# Lecture 0: Intro to Course

Structure:

* Tuesday
  + 2h lecture at 10:00-12:00
  + Lab (LB02 at 14:00-15:00)
  + Quiz due next Tuesday

# Lecture 1: Intro to Numpy

Vectorised operation – an operation applied to a whole matrix/sequence without iteration

ndarray (tensor)

* Multidimensional numerical array (of floating-point numbers)
* Very powerful, although simple in structure
* Why use them:
  + Object expression:
    - Images (2D arrays of brightness values), sounds (1D array of sound pressure levels), videos (3D array of brightness values and time)
    - Scientific data
    - 3D graphics
  + Abstraction and elegance
    - Easy to do vectorised operations
  + Mathematical power
    - Linear algebra provides tools to work with matrices and vectors
  + Efficiency
    - ndarrays are **compact** (memory efficient) and **computationally efficient**
  + Deep learning
* Types:
  + 1D array – vector
  + 2D array – matrix
  + 3D array – tensor
* Axes
  + Matrix has 2: rows (0) and columns (1)
  + Vector has 1: axis 0
* Properties:
  + fixed, predefined size
  + fixed, predefined type (of all elements)
  + only hold numbers (ints or float)
  + inherently multidimensional
  + required to be “rectangular”
* Operations
  + Map
    - Apply functions or arithmetic operations elementwise
  + Concatenate and repeat
  + Generate
  + Reorder
    - reverse/flip axes, sort axes, exchange rows/columns (transpose)
* NumPy
  + Every array is characterised by:
    - type of elements (dtype)
      * float64, float32, int32, uint8
    - shape (dimensions)
  + Order
    - Order of shape is: rows, columns, depth/frames/channels/planes/etc
  + Creating
    - Convert from another sequence type (list) (np.array())
    - Blank and filled with some values
      * np.empty(shape)
      * np.zeros(shape)
      * np.ones(shape)
      * np.full(shape, value)
      * \_like variants create blank arrays with same shape and dtype of another array (y = np.zeros\_like(x) )
    - Filled with random values
      * np.random.randint(a,b,shape) uniform ints between a and (excluding) b
      * np.random.uniform(a,b,shape) uniform floats between a and b
      * np.random.normal(mean,std,shape) distributed floats
    - Loaded from disk
      * np.load
      * Text files
        + np.loadtxt(filename)
        + np.savetxt(array, filename)
  + Ranges
    - arange
      * np.arange(start=0, end, step=1)
      * Step can be negative/fractional
    - Linspace
      * np.linspace(start, stop, steps)
      * Inclusive of stop
  + Slicing and indexing
    - Index
      * Using []
      * arr[0,1]
      * Indexing reduces the rank of an array (usually). It selects a rectangular subset where 1 dimension is a singleton, and removes that dimension
    - Slicing
      * start:end:stop
      * Slicing does not change the rank of an array. It selects a rectangular subset with the same number of dimensions

Transposition

* .T
* Does nothing to 1D, reverses order of dimensions in >2D arrays

Flip, rotate

* Single operation using indexing
* Left-right flip: x[:, ::-1]
* Up-down flip: x[::-1, :]
* Rotate 90 degrees: x.T[::-1, :] (transpose + up-down flip)

Concatenate and stack

* Concatenate – joins arrays along an **existing** dimension
* Stack – stacks arrays along a **new** dimension

Tiling

* Repeating arrays
* np.tile(array, tiles)

Map – application of a function to each element of a sequence

* Certain rules about what operations can be applied together (if more than two arguments, they must have compatible shapes (each element of x must be paired with an element of y):
  + Same shape
    - Elementwise array arithmetic (both sides of an operator have exactly the same shape)
  + Not the same shape
    - Scalar arithmetic (one side of the operator is a scalar, another array)
* Broadcasting
  + Allows arrays to be automatically expanded to allow operators between certain shapes of arrays
  + The way in which arithmetic operations are done on arrays when the operands are of different shapes

Selection and Masking

* Comparisons between arrays result in Boolean arrays

Fancy indexing

* Boolean indexing
  + Index with Boolean arrays
* Index with arrays of integers -len(arr) to len(arr)
  + Result is the values in those positions

Reduction (fold)

* Reducing an array to some final scalar
* Sum, mean, etc
* Can reduce in specific axes

Accumulation

* Stores intermediate results (like in running sum/prod)
* np.cumsum, np.cumprod
* np.diff gives one less dimension because of intermediate results
* np.gradient used in image processing

Finding

* Find indices that satisfy criteria
* np.argmax, np.argmin, np.argsort finds indices to sort in some way, np.nonzero

# Lecture 2: Floats; Arrays in Memory; Tensors

## Floating point numbers

* Algebraic properties of operators on real numbers (associativity, distributivity, commutativity) are **not** preserved with the representation of numbers used for computation
* Order of calculations **matters** (even if algebraically, they’re the same) because of round-off error

Number types

* Graphical user interface

  Description automatically generated with medium confidence
  + Default for np arrays: if ints, int32; if floats, float64
* Wrap-around after overflow is well-defined

