SYDE 556/750
Simulating Neurobiological Systems
Lecture 3: Representation

Andreas Stöckel

Based on lecture notes by
Chris Eliasmith and Terrence C. Stewart

January 14 & 16, 2020

UNIVERSITY OF
WATERLOO

Accompanying Readings: Chapter 2 of Neural Engineering

Contents

1   Introduction

Note: In the last lecture, we had a look at a the Leaky Integrate-and-Fire neuron model as an approximation of the behaviour of biological neurons. In this lecture, we discuss a theory of neural representation. Together, these two ideas will give us a theory of what "the neural code" may be.

In order to survive in natural environments, animals must build representations of their surroundings. This includes the physical properties of objects in their vicinity, such as *velocity*, *location*, *colour*, etc., but also non-physical properties such as whether an object is *edible*, or *dangerous*. Of course such representations are not only limited to external objects, but may carry information about the own body (e.g., "proprioception", configuration of limbs; location in space), up to abstract thought and emotions.

For now, and mostly because it is the easiest to reason about, we shall restrict ourselves to relatively low-level representations of sensory information. In this case, the fact that a neuron is representing an external quantity is directly evident from correlation between the quantity and the neuron's activity. We saw examples of such representations in the first lecture. For example, in the classical Hubel and Wiesel experiment (🖥 Video, [1]), neural activity is correlated with the location and orientation of a bar of light (an "edge") within the visual field. Similarly, in the case of place cells (🖥 Video), neural activity is correlated with the location of the animal in space.

A central claim of the Neural Engineering Framework (NEF) is that values **x** – e.g., the physical quantities mentioned above – are represented in the neural activity **α** of *groups* of neurons. Furthermore, we claim that there is a non-linear encoding process that translates a value into a neural activity pattern, and a linear decoding process that estimates the represented value from neural activity patterns.

> **NEF Principle 1 – Representation**
> *Groups* ("populations", or "ensembles") of neurons *represent* values via nonlinear encoding and linear decoding.

We will first discuss the encoding and decoding of values in a more abstract sense, and then propose a particular mechanism by which values are encoded in the NEF. Since *encoding* only takes us half of the way, we then discuss *decoding*, in particular in the presence of noise in the neural activities.

Notaton: Scalars, functions are italic letters (d, n, N, f,...), vectors are bold lowercase letters (**α**, **e**, **x**), matrices are bold uppercase letters (**A**), sets are double-struck uppercase letters ($\mathbb{R}$, $\mathbb{N}$). $\mathbb{A} \to \mathbb{B}$ denotes a function mapping from the domain $\mathbb{A}$ onto a codomain $\mathbb{B}$.

$d \in \mathbb{N}$	Dimensionality of the value represented by a neuron population
$n \in \mathbb{N}$	Number of neurons in a neuron population
$N \in \mathbb{N}$	Number of samples

$\langle \mathbf{x}, \mathbf{y} \rangle : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}$	Dot product between two k-dimensional vectors $\mathbf{x}$ and $\mathbf{y}$
$G[J] : \mathbb{R} \to \mathbb{R}^+$	Neural response for an input current J (*response curve*)
$a_i(\mathbf{x}) : \mathbb{R}^d \to \mathbb{R}^+$	Neural response of neuron i in a neural population (*tuning curve*)
$\mathbf{a}(\mathbf{x}) : \mathbb{R}^d \to (\mathbb{R}^+)^n$	Neural response of all neurons in a neural population
$\mathbf{e}_i \in \mathbb{R}^d$	Encoding vector, or "preferred direction", of neuron i in a population
$\alpha_i \in \mathbb{R}$	The i-th neuron's gain factor
$J_i^{\mathrm{bias}} \in \mathbb{R}$	The i-th neuron's bias current
$\mathbf{x}_j \in \mathbb{R}^d$	j-th (out of N) input sample
$\hat{\mathbf{x}}_j \in \mathbb{R}^d$	Decoded j-th input sample
$\mathbf{X} \in \mathbb{R}^{d \times N}$	Matrix of input samples
$\mathbf{A} \in \mathbb{R}^{n \times N}$	Matrix of sampled population activities
$\mathbf{d}_i \in \mathbb{R}^n$	Decoding vector for the i-th neuron in the population
$\mathbf{D} \in \mathbb{R}^{d \times n}$	Decoding matrix

2 Codes: Representing Information

Human societies have relied on exchanging coded information for millennia. Apart from written and spoken language themselves, examples of such codes include flag semaphores, or Morse code [2].

These codes have in common that there is some kind of *encoding* process that turns an original message $\mathbf{x}$ into a coded message $\mathbf{a}$, and a *decoding* process that reconstructs $\mathbf{x}$ from $\mathbf{a}$. If $\mathbf{x}$ can be perfectly reconstructed, this is called *lossless coding*, otherwise *lossy coding*.

2.1 Lossless Codes

In the case of a lossless code, there exists an encoding function f, as well as its inverse f^{-1}. It holds

$$\mathbf{a} = f(\mathbf{x}), \qquad\qquad \text{Encoding}$$
$$\mathbf{x} = f^{-1}(\mathbf{a}) = g(f(\mathbf{x})). \qquad\qquad \text{Decoding}$$

Morse Code Morse code is an example of such a lossless coding scheme. A function f takes a character sequence and turns it into valid Morse code, consisting of dots ($\cdot$) and dashes ($-$):

$$f : \mathcal{S}(\{A, \ldots, Z, \_\}) \to \mathbb{M},$$
$$f^{-1} : \mathbb{M} \to \mathcal{S}(\{A, \ldots, Z, \_\}),$$

where $\mathbb{M} \subset \mathcal{S}(\{\cdot, -\})$ is the set of valid Morse messages.

Here, $\mathcal{S}$ denotes the set of all sequences consisting of the given elements (i.e., the union over all Cartesian powers).

Binary numbers As a more concrete example, let's consider an n-bit binary coding of natural numbers. Here, our coded message $\mathbf{a} = (a_0, a_1, a_2, a_3, \ldots, a_{n-1})$ where $a_i \in \{0, 1\}$, while our original value x is a natural number between zero and $2^n - 1$.

We can write down the encoding function f

$$a_i = \left(f(x)\right)_i = \begin{cases} 1 & \text{if } x - 2^i \left\lfloor \frac{x}{2^i} \right\rfloor > 2^{i-1}, \\ 0 & \text{otherwise}. \end{cases}$$

This function is nonlinear – we cannot write it as a matrix-vector product. The decoding function f^{-1} is given as

$$x = f^{-1}(\mathbf{a}) = \sum_{i=0}^{n-1} 2^i a_i.$$

This function is *linear* in $\mathbf{a}$ because we can write it as a matrix-vector product

$$x = f^{-1}(\mathbf{a}) = \mathbf{F}\mathbf{a} = \begin{pmatrix} 1 & 2 & \ldots & 2^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}.$$

Since a single value x is represented in multiple "units" (here: digits of a binary number), we call this kind of coding scheme a *distributed* representation.

