Training pre-trained language models using prefix and prompt tuning

Human Language Technologies Report

Diego Arcelli - 647979

Academic Year 2022-2023

UNIVERSITÀ DI PISA

Contents

1 Introduction

In this work I experimented with alternative techniques to fine-tuning, that can be used to train a pre-trained transformers on a downstream task. Specifically I tested two recently proposed techniques called prefix-tuning and prompt-tuning. The idea of these two methods is to keep the parameters of the pre-trained transformer fixed and train some additional prompts which, in the case of prompt-tuning, are added to the embedding of the input text. In prefix-tuning this process is repeated for all the self-attention layers of the transformer architecture. To compare the performances of these different training techniques I selected four downstream tasks on which to apply the three techniques and compare their results. These four tasks are: text summarization, table-to-text generation, sentences similarity prediction and paraphrase detection. More details about these tasks will be given in the next sections.

2 Background

2.1 Language modeling

A language model is a probabilistic model of natural language, which can be used to compute the probability distribution of a sequence of words based on a text corpora. The idea is to have a model which is able to compute the probability $P(w_1, \ldots, w_n)$, which is the probability of a sequence of n words, or $P(w_n|w_1 \ldots, w_{n-1})$, which is the probability of a word given the previous ones. In the past, in order to compute this kind of probability distributions, pure statistical models were used, such as the N-gram language model. These kind of pure statistical models have been surpassed by neural models, which use neural networks in order to compute the probability distribution. In particular, since a sentence can be modeled as a sequence of words taken from a vocabulary, recurrent neural networks (RNNs) are used. A RNN is a particular type of neural network which can handle indefinitely long input sequences, by maintaining an internal state which is updated whenever an element of the sequence is fed as input to the network. So if we have an input sequence $x_1, \ldots, x_L$, when we consider the word x_t we compute its embedding $e_t = Ex_t$ using an embedding matrix E, which is fed to recurrent neural network to update the internal state $h_t = f(We_t + Uh_{t-1})$ where f is an activation function and W and U are two trainable parameters matrices. The internal hidden states is used to compute $P(x_t|x_1, \ldots, x_{t-1}) = y_t = \text{Softmax}(Vh_t)$ where V is another trainable matrix. Given a big enough text corpora, a model like this one can be trained using back-propagation through time (BPTT).

2.2 Attention and Transformers

The main limitation of RNNs is the inability of handling long term dependencies, because of the vanishing gradient problem, which basically makes the model unable to learn when the input sequences are too long. In order to solve this problem in [1] the authors uses the self-attention mechanism. The idea is to compute the representation of one of the element of the input sequence as a weighed combination of all the other elements. We consider again a sequence $x_1, \ldots, x_L$. For each element of the sequence x_l we compute the query, key and value representations: $q_l = W^Q x_l$, $k_l = W^K x_l$ and $v_l = W^V x_l$, where W^K, W^Q and W^V are three trainable matrices. Then we focus on one specific query representation q_l and we compute its similarity with respect to all the key representations $s_i = q_l^T k_i$. Then we compute the attention coefficients $\alpha_i = \frac{\exp(s_i)}{\sum_{j=1}^{L} \exp(s_j)}$ which we then use to compute the weighted combination $h_i = \sum_{j=1}^{L} \alpha_j v_j$. We then repeat the same operation using all the other elements as query the representations of all the elements of the sequence. We can express the same computation in a more compact form, organizing the elements of the sequence as rows of a matrix $X \in \mathbb{R}^{L \times n}$, where n is the dimension of each element and L is the sequence length. Then we compute the query, key and value representations as $Q = W^Q X$, $K = W^K X$ and $V = W^V X$, and then compute the self-attention as:

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

The attention mechanism has been used to define the transformer architecture, which is basically a network of stacked attention layers. In the original definition in [1] it is and encoder-decoder network composed of multi-headed attention layers defined as shown in figure 1.

