MODELING NEURAL CIRCUITS MADE SIMPLE

with Python

ROBERT ROSENBAUM

2022

All models are wrong, but some models are useful.

– George Box

CONTENTS

INTRODUCTION

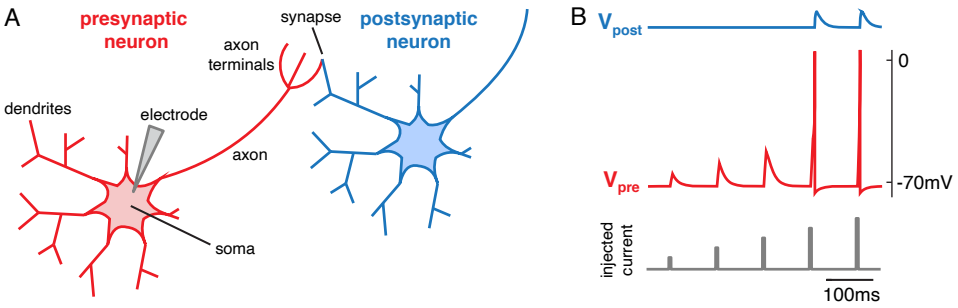

Figure 1: **A)** A diagram of two neurons. An electrode (gray) injects electrical current into the presynaptic neuron (red), which connects to the postsynaptic neuron (blue) at a synapse. **B)** The membrane potentials (V_{pre} and V_{post}) of the two neurons in response to current pulses injected into the presynaptic neuron. Sufficiently strong injected current evokes action potentials or "spikes" in the presynaptic neuron's membrane potential, which then evoke responses in the postsynaptic neuron's membrane. This book describes mathematical models of the dynamics sketched in this figure, and of networks of interconnected neurons.

Neurons are arguably the most important type of cell in the nervous system. In this book, we will develop mathematical and computational models of neurons and networks of neurons, focusing primarily on neurons in the **cerebral cortex**, which is widely viewed as the central processing area of mammal's brains. There is a huge diversity of neuron types with different properties, but the prototypical cortical neuron is in the ballpark of 10μm (micrometers) in size and composed of three parts (Figure 1A): the soma, axon, and dendrites. **Dendrites** are tree-like structures on which neurons receive input from other neurons at connections called **synapses**. The **soma** is the cell body where these inputs are integrated. The neuron's response to its inputs propagates down the **axon** where it can be communicated to other neurons.

Neurons maintain a negatively charged electrical potential across their membrane, meaning that the ratio of negatively to positively charged ions is greater inside the cell than outside the cell. Specifically, the potential across the neuron's membrane is usually somewhere in the ballpark of -70mV (millivolts). This electrical potential is called the neuron's **membrane potential**, which we denote V. The membrane itself is highly resistive, meaning that it is not easily permeated by ions. Instead, the membrane potential is primarily modulated by ions flowing through **ion pumps** and **ion channels** in the membrane. The flow of charged ions through ion channels and pumps creates electrical currents that affect the neuron's membrane potential.

When a neuron's membrane potential reaches a threshold around $V \approx -55$mV, the opening and closing of different ion channels creates an **action potential** or **spike**, which is a deviation of V to around 0-10mV that lasts about 1-2ms (Figure 1B). The sequence of a neuron's action potentials is called its **spike train**.

Spikes propagate down the neuron's axon where they activate synapses. The synapses open ion channels on the postsynaptic neuron's membrane, causing a brief pulse of

current across the postsynaptic neuron's membrane (Figure 1B). Synapses are a primary means of communication between neurons.

Neurons encode information in the timing and frequency of their spikes and they communicate this information to other neurons through synapses. The frequency with which a neuron spikes over a period of time is called the neuron's **firing rate**. To see a nice illustration of how properties of visual stimuli affect a neuron's firing rate, search online for videos of Hubel and Wiesel's recordings from neurons in cat primary visual cortex. In these videos, each little clicking sound is an action potential. You can hear that the recorded neuron's firing rate is elevated by bars of light at a particular angle and particular location in the cat's visual field. The neuron's spike train reflects information about the cat's visual stimulus. This information is communicated to other neurons in the cat's brain and ultimately these neurons piece together the cat's perception of what it sees and how it responds to this perception. Understanding exactly how all of this happens is an enormous unsolved puzzle, one of the greatest frontiers of modern science. One piece of the puzzle is to understand the dynamics that emerge from networks of interconnected neurons. In this book, we will build a basic toolbox of mathematical and computational approaches for tackling this piece of the puzzle.

References to Appendix A appear like this. See Appendix A.1 for a brief introduction to Python.

HOW TO USE THIS BOOK. This book is assumes a background in first-semester calculus (derivatives and integrals), basic linear algebra (matrix products), and first-semester probability or statistics (expectations and variance). All other mathematical background is reviewed in Appendix A. Next to each paragraph that introduces a new mathematical topic, there is reference to the relevant section in Appendix A in red text in the margin. If you need to review that topic, follow the reference before reading on. Otherwise, you can ignore the reference and continue reading.

Figures in the book are accompanied by Python notebooks containing code to reproduce the figure. The file names of each Python notebook is referenced in the associated figure caption. If you are not familiar with Python or NumPy, or if you just need a review, see Appendix A.1 and the Python notebook `PythonIntro.ipynb`. Important equations in the book appear in boxes like this:

A very important equation

$$E = mc^2$$

References to Appendix B of extended models appear like this.

The main text of this book was whittled down to include a minimal thread of material to build a backbone for modeling neural circuits. This approach assures that the book can be covered in a one-semester course and also makes the book more amenable to self-learning. Many important topics and models are omitted from the book, but the book should equip the reader to learn new models and concepts. To this end, Appendix B contains a supplementary treatment of several additional models and topics. References to Appendix B appear as blue text in the margin.

Some sections in this book were inspired by similar sections in the two excellent textbooks, *Theoretical Neuroscience* [1] and *Neuronal Dynamics* [2], which can serve as supplements to this book. For a very nice exposition about the history of computational neuroscience and some of its fundamental concepts, see Grace Lindsay's popular science book, *Models of the Mind* [3].

1 SIMPLIFIED MODELS OF SINGLE NEURONS

1.1 THE LEAKY INTEGRATOR MODEL

Neurons maintain a negative electrical potential across their membrane. We will use the variable V to denote the potential across a neuron's membrane, called the **membrane potential**. A neuron's membrane potential can be modeled as a leaky capacitor which gives rise to the ordinary differential equation (ODE)

See Appendix A.2 for an introduction to ODEs.

$$C_m \frac{dV}{dt} = I$$

where I is the current across the membrane and C_m is the membrane's capacitance. You may have seen this equation before when studying resistor-capacitor (RC) circuits.

Really, I and C_m represent the average current and capacitance *per unit area* of the membrane, but we will ignore that fact and ignore spatial variations in V along the neuron's membrane. A real neuron for which V does not vary much across space is called "electronically compact." A neuron model that ignores the spatial variation is called a "point neuron" model. We only consider point neuron models in this text.

A current that increases the membrane potential ($I > 0$) is called an **inward** or **depolarizing** current. A current that decreases the membrane potential ($I < 0$) is called an **outward** or **hyperpolarizing** current. With this terminology, we are implicitly assuming that the neuron is negatively charged and we are measuring the inside potential versus the outside. Because of the way that currents are recorded in experiments, they are sometimes measured backwards (outside-vs-inside) so that $I > 0$ is outward and $I < 0$ is inward, but we will stick to the "$I > 0 \leftrightarrow$ inward" convention in this book.

The lipid bi-layer making up a neuron's membrane is highly resistive, so ions do not pass through the membrane itself very easily. For the most part, ions pass through the membrane through two mechanisms: ion pumps and ion channels. **Ion pumps** use energy to maintain ion concentration differences across cell membrane. Ion pumps allow neurons to maintain a negative potential.

Ion channels can be thought of as pores that allow selected ion types to pass through membrane. Ions diffuse through on their own based on the concentration gradient and the electrical gradient (unlike pumps where ions are forced through). There are thousands of types of channels in brain and often dozens of types on a single neuron's membrane. Different channels admit different types of ions to pass through and ion channels are driven to open and close based on a variety of different factors. For example, ion channels in your ear are opened and closed mechanically by vibrations of tiny hairs!

For most cortical neurons, the overall effects of many ion channels and pumps are relatively steady when the membrane potential is near rest (V around -70mV). As a result, we can get a reasonable approximation to the membrane current near rest by

lumping together the currents induced by many ion pumps and ion channels into one *effective* current called the **leak current**, which is defined by

$$I_L = -g_L(V - E_L). \tag{1.1}$$

The constant $g_L > 0$ is called the **leak conductance** and approximately quantifies how easily ions pass through the membrane's ion channels. The constant E_L is the **equilibrium** or **resting potential** of the neuron, sometimes also called the **leak reversal potential**. The idea is that different ion pumps and ion channels pull V in different directions with different strengths. The equilibrium potential, E_L, is the value of V at which all of these forces cancel out, so there is no current. When V is away from E_L, then g_L measures how strongly V is pulled back toward E_L. When $V > E_L$, we say that the membrane potential is **depolarized** and when $V < E_L$, we say that it is **hyperpolarized**. A more detailed derivation of Eq. (1.1) and a description of how to model ion channels more generally is given in Appendix B.1.

See Appendix B.1 for a more detailed description of ion current models and a derivation of Eq. (1.1).

In addition to the leak current, I_L, we might also want to model additional sources of current such as the current injected by a scientist's electrode (as in Figure 1A). To this end, we define the membrane current as

$$I = I_L + I_x$$

where I_x is any external source of current that we want to model. We will sometimes refer to I_x as the neuron's **input current**, **external input**, or simply **input**. Putting this together gives:

$$C_m \frac{dV}{dt} = -g_L(V - E_L) + I_x(t)$$

Despite its simplicity, this model can do a decent job of explaining the salient **sub-threshold** or **passive** properties of some neurons, *i.e.* the behavior of $V(t)$ below the spiking threshold and therefore in the absence of spikes. The model can be simplified by setting

$$\tau_m = \frac{C_m}{g_L}$$

and rescaling the input current by taking $I_x \leftarrow I_x/g_L$ to get the version of the model that we will use in this textbook,

The leaky integrator model

$$\tau_m \frac{dV}{dt} = -(V - E_L) + I_x(t). \tag{1.2}$$

Eq. (1.2) defines the **leaky integrator model**. The parameter, τ_m, is called the **membrane time constant** and sets the timescale of the membrane potential dynamics. Typical cortical neurons have membrane time constants around 5-20ms.

Note that $I_x(t)$ is, strictly speaking, not a current in Eq. (1.2) because it has dimensions of electrical potential (same as $V(t)$), typically measured in units mV. This is due to our rescaling by g_L. However, we will still refer to it as a "current" since it is proportional

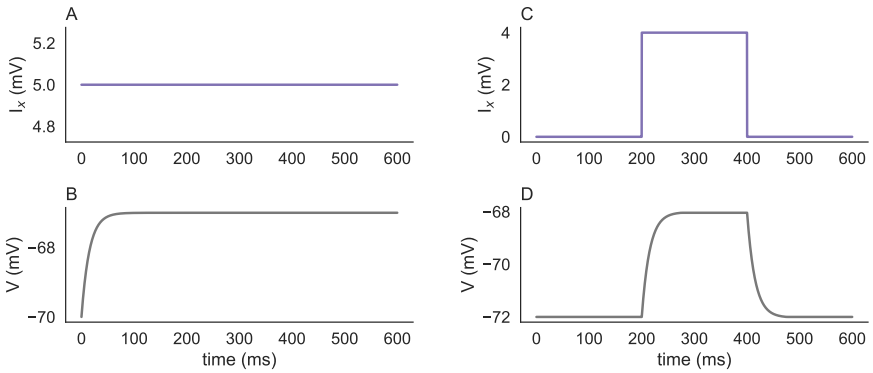

Figure 1.1: Leaky integrator model with time-constant and time-dependent input. The membrane potential, $V(t)$, and input current, I_x, of the leaky integrator model with time-constant input current, $I_x(t) = I_0 = 4$mV (Left) and with a square-wave time-dependent input, $I_x(t)$ (Right). Parameters are $\tau_m = 15$ms, $E_L = -72$, and $V(0) = -70$. See `LeakyIntegrator.ipynb` for code to produce these figures.

to the actual external current and it still models a current. Some people additionally re-parameterize V by taking to $V \leftarrow V + E_L$ and rescale time by taking $t \leftarrow t/\tau_m$ to get

$$\frac{dV}{dt} = -V + I_x(t).$$

However, we will stick to the form in Eq. (1.2).

We next derive and interpret solutions to Eq. (1.2), first for the case of time-constant input, $I_x(t) = I_0$. When $I_x(t) = I_0$, Eq. (1.2) is an autonomous, linear (and also separable) differential equation, which has a solution given by

See Appendix A.3 for a review of ODEs for exponential decay.

$$V(t) = (V_0 - E_L - I_0)\, e^{-t/\tau_m} + E_L + I_0 \tag{1.3}$$

where $V(0) = V_0$ is the initial condition. Eq. (1.3) represents an exponential decay to $E_L + I_0$. The timescale of this decay is set by τ_m. Roughly speaking, τ_m is the amount of time required for the membrane potential to get a little past halfway from $V(0)$ to $E_L + I_0$ (specifically, it gets a proportion $1 - e^{-1} \approx 0.63$ of the way).

In Python, we would implement the solution as

```
V=(V0-EL-I0)*exp(-time/taum)+EL+I0;
```

where `time` is a NumPy array representing discretized time. See Figure 1.1A,B and `LeakyIntegrator.ipynp` for a more complete simulation of the leaky integrator with time-constant input.

We next consider the leaky integrator with a time-dependent input, $I_x(t)$. When there is a time-dependent input current, $I_x(t)$, Eq. (1.2) is an inhomogeneous, linear differential equation. The solution to Eq. (1.2) can be written as a convolution of $I_x(t)$ with a kernel. Specifically, the solution can be written as

See Appendices A.4 and A.5 for a review of convolutions and linear ODEs with time-dependent forcing.

$$V(t) = (V_0 - E_L)\, e^{-t/\tau_m} + E_L + (k * I_x)(t) \tag{1.4}$$

where $V_0 = V(0)$, $*$ denotes convolution, and the kernel is defined by

$$k(s) = \begin{cases} \frac{1}{\tau_m} e^{-s/\tau_m} & t \geq 0 \\ 0 & s < 0 \end{cases}$$
$$= \frac{1}{\tau_m} e^{-s/\tau_m} H(s)$$

Here,

$$H(s) = \begin{cases} 1 & s \geq 0 \\ 0 & s < 0 \end{cases}$$

is the Heaviside step function. We have implicitly assumed that $I_x(t) = 0$ for $t < 0$, *i.e.*, the input starts at $t = 0$.

When $V_0 = E_L$ (or $t \gg \tau_m$), the first term in Eq. (1.4) can be ignored so

$$V(t) = E_L + (k * I_x)(t).$$

In other words, the membrane potential is a filtered version of the input (plus E_L). Since $k(s) = 0$ for $s < 0$, this is a causal filter, meaning that $V(t_0)$ only depends on values of $I_x(t)$ in the past ($t < t_0$). Also, $\int k(t)dt = 1$, so $V(t)$ **is a running, weighted average of the recent history of** $I_x(t)$. The membrane time constant, τ_m, determines how quickly the neuron responds to changes in $I_x(t)$, equivalently how far in the past it averages $I_x(t)$. Another way of thinking about this equation for $V(t)$ is that changes to $I_x(t)$ get integrated into $V(t)$ and then forgotten over a timescale of τ_m.

In Matlab, we would implement the solution as

```
# Define the convolution kernel
s=np.arange(-5*taum,5*taum,dt)
k=(1/taum)*np.exp(-s/taum)*(s>=0);
# Define V by convolution
V=EL+np.convolve(Ix,k,mode='same')*dt
```

Note that we defined the kernel, $k(s)$, over a time-interval $[-5\tau_m, 5\tau_m]$ because the time-window must be centered at $s = 0$ and because $k(s)$ is close to zero outside of this window (the window contains most of the "mass" of $k(s)$). A complete example of the leaky integrator with time-dependent input is given in Figure 1.1C,D and the second code cell of `LeakyIntegrator.ipynb`. Try out different time series for $I_x(t)$ to get an intuition for the model.

The leaky integrator does a reasonable job of describing sub-threshold (non-spiking) membrane dynamics, but does not capture action potentials, which occur when the membrane potential exceeds a threshold around -55mV. Referring back to Figure 1B, the leaky integrator model captures the membrane potential responses to the first three current pulses, but does not capture the action potential responses to the last two pulses. In Appendix B.2, we discuss the Hodgkin-Huxley model that describes how sodium and potassium ion channels produce action potentials. In the next section, we discuss a much simpler model that captures many of the salient features of action potential generation.

1.2 THE EXPONENTIAL INTEGRATE-AND-FIRE (EIF) MODEL

When the membrane potential of a neuron gets close to -55mV, sodium channels begin opening, causing an influx of sodium, which pushes the membrane potential higher (because sodium is positively charged). This causes more sodium channels to open, creating a positive feedback loop and a rapid upswing in the membrane potential. This rapid upswing is eventually shut down by the closing of sodium channels and the opening of potassium channels that pulls the membrane potential back down toward rest. The Hodgkin-Huxley model from Appendix B.2 captures these dynamics in great detail, but it is very complicated and computationally expensive to simulate, so we focus on a simplified model here.

See Appendix B.2 for a description of the Hodgkin Huxley model, which gives more biophysically detailed account of action potentials.

The upswing of action potentials and the effects of sub-threshold sodium currents can be captured by adding an exponential term to the leaky integrator model. The subsequent "reset" of the membrane potential back toward rest can be captured by a simple rule saying that every time $V(t)$ exceeds some threshold, V_{th}, we record a spike and reset $V(t)$ to V_{re}. Putting this together gives the **exponential integrate-and-fire (EIF) model** [4],

The exponential integrate-and-fire (EIF) model

$$\begin{aligned} \tau_m \frac{dV}{dt} &= -(V - E_L) + De^{(V-V_T)/D} + I_x(t) \\ V(t) > V_{th} &\Rightarrow \text{spike at time } t \text{ and } V(t) \leftarrow V_{re}. \end{aligned} \tag{1.5}$$

The second line in this equation describes the resetting of the membrane potential and it defines the class of "integrate-and-fire" models. More integrate-and-fire models are discussed in Appendix B.3. The term

See Appendix B.3 for a description of other simplified neuron models.

$$\Phi(V) = De^{(V-V_T)/D}$$

models the current induced by the opening of sodium channels that initiates the upswing of an action potential.

The parameter V_T should be chosen near the potential at which action potential initiation begins, *i.e.*, where sodium channels begin opening more rapidly ($V_T \approx -55$mV). Parameters for the EIF must satisfy $V_{re}, V_T < V_{th}$ and $D > 0$. Typically, we choose V_{re} near E_L. If we want to faithfully model the peak of an action potential, then we should choose $V_{th} \approx 0 - 10$mV, but choosing smaller values will not greatly affect spike times.

A detailed mathematical analysis shows that the membrane potential dynamics of the more detailed Hodgkin-Huxley model during the upswing of an action potential are accurately approximated by the sum of an exponential and linear function of V, as appear on the right side of Eq. (1.5) (see [5] or Chapter 5.2 of [2] for details). And careful experiments with real neurons also indicate that the EIF model accurate captures the upswing of an action potential [6]. For these and other reasons, the EIF model is sometimes regarded as an ideal **one-dimensional neuron model** [2, 4, 5] ("one-dimensional" meaning that the model is defined by a one-dimensional ODE).

Unlike the leaky integrator model, we cannot write an equation for the solution to Eq. (1.5). Instead, we can find an approximate, numerical solution to Eq. (1.5) using the forward Euler method as follows,

See Appendix A.6 for a review of the forward Euler method.

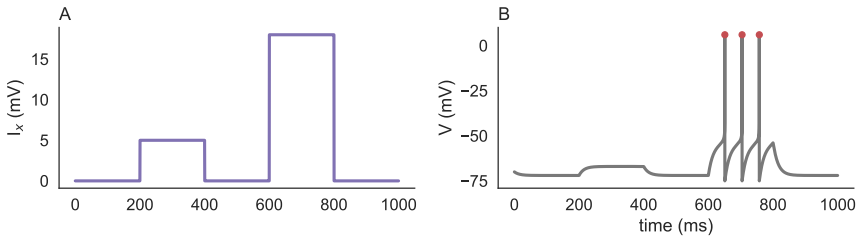

Figure 1.2: Simulation of an EIF neuron model. A) The input current and **B)** the membrane potential of an EIF neuron model. Red circles indicate the times at which the neuron spiked. Relevant parameters are $E_L = -72$, $V_{th} = 5$, $V_{re} = -75$, and $V_T = -55$mV. See `EIF.ipynb` for code to produce this figure.

```
for i in range(len(time)-1):
    # Euler step
    V[i+1]=V[i]+dt*(-(V[i]-EL)+DeltaT*np.exp((V[i]-VT)/DeltaT)+Ix[i])/
    taum
    # Threshold-reset condition
    if V[i+1]>=Vth:
        V[i+1]=Vre
        V[i]=Vth  # This makes plots nicer
        SpikeTimes=np.append(SpikeTimes,time[i+1])
```

See Figure 1.2 and `EIF.ipynb` for a complete simulation. The line `V[i]=Vth` sets the membrane potential to V_{th} just before it is reset to V_{re} in the event of a spike. This is added to make the plots of $V(t)$ look nicer, but has no effect on spike timing or dynamics. Try commenting out this line in `EIF.ipynb` to understand why it helps.

To better understand the dynamics of the EIF model, we begin with an intuitive explanation, then perform a more precise analysis. The intuitive explanation proceeds by considering two regimes of V.

1. **The leaky integrator regime.** When $V \ll V_T$ (meaning V is "way less than" V_T), then $\Phi(V) \approx 0$ so the EIF model behaves like a leaky integrator. The membrane potential is pulled toward $E_L + I_x$. The first 600ms in Figure 1.2 illustrates the leaky integrator regime.

2. **Spiking regime.** If V is larger than V_T, then $\Phi(V)$ gets larger and produces an inward current that models the opening of sodium channels. For sufficiently large V, this inward current creates a positive feedback loop: Increasing V causes $\Phi(V)$ to increase, which increases V further, causing $\Phi(V)$ to increase further, etc. This feedback loop models the upswing of the membrane potential at the initiation of an action potential. The EIF in Figure 1.2 is pushed into the spiking regime by the larger input pulse beginning at 600ms.

There is an intermediate regime in which the inward current, $\Phi(V)$, is appreciably larger than zero, but not strong enough to produce a positive feedback loop and initiate an action potential by itself. Instead, $\Phi(V)$ reduces the amount of positive input, $I_x > 0$, required to initiate an action potential. In this sense, $\Phi(V)$ implements a kind of **soft**

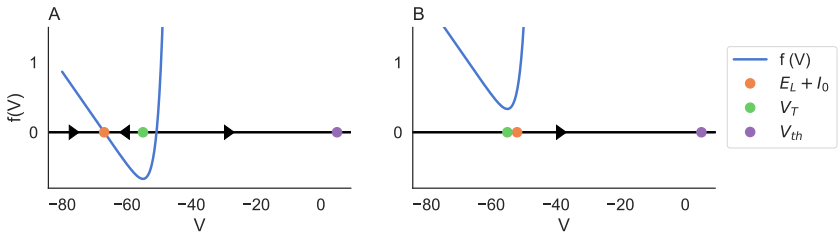

Figure 1.3: Phase line for the EIF model. A) Phase line for the EIF model with $I_0 = 5$mV and **B)** $I_0 = 20$mV. Parameters are $\tau_m = 15$ms, $E_L = -72$, $V_{th} = 5$, $V_{re} = -75$, $V_T = -55$, and $D = 2$. Draw the arrows and determine stability yourself. See `EIFphaseLine.ipynb` for code to produce this figure.

threshold for the EIF model near V_T: When $V > V_T$, an action potential is likely to occur, but it is not guaranteed.

The parameter D determines how soft the soft threshold is, *i.e.* how gradually the inward current increases as V increases toward V_T and beyond. When D is small, $\Phi(V) \approx 0$ whenever V is even just a little bit below V_T, but $\Phi(V)$ is large when V is just a little bit above V_T. This leads to a sharper threshold near V_T and a faster action potential upswing. When D is larger, $\Phi(V)$ increases more gradually when V is near V_T, leading to a softer threshold near V_T and a slower action potential upswing. Typically, we should choose $D \approx 1 - 5$mV.

See Appendix A.7 for a review of phase lines, fixed points, and stability for 1D ODEs.

To understand the dynamics of the EIF more precisely, we can consider the constant input case, $I_x(t) = I_0$, and perform a phase line analysis. The sub-threshold dynamics of the EIF with time-constant input obey

$$\frac{dV}{dt} = f(V)$$

where

$$f(V) = \frac{-(V - E_L) + De^{(V-V_T)/D} + I_0}{\tau_m}.$$

The phase line for $I_0 = 5$mV is plotted in Figure 1.3A. There is a stable fixed point near $E_L + I_0$ and an unstable fixed point a little above V_T. Therefore, as long as $V(0) < V_T$, no spike will occur because $V(t)$ will simply converge to the fixed point near $E_L + I_0$, similar to the leaky integrator model. Note also that $f(V)$ is approximately linear near $E_L + I_0$, because $\Phi(V)$ is small there, so the EIF behaves like the leaky integrator in this regime.

Increasing I_0 is equivalent to shifting up the blue curve, $f(V)$, in Figure 1.3. When we increase the input to $I_0 = 20$mV (Figure 1.3B), both fixed points disappear because $f(V) > 0$ for all V. Regardless of the initial condition, $V(0)$, the membrane potential will increase to V_{th} then reset to V_{re} and this process will repeat. Note that $f(V)$ grows exponentially when $V > V_T$, which means that $V'(t) = f(V)$ increases very rapidly whenever $V > V_T$. This explains the fast upswing in the action potentials in Figure 1.2B.

The existence of a stable and unstable fixed point when $I_0 = 5$mV and their disappearance when $I_0 = 20$mV in Figure 1.3 is a signature of a Saddle-Node bifurcation. Define I_{th} to be the value of I_0 at which the bifurcation occurs (where the two fixed points collide). This value allows us to precisely characterize the two regimes for an EIF with time-constant input.

1. **Sub-threshold regime:** If $I_0 \leq I_{th}$ then the membrane potential decays exponentially toward a fixed point and the neuron never spikes.
2. **Super-threshold regime:** If $I_0 > I_{th}$ then the membrane potential eventually reaches V_{th}, a spike is recorded, V is reset to V_{re}, and this process repeats indefinitely.

These two regimes are demonstrated by the response to the two different inputs given in Figure 1.2. Interestingly, despite the fact that the fixed points in the sub-threshold regime cannot be derived as a closed form expression, the threshold input can be written in closed form.

Exercise 1.2.1. The threshold input, I_{th}, of the EIF with time-constant input is defined as the value at which the neuron eventually spikes if $I_0 > I_{th}$, but never spikes $I_0 \leq I_{th}$. Derive a closed form equation for I_{th} for the EIF model. To verify your solution, plot the phase line and numerical simulations of the EIF model for $I_0 = I_{th} - 1\text{mV}$ and $I_0 = I_{th} + 1\text{mV}$.
Hint: The minimum of $f(V)$ can be found by setting $f'(V) = 0$, solving for V, then plugging this value of V back into $f(V)$. From Figure 1.3, we can see that the EIF spikes whenever the minimum of $f(V)$ is larger than zero. In your derivation, you will need to make a reasonable assumption that $V(0)$ and E_L are sufficiently smaller than V_T.

Exercise 1.2.2. The EIF model with super-threshold, time-constant input, $I_x(t) = I_0 > I_{th}$, spikes periodically. The interspike interval (ISI) is defined as the interval of time between a pair of consecutive spikes. We cannot derive an equation for the duration of an ISI, but we can compute it numerically by simulating the EIF with Euler's method. Use numerical simulations of the EIF model to compute the ISI duration, t_{ISI}, for several values of $I_0 > I_{th}$. Make a plot of t_{ISI} as a function of I_0. When a neuron spikes periodically, its firing rate is the reciprocal of the ISI duration, $r = 1/t_{ISI}$. Make a plot of r as a function of I_0. This is called an f-I curve (for "frequency-input").

1.3 MODELING SYNAPSES

We have so far considered very simple forms of external input, $I_x(t)$, modeling current injected by an electrode. We will now consider currents that model inputs from other neurons. A **synapse** is a connection between two neurons, the **presynaptic** neuron and **postsynaptic** neuron (see Fig. 1 and surrounding discussion). There are two fundamental types of synapses: ionotropic and metabotropic. We will focus on ionotropic synapses, which are faster and more "direct" than metabotropic synapses and we will not consider metabotropic synapses in this book.

When the presynaptic neuron spikes, neurotransmitter molecules are released and diffuse across the synapse to receptors on the postsynaptic neuron's membrane. These neurotransmitters open ion channels, which evokes a current across the postsynaptic neuron's membrane, called a **post-synaptic current (PSC)**. A PSC is transient because

the ion channels close again over the course of several milliseconds. Each PSC causes a transient response in the membrane potential of the postsynaptic neuron, called a **postsynaptic potential (PSP)**. In Figure 1B, the two bumps in the blue curve are PSPs evoked by the two spikes in the red curve.

There are two broad classes of (ionotropic) synapses: **excitatory** and **inhibitory**. Excitatory synapses evoke positive (inward) synaptic currents, pushing the membrane potential closer to threshold, thereby "exciting" the neuron. Inhibitory synapses evoke negative (outward) synaptic currents, pushing the membrane potential away from threshold, thereby "inhibiting" action potentials.

The polarity (excitatory or inhibitory) of a synapse is determined by the type of neurotransmitter released and the type of receptor on the postsynaptic neuron's membrane. The most common type of excitatory neurotransmitter in the cortex is **glutamate**. A common type of receptor for glutamate are α-amino-3-hydroxy-5-methyl-4-isoxazolepropionic acid (**AMPA**) receptors. The most common type of inhibitory neurotransmitter in cortex is γ-aminobutyric acid (**GABA**) and a common receptor type is **GABA**$_B$. The decay timescales of AMPA and GABA$_B$ receptors is around $\tau_s \approx 3 - 10$ ms with AMPA a little bit faster than GABA$_B$.

Dales Law says that a presynaptic neuron connects to all of its postsynaptic targets with the same "type" of synapse. For our purposes, this will be taken to mean that all postsynaptic targets of a single neuron are either excitatory or inhibitory. Laws in neuroscience are almost never completely universal, but Dale's law has very few exceptions. When considering cortical neurons of adult mammals under healthy conditions, one can safely assume Dale's law.

Because of Dale's law, we can classify neurons as **excitatory neurons** or **inhibitory neurons**, instead of just classifying *synapses* as excitatory or inhibitory. An excitatory neuron is a neuron that connects to all of its postsynaptic targets with excitatory synapses, and similarly for inhibitory neurons. In cortex, about 80% of neurons are excitatory neurons and 20% are inhibitory. Most excitatory neurons in cortex are **pyramidal neurons**, named for the pyramid-like appearance of their soma. Most inhibitory neurons in cortex are classified as **cortical interneurons**, where the "interneuron" label reflects the fact that they only project locally, to nearby neurons in the same cortical area or layer. In the cortex, long range projections to other cortical areas are almost exclusively made by excitatory (pyramidal) neurons.

See Appendix B.4 for a description of conductance-based synapse models.

The current generated by a synapse across the postsynaptic neuron's membrane is called the **synaptic current**. We will consider a model in which synaptic currents are generated directly from presynaptic spike times. This is called a **current-based synapse model**. A more biologically realistic approach generates a synaptic *conductance* from spike times and the synaptic current is generated from this conductance. These **conductance-based synapse models** are discussed in Appendix B.4 along with a mathematical approach to approximate conductance-based models with current-based models.

To construct out current-based model, first consider an excitatory presynaptic neuron and define s_j^e be the time of its jth spike. We can model the synaptic current for an excitatory synapse as a sum of **excitatory PSCs (EPSCs)**,

$$I_e(t) = J_e \sum_j \alpha_e(t - s_j^e). \tag{1.6}$$

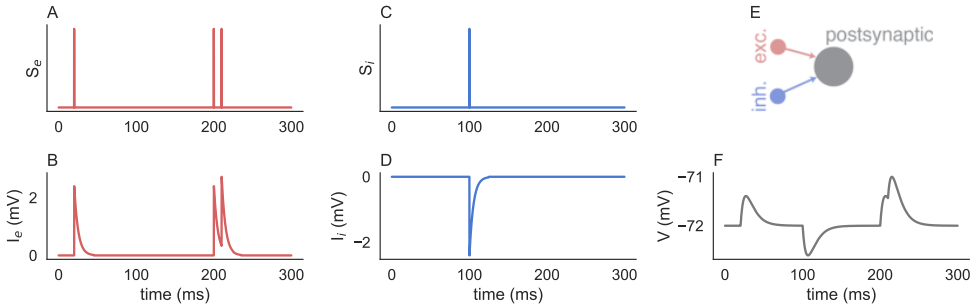

Figure 1.4: A leaky integrator model driven by two synapses. A) Excitatory spike density. Each vertical bar represents a spike in the excitatory presynaptic neuron, modeled as a Dirac delta function. **B)** Excitatory synaptic current generated by the spikes in A using an exponential synapse model. **C)** Inhibitory spike density. **D)** Inhibitory synaptic current. **E)** Schematic of an excitatory and inhibitory neuron connected to a postsynaptic neuron. **F)** Membrane potential of the postsynaptic neuron modeled as a leaky integrator. See `Synapses.ipynb` for code to produce this figure.

The term $\alpha_e(t) \geq 0$ is a non-negative function called the **excitatory post-synaptic current (EPSC) waveform** and J_e is a **synaptic weight** or **synaptic strength** which scales the sign and magnitude of the synaptic current. Hence, Eq. (1.6) can be interpreted as adding a "copy" of the EPSC waveform, $\alpha_e(t)$, scaled by the weight, J_e, at each presynaptic spike time, s_j^e. Figure 1.4B shows synaptic current generated by presynaptic spikes that are indicated by the vertical bars in Figure 1.4A. The e subscripts and superscript in Eq. (1.6) refers to the fact that this is an excitatory synapse. For inhibitory synapses, we just replace e with i. Note that we must have $J_e > 0$ and $J_i < 0$. We can additionally model the postsynaptic neuron's membrane potential using a leaky integrator model. Putting this all together gives a model of a postsynaptic neuron driven by an excitatory and inhibitory synapse,

$$\begin{aligned} \tau_m \frac{dV}{dt} &= -(V - E_L) + I_e(t) + I_i(t) \\ I_e(t) &= J_e \sum_j \alpha_e(t - s_j^e) \\ I_i(t) &= J_i \sum_j \alpha_i(t - s_j^i) \end{aligned} \tag{1.7}$$

Which function should we use for the PSC waveforms, $\alpha_e(t)$ and $\alpha_i(t)$? We should always have $\alpha_e(t) = \alpha_i(t) = 0$ when $t < 0$ because the postsynaptic response cannot precede the presynaptic spike. Since the sign and strength of the synapse is captured by J, we can always assume that

$$\int_0^\infty \alpha_a(t)dt = 1 \quad \text{for } a = e, i$$

because we can absorb the sign and magnitude of $\alpha_a(t)$ into J_a.

Most synaptic currents rise much faster than they decay, and their decay is approximately exponential. A very simple model of a synaptic current is therefore given by an instantaneous rise and exponential decay,

$$\alpha_a(t) = \frac{1}{\tau_a} e^{-t/\tau_a} H(t) = \begin{cases} \frac{1}{\tau_a} e^{-t/\tau_a} & t \geq 0 \\ 0 & t < 0 \end{cases} \tag{1.8}$$

for $a = e, i$ where

$$H(t) = \begin{cases} 1 & t \geq 0 \\ 0 & t < 0 \end{cases}$$

is the Heaviside step function and τ_a is the **synaptic time constant** or **decay time constant** that controls how quickly the PSC decays. The $1/\tau_a$ factor in the definition of $\alpha_a(t)$ assures that $\int \alpha_a(t) dt = 1$. This form of $\alpha_a(t)$ is sometimes called an **exponential synapse model**. Synaptic timescales of many ionotropic synapses in the cortex are around $\tau_a \approx 5 - 10$ms.

Figure 1.4 shows a full simulation of the model in Eqs. (1.7) exponential synapse models for $\alpha_e(t)$ and $\alpha_i(t)$. You can see in Figure 1.4F that each isolated presynaptic spike evokes a PSP. PSPs from excitatory presynaptic spikes are called **excitatory postsynaptic potentials (EPSPs)** and those from inhibitory presynaptic spikes are called an **inhibitory postsynaptic potentials (IPSPs)**. The **PSP amplitude** is the height of a PSP (the distance from the peak of the PSP to the resting membrane potential). PSP amplitudes in cortex are often between 0 and 1mV. PSP amplitudes are proportional to the synaptic weight, J_e or J_i, but also depend on the synaptic timescales, τ_e or τ_i, and membrane time constant, τ_m. When building simulations, it's easiest to choose the timescales and time constants first, then choose the synaptic weights by trial-and-error to get the PSP amplitude you want.

