New:

\[ E_s = \text{sum of LSTM states} \]
This idea of passing each cell of the decoder a weighted sum of the encoder states is called **attention**.

Different words in the output “pay attention” to different words in the input.
Review: “Attention Is All You Need”

A 2017 paper that introduced the **Transformer** model for machine translation

- Has no recurrent networks!
- **Only** uses attention

**Motivation:**

- RNN training is hard to parallelize since the previous word must be processed before next word
  - Transformers are trivially parallelizable
- Even with LSTMs / GRUs, preserving important linguistic context over **very** long sequences is difficult
  - Transformers don’t even try to remember things (every step looks at a weighted combination of **all** words in the input sentence)
Review: Transformer Model Overview

• The Transformer model breaks down into Encoder and Decoder blocks.

• At a high level, similar to the seq2seq architecture we’ve seen already...

• ...but there are no recurrent nets inside the Encoder and Decoder blocks!
• The Transformer model breaks down into Encoder and Decoder blocks.

• At a high level, similar to the seq2seq architecture we’ve seen already...

• ...but there are no recurrent nets inside the Encoder and Decoder blocks!

• For better performance, often stack multiple Encoder and Decoder blocks (deeper network)

Transformer Model Overview

- Let's look at what goes on inside one of these Encoder blocks

These per-word output vectors are analogous to the LSTM hidden states from the seq2seq2 model

- They should capture "what information about the input sentence is relevant to translating this word?"
Encoder Block Map

- Encoder block breaks down into two main parts: Self-Attention, and Feed Forward layers.

Encoder Block Map

• Encoder block breaks down into two main parts: Self-Attention, and Feed Forward layers.

• Self-Attention layer is applied to each word individually.

Encoder Block Map

- Encoder block breaks down into two main parts: Self-Attention, and Feed Forward layers.
- Self-Attention layer is applied to each word individually.

*Let’s revisit self-attention in detail!*
Review: Attention types

\[ \alpha_{t,i} = \text{align}(y_t, x_i) = \frac{\exp(\text{score}(s_{t-1}, h_i))}{\sum_{i'=1}^{n} \exp(\text{score}(s_{t-1}, h_{i'}))} \]

Softmax of some predefined alignment score.

How well two words \( y_t \) and \( x_i \) are aligned.

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Citation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self-Attention(&amp;)</td>
<td>Relating different positions of the same input sequence. Theoretically the self-attention can adopt any score functions above, but just replace the target sequence with the same input sequence.</td>
<td>Cheng2016</td>
</tr>
<tr>
<td>Global/Soft</td>
<td>Attending to the entire input state space.</td>
<td>Xu2015</td>
</tr>
<tr>
<td>Local/Hard</td>
<td>Attending to the part of input state space; i.e. a patch of the input image.</td>
<td>Xu2015; Luong2015</td>
</tr>
</tbody>
</table>
Self-Attention: Input’s attention on itself

What do we do next?

The animal didn’t cross the street because it was too tired.

Self-Attention: Input’s attention on itself

What do we do next?
Self-Attention: Input’s attention on itself
Self-Attention: Overview

• **The big idea:** Self-attention computes the output vector $z_i$ for each word via a weighted sum of vectors extracted from each word in the input sentence.

• Here, self-attention learns that “it” should pay attention to “the animal” (i.e. the entity that “it” refers to).

• Why the name **self**-attention? This describes attention that the input sentence pays to itself.

Self-Attention: Sketch
Self-Attention: Overview

How it works:
1. To determine how much attention a word should pay to each other other, we compute a query vector for the word and compare it to a key vector for every other word...
Self-Attention: Overview

How it works:

1. To determine how much attention a word should pay to each other other, we compute a query vector for the word and compare it to a key vector for every other word...
How it works:
1. To determine how much attention a word should pay to each other other, we compute a query vector for the word and compare it to a key vector for every other word... to compute our alignment score
Self-Attention: Overview

How it works:

1. To determine how much attention a word should pay to each other other, we compute a query vector for the word and compare it to a key vector for every other word... to compute our alignment score

2. To produce the output vector, we sum up the value vectors for each word, weighted by the score we computed in step 1
Self-Attention: Details

![Diagram showing self-attention with input, embedding, thinking, and machines.]
Self-Attention: Details

Extract 3 vectors from each word embedding

Self-Attention: Details

Extract 3 vectors from each word embedding

Each vector is obtained by multiplying the embedding with the respective weight matrix.

