Natural Language Processing

Class 5: The Transformer

Adam Faulkner

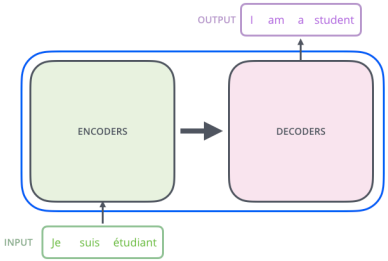
February 17, 2026

The Transformer: The neural architecture that made contemporary AI possible

- The standard architecture for building LLMs
- THE original Transformer architecture introduced in Attention is all you Need (2017), specifically the **Encoder-Decoder** — also called **sequence-to-sequence** — Transformer architecture WAS originally proposed for Machine Translation and is rarely used today
- Our focus will be **Decoder-only**, or **GPT**-style, Transformer which is the most widely used Transformer architecture
- In class 6 we'll cover **Encoder-only**, or **BERT**-style, Transformers and the finetuning paradigm
- In class, we'll make heavy-use of the Transformer Explainer to step through each step of a GPT-style Transformer.

The Transformer: The Encoder-Decoder architecture

- The **Encoder-Decoder** architecture was originally developed for the task of Machine Translation



Machine Translation

- Given a sentence in a **source** language, the Machine Translation (MT) task is then to generate a corresponding sentence in a **target** language.
- For example, given the French sentence

  `Je suis étudiant`

  we want to translate it to the English

  `I am a student.`

Machine Translation

- MT uses supervised machine learning: at training time the system is given a large set of parallel sentences (each sentence in a source language matched with a sentence in the target language), and learns to map source sentences into target sentences.
- These parallel corpora are taken from multiple sources — the human-translated sessions of the UN is a popular parallel corpus for MT.

Machine Translation

- In the transformer-based approach to MT, the encoder takes the input words $x = [x_1, \ldots, x_n]$ and produces an intermediate context h. At decoding time, the system takes h

$$\mathbf{h} = encoder(x)$$

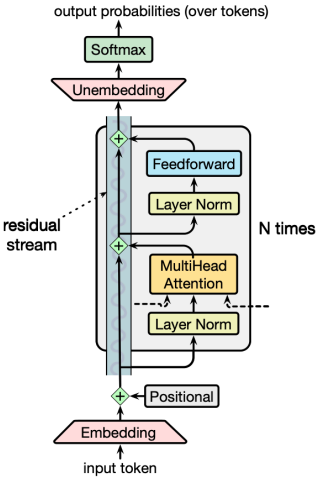
and, word by word, generates the output y:

$$y_{t+1} = decoder(\mathbf{h}, y_1, \ldots, y_t) \quad \forall t \in [1, \ldots, m]$$

Introduction to the Transformer Architecture

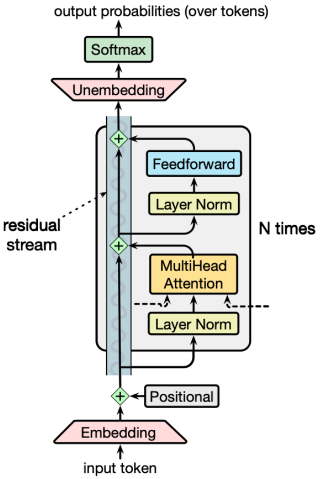
- The core architecture of LLMs
- Models left-to-right (sometimes called causal or autoregressive) language modeling, in which we are given a sequence of input tokens and predict output tokens one by one by conditioning on the prior context.

Introduction to the Transformer Architecture



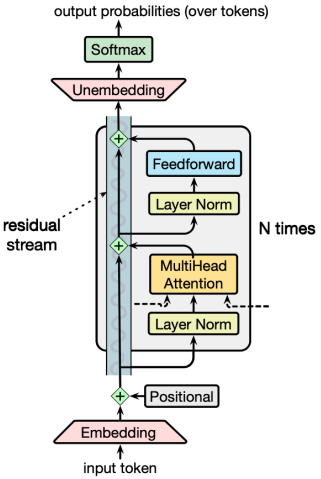
- This visual sketches the transformer architecture following a single token as it is passed up through the layers of the network.

