2.3 Transformers

As partially described above, RNNs face the following challenges:

1. Long-term dependencies are hard to learn (gradient vanishing).
2. Hidden states attempt to store information of any length into a vector of fixed size.
3. Recurrence prevents parallel computation during training.

The transformer (Vaswani+ '17) is a sequence to sequence model based entirely on attention. In a transformer, the encoder uses self attention to generate encodings, and the decoder attends to both input sequences and already outputted words. Transformers use several tricks:

- **Self-attention.** In vanilla attention, the query is the last decoder state and the keys and values are both the encoder states. In self-attention, all keys, values, and queries come from the output of the previous layer in the encoder, and each position in the encoder attends to all positions in the previous layer of the encoder. With self-attention, the encoder in a transformer produces a sequence of embeddings, with each embedding capturing the original word and information from other words that were attended to.
- **Multi-headed attention.** Instead of having a single attention function, we can use multiple attention heads so that the model can jointly attend to information from different representation subspaces at different positions (with a single head, averaging inhibits attending to different representations). You can think of each head as a filter or feature map in ConvNets.
- **Normalized dot-product attention.** In practice, dot-product attention is fast and more space efficient than using a single-layer neural net, and so given the matrices Q, K, and V, we use

  $$\text{attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \ . \tag{21}$$

  Graham Neubig, however, notes that this dot-product attention is a bit of a misnomer, since, as seen in Figure 2 (right) in their paper, the keys, values, and queries are multiplied by corresponding matrices, and so it is much more like bilinear dot-product attention.
- **Positional encodings.** Because transformers do not use recurrence or convolution, we must inject some information about the position of tokens into the sequence. We add positional encodings of the same dimension as word embeddings based on sine and cosine functions of different frequencies.
- **Layer normalization.** Computing a gradient requires knowledge of both the previous (layer below) and current activations (layer above), and so there are many interdependencies with layers (e.g., you changed layer l based on $l + 1$, but now you just updated $l + 1$). With layer normalization, we normalize the values in each layer to mean 0 and variance 1. This trick reduces covariate shift (gradient dependencies between each layer), helping the model to converge in fewer iterations.
- **Teacher forcing.** During test time, we still must generate each output sequentially. For training, however, we can employ *teacher forcing*, where we assume we have already gotten all previous outputs correct, allowing us to train in parallel.