Attention & Transformers

CMSC 473/673 - NATURAL LANGUAGE PROCESSING

Slides modified from Dr. Daphne Ippolito

Learning Objectives

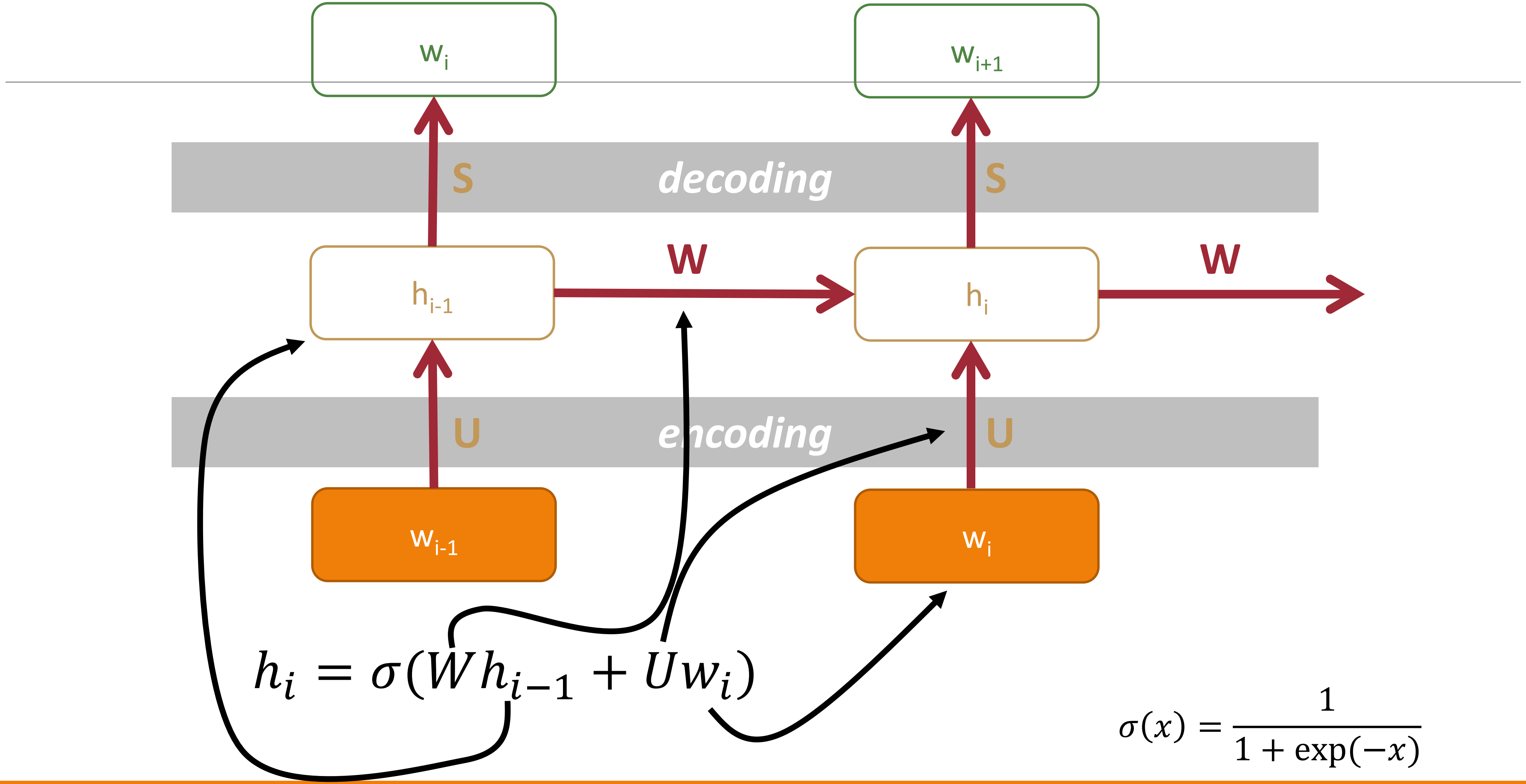
Review RNN architecture with an end-to-end diagram

Compare sequence-to-sequence RNNs to simple NNs & non-neural LMs

Compare sequence-to-sequence RNNs to transformers

Compare and contrast all LM types so far

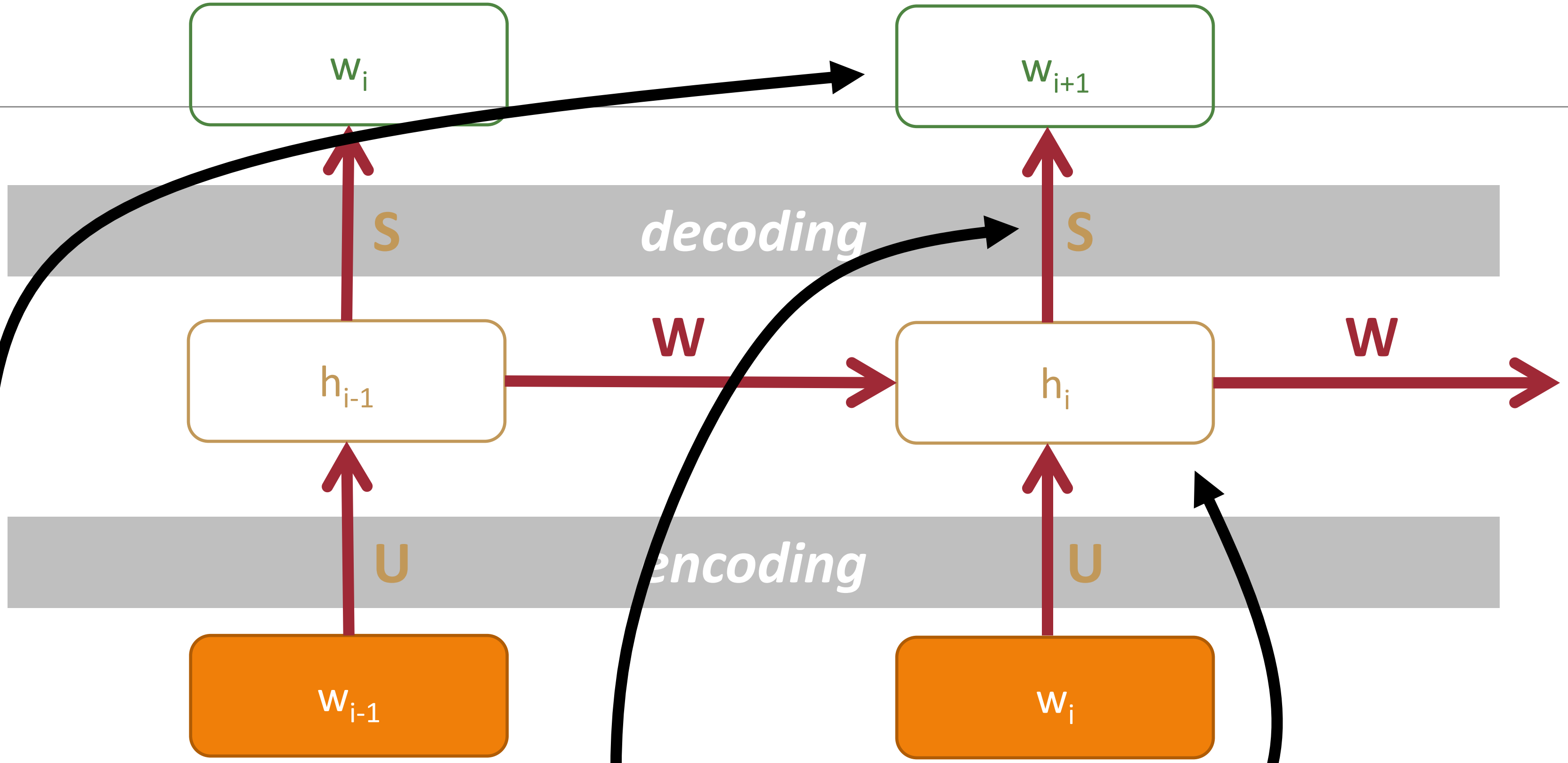
Review:
A *Simple* Recurrent Neural Network Cell



$$h_i = \sigma(W h_{i-1} + U w_i)$$

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

Review:
A *Simple* Recurrent Neural Network Cell

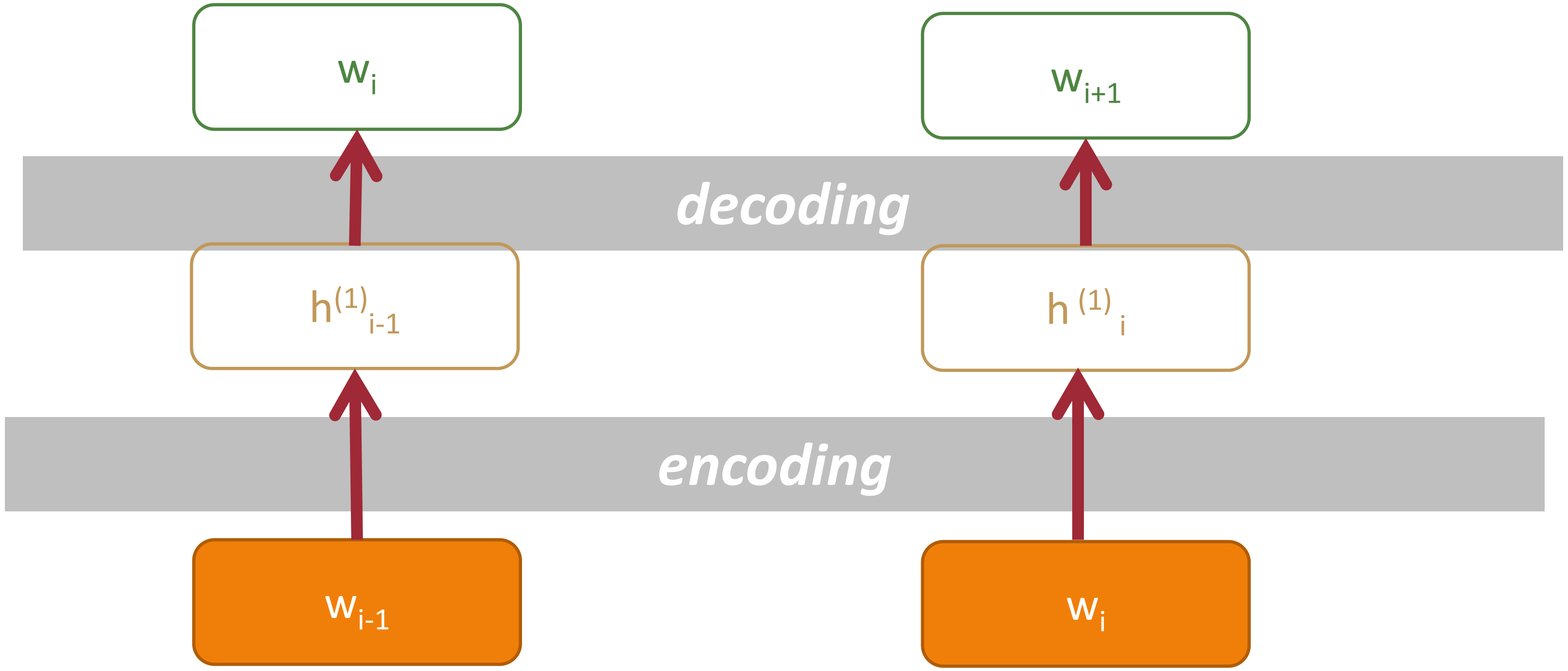


$$h_i = \sigma(W h_{i-1} + U w_i)$$

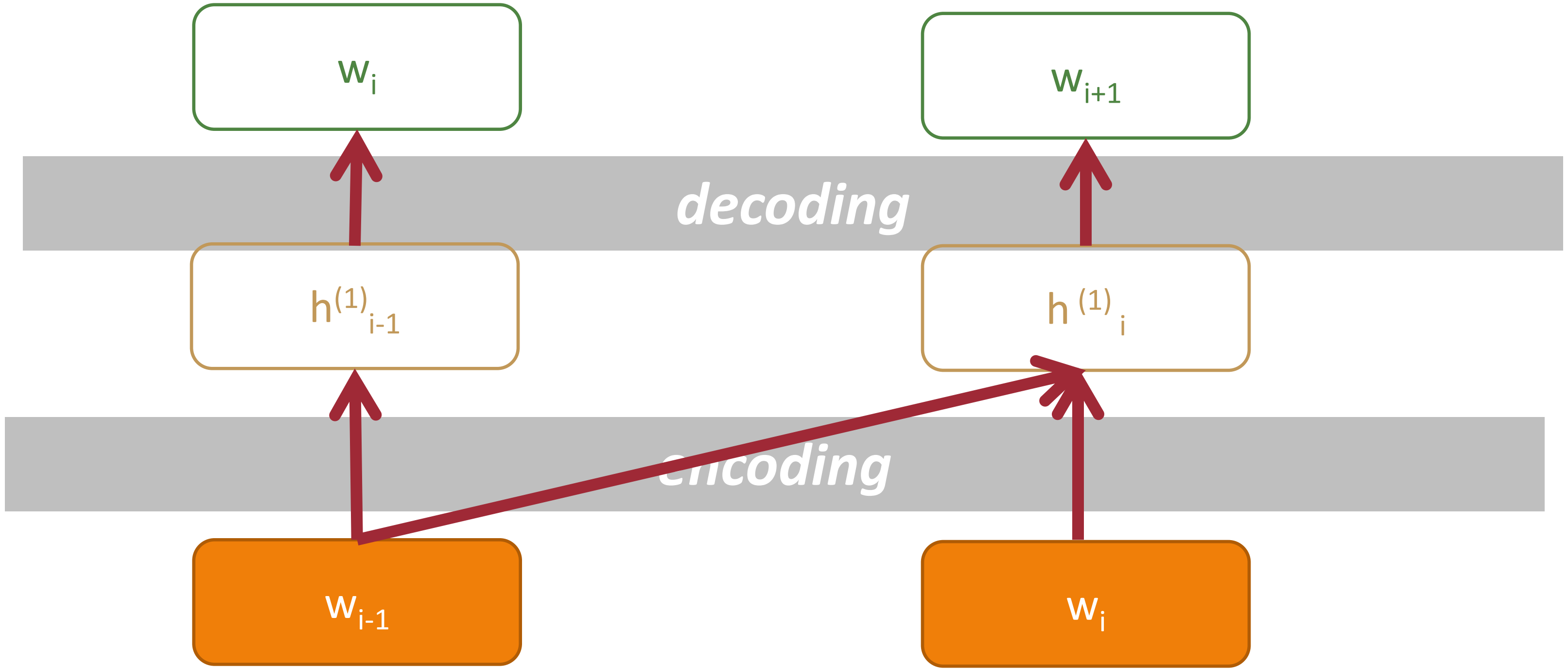
$$\hat{w}_{i+1} = \text{softmax}(S h_i)$$

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

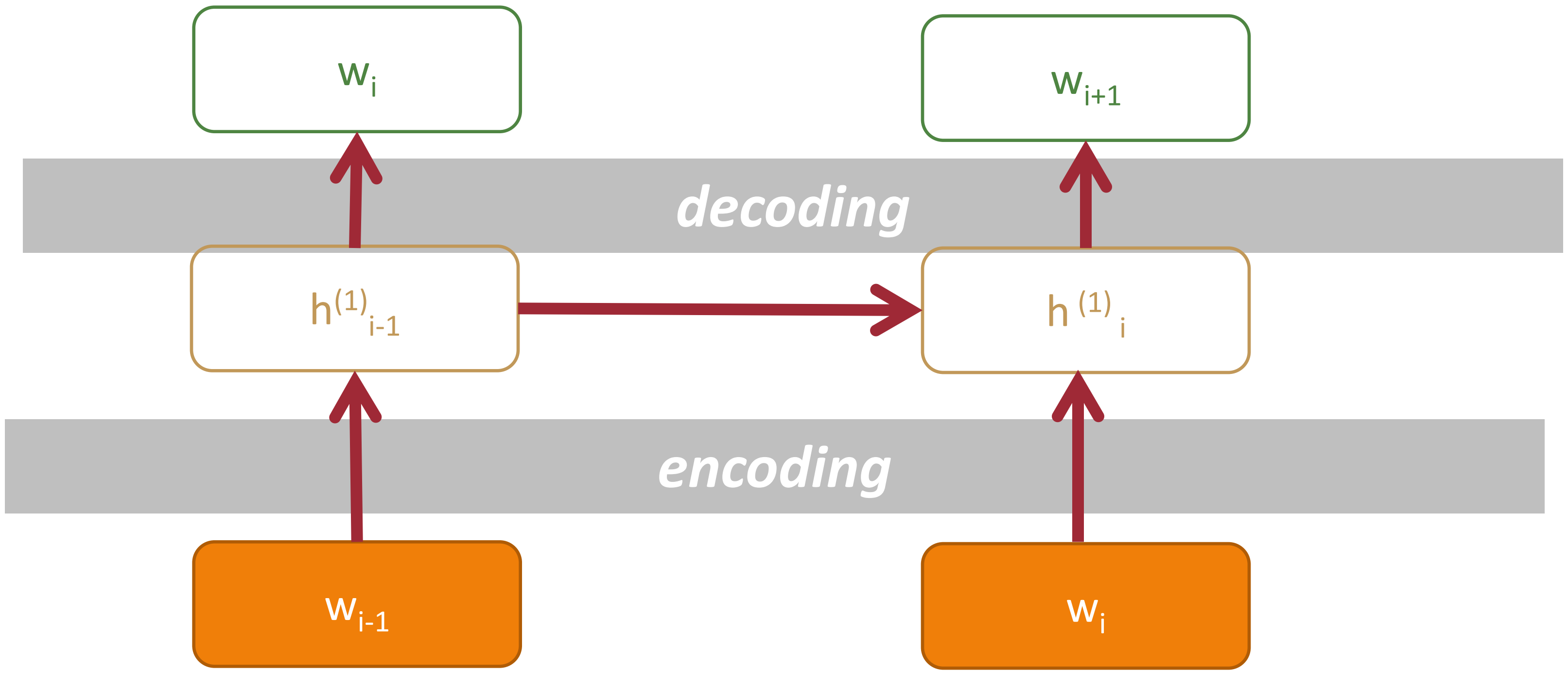
Feedforward Network



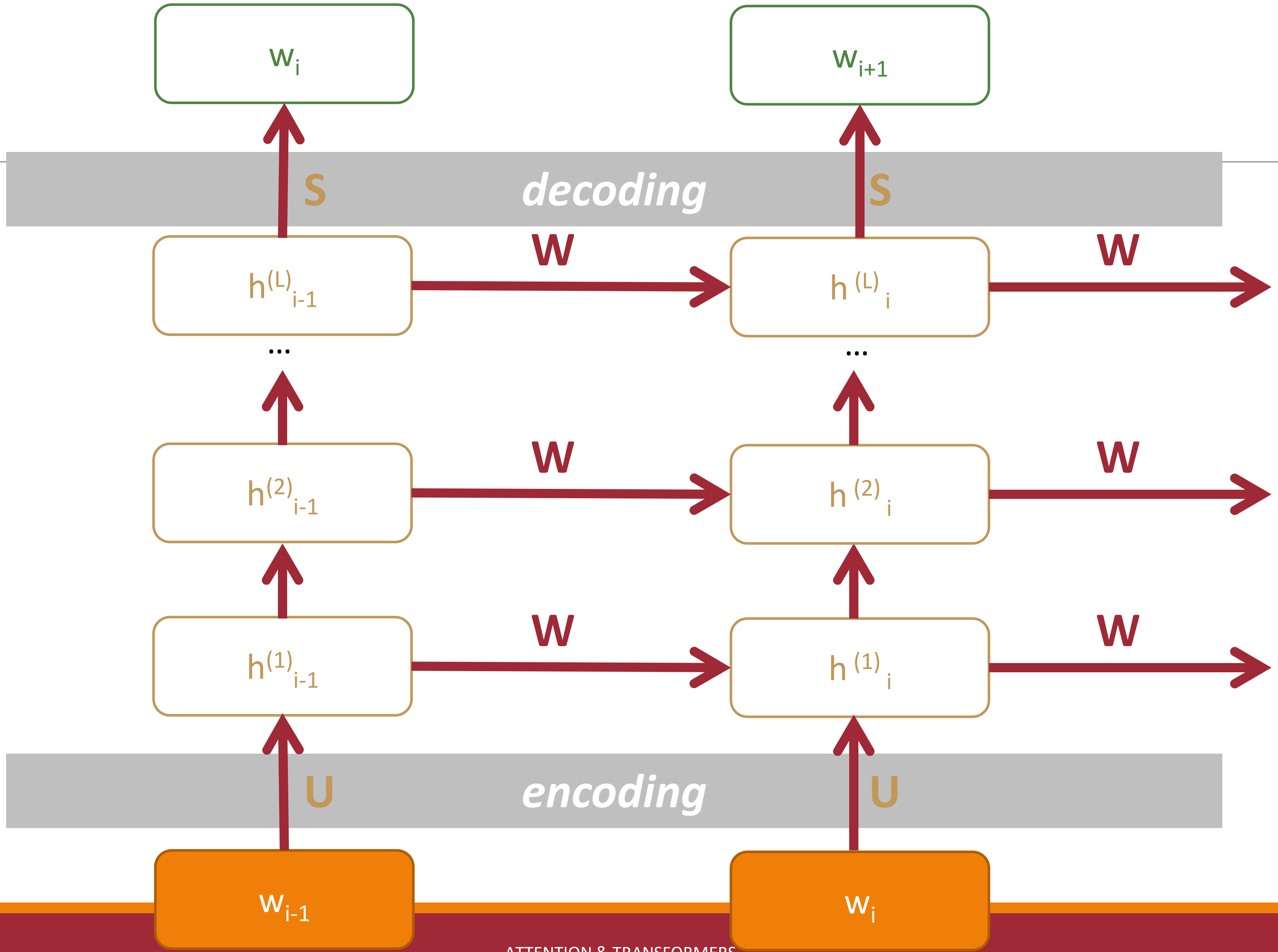
Tri-gram Feedforward Neural Network



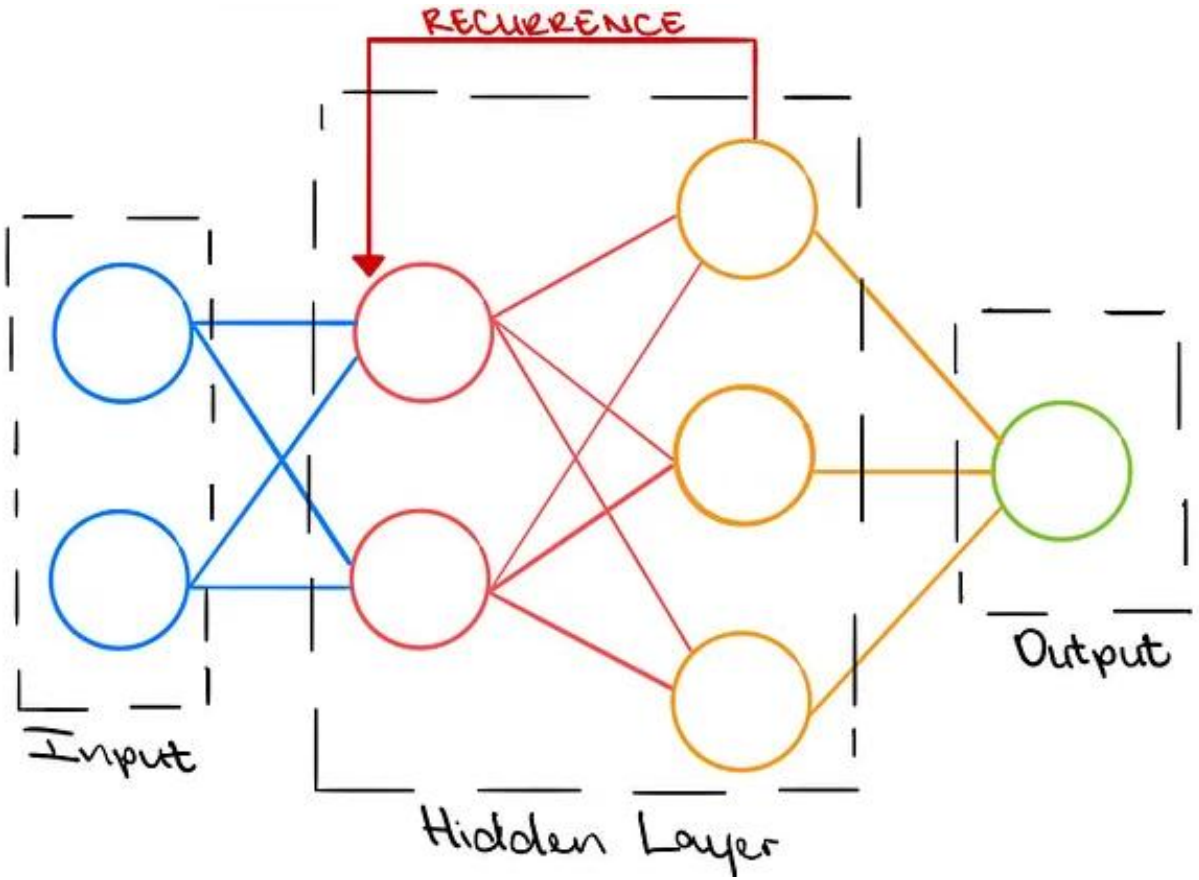
Recurrent Neural Network



Review: A Multi-Layer *Simple* Recurrent Neural Network Cell



Another way of illustrating it



https://towardsdatascience.com/introducing-recurrent-neural-networks-f359653d7020

