Advanced Methods in Text Analytics

Exercise 7: Large Language Models - Part 1

Daniel Ruffinelli

University of Mannheim

FSS 2025

Transfer Learning

Question a)

- What do we mean when we say that a model is "pre-trained"?
- Why don't we say the model is simply "trained"?
- What type of data do we need for pre-training a language model?

Transfer Learning

Answer 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 downstream task in mind.
- 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.

Transfer Learning

Question b)

- What are the two more common training objectives for pre-training large language models?

Transfer Learning

Answer b)

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

Transfer Learning

Question c)

- Let S be the set of sequences used for training language model p.
- Using p and S, give formal definitions for the training objectives mentioned in your previous answer.

Transfer Learning

Answer c) (1)

- The 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 MLM objective is defined as follows:

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

  where M_s is set of masked out tokens in sequence s and $A \setminus B$ is set difference.

Transfer Learning

Answer c) (2)

- In practice, we use empirical risk minimization with log loss during training.
- That is,

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

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

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

Transfer Learning

Question d)

- What type of data should we use for pre-training an LLM?
- What factors should we consider when choosing what data to train on?

Transfer Learning

Answer d)

- The main factors are: (i) size, (ii) quality, and (iii) domain.
- **Size.** The data should be large enough to capture wide range of linguistic patterns/concepts, allowing model to learn rich set of representations.
- It should also be large enough to avoid overfitting, easily done with a large number of parameters.
- **Quality**. Also important, but it can be measured in different ways. E.g. grammatical correctness, lexical variety, etc. What is good quality usually depends on goal with model.
- E.g. if goal is to train a model for speaking formal English, then University textbooks are likely a high quality source.
- But if 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.

Transfer Learning

Question e)

- We typically talk of fine-tuning a pre-trained model.
- What do we mean by fine-tuning?
- How is it different from "pre-training" a model?
- What type of data do we need for fine-tuning a language model?

Transfer Learning

Answer e)

- Fine-tuning refers to process of adapting a pre-trained model 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 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.

Transfer Learning

Question f)

- What is the main challenge behind traning models with a large number of parameters?

Transfer Learning

Answer 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).

Parameter Efficient Fine-Tuning

Context

- Given the large number of parameters in LLMs, a set of methods have focused on fine-tuning large-size models by tuning only a subset of their parameters.
- In this task, we go over some of the concepts behind such parameter efficient fine-tuning (PEFT) methods.

Parameter Efficient Fine-Tuning

Question a)

- What is the main idea behind parameter-efficient fine-tuning (PEFT)?
- Briefly describe two different PEFT methods.

Parameter Efficient Fine-Tuning

Answer a) (1)

- 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.

Parameter Efficient Fine-Tuning

Answer a) (2)

- 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.

Parameter Efficient Fine-Tuning

Question b)

- Let $\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ be the input sequence and $\boldsymbol{H} = \{\boldsymbol{h}_1, \ldots, \boldsymbol{h}_n\}$ the corresponding output sequence of the FNN operator in a transformer block, where $\text{FNN}(\boldsymbol{X})$ is parameterized by $\boldsymbol{W}_{up} \in \mathbb{R}^{d \times 4d}$ and $\boldsymbol{W}_{down} \in \mathbb{R}^{4d \times d}$, i.e. the up and down projections in FNN, respectively.
- In other words, $\boldsymbol{H} = \text{FNN}(\boldsymbol{X} \mid \boldsymbol{W}_{up}, \boldsymbol{W}_{down})$ (note that we omit biases and the residual connection for simplicity).
- Give a formal expression for the output of this FNN operator.

Parameter Efficient Fine-Tuning

Answer 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.

Parameter Efficient Fine-Tuning

Question c)

- Following the previous subtask, assume we fine-tune the FNN operator using the Houlsby adapters introduced in the lectures, and call this fine-tuned operator FNN'.
- Give a formal expression for FNN' given input sequence $\boldsymbol{X}$ (don't use any biases in the adapter).
- Make sure to formally define all parameters introduced by the adapter as well as their exact sizes.
- What are the hyperparameters used in these adapters? How are these hyperparameters typically set?

