Multi-Head Self-Attention

- Multiple “heads” analogous to different convolutional filters
- Let $E = \text{[sent len, embedding dim]}$ be the input sentence. This will be passed through three different linear layers to produce three mats:
  - Query $Q = EW^Q$: each token “chooses” what to attend to
  - Keys $K = EW^K$: these control what each token looks like as a “target”
  - Values $V = EW^V$: these vectors get summed up to form the output

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

dim of keys

Vaswani et al. (2017)
Self-Attention

Alammar, *The Illustrated Transformer*

- Example visualization of attention matrix $A$ (from assignment)
- Each row: distribution over what that token attends to. E.g., the first “V” attends very heavily to itself (bright yellow box)
- Your task on the HW: assess if the attentions make sense

Attention Maps

Multi-head Self-Attention

Just duplicate the whole computation with different weights:

Alammar, *The Illustrated Transformer*

1) This is our input sentence
2) We embed each word
3) Split into $h$ heads. We multiply $h$ or $1$ with weight matrices
4) Calculate attention using the resulting $QV/h$ matrices
5) Concatenate the resulting $h$ matrices, then multiply with weight matrix $W^o$ to produce the output of the layer

Multi-head Self-Attention
**Transformers**

**Architecture**

- Alternate multi-head self-attention with feedforward layers that **operate over each word individually**
  
  \[ FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2 \]

- These feedforward layers are where most of the parameters are

- Residual connections in the model: input of a layer is added to its output

- Layer normalization: controls the scale of different layers in very deep networks (not needed in A4)

**Dimensions**

- Vectors: \( d_{\text{model}} \)

- Queries/keys: \( d_k \), always smaller than \( d_{\text{model}} \)

- Values: separate dimension \( d_v \), output is multiplied by \( W^o \) which is \( d_v \times d_{\text{model}} \) so we can get back to \( d_{\text{model}} \) before the residual

- FFN can explode the dimension with \( W_1 \) and collapse it back with \( W_2 \)
  
  \[ FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2 \]

- Positions: \( d_{\text{internal}} \)

- Transformer Architecture

**Transformer Architecture**

<table>
<thead>
<tr>
<th>Model Name</th>
<th>( n_{\text{params}} )</th>
<th>( n_{\text{layers}} )</th>
<th>( d_{\text{model}} )</th>
<th>( n_{\text{heads}} )</th>
<th>( d_{\text{head}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPT-3 Small</td>
<td>125M</td>
<td>12</td>
<td>768</td>
<td>12</td>
<td>64</td>
</tr>
<tr>
<td>GPT-3 Medium</td>
<td>350M</td>
<td>24</td>
<td>1024</td>
<td>16</td>
<td>64</td>
</tr>
<tr>
<td>GPT-3 Large</td>
<td>760M</td>
<td>24</td>
<td>1536</td>
<td>16</td>
<td>96</td>
</tr>
<tr>
<td>GPT-3 XL</td>
<td>1.3B</td>
<td>24</td>
<td>2048</td>
<td>24</td>
<td>128</td>
</tr>
<tr>
<td>GPT-3 2.7B</td>
<td>2.7B</td>
<td>32</td>
<td>2560</td>
<td>32</td>
<td>80</td>
</tr>
<tr>
<td>GPT-3 6.7B</td>
<td>6.7B</td>
<td>32</td>
<td>4096</td>
<td>32</td>
<td>128</td>
</tr>
<tr>
<td>GPT-3 13B</td>
<td>13.0B</td>
<td>40</td>
<td>5140</td>
<td>40</td>
<td>128</td>
</tr>
<tr>
<td>GPT-3 175B or “GPT-3”</td>
<td>175.0B</td>
<td>96</td>
<td>12288</td>
<td>96</td>
<td>128</td>
</tr>
</tbody>
</table>

From GPT-3; \( d_{\text{head}} \) is our \( d_k \)

From Vaswani et al.
### Transformer Architecture

<table>
<thead>
<tr>
<th>description</th>
<th>FLOPs / update</th>
<th>% FLOPs MHA</th>
<th>% FLOPs FFN</th>
<th>% FLOPs attn</th>
<th>% FLOPs logit</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPT setups</td>
<td>4.3E+15</td>
<td>35%</td>
<td>44%</td>
<td>14.8%</td>
<td>5.8%</td>
</tr>
<tr>
<td>760M</td>
<td>2.5E+16</td>
<td>29%</td>
<td>56%</td>
<td>11.2%</td>
<td>3.3%</td>
</tr>
<tr>
<td>1.3B</td>
<td>2.5E+16</td>
<td>29%</td>
<td>56%</td>
<td>11.2%</td>
<td>3.3%</td>
</tr>
<tr>
<td>6.7B</td>
<td>1.1E+17</td>
<td>24%</td>
<td>65%</td>
<td>8.1%</td>
<td>2.4%</td>
</tr>
<tr>
<td>13B</td>
<td>4.1E+17</td>
<td>22%</td>
<td>69%</td>
<td>6.9%</td>
<td>1.6%</td>
</tr>
<tr>
<td>30B</td>
<td>9.0E+17</td>
<td>20%</td>
<td>74%</td>
<td>5.3%</td>
<td>1.0%</td>
</tr>
<tr>
<td>66B</td>
<td>9.5E+17</td>
<td>18%</td>
<td>77%</td>
<td>4.3%</td>
<td>0.6%</td>
</tr>
<tr>
<td>175B</td>
<td>2.4E+18</td>
<td>17%</td>
<td>80%</td>
<td>3.3%</td>
<td>0.3%</td>
</tr>
</tbody>
</table>