Review:
Defining A Simple RNN in Python

http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

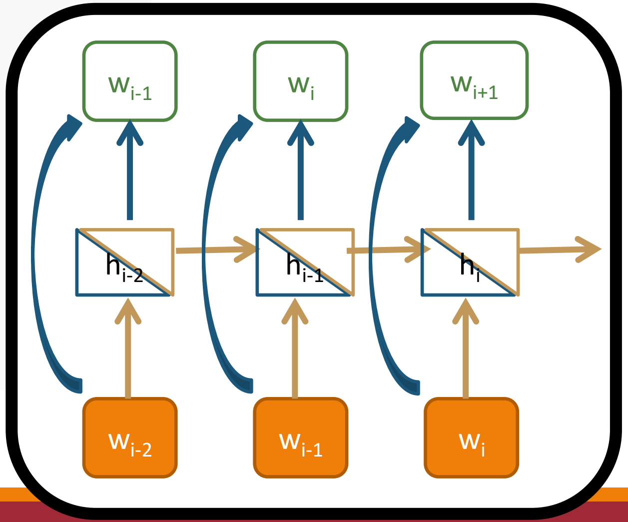
```
import torch.nn as nn
import torch.nn.functional as F

class CharRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharRNN, self).__init__()

        self.rnn = nn.RNN(input_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, line_tensor):
        rnn_out, hidden = self.rnn(line_tensor)
        output = self.h2o(hidden[0])
        output = self.softmax(output)

        return output
```



Review:
Training A Simple RNN in Python

http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

```
for iter in range(1, n_epoch + 1):
    rnn.zero_grad() # clear the gradients

    # create some minibatches
    # we cannot use dataloaders because each of our names is a different length
    batches = list(range(len(training_data)))
    random.shuffle(batches)
    batches = np.array_split(batches, len(batches) //n_batch_size )

    for idx, batch in enumerate(batches):
        batch_loss = 0
        for i in batch: #for each example in this batch
            (label_tensor, text_tensor, label, text) = training_data[i]
            output = rnn.forward(text_tensor)
            loss = criterion(output, label_tensor)
            batch_loss += loss

        # optimize parameters
        batch_loss.backward()
        nn.utils.clip_grad_norm_(rnn.parameters(), 3)
        optimizer.step()
        optimizer.zero_grad()

        current_loss += batch_loss.item() / len(batch)

    all_losses.append(current_loss / len(batches) )
    if iter % report_every == 0:
        print(f"{iter} ({iter / n_epoch:.0%}): \t average batch loss = {all_losses[-1]}")
    current_loss = 0

return all_losses
```

get predictions

eval predictions

compute gradient

perform SGD

Set t = 0
Pick a starting value θ_t
Until converged:
  for example(s) sentence i:
    1. Compute loss l on x_i
    2. Get gradient $g_t = l'(x_i)$
    3. Get scaling factor ρ_t
    4. Set $\theta_{t+1} = \theta_t - \rho_t * g_t$
    5. Set t += 1

Sequence-to-Sequence RNNs

Up until 2017 or so, neural language models were mostly built using recurrent neural networks.

Sequence to Sequence Learning with Neural Networks

Ilya Sutskever Google ilyasu@google.com

Oriol Vinyals Google vinyals@google.com

Quoc V. Le Google qvl@google.com

Abstract

Deep Neural Networks (DNNs) are powerful models that have achieved excellent performance on difficult learning tasks. Although DNNs work well whenever large labeled training sets are available, they cannot be used to map sequences to sequences. In this paper, we present a general end-to-end approach to sequence learning that makes minimal assumptions on the sequence structure. Our method uses a multilayered Long Short-Term Memory (LSTM) to map the input sequence to a vector of a fixed dimensionality, and then another deep LSTM to decode the target sequence from the vector. Our main result is that on an English to French translation task from the WMT'14 dataset, the translations produced by the LSTM achieve a BLEU score of 34.8 on the entire test set, where the LSTM's BLEU score was penalized on out-of-vocabulary words. Additionally, the LSTM did not have difficulty on long sentences. For comparison, a phrase-based SMT system achieves a BLEU score of 33.3 on the same dataset. When we used the LSTM to rerank the 1000 hypotheses produced by the aforementioned SMT system, its BLEU score increases to 36.5, which is close to the previous best result on this task. The LSTM also learned sensible phrase and sentence representations that are sensitive to word order and are relatively invariant to the active and the passive voice. Finally, we found that reversing the order of the words in all source sentences (but not target sentences) improved the LSTM's performance markedly, because doing so introduced many short term dependencies between the source and the target sentence which made the optimization problem easier.

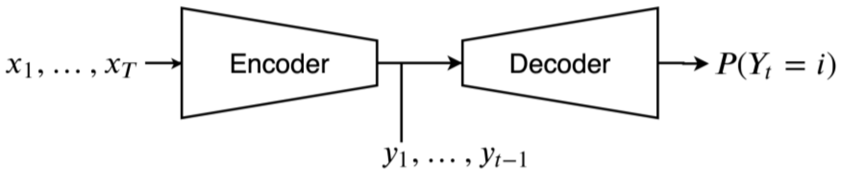
iv:1409.3215v3 [cs.CL] 14 Dec 2014

Generating Sequences With Recurrent Neural Networks

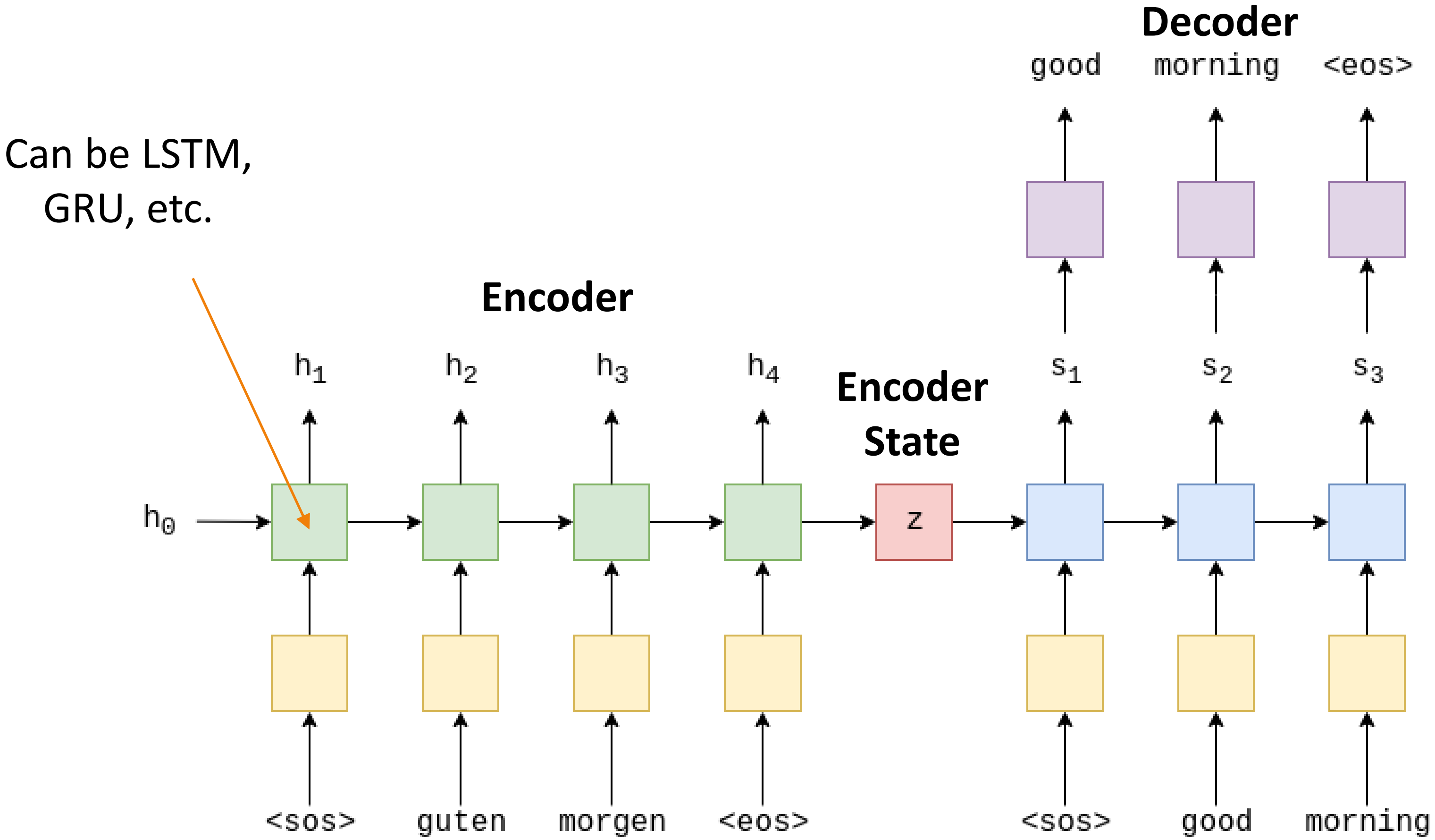
Alex Graves
Department of Computer Science
University of Toronto
graves@cs.toronto.edu

Abstract

This paper shows how Long Short-term Memory recurrent neural networks can be used to generate complex sequences with long-range structure, simply by predicting one data point at a time. The approach is demonstrated for text (where the data are discrete) and online handwriting (where the data are real-valued). It is then extended to handwriting synthesis by allowing the network to condition its predictions on a text sequence. The resulting system is able to generate highly realistic cursive handwriting in a wide variety of styles.

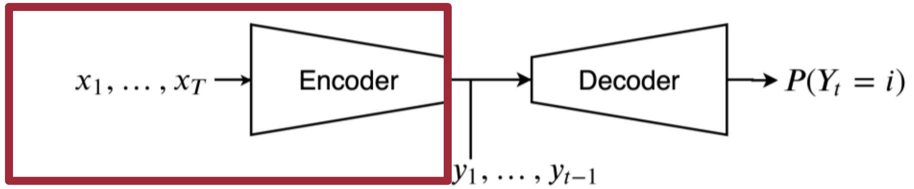


Review: Sequence-to-Sequence



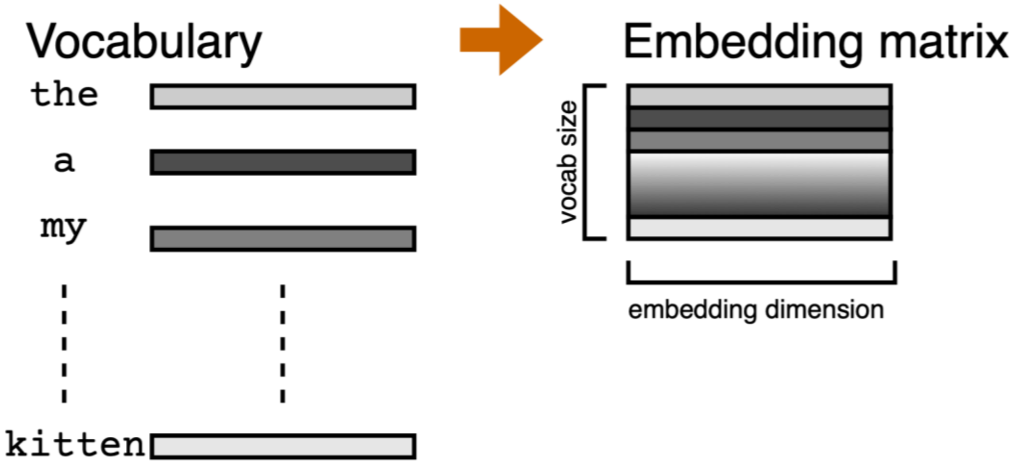
https://colab.research.google.com/github/bentrevett/pytorch-seq2seq/blob/main/1%20-%20Sequence%20to%20Sequence%20Learning%20with%20Neural%20Networks.ipynb#scrollTo=k6sRrL4wKsmi

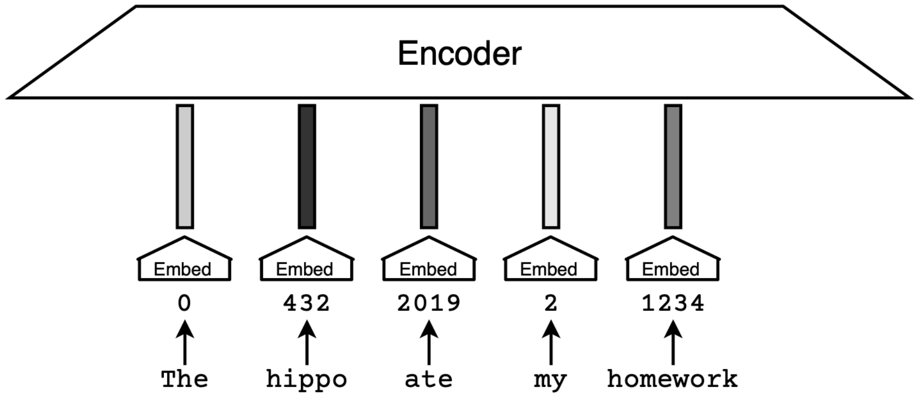
I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," in *Conference on Advances in Neural Information Processing Systems (NeurIPS)*, Montréal, Canada, 2014, pp. 3104–3112. https://proceedings.neurips.cc/paper_files/paper/2014/hash/a14ac55a4f27472c5d894ec1c3c743d2-Abstract.html

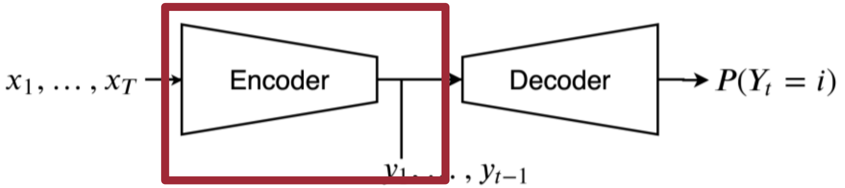


Inputs to the Encoder

The encoder takes as input the embeddings corresponding to each token in the sequence.

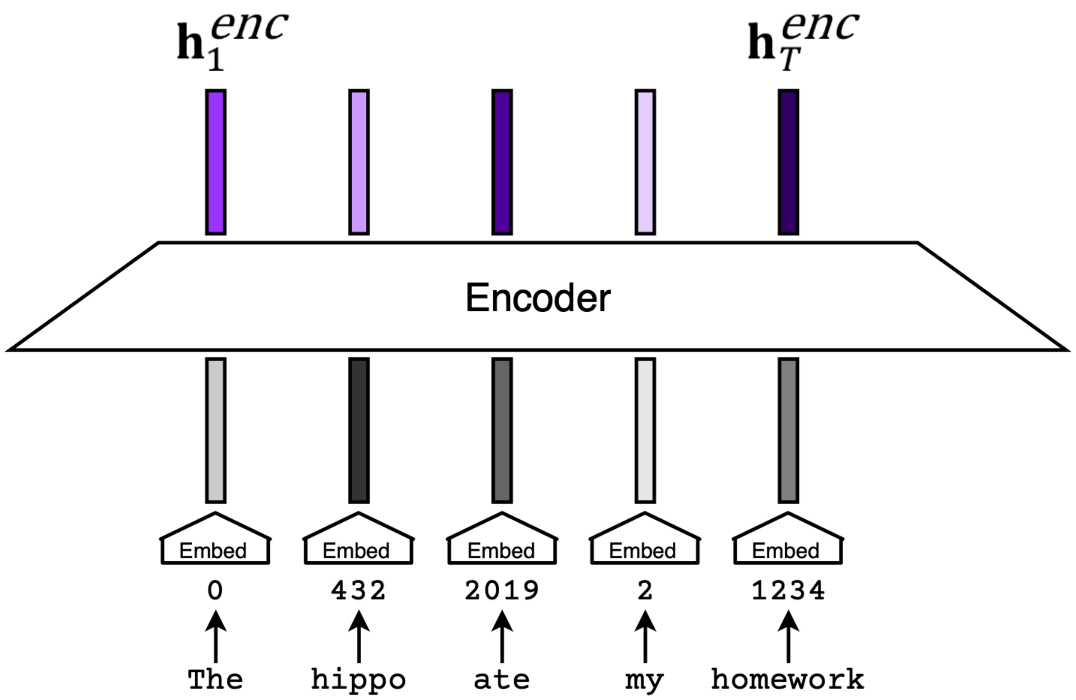


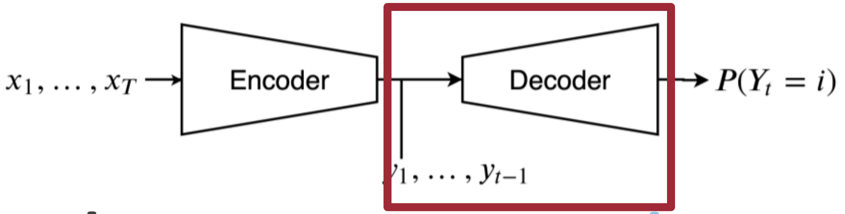




Outputs from the Encoder

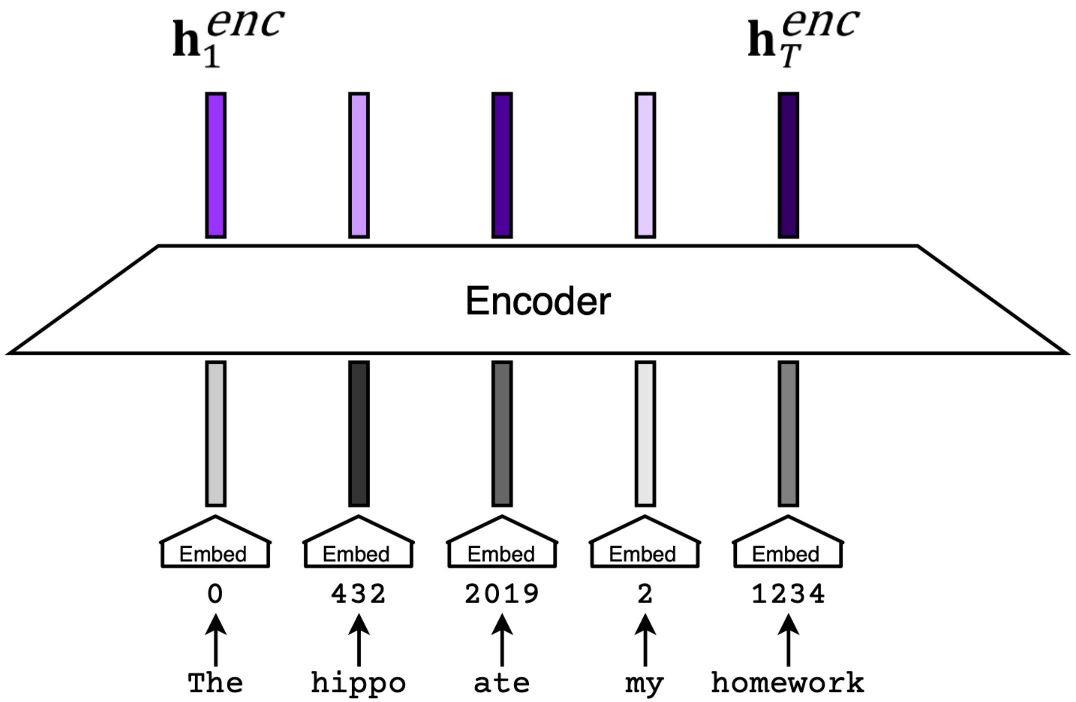
The encoder outputs a sequence of vectors. These are called the hidden state of the encoder.

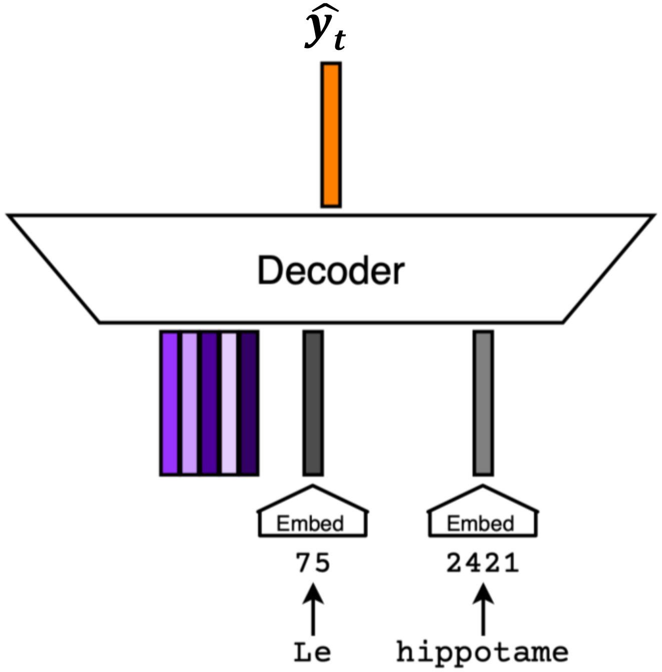


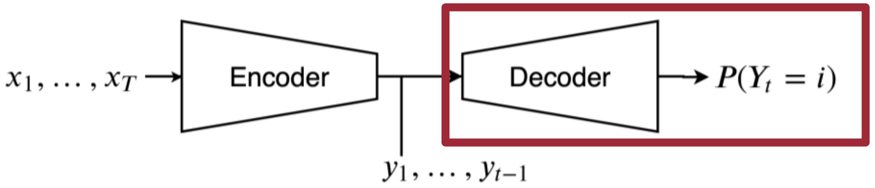


Inputs to the Decoder

The decoder takes as input the hidden states from the encoder as well as the embeddings for the tokens seen so far in the target sequence.

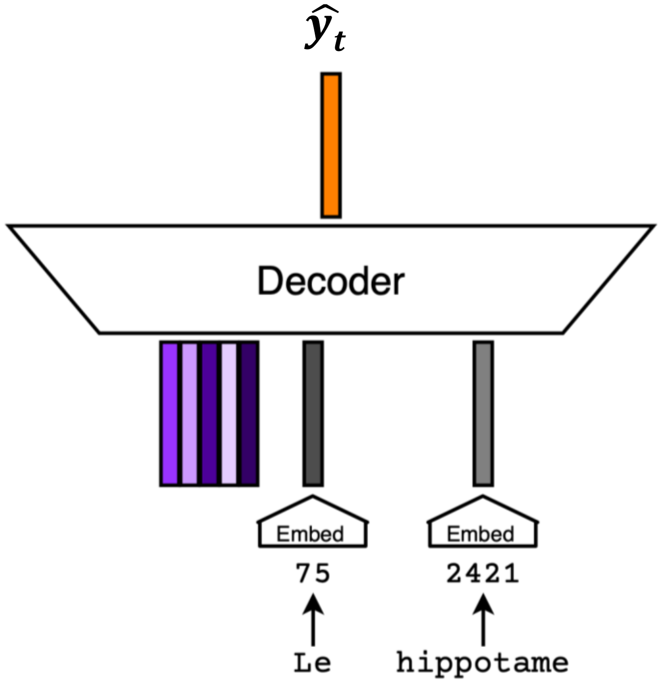






Outputs from the Decoder

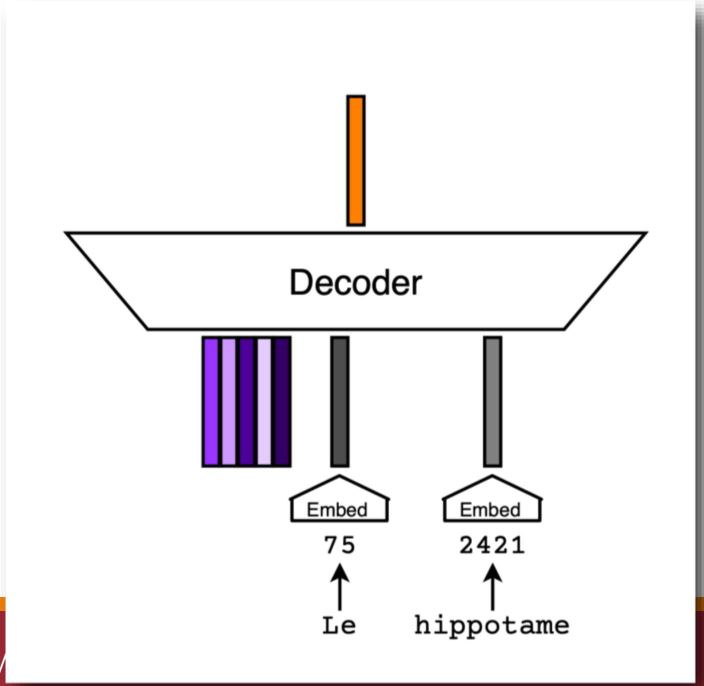
The decoder outputs an embedding $\widehat{\boldsymbol{yt}}$. The goal is for this embedding to be as close as possible to the embedding of the true next token.

