SYDE 556/750
Simulating Neurobiological Systems
Lecture 5: Feed-Forward Transformation

Andreas Stöckel and Chris Eliasmith

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

September 30, 2024

Accompanying Readings: Chapter 6 of Neural Engineering

Contents

1 Introduction

Note: Until now, we have been concerned with *representation* in individual populations of neurons. However, we ultimately want to be able to build neural *networks*. This means that we have to find a systematic way of connecting neural populations. Optimally, the connections should be chosen in such a way that information represented in the populations is *transformed* in a useful way.

We postulated that groups of neurons represent d-dimensional quantities $\mathbf{x}$ using nonlinear encoding. The encoding process can be further separated into two stages: translating an input $\mathbf{x}$ into a per-neuron input current J_i and applying the neural nonlinearity.

The current J_i that is being injected into the i-th neuron of a population is defined as

$$J_i = \alpha_i \langle \mathbf{e}_i, \mathbf{x} \rangle + J_i^{\mathrm{bias}} \,. \tag{1}$$

For rate neurons, this input current is turned into a spike rate according to the rate approximation $a_i = G[J_i]$. In the context of spiking neurons, the input current is translated into a spike train $a_i(t)$, a sum of Dirac-δ pulses.

We further postulated that the value being represented by a population can be estimated using a linear decoder $\mathbf{D}$. For spiking neurons, we added a filtering step using a filter h

$$\hat{\mathbf{x}} = \mathbf{D}\boldsymbol{\alpha} \quad \text{for rate neurons,} \qquad \hat{\mathbf{x}}(t) = \big((\mathbf{D}\boldsymbol{\alpha}(t)) \ast h\big)(t) \quad \text{for spiking neurons.} \tag{2}$$

Note: *Decoder computation.* We discussed two methods for computing decoders $\mathbf{D}$.

For the first method, we generated a random set of samples arranged in a matrix $\mathbf{X}$. Using the spike rate approximation $G[J]$, we computed an activity matrix $\mathbf{A}$ and obtained $\mathbf{D}$ using the solution to the L_2-regularised least-squares problem

$$\mathbf{D}^T = (\mathbf{A}\mathbf{A}^T + N\sigma^2\mathbf{I})^{-1}\mathbf{A}\mathbf{X}^T \,,$$

where we estimate σ to match the deviation between the spike rate estimate and the actual spike rates occurring in the network.

The second solution was to use a random input function $\mathbf{x}(t)$, and the recorded (filtered) population spike trains $\boldsymbol{\alpha}(t)$. We discretised these functions into matrices $\mathbf{A}$ and $\mathbf{X}$ and used the unregularised solution to the least-squares problem to obtain $\mathbf{D}$

$$\mathbf{D}^T = (\mathbf{A}\mathbf{A}^T)^{-1}\mathbf{A}\mathbf{X}^T \,.$$

As mentioned in the last lecture, these two methods should result in approximately the same decoders. While the second method is technically superior, as it accurately characterises the noise present in the neural population, the first method is computationally much cheaper. This is why – from now on – we will use the first method to compute the decoders. We can then use this decoding matrix in conjunction with spiking neurons.

Of course, just describing an individual population of neurons is not really useful when building large-scale brain models. We ultimately would like to describe how information is *transformed* while it is sent from one population of neurons to another. In the Neural Engineering Framework, connections between neural populations are performing these transformations.

NEF Principle 2 – Transformation
Connections between populations describe transformations of neural representations. These transformations are functions of the variables that are represented by neural populations.

When looking at transformations from the perspective of large-scale modelling, researchers are usually confronted with two very different questions.

- **How do brains *learn* transformations?** In other words, how are the connection weights between neuron populations formed in such a way during the lifetime of an animal that they implement a desired task.
- **What are *optimal* connection weights that compute a certain transformation?** Here, we assume that a brain has already learned to optimally perform a certain task. In that case, we would just like to know what the corresponding connection weights the system could have learned are. We would then like to use these weights in our model. Essentially, we are building a model of a system that is already an expert.

For now, we will mostly concern ourselves with the second question, i.e., we are trying to build models of "adult" or "expert" systems already capable of solving a certain task. We postulate what the transformation may be that the system has learned and compute the optimal weights that implement this transformation. We will talk about *learning*, i.e., building a system that learns connection weights while it is being executed in a later lecture.

Note: *Hypothesis generation and testing.* We should pause here and wonder what the scientific purpose of building neural models with optimal connection weights is. Especially since we do not assume that the brain itself employs the techniques for computing function decoders we discuss below. So, if not providing a theory of how synaptic weights are formed within brains, what is the purpose of the Neural Engineering Framework?

Most importantly, the Neural Engineering Framework can be used to test hypotheses regarding potential functions of neurobiological systems. That is, we can test whether such a system could implement a certain function *under optimal circumstances*. Conversely, if we are not able to implement a function using mathematically optimal connection weights (but taking other constraints into account), we can rule out this hypothesis.

Furthermore, building working functional models of neural systems remains challenging, even under the "optimal synaptic weights" assumption. Hence, the Neural Engineering Framework is not only an important tool for testing hypotheses ("can a function in theory be implemented in a certain way?"), but also for explorative generation of hypotheses ("this works within the NEF, could the brain be doing this?").

