

Lecture 10: More about Transformers and Pretraining

Presenter: Chu Đình Đức

Breaking (Transformer) news

- AlphaCode (a pre-trained Transformer-based code generation model) achieved a top 54.3% rating on Codeforces programming competitions

1553_D. Backspace python pass Layer 18 stop

✓ 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

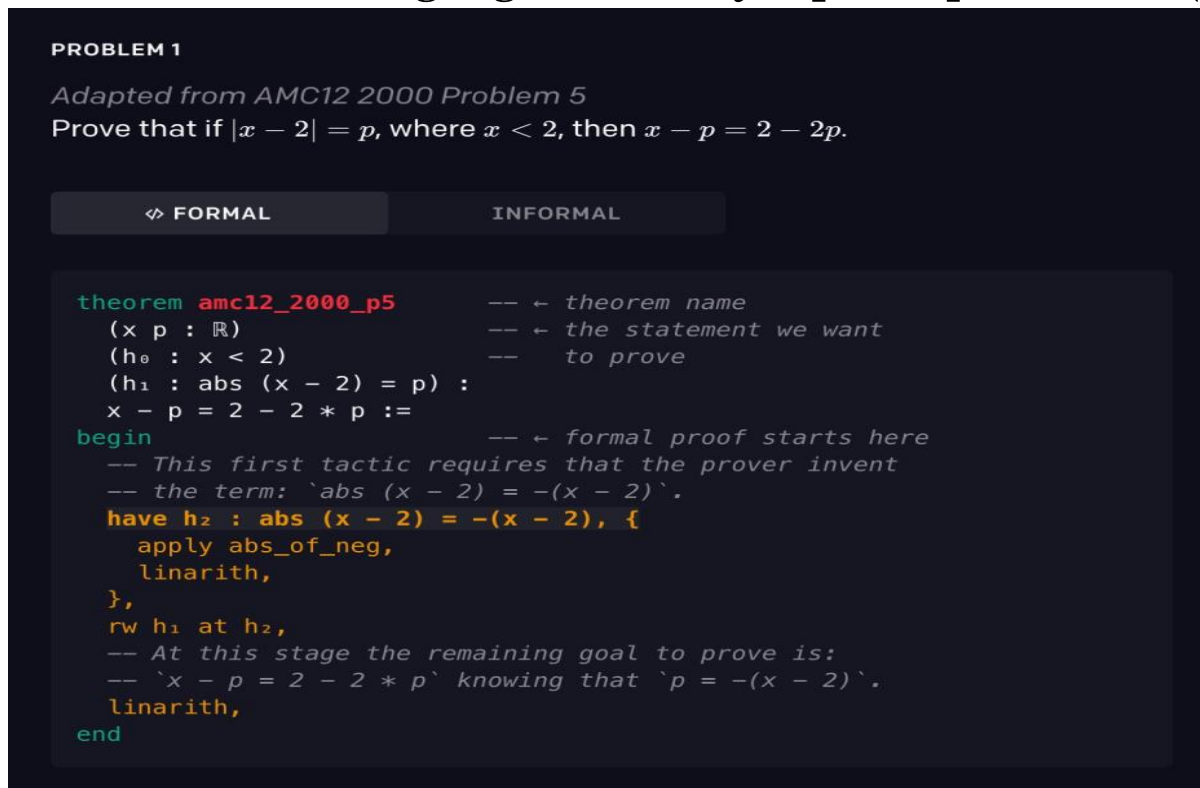
```
// RATING: 1288-
// TAGS: *special,implementation
// LANGUAGE IS python3-
// CORRECT SOLUTION-
// You are given two strings s and t, both consisting of lowercase English-
// letters. You are going to type the string s character by character, from the-
// first character to the last one.-
//
// When typing a character, instead of pressing the button corresponding to it,-
// you can press the "Backspace" button. It deletes the last character you have-
// typed among those that aren't deleted yet (or does nothing if there are no-
// characters in the current string). For example, if s is "abcbd" and you press-
// Backspace instead of typing the first and the fourth characters, you will get-
// the string "bd" (the first press of Backspace deletes no character, and the-
// second press deletes the character 'c'). Another example, if s is "abcaa" and-
// you press Backspace instead of the last two letters, then the resulting text-
// is "a".-
//
// Your task is to determine whether you can obtain the string t, if you type-
// the string s and press "Backspace" instead of typing several (maybe zero)-
// characters of s.-
//
// Input-
//
// The first line contains a single integer q (1 ≤ q ≤ 10^5) – the number of-
// test cases.-
//
// The first line of each test case contains the string s (1 ≤ |s| ≤ 10^5). Each-
// character of s is a lowercase English letter.-
//
// The second line of each test case contains the string t (1 ≤ |t| ≤ 10^5).-
// Each character of t is a lowercase English letter.-
//
```

