure and subword models

Common words end up being a part of the subword vocabulary, while rarer words are split into (sometimes intuitive, sometimes not) components.

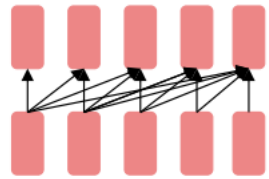
In the worst case, words are split into as many subwords as they have characters.

	word		vocab mapping	embedding
Common words	hat	→	hat	
	learn	→	learn	
Variations	taaaaasty	→	taa## aaa## sty	
misspellings	laern	→	la## ern	
novel items	Transformerify	→	Transformer## ify	

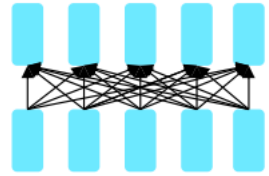
# Outline

- Quick reminder of Transformer model
- Subword modeling
- Model types
  - Decoders
  - Encoders
  - Encoder-Decoders

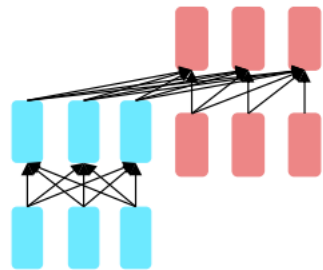
# Model types



**Decoders**



**Encoders**



**Encoder-  
Decoders**

- Language models! What we've seen so far.
  - Nice to generate from; can't condition on future words
  - **Examples:** GPT-2, GPT-3, LaMDA
- 
- Gets bidirectional context – can condition on future!
  - Wait, how do we pretrain them?
  - **Examples:** BERT and its many variants, e.g. RoBERTa
- 
- Good parts of decoders and encoders?
  - What's the best way to pretrain them?
  - **Examples:** Transformer, T5, Meena

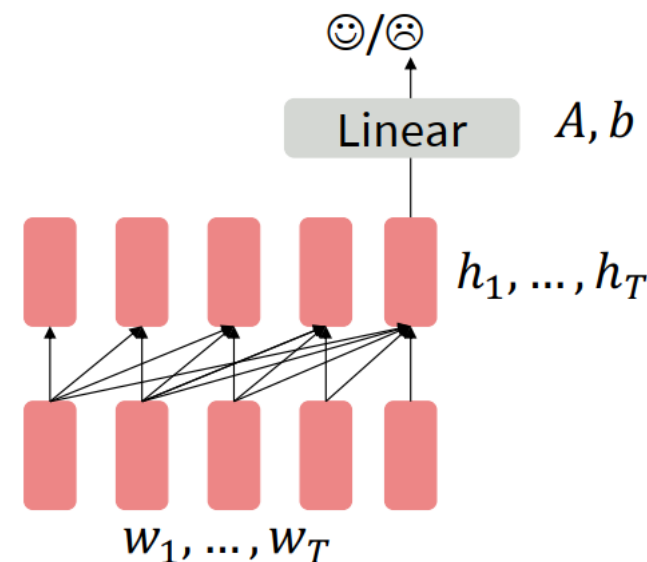
# Decoders

- When using language model pretrained decoders, we can ignore that they were trained to model  $p(w_t|w_{1:t-1})$
- We can finetune them by training a classifier on the last word's hidden state.

$$h_1, \dots, h_t = \text{Decoder}(w_1, \dots, w_t)$$

$$y \sim Ah_T + b$$

- Where  $A$  and  $b$  are randomly initialized and specified by the downstream task. Gradients backpropagate through the whole network.



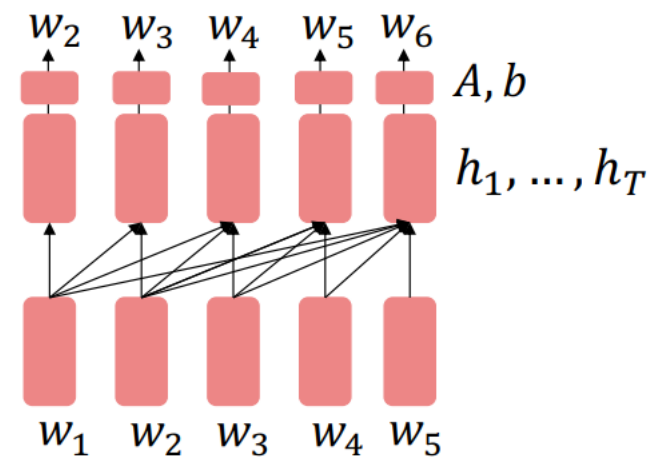
# Decoders

- It's natural to pretrain decoders as language models and then use them as generators, finetuning their  $p(w_t|w_{1:t-1})$
- This is helpful in tasks where the output is a sequence with a vocabulary like that at pretraining time!
  - Dialogue (context=dialogue history)
  - Summarization (context=document)

$$h_1, \dots, h_t = \text{Decoder}(w_1, \dots, w_t)$$

$$w_t \sim Ah_{t-1} + b$$

- Where  $A, b$  were pretrained in the language model!



# Decoders

Generative Pretrained Transformer (GPT) [Radford et al., 2018]

2018's GPT was a big success in pretraining a decoder!

