CS447: Natural Language Processing
http://courses.engr.illinois.edu/cs447

Lecture 12: Attention and Transformers

Julia Hockenmaier
juliahmr@illinois.edu
3324 Siebel Center

Lecture 12:
Attention and Transformers

Attention Mechanisms

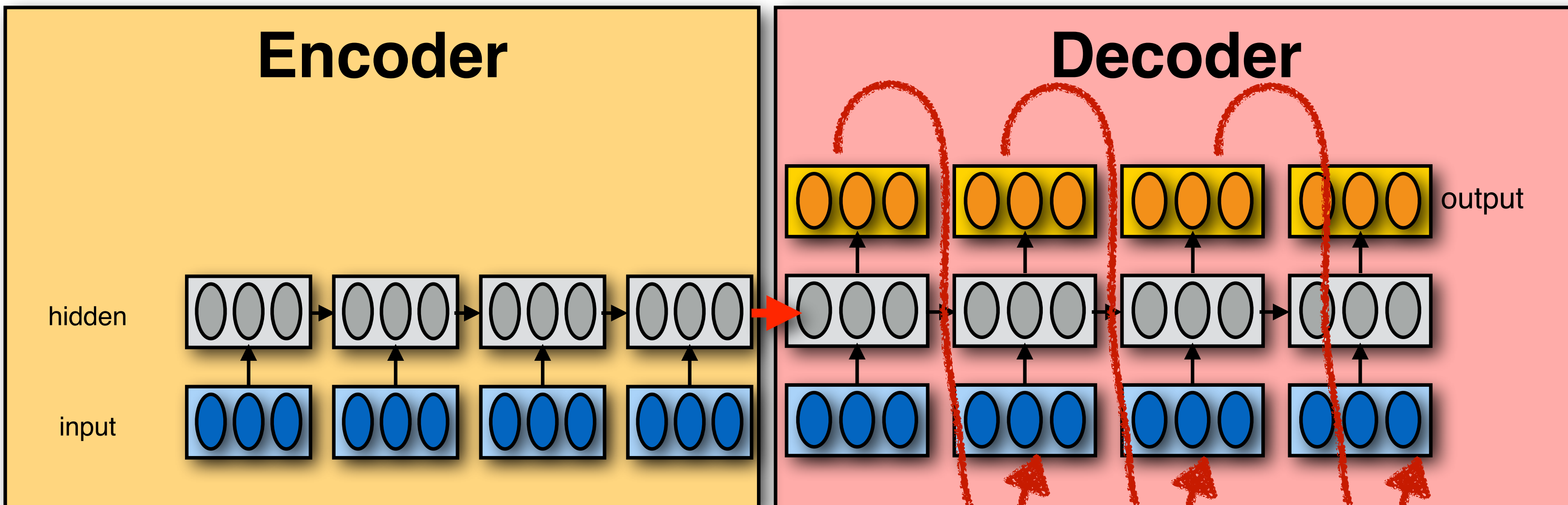
Encoder-Decoder (seq2seq) model

Task: Read an input sequence
and return an output sequence

- Machine translation: translate source into target language
- Dialog system/chatbot: generate a response