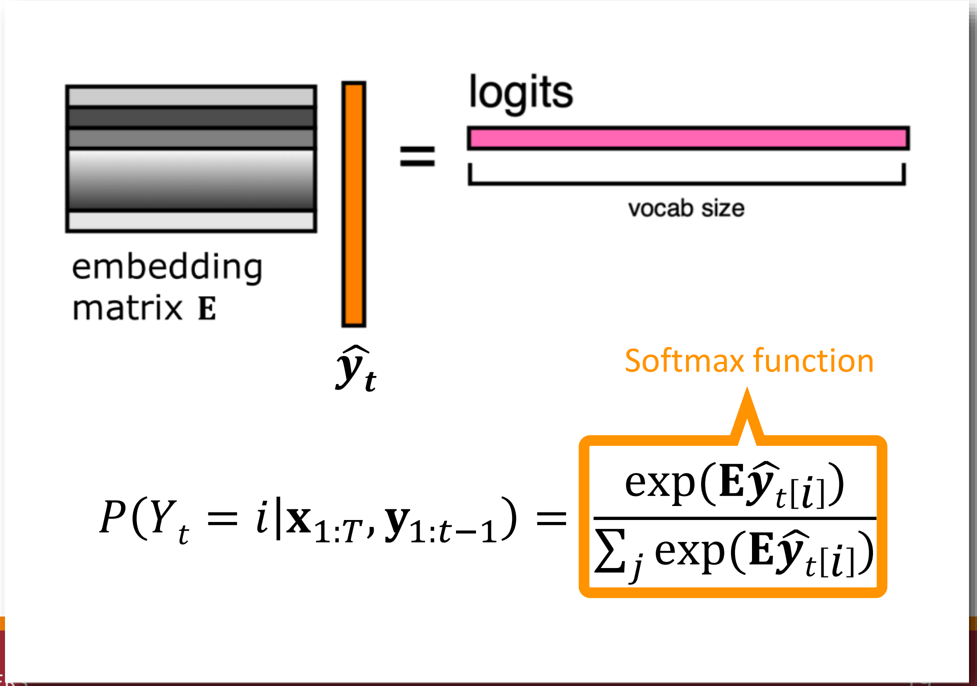


Turning $\widehat{yt}$ into a Probability Distribution

We can multiply the predicted embedding $\widehat{yt}$ by our vocabulary embedding matric to get a score for each vocabulary word. These scores are referred to as logits.

The **softmax function** then lets us turn the logits into probabilities.

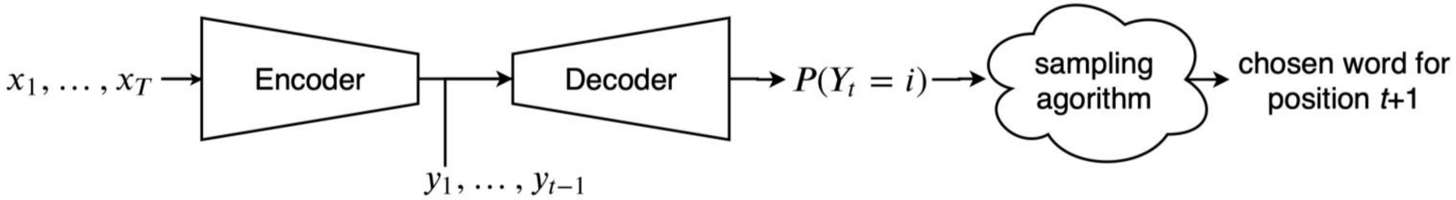




Generating Text

Also sometimes called decoding 🙄

To generate text, we need an algorithm that selects tokens given the predicted probability distributions.

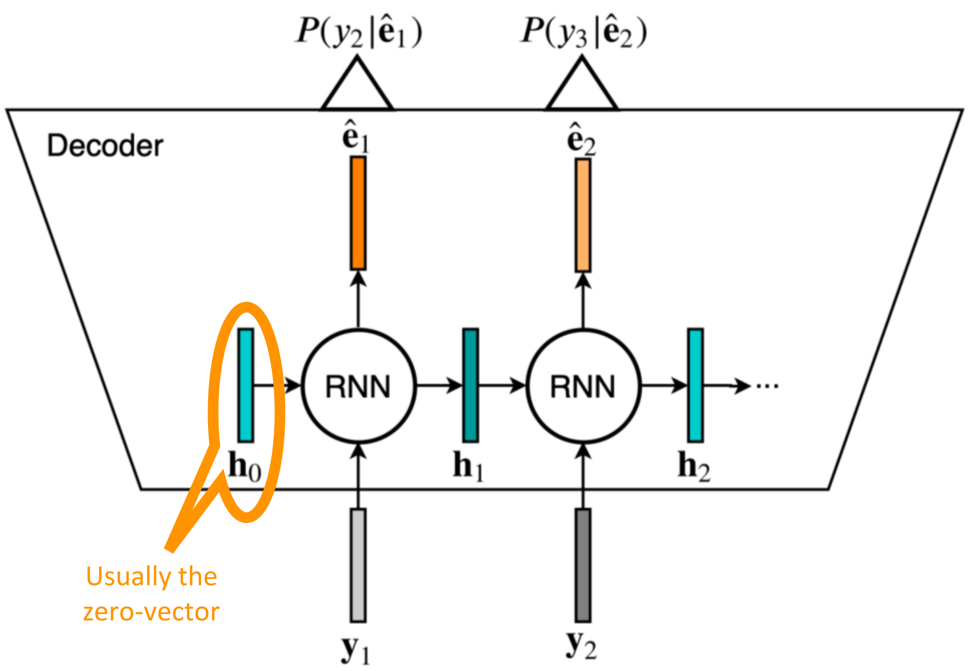


More on this in the next lecture!

$$h_i = \sigma(Wh_{i-1} + Uw_i)$$

RNNs - Single Layer Decoder

Same as we saw before



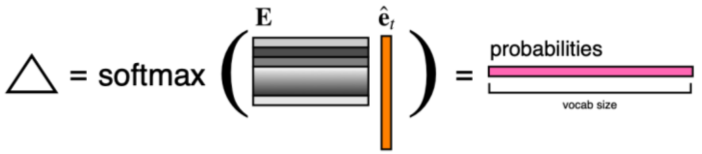
The current hidden state is computed as a function of the previous hidden state and the embedding of the current word in the target sequence.

$$\mathbf{h}_t = \text{RNN}(\mathbf{W}_{ih}\mathbf{y}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)$$

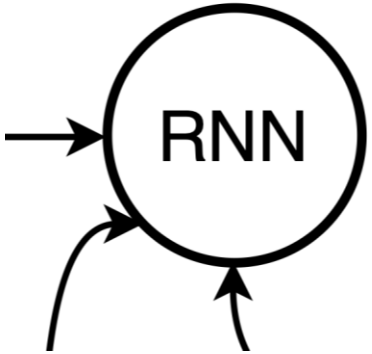
The current hidden state is used to predict an embedding for the next word in the target sequence.

$$\hat{\mathbf{e}}_t = \mathbf{b}_e + \mathbf{W}_{he}\mathbf{h}_t$$

This predicted embedding is used in the loss function:



What is the "RNN" unit?



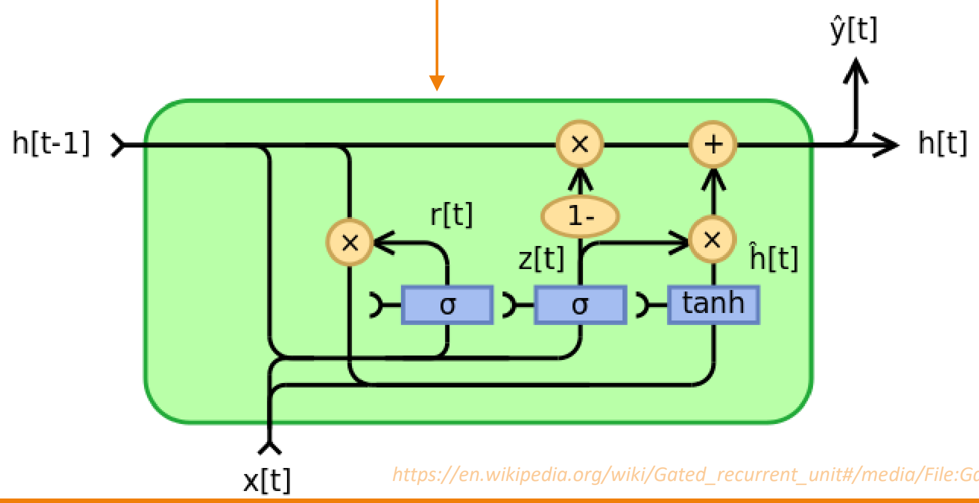
?

Review: LSTMs/GRUs

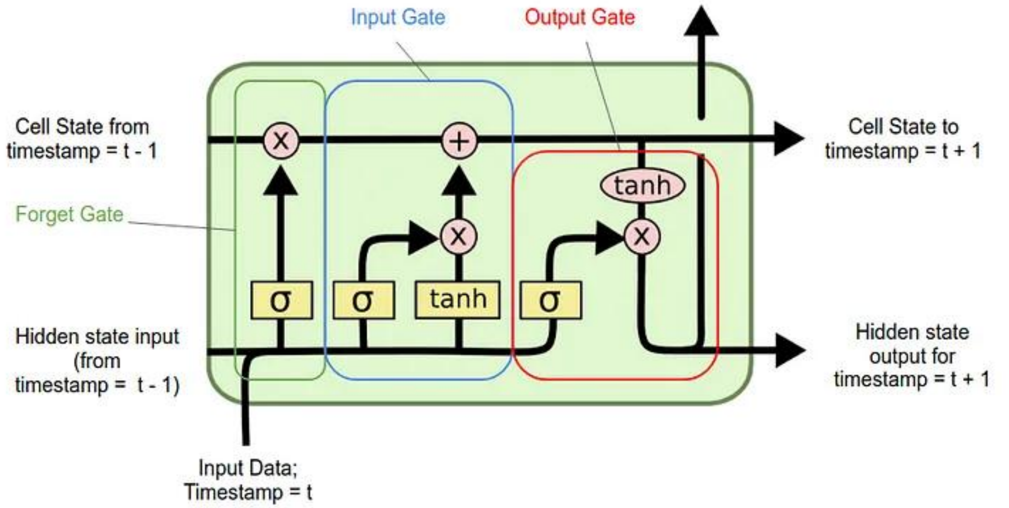
LSTM: Long Short-Term Memory (Hochreiter & Schmidhuber, 1997)

LSTMs were originally designed to keep around information for longer in the hidden state as it gets repeatedly updated.

GRU: Gated Recurrent Unit (Cho et al., 2014)

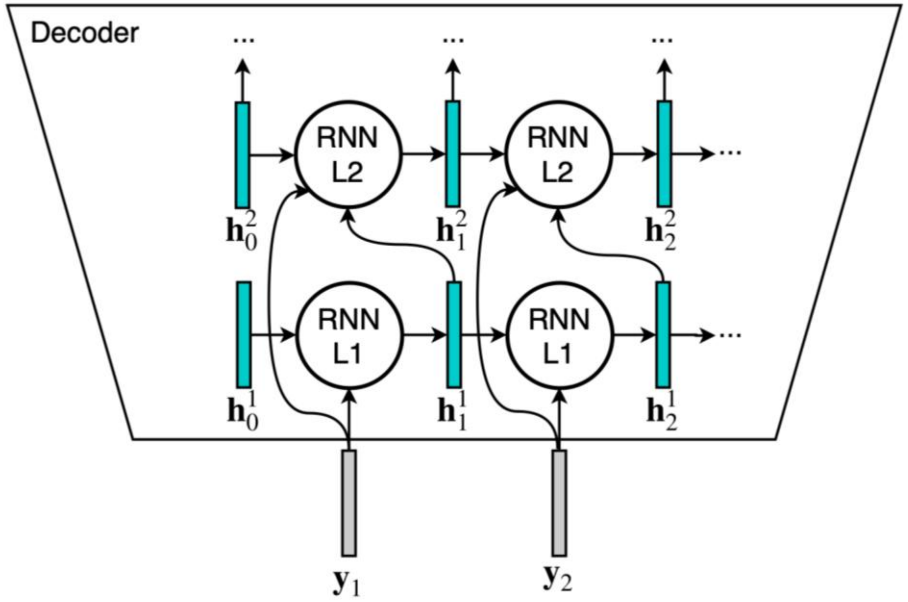


https://en.wikipedia.org/wiki/Gated_recurrent_unit#/media/File:Gated_Recurrent_Unit,_base_type.svg



https://medium.com/analytics-vidhya/lstms-explained-a-complete-technically-accurate-conceptual-guide-with-keras-2a650327e8f2

RNN Multi-Layer Decoder Architecture



Computing the next hidden state:

No history

For the first layer:

$$\mathbf{h}_t^1 = \text{RNN}(\mathbf{W}_{ih^1}\mathbf{y}_t + \mathbf{W}_{h^1 h^1}\mathbf{h}_{t-1}^1 + \mathbf{b}_h^1)$$

For subsequent layers:

$$\mathbf{h}_t^l = \text{RNN}(\mathbf{W}_{ih^l}\mathbf{y}_t + \mathbf{W}_{h^{l-1} h^l}\mathbf{h}_t^{l-1} + \mathbf{W}_{h^l h^l}\mathbf{h}_{t-1}^l + \mathbf{b}_h^l)$$

Predicting an embedding for the next token in the sequence:

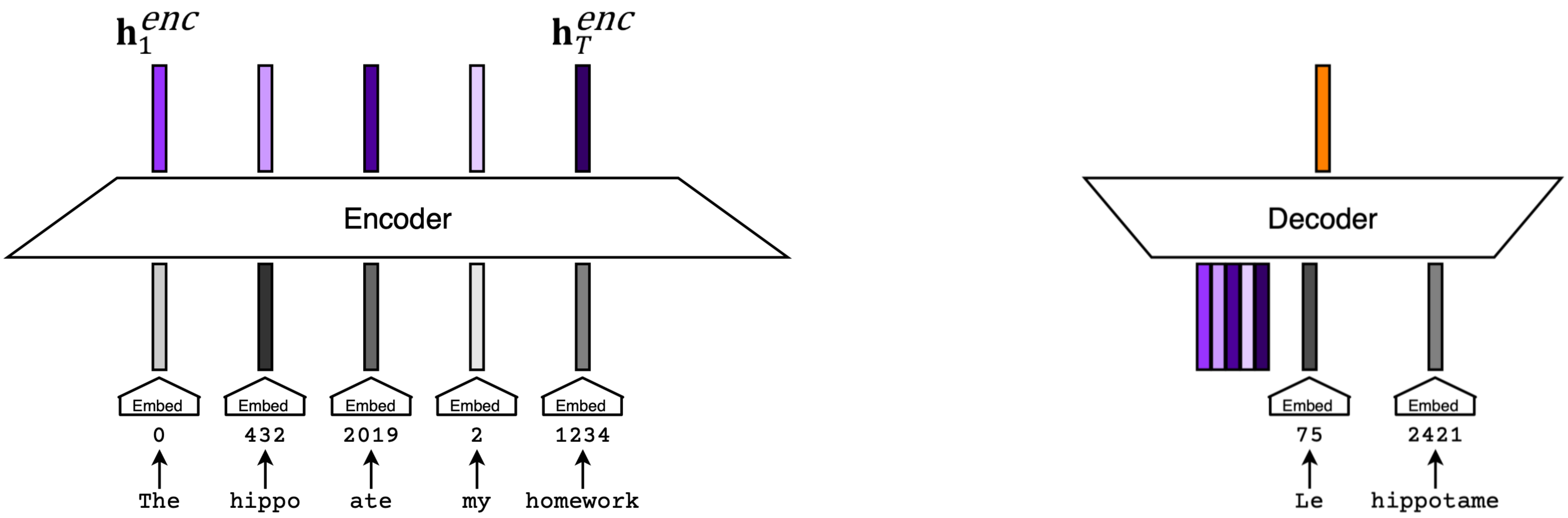
$$\hat{\mathbf{e}}t = \mathbf{b}_e + \sum_{l=1}^{L}\mathbf{W}_{h^l e}\mathbf{h}_t^l$$

Linear layer

Each of the b and W are learned bias and weight matrices.

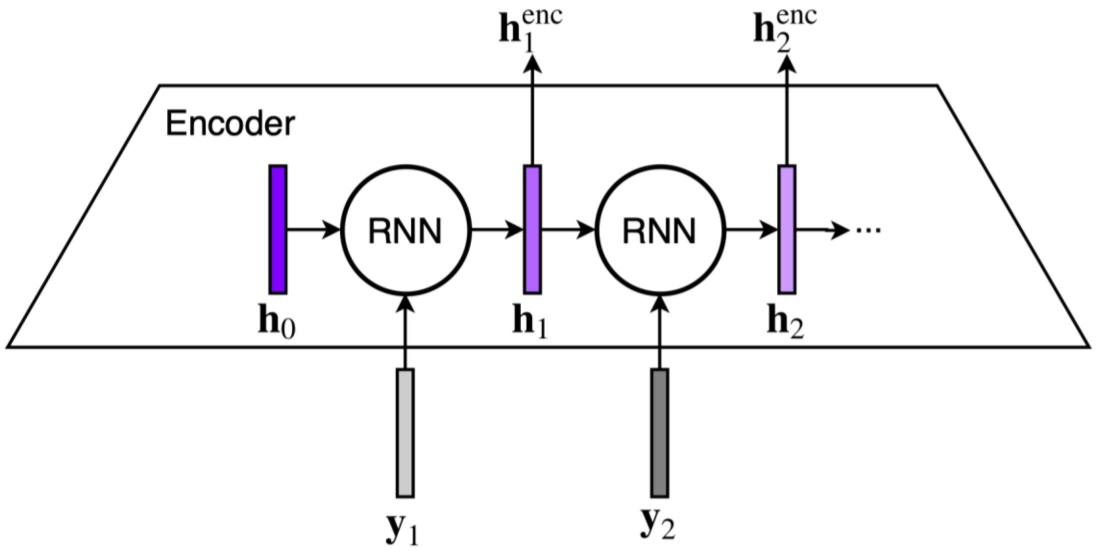
RNN Encoder-Decoder Architectures

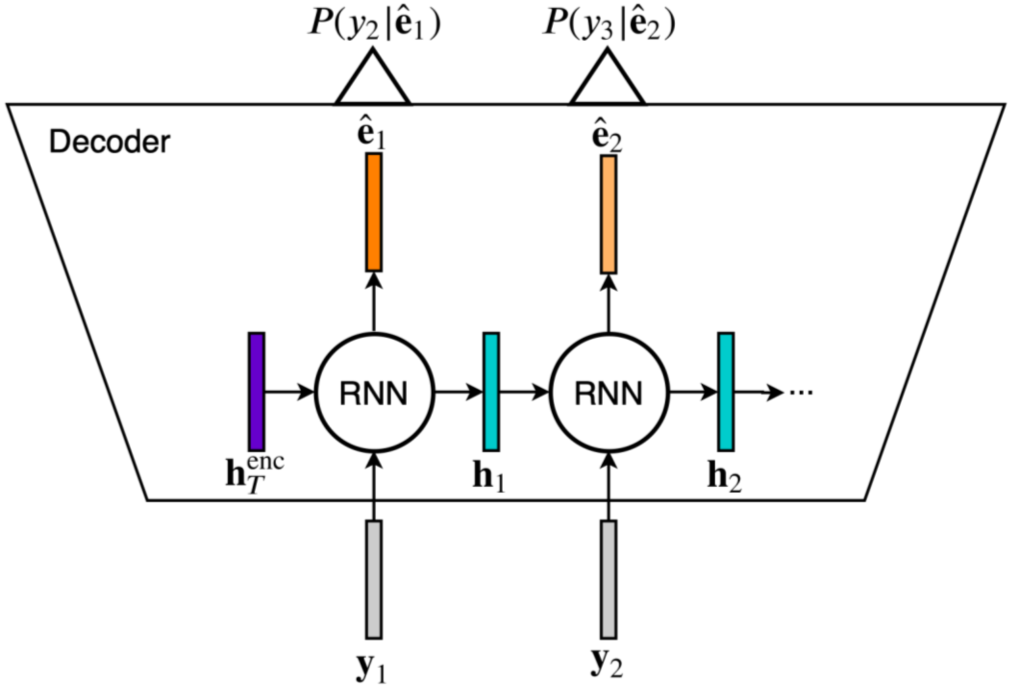
How do we implement an encoder-decoder model?



RNN Encoder-Decoder Architectures

Simplest approach: Use the final hidden state from the encoder to initialize the first hidden state of the decoder.





RNN Encoder-Decoder Architectures

[The, hippopotamus, …

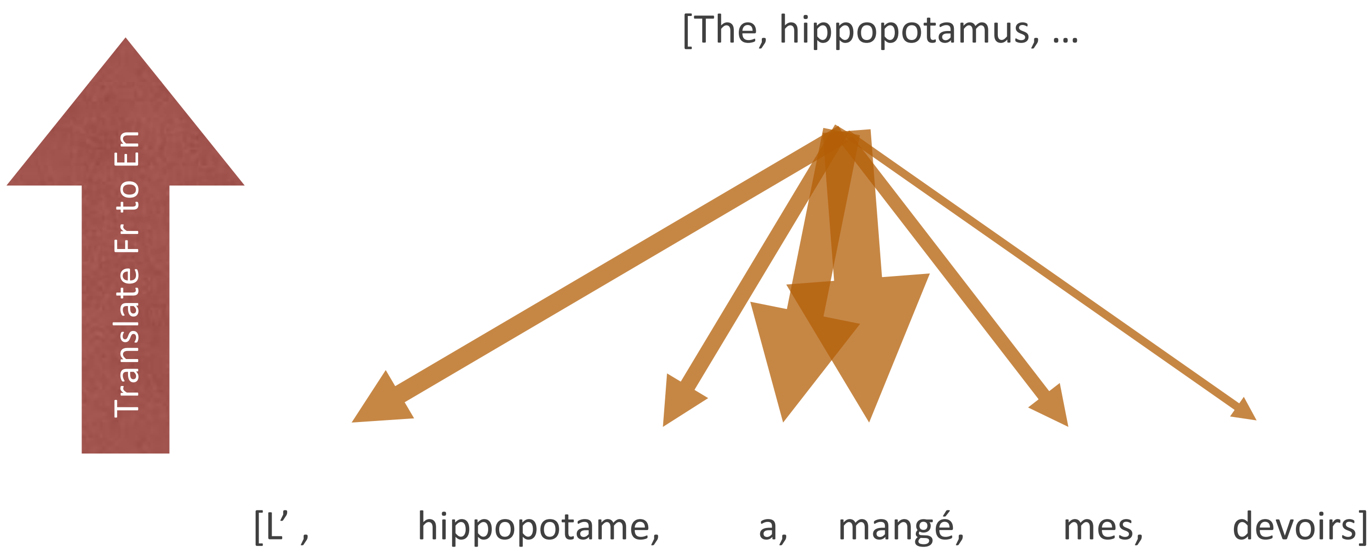
Translate Fr to En

When predicting the next English word, how much weight should the model put on each French word in the source sequence?

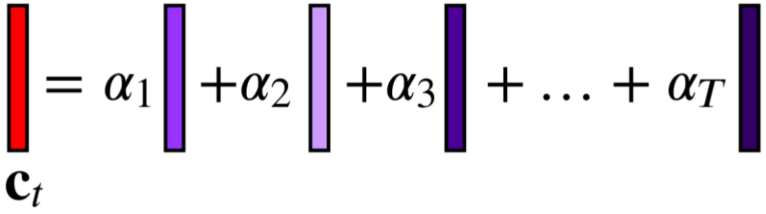
[L' , hippopotame, a, mangé, mes, devoirs]

Attention

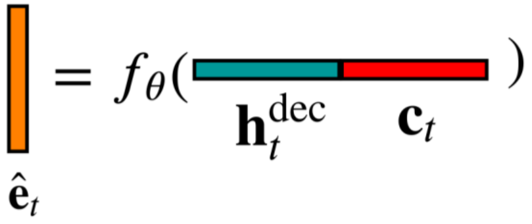
Better approach: an attention mechanism



Compute a linear combination of the encoder hidden states.



