

COMPSCI4073 Data Fundamentals H - 2023-24 - Wiki for making course notes

COMPSCI4073 Data Fundamentals H - 2023-24 - Course notes

TABLE OF CONTENTS

1. [Week 1: numerical basics](#)
 - 1.1. [Uses of arrays and array operations](#)
 - 1.2. [Vectorisation](#)
 - 1.3. [Typing and shape of arrays](#)
 - 1.4. [Array operations and examples](#)
 - 1.5. [No explicit iteration](#)
 - 1.6. [Vectors and matrices](#)
 - 1.7. [Mathematical operations](#)
 - 1.8. [Statically typed, rectangular arrays: ndarrays](#)
 - 1.9. [NumPy](#)
 - 1.10. [Randomisation](#)
 - 1.11. [arange](#)
 - 1.12. [linspace](#)
 - 1.13. [Open and save text files](#)
 - 1.14. [Slicing and indexing arrays](#)
 - 1.15. [Transposition](#)
 - 1.16. [Flip and rotate](#)
 - 1.17. [Cut + tape operations](#)
 - 1.18. [Tiling](#)
 - 1.19. [Boolean selection and masking, boolean arrays](#)
 - 1.20. [Fancy indexing](#)
 - 1.21. [Map](#)
 - 1.22. [Broadcasting](#)
 - 1.23. [Broadcasting is just automatic tiling](#)
 - 1.24. [Reduction](#)
 - 1.25. [Accumulation](#)
 - 1.26. [Finding](#)
2. [Week 2: numerical basics part 2](#)
 - 2.1. [Algebraic properties](#)
 - 2.2. [Number types](#)
 - 2.3. [Floats](#)
 - 2.4. [A number in \[1.0, 2.0\) and a shift](#)
 - 2.5. [IEEE 754](#)
 - 2.6. [Binary representation of floats](#)

[\[edit\]](#)[\[edit\]](#)

- 2.7. [Special features of floats](#)
- 2.8. [Special numbers](#)
- 2.9. [Roundoff error](#)
- 2.10. [Machine precision and \$\epsilon\$](#)
- 2.11. [Array layout and structure](#)
- 2.12. [Strides and shape](#)
- 2.13. [Dope fiends](#)
- 2.14. [C and Fortran order](#)
- 2.15. [NumPy types](#)
- 2.16. [Tensor operations - reshaping](#)
- 2.17. [Rank promotion](#)
- 2.18. [Swapping and rearranging axes](#)
- 2.19. [The swap, reshape, swap dance](#)
- 2.20. [Einstein summation notation \(einsum\)](#)
- 2.21. [Vectorisation](#)
- 2.22. [Implementing equations](#)
- 3. [Week 3: scientific visualisation](#)
- 3.1. [Grammar of Graphics](#)
- 3.2. [Figure and caption](#)
- 3.3. [Guides](#)
- 3.4. [Geoms](#)
- 3.5. [Basic 2D plots](#)
- 3.6. [Scatterplot](#)
- 3.7. [Bar chart](#)
- 3.8. [Line plot](#)
- 3.9. [Marking measurements](#)
- 3.10. [Ribbon plots](#)
- 3.11. [Bad plots \(things that may be wrong\)](#)
- 3.12. [Units](#)
- 3.13. [Axes and coordinate transform](#)
- 3.14. [Stats](#)
- 3.15. [Histograms: showing distributions](#)
- 3.16. [Ranking operations](#)
- 3.17. [Regression and smoothing](#)
- 3.18. [More on geoms](#)
- 3.19. [Markers: scalar attributes](#)
- 3.20. [Colour maps](#)
- 3.21. [Unsigned scalar](#)
- 3.22. [Scaling colourmaps](#)
- 3.23. [Lines](#)
- 3.24. [The staircase and the bar](#)
- 3.25. [Alpha and transparency](#)
- 3.26. [Aspect ratio](#)
- 3.27. [Coords in general](#)
- 3.28. [Log scales](#)
- 3.29. [Polynomial or power-law relationships](#)
- 3.30. [Negative numbers](#)
- 3.31. [Polar](#)
- 3.32. [Facets and layers](#)

[\[edit\]](#)

- 3.33. [Layers](#)
- 3.34. [Facets](#)
- 3.35. [Communicating uncertainty](#)
- 3.36. [Error bars](#)
- 4. [Week 4: Computational Linear Algebra I](#)
- 4.1. [Text and translation](#)
- 4.2. [Vector spaces](#)
- 4.3. [Vector operations](#)
- 4.4. [Using vectors](#)
- 4.5. [Vector data](#)
- 4.6. [Geometric operations](#)
- 4.7. [Machine learning applications](#)
- 4.8. [Image compression](#)
- 4.9. [Basic vector operations](#)
- 4.10. [Vector length](#)
- 4.11. [Different norms](#)
- 4.12. [Norm info and unit spheres](#)
- 4.13. [Unit vectors and normalisation](#)
- 4.14. [Inner products of vectors](#)
- 4.15. [Mean vector](#)
- 4.16. [High-dimensional vector spaces](#)
- 4.17. [Matrices and linear operators](#)
- 4.18. [Matrix notation](#)
- 4.19. [Matrices as maps](#)
- 4.20. [Linearity](#)
- 4.21. [Linear algebra - matrix operations](#)
- 4.22. [Application to vectors \(of matrices\)](#)
- 4.23. [The @ operator](#)
- 4.24. [Matrix multiplication](#)
- 4.25. [Multiplication algorithm](#)
- 4.26. [Time complexity of multiplication](#)
- 4.27. [Applying matrices to vectors](#)
- 4.28. [Column and row vectors](#)
- 4.29. [Composed maps](#)
- 4.30. [Commutativity](#)
- 4.31. [Transpose order switching](#)
- 4.32. [Covariance matrices](#)
- 4.33. [Covariance ellipses](#)
- 4.34. [Anatomy of a matrix](#)
- 4.35. [Special matrix forms](#)
- 5. [Week 5: Computational Linear Algebra II](#)
- 5.1. [Graphs as matrices](#)
- 5.2. [Computing graph properties](#)
- 5.3. [Edge weighted graphs](#)
- 5.4. [Stochastic matrices](#)
- 5.5. [Flow analysis: using matrices to model discrete problems](#)
- 5.6. [New matrix operations](#)
- 5.7. [Matrix powers \(exponentiation\)](#)
- 5.8. [Stable point](#)

[\[edit\]](#)[\[edit\]](#)

- 5.9. [Eigenvalues and eigenvectors](#)
- 5.10. [Finding leading eigenvector: the power iteration method](#)
- 5.11. [Computing eigenvectors and eigenvalues with NumPy](#)
- 5.12. [Formal definition of eigenvectors and eigenvalues](#)
- 5.13. [The eigenspectrum](#)
- 5.14. [Numerical instability of eigendecomposition algorithms](#)
- 5.15. [Principle Component Analysis \(PCA\)](#)
- 5.16. [Decomposition of the covariance matrix into its eigenvectors and eigenvalues](#)
- 5.17. [Reconstruction of the covariance matrix from its eigenvectors and eigenvalues](#)
- 5.18. [Approximating a matrix](#)
- 5.19. [Dimensionality reduction](#)
- 5.20. [Matrix properties from the eigendecomposition](#)
- 5.21. [Definite and semi-definite matrices](#)
- 5.22. [Things we can tell from eigenvectors/values](#)
- 5.23. [Matrix inversion](#)
- 5.24. [Computing the inverse of a matrix](#)
- 5.25. [Singular and non-singular matrices](#)
- 5.26. [Numerical stability of matrix inversion algorithms](#)
- 5.27. [Special cases](#)
- 5.28. [Solving problems with inversion](#)
- 5.29. [Singular value decomposition \(SVD\)](#)
- 5.30. [Relation to eigendecomposition](#)
- 5.31. [SVD decomposes any matrix into three matrices with special forms](#)
- 5.32. [Using the SVD](#)
- 5.33. [Inversion using SVD](#)
- 5.34. [Pseudo-inverse](#)
- 5.35. [Rank of a matrix](#)
- 5.36. [Condition number of matrix](#)
- 5.37. [Relation to singularity](#)
- 5.38. [Applying decompositions](#)
- 6. [Week 6: Optimisation I](#)
- 6.1. [What is optimisation?](#)
- 6.2. [Parameters and objective function](#)
- 6.3. [Minimising differences](#)
- 6.4. [Evaluating the objective function](#)
- 6.5. [Discrete vs. continuous](#)
- 6.6. [Focus: continuous optimisation in real vector spaces](#)
- 6.7. [Geometric median: optimisation in \$\mathbb{R}^2\$](#)
- 6.8. [An example of optimisation in \$\mathbb{R}^N\$](#)
- 6.9. [Constrained optimisation](#)
- 6.10. [Common constraint types](#)
- 6.11. [Constraints and penalties](#)
- 6.12. [Soft constraints](#)
- 6.13. [Relaxation of objective functions](#)
- 6.14. [Penalisation](#)
- 6.15. [Penalty functions](#)
- 6.16. [Convexity, global and local minima](#)
- 6.17. [Convex optimisation](#)
- 6.18. [Continuity](#)


[\[edit\]](#)

- 6.19. [Direct convex optimisation: linear least squares](#)
- 6.20. [Line fitting](#)
- 6.21. [Iterative optimisation](#)
- 6.22. [Regular search: grid search](#)
- 6.23. [Revenge of the curse of dimensionality](#)
- 6.24. [Density of grid search](#)
- 6.25. [Hyperparameters](#)
- 6.26. [Simple stochastic: random search](#)
- 6.27. [Metaheuristics](#)
- 6.28. [Locality](#)
- 6.29. [Hill climbing: local search](#)
- 6.30. [Temperature](#)
- 6.31. [Population](#)
- 6.32. [Pros and cons of genetic algorithms: population search](#)
- 6.33. [Memory](#)
- 6.34. [Memory + population: ant colony optimisation](#)
- 6.35. [Quality of optimisation](#)
- 6.36. [Guarantees of convergence](#)
- 6.37. [Tuning optimisation](#)
- 6.38. [What can go wrong?](#)
- 7. [Week 7: Optimisation II](#) [\[edit\]](#)
 - 7.1. [Deep neural networks](#)
 - 7.2. [Backpropagation](#)
 - 7.3. [Why not use heuristic search?](#)
 - 7.4. [Jacobian: matrix of derivatives](#)
 - 7.5. [Gradient vector: one row of the Jacobian](#)
 - 7.6. [Hessian: Jacobian of the gradient vector](#)
 - 7.7. [Differentiable objective functions](#)
 - 7.8. [Orders: zeroth, first, second](#)
 - 7.9. [Optimisation with derivatives](#)
 - 7.10. [Differentiability](#)
 - 7.11. [Lipschitz continuity \(no ankle breaking\)](#)
 - 7.12. [Lipschitz constant](#)
 - 7.13. [Analytical derivatives](#)
 - 7.14. [Computable exact derivatives](#)
 - 7.15. [Gradient: a derivative vector](#)
 - 7.16. [Gradient descent](#)
 - 7.17. [Downhill is not always the shortest route](#)
 - 7.18. [Why step size matters](#)
 - 7.19. [Gradient descent in 2D](#)
 - 7.20. [Gradients of the objective function](#)
 - 7.21. [Why not use numerical differences?](#)
 - 7.22. [Numerical problems](#)
 - 7.23. [Revenge of the curse of dimensionality again](#)
 - 7.24. [Improving gradient descent](#)
 - 7.25. [Automatic differentiation](#)
 - 7.26. [Programming language advances](#)
 - 7.27. [Autograd](#)
 - 7.28. [Limits of automatic differentiation](#)

- 7.29. [Stochastic relaxation and resolution](#)
- 7.30. [Stochastic gradient descent](#)
- 7.31. [Memory advantages of SGD](#)
- 7.32. [Heuristic enhancement](#)
- 7.33. [Using SGD](#)
- 7.34. [Linear regression with SGD](#)
- 7.35. [Random restart](#)
- 7.36. [Simple memory: momentum terms](#)
- 7.37. [Types of critical points](#)
- 7.38. [Second-order derivatives](#)
- 7.39. [Second-order optimisation](#)
- 8. [Week 8: Probability I](#) [\[edit\]](#)
 - 8.1. [What is probability?](#)
 - 8.2. [Bayesian/Laplacian view on probability](#)
 - 8.3. [Frequentist view of probability](#)
 - 8.4. [Objectivity and subjectivity](#)
 - 8.5. [Generative models: forward and inverse probability](#)
 - 8.6. [Axioms of probability](#)
 - 8.7. [Random variables and distributions](#)
 - 8.8. [Distributions](#)
 - 8.9. [Discrete and continuous](#)
 - 8.10. [Integration to unity](#)
 - 8.11. [Expectation](#)
 - 8.12. [Expectation and means](#)
 - 8.13. [Expectations of functions of \$X\$](#)
 - 8.14. [Samples and sampling](#)
 - 8.15. [The empirical distribution](#)
 - 8.16. [Computing the empirical distribution](#)
 - 8.17. [Uniform sampling](#)
 - 8.18. [Discrete sampling](#)
 - 8.19. [Joint, conditional, marginal](#)
 - 8.20. [Bigrams](#)
 - 8.21. [Log probabilities](#)
 - 8.22. [Comparing log-likelihoods \(example\)](#)
 - 8.23. [Bayes' Rule - inverting conditional distributions](#)
 - 8.24. [Nomenclature](#)
 - 8.25. [Integration over the evidence](#)
 - 8.26. [Natural frequency](#)
 - 8.27. [Bayes' rule for combining evidence](#)
 - 8.28. [Entropy](#)
 - 8.29. [Entropy is just the expectation of log-probability](#)
- 9. [Week 9: Probability II](#) [\[edit\]](#)
 - 9.1. [Continuous random variables - problems with continuous variables](#)
 - 9.2. [Probability distribution functions](#)
 - 9.3. [Support](#)
 - 9.4. [Cumulative distribution functions](#)
 - 9.5. [Location and scale](#)
 - 9.6. [Central limit theorem](#)
 - 9.7. [Multivariate distributions: distributions over \$\mathbb{R}^n\$](#)

- 9.8. [Multivariate uniform distribution](#)
- 9.9. [Multivariate normal distribution](#)
- 9.10. [Joint and marginal distributions](#)
- 9.11. [Monte Carlo](#)
- 9.12. [Inference - population and samples, statistics and parameters](#)
- 9.13. [Two worldviews](#)
- 9.14. [Three approaches to inference](#)
- 9.15. [Estimation](#)
- 9.16. [Direct estimation](#)
- 9.17. [Standard estimators](#)
- 9.18. [Maximum likelihood estimation: estimation by optimisation](#)
- 9.19. [Bayesian inference](#)
- 9.20. [Markov Chain Monte Carlo](#)
- 10. [Week 10: digital signals and time series](#)
- 10.1. [Time series and signals](#)
- 10.2. [Noise](#)
- 10.3. [Sampling: amplitude quantisation](#)
- 10.4. [Sampling theory](#)
- 10.5. [Nyquist limit](#)
- 10.6. [Aliasing](#)
- 10.7. [Filtering and smoothing - why filter?](#)
- 10.8. [A simple filter: moving averages](#)
- 10.9. [Sliding window](#)
- 10.10. [Using moving averages](#)
- 10.11. [Nonlinear filtering](#)
- 10.12. [Median filtering](#)
- 10.13. [Generalising moving averages - linear filters](#)
- 10.14. [Convolution](#)
- 10.15. [Algebraic properties and convolution](#)
- 10.16. [Simplest convolutions: Dirac delta functions](#)
- 10.17. [Convolutions with the delta function](#)
- 10.18. [Frequency domain](#)
- 10.19. [Fourier transform - sine wave decomposition](#)
- 10.20. [Correlation between signals](#)
- 10.21. [Transform](#)
- 10.22. [Fourier transform equation](#)
- 10.23. [Discrete Fourier Transform \(DFT\)](#)
- 10.24. [Complex components: phase and magnitude](#)
- 10.25. [Spectra](#)
- 10.26. [Reconstruction of a signal](#)
- 10.27. [Fast Fourier Transform \(FFT\)](#)
- 10.28. [Convolution Theorem](#)
- 10.29. [Frequency domain effects of filtering](#)
- 10.30. [Linear filters in the frequency domain](#)

[\[edit\]](#)[Open DF Moodle in new window](#)[NumPy documentation](#)[Past papers](#)



$$\begin{matrix} \mathbf{M} & = & \mathbf{U} & \mathbf{\Sigma} & \mathbf{V}^* \\ m \times n & & m \times m & m \times n & n \times n \end{matrix}$$

Week 1: numerical basics

[\[edit\]](#)

Uses of arrays and array operations

- Images, sounds, video
- Scientific data, 3D graphics (geometry)
- Abstraction and elegance (no explicit loops)
- Mathematical power (linear algebra)
- Efficiency (numerical arrays are compact, computationally efficient)
- Deep learning

Vectorisation

- Where we write code that acts on arrays of values simultaneously instead of sequentially
- Vectorised computing is a special case of parallel computing restricted only to numerical operations on fixed size arrays
- Modern CPUs have Single Instruction Multiple Data (SIMD) vectorised instructions (e.g. MMX, SSE, SSE2, SSE3 on x86, NEON on ARM, etc.)
- GPUs are array processors - much better than the CPU at vectorised computation
- GPUs are big groups of very simple processors - able to deal with data in numerical arrays very quickly but very slow with other data structures
- Therefore need to write code in terms of numerical arrays to take advantage of GPU

Typing and shape of arrays

- 1D array = vector, 2D array = matrix, 3D/above array = tensor

Array operations and examples

- Slice = slice out rectangular regions
 - e.g. get 2nd to 9th row of matrix with `x[1:9, :]`
 - or set the second column to 0 with `x[:, 1] = 0`
- Filter by criteria with `x[x < 0]`
- Reduce (aggregate across dimensions)
 - e.g. sum **"across"** rows with `np.sum(x, axis=0)`
- Map = apply functions or arithmetic operations element-wise
 - e.g. add 1 to all elements with `x+1`, add x and y with `x+y`, take the sin of every element with `np.sin(x)`
- Concatenate and repeat:
 - stick x and y on top of each other with `np.concatenate([x, y], axis=0)`
 - repeat x 8 times across columns with `np.tile(x, [1, 8])`
- Generate:
 - create arrays of all zeros with `np.full((8, 8), 0)`
 - create arrays containing a counting range with `np.arange(10)`
 - load/save arrays from/to files
- Reorder:
 - reverse/flip/sort axes
 - exchange rows/columns (transpose)

No explicit iteration

- Is simpler (no loops to write)
- Is faster (can be run in accelerated routines or hardware acceleration on GPU)

Vectors and matrices

- 1D arrays can represent vectors, signals
- 2D arrays (matrices) can have linear algebra done on them - they represent linear maps (a type of function operating on vectors in vector space)
- Example: multiplication operation applies the map when multiplying with vectors and composes the map when multiplying with other matrices

Mathematical operations

- Vector operations - apply geometric effects to vectors
 - dot product, cross product, norm
 - getting euclidean length of a vector
- Matrix operations - linear algebra operations like multiplication, transpose, inverse, matrix exponentials, decompositions
- Signal processing relevant operations: convolution (e.g. for blurring an image), Fourier transform, numerical gradients, cumulative summation

Statically typed, rectangular arrays: ndarrays

- ndarrays (n-dimensional arrays) represent sequences
- However, arrays are not like lists:
 - fixed, predefined size ("shape")
 - fixed, predefined type (all elements have the same type)
 - can only hold numbers (typically integer/floating-point)
 - inherently multidimensional
 - must be "rectangular" - same number of columns in each row, not ragged
- The type of an ndarray must be specified very precisely
- Usually, resizing or extending an array requires copying elements into a new array of the correct size first
- ndarrays need typing because they are stored as a block of raw numbers.
- ndarrays are a thin wrapper around raw blocks of memory. As a reason, they are much more efficiently stored and operations can be performed on them much faster.

