CSC-496/696: Natural Language Processing and Text as Data

Lecture 15: Transformers

Patrick Wu

Friday, October 25, 2024

Lecture Contents

Announcements

Midterm

- Midterm solutions are now uploaded
- I forgot to upload it last week...sorry.

Final Project

- A more detailed write-up of the final project is posted on Canvas
- Also includes information about the dataset for the task-driven version of the final project, if that is what you're interested in
- **I strongly suggest coming to office hours to discuss project ideas or approaches to the dataset that has been posted**

Final Project Data Posted

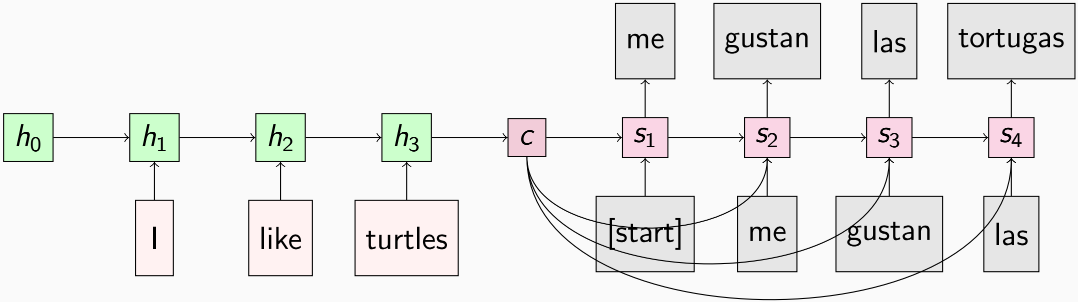
- The data for the task-driven version of the project is also posted
- You are also free to use this data for whatever research question you want
- Remember, the research question does not have to be groundbreaking. It doesn't even have to be novel—there is utility in replicating results using different approaches.

Assignment 3

- Due on Tuesday, October 29
- Future assignments will be less work to make time to complete the final project

Review of the Attention Mechanism

Encoder-Decoder with Context Sharing



$$h_t = f_W(x_t, h_{t-1})$$

$$s_t = g_U(y_{t-1}, s_{t-1}, c)$$

Context

- Why not just average h_1, h_2, and h_3?

Context

- Why not just average h_1, h_2, and h_3?
  - Effectively the same as c
  - We want to emphasize certain encoder hidden states over others at each decoding step
  - In other words, we want a dynamic context vector c_t for each decoder step

Context

- Why not just average h_1, h_2, and h_3?
  - Effectively the same as c
  - We want to emphasize certain encoder hidden states over others at each decoding step
  - In other words, we want a dynamic context vector c_t for each decoder step
- We use the *attention mechanism* to operationalize this idea

Intuition of Attention

- When you're reading, you connect previous information to what you're currently reading, but you don't weigh all previous information in the same way
- You are paying different amounts of *attention* to the previous information
- Your attention to previous information can also shift as you read further along

Intuition of Attention

- Similarly, the attention mechanism learns which encoder hidden states need more "attention" (weighed more) and which encoder hidden states need less "attention (weighed less)
- Dot-product attention uses the dot product to score how closely related a decoder hidden state is to every encoder hidden state
- We'll use the dot product to calculate a score between s_{t-1} and all encoder hidden states, h_j

$$\text{score}(s_{t-1}, h_j) = s_{t-1} \cdot h_j$$

Dot-Product Attention

After calculating this score for all encoder hidden states, we can then calculate the weights using the softmax

$$\alpha_{tj} = \text{softmax}(\text{score}(s_{t-1}, h_j)) = \frac{\exp(\text{score}(s_{t-1}, h_j)}{\sum_k \exp(\text{score}(s_{t-1}, h_j))}$$

α_{tj} is the weight between the decoder hidden state s_{t-1} and encoder hidden state h_j