> **Note:** A precise definition of *linearity* is provided in the notes for Lecture 2.
>
> We could imagine using such a binary coding scheme in the context of a neural network. In a population of n neurons, each neuron i would represent one binary digit. However, there are several problems with this approach.
>
> - Individual units represent vastly different magnitudes. Since biological neural networks are noisy and we are using a linear decoding scheme, noise in the unit representing the digit $i = 8$ is amplified by a factor of 256 compared to noise in the unit representing the digit $i = 0$.
>
> - We assume that our units can only have two possible values, zero and one, whereas neural units can have graded responses (cf. the response curve $G[J]$).
>
> - The encoding function is quite complicated. There is no neurophysiological evidence that brains use a binary coding scheme.

2.2 Lossy Codes

In contrast to lossless codes, lossy codes have no perfect inverse f^{-1}. Instead, we have a decoding function g that *approximates* the original value. Mathematically, we have

$$\mathbf{a} = f(\mathbf{x}), \qquad\qquad \text{Lossy Encoding}$$
$$\mathbf{x} \approx g(\mathbf{a}) = g(f(\mathbf{x})). \qquad\qquad \text{Lossy Decoding}$$

Examples of lossy codes   In mathematics and engineering, examples of such lossy codes include Principal Component Analysis (PCA; nonlinearly reduces a dataset onto its first *n* principal basis vectors), and coding schemes for audio and images, such as MP3 and JPEG (these coding schemes are based on the Discrete Cosine Transformation).

> *Aside:* Surprisingly, the Discrete Cosine Transformation (DCT) turns out to be a close to optimal *linear* transformation for the compression of natural signals. When computing the PCA of a large corpus of natural images or audio (the PCA optimally ensures that the most variance in the data is explained by the first dimensions), the resulting linear transformation is very similar to the DCT. However, the DCT of a signal $\mathbf{x}$ can be computed more efficiently (in $\mathcal{O}(d\log(d))$) compared to a general linear transformation (in $\mathcal{O}(d^2)$).

3   Neural Representation

As discussed in the introduction, there are some neuron populations in the brain that directly represent information about stimuli in the environment. In order to better understand these representations, we first have a look at so called *tuning curves*, mappings between external stimuli and the neural activity.

After looking at some real-world data, we combine our observations into a general encoding equation that captures as many properties of tuning curves as possible.

> **Note:** The coding scheme we are using for neural representation is both *distributed* and (in practice) *lossy*. In contrast to binary coding, our distributed coding scheme is more resilient to noise in the neural activities $\mathbf{\alpha}$. Lossyness mostly stems from neurons having a limited dynamic range (saturation effects) and – although this is not an inherent problem of the coding scheme itself – noise in the neural system.

3.1   Neural Tuning Curves

In the last lecture, we have discussed neural *response curves*. Response curves are a mapping between the current *J* injected into a neuron and the corresponding neural activities. *Tuning curves* are a similar concept. However, instead of plotting the current *J* over the neural response, we plot some varying (external) stimulus over the neural response. Figures 1 and 2 depict neural tuning curves as found in the neuroscience literature.

> **Note:** Tuning curves and *internal* stimuli. Neuroscientists usually define tuning curves as the relationship between an *external* stimulus and the neural response. In the NEF we extend this concept towards internal stimuli, i.e., neurons representing purely internal states, such as short-term memory. The lack of such tuning curves in the literature may be purely attributed to it being very hard to experimentally capture these relationships.

Overall, we can qualitatively distinguish between three types of tuning curves (cf. fig. 2d)
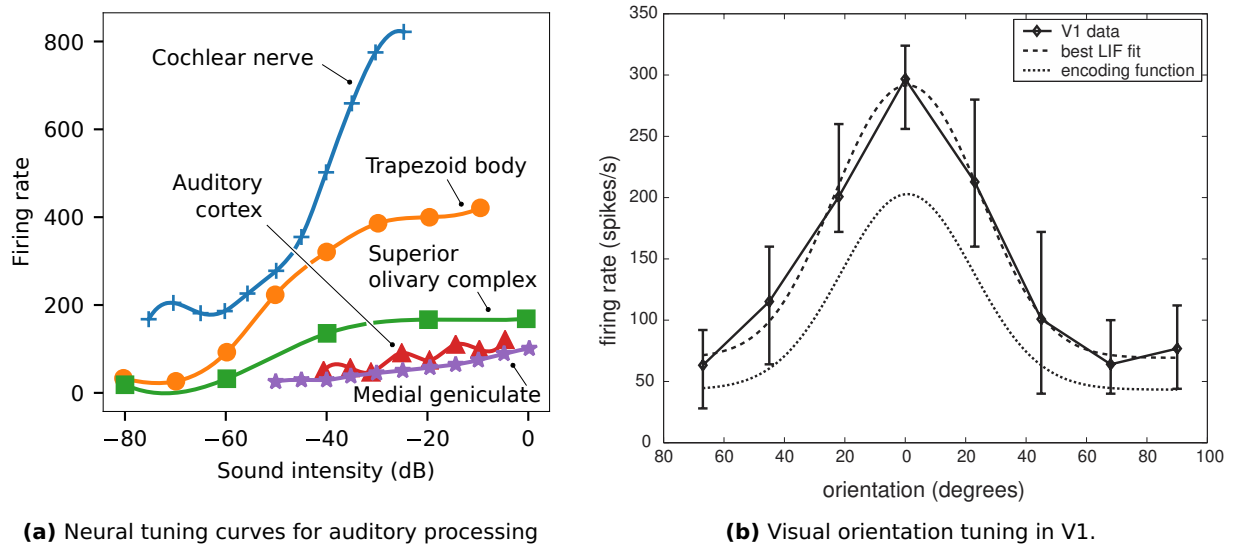
(a) Neural tuning curves for auditory processing

(b) Visual orientation tuning in V1.

