In-Context Learning and Indexing

Since GPT-2 (Radford et al.) and GPT-3 (Brown et al.), we have seen that generative large language models (LLMs) pretrained on a general text corpus are capable of in-context learning, which doesn't require us to further train or finetune pretrained LLMs if we want to perform specific or new tasks that the LLM wasn't explicitly trained on. Instead, we can directly provide a few examples of a target task via the input prompt, as illustrated in the example below.

```
Translate the following German sentences into English:

Example 1:
German: "Ich liebe Eis."
English: "I love ice cream."

Example 2:
German: "Draußen ist es stürmisch und regnerisch"
English: "It's stormy and rainy outside."

Translate this sentence:
German: "Wo ist die naechste Supermarkt?"
```

In-context learning is very useful if we don't have direct access to the model, for instance, if we are using the model through an API.

Related to in-context learning is the concept of hard prompt tuning where we modify the inputs in hope to improve the outputs as illustrated below.

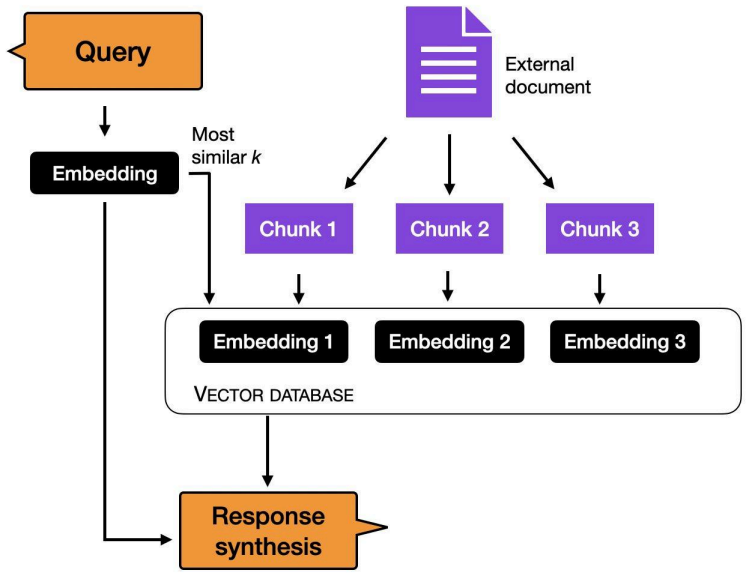
```
1) "Translate the English sentence '{english_sentence}' into German: {german_translation}"

2) "English: '{english_sentence}' | German: {german_translation}"

3) "From English to German: '{english_sentence}' -> {german_translation}"
```

By the way, we call it hard prompt tuning because we are modifying the input words or tokens directly. Later on, we will discuss a differentiable version referred to as soft prompt tuning (or often just called prompt tuning).

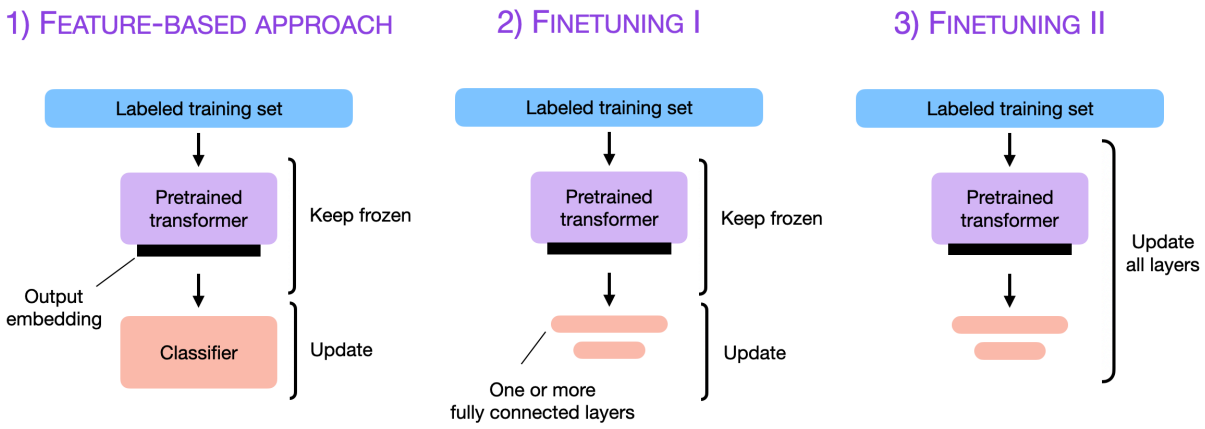
The prompt tuning approach mentioned above offers a more resource-efficient alternative to parameter finetuning. However, its performance typically falls short of finetuning, as it doesn't update the model's parameters for a specific task, which may limit its adaptability to task-specific nuances. Moreover, prompt tuning can be labor-intensive, as it often demands human involvement in comparing the quality of different prompts.

