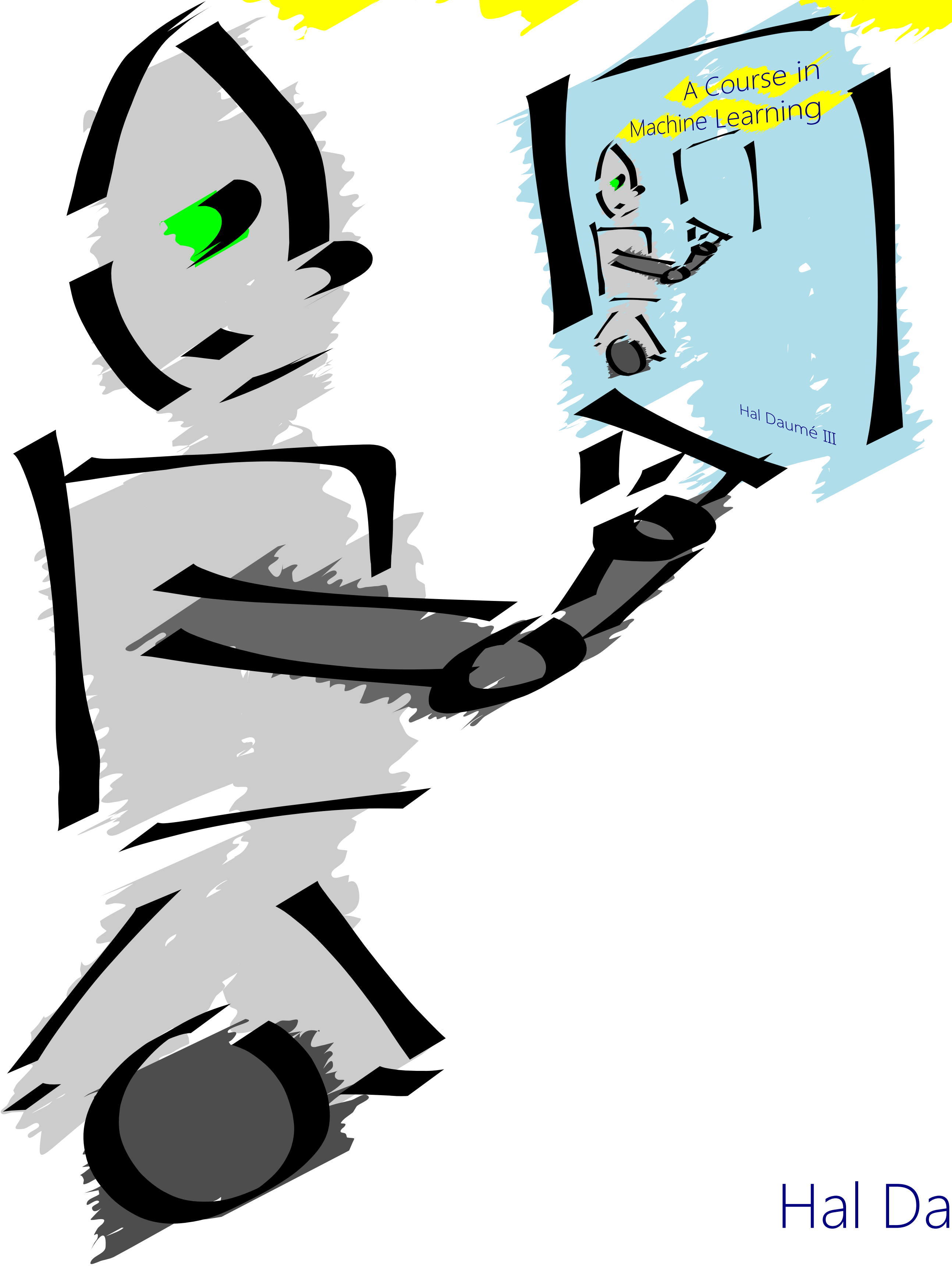
A Course in Machine Learning



Hal Daumé III

12 | Efficient Learning

–

Learning Objectives:

- Understand and be able to implement stochastic gradient descent algorithms.
- Compare and contrast small versus large batch sizes in stochastic optimization.
- Derive subgradients for sparse regularizers.
- Implement feature hashing.

