CS168: The Modern Algorithmic Toolbox
Lecture #6: Stochastic Gradient Descent and Regularization

Tim Roughgarden & Gregory Valiant*

April 13, 2016

1 Context

Last lecture we covered the basics of gradient descent, with an emphasis on the intuition behind and geometry underlying the method, plus a concrete instantiation of it for the problem of linear regression (fitting the best hyperplane to a set of data points). This basic method is already interesting and useful in its own right (see Homework #3).

This lecture we'll cover two extensions that, while simple, will bring your knowledge a step closer to the state-of-the-art in modern machine learning. The two extensions have different characters. The first concerns how to actually solve (computationally) a given unconstrained minimization problem, and gives a modification of basic gradient descent — "stochastic gradient descent" — that scales to much larger data sets. The second extension concerns problem formulation rather than implementation, namely the choice of the unconstrained optimization problem to solve (i.e., the objective function f). Here, we introduce the idea of "regularization," with the goal of avoiding overfitting the function learned to the data set at hand, even for very high-dimensional data.

2 Recap

Recall that an unconstrained minimization problem is defined by a function $f : \mathbb{R}^n \to \mathbb{R}$, and the goal is to compute the point $\mathbf{w} \in \mathbb{R}^n$ that minimizes this function. Recall the basic gradient descent method:

Gradient Descent (The Basic Method)

initialize $\mathbf{w} := \mathbf{w}_0$
while $\|\nabla f(\mathbf{w})\|_2 > \epsilon$ **do**

$$\mathbf{w} := \mathbf{w} - \alpha \cdot \nabla f(\mathbf{w}) \tag{1}$$

Recall that the parameter α is called the *step size* or *learning rate*. An alternative stopping rule (as seen in Homework #3) is to just run gradient descent for a fixed number of iterations and then return the final point.

In (1), both $\mathbf{w}$ and $\nabla f(\mathbf{w})$ are n-vectors, while α is a scalar. It's also worth zooming in to see what this update rule looks like in some coordinate, say the jth one:

$$w_j := w_j - \alpha \cdot \frac{\partial f}{\partial w_j}(\mathbf{w}). \tag{2}$$

The update (1) can be thought of as n updates of the form (2) being done in parallel (one per coordinate j).[1]

Recall the intuition behind the method. Gradient descent enters a contract with basic calculus. Calculus says that a function f differentiable at a point $\mathbf{w}$ can be locally well approximated by a linear function, namely $f(\mathbf{w}+\mathbf{z}) \approx f(\mathbf{w}) + \mathbf{z}^T \nabla f(\mathbf{w})$ for $\mathbf{z} \in \mathbb{R}^n$. This is analogous to drawing a tangent line to the graph of a univariate function (the $n = 1$ case). It's intuitively clear that the tangent line approximates the function well near the point of approximation, but not generally for faraway points. But a linear function $\mathbf{c}^T\mathbf{w}$ is easy to minimize — just move in the direction $-\mathbf{c}$, for a rate of decrease of $\|\mathbf{c}\|_2$. Combining these two facts motivates moving in the direction $-\nabla f(\mathbf{w})$, of "steepest descent," for a rate of decrease of $\|\nabla f(\mathbf{w})\|$. (The latter point explains the stopping rule — stop once the rate of improvement is too small to bother with.) Gradient descent's part of the contract is to only take a small step (as controlled by the parameter α), so that the guiding linear approximation is approximately accurate.

Under mild assumptions, gradient descent converges to a local minimum, which may or may not be a global minimum. If f is convex — meaning all chords lie above its graph — then gradient descent converges to a global minimum (under mild assumptions). Some important problems are convex (like the regression problems discussed today), while others are not (like the QWOP problem on Homework #3).

For the linear regression problem, the dimension n is the number of (real-valued) features or attributes of each data point $\mathbf{x}^1, \ldots, \mathbf{x}^m \in \mathbb{R}^n$. Also given are real-valued labels $y^1, \ldots, y^m \in \mathbb{R}$. We associate each vector $\mathbf{w} \in \mathbb{R}^n$ with the linear function $\mathbf{x} \mapsto \mathbf{w}^T\mathbf{x}$.[2] We

[1] What if you don't know the gradient? (E.g., on the QWOP project on Homework #3.) Remember that each coordinate of the gradient of f is the instantaneous rate of change of f as the ith coordinate is varied. So you can just estimate each coordinate $\frac{\partial f}{\partial w_j}(\mathbf{w})$ by $[f(w_1, \ldots, w_{i-1}, w_i + \eta, w_{i+1}, \ldots, w_n) - f(\mathbf{w})]/\eta$ for small η.