Figure 1: (a) Neural firing rates in different brain regions involved in auditory processing as a function of sound intensity in cats. Figure adapted from [3] (Chapter 8), data from [4]. ▨ *Code used to plot the data*. **(b)** Example of visual orientation tuning of a cell in primary visual cortex of a macaque monkey. Figure copied from [5], fig. 3.1.
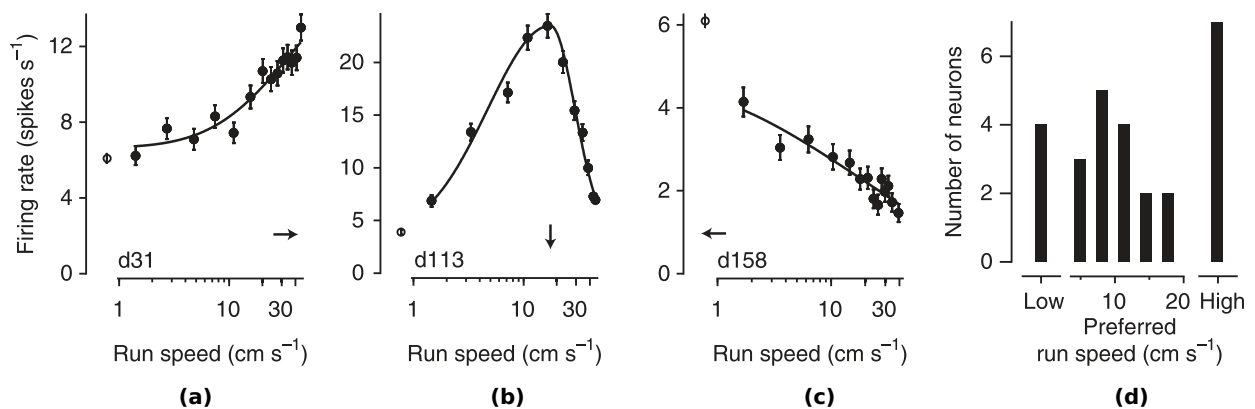

(a)   **(b)**   **(c)**   **(d)**

Figure 2: Tuning curves of individual neurons in visual cortex (layer V1) for mice running in the dark. **(a-c)** Neural firing rates over the running speed of the mouse. **(d)** Histogram over tuning curve types. Figures copied from [6].

i. Responses that *increase* with the intensity of the stimulus (stimulus and neural response are positively correlated; cf. figs. 1a and 2a).

ii. Responses that *decrease* with the intensity of the stimulus (stimulus and neural response are negatively correlated; cf. fig. 2c).

iii. Responses that have a *preferred stimulus*; their response is maximal if the stimulus has a certain value and decreases if the value deviates from that value (cf. figs. 1b and 2b).

Tuning curves of type *i.* can be easily explained by rescaling and offsetting the neural input current J to match the data. In other words, we have a simple *current translation function* $J_i(x)$ that converts the stimulus x into a current J that is being injected into the i-th neuron of our population:

$$J_i(x) = \alpha_i x + J_i^{\text{bias}}, \tag{1}$$

where α_i is the so called *gain factor* and J_i^{bias} is the *bias*. These parameters are different for each neuron in the population and model the diverse neural responses of individual neurons within the same brain area. Tuning curves of type *ii.* can be generated by choosing a negative gain α_i.

> **Example:** You can try to find parameters that qualitatively match the tuning curves in figs. 1 and 2 by playing around with the gain and offset parameters in ▨ *this Jupyter Notebook*.

This leaves us with tuning curves of type *iii.*, i.e. tuning curves having a preferred value. We generate these tuning curves by extending the current translation function eq. (1) to accepting vectorial quantities as input.

3.2 The Encoding Equation

In order to generate tuning curves that qualitatively match all types of tuning curves we have seen above, we let our value **x** be a d-dimensional vector. The dimensionality of the represented value d is specific to each neuron population. We can then choose the dot product as a measure of similarity (see note below):

$$J_i(\mathbf{x}) = \alpha_i \langle \mathbf{x}, \mathbf{e}_i \rangle + J_i^{\text{bias}},$$

where α_i, J_i^{bias} are the gain and bias terms for the i-th neuron in the population, $\mathbf{e}_i \in \mathbb{R}^d$ is the neuron's encoding – or "preferred direction vector", and $\langle \cdot, \cdot \rangle$ denotes the dot product between two vectors. Note that $\mathbf{e}_i$ is always normalised to unit length, that is $\|\mathbf{e}_i\| = 1$.

> **Note:** The *dot product* (also: inner product, scalar product) $\langle \mathbf{x}, \mathbf{y} \rangle$ of two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ is defined as
> $$\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=0}^{d} x_i y_i.$$

We implicitly treat all vectors as column vectors; a vector $\mathbf{x} \in \mathbb{R}^d$ is implicitly a $d \times 1$ matrix (i.e., a matrix of d rows and a single column). Hence, by the definition of matrix multiplication, the dot product is also given as

$$\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^\mathsf{T} \mathbf{y} \quad \text{(matrix dimensions: } (1 \times d) \times (d \times 1) \to (1 \times 1)).$$

In particular, note that the dot product can be interpreted as a vector similarity measure. Namely, according to the Euclidean dot product formula

$$\langle \mathbf{x}, \mathbf{y} \rangle = \cos(\angle(\mathbf{x}, \mathbf{y}))\|\mathbf{x}\|\|\mathbf{y}\|,$$

where $\|\cdot\|$ denotes the standard L_2-norm, and $\angle(\cdot, \cdot)$ is the angle inscribed between two vectors. Hence, the dot product is maximal if the angle between the two vectors is zero (they are pointing in the same direction, the cosine is $+1$) and zero if the two vectors are orthogonal. It is minimal (the cosine is -1) if the two vectors are pointing in exactly opposite directions.

Combining the current $J_i(\mathbf{x})$ with the neuron response curve $G[J]$ (where, again, G is specific to each neuron population) yields the "encoding equation", which describes the nonlinear representational encoding process performed by each neuron within a population

(Encoding Equation)

$$a_i(\mathbf{x}) = G[J_i(\mathbf{x})] = G[\alpha_i \langle \mathbf{x}, \mathbf{e}_i \rangle + J_i^{\mathrm{bias}}]. \tag{2}$$

Example: 1D Encoder   In the case of a one-dimensional encoder, the encoding "vector" can either be $+1$ or -1. This corresponds to o neurons that are either more active in the presence of a stimulus (positive encoder), or neurons that are more active in the absence of a stimulus (negative encoder).

Example: 2D Encoder   Figure 3b shows a tuning curve in a 2D representational space. Note that moving along the unit-circle in 2D-space generates the "preferred direction" tuning curve type observed in nature.

4   Decoding Represented Values

As we have seen in the binary coding example above, some codes possess a "complex", non-linear encoding scheme, but can be decoded using a simple linear decoder. Neuroscientists have similarly found that they can use a simple linear decoding scheme to extract represented values from neural populations – we will later discuss some more examples.

