Mathematics for Machine Learning

Ryan Kingery

Table of contents

Preface

This is my book Mathematics for Machine Learning.

1 Basic Math

You can understand machine learning at an intuitive level without knowing much math at all. In fact, you can get pretty far this way. People without strong math backgrounds build ML products, win Kaggle competitions, and write new ML frameworks all the time. However, at some point you may find yourself *really* wanting to understand how the algorithms work at a deeper level, and if you want to do that, you'll need to learn some math. Not a *huge* amount of math, but the basics of a several fundamental topics for sure. My plan in the next series of lessons is to get you up to speed on this "minimum viable math" you'll need to proceed further.

I'll start this sequence by reviewing math you've probably seen before in high school or college. Such topics include things like elementary arithmetic, algebra, functions, and multivariate functions. I'll also present the Greek alphabet since it's helpful to be able to read and write many of these letters in machine learning. Let's get started.

```
from utils.math_ml import *
```

1.1 Elementary Math

It's useful in machine learning to be able to read and manipulate basic arithmetic and algebraic equations, particularly when reading research papers, blog posts, or documentation. I won't go into depth on the basics of high school arithmetic and algebra. I do have to assume *some* mathematical maturity of the reader, and this seems like a good place to draw the line. I'll just mention a few key points.

1.1.0.1 Numbers

Recall that numbers can come in several forms. We can have,

- **Natural Numbers:** These are positive whole numbers $0, 1, 2, 3, 4, \cdots$. Note the inclusion of 0 in this group. Following the computer science convention I'll tend to do that. The set of all natural numbers is denoted by the symbol $\mathbb{N}$.
- **Integers:** These are any whole numbers $\cdots, -2, -1, 0, 1, 2, \cdots$, positive, negative, and zero. The set of all integers is denoted by the symbol $\mathbb{Z}$.

- **Rational Numbers:** These are any ratios of integers, for example

$$\frac{1}{2}, \frac{5}{4}, -\frac{3}{4}, \frac{1000}{999}, \cdots.$$

  Any ratio will do, so long as the *denominator* (the bottom number) is not zero. The set of all rational numbers is denoted by the symbol $\mathbb{Q}$.
- **Real Numbers:** These are any arbitrary decimal numbers on the number line, for example

$$1.00,\ 5.07956,\ -0.99999\ldots,\ \pi = 3.1415\ldots,\ e = 2.718\ldots,\ \cdots.$$

  They include as a special case both the integers and the rational numbers, but also include numbers that can't be represented as fractions, like π and e. The set of all real numbers is denoted by the symbol $\mathbb{R}$.
- **Complex numbers:** These are numbers with both real and imaginary parts, like $1+2i$ where $i = \sqrt{-1}$. Complex numbers include the real numbers as a special case. Since they don't really show up in machine learning we won't deal with these after this. The set of all complex numbers is denoted by the symbol $\mathbb{C}$.

1.1.0.2 Basic Algebra

You should be familiar with the usual arithmetic operations defined on these systems of numbers. Things like addition, subtraction, multiplication, and division. You should also at least vaguely recall the order of operations, which defines the order in which complex arithmetic operations with parenthesis are carried out. For example,

$$(5+1)\cdot\frac{(7-3)^2}{2} = 6\cdot\frac{4^2}{2} = 6\cdot\frac{16}{2} = 6\cdot 8 = 48.$$

You should be able to manipulate and simplify simple fractions by hand. For example,

$$\frac{3}{7}+\frac{1}{5} = \frac{3\cdot 5+1\cdot 7}{7\cdot 5} = \frac{22}{35} \approx 0.62857.$$

As far as basic algebra goes, you should be familiar with algebraic expressions like $x+5=7$ and be able to solve for the unknown variable x,

$$x = 7-5 = 2.$$

You should be able to take an equation like $ax+b=c$ and solve it for x in terms of coefficients a, b, c,

$$\begin{aligned} ax + b &= c \\ ax &= c - b \\ x &= \frac{c-b}{a}. \end{aligned}$$

You should also be able to expand simple expressions like this,

$$\begin{aligned} (ax-b)^2 &= (ax-b)(ax-b) \\ &= (ax)^2 - (ax)b - b(ax) + b^2 \\ &= a^2x^2 - abx - abx + b^2 \\ &= a^2x^2 - 2abx + b^2. \end{aligned}$$

1.1.0.3 Sets and Intervals

It's also worth recalling what a set is. Briefly, a **set** is a collection of *unique* **elements**. Usually those elements are numbers. To say that an element x is an element of a set S, we'd write $x \in S$, read "x is in S". If x is *not* in the set, we'd write $x \notin S$. For example, the set of elements $1, 2, 3$ can be denoted $S = \{1, 2, 3\}$. Then $1 \in S$, but $5 \notin S$.

I've already mentioned the most common sets we'll care about, namely the natural numbers $\mathbb{N}$, integers $\mathbb{Z}$, rational numbers $\mathbb{Q}$, and real numbers $\mathbb{R}$. Also of interest will be the **intervals**,

- Open interval: $(a, b) = \{x : a < x < b\}$.
- Half-open left interval: $(a, b] = \{x : a < x \leq b\}$.
- Half-open right interval: $[a, b) = \{x : a \leq x < b\}$.
- Closed interval: $[a, b] = \{x : a \leq x \leq b\}$.

Think of intervals as representing line segments on the real line, connecting a to b. I'll touch on sets more in coming lessons. I just want you to be familiar with the notation, since I'll occasionally use it.

1.1.1 Symbolic vs Numerical Computation

There are two fundamental ways to perform mathematical computations: numerical computation, and symbolic computation. You're familiar with both even though you may not realize it. **Numerical computation** involves crunching numbers. You plug in numbers, and get out numbers. When you type something like `10.5 / 12.4` in python, it will return a number, like `0.8467741935483871`. This is numerical computation.

```
10.5 / 12.4
```

```
0.8467741935483871
```

This contrasts with a way of doing computations that you learned in math class, where you manipulate symbols. This is called **symbolic computation**. Expanding an equation like $(ax - b)^2$ to get $a^2x^2 - 2abx + b^2$ is an example of a symbolic computation. You see the presence of abstract variables like x that don't have a set numeric value.

Usually in practice we're interested in numerical computations. We'll mostly be doing that in this book. But sometimes, when working with equations, we'll need to do symbolic computations as well. Fortunately, python has a library called SymPy, or sympy, that can do symbolic computation automatically. I won't use it a whole lot in this book, but it will be convenient in a few places to show you that you don't need to manipulate mathematical expressions by hand all the time.

To use sympy, I'll import `sympy` with the alias `sp`. Before defining a function to operate on, we first have to encode all the symbols in the problem as sympy `Symbol` objects. Once that's done, we can create equations out of them and perform mathematical operations.

Here's an example of using sympy to expand the equation above, $(ax - b)^2$.

```
import sympy as sp
```

```
a = sp.Symbol('a')
b = sp.Symbol('b')
x = sp.Symbol('x')
a, b, x
```

```
(a, b, x)
```

```
equation = (a * x - b) ** 2
expanded = sp.expand(equation, x)
print(f'expanded equation: {expanded}')
```

```
expanded equation: a**2*x**2 - 2*a*b*x + b**2
```

We can also use sympy to solve equations. Here's an example of solving the quadratic equation $x^2 = 6$ for its two roots, $x = \pm\sqrt{6}$.

```
equation = x**2 - 6
solutions = sp.solve(equation, x)
print(f'solutions = {solutions}')
```

```
solutions = [-sqrt(6), sqrt(6)]
```

Sympy has a lot of functionality, and it can be a very difficult library to learn due to its often strange syntax for things. Since we won't really need it all that often I'll skip the in depth tutorial. See the documentation if you're interested.

1.2 Univariate Functions

As I'm sure you've seen before, a mathematical function is a way to map inputs x to outputs y. That is, a function $f(x)$ is a mapping that takes in a value x and maps it to a unique value $y = f(x)$. These values can be either single numbers (called **scalars**), or multiple numbers (vectors or tensors). When x and y are both scalars, $f(x)$ is called a **univariate function**.

Let's quickly cover some of the common functions you'd have seen before in a math class, focusing mainly on the ones that show up in machine learning. I'll also cover a couple machine-learning specific functions you perhaps haven't seen before.

1.2.1 Affine Functions

The most basic functions to be aware of are the straight-line functions: constant functions, linear functions, and affine functions:

- Constant functions: $y = c$ or $x = c$
  - Examples: $y = 2$, $x = 1$
- Linear functions: $y = ax$
  - Examples: $y = -x$, $y = 5x$
- Affine functions: $y = ax + b$
  - Examples: $y = -x + 1$, $y = 5x - 4$

All constant functions are linear functions, and all linear functions are affine functions. In the case of affine functions, the value b is called the **intercept**. It corresponds to the value where the function crosses the y-axis. The value a is called the **slope**. It corresponds to the steepness of the curve, i.e. its height over its width (or "rise" over "run"). Notice linear functions are the special case where the intercept is *always* the origin $x = 0, y = 0$.

1.2.1.1 Plotting

We can plot these and any other univariate function $y = f(x)$ in the usual way you learned about in school. We sample a lot of (x, y) pairs from the function, and plot them on a grid with a horizontal x-axis and vertical y-axis.

Before plotting some examples I need to mention that plotting in python is usually done with the `matplotlib` library. Typically what we'd do to get a very simple plot is:

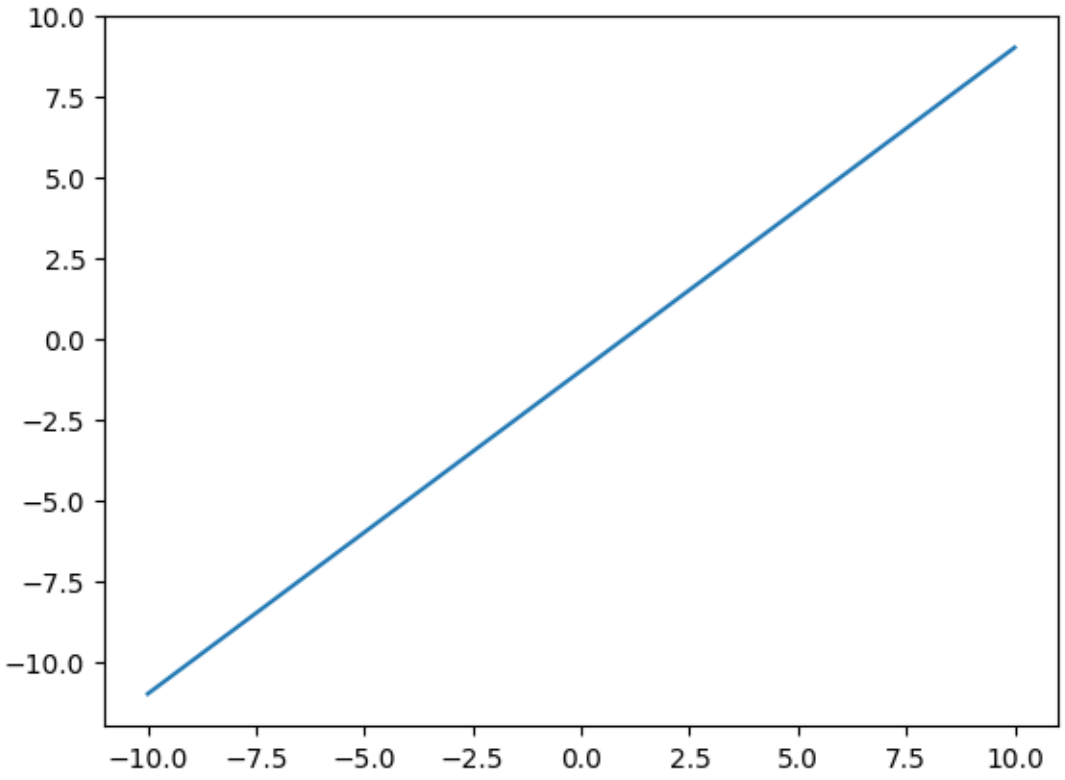
1. Import `plt`, which is the alias to the submodule `matplotlib.pyplot`
2. Get a grid of `x` values we want to plot, e.g. using `np.linspace` or `np.arange`
3. Get a grid of `y` values either directly, or by first defining a python function `f(x)`
4. Plot x vs y by calling `plt.(x, y)`, followed by `plt.show()`.

Note step (2) requires another library called `numpy` to create the grid of points. You don't *have* to use numpy for this, but it's typically easiest. Usually numpy is imported with the alias `np`. Numpy is python's main library for working with numerical arrays. We'll cover it in much more detail in future lessons.

Let me go ahead and load these libraries. I'll also show a simple example of a plot. What I'll do is define a grid `x` of 100 equally spaced points between -10 and 10, and plot the function $y = x - 1$ using the method described above.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
x = np.linspace(-10, 10, 100)
y = x - 1
plt.plot(x, y)
plt.show()
```

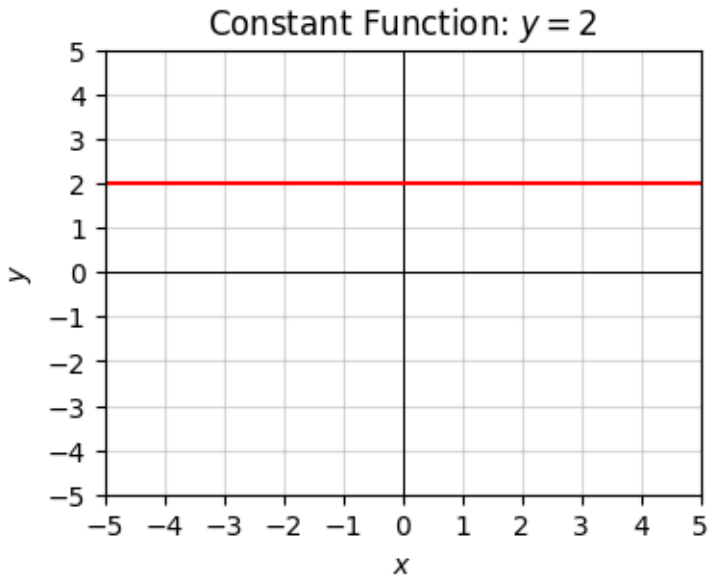


This plot is pretty ugly. It's too big, arbitrarily scaled, and doesn't include any information about what's being plotted against what. In matplotlib if you want to include all these things to make nice plots you have to include a bunch of extra style commands.

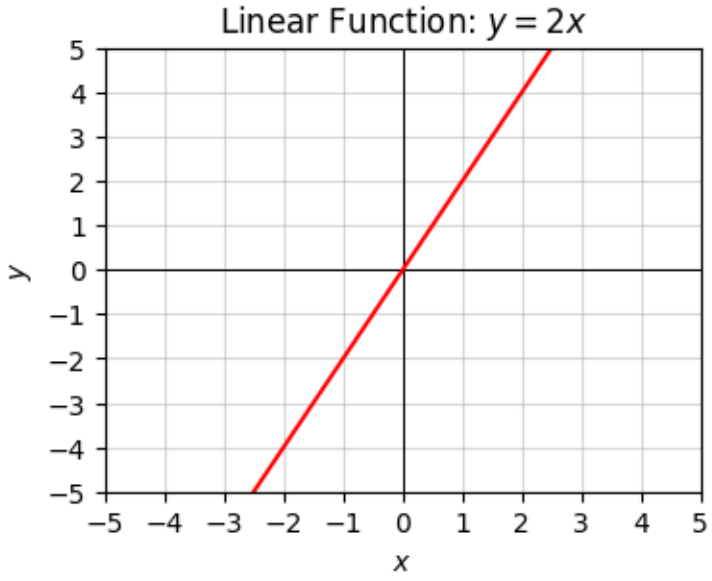
For this reason, for the rest of the plotting in this lesson I'm going to use a helper function `plot_function`, which takes in `x` and `y`, the range of `x` values we want to plot, and an optional title. I didn't think the details of this helper function were worth going into now, so I abstracted it away into the file `utils.py` in this same directory. It uses matplotlib like I described, but with a good bit of styling to make the plot more readable. If you really want to see the details perhaps the easiest thing to do is create a cell below using `ESC-B` and type the command `??plot_function`, which will print the code inside the function as the output.

Back to it, let's plot one example each of a constant function $y = 2$, a linear function $y = 2x$, and an affine function $2x - 1$.

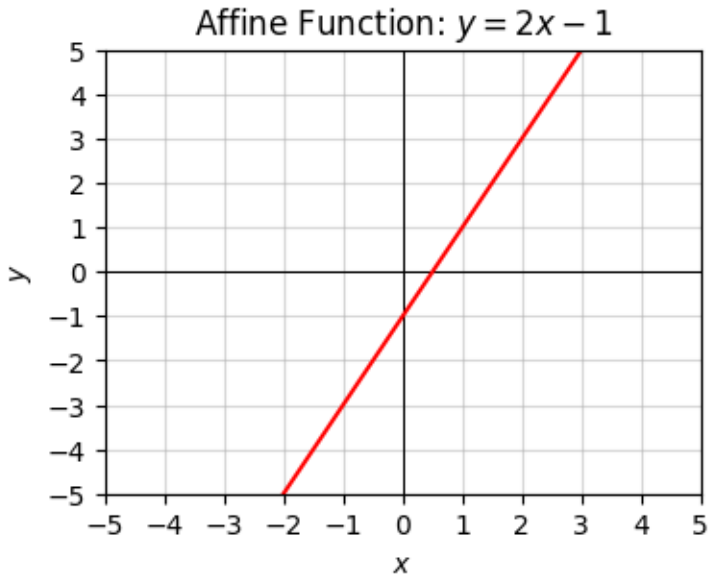
```
x = np.arange(-10, 10, 0.1)
f = lambda x: 2 * np.ones(len(x))
plot_function(x, f, xlim=(-5, 5), ylim=(-5, 5), ticks_every=[1, 1],
              title='Constant Function: $y=2$')
```




```
x = np.arange(-10, 10, 0.1)
f = lambda x: 2 * x
plot_function(x, f, xlim=(-5, 5), ylim=(-5, 5), ticks_every=[1, 1],
              title='Linear Function: $y=2x$')
```



```
x = np.arange(-10, 10, 0.1)
f = lambda x: 2 * x - 1
plot_function(x, f, xlim=(-5, 5), ylim=(-5, 5), ticks_every=[1, 1],
              title='Affine Function: $y=2x-1$')
```

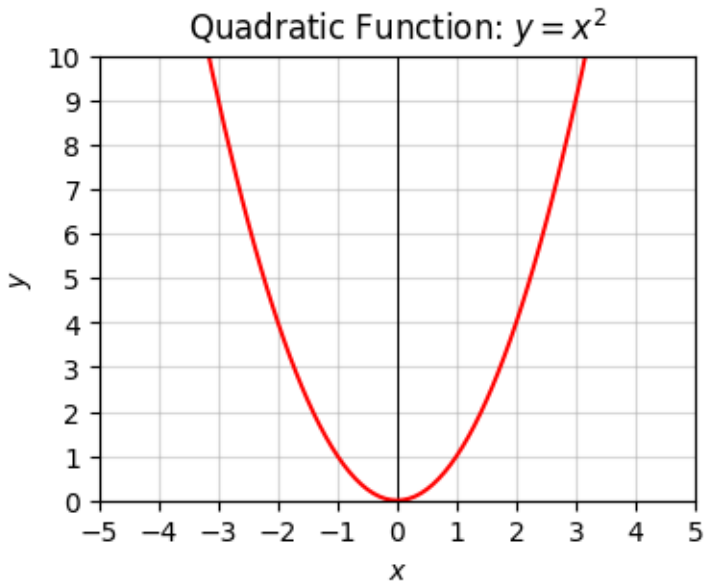


1.2.2 Polynomial Functions

Polynomial functions are just sums of positive integer powers of x, e.g. something like $y = 3x^2 + 5x + 1$ or $y = x^{10} - x^3 + 4$. The highest power that shows up in the function is called the **degree** of the polynomial. For example, the above examples have degrees 2 and 10 respectively. Polynomial functions tend to look like lines, bowls, or roller coasters that turn up and down some number of times.

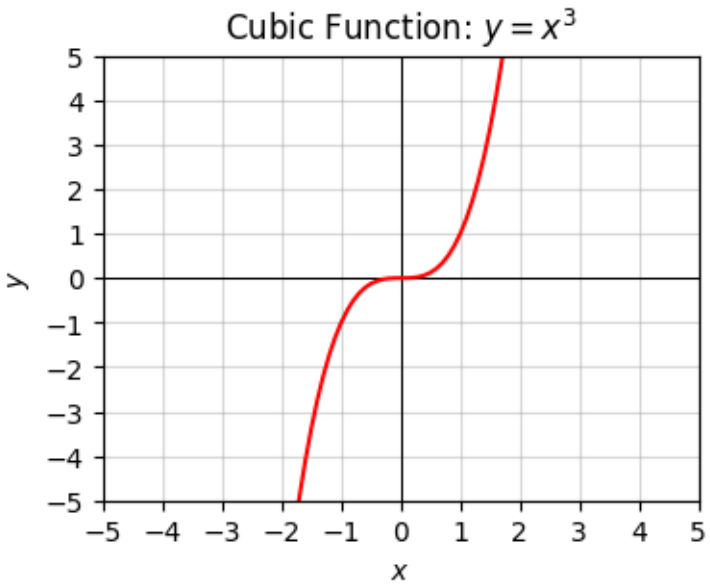
A major example is the quadratic function $y = x^2$, which is just an upward-shaped bowl. Its bowl-shaped curve is called a **parabola**. We can get a downward-shaped bowl by flipping the sign to $y = -x^2$.