Before we discuss finetuning in more detail, another method to utilize a purely in-context learning-based approach is indexing. Within the realm of LLMs, indexing can be seen as an in-context learning workaround that enables the conversion of LLMs into information retrieval

systems for extracting data from external resources and websites. In this process, an indexing module breaks down a document or website into smaller segments, converting them into vectors that can be stored in a vector database. Then, when a user submits a query, the indexing module calculates the vector similarity between the embedded query and each vector in the database. Ultimately, the indexing module fetches the top k most similar embeddings to generate the response.



The 3 Conventional Feature-Based and Finetuning Approaches



Feature-Based Approach

In the feature-based approach, we load a pretrained LLM and apply it to our target dataset. Here, we are particularly interested in generating the output embeddings for the training set,

which we can use as input features to train a classification model. While this approach is particularly common for embedding-focused like BERT, we can also extract embeddings from generative GPT-style model.

The classification model can then be a logistic regression model, a random forest, or XGBoost – but linear classification like logistic regression works best

```
model = AutoModel.from_pretrained("distilbert-base-uncased")

# ...
# tokenize dataset
# ...

# generate embeddings
@torch.inference_mode()
def get_output_embeddings(batch):
    output = model(
        batch["input_ids"],
        attention_mask=batch["attention_mask"]
    ).last_hidden_state[:, 0]
return {"features": output}

dataset_features = dataset_tokenized.map(
  get_output_embeddings, batched=True, batch_size=10)

X_train = np.array(imdb_features["train"]["features"])
y_train = np.array(imdb_features["train"]["label"])

X_val = np.array(imdb_features["validation"]["features"])
y_val = np.array(imdb_features["validation"]["label"])

X_test = np.array(imdb_features["test"]["features"])
y_test = np.array(imdb_features["test"]["label"])

# train classifier
from sklearn.linear_model import LogisticRegression

clf = LogisticRegression()
clf.fit(X_train, y_train)
```

```
print("Training accuracy", clf.score(X_train, y_train))
print("Validation accuracy", clf.score(X_val, y_val))
print("test accuracy", clf.score(X_test, y_test))
```

Finetuning I – Updating The Output Layers

A popular approach related to the feature-based approach described above is finetuning the output layers (we will refer to this approach as finetuning I). Similar to the feature-based approach, we keep the parameters of the pretrained LLM frozen. We only train the newly added output layers, analogous to training a logistic regression classifier or small multilayer perceptron on the embedded features.

In theory, this approach should perform similarly well, in terms of modeling performance and speed, as the feature-based approach since we use the same frozen backbone model. However, since the feature-based approach makes it slightly easier to pre-compute and store the embedded features for the training dataset, the feature-based approach may be more convenient for specific practical scenarios.

```
model = AutoModelForSequenceClassification.from_pretrained(
    "distilbert-base-uncased",
     num_labels=2
)

# freeze all layers
for param in model.parameters():
    param.requires_grad = False

# then unfreeze the two last layers (output layers)
for param in model.pre_classifier.parameters():
    param.requires_grad = True

for param in model.classifier.parameters():
    param.requires_grad = True

# finetune model
lightning_model = CustomLightningModule(model)
```

```
trainer = L.Trainer(
    max_epochs=3,
    ...
)

trainer.fit(
  model=lightning_model,
  train_dataloaders=train_loader,
  val_dataloaders=val_loader)

# evaluate model
trainer.test(lightning_model, dataloaders=test_loader)
```

