Bastien LE CHENADEC
bastien.le-chenadec@eleves.enpc.fr

Lab session # 3
ALTEGRAD 2023

11/29/23

1 Question 1

During training we want the model to predict the next token given the previous tokens. Thus we need masks to prevent the model from just looking at the future tokens.

Looking at the equation for the attention mechanism (1), we can see that the softmax function is applied to the dot product of the query and key vectors. If we set the future tokens to $-\infty$ in the dot product, the softmax function will output 0 for the future tokens. This is exactly what we want, since an attention of 0 means those tokens will be ignored.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}V\right) \tag{1}$$

Concerning the positional encoding, it is necessary in most transformer architectures because the model has no notion of order in the input sequence. However the order of the tokens matter for most tasks, including classification. Consider the following example, where the model has to predict the sentiment of the sentence.

- "I am happy, am I not ?"
- "I am not happy, am I ?"

These two sentences correspond to the same words in different orders. However they have opposed meanings in terms of sentiment. Thus the model needs to know the order of the words to perform well on this task.

2 Question 2

To perform classification instead of text prediction, we need to replace the classification head. This is for one part due to dimensions mismatch :

- The text prediction head takes as input a sequence of vectors of dimension d_{model} and outputs a sequence of vectors of dimension d_{vocab}.
- The classification head takes as input a sequence of vectors of dimension d_{model} and outputs a single vector of dimension $d_{\text{classes}} \ll d_{\text{vocab}}$.

Furthermore, the text prediction head is trained to predict the next token given the previous tokens. There is no reason why its weights would be optimal for classification. The point of training on text prediction is only to do self-supervised learning, i.e. to learn a good representation of the input sequence without having to label our data. This representation can then be used for other tasks, such as classification.

3 Question 3

How many trainable parameters does the model have ? Let's count them for each step considering the following variables :

- d_{model} : dimension of the embeddings (200)
- d_{vocab}: size of the vocabulary (50001)
- d_{ff}: internal dimension of the feed forward layers (200)
- n_{heads}: number of heads in the multi-head attention mechanism (2)

- n_{layers}: number of layers in the encoder (4)
- n_{classes}: number of classification classes (2)

Embedding :

A single matrix of size $d_{\text{vocab}} \times d_{\text{model}}$ according to the pytorch documentation of `Embedding`.

Positional encoding :

Looking at the positional encoding implementation we can see that it is deterministic and there are no trainable parameters.

Single transformer encoder layer :

Looking at the source code of `TransformerEncoderLayer` :

- First feed-forward `Linear` layer : $d_{\text{model}} \times d_{\text{ff}} + d_{\text{ff}}$.
- Second feed-forward `Linear` layer : $d_{\text{ff}} \times d_{\text{model}} + d_{\text{model}}$.
- First `LayerNorm` : $2 \times d_{\text{model}}$.
- Second `LayerNorm` : $2 \times d_{\text{model}}$.
- `MultiheadAttention` : in the pytorch implementation, the dimensions of the query, key and value matrices for each head are $d_{\text{model}} \times (d_{\text{model}}/n_{\text{heads}})$. Thus there is a total of $n_{\text{heads}} \times 3 \times d_{\text{model}} \times (d_{\text{model}}/n_{\text{heads}}) = 3 \times d_{\text{model}}^2$ parameters for the heads. The output matrix is of shape $d_{\text{model}} \times d_{\text{model}}$. The default pytorch implementation also adds $3 \times d_{\text{model}}$ bias parameters for the query, key and value matrices, and d_{model} bias parameters for the output matrix. Thus the total number of parameters for the `MultiheadAttention` is $4 \times d_{\text{model}}^2 + 4 \times d_{\text{model}}$. It does not depend on the number of heads.

Transformer encoder :

We simply need to multiply the number of parameters of a single transformer encoder layer by the number of layers.

Language modelling task :

We add a `Linear` layer with $d_{\text{model}} \times d_{\text{vocab}} + d_{\text{vocab}}$ parameters.