In general, having a linear decoder means that we can estimate the value $\hat{\mathbf{x}}$ represented by a
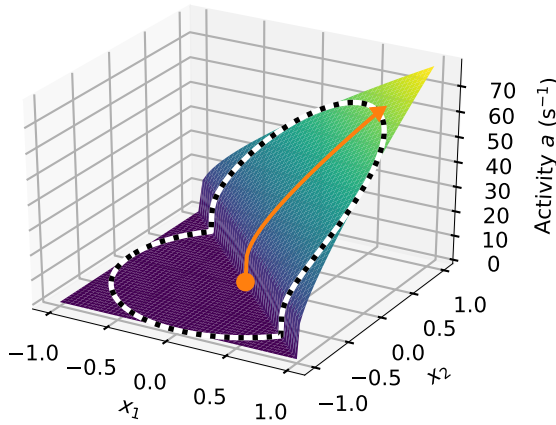
(a) 2D neuron tuning curve
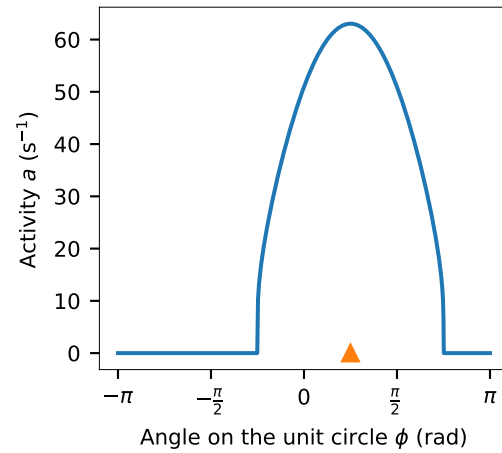
(b) 1D projection along the unit circle

Figure 3: 2D neuron tuning curve for the encoding vector $\mathbf{e} = (1/\sqrt{2}, 1/\sqrt{2})$. **(a)** 3D surface plot of the 2D neuron tuning curve. Axes correspond to the magnitude of the first two components of the represented value $\mathbf{x} = (x_1, x_2)$. Orange arrow corresponds to the encoding vector. Dotted line corresponds to the unit circle. **(b)** Neural activity along the unit circle is qualitatively similar to the bell-shaped tuning curves observed in biology. ⌨ *Code*

neural population by multiplying its current activity $\mathbf{a}$ with a decoding matrix $\mathbf{D}$:

$$(\mathbf{a})_i = G\big[\alpha_i \langle \mathbf{x}, \mathbf{e}_i \rangle + J_i^{\text{bias}}\big], \qquad \text{Encoding}$$

$$\hat{\mathbf{x}} = \mathbf{D}\mathbf{a}. \qquad \text{Decoding}$$

In the special case of a population representing a scalar value (i.e., the represented dimensionality $d = 1$) we have

$$(\mathbf{a})_i = G\big[\alpha_i \langle x, e_i \rangle + J_i^{\text{bias}}\big], \qquad \text{Encoding}$$

$$\hat{x} = \langle \mathbf{d}, \mathbf{a} \rangle = \sum_{i=1}^{n} d_i a_i. \qquad \text{Decoding}$$

Now that we now how to decode the represented value $\mathbf{x}$ *in principle*, we of course have to somehow compute these decoders.

4.1   Computing Identity Decoders

In order to compute the decoders, we first have to think about what the decoders should optimally do. The roles of the decoders is to estimate the represented value. In the optimal case we would like the decoded value $\hat{\mathbf{x}}$ to be equal to or as close as possible to the encoded value $\mathbf{x}$. That is, in general, we would like to minimize the error E (also called a *loss function*)

$$\arg\min_{\mathbf{D}} E = \int_{\mathbb{X}} \|\mathbf{x} - \hat{\mathbf{x}}\| \, \mathrm{d}\mathbf{x} = \int_{\mathbb{X}} \|\mathbf{x} - \mathbf{D}\mathbf{a}(\mathbf{x})\| \, \mathrm{d}\mathbf{x}, \qquad (3)$$

where $\mathbb{X}$ is a compact subspace of the possible represented values in $\mathbb{R}^d$ we are interested in.

📌 **Note:** The dynamic range of a neural representation is limited due to neural saturation and noise – it is not possible to represent all real numbers in a single neuron population. This is why we are using this "weird" integral over a subspace $\mathbb{X}$ – we must restrict ourselves to a range of numbers we want to be able to decode well.

The quantifier *compact* is just a mathematical technicality that ensures that $\mathbb{X}$ is actually a "closed range" we can integrate over, and not just individual vectors picked from $\mathbb{R}^d$.

Mathematical derivation   In order to make the math a little more digestible, assume that we want to find the decoders for scalar represented values ($d = 1$) and that the represented values we are interested in are over the interval $\mathbb{X} = [-1, 1]$. The above equation becomes

$$\arg\min_{\mathbf{d}} E = \arg\min_{\mathbf{d}} \sqrt{\int_{-1}^{1} (x - \langle \mathbf{d}, \mathbf{a}(x)\rangle)^2 \, dx} = \arg\min_{\mathbf{d}} \frac{1}{2}\int_{-1}^{1} \left(x - \sum_{i=1}^{n} d_i a_i(x) \right)^2 \, dx. \quad (4)$$

This equality holds because eliminating the square root and adding a factor $\frac{1}{2}$ does not change the location of the minimum. Since this is a quadratic equation we know that there is exactly one extremum. In this particular case, we also know that this extremum must be a minimum (since the quadratic term has a positive sign). Hence, we can find the $\mathbf{d}$ that minimizes the error by differentiating with respect to individual d_i and setting the derivative to zero:

$$\frac{\partial E}{\partial d_i} = \frac{1}{2}\int_{-1}^{1} 2\left(x - \sum_{j=1}^{n} d_j a_j(x) \right)(-a_i(x)) \, dx = \int_{-1}^{1} \sum_{j=1}^{n} a_j(x) d_j a_i(x) \, dx - \int_{-1}^{1} a_i(x) x \, dx \overset{!}{=} 0.$$

Rearranging yields the following equality at $\frac{\partial E}{\partial d_i} = 0$:

$$\int_{-1}^{1} a_i(x) x \, dx = \int_{-1}^{1} \sum_{j=1}^{n} a_j(x) d_j a_i(x) \, dx = \sum_{j=1}^{n} d_j \left(\int_{-1}^{1} a_j(x) a_i(x) \, dx \right).$$

Upon closer inspection we see that this is a system of n linear equations over $\mathbf{d} = (d_1, \ldots, d_n)$. Hence, we can write this in matrix notation as $\boldsymbol{\Upsilon} = \boldsymbol{\Gamma}\mathbf{d}$, where

$$(\boldsymbol{\Upsilon})_i = \int_{-1}^{1} a_i(x) x \, dx, \qquad\qquad (\boldsymbol{\Gamma})_{ij} = \int_{-1}^{1} a_j(x) a_i(x) \, dx.$$

Further evaluation of the integrals is – generally speaking – not possible in closed form. What we can do however is to approximate the integrals by (randomly) sampling. We pick N sample points $x_1, \ldots, x_N$. Then, the above equation becomes (approximately)

$$\frac{1}{N}\sum_{k=1}^{N} a_i(x_k) x_k = \frac{1}{N}\sum_{j=1}^{n} d_j \sum_{k=1}^{N} a_j(x_k) a_i(x_k).$$

