1 Vector Calculus

a Single Variable Calculus Review

If you have learned calculus before, you probably started with single-variable calculus, involving functions like $y = f(x)$ that are only a function of a single scalar variable. Vector calculus involves applying those same rules to functions that depend on vector quantities, like $y = f(\vec{x})$. But before we get to the vector case, it's worth reviewing some basic concepts from single variable calculus, because they generalize nicely to higher dimensions.

a.1 Functions

Let's start with the basics and define exactly what we mean by a *function*.

> **(def 1.1) Function**
>
> A function is a mathematical objects that relates two quantities to each other. Usually, these quantities are an input x which is in the *domain* of the function, and an output or target $y = f(x)$ in the *image* or *codomain* of the function. Unless otherwise stated, we will assume that all functions have real inputs and outputs.

Consider a quadratic function

$$f(x) = x^2.$$

This function takes a scalar number as an input, and returns a scalar number as an output. We can therefore describe this function with the notation

$$f : \mathbb{R} \to \mathbb{R},$$

indicating that the function f takes a real number input and returns a real number output. If instead the function took an n-vector input and returned a scalar, we would describe it as

$$f : \mathbb{R}^n \to \mathbb{R}.$$

Can you think of a function that would fit that description? There are many, though the two that you are most familiar with from the Linear Algebra portion of the course might be the sum and dot product.

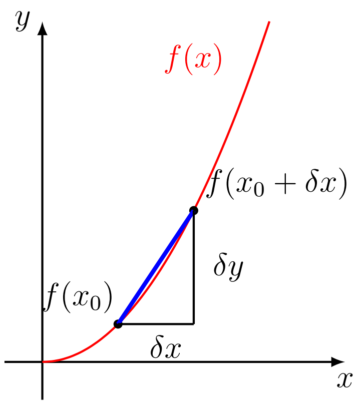

Figure 1: Linear approximation to the slope of a function

a.2 Derivatives

The derivative of a function tells us how quickly it changes at a particular point in its domain. In a discrete sense, we can approximate the derivative as the slope of the line between two points $f(x_0)$ and $f(x_0 + \delta x)$:

$$\frac{\delta y}{\delta x} = \frac{f(x_0+\delta x) - f(x_0)}{\delta x}. \tag{1.1}$$

Perhaps you remember the "rise over run" rule from high school mathematics. Here we rely on the same concept, as illustrated in Figure 1. The slope of the function $f(x)$ is not exactly the same as $\delta y/\delta x$, but it is a good approximation, and it would be a better approximation if δx was very small.

Taking that idea of δx being very small to its limit, we arrive at the definition of a *derivative*.

(def 1.2) Derivative

The derivative of a function $f(x)$ is defined as

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h} \tag{1.2}$$

In other words, the derivative is the slope of the line that is *tangent* to the function (i.e., the line in the same direction as the function that just barely touches the curve) at a given point x. Another way to think about it is that the derivative points in the direction of *steepest ascent* of $f(x)$, so that if you want to move along a function until you get to higher values, the derivative will point you in that direction.

But while Definition a.2 gives us nice intuition for what a derivative represents, it's not usually very helpful when it comes to calculating derivatives. For that, we turn to various rules which you ought to memorize.

a.3 Differentiation Rules

Denoting the derivative of a function f as f', we can state the following rules:

$$\text{Sum Rule: } (f(x)+g(x))' = f'(x)+g'(x)$$
$$\text{Product Rule: } (f(x)g(x))' = f'(x)g(x)+f(x)g'(x)$$
$$\text{Quotient Rule: } \left(\frac{f(x)}{g(x)}\right)' = \frac{g(x)f'(x)-f(x)g'(x)}{g(x)^2}$$
$$\text{Chain Rule: } g(f(x)) = f'(x)g'(f(x))$$

There are exhaustive lists you can find online with derivatives for common functions, but some that come up pretty often are:

- $f(x) = c \to f'(x) = 0$, for a constant c
- $f(x) = cx \to f'(x) = c$, for a constant c
- $f(x) = x^n \to f'(x) = nx^{n-1}$
- $f(x) = e^x \to f'(x) = e^x$
- $f(x) = \ln(x) \to f'(x) = 1/x$
- $f(x) = \sin(x) \to f'(x) = \cos(x)$
- $f(x) = \cos(x) \to f'(x) = -\sin(x)$

(ex 1.1) Differentiation Examples

Example 1: Find $f'(x)$ for $f(x) = 3x\sin(x)$.

Using the product rule and defining $f(x) = h(x)g(x)$, with $h(x) =$

$3x$ and $g(x) = \sin(x)$, we find:

$$f'(x) = 3\sin(x) + 3x\cos(x)$$

Example 2: Find $f'(x)$ for $f(x) = \ln(x^3)$

We'll use the chain rule here, the derivative of the inner function multiplied by the derivative of the outer function:

$$f'(x) = 3x^2\frac{1}{x^3} = \frac{3}{x}$$

b Partial Differentiation, Vector Functions, and Gradients

We will now introduce the "vector" part of vector calculus. The vector can be either the independent variable in a function, the function itself, or both. We'll address each of these cases individually, introducing new notation and terminology along the way.

b.1 Scalar-by-vector derivative

Consider a function $f(\vec{x}) : \mathbb{R}^n \to \mathbb{R}$ where $\vec{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$. In other words, f is a function of multiple variables (each of the components of $\vec{x}$ is an independent variable), but when you plug values in for each variable you get a scalar output.

For a concrete example, imagine that the vector $\vec{x} = (x, y)$ gives a latitude/longitude coordinate pair, and the function $f(\vec{x}) = p(x, y)$ is the atmospheric pressure at sea-level at a given latitude/longitude. Meteorologists are often interested in the spatial derivative of the atmospheric pressure, because that is one of the driving forces that makes the wind blow. But how do we define derivatives in this scenario with multiple independent variables, x and y? This relies on the concept of a *partial derivative.*

(def 1.3) Partial Derivative

The partial derivatives of a multivariable function $f(\vec{x})$ can be found

by differentiating with respect to each variable x_i individually, while treating all other variables $x_j, i \neq j$ as constants.
Example: Let $f(x,y) = \sin(x) + 3y^2$. The partial derivatives are given by:

$$\frac{\partial f}{\partial x} = \cos(x)$$
$$\frac{\partial f}{\partial y} = 6y$$

Once we have calculated all of the partial derivatives (i.e., the partial derivative with respect to each independent variable), we can define the *gradient* of a function.

