Master on Artificial Intelligence

Advanced Human Language Technologies

Transformers

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Outline

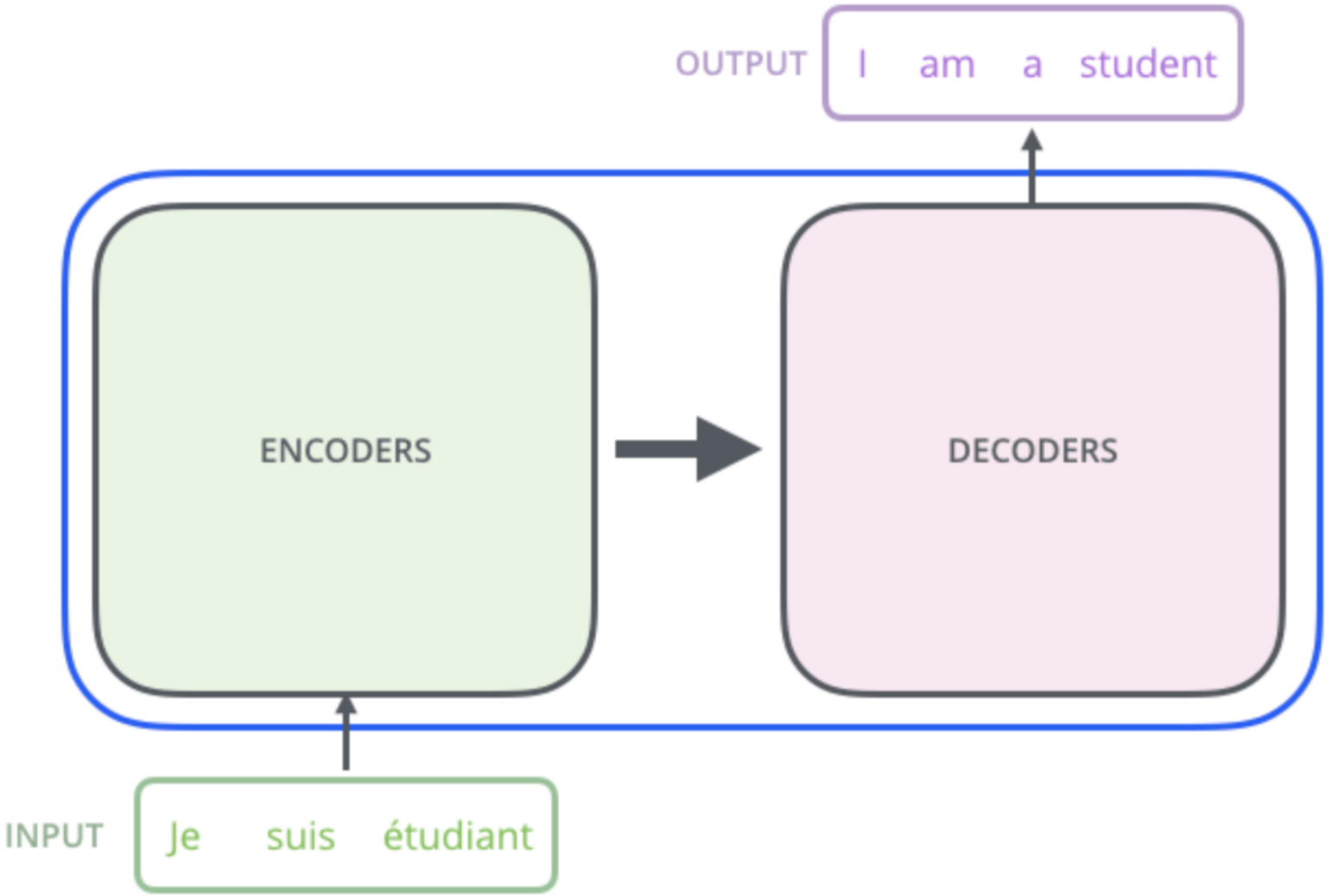
Outline

Sequence-to-Sequence Tasks

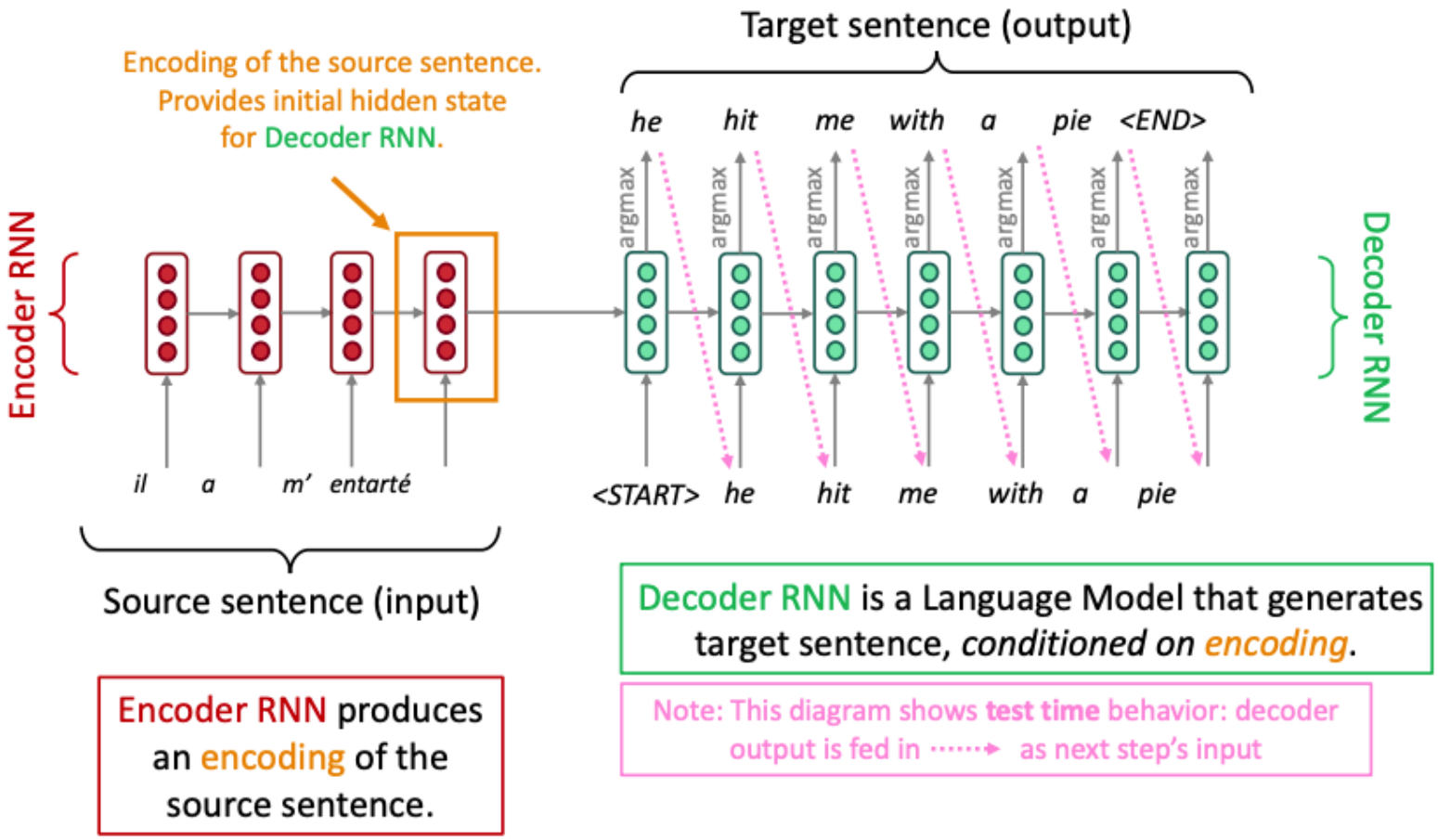
Many NLP tasks can be phrased as sequence-to-sequence:

- 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)

Sequence-to-Sequence Model



Sequence-to-Sequence Model



Outline

Neural Machine Translation

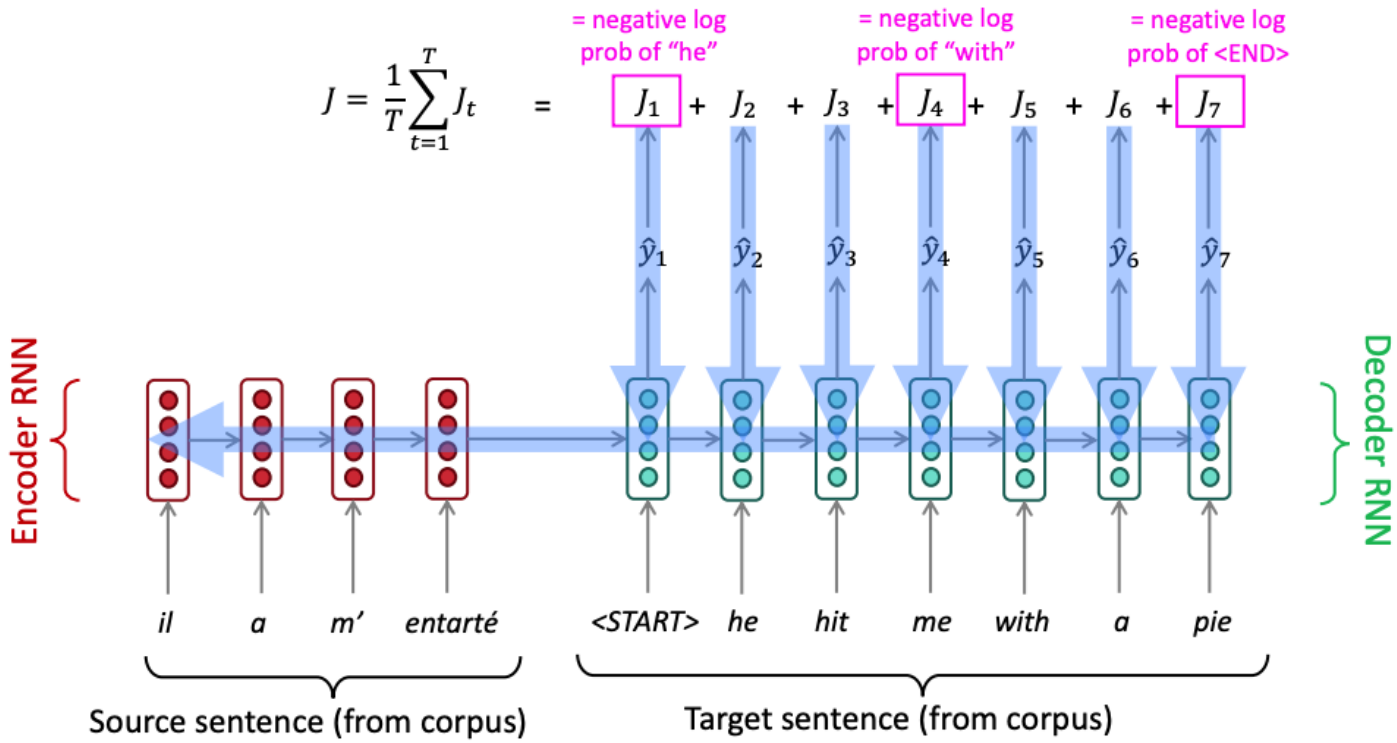
- The sequence-to-sequence model is an example of a Conditional Language Model.
- Language Model because the decoder is predicting the next word of the target sentence y.
- Conditional because its predictions are also conditioned on the source sentence x.

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

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

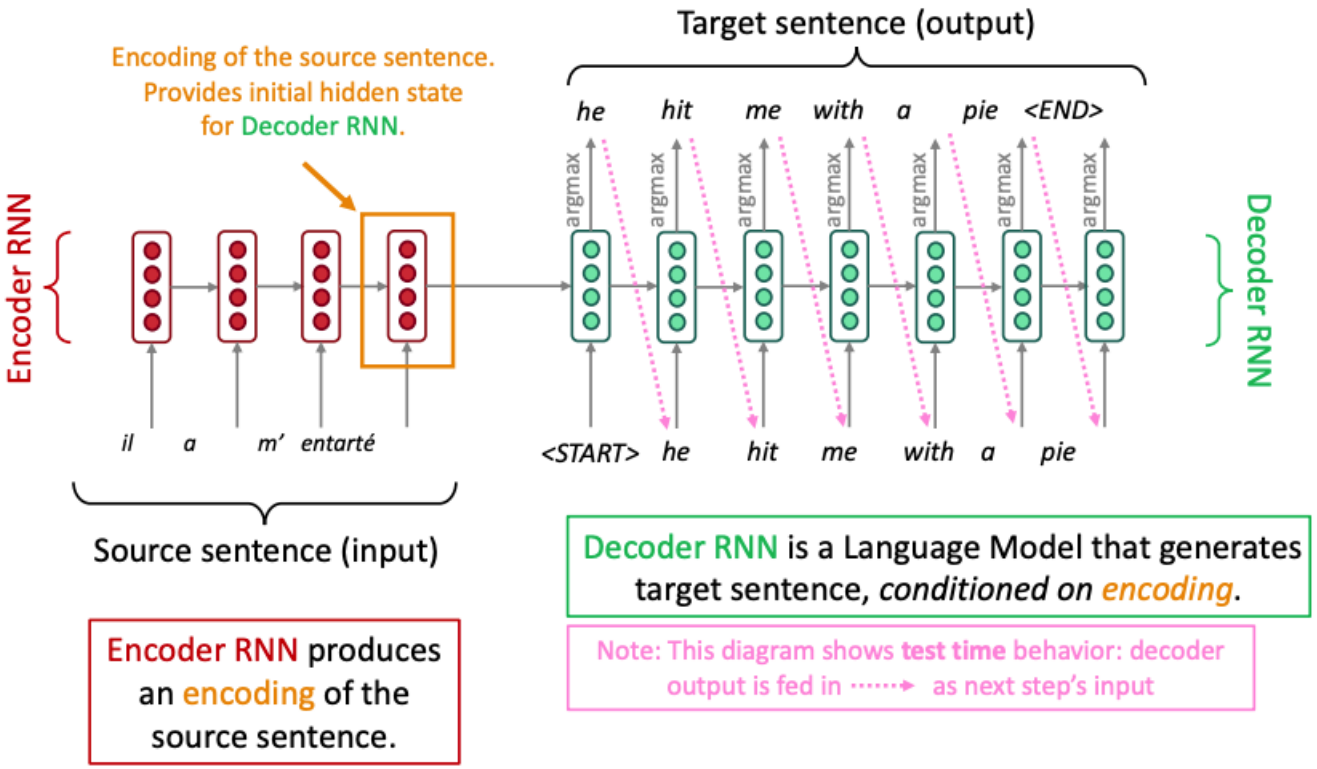
Teacher Forcing

- 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

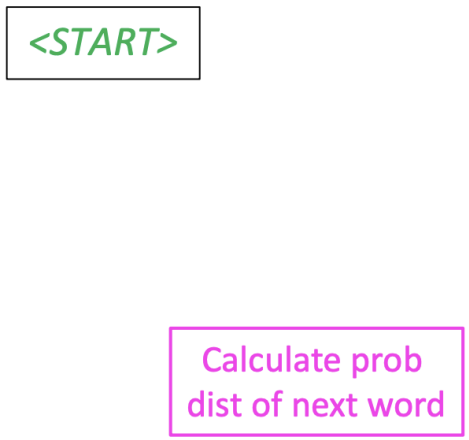
$$\arg\max_y P(y|x) = \arg\max_y \prod_{t=1}^{|y|} P(y_t|y_{<t}, x)$$

Beam Search Decoding

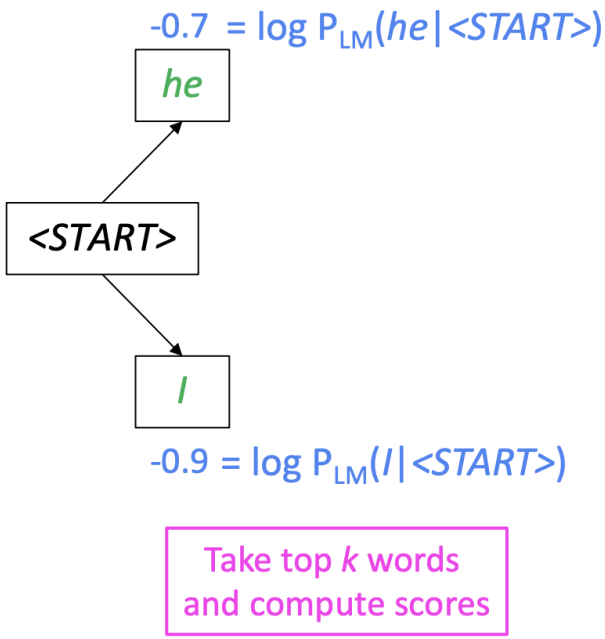
- Beam search is a more efficient decoding algorithm than exhaustive search
- On each step of the decoder, we keep track of the k most probable partial translations (which we call hypotheses)
- Each hypothesis has a score, its log probability
- We search for high-scoring hypotheses, tracking top k on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