Dot-Product Attention

Finally,

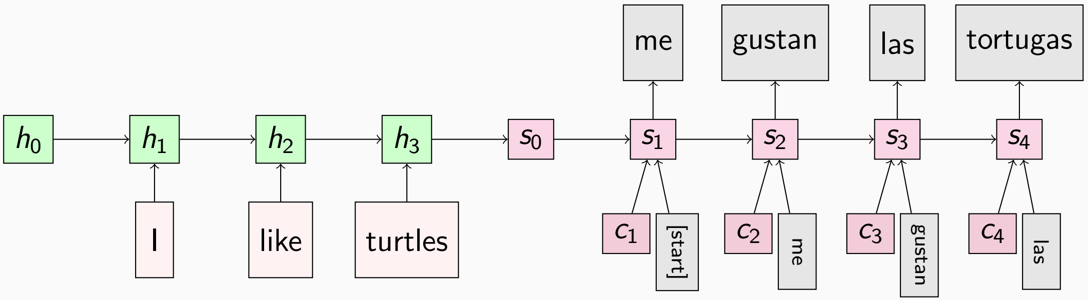
$$c_t = \sum_k \alpha_{tk} h_k$$

We can then calculate s_t as follows

$$s_t = g_U(y_{t-1}, s_{t-1}, c_t)$$

Notice that c_t is a weighted sum of *all* of the hidden state of the encoder. By backpropping through everything, we can learn which encoder states to pay more "attention" to at each decoding step.

Encoder-Decoder with Dot-Product Attention



Example: Calculating c_2

c_2 is the weighted sum of the encoder hidden states, h_1, h_2, and h_3. Note that the initial hidden state h_0 is **not** included in the calculation.

$$c_2 = \sum_{k=1}^{3} \alpha_{2k} h_k$$

We need to calculate α_{21}, α_{22}, and α_{23}.

Example: Calculating c_2

We calculate the dot products between s_1 (the previous hidden state of the decoder) and all the hidden states of the encoders.

$$\text{score}(s_1, h_1) = s_1 \cdot h_1 = \gamma_1$$
$$\text{score}(s_1, h_2) = s_1 \cdot h_2 = \gamma_2$$
$$\text{score}(s_1, h_3) = s_1 \cdot h_3 = \gamma_3$$

We can put these dot products into a vector:

$$\begin{bmatrix} \gamma_1 & \gamma_2 & \gamma_3 \end{bmatrix}$$

Example: Calculating c_2

We α values are calculated using the softmax over this vector

$$\alpha = \left[\frac{\exp(\gamma_1)}{\exp(\gamma_1)+\exp(\gamma_2)+\exp(\gamma_3)} \quad \frac{\exp(\gamma_2)}{\exp(\gamma_1)+\exp(\gamma_2)+\exp(\gamma_3)} \quad \frac{\exp(\gamma_3)}{\exp(\gamma_1)+\exp(\gamma_2)+\exp(\gamma_3)} \right]$$

Notice that this respects the rules of the softmax we had previously discussed: all numbers are positive, between 0 and 1, and all three numbers add up to 1.

The first element is α_{21}, the second element is α_{22}, and the third element is α_{23}.

Example: Calculating c_2

Then,

$$c_2 = \alpha_{21} h_1 + \alpha_{22} h_2 + \alpha_{23} h_3$$

Notice that h_1, h_2, and h_3 are vectors, so c_2 is a vector with the dimension of the encoder hidden states. In other words, c_2 **is a weighted sum of the encoder hidden states**.

c_2 is then used with the input "me" and s_1 in order to calculate s_2.

Attending

- When you read papers, you might run across the term “attending”

Attending

- When you read papers, you might run across the term "attending"
- This is just the mechanism by which each decoder state pays a different amount of "attention" (α weights) to each encoder hidden state

Attending

- When you read papers, you might run across the term "attending"
- This is just the mechanism by which each decoder state pays a different amount of "attention" (α weights) to each encoder hidden state
- Notice also that we attend to *every* encoder hidden state

Attending

- When you read papers, you might run across the term "attending"
- This is just the mechanism by which each decoder state pays a different amount of "attention" (α weights) to each encoder hidden state
- Notice also that we attend to *every* encoder hidden state
- No matter how inconsequential an encoder hidden state is, it will have *some* weight in the weighted average

Attending

- When you read papers, you might run across the term "attending"
- This is just the mechanism by which each decoder state pays a different amount of "attention" (α weights) to each encoder hidden state
- Notice also that we attend to *every* encoder hidden state
- No matter how inconsequential an encoder hidden state is, it will have *some* weight in the weighted average
- Called soft attention

Soft vs. Hard Attention

- Soft attention: attends to every encoder hidden state, even if the weight is very low
  - Advantage: differentiable, easy to train using backprop
  - Disadvantage: computationally more expensive
- Hard attention: focuses on specific parts, attending only to a specific subset of encoder hidden states
  - Advantage: more computationally efficient
  - Disadvantage: non-differentiable, which means we require reinforcement learning methods

Review Question 1

The dynamic context vector for each step of the decoder is a scalar value (a single number)

(A) True

(B) False

Review Question 2

An attention weight is a scalar value (a single number)

(A) True

(B) False

Review Question 3

What is the dimension of a context vector?

