Natural Language Understanding, Generation, and Machine Translation (2021–22)

School of Informatics, University of Edinburgh
Tom Sherborne

Tutorial 4: Transformers (Week 8)

Questions on this worksheet that ask for an expression or calculation generally have one correct answer, and are designed to concretely explore aspect of these models just beyond what we discussed in lecture, by asking you about some implications of model design. These questions are not intended to be difficult, but notice that they are not primarily about recalling information—they require you to engage with the material and pay attention to the details. You should expect that some of the *easier* exam questions may be of this form.

We also include some more open-ended questions that you will need to think about. It is intended to help you see how modelling choices impact the number of parameters and how this impacts your design decisions when applying the Transformer model to a task. Most of these questions are answerable from combined understanding of the lectures as well as your experience of Coursework 1. For a complete understanding, we advise reviewing Attention is all you Need, the paper which proposed the Transformer model, before beginning this tutorial.

1 Modelling a Transformer

A Transformer encoding layer consists of a multi-head self attention network and a feed-forward network with additional normalisation and skip-connection features. A Transformer decoder layer includes an additional multi-head attention network to incorporate information from the encoder during decoding.

Question 1:
Consider a single Transformer encoder layer using narrow self-attention. The self-attention projection size is 1024 (e.g. the size of $W_{\{q,k,v\}}$), the feed-forward projection size is 4096 and each layer has 16 heads.

   a. Calculate the number of weights in this single layer that will need to be learned? You can ignore normalisation parameters.

Students may look at the slides of Lecture 9 and then count up all the parameters in each sub-component of the Encoder model. This is obviously perfectly valid and ideally they should get the right number if they include all components. I'm showing here a roundabout way to calculate this using PyTorch that students may also use.

```
>>> from torch.nn import TransformerEncoderLayer
>>> d_model = 1024
>>> n_head = 16
>>> dim_ff = 4096
>>> dropout = 0.1
>>> encoder = TransformerEncoderLayer(d_model, n_head, dim_ff, dropout)
>>> for name, param in encoder.named_parameters():
...     print(name, param.shape, param.numel())

self_attn.in_proj_weight torch.Size([3072, 1024]) 3145728
```

```
self_attn.in_proj_bias torch.Size([3072]) 3072
self_attn.out_proj.weight torch.Size([1024, 1024]) 1048576
self_attn.out_proj.bias torch.Size([1024]) 1024
linear1.weight torch.Size([4096, 1024]) 4194304
linear1.bias torch.Size([4096]) 4096
linear2.weight torch.Size([1024, 4096]) 4194304
linear2.bias torch.Size([1024]) 1024
norm1.weight torch.Size([1024]) 1024
norm1.bias torch.Size([1024]) 1024
norm2.weight torch.Size([1024]) 1024
norm2.bias torch.Size([1024]) 1024
```

To get the number of weights we then want to sum up all the parameters not related to normalisation:

$$3145728 + 3072 + 1048576 + 1024 + 4194304 + 4096 + 4194304 + 1024 = 12,592,128$$

Note that if they did something akin to the following then this is wrong because of the normalisation parameters.

```
>>> sum(param.numel() for _, param in encoder.named_parameters())
12,596,224
```

They might have missed out the bias values which should result in a smaller value. If they missed out all bias terms then they should get 12,582,912. Strictly this is not wrong as you could make this network without biases but they should know they are there and usually a network will have bias terms they shouldn't forget. The multiplier of 3 (3072) vs expected value of 1024 everywhere is a convenient way to learn projections of query, key and value in 1 matrix. If you have time, you could consider that we don't actually need to know the number of heads to solve this as this value cancels out in narrow self-attention.

We now change the encoder to use *wide* self-attention with other parameters unchanged.

b. Calculate the number of weights in this new layer? Ignore normalisation parameters again.

For the previous question, we didn't need to really consider the number of heads as for h heads, we had $h \times \frac{k}{h}$ as the total dimensionality and thus only $k = 1024$ was relevant. We need to make two modifications for *wide* self attention. Firstly, each head is now of size k so we need to multiply the *output size* of the input projection weight and bias by h. Then, we also need to modify the *input size* of the output projection for self attention again by h. This should just end up being modifications of $\times 16$ to the previous calculation. You might disagree on how wide self attention should be implemented. So long as you can justify it, it probably is also a valid method.

$$3145728 \times \mathbf{16} + 3072 \times \mathbf{16} + 1048576 \times \mathbf{16}$$
$$+ 1024 + 4194304 + 4096 + 4194304 + 1024 = 75,552,768$$

Now we consider an encoder-decoder Transformer model for English sentence compression. The encoder and decoder each have six layers, as described above, and we also define a

vocabulary of 64,000 words with corresponding embeddings. Assume the Transformer layers use *narrow* self-attention, the embedding dimensionality is equal to the self-attention projection size and normalisation parameters can be similarly ignored.

   c. Calculate the number of weights in this full model. You will need to consider the encoder and decoder layers as well as additional input and output parameters required for this task.

We want them to consider the complete task here which might have 3 components:

   a. Embedding matrices for source and target languages

   b. A Transformer Encoder-Decoder model

   c. Linear output projection from the decoder to vocabulary dimension.

