CHAPTER

3 Linear Regression

In this chapter, we focus on the regression task, specifically applying one of the simplest possible regression models: the linear model. We use this class of functions to explore a number of fundamental tools that will be useful in the sequel, including matrix-based representations of the data, gradient-based optimization of our model, and further exploration of the notions of model complexity.

Suppose that we have observed a feature vector $x = [x_1, \ldots, x_n]$, and our prediction takes the form of a linear function:

$$f(x\,;\,\theta) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots$$

It is helpful for us to define our variables, features, etc. in the form of matrices, which will allow us to write some equations compactly and also translate well into code. To do so, let us define a "zero-th" feature $x_0^{(i)} = 1$ for all data i; then, we can express

$$f(x\,;\,\theta) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots = \theta \odot x^T$$

where $\theta = [\theta_0, \ldots, \theta_n]$ and $x = [x_0, \ldots, x_N]$ are vector representations of the model parameters and features (with $x_0 = 1$), and "$\odot$" is the usual vector-vector dot product.

Recall that our data for training, $D = \{(x^{(i)}, y^{(i)})\}$ consist of pairs of observed feature vectors $x^{(i)}$ and target values $y^{(i)}$, for $i = 1 \ldots m$. Let us stack these up vertically, giving a **data matrix** $\mathbf{X}$ and target vector $\mathbf{y}$:

$$\mathbf{y} = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix} \qquad\qquad \mathbf{X} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \ldots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \ldots & x_n^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} & \ldots & x_n^{(m)} \end{bmatrix}$$

Where possible, we will try to be consistent that the horizontal axis (column index) of vectors and matrices corresponds to the feature index, and the vertical axis (row index) corresponds to the data index[1]. Then, the (i, j)th entry of $\mathbf{X}$ is the jth feature of the ith data point: $\mathbf{X}_{ij} = x_j^{(i)}$. Again, here we have taken "feature zero" to be the constant $x_0^{(i)} = 1$.

[1]Note that the orientation of each data point corresponding to a row, with different features in each column, is completely arbitrary. Thus many texts or code may use the opposite convention, leading to transposition differences between equations.

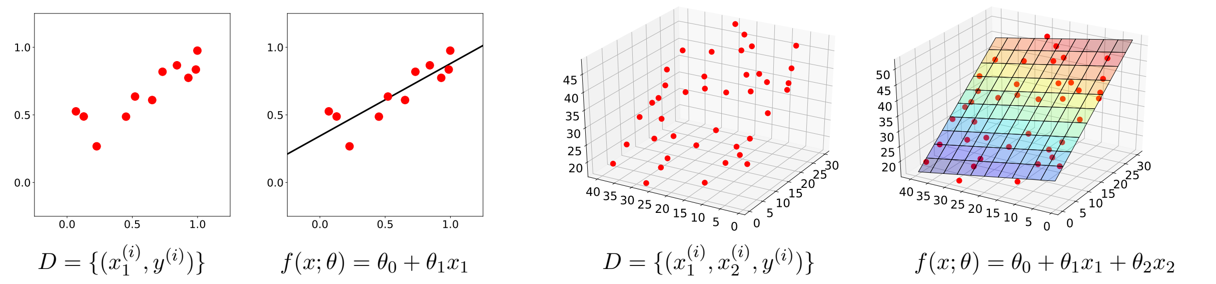


Figure 3.1: Scatterplot of data D and linear fit $f(x;\theta)$ for a single feature, $x = [x_1]$, and for a two-dimensional feature, $x = [x_1, x_2]$.

3.1 Optimization

The process of learning involves selecting the predictor out of our hypothesis class, i.e., here the set of linear models, that best fits our observed data D. To this end, we first select a loss function J, and then find a predictor (corresponding to a value of the parameter vector θ) that minimizes this loss function.

Visualizing the loss function

For a given observation (x, y), we can compute the error in our prediction (sometimes called the **error residual**) as $y - f(x;\theta)$. In general, we would like to minimize the overall size of these errors. A common, and as we shall see computationally convenient choice is the scaled Euclidean norm of the vector of residuals, corresponding to the mean squared error (MSE) cost,

$$J_{\text{MSE}}(\theta) = \frac{1}{m}\sum_i \left(y^{(i)} - f(x^{(i)}\,;\,\theta)\right)^2$$
$$= \frac{1}{m}\,\|(\mathbf{y} - \mathbf{X} \odot \theta^T)\|^2$$

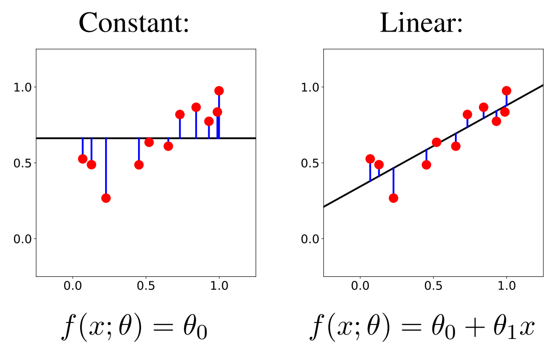


Figure 3.2: Error residuals (blue lines) on a particular data set (red) for a constant predictor (left) and linear predictor (right).

The loss function $J(\cdot)$ tells us the accuracy of a given parameter vector at predicting our training data. This is a function defined over the space of parameters: each possible value of θ corresponds to a different line[2]. If θ is two-dimensional, $\theta = [\theta_0, \theta_1]$, we can visualize the function J as a surface, or use colors or contours to suggest the three-dimensional height (cost) for each value of θ on a 2D plot.

In order to describe a learner and cost function, we can use two types of plot. In the first type, we show the feature and target values, along with the prediction function itself; we call this plot domain the "feature space", and can use it to show the fit of our current model (value of θ) against training data, as in Figure 3.2. The second type of plot shows the value of the cost function J as it ranges over many different values of θ. We denote the domain of these plots, such as Figure 3.3, the "parameter space", i.e., the set of all valid parameter vectors θ. Any particular model will be a point in this space, and the model with the least mean squared error is at the minimum of the function J.

[2]Note that $J(\theta)$ is also a (hidden) function of the data set $\mathbf{X}, \mathbf{y}$; we normally omit this dependence for brevity.

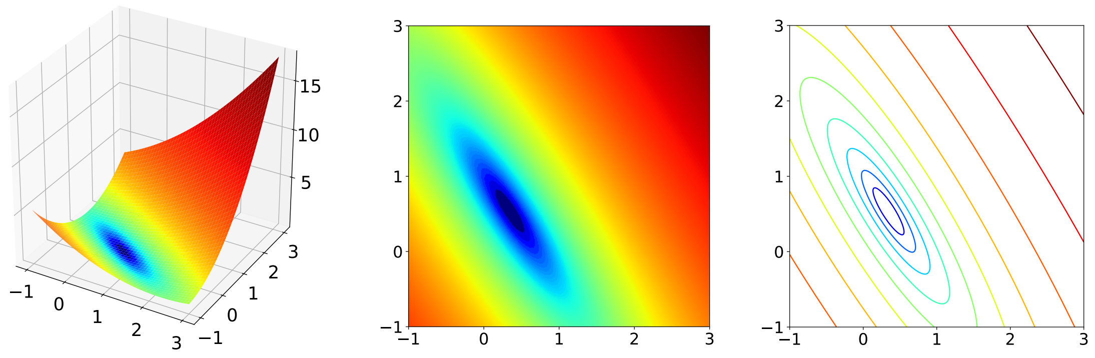


Figure 3.3: Three visualizations of the loss function $J_{\text{MSE}}(\theta)$, for the linear model in Figure 3.2(b), with $\theta = [\theta_0, \theta_1]$. Since J is a function of two parameters, we can visualize it as (a) a 3D surface; typically we simply (b) project such surfaces onto the feature space and display their value with color, or even just (c) show the contours (topography) of the resulting surface.

Gradient descent

In order to find a model that fits the data well, one option is to follow the local slope of $J(\theta)$ downward towards a local minimum. We can evaluate the gradient direction, i.e., the direction in which our cost function $J(\theta)$ has the greatest increase; going in the opposite direction gives us the direction of greatest decrease of our cost J. This gradient will be a vector of the same dimension as θ:

$$\nabla J(\theta) = \left[\frac{\partial J}{\partial \theta_0}, \frac{\partial J}{\partial \theta_1}, \ldots, \frac{\partial J}{\partial \theta_n} \right]$$

For our linear model, $f(x\,;\,\theta) = x \odot \theta^T$, and MSE loss, evaluating any one of these derivatives gives,

$$\frac{\partial J_{\text{MSE}}}{\partial \theta_j} = \frac{2}{m} \sum_i (y^{(i)} - x^{(i)} \odot \theta^T)\, x_j^{(i)}$$

from which it is easy to verify that we can compactly express,

$$\nabla J_{\text{MSE}}(\theta) = \frac{2}{m}\,(\mathbf{y}^T - \theta \odot \mathbf{X}^T) \odot \mathbf{X}. \tag{3.1}$$

In gradient descent, we simply initialize to some starting value of θ and repeatedly update by choosing a new θ by moving in the direction of steepest descent, e.g.,

$$\theta \leftarrow \theta - \alpha \nabla J(\theta)$$

Here, "$\leftarrow$" indicates that we are updating the value of θ using the quantity on the right. The value α is a step-size parameter that tells us how far to move in the direction of the gradient. The choice of step size can be very important in gradient descent methods, as it often controls how fast we converge to a local minimum. Values that are too small move very slowly; values that are too large can skip over or even oscillate around local minima[3].

