Mining Unstructured Data
12. RNN & LLM and LLM-based Assistants

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

FIB

Outline

Outline

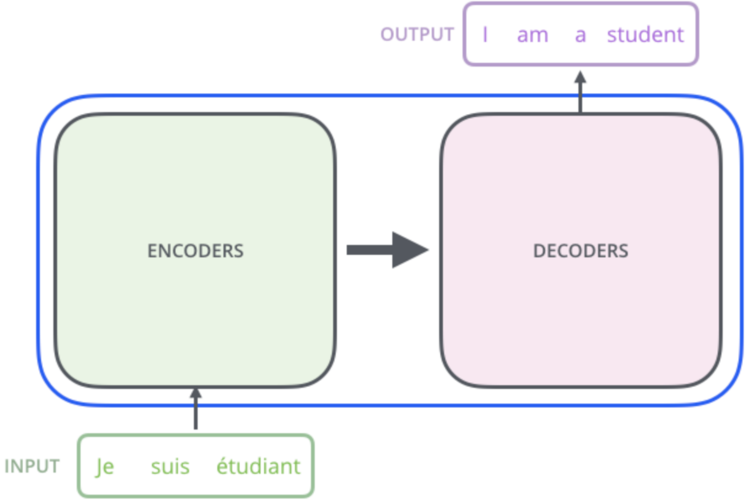
Sequence-to-Sequence Tasks

Many NLP tasks can be phrased as sequence-to-sequence:

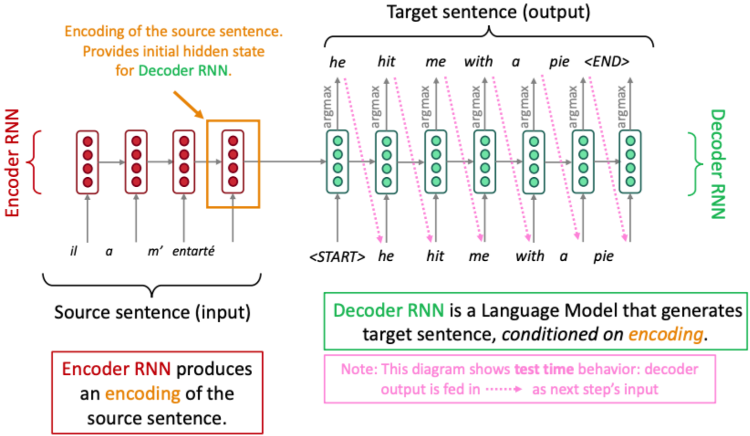
- Summarization (long text → short text)
- Dialogue (previous utterances → next utterance)
- Parsing (input text → output parse as sequence)
- Code generation (Natural Language → Python Code)
- Translation (source sentence → translation)

se la summarizaton e' solo tieni o non tieni questa frase potrebbbe non essere necessario usare seq-to-seq model

Sequence-to-Sequence Model



Sequence-to-Sequence Model



Outline

Neural Machine Translation

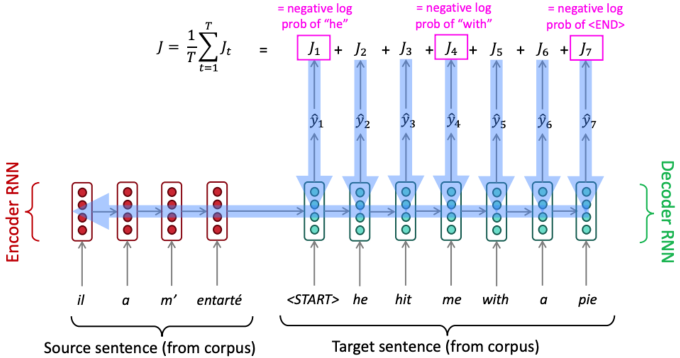
- The sequence-to-sequence model is an example of a Conditional Language Model.
- Language Model because the decoder is predicting the next word of the target sentence y.
- Conditional because its predictions are also conditioned on the source sentence x.

$$p(y|x) = \prod_{t=1}^{T_y} p(y_t|y_{<t}, x)$$

- NMT computes the conditional probability distribution $p(y|x)$.
- We train these models with a parallel corpus.

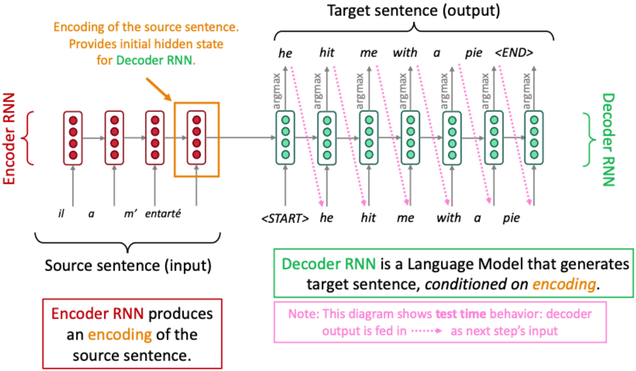
Teacher Forcing

- During training, we use a technique called teacher forcing:
  - We feed the network the correct target sequence as input for each time step, rather than the predicted output from the previous time step
  - This helps to stabilize the training process and improve the quality of the final translation



Greedy Decoding

- During inference, we use a technique called greedy decoding:
  - At each time step, we choose the word with the highest probability as the next output word
  - This can lead to suboptimal translations, as the network may get stuck in local optima



Exhaustive Search Decoding

- To find the optimal translation, we could track all possible sequences
- This means that on each step t, we track possible partial translations, where V is the vocabulary size
- However, this complexity is too expensive

$$\arg\max_y P(y|x) = \arg\max_y \prod_{t=1}^{|y|} P(y_t|y_{<t}, x)$$

Beam Search Decoding

- Beam search is a more efficient decoding algorithm than exhaustive search
- On each step of the decoder, we keep track of the k most probable partial translations (which we call hypotheses)
- Each hypothesis has a score, its log probability
- We search for high-scoring hypotheses, tracking top k on each step
- Beam search doesn't guarantee an optimal solution, but is more efficient than exhaustive search

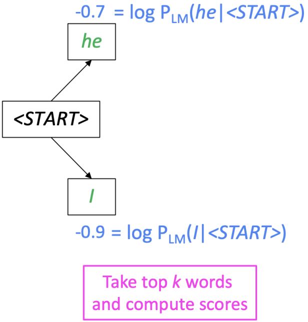
$$\text{score}(y_{1:t}) = \sum_{i=1}^{t} \log P(y_i | y_{<i}, x)$$

