CS 505: Introduction to Natural Language Processing

Wayne Snyder
Boston University

Lecture 18 – Intro to Transformers: Residual Connections; Attention & BRNNs; Positional Encodings; Transformer Architecture

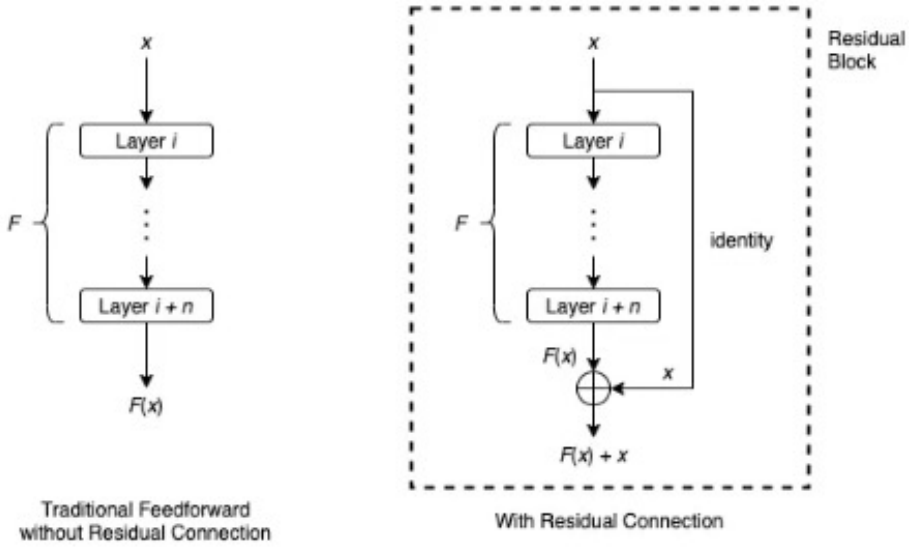
Lecture Plan

Introduction to the Transformer Architecture

- One more advanced feature of NNs: Residual Connections
- Attention and BRNNs (brief review)
- Back to fixed-length sequences: Positional Encodings
- Building the Transformer Architecture layer by layer

One more feature…. Residual Connections

Residual Connections are connections which skip a layer!

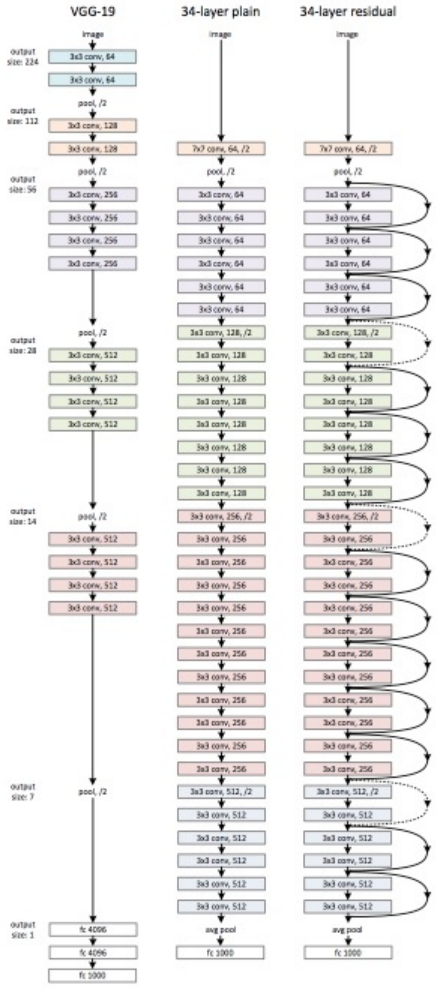


The main purpose of such a “short-circuit” connection is to provide additional pathways for backpropagation.

These were first tried to avoid vanishing gradients in very deep networks used for image classification….

One more feature…. Residual Connections

The deeper the network, the worse the performance, so the notion of a Residual Network was developed, and led to improved performance:



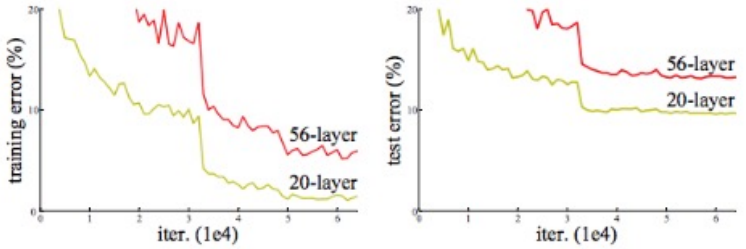
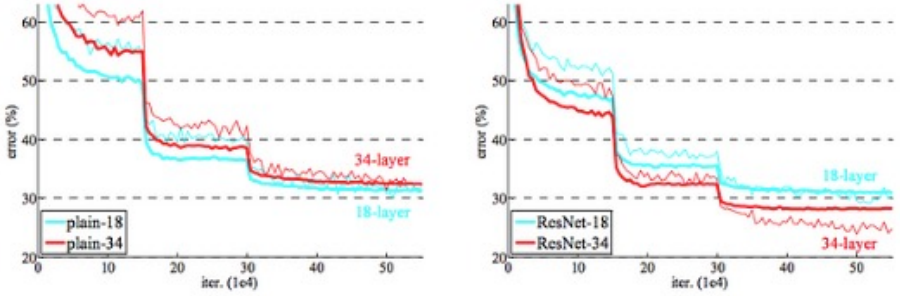


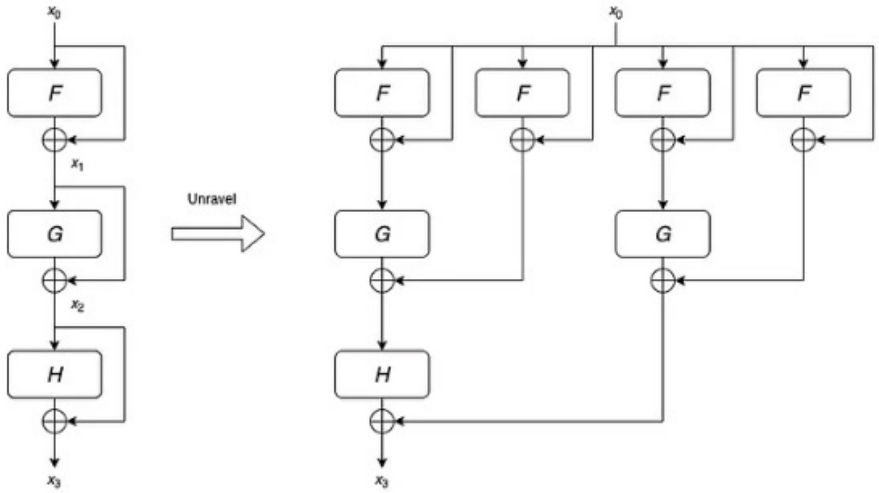
Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.