NumPy

- An array in numpy has a shape e.g. (5, 6) and a type e.g. float64
- Convert a list into a numpy array with `np.array(list)`
- Allocate memory for an array for a given shape (but don't initialise it) with `np.empty(shape)`
- Make new array and initialise all elements to 0 with `np.zeros(shape)`
- Same as above but 1 - `np.ones(shape)`
- Initialise all elements to value with `np.full(shape, value)`
- Can use `_like` to take an existing ndarray and make a new array with the same shape and dtype, filled with a value - e.g. `np.ones_like(array)`

Randomisation

- `np.random.randint(a, b, shape)` - make array with uniform random integers between a and b (exclusive)
- `np.random.uniform(a, b, shape)` - same as above but with floating point numbers instead of integers
- `np.random.normal(mu, sigma, shape)` - make array with normally distributed random floating point numbers between the given mean mu and std. dev sigma

arange

- Create vector of increasing values 0 to end (exclusive) with `np.arange(end)`
- Or start to end (exclusive) with `np.arange(start, end)`
- Or start to end (exclusive) with negative and/or fractional steps with `np.arange(start, end, step)`

linspace

- `np.linspace(start, stop, steps)` makes steps values between start and stop **inclusive**

Open and save text files

- `np.loadtxt(filename)` to open text file and `np.savetxt(array, filename)` to save to text file

Slicing and indexing arrays

- Slicing doesn't change rank of array - selects rectangular subset with same number of dimensions
- Indexing reduces the rank of an array (usually) - selects rectangular subsets where one dimension is a singleton and removes that dimension

Transposition

- Exchange rows and columns (no effect on 1D array, reverses the order of all dimensions in ≥ 2 D arrays)
- `x.T` for transpose of `X`
- Transposing is an $O(1)$ time operation - it changes how the array data is accessed without copying it

Flip and rotate

- `flip` and `rot90` exist for flip/rotate in a single operation using indexing

Cut + tape operations

- Join two arrays with `np.concatenate([x, y])` or stack them up with `np.stack([x, y])`
- If there are multiple dimensions, can specify axis to join on with `concatenate` (e.g. join on rows with `axis=0`)

Tiling

- Repeat arrays with `np.tile(array, tile_shape)`

Boolean selection and masking, boolean arrays

- Comparison mask with array e.g. `x > 5` results in array of same shape with element wise boolean result
- Can also do selection mask e.g. `x == y+2`
- `np.where(bool, a, b)` selects `a` where `bool` is `True` and `b` otherwise - `a` and `b` must be the same shape or broadcastable to the right shape
- `np.nonzero` converts boolean array to array of indices (e.g. `np.nonzero(x > 3)`)

Fancy indexing

- You can index arrays with an array of integer indices directly
- You can index arrays with boolean arrays

Map

- Elementwise operations on arrays
- Single argument `np.tan` or unary negative operations (`-x`)
- Two arguments (`x+y`, `x-1`), `np.maximum(x, y)`
- In a map operation with multiple arrays, their shapes must be compatible
- If the two arrays have the same shape then the operations are applied elementwise, otherwise one of the arrays will be broadcasted so that they have the same shape. Broadcasting involves tiling one of the arrays

until the two arrays have the same shape, if possible (more efficient than explicit broadcasting)

Broadcasting

- Elementwise array arithmetic is for when both sides of an operator have the same shape
- Scalar arithmetic is for when one side is a scalar and the other an array
- Broadcasting is the way in which arithmetic operations are done on arrays when the operands are of different shapes
- If the operands have the same number of dimensions then they must have the same shape - operations are done elementwise
- If one operand is an array with fewer dimensions than the other, then if the last dimensions of the first array match the shape of the second array, operations are well-defined
- e.g. if LHS is size (\dots, j, k, l) and RHS is (l) or (k, l) or (j, k, l) , then this is OK
- To perform operations column-wise, we can transpose then perform the operation then transpose back

Broadcasting is just automatic tiling

- When broadcasting, the array is repeated or tiled as needed to expand to the correct size, then the operation is applied
- e.g. adding a $(2, 3)$ array and a $(3,)$ array means repeating the $(3,)$ array into two identical rows

Reduction

- A reduction/aggregation applies a function to two elements within the array repeatedly
- e.g. $[1, 2, 3, 4]$ reduced with $*$ is $1 * 2 * 3 * 4 = 24$

```
1 2 3 4
5 6 7 8
```

Reduce on columns with "+":

```
1 + 2 + 3 + 4 = 10
5 + 6 + 7 + 8 = 26
```

Reduce on rows with "+":

```
1 2 3 4
+ + + +
5 6 7 8
```

```
=
6 8 10 12
```

Reduce on rows then columns:

```
1 + 2 + 3 + 4
+ + + +
5 + 6 + 7 + 8
```

```
=
6 + 8 + 10 + 12 = 36
```

- `np.any` = reduce with logical OR
- `np.all` = reduce with logical AND
- `np.min` = reduce with $\min(a, b)$
- `np.max` = reduce with $\max(a, b)$
- `np.sum` = reduce with $+$
- `np.prod` = reduce with $*$
- `np.mean(arr)` = `np.sum(arr) / len(arr)`
- `np.std(arr)` = duh

Accumulation

- The cumulative sum / running sum of an array is an array of the same size that stores the result of summing up every element up to that point
- This is similar to reduction but we keep intermediate values instead of collapsing to the final result - this is accumulation
- e.g. $[1, 2, 3, 4] = [1, 1+2, 1+2+3, 1+2+3+4] = [1, 3, 6, 10]$
- `np.cumsum` is the accumulation of $+$

- `np.cumprod` is the accumulation of *
- `np.diff` is the accumulation of - (but notice one less output than input)
- `np.gradient` is like `np.diff` but uses central differences to get same length output - it computes the gradient over every axis and returns them all in a list

Finding

- `np.argmax/argmin` = index of largest/smallest element (tie breaks by picking first occurrence)
- `np.argsort` - indices that would sort the array back into ascending order
- `np.nonzero` (find indices that are non-zero values)

Week 2: numerical basics part 2

[\[edit\]](#)

Algebraic properties

- Associativity, distributivity, commutativity are not preserved with the representation of numbers we used for computations (approximations)

Number types

- Can store integers or floating point numbers in arrays
- Integers represent whole numbers and have several types

name	bytes	min	max
int8	1	-128	127
uint8	1	0	255
int16	2	-32,768	32,767
uint16	2	0	65,535
int32	4	-2,147,483,648	2,147,483,647
uint32	4	0	4,294,967,295
int64	8	-9,223,372,036,854,775,808	+9,223,372,036,854,775,807
uint64	8	0	18,446,744,073,709,551,615

- Operations that exceed the maximum value of a type cause overflow
- Overflows have potentially undefined behaviour
- e.g. adding 8 to 120 int8 (exceeds 127) may result in 127, -128, or some other number
- Most systems will wrap around the result, computing the operation modulo the range of the integer type

Floats

- Floating point representation represents fractional parts of numbers
- Floating point representation directly supported by hardware in most cases
- Can represent a much wider range of values even compared to fixed point representation

A number in [1.0, 2.0) and a shift

- Floating point numbers are just a compact way to represent numbers of very large range, by allowing a fractional number with a standardised range (mantissa between 1.0 and just under 2.0) with a scaling or stretching factor (exponent, varies in steps of powers of 2)
- Powers of 2 - so they can be thought of as numbers between 1.0 and 2.0 that are shifted and stretched by doubling or halving repeatedly
- $\text{value} = \text{sign} * (1.[\text{mantissa}]) * (2^{\text{exponent}})$ (point not dot multiply)
- For a relatively small number of digits, a large range of numbers can be represented, but the precision is

- variable (very precise for numbers close to zero and coarser precision for numbers far from 0)
- The leading 1 in front of the mantissa is implicit - we know the mantissa represents a number between 1.0 and 2.0 (exclusive)
- The mantissa is always a positive number, stored as an integer such that it would be shifted until the first digit was just after the decimal point
- e.g. $00100111010001001000101_2$ is $1.00100111010001001000101_2$
- The float32 format is: 1 bit for the sign, 8 for the exponent, 23 for the mantissa
- The exponents in float32 are stored with an implied offset of -127, so exponent=0 is really exponent = -127

1 10000011 00100111010001001000101

- What do we know?

- The number is negative, because leading bit (sign bit) is 1.
- The mantissa represents $1.00100111010001001000101_2 = 1.153389573097229_{10}$
- The exponent represents $2^{131-127} = 2^4 = 16$ ($1000011_2 = 131_{10}$), because of the implied offset.

IEEE 754

- Dominant standard for floating point numbers - defines representation for floating point numbers and operations defined upon them

Name	Common name	Base	Digits	Decimal digits	Exponent bits	Decimal E max	Exponent bias	E min	E max	Notes
binary16	Half precision	2	11	3.31	5	4.51	$2^4-1 = 15$	-14	+15	not basic
binary32	Single precision	2	24	7.22	8	38.23	$2^7-1 = 127$	-126	+127	
binary64	Double precision	2	53	15.95	11	307.95	$2^{10}-1 = 1023$	-1022	+1023	
binary128	Quadruple precision	2	113	34.02	15	4931.77	$2^{14}-1 = 16383$	-16382	+16383	
binary256	Octuple precision	2	237	71.34	19	78913.2	$2^{18}-1 = 262143$	-262142	+262143	not basic

- Floats are either in single (e.g. float32) or double precision (e.g. float64) - GPUs are usually fastest by far with float32 but can do double precision float64 computations at some significant cost
- float32 is 32 bits or 4 bytes per number
- float64 is 64 bits or 8 bytes per number - most x86 CPUs have specialised float64 hardware (for x86, 80-bit long double representations)

Binary representation of floats

- Here is 1 in float64 0 0111111111 00.00
- sign bit 0 = positive, exponent $0111111111 = 1023$, $2^{1023-1023} = 2^0 = 1$, mantissa 0
- positive ($1.0 * 2^0$) = 1.0
- In float64, every integer from -2^{53} to 2^{53} is precisely representable as the mantissa has 53 bits

Special features of floats

- Floats can have exceptions happen during calculations (at hardware level)
- The OS/language determines the response (e.g. Unix sends SIGFPE to process)
- Exceptions are:
 - Invalid operation (operation without defined real number result, like $0.0 / 0.0$ and $\sqrt{-1.0}$)
 - Non-zero divided by zero
 - Overflow (exceeds limit of floating point number)
 - Underflow (smaller than smallest representable, so round to zero)
 - Inexact - occurs if computation produces inexact result due to rounding
- Exceptions can be trapped (signal to process, and process can halt or take its own action)
- Exceptions can be untrapped (do default operation for error like output `np.inf` for division by zero) without halting

- Normally invalid is trapped, inexact and underflow are not trapped
- Overflow and division by zero may or may not be trapped
- NumPy traps all except inexact (but normally just prints a warning and continues for all - can be configured to halt and raise an exception)

Special numbers

- In IEEE754, +0 and -0 exist - positive 0 has sign bit as 0, negative has it as 1
- +0 and -0 compare equal and work exactly the same in all operations, except the sign bit propagates from the other operand
- In IEEE754, +inf and -inf exist - infinities are explicitly encoded
- All 1s in the exponent and all 0s in the mantissa = infinity
- NaN represents invalid values:
 - $0 / 0$
 - inf / inf (either positive or negative inf)
 - $\text{inf} - \text{inf}$ or $\text{inf} + -\text{inf}$
 - $\text{inf} * 0$ or $0 * -\text{inf}$
 - $\text{sqrt}(x)$, if $x < 0$
 - $\log(x)$, if $x < 0$
 - Any other operation that would have performed any of these calculations internally
- NaN propagates - any floating point operation involving NaN outputs NaN
- Any comparison with NaN evaluates to false
- NaN is not equal to anything including itself, it is not $<$ or $>$ any other number
- NaN is not equivalent to False in Python
- Used both as output for operations that have gone wrong and as placeholder in arrays for missing data
- NaN has all 1s exponent and but all 0s mantissa - this means there is not a unique NaN - float64 has $(2^{52}) - 1$ NaNs

Roundoff error

```
(1.0e30 + 1.0) - 1.0e30 # wrong, severe roundoff error
```

- ```
(1.0e30 - 1.0e30) + 1.0 # no roundoff error
```
- Adding repeated tiny offsets to big values can cause roundoff error
- The ordering of operations important - avoid operations of numbers with wildly different magnitudes
- In some extreme cases distributive and associative rules don't apply to floating point numbers
- $x+y$  will have large error (magnitude error) if  $x$  and  $y$  have different magnitudes
- $x-y$  will have large error if  $x \approx y$  (cancellation error)
- Don't compare floats with  $==$
- Never use equality on floating point values or arrays unless you are specifically testing for roundoff error
- We must always compare floating point numbers by determining if their absolute difference is below some threshold epsilon:  $|x - y| < \epsilon$
- To check whether two values are close enough within some tolerance, use `np.allclose(x, y)`

## Machine precision and $\epsilon$

- Floating point representations have a relative error:

$$\epsilon = \frac{|\text{float}(x) - x|}{|x|}$$

- NumPy guarantees this error is always less than:

$$\epsilon \leq \frac{1}{2} 2^{-t}, \epsilon \leq 2^{-t-1}$$

where  $t$  is number of bits dedicated to the mantissa excluding the implied 1

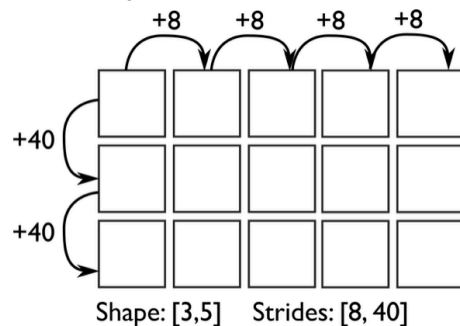
- Guarantee applies to both storage (relative error never  $> \epsilon$ ) and to operations (roundoff error in computations will have relative error  $< \epsilon$ )
- float64 epsilon is about  $2^{-53}$

## Array layout and structure

- A 256x128x4 ndarray of floating point numbers takes up 1MB of memory (ndarray.nbytes)
- You can use np.ravel to show how ndarrays are represented within memory

## Strides and shape

- To implement multidimensional indexing, the standard trick is to use striding
- Striding uses a set of memory offset constants ("strides") to specify how to index into the array, one per axis
- This allows efficient indexing into an array as if it were multidimensional, while still keeping it as a long sequence of numbers packed tightly into memory
- One stride per dimension, and each stride tells the system how many bytes forward to seek to find the next element for each dimension (called a stride because its how big of a step in memory to take to get to the next element)
- For a 1D array, there is one stride - the length of the numeric data type
- For a 2D array, there are two strides:



- One stride may be 8 (one float), the other might be  $8 * x.shape[0]$
- Strides are normally in bytes, not elements, to speed up memory access computations
- To find the array element at index  $[i,j]$  in a 2D matrix, the memory offset from the start of the number block will be:

$$i * stride[0] + j * stride[1]$$

- This generalises to higher dimensions (e.g. 3D, 4D tensors). To iterate through an array, the computations can simply increment by the appropriate stride to move to the next elements; for most operations though, incrementing by the first stride to visit each element in turn is enough
- Using strides, transposition becomes  $O(1)$  (e.g. for a 2D array, we just swap the stride values around)

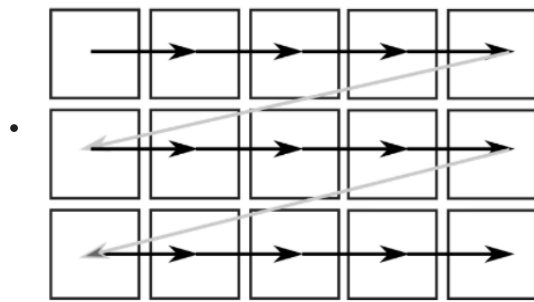
## Dope fiends

- The above type of representation is called a dope vector, which contains the striding information and is held separately from the data itself - a header which specifies how to index
- The alternative is an Illife vector, which uses nested pointers to refer to multidimensional arrays, but are much less efficient for large numerical operations than dope vectors  
e.g. a list of lists in Python `[ [1,2,3], [4,5,6], [7,8,8] ]` has an outer list referring to three inner lists
- Illife vectors can store ragged arrays trivially (no rectangularity requirement)

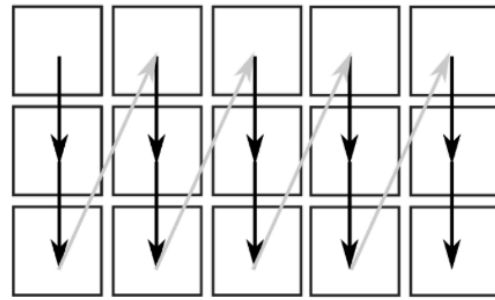
## C and Fortran order

- By default, NumPy uses C ordering - last index changes first, column wise then row-wise (this is the default in C-based languages)
- Fortran ordering (first index changes fastest) is used in some older software

### C order / row major



### Fortran order / column major

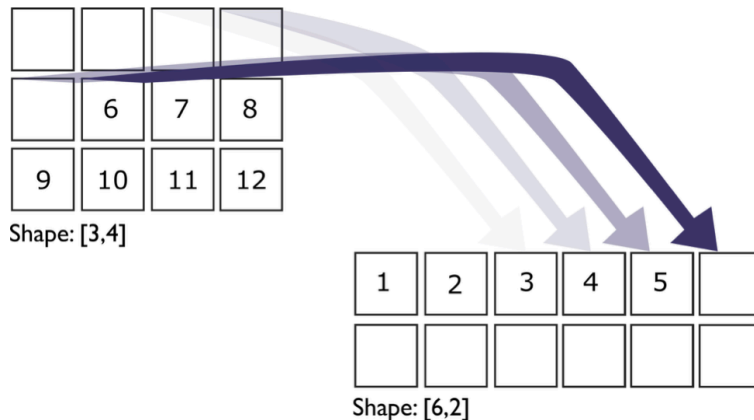


## NumPy types

- Unless specifying dtype= when calling np.array or np.zeros etc, arrays will be float64 if they have any floating point values when created, or int32 otherwise
- An array can be converted with .astype()

## Tensor operations - reshaping

- The number of and order of elements remain unchanged
- Only the positions at which the array wraps into the next dimension change
- The last dimension changes fastest, followed by the second last, etc.
- Reshaping "pours" the elements into a new mould
- ndarray.reshape(shape)
- Pouring fills the last dimension first:



## Rank promotion

- Rank-preserving operations like maps (x+1), slices (x[:2, :2]), and transposing don't change the number of dimensions
- Rank-reducing operations like indexing (x[0]), reduction (np.sum(x)), ravel(), and squeeze() reduce the number of dimensions
- Rank-promoting operations (like broadcasting which implicitly promotes tensors when the shapes can be repeated to match) add new dimensions to an array
- We can add (singleton) dimensions by indexing with None (i.e. x[:, None]), which transforms a 1D vector x into a 2D matrix with 1 column in each row
- Can also add dimensions by indexing with np.newaxis
- We can get rid of singleton dimensions with np.squeeze
- Can avoid listing all the indices with elision (e.g. x[2, .., 5] instead of x[2, :, :, :, 5])
- Multiplying a row vector by a column vector results in a matrix (the outer product)
- A 1D array is neither a row or column vector



## Swapping and rearranging axes

- `np.swapaxes(arr, axis1, axis2)` swaps any pair of axes
- If we have a colour video of shape (frame, width, height, 3), and we want to apply an operation on each column, we can temporarily swap the columns to the end, broadcast, and then swap back
- Axis rearrangement is usually a simple change of the array strides and shape; the array itself is not changed so  $O(1)$

## The swap, reshape, swap dance

- Sometimes we don't want to follow the pouring rule
- So we:
  - rearrange the axes
  - reshape the array
  - (optionally) rearrange the axes again

## Einstein summation notation (einsum)

- Simple way to reorder higher rank tensors using an arrow separated string with a letter for each dimension in current order to new order
- `ijk -> jik`

## Vectorisation

- Don't iterate
- Simple problems can be vectorised:
  - collecting inputs into suitable arrays (normally involves stacking arrays together or filling elements in a array or slicing existing arrays)
  - generating auxiliary arrays like indexing arrays, mesh grids, or blank/zero arrays
  - applying operations to combinations of the above arrays using operators/functions
  - select and collate, collecting the results into an output (typically with slicing, joining, and/or masking operations)

## Implementing equations

- If you see an equation of the form:

$$\sum_{i=0}^{n-1} f(x_i),$$

apply `f` to the array and sum the result

Example:

$$\sum_{i=0}^{n-1} \tan(x_i^2)$$

```
x = np.random.uniform(0, 1, 10) # 10 random numbers
np.sum(np.tan(x ** 2))
```

- If the index of the sum appears, use `np.arange` to generate it:

Example:

$$\sum_{i=0}^{n-1} x_i^{(i \bmod 2)} + i$$

```
ix = np.arange(len(x)) # generate index
np.sum(x ** (ix % 2) + ix)
```

- Nested sums

$$\begin{aligned} & \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} I_{xy} J_{yx} \frac{1}{n^2} \\ &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} I_{xy} J_{xy}^T \frac{1}{n^2} \\ &= \frac{1}{n^2} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} I_{xy} J_{xy}^T \end{aligned}$$

```
I = np.random.uniform(0, 1, (3, 3))
J = np.random.uniform(0, 1, (3, 3))
np.sum(I * J.T) / (I.shape[0] ** 2)
```

## Week 3: scientific visualisation

[\[edit\]](#)

### Grammar of Graphics

- Stat: a statistic computed from data, mapped onto visual features with the intent of compact data summarisation - mean and std. dev are stats, the binning of values in a histogram is a stat
- Mapping - transformation of data attributes to visual values - mapping selected attributes (stats or raw dataset values) to visual values using a scale to give units to the attribute
- Scale - specifies the transformation of the units in the dataset/stat to visual units, typically specifies the range of values to be mapped
- Guide - a visual reference indicating the meaning of a mapping, including the scale and the attribute being mapped - includes axis tick marks, labels, colour scales, legends
- Geom - the geometric representation of data after it has been mapped - includes points (which may have colour/size/shape), lines (colour, dash style, thickness), and patches/polygons (can have many attributes)
- Coord - a coordinate system that connects mapped data onto points on the plane (or in general to higher-dimensional coordinates like 3D positions) - spatial configuration of geoms and guides depends on coordinate system
- Layer - one set of geoms, with one mapping on one coordinate system - multiple layers may be overlaid onto a single coordinate system
- Facet - a different view on the same dataset, on a different coordinate system
- Figure - set of one or more facets
- Caption - explains visualisation to reader

### Figure and caption

- A figure is one or more facets that form a coherent visualisation of data - many graphs are single graphs (single facets), but some may include multiple facets
- Every figure needs a clear caption

### Guides

- Good plots have a well defined structure
- Every plot should have proper use of guides to explain mapping and coordinate system
- Axes labelled with any applicable units shown (these are guides for the coordinate system)
- Axes must have ticks indicating subdivisions of axes in labelled units for that axis
- Legends explaining what markers and lines mean (if more than one present) - these are guides which identify different layers in the plot
- Titles explaining what the plot is are an important guide
- A plot may have a grid to help the reader line up data with the axes (a guide for the coordinate system again)
- A plot may have annotations to point out relevant features

### Geoms

- To display data, plots have geoms (geometrical objects representing some element of the data to be plotted) - these include:
  - Lines/curves representing continuous functions (colours, thicknesses, styles)
  - Markers representing disconnecting points, which have sizes, colours, and styles
  - Patches representing shapes with area (like bars in a bar chart), which can come in many forms

### Basic 2D plots

- Most useful plots only involve a small number of relations between variables
- Often there are only two variables with a relation (one independent, one dependent)
- Considering  $y=f(x)$ , where  $x$  and  $y$  are scalar variables, the purpose of the plot is to visually describe the function  $f$  from the input (pairs of 1D vectors  $x, y$ )

- Independent variable (cause) on the x-axis, dependent (effect) on y-axis

## Scatterplot

- Marks (x, y) locations of measurements with markers (point geoms)

## Bar chart

- Draws bars proportional to y at position given by x (bars are patch geoms)

## Line plot

- Draws connected line segments between (x, y) positions, in the order that they are provided (these are line geoms)

## Marking measurements

- Common to show exact points of measurements, not just the lines between the points
- This is a plot with two layers that share the same coordinates (line geoms and point geoms)

## Ribbon plots

- Can plot triplets of vectors (x, y2, y1) - if both ys match an x, then the area between the two can be drawn with polygon geoms
- This gives a ribbon plot with varying thickness

## Bad plots (things that may be wrong)

- No guides (one or more of axes, ticks, labels, units, title, legend) are missing
- Bad coords (plots dependent on x and independent on y)
- Bad mappings (no distinct colours)
- Bad geoms (use of lines for data that is not a continuous curve)