For (1), we have two (64000, 1024) matrices.
For (2), we can modify the count we just worked out for the encoder:

```
EncoderLayerParams*6 + (2*SelfAttnParams + FeedForwardParams)*6
e.g. 12592128 * 6 +
(2*(3145728+3072+1048576+1024)+4194304+4096+4194304+1024)*6
= 176,295,936
```

   For (3), we have a linear layer of size (1024, 64000) and a bias (64000,1).
   The total count for everything should be around 372,967,936

The input and output vocabularies for this task are equal as we are encoding and decoding English. Therefore, we can use the same embedding matrix for both the encoder and decoder, referred to as *tying* the embedding matrices, and learn one embedding matrix used for both encoder and decoder.

   d. What advantage does *tying* these embedding matrices together have in terms of the learned word representation?

   e. What is the percentage change in the number of weights to be learned from this change?

Students should notice that the embedding matrices dominate the number of weights we learn but they aren't doing any of the sequence modelling. On top of this, we can learn word representations from both source and target contexts (and understanding + generation). This is why tying embeddings can be useful as we save on weights and can learn from more contexts. Of course, you may have your own ideas and opinions on this.

Tying the embeddings removes one (64000, 1024) size matrix to be learnt. Therefore the total weights should be 307,431,936. This is a 17.6% saving of learnable weights compared to before.

Question 2:
We now want to compare theoretical complexities between different models used in NLP tasks. Inspect Table 1 in Attention is all you Need with a focus on the "Complexity per Layer" column wherein n is the sequence length and d is the representation dimension, the same parameter as the self-attention projection size from Q1.

a. Consider the complexity bounds and your own knowledge of how NLP tasks are constructed – describe when a Transformer network have lower complexity than other networks.

b. Other than complexity – describe other constraining factors to be considered when planning experiments using neural networks for NLP. There is no right answer here, state your own ideas.

For (a), we want to consider a problem of scale for NLP tasks. According to this paper, a Transformer has approximate complexity of $O(n^2.d)$, so we are quadratically proportional to sequence length and linearly proportional to dimensionality. The opposite is true for RNNs with $O(n.d^2)$

As a case study, we can set d=1024 and compare between these models. For most NLP tasks here we can approximate that $n << d$ and therefore the Transformer has lower complexity than the RNN. It might be helpful to name and discuss a few tasks where this is true. Conversely, there might be some cases where the inverse is true (document translation?). In terms of increasing n, the additional complexity in a Transformer makes sense as we need an additional self attention computation against all prior sequence elements. For the RNN, we only need to compute the RNN cell with the new state and cumulative history. The intended insight for students is to think in terms of practical values and make a measured decision concerning which network is best and when.

However for (b), we now want to add in the practical other constraints that exist when running experiments. You could discuss that really this complexity is only a part of choosing models as well as performance and constraints such as GPU Memory, data availability, tunable hyperparameters etc. A typical Transformer is much larger than an RNN and space/speed constraints might arise before complexity can even be considered. This is an opportunity for students to share their own ideas.

---

2 Considering Permutations

The Transformer self-attention routine operates on *sets* of inputs without respect for sequence ordering. This model will produce identical outputs with varying combinations of inputs becayse the attention states between any two words will be the same regardless of word position. This gives the Transformer some interesting mathematical properties we want you to think about here.

Question 3:

a. For a transformer encoder model without positional embeddings – explain why the model is permutation **equivariant** but not **invariant**.

For encoder input, [A, B, C], into encoder f(X), producing [X, Y, Z] outputs - the output of [B, C, A] into f(X), will be [Y, Z, X].

More generally, equivariance for some permutation function $r \longrightarrow r(f(x)) = f(r(x))$.

> Note that this is slightly different to how the slides portray the problem and they state "the input is position invariant" which may confuse some. The inputs themselves have no sequence information without position embeddings (See Q3(d)) and the attention **for a single output state** is invariant to where the other words in the sequence are. The full Transformer is equivariant for the reason above.

Consider this model with an additional max pooling layer on the output to combine the outputs to produce one output vector.

    c. Explain why the whole model is now permutation **invariant** and not **equivariant**.

> Extending the previous result, the max-pooling of [X, Y, Z] will be the same as the max-pooling of [Y, Z, X]. It doesn't matter what order the inputs are in so the model is invariant to permutations.
>
> More generally, invariance for some permutation function $r \longrightarrow f(x) = f(r(x))$.

These properties are not desirable for sequence modelling in natural language processing. For this reason, we augment the inputs with **positional embeddings** to provide additional information to the input.

    d. What additional information is provided with this addition and how does it help sequence modelling? Explain your answer in relation to how the properties described above are affected.

> Positional embeddings provide exactly what their name states – they imbue the model input with positional information so that each vector is unique representing both semantic information and a token's position in the sequence. Therefore, a sentence such "I can can a can" won't end up with three identical representations for "can" and each representation is contextual on both adjacent words and word order.

Final discussion question: With this additional augmentation – what is the special desirable property the model now achieves in a sequence-to-sequence framework?

> The desirable property is that the model is a universal approximator for sequence-to-sequence functions since inputs and outputs are now unique given the additional information of positional embeddings. This result is based on this extremely complicated paper which I don't recommend mentioning but is put it here for completeness. We wouldn't expect students to arrive at this conclusion without help and this final question is more of a closing discussion point.