Advanced Methods in Text Analytics
Exercise 7: Large Language Models - Part 1
Solutions

Daniel Ruffinelli

FSS 2025

1 Transfer Learning

(a) A model is said to be *pre-trained* when the goal of the training process is to learn *general* representations of the input data, without a specific task in mind where these representations could be used. In that sense, the prefix "pre" indicates that the model is trained *before* being fine-tuned on a specific task. In the context of LLMs, fine-tuning is not done as frequently as in the era of pre-trained language models (PLMs), e.g. BERT. So we can speak of pre-training as the training of a model before it is actually used on a different set of (downstream) tasks.

(b) Pre-training usually involves training a model on a corpus of data using a *self-supervised* learning objective. The most common ones are: (i) causal language modeling (CLM) and (ii) masked language modeling (MLM).

(c) The causal language modeling (CLM) objective is defined as follows:

$$\text{CLM}(S) = \prod_{s=1}^{S} p(x_{|s|} \mid x_1, x_2, \ldots, x_{|s|-1}), \tag{1}$$

where x_i are tokens in sequence s. Similarly, the masked language modeling (MLM) objective is defined as follows:

$$\text{MLM}(S) = \prod_{s=1}^{S} \prod_{m=1}^{M_s} p(x_m \mid x_1, x_2, \ldots, x_{|s|} \setminus M_s), \tag{2}$$

where M_s is set of masked out tokens in sequence s and $A \setminus B$ is set difference. In practice, we use empirical risk minimization with log loss during training. That is,

$$\text{CLM}(S) = -\frac{1}{|S|} \sum_{s=1}^{S} \log p(x_{|s|} \mid x_1, x_2, \ldots, x_{|s|-1}), \tag{3}$$

$$\text{MLM}(S) = -\frac{1}{|M|} \sum_{s=1}^{S} \sum_{m=1}^{M_s} \log p(x_m \mid x_1, x_2, \ldots, x_{|s|} \setminus M_s), \tag{4}$$

where $M = \{M_1 \cup M_2 \cup \ldots \cup M_{|S|}\}$ is the set of all masked tokens in S.

(d) The main factors are: (i) size, (ii) quality, and (iii) domain. **Size.** The data should be large enough to capture a wide range of linguistic patterns and concepts, allowing the model to learn a rich set of representations for the language. It should also be large enough to avoid overfitting, easily done with a large number of parameters.

Quality. The quality is also important, but it can be measured in different ways. We can speak of quality in terms of grammatical correctness, or in terms of lexical variety, etc. What is good quality usually depends on the goal we have for the model. E.g. if the goal is to train a model for speaking formal English, then University textbooks are likely a high quality source. But if the goal in on colloquial English, then online forums are likely a better source.

Domain. Pre-training data should be a diverse collection of text from various sources that cover a set of domains of interest, e.g. medicine, biology, pop culture, etc.

(e) Fine-tuning refers to the process of taking a pre-trained model and adapting it to a specific downstream task by updating its parameters on a smaller labeled dataset that represents this task. Thus, fine-tuning is different from pre-training in that pre-training involves training a model without a specific *target* task in mind.

Fine-tuning typically relies on supervised data that represents the downtream task of interest, e.g. pairs of the form (review, sentiment) for sentiment analysis of product reviews. Thus, during fine-tuning, the model should learn task-specific patterns and improve its performance on the downstream task, at the potential cost of catastrophically forgetting what was learned during pre-training.

(f) The main challenge behind training models with a large number of parameters is the computational cost and memory requirements that come with the training process. To reduce runtime costs, specialized hardware is commonly used for training models, e.g. GPUs that are more efficient at computing common operations used during training, such as matrix products. In addition, large models can be memory-intensive, requiring large amounts of memory just to store the model parameters and intermediate activations during training (as computed soon in Task 3).