(A) The dimension of the hidden states of the decoder

(B) The dimension of the input embedding of the encoder

(C) The dimension of the output embedding of the encoder

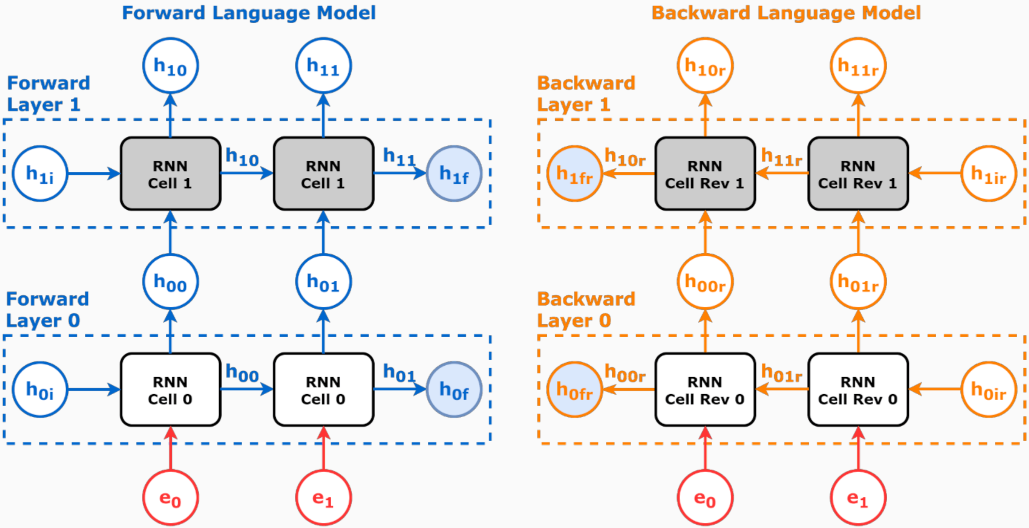
(D) The dimension of the hidden states of the encoder

Disadvantage of RNNs/LSTMs

- Researchers found *significant* improvements when using attention
- Widely explored in contexts such as text summarization, image captioning, and machine translation
- However, computations are still inherently sequential
    - Still must feed in one input at a time
    - We can parallelize using multiple sequences (batching), but we can't parallelize within the sequence
    - For example, we cannot calculate h_3 without calculating h_2

ELMo

- By no means are RNNs *bad*—still used for tasks like time-series predictions!
- In 2018, AI2 and the University of Washington released one of the first "large" language models called **E**mbeddings from **L**anguage **Mo**del, or ELMo
- ELMo is a model based on bidirectional LSTMs
- Because it was not an encoder-decoder framework, it **did not use attention**



Forward Language Model
Backward Language Model
Forward Layer 1
Forward Layer 0
Backward Layer 1
Backward Layer 0
h10
h11
h1i
RNN Cell 1
h1f
h00
h01
h0i
RNN Cell 0
h0f
e0
e1
h10r
h11r
h1fr
RNN Cell Rev 1
h1ir
h00r
h01r
h0fr
RNN Cell Rev 0
h0ir

Pretrained on Two Tasks

ELMo was trained on two tasks:

- Forward language modeling: ELMo predicts the next word in a sequence given the previous words

