ECE-GY 6143, Spring 2020

Lecture 11: Generalization, Intro to Unsupervised Learning

Generalization

Modern machine learning models (such as neural nets) typically have lots of parameters. For example, the best architecture as of the end of 2019 for object recognition, measured using the popular ImageNet benchmark dataset, contains *928 million* learnable parameters. This is far greater than the number of training samples (about 14 million).

This, of course, should be concerning from a *model generalization* standpoint. Recall that we had previously discussed the bias-versus variance issue in ML models. As the number of parameters increases, the bias decreases (because we have more tunable knobs to fit our data), but the variance increases (because there is greater amount of overfitting). We get a curve like this:

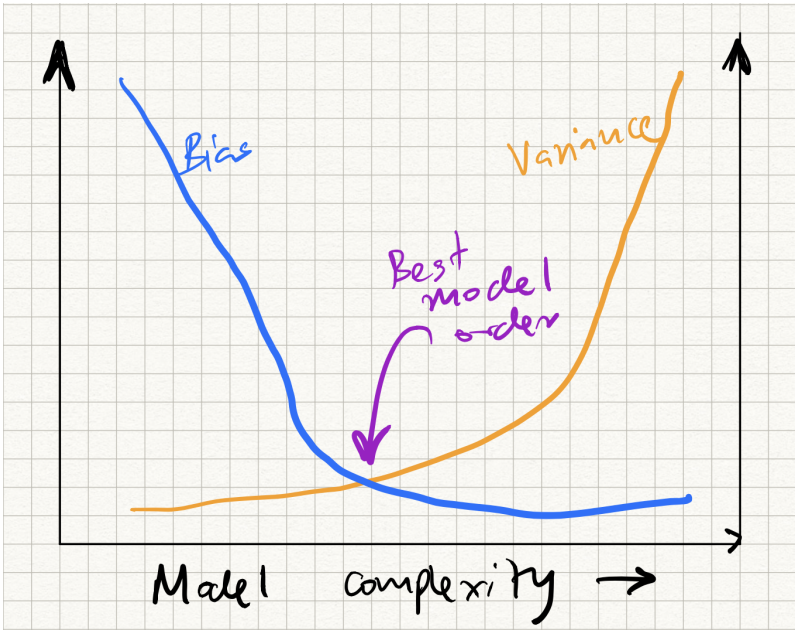

Figure 1: Bias-variance tradeoff

The question, then, is how to control the position on this curve so that we hit the sweet spot in the middle?

(It turns out that deep neural nets don't quite obey this curve – a puzzling phenomenon called "double descent" occurs, but let us not dive too deep into it here; Google it if you are interested.)

We had previously introduced *regularizers* as a way to mitigate shortage of training samples, and in neural nets a broader class of regularization strategies exist. Simple techniques such as adding an extra regularizer do not work in isolation, and a few extra tricks are required. These include:

- *Designing "bottlenecks" in network architecture*: One way to regularize performance is to add a *linear* layer of neurons that is "narrower" than the layer preceding and succeeding it. We will talk about unsupervised learning below, and interpret this in the context of PCA.
- *Early stopping*: We monitor our training and validation error curves during the learning process. We expect training error to (generally) decrease, and validation error to flatten out (or even start increasing). The gap between the two curves indicates generalization performance, and we stop training when this gap is minimized. The problem with this approach is that if we train using *stochastic* methods (such as SGD) the error curves can fluctuate quite a bit and early stopping may lead to sub-optimal results.
- *Weight decay*: This involves adding an L2-regularizer to the standard neural net training loss.
- *Dataset augmentation*: To resolve the imbalance between the number of model parameters and the number of training examples, we can try to increase dataset size. One way to simulate an increased dataset size is by artificially transforming an input training sample (e.g. if we are dealing with images, we can shift, rotate, flip, crop, shear, or distort the training image example) before using it to update the model weights. There are smarter ways of doing dataset augmentation depending on the application, and libraries such as PyTorch have inbuilt routines for doing this.
- *Dropout*: A different way to solve the above imbalance is to simulate a smaller model. This can be (heuristically) achieved via a technique called dropout. The idea is that at the start of each iteration, we introduce stochastic binary variables called *masks* – m_i – for each neuron. These random variables are 1 with probability p and 0 with probability $1 - p$. Then, in the forward pass, we "drop" individual neurons by masking their activations:

  $$h_i = m_i \phi(z_i)$$

  and in the backward pass, we similarly mask the corresponding gradients with m_i:

  $$\partial_{z_i} \mathcal{L} = \partial_{h_i} \mathcal{L} \cdot m_i \cdot \partial_{z_i} \phi(z_i).$$

  During test time, all weights are scaled with p in order to match expected values.
- *Transfer learning*: Yet another way to resolve the issue of small training datasets is to *transfer* knowledge from a different learning problem. For example, suppose we are given the task of learning a classifier for medical images (say, CT or X-Ray). Medical images are expensive to obtain, and training very deep neural networks on realistic dataset sizes may be infeasible. However, we could first learn a different neural network for a different task (say, standard object recognition) and *finetune* this network with the given available medical image data. The high level idea is that there may be common learned features across datasets, and we may not depend on re-learning all of them for each new task.