[3]Parameters such as the step size α that influence the amount by which the model is updated at each step are sometimes called a **learning rate**.

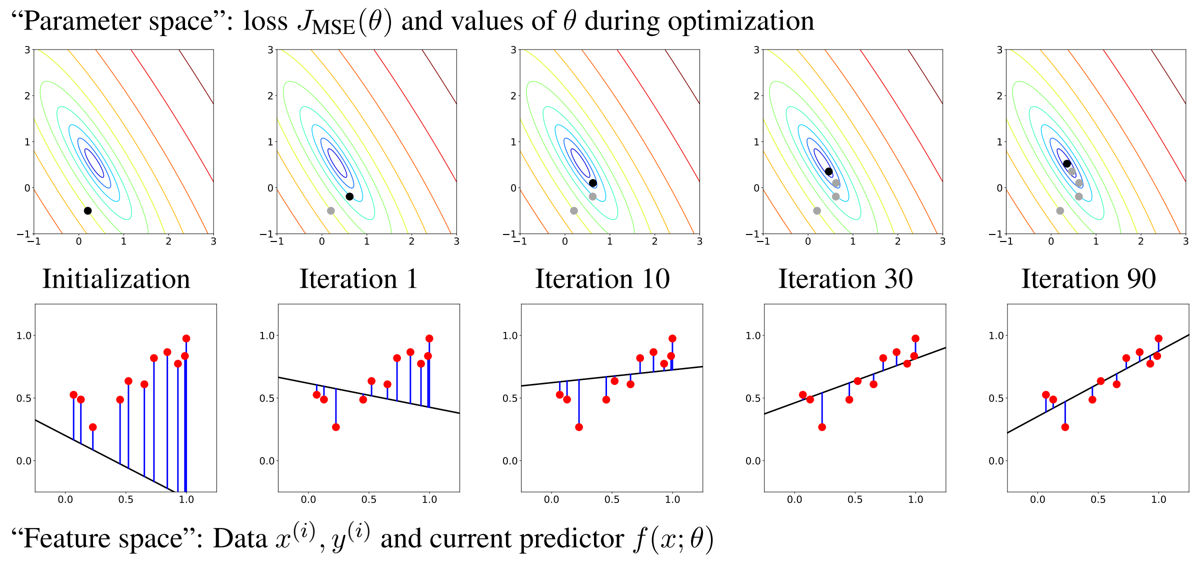


Figure 3.4: Optimization of the parameters θ via gradient descent. The top row shows the space of possible parameter settings (values of θ), displaying the sequence of models obtained during optimization of the loss function $J_{\text{MSE}}(\theta)$, along with the contours of J. The bottom row shows the current iteration's model (value of θ) in terms of its prediction $f(x\,;\,\theta)$ and current fit to the training data.

A common heuristic approach to setting the step size is to let α decrease with each iteration (step), for example choosing α to be inversely proportional to the iteration number: $\alpha = \frac{C}{C+s}$ at iteration s, where C is some constant (e.g., $C = 2$). Another, more sophisticated option is to use a **backtracking line search** to set α. We start with α relatively large, and check whether our update with this step size provides a sufficiently large decrease in the objective, e.g.,

$$J(\theta - \alpha \nabla J) \leq J(\theta) - c\,\alpha \|\nabla J\|^2$$

for some small constant c (say, 10^{-4}); this condition is called the *Armijo rule* [Armijo, 1966]. If the condition is not satisfied, we decrease α (say, to $\frac{\alpha}{2}$) and re-evaluate the condition. When the condition is satisified, we update θ and continue the algorithm.

We also need a criterion for stopping our updates. Typical stopping criteria measure the amount of change applied to θ at the current step, $\alpha\|\nabla J(\theta)\|$ (the vector length of the step), or alternatively the change in the value of the objective $J(\theta)$, after each step. If these values are small, our gradient steps are not significantly updating the solution θ, or not changing its value, and we may stop. Alternatively, stopping criteria based on the number of iterations or the total execution time are also common. Pseudocode for the overall procedure is shown in Algorithm 3.1.

Algorithm 3.1 (Batch) Gradient Descent

```
Initialize parameters $\theta$
repeat for each iteration $t$:
    Compute $\nabla J(\theta)$
    Select step size $\alpha_t$
    Update $\theta \leftarrow \theta - \alpha_t \nabla J(\theta)$
until ConvergenceCondition($t$, $\theta$, $J(\theta)$)
```

In Figure 3.4, we can see the behavior of gradient descent on the loss function J defined over parameter space (top; each point corresponds to a vector $\theta = [\theta_0\ \theta_1]$), and the induced predictors $\hat{y}(x) = \theta x^T$ (bottom; each value of θ corresponds to a different linear predictor for y). As our value of θ evolves from its initial value toward the minimum of J, the predictor defined by θ evolves toward a line that closely fits the data points.

Gradient descent is a type of *local search* method, in which we incrementally improve on our model by applying sequences of small changes. It can thus become trapped in **local minima**, corresponding values of the parameters θ where no small perturbation of θ can improve the models' fit. Such minima may not be **global minima**, i.e., attain the smallest possible value of the loss J. For example, in Fig. 3.5, we may become stuck at values of θ where every small perturbation increases J (blue circles), or where J is flat (a "plateau"; white circle). Only the lowest blue circle is a global minimum.

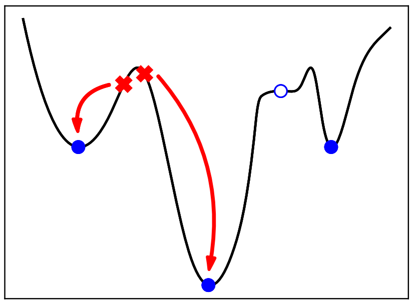


Figure 3.5: In general, gradient descent will converge to some local minimum (circles), or even potentially a plateau (white circle). It can be sensitive to initialization; two nearby initializations (red x) can converge to different minima.

Gradient methods are powerful in part because they are extremely simple, and can be applied to any objective function J that is smooth. They converge to a local minimum under fairly general conditions (such as a reasonable step size), and for nicely behaved functions one can analyze the convergence rate of the algorithm. For example, suppose that $J(\theta)$ is convex and that its gradient is L-Lipschitz continuous, so that

$$\|\nabla J(\theta) - \nabla J(\theta')\| \leq L\|\theta - \theta'\|,$$

i.e., the gradient of J does not change too quickly. Then one can show that after T iterations, we have $J(\theta) - J(\theta^*) \leq O(\frac{1}{T})$ where θ^* is the true optimum of J **?**. For more general functions, assuming that the region around a local minimum is nicely behaved, this gives us intuition about the asymptotic rate of convergence to that minimum for gradient descent.

Stochastic gradient descent

A useful variant of gradient descent often used in machine learning problems, particularly those with a large amount of data available, is **stochastic gradient descent** (SGD). The basic idea of SGD is avoid evaluating the loss J or its gradient ∇J, which generally involve the entire training data set, and instead to use a faster but stochastic (random) estimate. (To contrast, using the full gradient (3.1) is sometimes called **batch gradient descent**.)