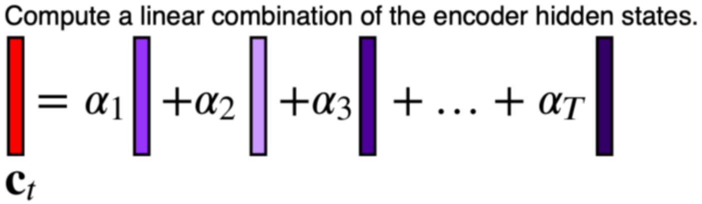
Decoder's prediction at position *t* is based on both the context vector and the hidden state outputted by the RNN at that position.

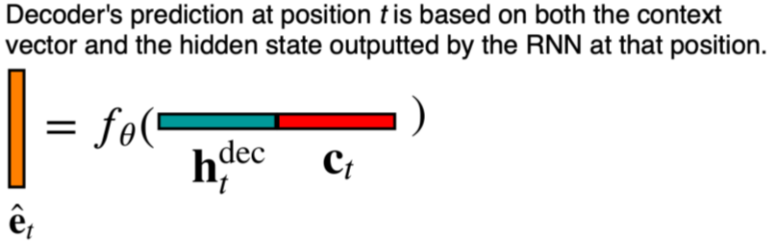


RNN Encoder-Decoder Architectures

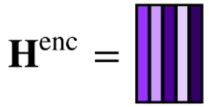
The tth context vector is computed as $\mathbf{c}t = \mathbf{H}^{\mathrm{enc}} at$

at[i] = softmax(att_score($\mathbf{h}_t^{\mathrm{dec}}$, $\mathbf{h}i^{\mathrm{enc}}$))



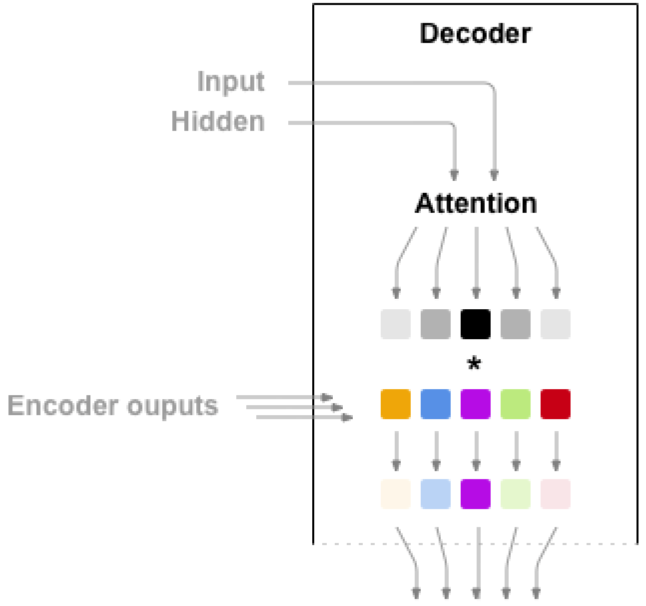


There are a few different options for the attention score:



$$\text{att\_score}(\mathbf{h}_t^{\mathrm{dec}}, \mathbf{h}_i^{\mathrm{enc}}) = \begin{cases} \mathbf{h}_t^{\mathrm{dec}} \cdot \mathbf{h}_{\boldsymbol{i}}^{\mathrm{enc}} & \text{dot product} \\ \mathbf{h}_t^{\mathrm{dec}}\, \mathbf{W}a\, \mathbf{h}_{\boldsymbol{i}}^{\mathrm{enc}} & \text{bilinear function} \\ w_{a1}^{\top} \tanh(\mathbf{W}a\mathbf{2}[\mathbf{h}_t^{\mathrm{dec}}, \mathbf{h}_{\boldsymbol{i}}^{\mathrm{enc}}]) & \text{MLP} \end{cases}$$

Attention




```
class BahdanauAttention(nn.Module):
    def __init__(self, hidden_size):
        super(BahdanauAttention, self).__init__()
        self.Wa = nn.Linear(hidden_size, hidden_size)
        self.Ua = nn.Linear(hidden_size, hidden_size)
        self.Va = nn.Linear(hidden_size, 1)

    def forward(self, query, keys):
        scores = self.Va(torch.tanh(self.Wa(query) + self.Ua(keys)))
        scores = scores.squeeze(2).unsqueeze(1)

        weights = F.softmax(scores, dim=-1)
        context = torch.bmm(weights, keys)

        return context, weights
```

https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html

Think-Pair-Share

What are some of the strengths of seq2seq models (compared to some of the earlier LMs we talked about)?

What are some of its weaknesses?

Limitations of Recurrent architecture

Slow to train.

- Can’t be easily parallelized.
- The computation at position t is dependent on first doing the computation at position t-1.

Difficult to access information from many steps back.

- If two tokens are K positions apart, there are K opportunities for knowledge of the first token to be erased from the hidden state before a prediction is made at the position of the second token.

Transformers

Since 2018, the field has rapidly standardized on the Transformer architecture

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

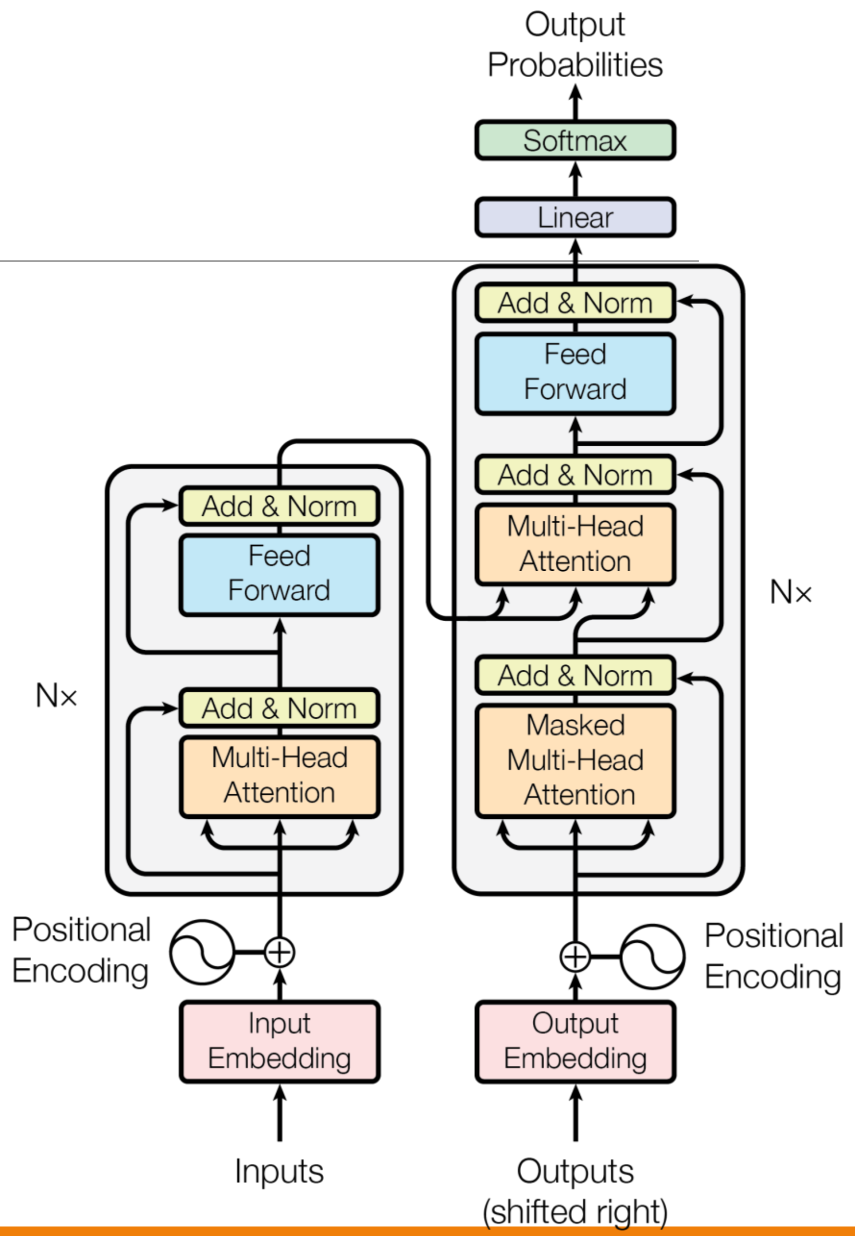
Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to

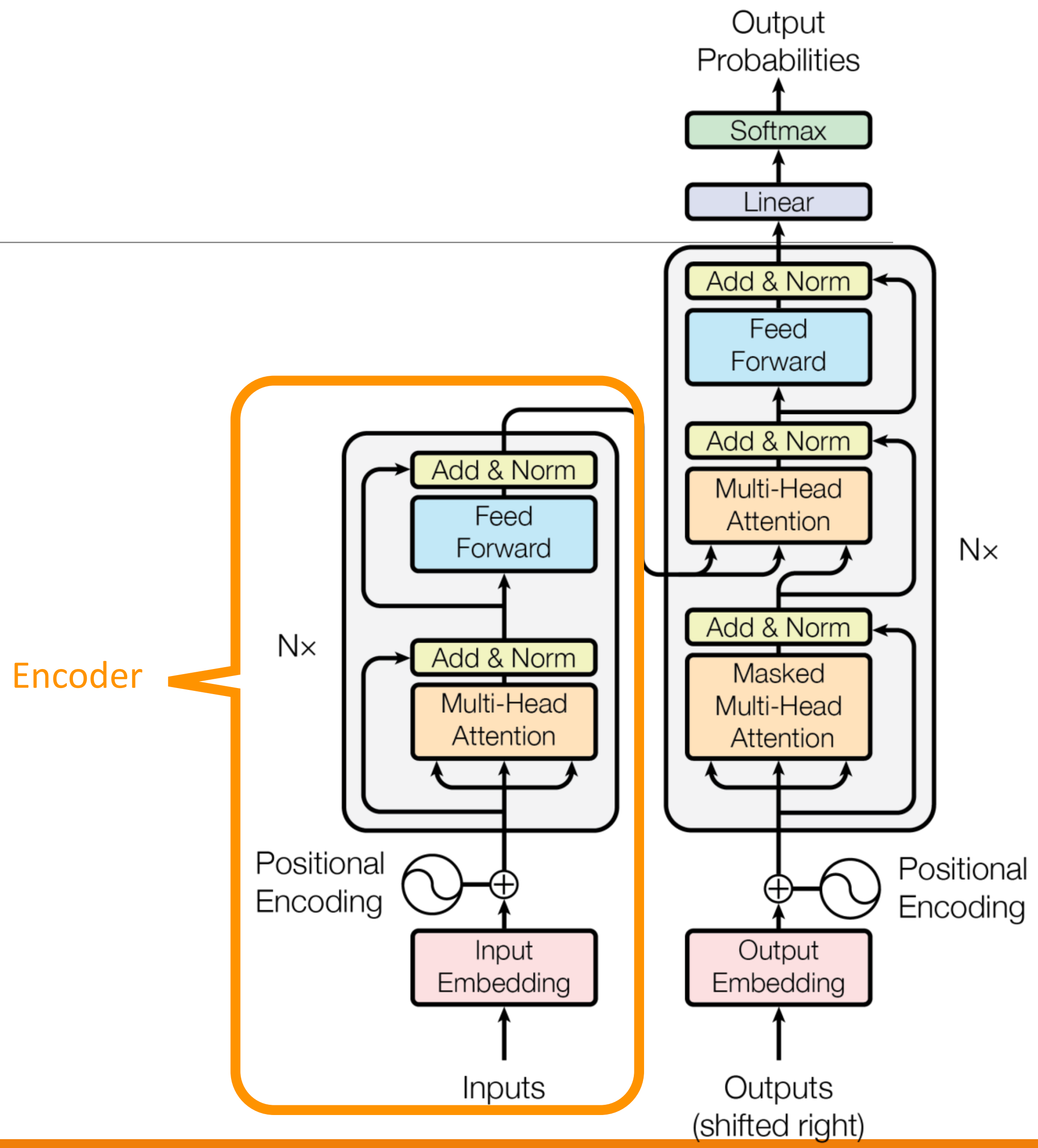
Transformers

The Transformer is a **non-recurrent** non-convolutional (feed-forward) neural network designed for language understanding

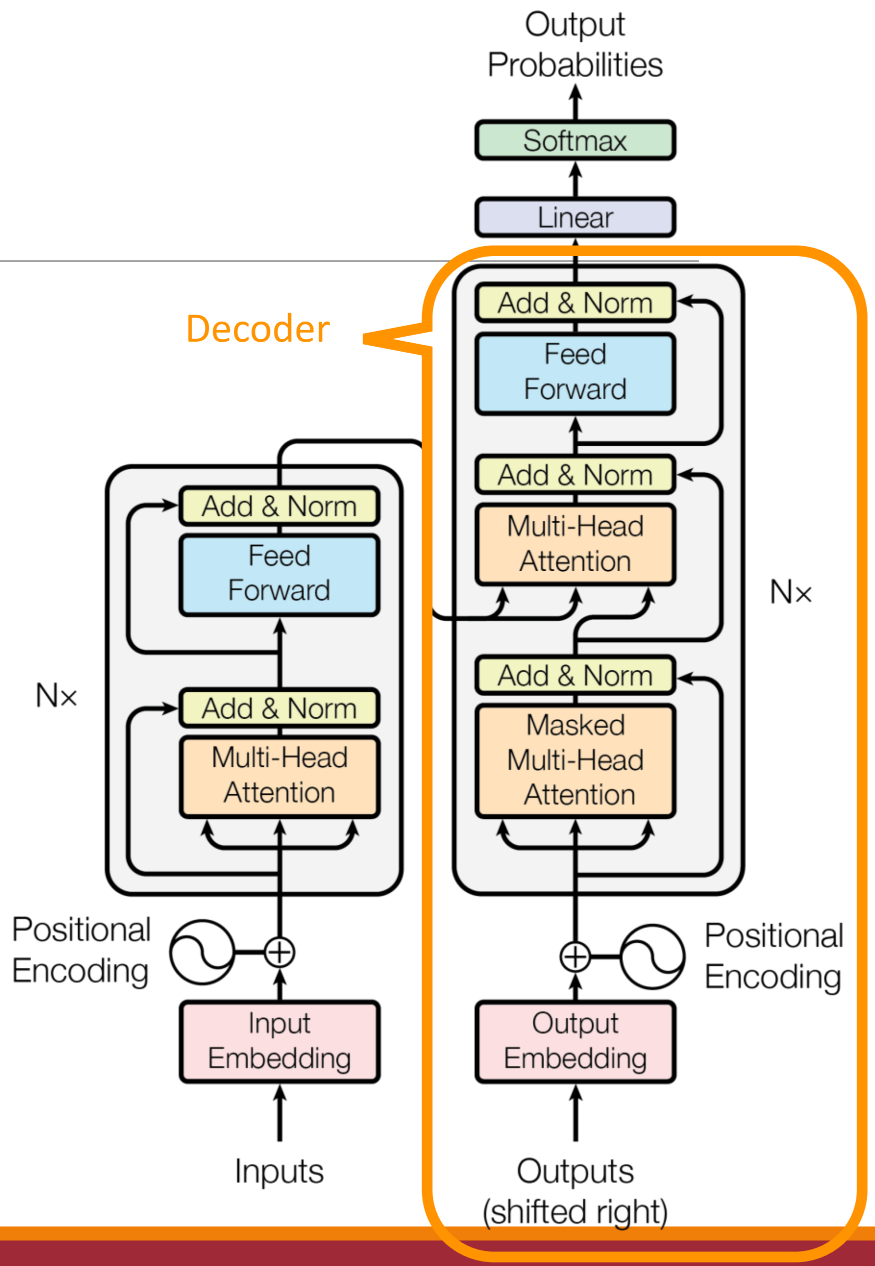
- introduces self-attention in addition to encoder-decoder attention