Finetuning II – Updating All Layers

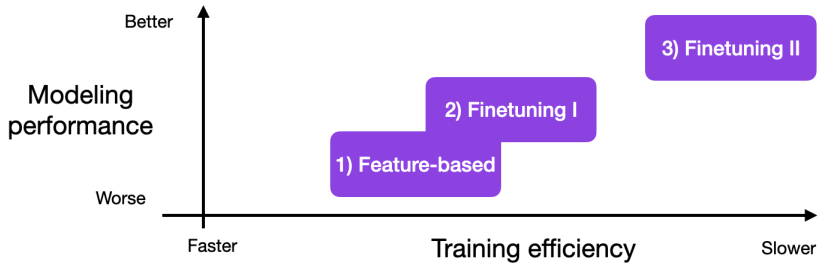
While the original BERT paper (Devlin et al.) reported that finetuning only the output layer can result in modeling performance comparable to finetuning all layers, which is substantially more expensive since more parameters are involved. For instance, a BERT base model has approximately 110 million parameters. However, the final layer of a BERT base model for binary classification consists of merely 1,500 parameters. Furthermore, the last two layers of a BERT base model account for 60,000 parameters – that's only around 0.6% of the total model size.

Our mileage will vary based on how similar our target task and target domain is to the dataset the model was pretrained on. But in practice, finetuning all layers almost always results in superior modeling performance.

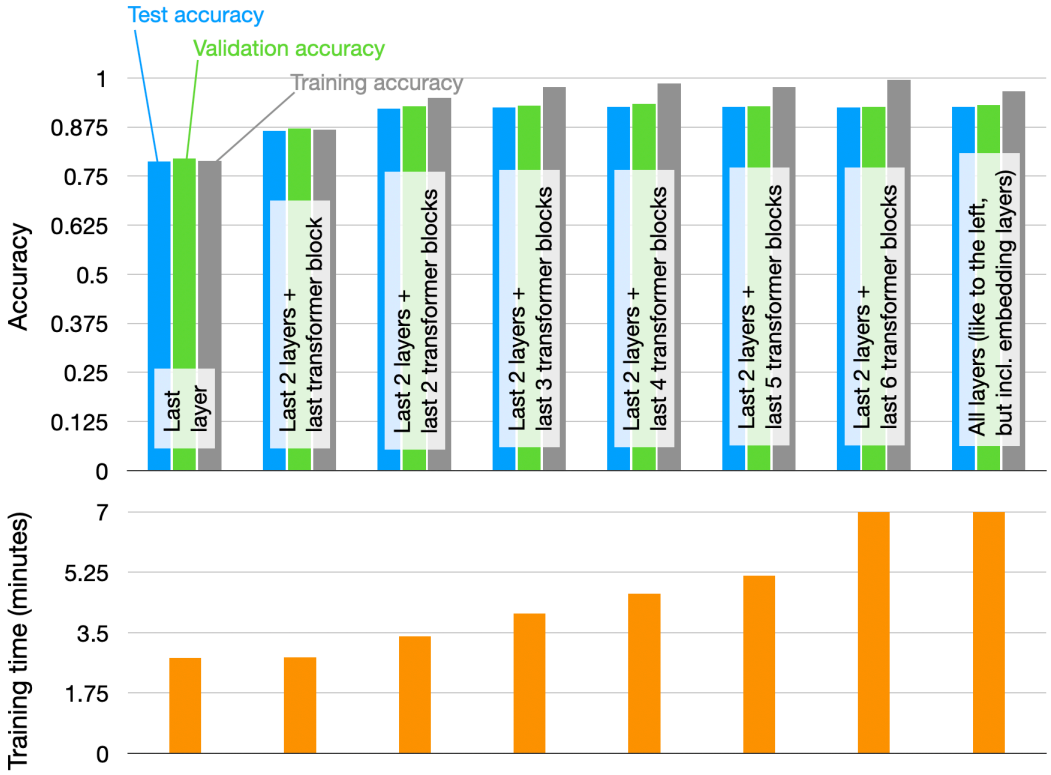
So, when optimizing the modeling performance, the gold standard for using pretrained LLMs is to update all layers (here referred to as finetuning II). Conceptually finetuning II is very similar to finetuning I. The only difference is that we do not freeze the parameters of the pretrained LLM but finetune them as well:

```
model = AutoModelForSequenceClassification.from_pretrained(
    "distilbert-base-uncased",
     num_labels=2
)

# freeze layers (which we don't do here)
# for param in model.parameters():
#    param.requires_grad = False


# finetune model
lightning_model = LightningModel(model)
```

```
trainer = L.Trainer(
    max_epochs=3,
    ...
)

trainer.fit(
  model=lightning_model,
  train_dataloaders=train_loader,
  val_dataloaders=val_loader)

# evaluate model
trainer.test(lightning_model, dataloaders=test_loader)
```