Introduction to the Transformer Architecture



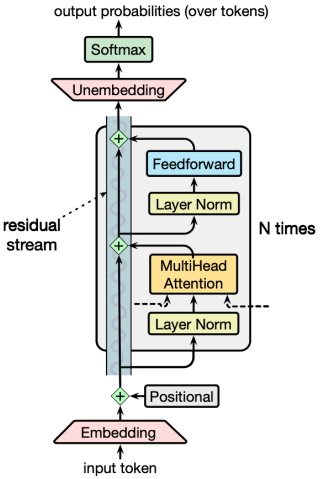
- Each token is first converted to an embedding from the embedding matrix. This is a linear layer that maps a token id to a vector embedding representing that token.

Introduction to the Transformer Architecture



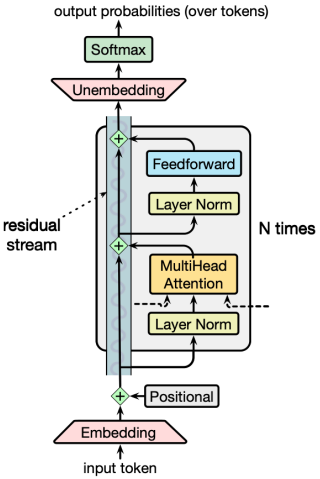
- Transformers also have a special mechanism for encoding the position/index of the token in the input string, which is simply added to the embedding. The resulting embedding represents both the word and its position. and is then passed through a set of N transformer blocks

Introduction to the Transformer Architecture



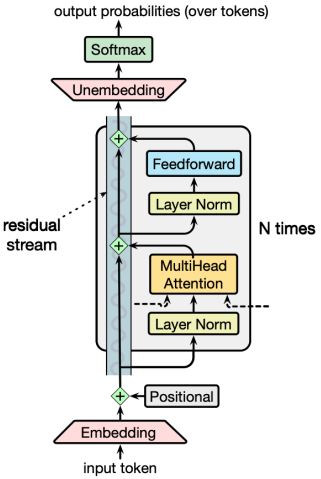
- Think of each of these transformer blocks as part of a stream in which the input embedding is directly passed up to the output, while simultaneously being enriched by the application of various processing modules: the multi-head attention layer, feedforward networks and the layer normalization. The value of the stream at any layer is the sum of the original embedding and all the outputs from all the previous layers and blocks

Introduction to the Transformer Architecture



- The core component is the multi-head attention layer, also called a self-attention layer. We'll describe attention in more detail in a bit.

Introduction to the Transformer Architecture



- After the N transformer blocks we take the output embedding that is produced by the final transformer block, pass it through an linear unembedding matrix U and then a softmax over the vocabulary to generate a distribution over possible next tokens. These last two components (the unembedding matrix and the softmax) are sometimes called the language modeling head

Why we need contextual representations of words

- Recall that, in **word2vec**-style embeddings, the representation of a word's meaning is always the same vector irrespective of the context.
- The word *chicken*, for example, is always represented by the same fixed vector. So, a static vector for the word *it* might somehow encode that this is a pronoun used for animals and inanimate entities.
- But, in context, *it* has a much richer meaning.
  - **The chicken** didn't cross the road because **it** was too tired.
  - The chicken didn't cross **the road** because **it** was too wide.
- *it* refers to the chicken in the first sentence but refers to *the road* in the second.
- We need a way to capture **both** meanings of *it*.

Why we need contextual representations of words

- Last week we learned about the concept of **polysemy** ("many meanings").
- We use context to identify which sense of a word is being conveyed in a sentence.
  - I walked along **the pond**, and noticed one of the trees along **the bank**.
- The context *the pond* and *the trees* tells us that *the bank* here refers to to the side of a pond or river and not a financial institution.