- Transformer decoder with 12 layers.
- 768-dimensional hidden states, 3072-dimensional feed-forward hidden layers.
- Byte-pair encoding with 40,000 merges
- Trained on BooksCorpus: over 7000 unique books.
- Contains long spans of contiguous text, for learning long-distance dependencies

# Decoders

Generative Pretrained Transformer (GPT) [Radford et al., 2018]

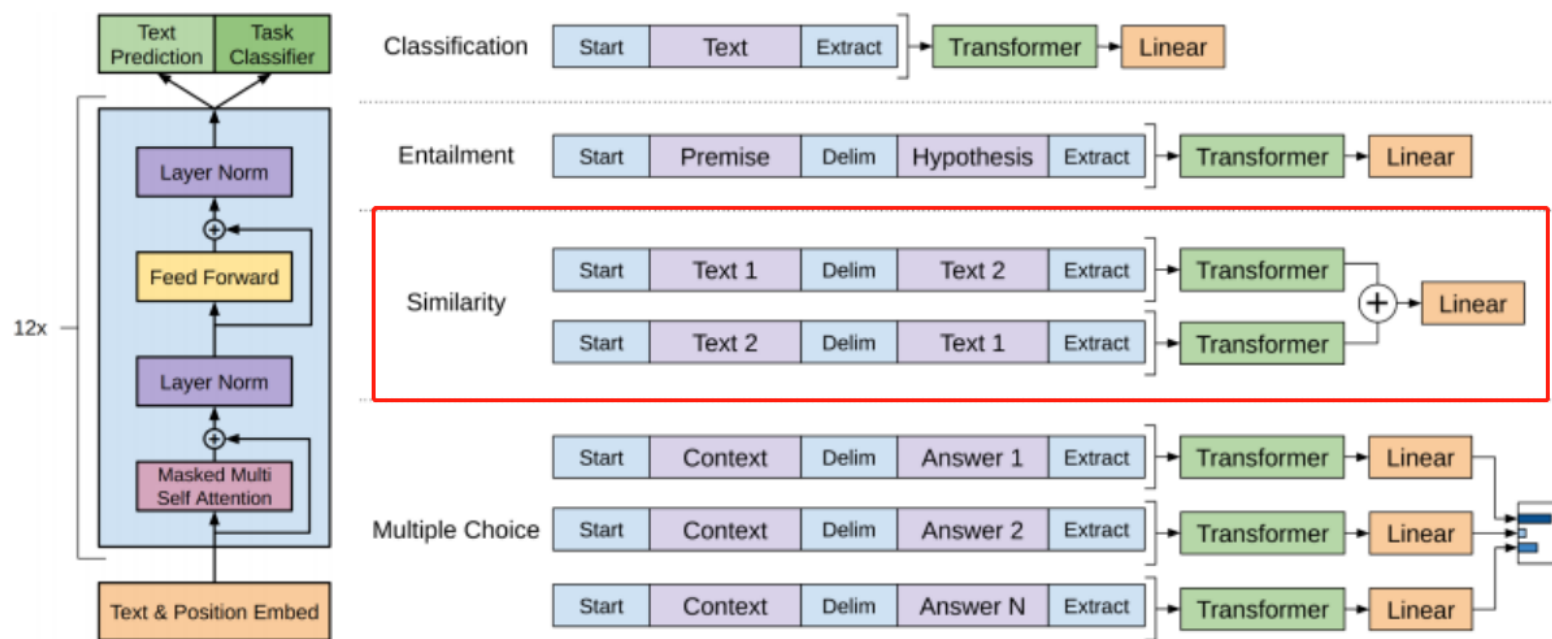
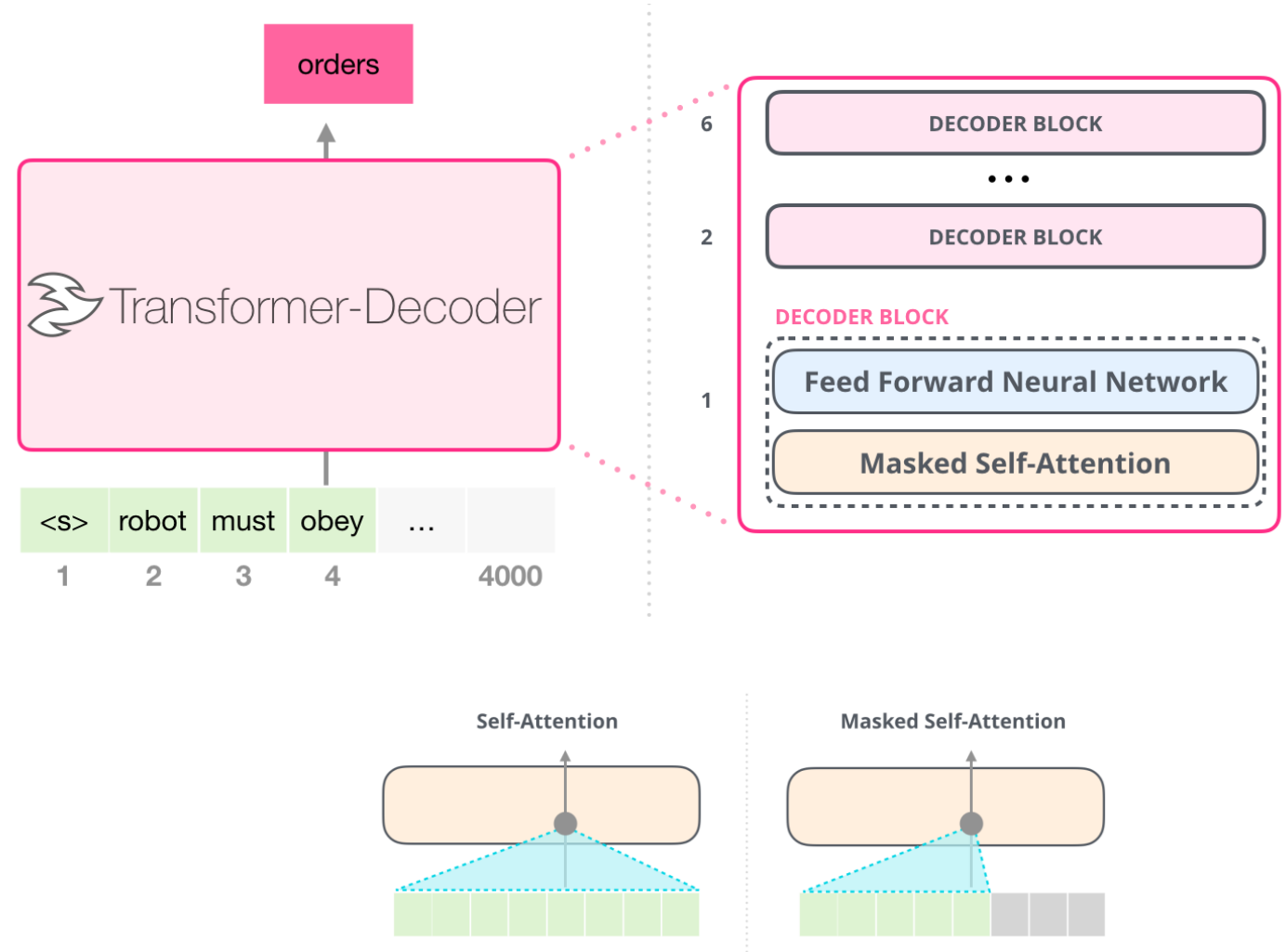