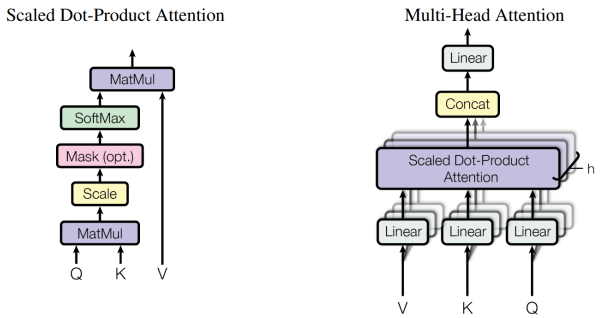

Figure 1: Multi-headed attention layer

The image on the left represent the computation of self-attention as explained before, while the image on the right shows the behavior of a multi-headed attention layer (MHA), which is defined as follows:

$$\text{MHA}(Q, K, V) = \text{Concatenation}(\text{head}_1, \ldots, \text{head}_n)W^O$$

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Basically attention is computed using n different query, key and value representations of the input sequence, and the results are concatenated. W_i^Q, W_i^K and W_i^V are learnable parameters matrices which are used to compute the different representations of the input query, key and value matrices. W^O is another learnable projection matrix which is applied to the concatenated outputs of the multi-headed attention. The full transformer architecture can be summarized in this figure:

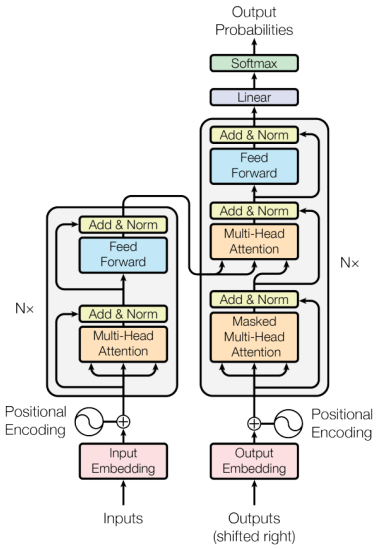

Figure 2: Transformer architecture

The encoder is composed by a stack of N blocks. In every block we compute the multi-headed attention as explained before, the output of the multi-headed attention is normalized using batch normalization and it is passed to a feed forward layer. The decoder works in an auto-regressive way, so it takes in input the output of the encoder and the previously generated tokens to produce the next token. The multi-headed attention layer of the decoder works in a sightly different way, since, in an auto-regressive scenario, we don't want to compute the attention score of a word using future words in the same sentence. Therefore, while computing attention for a given element of the sequence, the attention scores of the future element in the sequence are masked. This can be easily achieved in practice by setting to negative infinity the elements above the diagonal of the attention matrix, as shown in figure 3, where the matrix on the left is the attention matrix obtained computing $\frac{QK^T}{\sqrt{d_k}}$ in the attention equation.

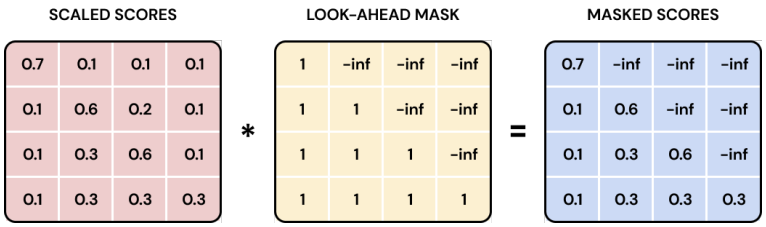

Figure 3: Masking attention matrix

Negative infinity is used as masking value so that, after the computation of the softmax, the negative infinities become zeroes.
Another relevant aspect is that the computation of attention doesn't depend by order of the elements in the sequence, if we shuffle the elements, the output of the computation is always the same. But in natural language the order of the words in a sentence is relevant. To solve this issue before feeding the input sequence to the transformer we add a positional encoding which is computed as follows:

$$PE_{(pos,2i)} = \sin\Big(pos/1000^{2i/d_{model}}\Big)$$

$$PE_{(pos,2i+1)} = \cos\Big(pos/1000^{2i/d_{model}}\Big)$$

where pos is the position of the element in the sequence and i is the dimension. This positional embedding matrix is summed in an element-wise way to the embedding of the input sequence.
After the definition of this transformer architecture many different variant where proposed, some of them maintained only the encoder (e.g. BERT) or only the decoder (e.g. GPT) of the original transformer architecture.