Contextual Embeddings

- The point of these examples is that these contextual words that help us compute the meaning of words in context can be quite far away in the sentence or paragraph.
- Transformers can build contextual representations of word meaning, **contextual embeddings**, by integrating the meaning of these helpful contextual words.
- In a contextual embeddings transformer, layer by layer, we build up richer and richer contextualized representations of the meanings of input tokens.
- At each layer, we compute the representation of a token i by combining information about i from the previous layer with information about the neighboring tokens to produce a contextualized representation for each word at each position.

Contextual Embeddings

- The Transformer is a neural network with a specific structure that includes a mechanism called **self-attention** or **multi-head attention**.
- **Attention** is the mechanism in the transformer that weighs and combines the representations from appropriate other tokens in the context from layer *k-1* to build the representation for tokens in layer *k*
- Self-attention is the method the Transformer uses to bake the understanding of other relevant words into the one we're currently processing

Contextual Embeddings

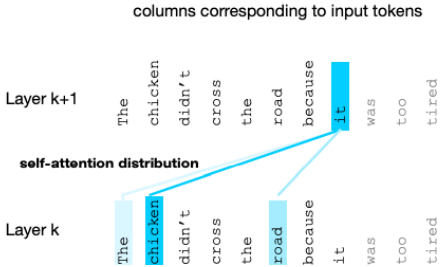


Figure 1: The self-attention weight distribution α that is part of the computation of the representation for the word **it** at layer $k+1$. In computing the representation for it, we attend differently to the various words at layer l, with darker shades indicating higher self-attention values. Note that the transformer is attending highly to the columns corresponding to the tokens **chicken** and **road**, a sensible result, since at the point where it occurs, it could plausibly corefer with **the chicken** or **the road**, and hence we'd like the representation for it to draw on the representation for these earlier words

Calculating Self-attention

- Attention computes a a vector representation for a token at a particular layer of a transformer, by selectively attending to and integrating information from prior tokens at the previous layer.
- Attention takes an input representation x_i corresponding to the input token at position i, and a context window of prior inputs $x_1 \dots x_{i-1}$, and produces an output a_i

Calculating Self-attention: A simplified view

- Before digging into the details, let's zoom out and look at a very simplified version of attention.
- At a basic level attention is really just a weighted sum of context vectors, with many clever knobs and levers added to determine how the weights are computed and what gets summed.
- Attention output a_i at token position i is simply the weighted sum of all the representations x_j, for all $j \leq i$. Note: For now we'll be assuming that the Attention calculations are only calculated for the preceding tokens — later we'll look at architectures that look at both preceding and future tokens
- We'll use α_{ij} to mean how much x_i should contribute to a_j

$$a_i = \sum_{j \leq i} \alpha_{ij} x_j$$

Calculating Self-attention: A simplified view

- The attention calculation simplified:

$$a_i = \sum_{j \leq i} \alpha_{ij} x_j$$

- Each α_{ij} is a scalar used for weighing the value of input x_j when summing up the inputs to compute a_i
- Each prior embedding is weighted proportionally to how similar it is to the current token i. So the output of attention is a sum of the embeddings of prior tokens weighted by their similarity with the current token embedding.
- We already have a metric for embedding similarity that we learned about last week — the dot product — and we use to calculate the similarity. The larger the score, the more similar the vectors that are being compared
- We then normalize these scores with a softmax to create the vector of weights

Calculating Self-attention: A simplified view

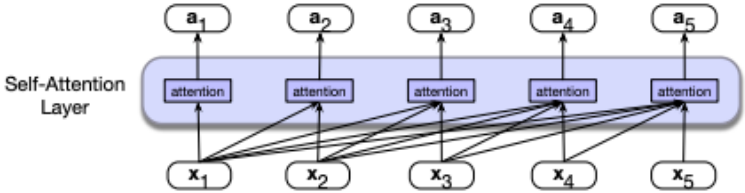


Figure 2: We compute a_3 by computing three scores: $x_3 \cdot x_1$, $x_3 \cdot x_2$, and $x_3 \cdot x_3$, normalizing them by a softmax, and using the resulting probabilities as weights indicating each of their proportional relevance to the current position i.