For instance, we can sometimes get the same modeling performance by training only half of the model (but more on parameter-efficient finetuning in the next section).



[Complete Code for Different Methods of Fine Tuning]

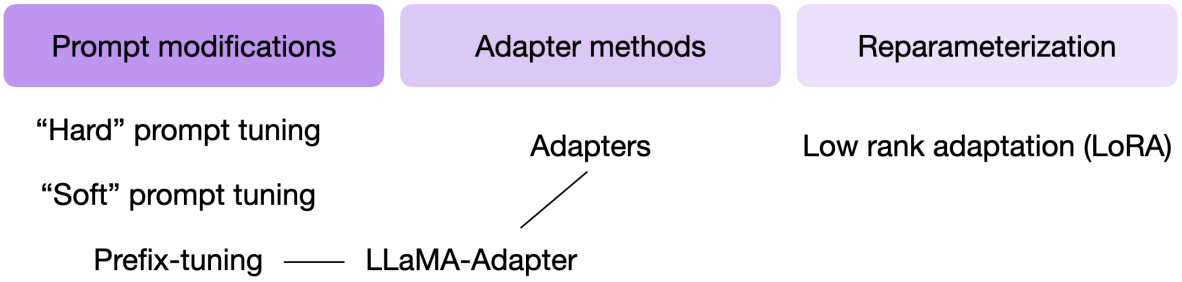
Parameter-Efficient Finetuning

Parameter-efficient finetuning allows us to reuse pretrained models while minimizing the computational and resource footprints.

Better modeling performance (reduces overfitting);

Less storage (majority of weights can be shared across different tasks).

Some of the most widely used PEFT techniques are summarized in the figure below.



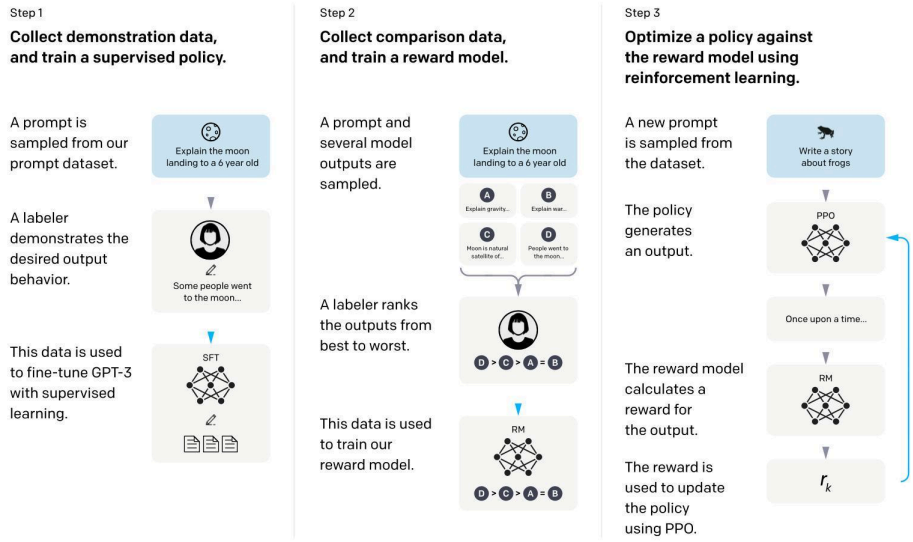
So, how do these techniques work?

In a nutshell, they all involve introducing a small number of additional parameters that we finetuned (as opposed to finetuning all layers as we did in the Finetuning II approach above). In a sense, Finetuning I (only finetuning the last layer) could also be considered a parameter-efficient finetuning technique. However, techniques such as prefix tuning, adapters, and low-rank adaptation, all of which “modify” multiple layers, achieve much better predictive performance (at a low cost).