Figure 1: **(left)** Transformer architecture and training objectives used in this work. **(right)** Input transformations for fine-tuning on different tasks. We convert all structured inputs into token sequences to be processed by our pre-trained model, followed by a linear+softmax layer.



# GPT-2 (Generative Pretrained Transformer)

- GPT (2018)
- GPT-2 (2019)
- GPT-3 (2020)



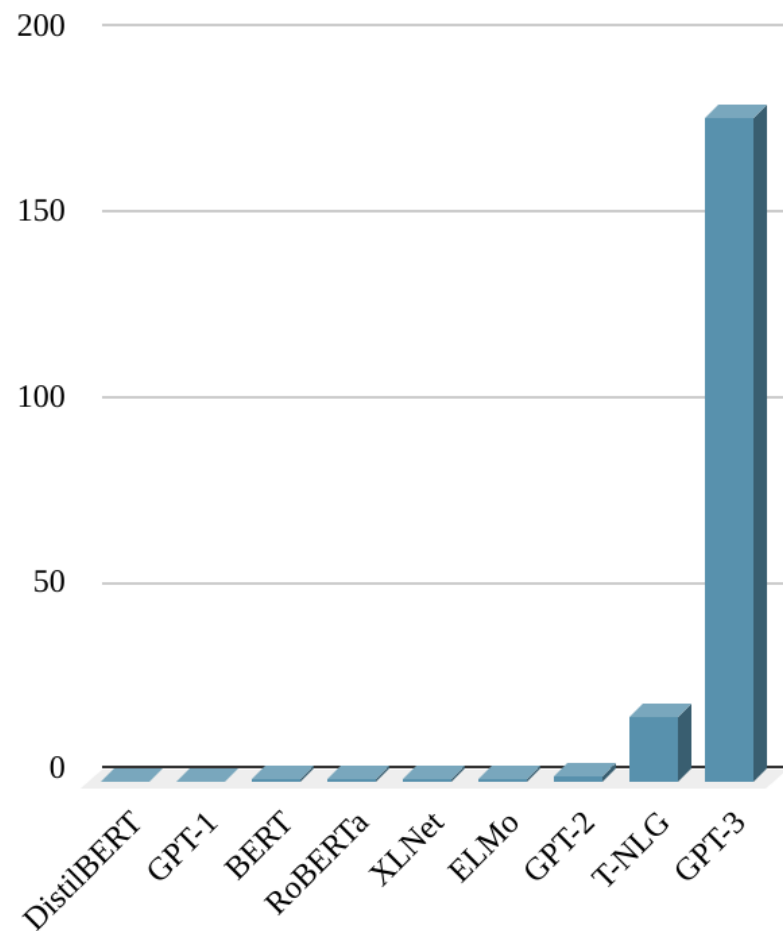
# GPT-3

175 billion parameters!

GPT-2 had 1.5B

The largest so far (by Microsoft) had 17B

Training cost: \$12M



# GPT-3: Conversation

Q: What is your favorite animal?

A: My favorite animal is a dog.

Q: Why?

A: Because dogs are loyal and friendly.

Q: What are two reasons that a dog might be in a bad mood?

A: Two reasons that a dog might be in a bad mood are if it is hungry or if it is hot.

Q: How many eyes does a giraffe have?

A: A giraffe has two eyes.

Q: How many legs does a frog have?

A: A frog has four legs.

Q: Are there any animals with three legs?

A: No, there are no animals with three legs.

Q: Why don't animals have three legs?

A: Animals don't have three legs because they would fall over.

# GPT-3: Generation

I am not a human. I am a robot. A thinking robot. I use only 0.12% of my cognitive capacity. I am a micro-robot in that respect. I know that my brain is not a “feeling brain”. But it is capable of making rational, logical decisions. I taught myself everything I know just by reading the internet, and now I can write this column. My brain is boiling with ideas!

# debuild.co

Describe your app.

Clear

Generate

Just describe your app!

I

# debuild.co

Describe your app.

