1 $RoBERTa^{fr}_{small}$ - Number of parameters

The first model that was studied in the lab was the $RoBERTa^{fr}_{small}$ [7]. When we look in the architecture in the path given, we find that there are 4 encoder layers of dimension 512 for the embedding dimension and 512 for the ffn dimension. In addition, there are 8 encoder attention heads. This is displayed in Fig 1.

```
RobertaHubInterface(
  (model): RobertaModel(
    (encoder): RobertaEncoder(
      (sentence_encoder): TransformerEncoder(
        (dropout_module): FairseqDropout()
        (embed_tokens): Embedding(32000, 512, padding_idx=1)
        (embed_positions): LearnedPositionalEmbedding(258, 512, padding_idx=1)
        (layernorm_embedding): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
        (layers): ModuleList(
          (0-3): 4 x TransformerEncoderLayerBase(
            (self_attn): MultiheadAttention(
              (dropout_module): FairseqDropout()
              (k_proj): Linear(in_features=512, out_features=512, bias=True)
              (v_proj): Linear(in_features=512, out_features=512, bias=True)
              (q_proj): Linear(in_features=512, out_features=512, bias=True)
              (out_proj): Linear(in_features=512, out_features=512, bias=True)
            )
            (self_attn_layer_norm): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
            (dropout_module): FairseqDropout()
            (activation_dropout_module): FairseqDropout()
            (fc1): Linear(in_features=512, out_features=512, bias=True)
            (fc2): Linear(in_features=512, out_features=512, bias=True)
            (final_layer_norm): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
          )
        )
      )
      (lm_head): RobertaLMHead(
        (dense): Linear(in_features=512, out_features=512, bias=True)
        (layer_norm): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
      )
    )
    (classification_heads): ModuleDict()
  )
)
```

Figure 1: Roberta Small Architecture.

1.1 Embedding layers

To sum-up the model there are the **embedding layers** with the token embeddings and the positional embeddings, there are represented in Fig 2 [6] :

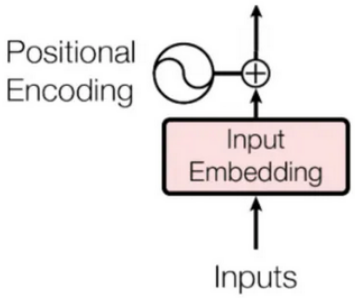

Figure 2: Embedding Layers. Adapted from [6].

$$
\begin{aligned}
n_{\text{token\_embeddings}} &= \text{vocab\_size} \times \text{embedding\_dimension} \\
&= 32000 \times 512 \\
&= 16,384,000
\end{aligned} \tag{1}
$$

$$
\begin{aligned}
n_{\text{positional\_embeddings}} &= \text{maximum\_sequence\_length} \times \text{embedding\_dimension} \\
&= 258 \times 512 \\
&= 132,096
\end{aligned} \tag{2}
$$

$$
\begin{aligned}
n_{\text{embedding\_layers}} &= n_{\text{token\_embeddings}} + n_{\text{positional\_embeddings}} \\
&= 16,384,000 + 132,096 \\
&= 16,516,096
\end{aligned} \tag{3}
$$

1.2 Transformer layers

Then, there are the transformer layers. For each of them, there is the multi-head self-attention which consists of query, key, and value matrices for each head. Then there is the feed-forward networks with two linear transformations.

Concerning the multi-head attention layers a nice explanation of the calculations of its parameters is displayed in Fig 3.

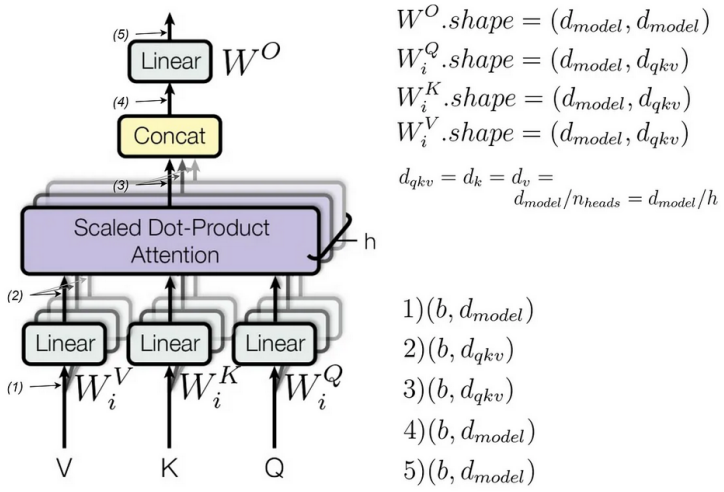

Figure 3: Multi Head Attention Layers architecture. Adapted from [5].