Reinforcement Learning with Human Feedback

In RLHF, human feedback is collected by having humans rank or rate different model outputs, providing a reward signal. The collected reward labels can then be used to train a reward model that is then in turn used to guide the LLMs adaptation to human preferences.

The reward model itself is learned via supervised learning (typically using a pretrained LLM as base model). Next, the reward model is used to update the pretrained LLM that is to be adapted to human preferences -- the training uses a flavor of reinforcement learning called proximal policy optimization (Schulman et al.).



Why use a reward model instead of training the pretained model on the human feedback directly? That's because involving humans in the learning process would create a bottleneck since we cannot obtain feedback in real-time.

Fine-tuning all layers of a pretrained LLM remains the gold standard for adapting to new target tasks, but there are several efficient alternatives for using pretrained transformers. Methods such as feature-based approaches, in-context learning, and parameter-efficient finetuning techniques enable effective application of LLMs to new tasks while minimizing computational costs and resources.

Moreover, reinforcement learning with human feedback (RLHF) serves as an alternative to supervised finetuning, potentially enhancing model performance.

Parameter-Efficient LLM Finetuning: Prompt Tuning And Prefix Tuning

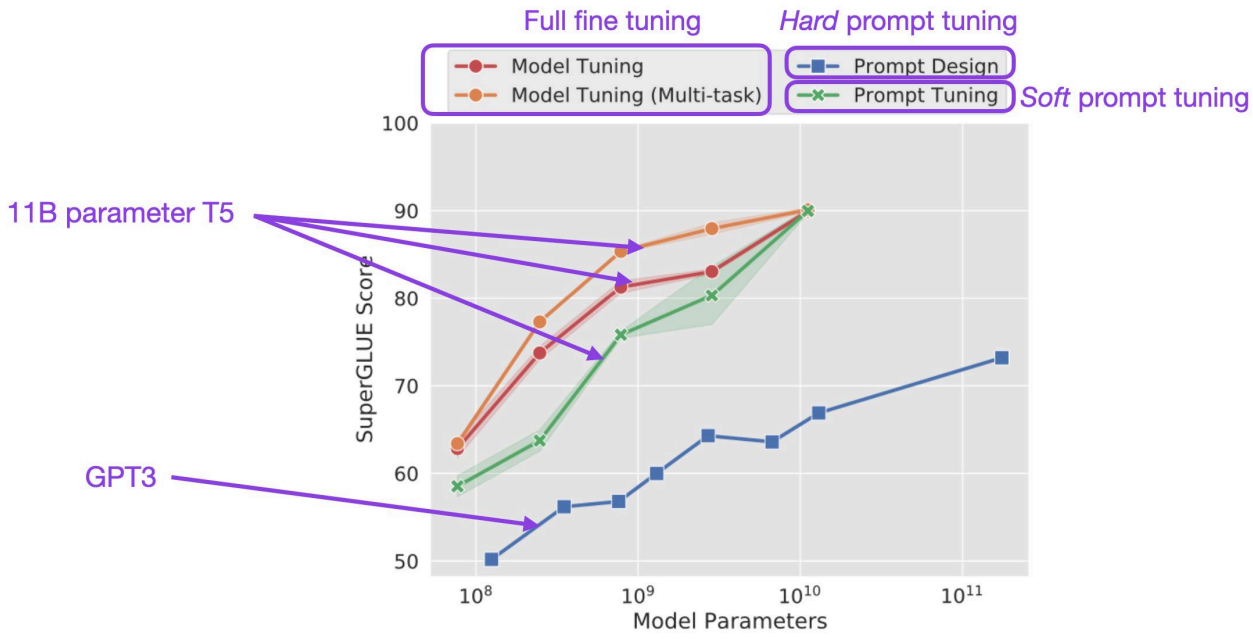
In contrast to hard prompt tuning, soft prompt tuning (Lester et al. 2021) concatenates the embeddings of the input tokens with a trainable tensor that can be optimized via backpropagation to improve the modeling performance on a target task.

