

Introduction

References

- [Nice overview \(\[https://lightning.ai/pages/community/article/understanding_llama_adapters/#prompt-tuning-and-prefix-tuning\]\(https://lightning.ai/pages/community/article/understanding_llama_adapters/#prompt-tuning-and-prefix-tuning\)\)](https://lightning.ai/pages/community/article/understanding_llama_adapters/#prompt-tuning-and-prefix-tuning).

The goal of Transfer Learning is to adapt a Pre-Trained model for a Source task (the "base" model) to solve a new Target task.

Fine-Tuning (additional training with Source task-specific examples) is a common method of adaptation.

But *Prompt Engineering* can be used for adaptation as well

- creating a prompt that adapts the text-continuation ("predict the next") behavior of a LLM
- to produce a solution to the Target task

Large Language Model as a Universal base model

Within the context of NLP tasks

- Text to text is a [Universal API \(NLP Universal Model.ipynb#A-Universal-API:-Text-to-text\)](#)
 - The Target task's input and output can both be re-formatted into text
- Large Language Models (LLM) can be a Universal "base" Model
 - convert all Target tasks into instances of the LM "predict the next" task
- Eliminating the need for Target task specific "head" layers to be appended to the base model

An essential part of the Universal API is converting an example of the Target task

- so that the text-continuation ("predict the next") task
- produces a solution to the Target Task

To illustrate, suppose we want our LLM base model to adapt to solving the task of Summarization.

A training example for the Summarization task might look like

{PREFIX} {DOC} Summary: {SUMMARY}

where

- {DOC} and {SUMMARY} are placeholders for the features (i.e., document) and target/label (i.e., the summary)
- {PREFIX} are *instructions* for the summarization task. For example
 - Produce a one paragraph summary of the following:

We refer to

- the text up to and including the **Summary** : as the *prompt*
 - the features of the converted example
- the remainder of the text (i.e., {SUMMARY}) as the *continuation*
 - the target of the converted example

The features for a test example (i.e, a request to summarize) would be the prompt without a continuation

```
{PREFIX} {DOC} Summary:
```

We would hope that the LLM's completion (continuation) of this prompt would be the target {SUMMARY}

This representation of the relationship between features and target for the Target task

- adapts the LLM
- by causing it to compute
 $\backslash\text{pr}\{\text{TARGET}\} \mid \{\text{PREFIX}\} \{\text{DOC}\}$
- rather than the native LLM objective
 $\backslash\text{pr}\{\text{TARGET}\} \mid \{\text{DOC}\}$

That is:

- $\{\text{PREFIX}\}$ *conditions* the LLM to produce a continuation
- that satisfies the Target Task

Thus, the PREFIX

- acts as an *instruction*
- describing the Target task
- which specializes the Universal Model to the Target task

Are there some versions of the instruction {PREFIX}

- that lead to a "better" answer (continuation)

Creating a good instruction is the goal of *Prompt Engineering*.

Many approaches to Prompt Engineering sound ad hoc.

- the best, however, are validated by experiments
 - Chain of Thought: "Let's think step by step"

We will explore methods to find the *optimal* instruction.

This will be called *Prompt Tuning*.

Prompt Design/Prompt Engineering via Tuning

A base model may be adapted to solve a Target Task *without fine-tuning*

- using Prompt Engineering
- crafting a prompt
 - that conditions the LLM text-continuation behavior
 - to produce output consistent with a Target Task

This is *parameter efficient* in that **no** existing parameters are changed, nor are any added.

The conditioning prompt usually consists of

- detailed "instructions"
- exemplars: examples of the input/output relationship for the Target task

In the above Summarization task example, we could imagine various choices for the instructions {PREFIX}

- Summarize the following article: [SEP]
- Produce a summary of: [SEP]
- A "summary" has the following properties ... Create a summary of: [SEP]
- Exemplars: a number of {DOC}:{SUMMARY} pairs

Does it matter which we choose ?

- the last two, being more specific, might be preferable
- but at the cost of using a greater fraction of the LLM model's fixed maximum Context length

Prompt engineering (prompt design) is the "art" of constructing prompts in order to get an LLM to solve a task.