[2] This would seem to prohibit the linear function from having an intercept — i.e., it is forced to map $\mathbf{0}$ to $\mathbf{0}$. This is for convenience and without loss of generality: to encode an intercept, preprocess the data points,

took the objective function f equal to the mean-squared error (MSE) achieved by a linear function, so

$$f(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m} E_i(\mathbf{w})^2, \tag{3}$$

where

$$E_i(\mathbf{w}) = \underbrace{\left(\sum_{j=1}^{n} w_j x_j^{(i)}\right)}_{\mathbf{w}\text{'s ``prediction'' of } \mathbf{x}^{(i)}\text{'s label}} - \underbrace{y^{(i)}}_{i\text{'s label}}$$

is the prediction error made by the function $\mathbf{w}$ for the label of the data point $\mathbf{x}^{(i)}$. The gradient of this f is

$$\nabla f(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m} 2E_i(\mathbf{w}) \cdot \mathbf{x}^{(i)}, \tag{4}$$

and so gradient descent moves in the opposite direction of this. Recall the interpretation: each data point $\mathbf{x}^{(i)}$ "votes" to change the coefficients $\mathbf{w}$ in the direction that would improve the prediction of its label as rapidly as possible (the direction $\mathbf{x}^{(i)}$ if $\mathbf{w}$ underestimates $y^{(i)}$, or $-\mathbf{x}^{(i)}$ if $\mathbf{w}$ overestimates $y^{(i)}$). Each vote is weighted according to the current error of $\mathbf{w}$ on that data point, and then these weighted votes are averaged to obtain the direction in which to move.

Finally, recall that each iteration of gradient descent takes $O(mn)$ time for linear regression — $O(n)$ time per data point. Observe that the work involved is unusually easy to parallelize — one can just distribute the data points across however many cores or machines are available, independently compute the summands in the gradient (4), and then aggregate the results.

3 How Big Are m and n?

How should we feel about a running time of $O(mn)$ per iteration of gradient descent? The answer clearly depends on how big m and n are. The values of m and n vary greatly across different data sets, and the choice of computational method to use depends on their values.

3.1 Different Data Sets

At one extreme, we have the case of small data sets. Almost all data sets collected by hand qualify as "small," which covers the bulk of the data sets that pre-date the 21st century. Classical statistics was developed with such manageable data sets in mind. One famous example is the Iris data set, which was collected by the botanist Edger Anderson and then statistically analyzed by Ronald Fisher (in 1936) [1]. Anderson measured the length and

adding a new first coordinate with $x_1^{(i)} = 1$ for every data point $\mathbf{x}^{(i)}$ (so coordinates $j = 2, 3, \ldots, n$ are the "real" ones). Then w_1 can encode whatever intercept you want.

width of the petal and the sepal of each iris, so each data point is a 4-tuple ($n = 4$). The number of data points is also small ($m = 150$) but this is a less important point.

These days, technology enables the collection of very large data sets (e.g., through Web crawling). For example, consider a collection of documents, each represented as a "bag of words." Recall this means representing a document as a (sparse) n-vector, where each coordinate counts the frequency of a given word in the document. The number n of dimensions is equal to the number of words you're keeping tracking of, which is often in the tens of thousands. The number m of documents in a given data set varies, but in general you want to use the biggest data sets that you can get your hands on. For example, there are roughly 5 million articles on Wikipedia; in this case, mn would be in the tens or hundreds of billions (i.e., pretty big).

The story is similar with images. For example, representing a 100-by-100-pixel image with a vector of pixel intensities (grayscale, say) results in $n = 10^4$. Image data set sizes vary, but there are some very big ones around (e.g., the Tiny Images dataset has roughly 80 million 32-by-32 images [2]). So mn is again in the tens or hundreds of billions.