One more feature…. Residual Connections

Why is it (sometimes) a good idea?

It provides alternate pathways for the forward pass, and shorter pathways for the backpropagation:



$$
\begin{aligned}
x_3 &= H(x_2) + x_2 \\
&= H(G(x_1) + x_1) + G(x_1) + x_1 \\
&= H(G(F(x_0) + x_0) + F(x_0) + x_0) + G(F(x_0) + x_0) + F(x_0) + x_0
\end{aligned}
$$

One more feature…. Residual Connections

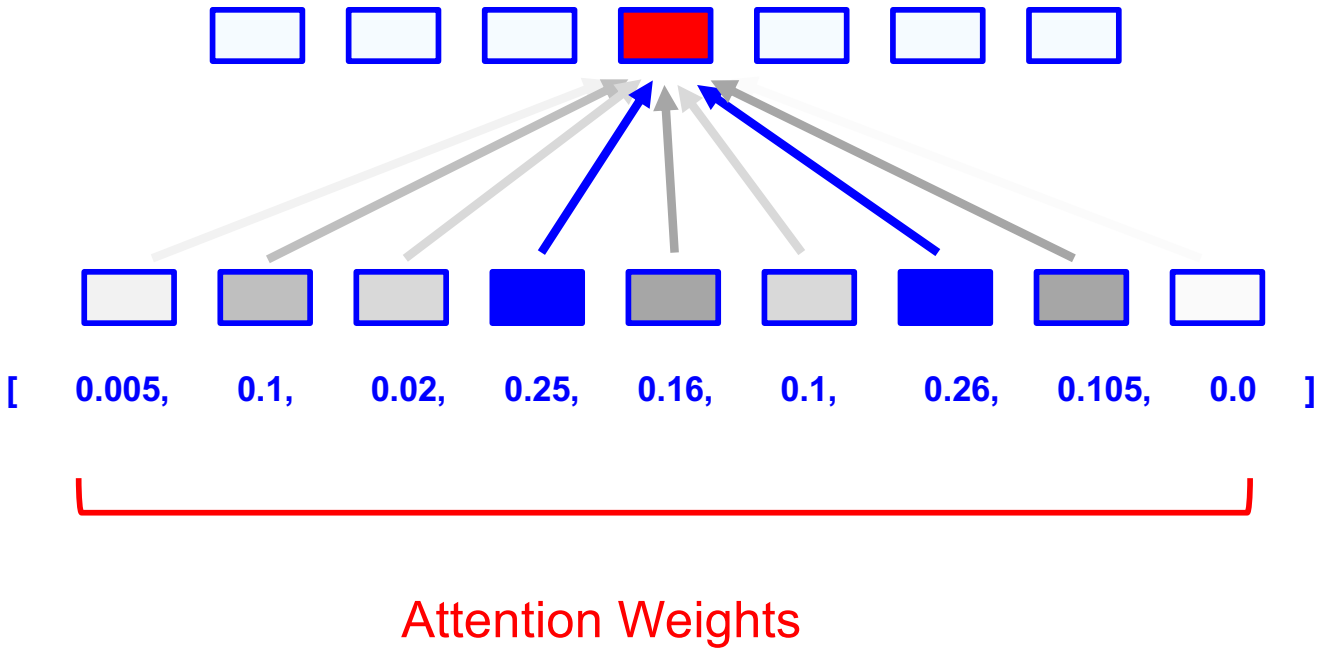
Residual connections are made in Pytorch are by just adding a few assignment statements!

- Residual connections can be made before the non-linear activation function, or after.
- Aggregation of residual vectors is typically done by addition, but concatenation is also possible (changing the width of the next layer).

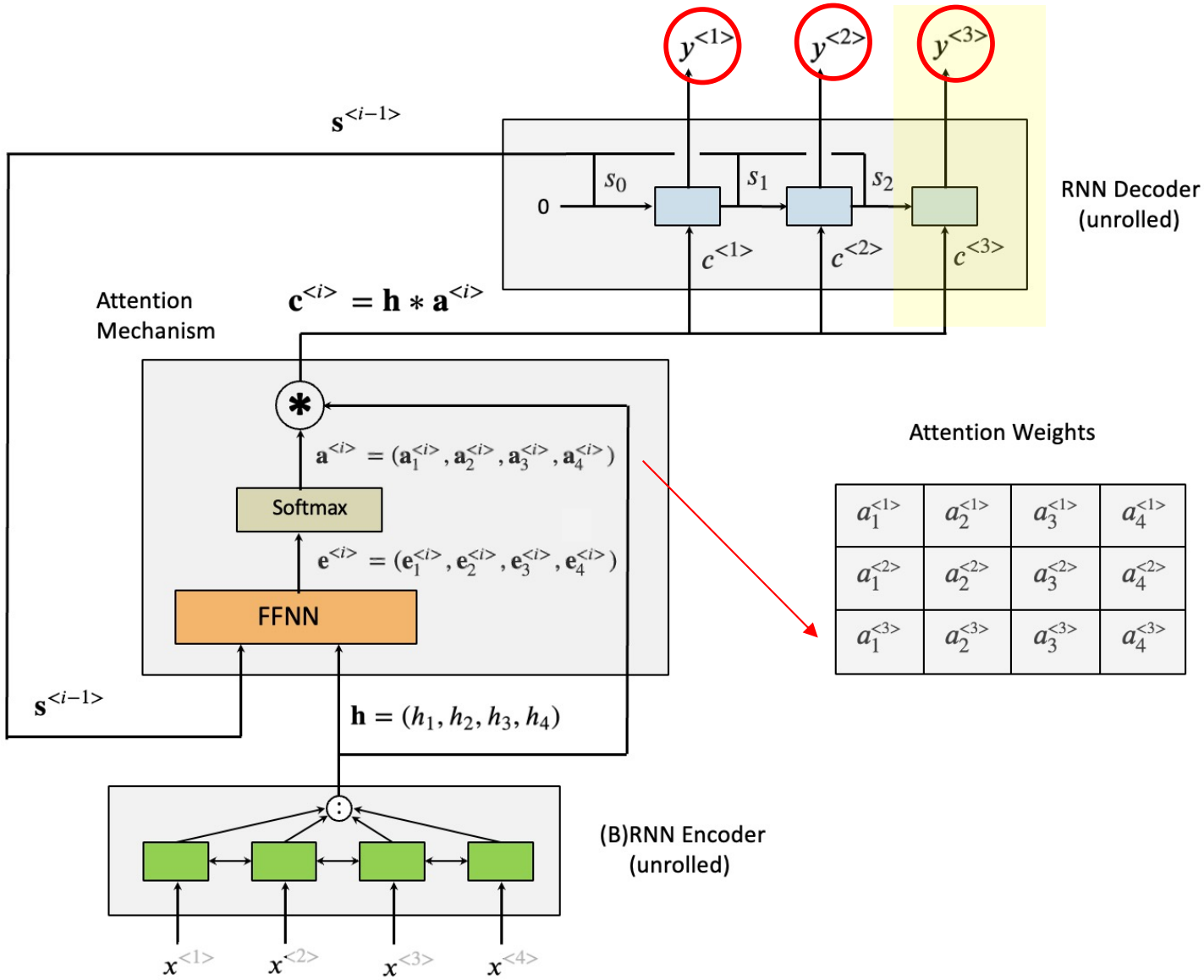
```
class ResidualBlock(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super().__init__()
        self.fc1  = nn.Linear(input_dim, hidden_dim)
        self.fc2  = nn.Linear(hidden_dim, input_dim)
        self.relu = nn.ReLU()

    def forward(self, x):
        residual1 = x                  # Original input for residual connection
        out = self.fc1(x)              # First fully connected layer
        out = self.relu(out)
        residual2 = out                # Output of first layer for residual connection: AFTER Relu
        out = self.fc2(out)            # Second fully connected layer
        out += residual1 + residual2   # Adding the residuals: BEFORE Relu
        out = self.relu(out)
        return out
```

