Transformers

Michele Garibbo

1 Overview

Here, I provide some notes on the Transformer architecture, by especially focusing on its core component: self-attention. Next, I provide a quick overview of the 3 main Transformer architectures. For more details on how the different Transformer components interact, you can look at my own commented implementation of the original Transformer architecture here.

2 Self-attention

The notes on self-attention are primarily based on the amazing blog post available at `https://peterbloem.nl/blog/transformers`.

2.1 Self-attention at the core

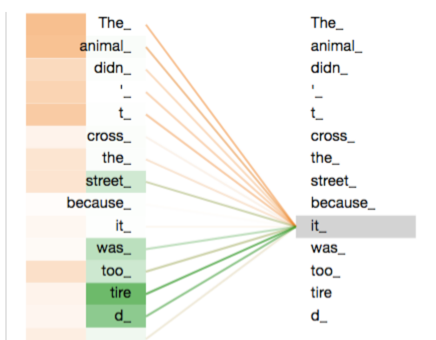

Figure 1: SA output, illustrating the relations between the word 'it' and all the other words captured by two attention heads (one orange, the other green). Image taken from here.

Self-attention (SA) is a sequence-to-sequence operation, taking a sequence, $\{x_1, x_2 \cdots, x_t\}$ and outputting a sequence, $\{y_1, y_2 \cdots, y_t\}$, where all vectors have dimension l. To compute the sequence, y, SA simply computes a weighted sum across all inputs, x,

$$w'_{ij} = x_i^\intercal x_j \tag{1}$$

$$w_{ij} = \frac{\exp w'_{ij}}{\sum_j \exp w'_{ij}} \tag{2}$$

$$y_i = \sum_j w_{ij} x_j \tag{3}$$

where w is not a traditional parameter, but it is a function of the input sequence. In this case, we take this function to be the dot-product across the input sequence (i.e., simplest case). Specifically, eq. 1 is computing y_i by simply summing over all inputs, each weighted by how "similar" (dot-product) each input is to x_i. The assumption is that the more similar an input is to x_i, the more it should contribute to y_i. I guess there is an implicit 'order bias' here, where output y_i should most likely be driven by inputs that are similar to x_i (once we introduce learnable weights, this should no longer be the case). As a result, y_i encodes the relation between x_i and the other input (embeddings) x_j (i.e., enabling to infer relations across elements of the input sequence). Since the dot product is unbounded, we squish everything between $[0, 1]$, by applying a softmax operation. By summing over all inputs to compute each output, SA allows any element in the sequence to arbitrarily contribute to each output without any 'order/temporal' limitation (e.g., unlike RNNs where each output depends on the previous time step). For instance, Fig. 1 show SA is able to infer the word 'it' has a strong relation to the initial words 'The animal', since the word 'it' refers to 'The animal' (i.e.,,the subject of the sentence). SA is able to infer this relation despite the two words being far apart in the sequence (i.e., 9 time steps a part).

Note, at the moment there is no learning involved, this is just a simple example to see how SA operates (see Fig. 2 for code implementation). Finally, it is worth noting that SA is **permutation equivariant**, which means if I permute the inputs (i.e., change the order) the SA output will be the same, just permuted (i.e., SA basically ignores the order of the sequence, just processing the sequence as a set). This is partially the reason why positional embeddings are added to the input embedding.

2.2 Self-attention in transformers

Self-attention in Transformers adds three key tricks to the core operations described above,

1) Query, keys, values:

Each time the input sequence x is used in the SA operation, we apply a linear

```
" Implement a simple self-attention mechanism, where x: input sequence, w: attention weights, y: output score """

batch_s = 34
l = 10 # input size
t = 20 # sequence length
x = torch.randn(batch_s, t, l)

## ============= Simplest self-attention (SA) mechanism (without keys, queries & values) =========

## ----------- My implementation ------------
w_prime = x @ x.transpose(-2,-1)
w = torch.nn.functional.softmax(w_prime,dim=2)
my_y = w @ x
```

Figure 2: Simplest SA implementation in Pytorch

transformation to it, allowing us to introduce learnable parameters,

$$q_i = \mathbf{W}_{\mathrm{q}} x_i \qquad k_i = \mathbf{W}_{\mathrm{k}} x_i \qquad v_i = \mathbf{W}_{\mathrm{v}} x_i \tag{4}$$

$$\tag{5}$$

$$w'_{ij} = q_i^{\intercal} k_j \tag{6}$$

$$w_{ij} = \frac{\exp w'_{ij}}{\sum_j \exp w'_{ij}} \tag{7}$$

$$y_i = \sum_j w_{ij} v_j \tag{8}$$

The vectors q_i, k_i, v_i are respectively called query, key and value, enabling SA to have controllable parameters, $\mathbf{W}_{\mathrm{q}}, \mathbf{W}_{\mathrm{k}}, \mathbf{W}_{\mathrm{v}}$.