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

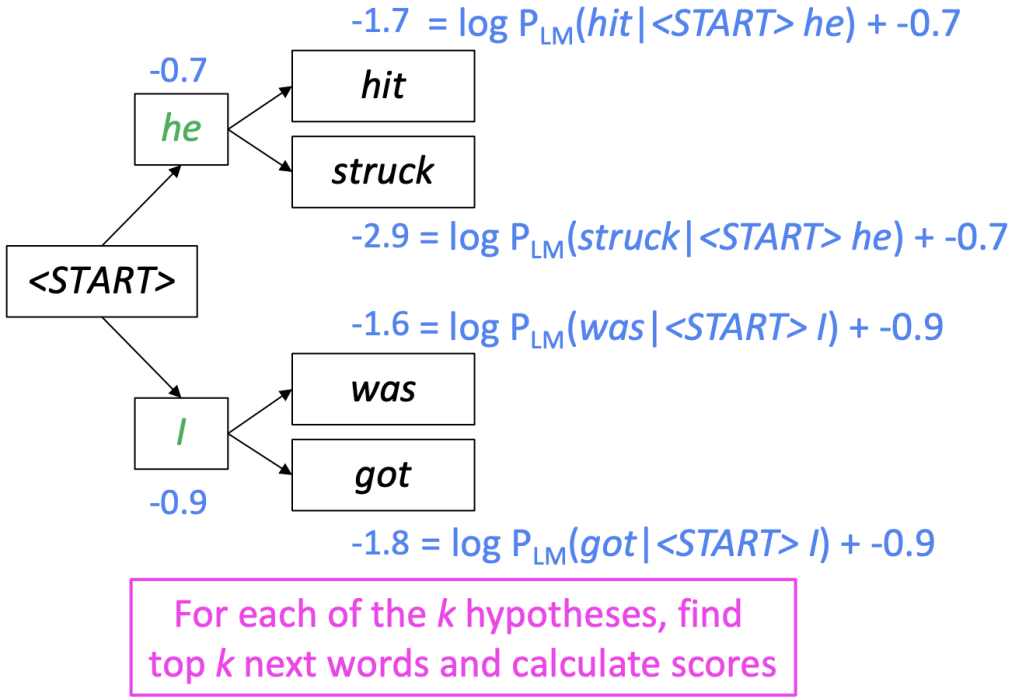
Beam Search Decoding (k=2)



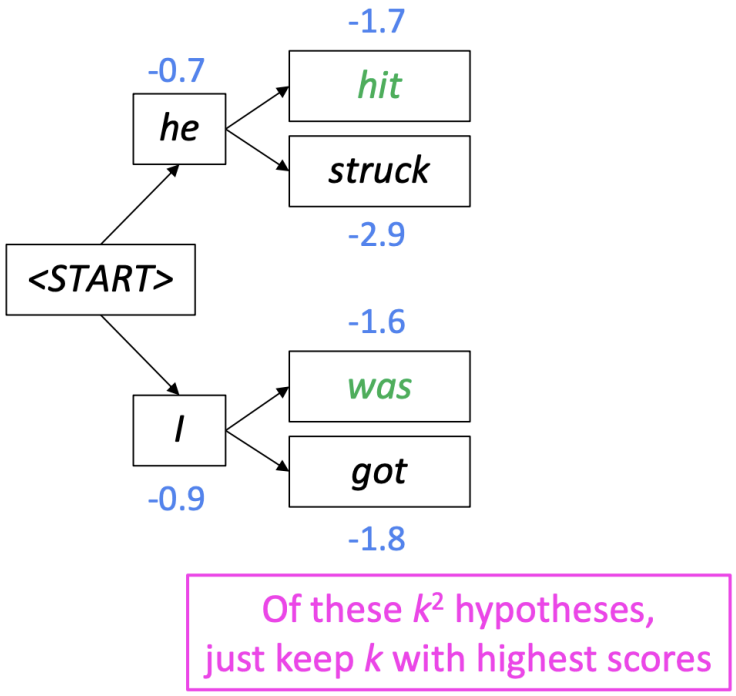
Beam Search Decoding (k=2)



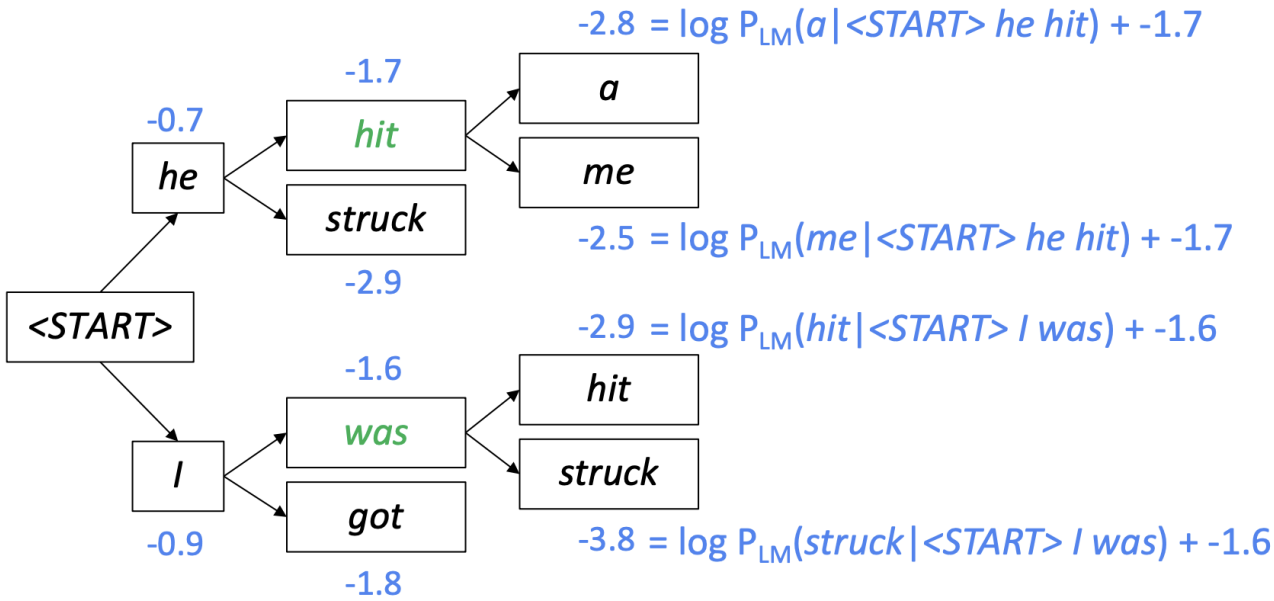
Beam Search Decoding (k=2)



Beam Search Decoding (k=2)

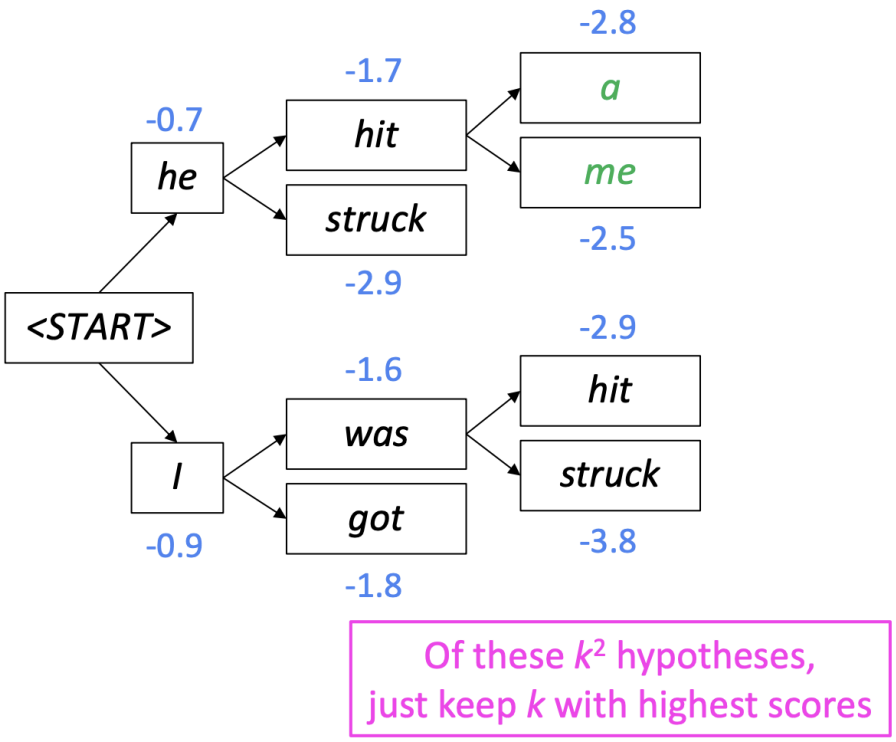


Beam Search Decoding (k=2)

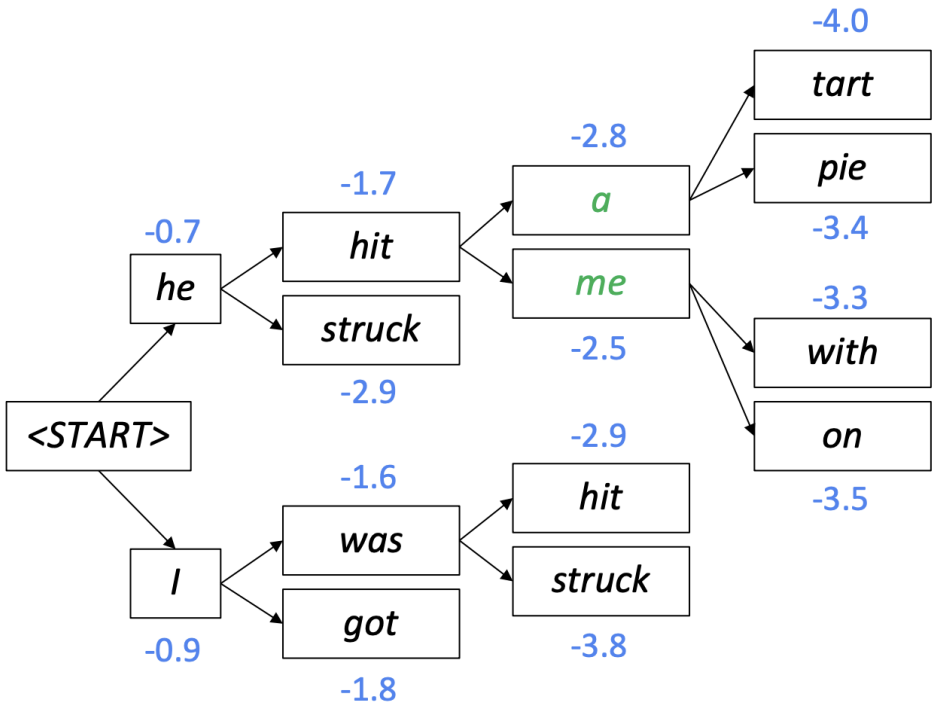


For each of the *k* hypotheses, find top *k* next words and calculate scores

Beam Search Decoding (k=2)

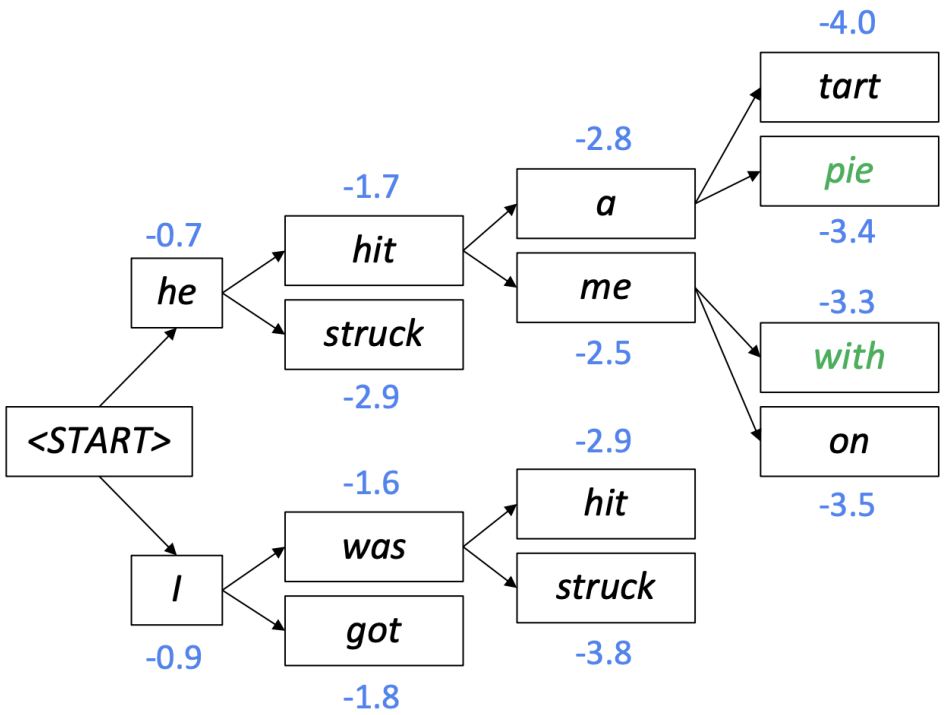


Beam Search Decoding (k=2)



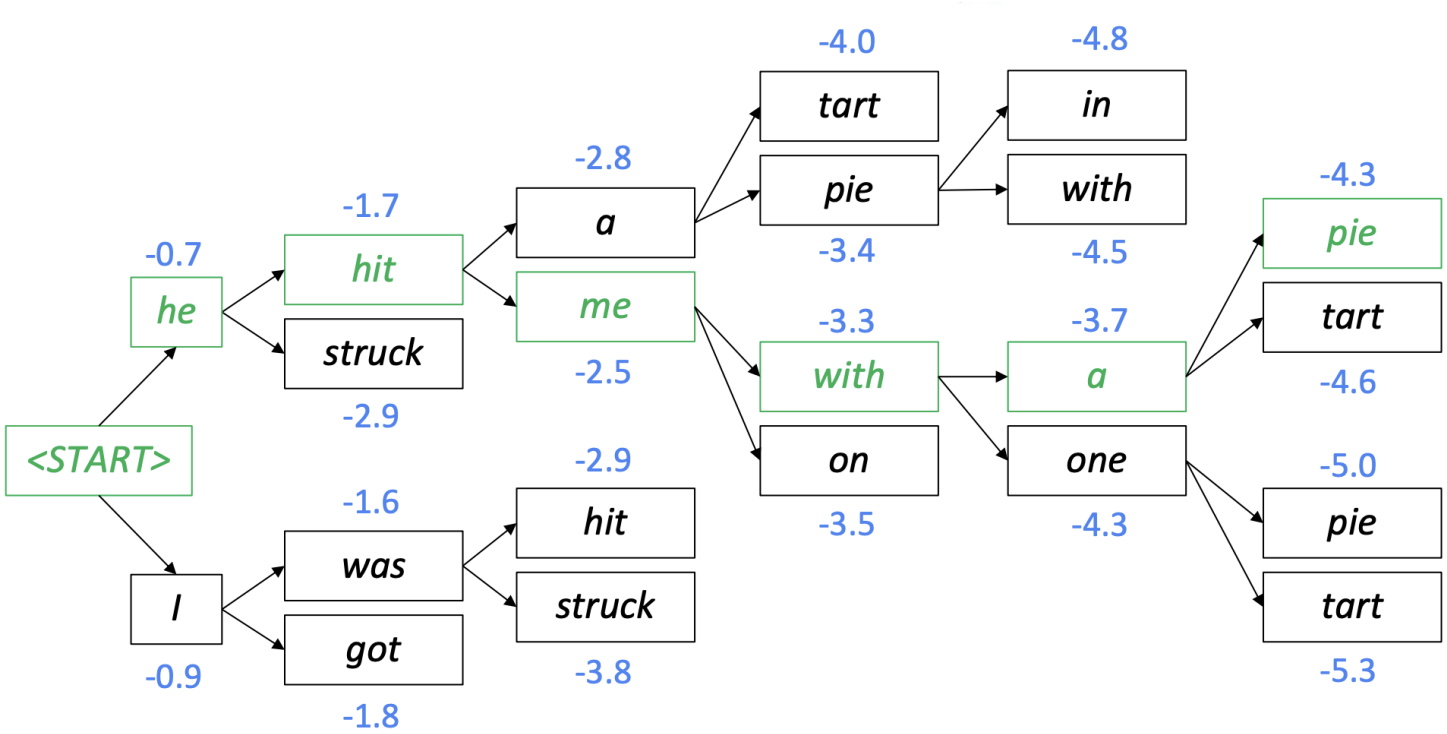
For each of the *k* hypotheses, find top *k* next words and calculate scores

Beam Search Decoding (k=2)



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

Beam Search Decoding (k=2)



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: longer hypotheses have lower scores.
- Solution: Normalize by length, selecting hypothesis with higher probability:

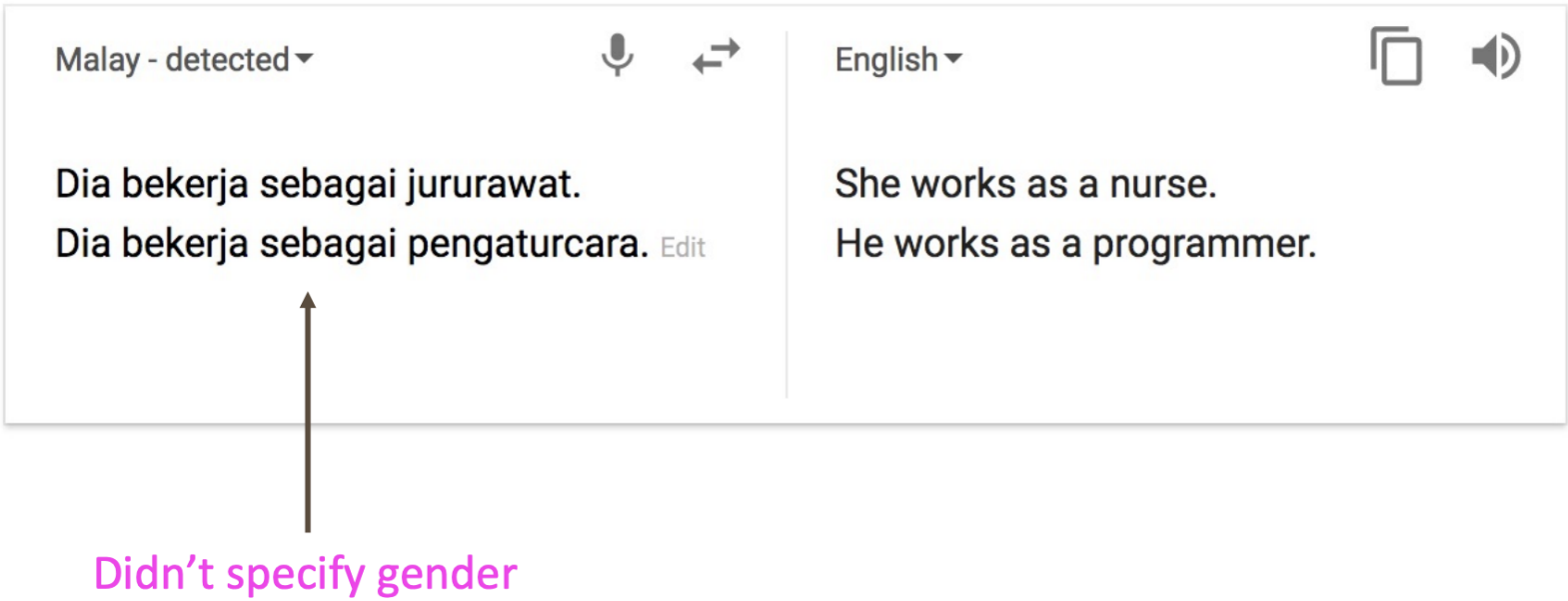
$$\hat{y} = \underset{y \in \mathcal{Y}}{\operatorname{argmax}} \frac{\log p(y|x)}{|y|^{\alpha}}$$

- the α parameter allows for fine-tuning between favoring shorter or longer sequences:
  - $\alpha = 1$: Normalize by sequence length, computing the average log probability per token.
  - $\alpha < 1$: Reduces the penalty on longer sequences, potentially favoring them.
  - $\alpha > 1$: Increases the penalty on longer sequences, potentially favoring shorter ones.