```
x = np.arange(-10, 10, 0.1)
f = lambda x: x ** 2
plot_function(x, f, xlim=(-5, 5), ylim=(0, 10), ticks_every=[1, 1],
              title='Quadratic Function: $y=x^2$')
```



The next one up is the cubic function $y = x^3$. The cubic looks completely different from the bowl-shaped parabola.

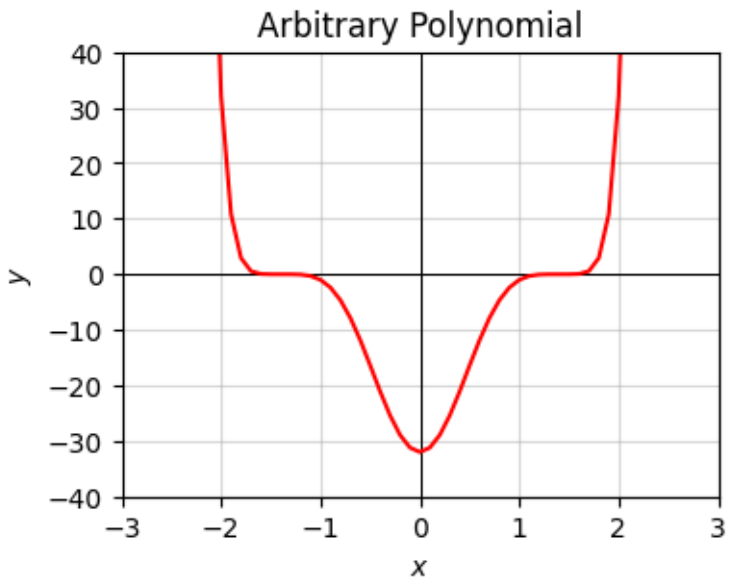
```
x = np.arange(-10, 10, 0.1)
f = lambda x: x ** 3
plot_function(x, f, xlim=(-5, 5), ylim=(-5, 5), ticks_every=[1, 1],
              title='Cubic Function: $y=x^3$')
```



Polynomials can take on much more interesting shapes than this. Here's a more interesting polynomial degree 10,

$$y = (x^2 - 1)^5 - 5(x^2 - 1)^4 + 10(x^2 - 1)^3 - 10(x^2 - 1)^2 + 5(x^2 - 1) - 1.$$

```
x = np.arange(-10, 10, 0.1)
def f(x):
    y = (x**2 - 1)**5 - 5 * (x**2 - 1)**4 + 10 * (x**2 - 1)**3 -
    10 * (x**2 - 1)**2 + 5 * (x**2 - 1) - 1
plot_function(x, f, xlim=(-3, 3), ylim=(-40, 40), ticks_every=[1, 10],
              title='Arbitrary Polynomial')
```



1.2.3 Rational Functions

Rational functions are functions that are ratios of polynomial functions. Examples might be $y = \frac{1}{x}$, or

$$y = \frac{x^3 + x + 1}{x^2 - 1}.$$

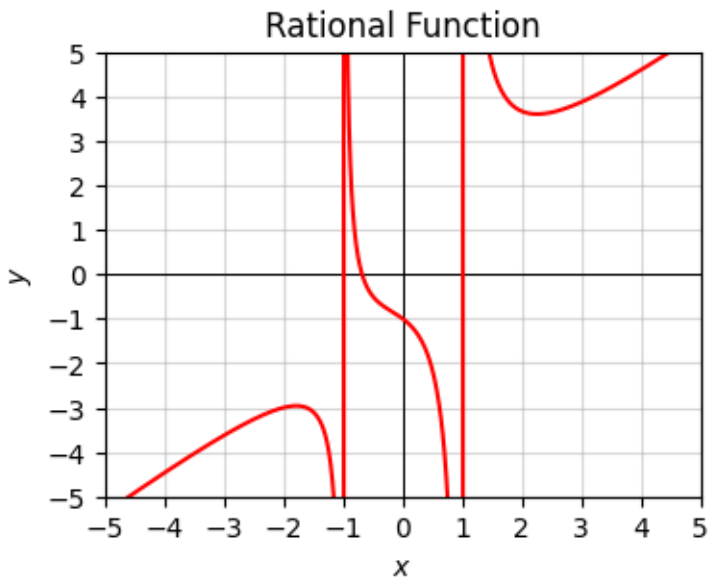
These functions typically look kind of like polynomial functions, but have points where the curve "blows up" to positive or negative infinity. The points where the function blows up are called **poles** or **asymptotes**.

Here's a plot of the function

$$y = \frac{x^3 + x + 1}{x^2 - 1}.$$

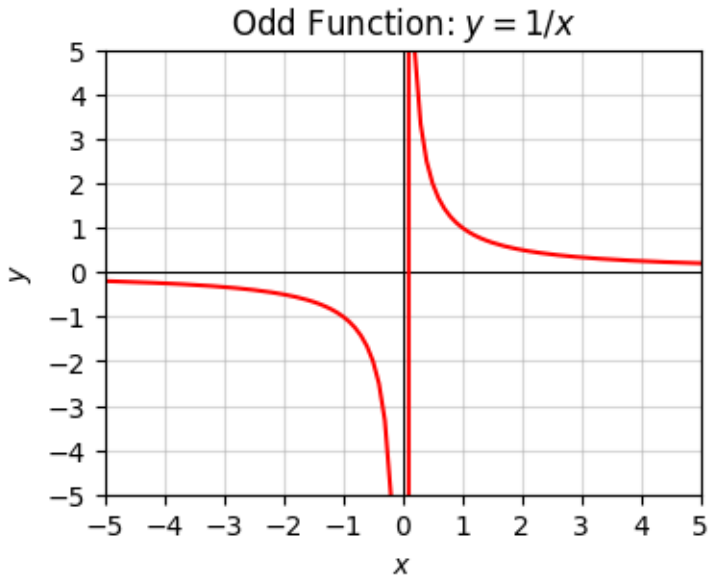
Notice how weird it looks. There are asymptotes (the vertical lines) where the function blows up at ± 1, which is where the denominator $x^2 - 1 = 0$.

```
x = np.arange(-10, 10, 0.01)
f = lambda x: (x ** 3 + x + 1) / (x ** 2 - 1)
plot_function(x, f, xlim=(-5, 5), ylim=(-5, 5), ticks_every=[1, 1],
              title='Rational Function')
```



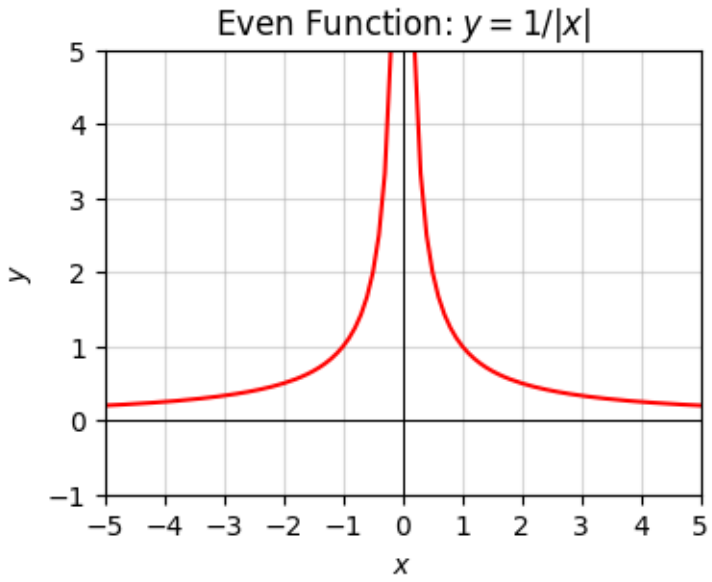
Here's a plot of $y = \frac{1}{x}$. There's an asymptote at $x = 0$. When $x > 0$ it starts at $+\infty$ and tapers down to 0 as x gets large. When $x < 0$ it does the same thing, except flipped across the origin $x = y = 0$. This is an example of an **odd function**, a function that looks like $f(x) = -f(x)$, which is clear in this case since $1/(-x) = -1/x$. Functions like the linear function $y = x$ and the cubic function $y = x^3$ are also odd functions.

```
x = np.arange(-10, 10, 0.1)
f = lambda x: 1 / x
plot_function(x, f, xlim=(-5, 5), ylim=(-5, 5), ticks_every=[1, 1],
              title='Odd Function: $y=1/x$')
```



A related function is $y = \frac{1}{|x|}$. The difference here is that $|x|$ can never be negative. This means $f(x) = f(-x)$. This is called an **even function**. Functions like this are symmetric across the y-axis. The quadratic function $y = x^2$ is also an even function.

```
x = np.arange(-10, 10, 0.1)
f = lambda x: 1 / np.abs(x)
plot_function(x, f, xlim=(-5, 5), ylim=(-1, 5), ticks_every=[1, 1],
              title='Even Function: $y=1/|x|$')
```



1.2.4 Power Functions

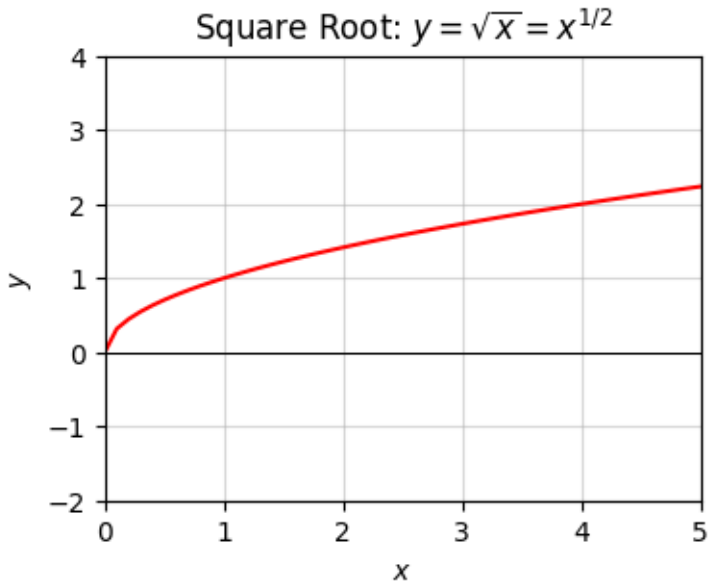
Functions that look like $y = \frac{1}{x^n}$ for some n are sometimes called inverse, hyperbolic. These can be represented more easily by using a negative power like $y = x^{-n}$, which means the exact same thing as $y = \frac{1}{x^n}$.

We can extend n to deal with things like square roots or cube roots or any kind of root as well by allowing n to be non-integer. For example, we can represent the square root function $y = \sqrt{x}$ as $y = x^{1/2}$, and the cube root $y = \sqrt[3]{x}$ as $y = x^{1/3}$. Roots like these are only defined when $x \geq 0$.

The general class of functions of the form $y = x^p$ for some arbitrary real number p are often called **power functions**.

Here's a plot of what the square root function looks like. Here y grows slower than a linear function, but still grows arbitrarily large with x.

```
x = np.arange(0, 10, 0.1)
f = lambda x: np.sqrt(x)
plot_function(x, f, xlim=(0, 5), ylim=(-2, 4), ticks_every=[1, 1],
              title='Square Root: $y=\sqrt{x}=x^{1/2}$')
```



Power functions obey the following rules:

Rule	**Example**
$x^0 = 1$	$2^0 = 1$
$x^{m+n} = x^m x^n$	$3^{2+5} = 3^2 3^5 = 3^8 = 6561$
$x^{m-n} = \frac{x^m}{x^n}$	$3^{2-5} = \frac{3^2}{3^5} = 3^{-3} \approx 0.037$
$x^{mn} = (x^m)^n$	$2^{2\cdot 5} = (2^2)^5 = 2^{10} = 1024$
$(xy)^n = x^n y^n$	$(2 \cdot 2)^3 = 2^3 2^3 = 4^3 = 2^6 = 64$
$\left(\frac{x}{y}\right)^n = \frac{x^n}{y^n}$	$\left(\frac{2}{4}\right)^3 = \frac{2^3}{4^3} = \frac{1}{8}$
$\left(\frac{x}{y}\right)^{-n} = \frac{y^n}{x^n}$	$\left(\frac{2}{4}\right)^{-3} = \frac{4^3}{2^3} = 2^3 = 8$
$x^{1/2} = \sqrt{x} = \sqrt[2]{x}$	$4^{1/2} = \sqrt{4} = 2$
$x^{1/n} = \sqrt[n]{x}$	$3^{1/4} = \sqrt[4]{3} \approx 1.316$
$x^{m/n} = \sqrt[n]{x^m}$	$3^{3/4} = \sqrt[4]{3^3} = \sqrt[4]{9} \approx 1.732$
$\sqrt[n]{xy} = \sqrt[n]{x}\sqrt[n]{y}$	$\sqrt[4]{3 \cdot 2} = \sqrt[4]{3}\sqrt[4]{2} \approx 1.565$
$\sqrt[n]{\frac{x}{y}} = \frac{\sqrt[n]{x}}{\sqrt[n]{y}}$	$\sqrt[4]{\frac{3}{2}} = \frac{\sqrt[4]{3}}{\sqrt[4]{2}} \approx 1.107$

It's important to remember that power functions *do not* distribute over addition, i.e.

$$(x + y)^n \neq x^n + y^n,$$

and by extension nor do roots,

$$\sqrt[n]{x+y} \neq \sqrt[n]{x} + \sqrt[n]{y}.$$

1.2.5 Exponentials and Logarithms

Two very important functions are the exponential function $y = \exp(x)$ and the logarithm function $y = \log(x)$. They show up surprisingly often in machine learning and the sciences, certainly more than most other special functions do.

The exponential function can be written as a power by defining a number e called Euler's number, given by $e = 2.71828\ldots$. Like π, e is an example of an irrational number, i.e. a number that can't be represented as a ratio of integers. Using e, we can write the exponential function in the more usual form $y = e^x$, where it's roughly speaking understood that we mean "multiply e by itself x times". For example, $\exp(2) = e^2 = e \cdot e$.

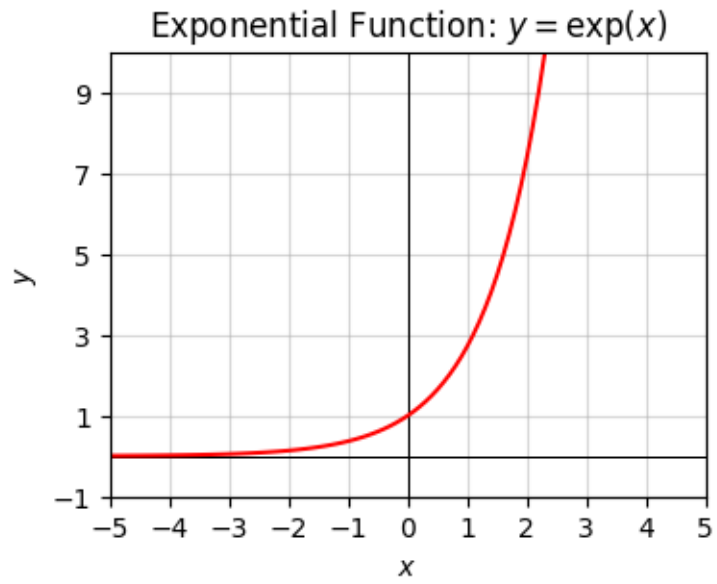
The logarithm is defined as the inverse of the exponential function. It's the unique function satisfying $\log(\exp(x)) = x$. The opposite is also true since the exponential must then be the inverse of the logarithm function, $\exp(\log(x)) = x$. This gives a way of mapping between the two functions,

$$\log(a) = b \quad \iff \quad \exp(b) = a.$$

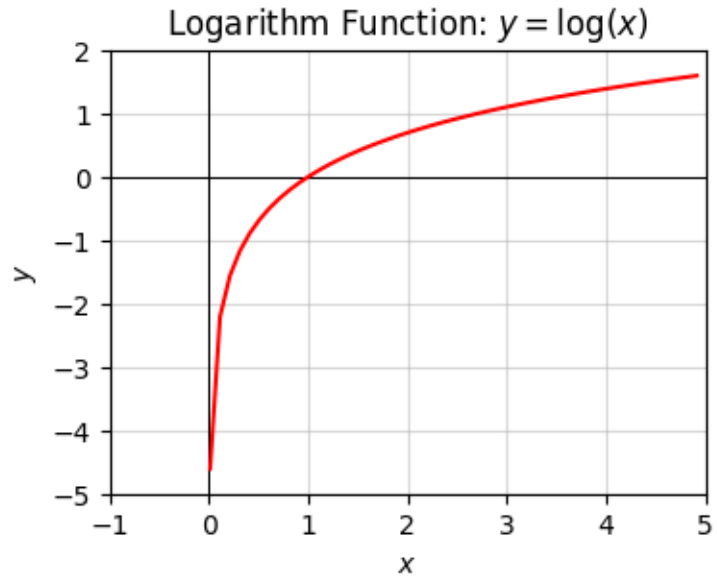
Here are some plots of what the exponential and logarithm functions look like. The exponential function is a function that blows up very, very quickly. The log function grows very, very slowly (much more slowly than the square root does).

Note the log function is only defined for positive-valued numbers $x \geq 0$, with $\log(+0) = -\infty$. This is dual to the exponential function only taking on $y \geq 0$.

