ML and Optimization Lec 23

- Seq2Seq model w/ attention
- Attention is all we need: seq2seq models w/o RNNs
- BERT: pretraining transformer models for NLP
Seq2Seq model

Input: $x_1, \ldots, x_m$
Output: $x_1, \ldots, x_T$

Encoder

Decoder

Issue: regardless of length of input sequence, all context/historical dependence of the output sequence is contained in a $d$-dimensional vector $h_m$ leads to poor performance (BLEU decreases w/ length of input sequence for Machine translation tasks)
Seq2Seq with attention

- Introduced in Bahdanau et al. 2015, Neural Machine Translation by Jointly Learning to Align and Translate

- Allows each output token to attend to each individual hidden state in the input sequence. Captures idea of focusing on relevant portion of input sequence. Avoids the need to capture everything relevant to all outputs $x_1, \ldots, x_T$ in the one vector $h_m$. 

Seq2Seq w/ attention

Idea: replace our previous hidden state update for decoder

\[ s_t = g(s_{t-1}, x_t') \]

w/ first finding which historical hidden states are most relevant:

\[ s_{t-1} \to x_t' \to [h_1, \ldots, h_m] \to \alpha_t \text{ a prob. dist. over } (h_1, \ldots, h_m) \]

then form a context vector

\[ c_t = \sum_{i=1}^{m} \alpha_t(h_i) \]

and update with

\[ s_t = g(s_{t-1}, c_t) \]
In the context of decoding, in the decoder, use attention over the hidden states of the encoder.

\[
\sum_{i=1}^{m} (a_i) h_i \quad \sum_{i=1}^{m} (a_2) h_i \quad \sum_{i=1}^{m} (a_T) h_i
\]
Options for computing the attention vectors $\alpha_t$

1) Option I (used in Bahdanau et al. 2015)

Take $\tilde{\alpha}_i = v^T \tanh (w^T \begin{bmatrix} h_i \\ s_{t-1} \end{bmatrix})$ for $i=1, \ldots, m$

Then $\alpha_t = \text{softmax}(\tilde{\alpha}) \in \mathbb{R}^m$

2) (Dot-product attention)

Idea: generate keys and queries as linear transforms of $h_i$ and $s_{t-1}$ respectively. Take logits $\tilde{\alpha}_i$ to be their inner-products.
Complexity of using attention

1) Simple ANN encoder models: $O(m + T)$

2) ANN + attention models $O(m \cdot T)$

because for each output token, most compute $x_t \in \mathbb{R}^m$ by comparing to all inputs
General Attention
Self-Attention

(replace the hidden states learned in RNNs w/ contextual representations)

\[
c_{i} = \sum_{j=1}^{m} (\alpha_{i})_{j} \cdot v_{j}
\]

where key \( k_{i} = \mathbf{W}_{K} x_{i} \)

query \( q_{i} = \mathbf{W}_{Q} x_{i} \)

value \( v_{i} = \mathbf{W}_{V} x_{i} \)

and \( (\alpha_{i})_{j} = \langle q_{i}, k_{j} \rangle \) and \( \alpha = \text{softmax}(\tilde{\alpha}) \)
Transformers: Attention is all we need

- 2017 Vaswani et al.: Attention is all we need

- Insight: attention alone suffices to contain all historical information we need, so can eliminate recursive structure.

- Replace hidden representations in RNNs w/ contextual representations $C_t$ computed using self-attention (for encoder)

- Use combination of self-attention in decoder and general attention against the encoder contextual representations to get contextual representations to predict output tokens
Multi-Head Self Attention

Just as certain CNNs are benefit from learning multiple contextual representations:

\[(\mathbf{W}_k)_i, (\mathbf{W}_Q)_i, (\mathbf{W}_V)_i \text{ for } i=1, \ldots, d\]
Contextual representations using multi-head self-attention

Self-Attention + Dense Layer
Skip-connections

Skip connections are connections between layers \( l \) and layers higher than \( l+1 \), e.g. DenseNet, where all layers are connected.

An efficient & popular type of skip-connection is a residual connection.

\[
\begin{align*}
    o_{l+1} &= \sigma \left( w_{l+1} o_l + b_{l+1} \right) \\
    o_l &= x_{l+1} \\
    o_{l+1} &= \sigma \left( w_{l+1} o_l + b_{l+1} \right) + o_l
\end{align*}
\]
Layer normalization

Like batch normalization this helps stabilize training by ensuring the distribution of the outputs of layer $l$ doesn't vary much over time, so layer $l+1$ doesn't have to waste optim time adapting.

$$\text{LayerNorm}(0) = y \cdot \frac{0 - \bar{\sigma}}{\sigma} + \beta,$$

where $y \in \mathbb{R}^{n_{l}}$ and $\beta \in \mathbb{R}^{n_{l}}$ are the parameters of the layer normalization layer, and

$$\bar{\sigma} = \frac{1}{n_{l}} \sum_{i=1}^{n_{l}} o_{i}, \quad \sigma^2 = \frac{1}{n_{l}} \sum_{i=1}^{n_{l}} (o_{i} - \bar{\sigma})^2.$$
Batch normalization

- Output neuron index
- Operates on minibatches
- Computes per neuron means and standard deviations over the minibatch
- Then batch normalization of an $o_i$ is given by $\gamma \cdot \left( \frac{o_i - \mu}{\sigma} \right) + \beta$

Layer normalization

- Operates on a single output
- $\overline{o} = \text{mean of } o$
- $\sigma = \text{std of } o$
- Even layer normalization of $o$ is given by $\gamma \cdot \left( \frac{o - \overline{o}}{\sigma} \right) + \beta$
Expressions for Attention

