Sequence Modelling

Mingfei Sun

Foundations of Machine Learning
The University of Manchester

A Quick Recap

In the previous lecture, we've covered:

- Multi-Layer Perceptron (MLP)
- Feedforward neural networks:

  Forward equations, Backpropagation equations
- Training of neural networks:

  Parameter initialization, Normalization, SGD optimization, and Regularization.

Outline

Intended Learning Outcomes

By the end of this lecture, you'll be able to

- Identify the key components in language models, Recurrent Neural Networks, and Transformers
- Explain how the Back-Propagation Through Time (BPTT) and Truncated BPTT can be used to optimize Recurrent Neural Networks
- Explain how the Attention is used to transform sequences in an equivariant or non-equivariant way
- Apply and implement self-attention to process sequential inputs

Outline

Language Models

A language model assigns a probability to a sequence of words, such that $\sum_{x \in \Sigma^*} p(x) = 1$

- Given the observed training text, how probable is this new utterance?
- Can compare different orderings of words (e.g. Translation):

$$p(\text{he likes apples}) > p(\text{apples likes he})$$

or choice of words (e.g., Speech Recognition)

$$p(\text{he likes apples}) > p(\text{he licks apples})$$

- Employ the chain rule to decompose the joint probability into a sequence of conditional probabilities:

$$p(x_1, x_2, x_3, ..., x_N) = p(x_1)p(x_2|x_1)p(x_3|x_1, x_2) \cdot \cdots \cdot p(x_N|x_1, x_2, ..., x_{N-1})$$

- Can model complex joint distributions by learning conditional distributions over the next word (x_n) given the history of words observed $(x_1, x_2, ..., x_{n-1})$

Language Models

The simple objective of modelling the next word given the observed history contains much of the complexity of natural language understanding.

- Consider predicting the extension of the utterance:

$$p(\cdot|\text{There she built a })$$

- With more context we are able to use our knowledge of both language and the world to heavily constrain the distribution over the next word:

$$p(\cdot|\text{Alice went to the beach. There she built a })$$

  There is evidence that human language acquisition partly relies on future prediction

- Language modelling is a time series prediction problem in which we must be careful to train on the past and test on the future

Outline

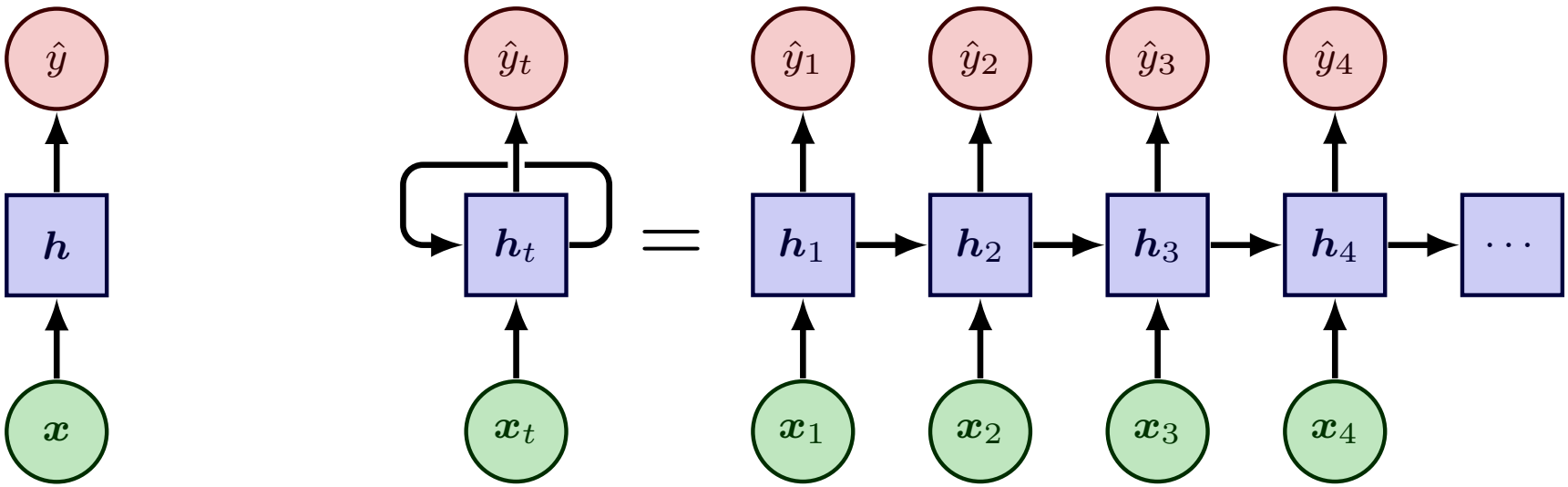
Recurrent Neural Networks (RNNs)

Feedforward

$$\boldsymbol{h} = g(\boldsymbol{V}\boldsymbol{x} + \boldsymbol{c})$$
$$\hat{y} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b}$$

Recurrent neural networks

$$\boldsymbol{h}_n = g(\boldsymbol{V}[\boldsymbol{x}_n; \boldsymbol{h}_{n-1}] + \boldsymbol{c})$$
$$\hat{y}_n = \boldsymbol{W}\boldsymbol{h}_n + \boldsymbol{b}$$



The unrolled recurrent network is a directed acyclic graph.

RNNs: Unrolled step-by-step

Consider an example of Next Word Prediction:

$$p(\cdot|\text{There she built a }\ )$$

How to build an RNN model for this?

1. Prepend the sentence with a Start symbol: $\langle s \rangle$
2. Formulate the input sequence as: ($\langle s \rangle$, There, she, built, a)

RNNs: Unrolled step-by-step

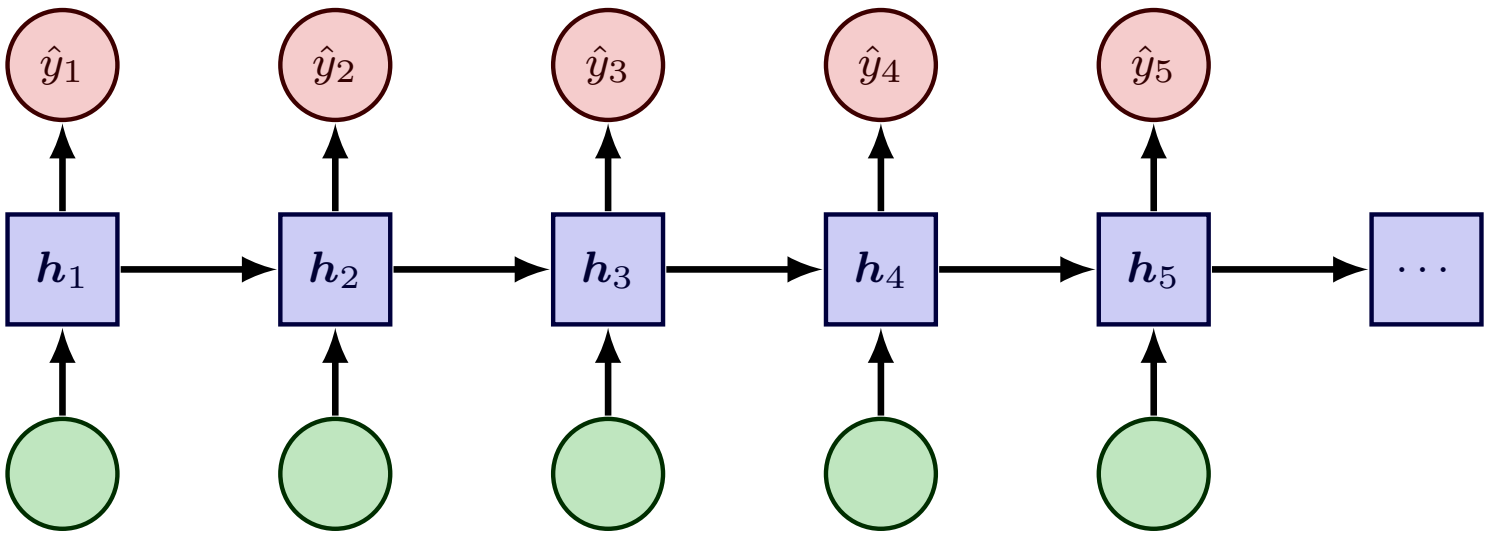
Consider an example of Next Word Prediction:

$$p(\cdot|\text{There she built a }\ )$$

How to build an RNN model for this?

1. Prepend the sentence with a Start symbol: $\langle s \rangle$
2. Formulate the input sequence as: ($\langle s \rangle$, There, she, built, a)

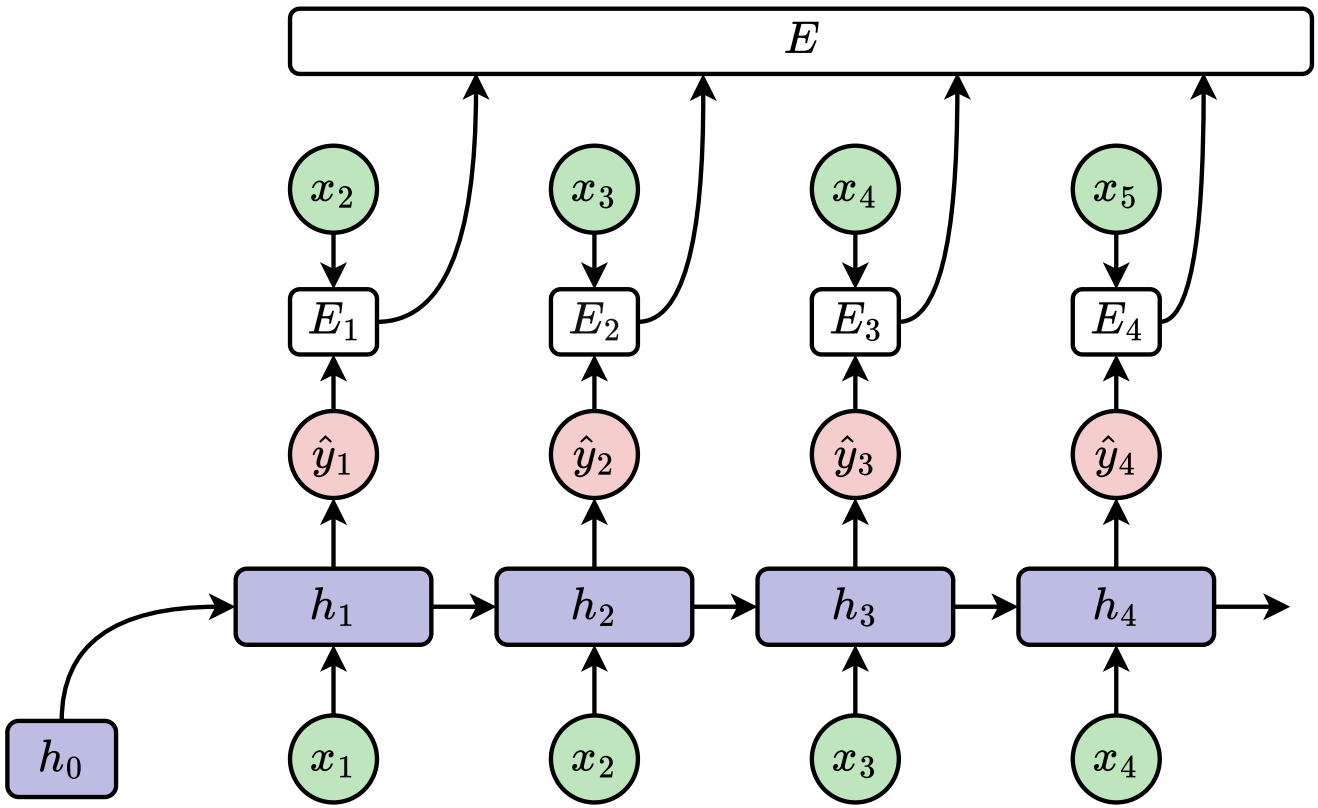
$$\boldsymbol{h}_n = g(\boldsymbol{V}[\boldsymbol{x}_n; \boldsymbol{h}_{n-1}] + \boldsymbol{c})$$