Unsupervised learning

We will now transition to a different class of data analysis techniques, broadly classified as *unsupervised learning*.

Until now, we have focused on problems of the following sort: given a series of examples of data points x and labels y, figure out a function f that maps x to y. The hope is that if this function is

"learned" well enough, then maybe we can use f for predicting the output for a new, unseen data point x^*.

The procedure is *supervised* in the sense that a *teacher* (nature, or some measurement mechanism, or a human being) gives you the labels/outputs y corresponding to the (training) data points.

Unsupervised learning works differently. Suppose I gave you 200 *unlabeled* car images – 100 images each of two car models that you have never seen before. (So imagine images corresponding to data points x_i, but there is no label information y_i tagged to each image.) I give you no information other than the raw image data points: a priori, you wouldn't know how many classes there are, whether or not there are exactly 2 models, or even whether they correspond to cars.

Suppose you looked over all 200 car images. Now, given an entirely new unseen image of one of these two car models (perhaps taken from a slightly different viewing angle, or a different color), can you *still* predict which model the new example belongs to?

A moment of reflection should lead you to believe: *yes*! By looking at the 200 images, your eyes (typically) should be able to figure out not only that there are two models, but also extrapolate to unseen examples. Observe that you were able to do this in a fully unsupervised manner, with no label information provided.

The reason why you were able to do this is due to the remarkable ability of biological systems to do unsupervised learning: it is natural for us to visually observe associations in data even without supervision. In the above example, we mentally grouped all 100 images of the first class into one category and all images of the second class into another category. We also (automatically) extracted salient features of each car model, so that when presented with a new example we can map it to one of the two categories.

Of course, the above example involves images, and our brain's ability to process image data is excellent (since the visual systems in our brain are very well tuned.) What if I changed the example to say, DNA fragments? If I gave you 200 genetic signatures, 100 corresponding to Species 1 and 100 others to Species 2, would you still be able to automatically classify them? Now, it is no longer clear – unless you are well-trained to detect patterns in DNA.

To many, this is the next big challenge in machine learning: automatically deducing (in a wholly unsupervised manner) interesting features in data.

Applications of unsupervised learning

Unsupervised learning algorithms are useful for:

- visualizing high dimensional datasets.
- finding common patterns in data.
- reducing the dimensionality of datasets to something more manageable.
- finding correlations among different features.

etc.

Principal Components Analysis

A key primitive in unsupervised learning is the notion of *principal components*, which is really a direct application of SVD.

Suppose a d-dimensional dataset (without labels) $\{x_1, x_2, \ldots, x_n\}$ is given. For simplicity, assume that the data is of zero mean. (If not, we can always just subtract the mean from each data point and center it around the origin.)

Imagine stacking up the data points into rows of an $n \times d$ matrix X. Our goal is to figure out interesting correlations within the data in an unsupervised manner. For example, if two features were somehow correlated (say the first two co-ordinates of each data point were proportional to each other) then it makes sense to identify such correlations automatically and maybe even discard one of them since there is no new information. From a geometric standpoint, we need to figure out the directions of *variation* in the data.

Mathematically, we can do the same trick that we have been doing before: consider a line represented by a parameter vector $w \in \mathbb{R}^d$ (normalized such that $\|w\| = 1$). In that case, the projections onto each data point have the expression $|\langle x_i, w\rangle|$. Therefore, the sample *variance* of the lengths of the projections is given by:

$$\begin{aligned} S(w) &= \frac{1}{n}\sum_{i=1}^{n} |\langle x_i, w\rangle|^2 \\ &= \frac{1}{n}\|Xw\|^2 \\ &= \frac{1}{n} w^T X^T X w. \end{aligned}$$

We need to figure out the direction w corresponding to maximum variance – subject to the constraint that $\|w\| = 1$. (The normalizing constraint is important; otherwise, we could arbitrarily scale up any w to get infinitely large $S(w)$.)

One can observe that the quantity

$$\frac{1}{n} X^T X = \frac{1}{n}\sum_{i=1}^{n} x_i x_i^T := \widehat{\Sigma}$$

which is a $d \times d$ matrix calleld the empirical *covariance matrix* of the data. So really we are asking for the top eigenvector of the data covariance:

$$\widehat{\Sigma} = \sum_{j=1}^{d} \lambda_j v_j v_j^T.$$

Recall that the top eigenvector is just the vector corresponding to the maximum among all the λ_j, but since the λ_i are arranged in decreasing order, the maximum corresponds to λ_1, and hence the top eigenvector is the vector $w = v_1$. This is called the *first principal component direction* of X.