It is an *inference time* technique

- does not modify parameters of base model
- in contrast to Fine Tuning

This has been treated more as an art ("GPT Whisperer") than a science.

- rules of thumb, without scientific validation

Hard prompt tuning

We can formalize prompt design as a formal task.

One can imagine $\{\text{PREFIX}\}$ as a sequence of token *variables*

$$\langle \text{TOK}_1 \rangle, \dots, \langle \text{TOK}_p \rangle$$

We can evaluate the quality of the prefix

- as measured through the evaluation of Performance Metric \mathcal{M}
- on an out of sample set of examples $\tilde{\mathbf{X}}$ that are instances of the Target task

Prompt design can be viewed as an optimization task

- finding the optimal tokens in the {PREFIX}
- we treat each token variable $\langle \text{TOK} \rangle$ as a *parameter* to be solved for

$$\text{PREFIX}^* = \underset{\mathbf{x} \in \tilde{X}}{\operatorname{argmax}} \text{PREFIXM}_{\mathbf{x}} (\text{pr} \mathbf{y} | \text{PREFIX}, \mathbf{x})$$

Because tokens are discrete (hard) values, we refer to this method as *Hard Prompt Fine-Tuning*

- optimizing the prompt at the *token* level

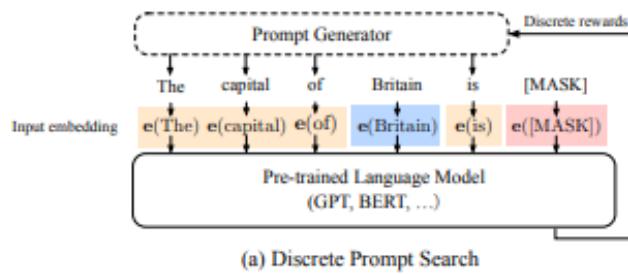
The fact that tokens are discrete (rather than continuous) values means

- we can't differentiate with respect to token values
- so can't optimize by Gradient Descent

Without differentiability, hard prompt fine tuning may devolve to

- an exhaustive (but finite) search for the optimal {PREFIX} sequence

Discrete Prompt Search



Attribution: <https://arxiv.org/pdf/2103.10385.pdf#page=3>

Soft prompt tuning; Prefix tuning

References

- [The Power of Scale for Parameter-Efficient Prompt Tuning](https://arxiv.org/pdf/2104.08691.pdf)
(<https://arxiv.org/pdf/2104.08691.pdf>)
- [Prefix-Tuning: Optimizing Continuous Prompts for Generation](https://arxiv.org/pdf/2101.00190.pdf)
(<https://arxiv.org/pdf/2101.00190.pdf>)

The problem with Hard Prompt Tuning is that we can't differentiate with respect to the tokens.

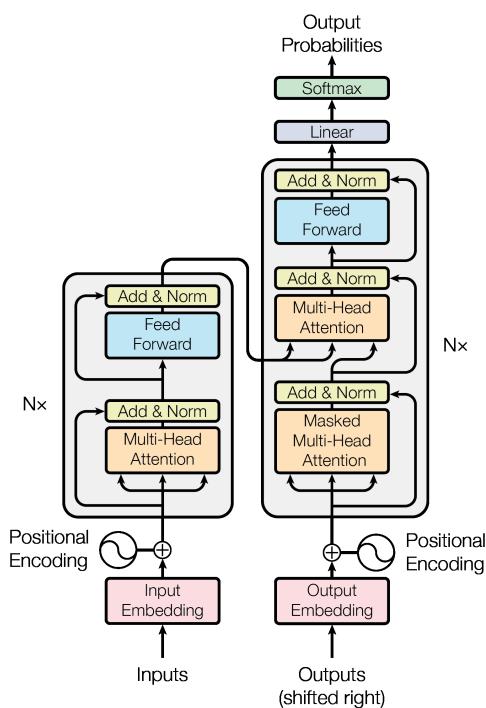
But

- an examination of the Transformer architecture
- which we will typically use to solve our NLP tasks
- offers a simple solution