Beam Search Decoding (k=2)

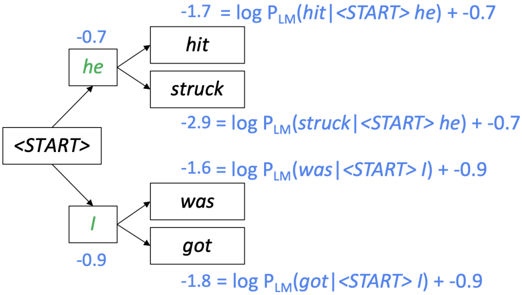
<START>

Calculate prob dist of next word

Beam Search Decoding (k=2)

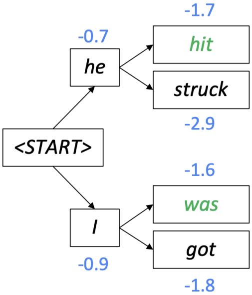


Beam Search Decoding (k=2)



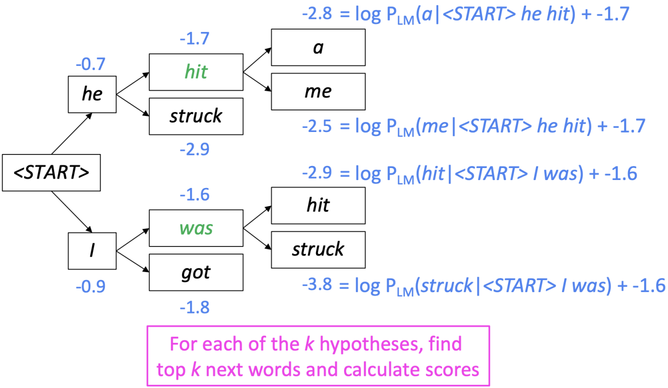
For each of the *k* hypotheses, find top *k* next words and calculate scores

Beam Search Decoding (k=2)

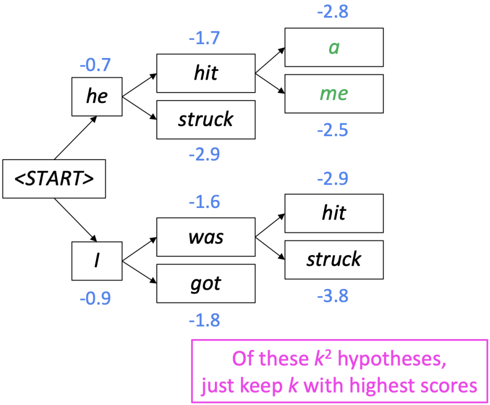


Of these k^2 hypotheses,
just keep k with highest scores

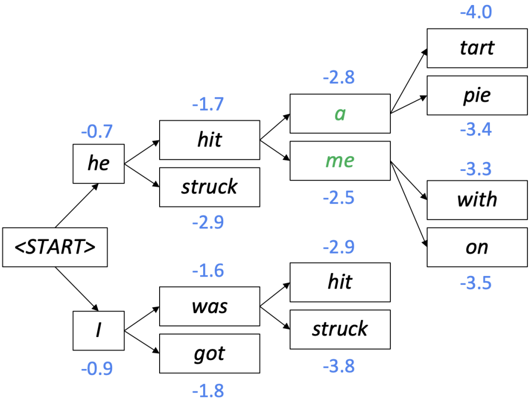
Beam Search Decoding (k=2)



Beam Search Decoding (k=2)

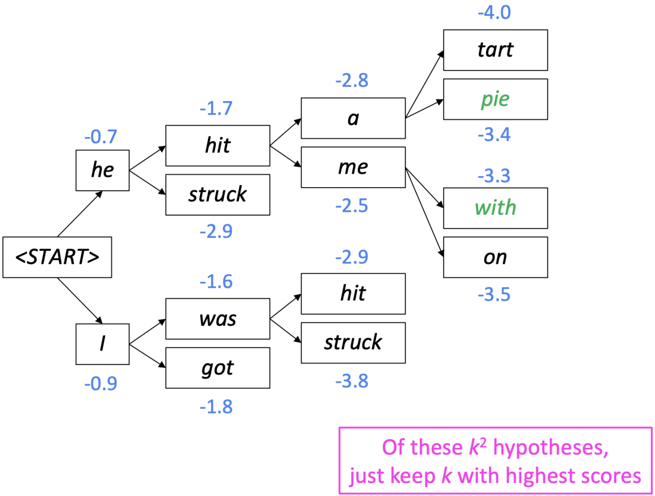


Beam Search Decoding (k=2)

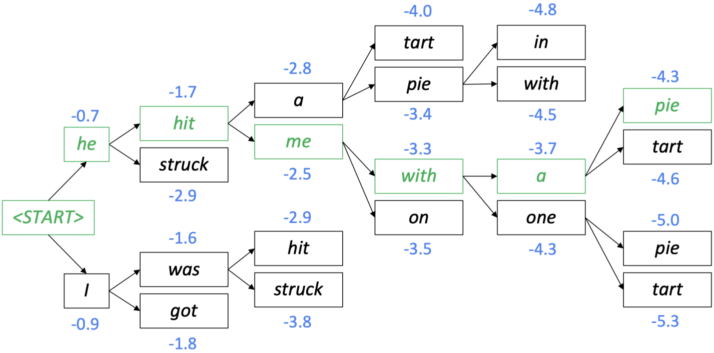


For each of the *k* hypotheses, find top *k* next words and calculate scores

Beam Search Decoding (k=2)



Beam Search Decoding (k=2)



Backtrack to obtain the full hypothesis

Beam Search Decoding - Stopping Criterion

- In greedy decoding, usually we decode until the model produces an END token:
  - START he hit me with a pie END
- In beam search decoding, different hypotheses may produce tokens on different timesteps.
- When a hypothesis produces END, that hypothesis is complete.
- Place it aside and continue exploring other hypotheses via beam search.
- Usually we continue beam search until:
  - We reach timestep T (where T is some pre-defined cutoff), or
  - We have at least n completed hypotheses (where n is pre-defined cutoff).

Beam Search Decoding - Selecting the Best Hypothesis

- Each hypothesis in our list of hypotheses has a score.
- Problem with this: longer hypotheses have lower scores.
- Fix: Normalize by length. Use this to select top one instead:

$$\hat{y} = \arg\max_{y \in \mathcal{Y}} \frac{1}{|y|^{\alpha}} \log p(y|x) \tag{1}$$

- More negative terms are added for longer hypotheses.