Outline

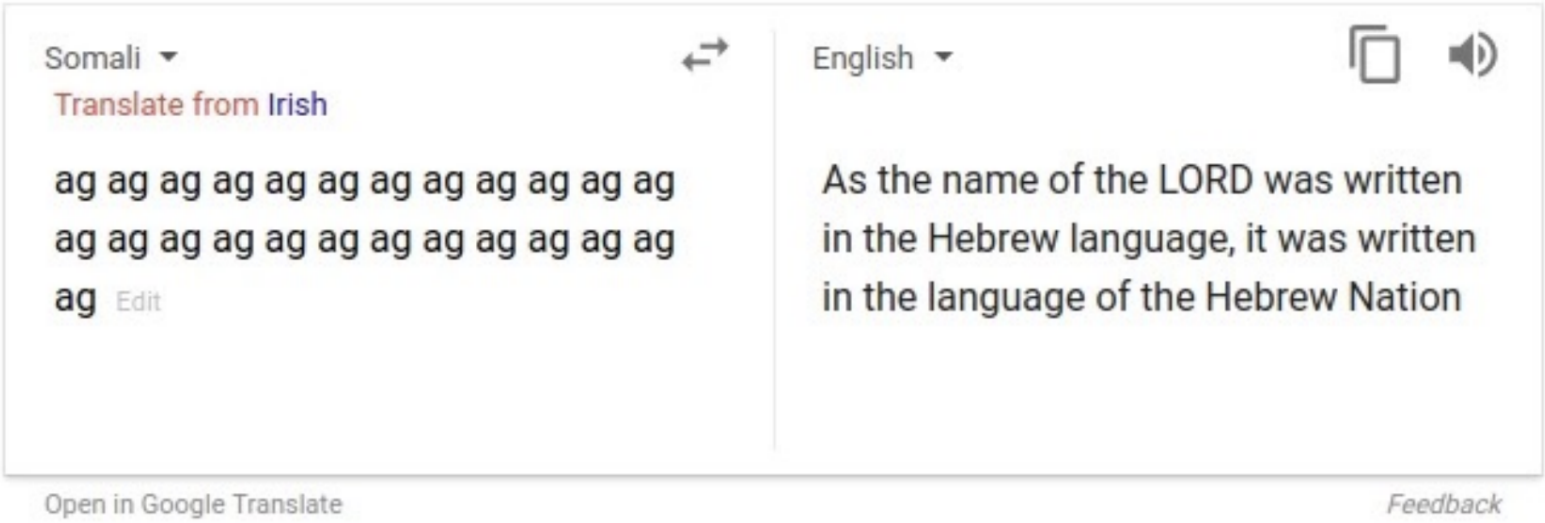
Seq2Seq does not solve MT

NMT picks up biases in training data

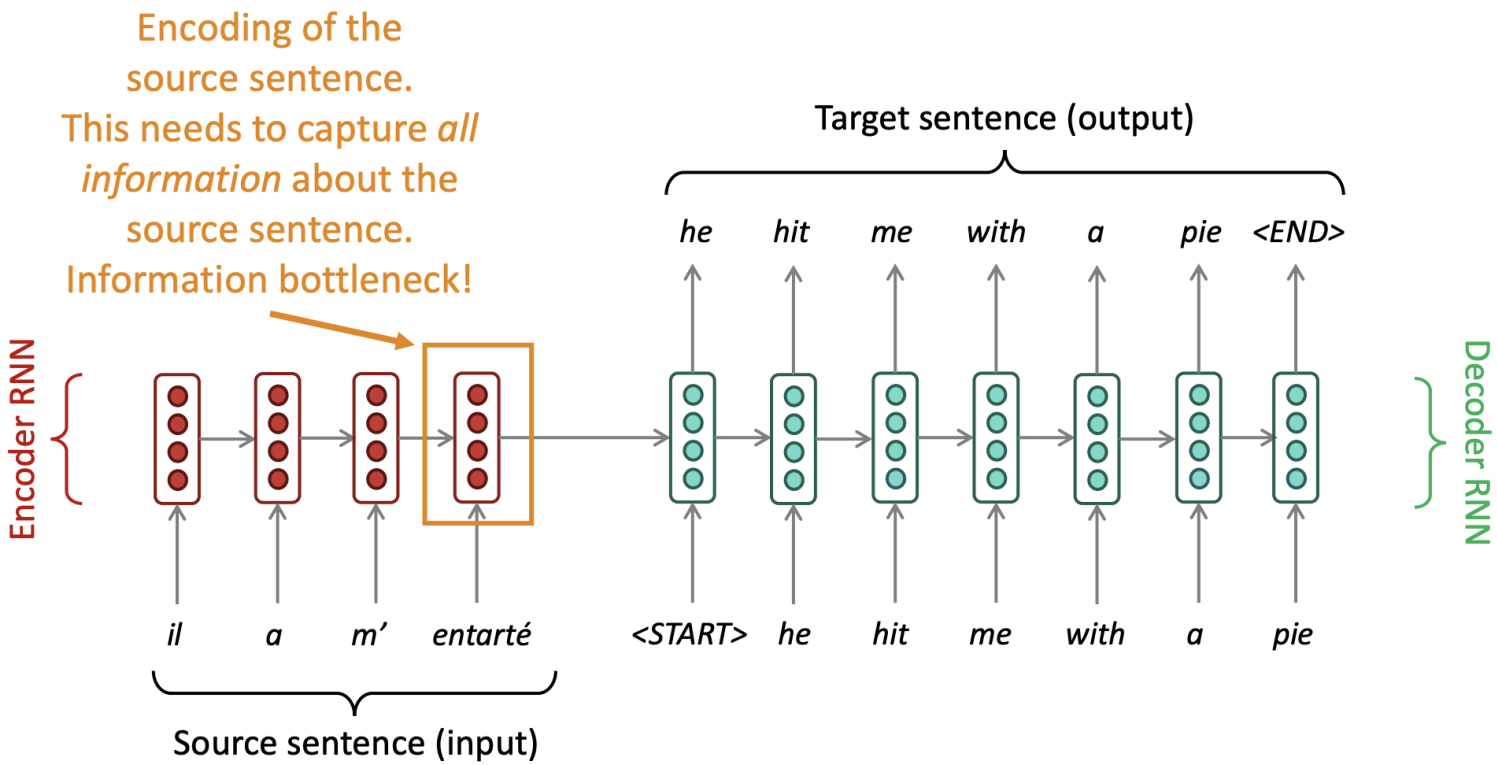


Seq2Seq does not solve MT

Hard to interpret systems do strange things



Sequence-to-sequence: the bottleneck problem



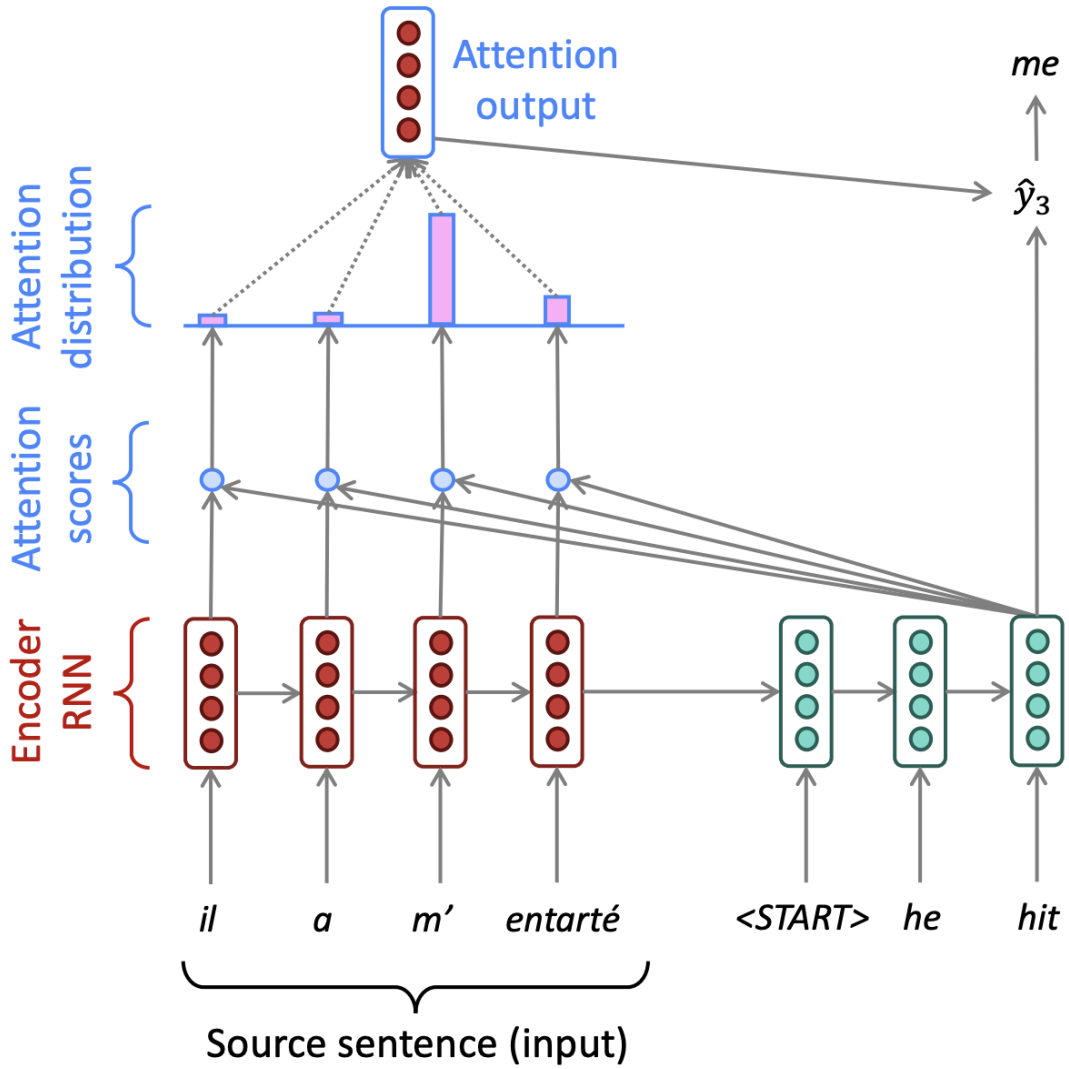
Outline

Outline

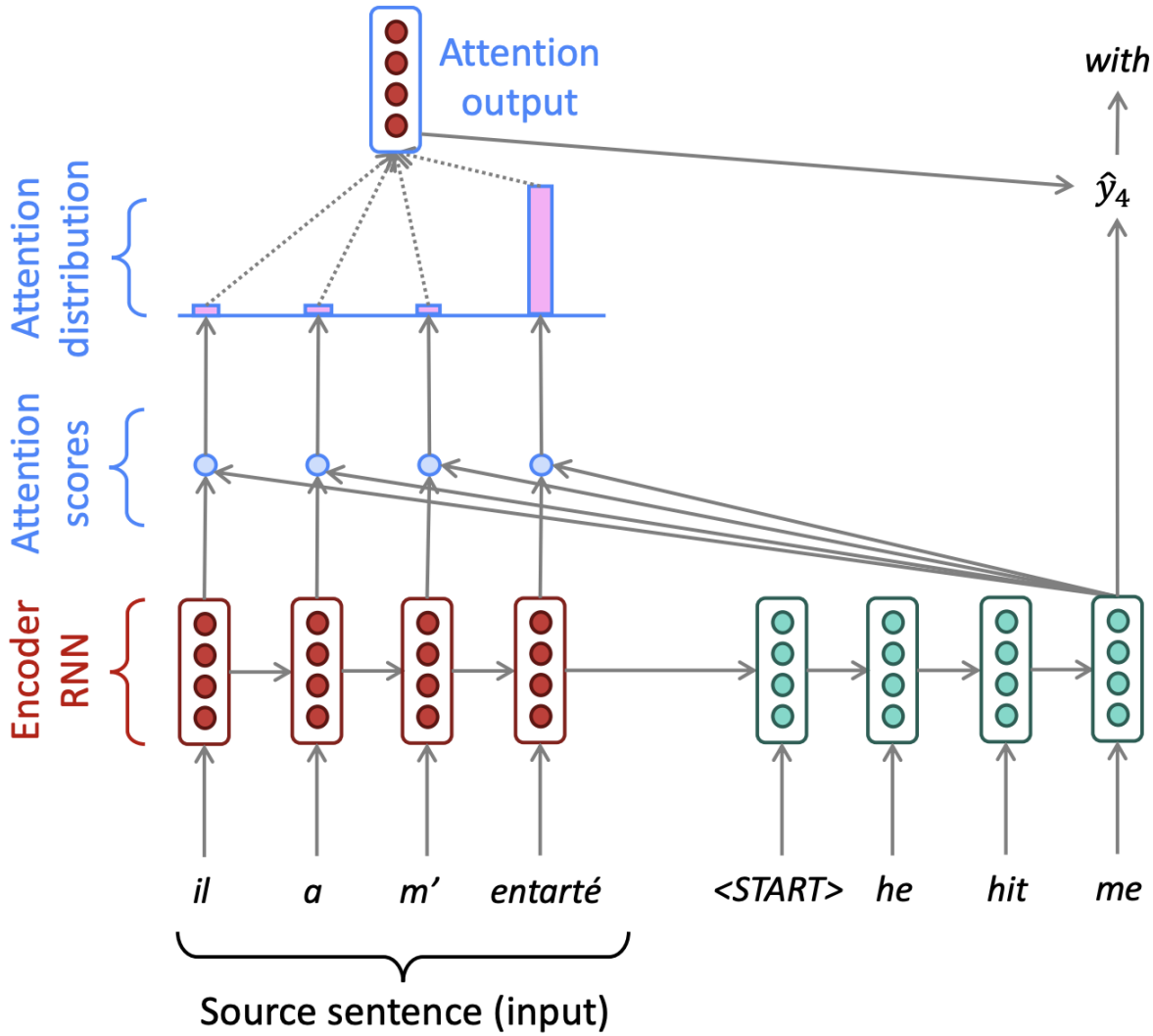
Attention

- 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

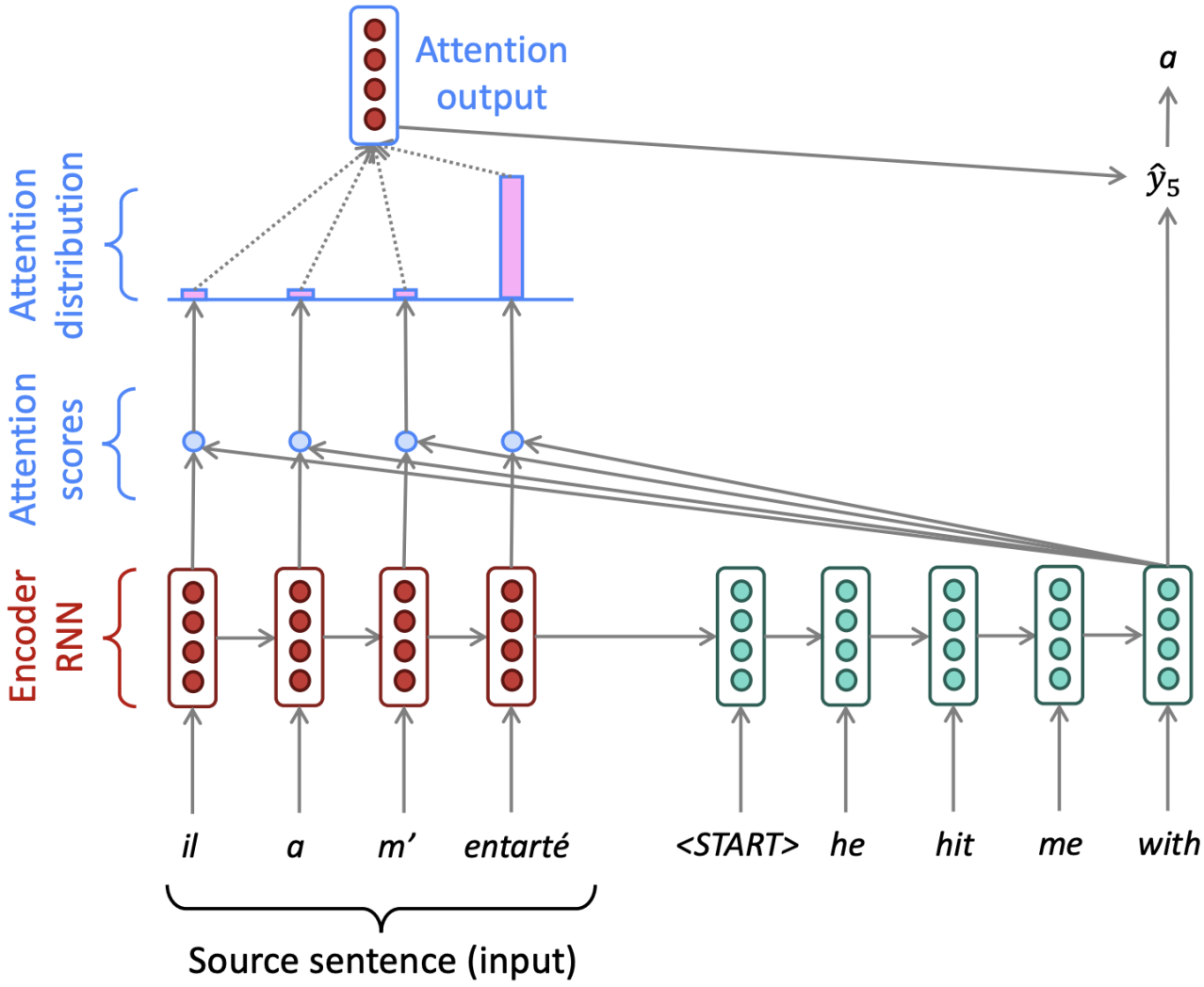
Attention



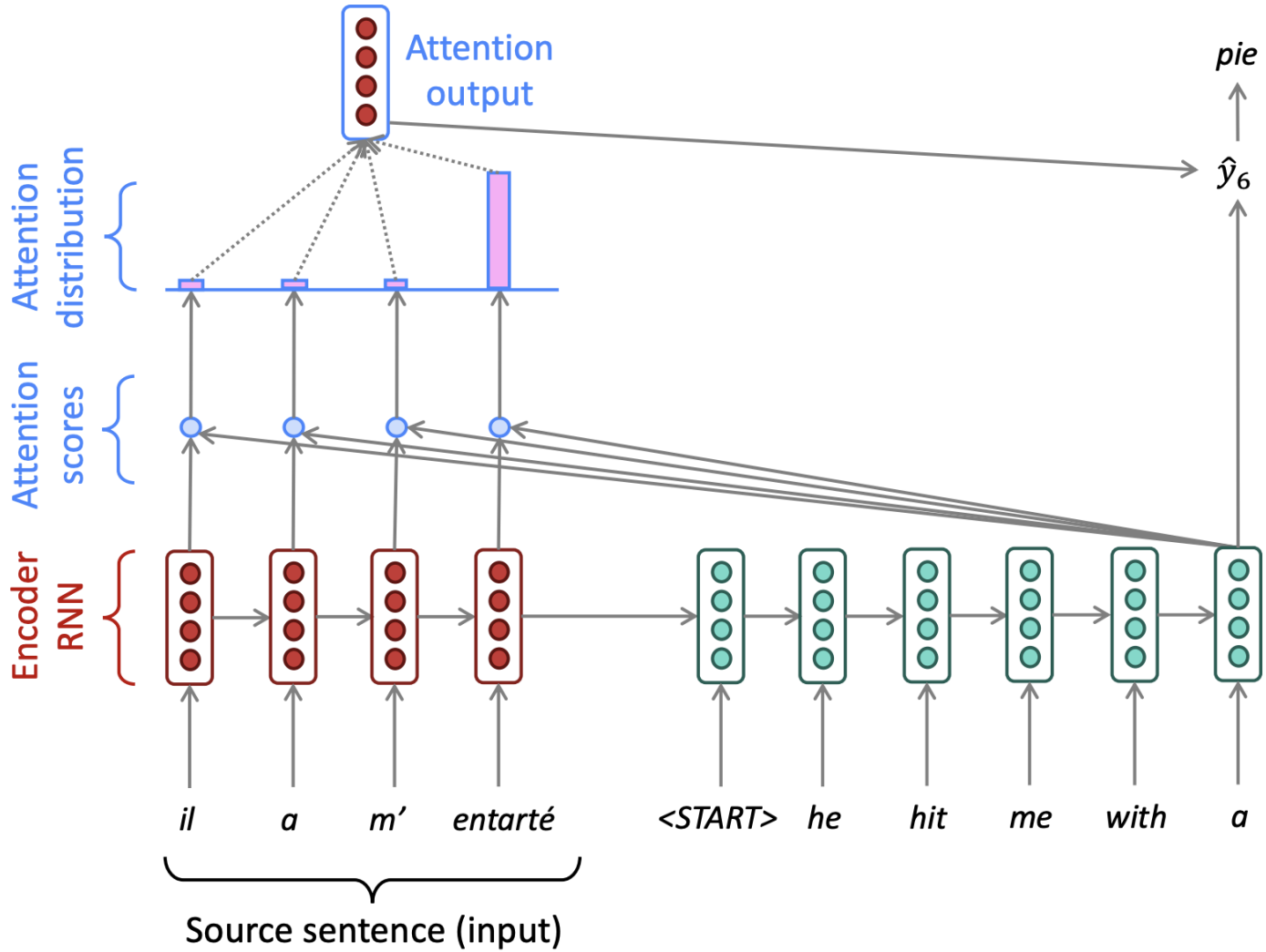
Attention



Attention



Attention



Outline

Basic Attention (with no parameters)

Suppose we have a sequence of n items (or *keys*), represented as $k_1, k_2, \ldots, k_n$, and we want to compute the attention weights for each item based on a query q.

$$s_i = k_i^T q$$

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

$$\text{Attention}(q, k_1, k_2, \ldots, k_n) = \sum_{i=1}^{n} w_i k_i$$

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).

Generalized Dot-Product Attention Formulas

In the dot-product attention mechanism, we compute the attention as a weighted sum of the value vectors, where value weights are given by the dot product of the query and key vectors:

$$\text{Attention}(Q, K, V) = \text{softmax}(QK^T)V = \sum_{i=1}^{n_k} \frac{\exp(qk_i)}{\sum_{j=1}^{n_k} \exp(qk_j)} v_i$$

where:

- Q $(n_q \times d_k)$ is the query matrix
- K $(n_k \times d_k)$ is the key matrix
- V $(n_k \times d_v)$ is the value matrix

d_k: dimensionality of key vectors
n_q: number of queries
n_k: number of keys
d_v: dimensionality of value vectors.

Generalized Attention Formulas

- We can generalise to other attention models using different similarity metrics to compute the value weights:

$$w_i = \text{softmax}(\text{score}(q, k_i))$$

- score is a similarity metric between the query vector and each input vector. Common choices are:
  - Dot product: $\text{score}(q, k_i) = q^T k_i$
  - Scaled dot product: $\text{score}(q, k_i) = q^T k_i / \sqrt{d}$
  - General: $\text{score}(q, k_i) = q^T W k_i$
  - Concat: $\text{score}(q, k_i) = v^T \tanh(W[q; k_i])$
  - Additive: $\text{score}(q, k_i) = v^T \tanh(W_1 q + W_2 k_i)$

d: dimensionality of query and input vectors
W, v: learned parameter matrices
$[q; k_i]$: concatenation of q and k_i

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

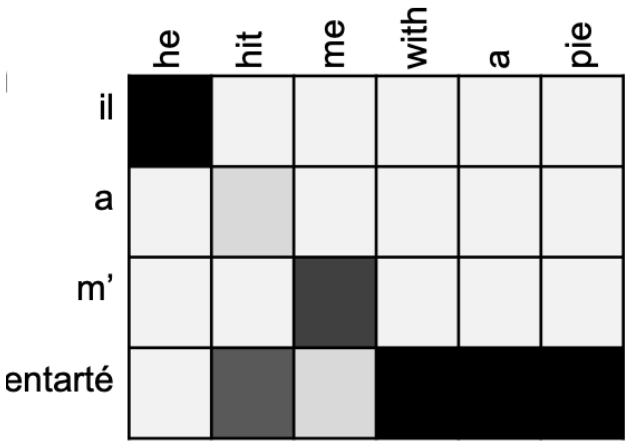
- In a traditional seq2seq model, the decoder receives a **fixed-length vector** (the context vector) that summarizes the entire source sequence. This creates a bottleneck, since the decoder must use this one vector to generate the entire target sequence.
- In an attention model, the whole encoded sequence is accessible by the decoder. Attention weights $\alpha_{i,j}$ determine which parts of the source sequence to focus on at each step.

$$e_{i,j} = v_a^T \tanh(W_a s_{i-1} + U_a h_j)$$
$$\alpha_{i,j} = \frac{\exp(e_{i,j})}{\sum_{k=1}^{n} \exp(e_{i,k})}$$
$$c_i = \sum_{j=1}^{n} \alpha_{i,j} h_j$$

s_{i-1}: decoder hidden state at previous time step
h_j: j-th encoder hidden state
v_a, W_a, U_a: learned parameters
c_i: context vector at time step i

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

Issues with RNNs: Linear interaction distance

Linear locality

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

- RNNs take $O(n)$ steps for distant word pairs to interact, which makes it hard to learn long-distance dependencies due to vanishing gradients.
- Meaning in sentences doesn't necessarily follow a *linear order*.
- This interaction distance limitation can be problematic for some NLP tasks, such as machine translation or sentiment analysis.

Solution

Transformers can capture non-linear dependencies more effectively, thanks to their attention mechanism.

Issues with RNNs: Lack of parallelizability

Parallelizability

Forward and backward passes in RNNs have $O(n)$ 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 hampers training on very large datasets, which can be a major issue in modern NLP applications.

Solution

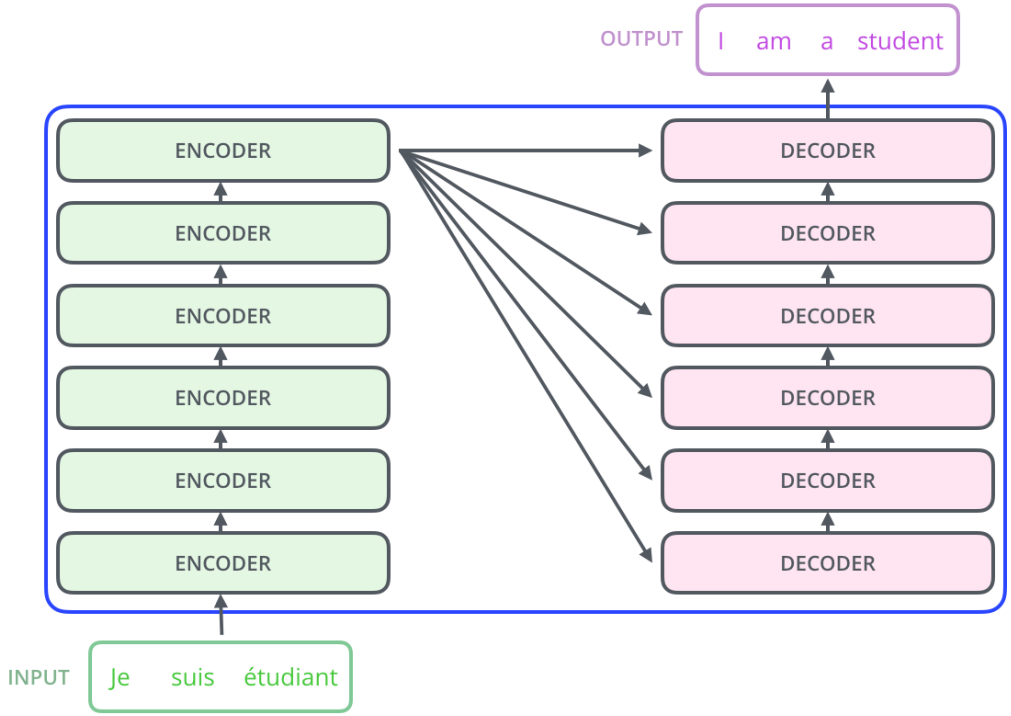
Transformers are highly parallelizable, allowing for much faster training on larger datasets.

The Transformer Model

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

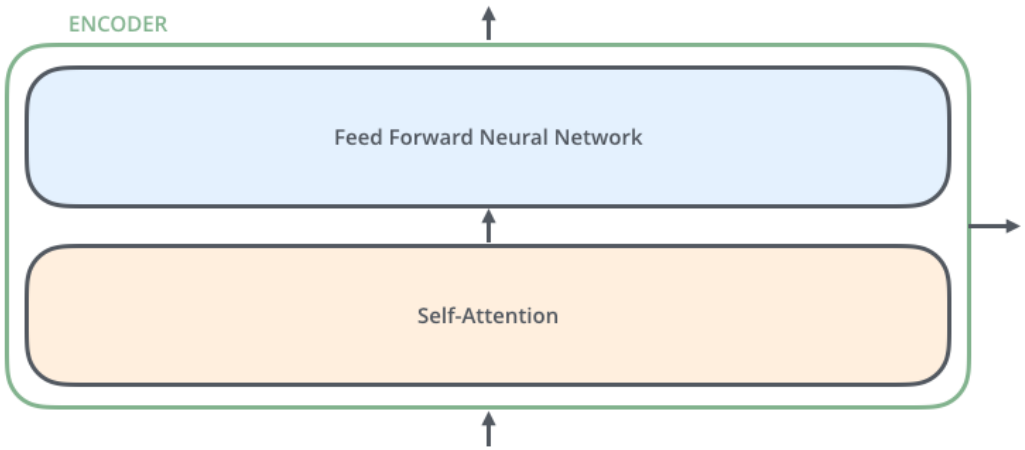
The Encoding and Decoding Components

- 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.