3.2 The Normal Equations for Small Data Sets

For small data sets, there is no need to run gradient descent to perform linear regression. The problem of minimizing the mean-squared error is so nice that it admits a closed-form solution. Specifically, the solution is

$$\mathbf{w} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y}, \tag{5}$$

where

$$\mathbf{X} = \begin{bmatrix} — & \mathbf{x}^{(1)} & — \\ — & \mathbf{x}^{(2)} & — \\ & \vdots & \\ — & \mathbf{x}^{(m)} & — \end{bmatrix}$$

is the $m \times n$ matrix with $\mathbf{x}^{(i)}$'s for rows (with the dummy coordinate $x_1^{(i)} = 1$ for all i), and

$$\mathbf{y} = \begin{bmatrix} y^1 \\ y^2 \\ \vdots \\ y^n \end{bmatrix}$$

denotes the m-vector of $y^{(i)}$'s. The n equations in (5) are called the *normal equations*;[3] the derivation is straightforward calculus (setting the gradient of MSE($\mathbf{w}$) to $\mathbf{0}$ and solving).

Solving the equations in (5) requires $O(n^3 + mn^2)$ time — $O(n^3)$ to invert the n by n matrix $\mathbf{X}^T\mathbf{X}$ and $O(mn^2)$ for the matrix multiplications. There is no reason not to use the normal equations to perform linear regression when n is reasonably small (in the 100s, say) and m is not extremely big.

[3]Kinda frightening to think about what the "abnormal" equations might look like...

4 Stochastic Gradient Descent: The Case of Large m

Once the number n of dimensions is moderately large (as is the case with documents or images), the matrix operations needed for solving the normal equations require a prohibitive amount of computation. If mn is not extremely large, then the per-iteration cost of $O(mn)$ required by basic gradient descent is fine.

In general you want to use as much data as possible, since more data allows you to learn richer and more accurate models. For very large m, it can be problematic to spend $O(mn)$ time computing the gradient (4) in every single iteration of gradient descent.[4] For concreteness, imagine that m is so large that computing the gradient requires one day of wall-clock time on the best computing platform available to you.

One way to think about the competing methods of gradient descent and solving the normal equations is as a trade-off between the number of iterations and the computation required per iteration. At one extreme, solving the normal equations can be regarded as a single-iteration algorithm, with a fair amount of work done in that iteration ($O(n^3 + mn^2)$). Gradient descent requires multiple iterations to converge to a solution — the number depends on many factors, but for now think of it as in the dozens or hundreds — but does only $O(mn)$ work per iteration. Can we take this idea further, doing still less work per iteration, at the cost of a small number of extra iterations? *Stochastic gradient descent* provides an affirmative answer to this question.

Stochastic gradient descent is defined for objective functions f that can be expressed as the average of simpler functions:

$$f(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m} f_i(\mathbf{w}).$$

Mean-squared error (3) is one example, and there are many others, especially in machine learning contexts. (For a non-example, see the QWOP problem on Homework #3.) For such a function, by linearity of derivatives, we have

$$\nabla f(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m} \nabla f_i(\mathbf{w}).$$

The update rule in stochastic gradient descent is

$$\mathbf{w} := \mathbf{w} - \alpha \cdot \nabla f_i(\mathbf{w}), \tag{6}$$

where $i \in \{1, 2, \ldots, m\}$. For example, when minimizing MSE, this update rule is

$$\mathbf{w} := \mathbf{w} - \alpha \cdot 2E_i(\mathbf{w}) \cdot \mathbf{x}^{(i)}.$$

[4]Thus there are two different bottlenecks to better machine learning: the data bottleneck, and the computation bottleneck. The major advances in applied machine learning over the past 5-10 years are largely attributable to groups that simultaneously have both massive amounts of data and also the massive computational resources required to process it (think Google Brain).

This is, instead of asking all data points for their votes and averaging as in (4), we only ask for the opinion of a *single* randomly chosen data point — the dictator for this iteration.[5]

How is i chosen? For intuition, imagine that we choose i uniformly at random from $\{1, 2, \ldots, m\}$. Then the expected value of the new value of $\mathbf{w}$ is

$$\begin{aligned}\mathbf{E}[\text{new } \mathbf{w}] &= \underbrace{\frac{1}{m}}_{\mathbf{Pr}[i \text{ chosen}]} \sum_{i=1}^{m} \underbrace{(\mathbf{w} - \alpha \cdot \nabla f_i(\mathbf{w}))}_{\text{new } \mathbf{w} \text{ if } i \text{ chosen}} \\ &= \mathbf{w} - \alpha \cdot \nabla f,\end{aligned}$$

where the expectation is over the random choice of i. That is, the expected effect of one iteration of stochastic gradient descent is exactly the same as the (deterministic) effect of one iteration of basic gradient descent!

