EECS 182 Deep Neural Networks
Fall 2022 Anant Sahai

Dis 9 Notes

1 Transformers

At a high-level, transformers consist of the Transformer Encoder and Transformer Decoders.

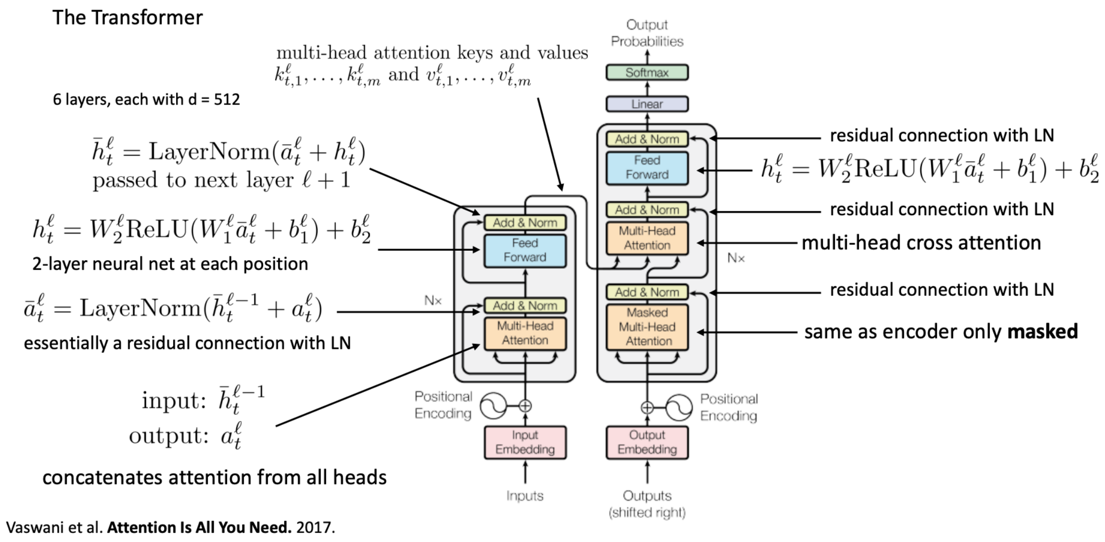
The Transformer
6 layers, each with d = 512
multi-head attention keys and values
$k^\ell_{t,1},\dots,k^\ell_{t,m}$ and $v^\ell_{t,1},\dots,v^\ell_{t,m}$
$\bar{h}^\ell_t = \text{LayerNorm}(\bar{a}^\ell_t + h^\ell_t)$
passed to next layer $\ell+1$
$h^\ell_t = W^\ell_2 \text{ReLU}(W^\ell_1 \bar{a}^\ell_t + b^\ell_1) + b^\ell_2$
2-layer neural net at each position
$\bar{a}^\ell_t = \text{LayerNorm}(\bar{h}^{\ell-1}_t + a^\ell_t)$
essentially a residual connection with LN
input: $\bar{h}^{\ell-1}_t$
output: a^ℓ_t
concatenates attention from all heads
Vaswani et al. Attention Is All You Need. 2017.
Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)
residual connection with LN
$h^\ell_t = W^\ell_2 \text{ReLU}(W^\ell_1 \bar{a}^\ell_t + b^\ell_1) + b^\ell_2$
residual connection with LN
multi-head cross attention
residual connection with LN
same as encoder only masked

Figure 1: Overview of Transformer architecture

Both operate similarly, except the Transformer Decoder takes x_{target} as input, but Transformer Encoder takes in x_{source} as input. In addition, there are several differences in cross-attention and self-attention operations. In particular, transformers are novel in that they add,

- Positional Encoding: Addresses lack of sequence information
- Multi-headed Attention: Allows querying multiple positions at each layer
- Non-linearities
- Masked Decoding: Prevent attention lookups into the future

Notations
To ensure a level of clarity, we will let B be the batch size, L_{source} represent the source sequence length, L_{target} be the target sequence length, D represent the model hidden dimension and H represent the number of attention heads.

In particular, transformers receive two sequences as input. The first is $x_{source} \in \mathbb{Z}^{B\times L_{source}}$ and the second is $x_{target} \in \mathbb{Z}^{B\times L_{target}}$. These are integer tensors, and each integer represents a word or token.

1.1 Transformer Encoder

Input & Positional Embedding The source tensor is embedded into the model hidden dimension, and produces a tensor $X_{source} \in \mathbb{R}^{B \times L_{source} \times D}$. We then add a positional encoding that differs for each sequence position in order to enable the model to differentiate the positions in the sequence. In general, we need this information since position of words in a sentence carries information.