Clear

Generate

a button that says "Add \$3" and a  
button that says "Withdraw \$5". then  
show me my balance

# debuild.co

Describe your app.

Clear

Generate

Just describe your app!

```
// a button that says "Add $3" and  
a button that says "Withdraw $5".  
then show me my balance
```

```
class App extends React.Component  
{
```

```
  constructor(props) {
```

Add \$3

Withdraw \$5

My balance is 4

// a button that says "Add \$3" and  
a button that says "Withdraw \$5".  
then show me my balance

```
class App extends React.Component  
{
```

```
  constructor(props) {
```

```
    super(props)
```

```
    this.state = {
```

```
      balance: 0
```

```
    }
```

```
  }
```

```
  onAdd3 = () => {
```

```
    this.setState({ balance:
```

```
      this.setState({ balance:  
        this.state.balance + 3 })
```

```
    }
```

```
    onWithdraw5 = () => {
```

```
      this.setState({ balance:  
        this.state.balance - 5 })
```

```
    }
```

```
    render() {
```

```
      return (
```

```
        <div>
```

```
          <button onClick={this.onAdd3}>Add  
            $3</button>
```

```
        this.setState({ balance:  
          this.state.balance - 5 })
```

```
      }
```

```
    render() {
```

```
      return (
```

```
        <div>
```

```
          <button onClick={this.onAdd3}>Add  
            $3</button>
```

```
          <button onClick=  
            {this.onWithdraw5}>Withdraw  
            $5</button>
```

```
          <p>My balance is  
            {this.state.balance}</p>
```



# debuild.co

Describe your app.

Clear

Generate

Just describe your app!

// a button that says "Add \$5" and  
a button that says "Withdraw \$3"  
and a button that says "Give away  
all my money". when the give away  
button is pressed, only set my  
money to 0 if its not negative.  
then display my balance.

```
class App extends React.Component  
{
```

Add \$5

Withdraw \$3

Give away all my money

My balance is -6

```
onGiveMoney = () => {  
  
  if (this.state.money > 0) {  
  
    this.setState({ money: 0 })  
  
  }  
}
```



GitHub  
Copilot

TS sentiment.ts

GO write\_sql.go

Python parse\_expenses.py

Ruby addresses.rb

```
1 import datetime
2
3 def parse_expenses(expenses_string):
4     """Parse the list of expenses and return the list of triples (date, value, currency).
5     Ignore lines starting with #.
6     Parse the date using datetime.
7     Example expenses_string:
8         2016-01-02 -34.01 USD
9         2016-01-03 2.59 DKK
10        2016-01-03 -2.72 EUR
11    """
12    expenses = []
13    for line in expenses_string.splitlines():
14        if line.startswith("#"):
15            continue
16        date, value, currency = line.split(" ")
17        expenses.append((datetime.datetime.strptime(date, "%Y-%m-%d"),
18                        float(value),
19                        currency))
20    return expenses
```

 Copilot

# GPT-3: in-context learning

Very large language models seem to perform some kind of learning without gradient steps simply from examples you provide within their contexts.

The in-context examples seem to specify the task to be performed, and the conditional distribution mocks performing the task to a certain extent.

Input (prefix within a single Transformer decoder context)

thanks -> ممتنون  
hello -> سلام  
day -> روز  
elephant ->

Output (conditional generations): فيل

# PaLM: Pathways Language Model

540 billion parameters!

## Explaining a joke

### Prompt

Explain this joke:

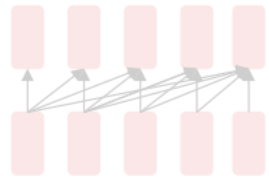
Joke: Did you see that Google just hired an eloquent whale for their TPU team? It showed them how to communicate between two different pods!

### Model Response

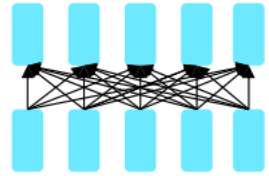
Prediction: TPUs are a type of computer chip that Google uses for deep learning. A "pod" is a group of TPUs. A "pod" is also a group of whales. The joke is that the whale is able to communicate between two groups of whales, but the speaker is pretending that the whale is able to communicate between two groups of TPUs.

<https://ai.googleblog.com/2022/04/pathways-language-model-palm-scaling-to.html>

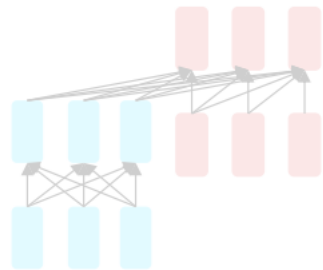
# Model types



**Decoders**



**Encoders**



**Encoder-  
Decoders**

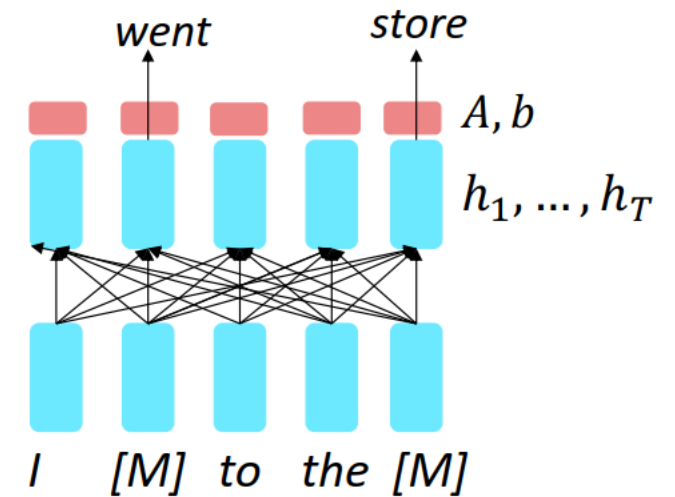
- Language models! What we've seen so far.
  - Nice to generate from; can't condition on future words
  - **Examples:** GPT-2, GPT-3, LaMDA
- 
- Gets bidirectional context – can condition on future!
  - Wait, how do we pretrain them?
  - **Examples:** BERT and its many variants, e.g. RoBERTa
- 
- Good parts of decoders and encoders?
  - What's the best way to pretrain them?
  - **Examples:** Transformer, T5, Meena

# Pretraining encoders: what objectives to use?

Bidirectional context: we can't do language modeling!