Parameter Efficient Fine-Tuning

Answer c)

- Let $\boldsymbol{A} \in \mathbb{R}^{d \times r}$, $\boldsymbol{B} \in R^{r \times d}$ be projection matrices in 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 number of trainable parameters low.

Parameter Efficient Fine-Tuning

Question d)

- Can Houlsby adapters only be applied to FNN operators?
- Why or why not?

Parameter Efficient Fine-Tuning

Answer 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.

Parameter Efficient Fine-Tuning

Question e)

- Now assume we fine-tune *only* the up projection of the FNN operator using LoRA adapters.
- Give a formal expression for FNN' given input sequence $\boldsymbol{X}$ (again, no biases in the adapter).
- Make sure to formally define all parameters introduced by LoRA as well as their exact sizes.
- As before, specify the hyperparameters used in LoRA adapters and say how they are typically set.

Parameter Efficient Fine-Tuning

Answer e)

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

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

- The addition comes from the LoRA adapter, which is applied using a residual connection around the weight matrix we are fine-tuning.
- Each adapter has $2dr$ number of parameters, where $r << d$ is the hyperparameter that controls the rank of both projection matrices used by LoRA adapters.

Parameter Efficient Fine-Tuning

Question f)

- Can LoRA adapters only be applied to FNN operators?
- Why or why not?

Parameter Efficient Fine-Tuning

Answer 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.

Parameter Efficient Fine-Tuning

Question g)

- If you haven't already, show that the application of LoRA adapters can be expressed as follows:

$$\text{FNN'}(\boldsymbol{X}) = \boldsymbol{X}(\boldsymbol{W}_{up} + \Delta\boldsymbol{W}_{up})\boldsymbol{W}_{down}, \tag{8}$$

  where $\Delta\boldsymbol{W}_{up}$ is the update introduced during fine-tuning.
- What is the advantage of this formulation?
- Can Houlsby adapters be expressed in a similar way?

Parameter Efficient Fine-Tuning

Answer g) (1)

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

$$\text{FNN'}(\boldsymbol{X}) = (\boldsymbol{X}\boldsymbol{W}_{up} + \boldsymbol{X}\boldsymbol{A}\boldsymbol{B})\boldsymbol{W}_{down} = \boldsymbol{X}(\boldsymbol{W}_{up} + \boldsymbol{A}\boldsymbol{B})\boldsymbol{W}_{down} \tag{9}$$

  where $\Delta\boldsymbol{W}_{up} = \boldsymbol{A}\boldsymbol{B}$.
- Advantage of this formulation: allows to see that learned adapters can be added back into original model after fine-tuning is done.
- This results in fine-tuned model without increase in number of parameters and without corresponding additional runtime costs in forward pass.
- It also allows for modularity: can we choose (i) which weight matrices to tune, and (ii) allows us to remove effect of the adapters from model by simply subtracting $\boldsymbol{A}\boldsymbol{B}$ from tuned weight matrix to recover original model.

Parameter Efficient Fine-Tuning

Answer g) (2)

- Such advantages not immediately possible with Houlsby adapters.
- For one, they were designed to behave similarly to sublayers in 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 residual space using linear projections that include a non-linear activation function in between.
- So if we want to remove impact of these adapters to recover original model, we would have to ensure non-linear activation function has an inverse function and that the learned projections are invertible.

Transformer-Based LLMs

Context

- In this task, we discuss the main components of a transformer-based large language model (LLM).
- In particular, we focus on a causal language model (CLM) similar to the GPT-3 model released by Brown et al. (2020).

Transformer-Based LLMs

Question a)

- Why do transformer-based language models require positional encodings?
- Given a language model with L transformer layers, how many positional encoding layers does this model use?