2 Building a Communication Channel

The simplest possible transformation is the identity function. Assume that we have two neural populations A and B, where A represents a d-dimensional vector $\mathbf{x} \in \mathbb{R}^d$ using n neurons, and B represents a vector $\mathbf{y} \in \mathbb{R}^{d'}$ using m neurons. We would now like to send the value $\mathbf{x}$ from A to B (assuming that $d = d'$), such that when simulating the network $\mathbf{x}$ and $\mathbf{y}$ are approximately the same at all times. This type of transformation is also called a *communication channel*, since we are merely sending information from one point in the system to another point in the system without altering it.

Let's try to think about how to build a communication channel "computing" $\mathbf{y} = \mathbf{x}$ in a biologically plausible neural network. Then, in the next section, we will take what we have learned and apply it to arbitrary transformations $\mathbf{y} = f(\mathbf{x})$.

2.1 Sequential Decoding and Encoding

Given what we know so far, the simplest way of building a communication channel is by sequential encoding and decoding. We are decoding the value $\hat{\mathbf{x}}$ represented by a population A, followed by subsequent re-encoding of that value as $\mathbf{y}$, the value represented by population B. Figure 1 depicts the overall setup, including the individual encoding stages, neural nonlinearity and the decoder.

An example of this type of communication channel is depicted in fig. 2. As we can see, sequential decoding and re-encoding to transfer to the represented value from population A to population B seems to work, although there is a slight time delay between the two populations due to the synaptic filters.

Note: *Location of the synaptic filter.* Notice that the synaptic filters h are missing in fig. 1. In biology, the synaptic filter would be placed at each individual connection from a pre-neuron to a post-neuron (see below).

However, since we assume that h is a linear filter (this is because we model the filter using convolution) and that it has a *unit DC gain*, placement of the filter does – mathematically speaking – not matter as long as it is placed before the nonlinearity of each neuron layer.

2.2 Synaptic Weights

Evidently, the method discussed above works. Unfortunately, it is biologically implausible. While neurons are often characterised as belonging to different "populations" or "layers", there is no equivalent of a dedicated decoder and encoder in biological neural networks that somehow translates between these layers. Instead, as we have discussed in previous lectures, individual neurons are connected via *synapses*.

Typically, there is an individual synapse for each connection from a pre-neuron j to a post-neuron i. Mathematically, we model synapses as having two properties: a synaptic weight w_{ij},

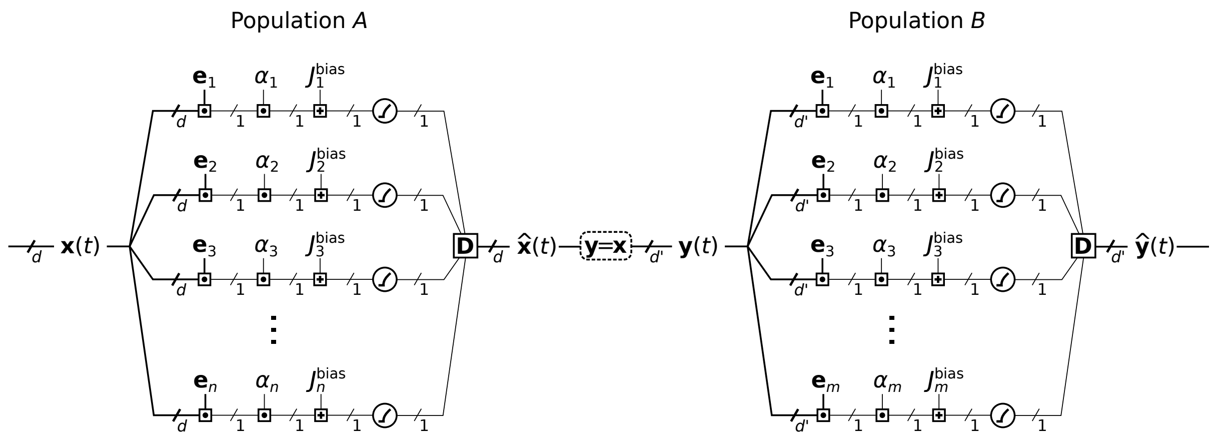

Population *A*
Population *B*
$\mathbf{x}(t)$
$\hat{\mathbf{x}}(t)$
$\mathbf{y}=\mathbf{x}$
$\mathbf{y}(t)$
$\hat{\mathbf{y}}(t)$
$\mathbf{e}_1$ α_1 J_1^{bias}
$\mathbf{e}_2$ α_2 J_2^{bias}
$\mathbf{e}_3$ α_3 J_3^{bias}
$\mathbf{e}_n$ α_n J_n^{bias}
$\mathbf{e}_m$ α_m J_m^{bias}
D
d
d'
1

Figure 1: Illustration of the process of encoding and decoding values represented in two populations of neurons. If we wanted to build a communication channel without taking biology into account, we could just decode the value represented in population *A* and then re-encode it in population *B*.