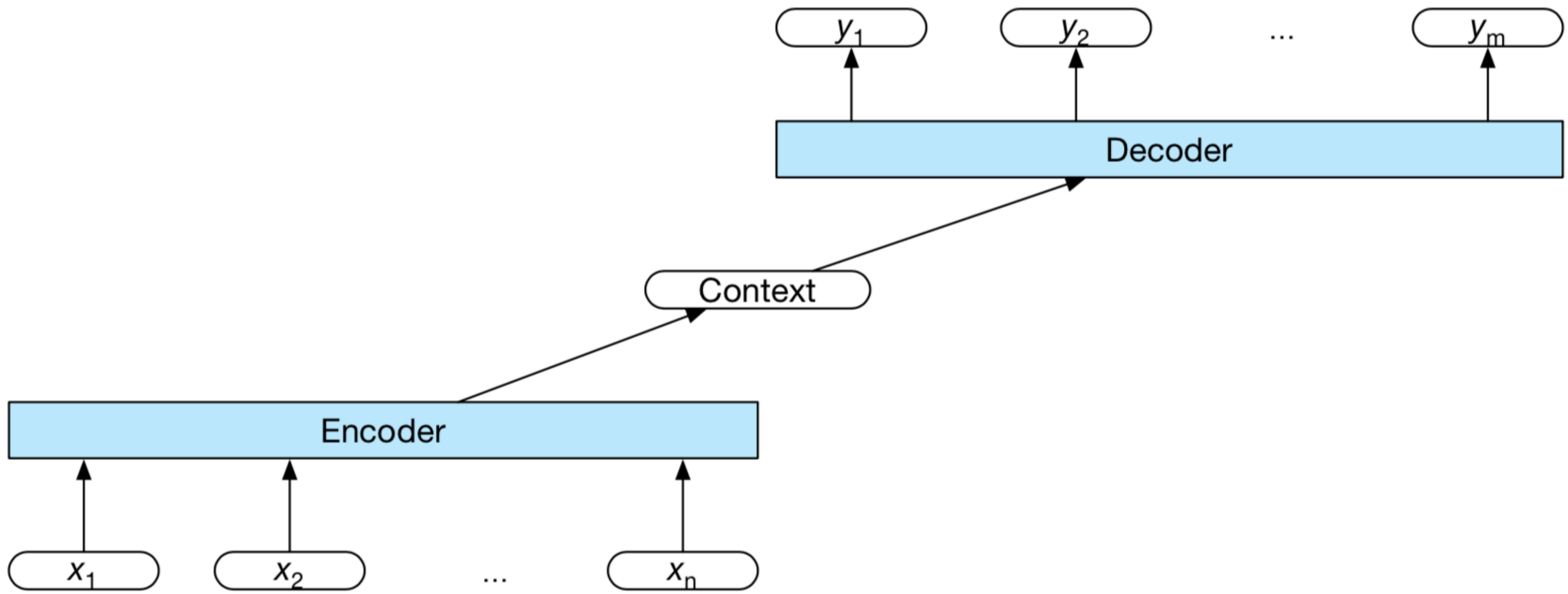
Reading the input sequence: **RNN Encoder**

Generating the output sequence: **RNN Decoder**



A more general view of seq2seq

Insight 1: In general, **any function of the encoder's output** can be used as a representation of the context we want to condition the decoder on.



Insight 2: We can feed the context in **at any time step** during decoding (not just at the beginning).

Adding attention to the decoder

Basic idea: Feed a *d*-dimensional representation of the entire (arbitrary-length) input sequence into the decoder *at each time step during decoding.*

This representation of the input can be a **weighted average of the encoder's representation of the input** (i.e. its output)

The **weights** of each encoder output element tell us how much attention we should pay to different parts of the input sequence

Since different parts of the input may be more or less important for different parts of the output, we want to **vary the weights** over the input during the decoding process.

(Cf. Word alignments in machine translation)

Adding attention to the decoder

We want to **condition the output** generation of the decoder on a **context-dependent representation of the input** sequence.

Attention computes a probability **distribution over the encoder's hidden states** that depends on the **decoder's current hidden state**

(This distribution is **computed anew for each output symbol**)

This attention distribution is used to compute **a weighted average of the encoder's hidden state vectors**.

