# Lecture 25: Online Learning. Multimodal learning. Tokenization.

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

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.

In particular, if our algorithm has low regret then it must also perform well on the underlying population distribution (since otherwise there would exist some good predictor with small expected loss).

**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.

**Multimodal Learning.** Multimodal learning refers generally to any situation in which data comes in multiple forms or modalities, but most multimodal models are VLMs (vision language models) For instance:

- In computer vision, we might want to input an image along with a prompt that asks questions about that image.
- In speech recognition, audio signals might come with text that summarizes what was said and who is speaking; this text may make it easier for models to produce a good transcript of the speech. Or, we might want to ask a question about the speech audio that can't be answered from the transcript alone.

One simple approach to multimodal learning is to learn separate models for each modality, then combine their predictions using some lightweight head model. However, this can be suboptimal if we want the model to be able to reason about what it sees.

A more sophisticated approach is to use an encoding model that maps images to tokens instead, meaning that we feed all of the modalities into a single neural network. The image-as-tokens is then treated like we would treat tokens coming from text. For example, consider a model that takes both images and text as input, in order to predict the objects present in the image. If we use a vision encoder, then our neural network can learn to identify objects in the image and also use the text to inform its predictions and to follow instructions.

**Byte-Pair Encoding.** In language processing tasks, one fundamental problem is how to represent strings of characters as tokens in a way that language models can efficiently learn from. There are many approaches to tokenization; here we will focus on **byte-pair encoding**, or BPE, which is by far the most popular approach used today. BPE was originally developed for text compression but has since been widely adopted in NLP.

The basic idea behind BPE is simple: given a large corpus of text data, we want to find a set of subwords (i.e. pieces of words) that can be combined to form any word in the corpus. We start with a vocabulary containing only individual characters. Then at each step, we look for the pair of symbols $a$ and $b$ such that the concatenation $ab$ appears most frequently in the corpus. We add $ab$ to the vocabulary, and replace every occurrence of $ab$ in the corpus with a new symbol $c$. This process continues until the vocabulary contains a specified number of symbols (what is the systems tradeoff here?). To tokenize a new sequence, we rerun these replacement rules in order on the new text.

One advantage of BPE is that it can handle out-of-vocabulary words that do not appear in the training data: it will represent them as a sequence of subwords. Since BPE represents each word as a sequence of subwords, and all possible single bytes are part of the vocabulary, we can represent any string using only the subwords in our vocabulary. This makes it especially useful for tasks such as language translation or text generation, where we need to be able to generate text that may contain words not seen during training. (It also means that not all token sequences can actually be the result of encoding a string!)