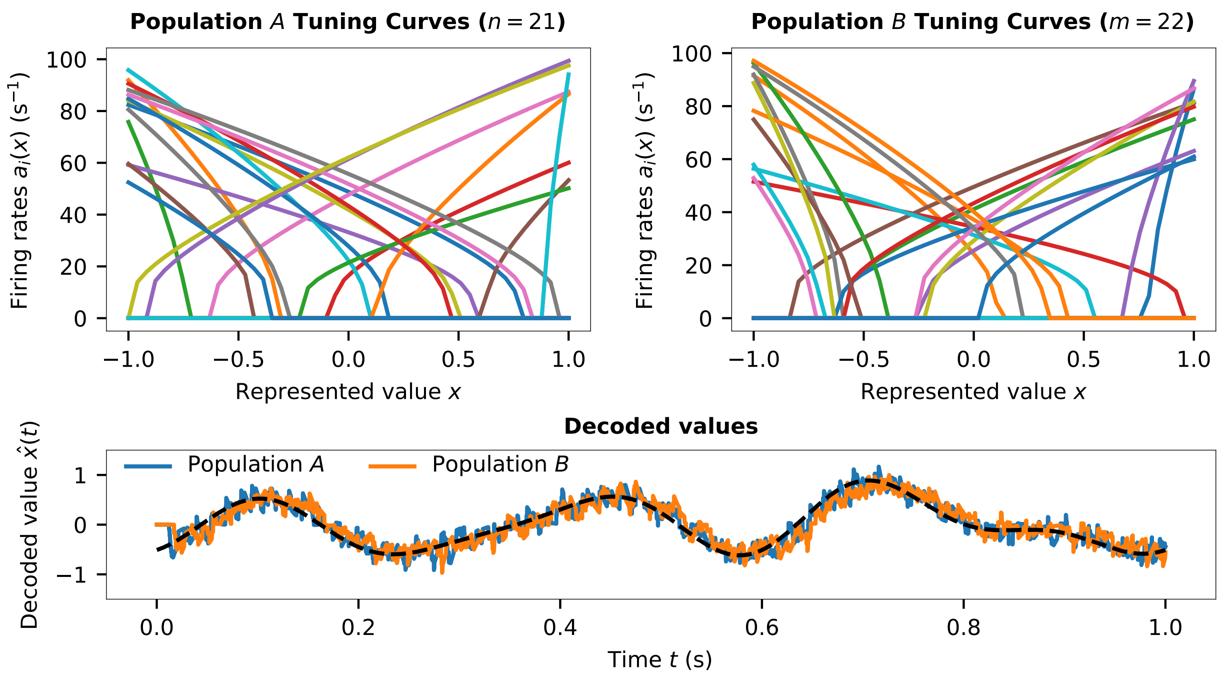


Figure 2: Experiment depicting the effect of sequential decoding and re-encoding. *Top:* tuning curves of the two neural populations. *Bottom:* Representations decoded from population *A* and *B*. The decoded output from *A* is fed as an input into the population *B*. Notice that the represented value is minimally delayed in population *B*.

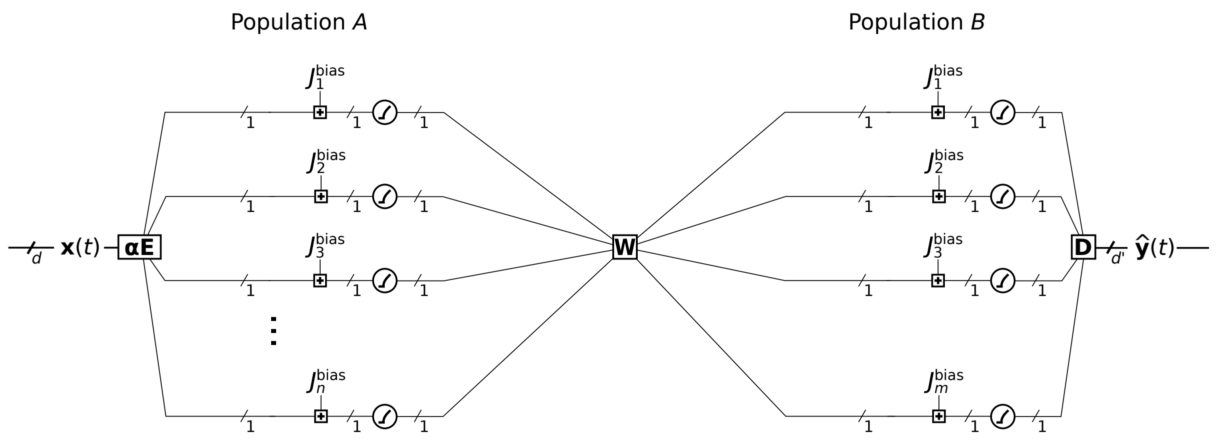


Figure 3: In biology, there are no encoders and decoders – individual neurons are connected via synapse, and we model each synapse as a synaptic weight w_{ij} (where i is the post-neuron index, and j is the pre-neuron index). We arrange these weights in a matrix $(\mathbf{W})_{ij} = w_{ij}$

as well as a synaptic filter h. The filter h describes the translation from a pre-neuron spike into a post-synaptic current, whereas the weight w_{ij} determines the magnitude of the current. Notice that we can arrange the synaptic weights w_{ij} into a matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$.