Outline

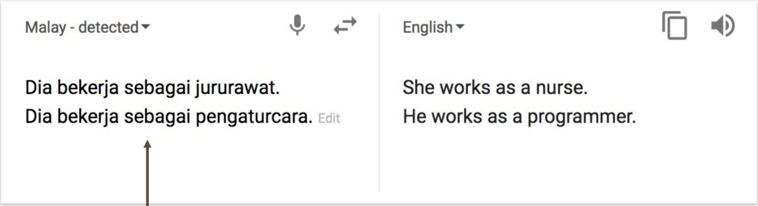
About the Success of NMT

Neural Machine Translation went from a fringe research attempt in 2014 to the leading standard method in 2016.

- 2014: First seq2seq paper published.
- 2016: Google Translate switches from SMT to NMT
- 2018+: Transformers have become the dominant architecture for NMT. Examples include:
  - BART (Facebook AI Research) (2019)
  - T5 (Google AI) (2020)
  - UNILM (Microsoft Research Asia) (2020)

NMT is far from solved

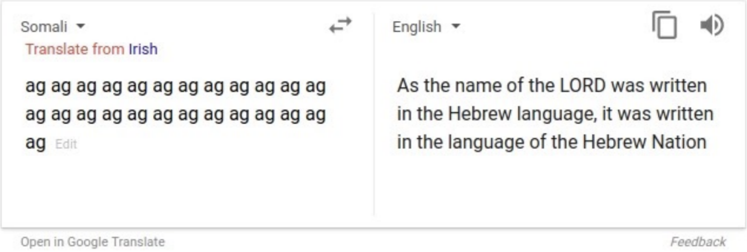
NMT picks up biases in training data



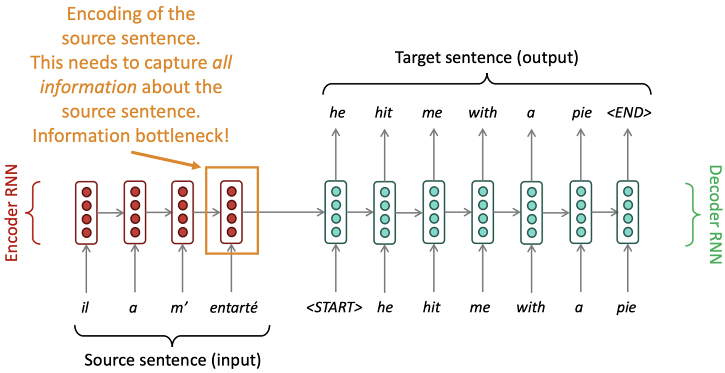
Didn't specify gender

NMT is far from solved (II)

Hard to interpret systems do strange things



Sequence-to-sequence: the bottleneck problem



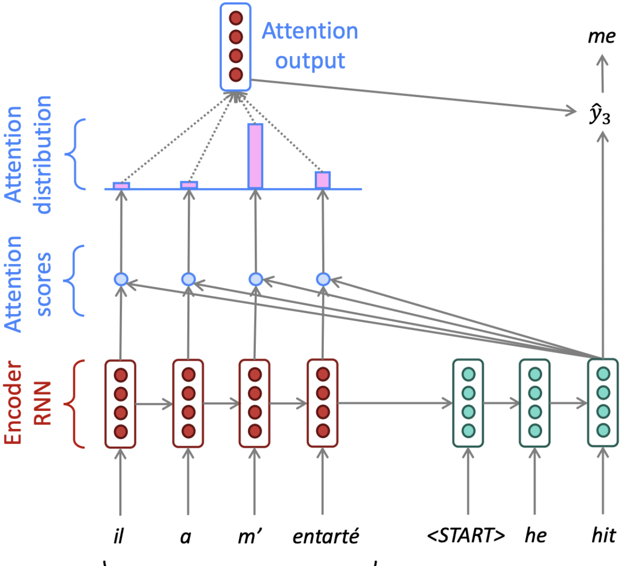
Outline

Outline

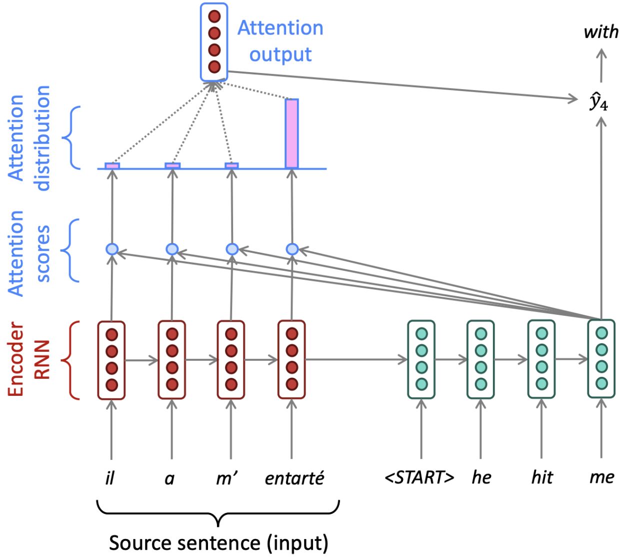
Attention

- Attention provides a solution to the bottleneck problem.
- **Core idea:** on each step of the decoder, use direct connection to the encoder to focus on a particular part of the source sequence

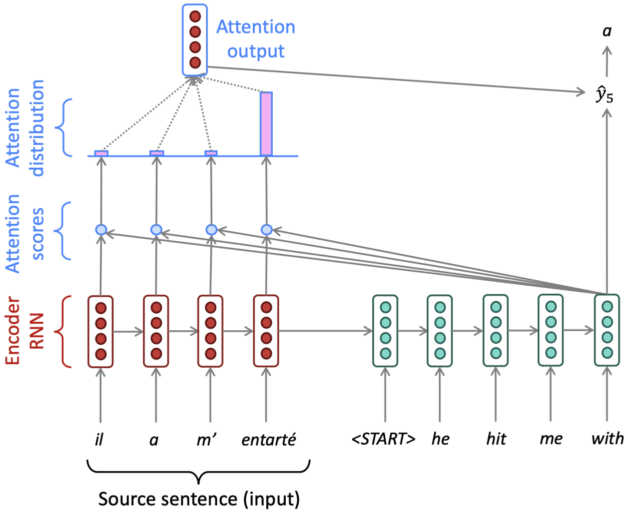
Attention (II)