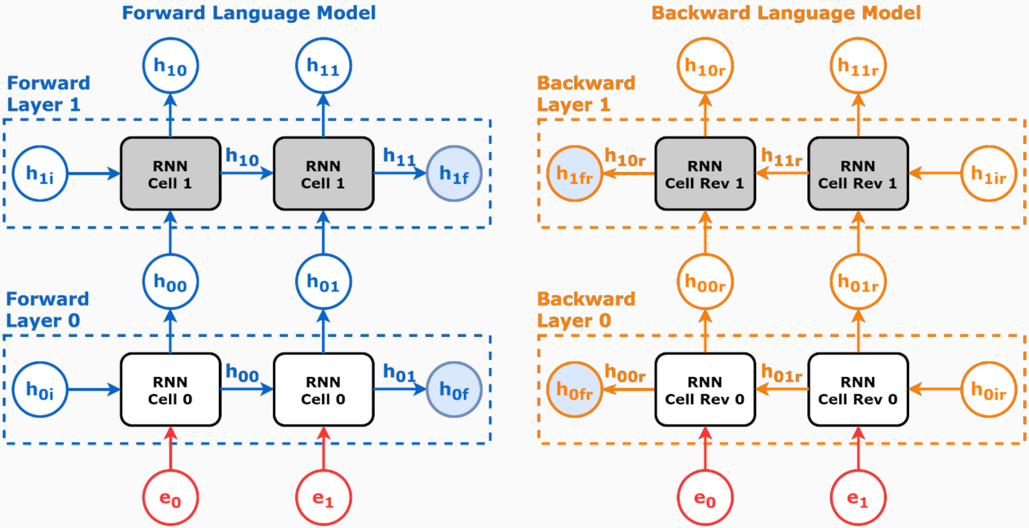
$$P(w_t | w_1, w_2, \ldots, w_{t-1})$$

- Backward language modeling: ELMo processes the text in reverse, from right to left, predicting the previous word based on future words in the sequence

$$P(w_t | w_{t+1}, w_{t+2}, \ldots, w_T)$$

Contextual Word Embeddings

- ELMo produces contextual word embeddings
- That is, the word embedding for the word “bank” changes based on its context
- In stark contrast to the previous word embedding methods, which produced a single word embedding for each unique word in the corpus


Forward Language Model
Backward Language Model
Forward Layer 1
Forward Layer 0
Backward Layer 1
Backward Layer 0
h10
h11
h1i
RNN Cell 1
h1f
h00
h01
h0i
RNN Cell 0
h0f
e0
e1
h10r
h11r
h1fr
RNN Cell Rev 1
h1ir
h00r
h01r
h0fr
RNN Cell Rev 0
h0ir

Attention is All You Need

Background

- Proposed in paper, "Attention is All You Need" by Vaswani et al. (2017)
  - Paper can be found here
- The original transformer, as described in the paper, is an encoder-decoder model
- Note: depending on the book or paper, people might capitalize the T in "transformer." Because transformers are so ubiquitous now in deep learning, most new papers do not capitalize the t.

Intuition of Transformers

- The original paper focused on machine translation
- Encoder module: processes the input text and encodes it into a series of vectors that capture the contextual information of the input
- Decoder module: takes these encoded vectors and generates the output text

Intuition of Transformers

- Much like an RNN, transformers take as input word embeddings
- A transformer **re-expresses** ("transforms") each word embedding as a weighted sum of all word embeddings in its context
- Recall: when using dot-product attention with RNNs, we calculated a context vector that would be unique for each decoder step by using a weighted sum of the hidden states of the encoder
- We'll use another form of attention here: **self-attention**

Intuition of Transformers

- The self-attention mechanism allows the model to weigh the importance of different words or tokens in a sequence relative to each other
- Allows the model to capture long-range dependencies and contextual relationships within the input data
  - Addresses a key weakness of RNNs that was partially addressed by attention
- In other words, self-attention allows each position in the input sequence to "attend to" all other positions in the same sequence when calculating the representation of a sequence

Why Self-Attention?

- Consider the sentence: "The chicken didn't cross the road because it was too tired."

Why Self-Attention?

- Consider the sentence: "The chicken didn't cross the road because it was too tired."
- Consider another sentence: "The chicken didn't cross the road because it was too wide."

Why Self-Attention?

- Consider the sentence: "The chicken didn't cross the road because it was too tired."
- Consider another sentence: "The chicken didn't cross the road because it was too wide."
- "It" refers to different nouns in each sentence.
  - In the first sentence, "it" refers to the chicken.
  - In the second sentence, "it" refers to the road.

Why Self-Attention?

- Consider the sentence: "The chicken didn't cross the road because it was too tired."
- Consider another sentence: "The chicken didn't cross the road because it was too wide."
- "It" refers to different nouns in each sentence.
  - In the first sentence, "it" refers to the chicken.
  - In the second sentence, "it" refers to the road.
- If we read the sentences from left to right, we get: The chicken didn't cross the road because it

Why Self-Attention?