## Units

- If an axis represents real world units (dimensional quantities), this units should always be specified in the axis labels
- Use units of an appropriate scale (e.g. don't use microseconds for monthly data)
- If the quantities are truly dimensionless (bare numbers) then the axes can omit visible units (but should be clearly labelled, e.g. with "relative growth" or "Mach number")
- Never use the index of an array as the x-axis unless it is the only reasonable choice (e.g. use real time elapsed over sample number)
- Avoid rescaled units (if possible rescale manually and insert scaling in the axis label instead of letting matplotlib/other libraries do it)

## Axes and coordinate transform

- An axis specifies the scaling and offset of data by a coordinate system or coord
- The mapping of determined by axis limits (min, max values to be displayed)

## Stats

- A stat is a statistic of a dataset (i.e. a function of the data)
- A stat summarises data in some way
- Examples of stats:
  - aggregate summary statistics (mean, median, std. dev, max/min, IQR)
  - binning operations (categorise data into a number of discrete bins and count the data points falling into)

those bins)

- smoothing and regression (find approximating functions to datasets, like linear regression which fits a line through data)

## Histograms: showing distributions

- Combines a binning operation with a standard 2D bar chart (bar geoms)

## Ranking operations

- Sorted bar charts are an alternative view of a 1D vector (which is a plot of a value against its rank within the array). The value-to-rank operation is the stat which is applied

## Regression and smoothing

- Regression involves finding an approximating function to some data - usually a simpler function - most familiar is linear regression ( $f(x) = mx + c$ ),  $f(x) \approx y$
- A good smoothing of data reveals the attributes of interest to the reader but obscures irrelevant details

## More on geoms

- Markers are geoms that represent bare points in a coordinate system - typically a visual record of a discrete observation
- Markers are useful for identifying different layers in a plot (e.g. show squares for points of a given layer, circles for another)
- Colour choice should be considered, especially for black and white printouts (colour differences will become shade only) + colourblind accommodations

## Markers: scalar attributes

- Markers can display a third or even fourth scalar attribute instead of identifying layers in a plot
- This is done by modulating the scaling or colouring of each marker (e.g. colour for earthquake depth, scaling for magnitude, x/y pos)

## Colour maps

- Colouring of markers is done via a colour map (maps scalar values to colours, usually specified in RGB)
- Colour maps should always be presented with a colour bar which shows the mapping of values to colours

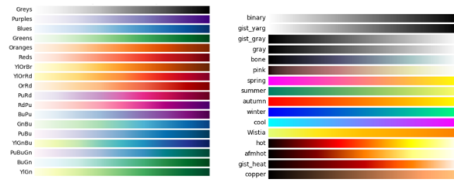
## Unsigned scalar

- If the data is a positive scalar, use a colour map with monotonically varying brightness - as the attribute increases, the colour map should get consistently lighter or darker
- viridis and magma are good because they are perceptually uniform (change in value corresponds to perceptually uniform change in intensity across the whole scale)
- Can also use grayscale or monochrome maps, but colours with brightness+hue are often easier to interpret

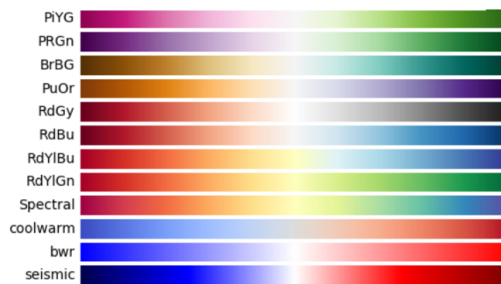
### For unsigned values



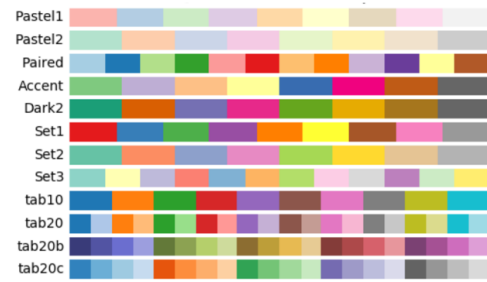
### Usable for unsigned values (but generally inferior)



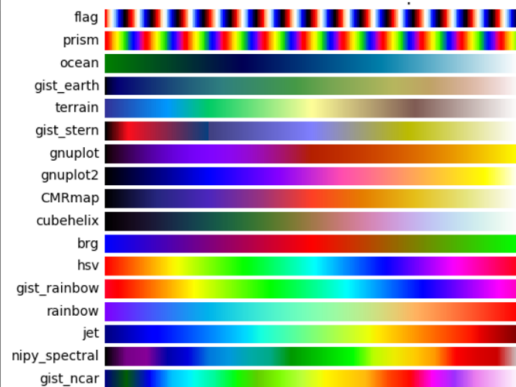
### For signed values



### For discrete values



### For specialist uses only (e.g. terrain rendering in maps)



- Monotonic brightness means increase in data value always leads to increase in visual brightness
- Perceptual linearity (even "notches") are good

## Scaling colourmaps

- Scale data to colourscales appropriately, and always provide the colour bar
- It must be possible to invert the visual units to recover the data units, and the colour bar is essential for this purpose

## Lines

- Geoms that connect points together
- Line geoms can have variable thickness/colour
- Dash patterns are good to distinguish lines without relying on colour (e.g. if printed form does not have colour available or readers are colour blind)

## The staircase and the bar

- Sometimes data is not continuous (e.g. you can't be halfway through a coin toss), so you can use a staircase/step plot instead which doesn't interpolate
- If the measurements of x are naturally discrete (e.g. days of the week, experiment conditions) then bar charts may be suitable

## Alpha and transparency

- Can use different alpha/opacity values for geoms - useful when they overlap to show ones layered behind or

for emphasis/de-emphasis

- Can be hard to interpret if misused

## Aspect ratio

- Never squash/stretch images when displayed
- Aspect ratio of the coord defined by figure size and portion of the figure dedicated to a subplot by default - can be manually adjusted with `set_aspect(ratio_num)` in matplotlib to force the visual units to span equal data units

## Coords in general

- A coordinate system encompasses projection onto 2D plane, might include transformations like polar/logarithmic coordinates, 3D perspective projection

## Log scales

- Can be log in both x and y-axis, or only one (semilog y or semilog x), "log-log" for all

## Polynomial or power-law relationships

- Log-log scales are useful when there is a (suspected) polynomial relationship between variables - the relationship will appear as a straight line on a log-log plot

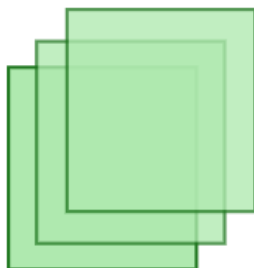
## Negative numbers

- Downside of log scales - log of negative numbers undefined, diverges to -inf at 0
- This means signed data cannot be easily plotted on a log scale (at least if the sign is meaningful and cannot be just shifted to a positive number by adding on a constant)
- There are modified log scales that "cut out" a small region around 0, and then plot `log(abs(x)) * sign(x)` -- the symmetric logarithm or `symlog`. The "cut out" region is plotted as linear in that range. This distorts the plot slightly, but is usually acceptable.

## Polar

- Maps two values onto an angle theta and a radius r
- Most widely used for data that originated from an angular measurement
- Can be used when it makes sense for one of the axes to "wrap around" smoothly
- No natural way to represent negative numbers (radii below zero) on a polar plot
- Generally, reserve polar plots for mappings where data mapping onto r is positive

## Facets and layers



LAYERED



FACETED

- Multiple geoms in a single visualisation can be rendered as:
  - distinct layers superimposed on same set of coords
  - distinct facets on separate sets of coords (with separate scales and guides)

## Layers

- Appropriate when two or more views on a dataset are closely related and the data mapping is in the same units
- Legend essential guide to distinguish differing layers
- When using multiple layers, legend should almost always be used to indicate which geom relates to which dataset attribute
- Occasionally visualisations have double y axes - same x mapping but 2 different y mappings (2 slightly different coords layered over each other), avoid this as difficult to interpret and can mislead reader

## Facets

- Often much better approach than layers
- No need for them to share scaling/range of data between each other
- If two facets show the same attribute, they should if possible have the same scaling applied to make comparisons easy

## Communicating uncertainty

- Data often has observation error (e.g. thermometer reading is not real air temperature)
- Must represent uncertainty on plots for honesty
- Error bars, box plots, dot plots show uncertainty well

## Error bars

- Can use standard deviation, standard error, confidence intervals, nonparametric intervals like IQR

## Week 4: Computational Linear Algebra I

[\[edit\]](#)

### Text and translation

- Text as represented by strings in memory has weak structure
- Comparison functions for strings (e.g. edit distance, hamming distance) captures only character-level semantics
- String operations are character-level operations (concatenation, reversal)
- Instead of lookup for translation, need vector space solution (e.g. word2vec)
- Uses word embeddings (maps semantics onto spatial relations and imbues words with geometric structure), rather than working at level of characters/strings

### Vector spaces

- Vectors are ordered tuples of real numbers  $[x_1 \ x_2 \ x_n]$ ,  $x_i \in \mathbb{R}$
- Number sets denoted as follows:

$\mathbb{R}$ - the set of real numbers;

$\mathbb{R}_{\geq 0}$ - the set of non-negative real numbers;

$\mathbb{R}^n$ - the set of tuples of real numbers, of length  $n$ ;

$\mathbb{R}^{n \times m}$ - the set of 2D arrays (i.e. matrices) with  $n$  rows and  $m$  columns.

$(\mathbb{R}^n, \mathbb{R}^n) \rightarrow \mathbb{R}$ - an operation that maps two vectors in  $\mathbb{R}^n$  to  $\mathbb{R}$ .

## Vector operations

- Any vector of given dimension  $n$  lies in  $\mathbb{R}^n$  (an  $n$ -dimensional real vector space), the set of possible vectors of length  $n$  containing real elements
- Scalar multiplication defines  $a \cdot x$  for any scalar  $a$ , elementwise scaling:  $[a \cdot x_1, a \cdot x_2, \dots, a \cdot x_n]$
- Vector addition defines  $x + y$  for two vectors  $x, y$  of equal dimension - elementwise sum
- We will consider vector spaces equipped with two extra operations:
  - a norm  $\|x\|$  which allows the length of vectors to be measured
  - an inner product  $x \cdot y$  which allows the angles of two vectors to be compared - the inner product of two orthogonal vectors is 0.
- For real vectors,  $x \cdot y = x_1 y_1 + x_2 y_2 + x_3 y_3 + \dots + x_d y_d$
- When a vector space has a norm it is a topological vector space - the space is continuous and we can talk about vectors being "close together" or having a "neighbourhood" around them
- When a vector space has an inner product defined, it is an inner product space - it makes sense to talk about angles between vectors

## Using vectors

- Vectors are used a lot in data science because they can be:
  - composed (via addition)
  - compared (via norms/inner products)
  - weighted (by scaling)

## Vector data

- Datasets are often stored as 2D tables - can be seen as lists of vectors
- Each row is a vector representing an "observation"
- Each column represents one element of the vector across many observations
- Each row can be seen as a vector in  $\mathbb{R}^n$
- The whole matrix is a sequence of vectors in the same vector space
- So we can make geometric statements about tabular data

## Geometric operations

- Most obvious use of vectors is to represent 2D or 3D geometric data
- Standard 3D vector transformations include:
  - scaling, rotation, flipping (mirroring), and translation (shifting)
  - as well as colour space transforms or estimating the surface normals of a triangle mesh (which way the triangles are pointing)
- Graphical pipelines process everything (spatial position, surface normal direction, texture coordinates, colours) as large arrays of vectors
- Graphics programming = pack data into low (2, 3, 4D) dimensional vector arrays on the CPU, process them quickly on the GPU with a shader language

## Machine learning applications

- ML relies heavily on vector representation
- Typical ML process involves transforming some data onto feature vectors, creating a function that transforms feature vectors to a prediction (e.g. a class label)
- The feature vectors are simply an encoding of the data in vector space
- Most ML algorithms can be seen as doing geometric operations (comparing distances, warping space, computing angles, etc.)
- One of the simplest effective ML algorithms is  $k$  nearest neighbours - involves a training set of data consisting of  $x_i$  (feature vector),  $y_i$  (label) pairs
- When a new feature needs to be classified to make predictions, the  $k$  nearest vectors in the training set



under a norm are computed - the output prediction is the class label that occurs the most times among these  $k$  neighbours ( $k$  is preset in some way and is often around 3-12)

- We expect nearby vectors to share common properties - so to find a property we don't know for a given vector, we look at the properties of nearby vectors

## Image compression

- Images are represented as 2D arrays of brightness normally
- But operations we can do are limited if working on just a single pixel (little meaning just as a single letter has little meaning)
- Groups of pixels (e.g. 8x8 square) can be unravelled into 64 dimensional vectors - these can be treated as elements of a vector space
- Many image compression algorithms take advantage of this
- A common approach is to split images into patches, treating them as vectors  $x_1, \dots, x_n$
- The vectors are clustered to find a small number of vectors  $y_1, \dots, y_m$ ,  $m \ll n$ , that are a reasonable approximation of nearby vectors
- Instead of storing the whole image, the vectors for the small number of representative vectors  $y_i$  are stored (the codebook)
- The rest of the image is represented as the indices of the closest matching vector in the codebook (i.e. the vector  $y_j$  that minimises  $\|x_i - y_j\|$ )
- This is vector quantisation because it quantises the vector space into a small number of discrete regions (in this case, we map visual similarity to spatial relationships)

## Basic vector operations

- Addition and multiplication = elementwise weighted sums for vectors of the same dimension, in the same vector space
- Linear interpolation (lerping) between two vectors  $x, y$  can give us a smooth progression by giving points on the line between  $x, y$ , governed by a parameter  $\alpha$ :  
 $\text{lerp}(x, y, \alpha) = (1 - \alpha)x + (\alpha)y$

## Vector length

- The Euclidean length of a vector  $x$  (written as  $\|x\|$ ) can be computed directly with `np.linalg.norm`
- L2 norm is default of `np.linalg.norm`, equivalent to:

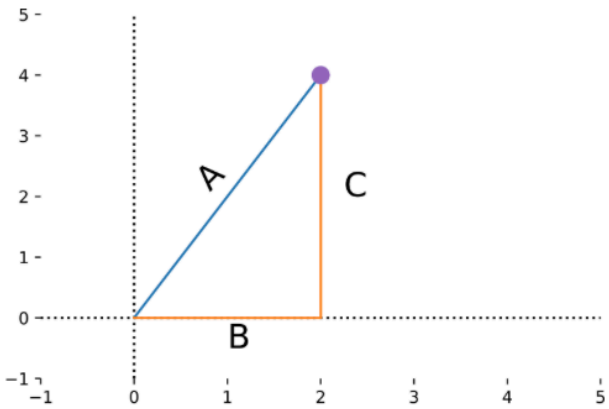
$$\|x\|_2 = \sqrt{x_0^2 + x_1^2 + x_2^2 + \dots + x_n^2}$$

## Different norms

- Default norm = Euclidean norm (L2 norm)
- Vector space of real vectors with Euclidean norm is called Euclidean space
- $\|x - y\|_2$  is the distance between  $x$  and  $y$
- Other norms (ways of measuring lengths of a vector) can be more appropriate in some contexts, like Lp/Minkowski norm:

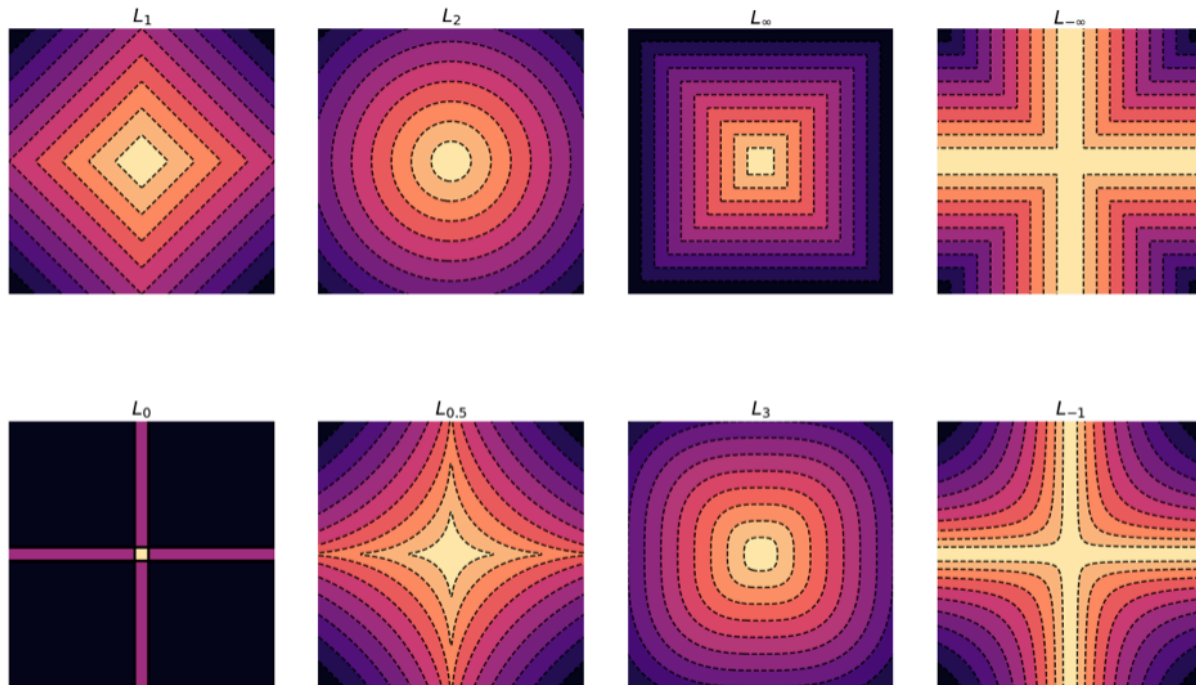
$$\|x\|_p = \sqrt[p]{x_0^p + x_1^p + \dots + x_n^p} = \sqrt[p]{\sum_{i=1}^n x_i^p}$$

## Norm info and unit spheres



$L_2 = |A|$   
 $L_1 = |B| + |C|$   
 $L_\infty = \max(|B|, |C|) = |C|$

|           | p | Notation              | Common name                    | Effect                   | Uses                                          | Geometric view                          |
|-----------|---|-----------------------|--------------------------------|--------------------------|-----------------------------------------------|-----------------------------------------|
|           | 2 | $\ x\ $ or $\ x\ _2$  | Euclidean norm                 | Ordinary distance        | Spatial distance measurement                  | Sphere just touching point              |
|           | 1 | $\ x\ _1$             | Taxicab norm; Manhattan norm   | Sum of absolute values   | Distances in high dimensions, or on grids     | Axis-aligned steps to get to point      |
|           | 0 | $\ x\ _0$             | Zero pseudo-norm; non-zero sum | Count of non-zero values | Counting the number of "active elements"      | Numbers of dimensions not touching axes |
| $\infty$  |   | $\ x\ _{\text{inf}}$  | Infinity norm; max norm        | Maximum element          | Capturing maximum "activation" or "excursion" | Smallest cube enclosing point           |
| $-\infty$ |   | $\ x\ _{-\text{inf}}$ | Min norm                       | Minimum element          | Capturing minimum excursion                   | Distance of point to closest axis       |



## Unit vectors and normalisation

- A unit vector has norm 1 (the definition of a unit vector depends on the norm used)
- Normalising by the Euclidean norm can be done by scaling the vector  $\mathbf{x}$  by  $1/\|\mathbf{x}\|_2$
- A unit vector nearly always refers to a vector with Euclidean norm 1
- A unit vector is "pure direction", always lying on the surface of the norm's unit sphere

## Inner products of vectors

- An inner product  $(\mathbb{R}^N \times \mathbb{R}^N) \rightarrow \mathbb{R}$  measures the angle between two real vectors
- The inner product on real vector spaces is the dot product
- It is related to the cosine distance:

$$\cos(\theta) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|}$$

- The dot product of two vectors in  $\mathbb{R}^n$  is given by:

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=0}^n x_i y_i$$

the sum of the elementwise products

- We can use the inner product to compare vectors of different magnitudes (it only depends on their directions)
- Inner product is only defined for vectors of the same dimension, and only in inner product spaces

## Mean vector

- The mean vector of a collection of  $N$  vectors is defined as the sum of the vectors multiplied by  $1/N$  (or the sum of the vectors divided by  $N$ ):

$$\text{mean}(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n) = \frac{1}{N} \sum_i \mathbf{x}_i$$

- The mean vector is the geometric centroid of a set of vectors ("centre of mass" of the vectors)
- If we have vectors stacked up in a matrix  $\mathbf{X}$ , one vector per row, `np.mean(x, axis=0)` will calculate the mean vector
- We can centre a dataset stored as an array of vectors to zero mean by just subtracting the mean vector from

each row

- No simple direct algorithm to compute the geometric median (the operation cannot be decomposed into scalar addition and scalar multiplication)

## High-dimensional vector spaces

- Most work with vectors is in high-dimension (>3), e.g. 512x512 image lives in  $\mathbb{R}^{262144}$
- Geometric properties of high-D spaces are very counter-intuitive
- The volume of space increases exponentially with d
- There is a lot of empty space in high dimensions, and where data is sparse it can be difficult to generalise in high-dimensional spaces
- Some research areas (e.g. genetic analysis) often have  $n \ll d$  (many fewer samples than measurement features, like 20k vectors with 1M dimensions each)
- Curse of dimensionality - many algorithms that work really well in low dimensions break down in higher ones, as dimension increases generalisation gets exponentially harder
- High-D histograms don't work because of the curse of dimensionality - e.g. if we had 10 different measurements and we wanted 20 bins each, we'd need  $20^{10}$  bins or over 10 trillion bins (over 10TB of memory already)

## Matrices and linear operators

- Matrices are 2D arrays of reals;  $\mathbb{R}^{m \times n}$
- Vectors represent "points in space"
- Matrices represent operations that do things to those points in space
- The operations represented by matrices are a particular class of functions of vectors ("rigid" transformations)
- Matrices are a compact way of writing down these operations (encoding of a function)
- Matrices can be added/subtracted, scaled, transposed, applied, and multiplied together (composed)
  - They can be added and subtracted  $C = A + B$ 
    - $(\mathbb{R}^{n \times m}, \mathbb{R}^{n \times m}) \rightarrow \mathbb{R}^{n \times m}$
  - They can be scaled with a scalar  $C = sA$ 
    - $(\mathbb{R}^{n \times m}, \mathbb{R}) \rightarrow \mathbb{R}^{n \times m}$
  - They can be transposed  $B = A^T$ ; this exchanges rows and columns
    - $\mathbb{R}^{n \times m} \rightarrow \mathbb{R}^{m \times n}$
  - They can be applied to vectors  $y = Ax$ ; this **applies** a matrix to a vector.
    - $(\mathbb{R}^{n \times m}, \mathbb{R}^m) \rightarrow \mathbb{R}^n$
  - They can be multiplied together  $C = AB$ ; this **composes** the effect of two matrices
    - $(\mathbb{R}^{p \times q}, \mathbb{R}^{q \times r}) \rightarrow \mathbb{R}^{p \times r}$