The Attention Mechanism

Recall: Attention refers the ability to focus on particular words in the backward and forward context; the pattern of what words matter in which context can be learned by the network. The pattern can be represented by a probability distribution over the sequence of input tokens:

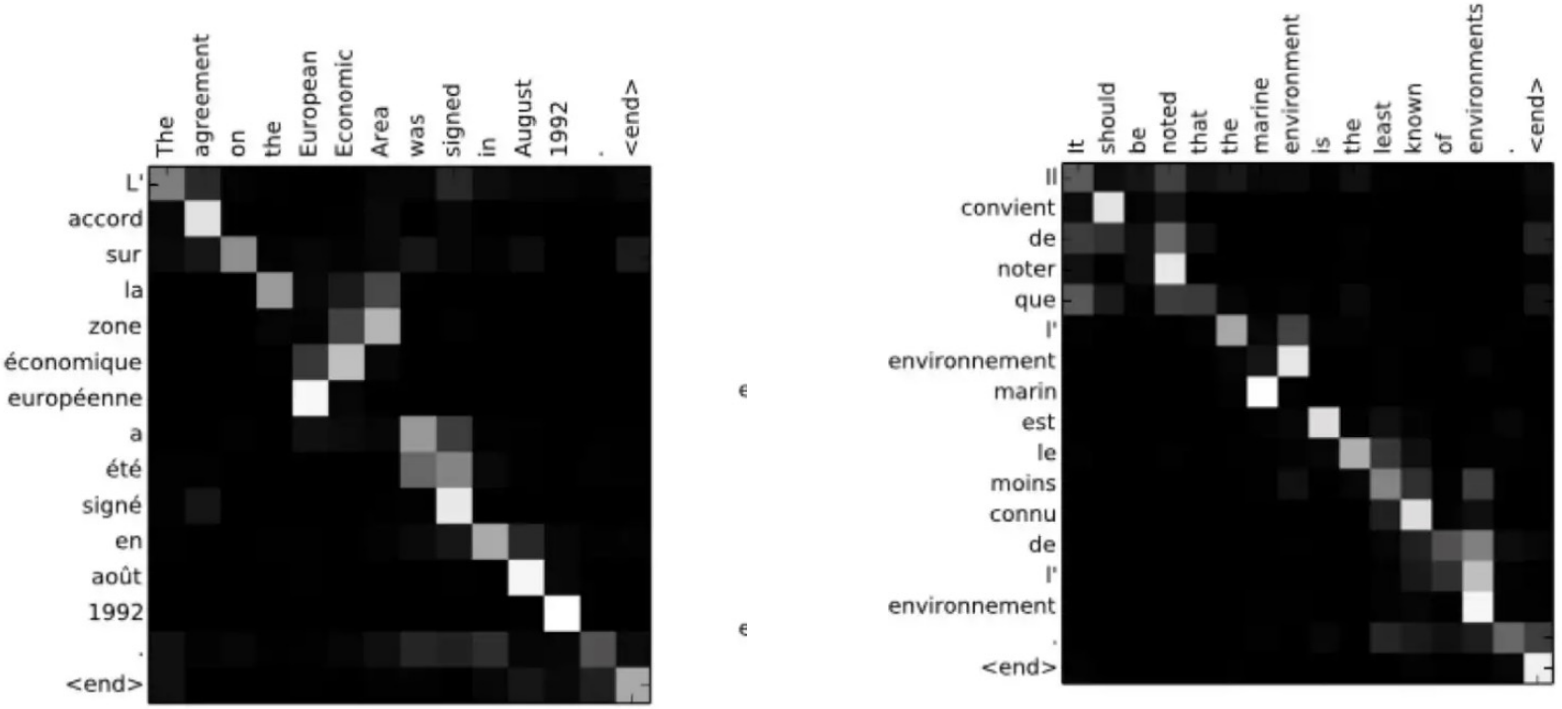


The Attention Mechanism plus BRNNs



The Attention Mechanism

Displaying the activation matrix shows how attention was applied to the translation:



Attention vs. BRNNs

BUT, it seemed that the ”vanishing gradients” problem prevented Attention from performing as well as expected!

SO… researchers went back to the use of fixed-length sequences, NOT using RNNs….