2 Parameter Efficient Fine-Tuning

(a) Parameter-efficient fine-tuning (PEFT) method aim to reduce the computational cost and memory requirements of fine-tuning large language models by updating only a subset of the parameters in the pre-trained model.

One common approch is the use of adapters, which are small parameterized components added to the pre-trained architecture, typically a transformer-based LM, which modify the model's forward pass. Then, during fine-tuning, only the parameters of the adapters are updated, while the parameters of the pre-trained model are kept *frozen*. Adapters can be placed anywhere in the model's architecture and thus have different impacts on the forward pass. In addition, adapters can themselves be somewhat involved, e.g. by including an attention mechanism.

Another approach to PEFT is **prompt tuning**, which adds a small number of task-specific tokens to the input sequence during fine-tuning. These tokens are used to guide the model towards the task of interest, e.g. by specifying the task in the promp given to the model, e.g. "summarize this:" During fine-tuning, only these special tokens are updated, giving

the model a small set of parameters that can be used to modify its outputs to better fit the downstream task of interest.

(b) We have:

$$\text{FNN}(\boldsymbol{X}) = f(\boldsymbol{X}\boldsymbol{W}_{up})\boldsymbol{W}_{down}, \tag{5}$$

where f is a non-linear activation function, originally ReLU, lately different ones like SwiGLU.

(c) Let $\boldsymbol{A} \in \mathbb{R}^{d\times r}$, $\boldsymbol{B} \in R^{r\times d}$ be the projection matrices in the adapter. Then:

$$\text{FNN'}(\boldsymbol{X}) = \text{FNN}(\boldsymbol{X}) + g(\text{FNN}(\boldsymbol{X})\boldsymbol{A})\boldsymbol{B}, \tag{6}$$

where g is a non-linear activation function (multiple ones were originally tested). That is, Houlsby adapters are applied sequentially after the operator being fine-tuned, and they include a residual connection as with the other components in the transformer block (hence the addition). Thus, Houslby adapters effectively add a new *sequential* component to each transformer block. Each adapter has $2dr$ number of parameters, where r is a hyperparameter, the bottleneck dimension used to control the size/capacity of the adapter. Typically, $r << d$ to keep the number of trainable parameters low.

(d) Houlsby adapters can be placed after any layer or sublayer in the model. In fact, Houlsby originally added one adapter after the multi-head attention operator and another after the FNN operator in each transformer block.

(e) Let $\boldsymbol{A} \in \mathbb{R}^{d\times r}$, $\boldsymbol{B} \in R^{r\times 4d}$ be the projection matrices in the LoRA adapter. Then:

$$\text{FNN'}(\boldsymbol{X}) = f(\boldsymbol{X}\boldsymbol{W}_{up} + \boldsymbol{X}\boldsymbol{A}\boldsymbol{B})\boldsymbol{W}_{down} \tag{7}$$

The addition comes from the LoRA adapter, which is applied using a residual connection around the weight matrix we are fine-tuning. $\boldsymbol{A}$ has dr parameters and $\boldsymbol{B}$ has $4dr$ parameters, for a total of $5dr$ for the entire adapter, where $r << d$ is the hyperparameter that controls the rank of both projection matrices used by LoRA adapters.

(f) LoRA adapters can be applied to any projection matrix, e.g. the projections used in self-attention layers. This should be clear from the fact that in the FNN operator, we applied a LoRA adapter only to one of the two possible projection matrices. In fact, in the original paper, LoRA adapters were applied to matrices $\boldsymbol{W}^Q$ and $\boldsymbol{W}^V$ in self-attention layers.

(g) This is easy to see from Eq. 7 and the fact that matrix products are distributive w.r.t. matrix sum. That is,

$$\text{FNN'}(\boldsymbol{X}) = f(\boldsymbol{X}\boldsymbol{W}_{up} + \boldsymbol{X}\boldsymbol{A}\boldsymbol{B})\boldsymbol{W}_{down} = f(\boldsymbol{X}(\boldsymbol{W}_{up} + \boldsymbol{A}\boldsymbol{B}))\boldsymbol{W}_{down} \tag{8}$$

where $\Delta\boldsymbol{W}_{up} = \boldsymbol{A}\boldsymbol{B}$.