Dependencies:

So far, our focus has been on *models* of learning and basic algorithms for those models. We have not placed much emphasis on how to learn *quickly*. The basic techniques you learned about so far are enough to get learning algorithms running on tens or hundreds of thousands of examples. But if you want to build an algorithm for web page ranking, you will need to deal with millions or billions of examples, in hundreds of thousands of dimensions. The basic approaches you have seen so far are insufficient to achieve such a massive scale.

In this chapter, you will learn some techniques for scaling learning algorithms. This are useful even when you do not have billions of training examples, because it's always nice to have a program that runs quickly. You will see techniques for speeding up both model training and model prediction. The focus in this chapter is on linear models (for simplicity), but most of what you will learn applies more generally.

12.1 What Does it Mean to be Fast?

Everyone always wants fast algorithms. In the context of machine learning, this can mean many things. You might want fast training algorithms, or perhaps training algorithms that scale to very large data sets (for instance, ones that will not fit in main memory). You might want training algorithms that can be easily parallelized. Or, you might not care about training efficiency, since it is an offline process, and only care about how quickly your learned functions can make classification decisions.

It is important to separate out these desires. If you care about efficiency at training time, then what you are really asking for are more efficient learning algorithms. On the other hand, if you care about efficiency at test time, then you are asking for *models* that can be quickly evaluated.

One issue that is not covered in this chapter is parallel learning.

This is largely because it is currently not a well-understood area in machine learning. There are many aspects of parallelism that come into play, such as the speed of communication across the network, whether you have shared memory, etc. Right now, this the general, poor-man's approach to parallelization, is to employ ensembles.

12.2 Stochastic Optimization

During training of most learning algorithms, you consider the entire data set simultaneously. This is certainly true of gradient descent algorithms for regularized linear classifiers (recall Algorithm 6.4), in which you first compute a gradient over the entire training data (for simplicity, consider the unbiased case):

$$\boldsymbol{g} = \sum_n \nabla_{\boldsymbol{w}} \ell(y_n, \boldsymbol{w} \cdot \boldsymbol{x}_n) + \lambda \boldsymbol{w} \tag{12.1}$$

where $\ell(y, \hat{y})$ is some loss function. Then you update the weights by $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \boldsymbol{g}$. In this algorithm, in order to make a *single* update, you have to look at *every* training example.

When there are billions of training examples, it is a bit silly to look at every one before doing anything. Perhaps just on the basis of the first few examples, you can already start learning something!

Stochastic optimization involves thinking of your training data as a big distribution over examples. A draw from this distribution corresponds to picking some example (uniformly at random) from your data set. Viewed this way, the optimization problem becomes a **stochastic optimization** problem, because you are trying to optimize some function (say, a regularized linear classifier) over a probability distribution. You can derive this intepretation directly as follows:

$$\boldsymbol{w}^* = \arg\max_{\boldsymbol{w}} \sum_n \ell(y_n, \boldsymbol{w} \cdot \boldsymbol{x}_n) + R(\boldsymbol{w}) \qquad \text{definition} \tag{12.2}$$

$$= \arg\max_{\boldsymbol{w}} \sum_n \left[\ell(y_n, \boldsymbol{w} \cdot \boldsymbol{x}_n) + \frac{1}{N} R(\boldsymbol{w}) \right] \qquad \text{move } R \text{ inside sum} \tag{12.3}$$

$$= \arg\max_{\boldsymbol{w}} \sum_n \left[\frac{1}{N} \ell(y_n, \boldsymbol{w} \cdot \boldsymbol{x}_n) + \frac{1}{N^2} R(\boldsymbol{w}) \right] \qquad \text{divide through by } N \tag{12.4}$$

$$= \arg\max_{\boldsymbol{w}} \mathbb{E}_{(y,\boldsymbol{x}) \sim D} \left[\ell(y, \boldsymbol{w} \cdot \boldsymbol{x}) + \frac{1}{N} R(\boldsymbol{w}) \right] \qquad \text{write as expectation} \tag{12.5}$$