The factor $1/N$ accounting for the discretisation of the integral cancels out. In matrix notation, letting $(\mathbf{A})_{ik} = a_i(x_k)$ and $\xi = (x_1, \ldots, x_k)$ we get

$$\mathbf{d} = \boldsymbol{\Gamma}^{-1}\boldsymbol{\Upsilon} \approx (\mathbf{A}\mathbf{A}^T)^{-1}\mathbf{A}\xi^T, \qquad\qquad \text{where } \boldsymbol{\Upsilon} \approx \mathbf{A}\xi \text{ and } \boldsymbol{\Gamma} \approx \mathbf{A}\mathbf{A}^T.$$

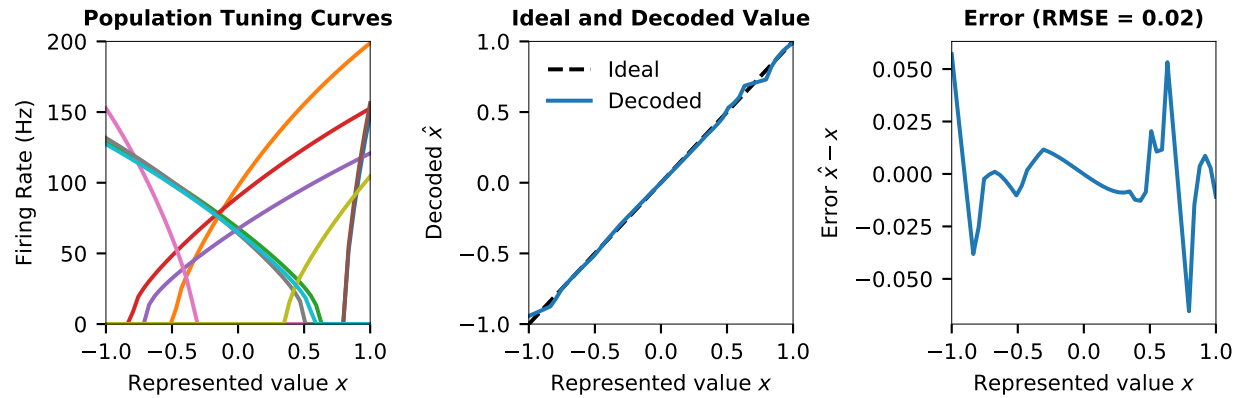In general, solving eq. (3) directly (i.e., without assuming $d = 1$ and $\mathbb{X} = [-1, 1]$), we get

Figure 4: Example of decoding the represented values from a population of neurons. *Left:* Tuning curves for a population of ten neurons. Tuning curves were chosen such that the x-intercepts of the tuning curves are sampled from a uniform distribution over the interval $[-1, 1]$. The maximum firing rate over that interval is uniformly sampled from $[100, 200]$. Neuron encoders were randomly selected to be either $+1$ or -1. *Middle:* Represented value and the decoded value on the same plot. *Right:* Decoding error $\hat{x} - x$. ⌨ *Code*

> *(Computing decoders without taking noise into account)*
>
> $$\mathbf{D}^\mathsf{T} \approx (\mathbf{AA}^T)^{-1}\mathbf{AX}^\mathsf{T}. \qquad (5)$$

Here, $A = \big(\boldsymbol{\alpha}(\mathbf{x}_1), \ldots (\mathbf{x_N})\big)$ is a matrix containing the population activities as column vectors for each sample $\mathbf{x}_k$. $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_k)$ is a matrix containing each input sample as a column vector.

To summarize, in order to solve for decoders we first randomly sample a set of N values $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_k)$ we would like to represent. Using the encoding equation in eq. (2) we then compute what the population activities would be for each of these samples, giving us $A = \big(\boldsymbol{\alpha}(\mathbf{x}_1), \ldots (\mathbf{x_N})\big)$. We then plug these matrices into eq. (5), giving us the decoding matrix $\mathbf{D}$. Figure 4 shows an example depicting the overall encoding and decoding process.

✎ **Note:** *Least squares.* The problem we just solved is a *linear least squares optimisation problem*, and the solution given in eq. (5) has been independently discovered in the early 19th century by the mathematicians Adrien-Marie Legendre and Carl Friedrich Gauss.

✎ **Note:** *Moore-Penrose Pseudo Inverse.* The term "$(\mathbf{AA}^T)^{-1}\mathbf{A}$" is also called Moore-Penrose Pseudo Inverse of a matrix $\mathbf{A}$. It is sometimes written as $\mathbf{A}^+$. The name "Pseudo Inverse" stems from the fact that it performs an inverse-like operation for non-square matrices.

🐍 In Python, you can solve the above linear least-squares problem using the following code:

```python
import numpy as np
A = np.array(...) # n x N array
```

```
X = np.array(...) # d x N array
D = np.linalg.lstsq(A.T, X.T, rcond=None)[0].T
```

Have a look at the documentation of `lstsq` for a precise description of its behaviour.

4.2 Sources of Noise in Neural Systems

We now know how to compute linear decoders **D** that estimate the value represented by an ensemble of neurons. However, we implicitly made the assumption that each neuron perfectly implements its response curve $G[J]$. We know that this is not true in biological systems.

Noise due to spike rate estimation errors The response curve $G[J]$ is only a first-order approximation of a neuron's behaviour. We know that in real biological systems neurons communicate using spikes – and it is impossible to perfectly estimate it's firing rate over a short time window (cf. the time and frequency uncertainty principle [7]). This measurement error can be interpreted as *noise*.

Biological sources of error Nervous systems contain various sources of error, including, but not limited to the following (cf. [8], Chapter 5.1)

- **Axonal jitter.** The action potential transmission speed varies between spikes. This is because action potentials are transported *actively*, i.e., they do not travel along the axon due to passive electrical properties of the axon, but are constantly renewed at the so called "Nodes of Ranvier".

- **Neurotransmitter vesicle release failure.** Only 10-30% of pre-synaptic spikes generate a post-synaptic event – one reason being that the release of neurotransmitter vesicles is stochastic.

- **Amount of neurotransmitter per vesicle.** The amount of neurotransmitters in a vesicle varies between vesicles. The intensity of the post-synaptic response depends on the specific vesicle that releases neurotransmitter into the synaptic cleft.

- **Ion channel noise.** The excitatory and inhibitory post-synaptic currents are generated by ion-channels in the neuron's cell membrane opening and closing in response to receiving neuro transmitters. These ion-channels are binary: they can either be fully open or fully closed. The graded nature of post-synaptic currents stems from the fact that many ion-channels open and close in a stochastic fashion.

- **Thermal noise.** In general, on the level of molecular biology, many processes rely on stochastic events triggered by thermal noise – this can be seen as an explanation of the noisiness of – among others – the above processes.