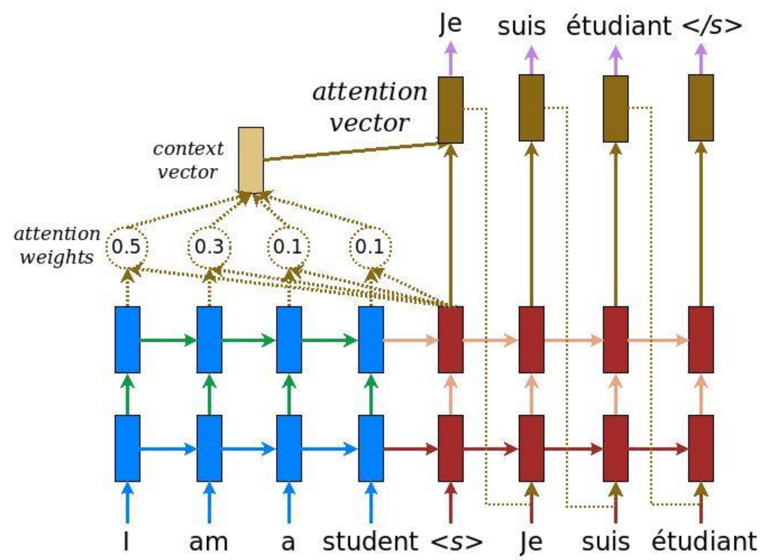
This **context-dependent embedding of the input sequence** is fed into the output of the decoder RNN.

Attention, more formally

Define a **probability distribution** $\alpha^{(t)} = \left(\alpha_1^{(t)}, \ldots, \alpha_S^{(t)}\right)$
over the ***S*** **elements of the input** sequence
that depends on the current output element *t*

Use this distribution to compute a **weighted average of the encoder's output** $\sum_{s=1..S} \alpha_s^{(t)} \mathbf{o}_s$ or hidden states $\sum_{s=1..S} \alpha_s^{(t)} \mathbf{h}_s$

and feed that into the decoder.



hhttps://www.tensorflow.org/tutorials/text/nmt_with_attention

Attention, more formally

1. Compute **a probability distribution** $\alpha^{(t)} = \left(\alpha_1^{(t)}, \ldots, \alpha_S^{(t)}\right)$
over the *encoder's* hidden states $\mathbf{h}^{(s)}$
that depends on the *decoder's* current $\mathbf{h}^{(t)}$

$$\alpha_s^{(t)} = \frac{\exp\left(s(\mathbf{h}^{(t)}, \mathbf{h}^{(s)})\right)}{\sum_{s'} \exp\left(s(\mathbf{h}^{(t)}, \mathbf{h}^{(s')})\right)}$$

2. Use $\alpha^{(t)}$ to compute **a weighted avg.** $\mathbf{c}^{(t)}$ of the *encoder's* $\mathbf{h}^{(s)}$:

$$\mathbf{c}^{(t)} = \sum_{s=1..S} \alpha_s^{(t)} \mathbf{h}^{(s)}$$

3. Use both $\mathbf{c}^{(t)}$ and $\mathbf{h}^{(t)}$ to compute **a new output** $\mathbf{o}^{(t)}$, e.g. as

$$\mathbf{o}^{(t)} = \tanh\left(W_1 \mathbf{h}^{(t)} + W_2 \mathbf{c}^{(t)}\right)$$

Defining Attention Weights

Hard attention (degenerate case, non-differentiable):

$\alpha^{(t)} = \left(\alpha_1^{(t)}, \ldots, \alpha_S^{(t)}\right)$ is a **one-hot vector**

(e.g. 1 = most similar element to decoder's vector, 0 = all other elements)

Soft attention (general case):

$\alpha^{(t)} = \left(\alpha_1^{(t)}, \ldots, \alpha_S^{(t)}\right)$ is **not a one-hot**

— Use the **dot product** (no learned parameters):

$$s\left(\mathbf{h}^{(t)}, \mathbf{h}^{(s)}\right) = \mathbf{h}^{(t)} \cdot \mathbf{h}^{(s)}$$

— Learn a **bilinear matrix** W:

$$s\left(\mathbf{h}^{(t)}, \mathbf{h}^{(s)}\right) = \left(\mathbf{h}^{(t)}\right)^T W \mathbf{h}^{(s)}$$