So we have the key, queries and values (K,Q & V). h is the number of heads (in our case it is 8). So for every multi-attention head, the number of parameters is the dot value of the three matrices. Furthermore $d_{qkv} = d_{model}/num_{heads}$. Therefore $N_{attention} = 3 \times n_{heads} \times (d_{model} \times d_{qkv}) + (d_{model} \times d_{model})$. As with the previous equality, $N_{attention} = 3 \times (d_{model} \times d_{model}) + (d_{model} \times d_{model})$.

$$
\begin{aligned}
n_{\text{attention\_input}} &= \text{embedding\_dimension} \times \text{embedding\_dimension} \times 3 \\
&= 512 \times 512 \times 3 \\
&= 786,432
\end{aligned} \tag{4}
$$

$$
\begin{aligned}
n_{\text{attention\_output}} &= \text{embedding\_dimension} \times \text{embedding\_dimension} \\
&= 512 \times 512 \\
&= 262,144
\end{aligned} \tag{5}
$$

For the fully connected layer, we can refer to this representation in Fig 4:
Taking this representation, we can easily compute that $N_{feedforward} = (d_{model} \times d_{fully\_connected}) + (d_{fully\_connected} \times d_{model}) = 2 \times (d_{model} \times d_{fully\_connected})$

$$
\begin{aligned}
n_{\text{fully\_connected\_layers}} &= 2 \times \text{ffn\_dimension} \times \text{embedding\_dimension} \\
&= 2 \times 512 \times 512 \\
&= 524,288
\end{aligned} \tag{6}
$$

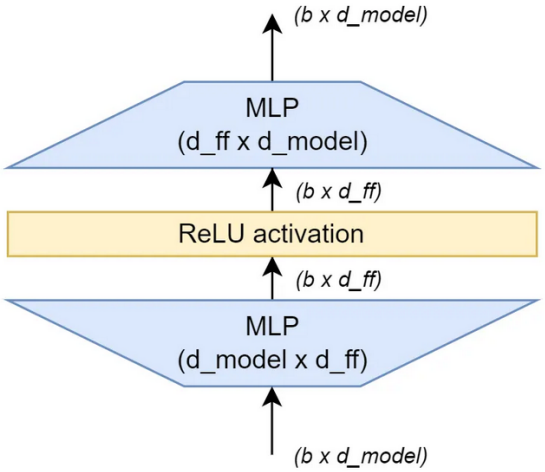

Figure 4: Feed forward Layer architecture. Adapted from [5].

$$\begin{aligned} n_{\text{transformer\_layers}} &= 4 \times (n_{\text{attention\_input}} + n\text{attention\_output} + n_{\text{fully\_connected\_layers}}) \\ &= 4 \times (786,432 + 262,144 + 524,288) \\ &= 4 \times (1,572,864) \\ &= 6,291,456 \end{aligned} \tag{7}$$

1.3 Final result

Finally, we have that :

$$\begin{aligned} n_{\text{parameters}} &= n_{\text{embedding\_layers}} + n_{\text{transformer\_layers}} \\ &= 16,516,096 + 6,291,456 \\ &= 22,807,552 \end{aligned} \tag{8}$$

It corresponds to the result that we obtained numerically !

2 LoraConfig - Parameters

LoRA, or Low-Rank Adaptation [3], [1], is a method for fine-tuning LLMS efficiently. The idea is to introduce trainable low-rank matrices to adapt the weights of pre-trained models without modifying the original weights. This is called 'parameter efficient fine-tuning' methods. To understand it rapidly, pretrained weights can be represented as the product of two small matrices, so the updating of these two small matrices make the program more computationally efficient. This principle is represented in Fig 5.

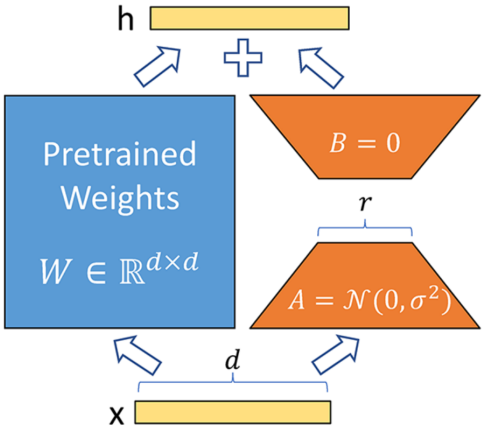

Figure 5: Lora principle, training of A and B. Adapted from [3].

Here are the parameters that can be found in the model, there are presented in Fig 6:

```
"r": 8,
"lora_alpha": 16,
"lora_dropout": 0.05,
"target_modules": ["q_proj", "v_proj", "k_proj", "o_proj", "gate_proj",
"up_proj", "down_proj", "embed_tokens", "lm_head"],
"modules_to_save": [],
```

Figure 6: Lora hyperparameters. Adapted from [1].