Floating point representation

* A number in [1.0,2.0] and a shift
  + mantissa (fractional number with a standardised range) and exponent (scaling/stretching factor)
* Sign, exponent, mantissa
  + Scientific notation:
    - 5.4402 \* 10 ^ 3
    - [mantissa] \* 10 ^ [exponent]
  + Binary floating point
    - sign \* (1.[mantissa]) \* (2^exponent)
* Memory
  + 1 sign bit
  + 8 exponent bits
  + 23 mantissa bits
* Sign bit – 0 for positive, 1 for negative
* Standard – IEEE 754
  + Almost all floating point computations are either done in single precision (float32/float) or double precision (float64/double)
  + float32
    - 32 bits (4 bytes) per number
  + There are also floating-point decimals
* Integers in floats
  + For float64, ever integer from -2^53 to 2^53 is precisely representable
  + Javascript uses float64 for everything (there is not integer representation)

Float exceptions

* + Invalid Operation (undefined operation (0.0 /0.0) -> NaN)
  + Division by Zero (-> infinity)
  + Overflow (exceeds max limit -> infinity)
  + Underflow (-> 0)
  + Inexact (rounding) (-> closest number)
* Trapped or untrapped
  + Untrapped errors can be set to raise exceptions – np.seterr(under=”raise”)

Special numbers

* Zero – positive and negative
  + 0.0 == -0.0
  + Sign bit propagates (in multiplication/division)
* Infinity – pos or neg
  + np.inf
  + inf + 1 = inf
  + inf \* 2 = inf
  + inf \* 0 = NaN
  + inf – inf = NaN
* NaN – Not a Number
  + Invalid values:
    - 0 / 0, inf / inf, inf – inf, inf \* -inf, inf \* 0, 0 \* -inf, sqrt(x) if x<0
  + Properties:
    - Propagates
    - NaN is not equal to anything, including itself
    - Not equivalent to False in Python
      * Truthy in Python
  + np.isnan() is the same as var != var
  + Usage:
    - Mask in graph for null values

Roundoff error

* Order of

Laws of floating-point disaster

* Basic rules:
  + x+y will have large error if x and y have different magnitudes (magnitude error)
  + x – y will have large error if x ~= y (cancellation error)
* Don’t compare floats with ==
  + Including arrays
  + (unless testing for roundoff error)
  + Instead, compare with threshold:
* NumPy has Allclose
  + Compares relative and absolute epsilon tests

Machine precision and epsilon

* Relative error:
* IEEE 754 guarantees this error is always less than
  + where t is the number of bits dedicated to the mantissa, excluding the implied 1
* Varies depending on hardware

## Arrays

Memory

* Whatever the dimension of the array, it is stored in a dense block/sequence of numbers
* Strides and shape
  + Striding – using a set of memory offset constants (strides) which specify how to index into the array, one per axis
  + Ex:
    - shape = [3,5]
    - Strides: [8,40]
  + Strided 2D array:
    - array[i,j]
    - i \* stride[0] + j \* stride[1]
* Dope fiends
  + Dope vector
    - refers to the striding info, a header which specifies how to index
    - More efficient for parallel operations
  + Illife vectors
    - uses nested pointers to refer to multidimensional arrays
    - Can be used for more flexible shapes of an array
* C and Fortran order
  + Row-major/C ordering
    - columns, then rows (going through all of row first)
  + Column-major/Fortran ordering
    - rows, then columns (going through all of column first)

## Tensor operations

* Reshape
  + Different from np.fliplr() and np.transpose()
  + array.reshape()
  + Pouring (just metaphorically)
  + Changes dope vector (no ordering changed in memory, just header)
  + Advanced: squeezing and adding dimensions
    - Possible to do outer product of vectors by promoting their dimensions
    - np.newaxis() == None
    - Add dimensions with reshape()
    - Remove dimensions with indexing
    - Elided axes
      * Filling in appropriate dimensions as required
      * “…” in dimensions
  + Swapping and rearranging axes
    - Einstein Summation notation is a good generalisation
      * One-letter names for dimensions
      * ijk -> jik

# Lecture 3: Visualisation

Goal – Informative and Beautiful

* Conceptually correct – data is represented visually in a way that makes sense
* Technically correct – details of data representation are complete and accurate
* Aesthetically correct – data is represented in a visually pleasing way

Grammar of Graphics

Diagram

Description automatically generated

* **Stat** – statistic computed from data, which is then mapped onto visual features, with the intent of summarising the data compactly. Ex: mean, std, binning of values in a histogram
* **Mapping** – represents a transformation of data attributes to visual values. Maps *selected attributes* (which includes 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. Ex: Richter scale to x position, altitude to colour, condition number to point size. Typically specifies range of values to be mapped
  + **Guide** – visual reference which indicates the meaning of a mapping, including the scale and the attribute being mapped. Includes axis tick marks and labels, colour scales, legends.
* **Geom** – geometric representation of data after it has been mapped. Include points (which may have size, colour, shape), lines (colour, dash styles, thickness) and patches/polygons
* **Coord** – coordinate system, which connects mapped data onto point on the plane (in general, to higher-dimensional coordinates). The spatial configuration of geoms and guides depends on the coordinate system
* **Layer** – one set of geoms, with 1 mapping on 1 coordinate system. Multiple layers may be overlaid on a single coordinate system.
* **Facet** – different view of the same dataset, on a separate coordinate system. One facet might have several layers
* **Figure** – comprises a set of 1+ facets
* **Caption** – explains the visualisation to the reader