— Learn **separate weights** for the hidden states:

$$s\left(\mathbf{h}^{(t)}, \mathbf{h}^{(s)}\right) = \mathbf{v}^T \tanh\left(W_1 \mathbf{h}^{(t)} + W_2 \mathbf{h}^{(s)}\right)$$

Lecture 12:
Attention and Transformers

Transformers

Vashwani et al. Attention is all you need, NIPS 2017

Transformers

Sequence transduction model based on **attention** (**no convolutions or recurrence**)

— easier to parallelize than recurrent nets

— faster to train than recurrent nets

— captures more long-range dependencies than CNNs with fewer parameters

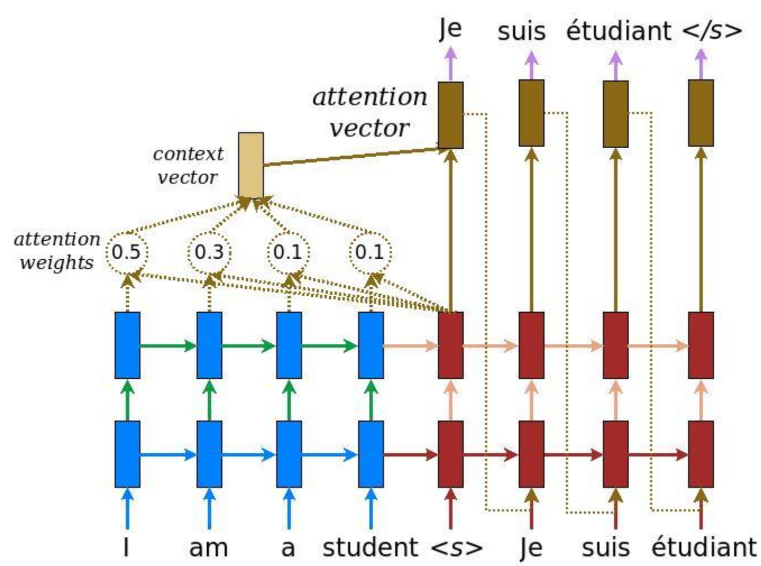
Transformers use stacked self-attention and position-wise, fully-connected layers for the encoder and decoder

Transformers form the basis of BERT, GPT(2-3), and other state-of-the-art neural sequence models.

Seq2seq attention mechanisms

Define a **probability distribution** $\alpha^{(t)} = \left(\alpha_1^{(t)}, \ldots, \alpha_S^{(t)}\right)$ over the ***S* elements of the input** sequence **that depends on the current output** element *t*

Use this distribution to compute a **weighted average of the encoder's output** $\sum_{s=1..S} \alpha_s^{(t)} \mathbf{o}_s$ or hidden states $\sum_{s=1..S} \alpha_s^{(t)} \mathbf{h}_s$ and feed that into the decoder.



hhttps://www.tensorflow.org/tutorials/text/nmt_with_attention

Self-Attention

Attention so far (in seq2seq architectures):

In the ***decoder*** (which has access to the complete input sequence), compute attention weights over ***encoder*** positions that depend on each ***decoder*** position

Self-attention:

If the ***encoder*** has access to the complete input sequence, we can also compute attention weights over ***encoder*** positions that depend on each ***encoder*** position

self-attention: ***encoder***

For each ~~*decoder*~~ position *t*…,

…Compute an attention weight for each *encoder* position *s*

…Renormalize these weights (that depend on *t*) w/ softmax to get a new weighted avg. of the input sequence vectors

Self-attention: Simple variant

Given *T* *k*-dimensional ***input* vectors** $\mathbf{x}^{(1)} \ldots \mathbf{x}^{(i)} \ldots \mathbf{x}^{(T)}$,
compute *T* *k*-dimensional ***output* vectors** $\mathbf{y}^{(1)} \ldots \mathbf{y}^{(i)} \ldots \mathbf{y}^{(T)}$
where each **output** $\mathbf{y}^{(i)}$ is a **weighted average** of the **input** vectors, and where the **weights** w_{ij} depend on $\mathbf{y}^{(i)}$ and $\mathbf{x}^{(j)}$

$$\mathbf{y}^{(i)} = \sum_{j=1..T} w_{ij} \mathbf{x}^{(j)}$$

Computing weights w_{ij} naively (no learned parameters)