$$\text{where } D \text{ is the training data distribution} \tag{12.6}$$

Given this framework, you have the following general form of an

```
Algorithm 33 StochasticGradientDescent($\mathcal{F}$, $\mathcal{D}$, $S$, $K$, $\eta_1$, $\dots$)
$\boldsymbol{z}^{(0)} \leftarrow \langle 0, 0, \dots, 0 \rangle$                    // initialize variable we are optimizing
for $k = 1 \dots K$ do
    $D^{(k)} \leftarrow$ $S$-many random data points from $\mathcal{D}$
    $\boldsymbol{g}^{(k)} \leftarrow \nabla_{\boldsymbol{z}} \mathcal{F}(D^{(k)}) \big|_{\boldsymbol{z}^{(k-1)}}$          // compute gradient on sample
    $\boldsymbol{z}^{(k)} \leftarrow \boldsymbol{z}^{(k-1)} - \eta^{(k)} \boldsymbol{g}^{(k)}$          // take a step down the gradient
end for
return $\boldsymbol{z}^{(K)}$
```

optimization problem:

$$\min_{\boldsymbol{z}} \quad \mathbb{E}_{\zeta}[\mathcal{F}(\boldsymbol{z}, \zeta)] \tag{12.7}$$

In the example, ζ denotes the random choice of examples over the dataset, $\boldsymbol{z}$ denotes the weight vector and $\mathcal{F}(\boldsymbol{w}, \zeta)$ denotes the loss on that example *plus* a fraction of the regularizer.

Stochastic optimization problems are formally *harder* than regular (deterministic) optimization problems because you do not even get access to exact function values and gradients. The only access you have to the function $\mathcal{F}$ that you wish to optimize are noisy measurements, governed by the distribution over ζ. Despite this lack of information, you can still run a gradient-based algorithm, where you simply compute *local* gradients on a current sample of data.

More precisely, you can draw a data point at random from your data set. This is analogous to drawing a single value ζ from its distribution. You can compute the gradient of $\mathcal{F}$ just at that point. In this case of a 2-norm regularized linear model, this is simply $\boldsymbol{g} = \nabla_{\boldsymbol{w}} \ell(y, \boldsymbol{w} \cdot \boldsymbol{x}) + \frac{1}{N}\boldsymbol{w}$, where $(y, \boldsymbol{x})$ is the random point you selected. Given this *estimate* of the gradient (it's an estimate because it's based on a single random draw), you can take a small gradient step $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \boldsymbol{g}$.

This is the **stochastic gradient descent** algorithm (**SGD**). In practice, taking gradients with respect to a *single* data point might be too myopic. In such cases, it is useful to use a small **batch** of data. Here, you can draw 10 random examples from the training data and compute a small gradient (estimate) based on those examples: $\boldsymbol{g} = \sum_{m=1}^{10} \nabla_{\boldsymbol{w}} \ell(y_m, \boldsymbol{w} \cdot \boldsymbol{x}_m) + \frac{10}{N}\boldsymbol{w}$, where you need to include 10 counts of the regularizer. Popular batch sizes are 1 (single points) and 10. The generic SGD algorithm is depicted in Algorithm 12.2, which takes K-many steps over batches of S-many examples.

In stochastic gradient descent, it is *imperative* to choose good step sizes. It is also very important that the steps get smaller over time at a reasonable slow rate. In particular, convergence can be guaranteed for learning rates of the form: $\eta^{(k)} = \frac{\eta_0}{\sqrt{k}}$, where η_0 is a fixed, initial step size, typically 0.01, 0.1 or 1 depending on how quickly you expect the algorithm to converge. Unfortunately, in comparisong to gradient descent, stochastic gradient is quite sensitive to the selection of a good learning rate.

