Lecture 22 Scribe Notes

Ann-Katrine Christiansen & Jesper Hauch

November 8th, 2022

1 Continue Fine-tuning

In the last lecture, material regarding large language transformer based models was covered. More specifically, it was seen how to utilize models, such as BERT and GPT, for tasks different from their surrogate task. Fine-tuning models to a different task, ensured savings on computation and a better starting point for further training. Table 1 shows different methods of fine-tuning covered in this lecture and the last lecture.

	"Feature Extraction" Train new head	"Fine-tuning" Retrain everything	"Prompt Engineering" No gradient step, zero shot, few shot	"Prompt Tuning" In "computer-ese" w/ gradients
Number of parameters that need to be trained	Small to Medium	Huge	None	Small
Amount of task specific training data it can handle	Small to Huge	Small to Huge	Small	Small to Huge
Multi task scalability	Good	Bad	Good	Good
Performance on desired task	OK	Good	Bad to OK	Good

Table 1: Story of fine-tuning so far.

1.1 Feature Extraction

The first column of Table 1 describes a classical approach, where the pre-trained model is used to extract features as previously seen in PCA and k-means. This approach is conducted by removing the head of the pre-trained model and replacing it with a new task-specific head. During training, the weights of the pre-trained model are freezed and only the new head is trained to perform the task. The new task-specific head can choose to ignore or give precedence to certain parts of the pre-trained model depending on their relevance. This is why adding and training a new head works. There are multiple ways to extract the weights/features from the pre-trained model, for instance taking the weights of the last layer or the average of the top k layers as seen in the last lecture. This is known as the feature extraction paradigm.

In this paradigm, a small to medium number of parameters need to be trained since only the new head is trained for the different task. When training a new head, SGD based training is used and the approach can therefore handle any amount of data (the more, the better). Additionally, this paradigm is good in multitask scalability since a new head can be trained for each task. The performance on a desired task is viewed to be "OK", since only the head of the model is trained. Thus, the model only have a few parameters to adapt to the specific task.

1.2 Fine-tuning

Another approach, seen in column two of Table 1, is to not freeze the weights of the pre-trained model and use it as an advantageous starting point for training the model to perform a different task. This paradigm is called fine-tuning. Different strategies regarding the head of the pre-trained model have been used, where one keeps the head for the new task and another removes the head and replaces it with a new task-specific one.

Contrary to the feature extraction paradigm, retraining the entire model entails training a huge number of parameters. However, like the feature extraction paradigm, fine-tuning can handle any amount of data due to SGD based training. Unfortunately, fine-tuning scales badly for multiple tasks. This is due to the large size of the model which needs to be trained separately for each task and catastrophic forgetting/interference (explained more in depth in Section 2). Fine-tuning can still achieve good performance on specific tasks despite the poor multi-task scalability.

1.3 Prompt Engineering

A third approach, basic prompt engineering, treats the entire pre-trained model as a black box, where information from the model is retrieved with prompts. The prompts can be interpreted as questions framed in such a way, that makes the model do the task that you want. There are different ways to construct the prompt, where zero shot and few shot was covered in the last lecture. Examples of these can be found in Box 1. The internals of the model are not considered directly and no additional training is performed. Thus, training parameters using gradient steps is not possible in basic prompt engineering. In this way, you rely on the model's learned embeddings of a language to perform a different task.

Basic prompt engineering can only handle a limited amount of training data, since it must be included in the prompt given to the model for a specific task. Due to the transformer like architecture of the pre-trained model, there is a limit to how much training data can be considered at once due to the quadratic scaling of complexity. Different methods exist to

compress the training data before constructing the prompt. For example, utilizing concepts used in support vector machines, where some data points are considered more important than others. In this paradigm, you have one prompt per task which ensures good multitask scalability as no additional training is needed. For prompt engineering, it is surprising that it even works but performance is not great compared to the other paradigms.

Box 1: Examples of zero shot and few shot

Zero shot is when the model is asked a question directly without any training examples. A zero shot example is to ask the model *"What is the capital of California?"*

Few shot is when the model is provided with a few training examples included in the question or prompt. A few shot example is *"The capital of Massachusetts is Boston. The capital of Arizona is Phoenix. What is the capital of California?"*

In both cases the model is expected to answer *"Sacramento"*.

1.4 Prompt Tuning

Prompt tuning is the last paradigm covered in this lecture and arose from the wish to not be limited by the dataset size and lack of gradients in prompt engineering. The main idea is to create prompts in the computer's continuous spaced vector language ("computer-ese") instead of English or any other human language. In prompt tuning, you start out with an English language prompt in vector form, which is updated by gradient steps to make the prompt more understandable for the computer. Tuning the prompt greatly improves performance seen in basic prompt engineering while maintaining scalability. This paradigm will be covered further in the next lecture.