Rather than independently choosing an index i each iteration, standard practice is to order the data points in some way and perform one or more passes over them in this order. The order $i = 1, 2, \ldots, m$ is often used, or if you want to be safe you can randomly order the points. Summarizing:

Stochastic Gradient Descent

initialize $\mathbf{w} := \mathbf{w}_0$
`// optionally, randomly reorder the indices` $\{1, 2, \ldots, m\}$
repeat
  for $i = 1$ to m **do**
    $\mathbf{w} := \mathbf{w} - \alpha \cdot \nabla f_i(\mathbf{w})$

Each iteration of the main loop (i.e., each pass over the coordinates) is called an *epoch*. The number of epochs can be as small as one, or more can be used, computation time permitting.

The primary benefit of stochastic gradient descent is clear: since only one data point is used, each iteration takes only $O(n)$ time. Thus, an entire epoch of stochastic gradient descent takes roughly the same amount of time as just one iteration of gradient descent. When there is a limited amount of computation available, the question then is: which performs better, k epochs of stochastic gradient descent, or k iterations of gradient descent? (Here k is however large you can get away with the available computational resources and time constraints.) Very commonly, the answer is the former, and stochastic gradient descent winds up being a big win. Stochastic gradient descent (and optimized variants of it) is pretty much the dominant paradigm in modern large-scale machine learning.[6]

A couple of implementation notes. Because of the variance involved in stochastic gradient descent — in a given iteration, a poor choice of a data point can lead you in the completely

[5] In this context, the basic version of gradient descent (Section 2) is sometimes called *batch* gradient descent, as the whole batch of data points is used for every update step.

[6] For example, when you hear buzzwords like "backpropagation in neural networks," it's typically just referring to an efficient implementation of stochastic gradient descent.

incorrect direction — the step size α is often set to a smaller value than in gradient descent. If the step size is too large, then stochastic gradient descent can fail to converge. Second, in practice one often interpolates between the extreme cases of batch gradient descent (where all m data points are used every iteration) and stochastic gradient descent (where only one data point is used per iteration) using "mini-batches." This just means that, each iteration, the gradient terms are computed for a small number (e.g., 128) of data points, with the average of these terms used to compute the next point. With a properly vectorized implementation, this can be a nearly costless way to decrease the variance in the direction moved each iteration.

5 Increasing the Number of Dimensions

5.1 Solving the Right Problem

So far we've discussed computational issues — how to actually implement gradient descent so that it scales to very large data sets. We've been taking the function f as given (e.g., as MSE). How do we know that we're minimizing the "right" f?

One reason that minimizing MSE might be the wrong optimization problem is because of simple typechecking errors. The point of linear regression is to predict real-valued labels for data points. But what if we don't want to predict a real value? For example, what if we want to predict a binary value, like whether or not a given email is spam? There turn out to be analogs of linear regression and the mean-squared error objective for several other prediction tasks. For example, for binary classification, the most common approach is called "logistic regression," where the linear prediction functions we've been studying are replaced by functions bounded between 0 and 1,[7] and the analog of mean-squared error is called "log loss." You can learn much more about other prediction tasks in a machine learning course like CS229, and we won't duplicate that material here. The template for modeling and implementing these other prediction tasks follows the exact same one used here for linear regression, so you're now well-positioned for a deeper study.[8]

Even if your goal is to predict real-valued labels for points, linear regression might not be good enough. To understand the issue, consider Figure 1, which illustrates an example with $n = 1$. There is no line that is a good fit for these data points. So while linear regression will indeed result in the linear function with the minimum-possible MSE, this optimal MSE will be large. On the other hand, it's clear that there is a quadratic function that does fit these data points quite closely. This motivates computing the best predictor from a class of functions that includes quadratic functions and possibly higher-order polynomials.

[7] Specifically, $\mathbf{x} \mapsto 1/(1 + \exp(-\mathbf{w}^T\mathbf{x}))$ rather than $\mathbf{x} \mapsto \mathbf{w}^T\mathbf{x}$.