2.3 BERT

The Bidirectional Encoder Representations from Transformers (BERT) is an encoder-only transformer architecture introduced by Google in 2018. There are two different versions of the model: the base version is composed of 12 transformer blocks and it has a embedding dimension of 768 (110 millions parameters), while the large version is composed of 24 transformer block and it has an embedding dimension of 1024 (340 millions parameters). Both version are trained on a large dataset of 3.3 billions words, where 2.2 billions come from Wikipedia and the remaining ones come from Google's Books. The model is trained simultaneously on two tasks: masked language modeling and next sentence prediction. Masked language modeling consists of masking some words in a sentence and using the model to predict the masked words. In particular during BERT training, after the tokenization of the sentence, each token is substituted with 15% of probability with the special token `[MASK]`, and then the tokenized text is passed as input to the model which has to predict which was the masked tokens. In order to do so, a dense layer which has an output neuron for each possible token of BERT's vocabulary is used to predict the

probability distribution over the vocabulary of the masked tokens. Next sentence prediction consists of predicting whether a sentence is the subsequent of another one. In order to do this with BERT we use two special tokens: the `[SEP]` token is placed between the two tokenized sentences and the `[CLS]` tokens which is placed at the beginning of the first tokenized sentence. The `[SEP]` is used to mark the end of the first sentence and the beginning of the second one, while the `[CLS]` token is used to represent the full input sequence, since its embedding is used as input of the binary classification model which predicts if the second sentence is the subsequent of the first one.
After this pre-training phase on these two tasks, the model can fine-tuned on specific downstream-tasks by adding a new classification head on top of BERT and train all the parameters using the dataset of the new task. For instance if we want to fine-tune the model for a sentiment analysis task we can use the embedding of the `[CLS]` token as input of a binary classifier.

2.4 T5

Text-to-Text Transfer Transformer (T5) is an encoder-decoder architecture like the original transformer architecture, introduced by Google in 2020. The model is pre-trained on a variety of tasks, using a dataset extracted from Common Crawl and other minor datasets. The main task on which T5 is pre-trained is similar to the masked language modeling of BERT: some parts of the input sentences are masked and the model is trained to generate the missing parts. An example can be seen in figure 4.

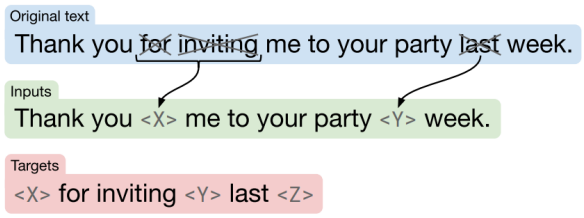

Figure 4: T5 unsupervised pre-training task

A major difference with respect to BERT is that while pre-training the model on this task, the model is also trained on a variety of different supervised tasks, such as tasks from the GLUE benchmark. The peculiarity of T5 is that all the tasks are converted in a text-to-text format. Basically a task specific prefix, which explains what the models has to do, is added at the beginning of the input sentences of that that task. For instance in a translation task from English to Italian we can use the prefix "translate English to Italian: ". Also the the output of the model is always a generated sequence of tokens, for instance for a sentiment analysis task instead of using a classification head to predict one of two classes we can make the model generate the string "positive" for

positive samples and ”negative” for negative samples. We can see some other examples in figure 5.

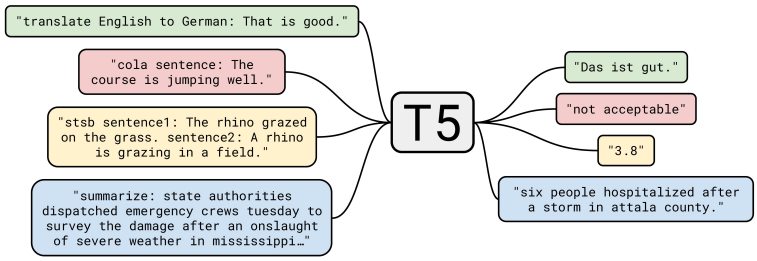

Figure 5: Examples of how to convert tasks to text-to-text format

As for BERT, there are many variants of the model of different sizes. For the experiment we will use the small version which is composed of 6 transformer block (both for the encoder and the decoder), it has an embedding dimension of 512 and the number of trainable parameters is 60 millions.