- Consider the sentence: "The chicken didn't cross the road because it was too tired."
- Consider another sentence: "The chicken didn't cross the road because it was too wide."
- "It" refers to different nouns in each sentence.
  - In the first sentence, "it" refers to the chicken.
  - In the second sentence, "it" refers to the road.
- If we read the sentences from left to right, we get: The chicken didn't cross the road because it
- At this point, we don't know what "it" is referring to.

Why Self-Attention?

- Takeaway: language is not something that is *relationally* left to right
- The English language is read left to right, but words relate to each other in both directions
- One of the fundamental limitations of RNNs was that you must walk through the sequence one word at a time
- Attention had solved many of these issues, but again, we cannot calculate the context vectors c_t in parallel

Self-Attention (Simplified)

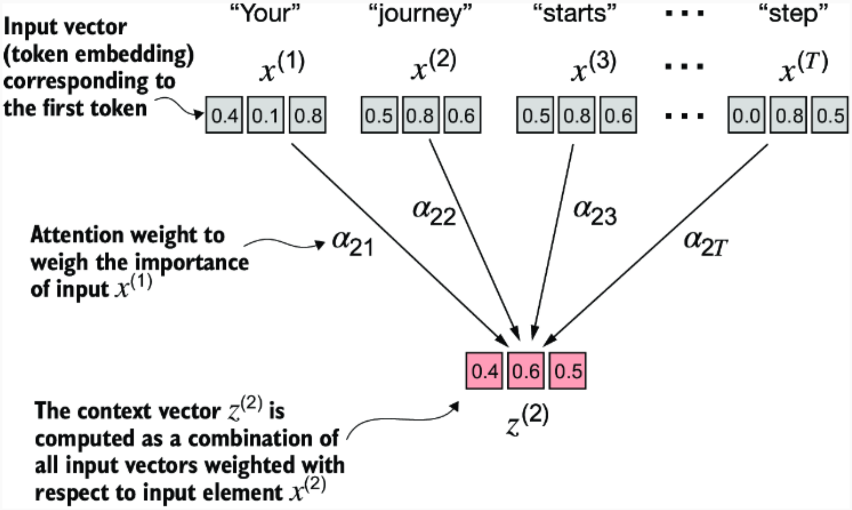


Figure from Raschka (2024). $z^{(2)}$ is calculated using attention weights with respect to

Self-Attention (Simplified)

Specifically (in this simplified version of self-attention),

$$z^{(2)} = \sum_{t=1}^{T} \alpha_{2t} x^{(t)}$$

where

$$\alpha_{2t} = \mathrm{softmax}(\mathrm{score}(x^{(2)}, x^{(t)}))$$

for all $t \in \{1, ..., T\}$, and

$$\mathrm{score}(x^{(2)}, x^{(t)}) = x^{(2)} \cdot x^{(t)}$$

Self-Attention (Simplified)

- This is why it's called *self*-attention—it is the attention the sentence "pays to itself"

Self-Attention (Simplified)

- This is why it's called *self*-attention—it is the attention the sentence "pays to itself"
- We end up with features $z^{(1)}, z^{(2)}, \ldots, z^{(T)}$ that are transformed features of the input features

Self-Attention (Simplified)

- This is why it's called *self*-attention—it is the attention the sentence "pays to itself"
- We end up with features $z^{(1)}, z^{(2)}, \ldots, z^{(T)}$ that are transformed features of the input features
- Each transformed feature is a weighted sum of all the other features

Self-Attention (Simplified)

- This is why it's called *self*-attention—it is the attention the sentence "pays to itself"
- We end up with features $z^{(1)}, z^{(2)}, \ldots, z^{(T)}$ that are transformed features of the input features
- Each transformed feature is a weighted sum of all the other features
- Model can learn *very* complex relationships between the input features!

Self-Attention: More Details

In self-attention, each input vector plays three roles:

- Query: the current element being compared to all other inputs
  - In the previous example, $x^{(2)}$ is the query
- Key: the input being compared to the query to determine the similarity weight
  - In the previous example, this is the x_t in $\mathrm{score}(x^{(2)}, x^{(t)}) = x^{(2)} \cdot x^{(t)}$
- Value: the input that gets weighted and summed up to compute the output for the current element
  - In the previous example, recall that $z^{(2)} = \sum_{t=1}^{T} \alpha_{2t} x^{(t)}$. $x^{(t)}$ is called the value.

