Lecture 16: Beyond simple supervised learning.

CS4787/5777 — Principles of Large-Scale ML Systems

Spill-over from last time: residual connections. ResNets in torch.

Recall. The standard supervised learning setup starts with a dataset $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \subset \mathcal{X} \times \mathcal{Y}$ and then typically minimizes the empirical risk

$$\ell(w) = \frac{1}{n}\sum_{i=1}^{n} \ell(w; x_i, y_i) = \frac{1}{n}\sum_{i=1}^{n} \text{loss}(h_w(x_i); y_i) + \lambda \cdot \text{regularizer}(w).$$

You might also remember *unsupervised learning* from your previous ML class. This refers to the case where there is no label and our dataset is just $\{x_1, x_2, \ldots, x_n\}$. But there are many other things we can do beyond this simple supervised learning setup.

Data Augmentation. Problem: sometimes our data set is too small to train the model we want.

Solution: just "make up" more data. Transform examples from the dataset to produce new ones, effectively increasing the size of the dataset. Concretely, let $\mathcal{A}$ be a random function from the example space to itself (i.e. from $\mathcal{X}$ to $\mathcal{X}$); we can equivalently think of $\mathcal{A}$ as mapping from $\mathcal{X} \times [0, 1]$ to $\mathcal{X}$, where the unit interval $[0, 1]$ denotes a random number that's used to seed the transformation. We then consider the following *data-augmented SGD update*:

$$w_{t+1} = w_t - \alpha_t \nabla \ell(w_t; \mathcal{A}(x_t; \eta_t), y_t),$$

where (x_t, y_t) denotes the example from $\mathcal{D}$ chosen for use in the tth step (e.g. by random reshuffling) and $\eta_t \in [0, 1]$ is an independently selected random seed for the augmentation function. That is, DA randomly transforms each example before using it! Note that we can combine this with pretty much everything we've done so far: SGD, Momentum, Adam, etc.

Key idea: Need to have the transformation tend to preserve the class. Otherwise, we're going to be generating nonsense: examples which are labeled incorrectly.

What are some transformations we could use in data augmentation? When might they be a bad idea?

Data augmentation can be seen as a type of regularization where we try to constrain the model to be approximately invariant to a group action. Data augmentation in PyTorch can be using the `transforms` module: makes it really easy to implement.

Semi-Supervised Learning. Motivation: data is cheap; labels are expensive. *Do we need to label it all?*

Idea: combine a large amount of unlabeled data with a small amount of labeled data. *Basically a fusion between supervised and unsupervised learning.* Common assumptions:

- Smoothness assumption: if two points x_1 and x_2 are close, their label is also close. *Q: What classic supervised learning algorithm makes the same assumption?*
- Clustering assumption: the data is split into clusters; if two points x_1 and x_2 share a cluster, their label is the same. *Q: How might we use this to learn?*
- Manifold assumption: the data lies on a low-dimensional manifold. *Q: How might we use this to learn?*

Approaches to semi-supervised learning:

- Generative modeling
- Self-labeling: repeatedly train a supervised learning method and then use it to label more unlabeled data

When might semi-supervised learning work well? When might it be a bad idea?

Weak supervision. Similar to semi-supervised learning, except the labels themselves are noisy or imprecise. Sources of noisy labels:

- Crowdsourcing from non-experts
- Data programming: use functions to heuristically label examples

When might weak supervision work well? When might it be a bad idea?

Self-supervised learning. Extract a supervision signal from the data itself, usually leveraging domain-specific structure we know the data has.

Example: fill-in-the-blanks for computer vision. Take a image. Remove patches from the image. Train a DNN to recover the original image from the version with the patches removed. Then use part of this network as an initialization for a downstream supervised task.

Transfer learning. Train a model on one task. Apply it to another task.

When might transfer learning be useful? When might it be a bad idea?