13.2. Transformer Networks

François Fleuret

https://fleuret.org/ee559/

Feb 2, 2020
The most powerful language models (as of 03.07.2019) take the form of a sequence of attention-based revisions of the representation.

The original sequence structure of the signal is secondary and provided indirectly to the processing through additional inputs.
Vaswani et al. (2017) use the terminology of Graves et al. (2014): attention is an averaging of values associated to keys matching a query.

With $Q$ the tensor of row queries, $K$ the keys, and $V$ the values,

$$Y_j = \sum_i \text{softmax}_i(Q_j, K_i^T)V_i$$

or

$$Y = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V \in \mathbb{R}^{a \times d},$$

where

$$Q \in \mathbb{R}^{a \times b}, K \in \mathbb{R}^{c \times b}, V \in \mathbb{R}^{c \times d}.$$
\[ K^T \]

\[ Q \]

\[ QK^T \]

\[ \text{Softmax} \]

\[ A \]

\[ Y = AV \]
3.2.1 Scaled Dot-Product Attention

We call our particular attention “Scaled Dot-Product Attention” (Figure 2). The input consists of queries and keys of dimension $d_k$, and values of dimension $d_v$. We compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix $Q$. The keys and values are also packed together into matrices $K$ and $V$. We compute the matrix of outputs as:

$$\text{Attention}(Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right) V$$

The two most commonly used attention functions are additive attention [2], and dot-product (multiplicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of $\frac{1}{\sqrt{d_k}}$. Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

While for small values of $d_k$ the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of $d_k$ [3]. We suspect that for large values of $d_k$, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients. To counteract this effect, we scale the dot products by $\frac{1}{\sqrt{d_k}}$.

3.2.2 Multi-Head Attention

Instead of performing a single attention function with $d_{\text{model}}$-dimensional keys, values and queries, we found it beneficial to linearly project the queries, keys and values $h$ times with different, learned linear projections to $d_k$, $d_k$ and $d_v$ dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding $d_v$-dimensional output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 2.

(Vaswani et al., 2017)

$$\text{MultiHead}(Q, K, V) = \text{Concat} (H_1, \ldots, H_h) W^O$$

$$H_i = \text{Attention} \left( QW_i^Q, KW_i^K, VW_i^V \right), \ i = 1, \ldots, h$$

with

$$W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}, \ W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}, \ W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}, \ W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$$. 
Attention disregards positioning in the sequence, Vaswani et al. provide this information through a **positional encoding**, added to the original input.

It has the same dimension as $d_{\text{model}}$ and is of the form

\[
PE_{t,2i} = \sin \left( \frac{t}{10,000 \cdot 2^i d_{\text{model}}} \right)
\]

\[
PE_{t,2i+1} = \cos \left( \frac{t}{10,000 \cdot 2^i+1 d_{\text{model}}} \right)
\]
3.1 Encoder and Decoder Stacks

Encoder: The encoder is composed of a stack of \( N = 6 \) identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection [1] around each of the two sub-layers, followed by layer normalization [1]. That is, the output of each sub-layer is

\[
\text{LayerNorm}(x + \text{Sublayer}(x)),
\]

where \( \text{Sublayer}(x) \) is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension \( d_{\text{model}} = 512 \).

Decoder: The decoder is also composed of a stack of \( N = 6 \) identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with the fact that the output embeddings are offset by one position, ensures that the predictions for position \( i \) can depend only on the known outputs at positions less than \( i \).

3.2 Attention

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

(Vaswani et al., 2017)
The Universal Transformer (Dehghani et al., 2018) is a similar model where all the blocks are identical, resulting in a **recurrent model that iterates over consecutive revisions of the representation** instead of positions.

The positional embedding is expended with the block index $1 \leq t \leq T$

\[
\begin{align*}
    P_{i,2j}^t &= \sin \left( \frac{i}{10,000 \frac{2j}{d_{model}}} \right) + \sin \left( \frac{t}{10,000 \frac{2j}{d_{model}}} \right) \\
    P_{i,2j+1}^t &= \cos \left( \frac{i}{10,000 \frac{2j}{d_{model}}} \right) + \cos \left( \frac{t}{10,000 \frac{2j}{d_{model}}} \right)
\end{align*}
\]
The Universal Transformer (Dehghani et al., 2018) is a similar model where all the blocks are identical, resulting in a **recurrent model that iterates over consecutive revisions of the representation instead of positions.**

The positional embedding is expended with the block index $1 \leq t \leq T$

$$P^t_{i,2j} = \sin \left( \frac{i}{10,000 \frac{2j}{d_{model}}} \right) + \sin \left( \frac{t}{10,000 \frac{2j}{d_{model}}} \right)$$

$$P^t_{i,2j+1} = \cos \left( \frac{i}{10,000 \frac{2j}{d_{model}}} \right) + \cos \left( \frac{t}{10,000 \frac{2j}{d_{model}}} \right)$$

Additionally the number of steps is modulated per position dynamically.
Transformers trained on NLP tasks
GPT is a transformer trained for generating.