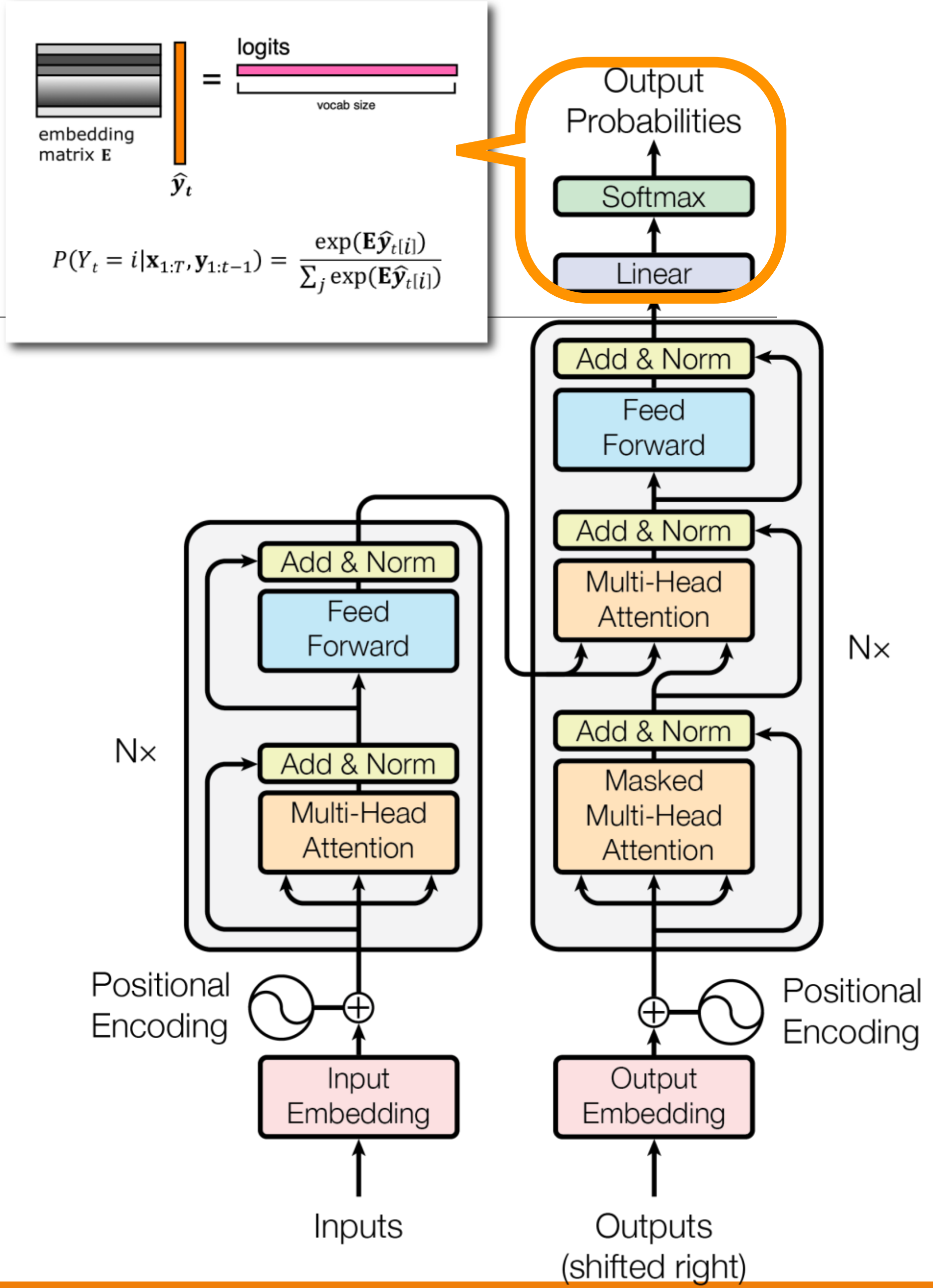
Transformers



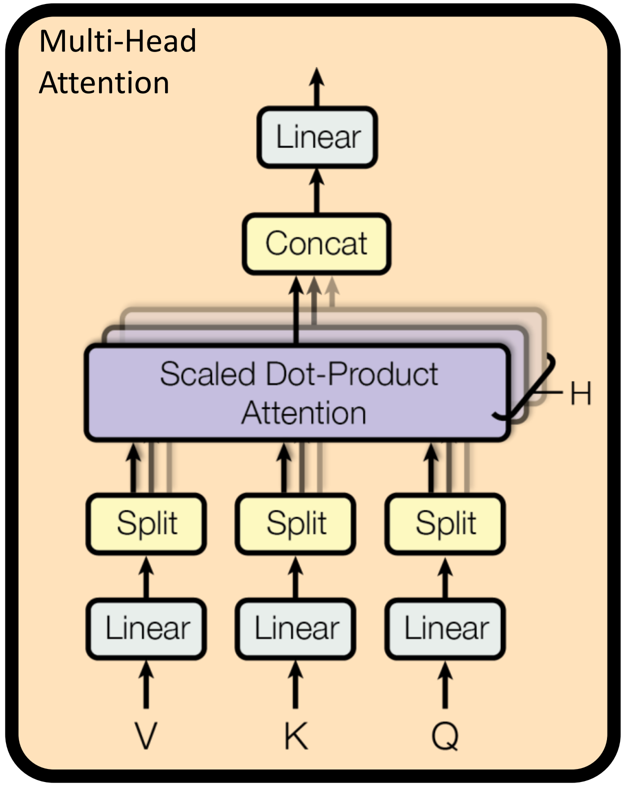
Transformers

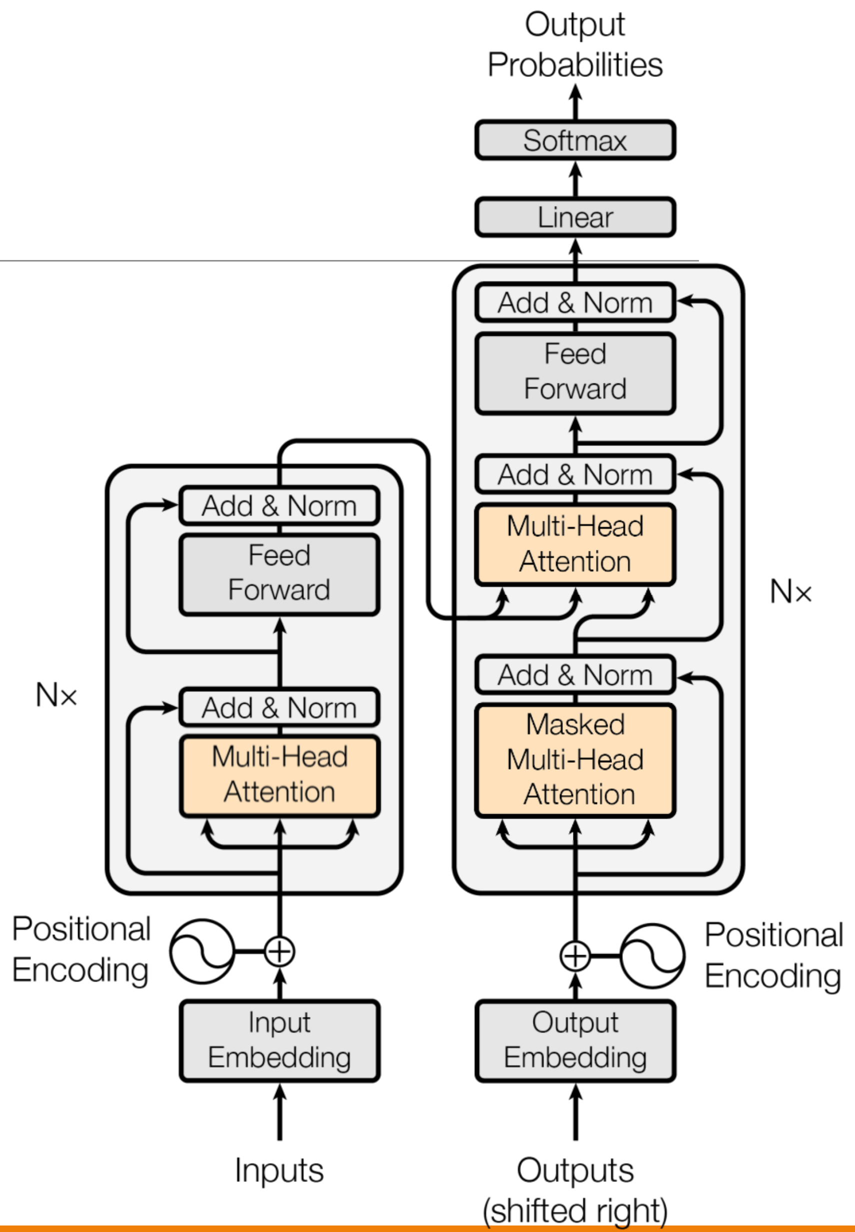


Transformers

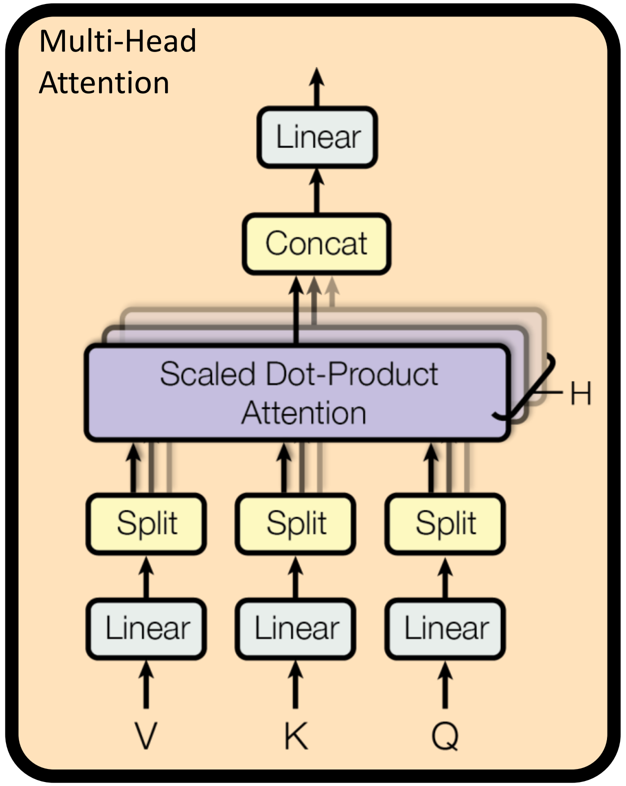


Attention Mechanism

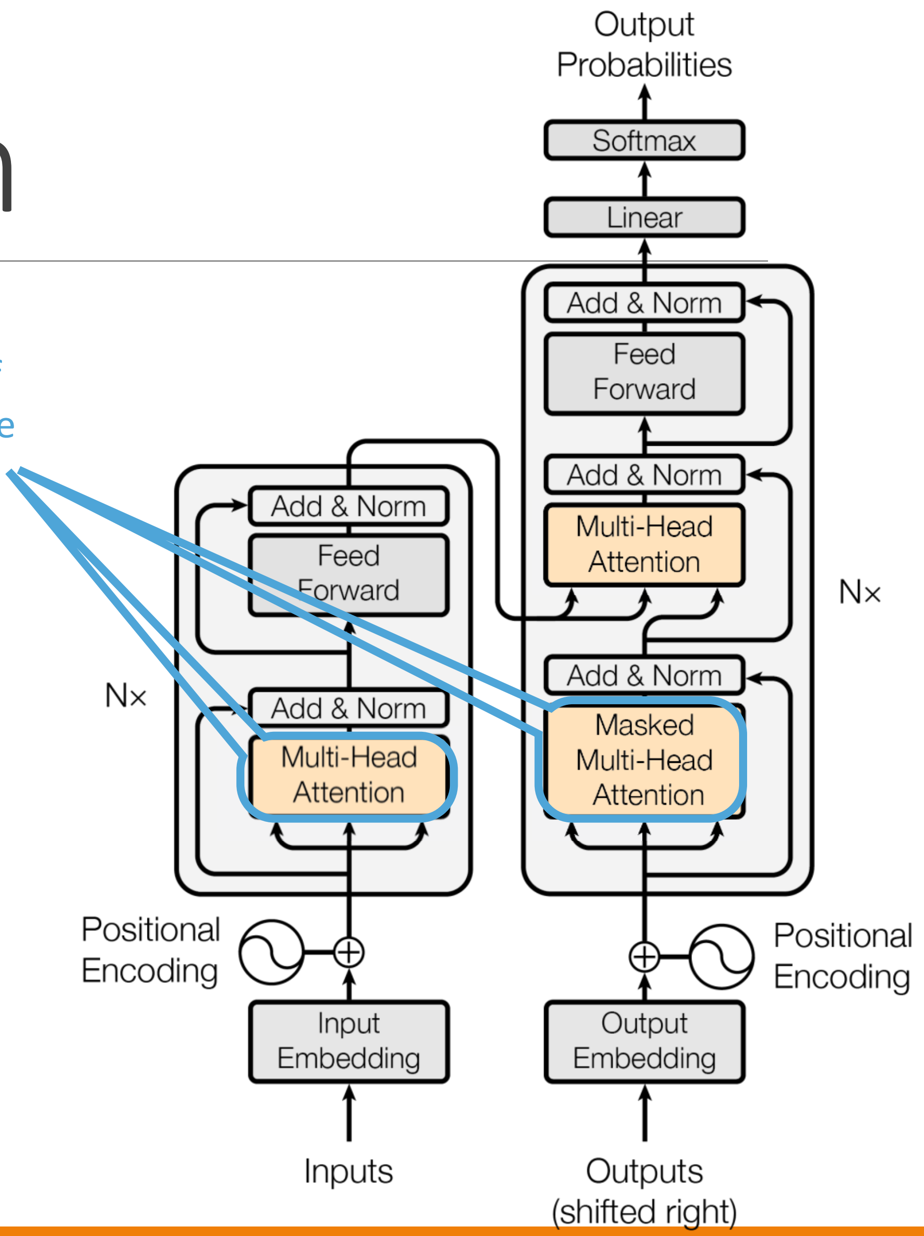




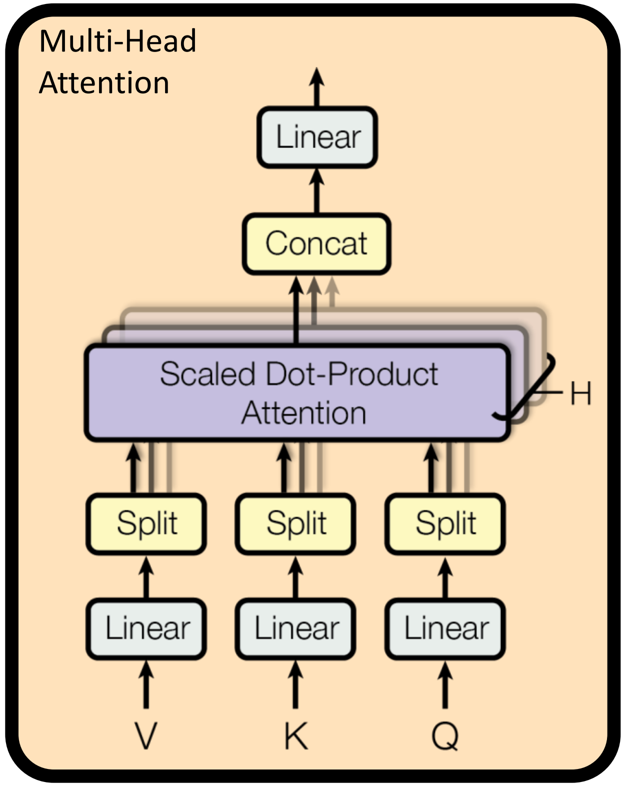
Multi-Head Attention



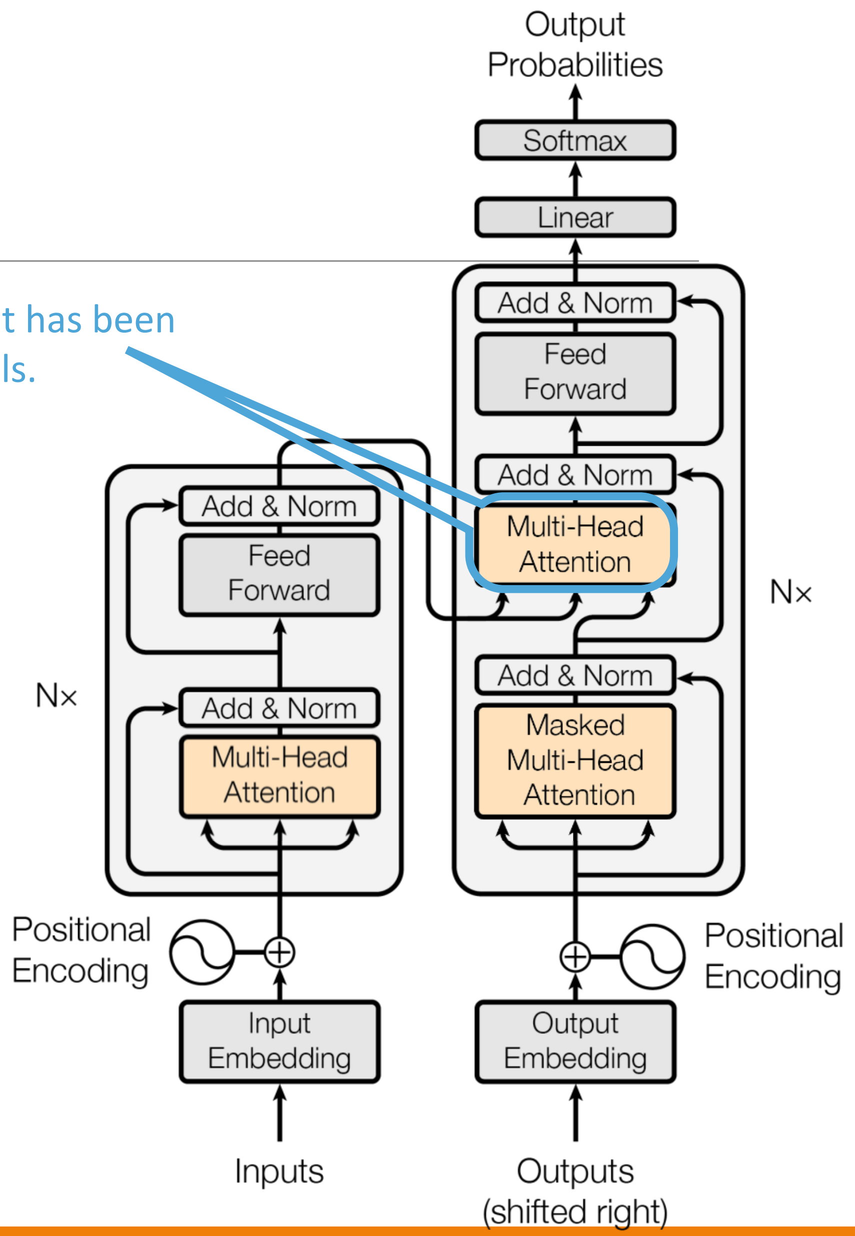
Self-attention between a sequence of hidden states and that same sequence of hidden states.



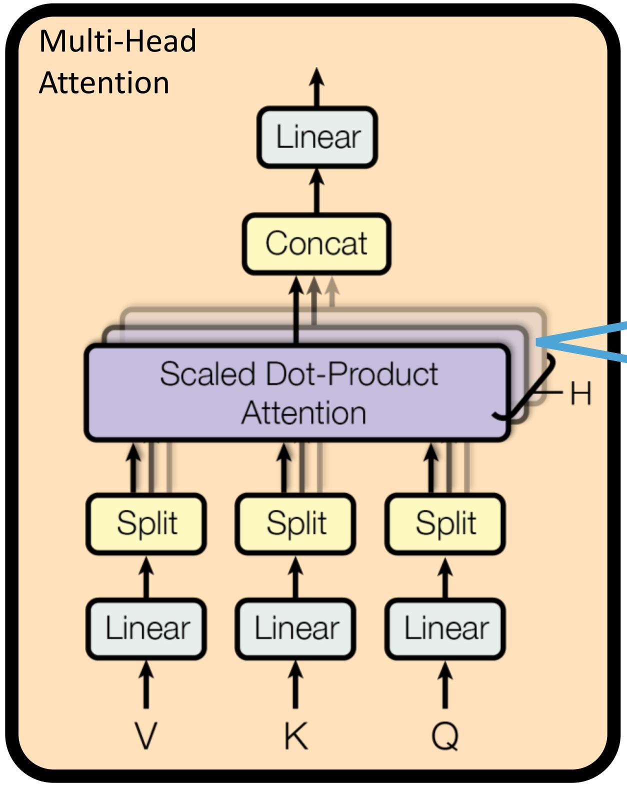
Multi-Head Attention

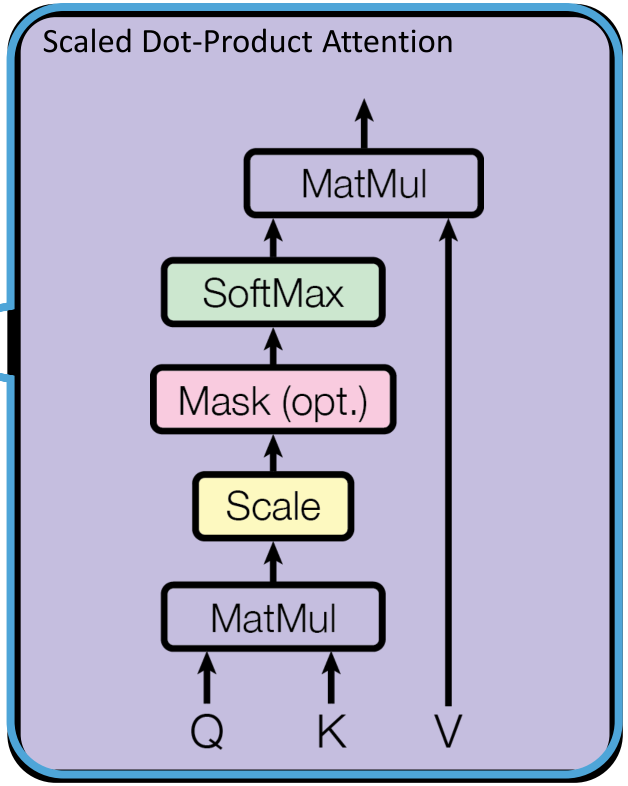


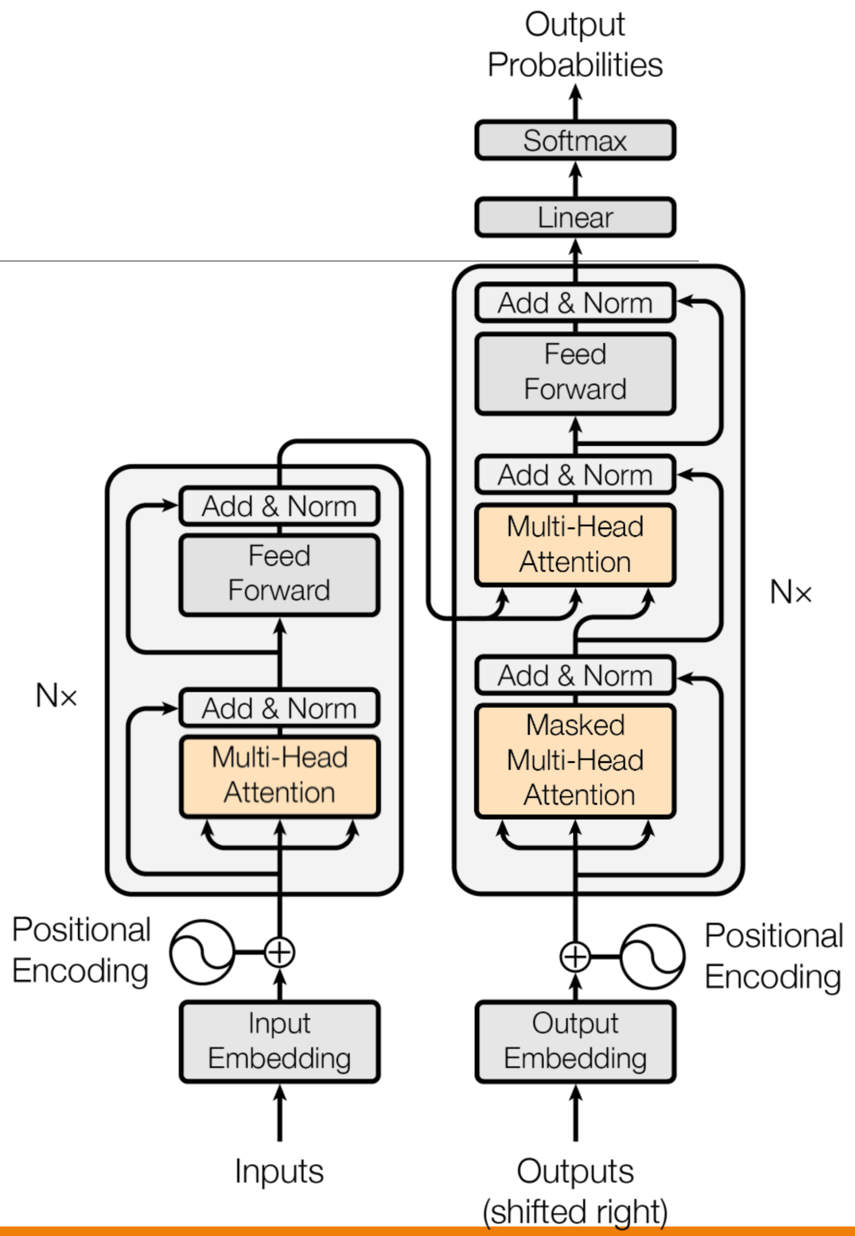
Encoder-decoder attention, like what has been standard in recurrent seq2seq models.



Attention Mechanism

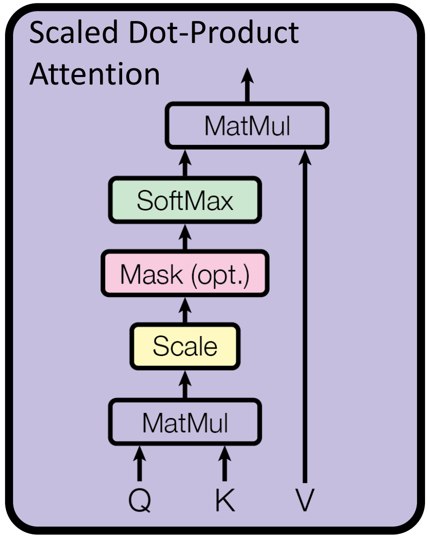






Scaled Dot-Product Attention

The scaled dot-product attention mechanism is almost identical to the one we looked at, but let's turn it into matrix multiplications.



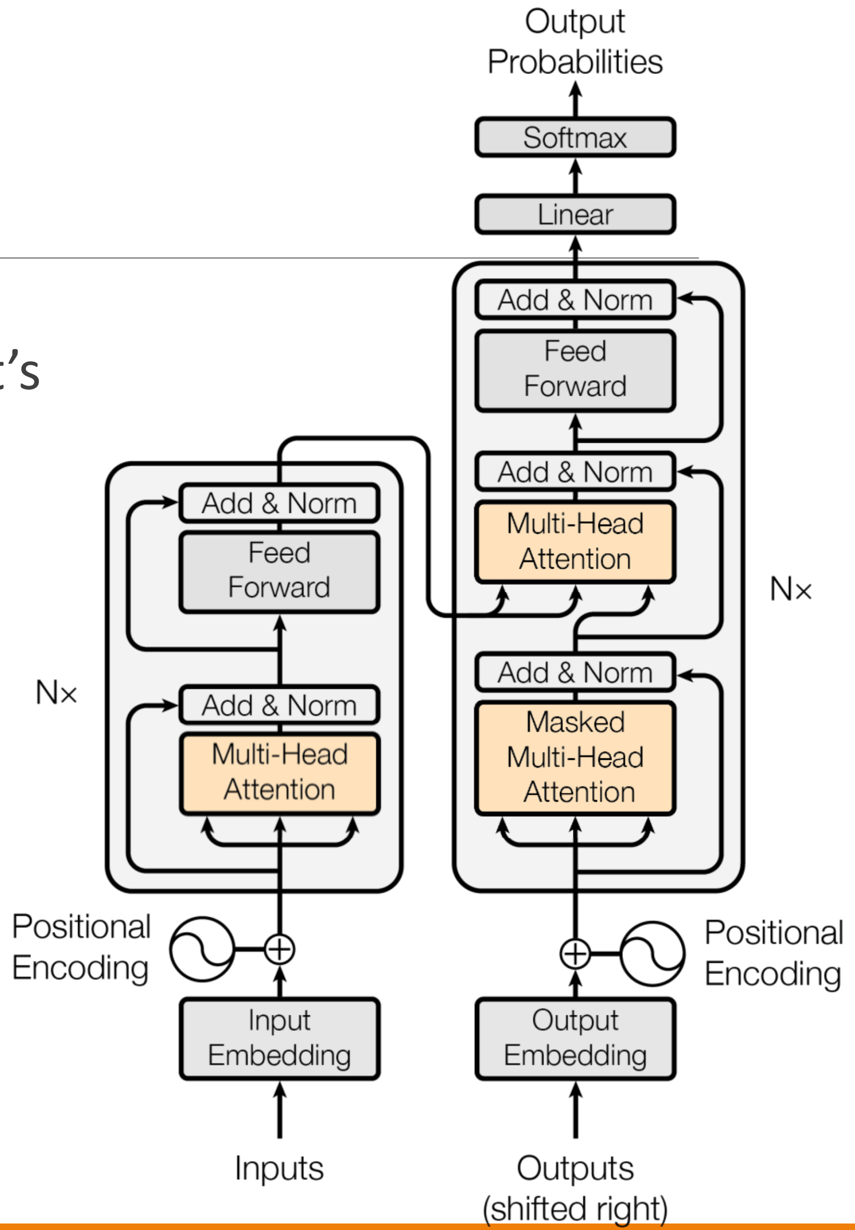
The query: Q $\in R^{Txdk}$

The key: K $\in R^{T'xdk}$

The value: V $\in R^{Txdk}$

This is the α vector we learned about before.

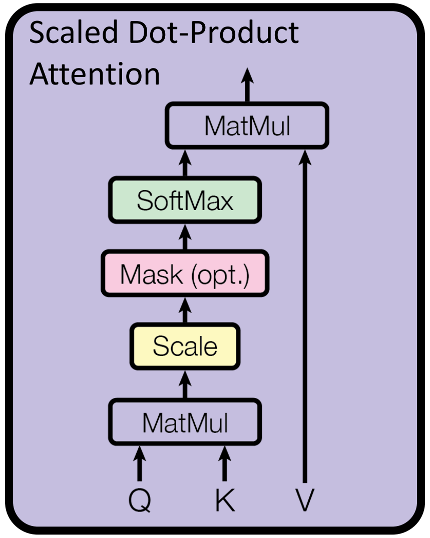
$$\text{Attention}(Q,K,V) = \boxed{\text{softmax}\left(\frac{\mathbf{QK}^T}{\sqrt{d_k}}\right)} V$$



Query, Key, Value

These attributes make more sense as a metaphor for a search engine

- (The original transformer paper was written by Google Brain/Google Research people)



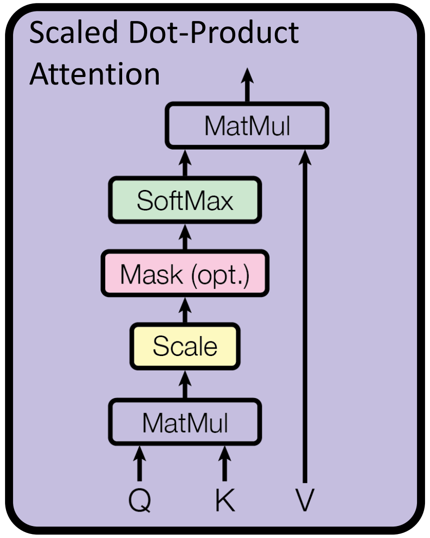
Query: The search query

Keys: The webpages retrieved

Values: The answer to the query

Scaled Dot-Product Attention

The scaled dot-product attention mechanism is almost identical to the one we looked at, but let's turn it into matrix multiplications.