```
x = np.arange(-5, 5, 0.1)
f = lambda x: np.exp(x)
plot_function(x, f, xlim=(-5, 5), ylim=(-1, 10), ticks_every=[1, 2],
              title='Exponential Function: $y=\exp(x)$')
```




```
x = np.arange(0.01, 5, 0.1)
f = lambda x: np.log(x)
plot_function(x, f, xlim=(-1, 5), ylim=(-5, 2), ticks_every=[1, 1],
              title='Logarithm Function: $y=\log(x)$')
```



The exponential and logarithm functions I defined are the "natural" way to define these functions. We can also have exponential functions in other bases, $y = a^x$ for any positive number a. Each a has an equivalent logarithm, written $y = \log_a(x)$. The two functions $y = a^x$ and $y = \log_a(x)$ are inverses of each other. When I leave off the a, it's assumed that all logs are the natural base $a = e$, sometimes also written $\ln(x)$.

Two common examples of other bases that show up sometimes are the base-2 functions 2^x and $\log_2(x)$, and the base-10 functions 10^x and $\log_{10}(x)$. Base-2 functions in particular show up often in computer science because of the tendency to think in bits. Base-10 functions show up when we want to think about how many digits a number has.

Here are some rules that exponentials and logs obey:

Rule	**Example**
$e^0 = 1$	
$\log(1) = 0$	
$\log(e) = 1$	
$e^{a+b} = e^a e^b$	$e^{2+5} = e^2 e^5 = e^8 \approx 2980.96$
$e^{a-b} = \frac{e^a}{e^b}$	$e^{2-5} = \frac{e^2}{e^5} = e^{-3} \approx 0.0498$
$e^{ab} = (e^a)^b$	$e^{2 \cdot 5} = (e^2)^5 = e^{10} \approx 22026.47$
$a^b = e^{b \log(a)}$	$2^3 = e^{3 \log(2)} = 8$
$\log(ab) = \log(a) + \log(b)$	$\log(2 \cdot 5) = \log(2) + \log(5) = \log(10) \approx 2.303$
$\log\left(\frac{a}{b}\right) = \log(a) - \log(b)$	$\log\left(\frac{2}{5}\right) = \log(2) - \log(5) \approx -0.916$
$\log(a^b) = b \log(a)$	$\log(5^2) = 2 \log(5) \approx 3.219$
$\log_a(x) = \frac{\log(x)}{\log(a)}$	$\log_2(5) = \frac{\log(5)}{\log(2)} \approx 2.322$

Here's an example of an equation involving exponentials and logs. Suppose you have n bits of numbers (perhaps it's the precision in some float) and you want to know how many *digits* this number takes up in decimal form (what you're used to). This would be equivalent to solving the following equation for x,

$$\begin{aligned} 2^n &= 10^x \\ \log(2^n) &= \log(10^x) \\ n \log(2) &= x \log(10) \\ x &= \frac{\log(2)}{\log(10)} \cdot n \\ x &\approx 0.3 \cdot n. \end{aligned}$$

For example, you can use this formula to show that 52 bits of floating point precision translates to about 15 to 16 digits of precision. In numpy, the function `np.log` function calculates the (base-e) log of a number.

```
n = 52
x = np.log(2) / np.log(10) * n
print(f'x = {x}')
```

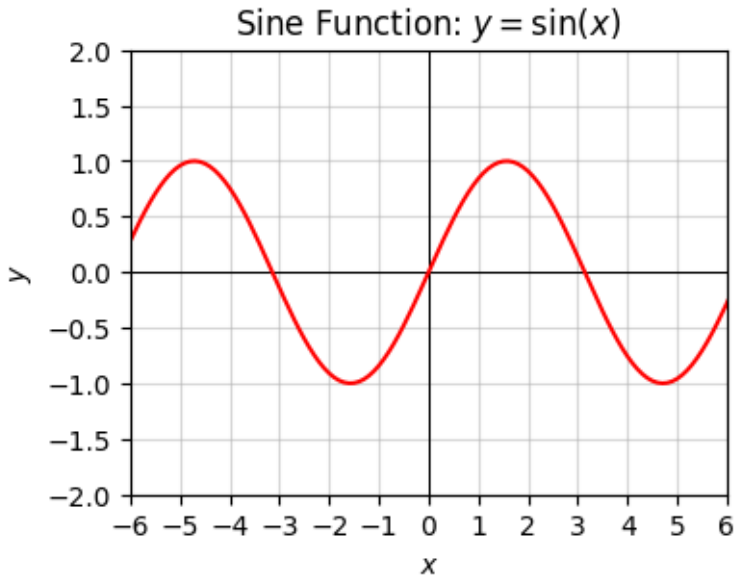
```
x = 15.65355977452702
```

1.2.6 Trigonometric Functions

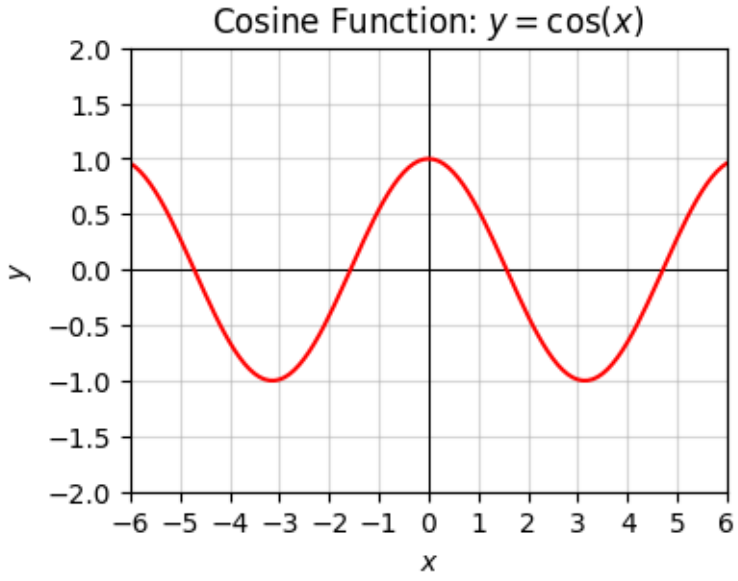
Other textbook functions typically covered in math courses are the trig functions: sine, cosine, tangent, cosine, cosecant, and cotangent. Of these functions, the most important to know are the sine function $y = \sin x$, the cosine function $y = \cos x$, and *sometimes* the tangent function $y = \tan x$.

Here's what their plots look like. They're both waves that repeat themselves, in the sense $f(x + 2\pi) = f(x)$. The length for the function to repeat itself is called the *period*, in this case $2\pi \approx 6.28$. Note that the cosine is just a sine function that's shifted right by $\frac{\pi}{2} \approx 1.57$.

```
x = np.arange(-10, 10, 0.1)
f = lambda x: np.sin(x)
plot_function(x, f, xlim=(-6, 6), ylim=(-2, 2),  ticks_every=[1, 0.5],
              title='Sine Function: $y=\sin(x)$')
```




```
x = np.arange(-10, 10, 0.1)
f = lambda x: np.cos(x)
plot_function(x, f, xlim=(-6, 6), ylim=(-2, 2), ticks_every=[1, 0.5],
              title='Cosine Function: $y=\cos(x)$')
```



Trig functions don't really show up that much in machine learning, so I won't remind you of all those obscure trig rules you've forgotten. I'll just mention that we can define all the other trig functions using the sine and cosine as follows,

$$\begin{aligned}\tan x &= \frac{\sin x}{\cos x},\\ \csc x &= \frac{1}{\sin x},\\ \sec x &= \frac{1}{\cos x},\\ \cot x &= \frac{1}{\tan x} = \frac{\cos x}{\sin x}.\end{aligned}$$

We can talk about the *inverse* of trig functions as well. These are just the functions that undo the trig operations and give you back the angle (in radians). Since none of the trig functions are monotonic, we can't invert them on the whole real line, but only on a given range.

Below I'll just list the inverse sine, cosine, and tangent functions and their defined input and output ranges. Note by historical convention, these inverse functions are usually called the **arcsine**, **arccosine**, and **arctangent** respectfully.

Inverse Function	**Input Range**	**Output Range**
$y = \arcsin x = \sin^{-1} x$	$-1 \leq x \leq 1$	$-90^\circ \leq y \leq 90^\circ$
$y = \arccos x = \cos^{-1} x$	$-1 \leq x \leq 1$	$0^\circ \leq y \leq 180^\circ$
$y = \arctan x = \tan^{-1} x$	$-\infty < x < \infty$	$-90^\circ \leq y \leq 90^\circ$

1.2.7 Piecewise Functions

The functions covered so far are examples of **continuous functions**. Their graphs don't have jumps or holes in them anywhere. Continuous functions we can often write using a single equation, like $y = x^2$ or $y = 1 + \sin(x)$. We can also have functions that require more than one equation to write. These are called **piecewise functions**. Piecewise functions usually aren't continuous, but sometimes can be.

An example of a discontinuous piecewise function is the unit step function $y = u(x)$ given by

$$y = \begin{cases} 0 & x < 0, \\ 1 & x \geq 0. \end{cases}$$

This expression means $y = 0$ whenever $x < 0$, but $y = 1$ whenever $x \geq 0$. It breaks up into two pieces, one horizontal line $y = 0$ when x is negative, and another horizontal line $y = 1$ when x is positive.

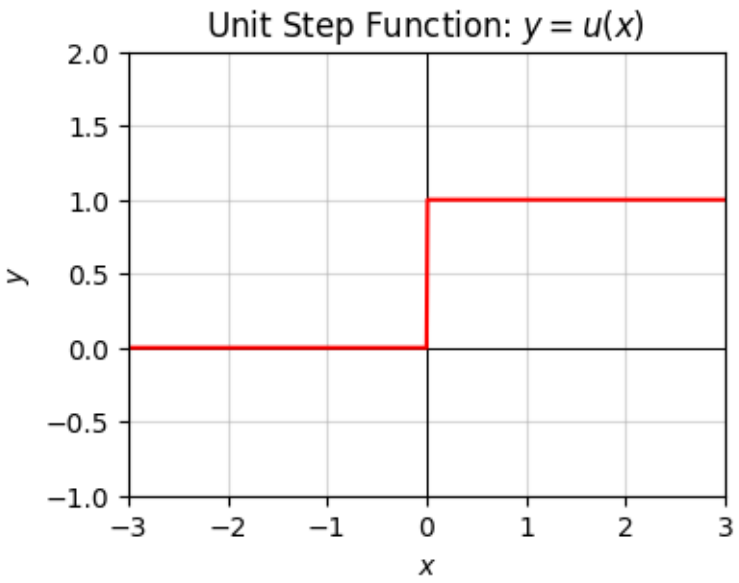
Using Boolean expressions, we can also write this function in a more economical way by agreeing to identify $x = 1$ with TRUE and $x = 0$ with FALSE, which python does by default. In this notation, we can write

$$u(x) = [x \geq 0],$$

which means exactly the same thing as the piecewise definition, since $x \geq 0$ is only true when (you guessed it), $x \geq 0$.

Here's a plot of this function. Note the discontinuous jump at $x = 0$.

```
x = np.arange(-10, 10, 0.01)
f = lambda x:  (x >= 0)
plot_function(x, f, xlim=(-3, 3), ylim=(-1, 2), ticks_every=[1, 0.5],
              title='Unit Step Function: $y=u(x)$')
```



An example of a piecewise function that's continuous is the **ramp function**, defined by

$$y = \begin{cases} 0 & x < 0, \\ x & x \geq 0. \end{cases}$$

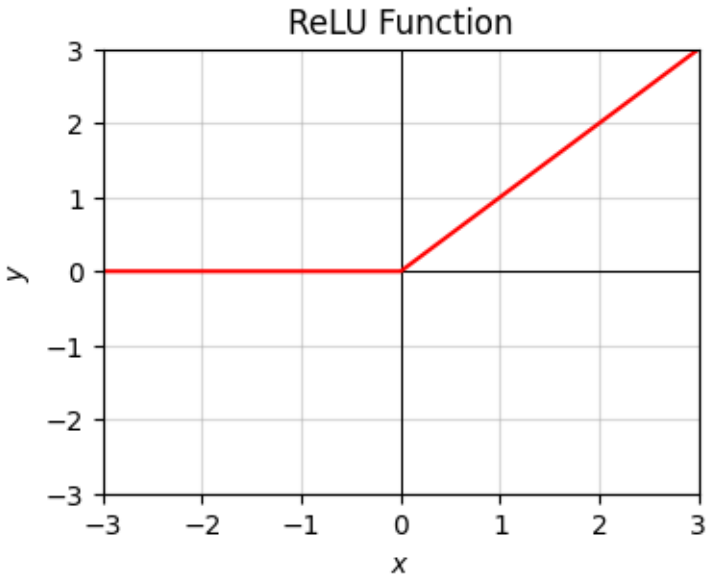
This function gives a horizontal line $y = 0$ when x is negative, and a 45° line $y = x$ when x is positive. Both lines connect at $x = 0$, but leave a kink in the graph.

Another way to write the same thing using Boolean expressions is $y = x \cdot [x \geq 0]$, which is of course just $y = x \cdot u(x)$.

In machine learning it's more common to write the ramp function using the max function as $y = \max(0, x)$. This means, for each x, take that value and compare it with 0, and take the maximum of those two. That is, if x is negative take $y = 0$, otherwise take $y = x$. It's also more common to call this function a **rectified linear unit**, or **ReLU** for short. It's an ugly, unintuitive name, but unfortunately it's stuck in the field.

Here's a plot of the ramp or ReLU function. Notice how it stays at $y = 0$ for a while, then suddenly "ramps upward" at $x = 0$.

```
x = np.arange(-10, 10, 0.1)
f = lambda x:  x * (x >= 0)
plot_function(x, f, xlim=(-3, 3), ylim=(-3, 3), ticks_every=[1, 1],
              title='ReLU Function')
```



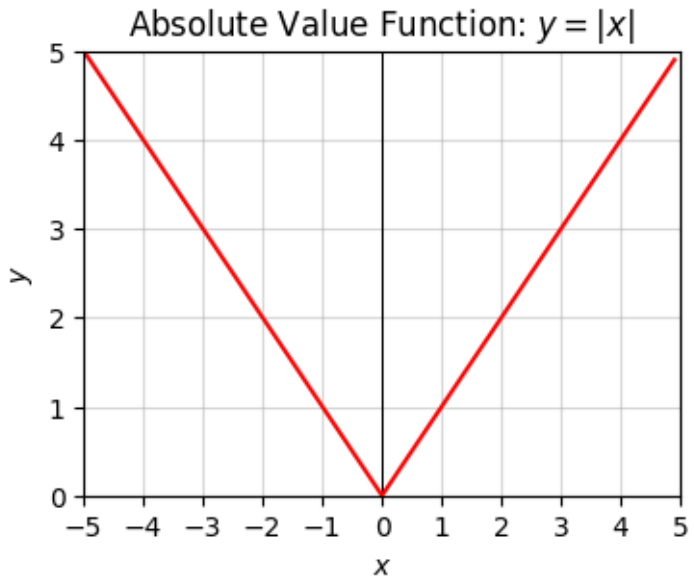
Last, I'll mention here the **absolute value** function $y = |x|$, defined by the piecewise function

$$y = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0. \end{cases}$$

The absolute value just ignores negative signs and makes everything positive. The function looks like the usual line $y = x$ when positive, but like the negative-sloped line $y = -x$ when

negative. At $x = 0$ the two lines meet, creating a distinctive v-shape. To get the absolute value function in python, use `abs` or `np.abs`.

```
x = np.arange(-5, 5, 0.1)
f = lambda x: abs(x)
plot_function(x, f, xlim=(-5, 5), ylim=(0, 5), ticks_every=[1, 1],
              title='Absolute Value Function: $y=|x|$')
```



1.2.8 Composite Functions

We can also have any arbitrary hybrid of the above functions. We can apply exponentials to affine functions, logs to sine functions, sines to exponential functions. In essence, this kind of layered composition of functions is what a neural network is as we'll see later on.

Math folks often write an abstract compositional function as a function applied to another function, like $y = f(g(x))$ or $y = (f \circ g)(x)$. These can be chained arbitrarily many times, not just two. Neural networks do just that, often hundreds or thousands of times.

Consider, for example, the function composition done by applying the following functions in sequence:

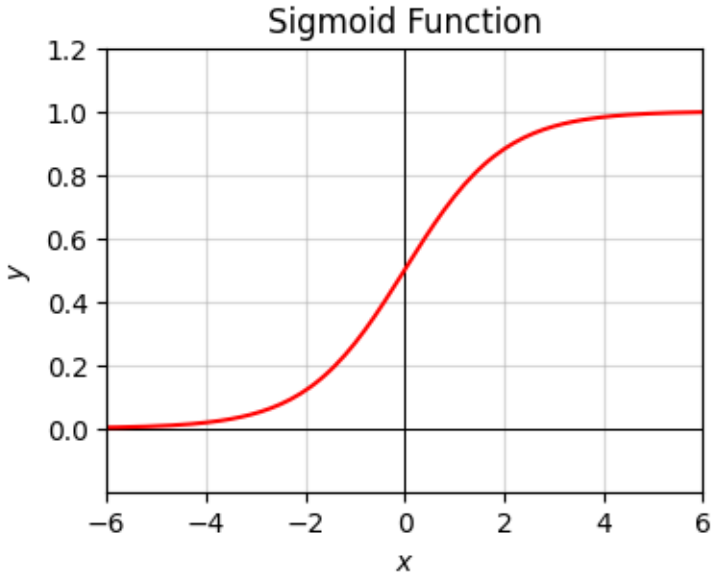
- an affine function $f(x) = wx + b$
- followed by a linear function $g(x) = -x$
- followed by an exponential function $h(x) = e^x$
- followed by a rational function $r(x) = \frac{1}{x}$

to get the full function

$$y = r(h(g(x))) = \frac{1}{1 + e^{-(wx+b)}}.$$

Here's a plot of what this function looks like for the "standard form" where $w = 1, b = 0$. Notice that $0 \leq y \leq 1$. The values of x get "squashed" to values between 0 and 1 after the function is applied.