RNNs: Forward pass

We can run forward equations (black edges) as usual:

$$E = \frac{1}{N} \sum_{n=1} E_n(x_n, \hat{y}_n)$$

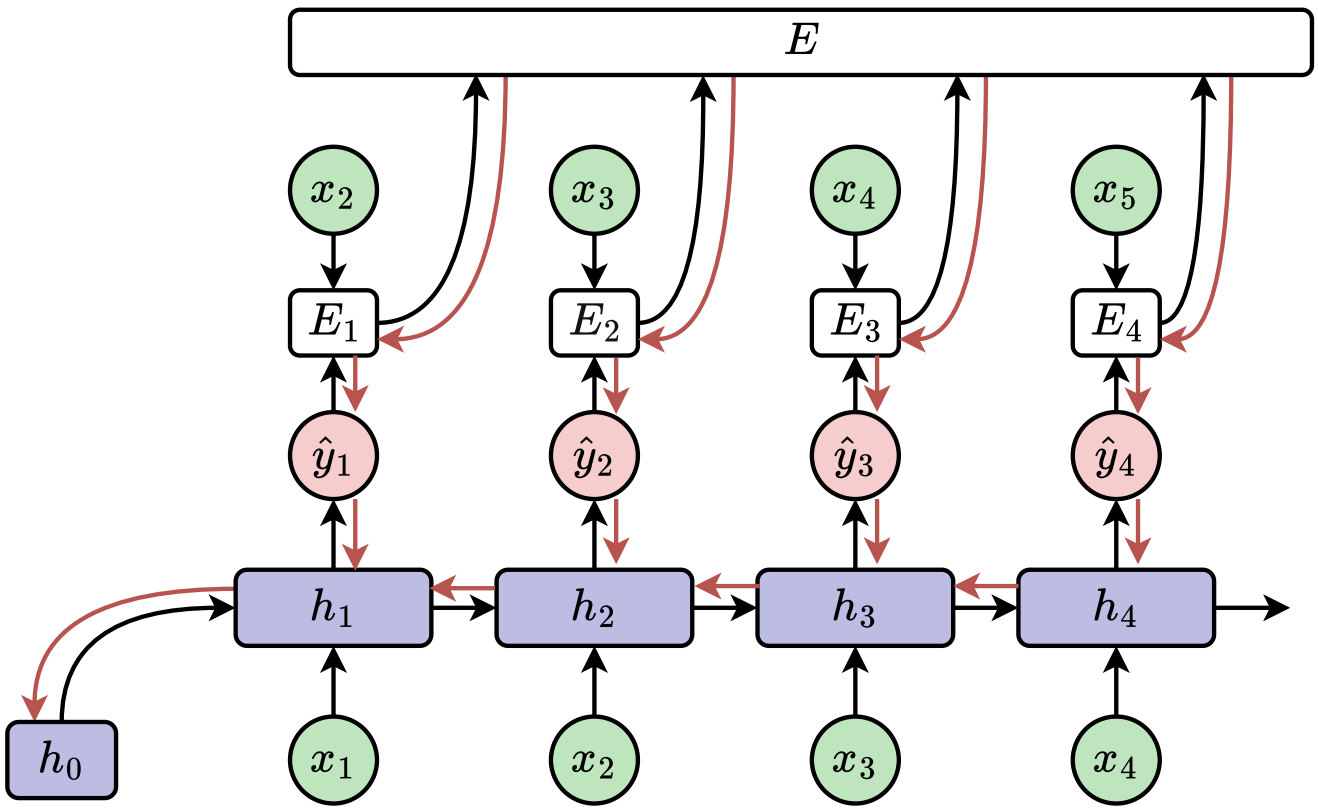


RNNs: Back-Propagation Through Time (1)

We can run forward equations as usual:

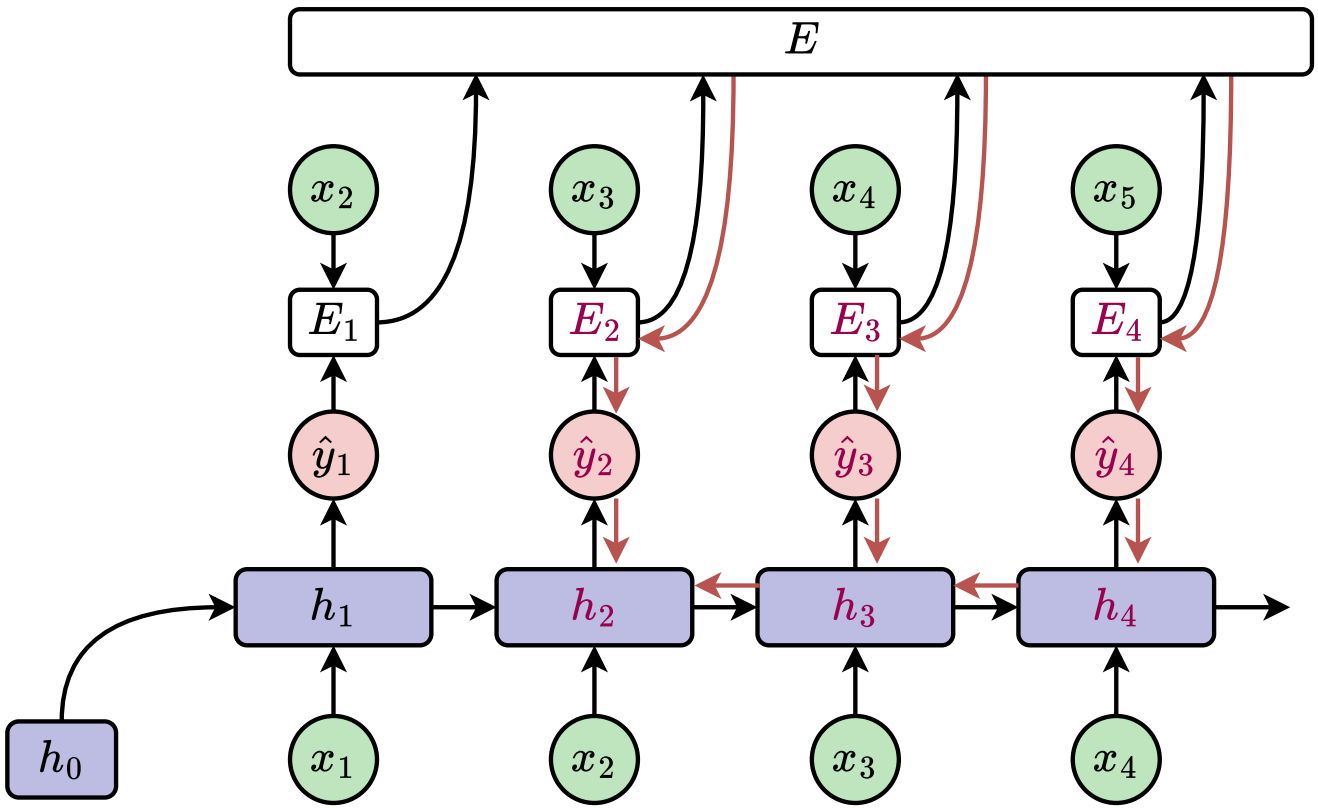
$$E = \frac{1}{N} \sum_{n=1} E_n(x_n, \hat{y}_n)$$

Back-propagation (red edges) gives gradients for parameter update:



RNNs: Back-Propagation Through Time (2)

$$\frac{\partial E}{\partial h_2} = \frac{\partial E}{\partial E_2}\frac{\partial E_2}{\partial \hat{y}_2}\frac{\partial \hat{y}_2}{\partial \boldsymbol{h}_2} + \frac{\partial E}{\partial E_3}\frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial \boldsymbol{h}_3}\frac{\partial \boldsymbol{h}_3}{\partial \boldsymbol{h}_2} + \frac{\partial E}{\partial E_4}\frac{\partial E_4}{\partial \hat{y}_4}\frac{\partial \hat{y}_4}{\partial \boldsymbol{h}_4}\frac{\partial \boldsymbol{h}_4}{\partial \boldsymbol{h}_3}\frac{\partial \boldsymbol{h}_3}{\partial \boldsymbol{h}_2} + \dots$$

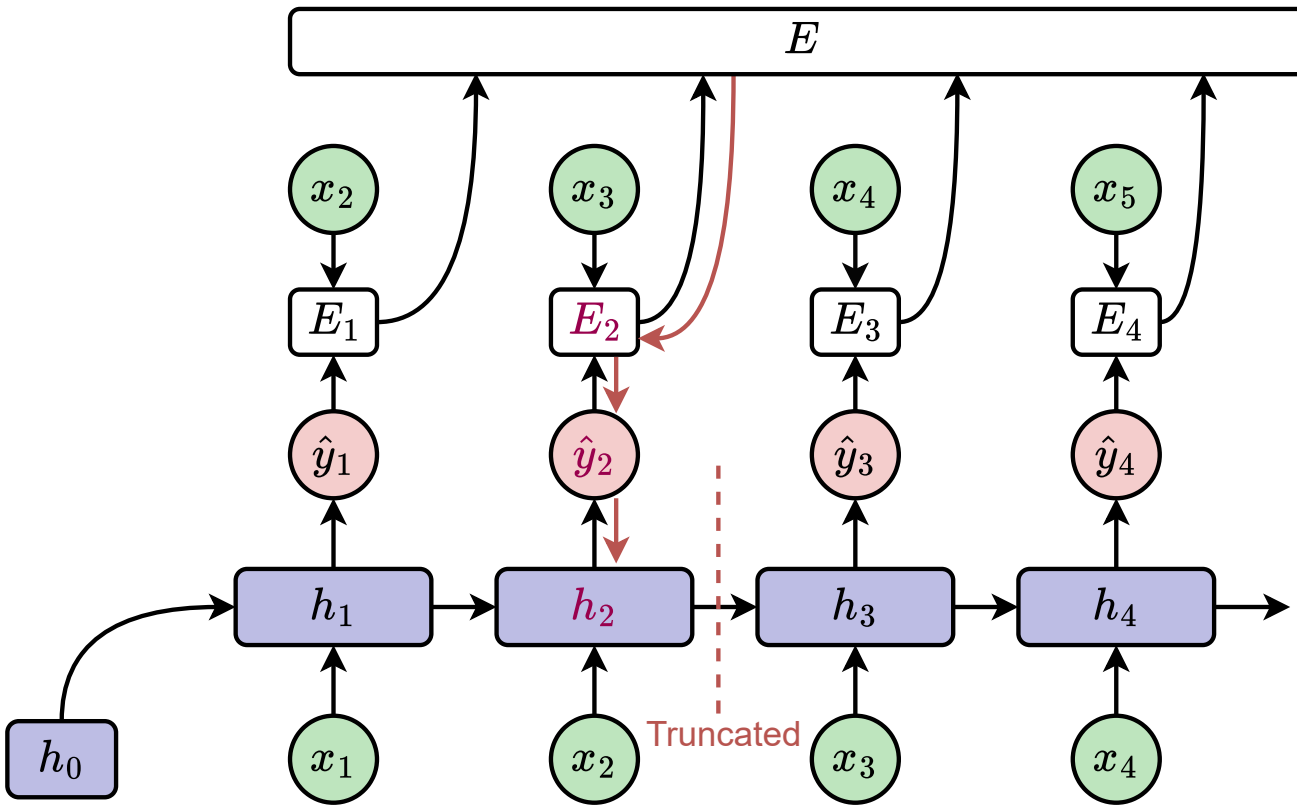


This gradient update is called **Back-Propagation Through Time (BPTT)**: the derivatives at timestep n depend on those at future timesteps:

RNNs: Back-Propagation Through Time (3)

If we break dependencies after a fixed number of timesteps we get **Truncated BPTT**:

$$\frac{\partial E}{\partial \boldsymbol{h}_2} \approx \frac{\partial E}{\partial E_2} \frac{\partial E_2}{\partial \hat{y}_2} \frac{\partial \hat{y}_2}{\partial \boldsymbol{h}_2}$$



RNNs: Exploding and Vanishing Gradients (1)

An RNN Model needs to discover and represent long-range dependencies:

$$p(\text{sandcastle}|\text{Alice went to the beach. There she built a }\_)$$

While an RNN model can represent such dependencies in theory, can it learn them?

