Natural Language Processing

Class 5: The Transformer

Adam Faulkner

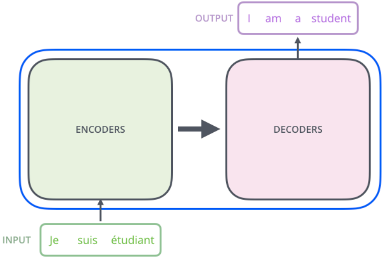
September 30, 2025

The Transformer: The neural architecture that made contemporary AI possible

- The standard architecture for building LLMs
- Today, we'll be discussing the original Transformer architecture introduced in Attention is all you Need (2017), specifically the **Encoder-Decoder** — also called **sequence-to-sequence** — Transformer architecture originally proposed for Machine Translation
- In class 6 we'll cover **Encoder-only**, or **BERT**-style, Transformers and **Decoder-only**, or **GPT**-style, Transformers

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

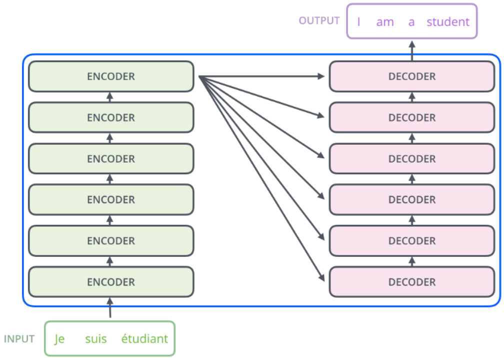
$$\mathbf{h} = encoder(x)$$

and, word by word, generates the output y:

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

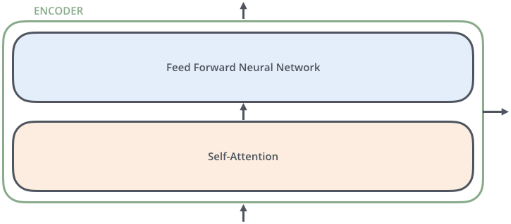
The Encoder

- The encoding component is a stack of encoders (the *Attention is all you Need* paper stacks six of them on top of each other but there many possible arrangements).
- The decoding component is a stack of decoders of the same number.



The Encoder

- The encoder's inputs first flow through a **self-attention** layer—a layer that helps the encoder look at other words in the input sentence as it encodes a specific word.



- Let's take a look at the attention mechanism in more detail.

Why we need contextual representations of words

- Recall from last week 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

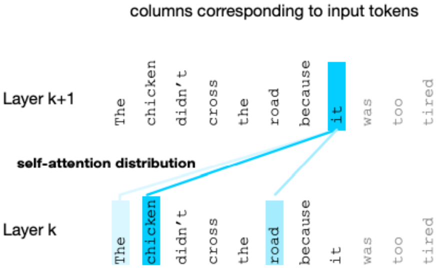


Figure 1: The self-attention weight distribution α that is part of the computation of the representation for the word **it** at layer $k+1$. In computing the representation for it, we attend differently to the various words at layer l, with darker shades indicating higher self-attention values. Note 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 $x_1 \ldots x_{i-1}$, 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