SGD is most easily applied to losses that are **decomposable**, meaning that they can be written as a sum or average over the collection of training points:

$$J(\theta) = \frac{1}{m}\sum_i J^{(i)}(\theta)$$

where $J^{(i)}$ is a loss defined on a single data point, e.g., $J^{(i)}_{\text{MSE}}(\theta) = (y^{(i)} - x^{(i)} \odot \theta^T)^2$ for the mean squared error. For decomposable losses, linearity of the derivative tells us that $\nabla J = \frac{1}{m}\nabla J^{(i)}$. Thus, we can select a direction to update our model that is correct on average by simply selecting a data point i at random:

Algorithm 3.2 Stochastic Gradient Descent

```
Initialize parameters $\theta$; $t = 0$
repeat
    Shuffle data randomly
    for each data point $i$: do
        Compute $\nabla J^{(i)}(\theta)$
        Select step size $\alpha_t$
        Update $\theta \leftarrow \theta - \alpha_t \nabla J^{(i)}(\theta)$
        $t \leftarrow t + 1$
    end for
until ConvergenceCondition($t$, $\theta$, $J(\theta)$)
```

$$i \sim \text{Uniform}(\{1, \ldots, m\}) \qquad \Rightarrow \qquad \mathbb{E}\Big[\nabla J^{(i)}(\theta)\Big] = \nabla J(\theta)$$

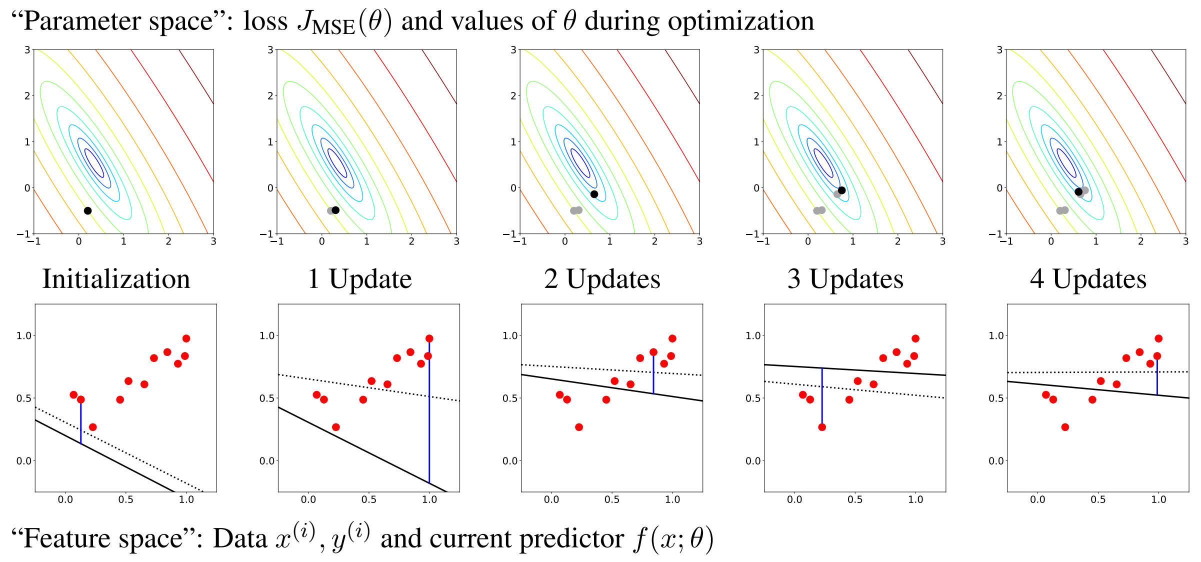


Figure 3.6: Stochastic gradient descent example. After initializing θ (top left), we select a data point i at random and compute its error residual (blue line) and gradient $\nabla J^{(i)}$, then update the model to $\theta + \alpha \nabla J^{(i)}$ (dashed line). At the next step, we select a new data point and repeat. Our sequence of models (top row) moves toward the minimum of J, but with more randomness than batch gradient descent.

In other words, each data point provides us a (random, noisy) estimate of the correct direction to move to improve our fit to the data set overall.[4] In practice, it is usually more efficient to sample i without replacement, which is equivalent to randomly shuffling the data before passing through them sequentially; see Algorithm 3.2.

To compare this behavior to batch gradient descent, consider the amount of progress made in optimizing the function after each **epoch**, or a single pass through the training data set. Using the batch gradient method, one epoch is sufficient to evaluate the gradient and update the parameters θ once, inevitably leaving the model very close to its initialized value. In SGD, one epoch is sufficent to perform m updates of the paramters. These updates are noisy, but if the gradient direction is easy to determine from only a few data points – as is especially common early in the optimization, when the model is not very good – we will make significantly more progress toward minimizing J. Thus, SGD can be much more efficient than batch gradient descent when m is very large.

As an extreme case, imagine that we doubled the size of our data D by simply replicating each point twice. This would not change J (since it is an average) or its gradient at all; but batch gradient descent would proceed twice as slowly as before, while SGD's performance would be unchanged.

Near an optimum, however, we see that $\nabla J^{(i)}$ is only zero in expectation – any particular data point i will often have a non-zero gradient, and so our updates will bounce around the optimum. This property complicates both step size selection and convergence conditions. Unlike in batch gradient where each update is guaranteed to decrease the loss J for a sufficiently small step, in SGD our update direction may not improve the full loss J. Similarly, evaluating the full loss J requires $O(m)$ work, making methods like backtracking line search too expensive. In practice, we can select a sequence of step sizes α_t at

[4]When SGD updates using a single data point at a time, it is sometimes called "online" gradient descent, since it can also be applied to streaming data as it arrives, without storing any data history.

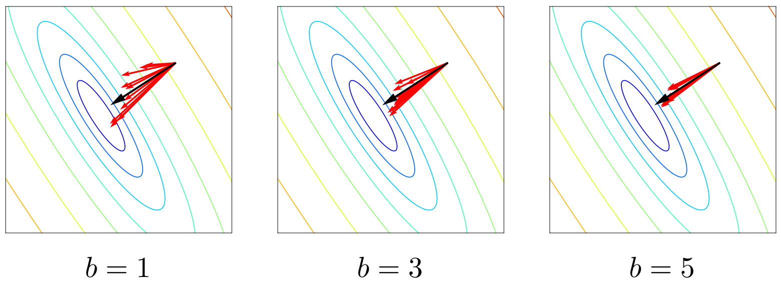


Figure 3.7: Mini-batch gradient estimates. We can reduce the amount of randomness in SGD by using mini-batch estimates. When the batch size $b = 1$, this reduces to standard SGD: each data point i corresponds to a gradient $\nabla J^{(i)}$ (red arrows), the average of which is the batch gradient (black arrow). If we average over b data points, we can reduce the randomness in the gradient estimate, making for a smoother descent at the cost of more computation per update.

iteration t that satisfy the *Robbins-Monro* conditions [Robbins and Monro, 1951]:

$$\sum_{t=0}^{\infty} \alpha_t = \infty \qquad \qquad \sum_{t=0}^{\infty} (\alpha_t)^2 < \infty,$$

such as the common choice $\alpha = \frac{C}{C+t}$ for some constant C. Under mild conditions on J, the Robbins-Monro conditions ensure that our estimate θ converges almost surely[5] to a local optimum.

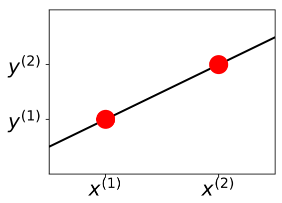
An example execution of stochastic gradient descent on our simple dataset is shown in Figure 3.6. After initializing our model parameters θ, instead of computing the gradient ∇J as in Figure 3.4, we instead select just one data point i, comput the gradient, and update θ. The figure shows the predictions of the current model $f(x; \theta)$ (solid line) as well as of the updated model $f(x; \theta + \alpha \nabla J^{(i)})$ (dashed line) in each plot for easy comparison. As we continue to select data points and update θ, our model fit improves and we move toward the minimum of $J(\theta)$, but less smoothly and with more randomness than when using the full gradient ∇J. Any given update may not change θ very much, or may even worsen the overall fit, but over time we make progress toward minimizing J.

Stochastic gradient descent provides us with more frequent updates to the parameters θ, at the price that those updates are now only noisy estimates of the desired gradient direction. A technique called **mini-batch** trades off these two effects – rather than computing the average over all m data (as in batch gradient), or a single data point (as in SGD), we compute the average gradient over a subset of b randomly selected data points. This reduces the stochasticity of SGD, at the price of fewer parameter updates per epoch. Figure 3.7 shows samples of the mini-batch gradient at a particular θ, for several values of b. As b increases, the sampled gradients become closer to the exact, batch gradient (here, equivalent to $b = m = 11$). In practice, b can be held fixed, or can increase over time as the model is trained (since near convergence, data points are intuitively more likely to point in different directions, since the average should be near zero).

Closed form optimum for squared error

A convenient property of the linear regression function and mean squared error loss is that its optimum can actually be computed in closed form. A simple example helps illustrate the idea.

[5]Meaning, sequences of updates for which θ does not converge have probability zero.



Suppose we have a simple regression problem with a single, scalar feature x. Given two training points, $D = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)})\}$, with $x^{(1)} \neq x^{(2)}$, there is a unique line that passes through them, and it is trivial to solve for the parameters of this line, by solving the equations:

$$y^{(1)} = \theta_0 + \theta_1\, x^{(1)}$$
$$y^{(2)} = \theta_0 + \theta_1\, x^{(2)}$$

or, equivalently, $\mathbf{y} = \mathbf{X} \odot \theta^T$. The condition $x^{(1)} \neq x^{(2)}$ ensures that $\mathbf{X}$ is full rank, and thus invertible, so that we can solve $\theta = \mathbf{y}^T \odot (\mathbf{X}^T)^{-1}$.