(def 1.4) Gradient

The gradient of a multivariable function $f(\vec{x})$ is the collection of all partial derivatives,

$$\nabla_{\vec{x}} f = \left[\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n}\right]$$

We will follow the convention in MML that the gradient is a row vector of size $1 \times n$ for $\vec{x} \in \mathbb{R}^n$. This makes some matrix math easier.
Example 1: Let $f(x,y) = \sin(x) + 3y^2$. Defining $\vec{x} = (x,y)$, the gradient is given by:

$$\nabla_{\vec{x}} f = [\cos(x), 6y]$$

Example 2: Let $f(\vec{x}, \vec{b}) = \vec{b}^\top \vec{x}$. The gradient $\nabla_{\vec{x}} f$ can be seen by expanding the function:

$$f(\vec{x}, \vec{b}) = x_1 b_1 + x_2 b_2 + \cdots + x_n b_n$$

The gradient with respect to each x_i is its corresponding scalar multiplier b_i. Therefore, $\nabla_{\vec{x}} f = \vec{b}^\top$.

Differentiating with respect to a vector $\vec{x}$ introduces complications regarding the differentiation rules we saw in Section a.3, because when we deal with vectors and matrices, multiplication does not commute (and may not even be defined depending on the order of the operands). We therefore need these more general differentiation rules for the case of $\vec{x} \in \mathbb{R}^n$:

$$\text{Sum Rule: } \frac{\partial}{\partial \vec{x}}(f(\vec{x}) + g(\vec{x})) = \frac{\partial f}{\partial \vec{x}} + \frac{\partial g}{\partial \vec{x}}$$

$$\text{Product Rule: } \frac{\partial}{\partial \vec{x}}(f(\vec{x})g(\vec{x})) = \frac{\partial f}{\partial \vec{x}}g(\vec{x}) + f(\vec{x})\frac{\partial g}{\partial \vec{x}}$$

$$\text{Chain Rule: } \frac{\partial}{\partial \vec{x}}(g(f(\vec{x})) = \frac{\partial g}{\partial f}\frac{\partial f}{\partial \vec{x}}$$

b.2 Vector-by-scalar derivative

In the previous section, our functions took a vector input and produced a scalar output. Now we consider a *vector function*, which is simply a vector that has functions for each of its components.

Let's start with the simpler case and consider $\vec{f}(x) : \mathbb{R} \to \mathbb{R}^m$, where the independent variable x is a scalar but the output is an m-vector. For example, we might have a function $\vec{r}(t) = (x(t), y(t), z(t))$ that gives the 3D position (x, y, z) of a flying vehicle as a function of time t. If we are interested in the velocity of the vehicle at time t, that is defined as the derivative of the position with respect to time, and we can calculate it using the *vector-by-scalar derivative* rule.

(def 1.5) Vector-by-scalar derivative

The derivative of a vector function $\vec{f}(x)$ with $\vec{f} \in \mathbb{R}^m$ is given by the column vector

$$\frac{\mathrm{d}\vec{f}}{\mathrm{d}x} = \begin{bmatrix} \frac{\mathrm{d}f_1}{\mathrm{d}x} \\ \vdots \\ \frac{\mathrm{d}f_m}{\mathrm{d}x} \end{bmatrix}$$

Example: Let $x(t) = \sin(t)$, $y(t) = \cos(t)$, and $z(t) = t^2$ denote the 3D position of an object as a function of time when collected in the vector $\vec{r}(t) = (x, y, z)$. The derivative, which gives the velocity $\vec{v}$ at time t, is given by

$$\frac{\mathrm{d}\vec{r}}{\mathrm{d}t} = \vec{v}(t) = \begin{bmatrix} \cos(t) \\ -\sin(t) \\ 2t \end{bmatrix}$$

b.3 More on the chain rule

Of all the differentiation rules, the chain rule is the most important in machine learning applications (see Section d). An easy way to remember how it works is by thinking about derivatives like division, and letting partial differentials in the chain "cancel" each other (this isn't mathematically correct, but it is helpful).

For example, say we have a function $f(x) = c(h(g(x)))$. Then the derivative with respect to x is

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\mathrm{d}c}{\mathrm{d}h}\frac{\mathrm{d}h}{\mathrm{d}g}\frac{\mathrm{d}g}{\mathrm{d}x} \tag{1.3}$$

Let's now consider the results of the previous two sections and see an example of how the chain rule applies in the vector case.

(ex 1.2) Chain Rule

Consider $f(\vec{x}) = e^{x_1} + 2x_2^3$, with $x_1 = 2t$ and $x_2 = t^2$. What is the derivative $\frac{\mathrm{d}f}{\mathrm{d}t}$?

By the chain rule, we have

$$\frac{\mathrm{d}f}{\mathrm{d}t} = \frac{\mathrm{d}f}{\mathrm{d}\vec{x}}\frac{\mathrm{d}\vec{x}}{\mathrm{d}t}$$

The first term can be differentiated using the scalar-by-vector differentiation rule from Section b.1, i.e.,

$$\frac{\mathrm{d}f}{\mathrm{d}\vec{x}} = \nabla_{\vec{x}} f = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} \end{bmatrix}$$

The second term follows the vector-by-scalar rule from Section b.2,

$$\frac{\mathrm{d}\vec{x}}{\mathrm{d}t} = \begin{bmatrix} \frac{\partial x_1}{\partial t} \\ \frac{\partial x_2}{\partial t} \end{bmatrix}$$

These get multiplied together in *matrix multiplication* to produce

$$\frac{\mathrm{d}f}{\mathrm{d}t} = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} \end{bmatrix} \begin{bmatrix} \frac{\partial x_1}{\partial t} \\ \frac{\partial x_2}{\partial t} \end{bmatrix} = \frac{\partial f}{\partial x_1}\frac{\partial x_1}{\partial t} + \frac{\partial f}{\partial x_2}\frac{\partial x_2}{\partial t}$$

Plugging in numbers, we have:

$$\frac{\partial f}{\partial x_1} = e^{2t}$$
$$\frac{\partial x_1}{\partial t} = 2$$
$$\frac{\partial f}{\partial x_2} = 6t^4$$
$$\frac{\partial x_2}{\partial t} = 2t$$

Which results in:

$$\frac{\mathrm{d}f}{\mathrm{d}t} = 2e^{2t} + 12t^5$$

The example above highlights an application of the multivarible chain rule. We define that more generally below.

(def 1.6) Multivariable Chain Rule

Given a multivariable function $f(x, y)$ where $x = x(t)$ and $y = y(t)$, the derivative $\frac{\mathrm{d}f}{\mathrm{d}t}$ is given by

$$\frac{\mathrm{d}f}{\mathrm{d}t} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t}$$

This logic extends to any number of variables: the derivative with respect to the independent variable we seek is the sum of the chained partial derivatives of all of the intermediate variables.

b.4 Vector-by-vector derivative

The most complicated case combines the complications we addressed in the previous two sections. We now consider a vector function whose input is also a vector, i.e., $\vec{f}(\vec{x}) : \mathbb{R}^n \to \mathbb{R}^m$. For a vector $\vec{x} \in \mathbb{R}^n$, we can visualize the function as

$$\vec{f}(\vec{x}) = \begin{bmatrix} f_1(\vec{x}) \\ \vdots \\ f_m(\vec{x}) \end{bmatrix} \tag{1.4}$$

where each of the functions f_i maps $\mathbb{R}^n \to \mathbb{R}$. The gradient of such a function is called the *Jacobian.*

(def 1.7) Jacobian

The Jacobian of a vector function $\vec{f}(\vec{x})$, with $\vec{f} = (f_1(\vec{x}), \ldots, f_m(\vec{x}))$ and $\vec{x} \in \mathbb{R}^n$ is defined as the $m \times n$ matrix

$$J = \nabla_{\vec{x}} \vec{f} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \tag{1.5}$$

In other words, the element at row i and column j is given by $J(i,j) = \frac{\partial f_i}{\partial x_j}$.
Caution: The term Jacobian is often used to describe derivatives of any function, vector or scalar.
Example: Let A be an $m \times n$ matrix, $\vec{x} \in \mathbb{R}^n$ an n-vector, and $\vec{f}(\vec{x}) = A\vec{x} : \mathbb{R}^n \to \mathbb{R}^m$. What is the Jacobian $\nabla_{\vec{x}} \vec{f}$?