Almost immediately

- the Transformer changes the discrete token values
- into *continuous* vectors: the Embeddings

Transformer (Encoder/Decoder)



The Input Embedding layer

- maps a token
 - encoded as a OHE vector of length $|\mathcal{V}|$, indicating the index within Vocabulary \mathcal{V}
- to an "embedding" vector of length d (the internal dimension of the output of all layers in the Transformer)

This mapping is (conceptually) implemented via a matrix

- of size $(|\mathcal{V}| \times d)$
- whose elements are *parameters* that are solved for during Gradient Descent

Denoting the embedding of token $\boxed{\text{\tt \backslash x_tt}}$ as $e(\textcolor{red}{\text{\tt \backslash x}})$

- the Input Embedding layer transforms input sequence of *discrete* token values

$$\textcolor{red}{\text{\tt \backslash x}}_{(0)}, \dots, \textcolor{red}{\text{\tt \backslash x}}_{(\bar{T})}$$

to sequence of *continuous* embedding values

$$e(\textcolor{red}{\text{\tt \backslash x}}_{(0)}), \dots, e(\textcolor{red}{\text{\tt \backslash x}}_{(\bar{T})})$$

So

- rather than adding a prefix of tokens
 $\langle \text{TOK}_1 \rangle, \dots, \langle \text{TOK}_p \rangle$
- to input $\backslash \text{x}$
- resulting in input
 $\langle \text{TOK}_1 \rangle, \dots, \langle \text{TOK}_p \rangle \ \backslash \text{x}$

We add

- a prefix of embeddings
 $\tilde{e}_{(1)}, \dots, \tilde{e}_{(p)}$
- to the sequence that is the embedding of sequence $\backslash \text{x}$
 $e(\backslash \text{x}_{(0)}), \dots, e(\backslash \text{x}_{(\bar{T})})$
- resulting in the output of the Embedding layer producing
 $\tilde{e}_{(1)}, \dots, \tilde{e}_{(p)} \ e(\backslash \text{x}_{(0)}), \dots, e(\backslash \text{x}_{(\bar{T})})$

$$\tilde{e}_{(1)}, \dots, \tilde{e}_{(p)}$$

- are called *pseudo tokens*
- they are embedding vectors
- that *don't necessarily correspond* to any true token value in the vocabulary
 - just vectors of length d

Most significantly

- they are *parameters*
- that are solved for by Gradient Descent !

We solve the optimization problem

- find the optimal prefix of *embeddings*
- rather than the optimal prefix of tokens

Since the embedding of pseudo tokens does not have to be human-readable

- we can use a very small number of them
- we can place them anywhere in $\backslash x$, not just as a prefix
- the special case where the placeholders are restricted to a prefix of $\backslash x$ is called *prefix tuning*

In effect: the embeddings of pseudo tokens

- represent instructions to perform the Target task
- written in non-human language