## Matrix notation

$$A \in \mathbb{R}^{n \times m} = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & \dots & a_{2,m} \\ \dots & \dots & \dots & \dots \\ a_{n,1} & a_{n,2} & \dots & a_{n,m} \end{bmatrix}, a_{i,j} \in \mathbb{R}$$

## Matrices as maps

- Matrices represent linear maps - functions applied to vectors which outputs vectors
- Matrices are applied to vectors by multiplying them:

$$Ax = f(x)$$

this is equivalent to applying some function  $f(x)$  to the vectors

- The property of linearity means that for a matrix A taking m dimensional vectors to n dimensional vectors ( $\mathbb{R}^m \rightarrow \mathbb{R}^n$ ):
  - all straight lines remain straight
  - all parallel lines remain parallel
  - the origin does not move

## Linearity

- $$\begin{aligned} f(\mathbf{x} + \mathbf{y}) &= f(\mathbf{x}) + f(\mathbf{y}) &= A(\mathbf{x} + \mathbf{y}) &= A\mathbf{x} + A\mathbf{y}, \\ f(c\mathbf{x}) &= cf(\mathbf{x}) &= A(c\mathbf{x}) &= cA\mathbf{x}, \end{aligned}$$
- A linear map is a function  $f: \mathbb{R}^m \rightarrow \mathbb{R}^n$  that satisfies the linearity conditions
- An  $n \times n$  matrix maps from the vector space to itself (a linear transform)
- If a map satisfies  $A\mathbf{x} = \mathbf{x}$  (i.e.  $f(f(\mathbf{x})) = f(\mathbf{x})$ ), then it is a linear projection (e.g. it projects 3D points into a 2D plane)

## Linear algebra - matrix operations

- Addition:

$$A + B = \begin{bmatrix} a_{1,1} + b_{1,1} & a_{1,2} + b_{1,2} & \dots & a_{1,m} + b_{1,m} \\ a_{2,1} + b_{2,1} & a_{2,2} + b_{2,2} & \dots & a_{2,m} + b_{2,m} \\ \dots & \dots & \dots & \dots \\ a_{n,1} + b_{n,1} & a_{n,2} + b_{n,2} & \dots & a_{n,m} + b_{n,m} \end{bmatrix}$$

- Scalar multiplication:

$$cA = \begin{bmatrix} ca_{1,1} & ca_{1,2} & \dots & ca_{1,m} \\ ca_{2,1} & ca_{2,2} & \dots & ca_{2,m} \\ \dots & \dots & \dots & \dots \\ ca_{n,1} & ca_{n,2} & \dots & ca_{n,m} \end{bmatrix}$$

## Application to vectors (of matrices)

- We can apply a matrix to a vector - we write it as a product  $A\mathbf{x}$
- This is equivalent to applying the function  $f(\mathbf{x})$
- If  $A$  is  $\mathbb{R}^{n \times m}$  and  $\mathbf{x}$  is  $\mathbb{R}^m$  then this will map from an  $m$  dimensional vector to an  $n$  dimensional vector space
- Applying a matrix to a vector forms a weighted sum of the elements of the vector
- In particular - take each element of the vector  $\mathbf{x}$ ,  $x_1, x_2, \dots, x_m$ , and multiply it with the corresponding column of  $A$ , and sum these together

## The @ operator

- We can use @ to form products of vectors and matrices in NumPy

## Matrix multiplication

- If  $A$  represents linear transform  $f(\mathbf{x})$  and  $B$  represents linear transform  $g(\mathbf{x})$ , then  $BA\mathbf{x} = g(f(\mathbf{x}))$
- Multiplying two matrices is equivalent to composing the linear functions they represent - and it results in a matrix which has that effect
- Composition of linear maps is read right to left - to apply transformation  $A$ , then  $B$ , we form the product  $BA$

## Multiplication algorithm

- Matrix multiplication is only defined for two matrices  $A, B$  if:  
 $A$  is  $p \times q$ ,  $B$  is  $q \times r$
- $A$  represents a map  $\mathbb{R}^q \rightarrow \mathbb{R}^p$  and  $B$  represents a map  $\mathbb{R}^r \rightarrow \mathbb{R}^q$
- The output of  $A$  must match the dimension of the input of  $B$ , or the operation is undefined

If  $C = AB$  then

$$C_{ij} = \sum_k a_{ik} b_{kj}$$

- The element at  $C_{ij}$  is the sum of the elementwise product of the  $i$ th row and the  $j$ th column, which will be the same size by the requirement above
- Matrix multiplication is applied by `np.dot(a, b)` or by `a @ b`

- Always use @ for matrix multiplication

## Time complexity of multiplication

- $O(pqr)$ , or  $O(n^3)$  if multiplying two square matrices
- Complexity can be reduced for special forms (e.g. diagonal, triangular, sparse, banded) matrices
- All known general multiplication algorithms are  $> O(N^2)$  but  $< O(N^3)$

## Applying matrices to vectors

- The same algorithm for multiplying two matrices applies to multiplying a matrix by a vector if we assume a  $m$  dimensional vector  $x \in \mathbb{R}^m$  is represented as a  $m \times 1$  column vector
- Then the product  $Ax$  is application of the linear map defined by  $A$  to vector  $x$
- $A$  must be of dimension  $n \times m$  for this operation to be defined
- If  $A$  is  $m \times m$  then it is a linear transform, and the result is just another vector of the same dimension

## Column and row vectors

- The transpose of a column vector is a row vector
- The product of a  $M \times 1$  with a  $1 \times N$  vector is an  $M \times N$  matrix
- This is the outer product of two vectors  $\mathbf{x} \otimes \mathbf{y} = \mathbf{x} \mathbf{y}^T$
- The product of a  $1 \times N$  with an  $N \times 1$  vector is a  $1 \times 1$  matrix (scalar)
- This is the inner product of two vectors  $\mathbf{x} \cdot \mathbf{y} = \mathbf{x} \mathbf{y}^T$

## Composed maps

- If  $A$  represents  $f(x)$  and  $B$  represents  $g(x)$  then the product  $BA$  represents  $g(f(x))$
- Multiplication is composition
- $BAx = B(Ax)$  means do  $A$  to  $x$ , then do  $B$  to the result

## Commutativity

- Matrix multiplication does not commute  $AB \neq BA$
- in  $p \times q$ ,  $q \times r$ , unless  $p = q = r$  then the multiplication is not even defined if the operands are switched since it would be multiplying  $q \times r$  by  $p \times q$

## Transpose order switching

- $(AB)^T = B^T A^T$
- $(A + B)^T = A^T + B^T$

## Covariance matrices

- $$\sigma^2 = \frac{1}{N-1} \sum_{i=0}^{N-1} (x_i - \mu_i)^2$$
- Variance (dataset spread) can be generalised to the multidimensional case
- In the 1D case this is the sum of the squared differences of each element from the mean of the vector
- The standard deviation  $\sigma$  is the square root of the variance and is more often used because it is in the same units as the elements of  $x$
- In the multidimensional case, to get the spread of an  $N \times d$  data matrix  $X$  ( $N$   $d$ -dimensional vectors), we need to compute the covariance of every dimension with every other dimension
- $$\Sigma_{ij} = \frac{1}{N-1} \sum_{k=1}^N (X_{ki} - \mu_i)(X_{kj} - \mu_j)$$

- This is the average squared difference of each column of data from the average of every column - this forms a 2D array  $\Sigma$ , which has entries in element  $i, j$
- It is a special form of matrix - square, symmetric, and positive semi-definite
- Directly provided by NumPy with `np.cov(x)`

## Covariance ellipses

- The covariance matrix captures the spread of data, including any correlations between dimensions
- Can be seen as capturing an ellipse that represents a dataset
- The mean vector represents the centre of the ellipse
- The covariance matrix represents the shape of the ellipse
- This ellipse is often called the error ellipse
- The covariance matrix represents an (inverse) transform of a unit sphere to an ellipse covering the data
- The mean vector and covariance matrix capture the idea of "centre" and "spread" of a collection of points in a vector space

## Anatomy of a matrix

- Matrix entries can be either diagonal or off-diagonal
- Matrices which are all zero outside the main diagonal are diagonal matrices

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

- The anti-diagonal (set of elements  $A_{i[N-i]}$  for an  $N \times N$  matrix:

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

## Special matrix forms

- Identity matrix = all values are zero except 1s along the main diagonal
- Identity matrix has no effect when multiplied by another matrix or vector
- Generated by `np.eye(n)`
- Any scalar multiple of the identity corresponds to a function which uniformly scaled vectors  $(cI)x = cx$
- Zero matrix is all zeros
- Square matrix is  $n \times n$
- Only square matrices have an inverse, have determinants, have eigendecomposition
- A square matrix is a triangular matrix if it has non-zero elements only above (upper triangular) or only below (lower triangular) the diagonal, inclusive of the diagonal

Upper triangular

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 5 & 6 & 7 \\ 0 & 0 & 8 & 9 \\ 0 & 0 & 0 & 10 \end{bmatrix}$$

Lower triangular

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 9 & 10 \end{bmatrix}$$

## Week 5: Computational Linear Algebra II

[\[edit\]](#)

### Graphs as matrices

- Digraphs can be represented by adjacency matrix (each vertex has a row and a column)

### Computing graph properties

- From adjacency matrix - can easily compute:  
out-degree, in-degree, whether matrix is symmetric (matrix transpose will be equal to matrix)
- Directed graph can be converted to undirected graph by computing  $A' = A + A^T$
- This is equivalent to making all the arrows bi-directional
- If there are non-zero elements on the diagonal, that means there are edges connecting vertices to themselves (self-transitions)

## Edge weighted graphs

- Can have same as adjacency matrix but with weights instead of binary
- If total flow out of a vertex is  $> 1$  (its row sums to  $> 1$ ), it is source of mass
- If total flow out of a vertex is  $< 1$  (its row sums to  $< 1$ ), it is a sink
- If total flow is exactly 1 (row sums to 1), it conserves mass

## Stochastic matrices

- Matrix that preserves mass under flow is a stochastic matrix
- A stochastic matrix is a square matrix of non-negative real numbers, with each row summing to 1.0
- A directed graph with a stochastic adjacency matrix is called a Markov chain
- Doubly-stochastic matrices are those where the columns also sum to 1.0 (graphs where flow is conserved in both directions)

## Flow analysis: using matrices to model discrete problems

- Can represent flow at a given time as linear map (square adjacency matrix)
- Can analyse discrete (connectivity of graphs) problem with continuous mathematical tools (vectors/matrices)
- Can predict state of system given initial state (e.g. initial package distribution)  $x_0$ :  
 $x_1 = Ax_0$
- Thus can simulate flow over whole network in one go with just one matrix multiplication (this is vectorised so can take advantage of GPU)

## New matrix operations

- Matrices can be exponentiated:  $C = A^n$  (repeats effect of matrix)
- Matrices can be inverted:  $C = A^{-1}$  (reverse effect of matrix)
- Can find eigenvalues:  
 $Ax_i = \lambda_i x_i$   
this identifies specific vectors  $x_i$  that are only scaled by a factor  $\lambda$  when transformed by a matrix  $A$
- Matrices can be factorised:  $A = U\Sigma V^T$  - any matrix can be expressed as the product of three other matrices with special forms
- We can measure the determinant, trace, and condition number of  $A$  numerically

## Matrix powers (exponentiation)

- $A^k = AA \dots A$  is defined for square matrices (powers of a matrix)
- Matrix exponentiation is the repeated application of a matrix - can only be defined for square matrices (otherwise the dimensions would change after the first step and we wouldn't be able to reapply the same matrix)

## Stable point

- For a conserving adjacency matrix, there will eventually be a steady state (this vector is one of the eigenvectors of the adjacency matrix)

## Eigenvalues and eigenvectors



- A matrix represents a linear transform (rotation and scaling operation on vectors)
- Vectors that don't get rotated when multiplied by the matrix (only scaled) are eigenvectors
- Known as "fundamental" or "characteristic" vectors as they have some stability
- Scaling factors that a matrix applies to its eigenvectors are eigenvalues

## Finding leading eigenvector: the power iteration method

- We can repeatedly apply a square matrix  $A$  to a vector  $x$  of any length
- Then take the resulting vector and apply  $A$  repeatedly
- If we do this to column vectors, the result generally explodes or collapses to 0
- We can normalise the resulting vector after each application of the matrix to fix this (using the L-infinity norm)

$$x_n = \frac{Ax_{n-1}}{\|Ax_{n-1}\|_\infty}$$

- This is power iteration
- Regardless of what vector  $x_0$  is used to start, the power iteration method always approaches a fixed vector (possibly with sign flips)
- This is true for almost every square matrix
- The vector that results from power iteration is the leading (largest) eigenvector
- We can write the scaling effect of the  $A$  on an eigenvector  $x$  as follows:

$$Ax = \lambda x,$$

where  $\lambda$  is the eigenvalue - can calculate  $\lambda$  by dividing  $Ax$  element-wise by  $x$

## Computing eigenvectors and eigenvalues with NumPy

- `np.linalg.eigh` - gives all eigenvectors and eigenvalues of a symmetric matrix
- In general, an  $n \times n$  matrix will have  $n$  eigenvalues and  $n$  eigenvectors
- The eigenvectors are orthogonal (dot prod between any pair of them is 0)
- For very large matrices, computing the leading eigenvector manually with power iteration is much faster than `np.linalg.eigh`

## Formal definition of eigenvectors and eigenvalues

- Any square matrix  $A$  may have vectors such that:  
 $Ax_i = \lambda_i x_i$
- Each vector  $x_i$  satisfying this equation is an eigenvector, and each corresponding factor  $\lambda_i$  is known as an eigenvalue
- Eigenproblems can be tackled with eigenvalues and eigenvectors
- For any matrix the eigenvalues are uniquely determined but the eigenvectors are not

## The eigenspectrum

- Eigenspectrum is just a sequence of absolute eigenvalues ordered ascending by magnitude (ranked by importance)

## Numerical instability of eigendecomposition algorithms

- `np.linalg.eigh` can suffer from numerical instabilities due to rounding errors resulting from limitations on floating point precision, so sometimes the smallest eigenvectors are not completely orthogonal
- If you are using non-symmetric matrices, use `np.linalg.eig` (general purpose eigenvector solver) - more prone to numerical instability, so check that these eigenvectors are orthogonal

## Principle Component Analysis (PCA)

- The eigenvectors of the covariance matrix, scaled by their eigenvalues, form the principal axes of the long thin ellipse formed by covariance matrix data that is strongly correlated

## Decomposition of the covariance matrix into its eigenvectors and eigenvalues

- The eigenvectors of the covariance matrix are called the principal components and they tell us the directions in which data varies the most
- Useful in high-dimensional data sets where the variables may be correlated in complicated manner
- Direction of principal component  $i$  is given by the eigenvector  $x_i$  and the length is given by  $\sqrt{\lambda_i}$

## Reconstruction of the covariance matrix from its eigenvectors and eigenvalues

$$\Sigma = Q\Lambda Q^T$$

- where  $Q$  is a matrix of unit eigenvectors  $x_i$  (same as the output `np.linalg.eig`) and  $\Lambda$  is a diagonal matrix of eigenvalues ( $\lambda_i$  on the diagonal, zero elsewhere).
- Can recover covariance matrix with constituent eigenvectors and eigenvalues

## Approximating a matrix

- Covariance matrix can be large if it comes from a very high dimensional data set
- So we just store first few principal components and reconstruct approximation to it (if we keep the largest ones it should retain most of the information)
- If eigenspectrum contains one large eigenvalue and many small ones - one vector may easily approximate the matrix
- If all eigenspectrum eigenvalues are similar magnitude - not easily approximated

## Dimensionality reduction

- Useful to project original dataset data to the few principal components we keep from the covariance matrix
- Multiply dataset matrix by each component and save the projected data into a new lower-dimensional matrix
- Common to reduce high-D dataset to 2D to see data clusters and other structure

## Matrix properties from the eigendecomposition

- Trace = sum of diagonal values
- $\text{Tr}(A) = a_{1,1} + a_{2,2} + \dots + a_{n,n}$
- Trace is same as sum of eigenvalues of  $A$ :

$$\text{Tr}(A) = \sum_{i=1}^n \lambda_i$$

- Determinant same as how much space expands or contracts after linear transform, equal to product of eigenvalues of the matrix:

$$\det(A) = \prod_{i=1}^n \lambda_i$$

if any eigenvalue of  $A$  is 0,  $\det(A)$  is 0, and the transformation collapses at least one dimension to be completely flat - so the transformation is irreversible and information is lost

## Definite and semi-definite matrices

- All eigenvalues  $> 0$  = positive definite
- All eigenvalues  $\geq 0$  = positive semi-definite
- All eigenvalues  $< 0$  = negative definite
- All eigenvalues  $\leq 0$  = negative semi-definite

## Things we can tell from eigenvectors/values

- One or more zero eigenvalues = matrix performs a transform that collapses one or more dimensions in vector space, and  $A$  is singular (un-invertible)
- Eigenvectors corresponding to larger (absolute) eigenvalues are more "important" - they represent directions

in which data will get stretched most

- Eigenspectrum flat (all eigenvalues similar values) = A represents transform stretching vectors almost equally in all directions
- Eigenspectrum with few large eigenvalues and many small = vectors get stretched along a few direction, shrink away to nothing along others

## Matrix inversion

- $A^{-1}(A\mathbf{x}) = \mathbf{x}$ ,
- $A^{-1}A = I$
- $(A^{-1})^{-1} = A$
- $(AB)^{-1} = B^{-1}A^{-1}$
- Left-multiplication of matrix by its inverse = same as division (reversing effect of original matrix)

## Computing the inverse of a matrix

- We use the singular value decomposition (SVD)
- We use `np.linalg.inv`
- Inversion creates a matrix that undoes the transformation performed by another matrix so long as no information was lost in the transformation
- Only square matrices can be inverted (i.e.  $\det(A) \neq 0$ )
- Non-square matrices cause dimensional collapse - not uniquely reversible
- Invertible matrices must represent a bijection

## Singular and non-singular matrices

- Matrix with  $\det(A) = 0$  is singular, has no inverse
- Matrix which is invertible is non-singular

## Numerical stability of matrix inversion algorithms

- Many repeated floating point operations = many opportunities for roundoff accumulation
- Important to find matrix inversion algorithms that converge reliably to right answer (numerically stable)
- Hard to compute inverses directly in a stable form - so many matrices that could theoretically be inverted cannot be with floating point representation

## Special cases

- Orthogonal matrix = inverse and transpose the same, so just transpose
- Diagonal matrix = the inverse is just another diagonal matrix with diagonal elements that are the reciprocal of each of the original diagonal elements ( $O(n)$  so much faster than standard inversion)

## Solving problems with inversion

- Can "look into the past" by undoing steps (applying the inverse repeatedly)
- Can solve linear systems simply when A is square ( $A^{-1}$  is defined)
- If  $A\mathbf{x} = \mathbf{y}$ , left multiplying both sides by inverse gets:

$$A^{-1}A\mathbf{x} = A^{-1}\mathbf{y}$$

$$I\mathbf{x} = A^{-1}\mathbf{y}$$

$$\mathbf{x} = A^{-1}\mathbf{y}$$

## Singular value decomposition (SVD)

The SVD produces a decomposition which splits **ANY** matrix up into three matrices:

$$A = U\Sigma V^T, [\clubsuit]$$

where

- $A$  is any  $m \times n$  matrix,
- $U$  is a **square unitary**  $m \times m$  matrix, whose columns contain the **left singular vectors**,
- $V$  is an **square unitary**  $n \times n$  matrix, whose columns contain the **right singular vectors**,
- $\Sigma$  is a diagonal  $m \times n$  matrix, whose diagonal contains the **singular values**.
- Unitary matrix = conjugate transpose is equal to inverse
- If  $A$  is real,  $U$  and  $V$  will have rows and columns that all have unit norm
- Can compute SVD with `np.linalg.svd`
- Diagonal of  $\Sigma$  is set of singular values (not quite same as eigenvalues), always real positive numbers

## Relation to eigendecomposition

The SVD is the same as:

- taking the eigenvectors of  $A^T A$  to get  $U$
- taking the square root of the *absolute* value of the eigenvalues  $\lambda_i$  of  $A^T A$  to get  $\Sigma_i = \sqrt{|\lambda_i|}$
- taking the eigenvectors of  $AA^T$  to get  $V^T$
- For a symmetric positive semi-definite matrix  $A$ , the eigenvectors are the columns of  $U$  or the columns of  $V$  - the eigenvalues are in  $\Sigma$ .

## SVD decomposes any matrix into three matrices with special forms

- Special = much easier to work with than general
- $U$  and  $V$  are orthogonal (pure rotation matrix)
- $\Sigma$  is diagonal (pure scaling matrix)

## Using the SVD

- Can compute "square root" of matrix  $A^{1/2}$  or use SVD to raise to any power in just one operation, provided it is a square symmetric matrix
- Raising matrix to fractional power = "part do" operation
- Invert = undo

$$A^n = V\Sigma^n U^T$$

- For a symmetric matrix, this is the same as:

$$A^n = U\Sigma^n V^T$$

- $A^{1/2}$  != elementwise square root

## Inversion using SVD

- Once a matrix is in SVD form, we can efficiently invert it
- For a non-symmetric matrix we use:

$$A^{-1} = V\Sigma^{-1}U^T$$

- This can be computed in  $O(n)$  time because we just take the reciprocal of each the diagonal elements of  $\Sigma$

## Pseudo-inverse

- Can pseudo-invert a matrix  $A^+$  to approximately undo an operation even for non-square  $A$
- Can approximately solve systems of equations where the input variable count is different to the output variable count

We can find approximate solutions for  $\mathbf{x}$  in the equation:

$$\mathbf{Ax} = \mathbf{y},$$

or in fact simultaneous equations of the type

$$\bullet \quad \mathbf{AX} = \mathbf{Y}$$

The pseudo-inverse of  $\mathbf{A}$  is just

$$\mathbf{A}^+ = \mathbf{V}\mathbf{\Sigma}^{-1}\mathbf{U}^T,$$

which is the same as the standard inverse computed via SVD, but taking care that  $\mathbf{\Sigma}$  is the right shape - appropriate zero padding is required!