The total comes to :

$$d_{\text{vocab}} \times d_{\text{model}} + n_{\text{layers}} \times (d_{\text{ff}} \times (d_{\text{model}} + 1) + d_{\text{model}} \times (d_{\text{ff}} + 1) + 8 \times d_{\text{model}} + 4 \times d_{\text{model}}^2) + d_{\text{vocab}} \times (d_{\text{model}} + 1)$$

Application : $\boxed{21018401}$

Classification task :

We add a `Linear` layer with $d_{\text{model}} \times n_{\text{classes}} + n_{\text{classes}}$ parameters.

The total comes to :

$$d_{\text{vocab}} \times d_{\text{model}} + n_{\text{layers}} \times (d_{\text{ff}} \times (d_{\text{model}} + 1) + d_{\text{model}} \times (d_{\text{ff}} + 1) + 8 \times d_{\text{model}} + 4 \times d_{\text{model}}^2) + n_{\text{classes}} \times (d_{\text{model}} + 1)$$

Application : $\boxed{10968602}$

This is exactly what we get when running :
`print(sum(p.numel() for p in model.parameters() if p.requires_grad))`

Most of the trainable weights in our case come from the embedding and unembedding layers, because d_{model} is small.

4 Question 4

It is clear that having a pre-trained model is better than training from scratch. The accuracy is pretty high from the get-go. However when training from scratch, the model quickly catches up in terms of accuracy.

In terms of accuracy after 15 epochs, the pre-trained model is slightly better than the model trained from scratch. However the difference is not that big. The pre-trained model has an accuracy of 0.79, while the model trained from scratch has an accuracy of 0.75.

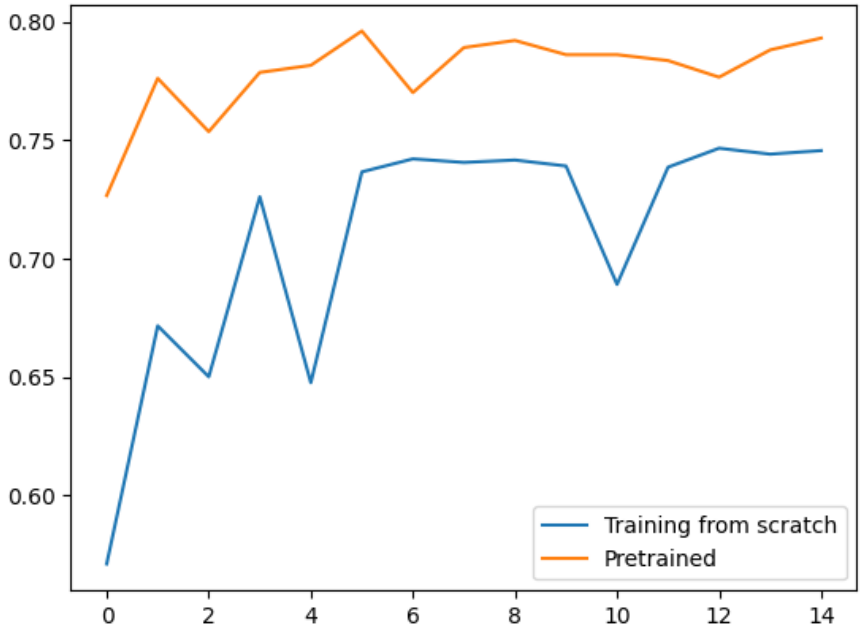

Figure 1: Validation accuracy for the pre-trained model and the model trained from scratch.

These disappointing results should not lead us to conclude that pre-training is useless. Indeed the task at hand is pretty easy (two-classes classification) and could potentially be learned only from the presence or absence of some words in the sentence. Thus the model does not need to learn a good representation of the input sequence to perform well on this task. However for more complex tasks, a good representation of the input sequence is crucial. In this case, pre-training is very useful.

5 Question 5

In this notebook, we pre-trained the model on an unidirectional language modelling task. This means that the model is trained to predict the next token given the previous tokens. In [1], the authors argue that this is not optimal for tasks that require bidirectional context, such as question answering or sentence-level tasks. In this paper, they introduced a masked language modelling task, where the model is trained to predict the masked tokens given the other tokens. This allows the model to learn bidirectional context and overcome the limitations of unidirectional language modelling.

References

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. 2018.