```
x = np.arange(-10, 10, 0.1)
f = lambda x:  1 / (1 + np.exp(-x))
plot_function(x, f, xlim=(-6, 6), ylim=(-0.2, 1.2), ticks_every=[2, 0.2],
              title='Sigmoid Function')
```



This function is called the **sigmoid** function. The sigmoid is very important in machine learning since it in essence creates probabilities. We'll see it a lot more. The standard form sigmoid function, usually written $\sigma(x)$, is given by

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Arbitrary affine transformations of the standard form would then be written as $\sigma(wx + b)$.

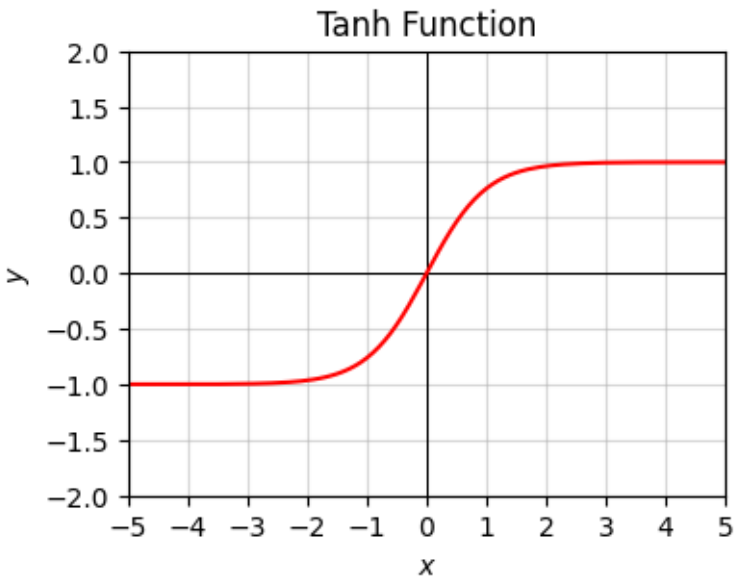
A similar looking function shows up sometimes as well called the **hyperbolic tangent** or **tanh** function, which has the (standard) form

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

The tanh function looks pretty much the same as the sigmoid except it's rescaled vertically so that $-1 \leq y \leq 1$.

Here's a plot of the tanh function. Notice how similar it looks to the sigmoid with the exception of the scale of the y-axis.

```
x = np.arange(-10, 10, 0.1)
f = lambda x: (np.exp(x) - np.exp(-x)) / (np.exp(x) + np.exp(-x))
plot_function(x, f, xlim=(-5, 5), ylim=(-2, 2), ticks_every=[1, 0.5],
              title='Tanh Function')
```



1.2.9 Function Transformations

Suppose we have some arbitrary function $f(x)$ and we apply a series of compositions to get a new function

$$g(x) = a \cdot f(b \cdot (x + c)) + d.$$

We can regard each parameter a, b, c, d as doing some kind of geometric transformation to the graph of the original function $f(x)$. Namely,

- a re-scales the function vertically (if a is negative it also flips $f(x)$ upside down)
- b re-scales the function horizontally (if b is negative it also flips $f(x)$ left to right)

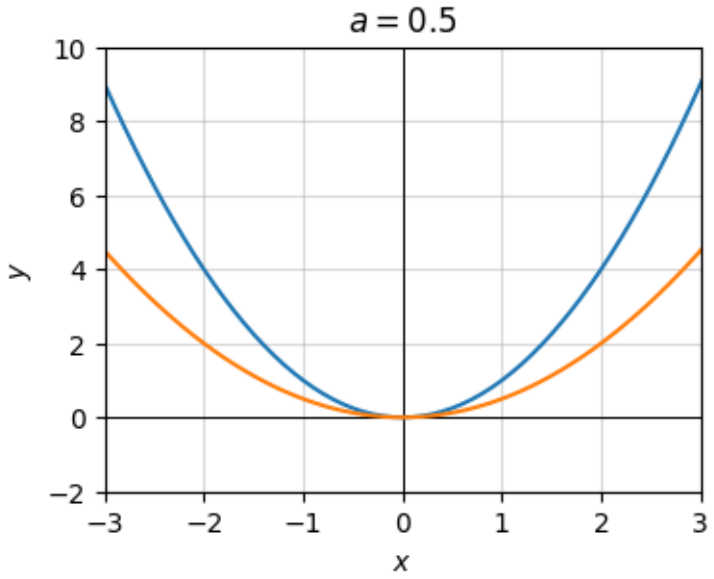
- c shifts the function horizontally (left if c is positive, right if c is negative)
- d shifts the function vertically (up if d is positive, down if d is negative)

Here's an example of how these work. Consider the function $f(x) = x^2$. We're going to apply each of these transformations one by one to show what they do to the graph of $f(x)$.

First, let's look at the transformation $g(x) = \frac{1}{2}f(x) = \frac{1}{2}x^2$. Here $a = \frac{1}{2}$ and the rest are zero. I'll plot it along side the original graph (the blue curve). Notice the graph gets flattened vertically by a factor of two (the orange curve).

```
x = np.arange(-10, 10, 0.1)
f = lambda x: x ** 2
```

```
a = 1 / 2
g = lambda x: a * x ** 2
plot_function(x, [f, g], xlim=(-3, 3), ylim=(-2, 10), ticks_every=[1, 2],
              title=f'$a={a}$')
```

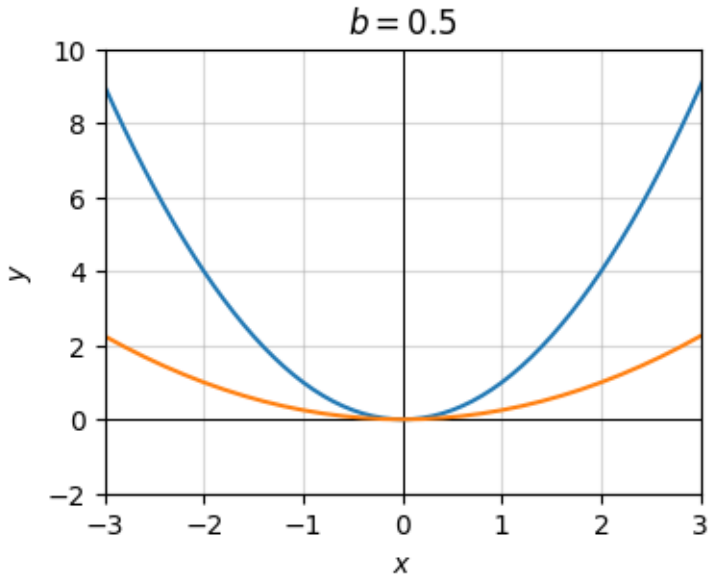


Now consider at the transformation

$$g(x) = f(\frac{1}{2}x) = \left(\frac{1}{2}x\right)^2.$$

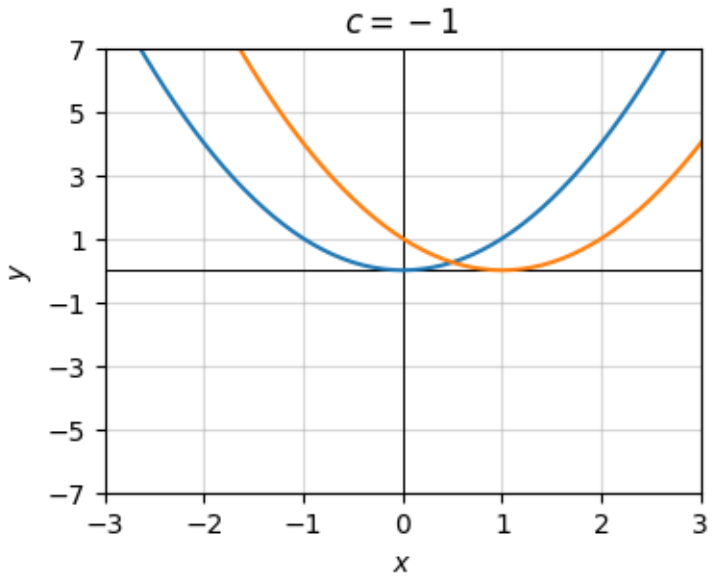
Here $b = \frac{1}{2}$ and the rest are zero. Notice the graph again gets flattened but in a slightly different way.

```
b = 1 / 2
g = lambda x: (b * x) ** 2
plot_function(x, [f, g], xlim=(-3, 3), ylim=(-2, 10), ticks_every=[1, 2],
              title=f'$b={b}$')
```



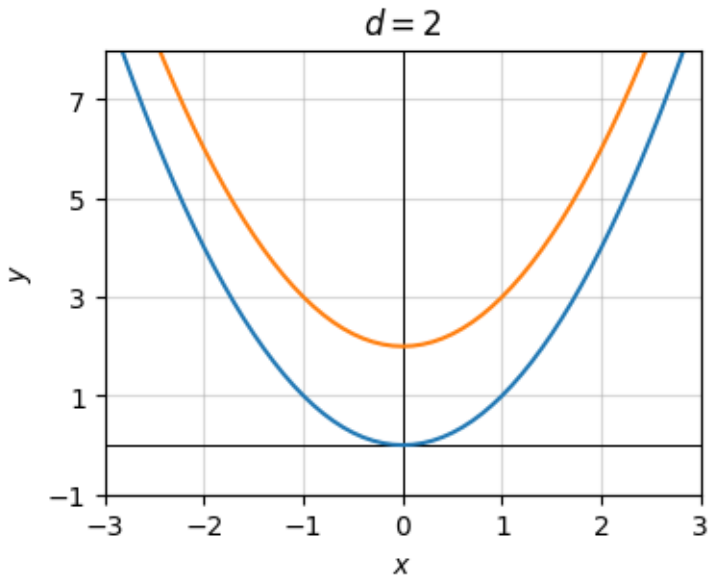
Next, consider the transformation $g(x) = f(x - 1) = (x - 1)^2$. Here $c = 1$ and the rest are zero. Notice the graph's shape doesn't change. It just gets shifted *right* by $c = 1$ since c is negative.

```
c = -1
g = lambda x: (x + c) ** 2
plot_function(x, [f, g], xlim=(-3, 3), ylim=(-7, 7), ticks_every=[1, 2],
              title=f'$c={c}$')
```



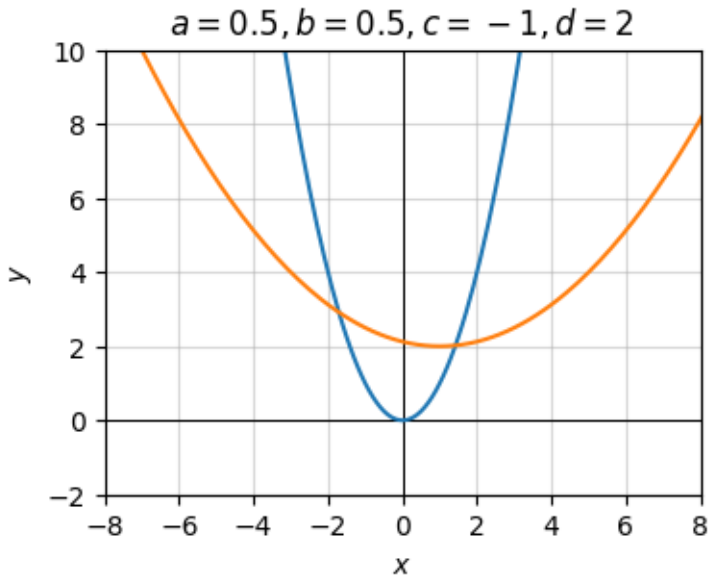
Finally, let's look at the transformation $g(x) = f(x) + 2 = x^2 + 2$. Here $d = 2$ and the rest are zero. Notice again the graph's shape doesn't change. It just gets shifted *up* by $d = 2$ units.

```
d = 2
g = lambda x: x ** 2 + d
plot_function(x, [f, g], xlim=(-3, 3), ylim=(-1, 8), ticks_every=[1, 2],
              title=f'$d={d}$')
```



Let's now put them all together to see what happens. We should see rescaling in both directions and shifts in both directions. It's hard to see in the plot, but it's all there if you zoom in. The vertex of the parabola is at the point $x = c = 1, y = d = 2$. And the stretching factors due to $a = b = 1/2$ are both acting to flatten the parabola.

```
g = lambda x: a * (b * (x + c)) ** 2 + d
plot_function(x, [f, g], xlim=(-8, 8), ylim=(-2, 10), ticks_every=[2, 2],
              title=f'$a={a}, b={b}, c={c}, d={d}$')
```



1.3 Multivariate Functions

What we've covered thus far only deals with univariate functions, functions where $y = f(x)$, but x and y are just single numbers, i.e. scalars. In machine learning we're almost always dealing with multivariate functions with *lots* of variables, sometimes billions of them. It turns out that most of what I've covered so far extends straight forwardly to multivariate functions with some small caveats, which I'll cover below.

Simply put, a multivariate function is a function of multiple variables. Instead of a single variable x, we might have several variables, e.g. $x_0, x_1, x_2, x_3, x_4, x_5$,

$$y = f(x_0, x_1, x_2, x_3, x_4, x_5).$$

If you think about mathematical functions analogously to python functions it shouldn't be surprising functions can have multiple arguments. They usually do, in fact.

Here's an example of a function that takes two arguments x and y and produces a single output z, more often written as a *bivariate function* $z = f(x, y)$. The example I'll look at is $z = x^2 + y^2$. I'll evaluate the function at three points:

- $x = 0$, $y = 0$,
- $x = 1$, $y = -1$,
- $x = 0$, $y = 1$.

The main thing to notice is the function does exactly what you think it does. If you plug in 2 values, you get out 1 value.

```
f = lambda x, y: x ** 2 + y ** 2
print(f'z = f{(0, 0)} = {f(0, 0)}')
print(f'z = f{(1, -1)} = {f(1, -1)}')
print(f'z = f{(0, 1)} = {f(0, 1)}')
```

```
z = f(0, 0) = 0
z = f(1, -1) = 2
z = f(0, 1) = 1
```

We can also have functions that map multiple inputs to multiple outputs. Suppose we have a function that takes in 2 values x_0, x_1 and outputs 2 values y_0, y_1. We'd write this as $(y_0, y_1) = f(x_0, x_1)$.

Consider the following example,

$$(y_0, y_1) = f(x_0, x_1) = (x_0 + x_1, x_0 - x_1).$$

This is really just two functions, both functions of x_0 and x_1. We can completely equivalently write this function as

$$y_0 = f_1(x_0, x_1) = x_0 + x_1,$$
$$y_1 = f_2(x_0, x_1) = x_0 - x_1.$$

Here's this function defined and evaluated at the point $x_0 = 1$, $x_1 = 1$.

```
f = lambda x0, x1: (x0 + x1, x0 - x1)
print(f'(y0, y1) = {f(1, 1)}')
```

```
(y0, y1) = (2, 0)
```

For now I'll just focus on the case of multiple inputs, single output like the first example. These are usually called **scalar-valued functions**. We can also have **vector-valued functions**, which are functions whose *outputs* can have multiple values as well. I'll focus on scalar-valued functions here.

A scalar-valued function of n variables $x_0, x_1, \cdots, x_{n-1}$ has the form

$$y = f(x_0, x_1, \cdots, x_{n-1}).$$

Note n can be as large as we want it to be. When working with deep neural networks (which are just multivariate functions of a certain form) n can be huge. For example, if the input is a 256×256 image, the input might be $256^2 = 65536$ pixels. For a 10 second audio clip that's sampled at 44 kHz, the input might be $10 * 44k = 440k$ amplitudes. Large numbers indeed.

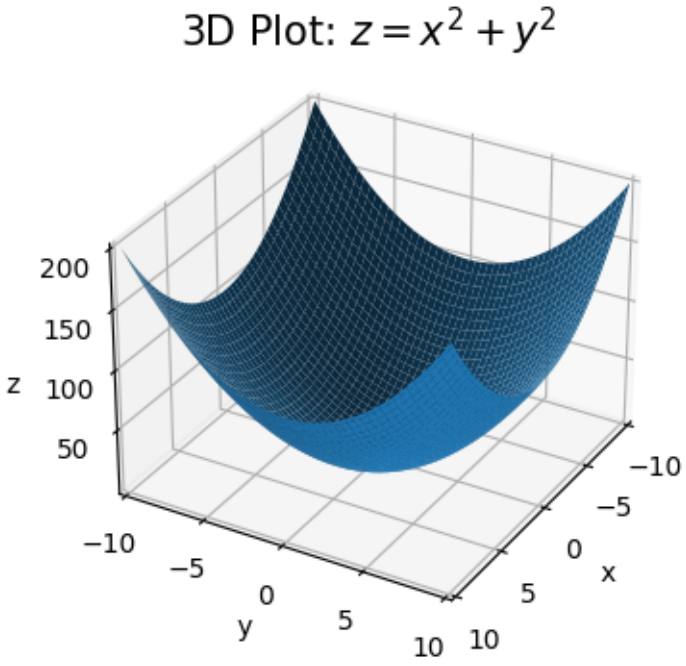
Calculating the output of multivariate functions is just as straight-forward as for univariate functions pretty much. Unfortunately, visualizing them is much harder. The human eye can't see 65536 dimensions, only 3 dimensions. This in some sense means we need to give up on the ability to "graph" a function and instead find other ways to visualize it.

One thing that sometimes help to visualize high dimension functions is to pretend they're functions of two variables, like $z = f(x, y)$. In this special case we can visualize the inputs as an xy-plane, and the output as a third axis sticking out perpendicular to the xy-plane from the origin. Each x, y pair will map to one unique z value. Done this way, we won't get a graph of a *curve* as before, but a *surface*.

Here's an example of what this might look like for the simple function $z = x^2 + y^2$. I'll plot the function on the domain $-10 \leq x \leq 10$ and $-10 \leq y \leq 10$ using the helper function `plot_3d`. It takes in two lists of values `x` and `y`. I'll use `np.linspace` to sample 100 points from -10 to 10 for each. Then I'll define a lambda function that maps `x` and `y` to the output `z`. Passing these three arguments into the helper function gives us our 3D plot.

```
x = np.linspace(-10, 10, 100)
y = np.linspace(-10, 10, 100)
f = lambda x, y: x**2 + y**2
```

```
plot_function_3d(x, y, f, title='3D Plot: $z=x^2+y^2$',
                 ticks_every=[5, 5, 50], labelpad=5, dist=12)
```



Notice how the plot looks like an upward facing bowl. Imagine a bowl lying on a table. The table is the xy-plane. The bowl is the surface $z = x^2 + y^2$ we're plotting. While the plot shows the general idea what's going on, 3D plots can often be difficult to look at. They're often slanted at funny angles and hide important details.

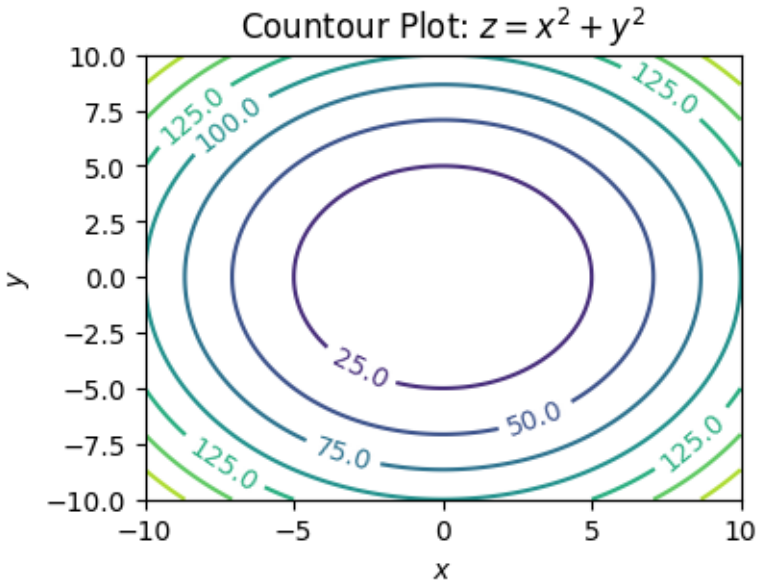
Here's another way we can visualize the same function: Rather than create a third axis for z, we can plot it directly on the xy-plane as a 2D plot. Since we're dealing with a surface, not a curve, we have to do this for lots of different z values, which will give a *family* of curves. For example, we might plot all of the following curves corresponding to different values of z in the xy-plane,

$$25 = x^2 + y^2, \tag{1.1}$$
$$50 = x^2 + y^2, \tag{1.2}$$
$$75 = x^2 + y^2, \tag{1.3}$$
$$100 = x^2 + y^2, \tag{1.4}$$
$$125 = x^2 + y^2, \tag{1.5}$$
$$150 = x^2 + y^2. \tag{1.6}$$

Doing this will give a family of curves on one 2D plot, with each curve representing some value of z. In our example, these curves are all circles of radius z^2. Each curve is called a **level curve** or **level set**.

These kinds of plots are called **contour plots**. A contour map can be thought of as looking at the surface from the top down, where each level set corresponds to slicing the function $z = f(x, y)$ horizontally for different values of z. This trick is often used in topographical maps to visualize 3D terrain on a 2D sheet of paper. Here is a contour plot for $z = x^2 + y^2$ using the above level curves.

