Martin JOLIF
martinjolif@gmail.com

Lab session # 2
ALTEGRAD 2024

10/19/24

1 Question 1

To train our classification model, we first have to use a generative-pre-training. In this phase the language model learns to predict the next token given the previous ones. To this end, we use an entire sentence and hide the end of this sentence and try to predict the next token given the past elements. The role of the **square mask** is to hide the end of the sentence. It's done by putting their attention weights to -infinity. Therefore, the attention from the transformers only considers the past tokens and can predict the next token correctly (without considering future tokens).

The role of the **positional-encoding** is to add the position information of the tokens in the sentence before applying the transformer architecture. Indeed, the transformer architecture isn't based on recurrent neural networks and takes not into account the positions of the words as input. Therefore, to take in account the positions of the tokens, we add this information through the positional-encoding which is very important to understand correctly natural language.

2 Question 2

We have two different goals which are not exactly the same :

- **Language-modeling** First we want to train a model that has to predict the next token given the previous ones. In that case, our output is a vector of length : n_{tokens}. The token corresponding to the index with the highest value of the output vector is the predicted next token.
- **Classification** The second objective is to classify book reviews into positive or negative ones. In that case, our output vector is a vector of length : $n_{classes} = 2$. The index of the output vector with the highest value is the predicted class.

For the training phase, the two tasks have the same architecture, but the size of the last linear layer is therefore not the same between the language-modeling ($nhid \rightarrow n_{tokens}$) and the classification ($nhid \rightarrow n_{classes} = 2$) tasks.

Moreover, for the inference phase, the language modeling output should be shifted into the input at every prediction step instead of a one shot prediction in the case of the classification task.

This is why we have to replace the classification head depending on the task.

3 Question 3

First we will compute the numbers of trainable parameters of the base model :

- The **embedding layer** has $num\_embedding * embedding\_dim = n\_token * nhid = 50,001 * 200 = 10,000,200$ trainable parameters.
- The **Positional encoding** layer has no trainable parameters.
- For the **transformer block**. The transformer encoder architecture is composed of 1 multi-head-attention blocks, 2 normalization layers and 1 feed-forward-network (see figure 2).
  - According to [2] the multi-head attention block has the following number of parameters:

$$(d_{model}^2 + d_{model}) + (d_{model} * \frac{d_{model}}{nhead} + \frac{d_{model}}{nhead}) * nhead * 3 = 160,800 \tag{1}$$

    The times 3 at the end comes from the three matrix W^Q, W^K, W^V.
  - A normalization layer has $2 * d_{model} = 400$ parameters. So it gives 800 for 2 normalization layers.
  - The feed-forward-network is composed of 2 linear layers which gives :
    $(d_{model} * d_{ff} + d_{ff}) + (d_{ff} * d_{model} + d_{model})$ where in our case $d_{ff} = nhid = 200$. It gives $80,400$ parameters.

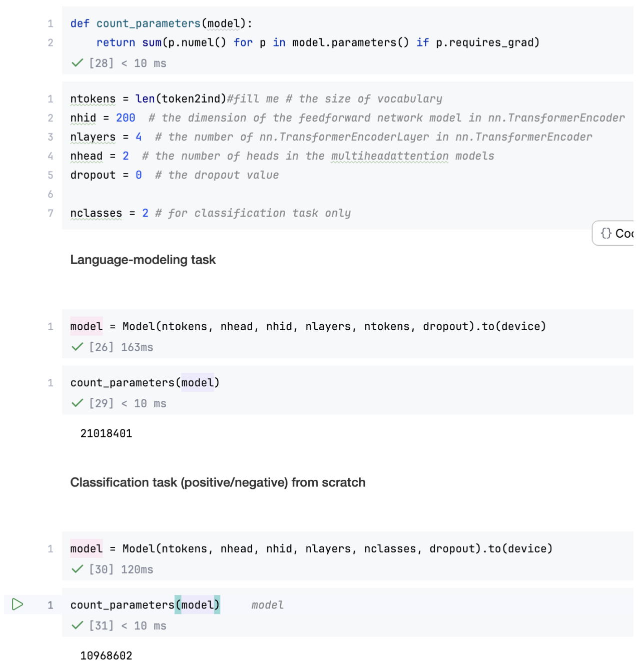
So, in total, a transformer block has $242,000$ parameters. For 4 transformers blocks, it gives $968,000$ trainable parameters.