- **Network effects.** Even simple, non-noisy recurrent networks of excitatory and inhibitory spiking neurons can produce highly irregular spike patterns.
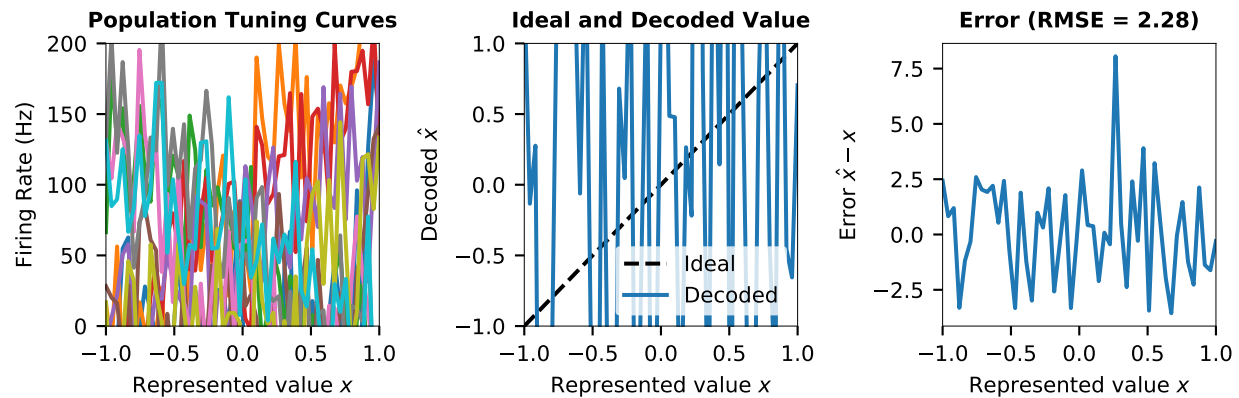
Figure 5: Decoding the represented value from a population of neurons with added noise (Gaussian, $\mu = 0$, $\sigma = 0.1 \max(A)$). See fig. 4 for the complete description. ▦ *Code*

Given these considerations, we should add a zeroth principle to the NEF:[1]

> **NEF Principle 0 – Noise**
> Biological neural systems are subject to significant amounts of noise from various sources. Any analysis of such systems must take the effects of noise into account.

Modelling Noise To test whether our modelling framework still holds when we take noise into account, we should somehow be able to include noise to our models. There are two potential ways how we can accomplish this. Either we add more detail to our neuron models and simulate the noise sources, or we – for now – simply add noise from a distribution to our neural activities **α**.

Figure 5 depicts the decoding accuracy when adding Gaussian noise to the neural activities. As clearly visible, our current method for computing the decoders is incapable of properly dealing with this noise.[2] Can we do better? Let's go back to the drawing board and do some more math.

4.3 Computing Decoders Taking Noise Into Account

The formula for computing the decoders given in eq. (5) has three problems.

1. As discussed above, neural activities are always noisy (see discussion above) – we did not account for this in our derivation of eq. (5).

2. From a mathematical perspective, $\mathbf{A}\mathbf{A}^\mathsf{T}$ may not always be invertible – this may happen if individual rows/columns in $\mathbf{A}$ are linearly dependent.

---

[1] In the text this principle is called an "addendum" and listed as the fourth principle. I think that this principle is so fundamental that it should be number zero in the list.   [2] To be fair, the example depicted in fig. 5 has been consciously chosen to be close to a worst-case scenario.

3. From a numerics perspective, even if $\mathbf{A}\mathbf{A}^\mathsf{T}$ happens to be invertible (in a mathematical sense), doing so may be numerically instable if individual rows/columns in $\mathbf{A}$ are almost linearly dependent. This may happen if some neurons have approximately the same parameters.

Interestingly, addressing the first problem, i.e., taking noise into account when deriving eq. (5), will magically solve the other two problems.

Mathematical derivation Again, we derive the math under the assumption that $d = 1$ (i.e., we are representing scalar values), but the result can be easily generalised to multiple dimensions.

Furthermore, we assume that there is independent noise sampled from a Gaussian distribution with mean zero and a standard deviation σ superimposed onto each neural activitiy. That is, whenever we try to measure the neural activity of neuron i for an input x, we really get a value $a_i(\mathbf{x}) + \eta$, where η is a random variable that has been sampled from $\mathcal{N}(0, \sigma^2)$.

Plugging this into our derivation from above and minimizing the expectation value $\mathbb{E}$, we get

$$\arg\min_{\mathbf{d}} E = \arg\min_{\mathbf{d}} \mathbb{E}\left[\frac{1}{2} \int_{-1}^{1} \left(x - \sum_{i=1}^{n} d_i(a_i(x) + \eta) \right)^2 \mathrm{d}x \right]_\eta \qquad \text{where } \eta \sim \mathcal{N}(0, \sigma^2)$$

$$= \arg\min_{\mathbf{d}} \mathbb{E}\left[\int_{-1}^{1} \left(x - \sum_{i=1}^{n} d_i a_i(x) - \sum_{i=1}^{n} d_i \eta \right)^2 \mathrm{d}x \right]_\eta$$

$$= \arg\min_{\mathbf{d}} \frac{1}{2} \int_{-1}^{1} \left(x - \sum_{i=1}^{n} d_i a_i(x) \right)^2 \mathrm{d}x + \sum_{i=1}^{n}\sum_{j=1}^{n} d_i d_j \mathbb{E}\left[\eta_i \eta_j \right]_{\eta_i, \eta_j}.$$

Note that the cross-terms in the above double-sum disappear: the expectation value of the product between two independent, mean zero random variables is zero. We get

$$\arg\min_{\mathbf{d}} E = \arg\min_{\mathbf{d}} \frac{1}{2} \int_{-1}^{1} \left(x - \sum_{i=1}^{n} d_i a_i(x) \right)^2 \mathrm{d}x + \sum_{i=1}^{n} d_i^2 \mathbb{E}\left[\eta_i^2 \right]_{\eta_i}.$$

Since the mean of the distribution is zero, the remaining expectation value is exactly the variance of η:

$$\arg\min_{\mathbf{d}} E = \arg\min_{\mathbf{d}} \frac{1}{2} \int_{-1}^{1} \left(x - \sum_{i=1}^{n} d_i a_i(x) \right)^2 \mathrm{d}x + \sigma^2 \sum_{i=1}^{n} d_i^2. \qquad (6)$$