Self-Attention (Full Version)

Because each vector plays three roles, we can add three weight matrices that will dramatically increase the ability to capture even more nuanced patterns

$$q^{(i)} = x^{(i)} W^q$$

$$k^{(i)} = x^{(i)} W^k$$

$$v^{(i)} = x^{(i)} W^v$$

Eventually, we will learn each weight matrix through backpropagation.

Self-Attention (Full Version)

$$q^{(i)} = x^{(i)} W^q; k^{(i)} = x^{(i)} W^k; v^{(i)} = x^{(i)} W^v$$

$$\text{score}(x^{(i)}, x^{(t)}) = \frac{q_i \cdot k_t}{\sqrt{d_k}}$$

$$\alpha_{it} = \text{softmax}(\text{score}(x^{(i)}, x^{(t)}) \quad \forall t \in \{1, \ldots, T\}$$

$$z^{(i)} = \sum_{t=1}^{T} \alpha_{it} x^{(t)}$$

Self-Attention (Full Version)

- Notice that we calculate the score by using the dot product, but also dividing by the square root of the key
- This is called scaled dot-product attention
- Prevents the dot product from blowing up
- But unlike using cosine similarity, it doesn't limit the dot product to between -1 and 1

Review Question 4

What is the primary function of self-attention?

(A) To attend to different parts of the input sequence by focusing on its own elements

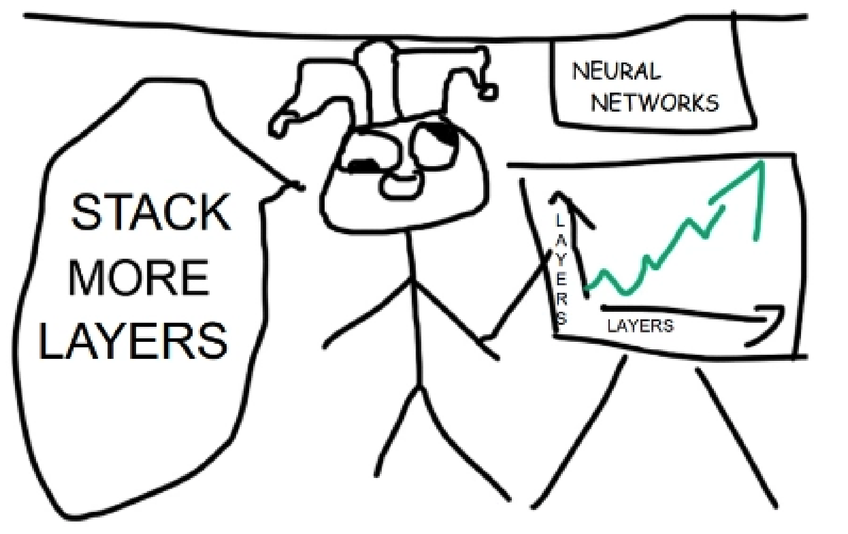
(B) To combine input sequences with random weights

(C) To ignore irrelevant information from the input and focus on a fixed part

(D) To compute the overall sentiment of the sentence

Multi-head Self-Attention

- What we just described is a **single attention head**
- In transformers, we can use multiple heads; we can even greater diversity in representing the inputs because the weight matrices associated with the query, key, and value can all be different
- Intuition: multi-head attention allows each self-attention head to learn distinct patterns within the sentence
- We can concatenate the output of each attention head, and then project it down to our original input dimensionality using a weight matrix W^O
- Equations (9.14) to (9.19) in SLP Ch. 9 shows all the mathematical details of multi-headed attention



STACK MORE LAYERS
NEURAL NETWORKS
LAYERS
LAYERS

Unmasked vs. Masked Attention

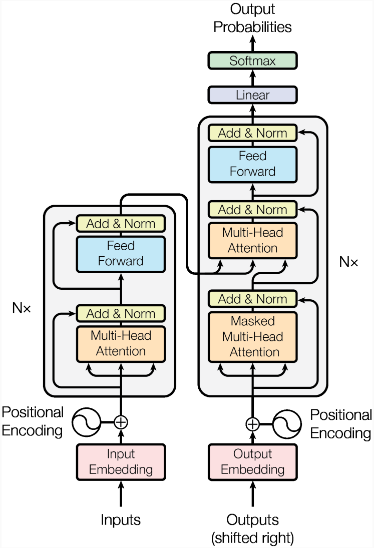
- In the above, we allowed each query to attend to all preceding *and* subsequent outputs
- But sometimes we don't want to do that—we want the model to only learn from the preceding input
    - Especially important if we are training generative models!!
- We can use masked attention, where the model is only allowed to attend to previous inputs
- Using our running example, $z^{(2)}$ would only be calculated using $x^{(2)}$ and $x^{(1)}$