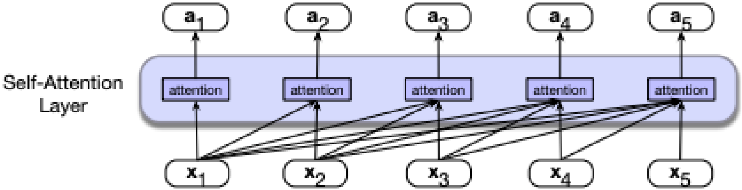


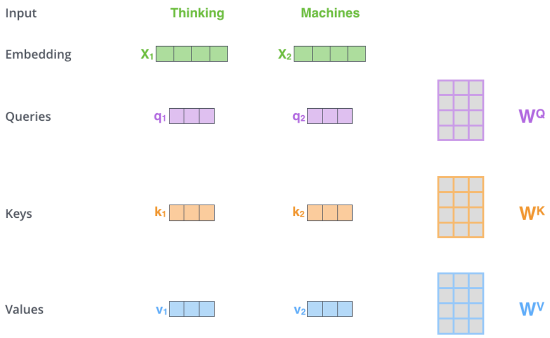
Figure 2: We compute a_3 by computing three scores: $x_3 \cdot x_1, x_3 \cdot x_2$, and $x_3 \cdot x_3$, normalizing them by a softmax, and using the resulting probabilities as weights indicating each of their proportional relevance to the current position i.

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

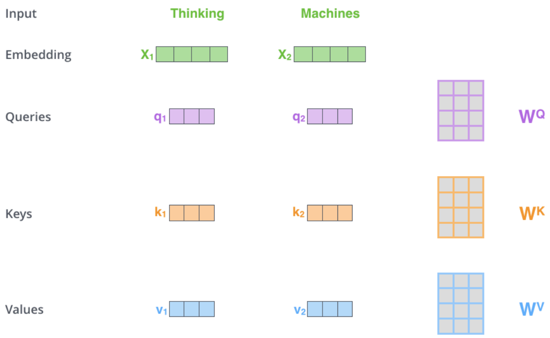
Step 1: Create Query, Key, and Value vectors from each input embedding

- For each word, we create Key, Query, and a Value vector. These vectors are created by multiplying the embedding by three matrices that we trained during the training process.



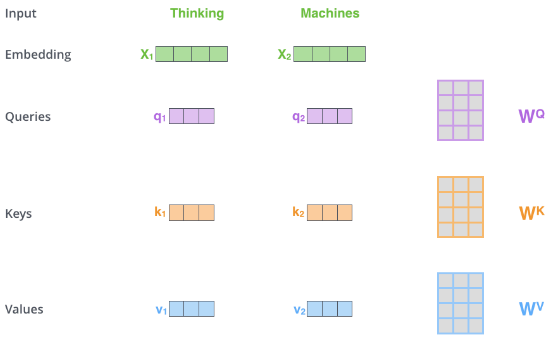
Step 1: Create Query, Key, and Value vectors from each input embedding

- Notice that these new vectors are smaller in dimension than the embedding vector. Their dimensionality is 64, while the embedding and encoder input/output vectors have dimensionality of 512.



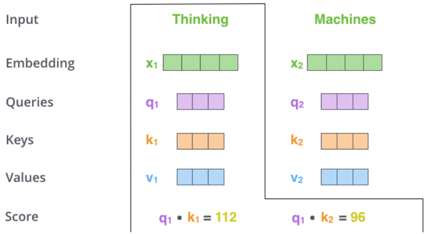
Step 1: Create Query, Key, and Value vectors from each input embedding

- Multiplying x_1 by the W^Q weight matrix produces q_1, the Query vector associated with that word. We create a Query, Key, and Value projection of each word in the input sentence.



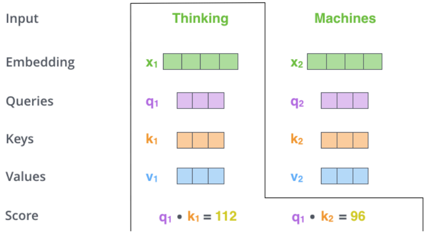
Step 2: Calculate a score by taking the dot product of the query vector with the key vector of the word we're scoring

- The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.



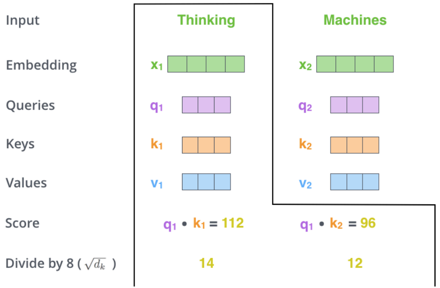
Step 2: Calculate a score by taking the dot product of the query vector with the key vector of the word we're scoring

- For example, if we're processing the self-attention for the word in position 1, the first score would be the dot product of q_1 and k_1. The second score would be the dot product of q_1 and k_2.