Given more than two data points, of course, there may be no line which passes exactly through the examples; such a system is called *over-determined*. The preceding idea generalizes by solving for the optimum of the squared error loss, i.e, the point at which the gradient is zero. Plugging in our formula for the gradient, we have:

$$\nabla J_{\text{MSE}}(\theta) = (\mathbf{y}^T - \theta \odot \mathbf{X}^T) \odot \mathbf{X} = 0$$

Distributing $\mathbf{X}$ inside and rearranging gives a quadratic equation, the solution of which is the minimum squared error (MSE) estimator

$$\hat{\theta}_{\text{MSE}} = \mathbf{y}^T \odot \mathbf{X} \odot (\mathbf{X}^T \odot \mathbf{X})^{-1} \tag{3.2}$$

The term $(\mathbf{X} \odot (\mathbf{X}^T \odot \mathbf{X})^{-1})$ is called the (Moore-Penrose) pseudo-inverse. For non-square $\mathbf{X}$, for example $m > n$ (an overconstrained set of equations), the pseudo-inverse can be thought of as a generalization of the standard matrix inverse $(\mathbf{X}^T)^{-1}$; if $m = n$ and $\mathbf{X}$ is full rank, the two will be equivalent.

Given that (3.2) exactly minimizes the mean squared error loss for linear regression, why should we need methods like batch or stochastic gradient descent? The answer is two-fold. First, few combinations of loss functions and models can be minimized exactly in closed form. While MSE is a common and useful loss, we shall see in the sequel that other choices of loss function may be useful in some settings, and do not lead to a closed form estimator. Similarly, non-linear models require a more general optimization technique. Gradient-based optimization is easily applied to a wide variety of models and losses.

Second, even for linear regression with an MSE loss, there can be computational reasons to prefer gradient optimization. The computation (3.2) requires computing an $n \times n$ matrix inverse, which has a practical[6] computational complexity of $O(n^{2.8})$ Strassen [1969], and requires $O(n^2)$ memory, which often becomes impractical beyond a few thousand features. Similarly, if m is extremely large, using SGD or a similar online method we may be able to get close to the optimal parameters before making even a single pass through all m data points. **streaming, sublinear cite?**

Common variants of gradient descent

Gradient descent is one of the workhorses of optimization, particularly in machine learning, which has led to considerable effort to make it faster and more automatable and reliable in practice. However, gradient and stochastic gradient descent methods are often quite sensitive to the step size schedule (step size as a function of iteration). Although the Robbins-Monroe conditions enable theoretic analysis of the rate of convergence to a local optimum, in practice they may not perform optimally.

One major issue with gradient-based optimization is the notion of a **poorly conditioned** objective. Mathematically, we say that a matrix A is poorly conditioned if the ratio of its largest to smallest eigenvalue, λ_1/λ_n, (called the **condition number** of A) is large. Geometrically, if we envision minimizing

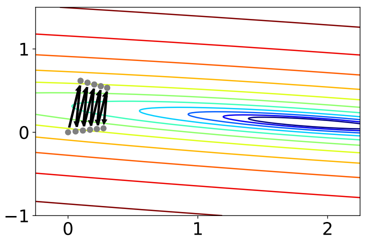
[6] Slightly better rates can be achieved asymptotically, but are not generally used for matrices of realistic size.

the quadratic[7] function, $J(\theta) = \|A\theta - y\|^2$, our objective will have ellipsiodal iso-contours, with the scale of the ellipses' axes given by the eigenvalues λ_i. Visualizing in $n = 2$ dimensions, if A is poorly conditioned, the ellipses appear long and thin.

In gradient optimization, this usually causes optimization to be slow. In particular, any step size that is small enough to avoid increasing the objective as it moves across the narrow direction of the ellipse, will be so small as to make little progress in moving along the wider dimension, so that many gradient updates are needed to reach the optimum.

Example 3-1 : Poorly conditioned objective

Consider a linear regression problem with MSE objective defined by two data points, $(x^{(1)}, y^{(1)}) = (1, 2)$ and $(x^{(2)}, y^{(2)}) = (10, 3)$. The optimal value of θ is $\theta^* = [\theta_0^*, \theta_1^*] = [1.89, 0.11]$, while the matrix $A = X \odot X^T$ has condition number $102.2/0.79 \approx 130$, mildly ill-conditioned.



Examining the objective J_{MSE} at right, we can see that the loss valley is much thinner in the θ_1 direction than θ_0. Intuitively, a small change ϵ in θ_1, e.g., $\theta_1 + \epsilon$, will change the predictions (particularly at $x^{(2)}$) much more than the same change to θ_0. We therefore must take small gradient steps to avoid increasing the objective – but then we make slow progress along the θ_0 directon. The figure shows the trajectory of a step size as high as is feasible, but which remains slow to converge toward the optimum in θ_0.

In this particular case, we can improve the conditioning of A by **normalizing** the data X before learning – usually, shifting the data to have zero mean and scaling them to unit variance. Poor conditioning can still appear, however, for example when two features i, j are highly correlated, or larger sets of features are nearly linearly dependent. Thus, although normalizing the data can help, it is not a panacea.

Polyak-Ruppert averaging. One simple technique that can increase robustness to the step size and allow for a slower rate of decrease in step size is to apply averaging of the optimization trajectory. In particular, we apply Algorithm 3.2, but add a step after updating θ, namely,

$$\bar{\theta} \leftarrow \bar{\theta} + \frac{1}{t}(\bar{\theta} - \theta)$$

This maintains in $\bar{\theta}$ a a running average of the values of θ during the course of the algorithm. We then return $\bar{\theta}$ as our estimate of the optimum, rather than the final θ.

Revisiting Example 3-1, maintaining a large step size will prevent convergence in the θ_1 direction, as each step hops over the minimum. However, the *average* of these values will quickly concentrate in between the hops. This allows the average to converge even while the step size remains high, allowing for more rapid progress in the θ_0 direction.

Momentum for gradient descent. Another way to try to improve gradient optimization behavior is to introduce an artificial **momentum** component to the trajectory. In particular, we replace the usual update to θ with:

$$G_t = \gamma G_{t-1} + \alpha \nabla J(\theta)$$
$$\theta \leftarrow \theta - G_t$$

[7] More generally, for a more complex loss function, we can envision looking at a second-order approximation to J around some nearby local optimum.

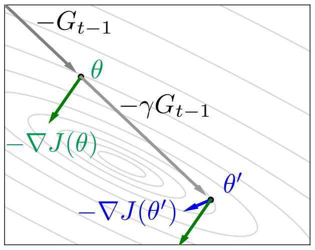
G_t is a geometrically weighted average of our past gradients; this means that our step goes in a direction that includes both the current fastest descent direction, but also the recent history of directions, smoothing out any oscillations in the gradient direction, while moving farther along directions in which the gradient has remained consistent. Intuitively, using momentum is analagous to rolling a ball downhill – the gradient provides a downward force (gravity), but the ball tends to keep going in the same direction until the gradient forces it to change.

Accelerated gradient. **?** proposed a method that improves even the asymptotic convergence rate of gradient descent. The key idea is to use momentum, but rather than adapt our trajectory using the gradient at the current position (analagously to a rolling ball), we instead predict our position a short ways into the future, and "pre-emptively" correct our direction. In particular, we replace the momentum update with:

$$G_t = \gamma G_{t-1} + \alpha \nabla J(\theta - \gamma G_{t-1})$$
$$\theta \leftarrow \theta - G_t$$

The value $\theta - \gamma G_{t-1}$ corresponds to our predicted position, and we use the gradient at this position to update, rather than at our current location θ. This makes the combination of the two terms look more like a momentum step, plus a correction from that point. **add convergence rate results?**

As a visualization, consider the trajectory at right. As we update our parameter value to θ, our previous gradients have been accumulated in the momentum term (gray vector). A standard momentum update evaluates the negative gradient at θ (green vector), and then updates θ using the gray plus green vectors. In contrast, accelerated gradient computes the negative gradient at $\theta' = \theta - \gamma G_t$ (blue vector), updating θ using the gray plus blue vectors. This gives a better correction to our direction, given that the momentum overshoots the nearby minimum.



Adaptive step sizes. In many machine learning models, it can be common that the different elements of θ may act on the prediction in very different ways. Some parameters may directly affect the output value, while others pass through many computations before reaching the prediction (see, for example, neural network models in Chapter 6). Even in simple linear regression, some features' distributions may mean that they are updated more or less often.

For example, suppose we would like to use linear regression to predict the average audience rating of a movie based on "basic information" about the movie – say, its genre, director, and lead actors. We decide to use binary indicator features (so, $x_1 = 1$ if the genre is Science Fiction, and zero otherwise; $x_2 = 1$ if the genre is Drama, and zero otherwise, etc.). Now, some of these features (such as the genre features) are far more frequently non-zero than others (say, a feature indicating whether Nicholas Cage is in a movie).

