Deep Neural Networks **(UC Berkeley CS182/282A, Spring 2023)**

Lecture 20: Fine-Tuning & Prompts

10 April 2023

Lecturer: Anant Sahai *Scribe: Romit Barua*

The topics covered today including Fine-Tuning and Prompting is very relevant in deep learning today.

1 Fine-Tuning

We often have a pre-trained model that was trained using lots of data and self-supervised methods. Historically, this has been thought of as a glorified, often non-linear, PCA. We want to use this model as a component or building block for a specific task of interest.

1.1 Classic Approach: Linear Probing

The classic approach is to treat the pre-trained model as a feature extractor.
Given task specific data $\{(x_i, y_i)\}$, you can construct the following;

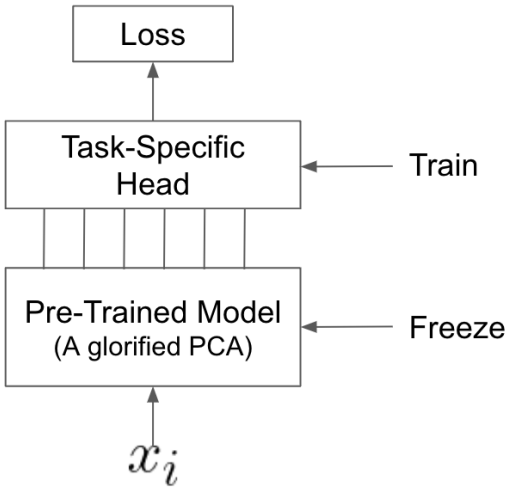


Figure 1: An example linear probing model in which only the task-specific head is trained. All other layers are frozen.

As shown in Figure 1, you just train the head, freezing all weights outside of those in the task-specific head.

There are a number of design choices you can make in terms of actually generating the features. You could A) Generate features from the output of the penultimate layer, B) use outputs from previous layers, C) Average the outputs of various layers together, etc. The method to be used for a given problem can be viewed as a hyper-parameter search.

1.2 Full Fine-Tuning

An alternative approach to the Classic Approach is to fully fine-tune the model. With the full fine-tuning method, you treat the pre-trained model as a building block for your new model with a good initialization. Importantly, you don't view the pre-trained model weights as correct for the given task, but rather you view them as a greater starting point than you would otherwise have with random initialization methods.

The initial intuition is to train all weights in the pre-trained model together, and allow the model gradients to update these weights. However, even if you train everything together, you have to introduce a task-specific head. Practically speaking, there are different choices in regards to how you initialize and implement the training:

Approach #0: Random Initialization of the task-specific head and begin the fine-tuning process

Approach #0 has the major issue where at the beginning of training, the task-specific head is not aligned to the task at all. As such, it will send bad gradients through to the pre-trained layers, potentially causing harmful changes to the pre-trained weights.

Approach #1: Random Initialization of the task-specific head and only train the task-specific head, keeping the other pre-trained weights static. Once the task-specific head has trained and is giving reasonable non-random gradients, we can begin fine-tuning as normal. This approach involves a sequential implementation of feature extraction and fine-tuning.

By allowing the task-specific head to first train on its own, it solves the key issue with Approach #0, ensuring that the task-specific weights are more aligned to the task before allowing their gradients to impact the weights of the pre-trained layers

1.3 Picking a Method

Though empirical testing, people how found that the performance advantage of using the full fine-tuning approach to be substantial. However, there are various considerations when decided between the classic approach and full-fine tuning

(1) The Classic approach runs considerably faster as there are fewer parameters to train.
(2) The Classic approach may be used in situations where the full pre-trained model does not fit in memory, or is large enough for memory resources to be problematic.
(3) Full Fine-Tuning has been shown to have considerably higher accuracy when compared to the Classical approach.

1.4 Further Improvements

1.4.1 Strive to get better pre-trained models

A common approach to improving the task-specific models is to improve the pre-trained model itself. If the pre-trained model is better, than it is likely that the improvement will be passed on when training on alternatives tasks as well.

There are various ways that one can improve the pre-trained model including increasing the size of the network, using more data, **using better and cleaner data**, using more proxy tasks, **meta-learning**[1], etc.

1.4.2 Do something in-between full fine-tuning and classic approaches

You may run into situations where you have more computation power than required to run the Classic approach, but you don't have enough computation power to run full-fine tuning. In those instances, you may consider an approach that is somewhere in-between the class approach and full fine-tuning.