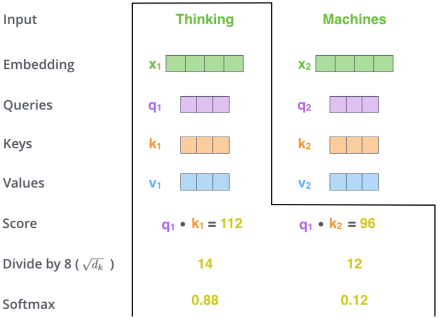
Step 3: Take square root of the dimension of the key vectors

- To ensure stable gradients, we next take the square root of the dimension of the key vectors, i.e., divide by 8 which is the square root of the 64-dimension vectors.



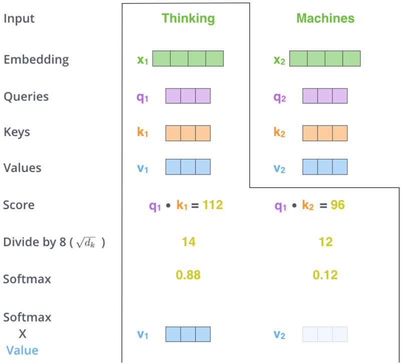
Step 4: Take the softmax to determine how much each word will be expressed at this position

- As we learned in a past class, softmax results in a probability distribution.



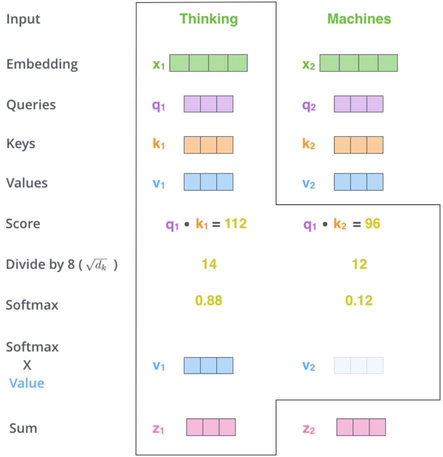
Step 5: Multiply each value vector by the softmax score

- The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example).

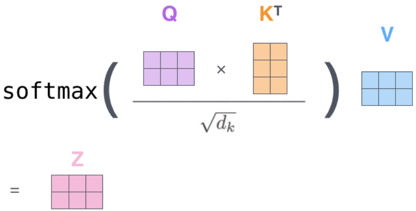


Step 6: Sum up the weighted value vectors.

- This produces the output of the self-attention layer at this position



The entire self-attention calculation in matrix form

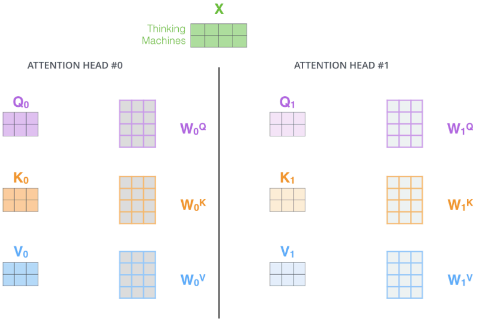


Multi-head attention

- 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.

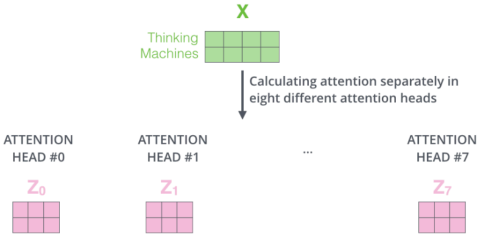
Multi-head attention

- 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}.



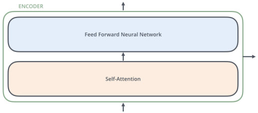
Multi-head attention

- We perform the same self-attention calculation for each of the eight different different weight matrices.