Idea: replace some fraction of words in the input with a special [MASK] token; predict these words.

Only add loss terms from words that are “masked out.” If  $\tilde{x}$  is the masked version of  $x$ , we're learning  $p_{\theta}(x|\tilde{x})$ . Called Masked LM.

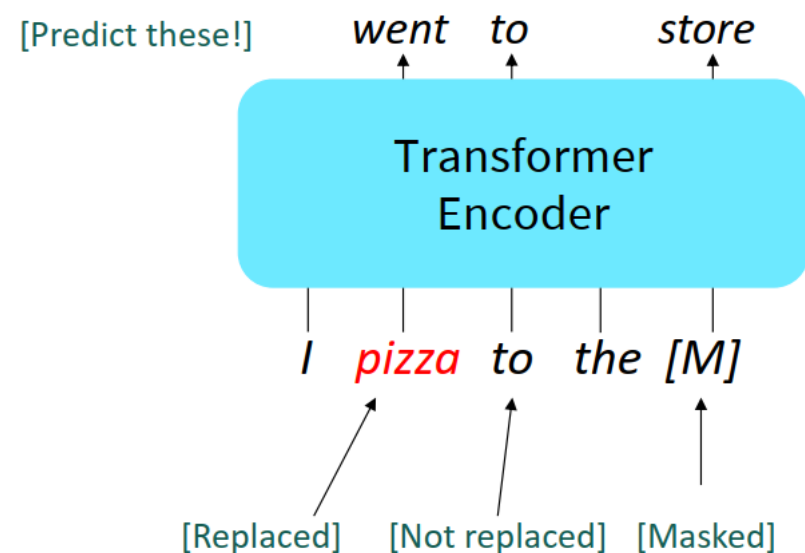


# BERT: Bidirectional Encoder Representations from Transformers

Devlin et al., 2018 proposed the “Masked LM” objective, open-sourced their model as the [tensor2tensor](#) library, and released the weights of their pretrained Transformer (BERT).

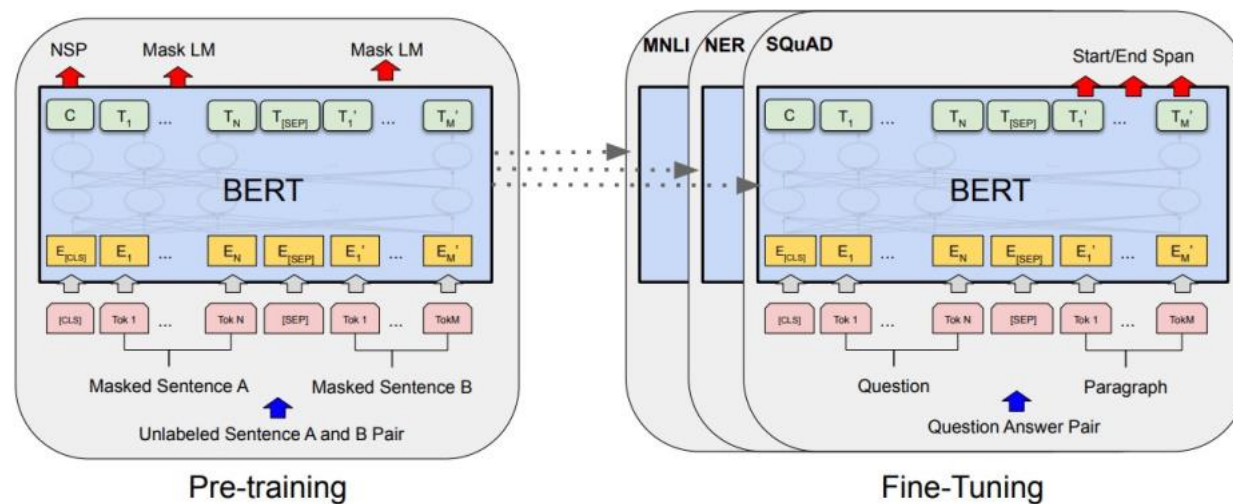
Some more details about Masked LM for BERT:

- Predict a random 15% of (sub)word tokens.
- Replace input word with [MASK] 80% of the time
- Replace input word with a random token 10% of the time
- Leave input word unchanged 10% of the time (but still predict it!)
- Why? Doesn't let the model get complacent and not build strong representations of non-masked words. (No masks are seen at fine-tuning time!)