Transformers

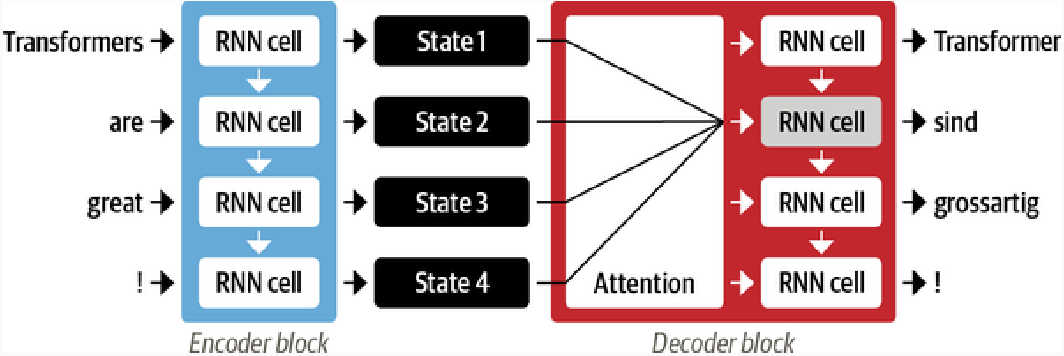
Self-Attention and Transformers

- The paper is called “Attention is All You Need” because the authors show that you only need the self-attention mechanism to process sequential data
- By dispensing with recurrence, we can parallelize calculations, yielding much faster training time
- Self-attention is the heart of transformers

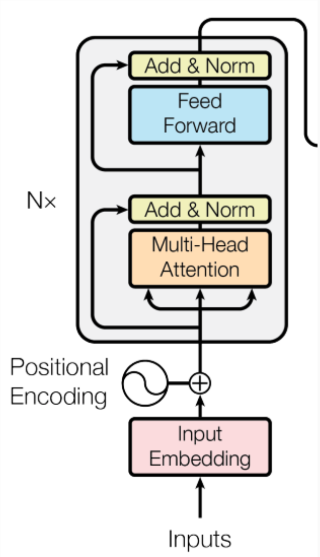
Transformers (Full Diagram)



Encoder-Decoder Architecture of the Transformer



Transformer Encoder



Encoder: Multi-Head Self-Attention

This is just what we talked about in the first half of this lecture

Encoder: Feed Forward

- The feed-forward sublayer in the encoder (and the decoder) is a simple two-layer fully connected neural network
- But instead of processing the whole sequence of embeddings as a single vector, it processes each embedding independently
- The hypothesis is that this is where most of the capacity and memorization happens
- Typically, transforms the input features from k elements to $4k$ elements, and then back down to k elements

Encoder: Add & Norm

- A transformer uses skip (or residual) connections
  - Won't talk about the details in this class, but it is a concept borrowed from computer vision, which allowed convolutional neural networks to be rapidly scaled up
- Layer normalization normalizes each input in the batch to have zero mean and a variance of 1
- Different models move around the layer normalization for training efficiency purposes

Encoder: Positional Encoding

- Recall that one issue is that unmasked self-attention does not recognize word order—it just calculates the relevancy between a given query and the keys

Encoder: Positional Encoding

- Recall that one issue is that unmasked self-attention does not recognize word order—it just calculates the relevancy between a given query and the keys
- We're going to add more embeddings: augment the token embeddings with a position-dependent vector

Encoder: Positional Encoding

- Recall that one issue is that unmasked self-attention does not recognize word order—it just calculates the relevancy between a given query and the keys
- We're going to add more embeddings: augment the token embeddings with a position-dependent vector
- These embeddings can learn information about the positions of words

Encoder: Positional Encoding

- Recall that one issue is that unmasked self-attention does not recognize word order—it just calculates the relevancy between a given query and the keys
- We're going to add more embeddings: augment the token embeddings with a position-dependent vector
- These embeddings can learn information about the positions of words
- In fancier terms, self-attention and the feed-forward layers are said to be permutation equivariant