RNNs: Exploding and Vanishing Gradients (1)

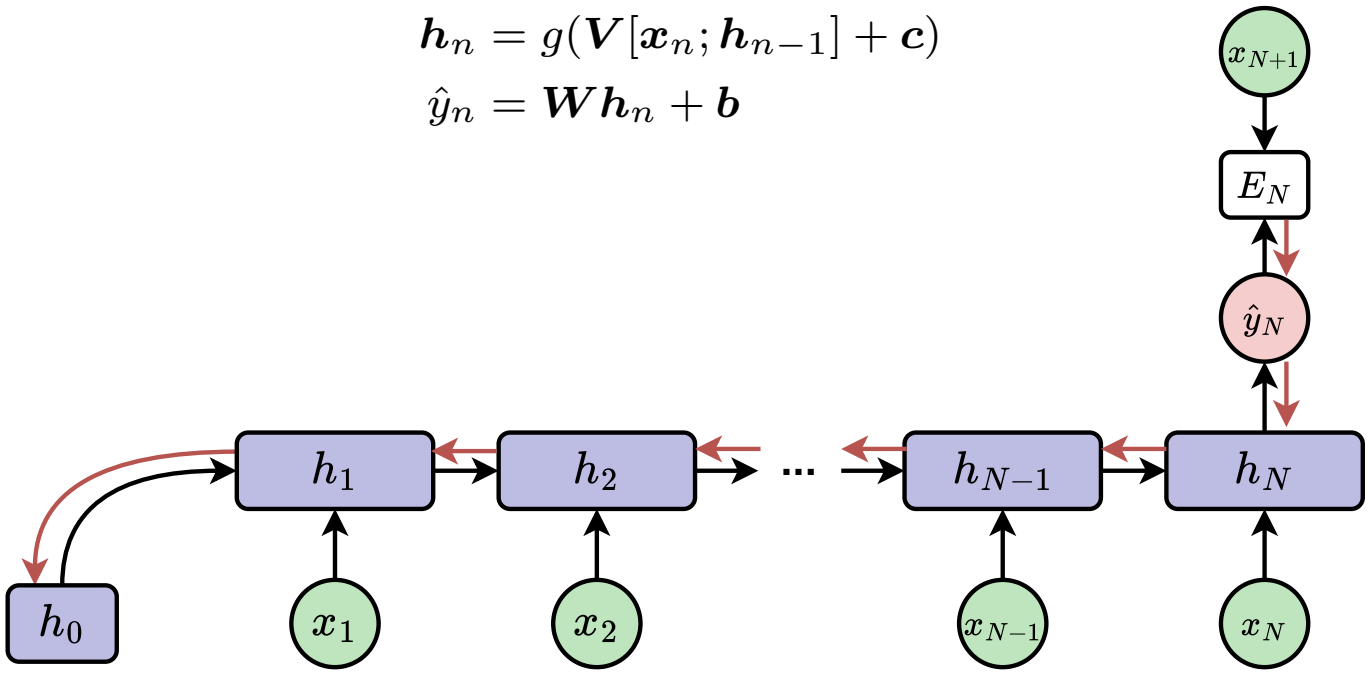
An RNN Model needs to discover and represent long-range dependencies:

$$p(\text{sandcastle}|\text{Alice went to the beach. There she built a }\ )$$

While an RNN model can represent such dependencies in theory, can it learn them?

- Consider the path of partial derivatives linking a change in E_N to changes in $\boldsymbol{h}_1$:

$$\boldsymbol{h}_n = g(\boldsymbol{V}[\boldsymbol{x}_n; \boldsymbol{h}_{n-1}] + \boldsymbol{c})$$
$$\hat{y}_n = \boldsymbol{W}\boldsymbol{h}_n + \boldsymbol{b}$$



$$\frac{\partial E_N}{\partial \boldsymbol{h}_1} = \frac{\partial E_N}{\partial \hat{y}_N}\frac{\partial \hat{y}_N}{\partial \boldsymbol{h}_N}\left(\prod_{n\in\{N,\ldots,2\}} \frac{\partial \boldsymbol{h}_n}{\partial \boldsymbol{h}_{n-1}}\right)$$

RNNs: Exploding and Vanishing Gradients (2)

Assume $\boldsymbol{V}$ decomposes into $\boldsymbol{V}_x$ and $\boldsymbol{V}_h$ and g is an element-wise activation:

$$\boldsymbol{h}_n = g(\boldsymbol{V}[\boldsymbol{x}_n; \boldsymbol{h}_{n-1}] + \boldsymbol{c}) = g(\underbrace{\boldsymbol{V}_x \boldsymbol{x}_n + \boldsymbol{V}_h \boldsymbol{h}_{n-1} + \boldsymbol{c}}_{\boldsymbol{z}_n})$$

Calculate the gradients: $\frac{\partial \boldsymbol{h}_n}{\partial \boldsymbol{z}_n} = \text{diag}\left(g'(\boldsymbol{z}_n)\right), \quad \frac{\partial \boldsymbol{z}_n}{\partial \boldsymbol{h}_{n-1}} = \boldsymbol{V}_h.$

$$\begin{aligned}
\frac{\partial E_N}{\partial \boldsymbol{h}_1} &= \frac{\partial E_N}{\partial \hat{y}_N} \frac{\partial \hat{y}_N}{\partial \boldsymbol{h}_N} \left(\prod_{n \in \{N, \ldots, 2\}} \frac{\partial \boldsymbol{h}_n}{\partial \boldsymbol{z}_n} \frac{\partial \boldsymbol{z}_n}{\partial \boldsymbol{h}_{n-1}} \right) \\
&= \frac{\partial E_N}{\partial \hat{y}_N} \frac{\partial \hat{y}_N}{\partial \boldsymbol{h}_N} \left(\prod_{n \in \{N, \ldots, 2\}} \text{diag}\left(g'(\boldsymbol{z}_n)\right) \boldsymbol{V}_h \right)
\end{aligned}$$

For the repeated multiplication of $\boldsymbol{V}_h$, if the largest eigenvalue of $\boldsymbol{V}_h$ is

- 1, then gradient can be propagated,
- > 1, the product will grow exponentially (explode),
- < 1, the product shrinks exponentially (vanishes).

The most popular solution: to change the network architecture to include gated units[1]

[1] Christopher Olah: Understanding LSTM Networks (https://colah.github.io/posts/2015-08-Understanding-LSTMs/)

Outline

Issues with recurrent neural networks

Lack of parallelizability: Forward and backward passes have $\mathcal{O}(\text{sequence length})$ unparallelizable operations

- GPUs can perform a bunch of independent computations at once!
- But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
- Inhibits training on very large datasets!

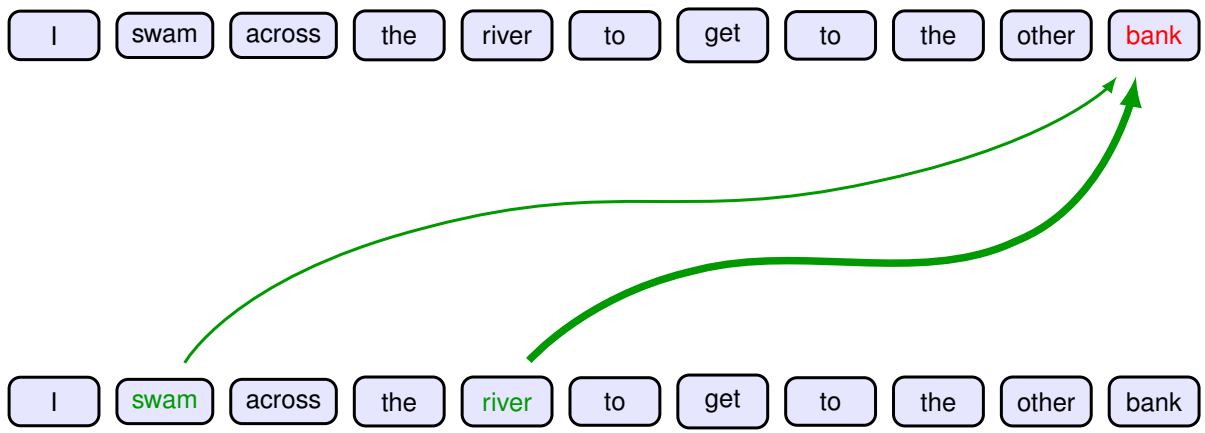
If not recurrence, then what? How about **attention**?

Attention

Consider the following two sentences:

- I swam across the river to get to the other bank.
- I walked across the road to get cash from the bank.

Task: to determine the appropriate interpretation of 'bank'



Attention: a neural network should *attend* to, in other words rely more heavily on, specific words from the rest of the sequence.

Input Representation: word embedding

How to convert the words into a numerical representation?

- **One-hot encodding**:
  - Define a fixed dictionary and introduce vectors of length equal to dictionary size
  - To encode the k-th word with a vector $\boldsymbol{x}_n$ having a 1 in position k and 0 elsewhere
  - Results in vectors of very high dimensionality if the dictionary is large
- **Word embedding**: to map words into a lower-dimensional space
  - Defined a matrix $\boldsymbol{E}$ of size $D \times K$: D the dimensionality of the embedding space, K the dimensionality of the dictionary

    $$\boldsymbol{v}_n = \boldsymbol{E}\boldsymbol{x}_n$$

    Vector $\boldsymbol{v}_n$ is the corresponding column of the matrix $\boldsymbol{E}$ (can be learned, e.g., *word2vec*[2])
  - Learned embedding space often has an even richer semantic structure:

    $$\boldsymbol{v}(\text{Paris}) - \boldsymbol{v}(\text{France}) \approx \boldsymbol{v}(\text{Rome}) - \boldsymbol{v}(\text{Italy})$$

    'Paris is to France as Rome is to Italy'
  - As a pre-processing step: part of the overall end-to-end training

[2] Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean, Efficient Estimation of Word Representations in Vector Space, 2013

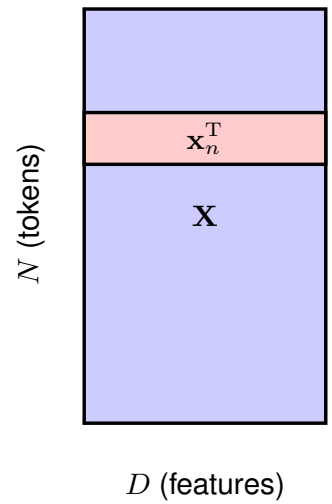
Input Representation

One sentence after embedding:

- A set of vectors $\{\boldsymbol{x}_n\}$ of dimensionality D, where $n = 1, ..., N$
- Vectors are *tokens*, which might correspond to a word or byte pair[3]

To combine data vectors into a matrix $\boldsymbol{X}$ of dimensions $N \times D$:

- n-th row comprises the token vector $\boldsymbol{x}_n^\top$
- This matrix represents one set of input tokens.
- For most applications, we will require a data set containing many sets of tokens.