In such a case, you have a number of potential options
(1) Update just a **subset** of weights in the pre-trained model

To do so, there are some implementation considerations. You need to store the activations and other various information required to actually get the gradient to the weights that you are interested in updating.

There are various possible choices you can make in terms of determining which subset of weights to update:

(1A) Update a different set of weights as you go

This method is similar to coordinate descent[2] in that you have to keep information about all the coordinates surrounding the coordinate being updated. This type of method may be used in an instance where one cannot even execute one step of SGD due to the memory constraints driven by the size of the model.

(1B) Selectively unfreeze the model from top-down (eg. fine-tune the top k layers)

Recent papers also present the idea of exclusively unfreezing the bottom layers of the network. The idea is that the bottom layer modify the direct input data, and if the input data for the new task is slightly different from the data used to train the pre-trained model, the bottom layers may have to modify how they interact with the new data.

[1]The goal of pre-trained models is not to just achieve the task it is trained on, but rather to have the ability to adapt to a variety of tasks. Meta-Learning focuses on improving this process. Professor Sahai will be covering this topic in more detail in subsequent lectures.

[2]To get more details on coordinate approach, go back and review materials from prior optimization classes

(1C) Selectively unfreeze based on the nature of the task

This is similar to the idea above. For example, if you know the input data has shifted, you can re-train the bottom layers, whereas if the input data distribution is approximately the same, there may be greater benefit from focusing on updating the weights in the upper layers.

(1D) Selectively unfreeze Attention

When you are fine-tuning a transformer based model, you have two components: the Multi-Layered Perceptron (MLP) and Attention. If you problem has different context-dependence compared to the original training task, then you can choose to unfreeze exclusively the attention components (keys, values, queries), while keeping other aspects of the architecture static.

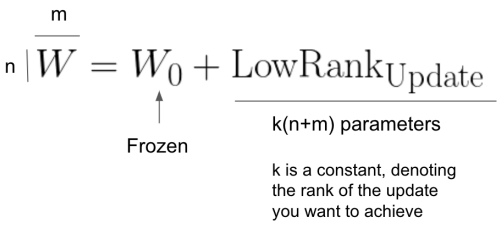
It is important to understand why freezing aspects of the model helps us with run-time memory conservation. We still need to be able to access the pre-trained weights to perform a network forward pass. However, when using an optimizer like Adam, in addition to just the weights, we need to store the momentum terms (average gradients so far for every weight) and the the size of the parameters so that we can make an adaptive step-size. In this scenario, by training everything we are increasing our memory requirements by a factor of 3. This can be even further complicated if the pre-trained model itself is too big to load in memory and one has to load and run each layer at a time. In such a case, we lose activations and information everytime we release a layer from memory, making it more difficult to run gradient steps from earlier layers.

(2) Don't Independently Update Each Weight: Low Rank Updates

If you are doing fine-tuning, you will most likely be moving in a subset of directions. In other-words, some parameters and weights are more important and are being focused on more during the gradient step than others. Another way to think about it is that many models today have billions of parameters, and often more parameters than you have data. In such a case, it is going to be challenging to usefully modify all the parameters anyways. As such, we need that the pattern will be representable in a subset of directions.

When you are running full fine-tuning, it will approximate a low-rank update anyways by moving the important parameters a lot and moving the less important parameters very little. This however still requires too much memory as we are still saving all the relevant information for the unimportant parameters.

In order to perform low rank update, we introduce factorizations to your parameters, such that they will share update information.



The idea of low rank updates can be generalized to what people call adaptors. These are other small structures put inside the pre-trained model to be tuned, keeping everything else frozen. The low rank update can be viewed as a kind of an adaptor.

One way to think about this is to think about SVD:

$$W = \sum_{i=1}^{min(n,m)} \sigma_i u_i v_i^T$$

(n, m)

You can think of SVD as a sum of rank 1 updates, starting with the most important updates.

(3) Work with a Quantized Model to help fit in memory

This can be viewed in a similar manner as compression. More details on this can be found in the Appendix.

2 Prompts

Recall that Word2Vec is trained to take words and embed them in a vector space. One key finding was that these embeddings were able to do analogy problems, even though they were not directly trained to do analogy problems.

2.1 GPT-Style Models

GPT-style models are trained on the task of auto-complete (next token prediction). People then tried to see if these GPT-style models could be used to solve non-autocomplete based problems.