Dot product: $w'_{ij} = \sum_k x_k^{(i)} x_k^{(j)}$

Followed by softmax: $w_{ij} = \dfrac{\exp(w'_{ij})}{\sum_j \exp(w'_{ij})}$

Towards more flexible self-attention

To compute $\mathbf{y}^{(i)} = \sum_{j=1..T} w_{ij}\mathbf{x}^{(j)}$, we must…

… take the element $\mathbf{x}^{(i)}$ …

…decide the weight w_{ij} of each $\mathbf{x}^{(j)}$ depending on $\mathbf{x}^{(i)}$

… average all elements $\mathbf{x}^{(j)}$ according to their weights

Observation 1: Dot product-based weights are large when $\mathbf{x}^{(i)}$, $\mathbf{x}^{(j)}$ are similar. But we may want a more flexible approach.

Idea 1: *Learn* attention weights w_{ij} that depend on $\mathbf{x}^{(i)}$ and $\mathbf{x}^{(j)}$ in a manner that works best for the task

Observation 2: This weighted average is still just a simple function of the original $\mathbf{x}^{(j)}$s

Idea 2: *Learn* weights that re-weight the elements of $\mathbf{x}^{(j)}$ in a manner that works best for the task

Self-attention with queries, keys, values

Let's add learnable parameters (three $k \times k$ weight matrices $\mathbf{W}$),
that allow us turn any input vector $\mathbf{x}^{(i)}$ into **three versions**:

– **Query** vector $\mathbf{q}^{(i)} = \mathbf{W}_q \mathbf{x}^{(i)}$ to compute averaging weights *at* pos. i

– **Key** vector: $\mathbf{k}^{(i)} = \mathbf{W}_k \mathbf{x}^{(i)}$ to compute averaging weights *of* pos. i

– **Value** vector: $\mathbf{v}^{(i)} = \mathbf{W}_v \mathbf{x}^{(i)}$ to compute the *value* of pos. i to be averaged

The **attention weight** of the *j*-th position used in the weighted average
at the ***i*-th position** depends on the **query of *i*** and the **key of *j***:

$$w_j^{(i)} = \frac{\exp\left(\mathbf{q}^{(i)}\mathbf{k}^{(j)}\right)}{\sum_j \exp\left(\mathbf{q}^{(i)}\mathbf{k}^{(j)}\right)} = \frac{\exp\left(\sum_l q_l^{(i)} k_l^{(j)}\right)}{\sum_j \exp\left(\sum_l q_l^{(i)} k_l^{(j)}\right)}$$