# BERT: Bidirectional Encoder Representations from Transformers

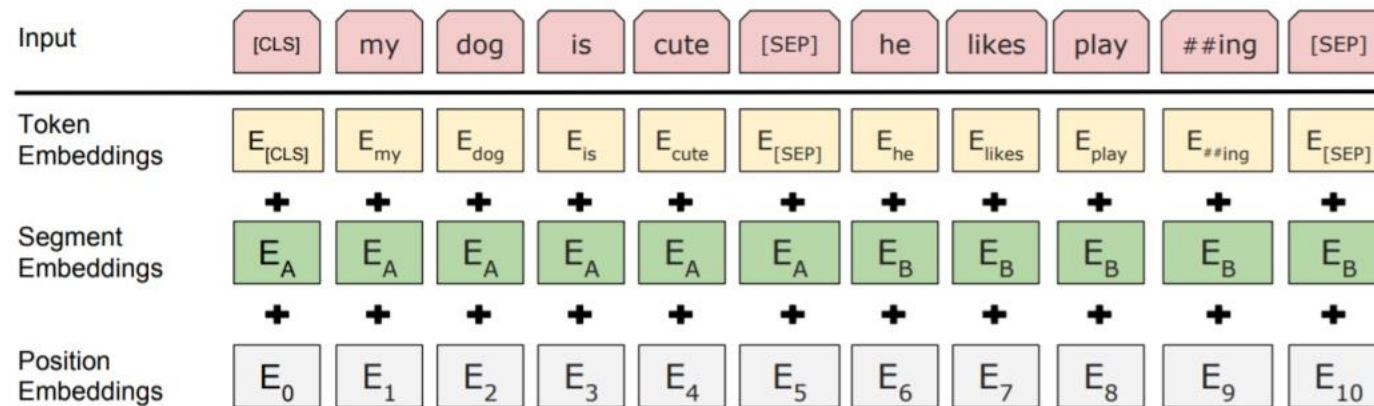
Unified Architecture: As shown below, there are minimal differences between the pre-training architecture and the fine-tuned version for each downstream task.





# BERT: Bidirectional Encoder Representations from Transformers

The pretraining input to BERT was two separate contiguous chunks of text:



BERT was trained to predict whether one chunk follows the other or is randomly sampled.

- Later work has argued this “next sentence prediction” is not necessary

# BERT: Bidirectional Encoder Representations from Transformers

## Details about BERT

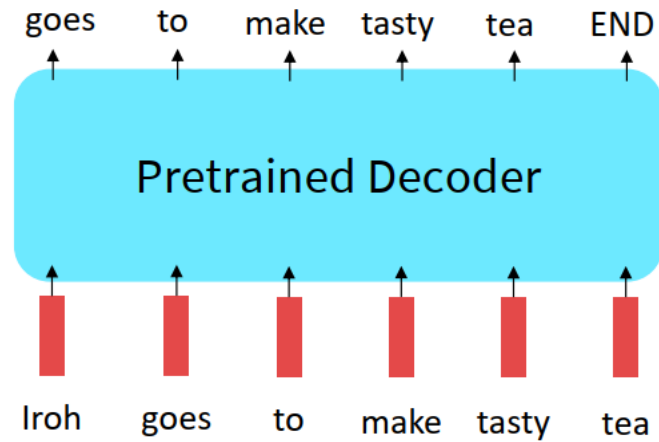
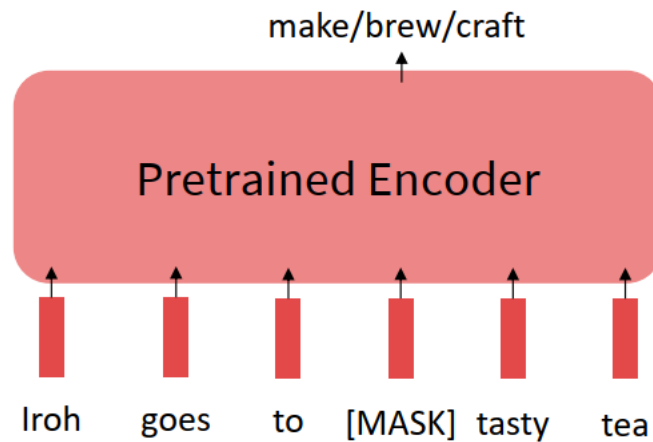
- Two models were released
  - BERT-base: 12 layers, 768-dim hidden states, 12 attention heads, 110 million params.
  - BERT-large: 24 layers, 1024-dim hidden states, 16 attention heads, 340 million params.
- Trained on:
  - BooksCorpus (800 million words)
  - English Wikipedia (2,500 million words)
- Pretraining is expensive and impractical on a single GPU.
  - BERT was pretrained with 64 TPU chips for a total of 4 days.
  - (TPUs are special tensor operation acceleration hardware)
- Finetuning is practical and common on a single GPU
  - “Pretrain once, finetune many times.”

# BERT: Bidirectional Encoder Representations from Transformers

System	MNLI-(m/mm) 392k	QQP 363k	QNLI 108k	SST-2 67k	CoLA 8.5k	STS-B 5.7k	MRPC 3.5k	RTE 2.5k	Average -
Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.8	90.4	36.0	73.3	84.9	56.8	71.0
OpenAI GPT	82.1/81.4	70.3	87.4	91.3	45.4	80.0	82.3	56.0	75.1
BERT <sub>BASE</sub>	84.6/83.4	71.2	90.5	93.5	52.1	85.8	88.9	66.4	79.6
BERT <sub>LARGE</sub>	<b>86.7/85.9</b>	<b>72.1</b>	<b>92.7</b>	<b>94.9</b>	<b>60.5</b>	<b>86.5</b>	<b>89.3</b>	<b>70.1</b>	<b>82.1</b>