Recall that the matrix multiplication $A\vec{x}$ is equivalent to the dot product of $\vec{x}$ with each row of A, which we'll denote $\vec{r}_i$:

$$\vec{f} = A\vec{x} = \begin{bmatrix} \vec{r}_1 \cdot \vec{x} \\ \vdots \\ \vec{r}_m \cdot \vec{x} \end{bmatrix} \in \mathbb{R}^m$$

We can expand each component of $\vec{f}$ as

$$\begin{bmatrix} f_1(\vec{x}) \\ \vdots \\ f_m(\vec{x}) \end{bmatrix} = \begin{bmatrix} \vec{r}_1 \cdot \vec{x} \\ \vdots \\ \vec{r}_m \cdot \vec{x} \end{bmatrix} = \begin{bmatrix} A_{11}x_1 + A_{12}x_2 + \ldots A_{1n}x_n \\ \vdots \\ A_{m1}x_1 + A_{m2}x_2 + \ldots A_{mn}x_n \end{bmatrix}, \tag{1.6}$$

where A_{ij} denotes the component of A at row i and column j.

Now look at Equation 1.5, and consider the first row of the right-most matrix in Equation 1.6. The derivatives we seek are terms like...

$$\begin{aligned} \frac{\partial f_1}{\partial x_1} &= A_{11} \\ \frac{\partial f_1}{\partial x_2} &= A_{12} \\ &\vdots \\ \frac{\partial f_m}{\partial x_n} &= A_{mn} \end{aligned}$$

Therefore, the Jacobian of $\vec{f} = A\vec{x}$ is $\nabla_{\vec{x}}\vec{f} = A$, which looks much like the single variable derivative of a function like $f(x) = ax$!

And one more nice identity before moving on.

(def 1.8) Gradient of a quadratic form

The expression $\vec{x}^\top A\vec{x}$ is often referred to as a *quadratic form.* The gradient of this expression, when A is a symmetric matrix, is given by

$$\nabla_{\vec{x}}\vec{x}^\top A\vec{x} = 2A\vec{x}. \tag{1.7}$$

We won't prove this, but you can use it whenever it is helpful.

c Gradients of Matrices

c.1 Dimensionality

Believe it or not, we can end up with Jacobian matrices with more than two dimensions. We call these higher-dimensional objects *tensors*, and they can result from, e.g., taking a gradient of a matrix with respect to another matrix.

(ex 1.3) Matrix-matrix gradient

Consider a matrix $A \in \mathbb{R}^{m\times n}$ and a matrix $B \in \mathbb{R}^{p\times q}$. The Jacobian matrix would be a 4-dimensional tensor with components

$$J_{ijkl} = \frac{\partial A_{ij}}{B_{kl}}$$

In general, you can find the dimension of the Jacobian matrix by adding together the dimensions of the objects involved in the differentiation. This leads to a more general view of the derivatives we have examined so far:

- A scalar-by-vector derivative like the gradient $\nabla_{\vec{x}} f$ will end up as a 1D vector (one dimension from $\vec{x}$, zero from f)
- A vector-by-scalar derivative like $\frac{\mathrm{d}\vec{f}}{\mathrm{d}x}$ will end up as a 1D vector (one dimension from $\vec{f}$, zero from x)
- A vector-by-vector derivative like the gradient $\nabla_{\vec{x}} \vec{f}$ will end up as a 2D matrix (one dimension from $\vec{x}$, one from $\vec{f}$).
- And on and on, up to matrix-by-vector (3D Jacobian), matrix-by-matrix (4D Jacobian), and tensor-by-tensor (ND Jacobian).

c.2 Useful Rules

We will now present a few of the most commonly-used rules for taking gradients with respect to matrices. We won't use them much in this class, but this will hopefully serve as a helpful reference for future use.

Consider a function $f : \mathbb{R}^{m\times n} \to \mathbb{R}$ which takes a matrix A and maps it to a scalar. We can define the derivative of f with respect to the matrix A as

$$\nabla_A f = \begin{bmatrix} \frac{\partial f}{\partial A_{11}} & \cdots & \frac{\partial f}{\partial A_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial A_{m1}} & \cdots & \frac{\partial f}{\partial A_{mn}} \end{bmatrix} \tag{1.8}$$

Lots of the properties needed for matrix derivatives involve the tracer operator, which for an $n \times n$ diagonal matrix A is defined as $\mathrm{Tr}(A) = \sum_{i=1}^{n} A_{ii}$

(i.e., the sum of the diagonal elements). Some more nice properties of the trace for square matrices, before we even get to derivatives, are:

- $\operatorname{Tr}(AB) = \operatorname{Tr}(BA)$
- $\operatorname{Tr}(ABC) = \operatorname{Tr}(CAB) = \operatorname{Tr}(BCA)$
- $\operatorname{Tr}(ABCD) = \operatorname{Tr}(DABC) = \operatorname{Tr}(CDAB) = \operatorname{Tr}(BCDA)$
- $\operatorname{Tr}(A) = \operatorname{Tr}\left(A^\top\right)$
- $\operatorname{Tr}(A + B) = \operatorname{Tr}(A) + \operatorname{Tr}(B)$
- $\operatorname{Tr}(aA) = a \operatorname{Tr}(A)$

And properties involving the derivative with respect to A:

- $\nabla_A \operatorname{Tr}(AB) = B^\top$
- $\nabla_{A^\top} f(A) = (\nabla_A f(A))^\top$
- $\nabla_A \operatorname{Tr}\left(ABA^\top C\right) = CAB + C^\top AB^\top$
- $\nabla_A \det(A) = \det(A)(A^{-1})^\top$

Section 5.5 of MML gives another list of useful matrix gradient properties.

d Backpropagation and Automatic Differentiation

The preceding sections were filled with various rules. This section is about the application of those rules to backpropagation, an algorithm is used for training deep neural networks.

d.1 Background on deep neural networks

Say we are developing a regression model. We have one sample with n features in an array $\vec{x} = (x_1, x_2, \ldots, x_n)$.

A linear regression model might look like:

$$\hat{y} = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$$

where w_i are the weights (i.e., slopes or coefficients) and b is the bias (i.e., intercept).