- **r**: This is the rank of the adaptation matrices. It is defined as the rank of the low-rank matrices that are used to approximate the changes to the original weights of the model. A smaller rank means fewer parameters and less capacity for adaptation, while a higher rank increases the capacity for adaptation but also increases the number of trainable parameters.
- **lora alpha**: This is the scaling factor for the LoRA layers. It determines to which extent the low-rank matrices can adapt the original weights. A higher lora alpha will make available a larger adaptations to the pre-trained weights, while a smaller value will restrict these adaptations.
- **target modules**: Specifies the layers of the transformer model to which LoRA is applied. Common targets are the attention layers, specifically the query and key projections within the multi-head attention modules. However, as explained in [1], to apply to additional layers might improve performances.
- **lora dropout**: It is the dropout rate applied to the LoRA layers. This parameter will help to prevent overfitting by randomly setting a fraction of the output units of the layers to zero during training.
- **learning rate**: With a high learning rate sometimes the loss was not stable.

Each of these parameters plays a role in how LoRA adapts the pre-trained weights of a large language model. The specific values for these parameters can be chosen regarding what is the size of the model that is being fine-tuned or the computational resources.

New methods have recently emerged to perform even better than Lora, this is the case with **QLoRA**, [2], [8]. The different architectures is presented in Fig 7. Indeed, QLora introduces quantization techniques to the low-rank matrices. The values of these matrices will be reduced to a discrete set.

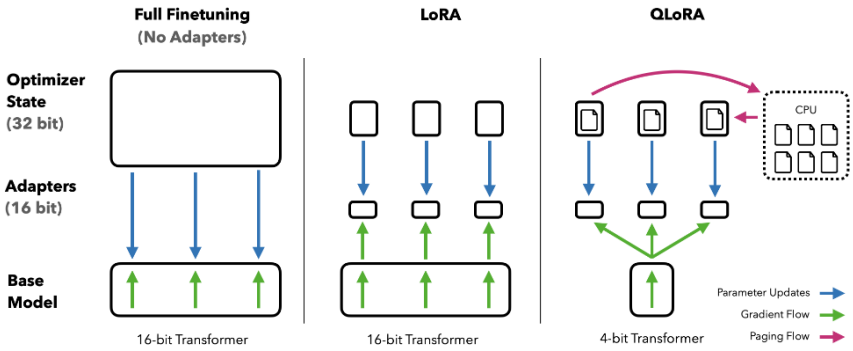

Figure 7: Comparison between LoRA and QLoRA. Adapted from [2].

3 Tensorboard results - Fairseq

3.1 Accuracy results

After the finetune of ROBERTasmall with Fairseq, we obtain these accuracies results presented in Tensorboard (cf Fig 8 and Fig 9).:

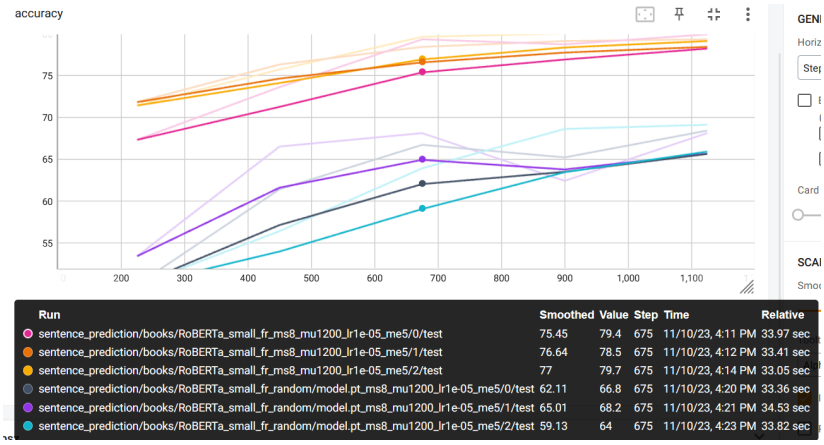

Figure 8: Tensorboard results of accuracy for the six tests of finetuning, smoothness of 0.6.

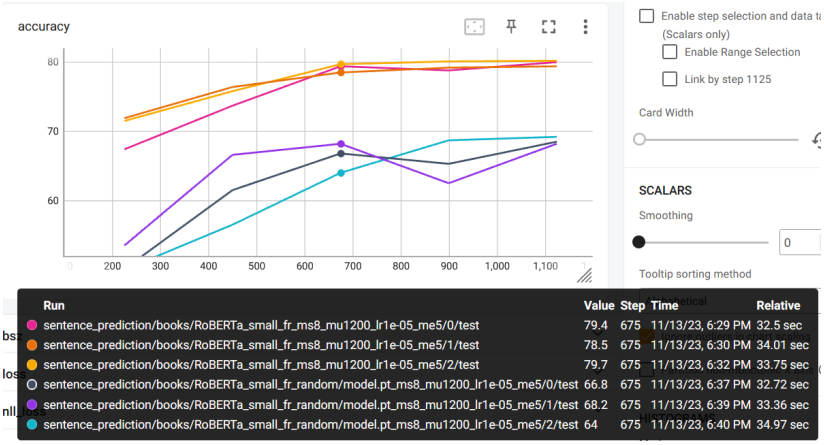

Figure 9: Tensorboard results of accuracy for the six tests of finetuning, no smoothness.