Notice that this direction is a d-dimensional vector. We can define the *projection* of the data along with vector; this gives us the first *principal component score* of the data:

$$p_1 = Xw = Xv_1 = \left[x_1^T v_1; x_2^T v_1; \ldots x_n^T v_1\right]$$

We can iteratively do this for $v_2, v_3, \ldots$ and these give us successive new (orthogonal) principal components of the data.

This gives us a fairly simple way to visualize high dimensional data – simply compute the first 2 (or 3) principal component scores and plot them! Often, interesting patterns in data which cannot be obviously visualized can be discovered via PCA. PCA is a core routine in most ML software packages, but keep in mind the above principles.

Because of the orthonormality of the principal component directions $(v_1, \ldots, v_n)$, they form a new *basis* for the data points. In other words, each data point can be *reconstructed* as:

$$x_i \approx \sum_{j=1}^{n} p_{ij} v_j = \sum_{j=1}^{n} \langle x_i, v_j \rangle v_j.$$

If we truncate the above basis summation after k terms, then we get a *best-k* term reconstruction of the data (where "best" is measured in terms of explained variance in the data).

Drawbacks of PCA

PCA is well-established and some version of PCA has been around for close to a century now. However, there are several other techniques for exploratory data analysis with all kinds of acronyms - CCA, LDA, NMF, ICA, etc - which we won't cover in great detail here.

One drawback of PCA is that the principal directions and/or scores are not particularly interpretable. Imagine, for example, computing the principal components of a collection of *images*. Image data are typically positive-valued data since they represent light intensity values. If we interpret the principal directions as "building blocks" of the data, then each data point (image) can be viewed as a synthesis of some combination of these principal directions.

However, eigenvectors are orthogonal by definition, and therefore all their coordinates cannot be all positive. In other words, we are trying to represent positive-valued data (images) in terms of building blocks that are not necessarily image-like. Other techniques such as non-negative matrix factorization (NMF) helps mitigate this issue to some extent.

PCA in Practice

Some more guidelines on performing PCA:

- Center your data. Compute the mean (average) data point and subtract from each point.
- Normalize features by dividing the (centered) data by feature variances (these are called z-scores).
- Beware of outliers. These include anomalous data points or corrupted features, and can arise due to measurement error or some other source of data corruption. PCA is notoriously susceptible to corruptions/outliers in the data (there are techniques such as outlier-resistant *robust* PCA which will be discussed later.)
- Beware of missing data. PCA does *not* work very well if a large fraction of the observed data happen to be missing. (There are techniques to deal with this approach called *matrix completion*.)

- Keep in mind that PCA is a technique to reduce dimensionality from d to some more manageable number k. Choose k wisely! For plotting/visualization purposes, k cannot be bigger than 2 or 3.
- One good guideline to choose k is to plot the eigenvalues of the covariance matrix λ_j, and see how they decay. The number of *big* singular values (where *big* is not particularly well-defined) should give you an idea of what k should be.

Unsupervised learning and neural networks

PCA can be viewed as a very simple form of something called a "linear auto-encoder".

To see why this is the case, consider a data matrix X and perform a PCA decomposition for k components. The principal component scores, $\{Xv_1, Xv_2, \ldots, Xv_k\}$ can be stacked into a $n \times k$ matrix $XW := Z$, where $W = [v_1, \ldots v_k]$ is orthonormal. The calculation of the scores can be viewed as an "encoding" of the data. The best k-term reconstruction of X (in terms of explained variance) is given by:

$$X^{(k)} = ZW^T = XWW^T.$$

This can be viewed as the "decoding" of the data from the PCA scores.

We can view this operation layer-wise as a two-layer neural network (with identity activation functions), where the first layer implements the encoding (multiplication with W), and the second layer implements the decoding (multiplication with \$W^T). There is weight-sharing going on here, since the first and second layer weight matrices are transposes of each other.

In this manner, one can view PCA as a form of learning the weights of a shallow, *linear*, neural network whose purpose is to reconstruct each data point. The loss used to learn this network is the (sample) variance captured, which can be written in terms of the ℓ_2-loss.

(There is an extra orthonormality constraint in the weights which can be viewed as an additional form of regularization.)

We can imagine extending this notion of an "auto-encoder" to more complex architectures – introducing nonlinearities, additional layers, etc. Autoencoders have been used as unsupervised pre-trained modules in larger neural nets; like PCA, they capture the main directions of data points – even ones that are unlabeled – and can reduce the burden of training very large networks.