[3] Byte pair encoding: `https://en.wikipedia.org/wiki/Byte_pair_encoding`

Self-attention (1)

Task: to design such input transformation that:

- Takes $\boldsymbol{X}$ as input and outputs a transformed matrix $\boldsymbol{Y}$ of the same dimensionality

$$\boldsymbol{Y} = \text{TransformerLayer}[\boldsymbol{X}].$$

- Can apply multiple times in succession to construct deep networks

Question: Given a set of tokens $\boldsymbol{x}_1, ..., \boldsymbol{x}_N$ in an embedding space, how to map it to another set $\boldsymbol{y}_1, ..., \boldsymbol{y}_N$ having the same token number but in a new embedding space?

- $\boldsymbol{y}_n$ should depend not just on $\boldsymbol{x}_n$ but on $\boldsymbol{x}_1, ..., \boldsymbol{x}_N$, e.g., a linear combination

$$\boldsymbol{y}_n = \sum_{m=1}^{N} a_{nm} \boldsymbol{x}_m,$$

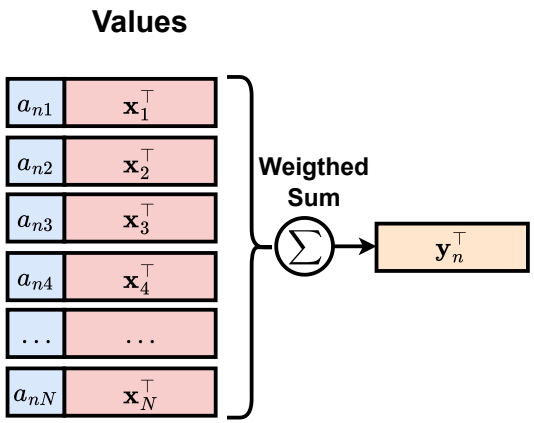
  where a_{nm} are called *attention weights*.

- Coefficients should be close to zero for input tokens that have little influence on the output $\boldsymbol{y}_n$ and largest for inputs that have most influence.

- *How to obtain the attention weights?*

Self-attention (2)

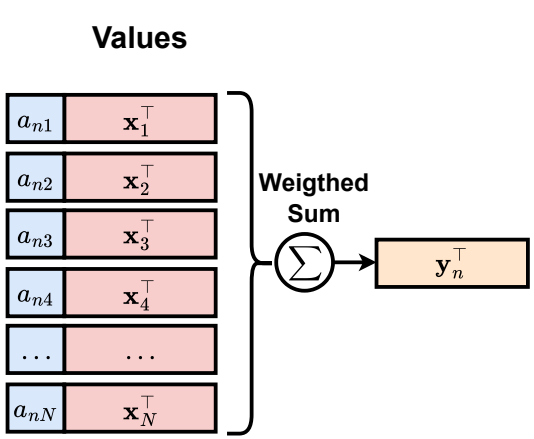
Linear combinations:
$\boldsymbol{x}_n$ are value vectors to span the output space

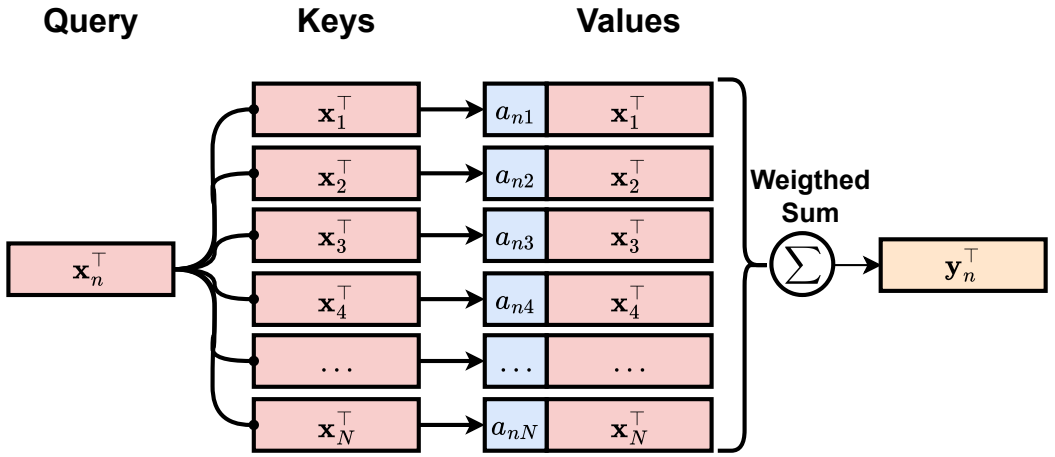


Self-attention (2)

Linear combinations:
$\boldsymbol{x}_n$ are value vectors to span the output space

Transform $\boldsymbol{x}_n$ to $\boldsymbol{y}_n$:
$\boldsymbol{y}_n$ should be closely related to $\boldsymbol{x}_n$; use $\boldsymbol{x}_n$ to measure the similarity to all $\boldsymbol{x}_i$ $\forall i$

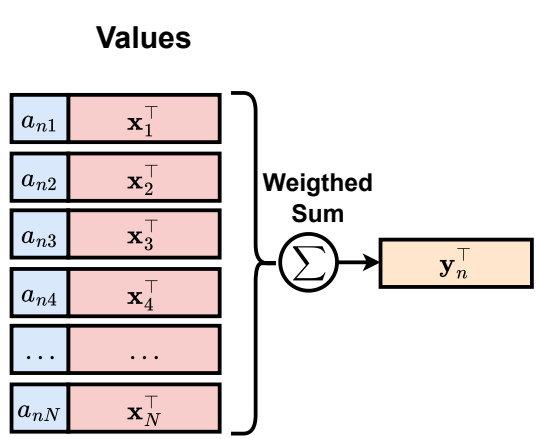


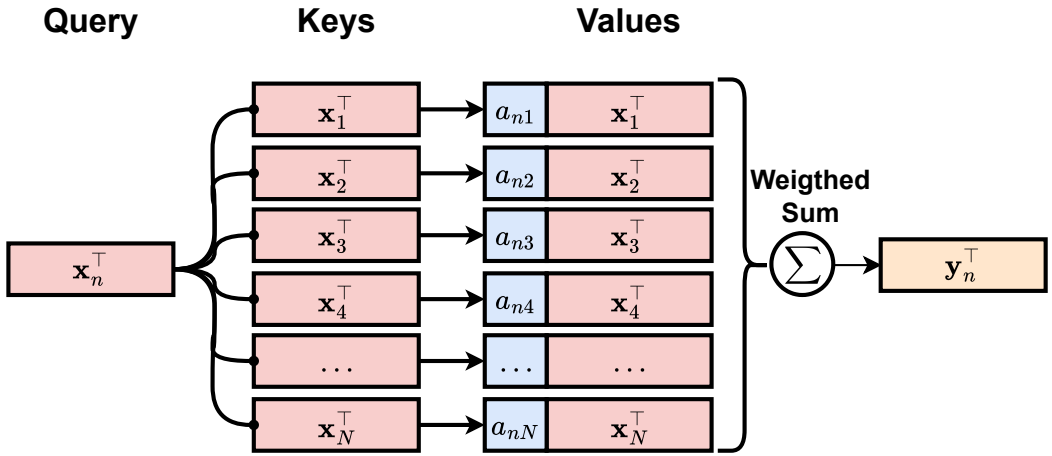


Self-attention (2)

Linear combinations:
$\boldsymbol{x}_n$ are value vectors to span the output space

Transform $\boldsymbol{x}_n$ to $\boldsymbol{y}_n$:
$\boldsymbol{y}_n$ should be closely related to $\boldsymbol{x}_n$; use $\boldsymbol{x}_n$ to measure the similarity to all $\boldsymbol{x}_i$ $\forall i$





Self-attention:

- Vectors $\boldsymbol{x}_n$ as a *value vector* that will be used to create the output tokens
- Vectors $\boldsymbol{x}_n$ direcly as a *key vector* for input token n
- Vectors $\boldsymbol{x}_m$ as a *query vector* for output $\boldsymbol{y}_m$

Namely, vectors $\boldsymbol{x}_n$ are being used as value, key, and query simultaneously.

Self-attention (3)

To measure the similarity between query and key vectors:

- Dot product $\boldsymbol{x}_n^\top \boldsymbol{x}_m$ as similarity measure
- Define the weighting coefficients a_{nm} by using *softmax* to transform dot products:

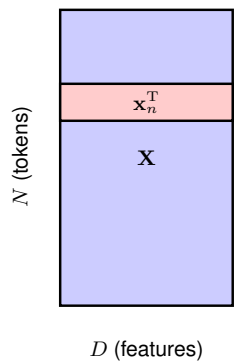
$$a_{nm} = \frac{\exp\left(\boldsymbol{x}_n^\top \boldsymbol{x}_m\right)}{\sum_{m'=1}^{N} \exp\left(\boldsymbol{x}_n^\top \boldsymbol{x}_{m'}\right)}.$$

Note that in this case there is no probabilistic interpretation of the softmax function and it is simply being used to normalize the attention weights appropriately.

Write in matrix notation by using the data matrix $\boldsymbol{X}$ and the output matrix $\boldsymbol{Y}$:

$$\boldsymbol{Y} = \mathrm{softmax}[\boldsymbol{X}\boldsymbol{X}^\top]\boldsymbol{X}.$$

$\mathrm{softmax}[\boldsymbol{L}]$ takes the exponential of every elemnt of a matrix $\boldsymbol{L}$ and then normalises each row independently.



Dot-product self-attention:

- Same sequence to determine the queries, keys and values
- Dot product as similarity measure

Transformation from $\{\boldsymbol{x}_n\}$ to $\{\boldsymbol{y}_n\}$ is fixed: no learnable parameters.

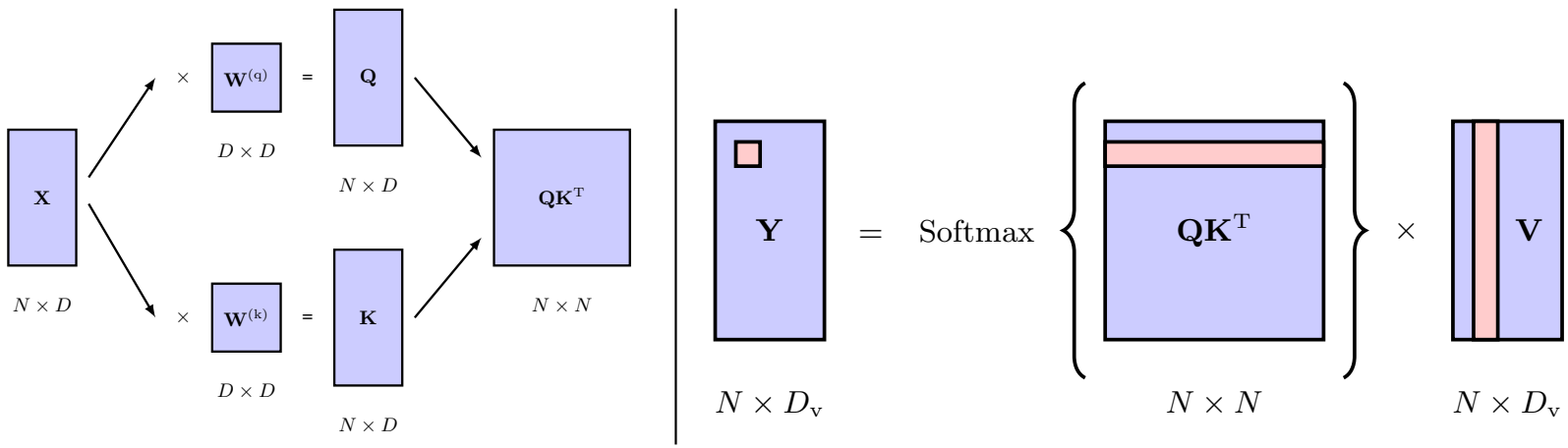
Self-attention with parameters (1)

Defining linear transformations separately for query, key, and value matrices:

$$\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}^{(q)}, \qquad \boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}^{(k)}, \qquad \boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}^{(v)},$$

where weight matrices $\boldsymbol{W}^{(q)} \in \mathbb{R}^{D \times D_q}$, $\boldsymbol{W}^{(k)} \in \mathbb{R}^{D \times D_k}$, and $\boldsymbol{W}^{(v)} \in \mathbb{R}^{D \times D_v}$ represent parameters that will be learned during training. (Typically, $D_k = D$)

$$\boldsymbol{Y} = \text{softmax}[\underbrace{\boldsymbol{Q}\boldsymbol{K}^\top}_{N \times N}] \underbrace{\boldsymbol{V}}_{N \times D_v}$$



We can also include bias parameters in linear transformations (by augmenting the data matrix $\boldsymbol{X}$ with an additional column and weight matrices with an additional row)

Scaled attention with parameters (2)

Gradients of $\mathrm{softmax}$ function can become exponentially small for inputs of high magnitde, e.g., ReLU activation function.