Basic 2D plots

* Dependent and Independent values
  + x – independent
  + y – dependent
* Scatterplot – points
* Bar chart
* Line plot
  + Continuous scale
* Combinations
  + Marking measurements with scatterplot and line plot
* Rhythm plot
  + Range, uncertainties

Layers vs Facets

* Layers have same coordinate systems
* Facets – different views of the same data

Stats

* Statistic of a Dataset (function of the data)
* Summarise data in some way
* Ex:
  + **aggregate summary statistics**: measures of central tendency (mean, median) and deviation (standard deviation, max/min, interquartile range)
    - Box plot
    - Violin plot
      * Uses a smoothing technique called kerning (???)
  + **binning operations**, which categorise data into a number of discrete bins and count the data points falling into those bins
    - Histograms
      * Combination of a binning operation with a standard 2D bar chart
  + **smoothing and regression**, which find approximate functions to datasets, like linear regression which fits a line through the data

Geoms

* Markers
  + Shape
  + Scalar attributes
    - Size
* Colour choices
  + Unsigned scalar
    - Monotonically varying brightness (as attribute increases, colour map should get consistently lighter/darker). viridis, magma – perceptually uniform
* Lines
  + Only if there are continuous data points
  + Staircase lines an option if there is no continuous data
* Transparency

Coordinate systems

* Size of figure matters
* Aspect ratio
* Log-log plot
  + Good for polynomial (power law) relationships
* Polar
  + Possible to do circular graph for theta and r

Uncertainty

* Error bars
* Box plots
* Violin bars

# Lecture 4: Vector Spaces and Matrices

Computational Linear Algebra

Vector space

* Mathematical space with 2 operations:
  + scalar multiplication
  + vector addition
* 2 additional operations
  + norm (length of vectors)
  + inner product (angle between vectors)

Apps:

* Geometry
* Machine Learning

Basic vector operations

* Addition and multiplication with weighted sums
* Linear interpolation
  + Line between 2 vectors
* Normalisation
  + Pushes to a square

Inner product:

High-dimensional vector spaces

* Curse of observability
  + As dimensionality increases (more observations), data becomes sparser (because there’s limited data)

Matrices and linear operators

* Uses of matrices:
  + Represent operations that do things to vectors (points in space)
* Maps
  + Functions applied to vectors which output vectors
  + Matrix transform
    - n x m matrix represents a function taking m to n dimensional vectors such that all straight lines remain straight and all parallel lines remain parallel, and the origin doesn’t move (**linearity**)
  + Matrices represent linear maps/functions
  + Applying same matrix/function twice+ is the same as applying once – **linear projection**
  + Every linear map of real vectors can be represented as a matrix
* Operations:
  + Addition and subtraction
  + Scalation
  + Transposition
  + Application to vectors
    - Form a weighted sum of the elements of the vector
  + Multiplication
    - Equivalent to composing the linear functions they represent, and it results in a matrix which has that effect
* Vector operations with transposition
  + (outer product)
  + (inner product)
* Commutativity
* Covariance matrix
  + Measures the spread of a dataset, how the data covaries (relates)
  + Variance – square of standard deviation, measures spread
  + Covariance (error) ellipse
    - Covariance matrix captures **correlations between dimensions**
    - Correlation expressed as ellipse
    - Centre – mean vector
    - Shape – covariance matrix
* Anatomy
  + Diagonal (a11, a22, …, ann)
    - Diagonal matrix – pure scaling, no rotation
  + Off-diagonal
* Unique matrices:
  + Identity matrix (diagonal is all 1s, other elements are 0)
    - IA = A = AI and Ix=x
    - => No effect
  + Zero matrix
    - Matches everything to zero vector
  + Square
    - Equal number of rows and columns
    - Only ones that have:
      * an inverse
      * determinants
      * eigendecomposition
  + Triangular
    - Square and has non-zero elements only diagonal and:
      * above – Upper triangular
      * below – Lower triangular

# Lecture 5: Computational Linear Algebra

Adjacency matrix

* Square matrix A whose elements are all 0, except where there is an edge from Vi to Vj, in which case
* Properties:
  + Out-degree of each vertex (no. of edges leaving vertex) – sum of each row
  + In-degree of each vertex (no. of edges entering vertex) – sum of each column
  + If matrix symmetric (equal to transpose) => undirected graph
  + Directed graph A can be converted to undirected graph A’ = A + A^T
  + If there are non-zero elements on diagonal, there are edges connecting vertices to themselves (self-transitions)
* Graphs
  + Flows of “mass”
    - if total flow out of vertex > 1 (sum(row) > 1), **source of mass**
    - if total flow out of vertex is < 1 (sum(row) < 1), **sink**
* Conserving adjacency matrix
  + sum(each row) = 1

A=-A^T – skew-symmetric

Matrix operations

* Exponentiation
  + A^3 = AAA
  + Repeats effect of matrix
* Inversion
  + Undoes effect of matrix
* Finding eigenvalues
* Factorisation (SVD)
* Measuring properties (determinant, trace, condition number)

## Eigenvalues and eigenvectors

Eigenvectors

* vectors that don’t get rotated when multiplied by the matrix, they only get scaled (stretched/compressed)

Eigenvalue

* The scaling factor of the eigenvector

**Leading eigenvector** found by the **power iteration**:

* Fixed vector (possibly with signed flips)
* Largest eigenvalue
  + Lambda only by reordering equation