> **Note:** *Equivalence of "taking Gaussian noise into account" and L_2-regularisation.* Equation (6) can be interpreted in a different, but extremely useful manner. Essentially, our error expression is exactly the same as when we set out to derive this equation without noise, cf. eq. (4). The difference is that we penalize the magnitude of the coefficients d_i: the larger $|d_i|$, the larger the error E will be.
>
> Adding such a weight penalisation term is commonly referred to as *regularisation*. Gener-
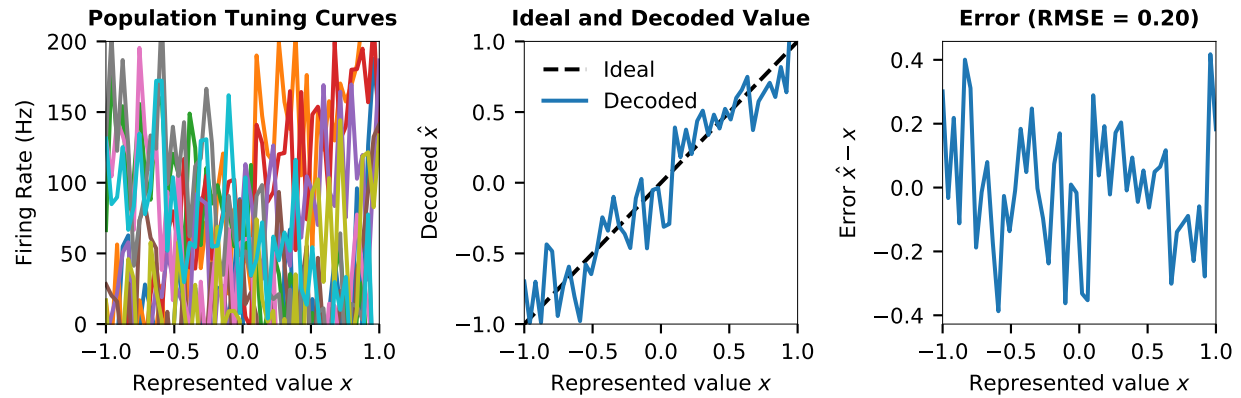
Figure 6: Decoding the represented value from a population of neurons with added noise (Gaussian, $\mu = 0$, $\sigma = 0.2 \max(A)$) while accounting for noise. See fig. 4 for the complete description. ⌨ *Code*

ally, L_ℓ-regularisation (where ℓ is a positive number, usually one or two) adds the following loss term to E:

$$\lambda \sum_{i=1}^{n} |d_i|^\ell \,,$$

where λ is the so called "regularisation factor". Hence, in the case above, we are using L_2 regularisation with $\lambda = \sigma^2$.

The effect of regularisation is the following: when solving for **d**, we try to balance between minimizing the error and not having large decoder coefficients d_i^2. Since the coefficients are being squared, the optimal solution will be small, non-zero decoder coefficients that effectively "average" over the activities from multiple neurons. This averaging is also able to reduce the impact of noise on the activities.

We can now continue to solve for **d** as above. After some linear algebra, we arrive at the following equation (here, directly for arbitrary represented dimensions $d \geq 1$)

(Computing decoders taking noise into account)

$$\mathbf{D}^\mathsf{T} \approx \left(\mathbf{A}\mathbf{A}^\mathsf{T} + N\sigma^2 \mathbf{I} \right)^{-1} \mathbf{A}\mathbf{X}^\mathsf{T}, \text{ where } \mathbf{I} \text{ is the } n \times n \text{ identity matrix} \qquad (7)$$

Adding a scaled version of the identity matrix to $\mathbf{A}\mathbf{A}^T$ ensures that the resulting matrix is always invertible. If σ^2 is large enough, this inversion is possible in a numerically stable way. As visible in fig. 6, the decoding error is now in an acceptable range, even if we add noise to the pre-population.

In Python, you can solve the above regularised linear least-squares problem using the following code:

```python
import numpy as np
A = np.array(...) # n x N array
```

```
X = np.array(...) # d x N array
D = np.linalg.lstsq(
        A @ A.T + N * np.square(sigma) * np.eye(n), A @ X.T,
        rcond=None)[0].T
```

Using `lstsq` is numerically more stable than manually inverting the matrix using a function such as `np.linalg.inv`.

📌 **Note:** Alternative derivation of eq. (7). The matrix $\mathbf{A} \in \mathbb{R}^{n \times N}$ contains samples of neural activities for different represented values. In particular, we have n rows (one for each neuron in the population) for N samples $\mathbf{x}_1, \ldots, \mathbf{x}_N$ in each row. Furthermore, we assume that there is Gaussian noise superimposed onto our measurements. So we have

$$\mathbf{A} = \begin{pmatrix} a_1(\mathbf{x}_1) + \eta_{1,1} & \ldots & a_1(\mathbf{x}_N) + \eta_{1,N} \\ \vdots & \ddots & \vdots \\ a_n(\mathbf{x}_1) + \eta_{n,1} & \ldots & a_n(\mathbf{x}_N) + \eta_{n,N} \end{pmatrix}, \quad \text{where } \eta_{i,j} \sim \mathcal{N}(0, \sigma^2)$$

$$= \begin{pmatrix} a_1(\mathbf{x}_1) & \ldots & a_1(\mathbf{x}_N) \\ \vdots & \ddots & \vdots \\ a_n(\mathbf{x}_1) & \ldots & a_n(\mathbf{x}_N) \end{pmatrix} + \begin{pmatrix} \eta_{1,1} & \ldots & \eta_{1,N} \\ \vdots & \ddots & \vdots \\ \eta_{n,1} & \ldots & \eta_{n,N} \end{pmatrix}$$

$$= \mathbf{A}_{\mathrm{GT}} + \mathbf{E}.$$

The matrix $\mathbf{A}_{\mathrm{GT}}$ is a hypothetical matrix of unknown "ground truth" values and $\mathbf{E}$ is the matrix of zero mean Gaussian noise terms. Looking at the term $\mathbf{A}\mathbf{A}^\mathsf{T}$ from eq. (5) we find

$$\mathbf{A}\mathbf{A}^\mathsf{T} = (\mathbf{A}_{\mathrm{GT}} + \mathbf{E})(\mathbf{A}_{\mathrm{GT}} + \mathbf{E})^\mathsf{T} = \mathbf{A}_{\mathrm{GT}}\mathbf{A}_{\mathrm{GT}}^\mathsf{T} + \mathbf{A}_{\mathrm{GT}}\mathbf{E}^\mathsf{T} + \mathbf{E}\mathbf{A}_{\mathrm{GT}}^\mathsf{T} + \mathbf{E}\mathbf{E}^\mathsf{T}.$$