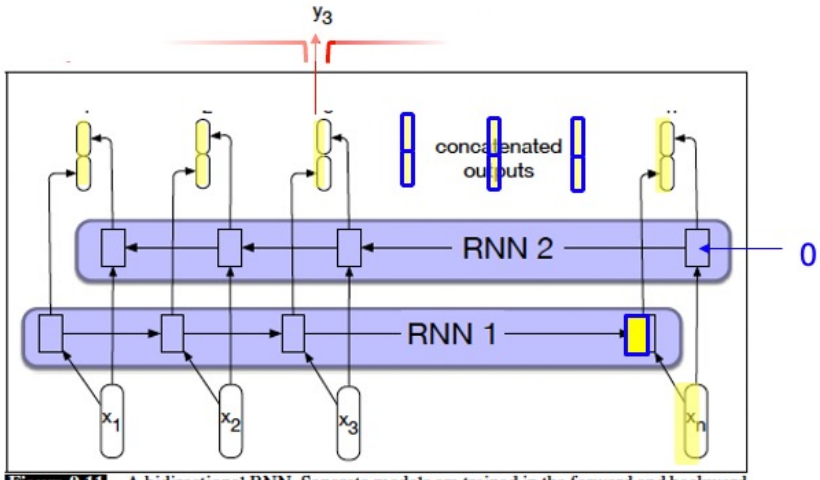
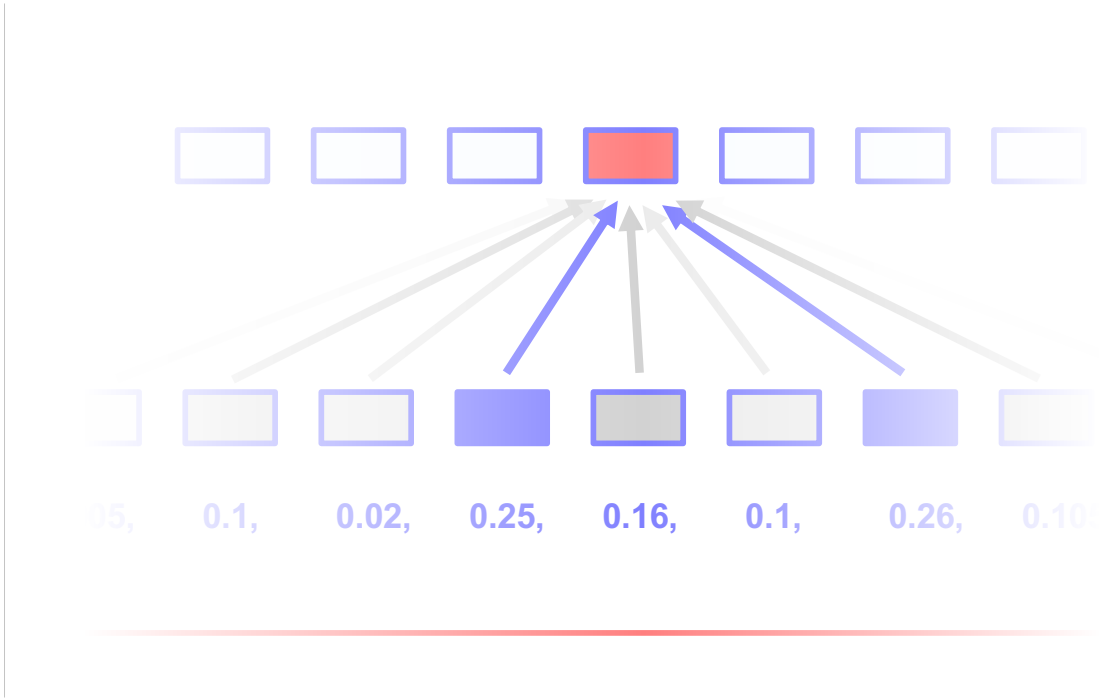


Figure 9.11 A bidirectional RNN. Separate models are trained in the forward and backward directions, with the output of each model at each time point concatenated to represent the bidirectional state at that time point.

Attention vs. BRNNs

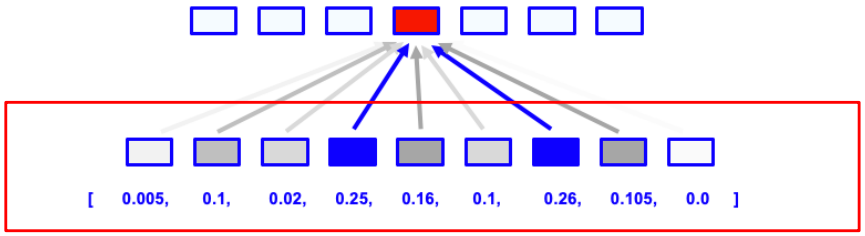
BUT, it seemed that the ”vanishing gradients” problem prevented Attention from performing as well as expected!



SO… researchers went back to the use of fixed-length sequences, NOT using RNNs….

The Return of Fixed-Length Sequences

The focus on the Attention mechanism make researchers give up on the notion of Recurrent Networks with arbitrary-length sequences, and go back to fixed-length sequences:



In the original transformers paper, the maximum size was 512 tokens. GPT-3 used sequences of length 2048. There is no theoretical upper-bound, just a matter of the complexity of the model and the time to train it.

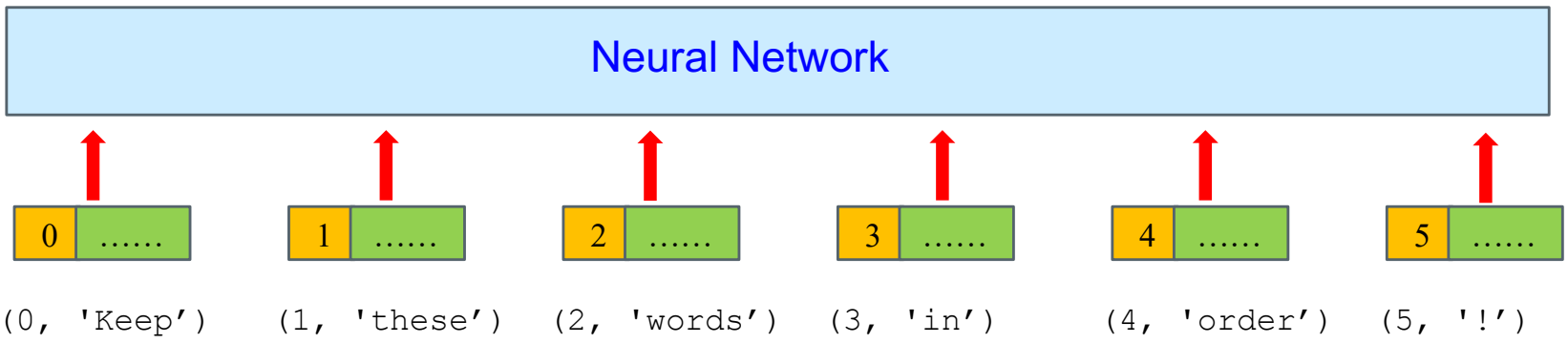
But recall that a neural network does not care what order its input is in, and Attention does not inherently represent ordering, but we need to represent order, so:

How do we encode the order of the words in a sentence for a neural network?