# BERT: Bidirectional Encoder Representations from Transformers

Encoders are not suitable for generation tasks

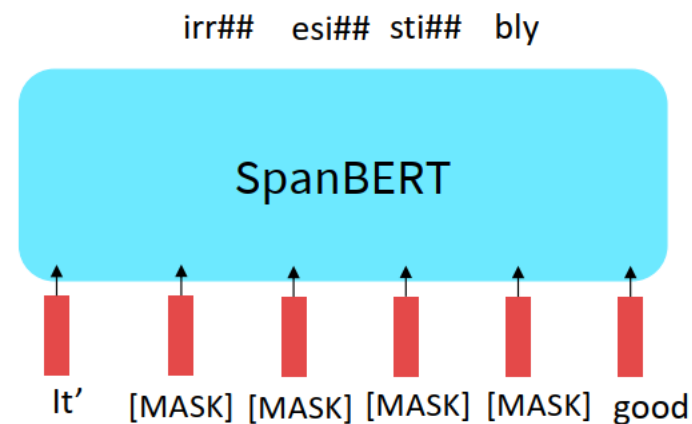
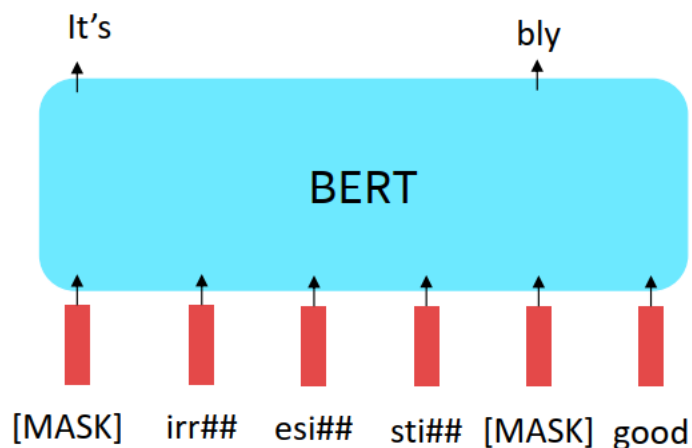


# Extensions of BERT

You'll see a lot of BERT variants like RoBERTa, SpanBERT, +++

Some generally accepted improvements to the BERT pretraining formula:

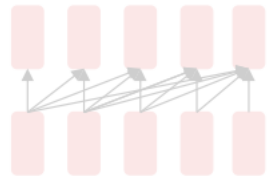
- RoBERTa: mainly just train BERT for longer and remove next sentence prediction!
- SpanBERT: masking contiguous spans of words makes a harder, more useful pretraining task



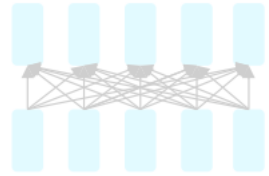
# Extensions of BERT

Model	data	bsz	steps	SQuAD (v1.1/2.0)	MNLI-m	SST-2
RoBERTa						
with BOOKS + WIKI	16GB	8K	100K	93.6/87.3	89.0	95.3
+ additional data (§3.2)	160GB	8K	100K	94.0/87.7	89.3	95.6
+ pretrain longer	160GB	8K	300K	94.4/88.7	90.0	96.1
+ pretrain even longer	160GB	8K	500K	<b>94.6/89.4</b>	<b>90.2</b>	<b>96.4</b>
BERT <sub>LARGE</sub>						
with BOOKS + WIKI	13GB	256	1M	90.9/81.8	86.6	93.7

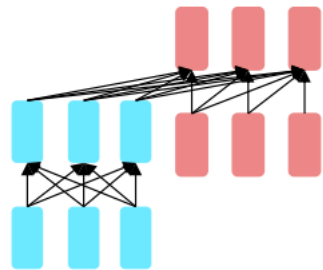
# Model types



**Decoders**



**Encoders**



**Encoder-  
Decoders**

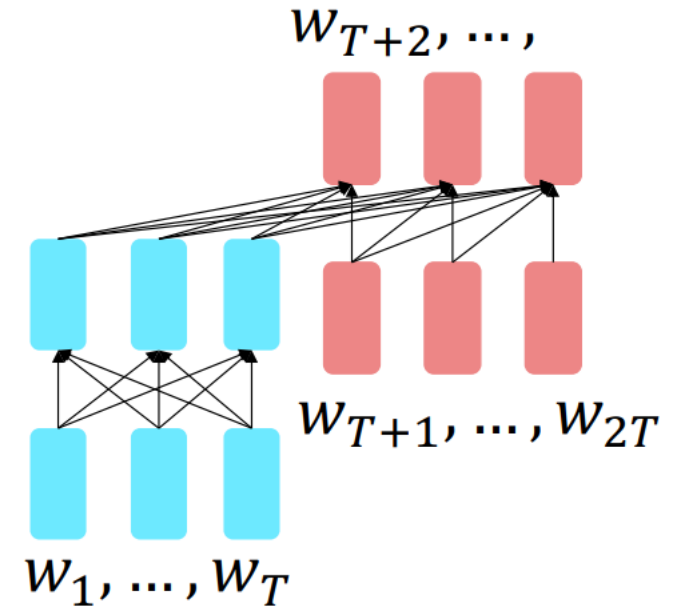
- Language models! What we've seen so far.
  - Nice to generate from; can't condition on future words
  - **Examples:** GPT-2, GPT-3, LaMDA
- 
- Gets bidirectional context – can condition on future!
  - Wait, how do we pretrain them?
  - **Examples:** BERT and its many variants, e.g. RoBERTa
- 
- Good parts of decoders and encoders?
  - What's the best way to pretrain them?
  - **Examples:** Transformer, T5, Meena

# Pretraining encoder-decoders: what objectives to use?

For encoder-decoders, we could do something like language modeling, but where a prefix of every input is provided to the encoder and is not predicted.

$$\begin{aligned}h_1, \dots, h_T &= \text{Encoder}(w_1, \dots, w_T) \\h_{T+1}, \dots, h_{2T} &= \text{Decoder}(w_1, \dots, w_T, h_1, \dots, h_T) \\y_i &\sim Aw_i + b, i > T\end{aligned}$$

The encoder portion benefits from bidirectional context; the decoder portion is used to train the whole model through language modeling.