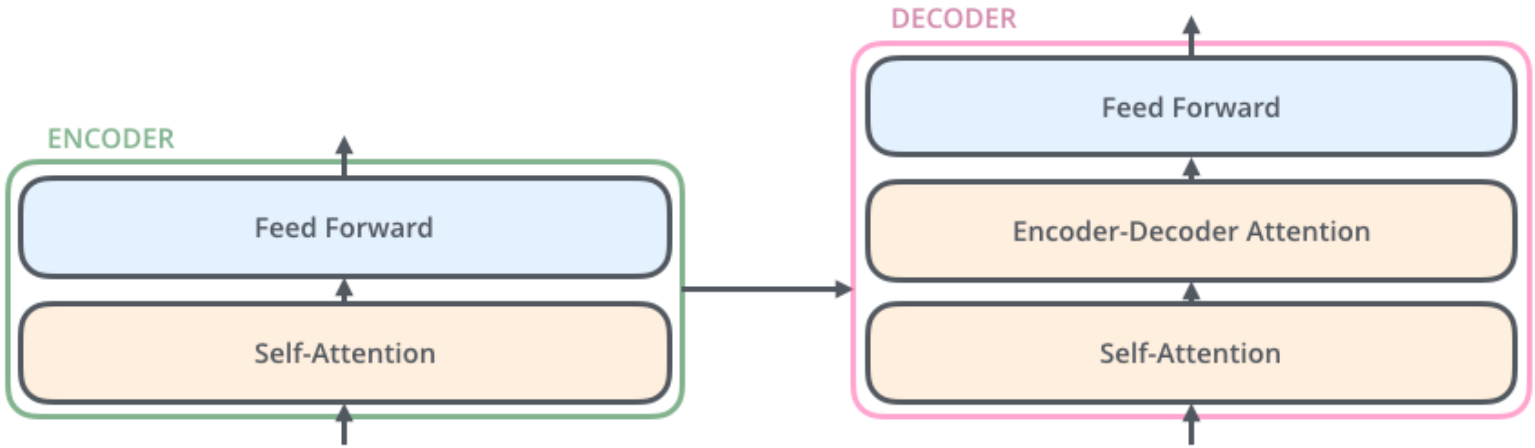
The Encoder Sub-Layers

- 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.

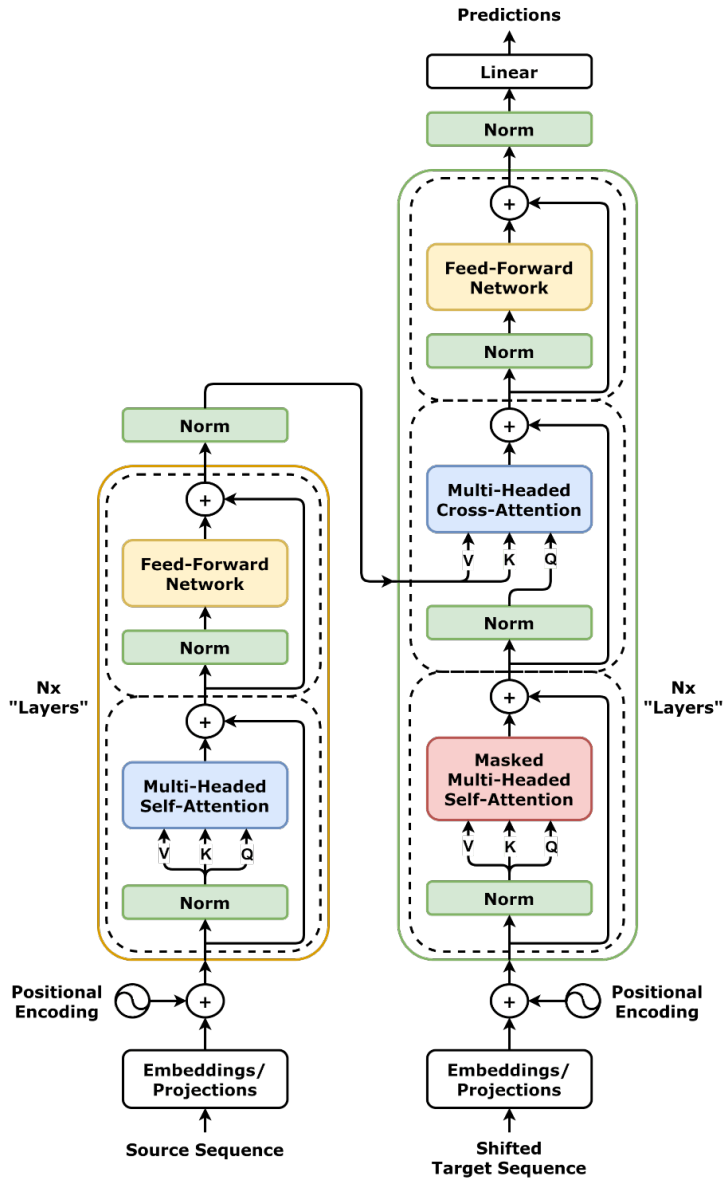


The Decoder Sub-Layers

- 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.

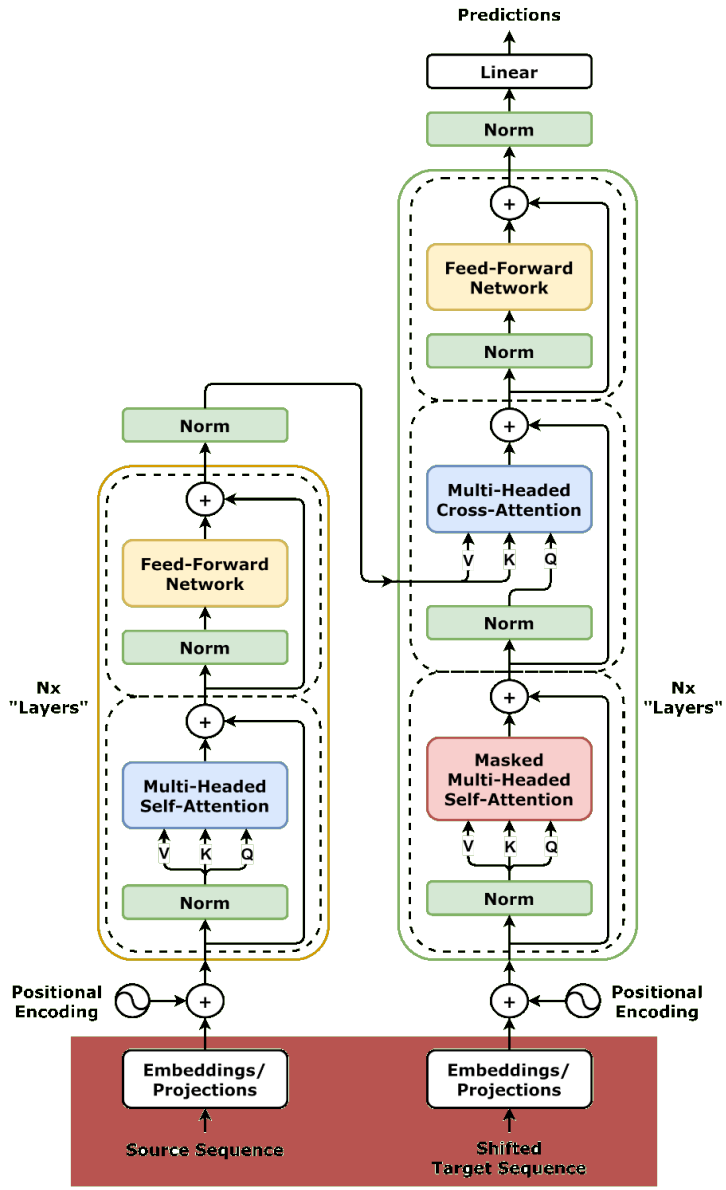


Detailed architecture



Outline

The Input Embedding



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.

The Input Embedding

Byte Pair Encoding

- BPE: 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 NLP tasks such as subword tokenization and neural machine translation.

The Input Embedding

Byte Pair Encoding Algorithm

1. Initialize a vocabulary with all the 1-gram tokens in the text data.
2. Count the frequency of all the token pairs in the text data.
3. Merge the most frequent token pair into a new token and add it to the vocabulary.
4. Encode the text data by replacing each new token pair with its corresponding merged token.
5. Repeat steps 2-4 until a desired vocabulary size or compression ratio is reached.

The Input Embedding

Byte Pair Encoding Example

- Assume we have as text data:
  `low lower newest widest`
- Initial vocabulary is:
  {`l`, `o`, `w`, `[wsp]`, `e`, `r`, `n`, `s`, `t`, `i`, `d`}
- The most frequent token pairs are: `lo` and `es`
- We merge each pair into a new token and add them to the vocabulary, which is now:
  {`l`, `o`, `w`, `[wsp]`, `e`, `r`, `n`, `s`, `t`, `i`, `d`, ⟨`lo`⟩, ⟨`es`⟩}
- The re-encoded text data is:
  ⟨`lo`⟩`w` ⟨`lo`⟩`wer new`⟨`es`⟩`t wid`⟨`es`⟩`t`

The Input Embedding

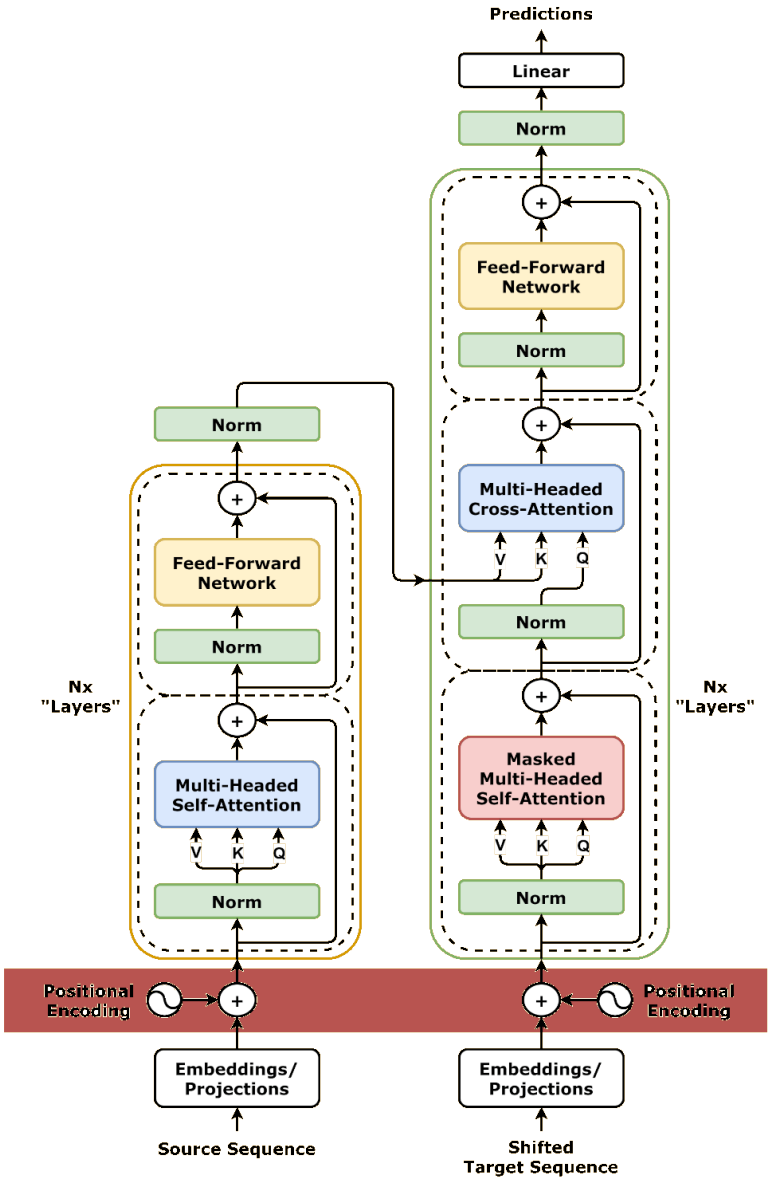
Byte Pair Encoding Example

- We repeat the process with new encoded data:
  ⟨lo⟩w ⟨lo⟩wer new⟨es⟩t wid⟨es⟩t
- The most frequent token pairs are: ⟨lo⟩w and ⟨es⟩t
- We merge them as new tokens and add them to the vocabulary, which is now:
  {l, o, w, [wsp], e, r, n, s, t, i, d, ⟨lo⟩, ⟨es⟩, ⟨low⟩, ⟨est⟩}
- The re-encoded text data is:
  ⟨low⟩ ⟨low⟩er new⟨est⟩ wid⟨est⟩

Original text of 23 1-char tokens has been encoded in a 15-token sequence (0.65 compression rate).

Outline

Positional Encoding



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 during dot-product attention.

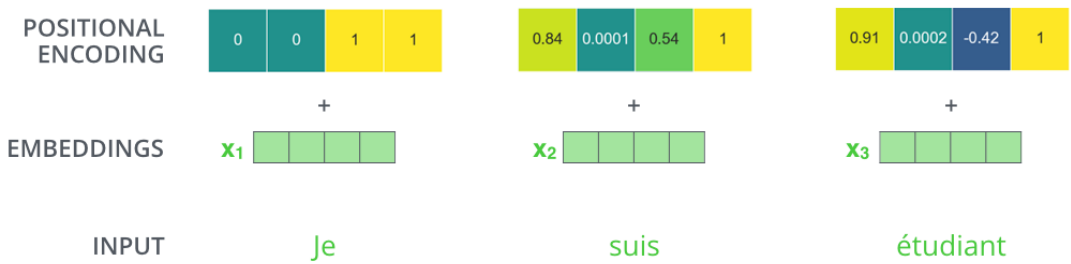


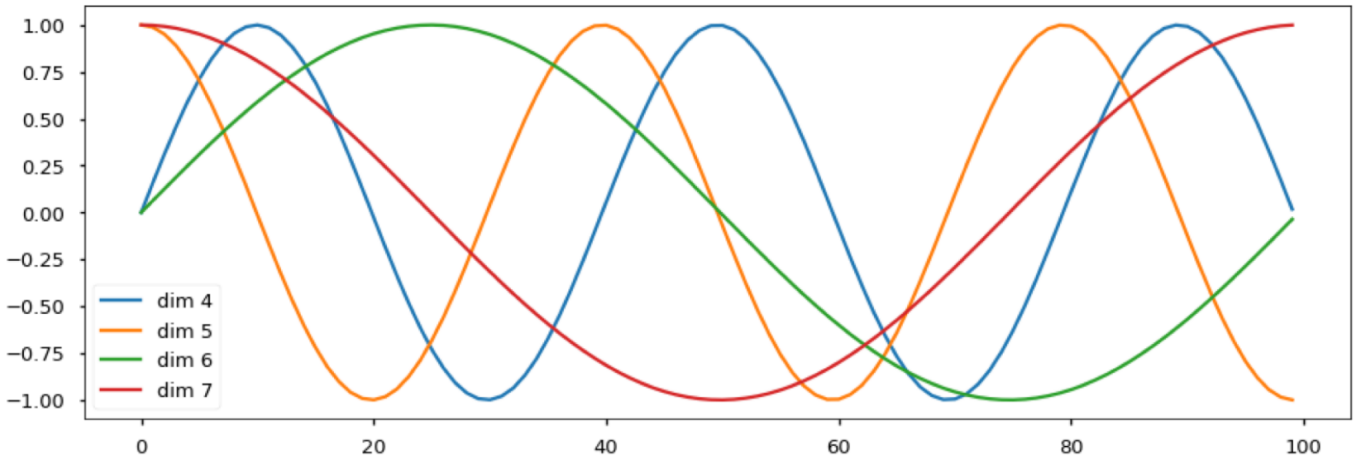
Figure: Example of positional encoding with a embedding size of 4

Computing Positional Encoding

- Positional encoding vectors are generated using a combination of sine and cosine functions, which are concatenated to form a different vector for each position in the sentence.

$$PE_{(pos,i)} = \begin{cases} sin\left(\frac{pos}{10000^{\frac{i}{d}}}\right) & \text{if } i \text{ is even} \\ cos\left(\frac{pos}{10000^{\frac{i}{d}}}\right) & \text{if } i \text{ is odd} \end{cases}$$

- Positional encoding vectors are added to input embeddings, modulating them.



