CS 182 Lecture 12: Transformers

Idea: Can we transform RNN to purely attention-based model?
→ but, how do we overcome temporal dependencies?

Self-Attention: For each x_i, encode h_i. Then $v_i = v(h_i)$, ← learned (W_v) "value"
For every q_i and k_j, compute and $k_i = k(h_i)$ ← learned (W_k) "key"
attention scores. ($s_{i,j}$) and $q_i = q(h_i)$ ← learned (W_q) "query"
and respective softmax ($\alpha_{i,j}$) → $a_i = \sum_j \alpha_{i,j} v_j$ ← multiply by each value.
↑ activation. This can be inputted into another s-a layer!

essentially a layer.

everything processed w.r.t no assumed order.

Transformer: involves self-attention, positional encoding (addresses sequence info), multi-headed attention (query multiple positions), adding nonlinearities, masked decoding (prevents attention lookups in the future).

→ position of words carries information, so add some info to x_t to indicate that.
↳ $h_t = f(x_t, t)$ relative encoding → absolute encoding.

Let this be p_t.

Frequency-based representations for positional encoding → $PE(pos, 2i) = \sin(pos/10000^{2i/d})$, $PE(pos, 2i+1) = \cos(pos/10000^{2i/d})$ ← from paper; oscillate slow $-1/1$.
OR, we could learn this $d \times T$ matrix. More flexible/optimal. More complex.
Input to self-attention (from paper): $emb(x_t) + p_t$.

$a_i = [a_{i,1}, \ldots, a_{i,n}]$

Multi-Head Attention: ideally would want to incorporate more than 1 time step.
↳ have multiple k/q/v for every time step. (called heads, learned independently).
Naive self-attention lacks "expression" of picked information. Fully linear.

"position-wise feedforward networks"
↳ alternate self-attention and nonlinearity
e.g. $h_t^\ell = \sigma(W^\ell a_t^\ell + b^\ell)$. ← non-linear, learned. Basically NN layer.
↑ output from prev. self-attention layer.

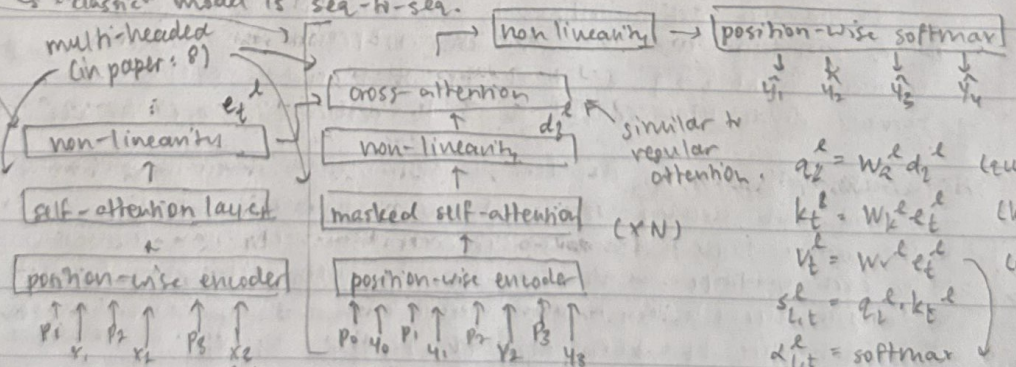
Big issue: self-attention can "see" the future. Not inherently sequential.
→ Masked attention → does not allow lookups past t.
$s_{i,j} = \begin{cases} q_i \cdot k_j & \text{if } i \geq j \\ -\infty \text{ (MIN)} & \text{otherwise (becomes 0 in softmax).} \end{cases}$

"Transformer" → transforms one sequence into another at each layer.
↳ "classic" model is seq-to-seq.

decoder block (processes y_i shifted right)

encoder block (processes x)

in paper, $N = 6$.



$q_t^\ell = W_q^\ell d_t^{\ell-1}$ (query)
$k_t^\ell = W_k^\ell e_t^\ell$ (key)
$v_t^\ell = W_v^\ell e_t^\ell$ (value)
$s_{t,t'}^\ell = q_t^\ell \cdot k_{t'}^\ell$
$\alpha_{t,t'}^\ell = \text{softmax}$
$c_t^\ell = \sum_{t'} \alpha_{t,t'}^\ell v_{t'}^\ell$

→ LayerNorm over BatchNorm to handle layers, not batches (since sequences are diff. lengths).
μ, σ are scalars, aggregate over different dimensions of a, not all a.
↳ straightforward to compute!

→ implement LN + skip connection
input: $\bar{h}_t^{\ell-1}$
output: a_t^ℓ → $\bar{a}_t^\ell = \text{LayerNorm}(\bar{h}_t^{\ell-1} + a_t^\ell)$
"modify", not change, maintain well-behaved gradients

Benefits of Transformer:
- long-range connections
- much easier to parallelize
↳ actually, cheap (?!)
- adaptive, can be made deeper.
- practical improvement