How do we get these weight matrices?

These matrices are the trainable parameters of the network.

Self-Attention: Details

Computing self-attention for “Thinking”

What do we calculate next?

Self-Attention: Details

Computing self-attention for “Thinking”

1. **Score**: Dot product query vector for “Thinking” \((q_1)\) with the key vectors of each word in the sentence \((k_1, k_2, \ldots, k_n)\).
Computing self-attention for “Thinking”

1. **Score**: Dot product query vector for “Thinking” ($q_1$) with the key vectors of each word in the sentence ($k_{1,2,...,n}$).

What this is essentially asking is: *How much should “Thinking” pay attention to each other word in the sequence?*

Query vectors are asking the question and key vectors respond.

What do we calculate next?

Self-Attention: Details

Computing self-attention for “Thinking”

1. **Score:** Dot product query vector for “Thinking” \((q_1)\) with the key vectors of each word in the sentence \((k_1, k_2, \ldots, k_n)\).

2. **Scale:** Divide each score by square root of key vector dimensionality. Results in more stable gradients.
Self-Attention: Details

Computing self-attention for “Thinking”

1. **Score**: Dot product query vector for “Thinking” \((q_1)\) with the key vectors of each word in the sentence \((k_{1,2,...,n})\).
2. **Scale**: Divide each score by square root of key vector dimensionality. Results in more stable gradients.
3. **Softmax**: Apply softmax layer.

Self-Attention: Details

Computing self-attention for “Thinking”

1. **Score**: Dot product query vector for “Thinking” \((q_1)\) with the key vectors of each word in the sentence \((k_1,2,...,n)\).
2. **Scale**: Divide each score by square root of key vector dimensionality. Results in more stable gradients.
3. **Softmax**: Apply softmax layer.

By applying softmax, we transform the scores into attention weights.

What do we calculate next?
Self-Attention: Details

Computing self-attention for “Thinking”

1. **Score**: Dot product query vector for “Thinking” (q₁) with the key vectors of each word in the sentence (k₁,₂,...,n).
2. **Scale**: Divide each score by square root of key vector dimensionality. Results in more stable gradients.
3. **Softmax**: Apply softmax layer.
4. **Weighting**: Multiply value vector of each word in the sentence (v₁,₂,...,n) with the respective softmax values.
Computing self-attention for “Thinking”

1. **Score**: Dot product query vector for “Thinking” ($q_1$) with the key vectors of each word in the sentence ($k_{1,2,…,n}$).
2. **Scale**: Divide each score by square root of key vector dimensionality. Results in more stable gradients.
3. **Softmax**: Apply softmax layer.
4. **Weighting**: Multiply value vector of each word in the sentence ($v_{1,2,…,n}$) with the respective softmax values.

The idea here is that the value vectors store the contextual information that each word provides.
Self-Attention: Details

Computing self-attention for “Thinking”

1. **Score:** Dot product query vector for “Thinking” \((q_1)\) with the key vectors of each word in the sentence \((k_{1,2,...,n})\).
2. **Scale:** Divide each score by square root of key vector dimensionality. Results in more stable gradients.
3. **Softmax:** Apply softmax layer.
4. **Weighting:** Multiply value vector of each word in the sentence \((v_{1,2,...,n})\) with the respective softmax values.
5. **Sum:** Sum up weighted value vectors \((v_{1,2,...,n})\) into one final self-attention vector for “Thinking” \((z_1)\)

---

Self-Attention: Details

Computing self-attention for “Thinking”