- Get inverse of covariance matrix with zero padding by using `np.linalg.pinv`
- If the problem does not have an exact solution, the pseudo-inverse will give the closest result according to the  $L2$  norm

## Rank of a matrix

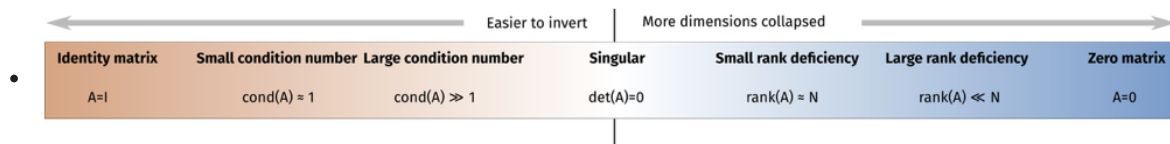
- Equal to number of non-zero singular values
- If rank = matrix size, it is a full rank matrix
- A full rank matrix has non-zero determinant and will be invertible
- Rank tells us number of dimensions of represented parallelotope by the transform
- Non-full rank = singular (non-invertible), deficient rank
- Rank  $\ll$  size = low rank

## Condition number of matrix

- Ratio of largest singular value to smallest
- Only defined for full rank matrices
- Measures how sensitive inversion of matrix is to small changes
- Small condition number = well-conditioned, unlikely to cause numerical issues
- Large condition number = ill-conditioned, numerical issues likely to be significant
- Ill-conditioned matrix is almost singular so inversion will lead to invalid results (floating point roundoff errors)

## Relation to singularity

- Singular matrix  $\mathbf{A}$  un-invertible and has  $\det(\mathbf{A}) = 0$
- Singularity is a binary property (true/false)



- Rank = how singular matrix is (how many dimensions lost in transform)
- Condition number = how close non-singular matrix is to being singular
- Nearly singular matrix may become effectively singular due to floating point roundoff errors

## Applying decompositions

- Whitening a dataset removes all linear correlations within it
- It is a normalising step to standardise data before analysis
- Whitening a dataset stored in matrix  $\mathbf{X}$ :

$$\mathbf{X}^w = (\mathbf{X} - \boldsymbol{\mu})\mathbf{\Sigma}^{-1/2}$$

- Where  $\boldsymbol{\mu}$  is the mean vector, and  $\mathbf{\Sigma}$  is the covariance matrix
- Equation subtracts mean of each column from every element within that column, so each column is centred on 0
- Then it multiplies by inverse square root of covariance matrix (similar to

dividing each column of  $X$  by its std. dev to normalise spread of values in each column)

- Whitening in summary:
  - centres data around its mean, so it has zero mean
  - squashes the data into spherical distribution (gives it unit covariance)

## Week 6: Optimisation I

[\[edit\]](#)

### What is optimisation?

- Process of adjusting things to make them better (searching for optimal solution efficiently by using mathematical structure of the problem space)
- No special cases - formulate problems so that generic algorithms can solve them

### Parameters and objective function

- Parameters = things we can adjust, which might be a scalar or vector or other array of values (denoted  $\theta$ )
- Parameters exist in a parameter space - set of all possible configurations of parameters denoted  $\Theta$  (this space is often a vector space like  $\mathbb{R}^n$ , but doesn't have to be)
- If parameters lie in a vector space, we talk about the parameter vector  $\theta$
- The objective function - maps parameters onto a single numerical measure of how good the configuration is:  $L(\theta)$
- The output of the objective function (a.k.a. loss function) is a single scalar
- Desired output of optimisation algorithm is the parameter configuration that minimises the objective function
- $\theta^* = \arg \min_{\theta \in \Theta} L(\theta)$
- $\theta^*$  is the parameter configuration we want to find to minimise obj func
- $\Theta$  is the set of all possible configurations that  $\theta$  could take on (e.g.  $\mathbb{R}^n$ )
- Most optimisation problems have constraints - limitations on the parameters
- Any maximisation problem can be reframed as a minimisation problem by a sign switch - so considering the objective function as a minimised cost does not lose generality

### Minimising differences

- Objective function is often the measured distance between an output and a reference
- We have some function  $y' = f(x; \theta)$  that produces an output from an input  $x$  governed by a set of parameters  $\theta$
- We measure the difference between the output and some reference  $y$  (e.g. using a vector norm):
 
$$L(\theta) = \|y' - y\| = \|f(x; \theta) - y\|$$
- Optimisation only ever adjusts  $\theta$ , and the vector  $x$  is considered fixed during optimisation ( $x$  might be a collection of real-world measurements, e.g. the keys pressed on a synthesiser which do affect the sound but that aren't optimised, and  $\theta$  might be the knob settings)

### Evaluating the objective function

- Can be expensive (literally dangerous or costly sometimes, but often computationally (e.g. invert 10000x10000 matrix))
- A good optimisation algorithm will find the optimal configuration with few queries of the objective functions
- There must be mathematical structure to guide the search for this
- Without any structure, would have to brute force iterations and choose best outcome (not typically feasible)

### Discrete vs. continuous

- If the parameters are in a continuous space (typically  $\mathbb{R}^n$ ) then the problem is one of continuous optimisation

- Otherwise discrete optimisation
- Continuous optimisation usually easier because we can exploit the concept of smoothness and continuity

## Focus: continuous optimisation in real vector spaces

- $\theta \in \mathbb{R}^n = [\theta_1, \theta_2, \dots, \theta_n]$ ,
- $\theta^* = \arg \min_{\theta \in \mathbb{R}^n} L(\theta)$ , subject to constraints
- The above is the problem of searching a continuous vector space to find the point where  $L(\theta)$  is smallest
- Typically encounter problems where the objective function is smooth and continuous in this vector space
- Parameters being elements of a continuous space does not necessarily imply that the objective function is continuous in that space
- Some optimisation problems are iterative (generate successively better approximations to a solution)
- Can have direct (finding minimum exactly in one step) (unimportant for DF)
- The objective function maps points in space to values (i.e. it defines curve/surface/density which varies across space)
- We want to find a point in space where this is as small as possible as quickly as possible (without going through any "walls" we have defined as constraints)

## Geometric median: optimisation in $\mathbb{R}^2$

- For problem of finding median of a >1D dataset - we know the median minimises the sum of distances to all vectors in the dataset
- e.g. for 2D, parameters would be 2D positions
- Objective function would be sum of distances between a point and a collection of target points  $\mathbf{x}_i$ :

$$L(\theta) = \sum_i \|\theta - \mathbf{x}_i\|_2$$

- We can solve this starting for some random initial condition (guessed  $\theta$ )

## An example of optimisation in $\mathbb{R}^N$

- If in a higher dimension we are trying to find evenly spaced points (with respect to some norm) - we have to optimise a whole collection of points
- We can roll them all up into a single parameter vector
- We can define parameters as an array of 2D positions (we "unpacked" a sequence of 2D points into a higher dimensional vector, so that a whole configuration of points is a single point in vector space)
- The loss function can be the sum of squares of differences between the Euclidean pairwise distances between points and some target distance:

$$\sum_i \sum_j (\alpha - \|x_i - x_j\|_2)^2$$

- This tries to find a configuration of points that are all  $\alpha$  units apart, starting from a random initial condition (e.g. 64 2D points (128 dimensional  $\theta$ ))

## Constrained optimisation

- Sometimes minimising the objective function does not solve the problem - an infeasible set of parameters may produce the minimum loss

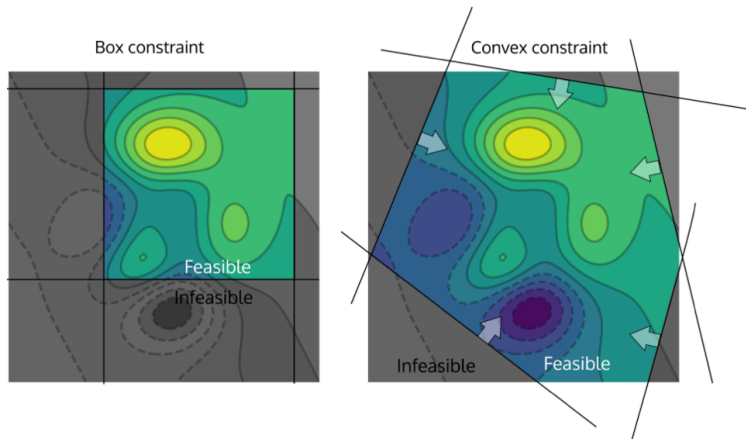
$$\theta^* = \arg \min_{\theta \in \Theta} L(\theta) \text{ subject to } c(\theta) = 0,$$

$$\theta^* = \arg \min_{\theta \in \Theta} L(\theta) \text{ subject to } c(\theta) \leq 0,$$

- $c(\theta)$  is a function that represents the constraints
- We limit the feasible set of the parameters
- Equality constraint = constrain parameters to surface (to represent a trade off) - e.g.  $c(\theta) = \|\theta\|_2 - 1$  forces the parameters to lie on surface of unit sphere

- Inequality constraint = constraint parameters to a volume to represent bounds on the values - e.g.  $c(\theta) = \|\theta\|_\infty$   
- 10 forces the parameters to lie within a box extending  $(-10, 10)$  around the origin

## Common constraint types



- Box constraint =  $\theta$  must lie within a box inside  $R^n$
- Convex constraint = constraint is a collection of inequalities on a convex sum of the parameters  $\theta$
- Box constraints are a specific subclass of convex constraint - equivalent to the feasible set being limited by the intersection of many planes/hyperplanes (possibly an infinite number for curved convex constraints)
- Unconstrained optimisation = any parameter config in search space is possible - often leads to unhelpful results

## Constraints and penalties

- Feasible set is typically not the entire vector space
- Can use constrained optimisation (use an optimisation algorithm that inherently supports hard constraints)
- Straightforward for some kinds of optimisation but hard for others
- Typically constraints are specified as a convex region or a simple (hyper)rectangular region of the space - a box constraint
- Pros - guarantees that solution satisfies constraints, may be able to use constraints to speed up optimisation
- Cons - may be less efficient than unconstrained optimisation, fewer algorithms available, may be hard to specify feasible region

## Soft constraints

- Apply penalties to the objective function to "discourage" solutions that violate the constraints (but soft i.e. there is a tolerance for violation)
- For soft constraints the penalties are just terms added to the objective function
- Optimiser stays the same but the objective function is modified:  
$$L(\theta) = L(\theta) + \lambda(\theta)$$
 where  $\lambda(\theta)$  is a penalty function with an increasing value as the constraints are more egregiously violated
- Pros - any optimiser can be used, can deal with soft constraints sensibly
- Cons - may not respect important constraints, particularly if they are very sharp, can be hard to formulate constraints as penalties, cannot take advantage of efficient search in constrained regions of space

## Relaxation of objective functions

- Can be much harder to solve discrete/constrained optimisation problems efficiently
- So some algorithms try to find similar continuous/unconstrained optimisation problems to solve instead
- This is relaxation - relaxed version of the problem is solved instead of the original hard problem
- Sometimes can absorb constraints in a problem into the objective function, to convert a constrained problem



to unconstrained

## Penalisation

- Refers to terms which augment an objective function to minimise some other property of the solution (typically to approximate constrained optimisation)
- Widely used in approximation problems to find solutions that generalise well (tuned to approximate some data but not too closely)
- This is relaxation of a problem with hard constraints (which need specialised algorithms) to a problem with a simple objective function which works with any objective function

## Penalty functions

- Penalty function is just a term added to an objective function to disfavour "bad solution"
- e.g. for stone throw, could have a maximum throw strength constraint  
could use a constrained optimisation algorithm (don't even search for solutions exceeding max strength)  
or could add a penalty term (function) that increases exponentially after the max strength

## Convexity, global and local minima

- An objective function may have local minima (local minimum is any point where the objective function increases in every direction around that point/parameter setting)
- Convex = objective function has single global minimum
- Convexity implies finding any minimum is the global minimum - we can stop searching in a convex problem (or if we find no minimum we can also stop)

## Convex optimisation

- If the objective function is convex and any constraints form convex portions of the search space, the problem is convex optimisation
- There are very efficient methods for solving these problems even with tens of thousands of variables
  - linear programming (constraints and objective function are linear)
  - quadratic programming (constraints and objective function are quadratic)
- Non-convex problems require the use of iterative methods (although some ways to approximate non-convex with convex problems)

## Continuity

- An objective function is continuous if for some very small adjustment to  $\theta$  there is an arbitrarily small change in  $L(\theta)$
- This means no sudden jumps in value if we move slowly through the space of  $\theta$
- If a function is discontinuous local search methods are not guaranteed to converge to a solution
- Optimisation for discontinuous objective functions is typically much harder than for continuous ones

## Direct convex optimisation: linear least squares

- Sometimes we have an optimisation problem which we can specify such that the solution can be computed in one step
- Linear least squares solves objective functions of the form:

$$\arg \min_x L(\mathbf{x}) = \|\mathbf{Ax} - \mathbf{y}\|_2^2,$$

- It finds  $\mathbf{x}$  closest to the solution  $\mathbf{Ax} = \mathbf{y}$  in the sense of minimising the squared L2 norm (squared just to make the algebra easier to derive)
- Equation is convex - quadratic function that has a single global minimum even in multiple dimensions
- We know it is convex as it has no terms with powers greater than 2 and so is quadratic

- Quadratic functions only ever have zero or one minimum

## Line fitting

- $y = mx + c \rightarrow$  simplest possible linear regression
- If we want to find the  $m$  and  $c$ , such that the squared distance to a set of observed  $(x, y)$  data points is minimised, this is a search over the  $\theta = [m, c]$  space (parameters)
- Objective function is  $L(\theta) = \sum_i (y - mx_i + c)^2$  for some known data points  $[x_0, y_0], [x_1, y_1]$  etc
- We can solve it directly with the pseudo-inverse via the SVD in one step

## Iterative optimisation

- Involves making a series of steps in parameter space
- A current parameter vector (or collection of them) is adjusted at each iteration, to try and decrease the objective function, until optimisation terminates after termination criteria are met
- Iterative optimisation algorithm:
  - choose a starting point  $x_0$
  - while objective function changing
    - adjust parameters
    - evaluate objective functions
    - if better solution found than any so far, record it
  - return best parameter set found

## Regular search: grid search

- Straightforward, but inefficient optimisation algorithm for multidimensional problems
- Simply samples the parameter space by equally dividing the feasible set in each dimension (usually with a fixed number of divisions per dimension)
- Evaluate objective function at each  $\theta$  on the grid, and the lowest loss  $\theta$  is tracked
- Simple and can work for 1D optimisation problems - sometimes used to optimise hyper parameters of ML problems where objective function may be complex but finding the absolute minimum isn't essential

## Revenge of the curse of dimensionality

- Just searching every possible parameter config doesn't scale
- Past 1D and 2D, breaks down completely - would need many evaluations of the objective function, even just 3 points in each dimension is unacceptable

## Density of grid search

- If objective function not very smooth, much denser grid needed to catch any minima
- Pros of grid search - works for any continuous parameter space, requires no knowledge of objective function, trivial to implement
- Cons - incredibly inefficient, must specify search space bounds in advance, highly biased to finding things near "early corners" of the space, depends heavily on number of divisions chosen, hard to tune so that minima are not missed entirely

## Hyperparameters

- Grid search depends on the range searched and the spacing of the divisions on the grid
- Most optimisation algorithms have similar properties that can be tweaked
- These properties that affect the way the optimiser finds a solution are called hyperparameters (not params of objective function but do affect obtained results)
- Perfect optimiser would have no hyperparameters - a solution should not depend on how it was found
- But in practice, all useful optimisers have some which will affect their performance - fewer = better as less cumbersome to tune

## Simple stochastic: random search

- The simplest such algorithm, which makes no assumptions other than we can draw samples from the parameter space, is random search
- Process is simple:
  - guess random parameter  $\theta$
  - check the objective function  $L(\theta)$
  - if  $L(\theta) < L(\theta^*)$  (the previous best parameter  $\theta^*$ ), set  $\theta^* = \theta$
- Termination condition can be fixed number of iterations after the last change in the best loss
- Pros - random search cannot get trapped in local minima, because no local structure is used to guide the search, requires no knowledge of the structure of the objective function (not even a topology), very simple to implement, almost always better than grid search
- Cons - extremely inefficient and is usually only appropriate if there is no other mathematical structure to exploit, must be possible to randomly sample from the parameter space (usually is), results do not necessarily get better over time - no way to predict how optimisation proceeds

## Metaheuristics

- Can use the following to improve random search:
  - locality (take advantage of fact that objective function is likely to have similar values for similar parameter configurations (assumes continuity of the objective function))
  - temperature (can change the rate of movement in the parameter space over the course of optimisation, assuming existence of local optima)
  - memory (can record good or bad steps in the past and avoid/revisit them)

## Locality

- Local search refers to the class of algorithms that make incremental changes to a solution
- These can be more efficient than random/grid search when there is some continuity to the objective function
- But they can get trapped in local minima, not reaching the global minimum
- Since they are used exclusively for non convex problems, this can be a problem
- This implies that the output of the optimisation depends on the initial conditions
- The result might find one local minimum starting from one location, and a different local minimum starting from another
- Local search can be thought of as performing trajectory (a path) through the parameter space, which should hopefully move from higher loss to lower loss

## Hill climbing: local search

- Hill climbing is a modification of random search which assumes some topology of the parameter space, so that there is a meaningful concept of a neighbourhood of a parameter vector; that we can make incremental changes to it
- It is a form of local search that randomly samples configurations near the current best parameter vector - makes incremental adjustments and keeps transitions to neighbouring states only if they improve the loss
- Simple hill climbing adjusts just one of the parameter vector elements at a time, examining each "direction" in turn, and taking a step if it improves loss
- Stochastic hill climbing makes a random adjustment to the parameter vector, then accepts/rejects depending on if the result improved
- Because hill climbing is a local search algorithm, it can get stuck in local minima
- Basic hill climbing has no defence against this and will get trapped in poor solutions if they exist
- Simple hill climbing can also get stuck between ridges and all forms of hill climbing struggle with plateaus where the loss function changes slowly
- Pros - not much more complicated than random search, can be much faster than random search
- Cons - hard to choose how much of an adjustment to make, can get stuck in minima, struggles with objective

function regions that are relatively flat, requires that the objective function be (approximately) continuous

- Can be tweaked:
  - adaptive local search: the size of the neighbourhood can be adapted (e.g. if no improvement in  $n$  iterations, increase size of random steps)
  - multiple restarts: can be used to try and avoid getting stuck in local minima by running the process several times for random initial guesses (another meta-heuristic (applied to the search algorithm itself))

## Temperature

- Simulated annealing extends hill-climbing with the ability to sometimes randomly go uphill, instead of always going downhill
- Uses a temperature schedule that allows more uphill steps at the start of the optimisation and fewer ones later in the process - used to overcome ridges and avoid getting stuck in local minima
- The idea is that allowing random "bad jumps" early in a process can help find a better overall configuration

## Population

- Can use a population of multiple competing potential solutions
- Uses some of:
  - mutation (introducing random variation)
  - natural selection (solution selection)
  - breeding (interchange between solutions)
- These are known as "genetic algorithms"
- All genetic algorithms maintain some population of potential solutions (a set of vectors  $\theta_1, \theta_2, \theta_3, \dots$ ) and some rule is used to preserve some members of the population and cull others
- Parameter set is referred to as the genotype of a solution
- Simple population approaches use small random perturbations and simple selection rule (e.g. keep top 25% of solutions ordered by loss)
- Each iteration perturbs the solutions slightly by random mutation, culls the weakest ones, then copies the "fittest" solutions a number of times to produce the offspring for the next step - population size is held constant between iterations
- Effectively random local search with population
- Can explore a larger area of the space than simple local search and maintain multiple possible hypotheses about what might be optimal at a given time
- Can have crossover rules - crossover introduces some combination of the fittest solutions as the next iteration instead of just copying the parents
- Crossover "merges" two parameter vectors to form a new one (could be more than two)

## Pros and cons of genetic algorithms: population search

- Pros - easy to understand, applicable to many problems, requires only weak knowledge of the objective function, can be applied to problems with both discrete and continuous components, some robustness against local minima although hard to control, great flexibility in parameterisation, mutation schemes, crossover schemes, fitness/selection functions, etc.
- Cons - many "hyperparameters" to tune which radically affect optimisation performance (hard to choose them), no guarantee of convergence (ad hoc), very slow compared to using stronger knowledge of the objective function, many evaluations of objective function are required (one per population member per iteration)

## Memory

- No concept of memory in above optimisation algorithms (all just investigate some part of solution space, check the loss, and move on)
- They may check the same / similar solutions repeatedly
- This inefficiency can be mitigated with memory (for good/bad parts of the parameter space) - we want to

remember good paths in solution space

## Memory + population: ant colony optimisation

- Combines memory and population heuristics - uses stigmergy to optimise problems
- Population of parameter sets = ants
- Memory of good paths through the space = "pheromones"
- Ants who find good parts of the space (i.e. low objective function) leave a trail of positive "pheromones" by storing marker vectors
- Other ants will move towards those pheromones
- The pheromones evaporate over iterations so that the ants don't get constrained into one tiny part of the space
- ACO is good for path-finding and route-finding algorithms, where the memory structure of the pheromone trail corresponds to the solution structure
- Pros - can be very effective in spaces where good solutions are separated by large, narrow valleys, can use fewer evaluations of the objective function than genetic algorithms if pheromones are effective, and when it works, it works very well
- Cons - moderately complex algorithm to implement, no guarantee of convergence (ad hoc), even more hyperparameters than genetic algorithms

## Quality of optimisation

- In convex optimisation, convergence is when the global minimum has been found
- In non-convex optimisation, convergence is when a local minimum has been found that the algorithm cannot escape
- Good optimisation algorithm converges quickly (drop in the objective function should be steep, so that each iteration is making a big difference)
- Bad optimisation does not converge at all (may wander forever or diverge to infinity)
- Many optimisation algorithms only converge under certain conditions (depends on initial conditions)

## Guarantees of convergence

- Some optimisation algorithms are guaranteed to converge if a solution exists
- Others (e.g. most heuristic optimisation algorithms) are not guaranteed to converge even if a solution exists
- A random search might wander the space of probabilities forever and might not even find a specific configuration to minimise (or even reduces) the loss
- Iterative solution objective functions can be plotted (value against iterations) - useful for diagnosing convergence problems

## Tuning optimisation

- Optimisation turns specific problems into ones that can be solved with a general algorithm (as long as we can write down an objective function)
- However, optimisation algorithms have hyperparameters - which affect the way in which the search for the optimum value is carried out
- Using optimisers requires adjusting the hyperparameters
- Use the right algorithm:
  - if you know the problem is least-squares, use a specialised least-squares solver (might be able to directly solve with pseudo-inverse)
  - if you know the problem is convex, use a convex solver (radically more efficient than any other choice if applicable)
  - if you know the derivatives of the objective function, or can compute them with automatic differentiation, use a first-order method (or second-order if possible)
  - if you don't know any of these things, use a general purpose zeroth-order solver (simulated annealing or genetic algorithm)

## What can go wrong?

- Slow progress in local search if steps made are too small
- Noisy and diverging performance in local search (if jumps and steps are too large and the optimiser bounces around hopelessly)
- Optimisers getting stuck (usually at critical points of the objective function)
- Plateaus can cause memoryless algorithms to wander, and derivative-based ones to cease moving entirely (mitigate with momentum and other forms of memory)
- Local minima can completely trap pure local search methods and halt progress (meta heuristics like random restart mitigate this)
- Saddle points can trap/slow gradient descent methods (which have trouble finding best direction to go in when the function is increasing in some directions and decreasing in others)
- Very steep or discontinuous objective functions can produce insurmountable barriers for gradient descent (stochastic methods like stochastic gradient descent can "blur out" these boundaries and still make progress)

## Week 7: Optimisation II

[\[edit\]](#)

### Deep neural networks

- Basic problem of deep learning is to find an approximating function
- In a simple model - given some observations  $x_1$  to  $x_n$ , and some observations  $y_1$  to  $y_n$ , find a function  $y' = f(x; \theta)$  with parameters  $\theta$ , such that we have:

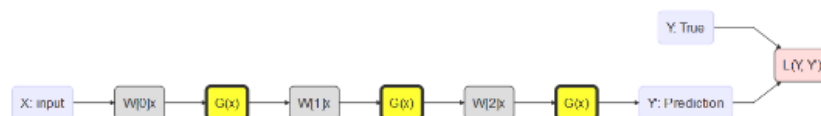
$$\theta^* = \arg \min_{\theta} \sum_i ||f(x_i; \theta) - y_i||$$

where distance is some measure of how close the output of  $f$  and the expected output  $y_i$  are

- We want to learn  $f$  such that we can generalise the transform to an unseen  $x$  (an obvious optimisation problem)
- But deep neural networks can have tens of millions of parameters - a very long  $\theta$  vector - how can we adjust all of these and optimise in a reasonable time in such a large parameter space?

