Neuromatch Academy: Algebra Refresher - Summary Sheet[1]

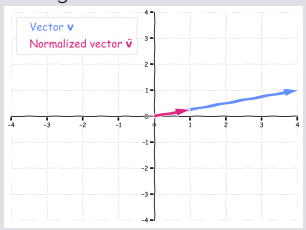
Vectors (W0D3T1)

A Vector

A vector, $\mathbf{v}$, is a short hand way of representing a list of numbers like x and y coordinates:

$$\mathbf{v} = \begin{bmatrix} 4 \\ 1 \end{bmatrix}, \tag{1}$$

A 2-D vector $\mathbf{v}$ has a direction and a length. A normalized vector $\tilde{\mathbf{v}}$ has a length 1.



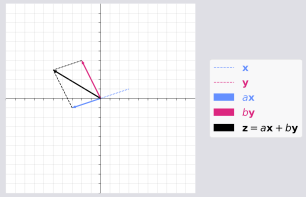
Linear Combination of Vectors

Two vectors $\mathbf{x}$ and $\mathbf{y}$ can be added together and multiplied by parameters a and b to get a new vector $\mathbf{z}$

$$\mathbf{z} = a\mathbf{x} + b\mathbf{y} \tag{2}$$

given $\mathbf{x} = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$ and $\mathbf{y} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$, and $a = -1$ and $b = 2$

$$\begin{bmatrix} -6 \\ 3 \end{bmatrix} = -1 \begin{bmatrix} 3 \\ 1 \end{bmatrix} + 2 \begin{bmatrix} -1 \\ 2 \end{bmatrix}.$$



The Geometry of the Dot Product $\mathbf{w} \cdot \mathbf{r}$

An alternate way of defining the dot product is as the multiple of the lengths of the two vectors and the angle between them θ:

$$\mathbf{x} \cdot \mathbf{y} = ||\mathbf{x}|| \cdot ||\mathbf{y}|| \cdot \cos(\theta). \tag{3}$$

Vectors (W0D3T1)

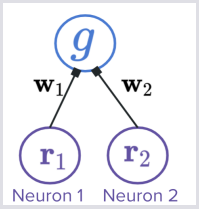
Dot Product $\mathbf{w} \cdot \mathbf{r}$

Given two retinal neurons with varying firing rates (r_1 and r_2). The retinal firing rates can be represented as the vector $\mathbf{r} = \begin{bmatrix} r_1 \\ r_2 \end{bmatrix}$.

The weights from each of these to an LGN neuron. The weights are represented with the vector $\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$.

The LGN firing rate is the dot product of the retinal firing rate vector and the weight vector:

$$g = \mathbf{w} \cdot \mathbf{r} = w_1 r_1 + w_2 r_2 \tag{4}$$



Matrices (W0D3T2)

Intro to Matrices

We will look at a group of 2 LGN neurons which get input from 2 retinal neurons: we will call the population of LGN neurons population p. Below, we have the system of linear equations that dictates the neuron models for each population. r_1 and r_2 correspond to the retinal neural activities (of neuron 1 and 2). g_{p_1} and g_{p_2} correspond to the responses of the LGN neurons 1 and 2 in population p.

$$r_1 + 3r_2 = g_{p_1} \tag{5}$$
$$2r_1 + r_2 = g_{p_2} \tag{6}$$

Cast each equation (i.e., g_{p_1} and g_{p_2}) as a matrix-vector multiplication:

$$\mathbf{g}_p = \mathbf{P}\mathbf{r} \tag{7}$$

where

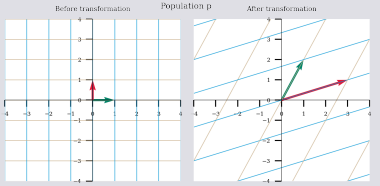
$$\mathbf{P} = \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix} \tag{8}$$

is the weight matrix to population p.

Matrices (W0D3T2)

Matrices as Linear Transformations

Matrices can be thought of as enacting linear transformations. When multiplied with a vector, they transform it into another vector. In fact, they are transforming a grid of space in a linear manner: the origin stays in place and grid lines remain straight, parallel, and evenly spaced.



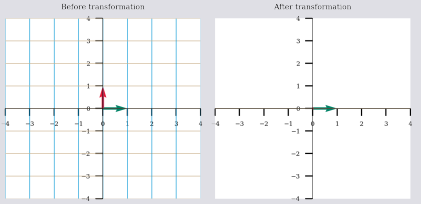
Eigenvalues & Eigenvectors

Eigenvectors, $\mathbf{v}$ of a matrix $\mathbf{W}$ are vectors that, when multipled by the matrix, equal a scalar multiple of themselves. That scalar multiple is the corresponding eigenvalue, λ.

$$\mathbf{W}\mathbf{v} = \lambda\mathbf{v} \tag{9}$$

If we have one eigenvector for a matrix, we technically have an infinite amount: every vector along the span of that eigenvector is also an eigenvector. So, we often use the unit vector in that direction to summarize all the eigenvectors along that line.

Just by looking at eigenvectors before and after a transformation, can you describe what the transformation is in words? Try for each of the two plots below.



Matrix Multiplication

We sometimes want to multiple two matrices together, instead of a matrix with a vector. Let's say we're multiplying matrices $\mathbf{A}$ and $\mathbf{B}$ to get $\mathbf{C}$:

$$\mathbf{C} = \mathbf{A}\mathbf{B}. \tag{10}$$

We take the dot product of each row of A with each column of B. The resulting scalar is placed in the element of $\mathbf{C}$ that is the same row (as the row in A) and column (as the column in B). So the element of $\mathbf{C}$ at row 4 and column 2 is the dot product of the 4th row of $\mathbf{A}$ and the 2nd column of $\mathbf{B}$. We can write this in a formula as:

$$\mathbf{C}_{\text{row i, column j}} = \mathbf{A}_{\text{row i}} \cdot \mathbf{B}_{\text{column j}} \tag{11}$$

[1]'t Hart et al., (2022). Neuromatch Academy: a 3-week, online summer school in computational neuroscience. Journal of Open Source Education, 5(49), 118. https://doi.org/10.21105/jose.00118

Neuromatch Academy: Calculus Refresher - Summary Sheet[2]

Differentiation and Integration (W0D4T1)

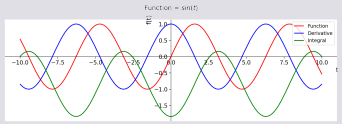
Introduction

Differentiation of a function $f(t)$ gives you the derivative of that function

$$\frac{d(f(t))}{dt}. \tag{12}$$

A derivative captures how sensitive a function is to slight changes in the input for different ranges of inputs.
Integration can be thought of as the reverse of differentiation

$$\int f(t)dt. \tag{13}$$



Analytical Differentiation

When we find the derivative analytically, we are finding the exact formula for the derivative function. To do this, instead of having to do some fancy math every time, we can consult a list of common derivatives such as:
The Product Rule

$$f(t) = u(t) \cdot v(t) \tag{14}$$

$$\frac{d(f(t))}{dt} = v \cdot \frac{du}{dt} + u \cdot \frac{dv}{dt} \tag{15}$$

The Chain Rule:

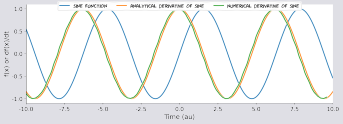
$$\frac{dr}{da} = \frac{dr}{dt} \cdot \frac{dt}{da}. \tag{16}$$

Numerical Differentiation

Formally, the derivative of a function $f(x)$ at any value a is given by the finite difference formula (FD):

$$FD = \frac{f(a+h) - f(a)}{h} \tag{17}$$

As $h \rightarrow 0$, the FD approaches the actual value of the derivative.



Differentiation and Integration (W0D4T1)

Functions of Multiple Variables

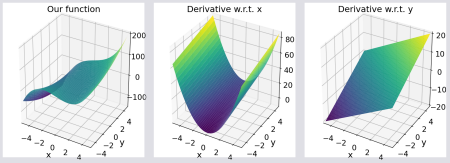
In most cases, we encounter functions of multiple variables. For example, in the brain, the firing rate of a neuron is a function of both excitatory and inhibitory input rates. In the following, we will look into how to calculate derivatives of such functions.
When we take the derivative of a multivariable function with respect to one of the variables it is called the "*partial derivative"*. For example if we have a function:

$$f(x, y) = x^3 + 2xy + y^2 \tag{18}$$

The we can define the partial derivatives as

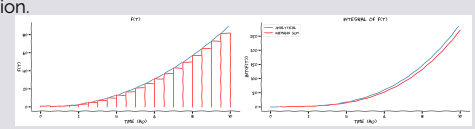
$$\frac{\partial(f(x,y))}{\partial x} = 3x^2 + 2y + 0 \tag{19}$$

$$\frac{\partial(f(x,y))}{\partial y} = 0 + 2x + 2y \tag{20}$$



Numerical Integration (Riemann Sum)

Geometrically, integration is the area under the curve. This interpretation gives two formal ways to calculate the integral of a function numerically.
If we wish to integrate a function $f(t)$ with respect to t, then first we divide the function into n intervals of size $dt = a - b$, where a is the starting of the interval. Thus, each interval gives a rectangle with height $f(a)$ and width dt. By summing the area of all the rectangles, we can approximate the area under the curve. As the size dt approaches to zero, our estimate of the integral approaches the analytical calculation.



Differential Equations (W0D4T2)

Introduction

Differential Equations are mathematical equations that describe how something like population or a neuron changes over time. The reason why differential equations are so useful is they can generalise a process such that one equation can be used to describe many different outcomes. The general form of a first order differential equation is:

$$\frac{d}{dt}y(t) = f\left(t, y(t)\right) \tag{21}$$

which can be read as "the change in a process y over time t is a function f of time t and itself y".

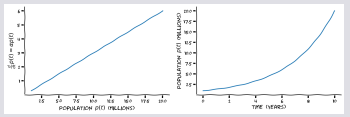
Population Differential Equation

The linear population equation

$$\frac{d}{dt}p(t) = \alpha p(t),\ p(0) = P_0 \tag{22}$$

has the exact solution:$p(t) = P_0 e^{\alpha t}$. The exact solution written in words is:

"Population" = "grows/declines exponentially as a function of time and birth rate".

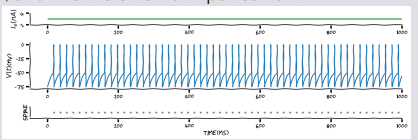


The leaky integrate and fire (LIF) model

The Leaky Integrate and Fire Model is a linear differential equation that describes the membrane potential (V) of a single neuron which was proposed by Louis Édouard Lapicque in 1907.
The subthreshold membrane potential dynamics of a LIF neuron is described by

$$\tau_m \frac{dV}{dt} = -(V - E_L) + R_m I \tag{23}$$

where τ_m is the time constant, V is the membrane potential, E_L is the resting potential, R_m is membrane resistance, and I is the external input current.



[2] 't Hart et al., (2022). Neuromatch Academy: a 3-week, online summer school in computational neuroscience. Journal of Open Source Education, 5(49), 118. https://doi.org/10.21105/jose.00118

Numerical Methods (W0D4T3)

Euler Method

The Euler method is one of the straight forward and elegant methods to approximate a differential. It was designed by Leonhard Euler (1707-1783). Simply put we just replace the derivative in the differential equation by the formula for a line and re-arrange.
The slope is the rate of change between two points. The formula for the slope of a line between the points $(t_0, y(t_0))$ and $(t_1, y(t_1))$ is given by:

$$m = \frac{y(t_1) - y(t_0)}{t_1 - t_0} = \frac{\Delta y_0}{\Delta t_0},$$

where $\Delta y_0 = y_1 - y_0$ and $\Delta t_0 = t_1 - t_0$ or in words as

$$m = \frac{\text{Change in y}}{\text{Change in t}}.$$

The slope can be used as an approximation of the derivative such that

$$\frac{d}{dt}y(t) \approx \frac{y(t_0 + \Delta t) - y(t_0)}{t_0 + \Delta t - t_0} = \frac{y(t_0 + dt) - y(t_0)}{\Delta t}$$

where Δt is a time-step.

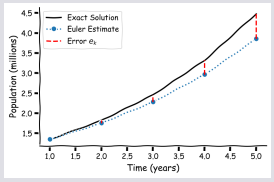
Population Differential Equation

To numerically estimate the population differential equation we replace the derivative with the slope of the line to get the discrete (not continuous) equation where p_1 is the estimate of $p(t_1)$. Let $\Delta t = t_1 - t_0$ be the time-step and re-arrange the equation gives

$$p_1 = p_0 + \Delta t(\alpha p_0)$$

where p_1 is the unknown future, p_0 is the known current population, Δt is the chosen time-step parameter and α is the given birth rate parameter.
The solution of the Euler method p_1 is an estimate of the exact solution $p(t_1)$ at t_1 which means there is a bit of error e_1 which gives the equation

$$e_1 = p(t_1) - p_1,$$
$$\text{Error} = \text{Exact-Estimate}.$$



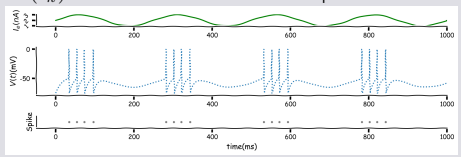
Numerical Method (W0D4T3)

Linear Integrate and Fire

The solution of the LIF can be estimated by applying the Euler method to give the difference equation:

$$V[k+1] = V[k] + \Delta t\left(\frac{-(V[k] - E_L) + R_m I[k]}{\tau_m}\right),$$
$$\text{for } k = 0 \cdots n-1,$$

where $V[k]$ is the estimate of the membrane potential at time point $t[k]$, $V[k+1]$ is the unknown membrane potential at $t[k+1]$, $V[k]$ is known membrane potential, E_L, R_m and τ_m are known parameters, Δt is a chosen time-step and $I(t_k)$ is a function for an external input current.



Systems of Differential Equations

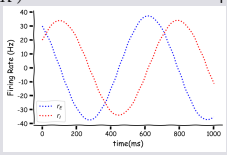
We now model a collection of neurons using a differential equation which describes the firing rate of a population of neurons. We will model the firing rate r of two types of populations of neurons which interact, the excitation population firing rate r_E and inhibition population firing rate r_I. The two coupled differential equations with weights w are:

$$\tau_E \frac{dr_E}{dt} = w_{EE} r_E + w_{EI} r_I, \tag{24}$$
$$\tau_I \frac{dr_I}{dt} = w_{IE} r_E + w_{II} r_I, \tag{25}$$

The solutions can be approximated using the Euler method such that the equations become:

$$r_E[k+1] = r_E[k] + \Delta t\left(\frac{w_{EE} r_E[k] + w_{EI} r_I[k]}{\tau_E}\right),$$
$$r_I[k+1] = r_I[k] + \Delta t\left(\frac{w_{II} r_I[k] + w_{IE} r_E[k]}{\tau_I}\right),$$
$$\text{for } k = 0, \cdots n-1,$$

where $r_E[k]$ and $r_I[k]$ are the numerical estimates of the firing rate of the excitation population $r_E(t_k)$ and inhibition population $r_I(t_K)$ and Δt is the time-step.

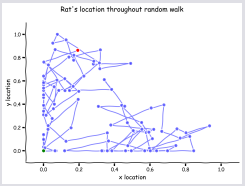


Neuromatch Academy: Statistics Refresher - Summary Sheet[3]

Statistics (W0D5T1) - Probability Distributions

Random Walk

Stochastic models can be used to create models of behaviour. As an example, imagine that a rat is placed inside a novel environment, a box. We could try and model its exploration behaviour by assuming that for each time step it takes a random uniformly sampled step in any direction (simultaneous random step in x direction and random step in y direction)



Discrete Distributions Differentiation

A simple way to model random behaviour is with a single **Bernoulli trial**, that has two outcomes, $Left, Right$, with probability $P(Left) = p$ and $P(Right) = 1 - p$ as the two mutually exclusive possibilities (whether the rat goes down the left or right arm of the maze).
The binomial distribution simulates n number of binary events, such as the $Left, Right$ choices of the random rat in the T-maze is given by:

$$P(k|n,p) = \left(\binom{n}{k}\right) p^k (1-p)^{n-k}$$
$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

where, p is the probability of turning left, n is the number of binary events, or trials, and k is the number of times the rat turned left. The term $\binom{n}{k}$ is the binomial coefficient. **The Poisson distribution** is a 'point-process', meaning that it determines the number of discrete 'point', or binary, events that happen within a fixed space or time, allowing for the occurrence of a potentially infinite number of events. The Poisson distribution is specified by a single parameter λ that encapsulates the mean number of events that can occur in a single time or space interval. The formula for a Poisson distribution is:

$$P(x) = \frac{\lambda^x e^{-\lambda}}{x!} \tag{26}$$

where λ is a parameter corresponding to the average outcome of x.

Statistics (W0D5T1) - Probability Distributions

Continuous Distributions

We do not have to restrict ourselves to only probabilistic models of discrete events. While for discrete outcomes we can ask about the probability of an specific event ("what is the probability this neuron will fire 4 times in the next second"), this is not defined for a continuous distribution ("what is the probability of the BOLD signal being exactly 4.000120141..."). Hence we need to focus on intervals when calculating probabilities from a continuous distribution.
If we want to make predictions about possible outcomes for a continuous distribution we can use the integral

$$\int_{x_1}^{x_2} P(x)dx \tag{27}$$

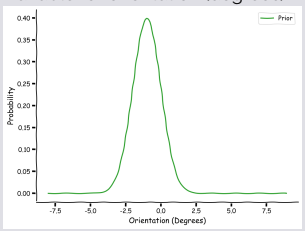
where $P(x)$ is a probability density function.

Gaussian Distribution

The most widely used continuous distribution is probably the Gaussian (also known as Normal) distribution. It is extremely common across all kinds of statistical analyses. Because of the central limit theorem, many quantities are Gaussian distributed. Gaussians also have some nice mathematical properties that permit simple closed-form solutions to several important problems.
The equation for a Gaussian probability density function is:

$$f(x;\mu,\sigma^2) = \mathcal{N}(\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right) \tag{28}$$

where $\mu = -1$ is the mean, $\sigma = 1$ is the variance and x is the random variable for orientation (degrees).



Statistics (W0D5T2) - Statistical Inference

Basic Probability

Probability has to be in the range 0 to 1 $P(A) \in [0,1]$ and the complementary can always be defined as

$$P(\neg A) = 1 - P(A).$$

When we have two variables, the **conditional probability** of A given B is

$$P(A|B) = P(A \cap B)/P(B) = P(A,B)/P(B)$$

while the **joint probability** of A and B is

$$P(A \cap B) = P(A,B) = P(B|A)P(A) = P(A|B)P(B)$$

We can then also define the process of **marginalisation** (for discrete variables) as

$$P(A) = \sum P(A,B) = \sum P(A|B)P(B)$$

where the summation is over the possible values of B.
As an example if B is a binary variable that can take values $B+$ or $B0$ then

$$P(A) = \sum P(A,B) = P(A|B+)P(B+) + P(A|B0)P(B0).$$

For continuous variables marginalization is given as

$$P(A) = \int P(A,B)dB = \int P(A|B)P(B)dB.$$

Markov chains

The Markov property specifies that you can fully encapsulate the important properties of a system based on its current state at the current time, any previous history does not matter. It is memoryless.
As an example imagine that a rat is able to move freely between 3 areas: a dark rest area ($state = 1$), a nesting area ($state = 2$) and a bright area for collecting food ($state = 3$). Every 5 minutes (timepoint i) we record the rat's location. We can use a **"categorical distribution"** to look at the probability that the rat moves to one state from another.
We can model this as a Markov chain, so the animal is only in one of the states at a time and can transition between the states. We want to get the probability of each state at time $i+1$.

$$\begin{aligned} P(state_{i+1} = 1) = \\ P(state_{i+1} = 1|state_i = 1)P(state_i = 1) \\ +P(state_{i+1} = 1|state_i = 2)P(state_i = 2) \\ +P(state_{i+1} = 1|state_i = 3)P(state_i = 3) \end{aligned}$$

[3] 't Hart et al., (2022). Neuromatch Academy: a 3-week, online summer school in computational neuroscience. Journal of Open Source Education, 5(49), 118. https://doi.org/10.21105/jose.00118

Statistics (WOD5T2) - Statistical Inference

Statistical inference and likelihood

If we do not know the parameters μ, σ that generated the data, we can try to **infer** which parameter values (given our model) gives the best (highest) likelihood. This is what we call statistical inference: trying to infer what parameters make our observed data the most likely or probable.
A generative model (such as the Gaussian distribution from the previous tutorial) allows us to make predictions about outcomes.
After we observe n data points, we can evaluate our model (and any of its associated parameters) by calculating the **likelihood** of our model having generated each of those data points x_i.

$$P(x_i|\mu,\sigma) = \mathcal{N}(x_i,\mu,\sigma) \tag{29}$$

For all data points $\mathbf{x} = (x_1, x_2, x_3, ...x_n)$ we can then calculate the likelihood for the whole dataset by computing the product of the likelihood for each single data point.

$$P(\mathbf{x}|\mu,\sigma) = \prod_{i=1}^{n} \mathcal{N}(x_i,\mu,\sigma) \tag{30}$$

While the likelihood may be written as a conditional probability ($P(x|\mu,\sigma)$), we refer to it as the **likelihood function**, $L(\mu,\sigma)$. This slight switch in notation is to emphasize our focus: we use likelihood functions when the data points $\mathbf{x}$ are fixed and we are focused on the parameters.
Our new notation makes clear that the likelihood $L(\mu,\sigma)$ is a function of μ and σ, not of $\mathbf{x}$.

Maximum likelihood

Implicitly, by looking for the parameters that give the highest likelihood in the last section, we have been searching for the **maximum likelihood** estimate

$$(\hat{\mu},\hat{\sigma}) = \text{argmax}_{\mu,\sigma} L(\mu,\sigma) = \text{argmax}_{\mu,\sigma} \prod_{i=1}^{n} \mathcal{N}(x_i,\mu,\sigma). \tag{31}$$

We want to do inference on this data set, i.e. we want to infer the parameters that most likely gave rise to the data given our model. Intuitively that means that we want as good as possible a fit between the observed data and the probability distribution function with the best inferred parameters. We can search for the best parameters manually by trying out a bunch of possible values of the parameters, computing the likelihoods, and picking the parameters that resulted in the highest likelihood.

Statistics (WOD5T2) - Statistical Inference

Bayesian Inference

For Bayesian inference we do not focus on the likelihood function $L(y) = P(x|y)$, but instead focus on the posterior distribution:

$$P(y|x) = \frac{P(x|y)P(y)}{P(x)} \tag{32}$$

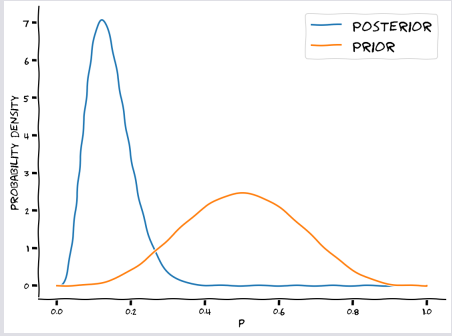
which is composed of the "likelihood" function $P(x|y)$, the "prior" $P(y)$ and a normalising term $P(x)$ (which we will ignore for now).
While there are other advantages to using Bayesian inference (such as the ability to derive Bayesian Nets, see optional bonus task below), we will start by focusing on the role of the prior in inference.

Conjugate priors

Bayesian inference can be used for any likelihood distribution, but it is a lot more convenient to work with **conjugate** priors, where multiplying the prior with the likelihood just provides another instance of the prior distribution with updated values.
For the binomial likelihood it is convenient to use the beta distribution as a prior

$$f(p;\alpha,\beta) = \frac{1}{\mathrm{B}(\alpha,\beta)} p^{\alpha-1}(1-p)^{\beta-1} \tag{33}$$

where B is the beta function, α and β are parameters, and p is the probability of the rat turning left or right. The beta distribution is thus a distribution over a probability.
Given a series of Left and Right moves of the rat, we can now estimate the probability that the animal will turn left. Using Bayesian Inference, we use a beta distribution **prior**, which is then multiplied with the **likelihood** to create a **posterior** that is also a beta distribution, but with updated parameters.



Neuromatch Academy Model Types - Summary Sheet[4]

"What" models

Overview

We will explore **"What"** models, used to describe the data. To understand what our data looks like, we will visualize it in different ways. Then we will compare it to simple mathematical models. Specifically, we will:

- Use spiking activity from hundreds of neurons and understand how it is organized
- Visualize characteristics of the spiking activity across the population
- Compute the distribution of "inter-spike intervals" (ISIs) for a single neuron
- Consider several formal models of this distribution's shape and fit them to the data "by hand"

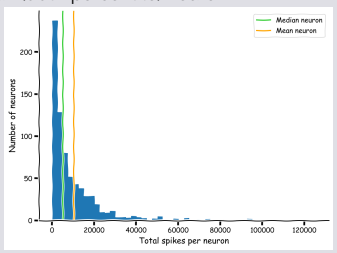
Exploring the Steinmetz dataset

We consider a subset of data from a study of Steinmetz et al. (2019). In this study, Neuropixels probes were implanted in the brains of mice. Electrical potentials were measured by hundreds of electrodes along the length of each probe. Each electrode's measurements captured local variations in the electric field due to nearby spiking neurons. A spike sorting algorithm was used to infer spike times and cluster spikes according to common origin: a single cluster of sorted spikes is causally attributed to a single neuron.
In particular, a single recording session of spike times and neuron assignments was loaded and assigned to *spike times* in the preceding setup. Typically a dataset comes with some information about its structure. However, this information may be incomplete.
You might also apply some transformations or "preprocessing" to create a working representation of the data of interest, which might go partly undocumented depending on the circumstances. In any case it is important to be able to use the available tools to investigate unfamiliar aspects of a data structure.

"What" models

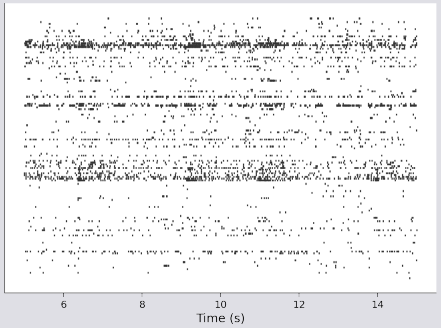
Plotting total spike counts

The number of spikes over the entire recording is variable between neurons. More generally, some neurons tend to spike more than others in a given period. Let's explore what the distribution of spiking looks like across all the neurons in the dataset.
The plot shows that the majority of neurons are relatively "quiet" compared to the mean, while a small number of neurons are exceptionally "loud": they must have spiked more often to reach a large count. If the mean neuron is more active than 68% of the population, what does that imply about the relationship between the mean neuron and the median (50th percentile) neuron?



Visualizing neuronal spiking activity

A "raster" plot, where the spikes from each neuron appear in a different row.
Plotting a large number of neurons can give you a sense for the characteristics in the population.



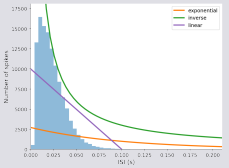
"What" models

Inter-spike intervals and their distributions

Given the ordered arrays of spike times, what can we ask next?
Scientific questions are informed by existing models. So, what knowledge do we already have that can inform questions about this data?
We know that there are physical constraints on neuron spiking. Spiking costs energy, which the neuron's cellular machinery can only obtain at a finite rate. Therefore neurons should have a refractory period: they can only fire as quickly as their metabolic processes can support, and there is a minimum delay between consecutive spikes of the same neuron. More generally, we can ask "how long does a neuron wait to spike again?" or "what is the longest a neuron will wait?" Can we transform spike times into something else, to address questions like these more directly? We can consider the inter-spike times (or interspike intervals: ISIs). These are simply the time differences between consecutive spikes of the same neuron. In general, the shorter ISIs are predominant, with counts decreasing rapidly (and smoothly, more or less) with increasing ISI. However, counts also rapidly decrease to zero with decreasing ISI below the maximum of the distribution (8-11 ms). The absence of these very low ISIs agrees with the refractory period hypothesis: the neuron cannot fire quickly enough to populate this region of the ISI distribution.

What is the functional form of an ISI distribution?

The ISI histograms seem to follow continuous, monotonically decreasing functions above their maxima. The function is clearly non-linear. Could it belong to a single family of functions?
To motivate the idea of using a mathematical function to explain physiological phenomena, let's define a few different function forms that we might expect the relationship to follow: exponential, inverse, and linear.



The exponential function can be made to fit the data much better than the linear or inverse function.

[4] 't Hart et al., (2022). Neuromatch Academy: a 3-week, online summer school in computational neuroscience. Journal of Open Source Education, 5(49), 118. https://doi.org/10.21105/jose.00118

"How" models

Overview

We will explore models that can potentially explain **"How"** the spiking data we have observed is produced. To understand the mechanisms we will build simple neuronal models and compare their spiking response to real data. We will:

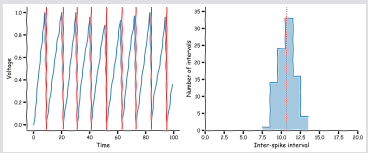
1. Simulate a simple "leaky integrate-and-fire" neuron model
2. Make the model more complicated — but also more realistic—by adding more physiologically-inspired details

How does a neuron spike?

A neuron charges and discharges an electric field across its cell membrane. The state of this electric field can be described by the *membrane potential*. The membrane potential rises due to excitation of the neuron, and when it reaches a threshold a spike occurs. The potential resets, and must rise to a threshold again before the next spike occurs.
One of the simplest models of spiking neuron behavior is the linear integrate-and-fire (LIF) model neuron. In this model, the neuron increases its membrane potential V_m over time in response to excitatory input currents I scaled by some factor α:

$$dV_m = \alpha I \tag{34}$$

Once V_m reaches a threshold value a spike is produced, V_m is reset to a starting value, and the process continues. Here, we will take the starting and threshold potentials as 0 and 1, respectively. So, for example, if $\alpha I = 0.1$ is constant—that is, the input current is constant—then $dV_m = 0.1$, and at each timestep the membrane potential V_m increases by 0.1 until after $(1-0)/0.1 = 10$ timesteps it reaches the threshold and resets to $V_m = 0$, and so on.
Note that we define the membrane potential V_m as a scalar: a single real (or floating point) number. However, a biological neuron's membrane potential will not be exactly constant at all points on its cell membrane at a given time. We could capture this variation with a more complex model (e.g. with more numbers). Do we need to?
The proposed model is a 1D simplification. There are many details we could add to it, to preserve different parts of the complex structure and dynamics of a real neuron. If we were interested in small or local changes in the membrane potential, our 1D simplification could be a problem. However, we'll assume an idealized "point" neuron model for our current purpose.



"How" models

Spiking Inputs

Given our simplified model for the neuron dynamics, we still need to consider what form the input I will take. How should we specify the firing behavior of the presynaptic neuron(s) providing the inputs to our model neuron?
Unlike in the simple example the input current is generally not constant. Physical inputs tend to vary with time. We can describe this variation with a distribution.
We'll assume the input current I over a timestep is due to equal contributions from a non-negative (≥ 0) integer number of input spikes arriving in that timestep. Our model neuron might integrate currents from 3 input spikes in one timestep, and 7 spikes in the next timestep. We should see similar behavior when sampling from our distribution.
Given no other information about the input neurons, we will also assume that the distribution has a mean, and that the spiking events of the input neuron(s) are independent in time. Are these reasonable assumptions in the context of real neurons?
A suitable distribution given these assumptions is the Poisson distribution, which we'll use to model I:

$$I \sim \text{Poisson}(\lambda) \tag{35}$$

where λ is the mean of the distribution: the average rate of spikes received per timestep.

Inhibitory signals

Our linear IF neuron from the previous section was indeed able to produce spikes. However, our ISI histogram doesn't look much like an empirical ISI histograms, which has an exponential-like shape. What is our model neuron missing, given that it doesn't behave like a real neuron?
In the previous model we only considered excitatory behavior. We know, however, that there are other factors that can drive V_m down. First is the natural tendency of the neuron to return to some steady state or resting potential. We can update our previous model as follows:

$$dV_m = -\beta V_m + \alpha I \tag{36}$$

where V_m is the current membrane potential and β is some leakage factor. This is a basic form of the popular LIF model.
We also know that in addition to excitatory presynaptic neurons, we can have inhibitory presynaptic neurons as well. We can model these inhibitory neurons with another Poisson random variable:

$$I = I_{\text{exc}} - I_{\text{inh}} \tag{37}$$

$$I_{\text{exc}} \sim \text{Poisson}(\lambda_{\text{exc}}) \tag{38}$$

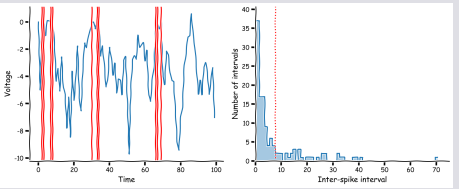
$$I_{\text{inh}} \sim \text{Poisson}(\lambda_{\text{inh}}) \tag{39}$$

where λ_{exc} and λ_{inh} are the average spike rates (per timestep) of the excitatory and inhibitory presynaptic neurons, respectively.

"How" models

LIF + inhibition neuron

1. Raising the excitatory rate while keeping inhibitory rate the same results in more frequent firing and shorter ISIs. This makes intuitive sense - more excitatory input means higher responses.
2. Raising the inhibitory rate while keeping the excitatory rate the same results in less frequent firing and longer ISIs. This makes intuitive sense - more inhibitory input means lower responses.
3. If the excitatory and inhibitory rates are equal, the average inter-spike interval stays the same even if you raise both. This is because they balance each other out.
4. Yes, these ISIs look more exponential, like what we observed.



Notation

V_m	membrane potential
dV_m	change in membrane potential
C_m	membrane capacitance
I	input current
R_m	membrane resistance :
V_{rest}	resting potential
α	scaling factor for input current
β	leakage factor
λ	average spike rate
λ_{exc}	average spike rate for excitatory neurons
λ_{inh}	average spike rate for inhibitory neurons

"Why" models

Overview

We will explore models and techniques that can potentially explain **why** the spiking data we have observed is produced the way it is.
To understand why different spiking behaviors may be beneficial, we will learn about the concept of entropy. Specifically, we will:

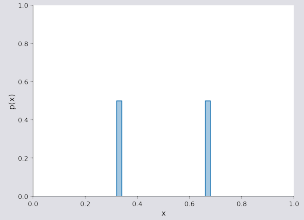
1. Write code to compute formula for entropy, a measure of information
2. Compute the entropy of a number of toy distributions
3. Compute the entropy of spiking activity from the Steinmetz dataset

Optimization and Information

Neurons can only fire so often in a fixed period of time, as the act of emitting a spike consumes energy that is depleted and must eventually be replenished. To communicate effectively for downstream computation, the neuron would need to make good use of its limited spiking capability. This becomes an optimization problem:
What is the optimal way for a neuron to fire in order to maximize its ability to communicate information?
In order to explore this question, we first need to have a quantifiable measure for information. Shannon introduced the concept of entropy to do just that, and defined it as

$$H_b(X) = -\sum_{x \in X} p(x) \log_b p(x) \tag{40}$$

where H is entropy measured in units of base b and $p(x)$ is the probability of observing the event x from the set of all possible events in X. See the Bonus Section 1 for a more detailed look at how this equation was derived.
The most common base of measuring entropy is $b = 2$, so we often talk about **bits** of information, though other bases are used as well (e.g. when $b = e$ we call the units *nats*).
A distribution with mass split equally between two points looks like:



Here, the entropy calculation is: $-(0.5 \log_2 0.5 + 0.5 \log_2 0.5) = 1$ ere is 1 bit of entropy. This means that before we take a random sample, there is 1 bit of uncertainty about which point in the distribution the sample will fall on: it will either be the first peak or the second one.