1.5 Note on Training

In essence, the specific parts of the model architecture, that are trained when learning a new task, is illustrated in Figure 1. In all the aforementioned paradigms, you execute batches at training and you can therefore also execute batches at use/test time. For instance, in prompt engineering you can group together different tasks in a batch and execute all at once. The same applies for feature extraction, where all features from the model can be extracted at once, and then used to run the specific head. When training a model to do multiple tasks at once, each task has its own loss function allowing it to adjust independently.

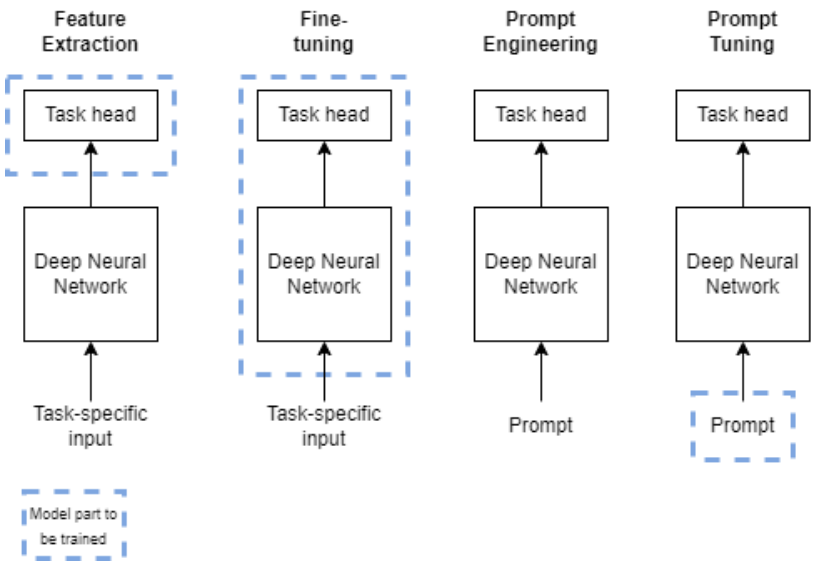

Figure 1: Illustration of model parts that are trained when learning new tasks.

2 Catastrophic Forgetting/Interference

Catastrophic Forgetting describes the phenomena of deep neural networks forgetting how to perform old tasks when trained to do new ones. The old task can be interpreted as the surrogate task performed during pre-training, whereas the new task can be the fine-tuning task at hand. A simple example of catastrophic forgetting in computer vision is found in Box 2.

Catastrophic forgetting is especially of interest in continual learning, where models learns a series of tasks sequentially. Traditionally, this has been largely important for people working in artificial intelligence, since they are trying to achieve models that can keep learning in a real environment. Counterintuitively, the phenomena of catastrophic forgetting can still occur even when continual learning is not trying to be performed.

Catastrophic forgetting does not align with our intuition of how deep learning models work. From a convolutional neural network perspective, our intuition is that early layers learn more basic low-level features, such as edges, local configurations of edges, component pieces, and textures, whereas the last layer learns task-specific features. In reality, this intuition is wrong, since earlier layers might already be somewhat task-specific. The somewhat task-specific means that there is distilling information that is generic to the problem domain. However, this distillation is favoring information that is relevant to the task and irrelevant information is favored less. This corresponds with what is seen in Table 1, where better performance on a desired task is found when the entire model is retrained in the fine-tuning paradigm as opposed to only training the head in feature extraction.

The fact that our intuition is different from reality gives rise to two questions.

1. How are task-specific features learned in early layers?
2. Why can learning these task-specific features early on break performance on previous tasks?

For the first question, it turns out, that when skip connections were introduced in neural network architectures, it became possible for early layers to learn task-specific features. This is a consequence of the weights being directly influenced by the loss from the final layer, which they can adjust to accordingly. The ability to learn task-specific features in early layers turns out to be beneficial in other contexts, such as being able to fit very large models on mobile devices as a result of early exiting. To answer the second question, if the earlier layers have adjusted and shifted out of alignment, the final head is unsuccessfully trying to pull all the layers back into alignment to perform the previous task. The information from the previous task might still be present in the model, which can be evident by the great performance achieved when retraining the previous task head.

Box 2: Example of catastrophic forgetting in image classification

Consider learning image classification with a convolutional neural network and dividing the training data to have each of the classes one at a time when training. All the data is utilized but the ordering in which classes occur is simply changed, as opposed to the random shuffle, that is normally done.

If the above procedure is followed, it can be observed that the model will become good at doing the first class. After the first class the model will do bad at the second class at first, but performance will increase as the model sees more examples of the class. This pattern will repeat itself throughout the rest of the training data. If the model is presented with an image seen from a previous class after a while, it will have forgotten how to correctly classify it even though it was able to do it earlier.

2.1 Solving Catastrophic Forgetting

A remedy for catastrophic forgetting can be to make early layers less task-specific but it is not considered as the main objective. The main objective is to ensure that the old task heads adapt while training new ones. As a result, early layers will have less task-specific information. Approaches that try to achieve this are provided below:

- The naive approach.

- Replay during training.
- Learning without Forgetting.

The naive approach is to do batch learning by having different tasks in the same batch instead of continual learning. In this way, the early layers learn low-level features that are good across all tasks and the heads learn to classify appropriately for all the tasks. This approach is not helpful as each time a new task is introduced, it is necessary to train on all tasks again.