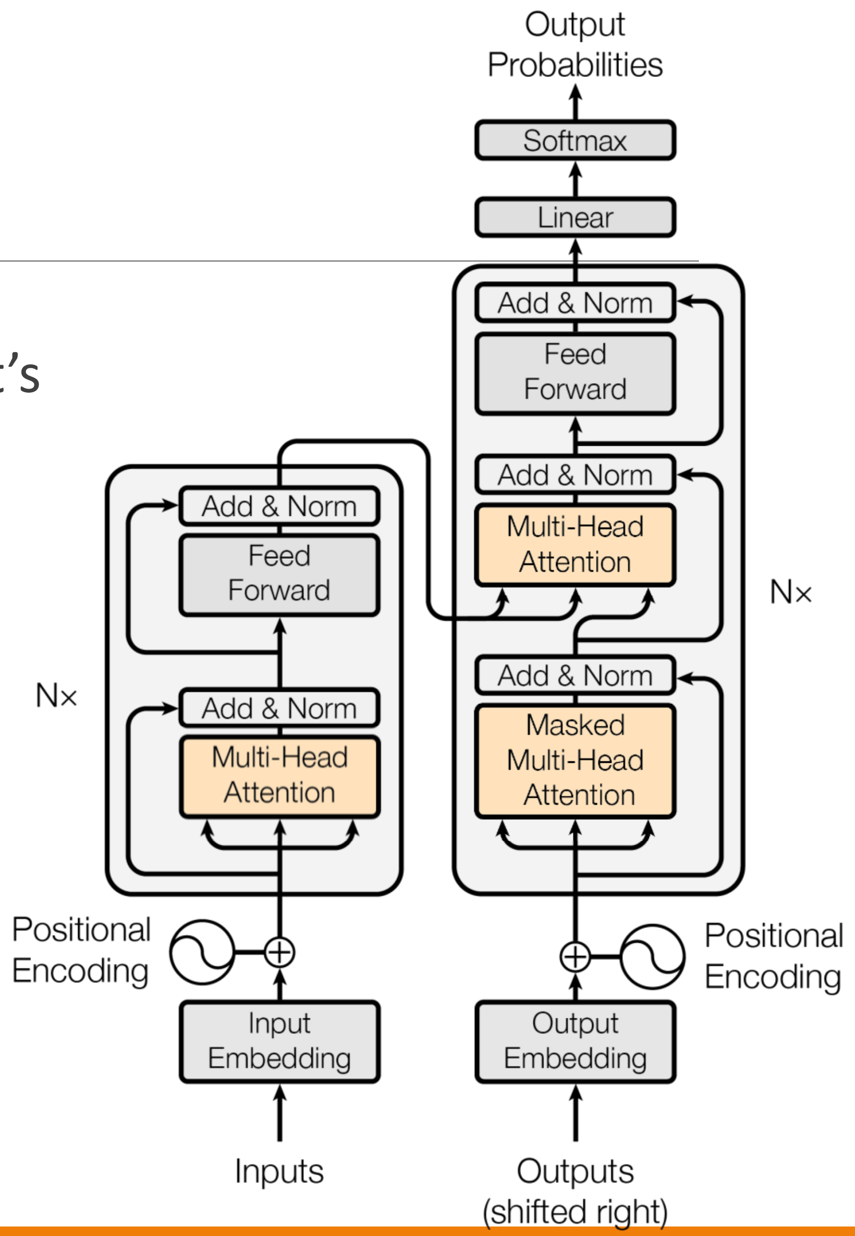
The query: Q ∈ R^{Txdk}

The key: K ∈ $R^{T'xdk}$

The value: V ∈ R^{Txdk}

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

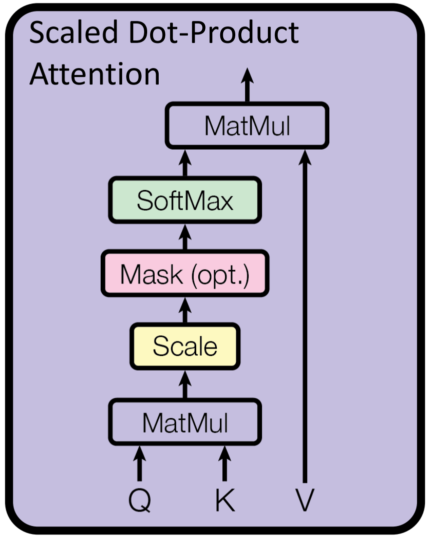
The $\sqrt{d_k}$ in the denominator prevents the dot product from getting too big



Scaled Dot-Product Attention

The scaled dot-product attention mechanism is almost identical to the one we looked at, but let's turn it into matr

$$\text{att\_score}(\mathbf{h}_t^{\text{dec}}, \mathbf{h}_i^{\text{enc}}) = \begin{cases} \mathbf{h}_t^{\text{dec}} \cdot \mathbf{h}_i^{\text{enc}} & \text{dot product} \\ \mathbf{h}_t^{\text{dec}} \, \mathbf{W}a \, \mathbf{h}_i^{\text{enc}} & \text{bilinear function} \\ w_{a1}^{\top} \tanh(\mathbf{W}a\mathbf{2}[\mathbf{h}_t^{\text{dec}}, \mathbf{h}_i^{\text{enc}}]) & \text{MLP} \end{cases}$$



The query: Q ∈

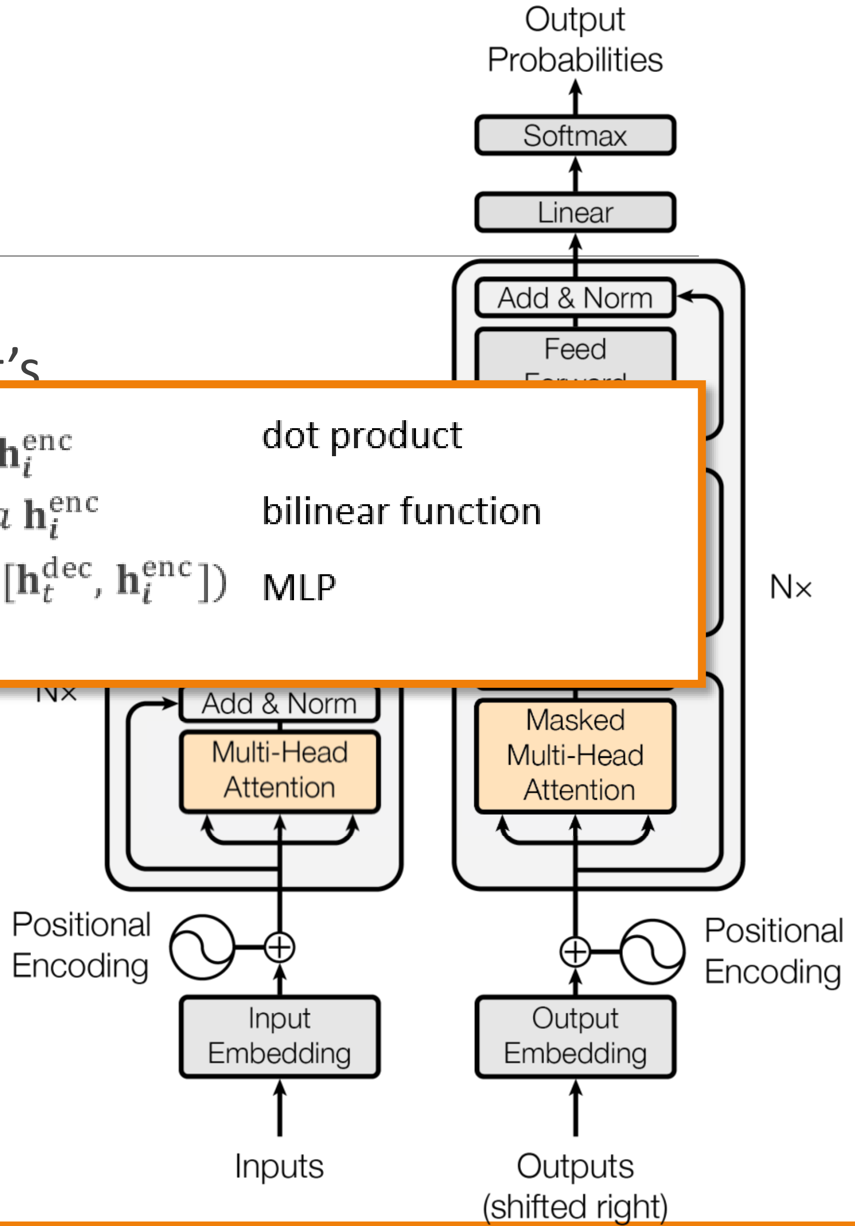
The key: K ∈ R^{Txdk}

The value: V ∈ R^{Txdk}

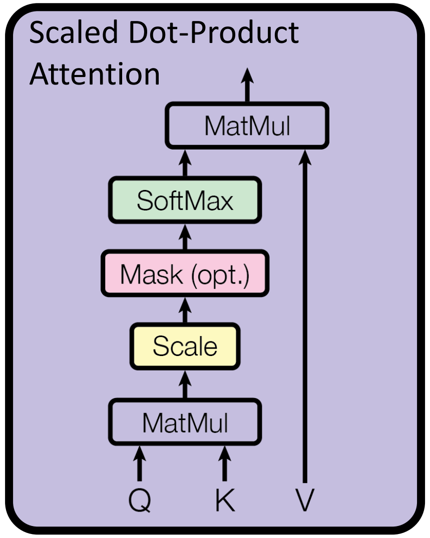
This is the dot-product scoring function from previous slides

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

The $\sqrt{d_k}$ in the denominator prevents the dot product from getting too big



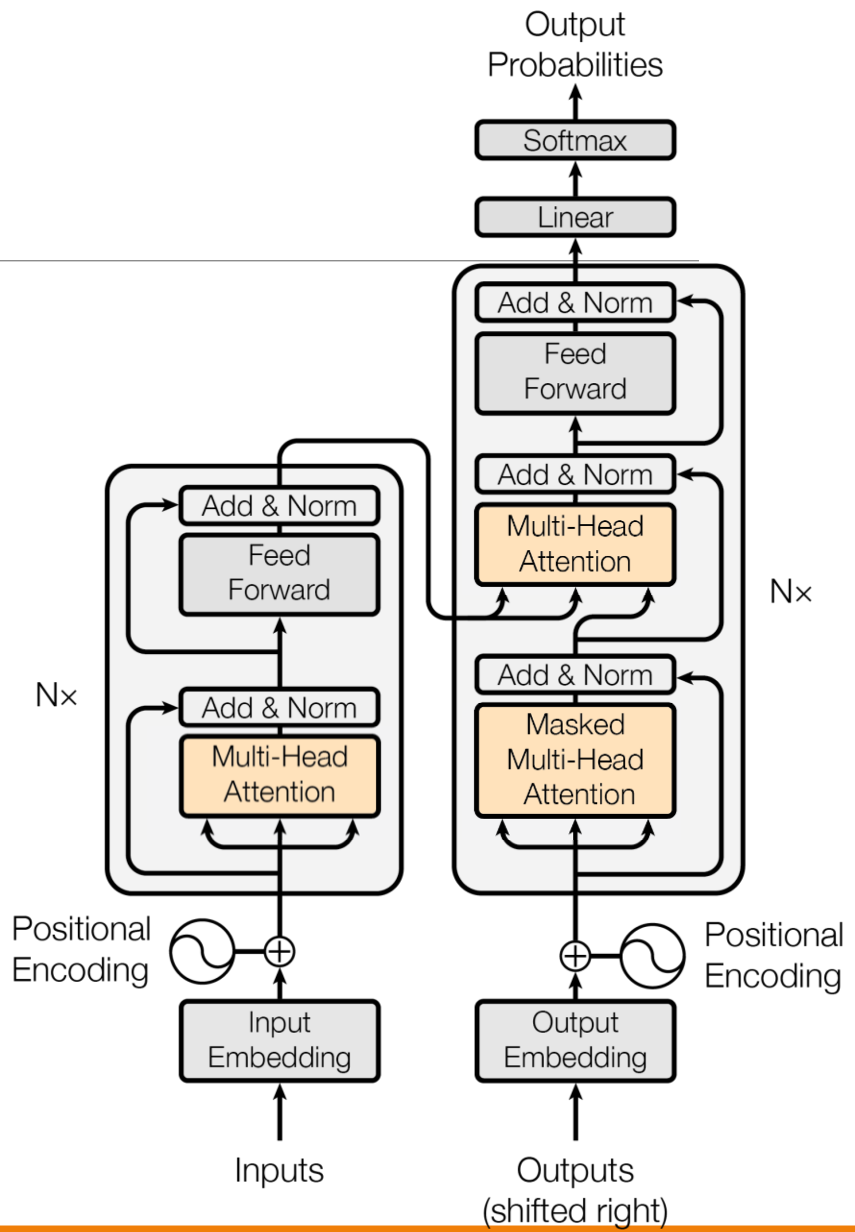
Scaled Dot-Product Attention



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

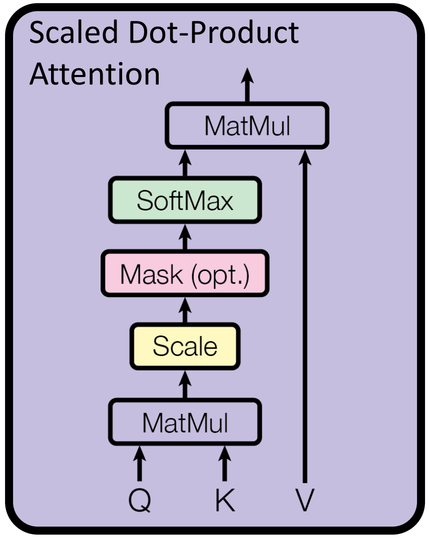
<u>The rough algorithm:</u>

- For each vector in Q (query matrix), take the linear sum of the vectors in V (value matrix)
- The amount to weigh each vector in V is dependent on how "similar" that vector is to the query vector
- "Similarity" is measured in terms of the dot product between the vectors



Scaled Dot-Product Attention

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

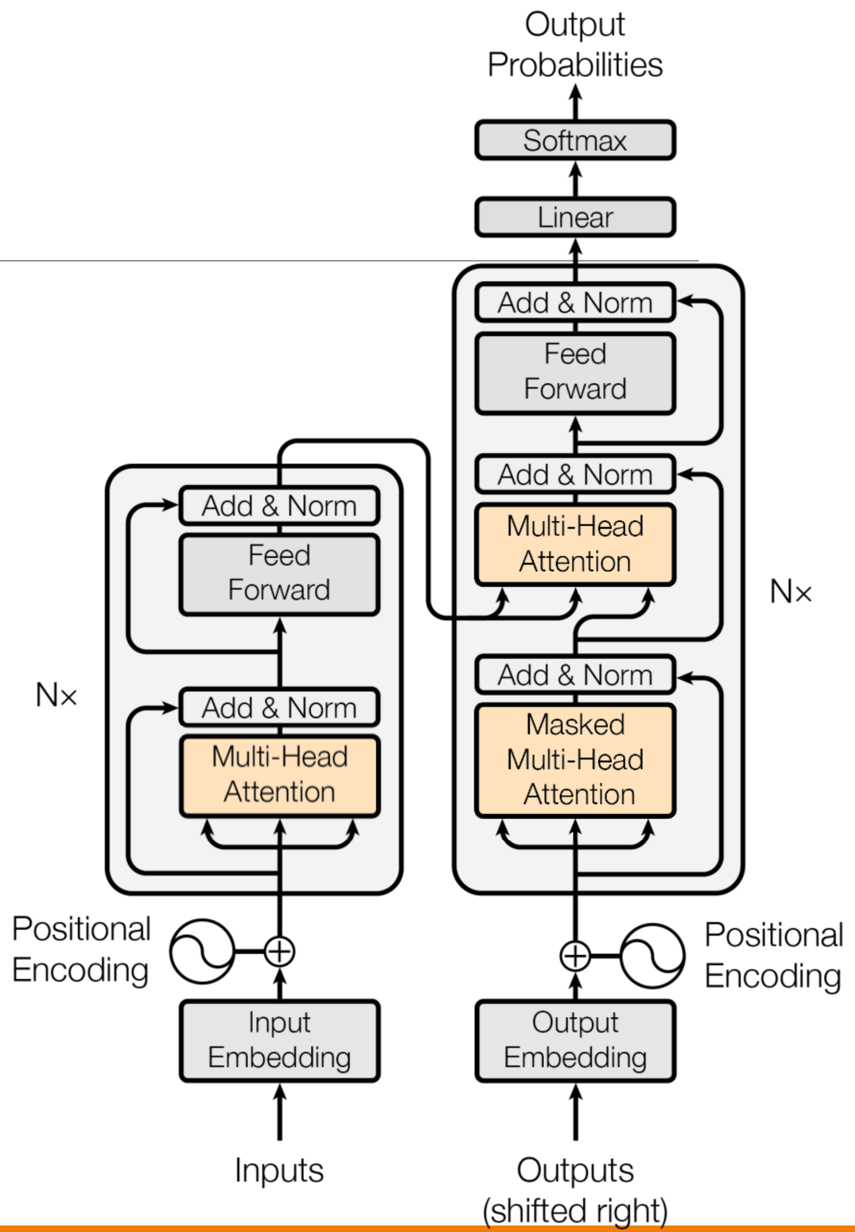


For self-attention:

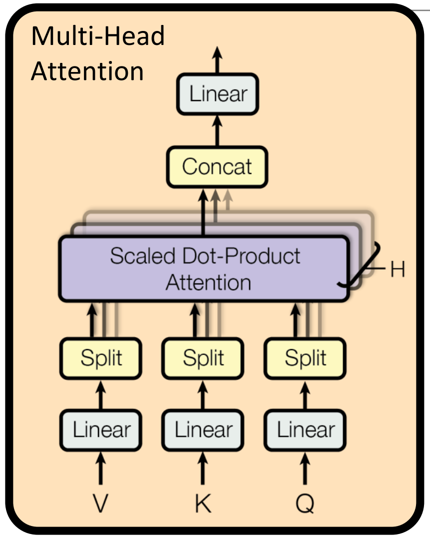
Keys, queries, and values all come from the outputs of the previous layer

For encoder-decoder attention:

Keys and values come from encoder's final output. Queries come from the previous decoder layer's outputs.



Multi-Head Attention



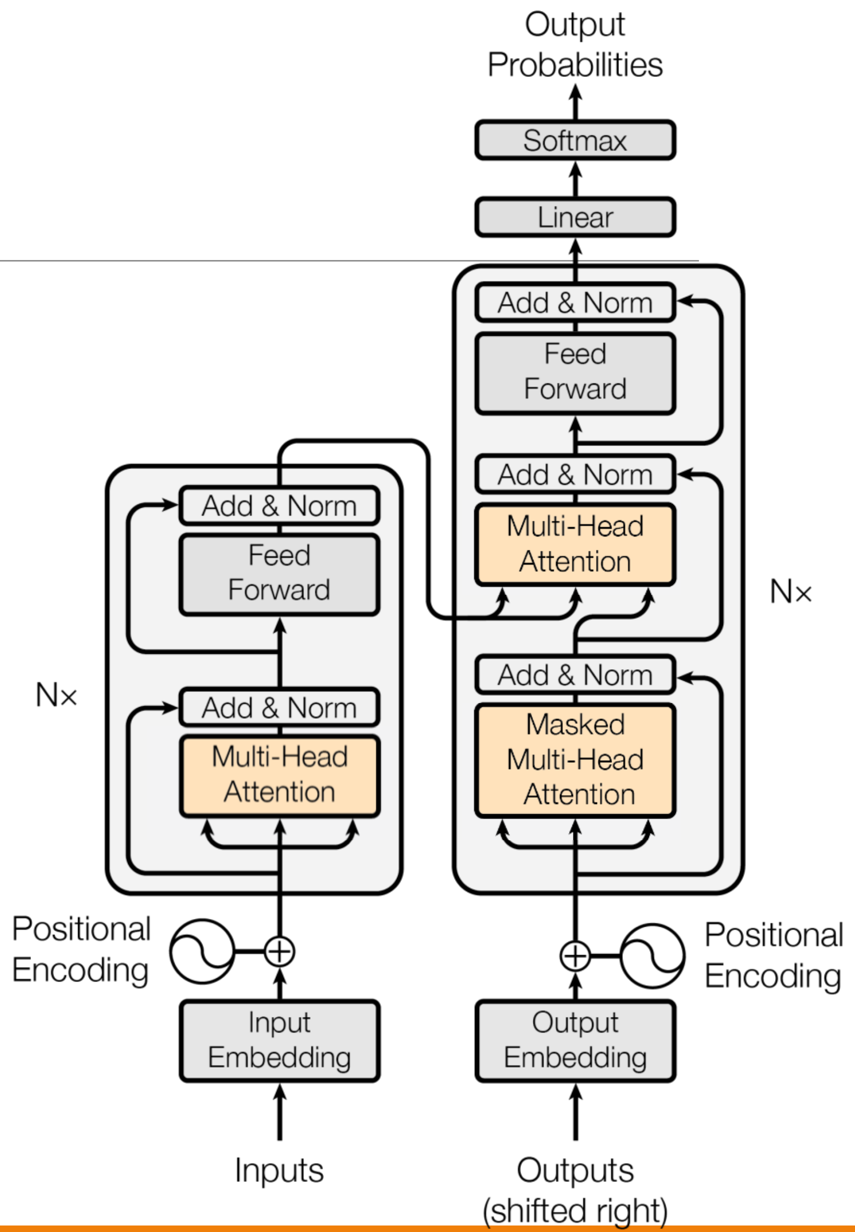
$$\text{Attention(Q,K,V)} = \text{softmax}\left(\frac{\mathbf{QK}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

$$\text{MultiHeadAtt}(\mathbf{Q,K,V}) = \text{Concat}(\text{head}_1, \ldots \text{head}_h)\mathbf{W}^O$$

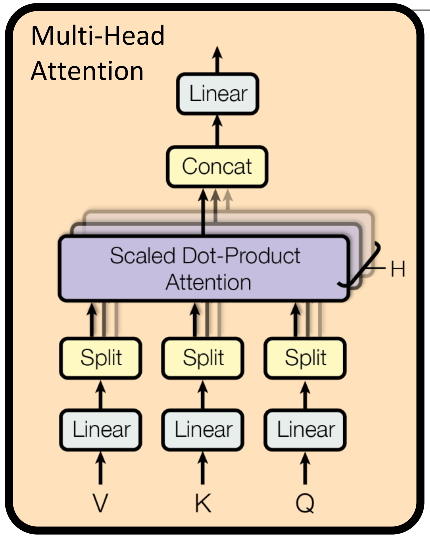
Instead of operating on **Q**, **K**, and **V** mechanism projects each input into a smaller dimension. This is done h times.

The attention operation is performed on each of these “heads,” and the results are concatenated.

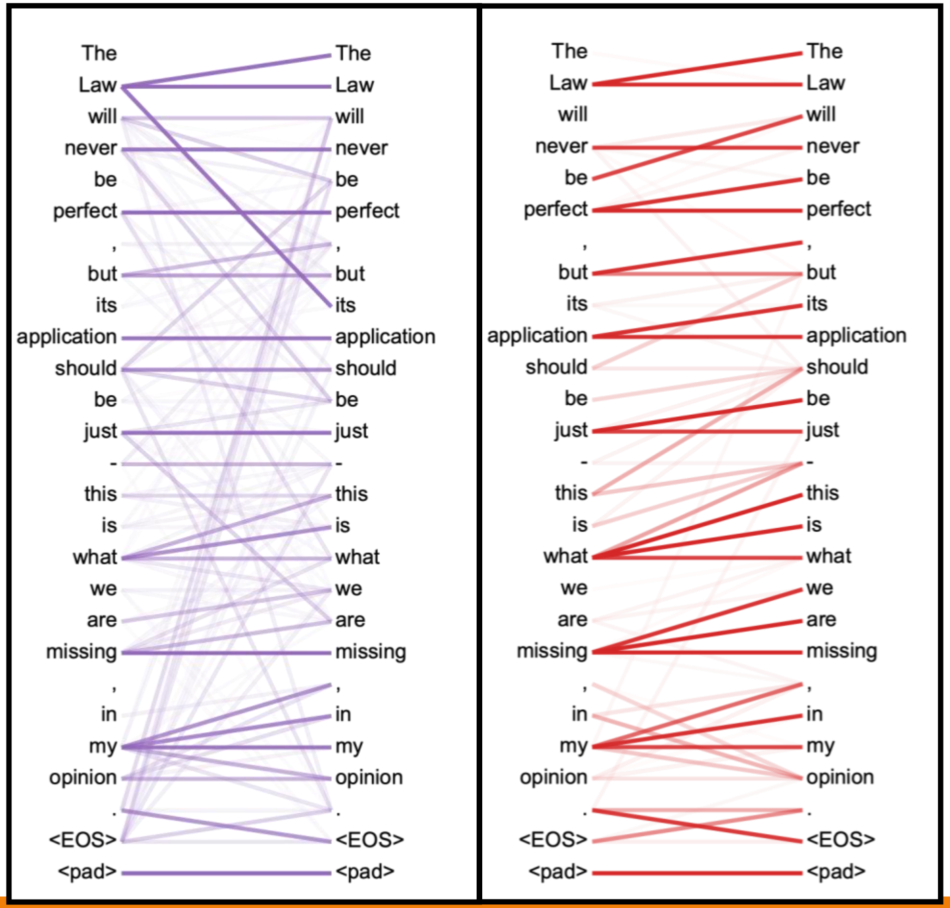
Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions.