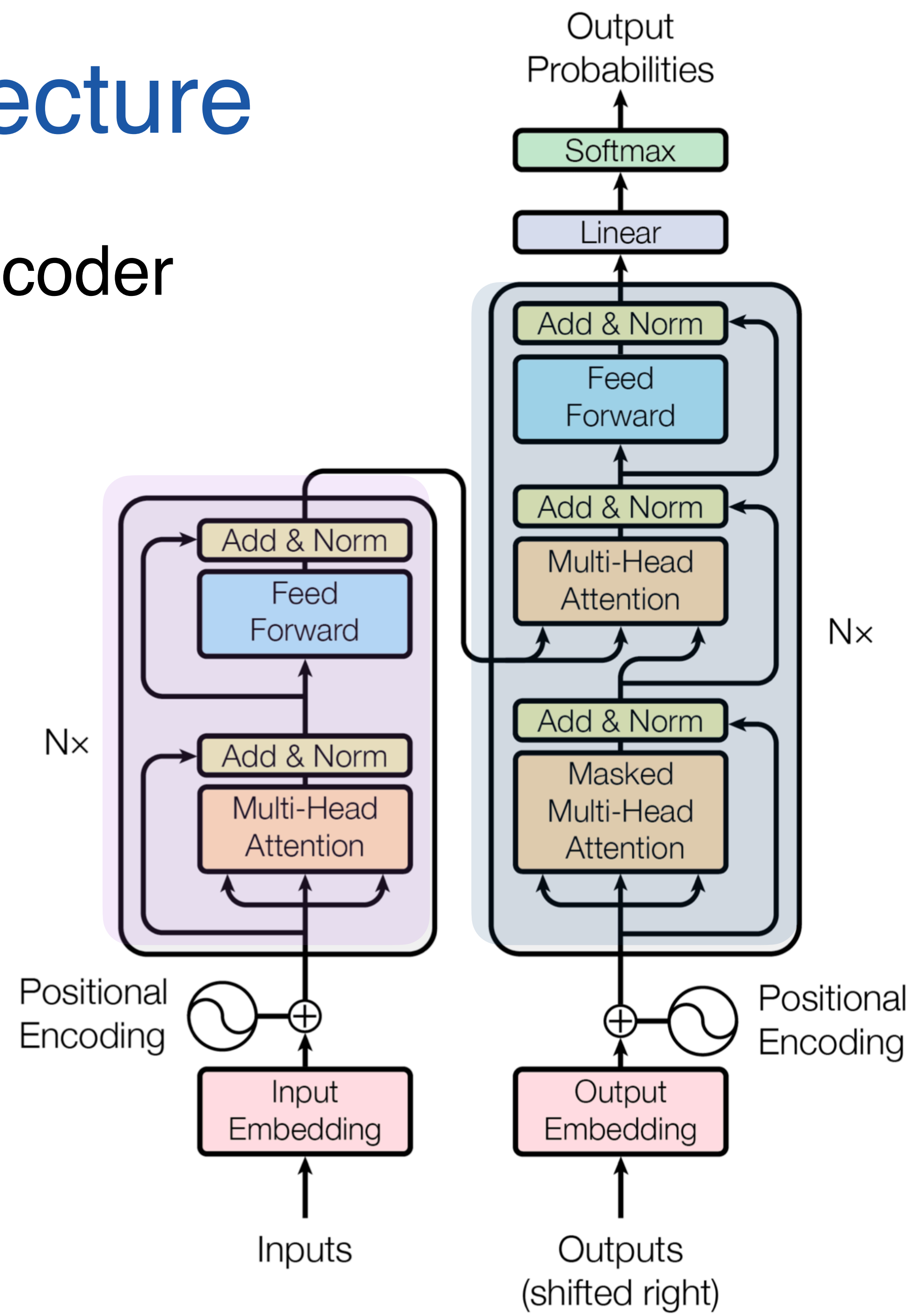
The **new output vector for the *i*-th position** depends on
the **attention weights** and **value** vectors of all **input positions j**:

$$\mathbf{y}^{(i)} = \sum_{j=1..T} w_j^{(i)} \mathbf{v}^{(j)}$$

Transformer Architecture

Non-Recurrent Encoder-Decoder architecture

- No hidden states
- Context information captured via attention and positional encodings
- Consists of stacks of layers with various sublayers



Encoder

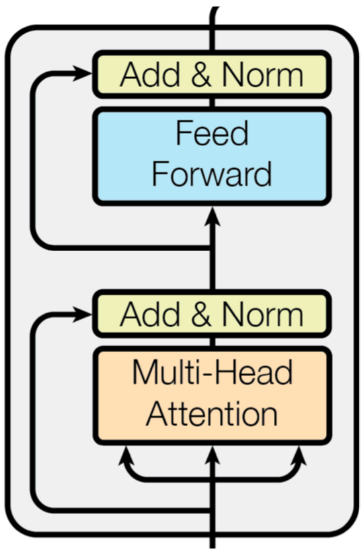
A stack of **N=6 identical layers**

All layers and sublayers are 512-dimensional

Each layer consists of **two sublayers**

— one **multi-head self attention** layer

— one **position-wise feed forward** layer



Each sublayer is followed by an **"Add & Norm"** layer:

… a **residual connection** $\mathbf{x} + \text{Sublayer}(\mathbf{x})$

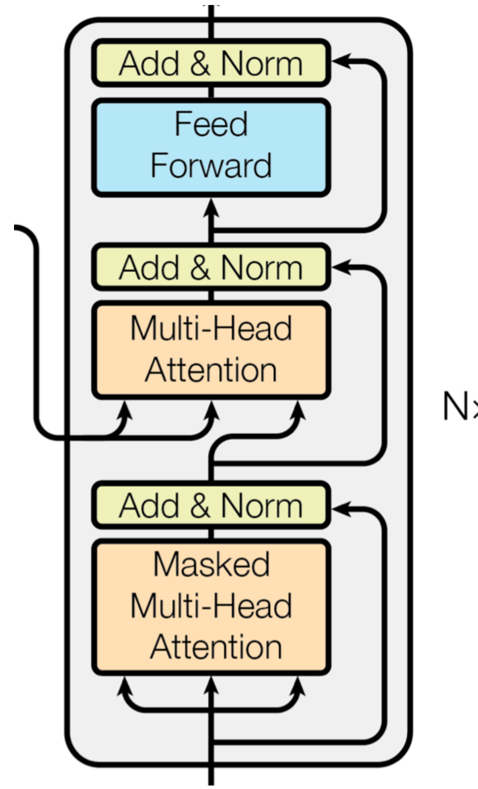
(the input $\mathbf{x}$ is added to the output of the sublayer)

... followed by a **normalization step**

(using the mean and standard deviation of its activations)

$$\text{LayerNorm}\big(\mathbf{x} + \text{Sublayer}(\mathbf{x})\big)$$

Decoder



A stack of N=6 identical layers
All layers and sublayers are 512-dimensional

Each layer consists of **three** sublayers
— one **masked multi-head self attention layer**
over **decoder** output
(masked, i.e. ignoring future tokens)
— one **multi-headed attention layer**
over **encoder** output
— one **position-wise feed forward layer**

Each sublayer has a residual connection
and is normalized: $\text{LayerNorm}\big(\mathbf{x} + \text{Sublayer}(\mathbf{x})\big)$

Multi-head attention

Just like we use **multiple filters (channels)** in CNNs, we can use **multiple attention heads** that each have their own sets of key/value/query matrices.

Multi-Head attention

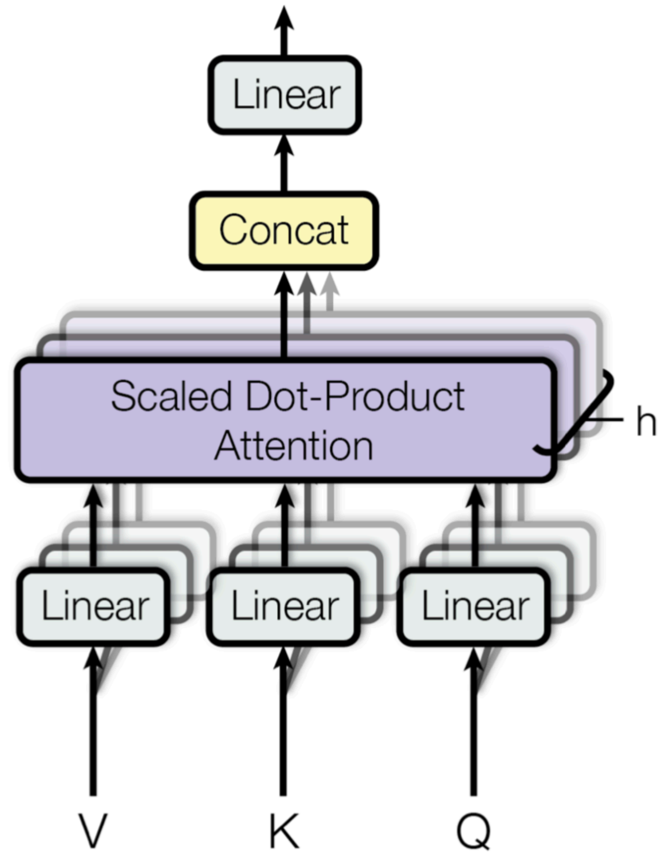
— Learn *h* different linear projections of *Q*, *K*, *V*