Note, for each query q_i, the softmax 'picks' the dot product, w'_{ij}, encoding the key, k_j that matches most with the given query q_i, This is because each row w'_i encodes the dot product for the same query i across different keys, j and the softmax is computed across the row of w' (i.e., across different keys). As a result, any row w_i has one high entry (i.e., the dot product of the key j matching most the query i) with all other entries being close to zero. Intuitively, this follows from considering the softmax as a 'differentiable max' function, increasing the largest value and sending all others value to approximately zero. Finally, when we multiply each entry in w_i with the value v_j, only one value will be selected (since most entries in w_i are close to zero due to the softmax). Basically, the idea is, for any given query, to select the value for which the key matched most the given query.

2) Normalising the dot product:

The soft-max operation is very sensitive to large values (i.e., exponential function). This kills gradients, slowing or even impairing learning. Therefore, it is best to normalise the dot product by size of the input dimension, before passing it to the soft-max,

$$w'_{ij} = \frac{q_i^\intercal x_j}{\sqrt{l}} \tag{9}$$

where l is the size of input dimension. Note, this makes sense since the dot product tend to scale with the input dimension.

3) Multi-head attention:

Consider the following sentence, "Mary gave roses to Susan", represented by input embeddings,

$$x_{\text{Mary}},\ x_{\text{gave}},\ x_{\text{roses}},\ x_{\text{to}},\ x_{\text{Susan}} \tag{10}$$

After the SA operation, you get another sequence,

$$y_{\text{Mary}},\ y_{\text{gave}},\ y_{\text{roses}},\ y_{\text{to}},\ y_{\text{Susan}} \tag{11}$$

where, for instance, y_{gave} is fully determined by the weighted sum over all inputs, each input weighted by "how close" the input is to x_{gave}. However, the word 'gave' has different relations to different parts of the sentence. Specifically, 'Mary' expresses who is giving, while 'Susan' expresses who is receiving and 'roses' expresses what is given. The dot product between input words only captures an overall difference in relations across input words (e.g., x_{gave}-x_{Mary} vs x_{gave}-x_{Susan}), determining the different amounts by which each of these word should contribute to y_{gave} (i.e., the size of the weight in the weighted sum). However, it cannot capture the different ways in which each word should contribute to y_{gave}. In practice, I think this means that each input cannot affect the dimension of y_{gave} differently depending on the context, but can only scale up or down the entire contribution of the input to y_{gave}. This issue can be overcome by introducing multiple SA operations in parallel, where each SA operation uses different/independent weighting in the sum. Next, we can combine all these different ways for x_{Mary} and x_{Susan} to influence y_{gave}, by concatenating them into a single vector and passing this concatenated vector through a learnable linear transformation, $\mathbf{W}_0$ to reduce the dimension back to l.

A nice illustration of multi-head attention can be found in Fig. 1, where the two attention heads (represented by the color orange and green) capture different relations between the word 'it' and all the other words. Specifically, the 'orange' head seems to focus on the relation between 'it' and 'the animal', while the 'green' head seems to focus on the relation between 'it' and 'tired'.

Efficient multi-head attention:

Employing h attention heads, implies we end up with h-times more parameters, greatly reducing learning speed. This is because each attention head, r, needs its own 3 matrices, $\mathbf{W}_{\text{q}}^r, \mathbf{W}_{\text{k}}^r, \mathbf{W}_{\text{v}}^r$, to build different/independent weighted sum (i.e., different ways for the inputs to determine an output y_i). It turns out we can achieve multi-head SA, which is approximately as fast a single-head SA. The way to do this is to project the input, x onto low-dimensional queries, q, keys, k and values, v, inside each head.

For instance, imagine our input embedding is $l = 256$, we can project this to low dimensional queries, keys and values for each head. In particular, if we have $h = 4$ heads, we want to project this to a $l/h = 64$ subspace. The way to do this is to employ linear transformation for queries, keys and values, $\mathbf{W}_\mathrm{q}^r, \mathbf{W}_\mathrm{k}^r, \mathbf{W}_\mathrm{v}^r$, that are 256×64 matrices. Therefore, each attention head project the input onto low-dimensional queries, keys and values. As a result, we end up with $3hl\frac{l}{h}$ parameters, which is equal to $3l^2$ parameters (i.e., the number of parameters for a single attention head).