In this setting, it can be difficult to discover the usefulness of features that are highly informative, but often zero. To improve performance, **?** proposed the **AdaGrad** family of algorithms, in which each feature (or more generally, each parameter) is associated with a scaling term that can modulate its step size. Specifically, we maintain a sum of the squared gradient values in each dimension of θ:

$$S_t = S_{t-1} + (\nabla J(\theta) \cdot \nabla J(\theta))$$
$$\theta \leftarrow \theta - \alpha(S_t + \epsilon)^{-\frac{1}{2}} \cdot \nabla J(\theta)$$

where "·" is an element-wise product, so that S_t is a vector of the same size as ∇J which accumulates the elementwise squared values of the gradients $\nabla J(\theta)$. As this sum of squares increases, our gradient step size in each dimension of θ naturally decreases with the square root of the entries of S_t. A small constant ϵ is used to bound this denominator away from zero. This scaling causes parameters that have experienced many significant-magnitude gradient updates to have a smaller step size than parameters that have only infrequently had a significant gradient value.

AdaGrad can be combined with the exponential moving averages of the momentum technique, resulting in the adaptive moment estimation or **Adam** algorithm [**?**]. Adam computes the moving average of both the gradient (a momentum-like term) and the squared gradients (similar to AdaGrad) to select a parameter-specific step size:

$$\begin{aligned} G_t &= \gamma G_{t-1} + \nabla J(\theta) \\ S_t &= \beta S_{t-1} + (\nabla J(\theta) \cdot \nabla J(\theta)) \\ \theta &\leftarrow \theta - \alpha (S_t^{\frac{1}{2}} + \epsilon)^{-1} \cdot G_t \end{aligned}$$

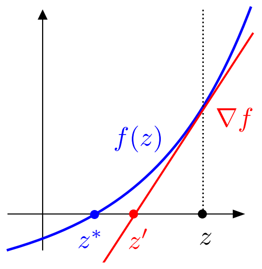
where "$\cdot$" is the elementwise product of the two vectors, and the square-root and inverse operations are similarly performed elementwise on the vector S_t.

The Adam algorithm is sufficiently effective to make it a popular black-box optimization technique. It has implementations available in TensorFlow [**?**] and PyTorch [**?**], and is a common choice for training many large-scale models, particularly large neural networks. Informally speaking, the idea of scaling the gradient can be viewed as a cheaper and easier to compute analogue to the class of second-order methods, which transform the gradient using the curvature of the function (the second derivatives of J) to speed up convergence.

Second-order methods

While gradient descent based methods are very powerful and easy to apply, more information about the loss J can often be used to speed up convergence and improve performance. *Second-order* methods evaluate or approximate the second derivative of J, and use this information to find a minimum more quickly. Just as the first derivative (gradient) gives us information about how rapidly J is changing, the second derivative (the curvature of J) tells us how rapidly the gradient is changing. A classic example is Newton's method.

Newton's method. Suppose that we want to find the root of some function f, i.e., z^* such that $f(z^*) = 0$. Newton's method for root finding works as follows: we initialize to some point z, and compute the tangent to f, $\nabla f(z)$. The tangent line crosses zero at



$$\nabla f(z) = \frac{0 - f(z)}{z' - z} \qquad \Rightarrow \qquad z' = z - \frac{f(z)}{\nabla f(z)},$$

so updating to z' should move us toward the root; see the figure at right for an illustration.

To use Newton's method for minimizing a loss function J, we want to find a root of $\nabla J(\theta)$:

$$\nabla\nabla J(\theta) = \frac{0 - \nabla J(\theta)}{\theta' - \theta} \qquad \Rightarrow \qquad \theta' = \theta - \frac{\nabla J(\theta)}{\nabla\nabla J(\theta)}$$

We can interpret this as a gradient step, with step size $\alpha = (\nabla\nabla J)^{-1}$, i.e., the inverse curvature of the loss J. Thus, if the derivative is changing very rapidly (high curvature), we will take small steps, while if the derivative is changing slowly we will take much larger steps. It is also easy to see that, if J is quadratic, this update will exactly minimize J in a single step; Newton's method can thus also be viewed as fitting a quadratic approximation to J at the current value of θ, and then updating θ to the minimum of that quadratic.

Newton's method is not guaranteed to converge, but often works well in practice and, when it converges, is often extremely fast. It works particularly well on nicely behaved and smooth functions; in the neighborhood around any minimum, such functions are usually locally quadratic, and thus once Newton's method gets near enough to the minimum it will converge very rapidly.

However, a drawback of Newton's method is its computation and storage in high dimensional optimization problems. For θ of dimension n, following the same derivation gives $\theta' = \theta - (\nabla J)(\nabla^2 J)^{-1}$ where $\nabla^2 J$ is the matrix of second derivatives, or **Hessian** of J. The Hessian has size $n \times n$, and so computing its inverse requires $O(n^{2.8}) \approx O(n^3)$ work. (For linear regression with MSE loss, a Newton update is exactly equivalent to solving via the pseudo-inverse.) Even computing and representing the Hessian matrix can be time- and memory-prohibitive for models with many parameters.

Quasi-Newton methods. **fill in** In large models, it is often impractical to compute, store, and use the full Hessian matrix. A class of algorithms called **quasi-Newton methods** maintain an approximation to the Hessian instead, which is then updated using only gradient information. Perhaps the most well-known quasi-Newton method is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm, or its "limited memory" variant, L-BFGS, which avoids representing the Hessian and thus can be used even for high-dimensional problems.

details

3.2 Increasing the number of features

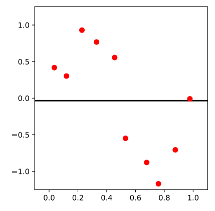
So far we have considered linear functions of several observed features, e.g., $[x_1, x_2]$. Suppose that we have only one feature, ($x = x_1$), but would like our predictor to be a *nonlinear* function of x, for example, $f(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$. We can simply define "new" features $x_2 = x^2$, $x_3 = x^3$, and so on (just as we defined $x_0 = 1 = x^0$), and this predictor becomes simply $f(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3$. In other words, our predictor still fits the linear regression form, but in a new "feature space" with additional features that are deterministic functions of our original observations. Applying our least-squares estimator gives a polynomial fit to the data, of the form

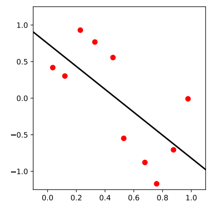
$$f(x) = \sum_{j=0}^{p} \theta_j \, x^j .$$

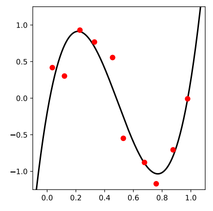
More generally, if we think that our target variable y is likely to be linearly related to some complex function of several observed variables, that combination can also be added as a new, observed feature.

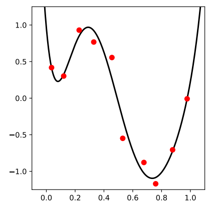
Example 3-2 : Fitting polynomial functions

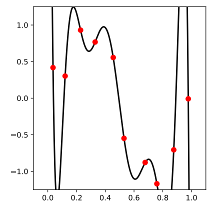
Given a small data set of $m = 10$ points, we can fit polynomials of various degrees p, from $p = 0$ (a constant predictor, $f(x;\theta) = \theta_0$), to $p = 9$ (i.e., $f(x;\theta) = \theta_0 + \theta_1 x + \ldots + \theta_9 x^9$).


$p = 0$


$p = 1$


$p = 3$


$p = 6$


$p = 9$

There are two equivalent ways to view this process. First, viewing f as a polynomial function of the scalar x, we see that we have created a "more complex" functional predictor, $\hat{y} = f(x)$. By making $f(x)$ more flexible, the set of functions we can represent is now larger (e.g., increasing from the set of all lines to the set of all cubic polynomials).

The alternative viewpoint is to forget that the features x_j are now deterministically related to one another, and to imagine that we have added a number of extra measurements (features) with which to predict y. From this perspective, we are learning a linear predictor from data, but those data lie in a higher dimensional space. (We will return to these two perspectives later, in classification.)

From this latter perspective, it is often useful to think of linear classifiers on (pre-specified) *feature transforms* of x. In other words, x may be the features we are originally given to solve our prediction problem, but we are free to transform them before using them, giving

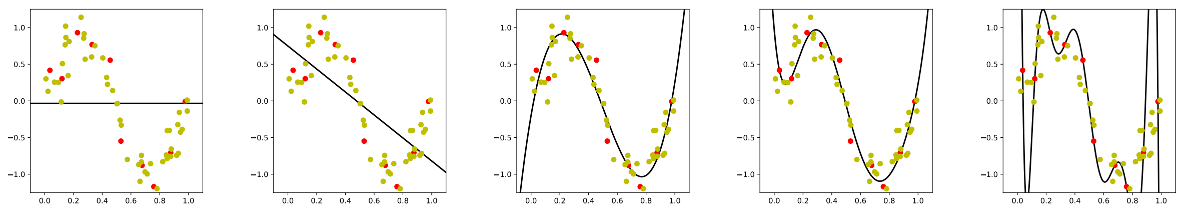
$$f(x) = \theta \odot \Phi(x)^T$$

where Φ is any transformation of x that might be helpful: scaling the data to a desired range of values, augmenting x to include the constant feature, or non-linear transformations of one feature or a combination of features. Our linear regression model will do well (predict accurately) if we have features $\Phi_j(x)$ which are linearly related to the target y – for example, a model for predicting real estate prices could depend on the total square footage of the building, which might be the product of several input features (e.g., length times width times number of floors).