Solution

```
t=int(input())-
for i in range(t)-
    s=input()-
    t=input()-
    a=[]-
    b=[]-
    for j in s:-
        a.append(j)-
    for j in t:-
        b.append(j)-
    a.reverse()-
    b.reverse()-
    e=[]-
    while len(b)!=0 and len(a)!=0:-
        if a[0]==b[0]:-
            e.append(b.pop(0))-
            a.pop(0)-
        elif a[0]!=b[0] and len(a)!=1:-
            a.pop(0)-
            a.pop(0)-
        elif a[0]!=b[0] and len(a)==1:-
            a.pop(0)-
    if len(b)==0:-
        print("YES")-
    else:-
        print("NO")-
```

More Breaking (Transformer) news

- Pre-trained Transformer-based theorem prover sets new state-of-the-art (41.2% vs. 29.3%) on a collection of challenging math Olympiad questions (miniF2F)



The screenshot shows a web-based interface for a theorem prover. At the top, it says "PROBLEM 1" and "Adapted from AMC12 2000 Problem 5". The problem statement is: "Prove that if $|x - 2| = p$, where $x < 2$, then $x - p = 2 - 2p$." Below the problem statement are two tabs: "FORMAL" and "INFORMAL". The "FORMAL" tab is selected, showing a Lean proof script. The script starts with a theorem declaration: `theorem amc12_2000_p5`. It then declares variables `(x p : ℝ)` and hypotheses `(h₀ : x < 2)` and `(h₁ : abs (x - 2) = p)`. The goal is to prove `x - p = 2 - 2 * p`. The proof begins with a `begin` block. The first tactic is `have h₂ : abs (x - 2) = -(x - 2), {`, followed by `apply abs_of_neg,` and `linarith,`. Then, `rw h₁ at h₂,` is used. Finally, `linarith,` is used to complete the proof, followed by `end`.