In pseudocode, this looks like as follows:

```
soft_prompt = torch.nn.Parameter( # Make tensor trainable
    torch.rand(num_tokens, embed_dim)) # Initialize soft prompt tensor

def input_with_soft_prompt(x, soft_prompt) :
    x = concatenate([soft_prompt, x], # Prepend soft prompt to input
                    dim=seq_len)
    return x

# train soft prompt tensor via gradient descent
train(model(input_with_soft_prompt(x)))

# use model with soft prompts
model(input_with_soft_prompt(x))
```

Soft prompts differ from the discrete text prompts in that they are acquired through back-propagation and is thus adjusted based on loss feedback from a labeled dataset.

Soft prompt tuning is significantly more parameter-efficient than full-finetuning, although the modeling performance of soft prompt-finetuned model can be slightly worse as shown in the figure below.

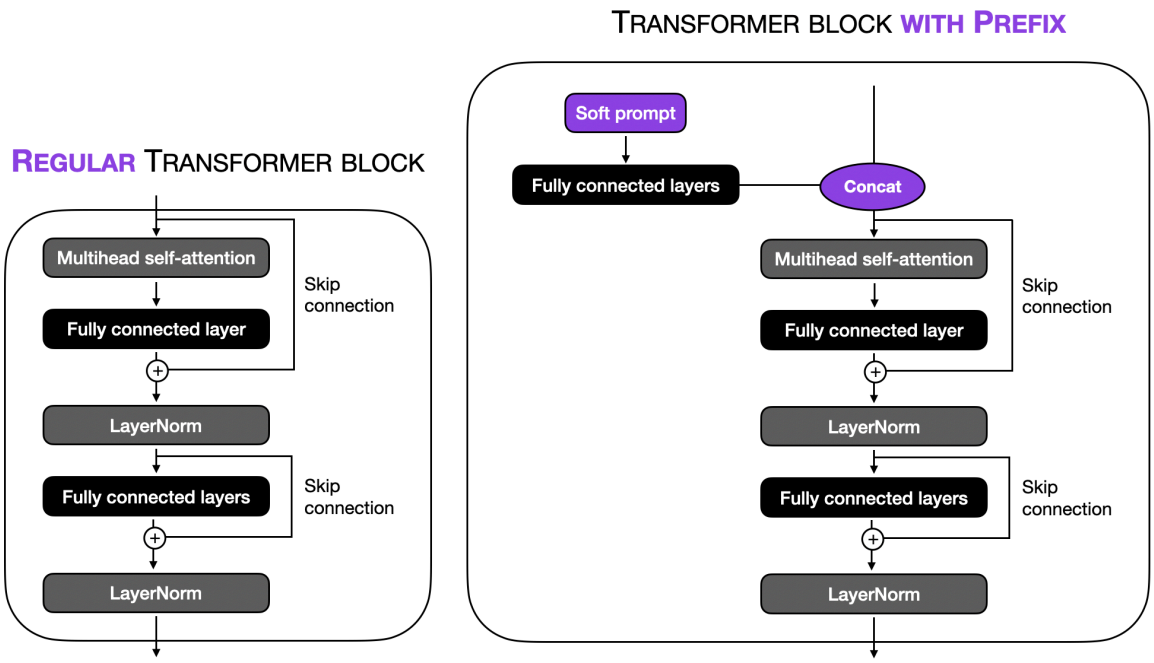


Storage efficiency

If we finetune a pretrained model for a specific task, we have to keep a separate copy of the entire model for each of these tasks. However, with prompt tuning, only a small task-specific (soft) prompt needs to be stored for each task. For instance, a T5 "XXL" model requires 11 billion parameters for each copy of the fine-tuned model. In contrast, the tuned prompts only require 20,480 parameters per task, assuming a prompt length of 5 tokens and a 4096-dimensional embedding size. This represents a reduction of over five orders of magnitude.

From Prompt Tuning to Prefix Tuning

Now, a specific, independently developed flavor of prompt tuning is prefix tuning (Li & Liang 2021). The idea in prefix tuning is to add trainable tensors to each transformer block instead of only the input embeddings, as in soft prompt tuning. Also, we obtain the soft prompt embedding via fully connected layers (a mini multilayer perceptron with two layers and a nonlinear activation function in between). The following figure illustrates the difference between a regular transformer block and a transformer block modified with a prefix.