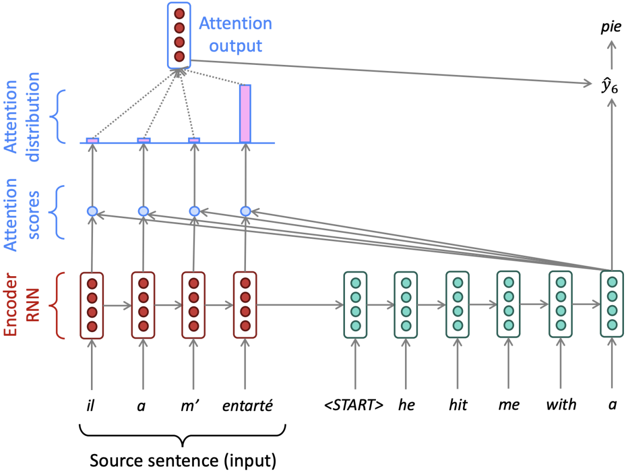
Attention (III)



Attention (IV)



Attention (V)



Attention output
pie
$\hat{y}_6$
Attention distribution
Attention scores
Encoder RNN
il
a
m'
entarté
<START>
he
hit
me
with
a
Source sentence (input)

Outline

Basic Attention (with no parameters)

Suppose we have a sequence of n items, represented as $x_1, x_2, \ldots, x_n$, and we want to compute the attention weights for each item based on a query q.

$$s_i = x_i^T q$$

$$w_i = \frac{\exp(s_i)}{\sum_{j=1}^{n} \exp(s_j)}$$

$$\text{Attention}(q, x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} w_i x_i$$

This gives us a weighted representation of the sequence based on the query. Note that the attention weights are computed based solely on the query and the representations of the items, with **no learned parameters**.

Generalization of Attention

- We can use attention in many architectures (not just seq2seq) and many tasks (not just MT)
- General definition of attention:
  - Given a set of vector **values** and **keys**, and a vector **query**, attention is a technique to compute a weighted sum of the values, dependent on the query and keys.
- Intuition:
  - The weighted sum is a selective summary of the information contained in the values, where the query and keys determine which values to focus on.
  - Attention is a way to obtain a **fixed-size** representation of an arbitrary set of representations (the values), dependent on some other representation (the query).

Generalised Dot-Product Attention Formulas

In the dot-product attention mechanism, we compute the attention weights as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V,$$

where Q, K, and V are the query, key, and value matrices, respectively, and d_k is the dimensionality of the key vectors.

- Q is of size (batch size $\times\, n_q \times d_k$)
- K is of size (batch size $\times\, n_k \times d_k$)
- V is of size (batch size $\times\, n_k \times d_v$)

Here, n_q and n_k denote the number of queries and keys, respectively, and d_v is the dimensionality of the value vectors.

Generalised Dot-Product Attention Formulas (II)

We can also write the attention function as a weighted sum of the value vectors, where the attention weights are given by the dot product of the query and key vectors:

$$\text{Attention}(Q, K, V) = \sum_{i=1}^{n_k} \alpha_i v_i,$$

where $\alpha_i = \frac{\exp(qk_i/\sqrt{d_k})}{\sum_{j=1}^{n_k} \exp(qk_j/\sqrt{d_k})}$ is the attention weight for the i-th key, and v_i is the corresponding value vector.

Generalised Attention Formulas

- We can generalise to other attention models:

$$\alpha_i = \mathrm{softmax}(\mathrm{score}(q, h_i))$$

- score is a function that computes the similarity between the query vector and each input vector. Common choices for the score are:
  - Dot product: $\mathrm{score}(q, h_i) = q^T h_i$
  - Scaled dot product: $\mathrm{score}(q, h_i) = \frac{q^T h_i}{\sqrt{d}}$
  - General: $\mathrm{score}(q, h_i) = q^T W h_i$
  - Concat: $\mathrm{score}(q, h_i) = v^T \tanh(W[q; h_i])$
  - Additive: $\mathrm{score}(q, h_i) = v^T \tanh(W_1 q + W_2 h_i)$
- Where d is the dimensionality of the query and input vectors, W and v are learned parameter matrices, and $[q; h_i]$ denotes the concatenation of the query and i-th input vector.

Outline

Advantages of Attention

- Attention significantly improves NMT performance
- Attention provides more “human-like” model of the MT process
    - We look back at the source sentence while translating, rather than remembering it all
- Attention solves the bottleneck problem
    - Attention allows decoder to look directly at source; bypass bottleneck
- Attention helps with the vanishing gradient problem
    - Provides shortcut to faraway states
- Attention provides some interpretability
    - By inspecting attention distribution, we see what the decoder was focusing on
    - The network learns alignment by itself

Attention Solves the Bottleneck Problem

In a traditional seq2seq model, the decoder receives a **fixed-length vector** (the context vector) that summarizes the entire source sequence. This creates a bottleneck, as the decoder must use this one vector to generate the entire target sequence.

$$e_{i,j} = v_a^T \tanh(W_a s_{i-1} + U_a h_j)$$

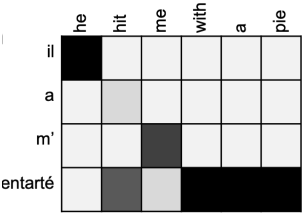
$$\alpha_{i,j} = \frac{\exp(e_{i,j})}{\sum_{k=1}^{n} \exp(e_{i,k})}$$

$$c_i = \sum_{j=1}^{n} \alpha_{i,j} h_j$$

where s_{i-1} is the decoder hidden state at the previous time step, h_j is the j-th encoder hidden state, v_a, W_a, and U_a are learned parameters, and c_i is the context vector at time step i. The attention weights $\alpha_{i,j}$ determine which parts of the source sequence to focus on at each step.

Attention Solves the Bottleneck Problem (II)

This allows the decoder to attend to different parts of the source sequence at each step, and thus avoid the bottleneck problem.



Outline

Outline

Issues with RNNs: Linear interaction distance

Linear locality

RNNs encode linear locality, which is useful since nearby words often affect each other's meaning.

Problem

However, RNNs take $O(\text{sequence length})$ steps for distant word pairs to interact, which makes it hard to learn long-distance dependencies due to vanishing gradients. Additionally, the meaning in sentences doesn't necessarily follow a *linear order*.

- This linear interaction distance limitation can be problematic for some NLP tasks, such as machine translation or sentiment analysis.
- This motivates the use of alternative models that can capture non-linear dependencies more effectively, such as transformers.

Issues with RNNs: Lack of parallelizability

Parallelizability

Forward and backward passes in RNNs have $O(\text{sequence length})$ unparallelizable operations.