Now let's add a non-linearity to this model:

- $\hat{y}$ defined above will become an intermediate output z.
- z will be fed into a nonlinear function $\max(z, 0)$, which we call ReLU (rectified linear unit).
- The output ReLU(z) will be defined as a.

All told, this results in a prediction:

$$a = \text{ReLU}(z) = \text{ReLU}(w_1x_1 + w_2x_2 + \cdots + w_nx_n + b)$$

Now, let's imagine we build p of these nonlinear models, so we can generalize the above to:

$$a_j = \text{ReLU}(z_j) = \text{ReLU}(w_{j1}x_1 + w_{j2}x_2 + \cdots + w_{jn}x_n + b_j)$$

for $j = 1, 2, \ldots, p$.

Finally, with our p outputs from p nonlinear models, we can map to a single prediction $\hat{y}$ with another set of coefficients w and b.

- We'll denote these with a superscript [2] to indicate that they are in the second layer.
- Similarly, we'll append a [1] to our previous w and b coefficients.

This results in:

$$\hat{y} = w_1^{[2]}a_1^{[2]} + w_2^{[2]}a_2^{[2]} + \cdots + w_p^{[2]}a_p^{[2]} + b^{[2]} \tag{1.9}$$

where each $a_j^{[2]}$, $j = 1, 2, \ldots, p$, is given by:

$$a_j^{[2]} = \text{ReLU}(z_j^{[1]}) = \text{ReLU}(w_{j1}^{[1]}x_1 + w_{j2}^{[1]}x_2 + \cdots + w_{jn}^{[1]}x_n + b_j^{[1]})$$

Terminology:

- We would call this a two-layer neural network because we require two sets of weights, $w^{[1]}, b^{[1]}$ and $w^{[2]}, b^{[2]}$.
- $w^{[2]}, b^{[2]}$ are associated with the output layer.
- $w^{[1]}, b^{[1]}$ are associated with the hidden layer.

  - A neural network can have many hidden layers; these are called "deep" neural nets.
  - Each hidden layer has an associated activation function, which is generally nonlinear. Here we used ReLU, the most popular, but there are others.

- Each of the a_j in the hidden layer is called a **neuron** or a **unit**.
- The hidden layer size = p, the number of units in the layer.

For this example, we simplified $\vec{x}$ to be one sample with n features. Usually, X has m samples with n features each (i.e., it's an $m \times n$ matrix). This means that our weights W will be matrices, biases b will be vectors, and all the scalar algebra we wrote before is generalized to matrix algebra.

d.2 Backpropagation

When we make a prediction like $\hat{y}$ in Equation 1.9 using a neural network, we define the *loss* or the *cost* J associated with that prediction based on how far away it is from the true values y that we are trying to predict. A common cost function is the squared loss,

$$J(\theta) = \|\hat{y}(\theta) - y\|^2, \tag{1.10}$$

where $\theta = \{W, b\}$ is the set of all weights and biases. Depending on the dimension of the data in the problem (how many samples, how large each sample is), W and b can be scalars, vectors, matrices, or even higher-dimensional tensors. In this section, we'll just leave it general; the important thing is that the cost J is a scalar value, which as we know from the preceding sections, can be readily differentiated with respect to higher-dimensional objects like vectors and matrices.

Think back to calculus and how we find the minimum value of a function: if our goal is to minimize the cost function $J(\theta)$, we can accomplish that by differentiating J with respect to θ, and then finding θ (i.e., W and b values) in each layer that make $\frac{\partial J}{\partial \theta}$ as small as possible.

How does chain rule come into play? If we have K layers in our neural network, then our prediction is the composition of K nonlinear functions, i.e.,

$$\hat{y} = f_K(f_{K-1}(\dots(f_1(\theta_0, x_0))) \tag{1.11}$$

Here, each f_i takes as its argument the output of the previous layer, going all the way back to the input data x_0 and input layer parameters θ_0. Connecting this to our notation from the previous section but denoting the layer number with a subscript rather than a superscript,

$$f_i(z_{i-1}(\theta_{i-1})) = a_i = \sigma(z_{i-1}(\theta_{i-1})) = \sigma(W_{i-1}x_{i-1} + b_{i-1}),$$

where σ is our nonlinear activation function, often a ReLU.

Because the goal is to find the derivative of J with respect to all θ_j for $j = 0, \ldots, K-1$, we can apply the chain rule to Equation 1.11 as:

$$\frac{\partial J}{\partial \theta_{K-1}} = \frac{\partial J}{\partial f_K}\frac{\partial f_k}{\partial \theta_{K-1}} \tag{1.12}$$

$$\frac{\partial J}{\partial \theta_{K-2}} = \frac{\partial J}{\partial f_K}\frac{\partial f_K}{\partial f_{K-1}}\frac{\partial f_{K-1}}{\partial \theta_{K-2}} \tag{1.13}$$

$$\frac{\partial J}{\partial \theta_{K-3}} = \frac{\partial J}{\partial f_K}\frac{\partial f_K}{\partial f_{K-1}}\frac{\partial f_{K-1}}{\partial f_{K-2}}\frac{\partial f_{K-2}}{\partial \theta_{K-3}} \tag{1.14}$$

etc., or in general:

$$\frac{\partial J}{\partial \theta_j} = \frac{\partial J}{\partial f_K}\frac{\partial f_K}{\partial f_{K-1}}\frac{\partial f_{K-1}}{\partial f_{K-2}}\cdots\frac{\partial f_{j+2}}{\partial f_{j+1}}\frac{\partial f_{j+1}}{\partial \theta_j} \tag{1.15}$$

Not only does the chain rule allow us to calculate the gradient of J with respect to all parameters, it allows us to do it quickly: Note that most of the terms in Equation 1.14 appeared in a previous equation for the derivative with respect to parameters further up the chain.

d.3 Automatic Differentiation

Backpropagation is a version of a more general algorithm called Automatic Differentiation. In automatic differentiation, we capitalize on the fact that complicated functions can be broken down into simple components, and the derivative of the complicated function can be expressed by applying the chain rule to the derivatives of the simple components.