Numpy

* np.linal.eig(matrix) returns tuple of eigenvalues, eigenvectors

Formal definition:

* where
* Each xi is the eigenvector and each lambda i is the eigenvalue

Eigen spectrum: sequence of eigenvalues **in order of absolute value**:

Principal Component Analysis (PCA)

* Principal components – eigenvectors of the covariance matrix
  + Show directions in which the data varies most

Trace

* Sum of diagonal values:
* (**sum** of eigenvalues)
* Measuring the **perimeter** of the parallelotope of a unit cube transformed by the matrix

Determinant: **product** of eigenvalues:

* If det(A)=0, A is **singular** and cannot be inverted (information has been lost)
* The **volume** of the parallelotope of a unit cube transformed by the matrix

Definiteness:

* Positive definite – all eigenvalues are positive
* Positive semi-definite – all eigenvalues are non-negative
* (vice versa, but not encountered in practice)

Inverse:

* Definitions:
* np.linalg.inv(matrix)
* Only for square matrices
* Singularity
  + Singular matrices (det(A)=0) have no inverse
  + Non-singular matrices are invertible
* Numerical stability
  + Avoid inverting whenever possible – extremely numerically unstable
  + Use special cases:
    - Orthogonal matrix (rows and coluns are all orthogonal unit vectors)
      * O(1) time
    - Diagonal matrix (all non-diagonal elements are 0)
      * (reciprocal of diagonal elements of A)
      * O(n) time
    - Positive-definite matrix
      * …
    - … (doesn’t matter)
* Inverse of sparse matrix is in general not sparse
  + Avoid inverting
* Benefit:
  + Solving linear systems

**SVD (Singular value decomposition)**

* Any matrix
  + A – m x n
  + U – **square unitary** m x m matrix, called the **left singular vectors**
  + – diagonal m x n matrix, whose diagonal is the **singular values**
  + V – **square unitary** n x n matrix, whose columns contain the **right singular vectors**
* Unitary: transpose = inverse. If A is real, U and V are orthogonal
* Relation to eigendecomposition:
  + taking the eigenvectors of to get U
  + taking the square root of the absolute value of the eigenvalues of to get
  + taking the eigenvectors of to get
* Rotation -> scale -> rotation
* np.linalg.svd(matrix) returns u, sigma, vt
* Relation to inversion:
  + Pseudo-inversion
    - (for non-square matrix A)
    - Fitting lines/planes
* Apps:
  + Finding low-rank approximations to a matrix
  + Raise a matrix to any power (in a single operation), provided it’s a square symmetric matrix

Rank, condition number

* Rank – number of non-zero singular values (elements of )
  + If rank(A) = n, then A is full-rank
    - Non-zero determinant
    - Invertible
  + If rank(A) < n, then A is singular/rank-deficient
    - If rank(A) << n, then A is low rank
* Condition number – ratio of the smallest to largest singular value
  + Only defined for full rank matrices
  + cond(A) – small => well-conditioned
    - Unlikely to cause numerical issues
  + cond(A) – large => ill-conditioned
    - Inversion will be numerically inaccurate

Singularity

* Singular matrix A => uninvertible, det(A)=0
* Singularity is a binary property
* Rank – “how singular” the matrix is, i.e., how many dimensions are lost in the transform
* Condition number – how close a non-singular matrix is to being singular. A matrix which is nearly singular may become effectively singular due to floating point roundoff errors

Whitening

* Destroys all linear correlations in a dataset
* **Normalisation**
* Removes mean and “divides out” the covariance
* If N x D data matrix X, then it’s possible to whiten (spherify) it by computing:
  + – mean vector (row vector containing the mean of each column in X)
  + – **covariance** matrix (not matrix of singular values)
* How:
  + Centres the data around its mean => **0 mean**
  + “Squashes” the data so that its distribution is spherical and has **unit covariance**

# Lecture 6: Introduction to Optimisation

Optimisation equation:

* - solution
* – **parameters** (packed into a vector)
* – objective function (maps any to a single number)
* – constraints, defined region where valid lie

Parameters

* Things possible to adjust (scalar/vector)
* Parameter space () – set of all possible configurations of parameters
  + Often a vector space
  + Parameter vector

Objective function (loss function, fitness function, utility function, energy surface)

* Returns 1 number (single numerical measure) saying how good a particular configuration is
* Smaller values – better
* Convex objective functions have just 1 minimum; non-convex have many minima
* Want to deal with continuous and smooth objective functions

Constraints

* Limitations on the parameters
* Define a region of the parameter space that is feasible – **the feasible set/region**
* Approximate constraints by penalising – adding a term to the objective function that makes it larger when the constraints are violated
* **Equality** constraint:
  + – function that represents the constraints
  + Constraining parameters to a **surface** (trade-off)
  + Might be using when trading off items where the total value must remain unchanged
* **Inequality** constraint:
  + – function that represents the constraints
  + Constraining parameters to a **volume** (bounds on values)
* Common constraint types:
  + Box
    - Requirement that lie within a box inside
    - Inequality constraint
  + Convex
    - Collection of inequalities on a convex sum of the parameters
    - Equivalent to the feasible set being limited by the intersection of **planes/hyperplanes** (even infinite)
* Unconstrained
  + Often not useful
* Constrained optimisation:
  + Pros:
    - Guarantees that solution will satisfy constraints
    - May be able to use constraints to speed up optimisation
  + Cons:
    - May be less efficient that unconstrained
    - Fewer algorithms available
    - May be hard to specify feasible region
  + Soft constraints
    - Apply penalties to “discourage” solutions that violate the constraints
    - Optimiser stays the same, but objective function modified to:
      * – **penalty function** (bigger violation of constraints => bigger value)
      * Pros:
        + Any optimiser can be used
        + Can deal with *soft* constraints sensibly
      * Cons:
        + May not respect important constraints (especially if they’re sharp)
        + Can be hard to formulate constraints as penalties
        + Cannot take advantage of efficient search in constrained regions of space