(Radford, 2018)
“GPT-2 is a large transformer-based language model with 1.5 billion parameters, trained on a dataset of 8 million web pages. GPT-2 is trained with a simple objective: predict the next word, given all of the previous words within some text. The diversity of the dataset causes this simple goal to contain naturally occurring demonstrations of many tasks across diverse domains. GPT-2 is a direct scale-up of GPT, with more than 10X the parameters and trained on more than 10X the amount of data.”

(Radford et al., 2019)
BERT (Devlin et al., 2018) is a transformer pre-trained with:

- Masked Language Model (MLM), that consists in predicting [15% of] words which have been replaced with a “MASK” token.
- Next Sentence Prediction (NSP), which consists in predicting if a certain sentence follows the current one.

It is then fine-tuned on multiple NLP tasks. (Conneau et al., 2019)
Figure 3: Differences in pre-training model architectures. BERT uses a bidirectional Transformer. OpenAI GPT uses a left-to-right Transformer. ELMo uses the concatenation of independently trained left-to-right and right-to-left LSTMs to generate features for downstream tasks. Among the three, only BERT representations are jointly conditioned on both left and right context in all layers. In addition to the architecture differences, BERT and OpenAI GPT are fine-tuning approaches, while ELMo is a feature-based approach.

(Devlin et al., 2018)
We note that in the literature the bidirectional Transformer architecture across different tasks. There is mini-
as a running example for this section. There are two steps in our framework:

1. Pre-training
2. Fine-tuning

Figure 1: 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. \([\text{CLS}]\) is a special symbol added in front of every input example, and \([\text{SEP}]\) is a special separator token (e.g. separating questions/answers).

(Devlin et al., 2018)
SST-2 The Stanford Sentiment Treebank is a binary single-sentence classification task consisting of sentences extracted from movie reviews with human annotations of their sentiment (Socher et al., 2013).

CoLA The Corpus of Linguistic Acceptability is a binary single-sentence classification task, where the goal is to predict whether an English sentence is linguistically “acceptable” or not (Warstadt et al., 2018).

STS-B The Semantic Textual Similarity Benchmark is a collection of sentence pairs drawn from news headlines and other sources (Cer et al., 2017). They were annotated with a score from 1 to 5 denoting how similar the two sentences are in terms of semantic meaning.

MRPC Microsoft Research Paraphrase Corpus consists of sentence pairs automatically extracted from online news sources, with human annotations for whether the sentences in the pair are semantically equivalent (Dolan and Brockett, 2005).

RTE Recognizing Textual Entailment is a binary entailment task similar to MNLI, but with much less training data (Bentivogli et al., 2009).

WNLI Winograd NLI is a small natural language inference dataset (Levesque et al., 2011). The GLUE webpage notes that there are issues with the construction of this dataset, and every trained system that’s been submitted to GLUE has performed worse than the 65.1 baseline accuracy of predicting the majority class. We therefore exclude this set to be fair to OpenAI GPT. For our GLUE submission, we always predicted the majority class.

Note that we only report single-task fine-tuning results in this paper. A multitask fine-tuning approach could potentially push the performance even further. For example, we did observe substantial improvements on RTE from multitask training with MNLI. 

https://gluebenchmark.com/faq

Figure 4: Illustrations of Fine-tuning BERT on Different Tasks.

(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG

(b) Single Sentence Classification Tasks: SST-2, CoLA

(c) Question Answering Tasks: SQuAD v1.1

(d) Single Sentence Tagging Tasks: CoNLL-2003 NER

(Devlin et al., 2018)
**Head 8-10**

- **Direct objects** attend to their verbs
- 86.8% accuracy at the dobj relation

**Head 8-11**

- **Noun modifiers** (e.g., determiners) attend to their noun
- 94.3% accuracy at the det relation

(Clark et al., 2019)
Head 7-6
- Possessive pronouns and apostrophes attend to the head of the corresponding NP
- 80.5% accuracy at the poss relation

Head 4-10
- Passive auxiliary verbs attend to the verb they modify
- 82.5% accuracy at the auxpass relation

(Clark et al., 2019)
- **Prepositions** attend to their objects
- 76.3% accuracy at the `pobj` relation

- **Noun modifiers** (e.g., determiners) attend to their noun
- 94.3% accuracy at the `det` relation

- **Direct objects** attend to their verbs
- 86.8% accuracy at the `dobj` relation

- **Possessive pronouns and apostrophes** attend to the head of the corresponding NP
- 80.5% accuracy at the `poss` relation

- **Passive auxiliary verbs** attend to the verb they modify
- 82.5% accuracy at the `auxpass` relation

- **Coreferent mentions** attend to their antecedents
- 65.1% accuracy at linking the head of a coreferent mention to the head of an antecedent

(Clark et al., 2019)
Attention in computer vision
Wang et al. (2018) proposed an attention mechanism for images, following the model from Vaswani et al. (2017).

\[ y = \text{softmax} \left( (W_\theta x)^T (W_\phi x) \right) W_g x. \]
Wang et al. insert “non-local blocks” in residual architectures and get improvements on both video and images classification.