We can categorize synaptic weights w_{ij} into three different types:

- **Excitatory synapses** $w_{ij} > 0$**:** Excitatory synapses are characterized by evoking a positive post-synaptic current. That is, a spike from a pre-neuron arriving at an excitatory synapse will, on average, increase the firing rate of the post-neuron.
- **Inhibitory synapses** $w_{ij} < 0$**:** Conversely, inhibitory synapses are characterized by evoking a negative post-synaptic current. Spikes arriving at inhibitory synapses will decrease the average firing rate of the post-neuron.
- **No connection** $w_{ij} = 0$**:** By writing down a connection weight matrix $\mathbf{W}$, we implicitly assume that there is all-to-all connectivity between two populations of neurons. However, if a synaptic weights w_{ij} happens to be exactly zero, this is equivalent to there being no connection between the pre-neuron j and the post-neuron i.

Note: *Synaptic filters and synaptic weights.* Notice that we could collapse the synaptic weight w_{ij} into the synaptic filter. In other words, instead of assuming that all synapses have the same synaptic filter, we could assign a scaled version of h to each synapse, resulting in synaptic filters $h_{ij} = w_{ij}h$.

The reason why we are not building our model in this way is solely due to computational complexity. By assuming that all synapses have the same synaptic filter h, we can (as mentioned in the previous note) exploit the linearity of convolution and collapse all convolutions into a single one.

2.3 Synapse Models and Biological Correlates

As noted above, the synaptic weights w_{ij}, as well as the presence of a synaptic filter h, are integral parts of a synapse *model*. That is, these two properties are a high-level abstraction of what is actually happening in biology.

In particular, the synapse model h summarizes the mechanisms that result in the distinct shape of the post-synaptic current, whereas the magnitude of the post-synaptic current is summarized in the weights w_{ij}. In biology, the magnitude of the response is influenced by a multitude of factors.

- **Presence of a synapse.** If there is no connection between two neurons, the magnitude of the response, and thus w_{ij}, is zero.
- **Neurotransmitter type.** The type of neurotransmitter emitted by the pre-synapse determines whether the post-synaptic current is excitatory or inhibitory. In biology, each neuron can only produce one kind of neurotransmitter, a fact known as *Dale's principle* [1]. Correspondingly, neurons themselves act (in most cases) either excitatorily *or* inhibitorily on the post-neuron.
- **Synaptic physiology.** Among other factors, the number of synaptic vesicles, amount of neurotransmitter within a vesicle, number of receptors in the post synapse, determine the magnitude of the response.

  Furthermore, a pre-neuron may connect to the same post-neuron multiple times. As an extreme example of this, consider individual neurons in the Inferior Olive of the cerebellum, which connect to Purkinje cells via axons called "climbing fibres". Surprisingly, each Inferior Olive neuron connects to *exactly* one Purkinje cell, yet it does thousands of times.

  Note that these properties can change over time, essentially modulating the synaptic weights; these longer-term changes in synaptic physiology are exactly what we refer to as the aforementioned *learning*.

Note: *Current-based synapses.* In case we define our filter h as follows

$$h(t) = \begin{cases} \frac{1}{\tau} e^{-t/\tau} & \text{if } t \geq 0\,, \\ 0 & \text{otherwise}\,, \end{cases}$$

the specific synapse model we are using here is commonly referred to as a "current-based synapse model with exponential decay". "Current-based" because we assume that synapses directly translate spikes into synaptic currents, "exponential decay" because of the form of the synaptic filter [2].

Of course, far more complex synapse models exist. We are going to talk about some of those – in particular *conductance-based* synapses (not to be confused with conductance-based neuron models, such as the Hodgkin Huxley model) – in more detail when we talk about incorporating more biological realism into the Neural Engineering Framework.

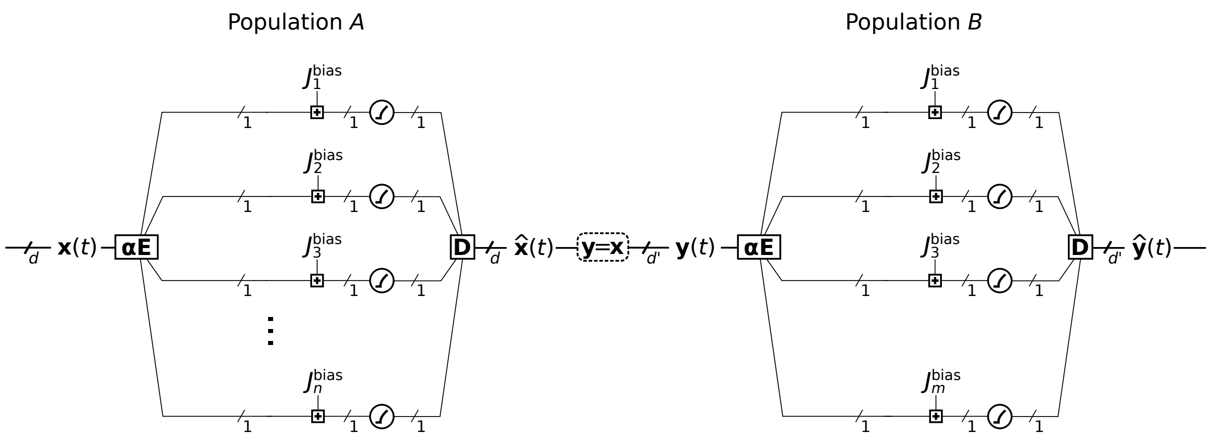


Figure 4: Decoding and encoding as linear operators. We can rearrange the linear parts of the encoding process into an encoding matrix **E**.