[8] One exception: many of the other tasks, including logistic regression, do not admit a closed-form solution analogous to the normal equations. Thus gradient descent is even more relevant for such tasks than for linear regression.

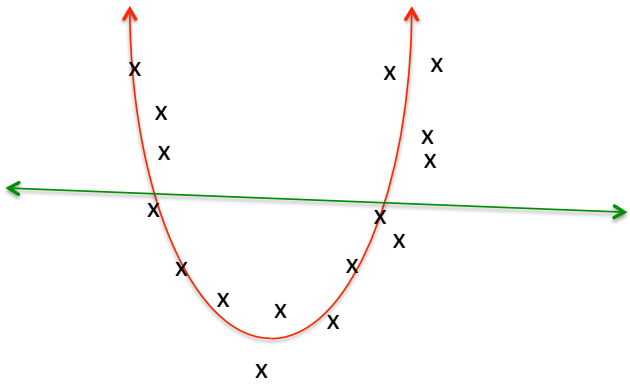

Figure 1: A point set where there is no linear prediction function with small MSE, but there is a quadratic prediction function with small MSE.

5.2 Encoding Nonlinear Relationships with Extra Dimensions

There is a slick way of adapting the tools we've developed to the case of nonlinear prediction functions. The idea is to *increase the number of dimensions* in the data points.[9] For instance, with $n = 1$, consider mapping each data point $x^{(i)} \in \mathbb{R}$ to a $(d+1)$-dimensional vector:

$$x \mapsto \hat{\mathbf{x}} = (1, x, x^2, \ldots, x^d).$$

For example, if $d = 4$, a point with value 3 is mapped to the 5-tuple $(1, 3, 9, 81, 243)$.

Now imagine solving the linear regression problem not with the original data points $x^{(1)}, \ldots, x^{(m)}$, but with their expanded versions $\hat{\mathbf{x}}^{(1)}, \ldots, \hat{\mathbf{x}}^{(m)}$. The result is a linear function in $d+1$ dimensions, specified by coefficients $w_0, \ldots, w_d$. The prediction of $\mathbf{w}$ for an expanded data point $\hat{\mathbf{x}}^{(i)}$ is $\mathbf{w}^T\hat{\mathbf{x}}^{(i)} = \sum_{j=0}^{d} w_j(x^{(i)})^j$. We can interpret the number $\sum_{j=0}^{d} w_j(x^{(i)})^j$ as a nonlinear prediction for the original data point $x^{(i)}$. For example, the prediction $y = x^2 - x + 2$ (for the original one-dimensional points) corresponds to the coefficient vector $\mathbf{w} = (2, -1, 2, 0, 0, \ldots, 0)$. Linear regression in the expanded space minimizes the MSE of any linear function of the $\hat{\mathbf{x}}^{(i)}$, and therefore the MSE of any degree-d polynomial function of the $x^{(i)}$'s.[10] For $n > 1$, one can analogously preprocess the data points to add one extra coordinate for each monomial of degree at most d (e.g., for $d = 2$, all products of pairs of the coordinates of a data point).[11]

[9] We already saw a simple example of this principle, when we showed how linear functions in $n-1$ dimensions with an intercept can be encoded using linear functions in n dimensions with no intercept (by adding a "dummy" coordinate in which points have value 1).

[10] How can we do nonlinear computations using a linear function? Because the nonlinear part is carried out in the preprocessing step of mapping each x to $\hat{\mathbf{x}}$. Given $\hat{\mathbf{x}}$, the prediction of its label is just a linear function of the coefficient vector $\mathbf{w}$.

[11] You might be concerned about the number of additional coordinates that this idea creates as n and d grow large. If you study machine learning more deeply, you'll learn about the "kernel trick," which allows you to deduce the result of certain computations (such as a gradient descent update step) on the expanded data points (the $\hat{\mathbf{x}}^{(i)}$'s) without ever explicitly constructing these expanded representations.