"Why" models

Entropy

Likewise, if we make one of the peaks taller and the other one shorter, the entropy will decrease because of the increased certainty that the sample will fall on one point and not the other: : $-(0.2 \log_2 0.2 + 0.8 \log_2 0.8) \approx 0.72$
If we split the probability mass among even more points, the entropy continues to increase. Let's derive the general form for N points of equal mass, where $p_i = p = 1/N$:

$$-\sum_i p_i \log_b p_i = -\sum_i^N \frac{1}{N} \log_b \frac{1}{N} \tag{41}$$

$$= -\log_b \frac{1}{N} \tag{42}$$

$$= \log_b N \tag{43}$$

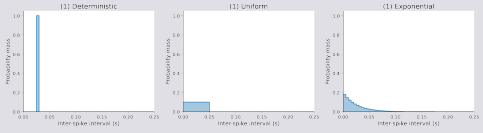
If we have N discrete points, the *uniform distribution* (where all points have equal mass) is the distribution with the highest entropy: $\log_b N$. This upper bound on entropy is useful when considering binning strategies, as any estimate of entropy over N discrete points (or bins) must be in the interval $[0, \log_b N]$

Information

We'll consider three hypothetical neurons that all have the same mean ISI, but with different distributions:

1. Deterministic
2. Uniform
3. Exponential

Fixing the mean of the ISI distribution is equivalent to fixing its inverse: the neuron's mean firing rate. If a neuron has a fixed energy budget and each of its spikes has the same energy cost, then by fixing the mean firing rate, we are normalizing for energy expenditure. This provides a basis for comparing the entropy of different ISI distributions. In other words: if our neuron has a fixed budget, what ISI distribution should it express (all else being equal) to maximize the information content of its outputs?
Let's construct our three distributions and see how their entropies differ.



1. Deterministic: 0.00 bits
2. Uniform: 3.32 bits
3. Exponential: 3.77 bits

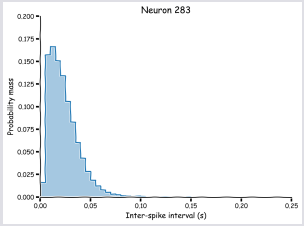
"Why" models

Calculate entropy of ISI distributions from data

In the previous example we created the PMFs by hand to illustrate idealized scenarios. How would we compute them from data recorded from actual neurons?
One way is to convert the ISI histograms we've previously computed into discrete probability distributions using the following equation:

$$p_i = \frac{n_i}{\sum_i n_i} \tag{44}$$

where p_i is the probability of an ISI falling within a particular interval i and n_i is the count of how many ISIs were observed in that interval.



Entropy for Neuron 283: 3.36 bits

Summary

We used different types of models to understand the spiking behavior of neurons recorded in the Steinmetz data set.

- We used "what" models to discover that the ISI distribution of real neurons is closest to an exponential distribution
- We used "how" models to discover that balanced excitatory and inhibitory inputs, coupled with a leaky membrane, can give rise to neuronal spiking with exhibiting such an exponential ISI distribution
- We used "why" models to discover that exponential ISI distributions contain the most information when the mean spiking is constrained

Notation

$H(X)$	entropy of random variable X
b	base, e.g. b=2 or b=e
x	event x
$p(x)$	probability of observing event x
ISI	interspike interval
n_i	count of observed ISIs in interval i
p_i	probability of of an ISI falling within a particular interval i

Neuromatch Academy Modeling Practice - Summary Sheet[5]

Modeling Practice

Framing the Question

We will try to clarify the process of computational modeling, by thinking through the logic of modeling a project.
We assume that you have a general idea of a project in mind, i.e. a preliminary question, and/or phenomenon you would like to understand. You should have started developing a project idea yesterday with this brainstorming demo. Maybe you have a goal in mind. We will now work through the 4 first steps of modeling (Blohm et al., 2019):
Framing the question:

1. finding a phenomenon and a question to ask about it
2. understanding the state of the art
3. determining the basic ingredients
4. formulating specific, mathematically defined hypotheses

The remaining steps 5-10 are covered later.
Importantly, we will guide you through Steps 1-4 today. After you do more work on projects, you likely have to revise some or all of these steps "before" you move on to the remaining steps of modeling. "Models" here can be data analysis pipelines, not just computational models...

Modeling Practice

Step 1 - Asking your own question

You should already have a project idea from your brainstorming yesterday. **Write down the phenomenon, question and goal(s) if you have them.**
As a reminder, here is what you should discuss and write down:
What exact aspect of data needs modeling?

- Answer this question clearly and precisely! Otherwise you will get lost (almost guaranteed)
- Write everything down!
- Also identify aspects of data that you do not want to address (yet)

Define an evaluation method!

- How will you know your modeling is good?
- E.g. comparison to specific data (quantitative method of comparison?)

For computational models: think of an experiment that could test your model

- You essentially want your model to interface with this experiment, i.e. you want to simulate this experiment

You can find interesting questions by looking for phenomena that differ from your expectations. In **what** way does it differ? **How** could that be explained (starting to think about mechanistic questions and structural hypotheses)? **Why** could it be the way it is? What experiment could you design to investigate this phenomenon? What kind of data would you need?

Step 1 - Make sure to avoid the pitfalls!

Question is too general

- Remember: science advances one small step at the time. Get the small step right…

Precise aspect of phenomenon you want to model is unclear

- You will fail to ask a meaningful question<

You have already chosen a toolkit

- This will prevent you from thinking deeply about the best way to answer your scientific question

You don't have a clear goal

- What do you want to get out of modeling?

You don't have a potential experiment in mind

- This will help concretize your objectives and think through the logic behind your goal

Modeling Practice

Step 2 - Understanding the state of the art & background

Here is what you should get out of it:
Survey the literature

- What's known?
- What has already been done?
- Previous models as a starting point?
- What hypotheses have been emitted in the field?
- Are there any alternative / complementary modeling approaches?

What skill sets are required?

- Do I need to learn something before I can start?
- Ensure that no important aspect is missed

Potentially provides specific data sets / alternative modeling approaches for comparison

[5] 't Hart et al., (2022). Neuromatch Academy: a 3-week, online summer school in computational neuroscience. Journal of Open Source Education, 5(49), 118. https://doi.org/10.21105/jose.00118

Modeling Practice

Step 3 - Determining the basic ingredients

This will allow you to think deeper about what your modeling project will need. It's a crucial step before you can formulate hypotheses because you first need to understand what your modeling approach will need. There are 2 aspects you want to think about:
1. What parameters / variables are needed?

- Constants?
- Do they change over space, time, conditions…?
- What details can be omitted?
- Constraints, initial conditions?
- Model inputs / outputs?

2. Variables needed to describe the process to be modeled?

- Brainstorming!
- What can be observed / measured? latent variables?
- Where do these variables come from?
- Do any abstract concepts need to be instantiated as variables?
- E.g. value, utility, uncertainty, cost, salience, goals, strategy, plant, dynamics
- Instantiate them so that they relate to potential measurements!

This is a step where your prior knowledge and intuition is tested. You want to end up with an inventory of *specific* concepts and/or interactions that need to be instantiated.

Step 3 - Pitfalls

I'm experienced, I don't need to think about ingredients anymore

- Or so you think…

I can't think of any ingredients

- Think about the potential experiment. What are your stimuli? What parameters? What would you control? What do you measure?

I have all inputs and outputs

- Good! But what will link them? Thinking about that will start shaping your model and hypotheses

I can't think of any links (= mechanisms)

- You will acquire a library of potential mechanisms as you keep modeling and learning
- But the literature will often give you hints through hypotheses
- If you still can't think of links, then maybe you're missing ingredients?

Modeling Practice

Step 4 - Formulating your hypothesis

Once you have your question and goal lined up, you have done a literature review (let's assume for now) and you have thought about ingredients needed for your model, you're now ready to start thinking about *specific* hypotheses. Formulating hypotheses really consists of two consecutive steps:

1. You think about the hypotheses in words by relating ingredients identified in Step 3

- What is the model mechanism expected to do?
- How are different parameters expected to influence model results?

2. You then express these hypotheses in mathematical language by giving the ingredients identified in Step 3 specific variable names.

- Be explicit, e.g. $y(t) = f(x(t), k)$ but $z(t)$ doesn't influence y

There are also "structural hypotheses" that make assumptions on what model components you hypothesize will be crucial to capture the phenomenon at hand.

Important: Formulating the hypotheses is the last step before starting to model. This step determines the model approach and ingredients. It provides a more detailed description of the question / goal from Step 1. The more precise the hypotheses, the easier the model will be to justify.

Step 4 - Pitfalls

I don't need hypotheses, I will just play around with

- Hypotheses help determine and specify goals. You can (and should) still play…

My hypotheses don't match my question (or vice versa)

- This is a normal part of the process!
- You need to loop back to Step 1 and revisit your question / phenomenon / goals

I can't write down a math hypothesis

- Often that means you lack ingredients and/or clarity on the hypothesis
- OR: you have a "structural" hypothesis, i.e. you expect a certain model component to be crucial in explaining the phenomenon / answering the question

Modeling Practice Example - The Train illusion

Step 1 - Asking your own question

The train illusion occurs when sitting on a train and viewing another train outside the window. Suddenly, the other train "seems" to move, i.e. you experience visual motion of the other train relative to your train. But which train is actually moving?
Often people have the wrong percept. In particular, they think their own train might be moving when it's the other train that moves; or vice versa. The illusion is usually resolved once you gain vision of the surroundings that lets you disambiguate the relative motion; or if you experience strong vibrations indicating that it is indeed your own train that is in motion.
We asked the following (arbitrary) question for our demo project: "How do noisy vestibular estimates of motion lead to illusory percepts of self motion?"

Step 2 - Understanding the state of the art & background

This is where you would do a literature search to learn more about what's known about self-motion perception and vestibular signals. You would also want to examine any attempts to model self-motion, perceptual decision making and vestibular processing.

Step 3 - Determining the basic ingredients

We determined that we probably needed the following ingredients for our model:

- Vestibular input: $v(t)$
- Binary decision output: d - time dependent?
- Decision threshold: θ
- A filter (maybe running average?): f
- An integration mechanism to get from vestibular acceleration to sensed velocity: $\int$

Step 4 - Formulating your hypothesis

Our main hypothesis is that the strength of the illusion has a linear relationship to the amplitude of vestibular noise. Mathematically, this would write as $S = k \cdot N$ where S is the illusion strength and N is the noise level, and k is a free parameter. we could simply use the frequency of occurance across repetitions as the "strength of the illusion" We would get the noise as the standard deviation of $v(t)$, i.e.

$$N = E[v(t)^2],$$

where E stands for the expected value. Do we need to take the average across time points? doesn't really matter because we have the generative process, so we can just use the σ that we define

Modeling Practice

Modelling the Question

How-to-model guide.
Implementing the model

5. Toolkit selection
6. Planning the model
7. Implementing the model

Model testing

8. Completing the model
9. Testing and evaluating the model

Publishing

10. Publishing models

References

1. Blohm G, Kording KP, Schrater PR (2020). *A How-to-Model Guide for Neuroscience* eNeuro, 7(1). https://doi.org/10.1523/ENEURO.0352-19.2019
2. Mensh B, Kording K (2017). *Ten simple rules for structuring papers.* PLOS Comput Biol 13(9): e1005619. https://doi.org/10.1371/journal.pcbi.1005619

Modeling Practice

Step 5 - Selecting the toolkit

In selecting the right toolkit, i.e. the right mathematics, computer science, engineering, or physics, etc approaches, you should consider the following important rules:

1. Determine the level of abstraction
2. Select the toolkit that best represents the ingredients and hypotheses
3. Toolkit should express all needed relationships, mechanisms and concepts
4. Keep it as simple as possible!

Guiding questions:
What is the most appropriate approach to answer your question?

- What level of abstraction is needed?
- Determine granularity / scale based on hypotheses & goals
- Stay as high-level as possible, but be as detailed as needed!!!

Select the toolkit

- Requires prior knowledge about flexibility / limitations of toolkit
- Often more than one option possible
- Some toolkits are more flexible, span a wider range of behaviour and/or are lumpable
- Also determines how the model will be solved, i.e. simulated

Viewing modeling as a decision process] might help providing clarity regarding different model types, and how framing the problem and stating your goals influences the toolkit selection. Don't be afraid to pursue goals that no one else pursues; diversity of models should be encouraged because it results in complementary model considerations.

Step 5 - Pitfalls

Choosing a toolkit for the toolkit's sake (e.g. DL because it's cool to do deep learning)

- this will prevent you to really answer your research question and/or speak to your hypotheses

Being at the wrong level of abstraction (see W1D1 intro)

- too complex toolkits will have too many parameters you can't meaningfully constrain, and/or that add needless complexity
- too simple toolkits will lack means to implement the details you care about

Not knowing any toolkits

- this highlights a lack of literature review and/or background work to learn about the tools used by the field

Modeling Practice

Step 6 - Planning / drafting the model

Planning the model involves thinking about the general outline of the model, its components and how they might fit together. You want to draw a model diagram, make some sketches and formalize necessary equations. This step will thus outline a plan of implementation. Once you have that plan, this will hugely facilitate the actual implementation of the model in computer code.
Your model will have:

- **inputs**: the values the system has available - this can be broken down into data, and parameters
- **outputs**: these are the predictions our model will make that you could potentially measure (e.g. in your idealized experiment)
- **model functions**: A set of functions that perform the hypothesized computations.

You will thus need to define a set of functions that take your data and some parameters as input, can run your model, and output a prediction for a hypothetical measurement.
Guiding principles:
Keep it as simple as possible!
Don't get lost in details
Draft on paper: start with a flow diagram

- Draw out model components (boxes)
- What influences what? (arrows)

Then consider each model box separately

- Draft internal workings in terms of equations
- This might require a lot of work…
- Relate box inputs to box outputs!
- Keep in mind that the model should include a way to relate model variables to measurements
- Use the question, ingredients, and hypotheses to ensure you have all required components

Goal: Put in place all the components of the hypothesized relationships and explanations.

Step 6 - Pitfalls

I don't need to draft the model, I have it clearly in my mind

- you might think you do, but experience shows you're likely missing many important aspects

I can just make a rough draft

- the more detailed the draft, the easier it will be to implement the model in computer code
- rough drafts tend to forget important details that you need to think about, e.g. signals needed (where do they come from?), parameters to specify (how to constrain them?), etc.

Draft is too detailed or not detailed enough

- too detailed: you're writing our recursions, etc
- not detailed enough: you have no idea what's inside "boxes"

Modeling Practice

Step 7 - Implementing the model

This is the step where you finally start writing code! Separately implement each box, icon, or flow relationship identified in Step 6. **Test** each of those model components separately! (This is called a *unit test*). Unit testing ensures that each model components works are expected/planned.
Guiding principles:
Start with the easiest possible implementation

- Test functionality of model after each step before adding new model components (unit tests)
- Simple models can sometimes accomplish surprisingly much…

Add / remove different model elements

- Gain insight into working principles
- What's crucial, what isn't?
- Every component of the model must be crucial!

Make use of tools to evaluate model behavior

- E.g. graphical analysis, changing parameter sets, stability / equilibrium analyses, derive general solutions, asymptotes, periodic behaviour, etc.

Goal: Understand how each model component works in isolation and what the resulting overall model behavior is.

Step 7 - Pitfalls

Building the whole model at once without testing components

- you will make mistakes. Debug model components as you go!
- debugging a complex model is close to impossible. Is it not working because individual components are not working? Or do components not "play nice" together?

Not testing if individual components are important

- It's easy to add useless components to a model. They will be distracting for you and for readers

Not testing model functionality step by step as you build up the model

'd be surprised by what basic components often can alrealy achieve... E.g. our intuition is really bad when it comes to dynamical systems

Not using standard model testing tools

- each field has developped specific mathematical tools to test model behaviors. You'll be expected to show such evaluations. Make use of them early on!

Modeling Practice

Step 8 - Completing the model

Determining what you're done modeling is a hard question. Referring back to your original goals will be crucial. This is also where a precise question and specific hypotheses expressed in mathematical relationships come in handy.
Note: you can always keep improving our model, but at some point you need to decide that it is finished. Once you have a model that displays the properties of a system you are interested in, it should be possible to say something about your hypothesis and question. Keeping the model simple makes it easier to understand the phenomenon and answer the research question.
Guiding principles:
Determine a criterion
Refer to steps 1 (goals) and 4 (hypotheses)

- Does the model answer the original question sufficiently?
- Does the model satisfy your own evaluation criteria?
- Does it speak to the hypotheses?
- Can the model produce the parametric relationships hypothesized in step 4?

Make sure the model can speak to the hypothesis. Eliminate all the parameters that do not speak to the hypothesis.

Goal: Determine if you can answer your original research question and related hypotheses to your satisfaction. If the original goal has not been met you need to go back to the drawing board!

Step 8 - Pitfalls

Forgetting to specify or use a criterion for model completion (in Step 1!)

- This is crucial for you not to get lost in an endless loop of model improvements

Thinking the model can answer your question / hypotheses without checking

- always check if all questions and hypotheses can be answered / tested
- you will fail on your own benchmarks if you neglect this

You think you should further improve the model

- This is only warranted if your model cannot answer your hypotheses / questions and/or meet your goals
- remember: you can always improve a model, but you want to focus on the question / hypotheses / goals at hand!

Modeling Practice

Step 9 - Testing and evaluating the model

Every models needs to be evaluated quantitatively. There are many ways to achieve that and not every model should be evaluated in the same way. Ultimately, model testing depends on what your goals are and what you want to get out of the model, e.g. qualitative vs quantitative fit to data.
Guiding principles:
By definition a model is always wrong!

- Determine upfront what is "right enough" for you

Ensure the explicit interfacing with current or future data

- model answers the questions/hypotheses/goals with a sufficient amount of detail

Quantitative evaluation methods (see also W1D3)

- Statistics: how well does the model fit data?
- Predictability: does the model make testable predictions?
- Breadth: how general is the model?

Comparison against other models (BIC, AIC, etc.)

- This is often not easy to do in a fair way… Be nice and respectful to other models. Does the model explain previous data? (this is called the subsumption principle in physics!)

A good model should provide insight that could not have been gained or would have been hard to uncover without the model
Remember, a model is a working hypotheses; a good model should be falsifiable!
Goal: You want to demonstrate that your model works well. You also want to make sure you can interpret the model's meaning and findings. I.e. what did the model allow you to learn that was not apparent from the data alone?

Step 9 - Pitfalls

Thinking your model is bad

- does it answer the question / hypotheses and meet your goals? Does it provide the level of explanation and insights you set out to gain? **Then it's probably good enough!**

Not providing any quantitative evaluation

- Just do it, it's something that's expected

Not thinking deeply about your model and what you can learn from it

- this is likely the most important pitfall. You want to learn as much as you can from a model, especially about aspects that you cannot gain from the data alone
- model interpretation can open new avenues for research and experimentation. That's part of why we want to model in the first place! **A model is a hypothesis!**

Modeling Practice

Step 10 - Publishing the model

Guiding principles:
Know your target audience!

- How much math details? How much explanation of math?
- What's the message?
- Should be experimentalists in most cases!!!
- Provide intuitive explanations, analogies, etc.
- Famous researcher: "a good modeller knows how to relate to experimentalists" Clearly describe what the goals, hypotheses and performance criteria were
- Prevents from false expectation of what the model should be doing

A graphical representation is worth 1000 words (or more)
Show model simulations in parallel to data

- Much more convincing!

Publish enough implementation details

- A good model has to be reproducible!

Goal: Make sure your model is well received **AND USED** by the community. In order for our model to impact the field, it needs to be accepted by our peers, and order for that to happen it matters how the model is published.

Step 10 - Pitfalls

Not thinking of your target audience

- no one will understand you if you write for yourself because only you know what you know...

Forgetting about Steps 1-4

- clearly spelling out steps 1-4 allows the reader to appreciate your goals and sets limits on model criticism
  - no one can tell you your model didn't do something you never set out to do
  - prevents unreasonable claims / rejection of paper

Thinking you don't need figures to explain your model

- your model draft is a great starting point!
- make figures that provide intuition about model behavior (just like you would create figures to provide intuition about experimental data)

my code is too messy to be published

- not an option (many journal now rightfully require it)
- code cleanly right from the start
  - model drafting should help with that
  - comment your code! Then comment more...

Modeling Practice

Writing the abstract of a modeling paper

Abstracts are very stereotyped pieces of writing that contain highly condensed information. To write a summary (= abstract) of your modeling, you can use these questions as a guide:

- **What is the phenomena**? Here summarize the part of the phenomena which your modeling addresses.
- **What is the key scientific question?**: Clearly articulate the question which your modeling tries to answer.
- **What was our hypothesis?**: Explain the key relationships which we relied on to simulate the phenomena.
- **How did your modeling work?** Give an overview of the model, it's main components, and how the modeling works. "Here we ... "
- **What did you find? Did the modeling work?** Explain the key outcomes of your modeling evaluation.
- **What can you conclude?** Conclude as much as you can *with reference to the hypothesis*, within the limits of the modeling.
- **What are the limitations and future directions?** What is left to be learned? Briefly argue the plausibility of the approach and/or what you think is essential that may have been left out.

Instructions: write down your answer to each of those questions (1-2 sentences each, max!). When you're done, stick the sentences together... Now you have an abstract!

Modeling Practice

Guidance for paper writing

There are good guidelines for structuring and writing an effective paper (e.g., [Mensh Kording, 2017]), all of which apply to papers about models. There are some extra considerations when publishing a model. In general, you should explain each of the steps in the paper:

- **Introduction:** Steps 1 2 (maybe 3)
- **Methods:** Steps 3-7, 9
- **Results:** Steps 8 9, going back to 1, 2 4

Here are some great materials to help you with paper writing:

- Ten Simple Rules for Better Figures
- How to Write Strong and Effective Figure Legends
- How to Create a Consistent Writing Schedule: 10 Tips for Writers

The audience for all of this should be experimentalists, as they are the ones who can test predictions made by your model and collect new data. In this way your models can impact future experiments, and future data can then be modeled (see modeling process schematic below). Remember your audience - it is always hard to clearly convey the main points of your work to others, especially if your audience doesn't necessarily create computational models themselves.

In addition, you should provide a visualization of the model, and upload the code implementing the model and the data it was trained and tested on to a repository (e.g. GitHub and OSF).

Suggestion

For every modeling project, a very good exercise is to ***first*** write a short, 100-word abstract of the project plan and expected impact. This forces focussing on the main points: describing the relevance, question, model, answer and what it all means very succinctly. This allows you to decide to do this project or not **before you commit time writing code for no good purpose**. Notice that this is really what we've walked you through carefully in this guide!

Neuromatch Academy: Model Fitting - Summary Sheet[6]

Linear regression with MSE

Mean Squared Error (MSE)

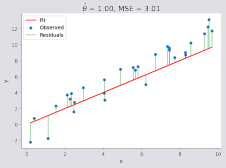
Linear least squares regression is an old but gold optimization procedure that we are going to use for data fitting. Least squares (LS) optimization problems are those in which the objective function is a quadratic function of the parameter(s) being optimized.
Suppose you have a set of measurements: for each data point or measurement, you have y_i (the "dependent" variable) obtained for a different input value, x_i (the "independent" variable). Suppose we believe the measurements are proportional to the input values, but are corrupted by some (random) measurement errors, ϵ_i, that is:

$$y_i = \theta x_i + \epsilon_i \tag{45}$$

for some unknown slope parameter θ. The least squares regression problem uses **mean squared error (MSE)** as its objective function, it aims to find the value of the parameter θ by minimizing the average of squared errors:

$$\min_\theta \frac{1}{N}\sum_{i=1}^{N}(\epsilon_i)^2 \tag{46}$$

$$\min_\theta \frac{1}{N}\sum_{i=1}^{N}(y_i - \theta x_i)^2 \tag{47}$$



Least-Squares Optimization

The MSE value relies on a grid of hand-specified values. If we didn't pick a good range to begin with, or with enough granularity, we might miss the best possible estimator. Instead of finding the minimum MSE from a set of candidate estimates, let's solve for it analytically. We can do this by minimizing the cost function. Mean squared error is a convex objective function, therefore we can compute its minimum using calculus for find the best estimate:

$$\hat{\theta} = \frac{\mathbf{x}^\top \mathbf{y}}{\mathbf{x}^\top \mathbf{x}} \tag{48}$$

where $\mathbf{x}$ and $\mathbf{y}$ are vectors of data points.

Linear regression with MLE

Gaussian noise

In the MSE we made the assumption that the data was drawn from a linear relationship with noise added.
In that case we treated the noise as simply a nuisance, but what if we factored it directly into our model?
The noise component ϵ is often modeled as a random variable drawn from a Gaussian distribution (also called the normal distribution).
The Gaussian distribution is described by its probability density function (pdf)

$$\mathcal{N}(x;\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2} \tag{49}$$

and is dependent on two parameters: the mean μ and the variance σ^2. We often consider the noise signal to be Gaussian "white noise", with zero mean and unit variance $\epsilon \sim \mathcal{N}(0,1)$.

Probabilistic Models

Consider again our simplified model $y = \theta x + \epsilon$ where the noise has zero mean and unit variance $\epsilon \sim \mathcal{N}(0,1)$. We can now also treat y as a random variable drawn from a Gaussian distribution where $\mu = \theta x$ and $\sigma^2 = 1$, $y \sim \mathcal{N}(\theta x, 1)$, which is to say that the probability of observing y given x and parameter θ is

$$p(y|x,\theta) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}(y-\theta x)^2} \tag{50}$$

Likelihood Estimation

Given the inherent uncertainty when dealing in probabilities, we talk about the likelihood that some estimate $\hat{\theta}$ fits our data. The likelihood function $\mathcal{L}(\theta)$ is equal to the probability density function parameterized by that θ:

$$\mathcal{L}(\theta|x,y) = p(y|x,\theta) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(y-\theta x)^2} \tag{51}$$

Since we have assumed that the noise affects each output independently, we can factorize the likelihood, and write:

$$\mathcal{L}(\theta|\mathbf{x},\mathbf{y}) = \prod_{i=1}^{N} \mathcal{L}(\theta|x_i,y_i), \tag{52}$$

where we have N data points $\mathbf{x} = [x_1,\ldots,x_N]$ and $\mathbf{y} = [y_1,\ldots,y_N]$.

Linear regression with MLE

Finding the Maximum Likelihood Estimator (MLE)

We want to find the parameter value $\hat{\theta}$ that makes our data set most likely:

$$\hat{\theta}_{\text{MLE}} = \underset{\theta}{\text{argmax}}\, \mathcal{L}(\theta|X,Y) \tag{53}$$

We discussed how taking the logarithm of the likelihood helps with numerical stability, the good thing is that it does so without changing the parameter value that maximizes the likelihood. Indeed, the $\log()$ function is "monotonically increasing", which means that it preserves the order of its inputs. So we have:

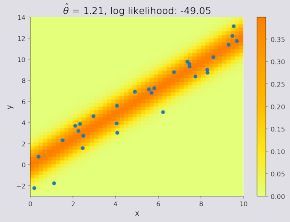
$$\hat{\theta}_{\text{MLE}} = \underset{\theta}{\text{argmax}} \sum_{i=1}^{m} \log \mathcal{L}(\theta|x_i,y_i) \tag{54}$$

Now substituting our specific likelihood function and taking its logarithm, we get:

$$\hat{\theta}_{\text{MLE}} = \underset{\theta}{\text{argmax}} [-\frac{N}{2}\log 2\pi\sigma^2 - \frac{1}{2\sigma^2}\sum_{i=1}^{N}(y_i - \theta x_i)^2]. \tag{55}$$

Note that maximizing the log likelihood is the same as minimizing the negative log likelihood (in practice optimization routines are developed to solve minimization not maximization problems). Because of the convexity of this objective function, we can take the derivative of our negative log likelihhood, set it to 0, and solve - just like our solution to minimizing MSE.

$$\frac{\partial \log \mathcal{L}(\theta|x,y)}{\partial\theta} = \frac{1}{\sigma^2}\sum_{i=1}^{N}(y_i - \theta x_i)x_i = 0 \tag{56}$$

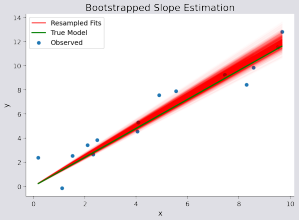


[6] 't Hart et al., (2022). Neuromatch Academy: a 3-week, online summer school in computational neuroscience. Journal of Open Source Education, 5(49), 118. https://doi.org/10.21105/jose.00118

Model Fitting - Confidence Intervals and Bootstrapping and Cross-validation

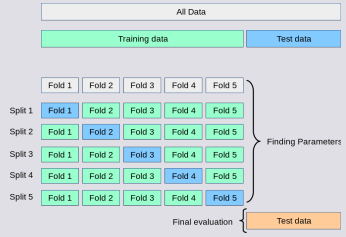
Confidence Intervals and Bootstrapping

Bootstrapping is a resampling procedure that allows to build confidence intervals around inferred parameter values. It is a widely applicable and very practical method that relies on computational power and pseudo-random number generators (as opposed to more classical approaches than depend on analytical derivations)
The idea is to generate many new synthetic datasets from the initial true dataset by randomly sampling from it, then finding estimators for each one of these new datasets, and finally looking at the distribution of all these estimators to quantify our confidence.
Note that each new resampled datasets will be the same size as our original one, with the new data points sampled with replacement i.e. we can repeat the same data point multiple times.



Cross-validation

A commonly used method for model selection is to ask how well the model predicts new data that it hasn't seen yet. But we don't want to use test data to do this, otherwise that would mean using it during the training process! One approach is to use another kind of held-out data which we call **validation data**: we do not fit the model with this data but we use it to select our best model.
We often have a limited amount of data though (especially in neuroscience), so we do not want to further reduce our potential training data by reassigning some as validation. Luckily, we can use **k-fold cross-validation**! In k-fold cross validation, we divide up the training data into k subsets (that are called *folds*, see diagram below), train our model on the first k-1 folds, and then compute error on the last held-out fold.



Multiple Linear Regression and Polynomial Regression

Multiple Linear Regression

We can easily extend univariate regression to the multivariate scenario by adding another parameter for each additional feature

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_d x_d + \epsilon \quad (57)$$

where θ_0 is the intercept and d is the number of features (it is also the dimensionality of our input).
We can condense this succinctly using vector notation for a single data point

$$y_i = \boldsymbol{\theta}^\top \mathbf{x}_i + \epsilon \quad (58)$$

and fully in matrix form

$$\mathbf{y} = \mathbf{X}\boldsymbol{\theta} + \epsilon \quad (59)$$

where $\mathbf{y}$ is a vector of measurements, $\mathbf{X}$ is a matrix containing the feature values (columns) for each input sample (rows), and $\boldsymbol{\theta}$ is our parameter vector. This matrix $\mathbf{X}$ is often referred to as the design matrix. To find an optimal vector of parameters $\hat{\boldsymbol{\theta}}$ we use:

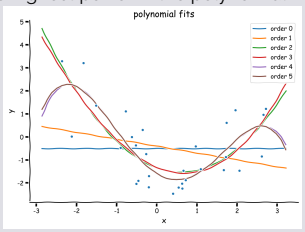
$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}. \quad (60)$$

Polynomial Regression

The polynomial regression is an extension of linear regression, the dependent variable y given the input values x. The key change is the type of relationship between inputs and outputs that the model can capture. With polynomial regression, we model the outputs as a polynomial equation based on the inputs. For example, we can model the outputs as:

$$y = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \epsilon \quad (61)$$

We can change how complex a polynomial is fit by changing the order of the polynomial. The order of a polynomial refers to the highest power in the polynomial.



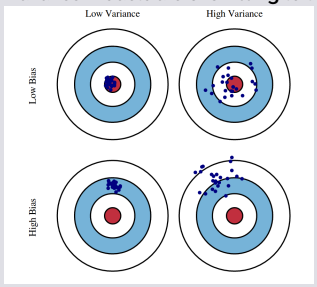
Model Selection: Bias-variance trade-off

Train and Test

The data used for the fitting procedure for a given model is the **training data**. We computed MSE on the training data of our polynomial regression models and compared training MSE across models. An additional important type of data is **test data**. This is held-out data that is not used (in any way) during the fitting procedure. When fitting models, we often want to consider both the train error (the quality of prediction on the training data) and the test error (the quality of prediction on the test data).

Bias-Variance Tradeoff

Finding a good model can be difficult. One of the most important concepts to keep in mind when modeling is the **bias-variance tradeoff**.
Bias is the difference between the prediction of the model and the corresponding true output variables you are trying to predict. Models with high bias will not fit the training data well since the predictions are quite different from the true data. These high bias models are overly simplified - they do not have enough parameters and complexity to accurately capture the patterns in the data and are thus **underfitting**.
Variance refers to the variability of model predictions for a given input. Essentially, do the model predictions change a lot with changes in the exact training data used? Models with high variance are highly dependent on the exact training data used - they will not generalize well to test data. These high variance models are **overfitting** to the data.



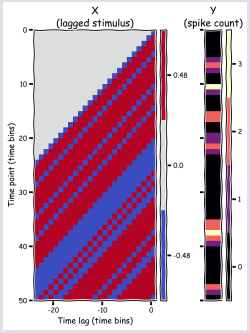
Neuromatch Academy: Generalised Linear Models - Summary Sheet[7]

Generalized Linear Models

Create design matrix

To create the **design matrix** which organizes the stimulus intensities in matrix form such that the ith row has the stimulus frames preceding timepoint i.