Efficient multi-head attention implementation:

When we are implementing multi-head attention, there is no need to initialise 3 different $\mathbf{W}_\mathrm{q}^r, \mathbf{W}_\mathrm{k}^r, \mathbf{W}_\mathrm{v}^r \in \mathbb{R}^{l \times \frac{l}{h}}$ for each head. We can actually initialise three single matrices $\mathbf{W}_\mathrm{q}^r, \mathbf{W}_\mathrm{k}^r, \mathbf{W}_\mathrm{v}^r \in \mathbb{R}^{l \times l}$ then, slicing the resulting vectors to apply separate attention operation (i.e., the dot product). Fig. 4 and Fig. 5 shows two different implementations of a multi-head self-attention 'forward pass'. Fig. 5 is a less efficient implementations due to a for-loop (i.e., cannot run in parallel), but gives better insights on what is happening. Fig. 4 is a more efficient way to implement self-attention. Both implementation relies on the same class initialisation, which can be found in Fig. 3.

Note, depending on the transformer architecture (see Section 3.2), we may need to change the implementation of self-attention slightly. For instance in encoder-decoder transformers, we need to pass at least 2 different variables as input to the `forward` method of the `SelfAttention` class (though, for clarity, 3 variables are typically passed, which are denoted as "query", "keys" and "values"). This is because in some of the multi-attention layers of the encoder-decoder transformer we use the decoder sequence to compute the query for the encoder output, which in turn is used to compute the keys and values. Therefore, we cannot pass a single input variable, `x`, to the `forward` method in Fig. 4. Additionally, we typically need to include a mask to performs masked self-attention.

```python
class SelfAttention(nn.Module):

    def __init__(self, l, h):
        """
        Implement (fast) multi-head attention mechanism by reducing dim of input to each head to l/h size
        Args:
            l: input size
            h: n. of attention head
        """

        super().__init__()
        self.l = l
        self.h = h

        ## Ensure n. heads compatible with input size to implement fast multi-head operation
        ## by reducing dim of input to each head to l/h size
        assert l%h ==0 , "n. of heads must be compatible with input size"

        self.chunk_dim = l//h

        ## Implement multiple (head) queries, keys and values matrices as 3 'big' matrices
        ## instead of initialising 3 different (q,k,v) matrices for each head
        self.q_mat = nn.Linear(l,l,bias=False)
        self.k_mat = nn.Linear(l,l,bias=False)
        self.v_mat = nn.Linear(l,l,bias=False)

        ## Initialise linear transform to map concatenated multi-head values
        ## back to a unified vector
        self.unifyheads = nn.Linear(l,l)
```

Figure 3: Multi-head attention class initialisation.

```python
    def my_forward(self, x):
        """
        This provides my own implementation of self attenttion, using a for loop (suboptimal, can't be paralellised)
        Args:
            x: input sequence [batch_s, t, l]
        """

        # Compute queries, values and key in one go for all heads
        q = self.q_mat(x)
        k = self.k_mat(x)
        v = self.v_mat(x)

        ## Use loop to slice q,k,v across dimensions to perform self-attention across different heads
        norm_dot = torch.sqrt(torch.tensor(self.chunk_dim)) # pre-compute to normalise dot product
        t=0
        y=[]
        for _ in range(self.h):
            w_prime = q[...,t:t+self.chunk_dim] @ k[...,t:t+self.chunk_dim].transpose(-2,-1) / norm_dot
            w = torch.softmax(w_prime,dim=-1)
            y.append(w @ v[..., t:t+self.chunk_dim])
            t+= self.chunk_dim

        y = torch.cat(y,dim=-1)

        return self.unifyheads(y)
```

Figure 4: A sub-optimal implementation of the multi-head attention forward pass (i.e., cannot be parallelised due to the loop). However, I find this implementation quite useful in illustrating how multi-head attention works, by dividing the embedding dimension into non-overlapping chunks and, feeding a different chunk to each attention head.

```python
def forward(self, x):
    """
    Efficiently implepented self attention without for loop based on tutorial
    The trick is to treat different heads' q,k,v as different batch elements
    and use batch matrix multiplication to process chunks in parallel
    Args:
        x: input seq [batch_s, t, l]
    """
    b, t, l = x.size()

    # Compute queries, values and key in one go for all heads
    q = self.q_mat(x)
    k = self.k_mat(x)
    v = self.v_mat(x)

    # ==== Perfor separate self-attention (head) operations by dividing into chunks ====

    # ---- First create a new dimension that 'iterate' over the heads ----
    keys = k.view(b,t,self.h, self.chunk_dim)
    queries = q.view(b,t,self.h, self.chunk_dim)
    values = v.view(b,t,self.h, self.chunk_dim)

    # --- Second compute the dot product ----
    # Since this is the same operation for every head we can fold the heads into the batch dimension, and use batch matrix multiplication
    # in this way each parallel head computation treated as part of larger batch

    # NOTE: to do so need .transpose() head dim and seq dim since, head and batch dims are not
    # next to each other so cannot merge them directly
    # NOTE: if use .reshape() no need to use contiguous(), since reshape() creates new Tensor with contingous memory after transpose()
    # but here just need new view, but where memory is contiguous rather copying the Tensor to new shape
    keys = keys.transpose(1,2).contiguous().view(b*self.h, t, self.chunk_dim)
    queries = queries.transpose(1,2).contiguous().view(b*self.h, t, self.chunk_dim)
    values = values.transpose(1,2).contiguous().view(b*self.h, t, self.chunk_dim)

    # compute dot product
    w_prime = torch.bmm(queries, keys.transpose(1,2))
    # scale dot product
    w = w_prime / (self.chunk_dim ** (1/2))
    # normalize
    w = torch.softmax(w, dim=2)

    # ---- Third apply self-attention to values -----
    y_prime = torch.bmm(w, values).view(b, self.h, t, self.chunk_dim)

    # ---- Fourth bring back to original dimension ----
    y = y_prime.transpose(1,2).contiguous().view(b, t, self.h * self.chunk_dim)

    return self.unifyheads(y)
```