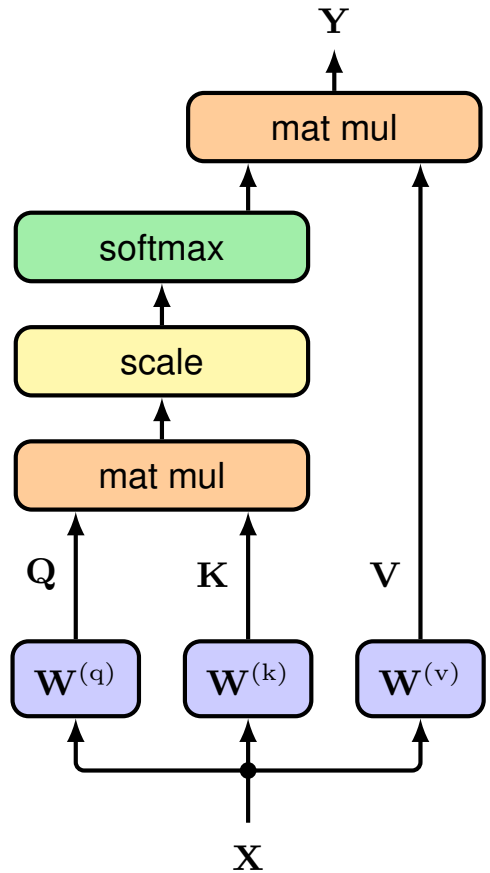
To prevent this, we can *re-scale the product* before applying the softmax function.

- If the elements of the query and key vectors are all independent random variables with zero mean and unit variance, the variance of the dot product will be D_k.

- To normalize the product using the standard deviation given by square root of D_k:

$$\boldsymbol{Y} = \text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{D_k}}\right)\boldsymbol{V}$$

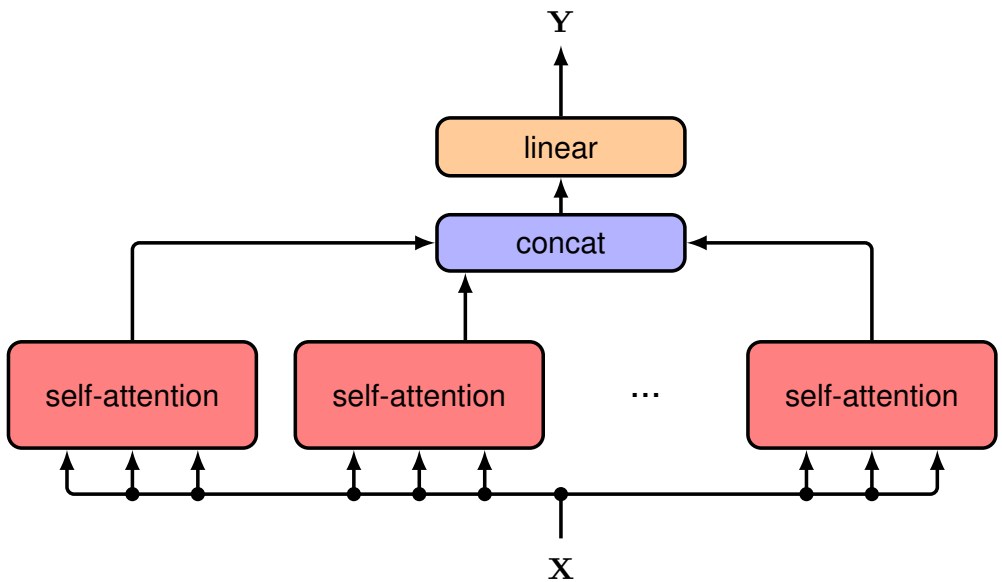
  a.k.a **scaled dot-product self-attention**.



Multi-head Attention

- The attention layer describe so far allows the output vectors to attend to data-dependent patterns of input vectors and is called an *attention head*.
- Use multiple attention heads in parallel to capture multiple patterns: i.e., identically structured copies with independent learnable parameters, similar to filters in CNN.
- Suppose we have H heads indexed by $h = 1, ..., H$:

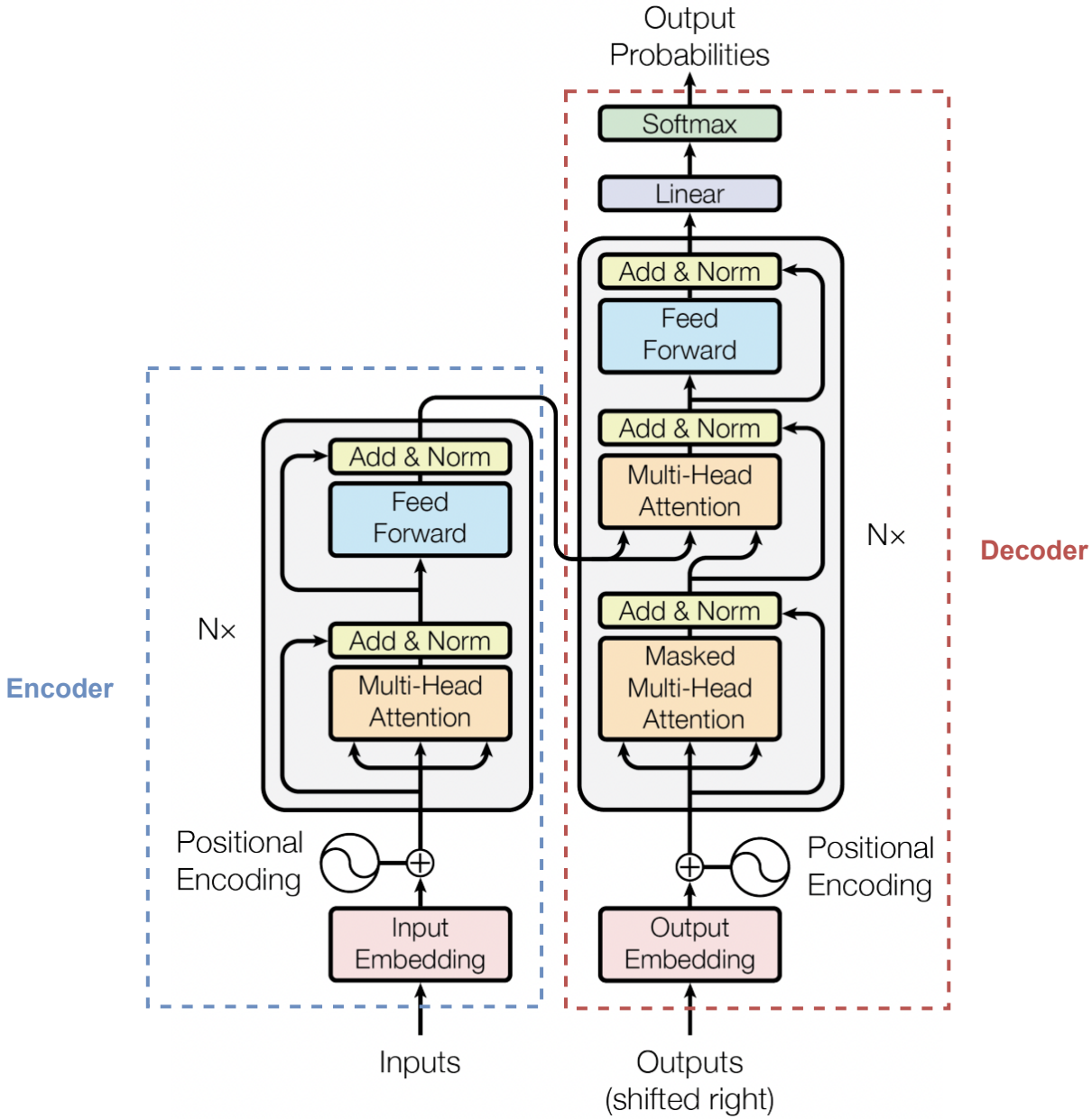
$$\boldsymbol{H}_h = \text{Attention}\left(\boldsymbol{Q}_h, \boldsymbol{K}_h, \boldsymbol{V}_h\right).$$



- The heads are first concatenated into a single matrix and the result is then linearly transformed using a matrix $\boldsymbol{W}^{(o)}$ to give a combined output in the form:

$$\boldsymbol{Y}(\boldsymbol{X}) = \text{Concat}[\boldsymbol{H}_1, ..., \boldsymbol{H}_H]\boldsymbol{W}^{(o)}.$$

Other layers in transformers (1)



See paper by Vaswani et al. (2017)

Other layers in transformers (2)

Residual connections and layer normalization:

- To improve training efficiency, transformers use *residual connections* that bypass the multi-head structure, together with *layer normalization*:

$$\boldsymbol{Z} = \text{LayerNorm}[\boldsymbol{Y}(\boldsymbol{X}) + \boldsymbol{X}].$$

- The layer normalization can be replaced by *pre-norm* (more effective):

$$\boldsymbol{Z} = \boldsymbol{Y}(\boldsymbol{X}') + \boldsymbol{X}, \quad \text{where} \quad \boldsymbol{X}' = \text{LayerNorm}[\boldsymbol{X}].$$

Feedforward layers:

- Non-linearity enters through attention weights via $\mathrm{softmax}$ function.
- Output vectors are still constrained to lie in the subspace spanned by the input vectors, which limits the expressive capabilities of the attention layer.
- To post-process the output of each layer using a standard nonlinear neural network with D inputs and D outputs, denoted $\text{FF}[\cdot]$ (sometimes also as $\text{MLP}[\cdot]$).

$$\tilde{\boldsymbol{X}} = \text{LayerNorm}[\text{FF}[\boldsymbol{Z}] + \boldsymbol{Z}].$$

- The same shared network is applied to each of the output vectors.
- The pre-norm form is $\tilde{\boldsymbol{X}} = \text{FF}[\boldsymbol{Z}'] + \boldsymbol{Z}, \quad \text{where} \quad \boldsymbol{Z}' = \text{LayerNorm}[\boldsymbol{Z}]$.

Other layers in transformers (3)

Positional encoding:

- $\boldsymbol{W}_h^{(q)}$, $\boldsymbol{W}_h^{(k)}$ and $\boldsymbol{W}_h^{(v)}$ are shared across input tokens: permuting the order of input tokens, i.e., rows of $\boldsymbol{X}$, results in the same permutation of rows of output matrix $\tilde{\boldsymbol{X}}$.
- The transformer is *equivariant* with respect to input permutations.
  - The food was bad, not good at all!
  - The food was good, not bad at all!