Let's look at an example from the MML textbook (I think their derivation is sort of confusing, so I'll reproduce it here in a simpler form). We have a function

$$f(x) = \sqrt{x^2 + \exp(x^2)} + \cos\Big(x^2 + \exp\big(x^2\big)\Big), \tag{1.16}$$

and we want to find the derivative with respect to x. We can define the intermediate variables

$$\begin{aligned}
a &= x^2 \\
b &= \exp(a) \\
c &= a + b \\
d &= \sqrt{c} \\
e &= \cos(c) \\
f &= d + e
\end{aligned}$$

We will now define the gradient of the output, f, with respect to each of the intermediate variables. The easiest way is to start at the final definition of f and work backwards towards your first initial variable, a. From the multivariate chain rule, any intermediate variable that appears in multiple other intermediate variables requires summation.

$$\begin{aligned}
\frac{\partial f}{\partial e} &= 1 \\
\frac{\partial f}{\partial d} &= 1 \\
\frac{\partial f}{\partial c} &= \frac{\partial f}{\partial d}\frac{\partial d}{\partial c} + \frac{\partial f}{\partial e}\frac{\partial e}{\partial c} = (1)(\frac{c^{-1/2}}{2}) - (1)\sin(c) = \frac{1}{2\sqrt{c}} - \sin(c) \\
\frac{\partial f}{\partial b} &= \frac{\partial f}{\partial c}\frac{\partial c}{\partial b} = \frac{\partial f}{\partial c}(1) \\
\frac{\partial f}{\partial a} &= \frac{\partial f}{\partial b}\frac{\partial b}{\partial a} + \frac{\partial f}{\partial c}\frac{\partial c}{\partial a} = \frac{\partial f}{\partial b}\exp(a) + \frac{\partial f}{\partial c}(1)
\end{aligned}$$

Now we are ready to calculate $\frac{\mathrm{d}f}{\mathrm{d}x}$. Because x only appears in one of the intermediate expressions, all we need to do is:

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial x} = \frac{\partial f}{\partial a}(2x). \tag{1.17}$$

Plugging in the values from above,

$$
\begin{aligned}
&\frac{\mathrm{d}f}{\mathrm{d}x} \\
&= \frac{\partial f}{\partial a}(2x) \\
&= (2x)\left(\frac{\partial f}{\partial b}\exp(a) + \frac{\partial f}{\partial c}\right) \\
&= (2x)\left(\exp(a)\left(\frac{1}{2\sqrt{c}} - \sin(c)\right) + \left(\frac{1}{2\sqrt{c}} - \sin(c)\right)\right) \\
&= 2x\left(\frac{1}{2\sqrt{c}} - \sin(c)\right)(1 + \exp(a)) \\
&= 2x\left(\frac{1}{2\sqrt{x^2 + \exp(x^2)}} - \sin\left(x^2 + \exp\left(x^2\right)\right)\right)\left(1 + \exp\left(x^2\right)\right)
\end{aligned}
$$

2 Optimization

In machine learning and data science, our goal is usually to find the set of model parameters that makes a model best match our data. In this section, we will quantify what we mean by "best" and introduce a class of techniques, broadly referred to as optimization methods, that will allow us to find the best parameters.

a Mathematical Optimization Problems

An optimization problem is any problem of the form

$$
\begin{array}{ll}
\text{minimize} & f_0(\vec{x}) \\
\text{subject to} & f_i(\vec{x}) \le b_i, \text{ for } i = 1, \dots, m
\end{array} \tag{2.1}
$$

In Equation 2.1, $\vec{x}$ is the *optimization variable* of the problem, $f_0 : \mathbb{R}^n \to \mathbb{R}$ is the *cost function* or *objective function*, and the functions $f_i : \mathbb{R}^n \to \mathbb{R}$ are the *constraint functions*, which depend on the limits or bounds b_i. A solution to the optimization problem is given by the *optimal* vector x^* that has the smallest value of $f_0(\vec{x})$ among all the vectors that satisfy the constraints.

As formulated in Equation 2.1, optimization problems can be very general. The components of the optimization variable $\vec{x}$ could represent practically anything. Some common applications for optimization problems could be:

- Allocating holdings of n different stocks in a portfolio, subject to constraints on total available capital for investments.
- Finding optimal locations for n different warehouses in a supply chain, subject to restrictions on maximum distance between any two warehouses.
- Determining the thrust force that a rocket booster engine should produce at n times during flight, subject to constraints on total fuel available.
- Power output of a municipal power plant at n times during the day, subject to constraints on total power available and minimum supply requirements for certain regions.

Our application will be rather specific: the optimization problem that we consider will be the problem of finding optimal parameters, stored in a vector $\vec{\theta}$, that minimize a cost function $f_0(\vec{\theta}) : \mathbb{R}^n \to \mathbb{R}$, with the cost function f_0 describing the error in a data-driven model. In mathematical terms:

$$\begin{aligned} &\min_{\vec{\theta}} f_0(\vec{\theta}) \\ &\text{subject to } f_i(\vec{\theta}) \leq b_i, \text{ for } i = 1, \ldots, m \end{aligned} \tag{2.2}$$

The notation on the first line means "find $\vec{\theta}$ that minimizes the objective function." A specific example of the type of problem described by Equation 2.2 is Linear Regression, where we have a matrix $A \in \mathbb{R}^{m \times n}$ containing m samples of n features, a target vector $\vec{b} \in \mathbb{R}^m$, and unknown parameters $\vec{\theta} \in \mathbb{R}^n$. In this case, the objective function is the squared 2-norm of the residual given by $\|A\vec{\theta} - \vec{b}\|^2$ and we do not specify any constraints:

$$\min_{\vec{\theta}} \|A\vec{\theta} - \vec{b}\|^2 \tag{2.3}$$

The problems we look at in this class will not generally involve constraints, but they can certainly arise: for example, you might want the intercept of your linear regression to be non-negative, or you may want to restrict your model from predicting values that are unphysically high or low.

b One-dimensional Optimization

For one-dimensional functions, we can often solve Equation 2.1 analytically (i.e., we can find an exact solution, rather than a numerical approximation). The example from MML is instructive so we reproduce it here.

(ex 2.1) One-dimensional Optimization

Consider the one-dimensional (or single-variable) objective function

$$f_0(x) = x^4 + 7x^3 + 5x^2 - 17x + 3 \tag{2.4}$$

Our goal is to find the scalar value of x that minimizes $f(x)$. One way to do this is to calculate the derivative with respect to x and find any location that corresponds to zero slope. This will indicate a maximum or minimum in the function. Differentiating:

$$\frac{\mathrm{d}f_0}{\mathrm{d}x} = 4x^3 + 21x^2 + 10x - 17 = 0 \tag{2.5}$$

We can now plot both the objective function and its derivative to see where the minimum value might be. Anywhere that the derivative crosses the x axis would be a candidate, and it looks like the crossing near $x = -4.5$ has the lowest value (it's really -4.48... but we won't worry about finding the exact solution).