Discrete Prompt Search

GPT Understands, Too

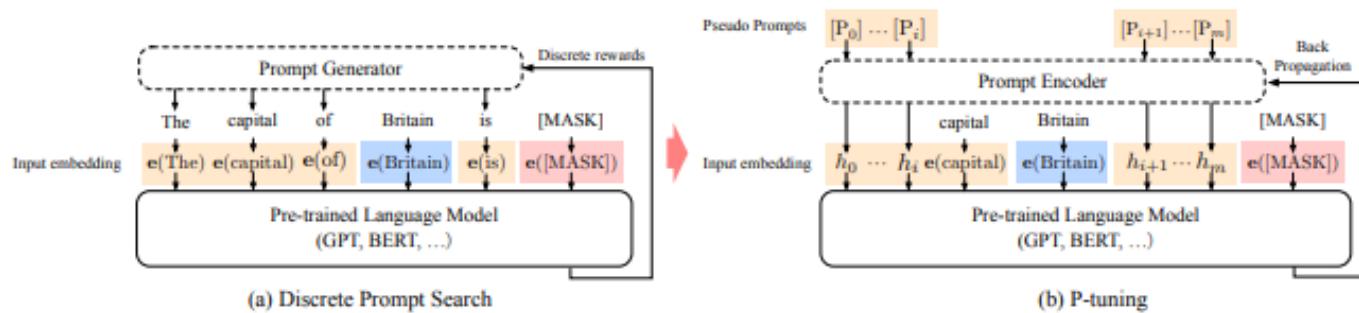


Figure 2. An example of prompt search for “The capital of Britain is [MASK]”. Given the context (blue zone, “Britain”) and target (red zone, “[MASK]”), the orange zone refer to the prompt tokens. In (a), the prompt generator only receives discrete rewards; on the contrary, in (b) the pseudo prompts and prompt encoder can be optimized in a differentiable way. Sometimes, adding few task-related anchor tokens (such as “capital” in (b)) will bring further improvement.

Attribution: <https://arxiv.org/pdf/2103.10385.pdf#page=3>

During Soft Prompt Tuning

- we use a small number of Target task examples
- to learn the embeddings for the pseudo tokens
- keeping the embeddings of non-pseudo tokens and all other weights of the base model frozen

Since only the embeddings of the new pseudo tokens are learned

- **all** Target task specific information from the Fine-Tuning Target training dataset
- is encoded in the new embeddings

Soft prompt tuning: refinements

Recall

- the embeddings of pseudo tokens act as a kind of "instruction" to perform the Target task
- Transformer blocks are stacked in many models
 - thus, there is an embedding in each Transformer block in the stack

Our initial description of prompt tuning created pseudo tokens only at the first block in the stack of Transformer blocks.

Different methods have been tried to add embeddings at the pseudo token positions at *other* blocks in the stack.

One reference [Prefix-Tuning: Optimizing Continuous Prompts for Generation \(<https://arxiv.org/pdf/2101.00190.pdf>\)](#) learns embeddings corresponding to the positions of pseudo tokens

- at every level of the stack

Another reference ([LLaMA-Adapter \(<https://arxiv.org/pdf/2303.16199.pdf#page=3>\)](#)) learns embeddings corresponding to the positions of pseudo tokens

- only at the *top-most* levels of the stack
- perhaps consistent with the result of removing spans of Adapters reported in the Adapter section above
 - adaptation is most influential at the *top* levels of the stack

Results: Adaptation via prompts

Space efficiency

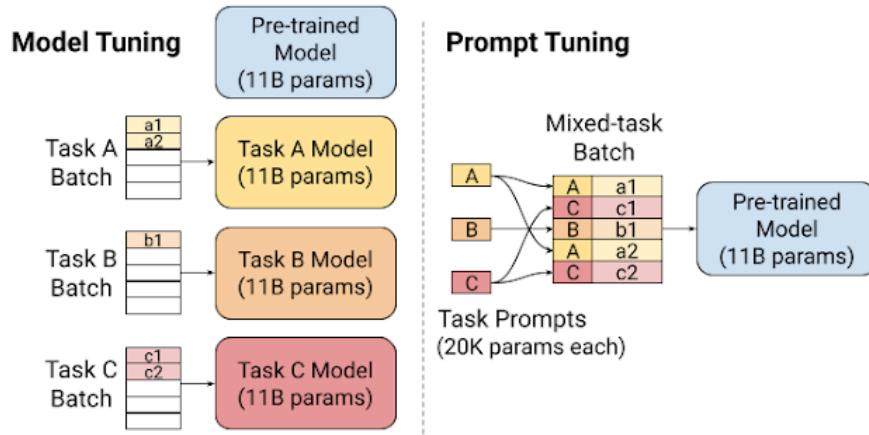
Suppose we have 3 Target tasks: A, B, C.

Fine-Tuning (*model tuning*) each results in 3 copies of the large base model.

In contrast, since the base model is shared in Prompt Tuning

- We can *separately* learn embeddings for placeholder tokens for each of the 3 tasks
- Place the embeddings for each within the Input Embedding
 - e.g., as rows of the Embedding matrix
- to solve the 3 tasks in a single instance of the base model
- by pre-pending the prefix for the appropriate task to each inference-time example

Adaptation via prompts



Left: With model tuning, incoming data are routed to task-specific models. Right: With prompt tuning, examples and prompts from different tasks can flow through a single frozen model in large batches, better utilizing serving resources.