  They contain the same tokens but have very different meanings: token order is crucial for most sequential processing tasks.
- To construct a position encoding vector $\boldsymbol{r}_n$ associated with each input position n:
  - Concatenation with $\boldsymbol{x}_n$ leads to increased dimensionality.
  - Simply add the position vectors onto the token vectors to give $\tilde{\boldsymbol{x}}_n = \boldsymbol{x}_n + \boldsymbol{r}_n$.
  - Residual connection allows it to be passed from one transformer layer to the next

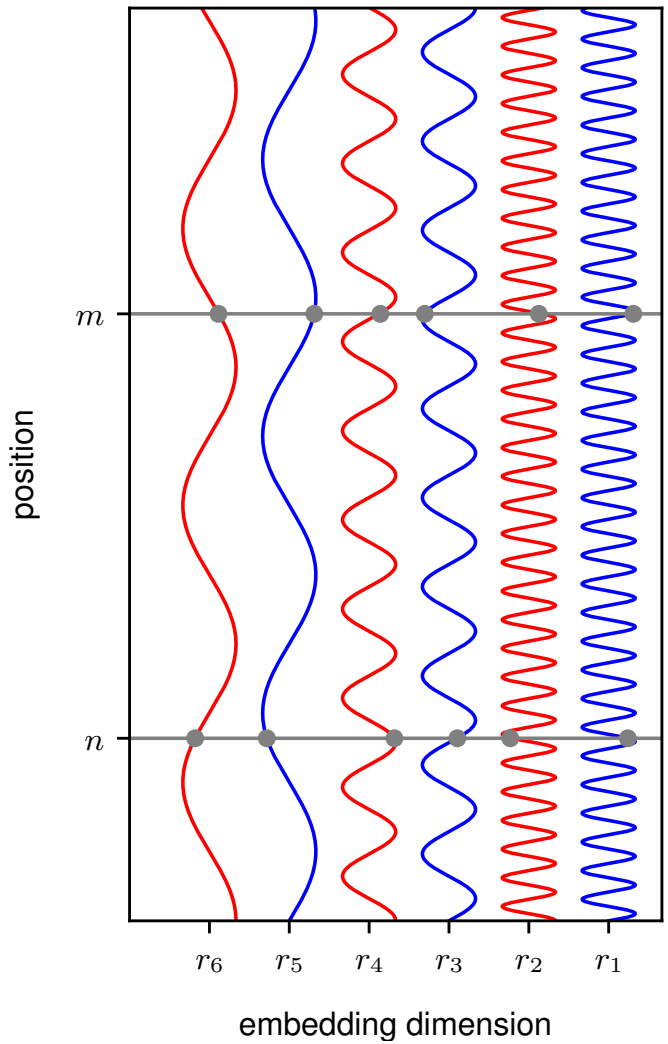
Other layers in transformers (4)

Positional encoding:

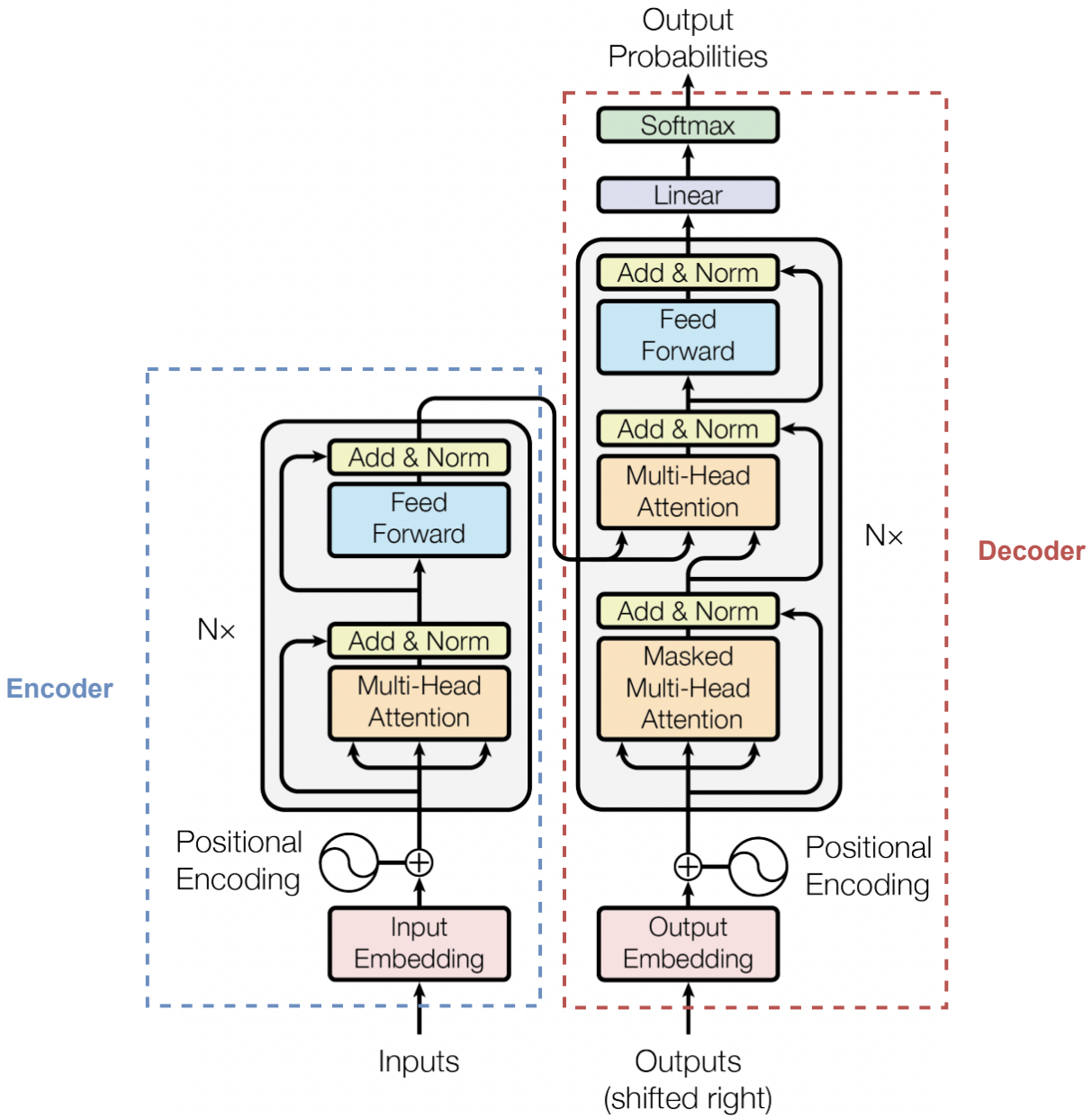
- Sinusoidal positional encoding (bounded, relative position): for a given position n the associated position-encoding vector has components r_{ni} given by

$$r_{ni} = \begin{cases} \sin\left(\frac{n}{L^{i/D}}\right), & \text{if } i \text{ is even,} \\ \cos\left(\frac{n}{L^{(i-1)/D}}\right), & \text{if } i \text{ is odd.} \end{cases}$$

  Encoding at $n + k$ can be represented as a linear combination of encoding at n, in which coefficients do not depend on the absolute position but only on k.



Encoder + Decoder



See paper by Vaswani et al. (2017)

Encoder transformers (1)

Encoder: takes sequences as input and produces fixed-length vectors.

Bidirectional Encoder Representations from Transformers (BERT):

- To pre-train a language model using a large corpus of text
- Then to fine-tune the model using *transfer learning* for a broad range of downstream tasks each of which requires a smaller application-specific dataset.

Training procedure:

- A randomly chosen subset of the tokens, say 15%, are replaced with a special token denoted $\langle$mask$\rangle$. The model is trained to predict the missing tokens at the corresponding output nodes (i.e., *self-supervised learning*).
  - Orignal input: I swam across the river to get to the other bank.
  - Masked input: I $\langle$mask$\rangle$ across the river to get to the $\langle$mask$\rangle$ bank.
- Term 'bidirectional' refers to that the network sees words both before and after the masked word and can use both sources of information to make a prediction.
- In practice, of the 15% of randomly selected tokens, 80% are replaced with $\langle$mask$\rangle$, 10% are replaced with a word selected at random from the vocabulary, and 10% are retained, but they still have to be correctly predicted at the output.[4]

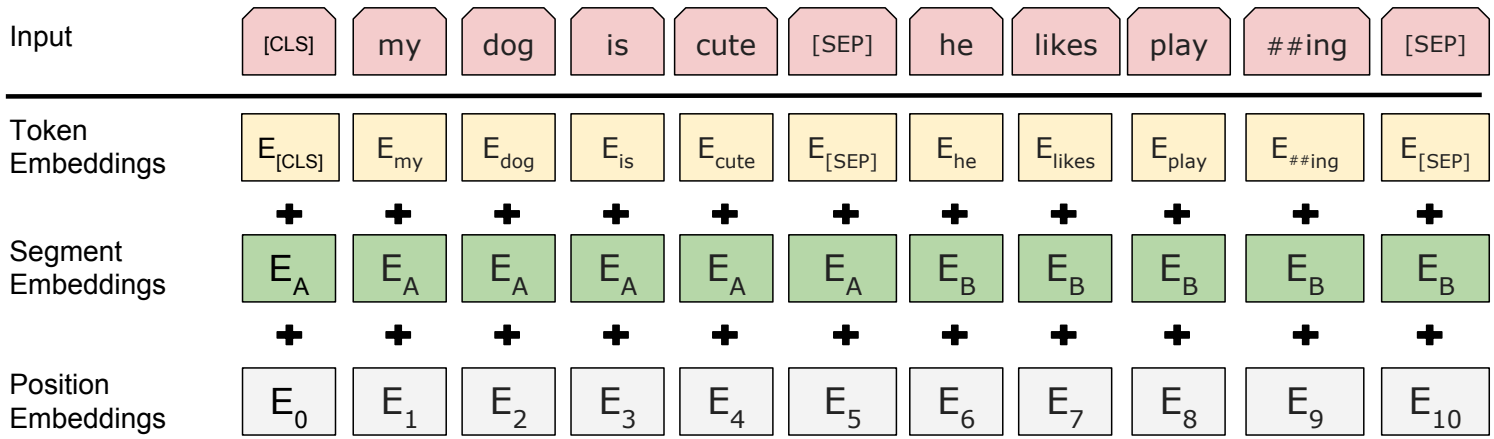
[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova, BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, 2018

Encoder transformers (2)

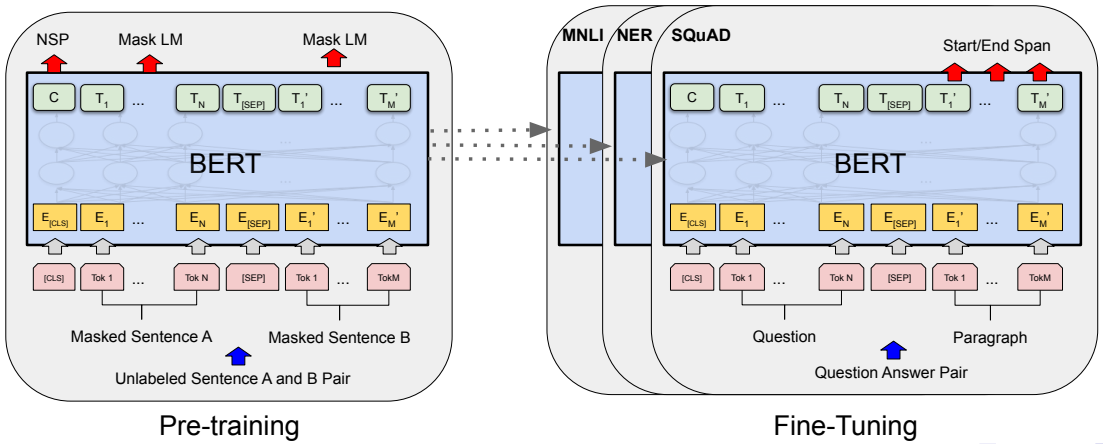
Bidirectional Encoder Representations from Transformers (BERT):

This use of self-supervised learning led to a paradigm shift in which a large model is first pre-trained using unlabelled data and then subsequently fine-tuned using supervised learning based on a much smaller set of labelled data.