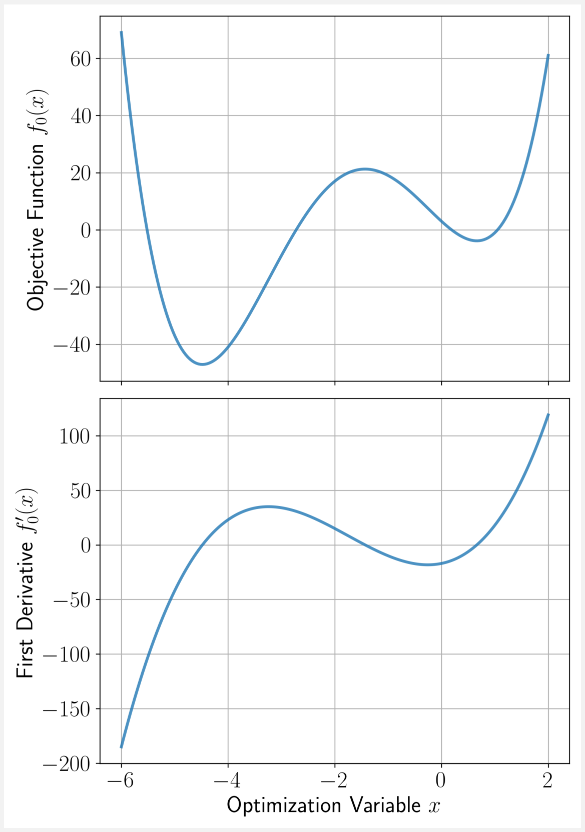

Figure 2: The objective function and its first derivative

The tricky part about this problem is that there are 3 locations, called critical points, where $f_0'(x) = 0$: $x = -4.5$, $x = -1.4$, and $x = 0.7$, but only one of those locations is the *global minimum*. The critical point near $x = -1.4$ is actually a local maximum, and if we hadn't plotted it, we would only be able to determine that by the sign of the second derivative, which gives the curvature of the function:

- Positive curvature $f_0''(x) > 0$ at the critical point indicates a local minimum.
- Negative curvature $f_0''(x) < 0$ at the critical point indicates a local maximum.

In summary: we seek $x = x^*$ such that $f_0(x)$ is minimized, and we find that by finding the smallest $f_0(x)$ value where $f_0'(x) = 0$ and $f_0''(x) > 0$.

c Higher-dimensional Problems and Gradient Descent

For general problems of the form

$$\min_{\vec{\theta}} f_0(\vec{\theta}), \tag{2.6}$$

we won't be able to find an analytical solution for the values of θ that minimize f_0. This is especially true if $\vec{\theta}$ is high-dimensional, with hundreds, thousands, or even millions of parameters that need to be fit. However, if we assume that f_0 is at least differentiable, then we can use an algorithm called *gradient descent* to find an approximate solution to $\vec{\theta}$.

The basic idea is this: start with an initial guess for $\vec{\theta}$ and call it $\vec{\theta}_0$. Then repeatedly update $\vec{\theta}$ to $\vec{\theta}_1, \vec{\theta}_2$, etc., in a way that makes $f_0(\vec{\theta}_k)$ progressively smaller with each update, until hopefully we converge on a value of $\vec{\theta}$ that minimizes f_0. But how do we update $\vec{\theta}$?

Recall that the gradient of a function with respect to a particular variable can be interpreted geometrically as the direction of steepest ascent. This extends to an arbitrary number of dimensions, but it is easiest to conceptualize in 3D. Picture yourself in the bowl in Figure 3, where the value of the objective function $f(x, y) = x^2 + y^2$ gets larger towards the top, and smaller near the bottom of the bowl. The gradient of the function at various locations (black arrows) point in the direction of steepest ascent, orthogonal to the contour lines (or level sets) of the function.

If the gradient points in the direction of steepest ascent, then we want to go in the opposite direction in order to find the minimum value of the function. This leads us to our update rule for gradient descent:

$$\vec{\theta}_{k+1} = \vec{\theta}_k - \gamma_k \nabla_{\vec{\theta}} f_0(\vec{\theta}_k), \tag{2.7}$$

where γ_k is the *step size* or *learning rate* for our update that determines how far we move down the gradient with each step. We will illustrate how this works with an example.

(ex 2.2) Gradient Descent

Consider the quadratic function

$$f(\vec{x}) = \left(\frac{x_1 - a}{4}\right)^2 + \left(\frac{x_2 - b}{3}\right)^2 \tag{2.8}$$

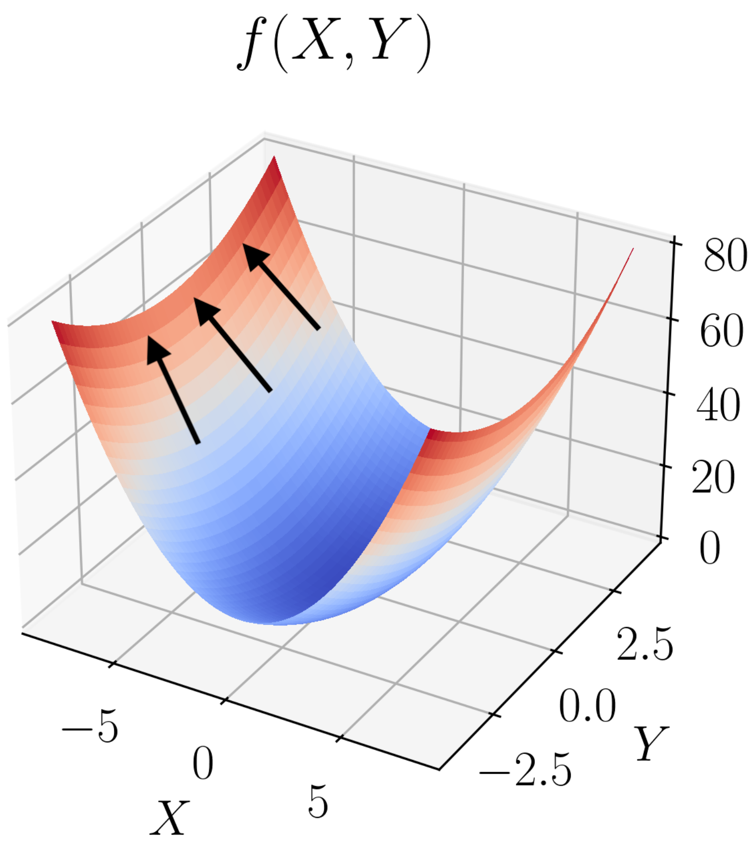

Figure 3

You may be able to see that this function is minimized for $\vec{x} = (x_1, x_2) = (a, b)$, but let's solve it instead using gradient descent. The following Python code implements a solution:

```python
# Defining our function over some domain
x1 = np.linspace(-10, 10, 100)
x2 = np.linspace(-10, 10, 100)
X1, X2 = np.meshgrid(x1, x2)
a = 2
b = -3
def fxy(x1, x2, a, b):
  # Defines quadratic f(x1,x2)
  f = ((x1 - a)/4)**2 + ((x2 - b)/3)**2
  return f

x_guess = np.array([8, 9])  # Initial guess
cost_reduction = np.inf
tolerance = 1e-6  # convergence tolerance
learning_rate = 1e-2
x_history = [x_guess]

while cost_reduction > tolerance:
  # cost with guess from previous iteration
  cost_pre = fxy(x_guess[0], x_guess[1], a, b)

  # gradient of the cost from the
  # partial derivatives evaluated at x_guess
  dfdx1 = (x_guess[0] - a) / 2
  dfdx2 = 2 * (x_guess[1] - b) / 3
  gradient = np.array([dfdx1, dfdx2])

  # Updating x_guess
  x_guess = x_guess - learning_rate * gradient

  # Calculating new cost
  cost_post = fxy(x_guess[0], x_guess[1], a, b)
  x_history.append(x_guess)
```

```
# How much has the cost gone down
cost_reduction = np.abs(cost_post - cost_pre)
```

Once the algorithm converges, the value in `x_guess` is very close to the function's true minimum: `array([ 2.00002289, -2.9997428 ])`

We can see the convergence behavior for different choices of learning rate. The smaller the learning rate, the more iterations it takes to converge, but the smoother the behavior. A larger learning rate will be "jumpy" but ultimately converges much more quickly. In the next section, we will learn about more advanced methods that perform better than using a constant learning rate.