Positional Encodings

The first thing you might try as a positional encoding is simply indices, aggregated with the embeddings:

“Keep these words in order !”



But this really doesn’t work well!

A simple linear sequence:

- Doesn’t generalize well to arbitrary lengths of sentences,
- Doesn’t capture the relative position of words in sentences of different lengths.
- Neural networks work better with continuous (floating point) rather than discrete (integer) representations.

Positional Encodings

There are a number of different positional encodings used in current transformers.

Essentially, they all encode the position using an “positional embedding vector.”

The original design used a truly bizarre encoding using sines and cosines:

Sequence	Index of token, k	Positional Encoding Matrix with d=4, n=100 i=0	i=0	i=1	i=1
I	0	$P_{00}=\sin(0)=0$	$P_{01}=\cos(0)=1$	$P_{02}=\sin(0)=0$	$P_{03}=\cos(0)=1$
am	1	$P_{10}=\sin(1/1)=0.84$	$P_{11}=\cos(1/1)=0.54$	$P_{12}=\sin(1/10)=0.10$	$P_{13}=\cos(1/10)=1.0$
a	2	$P_{20}=\sin(2/1)=0.91$	$P_{21}=\cos(2/1)=-0.42$	$P_{22}=\sin(2/10)=0.20$	$P_{23}=\cos(2/10)=0.98$
Robot	3	$P_{30}=\sin(3/1)=0.14$	$P_{31}=\cos(3/1)=-0.99$	$P_{32}=\sin(3/10)=0.30$	$P_{33}=\cos(3/10)=0.96$

Positional Encoding Matrix for the sequence ‘I am a robot’

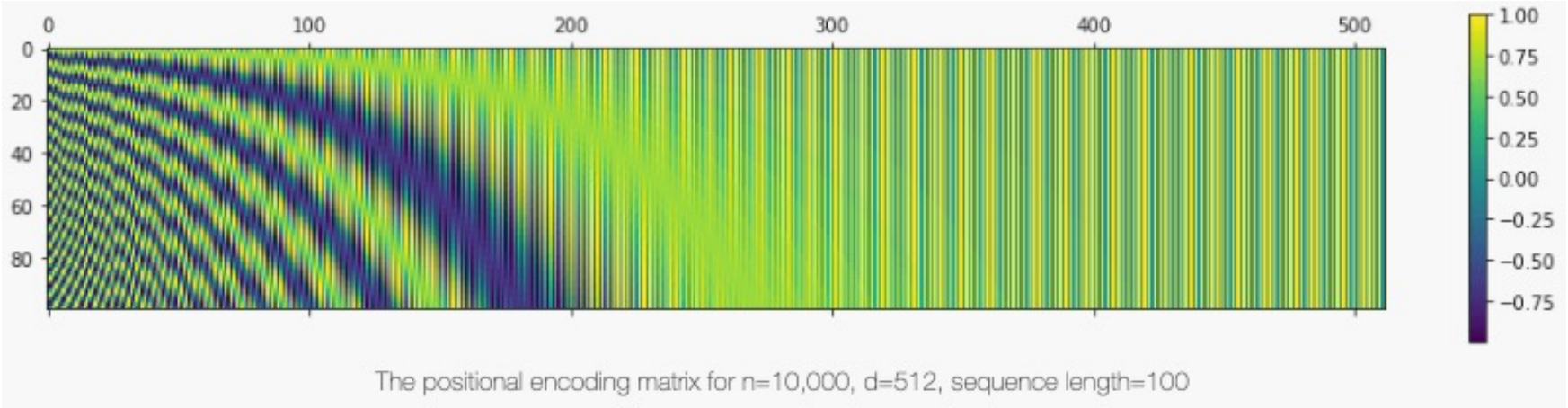
Dimension of the embedding

```
[[ 0.          1.          0.          1.        ]
 [ 0.84147098  0.54030231  0.09983342  0.99500417]
 [ 0.90929743 -0.41614684  0.19866933  0.98006658]
 [ 0.14112001 -0.9899925   0.29552021  0.95533649]]
```

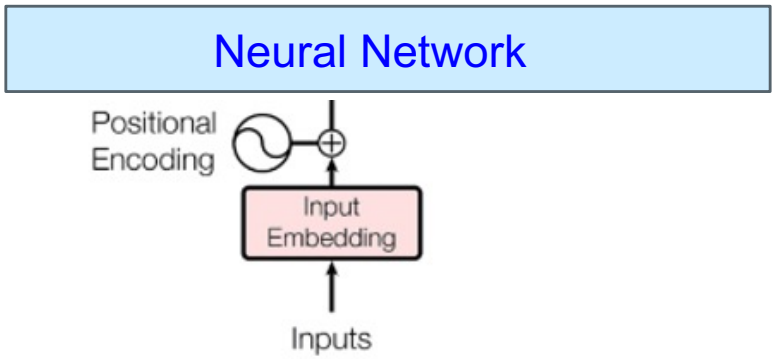
Position in sequence:
0
1
2
3

Positional Encodings

The sine/cosine embedding depends only on the position, not the word, here is a graphical display of the sequence numbers 0 .. 99, with dimension 512:



The positional embeddings and the word embeddings are aggregated by adding them together before entering the network:



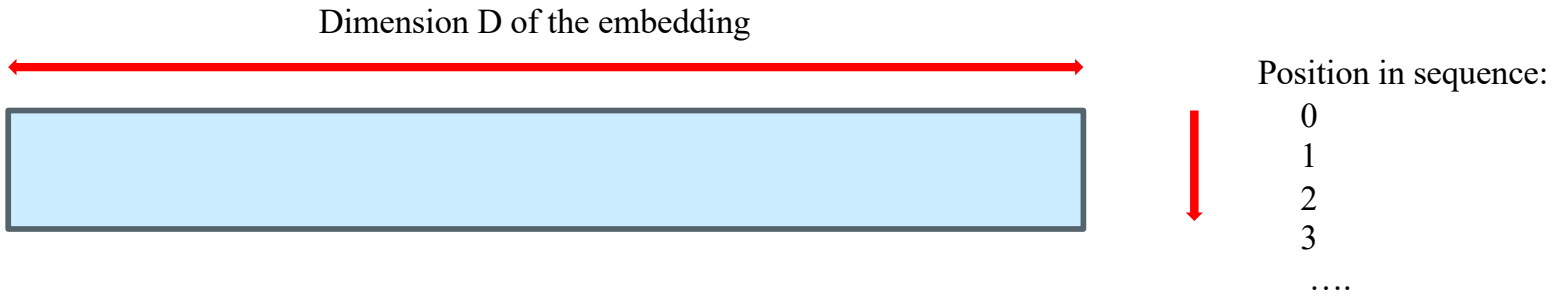
Advantages of this bizarre scheme are:

- Continuous representation as floats;
- Positional embeddings are in useful range [0..1];
- Each position has a unique encoding which represents the relative position of words.

Positional Encodings

The second major positional encoding, used by BERT and GPT, are Learned Positional Embeddings.

There is, again, an embedding matrix, which is initialized randomly:

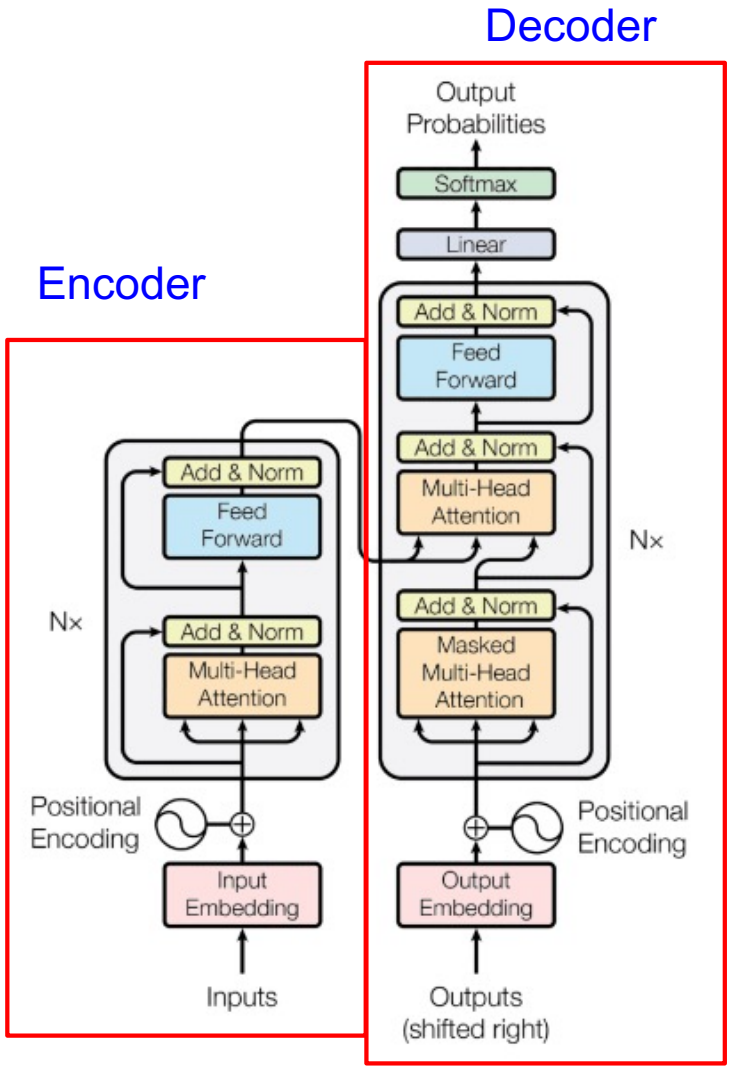


These are vector embeddings which are learned during training, just as with word embeddings. The key points here are:

- The position is simply a ”code” which identifies each position by a distinct location in the vector space with D dimensions;
- During training, the network can learn a positional embedding which represents useful information about token positions;
- Because these are learned, they can represent more complex relationships than sinusoidal embeddings, which are static.
- Just like sinusoidal embeddings, they are aggregated with word embeddings by addition.

Finally, The Transformer Architecture

The Transformer architecture puts together many of the ideas we've been discussing for the last week into a multi-stage encoder and a multi-stage decoder.



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

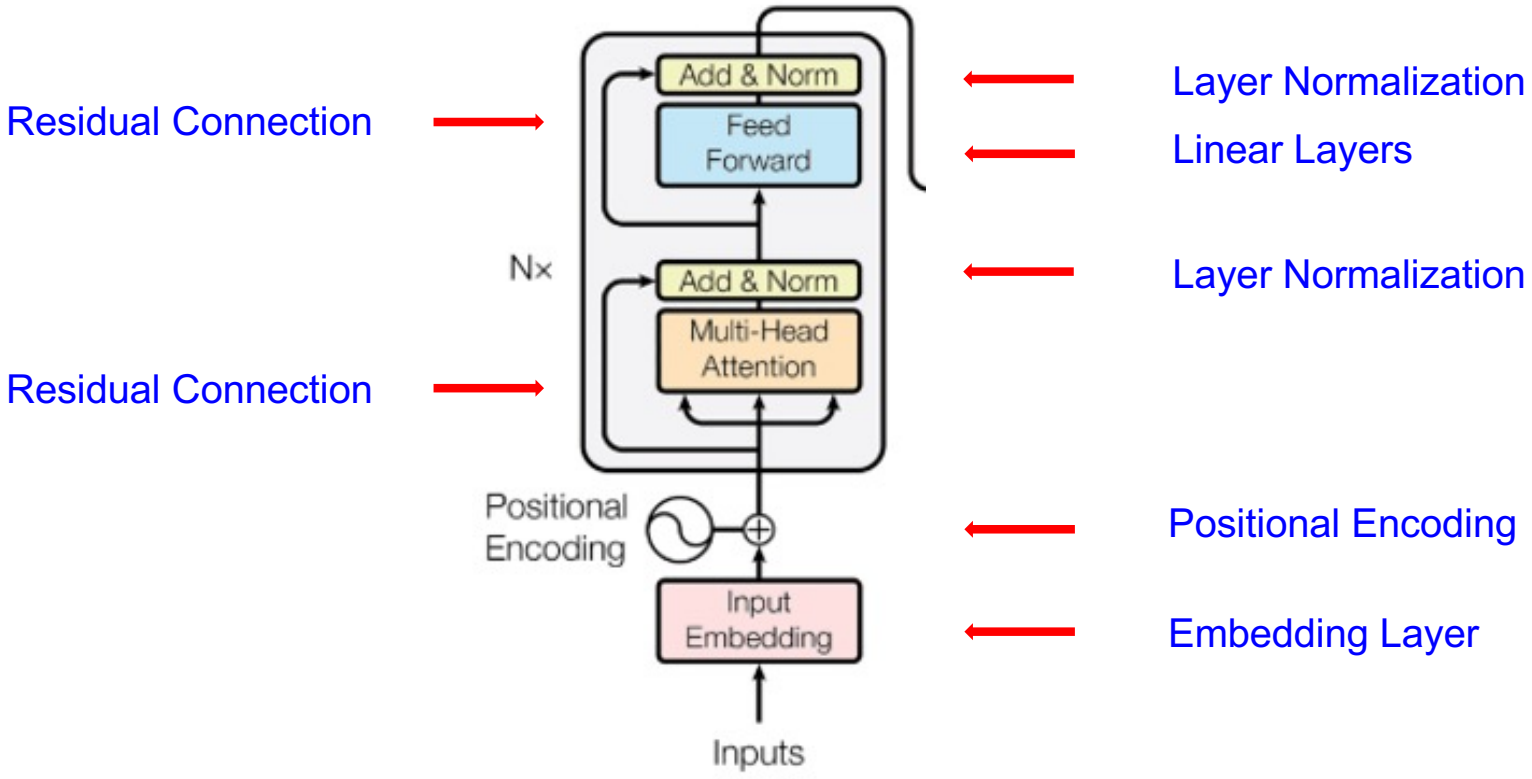
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