In this example, we will create the design matrix $\mathbf{X}$ using $d = 25$ time lags. That is, $\mathbf{X}$ should be a $T \times d$ matrix. $d = 25$ is a choice we're making based on our prior knowledge of the temporal window that influences RGC responses. Here, spike count is $\mathbf{Y}$ is predicted from out stimulus $\mathbf{X}$,

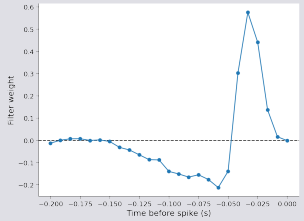


Fit Linear-Gaussian regression model

The maximum likelihood estimate of θ in this model can be solved analytically using the equation:

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}. \tag{62}$$

The resulting maximum likelihood filter estimates are:



Generalized Linear Models

Poisson regression

Poisson regression is a generalized linear model form of regression analysis used to model count data, like spikes. In the Poisson GLM,

$$\log P(\mathbf{y} \mid \mathbf{X}, \theta) = \sum_t \log P(y_t \mid \mathbf{x_t}, \theta), \tag{63}$$

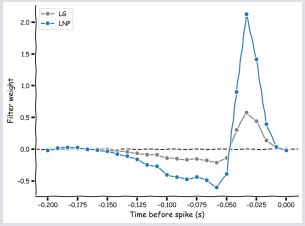
where

$$P(y_t \mid \mathbf{x_t}, \theta) = \frac{\lambda_t^{y_t} \exp(-\lambda_t)}{y_t!}, \text{with rate } \lambda_t = \exp(\mathbf{x_t}^\top \theta). \tag{64}$$

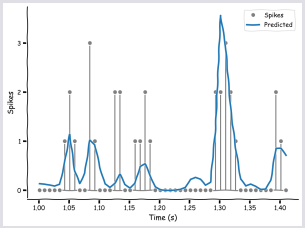
Now, taking the log likelihood for all the data we obtain: $\log P(\mathbf{y} \mid X, \theta) = \sum_t (y_t \log(\lambda_t) - \lambda_t - \log(y_t!))$. Because we are going to minimize the negative log likelihood with respect to the parameters θ, we can ignore the last term that does not depend on θ. For faster implementation, let us rewrite this in matrix notation:

$$\mathbf{y}^\top \log(\lambda) - \mathbf{1}^\top \lambda, \text{with rate } \lambda = \exp(\mathbf{X}\theta) \tag{65}$$

Finally, don't forget to add the minus sign for your function to return the negative log likelihood.



Spike Prediction



Generalized Linear Models

Logistic regression

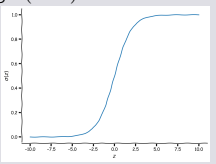
Logistic Regression is a binary classification model. It is a GLM with a logistic link function and a Bernoulli (i.e. coin-flip) noise model. The fundamental input/output equation of logistic regression is:

$$\hat{y} \equiv p(y = 1|x, \theta) = \sigma(\theta^T x) \tag{66}$$

Note that we interpret the output of logistic regression, $\hat{y}$, as the probability that $y = 1$ given inputs x and parameters θ.
Here $\sigma()$ is a "squashing" function called the sigmoid function or logistic function. Its output is in the range $0 \leq y \leq 1$. It looks like this:

$$\sigma(z) = \frac{1}{1 + \exp(-z)} \tag{67}$$

Recall that $z = \theta^T x$. The parameters decide whether $\theta^T x$ will be very negative, in which case $\sigma(\theta^T x) \approx 0$, or very positive, meaning $\sigma(\theta^T x) \approx 1$.



Regularisation

Regularization forces a model to learn a set solutions you a priori believe to be more correct, which reduces over-fitting because it doesn't have as much flexibility to fit idiosyncrasies in the training data. This adds model bias, but it's a good bias because you know (maybe) that parameters should be small or mostly 0.
L_2 regularization
Regularization comes in different flavors. A very common one uses an L_2 or "ridge" penalty. This changes the objective function to

$$-\log \mathcal{L}'(\theta|X, y) = -\log \mathcal{L}(\theta|X, y) + \frac{\beta}{2} \sum_i \theta_i^2, \tag{68}$$

where β is a *hyperparameter* that sets the *strength* of the regularization.

[7] 't Hart et al., (2022). Neuromatch Academy: a 3-week, online summer school in computational neuroscience. Journal of Open Source Education, 5(49), 118. https://doi.org/10.21105/jose.00118

Generalized Linear Models

Logistic regression

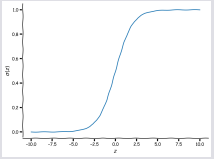
Logistic Regression is a binary classification model. It is a GLM with a logistic link function and a Bernoulli (i.e. coin-flip) noise model. The fundamental input/output equation of logistic regression is:

$$\hat{y} \equiv p(y = 1|x, \theta) = \sigma(\theta^T x) \tag{69}$$

Note that we interpret the output of logistic regression, $\hat{y}$, as the probability that $y = 1$ given inputs x and parameters θ.
Here $\sigma()$ is a "squashing" function called the sigmoid function or logistic function. Its output is in the range $0 \leq y \leq 1$. It looks like this:

$$\sigma(z) = \frac{1}{1 + \exp(-z)} \tag{70}$$

Recall that $z = \theta^T x$. The parameters decide whether $\theta^T x$ will be very negative, in which case $\sigma(\theta^T x) \approx 0$, or very positive, meaning $\sigma(\theta^T x) \approx 1$.



Regularisation

Regularization forces a model to learn a set solutions you a priori believe to be more correct, which reduces over-fitting because it doesn't have as much flexibility to fit idiosyncrasies in the training data. This adds model bias, but it's a good bias because you know (maybe) that parameters should be small or mostly 0.
L_2 regularization
Regularization comes in different flavors. A very common one uses an L_2 or "ridge" penalty. This changes the objective function to

$$-\log \mathcal{L}'(\theta|X, y) = -\log \mathcal{L}(\theta|X, y) + \frac{\beta}{2} \sum_i \theta_i^2, \tag{71}$$

where β is a *hyperparameter* that sets the *strength* of the regularization.

Neuromatch Academy: Dimensionality Reduction - Summary Sheet[8]

Dimensionality Reduction

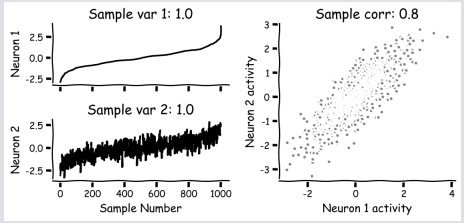
Generate Correlated Multivariate Data

Multivariate data can be visualized as a cloud of points in a high-dimensional vector space. The geometry of this cloud is shaped by the covariance matrix.
The covariance can be found from the equation above:

$$\text{cov}(x_1, x_2) = \rho\sqrt{\sigma_1^2\sigma_2^2}. \tag{72}$$

The covariance matrix for two dimensions has the following form:

$$\mathbf{\Sigma} = \begin{pmatrix} \text{var}(x_1) & \text{cov}(x_1, x_2) \\ \text{cov}(x_1, x_2) & \text{var}(x_2) \end{pmatrix}. \tag{73}$$



Orthonormal Basis

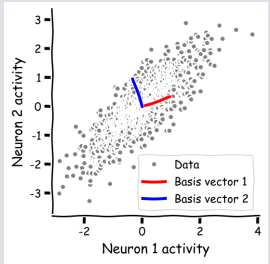
Two vectors are orthonormal if:

1. They are orthogonal (i.e., their dot product is zero):

$$\mathbf{u} \cdot \mathbf{w} = u_1 w_1 + u_2 w_2 = 0 \tag{74}$$

2. They have unit length:

$$||\mathbf{u}|| = ||\mathbf{w}|| = 1 \tag{75}$$



Principal Component Analysis

Eigenvectors of the Sample Covariance Matrix

PCA represents data in a new orthonormal basis defined by the eigenvectors of the covariance matrix. The bivariate normal data with a specified covariance matrix $\mathbf{\Sigma}$, whose (i, j)th element is:

$$\Sigma_{ij} = E[x_i x_j] - E[x_i]E[x_j]. \tag{76}$$

We use the sample covariance matrix, $\hat{\mathbf{\Sigma}}$, which is calculated directly from the data. The (i, j)th element of the sample covariance matrix is:

$$\hat{\Sigma}_{ij} = \frac{1}{N_{\text{samples}}} \mathbf{x}_i^T \mathbf{x}_j - \bar{\mathbf{x}}_i \bar{\mathbf{x}}_j, \tag{77}$$

where $\mathbf{x}_i = [x_i(1), x_i(2), \ldots, x_i(N_{\text{samples}})]^T$ is a column vector representing all measurements of neuron i, and $\bar{\mathbf{x}}_i$ is the mean of neuron i across samples:

$$\bar{\mathbf{x}}_i = \frac{1}{N_{\text{samples}}} \sum_{k=1}^{N_{\text{samples}}} x_i(k). \tag{78}$$

If we assume that the data has already been mean-subtracted, then we can write the sample covariance matrix in a much simpler matrix form:

$$\hat{\mathbf{\Sigma}} = \frac{1}{N_{\text{samples}}} \mathbf{X}^T \mathbf{X}. \tag{79}$$

where $\mathbf{X}$ is the full data matrix (each column of $\mathbf{X}$ is a different $\mathbf{x_i}$).

Principal Component Analysis

PCA by projecting data onto the eigenvectors

To perform PCA project the data onto the eigenvectors of the covariance matrix, i.e.:

$$\mathbf{S} = \mathbf{XW} \tag{80}$$

where $\mathbf{S}$ is an $N_{\text{samples}} \times N$ matrix representing the projected data (also called *scores*), and $\mathbf{W}$ is an $N \times N$ orthogonal matrix, each of whose columns represents the eigenvectors of the covariance matrix (also called *weights* or *loadings*). Since $\mathbf{W}$ is an orthogonal matrix, $\mathbf{W}^{-1} = \mathbf{W}^T$. So by multiplying by $\mathbf{W}^T$ on each side we can rewrite this equation as

$$\mathbf{X} = \mathbf{SW}^T. \tag{81}$$

To reconstruct the data from a low-dimensional approximation, we just have to truncate these matrices. Let's denote $\mathbf{S}_{1:K}$ and $\mathbf{W}_{1:K}$ the matrices with only the first K columns of $\mathbf{S}$ and $\mathbf{W}$, respectively. Then our reconstruction is:

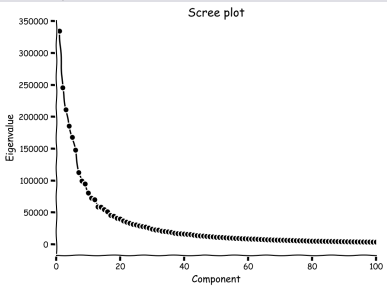
$$\hat{\mathbf{X}} = \mathbf{S}_{1:K}(\mathbf{W}_{1:K})^T. \tag{82}$$

[8] 't Hart et al., (2022). Neuromatch Academy: a 3-week, online summer school in computational neuroscience. Journal of Open Source Education, 5(49), 118. https://doi.org/10.21105/jose.00118

Dimensionality Reduction & Reconstruction

Scree Plot

Each eigenvalue describes the variance of the data projected onto its corresponding eigenvector. This is an important concept because it allows us to rank the PCA basis vectors based on how much variance each one can capture in a scree plot. A scree plot is used to help choose which and how many components to use:

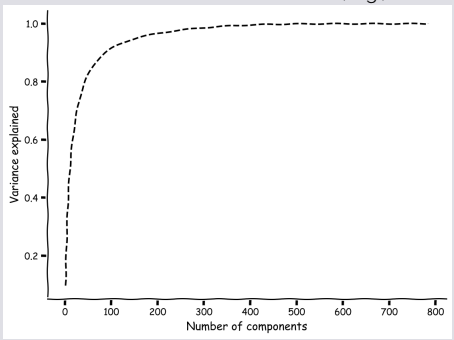


Calculate the variance explained

Another common way to determine the intrinsic dimensionality is by considering the variance explained. This can be examined with a cumulative plot of the fraction of the total variance explained by the top K components, i.e.,

$$\text{var explained} = \frac{\sum_{i=1}^{K} \lambda_i}{\sum_{i=1}^{N} \lambda_i} \tag{83}$$

where λ_i is the i^{th} eigenvalue and N is the total number of components (the original number of dimensions in the data). The intrinsic dimensionality is often quantified by the K necessary to explain a large proportion of the total variance of the data (often a defined threshold, e.g., 90%).

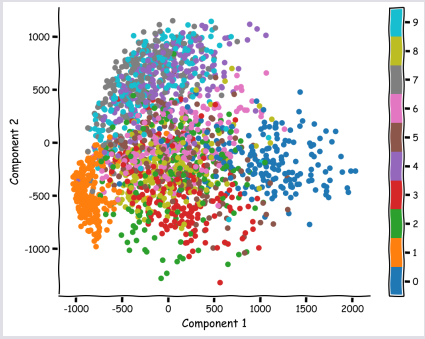


Nonlinear Dimensionality Reduction

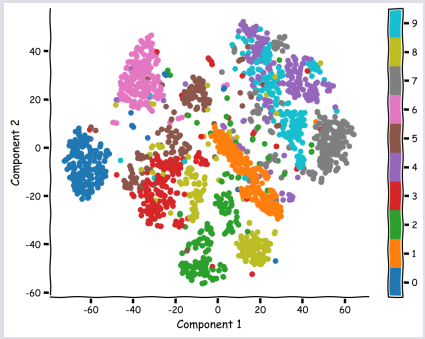
Unsupervised Reduction

PCA and t-SNE are unsupervised non-linear dimensionality reduction methods. Nonlinear methods can be more powerful, they can also be sensitive to noise. In contrast, linear methods are useful for their simplicity and robustness. Comparing PCA and t-SNE for data visualization. Using t-SNE, we could visualize clusters in the data corresponding to different digits. While PCA was able to separate some clusters (e.g., 0 vs 1), it performed poorly overall. However, the results of t-SNE can change depending on the choice of perplexity.

Visualize MNIST in 2D using PCA



Visualize MNIST in 2D using t-SNE



Neuromatch Academy: Deep Learning - Summary Sheet[9]

Neural Network

Deep feed-forward networks

We can build a linear network with no hidden layers, where the stimulus prediction y is a product of weights $\mathbf{W}_{out}$ and neural responses $\mathbf{r}$ with an added term $\mathbf{b}$ which is called the bias term. When you fit a linear model such as this you minimize the squared error between the predicted stimulus y and the true stimulus $\tilde{y}$, this is the "loss function".

$$L = (y - \tilde{y})^2 \tag{84}$$

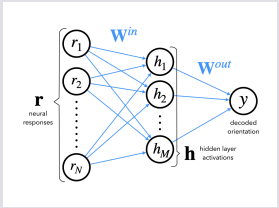
$$= ((\mathbf{W}^{out}\mathbf{r} + \mathbf{b}) - \tilde{y})^2 \tag{85}$$

The solution to minimizing this loss function in a linear model can be found in closed form. If we use a simple linear model for this data we are able to predict the stimulus within 2-3 degrees.
Let's add a hidden layer with M units to this linear model, where now the output y is as follows:

$$\mathbf{h} = \mathbf{W}^{in}\mathbf{r} + \mathbf{b}^{in}, \quad [\mathbf{W}^{in} : M \times N,\ \mathbf{b}^{in} : M \times 1], \tag{86}$$

$$y = \mathbf{W}^{out}\mathbf{h} + \mathbf{b}^{out}, \quad [\mathbf{W}^{out} : 1 \times M,\ \mathbf{b}^{in} : 1 \times 1], \tag{87}$$

The M-dimensional vector $\mathbf{h}$ denotes the activations of the *hidden layer* of the network. The blue components of this diagram denote the *parameters* of the network, which we will later optimize with gradient descent. These include all the weights and biases $\mathbf{W}^{in}, \mathbf{b}^{in}, \mathbf{W}^{out}, \mathbf{b}^{out}$. The *weights* are matrices of size (of outputs, of inputs) that are multiplied by the input of each layer, like the regression coefficients in linear regression.



Neural Network

Activation Functions

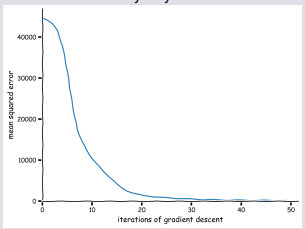
To extend the set of computable input/output transformations to more than just weighted sums, we'll incorporate a non-linear activation function in the hidden units. This is done by simply modifying the equation for the hidden layer activations to be

$$\mathbf{h}^{(n)} = \phi(\mathbf{W}^{in}\mathbf{r}^{(n)} + \mathbf{b}^{in}) \tag{88}$$

where ϕ is referred to as the activation function. Using a non-linear activation function will ensure that the hidden layer performs a non-linear transformation of the input, which will make our network much more powerful. In practice, deep networks always use non-linear activation functions. The most common non-linearity used is the rectified linear unit (or ReLU), which is a max(0, x) function.

Gradient Descent

In gradient descent we compute the gradient of the loss function with respect to each parameter (all W's and b's). We then update the parameters by subtracting the learning rate times the gradient.
Let's visualize this loss function L with respect to a weight w. If the gradient is positive (the slope $\frac{dL}{dw} > 0$) as in this case then we want to move in the opposite direction which is negative. So we update the w accordingly in the negative direction on each iteration. Once the iterations complete the weight will ideally be at a value that minimizes the cost function.
In reality these cost functions are not convex like this one and depend on hundreds of thousands of parameters. There are tricks to help navigate this rocky cost landscape such as adding momentum or changing the optimizer but we won't have time to get into that today. There are also ways to change the architecture of the network to improve optimization, such as including skip connections. These skip connections are used in residual networks and allow for the optimization of many layer networks.



Convolution Neural Network

Introduction to 2D convolutions

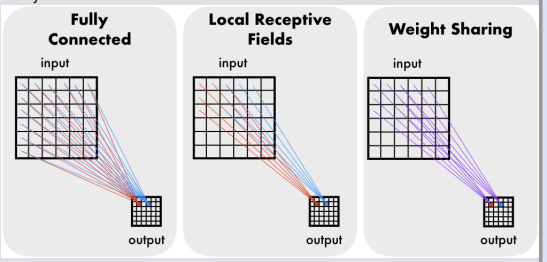
A 2D convolution is an integral of the product of a filter f and an input image I computed at various positions as the filter is slid across the input. The output of the convolution operation at position (x, y) can be written as follows, where the filter f is size (K, K):

$$C(x, y) = \sum_{k_x=-K/2}^{K/2} \sum_{k_y=-K/2}^{K/2} f(k_x, k_y) I(x+k_x, y+k_y) \tag{89}$$

This convolutional filter is often called a kernel.

Convolutional Layers

In a fully connected layer, each unit computes a weighted sum over all the input units. In a convolutional layer, on the other hand, each unit computes a weighted sum over only a small patch of the input, referred to as the unit's *receptive field*. When the input is an image, the receptive field can be thought of as a local patch of pixels.
In a fully connected layer, each unit uses its own independent set of weights to compute the weighted sum. In a convolutional layer, all the units (within the same channel) share the same weights. This set of shared weights is called the convolutional filter or kernel. The result of this computation is a convolution, where each unit has computed the same weighted sum over a different part of the input. This reduces the number of parameters in the network substantially.



[9] 't Hart et al., (2022). Neuromatch Academy: a 3-week, online summer school in computational neuroscience. Journal of Open Source Education, 5(49), 118. https://doi.org/10.21105/jose.00118

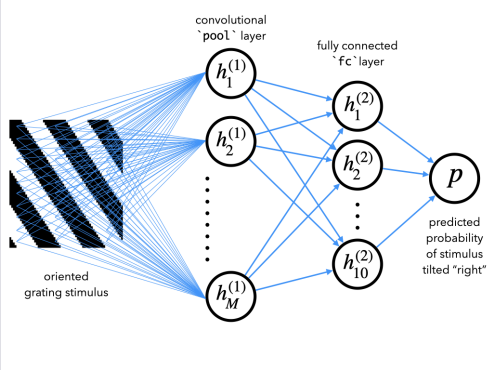
Building and Evaluating Normative Encoding Models

Setting up Deep Network and Neural Data

We will build our normative encoding model by optimizing its parameters to solve an orientation discrimination task.
To do this, we will use a convolutional neural network (CNN). Here, we will use a CNN that performs two-dimensional convolutions on the raw stimulus image (which is a 2D matrix of pixels), rather than one-dimensional convolutions on a categorical 1D vector representation of the stimulus. CNNs are commonly used for image processing.
The particular CNN we will use here has two layers:

1. a convolutional layer, which convolves the images with a set of filters
2. a fully connected layer, which transforms the output of this convolution into a 10-dimensional representation

Finally, a set of output weights transforms this 10-dimensional representation into a single scalar p, denoting the predicted probability of the input stimulus being tilted right.



Building and Evaluating Normative Encoding Models

Representational Dissimilarity matrix (RDM)

We noticed above some similarities and differences between the population responses in the mouse primary visual cortex and in different layers in our model. To quantify this we'll use a technique called Representational Similarity Analysis. The idea is to look at the similarity structure between representations of different stimuli. We can say that a brain area and a model use a similar representational scheme if stimuli that are represented (dis)similarly in the brain are represented (dis)similarly in the model as well.
We begin by computing the representational dissimilarity matrix (RDM) for the mouse V1 data and each model layer. This matrix, which we'll call $\mathbf{M}$, is computed as one minus the correlation coefficients between population responses to each stimulus. We can efficiently compute this by using the z-scored responses.
The z-scored response of all neurons $\mathbf{r}$ to stimulus s is the response mean-subtracted across neurons i and normalized to standard deviation 1 across neurons i where N is the total number of neurons:

$$\mathbf{z}^{(s)} = \frac{\mathbf{r}^{(s)} - \mu^{(s)}}{\sigma^{(s)}} \tag{90}$$

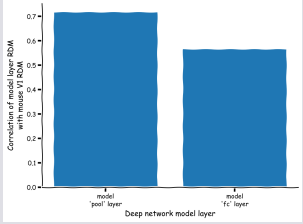
where $\mu^{(s)} = \frac{1}{N}\sum_{i=1}^{N} r_i^{(s)}$ and $\sigma^{(s)} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(r_i^{(s)} - \mu^{(s)}\right)^2}$.
Then the full matrix can be computed as:

$$\mathbf{M} = 1 - \frac{1}{N}\mathbf{Z}\mathbf{Z}^T \tag{91}$$

where $\mathbf{Z}$ is the z-scored response matrix with rows $\mathbf{r}^{(s)}$ and N is the number of neurons (or units).

Determining Representation Similarity

To quantify how similar the representations are, we can simply correlate their dissimilarity matrices. For this, we'll again use the correlation coefficient. Note that dissimilarity matrices are symmetric ($M_{ss'} = M_{s's}$), so we should only use the off-diagonal terms on one side of the diagonal when computing this correlation to avoid overcounting. Moreover, we should leave out the diagonal terms, which are always equal to 0, so will always be perfectly correlated across any pair of RDM's.



Building and Evaluating Normative Encoding Models

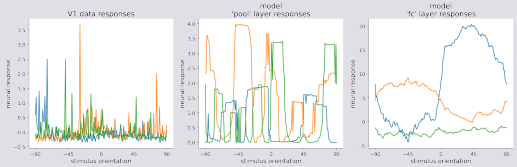
Qualitative Comparisons of CNNs and Neural Activity

To visualize the representations in the data and in each of these model layers, we'll use two classic techniques from systems neuroscience:

1. **tuning curves**: plotting the response of single neurons (or units, in the case of the deep network) as a function of the stimulus orientation
2. **dimensionality reduction**: plotting full population responses to each stimulus in two dimensions via dimensionality reduction. We'll use the non-linear dimensionality reduction technique t-SNE for this. We use dimensionality reduction because there are many units and it's difficult to visualize all of them at once. We use a non-linear dimensionality reduction technique because it can capture complex relationships between stimuli.

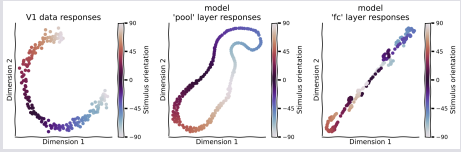
Tuning Curves

Below, we show some example tuning curves for different neurons and units in the trained CNN.



Dimensionality Reduction of Representations

We can visualize a dimensionality-reduced version of the internal representations of the mouse primary visual cortex or CNN internal representations in order to potentially uncover informative structure. Here, we use PCA to reduce the dimensionality to 20 dimensions, and then use tSNE to further reduce dimensionality to 2 dimensions.



Neuromatch Academy: Linear Systems - Summary Sheet[10]

Linear Dynamical Systems

One-dimensional Differential Equations

Differential equations are equations that express the **rate of change** of the state variable x. One typically describes this rate of change using the derivative of x with respect to time (dx/dt) on the left hand side of the differential equation:

$$\frac{dx}{dt} = f(x)$$

A common notational short-hand is to write $\dot{x}$ for $\frac{dx}{dt}$. The dot means "the derivative with respect to time". We can simulate an ordinary differential equation by approximating or modeling time as a discrete list of time steps $t_0, t_1, t_2, \ldots$, such that $t_{i+1} = t_i + dt$. We can get the small change dx over a small duration dt of time from the definition of the differential:

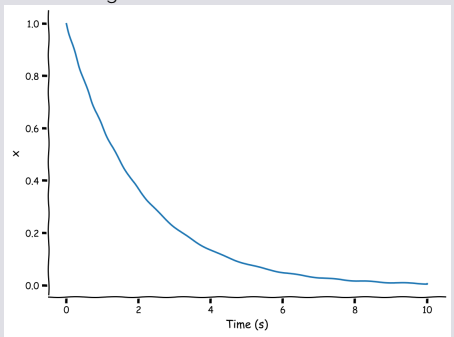
$$\dot{x} = \frac{dx}{dt} \quad (92)$$

$$dx = \dot{x}\, dt \quad (93)$$

So, at each time step t_i, we compute a value of x, $x(t_i)$, as the sum of the value of x at the previous time step, $x(t_{i-1})$ and a small change $dx = \dot{x}\, dt$:

$$x(t_i) = x(t_{i-1}) + \dot{x}(t_{i-1})dt$$

This very simple integration scheme, known as **forward Euler integration**, works well if dt is small and the ordinary differential equation is simple.
The solution of the differential equation $\dot{x} = ax$ using forward Euler integration is:



Linear Dynamical Systems

Multi-Dimensional Dynamics

Adding one additional variable (or dimension) adds more variety of behaviors. Additional variables are useful in modeling the dynamics of more complex systems with richer behaviors, such as systems of multiple neurons. We can write such a system using two linear ordinary differential equations:

$$\dot{x}_1 = a_{11}x_1 \quad (94)$$

$$\dot{x}_2 = a_{22}x_2 \quad (95)$$

So far, this system consists of two variables (e.g. neurons) in isolation. To make things interesting, we can add interaction terms:

$$\dot{x}_1 = a_{11}x_1 + a_{12}x_2 \quad (96)$$

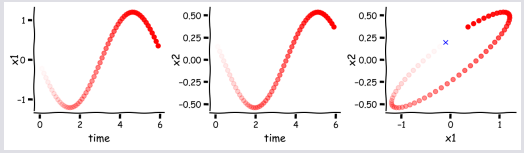
$$\dot{x}_2 = a_{21}x_1 + a_{22}x_2 \quad (97)$$

We can write the two equations that describe our system as one (vector-valued) linear ordinary differential equation:

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}$$

For two-dimensional systems, $\mathbf{x}$ is a vector with 2 elements (x_1 and x_2) and $\mathbf{A}$ is a 2×2 matrix with

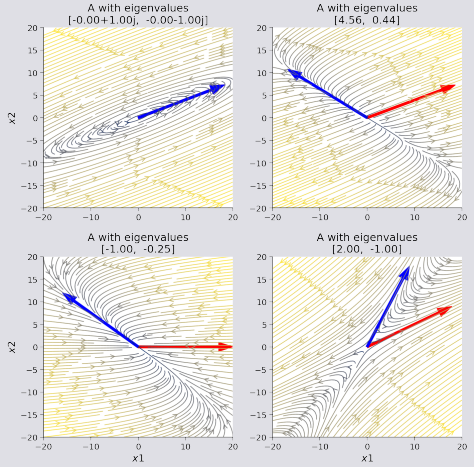
$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}.$$



Linear Dynamical Systems

Stream Plots

It's a bit tedious to plot trajectories one initial condition at a time!
Fortunately, to get an overview of how a grid of initial conditions affect trajectories of a system, we can use a *stream plot*.
We can think of a initial condition $\mathbf{x}_0 = (x_{1_0}, x_{2_0})$ as coordinates for a position in a space. For a 2x2 matrix $\mathbf{A}$, a stream plot computes at each position $\mathbf{x}$ a small arrow that indicates $\mathbf{A}\mathbf{x}$ and then connects the small arrows to form *stream lines*. Remember from the beginning of this tutorial that $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}$ is the rate of change of $\mathbf{x}$. So the stream lines indicate how a system changes. If you are interested in a particular initial condition $\mathbf{x}_0$, just find the corresponding position in the stream plot. The stream line that goes through that point in the stream plot indicates $\mathbf{x}(t)$.
Using some helper functions, we show the stream plots for each option of A that you examined in the earlier interactive demo. We included the eigenvectors of $\mathbf{A}$ as a red line (1st eigenvalue) and a blue line (2nd eigenvalue) in the stream plots.



[10] 't Hart et al., (2022). Neuromatch Academy: a 3-week, online summer school in computational neuroscience. Journal of Open Source Education, 5(49), 118. https://doi.org/10.21105/jose.00118

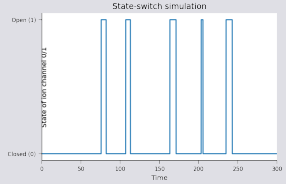
Markov Processes

Introduction

We will now look at **probabilistic** dynamical systems. You may sometimes hear these systems called *stochastic*. In a probabilistic process, elements of randomness are involved. Every time you observe some probabilistic dynamical system, starting from the same initial conditions, the outcome will likely be different. Put another way, dynamical systems that involve probability will incorporate random variations in their behavior.
For some probabilistic dynamical systems, the differential equations express a relationship between $\dot{x}$ and x at every time t, so that the direction of x at every time depends entirely on the value of x. Said a different way, knowledge of the value of the state variables x at time t is all the information needed to determine $\dot{x}$ and therefore x at the next time.
This property that the present state entirely determines the transition to the next state is what defines a **Markov process** and systems obeying this property can be described as **Markovian**.

Telegraph Process

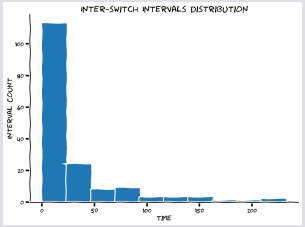
Let's consider a Markov process with two states, where switches between each two states are probabilistic (known as a telegraph process). To be concrete, let's say we are modeling an *ion channel in a neuron that can be in one of two states: Closed (0) or Open (1).*
If the ion channel is Closed, it may transition to the Open state with probability $P(0 \to 1|x = 0) = \mu_{c2o}$. Likewise, If the ion channel is Open, it transitions to Closed with probability $P(1 \to 0|x = 1) = \mu_{o2c}$.
We simulate the process of changing states as a Poisson process. The Poisson process is a way to model discrete events where the average time between event occurrences is known but the exact time of some event is not known. Importantly, the Poisson process dictates the following points:
1. The probability of some event occurring is independent from all other events.
2. The average rate of events within a given time period is constant.
3. Two events cannot occur at the same moment. Our ion channel can either be in an open or closed state, but not both simultaneously.
In the plot below, we will use the Poisson process to model the state of our ion channel at all points t within the total simulation time T. We also track at which times throughout the simulation the state makes a switch.



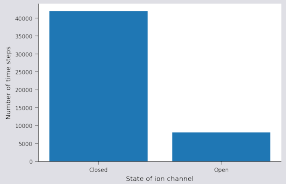
Markov Processes

Telegraph Process

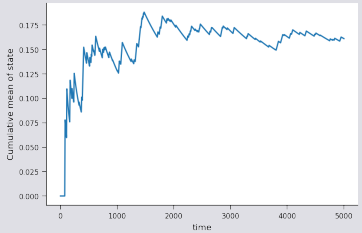
We now have *switch times*, which is a list consisting of times when the state switched. Using this, calculate the time intervals between each state switch and store these in a list called *inter switch intervals*, plotted below.



We generate a bar graph to visualize the distribution of the number of time-steps spent in each of the two possible system states during the simulation.



Even though the state is discrete –the ion channel can only be either Closed or Open– we can still look at the *mean state* of the system, averaged over some window of time.
Since we've coded Closed as $x = 0$ and Open as $x = 1$, conveniently, the mean of x over some window of time has the interpretation of *fraction of time channel is Open*.
Let's also take a look at the fraction of Open states as a cumulative mean of the state x. The cumulative mean tells us the average number of state-changes that the system will have undergone after a certain amount of time.



Markov Processes

Distributional Perspective

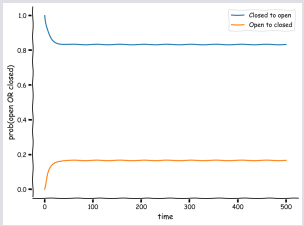
We can run a simulation many times and gather empirical distributions of open/closed states. Alternatively, we can formulate the exact same system probabilistically, keeping track of the probability of being in each state.
The same system of transitions can then be formulated using a vector of 2 elements as the state vector and a dynamics matrix $\mathbf{A}$. The result of this formulation is a "state transition matrix":

$$\begin{bmatrix} C \\ O \end{bmatrix}_{k+1} = \mathbf{A} \begin{bmatrix} C \\ O \end{bmatrix}_k = \begin{bmatrix} 1-\mu_{c2o} & \mu_{o2c} \\ \mu_{c2o} & 1-\mu_{o2c} \end{bmatrix} \begin{bmatrix} C \\ O \end{bmatrix}_k.$$

Each transition probability shown in the matrix is as follows:

1. $1 - \mu_{c2o}$, the probability that the closed state remains closed.
2. μ_{c2o}, the probability that the closed state transitions to the open state.
3. μ_{o2c}, the probability that the open state transitions to the closed state.
4. $1-\mu_{o2c}$, the probability that the open state remains open.

Notice that this system is written as a discrete step in time, and $\mathbf{A}$ describes the transition, mapping the state from step k to step $k+1$. This is different from what we did in the exercises above where $\mathbf{A}$ had described the function from the state to the time derivative of the state.



Equilibrium of the telegraph process

Since we have now modeled the propagation of probabilities by the transition matrix $\mathbf{A}$ in Section 2, let's connect the behavior of the system at equilibrium with the eigendecomposition of $\mathbf{A}$. The eigenvalues of $\mathbf{A}$ tell us about the stability of the system, specifically in the directions of the corresponding eigenvectors.

Combining Determinism and Stochasticity

Introduction

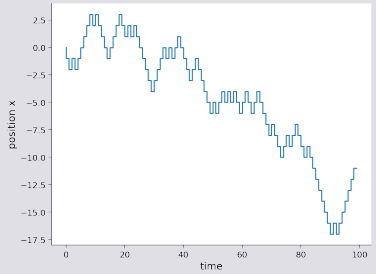
We've spent some time familiarizing ourselves with the behavior of such systems when their trajectories are (1) entirely predictable and deterministic, or (2) governed by random processes, it's time to consider that neither is sufficient to describe neuroscience. Instead, we are often faced with processes for which we know some dynamics, but there are some random aspects as well. We call these **dynamical systems with stochasticity**.