Self-attention:

\[ \alpha_i = \begin{bmatrix} \alpha_i^T & \cdots & \alpha_i^T k_m \end{bmatrix} \]

is the logits vector (row vector) for the \( i \)th attention vector

\[ \alpha_i^T = \text{softmax}(\omega_i^T) \]

is my attention vector

then our contextual embedding for the \( i \)th token is

\[ c_i = \sum_{j=1}^{m} (\alpha_i)_{ij} v_j^T \]

let \( X = \begin{bmatrix} x_1^T & \cdots & x_m^T \end{bmatrix} \in \mathbb{R}^{m \times d} \)

and \( C = \begin{bmatrix} c_1^T \\ \vdots \\ c_m^T \end{bmatrix} \in \mathbb{R}^{m \times d_v} \)
Then
\[ C = \text{Aff}(X; X) \]

where the attention layer has parameters \( \omega_0 > \omega_k > \omega_v \).

To compute \( C \), write
\[ K = X \omega_k = \begin{bmatrix} x_1^T \omega_k \\ \vdots \\ x_m^T \omega_k \end{bmatrix} \]
\[ Q = X \omega_q = \begin{bmatrix} x_1^T \omega_q \\ \vdots \\ x_m^T \omega_q \end{bmatrix} \]
\[ V = X \omega_v = \begin{bmatrix} x_1^T \omega_v \\ \vdots \\ x_m^T \omega_v \end{bmatrix} \]

Then note that
\[ C = \begin{bmatrix} c_0^T \\ \vdots \\ c_m^T \end{bmatrix} = \text{softmax} \left( \frac{Q K^T}{\sqrt{d_k}} \right) V \]

row-wise
Similarly, a general attention layer takes two sequences $X \in \mathbb{R}^{m \times d}$ and $X' \in \mathbb{R}^{T \times d}$.

So a general attention layer computes

$$ C = \text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right) V $$

where

$$ K = X W_K $$

$$ V = X' W_V $$

$$ Q = X' W_Q $$
Transformer Architecture

Encoder:
- use multiple layers of multi-head self-attention with residual connections, layer normalization, and dense nonlinear transforms.

Decoder:
- use multiple layers of multi-head self-attention, multi-head general attention, with residual connections, layer normalizations, and dense nonlinear transforms.

NB: we use layer normalization in Transformer blocks for the same reasons as in RNNs: variable sequence lengths and potentially small batch sizes.
Encoder:

\[ [c_1, \ldots, c_m] \]

contextual
representations

Block N

\[ [\ldots] \]

Block 1

\[ x_1, \ldots, x_m \]

token embeddings

where each embedding block has its own parameters and computes:

Given input \( X = \begin{bmatrix} x_1 \; \ldots \; x_M \end{bmatrix} \in \mathbb{R}^{m \times d} \),

\[ y = \text{LayerNorm} \left( X + \text{Multi-head Att}(X, X) \right) \]

recalling \( \text{Multi-head Att}(X, X) = \left[ A_{h_1}(X, X) \mid \ldots \mid A_{h_H}(X, X) \right] W \)

concatenates in different attention outputs together then takes linear combination to learn a final \( d \)-dim representation

\[ C = \text{LayerNorm} \left( y + \text{FF}(y) \right) \]

where \( \text{FF} \) applies the same feedforward dense+ReLU layer to each row of \( y \)

\[ X \xrightarrow{\text{Block}} C \]
During training, use teacher forcing.

Each block has its own parameters and computes, given inputs $X' = \begin{bmatrix} x_1' \\ \vdots \\ x_T' \end{bmatrix}$ and the final representations from the encoder of the input sequence $C = \begin{bmatrix} c_1 \\ \vdots \\ c_m \end{bmatrix}$:

(i) $Y = \text{LayerNorm} \left( X' + \text{MultiHeadAtt}(X', X') \right)$

(ii) $Z = \text{LayerNorm} \left( Y + \text{MultiHeadAtt}(C, Y) \right)$

(iii) $O = \text{LayerNorm} \left( Z + \text{FF}(Z) \right)$

In the self-attention layers, the attention is masked: we set $(a_{i,j})_j = 0$ for $j > i$ to ensure we only predict using historical info.
During inference time, we autoregressively use the decoder to predict the next token, add it onto our input sequence, and pass it back through the decoder to get the next token. Repeat until we either get an <EOS> token or we’ve reached the maximum sequence length we set.
ENCODER #1
- Positional Encoding
- Add & Normalize
- Self-Attention

ENCODER #2
- Add & Normalize
- Self-Attention
- Feed Forward
- Add & Normalize

DECODER #1
- Add & Normalize
- Encoder-Decoder Attention
- Add & Normalize
- Self-Attention

DECODER #2
- Softmax
- Linear
- Add & Normalize
- Feed Forward
- Add & Normalize
Positional Encodings

Issue w/ Transformers as is: they don’t properly learn positional relations (e.g. that “the cat ate its food” should have a different contextual rep from “the food ate its cat”).

Soln: replace \( X \in \mathbb{R}^{m \times d} \) with \( X + P \) where \( P \in \mathbb{R}^{m \times d} \) encodes the positions 1 through \( n \)

\[
P_{i,j} = \begin{cases} 
  \sin(i \omega j/d) & \text{if } j \text{ even} \\
  \cos(i \omega (j-1)/d) & \text{if } j \text{ odd}
\end{cases}
\]

(Refer to preceding diagram to see where positional embeddings are added.)