2.4 Computing Synaptic Weights

We now know that our previous approach – sequential decoding and re-encoding – works, but is biologically implausible because it does not describe neural connectivity in terms of synaptic weights w_{ij}. However, looking at the underlying equations we will see that this is not really true, and that we can in fact continue to use decoding and re-encoding as a mathematical abstraction.

Combining the current-encoding equation from eq. (1) with the decoding equation from eq. (2), we can write current that is being injected into the i-th neuron of the post-poulation as part of the decoding and re-encoding process as

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

Where $\boldsymbol{\alpha}(\mathbf{x})$ is the activity of the pre-population and $\mathbf{D}$ is the identity decoder of the pre-population. Expanding the scalar product and rearranging we get

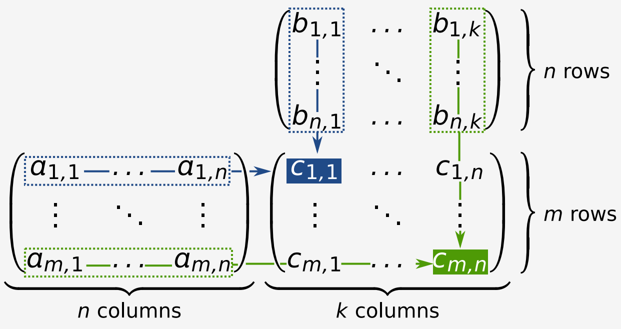
$$\begin{aligned} J_i &= \alpha_i \sum_{k=1}^{d} e_{ik} \sum_{j=1}^{n} d_{kj} a_j(\mathbf{x}) + J_i^{\text{bias}} = \alpha_i \sum_{k=1}^{d} \sum_{j=1}^{n} e_{ik} d_{kj} a_j(\mathbf{x}) + J_i^{\text{bias}} \\ &= \sum_{j=1}^{n} \underbrace{\alpha_i \sum_{k=1}^{d} e_{ik} d_{kj}}_{w_{ij}} a_j(\mathbf{x}) + J_i^{\text{bias}} = \sum_{j=1}^{n} w_{ij} a_j(\mathbf{x}) + J_i^{\text{bias}} \,. \end{aligned}$$

That is, we have now re-written the post-current of neuron i purely in terms of a weighted sum of the pre-activities and a bias current. The coefficients w_{ij} can be interpreted as exactly the weights we have been looking for.

Comparing fig. 1 to fig. 4 provides an alternative view on the above equations. Essentially, we can collapse the linear parts of the encoding process (multiplication with the encoding vectors $\mathbf{e}_i$ and scaling by α_i) into a single matrix $\mathbf{E}$. We then have two chained linear operators, described by the decoding matrix $\mathbf{D}$ and the encoding matrix $\mathbf{E}$. The synaptic weight matrix $\mathbf{W}$ is then given as the product between the two, that is $\mathbf{W} = \mathbf{ED}$.

Using these synaptic weights $\mathbf{W}$ is hence *mathematically equivalent* to sequential decoding and re-encoding. Whenever we want to have access to the biologically plausible synaptic weights we can just compute $\mathbf{ED}$; otherwise we can continue to use encoders and decoders.

Note: *Low-rank matrices, factorization, and computational complexity.* There is a good reason why we should continue to use sequential decoding and re-encoding in our simulations of neurobiological systems: once again, this boils down to computational complexity.



The above figure illustrates the process of multiplying to matrices $\mathbf{C} = \mathbf{AB}$, where $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{B} \in \mathbb{R}^{n \times k}$, and $\mathbf{C} \in \mathbb{R}^{m \times k}$. Naïvely, each cell in $\mathbf{C}$ is the inner product between two vectors of size n. Since there are $m \times k$ cells in the matrix $\mathbf{C}$, the overall complexity is in $\mathcal{O}(mkn)$, or $\mathcal{O}(n^3)$ for the special case $m = k = n$. There are more clever, yet mostly impractical algorithms that (as of writing) bring the complexity down to $\mathcal{O}(n^{2.3728639})$ (see Wikipedia).

The computational complexity of multiplying the pre-population activities $\mathbf{a} \in \mathbb{R}^n$ with the weight matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$ is hence $\mathcal{O}(nm)$, that is $\mathcal{O}(n^2)$ for the special case $n = m$.

Conversely, assume that the dimensionality represented by the neuron populations A and B is d. The decoding process $\hat{\mathbf{x}} = \mathbf{D}\mathbf{a}$ has a computational complexity of $\mathcal{O}(dn)$, the encoding process $\mathbf{J} = \mathbf{E}\hat{\mathbf{x}} + \mathbf{J}_\mathrm{bias}$ has a complexity of $\mathcal{O}(dm)$. That makes for a total complexity of $\mathcal{O}(d(n+m))$, which is approximately $\mathcal{O}(dn)$ for $n = m$. Further assuming that d is a small constant we get a linear complexity $\mathcal{O}(n)$ for the sequential decoding and re-encoding process, compared to the quadratic complexity $\mathcal{O}(n^2)$ for the full weight matrix.