- Input representation:

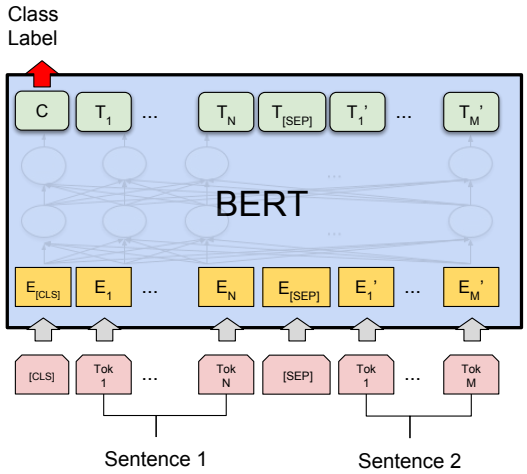


- Pre-training and fine-tuning:

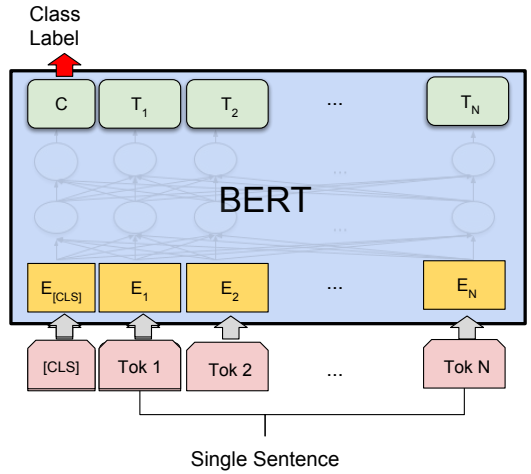


Encoder transformers (3)

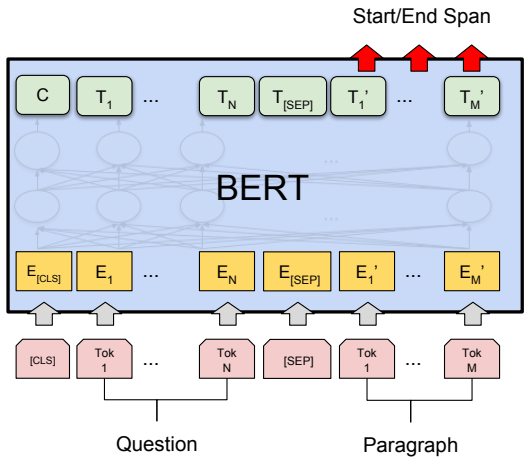
Bidirectional Encoder Representations from Transformers (BERT):



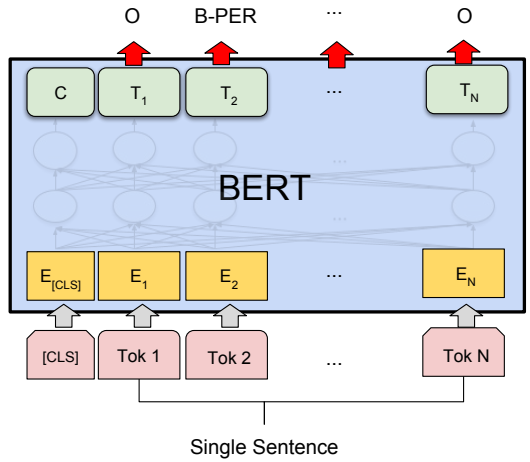
(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG



(b) Single Sentence Classification Tasks: SST-2, CoLA



(c) Question Answering Tasks: SQuAD v1.1

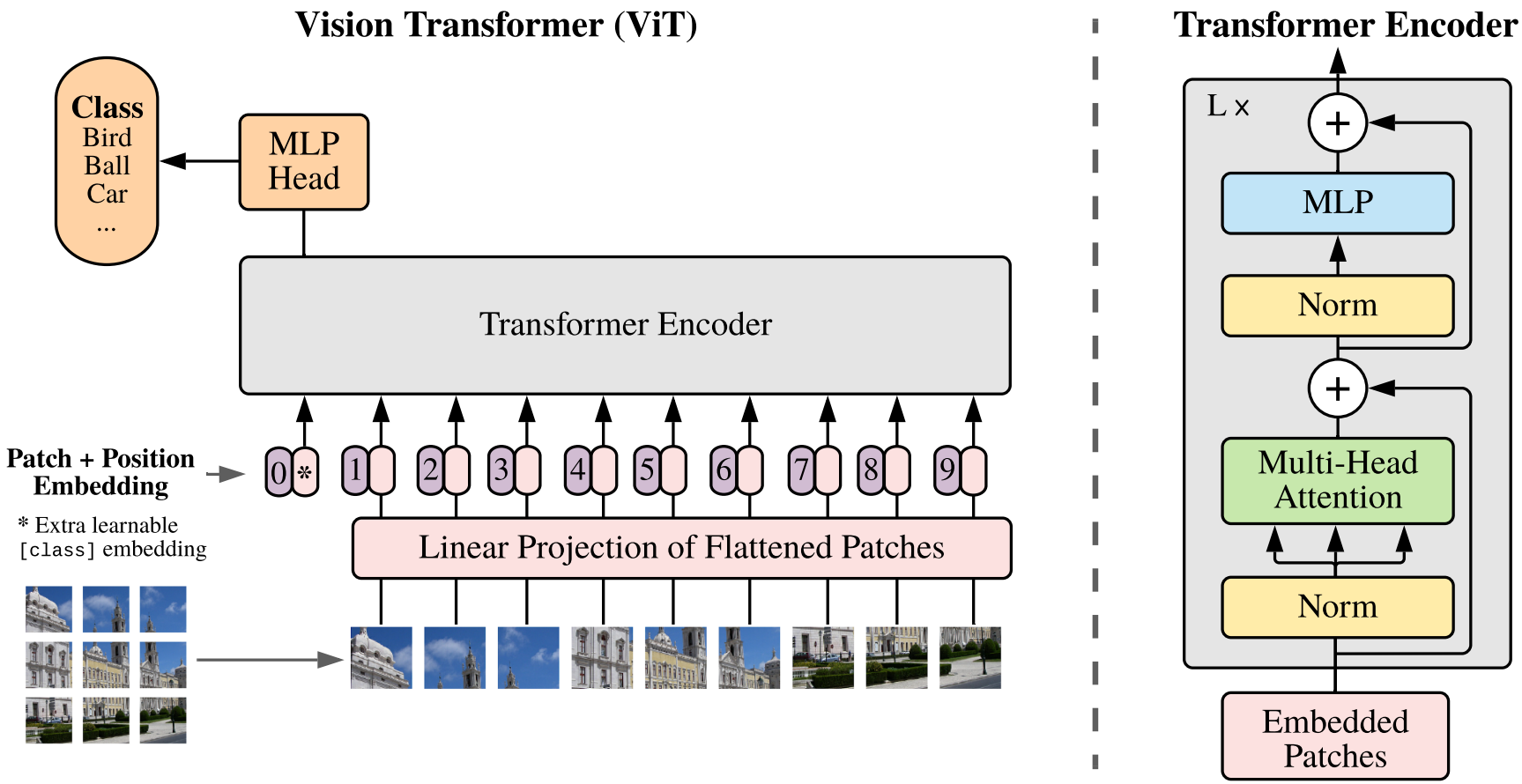


(d) Single Sentence Tagging Tasks: CoNLL-2003 NER

Encoder transformers (4)

Vision transformers

- Slices an image to patches
- Embed all patches to form tokens
- Apply encoder transformers with pre-training



Decoder transformers (1)

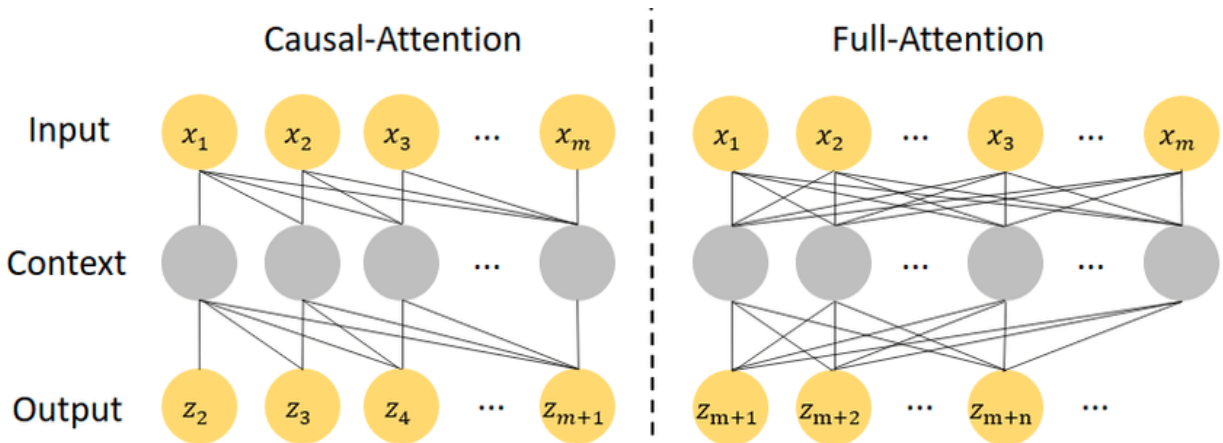
Decoder: used as generative models that create output sequence of tokens,

Autoregressive model

- Take as input a sequence consisting of the first $n-1$ tokens, and its corresponding output represents the conditional distribution for token n
- Draw a sample from this distribution and extend the sequence to n tokens
- Feed back the new sequence through model to give a distribution over token $n+1$

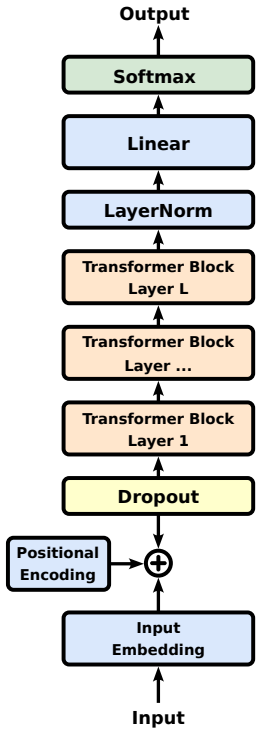
Generative Pre-trained Transformers (GPT)

