Question 1

In the implementation, the square mask is used to prevent the model from attending to future positions in the input sequence during the self-attention mechanism. This is necessary because the self-attention mechanism computes a weighted sum of the input sequence, and we want the model to focus on the current position and the previous positions, but not the future positions.

On the other hand, The positional encoding is used to inject information about the position of each word in the input sequence into the model. Since the self-attention mechanism does not have any inherent notion of word order, the positional encoding is added to the input embeddings to provide this information.

Question 2

The main difference between language modeling and classification tasks lies in their objectives and output structures. In language modeling, the model predicts the next token in a sequence, outputting a probability distribution over the vocabulary for each word, requiring a `nn.Linear(hidden_size, vocab_size)` head. In contrast, classification tasks aim to predict a single class for the entire sequence, needing a `nn.Linear(hidden_size, num_classes)` head. Hence, we replace the classification head to map the sequence representation to the required class probabilities, usually following some pooling of the hidden states.

Question 3

Let's compute the learnable parameters layer by layer.

- *language modeling* task.

  In the `Embedding block`, we have $n_{token} \times n_{hid}$ parameters. In our case, $n_{token} = 50,001$ and $n_{hid} = 200$. Thus, we have in total **10,000,200 learnable parameters** for the `Embedding block`.

  The `Positional encoding` part does not have trainable parameters.

  Let's break into the `Transformer layers`. To understand it, we will compute the parameters of one transformer layer. For one `Transformer layer`:

  - `Self-attention`:
    * W_Q, W_K, W_V: (n_{hid}, n_{hid}) each. We have thus $200 \times 200 \times 3 = 120,000$ parameters that can be learnable.
      In addition, we have the associated bias: $200 \times 3 = 600$.
    * projection back to the n_{hid} dimension: $200 \times 200 = 40,000$.
      With again a bias of 200 parameters.

    In total, for the `Self-attention` layer, we have $120,000 + 600 + 40,000 + 200 =$ **160,000** learnable parameters.
  - `Feedforward`:
    * normalization 1: $weights = 200$, $bias = 200$, so we have 400 parameters.
    * normalization 2: $weights = 200$, $bias = 200$, so we have 400 parameters.
    * linear 1: $n_{hid} \times n_{hid} + n_{hid} = 200 \times 200 + 200 = 40,200$ parameters.
    * linear 2: same as above, $40,200$ parameters to learn.

    In total, for `Feedforward`, we have $400 \times 2 + 40,200 \times 2 =$ **81,200** learnable parameters.

  Thus, we can compute all the possible trainable parameters for one `Transformer layer`, and it's $160,000 + 81,200 =$ **261,200**.

Now that we have number of parameters for one transformer layer, we can have it for 4 transformer layers, and we obtain $4 \times 261,200 =$ **968,000 parameters**.

The last thing to compute is the number of learnable parameters in the last layer of the model architecture, the `Classification Head`. In there, it's just a linear layer with $W \in \mathbb{R}^{n_{classes} \times n_{hid}}$ and $b \in \mathbb{R}^{n_{classes}}$. Thus, we have $50,001 \times 200 + 50,001 =$ **10,050,020** parameters to learn.

Finally, in the case of *language modeling* task, the model does have

$10,000,200 + 968,000 + 10,050,020 =$ **21,018,401**

learnable parameters.

- *classification* task For this task, it's basically almost the same as above, we just need to change the `Classification head`, that outputs now 2 instead of $n_{classes} = 50,001$. We have thus $2 \times 200 + 2 = 402$ learnables parameters.

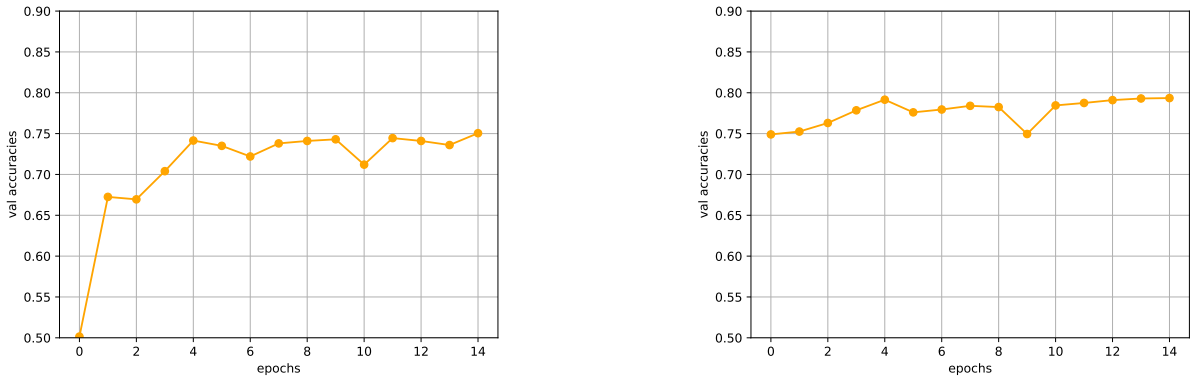
Finally, in the case of *Classification* task, the model does have

$10,000,200 + 968,000 + 402 =$ **10,969,602**

learnable parameters.

Question 4

Results of Task 7:



(a) Evolution of val accuracy of the from scratch model trained during 15 epoch.

(b) Evolution of val accuracy of the pretrained model trained during 15 epoch.

Figure 1: Val accuracies observation of the 2 tested models.

Interpretation of the results:

The pretrained model generally performs better, with a higher peak accuracy $(0.81$ vs $0.76)$ and average. The pretrained weights thus provide a better starting point, allowing the model to learn more effectively.
As the learning rate is really small, those observations are quite normal, with a very low improvement of the val accuracy after 5-6 epochs.

Additionnaly, the stabilization at 0.75 of the scratch model suggests that the model has difficulty to classify correctly. This could be due to insufficient training data (arround 1600 normal sentences), or maybe tuning better the hyperparameters that are tuned for fine-tuning and transfert learning. On the other hand, the

pretrained model is better at distinguishing the two classes. However, this model has also variability in accuracy, and may need also more training data.
For both models, the tendancy of constant evolution during epoch might be due to overfitting from training data.

The pretrained model shows promise with higher peak accuracy, indicating that transfer learning is beneficial for your task. Further fine-tuning, data augmentation, and hyperparameter tuning can help improve both models' performance.

Question 5

Transformers used for language model pre-training are usually implemented with as a left-to-right architecture, which means that every tokens can only have informations from previous ones. In this notebook, we used the `TransformerEncoderLayer` from Pytorch, based on a traditional left-to-right approach. For language modeling tasks, it's well efficient. One problem is that is less adaptable when using those models for other specfic tasks. For instance, for what we do in this notebook, sentiment analysis, it could be more efficient to have access to all tokens of the sentences, as we want a general understanding of the text review and not just predict the futur word.

One solution proposed in [1] is to use a bidirectional approach, such as Bidirectional RNNs/LSTMs (we can refer to the previous Lab session). Introduced in the BERT paper ([1]), they present masked language models, by masking tokens and predicting them based on the entire context (left and right) during the unsupervised task. This could lead to a better understanding of the global review. This kind of limitation task can be known as representation learning.
With a unidirectional Language modelling, representation learning may be less robust for tasks that require understanding of the context (from both direction). In comparaison, Masked Language models can learn easily more contextually representations, as it must understand the entire sequence to predict the masked tokens.

References

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