Approximation formulation:

* Objective function measures, for a given input x, how different a predicted output y’ is from the real y, where f is a function of x, whose behaviour is controlled by
* Evaluating objective function may be expensive (long/dangerous/costly/hard), so important to have mathematical structure to guide the search

Types of optimisation:

* Discrete
  + Parameters are discrete
* Continuous
  + Parameters are in a continuous space (typically )
  + Usually easier because of ability to exploit **smoothness** and **continuity**

Properties of optimisation:

* Parameters (always)
* Objective function (always)
* Constraints (usually)

Properties of objective function

* Local minimum
  + Point at which objective function increases in every direction
* **Convexity**
  + **Convex** – if it has a single, **global minimum**
  + Implication – finding any minimum is finding best one
  + If it’s convex and any constraints form convex portions of the search space => convex optimisation
    - Efficient methods:
      * Linear programming (if constraints and objective function are linear)
      * Quadratic programming (if quadratic objective function and linear constraints)
      * Specialised: semi-quadratic programming, quadratically constrained quadratic program
  + Non-convex problems require use of **iterative** methods
* Continuity
  + Continuous – if, for some very small adjustments to , there is an *arbitrarily*small change in
  + “No surprises”

Hyperparameters

* Properties which affect the **way** in which the optimiser finds a solution
* Not parameters of the objective function

Algorithms

* Direct convex optimisation: **least squares**
  + Solves objective functions of the form:
  + Finds x closes to solution Ax=y by minimising the squared L2 norm
  + Convex equation (quadratic function, must have 0/1 minimum)
  + Line fitting
    - Finding gradient m and offset c for line equation
* Iterative:
  + Worst – **grid search**:
    - try systematic combinations of parameters (in a grid pattern) and return the best result.
    - Pros:
      * Works for any continuous parameter space
      * Requires no knowledge of the objective function
      * Trivial to implement
    - Cons:
      * **Incredibly** inefficient
      * Must specify search space bounds in advance
      * Highly biased to finding things near the “early corners” of the space
      * Depends heavily on number of divisions chose
      * Sensitive to the curse of dimensionality
      * Need to know how continuous function is
  + Simple stochastic: **Random search**:
    - guess parameters and return best
    - Pros:
      * Cannot get trapped in local minima
      * Requires no knowledge of the structure of the objective function (not even topology)
      * Very simple to implement
      * Better that grid search (almost always)
    - Cons:
      * **Extremely inefficient** (only appropriate if no other mathematical structure to exploit)
      * Must be possible to randomly sample from parameter space (usually easy)
      * Results do not necessarily get better over time – completely random
    - Slow, but better than grid search
  + Using **Metaheuristics**
    - Locality – local search
      * Objective function is likely to have similar values for similar parameter configurations
      * Assumes continuity
      * Making **incremental** changes to a solution
      * Cons:
        + Can get trapped in local minima
        + => depends on initial conditions
      * Hill climbing
        + Uses concept of **neighbourhood** of a parameter vector
        + Makes incremental adjustments inside that neighbourhood, keeps transition if it improves the loss
        + Stochastic – random adjustment
        + 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 local minima

Struggles with objective function regions that are relatively flat

Requires that the objective function be (approximately) continuous

* + - * + Tweaks:

Adaptive local search (changes size of neighbourhood)

Multiple restarts (try to avoid getting stuck)

* + - Temperature
      * Changes the rate of movement in the parameter space over the course of an optimisation
      * Assumes existence of local optima
    - Population
      * Tracks multiple simultaneous parameter configurations and selects/mixes among them
      * Analogue to **evolution**
      * Involves:
        + **Mutation** (introducing random variation)
        + **Natural selection** (solution selection)
        + **Breeding** (interchange between solutions)
      * **Genetic algorithms**
        + Parameter set – **genotype** of a solution
        + Crossover (breeding) rules
        + 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 parametrisation: mutation schemes, crossover schemes, fitness functions, selection functions, etc

* + - * + Cons:

Many hyper-parameters to tune which radically affect the performance of the optimisation

No guarantee of convergence

(Very) slow compared to using stronger knowledge of the objective function

Many evaluations of objective function required: one per population member per iteration

* + - Memory
      * Records good/bad steps in the past and avoids revisiting them
      * Ant colony optimisation
        + Combines population and memory (paths between possible minima)
        + Stigmergy

Mechanism of spontaneous, indirect coordination between agents/actions, where the trace left in the environment by an action stimulates the performance of subsequent action