Example of Positional Encoding

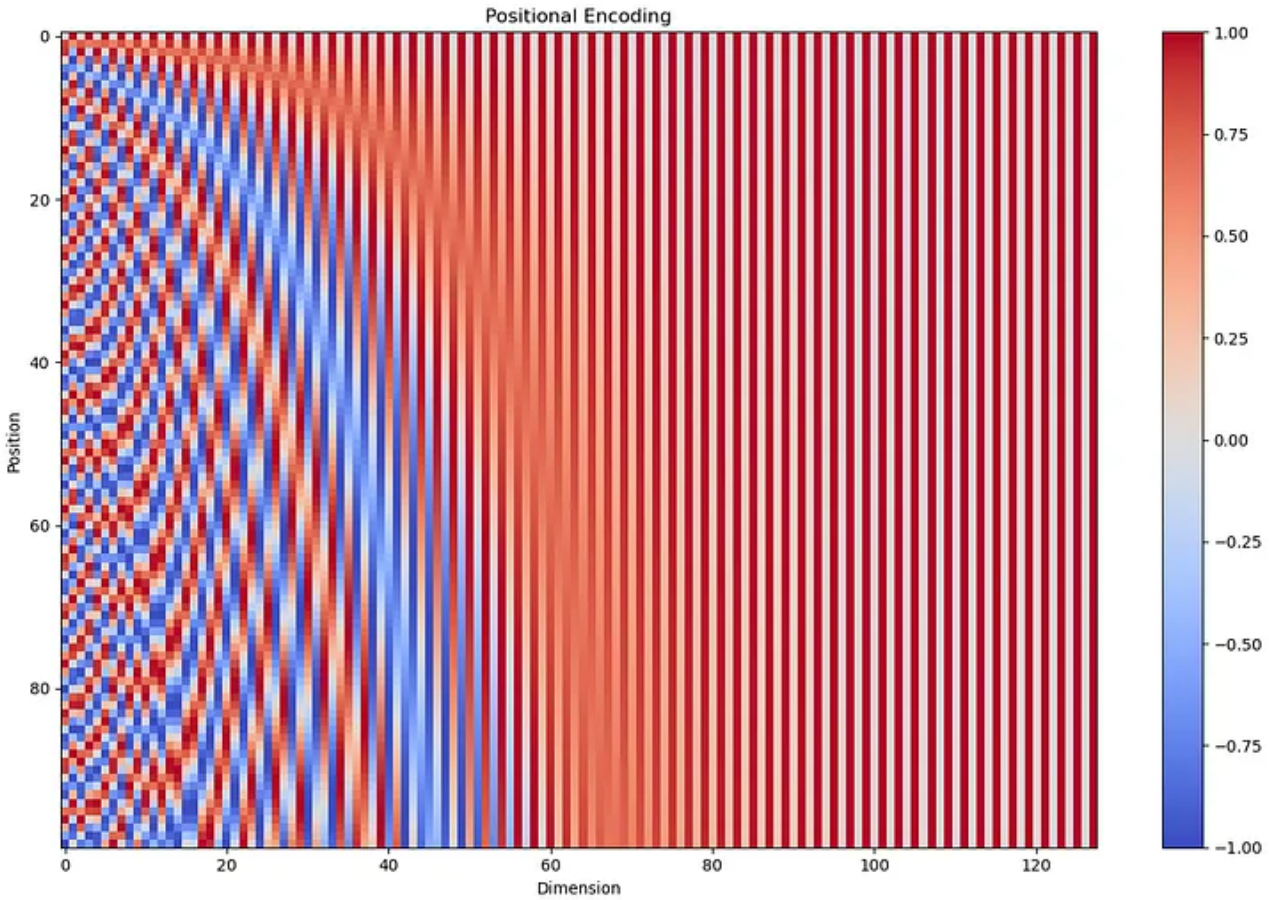


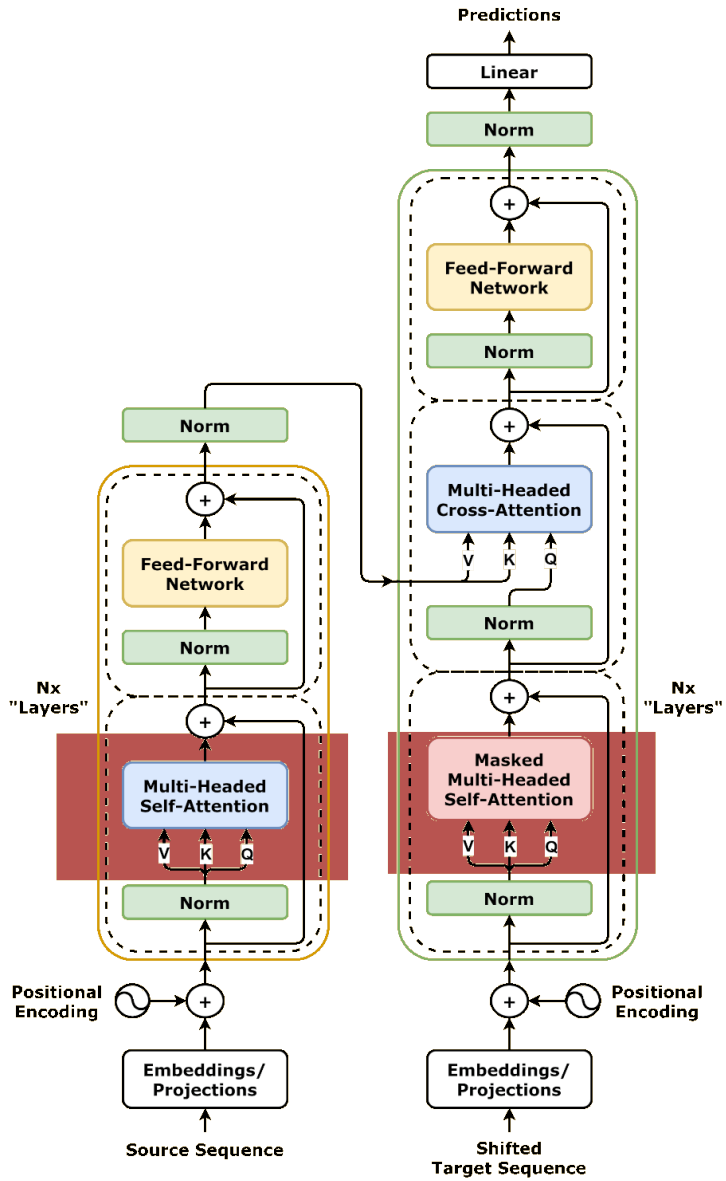
Figure: Example of positional encoding for 100 words with an embedding size of 127.

Positional Encoding Properties

- Leftmost positions of modulated embeddings encode the position. Rightmost positions suffer less or minor changes, preserving the information about the word.
- Due to sin and cos properties, $PE(pos + t) = M_t \cdot PE(pos)$
  (i.e. the embedding for a position $pos + t$ can be computed as a fixed linear transformation M_t of the embedding for position pos)
  This pattern can be learnt by the model so it is aware of the distance between two words.
- Attention layers will be able to compare two words, taking into account both their embeddings and the distance between them.
- This method can be scaled to any sequence length, allowing the model to process a sentences longer than any of those in the training set.

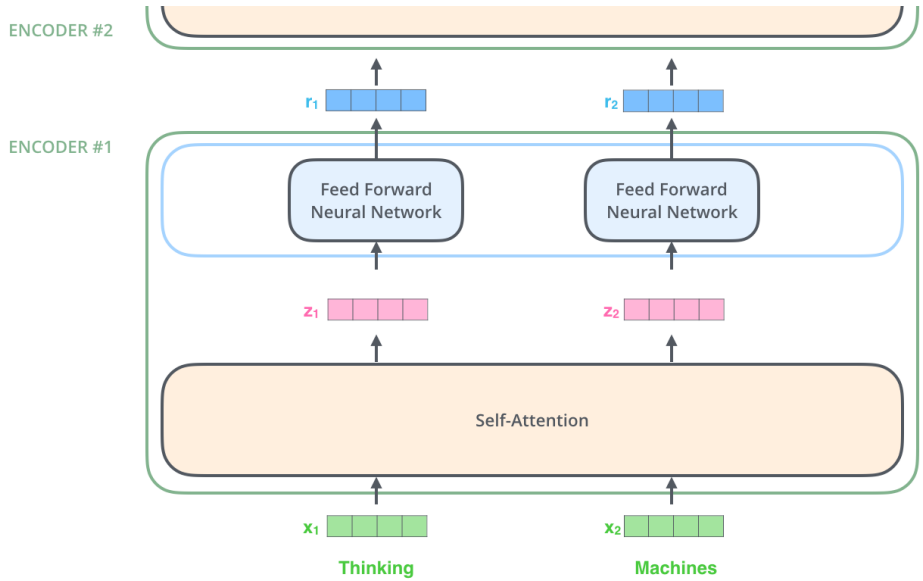
Outline

Self Attention



Self Attention

- 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.



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.
- 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

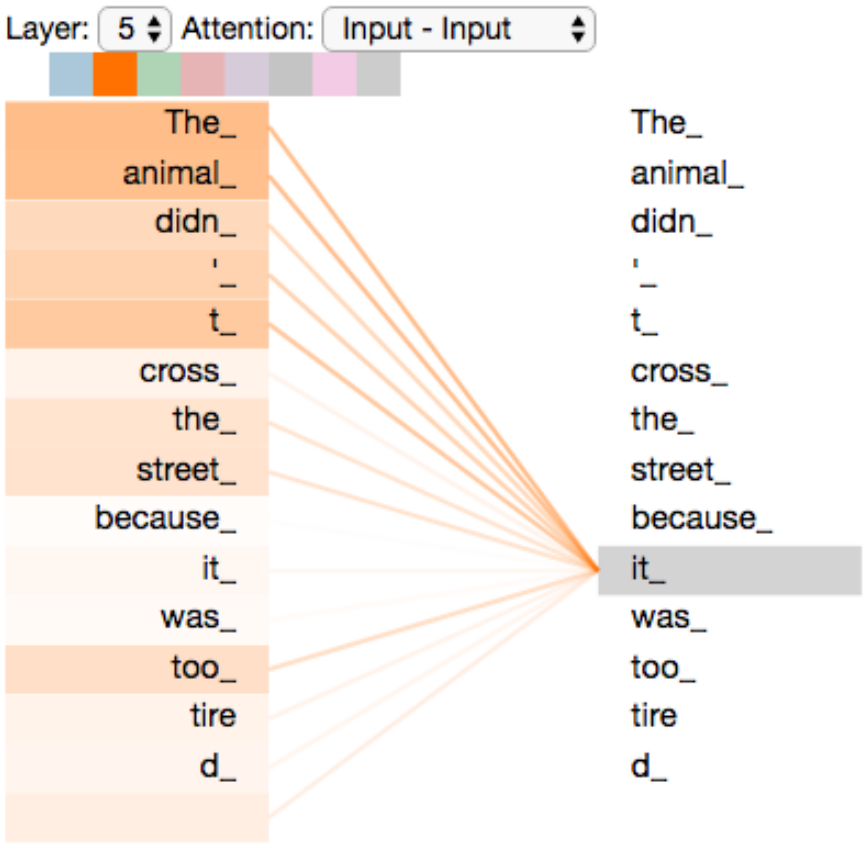
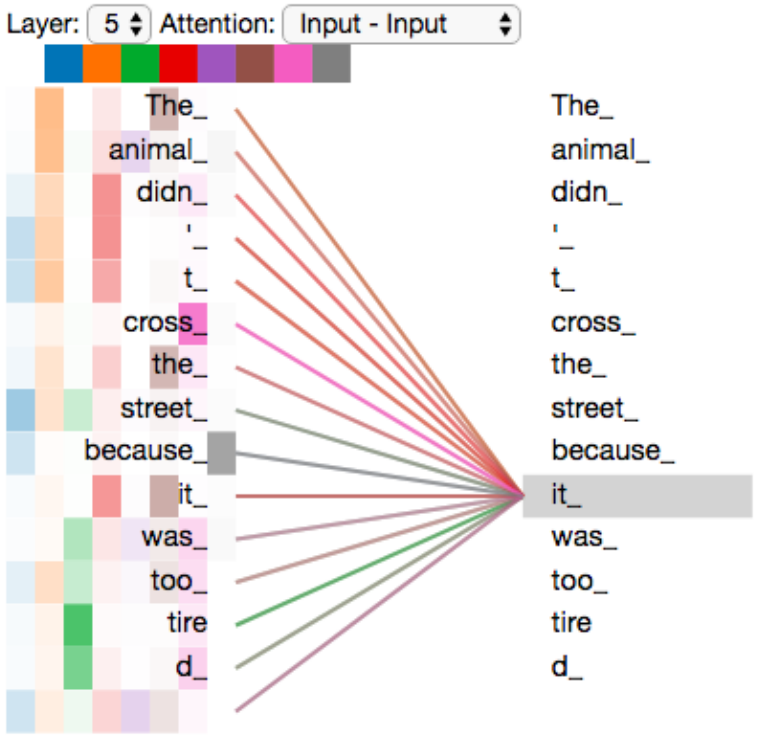


Figure: Example of self-attention in encoder #5 of the Transformer model. Part of the attention mechanism is focusing on *The animal*, and this information is incorporated into the encoding of *it*.

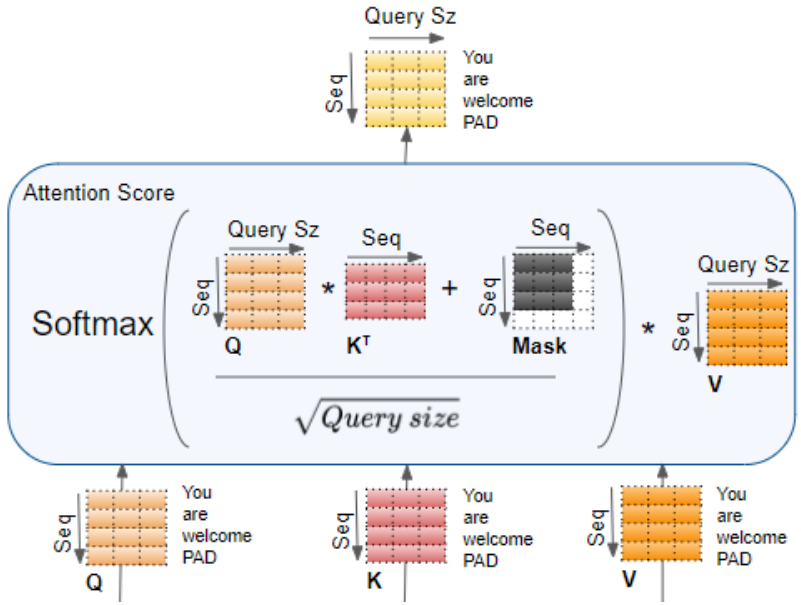
Multi-head Self-Attention

- 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



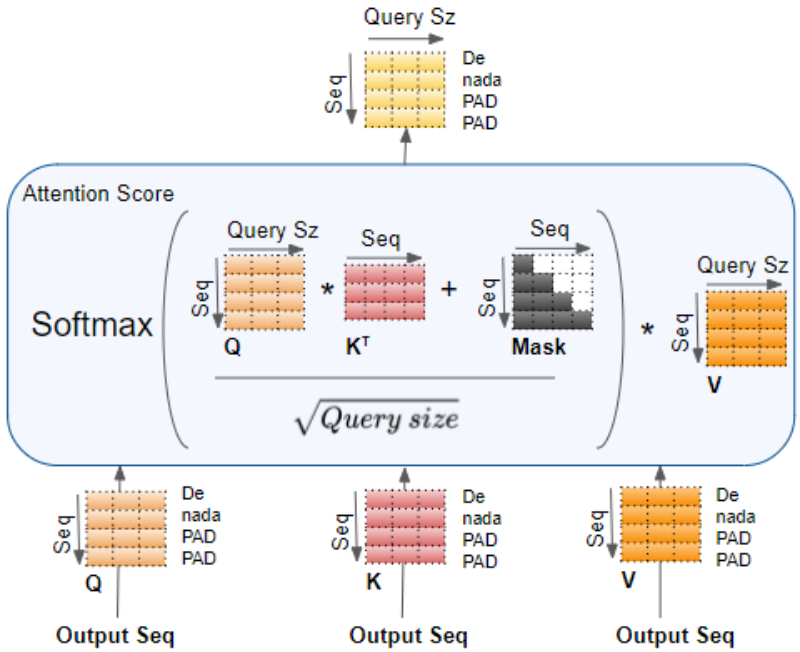
Masked Attention

- As seen above, attention layers compute:
  $\text{Attention}(Q, K, V) = \text{softmax}(QK^T)V$
- A mask can be added to filter out some of the sequence words:
  $\text{Attention}(Q, K, V) = \text{softmax}(QK^T + M)V$
- E.g. we can make the encoder consider only words to the left of the query:



Masked Attention

- As seen above, attention layers compute:
  $\text{Attention}(Q, K, V) = \text{softmax}(QK^T)V$
- A mask can be added to filter out some of the sequence words:
  $\text{Attention}(Q, K, V) = \text{softmax}(QK^T + M)V$
- Or, more importantly, we can prevent the decoder to see future tokens in output sequence during training:



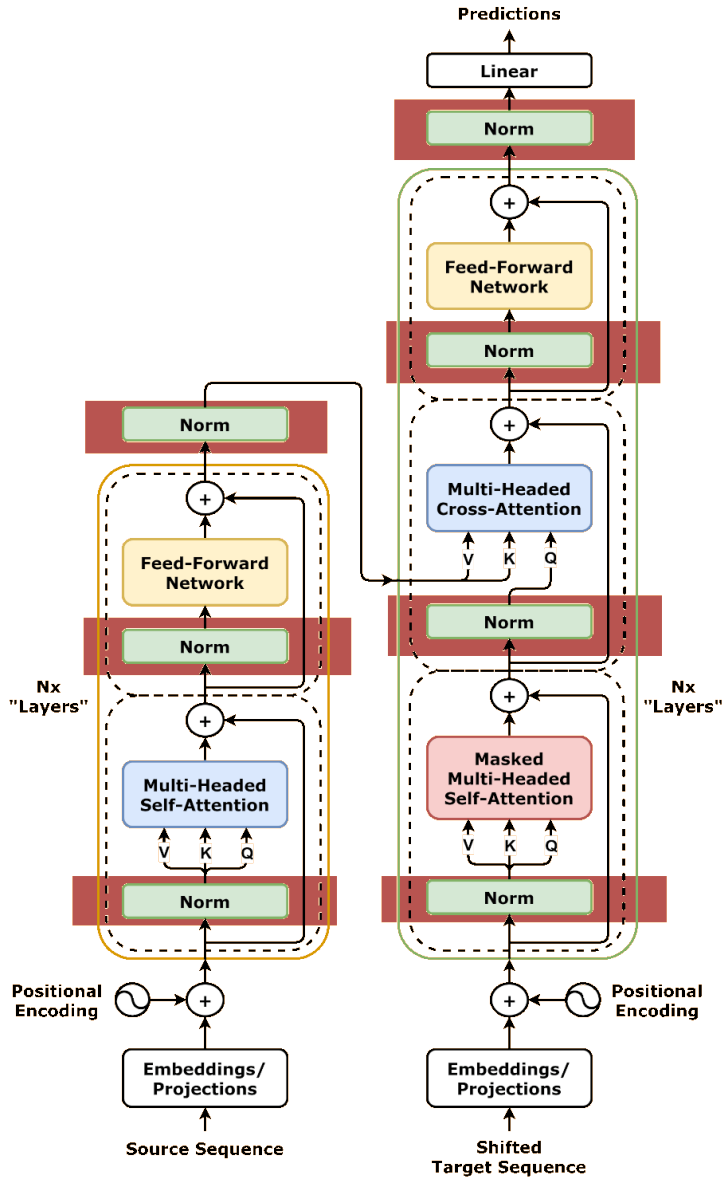
Masked attention

- Masking is achieved by adding a mask matrix to the attention weights before the softmax operation.
- Masking allows to parallelize the training
- Attention scores for future words are set to $-\infty$, adding a mask matrix with the same shape as the attention weights, with $-\infty$ values in positions corresponding to future words:

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

Outline

Layer Normalization



Layer Normalization

Layer normalization helps models train faster by cutting down on uninformative variation in hidden vector values.