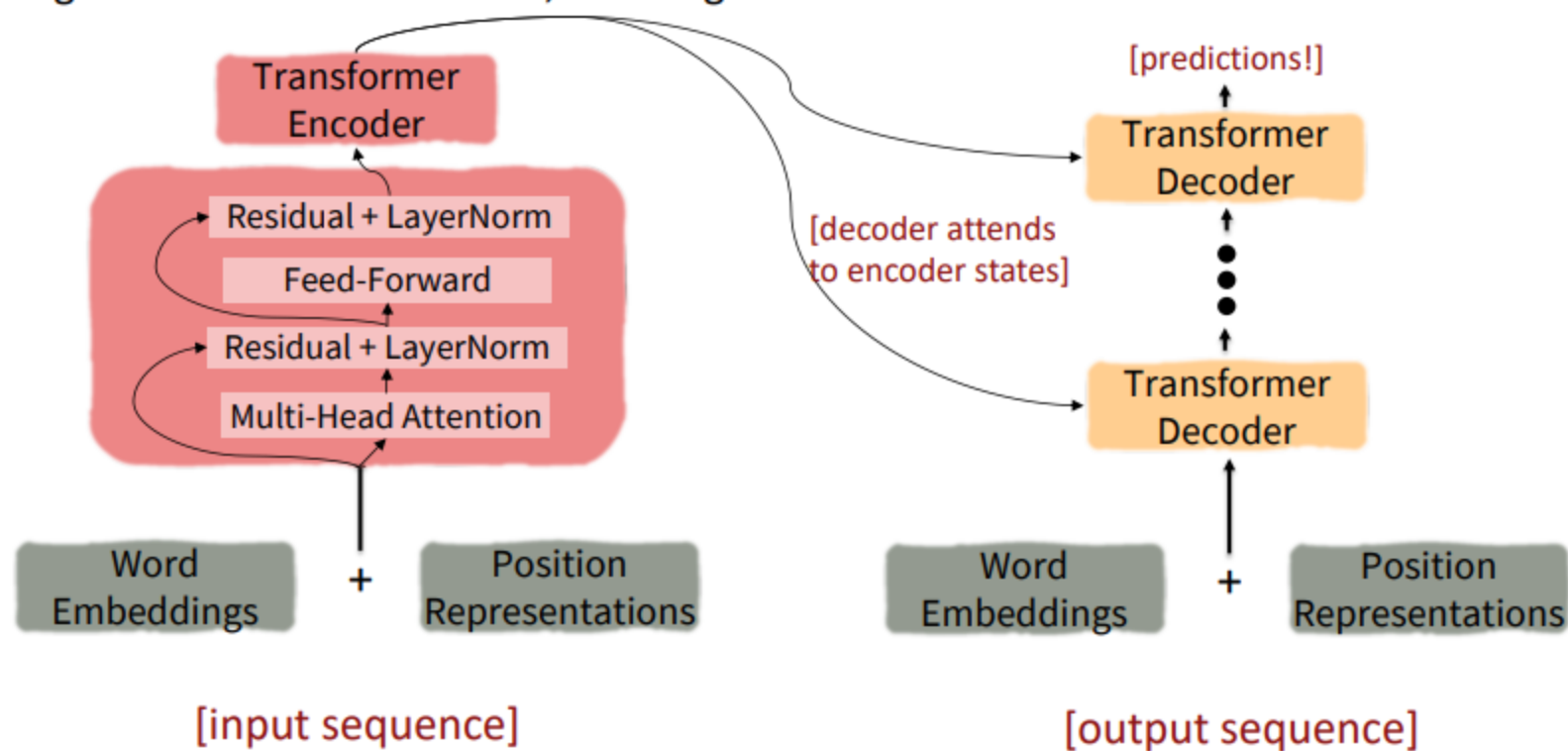
```
theorem amc12_2000_p5      -- ← theorem name
  (x p : ℝ)                -- ← the statement we want
  (h₀ : x < 2)              -- ← to prove
  (h₁ : abs (x - 2) = p) :
  x - p = 2 - 2 * p :=
begin                      -- ← formal proof starts here
  -- This first tactic requires that the prover invent
  -- 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
```

Lecture plan

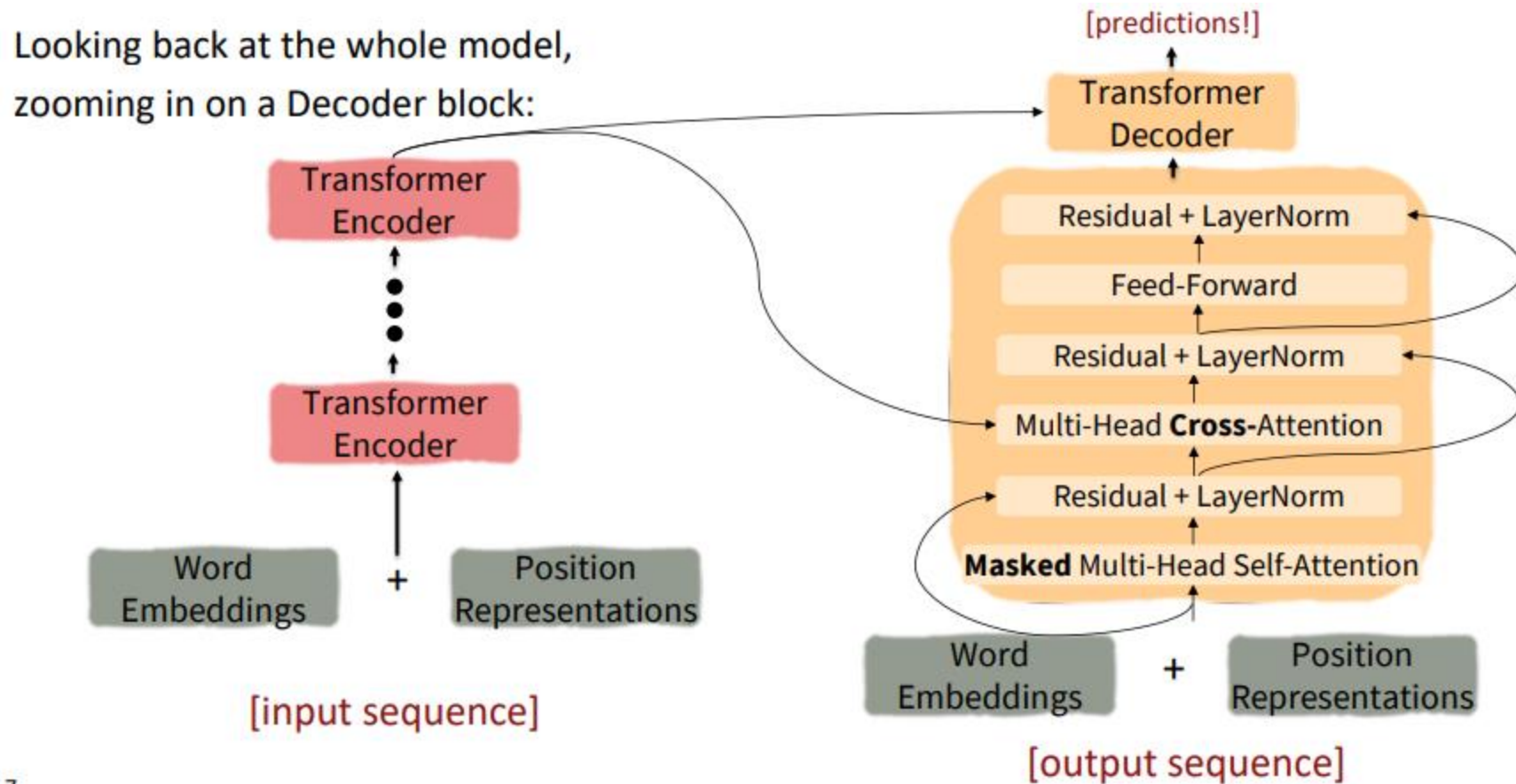
1. Quick review of Transformer model
2. Brief note on subword modeling
3. Motivating model pretraining from word embeddings
4. Model pretraining three ways
 1. Decoders
 2. Encoders
 3. Encoder-Decoders