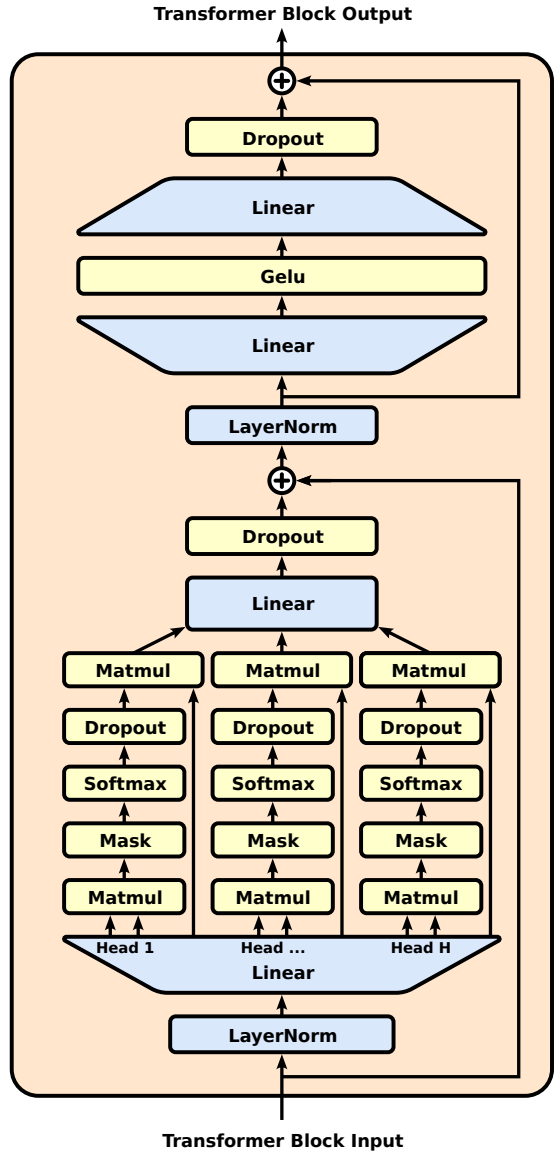
- Shift input sequence to the right by one step: $\boldsymbol{x}_n$ corresponds to $\boldsymbol{y}_{n+1}$, with target $\boldsymbol{x}_{n+1}$, and prepend a special token $\langle \text{start} \rangle$ in the first position of input sequence
- *Causal attention*: set to zero all of attention weights of any later token in the sequence, i.e., setting corresponding pre-activation values to $-\infty$ before $\mathrm{softmax}$



Decoder transformers (2)

Generative Pre-trained Transformer (GPT)





[5]

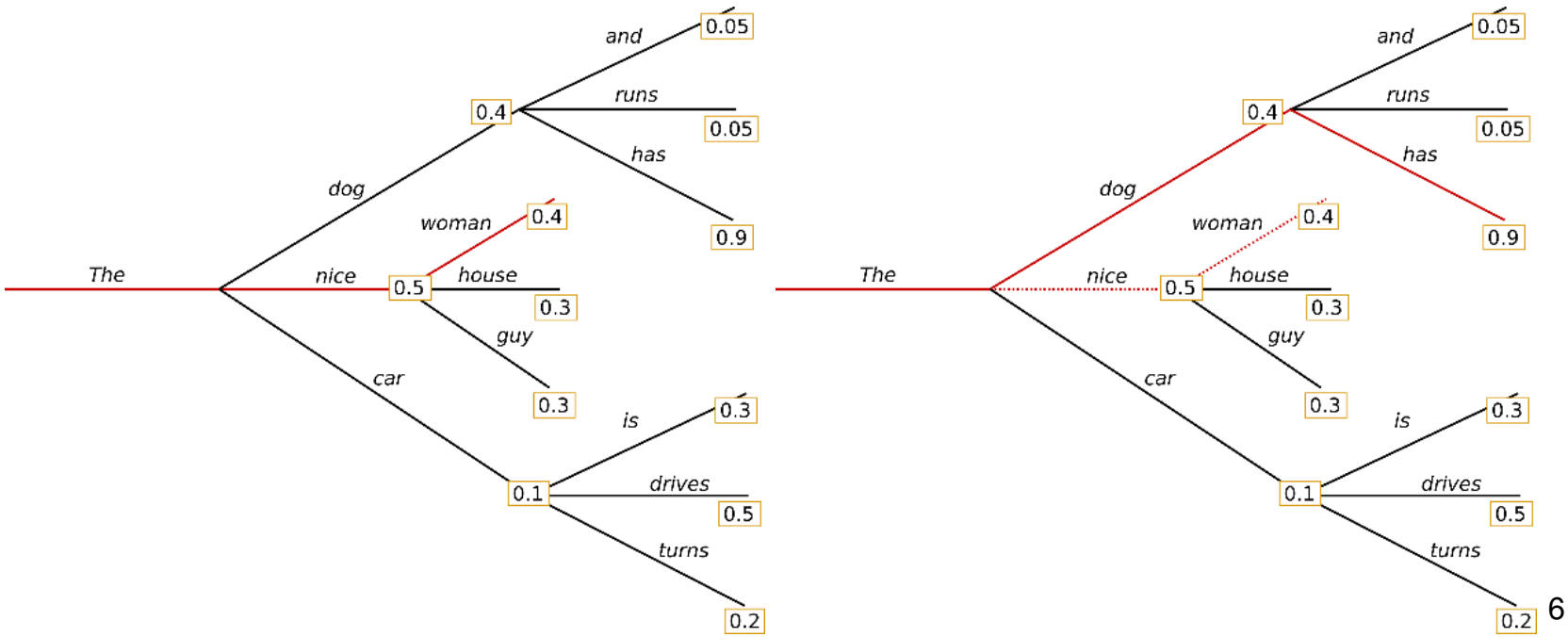
[5] Image from: https://en.wikipedia.org/wiki/Generative_pre-trained_transformer

Decoder transformers (3)

How to generate the next token?

Sampling scheme:

- *Greedy search*: simply to select tokens with the *highest probability*.
- *Beam search*: to maintain a set of B hypotheses, where B is called the *beam width*, each consisting of a sequence up to step $n + B$. To select tokens with the *highest total probability* of the textended sequences.



[6]

[6] Image from: `https://huggingface.co/blog/how-to-generate`

Sequence-to-sequence (seq2seq) transformers

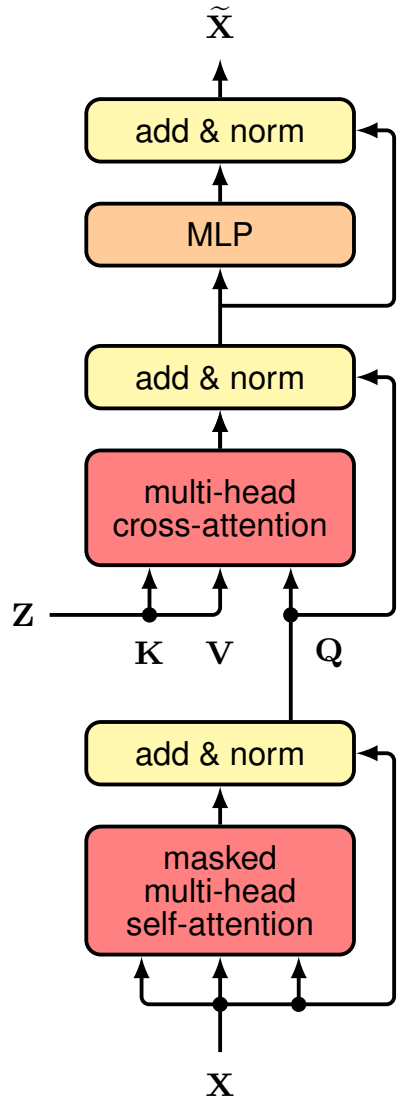
Cross-attention: combine encoder with decoder

Consider sequence-to-sequence modeling (Seq2seq):
For example, *translation task*: translating an English sentence into a Dutch sentence.

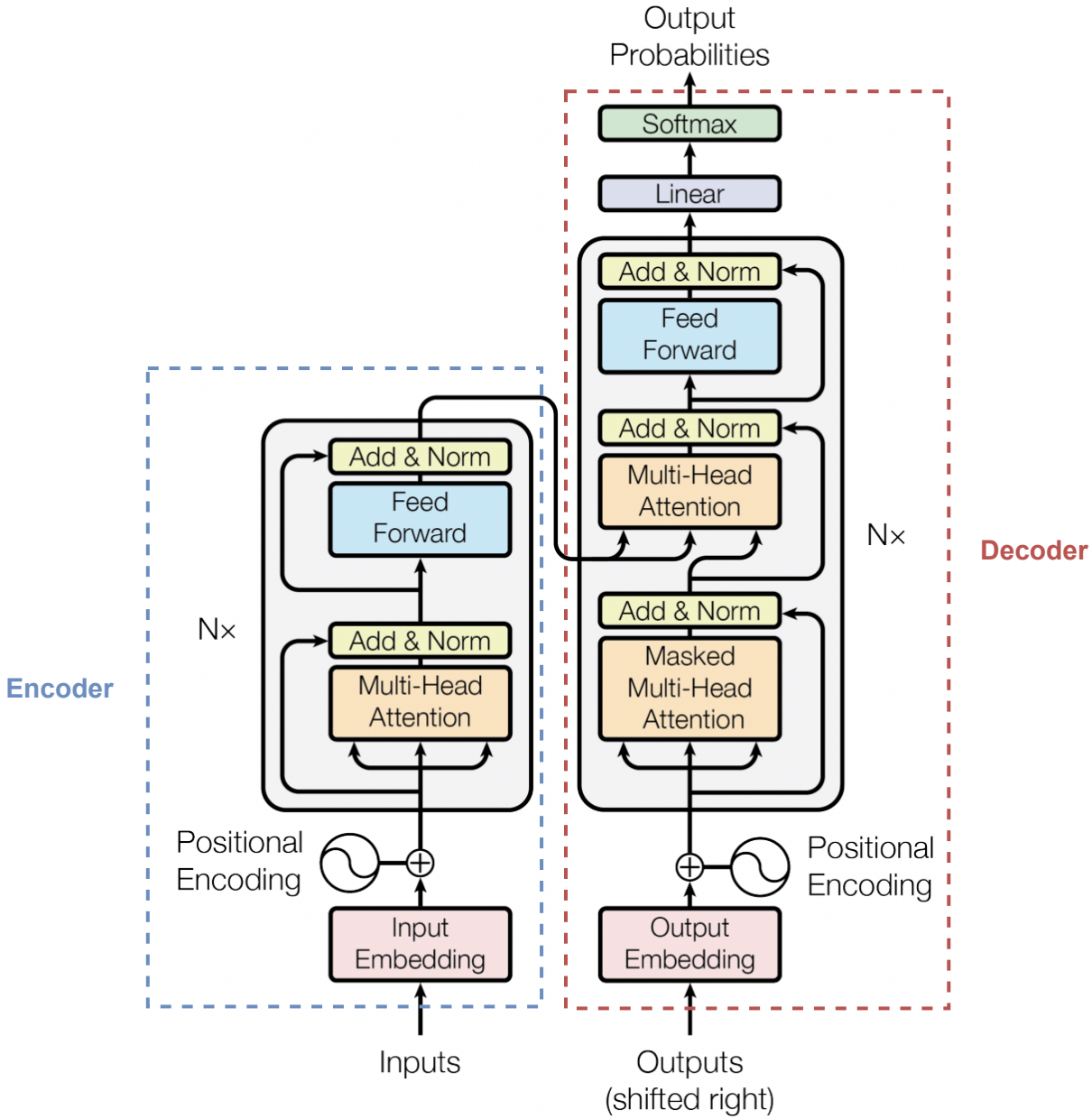
1. Use an encoder transformer to map the input token sequence into a suitable internal representation $\mathcal{Z}$
2. Use a decoder transformer to generate the token sequence corresponding to the Dutch output, token by token
3. Condition the outputon the entire input sequence corresponding to the English sentence

To incorporate $\mathcal{Z}$ into the decoder, we use the **cross attention** in *decoder*:

- Query vectors come from the sequence being generated, i.e., from decoder
- Key and value vectors come from the sequence represented by $\mathcal{Z}$, i.e., from encoder



Full transformers



See paper by Vaswani et al. (2017)

Recap

Language Models

Recurrent Neural Networks

Transformers

END LECTURE