The advantage of this formulation is that it allows us to see that the learned adapters can be added back into the original model after learning is done. This results in a fine-tuned model without an increase in number of parameters and without the corresponding additional runtime costs in the forward pass. In addition, it allows for modularity, as not only can we choose which matrices to specifically tune, but it allows us to remove the effect of the adapters from the model by simply subtracting $\boldsymbol{A}\boldsymbol{B}$ from the tuned weight matrix to recover the original model.

Such advantages are not immediately possible with Houlsby adapters. For one, they were designed to behave similarly to the sublayers in a transformer block, so it's not clear what impact they would have when applied to, e.g., matrix $\boldsymbol{W}^K$ in a self-attention layer. In addition, Houlsby adapters make changes to the residual space using linear projections that include a non-linear activation function in between, so if we want to remove the impact of these adapters to recover the original model, we would have to ensure the non-linear activation function has an inverse function and that the learned projections are invertible.

3 Transformer-Based Large Language Models

(a) Transformer-based language models require positional encodings to provide the model with information about the position of each token in the input sequence. This is because the transformer architecture does not have any built-in mechanism to understand the order of the tokens in the input sequence, unlike RNNs which process tokens sequentially, thus naturally having a "sense of time". The positional encodings are *typically* added to the input embeddings before feeding them into the first transformer layer. Thus, a language model with L transformer layers uses a single positional encoding layer, independently of the number of transformer layers in the model. However, some positional encoding layers are added to each transformer layer.

(b) The language model head in a transformer-based language model is responsible for producing the distribution used for predicting the next token given an input sequence. It is a softmax layer that projects the contextualized representations of the input tokens to the vocabulary space and applies the softmax function to normalize the logit scores into a probability distribution.

The weights of the language model head are typically tied to the input embeddings, because both the static embedding matrix and the projection matrix in the language model head are of the same size. This means that the weights of the language model head are the same as the weights of the input embeddings. This weight tying helps in reducing the number of parameters in the model and improves the generalization of the model.

(c) The output of a multi-head attention operator MHA that uses n self-attention heads SA_i is the concatenation of the outputs of each head. To ensure the size of the output is the same as the input, these concatenated outputs are projected to the same size as the input. That is,

$$\text{MHA} = (\text{SA}_1 \oplus \text{SA}_2 \oplus \ldots \oplus \text{SA}_n)\boldsymbol{W}^O, \tag{9}$$

where $\oplus$ denotes the concatenation operation. The parameters of the multi-head attention operator are the parameters of each attention head plus the output projection matrix $\boldsymbol{W}^O$. In other words,

$$\boldsymbol{\theta}_{\text{MHA}} = [\boldsymbol{\theta}_{\text{SA}_1}, \boldsymbol{\theta}_{\text{SA}_2}, \ldots, \boldsymbol{\theta}_{\text{SA}_n}, \boldsymbol{W}^O]. \tag{10}$$

(d) Let's start with what we know.

(i) The size of query, key and value vectors in each self-attention layer SA_i are each of size 128. Specifically, the size of the value vectors tells us that the output of each self-attention head is of size 128.

(ii) The size of the output representations of a MHA operator are the same size as the input representations, in this case $d_H = 12288$ (the hint).

(iii) The MHA operation is given by Eq. 9.

(iv) The projection inside the multi-head attention operator, i.e. $\boldsymbol{W}^O$, does not change the dimension of its inputs.