1. Quick review of Transformer model

Looking back at the whole model, zooming in on an Encoder block:



1. Quick review of Transformer model



2. Brief note on subword modeling

- We assume a fixed vocab of tens of thousands of words, built from the training set. All novel words seen at test time are mapped to a single UNK
- However, finite vocabulary assumptions make even less sense in many languages
- Example
 - low, lower, lowest, strong, stronger, strongest – low, strong, er, est
 - Our goal is to save sequences of characters which appear with high frequency, let see BPE algorithm!

2. Brief note on subword modeling






- Subword modeling in NLP encompasses a wide range of methods for reasoning about structure below the word level (parts of words, characters, bytes)
 - The dominant modern paradigm is to learn a vocabulary of parts of words (subword tokens)
 - At training and testing time, each word is split into a sequence of known subwords
- BPE (Byte-Pair Encoding)
 - Start with a vocabulary containing only characters and an "end-of-word" symbol
 - Using a corpus of text, find the most common pair of adjacent characters "a,b"; add subword "ab" to the vocab
 - Replace instances of the character pair with the new subword, repeat until desired vocab size





2. Brief note on subword modeling

- BPE example:
 - ("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)
 - vocab: h u g p n b s
 - ("h" "u" "g", 10), ("p" "u" "g", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "u" "g" "s", 5)
 - "u" followed by "g" occurs $10+5+5=20$ times, "ug" is add to the vocabulary
 - vocab: h u g p n b s ug
 - ("h" "ug", 10), ("p" "ug", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "ug" "s", 5)
 - "u" followed by "n" occurs $12+4=16$ times, "un" is add to the vocabulary
 - vocab: vocab: h u g p n b s ug un
 - "h" followed by "ug" occurs $10+5=15$ times, "hug" is add to the vocabulary
 - vocab: vocab: h u g p n b s ug un hug
 - ("hug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("hug" "s", 5)
 - *repeat until desired vocab size*

2. Brief note on subword modeling

- Result

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

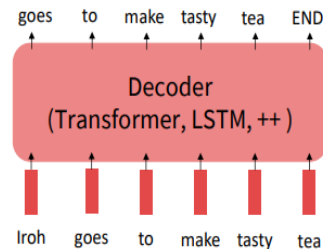
	word		vocab mapping	embedding
Common words	hat	→	hat	
	learn	→	learn	
Variations	taaaaasty	→	taa## aaa## sty	
misspellings	laern	→	la## ern	
novel items	Transformerify	→	Transformer## ify	