See Appendix A.8 for a review of Dirac delta functions.

Before describing how to simulate the model in Eqs. (1.7) to generate Figure 1.4, we first need to describe two ways of reformulating the model that make it easier to simulate. First, we write the presynaptic spike trains as sums of Dirac delta functions,

$$S_e(t) = \sum_j \delta(t - s_j^e)$$
$$S_i(t) = \sum_j \delta(t - s_j^i)$$

This representation of a spike train is called a **spike density**. Specifically, the spike density representation of a spike train is a time series with a Dirac delta function at each spike time. Spike density representations of spike trains can simplify the mathematics and coding in many situations in computational neuroscience. Figures 1.4A,C show spike density representations of the presynaptic spike trains when vertical bars are used to represent Dirac delta functions. Python code for turning a list of spike times into a spike density is given

```
Se=np.zeros_like(time)
Si=np.zeros_like(time)
Se[(ExcSpikeTimes/dt).astype(int)]=1/dt
Si[(InhSpikeTimes/dt).astype(int)]=1/dt
```

where `ExcSpikeTimes` and `InhSpikeTimes` are vectors of spike times. This code sets indices corresponding to spike times to the value `1/dt`, representing a Dirac delta function in discrete time. Conversion to an integer using `astype(int)` always rounds down in NumPy, so this code should avoid indexing errors if `time=np.arange(0,T,dt)` and all the entries in the spike time vectors are less than T.

Using the spike density representation of the presynaptic spike train allows us to write the synaptic currents as convolutions, so we can reformulate our model as

$$\begin{aligned} \tau_m \frac{dV}{dt} &= -(V - E_L) + I_e(t) + I_i(t) \\ I_e(t) &= J_e(\alpha_e * S_e)(t) \\ I_i(t) &= J_i(\alpha_i * S_i)(t) \end{aligned} \tag{1.9}$$

Think about why Eqs. (1.7) and (1.9) are equivalent. To simulate this formulation of the model in Python, we would use the code

```
# Use convolutions to define synaptic currents
s=np.arange(-5*taue,5*taue,dt)
alphae=(1/taue)*np.exp(-s/taue)*(s>=0)
alphai=(1/taui)*np.exp(-s/taui)*(s>=0)
Ie=Je*np.convolve(Se,alphae,'same')*dt
Ii=Ji*np.convolve(Si,alphai,'same')*dt
# Euler solver to compute Is and V
V[0]=EL
for i in range(len(time)-1):
    V[i+1]=V[i]+dt*(-(V[i]-EL)+Ie[i]+Ii[i])/taum
```

The first code cell of `Synapses.ipynb` contains the full code to generate Figure 1.4 using this formulation.

The second way to reformulate the synapse model from Eqs. (1.7) only works for exponential synapse models. For exponential synapse models, the exponential decay that occurs between spikes can be represented by a linear ODE. Therefore, the synapse model can be defined by a linear ODE with the added condition that we increment I_s at each presynaptic spike. The spike density representation again makes the model easier to formulate. Specifically, the model can be written as

Leaky integrator with excitatory and inhibitory synapses

$$\begin{aligned} \tau_m \frac{dV}{dt} &= -(V - E_L) + I_e(t) + I_i(t) \\ \tau_e \frac{dI_e}{dt} &= -I_e + J_e S_e \\ \tau_i \frac{dI_i}{dt} &= -I_i + J_i S_i \\ S_e(t) &= \sum_j \delta(t - s_j^e) \\ S_i(t) &= \sum_j \delta(t - s_j^i). \end{aligned} \tag{1.10}$$

The idea is that each delta function in $S_e(t)$ causes a jump of size J_e/τ_e in $I_e(t)$, then

$I_e(t)$ decays exponentially toward zero between spikes (and similarly for $I_i(t)$). Indeed, Eq. (1.10) is equivalent to Eqs. (1.7) and (1.9) (assuming that exponential synapse models are used and initial conditions are $I_e(0) = I_i(0) = 0$).

In Python, we can use the forward Euler method to solve Eqs. (1.10) as follows,

```
V[0]=EL
for i in range(len(time)-1):
    V[i+1]=V[i]+dt*(-(V[i]-EL)+Ie[i]+Ii[i])/taum
    Ie[i+1]=Ie[i]+dt*(-Ie[i]+Je*Se[i])/taue
    Ii[i+1]=Ii[i]+dt*(-Ii[i]+Ji*Si[i])/taui
```

The second code cell in `Synapses.ipynb` contains the full code to generate Figure 1.4 using this formulation.

The different formulations of the synapse model described above each have their own advantages and disadvantages. The formulation in Eqs. (1.7) is easy to understand and does not require a spike density representation of the presynaptic spike trains, but it is difficult to implement directly in code. The formulation in Eqs. (1.9) is easy to implement in code, but it requires knowledge of all spike times ahead of time. Later, we will consider networks in which spikes are generated while simulating membrane potentials, so we cannot define synaptic currents before solving for the membrane potentials. The formulation in Eqs. (1.10) is also easy to implement in code and allows us to solve for membrane potentials and synaptic currents simultaneously, which will be useful when modeling networks. Before we can start modeling networks, however, we must learn to model neural variability.

Exercise 1.3.1. Using the exponential PSC waveform causes $I_e(t)$ and $I_i(t)$ to jump discontinuously after each presynaptic spike. In reality, the currents rise continuously, but quickly. Some studies use a difference of exponentials,

$$\alpha_a(t) = \frac{1}{\tau_2 - \tau_1}\left(e^{-t/\tau_2} - e^{-t/\tau_1}\right) H(t)$$

for $a = e, i$ to obtain a continuous PSC. In a new file, reproduce Figure 1.4, but replace the exponential PSCs with a difference of exponentials, using $\tau_2 = 4$ms and $\tau_1 = 0.5$ms (called the "decay" and "rise" timescales respectively). This is more difficult to represent with ODEs, so you should use the formulation of the model from Eqs. (1.9) (the first code cell in `Synapses.ipynb`). Visually compare the results in the two cases. If the ultimate goal is to model $V(t)$, do we need the more realistic, but more complicated difference of exponentials model?

2 MEASURING AND MODELING NEURAL VARIABILITY

2.1 SPIKE TRAIN VARIABILITY, FIRING RATES, AND TUNING

Figure 2.1A shows a recording of a membrane potential from a real neuron in the brain of a rat[1] [7]. Some features match what we say in Chapter 1: The membrane potential hovers near -70mV with the exception of brief action potentials to near 0mV. However, there are other features that are noticeably different. Most notably, the membrane potential fluctuates in a seemingly random fashion between spikes and the spike times are irregular. These types of irregularity are ubiquitous in recordings made from living animals and are broadly known as **neural variability**. The causes and effects of neural variability are a topic of active research and debate.

To model and quantify neural variability, we will begin by studying the irregularity of spike times, which is often called **spike timing variability**. To begin studying neural variability, we first define the **spike count** over a time interval,

Spike count definition

$$N(a,b) = \#\text{ of spikes in } [a,b].$$

Given a single recording, we can compute the spike count over the entire recording interval, for example $N(0,5000) = 11$ for the recording in Figure 2.1A. We can also compute the spike counts over consecutive intervals of a given length. The following code computes firing rates over consecutive intervals of length 500ms,

```
dtRate=500
SpikeCountTime=np.arange(0,5001,dtRate)
SpikeCounts=np.histogram(SpikeTimes,SpikeCountTime)[0]
```

The call to the `histogram` function counts the number of elements in `SpikeTimes` between every pair of consecutive elements in `SpikeCountTime`. The results are shown in Figure 2.1B and full code to produce the figure is given in `OneRealSpikeTrain.ipynb`.

As discussed in Chapter 1, a neuron will not typically spike twice within a 2ms time window. Therefore, if we discretize time into very small intervals, $dt < 2$ms, then each bin should contain either one spike or zero spikes. This is called a **binarized spike train** because it is a binary time series (0's and 1's). See Figure 2.1C for an example of a binarized spike train with bin width $dt = 0.1$ms.

We're often interested in the frequency of spikes, *i.e.*, the number of spikes per unit time, which is called a neuron's **firing rate** or just **rate**,

1 Thanks to Micheal Okun and Ilan Lampl for sharing the data from Figure 2.1. More information about the data and its collection can be found in [7].

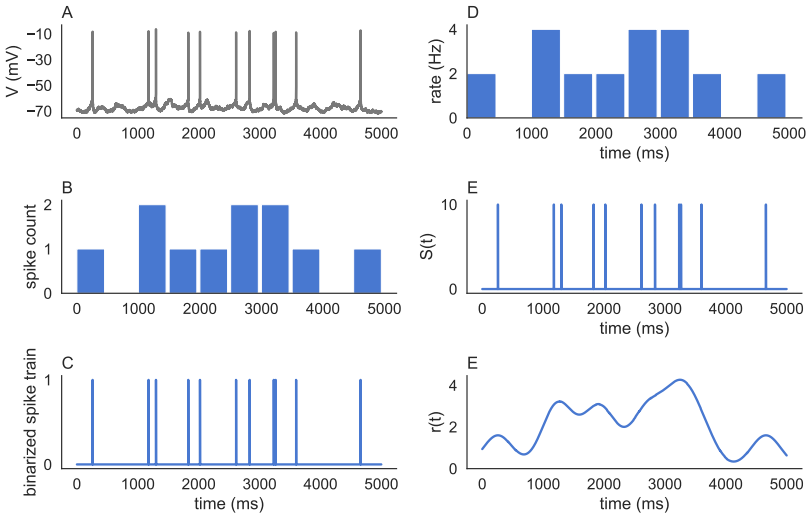

Figure 2.1: Membrane potential, spike counts, and time dependent firing rate of a real neuron. A) The membrane potential recorded from a rat cortical neuron. The potential was shifted downward to correct for possible recording artifacts. **B)** Spike counts over consecutive 500ms intervals. **C)** Time-dependent firing rate computed over the same intervals. See `OneRealSpikeTrain.ipynb` for code to produce this plot.

Firing rate definition

$$r = \text{firing rate} = \frac{\#\text{ of spikes}}{\text{length of time window}} = \frac{N(a,b)}{b-a}.$$

Firing rates have dimension "number of spikes per unit time." If we think of "number of spikes" as dimensionless, then this is a frequency. If time is measured in ms then r has units "spike per ms" or kiloHertz (kHz). But we usually report firing rates in units of spikes per second or Hertz (Hz). Since we usually keep track of time in ms, this often requires us to rescale firing rates to Hz by multiplying by 1000 (because 1kHz=1000Hz). Neurons in the cortex usually spike at around 1-50Hz.

Each measure of the spike count discussed above has an analogue in firing rates. As with spike counts, we can measure the firing rate over an entire recording, which can be called the **time-averaged firing rate**. For example, the time-averaged rate in the recording from Figure 2.1A is 2.2Hz. We can also measure the firing rate over sequential time intervals, which is called a **time-dependent firing rate**, such as the time-dependent rate plotted in Figure 2.1D. Note that the firing rates in Figure 2.1D are just the spike counts from Figure 2.1B scaled by a factor of 2 since the time intervals are $b - a = 0.5$ seconds long. If we measure firing rates over small time intervals, $dt < 2$ms, then each bin in the time-dependent rate should contain either a zero or a $1/dt$. This is just a discrete-time representation of the spike density,

$$S(t) = \sum_j \delta(t - s_j)$$

where s_j is the jth spike time. Figure 2.1E shows the time-dependent firing rate (or discretized spike density) with a time bin size of $dt = 0.1$ms.

The firing rate estimate in Figure 2.1D is discontinuous and jagged because we counted spikes over non-overlapping intervals. We often want an estimate of the firing rate that is continuous in time. One option is to use overlapping time intervals. A more parsimonious example is to convolve the spike density with a smoothing kernel,

$$r(t) = (k * S)(t)$$

where $k(t)$ is a kernel satisfying $\int k(t)dt = 1$. The resulting time series, $r(t)$, is also called a time-dependent firing rate, but it is smoother than the one defined by counting spikes over disjoint time intervals. Using a Gaussian-shaped kernel is common and can be accomplished in code as follows

```
sigma=250
s=np.arange(-3*sigma,3*sigma,dt)
k=np.exp(-(s**2)/(2*sigma**2))
k=k/(sum(k)*dt)
SmoothedRate=np.convolve(k,S,'same')*dt
```

Figure 2.1F shows the smoothed rate obtained by applying this method to the spike times from that figure. See `OneRealSpikeTrain.ipynb` for the complete code.

The **interspike intervals (ISIs)** of a spike train are defined as the time intervals between consecutive spikes. Specifically, if s_j is the jth spike time then the jth ISI is

$$I_j = s_{j+1} - s_j$$

and they can be computed in NumPy using the `diff` function,

```
ISIs=np.diff(SpikeTimes)
```

Spike timing variability is often quantified by the **coefficient of variation (CV)** of the ISIs, defined as the standard deviation of the ISIs divided by their mean ISI,

$$CV = \frac{\text{std}(ISI)}{\text{mean}(ISI)}$$

where std stands for the standard deviation. Because the mean and standard deviation of the ISIs have the same units, the CV is a dimensionless quantity. A periodic spike train has $\sigma_{ISI} = 0$ so $CV = 0$. A larger CV implies more irregular spike times.

Exercise 2.1.1. Compute the CV for the spike train in `OneRealSpikeTrain.ipynb`

If all you think all of this analysis seems over-the-top for the handful of spikes in Figure 2.1, you're correct. More commonly, a neuron will be recorded for a longer duration, or it will be recorded across repetitions of the same experiment (*e.g.*, several presentations of the same stimulus). Each repetition is sometimes called a **trial**.

The data file `SpikeTimes1Neuron1Theta.npz` contains spike times recorded from a neuron in a monkey's visual cortex over the course of 200 trials[2]. During each 1s trial,

2 Thanks to Adam Kohn and Matthew Smith for sharing the data which was recorded from an array of extracellular electrodes. More information about the data and its collection can be found in [8]

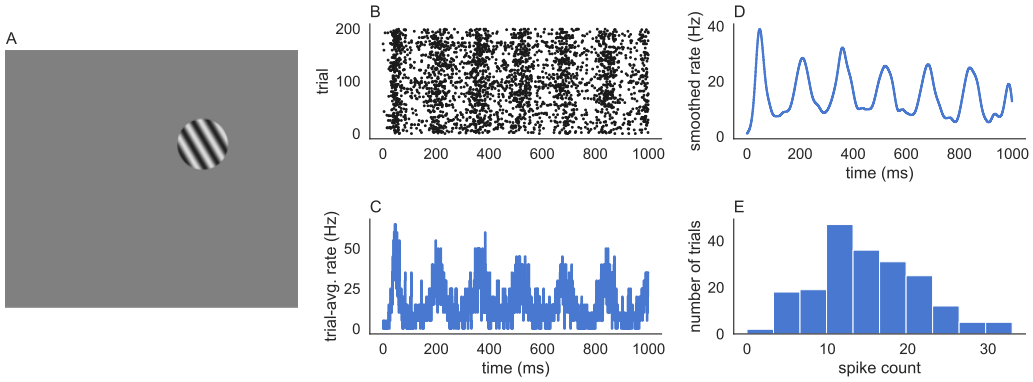

Figure 2.2: Analysis of spike trains recorded from a real neuron across 200 trials. A) Example of a drifting grating visual stimulus. The white/black lines drift slowly across the circle. **B)** Raster plot. Each dot is a spike at the indicated time and trial. **C)** Time-dependent, trial-averaged firing rate computed by counting the number of spikes across all neurons in each time bin. **D)** Smoothed, trial-averaged firing rate. **E)** Histogram of spike counts across trials. Code to produce this figure can be found in `MultiTrialSpikeTrains.ipynb`.

the monkey watched the same "drifting grating" movie in which angled bars drifted across part of the monkey's visual field (see Figure 2.2A for a still image of a drifting grating). The file has a variable `theta` representing the angle of the stimulus for this recording, which is 300°.

The spike times of every spike across all trials is stored in `SpikeTimes` and the corresponding trial numbers are stored in `TrialNumbers`. For example, `SpikeTimes[3]==4.36` and `TrialNumbers[3]==42`, indicating that there was a spike on trial number 42 at 4.36ms after the start of the trial. Figure 2.2B shows a **raster plot** of the spike times. In a raster plot, each dot represents a spike at the corresponding time and trial. A raster plot can be generated as follows,

```
plt.plot(SpikeTimes,TrialNumbers,'.')
```

See `MultiTrialSpikeTrains.ipynb` for the full code to generate Figure 2.2B.

The list of spike times and trial numbers is a memory-efficient way to store multiple spike trains, but it can make it difficult to perform operations on the data. Another option is to use an array of spike densities. Mathematically, we can think of this as a vector, $\boldsymbol{S}$, of spike densities where the kth entry of the vector represents the spike density of the kth trial,

$$\boldsymbol{S}_k(t) = \sum_j \delta(t - s_j^k)$$

In NumPy, it would be represented with an array in which `S[k,:]` is the kth spike density. This array can be created as follows,

```
S=np.zeros((NumTrials,len(time)))
S[TrialNumbers,(SpikeTimes/dt).astype(int)]=1/dt
```

Once this array is created, it becomes very easy to compute firing rates by taking averages of `S` across time and/or trials. The **time-averaged, trial-averaged firing rate** is a single number computed by

```
TrialAvgTimeAvgRate=np.mean(S)
```

For the data shown in Figure 2.2, we get 15.175Hz. We can alternatively compute the **time-dependent, trial-averaged firing rates** by averaging over trials only,

```
TrialAvgRates=np.mean(S,axis=0)
```

The result is plotted in Figure 2.2C. The oscillations apparent in Figure 2.2C are caused by the periodic movement of the drifting grating stimulus as it drifts. While the oscillatory trend in Figure 2.2C is visible, it appears choppy and noisy. A smoothed firing rate can be obtained by convolving with a Gaussian kernel. You might be tempted to convolve each `S[k,:]` with a kernel and then average, but it is equivalent (and much simpler) to convolve `TrialAvgRates` with the same kernel,

```
sigma=10
s=np.arange(-3*sigma,3*sigma,dt)
k=np.exp(-(s**2)/(2*sigma**2))
k=k/(sum(k)*dt)
SmoothedRate=np.convolve(k,TrialAvgRates,'same')*dt
```

The result is plotted in Figure 2.2D and the oscillations are more clearly visible.

So far, we have averaged the data across trials, but Figure 2.2B shows clear variability across trials as well. We can compute the spike count on each trial as the Riemmann sum of `S` across time

```
SpikeCounts=np.sum(S,axis=1)*dt
```

After running this line of code, `SpikeCounts` is a vector for which each entry is the spike count on a corresponding trial. The spike count data can be visualized by plotting a histogram,

```
plt.hist(SpikeCounts)
```

The result is plotted in Figure 2.2E. Even though the average spike count is 15.175, there is substantial variability across trials. In other words, even though the animal is viewing the same drifting grating stimulus on every trial, the neuron emits a different number of spikes on each trial. This is known as **trial-to-trial variability** or simple **trial variability**. Trial variability of spike counts is often measured using the **Fano factor**, defined as the ratio between the variance and mean of spike counts across trials,

$$FF = \frac{\text{variance of spike counts}}{\text{mean spike count}} = \frac{\text{var}(N(a,b))}{\text{mean}(N(a,b))}$$

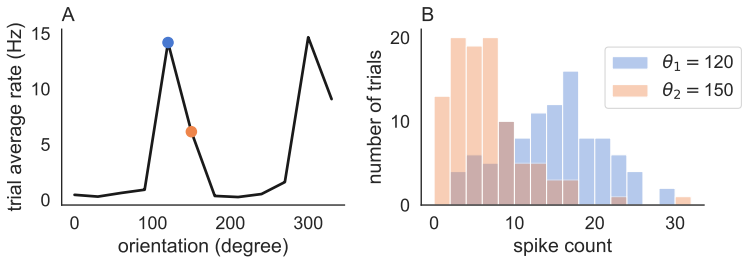

Figure 2.3: Orientation tuning curve and spike count histograms from one neuron. A) Tuning curve (trial averaged rate as a function of orientation) for a neuron recorded in monkey V1. Blue and pink dots mark the orientations $\theta_1 = 120°$ and $\theta_2 = 150°$. **B)** Histogram of spike counts across trials for the θ_1 and θ_2. Code to produce this plot can be found in `OrientationTuningCurve.ipynb`.

The Fano factor can be computed over any interval, $[a, b]$, but we often just use the entire recording, $[0, T]$. If a neuron spikes exactly the same number of times on every trial then $\text{var}(N) = 0$ so $FF = 0$. More generally, a larger Fano factor implies greater trial-to-trial variability.

Exercise 2.1.2. Compute the Fano Factor for the spike trains `MultiTrialSpikeTrains.ipynb`

Neurons encode information about stimuli and behavior in their spike trains. For example, neurons in the visual cortex change their firing rates in response to changes in a visual stimulus and, similarly, neurons in the motor cortex change their firing rates during motor behavior. It is widely believed that the encoding of information in neurons' spike trains forms the basis of perception, cognition, and behavior, but the details of how this happens are not understood.

A classical example of neural coding is **orientation tuning** in primary visual cortex (V1). In the 1950s and 60s, David Hubel and Torsten Wiesel discovered that some neurons in cat V1 increased their firing rates when a bar of light passed through a small region of the cat's visual field. Different neurons respond to bars of light in different regions of the visual field. A neuron's **receptive field** is the region to which it responds. Many neurons' firing rates depend on the angle or "orientation" of the bar of light. The orientation that evokes the highest firing rate is called the neuron's **preferred orientation**. Hence, firing rates of neurons in V1 encode the location and orientation of bars of light. The encoding of orientations in V1 is still widely studied today.

The spike trains shown in Figure 2.2 were recorded in response to a drifting grating at one particular orientation ($\theta = 300°$), but the neuron was actually recorded across 12 different orientations $(0, 30, 60, \ldots, 330°)$ with several trials for each orientation. The file `SpikeCounts1Neuron12Thetas.npz` contains an array with the spike counts for all 100 trials on each of the 12 orientations. Trial-averaged firing rates for each orientation can be computed an plotted by

```
Rates=SpikeCounts/T
TrialAvgRates=np.mean(Rates,axis=0)
plt.plot(AllOrientations,1000*TrialAvgRates)
```

The result (Figure 2.3A) is called the neuron's **tuning curve** because it depicts how the neuron is "tuned" to each orientation, *i.e.*, how the neuron's trial-averaged firing rate is affected by each orientation. Why do there appear to be two preferred orientations? Think about the properties of the drifting grating stimulus to answer this question.

Figure 2.3A only shows the trial-averaged response of the neuron, but we know from above that there is trial-to-trial variability. Indeed, Figure 2.3B shows the histograms of spike counts for each of the two orientations. Let's suppose you tried to decode infer the orientation of the stimulus by looking at the neuron's spike count. You would not be able to correctly infer the orientation on every trial (why?). However, there are millions of neurons in the animal's visual cortex, each with their own tuning curve. If you could observe the spike counts of all of them (and you knew their tuning curves), you could infer the orientation more accurately. The science of understanding how stimuli are encoded in neural population activity, and how they can be decoded, is called **neural population coding**. Appendix B.5 discusses neural population coding in more depth. The last exercise in Section 4.3, uses an artificial neural network to decode spike counts from neural populations.

See Appendix B more in-depth discussion of ne population codi

2.2 MODELING SPIKE TRAIN VARIABILITY WITH POISSON PROCESSES

The causes and effects of spike timing variability are not fully understood, but we often need to model it. To account for the variability of real spike trains, we can model them as a stochastic process. A **stochastic process** is similar to a random variable, but it is a random function of time. Specifically, we will model a spike train as a special type of stochastic process called a **point process**, which is a stochastic process representing discrete events in time. Here, those events are spikes. Two common ways to represent point processes are the counting process,

$$n(t) = \text{ \# of spike in } [0,t] = N(0,t)$$

and the spike density, $S(t)$. Modeling spike trains as stochastic processes allows us to make precise definitions of probabilities, statistics, and other properties. For example, the **instantaneous firing rate** is defined as

Instantaneous firing rate definition

$$r(t) = \lim_{\delta \to 0} \frac{E[N(t, t+\delta)]}{\delta} = \frac{d}{dt} E[n(t)] = E[S(t)] \tag{2.1}$$

where $E[\cdot]$ denotes expectation. The last equality is difficult to make precise because $S(t)$ is an unusual type of process (being composed of Dirac delta functions), but it will be useful later. Intuitively, $\frac{d}{dt} n(t) = S(t)$ because $n(t) = \int_0^t S(\tau) d\tau$, so $\frac{d}{dt} E[n(t)] = E[n'(t)] = E[S(t)]$. This intuition will have to suffice for our purposes. The spike count, $N(a,b)$, is a random number and

$$E[N(a,b)] = \int_a^b r(t) dt.$$

A stochastic process is said to be **stationary** if its statistics do not change across time, *i.e.*, the statistics of $x(t)$ are the same as those of $y(t) = x(t + t_0)$. For point processes, stationarity can be phrased as follows,

> **Definition:** A point process is stationary if $N(a, b)$ has the same distribution as $N(a + t_0, b + t_0)$ for any t_0 and any $a < b$.

In other words, the distribution of $N(a, b)$ only depends on $b - a$. Therefore, for stationary point processes, we can often just talk about $n(t)$ instead of $N(a, b)$ since they have the same statistics when $t = b - a$. This property gives the following useful theorem:

> **Theorem:** A stationary point process has a constant rate, $r(t) = r$, and therefore $E[n(t)] = rt$.

Poisson processes provide a canonical statistical model of noisy spike trains. The simplest and most common version of a Poisson process is the **homogeneous Poisson process**, sometimes called the **stationary Poisson process**. We will use the term **Poisson process** to mean "homogeneous Poisson process." Inhomogeneous Poisson processes are discussed later. A Poisson process is defined as follows,

> **Definition:** A (homogeneous) Poisson process is any stationary point process having the **memoryless property**: $N(t_1, t_2)$ is independent from $N(t_3, t_4)$ whenever $[t_1, t_2]$ is disjoint from $[t_3, t_4]$.

The basic idea is that a spike is equally likely to occur in a Poisson process at any point of time, regardless of how many spikes occur over any interval of time before or after it. This is, of course, not exactly true of real spike trains, but it serves as a simplifying approximation.

The Poisson process gets its name from the following property:

> **Theorem:** The spike counts of a Poisson process obey a Poisson distribution,
>
> $$\Pr(n(t) = n) = \frac{(rt)^n}{n!} e^{-rt}.$$

A Poisson distribution has the same variance and mean, $\text{var}(n(t)) = E[n(t)] = rt$, which implies that Poisson processes have a Fano factor of 1 over any time window:

$$FF_{Poisson} = \frac{\text{var}(N(a, b))}{E[N(a, b)]} = 1.$$

for any $a < b$. Of course, a sample Fano factor computed from sample Poisson processes will not be exactly equal to 1, but it should converge to 1 as the number of trials goes to ∞, by the law of large numbers. A Fano factor of 1 is interpreted as a baseline amount of trial-to-trial variability. The phrases **sub-Poisson** and **super-Poisson** are sometimes used to refer to spike trains with $FF < 1$ or $FF > 1$ respectively.

We can also consider variability in the ISIs, $I_j = s_{j+1} - s_j$ where s_j is the jth spike time. Poisson processes have exponentially distributed ISIs,

> **Theorem:** The inter-spike intervals (ISIs) of a Poisson process are i.i.d. with an exponential probability density function
>
> $$p_I(t) = re^{-rt} H(t)$$
>
> where $H(t)$ is the Heaviside step function.

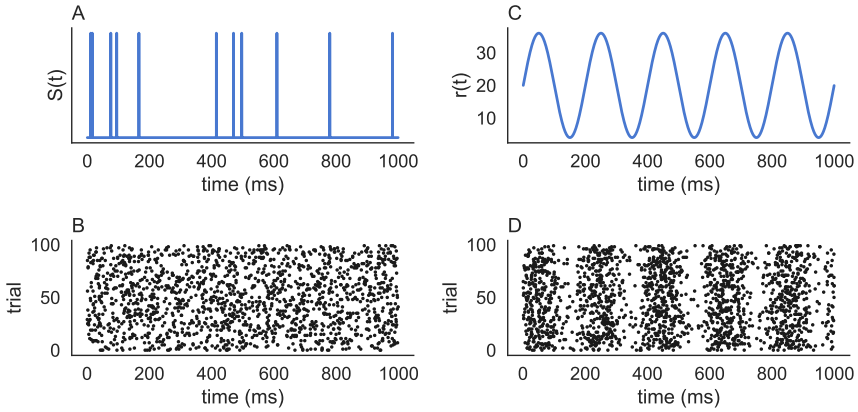

Figure 2.4: Poisson processes. A) Raster plot of a single Poisson process with $r = 15$Hz. Each vertical bar is a spike. **B)** Raster plot of 100 i.i.d. realizations of a Poisson processes with $r = 15$Hz. Each dot is a spike. **C,D)** Firing rate and raster plot of 100 i.i.d. realizations of an inhomogeneous Poisson process. See `PoissonProcesses.ipynb` for code to produce these plots.

Since exponential distributions have the same mean and standard deviation ($\text{std}(I_j) = E[I_j] = 1/r$), the CV of a Poisson process is 1,

$$CV_{Poisson} = 1.$$

Therefore, $CV = 1$ is interpreted as a baseline amount of ISI variability. The phrases **sub-Poisson** and **super-Poisson** are sometimes also used to refer to $CV < 1$ and $CV > 1$ respectively. If $CV < 1$ then spikes are closer to periodic. If $CV > 1$ then spikes occur in a bursty manner (periods of fast spiking and periods of relative silence).

There are several different algorithms for generating realizations of Poisson processes and we will consider two of them. First, an algorithm that generates spike times directly

> **Poisson Process Algorithm 1:** To generate spike times of a Poisson process in the interval $[0, T]$, first generate the spike count from a Poisson distribution with mean rT, then distribute the spike times uniformly in the interval.

In Python, this is implemented by:

```
N=np.random.poisson(r*T)
SpikeTimes=np.sort(np.random.rand(N)*T)
```

Exercise 2.2.1. Generate a Poisson process with rate $r = 10$Hz over a time interval of duration $T = 5$s and compute the sample CV. Repeat this process 5 times with a newly generated Poisson process to see how the sample CV varies each time. Then do the same thing for $T = 20$s and $T = 100$s.

The next algorithm generates a spike density, $S(t)$, instead of spike times.

> **Poisson Process Algorithm 2:** To generate a spike density representation of a Poisson process over the interval $[0, T]$, first make sure that $r * dt$ is small.

Then set each bin of $S(t)$ to $1/dt$ independently with probability $r * dt$ and set all other bins to zero.

For a binarized spike train, you would just set the bins to 1 instead of $1/dt$. In Python, this algorithm can be implemented in a single line,

```
S=np.random.binomial(1, r*dt, len(time))/dt
```

Figure 2.4A shows a spike density of a Poisson process. Algorithm 2 assumes that $r * dt$ is small enough so that you can safely ignore the small probability of two spikes in the same bin. You can avoid this assumption by sampling each bin from a Poisson distribution with mean $r * dt$,

```
S=np.random.poisson(r*dt, len(time))/dt
```

This approach can produce multiple spikes per bin and it runs more slowly when `time` is large, but it will produce more accurate Poisson statistics when $r * dt$ is not small.

We can easily switch between spike-time and time-series representations of spike trains as follows:

```
# From spike density to spike times
SpikeTimes=np.nonzero(S)[0]*dt
# From spike times to spike density
S=zeros_like(time)
S[(SpikeTimes/dt).astype(int)]=1/dt
```

To generate multiple i.i.d. trials of a Poisson process (*e.g.*, to model multiple trials), we can just change the size of the generated spike density,

```
S=np.random.binomial(1, r*dt, (NumTrials,len(time)))/dt
```

Figure 2.4B shows a raster plot of 100 trials of a Poisson process with $r = 15$Hz.

Exercise 2.2.2. Generate 10 realizations of a Poisson process with rate $r = 10$Hz over a time interval of duration $T = 1$s and compute the sample Fano factor. Repeat this process 5 times with newly generated Poisson processes to see how the sample Fano factor varies over trials. Then do the same thing for 100 Poisson processes.

Poisson processes provide a rough approximation to cortical spike train statistics. For example, the ISI distributions of cortical neurons often resemble exponential distributions (except for a lack of very short ISIs), the CV of cortical neurons are often in the range of $0.75 - 1.25$ and Fano factors are also close to 1 [9–12].

One caveat to the claim that cortical spike trains are Poisson-like is that it implicitly assumes the spike trains are stationary. However, we often want to model neurons responding to time-dependent stimuli with time-dependent changes in firing rate. For example, in Figure 2.2, the firing rate of the neuron oscillated due to the periodic nature

of the drifting grating stimulus. In these cases, we should model the spike train as a non-stationary point process. A standard model for non-stationary spike trains is the **inhomogeneous Poisson process**, which is defined as follows,

> **Inhomogeneous Poisson Process Definition:** Given a non-negative function, $r(t)$, an inhomogeneous Poisson process with rate $r(t)$ is a point process that has the memoryless property and satisfies
>
> $$E[N(a,b)] = \int_a^b r(t)dt$$
>
> for any $a \leq b$.

An inhomogeneous Poisson process with a constant rate, $r(t) = r_0$, is a homogeneous Poisson process. Just like the homogeneous Poisson process, the inhomogeneous Poisson process has a Fano Factor equal to 1 over any time interval.

To generate an inhomogeneous Poisson process, we can generalize the Poisson Process Algorithm 2 to a time-dependent rate:

> **Inhomogeneous Poisson Process Algorithm:** First choose a dt small enough so that $r(t) * dt$ is very small for all t. Then set the value of each bin to $1/dt$ with probability $r(t) * dt$ and other bins to zero.

In Python, we can generate spike density representations of inhomogeneous Poisson processes in exactly the same way we do for homogeneous processes, except that `r` is a time series (it has the same size as `time`) instead of a scalar. For example, the code below generates an inhomogeneous Poisson process with a sinusoidal rate,

```
r=(20+16*np.sin(2*np.pi*time/200))/1000
S=np.random.binomial(1,r*dt,(NumTrials,len(time)))/dt
```

The resulting rate and raster plot is shown in Figure 2.4C,D. Compare to the real spike trains from Figure 2.2B. Complete code for Figure 2.4 can be found in `PoissonProcesses.ipynb`.

Now that we understand how to model spike trains with spike timing variability, let's look at neurons driven by synapses with variable presynaptic spike times.

2.3 MODELING A NEURON WITH NOISY SYNAPTIC INPUT

In Section 1.3, we modeled a postsynaptic neuron receiving input from a single excitatory and single inhibitory synapse, which was not sufficient to drive the membrane potential to threshold. We now consider a model in which a single EIF receives input from K_e excitatory and K_i inhibitory synapses (Figure 2.5A). The model is defined by the following equations:

EIF with input from several excitatory and inhibitory synapses

$$\begin{aligned}
\tau_m \frac{dV}{dt} &= -(V - E_L) + De^{(V-V_T)/D} + I_e(t) + I_i(t) \\
\tau_e \frac{dI_e}{dt} &= -I_e + \boldsymbol{J}^e \cdot \boldsymbol{S}^e \\
\tau_i \frac{dI_i}{dt} &= -I_i + \boldsymbol{J}^i \cdot \boldsymbol{S}^i \\
V(t) > V_{th} &\Rightarrow \text{spike at time } t \text{ and } V(t) \leftarrow V_{re}
\end{aligned} \tag{2.2}$$

Here, $\boldsymbol{S}^e(t)$ and $\boldsymbol{S}^i(t)$ are $K_e \times 1$ and $K_i \times 1$ vectors of spike densities. We use superscripts in place of subscripts for vectors and matrices so that we can use subscripts for indices. Specifically, $\boldsymbol{S}^e_k(t)$ is the spike density of presynaptic excitatory neuron k. Similarly, $\boldsymbol{J}^e$ and $\boldsymbol{J}^i$ are vectors of synaptic weights, so $\boldsymbol{J}^e_k$ is the synaptic weight from presynaptic excitatory neuron k. The dot product represents the sum,

$$\boldsymbol{J}^a \cdot \boldsymbol{S}^a(t) = \sum_{k=1}^{K_a} \boldsymbol{J}^a_k \boldsymbol{S}^a_k(t), \quad a = e, i$$

We generate Poisson presynaptic spike trains the same way we generated multi-trial Poisson processes,

```
Se=np.random.binomial(1,re*dt,(Ke,len(time)))/dt
```

and similarly for `Si`. The shape of `re` determines whether all the neurons have the same rates or different rates and whether the Poisson processes are homogeneous. For example, if `re` is a scalar, then all neurons will have the same rate and all spike trains will be homogeneous. To generate inhomogeneous processes, all with different rates, you'd need to use an `re` that has shape `(Ke, len(time))`. Synaptic weight vectors can also be uniform or heterogeneous. Uniform weights ($\boldsymbol{J}^e_k = j_e$) can be generated using

```
je=15.0
Je=je+np.zeros(Ke)
```

and similarly for `Ji`. Figure 2.5 shows the results with $K_e = 200$ excitatory neurons with rates $r_e = 8$Hz and $K_i = 50$ inhibitory neurons with rates $r_i = 15$Hz. Since a sum of Poisson processes is a Poisson process, only the product of K_a and r_a are important whenever $\boldsymbol{J}^a$ is uniform. For example postsynaptic neuron in Figure 2.5 is mathematically equivalent to one in a simulation with $K_e = 100$ and $r_e = 16$Hz. The Euler step to update synaptic currents is written as

```
Ie[i+1]=Ie[i]+dt*(-Ie[i]+Je@Se[:,i])/taue
```

where `@` is the NumPy symbol for a dot product or matrix multiplication. Euler step updates to `Ii` and `V` are similar. Complete code for Figure 2.5 is given in `EIFwithPoissonSynapses.ipynb`.