2.5 Metrics

There are several metrics which can be used for the automatic evaluation of a text generation task (e.g. text summarization, machine translation), where we want to to measure how good the model is behaving by comparing the generated text with one or more text references. One for these metric is the Recall-Oriented Understudy for Gisting Evaluation (ROUGE). ROUGE works using n-grams of the predicted and reference texts, in particular the ROUGE-n-precision computes the proportion of n-grams of shared by the two texts:

$$\text{ROUGE-}n\text{-recall} = \frac{\text{Number of shared } n\text{-grams}}{\text{Number of reference } n\text{-grams}}$$

$$\text{ROUGE-}n\text{-precision} = \frac{\text{Number of shared } n\text{-grams}}{\text{Number of predicted } n\text{-grams}}$$

$$\text{ROUGE-}n\text{-F1} = 2 \cdot \frac{\text{ROUGE-}n\text{-precision} \cdot \text{ROUGE-}n\text{-recall}}{\text{ROUGE-}n\text{-precision} + \text{ROUGE-}n\text{-recall}}$$

So if we use ROUGE-1 we work at level of words (uni-grams), with ROUGE-2 we work using bi-grams and so on. There is also a variant called ROUGE-L where, during the computation of the precision and the recall, in the numerator instead of using the number of overlapping n-grams we use we use the length of the longest common sub-sequence between the reference and the predicted texts, while in the denominator we replace the number of n-grams with the length of the text.
Another widely used metrics is the Bilingual Evaluation Understudy (BLEU) score. Like ROUGE, BLEU compares n-grams of the generated text with one or more reference texts, in particular it focuses on a modified version of the

n-gram precision. Consider for example the reference sentence "the cat is on the table" and the predicted sentence "the the the the". Using the normal uni-gram precision is $4/4 = 1$, since all the words of the generated text appears in the reference text, even thou they're the same word. To solve this issue in the modified precision computation each word gets credit for the maximum number of times it appears on the reference text. So, in the before example, since the word "the" appears only 2 times in the reference text the modified precision is $2/4 = 0.5$. The example was made in the case of uni-grams but the same reasoning can be generalized in the case of n-grams:

$$p_n = \frac{\sum_{n\text{-gram}\in\hat{y}} \text{Count}_{\text{clip}}(n\text{-gram})}{\sum_{n\text{-gram}\in\hat{y}} \text{Count}(n\text{-gram})}$$

where $\hat{y}$ is the set of n-grams contained in the generated text, Count(n-gram) count how many times n-gram appears in the generated text and $\text{Count}_{\text{clip}}(n\text{-gram})$ is the function that counts how many times time the n-gram appears in the generated text, clipping the count to the number of times that n-gram appears in the reference. The BLEU score is computed as the geometric average of the modified 1-gram, 2-gram, 3-gram and 4-gram precision:

$$\text{BLEU} = \sqrt[4]{p_1 \cdot p_2 \cdot p_3 \cdot p_4}$$

2.6 Prefix-Tuning

In [2] an alternative to fine-tuning called prefix-tuning has been proposed. While preforming fine-tuning to adapt the model for a downstream task, we train all the parameters of the network. So if we have N downstream tasks we'll end up with N different models. The idea of prefix-tuning is to instead keep the parameters of the pre-trained network frozen, and train a set of prompts which are concatenated as prefixes to the input of each transformer's layer. So formally speaking, suppose we have a transformer architecture of n layers $h_1, h_2, \ldots, h_n$ where each $h_i \in \mathbb{R}^{L\times D}$, D is the embedding dimension and L is the sequence length. The idea of prefix tuning is to attach to each h_i a trainable prompt $p_i \in \mathbb{R}^{L_p\times D}$ where L_p is the sequence length of the prompt and use as input of h_{i+1} the concatenated tensor $[p_i, h_i] \in \mathbb{R}^{L_p+L\times D}$. At training time we freeze all the parameters of the pre-trained transformer and we only train every p_i and other task specific parameter, for instance if we are in a classification task we also train the parameters of the classification head. Actually, since the concatenation of the prompts would cause the sequence length dimension to increase, ending up with a sequence length of $L + n \cdot L_p$, the concatenation mechanism works in a slightly different way. Consider the normal attention mechanism explained before:

$$\text{MHA}(Q, K, V) = \text{Concatenation}(\text{head}_1, \ldots, \text{head}_n)W^O$$