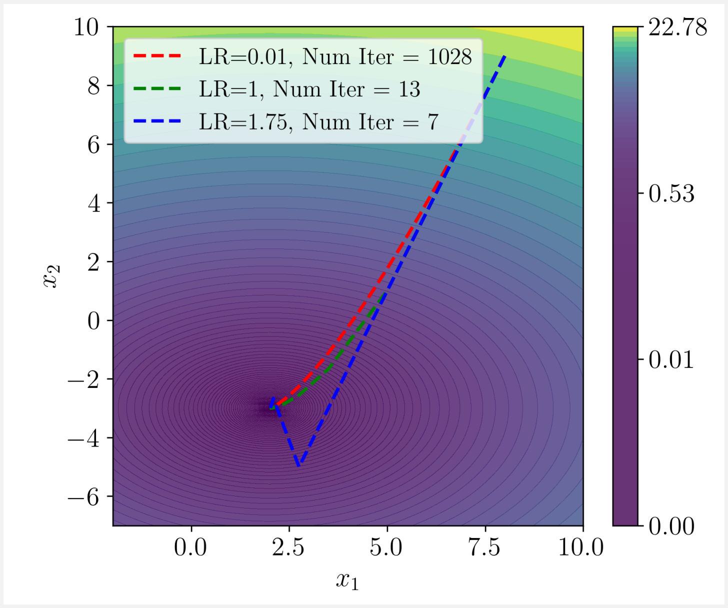

Figure 4: Gradient descent convergence for different learning rates

d Other Flavors of Gradient Descent

The standard approach to gradient descent in Equation 2.7 is sometimes called "vanilla" gradient descent, implying that there are other, more interesting flavors available. This section will cover those alternatives.

d.1 Gradient Descent with Momentum

As we saw in the Python example above, the nature and speed of convergence for gradient descent depends on the learning rate. As an alternative to using a constant value, one technique called *gradient descent with momentum* adapts the learning rate dynamically during the optimization procedure. Mathematically, this looks like

$$\vec{\theta}_{k+1} = \vec{\theta}_k - \gamma_k z_{k+1}$$
$$z_{k+1} = \alpha z_k + \nabla_{\vec{\theta}} f_0(\vec{\theta}_k)$$

What's going on here?

- We have a new paramter α. If $\alpha = 0$, we recover vanilla gradient descent (Equation 2.7).
- If $\alpha > 0$ (it's usually around $\alpha = 0.9$), then our update becomes a linear combination of the gradient at the current step and the gradient at the previous step.

This consideration of past gradients turns out to be extremely helpful by reducing the influence of local minima in the objective function. The analogy is to imagine a heavy ball rolling down a hill – the more momentum it has, the less likely it is to get stuck in a small valley, and the more likely it is to power through to a global minimum further down the hill. In Figure 5, we see convergence with 5 times fewer iterations by using momentum with $\alpha = 0.8$.

d.2 Stochastic Gradient Descent

In the context of machine learning, we seek to minimize the objective function evaluated on all of the examples in our training dataset. One way to do this is to add up the gradients for samples x_i for $i = 1, \ldots, m$ during gradient descent, i.e.,

$$\nabla_{\vec{\theta}} f_0(\vec{x}, \vec{\theta}) = \sum_{i=1}^{m} \nabla_{\vec{\theta}} f_0(\vec{x}_i, \vec{\theta}) \tag{2.9}$$

This is often called "batch" gradient descent, and it can get expensive computationally if you have a large number of samples and/or gradients are difficult to compute.

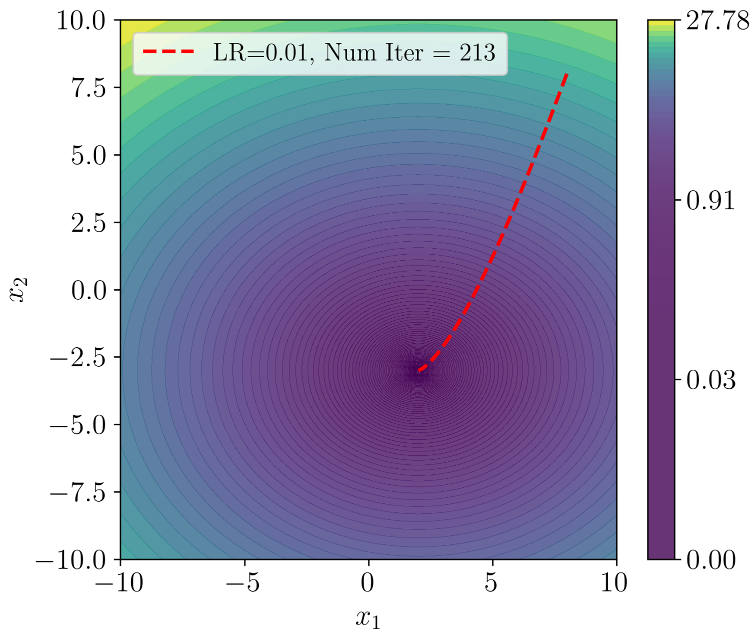

Figure 5

An alternative to the batch approach is stochastic gradient descent. Instead of computing the gradients with consideration of all m samples before making a parameter update, you instead choose a subset of the samples of size $s < m$:

$$\nabla_{\vec{\theta}} f_0(\vec{x}, \vec{\theta}) \approx \sum_{i=1}^{s} \nabla_{\vec{\theta}} f_0(\vec{x}_i, \vec{\theta}) \tag{2.10}$$

In some cases, a single randomly selected sample at each iteration is sufficient ($s = 1$). The gradients on the subsampled data will differ from the true full-batch gradient for a given iteration, but the approximation will be *unbiased*, meaning that the over- and under-estimated gradients will cancel each other out on average, and the algorithm will converge to the optimal solution regardless.

d.3 Advanced Optimizers

These days, most machine learning models are trained using advanced versions of gradient descent like AdaGrad (Duchi, 2011) or Adam (Kingma & Ba, 2014). We won't worry about the details in this class, but know that machine learning packages like Pytorch and Tensorflow have built-in implementations of these methods so that you can easily use them for training your models.

e Convex Optimization and Beyond

Many machine learning algorithms are formulated as a special type of optimization problem called a *convex optimization* problem, which has the form

$$\begin{array}{ll} \text{minimize} & f_0(\vec{x}) \\ \text{subject to} & f_i(\vec{x}) \leq 0, \ \text{for } i = 1, \ldots, m \\ & \vec{a}_i^\top \vec{x} = b_i, \ \text{for } i = 1, \ldots, p. \end{array} \tag{2.11}$$

As usual, f_0 is the objective function and $f_1, \ldots, f_m$ are the inequality constraints, and we add equality constraints via the affine functions $\vec{a}_i^\top \vec{x} = b_i$. This looks very similar to Equation 2.1, but we now add the restrictions that $f_0, \ldots, f_m$ are convex functions and that the intersection of the domains of all f_i is a convex set. We don't have time in this class to learn all of the conditions for what makes a function or a set convex, but we will cover the very basics and learn why convex optimization is so useful.

e.1 Convex functions

In a convex function, any local minimum is a global minimum. This is an extremely desirable property in an objective function, because it means we don't have to worry about situations like in Figure 2 where there are multiple local minima. Gradient descent won't get "stuck" in a local minimum, it will just ride all the way to the bottom of the hill.