There is one more practical issues related to the use of SGD as a learning algorithm: do you *really* select a random point (or subset of random points) at each step, or do you stream through the data in order. The answer is akin to the answer of the same question for the perceptron algorithm (Chapter 3). If you do not permute your data at all, very bad things can happen. If you *do* permute your data once and then do multiple passes over that same permutation, it will converge, but more slowly. In theory, you really should permute every iteration. If your data is small enough to fit in memory, this is not a big deal: you will only pay for cache misses. However, if your data is too large for memory and resides on a magnetic disk that has a slow seek time, randomly seeking to new data points for each example is prohibitivly slow, and you will likely need to forgo permuting the data. The speed hit in convergence speed will almost certainly be recovered by the speed gain in not having to seek on disk routinely. (Note that the story is very different for solid state disks, on which random accesses really are quite efficient.)

12.3 Sparse Regularization

For many learning algorithms, the test-time efficiency is governed by how many features are used for prediction. This is one reason decision trees tend to be among the fastest predictors: they only use a small number of features. Especially in cases where the actual *computation* of these features is expensive, cutting down on the number that are used at test time can yield huge gains in efficiency. Moreover, the amount of memory used to make predictions is also typically governed by the number of features. (Note: this is *not* true of kernel methods like support vector machines, in which the dominant cost is the number of support vectors.) Furthermore, you may simply *believe* that your learning problem can be solved with a very small number of features: this is a very reasonable form of inductive bias.

This is the idea behind sparse models, and in particular, sparse regularizers. One of the disadvantages of a 2-norm regularizer for linear models is that they tend to never produce weights that are *exactly* zero. They get close to zero, but never hit it. To understand why, as a weight w_d approaches zero, its gradient *also* approaches zero. Thus, even if the weight *should* be zero, it will essentially never get there because of the constantly shrinking gradient.

This suggests that an alternative regularizer is required to yield a sparse inductive bias. An ideal case would be the zero-norm regularizer, which simply counts the number of non-zero values in a vector: $||\boldsymbol{w}||_0 = \sum_d [w_d \neq 0]$. If you could minimize this regularizer, you would be explicitly minimizing the number of non-zero features. Unfortunately, not only is the zero-norm non-convex, it's also discrete. Optimizing it is NP-hard.

A reasonable middle-ground is the one-norm: $||\boldsymbol{w}||_1 = \sum_d |w_d|$. It is indeed convex: in fact, it is the tighest ℓ_p norm that *is* convex. Moreover, its gradients do not go to zero as in the two-norm. Just as hinge-loss is the tightest convex upper bound on zero-one error, the one-norm is the tighest convex upper bound on the zero-norm.

At this point, you should be content. You can take your subgradient optimizer for arbitrary functions and plug in the one-norm as a regularizer. The one-norm is surely non-differentiable at $w_d = 0$, but you can simply choose any value in the range $[-1, +1]$ as a subgradient at that point. (You should choose zero.)

Unfortunately, this does not quite work the way you might expect. The issue is that the gradient might "overstep" zero and you will never end up with a solution that is particularly sparse. For example, at the end of one gradient step, you might have $w_3 = 0.6$. Your gradient might have $g_6 = 0.8$ and your gradient step (assuming $\eta = 1$) will update so that the new $w_3 = -0.2$. In the subsequent iteration, you might have $g_6 = -0.3$ and step to $w_3 = 0.1$.

This observation leads to the idea of **trucated gradients**. The idea is simple: if you have a gradient that would step you over $w_d = 0$, then just set $w_d = 0$. In the easy case when the learning rate is 1, this means that if the *sign* of $w_d - g_d$ is different than the sign of w_d then you truncate the gradient step and simply set $w_d = 0$. In other words, g_d should never be *larger* than w_d Once you incorporate learning rates, you can express this as:

$$g_d \leftarrow \begin{cases} g_d & \text{if } w_d > 0 \text{ and } g_d \leq \frac{1}{\eta^{(k)}} w_d \\ g_d & \text{if } w_d < 0 \text{ and } g_d \geq \frac{1}{\eta^{(k)}} w_d \\ 0 & \text{otherwise} \end{cases} \tag{12.8}$$

This works quite well in the case of subgradient descent. It works somewhat less well in the case of *stochastic* subgradient descent. The problem that arises in the stochastic case is that wherever you choose to stop optimizing, you will have just touched a single example (or small batch of examples), which will increase the weights for a lot of features, before the regularizer "has time" to shrink them back down to zero. You will still end up with somewhat sparse solutions, but not as sparse as they could be. There are algorithms for dealing with this situation, but they all have a heuristic flavor to them and are beyond the scope of this book.