3.3 Overfitting and Model Selection

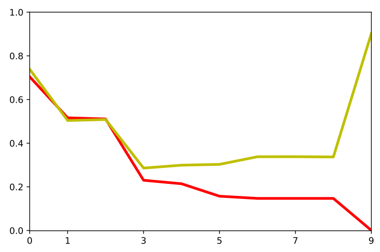
As we saw in the polynomial fits of Example 3-2, given 11 data points and a degree-10 polynomial (with 11 coefficients), we can predict all of our training data exactly! And yet, something about that predictor is not satisfying – it does not "look like" a good predictor. In fact, we have **overfit** to the data – we have chosen such a complex predictor that, although it is able to reconstruct the training data, we have no faith that it will accurately predict new data.

We can see the **generalization error** or **test error** of a predictor simply by gathering more data, and evaluating our cost function (e.g., the mean squared error) on those new points (shown in green). What we find is that, for very simple predictors ($p = 0$, $p = 1$), the performance on new test data is much like the performance on the training data. As the function gets more complex, however, the training error continues to decrease – but the test error does not. By $p = 9$, the training error is zero, but the test error has gotten worse.



$p = 0$ $p = 1$ $p = 3$ $p = 6$ $p = 9$

We can see this effect directly by plotting training and test error as a function of polynomial degree p. For very simple predictors, we are unable to capture the complexity of the dependence between x and y; but for overly complex predictors, we memorize the values of y at the expense of generalization.



Notionally, the "p" axis can be thought of as "complexity", and we find a similar curve whenever the complexity of our learner increases. We will see that much of the practical aspects of machine learning come down to choosing and controlling our position on this curve, increasing the complexity when under-fitting and decreasing it when overfitting.

Bias and Variance

A useful way to view the notion of overfitting is through the (frequentist) statistical concepts of bias and variance. Suppose that we train our model on a training data set D. Our training process will fit the model, finding a value for the parameters θ; since this depends on the data we used for training, we can explicitly write the trained parameters as $\theta(D)$.

Now, let us view our data set D as a random variable: a random subsample of examples we could have collected, each drawn from some true underlying distribution $p(x, y)$. Because D is random, $\theta(D)$ and thus the prediction $f(x; \theta(D))$ are as well. How much variability do we expect from different data sets? If f is a very simple function (such as a constant predictor), we would expect that it will not be too different, but may also not match the optimal predictor very closely (recall that, for squared error, the optimal $f^*(x) = \mathbb{E}[y|x]$). As we allow f to be more flexible, it becomes more likely to be able to represent $f^*(x)$, but may also exhibit more randomness due to D.

We can quantify these two concepts in terms of **bias** and **variance**. Intuitively, the bias measures the accuracy of $f(x; \theta(D))$ *on average* over draws of D, while the variance is the variance of $f(x)$ due to randomness in D:

$$\text{Bias}_f(x) = \bar{f}(x) - f^*(x) = \mathbb{E}_D\big[\, f(x; \theta(D))\big] - \mathbb{E}[y|x]$$

$$\text{Var}_f(x) = \mathbb{E}_D\Big[\big(f(x) - \bar{f}(x)\big)^2\Big] \qquad \text{where} \quad \bar{f}(x) = \mathbb{E}_D\big[f(x)\big].$$

Notice that, as expressed here, the bias and variance are functions of the location x to be predicted.

Let us see an explicit example of how bias and variance arise in a supervised regression problem. Suppose that, as a thought experiment, we imagine making five clones of ourselves to go out and collect training data for our regression model. Each of our clones collects a different training data set D, each drawn from $p(x, y)$, but differing in exactly what examples were collected. We denote each clone's data by a color (red, blue, green, purple, cyan). Then, each clone estimates the optimal parameters $\theta(D)$ using their training set, for several different models (polynomial fits with degree $p = 1$, 3, and 6). The predicted models found by each clone are shown in Figure 3.8.

We can see that the models are quite similar for $p = 1$, since even a few data are sufficient to estimate the linear fit's parameters, $[\theta_0, \theta_1]$, pretty well. However, the models obtained by performing a cubit fit ($p = 3$) show more variation (the red clone, in particular, predicts very differently on the right side of the plot), and our higher-order fit ($p = 6$) are far less similar to one another: the variance of f is increasing. (Note also that, not surprisingly, the models have less variation in the center of the data than at the edges.) However, although the linear fits are similar across all datasets (low variance), they are not able to accurately reflect the true relationship between x and y (the optimal f^*) – in other words, the linear model has higher bias than the more flexible models.

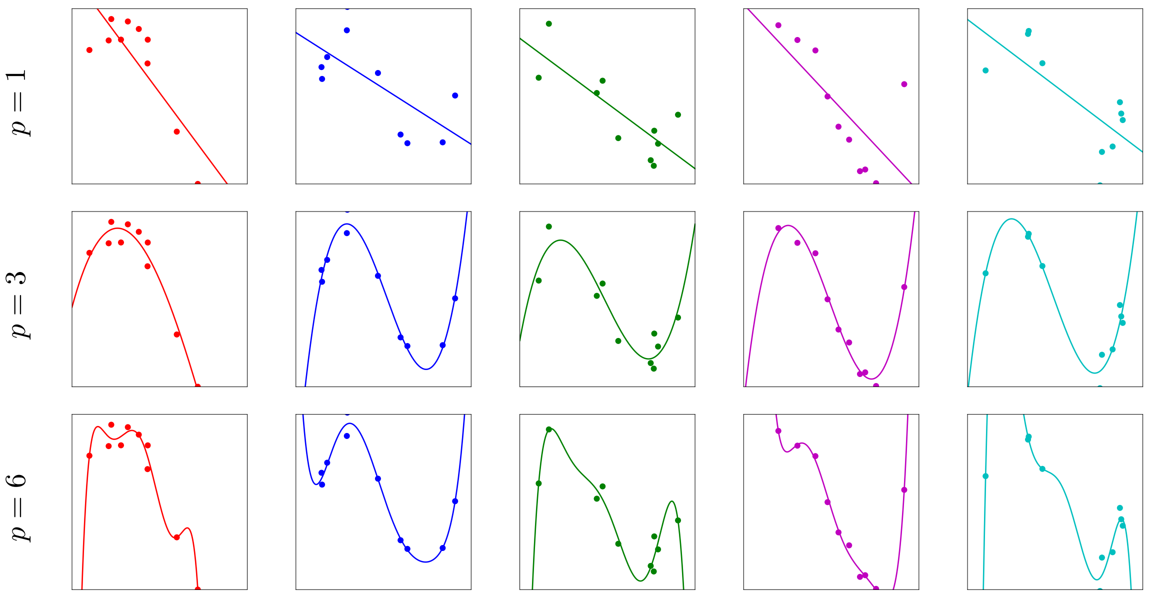
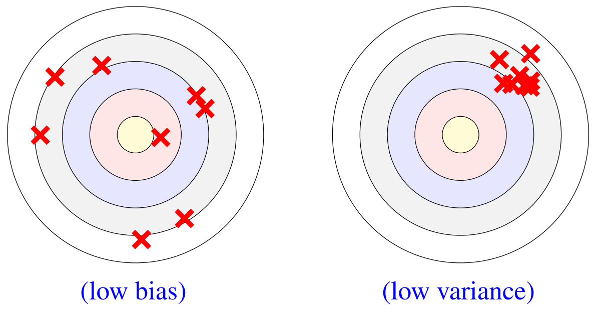


Figure 3.8: Variability of our polynomial predictor $f(x; \theta(D))$, using the minimum MSE parameters $\theta(D)$ estimated using the data set D, across five different draws of D (columns) and different polynomial degrees p (rows). At lower degrees, the trained model f is fairly similar across draws, while at higher degrees the predictions show significantly more variation.

Bias and variance are different types of errors, but both contribute to the potential for poor performance. In the targets at right, both estimators (red "x" markers) have the same overall error. However, in the first, the estimator has low variance (the markers are tightly clustered), but high bias (high average error). In contrast, on the second target, the estimators have low bias (their average is close to the center), but high variance.