### Backpropagation

- Traditional neural network consists of a series of layers, each of which is a linear map (a matrix multiply) followed by a simple, fixed, nonlinear function
- Think: rotate, stretch (linear map), and fold (simple fixed nonlinear folding)
- The output of one layer is the input for the next



*Image: a 3 layer deep network. Each layer consists of a linear map  $W$  applied to the input  $x$  from the previous layer, followed by a fixed nonlinear function  $G(x)$*

- The linear map in each layer is specified by a weight matrix
- The network is completely parameterised by the entries of the weight matrices for each layer (all of the entries of the matrices can be seen as equivalent to the parameter vector  $\theta$ )
- The nonlinear function  $G(x)$  is fixed for every layer and cannot vary (it is often a simple function that "squashes" the range of the output in some way)
- Only the weight matrices change during optimisation:  

$$y_i = G(W_i x_i + b_i)$$
- This particular construction (under certain conditions) has the massive advantage that the derivative of the objective function with respect to the weights can be computed for every weight in the network at the same time, even when multiple layers are composed together

- The algorithm that does that is called backpropagation (an algorithm for automatic differentiation)
- Derivative with respect to the weights means we can tell how much of an effect each weight will have on the prediction, for every weight in the whole network, in one go
- Can imagine all elements of weight matrices concatenated into a single vector  $\theta$  - so that we can get the gradient of the objective function w.r.t.  $\theta$
- This makes the optimisation "easy" - we can walk in the direction that takes us downhill fastest

## Why not use heuristic search?

- Heuristic search methods (random search, simulated annealing, genetic algorithms) are easy to understand/implement and are broadly applicable
- But they can be very slow (may need many iterations to approach a minimum and require significant computation per iteration)
- No guarantee of convergence/any progress (search can get stuck or drift over plateaus)
- Huge number of hyperparameters that can be tweaked (temperature schedules, size of population, memory structure, etc) - optimal choice of these parameters becomes an optimisation problem in itself
- Heuristic search is inadequate for optimisation problems like DNNs
- Too slow to make progress in training networks with millions of parameters
- First-order optimisation is applied instead (using first-order algorithms, which can be orders of magnitude faster than heuristic search)

## Jacobian: matrix of derivatives

- Generalise definition of first and second derivative to a vector function  $y=f(x)$  - so we have a derivative between every input component and every output component at any specific input  $x$
- We collect this information into a Jacobian matrix below (characterising the slope at a specific input point  $x$ )
- If  $x \in \mathbb{R}^n$  and the output  $y \in \mathbb{R}^m$ , we have an  $m \times n$  matrix

$$f'(x) = J = \begin{bmatrix} \frac{\partial y_0}{\partial x_0} & \frac{\partial y_0}{\partial x_1} & \dots & \frac{\partial y_0}{\partial x_n} \\ \frac{\partial y_1}{\partial x_0} & \frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial y_1}{\partial x_n} \\ \dots & \dots & \dots & \dots \\ \frac{\partial y_m}{\partial x_0} & \frac{\partial y_m}{\partial x_1} & \dots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

- This tells us how much each component of the output changes as we change any component of the input (the generalised slope of a vector valued function)
- In the case where  $f$  maps  $\mathbb{R}^n \rightarrow \mathbb{R}^n$  (from a vector space to the same vector space), we have a square  $n \times n$  matrix  $J$  from which we can compute the determinant, take the eigendecomposition (or in some cases invert)
- In many cases, the Jacobian is simple, just one single row
- This applies in cases where we have a scalar function  $y = f(x)$  where  $y \in \mathbb{R}$ , where  $y$  is a one dimensional input from an  $n$  dimensional input)
- This is the situation we have with a loss function  $L(\theta)$  that is a scalar function of a vector input - the single row Jacobian is the gradient vector here

## Gradient vector: one row of the Jacobian

- $\nabla f(x)$  is the gradient vector of a (scalar) function of a vector
- Equivalent to first derivative for vector functions
- We have one (partial) derivative per component of  $x$
- This tells us how much  $f(x)$  would vary if we made tiny changes to each dimension independently

$$\nabla L(\theta) = \left[ \frac{\partial L(\theta)}{\partial \theta_1}, \frac{\partial L(\theta)}{\partial \theta_2}, \dots, \frac{\partial L(\theta)}{\partial \theta_n} \right]$$

- If  $L(\theta)$  is a map  $\mathbb{R}^n \rightarrow \mathbb{R}$ , (i.e. a scalar function, like an ordinary objective function) then  $\nabla L(\theta)$  is a vector valued map  $\mathbb{R}^n \rightarrow \mathbb{R}^n$ ;
  - If  $L(\theta)$  was a map  $\mathbb{R}^n \rightarrow \mathbb{R}^m$ , then  $\nabla L(\theta)$  is matrix valued map  $\mathbb{R}^n \rightarrow \mathbb{R}^{m \times n}$ ;
- $\nabla L(\theta)$  is a vector which points in the direction of the steepest change in  $L(\theta)$

## Hessian: Jacobian of the gradient vector

- $\nabla^2 f(\mathbf{x})$  is the Hessian matrix of a (scalar) function of a vector
- Equivalent to second derivative for vector functions
- From above rules it is just the Jacobian of a vector valued function, and thus a matrix valued map from  $\mathbb{R}^n$  to  $\mathbb{R}^{n \times n}$
- This means we can see that the second derivative even of a scalar valued function scales quadratically with the dimension of its input
- If the original function was a vector we'd get a Hessian tensor

$$H(L) = \nabla \nabla L(\theta) = \nabla^2 L(\theta) = \begin{bmatrix} \frac{\partial^2 L(\theta)}{\partial \theta_1^2} & \frac{\partial^2 L(\theta)}{\partial \theta_1 \partial \theta_2} & \frac{\partial^2 L(\theta)}{\partial \theta_1 \partial \theta_3} & \dots & \frac{\partial^2 L(\theta)}{\partial \theta_1 \partial \theta_n} \\ \frac{\partial^2 L(\theta)}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 L(\theta)}{\partial \theta_2^2} & \frac{\partial^2 L(\theta)}{\partial \theta_2 \partial \theta_3} & \dots & \frac{\partial^2 L(\theta)}{\partial \theta_2 \partial \theta_n} \\ \dots & \dots & \dots & \dots & \dots \\ \frac{\partial^2 L(\theta)}{\partial \theta_n \partial \theta_1} & \frac{\partial^2 L(\theta)}{\partial \theta_n \partial \theta_2} & \frac{\partial^2 L(\theta)}{\partial \theta_n \partial \theta_3} & \dots & \frac{\partial^2 L(\theta)}{\partial \theta_n^2} \end{bmatrix}$$

## Differentiable objective functions

- For some objective functions we can just compute the exact derivatives (e.g. if the objective function has a single scalar parameter  $\theta$ ):

$$L(\theta) = \theta^2$$

then, from basic calculus, the derivative with respect to  $\theta$  is just:

$$L'(\theta) = 2\theta.$$

- If we know the derivative we can move in "good directions" - down the slope of the objective function towards a minimum
- Becomes slightly more involved for multidimensional objective functions (where  $\theta$  has  $>1$  component) - we get a gradient vector instead of a scalar derivative - but still useful in the same way

## Orders: zeroth, first, second

- Iterative algorithms are:
- Zeroth order optimisation algorithms if they only require evaluation of the objective function  $L(\theta)$  - e.g. random search, simulated annealing
- First order if they only require evaluation of  $L(\theta)$  and its derivative  $\nabla L(\theta)$  - e.g. the family of gradient descent methods
- Second order if they require evaluation of  $L(\theta)$ ,  $\nabla L(\theta)$ , and  $\nabla \nabla L(\theta)$  - e.g. quasi-Newtonian optimisation

## Optimisation with derivatives

- If we know (or can compute) the gradient of an objective function, we know the slope of the function at any given point
- This gives us the direction of fastest increase and the steepness of the slope

## Differentiability



- A smooth function has continuous derivatives up to some order
- Smoother functions are typically easier to do iterative optimisation on, because small changes in the current approximation are likely to lead to small changes in the objective function
- A function is  $C^n$  continuous if the  $n$ th derivative is continuous
- First order optimisation uses the (first) derivatives of the objective function with respect to the parameters
- We can only find this if the objective function is at least  $C^1$  continuous i.e. no step changes anywhere in the function/its derivative, and only if the objective function is differentiable (gradient is defined everywhere)
- Many objective functions satisfy these conditions, and first-order methods can be vastly more efficient than zeroth-order methods
- For some classes of functions (e.g. convex) there are known bounds on the number of steps required to converge for specific first-order optimisers

## Lipschitz continuity (no ankle breaking)

- First-order (and higher-order) continuous optimisation algorithms need more than just  $C^1$  continuity and require them to be Lipschitz continuous
- For functions  $\mathbb{R}^n$  to  $\mathbb{R}$ , this is equivalent to saying the gradient is bounded and never changes faster than some constant (a fixed  $K$ ):

$$\frac{\partial L(\theta)}{\partial \theta_i} \leq K$$

## Lipschitz constant

- $K$  is found by this formula:

$$K = \sup \left[ \frac{|f(x) - f(y)|}{|x - y|} \right]$$

where sup is the supremum (the smallest value that is larger than every value of this function)

- A smaller  $K$  means a function that is smoother
- $K = 0$  is totally flat
- $K$ 's value may not always be precisely known

## Analytical derivatives

- If we have analytical derivatives (the derivative as a written expression), we can just compute the derivative, solve for  $f'(x) = 0$  to finding all the turning points, then check if any of the solutions have a positive second derivative  $f''(x) = 0$  (therefore indicating a minimum)

## Computable exact derivatives

- Analytical derivative approach doesn't require any iteration - we instantly get exact pointwise derivatives
- In contrast, we can evaluate  $f'(x)$  for any  $x$  but not write it down in closed form
- In this case we can still make optimisation much faster by taking steps to run "downhill" as fast as possible (requires ability to compute gradient at any point on the objective function)

## Gradient: a derivative vector

- The vector of derivatives is given by the following:

$$\nabla L(\theta) = \left[ \frac{\partial L(\theta)}{\partial \theta_1}, \frac{\partial L(\theta)}{\partial \theta_2}, \dots, \frac{\partial L(\theta)}{\partial \theta_n} \right]$$

•

**Note:**  $\frac{\partial L(\theta)}{\partial \theta_1}$  just means the change in  $L$  in the direction  $\theta_1$  at the point  $\theta$ .

- This is the gradient vector (at any given point, the gradient of a function points in the direction of fastest increase) - the magnitude of this vector is the rate of change of the function ("steepness")

## Gradient descent

- The basic first-order algorithm is called gradient descent
- You give it an initial guess  $\theta^{(0)}$ :

$$\theta^{(i+1)} = \theta^{(i)} - \delta \nabla L(\theta^{(i)})$$

where  $\delta$  is a scaling hyper parameter - the step size

- The step size might be fixed or might be chosen adaptively by an algorithm like line search
- This means the optimiser will make moves where the objective function drops most quickly
- Simple version:
  - starting somewhere  $\theta^{(0)}$
  - repeat:
    - check how steep the ground is in each direction  $v = \nabla L(\theta^{(i)})$
    - move a little step  $\delta$  in the steepest direction  $v$  to find  $\theta^{(i+1)}$
- $\theta^{(i)}$  does not mean the  $i$ th power of  $\theta$ , just the  $i$ th  $\theta$  in a sequence of iterations  $\theta^{(0)}, \theta^{(1)}, \theta^{(2)}, \dots$

## Downhill is not always the shortest route

- We are trying to follow the steepest slope not the shortest route

## Why step size matters

- The step size  $\delta$  is critical for success
- If it's too small, convergence will be slow
- If it's too large, then optimisation behaviour can become unpredictable (the gradient function could change significantly (change sign) over the space of a step)

## Gradient descent in 2D

- Same as 1D but we need to be able to get the gradient vector at any point in the parameter space instead of a simple 1D derivative - no code changes

## Gradients of the objective function

- For first-order optimisation we need the derivative of the objective function to be available
- This does not apply directly to empirical optimisation (but if we have a computational model, it can be optimised) - favour building models when optimising

## Why not use numerical differences?

- Definition of differentiation of a function  $f(x)$ :

$$\frac{d}{dx} f(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x-h)}{2h}$$

- If we can evaluate  $L(\theta)$  anywhere, we could just evaluate  $L(\theta + h)$  and  $L(\theta - h)$  for some small  $h$
- This is numerical differentiation (and these are finite differences)
- Works fine for reasonably smooth one-dimensional functions

## Numerical problems

- It is difficult to choose  $h$  such that the function is not misrepresented by an excessive value but numerical issues do not dominate the result

Remember that finite differences violates *all* of the rules for good floating point results:

$$\frac{f(x+h) - f(x-h)}{2h}$$

- (a) it adds a small number  $h$  to a potentially much larger number  $x$  (*magnitude error*)
- (b) it then subtracts two very similar numbers  $f(x+h)$  and  $f(x-h)$  (*cancellation error*)
- (c) then it divides the result by a very small number  $2h$  (*division magnification*)

It would be hard to think of a simple example that has more potential numerical problems than finite differences!

## Revenge of the curse of dimensionality again

- Not useful in high dimensions, even if we could deal with numerical issues
- To evaluate the gradient at a point  $x$  we'd have to compute the numerical differences in every dimension
- If  $\theta$  had 1 million dimensions we'd need 2 million evaluations of  $L(\theta)$  for each individual derivative evaluation
- The overhead from this gradient evaluation would drown out the acceleration of first-order methods over zeroth-order

## Improving gradient descent

- The gradient of the loss function  $L'(\theta) = \nabla L(\theta)$  must be computable at any point  $\theta$  - automatic differentiation helps with this
- Gradient descent can get stuck in local minima - this is an inherent aspect of gradient descent methods, which will not find global minima except when the function is convex and the step size is optimal - use random restart and momentum to reduce sensitivity to local minima
- Gradient descent only works on smooth, differentiable objective functions - use stochastic relaxation to introduce randomness, allowing very steep functions to be optimised
- Gradient descent can be very slow if the objective function and/or the gradient is slow to evaluate - stochastic gradient descent can massively accelerate optimisation if the objective function can be written as a simple sum of many subproblems

## Automatic differentiation

- If we know analytically the derivative of the objective function in closed form, we can solve the problem
- But it can be constraining to have to manually work out the derivative of the objective function (e.g. complex multidimensional problem)
- Automatic differentiation can take a function (usually written as a subset of a full programming language)

## Programming language advances

- Vectorised programming - provides GPU accelerated operations over tensors
- Differentiable programming - automatic differentiation of vectorised code, producing exact derivatives of tensor algorithms
- Probabilistic programming - allows values to be uncertain, with (tensor, differentiable) random variables as first class values

## Autograd

- Provides automatic differentiation for virtually any NumPy code
- Using automatic differentiation, we can write down the form of the objective function as a straightforward computation, and get the derivatives of the function "for free"
- Makes it extremely inefficient to perform first-order optimisation

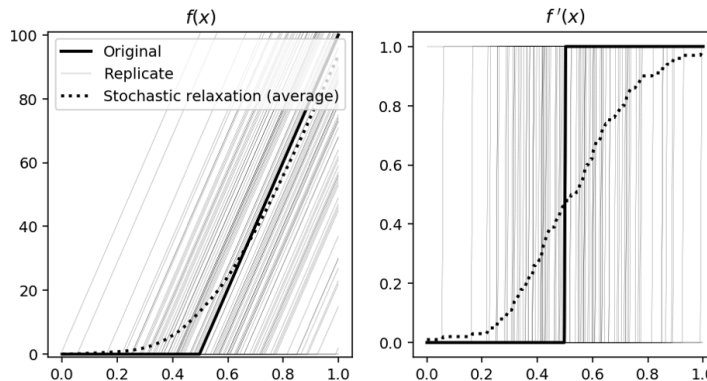
## Limits of automatic differentiation

- Only available for functions that are differentiable
- First-order gradient vectors often computable in reasonable time - becomes very difficult to compute second

derivatives of multidimensional vectors

## Stochastic relaxation and resolution

- Need a roughly continuous fitness function (smooth path from poor fitness to good fitness) to have some steps accepted in optimisation
- Although every specific case is a binary choice, it is averaged over many random instances, where the conditions may be slightly different, and averaging may show some minor change offering an advantage
- This averaging is stochastic relaxation
- An apparently impossibly steep gradient is rendered Lipschitz continuous by integrating over many different random conditions
- e.g. a very steep function may have a very large derivative at one point and zero derivative at other parts - but if we average over lots of cases where the step position is very slightly shifted, we get a smooth function



## Stochastic gradient descent

- Gradient descent evaluates the objective function and its gradient at each iteration before making a step - this can be expensive to do (e.g. optimising function approximations with large data sets like in ML)
- If the objective function can be broken down into small parts, the optimiser can do gradient descent on randomly selected parts independently, which may be much faster
- This is called SGD, because the steps it takes depend on the random selection of the parts of the objective function
- This works if the objective function can be written as a sum:

$$L(\theta) = \sum_i L_i(\theta)$$

i.e. the objective function is composed of the sum of many simple sub-objective functions  $L_1(\theta), L_2(\theta), \dots, L_n(\theta)$

- This type of form often occurs when matching parameters to observations (approximation problems)
- In these cases, we have many training examples  $x_i$  with known matching outputs  $y_i$
- We want to find a parameter vector  $\theta$  such that:

$$L(\theta) = \sum_i \|f(x_i; \theta) - y_i\|$$

is minimised (the difference between the model output and the expected output is minimised, summing over all training examples)

- Differentiation is a linear operator, so we can interchange summation, scalar multiplication, and differentiation
- $\frac{d}{dx}(af(x) + bg(x)) = a\frac{d}{dx}f(x) + b\frac{d}{dx}g(x)$
- $\nabla \sum_i \|f(x_i; \theta) - y_i\| = \sum_i \nabla \|f(x_i; \theta) - y_i\|$
- Above means that the gradient of the summation of all output differences is the same as the summation of the gradient of all output differences
- So we can take any subset of training samples and outputs, compute the gradient for each sample, and make

- a move according to the computed gradient of the subset
- Over time, the random subset selection will (hopefully) average out
- Each subset is called a minibatch and one run through the whole dataset (i.e. enough batches so that every data item has been "seen" by the optimiser) is called an epoch

## Memory advantages of SGD

- Minibatches provide a good enough approximation of the right direction to move in by computing the gradient just on a subsample
- Gets around having to store whole model in VRAM
- Can also improve memory hierarchy (small batches = fewer cache misses, better performance)

## Heuristic enhancement

- SGD can reduce chance of getting stuck in minima
- Random partitioning of the objective function in each minibatch adds noise to the optimisation process (noise could help it go uphill over a maxima)
- Adding noise is a heuristic search approach (no guarantee it will improve or not worsen result), but it is often very effective
- Essentially getting benefits from a limited form of stochastic relaxation (we are smoothing out the objective function by averaging over the random subsamples, so SGD can work well even if the objective function is not quite Lipschitz continuous (or has a bad Lipschitz constant))

## Using SGD

- No guarantee it even moves in the right direction
- In practice it can be very efficient for many real world problems (e.g. finding parameters  $m, c$  to minimise an  $mx + c \cdot y'$  and  $y$  function squared error, by computing the gradient on a random subset)

## Linear regression with SGD

- Can find best fit line of many points with only one pass of data - we can divide the problem into a lot of sums of smaller problems (fitting a line on a few random points at a time), which are all part of one big problem
- Vastly more efficient than computing gradient for entire dataset

## Random restart

- Gradient descent gets trapped in minima easily
- Once it is in an attractor basin of a local minima it cannot easily get out
- SGD noise can help the optimiser get out of small ridges/peaks but not deep minima
- Simple heuristic is to just run gradient descent until it gets stuck, and then randomly restart with different initial conditions and try again
- This is repeated a number of times, hopefully with one of the optimisation paths ending in the global minimum
- This metaheuristic works for any local search method (hill climbing, simulated annealing)