Random Walks

To begin, let's first take a gander at how life sometimes wanders around aimlessly. One of the simplest and best-studied living systems that has some interesting behaviors is the *E. coli* bacterium, which is capable of navigating odor gradients on a substrate to seek a food source. Larger life (including flies, dogs, and blindfolded humans) sometimes use the same strategies to guide their decisions.
Here, we will consider what the *E. coli* does in the absence of food odors. What's the best strategy when one does not know where to head? Why, flail around randomly, of course!
The **random walk** is exactly that — at every time step, use a random process like flipping a coin to change one's heading accordingly. Note that this process is closely related to **Brownian motion**, so you may sometimes hear that terminology used as well.
Let's start with a **one-dimensional random walk**. A bacterium starts at $x = 0$. At every time step, it flips a coin (a very small, microscopic coin of protein mintage), then heads left $\Delta x = -1$ or right $\Delta x = +1$ for with equal probability. For instance, if at time step 1 the result of the coin flip is to head right, then its position at that time step becomes $x_1 = x_0 + \Delta x = 1$. Continuing in this way, its position at time step $k + 1$ is given by

$$x_{k+1} = x_k + \Delta x$$

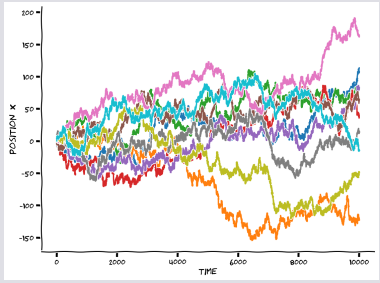
We will simulate this process below and plot the position of the bacterium as a function of the time step.



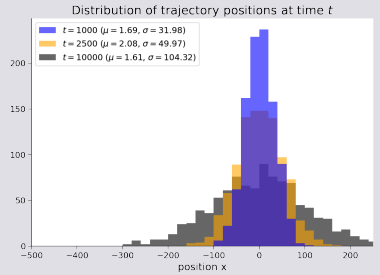
Combining Determinism and Stochasticity

Random Walks Simulations

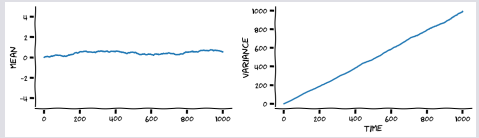
We will plot 10 random walks for 10000 time steps each where the steps have a standard normal distribution (with mean μ and standard deviation σ).



We see that the trajectories all look a little different from each other. But there are some general observations one can make: at the beginning almost all trajectories are very close to $x = 0$, which is where our bacterium started. As time progresses, some trajectories move further and further away from the starting point. However, a lot of trajectories stay close to the starting point of $x = 0$.
Now let's take a look in the next cell at the distribution of bacteria positions at different points in time, analyzing all the trajectories from above.



The plot of the mean and variance of our bacterium's random walk as a function of time.



Combining Determinism and Stochasticity

The Ornstein-Uhlenbeck (OU) process

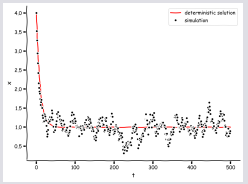
Our goal is now to build on this model to construct a **drift-diffusion** model (DDM). DDM is a popular model for memory, which as we all know, is often an exercise in hanging on to a value imperfectly. Decision-making and memory will be the topic for tomorrow, so here we build the mathematical foundations and develop some intuition for how such systems behave!
To build such a model, let's combine the random walk model with the first order discrete differential equation. We have to modify our analytic solution of the differential equation slightly:

$$x_k = x_\infty(1 - \lambda^k) + x_0\lambda^k.$$

The dynamics of this process starts at x_0 and decay towards x_∞.
Now we are ready to take this basic, deterministic difference equation and add a diffusion process on top of it!
As a point of terminology: this type of process is commonly known as a **drift-diffusion model** or **Ornstein-Uhlenbeck (OU) process**. The model is a combination of a drift term toward x_∞ and a diffusion term that walks randomly. You may sometimes see them written as continuous stochastic differential equations, but here we are doing the discrete version to maintain continuity in the tutorial. The discrete version of our OU process has the following form:

$$x_{k+1} = x_\infty + \lambda(x_k - x_\infty) + \sigma\eta$$

where η is sampled from a standard normal distribution ($\mu = 0, \sigma = 1$).



Variance of the OU process

As we can see, the *mean* of the process follows the solution to the deterministic part of the governing equation. But what about the variance?
Unlike the random walk, because there's a decay process that "pulls" x back towards x_∞, the variance does not grow without bound with large t. Instead, when it gets far from x_∞, the position of x is restored, until an equilibrium is reached.
The equilibrium variance for our drift-diffusion system is

$$\text{Var} = \frac{\sigma^2}{1 - \lambda^2}.$$

Notice that the value of this equilibrium variance depends on λ and σ. It does not depend on x_0 and x_∞.

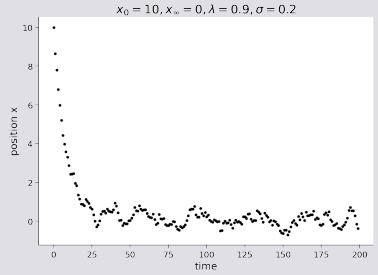
Autoregressive models

Fitting data to the OU process

Our process the drift-diffusion (OU) process had following form:

$$x_{k+1} = x_\infty + \lambda(x_k - x_\infty) + \sigma\eta$$

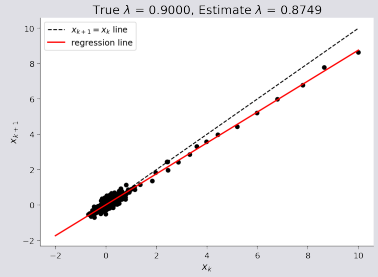
where η is sampled from a standard normal distribution. For simplicity, we set $x_\infty = 0$. Let's plot a trajectory for this process again below. Take note of the parameters of the process because they will be important later.



What if we were given these positions x as they evolve in time as data, how would we get back out the dynamics of the system λ?
Since a little bird told us that this system takes on the form

$$x_{k+1} = \lambda x_k + \eta,$$

where η is noise from a normal distribution, our approach is to solve for λ as a **regression problem**.
As a check, let's plot every pair of points adjacent in time (x_{k+1} vs. x_k) against each other to see if there is a linear relationship between them.
Hooray, it's a line! This is evidence that the dynamics that generated the data is linear. We can now reformulate this task as a regression problem.



This model is **autoregressive**, where auto means self. In other words, it's a regression of the time series on itself from the past. The equation as written above is only a function of itself from one step in the past, so we can call it a first order autoregressive model and plot.

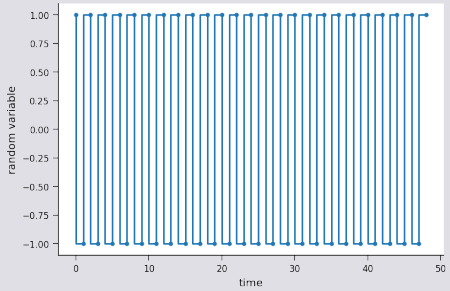
Autoregressive models

Higher order autoregressive models

Now that we have established the autoregressive framework, generalizing for dependence on data points from the past is straightforward. **Higher order** autoregression models a future time point based on more than one point in the past.
In one dimension, we can write such an order-r model as

$$x_{k+1} = \alpha_0 + \alpha_1 x_k + \alpha_2 x_{k-1} + \alpha_3 x_{k-2} + \cdots + \alpha_{r+1} x_{k-r},$$

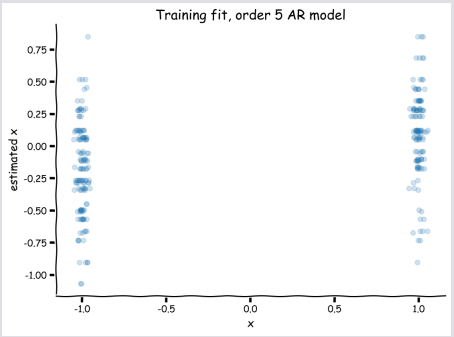
where the α's are the $r+1$ coefficients to be fit to the data available.
These models are useful to account for some **history dependence** in the trajectory of time series. This next part of the tutorial will explore one such time series, and you can do an experiment on yourself!
In particular, we will explore a binary random sequence of 0's and 1's that would occur if you flipped a coin and jotted down the flips.
The difference is that, instead of actually flipping a coin (or using code to generate such a sequence), you – yes you, human – are going to generate such a random Bernoulli sequence as best as you can by typing in 0's and 1's. We will then build higher-order AR models to see if we can identify predictable patterns in the time-history of digits you generate.



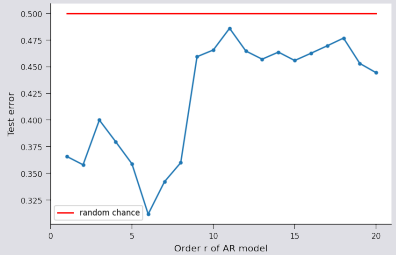
Autoregressive models

Understanding autoregressive parameters

Truly random sequences of numbers have no structure and should not be predictable by an AR or any other models. However, humans are notoriously terrible at generating random sequences of numbers! (Other animals are no better...)
To test out an application of higher-order AR models, let's use them to model a sequence of 0's and 1's that a human tried to produce at random.
If the digits really have no structure, then we expect our model to do about as well as guessing, producing an error rate of 0.5. Let's see how well we can do!
Fit a order-5 ($r = 5$) AR model to the data vector x. We will then plot the observations against the trained model. Note that this means we are using a sequence of the previous 5 digits to predict the next one.
Additionally, output from our regression model is continuous (real numbers) whereas our data are scalar (+1/-1). So, we will take the sign of our continuous outputs (+1 if positive and -1 if negative) as our predictions to make them comparable with data. Our error rate will simply be the number of mismatched predictions divided by the total number of predictions.



Let's now try **AR models of different orders** systematically, and plot the test error of each.



Notice that there's a sweet spot in the test error! The 6th order AR model does a really good job here, and for larger r's, the model starts to overfit the training data and does not do well on the test data.

Neuromatch Academy: Biological Neuron Models - Summary Sheet[11]

The Leaky Integrate-and-Fire (LIF) Neuron Model

The Leaky Integrate-and-Fire (LIF) model

The basic idea of LIF neuron was proposed in 1907 by Louis Édouard Lapicque, long before we understood the electrophysiology of a neuron (see a translation of Lapicque's paper.

The subthreshold membrane potential dynamics of a LIF neuron is described by

$$C_m \frac{dV}{dt} = -g_L(V - E_L) + I \tag{98}$$

where C_m is the membrane capacitance, V is the membrane potential, g_L is the leak conductance ($g_L = 1/R$, the inverse of the leak resistance R mentioned in previous tutorials), E_L is the resting potential, and I is the external input current.

Dividing both sides of the above equation by g_L gives

$$\tau_m \frac{dV}{dt} = -(V - E_L) + \frac{I}{g_L}, \quad (\ddagger) \tag{99}$$

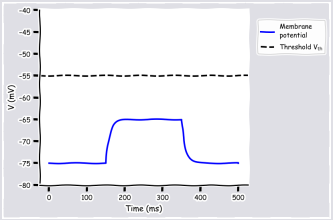
where the τ_m is membrane time constant and is defined as $\tau_m = C_m/g_L$.

Note that dividing capacitance by conductance gives units of time!

Below, we will use Eqn.(‡) to simulate LIF neuron dynamics. If I is sufficiently strong such that V reaches a certain threshold value $V_{\rm th}$, V is reset to a reset potential $V_{\rm reset} < V_{\rm th}$, and voltage is clamped to $V_{\rm reset}$ for $\tau_{\rm ref}$ ms, mimicking the refractoriness of the neuron during an action potential:

if $V(t_{\rm sp}) \geq V_{\rm th}$: $V(t) = V_{\rm reset}$ for $t \in (t_{\rm sp}, t_{\rm sp} + \tau_{\rm ref}]$

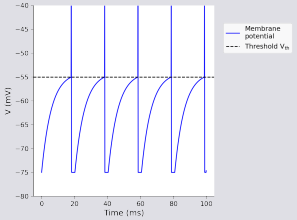
where $t_{\rm sp}$ is the spike time when $V(t)$ just exceeded $V_{\rm th}$.



The Leaky Integrate-and-Fire (LIF) Neuron Model

Response of an LIF model to different types of input currents

In the following section, we see how to inject direct current and white noise to study the response of an LIF neuron. The neuron generates a spike. But this is just a cosmetic spike only for illustration purposes. In an LIF neuron, we only need to keep track of times when the neuron hits the threshold so the postsynaptic neurons can be informed of the spike.

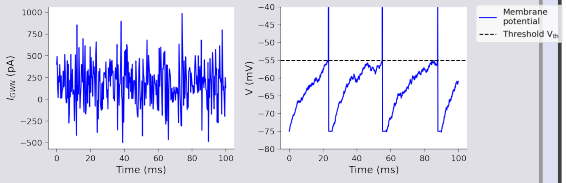


Gaussian white noise (GWN) current

Given the noisy nature of neuronal activity *in vivo*, neurons usually receive complex, time-varying inputs.

To mimic this, we will now investigate the neuronal response when the LIF neuron receives Gaussian white noise $\xi(t)$ with mean 0 ($\mu = 0$) and some standard deviation σ.

Note that the GWN has zero mean, that is, it describes only the fluctuations of the input received by a neuron. We can thus modify our definition of GWN to have a nonzero mean value μ that equals the DC input, since this is the average input into the cell. The cell below defines the modified gaussian white noise currents with nonzero mean μ.



The Leaky Integrate-and-Fire (LIF) Neuron Model

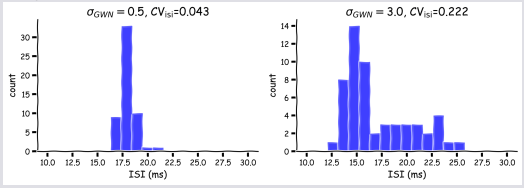
Firing rate and spike time irregularity

When we plot the output firing rate as a function of GWN mean or DC value, it is called the input-output transfer function of the neuron (so simply F-I curve).

Spike regularity can be quantified as the **coefficient of variation (CV) of the interspike interval (ISI)**:

$$\mathrm{CV}_{\rm ISI} = \frac{std(\mathrm{ISI})}{mean(\mathrm{ISI})} \tag{100}$$

A Poisson train is an example of high irregularity, in which $\mathbf{CV_{ISI} = 1}$. And for a clocklike (regular) process we have $\mathbf{CV_{ISI} = 0}$ because of **std(ISI)=0**.



Summary

You've a leaky integrate-and-fire (LIF) neuron model from scratch, and studied its dynamics in response to various types of inputs, having:

- simulated the LIF neuron model
- driven the LIF neuron with external inputs, such as direct current and Gaussian white noise
- studied how different inputs affect the LIF neuron's output (firing rate and spike time irregularity),

with a special focus on low rate and irregular firing regime to mimic real cortical neurons. The next tutorial will look at how spiking statistics may be influenced by a neuron's input statistics.

If you have extra time, look at the bonus sections below to explore a different type of noise input and learn about extensions to integrate-and-fire models.

[11] 't Hart et al., (2022). Neuromatch Academy: a 3-week, online summer school in computational neuroscience. Journal of Open Source Education, 5(49), 118. https://doi.org/10.21105/jose.00118

Effects of Input Correlation

Correlations (Synchrony)

Correlation or synchrony in neuronal activity can be described for any readout of brain activity. Here, we are concerned with the spiking activity of neurons.
In the simplest way, correlation/synchrony refers to coincident spiking of neurons, i.e., when two neurons spike together, they are firing in **synchrony** or are **correlated**. Neurons can be synchronous in their instantaneous activity, i.e., they spike together with some probability. However, it is also possible that spiking of a neuron at time t is correlated with the spikes of another neuron with a delay (time-delayed synchrony).

Origin of synchronous neuronal activity:

- Common inputs, i.e., two neurons are receiving input from the same sources. The degree of correlation of the shared inputs is proportional to their output correlation.
- Pooling from the same sources. Neurons do not share the same input neurons but are receiving inputs from neurons which themselves are correlated.
- Neurons are connected to each other (uni- or bi-directionally): This will only give rise to time-delayed synchrony. Neurons could also be connected via gap-junctions.
- Neurons have similar parameters and initial conditions.

Implications of synchrony

When neurons spike together, they can have a stronger impact on downstream neurons. Synapses in the brain are sensitive to the temporal correlations (i.e., delay) between pre- and post-synaptic activity, and this, in turn, can lead to the formation of functional neuronal networks - the basis of unsupervised learning.
Synchrony implies a reduction in the dimensionality of the system. In addition, correlations, in many cases, can impair the decoding of neuronal activity.
A simple model to study the emergence of correlations is to inject common inputs to a pair of neurons and measure the output correlation as a function of the fraction of common inputs.
Here, we are going to investigate the transfer of correlations by computing the correlation coefficient of spike trains recorded from two unconnected LIF neurons, which received correlated inputs.
The input current to LIF neuron i $(i = 1, 2)$ is:

$$\frac{I_i}{g_L} = \mu_i + \sigma_i(\sqrt{1-c}\xi_i + \sqrt{c}\xi_c) \quad (1) \qquad (101)$$

where μ_i is the temporal average of the current. The Gaussian white noise ξ_i is independent for each neuron, while ξ_c is common to all neurons. The variable c $(0 \leq c \leq 1)$ controls the fraction of common and independent inputs. σ_i shows the variance of the total input.
So, first, we will generate correlated inputs.

Effects of Input Correlation

Compute the correlation

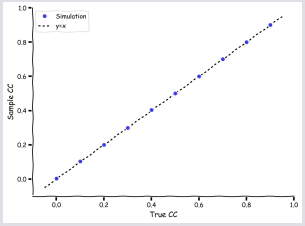
The *sample correlation coefficient* between two input currents I_i and I_j is defined as the sample covariance of I_i and I_j divided by the square root of the sample variance of I_i multiplied with the square root of the sample variance of I_j. In equation form:

$$r_{ij} = \frac{cov(I_i, I_j)}{\sqrt{var(I_i)}\sqrt{var(I_j)}} \qquad (102)$$

$$cov(I_i, I_j) = \sum_{k=1}^{L}(I_i^k - \bar{I}_i)(I_j^k - \bar{I}_j) \qquad (103)$$

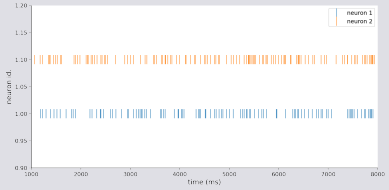
$$var(I_i) = \sum_{k=1}^{L}(I_i^k - \bar{I}_i)^2 \qquad (104)$$

where $\bar{I}_i$ is the sample mean, k is the time bin, and L is the length of I. This means that I_i^k is current i at time $k \cdot dt$. Note that the equations above are not accurate for sample covariances and variances as they should be additionally divided by L-1 - we have dropped this term because it cancels out in the sample correlation coefficient formula.
The *sample correlation coefficient* may also be referred to as the *sample Pearson correlation coefficient*. Here, is a beautiful paper that explains multiple ways to calculate and understand correlations Rodgers and Nicewander 1988.



Measure the correlation between spike trains

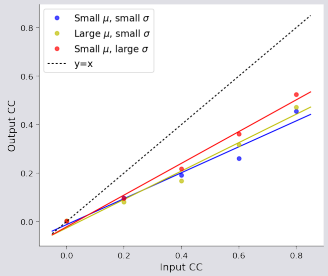
After recording the spike times of the two neurons, how can we estimate their correlation coefficient?
In order to find this, we need to bin the spike times and obtain two time series. Each data point in the time series is the number of spikes in the corresponding time bin.



Effects of Input Correlation

How do the mean and standard deviation of the Gaussian white noise (GWN) affect the correlation transfer function?

The correlations transfer function appears to be linear. The input/output transfer function of LIF neurons for correlations, instead of the transfer function for input/output firing rates. What would you expect to happen to the slope of the correlation transfer function if you vary the mean and/or the standard deviation of the GWN of the inputs ?



What is the rationale behind varying μ and σ?

The mean and the variance of the synaptic current depends on the spike rate of a Poisson process. We can use something called Campbell's theorem to estimate the mean and the variance of the synaptic current:

$$\mu_{\text{syn}} = \lambda J \int P(t) \qquad (105)$$

$$\sigma_{\text{syn}} = \lambda J \int P(t)^2 dt \qquad (106)$$

where λ is the firing rate of the Poisson input, J the amplitude of the postsynaptic current and $P(t)$ is the shape of the postsynaptic current as a function of time. Therefore, when we varied μ and/or σ of the GWN, we mimicked a change in the input firing rate. Note that, if we change the firing rate, both μ and σ will change simultaneously, not independently. Here, since we observe an effect of μ and σ on correlation transfer, this implies that the input rate has an impact on the correlation transfer function.

Synaptic transmission - Models of static and dynamic synapses

Static synapses

Synaptic input *in vivo* consists of a mixture of **excitatory** neurotransmitters, which depolarizes the cell and drives it towards spike threshold, and **inhibitory** neurotransmitters that hyperpolarize it, driving it away from spike threshold. These chemicals cause specific ion channels on the post-synaptic neuron to open, resulting in a change in that neuron's conductance and, therefore, the flow of current in or out of the cell.
This process can be modeled by assuming that the presynaptic neuron's spiking activity produces transient changes in the postsynaptic neuron's conductance ($g_{\mathrm{syn}}(t)$). Typically, the conductance transient is modeled as an exponential function.
Such conductance transients can be generated using a simple ordinary differential equation (ODE):

$$\frac{dg_{\mathrm{syn}}(t)}{dt} = \bar{g}_{\mathrm{syn}} \sum_k \delta(t - t_k) - g_{\mathrm{syn}}(t)/\tau_{\mathrm{syn}} \quad (107)$$

where $\bar{g}_{\mathrm{syn}}$ (often referred to as synaptic weight) is the maximum conductance elicited by each incoming spike, and τ_{syn} is the synaptic time constant. Note that the summation runs over all spikes received by the neuron at time t_k.
Ohm's law allows us to convert conductance changes to the current as:

$$I_{\mathrm{syn}}(t) = g_{\mathrm{syn}}(t)(V(t) - E_{\mathrm{syn}}) \quad (108)$$

The reversal potential E_{syn} determines the direction of current flow and the excitatory or inhibitory nature of the synapse.
Thus, incoming spikes are filtered by an exponential-shaped kernel, effectively low-pass filtering the input. In other words, synaptic input is not white noise, but it is, in fact, colored noise, where the color (spectrum) of the noise is determined by the synaptic time constants of both excitatory and inhibitory synapses.
In a neuronal network, the total synaptic input current I_{syn} is the sum of both excitatory and inhibitory inputs. Assuming the total excitatory and inhibitory conductances received at time t are $g_E(t)$ and $g_I(t)$, and their corresponding reversal potentials are E_E and E_I, respectively, then the total synaptic current can be described as:

$$I_{\mathrm{syn}}(V(t), t) = -g_E(t)(V - E_E) - g_I(t)(V - E_I) \quad (109)$$

Synaptic transmission - Models of static and dynamic synapses

Static synapses

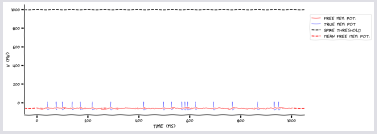
Accordingly, the membrane potential dynamics of the LIF neuron under synaptic current drive become:

$$\tau_m \frac{dV(t)}{dt} = -(V(t) - E_L) - \frac{g_E(t)}{g_L}(V(t) - E_E) - \frac{g_I(t)}{g_L}(V(t) - E_I) + \frac{I_{\mathrm{inj}}}{g_L} \quad (‡)$$

I_{inj} is an external current injected in the neuron, which is under experimental control; it can be GWN, DC, or anything else.
We will use Eq. (‡) to simulate the conductance-based LIF neuron model below.
In the previous tutorials, we saw how the output of a single neuron (spike count/rate and spike time irregularity) changes when we stimulate the neuron with DC and GWN, respectively. Now, we are in a position to study how the neuron behaves when it is bombarded with both excitatory and inhibitory spikes trains – as happens *in vivo*.
What kind of input is a neuron receiving? When we do not know, we choose the simplest option. The simplest model of input spikes is given when every input spike arrives independently of other spikes, i.e., we assume that the input is Poissonian.

Measure the mean free membrane potential

The conductance-based LIF neuron with presynaptic spike trains with rate 10 Hz for both excitatory and inhibitory inputs with 80 excitatory presynaptic spike trains and 20 inhibitory ones.
The CV_{ISI} can describe the irregularity of the output spike pattern. Now, we will introduce a new descriptor of the neuron membrane, i.e., the **Free Membrane Potential (FMP)** – the membrane potential of the neuron when its spike threshold is removed.
Although this is completely artificial, calculating this quantity allows us to get an idea of how strong the input is. We are mostly interested in knowing the mean and standard deviation (std.) of the FMP. In the exercise, you can visualize the FMP and membrane voltage with spike threshold.



Synaptic transmission - Models of static and dynamic synapses

Short-term synaptic plasticity

We modeled synapses with fixed weights. Now we will explore synapses whose weights change in some input conditions.
Short-term plasticity (STP) is a phenomenon in which synaptic efficacy changes over time in a way that reflects the history of presynaptic activity. Two types of STP, with opposite effects on synaptic efficacy, have been experimentally observed. They are known as Short-Term Depression (STD) and Short-Term Facilitation (STF).
The mathematical model (for more information see here) of STP is based on the concept of a limited pool of synaptic resources available for transmission (R), such as, for example, the overall amount of synaptic vesicles at the presynaptic terminals. The amount of presynaptic resource changes in a dynamic fashion depending on the recent history of spikes.
Following a presynaptic spike, (i) the fraction u (release probability) of the available pool to be utilized increases due to spike-induced calcium influx to the presynaptic terminal, after which (ii) u is consumed to increase the post-synaptic conductance. Between spikes, u decays back to zero with time constant τ_f and R recovers to 1 with time constant τ_d. In summary, the dynamics of excitatory (subscript E) STP are given by:

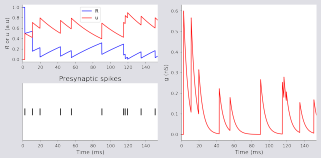
$$\frac{du_E}{dt} = -\frac{u_E}{\tau_f} + U_0(1 - u_E^-)\delta(t - t_{\mathrm{sp}})$$
$$\frac{dR_E}{dt} = \frac{1 - R_E}{\tau_d} - u_E^+ R_E^- \delta(t - t_{\mathrm{sp}})$$
$$\frac{dg_E(t)}{dt} = -\frac{g_E}{\tau_E} + \bar{g}_E u_E^+ R_E^- \delta(t - t_{\mathrm{sp}})$$

where U_0 is a constant determining the increment of u produced by a spike. u_E^- and R_E^- denote the corresponding values just before the spike arrives, whereas u_E^+ refers to the moment right after the spike. $\bar{g}_E$ denotes the maximum excitatory conductance, and $g_E(t)$ is calculated for all spike-times k, and decays over time with a time constant τ_E. Similarly, one can obtain the dynamics of inhibitory STP (i.e., by replacing the subscript E with I). The interplay between the dynamics of u and R determines whether the joint effect of uR is dominated by "depression" or "facilitation". In the parameter regime of $\tau_d \gg \tau_f$ and for large U_0, an initial spike incurs a large drop in R that takes a long time to recover; therefore, the synapse is STD-dominated. In the regime of $\tau_d \ll \tau_f$ and for small U_0, the synaptic efficacy is increased gradually by spikes, and consequently, the synapse is STF-dominated. This phenomenological model successfully reproduces the kinetic dynamics of depressed and facilitated synapses observed in many cortical areas.

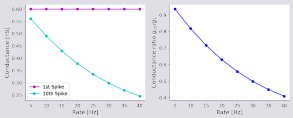
Synaptic transmission - Models of static and dynamic synapses

Short-term synaptic depression (STD)

Below shows how Short-term synaptic depression (STD) changes for different firing rates of the presynaptic spike train and how the amplitude synaptic conductance g changes with every incoming spike until it reaches its stationary state.

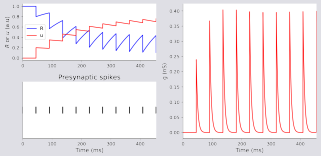


Here, we will study how the ratio of the synaptic conductance corresponding to the first and 10th spikes change as a function of the presynaptic firing rate (experimentalists often take the ratio of first and second PSPs).
Below shows the STD conductance ratio with different input rates.

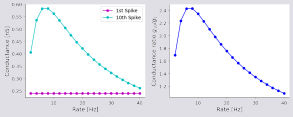


Short-term synaptic facilitation (STF)

Below, we see an illustration of a short-term facilitation example.



Here, we will study how the ratio of the synaptic conductance corresponding to the 1^{st} and 10^{th} spike changes as a function of the presynaptic rate.
Below STF conductance ratio with different input rates.



Neuromatch Academy: Dynamic Networks - Summary Sheet[12]

Neural Rate Models

Dynamics of a single excitatory population

Individual neurons respond by spiking. When we average the spikes of neurons in a population, we can define the average firing activity of the population. In this model, we are interested in how the population-averaged firing varies as a function of time and network parameters. Mathematically, we can describe the firing rate dynamic of a feed-forward network as:

$$\tau \frac{dr}{dt} = -r + F(I_{\text{ext}})$$

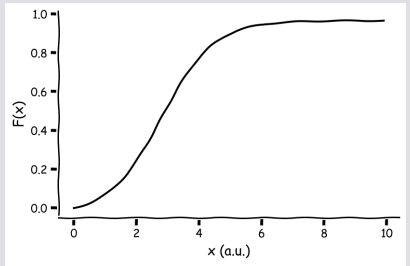
$r(t)$ represents the average firing rate of the excitatory population at time t, τ controls the timescale of the evolution of the average firing rate, I_{ext} represents the external input, and the transfer function $F(\cdot)$ (which can be related to f-I curve of individual neurons described in the next sections) represents the population activation function in response to all received inputs.

F-I (firing rate vs. input) curve

In electrophysiology, a neuron is often characterized by its spike rate output in response to input currents. This is often called the F-I curve, denoting the output spike frequency (F) in response to different injected currents (I).
The transfer function $F(\cdot)$ in Equation 1 represents the gain of the population as a function of the total input. The gain is often modeled as a sigmoidal function, i.e., more input drive leads to a nonlinear increase in the population firing rate. The output firing rate will eventually saturate for high input values.
A sigmoidal $F(\cdot)$ is parameterized by its gain a and threshold θ.

$$F(x; a, \theta) = \frac{1}{1 + \mathrm{e}^{-a(x-\theta)}} - \frac{1}{1 + \mathrm{e}^{a\theta}}$$

The argument x represents the input to the population. Note that the second term is chosen so that $F(0; a, \theta) = 0$.



Neural Rate Models

Fixed points of the single population system

We can now extend our feed-forward network to a recurrent network, governed by the equation:

$$\tau \frac{dr}{dt} = -r + F(w \cdot r + I_{\text{ext}}) \qquad (+)$$

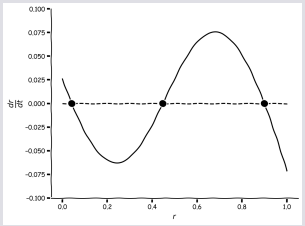
where as before, $r(t)$ represents the average firing rate of the excitatory population at time t, τ controls the timescale of the evolution of the average firing rate, I_{ext} represents the external input, and the transfer function $F(\cdot)$ (which can be related to f-I curve of individual neurons described in the next sections) represents the population activation function in response to all received inputs. Now we also have w which denotes the strength (synaptic weight) of the recurrent input to the population.
As you varied the two parameters in the last Interactive Demo, you noticed that, while at first the system output quickly changes, with time, it reaches its maximum/minimum value and does not change anymore. The value eventually reached by the system is called the **steady state** of the system, or the **fixed point**. Essentially, in the steady states the derivative with respect to time of the activity (r) is zero, i.e. $\frac{dr}{dt} = 0$.
We can find that the steady state of the Equation ($+$) by setting $\frac{dr}{dt} = 0$ and solve for r:

$$-r_{\text{steady}} + F(w \cdot r_{\text{steady}} + I_{\text{ext}}; a, \theta) = 0 \qquad (\ddagger)$$

When it exists, the solution of Equation ($\ddagger$) defines a **fixed point** of the dynamical system in Equation ($+$). Note that if $F(x)$ is nonlinear, it is not always possible to find an analytical solution, but the solution can be found via numerical simulations. In the specific case of $w = 0$, we can also analytically compute the solution of Equation ($+$) and deduce the role of τ in determining the convergence to the fixed point:

$$r(t) = \left[F(I_{\text{ext}}; a, \theta) - r(t=0)\right](1 - \mathrm{e}^{-\frac{t}{\tau}}) + r(t=0)$$

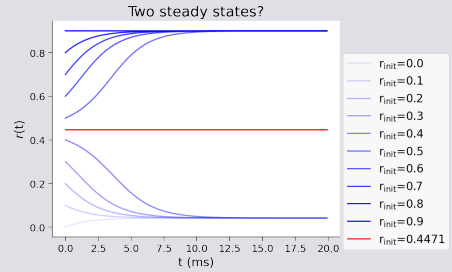
We can now numerically calculate the fixed point with a root finding algorithm.



Neural Rate Models

Fixed points of the single population system

Relationship between trajectories fixed points
Let's examine the relationship between the population activity over time and the fixed points.
Here, let us first set $w = 5.0$ and $I_{\text{ext}} = 0.5$, and investigate the dynamics of $r(t)$ starting with different initial values $r(0) \equiv r_{\text{init}}$.
We will plot the trajectories of $r(t)$ with different initial values $r(0) \equiv r_{\text{init}} = 0.0, 0.1, 0.2, ..., 0.9$. We have three fixed points but only two steady states showing up - what's happening?
It turns out that the stability of the fixed points matters. If a fixed point is stable, a trajectory starting near that fixed point will stay close to that fixed point and converge to it (the steady state will equal the fixed point). If a fixed point is unstable, any trajectories starting close to it will diverge and go towards stable fixed points. In fact, the only way for a trajectory to reach a stable state at an unstable fixed point is if the initial value **exactly** equals the value of the fixed point.
We can simulate the trajectory if we start at the unstable fixed point: you can see that it remains at that fixed point (the red line below).