More formally, consider the expected squared error of a regression model predicting a test point (x, y), in expectation over both the target point (x, y) and the training set D used to build the model. We can show that this decomposes into three parts:

$$\mathbb{E}_{D,x,y}\Big[\big(y - f(x;\theta(D))\big)^2\Big]$$
$$= \mathbb{E}_{x,y}\Big[(y - f^*(x))^2\Big] + \mathbb{E}_x\Big[(f^*(x) - \bar{f}(x))^2\Big] + \mathbb{E}_{D,x}\Big[(\bar{f}(x) - f(x;\theta(D)))^2\Big]$$

The first of these terms is the expected squared error of the optimal predictor, $f^*(x) = \mathbb{E}[y|x]$; this represents the minimum mean-squared error possible for *any* predictor. The second and third correspond to the squared bias, $\text{Bias}_f(x)^2$, and the variance, $\text{Var}_f(x)$, respectively (in expectation over x).

It is sometimes useful to talk about the amount of error we are experiencing due to bias and variance. In this case we can condense our bias and variance functions into scalar quantities:

$$\text{Bias} = \mathbb{E}_x\Big[\big(\text{Bias}_f(x)\big)^2\Big] \qquad\qquad \text{Var} = \mathbb{E}_x\Big[\text{Var}_f(x)\Big]$$

which assesses the size of these effects across the distribution of the test data $p(x, y)$.

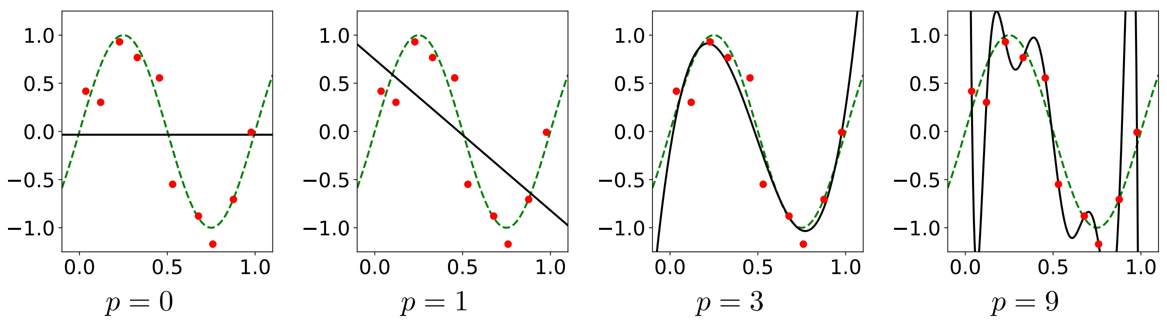
When the bias is the dominant factor in the error, this indicates that our model is underfitting – our model is insufficiently flexible to approximate f^*, even in expectation over the data set D. When variance is the dominant factor, the model is overfitting – our model is so flexible that different data sets D would produce very different prediction functions. Returning to our example, we can see that the flexible, degree $p = 3$ model fits its training data well, but does not fit the other data sets, as illustrated by the fact that each data set would choose a very different estimate of f. This type of behavior is a hallmark of overfitting.

Regularization

In the previous section, we saw that the more features we had available to our regression function f, the more propensity it had to overfit to the data. If we want to retain a large number of features, but avoid high variance, we will need to introduce other forms of complexity control to our model. One useful technique for controlling overfitting is **regularization**. Let us again consider polynomial regression; a key observation is that overfitting in this setting is usually also reflected in the parameters (coefficients) selected by the fitting process:

Example 3-3 : Sinusoidal data

Consider a set of training data (red points), whose true relationship between x and y is a sinusoidal function (green dashes). We wish to fit a regression $f(x)$ using polynomial functions of x. Note that no finite degree polynomial is able to perfectly represent $\sin(x)$, but that higher degrees allow us to come closer to $\sin(x)$ in theory. In practice, however, the small number of data mean that higher degree polynomials also overfit; by $p = 9$ we exactly reconstruct the training points:



This observed "overfitting" of the polynomial is also reflected in the parameters (coefficients) selected by the fitting process; for example, the estimated predictors (black lines) in each plot correspond to the following polynomials:

$p = 0$: $-.03$
$p = 1$: $0.7 \; -1.7\,x$
$p = 3$: $-.19 \; +12\,x \; -40\,x^2 \; +30\,x^3$
$p = 9$: $13 \; -712\,x +1263\,x^2 \; -107737\,x^3 +516391\,x^4 -1486497\,x^5 +2619787\,x^6 -2764356\,x^7 +1601922\,x^8 -391730\,x^9$

By $p = 9$, the coefficients have become extremely large, in order to pass through all ten data points.

From this viewpoint, a possible option to avoid overfitting is to allow ourselves to use high-degree polynomial functions, but not allow the optimization to select extremely large coefficients. We could thus arrive at a constrained minimization problem,

$$\theta^* = \min_\theta J(\theta) \qquad \text{subject to} \qquad \|\theta\|_2^2 = \sum_j |\theta_j|^2 \le C \tag{3.3}$$

where C controls how large we allow our coefficients to become by limiting the squared Euclidean length of the vector θ, $\|\theta\|_2^2$.

Rather than solve the constrained minimum MSE problem in (3.3), it is often easier to solve an equivalent, unconstrained optimization problem:

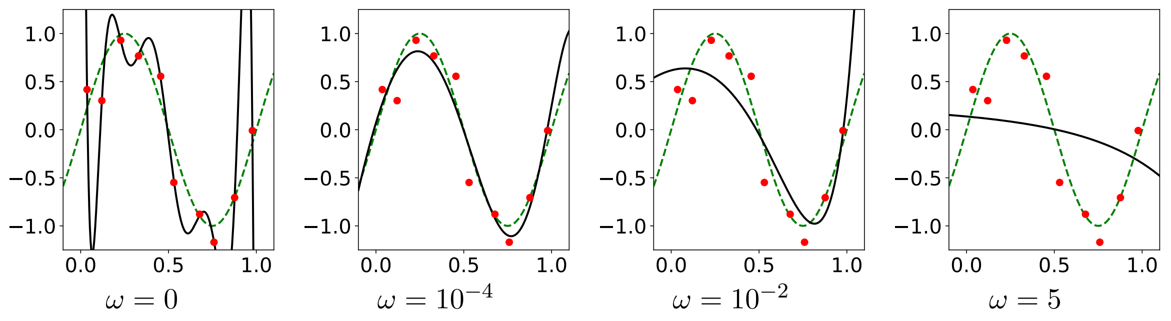
$$\theta^* = \min_\theta \;\; J(\theta) + \omega\|\theta\|_2^2 \tag{3.4}$$

in which we balance the "costs" of our loss on the data $J(\cdot)$, relative to the magnitude of the coefficients $\|\theta\|^2$, and ω controls the relative balance. If we take $\omega = 0$, this will correspond to the unconstrained problem (i.e., $C = \infty$), while if we increase ω sufficiently, we will ignore the data term J and simply choose all $\theta_j = 0$ (corresponding to $C = 0$).

The values of ω and C are related through the concept of *Lagrangian methods* for constrained optimization, discussed in more detail in Chapter 5 (section 5.2). Intuitively, suppose that we wish to solve the constrained form (3.3) for some C, but have code that solves the penalized form (3.4), producing $\theta^*(\omega)$ for any given ω. Now, for some $\omega \ge 0$, if $\|\theta^*(\omega)\|_2^2 > C$ (violating the desired constraint), we can simply increase the penalty ω and re-solve, giving a solution with smaller magnitude parameters. Conversely, if $\|\theta^*(\omega)\|_2^2 < C$, we can decrease ω, which will allow J to be reduced at the expense of increasing the parameter magnitude. We stop when either $\omega = 0$ and $\|\theta^*(\omega)\|_2^2 < C$, in which case C does not constrain the minimizing value of $J(\theta)$, or when $\|\theta^*(\omega)\|_2^2 = C$ (so that decreasing ω will violate the constraint, and increasing ω will constrain θ too much). This implies an implicit (and data-dependent) mapping between any $\omega \in [0, \infty)$ and any $C \in (0, \|\theta^*(\omega = 0)\|_2^2]$, so that either method can be used to control complexity in equivalent ways.

Example 3-4 : Sinusoidal data with regularization

Returning to our sinusoidal data from Example 3-3, let us continue to fit a high-degree polynomial ($p = 9$), but use increasing amounts of ℓ_2 regularization ω in our objective function to control the overfitting. When $\omega = 0$, we perform no regularization, and get the minimum MSE fit; as ω increases, we prioritize small coefficients θ_j over small reductions in the MSE.