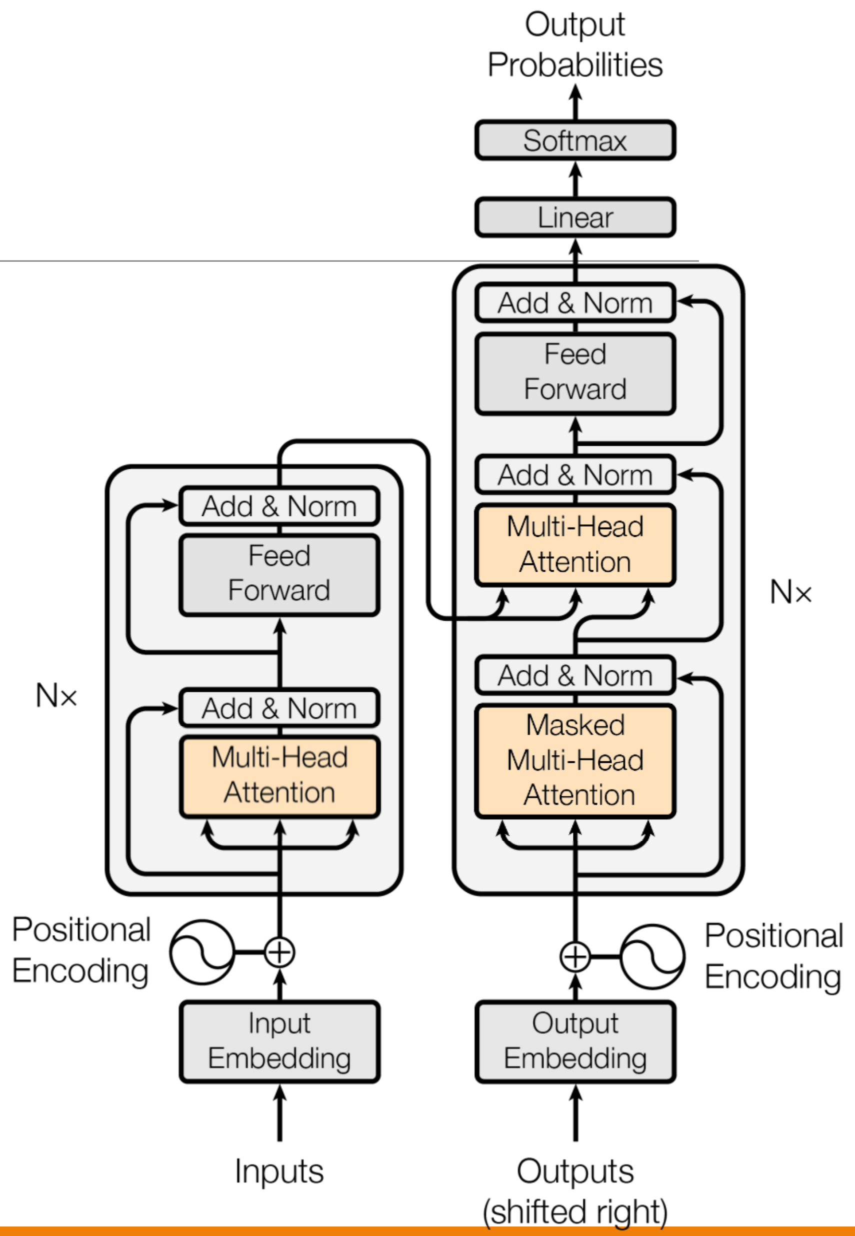


Multi-Head Attention



Two different self-attention heads:

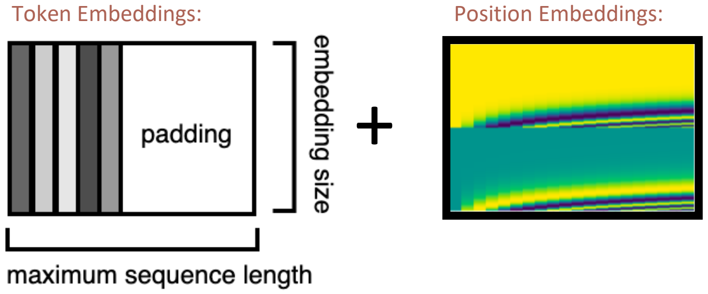




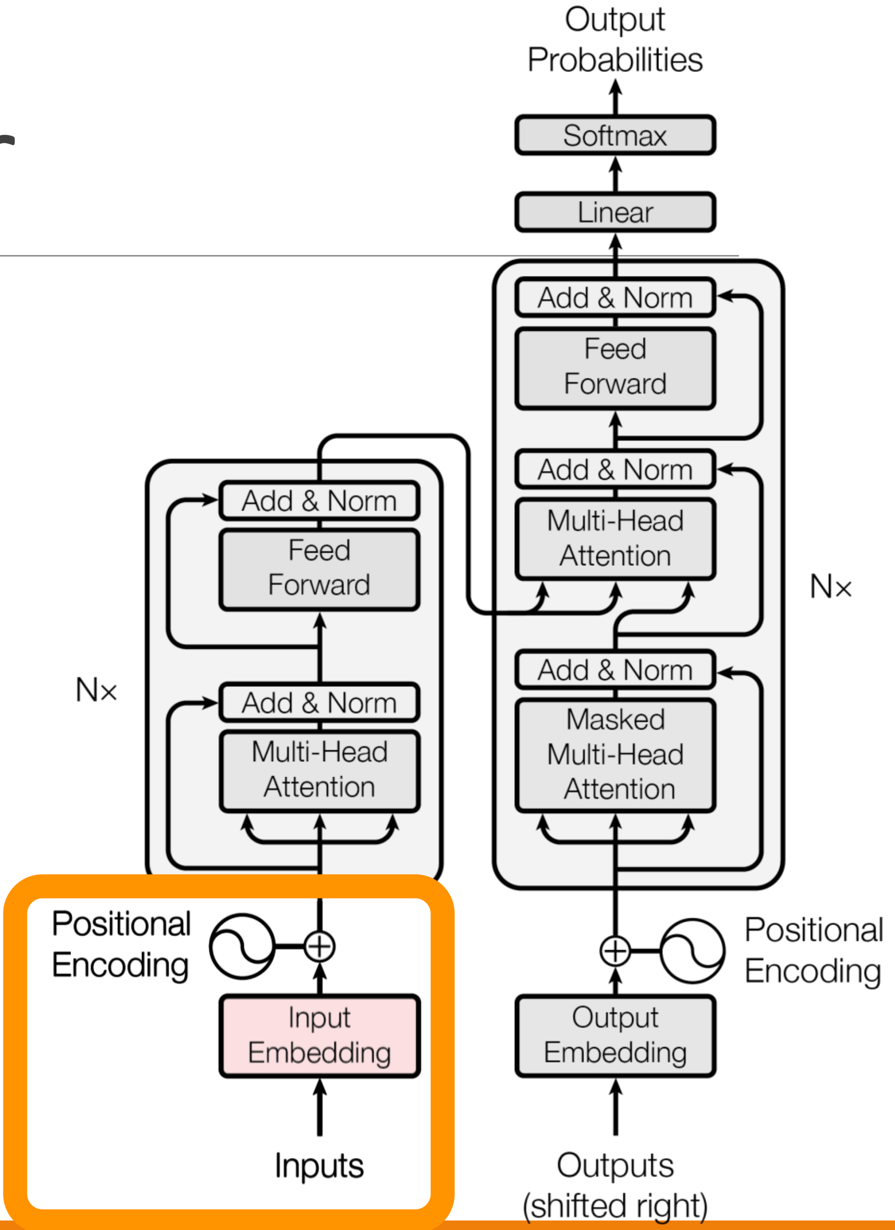
Inputs to the Encoder

The input into the encoder looks like:

$\mathbf{H}_0^{\text{enc}} =$

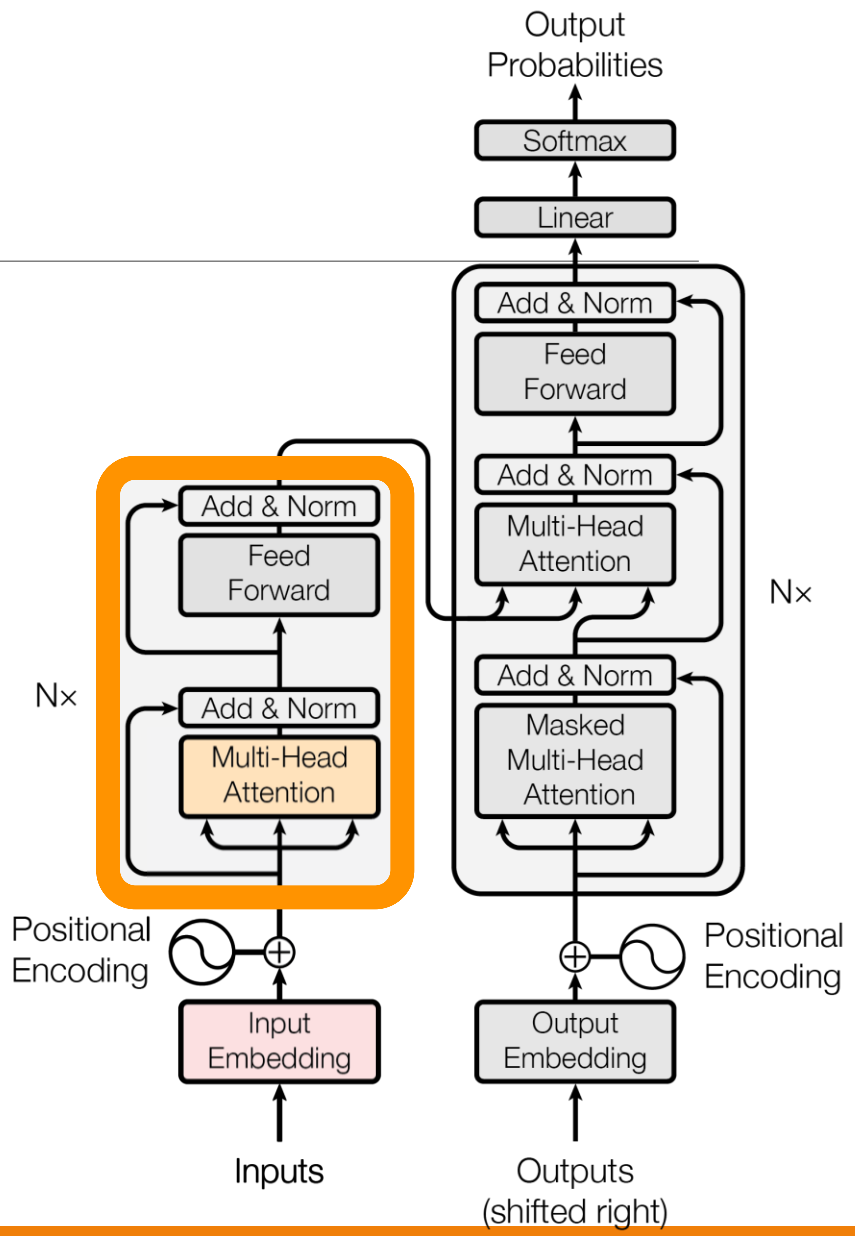


= token embeddings + position embeddings



The Encoder

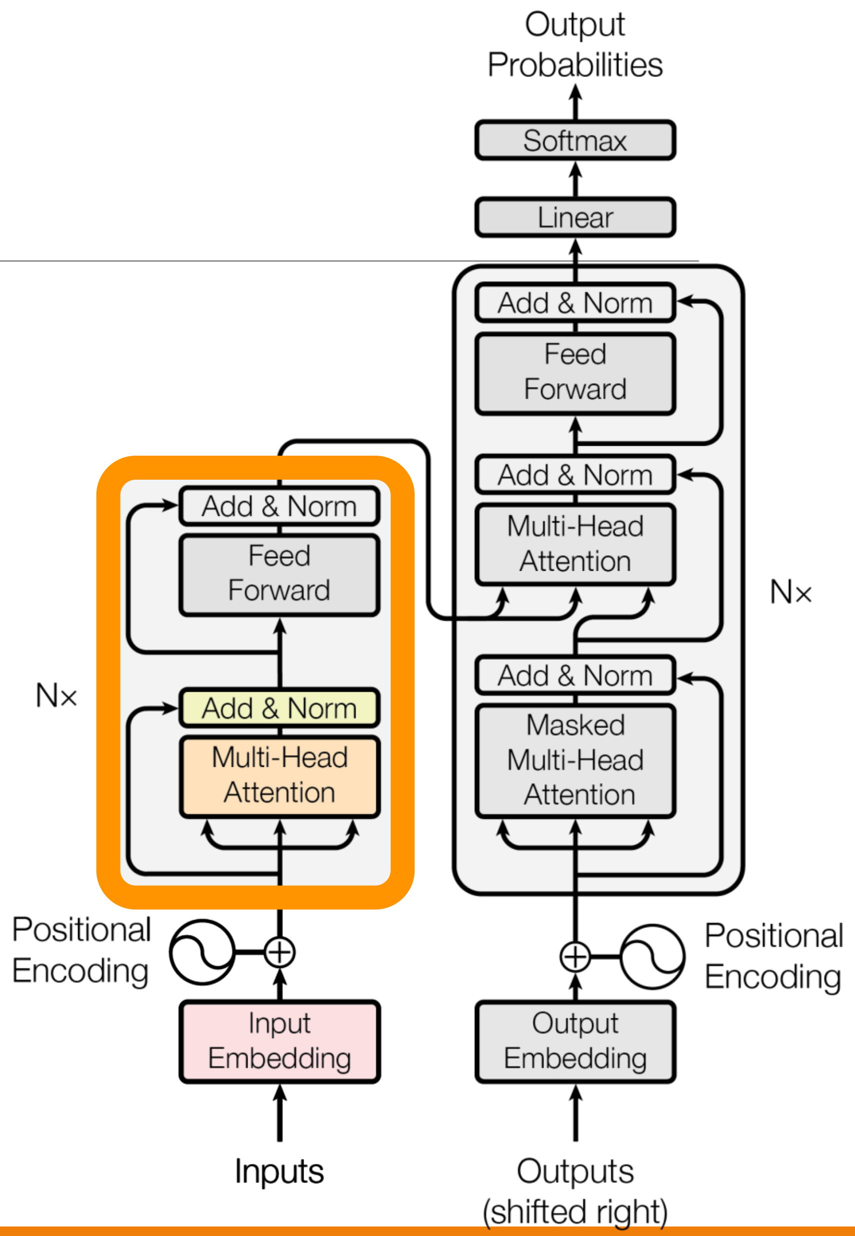
Multi-Head Attention $= \text{MultiHeadAtt}(\mathbf{H}_i^{enc}, \mathbf{H}_i^{enc}, \mathbf{H}_i^{enc})$



The Encoder

$\boxed{\text{Multi-Head Attention}}$ = MultiHeadAtt($\mathbf{H}_i^{enc}$, $\mathbf{H}_i^{enc}$, $\mathbf{H}_i^{enc}$)

$\boxed{\text{Add \& Norm}}$ = LayerNorm($\boxed{\text{Multi-Head Attention}}$ + $\mathbf{H}_i^{enc}$)

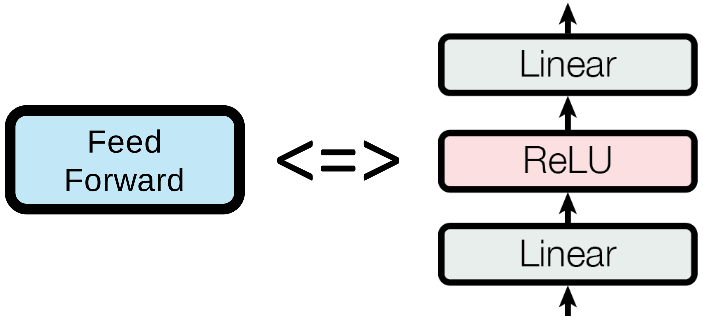


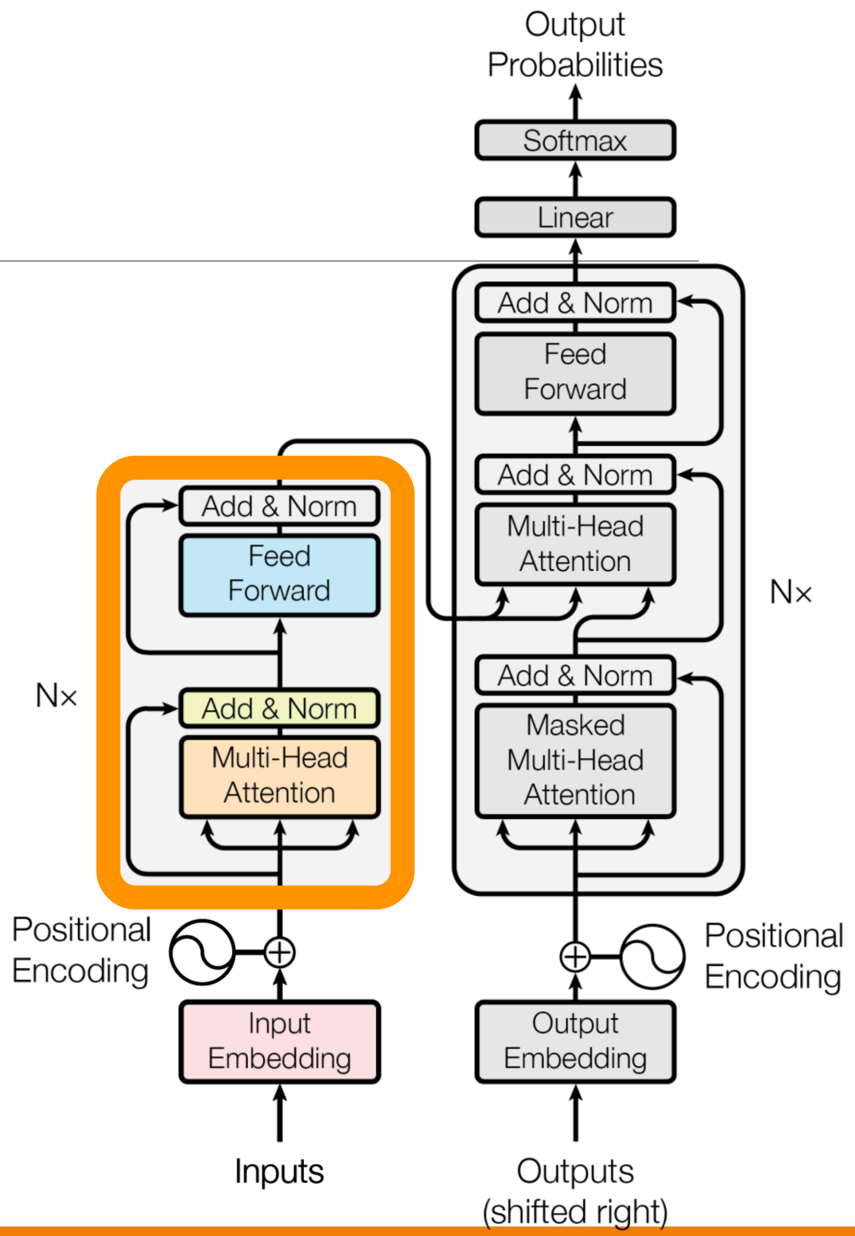
The Encoder

$$\boxed{\text{Multi-Head Attention}} = \text{MultiHeadAtt}(\mathbf{H}_i^{enc}, \mathbf{H}_i^{enc}, \mathbf{H}_i^{enc})$$

$$\boxed{\text{Add \& Norm}} = \text{LayerNorm}(\boxed{\text{Multi-Head Attention}} + \mathbf{H}_i^{enc})$$

$$\boxed{\text{Feed Forward}} = max(0, \boxed{\text{Add \& Norm}}\,\mathbf{W}_1 + b_1)\mathbf{W}_2 + b_2$$





The Encoder

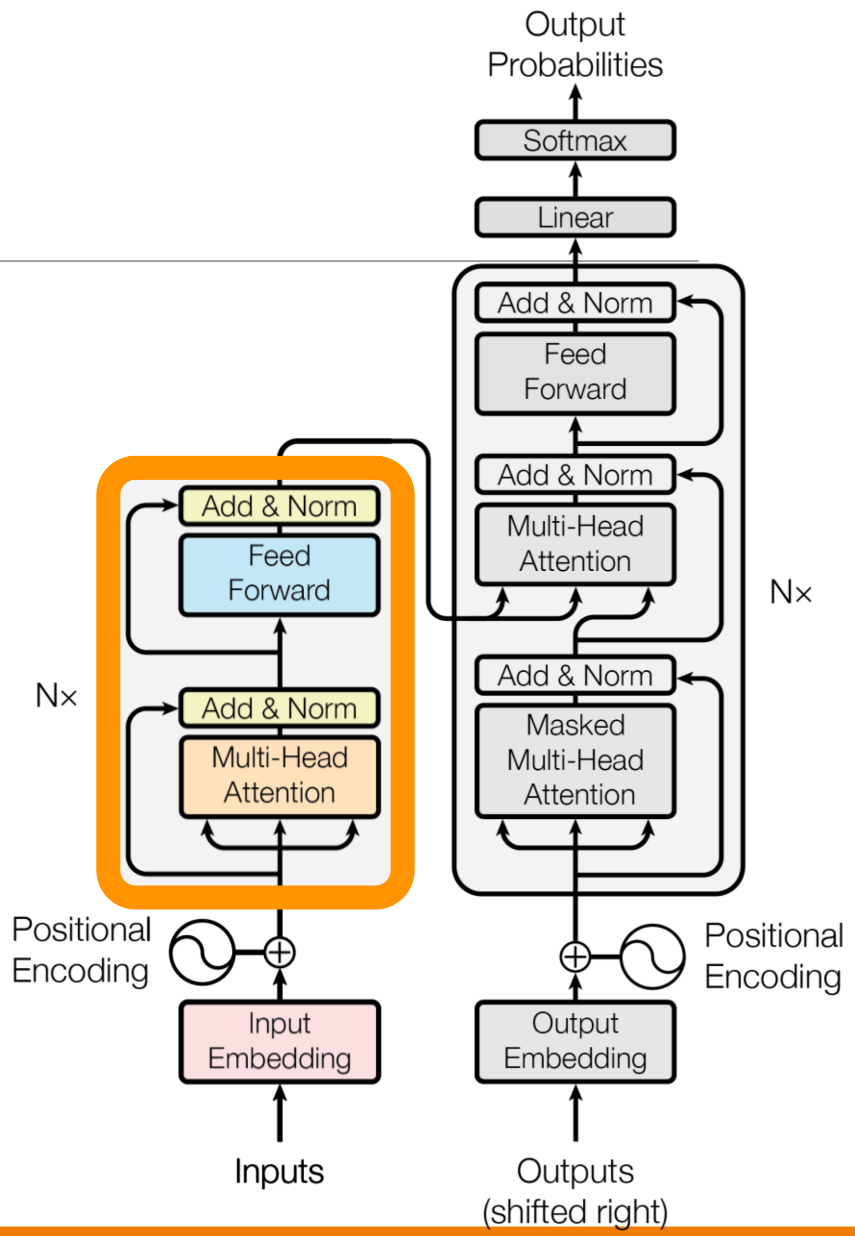
$$\boxed{\text{Multi-Head Attention}} = \text{MultiHeadAtt}(\mathbf{H}_i^{enc}, \mathbf{H}_i^{enc}, \mathbf{H}_i^{enc})$$

$$\boxed{\text{Add \& Norm}} = \text{LayerNorm}(\boxed{\text{Multi-Head Attention}} + \mathbf{H}_i^{enc})$$

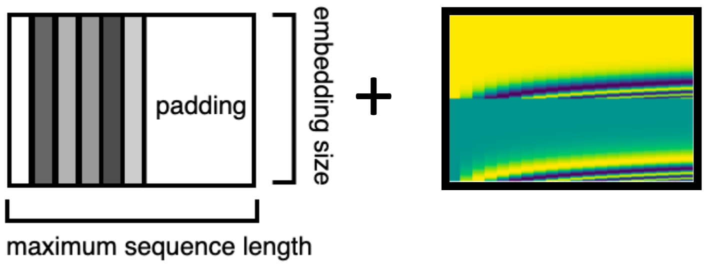
$$\boxed{\text{Feed Forward}} = max(0, \boxed{\text{Add \& Norm}}\,\mathbf{W}_1 + b_1)\mathbf{W}_2 + b_2$$

$$\boxed{\text{Add \& Norm (2)}} = \text{LayerNorm}(\boxed{\text{Feed Forward}} + \mathbf{H}_i^{enc})$$