The expectation value of the terms $\mathbf{A}_{\mathrm{GT}}\mathbf{E}^\mathsf{T}$ and $\mathbf{E}\mathbf{A}_{\mathrm{GT}}^\mathsf{T}$ is zero:

$$\mathbb{E}\big[(\mathbf{A}_{\mathrm{GT}}^\mathsf{T}\mathbf{E})_{ij}\big]_{\mathbf{E}} = \mathbb{E}\big[(\mathbf{E}^\mathsf{T}\mathbf{A}_{\mathrm{GT}})_{ij}\big]_{\mathbf{E}} = \mathbb{E}\big[\sum_{k=1}^{N} a_i(\mathbf{x}_k)\eta_{j,k}\big]_{\mathbf{E}} = \mathbb{E}\big[\sum_{k=1}^{N} a_j(\mathbf{x}_k)\eta_{i,k}\big]_{\mathbf{E}} = 0,$$

whereas the term $\mathbb{E}\big[\mathbf{E}^\mathsf{T}\mathbf{E}\big]$ evaluates to $N\sigma^2\mathbf{I}$. Hence, for $N \to \infty$ (which, according to the law of large numbers, is bringing us close to the expectation value), we get

$$(\mathbf{A}\mathbf{A}^T)^{-1}\mathbf{A}\mathbf{X}^\mathsf{T} \approx (\mathbf{A}_{\mathrm{GT}}\mathbf{A}_{\mathrm{GT}}^\mathsf{T} + N\sigma^2\mathbf{I})^{-1}\mathbf{A}_{\mathrm{GT}}\mathbf{X}^\mathsf{T}.$$

Comparing this to eq. (7), we see that we implicitly assume in eq. (7) that our randomly sampled activities $\mathbf{A}$ are equal to the hidden ground truth $\mathbf{A}_{\mathrm{GT}}$, which – on average – is correct, since we assume that $\eta_{i,j}$ has a zero mean. We then manually add the term $N\sigma^2\mathbf{I}$, which we would get if we let N go towards infinity – without actually needing infinitely many samples. Pretty clever, since this saves us a lot of time!

However, the other way round, this derivation shows us that we don't really need to add the term $N\sigma^2\mathbf{I}$ to our sampled activities if only N is large enough (in particular, $N \gg n$).

4.4 Analysing Sources of Error

———————————————— 🚧 To be filled in 🚧 ————————————————

5   Building a model of a neural population

With the knowledge we gained above, let's discuss how to practically build a model of a neuron population representing a d-dimensional vector $\mathbf{x}$. This can be roughly mapped onto the methodology described in [5], Chapter 1.5.

Step 1 and 2: System Description and Design Specification   First, we have to make a decision regarding the dimensionality d of the quantity $\mathbf{x}$ we want to represent, as well as what the range of values $\mathbb{X}$ is. Normally, we define $\mathbb{X}$ as a d-dimensional hyper-ball with radius r, that is $\mathbb{X} = \left\{ \mathbf{x} \mid \|\mathbf{x}\| \leq r, \mathbf{x} \in \mathbb{R}^d \right\}$ (typically $r = 1$).

Furthermore, we have to specify how many neurons n are in the population, and what their tuning curves are. This will have an impact on the precision with which the values are represented. The tuning curves will determine the parameters we choose for α_i, J_i^{bias}, $\mathbf{e}_i$.

In case we have no specific information about the tuning curves, we select these parameters for each neuron in the population such that. . .

- . . . the encoders $\mathbf{e}_i$ should be uniformly sampled from the unit-sphere. This can be achieved by sampling each encoder component from a normal distribution (i.e., a Gaussian distribution with mean zero and variance one) and normalising the resulting vectors to length one.

- . . . the maximum neural firing rate is limited to the maximum rate we observe in biology. The maximum firing rate is the rate we get when the represented value $\mathbf{x}$ is aligned with the encoding vector $\mathbf{e}_i$, i.e. $\langle \mathbf{x}, \mathbf{e}_i \rangle = r$. Typically, the maximum firing rates of a population should be uniformly distributed between $100\,\mathrm{s}^{-1}$ to $200\,\mathrm{s}^{-1}$. Per default, Nengo uses a maximum firing rate range between $200\,\mathrm{s}^{-1}$ to $400\,\mathrm{s}^{-1}$, which is typically to large for most brain regions.

- . . . the x-intercepts, i.e. the value of $\langle \mathbf{x}, \mathbf{e}_i \rangle$ at which the neuron just starts to have a nonzero activity should be uniformly sampled from $[-r, r]$.

Step 3: Implementation   After we have chosen the parameters above, we can encode values $\mathbf{x}$ in the population. In other words, we can take an $\mathbf{x}$ and compute the neural population response $\mathbf{a}(\mathbf{x})$ according to the encoding equation. We can then uniformly sample N samples $\mathbf{x}_1, \ldots, \mathbf{x}_N$ from $\mathbb{X}$ and compute neural activation matrix $\mathbf{A}$.

Finally, this allows us to compute the decoders $\mathbf{D}$ according to eq. (7), where σ depends on the amount of noise we expect in the neural activities. As a rule of thumb, a value of $\sigma = 0.2 \max\{\mathbf{A}\}$ is a good starting point.

📌 **Note:** This might all still seem a little abstract – for this reason, you will implement all these steps in Assignment 1.

5.1 Example: Horizontal Eye Control (1D)

——————————————— 🚧 To be filled in 🚧 ———————————————

5.2 Example: Arm Movements (2D)

——————————————— 🚧 To be filled in 🚧 ———————————————

5.3 Example: Higher-dimensional Tuning

——————————————— 🚧 To be filled in 🚧 ———————————————

References

[1] David H. Hubel and Torsten N. Wiesel. "Receptive Fields of Single Neurones in the Cat's Striate Cortex". In: *The Journal of physiology* 148.3 (1959), pp. 574–591. DOI: 10.1113/jphysiol.1959.sp006308. URL: https://physoc.onlinelibrary.wiley.com/doi/abs/10.1113/jphysiol.1959.sp006308.

[2] *ITU-R M.1677-1: International Morse Code*. Oct. 2009. URL: https://www.itu.int/rec/R-REC-M.1677-1-200910-I/.

[3] Michael D. Mann. *The Nervous System In Action*. 1997. URL: http://michaeldmann.net/The%5C%20Nervous%5C%20System%5C%20In%5C%20Action.html (visited on 01/13/2020).

[4] Y. Katsuki. "Neural Mechanism of Auditory Sensation in Cats". In: *Sensory Communication*. Ed. by W.A. Rosenblith. MIT Press, 1969. ISBN: 978-0-262-51842-0.

[5] Chris Eliasmith and Charles H. Anderson. *Neural Engineering: Computation, Representation, and Dynamics in Neurobiological Systems*. Cambridge, Massachusetts: MIT Press, 2003. 380 pp. ISBN: 978-0-262-55060-4.

[6] Aman B Saleem et al. "Integration of Visual Motion and Locomotion in Mouse Visual Cortex". In: *Nature Neuroscience* 16.12 (Dec. 2013), pp. 1864–1869. ISSN: 1546-1726. DOI: 10.1038/nn.3567.

[7] Dennis Gabor. "Theory of Communication. Part 1: The Analysis of Information". In: *Journal of the Institution of Electrical Engineers-Part III: Radio and Communication Engineering* 93.26 (1946), pp. 429–441.

[8]   Wulfram Gerstner and Werner M. Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. LCCB: 2002067657. Cambridge University Press, 2002. ISBN: 978-0-521-89079-3.