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

The straightforward implementation of prefix tuning would add the prompt p as follows:

$$\text{MHA}([p, Q], [p, K], [p, V])$$

but in this case the sequence length would increase as explained. To prevent this the prompt $p \in \mathbb{R}^{L_p \times D}$ is split along the sequence length dimension in $p = [p_K, p_V]$ where both p_K and p_V are in $\mathbb{R}^{L_p/2 \times D}$ and we compute the multi-headed attention layer as:

$$\text{MHA}(Q, [p_K, K], [p_V, V])$$

In this way the output of the multi-headed attention is still in $\mathbb{R}^{L \times D}$.

2.7 Prompt-Tuning

In [3] the authors further simplified the idea of prefix-tuning, proposing a new technique called prompt-tuning which applies the mechanism of prefix-tuning only to the first layer of the transformer architecture. In this way the number of trainable parameters is reduced even more. In this case we can see the transformer as a function $f = f_e \circ f_r$ where f_e is the input embedding layer and f_r is the stack of multi-headed attention layers. So f_e receives as input a sequence of tokens x of length L and produces as output $x_e = f_e(x) \in \mathbb{R}^{L \times D}$ where D is the embedding dimension. Then x_e is passed to f_r to compute the output of the transformer. In prompt-tuning before passing x_e to f_r, we concatenate to it the trainable prompt $p \in \mathbb{R}^{L_p \times D}$, hence f_r receives as input $[p, x_e] \in \mathbb{R}^{L+L_p \times D}$. Like in the case of prefix-tuning we freeze all the parameters of the pre-trained transformer and we only train p and the optional classification head if needed by the task. Unlike prefix-tuning, prompt-tuning doesn't adjust the attention computation to maintain the original sequence length, so the output of the transformers will have the sequence length dimension of size $L + L_p$.

3 Experiments

In order to compare traditional fine-tuning with prefix and prompt tuning I considered four downstream task:

- Predict whether two questions have the same meaning using the Quora Question Pairs2 dataset
- Computing the similarity between two sentences using the Semantic Textual Similarity Benchmark dataset
- Text summarization using the BillSum dataset
- Table to text using the E2E dataset

For each dataset we considered a suitable transformer model to specialize on the task and we train it using fine-tuning, prefix-tuning and prompt-tuning to compare the results. Specific details for each task will be given in the following section.
All the models have been trained with stochastic gradient descent, using Adam as optimizer with parameters $\beta_1 = 0.999$ and $\beta_2 = 0.9$ and also with a linear learning rate decay. We also use L2 regularization using a regularization coefficient of 0.1. The experiments have been performed on a Tesla T4 and a Tesla P100 GPUs.
For the implementation of the prompting techniques I used the Parameter Efficient Fine-Tuning (PEFT) library [1], which integrates prompt and prefix tuning techniques in the models available in the `Transformers` library. The code of the experiments is available in this Github repository: `https://github.com/DiegoArcelli/Prompt-Tuning-NML`.
Note that in the results FT stands for fine-tuning, Pre-T stands for prefix-tuning and Pro-T stands for prompt-tuning.

3.1 Quora Question Pairs2

The Quora Question Pairs2 (QQP2) dataset contains pairs of questions taken from the Quora website and the goal is to predict whether the two questions have the same meaning or not, hence it can be set as a binary classification task. The dataset already provide the train, validation and test split of the following sizes:

- Train set: 291.200
- Validation set: 74.000
- Test set: 72.800

In all train and validation splits the percentage of samples of class 0 is the 63%, while the percentage of samples of class 1 is the 37%. The labels of the test set are not publicly available so the class balancing cannot be established. The metrics associated to the task are the accuracy and the F1-score, which is the harmonic average between the precision and the recall. The model chosen for the task is the uncased BERT base, on which we attach a linear classification head that uses the embedding of the `[CLS]` token to classify the input, while the two tokenized sentences are separated by the `[SEP]` token. The classification layer is a linear layer The loss used for the task is the binary cross-entropy loss function:

$$\text{Loss}(y, \hat{y}) = -\frac{1}{n}\sum_{i=0}^{n-1}\Big(y \cdot \log(\hat{y}_{0,i}) + (1-y)\cdot\log(\hat{y}_{1,i})\Big)$$

