Lecture 28: Online Learning. Recap. The future of ML?

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 w, that is

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

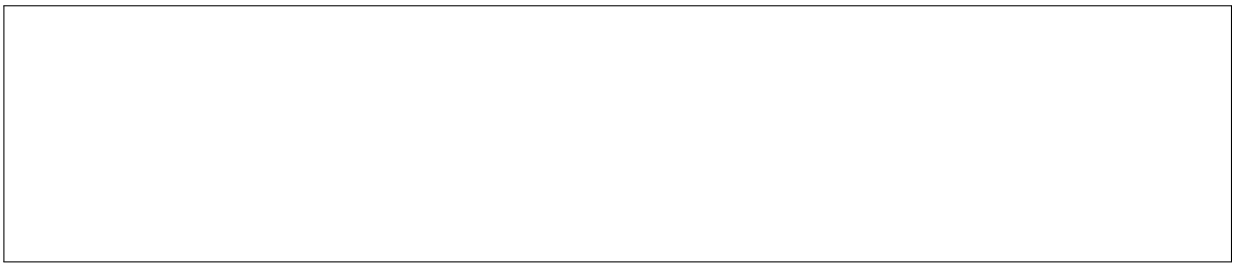
[1] Sometimes in practice, the learner does not get the label y immediately but rather a short time later.

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.

Summary and open questions.

Scaling machine learning methods is increasingly important. In this course, we addressed the high-level question: **What principles underlie the methods that allow us to scale machine learning?** To answer this question, we used techniques from three broad areas: statistics, optimization, and systems. We articulated three broad principles, one in each area.

- **Statistics Principle: Make it easier to process a large dataset by processing a small random subsample instead.**
- **Optimization Principle: Write your learning task as an optimization problem, and solve it via fast general algorithms that update the model iteratively.**
- **Systems Principle: Use algorithms that fit your hardware, and use hardware that fits your algorithms.**

We covered many techniques in this class...**but there are lots of open questions left!**

▷ Open question: **Is scaling really all we need for ML**?

- Recent trend is to run bigger and bigger models!
- e.g. GPT-3 is a language model that has 175 billion parameters
- We can use these large models for zero-shot learning, and this often outperforms non-transfer-learning approaches
- Performance of these models seems to improve further with size, following so-called "scaling laws"
- Is scaling up the size of modern transformers where we should expect to see the most gains? Should we devote most of our resources to this?
- What is the right way to fine-tune foundation models for a target task?
  - Fine-tune all the weights
  - Fine-tune a prompt
  - Fine-tune some smaller subset of the weights
  - ...or learn a new network alongside the main one

▷ Open problem: **reproducibility and debugging of machine learning systems**.

- Most of the algorithms we discussed in class are randomized, and random algorithms are hard to reproduce.
- Even when we don't use explicitly randomized methods, floating point imprecision can still make results difficult to reproduce exactly.
  - For hardware efficiency, the compiler loves to reorder floating point operations (this is sometimes called fast math mode) which can introduce slight differences in the output of an ML system.

  - As a result, even running the same learning algorithm on the same data on different ML frameworks *can* result in different learned models!

- Reproducibility is also made more challenging when hyperparameter optimization is used.
  - Unless you have the random seed, it's impossible to reproduce someone else's random search.
  - Hyperparameter optimization provides lots of opportunity for (possibly unintentional) cheating, where the test set is used improperly.
- ML models are difficult to debug because they often **learn around bugs**.

▷ Open problem: **more scalable distributed machine learning**.

- Distributed machine learning has this fundamental tradeoff with the batch size.
  - Larger batch size good for systems because there's more parallelism.
  - Smaller batch size good for statistics because we can make more "progress" per gradient sample. (For the same reason that SGD is generally better than gradient descent.)
- Communication among workers outside a pod can have high latency in distributed learning.
- The datacenters of the future will likely have many heterogeneous workers available.
  - How can we best distribute a learning workload across heterogeneous workers?
- When running many workers in parallel, the performance will start to be bound by **stragglers**, workers that take longer to work than their counterparts. How can we deal with this while still retaining performance guarantees?

▷ Open problem: **robustness to adversarial examples**.

- It's easy to construct examples that fool a deep neural network.
- How can we make our scalable ML methods provably robust to these type of attacks?

Thank you, and please submit course evaluations!