The membrane potential and postsynaptic spike times in Figure 2.5F are noisy and qualitatively similar to the real neuron in Figure 2.1A. Hence, Poisson presynaptic spike times provide a simple approach to modeling postsynaptic neural variability.

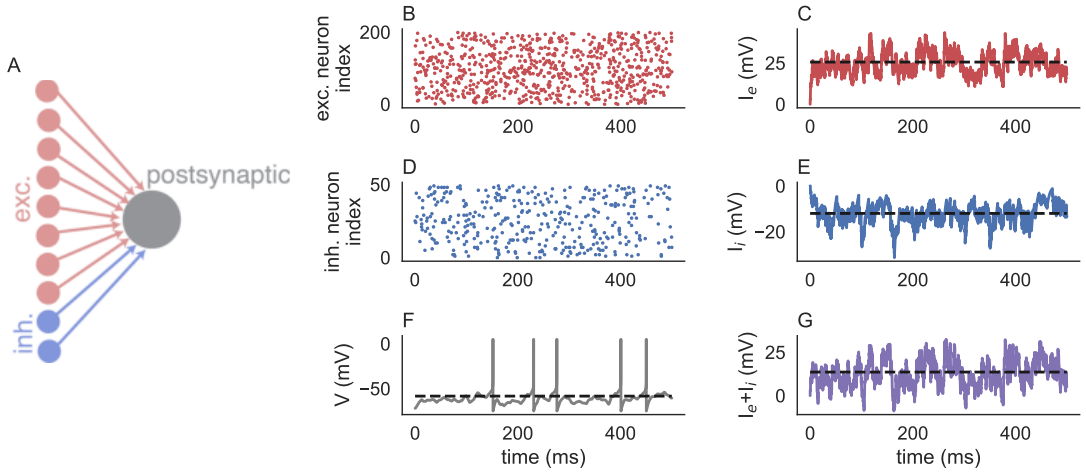

Figure 2.5: An EIF driven by synaptic input from several Poisson-spiking presynaptic neurons. A) Schematic of the model with $K_e = 8$ excitatory and $K_i = 2$ inhibitory presynaptic neurons. **B)** Raster plot of $K_e = 200$ excitatory presynaptic neurons. **B)** The resulting excitatory synaptic current. Dashed line shows the stationary mean. **C,D)** Same, but for $K_i = 50$ inhibitory presynaptic neurons. **F)** Membrane potential of the postsynaptic neuron. Dashed line shows the mean free membrane potential. **G)** Total synaptic input. See `EIFwithPoissonSynapses.ipynb` for code to produce these plots.

We can use the resulting mathematical model to better understand how the statistics of presynaptic spike trains map to the statistics of the postsynaptic spike train. First consider the expectation, $E[I_e(t)]$. This is what you would get if you simulated the model in Eq. (2.2) many times (with new random numbers each time), then averaged $I_e(t)$ over trials. To compute $E[I_e(t)]$, we can take expectations in Eq. (2.2) to get

$$\tau_e \frac{dE[I_e]}{dt} = -E[I_e] + E[\boldsymbol{J}^e \cdot \boldsymbol{S}^e] = -E[I_e] + \boldsymbol{J}^e \cdot \boldsymbol{r}^e$$

where $\boldsymbol{r}^e$ is the vector of firing rates and the first equality follows from Eq. (2.1). When $\boldsymbol{J}^e$ and $\boldsymbol{r}^e$ are uniform ($\boldsymbol{J}^e_k = j_e$ and $\boldsymbol{r}^e_k = r_e$, as in Figure 2.5), then we have

$$\boldsymbol{J}^e \cdot \boldsymbol{r}^e = K_e j_e r_e.$$

When they are not uniform the derivation below can proceed without this substitution. Repeating this analysis for $I_i(t)$, we see that mean synaptic inputs obey,

$$\begin{aligned} \tau_e \frac{dE[I_e]}{dt} &= -E[I_e] + K_e j_e r_e \\ \tau_i \frac{dE[I_i]}{dt} &= -E[I_i] + K_i j_i r_i \end{aligned} \tag{2.3}$$

These equations are valid for homogeneous and inhomogeneous presynaptic spike trains ($r_e(t)$ and $r_i(t)$ time-dependent or time-constant). When r_e and r_i are time-constant, $E[I_e(t)]$ and $E[I_i(t)]$ decay exponentially to the fixed points

$$\begin{aligned} \bar{I}_e &= \lim_{t\to\infty} E[I_e(t)] = K_e j_e r_e \\ \bar{I}_i &= \lim_{t\to\infty} E[I_i(t)] = K_i j_i r_i. \end{aligned} \tag{2.4}$$

These are called the **stationary mean values** of $I_e(t)$ and $I_i(t)$. The expectation approaches these values after a brief transient determined by the synaptic time constants, $\tau_e, \tau_i \approx 5$ms. In Figure 2.5B,D, you can see that the synaptic currents (red and blue) tend to fluctuate around their stationary mean values (black dashed) after a brief transient. Along these lines, Eq. (2.4) can be interpreted as saying that, after a transient,

$$\begin{aligned} I_e(t) &= \bar{I}_e + \text{noise} \\ I_i(t) &= \bar{I}_i + \text{noise} \end{aligned} \tag{2.5}$$

One way of interpreting these results is that, if you averaged $I_e(t)$ over many trials, then the trial-average at each t would converge to $\bar{I}_e$. Another interpretation is that, if you take a single trial of $I_e(t)$, but average it over a long time interval, then the time-average would also converge to $\bar{I}_e$. In other words,

$$\lim_{T\to\infty} \frac{1}{T} \int_0^T I_e(t)dt = \bar{I}_e.$$

Exercise 2.3.1. Compare trial-averaged currents to time-averaged currents in simulations. Simulate a synaptic current, $I_e(t)$ (you do not need to simulation $V(t)$ or $I_i(t)$) over a fixed time interval of duration $T = 100$ms. Repeat this in a for-loop over many trials and compute the trial-averaged value of $I_e(T)$. Compare the trial-average to $\bar{I}_e = K_e J_e r_e$ as the number of trials increases. Then simulate $I_e(t)$ for one trial and compute the time-average. Compare the time-average to $\bar{I}_e = K_e J_e r_e$ for increasing values of T. The time-average is more accurate if you throw away the transient from the first 25ms. This is called a burn-in period. An estimate of an expectation obtained by averaging across trials or time is called a **Monte-Carlo estimate**. When we don't have a closed form expression for an expectation, sometimes Monte-Carlo estimates are the best we can do.

This analysis of the expectations of I_e and I_i by taking expectations in Eq. (2.2) is an example of a **mean-field theory** of neural networks. Our analysis relied on linearity of the ODEs defining I_e and I_i in Eq. (2.2) (How?). The equations that define $V(t)$ in Eqs. (2.2) are not linear, so mean-field theory cannot be applied so easily. Indeed, there is no known closed form expression for the postsynaptic firing rate or stationary mean membrane potential, but some approximations can be obtained. Combining Eqs. (2.2) and (2.5) gives

$$\tau_m \frac{dV}{dt} = -(V - E_L) + De^{(V-V_T)/D} + \bar{I} + \text{noise} \tag{2.6}$$

where

$$\bar{I} = \bar{I}_e + \bar{I}_i \tag{2.7}$$

is the stationary mean input. In other words, the model in Eq. (2.2) behaves like an EIF driven by time-constant input with added noise.

Here's the interesting part: In the exercise at the end of Section 1.2, you were asked to derive the threshold input required to drive an EIF with time-constant input to spike. If you apply your result to the EIF from Figure 2.5, you should get $I_{th} = 15$mV. The neuron in Figure 2.5 clearly spikes, but $\bar{I} = 12.75$mV is sub-threshold. Without the

noise term in Eq. (2.6), the EIF wouldn't spike, but in the presence of noise, it does spike! This is known as **noise-driven** or **fluctuation-driven** spiking. The idea is that the stationary mean input drives the membrane potential toward threshold, but not over threshold. Then the membrane potential fluctuations (produced by noisy synaptic input) occasionally push the membrane potential over threshold to generate spikes [9, 11, 13]. In this way, noise can increase the firing rate of neurons.

Another way to understand noise-driven spiking is to first replace the EIF model in Eq. (2.2) with a leaky integrator,

$$\tau_m \frac{dV_0}{dt} = -(V_0 - E_L) + I_e(t) + I_i(t)$$

This V_0, which we get by ignoring active currents and spiking, is sometimes called the **free membrane potential**. This equation *is* linear, so we can apply the same mean-field approach that we used above to get

$$\tau_m \frac{dE[V_0]}{dt} = -(E[V_0] - E_L) + E[I_e] + E[I_i].$$

Therefore, the **stationary mean free membrane potential** is given by

$$\overline{V}_0 = \lim_{t\to\infty} E[V_0(t)] = E_L + \overline{I}$$

which is plotted as a dashed line in Figure 2.5F. Hence, the dynamics of the free membrane potential look like

$$V_0(t) = \overline{V}_0 + \text{noise}$$

In Figure 2.5F, the *free* membrane potential, $V_0(t)$, would just fluctuate around $\overline{V}_0 = -59.25\text{mV}$ (the dashed line). The actual membrane potential (gray curve in Figure 2.5F) also fluctuates around $\overline{V}_0$, but whenever the fluctuations drive it near $V_T = -55\text{mV}$, they recruit the nonlinear, exponential terms in the ODE for V (which are absent in the equation for V_0) and a spike can be generated. The resulting postsynaptic spike train is irregular and Poisson-like because it is driven by random fluctuations over threshold. Hence, spike timing variability in the presynaptic spike trains drives spike timing variability in the postsynaptic spike train.

If parameters are chosen differently (*e.g.*, by increasing r_e or j_e), then we can have $\overline{I} > I_{th}$, in which case the neuron is driven to spike by the mean input, *i.e.*, the neuron would spike even if we replaced the time varying input $I_e(t) + I_i(t)$ with its stationary mean, $\overline{I}$. However, noise still introduces some irregularity in the spike times. This is called **mean-driven** or **drift-driven** spiking.

Figure 2.6A,B compares the membrane potentials in fluctuation- and drift-driven regimes. The black curve in Figure 2.6C shows how the firing rate depends on the stationary mean synaptic input as r_e is increased. This curve, showing a neuron's firing rate as a function of its mean input, is called an **f-I curve**.

The exercise at the end of Section 1.2 asked you to plot an f-I curve for an EIF driven by time-constant input, $I(t) = I_0$. The f-I curve in Figure 2.6C shows the same thing, but for a neuron driven by Poisson synaptic input. The vertical dashed line shows the cutoff, I_{th}, between the fluctuation- and drift-driven regimes. An f-I curve for an EIF with time-constant input (such as the one from the exercise in Section 1.2) would be

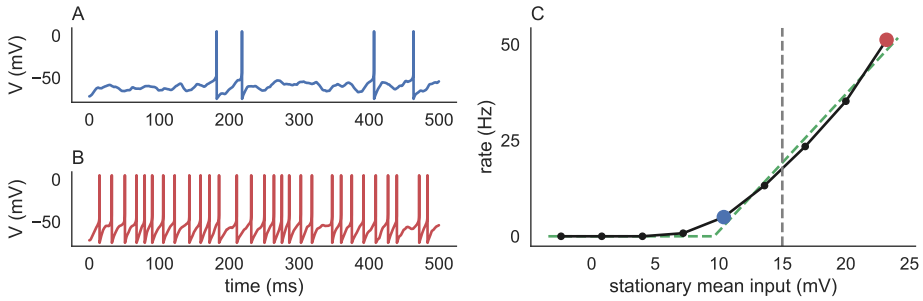

Figure 2.6: Computing an f-I curve for an EIF driven by Poisson synaptic input. A,B) Membrane potential of an EIF in fluctuation-driven and drift-driven regimes. **C)** An f-I curve for an EIF. The firing rate was plotted as a function of stationary mean synaptic input, $\overline{I}$, as r_e was varied. The blue and red dots correspond to the membrane potentials from A and B. The vertical dashed line shows the cutoff between the fluctuation-driven and drift-driven regimes (at $\overline{I} = I_{th}$). See `EIFfIcurve.ipynb` for code to produce this figure.

zero to the left of the dashed line. Hence, the positivity of the f-I curve to the left of the dashed line in Figure 2.6C demonstrates noise-driven spiking.

The f-I curve in Figure 2.6 gives the impression that the firing rate is a function of the stationary mean input, $\overline{I}$ alone, but this is not true. Two sets of parameter values that give the same value of $\overline{I}$ might produce two different firing rates. For example, if you multiply the value of j_i by 2 and multiply value of r_i by 1/2, then $\overline{I}$ does not change, but the postsynaptic firing rate will change. However, as the exercise below demonstrates, the f-I curve is *approximately* a function of $\overline{I}$ across a reasonably large range of parameters.

> **Exercise 2.3.2.** Repeat the simulations in `EIFfIcurve.ipynb`, but increase and/or decrease the values of j_e, r_i, and/or j_i by a factor of up to 2. Compare the f-I curve you get to the original dashed-green curves from Figure 2.6C by plotting them together on the same axis. You might need to change the range of r_e values to get a similar range of $\overline{I}$ values for a side-by-side comparison. Next, repeat the simulations in `EIFfIcurve.ipynb`, but generate the f-I curve by varying j_e, r_i, or j_i while keeping r_e fixed. Make the same side-by-side comparison. You should see that the f-I curve is similar in each case.

Motivated by these observations, we can make the approximation

$$r \approx f(\overline{I}).$$

Combining this with equations Eq. (2.4) and Eq. (2.7) gives

Stationary mean-field approximation for a neuron with several synaptic inputs

$$r \approx f(w_e r_e + w_i r_i) \tag{2.8}$$

where

$$w_a = K_a j_a$$

is called a **mean-field synaptic weight** which quantifies the combined strength of all synapses from presynaptic population $a = e, i$ onto the postsynaptic neuron.

Eq. (2.8) provides an **mean-field approximation** of postsynaptic firing rates. To predict the postsynaptic firing rates from a set of parameters, we only need to specify a function, f, to use as the approximate f-I curve. A simple but useful family of f-I curves are given by **rectified linear** or **threshold-linear** functions [14],

$$f(I) = (I-\theta)gH(I-\theta) = \begin{cases} (I-\theta)g & I \geq \theta \\ 0 & I < \theta \end{cases}$$

where $H(\cdot)$ is the Heaviside step function, θ is a threshold below which the firing rate is zero and g is a **gain**, which quantifies the slope or derivative of the f-I curve when $r > 0$. In Python, we can use a curve fitting function to fit θ and g to our simulated data,

```
from scipy.optimize import curve_fit
def f(IBar, g, theta):
    return g*(IBar-theta)*(IBar>theta)
params,_=curve_fit(fIfit, IBars, rs)
gfit=params[0]
thetafit=params[1]
```

The dashed green curve in Figure 2.6C shows the fit. While it's not perfect, it does provide a reasonable approximation. In the next chapter, we will use these approximations to understand the behavior of networks of neurons. Improved approximations can be achieved by using stochastic analysis to account for the effect of presynaptic spike timing variability on postsynaptic firing rates [13, 15–19], but this approach is outside the scope of this book.

Exercise 2.3.3. Repeat the simulations in `EIFfIcurve.ipynb`, but instead of plotting the postsynaptic firing rate on the vertical axis, plot the CV of the postsynaptic neuron's spike train. You'll want to increase T and restrict to parameter values in which the neuron spikes at least 30 times to make sure you get enough spikes for a decent estimate of the CV. How does the CV compare in the fluctuation-dominated versus drift-driven regimes?

3 MODELING NETWORKS OF NEURONS

A network of neurons is a group of neurons with some synaptic connections between them. We have already considered two networks in which one neuron receives synaptic input from all the other neurons (Figures 1.4 and 2.5).

Network models that represent individual spikes are called **spiking network models**. The models in Figures 1.4 and 2.5 are spiking network models because the EIF neurons spike. Sections 3.1 and 3.2 focus on spiking network models. Section 3.3 focuses on **rate network models** in which neurons' firing rates are modeled directly without representing individual spike times.

A network is called **recurrent** if you can follow arrows to get from one neuron back to itself. Otherwise, a network is called **feedforward**. Cortical neuronal networks are recurrent, but we will begin by modeling feedforward networks because they are simpler and studying them first will help us study recurrent networks next.

3.1 FEEDFORWARD SPIKING NETWORKS AND THEIR MEAN-FIELD APPROXIMATION

Feedforward networks are often arranged in **layers** where each layer (except for the first) receives synaptic input from the layer before it. We will begin by considering the simplest feedforward network with two layers (Figure 3.1A). The first layer consists of N_e excitatory and N_i inhibitory neurons, which provide synaptic input to a second layer of N neurons. We don't need to distinguish between excitatory and inhibitory neurons in the second layer because they do not provide input to any other neurons.

Figure 3.1A shows two ways to sketch the network. In the top schematic, each circle represents a neuron and arrows are individual synapses. This approach is cumbersome for large networks. The bottom schematic shows a simpler approach in which each circle is a population and arrows indicate connected populations. Note, in the bottom schematic, an arrow between two populations does not mean that *every* pair of neurons are connected, but only that *some* connections exist in the indicated direction.

The model for Figure 3.1A is identical to the model considered for Figure 2.5 from Section 2.3 except we now have several postsynaptic neurons instead of just one, and each postsynaptic neuron only receives input from a subset of the presynaptic layer. The model is defined by the equations

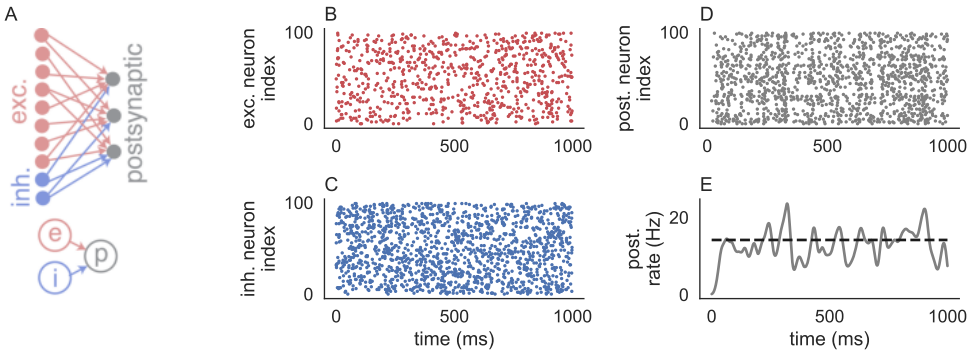

Figure 3.1: A feedforward spiking network. A) Two ways to schematicize a feedforward network. A layer of postsynaptic neurons (gray) receives synaptic input from a layer of presynaptic excitatory neurons (red) and inhibitory neurons (blue). In the top schematic, each circle is a neuron. In the bottom, each circle is a population. **B,C,D)** Raster plots of 100 excitatory, inhibitory, and postsynaptic neurons from a simulation with $N_e = 2000$ excitatory, $N_i = 500$ inhibitory, and $N = 100$ postsynaptic neurons. **E)** Estimated time-dependent firing rate of postsynaptic neurons (gray) and the mean-field approximation of the firing rates (black dashed). Code to reproduce this figure can be found in `FeedFwdNet.ipynb`.

A feedforward spiking network

$$\begin{aligned}
\tau_m \frac{d\boldsymbol{V}}{dt} &= -(\boldsymbol{V} - E_L) + De^{(\boldsymbol{V}-V_T)/D} + \boldsymbol{I}^e(t) + \boldsymbol{I}^i(t) \\
\tau_e \frac{d\boldsymbol{I}^e}{dt} &= -\boldsymbol{I}^e + J^e \boldsymbol{S}^e \\
\tau_i \frac{d\boldsymbol{I}^i}{dt} &= -\boldsymbol{I}^i + J^i \boldsymbol{S}^i \\
\boldsymbol{V}_j(t) > V_{th} &\Rightarrow \text{spike at time } t \text{ and } \boldsymbol{V}_j(t) \leftarrow V_{re}
\end{aligned} \tag{3.1}$$

This is similar to Eqs. (2.2) in Section 2.3 except the equations now have a slightly different interpretation. Specifically, $\boldsymbol{V}(t)$ now represents an $N \times 1$ vector of membrane potentials. Also, J^e is now a $N \times N_e$ matrix of synaptic weights and J^i is an $N \times N_i$ matrix. These are called **connectivity matrices** or **weight matrices**. The entry, J^e_{jk}, represents the synaptic weight from excitatory neuron $k = 1, \ldots N_e$ to postsynaptic neuron $j = 1, \ldots, N$ and similarly for the $N \times N_i$ weight matrix, J^i. Note that, counter to intuition, the first index in a weight matrix refers to the postsynaptic neuron and the second index refers to the presynaptic neuron (J^e_{jk} is "from k to j" not "from j to k"). While this seems backwards, it is necessary for the matrix multiplication to make sense. Specifically, the jth element of the matrix products are expanded as

$$[J^b \boldsymbol{S}^b]_j = \sum_{k=1}^{N_b} J^b_{jk} \boldsymbol{S}^b_k$$

for $b = e, i$ and $j = 1, \ldots, N$. We again model presynaptic spike trains, $\boldsymbol{S}^e_k(t)$ and $\boldsymbol{S}^i_k(t)$, as Poisson processes.

In the cortex, most neurons are not synaptically connected, even if they are nearby, and connectivity appears to be partly random. A simple model of this randomness is provided by a random connection matrix defined by

$$J^b_{jk} = \begin{cases} j_b & \text{with probability } p_b \\ 0 & \text{otherwise} \end{cases}$$

where p_b is the connection probability and j_b is the synaptic weight for neurons from presynaptic population $b = e, i$. These connection matrices can be generated as

```
Je=je*np.random.binomial(1,pe,(N,Ne))
```

and similarly for `Ji`. As in Section 2.3, each presynaptic spike train, $\boldsymbol{S}^e_j(t)$ and $\boldsymbol{S}^i_j(t)$, can be modeled as a Poisson process with rates r_e and r_i.

The code to simulate this network model is similar to the code in `EIFwithPoissonSynapses.ipynb` from Section 2.3 except for a few differences. The terms `V`, `Ie`, or `Ii` need to be initialized as arrays, *e.g.*,

```
Ie=np.zeros((N,len(time)))
```

and updated like

```
Ie[:,i+1]=Ie[:,i]+dt*(-Ie[:,i]+Je@Se[:,i])/taue
```

See `FeedFwdNet.ipynb` for complete code.

Each individual postsynaptic neuron behaves just like the single postsynaptic neuron modeled in Section 2.3 except the number of excitatory and inhibitory synapses, K_e and K_i, received by each neuron is random instead of fixed. The *expected* number of excitatory and inhibitory synaptic inputs received by each postsynaptic neuron are given by $E[K_e] = p_e N_e$ and $E[K_i] = p_i N_i$. Therefore, the stationary mean-field synaptic inputs are now given by

$$\bar{I}_e = N_e p_e j_e r_e$$
$$\bar{I}_i = N_i p_i j_i r_i$$

Mathematically speaking,

$$\bar{I}_a = \lim_{t\to\infty} E[I^a_j(t)]$$

where the expectation is taken over randomness in $S^a(t)$ *and* randomness in J_a.

Exercise 3.1.1. In an exercise in Section 2.3, we pointed out that that $\bar{I}_e$ could be estimated by averaging over trials or over time. This is no longer true because randomness in J^e cannot be averaged out by a time-average. This type of randomness is called **quenched randomness**. Instead, randomness in J^e can be averaged over postsynaptic neurons. Specifically,

$$\lim_{T,N\to\infty} \frac{1}{N}\sum_{j=1}^{N} \frac{1}{T}\int_0^T I^e_j(t)dt = \bar{I}_e$$

From a more practical perspective, an accurate estimate of the mean can be obtained by averaging over postsynaptic neurons *and* over time. Try averaging over both in a simulation and compare the averages to the exact value, $\bar{I}_e = N_a p_a j_a r_e$, for increasing values of N and/or T.

As in Section 2.3, we cannot derive an exact mean-field theory of postsynaptic firing rates, but we can again use an approximate f-I curve, $r \approx f(\bar{I})$ where

$$\bar{I} = \bar{I}_e + \bar{I}_i = N_e p_e j_e r_e + N_i p_i j_i r_i$$

Putting this together gives a mapping from presynaptic to postsynaptic rates,

Stationary mean-field approximation for a feedforward network

$$r \approx f(w_e r_e + w_i r_i). \tag{3.2}$$

where

$$w_a = N_a p_a j_a$$

is the mean-field synaptic weight for this network model. Eq. (3.2) is identical to Eq. (2.8) except that w_e and w_i are defined using the expected number of inputs, $E[K_a] = N_a p_a$, in place of the deterministic number of inputs, K_a, used in Eq. (2.8). More generally, mean-field synaptic weights can be defined by

$$w_a = N_a E[J^a_{jk}].$$

Figure 3.1E shows the mean-field approximation from Eq. (3.2) (dashed line) compared to the mean rate estimated from the full simulation (gray curve). This comparison shows that the relatively simple Eq. (3.2) does a decent job of describing firing rates without needing to simulate an entire network (although note that we needed to simulate the network to fit the f-I curve initially).

Here, we restricted ourselves to a network with just one presynaptic population and two postsynaptic populations, but this approach can easily be extended to feedforward networks with several postsynaptic populations, giving rise to mean-field equations of the form

Feedforward rate network model.

$$\boldsymbol{r} = f(W_x \boldsymbol{r}_x). \tag{3.3}$$

where $\boldsymbol{r}$ and $\boldsymbol{r}_x$ are vectors of the pre- and post-synaptic populations' mean firing rates. We replaced the $\approx$ with $=$ because equations like Eq. (3.3) are sometimes used as models in themselves instead of just being used to approximate mean firing rates in spiking network models. In these situations, they are sometimes called **feedforward rate network models** or just **rate models** or **rate networks**. In Section 4.3, we will see how Eq. (3.3) can be used to build artificial neural networks for machine learning. We also restricted ourselves above to networks with a single presynaptic layer and a single postsynaptic layer. Adding more layers to equations of the form Eq. (3.3) gives us multi-layered (or "deep") artificial neural networks, which are some of the most powerful algorithms for machine learning. These ideas are discussed further in Section 4.3.

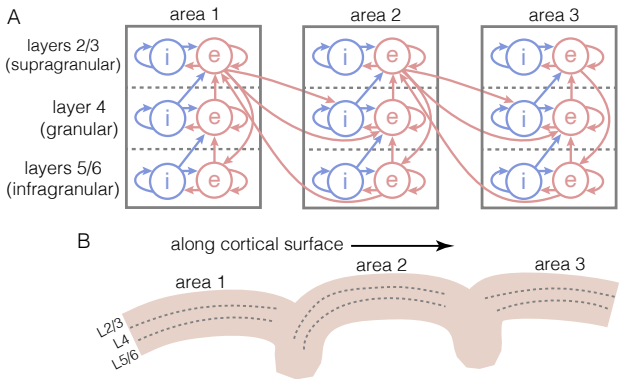

Figure 3.2: Diagram of a simplified cortical circuit model. A) Cortex is layered in two senses. Cortical areas form layered hierarchies, and there is a stereotyped architecture of layers within each area. This diagram, adapted from the "canonical" architecture described in [20], shows some of the dominant connectivity pathways between cortical areas and layers. **B)** Viewing the cortex as a wrinkled sheet, cortical areas are arranged along the surface of the sheet and layers are arranged along its depth.

Multilayered networks mimic the layered architecture of the cortex. The cortex is layered in two separate senses:

First, some **cortical areas** are arranged in a layered, hierarchical fashion (Figure 3.2A). A well known cortical hierarchy is the ventral stream of the visual cortex. Visual stimuli from the retina pass through the thalamus to the primary visual cortex (V1) which responds to simple visual features like the orientation (angle) of edges. V1 transmits its responses to V2, and so on to higher visual cortical areas that respond to increasingly complex visual features like shapes and faces. Connections between cortical areas are primarily excitatory. If we visualize the cortex as a folded up sheet, cortical areas are located in different regions along the surface of the sheet (Figure 3.2B).

Secondly, each cortical area is composed of several **cortical layers** and these layers are connected with some stereotyped motifs (Figure 3.2A). For example, cortical layer 4 typically receives input from lower cortical areas, then sends synaptic projections to layers 2/3. Cortical layers are arranged along the depth of the cortical sheet (Figure 3.2B).

Despite its layered architecture, the cortex is by no means feedforward, as you can see in Figure 3.2A. First, connections between cortical areas along a hierarchy exist in both directions: feedforward and feedback (*e.g.*, V2 projects to V1). Secondly, nearby neurons within the same cortical area and cortical layer are interconnected with each other, forming "local" recurrent networks. For these reasons, we next consider recurrent network models.

3.2 RECURRENT SPIKING NETWORKS AND THEIR MEAN-FIELD APPROXIMATION

Despite the complexity of Figure 3.2A, it is still a gross simplification of a real cortical circuit. Among other factors, there are many connectivity pathways not pictured in the diagram. Moreover, within a single layer and area, there are numerous subtypes of inhibitory neurons and connectivity depends on the neurons' subtype and the distance

between neurons. It would be an enormous task to account for most of these details in a single model, so we should start with a much simpler model.

In this section, we develop a simplified model of a small, local patch of interconnected excitatory and inhibitory neurons within a single area and layer. The model is sketched in Figure 3.3A. The excitatory and inhibitory neurons connect to each other and also receive synaptic input from an external population, x, representing synaptic input from different cortical layers or areas. Since connections between cortical areas are primarily excitatory, we will assume that x is an excitatory presynaptic population.

The spiking model for this network is defined by the equations

A recurrent spiking network model

$$\begin{aligned}
\tau_m \frac{d\boldsymbol{V}^e}{dt} &= -(\boldsymbol{V}^e - E_L) + De^{(\boldsymbol{V}^e - V_T)/D} + \boldsymbol{I}^{ee}(t) + \boldsymbol{I}^{ei}(t) + \boldsymbol{I}^{ex}(t) \\
\tau_m \frac{d\boldsymbol{V}^i}{dt} &= -(\boldsymbol{V}^i - E_L) + De^{(\boldsymbol{V}^i - V_T)/D} + \boldsymbol{I}^{ie}(t) + \boldsymbol{I}^{ii}(t) + \boldsymbol{I}^{ix}(t) \\
\tau_b \frac{d\boldsymbol{I}^{ab}}{dt} &= -\boldsymbol{I}^{ab} + J^{ab}\boldsymbol{S}^b, \quad a = e,i, \quad b = e,i,x \\
\boldsymbol{V}_j^a(t) > V_{th} &\Rightarrow \text{spike at time } t \text{ and } \boldsymbol{V}_j^a(t) \leftarrow V_{re}, \quad a = e,i
\end{aligned} \tag{3.4}$$

The first equation describes the $N_e \times 1$ vector of excitatory neurons' membrane potentials, the second equation is the same for inhibitory neurons, and the last equation defines their threshold-reset conditions. The term $\boldsymbol{I}^{ab}(t)$ is the synaptic input from population b to population a. For example, $\boldsymbol{I}^{ei}(t)$ is the $N_e \times 1$ vector of total inhibitory input to excitatory neurons. The connection matrix, J^{ab}, is an $N_a \times N_b$ matrix of synaptic weights. Spikes in the external population are generated as Poisson processes, each with firing rate r_x. As above, connection matrices are random,

$$J_{jk}^{ab} = \begin{cases} j_{ab} & \text{with probability } p_{ab} \\ 0 & \text{otherwise.} \end{cases}$$

Simulating large networks can be computationally expensive in terms of runtime and memory. One source of inefficiency is the use of large arrays to represent time-varying vectors. For example, the excitatory membrane potential, $\boldsymbol{V}^e(t)$, can be stored as a $N_e \times N_t$ vector where $N_t = T/dt$ is the number of time bins. For example, the update to $I^{ee}(t)$ in an Euler step might look like

```
Iee[:,i+1]=Iee[:,i]+dt*(-Iee[:,i])/taue
```

as in the `FeedFwdNet.ipynb` code used generate Figure 3.1. However, if we don't need to keep track of the history of `Iee` then we can store it as a $N_e \times 1$ vector. In this case, an Euler step looks like

```
Iee=Iee+dt*(-Iee)/taue
```

The new value of `Iee` overwrites the old value. We can of course do the same for all synaptic currents, $\boldsymbol{I}^{ab}(t)$, and membrane potentials, $\boldsymbol{V}^a(t)$. The downside to this

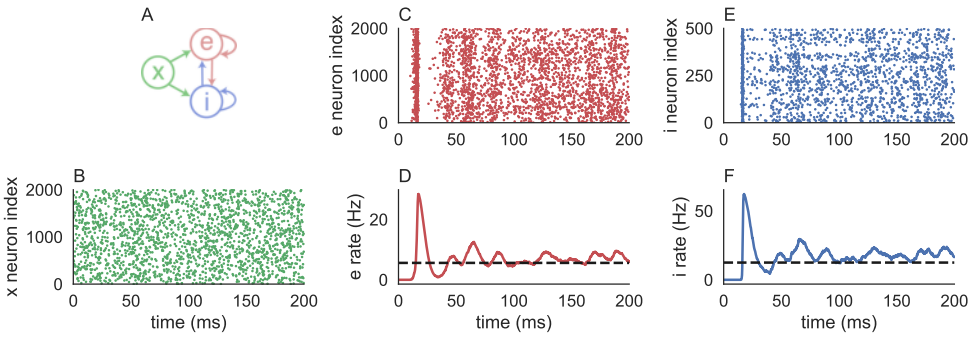

Figure 3.3: Simulation of a recurrent spiking network. A) A schematic of the network. An external excitatory population sends synaptic input to excitatory and inhibitory populations, which are recurrently connected. **B)** Raster plot of the external spike trains, which are modeled as Poisson processes. **C)** Raster plot of the excitatory spike trains, which are modeled using EIF neuron models. **D)** Smoothed estimate of the population-averaged, time-dependent firing rate of the excitatory population. Dashed black line shows mean-field approximation to the firing rates. **E,F)** Same as C,D, but for the inhibitory population. Code to reproduce this figure can be found in `RecurrentSpikingNet.ipynb`.

approach is that you cannot generate plots or statistics of the membrane potentials or synaptic currents, but this is not a problem if we're only interested in analyzing spike trains, which is often the case.

We generally want to keep track of the spike trains across time, so we would not use the same approach to store the spike densities $\boldsymbol{S}^e(t)$, $\boldsymbol{S}^i(t)$, and $\boldsymbol{S}^x(t)$. However, we can gain a lot of efficiency by storing the spike trains as lists of spike times instead of spike densities. To generate the external spike trains as Poisson processes, we can do

```
nsx=np.random.poisson(rx*T*Nx) # Number of spikes
SxTimes=np.sort(np.random.rand(nsx)*T) # Random spike times
SxIndices=np.random.randint(Nx,size=nsx) # Random neuron Indices
```

Here, `nsx` is the total spike count across all external neurons. Once the spike count is set, the spike times are uniformly distributed on the interval $[0, T]$, and the associated neuron indices are uniformly distributed on the integers from 0 to $N_x - 1$. It's a bit tricky to update the external synaptic currents using this representation of the external spike trains. On iteration `i` through the time loop, we can do

```
    while iX<len(SxTimes) and SxTimes[iX]<i*dt:
        Iex=Iex+Jex[:,SxIndices[iX]]/taux
        Iix=Iix+Jix[:,SxIndices[iX]]/taux
        iX+=1
```

where `iX` is initialized to zero outside the time loop. This code finds all spike times in `SxTimes` that occur before the current time ($t = i * dt$) and increments the synaptic currents using the associated column of the weight matrices. Spike times must be sorted for this approach to work.