Transformer-Based LLMs

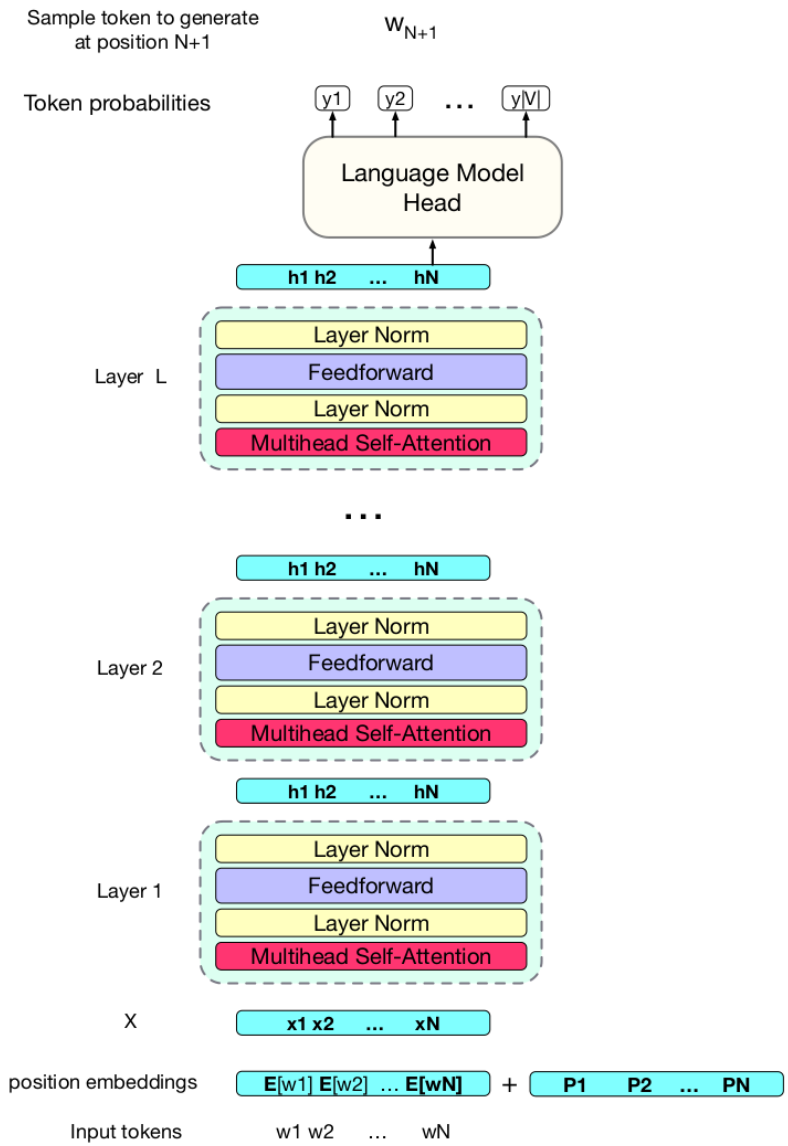
Answer a) (1)

- Positional encodings are required to provide the transformer model with information about 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".
- Positional encodings are *typically* added to 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.

Transformer-Based LLMs

Answer a) (2)

- Recall from lecture: LMs as stacked layers of transformers



Transformer-Based LLMs

Question b)

- What is the role of the language model head in a transformer-based language model?
- What sort of "weight tying" is typically used with the language model head?

Transformer-Based LLMs

Answer b)

- **LM head in an LM:** responsible for producing the distribution used for predicting the next token given an input sequence.
- **Common implementation:** softmax layer that projects contextualized (hidden) representations of the input tokens to the vocabulary space and applies softmax function to normalize logit scores into a probability distribution.
- **Weight tying:** weights of language model head are typically tied to (static) input embeddings, because both (static) embedding matrix and projection matrix in the LM head are of the same size.
- I.e., 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.

Transformer-Based LLMs

Question c)

- As done in previous exercises, give a formal expression for the output of a multi-head attention operator MHA that uses n self-attention heads SA_i?
- What are the parameters $\boldsymbol{\theta}_{\text{MHA}}$ assuming each SA_i operator is parameterized by $\boldsymbol{\theta}_{\text{SA}_i}$?