[1] https://github.com/huggingface/peft

where n is the number of input sentences in the current batch, $y_i \in \{0, 1\}$ is the label of the i-th sample, $\hat{y}_{0,i}$ and $\hat{y}_{1,i}$ are the predicted probabilities for the i-th input sample to be respectively of class 0 and 1. The hyperparameters used in the three training strategies are reported in table 1. In the table we also show how many model's parameter are trained in each different training strategies.

	Epochs	**Learning rate**	**Batch size**	**Parameters**	**Trainable parameters**
FT	3	0.00002	16	109.483.778	109.483.778 (100%)
Pre-T	4	0,02	16	109.852.418	370.178 (0,337%)
Pro-T	5	0,02	16	109.499.138	16.898 (0,015%)

Table 1: Hyperparameters and trainable parameters for QQP2 dataset

The number of parameters associated to the classification head is just 1538, since we use a 768×2 linear layer for the classification. The performances obtained with the three techniques are reported in table 2.

	Train	**Validation**			**Test**	
	Loss	**Loss**	**Acc.**	**F1**	**Acc.**	**F1**
FT	0,216	0,364	90,7	87,4	70,6	89,0
Pre-T	0,422	0,323	85,5	80,5	65,0	85,5
Pro-T	0,529	0.434	78,7	74,8	55,3	76,1

Table 2: Results for QQP2 dataset

In all the three cases we notice that there is a big change between the validation and the test set for the accuracy, which decreases of around the 20%, while the F1-score is more or less the same. The best performing model is the one trained with fine-tuning, while the model trained with prefix-tuning achieves lower performances in particular the test accuracy decreases of the 5,6% and the test F1 metric decreases of the 3,5%. With prompt-tuning the distance in performances is even higher, with a drop in the test accuracy of the 14,7% and the difference in the test F1 metric of the 12,9%. So in this case the model trained with fine tuning outperforms the models trained with prompt-tuning and prefix-tuning.

3.2 Semantic Textual Similarity Benchmark

The Semantic Textual Similarity Benchmark (STSB) dataset contains pairs of sentences which are annotated with a score form 1 to 5 which measures the semantic similarity between the two sentences, so the highest is the score the more the two sentences have the same meaning. It can be set as a regression

task using the MSE as training loss:

$$\text{Loss}(y, \hat{y}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \hat{y}_i)^2$$

where n is the number of inputs sentences in the current batch, y_i is the label of the i-th input sample and $\hat{y}_i$ is the score predicted by the model for the i-th sample.
The metrics associated with the task are the Pearson and the Spearman correlation indexes. Like for the QQP2 dataset, even in this case the available test set is not labeled, hence again we randomly select the 20% of the train set to use as a test set. The final sizes of the splits are:

- Train set: 4.600
- Validation set: 1.500
- Test set: 1.150

Again we tokenize the two sentences separating them with the `[SEP]` token and we use the embedding of the `[CLS]` token as input of a linear classifier that has to predict the similarity score. In table 3 we report the chosen hyperparameters and the number of trainable parameters and in table 4 we report the results obtained. We again use the uncased BERT base model and this case the number of parameters associated to the classifier is 769, since we used a 768×1 linear layer.