1. **Score**: Dot product query vector for “Thinking” ($q_1$) with the key vectors of each word in the sentence ($k_{1,2,...,n}$).
2. **Scale**: Divide each score by square root of key vector dimensionality. Results in more stable gradients.
3. **Softmax**: Apply softmax layer.
4. **Weighting**: Multiply value vector of each word in the sentence ($v_{1,2,...,n}$) with the respective softmax values.
5. **Sum**: Sum up weighted value vectors ($v_{1,2,...,n}$) into one final self-attention vector for “Thinking” ($z_1$)

We are weighting the context provided by each word by the amount of attention we should pay.

Self-Attention as a Matrix Computation

Each row of $X$ is a word embedding of a word in our sentence.

What would be the dimensions of the weight matrices to calculate the query, key, and value?

What would be the dimensions of the query, key, and value matrices?

Apply the steps of calculating attention weights on the query and key matrices.

What is the dimension of attention weight matrix?

Multiply the attention weight matrix to value matrix produce the output matrix.

What are the dimensions of output matrix?

Get your pens/papers or tablets ready!
Self-Attention as a Matrix Computation

Each row of $X$ is a word embedding of a word in our sentence.
Each row of $X$ is a word embedding of a word in our sentence.

Matrix multiplication with $W^Q$, $W^K$, and $W^V$ gives us matrices $Q$, $K$, and $V$, where the $i^{th}$ rows of each matrix represent the vectors $q_i$, $k_i$, and $v_i$. 

Self-Attention as a Matrix Computation

Matrix multiplying $Q$ and the transpose of $K$ calculates all the “score” values.

Dividing by $\sqrt{d_k}$ correctly scales values.

Self-Attention as a Matrix Computation

Matrix multiplying $Q$ and the transpose of $K$ calculates all the “score” values.

Dividing by $\sqrt{d_k}$ correctly scales values.

The result is a $Z$ matrix where the $i$th row represents the self-attention vector $z_i$.

Multiplying the resulting vector with $V$ properly weighs the $v_i$ vectors.

Encoder Block Map

- Encoder block breaks down into two main parts: Self-Attention, and Feed Forward layers.
- Self-Attention layer is applied to each word individually.

Encoder Block Map

- Encoder block breaks down into two main parts: Self-Attention, and Feed Forward layers.
- Self-Attention layer is applied to each word individually.
- Feed Forward layer is applied to each word individually.

Encoder Block Map

- Encoder block breaks down into two main parts: Self-Attention, and Feed Forward layers.

- Self-Attention layer is applied to each word individually.

- Feed Forward layer is applied to each word individually.

- The outputs of the feed forward layer are then passed as the inputs of the next encoder block.

Encoder Block Map

- Encoder block breaks down into two main parts: Self-Attention, and Feed Forward layers.
- Self-Attention layer is applied to each word individually.
- Feed Forward layer is applied to each word individually.
- The outputs of the feed forward layer are then passed as the inputs of the next encoder block.
- *But we forgot about something...*

What are we missing?

Hint: Remember – we are not using RNNs anymore.
Have we neglected/lost any information about the original input sequence?
Positional Encodings

- Instead of passing Embedding vector to encoder, we pass Embedding with Time Signal vector.

- Positional Encoding is embedding_size vector that encodes information about the position of a word in a sequence.

- Positional Encodings can be learned or defined by a fixed function.

- We add the Positional Encoding to the Embedding to get our Embedding with Time Signal vector.
Positional Encodings

Where do these numbers come from?

- Carefully-chosen sinusoidal patterns such that when we add them to the embedding vectors, their dot products with each other reflect the distance between them in the sentence.

More to come on Transformer!

- Multi-headed attention
- Modifications for efficiency
- Decoder

Recap

Seq-to-seq using transformers

- RNNs cannot be parallelized
- Can forget information
- Transformers – Encoder-Decoder with just attention

Encoder module

- Self-attention
- Fully connected layers
- Positional encodings