3. Motivating model pretraining from word embeddings

- Recall the language modeling task:
 - Model $P(w_t | w_{1:t-1})$, the probability distribution over words given their past contexts
 - There's lots of data for this (In English)
- Pretraining through language modeling
 - Train a neural network to perform language modeling on a large amount of text
 - Save the network parameters
- Pretraining can improve NLP applications by serving as parameter initialization

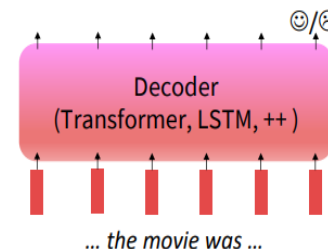
Step 1: Pretrain (on language modeling)

Lots of text; learn general things!



Step 2: Finetune (on your task)

Not many labels; adapt to the task!



4.1 Pretraining decoders

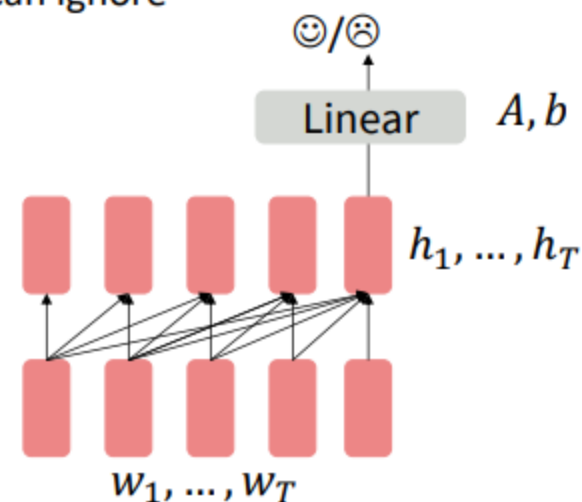
When using language model pretrained decoders, we can ignore that they were trained to model $p(w_t|w_{1:t-1})$.

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.



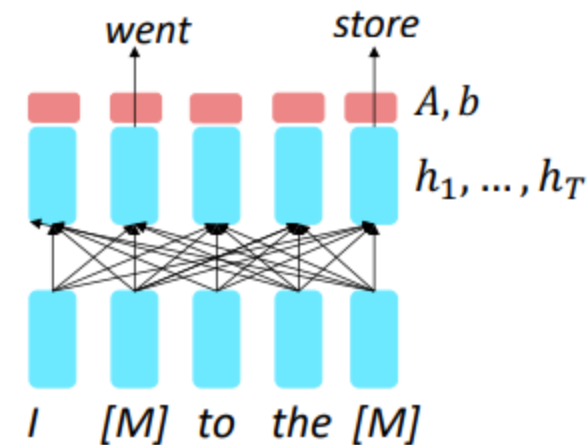
[Note how the linear layer hasn't been pretrained and must be learned from scratch.]

4.2 Pretraining encoders

So far, we've looked at language model pretraining. But **encoders get bidirectional context**, so 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**.

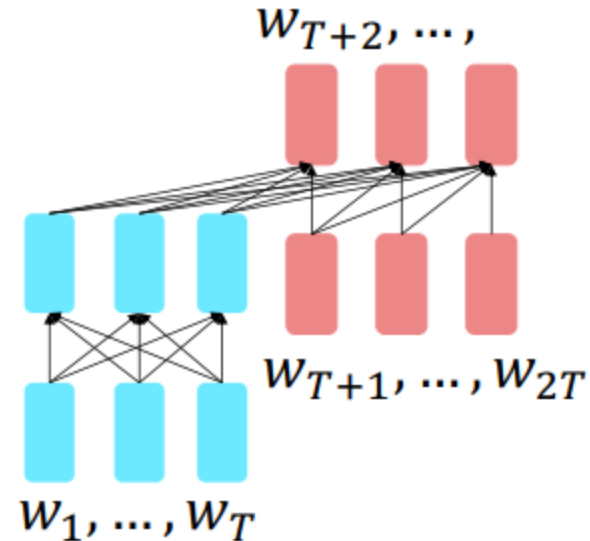


4.3 Pretraining encoder-decoders

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.



THANK YOU
Any question?