	Epochs	**Learning rate**	**Batch size**	**Parameters**	**Trainable parameters**
FT	5	0,00002	16	109.484.547	109.484.547 (100%)
Pre-T	8	0,02	16	109.853.187	369.409 (0,336%)
Pro-T	20	0,02	16	109.499.907	16.129 (0.015%)

Table 3: Hyperparameters and trainable parameters for STSB dataset

	Train	**Validation**			**Test**	
	Loss	**Loss**	**Pearson**	**Spearman**	**Pearson**	**Spearman**
FT	0,443	0,466	0,894	0,892	0,852	0,839
Pre-T	0,881	0,583	0,862	0.860	0,824	0,813
Pro-T	1,346	0,828	0,823	0,824	0,760	0,739

Table 4: Results for STSB dataset

In this case the performance differences that we obtain between the validation and the test set are not so huge like in the previous case. Even in this

case the model trained with fine-tuning is the one that reaches the best performances. The model trained with prefix-tuning is still worse than the one trained with fine-tuning, since the Pearson correlation metric is lower than the 0,028 for the test results, while the Spearman correlation is lower than the 0,028. For prompt-tuning the difference in the test set is 0,092 (Pearson) and 0,1 (Spearman).

3.3 BillSum

The BillSum dataset contains text of bills and human-written summaries from the US Congress and California Legislature. Each record contains the text of the bill, a summary of the text and a title of the text. For the purposes of the task we won't consider the title, and we'll only keep the text and the summary columns. The goal of course is to train a model which is able to receive the text as input an produce a summary. The model used is the T5 small, adding the input text the task prefix "summarize: ". The dataset provides a train and a test set, but not a validation set, so in this case we randomly select the 20% of the train set to form a validation set. The size of the three splits are:

- Train set: 15.160
- Validation set: 3.789
- Test set: 3.269

The metric used for the evaluation is the ROUGE metric. The loss used for the training is the cross-entropy:

$$\text{Loss}(y, \hat{y}) = -\frac{1}{n} \sum_{i=0}^{n-1} \sum_{l=0}^{L-1} \sum_{c=0}^{V-1} \mathbb{I}[y_{l,i} = c] \cdot \log(\hat{y}_{c,l,i})$$

where V is the vocabulary size, n is the number of elements in the batch, L is the input sequence length, $\hat{y}_{c,l,i}$ is the probability of the l-th sequence element of the i-th input sequence to be token c of the vocabulary and $y_{l,i}$ is the target token for the l-th sequence element of the i-th input sequence. Basically for each token in the target sequence, the cross-entropy loss is calculated between the predicted probability distribution for that token and the actual target token. The individual token-level losses are summed to compute the overall loss for the entire sequence, and then the sequence-level losses are averaged over all the sequences in the batch to compute the current batch loss.
Again we report in table 5 the information about hyperparameters and trainable parameters, and in table 6 we report the results.

	Epochs	Learning rate	Batch size	Parameters	Trainable parameters
FT	2	0,00002	4	60.506.624	60.506.624 (100%)
Pre-T	4	0,02	4	60.629.504	122.880 (0,203%)
Pro-T	8	0,02	4	60.527.104	20.480 (0,034%)

Table 5: Hyperparameters and trainable parameters for STSB dataset

	Train	Validation			
	Loss	Loss	ROUGE-1	ROUGE-2	ROUGE-L
FT	2,109	2,148	16,3	12,8	15,6
Pre-T	2,111	2,140	16,8	13,3	16,2
Pro-T	2,483	2,467	15,8	12,2	15,0

	Test			
	Loss	ROUGE-1	ROUGE-2	ROUGE-L
FT	2,083	16,0	12,6	15,4
Pre-T	2,094	16,7	13,3	16,1
Pro-T	2,439	15,8	12,2	15,0

Table 6: Results for BillSum dataset

Focusing again on the test results, we can see that in this case the results are much better. Prefix-tuning obtains better results than fine-tuning for all the three metrics. Prompt-tuning is still a bit worse than fine-tuning but with respect to the previous tasks the difference now much smaller.

3.4 E2E

The E2E dataset contains information about restaurants codified in a tabular form, and each table is associated with a natural language sentence which describes the table. The information contained in the tabular representation are reported in table 7 and some examples of records contained in the dataset are reported in table 8.

Attribute	Data type	Example value
name	verbatim string	The Egale, ...
eatType	dictionary	restaurant, pub, ...
familyFriendly	boolean	Yes / No
priceRange	dictionary	cheap, expensive, ...
food	dictionary	French, Italian, ...
near	verbatim string	market square, ...
area	dictionary	riverside, city center, ...
customerRating	enumerable	1 of 5 (low), 4 of 5 (high), ...