Representing the spike trains, $\boldsymbol{S}^e(t)$ and $\boldsymbol{S}^i(t)$, using the same convention is even trickier because the spike times are generated during the simulation, not before it. We can initialize them to store a maximum number of spikes. For example, if we expect the average excitatory firing rate to be less than 50Hz, we can do

```
eMaxNumSpikes=int(Ne*T*50/1000)
SeTimes=np.zeros(eMaxNumSpikes)
SeIndices=np.zeros(eMaxNumSpikes)
```

Inside the time loop, we have

```
    Inds=np.nonzero(Ve>=Vth)[0] # Find which neurons spiked
    Ve[Inds]=Vre # Reset membrane potentials
    if eNumSpikes<eMaxNumSpikes and len(Inds)>0:
        SeTimes[eNumSpikes:eNumSpikes+len(Inds)]=i*dt # Store times
        SeIndices[eNumSpikes:eNumSpikes+len(Inds)]=Inds # and indices
    Iee=Iee+np.sum(Jee[:,Inds],axis=1)/taue # Update synaptic currents
    Iie=Iie+np.sum(Jie[:,Inds],axis=1)/taue
    eNumSpikes+=len(Inds) # Update spike count
```

where `eNumSpikes` is initialized to zero outside the time loop. Complete code to simulate a recurrent spiking network can be found in `RecurrentSpikingNet.ipynb`

We could also increase the efficiency of the code by taking advantage of the sparsity of the connection matrices, J_{ab}, but this is only very beneficial for networks with more than around $100,000$ neurons, so we won't bother.

Altogether, the code for simulating a recurrent spiking network is pretty long and complicated. An alternative to coding a network simulation from scratch is to use a **neural simulation software** package that implements all of the details for you. A popular neural simulation software package is the BRIAN Simulator [21], which lets you efficiently simulate a recurrent spiking network in a few lines of code and makes it easy to easily switch neuron models and other network properties. The basics of using BRIAN are reviewed in Appendix B.6.

See Appendix B an introduction BRIAN neural simulation soft

Figure 3.3 shows spike trains from a simulation with $N_x = N_e = 2000$ and $N_i = 500$. Early in the simulation, firing rates increase quickly as many neurons spike at nearly the same time (Figure 3.3C–F). This occurs because the initial conditions of the membrane potentials and synaptic currents cause many neurons to cross threshold at nearly the same time. After a transient oscillation, the firing rates settle down and fluctuate around a steady-state value.

To describe the steady-state, we can adapt the mean-field theory we developed for feedforward networks. The mean-field synaptic inputs to excitatory and inhibitory neurons are given by

$$\begin{aligned} \bar{I}_e &= w_{ee} r_e + w_{ei} r_i + w_{ex} r_x \\ \bar{I}_i &= w_{ie} r_e + w_{ii} r_i + w_{ix} r_x \end{aligned} \tag{3.5}$$

where

$$w_{ab} = N_b E[J_{jk}^{ab}] = N_b p_{ab} j_{ab}$$

is the mean-field synaptic weight and r_b is the firing rate. Eq. (3.5) can be written in matrix form as

$$\overline{\boldsymbol{I}} = W\boldsymbol{r} + W_x r_x \tag{3.6}$$

where

$$\overline{\boldsymbol{I}} = \begin{bmatrix} \overline{I}_e \\ \overline{I}_i \end{bmatrix}, \quad \boldsymbol{r} = \begin{bmatrix} r_e \\ r_i \end{bmatrix}, \quad W = \begin{bmatrix} w_{ee} & w_{ei} \\ w_{ie} & w_{ii} \end{bmatrix}, \quad \text{and} \quad W_x = \begin{bmatrix} w_{ex} \\ w_{ix} \end{bmatrix}. \tag{3.7}$$

Eq. (3.6) would be an exact equation under the assumption that r_e and r_i are the stationary mean firing rates. However, we do not know the values of r_e and r_i since they are determined by the complicated dynamics of the network. To approximate them, we can use the mean-field approximation, $\boldsymbol{r} \approx f(\overline{\boldsymbol{I}})$, from Section 3.1 to get

Stationary mean-field approximation for a recurrent network

$$\boldsymbol{r} \approx f(W\boldsymbol{r} + W_x r_x). \tag{3.8}$$

Strictly speaking, Eq. (3.7) approximates stationary firing rates: If $r_x(t) = r_x$ is fixed in time then Eq. (3.7) approximates the stationary mean value of postsynaptic rates. However, it can also be used to approximate instantaneous rates. As long as $r_x(t)$ changes more slowly than the intrinsic dynamics of synapses and neurons (determined in part by τ_e, τ_i, and τ_m) then Eq. (3.8) can be used to approximate $\boldsymbol{r}(t)$ by plugging in $r_x(t)$ directly.

In contrast to Eq. (3.2), $\boldsymbol{r}$ appears on both sides of Eq. (3.8). This is called an **implicit equation** because $\boldsymbol{r}$ is defined implicitly as a solution. Recurrent networks produce *implicit* mean-field equations and feedforward networks produce *explicit* mean-field equations. In general, Eq. (3.8) can have one solution, many solutions, or no solutions.

If we use the threshold-linear f-I curve, $f(I) = (I-\theta)gH(I-\theta)$ from Section 2.3, we can look for solutions in which $r_e, r_i > 0$. Any such solution satisfies the linear equation

$$\boldsymbol{r} \approx (W\boldsymbol{r} + W_x r_x - \theta)g \tag{3.9}$$

which has the explicit solution,

$$\boldsymbol{r} \approx [I - gW]^{-1}(W_x r_x - \theta)g \tag{3.10}$$

where I is the 2×2 identity matrix. This solution is plotted as dashed lines in Figure 3.3D,F. Despite the complexity of the spiking network model and the strong simplifying assumption of a threshold-linear f-I curve, the solution provides a reasonable approximation.

ppendix B.9 for a scription of how 3.8) can be used del suppression nd competition.

Like Eq. (3.2), Eq. (3.8) can also be used as a model in itself and it can be extended to incorporate several populations. Appendix B.9 describes how Eq. (3.8) can be used to model suppression and competition, which is are widely observed phenomena in which increased activity in one sub-population of excitatory neurons tends to suppress activity in other excitatory sub-populations.

3.3 DYNAMICAL RATE NETWORK MODELS

The spiking network models we considered above are relatively complicated with a large number of parameters. In many cases, we are only interested in understanding firing rates in a network. In such cases, the mean-field approximations from Eqs. (3.2) and (3.8) provide a simpler approach. Instead of using them to approximate firing rates in spiking networks, we could interpret them as a models in themselves. This would bypass the need for a spiking network model altogether. However, these equations do not describe intrinsic dynamics generated by a network. Cortical circuits exhibit rich dynamics like oscillations, some of which are generated intrinsically (as opposed to being driven completely by external synaptic input). **Dynamical rate network models** or **rate models** or **rate networks** provide a way to model intrinsic rate dynamics without the complexity of a spiking network.

Dynamical rate networks can be derived as an approximation to spiking networks, with different approaches leading to different formulations of rate network models. We relegate those details to Appendix B.7 and focus on a single type of model here,

See Appendix B for alternative formulations of rate network models and their derivations

Recurrent dynamical rate network model

$$\boldsymbol{\tau} \circ \frac{d\boldsymbol{r}}{dr} = -\boldsymbol{r} + f(W\boldsymbol{r} + \boldsymbol{X}) \tag{3.11}$$

This equation models the dynamics of M firing rates, stored in the $M \times 1$ vector, $\boldsymbol{r}$. The term $\boldsymbol{\tau}$ is an $M \times 1$ vector of time constants that controls how quickly each firing rate changes in response to changes to the associated neurons' inputs. The $\circ$ represents an element-wise product between two vectors, also called the Hadamard product. If $\boldsymbol{z} = \boldsymbol{x} \circ \boldsymbol{y}$ then $\boldsymbol{z}_k = \boldsymbol{x}_k \boldsymbol{y}_k$. In other words, $\circ$ performs the same operation as $*$ in NumPy. In Eq. (3.11), this means that each term in the vector can have its own time constant. If a single time constant is chosen, then we can use a scalar τ and omit the $\circ$. The $M \times 1$ vector $\boldsymbol{X}$ models external input to the network, which can be time-varying or time-constant, W is an $M \times M$ recurrent connectivity matrix, and $f : \mathbb{R} \to \mathbb{R}$ is an f-I curve, which is applied pointwise as usual.

Each element of $\boldsymbol{r}$ can be interpreted as the mean firing rate of a population of neurons, or each element can be interpreted as an individual neuron. In other words, Eq. (3.11) can model a network of M interconnected neural populations, or a network of M interconnected neurons.

When Eq. (3.11) is interpreted as a model of M populations, Eq. (3.11) is sometimes called a **dynamical mean-field equation** because it generalizes the *static* mean-field equation in Eq. (3.8) to account for intrinsic dynamics. This connection between Eq. (3.11) and Eq. (3.8) can be made more direct by taking $\boldsymbol{X} = W_x r_x$. In this case, when $\boldsymbol{X}(t) = \boldsymbol{X}$ is constant, any fixed point of Eq. (3.11) is given by Eq. (3.8).

See Appendix A for a review of fixed points and stability of systems of ODEs

An alternative formulation of rate networks uses a system of ODEs for the synaptic inputs instead of the rates,

$$\boldsymbol{\tau} \circ \frac{d\boldsymbol{I}}{dt} = -\boldsymbol{I} + Wf(\boldsymbol{I} + \boldsymbol{X}). \tag{3.12}$$

Then firing rates are given by $\boldsymbol{r} = f(\boldsymbol{I} + \boldsymbol{X})$. Formulations like Eq. (3.12) are more common in some sub-fields. The relationship between Eq. (3.12) and Eq. (3.11) is discussed in Appendix B.7, but we will focus on the formulation in Eq. (3.12).

If we use Eq. (3.11) to model the excitatory-inhibitory network considered in Section 3.2, we would take $M = 2$ and it is nicer to write the system in the form

Excitatory-Inhibitory rate network model

$$\begin{aligned} \tau_e \frac{dr_e}{dt} &= -r_e + f_e(w_{ee}r_e + w_{ei}r_i + X_e) \\ \tau_i \frac{dr_i}{dt} &= -r_i + f_e(w_{ie}r_e + w_{ii}r_i + X_i) \end{aligned} \tag{3.13}$$

The model in Eq. (3.13) is often called a **Wilson-Cowan style model**. This name comes from a similar set of equations, often called the "Wilson-Cowan equations," that were proposed by Hugh Wilson and Jack Cowan for modeling interacting excitatory and inhibitory populations [22, 23].

To match the fixed points from the model considered in Section 3.2, we would take $X_a = w_{ax}r_x$ in Eq. (3.13). We might be tempted to choose τ_e and τ_i to match the synaptic time constants used in spiking networks, but this is not necessarily correct because in Eq. (3.13) they represent the combined timescales of synaptic dynamics and neuronal dynamics (see Appendix B.7 for more details). Generally speaking, time constants for dynamical rate models should be in the range of 5-50ms and should be larger for populations with slower synapses. Note that Eq. (3.13) is a special case of Eq. (3.11).

We established that dynamical rate models have fixed points given by the mean-field approximation in Eq . (3.8), but what about the stability of the fixed points? The Jacobian matrix for Eq. (3.11) is given by

$$J = \frac{1}{\boldsymbol{\tau}} \circ [-I + GW]$$

where I is the identity matrix, and G is a diagonal matrix with entries $G_{kk} = g_k = f'(\boldsymbol{I}_k)$ defined as the derivative of the f-I curve at the fixed point (here $\boldsymbol{I} = W\boldsymbol{r} + \boldsymbol{X}$ is the input at the fixed point). The term g_k is called the **gain** of rate k and quantifies the sensitivity of the firing rate to input perturbations. If all the eigenvalues of J have negative real part at a particular fixed point, then that fixed points is stable. If any eigenvalues have positive real part, it's unstable.

Figure 3.4A,B shows a simulation of an excitatory-inhibitory rate network (Eq. (3.11) or (3.13)) using parameters derived from the mean-field theory developed in the previous section and choosing $\tau_e = 30$ms, $\tau_i = 15$ms. Checking the eigenvalues shows that the fixed point is stable, which is confirmed numerically by the convergence of the firing rates toward the fixed points (dashed lines).

As expected, the firing rates from from the spiking network in Figure 3.3 converge to a similar steady state to the firing rates in the rate network in Figure 3.4A,B (note that the dashed lines are the same). However, the dynamics during the relaxation to the steady-state rates is very different. Indeed, rate network models do not typically do a good job of approximating dynamics of spiking network models, even when they're tuned to predict steady-state rates accurately. This might at first seem damning for rate models, but recall that spiking network models are already a gross simplification of real neural circuits. Just because the spiking model is more detailed does not mean that it is necessarily more accurate.

When using simplified models with so many parameters chosen nearly arbitrarily, it is not practical to seek quantitatively accurate results. In other words, we shouldn't expect

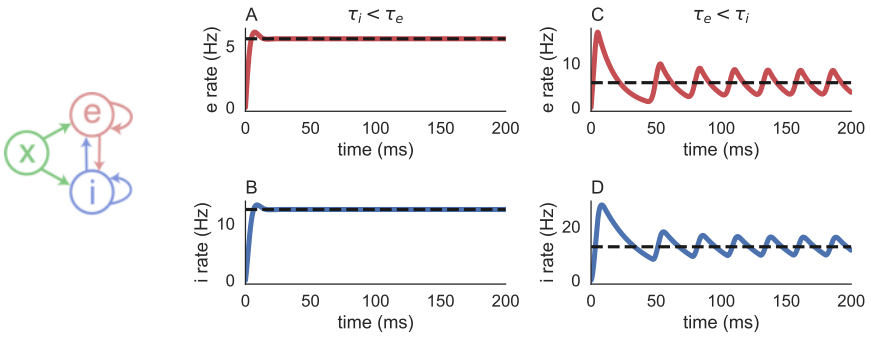

Figure 3.4: Simulation of a recurrent dynamical rate network model. A,B) Excitatory (red) and inhibitory (blue) firing rates from a simulation with $\tau_e = 30$ms and $\tau_i = 15$ms. Rates quickly approach their fixed point (dashed lines), indicating stability. **C,D)** Same simulation with $\tau_e = 15$ms and $\tau_i = 30$ms. Rates oscillate around their fixed points, indicating a Hopf bifurcation. Parameters were chosen to match the mean-field theory from Figure 3.3, so the dashed lines are in the same place. Code to reproduce this figure can be found in `RecurrentRateNet.ipynb`.

our model to tell us whether firing rates in a true neural circuit will be closer to 10Hz or 15Hz. Instead, simplified models should provide insight into general principles like the qualitative dependence of rates and dynamics on various parameters. To demonstrate this, we next use the rate model to understand the emergence of oscillations in excitatory-inhibitory networks.

Oscillations in dynamical systems can emerge through a Hopf bifurcation where a pair of complex eigenvalues changes from having negative to positive real part (see Appendix A.9 for background on Hopf bifurcations). In two-dimensional systems like Eq. (3.13), this happens when the trace of the Jacobian matrix changes from negative to positive. For Eq. (3.13), the trace is given by

$$Tr(J) = \frac{g_e w_{ee} - 1}{\tau_e} + \frac{g_i w_{ii} - 1}{\tau_i} \tag{3.14}$$

and recall that stability requires $Tr(J) < 0$. Note that $w_{ii} < 0$, so the only positive contribution to the trace is the $g_e w_{ee}$ term.

When $g_e w_{ee} < 1$, the trace is always negative and we don't need to worry about it (although we still need the determinant to be negative for stability). Excitatory-inhibitory networks with

See Appendix E more details on inhibitory stabi networks and b networks.

$$g_e w_{ee} > 1$$

are called **inhibitory-stabilized networks** because the network would be unstable without an inhibitory population (check this for yourself), so inhibition is required to stabilize the network [24, 25]. Cortical neurons can receive a large number of synaptic inputs, including the local excitation modeled by w_{ee}, so it is commonly assumed that coupling is at least strong enough to satisfy $g_e w_{ee} > 1$. See Appendix B.8 for more details on inhibitory stabilized networks, strong coupling, and their implications.

So let's assume that the network is inhibitory stabilized, $g_e w_{ee} > 1$, which is the case for the network in Figure 3.4A,B. Then the inhibitory term in Eq. (3.14) needs to be large enough in magnitude to cancel the excitatory term,

$$\left|\frac{g_i w_{ii} - 1}{\tau_i}\right| > \left|\frac{g_e w_{ee} - 1}{\tau_e}\right|.$$

Let's focus on the impact of the time constants here. Stability is encouraged by fast inhibition and comparatively slower excitation ($\tau_i < \tau_e$). If inhibition becomes too slow compared to excitation (τ_i too large compared to τ_e), the trace will become positive, producing oscillations through a Hopf bifurcation. Using τ_e and τ_i to control the trace is especially nice (from a mathematical perspective) because the sign of the determinant does not depend on τ_e and τ_i, so we don't have to worry about the effects of changing τ_e and τ_i on the determinant.

In summary, if the excitatory-inhibitory model in Eq. (3.13) is in the inhibitory-stabilized regime, sufficiently slow inhibition or fast excitation will give rise to oscillations. Indeed, Figure 3.4C,D demonstrates oscillations that arise when we swap the values of τ_e and τ_i.

Mechanistically, the oscillations can be understood as follows: Recurrent excitation in the network is strong enough to produce a runaway positive feedback loop, but inhibition helps shut down this loop before it can explode. These are defining properties of an inhibitory stabilized network. If inhibition is fast enough or excitation slow enough, inhibition shuts down the runaway excitation before it can grow at all. This is the stable condition. If inhibition is too slow or excitation is too fast, the runaway excitation starts taking off before inhibition can shut it down, then process starts over again, producing an oscillation. These types of oscillations are sometimes called PING oscillations because they arise from the interaction between Pyramidal (excitatory) and INhibitory neurons (or INterneurons) and they are believed to be a source of fast (Gamma) oscillations in the brain [26–29].

We have used a highly simplified model to reach a very general conclusion: When recurrent inhibition is too slow compared to recurrent excitation, oscillations emerge. In the exercise below, you can test this conclusion in a spiking network model in which we cannot perform stability analysis directly.

> **Exercise 3.3.1.** Let's test whether slow inhibition or fast excitation produce oscillations in a spiking network. Repeat the simulation from Figure 3.4, but switch the values of τ_e and τ_i by setting $\tau_e = 4$ms and $\tau_i = 6$ms. You should see strong oscillations emerge.
> Interestingly, in real cortical circuits, inhibition is a little bit slower than the fastest form of excitation (AMPAergic synapses). But computational models in the literature (including many of my own papers [30–33]) often take inhibition to be slower than excitation to avoid strong oscillations. This raises the question of how cortex manages to avoid strong oscillations. I don't know the answer to this question.

4 MODELING PLASTICITY AND LEARNING

All of the models considered so far in the book don't really *do* anything in the sense that they don't learn or solve any problems. The primary purpose of the brain is presumably to *do* things, or at least to tell the body to do things. Hence, all of the models considered so far arguably ignore the central purpose of the brain. This does not necessarily mean that the models are useless. They can be useful for understanding and interpreting recorded neural data, and they can be viewed as building blocks from which we can build models that actually "do something."

That said, there is an argument to be made that we should focus on models that can do things. If we were modeling an electric motor, we would presumably want to use a model that actually rotates a rotor. Likewise, if we are modeling the brain, perhaps we should use models that actually learn some task, even a simple one. In this chapter, we start building models that can learn and perform simple tasks. To begin with, we need to understand how to model synaptic plasticity, which is a primary mechanism of learning in the brain.

4.1 SYNAPTIC PLASTICITY

In neural circuits, the strength of synaptic connections changes slowly over time, an effect known as **synaptic plasticity**. An increase in synaptic strength is called **facilitation** and a decrease is called **depression**. Synapses can change strength transiently, for a duration of milliseconds or seconds, which is called **short term plasticity**. We will not discuss short term plasticity in this book, but instead focus on **long term plasticity** in which changes to synaptic strengths are static. Long term plasticity is widely believed to be the primary mechanism behind learning and memory in the brain, but the precise mechanics of how learning and memory emerge from plasticity are not fully understood.

We will not go into the biophysical details of why synapses change strength, but it is partially caused by an influx of Calcium into a neuron after an action potential. As such, changes to the strength of a synapse can depend on the spike times and firing rates of the pre- and post-synaptic neurons.

The most well-known type of plasticity is **Hebbian plasticity** (named after Donald Hebb) in which the increase in synaptic strength is proportional to the firing rates of pre- and post-synaptic neurons [34]. Hebbian plasticity is often described using the idiom

> "Neurons that fire together wire together."

In other words, if two neurons tend to spike nearby in time, or increase their firing rates at the same time, then their synaptic coupling gets stronger. In Appendix B.10, we describe how a form of Hebbian plasticity can give rise to assembly formation and associative memory in **Hopfield network models**.

See Appendix B a description of Hopfield netwo which Hebbian plasticity imple associative mem

For a the dynamical rate network model from Eq. (3.11), a simple form of pure Hebbian plasticity can be defined by

$$\frac{dW_{jk}}{dt} = c\boldsymbol{r}_j\boldsymbol{r}_k.$$

or, equivalently,

$$\frac{dW}{dt} = c\boldsymbol{r}\boldsymbol{r}^T$$

where c is a constant. The problem with this form of pure Hebbian plasticity is that it can be unstable. Since W_{jk} can only increase, it has a tendency to grow without bound. This effect is amplified by a positive feedback loop: If W_{jk} increases, it cases $\boldsymbol{r}_j$ to increase, which causes W_{jk} to increase, *etc.* There are many biologically motivated approaches for resolving this instability. We will discuss a form of Hebbian-like plasticity that is self-stabilizing.

In particular, we will build and analyze a model of **homeostatic inhibitory synaptic plasticity** in which inhibitory synapses onto excitatory neurons are modulated in a way that pushes the excitatory firing rates toward a stable target rate [35–41]. First consider a single excitatory neuron receiving synaptic input from a single inhibitory neuron with synaptic strength $J_{ei} < 0$. The rule is defined by

$$\begin{aligned}
\frac{dJ_{ei}}{dt} &= -\epsilon\left[(y_e - 2r_0)\,S_i - S_e y_i\right] \\
\tau_y\frac{dy_e}{dt} &= -y_e + S_e \\
\tau_y\frac{dy_i}{dt} &= -y_i + S_i
\end{aligned} \tag{4.1}$$

Here, $S_e(t)$ and $S_i(t)$ are spike densities for the excitatory and inhibitory spike trains, $J_{ei} < 0$ is the synaptic weight from the i neuron to the e neuron, $r_0 > 0$ is a parameter called the **target rate** (for reasons we'll see soon), and $\epsilon > 0$ is a **learning rate** which controls how quickly the synaptic weight changes. The terms $y_e(t)$ and $y_i(t)$ are called **eligibility traces**, and they serve as continuous-time estimates of the neurons' recent firing rates. Note that their mean-field values are the rates, r_e and r_i. The timescale, τ_y, of the eligibility traces is related to the timescale of intracellular calcium and should be taken to be around $\tau_y \approx 100 - 1000$ms.

The last equation in (4.1) is the plasticity rule itself. Let's interpret what it is saying. Since $S_e(t)$ and $S_i(t)$ are sums of Dirac delta functions, J_{ei} is only updated after a spike in one of the neurons. After each inhibitory spike, J_{ei} is decremented by $y^e(t) - 2r_0$. After each excitatory spike, it is decremented by $y^i(t)$. Note that $J_{ei} < 0$ since it is inhibitory, so decrementing it makes the synapse stronger (more inhibitory).

Plasticity rules that depend on the timing of pre- and/or post-synaptic spikes are called **spike-timing dependent plasticity (STDP)** rules. They are widely observed in cortical networks. To understand the dependence of the weight change on spike timing, let's consider how the synaptic weight changes in response to a single spike in each neuron (Figure 4.1A). Suppose that the inhibitory neuron (which is presynaptic) spikes at time $t_{pre} > 0$ and the excitatory neuron (which is postsynaptic) spikes at time $t_{post} > 0$. Further assume that those are the only two spikes in recent history, so we can set initial conditions $y_e(0) = y_i(0) = 0$.

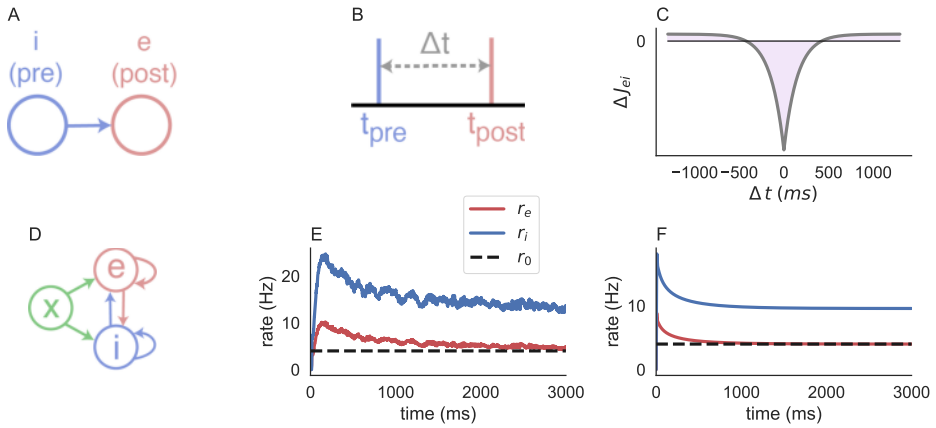

Figure 4.1: Inhibitory synaptic plasticity in a pair of neurons and a network. A) Diagram of a single inhibitory synapse onto an excitatory neuron. **B)** Each neuron spikes once and $\Delta t = t_{post} - t_{pre}$ is the delay between spikes. **C)** The change in synaptic weight, ΔJ_{ei} as a function of Δt. **D,E)** A recurrent spiking network like the one in Figure 3.3 except J^{ei} evolves through synaptic plasticity, which pushes excitatory firing rates (red) toward their target (black dashed). **F)** A mean-field rate network model approximated the firing rate dynamics from the spiking network. Code to reproduce this figure can be found in `SynapticPlasticity.ipynb`.

Let's first consider what happens when the inhibitory (presynaptic) neuron spikes first ($t_{pre} < t_{post}$). All terms are zero before the presynaptic spike time. In particular, $y_e(t_{pre}) = 0$, so at the time of the presynaptic spike, the synaptic weight will change by

$$\Delta J_{ei} = 2\epsilon r_0.$$

Next, the excitatory (postsynaptic) neuron spikes and causes an additional change given by

$$\Delta J_{ei} = -\epsilon y_i(t_{post}).$$

Now note that $y_i(t)$ is defined by an exponential decay after the inhibitory (presynaptic) spike time, so

$$y_i(t_{post}) = \frac{1}{\tau_y} e^{-(t_{post}-t_{pre})/\tau_y}$$

Taken together, the total change caused by the two spikes is given by

$$\Delta J_{ei} = -\frac{\epsilon}{\tau_y}\left(e^{-(t_{post}-t_{pre})/\tau_y} - 2r_0\right).$$

Now let's compute the weight change when the excitatory (postsynaptic) neuron spikes first. By similar reasoning, we have

$$\Delta J_{ei} = -\frac{\epsilon}{\tau_y}\left(e^{-(t_{pre}-t_{post})/\tau_y} - 2r_0\right).$$

Putting this all together, gives an unconditional weight change of

$$\Delta J_{ei} = -\frac{\epsilon}{\tau_y}\left(e^{-|\Delta t|/\tau_y} - 2r_0\right)$$

where

$$\Delta t = t_{post} - t_{pre}$$

is the time elapsed between the two spikes. This curve is plotted in Figure 4.1B. Synaptic plasticity is often measured in experiments using **paired pulse protocol** in which a pair of synaptically connected neurons is driven to spike consecutively. The change in synaptic strength is computed (averaged across many trials) and plots like Figure 4.1B are created for real neurons.

This paired pulse approach for quantifying the synaptic plasticity in simulations or experiments is lacking. Under natural settings, pairs of spikes in pre-synaptic and post-synaptic neurons do not occur in isolation, but the neurons are spiking continuously in response to inputs from other neurons. To capture this more realistic scenario, we next consider a recurrent spiking network with inhibitory plasticity, defined by

$$\begin{aligned}
\tau_m \frac{d\boldsymbol{V}^e}{dt} &= -(\boldsymbol{V}^e - E_L) + De^{(\boldsymbol{V}^e - V_T)/D} + \boldsymbol{I}^{ee}(t) + \boldsymbol{I}^{ei}(t) + \boldsymbol{I}^{ex}(t) \\
\tau_m \frac{d\boldsymbol{V}^i}{dt} &= -(\boldsymbol{V}^i - E_L) + De^{(\boldsymbol{V}^i - V_T)/D} + \boldsymbol{I}^{ie}(t) + \boldsymbol{I}^{ii}(t) + \boldsymbol{I}^{ix}(t) \\
\tau_b \frac{d\boldsymbol{I}^{ab}}{dt} &= -\boldsymbol{I}^{ab} + J^{ab}\boldsymbol{S}^b, \quad a = e, i, \quad b = e, i, x \\
\boldsymbol{V}_j^a(t) &> V_{th} \Rightarrow \text{spike at time } t \text{ and } \boldsymbol{V}_j^a(t) \leftarrow V_{re}, \quad a = e, i \\
\tau_y \frac{d\boldsymbol{y}^a}{dt} &= -\boldsymbol{y}^a + \boldsymbol{S}^a \quad a = e, i \\
\frac{dJ^{ei}}{dt} &= -\epsilon \left[(\boldsymbol{y}^e - 2r_0) \left[\boldsymbol{S}^i\right]^T - \boldsymbol{S}^e \left[\boldsymbol{y}^i\right]^T \right] \circ \Omega^{ei}
\end{aligned} \tag{4.2}$$

Eq. (4.2) is arguably the most complicated model that we will consider in this book. The first four equations are the same as in Section 3.2 and the last two implement the inhibitory plasticity rule at the network level. The notation $[\boldsymbol{S}^i]^T$ and $[\boldsymbol{y}^i]^T$ denoted the transpose of each vector. The matrix Ω_{ei} is just a binarized version of J^{ei},

$$\Omega_{jk}^{ei} = \begin{cases} 1 & J_{jk}^{ei}(0) \neq 0 \\ 0 & J_{jk}^{ei}(0) = 0. \end{cases}$$

where $J^{ei}(0)$ is the initial value of the matrix J^{ei}. Recall that $\circ$ denotes element-wise multiplication. Hence, the inclusion of Ω^{ei} in Eq. (4.2) makes sure that the plasticity rule is only applied to connected neurons, so the strength of connection between unconnected neurons remains zero.

In code, we do not store the spike trains as spike densities, so the updates to $\boldsymbol{y}^a(t)$ and $J^{ei}(t)$ need to be implemented more carefully. To update $y^a(t)$ and J^{ei} in response to excitatory spike, we do

```
Inds=np.nonzero(Ve>=Vth)[0]
ye[Inds]=ye[Inds]+1/tauy
Jei[Inds,:]=Jei[Inds,:]-dt*eta*yi*Omegaei[Inds,:]
```

The first line of code finds which excitatory neurons spikes and the next two lines update $\boldsymbol{y}^e$ and J^{ei} accordingly. The updates to $\boldsymbol{y}^i$ and J^{ei} after an inhibitory spike are

implemented similarly. Note that elements in J^{ei} can become positive from this plasticity rule, which is not biologically realistic. If we want to prevent this, we can add the line

```
Jei[Inds,:]=np.minimum(Jei[Inds,:],0)
```

after updating weights. See `SynapticPlasticity.ipynb` for a full implementation of the model in Eqs. (4.2). Figure 4.1D shows firing rates from a simulation. Note that the excitatory firing rates (red) seem to approach the target rate (dashed black). We next use a mean-field model to explain this result.

To derive a dynamical mean-field model of the spiking network simulation in Eqs. (4.2), we can first use the rate network model from Eq. (3.13) to model firing rate dynamics. The entire matrix, J^{ei}, is reduced to a single mean-field weight, w_{ei}, in this approximation. Now we need to use the last equation in Eqs. (4.2) to derive a mean-field approximation to the dynamics of w_{ei}. Recall from Section 3.1 that the relationship between w_{ei} and J^{ei} should be

$$w_{ei} = E[J^{ei}]N_i.$$

Now note that the expectation of the spike densities and the eligibility traces are equal to the rates, $E[\boldsymbol{S}^a] = E[\boldsymbol{y}^a] = r_a$. Putting these two facts together with the last equation in Eq. (4.2) gives

$$\frac{dw_{ei}}{dt} = -\epsilon\left[(r_e - 2r_0)r_i + r_e r_i\right] p_{ei} N_i.$$

Simplifying this expression and putting it into a mean-field rate model gives

A rate network model with homeostatic inhibitory plasticity

$$\begin{aligned}
\tau_e \frac{dr_e}{dt} &= -r_e + f_e(w_{ee}r_e + w_{ei}r_i + X_e) \\
\tau_i \frac{dr_i}{dt} &= -r_i + f_e(w_{ie}r_e + w_{ii}r_i + X_i) \\
\frac{dw_{ei}}{dt} &= -\epsilon_r(r_e - r_0)r_i
\end{aligned} \tag{4.3}$$

where

$$\epsilon_r = 2\epsilon p_{ei} N_i$$

is a rescaled learning rate, $X_e = w_{ex}r_x$, and $X_i = w_{ix}r_x$. The last equation in Eq. (4.3) shows that the inhibitory plasticity rule is almost like a pure Hebbian rule, except for the subtraction of r_0 from $\boldsymbol{r}_e$.

Figure 4.1 shows simulations of this rate model, which capture the overall trends seen in the spiking network. More importantly, Eq. (4.3) helps us understand why excitatory rates approach r_0: Any fixed point of the system in Eq. (4.3) must satisfy $r_e = r_0$ (unless $r_i = 0$, which is not the case here). Hence, the inhibitory synaptic plasticity rule pushes excitatory firing rates toward the target rate, r_0.

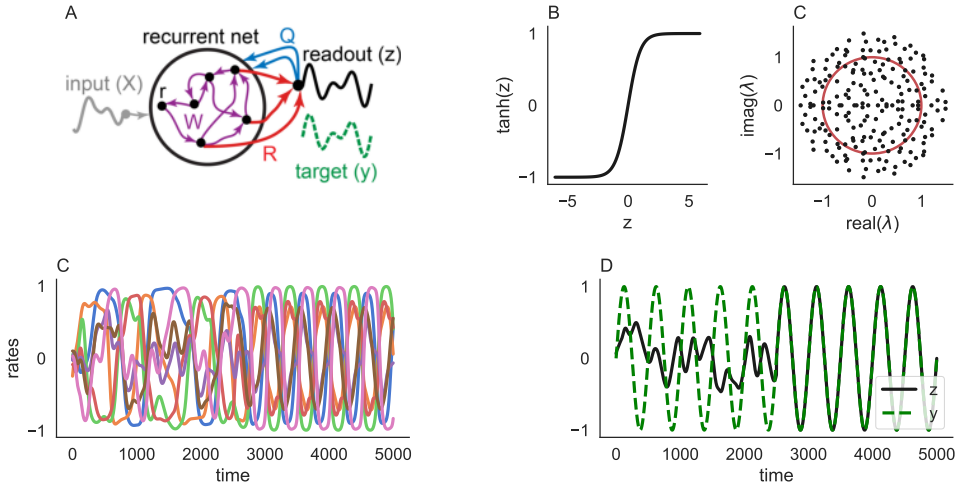

Figure 4.2: Training readouts from a chaotic recurrent neural network. A) Diagram of a recurrent neural network (RNN) with readout trained to produce a target time series. **B)** Hyperbolic tangent function used as an f-I curve. **C)** Eigenvalues of W. Unit circle shown in red. **D)** A sample of 7 firing rates out of $N = 200$ from a network simulation without external input, $\boldsymbol{X}(t) = 0$. **E)** Output (z) and target ($\boldsymbol{y}$). Learning and feedback were only enabled after time $t = 2500$. The target is a one-dimensional ($M = 1$) sine wave. Code to reproduce this figure can be found in `RNN.ipynb`.

Exercise 4.1.1. The fixed point analysis above only applies for time-constant input $X_e(t) = X_e$ and $X_i(t) = X_i$. When external input changes in time, the network cannot generally maintain a fixed firing rate $r_e = r_0$. Try running a rate network simulation in which external input changes in time. For example, try a simulation where $X_e(t)$ and $X_i(t)$ change their values halfway through the simulation.

4.2 TRAINING RECURRENT NEURAL NETWORK MODELS

We began this chapter by saying that we will build models that learn to "do something," but the model in the previous section hardly fulfills that promise because it only learns to produce a constant target rate, which is not a very useful task. We are gradually building up to models that can perform more difficult and useful tasks. In this section, we raise the bar just a little higher to build a model that can produce a time-varying target, $\boldsymbol{y}(t)$, instead of a constant target, r_0.