As we increase ω, the optimal parameters θ^* become smaller:

$\omega = 0$:	13	$-712\,x$	$+1263\,x^2$	$-107737\,x^3$	$+516391\,x^4$	$-1486497\,x^5$	$+2619787\,x^6$	$-2764356\,x^7$	$+1601922\,x^8$	$-391730\,x^9$
$\omega = 10^{-4}$:	$.07$	$+6.6\,x$	$-12.8\,x^2$	$-9.8\,x^3$	$+4.2\,x^4$	$+11\,x^5$	$+10\,x^6$	$+4.3\,x^7$	$-2.8\,x^8$	$-9.8\,x^9$
$\omega = 5$:	0.13	$-0.19\,x$	$-0.15\,x^2$	$-0.11\,x^3$	$-0.07\,x^4$	$-0.04\,x^5$	$-0.02\,x^6$	$-0.01\,x^7$	$-0.01\,x^8$	$-0.0\,x^9$

When ω is large, we end up with very small coefficients, despite a resulting poor fit to the actual training examples; in this case, we say that we have over-regularized our learner.

We are also not restricted to choose the squared Euclidean length $\|\theta\|_2^2$ to constrain our parameters. A simple generalization is to replace the length with a more general norm function, such as the ℓ_p norm:

$$\ell_p(\theta) = \|\theta\|_p = \Big(\sum_j |\theta_j|^p\Big)^{\frac{1}{p}}$$

When $p = 2$, this corresponds to the usual Eulidean length or distance function. When $p = 1$, it is sometimes called the **Manhattan distance**, since in two dimensions it measures the sum of the north/south and east/west distances, as if measuring the distance we would need to travel if we walked on a rectangular grid of streets.

Note that it is common to constrain or penalize $\|\theta\|_p^p$, rather than the norm, since the two can be made equivalent by a simple adjustment of the constraint C or scaling ω; for example, if $\|\theta\|_p \le C$, then $\|\theta\|_p^p \le C^p$.

In general, $p \le 1$ encourages the vector θ to be *sparse*, so that some entries of θ will be exactly zero, indicating that these features are not used by the predictor. Notice $p \le 1$ are "sharp", so that moving even slightly off $\theta_j = 0$ incurs non-zero penalty. At one extreme, we have the limit,

$$\|\theta\|_0 = \lim_{p\to 0} \Big(\sum_j |\theta_j|^p\Big)^{\frac{1}{p}} = \sum_j \mathbb{1}[\theta_j \ne 0],$$

i.e., $\|\theta\|_0$ measures the number of non-zero parameters, and thus the constraint $\|\theta\|_0 \le C$ implies a *feature selection* criterion, that our predictor may use at most C of the available features in its prediction. However, since $\|\theta\|_0$ is not differentiable, we typically use some higher value of p. A popular choice is $p = 1$, since it both encourages some sparsity and is convex; for regression, performing an ℓ_1-regularized least-squares fit is commonly called the *Lasso* method Tibshirani [1996].

Conversely, when $p > 1$ the penality is flat near $\theta_j = 0$ (e.g., take the derivative heading toward zero), so that many small non-zero parameters will be penalized less than a few large-valued parameters. The ℓ_∞ norm,

$$\|\theta\|_\infty = \lim_{p\to\infty} \Big(\sum_j |\theta_j|^p\Big)^{\frac{1}{p}} = \max_j |\theta_j|,$$

measures the magnitude of the largest parameter, so that the constraint $\|\theta\|_\infty \le C$ enforces a bound on all coefficients: $\theta_j \in [-C, C]$.

Both ℓ_0 and ℓ_∞ have natural interpretations in terms of complexity control: ℓ_0 means we cannot use too many features, although we have flexibility in which features we use and what weights we assign them, while ℓ_∞ allows all features to be used but enforces that no one feature can have too much weight, or importance, in the prediction. But more generally, any form of regularization serves as a form of variance control. The regularization term encourages our model to take on a particular value that is independent of the actual observed data – for regularization by $\|\theta\|_p$, we encourage θ to be the all-zeros vector. This typically increases our bias: if our estimator was nearly unbiased before, $f^* \approx \bar{f}$, adding ℓ_p regularization will shift $\bar{f}$ toward the all-zero model. On the other hand, it decreases the variance of

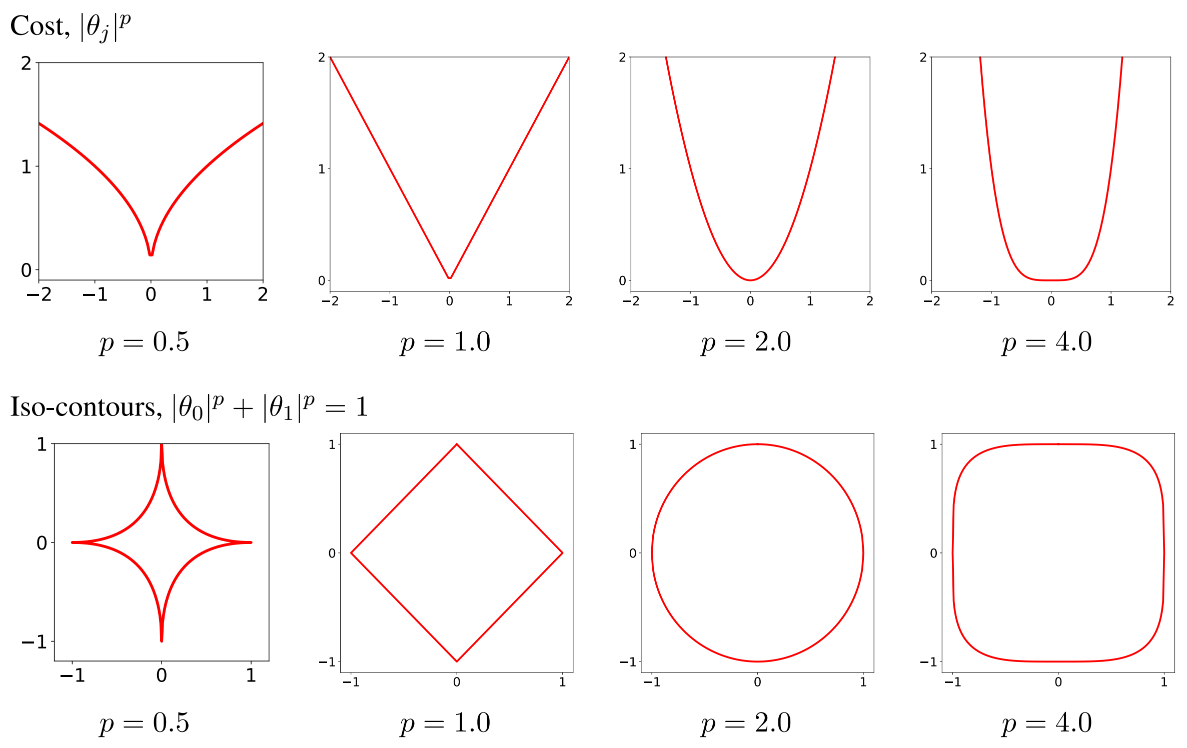


Figure 3.9: Regularization norms $\ell_p(\theta) = \left(\sum_i |\theta_i|^p\right)^{1/p}$ for various p. When p is large, small parameters ($|\theta_i| < 1$) cost significantly less than larger parameter values ($|\theta_i| > 1$); in contrast, when p is small, cost increases sharply around $\theta_i = 0$, then grows more slowly. For this reason, regularizing with small p (i.e., $p \leq 1$) tends to prefer sparse parameters, in which many of the θ_i are exactly zero, while $p > 1$ tends to prefer models in which there are no large parameter values.

our predictor, since for any data set D we are now closer to zero, making the model found with various draws of D more similar to one another. The stronger our regularization penalty ω, the more heightend this effect – if ω is sufficiently large, our optimization will essentially ignore the data D and return $\theta = 0$, resulting in zero variance. In frequentist statistics, this effect is sometimes called *shrinkage*, as it shrinks the value of the coefficients toward zero.

Usefully, however, this variance reduction holds no matter what value of θ we encourage. If we can encourage a value that is close to f^*, we can reduce the variance without significantly increasing the bias. For example, suppose we are learning a predictor for house prices in our area, using observable features. Although we may only have a small number of actual observations to use in learning our model, we may know some general relationships (price per square foot, or the impact of various upgrades) from data from other areas, or simply our own general background knowledge. If we regularize our model by encouraging it to be close to this "prior" model, e.g., by $\|\theta - \theta^0\|_2$, we may be able to reduce our model's variance without increasing its bias, giving a better overall predictor.

Different implementations of regularization may differ slightly in their details. For example, one common alternative form of the regularized estimator (3.4) is to scale by the number of data – noting that (in our convention) the loss J is an average over m data points, we might regularize by the factor $\frac{\omega}{m}$, instead of by ω. Obviously, this is simply a transformation of the regularization coefficient, but changes the effect of using the same value of ω on differently-sized data sets. If we regularize by $\frac{\omega}{m}$, we will find that as we acquire more data, the effects of our regularization (for fixed ω) are decreased; if we are given sufficiently many data for training, we are unlikely to overfit and may not need or want much regularization[8].

[8]This scaling defines a correspondence between regularization penalties for variance reduction in frequentist statistics, and the use of prior distributions and maximum *a posteriori* esitimation in Bayesian statistics.