Table 7: Information of the restaurants in the E2E dataset

Table representation	Text representation
name[The Vaults], eatType[pub], priceRange[more than £30], customer rating[5 out of 5], near[Café Adriatic]	The Vaults pub near Café Adriatic has a 5 star rating. Prices start at £30.
name[Loch Fyne], food[French], customer rating[high], area[riverside], near[The Rice Boat]	For luxurious French food, the Loch Fyne is located by the river next to The Rice Boat.
name[Aromi], eatType[coffee shop], food[French], customer rating[low], area[city centre], familyFriendly[no]	In the city centre lies Aromi, a French coffee shop for adults with a low customer rating.

Table 8: Examples of records of the E2E dataset

The goal is to train a model which is able to receive as input a string representing the tabular representation and generates the textual representation. The dataset provides all the three splits:

- Train set: 42.100 examples
- Validation set: 4.670 examples
- Test set: 4.690 examples

To perform this task we use again the T5 small model. The model receives as input a string representing the tabular representation, so a string of the form. "attribute1[value1], attribute2[value2], ...", and to respect the text-to-text format of T5 we also add as prefix the task prompt "summarize table: ". For this task the selected metrics are the ROUGE and the BLEU scores. The loss is the same we used for the BillSum dataset.

	Epochs	Learning rate	Batch size	Parameters	Trainable parameters
FT	2	0,00002	8	60.506.624	60.506.624 (100%)
Pre-T	4	0,02	8	60.629.504	122.880 (0,203%)
Pro-T	8	0,02	8	60.527.104	20.480 (0,034%)

Table 9: Hyperparameters and trainable parameters for E2E dataset

	Train	Validation				
	Loss	Loss	ROUGE-1	ROUGE-2	ROUGE-L	BLEU
FT	1,262	1,186	56,4	32,5	44,6	16,6
Pre-T	1,331	1,258	55,9	33,2	44,8	17,1
Pro-T	1,454	1,354	57,3	32,6	44,8	16,1

	Test				
	Loss	ROUGE-1	ROUGE-2	ROUGE-L	BLEU
FT	1,221	56,1	31,4	43,1	14,5
Pre-T	1,286	57,5	33,2	44,2	15,6
Pro-T	1,402	59.7	34,1	45,6	15,3

Table 10: Results for E2E dataset

In this case not only the performances of prefix-tuning but also the performances of prompt-tuning are really close to the ones of fine-tuning, and for some of the metrics the model trained on prefix-tuning reaches even better values of some of the metrics.

4 Conclusions

The performed experiments show quite different results. In the first classification task we can see that fine-tuning outperforms both prompting techniques, and the same thing happens for the regression task. Instead on the two text generation tasks (table-to-text and summarization) the performances of prompting techniques are quite close and sometimes even better than the performances of the traditional fine-tuning technique.
So in general it seems that prefix and prompt tuning work better when used for text generation tasks, since the best results are obtained on the table-to-text and summarization tasks, especially in the summarization case as we saw the results of prompt tuning are even better than the results of fine-tuning. While in the classification and regression task, where we use and encoder-only transformer the results of fine-tuning are significantly better than the other two techniques.
One major limitation of my work is that I conducted the experiments testing

manually different values for the prompt length parameters. It could have been better to perform a grid search on this parameter.
In term of time required to train the model for a single training epoch, prompt and prefix tuning are a bit faster then fine-tuning. For instance, for the QQP2 dataset the time required to train the model for one epoch using fine-tuning was around 4 hours, while for prompt and prefix tuning was around 3 hours. But if we also take into account the fact that prompting techniques might requires more training epochs than fine-tuning (and it was the case for my experiment), then the time required to complete the training of the model can even be higher using prompting techniques than classical fine-tuning. One advantage of prompting techniques could be in term of memory occupation, since if we have the same pre-trained model which must be specialized for a certain number or downstream tasks, if we use fine-tuning we have to store the different fine-tuned version of the original model for each task, and we know that these model can be very large. Using prompting techniques instead we can store just the pre-trained model and different task specific prompts, which in term of number of parameters are just a very small fraction of the ones of the pre-trained model.

References

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[2] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190*, 2021.

[3] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021.