Credit: Stephen Roller on Twitter

### Transformers: Position Sensitivity

- If this is in a longer context, we want words to attend *locally*
- But transformers have *no notion of position* by default

Vaswani et al. (2017)

### Transformers: Position Sensitivity

- Encode each sequence position as an integer, add it to the word embedding vector
- Why does this work?

**The ballerina is very excited that she will dance in the show.**

**Transformers**

- Alternative from Vaswani et al.: sines/cosines of different frequencies (closer words get higher dot products by default)

Alammar, *The Illustrated Transformer*
Transformers: Complete Model

- Original Transformer paper presents an **encoder-decoder** model
- Right now we don’t need to think about both of these parts — will return in the context of MT
- Can turn the encoder into a decoder-only model through use of a triangular causal attention mask (only allow attention to previous tokens)

---

Transformer Language Modeling

What do Transformers produce?

- **Encoding of each word** — can pass this to another layer to make a prediction (like predicting the next word for language modeling)
- Like RNNs, Transformers can be viewed as a transformation of a sequence of vectors into a sequence of context-dependent vectors

---

Transformer Language Modeling

- $P(w|\text{context}) = \frac{\exp(w \cdot h_i)}{\sum_{w'} \exp(w' \cdot h_i)}$

- Equivalent to:
  - $P(w|\text{context}) = \text{softmax}(Wh_i)$

- $W$ is a (vocab size) x (hidden size) matrix; linear layer in PyTorch (rows are word embeddings)
**Training Transformer LMs**

- Input is a sequence of words, output is those words shifted by one,
- Allows us to train on predictions across several timesteps simultaneously (similar to batching but this is NOT what we refer to as batching)

\[
\text{loss} = - \log P(w^* | \text{context})
\]

\[
\text{Total loss} = \sum \text{of negative log likelihoods at each position}
\]

\[
\text{loss} = \text{nn.NLLLoss()}
\]

\[
\text{loss} += \text{loss}_\text{fcn}(\log \text{probs}, \text{ex.output.tensor})
\]

\[
\text{[seq len, num output classes]} \quad [\text{seq len}]
\]

**Batching is a little tricky with NLLLoss: need to collapse [batch, seq len, num classes] to [batch * seq len, num classes]. You do not need to batch**

**Batched LM Training**

- Multiple sequences and multiple timesteps per sequence

**A Small Problem with Transformer LMs**

- This Transformer LM as we’ve described it will *easily* achieve perfect accuracy. Why?
- With standard self-attention: “I” attends to “saw” and the model is “cheating”. How do we ensure that this doesn’t happen?
Attention Masking

- We want to prohibit
- Key words
  <s> I saw the dog
  <s> I saw the dog
  Query words
  saw
  the
  dog
- We want to mask out everything in red (an upper triangular matrix)

Implementing in PyTorch

- nn.TransformerEncoder can be built out of nn.TransformerEncoderLayers, can accept an input and a mask for language modeling:

```python
# Inside the module; need to fill in size parameters
layers = nn.TransformerEncoderLayer([...])
transformer_encoder = nn.TransformerEncoder(encoder_layers, num_layers=[...]) [...]
# Inside forward(): puts negative infinities in the red part
mask = torch.triu(torch.ones(len, len) * float('-inf'), diagonal=1)
output = transformer_encoder(input, mask=mask)
```
- You cannot use these for Part 1, only for Part 2

LM Evaluation

- Accuracy doesn’t make sense — predicting the next word is generally impossible so accuracy values would be very low
- Evaluate LMs on the likelihood of held-out data (averaged to normalize for length)
  \[
  \frac{1}{n} \sum_{i=1}^{n} \log P(w_i|w_1, \ldots, w_{i-1})
  \]
- Perplexity: \( \exp(\text{average negative log likelihood}) \). Lower is better
  - Suppose we have probs 1/4, 1/3, 1/4, 1/3 for 4 predictions
  - Avg NLL (base e) = 1.242 Perplexity = 3.464 <= geometric mean of denominators

Takeaways

- Transformers are going to be the foundation for the much of the rest of this class and are a ubiquitous architecture nowadays
- Many details to get right, many ways to tweak and extend them, but core idea is the multi-head self attention and their ability to contextualize items in sequences
- Next: machine translation and seq2seq models (conditional language modeling)