[12] 't Hart et al., (2022). Neuromatch Academy: a 3-week, online summer school in computational neuroscience. Journal of Open Source Education, 5(49), 118. https://doi.org/10.21105/jose.00118

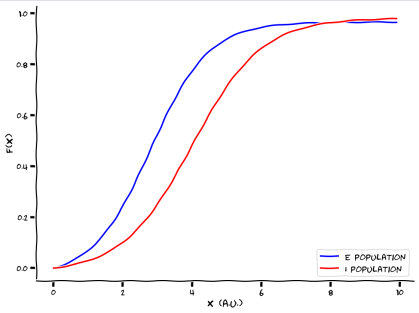
Wilson-Cowan Model

Mathematical description of the WC model

Many of the rich dynamics recorded in the brain are generated by the interaction of excitatory and inhibitory subtype neurons. Here, similar to what we did in the previous tutorial, we will model two coupled populations of E and I neurons (**Wilson-Cowan** model). We can write two coupled differential equations, each representing the dynamics of the excitatory or inhibitory population:

$$\tau_E \frac{dr_E}{dt} = -r_E + F_E(w_{EE}r_E - w_{EI}r_I + I_E^{\text{ext}}; a_E, \theta_E)$$
$$\tau_I \frac{dr_I}{dt} = -r_I + F_I(w_{IE}r_E - w_{II}r_I + I_I^{\text{ext}}; a_I, \theta_I)$$

$r_E(t)$ represents the average activation (or firing rate) of the excitatory population at time t, and $r_I(t)$ the activation (or firing rate) of the inhibitory population. The parameters τ_E and τ_I control the timescales of the dynamics of each population. Connection strengths are given by: w_{EE} (E $\rightarrow$ E), w_{EI} (I $\rightarrow$ E), w_{IE} (E $\rightarrow$ I), and w_{II} (I $\rightarrow$ I). The terms w_{EI} and w_{IE} represent connections from inhibitory to excitatory population and vice versa, respectively. The transfer functions (or F-I curves) $F_E(x; a_E, \theta_E)$ and $F_I(x; a_I, \theta_I)$ can be different for the excitatory and the inhibitory populations.



Wilson-Cowan Model

Numerically integrate the Wilson-Cowan equations

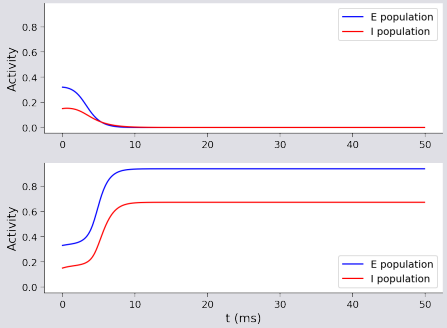
Once again, we can integrate our equations numerically. Using the Euler method, the dynamics of E and I populations can be simulated on a time-grid of stepsize Δt. The updates for the activity of the excitatory and the inhibitory populations can be written as:

$$r_E[k+1] = r_E[k] + \Delta r_E[k]$$
$$r_I[k+1] = r_I[k] + \Delta r_I[k]$$

with the increments

$$\Delta r_E[k] = \frac{\Delta t}{\tau_E}[-r_E[k] + F_E(w_{EE}r_E[k] - w_{EI}r_I[k] + I_E^{\text{ext}}[k]; a_E, \theta_E)]$$
$$\Delta r_I[k] = \frac{\Delta t}{\tau_I}[-r_I[k] + F_I(w_{IE}r_E[k] - w_{II}r_I[k] + I_I^{\text{ext}}[k]; a_I, \theta_I)]$$

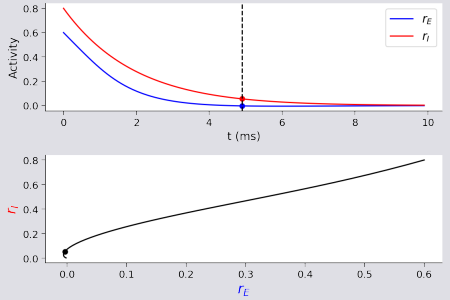
The two plots above show the temporal evolution of excitatory (r_E, blue) and inhibitory (r_I, red) activity for two different sets of initial conditions.



Wilson-Cowan Model

Phase plane analysis

Just like we used a graphical method to study the dynamics of a 1-D system in the previous tutorial, here we will learn a graphical approach called **phase plane analysis** to study the dynamics of a 2-D system like the Wilson-Cowan model.
So far, we have plotted the activities of the two populations as a function of time. Instead, we can plot the two activities $r_E(t)$ and $r_I(t)$ against each other at any time point t. This characterization in the 'rI-rE' plane $(r_I(t), r_E(t))$ is called the **phase plane**. Each line in the phase plane indicates how both r_E and r_I evolve with time.



Nullclines of the Wilson-Cowan Equations

An important concept in the phase plane analysis is the "nullcline" which is defined as the set of points in the phase plane where the activity of one population (but not necessarily the other) does not change.
In other words, the E and I nullclines of Equation (1) are defined as the points where $\frac{dr_E}{dt} = 0$, for the excitatory nullcline, or $\frac{dr_I}{dt} = 0$ for the inhibitory nullcline. That is:

$$-r_E + F_E(w_{EE}r_E - w_{EI}r_I + I_E^{\text{ext}}; a_E, \theta_E) = 0$$
$$-r_I + F_I(w_{IE}r_E - w_{II}r_I + I_I^{\text{ext}}; a_I, \theta_I) = 0$$

Wilson-Cowan Model

Compute the nullclines of the Wilson-Cowan model

Along the nullcline of excitatory population , you can calculate the inhibitory activity by rewriting it as

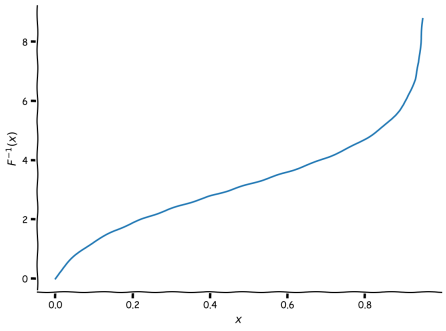
$$r_I = \frac{1}{w_{EI}}\left[w_{EE}r_E - F_E^{-1}(r_E; a_E, \theta_E) + I_E^{\text{ext}}\right].$$

where $F_E^{-1}(r_E; a_E, \theta_E)$ is the inverse of the excitatory transfer function (defined below).
Along the nullcline of inhibitory population, you can calculate the excitatory activity by rewriting it as

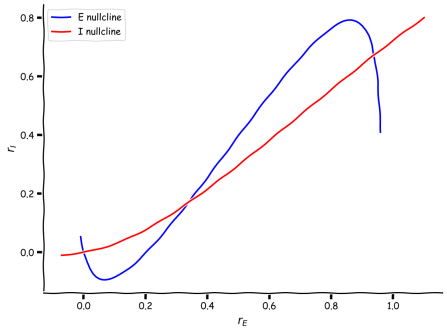
$$r_E = \frac{1}{w_{IE}}\left[w_{II}r_I + F_I^{-1}(r_I; a_I, \theta_I) - I_I^{\text{ext}}\right]$$

where $F_I^{-1}(r_I; a_I, \theta_I)$ is the inverse of the inhibitory transfer function.
The inverse of the sigmoid shaped "f-I" function that we have been using is:

$$F^{-1}(x; a, \theta) = -\frac{1}{a}\ln\left[\frac{1}{x + \frac{1}{1+\mathrm{e}^{a\theta}}} - 1\right] + \theta.$$



Now you can plot the nullclines



Wilson-Cowan Model

Vector field

How can the phase plane and the nullcline curves help us understand the behavior of the Wilson-Cowan model?
The activities of the E and I populations $r_E(t)$ and $r_I(t)$ at each time point t correspond to a single point in the phase plane, with coordinates $(r_E(t), r_I(t))$. Therefore, the time-dependent trajectory of the system can be described as a continuous curve in the phase plane, and the tangent vector to the trajectory, which is defined as the vector $\left(\frac{dr_E(t)}{dt}, \frac{dr_I(t)}{dt}\right)$, indicates the direction towards which the activity is evolving and how fast is the activity changing along each axis. In fact, for each point (E, I) in the phase plane, we can compute the tangent vector $\left(\frac{dr_E}{dt}, \frac{dr_I}{dt}\right)$, which indicates the behavior of the system when it traverses that point.
The map of tangent vectors in the phase plane is called **vector field**. The behavior of any trajectory in the phase plane is determined by

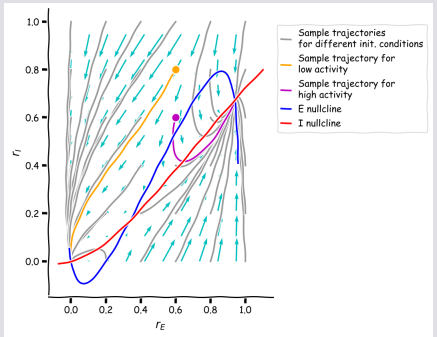
1. the initial conditions $(r_E(0), r_I(0))$, and
2. the vector field $\left(\frac{dr_E(t)}{dt}, \frac{dr_I(t)}{dt}\right)$.

In general, the value of the vector field at a particular point in the phase plane is represented by an arrow. The orientation and the size of the arrow reflect the direction and the norm of the vector, respectively.
To compute and plot the vector field we use

$$\frac{dr_E}{dt} = [-r_E + F_E(w_{EE}r_E - w_{EI}r_I + I_E^{\text{ext}}; a_E, \theta_E)]\frac{1}{\tau_E}$$
$$\frac{dr_I}{dt} = [-r_I + F_I(w_{IE}r_E - w_{II}r_I + I_I^{\text{ext}}; a_I, \theta_I)]\frac{1}{\tau_I}.$$

The phase plane plot below shows us that:

- Trajectories seem to follow the direction of the vector field.
- Different trajectories eventually always reach one of two points depending on the initial conditions.
- The two points where the trajectories converge are the intersection of the two nullcline curves.



Neuromatch Academy: Bayesian Decisions - Summary Sheet[13]

Bayes with a binary hidden state

Introduction

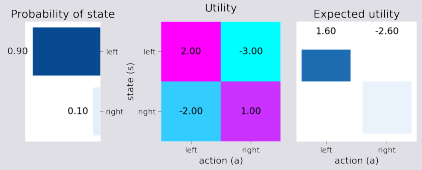
We will introduce the fundamental building blocks for Bayesian statistics:
1. How do we combine the possible loss (or gain) for making a decision with our probabilistic knowledge?
2. How do we use probability distributions to represent hidden states?
3. How does marginalization work and how can we use it?
4. How do we combine new information with our prior knowledge?

Gone Fishin'

We are going to explore **binary hidden state problem** by going fishing. We need to decide which side to fish on–the hidden state. We know fish like to school together. On different days the school of fish is either on the left or right side, but we don't know what the case is today. We define our knowledge about the fish location as a distribution over the random hidden state variable. Using our probabilistic knowledge, also called our **belief** about the hidden state, we will explore how to make the decision about where to fish today, based on what to expect in terms of gains or losses for the decision. The gains and loss are defined by the utility of choosing an action, which is fishing on the left or right.

Deciding where to fish

Let's start to get a sense of how all this works using an example. First, make sure we understand how the expected utility of each action is being computed from the probabilities and the utility values. In the initial state: the probability of the fish being on the left is 0.9 and on the right is 0.1. The expected utility of the action of fishing on the left is then $U(s = \text{left}, a = \text{left})p(s = \text{left}) + U(s = \text{right}, a = \text{left})p(s = \text{right}) = 2(0.9) + -2(0.1) = 1.6$. Essentially, to get the expected utility of action a, we are doing a weighted sum over the relevant column of the utility matrix (corresponding to action a) where the weights are the state probabilities.



Bayes with a binary hidden state

Likelihood of the fish being on either side

First, we'll think about what it means to take a measurement (also often called an observation or just data) and what it tells us about what the hidden state may be. Specifically, we'll be looking at the **likelihood**, which is the probability of our measurement (m) given the hidden state (s): $P(m|s)$. Remember that in this case, the hidden state is which side of the dock the school of fish is on. We will watch someone fish (for let's say 10 minutes) and our measurement is whether they catch a fish or not. We know something about what catching a fish means for the likelihood of the fish being on one side or the other.

Guessing the location of the fish

Let's say we go to a different dock to fish. Here, there are different probabilities of catching fish given the state of the world. At this dock, if we fish on the side of the dock where the fish are, we have a 70% chance of catching a fish. If we fish on the wrong side, we will catch a fish with only 20% probability. These are the likelihoods of observing someone catching a fish! That is, we are taking a measurement by seeing if someone else catches a fish!
We see a fisher-person is fishing on the left side.
In the example, we tried to guess where the school of fish was based on the measurement we took (watching someone fish). We did this by choosing the state (side where we think the fish are) that maximized the probability of the measurement. In other words, we estimated the state by maximizing the likelihood (the side with the highest probability of measurement given state $P(m|s)$). This is called maximum likelihood estimation (MLE).
But, what if we had been going to this dock for years and we knew that the fish were almost always on the left side? This should probably affect how we make our estimate – we would rely less on the single new measurement and more on our prior knowledge. This is the fundamental idea behind Bayesian inference.

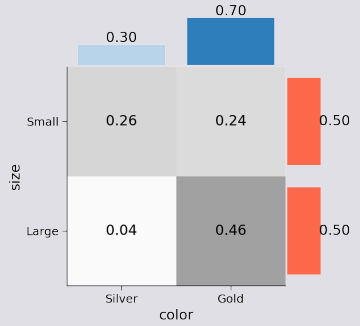
Bayes with a binary hidden state

Correlation

We are going to take a step back for a bit and think more generally about the amount of information shared between two random variables. We want to know how much information we gain when we observe one variable (take a measurement) if we know something about another. We will see that the fundamental concept is the same if we think about two attributes, for example the size and color of the fish, or the prior information and the likelihood.

Covarying probability distributions

The relationship between the marginal probabilities and the joint probabilities is determined by the correlation between the two random variables - a normalized measure of how much the variables covary. We can also think of this as gaining some information about one of the variables when we observe a measurement from the other.
Here, we want to think about how the correlation between size and color of these fish changes how much information we gain about one attribute based on the other.



[13] 't Hart et al., (2022). Neuromatch Academy: a 3-week, online summer school in computational neuroscience. Journal of Open Source Education, 5(49), 118. https://doi.org/10.21105/jose.00118

Bayes with a binary hidden state

Marginalisation

We may want to find the probability of one variable while ignoring another: we will do this by averaging out, or marginalizing, the irrelevant variable.
We will think of this in two different ways.
In the first math exercise, we will think about the case where we know the joint probabilities of two variables and want to figure out the probability of just one variable. To make this explicit, let's assume that a fish has a color that is either gold or silver (our first variable) and a size that is either small or large (our second). We could write out the the **joint probabilities**: the probability of both specific attributes occuring together. For example, the probability of a fish being small and silver, $P(X = \text{small}, Y = \text{silver})$, is 0.4. The following table summarizes our joint probabilities:

P(X, Y)	Y = silver	Y = gold
X = small	0.4	0.2
X = large	0.1	0.3

We want to know what the probability of a fish being small regardless of color. Since the fish are either silver or gold, this would be the probability of a fish being small and silver plus the probability of a fish being small and gold. This is an example of marginalizing, or averaging out, the variable we are not interested in across the rows or columns.. In math speak: $P(X = \text{small}) = \sum_y P(X = \text{small}, Y)$. This gives us a **marginal probability**, a probability of a variable outcome (in this case size), regardless of the other variables (in this case color).
More generally, we can marginalize out a second irrelevant variable y by summing over the relevant joint probabilities:

$$p(x) = \sum_y p(x, y)$$

To find the marginal probability of a measurement we will remove an unknown (the hidden state). We will do this by marginalizing out the hidden state. In this case, we know the conditional probabilities of the measurement given state and the probabilities of each state. We can marginalize using:

$$p(m) = \sum_s p(m|s)p(s)$$

These two ways of thinking about marginalization (as averaging over joint probabilities or conditioning on some variable) are equivalent because the joint probability of two variables equals the conditional probability of the first given the second times the marginal probability of the second:

$$p(x, y) = p(x|y)p(y)$$

Bayes with a binary hidden state

Computing marginal probabilities

The probability of a fish being silver is the joint probability of it being small and silver plus the joint probability of it being large and silver:

$$P(Y = silver) = P(X = small, Y = silver) + P(X = large, Y = silver) = 0.4 + 0.1 = 0.5$$

This is all the possibilities as in this scenario, our fish can only be small or large, silver or gold. So the probability is 1 - the fish has to be at least one of these.
First we compute the marginal probabilities

$$P(X = small) = P(X = small, Y = silver) + P(X = small, Y = gold) = 0.4 + 0.2 = 0.6$$

$$P(Y = gold) = P(X = small, Y = gold) + P(X = large, Y = gold) = 0.5$$

We already know the joint probability:

$$P(X = small, Y = gold) = 0.2$$

We can now use the given formula:

$$P(X = smallorY = gold) = P(X = small) + P(Y = gold) - P(X = small, Y = gold) = 0.6 + 0.5 - 0.2 = 0.9$$

Computing marginal likelihood

When we normalize to find the posterior, we need to determine the marginal likelihood–or evidence–for the measurement we observed. To do this, we need to marginalize as we just did above to find the probabilities of a color or size. Only, in this case, we are marginalizing to remove a conditioning variable! In this case, let's consider the likelihood of fish (if we observed a fisher-person fishing on the *right*).

p(m||s)	m = fish	m = no fish
s = left	0.1	0.9
s = right	0.5	0.5

The table above shows us the **likelihoods**, just as we explored earlier.
We want to know the total probability of a fish being caught, $P(m = \text{fish})$, by the fisher-person fishing on the right. (We would need this to calculate the posterior.) To do this, we will need to consider the prior probability, $p(s)$, and marginalize over the hidden states!
This is an example of marginalizing, or conditioning away, the variable we are not interested in as well.

Bayes with a binary hidden state

Computing marginal likelihood

Given the priors Priors

$$P(s = left) = 0.3$$

$$P(s = right) = 0.7$$

and the likelihoods

$$P(m = fish|s = left) = 0.1$$

$$P(m = fish|s = right) = 0.5$$

$$P(m = nofish|s = left) = 0.9$$

$$P(m = nofish|s = right) = 0.5$$

We can calculate the marginal likelihood (evidence):

$$P(m = fish) = P(m = fish, s = left) + P(m = fish, s = right) = P(m = fish|s = left)P(s = left) + P(m = fish|s = right)P(s = right) = 0.1 * 0.3 + .5 * .7 = 0.38$$

We can calculate the marginal likelihood (evidence):

$$P(m = fish) = P(m = fish, s = left) + P(m = fish, s = right) = P(m = fish|s = left)P(s = left) + P(m = fish|s = right)P(s = right) = 0.1 * 0.6 + .5 * .4 = 0.26$$

Bayes with a binary hidden state

Bayes' Rule and the Posterior

Marginalization is going to be used to combine our prior knowledge, which we call the **prior**, and our new information from a measurement, the **likelihood**. Only in this case, the information we gain about the hidden state we are interested in, where the fish are, is based on the relationship between the probabilities of the measurement and our prior. We can now calculate the full posterior distribution for the hidden state (s) using Bayes' Rule. As we've seen, the posterior is proportional to the prior times the likelihood. This means that the posterior probability of the hidden state (s) given a measurement (m) is proportional to the likelihood of the measurement given the state times the prior probability of that state:

$$P(s|m) \propto P(m|s)P(s) \tag{110}$$

We say proportional to instead of equal because we need to normalize to produce a full probability distribution:

$$P(s|m) = \frac{P(m|s)P(s)}{P(m)} \tag{111}$$

Normalizing by this $P(m)$ means that our posterior is a complete probability distribution that sums or integrates to 1 appropriately. We now can use this new, complete probability distribution for any future inference or decisions we like! In fact, as we will see tomorrow, we can use it as a new prior! Finally, we often call this probability distribution our beliefs over the hidden states, to emphasize that it is our subjective knowledge about the hidden state.

For many complicated cases, like those we might be using to model behavioral or brain inferences, the normalization term can be intractable or extremely complex to calculate. We can be careful to choose probability distributions where we can analytically calculate the posterior probability or numerical approximation is reliable. Better yet, we sometimes don't need to bother with this normalization! The normalization term, $P(m)$, is the probability of the measurement. This does not depend on state so is essentially a constant we can often ignore. We can compare the unnormalized posterior distribution values for different states because how they relate to each other is unchanged when divided by the same constant. We will see how to do this to compare evidence for different hypotheses tomorrow. (It's also used to compare the likelihood of models fit using maximum likelihood estimation)

In this relatively simple example, we can compute the marginal likelihood $P(m)$ easily by using:

$$P(m) = \sum_s P(m|s)P(s) \tag{112}$$

We can then normalize so that we deal with the full posterior distribution.

Bayes with a binary hidden state

Calculating a posterior probability

Given the priors Priors

$$P(s = left) = 0.3$$

$$P(s = right) = 0.7$$

and the likelihoods

$$P(m = fish|s = left) = 0.1$$

$$P(m = fish|s = right) = 0.5$$

$$P(m = nofish|s = left) = 0.9$$

$$P(m = nofish|s = right) = 0.5$$

Calculate the posterior probability that the school is on the left if the fisher-person catches a fish: $p(s = \text{left}|m = \text{fish})$ (hint: normalize by computing $p(m = \text{fish})$).

Using Bayes rule, we know that

$$P(s = left|m = fish) = \frac{P(m = fish|s = left)P(s = left)}{P(m = fish)}$$

Let's first compute P(m = fish):

$$P(m = fish) = P(m = fish|s = left)P(s = left) + P(m = fish|s = right)P(s = right) = 0.5 * 0.3 + .1 * .7 = 0.22$$

Now we can plug in all parts of Bayes rule:

$$P(s = left|m = fish) = \frac{P(m = fish|s = left)P(s = left)}{P(m = fish)} = \frac{0.5 * 0.3}{0.22} = 0.68$$

Calculate the posterior probability that the school is on the right if the fisher-person does not catch a fish: $p(s = \text{right}|m = \text{no fish})$. Using Bayes rule, we know that

$$P(s = right|m = nofish) = \frac{P(m = nofish|s = right)P(s = right)}{P(m = nofish)}$$

Let's first compute P(m = no fish):

$$P(m = nofish) = P(m = nofish|s = left)P(s = left) + P(m = nofish|s = right)P(s = right) = 0.5 * 0.3 + .9 * .7 = 0.78$$

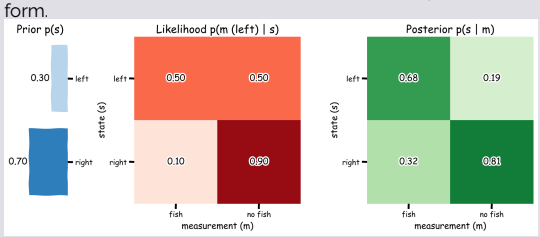
Now we can plug in all parts of Bayes rule:

$$P(s = right|m = nofish) = \frac{P(m = nofish|s = right)P(s = right)}{P(m = nofish)} = \frac{0.9 * 0.7}{0.78} = 0.81$$

Bayes with a binary hidden state

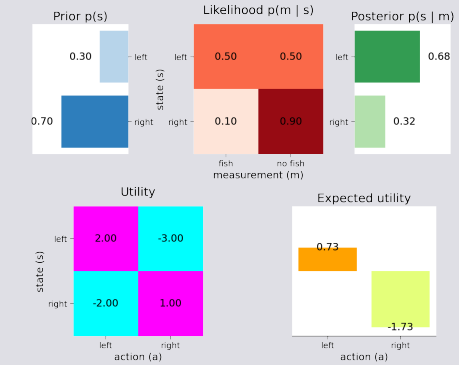
Computing Posteriors

Let's implement our above math to be able to compute posteriors for different priors and likelihoods in graphical form.



Making Bayesian fishing decisions

We consider the expected utility of an action based on our belief (the posterior distribution) about where we think the fish are. Now we have all the components of a Bayesian decision: our prior information, the likelihood given a measurement, the posterior distribution (belief) and our utility (the gains and losses). This allows us to consider the relationship between the true value of the hidden state, s, and what we *expect* to get if we take action, a, based on our belief!



Bayesian inference and decisions with continuous hidden state

Astrocat!

Let's say you are a cat astronaut - Astrocat! You are navigating around space using a jetpack and your goal is to chase a mouse.
Since you are a cat, you don't have opposable thumbs and cannot control your own jet pack. It can only be controlled by ground control back on Earth.
For them to be able to guide you, they need to know where you are. They are trying to figure out your location. They have prior knowledge of your location - they know you like to hang out near the space mouse. They can also get an unreliable quick glimpse: they are taking a measurement of the hidden state of your location.
They will try to figure out your location using Bayes rule and Bayesian decisions - as we will see throughout this tutorial.

Astrocat is in space and we are considering the position along one dimension. So, the hidden state, s, is the true location. The satellites represent potential loss, and the space mouse, gain. Using indirect measurements, you as ground control, can estimate where Astrocat is or decide where it's likely better to send Astrocat.
Remember, in this example, you can think of yourself as a scientist trying to decide where we believe Astrocat is, how to select a point estimate (single guess of location) based on possible errors, and how to account for the uncertainty we have about the location of the satellite and the space mouse. In fact, this is the kind of problem real scientists working to control remote satellites face! However, we can also think of this as what your brain does when it wants to determine a target to make a movement or hit a tennis ball!
A number of classic experiments use this kind of framing to study how "optimal" human decisions or movements are! Some examples are in the further reading document.

Probability distribution of Astrocat location

We are going to think first about how Ground Control should estimate his position. We won't consider measurements yet, just how to represent the uncertainty we might have in general. We are now dealing with a continuous distribution - Astrocat's location can be any real number. In the last tutorial, we were dealing with a discrete distribution - the fish were either on one side or the other.
So how do we represent the probability of each possible point (an infinite number) where the Astrocat could be? The Bayesian approach can be used on any probability distribution. While many variables in the world require representation using complex or unknown (e.g. empirical) distributions, we will be using the Gaussian distributions or extensions of it.

Bayesian inference and decisions with continuous hidden state

The Gaussian distribution

Tne distribution we will use throughout this tutorial is the **Gaussian distribution**, which is also sometimes called the normal distribution.
This is a special, and commonly used, distribution for a couple reasons. It is actually the focus of one of the most important theorems in statistics: the Central Limit Theorem. This theorem tells us that if you sum a large number of samples of a variable, that sum is normally distributed *no matter what* the original distribution over a variable was. This is a bit too in-depth for us to get into now but check out links in the Bonus for more information. Additionally, Gaussians have some really nice mathematical properties that permit simple closed-form solutions to several important problems. As we will see later in this tutorial, we can extend Gaussians to be even more flexible and well approximate other distributions using mixtures of Gaussians. In short, the Gaussian is probably the most important continuous distribution to understand and use.
Gaussians have two parameters. The **mean** μ, which sets the location of its center. Its "scale" or spread is controlled by its **standard deviation** σ or its square, the **variance** σ^2. These can be a bit easy to mix up: make sure you are careful about whether you are referring to/using standard deviation or variance.
The equation for a Gaussian distribution on a variable x is:

$$\mathcal{N}(\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right) \tag{113}$$

In our example, x is the location of the Astrocat in one direction. $\mathcal{N}(\mu, \sigma^2)$ is a standard notation to refer to a $\mathcal{N}$ormal (Gaussian) distribution. For example, $\mathcal{N}(0, 1)$ denotes a Gaussian distribution with mean 0 and variance 1.

Multiplying Gaussians

When we multiply Gaussians, we are not multiplying random variables but the actual underlying distributions. If we multiply two Gaussian distributions, with means μ_1 and μ_2 and standard deviations σ_1 and σ_2, we get another Gaussian. The Gaussian resulting from the multiplication will have mean μ_3 and standard deviation σ_3 where:

$$\mu_3 = a\mu_1 + (1-a)\mu_2 \tag{114}$$

$$\sigma_3^{-2} = \sigma_1^{-2} + \sigma_2^{-2} \tag{115}$$

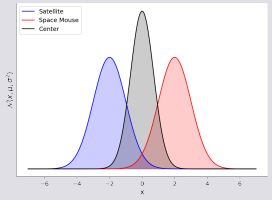
$$a = \frac{\sigma_1^{-2}}{\sigma_1^{-2} + \sigma_2^{-2}} \tag{116}$$

This may look confusing but keep in mind that the information in a Gaussian is the inverse of its variance: $\frac{1}{\sigma^2}$. Basically, when multiplying Gaussians, the mean of the resulting Gaussian is a weighted average of the original means, where the weights are proportional to the amount of information of that Gaussian.

Bayesian inference and decisions with continuous hidden state

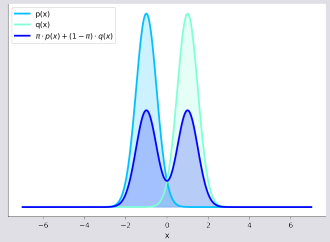
Multiplying Gaussians

Multiplying two Gaussians, imagine we want to find the middle location between the satellite and the space mouse. This would be the center (average) of the two locations. Because we have uncertainty, we need to weigh our uncertainty in thinking about the most likely place.



Mixtures of Gaussians

What if our continuous distribution isn't well described by a single bump? For example, what if the Astrocat is often either in one place or another - a Gaussian distribution would not capture this well! We need a multimodal distribution. Luckily, we can extend Gaussians into a *mixture of Gaussians*, which are more complex distributions.
In a Gaussian mixture distribution, you are essentially adding two or more weighted standard Gaussian distributions (and then normalizing so everything integrates to 1). Each standard Gaussian involved is described, as normal, by its mean and standard deviation. Additional parameters in a mixture of Gaussians are the weights you put on each Gaussian (). The following demo should help clarify how a mixture of Gaussians relates to the standard Gaussian components. We will not cover the derivation here but you can work it out as a bonus exercise.
Mixture distributions are a common tool in Bayesian modeling and an important tool in general.
We will examining a mixture of two Gaussians. We will have one weighting parameter, π, that tells us how to weigh one of the Gaussians. The other is weighted by $1 - \pi$.



Bayesian inference and decisions with continuous hidden state

Utility Loss Estimators

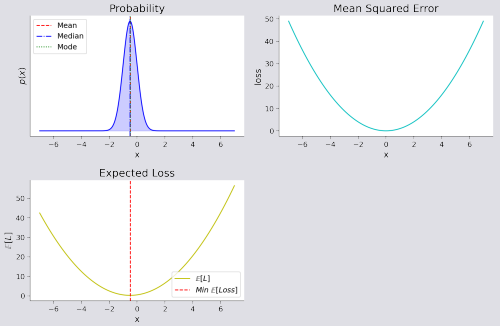
There are lots of different possible loss functions. We will focus on three: **mean-squared error** where the loss is the difference between truth and estimate squared, **absolute error** where the loss is the absolute difference between truth and estimate, and **Zero-one Loss** where the loss is 1 unless we're exactly right (the estimate equals the truth). We can represent these with the following formulas:

$$\text{Mean Squared Error} = (\mu - \hat{\mu})^2 \quad (117)$$

$$\text{Absolute Error} = |\mu - \hat{\mu}| \quad (118)$$

$$\text{Zero-One Loss} = \begin{cases} 0, & \text{if } \mu = \hat{\mu} \\ 1, & \text{otherwise} \end{cases} \quad (119)$$

We will now explore how these different loss functions change our expected utility!

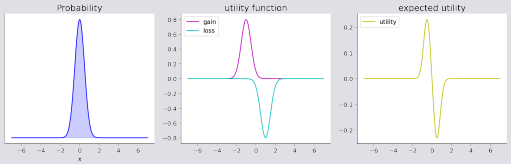


You can see that what coordinates you would provide for Astrocat aren't necessarily easy to guess just from the probability distribution. You need the concept of utility/loss and a specific loss function to determine what estimate you should give.
For symmetric distributions, you will find that the mean, median and mode are the same. However, for distributions with *skew*, like the Gamma distribution or the Exponential distribution, these will be different. You will be able to explore more distributions as priors below.

Bayesian inference and decisions with continuous hidden state

A more complex loss function

The loss functions we just explored were fairly simple and are often used. However, life can be complicated and in this case, Astrocat cares about both being near the space mouse and avoiding the satellites. This means we need a more complex loss function that captures this!
We know that we want to estimate Astrocat to be closer to the mouse, which is safe and desirable, but further away from the satellites, which is dangerous! So, rather than thinking about the "Loss" function, we will consider a generalized utility function that considers gains and losses that *matter* to Astrocat!



Correlation and marginalization

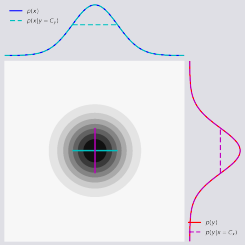
If the two variables in a two dimensional Gaussian are independent, looking at one tells us nothing about the other. But what if the the two variables are correlated (covary)? The covariance of two Gaussians with means μ_X and μ_Y and standard deviations σ_X and σ_Y is:

$$\sigma_{XY} = E[(X - \mu_X)(Y - \mu_Y)] \quad (120)$$

$E[\cdot]$ here denotes the expected value. So the covariance is the expected value of the random variable X minus the mean of the Gaussian distribution on X times the random variable Y minus the mean of the Gaussian distribution on Y.
The correlation is the covariance normalized, so that it goes between -1 (exactly anticorrelated) to 1 (exactly correlated).

$$\rho_{XY} = \frac{\sigma_{XY}}{\sigma_X \sigma_Y} \quad (121)$$

These are key concepts and while we are considering two hidden states (or two random variables), they extend to N dimensional vectors of Gaussian random variables. You will find these used all over computational neuroscience.



Bayesian inference and decisions with continuous hidden state

Bayes' theorem for continuous distributions

The continuous case allows us to consider how the shape of the posterior distribution can differ from the prior. The Gaussian case is the most fundamental, but asymmetric priors (or likelihoods) and posteriors allow us to see how the mean, median and mode can be affected differently when we apply Bayes' theorem.