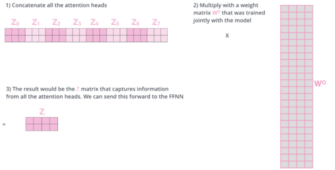


Multi-head attention

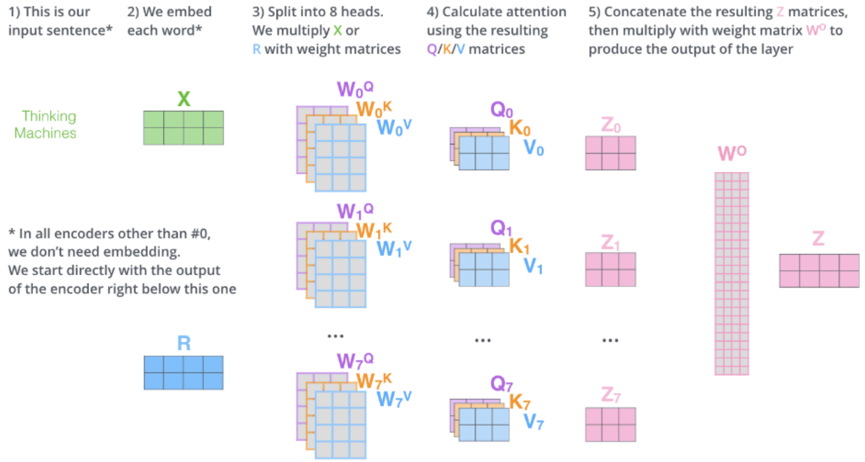
- Recall that we need to pass the result of self-attention to a FFN:



- The FFN is not expecting eight matrices — it's expecting a single matrix (a vector for each word).
- So condense these eight down into a single matrix by concatenating the matrices, and then multiplying them by an additional weights matrix W^O.

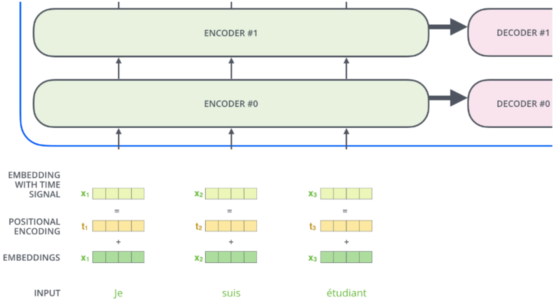


Multi-head attention



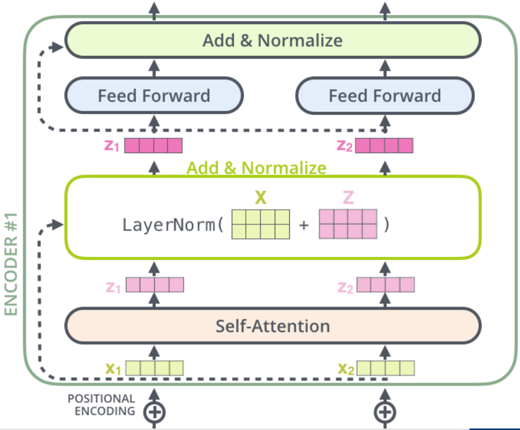
Positional encodings

- 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

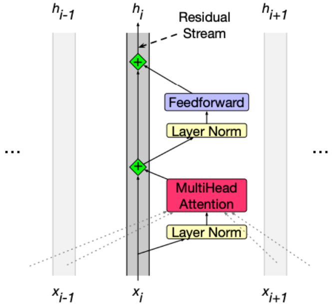
- The term layer normalization is a bit confusing; layer normalization is not applied to an entire transformer layer, but just to the embedding vector of a single token.
- Remember the point of normalization in NNs: to keep the values in a range that facilitates gradient-based training.
- Layer norm 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.