Transformer-Based LLMs

Answer 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,

$$\text{MHA} = (\text{SA}_1 \oplus \text{SA}_2 \oplus \ldots \oplus \text{SA}_n)\boldsymbol{W}^O, \tag{10}$$

  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,

$$\boldsymbol{\theta}_{\text{MHA}} = [\boldsymbol{\theta}_{\text{SA}_1}, \boldsymbol{\theta}_{\text{SA}_2}, \ldots, \boldsymbol{\theta}_{\text{SA}_n}, \boldsymbol{W}^O]. \tag{11}$$

Transformer-Based LLMs

Question d)

- Assume you have a pre-trained causal language model CLM parameterized by $\boldsymbol{\theta}_{CLM}$.
- The model is implemented with transformers and follows the GPT-3 architecture, meaning the size of the input and output representations in each transformer layer is $d_H = 12288$.
- If each attention head projects input representations to dimension $d_{\mathrm{SA}_i} = 128$ and the projection *inside* the multi-head attention operator MHA does not change the dimension of its inputs, how many attention heads does each MHA operator have?
- Give a numeric answer and show the calculations that lead to your answer.
- **Hint:** remember that the size of the output representations of a MHA operator are also the same size as its input representations.

Transformer-Based LLMs

Answer d) (1)

- Let's start with what we know.
  1. The output of each self-attention layer SA_i is of size 128.
  2. The size of the output representations of a MHA operator are the same size as the input representations, in this case $d_H = 12288$.
  3. The MHA operation is given by Eq. 10.
  4. 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:

$$n = \frac{12288}{128} = 96. \tag{12}$$

Transformer-Based LLMs

Answer d) (2)

- 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.

Transformer-Based LLMs

Question e)

- Given that the model above follows the GPT-3 architecture, it has $L = 96$ transformer layers, a maximum input sequence length of 2048, and a vocabulary size of $V = 50000$.
- How many parameters does this CLM model have? I.e. size of θ_{CLM}? Which component has the most parameters?
- Give numeric answers, show calculations that lead to them.
- To answer the question, you may ignore all biases and layer normalization operators.
- In addition, assume:
  - Positional embeddings are not learned,
  - Weight tying takes place in LM head,
  - FNN layers inside each transformer layer project representations up to a space 4 times the size of the input and then back down (see forward pass of transformer layer in Exercise 6).

Transformer-Based LLMs

Answer e) (1)

- The components that such a CLM has are:
  1. a static embedding matrix followed by
  2. a positional embeddings layer followed by
  3. a stack of L transformer layers followed by
  4. 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.

Transformer-Based LLMs

Answer e) (2)

- So, total number of parameters in this architecture would be:

$$|\boldsymbol{\theta}_{CLM}| = \sum_{i=1}^{L} (|\boldsymbol{\theta}_{\text{MHA}_i}| + |\boldsymbol{\theta}_{\text{FNN}_i}|) + |\boldsymbol{\theta}_{lm\_head}|. \tag{13}$$

  where $\boldsymbol{\theta}_{\text{MHA}_i}$ and $\boldsymbol{\theta}_{\text{FNN}_i}$ are parameters in multi-head attention and FNN components in i-th transformer layer, resp.
- 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 $|\boldsymbol{\theta}_{\text{MHA}_i}| = (96 \times 4718592) + 12288^2 = 452984832 + 150994944 = 603979776 \approx 604M$.
  - $|\boldsymbol{\theta}_{\text{FNN}_i}| = 2 \times (12288 \times (12288 \times 4)) = 12288^2 \cdot 8 = 1207959552 \approx 1.2B$.
  - $|\boldsymbol{\theta}_{lm\_head}| = d_H \times V = 12288 \times 50000 = 614400000 \approx 614M$.

Transformer-Based LLMs

Answer e) (3)

- All together, we have:

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

- Indeed, largest variant of GPT-3 has ≈ 175 billion parameters.
- 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 further store gradients of all parameters during backward pass (doubling memory requirements).
- Hence the massive engineering effort to train these models models in a distributed manner over thousands of GPUs.