Figure 2. A spacetime **non-local block**. The feature maps are shown as the shape of their tensors, e.g., $T \times H \times W \times 1024$ for 1024 channels (proper reshaping is performed when noted). “$\odot$” denotes matrix multiplication, and “$\oplus$” denotes element-wise sum. The softmax operation is performed on each row. The blue boxes denote $1 \times 1 \times 1$ convolutions. Here we show the embedded Gaussian version, with a bottleneck of 512 channels. The vanilla Gaussian version can be done by removing $\theta$ and $\phi$, and the dot-product version can be done by replacing softmax with scaling by $1/N$.

(Wang et al., 2018)
Figure 3. Examples of the behavior of a non-local block in res3 computed by a 5-block non-local model trained on Kinetics. These examples are from held-out validation videos. The starting point of arrows represents one $x_i$, and the ending points represent $x_j$. The 20 highest weighted arrows for each $x_i$ are visualized. The 4 frames are from a 32-frame input, shown with a stride of 8 frames. These visualizations show how the model finds related clues to support its prediction.

(Wang et al., 2018)
The problem of long range interactions has been tackled in sequence modeling through the use of attention. Attention has ... competitive results including text-to-speech [36] and generative sequence models [37, 38]. Several efforts have

Ramachandran et al. (2019) replaced convolutions with local attention.

\[
y_{i,j} = \sum_{(a,b) \in \mathcal{N}(i,j)} W_{i-a,j-b}x_{a,b} \quad \text{(Convolution)}
\]

\[
y_{i,j} = \sum_{(a,b) \in \mathcal{N}(i,j)} \text{softmax}_{a,b} \left( (W_Q x_{i,j})^T (W_K x_{a,b}) \right) v_{a,b} \quad \text{(Local attention)}
\]

Figure 2: An example of a $3 \times 3$ convolution. The output is the inner product between the local window and the learned weights.

Figure 3: An example of a local attention layer over spatial extent of $k = 3$.

(Ramachandran et al., 2019)
Table 1: ImageNet classification results for a ResNet network with different depths. Baseline is a standard ResNet, Conv-stem + Attention uses spatial convolution in the stem and attention everywhere else, and Full Attention uses attention everywhere including the stem. The attention models outperform the baseline across all depths while having 12% fewer FLOPS and 29% fewer parameters.

Figure 5: Comparing parameters and FLOPS against accuracy on ImageNet classification across a range of network widths for ResNet-50. Attention models have fewer parameters and FLOPS while improving upon the accuracy of the baseline.

(Ramachandran et al., 2019)
Implementation example
PyTorch allows to operate with matrices or linear modules over tensors of dimensions greater than 2.

```python
>>> a = torch.empty(100, 2, 3, 4)
>>> m = nn.Linear(4, 11)
>>> m(a).size()
torch.Size([100, 2, 3, 11])
```
matmul allows structured batch matrix products.

```python
>>> a = torch.empty(11, 9, 2, 3).normal_(
>>> b = torch.empty(11, 9, 3, 4).normal_(
>>> m = a.matmul(b)
>>> m.size()
torch.Size([11, 9, 2, 4])
>>> m[7, 1]
tensor([[[-0.0234,  0.9557, -0.8338,  2.2662],
             [-0.1781,  0.2189, -0.5915, -0.1888]])
>>> a[7, 1].mm(b[7, 1])
tensor([[[-0.0234,  0.9557, -0.8338,  2.2662],
             [-0.1781,  0.2189, -0.5915, -0.1888]])
>>> m[3, 0]
tensor([[-0.7566, -0.1191,  0.9691,  0.6992],
            [ 0.7131, -0.7072, -0.4597, -1.6044]])
>>> a[3, 0].mm(b[3, 0])
tensor([[-0.7566, -0.1191,  0.9691,  0.6992],
            [ 0.7131, -0.7072, -0.4597, -1.6044]])
```
class AttentionLayer2d(nn.Module):
    def __init__(self, dim_x, dim_k, dim_v):
        super(AttentionLayer2d, self).__init__()
        self.dim_k = dim_k
        self.W_q = nn.Linear(dim_x, dim_k, bias=None)
        self.W_k = nn.Linear(dim_x, dim_k, bias=None)
        self.W_v = nn.Linear(dim_x, dim_v, bias=None)

    def forward(self, x):
        # NCHW -> NKC
        u = x.permute(0, 2, 3, 1)
        u = u.view(u.size(0), -1, u.size(3))

        q = self.W_q(u)
        k = self.W_k(u)
        v = self.W_v(u)

        a = q.matmul(k.transpose(1, 2)) / math.sqrt(self.dim_k)
        # a will operate on the right, it should be row-normalized
        a = torch.softmax(a, 2)
        y = a.matmul(v)

        # NKC -> NCHW
        y = y.reshape(x.size(0), x.size(2), x.size(3), -1)
        y = y.permute(0, 3, 1, 2)

        return y
The end
References