Converge

* For convex, reaching global minimum. For non-convex, reaching local minimum

Tuning optimisaation

* Use right algorithm
  + Problem is least-squares -> use least-squares solver
  + Convex problem -> use convex solver
  + Know derivates of objective function -> first-order method
  + No knowledge -> zeroth-order solver (simulated annealing/genetic algorithm)
* What can go wrong?
  + Slow progress
    - Increase size of step
  + Noisy/diverging progress
    - Decrease step size

# Lecture 7: Optimisation (Gradient Descent)

Problem: find an approximating function with parameters such that

Neural networks consist of **layers**

* linear maps (matrix multiplications) followed by simple, fixed non-linear functions
* Output of previous layer is input for next
* The linear map/matrix – **weight matrix**
  + G(x) – non-linear function

Uses:

* derivative of the objective function with respect to the weights.
* Algorithm called **backpropagation** (algorithm for **automatic differentiation**)

Gradient of a function at a particular input is a matrix (Jacobian), which tells you how much each output changes for an infinitesimal change in the inputs

First-order – takes first derivatives

**Jacobian**: matrix of derivatives (J)

* Characterises slope *at a specific point* x. If input and output , then we have an **m x n** matrix

**Gradient vector**: one row of the Jacobian

* – gradient vector of a (scalar) function of a vector
* One partial derivative per component of x (how much f(x) would vary if we made tiny changes to each dimension *independently*)
* Usually work with

**Hessian**: Jacobian of the gradient vector

* Second derivative for vector functions

Iterative algorithm classifications:

* **Zeroth order** – only evaluation of the objective function
* **First order** – evaluation of and its derivative
  + ex: gradient descent
* **Second order** – evaluation of , , and
  + ex: quasi-Newtonian optimisation

Conditions

* Differentiability
  + Smooth function has continuous derivatives up to some order
  + **continuous** – its nth derivative is continuous
  + First order optimisations can be used only if objective function is:
    - at least continuous
    - differentiable (gradient defined everywhere)
* Lipschitz continuity
  + First-order (and higher-order) continuous optimisation algorithms put a stronger requirement on functions
  + Equivalent to saying that the gradient is bounded and the function cannot change more quickly than some constant; there is a maximum steepness.
    - for all i and some constant K (Lipschitz constant)
    - Measure of how wide the cone that only touches the function once is
    - Measure of how smooth the function is, or equivalently, the **maximum steepness** of the objective function at any point.
    - supremum – smallest value that is larger than every value of the function

Analytical derivatives

* [mathematical]

Computable exact derivatives

Gradient: A derivative vector

Gradient descent:

* – step size (hyperparameter)
* Algorithm:
  + Starting somewhere with theta0
  + Repeat:
    - Check how steep the ground is in each direction (Nabla)
    - Move a little step delta in the steepest direction (Nabla) to find theta(i+1)

Step size should be proportional to Lipschitz constant

Gradient descent is:

* Delta L(theta) – steepest direction of the objective function
* If we go in that direction
* Conditions:
  + L(theta) is continuous, has at least 1 continuous derivative, and is Lipschitz continuous
  + Need to be able to find
* Properties:
  + Sensitive to step size (hyperparameter)
  + Can get stuck in local minima, only going to find global minimum for convex objective functions

Differentiation of the objective function

* Numerical
  + Massive problems with stability (floating point errors in formula of first derivatives)
  + Bad in high dimensions
* Symbolic
  + Closed form derivative
  + Gives value to any point
  + e.g., sin(x) to cos(x)
* Automatic
  + Gives value to a single point in a function
  + Works for matrices and vectors
  + 2 types:
    - forward
      * one column of the Jacobian
    - backward
      * one row of the Jacobian
      * useful for many inputs and few outputs, as in approximation

Improving gradient descent

* Automatic differentiation
  + Helps with computing gradient of loss function at any point
  + Differentiable programming
    - ex: autograd, JAX, pytorch, tensorflow
    - Automatically differentiates vectorised code, producing exact derivatives of tensor algorithms
  + Autograd
  + Limits:
    - Not all functions are differentiable
      * But can go around that (stochastic relaxation)
* Stochastic relaxation
  + Average over many random instances, where the conditions are slight different
* **Stochastic gradient descent (SGD)**
  + If problem is a sum of smaller problems
  + Choose some random subset, get the gradient of the sum for that subset, move in that direction
  + Subset – **minibatch**
  + One run through the whole dataset – **epoch**
  + 2 effects:
    - Needs less memory (usually)
    - Smooths out optimisation by adding noise
* Random restart
  + Start in different points
* Momentum
  + Keep moving in a good direction, even if the gradient changes a bit (like a heavy ball rolling)
* Others (line search, adaptive models (ADAM, AdaMax)

Calculus to know:

* <https://explained.ai/matrix-calculus/index.html>
* Derivative def:

Critical points

* Points where derivative goes to 0
* types:
  + Maxima
    - Gradient decreases in all dimensions
  + Minima
    - Gradient increases in all dimensions
  + Saddle points
    - Gradient has different signs in different dimensions (like potato crisp)
  + Plateau
    - Gradient 0 in all dimensions
  + Ridge
    - Gradient 0 in 1 dimension
* Matrix of second derivatives (Hessian), its eigendecomposition
  + For every parameter theta i, it says how steepness changes for every other component theta j
* Eigenvalues:
  + All eigenvalues are positive => **positive definite** (minimum)
  + All eigenvalues are negative => **negative definite** (maximum)
  + Mixed sign => saddle point
* All positive/negative but with some 0s => **semidefinite** (plateau/ridge)