Note that in the figure above, the “fully connected layers” refer to a small multilayer perceptron (two fully connected layers with a nonlinear activation function in-between). These fully connected layers embed the soft prompt in a feature space with the same dimensionality as the transformer-block input to ensure compatibility for concatenation.

```
def transformer_block_with_prefix(x, soft_prompt):
    soft_prompt = FullyConnectedLayers(soft_prompt)  # Prefix
    x = concatenate([soft_prompt, x],                # Prefix
                    dim=seq_len)                     # Prefix
    residual = x
    x = self_attention(x)
    x = LayerNorm(x + residual)
    residual = x
    x = FullyConnectedLayers(x)
    x = LayerNorm(x + residual)
    return x
```

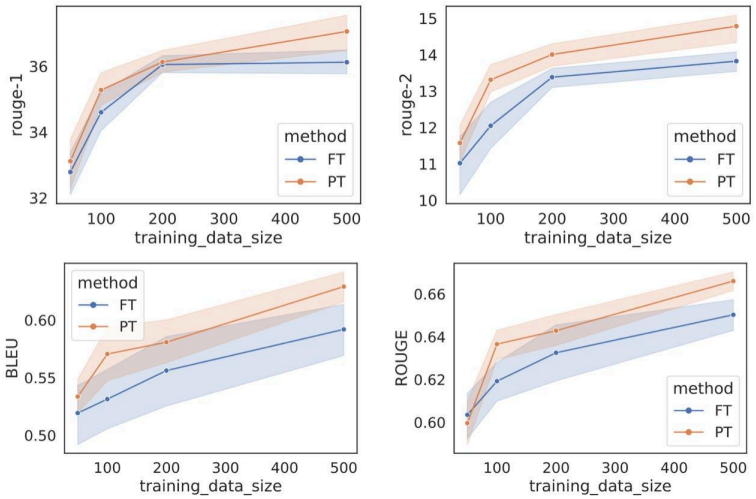
According to the original prefix tuning paper, prefix tuning achieves comparable modeling performance to finetuning all layers while only requiring the training of 0.1% of the parameters – the experiments were based on GPT-2 models. Moreover, in many cases, prefix tuning even outperformed the finetuning of all layers, which is likely because fewer parameters are involved, which helps reduce overfitting on smaller target datasets.

Prefix tuning is competitive with full finetuning

50k examples → E2E; 22k examples → WebNLG; 82k examples → DART

	E2E					WebNLG									DART					
	BLEU	NIST	MET	R-L	CIDEr	BLEU			MET			TER ↓			BLEU	MET	TER ↓	Mover	BERT	BLEURT
						S	U	A	S	U	A	S	U	A						
GPT-2$_{\text{MEDIUM}}$																				
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$_{\text{LARGE}}$																				
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	-	-	-	-	-	-	-	-	-

Prefix tuning outperforms full finetuning in low data settings



Lastly, to clarify the use of soft prompts during inference: after learning a soft prompt, we have to supply it as a prefix when performing the specific task we finetuned the model on. This allows the model to tailor its responses to that particular task. Moreover, we can have multiple soft prompts, each corresponding to a different task, and provide the appropriate prefix during inference to achieve optimal results for a particular task.

Prefix Versus Prompt Tuning

How do soft prompt tuning and prefix tuning compare performance-wise? Unfortunately, since both methods were developed independently and published around the same time, the respective papers don't include a direct comparison. Furthermore, when searching the later parameter-efficient LLM literature, I couldn't find a benchmark that included both of these methods.

Prefix tuning modifies more layers of the model by inserting a task-specific prefix to the input sequence, thus requiring more parameters to be finetuned. On the other hand, soft prompt

tuning involves only finetuning the input prompt embeddings, resulting in fewer parameters being updated. This may make soft prompt tuning more parameter-efficient than prefix tuning, but it could also limit its capacity to adapt to the target task.

Regarding performance, it is reasonable to expect that prefix tuning might perform better since it has more parameters to adjust the model to the new task. However, this may come at the cost of increased computational resources and a higher risk of overfitting. On the other hand, soft prompt tuning might be more computationally efficient, but the smaller number of finetuned parameters could limit the modeling performance.