- GPUs can perform a bunch of independent computations at once, which is great for speeding up training.
- However, future RNN hidden states can't be computed in full before past RNN hidden states have been computed.
- This lack of parallelizability inhibits training on very large datasets, which can be a major issue in modern NLP applications.

Solution

Transformers are highly parallelizable, allowing for much faster training on larger datasets.

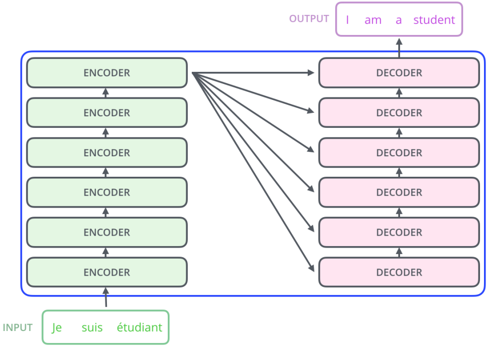
Outline

The Transformer Model

Source: https://jalammar.github.io/illustrated-transformer/

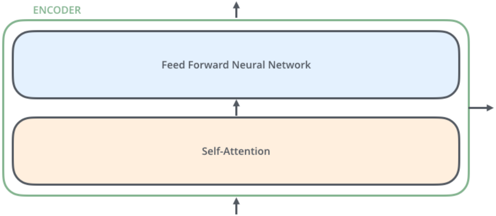
The Encoding and Decoding Components

- The Transformer consists of an encoding component, a decoding component, and connections between them.
- The encoding component is a stack of encoders, and the decoding component is a stack of decoders.
- The encoders and decoders are connected by attention layers.



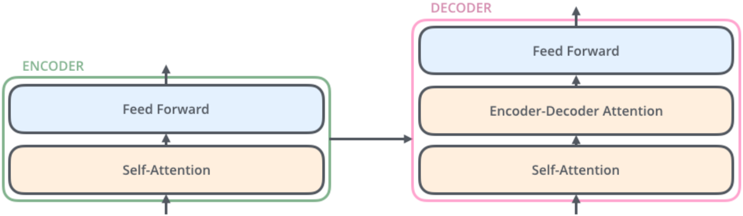
The Encoder Sub-Layers

- Each encoder has two sub-layers: a self-attention layer and a feed-forward neural network.
- The self-attention layer helps the encoder look at other words in the input sentence as it encodes a specific word.
- The feed-forward network is applied independently to each position.
- The exact same structure is used for all encoders, but they do not share weights.



The Decoder Sub-Layers

- The decoder has both the self-attention and feed-forward layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence.
- The attention layer helps the decoder generate the output sequence.



The Input Embedding

- Each input word is turned into a vector using an embedding algorithm.
- The embedding size is typically 512.
- The embedding only happens in the bottom-most encoder.
- Each encoder receives a list of vectors, the size of which is a hyperparameter we can set.

Figure: Word embeddings for input sequence.

What is Byte Pair Encoding?

- A simple and efficient compression algorithm that reduces the size of text data by replacing frequent pairs of bytes with a single byte.
- Originally proposed by Philip Gage in 1994 for compressing English text files.
- Later adapted for natural language processing tasks such as subword tokenization and neural machine translation.

How does Byte Pair Encoding work?

1. Initialize a vocabulary with all the unique bytes in the text data and a special end-of-word symbol (e.g. $\langle/\mathrm{w}\rangle$).
2. Count the frequency of all the byte pairs in the text data.
3. Merge the most frequent byte pair into a new byte and add it to the vocabulary.
4. Repeat steps 2 and 3 until a desired vocabulary size or compression ratio is reached.
5. Encode the text data by replacing each byte pair with its corresponding merged byte.

Example of Byte Pair Encoding

- Suppose we have a text data: “low lower newest wides”
- The initial vocabulary is: {"l", "o", "w", " ", "e", "r", "n", "s", "t", "i", "d", $\langle/\text{w}\rangle$}
- The most frequent byte pair is: ("e", "s")
- We merge ("e", "s") into a new byte: $\langle\text{es}\rangle$ and add it to the vocabulary.
- The new vocabulary is: {"l", "o", "w", " ", "e", "r", "n", "s", "t", "i", "d", $\langle/\text{w}\rangle$, $\langle\text{es}\rangle$}
- The new text data is: “low lower new$\langle\text{es}\rangle$t wid$\langle\text{es}\rangle$t”
- We repeat this process until we get the final vocabulary and encoded text data.

The Encoding Process

- Each word in the input sequence flows through its own path in the encoder.
- Dependencies between paths are captured in the self-attention layer.
- The same feed-forward layer weights are applied for all paths.

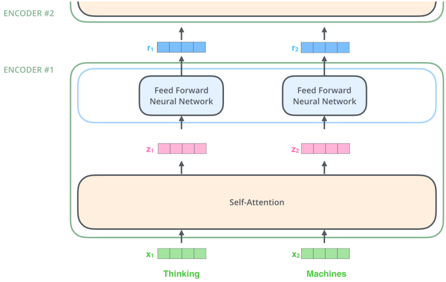


Figure: Encoding process in the Transformer.

Outline

Self-Attention

- Self-attention is a key component of the Transformer model that allows it to understand the context of a word in a sentence.
- It allows the model to associate a word with other relevant words in the sentence.
- For example, when processing the sentence "The animal didn't cross the street because it was too tired", self-attention helps the model understand that "it" refers to "animal".
- Self-attention works by looking at other positions in the input sequence for clues that can help lead to a better encoding for the current word.
- The Transformer uses self-attention to incorporate the representation of other relevant words into the one it is currently processing.

Self-Attention

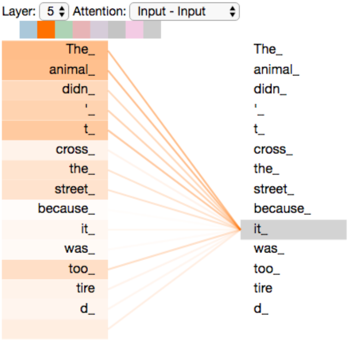
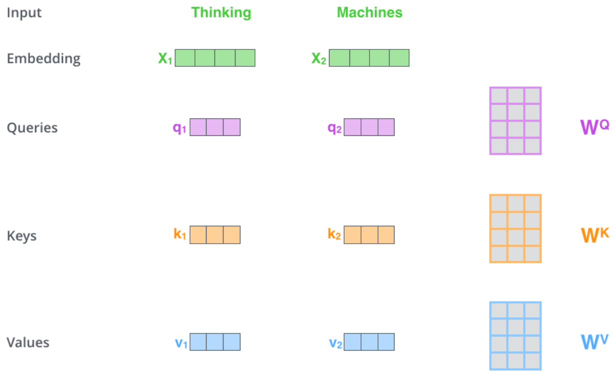


Figure: Example of self-attention in encoder #5 of the Transformer model. Part of the attention mechanism is focusing on "The Animal", and this information is incorporated into the encoding of "it".