Calculating Self-attention: the Query, Key, and Value matrices

- Let's introduce the **Attention head**, the version of attention that's used in transformers. The attention head allows us to distinctly represent three different roles that each input embedding plays during the course of the attention process:
  - As the *current element* being compared to the preceding inputs. Well refer to this role as a **query**.
  - In its role as a *preceding input* that is being compared to the current element key to determine a similarity weight. Well refer to this role as a **key**.
  - And finally, as a **value** of a preceding element that gets weighted and summed up to compute the output for the current element.
- To capture these three different roles, transformers introduce weight matrices W^Q, W^K, and W^V

Calculating Self-attention

$$
\begin{aligned}
\mathbf{q}_i = \mathbf{x}_i\mathbf{W}^{\mathbf{Q}};\ \ \mathbf{k}_j &= \mathbf{x}_j\mathbf{W}^{\mathbf{K}};\ \ \mathbf{v}_j = \mathbf{x}_j\mathbf{W}^{\mathbf{V}} \\
\text{score}(\mathbf{x}_i,\mathbf{x}_j) &= \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}} \\
\alpha_{ij} &= \text{softmax}(\text{score}(\mathbf{x}_i,\mathbf{x}_j))\ \ \forall j \le i \\
\mathbf{head}_i &= \sum_{j \le i} \alpha_{ij}\mathbf{v}_j \\
\mathbf{a}_i &= \mathbf{head}_i\mathbf{W}^{\mathbf{O}}
\end{aligned}
$$

Equations for computing self-attention for a single self-attention output vector a_i from a single input vector x_i

- When we are computing the similarity of the current element x_i with some prior element x_j , well use the dot product between the current elements query vector q_i and the preceding elements key vector k_j .

Calculating Self-attention

$$
\begin{aligned}
\mathbf{q}_i = \mathbf{x}_i\mathbf{W}^{\mathbf{Q}};\ \ \mathbf{k}_j &= \mathbf{x}_j\mathbf{W}^{\mathbf{K}};\ \ \mathbf{v}_j = \mathbf{x}_j\mathbf{W}^{\mathbf{V}} \\
\text{score}(\mathbf{x}_i, \mathbf{x}_j) &= \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}} \\
\alpha_{ij} &= \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j))\ \ \forall j \leq i \\
\mathbf{head}_i &= \sum_{j \leq i} \alpha_{ij}\mathbf{v}_j \\
\mathbf{a}_i &= \mathbf{head}_i\mathbf{W}^{\mathbf{O}}
\end{aligned}
$$

Equations for computing self-attention for a single self-attention output vector a_i from a single input vector x_i

The result of a dot product can be an arbitrarily large and can lead to numerical issues and loss of gradients during training. To avoid this, we scale the dot product by a factor related to the size of the embeddings, via dividing by the square root of the dimensionality of the query and key vectors (d_k).

Calculating Self-attention

$$\begin{aligned}
\mathbf{q}_i = \mathbf{x}_i\mathbf{W}^{\mathbf{Q}};\ \ \mathbf{k}_j &= \mathbf{x}_j\mathbf{W}^{\mathbf{K}};\ \ \mathbf{v}_j = \mathbf{x}_j\mathbf{W}^{\mathbf{V}} \\
\text{score}(\mathbf{x}_i,\mathbf{x}_j) &= \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}} \\
\alpha_{ij} &= \text{softmax}(\text{score}(\mathbf{x}_i,\mathbf{x}_j))\ \ \forall j \le i \\
\mathbf{head}_i &= \sum_{j \le i} \alpha_{ij}\mathbf{v}_j \\
\mathbf{a}_i &= \mathbf{head}_i\mathbf{W}^{\mathbf{O}}
\end{aligned}$$

Equations for computing self-attention for a single self-attention output vector a_i from a single input vector x_i