As our models become more capable of solving real tasks, they will also become more abstract and removed from biology. In this section, we omit the biological details of spiking to start with a recurrent rate network model. Specifically, the model is defined by (Figure 4.2A)

Recurrent neural network (RNN) model

$$\begin{aligned} \tau \frac{d\boldsymbol{r}}{dt} &= -\boldsymbol{r} + f(W\boldsymbol{r} + \boldsymbol{X} + Q\boldsymbol{z}) \\ \boldsymbol{z} &= R\boldsymbol{r} \end{aligned} \tag{4.4}$$

Similar to the rate network model in Eq. (3.11) from Section 3.3, $\boldsymbol{r}(t) \in \mathbb{R}^N$ is vector of firing rates, $\tau > 0$ is a time constant, f is an f-I curve, and $\boldsymbol{X}(t)$ is external input. An alternative formulation of RNN models [42] starts from the formulation of rate networks from Eq. (3.12) instead of Eq. (3.11).

Unlike Eq. (3.11), we now have an additional term $\boldsymbol{z}(t) \in \mathbb{R}^M$, which is a **readout** from the network. This is interpreted as output from the network and the goal is to find an R that makes $\boldsymbol{z}(t)$ match a target time series, $\boldsymbol{y}(t) \in \mathbb{R}^M$. The matrix $R \in \mathbb{R}^{M \times N}$ is a readout matrix and $Q \in \mathbb{R}^{N \times M}$ is a feedback matrix that injects the readout back into the network.

The presence of feedback from the readout, $\boldsymbol{z}(t)$, is the only technical difference between Eqs. (4.4) and (3.11), but we will also use very different parameters. In Section 3.3, we considered rate network models with just two populations, representing the average excitatory and inhibitory firing rates. For Eq. (4.4), we will consider a much larger network. In general, we should use at least $N = 100$ for things to work well. For the example considered here, we choose $N = 200$. Firing rates are also interpreted more abstractly. Instead of interpreting $\boldsymbol{r}(t)$ as a vector of literal firing rates in physical units like Hz, we assume that rates take values in the interval $[-1, 1]$, meaning that they can even be negative. To achieve this, we use the hyperbolic tangent as an f-I curve,

$$f(z) = \tanh(z) = \frac{e^{2z} - 1}{e^{2z} + 1}.$$

This is an "S-shaped" function, which is strictly increasing and bounded above and below by 1 and -1 respectively (Figure 4.2B). Such S-shaped functions are called **sigmoidal** functions. While allowing negative rates might seem odd, the idea is that more realistic rate values can be re-scaled and shifted to lie in $[-1, 1]$, so it shouldn't matter. Moreover, this model is more abstract and the "rates" here might not be intended to represent literal firing rates, but are just a proxy for "neural activity" in general. We similarly allow entries in the connectivity matrices to take on positive and negative values, without obeying Dale's law. The recurrent connectivity matrix, W, is drawn with independent normally distributed entries with variance given by ρ/N. In PyTorch, the matrix is generated by

```
W=rho*np.random.randn(N,N)/np.sqrt(N)
```

To better understand the dynamics of the network, let's first analyze the network without feedback, $Q = 0$, and without input, $\boldsymbol{X} = 0$. In the absence of feedback, the model is equivalent to the rate network model from Eq. (3.11). Since $f(0) = \tanh(0) = 0$ there is a fixed point at $\boldsymbol{r} = 0$. Moreover, since $f'(0) = \tanh'(0) = 1$, the corresponding Jacobian matrix is

$$J = \frac{1}{\tau}\left[-I + W\right].$$

where I is the identity matrix. It is not difficult to check that the eigenvalues of J satisfy

$$\Lambda(J) = \frac{1}{\tau}\left[-1 + \Lambda(W)\right] \tag{4.5}$$

where $\Lambda(W)$ are the eigenvalues of W. Therefore, we can determine the stability of the fixed point from the eigenvalues of W. But what are the eigenvalues of a large, random matrix like W? Fortunately, **Girko's circular law** says that if W is a random $N \times N$ matrix with i.i.d. entries satisfying $E[W_{jk}] = 0$ then its eigenvalues are approximately uniformly distributed in a circle centered at the origin in the complex plane. The radius of this circle is called the matrix's **spectral radius** and it is approximated by [43–45]

$$r \approx \sqrt{N\text{var}(W_{jk})}$$

where $\text{var}(W_{jk})$ is the variance of the entries of W and the approximation becomes increasingly accurate as $N \to \infty$. A precise statement of the theorem requires more background in probability theory than is appropriate for this book. Figure 4.2C shows the eigenvalues of W when $N = 200$ and $\rho = 1.5$. The red circle shows a circle of radius 1 for comparison.

From Eq. (4.5), we can conclude that the eigenvalues of J lie in a circle of radius r/τ centered at $-1/\tau$ where r is the spectral radius of W. Hence, stability is likely whenever $\rho < 1$ and instability is likely when $\rho > 1$. It might be easier to think about the eigenvalues of τJ. The fixed point is stable when the eigenvalues of τJ have negative real part. The eigenvalues of τJ are given by $\Lambda(\tau J) = \Lambda(W) - 1$, *i.e.*, they are given by shifting the points in Figure 4.2C to the left by one unit. Hence, the fixed point is stable if the spectral radius of W is smaller than 1, *i.e.*, if all of the dots in Figure 4.2C lie within the red circle. Randomness in the eigenvalues can make the true spectral radius different from ρ, but when N is large, the transition between stability and instability is very likely to occur very close to $\rho = 1$.

In summary, if ρ is sufficiently smaller than 1, the fixed point at $\boldsymbol{r} = 0$ is stable and $\boldsymbol{r}(t) \to 0$ as t increases. When ρ is sufficiently larger than 1, the fixed point is unstable. What kind of dynamics are produced by this instability? Note that $\boldsymbol{r}_j(t) \in [0, 1]$, so the rates can't blow up. Instead, stability is lost through a high-dimensional bifurcation that produces intricate, chaotic dynamics [45, 46]. Essentially, each rate $\boldsymbol{r}_j(t)$ traces out a complicated, but relatively smooth time series. The magnitude of τ and ρ control the timescale and disorderliness of the dynamics. The first half of Figure 4.2D (up to time 2500) shows a sample of firing rates when $\rho = 1.5$, feedback is turned off.

The idea behind the RNN models studied in this section is that the dynamics of $\boldsymbol{r}(t)$ are rich and high-dimensional, so we should be able to "mine" them to produce an arbitrary time series. In other words, there should be a readout matrix, R, for which $\boldsymbol{z}(t) = R\boldsymbol{r}(t)$ match our target time series, $\boldsymbol{y}(t)$. We can quantify the "error" of the network by the Euclidean distance between $\boldsymbol{z}(t)$ and $\boldsymbol{y}(t)$,

$$e = \|\boldsymbol{z} - \boldsymbol{y}\|^2 = \sum_{j=1}^{M} (\boldsymbol{z}_j - \boldsymbol{y}_j)^2.$$

We'd like to find an update to R that reduces e. Let's consider what happens when we change R by a little bit to get a new matrix,

$$R' = R + \Delta R$$

and we assume that ΔR is small in the sense that $\|\Delta R\boldsymbol{u}\|$ is much smaller than $\|\boldsymbol{u}\|$ for any vector, $\boldsymbol{u}$. This is a common way of quantifying the magnitude of a matrix. The change to R causes a change to the readout given by

$$\Delta \boldsymbol{z} = \boldsymbol{z}' - \boldsymbol{z} = R'\boldsymbol{r} - R\boldsymbol{r} = \Delta R\boldsymbol{r}$$

where we assume that we're measuring $\boldsymbol{z}'$ just after the change to R and measuring $\boldsymbol{z}$ just before. Note that $\|\Delta \boldsymbol{z}\| = \|\Delta R\boldsymbol{r}\|$ is small because of our assumption that ΔR is small and because $\|\boldsymbol{r}\|$ cannot be very large. Above, we assumed that changing R does not affect $\boldsymbol{r}$. This assumption would be clearly valid in the absence of feedback ($Q = 0$). In the presence of feedback, it is still valid because we measure $\boldsymbol{z}'$ immediately after updating R, before feedback has a chance to change $\boldsymbol{r}$. Let's now compute the change to the error,

$$\begin{aligned}
\Delta e &= e' - e \\
&= \|\boldsymbol{z}' - \boldsymbol{y}\|^2 - \|\boldsymbol{z} - \boldsymbol{y}\|^2 \\
&= (\boldsymbol{z}' - \boldsymbol{y})^T(\boldsymbol{z}' - \boldsymbol{y}) - (\boldsymbol{z} - \boldsymbol{y})^T(\boldsymbol{z} - \boldsymbol{y}) \\
&= (\boldsymbol{z} + \Delta \boldsymbol{z} - \boldsymbol{y})^T(\boldsymbol{z} + \Delta \boldsymbol{z} - \boldsymbol{y}) - (\boldsymbol{z} - \boldsymbol{y})^T(\boldsymbol{z} - \boldsymbol{y}) \\
&= 2(\boldsymbol{z} - \boldsymbol{y})^T\Delta \boldsymbol{z} + \|\Delta \boldsymbol{z}\|^2
\end{aligned}$$

Since $\|\Delta \boldsymbol{z}\|$ is small, we therefore have

$$\Delta e \approx 2(\boldsymbol{z} - \boldsymbol{y})^T\Delta \boldsymbol{z} = 2(\boldsymbol{z} - \boldsymbol{y})^T\Delta R\boldsymbol{r} \tag{4.6}$$

To decrease the error, we want to make sure that $\Delta e < 0$. To achieve this, we can take

Least mean squares (LMS) update rule for readout matrices.

$$\Delta R = -\epsilon(\boldsymbol{z} - \boldsymbol{y})\boldsymbol{r}^T \tag{4.7}$$

where $\epsilon > 0$ is a small learning rate that is included to make sure that ΔR is small in magnitude.

Exercise 4.2.1. Use Eq. (4.6) to prove $\Delta e < 0$ for the LMS update in Eq. (4.7). **Note:** You implicitly assume that $\epsilon > 0$ is sufficiently small when you use Eq. (4.6). For a more formal derivation, use $\Delta e = 2(\boldsymbol{z} - \boldsymbol{y})^T\Delta \boldsymbol{z} + \mathcal{O}(\epsilon^2)$ in place of Eq. (4.6) where $\mathcal{O}(\epsilon^2)$ goes to zero like ϵ^2 as $\epsilon \to 0$.

The first half of Figure 4.2C,D (after time 2500) shows firing rates, readout, and targets after feedback and learning are turned on (using the LMS rule to update R). The readout quickly converges to the target.

The practice of training a RNN by updating only a set of readout weights is called **reservoir computing** and the corresponding network model is called an **echo state network** or **liquid state machine** [42, 47–50]. There are many variants of reservoir computing algorithms. For example, some models do not use feedback ($Q = 0$), but feedback tends to improve learning. Some models include an input, $\boldsymbol{X}(t)$, and use a target that depends on the input so the network needs to learn a mapping from an input time series, $\boldsymbol{X}(t)$, to an output time series, $\boldsymbol{z}(t)$. Improved learning rules have been

developed [42] in addition to "reward-modulated" learning rules that do not require knowledge $\boldsymbol{y}(t)$, but only need $e(t) = \|\boldsymbol{z}(t) - \boldsymbol{y}(t)\|^2$ to update R [51–53].

The exercises below identify some weaknesses of the LMS learning rule as it is defined above and provide some approaches to improve it.

Exercise 4.2.2. In Figure 4.2, we only compared the feedback and target while learning was turned on. Ideally, the network would produce small error even after learning is turned off (R held fixed after learning). Repeat the simulation in Figure 4.2, but turn off learning by holding R fixed after some time (keep the feedback turned on). You will see that the output does not match the target after learning is turned off. This is due to the fact that our derivation of ΔR only required that the error is smaller immediately after updating R, but it will not necessarily keep the error small if R is not updated again on the next time step. In essence, with the LMS learning rule, the network doesn't really *learn* to produce the target, but the readout just "chases" the target around [42].

Exercise 4.2.3. Previous work by David Sussillo and Larry Abbott [42] showed that more stable learning can be achieved by modified learning rules called "FORCE learning." One form of FORCE learning is similar to LMS, but the learning rate decreases as the error decreases. A more powerful and widely used approach is an improved learning rule called **recursive least squares (RLS)**,

$$\begin{aligned} \Delta R &= -\epsilon(\boldsymbol{z} - \boldsymbol{y})\boldsymbol{r}^T P \\ \Delta P &= -\frac{\epsilon_P}{1 + \boldsymbol{r}^T P \boldsymbol{r}} P \boldsymbol{r}\boldsymbol{r}^T P \end{aligned} \tag{4.8}$$

where $\epsilon_P > 0$ and P is an $N \times N$ matrix initialized to $P = cI$ where I is the identity matrix and $c > 0$ is a constant. Under this rule, $P = \sum_t \boldsymbol{r}(t)\boldsymbol{r}^T(t) + (1/c)I$ where I is the identity matrix and the sum is over all discrete time points used in the Euler loop. This is a "regularized" estimate of the un-normalized correlation between elements of $\boldsymbol{r}(t)$ across time where the $(1/c)I$ term is a regularizer. Smaller values of c produce faster learning, but too small values can cause instabilities. We should always choose c much larger than $1/N$ and typically values in $c \in [0.01, 1]$ tend to work well. This rule is computationally expensive because the computation of P is expensive, but the computation time can be improved by storing the value of $P\boldsymbol{r}$ into a temporary variable like,

$$\begin{aligned} \boldsymbol{u} &= P\boldsymbol{r} \\ \Delta P &= -\frac{\epsilon_P}{1 + r^T \boldsymbol{u}} \boldsymbol{u}\boldsymbol{u}^T \end{aligned}$$

Using the fact that P is symmetric, you can see that this produces the same value of ΔP, but requires fewer matrix products. Repeat the previous exercise using RLS learning in place of LMS learning. Find parameters that produce small error even after learning is turned off. The RLS learning rule is capable of learning complex tasks with multiple output dimensions ($M > 1$) and where the target, $\boldsymbol{y}(t)$, depends on the history of the input, $\boldsymbol{X}(t)$ [42].

While the RNN models and learning rules considered here are far removed from biology, the dynamics of $\boldsymbol{r}(t)$ in a trained RNN share statistical features of neural activity recorded in animals performing similar motor tasks and can therefore be used as abstract models of neural activity during motor tasks [52–56].

Technically, the feedback matrix, R, represents a recurrent connectivity matrix (at least when $Q \neq 0$) because there is a recurrent loop from $\boldsymbol{r}$ to $\boldsymbol{z}$ and back to $\boldsymbol{r}$ [42]. However, most of the recurrence in the network is contained in the matrix, W, which is not trained by the learning rules considered in this section. If we want to learn W, it is more common to use gradient descent methods developed for artificial neural networks, which are the topic of the next section.

4.3 PERCEPTRONS AND ARTIFICIAL NEURAL NETWORKS

You may have heard of "deep neural networks" which are a leading method for machine learning. Even if you haven't heard of them, you've very likely used them. If you've ever translated a webpage online, used an app that utilizes facial recognition or voice recognition, or many of the other seemingly magical modern applications of artificial intelligence, then you have very likely benefited from the power of deep neural networks. What is the word "neural" doing in "deep neural networks"? The development of deep neural networks and their predecessors were inspired by biological neural networks. Indeed, deep neural networks are a type of artificial neural networks, which are closely related to rate network models.

Let us start by considering a **feedforward, single-layer, fully connected artificial neural network (ANN)** which is also called a **single-layer perceptron** [57]. A single-layer perceptron is equivalent to a feedforward mean-field equation. In particular a single-layer perceptron can be defined by removing recurrent connections (setting $W = 0$) in Eq. (3.8) to get $\boldsymbol{r} = f(W_x r_x)$. However, we will switch to notational conventions more commonly used for artificial neural networks,

Single layer perceptron

$$\boldsymbol{v} = f(W\boldsymbol{x}). \tag{4.9}$$

which can also be written as $\boldsymbol{v} = f(\boldsymbol{z})$ where

$$\boldsymbol{z} = W\boldsymbol{x}.$$

This gives us a mapping from a vector of inputs, $\boldsymbol{x} \in \mathbb{R}^{N_0}$ to outputs, $\boldsymbol{v} \in \mathbb{R}^{N_1}$. The outputs are also sometimes called **activations**, $W \in \mathbb{R}^{N_1 \times N_0}$ is the **feedforward weight matrix**, and $f : \mathbb{R} \to \mathbb{R}$ is called the **activation function** which is applied pointwise. The term "fully connected" refers to the fact that W connects every element of $\boldsymbol{x}$ to every element of $\boldsymbol{z}$ or $\boldsymbol{v}$. Perceptrons are fully connected by definition, but ANNs are a more general class of networks.

In **supervised learning**, we begin with a data set of m inputs and labels

$$\{\boldsymbol{x}^i, \boldsymbol{y}^i\}_{i=1}^m$$

and the goal is to find a weight matrix, W, so that the relationship between $\boldsymbol{x}$ and $\boldsymbol{v}$ in Eq. (4.9) approximates the relationship between $\boldsymbol{x}^i$ and $\boldsymbol{y}^i$ from the data. We use

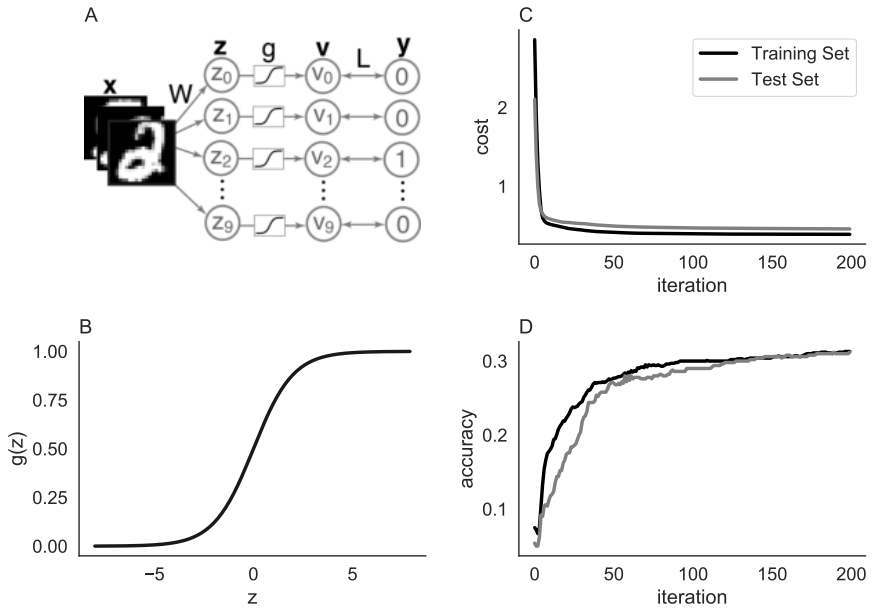

Figure 4.3: A perceptron trained on MNIST classification. A) Diagram of a perceptron with images as input and one-hot encoded labels. **B)** The logistic sigmoid loss function. **C,D)** The cost and accuracy on the training and test data, as a function of the gradient descent iteration step. Accuracy is defined as the proportion of inputs guessed correctly. Code to reproduce this figure can be found in `Perceptron.ipynb`.

superscripts instead of subscripts to enumerate the data because subscripts are used to index the vectors (so $\boldsymbol{x}^i_j$ is the jth entry of input i).

The perceptron should learn to approximate a function represented by the data. To measure the error of the network under a particular set of weights, we define a **cost function**,

$$J(W) = \frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{v}^i, \boldsymbol{y}^i)$$

where $\boldsymbol{v}^i = f(W\boldsymbol{x}^i)$ is the output of the network on input $\boldsymbol{x}^i$ and L is a **loss function** which returns a number representing how far apart $\boldsymbol{v}^i$ and $\boldsymbol{y}^i$ are. Sometimes the word "loss" is used to refer to the cost, but they are closely related and it's usually easy to tell what is meant from context.

For a specific example of supervised learning, we consider the MNIST data set, which consists of 28×28 grayscale images of hand-written digits, which are the inputs. The full MNIST data set contains $70,000$ images, but we will restrict ourselves to a subset of $m = 1000$ images. We represent the inputs as vectors in $N_0 = 28 * 28 = 784$ dimensions, *i.e.*, each input is a list of pixel values. The labels are 10-dimensional binary vectors

with a 1 in the entry associated with the digit. This is called a **one-hot encoding**. For example, the one-hot encoded label for a hand-written 2 is

$$\boldsymbol{y} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

where note that the first entry corresponds to digit 0, so 2 is the third digit. Outputs of the network should therefore have dimension $N_1 = 10$ and W should be 10×784. Figure 4.3A shows a diagram of the network. Since the entries of the labels are in the interval $[0, 1]$, we should use an activation function that returns outputs in $[0, 1]$. We will use the logistic sigmoid function,

$$f(z) = \sigma(z) = \frac{e^z}{e^z + 1}.$$

This is similar to the tanh sigmoid from Section 4.2 except its outputs lie in $[0, 1]$ instead of $[-1, 1]$ (Figure 4.3B). One nice property of the logistic sigmoid is that its derivative can be written in terms of the function itself,

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)).$$

We will use a **mean squared error (MSE)** loss function,

$$L(\boldsymbol{v}, \boldsymbol{y}) = \frac{1}{2}\|\boldsymbol{v} - \boldsymbol{y}\|^2 = \frac{1}{2}\sum_{j=1}^{10}(\boldsymbol{v}_j - \boldsymbol{y}_j)^2. \tag{4.10}$$

Technically, we should divide by 10 instead of 2 to get a "mean" squared error, but the difference is not important and the coefficient of $1/2$ makes the math work out nicely later. The goal, then, is to find a matrix, W, that achieves a small cost.

Finding parameters, W, to minimize a cost function is a form "training" or "learning" or, more directly, **optimization**. The most common way to train artificial neural networks is **gradient descent**. To understand gradient descent, think of $J(W)$ as a surface or landscape with each value of W representing a location on the landscape and $J(W)$ representing its height or "altitude." This is called the **loss landscape**. We want to find deep valleys in the loss landscape. A simple way to find deep valleys is to walk in the direction of steepest descent, *i.e.*, to update W in the direction along which $J(W)$ decreases most rapidly. We can use the gradient of $J(W)$ to find the direction of steepest descent. Since W is a matrix, the gradient is also a matrix (even though gradients are technically defined as vectors) with entries defined by

$$[\nabla_W J(W)]_{jk} = \frac{\partial J}{\partial W_{jk}} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial L(\boldsymbol{v}^i, \boldsymbol{y}^i)}{\partial W_{jk}} \tag{4.11}$$

In other words, each entry in $\nabla_W J(W)$ is the derivative of the cost with respect to the associated entry in W. The gradient, $\nabla_W J(W)$, at a particular value of W is a vector that

points in the steepest direction uphill. Therefore, the steepest direction downhill is the negative of the gradient. In gradient descent, we take small steps in the direction of the negative gradient. In other words, we make the following update to W,

$$\Delta W = -\epsilon \nabla_W J(W).$$

where $\epsilon > 0$ is the **learning rate**. After updating W, we compute the output, cost, and gradient again, then update W again using the new gradient. We repeat this procedure for some number of iterations.

Performing gradient descent requires us to compute the gradient of the loss with respect to the weights, which appears in Eq. (4.11). For the single-layer perceptron in Eq. (4.9) with the MSE loss in Eq. (4.10), this gradient is given by

$$\frac{\partial L(\boldsymbol{v}^i, \boldsymbol{y}^i)}{\partial W_{jk}} = (\boldsymbol{v}^i{}_j - \boldsymbol{y}^i_j) f'(\boldsymbol{z}^i_j) \boldsymbol{x}^i_k.$$

The corresponding update to W is known as **the delta rule** [58],

The delta rule

$$\Delta W = -\frac{\epsilon}{m} \sum_{i=1}^{m} \left[(\boldsymbol{v}^i - \boldsymbol{y}^i) \circ f'(\boldsymbol{z}^i) \right] \left[\boldsymbol{x}^i \right]^T \tag{4.12}$$

where recall that $\circ$ is element-wise multiplication and $\cdot^T$ is the transpose. Note that the LMS learning rule in Eq. (4.7) is actually a special case of the delta rule with $f'(z) = 1$. Indeed, the derivation of Eq. (4.7) in Section 4.2 was just a derivation of the gradient of e with respect to R for the feedforward network $\boldsymbol{z} = f(R\boldsymbol{r})$ with $f(z) = z$. Figure 4.3C shows the loss of a single-layer perceptron on MNIST as training proceeds. Code to reproduce the figure can be found in `Perceptron.ipynb`.

It's difficult to interpret performance looking at the loss alone. Instead, we can define the network's best "guess" on an input by the value of j at which $\boldsymbol{v}_j$ is the largest. We can then ask whether the network was correct, *i.e.*, whether the guess matches the true label. The **accuracy** of the network is then defined as the proportion of inputs on which the network's guess is correct. Figure 4.3D shows the accuracy during training. Note that a network guessing randomly would achieve an accuracy of 0.1, so the network performs well above chance, but still far below human performance.

So far, we only checked the network's accuracy on the data set on which it was trained. This is, in some sense, cheating because the network could just learn to memorize those inputs. Typically, the true goal of learning is to perform well on unseen inputs. In `Perceptron.ipynb`, we also checked the performance on separate set of data that was hidden from the network during training, which is called the **test data** or **validation data**. In this context, the data used to train the model is called the **training data**. The ability to perform well on unseen data is called **generalization**. The model performed similarly on the test and training data (Figure 4.3C,D) indicating that the network generalized well. Animals are excellent at generalizing and artificial neural networks strive to approximate this ability.

Exercise 4.3.1. *This is my favorite exercise in the book. It ties everything together!* The MNIST task is a nice benchmark, but not very biologically relevant. Let's

take a more biologically relevant task. In Section 2.1, we looked at how the orientation of a bar is encoded in the spike count of a real neuron. Appendix B.5 extends this analysis to populations of neurons using statistical approaches. Let's now train a single-layer perceptron (which is a biologically inspired model) to classify orientations from real spike counts. This can be viewed as a model of how the spiking activity of recorded neurons might be decoded by a downstream population (modeled by the perceptron) that computes what the monkey is seeing. The code in `DecodeSpikeCountsWithPerceptron.ipynb` loads a matrix, `x`, of spike counts from $N_0 = 112$ neurons in a monkey's visual cortex recorded over $m = 1000$ trials. On each trial, the monkey viewed a drifting grating stimulus with one of $N_1 = 12$ orientations. The matrix `y` contains the one-hot encoded orientations for each trial.
Train a single-layer perceptron to predict the orientation from the spike counts. Test your trained model on the test data in `xTest` and `yTest`.
After you complete this exercise, you will have built an algorithm that can *read a monkey's brain!* This approach can be extended to build brain-machine interfaces to create artificial eyes, control control robotic arms, *etc.*

The network in Figure 4.3 is a toy model intended to learn some basic principles of artificial neural networks. As the next exercise shows, the network's performance can be improved greatly just by using a different loss and activation function. A few more changes can produce a network capable of learning much more challenging tasks. For example, we can sample randomly from a large data set on each iteration instead of using the entire data set. This optimization procedure, known as stochastic gradient descent, can improve computational efficiency and generalization. We could add multiple layers to build a **multi-layered perceptron**. We can also replace the matrix-multiplication in some layers by two-dimensional convolutions, recurrent connections, or other functions. This more flexible class of multi-layered networks in which each layer can implement an arbitrary function are called **artificial neural networks** or (when they have several layers) **deep neural networks**. Deep neural networks are state-of-the-art for many machine learning applications. Computing gradients in deep networks is trickier, but it can be achieved using an algorithm known as **backpropagation**, which is a generalization of the delta rule. For recurrent neural networks, like the one from Eq. (4.4), the gradients of the loss with respect to the recurrent weights, W, can be computed using an algorithm known as backpropagation through time. It is still unknown whether and/or how the brain might implement or approximate backpropagation and/or backpropagation through time, and this is currently a highly active area of research [59–61]. Appendix B.11 discusses backpropagation and the issues with its potential approximation in biological neural circuits.

See Appendix E a description of backpropagation algorithm for tr multi-layered n networks and th associated probl credit assignme

Exercise 4.3.2. While the network in Figure 4.3 performed above chance, its performance is still below 50%, which is not great. Performance can be improved

by changing the activation and loss functions. In particular, we can combine a simple, linear activation function $f(z) = z$ with a more complicated loss function

$$\begin{aligned} L(\boldsymbol{v}, \boldsymbol{y}) &= \text{softmax}(\boldsymbol{v}, k) \\ &= -\log\left(\frac{\exp(\boldsymbol{v}_k)}{\sum_{j=1}^{M}\exp(\boldsymbol{v}_j)}\right) \\ &= -\boldsymbol{y}\cdot\boldsymbol{v} + \log\left(\sum_j \exp(\boldsymbol{v}_j)\right) \end{aligned}$$

where k is the index of the true label, *i.e.*, $\boldsymbol{y}_k = 1$. Note that log here refers to the natural logarithm (sometimes denoted ln). The expression inside of the first logarithm is called a **softmax** function and this loss is sometimes called the **categorical cross-entropy loss** for reasons that are outside the scope of this book. This loss function encourages $\boldsymbol{v}_k$ to be large while also encouraging all other $\boldsymbol{v}_j$ to be small. This imposes a form of competition (see Appendix B.9) between the outputs. Note that outputs are no longer restricted to $[0, 1]$. Outputs can be normalized by taking $p_i = \text{softmax}(\boldsymbol{v}, i)$ to get probabilities assigned to each category (since $\sum_j p_j = 1$). The delta rule only applies to the squared Euclidean loss function, so we can't use it for this loss, but the gradient is

$$\Delta W = -\frac{\epsilon}{m}\sum_{i=1}^{m}\left[\nabla_{\boldsymbol{v}} L\right]\left[\boldsymbol{x}^i\right]^T$$

where

$$\nabla_{\boldsymbol{v}} L = \frac{\exp(\boldsymbol{v})}{\sum_j \exp(\boldsymbol{v}_j)} - \boldsymbol{y}$$

is the gradient of $L(\boldsymbol{v}, \boldsymbol{y})$ with respect to $\boldsymbol{v}$.
Repeat Figure 4.3 (see code in `Perceptron.ipynb`) using gradient descent with a categorial cross entropy loss function.

We started our discussion of artificial neural networks by linking them to feedforward rate models, which were derived from an approximation to more biologically detailed spiking network models. However, artificial neural networks are very different from biological neural circuits. For example, artificial neural networks don't have action potentials, don't have separate excitatory and inhibitory neurons (they don't obey Dale's law), and the delta rule and backpropagation are very different from synaptic plasticity. Nevertheless, artificial neural networks are becoming one of the most popular and successful class of models for studying neural circuits. There are two perspectives for using artificial neural networks as models of biological neural circuits.

One perspective is that the finer details shouldn't matter. The idea is that the representations of stimuli learned by artificial neural networks should approximate those learned by real brains even if the details of their implementation are very different. This perspective implicitly relies on some form of universality, *i.e.*, that different learning algorithms and models trained on similar tasks will have similar representations. This perspective has some support: Networks designed for machine learning produce activations that are correlated with neural activity recorded from animals viewing the same inputs as stimuli [62–64].

Another perspective is that artificial neural networks *do* approximate the details of biological networks to some extent (via their relationship to feedforward rate models) and we should be able to obtain more accurate models by making artificial neural networks operate more like biological neural networks [65]. For example, we could impose Dale's law onto artificial neural networks [66], train the networks with biologically plausible plasticity rules, or train spiking networks instead of rate networks [67–72].

Each perspective likely has some merit. There are probably some biological details that can improve the similarity between activity in artificial and biological neural networks, and some biological details that do not matter much. Which details are important likely depends on which data are being modeled and/or what questions are being asked. For example, if we only care about modeling neural activity or behavior in a trained animal, then using biologically realistic learning rules might not be important. But if we care about neural activity or behavior during learning, then it might be more important to use biologically realistic learning rules.

As in many applications of computational and mathematical modeling, a primary challenge is the choice and design of the model, which is more of an art than a science. The model needs to be chosen in such a way that it can be expected to capture the phenomena being modeled and answer the questions being asked, but still be as simple as possible under those constraints. The following relevant quotation is often attributed to Einstein, but it is probably apocryphal [73],

> "Everything should be made as simple as possible, but no simpler."

This advice should serve as a guide for designing models of neural circuits and for designing mathematical models of natural phenomena more generally.

Part

APPENDICES

A MATHEMATICAL BACKGROUND

A.1 INTRODUCTION TO PYTHON AND NUMPY

Python is a general purpose programming language that is very popular in academic research. Some advantages of Python are that it's freely available, widely used, and there are many freely available Python packages that provide tools for all sorts of tasks. The NumPy package for Python provides a framework and many functions for common operations in numerical computing. The MatPlotLib package offers support for plotting data. The NumPy and MatPlotLib packages use syntax that is very similar to the standard syntax in Matlab.

One disadvantage of Python is that it is an interpreted language in contrast to compiled languages like C. Interpreted languages tend to be slower than compiled languages for many tasks. For example, loops in Python are much slower than those in C. But C is more difficult to learn, write, and debug. Fortunately, NumPy has built-in functions that allow you to avoid using loops when performing many common numerical operations. These built-in functions call pre-compiled C code that run much faster than the same function written directly in Python. The practice of using built-in functions instead of hand-written functions is called **vectorizing** your code.

There are several alternatives to Python. Matlab arguably has a simpler syntax, but it is expensive. Octave is a free, open source alternative to Matlab, but lacks some packages available in Matlab. Julia is an increasingly popular language that executes loops with similar speed to C using "just in time" (JIT) compilation.

All of the code associated with this book is written in Python notebooks, which are interactive environments for running blocks of Python code. Each block of code is called a "code cell" and there are also "text cells" used for documenting the code, *etc.* Notebooks are great for learning and for building small projects, but usually not appropriate for building larger projects, which are typically written in plain text files with a `.py` file extension.

The best way to learn a programming language is to use it, but first you need a basic understanding of the syntax, *etc.* **The code in `PythonIntro.ipynb` provides explanations, examples, and exercises to get familiar with the basics of Python, NumPy, and MatPlotLib.** Go through the code in `PythonIntro.ipynb`, make sure you understand *everything*, and complete all of the exercises. After that, you should be able to learn new concepts and tools as you go through the book. If you feel like you need more instruction after working through the code in `PythonIntro.ipynb`, there are numerous free resources for learning Python online. There will inevitably be programming problems that you get stuck on, bugs that you can't resolve, and things that you don't know how to do. This is true even for people who have years of experience in a programming language. One of the most important skills in programming is the ability to resolve problems like this by looking things up online and by experimenting with your code to find a solution.

A.2 INTRODUCTION TO ORDINARY DIFFERENTIAL EQUATIONS

A **differential equation** is an equation that relates an unknown function to its derivatives. For example, consider the equation

$$\frac{dy}{dt} = f(y,t) \tag{A.1}$$

where f is a known function, $y : \mathbb{R} \to \mathbb{R}$ is an unknown function, and $dy/dt = y'(t)$ is the derivative of y. The notation $y : \mathbb{R} \to \mathbb{R}$ means that y is a function that takes a real number as input and returns a real number. ODEs like Eq. (A.1) are sometimes also written using the notation

$$y' = f(y,t) \quad \text{or} \quad \dot{y} = f(y,t).$$

All of these equations mean the same thing: We are looking for a function $y(t)$ satisfying

$$y'(t) = f(y(t),t)$$

for all values of t in the domain being considered. Of course, y, t, and f can have different names too, like $u'(x) = g(u,x)$, $V'(t) = f(V,t)$, or $x'(t) = h(x,t)$, and the meaning is the same.

Eq. (A.1) is an **ordinary differential equation (ODE)** because $y(t)$ only depends on one variable, t, so the equation only contains "ordinary" derivatives, not partial derivatives like $\partial y(x,t)/\partial x$ or $\partial y(x,t)/\partial t$ (which give rise to partial differential equations (PDEs)). All differential equations considered in this book are ODEs. Eq. (A.1) is a **first order ODE** because it only contains first order derivatives, not higher derivatives like $y''(t)$. We will only consider first order equations in this book.

Eq. (A.1) is an ODE in **one-variable** because it contains only one unknown function, $y(t)$, which is a scalar, not a vector. First order ODEs in one-variable are sometimes called **one-dimensional ODEs** or **ODEs in one dimension**.