Self-Attention in Detail

- First step: create three vectors from each encoder's input vectors
  - Create Query, Key, and Value vectors by multiplying embedding by three matrices
  - Vectors are smaller in dimension than embedding vector (64 vs 512)

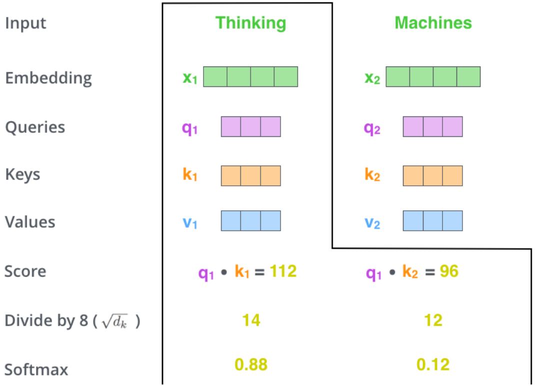
Self-Attention in Detail (I)



Self-Attention in Detail (II)

- Second step: calculate a score for each word in the input sentence
  - Score is calculated by taking dot product of query vector with key vector of respective word
- Third step: divide scores by $\sqrt{d}$ (scaled dot-product attention)
- Fourth step: pass through softmax operation
  - Softmax score determines how much each word will be expressed at this position

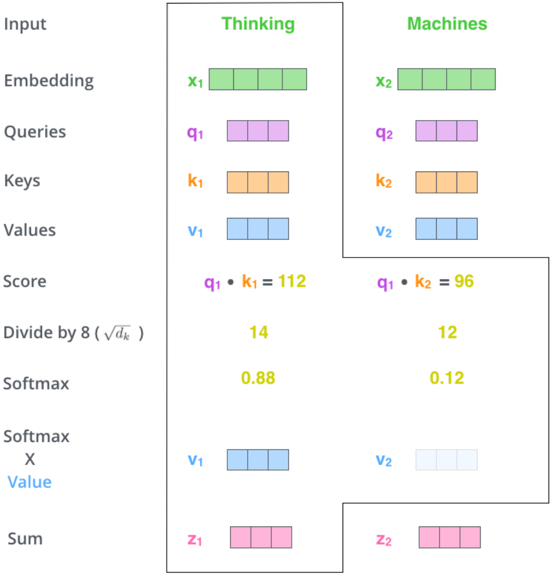
Self-Attention in Detail (II)



Self-Attention in Detail (III)

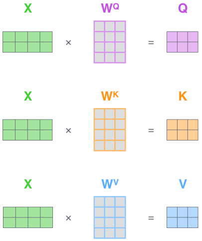
- Fifth step: multiply each value vector by the softmax score
- Sixth step: sum up the weighted value vectors to produce output of self-attention layer
- Calculation is done in matrix form for faster processing in actual implementation

Self-Attention in Detail (III)



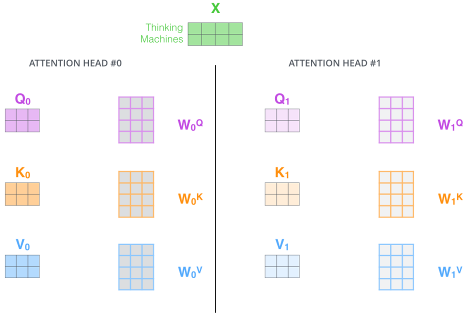
Matrix Calculation of Self-Attention

1. Calculate Query, Key, and Value matrices by multiplying embedding matrix X with weight matrices WQ, WK, and WV respectively.
2. X matrix represents input sentence with each row representing a word and q/k/v vectors having a different size than embedding vector.
3. Outputs of the self-attention layer can be calculated using a single formula.

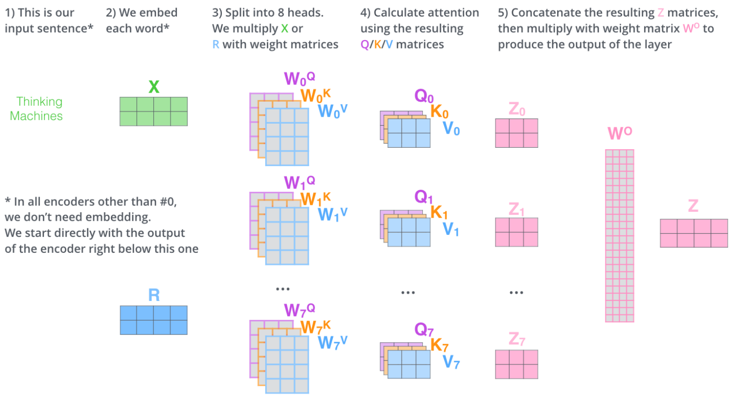


Multi-Head Self-Attention

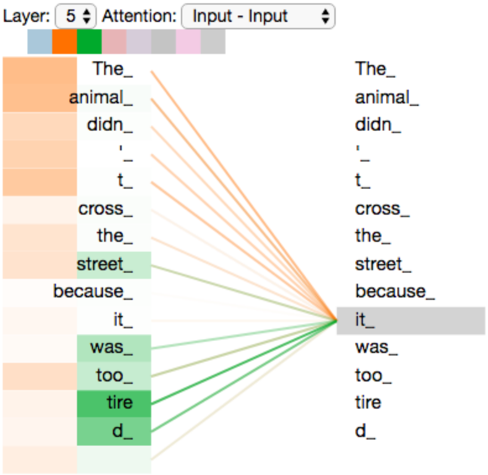
1. Multi-headed attention improves performance by expanding the model's ability to focus on different positions and **representation subspaces**
2. With multi-headed attention, we have multiple sets of Query/Key/Value weight matrices
3. We concatenate these matrices and multiply them by a weight matrix W_o to condense them into a single matrix



Multi-Head Self-Attention (II)

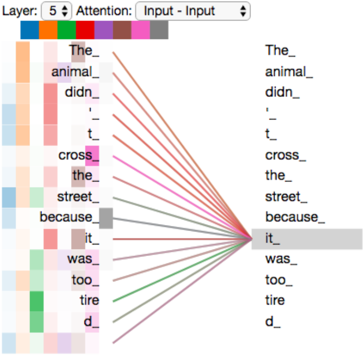


Attention Heads Example



Attention Heads Example (II)

- Different heads focus on different parts of the input
- The model's representation of a word can include information from multiple heads.
- When all heads are combined, the output can be harder to interpret



Outline

Representing The Order of The Sequence Using Positional Encoding

- In order to account for the order in the input sequence, the Transformer adds a vector to each input embedding.
- These vectors follow a specific pattern that the model learns, which helps it determine the position of each word or the distance between different words in the sequence
- The intuition is that adding these values to the embeddings provides meaningful distances between the embedding vectors once they're projected into $Q/K/V$ vectors and during dot-product attention.