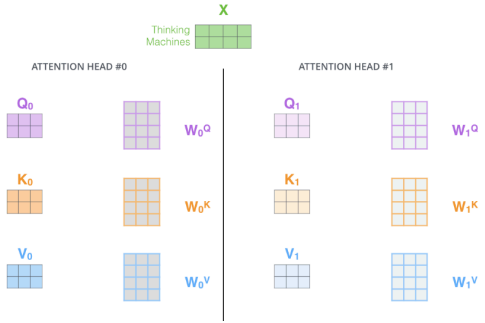
- We pass the scores through a softmax calculation resulting in an $_{i,j}$
- The output calculation for $head_i$ is now based on a weighted sum over the value vectors $\mathbf{v}$

Multi-head attention

- The preceding was a walkthrough of a single attention head. But transformers use multiple attention heads.
- The intuition is that each head might be attending to the context for different purposes: heads might be specialized to represent different linguistic relationships between context elements and the current token, or to look for particular kinds of patterns in the context.

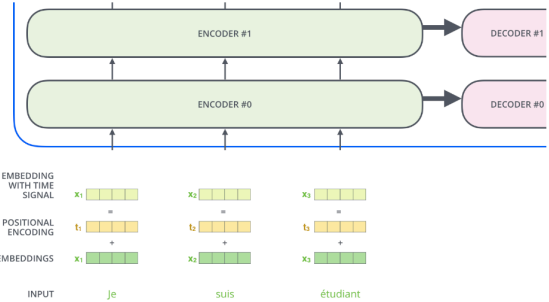
Multi-head attention

- In multi-head attention we have separate attention heads that reside in parallel layers, each with its own set of parameters.
- Each head models different aspects of the relationships among inputs. Thus each head i in a self-attention layer has its own set of key, query, and value matrices: W^{Ki}, W^{Qi}, and W^{Vi}.



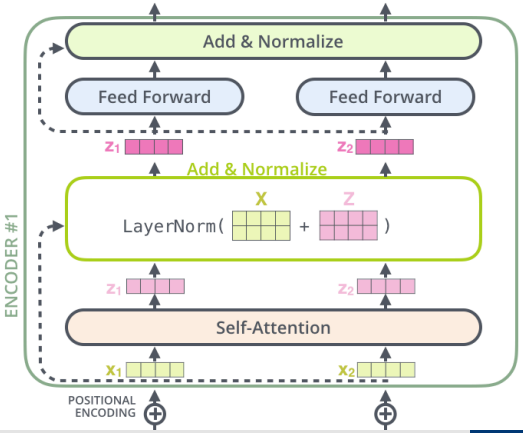
Positional encodings

- How do we account for the order of the words in the input sequence?
- We add a positional encoding that represents the sequential position of the token in the context. This is added to the input embedding for each word.



Layer normalization

- Each sub-layer (self-attention, FFN) in each encoder has a residual connection around it, and is followed by a layer-normalization step — the **LayerNorm** operation.



Layer normalization

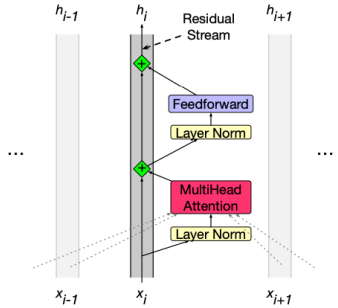
- The term layer normalization is a bit confusing; layer normalization is not applied to an entire transformer layer, but just to the embedding vector of a single token.
- Remember the point of normalization in NNs: to keep the values in a range that facilitates gradient-based training.
- Layer norm is a variation of the z-score from statistics, applied to a single vector in a hidden layer: the resulting vector has a zero mean and a standard deviation of one.

Residual connections

- A **residual connection** (also known as a **residual stream**, **skip connection** or **shortcut connection**) is a mechanism used to address the vanishing gradient problem in deep neural networks such as transformers.
- The idea behind a residual connection is to add the original input (or a modified version of it) to the output of a deeper layer. This helps mitigate the degradation of gradient information as it flows backward through multiple layers during training. Residual connections enable the network to learn incremental changes rather than trying to learn the entire transformation from scratch.