We will also consider **ODEs in higher dimensions**, which are sometimes called **systems of ODEs**. They can be written in vector form, like

$$\frac{d\boldsymbol{u}}{dt} = \boldsymbol{F}(\boldsymbol{u},t)$$

where $\boldsymbol{u} : \mathbb{R} \to \mathbb{R}^n$, meaning that $\boldsymbol{u}$ is a function that takes a real number as input and returns an $n \times 1$ vector. Similarly, $\boldsymbol{F} : \mathbb{R}^n \to \mathbb{R}^n$, meaning that $\boldsymbol{F}$ takes and returns a vector. We use boldface to denote vectors. Systems of ODEs can also be written as a list of n one-dimensional equations like

$$\begin{aligned}\frac{dx}{dt} &= f(x,y,t)\\ \frac{dy}{dt} &= g(x,y,t)\end{aligned}$$

where $x : \mathbb{R} \to \mathbb{R}$ and $y : \mathbb{R} \to \mathbb{R}$. This is a system of two ODEs or a two-dimensional system. We can write this in vector form by defining the vector

$$\boldsymbol{u}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix} \in \mathbb{R}^2.$$

We will initially focus on ODEs in one dimension, but will consider ODEs in higher dimensions later in the book.

ODEs like the one in Eq. (A.1) are typically paired with **initial conditions** of the form

$$y(t_0) = y_0 \tag{A.2}$$

which specifies the value at some time t_0. An ODE paired with an initial condition is called an **initial value problem (IVP)**,

$$\begin{aligned} \frac{dy}{dt} &= f(y,t) \\ y(t_0) &= y_0 \end{aligned} \tag{A.3}$$

Eq. (A.3) should be interpreted to mean that we are looking for a function, $y(t)$, satisfying

$$y'(t) = f(y(t),t) \text{ and } y(t_0) = y_0.$$

In many cases, we will take $t_0 = 0$ so the IVP will be written as

$$\begin{aligned} \frac{dy}{dt} &= f(y,t) \\ y(0) &= y_0. \end{aligned} \tag{A.4}$$

If $f(y,t)$ is continuous in t and $\partial f/\partial y$ is a continuous and bounded function of y and t then there exists a unique solution, $y(t)$, to the IVP in Eq. (A.3), at least over some interval $t \in [-T,T]$. Uniqueness means that there is only one function satisfying the IVP. The assumptions on $f(y)$ can be relaxed to some extent and, in many cases, the IVP will have a unique solution for all $t \in (-\infty,\infty)$. See a textbook on ODEs for a more in-depth discussion of the existence and uniqueness of solutions to ODEs.

In this book, we will forgo the theory of existence and uniqueness and just always assume that f is sufficiently nice that there is a unique solution to the IVP under consideration over the time interval under consideration.

For some ODEs, we can write down **closed form** solutions. A closed form solution is a solution for $y(t)$ that can be written in terms of known functions. For example, try the following exercise:

Exercise A.2.1. Find a solution the IVP

$$\begin{aligned} \frac{dy}{dt} &= 2t \\ y(0) &= 0. \end{aligned}$$

Hint: We are just looking for an anti-derivative or "indefinite integral" of $y'(t) = 2t$. Then find a solution to

$$\begin{aligned} \frac{dy}{dt} &= at \\ y(0) &= y_0. \end{aligned}$$

in terms of the parameters, a and y_0.

This approach of finding a closed form solution can work well for some simple ODEs. For example, this is the approach that we take for linear ODEs in Sections A.3 and A.5. However, for many ODEs, we cannot write down a closed form solution. For example, try to find a closed form solution to

$$\begin{aligned} \frac{dy}{dt} &= e^{\cos(y)} + t \\ y(0) &= 0. \end{aligned}$$

You will not succeed. If we cannot find a closed form solution, there are two alternative approaches we can use instead:

1. Find a numerical approximation to the solution. This approach requires plugging in specific numbers for any parameters that define the ODE and initial value. This approach is used in Section A.6.

2. Analyze properties of solutions without actually solving the equation. This approach can sometimes be more informative than a numerical solution because it can help you understand what happens for a variety of initial conditions and/or parameters. This approach is used in Section A.7.

A.3 EXPONENTIAL DECAY AS A LINEAR, AUTONOMOUS ODE

We begin by considering an IVP of the form

$$\begin{aligned} \tau \frac{dx}{dt} &= -x \\ x(0) &= x_0 \end{aligned} \tag{A.5}$$

where $x : \mathbb{R} \to \mathbb{R}$ is a scalar function and $\tau > 0$ is a scalar parameter. This ODE is **autonomous** because the right side, $f(x,t) = -x$, does not depend on t and it is **linear** because the right hand side is a linear function of x. This is one of the simplest differential equations, but understanding this equation can help to understand more complicated equations that come up in a lot of applications. You can check that the solution is given by

$$x(t) = x_0 e^{-t/\tau}. \tag{A.6}$$

Eq. (A.6) is the quintessential example of **exponential decay**: As time progresses, $x(t)$ decays exponentially toward zero. See the blue curve in Figure A.1 for a plot of a solution and `ExponentialDecay.m` for code to produce this plot.

Note that Eq. (A.6) is a solution to Eq. (A.5) for all $t \in (-\infty, \infty)$, not just $t > 0$. Therefore, Eq. (A.7) tells us about the future *and* the past values of $x(t)$.

The parameter, τ, is called the **time constant** of the decay and it determines how quickly the decay occurs. Larger τ means slower decay and smaller τ means faster decay. This can be seen by computing the proportion by which x changes over a time window of duration τ, starting at some time $t = t_1$,

$$\frac{x(t_1 + \tau)}{x(t_1)} = \frac{x_0 e^{-(t_1+\tau)/\tau}}{x_0 e^{-t_1/\tau}} = e^{-1} \approx 0.37.$$

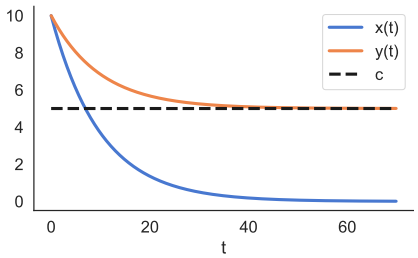

Figure A.1: Exponential decay. The exponentially decaying functions, $x(t)$ and $y(t)$, as defined by Eqs. (A.6) and (A.8). Parameters are $\tau = 10$ and $c = 5$. See `ExponentialDecay.ipynb` for Matlab code to produce this plot.

In other words, every τ units of time, x is multiplied by a factor of about 0.37. If you make the ballpark approximation $0.37 \approx 0.5$ then τ is a rough estimate of the half life of x, defined as the time it takes for x to be reduced by half. The true half life is actually closer to

$$t_{half} \approx 0.7\tau.$$

So the true half life is a little shorter than τ, but τ gives a ballpark approximation that's easier to remember and compute in your head. To see an illustration of this conclusion, note that the blue curve in Figure A.1 reaches the halfway point, $x(t) = 5$, just before time $t = \tau = 10$.

Exercise A.3.1. Derive an exact equation for the true half life and verify that it is approximately equal to 0.7τ.

Eq. (A.5) can be used to model processes that decay exponentially toward zero, but we often want to model processes that decay exponentially to other values. This is easily achieved by setting

$$y(t) = x(t) + c$$

which decays exponentially to c (since $x(t)$ decays to zero). What differential equation does $y(t)$ obey? This can be derived by computing

$$\tau\frac{dy}{dt} = \frac{dx}{dt} = -x = -(y - c) = -y + c$$

where we used the assumption that $y = x + c$. This tells us that the IVP

$$\begin{aligned} \tau\frac{dy}{dt} &= -y + c \\ y(0) &= y_0 \end{aligned} \tag{A.7}$$

should give solutions that decay exponentially to c. Indeed, you can check that the solution to Eq. (A.7) is given by

$$y(t) = (1 - e^{-t/\tau})c + y_0 = c + (y_0 - c)e^{-t/\tau}. \tag{A.8}$$

While Eq. (A.8) looks more complicated than Eq. (A.6), it is easy to verify that it just represents exponential decay toward c starting at $y(0) = y_0$. The half life now represents

the time to get halfway to c, instead of halfway to 0. See the red curve in Figure A.1 for a plot of a solution and `ExponentialDecay.m` for code to produce this plot.

Sometimes, initial conditions are given at some non-zero time, *i.e.*,

$$\begin{aligned} \tau \frac{dy}{dt} &= -y + c \\ y(t_0) &= y_0 \end{aligned} \tag{A.9}$$

It is easy to check that the solution to Eq. (A.9) is just given by sliding Eq. (A.8) over by t_0 to get

$$y(t) = c + (y_0 - c)e^{-(t-t_0)/\tau} \tag{A.10}$$

which also represents exponential decay toward c. Note that Eq. (A.10) is equivalent to Eq. (A.8) if we take $t_0 = 0$ and equivalent to Eq. (A.6) if we additionally take $c = 0$, so Eq. (A.10) is the most general form of exponential decay.

The next step will be to replace the c in Eq. (A.9) with a term that depends on time. First, we need to take a detour to introduce convolutions.

A.4 CONVOLUTIONS

Given two scalar functions, $f, g : \mathbb{R} \to \mathbb{R}$, their **convolution** is another scalar function denoted $(f * g)(t)$ defined by

$$(f * g)(t) = \int_{-\infty}^{\infty} f(s)g(t-s)ds.$$

If f or g has a bounded domain, we compute the integral assuming that they are zero outside of their domain. Convolution is commutative,

$$(f * g)(t) = (g * f)(t) = \int_{-\infty}^{\infty} g(s)f(t-s)ds$$

so you can use either of the two integrals above to represent the convolution. Convolution is also linear in each argument in the sense that

$$((f + g) * h)(t) = (f * h)(t) + (g * h)(t)$$

and

$$((cf) * g)(t) = c(f * g)(t)$$

for scalars, $c \in \mathbb{R}$.

Convolutions arise in many different contexts. For example, if X and Y are independent continuous random variables with probability density functions f_X and f_Y, then the probability density function of $Z = X + Y$ is $f_Z = f_X * f_Y$. The solution to many ordinary and partial differential equations can be written as a convolution. Convolutions are also used in signal processing, for example to smooth data. A two-dimensional version of convolutions, defined for functions $f, g : \mathbb{R}^2 \to \mathbb{R}$, is widely used for image processing and machine learning.

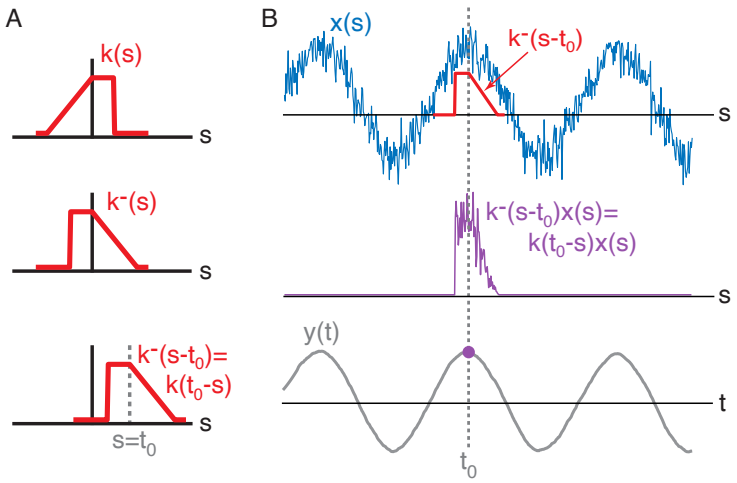

Figure A.2: Illustration of a convolution. A) A kernel, $k(s)$, is flipped to get $k^-(s) = k(-s)$, then shifted by t_0 to get $k^-(s - t_0) = k(t_0 - s)$. **B)** A signal, $x(s)$ (blue), is multiplied by the flipped and shifted kernel, $k^-(s - t_0)$ (red). The product (purple curve) is integrated to get the value of $y(t_0)$ (purple dot). Repeating this process for all values of $t = t_0$ gives the full curve, $y(t)$ (gray).

In this textbook, we focus on the convolution of a **signal**, $x(t)$, with a **kernel**, $k(t)$, which is sometimes called a **filter**. The absolute value of the kernel should have a finite integral,

$$\int_{-\infty}^{\infty} |k(s)| ds < \infty, \tag{A.11}$$

and should satisfy

$$\lim_{s \to \pm\infty} k(s) = 0.$$

Therefore, we normally think of a kernel as some kind of "bump," often centered at $s = 0$, for example $k(s) = e^{-s^2}$. The signal, $x(t)$, does not need to have a finite integral or converge to zero as $t \to \pm\infty$, but it must be bounded above and below

$$\max_t |x(t)| < \infty \tag{A.12}$$

More precisely, there needs to exist a finite number M for which $|x(t)| < M$ for all t. Unlike the kernel, the signal can be a time series that fluctuates indefinitely, for example $x(t) = \sin(t)$. The result of the convolution is given by

$$y(t) = (k * x)(t) = \int_{-\infty}^{\infty} x(s)k(t - s)ds. \tag{A.13}$$

When Eqs. (A.11) and (A.12) are satisfied, a theorem known as Young's inequality tells us that $\max_t |y(t)| < \infty$. In other words, the convolution of a kernel with a signal produces another signal. What does this new signal, $y(t)$, represent in terms of the kernel and the original signal, $x(t)$?

To get an intuition, let's think about the value of the convolution for some particular time, $t = t_0$. We have

$$y(t_0) = \int_{-\infty}^{\infty} x(s)k(t_0 - s)ds.$$

Now consider define the flipped version of the kernel: $k^-(s) = k(-s)$. In other words, $k^-(s)$ is just $k(s)$ with the time axis flipped (see Figure A.2, Left). The convolution can be written in terms of the flipped kernel as

$$y(t_0) = \int_{-\infty}^{\infty} x(s)k^-(s - t_0)ds.$$

Treated as a function of s, note that $k^-(s - t_0)$ is just the function $k^-(s)$ shifted to the right by an amount t_0 (see Figure A.2, Left). To get $y(t_0)$, we take $k^-(s - t_0)$, multiply it by an unshifted $x(s)$, then integrate the product. This operation is illustrated in Figure A.2, Right.

In summary, when performing a convolution at $t = t_0$, we are taking the integral of $x(s)$ weighted by the flipped and shifted kernel, $k^-(s - t_0)$. If we think of $k(s)$ as a bump centered at $s = 0$ then the convolution is given by flipping the bump, shifting it sideways, and taking the integral of $x(s)$ weighted by the flipped and shifted kernel.

If we additionally assume that $k(s) \geq 0$ and

$$\int_{-\infty}^{\infty} k(s)ds = 1$$

then the integral that defines $y(t_0)$ represents a **sliding, weighted average** of $x(s)$ near $s = t_0$. The shape of the bump represented by $k(s)$ determines how we weight the values of $x(s)$ near $s = t_0$ when computing the weighted average. If the the bump is wide and decays slowly, then we include values of $x(s)$ far from $s = t_0$ in our weighted average. If the bump is very narrow, then we only include values very close to $s = t_0$.

If $k(s) = 0$ for $s < 0$, *i.e.*, the bump represented by $k(s)$ lies solely on the positive time axis, then $k(t_0 - s) = 0$ whenever $s > t_0$ so we can write $y(t_0)$ as

$$y(t_0) = \int_{-\infty}^{\infty} x(s)k(t_0 - s)ds = \int_{-\infty}^{t_0} x(s)k(t_0 - s)ds.$$

This implies that the value of $y(t_0)$ only depends on values of s with $s < t_0$. In other words, $y(t)$ only depends on the past values of $x(t)$. For this reason, kernels for which $k(s) = 0$ for $s < 0$ are called **causal kernel** or, more commonly, a causal filter.

In applications of convolution, it is rare that the integral defining the convolution can be computed by hand. Instead, we typically approximate it numerically. If we wanted to use standard numerical integration, this could be a computationally expensive task because we need to approximate a new integral for each value of t. If we discretized time into 1000 bins, we would need to perform numerical integration 1000 times.

Fortunately, numerical approximations to convolutions can be computed very efficiently using an algorithm called the Fast Fourier Transform (FFT). More precisely, the FFT can be used to very efficiently perform a "discrete-time convolution." A discrete-time convolution is a convolution defined on discrete series in which the continuous-time functions, $x(t)$ and $k(t)$, in Eq. (A.13) are replaced by discrete-time sequences, x_n and k_n, and the integrals are replaced by sums. A discrete-time convolution can be used to represent a Riemann approximation to the integrals in Eq. (A.13). Therefore, a Riemann approximation to (continuous-time) convolutions can be computed efficiently using the FFT.

In Python, discrete time convolutions are implemented using the built in NumPy function, `convolve`. The code snippet below convolves a noisy signal with a Gaussian kernel to smooth the noise. The results are shown in Figure A.3 and the full code can be found in `ConvExample.ipynb`.

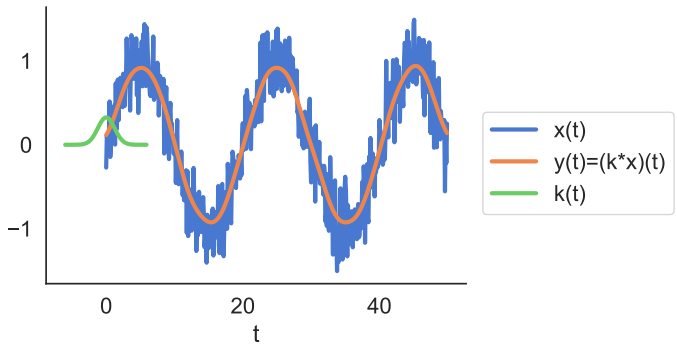

Figure A.3: Smoothing a noisy signal by convolving with a Gaussian kernel. A noisy sine wave, $x(t)$, is convolved with a Gaussian kernel, $k(t)$, and the results is a smoothed version of the signal, $y(t) = (k * x)(t)$. Code to produce this figure can be found in `ConvolutionExample.ipynb`.

```
# Discretized time
T=50; dt=.1; time=np.arange(0,T,dt)
# Define a noisy signal
x=np.sin(2*np.pi*time/20)+.25*np.random.randn(len(time))
# Define a Gaussian kernel and normalize it
sigma=3
s=np.arange(-2*sigma,2*sigma,dt)
k=np.exp(-(s**2)/sigma)
k=k/(np.sum(k)*dt)
# Perform convolution
y=np.convolve(x,k,'same')*dt
```

Let's go through this line of code step-by-step. The first line There are a few caveats for defining a kernel:

- Recall that convolution involves sliding the flipped kernel along the signal. For this to work well, the kernel must be much shorter than the signal, otherwise there's no room for sliding. To see this better, visualize sliding the red kernel along the blue signal in Figure A.2, Right. If the kernel were as wide as the signal, there would be no room to slide it. This is why we define the kernel over a much shorter discretized time vector than `time`.

- Due to the way `np.convolve` is defined, the numerical convolution only approximates the integrals defining the convolution if the kernel is centered at $t = 0$. This is why the discretized time vector `s` must be centered at 0.

- The time vector, `s`, should be wide enough to capture all of the values of `k` that are not close to zero. Since `k` is a Gaussian with width parameter `sigma`, using and interval of radius `2*sigma` is sufficient.

- If we want the convolution to represent an average then the integral of the kernel should be 1. This is why we divided `k` by the Riemann approximation to its integral in the line `k=k/(np.sum(k)*dt)`. Of course, there might also be instances

where we don't want the convolution to represent an average, so we would not need this line.

The last line performs the convolution to get the resulting signal, `y`. There are a few things to note about this line too:

- The option `'same'` tells `convolve` that the output, `y`, should be the same size as the first input, `x`. Recall that the convolution is performed by sliding the flipped kernel along the signal. Visualize sliding k along x in Figure A.3. If we slide the kernel all the way to the leftmost part of the signal, then it slides off the edge where `x` is not defined, and the same thing happens on the right edge. Specifically, it slides off when we're trying to compute $y(t_0)$ for t_0 that is within one "kernel-radius" of each edge. The kernel-radius is `2*sigma` in the code above. One solution to this problem is to slide the kernel only as far as it can go without sliding off the edge, but then we could not compute $y(t)$ at values of t that are close to the edge, so $y(t)$ would necessarily be shorter than $x(t)$ by two kernel radii. If we wanted to use this option, we'd use the option `'valid'` in place of `'same'`. The `'same'` option produces a `y` that is the same size as `x` by sliding the kernel past the edges of `x` and assuming that $x(t)$ is zero beyond its edges. This is called "zero padding" because it is equivalent to padding `x` with zeros on either side, then performing a 'valid' convolution. Problems known as **boundary effects** can arise when $x(t)$ very far from zero at its edges. These boundary effects only affect the value of $y(t)$ within a kernel-radius of each edge. As long as our kernel-radius is much shorter than our signals, these boundary effects might not be a big deal.
- We multiply the output of `convolve` by `dt` in the last line. This is because `convolve` performs a discrete-time convolution, which is defined by sums instead of integrals. Multiplying by `dt` turns these sums into Riemann sums that approximate the integrals in Eq. (A.13).

Exercise A.4.1. Run `ConvolutionExample.m` and try changing the signal, kernel, and other variables to get a feel for how convolutions work. For example, adding a large constant to `x` (moving it "up") can help visualize boundary effects. Changing the kernel radius will exaggerate these effects. Try changing the kernel to implement a causal filter. What do boundary effects look like for causal kernels? Why?

A.5 ONE-DIMENSIONAL LINEAR ODES WITH TIME-DEPENDENT FORCING

We now consider a single variable ODE driven by a time-dependent forcing term,

$$\begin{aligned} \tau \frac{dy}{dt} &= -y + I(t) \\ y(0) &= y_0 \end{aligned} \tag{A.14}$$

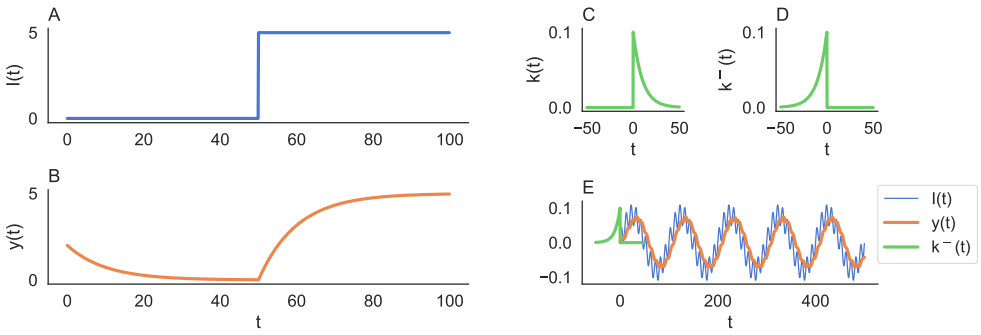

Figure A.4: Solutions to linear ODEs with time-dependent forcing. A) A step function forcing term, $I(t)$, defined by Eq. (A.15) and **B)** the resulting solution, $y(t)$, to Eq. (A.14) given by Eqs. (A.16) and (A.17) with $c = 5$, $y(0) = 2$, and $\tau = 10$. **C)** The kernel, $k(s)$, defined by Eq. (A.19) and **C)** the flipped kernel, $k^-(s) = k(-s)$. **D)** The solution, $y(t)$ under an oscillating forcing term, $I(t)$. The solution is obtained by sliding k^- along $I(t)$ and computing a weighted average (compare to Figures A.2 and A.3). Code to produce these plots can be found in `LinODE.ipynb`.

where $I(t)$ is some function of time. For simplicity, we assumed an initial condition at $t_0 = 0$ in Eq. (A.14) because solutions with initial conditions of the form $y(t_0) = y_0$ for $t_0 \neq 0$ are identical, but shifted in time.

If $I(t) = c$ then Eq. (A.14) is equivalent to Eq. (A.10), which produces exponential decay. Now, we are interested in solutions with time-dependent $I(t)$. The function, $I(t)$, is sometimes called a **forcing term.**

To get an intuition for solutions with time-dependent $I(t)$, first consider taking a step function forcing term,

$$I(t) = \begin{cases} 0 & t < t_1 \\ c & t \geq t_1 \end{cases}. \tag{A.15}$$

This function starts off at 0, then jumps to c at some time $t_1 > 0$.

Without solving the equation for this $I(t)$, we can already visualize what the solution should look like. Up until time t_1, we are just solving Eq. (A.5), so $y(t)$ will decay exponentially toward zero from the initial condition, y_0. After time t_1, we are solving Eq. (A.7), so $y(t)$ will decay exponentially toward c. In other words,

$$y(t) = y_0 e^{-t/\tau}, \quad t < t_1 \tag{A.16}$$

and $y(t_1) = y_0 e^{-t_1/\tau}$ so

$$y(t) = c + (y_0 e^{-t_1/\tau} - c)e^{-(t-t_1)/\tau}, \quad t \geq t_1. \tag{A.17}$$

We are basically gluing together two solutions with constant $I(t)$. See Figure A.4, Left for an example of this solution and `LinODEstep.m` for code to produce these plots. Technically, the solution we found is not differentiable at $t = t_1$, so Eq. (A.14) is not satisfied at $t = t_1$, but it is still satisfied everywhere else and this is the only solution that makes any sense, so we won't worry about it.

We can, of course, extend this idea to any piece-wise constant function, $I(t)$. The solution to Eq. (A.14) will just decay exponentially toward the new value of $I(t)$ every

time that $I(t)$ changes. But what about functions, $I(t)$, that change continuously in time? The same idea is valid: The solution, $y(t)$, is constantly trying to decay toward $I(t)$, but can't catch up because $I(t)$ is always changing. This is easy to see if you approximate a continuous $I(t)$ with a piecewise constant function that is constant over very short time intervals, which is exactly what we do when we define functions on discretized time intervals in Matlab.

Specifically, you can verify that the solution to Eq. (A.14) is given by

$$y(t) = y_0 e^{-t/\tau} + (k * I)(t) \tag{A.18}$$

where $*$ denotes convolution and

$$k(s) = \frac{1}{\tau} e^{-s/\tau} H(t) = \begin{cases} \frac{1}{\tau} e^{-s/\tau} & s \geq 0 \\ 0 & s < 0 \end{cases} \tag{A.19}$$

is an exponential kernel where $H(t)$ is the Heaviside step function ($H(s) = 1$ for $s \geq 0$ and $H(s) = 0$ for $s < 0$). See Figure A.4, Right for a visualization of this solution and `LinODEsin.m` for code to produce these plots.

The function, $k(s)$, is called the "Greens function" for the ODE in Eq. (A.14). Note that $k(s) = 0$ for $s < 0$, so this is a causal filter. In other words, $y(t)$ only depends on $I(s)$ for $s < t$. Also note that

$$\int_{-\infty}^{\infty} k(s) ds = 1$$

so $y(t)$ represents a running, weighted average of $I(t)$. Specifically, the value of $y(t)$ is given by the average values of $I(s)$ in the past ($s < t$) weighted by an exponential that decays with time-constant, τ. More recent values of $I(s)$ are weighted more heavily and values of $I(s)$ in the past are weighted less heavily. The parameter τ determines how quickly $y(t)$ "forgets" past values of $I(s)$. Roughly speaking, $y(t)$ is only affected by values of $I(s)$ in the interval $[t - 5\tau, 0]$ because the interval $[0, 5\tau]$ contains the almost all of the "mass" of $k(s)$, *i.e.*, because $\int_0^{5\tau} k(s) ds = 0.993 \approx 1$.

Exercise A.5.1. Consider the case where there is an additive constant in the ODE,

$$\tau \frac{dy}{dt} = -y + c + I(t)$$
$$y(0) = y_0.$$

Show that the solution is

$$y(t) = y_0 e^{-t/\tau} + c + (k * I)(t).$$

A.6 THE FORWARD EULER METHOD

We now consider a simple method for numerically approximating solutions to ODEs that is useful when we cannot find closed form solutions. We first consider ODEs in one dimension of the form

$$\begin{aligned} \frac{dx}{dt} &= f(x, t) \\ x(t_0) &= x_0. \end{aligned} \tag{A.20}$$

Numerically approximating solutions to differential equations is a major subject in applied mathematics, but most research is devoted to solving *partial* differential equations (PDEs) because ODEs, especially IVPs like Eq. (A.20), are much easier to solve numerically. The idea behind most numerical approximations is to first re-write Eq. (A.20) as

$$dx = f(x,t)dt. \tag{A.21}$$

This equation does not have a precise mathematical meaning because dx/dt does not represent a literal fraction. However, the equation can be interpreted to mean that

$$dx = x(t+dt) - x(t) \approx f(x(t),t)dt$$

when dt is sufficiently small. This can, in turn, be written as

$$x(t+dt) \approx x(t) + f(x(t),t)dt. \tag{A.22}$$

This is known as an **Euler step**. If we want to be more mathematically precise, suppose we know the value of $x(t)$ and we interpret $x(t) + f(x(t),t)dt$ as an approximation to the value of $x(t+dt)$. Then the error made by this approximation is decays to zero faster than dt decays to zero, specifically

$$\lim_{dt\to 0} \frac{\text{error}}{dt} = \frac{x(t+dt) - (x(t) + f(x(t),t)dt)}{dt} = 0. \tag{A.23}$$

How can we use Eq. (A.23) to approximation solutions to the ODE in Eq. (A.20)? We are implicitly assuming that we already know $x(t_0)$ as our initial condition. Now we can plug in $t = t_0$ into Eq. (A.22) to obtain an approximation to $x(t_0+dt)$,

$$x(t_0+dt) \approx x(t_0) + f(x(t_0),t_0)dt = x_0 + f(x_0,t_0).$$

Now that we have an approximation to $x(t_0+dt)$, we can take another Euler step to obtain an approximation to $x(t_0+2dt)$,

$$x(t_0+2dt) \approx x(t_0+dt) + f(x(t_0+dt),t_0+dt)dt$$

where we would need to plug-in our previous approximation for $x(t_0+dt)$. We can repeat this procedure to obtain approximations to $x(t_0+3dt)$, $x(t_0+4dt)$, etc. In summary, if we start with a discretized time vector starting at $t = t_0$, then we can approximate the solution to Eq. (A.20) at time points along this vector. The algorithm is arguably simpler written in Python:

```
# Initialize x
x=np.zeros_like(time)

# set initial condition
x[0]=x0

# Loop through time and perform Euler steps
for i in range(len(time)-1):
    x[i+1]=x[i]+f(x[i],time[i])*dt
```

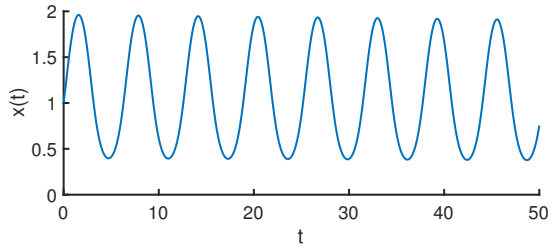

Figure A.5: Forward Euler Method Example. The numerical solution obtained by the forward Euler method with $f(x,t) = \sin(x)\cos(t)$, $dt = 0.01$, $t_0 = 0$, and $x_0 = 1$. Code to produce this figure can be found in `EulersMethod.ipynb`

or we can replace `f(x(i),time(i))` in the loop with a string of code representing the right hand side of our ODE. This procedure for approximating $x(t)$ is called the **Forward Euler method** or sometimes, just **Euler's method**. It is the simplest method for numerically approximating solutions to ODEs. Note that we can still apply Euler's method if the right hand side of our ODE does not depend on t. We just omit t and write $f(x)$. Everything else works out just the same. A more complete example of applying Euler's method is given in `EulersMethod.ipynb` and the results are plotted in Figure A.5.

> **Exercise A.6.1.** Modify the code in `EulersMethod.m` to solve an ODE for which you know a closed form solution (*e.g.*, an ODE from Sections A.3 or A.5) and compare the true solution to the approximation obtained using Euler's method for different values of dt.

If we replace Eq. (A.22) by an approximation that gives higher powers of dt in the denominator of Eq. (A.23), like dt^2, we get a **higher order** method, whereas the forward Euler method is a **first order** method. With higher order methods, you can use a larger time step, dt, and still get a smaller error. When dt is larger, the for loop will be shorter so the method will run faster. Higher order methods require smoothness assumptions on $f(x,t)$ and/or $x(t)$ so they can't be directly applied to integrate-and-fire neuron models (due to the discontinuity of $V(t)$ at reset) or to equations with Dirac delta functions like our synapse models. Firing rate models can benefit from higher order methods, but we will to stick with the forward Euler method in this textbook. See any good textbook on numerical analysis for more information on higher order methods.

The forward Euler method is easily extended to ODEs in higher dimensions or "systems of ODEs." Consider the system of two ODEs

$$\begin{aligned} \frac{dx}{dt} &= f(x,y,t) \\ \frac{dy}{dt} &= g(x,y,t) \\ x(t_0) &= x_0, \qquad y(t_0) = y_0 \end{aligned}$$

The forward Euler method for solving this system is essentially the same, but with two variables instead of one:

```
x=np.zeros_like(time)
```

```
y=np.zeros_like(time)
x[0]=x0
y[0]=y0
for i in range(len(time)-1):
   x[i+1]=x[i]+f(x[i],y[i],time[i])*dt
   y[i+1]=y[i]+g(x[i],y[i],time[i])*dt
```

Then we can do `plot(time,x)` and `plot(time,y)` to plot each solution or `plot(x,y)` to plot the trajectory of the solution in two dimensions. We can use the same approach to solve a system of any number of ODEs. If the equation is written in vector form:

$$\begin{aligned} \frac{d\boldsymbol{u}}{dt} &= \boldsymbol{F}(\boldsymbol{u}, t) \\ \boldsymbol{u}(0) &= \boldsymbol{u}_0 \end{aligned}$$

where $\boldsymbol{u} = (x, y)$ is a vector then we can do

```
u=np.zeros((2,len(time)))
u[:,0]=u0
for i in range(len(time)-1):
   u[:,i+1]=u[:,i]+F(u[:,i],time[i])*dt
```

Then we would do `plot(time,u(1,:))` and `plot(time,u(2,:))` or `plot(u(1,:),u(2,:))` to plot the solution. Euler's method ODEs in more than two dimensions is similar.

Exercise A.6.2. The Lorenz system is defined by

$$\begin{aligned} x' &= 10(y - x) \\ y' &= x(28 - z) - y \\ z' &= xy - (8/3)z \end{aligned}$$

It was originally developed as a model of convection in the atmosphere. It's not an accurate model, but it's widely studied for its mathematical properties. Use the forward Euler method with a time step of $dt = 0.01$ to solve the Lorenz system over the time interval $[a, b] = [0, 20]$ with initial conditions $x(0) = y(0) = z(0) = 1$. Plot the solution in three dimensions using the `plot3(x,y,z)` then use the rotate tool in Matlab to look at the solution from different angles after you plot it.

A.7 FIXED POINTS, STABILITY, AND BIFURCATIONS IN ONE DIMENSIONAL ODES

In this chapter, we consider ODEs where $f(x, t)$ does not depend on time,

$$\begin{aligned} \frac{dx}{dt} &= f(x) \\ x(t_0) &= x_0 \end{aligned} \tag{A.24}$$

Eq. (A.24) should be interpreted to mean that

$$x'(t) = f(x(t))$$

for all t in the domain under consideration. ODEs that can be written in the form of Eq. (A.24), where the right hand side depends only on x, are called **autonomous ODEs**. The LIF and EIF models with time-constant input, $I(t) = I_0$, can be written in this form by dividing both sides by τ and are therefore autonomous ODEs.

Autonomous ODEs have the advantage that we can understand the behavior of solutions without needing to solve them explicitly, either in closed form or numerically. This approach to studying the behavior of solutions to autonomous ODEs without computing solutions is sometimes called **dynamical systems theory**. The advantage to this approach is that we can understand the behavior of solutions across a range of different parameter values and initial conditions. Dynamical systems theory is a beautiful and fun area of mathematics to learn. This section and Section A.9 cover some of the basic topics in dynamical systems theory. For a more in-depth treatment, see the excellent book by Steven Strogatz [74], which is my favorite mathematics book.

The most important concept for understanding the behavior of solutions to autonomous ODEs is the notion of a fixed point. A **fixed point** to Eq. (A.24) is defined as a number, x^*, for which

$$f(x^*) = 0.$$

Some texts use x_0 to denote a fixed point, but we are using x_0 to denote an initial condition, so we use x^* to denote a fixed point instead.

Now consider what happens if we start at a fixed point ($x_0 = x^*$), *i.e.*, if our initial condition satisfies $f(x_0) = 0$. Let's first compute $x'(t_0)$ in that case:

$$x'(t_0) = f(x(t_0)) = f(x_0) = 0.$$

Hence, if $x(t)$ starts at a fixed point, its derivative starts at zero. Indeed, it is easy to verify that Eq. (A.24) is satisfied by the constant function

$$x(t) = x_0$$

when x_0 is a fixed point, *i.e.*, when $f(x_0) = 0$. Therefore, if an initial condition coincides with a fixed point then the solution to the ODE is a constant function. In other words,

If $x(t)$ starts at a fixed point, it stays there.

This very simple principle is our first step in understanding solutions to ODEs. To check your understanding, try this exercise:

Exercise A.7.1. Consider the IVP

$$\begin{aligned} x' &= x^2 - 1 \\ x(0) &= 1 \end{aligned}$$

Find the solution, $x(t)$, in closed form (don't overthink it; this step should be very easy if you think about the discussion above). Use Euler's method to compute a numerical solution with $dt = 0.01$ and $T = 5$. Compare the numerical solution to the closed form solution. Now think about exactly what the Euler step is doing in this example and why Euler's method gives the solution that it does.