Second-order optimisation

* Moves much more quickly through saddle points and plateaus
* Curse of dimensionality

# Lecture 8: Probability and Random Variables

Terms:

* Experiment (trial) – an occurrence with an uncertain outcome
  + ex: losing submarine (location unknown)
* Outcome – result of an experiment; a state of the world
  + ex: submarine is in [2,3]
* Sample space – set of **all possible** outcomes for an experiment
  + [1,1], [1,2], …
* Event – subset of possible outcomes with some common property
  + ex: die landing on 6
* Probability (of an event with respect to sample space) – *no. of outcomes in event*divided by *total no. of outcomes in sample space*
* Probability distribution – mapping of outcomes to probabilities (sum = 1). A random variable has a probability distribution which maps each outcome to a probability
  + ex: **P(X=x)**, probability that the submarine is in x (X – random variable)
* Random variable – represents unknown value, whose probability distribution is known
* Probability density/mass function – defines a probability distribution by mapping each outcome to a probability
* Observation – outcome directly observed, i.e., data
  + ex: submarine found in [1,5]
* Sample – an outcome simulated according to a probability distribution (**drawing samples**)
* Expectation/expected value – “average” value of a random variable

Philosophy of probability

* Bayesian
  + Calculus of belief
  + Probabilities – measures of degrees of belief
  + Key process – updating of beliefs
    - Prior belief
    - New evidence
    - Update of beliefs to calculate posterior
  + Parameters of population considered to be random variables, data to be known
  + “Subjective”
* Frequentist
  + Probabilities – long-term behaviour of repeated events
  + No priors
  + Parameters of population assumed to be fixed, data to be random
  + “Objective”

Generative models: forward and inverse probability

* **Generative process** – unknown process going on, the results of which can be observed
  + **Infer** how it’s governed
* **Forward probability** question – asks relating to the distribution of the observations (in the future)
  + “What is the probability that the next ball drawn will be white?”
* **Inverse probability** question – asks relating to unobserved variables that govern the process that generated the observations
  + “What is the distribution of white and black balls in the urn?”

Axioms of probability:

* **Boundedness**
  + all possible events A – probabilities are 0, or positive and less than or equal to 1
* **Unitarity**
  + for the complete set of possible outcomes (not events!) (sample space) – something always happens
* **Sum Rule**
  + probability of either A or B happening – sum of the independent probabilities minus probability of both happening
* **Conditional probability**
  + Probability that event A will happen **given we know B has happened**

Distributions

* Probability distribution – defines how likely different states of a random variable are
* Probability mass function:
  + for N observations x0, x1, …, xN
* Random variable types:
  + Discrete
    - distribution described by **probability mass function (PMF)**, which gives each outcome a specific value. Usually written , where
  + Continuous
    - **Probability density function (PDF)**, which specifies the spread of the probability over outcomes as a continuous function .

Expected value (outcome got on average if chosen random x)

* If a random variable takes on numerical values:
* Discrete RVs:
  + sum over x’s, where
* Expectation and means
  + expected value – true average of value of all outcomes observed if ran infinitely, i.e., population mean – the mean of the whole population
  + Mean of X is E[X] – **central tendency**
  + Variance of X is – **spread**
* Expectation of functions of X
  + sum over x’s, for discrete RVs, where
  + Careful:
* Decision theory
  + Combining scores (utility) with uncertainty (probability)

Samples

* Empirical distribution – estimating PMF that might be generating observations by counting each outcome seen divided by the total number of trials
  + i.e., normalised histogram of counts of occurrences of outcomes
  + – no. of times outcome x was observed
  + N – total no. of trials
* Random sampling procedures
  + Uniform sampling
    - Equal probability of any number in a range
    - Pseudo-random numbers
      * X – RV between a and b
      * U – equal possibility (uniform)
      * ~ - “distributed as”
  + Discrete sampling
    - Algorithm:
      * Choose any arbitrary ordering of all outcomes
      * Assign each outcome a “bin” (portion of interval [0,1] equal to its probability, so that interval is divided into consecutive non-overlapping regions []
      * Draw a uniform sample in the range [0,1]
      * Whichever “outcome bin” it lands in is the sample to draw

Distributions:

* Joint – seeing events happening simultaneously
  + probability that X and Y take the specific values simultaneously, i.e.,
  + 2 RVs are **independent** if they do not have any dependence on each other
* Marginal – calculating one outcome by summing from joint matrix
  + - PDF
    - PMF
  + Marginalisation – integration over 1+ variables from a joint distribution (removes those variables
* Conditional probability
* [example with Bigrams in lecture]

Representation and display of probability

* Odds
  + probability p
  + ex: 999:1 with p = 0.001
* Log-odds (logit)
  + scales proportionally to the number of 0s in the numerator of the odds
* Joint probability of multiple independent outcomes:
  + Problem: floating point underflow
  + Solution: log probabilities:
    - Uses identity log(AB) = log(A) + log(B)
* Likelihood
  + Function of data
  + Probability of an observation given the state of the model
  + For joint distributions:
    - But: underflow
    - Solution: log-likelihood

Estimation:

Entropy

* How “spread out” or “surprising” an outcome is

Bayes’ Rule:

* + 𝑃(𝐴|𝐵) (**posterior**) - what we want to know, or will know after the computation
  + 𝑃(𝐵|𝐴) (**likelihood**) -- how likely the event 𝐴 is to produce the evidence we see
  + 𝑃(𝐴) (**prior**) - how likely the event 𝐴 is regardless of evidence
  + 𝑃(𝐵) (**evidence**) - how likely the evidence 𝐵 is regardless of the event.
* Hypothesis H and data D:
* Natural frequency
* Bayes’ rule for combining evidence

Entropy

* Measure of the “surprise” an observer would have when observing draws from the distribution
  + expected value of the log-probability of an RV (“average” log-probability)

Distribution with **higher entropy** has outcomes that are **less predictable**.