Correspondingly, sequential decoding and re-encoding saves a lot of time compared to using actual synaptic weights, and is one of the reasons why Nengo is so fast compared to other spiking neural network simulators.

This trick is a special case of a more general concept called *matrix factorisation*. That is, matrix multiplication with a matrix $\mathbf{A} \in \mathbb{R}^{m \times k}$ is much faster if $\mathbf{A}$ can be decomposed – or *factorised* – into the product of two smaller matrices $\mathbf{A} = \mathbf{A}_1\mathbf{A}_2$, where $\mathbf{A}_1 \in \mathbb{R}^{m \times n}$ and $\mathbf{A}_2 \in \mathbb{R}^{n \times k}$, where the smallest possible n for which such a product can be found is also called the *rank* of a matrix. We can thus losslessly factorise a matrix and compute a fast matrix product if the matrix is of low rank, that is $n < \min\{m, k\}$.

3 Approximating Arbitrary Functions

Being able to build a communication channel using sequential decoding and re-encoding – which, as we have discussed above, is equivalent to using a biologically plausible synaptic weight matrix $\mathbf{W} = \mathbf{E}\mathbf{D}$ – does not seem to be particularly exciting. After all, we set out to compute arbitrary transformations $\mathbf{y} = f(\mathbf{x})$. However, we are actually quite close to our goal.

3.1 Computing Linear Function Decoders

As a special case, let's first think about computing arbitrary linear transformations $f(\mathbf{x}) = \mathbf{F}\mathbf{x}$, where $\mathbf{F} \in \mathbb{R}^{d' \times d}$ transforms the d-dimensional value represented by the neural population A into a d'-dimensional value that is going to be represented by the population B.

Again, we can sequentially decode using $\mathbf{D}$, then apply our linear transformation $\mathbf{F}$, and finally re-encode the transformed value for the post-population using $\mathbf{E}$. All these operations being linear, we can just roll the linear transformation $\mathbf{F}$ into the decoder $\mathbf{D}$, giving us a specialised *function decoder* $\mathbf{D}^f = \mathbf{F}\mathbf{D} \in \mathbb{R}^{d' \times n}$. Consequently, we could compute a synaptic weight matrix $\mathbf{W}^f = \mathbf{E}\mathbf{D}^f = \mathbf{E}\mathbf{F}\mathbf{D}$ that performs the desired transformation.

3.2 Computing Nonlinear Function Decoders

We can generalise the above idea of a function decoder $\mathbf{D}^f$ to nonlinear functions. This can be accomplished by slightly adapting the optimization problem we are solving when computing the decoder. Instead of minimising the error $\|\hat{\mathbf{x}} - \mathbf{x}\|$ (where $\hat{\mathbf{x}} = \mathbf{D}\boldsymbol{\alpha}(\mathbf{x})$) over the space of represented values $\mathbb{X}$, we can minimise the error $\|\hat{\mathbf{y}} - f(\mathbf{x})\|$, where $\hat{\mathbf{y}} = \mathbf{D}^f \boldsymbol{\alpha}(\mathbf{x})$:

$$\mathbf{D}^f = \arg\min_{\mathbf{D}^f} \frac{1}{|\mathbb{X}|} \int_{\mathbb{X}} \left\| \mathbf{D}^f \boldsymbol{\alpha}(\mathbf{x}) - f(\mathbf{x}) \right\|^2 \,\mathrm{d}\mathbf{x} \,.$$

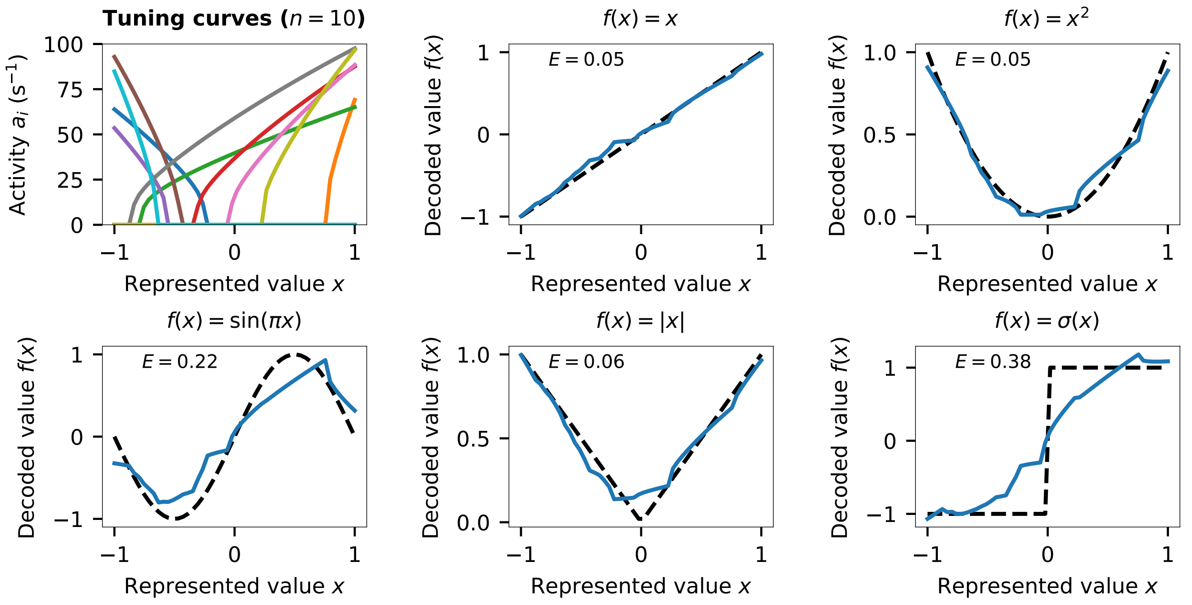
Note that this is essentially the same equation we were looking at when solving for the identity decoder – the only difference being that $\mathbf{x}$ has been replaced by $f(\mathbf{x}) = \mathbf{y}$.