Encoder Attention The Encoder Attention is self-attention. Specifically, in Transformer networks, we use the Scaled QKV Attention. In other words, we would like to build a representation of a single sequence such that every position in the sequence has information about every other position in the sequence. In particular, to enable this, we will use the Query-Key-Values (QKV) Attention. Our queries, keys and values will be tensors in $X_{source} \in \mathbb{R}^{B \times L_{source} \times D}$ and weight matrices will be $W_Q, W_K, W_V \in \mathbb{R}^{D \times D}$. Ultimately, we will retrieve,

$$\begin{aligned} Q &= X_{source} W_Q \\ K &= X_{source} W_K \\ V &= X_{source} W_V \end{aligned}$$

Using Q, K, V, we will compute the attention scores (tensor in $\mathbb{R}^{B \times L_{source} \times L_{source}}$). For each element in the batch, each entry i, j in the matrix would be $\frac{q_i^\top k_j}{\sqrt{D}}$ for scaled dot product attention. In matrix form, we would have $\frac{QK^\top}{\sqrt{D}}$. To produce weights over each position in the sequence, we want each score to sum to one over the keys K. To accomplish this, we take a softmax update over the last dimension of the attention scores. Then, to produce the attention update, we multiply these attention weights by our values V,

$$C_{update} = \mathsf{softmax}\left(\frac{QK^\top}{\sqrt{D}}\right) V$$

where $C_{update} \in \mathbb{R}^{B \times L_{source} \times D}$

One of the key changes in Transformers is the *multi-headed attention* mechanism. To turn it into multi-headed attention, we can take any such update matrices and reshape and permute the matrix from shape $B \times L_{source} \times D$ to $B \times H \times L_{source} \times \frac{D}{H}$.

We finally consider padding. In general, we operate on a batch of B sequences, but these sequences may not be the same length. We pad each sequence to L_{source}. To prevent our model from paying attention to padded positions, we add $-\infty$ to attention scores prior to the Softmax of any position that should be ignored.

Feedforward Layer The feedforward layer applies linear transformation to each position, apply a nonlinear activation, then applies a second linear transformation.

1.2 Transformer Decoder

Masked Decoder Self-Attention Masked decoder self-attention is the same as encoder self-attention, but with different masking. In particular, we would like every position to pay attention to all previous positions, but not future positions. To achieve this, we set attention score to $\frac{q_i^\top k_j}{\sqrt{D}}$ if $i \geq j$ and $-\infty$ otherwise.

Encoder-Decoder Attention Encoder-Decoder attention operated similarly as well, except that we have two sequences: (1) generate queries and (2) generate keys-values. Hence, we let $Q = X_{target}W_Q, K = X_{source}W_K, V = X_{source}W_V$, where X_{source} is the output of the transformer encoder on the source sequences.

In general, transformers are good for long-range connections, are easy to parallelize and transformers can be made much deeper than RNNs. On the other hand, attention computations are complex to implement and computations take $\mathcal{O}(n^2)$ time.

However, in practice, it turns out the benefits vastly outweigh the downsides, and transformers work better than RNNs and LSTMs in many cases.

2 Pretraining and Fine-Tuning

With unsupervised pretraining, the general idea is to use unlabeled data, which is often easily accessible (for example text data on the internet, in books, publications, etc.) in order to learn representations that can be useful for downstream tasks, such that not as much task-specific data is needed for good performance on that task.

To illustrate why we might expect this to be helpful, we can imagine we want to translate English sentences to French, and are given a labeled dataset of English/French sentence pairs. You can imagine this task would be really difficult if you had no prior knowledge of English, while being much more manageable if you came in with a general understanding of the English language already, which can be learned using unsupervised data (for example, all the English text we see on the internet).

At a high level, one simple way we can embed words in a context-dependent manner is to take a language model (for example an LSTM) trained on some task, and to run a sentence through it, taking the hidden state of the model as the embedding for each word. Since these language models presumably had to use the context in order to solve the task they were trained on, using the hidden state as an embedding should provide context-dependent representations of words.

ELMo: We note that if we simply ran an LSTM forward through a sentence to generate the embeddings of words, the embedding of each word would only depend on those that came before it, rather than the full context of the word. ELMo addresses this issue by simply training a bidirectional LSTM (both trained to predict the next/previous word), and concatenating hidden states of both directions together to form an embedding. ELMo has been largely replaced by other models in current NLP research, and the following models are more represenative of what is currently used in NLP today.

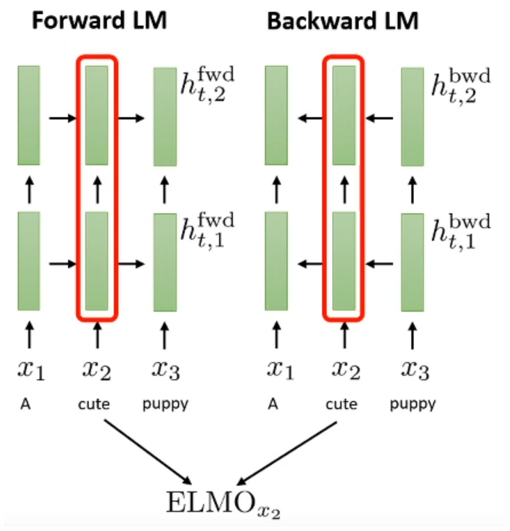

Figure 2: ELMo takes the hidden states in a bi-directional LSTM to generate word embeddings. The LSTMs are both trained via sequence prediction.