Example
Problem: Return the Capital of a Given Country
We can solve the problem by framing the question in the structure of an autocomplete problem.

Prompt/Input: "The capital of france is"

Model Returns: "Paris"

This is an example of **zero-shot learning**. Zero-shot learning is an attempt to learn a task from no previous examples. This is the lowest limit of few-shot learning in which the model is provided a small number of examples in order to learn what the query is asking.

In the example above, the sentence was generated from the following prompt template:

Prompt Template: "The capital of <insert country> is"

You can solve tasks by creating the appropriate prompt template. The entire pipeline with a blackbox GPT model can be seen below:

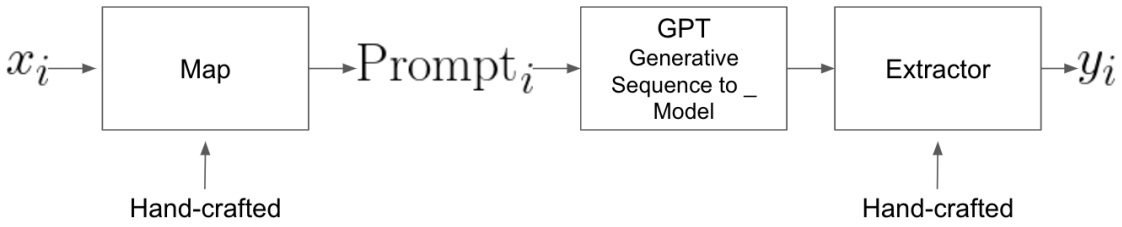


Figure 2: The entire prompting pipeline with a GPT-style model. The initial input map and extractor are handcrafted.

The area of generating prompts to solve specific tasks is generally known as prompt engineering. At present, prompt engineering is an aspirational term, not having fully reached the engineering label yet. The field is currently somewhere between human intuition and more formal principles.

2.2 Few Shot Learning

The main idea of few shot learning is to provide the model some, but not much, training data for the task through the prompt itself. The hope is that additional context will direct the model to provide better answers.

Continuing the example above, you could provide the model more context before asking for the capital for France:

New Prompt: "Let's play Capitals!
The capital of California of is Sacramento
The capital of India is New Delhi
The capital of America is Washington DC
The capital of France is"

While the performance of few-shot learning is better than that of zero-shot learning, the performance is still not great. This may be driven by two issues:

(1) The training data is greater than the context length of the model. In such situations, the training data won't fit in the prompt.

One option is to split the data into k-batches. Then you will feed each batch into the model one at a time and treat the outputs similar to that of a random forest. Further, taking inspiration from SVMs, we can say that there should be a duality between examples and features, and find the set that represent the best examples. In other words, you can train and figure out which training data is best for getting the solution.

This can be be considered basic prompt engineering as it treats the GPT model itself like a blackbox and does not allow for the updating of weights.

(2) We provide prompts in human-speech through tokens, but the natural language of computers is vectors.

To address this issue, we may choose to allow the prompts themselves to learn via gradient descent This process is called generating a **soft-prompt**. The key idea here is that we can have gradients go through and stop at the embedding layer. We do this as a result of the discretized nature of natural language.