The discussion above tells us how solutions behave when they start at fixed points, but this only gets us so far. What happens when initial conditions do not occur at fixed points? We know that solutions cannot reach a fixed point in that case, but what do they do instead?

First consider solutions to Eq. (A.24) that start with $x'(t_0) = f(x_0) > 0$ and consider what the solution is doing at any other time $t_1 \neq t_0$.

We first show that the solution can never reach a fixed point if $x'(t_0) > 0$ using a proof by contradiction. Assume that $x'(t_0) = f(x_0) > 0$ and that $x_1 = x(t_1)$ is a fixed point, *i.e.*, that $f(x_1) = 0$. Then note that $x(t)$ satisfies a new IVP of the form

$$\begin{aligned} \frac{dx}{dt} &= f(x) \\ x(t_1) &= x_1 \end{aligned}$$

where x_1 is a fixed point because $f(x_1) = 0$. But, if we assume that solutions are unique (see Section A.2 for a discussion of this assumption), then the only solution to this new IVP is the constant solution

$$x(t) = x_1.$$

This contradicts our original assumption that $x'(t_0) = f(x_0) > 0$ since the constant solution satisfied $x'(t) = 0$ for all t. Therefore, it is impossible to start at some x_0 with $f(x_0) > 0$ and reach a fixed point, x_1.

We next show that the solution can never change from increasing to decreasing using a proof by contradiction. Suppose that $x'(t_0) = f(x_0) > 0$ and $x'(t_1) < 0$ at some other time $t_1 \neq t_0$. Note that $x'(t) = f(x(t))$ is continuous whenever $x(t)$ and $f(x)$ are continuous functions (and $x(t)$ is continuous because it is differentiable). Therefore, by the intermediate value theorem, there must be some t_2 for which $x'(t_2) = 0$. But, similar to the argument above, this implies that $x(t) = x_2$ for all t, which contradicts our assumption that $x'(t_0) > 0$.

The two proofs above show that, if $x'(t_0) > 0$ then $x'(t) > 0$ for all t (since it is impossible to have $x'(t) = 0$ or $x'(t) < 0$). A similar argument shows that if $x'(t_0) < 0$ then $x'(t) < 0$ for all t. Putting this all together gives the following conclusion

> **If $x(t)$ satisfies an ODE of the form in Eq. (A.24), then the sign of $x'(t)$ does not change in time.**

Note that this conclusion relies on an assumption that $f(x)$ is continuous.

Exercise A.7.2. Consider the IVP

$$\begin{aligned} x' &= x^2 - 1 \\ x(0) &= 0 \end{aligned}$$

What is the sign of $x'(3)$? Use Euler's method to compute a numerical solution with $dt = 0.01$ and $T = 5$ to verify that $x(t)$ increases or decreases consistent with your conclusion. Now repeat both steps with the initial condition changed to $x(0) = -2$.

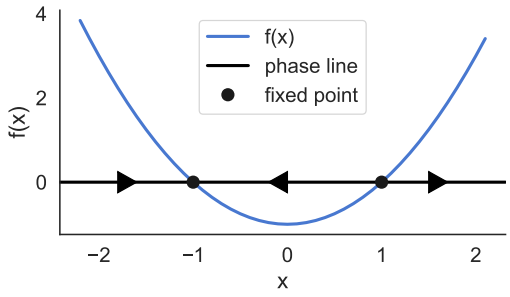

Figure A.6: Example of a Phase Line. A phase line (black line with arrows) for the ODE $x' = f(x) = x^2 - 1$. The function $f(x)$ is plotted in blue, the fixed points are indicated by black dots, and the arrows indicate whether $x(t)$ increases (rightward facing) or decreases (leftward facing) on each side of the fixed points. See `PhaseLine.ipynb` for code to generate this figure without the arrows.

With these conclusions, we can understand the behavior of solutions using a **phase line**, which is a sign chart for $f(x)$ that indicates which direction solutions go starting from which initial conditions. Fixed points are marked on the x-axis. On either side of each fixed point, we draw an arrow pointing to the right if $f(x) > 0$ in that region and pointing to the left if $f(x) < 0$ in that region. These arrows indicate which direction a solution, $x(t)$, would change if it is within that region. Figure A.6 illustrates a phase line for the ODE

$$x' = x^2 - 1$$

There are two fixed points at $x^* = \pm 1$. For initial conditions in the regions $x_0 < -1$ and $x_0 > 1$, solutions increase. For initial conditions satisfying $-1 < x_0 < 1$, solutions decrease. Compare the intuition implied by this phase line to the numerical solutions you computed in the exercise above.

Notice that both arrows surrounding the fixed point at $x^* = -1$ are pointing toward the fixed point. As a result, solutions that start near that fixed point converge toward it. This motivates the idea of stable fixed points. A fixed point is called **asymptotically stable** if solutions that start sufficiently close to the fixed point converge to the fixed point (see below for a discussion of why we need to specify "asymptotically"). More precisely:

Definition. *A fixed point, x^*, to Eq. (A.24) is called asymptotically stable if there is an $\epsilon > 0$ such that whenever $|x_0 - x^*| < \epsilon$, $\lim_{t\to\infty} x(t) = x^*$.*

Now let's look more closely at why the fixed point at $x^* = -1$ is asymptotically stable. Since x^* is a fixed point, $f(x^*) = 0$. To the left of the fixed point, $f(x) > 0$, which is why we drew a right-facing arrow. Immediately to the right of the fixed point, $f(x) < 0$, so we drew a left-facing arrow. In other words, $f(x)$ changed from positive to negative at $x^* = -1$ and this is what caused the arrows to point toward $x^* = -1$. A smooth function that changes from positive to negative at x^* is necessarily decreasing at x^*, *i.e.*, its derivative is negative. This motivates the following theorem,

Theorem. *Suppose x^* is a fixed point of Eq. (A.24) and $f'(x^*)$ exists. If $f'(x^*) < 0$ then x^* is an asymptotically stable fixed point.*

Now notice that both arrows surrounding the fixed point at $x^* = 1$ are pointing away from the fixed point. Therefore, solutions that start near $x^* = 1$ go away from it. This motivates the idea of unstable fixed points. A fixed point is **asymptotically unstable** if it is not stable, *i.e.*, if solutions can start arbitrarily close to the fixed point without converging to it. More specifically,

Definition. *A fixed point, x^*, to Eq. (A.24) is called asymptotically unstable if for any $\epsilon > 0$, there is an x_0 such that $|x_0 - x^*| < \epsilon$ and $\lim_{t\to\infty} x(t) \neq x^*$.*

The fixed point at $x^* = 1$ is asymptotically unstable because $f(x) < 0$ to the left of the fixed point and $f(x) > 0$ to the right of the fixed point, so the arrows point away from $x^* = 1$. Mirroring the discussion above, this motivates the following theorem,

Theorem. *Suppose x^* is a fixed point of Eq. (A.24) and $f'(x^*)$ exists. If $f'(x^*) > 0$ then x^* is an asymptotically unstable fixed point.*

Exercise A.7.3. Consider the ODE

$$x' = -(x+1)(x-1)(x-2)$$

Draw a phase line, find all fixed points, and classify their stability. Solve the equation numerically using Euler's method with different initial conditions to verify your results.

The reason that we needed to specify "asymptotically" in the definitions of stable/unstable above is that there are some borderline cases in which a fixed point is stable in some senses, but not others. Notably, for one-dimensional autonomous ODEs like Eq. (A.24), these only occur when $f'(x^*) = 0$. To see what can happen when $f'(x^*) = 0$, first consider the fixed point $x^* = 0$ for $f(x) = x^2$. By our definitions above, this fixed point is asymptotically unstable, but some texts classify it as "semi-stable" since $x(t) \to x^*$ for initial conditions on one side of the fixed point, but not the other. Indeed, "asymptotically unstable" is sometimes defined in a way that excludes semi-stable fixed points. Regardless of how this fixed point is classified, the behavior of solutions is still easily understood by drawing a phase line. Note that $f'(x^*) = 0$ does not imply that a fixed point is semi-stable. Consider, for example, $f(x) = x^3$.

Another example where stability is less clear is given by the trivial ODE defined by a constant zero function, $f(x) = 0$. Every x is a fixed point and the solution to Eq. (A.24) is constant, $x(t) = x_0$, for any initial condition $x(0) = x_0$. Solutions that start near a fixed point (but not at the fixed point) do not converge to that fixed point, but also do not travel away from it, *i.e.*, they stay near it. Some texts would classify these fixed points as "stable" but not "asymptotically stable" and some texts use the phrase "neutrally stable" for such fixed points.

The previous two paragraphs seem to cloud the waters around stability, but the core of the idea is still simple in almost all cases. Whenever $f'(x^*) < 0$ or $f'(x^*) > 0$, a fixed point is stable/unstable in every sense. When $f(x^*) = 0$, stability can be trickier to classify, but a phase line can still help understand the behavior of solutions. Generally speaking, **drawing a phase line is an easier and more informative way to understand properties of a fixed point than computing $f'(x^*)$.** To simplify terminology, we will

drop the "asymptotically" label and simply refer to fixed points with $f'(x^*) < 0$ or $f'(x^*) > 0$ as **stable** or **unstable** when there is no ambiguity.

One of the most useful things about the dynamical systems approach to ODEs described in this section is that we can understand the behavior of solutions to ODEs without even solving them. This is especially useful because it lets us study how solutions depend on parameter values. For example, the existence and stability of fixed points can depend on the parameters that define an ODE. A parameter value at which the number or stability of fixed points changes is called a **bifurcation**. The study of bifurcations is a central theme in dynamical systems. The most common type of bifurcation in one-dimensional ODEs is a **saddle-node bifurcation** in which two fixed points collide and disappear as a parameter is changed. The following exercise demonstrates a saddle-node bifurcation.

Exercise A.7.4. Consider the ODE

$$x' = x^2 + a$$

that depends on the parameter, a. Draw a phase line and classify all fixed points and their stability for the three cases: $a > 0$, $a = 0$, and $a < 0$. There is a saddle-node bifurcation at $a = 0$.

A.8 DIRAC DELTA FUNCTIONS

When modeling spike trains and when modeling ODEs with time-dependent forcing, we often want to model a very fast "pulse." We can define a pulse of width Δt at time $t = 0$ as

$$I(t) = \begin{cases} \frac{1}{\Delta t} & t \in [-\Delta t/2, \Delta t/2] \\ 0 & \text{otherwise} \end{cases}$$

This is just a rectangle of width Δt and height $1/\Delta t$ centered at $t = 0$. Note that

$$\int_{-a}^{a} I(t)dt = 1$$

whenever $a > \Delta t$, *i.e.*, the area under the rectangle is 1. A fast pulse is modeled by taking small Δt. As a mathematical abstraction, it is often useful to consider an infinitely fast pulse by taking $\Delta t \to 0$. However, note that $I(0) \to \infty$ in this limit, so $I(t)$ does not converge to a function in the usual sense. Dirac delta functions give us a way to work with infinitely fast pulses and treat them like functions.

The **Dirac delta function**, $\delta(t)$, is a "function" defined by $\delta(t) = 0$ for $t \neq 0$ and

$$\int_{-a}^{a} \delta(t)dt = 1$$

for any $a > 0$. We will use the term **delta function** as a shorthand for Dirac delta function.

We put "function" in quotation marks above because the Dirac delta function is not actually a function in the strict sense of the word. Any real function that satisfies

$\delta(t) = 0$ for $t \neq 0$ would also satisfy $\int_a^b \delta(t)dt = 1$ for all $a, b \in \mathbb{R}$. Intuitively, we can think of a Dirac delta function as an infinitely narrow and infinitely tall pulse, so "$\delta(0) = \infty$." There is a mathematically precise way to define Dirac delta functions as a "distribution," "generalized function," or "measure." Under these definitions, the delta function can only be evaluated inside of an integral, so we never need to ask about the value of $\delta(0)$.

The delta function can also be interpreted as a Gaussian probability density with mean zero and standard deviation zero,

$$\delta(t) = \lim_{\sigma \to 0} \frac{1}{\sigma\sqrt{2\pi}} e^{-t^2/\sigma^2}$$

and this interpretation also represents the limit of an infinitely narrow pulse at $t = 0$.

We often want to consider a pulse centered at some $t \neq 0$. To do this, we can just translate $\delta(t)$ to get $\delta(t - t_1)$, which has the property

$$\int_a^b \delta(t - t_1) = \begin{cases} 1 & a < t_1 < b \\ 0 & \text{otherwise} \end{cases}$$

and should be interpreted as an infinitely narrow pulse centered at $t = t_1$. It is sometimes useful to define the translated delta function as

$$\delta_{t_1}(t) = \delta(t - t_1).$$

Dirac delta functions have the following important property

$$\int_{-a}^{a} x(t)\delta(t)dt = x(0)$$

for $a > 0$ and, more generally,

$$\int_a^b x(t)\delta(t - t_1)dt = \int_a^b x(t)\delta_{t_1}(t)dt = x(t_1)$$

when $a < t_1 < b$. Indeed, in a more mathematically rigorous setting, this is essentially the definition of the delta function. As a consequence, delta functions are identities under convolution

$$(x * \delta)(t) = x(t)$$

and convolution with a translated delta functions implements a translation

$$(x * \delta_{t_1})(t) = x(t - t_1). \tag{A.25}$$

Now consider what happens when a Dirac delta function is the forcing term in a one-dimensional linear ODE,

$$\begin{aligned} \tau\frac{dx}{dt} &= -x + \delta(t - t_1) \\ x(0) &= 0 \end{aligned} \tag{A.26}$$

and let's assume that $t_1 > 0$. The first way to approach this problem is to use the solution derived in Section A.5. Specifically, from Eq. (A.18), we have

$$x(t) = (k * \delta_{t_1})(t)$$

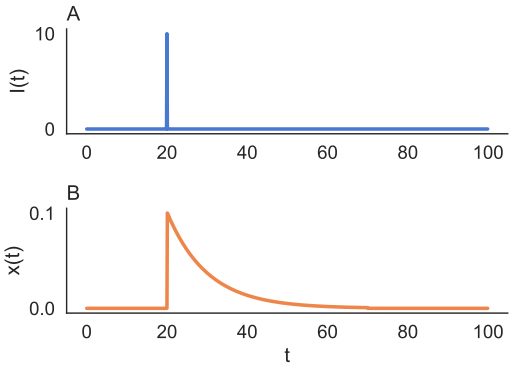

Figure A.7: Numerical representation of a Dirac delta function and driving a linear ODE. A) A numerical representation of $I(t) = \delta(t - t_1)$ with $t_1 = 20$ using a bin size of $dt = 0.1$. **B)** Numerical solution of Eq. (A.26) obtained using the $I(t)$ from A. Code to produce this figure can be found in `DiracDeltaFunctions.ipynb`.

where $\delta_{t_1}(t) = \delta(t - t_1)$ is the forcing term and $k(s) = \frac{1}{\tau}e^{-s/\tau}H(s)$ is an exponential kernel with $H(t)$ the Heaviside step function. From Eq. (A.25), therefore, we have

$$x(t) = \frac{1}{\tau}e^{-(t-t_1)/\tau}H(t - t_1) = \begin{cases} \frac{1}{\tau}e^{-(t-t_1)/\tau} & t \geq 0 \\ 0 & t < 0 \end{cases}. \tag{A.27}$$

In other words, $x(t) = 0$ for $t < t_1$, then jumps up to $1/\tau$ at time $t = t_1$, then decays back toward zero for $t > t_1$. Put more simply, $x(t)$ is just the exponential kernel, $k(t)$, shifted in time by t_1.

So far, we have considered the mathematical definition of a Dirac delta function, but how should we represent it numerically in code? If we are using discretized time with a step size of dt, we can represent a delta function by placing $1/dt$ in the associated time bin. For example, to represent the signal $I(t) = \delta(t - t_1)$, we would do

```
time=np.arange(0,T,dt)
I=np.zeros_like(time)
I[np.int(t1/dt)]=1/dt
```

This code assumes that $0 \leq t_1 < T$. Figure A.7A shows a numerical representation of a delta function with $t_1 = 20$ and $dt = 0.1$. This way of defining Dirac delta functions interacts nicely with Riemman integration, discrete convolutions, and the forward Euler method. For example, if we use the Riemman integral,

```
integral0T=sum(I)*dt
```

to approximate $\int_0^T I(t)dt$ then we will get the correct result because the `*dt` in this line of code cancels with the `1/dt` in the associated bin in I to give 1. To see how numerical representations of Dirac delta functions interact with discrete convolutions and the forward Euler method, let's consider the numerical solution of Eq. (A.26). The forward Euler method would look like

```
x[0]=0
for i in range(len(time)-1):
    x[i+1]=x[i]+dt*(-x[i]+I[i])/tau
```

where `I` is defined in the code snippet above. In this loop, `x[i+1]` will remain zero until we reach the time bin `i==np.int(t1/dt)` corresponding to time t_1. After that time bin, `x` will jump up to `x[i+1]=dt*(1/dt)/tau` which is equal to `1/tau`. After that, `x` will decay exponentially back to zero. This is exactly the behavior of the true solution in Eq. (A.27). Similarly, we can compute the solution from Eq. (A.26) using a discrete convolution

```
s=np.arange(-5*tau,5*tau,dt)
k=(1/tau)*np.exp(-s/tau)*(s>=0)
x=np.convolve(I,k,'same')*dt
```

which gives similar results. Figure A.7B shows a numerical solution to Eq. (A.26). The file `DiracDeltaFunctions.ipynb` contains code to produce Figure A.7B using the forward Euler method and using a numerical convolution.

A.9 FIXED POINTS, STABILITY, AND BIFURCATIONS IN SYSTEMS OF ODES

In Section A.7, we looked at fixed points and stability for autonomous ODEs in one dimension. We now extend some of those results to systems of ODEs, which can be viewed as ODEs in higher dimensions ($n > 1$). Recall that an autonomous system of ODEs can be written in the form

$$\frac{d\boldsymbol{u}}{dt} = \boldsymbol{F}(\boldsymbol{u}) \tag{A.28}$$

where $\boldsymbol{u} : \mathbb{R} \to \mathbb{R}^n$ and $\boldsymbol{F} : \mathbb{R}^n \to \mathbb{R}^n$. Given a system of ODEs and an initial condition, $\boldsymbol{u}(0) = u^0$, the system has a unique solution satisfying the initial condition (under some assumptions on $\boldsymbol{F}$). A system can also be written as a list of one-dimensional equations, for example if $n = 2$ then

$$\begin{aligned} \frac{\partial x}{\partial t} &= f(x, y) \\ \frac{\partial y}{\partial t} &= g(x, y) \end{aligned} \tag{A.29}$$

where $x, y\mathbb{R} \to \mathbb{R}$ and the two conventions are related by defining

$$\boldsymbol{u}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}.$$

As in one-dimensional ODEs, a **fixed point** for the system in Eq. (A.28) is again defined by a value, $\boldsymbol{u}^* \in \mathbb{R}^n$ satisfying

$$\boldsymbol{F}(\boldsymbol{u}^*) = 0$$

and fixed points again satisfy the property that

If $\boldsymbol{u}(t)$ starts at a fixed point, it stays there.

In other words, if $\boldsymbol{u}^0 = \boldsymbol{u}^*$ where $\boldsymbol{F}(\boldsymbol{u}^*) = 0$ then $\boldsymbol{u}(t) = \boldsymbol{u}^*$ for all t. Additionally, if a solution does not start at a fixed point then it cannot ever equal one. In other words, if $\boldsymbol{F}(\boldsymbol{u}^0) \neq 0$ then $\boldsymbol{F}(\boldsymbol{u}(t)) \neq 0$ for all t.

The definitions of stable and unstable fixed points are also the same for systems as for one-dimensional ODEs: A fixed point is stable if initial conditions sufficiently close to the fixed point converge to it, and it is unstable otherwise (see Section A.7 for more detailed definitions).

However, determining whether a fixed point is stable or unstable is more complicated for systems of ODEs. We will begin by considering **linear systems of ODEs** which are systems in which $F(\boldsymbol{u}) = A\boldsymbol{u}$ is a linear function, so

$$\begin{aligned} \frac{d\boldsymbol{u}}{dt} &= A\boldsymbol{u} \\ \boldsymbol{u}(0) &= \boldsymbol{u}^0 \end{aligned} \tag{A.30}$$

for some $n \times n$ matrix, A. Linear systems always have a fixed point at the zero vector,

$$\boldsymbol{u}^* = \mathbf{0}.$$

If A is non-singular (*i.e.*, invertible), then this is the only fixed point. The stability of the fixed point at zero is determined by the eigenvalues of A. Recall that eigenvalues are numbers, λ, for which there exists a vector, $\boldsymbol{v}$, satisfying

$$A\boldsymbol{v} = \lambda\boldsymbol{v}.$$

The vector, $\boldsymbol{v}$, is called the eigenvector associated with the eigenvalue, λ. Eigenvalues and eigenvectors can be real, imaginary, or complex. Complex eigenvalues always come in conjugate pairs, $\lambda = a \pm bi$. In general, an $n \times n$ matrix has n eigenvalues. To compute eigenvalues, note that $A\vec{v} = \lambda v$ implies that $[A - \lambda Id]\vec{v} = 0$ so that $A - \lambda Id$ is a singular matrix (where Id is the $n \times n$ identity matrix). Hence, we only need to solve $\det(A - \lambda Id) = 0$ for λ. Recall that, for a 2×2 matrix, the determinant is:

$$\det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = ad - bc.$$

In NumPy, eigenvalues and eigenvectors can be found by

```
lam,v=np.linalg.eig(A)
```

which returns a vector of all eigenvalues in `lam` and a matrix of eigenvectors in `v`.

The stability of linear systems is determined by the sign of the real part of the eigenvalues of A, as explained in the following theorem,

Theorem. *If **all** eigenvalues of A have **negative real part** then the fixed point at zero is **stable** for Eq. (A.30). If **at least one** eigenvalue of A has **positive real part** then the fixed point at zero is **unstable** for Eq. (A.30).*

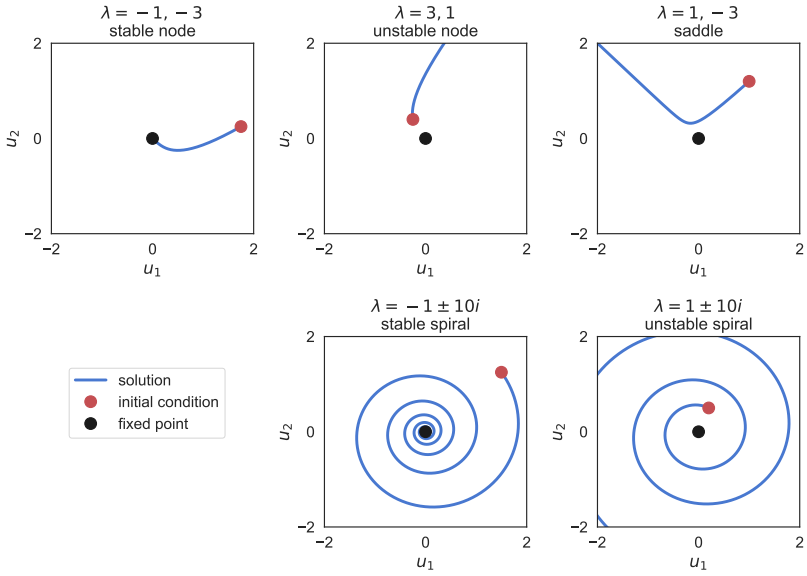

Figure A.8: Solutions of linear systems of ODEs with different eigenvalue patterns. The system in Eq. (A.30) solved using the forward Euler method for five different matrices, A. Eigenvalues, λ appear in the titles. Code to produce this figure can be found in `LinearSystemsOfODEs.ipynb`.

We will not consider systems for which A has eigenvalues with zero real part, which are singular matrices.

Figure A.8 shows numerical solutions of linear systems in $n = 2$ dimensions for various A with different eigenvalues. Two systems are stable and three unstable. Note that the qualitative appearance of the solutions are quite different: In some cases, the solutions spiral in or out, in other cases they do not. These properties are also determined by the eigenvalues. In particular,

> Consider Eq. (A.30) in $n = 2$ dimensions. If the eigenvalues of A are complex, then solutions draw spirals in the plane.

When the eigenvalues have negative real part, this is called a **stable spiral** or **spiral sink** and when they have positive real part, this is called an **unstable spiral** or **spiral source**. Spirals in the $\boldsymbol{u}(t)$ plane translate to oscillations of the individual components, $\boldsymbol{u}_1(t)$ and $\boldsymbol{u}_2(t)$, as shown in Figure A.9.

When all eigenvalues are real and negative, solutions just decay exponentially to zero and the system is called a **stable node** or **nodal sink**. When all eigenvalues are real and positive, solutions grow exponentially and the system is called an **unstable node** or **nodal source**. When all eigenvalues are real, but have different signs, solutions decay in some directions, but almost all solutions eventually grow exponentially. This is called a **saddle**. All of these solution types can be found in Figures A.8 and A.9.

> **Exercise A.9.1.** Come up with an arbitrary 2×2 matrix on your own. Compute the eigenvalues by hand and determine the stability and classification of the system. Now use the forward Euler method to solve the system numerically and compare the solution to what you expected from your classification. Try this a couple of times.

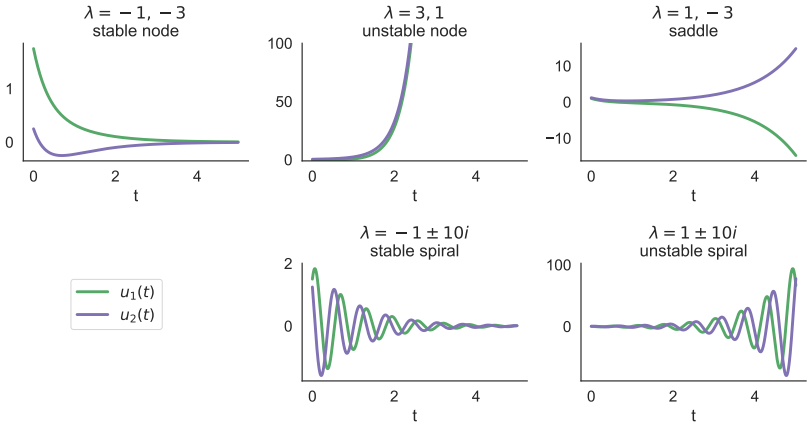

Figure A.9: Solutions of linear systems of ODEs with different eigenvalue patterns. Same as Figure A.8, but $\boldsymbol{u}_1(t)$ and $\boldsymbol{u}_2(t)$ are plotted as functions of t. Code to produce this figure can be found in `LinearSystemsOfODEs.ipynb`.

Two-dimensional systems can be classified without even computing the eigenvalues. First note that the determinant of a matrix is the product of its eigenvalues and the trace of a matrix is the sum of the eigenvalues. For $n = 2$,

$$\det(A) = \lambda_1\lambda_2, \quad \mathrm{Tr}(A) = \lambda_1 + \lambda_2. \tag{A.31}$$

Recall that the trace is defined as the sum of the diagonal entries,

$$\mathrm{Tr}\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = a + d.$$

From Eq. (A.31), it can be shown that a two-dimensional system has eigenvalues with negative real part (and is therefore stable) if and only if

$$\det(A) > 0 \text{ and } \mathrm{Tr}(A) < 0.$$

We can also use Eq. (A.31) to classify the type of system. If the determinant is negative, then the eigenvalues must be real (since $(a + bi)(a - bi) = a^2 + b^2 \geq 0$) and must have opposite sign, so the system must be a saddle. Distinguishing between nodes and spirals is a little bit trickier. From Eq. (A.31), it can be shown that

$$\lambda_{1,2} = \frac{T \pm \sqrt{T^2 - 4D}}{2} \tag{A.32}$$

where $T = \mathrm{Tr}(A)$ and $D = \det(A)$. From this, we can show that so $T^2 > 4D$ produces real eigenvalues (a node), and $T^2 < 4D$ means complex eigenvalues (a spiral). Putting this together, we can make a picture of the **Trace-Determinant plane**, in which the five different types of solutions discussed above are represented by five different regions on the plane of all T and D values (Figure A.10). Note that Eq. (A.32) can also be used to compute the eigenvalues directly, but it is only valid in $n = 2$ dimensions.

Exercise A.9.2. Repeat the previous exercise, but use the trace and determinant to determine the stability and classification without computing eigenvalues.

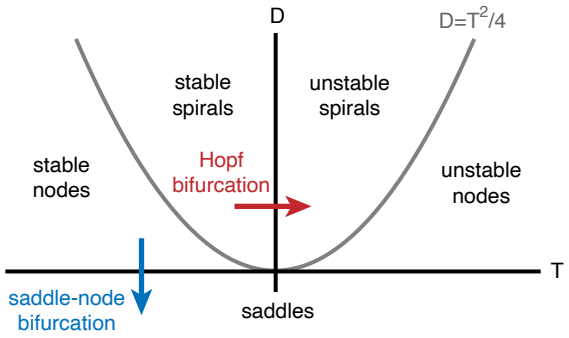

Figure A.10: The trace-determinant plane. The plane of all values of the trace (T) and determinant (D) of the matrix A from Eq. (A.30). The plane is split into five regions by the lines $T = 0$ and $D = 0$ along with the curve $D = T^2/4$. Each of these regions represents a different type of solution to Eq. (A.30). A Hopf bifurcation occurs during a transition between a stable and unstable spiral, *i.e.*, when T changes sign with $D > 0$.

We have so far only considered stability for *linear* systems of ODEs. It turns out that we can use what we learned for linear systems to determine the stability of nonlinear systems of ODEs.

Recall that for one-dimensional ODEs, stability is determined by the sign of $f'(x)$ at the fixed point. A similar result holds for systems of ODEs, but we need to generalize the notion of a derivative to a vector function $\boldsymbol{F} : \mathbb{R}^n \to \mathbb{R}^n$. The derivative of such a function is a matrix called a **Jacobian matrix**. Specifically, the Jacobian matrix of F at a point u^* is defined by an $n \times n$ matrix, J, with entries defined by

$$J_{jk} = \left.\frac{\partial \boldsymbol{F}_j}{\partial \boldsymbol{u}_k}\right|_{\boldsymbol{u}=\boldsymbol{u}^*}$$

In other words, the j,kth entry of the Jacobian is the derivative of the j entry of $\boldsymbol{F}$ with respect to the kth entry of its input, evaluated at the fixed point. This equation is easier to understand when we write it out for a system written in the form of Eq. (A.29). For this system, the Jacobian is given by

$$J = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{bmatrix}$$

where the derivatives are evaluated at the fixed point in question. Jacobian matrices are derivatives in the sense that they represent the best linear approximation to F at the fixed point. For this reason,

> The solution of a nonlinear system like Eq. (A.28) or Eq. (A.29) near its fixed point looks like the solution to the system
>
> $$\frac{d\boldsymbol{u}}{dt} = J\boldsymbol{u}$$
>
> near zero.

In particular, this means that the eigenvalues of the Jacobian tell us about the stability of the fixed point,

Theorem. *Consider the system given by Eq. (A.28) or Eq. (A.29) and let J be the Jacobian matrix computed at a fixed point. If* ***all*** *eigenvalues of J have* ***negative real part*** *then the fixed point is* ***stable****. If* ***at least one*** *eigenvalue of J has* ***positive real part*** *then the fixed point* ***unstable****.*

Moreover, if the Jacobian has complex eigenvalues then solutions will tend to draw spirals, just like in the associated linear system.

Exercise A.9.3. Consider the nonlinear system

$$\frac{\partial x}{\partial t} = -(1-y^2)x - y$$
$$\frac{\partial y}{\partial t} = x$$

Find the unique fixed point, find its stability, and classify it (node, spiral, or saddle). Then use the forward Euler method to simulate the system. Try different initial conditions and adjust the duration of the solution (T) to make sure you can see the asymptotic behavior.

For one-dimensional ODEs, we looked at saddle-node bifurcations in which changing a parameter of the ODE changed the stability of a fixed point. Systems of ODEs can exhibit saddle node bifurcations too when the sign of an eigenvalue changes sign to switch between a stable node and a saddle (hence the name). Looking a the trace-determinant plane we can see that this happens when D changes sign with $T > 0$ (Figure A.10).

Systems of ODEs can produce a different kind of bifurcation that cannot be produced by one-dimensional ODEs. When the stability of a spiral changes, it is called a **Hopf bifurcation**. Looking a the trace-determinant plane we can see that this happens when T changes sign with $D > 0$ (Figure A.10). When a stable spiral becomes unstable, it usually gives rise to a **stable limit cycle**, which is a periodic solution to which nearby solutions converge. The periodicity of limit cycles implies that they generate oscillations.

Exercise A.9.4. Consider the nonlinear system

$$\frac{\partial x}{\partial t} = -c(1-y^2)x - y$$
$$\frac{\partial y}{\partial t} = x$$

where μ is a parameter. Note that when $c = 1$, this is the same system considered in the previous exercise. Determine a value of c at which a Hopf bifurcation occurs. Then use the forward Euler method to simulate the system with different values of c on either side of the bifurcation. Plot the solution in the x-y plane (like the plots in Figure A.8), but also plot the solutions as functions of time (like the plots in Figure A.9) to see the oscillations.

B ADDITIONAL MODELS AND EXAMPLES

Warning: Text and figures in this appendix are incomplete.

B.1 MODELING ION CHANNEL CURRENTS

In Section 1.1, we used the leaky integrator model to approximate the membrane potential of a neuron below the action potential threshold. The leaky integrator model combines the effects of multiple ion channels and pumps into a single leak current. In this section, we describe how to model the effects that individual types of ion channels have on the membrane potential. The presentation here loosely follows that in Dayan and Abbott's textbook [1], which gives more detail.

There are two general categories of ion channels: **Ungated (passive) channels** are always open, so they passively allow ions to pass through all the time. **Gated (active) channels** open and close. The permeability of membrane to a particular ion depends on the number of ion channels open and therefore on the probability that each channel is open. The open probability of gated ion channels can depend on many things, for example the membrane potential, light, and the concentration of certain ions, neuromodulators, or neurotransmitters.

There are two factors that contribute to the flow of ions across an open channel: the concentration gradient and the electrical gradient. The effect of the concentration gradient is simple: When the concentration of a certain type of ion is greater inside the cell than outside the cell, then the concentration gradient tries to push this type of ion out. When the concentration is higher outside the cell, the concentration gradient tries to pull them in. In other words, the concentration gradient tries to equilibrate the inside and outside concentrations of each type of ion.

However, open channels do not simply cause inside/outside ion concentrations to equilibrate because electrical potential also plays a role. As mentioned before, the membrane potential is usually negative. This negative potential tries to pull positive ions inside the cell and push negative ions out. The strength of this force depends on the neuron's membrane potential: A more hyperpolarized membrane potential will push and pull the ions more strongly.

An ion's **reversal potential** is the membrane potential at which the effects of the concentration gradient and the electrical gradient cancel out. This is also sometimes called the **equilibrium potential** or **Nernst potential**. The reversal potential for ion type a is typically denoted E_a (*e.g.*, E_{Na} for sodium channels). When the membrane potential is at the reversal potential ($V = E_a$), the channel produces no current because the effect of the electrical and diffusive forces cancel. In other words, the net flow of ions is zero (just as many flow in as out).

What about when $V \neq E_a$? In this case, there will be a flow of ions, *i.e.*, a current. Ion channels act as resistors. They allow ions to pass through, but with some resistance

(because ions only flow so fast). You might have learned Ohm's law, $V = -IR$, in a physics class. Ohm's law can be re-written as

$$I = -gV$$

where I is the current across a resistor and $g = 1/R \geq 0$ is its conductance. However, this equation is for purely electrical circuits. The effects of concentration gradients changes the equation. In particular, the current should be zero when $V = E_a$, not necessarily when $V = 0$. The equation that describes the currents across an ion channel is

$$I_a = -g_a(V - E_a).$$

When $V > E_a$, there is a negative current which pulls V back down toward E_a. When $V < E_a$, there is a positive current that pulls V up toward E_a. So the membrane potential is pulled toward E_a by the ion channel. Therefore, ion channels with reversal potentials well above rest ($E_a > -55$mV) are generally depolarizing while those with reversal potentials well below rest ($E_a < -77$mV) are hyperpolarizing. Reversal potentials close to rest can be hyperpolarizing, depolarizing, or neutral depending on the value of V.

Note that each ion channel induces a current of the form given above. The currents induced by several parallel ion channels combine additively, so

$$I_{total} = -\sum_a g_a(V - E_a)$$

The effect of individual channels of the same type can be lumped together, so we really have one term in the sum for each type of ion channel, a. Some types of ion channels only (or at least mostly) allow a certain type of ion through, for example only sodium or potassium. Other ion channels allow multiple types of ions through. The types of ions that pass through an open ion channel is called the channel's **ion selectivity**.