Attribution: <https://arxiv.org/pdf/2104.08691.pdf#page=2>

Performance of various forms of adaptation

The following table compares various forms of adaptation

- Fine-tuning (model tuning)
- Adapter
 - see the module on [Parameter Efficient Fine Tuning \(ParameterEfficient_TransferLearning.ipynb\)](#)
- Prefix Tuning

The number in parenthesis next to the name of the adaptation is

- the size of the adapted parameters as a fraction of base model parameters.
- note that for all metrics except TER, a bigger performance number is better

We can see that Prefix Tuning

- using only a small number of adapted parameters (0.1% of base model parameters)
- performs similarly or *better* than full Fine-Tuning for many tasks
 - evaluated on base models which are the Medium and Large variants of GPT-2

Performance, by method of adaptation

	E2E					WebNLG								DART						
	BLEU	NIST	MET	R-L	CIDEr	S	BLEU	MET	TER ↓	BLEU	MET	TER ↓	Mover	BERT	BLEURT	U	A	U		
	U	A	S	U	A	S	U	A	U	A	U	A	U	A	U	A	U	A		
GPT-2 _{MEDIUM}																				
FINE-TUNE	68.2	8.62	46.2	71.0	2.47	64.2	27.7	46.5	0.45	0.30	0.38	0.33	0.76	0.53	46.2	0.39	0.46	0.50	0.94	0.39
FT-TOP2	68.1	8.59	46.0	70.8	2.41	53.6	18.9	36.0	0.38	0.23	0.31	0.49	0.99	0.72	41.0	0.34	0.56	0.43	0.93	0.21
ADAPTER(3%)	68.9	8.71	46.1	71.3	2.47	60.4	48.3	54.9	0.43	0.38	0.41	0.35	0.45	0.39	45.2	0.38	0.46	0.50	0.94	0.39
ADAPTER(0.1%)	66.3	8.41	45.0	69.8	2.40	54.5	45.1	50.2	0.39	0.36	0.38	0.40	0.46	0.43	42.4	0.36	0.48	0.47	0.94	0.33
PREFIX(0.1%)	69.7	8.81	46.1	71.4	2.49	62.9	45.6	55.1	0.44	0.38	0.41	0.35	0.49	0.41	46.4	0.38	0.46	0.50	0.94	0.39
GPT-2 _{LARGE}																				
FINE-TUNE	68.5	8.78	46.0	69.9	2.45	65.3	43.1	55.5	0.46	0.38	0.42	0.33	0.53	0.42	47.0	0.39	0.46	0.51	0.94	0.40
Prefix	70.3	8.85	46.2	71.7	2.47	63.4	47.7	56.3	0.45	0.39	0.42	0.34	0.48	0.40	46.7	0.39	0.45	0.51	0.94	0.40
SOTA	68.6	8.70	45.3	70.8	2.37	63.9	52.8	57.1	0.46	0.41	0.44	-	-	-	-	-	-	-	-	

Table 1: Metrics (higher is better, except for TER) for table-to-text generation on E2E (left), WebNLG (middle) and DART (right). With only 0.1% parameters, Prefix-tuning outperforms other lightweight baselines and achieves a comparable performance with fine-tuning. The best score is boldfaced for both GPT-2_{MEDIUM} and GPT-2_{LARGE}.

n.b., for the TER metric: smaller is better

Attribution: <https://arxiv.org/pdf/2101.00190.pdf#page=7>

Prefix length

How long does the prefix need to be ?