```
plot_countour(x, y, f, title='Countour Plot: $z=x^2+y^2$')
```



Notice how we get a bunch of concentric rings in the contour plot, each labeled by some value (their z values). These rings correspond to the circles I was talking about. You can visually imagine this plot as looking down from the top of the bowl. In the middle you see the bottom. The rings get closer together the further out you go, which indicates that the bowl is sloping steeper the further out we get.

We'll see more examples of multivariate functions in the coming lessons.

1.4 Systems of Equations

In machine learning we'll find ourselves frequently interested not just with single equations, but multiple equations each with many variables. One thing we might seek to do is solve these coupled systems, which means finding a solution that satisfies every equation simultaneously. Consider the following example,

$$\begin{aligned} x + \ \ y &= \ 2 \\ 2x - 3y &= 7. \end{aligned}$$

This system consists of two equations, $x + y = 2$, and $2x - 3y = 7$. Each equation contains two unknown variables, x and y. We need to find a solution for both x and y that satisfies both of these equations.

Usually the easiest and most general way to solve simple coupled systems like this is the **method of substitution**. The idea is to solve one equation for one variable in terms of the other, then plug that solution into the second equation to solve for the other variable. Once the second variable is solved for, we can go back and solve for the first variable explicitly. Let's start by solving the first equation for x in terms of y. This is pretty easy,

$$x = 2 - y.$$

Now we can take this solution for x and plug it into the second equation to solve for y,

$$\begin{aligned} 2x - 3y &= 7 \\ 2(2 - y) - 3y &= 7 \\ 4 - 5y &= 7 \\ 5y &= -3 \\ y &= -\frac{3}{5}. \end{aligned}$$

With y in hand, we can now solve for x, $x = 2 - y = 2 + \frac{3}{5} = \frac{13}{5}$. Thus, the pair $x = \frac{13}{5}$, $y = -\frac{3}{5}$ is the solution that solves both of these coupled equations simultaneously.

Here's sympy's solution to the same system. It should of course agree with what I just got, which it does.

```
x, y = sp.symbols('x y')
eq1 = sp.Eq(x + y, 2)
```

```
eq2 = sp.Eq(2 * x - 3 * y, 7)
sol = sp.solve((eq1, eq2), (x, y))
print(f'x = {sol[x]}')
print(f'y = {sol[y]}')
```

```
x = 13/5
y = -3/5
```

Notice that both of the equations in this example are *linear*, since each term only contains terms proportional to x and y. There are no terms like x^2 or $\sin y$ or whatever. Linear systems of equations are special because they can always be solved as long as there are enough variables. I'll spend a lot more time on these when I get to linear algebra.

We can also imagine one or more equations being *nonlinear*. Provided we can solve each equation one-by-one, we can apply the method of substitution to solve these too. Here's an example. Consider the nonlinear system

$$\begin{aligned} e^{x+y} &= 10 \\ xy &= 1. \end{aligned}$$

Let's solve the second equation first since it's easier. Solving for y gives $y = \frac{1}{x}$. Now plug this into the first equation and solve for x,

$$\begin{aligned} e^{x+y} &= 10 \\ e^{x+1/x} &= 10 \\ \log\left(e^{x+1/x}\right) &= \log 10 \\ x + \frac{1}{x} &= \log 10 \\ x^2 - \log 10 \cdot x + 1 &= 0 \\ x &= \frac{1}{2}\left(\log 10 \pm \sqrt{(\log 10)^2 - 4}\right) \\ x &\approx 0.581,\ 1.722. \end{aligned}$$

Note here I had to use the **quadratic formula**, which I'll assume you've forgotten. If you have a quadratic equation of the form $ax^2 + bx + c = 0$, then it will (usually) have exactly two solutions given by the formula

$$x = \frac{1}{2a}\Big(-b \pm \sqrt{b^2 - 4ac}\Big).$$

This means we have two different possible solutions for x, which thus means we'll also have two possible solutions to y since $y = \frac{1}{x}$. Thus, this system has *two* possible solutions,

$$\text{Solution 1: } x \approx 0.581,\ y \approx 1.722,$$

$$\text{Solution 2: } x \approx 1.722,\ y \approx 0.581.$$

It's interesting how symmetric these two solutions are. They're basically the same with x and y swapped. This is because the system has symmetry. You can swap x and y in the system above and not change the equation, which means the solutions must be the same up to permutation of x and y!

Here's sympy's attempt to solve this system.

```
x, y = sp.symbols('x y')
eq1 = sp.Eq(sp.exp(x + y), 10)
eq2 = sp.Eq(x * y, 1)
sol = sp.solve((eq1, eq2), (x, y))
print(f'x1 = {sol[0][0].round(5)} \t y1 = {sol[0][1].round(5)}')
print(f'x2 = {sol[1][0].round(5)} \t y2 = {sol[1][1].round(5)}')
```

```
x1 = 0.58079     y1 = 1.72180
x2 = 1.72180     y2 = 0.58079
```

In general, it's not even possible to solve a system of nonlinear equations except using numerical methods. The example I gave was rigged so I could solve it by hand. General purpose **root-finding** algorithms exist that can solve arbitrary systems of equations like this numerically, no matter how nonlinear they are.

To solve a nonlinear system like this numerically, you can use the scipy function `scipy.optimize.fsolve`. Scipy is an extension of numpy that includes a lot of algorithms for working with non-linear functions. To use `fsolve`, you have to define the system as a function mapping a list of variables to a list of equations. You also have to specify a starting point `x0` for the root finder. This tells it where to start looking for the root. Since nonlinear equations have multiple solutions, picking a different `x0` can and will often give you a different root. I won't dwell on all this since we don't really need to deal with root finding much in machine learning.

```
from scipy.optimize import fsolve

system = lambda xy: [np.exp(xy[0] + xy[1]) - 10, xy[0] * xy[1] - 1]
solution = fsolve(system, x0=(1, 1))
print(f'solution = {solution}')
```

```
solution = [0.5807888 1.7217963]
```

1.5 Sums and Products

1.5.1 Sums

We typically find ourselves performing operations on large numbers of numbers at a time. By far the most common operation is adding up a bunch of numbers, or **summation**. Suppose we have some **sequence** of n numbers $x_0, x_1, x_2, \cdots, x_{n-1}$. They could be anything, related by a function, or not. If we wanted to sum them together to get a new number x we could write

$$x = x_0 + x_1 + x_2 + \cdots + x_{n-1}.$$

But it's kind of cumbersome to always write like this. For this reason in math there's a more compact notation to write sums called **summation notation**. We introduce the symbol $\sum$ for "sum", and write

$$x = \sum_{i=0}^{n-1} x_i.$$

Read this as "the sum of all x_i for $i = 0, 1, \cdots, n-1$ is x". The index i being summed over is called a **dummy index**. It can be whatever we want since it never appears on the left-hand side. It gets summed over and then disappears. The lower and upper values $i = 0$ and $i = n-1$ are the **limits** of the summation. The limits need not always be $i = 0$ and $i = n-1$. We can choose them to be whatever we like as a matter of convenience.

Frequently summation notation is paired with some kind of **generating function** $f(i) = x_i$ that generates the sequence. For example, suppose our sequence is generated by the function $f(i) = i$, and we want to sum from $i = 1$ to $i = n$. We'd have

$$x = \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{1}{2} n(n+1).$$

The right-hand term $\frac{1}{2}n(n-1)$ is not obvious, and only applies to this particular sum. I just wrote it down since it's sometimes useful to remember. This is a special kind of sum called an **arithmetic series**. Here's a "proof" of this relationship using sympy.

```
i, n = sp.symbols('i n')
summation = sp.Sum(i, (i, 1, n)).doit()
print(f'sum i for i=1,...,n = {summation}')
```

```
sum i for i=1,...,n = n**2/2 + n/2
```

In the general case when we don't have nice rules like this we'd have to loop over the entire sum and do the sum incrementally.

In python, the equivalent of summation notation is the `sum` function, where we pass in the sequence we want to sum as a list. Here's the arithmetic sum up to $n = 10$, which should be $\frac{1}{2}10 \cdot (10+1) = 55$.

```
sum([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

```
55
```

Another useful sum to be aware of is the **geometric series**. A geometric series is a sum over a sequence whose generating function is $f(i) = r^i$ for some real number $r \neq 1$. Its rule is

$$x = \sum_{i=0}^{n-1} r^i = r^0 + r^1 + \cdots + r^{n-1} = \frac{1-r^n}{1-r}.$$

For example, if $n = 10$ and $r = \frac{1}{2}$, we have

$$x = \sum_{i=0}^{9} \left(\frac{1}{2}\right)^i = \frac{1-\left(\frac{1}{2}\right)^{10}}{1-\left(\frac{1}{2}\right)} = 2\left(1 - \frac{1}{2^{10}}\right) \approx 1.998.$$

```
r = 1 / 2
n = 10
sum([r ** i for i in range(n)])
```

```
1.998046875
```

Notice how the term $\left(\frac{1}{2}\right)^{10} \approx 0.00098$ is really small. We can practically ignore it. In fact, as $n \to \infty$ we can completely ignore it, in which case

$$x = \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i = \frac{1}{1 - \left(\frac{1}{2}\right)} = 2.$$

This is an example of the infinite version of the geometric series. If $0 \leq r \leq 1$, then

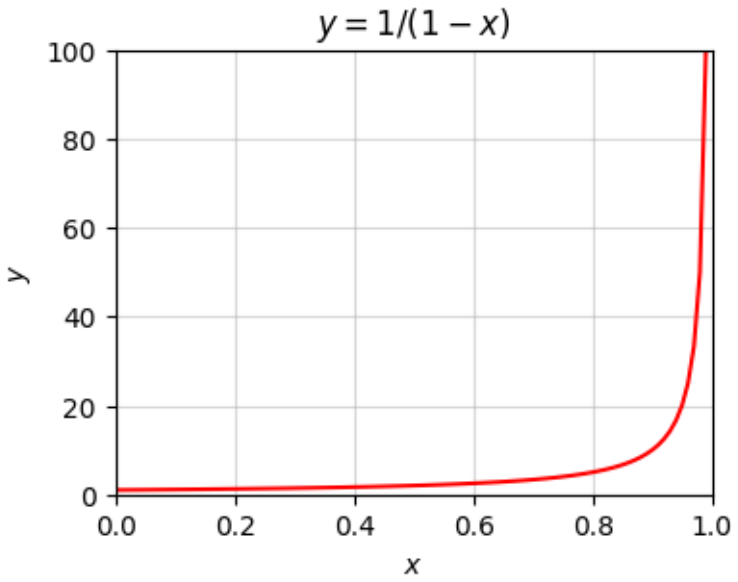
$$x = \sum_{i=0}^{\infty} r^i = r^0 + r^1 + r^2 + \cdots = \frac{1}{1-r}.$$

What happens when $r = 1$? Clearly the rule breaks down at this point, since the denominator becomes infinite. But it's easy enough to see what it is by writing out the sum,

$$x = \sum_{i=0}^{n-1} 1^i = 1^0 + 1^1 + \cdots + 1^{n-1} = \underbrace{1 + 1 + \cdots + 1}_{\text{n times}} = n.$$

In this case, if we send $n \to \infty$, then x clearly blows up to ∞ too. You can see this by plotting the function $y = \frac{1}{1-x}$ and observing it asymptotes at $x = 1$.

```
x = np.arange(0, 1, 0.01)
f = lambda x: 1 / (1 - x)
plot_function(x, f, xlim=(0, 1), ylim=(0, 100), ticks_every=[0.2, 20],
              title='$y=1/(1-x)$')
```



We can always factor constants c out of sums. This follows naturally from just expanding the sum out,

$$\sum_{i=0}^{n-1} cx_i = cx_0 + cx_1 + \cdots + cx_{n-1} = c(x_0 + x_1 + \cdots + x_{n-1}) = c\sum_{i=0}^{n-1} x_i.$$

Similarly, we can break sums up into pieces (or join sums together) as long as we're careful to get the index limits right,

$$\sum_{i=0}^{n-1} x_i = \sum_{i=0}^{k} x_i + \sum_{i=k+1}^{n-1} x_i.$$

We can have double sums (sums of sums) as well. If $x_{i,j}$ is some 2-index variable where $i = 0, \cdots, n-1$ and $j = 0, \cdots, m-1$, we can sum over both sets of indices to get $n \cdot m$ total terms,

$$\sum_{i=0}^{n-1}\sum_{j=0}^{m-1} x_{i,j} = \sum_{j=0}^{m-1}\sum_{i=0}^{n-1} x_{i,j} = x_{0,0} + x_{0,1} + \cdots x_{0,m-1} + \cdots + x_{n-1,0} + x_{n-1,1} + \cdots x_{n-1,m-1}.$$

Notice the two sums can swap, or **commute**, with each other, $\sum_i \sum_j = \sum_j \sum_i$. This follows by expanding the terms out like on the right-hand side and noting the must be equal in both cases.

1.5.2 Products

The notation I've covered for sums has an analogue for products, called **product notation**. Suppose we want to multiply n numbers $x_0, x_1, \cdots, x_{n-1}$ together to get some number x. We could write

$$x = x_0 \cdot x_1 \cdots x_{n-1},$$

but we have a more compact notation for this as well. Using the symbol $\prod$ in analogy to $\sum$, we can write

$$x = \prod_{i=0}^{n-1} x_i.$$

Read this as "the product of all x_i for $i = 0, 1, \cdots, n-1$ is x".

Unlike sums, python doesn't have a native function to calculate products of elements in a sequence, but numpy has one called `np.prod`. Here's an example. I'll calculate the product of all integers between one and ten.

$$x = \prod_{i=1}^{10} i = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 \cdot 8 \cdot 9 \cdot 10 = 3628800.$$

```
np.prod([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

```
3628800
```

Luckily, there aren't any common products to remember. It's just worth being familiar with the notation, since we'll occasionally use it.

Products don't obey quite the same properties sums do, so you have to be careful. When in doubt, just write out the product the long way and make sure what you're doing makes sense. For example, pulling a factor c out of a product gives a factor of c^n, not c, since there are c total products multiplied together,

$$x = \prod_{i=0}^{n-1} cx_i = cx_0 \cdot cx_1 \cdots cx_{n-1} = c^n(x_0 \cdot x_1 \cdots x_{n-1}) = c^n \prod_{i=0}^{n-1} x_i.$$

It's worth noting (because we'll use this fact), that we can turn products into sums by taking the log of the product,

$$\log\left(\prod_{i=0}^{n-1} x_i\right) = \sum_{i=0}^{n-1} \log x_i.$$

This follows from the rule $\log(x \cdot y) = \log x + \log y$, which extends to arbitrarily many products too.

1.6 Greek Alphabet

Like many other technical fields, machine learning makes heavy use of the Greek alphabet to represent variable names in mathematical equations. While not all Greek characters are used, certain ones are worth being aware of. Below is a table of the Greek letters. You don't need to memorize all of these letters, but it's worth referencing this table whenever you encounter a symbol you don't recognize.

2 Numerical Computation

In this lesson I'll discuss the basics of numerical computation. This includes how numbers are represented on a computer, as well as the topics of arrays and vectorization, which is the use of efficient array operations to speed up computations.This may seem too basic to mention, but it's actually very important. There's a lot of subtlety involved. Let's get started.

```
import numpy as np
from utils.math_ml import *
```

2.1 Integers

2.1.1 Basics

Recall the **integers** are whole numbers that can be positive, negative, or zero. Examples are 5, 100151, 0, -72, etc. The set of all integers is commonly denoted by the symbol $\mathbb{Z}$.

In python, integers (ints for short) are builtin objects of type `int` that more or less follow the rules that integers in math follow.

Among other things, the following operations can be performed with integers:

- Addition: $2 + 2 = 4$.
- Subtraction: $2 - 5 = -3$.
- Multiplication: $3 \cdot 3 = 9$
  - In python this is the `*` operator, e.g. `3 * 3 = 9`
- Exponentiation: $2^3 = 2 \times 2 \times 2 = 8$
  - In python this is the `**` operator, e.g. `2 ** 3 = 8`.
- Remainder (or Modulo): the remainder of 10 when divided by 3 is 1, written $10 \bmod 3 = 1$
  - In python this is the `%` operator, e.g. `10 % 3 = 1`.

If any of these operations are applied to two integers, the output will itself always be an integer.

Here are a few examples.

```
2 + 2
```

```
4
```

```
2 - 5
```

```
-3
```

```
3 * 3
```

```
9
```

```
10 % 3
```

```
1
```

```
2 ** 3
```

```
8
```

What about division? You can't always divide two integers and get another integer. What you have to do instead is called integer division. Here you divide the two numbers and then round the answer down to the nearest whole number. Since $5 \div 2 = 2.5$, the nearest rounded down integer is 2.

In math, this "nearest rounded down integer" 2 is usually called the **floor** of 2.5, and represented with the funny symbol $\lfloor 2.5 \rfloor$. Using this notation we can write the above integer division as

$$\lfloor \frac{5}{2} \rfloor = 2.$$

In python, integer division is done using the `//` operator, e.g. `5 // 2 = 2`. I'll usually write 5 // 2 instead of $\lfloor \frac{5}{2} \rfloor$ when it makes sense,

$$5 \text{ // } 2 = \lfloor \frac{5}{2} \rfloor = 2.$$

```
5 // 2
```

```
2
```

We can also do regular division `/` with ints, but the output will *not* be an integer even if the answer should be, e.g. `4 / 2`. Only integer division is guaranteed to return an integer. I'll get to this shortly.

```
4 / 2
type(4 / 2)
```

```
2.0
```

```
float
```

```
4 // 2
type (4 // 2)
```

```
2
```

```
int
```

Division by zero is of course undefined for both division and integer division. In python it will always raise a `ZeroDivisionError` like so.

