VERSINI Lucas
lucas.versini01@gmail.com





# 1 Question 1

During pretraining, the model will be given pieces of sentences (i.e., a list of tokens), and will have to predict the next token. Therefore the model must not have access to this token to be predicted, nor to the next ones. The mask prevents the model from accessing these ”future” tokens.

If we simply encode each token by an integer, then it means that when a word appears twice in some text, it will be encoded the same way both times. As a consequence, the order of the source tokens will have absolutely no impact on the way the input is treated by the model: the initial order of the tokens will be lost. To prevent this, we add a positional encoding to each token (which then contains not only the information on the initial word/token, but also its position in the sentence). This is done in the *PositionalEncoding* class, using $\cos$ and $\sin$ functions.

# 2 Question 2

- During pretraining, the model is given a sequence of tokens, and has to predict the next one. In fact, it outputs a vector whose size is the length of the vocabulary dictionary, and the predicted token is the token corresponding to the highest coefficient in the output.
- During training and when classifying a book review, the model outputs two numbers, corresponding to ”bad” and ”good” predictions (that is, we can turn this 2-dimensional vector into a probability distribution on the set {Good, Bad}).

So the main difference between the two heads is the output dimension.

# 3 Question 3

Let us see how many trainable parameters each part of the model has.

- The encoder has `num_embeddings` $\times$ `embedding_dim` $=$ `ntoken` $\times$ `nhid` trainable parameters;
- The positional encoder has no trainable parameter;
- For the transformer encoder, there are `nlayers` layers, and for each of these layers:
  - There are two linear layers, each with `nhid` $\times$ `nhid` weights and `nhid` biases, so each with `nhid` $\times$ (`nhid` $+ 1$) trainable parameters, hence $2 \times$ `nhid` $\times$ (`nhid` $+ 1$) trainable parameters;
  - There are two normalization layers, each with $2 \times$ `nhid` trainable parameters (weight and bias), hence $4 \times$ `nhid` parameters;
  - For the self-attention, there are $3 \times$ `nhid` $\times$ (`nhid` $+1$) parameters for $Q, K, V$, and `nhid` $\times$ (`nhid` $+1$) parameters for a projection, hence $4 \times$ `nhid` $\times$ (`nhid` $+ 1$) parameters.

So the body of the model has `ntoken` $\times$ `nhid` $+$ `nlayers` $\times (6 \times$ `nhid` $\times$ (`nhid` $+ 1) + 4 \times$ `nhid`$)$ trainable parameters, which is equal to `nhid` $\times$ (`ntoken` $+$ `nlayers` $\times (6 \times$ `nhid` $+ 10))$.

As for the head, it has $(1 +$ `nhid`$) \times$ `nclasses` parameters (`nclasses` being the dimension of the output, which in our case is either 2 or `ntoken` $= 100$).

So we have:

$$\boxed{\texttt{nhid} \times (\texttt{ntoken} + \texttt{nlayers} \times (6 \times \texttt{nhid} + 10)) + (1 + \texttt{nhid}) \times \texttt{nclasses}} \text{ trainable parameters.}$$

For `ntoken` $= 100$, `nhid` $= 200$, `nlayers` $= 4$, `nclasses` $\in \{2,$ `ntoken` $= 100\}$, we find:
$\boxed{1,008,100 \text{ parameters for the language modeling task}}$, and $\boxed{988,402 \text{ parameters for the classification task}}$.

We can even verify these values in Python, as can be seen in Fig. 1.

```python
ntokens = 100 # the size of vocabulary
nhid = 200  # hidden dimension
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
```

✓ 0.0s Python

```python
nclasses = ntokens
model_modeling = Model(ntokens, nhead, nhid, nlayers, nclasses, dropout).to(device)

# Compute number of parameters
num_params = sum(p.numel() for p in model_modeling.parameters())
print(f"Number of parameters: {num_params}")

# Compute number of parameters
print(f"Number of parameters: {nhid * (ntokens + nlayers * (6 * nhid + 10)) + (1 + nhid) * nclasses}")
```

✓ 0.0s Python

```
Number of parameters: 1008100
Number of parameters: 1008100
```

```python
nclasses = 2
model_classification = Model(ntokens, nhead, nhid, nlayers, nclasses, dropout).to(device)

# Compute number of parameters
num_params = sum(p.numel() for p in model_classification.parameters())
print(f"Number of parameters: {num_params}")

# Compute number of parameters
print(f"Number of parameters: {nhid * (ntokens + nlayers * (6 * nhid + 10)) + (1 + nhid) * nclasses}")
```

✓ 0.0s Python

```
Number of parameters: 988402
Number of parameters: 988402
```

Figure 1: Number of trainable parameters.

# 4 Question 4

The evolution of the accuracy for the two models on the classification task can be seen in Fig. 2.
We see that the model trained from scratch starts with an accuracy near $60\%$: this model has never been trained on text before, so it is initially not very good at classifying. However, the accuracy quickly increases to more than $75\%$, and then oscillates, without evolving much.

On the contrary, the pretrained model already starts with a high accuracy (near $75\%$), and though it increases during the first epochs, it then does not change much either.

So without much surprise, the pretrained model is better than the model trained from scratch, even in just one epoch.

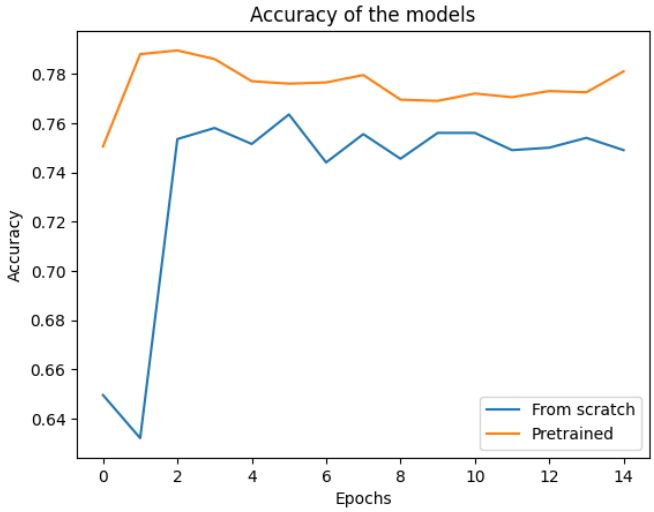


Figure 2: Accuracy for a pretrained model and a model trained from scratch.

# 5 Question 5

The model we used can only use information from the left to predict what comes to the right (this is done by the mask, as seen in Question 1.).
In [1], a bidirectional model (i.e., which uses context from the left and from the right) is introduced, yielding better results in several datasets, as explained in the article.
The idea is that a bidirectional model has access to potentially much more context than a unidirectional one, which can obviously highly improve the results. We can imagine a sentence like ”Everyone believed he was guilty, ____ the jury acquitted him after just an hour of deliberation.” A unidirectional model would likely predict ”because” or ”so”, while a bidirectional one would more likely predict ”yet”.

# References

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of naacL-HLT*, volume 1, page 2. Minneapolis, Minnesota, 2019.