As before, we can discretise the integral by sampling N samples $\mathbf{x}_k \in \mathbb{X}$. Then, the function decoder $\mathbf{D}^f \in \mathbb{R}^{d' \times n}$ is given as

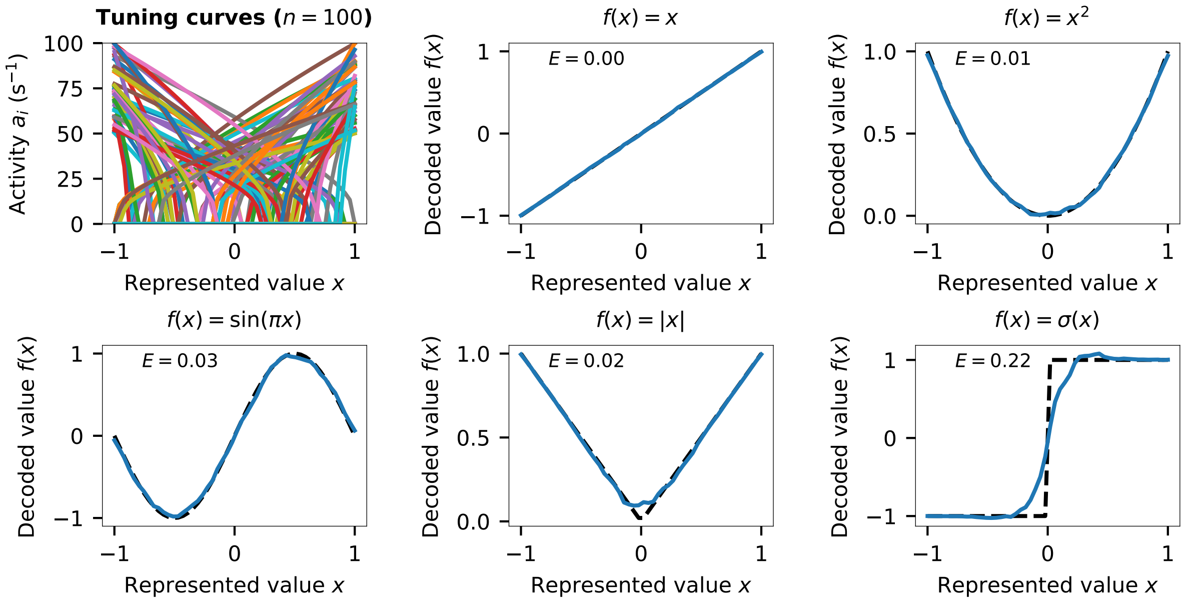
(Function Decoder)

$$\mathbf{D}^f = \left((\mathbf{A}\mathbf{A}^\mathrm{T} + N\sigma^2 \mathbf{I})^{-1} \mathbf{A}\mathbf{Y}^\mathrm{T} \right)^\mathrm{T} \,, \quad \text{where } \left(\mathbf{Y}\right)_{ik} = \left(f(\mathbf{x}_k)\right)_i \,. \tag{3}$$

Again, this is the same decoder computation equation as before. All we did is replacing the matrix $\mathbf{X} \in \mathbb{R}^{d \times N}$ by a matrix $\mathbf{Y} \in \mathbb{R}^{d' \times N}$, where each $\mathbf{y}_k = f(\mathbf{x}_k)$. In case $f(\mathbf{x}) = \mathbf{x}$ (the identity function), this method is *exactly* equivalent to the equation we used to compute the identity decoder $\mathbf{D}$. Figure 5 shows some examples of various functions being decoded from a neural population. The quality of the function approximation depends on the number of pre-neurons, as well as the properties of the function we are trying to compute.