Encoder: Positional Encoding

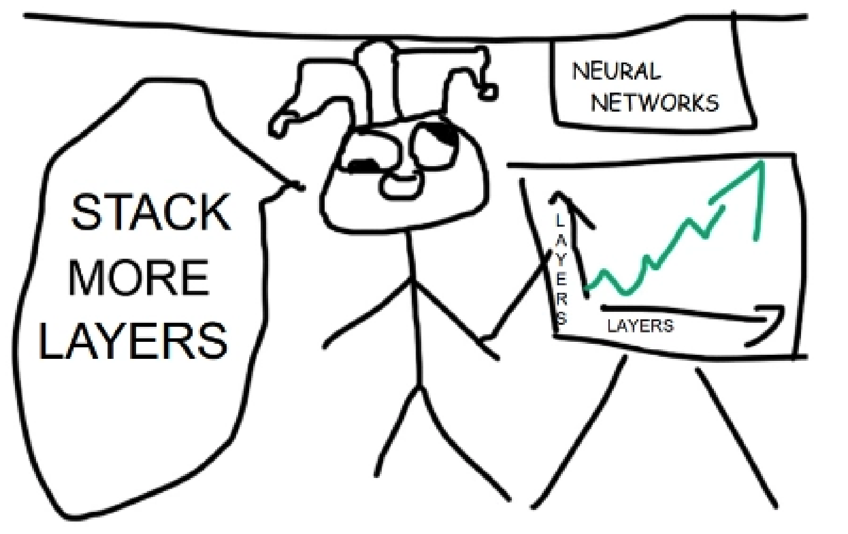
- Recall that one issue is that unmasked self-attention does not recognize word order—it just calculates the relevancy between a given query and the keys
- We're going to add more embeddings: augment the token embeddings with a position-dependent vector
- These embeddings can learn information about the positions of words
- In fancier terms, self-attention and the feed-forward layers are said to be permutation equivariant
- Positional embeddings break this permutation equivariance

Encoder: Positional Encoding

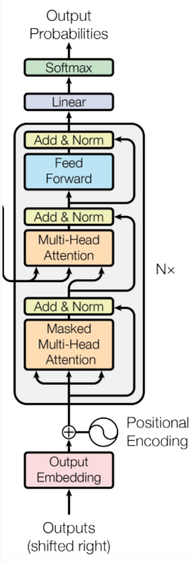
- Recall that one issue is that unmasked self-attention does not recognize word order—it just calculates the relevancy between a given query and the keys
- We're going to add more embeddings: augment the token embeddings with a position-dependent vector
- These embeddings can learn information about the positions of words
- In fancier terms, self-attention and the feed-forward layers are said to be permutation equivariant
- Positional embeddings break this permutation equivariance
- There are a few ways to calculate or learn these position embeddings, but we won't talk about these in this class

Stacking Encoders

- You can also *stack* encoders
- That is, you input your token embeddings into an encoder, go through the entire transformer encoder *block*, and then feed the output of that transformer encoder block into another transformer encoder block


STACK MORE LAYERS
NEURAL NETWORKS
LAYERS
LAYERS

Transformer Decoder



Decoder: Masked Multi-Head Self-Attention

- Ensures that the tokens we generate at each time step are only based on the past outputs and the current token being predicted
- We don't want the decoder to cheat by "peeking" ahead
- In other words, the current input can only attend to the previous inputs and itself, but not the subsequent inputs

Decoder: Encoder-Decoder Attention Layer

- Performs multi-head attention over the output values of the encoder stack, which act as the keys and values
- The intermediate representations of the decoder act as the queries
- This is the part of the decoder that relates the encoder output with the decoder input
    - Where language translation occurs!

Stacking Decoders

- You can also stack decoders as well
- Operates in a similar logic as stacking encoders

Transformers

- And that's it! We've covered transformers!
- Even though it looks complicated, we've now broken down (conceptually, at least) all the details of both the encoder and decoder
- Original paper focused on machine translation
- But as the paper captured the imaginations of many researchers, people realized that you could model language using only encoders or decoders

Large Language Models

- Modern large language models (LLMs) are all based on transformers
- There are encoder-only models (e.g., BERT), decoder-only models (e.g., GPT), and encoder-decoder models (e.g., T5)
- Decoder-only models are currently the center of attention at this moment, but all approaches dramatically improved benchmark results
- Large language models also allow us to *pretrain* a model on huge amounts of data, which then allows us to either use them directly for tasks of interest or we can *finetune* a model with a small amount of labeled data