CS 182 Lecture 11: Sequence to Sequence Models 7/5

"language model" - assigns probs to sequences of text (to even generate text)
↳ training data: natural sentences
↳ tokenize sentences (one-hot, embeddings).

end token - completes the sequence, learned by LM.
start token - starts sequence, learned by LM.
"finish" rest of sentence - force model to process given segment of sequence.

Conditional LM: instead of a_0 (initial state) $= 0$, (hidden activation)
(CNN encoder, RNN decoder). set $a_0 \leftarrow f(x)$, where $f(x) = CNN(\text{input})$ (essentially, vector encoding of desired content of sequence).
Training data: (image, words)

This approach is modular: encoder/decoder could be whatever. Ex. encoder is RNN.
↳ Sequence to sequence models
→ Typically two separate RNNs, diff weights, trained end-to-end
→ Realistically, stacked RNN layers, uses LSTM/GRU cells, diff. length sequences.
→ utilizations: translation, summarization, Q/A, Text-to-code.
→ Typically we reverse order of tokens to encoder (due to how memory is processed)

Decoding likely sequence: maximize product of all sequence probabilities.
Each seq output: $p(y_{i,t} \mid x_{i,1:T}, y_{i,0:t-1})$ ← conditioned on everything before it.
So $p(y_{i,1:T_y} \mid x_{i,1:T}) = \prod_{t=1}^{T_y} p(y_{i,t} \mid x_{i,1:T}, y_{i,0:t-1})$ ← product of conditional probs (chain rule)

decodings: for M words, M^T seqs for T length. Turns into search problem!
↳ expensive to find optimal seq. using exact method.

Approximate search to find largest prod: avoid super low probs. Target top k probs @ init.
↳ "Beam Search": store k best sequences so far, update each of them.
at each t,
1. for each hypothesis $y_{1:t-1,i}$ that we are tracking: ← k of these
   find top k tokens $y_{t,i,1}, \dots, y_{t,i,k}$
2. sort resulting k^2 length t sequences by total log prob
3. keep top k
4. advance each hypothesis to time t+1 (forward)

→ If top sequences ends abruptly, save it, "remove" it, move on.
→ continue until $t > T$ or # of seqs that end in <EOS> $> N$. ← predetermined.
→ perhaps divide total log prob by T to generate "score" for seq (longer seq → more penalty).

"Bottleneck" problem → for decoder, all information stored in potentially vast encoder → one layer of activations
→ instead, while decoding, "peek" at source sentence.
↳ for each x layer in encoder, generate key vector thru learned function. (what info is present)
↳ for each y layer in decoder, generate query vector thru function (what info we are looking for)
↳ compare query to key to find closest one.

learned, not selected, same throughout layers

Mathematically: $k_t = k(e_t)$, $q_l = q(d_l)$, $s_{t,l} = k_t \cdot q_l$ ← attention score for encoder step to decoder step.
Then, to get max $s_{t,l}$, use softmax (argmax is ideal but not differentiable)
$\alpha_{\cdot,l} = \text{softmax}(s_{\cdot,l})$, $\alpha_{t,l} = \frac{\exp(s_{t,l})}{\sum_{t'} \exp(s_{t',l})}$ ← softmax

★ Network is trying to "pay attention" to most relevant part of input

"send" $a_l = \sum_t \alpha_{t,l} e_t$ ← approx. to "best" e_t for given d_l
↳ i.e. $\hat{y}_l = f(d_l, a_l)$, pass into RNN layer (use for readout), pass into next decoder step (concatenate to hidden state) $\left(\bar{d}_l = \begin{bmatrix} d_{l-1} \\ a_{l-1} \\ x_l \end{bmatrix}\right)$

Attention Variants
→ $k_t = e_t$, $q_l = d_l$, so $s_{t,l} = e_t \cdot d_l$
→ $k_t = W_k e_t$, $q_l = W_q d_l$, so $s_{t,l} = e_t^T W_k^T W_q d_l = e_t^T W_s d_l$ ← learn this.
→ learned-value encoding: $a_l = \sum_t \alpha_t v(e_t)$ ← learn this