The Gaussian example

Bayes' rule tells us how to combine two sources of information: the prior (e.g., a noisy representation of Ground Control's expectations about where Astrocat is) and the likelihood (e.g., a noisy representation of the Astrocat after taking a measurement), to obtain a posterior distribution (our belief distribution) taking into account both pieces of information. Remember Bayes' rule:

$$\text{Posterior} = \frac{\text{Likelihood} \times \text{Prior}}{\text{Normalization constant}} \quad (122)$$

We will look at what happens when both the prior and likelihood are Gaussians. In these equations, $\mathcal{N}(\mu, \sigma^2)$ denotes a Gaussian distribution with parameters μ and σ^2:

$$\mathcal{N}(\mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right) \quad (123)$$

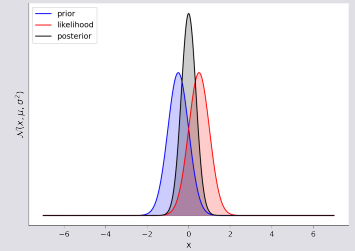
When both the prior and likelihood are Gaussians, Bayes Rule translates into the following form:

$$\text{Likelihood} = \mathcal{N}(\mu_{likelihood}, \sigma^2_{likelihood})$$

$$\text{Prior} = \mathcal{N}(\mu_{prior}, \sigma^2_{prior})$$

$$\text{Posterior} = \mathcal{N}\left(\frac{\sigma^2_{likelihood}\mu_{prior} + \sigma^2_{prior}\mu_{likelihood}}{\sigma^2_{likelihood} + \sigma^2_{prior}}, \frac{\sigma^2_{likelihood}\sigma^2_{prior}}{\sigma^2_{likelihood} + \sigma^2_{prior}}\right)$$

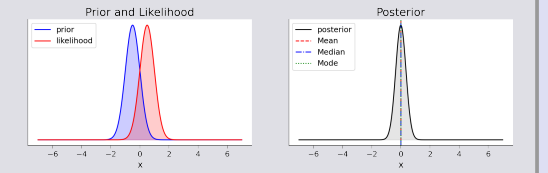
$$\propto \mathcal{N}(\mu_{likelihood}, \sigma^2_{likelihood}) \times \mathcal{N}(\mu_{prior}, \sigma^2_{prior})$$

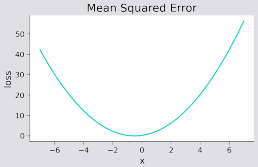


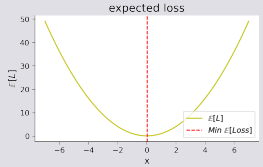
Bayesian inference and decisions with continuous hidden state

Bayesian estimation on the posterior

Bayesian decisions in continuous dimensions are the same as for the binary case. The only difference is that now, our Expected Utility is calculated using an integral and all of our probability distributions are continuous.

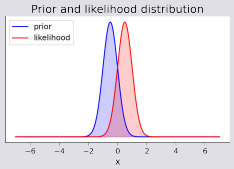


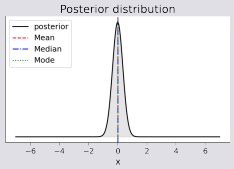


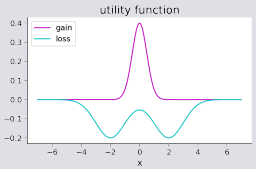


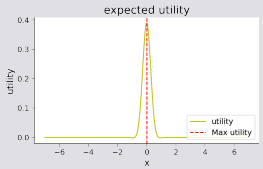
Bayesian decisions

Finally, we can combine everything we have learned so far! Now, let's imagine we have just received a new measurement of Astrocat's location. We need to think about how we want to decide where Astrocat is, so that we can decide how far to tell Astrocat to move. However, we want to account for the satellite and Space Mouse location in this estimation. If we make an error towards the satellite, it's worse than towards Space Mouse. So, we will use our more complex utility function than before.









Neuromatch Academy: Hidden Dynamics - Summary Sheet[14]

Sequential Probability Ratio Test

Overview

Here we will learn about Hidden Markov Models (HMMs), which allow us to infer things in the world from a stream of data. For the binary case, we start with a simple version where the latent state doesn't change, then we'll allow the latent state to change over time. The core learning objective is to understand and implement an algorithm to infer a changing hidden state from observations.
The HMM combines ideas from the linear dynamics lessons (which used Markov models) with inferences described in the Bayes day (which used Hidden variables). It also connects directly to later lessons in Optimal Control and Reinforcement Learning, which often use the HMM to guide actions.
The HMM is a pervasive model in neuroscience. It is used for data analysis, like inferring neural activity from fluorescence images. It is also a foundational model for what the brain should compute, as it interprets the physical world that is observed only through its senses.
Bayes' Theorem combines the sensory measurement m about a latent variable s with our prior knowledge. This produced a posterior probability distribution $p(s|m)$. Here we will allow for dynamic world states and measurements.
We will assume that the world state is binary (± 1) and constant over time, but allow for multiple observations over time. We will use the **Sequential Probability Ratio Test (SPRT)** to infer which state is true. This leads to the **Drift Diffusion Model (DDM)** where evidence accumulates until reaching a stopping criterion.

Sequential Probability Ratio Test

Sequential Probability Ratio Test as a Drift Diffusion Model

The Sequential Probability Ratio Test is a likelihood ratio test for determining which of two hypotheses is more likely. It is appropriate for sequential independent and identically distributed (iid) data. iid means that the data comes from the same distribution.
Let's return to what we learned yesterday. We had probabilities of our measurement (m) given a state of the world (s). For example, we knew the probability of seeing someone catch a fish while fishing on the left side given that the fish were on the left side $P(m = \text{catch fish}|s = \text{left})$.
Now let's extend this slightly to assume we take a series of measurements, from time 1 up to time t ($m_{1:t}$), and that our state is either +1 or -1. We want to figure out what the state is, given our measurements. To do this, we can compare the total evidence up to time t for our two hypotheses (that the state is +1 or that the state is -1). We do this by computing a likelihood ratio: the ratio of the likelihood of all these measurements given the state is +1, $p(m_{1:t}|s = +1)$, to the likelihood of the measurements given the state is -1, $p(m_{1:t}|s = -1)$. This is our likelihood ratio test. In fact, we want to take the log of this likelihood ratio to give us the log likelihood ratio L_T.

$$L_T = log \frac{p(m_{1:t}|s = +1)}{p(m_{1:t}|s = -1)}$$

Since our data is independent and identically distribution, the probability of all measurements given the state equals the product of the separate probabilities of each measurement given the state ($p(m_{1:t}|s) = \prod_{t=1}^{T} p(m_t|s)$). We can substitute this in and use log properties to convert to a sum.

$$\begin{aligned} L_T &= log \frac{p(m_{1:t}|s = +1)}{p(m_{1:t}|s = -1)} \\ &= log \frac{\prod_{t=1}^{T} p(m_t|s = +1)}{\prod_{t=1}^{T} p(m_t|s = -1)} \\ &= \sum_{t=1}^{T} log \frac{p(m_t|s = +1)}{p(m_t|s = -1)} \\ &= \sum_{t=1}^{T} \Delta_t \end{aligned}$$

In the last line, we have used $\Delta_t = log \frac{p(m_t|s=+1)}{p(m_t|s=-1)}$.

Sequential Probability Ratio Test

Sequential Probability Ratio Test as a Drift Diffusion Model

To get the full log likelihood ratio, we are summing up the log likelihood ratios at each time step. The log likelihood ratio at a time step (L_T) will equal the ratio at the previous time step (L_{T-1}) plus the ratio for the measurement at that time step, given by Δ_T:

$$L_T = L_{T-1} + \Delta_T$$

The SPRT states that if L_T is positive, then the state $s = +1$ is more likely than $s = -1$!

Sequential Probability Ratio Test

Let's assume that the probability of seeing a measurement given the state is a Gaussian (Normal) distribution where the mean (μ) is different for the two states but the standard deviation (σ) is the same:

$$p(m_t|s = +1) = \mathcal{N}(\mu, \sigma^2)$$
$$p(m_t|s = -1) = \mathcal{N}(-\mu, \sigma^2)$$

We can write the new evidence (the log likelihood ratio for the measurement at time t) as

$$\Delta_t = b + c\epsilon_t$$

The first term, b, is a consistant value and equals $b = 2\mu^2/\sigma^2$. This term favors the actual hidden state. The second term, $c\epsilon_t$ where $\epsilon_t \sim \mathcal{N}(0, 1)$, is a standard random variable which is scaled by the diffusion $c = 2\mu/\sigma$. You can work through proving this in the bonus exercise 0 below if you wish!
The accumulation of evidence will thus "drift" toward one outcome, while "diffusing" in random directions, hence the term "drift-diffusion model" (DDM). The process is most likely (but not guaranteed) to reach the correct outcome eventually.
Adding these Δ_t over time gives

$$L_T \sim \mathcal{N}\left(2\frac{\mu^2}{\sigma^2}T,\ 4\frac{\mu^2}{\sigma^2}T\right) = \mathcal{N}(bT, c^2T)$$

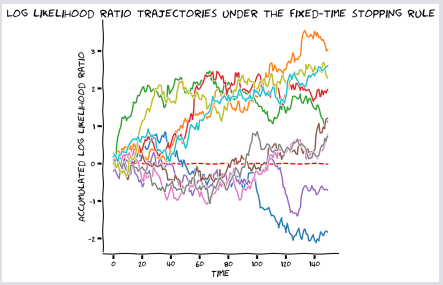
as claimed. The log-likelihood ratio L_t is a biased random walk — normally distributed with a time-dependent mean and variance. This is the **Drift Diffusion Model**.

[14] 't Hart et al., (2022). Neuromatch Academy: a 3-week, online summer school in computational neuroscience. Journal of Open Source Education, 5(49), 118. https://doi.org/10.21105/jose.00118

Sequential Probability Ratio Test

Plotting an SPRT model

Let's now generate simulated data with $s = +1$ and see if the SPRT can infer the state correctly.
The plot below visualizes 10 simulations of the DDM.



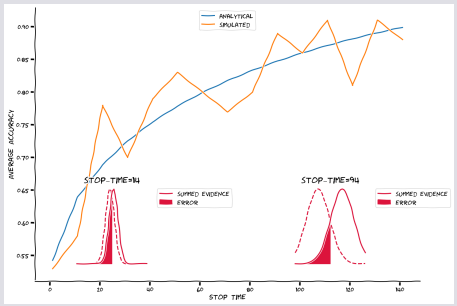
Note:

1. Higher noise, or higher sigma (σ), means that the evidence accumulation varies up and down more. You are more likely to make a wrong decision with high noise, since the accumulated log likelihood ratio is more likely to be negative at the end despite the true distribution being s = +1.
2. When sigma (σ) is very small, the cumulated log likelihood ratios are basically a linear diagonal line. This is because each new measurement will be very similar (since they are being drawn from a Gaussian with a tiny standard deviation)
3. You are more likely to be wrong with a small number of time steps before decision. There is more change that the noise will affect the decision. We will explore this in the next section.

Sequential Probability Ratio Test

Analyzing the DDM: accuracy vs stopping time

If you make a hasty decision (e.g., after only seeing 2 samples), or if observation noise buries the signal, you may see a negative accumulated log likelihood ratio and thus make a wrong decision. Let's plot how decision accuracy varies with the number of samples. Accuracy is the proportion of correct trials across our repeated simulations: $\frac{\#\text{ correct decisions}}{\#\text{ total decisions}}$.



In the figure above, we are plotting the simulated accuracies in orange. We can actually find an analytical equation for the average accuracy in this specific case, which we plot in blue. We will not dive into this analytical solution here but you can imagine that if you ran a bunch of different simulations and had the equivalent number of orange lines, the average of those would resemble the blue line.
In the insets, we are showing the evidence distributions for the two states at a certain time point. Recall from Section 1 that the likelihood ratio at time T for state of +1 is:

$$L_T \sim \mathcal{N}\left(2\frac{\mu^2}{\sigma^2}T,\ 4\frac{\mu^2}{\sigma^2}T\right) = \mathcal{N}(bT, c^2T) \qquad (124)$$

If the state is -1, the mean is the reverse sign. We are plotting this Gaussian distribution for the state equaling -1 (dashed line) and the state equaling +1 (solid line). The area in red reflects the error rate - this region corresponds to L_T being below 0 even though the true state is +1 so you would decide on the wrong state. As more time goes by, these distributions separate more and the error is lower.

Application

We have looked at the drift diffusion model of decisions in the context of the fishing problem. There are lots of uses of this in neuroscience! As one example, a classic experimental task in neuroscience is the random dot kinematogram [Newsome, Britten, Movshon 1989], in which a pattern of moving dots are moving in random directions but with some weak coherence that favors a net rightward or leftward motion. The observer must guess the direction. Neurons in the brain are informative about this task, and have responses that correlate with the choice, as predicted by the Drift Diffusion Model (Huk and Shadlen 2005).

Hidden Markov Model

Application

The world around us is often changing, but we only have noisy sensory measurements. Similarly, neural systems switch between discrete states (e.g. sleep/wake) which are observable only indirectly, through their impact on neural activity. **Hidden Markov Models (HMM)** let us reason about these unobserved (also called hidden or latent) states using a time series of measurements.
Here we'll learn how changing the HMM's transition probability and measurement noise impacts the data. We'll look at how uncertainty increases as we predict the future, and how to gain information from the measurements.
We will use a binary latent variable $s_t \in \{0, 1\}$ that switches randomly between the two states, and a 1D Gaussian emission model $m_t|s_t \sim \mathcal{N}(\mu_{s_t}, \sigma^2_{s_t})$ that provides evidence about the current state.

Binary HMM with Gaussian measurements

Here the latent state in an HMM is not fixed, but may switch to a different state at each time step. The time dependence is simple: the probability of the state at time t is wholly determined by the state at time $t-1$. This is called called the **Markov property** and the dependency of the whole state sequence $\{s_1, ..., s_t\}$ can be described by a chain structure called a Markov Chain.

Markov model for binary latent dynamics

Let's reuse the binary switching process: our state can be either +1 or -1. The probability of switching to state $s_t = j$ from the previous state $s_{t-1} = i$ is the conditional probability distribution $p(s_t = j|s_{t-1} = i)$. We can summarize these as a 2×2 matrix we will denote D for Dynamics.

$$D = \begin{bmatrix} p(s_t = +1|s_{t-1} = +1) & p(s_t = -1|s_{t-1} = +1) \\ p(s_t = +1|s_{t-1} = -1) & p(s_t = -1|s_{t-1} = -1) \end{bmatrix}$$

D_{ij} represents the transition probability to switch from state i to state j at next time step.
We can represent the probability of the *current* state as a 2-dimensional vector

$$P_t = [p(s_t = +1), p(s_t = -1)]$$

The entries are the probability that the current state is +1 and the probability that the current state is -1 so these must sum up to 1.
We then update the probabilities over time following the Markov process:

$$P_t = P_{t-1} D$$

If you know the state, the entries of P_{t-1} would be either 1 or 0 as there is no uncertainty.
Measurements
In a *Hidden Markov Model*, we cannot directly observe the latent states s_t. Instead we get noisy measurements $m_t \sim p(m|s_t)$.

Hidden Markov Model

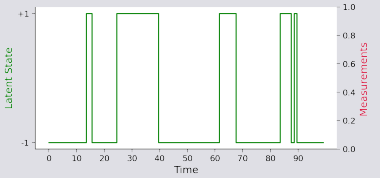
Simulate a binary HMM with Gaussian measurements

To implement a binary HMM with Gaussian measurements. Your HMM will start in State +1 and transition between states (both $-1 \to 1$ and $1 \to -1$) with probability switch probability. Each state emits measurements drawn from a Gaussian with mean $+1$ for State +1 and mean -1 for State -1. The standard deviation of both states is given by noise level.
To implement a binary HMM we have three steps:
STEP 1. Create the transition matrix

$$D = \begin{pmatrix} p_{\text{stay}} & p_{\text{switch}} \\ p_{\text{switch}} & p_{\text{stay}} \end{pmatrix} \tag{125}$$

with $p_{\text{stay}} = 1 - p_{\text{switch}}$.
STEP 2. Specify gaussian measurements $m_t|s_t$, by specifying the means for each state, and the standard deviation.
STEP 3. Use the transition matrix to specify the probabilities for the next state s_t given the previous state s_{t-1}.

The plot below shows an example HMM simulation.



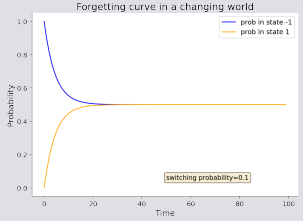
Applications

Measurements could be:

- fish caught at different times as the school of fish moves from left to right
- membrane voltage when an ion channel changes between open and closed
- EEG frequency measurements as the brain moves between sleep stat

Forgetting in a changing world

Even if we know the world state for sure, the world changes. We become less and less certain as time goes by since our last measurement. We'll plot how a Hidden Markov Model gradually "forgets" the current state when predicting the future without measurements.



Hidden Markov Model

Forward inference of HMM

As a recursive algorithm, let's assume we already have yesterday's posterior from time $t-1$: $p(s_{t-1}|m_{1:t-1})$. When the new data m_t comes in, the algorithm performs the following steps:

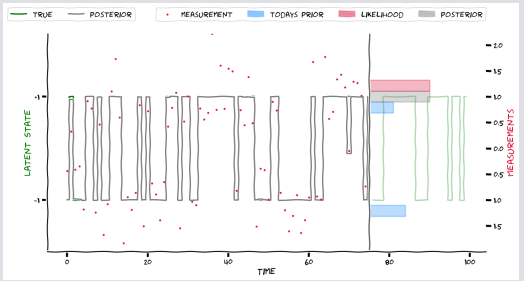
Predict: transform yesterday's posterior over s_{t-1} into today's prior over s_t using the transition matrix D:

$$\text{today's prior} = p(s_t|m_{1:t-1}) = p(s_{t-1}|m_{1:t-1})D$$

Update: Incorporate measurement m_t to calculate the posterior $p(s_t|m_{0:t})$

$$\text{posterior} \propto \text{prior} \cdot \text{likelihood} = p(m_t|s_t)p(s_t|m_{0:t-1})$$

The plot below shows an example HMM simulation.



The Kalman Filter

Application

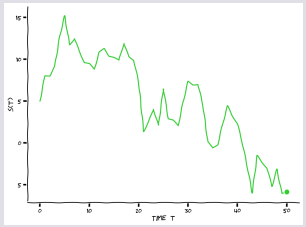
We have used Hidden Markov Models (HMM) to infer *discrete* latent states from a sequence of measurements. Here, we will learn how to infer a *continuous* latent variable using the **Kalman filter**, which is one version of an HMM.
You can imagine this inference process happening as Mission Control tries to locate and track Astrocat. But you can also imagine that the brain is using an analogous Hidden Markov Model to track objects in the world, or to estimate the consequences of its own actions. And you could use this technique to estimate brain activity from noisy measurements, for understanding or for building a brain-machine interface.

Simulating Astrocat's movements

We will simulate how Astrocat moves based on stochastic linear dynamics.
The linear dynamical system

$$s_t = Ds_{t-1} + w_{t-1}$$

determines Astrocat's position s_t. D is a scalar that models how Astrocat would like to change its position over time, and $w_t \sim \mathcal{N}(0, \sigma_p^2)$ is white Gaussian noise caused by unreliable actuators in Astrocat's propulsion unit.



1. When D is large, the state at time step t will depend heavily on the state at time step t_1. If we forget about the noise term, D = 2 would mean that the state at each time step is double the one before! So the state becomes huge and basically explodes towards infinity.
2. If D is a large negative number, the state at time t will be a different sign than the state at time step t_1. So the state will oscillate over the x axis.
3. When D is zero, the state at time t will not depend on the previous state, it will just be drawn from the noise distribution.

The Kalman Filter

Implementing a Kalman filter

A Kalman filter estimates a posterior probability distribution *recursively* over time using a mathematical model of the process and incoming measurements. This dynamic posterior allows us to improve our guess about Astrocat's position as new measures arrive; besides, its mean is the best estimate one can compute of Astrocat's actual position at each time step.
Follow this recipe to implement your own Kalman filter:
Step 1: Change yesterday's posterior into today's prior

Use the mathematical model to calculate how deterministic changes in the process shift yesterday's posterior, $\mathcal{N}(\mu_{s_{t-1}}, \sigma^2_{s_{t-1}})$, and how random changes in the process broaden the shifted distribution:

$$\begin{aligned} p(s_t|m_{1:t-1}) = & \quad p(Ds_{t-1} + w_{t-1}|m_{1:t-1}) \\ = & \quad \mathcal{N}(D\mu_{s_{t-1}} + 0, D^2\sigma^2_{s_{t-1}} + \sigma_p^2) \end{aligned}$$

Note that we use σ_p here to denote the process noise.
Step 2: Multiply today's prior by likelihood

Use the latest measurement of Astrocat's collar (fresh evidence) to form a new estimate somewhere between this measurement and what we predicted in Step 1. The next posterior is the result of multiplying the Gaussian computed in Step 1 (a.k.a. today's prior) and the likelihood, which is also modeled as a Gaussian $\mathcal{N}(m_t, \sigma_m^2)$:
Step 2a: add information from prior and likelihood

To find the posterior variance, we first compute the posterior information (which is the inverse of the variance) by adding the information provided by the prior and the likelihood:

$$\frac{1}{\sigma^2_{s_t}} = \frac{1}{D^2\sigma^2_{s_{t-1}} + \sigma_p^2} + \frac{1}{\sigma_m^2} \tag{126}$$

Now we can take the inverse of the posterior information to get back the posterior variance.
Step 2b: add means from prior and likelihood

To find the posterior mean, we calculate a weighted average of means from prior and likelihood, where each weight, g, is just the fraction of information that each Gaussian provides!

$$g_{\text{prior}} = \frac{\text{information}_{\text{prior}}}{\text{information}_{\text{posterior}}} \tag{127}$$

$$g_{\text{likelihood}} = \frac{\text{information}_{\text{likelihood}}}{\text{information}_{\text{posterior}}} \tag{128}$$

$$\bar{\mu}_t = g_{\text{prior}} D\mu_{s_{t-1}} + g_{\text{likelihood}} m_t \tag{129}$$

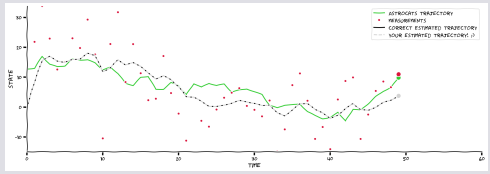
The Kalman Filter

Relationship to classic description of Kalman filter

We're teaching this recipe because it is interpretable and connects to past lessons about the sum rule and product rule for Gaussians. But the classic description of the Kalman filter is a little different. The above weights, g_{prior} and $g_{\text{likelihood}}$, add up to 1 and can be written one in terms of the other; then, if we let $K = g_{\text{likelihood}}$, the posterior mean can be expressed as:

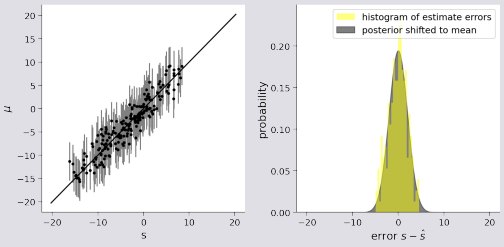
$$\bar{\mu}_t = (1-K)D\bar{\mu}_{t-1} + Km_t = D\bar{\mu}_{t-1} + K(m_t - D\bar{\mu}_{t-1}) \tag{130}$$

In classic textbooks, you will often find this expression for the posterior mean; K is known as the Kalman gain and its function is to choose a value partway between the current measurement m_t and the prediction from Step 1.



Compare states

How well do the estimates $\hat{s}$ match the actual values s? How does the distribution of errors $\hat{s}_t - s_t$ compare to the posterior variance? Why? Try different parameters of the Hidden Markov Model and observe how the properties change.
How do the measurements m compare to the true states?



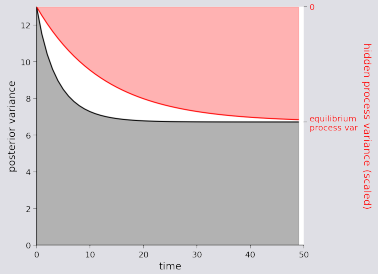
The Kalman Filter

How long does it take to find astrocat?

Here we plot the posterior variance as a function of time. Before mission control gets measurements, their only information about astrocat's location is the prior. After some measurements, they hone in on astrocat.

- How does the variance shrink with time?
- The speed depends on the process dynamics, but does it also depend on the signal-to-noise ratio (SNR)? (Here we measure SNR in decibels, a log scale where 1 dB means 0.1 log unit.)

The red curve shows how rapidly the latent variance equilibrates exponentially from an initial condition, with a time constant of $\sim 1/(1 - D^2)$.



Applications of Kalman filter in brain science

- Brain-Computer Interface: estimate intended movements using neural activity as measurements.
- Data analysis: estimate brain activity from noisy measurements (e.g., EEG)
- Model of perception: prey tracking using noisy sensory measurements
- Imagine your own! When are you trying to estimate something you cannot see directly?

There are many variants that improve upon the limitations of the Kalman filter: non-Gaussian states and measurements, nonlinear dynamics, and more.

Neuromatch Academy: Optimal Control - Summary Sheet[15]

Optimal Control for Discrete States

Overview

Optimal Control combines ideas from the Hidden Dynamics lessons (which used Hidden Markov Models) with maximizing utility described in the Bayes day (which combined a posterior with a utility function). It also connects directly to later lessons in Reinforcement Learning, which learns how to control before you understand the world. In contrast, Optimal Control assumes that you already know how the world works.
Optimal Control is a crucial model in motor neuroscience, because it provides a principled benchmark for how we expect an animal should move. It also is an important engineering method, used for brain-computer interfaces and clamping neurons to desired activity patterns.

Objective

Here, we will implement a **binary control** task: a Partially Observable Markov Decision Process (POMDP) that describes fishing. The agent (you) seeks reward from two fishing sites without directly observing where the school of fish is (yes, a group of fish is called a school!). This makes the world a Hidden Markov Model (HMM). Based on when and where you catch fish, you keep updating your belief about the fish location, i.e., the posterior of the fish given past observations. You should control your position to get the most fish while minimizing the cost of switching sides.
You've already learned about stochastic dynamics, latent states, and measurements. These first exercises largely repeat your previous work. Now we introduce **actions**, based on the new concepts of **control, utility, and policy**. This general structure provides a foundational model for the brain's computations because it includes a perception-action loop where the animal can gather information, draw inferences about its environment, and select actions with the greatest benefit. *How*, mechanistically, the neurons could actually implement these calculations is a separate question we don't address in this lesson.
We will:

- Use the Hidden Markov Models you learned about previously to model the world state.
- Use the observations (fish caught) to build beliefs (posterior distributions) about the fish location.
- Evaluate the quality of different control policies for choosing actions.
- Discover the policy that maximizes utility.

Optimal Control for Discrete States

Analyzing the Problem

Problem Setting

1. State dynamics: There are two possible locations for the fish: Left and Right. Secretly, at each time step, the fish may switch sides with a certain probability $p_{\text{sw}} = 1 - p_{\text{stay}}$. This is the binary switching model (*Telegraph process*) that you've seen in Linear Systems . The fish location, s^{fish}, is latent; you get measurements about it when you try to catch fish, like in Hidden Dynamics. This gives you a *belief* or posterior probability of the current location given your history of measurements.

2. Actions: Unlike past days, you can now **act** on the process! You may stay on your current location (Left or Right), or switch to the other side.

3. Rewards and Costs: You get rewarded for each fish you catch (one fish is worth 1 "point"). If you're on the same side as the fish, you'll catch more, with probability q_{high} per discrete time step. Otherwise, you may still catch some fish with probability q_{low}.

You pay a price of C points for switching to the other side. So you better decide wisely!
Maximizing Utility

To decide "wisely" and maximize your total utility (total points), you will follow a **policy** that prescribes what to do in any situation. Here the situation is determined by your location and your **belief** b_t (posterior) about the fish location (remember that the fish location is a latent variable).
In optimal control theory, the belief is the posterior probability over the latent variable given all the past measurements. It can be shown that maximizing the expected utility with respect to this posterior is optimal.
In our problem, the belief can be represented by a single number because the fish are either on the left or the right side. So we write:

$$b_t = p(s_t^{\text{fish}} = \text{Right} \mid m_{0:t}, a_{0:t-1}) \tag{131}$$

where $m_{0:t}$ are the measurements and $a_{0:t-1}$ are the actions (stay or switch).
Finally, we will parameterize the policy by a simple threshold on beliefs: when your belief that fish are on your current side falls below a threshold θ, you switch to the other side. Here, you will discover that if you pick the right threshold, this simple policy happens to be optimal!

Optimal Control for Discrete States

Analyzing the Problem

Problem Setting

1. State dynamics: There are two possible locations for the fish: Left and Right. Secretly, at each time step, the fish may switch sides with a certain probability $p_{\text{sw}} = 1 - p_{\text{stay}}$. This is the binary switching model (*Telegraph process*) that you've seen in Linear Systems . The fish location, s^{fish}, is latent; you get measurements about it when you try to catch fish, like in Hidden Dynamics. This gives you a *belief* or posterior probability of the current location given your history of measurements.

2. Actions: Unlike past days, you can now **act** on the process! You may stay on your current location (Left or Right), or switch to the other side.

3. Rewards and Costs: You get rewarded for each fish you catch (one fish is worth 1 "point"). If you're on the same side as the fish, you'll catch more, with probability q_{high} per discrete time step. Otherwise, you may still catch some fish with probability q_{low}.

You pay a price of C points for switching to the other side. So you better decide wisely!
Maximizing Utility

To decide "wisely" and maximize your total utility (total points), you will follow a **policy** that prescribes what to do in any situation. Here the situation is determined by your location and your **belief** b_t (posterior) about the fish location (remember that the fish location is a latent variable).
In optimal control theory, the belief is the posterior probability over the latent variable given all the past measurements. It can be shown that maximizing the expected utility with respect to this posterior is optimal.
In our problem, the belief can be represented by a single number because the fish are either on the left or the right side. So we write:

$$b_t = p(s_t^{\text{fish}} = \text{Right} \mid m_{0:t}, a_{0:t-1}) \tag{132}$$

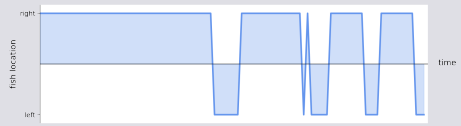
where $m_{0:t}$ are the measurements and $a_{0:t-1}$ are the actions (stay or switch).
Finally, we will parameterize the policy by a simple threshold on beliefs: when your belief that fish are on your current side falls below a threshold θ, you switch to the other side. Here, you will discover that if you pick the right threshold, this simple policy happens to be optimal!

[15] 't Hart et al., (2022). Neuromatch Academy: a 3-week, online summer school in computational neuroscience. Journal of Open Source Education, 5(49), 118. https://doi.org/10.21105/jose.00118

Optimal Control for Discrete States

Examining fish dynamics

Look at the dynamics of the fish moving from side to side while you stay in one place. With the probability (stay prob=0.9) of fish staying in the same location, and observe the resulting dynamics of the fish.

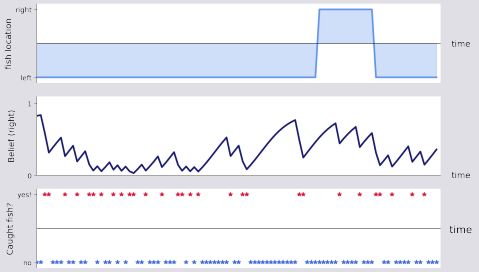


Examining the reward function

You can control your location, but we fix the fish's location by setting 'stay prob = 1'. Now that the fish are serenely swimming in one location, we can visually inspect the rewards when you're on the same side as the fish or on the other side.
When you're on the same side as the fish, you should have a higher probability of catching them (but watch out, since technically, you are *allowed* to adjust the sliders to other conditions!).

Belief dynamics and belief distributions

Now it's time to get an intuition on how beliefs are calculated. Here we define your belief about the fish location is just the posterior probability about that location given your measurements, $p(s_t|m_{0:t})$. Note that this is just like Hidden Dynamics!
If you'll always stay on the LEFT side, but the fish will move around. They'll stay on the same side with probability 'stay prob'. You only get to see fish you catch, not where the school of fish is. You have to use those measurements to infer the location of the school.



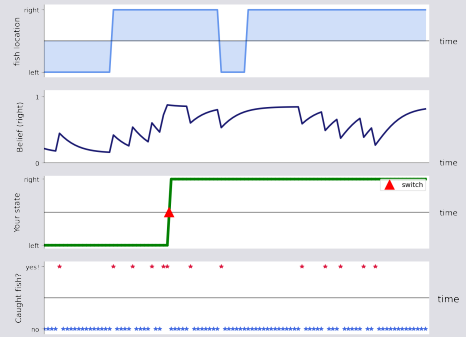
Optimal Control for Discrete States

Implementing a threshold policy

Now we'll switch the policy from the lazy policy used above to a threshold policy. You'll change your location whenever your belief is low enough that you're on the best side. This policy takes three inputs:

1. The *belief* about the fish state. For convenience, we will represent the belief at time t using a 2-dimensional vector. The first element is the belief that the fish are on the left, and the second element is the belief the fish are on the right. At every time step, these elements sum to 1.
2. Your location, represented as "Left" = -1 and "Right" = 1.
3. A belief *threshold* that determines when to switch. When your belief that you are on the same side as the fish drops below this threshold, you should move to the other location, and otherwise stay.

Dynamics with different thresholds



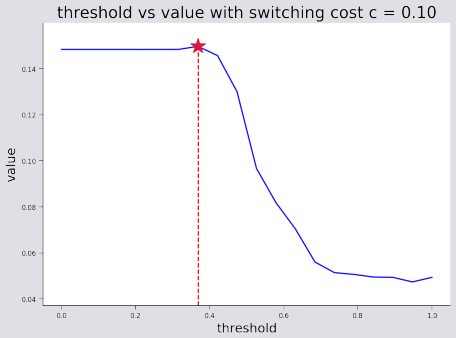
Optimal Control for Discrete States

Implementing a value function

Let's find out how good our threshold is. For that, we will calculate a **value function** that quantifies our utility (total points). We will use this value to compare different thresholds; remember, our goal is to maximize the amount of fish we catch while minimizing the effort involved in changing locations.
The value is the total expected utility per unit time.

$$V(\theta) = \frac{1}{T}\left(\sum_t R(s_t) - C(a_t)\right) \tag{133}$$

where $R(s_t)$ is the instantaneous reward we get at location s_t and $C(a_t)$ is the cost we paid for the chosen action. Remember, we receive one point for fish caught and pay cost points for switching to the other location.
We could take this average mathematically over the probabilities of rewards and actions. However, we can get the same answer by simply averaging the *actual* rewards and costs over a long time.



Summary

In this tutorial, you combined Hidden Markov Models with actions to solve an optimal control problem! This showed us the core formalism of the *Partially Observable Markov Decision Process* (POMDP).
Using observations (fish caught), you built beliefs (posterior distributions) that helped you estimate where the fish were. Next, you computed a value function that helped you evaluate the quality of different policies. Finally, using a brute force approach, you discovered an optimal policy that allowed you to catch as many fish as possible while minimizing the effort of switching your location.
The following tutorial will use continuous states and actions instead of the binary ones we used here. In continuous control, we can still use a POMDP, but we'll focus on control in the fully observed case, a Markov Decision Process (MDP), since the policy is still illuminating.