There are many other ways to take data points in their "natural" representation and expand them with additional dimensions. For example, with documents, in addition to having one dimension for each word (indicating the number of occurrences), one can have one dimension for each ordered pair of words (tracking how many times they occur consecutively) or for each "n-gram" (a sequence of n consecutive characters, which may or may not form a word). Similarly, for images, in addition to having one dimension per pixel, it's common to add dimensions corresponding to "patches," meaning small grids of adjacent pixels (e.g., 6-by-6 or 8-by-8 grids). One way to translate a patch into a real-valued attribute is to compute its distance (Euclidean, say) from one or more reference patches (e.g., one of blue sky, or one of an edge separating different objects). For both documents and images, these augmentations can drive the number of dimensions n into the millions or billions.

5.3 Overfitting

We mentioned how more data is always better, computational resources permitting. Are more features always better? The benefit is intuitively clear — better accuracy of fit (as in Figure 1). But there is a catch: with many features, there is a strong risk of *overfitting* — of learning a prediction function that is an artifact of the specific data set, rather than one that captures something more fundamental. Thus there is a trade-off between expressivity and predictive power.

To understand the issue, let's return to the case of $n = 1$ and the map $x \mapsto \hat{\mathbf{x}} = (1, x, x^2, \ldots, x^d)$. Suppose we take $d = m$, so that we are free to use polynomials with degree equal to the number of data points. Now we can get a mean-squared error of *zero*! The reason is just polynomial interpolation — given m pairs $(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})$, there is always a degree-m polynomial that passes through all of them (Figure 2). Is a MSE of zero good? Not necessarily. The polynomial in Figure 2 is quite "squiggly," and meanwhile there is a line that fits the points quite closely. If the true relationship between the $x^{(i)}$s and $y^{(i)}$s is indeed roughly linear (plus some noise), then the line will give much more accurate predictions on new data points x than the squiggly polynomial. In machine learning terminology, the squiggly polynomial fails to "generalize." Remember that the point of this exercise is to learn something "general," meaning a prediction function that transcends the particular data set and remains approximately accurate even for data points that you've never seen.

6 Regularization: The Case of Large n

6.1 Occam's Razor

It is common in modern machine learning to take a "kitchen sink" approach to defining the features of data points — throwing in every conceivably useful feature that you can think of, and taking n as large as possible. With this approach, it is essential to be on guard against overfitting. Philosophically, the solution is *Occam's Razor* — to be biased toward simpler models, on the basis that they are capturing something more fundamental, rather than some

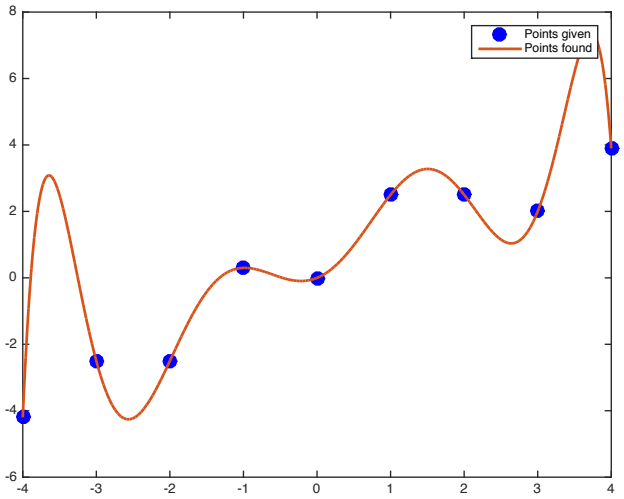

Figure 2: Using a high-degree polynomial to achieve zero MSE can result in a squiggly polynomial, even when there is a linear function with low MSE.

artifact of the specific data set.

Regularization is a concrete method for implementing the principle of Occam's razor. The idea is to add a "penalty term" to the optimization problem, such that more complex models incur a larger penalty. For the case of linear regression, the new optimization problem is to minimize

$$\text{MSE}(\mathbf{w}) + \text{penalty}(\mathbf{w}),$$

where penalty($\mathbf{w}$) is increasing with the "complexity" of $\mathbf{w}$. Thus a complex solution will be chosen over a simple solution only if it leads to a big decrease in the mean-squared error.[12]

6.2 L_2 Regularization

There are many ways to define the penalty term. We'll just consider the most widely used one, which has many names: *ridge regression*, L_2 *regularization*, or *Tikhonov regularization*. This just means that we set

$$\text{penalty}(\mathbf{w}) = \lambda \cdot \|\mathbf{w}\|^2_2,$$

where λ is a positive "hyperparameter," a knob that allows you to trade-off smaller MSE (preferred for small λ) versus smaller model complexity (preferred for large λ).[13] That is,

[12] The same idea can be used for other types of regression problems, like the logistic regression problem mentioned in Section 5.1.