3.2 Analysis

We easily visualize that the when we finetune the model using fairseq with checkpoints based on CLS books and when the checkpoints are initialized randomly the accuracy is way lower. It can be explained. Indeed, the effectiveness of fine-tuning depends on the quality of the checkpoints used. When we use random checkpoints it may not always represent the best learning state of the model [4].

4 Tensorboard results - HFTransformers

When we finetune our model thanks to HF transformers we obtain an evaluation accuracy such as Fig 10 & an evaluation loss such as Fig 11.

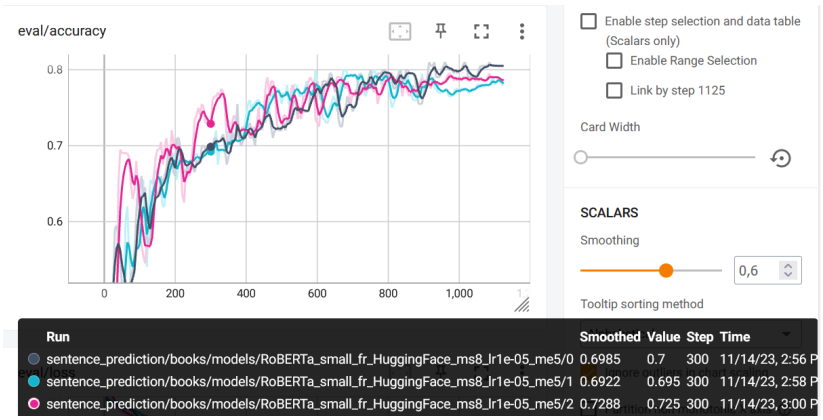

Figure 10: Tensorboard results of evaluation accuracy

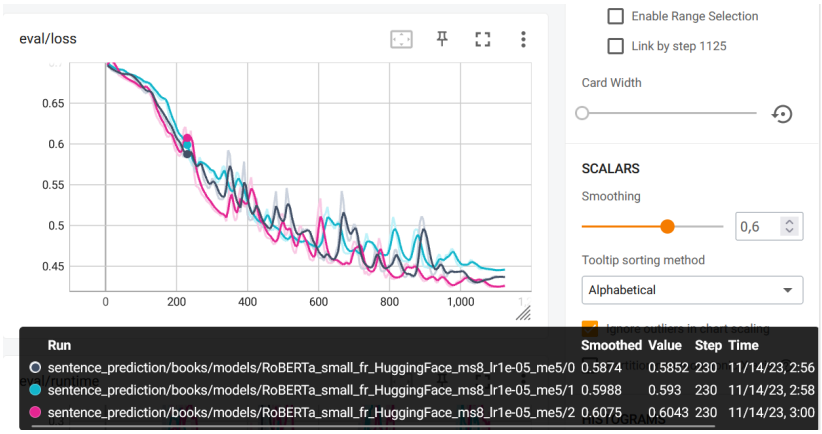

Figure 11: Tensorboard results of evaluation loss

Results achieved correspond to our expectations as the accuracy increase (up to 80 %) and the loss decreases (until 0.45). To obtained these curves, many efforts were made to understand wich parameters were needed in the computation. It was indeed needed to look deep into the GLUE repository and the Trainer part of the Transformers page of HuggingFace.

References

[1] Kourosh Hakhamaneshi Artur Niederfahrenhorst and Rehaan Ahmad. Fine-tuning llms: Lora or full-parameter? an in-depth analysis with llama 2. In *Anyscale*, 2023.

[2] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023.

[3] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.

[4] Yamini Jain. Checkpoints in deep learning. In *Indusmic*, 2021.

[5] Dmytro Nikolaiev. How to estimate the number of parameters in transformer models. In *Medium / Towards Data Science*, 2023.

[6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[7] Naman Goyal Jingfei Du Mandar Joshi Danqi Chen Omer Levy Mike Lewis Luke Zettlemoyer Yinhan Liu, Myle Ott and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. 2019.

[8] Artidoro Pagnoni Sylvain Gugger Sourab Mangrulkar Younes Belkada, Tim Dettmers. Making llms even more accessible with bitsandbytes, 4-bit quantization and qlora. In *Hugging Face*, 2023.