Figure 5: 'Optimal' implementation of the multi-head attention forward pass.

3 Transformers

Transformers aren't just SA, but they are an entire architecture. At the core of this architecture is the 'transformer' block, which can be stacked (e.g., similar to CNN, which are composed of stacked blocks/layers, each including convolution, max pooling operations etc.).

3.1 Transformer block

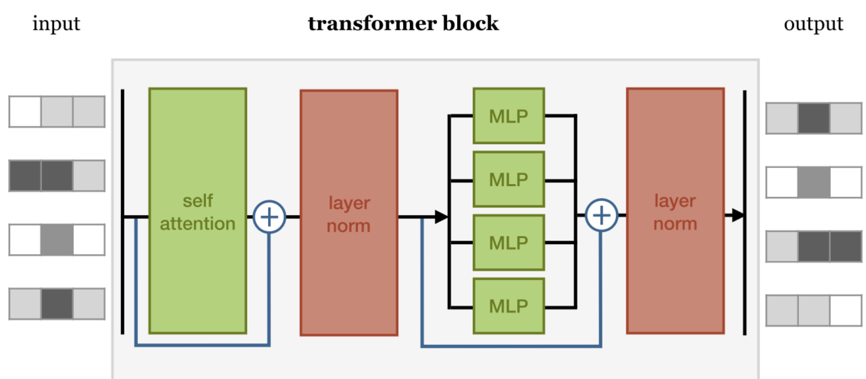

Figure 6: Transformer block, image taken from here.

A transformer block includes a SA operation, layer normalization, a feedforward layer and another layer normalization (see Fig. 6). Additionally, residual connections are added, typically, one skipping over the SA operation and another skipping over the feedforward operations. The order of these operations can be varied, although it is important to include each. At its core, each transformer is a series of stacked transformer blocks, with some key difference in the way these blocks are organised relative to the input and target sequences. Broadly, we can summarise these variations in 3 main types of Transformer architectures.

3.2 Transformer types

We can distinguish 3 main types of Transformer architectures,

1. **Encoder-Decoder**: This represents the original Transformer architecture (i.e., "Attention is all you need"). This model comprises of an encoder part, which encodes the input sequence into "context vectors" (i.e., vectors encoding the relation across elements of the input sequence). Plus, this model includes a decoder part, which first encodes the (masked) target sequence into "context vectors" and then uses these vectors to query the encoder output in order to produce the target sequence. This type of architecture is very useful for language translation, where we need to

map an input sequence (e.g., in one language) to a target output sequence (e.g., in a different language).

2. **Encoder-only**: This transformer architecture only comprises of the encoder part. Crucially, this architecture is typically trained to predict missing elements from an entire input sequence (e.g., words) by randomly masking elements in the input sequence with the target sequence being the unmasked input sequence. This is like the typical 'filling the gap' exercises that English language courses include. This training regime allows transformer encoders to acquire a bi-directional understanding of the input sequence (unlike next-token predictions in decoder models - see below).

3. **Decoder-only**: This transformer architecture only comprises of the decoder part. These models are trained to perform next-token prediction in parallel. This is done by masking all the following elements in a sequence for each token and shifting the target sequence by one (i.e., so that for each token, the transformer tries to predict the following token). As a result, during training, for each token, the model can only use the elements in the sequence up to that token to predict the next one (i.e., achieving a form of auto-regressive training, but in parallel, with no need to iterate through the model predictions). During 'inference', these models are auto-regressive where the current model prediction is fed back as an input to predict the next element in the sequence. Note, the same masking training procedure is typically used for the encoder-decoder transformer.

To get a better idea about these different components, I implemented my own Encoder-Decoder Transformer, which includes both encoder and decoder components. The implementation is based on the original paper "Attention is all you need". You can find my commented code here.