[13] "Hyperparameter" meaning a parameter that is chosen outside the learning procedure. (Whereas the w_j's, which are computed by the learning procedure and define the learned model, are simply "parameters.") So how is λ chosen? Typically, just by trial and error, to find the value that results in the learned model with the most predictive power. (Or, if you want to sound fancy, you can refer to this trial-and-error as "hyperparameter optimization" or "hyperparameter tuning.")

we identify "complex functions" with those with large weights.[14] For example, most of the current work in deep learning uses this form of regularization.[15]

To see how this addresses the overfitting problem discussed in Section 5.3 (Figure 2), we note that a degree-m polynomial passing through m points is likely to have all non-zero coefficients, including some large coefficients. A linear function has mostly zero coefficients, and will be preferred over the squiggly polynomial for data points with an approximately linear relationship (unless λ is very small).[16]

6.3 Applying Gradient Descent

All of the ideas covered in these two lectures extend easily to regularized regression. In our running example of linear regression with L_2 regularization, the objective is to minimize

$$f(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m} E_i(\mathbf{w})^2 + \lambda \sum_{j=1}^{n} w_j^2. \tag{7}$$

Since we've only added some new squared terms to the previous (convex) MSE objective function, we again have a convex function with only one local minimum (the global minimum). The gradient is the same as before (Section 2), except with an extra $2\lambda w_j$ term in each coordinate j:

$$\frac{\partial f}{\partial w_j}(\mathbf{w}) = \left(\frac{1}{m}\sum_{i=1}^{m} 2E_i(a)\cdot x_j^{(i)}\right) + 2\lambda w_j.$$

The interpretation of this new gradient is the same as before (a weighted average of votes by the data points), except that the new term also exerts a force pushing coefficients closer to 0, with greater force applied to the coefficients with larger magnitudes.

Gradient descent requires essentially the same amount of computation as in the unregularized linear regression problem, and remains equally easy to parallelize. For the stochastic gradient descent version, a single data point $\mathbf{x}^{(i)}$ is chosen each iteration and the update used is just

$$w_j := w_j - \alpha \cdot \left(2E_i(a)\cdot x_j^{(i)} + 2\lambda w_j\right)$$

for each coordinate j. (The $2\lambda w_j$ term is always included, no matter which data point $\mathbf{x}^{(i)}$ is chosen.)

[14] A detail: one generally doesn't penalize the weight that corresponds to the function's intercept (the first coordinate in the setup in Section 2), just the weights corresponding to the "real" coordinates of the data points.

[15] The second-most popular choice of penalty term is probably the ℓ_1 norm of $\mathbf{w}$ (i.e., penalty($\mathbf{w}$) = $\lambda \cdot \|\mathbf{w}\|_1$). This is naturally called L_1 *regression*, and the objective function is also referred to as "the lasso."

[16] Regularization can also be useful in low-dimensional problems, for example to reduce the sensitivity of the computed prediction function to outliers.

7 Lecture Take-Aways

1. For many machine learning problems, replacing the basic gradient descent method by stochastic gradient descent is crucial for accommodating very large data sets. While the former touches every data point every iteration (to average the corresponding gradient terms), the latter uses only one (or a small number) of data points in each iteration. Stochastic gradient descent is the dominant paradigm in modern machine learning (e.g., in most deep learning work).

2. More data is always better, as long as you have the computational resources to handle it.

3. More features (or dimensions) offer a trade-off, allowing more expressive power at the risk of overfitting the data set. Still, these days the prevailing trend is to include as many features as possible.

4. Regularization is key to pushing back on overfitting risk with high-dimensional data. The general idea is to trade off the "complexity" of the learned model (e.g., the magnitudes of weights of a linear prediction function) with its error on the data set. The goal of learning the simplest model with good explanatory power, on the basis that this explanation is the most likely to generalize to unseen points.

5. Adding regularization imposes essentially no extra computational demands on (stochastic) gradient descent.

References

[1] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188, 1936.

[2] A. Torralba, R. Fergus, and W. T. Freeman. 80 million tiny images: a large dataset for non-parametric object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(11):1958–1970, 2008.