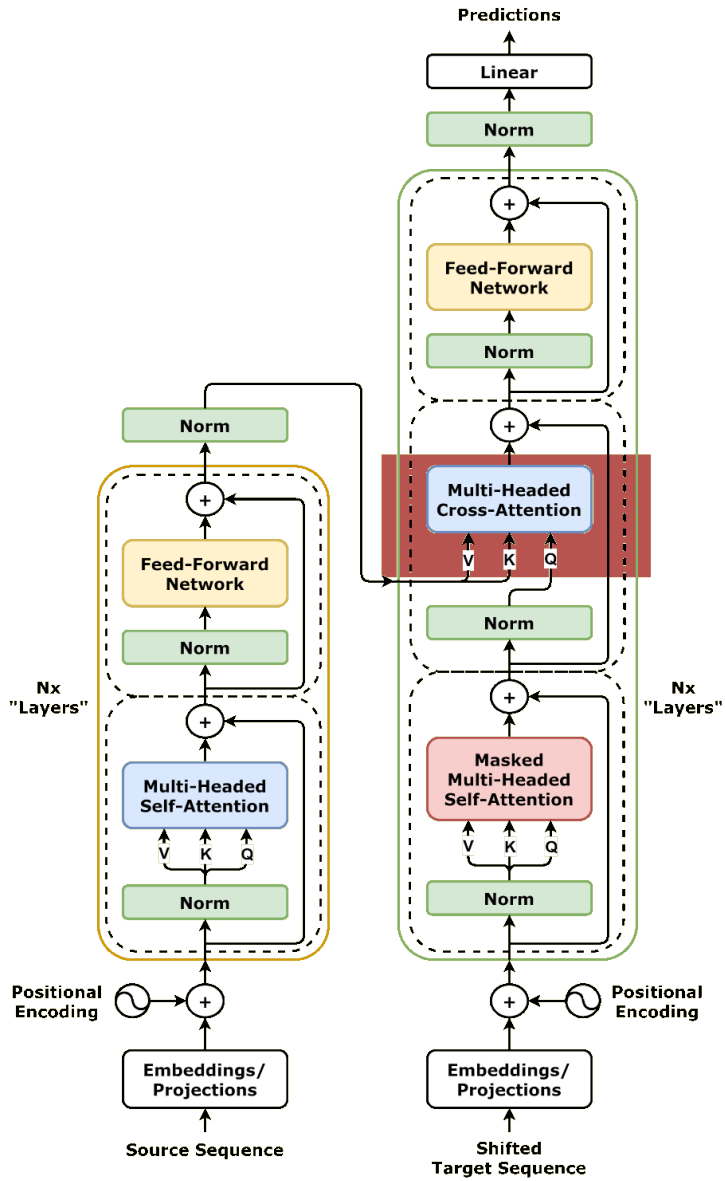
- This is achieved by normalizing the values to have a zero mean and unit standard deviation within each layer.
- It has been shown to improve the performance of a wide range of deep learning models, including transformers.
- This technique can help address issues with internal covariate shift and the vanishing gradient problem.

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

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

Outline

Encoder-Decoder Attention

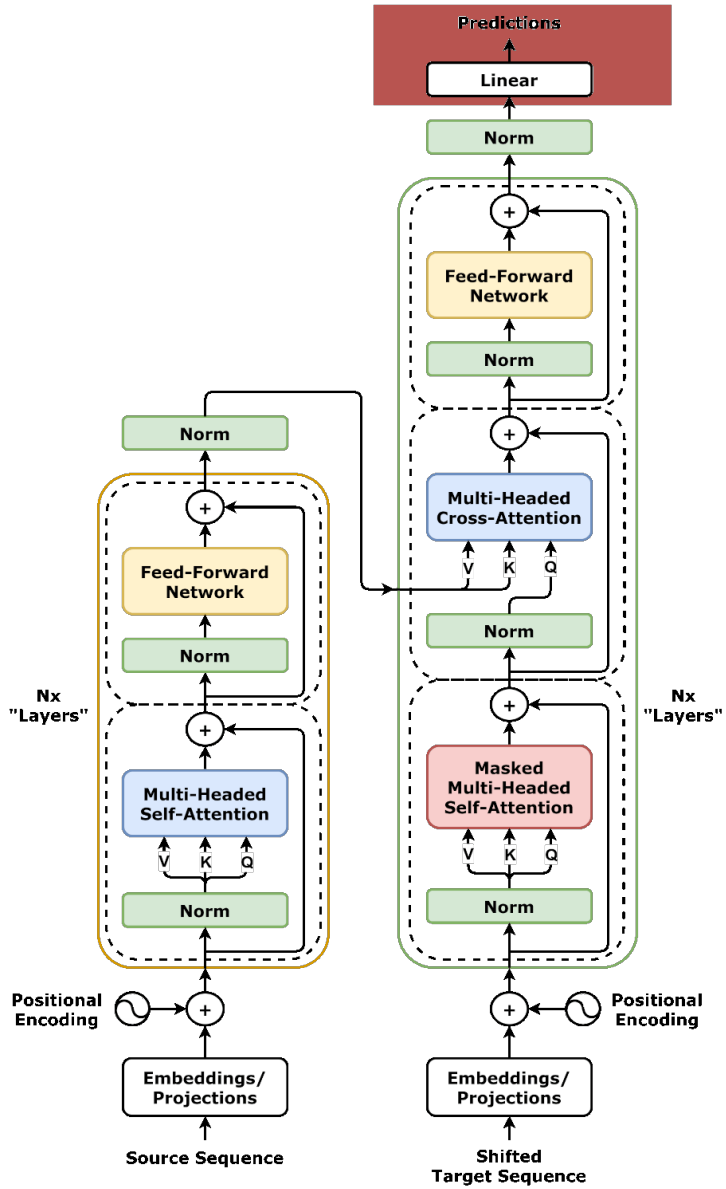


Encoder-Decoder Attention

- The output of the top encoder is feed to the cross attention (aka *encoder-decoder attention*) layer of each decoder.
- The cross attention layer works like multiheaded self-attention, but:
  - Queries matrix is fed with the previous decoder layer output (as in self-attention)
  - Keys and Values matrices are fed with the last encoder output.

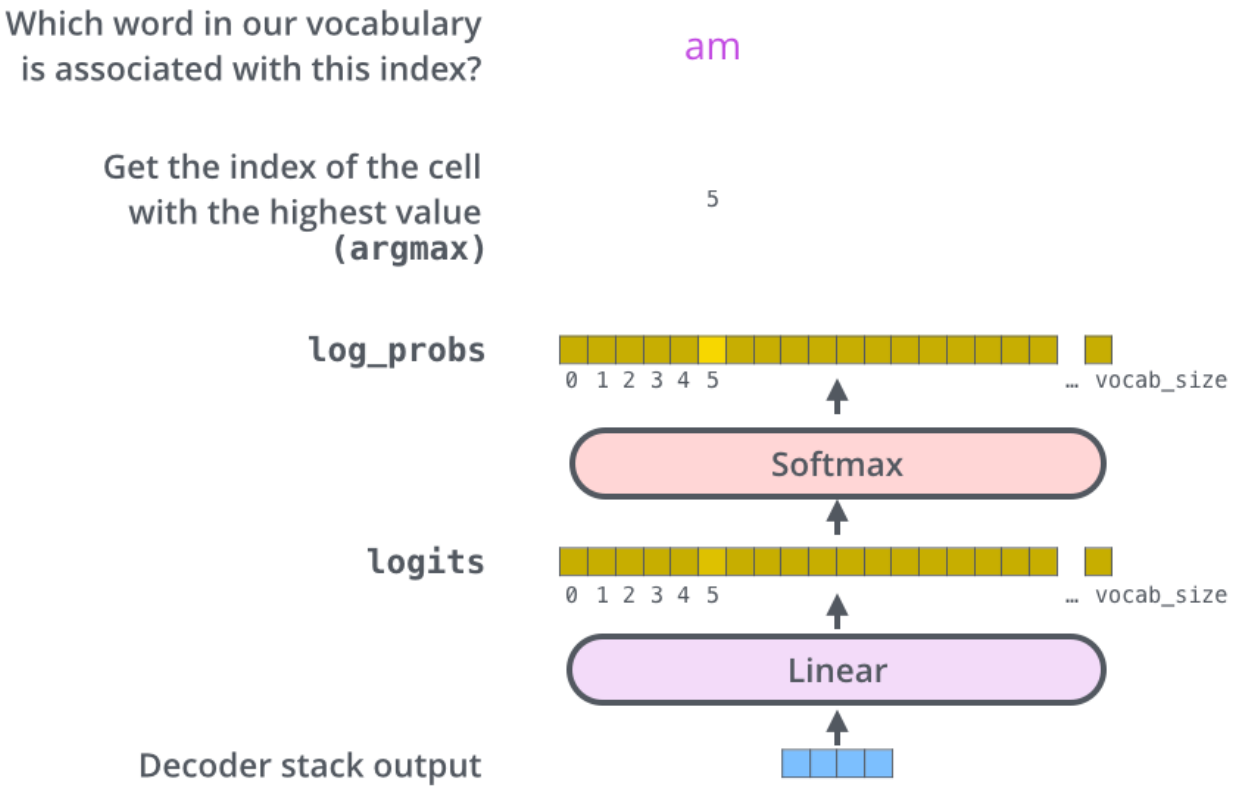
Outline

Final Linear Layer & Softmax



Final Linear Layer & Softmax

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



Outline

Contextual Embeddings

- The same word may have different meanings in different contexts:
  - *Please type everything in lowecase*
  - *What type of food do you like?*
- Classical word embeddings provide a unique representation for each word, regardless of the context. Polysemous words get *mixed* representations.
- Contextual embeddings provide different representations for the same word in different contexts.

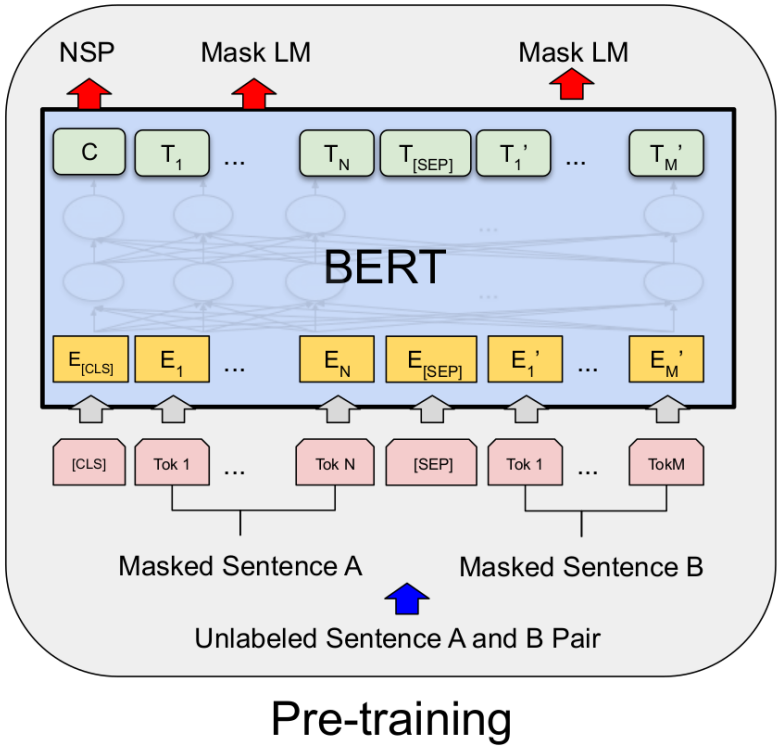
Contextual Embeddings

- Contextual embeddings are trained (similarly to classical embeddings) on a task that requires learning word representations
- Classical word embeddings distribute the hidden layer weights for each word
- Contextual word embeddings distribute the model able to produce the embeddings for each sentence
- This pre-trained model can be:
  - Fine-tuned to perform a different task
  - Used simply to compute embeddings which will be input for another task

BERT

Bidirectional Encoder Representations from Transformers

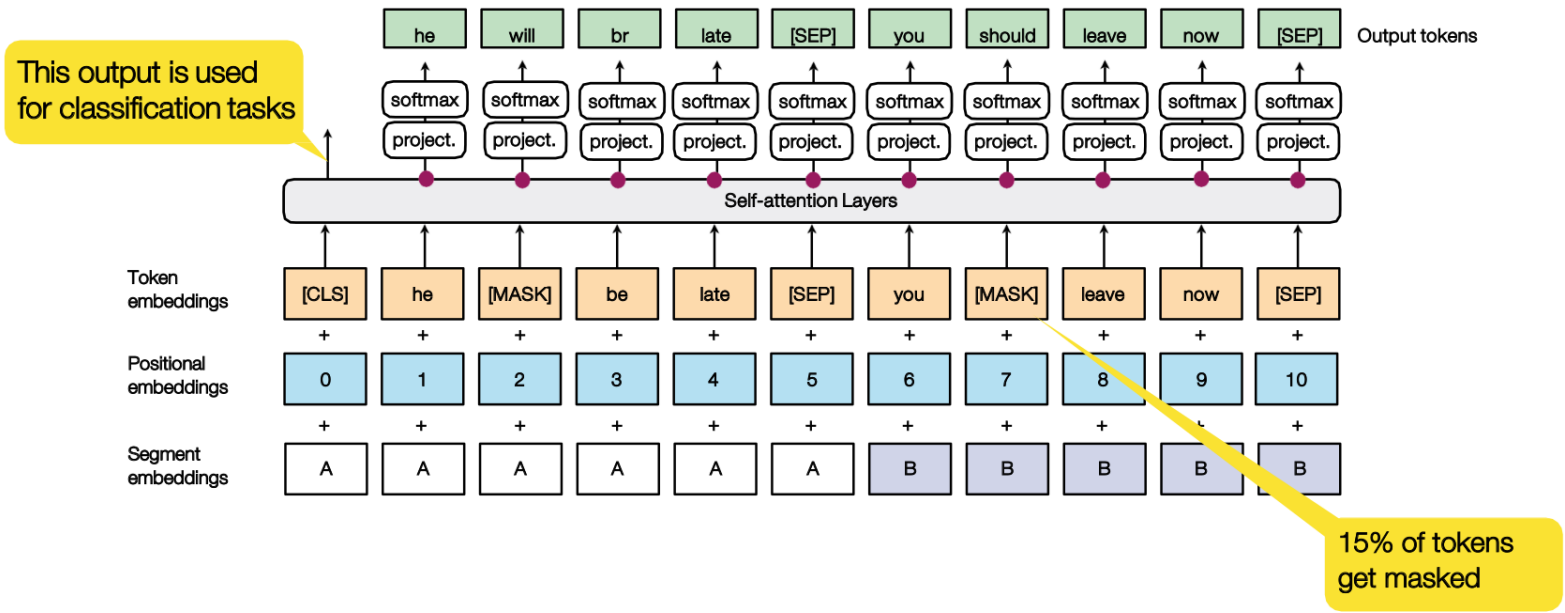
- Transformer architecture
- Encoder **only** (no decoder!)
- Pretrained on two tasks:
  - Masked LM
  - Next Sentence Prediction (NSP)