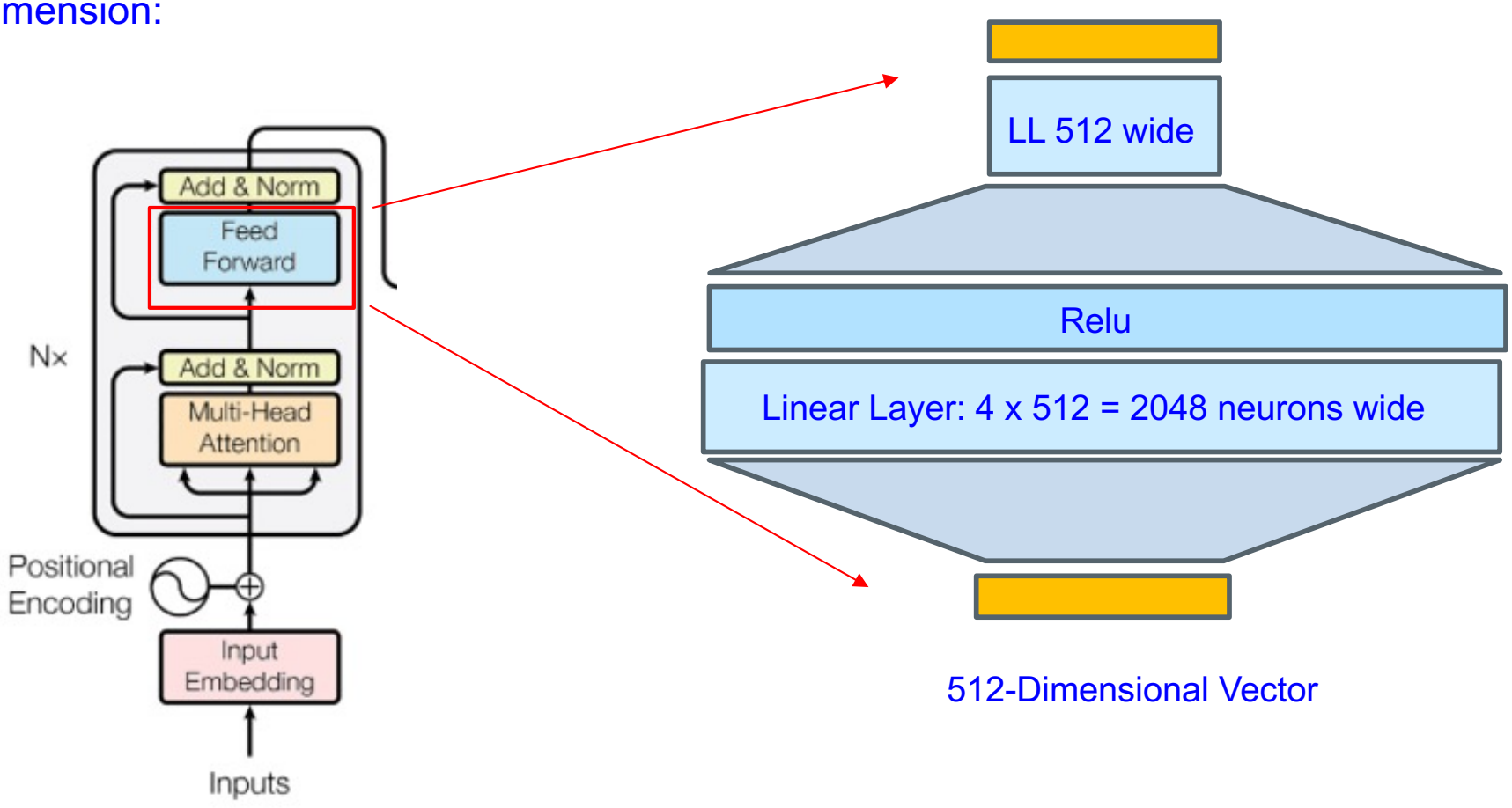
Transformer: Encoder

We already know many of the pieces which make up the encoder:



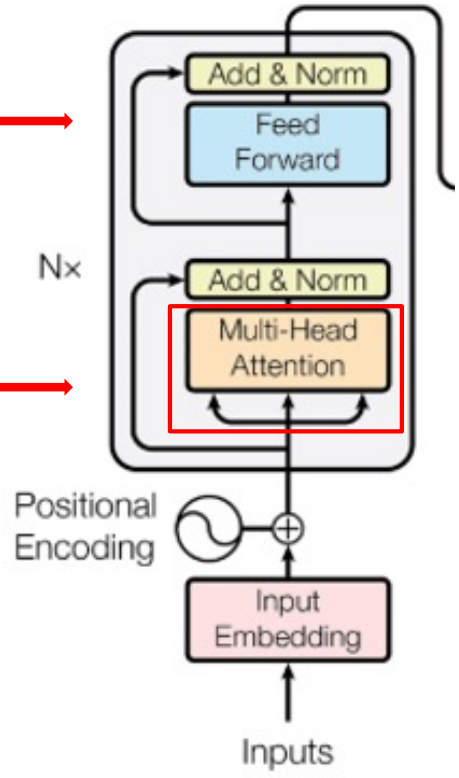
Transformer: Encoder

The Feed-Forward Network has two layers, which expand and then contract the hidden dimension:



Transformer: Multi-Head Attention

The center of the design is the Attention mechanism, here Multi-Head Attention….