GPT: GPT (and its successors GPT-2 and GPT-3) are high-capacity transformer-based language models trained on very large amounts of unlabeled text (e.g. text from the internet). Because they are forward generative language models, they model architectures consists only of a transformer decoder. While conceptually simple, these models can be incredibly powerful for generating text data, with the most recent version GPT-3 being able to generate text that is almost indistinguishable from text written by a human. The representations learned by GPT can also be effectively used for downstream tasks, but they may be a sub-optimal from some tasks because GPT is a forward language model, so its representations only incorporate context from past context, not the entire sequence of text.

BERT: One can imagine incorporating bidirectional context with a transformer-based language model in similar manner as ELMo, where we can learn both a forward and backward language model and concatenate their embeddings. However, while such an embedding would capture bidirectional context, the individual tasks of forward and backward language modeling are inherently unidirectional, so simply concatenating their embeddings may not learn representations that capture bidirectional relationships well. Instead, BERT relies on a *single* transformer encoder to generate embeddings that incorporate bidirectional context, using an inherently bidirectional pretraining task.

While the previous transformers we saw for sequence modeling relied on masked self-attention to avoid peeking into the future, our goal here is to digest the entire context of a word to produce an embedding, which eliminates the need for the mask. However, this presents a complication if we were to try train embeddings to predict the next word like ELMo or GPT. The issue here is that if we did unmasked self-attention, we can already directly see the next word in the input, making prediction completely trivial and preventing useful representations from being learned.

The solution is to simply change the unsupervised task. Instead of predicting the next word in sentence, we instead randomly mask out certain words in the input, and then train the embedding to predict the masked out words. In this way, our model is forced to learn context dependent word-level representations to predict the missing words.

In addition to learning word-level representations by predicting masked out words, BERT also tries to learn *sentence-level* representations. To train this, BERT takes in pairs of sentences, half of which are consecutive and half of which are paired randomly. It trains a binary classifier to predict whether the two sentences are consecutive or not.

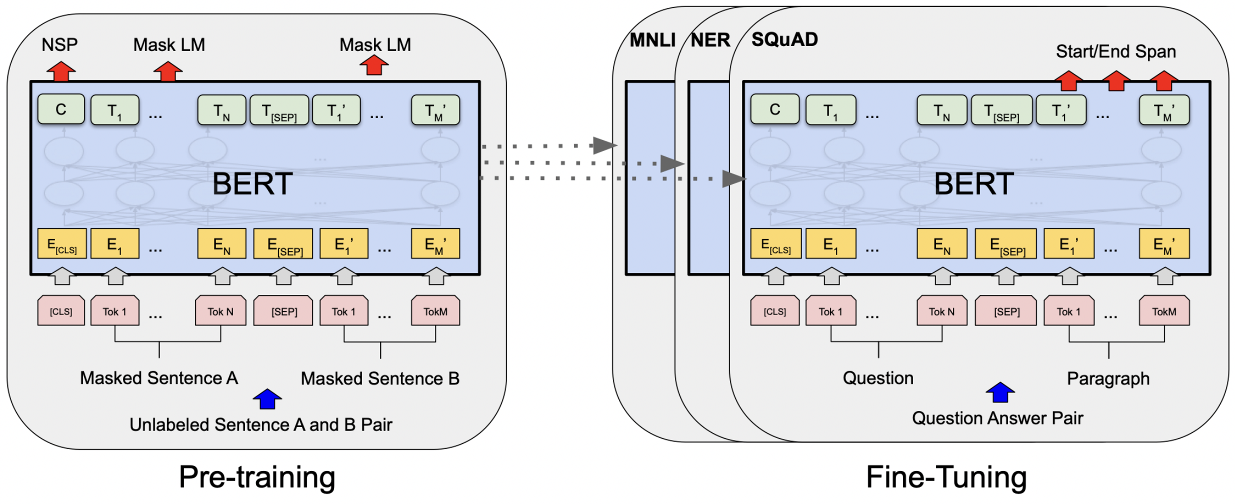

Figure 3: Overall pre-training and fine-tuning procedures for BERT. Apart from output layers, the same architectures are used in both pre-training and fine-tuning. The same pre-trained model parameters are used to initialize models for different down-stream tasks. During fine-tuning, all parameters are fine-tuned. [CLS] is a special symbol added in front of every input example, and [SEP] is a special separator token (e.g. separating ques- tions/answers).

This pretraining procedure gives BERT the ability to produce powerful represenations for downstream tasks that require language understanding. Such tasks include sentiment analysis, textual entailment, and question answering. Depending on the downstream task, we can either use the sentence level representation outputted by BERT or the word-level representations in the downstream task. We can use BERT for downstream tasks both by simply finetuning the entire model on the downstream tasks, or taking combinations of the hidden states as fixed representations. Typically, we use the last layer, the sum of the last 4 layers, or the concatenation of the last 4 layers as features for the downstream tasks.