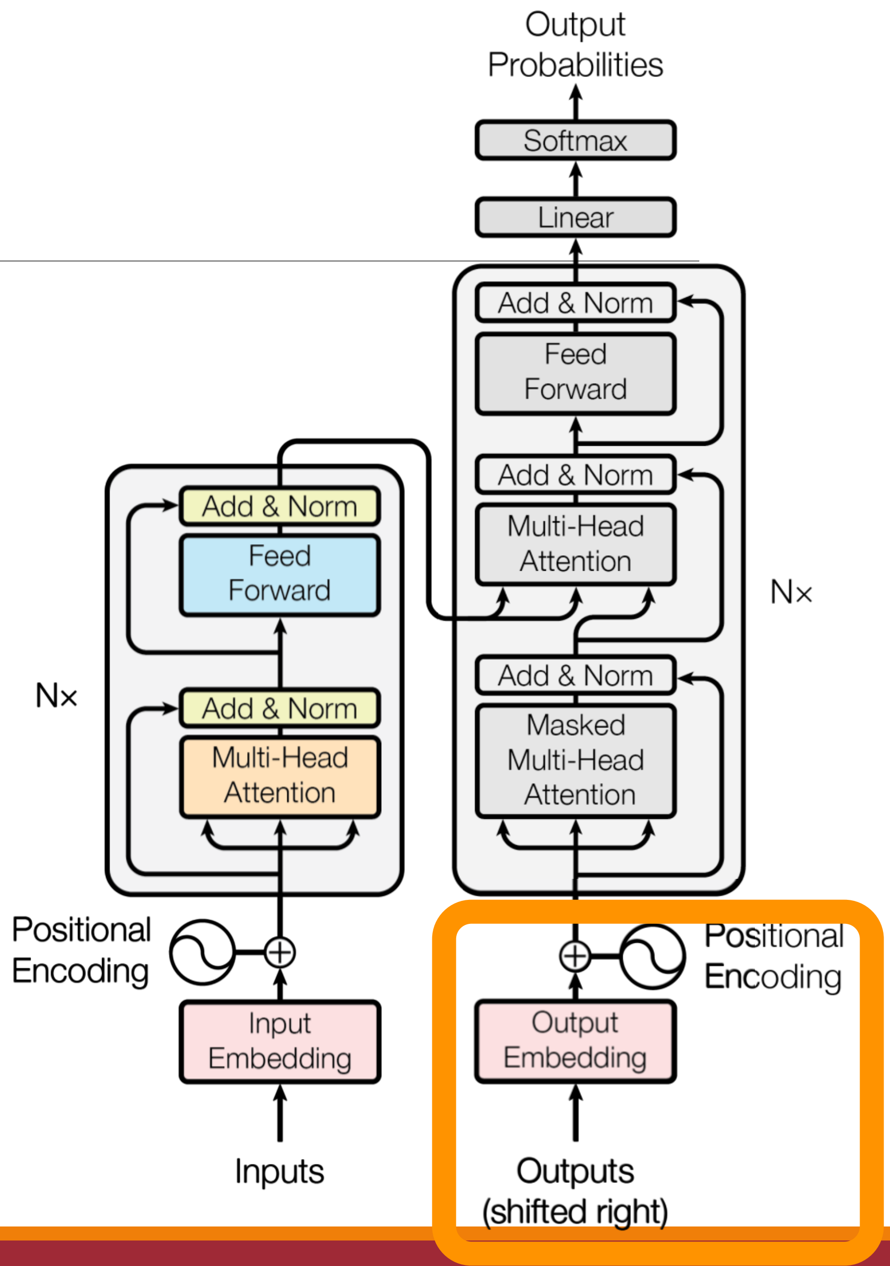
$$\mathbf{H}_{i+1}^{enc} = \boxed{\text{Add \& Norm (2)}}$$



The Decoder

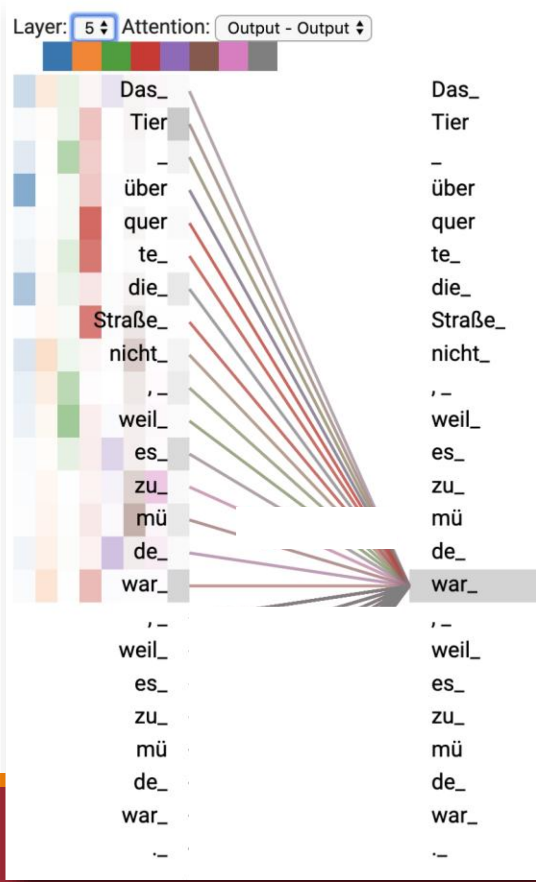
$\mathbf{H}_0^{\mathrm{dec}} =$ 

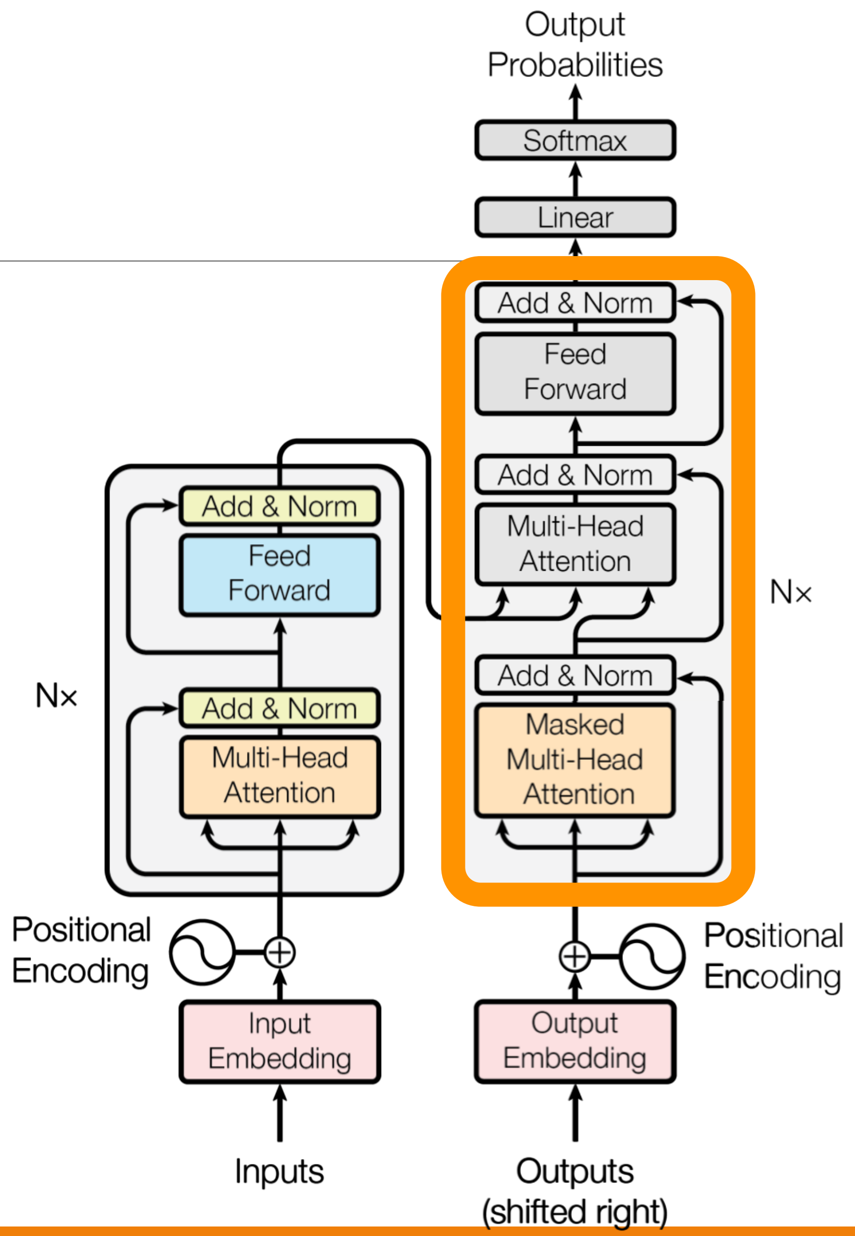
= token embeddings + position embeddings



The Decoder

Masked Multi-Head Attention $= \text{MaskedMultiHeadAtt}(\mathbf{H}_i^{dec}, \mathbf{H}_i^{dec}, \mathbf{H}_i^{dec})$

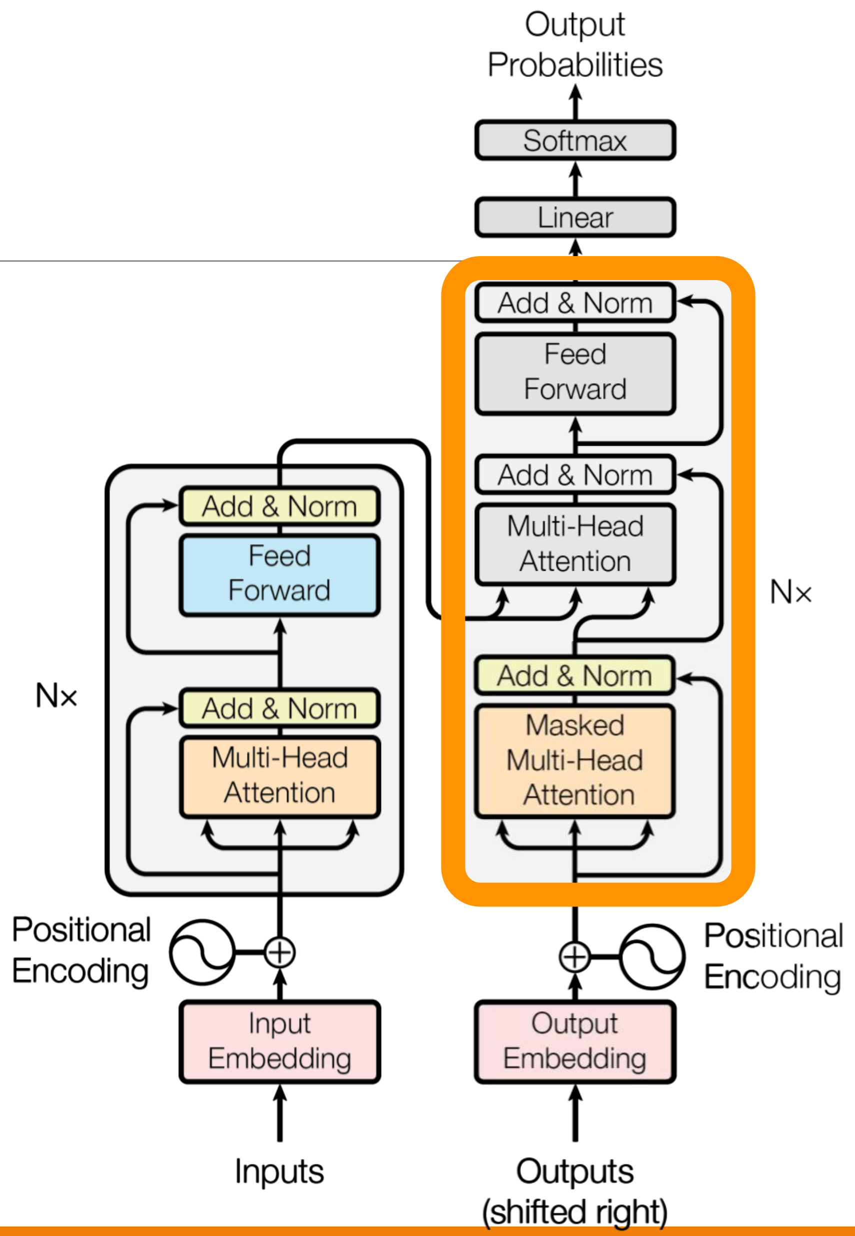




The Decoder

Masked Multi-Head Attention = MaskedMultiHeadAtt($\mathbf{H}_i^{dec}, \mathbf{H}_i^{dec}, \mathbf{H}_i^{dec}$)

Add & Norm = LayerNorm(Multi-Head Attention + $\mathbf{H}_i^{dec}$)

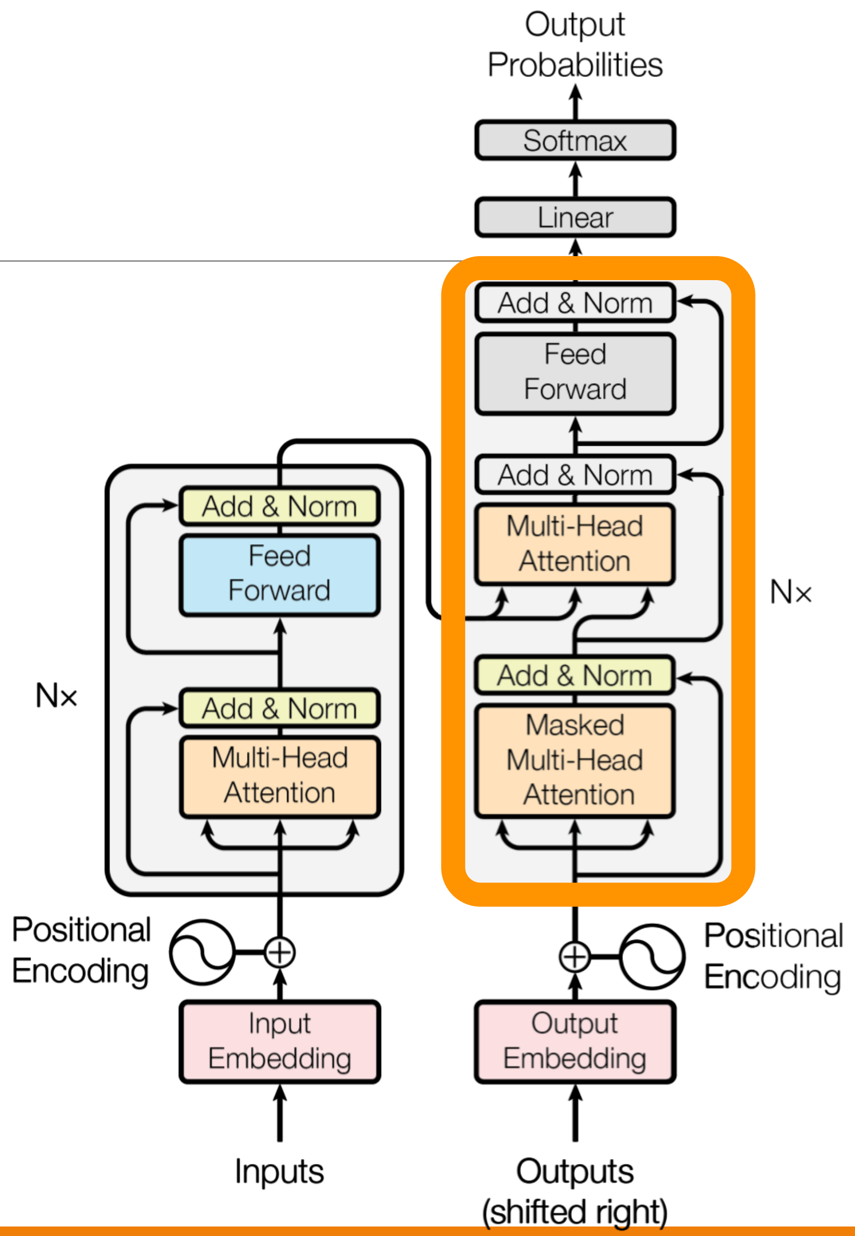


The Decoder

Masked Multi-Head Attention $= \text{MaskedMultiHeadAtt}(\mathbf{H}_i^{dec}, \mathbf{H}_i^{dec}, \mathbf{H}_i^{dec})$

Add & Norm $= \text{LayerNorm}($Multi-Head Attention $+ \mathbf{H}_i^{dec})$

Enc-Dec Multi-Head Attention $= \text{MultiHeadAtt}(\mathbf{H}_i^{dec}, \mathbf{H}_i^{enc}, \mathbf{H}_i^{enc})$



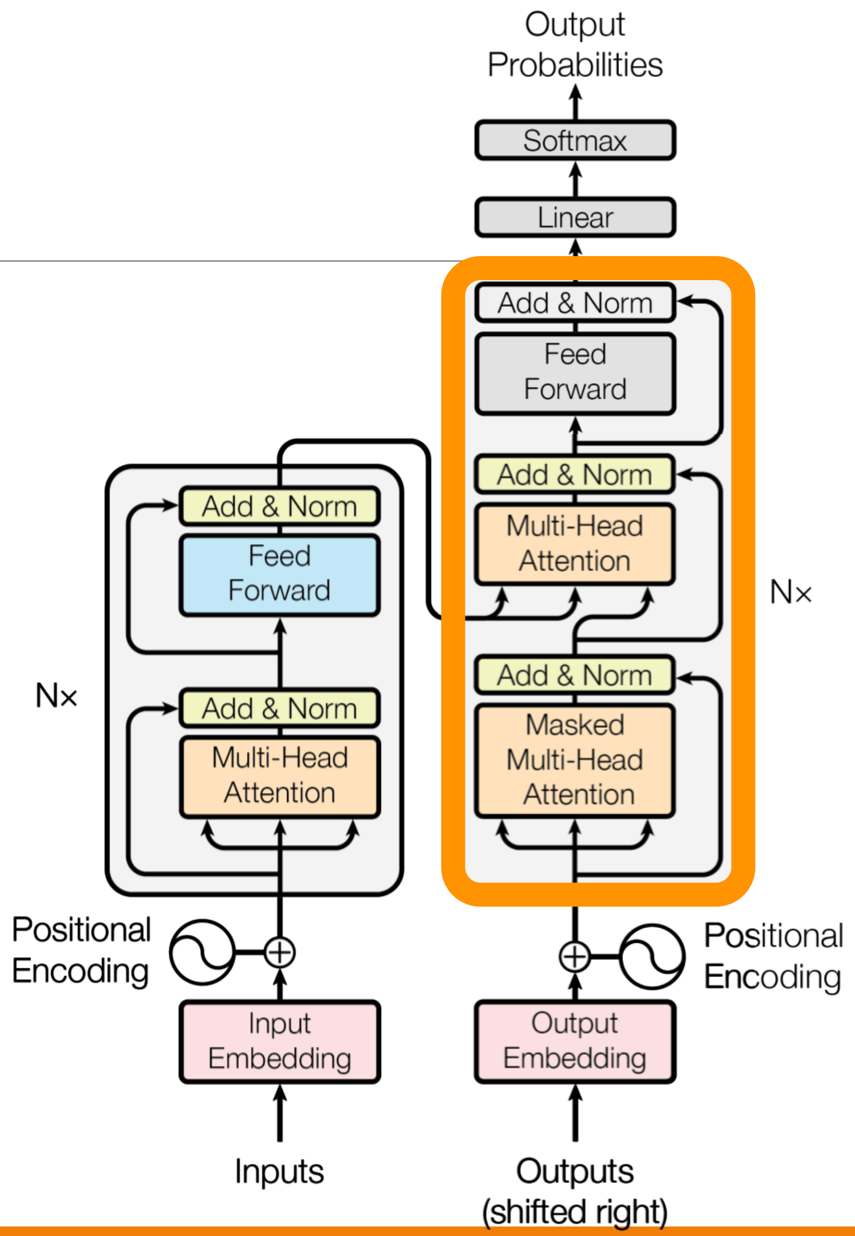
The Decoder

$\boxed{\text{Masked Multi-Head Attention}} = \text{MaskedMultiHeadAtt}(\mathbf{H}_i^{dec}, \mathbf{H}_i^{dec}, \mathbf{H}_i^{dec})$

$\boxed{\text{Add \& Norm}} = \text{LayerNorm}(\boxed{\text{Multi-Head Attention}} + \mathbf{H}_i^{dec})$

$\boxed{\text{Enc-Dec Multi-Head Attention}} = \text{MultiHeadAtt}(\mathbf{H}_i^{dec}, \mathbf{H}_i^{enc}, \mathbf{H}_i^{enc})$

$\boxed{\text{Add \& Norm (2)}} = \text{LayerNorm}(\boxed{\text{Multi-Head Attention}} + \boxed{\text{Add \& Norm}})$



The Decoder

$\boxed{\text{Masked Multi-Head Attention}} = \text{MaskedMultiHeadAtt}(\mathbf{H}_i^{dec}, \mathbf{H}_i^{dec}, \mathbf{H}_i^{dec})$

$\boxed{\text{Add \& Norm}} = \text{LayerNorm}(\boxed{\text{Multi-Head Attention}} + \mathbf{H}_i^{dec})$

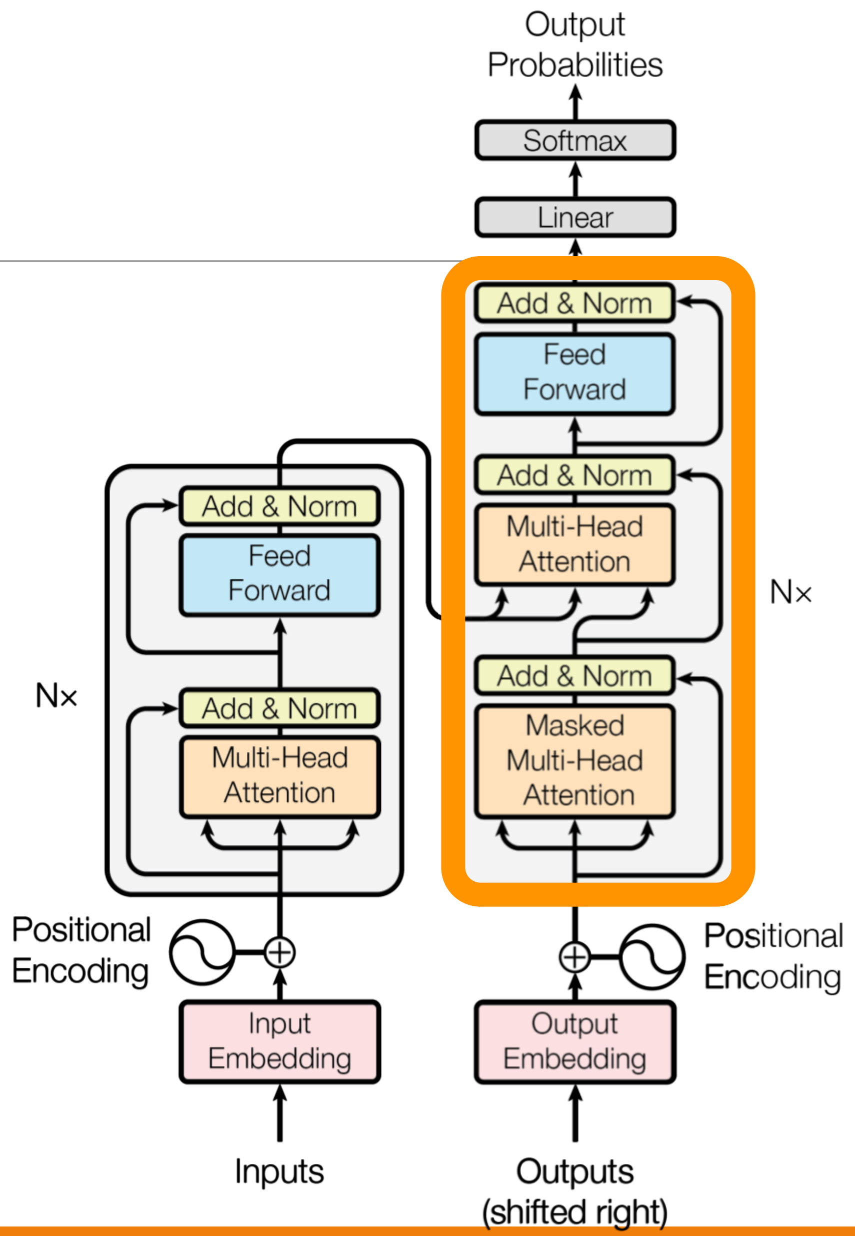
$\boxed{\text{Enc-Dec Multi-Head Attention}} = \text{MultiHeadAtt}(\mathbf{H}_i^{dec}, \mathbf{H}_i^{enc}, \mathbf{H}_i^{enc})$

$\boxed{\text{Add \& Norm (2)}} = \text{LayerNorm}(\boxed{\text{Multi-Head Attention}} + \boxed{\text{Add \& Norm}})$

$\boxed{\text{Feed Forward}} = max(0, \boxed{\text{Add \& Norm (2)}}\, \mathbf{W}_1 + b_1)\mathbf{W}_2 + b_2$

$\boxed{\text{Add \& Norm (3)}} = \text{LayerNorm}(\boxed{\text{Feed Forward}} + \boxed{\text{Add \& Norm (2)}})$

$\mathbf{H}_{i+1}^{dec} = \boxed{\text{Add \& Norm (3)}}$



Strengths of the Transformer Architecture

Training is easily parallelizable

- Larger models can be trained efficiently.

Does not “forget” information from earlier in the sequence.

- Any position can attend to any position.

What are some of its weaknesses?

Knowledge Check

Draw a “map” or table comparing & contrasting the following LMs that we talked about:

Count-based LMs

Maxent/Logistic Regression LMs

Simple (Forward) NNs

Simple RNNs

Seq2Seq RNNs

Transformers

Submit on Blackboard after class