```
4 / 0
```

```
ZeroDivisionError: division by zero
```

```
4 // 0
```

```
ZeroDivisionError: integer division or modulo by zero
```

2.1.2 Representing Integers

Just like every other data type, on a computer integers are actually represented internally as a sequence of bits. A **bit** is a "binary digit", 0 or 1. A sequence of bits is just a sequence of zeros and ones, e.g. 0011001010 or 1001001.

The number of bits used to represent a piece of data is called its **word size**. If we use a word size of n bits to represent an integer, then there are 2^n possible integer values we can represent.

If integers could only be positive or zero, representing them with bits would be easy. We could just convert them to binary and that's it. To convert a non-negative integer to binary, we just need to keep dividing it by 2 and recording its remainder (0 or 1) at each step. The binary form is then just the sequence of remainders, written right to left. More generally, the binary sequence of some arbitrary number x is the sequence of coefficients $b_k = 0, 1$ in the sum

$$x = \sum_{k=-\infty}^{\infty} b_k 2^k = \cdots + b_2 2^2 + b_1 2^1 + b_0 2^0 + b_{-1} 2^{-1} + b_{-2} 2^{-2} + \cdots .$$

Here's an example. Suppose we wanted to represent the number 12 in binary.

1. $12 \;//\; 2 = 6$ with a remainder of $0 = 12 \bmod 2$, so the first bit from the right is then 0.
2. $6 \;//\; 2 = 3$ with a remainder of $0 = 6 \bmod 2$, so the next bit is 0.
3. $3 \;//\; 2 = 1$ with a remainder of $1 = 3 \bmod 2$, so the next bit is 1.
4. $1 \;//\; 2 = 0$ with a remainder of $1 = 1 \bmod 2$, so the next bit is 1.

So the binary representation of 12 is 1100, which is the sequence of coefficients in the sum

$$12 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0.$$

Rather than keep doing these by hand, you can quickly convert a number to binary in python by using `bin`. It'll return a string representing the binary sequence of that number, prepended with the special prefix `0b`. To get back to the integer from, use `int`, passing in a base of `2`.

```
bin(12)
```

```
'0b1100'
```

```
int('0b110', 2)
```

```
6
```

This representation works fine for non-negative integers, also called the **unsigned integers** in computer science. To represent an unsigned integer with n bits, just get its binary form and prepend it with enough zeros on the left until all n bits are used. For example, if we used 8-bit unsigned integers then $n = 8$, hence representing the number 12 would look like 00000110. Simple, right?

Unsigned ints work fine if we never have to worry about negative numbers. But in general we do. These are called the **signed integers** in computer science. To represent signed ints, we need to use one of the bits to represent the sign. What we can do is reserve the left-most bit for the sign, 0 if the integers is positive or zero, 1 if the integer is negative.

For example, if we used 8-bit *signed* integers to represent 12, we'd again write 00000110, exactly as before. But this time it's understood that left-most 0 is encoding the fact that 12 is positive. If instead we wanted to represent the number -12 we'd need to flip that bit to a 1, so we'd get 10000110.

Let's now do an example of a simple integer system. Consider the system of 4-bit signed ints. In this simple system, $n = 4$ is the word size, and an integer x is represented with the sequence of bits

$$x \equiv sb_1b_2b_3,$$

where s is the sign bit and $b_1b_2b_3$ are the remaining 3 bits allowed to represent the numerical digits. This system can represent $2^4 = 16$ possible values in the range $[-2^3+1, 2^3-1] = [-8, 7]$, given in the following table:

Integer	Representation	Integer	Representation
-0	1000	+0	0000
-1	1001	1	0001
-2	1010	2	0010
-3	1011	3	0011
-4	1100	4	0100
-5	1101	5	0101
-6	1110	6	0110
-7	1111	7	0111

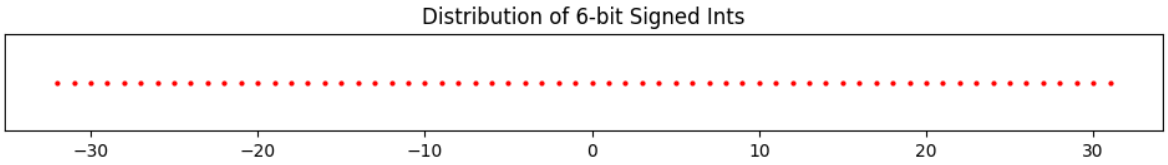
Note the presence of $-0 \equiv 1110$ in the upper left. This is because the system as I've defined it leaves open the possibility of two zeros, $+0$ and -0, since for zero the sign bit is redundant. A way to get around this is to encode the negative numbers slightly differently, by not just setting the sign bit to one, but also inverting the remaining bits and subtracting one from them. This is called the **two's complement representation**. It's how most languages, including python, actually represent integers. I won't go into this representation in any depth, except to say that it gets rid of the need for -0 and replaces it with -2^{n-1}.

Here's what that table looks like for 4-bit integers. It's almost the same, except there's no -0, instead a -8. Notice the positive integers look exactly the same. It's only the negative integers that look different. For them, the right three bits get inverted and added with a one.

Integer	Two's Complement	Integer	Two's Complement
-1	1111	0	0000
-2	1110	1	0001
-3	1101	2	0010
-4	1100	3	0011
-5	1011	4	0100
-6	1010	5	0101
-7	1001	6	0110
-8	1000	7	0111

It's worth visualizing what integers look like on the number line, if for nothing else than to compare it with what floats look like later on. Below I'll plot what a 6-bit signed integer system would look like. Such a system should go from -32 to 31. As you'd expect, we get a bunch of equally spaced points from -32 to 31.

```
n = 6
six_bit_ints = range(-2**(n-1), 2**(n-1))
plot_number_dist(six_bit_ints, title=f'Distribution of {n}-bit Signed Ints')
```



In python, integers are represented by default using a much bigger word size of $n = 64$ bits, called **long** integers, or **int64** for short. This means (using two's complement) we can represent $2^{64} = 18446744073709551616$ possible integer values in the range $[-2^{63}, 2^{63} - 1]$.

You can see from this that 64-bit integers have a minimum integer allowed and a maximum integer allowed, which are

$$\text{min\_int} = -2^{63} = -9223372036854775808, \qquad \text{max\_int} = 2^{63} - 1 = 9223372036854775807.$$

What I've said is technically only exactly true in older versions of pythons as well as other programming languages like C. It turns out newer versions of python have a few added tricks

that allow you to represent essentially arbitrarily large integers. You can see this by comparing it to numpy's internal int64 representation, which uses the C version. A numpy int64 outside the valid range will throw an overflow error.

```
min_int = -2 ** 63
max_int = 2 ** 63 - 1
```

```
min_int - 1
np.int64(min_int - 1)
```

```
-9223372036854775809
```

```
OverflowError: Python int too large to convert to C long
```

```
max_int + 1
np.int64(max_int + 1)
```

```
9223372036854775808
```

```
OverflowError: Python int too large to convert to C long
```

2.2 Floats

2.2.1 Basics

What if we want to represent decimal numbers or fractions instead of whole numbers, like 1.2 or 0.99999, or even irrational numbers like $\pi = 3.1415926\ldots$? To do this we need a new system of numbers that I'll call floating point numbers, or **floats**, for reasons I'll explain soon. Floats will be a computer's best attempt to represent the real numbers $\mathbb{R}$. They'll represent real numbers only approximately with some specified precision.

In python, floats are builtin objects of type `float`. Floats obey pretty much the same operations that integers do with some minor exceptions:

- Addition: $1.2 + 4.3 = 5.5$.
- Subtraction: $1.2 - 4.3 = -3.1$.
- Multiplication: $1.2 \times 4.3 = 5.16$.
- Exponentiation: $4.3^2 = 18.49$.

- Remainder (or Modulo): $4.3 \bmod 1.2 = 0.7$.
- Integer Division: $4.3 \,//\, 1.2 = 3.0$.
- Division: $4.3 \div 1.2$.

Let's verify the first few of these to see what's going on.

```
1.2 + 4.3
```

```
5.5
```

```
1.2 - 4.3
```

```
-3.0999999999999996
```

```
1.2 * 4.3
```

```
5.159999999999999
```

Most of them look right. But what the heck is going on with $1.2 - 4.3$ and 1.2×4.3? We're getting some weird trailing nines that shouldn't be there. This gets to how floats are actually represented on a computer.

2.2.2 Representing Floats

Representing real numbers on a computer is a lot more subtle than representing integers. Since a computer can only have a finite number of bits, they can't represent infinitely many digits, e.g. in irrational numbers like π. Using finite word sizes will necessarily have to truncate real numbers to some number of decimal places. This truncation will create an error in the calculation called **numerical roundoff**.

So how should we represent a decimal number using n bits? As an example, let's imagine we're trying to represent the number $x = 157.208$. Perhaps the first thing you might think of is to use some number of those bits to represent the integer part, and some number to represent the fractional part. Suppose you have $n = 16$ bits available to represent x. Then maybe you can use 8 bits for the integer part 157, and 8 bits for the fractional part 0.208. Converting both halves to binary, you'd get

$$157 \equiv 10011101, \quad 0.208 \equiv 0011010100111111.$$

Truncating both sequences to 8 bits (from the left), you could thus adopt a convention that $157.208 \equiv 10011101\ 00110101$.

This system is an example of a **fixed point** representation. This has to do with the fact that we're always using a fixed number of bits for the integer part, and a fixed number for the fractional part. The decimal point isn't allowed to **float**, or move around to allocate more bits to the integer or fractional part depending which needs more precision. The decimal point is **fixed**.

As I've suggested, the fixed point representation seems to be limited and not terribly useful. If you need really high precision in the fractional part, your only option is to use a larger word size. If you're dealing with really big numbers and don't care much about the fractional part, you also need a larger word size so you don't run out of numbers. A solution to this problem is to allow the decimal point to float. We won't allocate a fixed number of bits to represent the integer or fractional parts. We'll design it in such a way that larger numbers give the integer part more bits, and smaller numbers give the fractional part more bits.

The trick to allowing the decimal point to float is to represent not just the digits of a number but also its exponent. Think about scientific notation, where if you have a number like say $x = 1015.23$, you can write it as $1.01523 \cdot 10^3$, or `1.01523e3`. That 3 is the exponent. It says something about how big the number is. What we can do is convert a number to scientific notation. Then use some number of bits to represent the exponent 3 and some to represent the remaining part 1.01523. This is essentially the whole idea behind floating point.

In floating point representation, instead of using scientific notation with powers of ten, it's more typical to use powers of two. When using powers of two, the decimal part can always be scaled to be between 1 and 2, so they look like 1.567 or something like that. Since the 1. part is always there, we can agree it's always there, and only worry about representing the fractional part 0.567. We'll call this term the **mantissa**. Denoting the sign bit as s, the exponent as e, and the mantissa as m, we can thus right any decimal number x in a modified scientific notation of the form

$$x = (-1)^s \cdot (1 + m) \cdot 2^e.$$

Once we've converted x to this form, all we need to do is to figure out how to represent s, m, and e using some number of bits of n, called the floating point **precision**. Assume the n bits of precision allocate 1 bit for the sign, n_e bits for the exponent, and n_m bits for the mantissa, so $n = 1 + n_e + n_m$.

Here are the steps to convert a number x into its n-bit floating point representation.

- Given some number x, get its modified scientific notation form $x = (-1)^s \cdot (1 + m) \cdot 2^e$.
  - Determine the sign of x. If negative, set the sign bit to $s = 1$, else default to $s = 0$. Set $x = |x|$.
  - Keep performing the operation $x = x\ //\ 2$ until $1 \leq x \leq 2$. Keep track of the number of times you're dividing, which is the **exponent** e.

  - The remaining part will be some $1 \leq x \leq 2$. Write it in the form $x = 1 + m$, where m is the mantissa.

- Convert the scientific notation form into a sequence of n bits, truncating where necessary.
  - For reasons I'll describe in a second, it's good to add a **bias** term b to the exponent e before converting the exponent to binary. Let $e' = e + b$ be this modified exponent.
  - Convert each of e' and m into binary sequences, truncated to sizes n_e, and n_m respectively.
  - Concatenate these binary sequences together to get a sequence of $n = 1 + n_e + n_m$ total bits. By convention, assume the order of bit concatenation is the sign bit, then exponent bits, then the mantissa bits.

There are of course other ways you could do it, for example by storing the sequences in a different order. I'm just stating one common way it's done.

Since all of this must seem like Greek, here's a quick example. Let's consider the number $x = 15.25$. We'll represent it using $n = 8$ bits of precision, where $n_e = 4$ is the number of exponent bits, $n_m = 3$ is the number of precision bits, and $b = 10$ is the bias.

- Convert $x = 15.25$ to its modified scientific notation.
  - Since $x \geq 0$ the sign is positive, so $s = 0$.
  - Keep integer dividing x by 2 until it's less than 2. It takes $e = 3$ divisions before $x < 2$.
  - We now have $x = 1.90625 \cdot 2^3$. The mantissa is then $m = (1.90625 - 1) = 0.90625$.
  - In modified scientific notation form we now have $x = (-1)^0 \cdot (1 + 0.90625) \cdot 2^3$.

- Convert everything to binary.
  - Adding the bias to the exponent gives $e' = 3 + 10 = 13$.
  - Converting each piece to binary we get $e' = 13 \equiv 1101$, $m = 0.90625 \equiv 11101$.
  - Since m requires more than $n_m = 3$ bits to represent, truncate off the two right bits to get $m \equiv 111$.
    - This truncation will cause numerical roundoff, since 0.90625 truncates to 0.875. That's an error of 0.03125 that gets permanently lost.
  - The final representation is thus $x \equiv 0\ 1101\ 111$.

Below I show the example I just calculated. I print out both the scientific notation form and its binary representation.

```
represent_as_float(15.25, n=8, n_exp=4, n_man=3, bias=10)
```

```
scientific notation: (-1)^0 * (1 + 0.90625) * 2^3
8-bit floating point representation: 0 1101 111
```

So what's going on with the bias term b? Why do we need it? The easiest answer to give is that without it, we can't have negative exponents without having to use another sign bit for them. Consider a number like $x = 0.5$. In modified scientific notation this would look like $x = (-1)^0 \cdot (1 + 0) \cdot 2^{-1} = 2^{-1}$, meaning its exponent would be $e = -1$. Rather than have to keep yet another sign bit for the exponent, it's easier to just add a bias term b that ensures the exponent $e' = e + b$ is always non-negative. The higher the bias, the more precision we can show in the range $-1 < x < 1$. The trade-off is that we lose precision for large values of x.

On top of floats defined the way I mentioned, we also have some special numbers that get defined in a floating point system. These are ± 0, $\pm\infty$, and NaN or "not a number". Each of these numbers is allocated its own special sequence of bits, depending on the precision.

- $+0$ and -0: These numbers are typically represented using a biased exponent $e' = 0$ (all zero bits) and a mantissa $m = 0$ (all zero bits). The sign bit is used to distinguish between $+0$ and -0. In our example, these would be $+0 \equiv$ 0 0000 000 and $-0 \equiv$ 1 0000 000.
- $+\infty$ and $-\infty$: These numbers are typically represented using the max allowed exponent (all one bits) and a mantissa $m = 0$ (all zero bits). The sign bit is used to distinguish between $+\infty$ and $-\infty$. In our example, these would be $+\infty \equiv$ 0 1111 000 and $-\infty \equiv$ 1 1111 000.
- NaN: This value is typically represented using the max allowed exponent (all one bits) and a non-zero $m \neq 0$. The sign bit is usually not used for NaN values. Note this means we can have many different sequences that all represent NaN. In our example, any number of the form NaN $\equiv$ x 1111 xxx would work.

So I can illustrate some points about how floating point numbers behave, I'm going to generate *all possible* 8-bit floats (excluding the special numbers) and plot them on a number line, similar to what I did above with the 8-bit signed integers. I'll generate the floats using the using the helper function `gen_all_floats`, passing in the number of mantissa bits `n_man=3`, the number of exponent bits `n_exp=4`, and a bias of `bias=10`.

First, I'll use these numbers to print out some interesting statistics of this 8-bit floating point system.

```
eight_bit_floats = gen_all_floats(n=8, n_man=3, n_exp=4, bias=10)
print(f'Total number of 8-bit floats: {len(eight_bit_floats)}')
print(f'Most negative float: {min(eight_bit_floats)}')
print(f'Most positive float: {max(eight_bit_floats)}')
print(f'Smallest nonzero float: {min([x for x in eight_bit_floats if x > 0])}')
print(f'Machine Epsilon: {min([x for x in eight_bit_floats if x > 1]) - 1}')
```

```
Total number of 8-bit floats: 120
Most negative float: -56.0
```

```
Most positive float: 56.0
Smallest nonzero float: 0.001953125
Machine Epsilon: 0.25
```

We can see that this 8-bit system only contains 120 unique floats. We could practically list them all out. Just like with the integers, we see there's a most negative float, -56.0, and a most positive float, 56.0. The smallest float, i.e. the one closest to 0, is 0.001953125. Notice how much more precision the smallest float has than the largest ones do. The largest ones are basically whole numbers, while the smallest one has nine digits of precision. Evidently, floating point representations give much higher precision to numbers close to zero than to numbers far away from zero.

What happens if you try to input a float larger than the max, in this case 56.0? Typically it will **overflow**. This will result in either the system raising an error, or the number getting set to $+\infty$, in a sense getting "rounded up". Similarly, for numbers more negative than the min, in this case -56.0, either an overflow error will be raised, or the number will get "rounded down" to $-\infty$.

You have to be careful in overflow situations like this, especially when you don't know for sure which of these your particular system will do. It's amusing to note that python will raise an overflow error, but numpy will round to $\pm\infty$. Two different conventions to worry about. Just as amusing, when dealing with signed integers, it's numpy that will raise an error if you overflow, while python won't care. One of those things…

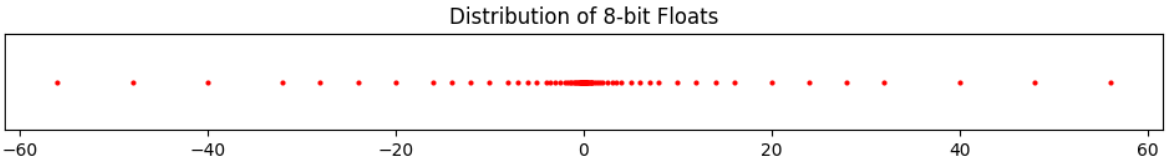
What happens when you try to input a float smaller than the smallest value, in this case 0.001953125? In this case, the number is said to **undeflow**. Usually underflow won't raise an error. The number will pretty much always just get set to $+0$ (or -0). This is again something you have to worry about, especially if you're dealing with small numbers in denominators, where they can lead to division by zero errors which *do* get raised.

Overflow and underflow errors are some of the most common numerical bugs that occur in deep learning, and usually result from not handling floats correctly to begin with.

I also printed out a special value called the **machine epsilon**. The machine epsilon, denoted ε_m, is defined as the smallest value in a floating point system that's larger than 1. In some sense, ε_m is a proxy for how finely you can represent numbers in a given n-bit floating point system. The smaller ε_m the more precisely you can represent numbers, i.e. the more decimal places of precision you get access to. In our case, we get $\varepsilon_m = 0.25$. This means numbers in 8-bit floating point tend to be 0.25 apart from each other on average, which means we can represent numbers in this system only with a measly 2-3 digits of precision.

With these numbers in hand let's now plot their distribution on the number line. Compare with the plot of the signed integers I did above.