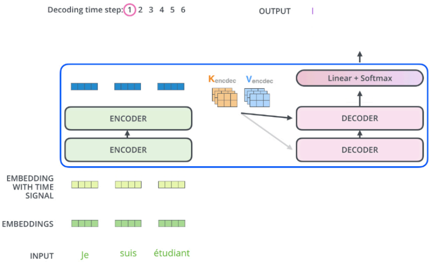
Residual connections

- 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.



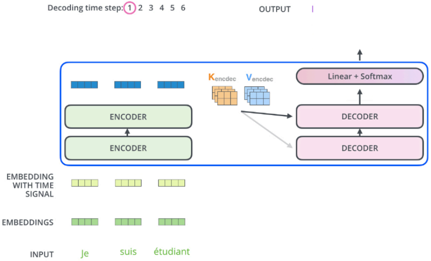
Cross attention

- The decoder transformer block includes an extra layer with a special kind of attention, **cross-attention** (also sometimes called **encoder-decoder attention**).
- Cross-attention has the same form as the multi-head attention in a normal transformer block, except that while the queries as usual come from the previous layer of the decoder, the keys and values come from the output of the encoder.



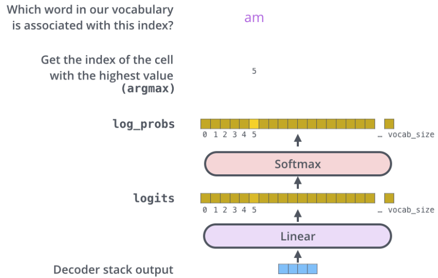
Cross-attention

- Cross-attention thus allows the decoder to attend to each of the source language words as projected into the entire encoder's final output representations.



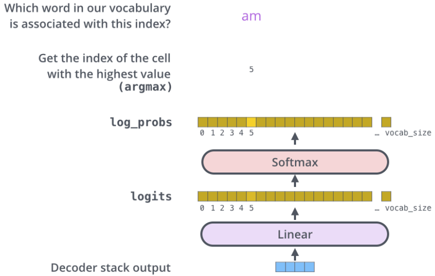
The Decoder: Final layer (linear layer + softmax)

- We're almost ready to generate our second token, *am*, the translation of *suis*.
- The decoder stack outputs a vector of floats. How do we turn that into a word? That's the job of the final Linear layer which is followed by a Softmax Layer.



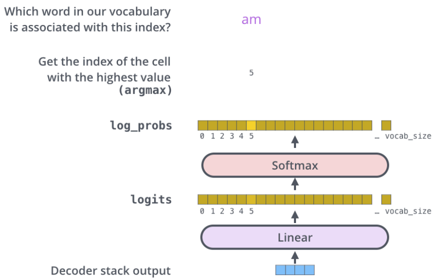
The Decoder: Final layer (linear layer + softmax)

- The Linear layer is a simple FFN that projects the vector produced by the stack of decoders, into a much, much larger vector called a **logits** vector.



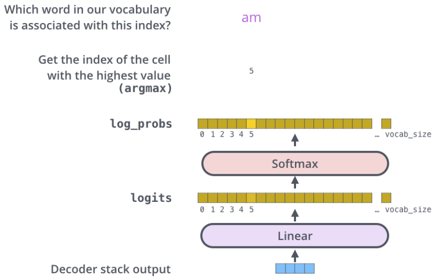
The Decoder: Final layer (linear layer + softmax)

- Let's assume that our model knows 10,000 unique English words (our model's output vocabulary) that it's learned from its training dataset. This would make the logits vector 10,000 cells wide — each cell corresponding to the score of a unique word. That is how we interpret the output of the model followed by the Linear layer



The Decoder: Final layer (linear layer + softmax)

- The softmax layer then turns those scores into probabilities (all positive, all add up to 1). The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step: *am*



Next class: Oct 21

Oct 7: No Class; Assignment 2 Due

Oct 14: No Class

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