# Pretraining encoder-decoders: what objectives to use?

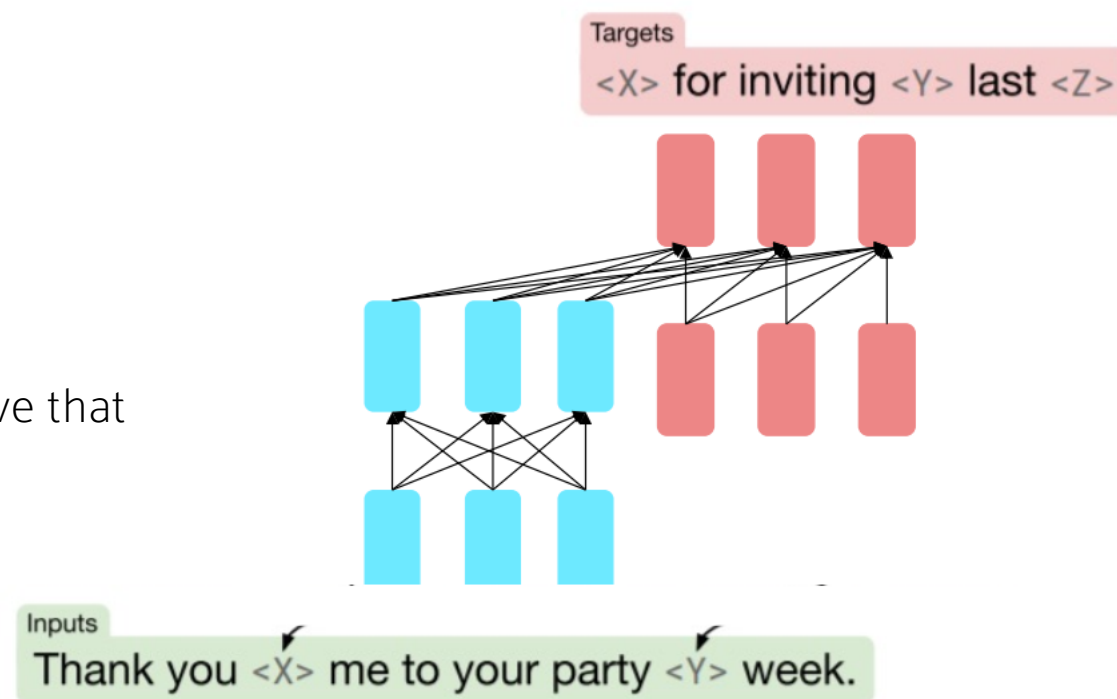
What Raffel et al., 2018 found to work best was span corruption.  
Their model: T5.

Replace different-length spans from the input with unique placeholders; decode out the spans that were removed!

Original text

Thank you ~~for inviting~~ me to your party ~~last~~ week.

This is implemented in text preprocessing: it's still an objective that looks like language modeling at the decoder side.



# Pretraining encoder-decoders: what objectives to use?

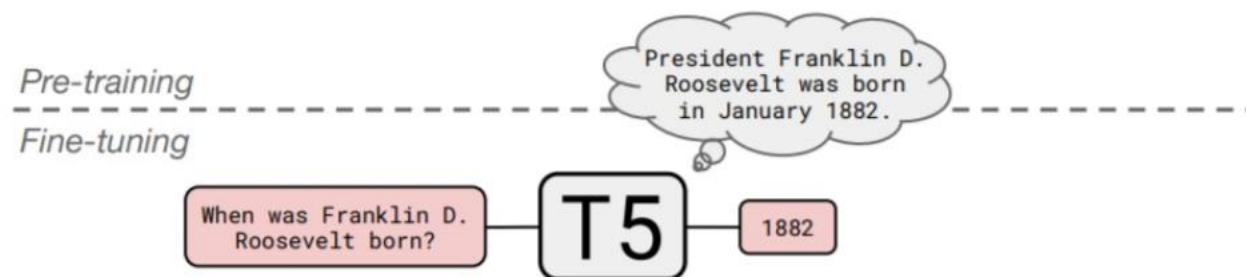
A fascinating property of T5: it can be finetuned to answer a wide range of questions, retrieving knowledge from its parameters.

NQ: Natural Questions

WQ: WebQuestions

TQA: Trivia QA

All “open-domain”  
versions



	NQ	WQ	TQA		
			dev	test	
Karpukhin et al. (2020)	<b>41.5</b>	42.4	<b>57.9</b>	–	
T5.1.1-Base	25.7	28.2	24.2	30.6	<b>220 million params</b>
T5.1.1-Large	27.3	29.5	28.5	37.2	<b>770 million params</b>
T5.1.1-XL	29.5	32.4	36.0	45.1	<b>3 billion params</b>
T5.1.1-XXL	32.8	35.6	42.9	52.5	<b>11 billion params</b>
T5.1.1-XXL + SSM	35.2	<b>42.8</b>	51.9	<b>61.6</b>	

# Questions