- how many pseudo tokens in the prompt

The results of several experiments show

- a small number (10) of pseudo tokens achieves most of the performance
- hence, the number of Target task specific parameters does not need to be large

Effect of Prefix Length on Adaptation via Prefix Tuning

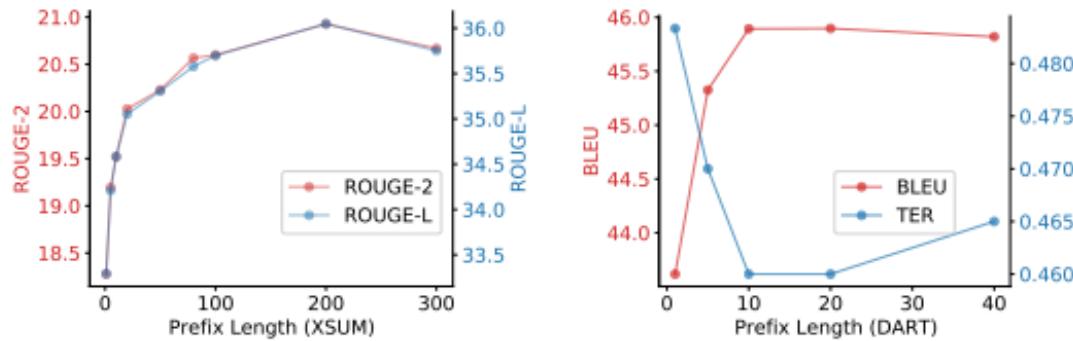


Figure 4: Prefix length vs. performance on summarization (left) and table-to-text (right). Performance increases as the prefix length increases up to a threshold (200 for summarization and 10 for table-to-text) and then a slight performance drop occurs. Each plot reports two metrics (on two vertical axes).

n.b., for the TER metric: smaller is better

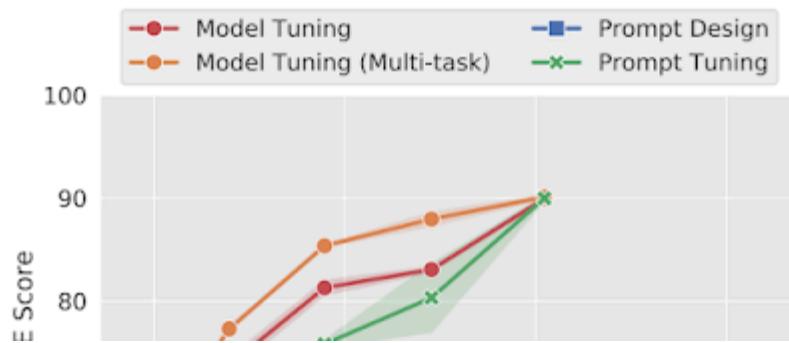
Performance as a function of base model size

The general ordering of adapted models, from best to worst is

- Fine-tuning (model tuning)
- Prompt tuning
- Prompt Design (Prompt Engineering)

However: the gap between Model Tuning and Prompt Tuning *disappears* as we use larger base models.

Adaptation by base model size



Non-language (continuous) tokens

This method creates embeddings for tokens that don't directly correspond to real tokens

- not Natural Language

Thus, they have the potential ability to encode more information than is available in Natural Language.

What might these non-language tokens represent ?

We observe that similar non-language tokens have been used in Reasoning Models

- solving a problem by Chain of Thought ("think step by step")
 - the reasoning chain are the steps that guide a final answer
- these non-language tokens represent "thoughts" in the chain of reasoning
 - that can't be articulated in speech (Natural Language)

Recent experiments have tried to interpret these tokens in the context of Reasoning Models

- by inverting the Embedding
 - mapping continuous embedding vectors
 - to a probability distribution over Natural Language tokens in the vocabulary

For embeddings

- that correspond to Natural Language tokens
 - the probability is concentrated at the single token for which the continuous vector is the embedding
- that are non-language
 - the probability is spread
 - over plausible continuations of the chain of thought
 - almost like *not committing* to a single choice for the continuation
 - preserving the option to decide (via a future token) on which of the plausible options to choose
 - "bread-first search"

For example

- consider a multiple choice question
- with an answer format

The answer is choice C because E

where

- C is a single element of the set of possible choices $\{a, b, c, d\}$
- E is the explanation for the choice C

A continuous token for C may encode near-identical probabilities for choices a and b

- we commit to one of a or b
- only *after* we have explored the reasoning in explanation E

In [2]: `print("Done")`

Done