# Lecture 9: Sampling and Inference

Problems with **continuous** random variables:

* Any specific P(X=x) = 0 for every possible x, yet any value in the *support* of the distribution function (where PDF is non-zero) is possible
* No direct way to sample (tricks needed)
* Cannot estimate true PDF from counts of observations
* Simple discrete distributions don’t have a concept of dimension

Probability density function – assigns density to a space

* Probability to do with lying in a region of this density function
* Support – domain it maps from where density is non-zero
  + Infinite support – support is -inf to inf
  + Compact support – -1.0 to 1.0
  + Semi-infinite support – 1.0 to inf
* CDF (Cumulative distribution function)
  + Always maps x to the codomain [0,1]
  + For a given value F\_X(x) tells how much probability mass there is that is less than or equal to x

PDF ex: The Normal (Gaussian) Distribution

* Assigns probabilities to real values x
* Density given by
  + – centre
  + (Inverse squared exponential function)
  + Z – [doesn’t matter]
* In general, distribution of continuous random variables:
* For Normal distribution:
  + mean
  + variance
* Location and scale
  + - Centre
    - The mean
    - Spread
    - The variance
* Benefits
  + Normal variables have many good mathematical properties -> easy to work with analytically
  + **Central Limit Theorem**
    - Any sum of random variables (however distributed) will tend to *Levy stable distribution* (e.g., normal distribution) as the number of variables increases

Multivariate distributions: distributions over

* Distributed over a vector space
* Continuous distributions generalise discrete variables (PMFs) to continuous spaces over via PDFs
* Distributions with PDFs over vector spaces – **multivariate distributions**
* Multivariate uniform distribution
  + Box with straight borders
  + Transformed
    - Over any box
    - Transformed vectors with matrix A
    - Shifted vectors with offset vector **b**
* Multivariate Normal distribution
  + Mean vector
  + Covariance matrix
  + Generalisation
    - A and **b**

Inference:

* In general
  + Population
    - parameter
      * True mean
    - Unknown set of outcomes (might be infinite)
  + Sample
    - statistic
      * Sample mean
  + Subset of population that has been observed
* Types:
  + Direct estimation
    - Look at observations, algorithmically adjust parameters, adjust process to match directly
    - Estimator functions
  + Maximum Likelihood
    - Compute likelihood of each observation, Tweak knobs to maximise likelihood, Optimise likelihood (usually negative log likelihood is easier)
    - Typically, gradient descent (if possible)
  + Bayesian Inference
    - From guesses over possible knob states, Update guesses based on likelihood, Result in **distributions** over knob settings
    - Bayes’ Rule
    - From prior to posterior
* Worldviews:
  + Bayesian inference
    - Consider parameters to be random variables, data - fixed
    - Belief in particular parameter settings
  + Frequentist inference
    - Consider parameters to be fixed, data - random
    - How to accurately estimate true parameters with increased sample size

Linear Regression

* Fitting of a line to observed data
* 1D:
* ND:
* Noisy data:
  + – noise term
* Assumption:

Direct estimation

* Estimators
  + ex: mean, variance
  + Computed via statistics (summarising functions that can be applied to data)
  + Mean
  + Variance
    - Squared difference of each value of a sequence from the mean of that sequence
    - Standard deviation – square root of variance
* Linear regression
  + Ordinary linear least-squares
    - Euclidean norm

Maximum likelihood estimation: estimation by optimisation

* Linear regression

Bayesian inference

* Inferring a posterior distribution over the parameters, given some prior belief and some evidence
* Likelihood function and a prior over parameters
* Linear regression
  + Using Markov Chain Monte Carlo
  + Probabilistic programming
    - Random variables – first-class (like grad for differentiation)
  + P(D) is likely intractable
    - No need to compute it:
      * Either draw sample from posterior distribution , or
      * Relative probability only
  + Markov Chain Monte Carlo
    - import pymc3
    - Priors
      * m, c,
    - Likelihood
    - Predictive posterior: sampling from the model
      * Distribution over observations
      * Drawing samples from model, while integrating over

# Lecture 10: Time Series, Signals

Sampled signals – continuous signals as sequences of numbers (usually integers)

* Spaced evenly (usually) – regular

Sliding window – moving average (mean)

* Linear filter
* Makes audio smoother (more continuous)
* Equal to

Non-linear filter – median filter

* Removes spikes in audio

Simplification

* Fewer oscillations
* Operation in the frequency domain

Aliasing

* Nyquist rate
  + Occurs when there are frequencies in the signal greater than half the sampling rate