## Simple memory: momentum terms

- Physical momentum is a simple form of the memory heuristic
- If you are going the right way now, keep going that way even if the gradient is not always quite downhill
- We can reduce the chance of (stochastic) gradient descent becoming trapped by small fluctuations in the objective function and "smooth" out the descent
- We introduce a velocity  $v$  and have the optimiser move in this direction - we gradually adjust  $v$  to align with the derivative:

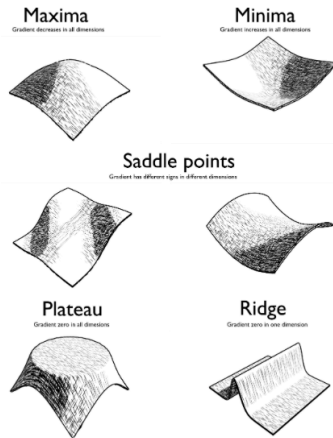
$$v = \alpha v + \delta \nabla L(\theta)$$

$$\theta^{(i+1)} = \theta^{(i)} - v$$

- Governed by the parameter  $\alpha$ :
  - $\alpha$  closer to 1.0 = more momentum in system, 0.0 = ordinary gradient descent

## Types of critical points

- Critical point = where the gradient vector components are the zero vector



## Second-order derivatives

- If the first order derivatives represent the "slope" of a function, then the second order derivatives represent the "curvature" of a function
- For every parameter component  $\theta_i$  the Hessian stores how the *steepness* of every other  $\theta_j$  changes
- The eigenvalues of the Hessian capture information about the type of critical point
- All strictly positive eigenvalues. = positive definite matrix, the point is a maximum
- All strictly negative = negative definite matrix, point is a maximum
- If eigenvalues have mixed sign, point is a saddle point
- If eigenvalues all positive/negative, but with some zeros, matrix is semidefinite and the point is plateau/ridge

## Second-order optimisation

- Uses the Hessian matrix to jump to the bottom of each local quadratic approximation in a single step
- This can skip over flat plains and escape from saddle points that slow down gradient descent
- Second order methods are generally much faster than first order
- But curse of dimensionality means simple second order methods do not work in high dimensions (memory constraint)
- Evaluating the Hessian matrix requires  $d^2$  computations and  $d^2$  storage
- Many ML applications have models with  $d > 1M$  parameters
- Second order optimisation can move much faster through saddle points and plateaus than first order methods like gradient descent, and can be particularly effective for low dimensional problems

## Week 8: Probability I

[\[edit\]](#)

### What is probability?

- Experiment (or trial) = an occurrence with an uncertain outcome
- Outcome = results of an experiment, one particular state of the world
- Sample space = the set of all possible outcomes for an experiment
- Event = a subset of possible outcomes with some common properties
- Probability = the probability of an event with respect to a sample space is the number of outcomes from the sample space that are in the event divided by the total number of outcomes in the sample space (ratio)

between 0 and 1)

- Probability distribution = a mapping of outcomes to probabilities that sum to 1
- Random variable = variable representing unknown value whose probability distribution we do know (variable associated with outcomes of a trial)
- Probability density / mass function = function that defines a probability distribution by mapping each outcome to a probability  $f_X(x), x \rightarrow \mathbb{R}$ .
- Observation = outcome that we directly observed (i.e. data)
- Sample = an outcome that we have simulated according to a probability distribution (we draw samples from a distribution)
- Expectation/expected value = the "average" value of a random variable

## Bayesian/Laplacian view on probability

- Bayesians treat probability as a calculus of belief
- Probabilities are measures of degrees of belief
- $P(A) = 0$  means a belief that event A cannot be true
- $P(A) = 1$  means a belief that event A is absolutely certain
- The probability quantifies our belief about the event given the information we have
- Bayesians allow for belief in states to be combined and manipulated via the rules of probability
- The key process in Bayesian logic is the updating of beliefs
- Given some prior belief and new evidence, update our belief to calculate the posterior (new probability of event)
- Bayesian inference requires that we accept priors over events (i.e. that we must explicitly quantify our assumptions with probability distributions - it is an extension of logic to uncertain information)

## Frequentist view of probability

- Alternative school of thought that considers probabilities to be only the long-term behaviour of repeated events (e.g. probability of coin toss heads is 0.5 because long term average proportion of occurrences is this)
- A frequentist does not accept phrases like "what is the probability it is sunny right now?" as there is no long term behaviour involved (it is only "now" once)
- It does not make sense in this world view to talk about the probability of events that can only happen once
- It does make sense in a frequentist view to ask "what is the probability it is sunny on a given day?", since we can measure this event on many different days

## Objectivity and subjectivity

- Bayesian theory sometimes said to be subjective as it requires the specification of prior belief, whereas frequentist models of probability do not admit the concept of priors and thus are objective
- Alternative view is that Bayesian model explicitly encodes uncertain knowledge and states universal formal rules for manipulating that knowledge, as formal logic does for definite knowledge
- Frequentist methods are objective in the sense that they make statements about universal truths (e.g. asymptotic behaviour), but they do not form a calculus of belief, and thus can't answer directly many questions of importance
- Bayesian: includes priors and probability is a degree of belief, (parameters of population considered random variables, data to be known)
- Frequentist: no priors, probability is the long-term frequency of events, (parameters of population assumed to be fixed, data to be random)

## Generative models: forward and inverse probability

- Key idea of probabilistic models is that of a generative process
- There is some unknown process going on, the results of which can be observed
- The process itself is governed by unobserved variables that we do not know but which we can infer
- Forward probability = questions related to distribution of the observations

- Inverse probability = questions related to unobserved variables that govern the process that generated the observations

## Axioms of probability

- Boundedness:  $0 \leq P(A) \leq 1$
- Unitarity:  $\sum_A P(A) = 1$  (complete set of possible outcomes (not events)  $A \in \sigma$  in a sample space  $\sigma$  sums to 1  
- something always happens)
- Sum rule:  
 $P(A \vee B) = P(A) + P(B) - P(A \wedge B)$   
i.e. the probability of event A or B happening is the sum of the independent probabilities minus the probability of both happening
- Conditional probability:  $P(A|B)$  = probability that event A will happen given that we know that event B already happened:

$$P(A|B) = \frac{P(A \wedge B)}{P(B)}$$

## Random variables and distributions

- Random variable = variable that can take on different values but we do not know what value it has ("unassigned")
- But we have some knowledge which captures the possible states the variable could take on, and their corresponding probabilities
- A random variable is written with a capital
- Can be discrete (e.g. outcome of dice throw)
- Can be discrete binary (e.g. is it raining outside?)
- Can be continuous (e.g. height of person we haven't met yet)

## Distributions

- Probability distribution defines how likely different states of a random variable are
- We can see X as the experiment and x as the outcome, with a function mapping every possible outcome to a probability
- $P(X = x)$ , the probability of random variable X taking on value x
- $P(X)$ , shorthand for probability of  $X = x$
- $P(x)$ , shorthand for probability of specific value  $X = x$

## Discrete and continuous

- Random variables can be continuous (e.g. person height) or discrete (e.g. value showing on face of a dice)
- Discrete variables - the distribution of a discrete random variable is described by a probability mass function (PMF) which gives each outcome a specific value (the PMF is usually written  $f_X(x)$  where  $P(X = x) = f_X(x)$ )
- Continuous variables - a continuous variable has a probability density function (PDF) which specifies the spread of the probability over outcomes as a continuous function  $f_X(x)$  (it is not the case that  $P(X = x) = f_X(x)$ )  
for PDFs

## Integration to unity

- A probability mass function or probability density function must sum/integrate to exactly 1, as the random variable in consideration must take on some value (consequence of unitarity)
- Every repetition of an experiment has exactly one outcome



- $\sum_i f_X(x_i) = 1$  for PMFs of discrete RVs
- $\int_x f_X(x) dx = 1$  for PDFs of continuous RVs

## Expectation

- If a random variable takes on numerical values, we can define the expectation or expected value of a random variable  $E[X]$  as:

$$E[X] = \int_x x f_X(x) dx$$

- For a discrete random variable with probability mass function  $P(X = x) = f_X(x)$  we write this as a summation:

$$E[X] = \sum_x x f_X(x)$$

- If there are only a finite number of possibilities then this is:

$$E[X] = P(X = x_1)x_1 + P(X = x_2)x_2 + \dots + P(X = x_n)x_n$$

- The expectation is the "average" of a random variable
- What we'd "expect to happen", the most likely overall outcome
- Weighted sum of all the outcomes of an experiment, where each outcome is weighed by its occurrence probability

## Expectation and means

- Expected value of a random variable is the true average of the value of all outcomes that would be observed if we ran the experiment an infinite number of times (the population mean, the mean of the whole, possibly infinite population of a random variable)
- The mean of a random variable  $X$  is just  $E[X]$ , it is a measure of central tendency
- The variance of a random variable  $X$  is  $\text{var}(X) = E[(X - E[X])^2]$ , it is a measure of spread

## Expectations of functions of $X$

- We can apply functions to random variables (e.g. squaring a random variable)

The expectation of any function  $g(X)$  of a continuous random variable  $X$  is defined as:

$$E[g(X)] = \int_x f_X(x)g(x)dx$$

- or

$$E[g(X)] = \sum_x f_X(x)g(x)$$

for a discrete random variable.

- $E[g(X)] \neq g(E[X])$
- Expected values are essential to making rational decisions, the central problem of decision theory
- They combine utility with probability

## Samples and sampling

- Samples are observed outcomes of an experiment
- Term interchangeable with observations, but samples normally refers to simulations/observations of concrete real world data
- We can sample from a distribution (simulating outcomes according to the probability distribution of those variables)
- We can observe data which comes from an external source, that we think might be generated by some probability distribution

- For discrete random variables this is easy (produce samples by drawing each outcome according to its probability)

## The empirical distribution

- For discrete data, can estimate the probability mass function that might be generating observations by counting each outcome seen, divided by the total number of trials
- This is called the empirical distribution
- Can be thought of as the normalised histogram of counts of occurrences of outcomes

## Computing the empirical distribution

- For discrete random variables, can always compute the empirical distribution from a series of observations:  

$$P(X = x) = n_x / N$$
 where  $n_x$  is the number of times outcome  $x$  was observed, and  $N$  is the total number of trials
- The empirical distribution approximates an unknown true distribution
- For very large samples of discrete variables, the empirical distribution will increasingly closely approximate the true PMF, assuming samples are drawn in an unbiased way
- However, this approach does not work usefully for continuous random variables (since we will only ever see each observed value once)

## Uniform sampling

- These are algorithms which can generate continuous random numbers which are uniformly distributed in an interval
- Actually pseudo-random in practice
- Designed to approximate the statistical properties of true random sequences
- All generators generate sequences of discrete symbols (bits/integers) which are then mapped onto floating point numbers in a specific range
- A uniformly distributed number has equal probability of taking on any value in its interval, and zero probability everywhere else
- Although this is sampling from a continuous PDF it is the key building block in sampling from arbitrary PMFs
- A uniform distribution is notated  $X \sim U(a, b)$ , meaning  $X$  is a random variable which may take on values between  $a$  and  $b$ , with equal possibility of any number in that interval
- Symbol  $\sim$  is read as "distributed as" - i.e. " $X$  is distributed as a uniform distribution in the interval  $[a, b]$ "
- In practice, not uniform across the reals as we can only sample valid floating point values (which are not uniformly distributed)

## Discrete sampling

- For a discrete probability mass function, we can sample outcomes according to any arbitrary PMF by partitioning the unit interval
- Algorithm:
  - choose any arbitrary ordering for the outcomes  $x_1, x_2, \dots$
  - assign each outcome a "bin" which is a portion of the interval  $[0, 1]$  equal to its probability, so that the interval is divided into consecutive non-overlapping regions  $[P(x_1) \rightarrow P(x_1) + P(x_2), P(x_1) + P(x_2) \rightarrow P(x_1) + P(x_2) + P(x_3), \dots]$
  - draw a uniform sample in the range  $[0, 1]$
  - whichever "outcome bin" it lands in is the sample to draw
- By definition of PMF, the sum of all the probabilities will be 1.0, so it will fill the interval  $[0, 1]$  perfectly with no gaps

## Joint, conditional, marginal

- The joint probability of two random variables is written  $P(X, Y)$  and gives the probability that  $X$  and  $Y$  take

specific values simultaneously (i.e.  $P(X = x) \wedge P(Y = y)$ )

- The marginal probability is the deviation of  $P(X)$  from  $P(X, Y)$  by integrating (summing) over all the possible outcomes of  $Y$ :

$$P(X) = \int_y P(X, Y) dy \text{ for a PDF.}$$

$$P(X) = \sum_y P(X, Y) \text{ for a PMF.}$$

- This allows us to compute a distribution over one random variable from a joint distribution by summing over all the possible outcomes of the other variable involved
- Marginalisation just means integration over one or more variables from a joint distribution: it removes those variables from the distribution
- Two random variables are independent if they do not have dependence on each other - if this is the case then the joint distribution is just the product of the individual distributions:  $P(X, Y) = P(X)P(Y)$ ; this is not true in the general case where the variables have dependence
- The conditional probability of a random variable  $X$  given a random variable  $Y$  is written as  $P(X | Y)$  and can be computed as:  
 $P(X | Y) = P(X, Y) / P(Y)$
- This tells us how likely the outcomes of  $X$  are if we fix the outcomes of  $Y$

## Bigrams

- Bigram model = uses every pair of units (e.g. character/word in text)
- Can have unigram, trigram, n-gram
- The joint distribution of bigrams  $P(C_i = c_i, C_{i-1} = c_{i-1})$  is given by the normalised count of each character pair
- The marginal distribution  $P(C_i = c_i)$  can be computed from  $P(C_i = c_i, C_{i-1} = c_{i-1})$  by summing over every possible character  $c_{i-1}$ , and likewise to marginalise to find  $P(C_{i-1} = c_{i-1})$
- The conditional distribution  $P(C_i = c_i | C_{i-1} = c_{i-1})$  is given by the joint distribution, divided by the counts of  $P(C_{i-1} = c_{i-1})$  - it tells us how likely we are to observe a specific character  $c_i$  given that we have observed a character  $c_{i-1}$  just beforehand
- Joint probability tells us how likely each possible pair of units is
- Marginal probability tells us how likely each unit is to occur
- Conditional probability tells us what unit to expect given the previous unit (lets us predict the next unit)

## Odds, log odds

- Odds of an event with probability  $p$  is defined by:  
 $\text{odds} = (1 - p) / p$
- The odds are more useful for discussing unlikely scenarios (999:1 odds makes more sense than  $p = 0.001$ )
- Log-odds or logit are particularly useful for very unlikely scenarios:  
 $\text{logit}(p) = \log(p / (1 - p))$
- The logit scales proportionally to the number of zeroes in the numerator of the odds
- Both odds and log odds are normally used to display results rather than to do computations
- But log probabilities are widely used for both computation and display as they help solve numerical problems in probability calculations

## Log probabilities

- The probability of multiple independent random variable taking on a set of values can be computed from the product:  
 $P(X, Y, Z) = P(X)P(Y)P(Z)$   
and in general:

$$P(X_1 = x_1, \dots, X_n = x_n) = \prod_{i=1}^n P(X_i = x_i)$$

- But computing these products could need multiplying a lot of values  $< 1$  (we get floating point underflow)

- Instead we use log probabilities which can be summed instead of multiplied to avoid underflow:

$$\log P(x_1, \dots, x_n) = \sum_{i=1}^n \log P(x_i)$$

- this uses the identity  $\log(AB) = \log(A) + \log(B)$
- The log-likelihood is just  $\log(P(B|A))$  and is often more convenient to work with than the raw likelihood
- We write  $L(x_i)$  to mean the likelihood of  $L(x_i)$
- The likelihood is not a probability but a function of data, and  $L(x_i) = f_X(x_i)$

## Comparing log-likelihoods (example)

- Can take the log likelihood of a text sequence under one PMF and subtract the log likelihood of another sequence under a different PMF
- If the value is positive (it is higher), then the first PMF is more likely to have generated the text, otherwise it is more likely the second one

## Bayes' Rule - inverting conditional distributions

- Sometimes we want to know  $P(A | B)$  but only have  $P(B | A)$
- In general  $P(A | B) \neq P(B | A)$
- Bayes' rule gives the correct way to invert the probability distribution:  

$$P(A | B) = (P(B | A) * P(A)) / P(B)$$

## Nomenclature

- $P(A | B)$  is called the posterior (what we want to know or will know after the computation)
- $P(B | A)$  is called the likelihood (how likely the event A is to produce the evidence we see)
- $P(A)$  is the prior (how likely the event A is regardless of evidence)
- $P(B)$  is the evidence (how likely the evidence B is regardless of the event)
- Bayes' rule gives a consistent rule to take some prior belief and combine it with observed data to estimate a new distribution which combines them
- Can substitute A with H (for hypothesis) and D (for data we observe) in the formula

## Integration over the evidence

- Can say the posterior probability is proportional to the product of the prior and the likelihood
- But to evaluate its value, we need to compute  $P(D)$  (the evidence)
- We can marginalise the  $P(D)$  from the joint distribution  $P(H, D)$ ; that is, integrating  $P(H, D)$  over every possible outcome of H for each possible D:

Because probabilities must add up to 1, we can write  $P(B)$  as:

$$P(D) = \sum_i P(D|H_i)P(H_i)$$

for a set of discrete outcomes  $A_i$  or

$$P(D) = \int_A P(D|H)P(H)dA$$

for a continuous distribution of outcomes.

- Generally difficult to compute, but for simple binary outcome cases, can write Bayes' rule as:

$$P(H = 1|D) = \frac{P(D|H = 1)P(H = 1)}{P(D|H = 1)P(H = 1) + P(D|H = 0)P(H = 0)}$$

## Natural frequency

- Means to explain the problem visually (fixed size population with proportions as counts) to prevent poor decision making

## Bayes' rule for combining evidence

- Bayes' rule is the correct way to combine prior belief and observation to update beliefs
- We always transform from one probability distribution (prior) to a new belief (posterior) using some observed evidence
- This can be used to "learn", where "learning" means updating a probability distribution based on observations
- Good for when uncertain info needs to be fused together (from multiple sources or over time)

## Entropy

- Key property of a probability distribution is entropy
- Measure of the "surprise" an observer would have when observing draws from the distribution
- It is the (log) measure of the number of distinct states a distribution could represent
- A flat, uniform distribution is very "surprising" because the values are very hard to predict
- A narrow, peaked distribution is unsurprising because the values are always very similar
- This is precise quantification - it gives the information in a distribution
- The units of information are normally bits (0/1 yes/no)
- The entropy tells you exactly how many bits are needed at minimum to communicate a value from a distribution to an observer who knows the distribution already
- Alternatively, can describe number of distinct states the distribution describes as  $p = 2^{H(X)}$  - this value is the perplexity, and can be fractional

## Entropy is just the expectation of log-probability

- The entropy of a (discrete) distribution of a random variable  $X$  can be computed as:

$$H(X) = \sum_x -P(X = x) \log_2(P(X = x))$$

- This is just the expected value of the log-probability of a random variable (the "average" log probability)

## Week 9: Probability II

[\[edit\]](#)

### Continuous random variables - problems with continuous variables

- Continuous random variables are defined by a PDF rather than PMF
- PDF has a number of complexities
- Probability of any specific value  $P(X = x)$  is 0 for every possible  $x$ , yet any value in the support of the distribution function (anywhere the PDF is non-zero) is possible
- No direct way to sample from the PDF in the same way as from the PMF (although there are ways to sample from continuous distributions)
- We cannot estimate the true PDF from simple observation counts as we did for the empirical distribution (we can never use these counts to "fill in" the PDF, as they only apply to a single value with zero "width")
- Can't do computations with continuous PDFs using Bayes' rule
- Simple discrete distributions have no concept of dimension - hard to represent the probability of a random variable taking on a vector value / distributions over other generalisations like matrices, complex numbers, quaternions, Riemannian manifolds

### Probability distribution functions

- The PDF  $f_X(x)$  of a random variable  $X$  maps a value  $x$  (which might be a real number, or a vector, or any continuous value) to a single number (the density at the point)
- It is a function (assuming a distribution over real vectors)  $\mathbb{R}^n \rightarrow \mathbb{R}^+$ , where  $\mathbb{R}^+$  is the positive real numbers, and

$$\int_x f_X(x) = 1$$

- The value of a PDF at any point is not a probability (because the probability of a continuous random variable taking on a specific value must be 0)

- Instead we state the probability of a continuous random variable  $X$  lying in a range  $(a, b)$ :

$$P(X \in (a, b)) = (a < X < b) = \int_a^b f_X(x)$$

## Support

- The support of a PDF is the domain it maps from where the density is non-zero ( $\text{supp}(x) = x$  such that  $f_X(x) > 0$ )
- Infinite support = non-zero value over an infinite domain (this is true of the normal distribution, but much more likely to be closer to the mean)
- Compact support = 0 density except from within a fixed interval
- Semi-infinite support = non-zero only after a fixed point

## Cumulative distribution functions

- The cumulative distribution function (CDF) of a real-valued random variable is:

$$F_X(x) = \int_{-\infty}^x f_X(x) = P(X \leq x)$$

- Unlike PDF, CDF always maps  $x$  to codomain  $[0, 1]$
- Gives how probability mass there is that is less than or equal to  $x$
- Can do stuff like:  $P(3 \leq X \leq 4) = F_X(4) - F_X(3)$

## Location and scale

- Normal distribution places the point of highest density to its centre  $\mu$  (the "mean") with a spread defined by  $\sigma^2$  (the "variance")
- This can be thought of as the location and scale of the density function respectively
- Location = where density is concentrated in PDF
- Scale = how spread out the density is

## Central limit theorem

- Consider the sum  $Y = X_1 + X_2 + X_3 + \dots$ ,
- If we form a sum of many random variables  $Y$ , then for almost any PDF that each random variable might have, the PDF of  $Y$  will be approximately normal ( $Y \sim N(\mu, \sigma)$ )

## Multivariate distributions: distributions over $\mathbb{R}^n$

- Continuous distributions generalise discrete variables (PMFs) (e.g. over  $\mathbb{Z}$ ) to continuous spaces over  $\mathbb{R}$  via PDFs
- Probability densities can be further generalised vector spaces, particularly to  $\mathbb{R}^n$  - extending PDFs to assign probability across an entire vector space, under the constraint that the (multidimensional integral)

$$\int_{\mathbf{x} \in \mathbb{R}^n} f_X(\mathbf{x}) = 1.$$

- This is the same as:

$$\int_{x_0=-\infty}^{x_0=\infty} \int_{x_1=-\infty}^{x_1=\infty} \dots \int_{x_n=-\infty}^{x_n=\infty} f_X([x_0, x_1, \dots, x_n]) dx_0 dx_1 \dots dx_n = 1.$$