Residual connections

- The initial vector is passed through a layer norm and attention layer, and the result is added back into the stream, in this case to the original input vector x_i.
- This summed vector is again passed through another layer norm and a feedforward layer, and the output of those is added back into the residual– h_i is resulting output token i.



The importance of sampling strategies

- The most common method for decoding in large language models is **sampling**
- Sampling from a model's distribution over words means to choose random words according to their probability assigned by the model
- A good sampling strategy will find the optimal tradeoff between **quality** and **diversity**
- Methods that emphasize the most probable words tend to produce generations that are rated by people as more accurate, more coherent, and more factual, but also more boring and more repetitive
- Methods that give a bit more weight to the middle-probability words tend to be more creative and more diverse, but less factual and more likely to be incoherent or otherwise low-quality

Greedy decoding

- **Greedy decoding**: Generate the most likely word given the context
- Make a choice that is locally optimal, whether or not it will turn out to have been the best choice with hindsight
- At each time step in generation, the output y_t is chosen by computing the probability for each possible output (every word in the vocabulary) and then choosing the highest probability word, (orthe *argmax*):

$$\hat{w}_t = argmax_w \in VP(w|\mathbf{w}_{<t})$$

- The main problem with greedy decoding is that because the words it chooses are (by definition) extremely predictable, the resulting text is generic and often quite repetitive

Greedy decoding

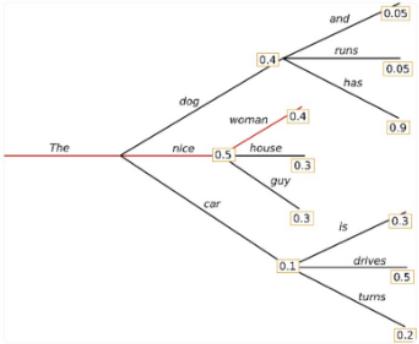


Figure 3: Generating the sentence *The nice woman* using greedy decoding

Top-k sampling

- Basically a generalization of greedy decoding to k tokens (top-k sampling with $k = 1$ is just greedy decoding)
- The top-k sampling algorithm
  1. For each word in the vocabulary V, use the language model to compute the likelihood of this word given the context
  2. Sort the tokens by their likelihood; keep top k
  3. Renormalize the scores of the k words to be a legitimate probability distribution
  4. Pick the highest-ranked token

Top-p or nucleus sampling

- In **top-p sampling** or **nucleus sampling** , we keep the top p percent of the probability mass, rather than the top k words
- Sample tokens with the highest probability scores until a specified threshold p is reached
- Tends to generate text that is more varied and creative then greedy or top-k sampling

Temperature sampling

- Divide the logits by a temperature parameter τ before we normalize it by passing it through the softmax
- Higher values of τ lead to greater variability, more creative text
- Lower values of τ lead to boring, generic text
- Why this works:
  - Softmax pushes high values toward 1 and low values toward 0
  - Te lower τ is, the larger the scores being passed to the softmax since dividing by a smaller fraction ≤ 1 results in making each score larger
  - When larger numbers are passed to a softmax the result is a distribution with increased probabilities of the most high-probability words and decreased probabilities of the low probability words

Temperature sampling

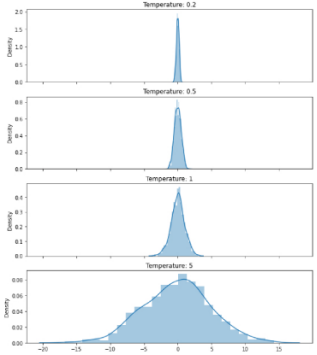


Figure 4: Temperature sampling: As the temperature nears 0, the probability of the most likely word approaches 1.

Next class: Feb 24

Assignment 2 Due

Next: Variant Transformer architectures and an introduction to finetuning

- Jurafsky & Martin Chapter 10: Masked Language Models
- Jurafsky & Martin Chapter 7: Large Language Models
- BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding
- Language Models are Unsupervised Multitask Learners
- Language Models are Few-Shot Learners