Optimal Control for Continuous State

Exploring a Linear Dynamical System (LDS) with Open-Loop and Closed-Loop Control

In this example, a cat is trying to catch a mouse in space. The location of the mouse is the goal state g, here a static goal. Later on, we will make the goal time-varying, i.e., $g(t)$. The cat's location is the state of the system s_t. The state has its internal dynamics: think of the cat drifting slowly in space. These dynamics are such that the state at the next time step s_{t+1} are a linear function of the current state s_t. There is some process noise affecting the state (think about engine corrosions on the little jetpack causing unintended movements) here modeled as Gaussian noise w_t.
The control input or action a_t is the action of the jet pack, which has an effect Ba_t on the state at the next time step s_{t+1}. Here, we will be designing the action a_t to reach the goal g, with known state dynamics.
Thus, our linear discrete-time system evolves according to the following equation:

$$\begin{aligned} s_{t+1} &= Ds_t + Ba_t + w_t \\ s_0 &= s_{init} \end{aligned}$$

with,
t: time step, ranging from 1 to T, where T is the time horizon.
s_t: state at time t.
a_t: action at time t (also known as "control input").
w_t: gaussian noise at time t.
D: transition matrix.
B: input matrix.

For simplicity, we will consider the 1D case, where the matrices reduce to scalars, and the states, control and noise are one-dimensional as well. Specifically, D and B are scalars.
We will consider the goal g to be the origin, i.e., $g = 0$, for Exercises 1 and 2.2. Later on, we will explore scenarios where the goal state changes over time $g(t)$.

Stability

The system is stable, i.e., the output remains finite for any finite initial condition s_{init}, if $|D| < 1$. Note that if the state dynamics are stable, the state eventually reaches 0 (No control is needed!). However, when $|D| > 1$ or the goal $g \neq 0$ selecting an appropriate sequence of actions becomes essential to completing the task.

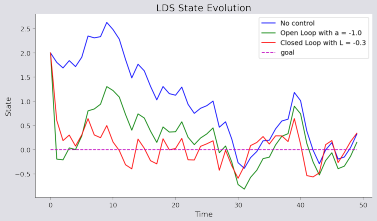
Open-loop control and Closed-loop linear control

In *open-loop control*, a_t is not a function of s_t. In "closed-loop linear control", a_t is a linear function of the state s_t. Specifically, a_t is the control gain, L_t, multiplied by s_t, i.e., $a_t = L_t s_t$.

Optimal Control for Continuous State

Explore no control vs. open-loop control vs. closed-loop control

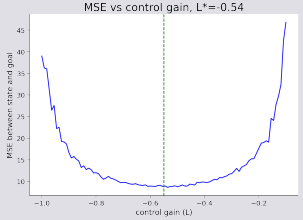
The plot below visualizes the effects of different kinds of control inputs.
1. For the no-control case, can you identify two distinct outcomes, depending on the value of D? Why?
2. The open-loop controller works well–or does it? Are there any problems in in challenging (high noise) conditions.
3. Does the closed-loop controller fare better with the noise? Vary the values of L and find a range where it quickly reaches the goal.



Exploring the closed-loop setting further

The plot below visualizes the MSE between the state and goal, as a function of control gain L. You should see a U-shaped curve, with a minimum MSE. The control gain at which the minimum MSE is reached, is the optimal constant control gain for minimizing MSE, here called the numerical optimum.
A green dashed line is shown $L = -\frac{D}{B}$ with $D = 1.1$ and $B = 2$. Why is this the theoretical optimal control gain for minimizing MSE of the state s to the goal $g = 0$? Examine how the states evolve with a constant gain L.

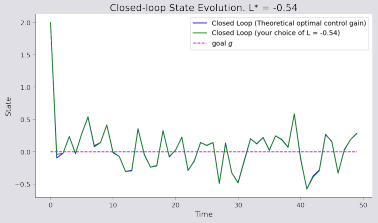
$$\begin{aligned} s_{t+1} &= Ds_t + Ba_t + w_t \\ &= Ds_t + B(Ls_t) + w_t \\ &= (D + BL)s_t + w_t \end{aligned}$$



Optimal Control for Continuous State

Explore no control vs. open-loop control vs. closed-loop control

Now, let's visualize the evolution of the system as we change the control gain.



Designing an optimal control input using a linear quadratic regulator (LQR)

Constraints on the system
Now we will start imposing additional constraints on our system. For example, if you explored different values for s_{init} above, you would have seen very large values for a_t in order to get to the mouse in a short amount of time. However, perhaps the design of our jetpack makes it dangerous to use large amounts of fuel in a single timestep. We certainly do not want to explode, so we would like to keep the actions a_t as small as possible while still maintaining good control.
Moreover, we had restricted ourselves to a static control gain $L_t \equiv L$. How would we vary it if we could?
This leads us to a more principled way of designing the optimal control input.
Setting up a cost function

In a finite-horizon LQR problem, the cost function is defined as:

$$\begin{aligned} J(\mathbf{s}, \mathbf{a}) &= J_{state}(\mathbf{s}) + \rho J_{control}(\mathbf{a}) \\ &= \sum_{t=0}^{T} (s_t - g)^2 + \rho \sum_{t=0}^{T-1} a_t^2 \end{aligned}$$

where ρ is the weight on the control effort cost, as compared to the cost of not being at the goal. Here, $\mathbf{a} = \{a_t\}_{t=0}^{T-1}$, $\mathbf{s} = \{s_t\}_{t=0}^{T}$. This is a quadratic cost function. In Exercise 2, we will only explore $g = 0$, in which case $J_{state}(\mathbf{s})$ can also be expressed as $\sum_{t=0}^{T} s_t^2$.
The goal of the LQR problem is to find control $\mathbf{a}$ such that $J(\mathbf{s}, \mathbf{a})$ is minimized. The goal is then to find the control gain at each time point, i.e.,

$$\text{argmin}_{\{L_t\}_{t=0}^{T-1}} J(\mathbf{s}, \mathbf{a})$$

where $a_t = L_t s_t$.

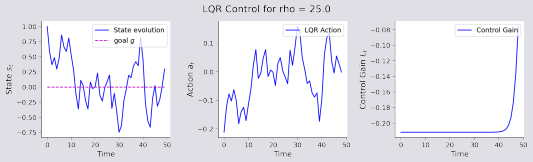
Optimal Control for Continuous State

Designing an optimal control input using a linear quadratic regulator (LQR)

The solution to Equation for LQR for a finite time horizon, can be obtained via Dynamic Programming. For details, check out this lecture by Stephen Boyd.
For an infinite time horizon, one can obtain a closed-form solution using Riccati equations, and the solution for the control gain becomes time-invariant, i.e., $L_t \equiv L$. For details, check out this other lecture by Stephen Boyd.
The cost function $J(\mathbf{s}, \mathbf{a})$ can be divided into two parts: $J_{state}(\mathbf{s})$ and $J_{control}(\mathbf{a})$.

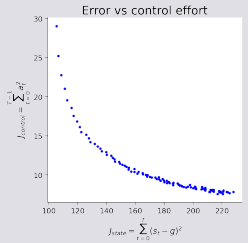
LQR to the origin

Here, we will use LQR controller to track a static goal at $g = 0$. We will explore how varying ρ (the weight on the control effort cost) affects the state trajectory, actions selected, and control gain.



The tradeoff between state cost and control cost

We will now plot them against each other for varying values of ρ to explore the tradeoff between state cost and control cost.



You should notice the bottom half of the curve, forming the tradeoff between the state cost and the control cost under optimal closed-loop linear control.
For a desired value of the state cost, we cannot reach a lower control cost than the curve in the above plot. Similarly, for a desired value of the control cost, we must accept that amount of state cost. For example, if you know that you have a limited amount of fuel, which determines your maximum control cost to be $J_{control}^{max}$.
You will be able to show that you will not be able to track your state with higher accuracy than the corresponding J_{state} as given by the graph above. This is thus an important curve when designing a system and exploring its control.

Optimal Control for Continuous State

LQR for tracking a time-varying goal

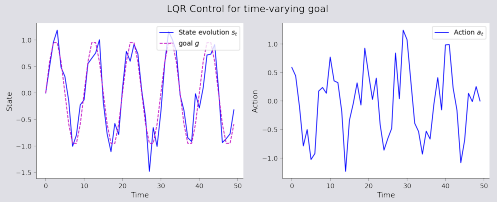
In a more realistic situation, the mouse would move around constantly. Suppose you were able to predict the movement of the mouse as it bounces from one place to another. This becomes your goal trajectory g_t.
When the target state, denoted as g_t, is not 0, the cost function becomes

$$J(\mathbf{a}) = \sum_{t=0}^{T} (s_t - g_t)^2 + \rho \sum_{t=0}^{T-1} (a_t - \bar{a}_t)^2$$

Here, $\bar{a}_t$ is the desired action based on the goal trajectory. In other words, the controller considers the goal for the next time step, and designs a preliminary control action that gets the state at the next time step to the desired goal. Specifically, without taking into account noise w_t, we would like to design $\bar{a}_t$ such that $s_{t+1} = g_{t+1}$. Thus we have

$$\begin{aligned} g_{t+1} &= Ds_t + B\bar{a}_t \\ \bar{a}_t &= \frac{-Ds_t + g_{t+1}}{B} \end{aligned}$$

The final control action a_t is produced by adding this desired action $\bar{a}_t$ with the term with the control gain $L_t(s_t - g_t)$.
The plot shows LQR tracks a time-varying goal for a sinusoidal goal function.



Optimal Control for Continuous State

Control of an partially observed state using a Linear Quadratic Gaussian (LQG) controller

In practice, the controller does not have full access to the state. For example, your jet pack in space may be controlled by Mission Control back on earth! In this case, noisy measurements m_t of the state s_t are taken via radar, and the controller needs to (1) estimate the true state, and (2) design an action based on this estimate.

Fortunately, the separation principle tells us that it is optimal to do (1) and (2) separately. This makes our problem much easier, since we already know how to do each step.

1) *State Estimation*
Can we recover the state from the measurement? yesterday you learned that the states $\hat{s}_t$ can be estimated from the measurements m_t using the **Kalman filter**.

2) *Design Action*
We learned about the LQR controller which designs an action based on the state. The separation principle tells us that it is sufficient to replace the use of the state in LQR with the *estimated* state, i.e.,

$$a_t = L_t \hat{s}_t \tag{134}$$

The state dynamics will then be:

$$s_{t+1} = Ds_t + Ba_t + w_t \tag{135}$$

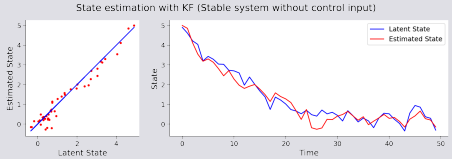
where w_t is the process noise $(\text{proc}_n oise), and the observation/measurement is$:

$$m_t = Cs_t + v_t \tag{136}$$

with C is the observation matrix and v_t is the measurement noise $(\text{meas}_n oise)$.
The combination of (1) state estimation and (2) action design using LQR is known as a **linear quadratic gaussian (LQG)**. Yesterday, you completed the code for the Kalman filter. Based on that, you will code up the LQG controller. For these exercises, we will return to using the goal $g = 0$, as in Section 2.

Optimal Control for Continuous State

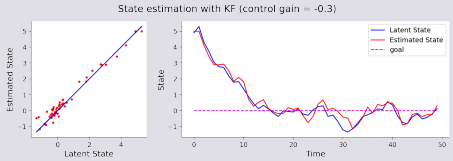
The Kalman filter in conjunction with a linear closed-loop controller (LQG Control)



LQG controller output with varying control gains

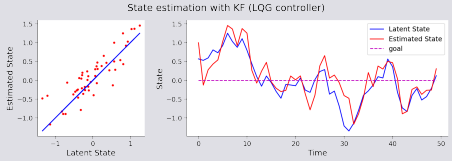
Here the Kalman filter with closed-loop feedback with the controller. We will first use an arbitrary control gain and a fixed value for measurement noise. We will then use the control gain that we calculated for the LQR system given different values for ρ (weight on the control effort).

1. Visualize the system dynamics s_t in closed-loop control with an arbitrary constant control gain. Vary this control gain.
2. Play arround with the remaining sliders. What happens when the process noise is high (low)? How about the measurement noise?



LQG controller with varying weight on the control effort costs

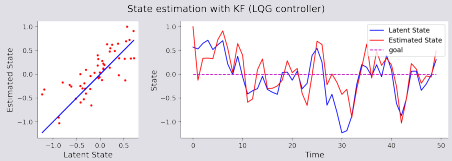
Now let's see the performance of the LQG controller as the parameter ρ changes. We will use an LQG controller gain, where the control gain is from a system with an infinite horizon; in this case, the optimal control gain turns out to be a constant.



Optimal Control for Continuous State

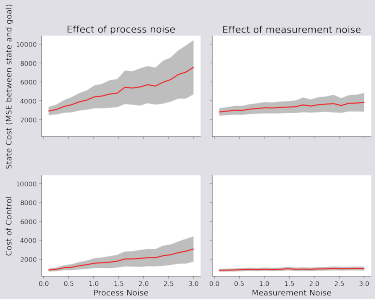
How does the process noise and the measurement noise influence the controlled state and desired action?

Process noise w_t and measurement noise v_t have very different effects on the controlled state.



Noise effects on the LQG

Finally, we will quantify how the state cost and control costs change when we change the process and measurement noise levels. To do so, we will run many simulations, stepping through levels of process and measurement noise, tracking MSE and cost of control for each.
Observe the effects of increasing the process and measurement noises in an unstable system. How do you interpret the results?



While both sources of noise have an effect on the controlled state, the process noise has a much larger effect. As the process noise w[t] increases, state cost (MSE between state and goal) and control cost increase drastically. You can get an intuition as to why using the sliders in the demo above. To make matters worse, as the process noise gets larger, you will also need to put in more effort to keep the system close to the goal. The measurement noise v[t] also has an effect on the accuracy of the controlled state. As this noise increases, the MSE between the state and goal increases. The cost of control in this case remains fairly constant with increasing levels of measurement noise.

Neuromatch Academy: Reinforcement Learning - Summary Sheet[16]

Learning to Predict

Overview

Reinforcement Learning (RL) is a framework for defining and solving a learning problem where an animal or agent knows or infers the state of the world and then learns the value of the states and actions that can be taken in them, by receiving a reward signal. Importantly, reinforcement learning provides formal, optimal descriptions of learning first derived from studies of animal behavior and then validated when the formal quantities used in the model were observed in the brain in humans and animals. It is probably one of the most widely used computational approaches in neuroscience.

Learning to Predict

Here, we will learn how to estimate state-value functions in a classical conditioning paradigm using Temporal Difference (TD) learning and examine TD-errors at the presentation of the conditioned and unconditioned stimulus (CS and US) under different CS-US contingencies. This will provide you with an understanding of both how reward prediction errors (RPEs) behave in classical conditioning and what we should expect to see if Dopamine represents a "canonical" model-free RPE.

Temporal difference learning

Environment:

- The agent experiences the environment in episodes or trials.
- Episodes terminate by transitioning to the inter-trial-interval (ITI) state and they are initiated from the ITI state as well. We clamp the value of the terminal/ITI states to zero.
- The classical conditioning environment is composed of a sequence of states that the agent deterministically transitions through. Starting at State 0, the agent moves to State 1 in the first step, from State 1 to State 2 in the second, and so on. These states represent time in the tapped delay line representation
- Within each episode, the agent is presented with a CS and US (reward).
- The CS is always presented at 1/4 of the total duration of the trial. The US (reward) is then delivered after the CS. The interval between the CS and US is specified by *reward time*.
- The agent's goal is to learn to predict expected rewards from each state in the trial.

Learning to Predict

General concepts

Return G_t: future cumulative reward, which can be written in a recursive form

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{137}$$

$$= r_{t+1} + \gamma G_{t+1} \tag{138}$$

where γ is discount factor that controls the importance of future rewards, and $\gamma \in [0, 1]$. γ may also be interpreted as probability of continuing the trajectory.
Value function $V_\pi(s_t = s)$: expectation of the return

$$V_\pi(s_t = s) = E[G_t \mid s_t = s, a_{t:\infty} \sim \pi] \tag{139}$$

$$= E[r_{t+1} + \gamma G_{t+1} \mid s_t = s, a_{t:\infty} \sim \pi] \tag{140}$$

With an assumption of **Markov process**, we thus have:

$$V_\pi(s_t = s) = E[r_{t+1} + \gamma V_\pi(s_{t+1}) \mid s_t = s, a_{t:\infty} \sim \pi] \tag{141}$$

$$= \sum_a \pi(a|s) \sum_{r,s'} p(s', r)(r + V_\pi(s_{t+1} = s')) \tag{142}$$

Temporal difference (TD) learning

With a Markovian assumption, we can use $V(s_{t+1})$ as an imperfect proxy for the true value G_{t+1} (Monte Carlo bootstrapping), and thus obtain the generalized equation to calculate TD-error:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{143}$$

Value updated by using the learning rate constant α:

$$V(s_t) \leftarrow V(s_t) + \alpha \delta_t \tag{144}$$

Reference: Temporal-Difference Learning

Definitions

TD-error:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{145}$$

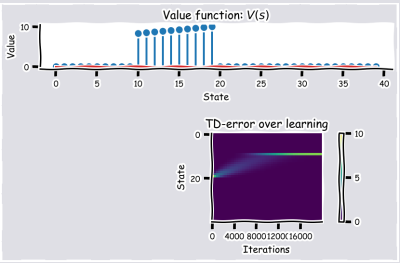
Value updates:

$$V(s_t) \leftarrow V(s_t) + \alpha \delta_t \tag{146}$$

Learning to Predict

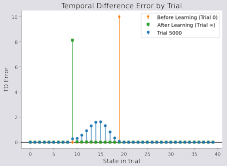
TD-learning with guaranteed rewards

TD-learning to estimate the state-value function in the classical-conditioning world with guaranteed rewards, with a fixed magnitude, at a fixed delay after the conditioned stimulus, CS. Save TD-errors over learning (i.e., over trials) so we can visualize them afterwards.
To simulate the effect of the CS, you should only update $V(s_t)$ during the delay period after CS. This period is indicated by the boolean variable *is delay*. This can be implemented by multiplying the expression for updating the value function by *is delay*.



US to CS Transfer

During classical conditioning, the subject's behavioral response (e.g., salivating) transfers from the unconditioned stimulus (US; like the smell of tasty food) to the conditioned stimulus (CS; like Pavlov ringing his bell) that predicts it. Reward prediction errors play an important role in this process by adjusting the value of states according to their expected, discounted return.
Use the widget below to examine how reward prediction errors change over time.
Before training (orange line), only the reward state has high reward prediction error. As training progresses (blue line, slider), the reward prediction errors shift to the conditioned stimulus, where they end up when the trial is complete (green line).
Dopamine neurons, which are thought to carry reward prediction errors *in vivo*, show exactly the same behavior!

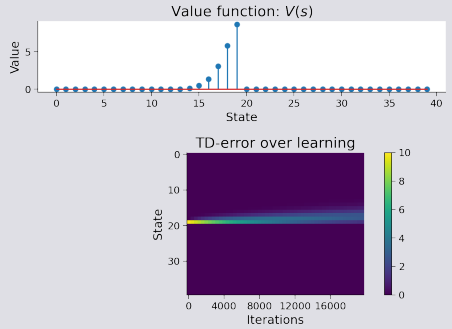


[16] 't Hart et al., (2022). Neuromatch Academy: a 3-week, online summer school in computational neuroscience. Journal of Open Source Education, 5(49), 118. https://doi.org/10.21105/jose.00118

Learning to Predict

Learning Rates and Discount Factors

Our TD-learning agent has two parameters that control how it learns: α, the learning rate, and γ, the discount factor. In Exercise 1, we set these parameters to $\alpha = 0.001$ and $\gamma = 0.98$ for you. Here, you'll investigate how changing these parameters alters the model that TD-learning learns. Before enabling the interactive demo below, take a moment to think about the functions of these two parameters. α controls the size of the Value function updates produced by each TD-error. In our simple, deterministic world, will this affect the final model we learn? Is a larger α necessarily better in more complex, realistic environments?
The discount rate γ applies an exponentially-decaying weight to returns occuring in the future, rather than the present timestep. How does this affect the model we learn? What happens when $\gamma = 0$ or $\gamma \geq 1$?
Use the widget to test your hypotheses.

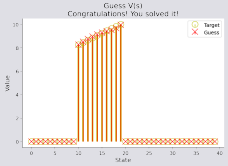


α determines how fast the model learns. In the simple, deterministic world we're using here, this allows the model to quickly converge onto the "true" model that heavily values the conditioned stimulus. In more complex environments, however, excessively large values of alpha can slow, or even prevent, learning, as we'll see later. γ effectively controls how much the model cares about the future: larger values of γ cause the model to weigh future rewards nearly as much as present ones. At γ=1, the model weights all rewards, regardless of when they occur, equally and when greater than one, it starts to prefer rewards in the future, rather than the present (this is rarely good). When γ=0, however, the model becomes greedy and only considers rewards that can be obtained immediately.

Learning to Predict

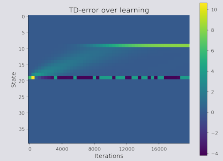
TD-learning with varying reward magnitudes

In the previous exercise, the environment was as simple as possible. On every trial, the CS predicted the same reward, at the same time, with 100% certainty. In the next few exercises, we will make the environment more progressively more complicated and examine the TD-learner's behavior.



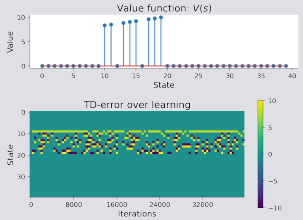
Examining the TD Error

The plot below shows the TD errors from our multi-reward environment. A new feature appears in this plot? What is it? Why does it happen?



TD-learning with probabilistic rewards

In this environment, we'll return to delivering a single reward of ten units. However, it will be delivered intermittently: on 20 percent of trials, the CS will be shown but the agent will not receive the usual reward; the remaining 80% will proceed as usual.
Run the cell below to simulate. How does this compare with the previous experiment?
Earlier in the notebook, we saw that changing α had little effect on learning in a deterministic environment. What happens if you set it to a large value, like 1, in this noisier scenario? Does it seem like it will *ever* converge?



Learning to Predict

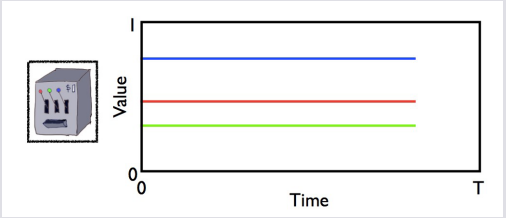
Summary

Here, we have developed a simple TD Learner and examined how its state representations and reward prediction errors evolve during training. By manipulating its environment and parameters (α, γ), you developed an intuition for how it behaves.
This simple model closely resembles the behavior of subjects undergoing classical conditioning tasks and the dopamine neurons that may underlie that behavior. You may have implemented TD-reset or used the model to recreate a common experimental error. The update rule used here has been extensively studied for more than 70 years as a possible explanation for artificial and biological learning.
However, you may have noticed that something is missing. We carefully calculated the value of each state, but did not use it to actually do anything. Using values to plan **Actions** is coming up next!

Learning to Act: Multi-Armed Bandits

Multi-Armed Bandits

Consider the following learning problem. You are faced repeatedly with a choice among k different options, or actions. After each choice you receive a reward signal in the form of a numerical value, where the larger value is the better. Your objective is to maximize the expected total reward over some time period, for example, over 1000 action selections, or time steps.
This is the original form of the k-armed bandit problem. This name derives from the colloquial name for a slot machine, the "one-armed bandit", because it has the one lever to pull, and it is often rigged to take more money than it pays out over time. The multi-armed bandit extension is to imagine, for instance, that you are faced with multiple slot machines that you can play, but only one at a time. Which machine should you play, i.e., which arm should you pull, which action should you take, at any given time to maximize your total payout.



While there are many different levels of sophistication and assumptions in how the rewards are determined, for simplicity's sake we will assume that each action results in a reward drawn from a fixed Gaussian distribution with unknown mean and unit variance. This problem setting is referred to as the *environment*, and the goal is to find the arm with the highest mean value.
We will solve this *optimization problem* with an *agent*, in this case an algorithm that takes in rewards and returns actions.

Learning to Act: Multi-Armed Bandits

Choosing an Action

The first thing our agent needs to be able to do is choose which arm to pull. The strategy for choosing actions based on our expectations is called a "policy" (often denoted π). We could have a random policy – just pick an arm at random each time – though this doesn't seem likely to be capable of optimizing our reward. We want some intentionality, and to do that we need a way of describing our beliefs about the arms' reward potential. We do this with an action-value function

$$q(a) = E[r_t | a_t = a] \tag{147}$$

where the value q for taking action $a \in A$ at time t is equal to the expected value of the reward r_t given that we took action a at that time. In practice, this is often represented as an array of values, where each action's value is a different element in the array.
Great, now that we have a way to describe our beliefs about the values each action should return, let's come up with a policy.
An obvious choice would be to take the action with the highest expected value. This is referred to as the *greedy* policy

$$a_t = \text{argmax}_a \; q_t(a) \tag{148}$$

where our choice action is the one that maximizes the current value function.
So far so good, but it can't be this easy. And, in fact, the greedy policy does have a fatal flaw: it easily gets trapped in local maxima. It never explores to see what it hasn't seen before if one option is already better than the others. This leads us to a fundamental challenge in coming up with effective policies.

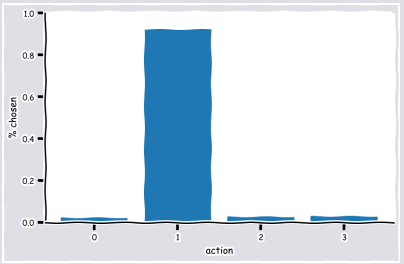
Learning to Act: Multi-Armed Bandits

The Exploitation-Exploration Dilemma

If we never try anything new, if we always stick to the safe bet, we don't know what we are missing. Sometimes we aren't missing much of anything, and regret not sticking with our preferred choice, yet other times we stumble upon something new that was way better than we thought.
This is the exploitation-exploration dilemma: do you go with your best choice now, or risk the less certain option with the hope of finding something better. Too much exploration, however, means you may end up with a sub-optimal reward once it's time to stop.
In order to avoid getting stuck in local minima while also maximizing reward, effective policies need some way to balance between these two aims.
A simple extension to our greedy policy is to add some randomness. For instance, a coin flip – heads we take the best choice now, tails we pick one at random. This is referred to as the ϵ-greedy policy:

$$P(a_t = a) = \begin{cases} 1 - \epsilon + \epsilon/N & \text{if } a_t = \text{argmax}_a \; q_t(a) \\ \epsilon/N & \text{else} \end{cases} \tag{149}$$

which is to say that with probability 1 - ϵ for $\epsilon \in [0, 1]$ we select the greedy choice, and otherwise we select an action at random (including the greedy option).
Despite its relative simplicity, the epsilon-greedy policy is quite effective, which leads to its general popularity.
The plot below shows the epsilon-greedy algorithm for deciding which action to take from a set of possible actions given their value function and a probability ϵ of simply choosing one at random.

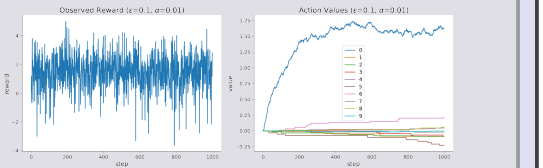


This is what we should expect, that the action with the largest value (action 1) is selected about (1-ϵ) of the time, or 90% for $\epsilon = 0.1$, and the remaining 10% is split evenly amongst the other options. Use the demo below to explore how changing ϵ affects the distribution of selected actions.

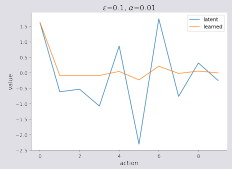
Learning to Act: Multi-Armed Bandits

Solving Multi-Armed Bandits

Now that we have both a policy and a learning rule, we can combine these to solve our original multi-armed bandit task. Recall that we have some number of arms that give rewards drawn from Gaussian distributions with unknown mean and unit variance, and our goal is to find the arm with the highest mean.
We can use our multi-armed bandit method to evaluate how our epsilon-greedy policy and learning rule perform at solving the task. First we will set our environment to have 10 arms and our agent parameters to $\epsilon = 0.1$ and $\alpha = 0.01$. In order to get a good sense of the agent's performance, we will run the episode for 1000 steps.



Alright, we got some rewards that are kind of all over the place, but the agent seemed to settle in on the first arm as the preferred choice of action relatively quickly. Let's see how well we did at recovering the true means of the Gaussian random variables behind the arms.

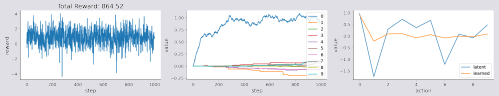


Well, we seem to have found a very good estimate for action 0, but most of the others are not great. In fact, we can see the effect of the local maxima trap at work – the greedy part of our algorithm locked onto action 0, which is actually the 2nd best choice to action 6. Since these are the means of Gaussian random variables, we can see that the overlap between the two would be quite high, so even if we did explore action 6, we may draw a sample that is still lower than our estimate for action 0.
However, this was just one choice of parameters. Perhaps there is a better combination?

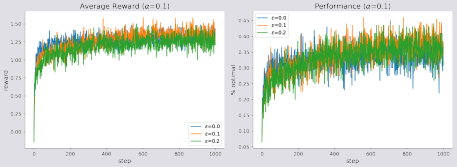
Learning to Act: Multi-Armed Bandits

Changing Epsilon and Alpha

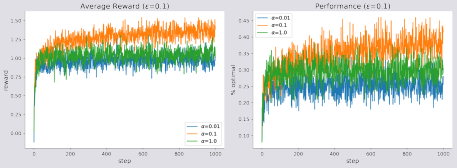
By varying the values of ϵ (exploitation-exploration trade-off), α (learning rate), and even the number of actions k, changes the behavior of our agent.



While we can see how changing the ϵ and α values impact the agent's behavior, this doesn't give us a great sense of which combination is optimal. Due to the stochastic nature of both our rewards and our policy, a single trial run isn't sufficient to give us this information. Let's run multiple trials and compare the average performance.
First we will look at different values for $\epsilon \in [0.0, 0.1, 0.2]$ to a fixed $\alpha = 0.1$. We will run 200 trials as a nice balance between speed and accuracy.



On the left we have plotted the average reward over time, and we see that while $\epsilon = 0$ (the greedy policy) does well initially, $\epsilon = 0.1$ starts to do slightly better in the long run, while $\epsilon = 0.2$ does the worst. Looking on the right, we see the percentage of times the optimal action (the best possible choice at time t) was taken, and here again we see a similar pattern of $\epsilon = 0.1$ starting out a bit slower but eventually having a slight edge in the longer run.
We can also do the same for the learning rates. We will evaluate $\alpha \in [0.01, 0.1, 1.0]$ to a fixed $\epsilon = 0.1$.



Again we see a balance between an effective learning rate. $\alpha = 0.01$ is too weak to quickly incorporate good values, while $\alpha = 1$ is too strong likely resulting in high variance in values due to the Gaussian nature of the rewards.

Learning to Act: Q-Learning

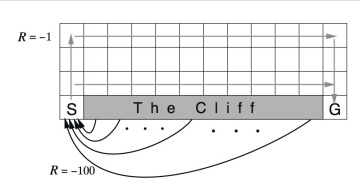
Overview

Here you will learn how to act in the more realistic setting of sequential decisions, formalized by Markov Decision Processes (MDPs). In a sequential decision problem, the actions executed in one state not only may lead to immediate rewards (as in a bandit problem), but may also affect the states experienced next (unlike a bandit problem). Each individual action may therefore affect all future rewards. Thus, making decisions in this setting requires considering each action in terms of their expected **cumulative** future reward.
We will consider here the example of spatial navigation, where actions (movements) in one state (location) affect the states experienced next, and an agent might need to execute a whole sequence of actions before a reward is obtained.

Markov Decision Processes

Grid Worlds

As pointed out, bandits only have a single state and immediate rewards for our actions. Many problems we are interested in have multiple states and delayed rewards, i.e. we won't know if the choices we made will pay off over time, or which actions we took contributed to the outcomes we observed.
In order to explore these ideas, we turn to the common problem setting: the grid world. Grid worlds are simple environments where each state corresponds to a tile on a 2D grid, and the only actions the agent can take are to move up, down, left, or right across the grid tiles. The agent's job is almost always to find a way to a goal tile in the most direct way possible while overcoming some maze or other obstacles, either static or dynamic.
For our discussion we will be looking at the classic Cliff World, or Cliff Walker, environment. This is a 4×10 grid with a starting position in the lower-left and the goal position in the lower-right. Every tile between these two is the "cliff", and should the agent enter the cliff, they will receive a -100 reward and be sent back to the starting position. Every tile other than the cliff produces a -1 reward when entered. The goal tile ends the episode after taking any action from it.



Given these conditions, the maximum achievable reward is -11 (1 up, 9 right, 1 down). Using negative rewards is a common technique to encourage the agent to move and seek out the goal state as fast as possible.

Learning to Act: Q-Learning

Q-Learning

Now that we have our environment, how can we solve it? One of the most famous algorithms for estimating action values (aka Q-values) is the Temporal Differences (TD) control algorithm known as Q-learning (Watkins, 1989).

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big(r_t + \gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\big)$$

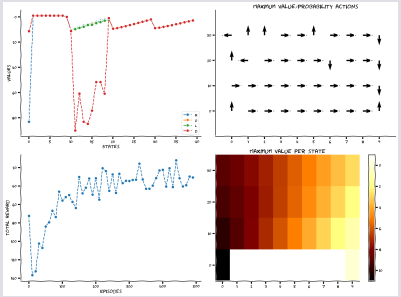
where $Q(s, a)$ is the value function for action a at state s, α is the learning rate, r is the reward, and γ is the temporal discount rate.
The expression $r_t + \gamma \max_a Q(s_{t+1}, a_{t+1})$ is referred to as the TD target while the full expression

$$r_t + \gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t), \tag{150}$$

i.e., the difference between the TD target and the current Q-value, is referred to as the TD error, or reward prediction error.
Because of the max operator used to select the optimal Q-value in the TD target, Q-learning directly estimates the optimal action value, i.e., the cumulative future reward that would be obtained if the agent behaved optimally, regardless of the policy currently followed by the agent. For this reason, Q-learning is referred to as an **off-policy** method.

Implement the Q-learning algorithm

The top left is a representation of the Q-table itself, showing the values for different actions in different states. Notably, going right from the starting state or down when above the cliff is clearly very bad.
The top right figure shows the greedy policy based on the Q-table, i.e. what action would the agent take if it only took its best guess in that state.
The bottom right is the same as the top, only instead of showing the action, it's showing a representation of the maximum Q-value at a particular state.
The bottom left is the actual proof of learning, as we see the total reward steadily increasing after each episode until asymptoting at the maximum possible reward of -11.



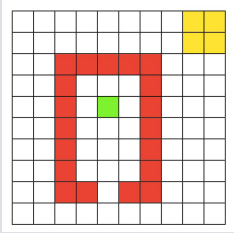
From Reinforcement Learning to Planning

Model-based RL

The algorithms introduced in the previous tutorials are all "model-free", as they do not require a model to use or control behavior. In this section, we will study a different class of algorithms called model-based. As we will see next, in contrast to model-free RL, model-based methods use a model to build a policy.