- Distributions with PDFs over vector spaces are called multivariate distributions - in many respects, they work the same as univariate continuous distributions, but they typically require more parameters to specify their form as they can vary over more dimensions

## Multivariate uniform distribution

- Assigns equal density to a box in a vector space, such that integral through all  $x$  in the box = 1
- To sample from it, we sample independently from a one-dimensional uniform distribution in the range  $[0, 1]$ ,

to get each element of our vector sample (this is a draw from an n-dimensional uniform distribution in the unit box)

- We can make a transformed uniform distribution by transforming the box with a matrix A and shift by adding an offset vector b

## Multivariate normal distribution

- Mean vector  $\mu$ , covariance matrix  $\Sigma$  are parameters for sampling from a unit ball with independent normal distribution in each dimension
- Mean vector linearly translates mass (dist from centre), covariance matrix applies a linear transformation (scale, rotate, shear)

## Joint and marginal distributions

- Joint probability density function = density over all dimensions
- Marginal probability density function = density over some sub-selection of dimensions

## Monte Carlo

- To compute the expectation of a function of a random variable can often be hard for continuous random variables

$$\mathbb{E}[g(X)] = \int_x f_X(x)g(x) dx \approx \frac{1}{N} \sum_{i=1}^N g(x_i)$$

- This integral may be intractable but we can use the RHS here to get an approximate expectation
- This is because it is often very easy to compute  $g(x)$  for any possible  $x$  but intractable to compute for all
- If we can somehow sample from the distribution  $P(X = x)$ , this approximation works, and it gets better as  $N$  gets larger

## Inference - population and samples, statistics and parameters

- Inferential statistics = estimating the properties of an unobserved "global" population of values from a limited set of observed samples - assumes that samples are being drawn from some underlying distribution
- Population = unknown set of outcomes, can be infinite
- Parameter = term describing whole population (e.g. mean weight)
- Sample = some subset of population that is observed
- Statistic = function of the sample data (e.g. mean weight of sample)
- Distributions (types of rules) and parameters (specifics of rules applied) codify "unknown entity" that generates observed data according to definite but unknown rules - we assume model has some randomness or stochastic elements (for ease of representation or because it really does)
- Inference = process of determining the parameters by looking at samples/observations that we have

## Two worldviews

- Bayesian inference = we consider parameters to be random variables that we want to refine a distribution over, and that data is fixed (known, observed) - we talk about belief in different parameter settings
- Frequentist inference = we consider parameters to be fixed, but data to be random (we talk about approaching an accurate estimate with increased sample size)

## Three approaches to inference

- Direct estimate of parameters (define functions of observations that estimate the values of parameters of distributions directly) - requires assumption of the form of distribution, very efficient but only works for particular kinds of model, need estimator functions for each specific distribution we want to estimate
- Maximum likelihood estimation of parameters (use optimisation to find parameter settings until they appear as likely as possible) - keep tweaking parameters until they best align with known observations, requires

iterative optimisation, but works for any model where the distribution has a known likelihood function (i.e. we can compute how likely observations were to have been generated by that model)

- Bayesian, probabilistic approaches explicitly encode belief about the behaviour of the "mysterious entity" using probability distributions - in bayesian models, we assume a distribution over the parameters themselves, and consider the parameters to be random variables
  - we have initial hypotheses for the parameters (prior) and use observations to update this belief to hone our parameter estimate to a tighter distribution (posterior)
  - we don't estimate a single parameter setting, but always have a distribution over possible parameters that changes as data is observed
  - more robust and coherent way to do inference but harder to represent and compute
  - requires priors over both parameters and a likelihood function
- Identical distribution assumption = assume all observations drawn from same underlying distribution
- Independence assumption = assume all observations independent of each other

## Estimation

- No way of estimating PMF directly from observations like discrete distributions
- Continuous distributions need estimator functions (given observation set, estimate the most likely parameters of a PDF defining a distribution that might have generated the observations)
- Form of distribution (model) must be pre-decided, then specific parameters can be calculated under the assumption of this model

## Direct estimation

- Could do inference, if assuming a particular form of distribution, using estimators of parameters (e.g. mean, variance) of the population distribution
- Estimators computed via statistics

## Standard estimators

- Mean = you know what this is
- Sample mean  $\hat{\mu}$  is good unbiased estimator of true population mean  $\mu$  (which is the expectation for a random variable  $X$ )
- Mean and sample mean measure central tendency
- Sample variance is the squared difference of each value of a sequence from the mean of that sequence:

$$\hat{\sigma}^2 = \frac{1}{N} \sum_{i=1}^N (x_i - \hat{\mu})^2.$$

- It is an estimator of the population variance  $\mathbb{E}[(X - \mathbb{E}[X])^2]$
- Sample standard deviation is just square root of sample variance
- Sample std. dev and variance measure spread of data

## Maximum likelihood estimation: estimation by optimisation

- We can often compute the likelihood of an observation being generated by a specific underlying random distribution
- For a PDF the likelihood of a value  $x$  is just the value of the PDF at  $x$ :  $f_X(x)$
- The likelihood is a function of the data under the assumption of some particular parameters
- The (log)likelihood of many independent observations is the product of the individual (log)likelihoods
  - $\log \mathcal{L}(x_1, \dots, x_n) = \sum_i \log f_X(x_i)$
- If we don't know estimators for the parameters, we can estimate it with optimisation and the likelihood function
- We tweak the parameters of distribution  $\theta$ , to get the largest likelihood values when feeding the samples to the likelihood function:  $\mathcal{L}(\theta|x)$



- We could then define an objective function to maximise the log-likelihood (or minimise the negative log-likelihood):

$$\theta^* = \arg \min_{\theta} L(\theta)$$

$$L(\theta) = -\log \mathcal{L}(\theta|x_1, \dots, x_n) = -\sum_i \log f_X(x_i; \theta),$$

assuming our  $f_X(x_i)$  can be written as  $f(x; \theta)$  to represent the PDF of  $f$  with some specific choice of parameters given by  $\theta$ .

- Could use gradient descent if lucky and parameters are differentiable

## Bayesian inference

- We have belief in parameter settings and we want to infer a distribution over some plausible (based on the data) parameter settings
- We want to infer a posterior distribution over the parameters given some prior belief and some evidence - we assume we have a likelihood function  $P(D | \theta)$  and a prior over parameters  $P(\theta)$  and we can use Bayes' rule in the form:

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}$$

## Markov Chain Monte Carlo

- Can implement procedure to sample from the (relative) posterior distribution via simple modification of simulated annealing algorithm
- Randomly walks through space of parameter settings - proposing small random tweaks to the parameter settings and accepting "jumps" if they make the estimate more likely, or with a probability proportional to the change in  $P(D|\theta)P(\theta)$  if not
- The advantage of this approach is that we can work with definite samples from  $\theta$  and no difficult integration
- Just requires way of evaluating prior  $P(\theta)$  and likelihood  $P(D | \theta)$  for any specific  $\theta$

## Week 10: digital signals and time series

[\[edit\]](#)

### Time series and signals

- Signals can be thought of as functions of time  $x_t = x(t)$
- Need to sample them to make them amenable for digital signal processing - into a numerical array with a fixed number of values for  $t$  and  $x(t)$
- Quantisation = force samples into a discrete set of values e.g. [-128, 127]
- If you have a continuous signal you sample it at a regular interval to get the numerical array - this is time quantisation
- Each measurement of  $x(t)$  is itself quantised to a fixed set of values, so it can be represented as a fixed-length number in memory (e.g. int8, uint16, float64) - this is amplitude quantisation
- Capture continuous data into 1D ndarray or 2D ndarray if data is vector valued
- Can use index of sampling array to implicitly store the time
- Do need to store sampling rate  $f_s$ , - so the spacing between samples is  $1 / f_s$  often specified in Hz
- We sample signals as it is a compact, efficient way to represent an approximation to a continuously varying function of an array of numbers
- Saves storage space due to no explicit storage
- Removing offset from signal = subtract value elementwise from array
- Mixing two signals = (weighted) elementwise sum of array
- Correlation between signals can be computed with elementwise multiplication
- Selecting region of signals = slicing
- Smoothing and regression can be applied to signal arrays

## Noise

- All measurements have noise:  
 $x(t) = y(t) + \epsilon(t)$  where  $\epsilon(t)$  is a random fluctuation signal
- Signal to noise ratio =  $S / N$ , where  $S$  is  $y(t)$  amplitude and  $N$  is  $\epsilon(t)$  amplitude
- Can get it in decibels:

$$\text{SNR}_{dB} = 10 \log_{10} \left( \frac{S}{N} \right)$$

- Increase of 10dB means 10x louder
- $20 \log_{10}$  = showing SNR in terms of amplitude
- Can't remove noise directly by subtracting it - but we can assume how  $y(t)$  should look and separate out parts of the signal that could not be part of it (this is filtering)

## Sampling: amplitude quantisation

- Make continuously varying signal  $f(t)$  discrete by reducing it to a number of evenly spaced distinct values
- e.g. 6 bit quantisation =  $2^6 = 64$  levels
- Amplitude quantisation introduces noise to signals because the difference between the value of a signal and the quantisation levels is random
- Coarser quantisation = less storage space, less precise circuitry, lower memory bandwidth, less computation time
- Hardware which transforms analogue to digital (ADC) always has a limited quantisation ability (can hear in audio)
- Residual (difference) between a high amplitude resolution signal and low can be plotted

## Sampling theory

- If we sample a signal with enough points frequently enough and it doesn't have too much high frequency content we can perfectly reconstruct it

## Nyquist limit

- Given sampling rate  $f_s$  we can recover an original continuous time signal  $x(t)$  perfectly from a sampled signal if the signal contains frequencies at most  $f_n = f_s/2$

## Aliasing

- If we don't sample often enough and ignore Nyquist rule, we get aliasing
- Unpleasant artifacts where high-frequency elements are folded over
- If we sample a signal component with frequency  $f_q > f_n$  we will observe an artificial component at  $f_n - (f_q \bmod f_n)$
- This creates phantom behaviour in the sampled signals which doesn't correspond to any real-world changes in the continuous signal value
- In video we get the wagon wheel effect (if frequency of movement exceeds half frame rate, the direction reverses)
- When image is reduced in resolution, it must be filtered to remove high-frequency components that cannot be represented at a lower resolution, or serious aliasing occurs
- When the resolution of an image is reduced, the sample spacing gets larger, so the signal must have less high-frequency content to be represented accurately

## Filtering and smoothing - why filter?

- Filtering takes advantage of temporal structure (real signals cannot have arbitrary changes) - at least some portion is normally predictable so we can use it to make a filter (predictive model) to clean up signals

## A simple filter: moving averages

- We expect (true) signals to change slowly, and unwanted noise in components which change quickly
- We can apply this model with a moving average (use sliding window of samples and compute their mean) - slide window along by one sample and take mean of new window until we reach the end of the signal:

$$y[t] = \frac{1}{K} \sum_{i=0}^{K-1} x[t + i - K/2]$$

## Sliding window

- Sampled signal of unbounded length turned into collection of fixed length vectors
- Signal is broken up into equally spaced chunks or windows of a common length K
- We can then process the N x K data matrix (N windows of K samples)
- If we have vector valued measurements, N x K x D (D = channels for audio)
- Moving average filter has one parameter K (window size)
- New sampled signal has K fewer samples

## Using moving averages

- Can apply moving averages to sound - this lowpass filters the sound, reducing high frequency content
- The longer the moving average window, the smoother the waveform and the more high frequencies are suppressed
- Could subtract the moving average from the original signal, leaving only the fast changing parts and removing the slow-varying trend
- Can apply moving averages spatially to images, taking a moving average of rows and columns to blur the image (this is a box blur) - not accurate but easy to compute quickly
- Can take an average across video frames to do temporal blur

## Nonlinear filtering

- Moving averages are linear filters as their output is a weighted sum of previous inputs (moving average)
- A weighted sum is a linear operation, satisfying the conditions of linearity:
  - $f(x + y) = f(x) + f(y)$
  - $f(ax) = af(x)$
  - $f(0) = 0$

## Median filtering

- Any filter which is not a weighted sum is a nonlinear filter
- The most common nonlinear filter is the median filter (same as moving average but using median instead of mean), based on a world model where most measurements are good but a small minority are corrupted (perhaps severely)
- Median filter sometimes called order filter (works by sorting the data points)
- Median over mean is more robust with extreme values but median filter can be slow to compute (sort operations or specialised median cascade algorithms)

## Generalising moving averages - linear filters

- Many commonly used filters are linear filters
- Well developed theoretical background and their properties can be analysed in detail
- Linear filter is any filter where output is weighted sum of original value and neighbouring values
- No other functions can form part of a linear filter
- Linear filters often efficient as CPUs (especially DSP specific ones) often have multiply and accumulate instruction (multiply value by a constant and accumulate into a (high precision) register)
- Example weighted sum of three samples linear filter:

$$f(x[t]) = 0.25x[t - 1] + 0.5x[t] + 0.25x[t + 1]$$

## Convolution

- Taking weighted sums of neighbouring values is called convolution
- Convolution is denoted by a  $*$  between two functions  $f$  and  $g$
- $f * g$  is the convolution of  $f$  and  $g$
- Convolution is defined for continuous functions  $f(x)$  and  $g(x)$
- Convolution of two vectors/matrices is written in the same way:

$$(x * y)[n] = \sum_{m=-M}^M x[n-m]y[m]$$

for two sampled signal vectors  $x[n]$  and  $y[m]$  of length  $N$  and  $M$

- The convolution kernel is the operation to be performed
- If we do  $x * y$ ,  $x$  is the signal to be transformed,  $y$  is the convolution kernel
- Convolution is like a window sliding across a signal/image, summing everything below the window
- Window shape = weights applied to form the weighted sum, computed at each possible offset of that window
- Moving average is a simple convolution ( $N$  elements, each with value  $1/N$ )

## Algebraic properties and convolution

- The convolution operator commutes and associates:  
 $f * g = g * f$   
 $f * (g * h) = (f * g) * h$
- Therefore if we have two filters (convolution kernels), we can convolve them into a new kernel which stores and can apply to many signals
- Convolution also distributes over addition/subtraction:  
 $f * (g + h) = f * g + f * h$

## Simplest convolutions: Dirac delta functions

- Useful tool to analyse system responses
- It is a function that is 0 everywhere except 0 (where it is infinity)
- Area underneath the infinite spike is 1
- Satisfies:

$$\int_{-\infty}^{\infty} \delta(x) = 1$$

$$\delta(x) = \begin{cases} 0, & x \neq 0 \\ \infty, & x = 0 \end{cases}$$

## Convolutions with the delta function

- Key property of the delta function is that  $f(x) * \delta(x) = f(x)$   
i.e. convolution of any function with the delta function does not change the original function - it is an identity element with respect to convolution
- For discrete data, the discretised version of the delta function is just an array of zeros with a single 1 (an impulse)
- If we feed a perfect impulse into a system we want to model, we can recover the convolution kernel (linear system identification)

## Frequency domain

- Can view signals as a sequence of amplitude measurements over time (the time domain)
- Or as a sum of oscillations with different frequencies (the frequency domain)
- A pure oscillation is a sine wave:  
 $x(t) = A \sin(2\pi\omega t + \theta).$
- $\omega$  is the frequency of oscillation

- $\theta$  is the phase of oscillation (offset to time)
- $A$  is the magnitude of the oscillation
- Frequency = repetition period of sinusoidal oscillation
- Higher frequency = shorter period
- A pure frequency is a sine wave with a specific period

## Fourier transform - sine wave decomposition

- The Fourier theorem tells us that any repeating function can be decomposed into sine waves by approximating it as a sum of sinusoids:

$$x(t) = \sum_i A_i \sin(\omega_i 2\pi t + \theta_i)$$

- This works for literally every real periodic signal

## Correlation between signals

- This is the elementwise product between the two signals  $c$ :

$$c = \sum_i a[t]b[t]$$

- $c$  is the unnormalised correlation between two signals or just the inner product of their vector representations
- If the two signals are unrelated,  $c \approx 0$  because the positive and negative products are equally likely and cancel each other out on average
- If  $a[t]$  and  $b[t]$  are close,  $c$  will be large and positive
- If  $a[t]$  and  $b[t]$  are inverses (negatives) of each other,  $c$  will be large and negative

## Transform

- $c$  is the amplitude/magnitude of the response to a test signal when correlated with another signal
- Quadrature is comparing a sine wave  $a[t] = \sin(\omega x)$  and a cosine wave  $a'[t] = \cos(\omega x) = \sin(\omega x + \pi/2)$  to compute  $c(\omega)$  and  $c'(\omega)$  (the cosine wave version of  $c(\omega)$ ) and we can work out the phase directly from the correlation with their value
- We have two sinusoids 90 degrees or a quarter cycle out of phase with each other
- Phase is defined as the angle of the vector formed of these two values:

$$\theta = \tan^{-1} \left( \frac{c(\omega)}{c'(\omega)} \right)$$

- Magnitude ignoring the phase is found by:

$$A = \sqrt{c(\omega)^2 + (c'(\omega))^2}$$

## Fourier transform equation

- Allows us to write any (periodic) function  $f(x)$  as an (infinite) sum of sinusoids - simple waves with distinct frequencies, amplitudes, and phases:

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \omega} dx$$

the Fourier transform gives a complex value  $\hat{f}(\omega)$  for each possible frequency  $\omega$

- Can use Euler's identity for complex exponentials to see the complex values for each frequency
- For real signals, we can deal with only the real part of the signal, but we still have a complex output  $\hat{f}(\omega)$
- The Fourier transform "compares" a function with every possible frequency of sine and cosine wave and returns how much of that frequency is present and what "phase" the sinusoidal wave is in
- The sine part and the cosine part are returned as the real and imaginary components of this value:

$$\hat{f}(\omega)_{\text{real}} = \int_{-\infty}^{\infty} f(x) \cos(-2\pi x\omega) dx$$

$$\hat{f}(\omega)_{\text{imag}} = \int_{-\infty}^{\infty} f(x) \sin(-2\pi x\omega) dx$$

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(x) \sin(-2\pi x\omega) dx + i \int_{-\infty}^{\infty} f(x) \cos(-2\pi x\omega) dx$$

- The Fourier transform can be inverted back into exactly the original function:

$$f(x) = \int_{-\infty}^{\infty} \hat{f}(\omega) e^{2\pi i x \omega} d\omega$$

(inverse Fourier transform)

## Discrete Fourier Transform (DFT)

- Discrete version of the Fourier transform for a vector of discrete data  $[x_0, \dots, x_{N-1}]$ :

$$F[k] = \sum_{j=0}^{N-1} x[j] e^{-2\pi i \frac{j}{N} k}$$

$$= \sum_{j=0}^{N-1} \left[ x[j] \cos\left(-2\pi \frac{j}{N} k\right) + i x[j] \sin\left(-2\pi \frac{j}{N} k\right) \right]$$

for the  $k$  frequency components of  $x$  ( $k = 0, 1, \dots, N-1$ ). The DFT has as many frequency components as  $x[t]$  has elements.

## Complex components: phase and magnitude

- Result of the DFT is a sequence of complex numbers  $F[k] = a + bi$
- Can write any complex number as a magnitude  $A$  (length) and an angle  $\theta$  in an Argand diagram
- The angle  $\theta$  is the phase, the magnitude  $A$  is the component's magnitude, the frequency of the component is given by the index  $k$ , and depends on sampling rate
- $X_0 = 0$ ,  $X_{N/2} = f_n$  (the Nyquist rate, half the original sample rate)
- For the  $k$ th component:

$$\text{freq} = f_n k/N$$

## Spectra

- Magnitude spectrum of signal = magnitude of each component (i.e. as a function of frequency)
- Phase spectrum of signal = phase of each component
- Magnitude often plotted on a logarithmic scale (decibels) to make it easier to see the relative magnitudes of the components
- Phase spectrum often plotted on a wrapped scale (i.e. modulo  $2\pi$ ) to make it easier to see the phase relationships between components
- Spectra that have one large primary band where energy is concentrated = main lobe of the spectrum
- Side lobes = smaller bands of energy present at other frequencies
- Side lobe level = ratio of energy in side lobes to energy in main lobe
- Operations often introduce unwanted side-lobes (form of distortion)

## Reconstruction of a signal

- Can perform FFT, decompose into components, and reconstruct with the  $k$  most important (loudest) frequency components

## Fast Fourier Transform (FFT)

- The DFT is very expensive to compute ( $O(N^2)$ ) due to lots of floating point operations like exponentiation

- FFT is much faster, divide-and-conquer algorithm,  $O(N \log N)$  best case
- Only best case complexity if  $N$  is a power of 2 (because standard FFT splits the signal recursively into two) and is  $O(N^2)$  for non power-of-2 inputs
- Some FFT variations run  $O(N \log N)$  for vectors with length which is highly composite, but still  $O(N^2)$  for prime  $N$

## Convolution Theorem

- States that the Fourier transform of the convolution of two signals is equal to the (elementwise) product of the Fourier transform of two signals
- Writing  $FT(f(x))$  to mean the Fourier transform of  $f(x)$  and  $IFT(f(x))$  to mean the inverse Fourier transform of  $f(x)$ :

$$FT(f(x) * g(x)) = FT(f(x))FT(g(x))$$

- We can compute the convolution by taking products in the frequency/spatial frequency domain, and transforming back to the time or spatial domain:

$$f(x) * g(x) = IFT[FT(f(x))FT(g(x))]$$

## Frequency domain effects of filtering

- Time domain convolution clearly affects the frequency spatial domain (the convolution theorem tells us how)
- We can use this to analyse the effects of filters before we apply them to images
- We can apply any frequency-based filter in the frequency domain by multiplying, but having to take the DFT can be much slower and more memory intensive than a small convolution (opposite is true for large convolutions)
- Several types of filter
- Can have a smoothing/lowpass filter to reduce high frequencies
- Can have a highpass filter to reduce low frequencies
- Can have a bandpass filter to reduce frequencies outside a certain band
- Can have a notch filter or bandstop filter to reduce frequencies inside a certain band

## Linear filters in the frequency domain

- A linear filter (effect of convolution) can change the amplitude of frequency components, but can never introduce new frequencies
- If a frequency is not present in the signal, only non-linear features can introduce new frequencies
- Proof:  $FT(f * g) = FT(f)FT(g)$ , so no matter what we make  $g$ , if any element of  $FT(f)$  is zero, it will always stay zero.