```
plot_number_dist(eight_bit_floats, title='Distribution of 8-bit Floats')
```



Notice how different this plot is from the ones for the signed integers. With the integers, the points were equally spaced. Now points close to 0 are getting represented much closer together than points far from 0. There are 74 of the 120 total points showing up just in the range $[-1, 1]$. That's over half!. Meanwhile, only 22 points total show up in the combined ranges of $[-60, -10]$ and $[10, 60]$. Very strange.

Feel free to play around with different floating point systems by using different choices for `n`, `n_man`, `n_exp`, and `bias`. Be careful, however, not to make `n_exp` too large or you may crash the kernel…

2.2.3 Double Precision

So how does python represent floats? Python by default uses what's called **double precision** to represent floats, also called **float64**. This means $n = 64$ total bits of precision are used, with $n_e = 11$, $n_m = 52$, and bias $b = 1023 = 2^{10} - 1$. Double precision allows for a *much* larger range of numbers than 8-bit precision does:

- The max value allowed is $2^{2^{n_e} - b} = 2^{1025} \approx 10^{308}$.
- The min value allowed is $-2^{2^{n_e} - b} = -2^{1025} \approx -10^{308}$.
- Numbers *outside* the range of about $[-10^{308}, 10^{308}]$ will *overflow*.
- The smallest values allowed are (plus or minus) $2^{-b+1} = 2^{-1022} \approx 10^{-308}$.
  - Using subordinal numbers, the smallest values are (plus or minus) $2^{-b-n_m+1} = 2^{-1074} \approx 10^{-324}$.
- Numbers *inside* the range of about $[-10^{-308}, 10^{-308}]$ will *underflow*.
  - Using subordinal numbers, this range is around $[-10^{-324}, 10^{-324}]$.
- The machine epsilon is $\varepsilon_m = 2^{-53} \approx 10^{-16}$.
- Numbers requiring more than about 15-16 digits of precision will get truncated, resulting in numerical roundoff.
- The special numbers $\pm\infty$, ± 0, and NaN are represented similarly as before, except using 64 bits.

To illustrate the point regarding numerical roundoff, here's what happens if we try to use double precision floating point to define the constant π to its first 100 digits? Notice it just gets truncated to its first 15 digits. Double precision is unable to keep track of the other 85 digits. They just get lost to numerical roundoff.

```
pi = 3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862
pi
```

```
3.141592653589793
```

Another thing to worry about is adding small numbers to medium to large sized numbers, e.g. $10 + 10^{-16}$, which will just get rounded down to 10.0.

```
10.0 + 1e-16
```

```
10.0
```

Numerical roundoff is often an issue when subtracting two floats. Here's what happens when we try to subtract two numbers that should be equal, $x = 0.1 + 0.2$ and $y = 0.3$. Instead of $y - x = 0$, we get $y - x \approx -5.55 \cdot 10^{-17}$. The problem comes from the calculation $x = 0.1 + 0.2$, which caused a slight loss of precision in x.

```
x = 0.1 + 0.2
y = 0.3
y - x
```

```
-5.551115123125783e-17
```

A major implication of these calculations is that you should *never* test floating points for exact equality because numerical roundoff can mess it up. If you'd tried to test something like `(y - x) == 0.0`, you'd have gotten the wrong answer. Instead, you want to test that `y - x` is less than some small number `tol`, called a *tolerance*, i.e. `abs(y - x) < tol`.

```
y - x == 0.0
```

```
False
```

```
tol = 1e-5
abs(y - x) < tol
```

```
True
```

Numerical roundoff explains why we got the weird results above when subtracting $1.2 - 4.3$. The imperfect precision in the two numbers resulted in a numerical roundoff error, leading in the trailing 9s that should've rounded up to -3.1 exactly. In general, subtracting floats is one of the most dangerous operations to do, as it tends to lead to the highest loss of precision in calculations. The closer two numbers are to being equal the worse this loss of precision tends to get.

I mentioned that double precision has a smallest number of $2^{-1022} \approx 10^{-308}$, but caveated that by saying that, by using a trick called **subordinal numbers**, we can get the smallest number down to about 10^{-324}. What did I mean by this? It turns out that the bits where the biased exponent $e' = 0$ (i.e. all exponent bits are zero) go mostly unused in the standard version of double precision. By using this zero exponent and allowing the mantissa m to take on all its possible values, we can get about 2^{52} more values (since the mantissa has 52 bits). This lets us get all the way down to $2^{-1022} \cdot 2^{-52} = 2^{-1074} \approx 10^{-324}$.

Python (and numpy) by default implements double precision with subordinal numbers, as we can see.

```
2 ** (-1074)
```

```
5e-324
```

```
2 ** (-1075)
```

```
0.0
```

The special numbers $\pm\infty$, ± 0, and NaN are also defined in double precision. In python (and numpy) they're given by the following commands,

- ∞: `float('inf')` or `np.inf`,
- $-\infty$: `float('-inf')` or `-np.inf`,
- ± 0: `0`,
- NaN: `float('nan')` or `np.nan`.

```
float('inf')
np.inf
```

```
inf
```

```
inf
```

```
float('-inf')
-np.inf
```

```
-inf
```

```
-inf
```

```
0
-0
```

```
0
```

```
0
```

```
float('nan')
np.nan
```

```
nan
```

```
nan
```

You may be curious what exactly NaN ("not a number") is and where it might show up. Basically, NaNs are used wherever values are undefined. Anytime an operation doesn't return a sensible value it risks getting converted to NaN. One example is the operation $\infty - \infty = \infty + (-\infty)$, which mathematically doesn't make sense. No, it's not zero…

```
float('inf') + float('-inf')
np.inf - np.inf
```

```
nan
```

```
nan
```

I'll finish this section by mentioning that there are two other floating point representations worth being aware of: **single precision** (or **float32**), and **half precision** (or **float16**). Single precision uses 32 bits to represent a floating point number. Half precision uses 16 bits. It may seem strange to even bother having these less-precise precisions lying around, but they do have their uses. For example, half precision shows up in deep learning as a more efficient way to represent the weights of a neural network. Since half precision floats only take up 25% as many bits as default double precision floats do, using them can yield a 4x reduction in model memory sizes. We'll see more on this later.

2.2.4 Common Floating Point Pitfalls

To cap this long section on floats, here's a list of common pitfalls people run into when working with floating point numbers, and some ways to avoid each one. This is probably the most important thing to take away from this section. You may find it helpful to reference later on. See this post for more information.

1. Numerical overflow: Letting a number blow up to infinity (or negative infinity)
    - Clip numbers from above to keep them from being too large
    - Work with the log of the number instead
    - Make sure you're not dividing by zero or a really small number
    - Normalize numbers so they're all on the same scale
2. Numerical underflow: Letting a number spiral down to zero
    - Clip numbers from below to keep them from being too small
    - Work with the exp of the number instead
    - Normalize numbers so they're all on the same scale
3. Subtracting floats: Avoid subtracting two numbers that are approximately equal
    - Reorder operations so approximately equal numbers aren't nearby to each other
    - Use some algebraic manipulation to recast the problem into a different form
    - Avoid differencing squares (e.g. when calculating the standard deviation)
4. Testing for equality: Trying to test exact equality of two floats
    - Instead of testing `x == y`, test for approximate equality with something like `abs(x - y) <= tol`
    - Use functions like `np.allclose(x, y)`, which will do this for you
5. Unstable functions: Defining some functions in the naive way instead of in a stable way
    - Examples: factorials, softmax, logsumexp
    - Use a more stable library implementation of these functions
    - Look for the same function but in log form, e.g. `log_factorial` or `log_softmax`

6. Beware of NaNs: Once a number becomes NaN it'll always be a NaN from then on

    - Prevent underflow and overflow
    - Remove missing values or replace them with finite values

2.3 Array Computing

In machine learning and most of scientific computing we're not interested in operating on just single numbers at a time, but many numbers at a time. This is done using *array operations*. The most popular library in python for doing numerical computation on arrays is numpy.

Why not just do numerical computations in base python? After all, if we have large arrays of data we can just put them in a list. Consider the following example. Suppose we have two tables of data, $\mathbf{A}$ and $\mathbf{B}$. Each table has $m = 5$ rows and $n = 3$ columns. The rows represent samples, e.g. measured in a lab, and the columns represent the variables, or *features*, being measured, call them x, y, and z, if you like. I'll define these two tables using python lists `A` and `B` below.

```
A = [[3.5, 18.1, 0.3],
     [-8.7, 3.2, 0.5],
     [-1.3, 8.4, 0.2],
     [5.6, 12.9, 0.9],
     [-6.8, 19.7, 0.7]]

B = [[-9.7, 12.5, 0.1],
     [-5.1, 14.1, 0.6],
     [-1.6, 3.7, 0.7],
     [2.3, 19.3, 0.9],
     [8.2, 9.7, 0.2]]
```

Suppose we wanted to add the elements in these two tables together, index by index, like this,

$$\begin{bmatrix} A[0][0]+B[0][0], & A[0][1]+B[0][1], & A[0][2]+B[0][2] \\ A[1][0]+B[1][0], & A[1][1]+B[1][1], & A[1][2]+B[1][2] \\ A[2][0]+B[2][0], & A[2][1]+B[2][1], & A[2][2]+B[2][2] \\ A[3][0]+B[3][0], & A[3][1]+B[3][1], & A[3][2]+B[3][2] \\ A[4][0]+B[4][0], & A[4][1]+B[4][1], & A[4][2]+B[4][2] \end{bmatrix}.$$

If we wanted to do this in python, we'd have to loop over all rows and columns and place the sums one-by-one inside an array $\mathbf{C}$, like this.

```
def add_arrays(A, B):
    n_rows, n_cols = len(A), len(A[0])
    C = []
    for i in range(n_rows):
        row = []
        for j in range(n_cols):
            x = A[i][j] + B[i][j]
            row.append(x)
        C.append(row)
    return C

C = add_arrays(A, B)
print(f'C = {np.array(C).round(2).tolist()}')
```

```
C = [[-6.2, 30.6, 0.4], [-13.8, 17.3, 1.1], [-2.9, 12.1, 0.9], [7.9, 32.2, 1.8], [1.4, 29.4,
```

Numpy makes this far easier to do. It implements *element-wise* array operatations, which allow us to operate on arrays with far fewer lines of code. In numpy, to perform the same adding operation we just did, we'd just add the two arrays together directly, $\mathbf{A} + \mathbf{B}$.

To use numpy operations we have to convert data into the native numpy data type, the numpy array. Do this by wrapping lists inside the function `np.array`. Once we've done this, we can just add them together in one line. This will simultaneously element-wise add the elements in the array so we don't have to loop over anything.

```
A = np.array(A)
B = np.array(B)
print(f'C = \n{A+B}')
```

```
C =
[[ -6.2  30.6   0.4]
 [-13.8  17.3   1.1]
 [ -2.9  12.1   0.9]
 [  7.9  32.2   1.8]
 [  1.4  29.4   0.9]]
```

This is really nice. We've managed to reduce a double foor loop of 8 lines of code down to just 1 line with no loops at all. Of course, there *are* loops happening in the background inside the numpy code, we just don't see them.

Numpy lets us do this with pretty much any arithmetic operation we can think of. We can element-wise add, subtract, multiply, or divide the two arrays. We can raise them to powers, exponentiate them, take their logarithms, etc. Just like we would do so with single numbers. In numpy, arrays become first class citizens, treated on the same footing as the simpler numerical data types `int` and `float`. This is called **vectorization**.

Here are a few examples of different vectorized functions we can call on `A` and `B`. All of these functions are done element-wise.

```
A - B
```

```
array([[ 13.2,   5.6,   0.2],
       [ -3.6, -10.9,  -0.1],
       [  0.3,   4.7,  -0.5],
       [  3.3,  -6.4,   0. ],
       [-15. ,  10. ,   0.5]])
```

```
A / B
```

```
array([[-0.36082474,  1.448     ,  3.        ],
       [ 1.70588235,  0.22695035,  0.83333333],
       [ 0.8125    ,  2.27027027,  0.28571429],
       [ 2.43478261,  0.66839378,  1.        ],
       [-0.82926829,  2.03092784,  3.5       ]])
```

```
A ** B
```

```
array([[5.27885788e-06, 5.25995690e+15, 8.86568151e-01],
       [           nan, 1.32621732e+07, 6.59753955e-01],
       [           nan, 2.62925893e+03, 3.24131319e-01],
       [5.25814384e+01, 2.71882596e+21, 9.09532576e-01],
       [           nan, 3.60016490e+12, 9.31149915e-01]])
```

```
np.sin(A)
```

```
array([[-0.35078323, -0.68131377,  0.29552021],
       [-0.66296923, -0.05837414,  0.47942554],
       [-0.96355819,  0.85459891,  0.19866933],
       [-0.63126664,  0.32747444,  0.78332691],
       [-0.49411335,  0.75157342,  0.64421769]])
```

If vectorization just made code easier to read it would be a nice to have. But it's more than this. In fact, vectorization also makes your code run much faster in many cases. Let's see an example of this. I'll again run the same operations above to add two arrays, but this time I'm going to **profile** the code in each case. That is, I'm going to time each operation over several runs and average the times. The ones with the lowest average time is faster than the slower one, obviously. To profile in a notebook, the easiest way is to use the `%timeit` magic command, which will do all this for you.

```
A = A.tolist()
B = B.tolist()
%timeit C = add_arrays(A, B)
```

```
2.61 µs ± 8.21 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

```
A = np.array(A)
B = np.array(B)
%timeit C = A + B
```

```
411 ns ± 0.75 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

Even with these small arrays the numpy vectorized array addition is almost 10 times faster than the python loop array addition. This difference becomes much more pronounced when arrays are larger. The arrays just considered are only of shape $(10, 3)$. We can easily confirm this in numpy using the methods `A.shape` and `B.shape`.

```
print(f'A.shape = {A.shape}')
print(f'B.shape = {B.shape}')
```

```
A.shape = (5, 3)
B.shape = (5, 3)
```

Let's try to run the add operations on much larger arrays of shape $(10000, 100)$. To do this quickly I'll use `np.random.rand(shape)`, which will sample an array with shape `shape` whose values are uniformly between 0 and 1. More on sampling in a future lesson. Running the profiling, we're now running about 100 times faster using numpy vectorization compared to python loops.

```
D = np.random.rand(10000, 100)
E = np.random.rand(10000, 100)
```

```
D = D.tolist()
E = E.tolist()
%timeit F = add_arrays(D, E)
```

```
119 ms ± 517 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
D = np.array(D)
E = np.array(E)
%timeit F = D + E
```

```
1.35 ms ± 60.2 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

So why is numpy vectorization so much faster than using native python loops? Because it turns out that numpy by and large doesn't actually perform array operations in python! When array operations are done, numpy compiles them down to low-level C code and runs the operations there, where things are much faster.

Not only that, numpy takes advantage of very efficient linear algebra functions written over the course of decades by smart people. These functions come from low-level FORTRAN and C libraries like BLAS and LAPACK. They're hand-designed to take maximum advantage of computational speed-ups where available. These include things like parallelization, caching, and hardware vectorization operations. Native python doesn't take advantage of *any* of these nice things. The moral is, if you want to run array operations efficiently, you need to use a numerical library like numpy or modern variants like pytorch.

2.3.1 Higher-Dimensional Arrays

The number of different dimensions an array has is called its **dimension** or **rank**. Equivalently, the rank or dimension of an array is just the length of its shape tuple. The arrays I showed above are examples of rank-2 or 2-dimensional arrays. We can define arrays with any number of dimensions we like. These arrays of different rank sometimes have special names:

- A 0-dimensional (rank-0) array is called a **scalar**. These are single numbers.
- A 1-dimensional (rank-1) array is called a **vector**. These are arrays with only one row.
- A 2-dimensional (rank-2) array is called a **matrix**. These are arrays with multiple rows.
- An array of dimension or rank 3 or higher is called a **tensor**. These are arrays with multiple matrices.

More on these when we get to linear algebra. Here are some examples so you can see what they look like. Note I'm using `dtype=np.float64` to explicitly cast the values as float64 when defining the arrays. Numpy's vectorization operations work for all of these arrays regardless of their shape.

```
scalar = np.float64(5)
scalar # 0-dimensional
```

```
5.0
```

```
vector = np.array([1, 2, 3], dtype=np.float64)
print(f'vector.shape = {vector.shape}')
print(f'vector = {vector}')
```

```
vector.shape = (3,)
vector = [1. 2. 3.]
```

```
matrix = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.float64)
print(f'matrix.shape = {matrix.shape}')
print(f'matrix = \n{matrix}')
```

```
matrix.shape = (2, 3)
matrix =
[[1. 2. 3.]
 [4. 5. 6.]]
```

```
tensor = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]], dtype=np.float64)
print(f'tensor.shape = {tensor.shape}')
print(f'tensor = \n{tensor}')
```

```
tensor.shape = (2, 2, 2)
tensor =
[[[1. 2.]
  [3. 4.]]

 [[5. 6.]
  [7. 8.]]]
```

Numpy also supports array aggregation operations as well. Suppose you have a matrix `A` and want to get the sum of the values in each row of `A`. To do this, you could use `np.sum(A, axis=1)`, where `axis` is the index of the dimension you want to sum over (the columns in this case). This will return a vector where the value at index i is the sum of elements in row i. To sum over *all* elements in the array, don't pass anything to `axis`.

```
A = np.array([[1, 2, 3], [-1, -2, -3], [1, 0, -1]], dtype=np.float64)
print(f'A = \n{A}')
print(f'sum over all A = {np.sum(A)}')
print(f'row sums of A = {np.sum(A, axis=1)}')
```

```
A =
[[ 1.  2.  3.]
 [-1. -2. -3.]
 [ 1.  0. -1.]]
sum over all A = 0.0
row sums of A = [ 6. -6.  0.]
```

Indexing into numpy arrays like `A` is more powerful than with python lists. Instead of having to awkwardly index like `A[1][0]`, write `A[1, 0]`. To get all values in column index `1`, write `A[:, 1]`. To get just the first and last row, we could just pass the index we want in as a list like this, `A[[0, -1], :]`.

```
print(f'A[1, 0] = {A[1][0]} = {A[1, 0]}')
```

```
A[1, 0] = -1.0 = -1.0
```

```
print(f'col 1 of A = {A[:, 1]}')
print(f'rows 0 and -1 of A = \n{A[[0, -1], :]}')
```

```
col 1 of A = [ 2. -2.  0.]
rows 0 and -1 of A =
[[ 1.  2.  3.]
 [ 1.  0. -1.]]
```

Numpy also supports Boolean masks as indexes. Suppose we want to get all the positive elements `x >= 0` in `A`. We could create a mask `A > 0`, and pass that into `A` as an index to pick out the positive elements only.

```
print(f'mask of (A >= 0) = \n{(A >= 0)}')
print(f'elements of (A >= 0) = \n{A[A >= 0]}')
```

```
mask of (A >= 0) =
[[ True  True  True]
 [False False False]
 [ True  True False]]
elements of (A >= 0) =
[1. 2. 3. 1. 0.]
```

2.4 Broadcasting

Broadcasting is a set of conventions for doing array operations on arrays with incompatible shapes. This may seem like a strange thing to do, but it turns out knowing how and when to broadcast can make your code much shorter, more readable, and efficient. All modern-day numerical libraries in python support broadcasting, including numpy, pytorch, tensorflow, etc. So it's a useful thing to learn.