Residual Connection

Residual Connection

Layer Normalization

Linear Layers

Layer Normalization

Positional Encoding

Embedding Layer with 512 dimensions

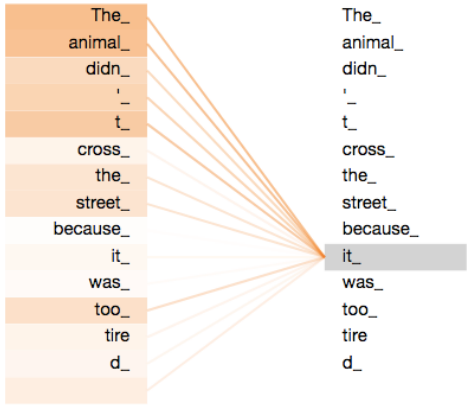
All the data paths shown are the width of a word embedding = 512, and each path is duplicated 512 times, once for each word in the input sequence!

The input sequence is 512 vectors, each 512 floats wide.

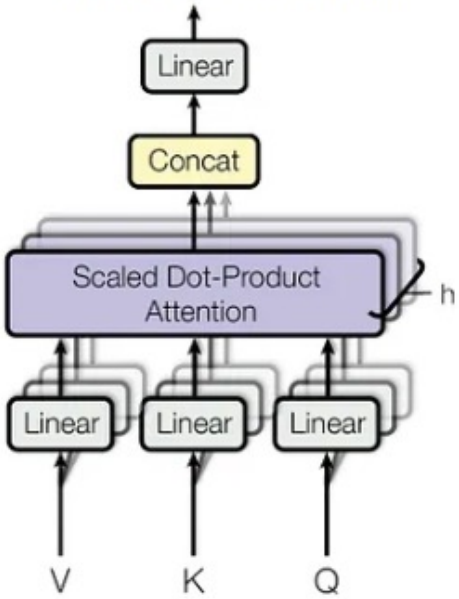
Transformer: Multi-Head Attention

The basic mechanism at work here is self-attention, where the input is processed to determine the dependencies between different words.

Self-attention is implemented by a series of linear transformations, scaled, and then softmaxed to produce the probability distribution which tells us how much each word depends on other words in the same sequence.

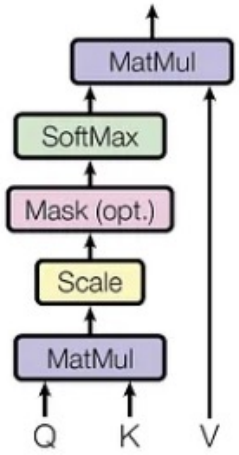


Self-Attention



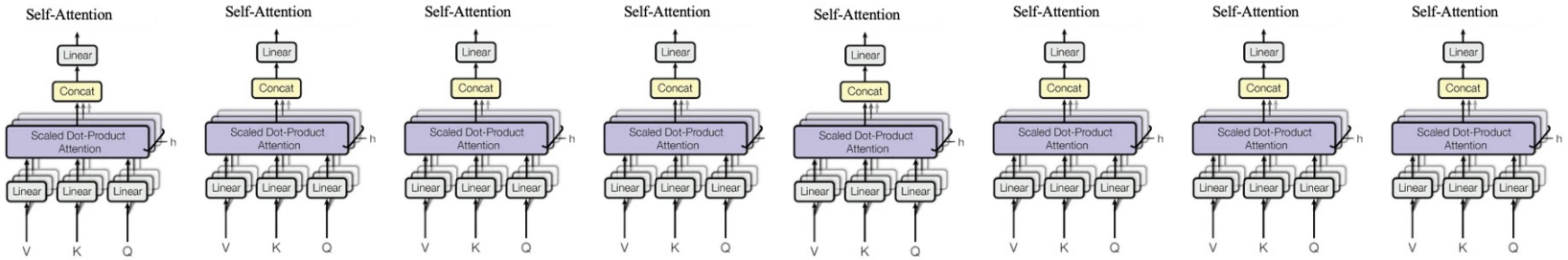
$$\text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Scaled Dot-Product Attention

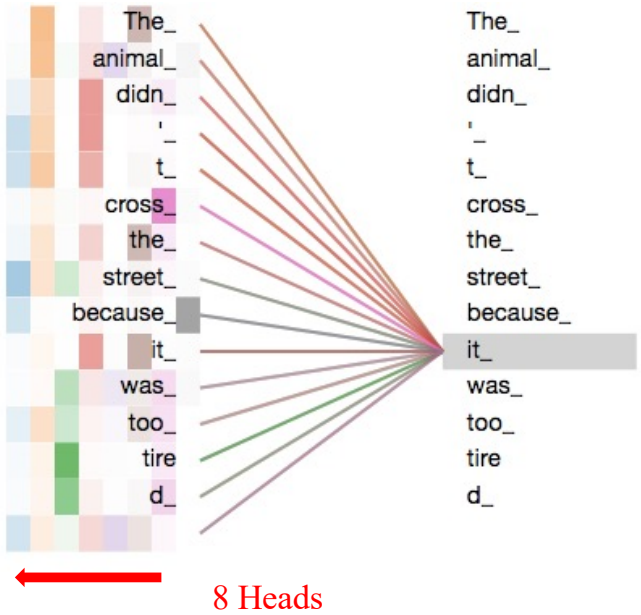


Transformer: Multi-Head Attention

Multi-Head Attention applies this self-attention mechanism as 8 (or more) self-attention Heads:



This allows the encoder to try to understand 8 different kinds of dependencies among the words in the input.



The original design applied 8 self-attention heads to sequences of length 512.

512 words

Transformer: Encoder

Then, this encoder layer is stacked 6, 8, or more layers deep!