But what is a model? A model (sometimes called a world model or internal model) is a representation of how the world will respond to the agent's actions. You can think of it as a representation of how the world *works*. With such a representation, the agent can simulate new experiences and learn from these simulations. This is advantageous for two reasons. First, acting in the real world can be costly and sometimes even dangerous. Learning from simulated experience can avoid some of these costs or risks. Second, simulations make fuller use of one's limited experience. To see why, imagine an agent interacting with the real world. The information acquired with each individual action can only be assimilated at the moment of the interaction. In contrast, the experiences simulated from a model can be simulated multiple times – and whenever desired – allowing for the information to be more fully assimilated.

Our RL agent will act in the Quentin's world, a 10x10 grid world.



In this environment, there are 100 states and 4 possible actions: right, up, left, and down. The goal of the agent is to move, via a series of steps, from the start (green) location to the goal (yellow) region, while avoiding the red walls. More specifically:

- The agent starts in the green state,
- Moving into one of the red states incurs a reward of -1,
- Moving into the world borders stays in the same place,
- Moving into the goal state (yellow square in the upper right corner) gives you a reward of 1, and
- Moving anywhere from the goal state ends the episode.

Now that we have our environment and task defined, how can we solve this using a model-based RL agent?

From Reinforcement Learning to Planning

Dyna-Q

In this section, we will implement Dyna-Q, one of the simplest model-based reinforcement learning algorithms. A Dyna-Q agent combines acting, learning, and planning. The first two components – acting and learning – are just like what we have studied previously. Q-learning, for example, learns by acting in the world, and therefore combines acting and learning. But a Dyna-Q agent also implements planning, or simulating experiences from a model– and learns from them.

In theory, one can think of a Dyna-Q agent as implementing acting, learning, and planning simultaneously, at all times. But, in practice, one needs to specify the algorithm as a sequence of steps. The most common way in which the Dyna-Q agent is implemented is by adding a planning routine to a Q-learning agent: after the agent acts in the real world and learns from the observed experience, the agent is allowed a series of k "planning steps". At each one of those k planning steps, the model generates a simulated experience by randomly sampling from the history of all previously experienced state-action pairs. The agent then learns from this simulated experience, again using the same Q-learning rule that you implemented for learning from real experience. This simulated experience is simply a one-step transition, i.e., a state, an action, and the resulting state and reward. So, in practice, a Dyna-Q agent learns (via Q-learning) from one step of **real** experience during acting, and then from k steps of **simulated** experience during planning.

There's one final detail about this algorithm: where does the simulated experiences come from or, in other words, what is the "model"? In Dyna-Q, as the agent interacts with the environment, the agent also learns the model. For simplicity, Dyna-Q implements model-learning in an almost trivial way, as simply caching the results of each transition. Thus, after each one-step transition in the environment, the agent saves the results of this transition in a big matrix, and consults that matrix during each of the planning steps. Obviously, this model-learning strategy only makes sense if the world is deterministic (so that each state-action pair always leads to the same state and reward), and this is the setting of the exercise below. However, even this simple setting can already highlight one of Dyna-Q major strengths: the fact that the planning is done at the same time as the agent interacts with the environment, which means that new information gained from the interaction may change the model and thereby interact with planning in potentially interesting ways.

From Reinforcement Learning to Planning

Dyna-Q Algorithm

Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in S$ and $a \in A$.
Loop forever:

(a) $S \leftarrow$ current (nonterminal) state

(b) $A \leftarrow \epsilon$-greedy(S, Q)

(c) Take action A; observe resultant reward, R, and state, S'

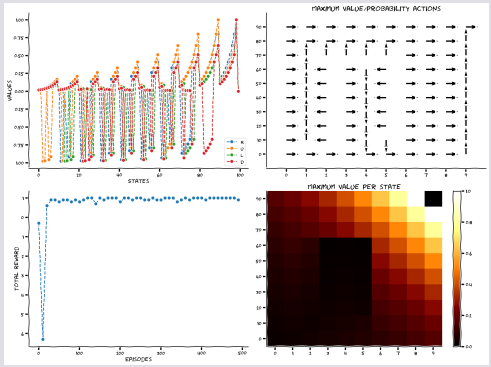
(d) $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$

(e) $Model(S, A) \leftarrow R, S'$ (assuming deterministic environment)

(f) Loop repeat k times: $S \leftarrow$ random previously observed state
$A \leftarrow$ random action previously taken in S
$R, S' \leftarrow Model(S, A)$
$Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$

Implementing Dyna-Q

The plot below show an implementation of Dyna-Q planning to try and solve Quentin's World. Notice that we set the number of planning steps $k = 10$.

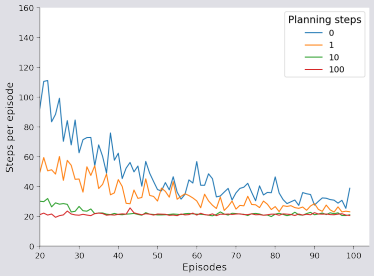


The Dyna-Q agent is able to solve the task quite quickly, achieving a consistent positive reward after only a limited number of episodes (bottom left).

From Reinforcement Learning to Planning

How much to plan?

We implemented a Dyna-Q agent with $k = 10$, we will try to understand the effect of planning on performance. How does changing the value of k impact our agent's ability to learn?
The following code is similar to what we just ran, only this time we run several experiments over several different values of k to see how their average performance compares. In particular, we will choose $k \in \{0, 1, 10, 100\}$. Pay special attention to the case where $k = 0$ which corresponds to no planning. This is, in effect, just regular Q-learning.
The following code will take a bit of time to complete. To speed things up, try lowering the number of experiments or the number of k values to compare.

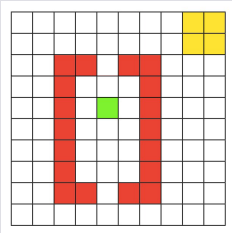


After an initial warm-up phase of the first 20 episodes, we should see that the number of planning steps has a noticeable impact on our agent's ability to rapidly solve the environment. We should also notice that after a certain value of k our relative utility goes down, so it's important to balance a large enough value of k that helps us learn quickly without wasting too much time in planning.

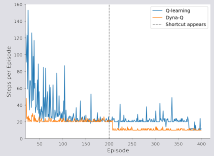
From Reinforcement Learning to Planning

When the world changes...

In addition to speeding up learning about a new environment, planning can also help the agent to quickly incorporate new information about the environment into its policy. Thus, if the environment changes (e.g. the rules governing the transitions between states, or the rewards associated with each state/action), the agent doesn't need to experience that change "repeatedly" (as would be required in a Q-learning agent) in real experience. Instead, planning allows that change to be incorporated quickly into the agent's policy, without the need to experience the change more than once.
In this final section, we will again have our agents attempt to solve Quentin's World. However, after 200 episodes, a shortcut will appear in the environment. We will test how a model-free agent using Q-learning and a Dyna-Q agent adapt to this change in the environment.



The following code again looks similar to what we've run previously. Just as above we will have multiple values for k, with $k = 0$ representing our Q-learning agent and $k = 10$ for our Dyna-Q agent with 10 planning steps. The main difference is we now add in an indicator as to when the shortcut appears. In particular, we will run the agents for 400 episodes, with the shortcut appearing in the middle after episode 200.
When this shortcut appears we will also let each agent experience this change once i.e. we will evaluate the act of moving upwards when in the state that is below the now-open shortcut. After this single demonstration, the agents will continue on interacting in the environment.



If all went well, we should see the Dyna-Q agent having already achieved near optimal performance before the appearance of the shortcut and then immediately incorporating this new information to further improve. In this case, the Q-learning agent takes much longer to fully incorporate the new shortcut.

From Reinforcement Learning to Planning

Summary

In this notebook, you have learned about model-based reinforcement learning and implemented one of the simplest architectures of this type, Dyna-Q. Dyna-Q is very much like Q-learning, but instead of learning only from real experience, you also learn from **simulated** experience. This small difference, however, can have huge benefits! Planning "frees" the agent from the limitation of its own environment, and this in turn allows the agent to speed-up learning – for instance, effectively incorporating environmental changes into one's policy.
Not surprisingly, model-based RL is an active area of research in machine learning. Some of the exciting topics in the frontier of the field involve (i) learning and representing a complex world model (i.e., beyond the tabular and deterministic case above), and (ii) what to simulate – also known as search control – (i.e., beyond the random selection of experiences implemented above).
The framework above has also been used in neuroscience to explain various phenomena such as planning, memory sampling, memory consolidation, and even dreaming!

Neuromatch Academy: Network Causality - Summary Sheet[17]

Interventions

Overview

Here we will describe causality, the tools we use to ask if and how a variable influences other variables. Causal questions are everywhere in neuroscience. How do neurons influence one another? How does a drug affect neurons? How does a stimulus affect behavior? We will talk about how we can answer questions of a causal kind. Causal questions are important all across neuroscience. For example, model fitting, machine learning, and dimensionality reduction, are often used to argue for or against causal models. For example, a regression may be used to argue that a brain region influences another brain region based on fMRI data. Today's materials give us a better understanding of the problems that come with the approach. There are tight links between causality and Bayesian statistics where Bayesian techniques are used for the estimation of causality (see e.g. the work of Judea Pearl). Causality is often seen as the bedrock of science, today's materials above all produce clarity about what it is.
Causality approaches are central across neuroscience. When we run experiments, we often randomly assign them to treatment groups vs control. Alternatively we stimulate animals at random points of time. These methods are all versions of randomized perturbations and probably constitute a good part of all of neuroscience. We also use model fitting frequently to drive arguments about how brains work. This is common for spike data, EEG data, imaging data etc. Lastly, we should be able to sometimes use instrumental variable techniques to estimate the effects of e.g. treatments with drugs. These materials are simultaneously at the heart of the field and are frequently ignored.

Interventions

Introduction

How do we know if a relationship is causal? What does that mean? And how can we estimate causal relationships within neural data?

The methods we'll learn today are very general and can be applied to all sorts of data, and in many circumstances. Causal questions are everywhere!

Defining and estimating causality

Let's think carefully about the statement "**A causes B**". To be concrete, let's take two neurons. What does it mean to say that neuron A causes neuron B to fire?
The *interventional* definition of causality says that:

$$(A \text{ causes } B) \quad \Leftrightarrow \quad (\text{If we force } A \text{ to be different, then } B \text{ changes})$$

To determine if A causes B to fire, we can inject current into neuron A and see what happens to B.

A mathematical definition of causality:
Over many trials, the average causal effect $\delta_{A\to B}$ of neuron A upon neuron B is the average change in neuron B's activity when we set $A = 1$ versus when we set $A = 0$.

$$\delta_{A\to B} = E[B|A = 1] - E[B|A = 0]$$

where $E[B|A = 1]$ is the expected value of B if A is 1 and $E[B|A = 0]$ is the expected value of B if A is 0.
Note that this is an average effect. While one can get more sophisticated about conditional effects (A only effects B when it's not refractory, perhaps), we will only consider average effects today.
Relation to a randomized controlled trial (RCT): The logic we just described is the logic of a randomized control trial (RCT). If you randomly give 100 people a drug and 100 people a placebo, the effect is the difference in outcomes.

Randomized controlled trial for two neurons

Let's pretend we can perform a randomized controlled trial for two neurons. Our model will have neuron A synapsing on Neuron B:

$$B = A + \epsilon$$

where A and B represent the activities of the two neurons and ϵ is standard normal noise $\epsilon \sim \mathcal{N}(0, 1)$.
To confirm if A is perturbed that B changes.
We can calculate the a difference in means of '0.990719' (so very close to one)

Interventions

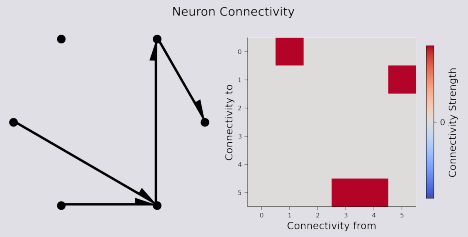
Simulating a system of neurons

Can we still estimate causal effects when the neurons are in big networks? This is the main question we will ask today. Let's first create our system, and the rest of today we will spend analyzing it. Here we introduce a big causal system (interacting neurons) with understandable dynamical properties and how to simulate it.
Our system has N interconnected neurons that affect each other over time. Each neuron at time $t + 1$ is a function of the activity of the other neurons from the previous time t.
Neurons affect each other nonlinearly: each neuron's activity at time $t + 1$ consists of a linearly weighted sum of all neural activities at time t, with added noise, passed through a nonlinearity:

$$\vec{x}_{t+1} = \sigma(A\vec{x}_t + \epsilon_t),$$

- $\vec{x}_t$ is an n-dimensional vector representing our n-neuron system at timestep t
- σ is a sigmoid nonlinearity
- A is our $n \times n$ "causal ground truth connectivity matrix" (more on this later)
- ϵ_t is random noise: $\epsilon_t \sim N(\vec{0}, I_n)$
- $\vec{x}_0$ is initialized to $\vec{0}$

A is a connectivity matrix, so the element A_{ij} represents the causal effect of neuron i on neuron j. In our system, neurons will receive connections from only 10% of the whole population on average.
We will create the true connectivity matrix between 6 neurons and visualize it in two different ways: as a graph with directional edges between connected neurons and as an image of the connectivity matrix.
Check your understanding: do you understand how the left plot relates to the right plot below?



[17] 't Hart et al., (2022). Neuromatch Academy: a 3-week, online summer school in computational neuroscience. Journal of Open Source Education, 5(49), 118. https://doi.org/10.21105/jose.00118

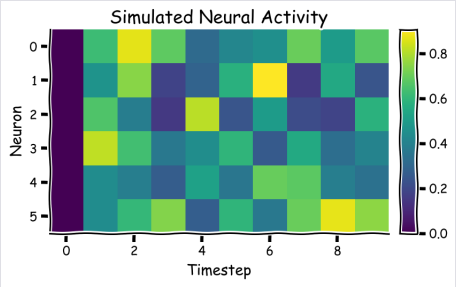
Interventions

System simulation

To simulation a system of six neurons we using a sigmoid σ function so that at every timestep the activity vector x is updated according to:

$$\vec{x}_{t+1} = \sigma(A\vec{x}_t + \epsilon_t).$$

giving the plot:



Random perturbation in our system of neurons

We want to get the causal effect of each neuron upon each other neuron. The ground truth of the causal effects is the connectivity matrix A.
Remember that we would like to calculate:

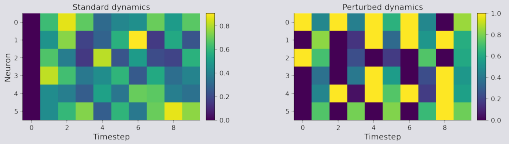
$$\delta_{A \to B} \quad = \quad E[B|A=1] - E[B|A=0]$$

We'll do this by randomly setting the system state to 0 or 1 and observing the outcome after one timestep. If we do this N times, the effect of neuron i upon neuron j is:

$$\delta_{x^i \to x^j} \quad \approx \quad \frac{1}{N} \sum_{t=0,\ t \text{ even}}^{N} [x^j_{t+1} | x^i_t = 1]$$

$$-\frac{1}{N} \sum_{t=0,\ t \text{ even}}^{N} [x^j_{t+1} | x^i_t = 0]$$

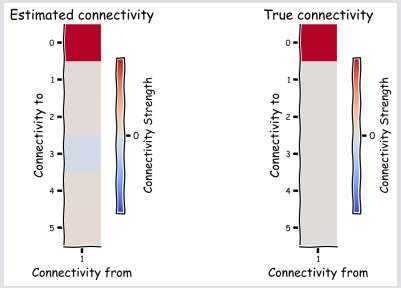
This is just the average difference of the activity of neuron j in the two conditions.
We are going to calculate the above equation, but imagine it like *intervening* in activity every other timestep.



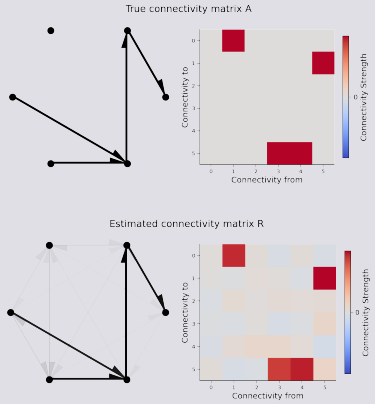
Interventions

Recovering connectivity from perturbed dynamics

Recall that we perturbed every neuron at every other timestep. Despite perturbing every neuron, in this exercise we are concentrating on computing the causal effect of a single neuron (we will look at all neurons effects on all neurons next). We want to exclusively use the timesteps without perturbation for x^j_{t+1} and the timesteps with perturbation for x^j_t in the formulas above.



We can quantify how close our estimated connectivity matrix is to our true connectivity matrix by correlating them. We should see almost perfect correlation between our estimates and the true connectivity - do we?
Note on interpreting A: Strictly speaking, A is not the matrix of causal effects but rather the dynamics matrix. So why compare them like this? The answer is that A and the effect matrix both are 0 everywhere except where there is a directed connection. So they should have a correlation of 1 if we estimate the effects correctly. (Their scales, however, are different. This is in part because the nonlinearity σ squashes the values of x to $[0, 1]$.)



We can again calculate the correlation coefficient between the elements of the two matrices. If it is almost 1 we have done a good job recovering the true causality of the system!

Correlations

Recovering connectivity from perturbed dynamics

Here, we implemented and explored the dynamical system of neurons we will be working with throughout all of the tutorials today. We also learned about the "gold standard" of measuring causal effects through random perturbations. As random perturbations are often not possible, we will now turn to alternative methods to attempt to measure causality. We will:

- Learn how to estimate connectivity from observations assuming **correlations approximate causation**
- Show that this only works when the network is small

Often, we can't force neural activities or brain areas to be on or off. We just have to observe. Maybe we can get the correlation between two nodes – is that good enough? The question we ask here is **when is correlation a "good enough" substitute for causation?**

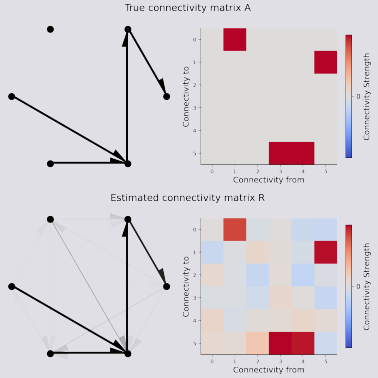
The answer is not "never", actually, but "sometimes".

Try to approximate causation with correlation

In small systems, correlation can look like causation. Let's attempt to recover the true connectivity matrix (A) just by correlating the neural state at each timestep with the previous state:

$$C = \vec{x_t} \vec{x_{t+1}}^{\top}.$$

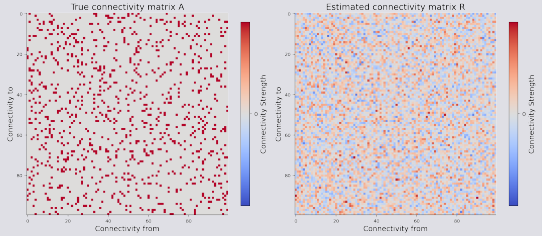
To calculate the connectivity matrix of a single neuron by calculating the correlation coefficients with every other neuron which correlate two vectors: 1) the activity of a selected neuron at time t 2) The activity of all other neurons at time $t+1$.
The matrix answer is the same as the summation form. Furthermore the estimated vs true connectivity look the same using the matrix calculation.



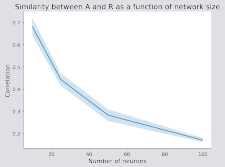
Correlations

Large systems

As our system becomes more complex however, correlation fails to capture causality. Let's jump to a much bigger system. Instead of 6 neurons, we will now use 100 neurons. How does the estimation quality of the connectivity matrix change?

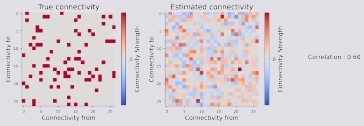


Correlation as a function of network size



Connectivity estimation as a function of the sparsity of A

You may rightly wonder if correlation only fails for large systems for certain types of A. Does connectivity estimation get better or worse with less sparsity?



Summary

Now for the takeaway. We know that for large systems correlation $\neq$ causation. But what about when we coarsely sample the large system? Do we get better at estimating the effective causal interaction between groups (=average of weights) from the correlation between the groups?
From our simulation above, the answer appears to be no: as the number of neurons per group increases, we don't see any significant increase in our ability to estimate the causal interaction between groups.

Simultaneous fitting/regression

Objectives

We have explored correlation as an approximation for causation and learned that correlation $\neq$ causation for larger networks. However, computing correlations is a rather simple approach, and you may be wondering: will more sophisticated techniques allow us to better estimate causality? Can't we control things?
Here we'll use some common advanced (but controversial) methods that estimate causality from observational data. These methods rely on fitting a function to our data directly, instead of trying to use perturbations or correlations. Since we have the full closed-form equation of our system, we can try these methods and see how well they work in estimating causal connectivity when there are no perturbations. Specifically, we will:

- Learn about more advanced (but also controversial) techniques for estimating causality
- conditional probabilities (**regression**)
- Explore limitations and failure modes
- understand the problem of **omitted variable bias**

Regression approach

You may be familiar with the idea that correlation only implies causation when there are no hidden *confounders*. This aligns with our intuition that correlation only implies causality when no alternative variables could explain away a correlation.
A confounding example:
Suppose you observe that people who sleep more do better in school. It's a nice correlation. But what else could explain it? Maybe people who sleep more are richer, don't work a second job, and have time to actually do homework. If you want to ask if sleep *causes* better grades, and want to answer that with correlations, you have to control for all possible confounds.

A confound is any variable that affects both the outcome and your original covariate. In our example, confounds are things that affect both sleep and grades.
Controlling for a confound: Confounds can be controlled for by adding them as covariates in a regression. But for your coefficients to be causal effects, you need three things:

1. All confounds are included as covariates
2. Your regression assumes the same mathematical form of how covariates relate to outcomes (linear, GLM, etc.)
3. No covariates are caused "by" both the treatment (original variable) and the outcome. These are colliders; we won't introduce it today (but Google it on your own time! Colliders are very counterintuitive.)

In the real world it is very hard to guarantee these conditions are met. In the brain it's even harder (as we can't measure all neurons). Luckily today we simulated the system ourselves.

Simultaneous fitting/regression

Fitting a General Linear Model (GLM)

We will use a regression approach to estimate the causal influence of all neurons to neuron 1. Specifically, we will use linear regression to determine the A in:

$$\sigma^{-1}(\vec{x}_{t+1}) = A\vec{x}_t + \epsilon_t, \tag{151}$$

where σ^{-1} is the inverse sigmoid transformation, also sometimes referred to as the **logit** transformation: $\sigma^{-1}(x) = \log(\frac{x}{1-x})$.
Let W be the $\vec{x}_t$ values, up to the second-to-last timestep $T-1$:

$$W = \begin{bmatrix} | & | & \dots & | \\ \vec{x}_0 & \vec{x}_1 & \dots & \vec{x}_{T-1} \\ | & | & \dots & | \end{bmatrix}_{n\times(T-1)} \tag{152}$$

Let Y be the $\vec{x}_{t+1}$ values for a selected neuron, indexed by i, starting from the second timestep up to the last timestep T:

$$Y = \begin{bmatrix} x_{i,1} & x_{i,2} & \dots & x_{i,T} \end{bmatrix}_{1\times(T-1)} \tag{153}$$

You then fit the following model:

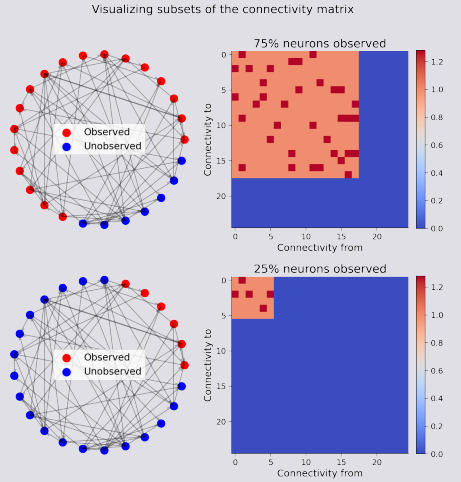
$$\sigma^{-1}(Y^T) = W^T V \tag{154}$$

where V is the $n \times 1$ coefficient matrix of this regression, which will be the estimated connectivity matrix between the selected neuron and the rest of the neurons.
We can see that multiple regression is better than simple correlation for estimating connectivity.

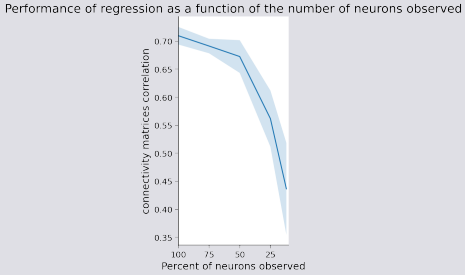
Simultaneous fitting/regression

Partially Observed Systems

If we are unable to observe the entire system, **omitted variable bias** becomes a problem. If we don't have access to all the neurons, and so therefore can't control them, can we still estimate the causal effect accurately?
We first visualize different subsets of the connectivity matrix when we observe 75% of the neurons vs 25%.
Recall the meaning of entries in our connectivity matrix: $A[i,j] = 1$ means a connectivity from neuron i to neuron j with strength 1.



Next, we will inspect a plot of the correlation between true and estimated connectivity matrices vs the percent of neurons observed over multiple trials. What is the relationship that you see between performance and the number of neurons observed?



Instrumental Variables

Objectives

We have seen that even more sophisticated techniques such as simultaneous fitting fail to capture causality in the presence of omitted variable bias. So what techniques are there for us to obtain valid causal measurements when we can't perturb the system? Here we will:

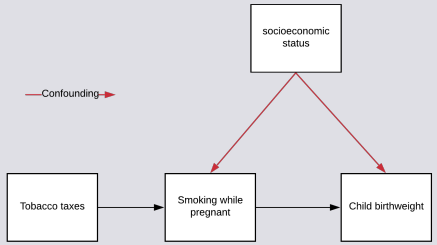
- learn about **instrumental variables**, a method that does not require experimental data for valid causal analysis
- explore benefits of instrumental variable analysis and limitations
- addresses **omitted variable bias** seen in regression
- less efficient in terms of sample size than other techniques
- requires a particular form of randomness in the system in order for causal effects to be identified

Instrumental Variables

If there is randomness naturally occurring in the system *that we can observe*, this in effect becomes the perturbations we can use to recover causal effects. This is called an **instrumental variable**. At high level, an instrumental variable must

1. Be observable
2. Affect a covariate you care about
3. **Not** affect the outcome, except through the covariate

It's rare to find these things in the wild, but when you do it's very powerful.



Instrumental Variables

A non-neuro example of an Instrumental Variables (IV)

A classic example is estimating the effect of smoking cigarettes while pregnant on the birth weight of the infant. There is a (negative) correlation, but is it causal? Unfortunately many confounds affect both birth weight and smoking. Wealth is a big one. Instead of controlling everything imaginable, one can find an IV. Here the instrumental variable is **state taxes on tobacco**. These

1. Are observable
2. Affect tobacco consumption
3. Don't affect birth weight except through tobacco

By using the power of IV techniques, you can determine the causal effect without exhaustively controlling for everything.
Let's represent our tobacco example above with the following notation:

- Z_{taxes}: our tobacco tax *instrument*, which only affects an individual's tendency to smoke while pregnant within our system
- T_{smoking}: number of cigarettes smoked per day while pregnant, our "treatment" if this were a randomized trial
- C_{SES}: socioeconomic status (higher means wealthier), a *confounder* if it is not observed
- $Y_{\text{birthweight}}$: child birthweight in grams, our outcome of interest

Let's suppose we have the following function for our system:

$$Y_{\text{birthweight}} = 3000 + C_{\text{SES}} - 2T_{\text{smoking}},$$

with the additional fact that C_{SES} is negatively correlated with T_{smoking}.
The causal effect we wish to estimate is the coefficient -2 for T_{smoking}, which means that if a mother smokes one additional cigarette per day while pregnant her baby will be 2 grams lighter at birth. We've provided a covariance matrix with the desired structure in the code cell below, so please run it to look at the correlations between our variables.
Correlation between SES status and cigarettes: -0.483
Correlation between SES status and birth weight: 0.740
We see what is exactly represented in our graph above: C_{SES} is correlated with both T_{smoking} and $Y_{\text{birthweight}}$, so C_{SES} is a potential confounder if not included in our analysis.
Let's say that it is difficult to observe and quantify C_{SES}, so we do not have it available to regress against. This is another example of the **omitted variable bias**.
What about Z_{taxes}?
Correlation between taxes and cigarettes: 0.519
Correlation between taxes and SES status: 0.009
Perfect! We see that Z_{taxes} is correlated with T_{smoking} (2) but is uncorrelated with C_{SES} (3). Z_{taxes} is also observable (1), so we've satisfied our three criteria for an instrument:

1. Z_{taxes} is observable
2. Z_{taxes} affects T_{smoking}
3. Z_{taxes} doesn't affect

$Y_{\text{birthweight}}$ except through T_{smoking} (ie Z_{taxes} doesn't affect or is affected by C_{SES})

Instrumental Variables

How IV works

The easiest way to imagine IV is that the instrument is **an observable source of "randomness"** that affects the treatment. In this way it's similar to the interventions we talked about in Tutorial 1.
But how do you actually use the instrument? The key is that we need to extract **the component of the treatment that is due only to the effect of the instrument**. We will call this component $\hat{T}$.

$$\hat{T} \leftarrow \text{The unconfounded component of } T$$

Getting $\hat{T}$ is fairly simple. It is simply the predicted value of T found in a regression that has only the instrument Z as input.
Once we have the unconfounded component in hand, getting the causal effect is as easy as regressing the outcome on $\hat{T}$.

IV estimation using two-stage least squares

The fundamental technique for instrumental variable estimation is **two-stage least squares**.
We run two regressions:

1. The first stage gets $\hat{T}_{\text{smoking}}$ by regressing T_{smoking} on Z_{taxes}, fitting the parameter $\hat{\alpha}$:

$$\hat{T}_{\text{smoking}} = \hat{\alpha} Z_{\text{taxes}} \tag{155}$$

2. The second stage then regresses $Y_{\text{birthweight}}$ on $\hat{T}_{\text{smoking}}$ to obtain an estimate $\hat{\beta}$ of the causal effect:

$$\hat{Y}_{\text{birthweight}} = \hat{\beta} \hat{T}_{\text{smoking}} \tag{156}$$

The first stage estimates the **unconfounded component** of T_{smoking} (ie, unaffected by the confounder C_{SES}), as we discussed above.
Then, the second stage uses this unconfounded component $\hat{T}_{\text{smoking}}$ to estimate the effect of smoking on $\hat{Y}_{\text{birthweight}}$.

Compute regression stage 1
Let's run the regression of T_{smoking} on Z_{taxes} to compute $\hat{T}_{\text{smoking}}$. We will then check whether our estimate is still confounded with C_{SES} by comparing the correlation of C_{SES} with T_{smoking} vs $\hat{T}_{\text{smoking}}$.
The result of the correlation between T and C of '-0.483' and between $\hat{T}$ and C of '0.009'.

Least squares regression stage 2
Now let's implement the second stage! We will again use a linear regression model with an intercept. We will obtain the estimated causal effect of the number of cigarettes (T) on birth weight (Y).

The result of estimated causal effect of '-1.984', This is quite close to the true causal effect of -2!

Instrumental Variables

IVs in our simulated neural system

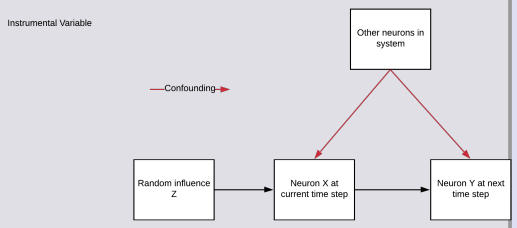
Now, say we have the neural system we have been simulating, except with an additional variable $\vec{z}$. This will be our instrumental variable.
We treat $\vec{z}$ as a source of noise in the dynamics of our neurons:

$$\vec{x}_{t+1} = \sigma(A\vec{x}_t + \eta\vec{z}_{t+1} + \epsilon_t) \tag{157}$$

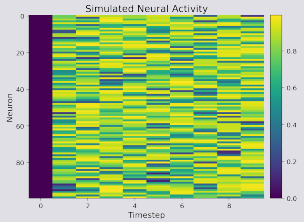
where η is what we'll call the "strength" of our IV, and $\vec{z}_t$ is a random binary variable, $\vec{z}_t \sim Bernoulli(0.5)$
Remember that for each neuron i, we are trying to figure out whether i is connected to (causally affects) the other neurons in our system *at the next time step*. So for timestep t, we want to determine whether $\vec{x}_{i,t}$ affects all the other neurons at $\vec{x}_{t+1}$. For a given neuron i, $\vec{z}_{i,t}$ satisfies the 3 criteria for a valid instrument.

What could z be, biologically?

Imagine z to be some injected current through an *in vivo* patch clamp. It affects each neuron individually, and only affects dynamics through that neuron.
The cool thing about IV is that you don't have to control z yourself - it can be observed. So if you mess up your wiring and accidentally connect the injected voltage to an AM radio, no worries. As long as you can observe the signal the method will work.



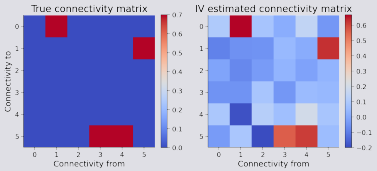
Simulate a system with IV



Instrumental Variables

Estimate IV for simulated neural system

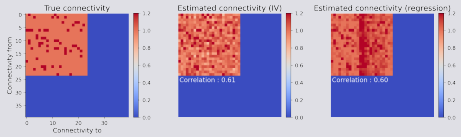
Since you just implemented two-stage least squares, let's see how our IV estimates do in recovering the connectivity matrix.



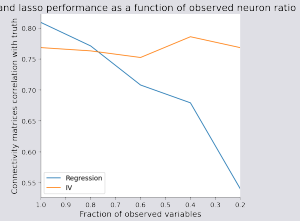
The IV estimates seem to perform pretty well! In the next section, we will see how they behave in the face of omitted variable bias.

IVs and omitted variable bias

Changing the ratio of observed neurons and look at the impact on the quality of connectivity estimation using IV vs regression. The plots below are for a ratio of 0.6.



We can also visualize the performance of regression and IV as a function of the observed neuron ratio below.



We see that IVs handle omitted variable bias (when the instrument is strong and we have enough data).
The costs of IV analysis

- we need to find an appropriate and valid instrument
- Because of the 2-stage estimation process, we need strong instruments or else our standard errors will be large