— Compute attention separately on each of these *h* versions

— Concatenate the resultant vectors

— Project this concatenated vector back down to a lower dimensionality with a weight matrix **W**

— Each attention head can use relatively low dimensionality



$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_\text{h})\mathbf{W}$$

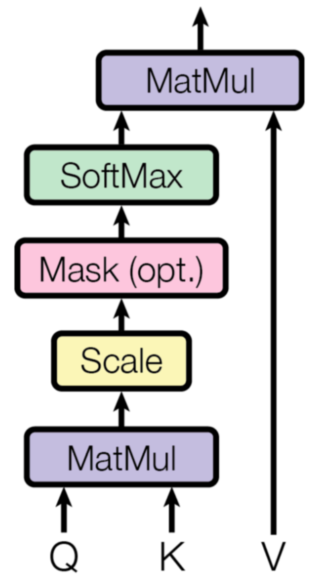
Scaling attention weights

Value of dot product grows with vector dimension *k*

To scale back the dot product, divide the weights by $\sqrt{k}$ before normalization:

$$w_j^{(i)} = \frac{\exp\left(\mathbf{q}^{(i)}\mathbf{k}^{(j)}\right)/\sqrt{k}}{\sum_j \left(\exp\left(\mathbf{q}^{(i)}\mathbf{k}^{(j)}\right)/\sqrt{k}\right)}$$

Scaled Dot-Product Attention



Position-wise feedforward nets

Each layer in the encoder and decoder contains
a feedforward sublayer FFN(**x**) that consists of…

… **one fully connected layer with a ReLU** activation
(that projects the 512 elements to 2048 dimensions),

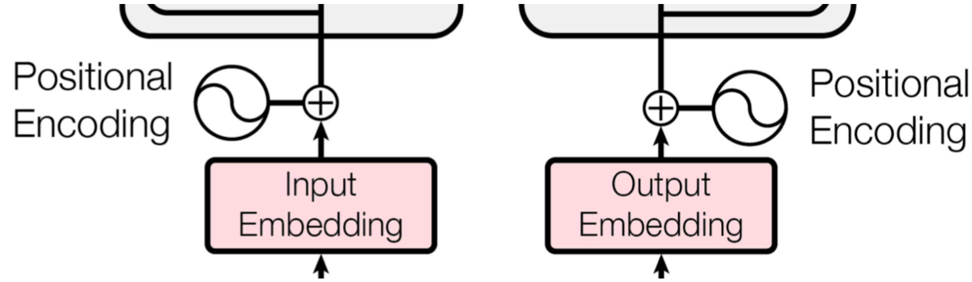
… followed by **another fully connected layer**
(that projects these 2048 elements back down to 512 dimensions)

$$\text{FFN}(\mathbf{x}) = \max\left(0, \mathbf{x}\mathbf{W}_1 + b_1\right) + \mathbf{W}_2 + b_2$$

Here **x** is the vector representation of the current position.
This is similar to 1x1 convolutions in a CNN.

Positional Encoding

How does this model capture sequence order?



Positional encodings have the same dimensionality as word embeddings (512) and are added in.

Each dimension *i* is a sinusoid whose frequency depends on *i*, evaluated at position *j*
(sinusoid = a sine or cosine function with a different frequency)

$$\mathrm{PE}_{(j,2i)} = \sin\left(\frac{j}{10000^{2i/d}}\right) \qquad \mathrm{PE}_{(j,2i+1)} = \cos\left(\frac{j}{10000^{2i/d}}\right)$$