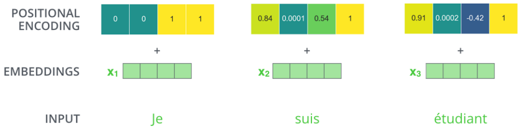


Figure: Example of positional encoding with a embedding size of 4

Real Example of Positional Encoding

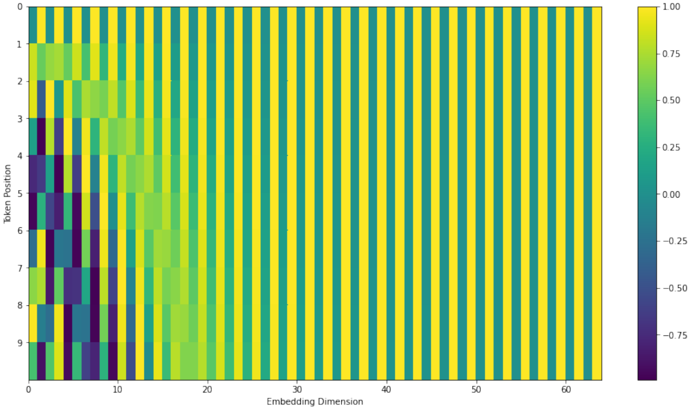


Figure: Real example of positional encoding for 10 words with an embedding size of 64.

Formula for Positional Encoding

- The original *"Attention is all you need"* paper proposes a pre-defined formula for the positional encoding
- The encoding values are generated using a combination of sine and cosine functions, which are concatenated to form each of the positional encoding vectors.
- This method provides the advantage of being able to scale to unseen lengths of sequences, allowing the trained model to translate a sentence longer than any of those in the training set.

$$PE_{(pos,2i)} = sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

$$PE_{(pos,2i+1)} = cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

Positional Encoding Effect

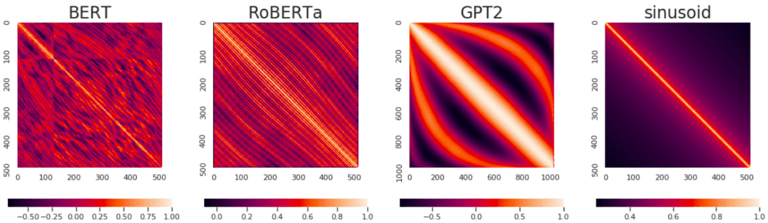


Figure: Position-wise similarity of multiple position embeddings. Note that larger models such as GPT2 process more tokens. Image from *Wang et Chen 2020*

for bert and roberta the pos embedding are learned
while for GPT2 and sinusoid its a formula

Example

For example, with $n = 5$ and $d = 4$, the positional encoding matrix is:

$$\begin{bmatrix} \sin(0/10000^{0/4}) & \cos(0/10000^{0/4}) & \sin(0/10000^{2/4}) & \cos(0/10000^{2/4}) \\ \sin(1/10000^{0/4}) & \cos(1/10000^{0/4}) & \sin(1/10000^{2/4}) & \cos(1/10000^{2/4}) \\ \sin(2/10000^{0/4}) & \cos(2/10000^{0/4}) & \sin(2/10000^{2/4}) & \cos(2/10000^{2/4}) \\ \sin(3/10000^{0/4}) & \cos(3/10000^{0/4}) & \sin(3/10000^{2/4}) & \cos(3/10000^{2/4}) \\ \sin(4/10000^{0/4}) & \cos(4/10000^{0/4}) & \sin(4/10000^{2/4}) & \cos(4/10000^{2/4}) \end{bmatrix}$$

$$\approx \begin{bmatrix} 0.0000 & 1.0000 & 0.0000 & 1.0000 \\ 0.8415 & 0.5403 & 0.0017 & 0.9999 \\ 0.9093 & -0.4161 & 0.0033 & 0.9999 \\ 0.1411 & -0.9900 & 0.0050 & 0.9999 \\ -0.7568 & -0.6536 & 0.0067 & 0.9999 \end{bmatrix}$$

Example (II)

$$\text{PE} \times \text{PE}^T =$$

$$\begin{bmatrix} 2.0000 & 1.3827 & 0.9126 & -0.8476 & -1.4104 \\ 1.3827 & 1.8325 & 0.8479 & -0.6605 & -1.1834 \\ 0.9126 & 0.8479 & 1.6806 & 0.6541 & -0.4506 \\ -0.8476 & -0.6605 & 0.6541 & 1.6989 & 1.2061 \\ -1.4104 & -1.1834 & -0.4506 & 1.2061 & 1.7998 \end{bmatrix}$$

We see that the elements in the diagonal have a much larger value. And the diagonal decreases the farther away from the diagonal. That is, closer positions have higher dot product values.

Outline

Training trick 1: Residual Connections

Residual connections

Residual connections are a technique that helps models train better. Instead of only propagating information forward through a series of layers, residual connections also allow information to flow directly through the layers via a shortcut connection.

- Residual connections were first introduced in the ResNet architecture for image classification.
- They can help address the vanishing gradient problem and improve gradient flow through the network.
- Residual connections have been shown to be effective in a wide range of deep learning models.
- This technique can make the loss landscape considerably smoother and make the training process easier and more efficient.

Training trick 2: Layer Normalization

Layer normalization

Layer normalization is a technique that helps models train faster by cutting down on uninformative variation in hidden vector values. This is achieved by normalizing the values to have a zero mean and unit standard deviation within each layer.