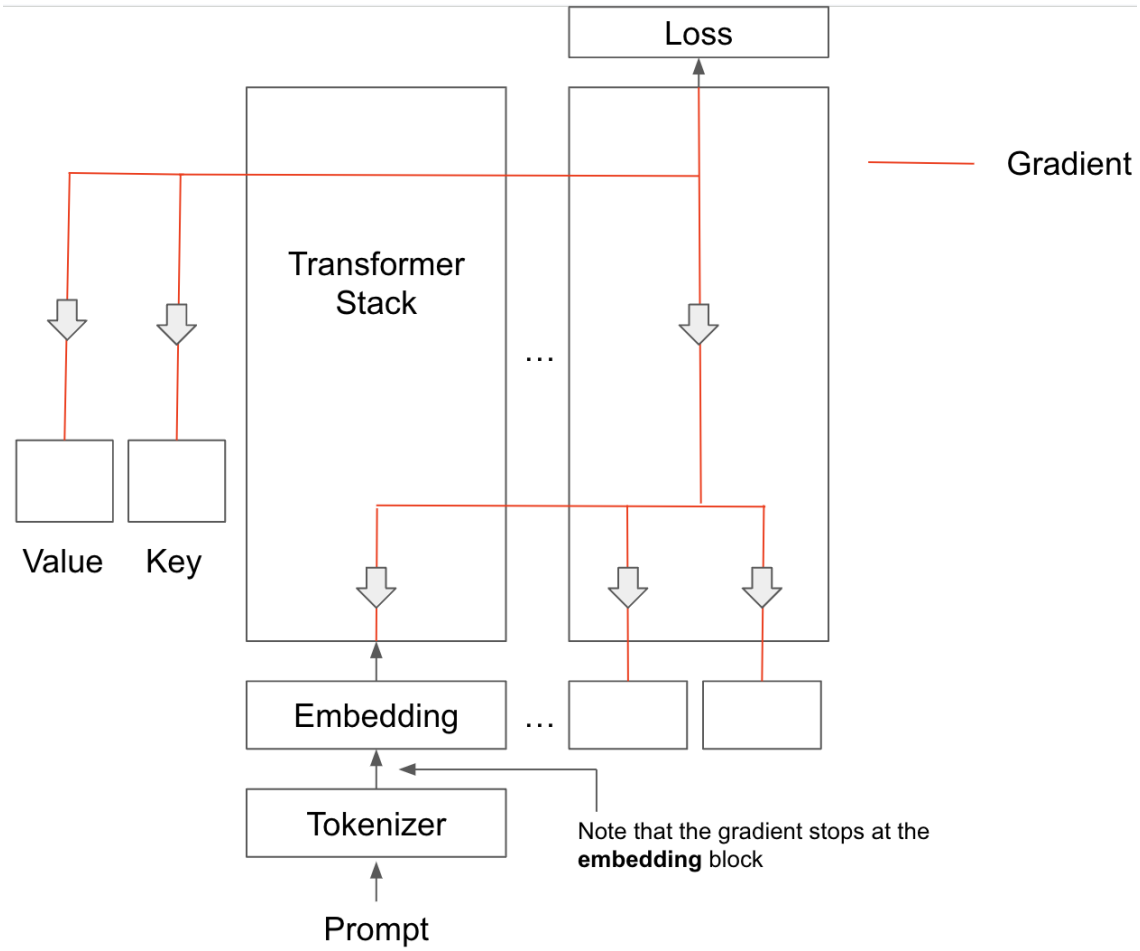


Figure 3: We can have the gradient flow from the loss function, directly to the embedding stage as well as other part of the architecture like the attention modules.

As shown in Figure 2, the gradient flows from the loss to the embedding module. More specifically, the embedding uses weight sharing, so the gradient flows to all of the shared components. The key idea is that the actual words in the prompt are less important when the model can learn the embedding representation itself. Prompt engineering can be thought of as a different, and broader kind of full fine-tuning.

Empirically, people found that the performance increase by soft-prompting was dramatic, with performance after just soft-prompting almost being almost equal to full fine-tuning. Further, the larger the GPT-models, the greater the benefit of soft-prompting.

One important thing to notice is that as the gradients move down, the only thing that they see are the key-value pairs from the previous block. Nothing states that the key-value pairs have to actually correspond to the transformer output for a provided prompt. Therefore, you can conduct soft prompting up the entire transformer stack, by updating the key value pairs directly.

One interesting idea is using repetition tasks to determine what the soft-prompts are saying. In such an example, one could theoretically ask the model to repeat back the original prompt and see how the prompt has changed or been translated. It can be further enhanced by taking the soft-prompt and injecting it directly into the model.

To summarize, with soft-prompting:

(A) Performance dramatically increases

(B) Only requires us to store a small number of parameters

(C) Memory usage is higher

3 Appendix

3.1 Quantization and Compression

It was stated above that one way to train and update weights is to use quantization, a method used for compression.

In compression (especially lossy compression), there are two things happening: (1) You transform the domain (ex. in JPEG compression it is the DCT [Fourier Transform]). Doing so, you will realize that most of the action is in a smaller subspace (2) You then use quantization to give appropriate precision to the places where there is action and nothing to places where there is no action

As stated previously, these models can be thought of as a glorified PCA. When using weight decay, the model will learn to favor specific directions. When performing full fine-tuning, we are locally working with a linearized version of the model around the neighborhood of the initialized weights. If a model has billions of parameters, it is most likely not having an equal amount of motion along all the different weight directions. There are likely some directions that are more important than others.

4 What we wish had been covered

Some key concepts that think we should have covered in more detail:

- Further discuss low rank updates
- Give more clarity in terms of the difference between low rank updates and Quantization
- Spend some more time talking about other applications of of soft-prompting