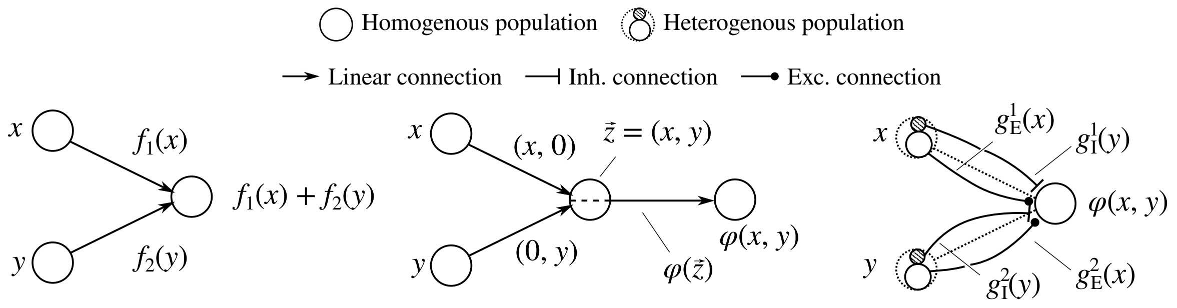


(a) Function decodings for $n = 10$ pre-synaptic neurons



(b) Function decodings for $n = 100$ pre-synaptic neurons

Figure 5: Decoding arbitrary univariate functions f from two neuron populations with different neuron counts n. The decoding error E (depicted is the RMSE) decreases with the number of neurons, yet depends on the kind of function that is being computed. ⌨ *Code*



(a) Linear combination of functions **(b)** Computing nonlinear functions **(c)** Dendritic computation

Figure 6: Computing functions of values represented in multiple pre-populations. In general, if represented values are spread across multiple pre-populations, we can only compute linear combinations of functions of the represented values **(a)**. In order to compute nonlinear functions of multiple values, all values need to be represented in a higher-dimensional neuron population **(b)**. A special case is dendritic computation, where nonlinear effects within the synapses of a neuron are exploited in order to compute nonlinear multivariate functions of values not represented in the same pre-population **(c)**.

Note: *Quality of function decodings.* When decoding a function f from a neural population, we are only approximating that function – the same way we were approximating the identity function when computing the identity decoder. Hence, we will see some error when computing transformations.

As mentioned above, and as visible in fig. 5, the decoding error depends on the number of pre-neurons n – the more neurons, the smaller the error. Of course, when encoding the decoded value in the post-population, we are adding a representational error by representing the decoded value in a population of m neurons.

Furthermore, the error depends on the specific function f we are trying to compute. In general, depending on the shape of the neural tuning curves, we can decode smooth, continuous functions quite well, whereas "jagged", discontinuous functions are not well approximated by decoding from a population of neurons. We will see why that is in a future lecture about analysing representations.

3.3 Multiple Pre-Populations

We can now approximate arbitrary transformations $f(\mathbf{x})$ in the connections between two neuron ensembles. Essentially, we just compute a function decoder $\mathbf{D}^f$ and then perform sequential decoding and re-encoding, which on demand can be turned into a biologically plausible neural network with a synaptic weight matrix $\mathbf{W}^f = \mathbf{E}\mathbf{D}^f$.

The next question would be what happens if we connect multiple pre-populations to the same post-population. In the Neural Engineering Framework, we assume that we just sum the post-synaptic currents evoked by both pre-populations. Hence, the current J_i flowing into the i-th pre-neuron is

$$J_i = \langle \mathbf{e}_i, \mathbf{D}^f \boldsymbol{\alpha}_1 \rangle + \langle \mathbf{e}_i, \mathbf{D}^g \boldsymbol{\alpha}_2 \rangle + J_i^\mathrm{bias},$$

where $\boldsymbol{\alpha}_1$ are the activities of the first pre-population, $\boldsymbol{\alpha}_2$ are the activities of the second pre-population, $\mathbf{D}^f$ is a function decoder decoding f from the first-propulation, and $\mathbf{D}^g$ decodes g from the second pre-population. Note that we can rewrite the above equation as

$$J_i = \langle \mathbf{e}_i, \mathbf{D}^f \boldsymbol{\alpha}_1 + \mathbf{D}^g \boldsymbol{\alpha}_2 \rangle + J_i^\mathrm{bias} \approx \langle \mathbf{e}_i, f(\mathbf{x}_1) + g(\mathbf{x}_2) \rangle + J_i^\mathrm{bias},$$

where $\mathbf{x}_1$ and $\mathbf{x}_2$ are the values represented by the two pre-populations. This means that we are essentially computing the sum of the two functions.

Note that this does not allow us to compute non-linear multivariate functions such as $\varphi(x, y) = xy$, if x and y are represented in separate pre-populations. However, we can compute multiplication by representing x and y in a two-dimensional population representing $\mathbf{z} = (x, y)$, and then decode φ out of this population (fig. 6).

References

[1] Piergiorgio Strata and Robin Harvey. “Dales Principle”. In: *Brain Research Bulletin* 50.5 (1999), pp. 349–350. ISSN: 0361-9230. DOI: `https://doi.org/10.1016/S0361-9230(99)00100-8`. URL: `http://www.sciencedirect.com/science/article/pii/S0361923099001008`.

[2] Arnd Roth and Mark C. W. van Rossum. “Modeling Synapses”. In: *Computational Modeling Methods for Neuroscientists*. Ed. by Erik De Schutter. The MIT Press, 2009, pp. 139–159.