- Layer normalization is similar to batch normalization, but is applied at the layer level instead of the batch level.
- It has been shown to improve the performance of a wide range of deep learning models, including transformers.
- This technique can help address issues with internal covariate shift and the vanishing gradient problem.

also avoids expIding gradient

Training trick 2: Layer Normalization (II)

The equation for layer normalization is:

$$\text{LayerNorm}(x_i) = \frac{a_i}{\sqrt{\sigma^2 + \epsilon}}(x_i - \mu) + b_i$$

where x_i is the input to the ith layer, μ and σ are the mean and standard deviation of the input, a_i and b_i are learnable scaling and shifting parameters for each layer, and ϵ is a small value (usually 10^{-5}) added for numerical stability.

Training trick 3: Scaled Dot Product Attention

Dot product attention

The dot product in the attention mechanism tends to take on extreme values, as its variance scales with dimensionality d.

Solution

To mitigate this issue, we can use a scaling factor of $1/\sqrt{d}$ for the dot product, which is called scaled dot product attention.

- Scaled dot product attention is used in the self-attention mechanism in transformers.
- This technique ensures that the dot product stays within a reasonable range, which can help with the stability of the training process.
- Scaled dot product attention is also more computationally efficient than other attention mechanisms

The Attention Block

- Each sub-layer in each encoder has a residual connection around it and is followed by a layer-normalization step.

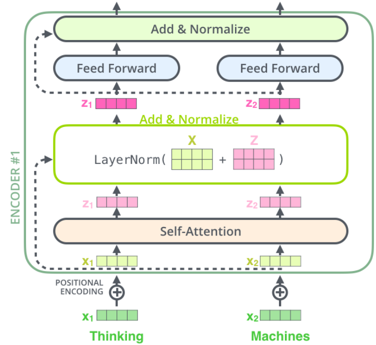


Figure: Visualization of the vectors and layer-norm operation associated with self-attention.

The Attention Block (II)

the output of softmax vocabulary is not words but BPE tokens that will need to be stitched to get the text

- This also applies to the decoder. A Transformer with 2 stacked encoders and decoders would look like this:

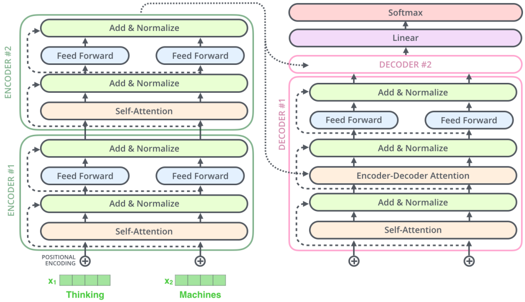


Figure: A Transformer with 2 stacked encoders and decoders.

Outline

Fixing the decoder problem: Masked attention

Problem

How do we prevent the decoder in a transformer network from "cheating" by looking ahead and "seeing" the answer when training on a language modeling objective?

Solution

Masked Multi-Head Attention: We mask (hide) information about future tokens from the model by setting attention scores to $-\infty$.

- To use self-attention in decoders, we need to ensure we can't peek at the future.
- At every timestep, we could change the set of keys and queries to include only past words, but this would be inefficient.

Fixing the decoder problem: Masked attention (II)

Masked attention

To enable parallelization, we mask out attention to future words by setting attention scores to $-\infty$.

- Masking is achieved by adding a mask matrix to the attention weights before the softmax operation.
- The mask matrix has the same shape as the attention weights, with $-\infty$ values in the positions where future words would be attended to.

Mask matrix

$$\text{mask}_{i,j} = \begin{cases} 0 & \text{if } j \leq i \\ -\infty & \text{otherwise} \end{cases}$$

- The masked attention scores are then passed through a softmax activation function to compute the weights

Outline

Encoder-Decoder Attention

- The output of the top encoder is transformed into attention vectors K and V to be used in the "encoder-decoder attention" layer of each decoder.
- Each step in the decoding phase outputs an element from the output sequence and repeats until a special symbol is reached.
- The self-attention layer in the decoder can only attend to earlier positions in the output sequence, achieved by masking future positions.
- The "encoder-decoder attention" layer works like multiheaded self-attention, with the Queries matrix from the layer below and the Keys and Values matrix from the output of the encoder stack.

The Decoder

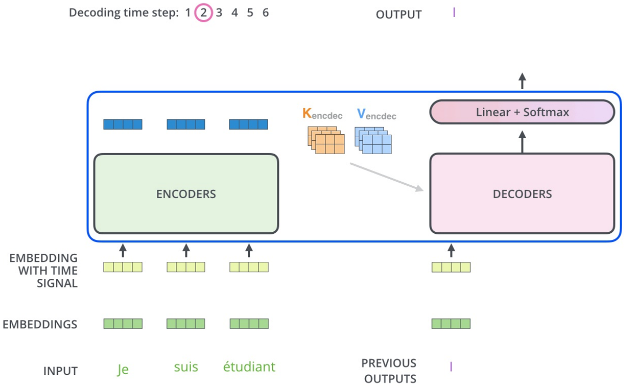


Figure: The decoder side of the Transformer architecture.

The Decoder (II)

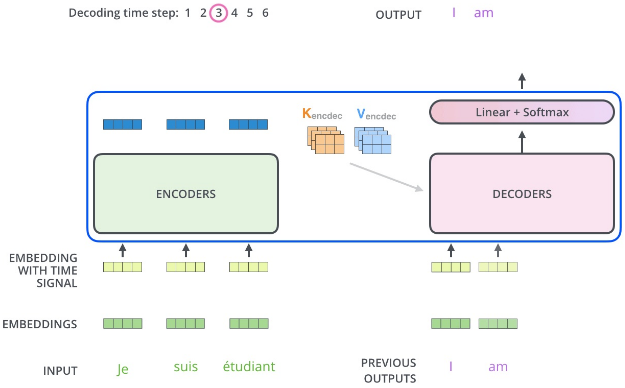


Figure: The decoder side of the Transformer architecture.

The Final Linear and Softmax Layer

- The final Linear layer projects the vector produced by the decoder to the vocabulary size
- The softmax layer turns those scores into probabilities

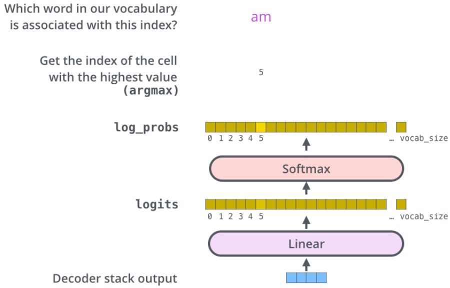


Figure: The final Linear and Softmax layers in the Transformer architecture.