But what makes a function convex? We can envision a simple convex function graphically in Figure 6. Note the following properties that indicate convexity:

- Given a function $f(x)$, a straight line drawn from any point $f(x_1)$ on the function to any other point $f(x_2)$ on the function will be above the function. Mathematically, this is stated as

$$\theta f(x_1) + (1-\theta) f(x_2) \geq f(\theta x_1 + (1-\theta) x_2), \tag{2.12}$$

for $0 \leq \theta \leq 1$. This is often called *Jensen's inequality.*

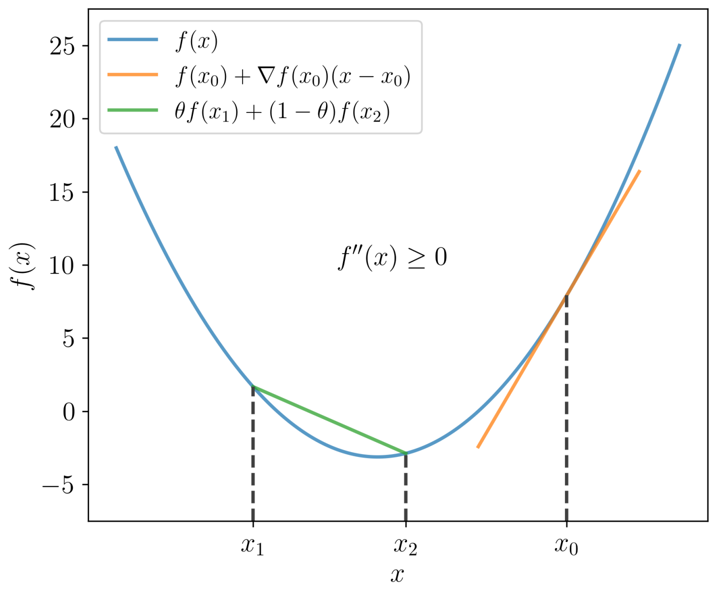

Figure 6: A convex function $f(x)$ with the various conditions for convexity

- A tangent line drawn anywhere on the function $f(x)$ will be below the function everywhere. Mathematically,

$$f(x) \geq f(x_0) + \nabla_x f(x_0)(x - x_0), \tag{2.13}$$

where the right-hand side is simply the first-order Taylor approximation of f near x_0, defining the tangent line at $x = x_0$.

- The curvature of the function will be nonnegative everywhere, i.e.,

$$f''(x) \geq 0. \tag{2.14}$$

For a function of a vector variable $f(\vec{x})$, this condition generalizes to the Hessian matrix being positive semidefinite, i.e.,

$$H = \nabla^2_{\vec{x}} f(\vec{x}) \succeq 0 \tag{2.15}$$

e.2 Convex sets

Convex optimization problems require not only convex objective and constraint functions; those functions must also be defined over a domain $\mathcal{C}$ that is a convex set. Thus, the set of possible solutions x^* will be a convex set.

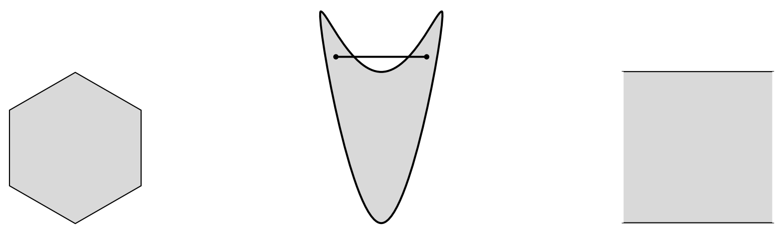

Figure 7: Some simple convex and nonconvex sets. *Left.* The hexagon, which includes its boundary (shown darker), is convex. *Middle.* The Starfleet Insignia type shape is not convex, since the line segment between the two points in the set shown as dots is not contained in the set. *Right.* The square contains some boundary points but not others, and is not convex. Figure adapted from Boyd and Vandenberghe, 2004.

We can visualize a convex set graphically in Figure 7. The property to remember that makes a set convex is that a straight line drawn from any point in the set to any other point in the set will remain in the set. Mathematically,

$$\theta x_1 + (1-\theta)x_2 \in \mathcal{C}, \tag{2.16}$$

for $x_1, x_2 \in \mathcal{C}$ and $0 \leq \theta \leq 1$.

f Tools for General Nonlinear Optimization

Sometimes you have a cost function that you need to minimize. You don't know if it is convex, and you don't have time to find out; you just know that you need to find a better set of parameters to make the cost lower. Luckily, you have options. The following example illustrates how to use `scipy.optimize.minimize` to minimize a general multivariable function.

(ex 2.3) scipy.optimize.minimize

Say we are given the objective function,

$$f_0(\vec{x}) : \mathbb{R}^{50} \to \mathbb{R} = \sum_{i=1}^{50} \left(0.1(x_i - i)^2 + \sin(x_i) \cdot \cos(x_{i+1}) + e^{-x_i^2}\right), \tag{2.17}$$

with the constraints

$$x_i \leq 1, i = 1, \ldots, 10$$
$$x_i \leq 20, i = 11, \ldots, 50$$

Our goal is to solve for the vector $\vec{x}$ that minimizes $f_0(\vec{x})$ subject to the constraints. The following Python code solves the problem:

```
import numpy as np
from scipy.optimize import minimize

def objective_function(x):
  return sum(
  0.1 * (x[i] - (i + 1)) ** 2
  + np.sin(x[i]) * np.cos(x[(i + 1) % 50])
  + np.exp(-x[i] ** 2)
  for i in range(50)
  )


# Initial guess for a vector of 50 elements
initial_guess = np.random.rand(50)

# Bounds
bounds = [(-np.inf, 1) for i in range(10)]
bounds += [(-np.inf, 20) for i in range(10, 50)]

# Call minimize
result = minimize(
  objective_function,
  x0=initial_guess,
  bounds=bounds
)
```

The optimal value of $\vec{x}$ will be stored as a numpy array in `result.x`.