2.4.1 Motivation

Let's start with a simple example. Suppose we have an array of floats defined below. We'd like to add 1 to every number in the array. How can we do it? One "pythonic" way might be to use a list comprehension like so. This will work just fine, but it requires going back and forth between arrays and lists.

```
x = np.array([1., 2., 3., 4., 5.])
print(f'x = {x}')

x_plus_1 = np.array([val + 1 for val in x])
print(f'x + 1 = {x_plus_1}')
```

```
x = [1. 2. 3. 4. 5.]
x + 1 = [2. 3. 4. 5. 6.]
```

What if we didn't want to go back and forth like that? It is slow after all. Anytime numpy has to handoff back to python or vice versa it's going to slow things down. Another thing we could try is to make a vector of ones of the same size as `x`, then add it to `x`. This is also fine, but it requires defining this extra array of ones just to add 1 to the original array.

```
ones = np.ones(len(x))
x_plus_1 = x + ones
print(f'x + 1 = {x_plus_1}')
```

```
x + 1 = [2. 3. 4. 5. 6.]
```

We'd *like* to be able to just add 1 to the array like we would with numbers. If `x` were a single number we'd just write `x + 1` to add one to it, right? But technically we can't do this if `x` is an array, since `x` has shape `(5,)` and 1 is just a number with no shape. This is where broadcasting comes in. Broadcasting says let's *define* the operation `x + 1` so that it *means* add 1 to every element of `x`.

```
x_plus_1 = x + 1
print(f'x + 1 = {x_plus_1}')
```

```
x + 1 = [2. 3. 4. 5. 6.]
```

This notation has the advantage of keeping array equations simple, while at the same time keeping all operations in numpy so that they run fast.

2.4.2 Broadcasting Rules

Suppose now that we have two arrays `A` and `B` of arbitrary shape and we want to operate on them, e.g. via the operations `+, -, *, /, //, **`. Here are the general broadcasting rules, quoted directly from the numpy documentation.

> **Numpy Documentation**When operating on two arrays, numpy compares their shapes element-wise. It starts with the trailing (i.e. rightmost) dimensions and works its way left. Two dimensions are **compatible** when 1. they are equal, or2. one of them is 1 If these conditions are not met, a `ValueError: operands could not be broadcast together` exception is thrown, indicating that the arrays have **incompatible** shapes. The size of the resulting array is the size that is not 1 along each axis of the inputs.

Let's look at an example. First, suppose `A` has shape `(2, 2, 3)` and `B` has shape `(3,)`. Suppose for simplicity that they're both arrays of all ones. Here's what this looks like, with `B` aligned to the right.

A :	2,	2,	3
B :			3
C :	2,	2,	3

Here are the broadcasting steps that will take place. Note that only `B` will change in this example. `A` will stay fixed.

- Numpy will start in the rightmost dimension, checking if they're equal.
- Begin with `A` of shape `(2, 2, 3)` and `B` of shape `(3,)`.
- In this case, the rightmost dimension is `3` in both arrays, so we have a match.
- Moving left by one, `B` no longer has anymore dimensions, but `A` has two, each `2`. These arrays are thus compatible.
- Numpy will now copy `B` to the left in these new dimensions until it has the same shape as `A`.
  - Copy values of B twice to get `B = [[1, 1, 1], [1, 1, 1]]` with shape `(2, 3)`.
  - Copy values of B twice again to get `B = [[[1, 1, 1], [1, 1, 1]], [[1, 1, 1], [1, 1, 1]]]` with shape `(2, 2, 3)`.
- The shapes of A and B are now equal. The output array `C` will have shape `(2, 2, 3)`.

Let's verify this is true on two simple arrays of ones. Let's also print out what `C` looks like. Since only copying is taking place we should just be adding 2 arrays of ones, hence the output should sum 2 arrays of ones, giving one array `C` of twos.

```
A = np.ones((2, 2, 3))
B = np.ones(3,)
print(f'A.shape = {A.shape}')
print(f'B.shape = {B.shape}')

C = A + B
print(f'C.shape = {C.shape}')
print(f'C = \n{C}')
```

```
A.shape = (2, 2, 3)
B.shape = (3,)
```

```
C.shape = (2, 2, 3)
C =
[[[2. 2. 2.]
  [2. 2. 2.]]

 [[2. 2. 2.]
  [2. 2. 2.]]]
```

Let's do one more example. Suppose now that `A` has shape `(8, 1, 6, 1)` and `B` has shape `(7, 1, 5)`. Here's a table of this case, again with `B` aligned to the right since it has the fewest dimensions.

$A:$	8,	1,	6,	1
$B:$		7,	1,	5
$C:$	8,	7,	6,	5

Here are the broadcasting steps that will take place.

- Starting again from the right, dimensions `1` and `5` don't match. But since `A` has a `1` rule (2) applies, so `A` will broadcast itself (i.e. copy its values) 5 times in this dimension to match `B`.
- Moving left by one we get `6` and `1`. Now `B` will broadcast itself in this dimension 6 times to match `A`.
- Moving left again we get `1` and `7`. Now `A` will broadcast itself in this dimension 7 times to match `B`.
- Last, we get `8` in `A` and `B` is out of dimensions, so `B` will broadcast itself 8 times to match `A`.
- The shapes of `A` and `B` are now equal. The output `C` thus has shape `(8, 7, 6, 5)`.

Here again is an example on two arrays of ones. Verify that the shapes come out right.

```
A = np.ones((8, 1, 6, 1))
B = np.ones((7, 1, 5))
print(f'A.shape = {A.shape}')
print(f'B.shape = {B.shape}')

C = A / B
print(f'C.shape = {C.shape}')
```

```
A.shape = (8, 1, 6, 1)
B.shape = (7, 1, 5)
C.shape = (8, 7, 6, 5)
```

That's pretty much all there is to broadcasting. It's a systematic way of trying to copy the dimensions in each array until they both have the same shape. All this broadcasting is done under the hood for you when you try to operate on two arrays of different shapes. You don't need to do anything but understand *how* the arrays get broadcast together so you can avoid errors in your calculations, sometimes very subtle errors.

This can be a bit confusing to understand if you're not used to it. We'll practice broadcasting a good bit so you can get the hang of it.

3 Calculus

In this lesson, I'll cover the basics of the calculus of a single variable. Calculus is essentially the study of the continuum. Important things that calculus seeks to understand are:

- Infinitesimals: How to manipulate numbers that are "infinitely" small.
- Differentiation: How one variable changes continuously in response to one or more other variables.
- Integration: How to add up infinitely many small numbers to get a finite number.
- Limits: What happens as one value gets closer to another.

Not all of these topics are equally important to know for machine learning, but I'll try to at least touch on each topic a little bit. Let's get started.

```
import numpy as np
import sympy as sp
import matplotlib.pyplot as plt
from utils.math_ml import *
import warnings

warnings.filterwarnings('ignore')
plt.rcParams["figure.figsize"] = (4, 3)
```

3.1 Infinitesimals

Fundamental to the understanding of calculus is the idea of an "infinitely small" number, called an infinitesimal. An **infinitesimal** is a number that's not 0 but so close to being 0 that you can't really tell it isn't 0. These small numbers are often written in math with letters like ε or δ. Think of them as *very very* tiny numbers, so tiny their square is basically 0:

$$\varepsilon > 0, \quad \varepsilon^2 \approx 0.$$

But what does this even mean? Here it might be helpful to recall our discussion of floating point numbers. Recall that we can't get infinite precision. In python's double precision floating point we can only get down to about $5 \cdot 10^{-324}$, or `5e-324`. If the square of a small number has a value *smaller* than about `5e-324` we'd literally get `0.0` as far as python is concerned.

Just for fun let's look at the really tiny number 10^{-300}, or `1e-300`. That's 300 decimal places of zeros before the 1 even shows up. Python thinks `1e-300` is just fine. But what happens if we square it? We should in theory get 10^{-600}, or 600 decimal places of zeros followed by a 1. But as far as floating point is concerned, the square is zero.

```
epsilon = np.float64(1e-300)
print(f'epsilon = {epsilon}')
print(f'epsilon^2 = {epsilon ** 2}')
```

```
epsilon = 1e-300
epsilon^2 = 0.0
```

Of course, you could argue that we could just go to a higher precision then. Use more bits. But eventually, if we keep making ε small enough we'll hit a point where $\varepsilon^2 = 0$. Thus, if it makes you feel better, when you see an infinitesimal just think "10^{-300} in double precision".

Aside: If you want to be *really* pedantic, you might say that it shouldn't matter what a computer does, since any positive number ε squared must still be greater than zero, no matter how small ε is. This is true for *real numbers* $\mathbb{R}$. But it turns out infinitesimals aren't real numbers at all. They lie in an extension of the real number line called the **hyperreal numbers**, denoted $\mathbb{R}^*$. In my opinion, this isn't an important distinction to worry about in applied calculus.

3.1.0.1 Infinitely Large Numbers

Similar to infinitesimals being numbers that can be really, really small, we can also talk about numbers being really, really big. These are called **infinitely large** numbers. In analogy to infinitesimals, infinitely large numbers are positive numbers N whose square is basically infinite,

$$N > 0, \quad N^2 \approx \infty.$$

We can get infinitely large numbers by inverting infinitesimals, and vice versa,

$$N = \frac{1}{\varepsilon}, \quad \varepsilon = \frac{1}{N}.$$

If 10^{-300} is a good rule of thumb for an infinitesimal, then 10^{300} is a good rule of thumb for an infinitely large number.

```
N = np.float64(1e300)
print(f'N = {N}')
print(f'N^2 = {N ** 2}')
```

```
N = 1e+300
N^2 = inf
```

3.1.0.2 First-Order Perturbations

Infinitesimals are especially interesting when added to regular numbers. These are called **first order perturbations**. For example, consider some finite number x. It could be 2 or -100 or whatever you want. Suppose now we add to it an infinitesimal number ε. Now suppose we have an output y that depends on x through a function $y = f(x) = x^2$. What happens to y if we perturb x to $x+\varepsilon$? That is, what is $f(x+\varepsilon) = (x+\varepsilon)^2$? Expanding the square, we have

$$f(x+\varepsilon) = (x+\varepsilon)^2 = x^2 + 2x\varepsilon + \varepsilon^2.$$

But since $\varepsilon^2 \approx 0$,

$$f(x+\varepsilon) = (x+\varepsilon)^2 \approx x^2 + 2x\varepsilon.$$

Okay, but what does this mean? Well, I can reformulate the question as follows: "If I change x by a little bit, how much does the function y change"? Call this change δ, the change in y due to x getting changed by ε. Since $\delta = f(x+\varepsilon) - f(x)$ by definition, we'd have

$$\delta = f(x+\varepsilon) - f(x) = (x+\varepsilon)^2 - x^2 \approx 2x\varepsilon.$$

That is, if we change x by a small amount ε, then y itself changes by a small amount $\delta = 2x\varepsilon$. Interestingly, how much y changes actually depends on which x we pick. If $x = 1$ then y changes by 2ε, just twice how much x is nudged. If $x = 1000$ though, then y changes by 2000ε, a much bigger change, but still infinitesimal. After all, $2000 \cdot 10^{-300} = 2 \cdot 10^{-297}$ is still really, really small.

3.2 Differentiation

3.2.1 Derivatives

When we talk about changing x by a little bit and asking how y changes we use a cleaner notation. Instead of writing $x+\varepsilon$, we'd write $x+dx$. Instead of writing $y+\delta$, we'd write $y+dy$. These values dx and dy are called **differentials**. Differentials are just like the infinitesimals I defined before, except the differential notation makes it clear what is a small change of what. Writing dx means "a little bit of x". Writing dy means "a little bit of y". This is where the term "differentiation" comes from.

Suppose x gets nudged a little bit to $x+dx$. Then $y = f(x)$ gets nudged to $y+dy = f(x+dx)$. Since $dy = f(x+dx) - f(x)$, evidently differentials can be thought of as the difference of two values infinitesimally close to each other. Anyway, re-expressing our example in terms of differentials, we get

$$y + dy = f(x + dx) = (x + dx)^2 = x^2 + 2xdx + (dx)^2 \approx x^2 + 2xdx,$$

or $dy = 2xdx$. This is how much y changes in response to the perturbation $x + dx$. Just for the hell of it, let's divide both sides by dx. Then we get

$$\frac{dy}{dx} = 2x.$$

This ratio of differentials is called the *derivative* of the function $y = x^2$. It's often pronounced "dydx" or "the derivative of y with respect to x". Notice there's no differential on the right-hand side, hence the derivative is not itself infinitesimal. It's a finite *ratio* of infinitesimals.

We can talk about any reasonably well-behaved function $y = f(x)$ having a derivative. If we change x by an infinitesimal amount dx, then y changes by an amount $dy = f(x+dx) - f(x)$. The **derivative** is just the ratio of these differential changes,

$$\frac{dy}{dx} = \frac{f(x + dx) - f(x)}{dx}.$$

Practically speaking, you can think of the derivative as a *rate* or a speed. It's the rate that y changes per unit x. Roughly speaking, if x changes by one unit, then y will change by $\frac{dy}{dx}$ units. When the derivative is large, y will change a lot in response to small changes in x. When the derivative is small, y will barely change at all in response to small changes in x. The *sign* of the derivative indicates whether the change is up or down.

Notice that the derivative is also itself a function, since it maps inputs x to outputs $\frac{dy}{dx}$. To indicate this functional relationship people sometimes write

$$\frac{dy}{dx} = \frac{d}{dx}f(x) \quad \text{or} \quad \frac{dy}{dx} = f'(x)$$

to make this functional relationship clear.

3.2.1.1 Numerical Differentiation

Now, how do we actually *calculate* a derivative? Perhaps the easiest thing to do is this: Suppose you have some function $y = f(x)$, and you want to find its derivative at a point x. Choose a small value dx. Then just take the ratio

$$\frac{dy}{dx} = \frac{f(x+dx) - f(x)}{dx}.$$

Suppose we wanted to find the derivative of $y = x^2$ at the point $x = 1$. This will return a numerical *value*, not a function, since we're plugging in a particular value of x. Expanding terms exactly, we'd have

$$\left.\frac{dy}{dx}\right|_{x=1} = \left.\frac{(x+dx)^2 - x^2}{dx}\right|_{x=1} = \frac{(1+dx)^2 - 1}{dx}.$$

Notation: Read the expression $|_{x=1}$ as "evaluated at $x = 1$". It's a common shorthand. Another way to write the same thing is $f'(1)$ or $\frac{d}{dx}f(1)$.

The output of this result will depend on what value of dx we choose. If dx were *exactly* infinitesimal, we already know the right answer should be $\left.\frac{dy}{dx}\right|_{x=1} = 2 \cdot 1 = 2$. Let's see what happens to our calculation for different choices of dx ranging from $dx = 1$ all the way down to $dx = 10^{-200}$. I'll also calculate the *error*, which is the predicted value 2 minus the calculated value. Smaller error is better, obviously.

```
f = lambda x: x ** 2
x0 = 1
dydx_exact = 2 * x0
for dx in [1, 0.1, 0.01, 0.001, 1e-4, 1e-5, 1e-10, 1e-100, 1e-200]:
    dy = f(x0 + dx) - f(x0)
    dydx = dy / dx
    error = dydx - dydx_exact
    print(f'dx = {dx:8.16f} \t dy/dx = {dydx:4f} \t error = {error:4f}')
```

```
dx = 1.0000000000000000      dy/dx = 3.000000    error = 1.000000
dx = 0.1000000000000000      dy/dx = 2.100000    error = 0.100000
dx = 0.0100000000000000      dy/dx = 2.010000    error = 0.010000
dx = 0.0010000000000000      dy/dx = 2.001000    error = 0.001000
dx = 0.0001000000000000      dy/dx = 2.000100    error = 0.000100
dx = 0.0000100000000000      dy/dx = 2.000010    error = 0.000010
dx = 0.0000000001000000      dy/dx = 2.000000    error = 0.000000
dx = 0.0000000000000000      dy/dx = 0.000000    error = -2.000000
dx = 0.0000000000000000      dy/dx = 0.000000    error = -2.000000
```

Starting with $dx = 1$ is a bad choice, with a huge error of 1.0. We're way off. Shrinking to $dx = 0.1$ puts us in the ball park with a value $\frac{dy}{dx} = 2.1$. You can see that making dx successively smaller and smaller makes the error successively smaller, in this case by a factor of 10 each time.

The error is getting smaller all the way down to about $dx = 10^{-10}$ before creeping up again as we make dx even smaller than that. This is due to the numerical roundoff of floating point numbers. We're subtracting two numbers $(1 + dx)^2 - 1$ that are very close to each other when dx is really small, which as you'll recall is one of the pitfalls to avoid when working with floating point numbers. For this reason, it's common in practice to choose "less small" values for dx when calculating derivatives numerically like this, e.g. $dx = 10^{-5}$.

This method we just used to calculate the derivative is, with some minor tweeks, exactly how derivatives are usually calculated on a computer. The process of calculating the derivative this way, directly from its definition essentially, is called **numerical differentiation**. We choose a small value of dx and just apply the formula for the derivative directly, with some minor tweaks to get better accuracy.

Aside: When calculating derivatives numerically, you can improve the error a lot by instead taking $dy = \frac{1}{2}(f(x+\frac{dx}{2}) - f(x-\frac{dx}{2}))$. This is called the *central difference method*. It's equivalent when dx is infinitesimal. For more details, see this article on finite difference methods.

3.2.1.2 Existence of the Derivative

What exactly did I mean when I say the function $f(x)$ needs to be "reasonably well behaved" in order to have a derivative? For one thing, the function needs to be **continuous**. Informally, you can think of a univariate function as being continuous if you can draw its graph on a piece of paper without lifting your pen. There are no jumps or holes anywhere in the functions' curve. A better way to say this is that $y = f(x)$ is continuous at a point x_0 provided $f(x) \approx f(x_0)$ whenever $x \approx x_0$.

Continuity is just *one* condition necessary for $f(x)$ to be differentiable. It also can't be too jagged in some sense. Derivatives don't make sense at points where there are kinks in the graph. In practice, however, this isn't a huge problem. We can just extend the derivative

to be what's called a subderivative. With a subderivative, you can roughly speaking take whatever value for the derivative you want at these kinks and it won't make a difference. This is how we in practice calculate the derivatives of neural networks. Typically, a neural network *does* have kinks, and lots of them. At these kinks, we just pick an arbitrary value for the derivative and go on about our day.

3.2.1.3 Second Derivatives

Since derivatives are themselves functions, we can take the derivative of the derivative too. If $\frac{d}{dx}f(x)$ is the derivative function, then the derivative of the derivative is just

$$\frac{d^2y}{dx^2} = \frac{d}{dx}\left(\frac{d}{dx}f(x)\right).$$

This is called a **second derivative**. Other ways to write the same thing are $f''(x)$ or $\frac{d^2}{dx^2}f(x)$.

As a quick example, the second derivative of our running function $y = x^2$ is the first derivative of $2x$, which is

$$\frac{d^2y}{dx^2} = \frac{2(x+dx) - 2x}{dx} = 2.$$

Evidently, the second derivative of $y = x^2$ is just the *constant* function $\frac{d^2y}{dx^2} = 2$.

Just as you can think of a first derivative as a rate or speed, you can think of the second derivative as a rate of a rate, or an acceleration. When the second derivative is large, the function is accelerating quickly, and the derivative or speed is rapidly changing. When the second derivative small, the function is barely accelerating at all, and the derivative or speed is roughly constant.

Aside: Just as I derived an explicit formula for the first derivative, I can do the same for the second derivative. The key is to introduce the *second differentials* $d^2y = d(dy)$ and $dx^2 = (dx)^2$. Think of these as squared infinitesimals, or differences of differences. Since $dy = df(x) = f(x+dx) - f(x)$, we have

$$\begin{aligned} d^2y = d(dy) &= d(f(x+dx) - f(x)) = d(f(x+dx)) - df(x) \\ &= (f((x+dx)+dx) - f(x+dx)) - (f(x+dx) - f(x)) \\ &= f(x+2dx) - 2f(x+dx) - f(x). \end{aligned}$$