12.4 Feature Hashing

As much as speed is a bottleneck in prediction, so often is memory usage. If you have a very large number of features, the amount of memory that it takes to store weights for all of them can become prohibitive, especially if you wish to run your algorithm on small devices. Feature hashing is an incredibly simple technique for reducing the memory footprint of linear models, with very small sacrifices in accuracy.

The basic idea is to replace all of your features with hashed versions of those features, thus reducing your space from D-many feature weights to P-many feature weights, where P is the range of the hash function. You can actually think of hashing as a (randomized) feature mapping $\phi : \mathbb{R}^D \to \mathbb{R}^P$, for some $P \ll D$. The idea is as follows. First, you choose a hash function h whose domain is $[D] = \{1, 2, \dots, D\}$ and whose range is $[P]$. Then, when you receive a feature vector $\boldsymbol{x} \in \mathbb{R}^D$, you map it to a shorter feature vector $\hat{\boldsymbol{x}} \in \mathbb{R}^P$. Algorithmically, you can think of this mapping as follows:

1. Initialize $\hat{\boldsymbol{x}} = \langle 0, 0, \dots, 0 \rangle$
2. For each $d = 1 \dots D$:
    (a) Hash d to position $p = h(d)$
    (b) Update the pth position by adding x_d: $\hat{x}_p \leftarrow \hat{\boldsymbol{x}}_p + x_d$
3. Return $\hat{\boldsymbol{x}}$

Mathematically, the mapping looks like:

$$\phi(\boldsymbol{x})_p = \sum_d [h(d) = p] x_d \quad = \sum_{d \in h^{-1}(p)} x_d \tag{12.9}$$

where $h^{-1}(p) = \{d : h(d) = p\}$.

In the (unrealistic) case where $P = D$ and h simply encodes a permutation, then this mapping does not change the learning problem at all. All it does is rename all of the features. In practice, $P \ll D$ and there will be collisions. In this context, a collision means that two features, which are really different, end up looking the same to the learning algorithm. For instance, "is it sunny today?" and "did my favorite sports team win last night?" might get mapped to the same location after hashing. The hope is that the learning algorithm is sufficiently robust to noise that it can handle this case well.

Consider the kernel defined by this hash mapping. Namely:

$$
\begin{aligned}
K^{(\text{hash})}(\boldsymbol{x}, \boldsymbol{z}) &= \phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{z}) && (12.10)\\
&= \sum_p \left(\sum_d [h(d) = p] x_d \right) \left(\sum_d [h(d) = p] z_d \right) && (12.11)\\
&= \sum_p \sum_{d,e} [h(d) = p][h(e) = p] x_d z_e && (12.12)\\
&= \sum_d \sum_{e \in h^{-1}(h(d))} x_d z_e && (12.13)\\
&= \boldsymbol{x} \cdot \boldsymbol{z} + \sum_d \sum_{\substack{e \neq d,\\ e \in h^{-1}(h(d))}} x_d z_e && (12.14)
\end{aligned}
$$

This **hash kernel** has the form of a linear kernel plus a small number of quadratic terms. The particular quadratic terms are exactly those given by collisions of the hash function.

There are two things to notice about this. The first is that collisions might not actually be bad things! In a sense, they're giving you a little extra representational power. In particular, if the hash function happens to select out feature pairs that benefit from being paired, then you now have a better representation. The second is that even if this doesn't happen, the quadratic term in the kernel has only a small effect on the overall prediction. In particular, if you assume that your hash function is pairwise independent (a common assumption of hash functions), then the *expected value* of this quadratic term is zero, and its variance decreases at a rate of $\mathcal{O}(P^{-2})$. In other words, if you choose $P \approx 100$, then the variance is on the order of 0.0001.

12.5 Exercises

Exercise 12.1. TODO. . .