So, the base model has $10,968,200$ parameters. We have to add the final linear layer of size $nhid * nclasses + nclasses = 10050201$ in the language modeling task ($nclasses = ntokens = 50,001$) and 402 in the classification task ($nclasses = 2$).

So finally,

- The **language-modeling** model has $21,018,401$ parameters.
- The **classification** model has $10,968,602$ parameters.

This is confirmed by PyTorch, by computing the number of trainable parameters (see figure 1).



```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
✓ [28] < 10 ms
```

```
ntokens = len(token2ind)#fill me # the size of vocabulary
nhid = 200  # the dimension of the feedforward network model in nn.TransformerEncoder
nlayers = 4  # the number of nn.TransformerEncoderLayer in nn.TransformerEncoder
nhead = 2  # the number of heads in the multiheadattention models
dropout = 0  # the dropout value

nclasses = 2 # for classification task only
```

Language-modeling task

```
model = Model(ntokens, nhead, nhid, nlayers, ntokens, dropout).to(device)
✓ [26] 163ms
```

```
count_parameters(model)
✓ [29] < 10 ms

21018401
```

Classification task (positive/negative) from scratch

```
model = Model(ntokens, nhead, nhid, nlayers, nclasses, dropout).to(device)
✓ [30] 120ms
```

```
count_parameters(model)    model
✓ [31] < 10 ms

10968602
```

Figure 1: Number of parameters of the models given by PyTorch

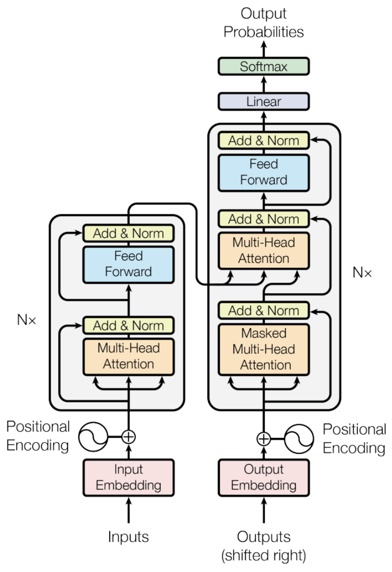


Figure 2: Transformer architecture from [4]

4 Question 4

According to figure 3, the accuracy of the pretrained model is always higher than the accuracy of the scratch model. Moreover, we observe that pretrained model starts with a relatively high accuracy of more than 75% and reaches quickly around the 5^{th} epoch with a maximum of 80%. On the other hand, the model trained from scratch starts with a bad accuracy of 55% (almost random choice) and increases rapidly around 73% at epoch 2 and has difficulties increasing at future epochs. That's confirm the says of [3], it's better to do a pretrained model and then do a transfer learning for a particular task.

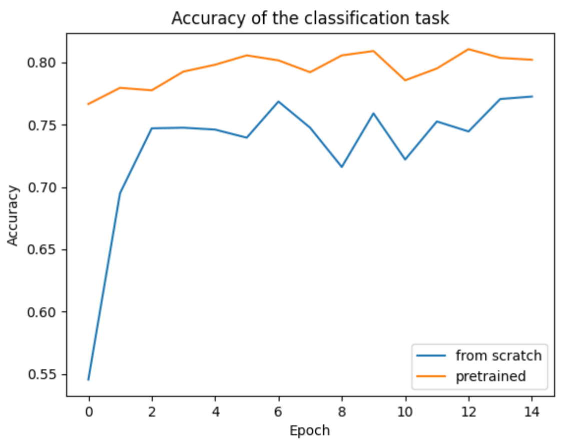


Figure 3: Accuracy of the classification task depending on the model and the epoch of training

5 Question 5

In this notebook, we use unidirectional language models to learn general language representations. The major issue with this idea is that every token can only attend to previous tokens in the self-attention layers of the Transformer (left-to-right architecture). This is not optimal to have a good understanding of natural language: words can be dependent on context from both directions, past and future. The authors of the paper [1] are solving this limitation by applying a Bidirectional Encoder Representation from Transformers (BERT). Instead of applying a mask to the end tokens as we do in our model, they hide random tokens from the input and try to predict them given the context from both future and past elements.

References

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. `https://arxiv.org/abs/1810.04805`, 2019.

[2] Dmytro Nikolaiev (Dimid). How to estimate the number of parameters in transformer models. `https://towardsdatascience.com/how-to-estimate-the-number-of-parameters-in-transformer-models-ca0f57d8dff0`, 2019.

[3] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need.