Pre-training

Images from `https://arxiv.org/abs/1810.04805`

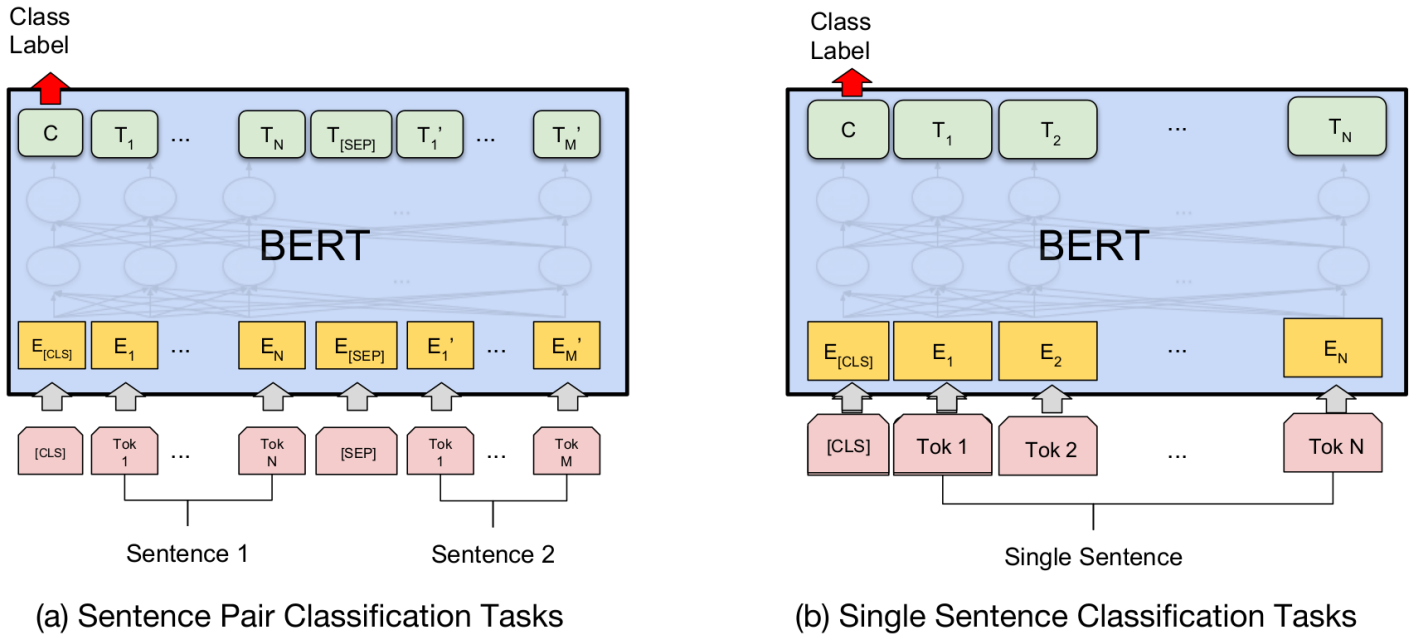
BERT pretraining input



- BERT input is tokenized in *subwords*
- Input consists of Segment encoding + positional encoding + token embeddings
- 15% of the tokens are masked.

BERT fine-tuning

BERT can be directly fine-tuned to perform NLP tasks



(a) Sentence Pair Classification Tasks

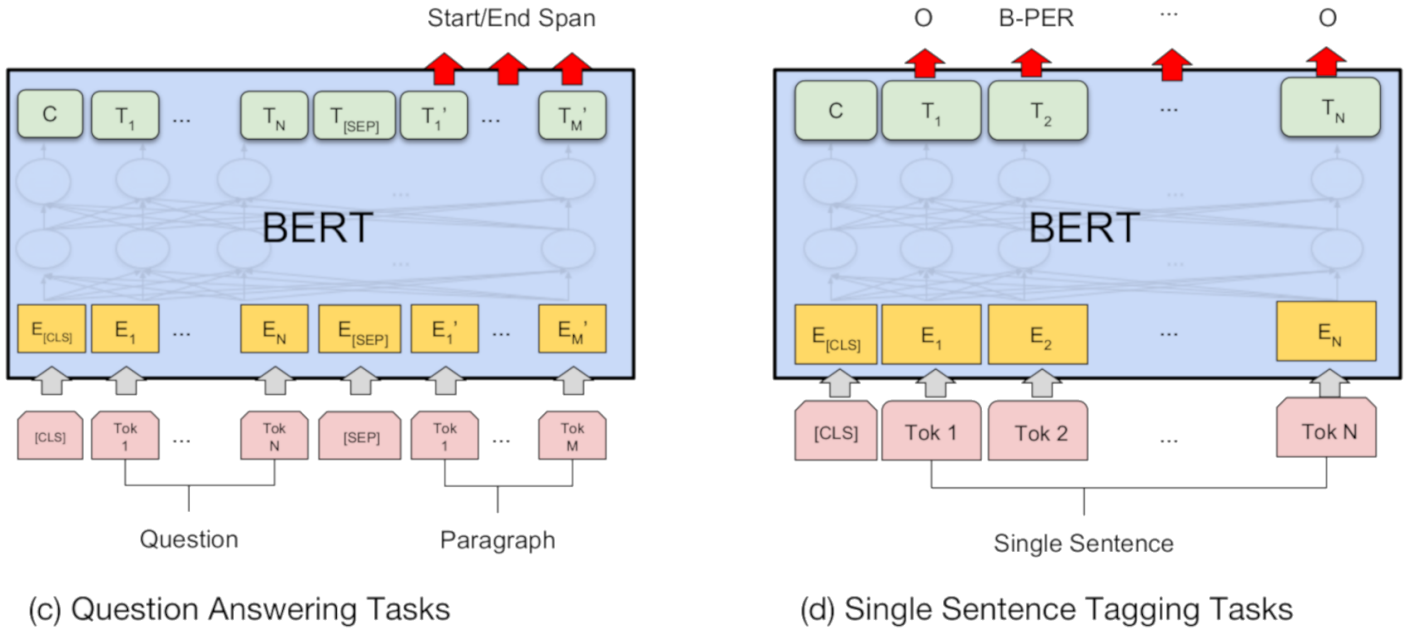
(b) Single Sentence Classification Tasks

a) Sentence pair classification (Textual entailment, similarity, paraphrasing, question-answer)

b) Single sentence classification (Sentiment analysis, grammaticality)

BERT fine-tuning

BERT can be directly fine-tuned to perform NLP tasks



(c) Question Answering Tasks (d) Single Sentence Tagging Tasks

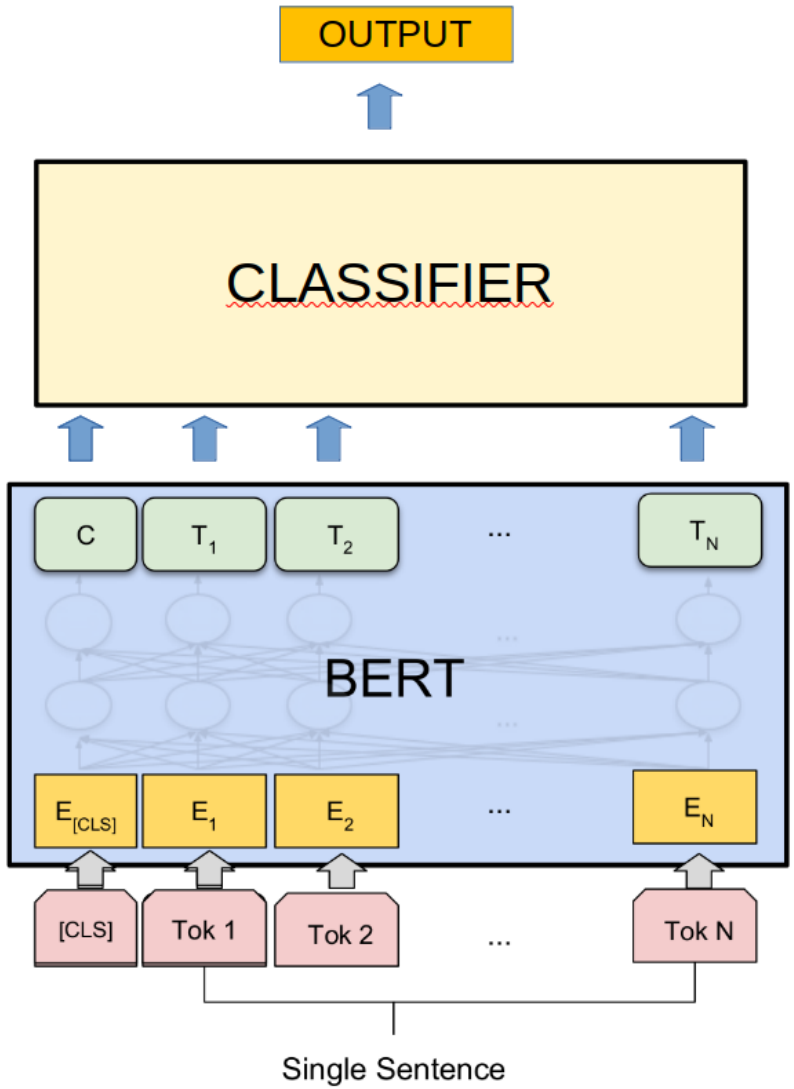
c) Question Answering (spot answer to sentece 1 inside sentece 2)

d) Sequence Tagging (e.g. NERC)

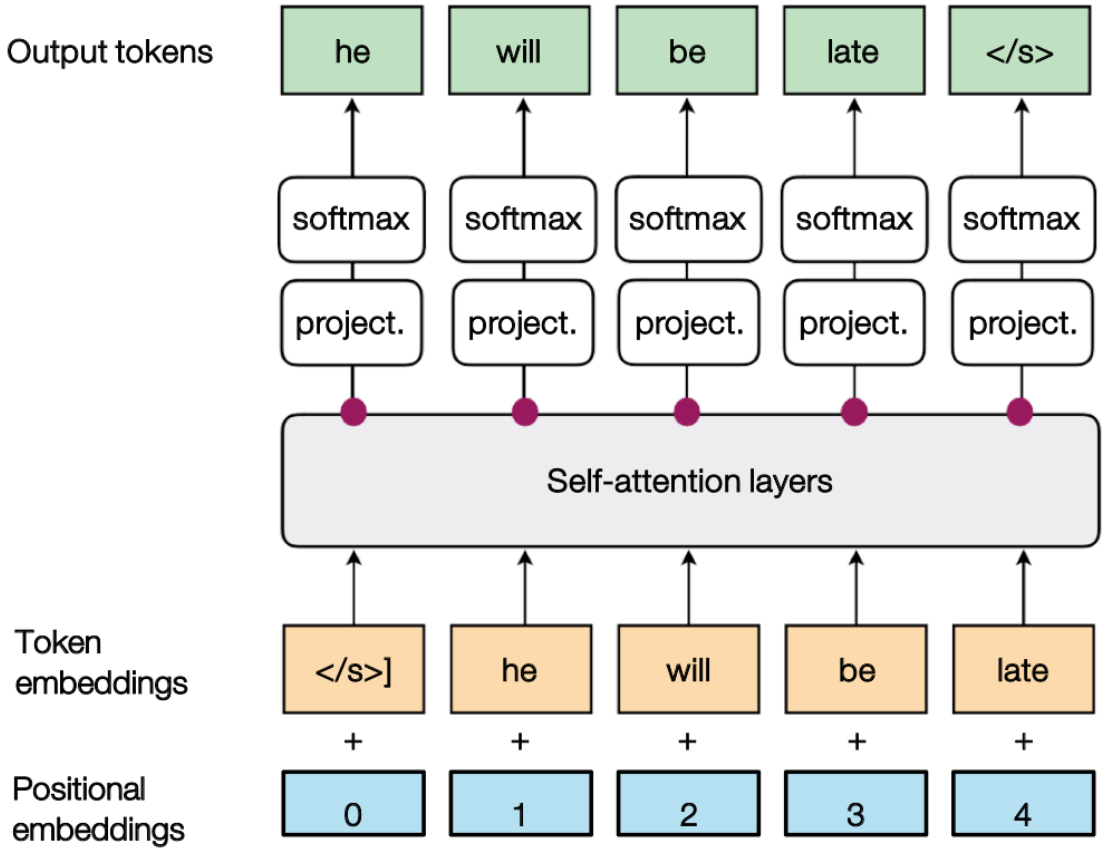
BERT as a preprocessor

BERT is an encoder that can be directly used as a preprocess for other NN architectures

- BERT output is a sequence of (contextual) word embeddings
- It may be fed either as a whole (e.g. to a transformer or CNN) or as a sequence (e.g. to an LSTM)
- BERT layers (all or part) may (or may not) be tuned when training the final classifier.



GPT: Generative Pretrained Transformer



- GPT input is tokenized in *subwords*
- Input consists of Segment encoding + positional encoding + token embeddings

GPT: Generative Pretrained Transformer

- Model is trained on unsupervised language modeling

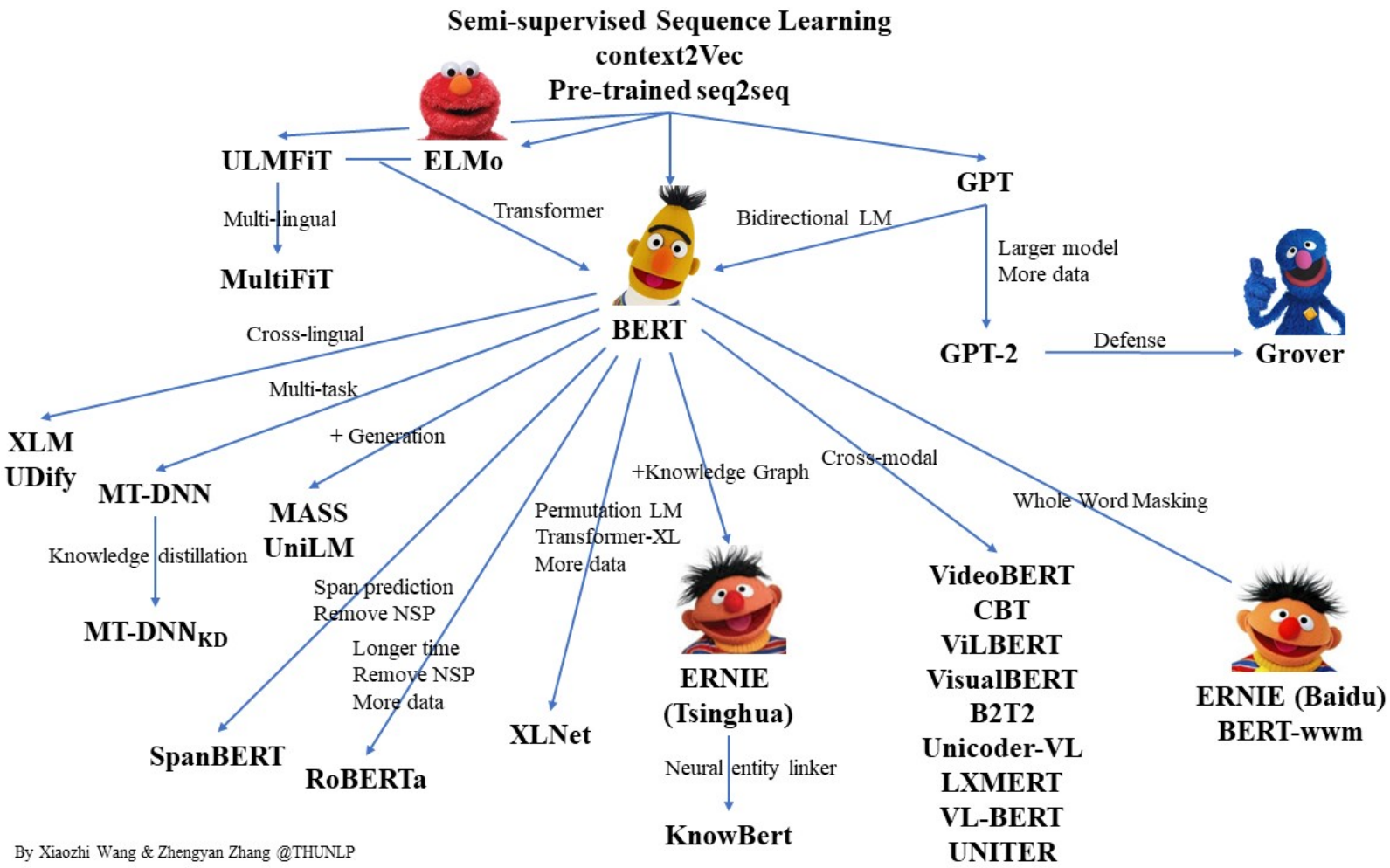
$$L_1(T) = \sum_i \log P(t_i | t_{i-k} \dots ti - 1; \theta)$$

- plus fine tuning on higher-level tasts (textual entailment, semantic similarity, question answering, etc.)

$$L_2(C) = \sum_{x,y} \log P(y | x_1 \dots x_n)$$

$$L_3(C) = L_2(C) + \lambda L_1(C)$$

Other related models



Acknowledgements

- Some slides in this session are based on images and ideas from lectures by
  - Marta Ruiz Costa-Jussà
  - José Adrián Rodríguez Fonollosa
  - Salvador Medina