There are numerous types of ion channels with different reversal potentials, etc. and there are also ion pumps that are not accounted for by this equation. We don't want to model all of these explicitly because it would give us a huge equation with an enormous number of parameters. We don't know the values of all of these parameters and we don't even know all of the ion types to include for a particular neuron. Instead of trying to model all of the ion channels, we can lump them all together into one effective current, $I_L = -g_L(V - E_L)$ to get the leaky integrator model from Section 1.1.

However, what if we want to model the effects of an individual type of ion channel? This could be important, for example, if we are modeling an experiment or drug that modifies some property of an ion channel. It will also be important in the next section when we model action potential generation. We can model individual ion channel types simply by adding them onto the leaky integrator model to get

$$C_m \frac{dV}{dt} = -g_L(V - E_L) + I_x(t) - g_a(V - E_a)$$

or, if we don't want to model the capacitance directly, we can use

$$\tau_m \frac{dV}{dt} = -(V - E_L) + I_x(t) - g_{a/L}(V - E_a)$$

where $g_{a/L} = g_a/g_L$ is just the ratio of the two conductances which can be chosen without specifying g_a or g_L individually. We could also add an ion current onto the EIF model if we wanted to include spiking,

$$\tau_m \frac{dV}{dt} = -(V - E_L) + I_x(t) + De^{(V-V_T)/D} + I_x(t) - g_{a/L}(V - E_a)$$
$$V(t) > V_{th} \Rightarrow \text{spike at time } t \text{ and } V(t) \leftarrow V_{re}.$$

For a passive ion channel, the models above are complete. We simply need to choose values of E_a and $g_a > 0$ or $g_{a/L} > 0$that match experimental recording.

For gated ion channels, the value of g_a is not constant, but depends on the number of open channels. Specifically,

$$g_a(t) = g_{a,max} p_a(t)$$

where $g_{a,max}$ is the conductance if they were all open and $p_a(t) \in [0, 1]$ is the proportion that are open. Ion channels open and close stochastically. If the cell membrane contains many ion channels and each channel is in an open state with independent probability, then the proportion that are open is approximately given by the probability that any given channel is open. To this end, we will interpret $p_a(t)$ as the probability that a channel is open, *i.e.*, we are modeling the mean value of $g_a(t)$ and ignoring its variability. A more realistic model would treat $g_a(t)$ as a stochastic process.

Many different factors can affect the open probability, $p_a(t)$, of a particular type of gated ion channel. In the next section, we will consider voltage-gated ion channels, which are affected by the membrane potential. The following exercise introduces a simpler example.

Exercise B.1.1. Optogenetics is a method for genetically modifying neurons to have ion channels that are sensitive to light. Some of the first optogenetic techniques were developed by genetically modifying animal neurons to express channelrhodopsins, which are a type of light-sensitive ion channel first discovered in algae. When a specific type of neuron expresses the channelrhodopsin, experimenters can evoke currents in that type of neuron by shining a laser on the brain. Optogenetics help neuroscientists study the effects of different neuron types in circuit function, and might someday be used for prosthetics or therapeutic purposes. Optogenetic ion channels can be engineered to have different ion selectivities. Simulate an EIF model with channelrhodopsin channels that are selective for sodium (Na^+). As a very simple model of the open probability of the channelrhodopsin channels, use

$$\tau_{ChR} \frac{dp_{ChR}}{dt} = -p_{ChR} + L(t)$$

where $L(t) = 1$ when the light is on and $L(t) = 0$ when the light is off. Use $\tau_{ChR} = 2\text{ms}$, $E_{ChR} = E_{Na} = 50\text{mV}$, $I_x(t) = 0$, and experiment with different values of $g_{ChR/L,max} > 0$ to get a steady-state firing rate of around 20Hz when the light is on.

B.2 THE HODGKIN–HUXLEY MODEL

In Section 1.2, we used the EIF model to approximate action potential generation. The model was derived as a rough approximation of the membrane potential dynamics during the inflow and outflow of sodium and potassium ions during an action potential. In this section, we derive a more detailed model of action potential generation famously developed by Alan Hodgkin and Andrew Huxley in the 1950s.

Hodgkin and Huxley began their work by performing experiments on the squid giant axon (not to be confused with a giant squid axon) to study its electrical properties. They ultimately derived a mathematical description of an action potential, through a series of three papers. They won the Nobel Prize in 1963 for their work.

The model they constructed is now referred to as the **Hodgkin-Huxley (HH) model**. For modeling other types of neurons in other species, parameters can be changed and other channel types can be added. Models that have the same basic format as the HH model, but with different parameters or added ion channels are called **Hodgkin-Huxley style models** and are widely used today. Most neuron models used today can be viewed as either simplifications or extensions of the HH model. This chapter provides a brief review of the original HH model. A more in-depth discussion can be found in many different textbooks and websites.

In the previous section, we wrote the conductance of an active channel in terms of its open probability, but we did not describe how to properly model the open probability. The opening and closing of ion channels depends on complicated biophysical processes. A common approximation is to assume that the channel being open requires the opening of one or more "gates." The opening of a gate requires multiple identical, independent processes to all be in an "active" state. We will refer to these processes as sub-gates. Consider an ion channel with one gate and k sub-gates. If $n(t)$ is the probability that each sub-gate is active, then the open probability of the channel is

$$p_a(t) = n^k(t).$$

The variable $n(t)$ is called an activation variable or gating variable. In reality, the opening and closing of ion channels is more complicated and parameters like k are typically fit to data instead of being derived from biophysical properties of the channel, but the model above does a good job of describing ion channel currents in many cases.

For voltage-gated ion channels, n depends on V, but V changes in response to changes in n since open channels induce a current. This makes it difficult to study the effects of channels, but also imparts neurons with most of their interesting and useful properties. To start with, we will get around this bi-directional dependence by assuming the neurons are "clamped" to a fixed voltage.

In actual recordings, the voltage can be held fixed using **voltage clamp**, which is a recording protocol in which the membrane potential (voltage) is held at specified fixed value by an electrode. This allows currents to be measured at fixed voltage. Voltage clamp is used to study properties of ion channels, etc.

The first voltage-dependent ion channel we will study is the **voltage-dependent potassium ($\mathbf{K}_v$) channel**. For the K_v channel, there is one gate with $k = 4$ sub-gates so

$$I_K = -\overline{g}_K n^4 (V - E_K).$$

where $P_0 = n^4$ is the probability that a K_v channel is open, equivalently the proportion of K_v channels that are open. Hodgkin and Huxley fit this model to experiments and

found: $\overline{g}_K = 36$ mS/cm^2, $E_K = -77$ mV (mS=millisiemens is a measure of conductance). Note that, since $\overline{g}_K$ measures conductance per unit area, current is also measured per unit area. This will be true throughout our calculations, but we will still just call them "conductance" and "current" and leave off the "per unit area"

K_v is a voltage-dependent channel (hence, the v subscript), so we also need to model the dependence of the gating variable, $n(t)$, on V. Recall that we are currently treating V as a constant since we are assuming the cell is in voltage clamp. The sub-gates open and close stochastically and the rate at which they open and close depends on voltage. Define $\alpha_n(V)$ to be the rate at which closed sub-gates open and $\beta_n(V)$ to be the rate at which open sub-gates close when the membrane potential is at V. This can be used to write a differential equation of the form

$$\frac{dn}{dt} = \alpha_n(V)(1-n) - \beta_n(V)n. \tag{B.1}$$

The first term represents the rate at which newly opened sub-gates are added (by closed sub-gates becoming open). The second term quantifies the rate at newly opened sub-gates are added (by open sub-gates becoming closed). One way of deriving and interpreting this equation is that a single sub-gate is a stochastic process, $N(t)$, called a continuous-time Markov chain that switches between two states, $N(t) \in \{0,1\}$, representing closed and open respectively. Under this model, $n(t) = \Pr(N(t) = 1) = E[N(t)]$, obeys Eq. (B.1) which is sometimes called a mean-field equation or rate equation and is closely related to the master equation for the Markov chain. Hence, $n(t)$ models the probability of a single sub-gate being open, but also approximates the proportion of sub-gates that are open whenever there is a large number of independent sub-gates.

While Eq. (B.1) is easy to interpret biophysically, it can be re-written in a way that is easier to interpret dynamically,

$$\tau_n(V)\frac{dn}{dt} = n_\infty(V) - n$$

where

$$\tau_n(V) = \frac{1}{\alpha_n(V) + \beta_n(V)}.$$

and

$$n_\infty(V) = \frac{\alpha_n(V)}{\alpha_n(V) + \beta_n(V)}$$

When V is clamped (i.e., constant in time), this is just a one-dimensional linear ODE for exponential decay ($\tau_n dn/dt = -n + n_\infty$), which has the solution (see Appendix A.3)

$$n(t) = (n_0 - n_\infty)e^{-t/\tau_n} + n_\infty.$$

In other words, when V is clamped, $n(t)$ decays exponentially to $n_\infty(V)$ with time constant $\tau_n(V)$. If $\tau_n(V)$ is large, it converges slowly. If $\tau_n(V)$ is small, it converges quickly.

The dependence of α_n and β_n on V is related to complicated molecular and cellular processes. Instead of modeling them explicitly, Hodgkin and Huxley empirically fit the dependence to experimental observations to get equations for α_n and β_n in terms of V.

$$\alpha_n(V) = \frac{0.01(V+55)}{1 - e^{-0.1(V+55)}}$$

$$\beta_n(V) = 0.125e^{-0.0125(V+65)}.$$

which have units 1/ms=kHz and V is measured in mV.

The other type of ion channel responsible for action potentials is the voltage-dependent sodium (Na_v) channel. Voltage dependent sodium channels are more complicated than potassium channels because they produce a **transient conductance** meaning that as V increases, they can switch from open to closed, then back to open again. This is because they essentially have two types of gates. When V is increased, one opens quickly while the other closes slowly. To distinguish between these two gates, it is common to say "open" and "closed" for the fast gate, and say "active" and "inactive" for the slow gate.

The fast gate is represented by the gating variable m and has $k = 3$ sub-gates. The slow gate is represented with an h and has $k = 1$ sub-gates. This gives a current of the form

$$I_{Na} = -\overline{g}_{Na} m^3 h (V - E_{Na})$$

where $\overline{g}_{Na} = 120$ mS/cm^2, $E_{Na} = 50$ mV, m is the open probability, h is the active probability. As above, the gating variables obey rate equations of the form

$$\frac{dm}{dt} = (1-m)\alpha_m - m\beta_m$$

$$\frac{dh}{dt} = (1-h)\alpha_h - h\beta_h$$

where

$$\alpha_m = \frac{0.1(V+40)}{1 - e^{-0.1(V+40)}},$$

$$\beta_m = 4e^{-0.0556(V+65)},$$

$$\alpha_h = 0.07e^{-0.05(V+65)},$$

and

$$\beta_h = \frac{1}{1 + e^{-0.1(V+35)}}$$

As above, these can also be written in terms of $m_\infty(V)$, $\tau_m(V)$, $h_\infty(V)$ and $\tau_h(V)$. In general, for all the gating variables $a = n, h, m$, we have

$$\tau_a(V)\frac{da}{dt} = a_\infty(V) - a$$

where

$$\tau_a(V) = \frac{1}{\alpha_a(V) + \beta_a(V)}.$$

and

$$a_\infty(V) = \frac{\alpha_a(V)}{\alpha_a(V) + \beta_a(V)}$$

The variables $a_\infty(V)$ give the "steady state" or fixed point of the gating variables $a = n, h, m$ when V is clamped and $\tau_a(V)$ gives the timescale over which this steady

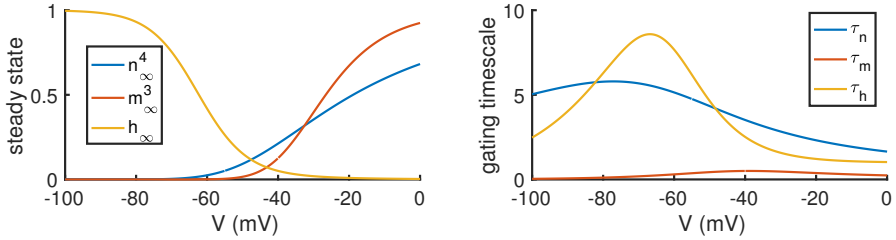

Figure B.1: Steady states and timescales for the three gating variables in the Hodgkin-Huxley model. Plots of $a_\infty^k(V)$ (Left) and $\tau_a(V)$ (Right) for $a = n, h, m$. See `HHgating.m` for code to produce this plot.

state value is approached. The dependence of each $a_\infty^k(V)$ and $\tau_a(V)$ on V is plotted in Figure B.1. We can use these plots to think through the membrane currents we should expect at different values of V.

At rest, $V < -60$mV or so, the Na channels are essentially closed because $m_\infty^3(V) \approx 0$ for $V < -60$mV, so $I_{Na} \approx 0$. The K channels are mostly closed at -60mV because n_∞^4 is just a little bit above zero. But even to the extent that they are open, they only help to keep the membrane potential below -60mV because $E_K = -77$mV, so I_K pulls the membrane potential down. Hence, if V starts below -60mV and we unclamp it, it will stay below -60mV unless an extra inward current is applied.

Now consider what would happen if we unclamp V and apply an inward current strong enough to bring V toward -50mV or so. The Na channels would start to open very quickly (because $m_\infty^3(V)$ is no longer so close to zero and $\tau_m(V)$ is very small). This opening of Na channels pulls V up more because $E_{Na} = 50$mV. This creates a positive feedback loop where V is increasing, causing $m(V)$ to get larger, which causes V to increase more, etc. This positive feedback loop is responsible for the fast "upswing" of the action potential.

As V increases, h slowly starts to decrease (because $h_\infty(V) \approx 0$ for larger V, but $\tau_h(V)$ is not so small), which slowly shuts down the positive feedback loop (because $h \approx 0$ closes the Na channels). At the same time, the K channels are slowly opening (because $n_\infty^4(V) > 0$ for larger V and $\tau_n(V)$ is not so small), which works to pull the membrane potential back down (because $E_K = -77$mV), also slowing the positive feedback loop. These combined effects of $n(t)$ and $h(t)$ end the action potential and pull the membrane potential back down toward rest.

To see how all this plays out in practice, we need to put all of the pieces together into one large model. At the expense of repeating some equations from above, the Hodgkin Huxley model in its entirety is given by

$$\begin{aligned}
C_m \frac{dV}{dt} &= -\overline{g}_L(V - E_L) - \overline{g}_K n^4 (V - E_K) - \overline{g}_{Na} m^3 h (V - E_{Na}) + I_{app} \\
\frac{dn}{dt} &= (1 - n)\alpha_n(V) - n\beta_n(V) \\
\frac{dh}{dt} &= (1 - h)\alpha_h(V) - h\beta_h(V) \\
\frac{dm}{dt} &= (1 - m)\alpha_m(V) - m\beta_m(V)
\end{aligned}$$

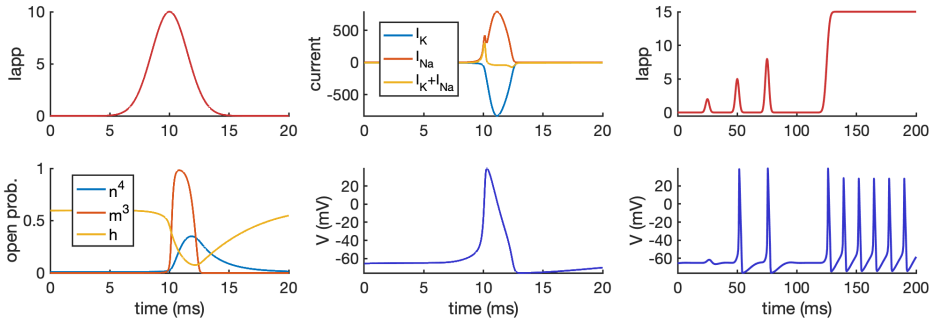

Figure B.2: Simulations of the Hodgkin-Huxley model. The left and center columns show a simulation where a single pulse of input evokes a single action potential. The right panel shows the response to three pulsatile inputs of different amplitudes and a sustained input. See `HHspikes.m` for code to produce this figure.

where

$$\alpha_n(V) = \frac{0.01(V+55)}{1-e^{-0.1(V+55)}},$$

$$\beta_n(V) = 0.125e^{-0.0125(V+65)},$$

$$\alpha_m(V) = \frac{0.1(V+40)}{1-e^{-0.1(V+40)}},$$

$$\beta_m(V) = 4e^{-0.0556(V+65)},$$

$$\alpha_h(V) = 0.07e^{-0.05(V+65)},$$

and

$$\beta_h(V) = \frac{1}{1+e^{-0.1(V+35)}}$$

with V measured in mV. Parameters values are $C_m = 1\mu\text{F/cm}^2$, $g_L = 0.3$ $m\text{S/cm}^2$, $E_L = -54.387$, $\overline{g}_K = 36$ mS/cm^2, $E_K = -77$ mV, $\overline{g}_{Na} = 120$ mS/cm^2, $E_{Na} = 50$ mV. The injected current, $I_{app}(t)$, models a current applied by an experimenter.

In `HHspikes.m`, we use the forward Euler method to solve these equations numerically for two different inputs, $I_{app}(t)$. The results are shown in Figure B.2. The left and middle panels zoom in on a single action potential. The input causes an increase in V, which results in a sharp increase in m^3, causing a sharp increase in I_{Na}, which produces the positive feedback loop that creates the upswing of the action potential. In the meantime, n^4 slowly increasing and h is slowly decreasing, which causes a slow decrease in I_K and I_{Na} that produces the downswing of the action potential. We used a Gaussian-shaped function for $I_{app}(t)$ instead of a square wave pulse because the HH model can respond in unexpected ways to rapid jumps in the applied current. The right panel shows the membrane potential response to multiple pulses of different strengths as well as a sustained, step-like input.

The role of each gating variable in an action potentials can be summarized as:

- m causes V to increase
- h stops V from increasing

- n causes V to decrease

The processes underlying the generation of an action potential can be summarized in more detail as follows:

1. A positive I_{app} causes an increase in V.
2. The increase in V causes an increase in m (since m_∞^3 is an increasing function of V) and the increase in m also causes an increase in V (since $E_{Na} = 55\text{mV}$). This creates a positive feedback loop and rapid increase in V. This all happens before h or n can change much because $\tau_m < \tau_h, \tau_n$.
3. After a millisecond or so, h starts to go to zero (because h_∞ is near zero for large V) causing the Na channels to close, which shuts down the positive feedback loop.
4. In the meantime, n has increased (i.e., K channels have opened) so, once the Na channels close, the membrane potential is pulled back down to negative values (since $E_K = -77\text{mV}$) as K ions rush *out* of the cell. This ends the action potential.
5. After h recovers, the cycle can repeat as long as there is still a positive I_{app} to push V large enough to start the positive feedback loop.

The mathematical biologist, Jim Keener, developed an interpretive dance for this process called the "Hodgkin-Huxley macarena." Search for videos of the dance online.

Some important concepts related to action potentials are listed below along with some instructions for investigating each one numerically.

- **Threshold:** For an action potential to occur, V needs to get large enough to start the positive feedback loop. The cutoff V to initiate an action potential is called the **threshold potential**, often denoted V_{th} or θ. When action potentials are driven by an applied current, $I_{app}(t)$, then the **threshold current** is the value of $I_{app}(t)$ needed to evoke an action potential. But note that the threshold current and potentials can depend on the shape of the current used to evoke the action potential. For example, a sharply rising current might evoke an action potential more easily than a slowly rising current. Use a step-like or pulse-like applied current to find the threshold potential and current for the HH model by changing the height of the step or pulse. Notice how sharp the threshold is.
- **Refractory Period:** Directly after an action potential, h is near zero so the Na channels are closed and therefore the neuron can't spike again until the h gates open back up. This lasts for about 2 ms after a spike. Try eliciting two action potentials close together with two strong pulsatile inputs. See how close together you can get them.
- **All-or-None Response:** An action potential either occurs or doesn't occur. The magnitude and shape of the action potential is mostly independent of the input that evoked it. Try evoking multiple action potentials with pulsatile inputs of varying strengths. See how, for a large range of pulse strengths, the action potential waveform is essentially the same.

PROBLEMS WITH THE HODGKIN-HUXLEY MODEL FOR MODELING LARGE NETWORKS. One problem with the HH model is that its parameters were fit to the dynamics of the squid giant axon. While the basic mechanisms of action potential generation is the same in other types of neurons, the specific dynamics can differ greatly across different neurons and neuron types. Indeed, you may have noticed that the resting potential for the HH model with $I_{app}(t) = 0$ is very different from the ≈ -70mV that is more typical of mammalian cortical neurons. HH style models that more accurately model mammalian cortical neurons can be developed by changing parameter values and adding other ion channels.

HH style models have many complicated properties beyond the simple property of producing an action potential in response to depolarizing input. These more complicated properties can make it difficult to understand the mechanisms underlying different dynamics observed in simulations. Simplified neuron models have much simpler dynamics so it is easier to understand what is going on in simulations. This simplicity comes at the expense of possibly omitting some properties of real neurons, though.

The time step, dt, we used for simulating the HH model is very small. The small time step is needed to capture the fast dynamics underlying action potential generation. You can verify this by increasing the time step, dt, in the HH simulations to see how it changes the dynamics. The use of a smaller time step causes simulations of the HH model to take longer. A larger time step could be used if we used a higher-order ODE solver instead of using forward Euler method, but this can only help so much because the fast dynamics during an action potential still require a relatively small time step. This computational inefficiency might not be an issue when modeling a single neuron for a short amount of time, but can be an issue when simulating large networks of many neurons for longer periods of time.

ACKNOWLEDGMENTS

I would like to thank Adam Kohn, Matthew Smith, Micheal Okun, and Ilan Lampl for providing the data used in Chapter 2 and Exercise 4.3.2. I would also like to thank David Toth, Vicky Zhu, and all of the other students whose feedback and comments helped shape the contents of this book.

BIBLIOGRAPHY

[1] P. Dayan and L. F. Abbott. *Theoretical Neuroscience*. Cambridge, MA: MIT Press, 2001.

[2] W. Gerstner, W. M. Kistler, R. Naud, and L. Paninski. *Neuronal dynamics: From single neurons to networks and models of cognition*. Cambridge University Press, 2014.

[3] G. Lindsay. *Models of the Mind: How Physics, Engineering and Mathematics Have Shaped Our Understanding of the Brain*. Bloomsbury Publishing, 2021.

[4] N. Fourcaud-Trocme, D. Hansel, C. van Vreeswijk, and N. Brunel. "How spike generation mechanisms determine the neuronal response to fluctuating inputs." In: *J Neurosci* 23 (2003), pp. 11628–11640.

[5] R. Jolivet, T. J. Lewis, and W. Gerstner. "Generalized integrate-and-fire models of neuronal activity approximate spike trains of a detailed model to a high degree of accuracy." In: *J Neurophysiol* 92.2 (2004), pp. 959–976.

[6] L. Badel, S. Lefort, T. K. Berger, C. C. Petersen, W. Gerstner, and M. J. Richardson. "Extracting non-linear integrate-and-fire models from experimental data using dynamic I–V curves." In: *Biological cybernetics* 99.4 (2008), pp. 361–370.

[7] M. Okun, A. Naim, and I. Lampl. "The subthreshold relation between cortical local field potential and neuronal firing unveiled by intracellular recordings in awake rats." In: *Journal of neuroscience* 30.12 (2010), pp. 4440–4448.

[8] M. A. Smith and A. Kohn. "Spatial and temporal scales of neuronal correlation in primary visual cortex." In: *Journal of Neuroscience* 28.48 (2008), pp. 12591–12603.

[9] W. R. Softky and C. Koch. "The highly irregular firing of cortical cells is inconsistent with temporal integration of random EPSPs." In: *Journal of neuroscience* 13.1 (1993), pp. 334–350.

[10] M. N. Shadlen and W. T. Newsome. "Noise, neural codes and cortical organization." In: *Current opinion in neurobiology* 4.4 (1994), pp. 569–579.

[11] M. N. Shadlen and W. T. Newsome. "The variable discharge of cortical neurons: implications for connectivity, computation, and information coding." In: *Journal of Neuroscience* 18.10 (1998), pp. 3870–3896.

[12] M. M. Churchland, M. Y. Byron, J. P. Cunningham, L. P. Sugrue, M. R. Cohen, G. S. Corrado, W. T. Newsome, A. M. Clark, P. Hosseini, B. B. Scott, et al. "Stimulus onset quenches neural variability: a widespread cortical phenomenon." In: *Nature neuroscience* 13.3 (2010), pp. 369–378.

[13] G. L. Gerstein and B. Mandelbrot. "Random walk models for the spike activity of a single neuron." In: *Biophysical journal* 4.1 (1964), pp. 41–68.

[14] C. Curto, A. Degeratu, and V. Itskov. "Encoding binary neural codes in networks of threshold-linear neurons." In: *Neural computation* 25.11 (2013), pp. 2858–2903.

[15] B. W. Knight. "Dynamics of encoding in a population of neurons." In: *The Journal of general physiology* 59.6 (1972), pp. 734–766.

[16] L. M. Ricciardi and L. Sacerdote. "The Ornstein-Uhlenbeck process as a model for neuronal activity." In: *Biological cybernetics* 35.1 (1979), pp. 1–9.

[17] D. J. Amit and M. Tsodyks. "Quantitative study of attractor neural network retrieving at low spike rates. I. Substrate-spikes, rates and neuronal gain." In: *Network: Computation in neural systems* 2.3 (1991), p. 259.

[18] B. Lindner, J. Garcıa-Ojalvo, A. Neiman, and L. Schimansky-Geier. "Effects of noise in excitable systems." In: *Physics reports* 392.6 (2004), pp. 321–424.

[19] M. J. Richardson. "Firing-rate response of linear and nonlinear integrate-and-fire neurons to modulated current-based and conductance-based synaptic drive." In: *Physical Review E* 76.2 (2007), p. 021919.

[20] A. M. Bastos, W. M. Usrey, R. A. Adams, G. R. Mangun, P. Fries, and K. J. Friston. "Canonical microcircuits for predictive coding." In: *Neuron* 76.4 (2012), pp. 695–711.

[21] M. Stimberg, R. Brette, and D. F. Goodman. "Brian 2, an intuitive and efficient neural simulator." In: *eLife* 8 (Aug. 2019). Ed. by F. K. Skinner, e47314. ISSN: 2050-084X. DOI: 10.7554/eLife.47314.

[22] H. R. Wilson and J. D. Cowan. "Excitatory and inhibitory interactions in localized populations of model neurons." In: *Biophysical journal* 12.1 (1972), pp. 1–24.

[23] H. R. Wilson and J. D. Cowan. "A mathematical theory of the functional dynamics of cortical and thalamic nervous tissue." In: *Kybernetik* 13.2 (1973), pp. 55–80.

[24] M. V. Tsodyks, W. E. Skaggs, T. J. Sejnowski, and B. L. McNaughton. "Paradoxical effects of external modulation of inhibitory interneurons." In: *Journal of neuroscience* 17.11 (1997), pp. 4382–4388.

[25] H. Ozeki, I. M. Finn, E. S. Schaffer, K. D. Miller, and D. Ferster. "Inhibitory stabilization of the cortical network underlies visual surround suppression." In: *Neuron* 62.4 (2009), pp. 578–592.

[26] G. Kopell Nancyand Ermentrout, M. Whittington, and R. Traub. "Gamma rhythms and beta rhythms have different synchronization properties." In: *Proceedings of the National Academy of Sciences* 97.4 (2000), pp. 1867–1872.

[27] M. A. Whittington, R. D. Traub, N. Kopell, B. Ermentrout, and E. H. Buhl. "Inhibition-based rhythms: experimental and mathematical observations on network dynamics." In: *International journal of psychophysiology* 38.3 (2000), pp. 315–336.

[28] N. Brunel and X.-J. Wang. "What determines the frequency of fast network oscillations with irregular neural discharges? I. Synaptic dynamics and excitation-inhibition balance." In: *Journal of neurophysiology* 90.1 (2003), pp. 415–430.

[29] J. A. Cardin, M. Carlén, K. Meletis, U. Knoblich, F. Zhang, K. Deisseroth, L.-H. Tsai, and C. I. Moore. "Driving fast-spiking cells induces gamma rhythm and controls sensory responses." In: *Nature* 459.7247 (2009), pp. 663–667.

[30] R. Rosenbaum, M. Smith, A Kohn, J. Rubin, and B Doiron. "The spatial structure of correlated neuronal variability." In: *Nature Neuroscience* 20.1 (2017), p. 107.

[31] C. Ebsch and R. Rosenbaum. "Imbalanced amplification: A mechanism of amplification and suppression from local imbalance of excitation and inhibition in cortical circuits." In: *PLoS Computational Biology* 14.3 (2018), e1006048.

[32] C. Baker, C. Ebsch, I. Lampl, and R. Rosenbaum. "Correlated states in balanced neuronal networks." In: *Physical Review E* 99.5 (2019), p. 052414.

[33] C. Baker, V. Zhu, and R. Rosenbaum. "Nonlinear stimulus representations in neural circuits with approximate excitatory-inhibitory balance." In: *PLoS Computational Biology* 16.9 (2020), e1008192.

[34] D. O. Hebb. *The organization of behavior: A neuropsychological theory*. Wiley and Sons, New York, NY, 1949.

[35] P. E. Castillo, C. Q. Chiu, and R. C. Carroll. "Long-term plasticity at inhibitory synapses." In: *Current opinion in neurobiology* 21.2 (2011), pp. 328–338.

[36] T. P. Vogels, H Sprekeler, F Zenke, C Clopath, and W Gerstner. "Inhibitory plasticity balances excitation and inhibition in sensory pathways and memory networks." In: *Science* 334.6062 (2011), pp. 1569–73. ISSN: 1095-9203.

[37] Y. Luz and M. Shamir. "Balancing feed-forward excitation and inhibition via Hebbian inhibitory synaptic plasticity." In: *PLoS computational biology* 8.1 (2012), e1002334.

[38] T. P. Vogels et al. "Inhibitory synaptic plasticity: spike timing-dependence and putative network function." In: *Frontiers in Neural Circuits* 7.119 (2013). ISSN: 1662-5110.

[39] G. Hennequin, E. J. Agnes, and T. P. Vogels. "Inhibitory Plasticity: Balance, Control, and Codependence." In: *Annu. Rev. Neurosci.* 40.1 (2017), pp. 557–579. ISSN: 0147-006X.

[40] A. Schulz, C. Miehl, M. J. Berry II, and J. Gjorgjieva. "The generation of cortical novelty responses through inhibitory plasticity." In: *Elife* 10 (2021), e65309.

[41] M. Capogna, P. E. Castillo, and A. Maffei. "The ins and outs of inhibitory synaptic plasticity: Neuron types, molecular mechanisms and functional roles." In: *European Journal of Neuroscience* 54.8 (2021), pp. 6882–6901.

[42] D. Sussillo and L. F. Abbott. "Generating coherent patterns of activity from chaotic neural networks." In: *Neuron* 63.4 (2009), pp. 544–557.

[43] J. Ginibre. "Statistical ensembles of complex, quaternion, and real matrices." In: *Journal of Mathematical Physics* 6.3 (1965), pp. 440–449.

[44] V. L. Girko. "Circular law." In: *Theory of Probability & Its Applications* 29.4 (1985), pp. 694–706.

[45] K. Rajan and L. F. Abbott. "Eigenvalue spectra of random matrices for neural networks." In: *Physical review letters* 97.18 (2006), p. 188104.

[46] H. Sompolinsky, A. Crisanti, and H.-J. Sommers. "Chaos in random neural networks." In: *Physical review letters* 61.3 (1988), p. 259.

[47] H. Jaeger. "The "echo state" approach to analysing and training recurrent neural networks-with an erratum note." In: *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report* 148.34 (2001), p. 13.

[48] W. Maass, T. Natschläger, and H. Markram. "Real-time computing without stable states: A new framework for neural computation based on perturbations." In: *Neural computation* 14.11 (2002), pp. 2531–2560.

[49] H. Jaeger. "Harnessing Nonlinearity: Predicting Chaotic." In: *Science* 1091277.78 (2004), p. 304.

[50] M. Lukoševičius and H. Jaeger. "Reservoir computing approaches to recurrent neural network training." In: *Computer Science Review* 3.3 (2009), pp. 127–149.

[51] G. M. Hoerzer, R. Legenstein, and W. Maass. "Emergence of complex computational structures from chaotic neural networks through reward-modulated Hebbian learning." In: *Cerebral cortex* 24.3 (2014), pp. 677–690.

[52] R. Pyle and R. Rosenbaum. "A Reservoir Computing Model of Reward-Modulated Motor Learning and Automaticity." In: *Neural Computation* 31.7 (2019), pp. 1430–1461.

[53] J. M. Murray and G. S. Escola. "Remembrance of things practiced with fast and slow learning in cortical and subcortical pathways." In: *Nature Communications* 11.1 (2020), pp. 1–12.

[54] V. Mante, D. Sussillo, K. V. Shenoy, and W. T. Newsome. "Context-dependent computation by recurrent dynamics in prefrontal cortex." In: *Nature* 503.7474 (2013), pp. 78–84.

[55] D. Sussillo. "Neural circuits as computational dynamical systems." In: *Current opinion in neurobiology* 25 (2014), pp. 156–163.

[56] D. Sussillo, M. M. Churchland, M. T. Kaufman, and K. V. Shenoy. "A neural network that finds a naturalistic solution for the production of muscle activity." In: *Nature neuroscience* 18.7 (2015), pp. 1025–1033.

[57] F. Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[58] B. Widrow and M. E. Hoff. *Adaptive switching circuits*. Tech. rep. Stanford Univ Ca Stanford Electronics Labs, 1960.

[59] J. C. Whittington and R. Bogacz. "Theories of error back-propagation in the brain." In: *Trends in cognitive sciences* 23.3 (2019), pp. 235–250.

[60] T. P. Lillicrap and A. Santoro. "Backpropagation through time and the brain." In: *Current opinion in neurobiology* 55 (2019), pp. 82–89.

[61] T. P. Lillicrap, A. Santoro, L. Marris, C. J. Akerman, and G. Hinton. "Backpropagation and the brain." In: *Nature Reviews Neuroscience* 21.6 (2020), pp. 335–346.

[62] S.-M. Khaligh-Razavi and N. Kriegeskorte. "Deep supervised, but not unsupervised, models may explain IT cortical representation." In: *PLoS computational biology* 10.11 (2014), e1003915.

[63] A. J. Kell, D. L. Yamins, E. N. Shook, S. V. Norman-Haignere, and J. H. McDermott. "A task-optimized neural network replicates human auditory behavior, predicts brain responses, and reveals a cortical processing hierarchy." In: *Neuron* 98.3 (2018), pp. 630–644.

[64] M. Schrimpf, J. Kubilius, M. J. Lee, N. A. R. Murty, R. Ajemian, and J. J. DiCarlo. "Integrative Benchmarking to Advance Neurally Mechanistic Models of Human Intelligence." In: *Neuron* (2020).

[65] B. A. Richards, T. P. Lillicrap, P. Beaudoin, Y. Bengio, R. Bogacz, A. Christensen, C. Clopath, R. P. Costa, A. de Berker, S. Ganguli, et al. "A deep learning framework for neuroscience." In: *Nature neuroscience* 22.11 (2019), pp. 1761–1770.

[66] J. Cornford, D. Kalajdzievski, M. Leite, A. Lamarquette, D. M. Kullmann, and B. A. Richards. "Learning to live with Dale's principle: ANNs with separate excitatory and inhibitory units." In: *International Conference on Learning Representations*. 2020.

[67] J. Guerguiev, T. P. Lillicrap, and B. A. Richards. "Towards deep learning with segregated dendrites." In: *ELife* 6 (2017), e22901.

[68] D. Huh and T. J. Sejnowski. "Gradient descent for spiking neural networks." In: *Advances in neural information processing systems* 31 (2018).

[69] E. O. Neftci, H. Mostafa, and F. Zenke. "Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks." In: *IEEE Signal Processing Magazine* 36.6 (2019), pp. 51–63.

[70] G. Bellec, F. Scherr, A. Subramoney, E. Hajek, D. Salaj, R. Legenstein, and W. Maass. "A solution to the learning dilemma for recurrent networks of spiking neurons." In: *Nature communications* 11.1 (2020), pp. 1–15.

[71] Y. Li, Y. Guo, S. Zhang, S. Deng, Y. Hai, and S. Gu. "Differentiable Spike: Rethinking Gradient-Descent for Training Spiking Neural Networks." In: *Advances in Neural Information Processing Systems* 34 (2021).

[72] A. Payeur, J. Guerguiev, F. Zenke, B. A. Richards, and R. Naud. "Burst-dependent synaptic plasticity can coordinate learning in hierarchical circuits." In: *Nature neuroscience* 24.7 (2021), pp. 1010–1019.

[73] A. Robinson. "Did Einstein really say that?" In: *Nature* 557.7703 (2018), pp. 30–31.

[74] S. H. Strogatz. *Nonlinear dynamics and chaos: with applications to physics, biology, chemistry, and engineering*. CRC press, 2018.