Lecture 27: Online Learning. Fine-tuning. In-context learning.

CS4787/5777 — Principles of Large-Scale Machine Learning Systems

Note: slicing does not allocate memory! It uses a view.

So far, we've looked at how to train models efficiently at scale, how to use capabilities of the hardware such as parallelism to speed up training, and even how to run inference efficiently. However, all of this has been in the context of so-called "batch learning" (also known as the traditional machine learning setup). In ML theory, this corresponds to the setting of PAC learning (probably approximately correct learning). In batch learning:

- There is typically some fixed dataset $\mathcal{D}$ of labeled training examples (x, y). This dataset is cleaned and preprocessed, then split into a training set, a validation set, and a test set.
- An ML model is trained on the training set using some large-scale optimization method, using the validation set to evaluate and set hyperparameters.
- The trained model is then evaluated once on the test set to see if it performs well.
- The trained model is possibly compressed for efficient deployment and inference.
- Finally, the trained model is deployed and used for whatever task it is intended. Importantly, once deployed, the model does not change!

It turns out that this batch learning setting is not the only setting in which we can learn, and not the only setting in which we can apply our principles!

Online learning takes place in a different setting, where learning and inference are interleaved rather than happening in two separate phases. Concretely, online learning loops the following steps:

- A new labeled example (x_t, y_t) is sampled from $\mathcal{D}$.
- The learner is given the example x_t and must make a prediction $\hat{y}_t = h_{w_t}(x_t)$.
- The learner is penalized by some loss function $\ell(\hat{y}_t, y_t)$.
- The learner is given the label y and can now update its model parameters w_t using (x, y) to produce a new vector of parameters w_{t+1} to be used at the next timestep.[1]

One way to state the goal of online learning is to minimize the **regret**. For any fixed parameter vector w, the regret relative to w is defined to be the extra loss incurred by not consistently predicting using the parameters

[1]Sometimes in practice, the learner does not get the label y immediately but rather a short time later.

w, that is

$$R(w,T) = \sum_{t=1}^{T} \ell(\hat{y}_t, y_t) - \sum_{t=1}^{T} \ell(h_w(x_t), y_t).$$

The regret relative to the entire hypothesis class is the worst-case regret relative to any parameters,

$$R(T) = \sup_w R(w,T) = \sum_{t=1}^{T} \ell(\hat{y}_t, y_t) - \inf_w \sum_{t=1}^{T} \ell(h_w(x_t), y_t).$$

You may recognize this last infimum as looking a lot like empirical risk minimization! We can think about the regret as being the amount of extra loss we incur by virtue of learning in an online setting, compared to the training loss we would have incurred from solving the ERM problem exactly in the batch setting.

What are some applications where we might want to use an online learning setup rather than a traditional batch learning approach?

Algorithms for Online Learning. One algorithm for online learning that is very similar to what we've discussed so far is **online gradient descent**. It's exactly what you might expect. At each step of the online learning loop, it runs

$$w_{t+1} = w_t - \alpha \nabla_{w_t} \ell(h_{w_t}(x_t), y_t).$$

This should be very recognizable as the same type of update loop as we used in SGD! In fact, we can use pretty much the same analysis that we used for SGD to bound the regret. In typical convex-loss settings, we can get

$$R(T) = O\left(\sqrt{T}\right).$$

In the online setting, regret grows naturally with time in a way that is very different from loss in the batch setting. If we look at the definition of regret, at each timestep it's adding a new component

$$\ell(\hat{y}_t, y_t) - \ell(h_w(x_t), y_t)$$

which tends to be positive for an optimally-chosen w. For this reason, we can't expect the regret to go to zero as time increases: the regret will be increasing with time, not decreasing. Instead, for online learning we generally want to get what's called **sublinear regret**: a regret that grows sublinearly with time. Equivalently, we can think about situations in which the *average regret*

$$\frac{R(T)}{T} = \frac{1}{T} \sum_{t=1}^{T} \ell(\hat{y}_t, y_t) - \inf_w \frac{1}{T} \sum_{t=1}^{T} \ell(h_w(x_t), y_t)$$

goes to zero. We can see that this happens in the case of online gradient descent, where

$$R(T) = O(\sqrt{T}) = o(T).$$

Making online learning scalable. Most of the techniques we've discussed in class are readily applicable to the online learning setting. For example, we can easily define minibatch versions of online gradient descent, use adaptive learning rate schemes, and even use hardware techniques like parallelism and low precision. If you're interested in more details about how to do this, there are a lot of papers in the literature. Many online-setting variants of SGD are subject to ongoing active research, particularly the question of how we should build end-to-end systems and frameworks to support online learning.

Large Language Models as Foundation Models.

A "foundation model" is a large general-purpose model that can be adapated to a wide variety of downstream tasks. It's "foundation" in the sense of being a support for future development.

Language model examples:

- BERT (an encoder-decoder model transformer)
- Llama (open-source transformer)
- GPT-k (basis for ChatGPT)

Vision examples:

- DALL-E (image generation)
- CLIP (multimodal)

How to learn with a foundation model.

In-context learning.

- doesn't require any training
- just pass input/output pairs to the language model and allow it to "learn" on the fly as it generates
- no SGD, ADAM, or AdaGrad

What might be some upsides and downsides of in-context learning?

Instruction tuning

- fine-tune all the weights of a foundation model to respond well to human instructions
- train an LLMs on a dataset of (INSTRUCTION, OUTPUT) pairs

- a variant is RLHF: Reinforcement Learning from Human Feedback
- usually produces a model that's better at general downstream tasks when provided an interpretable instruction
- **prompt engineering** is a big part of working with these models

Task-specific fine tuning

- run more gradient steps on the foundation model on your task's dataset
- this is basically classic transfer learning

What are the trade-offs of fine tuning?

Parameter-efficient fine tuning (PEFT).

Main idea: fine-tuning a large language model is expensive. But **why is it expensive**?

Instead, fix most of the parameters (set their requiresgrad to false) and update only a small subset of the weights, or some small "adapter" network added on.

- Prompt tuning: use gradients to update the **prompt embedding** vectors at the input
  - like learning a "magic word" that corresponds to your task
- Prefix tuning: use gradients to update the **prompt embedding** key-value vectors at each layer
- Low-rank adapters (LoRA): add a low-rank "adapter" AB^T to every linear layer, and only fine-tune the adapter