The first three points above tell us that $\boldsymbol{W}^O \in \mathbb{R}^{c \times 12288}$, where c is the size of the concatenations of the outputs of each self-attention head, i.e. $c = n \cdot 128$, where n is the number of attention heads we are looking for. But the fourth point above tells us that $\boldsymbol{W}^O$ is a square matrix, which means $c = 12288$. Thus, we have:

$$n = \frac{12288}{128} = 96. \tag{11}$$

The largest variant of GPT-3 has 96 attention heads in each multi-head attention operator, and the projection inside the multi-head attention indeed does not change the dimension of the concatenated outputs of each attention head. This means that this projection could be easily dropped from each multi-head attention layer without affecting the forward pass in this case. But this projection is nevertheless still used, likely because the large number of parameters is useful for the model.

(e) The components that such a CLM has are:

(i) a static embedding matrix followed by

(ii) a positional embeddings layer followed by

(iii) a stack of L transformer layers followed by

(iv) a language modeling head lm_head applied after the last transformer layer.

Let's count the number of parameters in each component, ignoring all biases and layer normalization operators, and knowing positional embeddings have no parameters.

First, lm_head is parameterized by $\boldsymbol{\theta}_{lm\_head} = \boldsymbol{W}_{lm\_head} \in \mathbb{R}^{d_H \times V}$. Given that we have weight tying, this matrix will also serve as our static embedding matrix. Second, each transformer layer is a multi-head attention layer MHA followed by an FNN layer, which projects the representations to some m-dimensional space, and then back down to the original input dimension (see Exercise 6 Task 1.b). So, the total number of parameters in such an architecture would be:

$$|\boldsymbol{\theta}_{CLM}| = \sum_{i=1}^{L} (|\boldsymbol{\theta}_{\text{MHA}_i}| + |\boldsymbol{\theta}_{\text{FNN}_i}|) + |\boldsymbol{\theta}_{lm\_head}|. \tag{12}$$

where $\boldsymbol{\theta}_{\text{MHA}_i}$ and $\boldsymbol{\theta}_{\text{FNN}_i}$ are the parameters in the multi-head attention and FNN components of the i-th transformer layer, respectively.

Let's now compute the number of parameters in each component.

- $\boldsymbol{\theta}_{\text{SA}} = [\boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^V]$, each of size 12288×128, so that $|\boldsymbol{\theta}_{\text{SA}_i}| = 3 \cdot 12288 \cdot 128 = 4718592 \approx 4.7M$.
- $|\boldsymbol{\theta}_{\text{MHA}_i}| = (96 \times |\boldsymbol{\theta}_{\text{SA}}|) + |\boldsymbol{W}^O|$, and since $|\boldsymbol{W}^O| = 12288^2$, we have that $|\boldsymbol{\theta}_{\text{MHA}_i}| = (96 \times 4718592) + 12288^2 = 452984832 + 150994944 = 603979776 \approx 604M$.
- $|\boldsymbol{\theta}_{\text{FNN}_i}| = 2 \times (12288 \times (12288 \times 4)) = 12288^2 \cdot 8 = 1207959552 \approx 1.2B$.
- $|\boldsymbol{\theta}_{lm\_head}| = d_H \times V = 12288 \times 50000 = 614400000 \approx 614M$.

All together, we have:

$$|\boldsymbol{\theta}_{CLM}| = 96 \cdot (604M + 1.2B) + 614M = 174550155264 \approx 175B. \tag{13}$$

Indeed, the largest variant of GPT-3 has around 175 billion parameters. The projection by the FNN operator does indeed project up to a space four times the size in the original transformers, and has remained common practice ever since, e.g. in the GPT-3 architecture.

Of course, most of the parameters come from the stack of transformer layers, but note that two thirds of those parameters come from the FNN operators in the transformer layers, as each of these takes has twice the parameters of a each multi-head attention operator. Now, assuming bfloat16, which is a 16-bit floating-point format commonly used in LLMs, we would need around 350GB of memory just to store this model. To train it, you need to further store the gradients of each of those parameters during the backward pass. This is why it's a massive engineering effort is needed so that these models can be trained in a distributed manner over thousands of GPUs.