A way to approximate the naive approach is the idea of **replay during training**. This idea is inspired by research in the neuroscience literature, which is described further in Box 3. The engineering approach to replay is to insert examples of the old tasks when training on the new task. As a result, when replaying old tasks to the model, it allows updating the heads of the old tasks so they are not forgotten when training on a new task. Replay is the golden standard of handling catastrophic forgetting today. However, replay incurs a minor problem related to storing examples of the old tasks in the replay buffer.

Another approach, described in the **Learning without Forgetting** paper is to only use new task data to train the network while preserving the original capabilities [1]. This is done by keeping a score for new task examples by creating pseudo-labels from predictions on the new task using the old heads. Afterwards the entire model is retrained on the new task using the pseudo-labels generated previously [2]. Therefore, in this approach, the old task heads are generating the target for the new task.

The learning without forgetting approach is related to *knowledge distillation*, where the general principle is that a learned neural network, that is reasonable good at doing a task, can be used to generate analog labels. This introduces the need for analog loss functions during training, such as mean squared error. The gradients calculated from the loss are used to update the new version of the old heads, and the lower layers. One variation of knowledge distillation is to treat the outputs of old tasks as probabilities, which can be modified by raising them to a (fractional) power, and use cross-entropy loss in retraining. A common choice is to take the square root of the probabilities, as this softens the distinction between large and small probabilities. Oppositely, raising probabilities to a non-fractional power, increases the distinction between large and small probabilities.

The performance of learning without forgetting is subpar compared to replay but is better than doing nothing. The problem with learning without forgetting compared to replay, is that distributional shifts are present, as only the new tasks distribution is observed and the old

task distributions are solely being updated by the new tasks. This causes decay of performing the old tasks.

Box 3: Connecting catastrophic forgetting to neuroscience

In the early development of deep neural networks, researchers found that catastrophic forgetting occurred when they tried to use a pre-trained network to learn a new task. From the perspective of viewing neural networks as some variation of a biological system, it was considered unrealistic that humans forget how to perform an old task when learning a new one.

In the neuroscience literature, it was found that when humans and animals dream, they replay experiences to build better memories. The replay of experiences was used to address the problem of catastrophic forgetting in artificial intelligence, when the concept of replaying training examples was introduced by using a replay buffer.

3 T5/BART

Recall talks about BERT and GPT from previous lectures. BERT is considered as an encoder-only transformer model and a masked auto-encoder, whereas GPT is considered as a decoder-only transformer model and training is predict-next-token. BERT is made for the feature extraction paradigm, since it is good at extracting context-specific features. For GPT, it is more targeted towards the prompt engineering paradigm, where generation of text is of importance.

Researchers found it odd that architectures only incorporated either an encoder or decoder strategy. As a result, Google and Facebook created T5 and BART respectively, which are both encoder-decoder transformer models and are basically the same idea. The idea was that researchers wanted to design a model that targeted the fine-tuning paradigm, which is the most widely used of the paradigms covered. T5 and BART are trained on massive corpora of text like BERT and GPT, except these architectures take advantage of their encoder-decoder architecture to provide greater flexibility for future/downstream tasks. The key idea of T5 and BART is a masked auto-encoder, where entire spans of tokens can be masked without the model knowing how many tokens are masked. This is unlike BERT, where spans of tokens cannot be masked without the model implicitly knowing how many tokens are masked given the positional encoding. An example of this can be found in Box 4.

Box 4: Example of masking in BERT, T5 and BART

Let us consider the following sentence:

"Thank you for inviting me to your party last week."

Imagine masking "for", "inviting", and "last" in the above sentence.

In BERT each masked word is replaced by a mask token. Therefore, masking this sentence would look like the following:

"Thank you <MASK1> <MASK2> me to your party <MASK3> week."

For T5 and BART, entire spans of tokens are masked without implicitly telling the model how many tokens are masked. Therefore, "for" and "inviting" are masked with the same mask token. The masked sentence looks like the following:

"Thank you <MASK1> me to your party <MASK2> week."

There is a subtle difference between how T5 and BART output their predictions to the masked tokens. T5 will fill in the masks by answering a prompt: <MASK1> is "for inviting" and <MASK2> is "last". BART will fill in the masked tokens by returning the unmasked sentence seen at the top of this box.

4 What we wish was explained more in detail

We found that the Learning without Forgetting approach could have been described better in the lecture by use of examples. Additionally, as indicated by one of the reviewers, it was unclear how the differences between BERT, T5, and BART end up affecting model performance.

References

[1] Zhizhong Li and Derek Hoiem. *Learning without Forgetting*. 2016. DOI: 10.48550/ARXIV.1606.09282. URL: https://arxiv.org/abs/1606.09282.

[2] La Tran. *Learning without forgetting simplified - Towards Data Science*. Nov. 2021. URL: https://towardsdatascience.com/learning-without-forgetting-simplified-33243bd0485a.